ПРОФЕССИОНАЛЬНЫЙ CMAKE

ПРАКТИЧЕСКОЕ РУКОВОДСТВО

КРЕЙГ СКОТТ

Профессиональный CMake: практическое руководство

Версия: 1.0.0

https://crascit.com

Книга переведена с помощью искусственного лингвистического интеллекта DeepL

deepl.com

*Под редакцией **Дрюнделя Хрюнделя***

Оглавление

Предисловие

Несколько лет назад я был удивлен отсутствием опубликованных материалов для обучения работе с CMake. Официальная справочная документация была полезным ресурсом для тех, кто готов к исследованию, но как способ постепенного, структурированного изучения CMake она не была идеальной. Было несколько вики-сайтов и персональных веб-сайтов с полезным содержанием, но было и много таких, которые содержали устаревшие или сомнительные советы и примеры. Существовал явный пробел, что означало, что новичкам в CMake было трудно научиться хорошим практикам, что приводило к тому, что многие были подавлены или разочарованы.

В то время я писал статьи в блоге, чтобы более продуктивно проводить свободное время и углубить свои технические знания в области разработки программного обеспечения. Я часто писал о тех областях, которые возникали в моем общении с коллегами по работе или в моей собственной деятельности по разработке, и я находил это полезным для себя и других. По мере того, как эта схема повторялась, родилась идея написать книгу. Прошло два с половиной года, и в результате появилась книга, которую вы сейчас читаете.

На этом пути произошел классический поворотный момент, о котором я сейчас вспоминаю с некоторым умилением. Коллега сетовал на одну особенность, которую он хотел бы видеть в CMake. Эта мысль закралась в мой мозг и сидела там несколько месяцев, пока однажды я не решил изучить, насколько сложно будет добавить эту возможность самостоятельно. В результате появилась функция тестовых приспособлений, которая теперь является частью CMake, но, что более важно, я был поражен положительным опытом, который я получил, внося этот вклад. Люди, инструменты и действующие процессы сделали работу над проектом действительно приятной. После этого я стал более глубоко вовлечен в проект и сейчас выполняю роль добровольного со-мейнтейнера.

Благодарности

Только когда вы приходите поблагодарить всех, кто внес свой вклад в процесс выпуска вашей книги, вы понимаете, как много людей принимало в этом участие. Такая работа не может быть выполнена без щедрости, терпения и проницательности других людей, и она не может быть успешной, если ее не подвергать испытаниям, проверять и переделывать. Она зависит от тех, кто был достаточно добр, чтобы (иногда неосознанно!) принять участие в этой деятельности, в той же степени, что и автор, и я не могу выразить благодарность этим людям за их доброту и мудрость.

Сообщество CMake не было бы таким сильным и энергичным, каким оно является сегодня, без постоянной поддержки Kitware и ее сотрудников, как в прошлом, так и в настоящем. Я хотел бы особо отметить Брэда Кинга, руководителя проекта CMake, который своим всеохватывающим и ободряющим подходом к работе с новыми участниками CMake сделал так, что такие люди, как я, стали очень желанными и почувствовали возможность участвовать в проекте. Я лично многому научился у него, просто наблюдая за тем, как он взаимодействует с разработчиками и пользователями, обеспечивая сильное лидерство и создавая атмосферу уважения к другим. Само собой разумеется, что многие участники проекта CMake на протяжении многих лет также заслуживают похвалы за свои усилия, которые часто предпринимались исключительно на добровольной основе. Я смиренно оцениваю масштаб вклада, который был сделан столь многими, и положительное влияние на мир разработки программного обеспечения.

Несколько человек великодушно согласились просмотреть материал этой книги, без которых пострадали бы и техническая точность, и читабельность. Все оставшиеся ошибки и недочеты полностью принадлежат мне. Коллеги-разработчики CMake Грегор Ясный и Кристиан Пфайффер были ценными помощниками на протяжении всего процесса рецензирования, и я искренне благодарен им за их предложения и проницательность. Спасибо также Нильсу Гладицу за его вклад, особенно в столь сжатые сроки. Я также хотел

бы поблагодарить моих прошлых коллег, Мэтта Болгера и Лахлана Хетертона, которые предоставили конструктивную обратную связь и напомнили мне о важности свежего взгляда.

Отдельного упоминания заслуживает мой коллега Майк Уэйк. Большая часть материала в этой книге была отработана и протестирована в реальных, активно разрабатываемых проектах. Были неправильные повороты и множество технических дискуссий о том, как улучшить вещи с точки зрения удобства использования и надежности. Его поддержка в предоставлении пространства и поощрения для проработки этих вопросов, а также его готовность терпеть краткосрочную (а иногда и не очень краткосрочную) боль сыграли важную роль в дистилляции многих процессов и методов до того, что работает на практике. Я также очень благодарен ему за своевременные советы и поддержку, которые он давал в нужный момент в некоторые из наиболее напряженных и изнурительных периодов.

Я также хотел бы выразить свою благодарность людям, стоящим за Asciidoctor, программным обеспечением, использованным для составления и подготовки этой книги. Несмотря на размер, сложность и технический характер материала, я был постоянно поражен тем, как оно сделало самопубликацию не просто жизнеспособным вариантом, но и приятным опытом с удивительно малым количеством практических ограничений. Путь от автора к читателю теперь намного короче и проще, чем всего несколько лет назад, что делает его доступным для большего числа потенциальных писателей и приносит пользу всему обществу. Спасибо за потрясающий инструмент!

Обложка книги и некоторые вспомогательные материалы на сайте - результат более наметанного глаза и понимания графического дизайна, чем мой собственный. Моему другу и дизайнеру Ви, то, как тебе удалось каким-то образом придать смысл моим случайным, несвязным идеям и противоречивым фрагментам, все еще озадачивает меня. Я не понимаю, как ты это делаешь, но мне нравится конечный результат!

В разделе благодарностей каждой книги автор неизменно благодарит членов семьи и супругов, и на это есть веские причины. Требуется огромное понимание и самопожертвование, чтобы терпеть вашу усталость, вашу неспособность делать многие вещи, которые делают обычные люди, и ваше неразумное решение посвятить проекту больше времени, чем им. Я действительно не могу выразить всю глубину моей благодарности моей жене за то, как ей удавалось быть такой поддерживающей и терпеливой на протяжении всего процесса написания и публикации этой книги. Я действительно очень удачливый человек.

Часть I: Основы

Попытка использовать любой инструмент, не разобравшись хотя бы в основах того, что он делает и как его следует использовать, скорее всего, приведет к разочарованию. С другой стороны, если потратить все свое время на изучение теории о чем-то без практического применения, это будет довольно скучным занятием и часто приводит к слишком идеалистическому пониманию. Эта первая часть книги следует логической прогрессии через более фундаментальные возможности и концепции CMake и построена таким образом, чтобы читатель мог сразу же экспериментировать и делать все более полезные вещи с каждой главой. Цель состоит в постепенном наращивании базовых знаний, необходимых для эффективного использования CMake, с акцентом на возможность сразу же применить эти знания на практике.

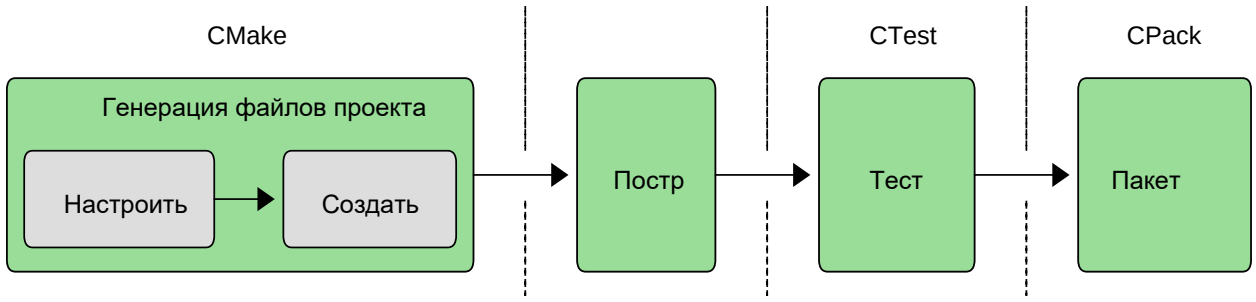
В первых нескольких главах основное внимание уделяется созданию базового исполняемого файла или библиотеки, что достаточно для быстрого знакомства с CMake начинающего разработчика. Последующие главы расширяют эти знания и демонстрируют, как получить максимальную отдачу от того, что может предложить CMake. Представленные методы направлены на использование в реальном мире, с намерением выработать хорошие привычки и научить надежным методам, которые могут применяться в очень больших проектах и справляться с более сложными сценариями.

Все последующие части книги в значительной степени опираются на материал, рассмотренный в этой первой части. Тем, кто уже давно использует CMake, темы могут показаться относительно знакомыми, но материал также включает в себя знания, полученные в ходе реализации реальных проектов и взаимодействия с сообществом CMake. Даже опытным пользователям будет полезно прочитать хотя бы раздел "Рекомендуемая практика" в конце каждой главы.

Глава 1. Введение

Будь то опытный разработчик или только начинающий карьеру программиста, невозможно избежать процесса ознакомления с рядом инструментов, чтобы превратить исходный код проекта в то, что может реально использовать конечный пользователь. Компиляторы, компоновщики, фреймворки для тестирования, системы упаковки и многое другое - все они вносят свой вклад в сложность развертывания высококачественного и надежного программного обеспечения. Хотя некоторые платформы имеют доминирующую среду IDE, которая упрощает некоторые аспекты этой задачи (например, Xcode и Visual Studio), проекты, которые должны поддерживать несколько платформ, не всегда могут использовать их возможности. Поддержка нескольких платформ добавляет дополнительные сложности, которые могут повлиять на все - от набора доступных инструментов до различных доступных возможностей и накладываемых ограничений. Обычного разработчика можно простить за то, что он теряет хотя бы часть рассудка, пытаясь уследить за всей этой картиной.

К счастью, существуют инструменты, которые делают этот процесс более управляемым. CMake - один из таких инструментов, а точнее, CMake - это набор инструментов, который охватывает все, начиная от настройки сборки и заканчивая созданием пакетов, готовых к распространению. Он не только охватывает процесс от начала до конца, но и поддерживает широкий спектр платформ, инструментов и языков. При работе с CMake необходимо понимать его мировоззрение. В общих чертах процесс от начала до конца, согласно CMake, выглядит примерно так:



На первом этапе берется общее описание проекта и генерируются файлы проекта для конкретной платформы, подходящие для использования со штатным инструментом сборки по выбору разработчика (например, make, Xcode, Visual Studio и т.д.). Хотя этот этап настройки - то, за что CMake наиболее известен, в набор инструментов CMake также входят CTest и CPack для управления последующими этапами тестирования и упаковки соответственно. Весь процесс от начала до конца может управляться из самого CMake, а этапы тестирования и упаковки доступны просто как дополнительные цели в сборке. Даже инструмент сборки может быть вызван CMake.

Прежде чем приступить к работе с CMake, разработчикам необходимо убедиться, что CMake установлен в их системе. Некоторые платформы обычно поставляются с уже установленным CMake (например, в большинстве дистрибутивов Linux CMake доступен через менеджер пакетов), но эти версии часто довольно старые. По возможности рекомендуется, чтобы разработчики работали с последним выпуском CMake. Это особенно актуально при разработке для платформ Apple, где такие инструменты, как Xcode и его SDK, быстро меняются, а требования магазинов приложений меняются со временем. Официальные пакеты CMake могут быть загружены и распакованы в каталог на машине разработчика без вмешательства в общесистемную установку CMake. Разработчикам рекомендуется воспользоваться этим преимуществом и оставаться относительно близкими к последнему стабильному выпуску CMake.

В наши дни CMake также поставляется с довольно обширной справочной документацией, доступной на официальном сайте CMake. Этот полезный ресурс очень удобен для поиска различных команд, опций, ключевых слов и т.д., и разработчики, вероятно, захотят добавить его в закладки для быстрого обращения к

нему. Список рассылки пользователей CMake также является отличным источником советов и рекомендуемым форумом, где можно задать вопросы, связанные с CMake, если документация не дает достаточного руководства.

Глава 2. Настройка проекта

Без системы сборки проект - это просто набор файлов. CMake вносит в это некоторый порядок, начиная с человекочитаемого файла CMakeLists.txt, который определяет, что и как должно быть собрано, какие тесты необходимо выполнить и какой пакет(ы) создать. Этот файл представляет собой независимое от платформы описание всего проекта, которое CMake затем превращает в файлы проектов инструментов сборки, специфичных для платформы. Как следует из названия, это обычный текстовый файл, который разработчики редактируют в своем любимом текстовом редакторе или среде разработки. Содержимое этого файла подробно рассматривается в последующих главах, а пока достаточно знать, что именно он управляет всем, что CMake будет делать при настройке и выполнении сборки.

Основополагающей частью CMake является концепция, согласно которой проект имеет исходный каталог и бинарный каталог. Исходный каталог - это место, где находится файл CMakeLists.txt, в котором организованы исходные файлы проекта и все остальные файлы, необходимые для сборки. Исходный каталог часто находится под контролем версий с помощью таких инструментов, как git, subversion или аналогичных.

Каталог бинарных файлов - это место, где создается все, что производится в процессе сборки. Часто его также называют каталогом сборки. По причинам, которые станут понятны в последующих главах, в CMake обычно используется термин *бинарный* каталог, но среди разработчиков более распространен термин каталог сборки. В этой книге предпочтение отдается последнему термину, поскольку он, как правило, более интуитивно понятен. CMake, выбранный инструмент сборки (например, make, Visual Studio и т.д.), CTest и CPack создают различные файлы в каталоге сборки и подкаталогах под ним. Исполняемые файлы, библиотеки, результаты тестирования и пакеты создаются в каталоге сборки. CMake также создает специальный файл CMakeCache.txt в каталоге сборки для хранения различной информации для повторного использования при последующих запусках. Обычно разработчикам не нужно беспокоиться о файле CMakeCache.txt, но в последующих главах будут рассмотрены ситуации, когда этот файл может быть полезен. Файлы проектов инструмента сборки (например, файлы проектов Xcode или Visual Studio, Makefiles и т.д.) также создаются в каталоге сборки и не предназначены для контроля версий. Файлы CMakeLists.txt являются каноническим описанием проекта, и сгенерированные файлы проекта следует считать частью выходных данных сборки.

Когда разработчик начинает работу над проектом, он должен решить, где будет находиться его каталог сборки по отношению к каталогу исходных текстов. Существует два подхода: сборка *в исходном коде* и сборка *вне исходного кода*.

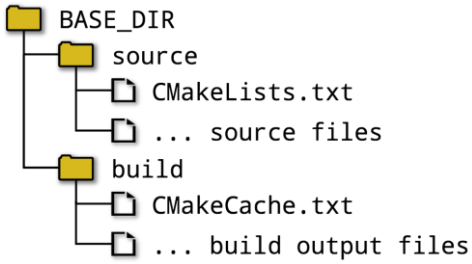
2.1. Сборки из исходного кода

Можно, хотя и не рекомендуется, чтобы исходный код и каталог сборки были одинаковыми. Такое расположение называется сборкой *в исходном коде*. Разработчики в начале своей карьеры часто начинают использовать этот подход из-за кажущейся простоты. Однако основная трудность сборки in-source заключается в том, что все результаты сборки смешиваются с исходными файлами. Отсутствие разделения приводит к загромождению каталогов всевозможными файлами и подкаталогами, что усложняет управление исходными текстами проекта и создает риск того, что результаты сборки будут перезаписывать исходные файлы. Это также усложняет работу с системами контроля версий, поскольку при сборке создается множество файлов, которые либо инструмент контроля исходных текстов должен знать, что их нужно игнорировать, либо разработчик должен вручную исключать их при коммитах. Еще один недостаток сборки в исходниках заключается в том, что очистка всех результатов сборки и начало работы с чистым деревом исходников может оказаться нетривиальной задачей. По этим причинам разработчикам не рекомендуется использовать сборки из исходных текстов, когда это возможно, даже для простых проектов.

2.2. Сборки вне источника

Более предпочтительным является расположение исходных текстов и каталогов сборки в разных каталогах, что называется сборкой *вне исходных текстов*. При этом исходные тексты и результаты сборки полностью отделены друг от друга, что позволяет избежать проблем смешивания, возникающих при сборке из исходных текстов. Преимущество сборок out-ofsource в том, что разработчик может создать несколько каталогов сборки для одного и того же исходного каталога, что позволяет настраивать сборки с различными наборами опций, например, отладочные и релизные версии и т.д.

В этой книге всегда будут использоваться сборки из исходных текстов и будет соблюдаться схема, при которой исходный текст и каталог сборки находятся под общим родительским каталогом. Каталог сборки будет называться *build* или как-то иначе. Например:



Вариация этого метода, используемая некоторыми разработчиками, заключается в том, чтобы сделать каталог сборки подкаталогом каталога исходного кода. Это дает большинство преимуществ сборки вне исходного кода, но при этом имеет некоторые недостатки, присущие сборке внутри исходного кода. Если нет веских причин для такой структуры, рекомендуется держать каталог сборки полностью вне дерева исходных текстов.

2.3. Генерация файлов проекта

После выбора структуры каталогов разработчик запускает программу CMake, которая считывает файл CMakeLists.txt и создает файлы проекта в каталоге сборки. Разработчик выбирает тип создаваемого файла проекта, выбирая определенный *генератор* файла проекта. Поддерживается целый ряд различных генераторов, наиболее часто используемые из них перечислены в таблице ниже.

Категория	Примеры генераторов	Мультиконфигурация
Visual Studio	Visual Studio 15 2017	Да
	Visual Studio 14 2015	
	⋮	
Xcode	Xcode	Да
Ниндзя	Ниндзя	Нет
Makefiles	Unix Makefiles	Нет
	MSYS Makefiles	
	MinGW Makefiles	
	NMake Makefiles	

Некоторые генераторы создают проекты, поддерживающие несколько конфигураций (например, Debug, Release и т.д.). Это позволяет разработчику выбирать между различными конфигурациями сборки без

необходимости повторного запуска CMake, что больше подходит для генераторов, создающих проекты для использования в средах IDE, таких как Xcode и Visual Studio. Для генераторов, не поддерживающих несколько конфигураций, разработчику приходится повторно запускать CMake, чтобы переключить сборку между Debug, Release и т. д. Это проще и часто имеет хорошую поддержку в средах IDE, не так тесно связанных с конкретным компилятором (Qt Creator, KDevelop и т.д.).

Самый простой способ запустить CMake - это утилита командной строки cmake. Самый простой способ вызвать ее - перейти в каталог сборки и передать cmake параметры для типа генератора и расположения дерева исходных текстов. Например:

```
mkdir build
cd build
cmake -G "Unix Makefiles" ../source
```

Если опция -G опущена, CMake выберет тип генератора по умолчанию, основываясь на платформе хоста. Для всех типов генераторов CMake выполнит ряд тестов и опросит систему, чтобы определить, как настроить файлы проекта. Это включает в себя проверку работоспособности компиляторов, определение набора поддерживаемых функций компилятора и различные другие задачи. Перед завершением работы CMake в журнал будет занесена различная информация, а в случае успеха появятся следующие строки:

```
-- Configuring done
-- Generating done
-- Build files have been written to: /some/path/build
```

Вышеизложенное подчеркивает, что создание файла проекта фактически включает два этапа: конфигурирование и генерацию. На этапе *конфигурирования* CMake считывает файл CMakeLists.txt и создает внутреннее представление всего проекта. После этого на этапе *генерации* создаются файлы проекта. Различие между конфигурированием и генерацией не имеет большого значения для базового использования CMake, но в последующих главах разделение конфигурирования и генерации становится важным. Более подробно это рассматривается в главе 10, *Выражения генератора*.

После завершения работы CMake сохраняет файл CMakeCache.txt в каталоге сборки. CMake использует этот файл для сохранения деталей, чтобы при повторном запуске повторно использовать информацию, вычисленную в первый раз, и ускорить создание проекта. Как будет описано в последующих главах, он также позволяет сохранять опции разработчика между запусками. Приложение с графическим интерфейсом, cmake-gui, доступно в качестве альтернативы запуску инструмента командной строки cmake, но знакомство с ним отложено до главы 5, *Переменные,* где его полезность станет более очевидной.

2.4. Запуск инструмента сборки

На этом этапе, когда файлы проекта уже доступны, разработчик может использовать выбранный инструмент сборки привычным для него способом. Каталог сборки будет содержать необходимые файлы проекта, которые могут быть загружены в IDE, прочитаны инструментами командной строки и т.д. В качестве альтернативы cmake может вызывать инструмент сборки от имени разработчика следующим образом:

```
cmake --build /some/path/build --config Debug --target MyApp
```

Это работает даже для типов проектов, которые разработчик, возможно, привык использовать в IDE, таких как Xcode или Visual Studio. Опция --build указывает на каталог сборки, используемый на этапе генерации проекта CMake. Для генераторов с несколькими конфигурациями опция --config указывает, какую конфигурацию собирать, в то время как генераторы с одной конфигурацией игнорируют опцию --config и полагаются на информацию, предоставленную на этапе генерации проекта CMake. Указание конфигурации сборки подробно рассматривается в главе 13, *Тип сборки*. Опция --target может быть использована для указания инструменту сборки, что именно нужно собрать, или, если она опущена, будет собрана цель по умолчанию.

Хотя разработчики обычно вызывают выбранный инструмент сборки непосредственно в повседневной разработке, вызов его через команду cmake, как показано выше, может быть более полезен в сценариях, управляющих автоматизированной сборкой. При использовании этого подхода простая сборка по сценарию может выглядеть следующим образом:

```
mkdir build
cd build
cmake -G "Unix Makefiles" ../source
cmake --build . --config Release --target MyApp
```

Если разработчик хочет поэкспериментировать с различными генераторами, достаточно изменить аргумент, задаваемый опцией -G CMake, и нужный инструмент сборки будет вызван автоматически. Для работы cmake --build инструмент сборки даже не обязательно должен быть в PATH пользователя (хотя он может потребоваться для начального этапа настройки при первом вызове cmake).

2.5. Рекомендуемые практики

Даже если вы только начинаете использовать CMake, рекомендуется взять за правило держать каталог сборки полностью отделенным от дерева исходных текстов. Хороший способ оценить преимущества такой схемы - установить две или более различных сборок для одного и того же каталога исходных текстов. Одна сборка может быть настроена на Debug, другая - на Release. Другой вариант - использовать разные генераторы проектов для разных директорий сборки, например, Unix Makefiles и Xcode. Это поможет выявить непреднамеренные зависимости от конкретного инструмента сборки или проверить различия в настройках компилятора для разных типов генераторов.

Может быть заманчиво сосредоточиться на использовании одного конкретного типа генератора проекта на ранних стадиях проекта, особенно если разработчик не привык писать кроссплатформенное программное обеспечение. Однако проекты имеют привычку выходить за рамки своих первоначальных рамок, и довольно часто им может потребоваться поддержка дополнительных платформ и, следовательно, различных типов генераторов. Периодическая проверка сборки с генератором проекта, отличным от того, который обычно использует разработчик, может сэкономить значительные средства в будущем, предотвращая появление специфичного для генератора кода там, где он не требуется. Это также поможет проекту получить преимущества любого нового типа генератора в будущем. Хорошей стратегией будет обеспечить сборку проекта с генератором по умолчанию на каждой интересующей платформе, плюс еще один тип. Генератор Ninja - отличный выбор для последнего, поскольку он имеет самую широкую поддержку платформ среди всех генераторов, а также создает очень эффективные сборки. Если проект создается скриптом, вызывайте инструмент сборки через cmake --build вместо того, чтобы вызывать инструмент сборки напрямую. Это позволит скрипту легко переключаться между типами генераторов без необходимости внесения изменений.

Глава 3. Минимальный проект

Все проекты CMake начинаются с файла под названием CMakeLists.txt, который должен быть помещен в вершину дерева исходных текстов. Считайте, что это файл проекта CMake, определяющий все, что касается сборки, от исходных текстов и целей до тестирования, упаковки и других пользовательских задач. Он может быть простым, состоять из нескольких строк, а может быть довольно сложным и содержать больше файлов из других каталогов. CMakeLists.txt - это обычный текстовый файл, который обычно редактируется напрямую, как и любой другой исходный файл проекта.

Продолжая аналогию с исходными текстами, CMake определяет свой собственный язык, в котором есть много знакомых программисту вещей, таких как переменные, функции, макросы, условная логика, циклы, комментарии к коду и так далее. Эти различные концепции и возможности будут рассмотрены в следующих нескольких главах, а пока цель состоит в том, чтобы получить простую сборку в качестве отправной точки. Ниже приведен минимальный, хорошо сформированный файл CMakeLists.txt, создающий базовый исполняемый файл.

```
cmake_minimum_required(VERSION 3.2)
project(MyApp)
add_executable(myExe main.cpp)
```

Каждая строка в приведенном выше примере выполняет встроенную *команду* CMake. В CMake команды похожи на вызовы функций в других языках, за исключением того, что, хотя они поддерживают аргументы, они не возвращают значения напрямую (но в следующей главе будет показано, как передавать значения обратно вызывающей стороне другими способами). Аргументы отделяются друг от друга пробелами и могут быть разделены на несколько строк:

```
add_executable(myExe
    main.cpp
    src1.cpp
    src2.cpp
)
```

Имена команд также не чувствительны к регистру, поэтому все следующие команды эквивалентны:

```
add_executable(myExe main.cpp)
ADD_EXECUTABLE(myExe main.cpp)
Add_Executable(myExe main.cpp)
```

Типичный стиль может быть разным, но в настоящее время более распространенным является использование строчных букв в именах команд (этого же правила придерживается документация CMake для встроенных команд).

3.1. Управление версиями CMake

CMake постоянно обновляется и расширяется для добавления поддержки новых инструментов, платформ и возможностей. Разработчики CMake тщательно следят за тем, чтобы поддерживать обратную совместимость с каждым новым выпуском, поэтому, когда пользователи обновляют CMake до новой версии, проекты должны продолжать собираться так же, как и раньше. Иногда требуется изменить определенное поведение CMake или ввести более строгие проверки и предупреждения в новых версиях. Вместо того чтобы требовать от всех

проектов немедленно решать эту проблему, CMake предоставляет механизмы *политики*, которые позволяют проекту сказать: "Ведите себя как CMake версии X.Y.Z". Это позволяет CMake исправлять ошибки внутри проекта и вводить новые возможности, но при этом сохранять ожидаемое поведение любого конкретного прошлого выпуска.

Основной способ, которым проект определяет детали ожидаемого поведения версии CMake, - это команда cmake_minimum_required(). Она должна быть первой строкой файла CMakeLists.txt, чтобы требования проекта были проверены и установлены прежде, чем что-либо еще. Эта команда делает две вещи:

- В нем указывается минимальная версия CMake, необходимая проекту. Если файл CMakeLists.txt будет обработан с версией CMake старше указанной, проект будет немедленно остановлен с ошибкой. Это гарантирует, что определенный минимальный набор функциональных возможностей CMake доступен перед началом работы.
- Он применяет настройки политики для соответствия поведения CMake указанной версии.

Использование этой команды настолько важно, что CMake выдаст предупреждение, если в файле CMakeLists.txt перед любой другой командой не будет вызвана cmake_minimum_required(). Ему необходимо знать, как задать поведение политики для всех последующих обработок. Для большинства проектов достаточно рассматривать cmake_minimum_required() как простое указание минимально необходимой версии CMake, как следует из ее названия. Тот факт, что она также подразумевает, что CMake должен вести себя так же, как эта конкретная версия, можно считать полезным побочным преимуществом. В главе 12, *Политика,* более подробно рассматриваются настройки политики и объясняется, как изменить это поведение при необходимости.

Типичная форма команды cmake_minimum_required() проста:

```
cmake_minimum_required(VERSION major.minor[.patch[.tweak]])
```

Ключевое слово VERSION должно присутствовать всегда, а предоставляемые сведения о версии должны содержать как минимум часть major.minor. В большинстве проектов нет необходимости указывать части patch и tweak, поскольку новые возможности обычно появляются только в минорных обновлениях версий (это официальное поведение CMake, начиная с версии 3.0). Только если требуется исправление конкретной ошибки, в проекте следует указывать часть patch. Более того, поскольку ни в одном выпуске CMake серии 3.x не использовался номер tweak, проектам также не нужно его указывать.

Разработчикам следует хорошо подумать о том, какая минимальная версия CMake должна быть необходима их проекту. Версия 3.2, возможно, является самой старой из всех, которые следует рассматривать для нового проекта, поскольку она предоставляет достаточно полный набор функций для современных методов CMake. Версия 2.8.12 имеет меньший набор функций, в ней отсутствует ряд полезных возможностей, но она может быть пригодна для старых проектов. В версиях до этой версии отсутствуют существенные возможности, которые делают невозможным использование многих современных методов CMake. При работе с быстро развивающимися платформами, такими как iOS, могут потребоваться совсем свежие версии CMake для поддержки последних релизов ОС и т.д.

В качестве общего правила следует выбирать самую последнюю версию CMake, которая не создаст значительных проблем для тех, кто собирает проект. Наибольшие трудности обычно испытывают проекты, которым необходимо поддерживать старые платформы, где поставляемая системой версия CMake может быть довольно старой. В таких случаях, если это вообще возможно, разработчикам следует рассмотреть возможность установки более свежей версии, а не ограничиваться очень старыми версиями CMake. С другой стороны, если проект сам будет зависимым для других проектов, то выбор более свежей версии CMake может стать препятствием для внедрения. В таких случаях может быть полезно вместо этого требовать самую старую

версию CMake, которая все еще обеспечивает необходимый минимум функций CMake, но при этом использовать возможности более поздних версий CMake, если они доступны (в главе 12, *Политика,* представлены методы достижения этой цели). Это предотвратит принуждение других проектов к требованию более свежей версии, чем обычно позволяет или предоставляет их целевая среда. Зависимые проекты всегда могут потребовать более новую версию, если захотят, но они не могут потребовать более старую версию. Основным недостатком использования самой старой работоспособной версии является то, что это может привести к большему количеству предупреждений об устаревании, поскольку новые версии CMake будут предупреждать о более старом поведении, чтобы побудить проекты обновить себя.

3.2. Команда project()

Каждый проект CMake должен содержать команду project(), которая появляется после вызова cmake_minimum_required(). Команда с наиболее распространенными опциями имеет следующий вид:

```
project(projectName
        [VERSION major[.minor[.patch[.tweak]]]]
        [LANGUAGES languageName ...]
)
```

Имя проекта (projectName) является обязательным и может содержать только буквы, цифры, знаки подчеркивания (_) и дефисы (-), хотя на практике обычно используются только буквы и, возможно, знаки подчеркивания. Поскольку пробелы не допускаются, имя проекта не обязательно окружать кавычками. Это имя используется для верхнего уровня проекта в некоторых генераторах проектов (например, Xcode и Visual Studio), а также используется в различных других частях проекта, например, в качестве значения по умолчанию для метаданных упаковки и документации, для предоставления специфических для проекта переменных и так далее. Имя является единственным обязательным аргументом для команды project().

Необязательные данные VERSION поддерживаются только в CMake 3.0 и более поздних версиях. Как и projectName, сведения о версии используются CMake для заполнения некоторых переменных и в качестве метаданных пакета по умолчанию, но кроме этого, сведения о версии не имеют никакого другого значения. Тем не менее, хорошей привычкой является определение версии проекта здесь, чтобы другие части проекта могли ссылаться на нее. В главе 19 "*Указание сведений о версии"* это подробно рассматривается и объясняется, как ссылаться на эту информацию о версии позже в файле CMakeLists.txt.

Необязательный аргумент LANGUAGES определяет языки программирования, которые должны быть включены для проекта. Поддерживаются следующие значения: C, CXX, Fortran, ASM, Java и другие. При указании нескольких языков разделяйте каждый пробелом. В некоторых особых ситуациях проекты могут захотеть указать, что никакие языки не используются, что можно сделать с помощью LANGUAGES NONE. Методы, представленные в последующих главах, используют преимущества этой формы. Если опция LANGUAGES не указана, CMake по умолчанию использует языки C и CXX. Версии CMake до 3.0 не поддерживают ключевое слово LANGUAGES, но языки все еще можно указать после имени проекта, используя старую форму команды следующим образом:

```
project(myProj C CXX)
```

Новым проектам рекомендуется указать минимальную версию CMake не ниже 3.0 и использовать вместо нее новую форму с ключевым словом LANGUAGES.

Команда project() делает гораздо больше, чем просто заполнение нескольких переменных. Одна из ее важных обязанностей - проверка компиляторов для каждого включенного языка и обеспечение их успешной

компиляции и компоновки. Проблемы с настройкой компилятора и компоновщика могут быть обнаружены на самом раннем этапе. После прохождения этих проверок CMake устанавливает ряд переменных и свойств, которые управляют сборкой для включенных языков. Если в файле CMakeLists.txt не вызывается функция project() или вызывается недостаточно рано, CMake неявно вызовет ее внутренне для языков по умолчанию C и CXX, чтобы обеспечить правильную настройку компиляторов и компоновщиков для других команд, которые на них полагаются. В последующих главах дается более подробная информация о настройке цепочки инструментов и демонстрируется, как запрашивать и изменять такие вещи, как флаги компилятора, расположение компилятора и т.д.

Когда проверки компилятора и компоновщика, выполняемые CMake, завершаются успешно, их результаты кэшируются, чтобы не повторять их при последующих запусках CMake. Эти кэшированные данные хранятся в каталоге сборки в файле CMakeCache.txt. Дополнительные сведения о проверках можно найти в подкаталогах в области сборки, но разработчикам обычно нужно заглядывать туда только при работе с новым или необычным компилятором или при настройке файлов цепочки инструментов для кросс-компиляции.

3.3. Создание базового исполняемого файла

Чтобы завершить наш минимальный пример, команда add_executable() указывает CMake на создание исполняемого файла из набора исходных файлов. Основная форма этой команды такова:

```
add_executable(targetName source1 [source2 ...])
```

В результате создается исполняемый файл, на который можно ссылаться в проекте CMake как на targetName. Это имя может содержать буквы, цифры, подчеркивания и дефисы. Когда проект будет собран, в каталоге сборки будет создан исполняемый файл с именем, зависящим от платформы, по умолчанию имя основано на имени цели. Рассмотрим следующий простой пример команды:

```
add_executable(myApp main.cpp)
```

По умолчанию имя исполняемого файла будет myApp.exe в Windows и myApp на платформах на базе Unix, таких как macOS, Linux и т. д. Имя исполняемого файла можно настроить с помощью свойств цели - функции CMake, представленной в главе 9 "*Свойства*". В одном файле CMakeLists.txt можно также определить несколько исполняемых файлов, вызывая функцию add_executable() несколько раз с разными именами целей. Если одно и то же имя цели используется более чем в одной команде add_executable(), CMake выдаст ошибку и выделит ее.

3.4. Комментирование

Прежде чем покинуть эту главу, будет полезно продемонстрировать, как добавлять комментарии в файл CMakeLists.txt. Комментарии широко используются во всей этой книге, и разработчикам рекомендуется взять за привычку комментировать свои проекты так же, как и обычный исходный код. CMake следует тем же соглашениям о комментировании, что и сценарии оболочки Unix. Любая строка, начинающаяся с символа #, рассматривается как комментарий. За исключением строк, заключенных в кавычки, все, что находится после символа # в строке файла CMakeLists.txt, также рассматривается как комментарий. Ниже показано несколько примеров комментариев и объединены понятия, представленные в этой главе:

```
cmake_minimum_required(VERSION 3.2)

# We don't use the C++ compiler, so don't let project()
# test for it in case the platform doesn't have one
project(MyApp VERSION 4.7.2 LANGUAGES C)

# Primary tool for this project
add_executable(mainTool
    main.c
    debug.c   # Optimized away for release builds
)

# Helpful diagnostic tool for development and testing
add_executable(testTool testTool.c)
```

3.5. Рекомендуемые практики

Убедитесь, что каждый проект CMake имеет команду cmake_minimum_required() в качестве первой строки файла CMakeLists.txt верхнего уровня. При выборе минимально необходимого номера версии следует помнить, что чем позже версия, тем больше возможностей CMake сможет использовать проект. Это также означает, что проект будет лучше адаптирован к новым релизам платформы или операционной системы, которые неизбежно вносят новые элементы, с которыми приходится иметь дело системам сборки. И наоборот, если создается проект, предназначенный для сборки и распространения как часть самой операционной системы (что характерно для Linux), минимальная версия CMake, скорее всего, будет диктоваться версией CMake, предоставляемой этим дистрибутивом.

Если для проекта может потребоваться CMake 3.0 или более поздняя версия, также полезно начать думать о номерах версий проекта как можно раньше и как можно скорее включить нумерацию версий в команду project(). Бывает очень трудно преодолеть инерцию существующих процессов и изменить способ работы с номерами версий на более поздних этапах жизни проекта. При выборе стратегии версионирования учитывайте такие популярные практики, как семантическое версионирование.

Глава 4. Построение простых целей

Как было показано в предыдущей главе, определить простой исполняемый файл в CMake относительно просто. Простой пример, приведенный ранее, требовал определения целевого имени исполняемого файла и перечисления исходных файлов для компиляции:

```
add_executable(myApp main.cpp)
```

Здесь предполагается, что разработчик хочет собрать базовый консольный исполняемый файл, но CMake также позволяет разработчику определять другие типы исполняемых файлов, такие как пакеты приложений на платформах Apple и GUI-приложения для Windows. В этой главе обсуждаются дополнительные опции, которые можно передать функции add_executable() для уточнения этих деталей.

Помимо исполняемых файлов, разработчикам также часто приходится собирать и компоновать библиотеки. CMake поддерживает несколько различных видов библиотек, включая статические, разделяемые, модули и фреймворки. CMake также предлагает очень мощные возможности для управления зависимостями между целями и способами компоновки библиотек. Вся эта область библиотек и работы с ними в CMake составляет основную часть данной главы. Концепции, рассмотренные здесь, будут широко использоваться во всей оставшейся части этой книги. Также приведены некоторые базовые примеры использования переменных и свойств, чтобы дать представление о том, как эти возможности CMake связаны с библиотеками и целями в целом.

4.1. Исполняемые файлы

Более полная форма основной команды add_executable() выглядит следующим образом:

```
add_executable(targetName [WIN32] [MACOSX_BUNDLE]
               [EXCLUDE_FROM_ALL]
               source1 [source2 ...]
)
```

Единственным отличием от формы, показанной ранее, являются новые необязательные ключевые слова.

WIN32

При сборке исполняемого файла на платформе Windows этот параметр указывает CMake на сборку исполняемого файла как приложения с графическим интерфейсом Windows. На практике это означает, что он будет создан с интерфейсом
точку входа WinMain() вместо просто main(), и она будет связана с опцией /SUBSYSTEM:WINDOWS. На всех остальных платформах опция WIN32 игнорируется.

MACOSX_BUNDLE

Когда эта опция присутствует, она направляет CMake на создание пакета приложений при сборке на платформе Apple. Вопреки тому, что следует из названия опции, она применима не только к macOS, но и к другим платформам Apple, таким как iOS. Точное действие этой опции несколько отличается для разных платформ. Например, на macOS компоновка пакета приложений имеет очень специфическую структуру каталогов, в то время как на iOS структура каталогов более плоская. CMake также будет генерировать базовый файл Info.plist для пакетов. Эти и другие детали более подробно рассматриваются в разделе 22.2, "Пакеты приложений". На платформах, отличных от Apple, ключевое слово MACOSX_BUNDLE игнорируется.

EXCLUDE_FROM_ALL

Иногда в проекте определяется несколько целей, но по умолчанию только некоторые из них должны быть собраны. Если цель не указана во время сборки, то по умолчанию собирается цель ALL (в зависимости от используемого генератора CMake, название может быть немного другим, например ALL_BUILD для Xcode). Если исполняемый файл определен с опцией EXCLUDE_FROM_ALL, он не будет включен в цель по умолчанию ALL. Исполняемый файл будет собран, только если он явно запрашивается командой сборки или если он является зависимостью для другой цели, которая является частью сборки ALL по умолчанию. Обычная ситуация, когда может быть полезно исключить цель из ALL - это когда исполняемый файл является инструментом разработчика, который нужен только время от времени.

Помимо вышеперечисленных, существуют и другие формы команды add_executable(), которые создают своего рода ссылку на существующий исполняемый файл или цель, а не определяют новый для сборки. Эти псевдоисполняемые файлы подробно рассматриваются в главе 16, *Типы целей*.

4.2. Определение библиотек

Создание простых исполняемых файлов является фундаментальной потребностью любой системы сборки. Для многих крупных проектов возможность создания библиотек и работы с ними также необходима для поддержания управляемости проекта. CMake поддерживает создание множества различных видов библиотек, заботясь о многих различиях платформ, но при этом поддерживая собственные особенности каждой из них. Цели библиотек определяются с помощью команды add_library(), которая имеет несколько разновидностей. Самая основная из них следующая:

```
add_library(targetName [STATIC | SHARED | MODULE]
            [EXCLUDE_FROM_ALL]
            source1 [source2 ...]
)
```

Эта форма аналогична тому, как add_executable() используется для определения простого исполняемого файла. Имя targetName используется в файле CMakeLists.txt для ссылки на библиотеку, причем по умолчанию из этого имени выводится имя собранной библиотеки в файловой системе. Ключевое слово EXCLUDE_FROM_ALL имеет точно такой же эффект, как и для add_executable(), а именно предотвращает включение библиотеки в цель ALL по умолчанию. Тип собираемой библиотеки задается одним из оставшихся трех ключевых слов STATIC, SHARED или MODULE.

STATIC

Указывает статическую библиотеку или архив. В Windows имя библиотеки по умолчанию будет targetName.lib, а на Unix-подобных платформах, как правило, libtargetName.a.

SHARED

Указывает разделяемую или динамически подключаемую библиотеку. В Windows имя библиотеки по умолчанию будет targetName.dll, на платформах Apple - libtargetName.dylib, а на других Unix-подобных платформах обычно libtargetName.so. На платформах Apple разделяемые библиотеки также могут быть помечены как фреймворки, эта тема рассматривается в разделе 22.3, "Фреймворки".

MODULE

Определяет библиотеку, которая в некотором роде похожа на разделяемую библиотеку, но предназначена для динамической загрузки во время выполнения, а не для непосредственного связывания с библиотекой или исполняемым файлом. Обычно это подключаемые модули или дополнительные компоненты, которые

пользователь может выбирать, загружать их или нет. На платформах Windows для DLL не создается библиотека импорта.

Можно не указывать ключевое слово, определяющее тип библиотеки для сборки. Если проект не требует конкретного типа библиотеки, предпочтительнее не указывать его и оставить выбор за разработчиком при сборке проекта. В таких случаях библиотека будет либо STATIC, либо SHARED, причем выбор определяется значением переменной CMake под названием BUILD_SHARED_LIBS. Если BUILD_SHARED_LIBS имеет значение true, то целью библиотеки будет разделяемая библиотека, в противном случае она будет статической. Работа с переменными подробно рассматривается в главе 5, *Переменные*, а пока один из способов установить эту переменную - включить опцию -D в командную строку cmake, как показано ниже:

```
cmake -DBUILD_SHARED_LIBS=YES /path/to/source
```

Вместо этого его можно задать в файле CMakeLists.txt, поместив перед любой командой add_library() следующую команду, но в этом случае разработчикам придется изменять этот файл, если они хотят его изменить (т.е. он будет менее гибким):

```
set(BUILD_SHARED_LIBS YES)
```

Как и для исполняемых файлов, цели библиотеки могут быть определены так, чтобы ссылаться на какой-либо существующий двоичный файл или цель, а не собираться проектом. Также поддерживается другой тип псевдобиблиотек, позволяющий собирать вместе объектные файлы, не прибегая к созданию статической библиотеки. Все это подробно рассматривается в главе 16, *Типы целей*.

4.3. Связывание целей

При рассмотрении целей, составляющих проект, разработчики обычно привыкли думать в терминах: библиотека A нуждается в библиотеке B, поэтому A связана с B. Это традиционный способ рассмотрения работы с библиотеками, когда идея о том, что одна библиотека нуждается в другой, очень упрощена. Однако в действительности между библиотеками может существовать несколько различных типов зависимостей:

PRIVATE

Частные зависимости указывают, что библиотека A использует библиотеку B в своей собственной внутренней реализации. Все остальное, что ссылается на библиотеку A, не обязано знать о B, потому что она является деталью внутренней реализации A.

PUBLIC

Публичные зависимости указывают, что библиотека A не только использует библиотеку B внутри, но и использует B в своем интерфейсе. Это означает, что A нельзя использовать без B, поэтому все, что использует A, также будет иметь прямую зависимость от B. Примером может быть функция, определенная в библиотеке A, которая имеет по крайней мере один параметр типа, определенного и реализованного в библиотеке B, поэтому код не может вызвать функцию из A без предоставления параметра, тип которого получен из B.

INTERFACE

Интерфейсные зависимости указывают, что для использования библиотеки A необходимо также использовать часть библиотеки B. Это отличается от публичной зависимости тем, что библиотека A не требует B внутри, она использует B только в своем интерфейсе. Примером того, как это может быть полезно, является работа с целевыми библиотеками, определенными с помощью INTERFACE-формы add_library(), например, при

использовании целевой библиотеки для представления зависимостей библиотеки только для заголовков (см. главу 16, *Типы целевых библиотек*).

CMake отражает этот более богатый набор отношений зависимости с помощью команды target_link_libraries(), а не только упрощенную идею необходимости связывания. В общем виде команда выглядит следующим образом:

```
target_link_libraries(targetName
     <PRIVATE|PUBLIC|INTERFACE> item1 [item2 ...]
    [<PRIVATE|PUBLIC|INTERFACE> item3 [item4 ...]]
    ...
)
```

Это позволяет проектам точно определить, как одна библиотека зависит от других. Затем CMake позаботится об управлении зависимостями во всей цепочке библиотек, связанных таким образом. Например, рассмотрим следующее:

```
add_library(collector src1.cpp)
add_library(algo src2.cpp)
add_library(engine src3.cpp)
add_library(ui src4.cpp)
add_executable(myApp main.cpp)

target_link_libraries(collector
    PUBLIC  ui
    PRIVATE algo engine
)
target_link_libraries(myApp PRIVATE collector)
```

В этом примере библиотека ui связана с библиотекой collector как PUBLIC, поэтому даже если myApp напрямую связан только с collector, myApp также будет связан с ui из-за этой связи PUBLIC. Библиотеки algo и engine, с другой стороны, связаны с collector как PRIVATE, поэтому myApp не будет напрямую связан с ними. В разделе 16.2, "Библиотеки", обсуждаются дополнительные поведения для статических библиотек, которые могут привести к дальнейшему связыванию для удовлетворения отношений зависимости, включая циклические зависимости.

В последующих главах представлены несколько других команд target_...(), которые еще больше расширяют информацию о зависимостях, передаваемую между целями. Они позволяют передавать флаги компилятора/линкера и пути поиска заголовков от одной цели к другой, когда они соединены командой target_link_libraries(). Эти возможности постепенно добавлялись начиная с CMake 2.8.11 и до 3.2 и привели к значительному упрощению и повышению надежности файлов CMakeLists.txt.

В последующих главах также обсуждается использование более сложных иерархий исходных каталогов. В таких случаях имя targetName, используемое в target_link_libraries(), должно быть определено командой add_executable() или add_library() в том же каталоге, из которого вызывается target_link_libraries().

4.4. Связывание нецелевых показателей

В предыдущем разделе все элементы, к которым выполнялась привязка, были существующими целями CMake, но команда target_link_libraries() является более гибкой. Помимо целей CMake, в команде target_link_libraries() в качестве элементов могут быть указаны следующие вещи:

Полный путь к файлу библиотеки

CMake добавит файл библиотеки в команду компоновщика. Если файл библиотеки изменится, CMake обнаружит это изменение и перекомпонует цель. Обратите внимание, что начиная с версии CMake 3.3, команда компоновщика всегда использует полный указанный путь, но до версии 3.3 были некоторые ситуации, когда CMake мог попросить компоновщика поискать библиотеку вместо этого (например, заменить /usr/lib/libfoo.so на -lfoo). Обоснование и детали поведения до версии 3.3 нетривиальны и в основном относятся к истории, но для заинтересованного читателя полный набор информации доступен в документации CMake под политикой CMP0060.

Обычное имя библиотеки

Если указано только имя библиотеки без пути, команда компоновщика будет искать эту библиотеку (например, foo становится -lfoo или foo.lib, в зависимости от платформы). Это типично для библиотек, предоставляемых системой.

Флаг ссылки

В качестве особого случая элементы, начинающиеся с дефиса, кроме -l или -framework, будут рассматриваться как флаги, добавляемые к команде компоновщика. Документация CMake предупреждает, что их следует использовать только для элементов PRIVATE, поскольку они будут перенесены на другие цели, если их определить как PUBLIC или INTERFACE, а это не всегда безопасно.

В дополнение к вышесказанному, по историческим причинам перед любым элементом может стоять одно из ключевых слов debug, optimized или general. Эффект этих ключевых слов заключается в дальнейшем уточнении того, когда следующий за ним элемент должен быть включен, на основании того, настроена ли сборка как отладочная (см. главу 13, *Тип сборки*). Если элементу предшествует ключевое слово debug, то он будет добавлен, только если сборка является отладочной. Если элементу предшествует ключевое слово optimized, то он будет добавлен, только если сборка не является отладочной. Ключевое слово general указывает, что элемент должен быть добавлен для всех конфигураций сборки, что в любом случае является поведением по умолчанию, если ключевое слово не используется. Ключевых слов debug, optimized и general следует избегать в новых проектах, поскольку существуют более ясные, гибкие и надежные способы добиться того же самого с помощью современных возможностей CMake.

4.5. CMake старого образца

Команда target_link_libraries() также имеет несколько других форм, некоторые из которых были частью CMake задолго до версии 2.8.11. Эти формы обсуждаются здесь для понимания старых проектов CMake, но их использование в новых проектах, как правило, не рекомендуется. Следует предпочесть полную форму, показанную ранее, с разделами PRIVATE, PUBLIC и INTERFACE, поскольку она более точно выражает природу зависимостей.

```
target_link_libraries(targetName item [item...])
```

Приведенная выше форма обычно эквивалентна тому, что элементы определяются как PUBLIC, но в некоторых ситуациях они могут рассматриваться как PRIVATE. В частности, если проект определяет цепочку библиотечных зависимостей с использованием как старой, так и новой формы команды, то форма старого стиля обычно будет рассматриваться как PRIVATE.

Другой поддерживаемой, но устаревшей формой является следующая:

```
target_link_libraries(targetName
    LINK_INTERFACE_LIBRARIES item [item...]
)
```

Это предшественник ключевого слова INTERFACE более новой формы, о которой говорилось выше, но его использование не рекомендуется в документации CMake. Его поведение может влиять на различные целевые свойства, при этом параметры политики контролируют это поведение. Это потенциальный источник путаницы для разработчиков, которого можно избежать, используя вместо него более новую форму INTERFACE.

```
target_link_libraries(targetName
    <LINK_PRIVATE|LINK_PUBLIC> lib [lib...]
   [<LINK_PRIVATE|LINK_PUBLIC> lib [lib...]]
)
```

Как и предыдущая форма старого стиля, эта форма является предшественницей версий PRIVATE и PUBLIC с ключевыми словами новой формы. Опять же, форма старого образца имеет ту же путаницу в отношении того, какие целевые свойства она затрагивает, и для новых проектов следует предпочесть форму с ключевыми словами PRIVATE/PUBLIC.

4.6. Рекомендуемые практики

Имена целей не обязательно должны быть связаны с именем проекта. Обычно в руководствах и примерах используется переменная для имени проекта и повторное использование этой переменной для имени исполняемой цели, например:

```
# Poor practice, but very common
set(projectName MyExample)
project(${projectName})
add_executable(${projectName} ...)
```

Это работает только для самых простых проектов и поощряет ряд плохих привычек. Рассматривайте имя проекта и имя исполняемого файла как отдельные, даже если изначально они начинаются одинаково. Задавайте имя проекта напрямую, а не через переменную, выбирайте имя цели в соответствии с тем, что делает цель, а не с тем, частью какого проекта она является, и предполагайте, что в конечном итоге проекту потребуется определить более одной цели. Это закрепляет лучшие привычки, которые будут важны при работе над более сложными многоцелевыми проектами.

При именовании целей для библиотек не поддавайтесь искушению начать или закончить имя с lib. На многих платформах (т.е. почти на всех, кроме Windows) при построении фактического имени библиотеки автоматически ставится префикс lib, чтобы привести его в соответствие с обычным соглашением платформы. Если целевое имя уже начинается с lib, то имена файлов библиотек в результате будут иметь вид liblibsomething.... что люди часто считают ошибкой.

Если нет веских причин для этого, старайтесь не указывать ключевое слово STATIC или SHARED для библиотеки, пока не станет известно, что она необходима. Это позволяет более гибко выбирать между статическими и динамическими библиотеками в качестве общей стратегии проекта. Переменная BUILD_SHARED_LIBS может быть использована для изменения значения по умолчанию в одном месте вместо того, чтобы изменять каждый вызов add_library().

Стремитесь всегда указывать ключевые слова PRIVATE, PUBLIC и/или INTERFACE при вызове команды target_link_libraries(), а не следовать старому синтаксису CMake, который предполагал, что все является PUBLIC. По мере усложнения проекта эти три ключевых слова оказывают все большее влияние на то, как обрабатываются межцелевые зависимости. Их использование с самого начала проекта также заставляет разработчиков задуматься о зависимостях между целями, что может помочь выявить структурные проблемы в проекте гораздо раньше.

Глава 5. Переменные

В предыдущих главах было показано, как определять основные цели и создавать результаты сборки. Само по себе это уже полезно, но CMake поставляется с целым рядом других функций, которые обеспечивают большую гибкость и удобство. Эта глава посвящена одной из самых фундаментальных частей CMake, а именно использованию переменных.

5.1. Основы переменных

Как и в любом другом вычислительном языке, переменные являются краеугольным камнем для выполнения задач в CMake. Самый простой способ определения переменной - это команда set(). Обычная переменная может быть определена в файле CMakeLists.txt следующим образом:

```
set(varName value... [PARENT_SCOPE])
```

Имя переменной, varName, может содержать буквы, цифры и знаки подчеркивания, причем буквы чувствительны к регистру. Имя также может содержать символы ./-+, но они редко встречаются на практике. Другие символы также возможны через косвенные средства, но, опять же, они не встречаются в обычном использовании.

В CMake переменная имеет определенную область видимости, подобно тому, как переменные в других языках имеют область видимости, ограниченную определенной функцией, файлом и т.д. Переменная не может быть прочитана или изменена за пределами своей области видимости. По сравнению с другими языками, область видимости переменных в CMake немного более гибкая, но пока, в простых примерах этой главы, считайте, что область видимости переменной является глобальной. Глава 7 *"Использование подкаталогов"* и глава 8 *"Функции и макросы"* знакомят с ситуациями, когда возникают локальные области видимости, и показывают, как ключевое слово PARENT_SCOPE используется для повышения видимости переменной в объемлющей области видимости.

CMake рассматривает все переменные как строки. В различных контекстах переменные могут интерпретироваться как другой тип, но в конечном итоге это просто строки. При задании значения переменной CMake не требует, чтобы эти значения заключались в кавычки, если только значение не содержит пробелов. Если задано несколько значений, они будут объединены вместе с точкой с запятой, отделяющей каждое значение - результирующая строка представляет собой то, как CMake представляет списки. Следующее должно помочь продемонстрировать поведение.

```
set(myVar a b c)    # myVar = "a;b;c"
set(myVar a;b;c)    # myVar = "a;b;c"
set(myVar "a b c")  # myVar = "a b c"
set(myVar a b;c)    # myVar = "a;b;c"
set(myVar a "b c")  # myVar = "a;b c"
```

Значение переменной можно получить с помощью ${myVar}, который можно использовать везде, где ожидается строка или переменная. CMake отличается особой гибкостью, поскольку эту форму можно использовать рекурсивно или указывать имя другой переменной для установки. Кроме того, CMake не требует определения переменных перед их использованием. Использование неопределенной переменной приводит к подстановке пустой строки без ошибок и предупреждений, как в сценариях оболочки Unix.

```
set(foo ab)               # foo   = "ab"
set(bar ${foo}cd)         # bar   = "abcd"
set(baz ${foo} cd)        # baz   = "ab;cd"
set(myVar ba)             # myVar = "ba"
set(big "${${myVar}r}ef") # big   = "${bar}ef" = "abcdef"
set(${foo} xyz)           # ab    = "xyz"
set(bar ${notSetVar})     # bar   = ""
```

Строки не ограничиваются одной строкой, они могут содержать встроенные символы новой строки. Они также могут содержать кавычки, которые необходимо экранировать обратным слешем.

```
set(myVar "goes here")
set(multiLine "First line ${myVar}
Second line with a \"quoted\" word")
```

Если используется CMake 3.0 или более поздняя версия, альтернативой кавычкам может быть использование синтаксиса скобок, вдохновленного lua, где начало содержимого обозначается [=[а конец]=]. Между квадратными скобками может находиться любое количество символов = в том числе и вообще никаких, но в начале и в конце должно быть использовано одинаковое количество символов =. Если за открывающими скобками сразу следует символ новой строки, то первая новая строка игнорируется, но последующие новые строки не игнорируются. Кроме того, никакие дальнейшие преобразования содержимого скобок не выполняются (т.е. не производится подстановка переменных или экранирование).

```
# Simple multi-line content with bracket syntax,
# no = needed between the square bracket markers
set(multiLine [[
First line
Second line
]])

# Bracket syntax prevents unwanted substitution
set(shellScript [=[
#!/bin/bash

[[ -n "${USER}" ]] && echo "Have USER"
]=])

# Equivalent code without bracket syntax
set(shellScript
"#!/bin/bash

[[ -n \"\${USER}\" ]] && echo \"Have USER\"
")
```

Как видно из приведенного выше примера, синтаксис скобок особенно хорошо подходит для определения такого содержимого, как сценарии оболочки Unix. Такое содержимое использует синтаксис ${...} для своих целей и часто содержит кавычки, но использование синтаксиса скобок означает, что их не нужно экранировать, в отличие от традиционного стиля определения содержимого CMake с помощью кавычек. Гибкость использования любого количества символов = между маркерами [и] также означает, что встроенные квадратные скобки не будут неправильно истолкованы как маркеры.

Глава 18 "*Работа с файлами*" содержит дополнительные примеры, которые подчеркивают ситуации, когда синтаксис скобок может быть лучшей альтернативой.

Переменная может быть снята либо вызовом unset(), либо вызовом set() без значения для именованной переменной. Следующие команды эквивалентны, без ошибки или предупреждения, если myVar еще не существует:

```
set(myVar)
unset(myVar)
```

Помимо переменных, определяемых проектом для собственного использования, на поведение многих команд CMake может влиять значение определенных переменных в момент вызова команды. Это обычная схема, используемая CMake для настройки поведения команд или изменения значений по умолчанию, чтобы их не приходилось повторять для каждой команды, определения цели и т.д. В справочной документации CMake для каждой команды обычно перечисляются все переменные, которые могут повлиять на поведение этой команды. В последующих главах этой книги также рассматривается ряд полезных переменных и то, как они влияют на сборку или предоставляют информацию о ней.

5.2. Переменные среды

CMake также позволяет получать и устанавливать значения переменных окружения, используя модифицированную форму обычной нотации переменных. Значение переменной окружения можно получить с помощью специальной формы $ENV{varName} и использовать ее везде, где можно использовать обычную форму ${varName}. Установка переменной окружения может быть выполнена так же, как и обычной переменной, только вместо переменной ENV{varName} указывается просто varName. Например:

```
set(ENV{PATH} "$ENV{PATH}:/opt/myDir")
```

Обратите внимание, однако, что такая установка переменной окружения влияет только на текущий запущенный экземпляр CMake. Как только запуск CMake завершится, изменение переменной окружения будет потеряно. В частности, изменение переменной окружения не будет видно во время сборки. Поэтому установка переменных окружения в файле CMakeLists.txt подобным образом редко бывает полезной.

5.3. Переменные кэша

В дополнение к обычным переменным, о которых говорилось выше, CMake также поддерживает кэш-переменные. В отличие от обычных переменных, время жизни которых ограничено обработкой файла CMakeLists.txt, кэш-переменные хранятся в специальном файле CMakeCache.txt в каталоге сборки и сохраняются между запусками CMake. После установки переменные кэша остаются установленными до тех пор, пока что-то явно не удалит их из кэша. Значение переменной кэша извлекается точно так же, как и обычной переменной (т.е. с помощью формы ${myVar}), но команда set() отличается, когда используется для установки переменной кэша:

```
set(varName value... CACHE type "docstring" [FORCE])
```

Когда присутствует ключевое слово CACHE, команда set() будет применяться не к обычной переменной, а к переменной кэша с именем varName. Кэш-переменные содержат больше информации, чем обычные переменные, включая номинальный тип и строку документации. При установке кэш-переменной необходимо

указать оба типа, хотя строка документации может быть пустой. Ни номинальный тип, ни строка документации не влияют на то, как CMake обращается с переменной, они используются только инструментами графического интерфейса, чтобы представить переменную пользователю в более подходящей форме. При обработке CMake всегда будет рассматривать переменную как строку, а тип используется только для улучшения работы пользователя в инструментах графического интерфейса. Тип должен быть одним из следующих:

BOOL

Переменная кэша представляет собой булево значение включения/выключения. Инструменты графического интерфейса используют флажок или аналогичный элемент для представления переменной. Базовое строковое значение, хранящееся в переменной, будет соответствовать одному из способов CMake представляет булевы как строки (ON/OFF, TRUE/FALSE, 1/0 и т.д. - подробности см. в разделе 6.1.1, "Основные выражения").

FILEPATH

Переменная cache представляет собой путь к файлу на диске. Инструменты GUI представляют пользователю диалог файла для изменения значения переменной.

PATH

Аналогично FILEPATH, но инструменты GUI представляют диалог, в котором выбирается каталог, а не файл.

STRING

Переменная рассматривается как произвольная строка. По умолчанию инструменты GUI используют однострочный виджет редактирования текста для манипулирования значением переменной. Проекты могут использовать свойства переменной кэша, чтобы предоставить предварительно определенный набор значений для инструментов GUI, которые вместо этого будут представлены в виде комбинированного окна или аналогичных элементов (см. раздел 9.6, "Свойства переменной кэша").

INTERNAL

Переменная не предназначена для того, чтобы быть доступной пользователю. Внутренние переменные кэша иногда используются для постоянной записи внутренней информации проекта, например, для кэширования результата интенсивного запроса или вычисления. Инструменты графического интерфейса не отображают ВНУТРЕННИЕ переменные.

Инструменты графического интерфейса обычно используют docstring в качестве всплывающей подсказки для переменной кэша или в качестве короткого однострочного описания при выборе переменной. doc-строка должна быть короткой и состоять из обычного текста (т.е. без HTML-разметки и т.д.). Ключевое слово FORCE рассматривается ниже.

Установка булевой переменной кэша является настолько распространенной необходимостью, что CMake предоставляет отдельную команду специально для этой цели. Вместо несколько многословной команды set() разработчики могут использовать option():

```
option(optVar helpString [initialValue])
```

Если initialValue опущено, будет использоваться значение по умолчанию OFF. Если указано, initialValue должно соответствовать одному из булевых значений, принимаемых командой set(). Для справки, вышеприведенное эквивалентно:

```
set(optVar initialValue CACHE BOOL helpString)
```

По сравнению с set(), команда option() более четко выражает поведение для булевых кэш-переменных, поэтому, как правило, предпочтительнее использовать именно ее.

Важным различием между обычными и кэш-переменными является то, что команда set() перезаписывает кэш-переменную только при наличии ключевого слова FORCE, в отличие от обычных переменных, где команда set() всегда перезаписывает уже существующее значение. Команда set() действует скорее как set-if-notset при использовании для определения переменных кэша, как и команда option() (которая не имеет возможности FORCE). Основная причина этого заключается в том, что переменные кэша в первую очередь предназначены для разработчиков в качестве точки настройки. Вместо того чтобы жестко кодировать значение в файле CMakeLists.txt как обычная переменная, можно использовать кэш-переменную, чтобы разработчик мог переопределить значение без необходимости редактировать файл CMakeLists.txt. Переменная может быть изменена интерактивными инструментами GUI или скриптами без необходимости изменять что-либо в самом проекте.

Часто не совсем понятно, что обычные и кэш-переменные - это две разные вещи. Можно иметь обычную переменную и переменную кэша с одинаковым именем, но с разными значениями. В таких случаях при использовании ${myVar} CMake будет извлекать значение нормальной переменной, а не кэш-переменной, или, говоря иначе, нормальные переменные имеют приоритет над кэш-переменными. Исключением является то, что при установке значения кэш-переменной, если эта кэш-переменная не существовала до вызова set() или если была использована опция FORCE, то любая нормальная переменная в текущей области видимости фактически удаляется. Обратите внимание, что это, к сожалению, означает возможность различного поведения при первом и последующих запусках CMake, поскольку при первом запуске кэш-переменная не будет существовать, а при последующих запусках будет существовать. Поэтому в первом запуске обычная переменная будет скрыта, а в последующих - нет. Пример должен помочь проиллюстрировать проблему.

```
set(myVar foo)                  # Local myVar
set(result ${myVar})            # result = foo
set(myVar bar CACHE STRING "")  # Cache myVar

set(result ${myVar})            # First run:       result = bar
                                # Subsequent runs: result = foo

set(myVar fred)
set(result ${myVar})            # result = fred
```

Говоря в общих чертах, в результате ${myVar} вернет последнее значение, которое было присвоено myVar, независимо от того, была ли это обычная переменная или переменная кэша. Обсуждения в главе 7 *"Использование подкаталогов"* и главе 8 *"Функции и макросы"* проясняют это поведение, объясняя, как область видимости переменной может повлиять на то, какое значение вернет ${myVar}.

5.4. Манипулирование переменными кэша

Используя set() и option(), проект может создать полезный набор точек настройки для своих разработчиков. Можно включать и выключать различные части сборки, задавать пути к внешним пакетам, изменять флаги компиляторов и компоновщиков и так далее. В последующих главах будут рассмотрены эти и другие способы использования переменных кэша, но сначала необходимо понять, как манипулировать этими переменными. Есть два основных способа, которыми разработчики могут это делать: из командной строки cmake или с помощью инструмента GUI.

5.4.1. Установка значений кэша в командной строке

CMake позволяет управлять переменными кэша непосредственно с помощью опций командной строки, передаваемых cmake. Основной рабочей лошадкой является опция -D, которая используется для определения значения переменной кэша.

```
cmake -D myVar:type=someValue ...
```

someValue заменит любое предыдущее значение кэш-переменной myVar. Поведение в основном такое же, как если бы переменная была назначена с помощью команды set() с опциями CACHE и FORCE. Опцию командной строки нужно указать только один раз, поскольку она сохраняется в кэше для последующих запусков и поэтому ее не нужно указывать при каждом запуске cmake. Можно указать несколько опций -D для одновременной установки нескольких переменных в командной строке cmake.

При определении переменных кэша таким образом их не нужно задавать в файле CMakeLists.txt (т.е. не требуется соответствующая команда set()). Кэш-переменные, определенные в командной строке, имеют пустую строку docstring. Тип также может быть опущен, в этом случае переменной присваивается специальный тип, аналогичный INTERNAL. Ниже показаны различные примеры задания переменных кэша через командную строку.

```
cmake -D foo:BOOL=ON ...
cmake -D "bar:STRING=This contains spaces" ...
cmake -D hideMe=mysteryValue ...
cmake -D helpers:FILEPATH=subdir/helpers.txt ...
cmake -D helpDir:PATH=/opt/helpThings ...
```

Обратите внимание, что при установке переменной кэша со значением, содержащим пробелы, все значение, заданное с помощью опции -D, должно быть заключено в кавычки.

Существует специальный случай для работы со значениями, изначально объявленными без типа в командной строке cmake. Если затем в файле CMakeLists.txt проекта попытаться установить ту же кэш-переменную и указать тип FILEPATH или PATH, то если значение этой кэш-переменной является относительным путем, CMake будет считать его относительным к каталогу, из которого был вызван cmake, и автоматически преобразует его в абсолютный путь. Это не очень надежно, поскольку cmake может быть вызван из любого каталога, а не только из каталога сборки. Поэтому разработчикам рекомендуется всегда указывать тип при задании переменной в командной строке cmake для переменной, которая представляет собой какой-то путь. Хорошей привычкой является вообще всегда указывать тип переменной в командной строке, чтобы в приложениях с графическим интерфейсом она отображалась в наиболее подходящем виде.

Также можно удалить переменные из кэша с помощью опции -U, которую можно повторять по мере необходимости для удаления более одной переменной. Обратите внимание, что опция -U поддерживает * и ? подстановочные знаки, но нужно быть осторожным, чтобы не удалить больше, чем предполагалось, и не оставить кэш в состоянии, не пригодном для сборки. В целом, рекомендуется удалять только конкретные записи без подстановочных знаков, если нет полной уверенности, что используемые подстановочные знаки безопасны.

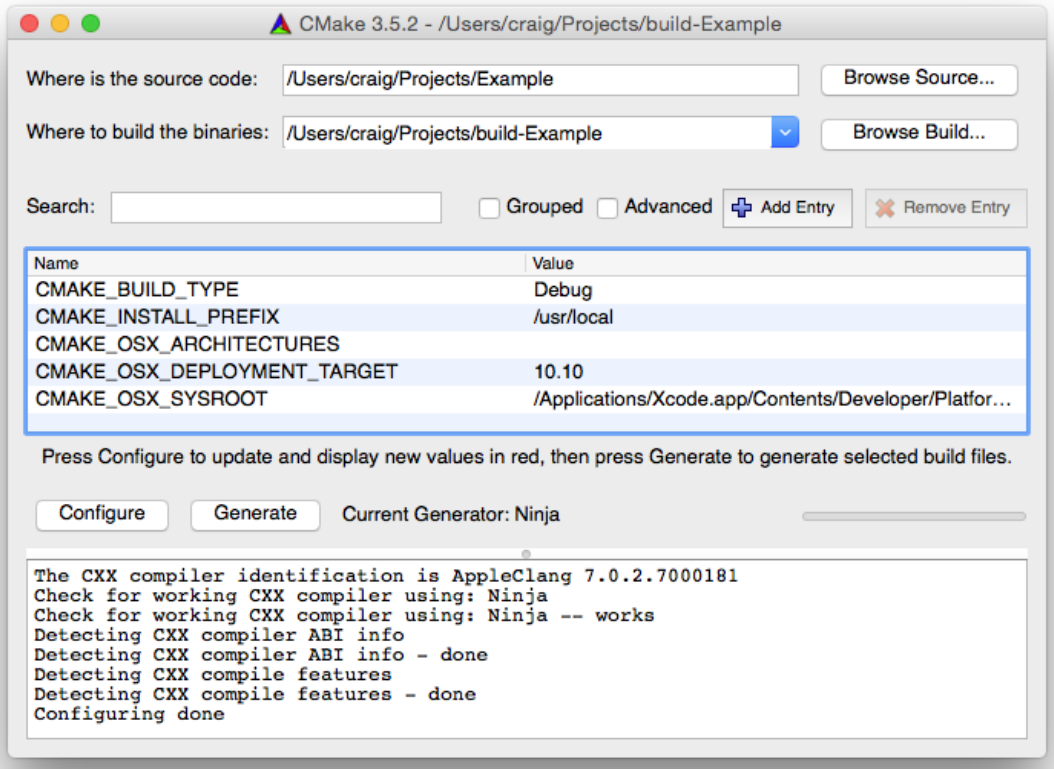
```
cmake -U 'help*' -U foo ...
```

5.4.2. Инструменты графического интерфейса CMake

Установка переменных кэша через командную строку является неотъемлемой частью автоматизированных сценариев сборки и всего, что управляет CMake с помощью команды cmake. Однако для повседневной разработки инструменты графического интерфейса, предоставляемые CMake, часто оказываются более удобными. CMake предоставляет два эквивалентных GUI-инструмента, cmake-gui и ccmake, которые позволяют разработчикам интерактивно управлять переменными кэша. cmake-gui - это полнофункциональное GUI-приложение, поддерживаемое на всех основных настольных платформах, тогда как ccmake использует

интерфейс на основе curses, который можно использовать только в текстовом окружении, например, через ssh-соединение. Оба приложения включены в официальные пакеты релизов CMake на всех платформах. Если в Linux используются пакеты, предоставляемые системой, а не официальные релизы, обратите внимание, что во многих дистрибутивах cmake-gui выделен в отдельный пакет.

Пользовательский интерфейс cmake-gui показан на рисунке ниже. Верхняя часть позволяет определить исходный код проекта и каталоги сборки, в средней части можно просматривать и редактировать переменные кэша, а в нижней части находятся кнопки Configure и Generate и соответствующая им область журнала.



Исходный каталог должен быть установлен в каталог, содержащий файл CMakeLists.txt в верхней части дерева исходных текстов проекта. Каталог сборки - это место, где CMake будет генерировать все выходные данные сборки (рекомендуемые схемы расположения каталогов обсуждались в главе 2, "*Настройка проекта*"). Для новых проектов необходимо установить оба параметра, но для существующих проектов установка каталога сборки также приведет к обновлению каталога исходных текстов, поскольку расположение исходных текстов хранится в кэше каталога сборки.

Двухэтапный процесс настройки CMake был представлен в разделе 2.3, "Генерация файлов проекта". На первом этапе читается файл CMakeLists.txt и в памяти создается представление проекта. Этот этап называется этапом *configure*. Если этап configure прошел успешно, то затем может быть выполнен этап *generate* для создания файлов проекта инструмента сборки в каталоге сборки. При запуске cmake из командной строки оба этапа выполняются автоматически, но в приложении с графическим интерфейсом они запускаются отдельно с помощью кнопок Configure и Generate. Каждый раз при запуске этапа configure обновляются переменные кэша, показанные в центре пользовательского интерфейса. Любые переменные, которые были добавлены недавно или изменили значение после предыдущего запуска, будут выделены красным цветом (при первой загрузке проекта все переменные отображаются выделенными). Хорошей практикой является повторный запуск этапа configure до тех пор, пока не останется никаких изменений, так как это обеспечивает надежное поведение для более сложных проектов, где включение некоторых опций может добавить дополнительные опции, которые

могут потребовать еще одного прохода configure. Когда все переменные кэша отображаются без красного выделения, можно запускать этап generate. Пример на предыдущем снимке экрана показывает типичный вывод журнала после выполнения этапа configure и отсутствия изменений в переменных кэша.

При наведении курсора мыши на любую из переменных кэша появится всплывающая подсказка, содержащая строку документа для этой переменной. Новые переменные кэша можно также добавить вручную с помощью кнопки Add Entry, что эквивалентно выполнению команды set() с пустой строкой docstring. Кэш-переменные можно удалить с помощью кнопки Remove Entry, хотя CMake, скорее всего, создаст эту переменную заново при следующем запуске.

Щелчок по переменной позволяет редактировать ее значение в виджете, соответствующем типу переменной. Булевы значения отображаются в виде флажка, файлы и пути имеют кнопку просмотра файловой системы, а строки обычно представлены в виде редактирования текстовой строки. В качестве особого случая кэш-переменным типа STRING можно задать набор значений для отображения в комбинированном окне в графическом интерфейсе CMake вместо отображения простого виджета ввода текста. Это достигается установкой свойства STRINGS кэш-переменной (подробно рассмотрено в разделе 9.6, "Свойства кэш-переменной", но для удобства показано здесь):

```
set(trafficLight Green CACHE STRING "Status of something")
set_property(CACHE trafficLight PROPERTY STRINGS Red Orange Green)
```

В приведенном выше примере кэш-переменная trafficLight изначально будет иметь значение Green. Когда пользователь попытается изменить trafficLight в cmake-gui, ему будет предоставлен combobox, содержащий три значения Red, Orange и Green, вместо простого виджета редактирования строки, который в противном случае позволил бы ввести произвольный текст. Обратите внимание, что установка свойства STRINGS для переменной не препятствует присвоению ей других значений, а влияет только на виджет, используемый cmake-gui при ее редактировании. Переменной по-прежнему можно присвоить другие значения с помощью команд set() в файле CMakeLists.txt или другим способом, например, вручную отредактировав файл CMakeCache.txt.

Переменные кэша также могут иметь свойство, отмечающее их как продвинутые или нет. Это тоже влияет только на способ отображения переменной в cmake-gui, но никак не на то, как CMake использует переменную во время обработки. По умолчанию cmake-gui отображает только не продвинутые переменные, что обычно представляет только основные переменные, в просмотре или изменении которых заинтересован разработчик. Включение опции Advanced показывает все переменные кэша, кроме тех, которые помечены как INTERNAL (единственный способ увидеть переменные INTERNAL - это отредактировать файл CMakeCache.txt с помощью текстового редактора, поскольку они не предназначены для непосредственного манипулирования разработчиками). Переменные могут быть помечены как расширенные с помощью команды mark_as_advanced() в файле CMakeLists.txt:

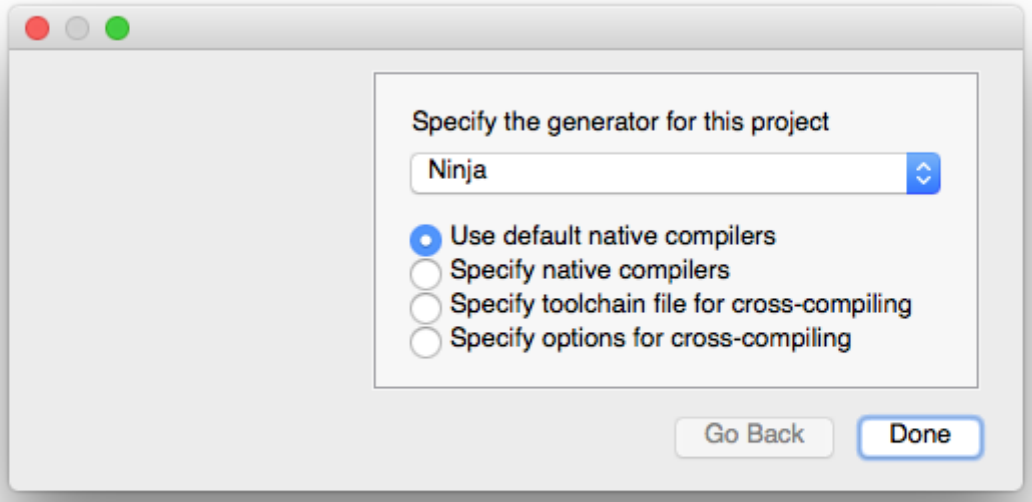
```
mark_as_advanced([CLEAR|FORCE] var1 [var2...])
```

Ключевое слово CLEAR гарантирует, что переменные не будут помечены как продвинутые, а ключевое слово FORCE гарантирует, что переменные будут помечены как продвинутые. Без ключевого слова переменные будут помечены как продвинутые, только если они еще не имеют состояния продвинутый/не продвинутый.

Выбор опции Grouped может упростить просмотр расширенных переменных, сгруппировав переменные вместе на основе начала имени переменной до первого подчеркивания. Другой способ фильтрации списка

отображаемых переменных - ввести текст в области поиска, в результате чего будут показаны только переменные, в имени или значении которых содержится указанный текст.

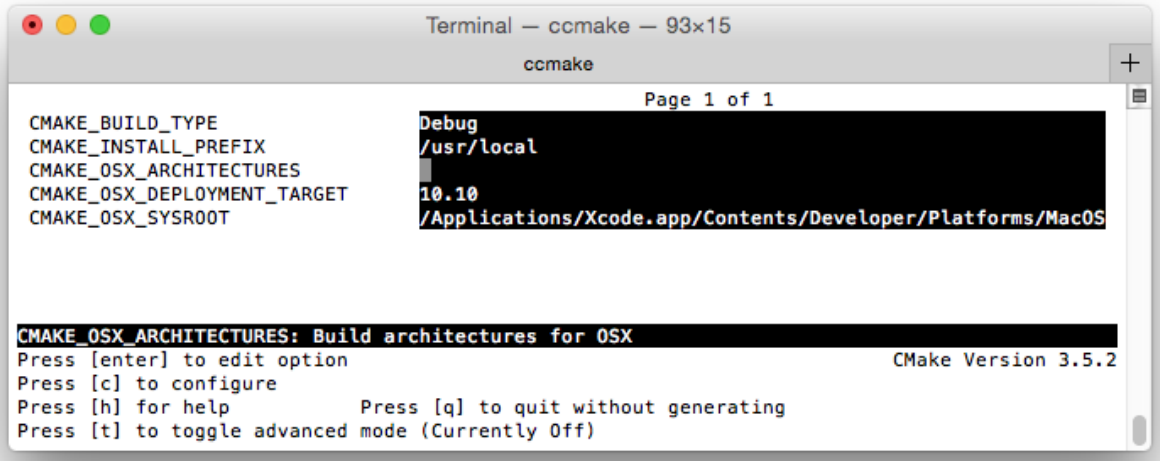
Когда этап configure выполняется впервые в новом проекте, перед разработчиком появляется диалог, подобный тому, что показан на следующем снимке экрана:



В этом диалоговом окне задается генератор CMake и цепочка инструментов. Выбор генератора обычно зависит от личных предпочтений разработчика, а доступные опции представлены в комбинированном окне. В зависимости от проекта, выбор генератора может быть более ограниченным, чем позволяют опции combobox, например, если проект полагается на функциональность, специфичную для генератора. Частым примером является проект, для которого требуется генератор Xcode из-за уникальных возможностей платформы Apple, таких как подпись кода и поддержка iOS/tvOS/watchOS. После выбора генератора для проекта его нельзя изменить без удаления кэша и начала работы заново, что при необходимости можно сделать из меню Файл.

Для представленных вариантов цепочки инструментов каждый из них требует от разработчика все больше информации. Использование родных компиляторов по умолчанию является обычным выбором для обычной разработки настольных систем, и выбор этого варианта не требует дополнительных сведений. Если требуется больший контроль, разработчик может переопределить родные компиляторы, указав пути к ним в последующем диалоге. Если имеется отдельный файл цепочки инструментов, его можно использовать для настройки не только компиляторов, но и целевого окружения, флагов компилятора и различных других вещей. Использование файла цепочки инструментов характерно для кросс-компиляции, которая подробно рассматривается в главе 21 "*Цепочки инструментов и кросс-компиляция*". Наконец, для максимального контроля разработчики могут указать полный набор опций для кросс-компиляции, но это не рекомендуется для обычного использования. Файл цепочки инструментов может предоставить ту же информацию, но его преимущество в том, что его можно использовать повторно по мере необходимости.

Инструмент ccmake предлагает все те же функции, что и приложение cmake-gui, но делает это через текстовый интерфейс



Вместо выбора каталогов исходников и сборки, как в cmake-gui, каталог исходников или сборки должен быть указан в командной строке ccmake, как и для команды cmake.

Основным недостатком интерфейса ccmake является то, что вывод журнала не так удобен, как в версии cmake-gui. Также не предусмотрена возможность фильтрации отображаемых переменных, а методы редактирования переменных не так богаты, как в cmake-gui. В остальном, инструмент ccmake является полезной альтернативой, когда полное приложение cmake-gui непрактично или недоступно, например, через терминальное соединение, которое не поддерживает переадресацию UI.

5.5. Отладочные переменные и диагностика

По мере усложнения проектов или при исследовании неожиданного поведения может оказаться полезным выводить диагностические сообщения и значения переменных во время выполнения CMake. Обычно для этого используется команда message().

```
message([mode] msg1 [msg2]...)
```

Если указано более одного msg, они будут объединены в одну строку без разделителей. Обычно это не то, что задумал разработчик, поэтому чаще всего используется один msg с сообщением, заключенным в кавычки для сохранения пробелов. Можно использовать переменные значения, которые будут оцениваться перед печатью результата. Например:

```
set(myVar HiThere)
message("The value of myVar = ${myVar}")
```

Это даст следующий результат:

```
The value of myVar = HiThere
```

Команда message() принимает необязательное ключевое слово mode, которое влияет на способ вывода сообщения и в некоторых случаях останавливает сборку с ошибкой. Распознаваемыми значениями режима являются:

STATUS

Случайная информация. Сообщениям обычно предшествуют два дефиса.

WARNING

Предупреждение CMake, обычно отображается красным цветом, если поддерживается (консоль командной строки cmake или область журнала cmake-gui). Обработка будет продолжена.

AUTHOR_WARNING

Подобно WARNING, но показывается только в том случае, если включены предупреждения разработчика (требуется опция -Wdev в командной строке cmake). Проекты не часто используют этот конкретный тип сообщений.

SEND_ERROR

Указывает на сообщение об ошибке, которое будет показано красным цветом, если поддерживается. Обработка будет продолжаться до завершения этапа конфигурирования, но генерация не будет выполнена. Это похоже на ошибку, которая позволяет попытаться продолжить обработку, но в конечном итоге все равно указывает на неудачу.

FATAL_ERROR

Обозначает жесткую ошибку. Сообщение будет напечатано, и обработка немедленно прекратится. В журнал также обычно записывается местоположение команды fatal message().

DEPRECATION

Специальная категория, используемая для регистрации сообщения об устаревании. Если переменная CMAKE_ERROR_DEPRECATED имеет значение boolean true, сообщение будет рассматриваться как ошибка. Если переменная CMAKE_WARN_DEPRECATED имеет значение boolean true, сообщение будет рассматриваться как предупреждение. Если ни одна из переменных не определена, сообщение не будет показано.

Если ключевое слово mode не указано, то сообщение считается важной информацией и регистрируется без каких-либо изменений. Однако следует отметить, что протоколирование в режиме STATUS - это не то же самое, что протоколирование сообщения без ключевого слова mode. При использовании режима STATUS сообщение будет напечатано в правильной последовательности с другими сообщениями CMake и будет предваряться двумя дефисами, тогда как при отсутствии ключевого слова mode дефисы не добавляются, и нередко сообщение появляется не по порядку относительно других сообщений, включающих ключевое слово mode.

Другим механизмом, который CMake предоставляет для отладки использования переменных, является команда variable_watch(). Она предназначена для более сложных проектов, где может быть непонятно, как переменная получила то или иное значение. Когда переменная находится под наблюдением, все попытки прочитать или изменить ее записываются в журнал.

```
variable_watch(myVar [command])
```

В подавляющем большинстве случаев достаточно просто перечислить переменную, за которой нужно следить, без дополнительной команды, поскольку она регистрирует все обращения к указанной переменной. Однако для окончательного контроля можно задать команду, которая будет выполняться каждый раз, когда переменная будет прочитана или изменена. Ожидается, что команда будет именем функции или макроса CMake (см. главу 8, *Функции и макросы*), и ей будут переданы следующие аргументы: имя переменной, тип доступа, значение переменной, имя текущего файла списка и стек файла списка (файлы списка рассматриваются в главе 7, *Использование подкаталогов*). Указание команды с помощью variable_watch() было бы очень редким явлением.

5.6. Обработка строк

По мере роста сложности проекта во многих случаях возникает необходимость в реализации более сложной логики управления переменными. Основным инструментом CMake для этого является команда string(), которая предоставляет широкий спектр полезных функций работы со строками. Эта команда позволяет проектам выполнять операции поиска и замены, сопоставления регулярных выражений, преобразования верхнего/нижнего регистра, удаление пробелов и другие общие задачи. Некоторые из наиболее часто используемых функций представлены ниже, но справочная документация CMake должна рассматриваться как канонический источник информации обо всех доступных операциях и их поведении.

Первый аргумент string() определяет выполняемую операцию, а последующие аргументы зависят от запрашиваемой операции. Эти аргументы обычно требуют как минимум одной входной строки и, поскольку команды CMake не могут возвращать значение, выходной переменной для результата операции. В приведенном ниже материале эта выходная переменная обычно называется outVar.

```
string(FIND inputString subString outVar [REVERSE])
```

FIND ищет подстроку в inputString и сохраняет индекс найденной подстроки в outVar (первый символ - индекс 0). Найдено первое вхождение, если не указано REVERSE, в этом случае будет найдено последнее вхождение. Если подстрока не встречается в inputString, то outVar будет присвоено значение -1.

```
set(longStr  abcdefabcdef)
set(shortBit def)

string(FIND ${longStr} ${shortBit} fwdIndex)
string(FIND ${longStr} ${shortBit} revIndex REVERSE)

message("fwdIndex = ${fwdIndex}, revIndex = ${revIndex}")
```

Это приводит к следующему результату:

```
fwdIndex = 3, revIndex = 9
```

Замена простой подстроки происходит по аналогичной схеме:

```
string(REPLACE matchString replaceWith outVar input [input...])
```

Операция REPLACE заменит каждое вхождение matchString во входных строках на replaceWith и сохранит результат в outVar. Если задано несколько входных строк, то перед поиском замен они объединяются без разделителя между строками. Это иногда может привести к неожиданным совпадениям, поэтому в большинстве случаев разработчики предоставляют только одну входную строку.

Регулярные выражения также хорошо поддерживаются операцией REGEX, причем доступно несколько различных вариантов, определяемых вторым аргументом:

```
string(REGEX MATCH    regex outVar input [input...])
string(REGEX MATCHALL regex outVar input [input...])
string(REGEX REPLACE  regex replaceWith outVar input [input...])
```

Регулярное выражение для сопоставления, regex, может использовать типичный базовый синтаксис регулярных выражений (полную спецификацию см. в справочной документации CMake), хотя некоторые общие функции, такие как отрицание, не поддерживаются. Перед подстановкой входные строки конкатенируются. Операция MATCH находит только первое совпадение и сохраняет его в outVar. MATCHALL находит все совпадения и сохраняет их в outVar в виде списка. REPLACE возвращает всю входную строку с заменой каждого совпадения на replaceWith. На совпадения можно ссылаться в replaceWith, используя обычные обозначения \1, \2 и т.д., но обратите внимание, что обратные слеши должны быть экранированы в CMake. Следующие примеры демонстрируют требуемый синтаксис:

```
set(longStr abcdefabcdef)

string(REGEX MATCHALL "[ace]" matchVar ${longStr})
string(REGEX REPLACE "([de])" "X\\1Y" replVar ${longStr})

message("matchVar = ${matchVar}")
message("replVar = ${replVar}")
```

В результате получается следующее:

```
matchVar = a;c;e;a;c;e
replVar = abcXdYXeYfabcXdYXeYf
```

Извлечение подстроки также возможно:

```
string(SUBSTRING input index length outVar)
```

Индекс - это целое число, определяющее начало подстроки, которую нужно извлечь из входной строки. Будет извлечено до символов длины, или, если длина равна -1, возвращаемая подстрока будет содержать все символы до конца входной строки. Обратите внимание, что в CMake 3.1 и более ранних версиях ошибка возникала, если length указывала на конец строки.

Длина строки может быть тривиально получена, и строки могут быть легко преобразованы в верхний или нижний регистр. Также легко удалить пробелы из начала и конца строки. Синтаксис для всех этих операций имеет одинаковую форму:

```
string(LENGTH  input outVar)
string(TOLOWER input outVar)
string(TOUPPER input outVar)
string(STRIP   input outVar)
```

CMake предоставляет другие операции, такие как сравнение строк, хеширование, временные метки и другие, но их использование менее распространено в повседневных проектах CMake. Заинтересованному читателю следует обратиться к справочной документации CMake по команде string() для получения подробной информации.

5.7. Списки

Списки активно используются в CMake. В конечном итоге списки представляют собой одну строку с элементами списка, разделенными точкой с запятой, что может сделать менее удобным манипулирование отдельными

элементами списка. Для облегчения таких задач CMake предоставляет команду list(). Как и для команды string(), list() в качестве первого аргумента ожидает операцию, которую необходимо выполнить. Вторым аргументом всегда является список, с которым нужно работать, и он должен быть переменной (т.е. передача необработанного списка типа a;b;c не допускается).

Самые основные операции со списком - это подсчет количества элементов и извлечение одного или нескольких элементов из списка:

```
list(LENGTH listVar outVar)
list(GET    listVar index [index...] outVar)
```

```
# Example
set(myList a b c)        # Creates the list "a;b;c"
list(LENGTH myList len)
message("length  = ${len}")

list(GET myList 2 1 letters)
message("letters = ${letters}")
```

Вывод приведенного выше примера будет следующим:

```
length  = 3
letters = c;b
```

Добавление и вставка элементов также является распространенной задачей:

```
list(APPEND listVar item [item...])
list(INSERT listVar index item [item...])
```

В отличие от случаев LENGTH и GET, APPEND и INSERT действуют непосредственно на listVar и изменяют его на месте, как показано в следующем примере:

```
set(myList a b c)
list(APPEND myList d e f)
message("myList (first)  = ${myList}")

list(INSERT myList 2 X Y Z)
message("myList (second) = ${myList}")
```

Что дает следующий результат:

```
myList (first)  = a;b;c;d;e;f
myList (second) = a;b;X;Y;Z;c;d;e;f
```

Поиск определенного элемента в списке происходит по ожидаемой схеме:

```
list(FIND myList value outVar)
```

```
# Example
set(myList a b c d e)
list(FIND myList d index)
message("index = ${index}")
```

Ожидаемый результат:

```
index = 3
```

Для удаления элементов предусмотрены три операции, каждая из которых изменяет список напрямую:

```
list(REMOVE_ITEM myList value [value...])
list(REMOVE_AT myList index [index...])
list(REMOVE_DUPLICATES myList)
```

Операция REMOVE_ITEM может быть использована для удаления одного или нескольких элементов из списка. Если элемента нет в списке, это не является ошибкой. REMOVE_AT, с другой стороны, указывает один или несколько индексов для удаления, и CMake завершит работу с ошибкой, если любой из указанных индексов окажется за концом списка. REMOVE_DUPLICATES гарантирует, что список будет содержать только уникальные элементы.

Элементы списка можно также упорядочить с помощью операций REVERSE или SORT (сортировка производится по алфавиту):

```
list(REVERSE myList)
list(SORT myList)
```

Для всех операций со списком, которые принимают индекс в качестве входных данных, индекс может быть отрицательным, чтобы показать, что отсчет начинается не с начала, а с конца списка. При таком использовании последний элемент в списке имеет индекс -1, второй последний -2 и так далее.

Выше описано большинство доступных подкоманд list(). Все они поддерживаются, по крайней мере, с версии CMake 3.0, поэтому проекты, как правило, могут рассчитывать на их наличие. Полный список поддерживаемых подкоманд, включая те, которые были добавлены в более поздних версиях CMake, можно найти в документации CMake.

5.8. Math

Еще одной распространенной формой манипулирования переменными являются математические вычисления. CMake предоставляет команду math() для выполнения базовой математической оценки:

```
math(EXPR outVar mathExpr)
```

Первый аргумент должен содержать ключевое слово EXPR, а mathExpr определяет выражение, которое будет вычислено, а результат будет сохранен в outVar. В выражении может использоваться любой из следующих операторов, которые имеют то же значение, что и в коде на языке C: + - * / % | & ^ ~ << >> * / %. Также поддерживаются круглые скобки, которые имеют обычное математическое значение. На переменные в mathExpr можно ссылаться с помощью обычной нотации ${myVar}.

```
set(x 3)
set(y 7)
math(EXPR z "(${x}+${y}) / 2")
message("result = ${z}")
```

Ожидаемый результат:

```
result = 5
```

5.9. Рекомендуемые практики

Если среда разработки позволяет, инструмент CMake GUI - полезный способ быстро и легко понять параметры сборки проекта и изменять их по мере необходимости в процессе разработки. Немного времени, потраченного на ознакомление с ним, упростит работу с более сложными проектами в дальнейшем. Он также дает разработчикам хорошую базу для работы, если им понадобится поэкспериментировать с такими вещами, как настройки компилятора, поскольку их легко найти и изменить в среде графического интерфейса.

Предпочтительнее предоставлять переменные кэша для управления включением необязательных частей сборки вместо того, чтобы кодировать логику в сценариях сборки вне CMake. Это позволит тривиально включать и выключать их в графическом интерфейсе CMake и других инструментах, которые понимают, как анализировать файлы CMakeLists.txt (все большее число сред IDE обретают такую возможность).

Старайтесь не полагаться на определение переменных окружения, кроме, возможно, вездесущего PATH или аналогичных переменных уровня операционной системы. Сборка должна быть предсказуемой, надежной и простой в настройке, но если она полагается на установку переменных окружения для корректной работы, это может стать причиной разочарования для начинающих разработчиков, которые пытаются настроить окружение сборки. Кроме того, окружение в момент запуска CMake может измениться по сравнению с тем, когда вызывается сама сборка. Поэтому по возможности предпочитайте передавать информацию непосредственно в CMake через переменные кэша.

Постарайтесь заблаговременно установить соглашение об именовании переменных. Для переменных кэша рассмотрите возможность группировки связанных переменных под общим префиксом, за которым следует знак подчеркивания, чтобы воспользоваться преимуществами того, как CMake GUI автоматически группирует переменные на основе того же префикса. Также учитывайте, что проект может однажды стать частью какого-то более крупного проекта, поэтому желательно, чтобы имя начиналось с имени проекта или чего-то, тесно связанного с проектом.

Старайтесь избегать определения в проекте переменных, не относящихся к кэшу, которые имеют те же имена, что и переменные кэша. Взаимодействие между этими двумя типами переменных может быть неожиданным для разработчиков, впервые использующих CMake. В последующих главах также освещаются другие распространенные ошибки и неправильное использование обычных переменных, имеющих одинаковые имена с переменными кэша.

Для сообщений журнала, которые должны оставаться частью сборки, старайтесь всегда указывать режим с помощью команды message(). Если сообщение носит общий информационный характер, лучше использовать STATUS, чем вообще не указывать ключевое слово mode, чтобы сообщения не появлялись в журнале сборки не по порядку. Временные отладочные сообщения часто используют без ключевого слова mode для удобства, но если они могут оставаться частью проекта в течение какого-либо времени, лучше, чтобы они тоже указывали режим (обычно STATUS).

CMake предоставляет большое количество предопределенных переменных, которые предоставляют подробную информацию о системе или влияют на определенные аспекты поведения CMake. Некоторые из этих переменных активно используются в проектах, например, те, которые определяются только при сборке для определенной платформы (WIN32, APPLE, UNIX и т.д.). Поэтому разработчикам рекомендуется время от времени просматривать страницу документации CMake со списком предопределенных переменных, чтобы ознакомиться с тем, что доступно.

Глава 6. Управление потоком

Обычной потребностью большинства проектов CMake является применение некоторых шагов только в определенных обстоятельствах. Например, проекты могут захотеть использовать определенные флаги компилятора только с определенным компилятором или при сборке для определенной платформы. В других случаях проекту может потребоваться итерация по набору значений или повторение некоторого набора шагов до тех пор, пока не будет выполнено определенное условие. Эти примеры управления потоком данных хорошо поддерживаются CMake и знакомы большинству разработчиков программного обеспечения. Вездесущая команда if() обеспечивает ожидаемое поведение if-then-else, а циклическое выполнение обеспечивается командами foreach() и while(). Все три команды обеспечивают традиционное поведение, реализованное в большинстве языков программирования, но они также имеют дополнительные возможности, характерные для CMake.

6.1. Команда if()

Современная форма команды if() выглядит следующим образом (могут быть предусмотрены несколько условий elseif()):

```
if(expression1)
    # commands ...
elseif(expression2)
    # commands ...
else()
    # commands ...
endif()
```

Очень ранние версии CMake требовали, чтобы выражение1 повторялось в качестве аргумента в выражениях else() и endif(), но начиная с версии CMake 2.8.0 этого не требуется. Хотя до сих пор нередко можно встретить проекты и примеры кода, использующие эту старую форму, ее не рекомендуется использовать в новых проектах, поскольку она может несколько запутать читателя. В новых проектах аргументы else() и endif() следует оставлять пустыми, как показано выше.

Выражения в командах if() и elseif() могут принимать самые разные формы. CMake предлагает традиционную булеву логику, а также различные другие условия, такие как тесты файловой системы, сравнение версий и проверка на существование вещей.

6.1.1. Основные выражения

Самым простым из всех выражений является единственная константа:

```
if(value)
```

Логика CMake в отношении того, что считать истинным и ложным, немного сложнее, чем в большинстве языков программирования. Для одной константы без кавычек правила следующие:

- Если value - это константа без кавычек со значением 1, ON, YES, TRUE, Y или ненулевое число, оно рассматривается как true. Тест не чувствителен к регистру.

- Если value - это константа без кавычек со значением 0, OFF, NO, FALSE, N, IGNORE, NOTFOUND, пустая строка или строка, заканчивающаяся на -NOTFOUND, она рассматривается как false. Опять же, тест не чувствителен к регистру.

- Если ни один из этих двух случаев не применим, он будет рассматриваться как имя переменной (или, возможно, как строка) и оцениваться далее, как описано ниже.

В следующих примерах для наглядности показана только часть команды if(...), соответствующее тело и endif() опущены:

```
# Examples of unquoted constants
if(YES)
if(0)
if(TRUE)

# These are also treated as unquoted constants because the
# variable evaluation occurs before if() sees the values
set(A YES)
set(B 0)
if(${A})  # Evaluates to true
if(${B})  # Evaluates to false

# Does not match any of the true or false constants, so proceed
# to testing as a variable name in the fallthrough case below
if(someLetters)

# Quoted value, so bypass the true/false constant matching
# and fall through to testing as a variable name or string
if("someLetters")
```

В документации CMake случай падения называется следующей формой:

```
if(<variable|string>)
```

На практике это означает, что выражение if является либо:

- Не заключенное в кавычки имя (возможно, неопределенной) переменной.

- Строка в кавычках.

Когда используется имя переменной без кавычек, значение переменной сравнивается с ложными константами. Если ни одна из них не совпадает со значением, результатом выражения будет true. Неопределенная переменная будет оцениваться как пустая строка, которая совпадает с одной из ложных констант и поэтому дает результат false.

```
# Common pattern, often used with variables defined
# by commands such as option(enableSomething "...")
if(enableSomething)
    # ...
endif()
```

Однако когда выражение if представляет собой строку в кавычках, поведение становится более сложным:

- В CMake 3.1 и более поздних версиях строка в кавычках всегда оценивается как false, независимо от значения строки (но это можно отменить с помощью настроек политики, см. главу 12, *"Политики"*).
- До версии CMake 3.1, если значение строки совпадало с именем существующей переменной, то строка в кавычках эффективно заменялась именем этой переменной (без кавычек), после чего тест повторялся.

И то, и другое может быть неожиданностью для разработчиков, но, по крайней мере, поведение CMake 3.1 всегда предсказуемо. Поведение до версии 3.1 иногда приводило к неожиданным заменам строк, когда значение строки совпадало с именем переменной, возможно, определенной где-то очень далеко от этой части проекта. Потенциальная путаница вокруг значений, заключенных в кавычки, означает, что обычно рекомендуется избегать использования аргументов в кавычках в форме if(something). Обычно существуют лучшие выражения сравнения, которые более надежно работают со строками, о них мы поговорим ниже в разделе 6.1.3, "Тесты сравнения".

6.1.2. Логические операторы

CMake поддерживает обычные логические операторы AND, OR и NOT, а также круглые скобки для контроля порядка старшинства.

```
# Logical operators
if(NOT expression)
if(expression1 AND expression2)
if(expression1 OR expression2)

# Example with parentheses
if(NOT (expression1 AND (expression2 OR expression3)))
```

Согласно обычным соглашениям, сначала оцениваются выражения в круглых скобках, начиная с самой внутренней круглой скобки.

6.1.3. Сравнительные тесты

CMake разделяет тесты сравнения на три отдельные категории: *числовые*, *строковые* и *номера версий*, но синтаксические формы всех тестов следуют одной и той же схеме:

```
if(value1 OPERATOR value2)
```

Два операнда, value1 и value2, могут быть либо именами переменных, либо (возможно, заключенными в кавычки) значениями. Если значение совпадает с именем определенной переменной, оно будет рассматриваться как переменная. В противном случае оно рассматривается как строка или значение напрямую. И снова, однако, значения в кавычках имеют неоднозначное поведение, аналогичное поведению базовых унарных выражений. До CMake 3.1 строка со значением в кавычках, совпадающим с именем переменной, заменялась значением этой переменной. Поведение CMake 3.1 и последующих версий использует заключенное в кавычки значение без подстановки, чего интуитивно ожидают разработчики.

Все три категории сравнения поддерживают один и тот же набор операций, но имена ОПЕРАТОРОВ для каждой категории разные. В следующей таблице приведены поддерживаемые операторы:

Numeric	String	Version numbers
LESS	STRLESS	VERSION_LESS
GREATER	STRGREATER	VERSION_GREATER
EQUAL	STREQUAL	VERSION_EQUAL
LESS_EQUAL[1]	STRLESS_EQUAL[1]	VERSION_LESS_EQUAL[1]
GREATER_EQUAL[1]	STRGREATER_EQUAL[1]	VERSION_GREATER_EQUAL[1]

[1] Доступно только в CMake 3.7 и более поздних версиях.

Численное сравнение работает, как и следовало ожидать, сравнивая значение левого операнда с правым. Обратите внимание, однако, что CMake обычно не выдает ошибку, если один из операндов не является числом, и его поведение не полностью соответствует официальной документации, когда значения содержат не только цифры. В зависимости от сочетания цифр и нецифр результатом выражения может быть true или false.

```
# Valid numeric expressions, all evaluating as true
if(2 GREATER 1)
if("23" EQUAL 23)
set(val 42)
if(${val} EQUAL 42)
if("${val}" EQUAL 42)

# Invalid expression that evaluates as true with at
# least some CMake versions. Do not rely on this behavior.
if("23a" EQUAL 23)
```

Сравнение номеров версий в некоторой степени напоминает расширенную форму числовых сравнений. Предполагается, что номера версий имеют вид major[.minor[.patch[.tweak]]], где каждый компонент является неотрицательным целым числом. При сравнении двух номеров версий сначала сравнивается основная часть. Только если главные компоненты равны, сравниваются второстепенные части (если они есть) и так далее. Отсутствующий компонент рассматривается как ноль. Во всех следующих примерах выражение имеет значение true:

```
if(1.2   VERSION_EQUAL   1.2.0)
if(1.2   VERSION_LESS    1.2.3)
if(1.2.3 VERSION_GREATER 1.2  )
if(2.0.1 VERSION_GREATER 1.9.7)
if(1.8.2 VERSION_LESS    2    )
```

Сравнение номеров версий имеет те же предостережения по надежности, что и числовые сравнения. Ожидается, что каждый компонент версии будет целым числом, но результат сравнения по существу не определен, если это ограничение не выполняется.

Для строк значения сравниваются лексикографически. Никаких предположений о содержимом строк не делается, но следует помнить о возможности возникновения ситуации подстановки переменной/строки, описанной ранее. Сравнение строк - одна из самых распространенных ситуаций, в которых происходят такие неожиданные подстановки.

CMake также поддерживает проверку строки на соответствие регулярному выражению:

```
if(value MATCHES regex)
```

Значение снова следует правилам переменной-строки, определенным выше, и сравнивается с регулярным выражением regex. Если значение совпадает, то выражение принимает значение true. Хотя документация CMake не определяет поддерживаемый синтаксис регулярных выражений для команд if(), она определяет его для других команд (например, см. документацию по команде string()). По сути, CMake поддерживает только базовый синтаксис регулярных выражений.

Круглые скобки могут быть использованы для захвата части совпадающего значения. Команда устанавливает переменные с именами вида CMAKE_MATCH_<n>, где <n> - группа, в которую нужно попасть. Вся совпадающая строка сохраняется в группе 0.

```
if("Hi from ${who}" MATCHES "Hi from (Fred|Barney).*")
    message("${CMAKE_MATCH_1} says hello")
endif()
```

6.1.4. Тесты файловой системы

CMake также включает набор тестов, которые можно использовать для запроса файловой системы. Поддерживаются следующие выражения:

```
if(EXISTS pathToFileOrDir)
if(IS_DIRECTORY pathToDir)
if(IS_SYMLINK fileName)
if(IS_ABSOLUTE path)
if(file1 IS_NEWER_THAN file2)
```

Вышеизложенное в основном не требует пояснений, но есть некоторые моменты, о которых следует знать. В частности, оператор IS_NEWER_THAN возвращает true, если отсутствует один из файлов или если оба файла имеют одинаковую временную метку (что включает в себя, если один и тот же файл указан для файла1 и файла2). Таким образом, не будет необычным проверить существование файлов file1 и file2 перед выполнением фактического теста IS_NEWER_THAN, поскольку результат IS_NEWER_THAN в случае отсутствия одного из файлов часто будет не таким, как интуитивно ожидает разработчик. При использовании IS_NEWER_THAN также следует указывать полные пути, поскольку поведение относительных путей не вполне определено.

Еще один момент, который следует отметить, заключается в том, что, в отличие от большинства других выражений if, ни один из операторов файловой системы не выполняет подстановку переменной/строки без ${}, независимо от кавычек.

6.1.5. Тесты на существование

Последняя категория выражений if поддерживает проверку наличия или отсутствия различных элементов CMake. Они могут быть особенно полезны в больших, более сложных проектах, где некоторые части могут присутствовать или не присутствовать, или быть включенными.

```
if(DEFINED name)
if(COMMAND name)
if(POLICY name)
if(TARGET name)
if(TEST name)   # Available from CMake 3.4 onwards
```

Каждая из вышеперечисленных команд вернет true, если сущность с указанным именем существует в точке, где выдается команда if.

DEFINED

Возвращает true, если переменная с указанным именем существует. Значение переменной не имеет значения, проверяется только ее существование. Это также можно использовать для проверки того, определена ли конкретная переменная окружения:

```
if(DEFINED SOMEVAR)         # Checks for a CMake variable
if(DEFINED ENV{SOMEVAR})    # Checks for an environment variable
```

COMMAND

Проверяет, существует ли команда, функция или макрос CMake с указанным именем. Это наиболее полезно для проверки того, определено ли что-то, прежде чем пытаться это использовать. Для команд, предоставляемых CMake, лучше проверять версию CMake, но для функций и макросов, предоставляемых проектом (см. главу 8 "*Функции и макросы*"), проверка их существования с помощью теста COMMAND может оказаться полезной.

POLICY

Проверяет, известна ли частичная политика в CMake. Имена политик обычно имеют вид CMPxxxx, где часть xxxx всегда представляет собой четырехзначное число. Подробнее об этом см. в главе 12 "*Политики*".

TARGET

Возвращает true, если цель CMake с указанным именем была определена одной из команд add_executable(), add_library() или add_custom_target(). Цель могла быть определена в любом каталоге, если она известна на момент выполнения проверки if. Этот тест особенно полезен в сложных иерархиях проектов, которые втягивают другие внешние проекты, и где эти проекты могут иметь общие зависимые подпроекты (т.е. этот тип if-теста можно использовать для проверки, определена ли уже цель, прежде чем пытаться ее создать).

TEST

Возвращает true, если тест CMake с указанным именем был ранее определен командой add_test() (подробно рассматривается в главе 24, *Тестирование*).

Последний тест на существование доступен в CMake 3.5 и более поздних версиях:

```
if(value IN_LIST listVar)
```

Это выражение вернет true, если переменная listVar содержит указанное значение, где value следует обычным правилам переменной-or-string, но listVar должно быть именем переменной list.

6.1.6. Общие примеры

Несколько вариантов использования if() настолько распространены, что заслуживают особого упоминания. Многие из них в своей логике опираются на предопределенные переменные CMake, особенно переменные, относящиеся к компилятору и целевой платформе. К сожалению, часто встречаются выражения, основанные на неправильных переменных. Например, рассмотрим проект, в котором есть два исходных файла C++, один для сборки с компиляторами Visual Studio или совместимыми с ними (например, Intel), а другой для сборки со всеми остальными компиляторами. Такая логика часто реализуется следующим образом:

```
if(WIN32)
    set(platformImpl source_win.cpp)
else()
    set(platformImpl source_generic.cpp)
endif()
```

Хотя это, вероятно, подойдет для большинства проектов, на самом деле это не выражает правильное ограничение. Рассмотрим, например, проект, созданный под Windows, но использующий компилятор MinGW. Для таких случаев source_generic.cpp может быть более подходящим исходным файлом. Вышеизложенное можно было бы более точно реализовать следующим образом:

```
if(MSVC)
    set(platformImpl source_msvc.cpp)
else()
    set(platformImpl source_generic.cpp)
endif()
```

Другой пример связан с условным поведением, основанным на используемом генераторе CMake. В частности, CMake предлагает дополнительные возможности при сборке с помощью генератора Xcode, которые не поддерживаются другими генераторами. Иногда в проектах делается предположение, что сборка для macOS означает, что будет использоваться генератор Xcode, но это не обязательно так (и часто это не так). Иногда используется следующая неправильная логика:

```
if(APPLE)
    # Some Xcode-specific settings here...
else()
    # Things for other platforms here...
endif()
```

Опять же, может показаться, что это правильно, но если разработчик попытается использовать другой генератор (например, Ninja или Unix Makefiles) на macOS, логика не сработает. Проверка платформы с помощью выражения APPLE не выражает правильного условия, вместо этого следует проверить генератор CMake:

```
if(CMAKE_GENERATOR STREQUAL "Xcode")
    # Some Xcode-specific settings here...
else()
    # Things for other CMake generators here...
endif()
```

Оба приведенных выше примера являются случаями тестирования платформы вместо сущности, к которой фактически относится ограничение. Это понятно, поскольку платформа - одна из самых простых вещей для понимания и тестирования, но ее использование вместо более точного ограничения может неоправданно

ограничить выбор генераторов, доступный разработчикам, или привести к совершенно неправильному поведению.

Другим распространенным примером, на этот раз используемым уместно, является условное включение цели на основе того, была ли установлена определенная опция CMake или нет.

```
option(BUILD_MYLIB "Enable building the myLib target")
if(BUILD_MYLIB)
    add_library(myLib src1.cpp src2.cpp)
endif()
```

Более сложные проекты часто используют вышеописанную схему для условного включения подкаталогов или выполнения ряда других задач на основе опции CMake или переменной кэша. Разработчики могут включить/выключить эту опцию или установить переменную в значения не по умолчанию без необходимости редактировать непосредственно файл CMakeLists.txt. Это особенно полезно для скриптовых сборок, управляемых системами непрерывной интеграции и т.д., которые могут захотеть включить или отключить определенные части сборки.

6.2. Петля

Другой распространенной необходимостью во многих проектах CMake является выполнение некоторого действия над списком элементов или для диапазона значений. Также может потребоваться многократное выполнение некоторого действия до тех пор, пока не будет выполнено определенное условие. Эти потребности хорошо удовлетворяет CMake, предлагая традиционное поведение с некоторыми дополнениями, облегчающими работу с функциями CMake.

6.2.1. foreach()

CMake предоставляет команду foreach(), позволяющую проектам выполнять итерационный перебор набора элементов или значений. Существует несколько различных форм команды foreach(), самой простой из которых является следующая:

```
foreach(loopVar arg1 arg2 ...)
    # ...
endforeach()
```

В приведенной выше форме для каждого значения argN в качестве аргумента устанавливается loopVar и выполняется тело цикла. Никакой проверки переменных/строк не производится, аргументы используются точно так, как указаны их значения. Вместо явного перечисления каждого элемента, аргументы также могут быть заданы одной или несколькими переменными списка, используя более общую форму команды:

```
foreach(loopVar IN [LISTS listVar1 ...] [ITEMS item1 ...])
    # ...
endforeach()
```

В этой более общей форме отдельные аргументы по-прежнему могут быть указаны с помощью ключевого слова ITEMS, но ключевое слово LISTS позволяет указать одну или несколько переменных списка. При использовании этой более общей формы необходимо указать один или оба из ITEMS и/или LISTS. Если указаны оба, то после

LISTS должно стоять ITEMS. Допускается, чтобы переменные списка listVarN содержали пустой список. Пример поможет прояснить использование этой более общей формы.

```
set(list1 A B)
set(list2)
set(foo WillNotBeShown)
foreach(loopVar IN LISTS list1 list2 ITEMS foo bar)
    message("Iteration for: ${loopVar}")
endforeach()
```

Вывод из вышеприведенного будет следующим:

```
Iteration for: A
Iteration for: B
Iteration for: foo
Iteration for: bar
```

Команда foreach() также поддерживает более C-подобную итерацию по диапазону числовых значений:

```
foreach(loopVar RANGE start stop [step])
```

При использовании формы RANGE в foreach() цикл выполняется с loopVar, установленным на каждое значение в диапазоне от start до stop (включительно). Если указана опция step, то после каждой итерации это значение прибавляется к предыдущему, и цикл останавливается, когда его результат становится больше stop.

Форма RANGE также принимает только один аргумент, как показано ниже:

```
foreach(loopVar RANGE value)
```

Это эквивалентно foreach(loopVar RANGE 0 value), что означает, что тело цикла будет выполнено (value + 1) раз. Это неудачно, так как более интуитивно понятно, что тело цикла будет выполняться значение раз. По этой причине, вероятно, будет более понятным избегать использования этой второй формы RANGE и вместо этого явно указать значения start и stop.

Подобно ситуации с командами if() и endif(), в очень ранних версиях CMake (т.е. до 2.8.0) все формы команды foreach() требовали, чтобы loopVar также был указан в качестве аргумента для endforeach(). Опять же, это ухудшает читабельность и дает мало преимуществ, поэтому в новых проектах не рекомендуется указывать loopVar в endforeach().

6.2.2. while()

Другой командой цикла, предлагаемой CMake, является while():

```
while(condition)
    # ...
endwhile()
```

Условие проверяется, и если оно оценивается как true (по тем же правилам, что и выражение в операторах if()), то выполняется тело цикла. Это повторяется до тех пор, пока условие не станет ложным или цикл не будет

завершен досрочно (см. следующий раздел). Опять же, в версиях CMake до 2.8.0 условие должно было повторяться в команде endwhile(), но это больше не нужно и активно не рекомендуется для новых проектов.

6.2.3. Прерывание циклов

Циклы while() и foreach() поддерживают возможность досрочного выхода из цикла с помощью команды break() или перехода к началу следующей итерации с помощью continue(). Эти команды ведут себя так же, как и их аналогичные команды языка Си, и обе действуют только на самый внутренний цикл. Следующий пример иллюстрирует поведение.

```
foreach(outerVar IN ITEMS a b c)
    unset(s)
    foreach(innerVar IN ITEMS 1 2 3)
        # Stop inner loop once string s gets long
        list(APPEND s "${outerVar}${innerVar}")
        string(LENGTH s length)
        if(length GREATER 5)
            break()       ①
        endif()

        # Do no more processing if outer var is "b"
        if(outerVar STREQUAL "b")
            continue()    ②
        endif()
        message("Processing ${outerVar}-${innerVar}")
    endforeach()

    message("Accumulated list: ${s}")
endforeach()
```

① Завершает цикл innerVar for досрочно.

② Завершает текущую *итерацию* innerVar и переходит к следующему элементу innerVar.

Вывод из приведенного выше примера будет следующим:

```
Processing a-1
Processing a-2
Processing a-3
Accumulated list: a1;a2;a3
Accumulated list: b1;b2;b3
Processing c-1
Processing c-2
Processing c-3
Accumulated list: c1;c2;c3
```

6.3. Рекомендуемые практики

Минимизируйте возможности непреднамеренной интерпретации строк как переменных в командах if(), foreach() и while(). Избегайте унарных выражений с кавычками, вместо них лучше использовать операцию сравнения строк. Настоятельно рекомендуется установить минимальную версию CMake не ниже 3.1, чтобы отключить старое поведение, которое позволяло неявно преобразовывать значения строк в кавычках в имена переменных.

Когда требуется согласование регулярных выражений в командах if(xxx MATCHES regex) и переменные группового захвата, обычно рекомендуется как можно быстрее сохранить результаты согласования CMAKE_MATCH_<n> в обычных переменных. Эти переменные будут перезаписаны следующей командой, выполняющей любую операцию с регулярными выражениями.

Предпочтительнее использовать команды циклов, которые позволяют избежать двусмысленного или вводящего в заблуждение кода. При использовании формы RANGE команды foreach() всегда указывайте начальное и конечное значения. При итерации по элементам подумайте, не является ли использование форм IN LISTS или IN ITEMS более ясным для понимания того, что делается, а не простой формы foreach(loopVar item1 item2 ...).

Глава 7. Использование подкаталогов

Для простых проектов хранение всего в одном каталоге - это нормально, но большинство реальных проектов склонны разделять свои файлы по нескольким каталогам. Обычно различные типы файлов или отдельные модули группируются в собственных каталогах, или файлы, принадлежащие к логическим функциональным группам, находятся в собственной части иерархии каталогов проекта. Хотя структура каталогов может определяться тем, как разработчики представляют себе проект, способ структурирования проекта также влияет на систему сборки.

Две фундаментальные команды CMake в любом многодиректорном проекте - add_subdirectory() и include(). Эти команды добавляют содержимое из другого файла или каталога в сборку, позволяя распределить логику сборки по иерархии каталогов, а не заставляя все определять на самом верхнем уровне. Это дает ряд преимуществ:

- Логика сборки *локализована*, что означает, что характеристики сборки могут быть определены в каталоге, где они имеют наибольшее значение.
- Сборки могут состоять из подкомпонентов, которые определяются независимо от проекта верхнего уровня, потребляющего их. Это особенно важно, если проект использует такие вещи, как подмодули git или встраивает деревья исходных текстов сторонних разработчиков.
- Поскольку каталоги могут быть самодостаточными, становится относительно тривиальным включать или выключать части сборки, просто выбирая, добавлять или не добавлять в данный каталог.

add_subdirectory() и include() имеют совершенно разные характеристики, поэтому важно понимать сильные и слабые стороны обоих.

7.1. add_subdirectory()

Команда add_subdirectory() позволяет проекту включить в сборку еще один каталог. Этот каталог должен иметь свой собственный файл CMakeLists.txt, который будет обработан в момент вызова команды add_subdirectory(), и для него будет создан соответствующий каталог в дереве сборки проекта.

```
add_subdirectory(sourceDir [ binaryDir ] [ EXCLUDE_FROM_ALL ])
```

SourceDir не обязательно должен быть подкаталогом в дереве источника, хотя обычно это так. Можно добавить любой каталог, при этом sourceDir может быть указан как абсолютный или относительный путь, последний - относительно текущего каталога *источника*. Абсолютные пути обычно нужны только при добавлении каталогов, находящихся вне основного дерева источника.

Обычно binaryDir указывать не нужно. Если этот параметр опущен, CMake создает в дереве сборки каталог с тем же именем, что и sourceDir. Если sourceDir содержит какие-либо компоненты пути, они будут зеркально отражены в binaryDir, созданном CMake. В качестве альтернативы, binaryDir может быть явно указан как абсолютный или относительный путь, причем последний оценивается относительно текущего каталога *бинарных файлов* (подробнее об этом будет рассказано ниже). Если sourceDir - это путь вне дерева исходных текстов, CMake требует указания binaryDir, поскольку соответствующий относительный путь уже не может быть построен автоматически.

Необязательное ключевое слово EXCLUDE_FROM_ALL предназначено для управления тем, должны ли цели, определенные в добавляемом подкаталоге, по умолчанию включаться во ВСЕ цели проекта. К сожалению, для

некоторых версий CMake и генераторов проектов оно не всегда действует так, как ожидается, и даже может приводить к сбоям сборки.

7.1.1. Переменные исходного и двоичного каталогов

Иногда разработчику необходимо знать расположение каталога сборки, соответствующего текущему каталогу исходного текста, например, при копировании файлов, необходимых во время выполнения или для выполнения пользовательской задачи сборки. С помощью add_subdirectory() структура каталогов как исходного дерева, так и дерева сборки может быть произвольно сложной. Даже может быть несколько деревьев сборки, используемых с одним и тем же исходным деревом. Поэтому разработчику требуется помощь CMake для определения интересующих его каталогов. Для этого CMake предоставляет ряд переменных, которые отслеживают исходные и двоичные каталоги для обрабатываемого в данный момент файла CMakeLists.txt. Следующие переменные, доступные только для чтения, обновляются автоматически по мере обработки CMake каждого файла. Они всегда содержат абсолютные пути.

CMAKE_SOURCE_DIR

Самый верхний каталог дерева *исходных текстов* (т.е. где находится самый верхний файл CMakeLists.txt). Эта переменная никогда не меняет своего значения.

CMAKE_BINARY_DIR

Самый верхний каталог дерева *сборки*. Эта переменная никогда не меняет своего значения.

CMAKE_CURRENT_SOURCE_DIR

Каталог файла CMakeLists.txt, который в данный момент обрабатывается CMake. Он обновляется каждый раз, когда новый файл обрабатывается в результате вызова add_subdirectory(), и восстанавливается обратно, когда обработка этого каталога завершена.

CMAKE_CURRENT_BINARY_DIR

Каталог сборки, соответствующий файлу CMakeLists.txt, который в данный момент обрабатывается CMake. Он изменяется при каждом вызове add_subdirectory() и восстанавливается при возврате add_subdirectory().

Пример должен помочь продемонстрировать поведение:

Top level CMakeLists.txt

```
cmake_minimum_required(VERSION 3.0)
project(MyApp)

message("top:   CMAKE_SOURCE_DIR         = ${CMAKE_SOURCE_DIR}")
message("top:   CMAKE_BINARY_DIR         = ${CMAKE_BINARY_DIR}")
message("top:   CMAKE_CURRENT_SOURCE_DIR = ${CMAKE_CURRENT_SOURCE_DIR}")
message("top:   CMAKE_CURRENT_BINARY_DIR = ${CMAKE_CURRENT_BINARY_DIR}")

add_subdirectory(mysub)

message("top:   CMAKE_CURRENT_SOURCE_DIR = ${CMAKE_CURRENT_SOURCE_DIR}")
message("top:   CMAKE_CURRENT_BINARY_DIR = ${CMAKE_CURRENT_BINARY_DIR}")
```

mysub/CMakeLists.txt

```
message("mysub: CMAKE_SOURCE_DIR         = ${CMAKE_SOURCE_DIR}")
message("mysub: CMAKE_BINARY_DIR         = ${CMAKE_BINARY_DIR}")
message("mysub: CMAKE_CURRENT_SOURCE_DIR = ${CMAKE_CURRENT_SOURCE_DIR}")
message("mysub: CMAKE_CURRENT_BINARY_DIR = ${CMAKE_CURRENT_BINARY_DIR}")
```

Для приведенного выше примера, если файл CMakeLists.txt верхнего уровня находится в каталоге /somewhere/src, а каталог сборки - /somewhere/build, будет сгенерирован следующий вывод:

```
top:   CMAKE_SOURCE_DIR         = /somewhere/src
top:   CMAKE_BINARY_DIR         = /somewhere/build
top:   CMAKE_CURRENT_SOURCE_DIR = /somewhere/src
top:   CMAKE_CURRENT_BINARY_DIR = /somewhere/build
mysub: CMAKE_SOURCE_DIR         = /somewhere/src
mysub: CMAKE_BINARY_DIR         = /somewhere/build
mysub: CMAKE_CURRENT_SOURCE_DIR = /somewhere/src/mysub
mysub: CMAKE_CURRENT_BINARY_DIR = /somewhere/build/mysub
top:   CMAKE_CURRENT_SOURCE_DIR = /somewhere/src
top:   CMAKE_CURRENT_BINARY_DIR = /somewhere/build
```

7.1.2. Область применения

В главе 5, *Переменные,* было кратко упомянуто понятие *области видимости*. Одним из эффектов вызова add_subdirectory() является то, что CMake создает новую область видимости для обработки файла CMakeLists.txt этого каталога. Эта новая область видимости действует как дочерняя область видимости вызывающей области видимости, что влечет за собой ряд последствий:

- Все переменные, определенные в вызывающей области видимы для дочерней области, и дочерняя область может читать их значения как любую другую переменную.
- Любая новая переменная, созданная в дочерней области видимости, не будет видна вызывающей области видимости.
- Любое изменение переменной в дочерней области видимости является локальным для этой дочерней области видимости. Даже если эта переменная существовала в вызывающей области видимости, переменная вызывающей области видимости остается неизменной. Переменная, измененная в дочерней области видимости, действует как новая переменная, которая отбрасывается при выходе обработки из дочерней области видимости.

Другими словами, при входе в дочернюю область видимости она получает копию всех переменных, определенных в вызывающей области видимости в данный момент времени. Любые изменения переменных в дочерней области выполняются в копии дочерней области, оставляя переменные вызывающей области неизменными. Пример лучше всего иллюстрирует это поведение:

CMakeLists.txt

```
set(myVar foo)
message("Parent (before): myVar    = ${myVar}")
message("Parent (before): childVar = ${childVar}")

add_subdirectory(subdir)

message("Parent (after):  myVar    = ${myVar}")
message("Parent (after):  childVar = ${childVar}")
```

subdir/CMakeLists.txt

```
message("Child  (before): myVar    = ${myVar}")
message("Child  (before): childVar = ${childVar}")

set(myVar bar)
set(childVar fuzz)

message("Child  (after):  myVar    = ${myVar}")
message("Child  (after):  childVar = ${childVar}")
```

Это приводит к следующему результату:

```
Parent (before): myVar    = foo     ①
Parent (before): childVar =         ②
Child  (before): myVar    = foo     ③
Child  (before): childVar =         ④
Child  (after):  myVar    = bar     ⑤
Child  (after):  childVar = fuzz    ⑥
Parent (after):  myVar    = foo     ⑦
Parent (after):  childVar =         ⑧
```

①myVar определен на родительском уровне.

②childVar не определен на родительском уровне, поэтому он оценивается как пустая строка.

③myVar все еще виден в дочерней области видимости.

④childVar остается неопределенным в дочерней области видимости до его установки.

⑤myVar изменен в дочерней области видимости.

⑥ ChildVar был установлен в дочерней области видимости.

⑦ Когда обработка возвращается в родительскую область видимости, myVar по-прежнему имеет значение, полученное до вызова add_subdirectory(). Модификация myVar в дочерней области видимости не видна для родительской.

⑧childVar был определен в дочерней области видимости, поэтому он не виден родителю и оценивается как пустая строка.

Приведенное выше поведение диапазона переменных подчеркивает одну из важных характеристик add_subdirectory(). Она позволяет добавляемому каталогу изменять любые переменные по своему

усмотрению, не затрагивая переменные в вызывающей области видимости. Это помогает изолировать вызывающую область видимости от потенциально нежелательных изменений.

Однако бывают случаи, когда желательно, чтобы изменение переменной, сделанное в добавленном каталоге, было видно вызывающей стороне. Например, каталог может отвечать за сбор набора имен исходных файлов и передачу их обратно родителю в виде списка файлов. Это и есть цель ключевого слова PARENT_SCOPE в команде set(). Когда используется PARENT_SCOPE, устанавливаемой переменной является переменная в родительской области видимости, а не в текущей области видимости. Важно отметить, что это *не* означает установку переменной как в родительской, *так и в* текущей области видимости. Немного изменив предыдущий пример, эффект PARENT_SCOPE становится понятным:

CMakeLists.txt

```
set(myVar foo)
message("Parent (before): myVar = ${myVar}")
add_subdirectory(subdir)
message("Parent (after):  myVar = ${myVar}")
```

subdir/CMakeLists.txt

```
message("Child  (before): myVar = ${myVar}")
set(myVar bar PARENT_SCOPE)
message("Child  (after):  myVar = ${myVar}")
```

Это приводит к следующему результату:

```
Parent (before): myVar = foo
Child  (before): myVar = foo
Child  (after):  myVar = foo    ①
Parent (after):  myVar = bar    ②
```

① Вызов set() не влияет на myVar в дочерней области видимости, поскольку ключевое слово PARENT_SCOPE указывает CMake на изменение родительского myVar, а не локального.

② Родительский myVar был изменен вызовом set() в дочерней области видимости.

Поскольку использование PARENT_SCOPE предотвращает изменение командой любой локальной переменной с тем же именем, это может ввести в заблуждение, если локальная область видимости не использует то же имя переменной, что и родительская. В приведенном выше примере более понятным набором команд будет:

subdir/CMakeLists.txt

```
set(localVar bar)
set(myVar ${localVar} PARENT_SCOPE)
```

Очевидно, что приведенный выше пример является тривиальным, но в реальных проектах может быть много команд, которые вносят свой вклад в создание значения localVar, прежде чем окончательно установить родительскую переменную myVar.

Область видимости влияет не только на переменные, политики и некоторые свойства также имеют схожее с переменными поведение в этом отношении. В случае с политиками каждый вызов add_subdirectory() создает новую область видимости, в которой изменения политики могут быть сделаны без влияния на настройки

политики родителя. Аналогично, существуют свойства каталога, которые могут быть установлены в файле CMakeLists.txt дочернего каталога, что не повлияет на свойства родительского каталога. Оба этих вопроса более подробно рассматриваются в соответствующих главах: Глава 12 "*Политики*" и Глава 9 "*Свойства*".

7.2. include()

Другим методом, который CMake предоставляет для получения содержимого из других каталогов, является команда include(), которая имеет следующие две формы:

```
include(fileName [OPTIONAL] [RESULT_VARIABLE myVar] [NO_POLICY_SCOPE])
include(module   [OPTIONAL] [RESULT_VARIABLE myVar] [NO_POLICY_SCOPE])
```

Первая форма в некоторой степени аналогична add_subdirectory(), но есть ряд важных различий:

- include() ожидает имя файла для чтения, тогда как add_subdirectory() ожидает каталог и будет искать файл CMakeLists.txt в этом каталоге. Имя файла, передаваемого в include(), обычно имеет расширение .cmake, но может быть любым.

- include() не вводит новую область действия переменной, в то время как add_subdirectory() вводит.

- Обе команды по умолчанию вводят новую область действия политики, но в команде include() можно указать не делать этого с помощью опции NO_POLICY_SCOPE (у add_subdirectory() такой опции нет). Более подробную информацию об обработке области действия политики см. в главе 12, Политика.

- Значение переменных CMAKE_CURRENT_SOURCE_DIR и CMAKE_CURRENT_BINARY_DIR не меняется при обработке файла, названного include(), тогда как для add_subdirectory() они меняются. Это будет рассмотрено более подробно в ближайшее время.

Вторая форма команды include() служит совершенно другой цели. Она используется для загрузки именованного модуля, что подробно рассматривается в главе 11, Модули. Все пункты, кроме первого, справедливы и для этой второй формы.

Поскольку значение CMAKE_CURRENT_SOURCE_DIR не меняется при вызове include(), может показаться, что включаемому файлу трудно определить каталог, в котором он находится. CMAKE_CURRENT_SOURCE_DIR будет содержать расположение файла, из которого был вызван include(), а не каталог, содержащий включаемый файл. Кроме того, в отличие от add_subdirectory(), где fileName всегда будет CMakeLists.txt, при использовании include() имя файла может быть любым, поэтому включаемому файлу может быть трудно определить свое имя. Для решения подобных ситуаций CMake предоставляет дополнительный набор переменных:

CMAKE_CURRENT_LIST_DIR

> Аналогична CMAKE_CURRENT_SOURCE_DIR, за исключением того, что она будет обновляться при обработке включенного файла. Это переменная, которую следует использовать, когда требуется каталог текущего обрабатываемого файла, независимо от того, как он был добавлен в сборку. Она всегда будет содержать абсолютный путь.

CMAKE_CURRENT_LIST_FILE

> Всегда указывает имя файла, который обрабатывается в данный момент. Он всегда содержит абсолютный путь к файлу, а не только имя файла.

CMAKE_CURRENT_LIST_LINE

> Удерживает номер строки обрабатываемого в данный момент файла. Эта переменная нужна редко, но может оказаться полезной в некоторых сценариях отладки.

Важно отметить, что вышеуказанные три переменные работают для *любого* файла, обрабатываемого CMake, а не только для тех, которые подтягиваются командой include(). Они имеют те же значения, что описаны выше, даже для файла CMakeLists.txt, подтягиваемого командой add_subdirectory(), в этом случае CMAKE_CURRENT_LIST_DIR будет иметь то же значение, что и CMAKE_CURRENT_SOURCE_DIR. Следующий пример демонстрирует такое поведение:

CMakeLists.txt

```
add_subdirectory(subdir)
message("====")
include(subdir/CMakeLists.txt)
```

subdir/CMakeLists.txt

```
message("CMAKE_CURRENT_SOURCE_DIR = ${CMAKE_CURRENT_SOURCE_DIR}")
message("CMAKE_CURRENT_BINARY_DIR = ${CMAKE_CURRENT_BINARY_DIR}")
message("CMAKE_CURRENT_LIST_DIR   = ${CMAKE_CURRENT_LIST_DIR}")
message("CMAKE_CURRENT_LIST_FILE  = ${CMAKE_CURRENT_LIST_FILE}")
message("CMAKE_CURRENT_LIST_LINE  = ${CMAKE_CURRENT_LIST_LINE}")
```

результат, подобный следующему:

```
CMAKE_CURRENT_SOURCE_DIR = /somewhere/src/foo
CMAKE_CURRENT_BINARY_DIR = /somewhere/build/foo
CMAKE_CURRENT_LIST_DIR   = /somewhere/src/foo
CMAKE_CURRENT_LIST_FILE  = /somewhere/src/foo/CMakeLists.txt
CMAKE_CURRENT_LIST_LINE  = 5
====
CMAKE_CURRENT_SOURCE_DIR = /somewhere/src
CMAKE_CURRENT_BINARY_DIR = /somewhere/build
CMAKE_CURRENT_LIST_DIR   = /somewhere/src/foo
CMAKE_CURRENT_LIST_FILE  = /somewhere/src/foo/CMakeLists.txt
CMAKE_CURRENT_LIST_LINE  = 5
```

Приведенный выше пример также подчеркивает еще одну интересную особенность команды include(). Ее можно использовать для включения содержимого файла, который уже был включен в сборку ранее. Если разные подкаталоги большого сложного проекта хотят использовать код CMake в некотором файле в общей области проекта, они могут включить() этот файл независимо друг от друга.

7.3. Досрочное завершение обработки

Бывают случаи, когда проект может захотеть прекратить обработку оставшейся части текущего файла и вернуть управление вызывающей стороне. Команда return() может быть использована именно для этой цели, но обратите внимание, что она не может вернуть значение вызывающей стороне. Ее единственным эффектом является завершение обработки текущей области видимости. Если return() не вызывается изнутри функции, она завершает обработку текущего файла, независимо от того, был ли он добавлен с помощью include() или add_subdirectory(). Эффект вызова return() внутри функции рассматривается в разделе 8.4, "Область видимости", включая особое внимание к распространенной ошибке, которая может привести к непреднамеренному возврату из текущего файла.

Как отмечалось в предыдущем разделе, разные части проекта могут включать один и тот же файл из нескольких мест. Иногда бывает желательно проверить это и включить файл только один раз, возвращаясь раньше для последующих включений, чтобы предотвратить многократную обработку файла. Это очень похоже на ситуацию с заголовками в C/C++, и довольно часто можно увидеть использование подобной формы защиты включения:

```
if(DEFINED cool_stuff_include_guard)
    return()
endif()

set(cool_stuff_include_guard 1)
# ...
```

В CMake 3.10 или более поздней версии это можно выразить более кратко и надежно с помощью специальной команды, поведение которой аналогично #pragma once в C/C++:

```
include_guard()
```

По сравнению с ручным написанием кода if-endif, этот способ более надежен, поскольку он обрабатывает имя защитной переменной внутри. Команда также принимает необязательный ключевой аргумент DIRECTORY или GLOBAL для указания другой области видимости, в которой следует проверять, не обрабатывался ли файл ранее, но эти ключевые слова вряд ли понадобятся в большинстве ситуаций. Если ни один из аргументов не указан, то предполагается переменная область видимости, и эффект в точности эквивалентен приведенному выше коду if-endif. GLOBAL гарантирует, что команда завершит обработку файла, если он был обработан до этого где-либо еще в проекте (т.е. область видимости переменной игнорируется). DIRECTORY проверяет наличие предыдущей обработки только в пределах текущей области каталогов и ниже.

7.4. Рекомендуемые практики

Выбор между использованием add_subdirectory() и include() для включения другого каталога в сборку не всегда очевиден. С одной стороны, add_subdirectory() проще и лучше справляется с сохранением относительной самодостаточности каталогов, поскольку создает собственную область видимости. С другой стороны, некоторые команды CMake имеют ограничения, которые позволяют им работать только с вещами, определенными в области видимости текущего файла, поэтому include() работает лучше в таких случаях. В разделе 28.5.1, "Целевые источники", обсуждаются некоторые аспекты этой темы.

В качестве общего руководства, для большинства простых проектов, вероятно, лучше использовать add_subdirectory() вместо include(). Это способствует более чистому определению проекта и позволяет в CMakeLists.txt для данного каталога больше сосредоточиться только на том, что этот каталог должен определять. По мере развития проекта он может начать использовать include() для некоторых директорий, где это имеет смысл. Следование этой стратегии будет способствовать лучшей локальности информации во всем проекте, а также будет иметь тенденцию вводить сложность только там, где это необходимо и где это приносит полезные преимущества. Не то чтобы include() сам по себе был сложнее, чем add_subdirectory(), но использование include() приводит к тому, что пути к файлам должны быть прописаны более явно, поскольку CMake считает текущим исходным каталогом не тот каталог, в который включен файл. Также ведутся работы по устранению некоторых ограничений, связанных с вызовом некоторых команд из разных каталогов, поэтому add_subdirectory(), вероятно, станет более гибким и более предпочтительным из этих двух методов.

Независимо от того, используется ли функция add_subdirectory(), include() или их комбинация, переменная CMAKE_CURRENT_LIST_DIR обычно является лучшим выбором, чем

CMAKE_CURRENT_SOURCE_DIR. Выработав привычку использовать CMAKE_CURRENT_LIST_DIR на ранней стадии, гораздо легче переключаться между add_subdirectory() и include() по мере роста сложности проекта и перемещать целые каталоги для реструктуризации проекта.

Если для проекта требуется CMake 3.10 или более поздняя версия, предпочтите использовать команду include_guard() без аргументов вместо явного блока if-endif в случаях, когда необходимо предотвратить многократное включение файла.

Глава 8. Функции и макросы

Оглядываясь на материал, рассмотренный в этой книге до сих пор, можно сказать, что синтаксис CMake уже начинает напоминать язык программирования. Он поддерживает переменные, логику if-then-else, циклы и включение других файлов для обработки. Неудивительно, что CMake также поддерживает такие распространенные в программировании понятия, как функции и макросы. Подобно их роли в других языках программирования, функции и макросы являются для проектов и разработчиков основным механизмом расширения функциональности CMake и инкапсуляции повторяющихся задач естественным образом. Они позволяют разработчику определять многократно используемые блоки кода CMake, которые можно вызывать так же, как обычные встроенные команды CMake. Они также являются краеугольным камнем собственной системы модулей CMake (рассматривается отдельно в главе 11, *Модули*).

8.1. Основы

Функции и макросы в CMake имеют очень схожие характеристики с их одноименными аналогами в C/C++. Функции вводят новую область видимости, а аргументы функции становятся переменными, доступными внутри тела функции. Макросы, с другой стороны, эффективно вставляют свое тело в точку вызова, а аргументы макроса заменяются простыми строковыми заменами. Это поведение отражает принцип работы функций и макросов #define в C/C++. Функция или макрос CMake определяется следующим образом:

```
function(name [arg1 [arg2 [...]]])
   # Function body (i.e. commands) ...
endfunction()

macro(name [arg1 [arg2 [...]]])
   # Macro body (i.e. commands) ...
endmacro()
```

После определения функция или макрос вызывается точно так же, как и любая другая команда CMake. Затем тело функции или макроса выполняется в месте вызова. Например:

```
function(print_me)
    message("Hello from inside a function")
    message("All done")
endfunction()

# Called like so:
print_me()
```

Как показано выше, аргумент name определяет имя, используемое для вызова функции или макроса, и оно должно содержать только буквы, цифры и знаки подчеркивания. Имя будет обрабатываться без учета регистра, поэтому соглашения о верхнем/нижнем регистре - это скорее вопрос стиля (в документации CMake принято, что имена команд все строчные, а слова разделяются подчеркиванием). Очень ранние версии CMake требовали повторения имени в качестве аргумента endfunction() или endmacro(), но новые проекты должны избегать этого, поскольку это только добавляет ненужный беспорядок.

8.2. Сущности обработки аргументов

Работа с аргументами в функциях и макросах одинакова, за исключением одного очень важного различия. Для функций каждый аргумент является переменной CMake и обладает всеми обычными свойствами переменной

CMake. Например, их можно проверять в операторах if() как переменные. В отличие от них, аргументы макросов являются заменителями строк, поэтому все, что было использовано в качестве аргумента для вызова макроса, по сути, вставляется туда, где этот аргумент появляется в теле макроса. Если аргумент макроса используется в операторе if(), он будет рассматриваться как строка, а не как переменная. Следующий пример и его результат демонстрируют разницу:

```
function(func arg)
    if(DEFINED arg)
        message("Function arg is a defined variable")
    else()
        message("Function arg is NOT a defined variable")
    endif()
endfunction()

macro(macr arg)
    if(DEFINED arg)
        message("Macro arg is a defined variable")
    else()
        message("Macro arg is NOT a defined variable")
    endif()
endmacro()

func(foobar)
macr(foobar)
```

```
Function arg is a defined variable
Macro arg is NOT a defined variable
```

Помимо этого различия, функции и макросы поддерживают одни и те же функции, когда речь идет об обработке аргументов. Каждый аргумент в определении функции служит чувствительной к регистру меткой для аргумента, который он представляет. Для функций эта метка действует как переменная, а для макросов - как подстановка строки. К значению этого аргумента можно обратиться в теле функции или макроса, используя обычную нотацию переменной, хотя аргументы макроса технически не являются переменными.

```
function(func myArg)
    message("myArg = ${myArg}")
endfunction()

macro(macr myArg)
    message("myArg = ${myArg}")
endmacro()

func(foobar)
macr(foobar)
```

И вызов func(), и вызов macr() выводят одно и то же:

```
myArg = foobar
```

В дополнение к именованным аргументам функции и макросы поставляются с набором автоматически определяемых переменных, которые позволяют обрабатывать аргументы в дополнение к именованным или вместо них:

ARGC

Это значение будет равно общему количеству аргументов, переданных функции. Считаются именованные аргументы плюс любые дополнительные неименованные аргументы, которые были переданы.

ARGV

Это списковая переменная, содержащая все аргументы, переданные функции, включая как именованные аргументы, так и любые дополнительные неименованные аргументы, которые были переданы.

ARGN

Как ARGV, за исключением того, что здесь содержатся только аргументы помимо именованных (т.е. необязательные, неименованные аргументы).

В дополнение к вышесказанному, на каждый отдельный аргумент можно ссылаться с именем вида ARG#, где # - номер аргумента (например, ARG1, ARG2 и т.д.). Это относится и к именованным аргументам, поэтому на первый именованный аргумент можно также ссылаться через ARG1 и т.д.

Типичные ситуации, в которых используются переменные ARG..., включают поддержку необязательных аргументов и реализацию команды, которая может принимать произвольное количество элементов для обработки. Рассмотрим функцию, которая определяет исполняемую цель, связывает эту цель с некоторой библиотекой и определяет для нее тестовый пример. Такая функция часто встречается при написании тестовых примеров (эта тема рассматривается в главе 24, *Тестирование*). Вместо того чтобы повторять шаги для каждого тестового случая, функция позволяет определить шаги один раз, после чего каждый тестовый случай становится простым определением в одну строку.

```
# Use a named argument for the target and treat all remaining
# (unnamed) arguments as the source files for the test case
function(add_mytest targetName)
    add_executable(${targetName} ${ARGN})

    target_link_libraries(${targetName} PRIVATE foobar)

    add_test(NAME    ${targetName}
             COMMAND ${targetName}
    )
endfunction()

# Define some test cases using the above function
add_mytest(smallTest small.cpp)
add_mytest(bigTest   big.cpp algo.cpp net.cpp)
```

Приведенный выше пример демонстрирует полезность переменной ARGN. Она позволяет функции или макросу принимать переменное количество аргументов, но при этом указывать набор именованных аргументов, которые должны быть предоставлены. Однако есть особый случай, который может привести к неожиданному поведению. Поскольку макросы рассматривают свои аргументы как подстановки строк, а не как переменные, если они используют ARGN в том месте, где ожидается имя переменной, то переменная, на которую они ссылаются, будет находиться в области видимости, из которой вызывается макрос, а не ARGN из собственных аргументов макроса. В следующем примере показана такая ситуация:

```
# WARNING: This macro is misleading
macro(dangerous)
    # Which ARGN?
    foreach(arg IN LISTS ARGN)
        message("Argument: ${arg}")
    endforeach()
endmacro()

function(func)
    dangerous(1 2)
endfunction()

func(3)
```

Вывод из вышеприведенного будет следующим:

```
Argument: 3
```

При использовании ключевого слова LISTS с функцией foreach() необходимо указать имя переменной, но ARGN, предоставленный для макроса, не является именем переменной. Когда макрос вызывается из другой функции, макрос в итоге использует *переменную* ARGN из этой функции, а не ARGN из самого макроса. Ситуация проясняется, если вставить содержимое тела макроса непосредственно в функцию, в которой он вызывается (что, собственно, и будет делать CMake):

```
function(func)
    # Now it is clear, ARGN here will use the arguments from func
    foreach(arg IN LISTS ARGN)
        message("Argument: ${arg}")
    endforeach()
endfunction()
```

В таких случаях следует сделать макрос функцией, а если он должен оставаться макросом, то не рассматривать аргументы как переменные. В приведенном выше примере реализация dangerous() может быть изменена на использование foreach(arg IN ITEMS ${ARGN}).

8.3. Аргументы ключевых слов

В предыдущем разделе было показано, как переменные ARG... можно использовать для обработки переменного набора аргументов. Этой функциональности достаточно для простого случая, когда требуется только один набор переменных или необязательных аргументов, но если необходимо поддерживать несколько необязательных или переменных наборов аргументов, обработка становится довольно утомительной. Кроме того, описанная выше базовая обработка аргументов является довольно жесткой по сравнению со многими встроенными командами CMake, которые поддерживают аргументы на основе ключевых слов и гибкое упорядочивание аргументов. Рассмотрим команду target_link_libraries():

```
target_link_libraries(targetName
     <PRIVATE|PUBLIC|INTERFACE> item1 [item2 ...]
    [<PRIVATE|PUBLIC|INTERFACE> item3 [item4 ...]]
    ...
)
```

В качестве первого аргумента требуется имя targetName, но после этого вызывающая сторона может предоставить любое количество секций PRIVATE, PUBLIC или INTERFACE в любом порядке, причем каждая секция может содержать любое количество элементов. Определяемые пользователем функции и макросы могут поддерживать такую же гибкость, используя команду cmake_parse_arguments():

```
include(CMakeParseArguments)  # Needed only for CMake 3.4 and earlier
cmake_parse_arguments(prefix
                      noValueKeywords
                      singleValueKeywords
                      multiValueKeywords
                      argsToParse)
```

Команда cmake_parse_arguments() раньше предоставлялась модулем CMakeParseArguments, но в CMake 3.5 она стала встроенной командой. Строка include(CMakeParseArguments) ничего не сделает в CMake 3.5 и более поздних версиях, в то время как для более ранних версий CMake она определит команду cmake_parse_arguments() (подробнее об использовании include() см. в главе 11, *Модули*). Приведенная выше форма обеспечивает доступность команды независимо от используемой версии CMake.

cmake_parse_arguments() принимает аргументы, переданные в качестве параметра argsToParse, и обрабатывает их в соответствии с заданными наборами ключевых слов. Обычно argsToParse задается в виде ${ARGN}, который представляет собой набор неименованных аргументов, передаваемых вложенной функции или макросу. Каждый из аргументов с ключевыми словами представляет собой список имен ключевых слов, поддерживаемых данной функцией или макросом, поэтому их следует заключить в кавычки, чтобы они были правильно разобраны.

Ключевые слова noValueKeywords определяют отдельные аргументы ключевых слов, которые действуют как булевы переключатели. Наличие ключевого слова указывает на одно, а его отсутствие - на другое. Однозначные ключевые слова требуют ровно одного дополнительного аргумента после ключевого слова при их использовании, в то время как многозначные ключевые слова требуют ноль или более дополнительных аргументов после ключевого слова. Хотя это и не обязательно, преобладает правило, что ключевые слова должны быть написаны в верхнем регистре, а слова разделяются подчеркиванием, если это необходимо. Обратите внимание, однако, что ключевые слова не должны быть слишком длинными, иначе они могут быть громоздкими в использовании.

Когда cmake_parse_arguments() вернется, для каждого ключевого слова будет доступна соответствующая переменная, имя которой состоит из указанного префикса, символа подчеркивания и имени ключевого слова. Например, с префиксом ARG переменная, соответствующая ключевому слову FOO, будет ARG_FOO. Если в argsToParse нет определенного ключевого слова, то соответствующая ему переменная будет пустой. Пример лучше всего иллюстрирует, как определяются и обрабатываются три различных типа ключевых слов:

```
function(func)
    # Define the supported set of keywords
    set(prefix       ARG)
    set(noValues     ENABLE_NET COOL_STUFF)
    set(singleValues TARGET)
    set(multiValues  SOURCES IMAGES)

    # Process the arguments passed in
    include(CMakeParseArguments)
    cmake_parse_arguments(${prefix}
                          "${noValues}"
                          "${singleValues}"
                          "${multiValues}"
                          ${ARGN})

    # Log details for each supported keyword
    message("Option summary:")
    foreach(arg IN LISTS noValues)
        if(${${prefix}_${arg}})
            message("  ${arg} enabled")
        else()
            message("  ${arg} disabled")
        endif()
    endforeach()

    foreach(arg IN LISTS singleValues multiValues)
        # Single argument values will print as a simple string
        # Multiple argument values will print as a list
        message("  ${arg} = ${${prefix}_${arg}}")
    endforeach()
endfunction()

# Examples of calling with different combinations
# of keyword arguments
func(SOURCES foo.cpp bar.cpp TARGET myApp ENABLE_NET)
func(COOL_STUFF TARGET dummy IMAGES here.png there.png gone.png)
```

Соответствующий вывод будет выглядеть следующим образом:

```
Option summary:
  ENABLE_NET enabled
  COOL_STUFF disabled
  TARGET = myApp
  SOURCES = foo.cpp;bar.cpp
  IMAGES =
Option summary:
  ENABLE_NET disabled
  COOL_STUFF enabled
  TARGET = dummy
  SOURCES =
  IMAGES = here.png;there.png;gone.png
```

По сравнению с базовой обработкой аргументов с помощью именованных аргументов и/или переменных ARG..., преимущества cmake_parse_arguments() многочисленны.

- Будучи основанным на ключевых словах, сайт вызова имеет улучшенную читабельность, поскольку аргументы, по сути, становятся самодокументирующимися. Другим разработчикам, читающим сайт вызова, обычно не нужно обращаться к реализации функции или ее документации, чтобы понять, что означает каждый из аргументов.
- Вызывающая сторона может выбирать порядок передачи аргументов.
- Вызывающая сторона может просто опустить те аргументы, которые не нужно предоставлять.
- Поскольку каждое из поддерживаемых ключевых слов должно быть передано в функцию cmake_parse_arguments(), и она обычно вызывается в верхней части функции, обычно очень ясно, какие аргументы поддерживает функция.
- Поскольку разбор аргументов, основанных на ключевых словах, выполняется командой cmake_parse_arguments(), а не специальным парсером, написанным вручную, ошибки разбора аргументов практически исключены.

8.4. Область применения

Фундаментальное различие между функциями и макросами заключается в том, что функции вводят новую область видимости, в то время как макросы этого не делают. Переменные, определенные или измененные внутри функции, не влияют на одноименные переменные вне функции. Любые изменения политики, сделанные внутри функции, также локальны для функции, поэтому настройки политики вызывающего пользователя не затрагиваются. Функция, по сути, является своей собственной автономной песочницей, в отличие от макроса, который имеет полный доступ к переменным и политикам вызывающей стороны, что означает, что макрос имеет возможность изменять окружение вызывающей стороны так, как не может функция.

В отличие от своих аналогов на C/C++, функции и макросы CMake не поддерживают возврат значения напрямую. Более того, поскольку функции вводят свою собственную область видимости, может показаться, что нет простого способа передать информацию обратно вызывающей стороне, но это не так. Тот же подход, что обсуждался для add_subdirectory() в разделе 7.1.2 "Область видимости", можно использовать и для функций. Ключевое слово PARENT_SCOPE команды set() можно использовать для изменения переменной в области видимости вызывающей функции, а не локальной переменной внутри функции. Хотя это не то же самое, что возврат значения из функции, это позволяет передать значение (или несколько значений) обратно вызывающей функции.

Распространенный подход заключается в том, чтобы разрешить передавать имя переменной в качестве аргумента функции, чтобы вызывающая сторона по-прежнему контролировала имена переменных, в которых устанавливаются результаты работы функции. Именно такой подход используется в cmake_parse_arguments(), причем ее аргумент prefix определяет префикс всех имен переменных, которые она устанавливает в области видимости вызывающей стороны. Следующий пример демонстрирует, как реализовать эту технику:

```
function(func resultVar1 resultVar2)
    set(${resultVar1} "First result" PARENT_SCOPE)
    set(${resultVar2} "Second result" PARENT_SCOPE)
endfunction()

func(myVar otherVar)
message("myVar: ${myVar}")
message("otherVar: ${otherVar}")
```

Выходными данными будут:

```
myVar: First result
otherVar: Second result
```

Другая альтернатива заключается в том, чтобы функция документировала переменные, которые она устанавливает, а не позволяла вызывающей стороне указывать имена переменных. Это менее желательно, поскольку снижает гибкость функции и открывает возможности для столкновения имен переменных. По возможности лучше использовать описанный выше метод, чтобы предоставить вызывающей функции контроль над устанавливаемыми или изменяемыми именами переменных.

С макросами можно работать так же, как и с функциями, указывая имена переменных, которые нужно установить, передавая их в качестве аргументов. Единственное отличие заключается в том, что ключевое слово PARENT_SCOPE не должно использоваться внутри макроса, поскольку он уже изменяет переменные в области видимости вызывающей стороны. Фактически, единственная причина, по которой можно использовать макрос вместо функции, это если в вызывающей области видимости нужно установить много переменных. Макрос будет влиять на вызывающую область видимости при каждом вызове set(), в то время как функция влияет на область видимости только тогда, когда PARENT_SCOPE явно передается функции set().

В разделе 7.3, "Досрочное завершение обработки", оператор return() обсуждался как способ досрочного завершения обработки в файле или функции. Как объяснялось выше, return() не возвращает значение, а только возвращает обработку в родительскую *область видимости*. Если return() вызывается внутри функции, обработка немедленно возвращается вызывающей стороне, т.е. остальная часть функции пропускается. Поведение return() внутри макроса, с другой стороны, совершенно иное. Поскольку макрос не вводит новую область видимости, поведение оператора return() зависит от места вызова макроса. Вспомните, что макрос эффективно вставляет свои команды в место вызова. Таким образом, любой оператор return() из макроса на самом деле возвращается из области видимости того, кто вызвал макрос, а не из самого макроса. Рассмотрим следующий пример:

```
macro(inner)
    message("From inner")
    return()  # Usually dangerous within a macro
    message("Never printed")
endmacro()

function(outer)
    message("From outer before calling inner")
    inner()
    message("Also never printed")
endfunction()

outer()
```

Вывод из вышеприведенного будет следующим:

```
From outer before calling inner
From inner
```

Чтобы показать, почему второе сообщение в теле функции никогда не печатается, вставьте содержимое тела макроса в то место, где он вызывается:

```
function(outer)
    message("From outer before calling inner")

    # === Pasted macro body ===
    message("From inner")
    return()
    message("Never printed")
    # === End of macro body ===

    message("Also never printed")
endfunction()

outer()
```

Теперь стало намного понятнее, почему оператор return() заставляет обработку покинуть функцию, даже если она изначально была вызвана изнутри макроса. Это подчеркивает опасность использования return() внутри макросов. Поскольку макросы не создают собственную область видимости, результат применения оператора return() часто оказывается не таким, как ожидалось.

8.5. Переопределение команд

Когда функция() или макрос() вызывается для определения новой команды, если команда с таким именем уже существует, недокументированное поведение CMake заключается в том, чтобы сделать старую команду доступной с тем же именем, но с добавлением подчеркивания. Это касается как встроенной команды, так и пользовательской функции или макроса. Разработчики, знающие об этом поведении, иногда испытывают искушение воспользоваться им, чтобы попытаться создать обертку вокруг существующей команды, например, так:

```
function(someFunc)
    # Do something...
endfunction()

# Later in the project...
function(someFunc)
    if(...)
        # Override the behavior with something else...
    else()
        # WARNING: Intended to call the original command, but it is not safe
        _someFunc()
    endif()
endfunction()
```

Если команда переопределяется таким образом только один раз, то она работает, но если она переопределяется снова, то исходная команда больше недоступна. Добавление одного подчеркивания для "сохранения" предыдущей команды применяется только к текущему имени, оно не применяется рекурсивно ко всем предыдущим переопределениям. Это может привести к бесконечной рекурсии, что демонстрирует следующий надуманный пример:

```
function(printme)
    message("Hello from first")
endfunction()

function(printme)
    message("Hello from second")
    _printme()
endfunction()

function(printme)
    message("Hello from third")
    _printme()
endfunction()

printme()
```

Наивно было бы ожидать, что результат будет следующим:

```
Hello from third
Hello from second
Hello from first
```

Но вместо этого первая реализация никогда не вызывается, потому что вторая в итоге вызывает саму себя в бесконечном цикле. Когда CMake обрабатывает вышеописанное, происходит вот что:

1. Первая реализация printme создается и становится доступной в виде команды с таким именем. Ранее команды с таким именем не существовало, поэтому дальнейших действий не требуется.
2. Встречается вторая реализация printme. CMake находит существующую команду с таким именем, поэтому он определяет имя _printme для указания на старую команду и устанавливает printme для указания на новое определение.
3. Встречается третья реализация printme. И снова CMake находит существующую команду с таким именем, поэтому он *переопределяет* имя _printme для указания на старую команду (которая является второй реализацией) и устанавливает printme для указания на новое определение.

Когда вызывается printme(), выполнение переходит к третьей реализации, которая вызывает _printme(). Она переходит во вторую реализацию, которая также вызывает _printme(), но _printme() снова указывает на вторую реализацию, и получается бесконечная рекурсия. Выполнение так и не достигает первой реализации.

В целом, можно переопределить функцию или макрос, если он не пытается вызвать предыдущую реализацию, как в приведенном выше обсуждении. Проекты должны просто считать, что новая реализация заменяет старую, а старая считается недоступной.

8.6. Рекомендуемые практики

Функции и макросы - отличный способ повторного использования одного и того же фрагмента кода CMake в рамках всего проекта. В целом, лучше использовать функции, а не макросы, поскольку использование новой области видимости в функции лучше изолирует воздействие функции на вызывающую область видимости. Макросы следует использовать только в тех случаях, когда содержимое тела макроса действительно должно быть выполнено в области видимости вызывающей функции. Такие ситуации должны быть относительно редкими. Чтобы избежать неожиданного поведения, также избегайте вызова return() внутри макроса.

Для всех функций и макросов, кроме самых тривиальных, настоятельно рекомендуется использовать обработку аргументов на основе ключевых слов, предоставляемую cmake_parse_arguments(). Это повышает удобство использования и надежность вызывающего кода (например, уменьшается вероятность перепутать аргументы). Это также позволяет легче расширять функцию в будущем, поскольку нет необходимости в упорядочивании аргументов или в том, чтобы все аргументы всегда были предоставлены, даже если они не относятся к делу.

Вместо того чтобы распределять функции и макросы по всему дереву исходных текстов, общепринятой практикой является назначение определенного каталога (обычно чуть ниже верхнего уровня проекта), где могут быть собраны различные файлы XXX.cmake. Этот каталог действует как каталог готовых к использованию функций, доступ к которым можно получить из любой точки проекта. Каждый из файлов может предоставлять функции, макросы, переменные и другие возможности по мере необходимости. Использование суффикса имени файла .cmake позволяет команде include() находить файлы как модули, что подробно рассматривается в главе 11, *Модули*. Он также позволяет инструментам IDE распознавать тип файла и применять подсветку синтаксиса CMake.

Не определяйте и не вызывайте функцию или макрос с именем, которое начинается с одного символа подчеркивания. В частности, не полагайтесь на недокументированное поведение, при котором старая реализация команды становится доступной под таким именем, когда функция или макрос переопределяет существующую команду. Если команда была переопределена более одного раза, ее первоначальная реализация больше недоступна. Это недокументированное поведение может быть даже удалено в будущей версии CMake, поэтому его не следует использовать. Аналогично, не переопределяйте ни одну встроенную команду CMake, считайте их недоступными, чтобы проекты всегда могли считать, что встроенные команды ведут себя в соответствии с официальной документацией, и чтобы не было возможности сделать исходную команду недоступной.

Глава 9. Свойства

Свойства влияют практически на все аспекты процесса сборки, начиная с того, как исходный файл компилируется в объектный файл, и заканчивая местом установки собранных двоичных файлов в пакетной программе установки. Они всегда привязаны к определенному объекту, будь то каталог, цель, исходный файл, тестовый пример, переменная кэша или даже сам процесс сборки в целом. Вместо того чтобы хранить отдельное значение, как это делает переменная, свойство предоставляет информацию, специфичную для объекта, к которому оно присоединено.

Для новичков в CMake свойства иногда путают с переменными. Хотя поначалу оба эти понятия могут показаться похожими по функциям и возможностям, свойства служат совершенно разным целям. Переменная не привязана к какому-либо конкретному объекту, и очень часто проекты определяют и используют свои собственные переменные. Сравните это со свойствами, которые обычно хорошо определены и документированы в CMake и всегда применяются к определенной сущности. Вероятно, путаница между этими двумя понятиями связана с тем, что значение свойства по умолчанию иногда задается переменной. Имена, которые CMake использует для связанных свойств и переменных, обычно следуют одной и той же схеме: имя переменной - это имя свойства с добавлением CMAKE_.

9.1. Общие команды свойств

CMake предоставляет ряд команд для работы со свойствами. Наиболее общие из них, set_property() и get_property(), позволяют установить и получить любое свойство для любого типа сущности. Эти команды требуют указания типа сущности в качестве аргументов команды, а также некоторой информации, специфичной для сущности.

```
set_property(entitySpecific
             [APPEND] [APPEND_STRING]
             PROPERTY propName [value1 [value2 [...]]])
```

entitySpecific определяет сущность, свойство которой устанавливается. Это должно быть *одно* из следующих значений:

```
GLOBAL
DIRECTORY [dir]
TARGET    [target1 [target2 [...]]]
SOURCE    [source1 [source2 [...]]]
INSTALL   [file1   [file2   [...]]]
TEST      [test1   [test2   [...]]]
CACHE     [var1    [var2    [...]]]
```

Первое слово каждого из перечисленных выше слов определяет тип сущности, свойство которой устанавливается. GLOBAL означает саму сборку, поэтому конкретное имя сущности не требуется. Для DIRECTORY, если не указан dir, используется текущий исходный каталог. Для всех остальных типов сущностей может быть перечислено любое количество элементов этого типа.

Ключевое слово PROPERTY обозначает все остальные аргументы как определяющие имя свойства и его значение(я). Имя propName обычно соответствует одному из свойств, определенных в документации CMake, некоторые из которых рассматриваются в последующих главах. Значение(я) зависит от свойства. В проекте также разрешается создавать новые свойства, помимо тех, которые уже определены CMake. Проект должен сам определить, что означают такие специфические для проекта свойства и как они могут повлиять на сборку.

Если вы решите сделать это, то для проектов будет разумно использовать какой-нибудь специфический для проекта префикс в имени свойства, чтобы избежать возможного столкновения имен со свойствами, определенными CMake или другими сторонними пакетами.

Ключевые слова APPEND и APPEND_STRING могут использоваться для управления тем, как именованное свойство обновляется, если оно уже имеет значение. Если ни одно из ключевых слов не указано, то заданное значение(я) заменяет любое предыдущее значение. Ключевое слово APPEND изменяет поведение, добавляя значение(я) к существующему, формируя список, тогда как ключевое слово APPEND_STRING берет существующее значение и добавляет новое значение(я), объединяя их как строки, а не как список (см. также специальное примечание для наследуемых свойств ниже). В следующей таблице показаны различия.

Previous Value(s)	New Value(s)	No Keyword	APPEND	APPEND_STRING
foo	bar	bar	foo;bar	foobar
a;b	c;d	c;d	a;b;c;d	a;bc;d

Команда get_property() имеет аналогичную форму:

```
get_property(resultVar entitySpecific
             PROPERTY propName
             [DEFINED | SET | BRIEF_DOCS | FULL_DOCS])
```

Ключевое слово PROPERTY всегда является обязательным и за ним всегда следует имя свойства, которое нужно получить. Результат извлечения хранится в переменной, имя которой задается resultVar. Часть entitySpecific аналогична части для set_property() и должна быть *одной* из следующих:

```
GLOBAL
DIRECTORY [dir]
TARGET    target
SOURCE    source
INSTALL   file
TEST      test
CACHE     var
VARIABLE
```

Как и раньше, GLOBAL относится к сборке в целом и поэтому не требует указания конкретного объекта. DIRECTORY может использоваться с указанием или без указания конкретной директории, при этом, если директория не указана, предполагается текущая директория источника. Для большинства других диапазонов необходимо назвать конкретную сущность в этом диапазоне, после чего будет получено запрашиваемое свойство, связанное с этой сущностью.

Тип VARIABLE немного отличается: имя переменной указывается как propName, а не присоединяется к ключевому слову VARIABLE. Это может показаться несколько неинтуитивным, но представьте ситуацию, если бы переменная была названа сущностью вместе с ключевым словом VARIABLE, как и для других ключевых слов типа сущности. В такой ситуации не нужно было бы ничего указывать для имени свойства. Возможно, вам поможет представление о VARIABLE как об указании текущей *области видимости*, тогда интересующее вас свойство - это переменная, названная propName. При таком понимании VARIABLE соответствует тому, как обрабатываются другие типы сущностей.

Если ни одно из необязательных ключевых слов не указано, то извлекается значение именованного свойства. Это типичное использование команды get_property(). Обратите внимание, что на практике использование области видимости VARIABLE в команде get_property() встречается относительно редко. Значения переменных можно получить напрямую с помощью синтаксиса ${}, что понятнее и проще, чем использование get_property().

Дополнительные ключевые слова могут быть использованы для получения подробной информации о свойстве, кроме его значения:

DEFINED

Результатом поиска будет булево значение, указывающее, было ли определено именованное свойство или нет. В случае запросов в области видимости VARIABLE результат будет истинным, только если именованная переменная была явно определена с помощью команды define_property() (см. ниже).

SET

Результатом поиска будет булево значение, указывающее, было ли именованное свойство установлено или нет. Это более сильный тест, чем DEFINED, поскольку он проверяет, действительно ли именованному свойству было присвоено значение (само значение не имеет значения). Свойство может возвращать TRUE для DEFINED и FALSE для SET, или наоборот.

BRIEF_DOCS

Получает строку краткой документации для именованного свойства. Если для свойства не была определена краткая документация, результатом будет строка NOTFOUND.

FULL_DOCS

Получает полную документацию для именованного свойства. Если для свойства не была определена краткая документация, результатом будет строка NOTFOUND.

Из необязательных ключевых слов все, кроме SET, имеют небольшую ценность, если только в проекте явно не вызвана define_property() для заполнения запрашиваемой информации для конкретной сущности. Эта редко используемая команда имеет следующий вид:

```
define_property(entityType
                PROPERTY propName [INHERITED]
                BRIEF_DOCS briefDoc [moreBriefDocs...]
                FULL_DOCS fullDoc [moreFullDocs...])
```

Важно отметить, что эта команда не устанавливает значение свойства, только его документацию и то, наследует ли оно свое значение из другого места, если оно не было установлено. Тип entityType должен быть одним из GLOBAL, DIRECTORY, TARGET, SOURCE, TEST, VARIABLE или CACHED_VARIABLE, а propName указывает на определяемое свойство. Сущность не указывается, хотя, как и для команды get_property(), в случае VARIABLE имя переменной указывается как propName. Краткие документы, как правило, не должны превышать одной относительно короткой строки, в то время как полные документы могут быть длиннее и при необходимости занимать несколько строк.

Если при определении свойства используется опция INHERITED, команда get_property() будет выполняться по цепочке до родительской области видимости, если это свойство не установлено в названной области видимости. Например, если запрашивается свойство DIRECTORY, но оно не установлено для указанного каталога, свойство родительского каталога запрашивается рекурсивно вверх по иерархии каталогов, пока свойство не будет найдено или не будет достигнут верхний уровень дерева источника. Если свойство не найдено в каталоге верхнего уровня, то выполняется поиск в области GLOBAL. Аналогично, если запрашивается свойство TARGET, SOURCE или TEST, но оно не установлено для указанной сущности, будет произведен поиск в области DIRECTORY (включая рекурсивный поиск по иерархии каталогов и, при необходимости, в области GLOBAL). Для переменных VARIABLE и CACHE такой функции цепочки не предусмотрено, так как они по своей конструкции уже привязаны к области видимости родительской переменной.

Наследование свойств INHERITED распространяется только на команду get_property() и аналогичные ей функции get_... для определенных типов свойств (рассматриваются в разделах ниже). При вызове set_property()

с опциями APPEND или APPEND_STRING учитывается только непосредственное значение свойства (т.е. наследование не происходит при выработке значения для добавления).

В CMake имеется большое количество предопределенных свойств каждого типа. Разработчикам следует обратиться к справочной документации CMake, чтобы узнать о доступных свойствах и их назначении. В последующих главах обсуждаются многие из этих свойств и рассматривается их связь с другими командами, переменными и возможностями CMake.

9.2. Глобальные свойства

Глобальные свойства относятся ко всей сборке в целом. Они обычно используются для изменения запуска инструментов сборки или других аспектов поведения инструментов, для определения структуры файлов проекта и для предоставления некоторой информации на уровне сборки.

В дополнение к общим командам set_property() и get_property(), CMake также предоставляет get_cmake_property() для запроса глобальных сущностей. Это не просто сокращение для get_property(), хотя его можно использовать для получения значения любого глобального свойства.

```
get_cmake_property(resultVar property)
```

Как и для get_property(), resultVar - это имя переменной, в которой будет храниться значение запрашиваемого свойства после возврата команды. Аргумент property может быть именем любого глобального свойства или одного из следующих псевдосвойств:

VARIABLES
: Возвращает список всех обычных (т.е. некэшированных) переменных.

CACHE_VARIABLES
: Возвращает список всех переменных кэша.

COMMANDS
: Возвращает список всех определенных команд, функций и макросов. Команды предварительно определяются CMake, в то время как функции и макросы могут быть определены либо CMake (обычно через модули), либо самими проектами. Некоторые из возвращаемых имен могут соответствовать недокументированным или внутренним сущностям, не предназначенным для прямого использования проектами. Имена могут быть возвращены в верхнем/нижнем регистре, отличном от того, как они были изначально определены.

MACROS
: Возвращает список только определенных макросов. Это будет подмножество того, что возвращает псевдосвойство COMMANDS, но обратите внимание, что верхний/нижний регистр имен может отличаться от того, что сообщает псевдосвойство COMMANDS.

COMPONENTS
: Возвращает список всех компонентов, определенных командами install(), которые рассматриваются в главе 25, *Установка.*

Эти псевдосвойства, доступные только для чтения, технически не являются глобальными свойствами (их нельзя получить, например, с помощью get_property()), но в общих чертах они очень похожи. Их можно получить только с помощью get_cmake_property().

9.3. Свойства каталога

Каталоги также поддерживают свой собственный набор свойств. Логически свойства каталогов находятся где-то между глобальными свойствами, которые применяются везде, и целевыми свойствами, которые влияют только на отдельные цели. Как таковые, свойства каталога в основном сосредоточены на установке значений по умолчанию для целевых свойств и переопределении глобальных свойств или значений по умолчанию для текущего каталога. Несколько свойств каталога, доступных только для чтения, также обеспечивают определенную степень интроспекции, содержа информацию о том, как сборка достигла каталога, какие вещи были определены в этой точке и т.д.

Для удобства CMake предоставляет специальные команды для установки и получения свойств каталога, которые немного более лаконичны, чем их общие аналоги. Команда setter определяется следующим образом:

```
set_directory_properties(PROPERTIES prop1 val1 [prop2 val2 ...])
```

Будучи более лаконичной, эта специфическая для каталога команда-установщик не имеет опции APPEND или APPEND_STRING. Это означает, что ее можно использовать только для установки или замены свойства, она не может быть использована для добавления к существующему свойству напрямую. Еще одним ограничением этой команды по сравнению с более общей set_property() является то, что она всегда применяется к текущему каталогу. Проекты могут использовать эту более специфическую форму там, где это удобно, и использовать общую форму в других местах, или для согласованности можно использовать более общую форму везде. Ни один из подходов не является более правильным, это скорее вопрос предпочтений.

Команда getter, специфичная для каталога, имеет две формы:

```
get_directory_property(resultVar [DIRECTORY dir] property)
get_directory_property(resultVar [DIRECTORY dir] DEFINITION varName)
```

Первая форма используется для получения значения свойства из определенного каталога или из текущего каталога, если аргумент DIRECTORY не используется. Вторая форма извлекает значение *переменной*, что может показаться не слишком полезным, но она предоставляет возможность получить значение переменной из другой области видимости каталога, отличной от текущей (если используется аргумент DIRECTORY). На практике эта вторая форма нужна редко, и ее использования следует избегать в сценариях, отличных от отладки сборки или подобных временных задач.

Для любой формы команды get_directory_property(), если используется аргумент DIRECTORY, именованный каталог должен быть уже обработан CMake. CMake не может знать свойства области каталогов, с которыми он еще не сталкивался.

9.4. Свойства цели

Немногие вещи в CMake оказывают такое сильное и прямое влияние на процесс сборки целей, как свойства цели. Они контролируют и предоставляют информацию обо всем - от флагов, используемых для компиляции исходных файлов, до типа и расположения собранных двоичных и промежуточных файлов. Некоторые свойства цели влияют на то, как цели представлены в проекте IDE разработчика, другие - на инструменты, используемые при компиляции/связывании. Короче говоря, свойства цели - это то место, где собирается и применяется большинство деталей о том, как на самом деле превратить исходные файлы в двоичные файлы.

В CMake появилось несколько методов для манипулирования свойствами цели. В дополнение к общим командам set_property() и get_property(), CMake также предоставляет некоторые эквиваленты, специфичные для конкретной цели, для удобства:

```
set_target_properties(target1 [target2...]
                      PROPERTIES
                      propertyName1 value1
                      [propertyName2 value2] ... )
get_target_property(resultVar target propertyName)
```

Как и команда set_directory_properties(), set_target_properties() не обладает полной гибкостью set_property(), но обеспечивает более простой синтаксис для распространенных случаев. Команда set_target_properties() не поддерживает добавление к существующим значениям свойств, и если для данного свойства необходимо указать значение в виде списка, команда set_target_properties() требует, чтобы это значение было указано в строковой форме, например, "this;is;a;list".

Команда get_target_property() - это упрощенная версия команды get_property(). Она сосредоточена исключительно на предоставлении простого способа получения значения целевого свойства и, по сути, является сокращенной версией общей команды.

В дополнение к общим и специфическим для конкретной цели геттерам и сеттерам свойств, CMake также имеет ряд других команд, которые изменяют свойства цели. В частности, семейство команд target_...() является важной частью CMake, и все проекты CMake, кроме самых тривиальных, обычно используют их. Эти команды определяют не только свойства конкретной цели, но и то, как эта информация может быть передана другим целям, которые с ней связываются. В главе 14, *Compiler And Linker Essentials,* подробно рассматриваются эти команды и то, как они связаны со свойствами цели.

9.5. Свойства источника

CMake также поддерживает свойства отдельных исходных файлов. Они позволяют тонко управлять флагами компилятора на основе каждого файла, а не для всех исходных текстов цели. Они также позволяют предоставлять дополнительную информацию об исходном файле для изменения того, как CMake или инструменты сборки обращаются с файлом, например, указывают, генерируется ли он как часть сборки, какой компилятор использовать для работы с ним, опции для некомпиляторных инструментов, работающих с файлом, и так далее.

Проекты редко должны запрашивать или изменять свойства исходных файлов, но для тех ситуаций, когда это необходимо, CMake предоставляет специальные команды setter и getter, чтобы облегчить задачу. Они работают по той же схеме, что и другие команды setter и getter, специфичные для свойств:

```
set_source_files_properties(file1 [file2...]
                            PROPERTIES
                            propertyName1 value1
                            [propertyName2 value2] ... )
get_source_file_property(resultVar sourceFile propertyName)
```

Опять же, для сеттера не предусмотрена функциональность APPEND, а геттер - это просто синтаксическое сокращение для общей команды get_property() и не предлагает никакой новой функциональности.

Проекты должны помнить, что свойства источника видны только целям, определенным в той же области каталогов. Если установка свойства источника происходит в другой области каталогов, цель не увидит изменения свойства и, следовательно, компиляция и т.д. этого исходного файла не будет затронута. Также следует помнить, что один исходный файл может быть скомпилирован в несколько целей, поэтому любые установленные свойства источника должны иметь смысл для всех целей, в которые он добавлен.

Прежде чем спешить начать использовать свойства источника, разработчикам следует знать об одной детали реализации, которая может стать серьезным препятствием для их использования в некоторых ситуациях. При использовании некоторых генераторов CMake (в частности, генератора Unix Makefiles) зависимости между исходными текстами и свойствами исходных текстов оказываются сильнее, чем можно было бы ожидать. В частности, если свойства источника используются для изменения флагов компилятора для конкретных исходных файлов, а не для всей цели, изменение флагов компилятора источника все равно приведет к перестройке всех исходных файлов цели, а не только затронутого исходного файла. Это ограничение того, как детали зависимости обрабатываются в настройках Makefile, где проверка того, изменились ли флаги компилятора каждого отдельного источника, приводит к непомерно большому снижению производительности, поэтому разработчики CMake решили реализовать зависимость на уровне цели. Типичный сценарий, в котором у проектов может возникнуть соблазн пойти по этому пути, - это передача сведений о версии только одному или двум источникам в качестве определений компилятора, но, как обсуждалось в разделе 19.2, "Доступ исходного кода к сведениям о версии", существуют лучшие альтернативы свойствам источника, которые не страдают от тех же побочных эффектов.

9.6. Свойства переменных кэша

Свойства переменных кэша немного отличаются по назначению от других типов свойств. По большей части, свойства переменных кэша направлены на то, как переменные кэша обрабатываются в графическом интерфейсе CMake и консольном инструменте ccmake, а не влияют на сборку каким-либо ощутимым образом. Для работы с ними также не предусмотрено никаких дополнительных команд, поэтому общие команды set_property() и get_property() должны использоваться с ключевым словом CACHE

В разделе 5.3, "Переменные кэша", обсуждался ряд аспектов переменных кэша, которые в конечном итоге отражаются в свойствах переменных кэша.

- Каждая переменная кэша имеет *тип*, который должен быть одним из BOOL, FILEPATH, PATH, STRING или INTERNAL. Этот тип можно получить с помощью функции get_property() с именем свойства TYPE. Тип влияет на то, как графический интерфейс CMake и ccmake представляют эту кэш-переменную в пользовательском интерфейсе и какой виджет используется для редактирования ее значения. Любая переменная с типом INTERNAL не будет отображаться ни в графическом интерфейсе CMake, ни в ccmake.
- Переменная кэша может быть помечена как расширенная с помощью команды mark_as_advanced(), которая на самом деле просто устанавливает булево свойство переменной кэша ADVANCED. Графический интерфейс CMake и инструмент ccmake предоставляют возможность показывать или скрывать расширенные переменные кэша, позволяя пользователю выбирать, сосредоточиться ли ему только на основных базовых переменных или увидеть полный набор не внутренних переменных.
- Строка справки переменной кэша обычно устанавливается в ходе вызова команды set(), но ее также можно изменить или прочитать с помощью свойства переменной кэша HELPSTRING. Эта строка справки используется в качестве подсказки в графическом интерфейсе CMake и в качестве однострочной подсказки в инструменте ccmake.
- Если переменная кэша имеет тип STRING, то CMake GUI будет искать свойство переменной кэша с именем STRINGS. Если оно не пустое, то ожидается, что это будет список допустимых значений для переменной, и тогда CMake GUI представит эту переменную в виде комбинированного окна с этими значениями, а не в

виде произвольного виджета для ввода текста. В случае с ccmake нажатие клавиши Enter на этой кэш-переменной приведет к циклическому перебору предоставленных значений. Обратите внимание, что CMake не требует, чтобы переменная кэша была одним из значений свойства STRINGS, это лишь удобство для графического интерфейса CMake и инструментов ccmake. Когда CMake выполняет свой шаг configure, он по-прежнему рассматривает переменную кэша как произвольную строку, поэтому переменной кэша можно придать любое значение либо в командной строке cmake, либо с помощью команд set() в проекте.

9.7. Другие типы собственности

CMake также поддерживает свойства отдельных тестов и предоставляет обычные специфичные для тестов версии команд property setter и getter:

```
set_tests_properties(test1 [test2...]
                     PROPERTIES
                     propertyName1 value1
                     [propertyName2 value2] ... )
get_test_property(resultVar test propertyName)
```

Как и их аналоги, это просто немного более сжатые версии общих команд, в которых отсутствует функция APPEND, но которые могут быть более удобными в некоторых обстоятельствах. Тесты подробно рассматриваются в главе 24 "*Тестирование*".

Другой тип свойств, поддерживаемый CMake, предназначен для установленных файлов. Эти свойства специфичны для используемого типа упаковки и, как правило, не нужны большинству проектов.

9.8. Рекомендуемые практики

Свойства являются важной частью CMake. Ряд команд позволяет устанавливать, изменять или запрашивать различные типы свойств, некоторые из которых имеют дополнительные последствия для зависимостей между проектами.

- Всеми свойствами, кроме специальных глобальных псевдосвойств, можно полностью управлять с помощью общей команды set_property(), что делает ее предсказуемой для разработчиков и предлагает гибкую функциональность APPEND там, где это необходимо. Специфические сеттеры свойств могут быть более удобными в некоторых ситуациях, например, позволяя устанавливать сразу несколько свойств, но отсутствие у них функциональности APPEND может подтолкнуть некоторые проекты к использованию команды set_property(). Ни то, ни другое не является правильным или неправильным, хотя распространенной ошибкой является использование команд, специфичных для свойств, для замены значения свойства вместо добавления к нему.

- Для целевых свойств настоятельно рекомендуется использовать различные команды target_...(), а не манипулировать связанными целевыми свойствами напрямую. Эти команды не только управляют свойствами конкретных целей, но и устанавливают отношения зависимости между целями, так что CMake может распространять некоторые свойства автоматически. В главе 14, *Compiler And Linker Essentials,* обсуждается ряд тем, которые подчеркивают сильное предпочтение команд target_...().

- Свойства исходного кода обеспечивают тонкую детализацию уровня контроля опций компилятора и т.д. Однако они могут оказать нежелательное негативное влияние на поведение сборки проекта. В частности, некоторые генераторы CMake могут пересобирать больше, чем это необходимо, когда изменяются параметры компиляции всего нескольких исходных файлов. Проектам следует рассмотреть возможность

использования других альтернатив свойствам исходного кода, где это возможно, таких как методы, приведенные в разделе 19.2, "Доступ исходного кода к сведениям о версии".

Глава 10. Выражения генератора

При запуске CMake разработчики склонны думать о нем как об одном шаге, который включает чтение файла CMakeLists.txt проекта и создание соответствующего набора проектных файлов, специфичных для генератора (например, файлы решений и проектов Visual Studio, проект Xcode, Unix Makefiles или входные файлы Ninja). Однако здесь есть два совершенно разных этапа. При запуске CMake конец выходного журнала обычно выглядит следующим образом:

```
-- Configuring done
-- Generating done
-- Build files have been written to: /some/path/build
```

Когда вызывается CMake, он сначала считывает и обрабатывает файл CMakeLists.txt в верхней части дерева исходников, включая любые другие файлы, которые он подтягивает. По мере выполнения команд, функций и т.д. создается внутреннее представление проекта. Это называется шагом *configure*. На этом этапе создается большая часть вывода в консольный журнал, включая содержимое команд message(). В конце этапа configure в журнал выводится сообщение -- Configuring done.

После того как CMake закончил чтение и обработку файла CMakeLists.txt, он выполняет шаг *генерации*. На этом этапе создаются файлы проекта инструмента сборки с использованием внутреннего представления, созданного на этапе configure. По большей части разработчики склонны игнорировать шаг генерации и воспринимать его как конечный результат конфигурирования. В журнале консоли почти всегда появляется сообщение -- Generating done сразу после завершения шага configure, так что это вполне объяснимо. Но есть ситуации, когда понимание разделения на два разных этапа особенно важно.

Рассмотрим проект, обработанный для генератора CMake с несколькими конфигурациями, например Xcode или Visual Studio. Когда читаются файлы CMakeLists.txt, CMake не знает, для какой конфигурации будет собираться цель. Это многоконфигурационная установка, поэтому существует более одного выбора (например, Debug, Release и т.д.). Разработчик выбирает конфигурацию во время *сборки, уже* после завершения работы CMake. Это может создать проблему, если файл CMakeLists.txt хочет сделать что-то вроде копирования файла в тот же каталог, что и конечный исполняемый файл для данной цели, поскольку расположение этого каталога зависит от того, какая конфигурация собирается. Необходим некий заполнитель, чтобы сказать CMake: "Для какой бы конфигурации ни собиралась программа, используйте каталог конечного исполняемого файла".

Это яркий пример функциональности, предоставляемой выражениями генератора. Они обеспечивают способ кодирования некоторой логики, которая не оценивается во время конфигурации, а откладывается до фазы генерации, когда записываются файлы проекта. Они могут использоваться для выполнения условной логики, вывода строк, предоставляющих информацию о различных аспектах сборки, таких как каталоги, имена вещей, детали платформы и многое другое. Они даже могут быть использованы для предоставления различного содержимого в зависимости от того, выполняется сборка или установка.

Выражения-генераторы нельзя использовать везде, но они поддерживаются во многих местах. В справочной документации CMake, если определенная команда или свойство поддерживает выражения-генераторы, в документации об этом будет сказано. Набор свойств, поддерживающих выражения-генераторы, со временем расширялся, а в некоторых выпусках CMake набор поддерживаемых выражений также расширялся. Проектам следует убедиться, что для минимальной версии CMake, которая им требуется, изменяемые свойства действительно поддерживают используемые выражения генератора.

10.1. Простая булева логика

Выражение генератора задается с помощью синтаксиса $<...>, где содержимое между угловыми скобками может принимать несколько различных форм. Как станет ясно в ближайшее время, важной особенностью является условное включение содержимого. Самыми основными выражениями генератора, которые позволяют это сделать, являются следующие:

```
$<1:...>
$<0:...>
$<BOOL:...>
```

Для $<1:...> результатом выражения будет часть ..., тогда как для $<0:...> часть ... игнорируется и результатом выражения будет пустая строка. Выражение $<BOOL:...> можно использовать для преобразования всего, что CMake распознает как ложное булево значение, в 0, а всего остального - в 1 (подробнее о том, что CMake считает ложным значением, см. обсуждение в разделе 6.1.1, "Основные выражения"). Вместе эти выражения-генераторы обеспечивают простой, но мощный способ опционального включения содержимого. Также поддерживаются логические операции:

```
$<AND:expr[,expr...]>
$<OR:expr[,expr...]>
$<NOT:expr>
```

Выражения AND и OR могут принимать любое количество аргументов, разделенных запятыми, и выдавать соответствующий логический результат, а NOT принимает только одно выражение и выдает отрицание своего аргумента. Поскольку AND, OR и NOT требуют, чтобы их выражения оценивались только в 0 или 1, подумайте о том, чтобы обернуть эти выражения в $<BOOL:...>, чтобы заставить логику более терпимо относиться к тому, что считать истинным или ложным выражением.

В CMake 3.8 и более поздних версиях логику if-then-else также можно очень удобно выразить с помощью специального выражения $<IF:...>:

```
$<IF:expr,val1,val0>
```

Как обычно, expr должен иметь значение 1 или 0. Результатом будет val1, если expr имеет значение 1, и val0, если expr имеет значение 0. До CMake 3.8 эквивалентная логика должна была быть выражена следующим более подробным способом, который требует, чтобы выражение было задано дважды:

```
$<expr:val1>$<$<NOT:expr>:val0>
```

Выражения-генераторы могут быть вложенными, что позволяет строить выражения произвольной сложности. В приведенном выше примере показано вложенное условие, но любая часть выражения-генератора может быть вложенной. Следующие примеры демонстрируют возможности, рассмотренные до сих пор:

Expression	Result	Notes
$<1:foo>	foo	
$<0:foo>		
$<true:foo>		Error, not a 1 or 0
$<$<BOOL:true>:foo>	foo	
$<$<NOT:0>:foo>	foo	
$<$<NOT:1>:foo>		
$<$<NOT:true>:foo>	foo	Error, NOT requires a 1 or 0
$<$<AND:1,0>:foo>		
$<$<OR:1,0>:foo>	foo	
$<1:$<$<BOOL:false>:foo>>		
$<IF:$<BOOL:${foo}>,yes,no>	Result will be yes or no depending on ${foo}	

Как и для команды if(), CMake также обеспечивает поддержку проверки строк, чисел и версий в выражениях генератора, хотя синтаксис немного отличается. Все приведенные ниже выражения имеют значение 1, если выполняется соответствующее условие, и 0 в противном случае.

```
$<STREQUAL:string1,string2>
$<EQUAL:number1,number2>
$<VERSION_EQUAL:version1,version2>
$<VERSION_GREATER:version1,version2>
$<VERSION_LESS:version1,version2>
```

Еще одно очень полезное условное выражение - проверка типа сборки:

```
$<CONFIG:arg>
```

Это значение будет равно 1, если arg соответствует фактически собираемому типу сборки, и 0 для всех остальных типов сборки. Обычно это используется для предоставления флагов компилятора только для отладочных сборок или для выбора различных реализаций для разных типов сборок. Например:

```
add_executable(myApp src1.cpp src2.cpp)

# Before CMake 3.8
target_link_libraries(myApp PRIVATE
    $<$<CONFIG:Debug>:checkedAlgo>
    $<$<NOT:$<CONFIG:Debug>>:fastAlgo>
)

# CMake 3.8 or later allows a more concise form
target_link_libraries(myApp PRIVATE
    $<IF:$<CONFIG:Debug>,checkedAlgo,fastAlgo>
)
```

Вышеприведенная команда свяжет исполняемый файл с библиотекой checkedAlgo для Debug-сборок и с библиотекой fastAlgo для всех остальных типов сборок. Выражение генератора $<CONFIG:...> - единственный способ надежно обеспечить такую функциональность, которая работает для всех генераторов проектов CMake,

включая генераторы с несколькими конфигурациями, такие как Xcode или Visual Studio. Более подробно эта тема рассматривается в разделе 13.2, "Распространенные ошибки".

CMake предлагает еще больше условных тестов, основанных на таких вещах, как детали платформы и компилятора, настройки политики CMake и т.д. Разработчикам следует обратиться к справочной документации CMake для получения полного набора поддерживаемых условных выражений.

10.2. Сведения о цели

Еще одно распространенное использование выражений-генераторов - предоставление информации о целях. Любое свойство цели может быть получено с помощью одной из следующих двух форм:

```
$<TARGET_PROPERTY:target,property>
$<TARGET_PROPERTY:property>
```

Первая форма предоставляет значение именованного свойства из указанной цели, а вторая форма извлекает свойство из цели, на которой используется выражение генератора.

Хотя TARGET_PROPERTY - это очень гибкий тип выражения, он не всегда является лучшим способом получения информации о цели. Например, CMake также предоставляет другие выражения, которые дают подробную информацию о каталоге и имени файла собранного двоичного файла цели. Эти более прямые выражения позволяют извлекать части некоторых свойств или вычислять значения на основе необработанных свойств. Наиболее общим из них является набор выражений генератора TARGET_FILE:

TARGET_FILE
: Это даст абсолютный путь и имя файла двоичного файла целевой библиотеки, включая любой префикс и суффикс файла, если это имеет значение для платформы (например, .exe, .dylib). Для платформ на базе Unix, где разделяемые библиотеки обычно имеют сведения о версии в имени файла, они также будут включены.

TARGET_FILE_NAME
: То же самое, что и TARGET_FILE, но без пути (т.е. предоставляется только часть имени файла).

TARGET_FILE_DIR
: То же, что и TARGET_FILE, но без имени файла. Это самый надежный способ получить каталог, в котором собирается конечный исполняемый файл или библиотека. Его значение отличается для разных конфигураций сборки при использовании генератора нескольких конфигураций, например, Xcode или Visual Studio.

Приведенные выше три выражения TARGET_FILE особенно полезны при определении пользовательских правил сборки для копирования файлов на этапах после сборки (см. раздел 17.2, "Добавление этапов сборки к существующей цели"). В дополнение к выражениям TARGET_FILE, CMake также предоставляет несколько специфических для библиотеки выражений, которые выполняют аналогичные функции, но немного по-другому обрабатывают префикс и/или суффикс имени файла. Эти выражения имеют имена, начинающиеся с TARGET_LINKER_FILE и TARGET_SONAME_FILE, и, как правило, используются не так часто, как выражения TARGET_FILE.

Проекты, поддерживающие платформу Windows, также могут получить подробную информацию о PDB-файлах для данной цели. Опять же, в основном это может быть использовано в задачах пользовательской сборки. Выражения, начинающиеся с TARGET_PDB_FILE, работают по аналогичной схеме, как и для TARGET_PROPERTY, предоставляя сведения о пути и имени файла PDB, используемого для цели, для которой используется выражение генератора.

Еще одно выражение генератора, относящееся к целям, заслуживает особого упоминания. CMake позволяет определить цель библиотеки как *объектную библиотеку*, то есть это не библиотека в обычном смысле, а просто набор объектных файлов, которые CMake ассоциирует с целью, но не приводит к созданию конечного библиотечного файла. Поскольку это объектные файлы, их нельзя компоновать как единое целое (хотя в CMake 3.12 это ограничение ослаблено). Вместо этого они должны быть добавлены к целям тем же способом, что и *исходные тексты*. Затем CMake включает эти объектные файлы на этапе компоновки так же, как и объектные файлы, созданные при компиляции исходных текстов цели. Это делается с помощью выражения генератора $<TARGET_OBJECTS:...>, которое перечисляет объектные файлы в форме, удобной для использования add_executable() или add_library().

```
# Define an object library
add_library(objLib OBJECT src1.cpp src2.cpp)

# Define two executables which each have their own source
# file as well as the object files from objLib
add_executable(app1 app1.cpp $<TARGET_OBJECTS:objLib>)
add_executable(app2 app2.cpp $<TARGET_OBJECTS:objLib>)
```

В приведенном выше примере отдельная библиотека для objLib не создается, но исходные файлы src1.cpp и src2.cpp компилируются только один раз. Это может быть более удобным для некоторых сборок, поскольку позволяет избежать затрат времени сборки на создание статической библиотеки или затрат времени выполнения на разрешение символов для динамической библиотеки, но при этом избежать необходимости компилировать одни и те же исходные тексты несколько раз.

10.3. Общая информация

Выражения генератора могут предоставлять информацию не только о целях. Можно получить информацию об используемом компиляторе, платформе, для которой собирается цель, имя конфигурации сборки и многое другое. Такие выражения обычно используются в более сложных ситуациях, например, для работы с пользовательским компилятором или для решения проблем, характерных для конкретного компилятора или набора инструментов. Эти выражения также могут использоваться не по назначению, поскольку может показаться, что они дают возможность создавать пути к объектам, которые в противном случае можно было бы получить более надежными методами, например, с помощью выражений TARGET_FILE или других возможностей CMake. Разработчикам следует хорошо подумать, прежде чем полагаться на более общие выражения генератора информации как на способ решения проблемы. Тем не менее, некоторые из этих выражений действительно могут быть использованы. Некоторые из наиболее распространенных и несколько полезных выражений перечислены здесь в качестве отправной точки для дальнейшего чтения:

$<CONFIG>

Оценивает тип сборки. Используйте это выражение вместо переменной CMAKE_BUILD_TYPE, поскольку эта переменная не используется в генераторах проектов с несколькими конфигурациями, таких как Xcode или Visual Studio. В предыдущих версиях CMake для этого использовалось устаревшее выражение $<CONFIGURATION>, но теперь проекты должны использовать только $<CONFIG>.

$<PLATFORM_ID>

Определяет платформу, для которой собирается целевая программа. Это может быть полезно в ситуациях кросс-компиляции, особенно когда сборка может поддерживать несколько платформ (например, сборки устройств и симуляторов). Это выражение генератора тесно связано с переменной CMAKE_SYSTEM_NAME, и проектам следует подумать, не будет ли использование этой переменной проще в их конкретной ситуации. $<C_COMPILER_VERSION> **,** $<CXX_COMPILER_VERSION>

В некоторых ситуациях может быть полезно добавлять содержимое только в том случае, если версия компилятора старше или новее некоторой определенной версии. Этого можно добиться с помощью выражений генератора $<VERSION_???:...>. Например, для создания строки OLD_COMPILER, если версия компилятора C++ меньше 4.2.0, можно использовать следующее выражение:

```
$<$<VERSION_LESS:$<CXX_COMPILER_VERSION>,4.2.0>:OLD_COMPILER>
```

Такие выражения, как правило, используются только в ситуациях, когда тип компилятора известен, а специфическое поведение компилятора должно обрабатываться проектом каким-то особым образом. Это может быть полезным приемом в определенных ситуациях, но это может снизить переносимость проекта, если он слишком сильно полагается на такие выражения.

$<НИЖНИЙ_ПРОПИСНОЙ:...> , $<ВЕРХНИЙ_ПРОПИСНОЙ:...>

С помощью этих выражений можно преобразовать любое содержимое в нижний или верхний регистр. Это может быть особенно полезно в качестве шага перед выполнением сравнения строк. Например:

```
$<STREQUAL:$<UPPER_CASE:${someVar}>,FOOBAR>
```

10.4. Рекомендуемые практики

По сравнению с другими функциональными возможностями, выражения генератора - это недавно добавленная возможность CMake. Из-за этого в большинстве материалов в Интернете и других местах, посвященных CMake, они, как правило, не используются, что очень жаль, поскольку выражения генераторов обычно более надежны и обеспечивают большую универсальность, чем старые методы. Есть несколько распространенных примеров, когда благонамеренное руководство приводит к логике, которая работает только для подмножества поддерживаемых генераторов проектов или платформ, но использование подходящих выражений генератора не привело бы к таким ограничениям. Это особенно верно в отношении логики проекта, которая пытается делать разные вещи для разных типов сборки. Поэтому разработчикам следует ознакомиться с возможностями, которые предоставляют выражения генератора. Эти выражения, упомянутые выше, являются лишь подмножеством того, что поддерживает CMake, но они составляют хорошую основу для покрытия большинства ситуаций, с которыми могут столкнуться большинство разработчиков.

При разумном использовании выражений генератора можно получить более лаконичные файлы CMakeLists.txt. Например, условное включение исходного файла в зависимости от типа сборки может быть выполнено относительно лаконично, как показал пример, приведенный ранее для $<CONFIG:...>. Такое использование сокращает количество логики ifthen-else, что приводит к улучшению читабельности, если выражения генератора не слишком сложны. Выражения-генераторы также отлично подходят для работы с содержимым, которое меняется в зависимости от цели или типа сборки. Ни один другой механизм в CMake не предлагает такой же степени гибкости и универсальности для обработки множества факторов, которые могут внести вклад в конечное содержимое, необходимое для конкретного целевого свойства.

И наоборот, легко переборщить и попытаться сделать все выражение генератором. Это может привести к чрезмерно сложным выражениям, которые в конечном итоге затуманивают логику и которые трудно отлаживать. Как всегда, разработчики должны отдавать предпочтение ясности, а не умности, и это особенно верно в отношении выражений-генераторов. Сначала подумайте, нет ли в CMake уже специального средства для достижения того же результата. Различные модули CMake предоставляют более целевую функциональность, направленную на конкретный пакет стороннего разработчика или на выполнение

определенных специфических задач. Существует также множество переменных и свойств, которые могут упростить или вовсе заменить необходимость использования выражений-генераторов. Несколько минут обращения к справочной документации CMake могут сэкономить много часов ненужной работы по созданию сложных выражений-генераторов, которые на самом деле не нужны.

Глава 11. Модули

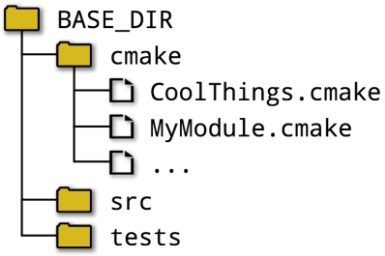
В предыдущих главах основное внимание было уделено основным аспектам CMake. Переменные, свойства, управление потоками, выражения генераторов, функции и т.д. - все это часть того, что можно считать языком CMake. В отличие от этого, модули - это предварительно созданные фрагменты кода CMake, построенные поверх основных возможностей языка. Они предоставляют богатый набор функциональных возможностей, которые проекты могут использовать для достижения самых разных целей. Будучи написанными и упакованными как обычный код CMake и, следовательно, читаемыми человеком, модули также могут быть полезным ресурсом для получения дополнительной информации о том, как выполнять задачи в CMake.

Модули собираются вместе и предоставляются в одном каталоге как часть релиза CMake. Проекты используют модули одним из двух способов: либо напрямую, либо как часть поиска внешнего пакета. Более прямой метод использования модулей использует команду include() для внедрения кода модуля в текущую область видимости. Это работает точно так же, как и поведение, уже обсуждавшееся в Разд. 7.2, "include()", за исключением того, что команде include() нужно передать только базовое имя модуля, а не полный путь или расширение файла. Все опции include() работают точно так же, как и раньше.

```
include(module [OPTIONAL] [RESULT_VARIABLE myVar] [NO_POLICY_SCOPE])
```

При задании имени модуля команда include() будет искать в четко определенном наборе мест файл, имя которого равно имени модуля (с учетом регистра) с добавлением .cmake. Например, include(FooBar) приведет к тому, что CMake будет искать файл с именем FooBar.cmake, а в системах, чувствительных к регистру, таких как Linux, имена файлов типа foobar.cmake не будут совпадать.

При поиске файла модуля CMake сначала обращается к переменной CMAKE_MODULE_PATH. Предполагается, что это список каталогов, и CMake будет искать в каждом из них по порядку. Будет использован первый подходящий файл, а если подходящий файл не найден или CMAKE_MODULE_PATH пуст или не определен, CMake будет искать в своем собственном внутреннем каталоге модуля. Такой порядок поиска позволяет проектам беспрепятственно добавлять свои собственные модули, добавляя каталоги в CMAKE_MODULE_PATH. Полезная схема - собрать все файлы модулей проекта в один каталог и добавить его в CMAKE_MODULE_PATH где-нибудь в начале файла CMakeLists.txt верхнего уровня. В приведенной ниже структуре каталогов показано такое расположение:



Затем в соответствующем файле CMakeLists.txt нужно только добавить каталог cmake в CMAKE_MODULE_PATH, после чего можно вызывать include(), используя только имя базового файла при загрузке каждого из модулей.

CMakeLists.txt:

```
cmake_minimum_required(VERSION 3.0)
project(Example)

list(APPEND CMAKE_MODULE_PATH "${CMAKE_SOURCE_DIR}/cmake")

# Inject code from project-provided modules
include(CoolThings)
include(MyModule)
```

Существует одно исключение из порядка поиска, используемого CMake для нахождения модуля. Если файл, вызывающий include(), находится в собственном внутреннем каталоге модулей CMake, то сначала будет произведен поиск во внутреннем каталоге модулей, *прежде чем* обратиться к CMAKE_MODULE_PATH. Это предотвращает случайную (или преднамеренную) замену официального модуля своим собственным и изменение задокументированного поведения.

Другим способом использования модулей является команда find_package(). Подробно это обсуждается в разделе 23.5, "Поиск пакетов", но пока что упрощенная форма этой команды без дополнительных ключевых слов демонстрирует ее базовое использование:

```
find_package(PackageName)
```

При таком использовании поведение очень похоже на include(), только CMake будет искать файл с именем FindPackageName.cmake, а не PackageName.cmake. Это метод, с помощью которого в сборку часто попадают сведения о внешнем пакете, включая такие вещи, как импортируемые цели, переменные, определяющие расположение соответствующих файлов, библиотек или программ, информация о дополнительных компонентах, сведения о версии и так далее. Набор опций и возможностей, связанных с find_package(), значительно богаче, чем у include(), и глава 23, *Finding Things,* посвящена подробному рассмотрению этой темы.

В оставшейся части этой главы представлен ряд интересных модулей, которые входят в состав релиза CMake. Это далеко не полный набор, но они дают представление о том, какие функции доступны. Другие модули будут представлены в последующих главах, где их функциональность тесно связана с обсуждаемой темой. Документация CMake содержит полный список всех доступных модулей, каждый из которых имеет свой собственный раздел справки, объясняющий, что предоставляет модуль и как его можно использовать. Однако имейте в виду, что качество документации варьируется от модуля к модулю.

11.1. Полезные пособия по развитию

Модуль CMakePrintHelpers предоставляет два макроса, которые делают печать значений свойств и переменных более удобной во время разработки. Они не предназначены для постоянного использования, а больше направлены на то, чтобы помочь разработчикам быстро и легко регистрировать информацию на время, чтобы помочь исследовать проблемы в проекте.

```
cmake_print_properties([TARGETS target1 [target2...]]
                       [SOURCES source1 [source2...]]
                       [DIRECTORIES dir1 [dir2...]]
                       [TESTS test1 [test2...]]
                       [CACHE_ENTRIES var1 [var2...]]
                       PROPERTIES property1 [property2...]
)
```

Этот макрос по сути объединяет get_property() и message() в один вызов. Необходимо указать только один из типов свойств, и каждое из названных свойств будет выведено для каждой сущности, указанной в списке. Это особенно удобно при регистрации информации для нескольких сущностей и/или свойств. Например:

```
add_executable(myApp main.c)
add_executable(myAlias ALIAS myApp)
add_library(myLib STATIC src.cpp)

include(CMakePrintHelpers)
cmake_print_properties(TARGETS myApp myib myAlias
                       PROPERTIES TYPE ALIASED_TARGET)
```

Выходными данными будут:

```
Properties for TARGET myApp:
  myApp.TYPE = "EXECUTABLE"
  myApp.ALIASED_TARGET = <NOTFOUND>
Properties for TARGET myLib:
  myLib.TYPE = "STATIC_LIBRARY"
  myLib.ALIASED_TARGET = <NOTFOUND>
Properties for TARGET myAlias:
  myAlias.TYPE = "EXECUTABLE"
  myAlias.ALIASED_TARGET = "myApp"
```

Модуль также предоставляет аналогичную функцию для регистрации значения одной или нескольких переменных:

```
cmake_print_variables(var1 [var2...])
```

Это работает для всех переменных, независимо от того, были ли они явно установлены проектом, автоматически установлены CMake или не были установлены вообще.

```
set(foo "My variable")
unset(bar)

include(CMakePrintHelpers)
cmake_print_variables(foo bar CMAKE_VERSION)
```

Выходные данные для вышеуказанной операции будут выглядеть примерно следующим образом:

```
foo="My variable" ; bar="" ; CMAKE_VERSION="3.8.2"
```

11.2. Конечность

При работе со встроенными платформами или проектами, предназначенными для широкого спектра архитектур, может быть желательно, чтобы проект знал об конечности целевой системы. Модуль TestBigEndian предоставляет макрос test_big_endian(), который компилирует небольшую тестовую программу для определения конечности целевой системы. Результат кэшируется, поэтому при последующих запусках CMake не нужно заново выполнять тест. Макрос принимает только один аргумент - имя переменной, в которой будет храниться булев результат (true означает, что система является big endian):

```
include(TestBigEndian)
test_big_endian(isBigEndian)
message("Is target system big endian: ${isBigEndian}")
```

11.3. Проверка существования и поддержки

Одной из наиболее обширных областей, охватываемых модулями CMake, является проверка существования или поддержки различных вещей. Все модули этого семейства работают принципиально одинаково: пишут небольшой тестовый код, а затем пытаются скомпилировать и, возможно, связать и запустить полученный исполняемый файл, чтобы подтвердить, поддерживается ли то, что проверяется в коде. Все эти модули имеют название, начинающееся с Check.

Некоторые из наиболее фундаментальных модулей Check... - это те, которые компилируют и компонуют короткий тестовый файл в исполняемый файл и возвращают результат успеха/неудачи. Имена этих модулей имеют форму Check<LANG>SourceCompiles, и каждый из них предоставляет связанный макрос для выполнения теста:

```
include(CheckCSourceCompiles)
check_c_source_compiles(code resultVar [FAIL_REGEX regex])

include(CheckCXXSourceCompiles)
check_cxx_source_compiles(code resultVar [FAIL_REGEX regex])

include(CheckFortranSourceCompiles)
check_fortran_source_compiles(code resultVar [FAIL_REGEX regex] [SRC_EXT extension])
```

Для каждого из макросов аргумент code должен быть строкой, содержащей исходный код, который должен создать исполняемый файл для выбранного языка. Результат попытки компиляции и компоновки кода хранится в resultVar как кэш-переменная, причем true означает успех. Ложные значения могут быть пустой строкой, сообщением об ошибке и т.д. в зависимости от ситуации. После того, как тест был выполнен один раз, последующие запуски CMake будут использовать кэшированный результат, а не выполнять тест заново. Это происходит, даже если тестируемый код был изменен, поэтому для принудительной повторной оценки необходимо вручную удалить переменную из кэша. Если указана опция FAIL_REGEX, то применяется дополнительный критерий. Если результат компиляции и линковки теста совпадает с регулярным выражением regex, то проверка будет считаться неудачной, даже если код компилируется и линкуется успешно.

```
include(CheckCSourceCompiles)
check_c_source_compiles("
    int main(int argc, char* argv[])
    {
        int myVar;
        return 0;
    }" noWarnUnused FAIL_REGEX "[Ww]arn")
if(noWarnUnused)
    message("Unused variables do not generate warnings by default")
endif()
```

В случае Fortran расширение файла может влиять на то, как компиляторы обрабатывают исходные файлы, поэтому расширение файла может быть явно указано с помощью опции SRC_EXT для получения ожидаемого поведения. Для случаев C или C++ эквивалентной опции нет.

Ряд переменных вида CMAKE_REQUIRED_... может быть установлен перед вызовом любого из макросов проверки компиляции, чтобы повлиять на то, как они компилируют код:

CMAKE_REQUIRED_FLAGS

Дополнительные флаги для передачи в командную строку компилятора после содержимого соответствующего параметра

переменные CMAKE_<LANG>_FLAGS и CMAKE_<LANG>_FLAGS_<CONFIG> (см. раздел 14.3, "Переменные компилятора и компоновщика"). Это должна быть одна строка с несколькими флагами, разделенными пробелами, в отличие от всех остальных переменных ниже, которые являются списками CMake.

CMAKE_REQUIRED_DEFINITIONS

Список CMake определений компилятора, каждое из которых задано в форме -DFOO или -DFOO=bar.

CMAKE_REQUIRED_INCLUDES

Указывает каталоги для поиска заголовков. Несколько путей должны быть указаны в виде списка CMake, при этом пробелы рассматриваются как часть пути.

CMAKE_REQUIRED_LIBRARIES

Список библиотек CMake для добавления на этапе компоновки. Не добавляйте к именам библиотек опцию -l или аналогичную, укажите только имя библиотеки или имя импортированной CMake цели (обсуждается в главе 16, *Типы целей*).

CMAKE_REQUIRED_QUIET

Если эта опция присутствует, макрос не будет печатать сообщения о состоянии.

Эти переменные используются для построения аргументов внутреннего вызова try_compile() для выполнения проверки. В документации CMake для try_compile() обсуждаются дополнительные переменные, которые могут влиять на проверку, а другие аспекты поведения try_compile(), связанные с выбором цепочки инструментов, рассматриваются в разделе 21.5, "Проверки компилятора".

Помимо проверки возможности сборки кода, CMake также предоставляет модули для проверки возможности успешного выполнения кода на C или C++. Успех измеряется кодом выхода исполняемого файла, созданного из предоставленного исходного текста, при этом 0 считается успехом, а все остальные значения означают

неудачу. Модули имеют структуру, схожую со структурой модуля компиляции, каждый из них предоставляет один макрос, реализующий проверку:

```
include(CheckCSourceRuns)
check_c_source_runs(code resultVar)

include(CheckCXXSourceRuns)
check_cxx_source_runs(code resultVar)
```

Для этих макросов нет опции FAIL_REGEX, так как успех или неудача определяется исключительно кодом завершения процесса тестирования. Если код не может быть собран, это также рассматривается как неудача. Все те же переменные, которые влияют на сборку кода для check_c_source_compiles() и check_cxx_source_compiles(), также влияют и на макросы этих двух модулей.

Для сборок с кросс-компиляцией на другую целевую платформу макросы check_c_source_runs() и check_cxx_source_runs() ведут себя совершенно по-разному. Они могут запустить код под симулятором, если были предоставлены необходимые данные, что, вероятно, значительно замедлит выполнение этапа CMake. Если детали симулятора не были предоставлены, макросы будут ожидать предопределенного результата через набор переменных и не будут пытаться ничего запускать. Эта довольно сложная тема рассматривается в документации CMake по команде try_run(), которую макросы используют для внутренних проверок.

Некоторые категории проверок настолько распространены, что CMake предоставляет для них специальные модули. Они устраняют большую часть шаблонов определения кода проверки и позволяют проектам указывать минимальный набор информации для проверки. Обычно это просто обертки вокруг макросов, предоставляемых одним из модулей Check<LANG>SourceCompiles, так что используется тот же набор переменных, что и для настройки построения кода проверки. Эти более специализированные модули проверяют флаги компилятора, символы препроцессора, функции, переменные, заголовочные файлы и многое другое.

Поддержка определенных флагов компилятора может быть проверена с помощью модулей Check<LANG>CompilerFlag, каждый из которых предоставляет единственный макрос с именем, следующим предсказуемому шаблону:

```
include(CheckCCompilerFlag)
check_c_compiler_flag(flag resultVar)

include(CheckCXXCompilerFlag)
check_cxx_compiler_flag(flag resultVar)

include(CheckFortranCompilerFlag)
check_fortran_compiler_flag(flag resultVar)
```

Макросы проверки флагов обновляют внутреннюю переменную CMAKE_REQUIRED_DEFINITIONS, чтобы включить флаг в вызов соответствующего макроса check_<LANG>_source_compiles() с тривиальным тестовым файлом. В качестве опции FAIL_REGEX также передается внутренний набор регулярных выражений отказа, проверяющий, приводит ли флаг к выдаче диагностического сообщения или нет. Результатом вызова будет значение true, если диагностическое сообщение не было выдано. Обратите внимание, что это означает, что любой флаг, который приводит к предупреждению компилятора, но успешно компилируется, все равно будет считаться не прошедшим проверку. Также следует помнить, что эти макросы предполагают, что любые флаги, уже присутствующие в соответствующих переменных CMAKE_<LANG>_FLAGS (см. раздел 14.3, "Переменные

компилятора и компоновщика"), сами по себе не выдают предупреждений компилятора. Если это так, то логика каждого из этих макросов проверки флагов будет нарушена, и результатом всех таких проверок будет неудача.

Еще два примечательных модуля - CheckSymbolExists и CheckCXXSymbolExists. Первый предоставляет макрос, который создает тестовый исполняемый файл на языке C, а второй делает то же самое с исполняемым файлом на C++. Оба проверяют, существует ли определенный символ как символ препроцессора (т.е. то, что может быть проверено с помощью оператора #ifdef), функция или переменная.

```
include(CheckSymbolExists)
check_symbol_exists(symbol headers resultVar)

include(CheckCXXSymbolExists)
check_cxx_symbol_exists(symbol headers resultVar)
```

Для каждого из элементов, указанных в заголовках (список CMake, если необходимо указать более одного заголовка), в исходный код теста будет добавлен соответствующий #include. В большинстве случаев проверяемый символ будет определен одним из этих заголовков. Результат проверки сохраняется в кэш-переменной resultVar обычным способом.

В случае с функциями и переменными символ должен разрешаться во что-то, что является частью исполняемого файла теста. Если функция или переменная предоставляется библиотекой, эта библиотека должна быть подключена как часть теста, что можно сделать с помощью переменной CMAKE_REQUIRED_LIBRARIES.

```
include(CheckSymbolExists)
check_symbol_exists(sprintf stdio.h HAVE_SPRINTF)

include(CheckCXXSymbolExists)
set(CMAKE_REQUIRED_LIBRARIES SomeCxxSDK)
check_cxx_symbol_exists(SomeCxxInitFunc somecxxsdk.h HAVE_SOMECXXSDK)
```

Существуют ограничения на тип функций и переменных, которые могут быть проверены этими макросами. Можно использовать только те символы, которые удовлетворяют требованиям именования для символа препроцессора. Последствия для check_cxx_symbol_exists() сильнее, так как это означает, что проверять можно только нешаблонные функции или переменные в глобальном пространстве имен, потому что любой скопинг (::) или маркеры шаблона (<>) не будут действительны для символа препроцессора. Также невозможно отличить различные перегрузки одной и той же функции, поэтому они также не могут быть проверены.

Существуют и другие модули, цель которых - обеспечить функциональность, сходную с функциями CheckSymbolExists или являющуюся их подмножеством. Эти модули либо относятся к более ранним версиям CMake, либо предназначены для языков, отличных от C или C++. Модуль CheckFunctionExists уже задокументирован как устаревший, а модуль CheckVariableExists не предлагает ничего такого, чего бы уже не предлагал CheckSymbolExists. Модуль CheckFortranFunctionExists может быть полезен для проектов, работающих с Fortran, но обратите внимание, что модуля CheckFortranVariableExists не существует. Fortran-проекты могут захотеть использовать CheckFortranSourceCompiles для обеспечения согласованности.

Более детальные проверки обеспечиваются другими модулями. Члены структуры могут быть проверены с помощью

CheckStructHasMember, конкретные прототипы функций C или C++ можно проверить с помощью CheckPrototypeDefinition, а размер непользовательских типов можно проверить с помощью CheckTypeSize. Возможны и другие проверки более высокого уровня, такие как CheckLanguage, CheckLibraryExists и различные

модули CheckIncludeFile.... По мере развития CMake в него продолжают добавляться дополнительные модули проверки, поэтому обратитесь к документации по модулям CMake, чтобы ознакомиться с полным набором доступных функций.

В ситуациях, когда выполняется несколько проверок или когда эффекты выполнения проверок должны быть изолированы друг от друга или от остальной части текущей области видимости, может быть обременительно вручную сохранять и восстанавливать состояние до и после проверок. В частности, часто требуется сохранять и восстанавливать различные переменные CMAKE_REQUIRED_.... Чтобы помочь в этом, CMake предоставляет модуль CMakePushCheckState, который определяет следующие три макроса:

```
cmake_push_check_state([RESET])
cmake_pop_check_state()
cmake_reset_check_state()
```

Эти макросы позволяют рассматривать различные переменные CMAKE_REQUIRED_... как набор и выталкивать и выгружать их состояние в/из виртуального стека. При каждом вызове cmake_push_check_state() начинается новая область виртуальных переменных только для переменных CMAKE_REQUIRED_... (а также переменной CMAKE_EXTRA_INCLUDE_FILES, которая используется только модулем CheckTypeSize). cmake_pop_check_state() наоборот, сбрасывает текущие значения переменных CMAKE_REQUIRED_... и восстанавливает их значения на предыдущем уровне стека. Макрос cmake_reset_check_state() удобен для очистки всех переменных CMAKE_REQUIRED_..., а опция RESET в cmake_push_check_state() также удобна для очистки переменных в рамках push. Однако обратите внимание, что до версии CMake 3.10 существовала ошибка, в результате которой опция RESET игнорировалась, поэтому для проектов, которым необходимо работать с версиями до 3.10, лучше использовать отдельный вызов cmake_reset_check_state().

```
include(CheckSymbolExists)
include(CMakePushCheckState)

# Start with a known state we can modify and undo later
cmake_push_check_state()   # Could use RESET option, but needs CMake >= 3.10
cmake_reset_check_state()  # Separate call, safe for all CMake versions
set(CMAKE_REQUIRED_FLAGS -Wall)
check_symbol_exists(FOO_VERSION foo/version.h HAVE_FOO)

if(HAVE_FOO)
  # Preserve -Wall and add more things for extra checks
  cmake_push_check_state()
  set(CMAKE_REQUIRED_INCLUDES foo/inc.h foo/more.h)
  set(CMAKE_REQUIRED_DEFINES -DFOOBXX=1)
  check_symbol_exists(FOOBAR "" HAVE_FOOBAR)
  check_symbol_exists(FOOBAZ "" HAVE_FOOBAZ)
  check_symbol_exists(FOOBOO "" HAVE_FOOBOO)
  cmake_pop_check_state()
  # Now back to just -Wall
endif()

# Clear all the CMAKE_REQUIRED_... variables for this last check
cmake_reset_check_state()
check_symbol_exists(__TIME__ "" HAVE_PPTIME)

# Restore all CMAKE_REQUIRED_... variables to their original values
# from the top of this example
cmake_pop_check_state()
```

11.4. Другие модули

CMake имеет отличную встроенную поддержку некоторых языков, особенно C и C++. Он также включает ряд модулей, которые обеспечивают поддержку языков в более расширяемом и настраиваемом виде. Эти модули позволяют сделать аспекты некоторых языков или пакетов, связанных с языками, доступными для проектов путем определения соответствующих функций, макросов, переменных и свойств. Многие из этих модулей предоставляются как часть поддержки вызовов find_package() (см. раздел 23.5, "Поиск пакетов"), в то время как другие предназначены для более прямого использования с помощью include(), чтобы ввести вещи в текущую область видимости. Приведенный ниже список модулей должен дать представление о доступной языковой поддержке:

- CSharpUtilities
- FindCUDA (но обратите внимание, что в последних версиях CMake эта функция была заменена поддержкой CUDA как самостоятельного языка первого класса).
- FindJava, FindJNI, UseJava
- FindLua
- FindMatlab
- FindPerl, FindPerlLibs
- FindPython, FindPythonInterp
- FindPHP4
- FindRuby
- НайтиСВИГ, ИспользоватьСВИГ
- FindTCL
- FortranCInterface

Кроме того, предусмотрены модули для взаимодействия с внешними данными и проектами, эта тема подробно рассматривается в главе 27 "*Внешнее содержимое*". Также предоставляется ряд модулей для облегчения различных аспектов тестирования и упаковки. Они тесно связаны с инструментами CTest и CPack, распространяемыми как часть пакета CMake, и подробно рассматриваются в главе 24 "*Тестирование*" и главе 26 "*Упаковка*".

11.5. Рекомендуемые практики

Коллекция модулей CMake предоставляет множество функциональных возможностей, построенных поверх основного языка CMake. Проект может легко расширить набор доступных функций, добавив свои собственные модули в определенный каталог и добавив этот путь в переменную CMAKE_MODULE_PATH. Использование CMAKE_MODULE_PATH предпочтительнее, чем жесткое кодирование абсолютных или относительных путей в сложных структурах каталогов в вызовах include(), так как это способствует отделению общей логики CMake от мест, где эта логика может быть применена. Это, в свою очередь, облегчает перемещение модулей CMake в разные каталоги по мере развития проекта или повторное использование логики в разных проектах. Действительно, нередко организация создает собственную коллекцию модулей, возможно, даже храня их в отдельном репозитории. Установив в каждом проекте соответствующее значение CMAKE_MODULE_PATH, эти многократно используемые строительные блоки CMake становятся доступными для широкого использования по мере необходимости.

Со временем разработчик, как правило, сталкивается со все большим количеством интересных сценариев, для которых модуль CMake может предоставить полезные сокращения или готовые решения. Иногда быстрое сканирование доступных модулей может дать неожиданную скрытую жемчужину, или новый модуль может предложить лучшую реализацию того, что проект до этого момента реализовывал некачественно. Преимущество модулей CMake заключается в потенциально большом количестве разработчиков и проектов, использующих их на различных платформах и в различных ситуациях, поэтому во многих случаях они могут быть более привлекательной альтернативой проектам, реализующим свою собственную логику вручную. Качество, однако, варьируется от одного модуля к другому. Некоторые модули начали свою жизнь довольно рано в CMake, и иногда они могут стать менее полезными, если не следить за изменениями в CMake или в областях, к которым эти модули относятся. Это может быть особенно верно для модулей Find..., которые могут не так тщательно отслеживать новые версии пакетов, которые они находят. С другой стороны, модули - это обычный код CMake, поэтому любой может изучать их, учиться на них, улучшать или обновлять, не изучая ничего сверх того, что необходимо для базового использования CMake в проекте. Фактически, они являются отличной отправной точкой для разработчиков, желающих приступить к работе над самим CMake.

Обилие различных модулей Check..., предоставляемых CMake, может быть неоднозначным благословением. У разработчиков может возникнуть соблазн слишком усердно проверять всевозможные вещи, что может привести к замедлению этапа конфигурирования ради иногда сомнительной выгоды. Подумайте, перевешивают ли преимущества затраты времени на внедрение и поддержку проверок и сложность проекта. Иногда нескольких разумных проверок достаточно для покрытия наиболее полезных случаев или для выявления тонкой проблемы, которая в противном случае может привести к трудноотслеживаемым проблемам в дальнейшем. Кроме того, при использовании любого из модулей Check... старайтесь изолировать логику проверки от области, в которой она может быть вызвана. Настоятельно рекомендуется использовать модуль CMakePushCheckState, но избегайте использования опции RESET в cmake_push_check_state(), если важна поддержка версий CMake до 3.10.

Глава 12. Политика

CMake развивался в течение длительного периода времени, внедряя новые функциональные возможности, исправляя ошибки и изменяя поведение некоторых функций для устранения недостатков или внесения улучшений. Хотя введение новых возможностей вряд ли вызовет проблемы для существующих проектов, созданных с помощью CMake, любое изменение поведения может привести к поломке проектов, если они полагаются на старое поведение. По этой причине разработчики CMake тщательно следят за тем, чтобы изменения были реализованы таким образом, чтобы сохранить обратную совместимость и обеспечить прямой, контролируемый путь миграции для обновления проектов до нового поведения.

Контроль над тем, какое поведение следует использовать - старое или новое - осуществляется с помощью механизмов политики CMake. В целом, политики - это не то, с чем разработчики сталкиваются часто, в основном только когда CMake выдает предупреждение о том, что проект полагается на поведение старой версии. Когда разработчики переходят на более свежий выпуск CMake, более новая версия CMake иногда выдает такие предупреждения, чтобы подчеркнуть, как проект должен быть обновлен для использования нового поведения.

12.1. Контроль политики

Функциональность политики CMake тесно связана с командой cmake_minimum_required(), которая была представлена еще в главе 3 "*Минимальный проект*". Эта команда не только указывает минимальную версию CMake, которая требуется проекту, но и устанавливает поведение CMake в соответствии с указанной версией. Таким образом, когда проект начинается с cmake_minimum_required(VERSION 3.2), это говорит о том, что требуется как минимум CMake 3.2, а также о том, что проект ожидает от CMake поведения, соответствующего релизу 3.2. Это дает проектам уверенность в том, что разработчики смогут обновить CMake до любой более новой версии в удобное для них время, и проект будет собираться так же, как и раньше.

Однако иногда в проекте может потребоваться более тонкий контроль, чем тот, который обеспечивает cmake_minimum_required(). Рассмотрим следующие сценарии:

- Проект хочет установить низкую минимальную версию CMake, но также хочет использовать преимущества более новых версий, если они доступны.
- Часть проекта не может быть изменена (например, она может быть получена из внешнего репозитория кода, доступного только для чтения), и она полагается на старое поведение, которое было изменено в новых версиях CMake. Однако остальная часть проекта хочет перейти на новое поведение.
- Проект сильно зависит от некоторого старого поведения, обновление которого потребует нетривиального объема работы. Некоторые части проекта хотят использовать новые возможности CMake, но старое поведение для этого конкретного изменения должно быть сохранено до тех пор, пока не будет выделено время для обновления проекта.

Это несколько распространенных примеров, когда одного лишь высокоуровневого контроля, обеспечиваемого командой cmake_minimum_required(), недостаточно. Более конкретный контроль над политиками осуществляется с помощью команды cmake_policy(), которая имеет несколько форм, действующих на разных уровнях детализации. Форма, действующая на самом грубом уровне, является близким родственником команды cmake_minimum_required():

```
cmake_policy(VERSION major[.minor[.patch[.tweak]]])
```

В этой форме команда изменяет поведение CMake в соответствии с указанной версией. Команда cmake_minimum_required() неявно вызывает эту форму для установки поведения CMake. Эти две команды в основном взаимозаменяемы, за исключением верхней части проекта, где вызов cmake_minimum_required()

является обязательным для обеспечения минимальной версии CMake. Помимо начала файла CMakeLists.txt верхнего уровня, использование cmake_policy() обычно более четко передает намерения, когда проекту нужно обеспечить поведение определенной версии для части проекта, как показано в следующем примере:

```
cmake_minimum_required(VERSION 3.7)
project(WithLegacy)

# Uses recent CMake features
add_subdirectory(modernDir)

# Imported from another project, relies on old behavior
cmake_policy(VERSION 2.8.11)
add_subdirectory(legacyDir)
```

CMake 3.12 расширяет эту возможность обратно совместимым образом, позволяя проекту указывать *диапазон* версий, а не одну версию в cmake_minimum_required() или cmake_policy(VERSION). Диапазон задается с помощью трех точек ... между минимальной и максимальной версией без пробелов. Диапазон указывает, что используемая версия CMake должна быть не меньше минимальной, а используемая версия должна быть меньше указанной максимальной и текущей версии CMake. Это позволяет проекту эффективно сказать: "Мне нужен как минимум CMake X, но я могу безопасно использовать политики от до CMake Y". В следующем примере показаны два способа, как проект может требовать только CMake 3.7, но при этом поддерживать более новое поведение для всех политик вплоть до CMake 3.12, если текущая версия CMake поддерживает их:

```
cmake_minimum_required(VERSION 3.7...3.12)
cmake_policy(VERSION 3.7...3.12)
```

Версии CMake до 3.12 будут видеть только один номер версии и игнорировать часть ...3.12, тогда как 3.12 и более поздние версии будут понимать, что это означает диапазон.

CMake также предоставляет возможность контролировать каждое изменение поведения по отдельности с помощью формы SET:

```
cmake_policy(SET CMPxxxx NEW)
cmake_policy(SET CMPxxxx OLD)
```

Каждому отдельному изменению поведения присваивается свой номер политики в виде CMPxxxx, где xxxx - всегда четыре цифры. Указывая NEW или OLD, проект указывает CMake на использование нового или старого поведения для данной политики. В документации CMake приводится полный список политик, а также объяснение поведения OLD и NEW для каждой из них.

Например, до версии 3.0 CMake позволял проекту вызывать get_target_property() с именем несуществующей цели. В этом случае значение свойства возвращалось как -NOTFOUND, а не выдавало ошибку, но, скорее всего, проект содержал неправильную логику. Поэтому, начиная с версии 3.0, при возникновении подобной ситуации CMake завершает работу с ошибкой. В случае, если проект полагался на старое поведение, он мог продолжать это делать, используя политику CMP0045 следующим образом:

```
# Allow non-existent target with get_target_property()
cmake_policy(SET CMP0045 OLD)

# Would halt with an error without the above policy change
get_target_property(outVar doesNotExist COMPILE_DEFINITIONS)
```

Необходимость установки политики NEW встречается реже. Одна из ситуаций - когда проект хочет установить низкую минимальную версию CMake, но при этом воспользоваться преимуществами более поздних функций, если используется более поздняя версия. Например, в CMake 3.2 была введена политика CMP0055 для обеспечения строгой проверки использования команды break(). Если проект по-прежнему хотел поддерживать сборку с более ранними версиями CMake, то дополнительные проверки должны были быть явно включены при запуске с более поздними версиями CMake.

```
cmake_minimum_required(VERSION 3.0)
project(PolicyExample)

if(CMAKE_VERSION VERSION_GREATER 3.1)
    # Enable stronger checking of break() command usage
    cmake_policy(SET CMP0055 NEW)
endif()
```

Проверка переменной CMAKE_VERSION является одним из способов определения наличия политики, но команда if() предоставляет более прямой способ, используя форму if(POLICY...). В качестве альтернативы вышеприведенное можно реализовать следующим образом:

```
cmake_minimum_required(VERSION 3.0)
project(PolicyExample)

# Only set the policy if the version of CMake being used
# knows about that policy number
if(POLICY CMP0055)
    cmake_policy(SET CMP0055 NEW)
endif()
```

Также можно получить текущее состояние определенной политики. Основная ситуация, когда может потребоваться чтение текущей настройки политики, - это файл модуля, который может быть предоставлен самим CMake или проектом. Однако было бы необычно, если бы модуль проекта изменил свое поведение на основе настроек политики.

```
cmake_policy(GET CMPxxxx outVar)
```

Значение, хранящееся в outVar, будет OLD, NEW или пустым. Команды cmake_minimum_required(VERSION...) и cmake_policy(VERSION...) сбрасывают состояние всех политик. Те политики, которые были введены в указанной версии CMake или ранее, сбрасываются в состояние NEW. Те политики, которые были добавлены после указанной версии, фактически сбрасываются в пустое состояние.

Если CMake обнаруживает, что проект делает что-то, что либо опирается на старое поведение, либо конфликтует с новым поведением, либо чье поведение неоднозначно, он может предупредить, что соответствующая политика не установлена. Эти предупреждения являются наиболее распространенным

способом ознакомления разработчиков с функциональностью CMake, связанной с политиками. Они предназначены для того, чтобы быть шумными, но информативными, побуждая разработчиков обновить проект в соответствии с новым поведением. В некоторых случаях предупреждение об устаревании может быть выдано, даже если политика была явно установлена, но это, как правило, относится только к политике, которая уже давно (много релизов) документирована как устаревшая.

CMAKE_POLICY_WARNING_CMP<NNN>) Иногда предупреждения политики не могут быть устранены немедленно, но предупреждения могут быть нежелательными. Предпочтительным способом решения этой проблемы является явная установка политики на желаемое поведение (OLD или NEW), что прекращает предупреждение. Однако это не всегда возможно, например, когда более глубокая часть проекта выполняет свой собственный вызов cmake_minimum_required(VERSION...) или cmake_policy(VERSION...), тем самым сбрасывая состояния политики.

В качестве временного способа обхода таких ситуаций CMake предоставляет функцию

Переменные CMAKE_POLICY_DEFAULT_CMPxxx и CMAKE_POLICY_WARNING_CMPxxx, где xxxx - обычный четырехзначный номер политики. Они не предназначены для установки проектом, а скорее разработчиком в качестве временной переменной кэша для включения/выключения предупреждения или для проверки, выдает ли проект предупреждения с включенной определенной политикой. В конечном счете, долгосрочным решением является устранение основной проблемы, на которую указывает предупреждение. Тем не менее, иногда в проекте может быть целесообразно установить одну из этих переменных, чтобы заглушить предупреждение, которое, как известно, не является вредным.

12.2. Сфера действия политики

Иногда настройки политики необходимо применить только к определенному разделу файла. Вместо того чтобы требовать от проекта вручную сохранять существующие значения политик, которые он хочет временно изменить, CMake предоставляет стек политик, который можно использовать для упрощения этого процесса:

```
cmake_policy(PUSH)
cmake_policy(POP)
```

Существующее состояние всех политик может быть сохранено с помощью операции PUSH, а текущее состояние отменено с помощью соответствующего POP. Каждый PUSH должен в конечном итоге иметь соответствующий POP. В промежутке между этими операциями проект может изменять настройки всех необходимых политик без необходимости явного сохранения каждой из них. Опять же, файлы модулей являются одним из наиболее распространенных мест, где стек политик может манипулироваться подобным образом. Простым примером может быть файл модуля, который временно устанавливает несколько политик следующим образом:

```
# Save existing policy state
cmake_policy(PUSH)

# Set some policies to OLD to preserve a few old behaviors
cmake_policy(SET CMP0060 OLD)  # Library path linking behavior
cmake_policy(SET CMP0021 OLD)  # Tolerate relative INCLUDE_DIRECTORIES

# Do various processing here...

# Restore earlier policy settings
cmake_policy(POP)
```

Некоторые команды неявно заносят новое состояние политики в стек и снова заносят его в определенный момент времени. Примером может служить команда add_subdirectory(), которая вводит новую область действия политики в стек при входе в указанный подкаталог и выводит ее снова, когда команда возвращается. Команда include() делает аналогичную вещь, выталкивая новую область действия политики перед началом обработки указанного файла и снова выталкивая ее, когда обработка этого файла завершена. Команда find_package() также делает то же самое, что и include(), выталкивая и выпрыгивая при начале и завершении обработки связанного с ней файла модуля FindXXX.cmake соответственно.

Команды include() и find_package() также поддерживают опцию NO_POLICY_SCOPE, которая предотвращает автоматический push-pop стека политик (add_subdirectory() не имеет такой опции). В очень ранних версиях CMake команды include() и find_package() автоматически не проталкивали и не разворачивали запись в стеке политик. Опция NO_POLICY_SCOPE была добавлена как способ для проектов, использующих более поздние версии CMake, вернуться к старому поведению для определенных частей проекта, но ее использование в целом не рекомендуется, а для новых проектов она должна быть излишней.

12.3. Рекомендуемые практики

По возможности, проекты должны предпочитать работать с политиками на уровне версий CMake, а не манипулировать конкретными политиками. Настройка политик в соответствии с поведением конкретного выпуска CMake облегчает понимание и обновление проекта, тогда как изменения отдельных политик может быть сложнее отследить через несколько уровней каталогов, особенно из-за их взаимодействия с изменениями политик на уровне версий, где они всегда сбрасываются.

При выборе способа указания версии CMake для соответствия, выбор между cmake_minimum_required(VERSION) и cmake_policy(VERSION) обычно падает на последнюю. Два основных исключения из этого правила - в начале файла CMakeLists.txt верхнего уровня проекта и в начале файла модуля, который может быть повторно использован в нескольких проектах. В последнем случае предпочтительнее использовать cmake_minimum_required(VERSION), поскольку проекты, использующие модуль, могут устанавливать свои собственные минимальные версии CMake, а модуль может иметь свои собственные требования к минимальной версии. За исключением этих случаев, cmake_policy(VERSION) обычно выражает намерение более четко, но обе команды фактически достигают одного и того же с точки зрения политики.

В случаях, когда проекту необходимо манипулировать определенной политикой, ему следует проверить, доступна ли эта политика, используя if(POLICY...), а не проверять переменную CMAKE_VERSION. Это приводит к большей согласованности кода. Сравните следующие два способа задания поведения политики и обратите внимание на то, как проверка и применение используют последовательный подход:

```
# Version-level policy enforcement
if(NOT CMAKE_VERSION VERSION_LESS 3.4)
    cmake_policy(VERSION 3.4)
endif()

# Individual policy-level enforcement
if(POLICY CMP0055)
    cmake_policy(SET CMP0055 NEW)
endif()
```

Если в проекте необходимо локально управлять несколькими отдельными политиками, окружите этот раздел вызовами cmake_policy(PUSH) и cmake_policy(POP), чтобы гарантировать, что остальная часть области видимости изолирована от изменений. Обратите особое внимание на все возможные операторы return(), которые выходят из этого участка кода, и убедитесь, что ни один push не остался без соответствующего pop.

Обратите внимание, что add_subdirectory(), include() и find_package() автоматически проталкивают и разворачивают запись в стеке политик, поэтому явное проталкивание и разворачивание не требуется, если только не нужно локально изменить настройки политики для небольшого участка извлекаемого файла. Проектам следует избегать ключевого слова NO_POLICY_SCOPE в этих командах, поскольку оно предназначено только для устранения изменений в поведении очень ранних версий CMake, и его использование редко подходит для новых проектов.

В крайнем случае, переменные CMAKE_POLICY_DEFAULT_CMPxxx и CMAKE_POLICY_WARNING_CMPxxx могут позволить разработчику или проекту обойти некоторые специфические ситуации, связанные с политикой. Разработчики могут использовать их для временного изменения значения по умолчанию определенного параметра политики или для предотвращения предупреждений о конкретной политике. Как правило, проекты должны избегать установки этих переменных, чтобы разработчики могли контролировать их локально, но в некоторых ситуациях они могут быть использованы для того, чтобы поведение или предупреждение о конкретной политике сохранялось даже при вызове cmake_minimum_required() или cmake_policy(VERSION). Там, где это возможно, проекты должны пытаться перейти на более новое поведение, а не полагаться на эти переменные.

Часть II: Сборки в глубину

В предыдущих главах были постепенно представлены наиболее фундаментальные аспекты CMake. Были представлены основные возможности языка, ключевые понятия и важные строительные блоки, обеспечивающие прочную основу для более глубокого изучения функциональности CMake.

В этой части книги основное внимание уделяется продуктам сборки. В главах рассматриваются инструментарий и конфигурация сборки, различные типы целей, выполнение пользовательских задач и работа с особенностями, специфичными для платформы. Хорошее понимание этих областей может стать разницей между хрупким, сложным проектом и надежным, простым в сопровождении.

Глава 13. Тип сборки

В этой и следующей главах рассматриваются две тесно связанные темы. Тип сборки (также известный как конфигурация сборки или схема сборки в некоторых инструментах IDE) - это элемент управления высокого уровня, который выбирает различные наборы поведения компилятора и компоновщика. Манипуляции с типом сборки являются темой этой главы, а в следующей главе представлены более конкретные детали управления опциями компилятора и компоновщика. Вместе эти главы охватывают материал, который каждый разработчик CMake обычно использует для всех, кроме самых тривиальных проектов.

13.1. Основы типа сборки

Тип сборки способен так или иначе повлиять практически на все, что связано со сборкой. В первую очередь он оказывает прямое влияние на поведение компилятора и компоновщика, но также влияет на структуру каталогов, используемых для проекта. Это, в свою очередь, может повлиять на то, как разработчик устанавливает свою локальную среду разработки, поэтому влияние типа сборки может быть весьма обширным.

Разработчики обычно думают о сборках как об одной из двух форм: отладочной или релизной. В отладочной сборке флаги компилятора используются для записи информации, которую отладчики могут использовать для связывания машинных инструкций с исходным кодом. Оптимизация часто отключается в таких сборках, чтобы сопоставление машинных инструкций и мест исходного кода было прямым и легко отслеживалось при выполнении программы. В сборке релиза, с другой стороны, обычно полностью включены оптимизации и не генерируется отладочная информация.

Это примеры того, что CMake называет *типом сборки*. Хотя проекты могут определять любые типы сборки по своему усмотрению, типов сборки по умолчанию, предоставляемых CMake, обычно достаточно для большинства проектов:

Debug

Без оптимизаций и с полной отладочной информацией, этот вариант обычно используется во время разработки и отладки, так как он обычно обеспечивает самое быстрое время сборки и лучший интерактивный опыт отладки.

Release

Этот тип сборки обычно обеспечивает полную оптимизацию для скорости и не содержит отладочной информации, хотя некоторые платформы могут генерировать отладочные символы в определенных обстоятельствах. Обычно этот тип сборки используется при сборке программного обеспечения для финальных производственных релизов.

RelWithDebInfo

Это своего рода компромисс между двумя предыдущими. Его цель - обеспечить производительность, близкую к производительности сборки Release, но при этом позволить некоторый уровень отладки. Как правило, применяется большинство оптимизаций для повышения скорости, но большинство функций отладки также включены. Поэтому этот тип сборки наиболее полезен, когда производительность сборки Debug неприемлема даже для отладочной сессии. Обратите внимание, что настройки по умолчанию для RelWithDebInfo отключают утверждения.

MinRizeRel

Этот тип сборки обычно используется только для сред с ограниченными ресурсами, таких как встроенные устройства. Код оптимизируется для размера, а не для скорости, и отладочная информация не создается.

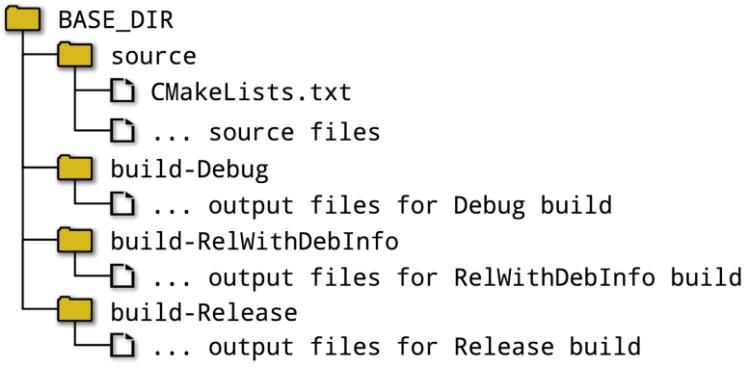
Каждый тип сборки приводит к различному набору флагов компилятора и компоновщика. Он также может изменять другие поведения, например, изменять, какие исходные файлы компилируются или какие библиотеки компонуются. Эти детали будут рассмотрены в следующих нескольких разделах, но прежде чем приступить к обсуждению, необходимо понять, как выбрать тип сборки и как избежать некоторых распространенных проблем.

13.1.1. Генераторы одиночной конфигурации

В разделе 2.3, "Генерация файлов проекта", были представлены различные типы генераторов проектов. Некоторые, такие как Makefiles и Ninja, поддерживают только один тип сборки для каждого каталога сборки. Для этих генераторов тип сборки должен быть выбран путем установки кэш-переменной CMAKE_BUILD_TYPE. Например, для конфигурирования и последующей сборки проекта с помощью Ninja можно использовать такие команды, как:

```
cmake -G Ninja -DCMAKE_BUILD_TYPE:STRING=Debug ../source
cmake --build .
```

Переменную кэша CMAKE_BUILD_TYPE можно также изменить в приложении CMake GUI, а не из командной строки, но конечный эффект будет тот же. Однако вместо того, чтобы переключаться между различными типами сборки, альтернативной стратегией является создание отдельных каталогов сборки для каждого типа сборки с использованием одних и тех же исходных текстов. Такая структура каталогов может выглядеть следующим образом:



При частом переключении между типами сборок такая схема позволяет избежать необходимости постоянно перекомпилировать одни и те же исходные тексты только потому, что изменились флаги компилятора. Это также позволяет одному генератору конфигурации эффективно действовать как генератор нескольких конфигураций, при этом среды IDE, такие как Qt Creator, поддерживают переключение между каталогами сборки так же легко, как Xcode или Visual Studio позволяют переключаться между схемами сборки или конфигурациями.

13.1.2. Несколько генераторов конфигурации

Некоторые генераторы, в частности Xcode и Visual Studio, поддерживают несколько конфигураций в одном каталоге сборки. Эти генераторы игнорируют переменную кэша CMAKE_BUILD_TYPE и вместо этого требуют от разработчика выбрать тип сборки в IDE или с помощью опции командной строки во время сборки. Конфигурирование и сборка таких проектов будет выглядеть примерно так:

```
cmake -G Xcode ../source
cmake --build . --config Debug
```

При сборке в среде Xcode IDE тип сборки контролируется схемой сборки, а в среде Visual Studio IDE тип сборки определяется текущей конфигурацией решения. Обе среды хранят отдельные каталоги для различных типов сборки, поэтому переключение между сборками не приводит к постоянным пересборкам. По сути, делается то же самое, что и в случае с несколькими директориями сборки, описанном выше для генераторов единой конфигурации, просто IDE управляет структурой директорий от имени разработчика.

13.2. Общие ошибки

Обратите внимание, что для генераторов с одной конфигурацией тип сборки задается во время *конфигурирования, в* то время как для генераторов с несколькими конфигурациями тип сборки задается во время *сборки*. Это различие очень важно, поскольку оно означает, что тип сборки не всегда известен, когда CMake обрабатывает файл CMakeLists.txt проекта. Рассмотрим следующий фрагмент кода CMake, который, к сожалению, встречается довольно часто, но демонстрирует неправильную схему:

```
# WARNING: Do not do this!
if(CMAKE_BUILD_TYPE STREQUAL "Debug")
    # Do something only for debug builds
endif()
```

Вышеописанное будет хорошо работать для генераторов на основе Makefile и Ninja, но не для Xcode или Visual Studio. На практике, практически любая логика, основанная на CMAKE_BUILD_TYPE в проекте, сомнительна, если она не защищена проверкой, подтверждающей, что используется генератор с одной конфигурацией. Для генераторов нескольких конфигураций эта переменная, скорее всего, будет пустой, но даже если это не так, ее значение следует считать ненадежным, поскольку сборка будет его игнорировать. Вместо ссылки на CMAKE_BUILD_TYPE в файле CMakeLists.txt проекты должны использовать другие более надежные альтернативные методы, такие как выражения генераторов, основанные на $<CONFIG:...>.

При создании сценариев сборки распространенным недостатком является предположение об использовании определенного генератора или неправильный учет различий между генераторами с одной и несколькими конфигурациями. В идеале разработчики должны иметь возможность изменить генератор в одном месте, а остальная часть сценария должна по-прежнему функционировать правильно. Удобно, что генераторы с одной конфигурацией игнорируют любые спецификации времени сборки, а генераторы с несколькими конфигурациями игнорируют переменную CMAKE_BUILD_TYPE, поэтому, указав оба параметра, скрипт может учесть оба случая. Например:

```
mkdir build
cd build
cmake -G Ninja -DCMAKE_BUILD_TYPE=Release ../source
cmake --build . --config Release
```

В приведенном выше примере разработчик может просто изменить имя генератора, указанное в параметре -G, и остальная часть сценария будет работать без изменений.

Отсутствие явной установки CMAKE_BUILD_TYPE для генераторов одиночной конфигурации также часто встречается, но обычно это не то, что задумал разработчик. Поведение, уникальное для генераторов одиночной конфигурации, заключается в том, что если CMAKE_BUILD_TYPE не задан, то тип сборки пуст. Это иногда приводит к неправильному пониманию некоторыми разработчиками того, что пустой тип сборки эквивалентен Debug, но это не так. Пустой тип сборки - это собственный уникальный, безымянный тип сборки. В таких случаях не используются специфические для конфигурации флаги компилятора или компоновщика, что часто просто приводит к вызову компилятора и компоновщика с минимальными флагами, и поэтому

поведение определяется собственным поведением компилятора и компоновщика по умолчанию. Хотя это часто может быть похоже на поведение типа сборки Debug, это ни в коем случае не гарантировано.

13.3. Пользовательские типы сборки

Иногда проект может захотеть ограничить набор типов сборки подмножеством типов по умолчанию или добавить другие пользовательские типы сборки со специальным набором флагов компилятора и компоновщика. Хорошим примером последнего является добавление типа сборки для профилирования или покрытия кода, оба из которых требуют специальных настроек компилятора и компоновщика.

Существует два основных места, где разработчик может увидеть набор типов сборки. При использовании генераторов нескольких конфигураций, таких как Xcode и Visual Studio, среда IDE предоставляет выпадающий список или аналогичный список, из которого разработчик выбирает конфигурацию, которую он хочет собрать. Для генераторов одиночных конфигураций, таких как Makefiles или Ninja, тип сборки вводится непосредственно для кэш-переменной CMAKE_BUILD_TYPE, но приложение CMake GUI может быть сделано так, чтобы вместо простого текстового поля редактирования отображался комбинированный список допустимых вариантов. Механизмы, лежащие в основе этих двух случаев, различны, поэтому их следует обрабатывать отдельно.

Набор типов сборки, известных генераторам мультиконфигурации, контролируется параметром CMAKE_CONFIGURATION_TYPES кэш-переменной, или, точнее, значением этой переменной в конце обработки файла CMakeLists.txt верхнего уровня. Первая встреченная команда project() заполняет кэш-переменную списком по умолчанию, если он еще не определен, но проекты могут изменять одноименную некэш-переменную после этого момента (изменение кэш-переменной небезопасно, так как может отменить изменения, сделанные разработчиком). Пользовательские типы сборки могут быть определены путем добавления их в CMAKE_CONFIGURATION_TYPES, а ненужные типы сборки могут быть удалены из этого списка.

Однако нужно быть осторожным, чтобы не установить CMAKE_CONFIGURATION_TYPES, если он еще не определен. До выхода CMake 3.9 очень распространенным подходом для определения того, используется ли генератор мультиконфигурации, была проверка того, что CMAKE_CONFIGURATION_TYPES не является пустым. Даже некоторые части самого CMake делали это до версии 3.11. Хотя этот метод обычно является точным, нередко встречаются проекты, в которых CMAKE_CONFIGURATION_TYPES устанавливается в одностороннем порядке, даже если используется один генератор конфигурации. Это может привести к принятию неверных решений относительно типа используемого генератора. Для решения этой проблемы в CMake 3.9 было добавлено новое глобальное свойство GENERATOR_IS_MULTI_CONFIG, которое устанавливается в true, если используется генератор с несколькими конфигурациями, предоставляя окончательный способ получения этой информации вместо того, чтобы полагаться на вывод из CMAKE_CONFIGURATION_TYPES. Несмотря на это, проверка CMAKE_CONFIGURATION_TYPES все еще является настолько распространенным шаблоном, что проекты должны продолжать изменять его только в том случае, если он существует, и никогда не создавать его самостоятельно. Следует также отметить, что до CMake 3.11 добавление пользовательских типов сборки в CMAKE_CONFIGURATION_TYPES было технически небезопасным. Определенные части CMake учитывали только типы сборки по умолчанию, но даже в этом случае проекты все еще могут с пользой определять пользовательские типы сборки в более ранних версиях CMake, в зависимости от того, как они будут использоваться. Тем не менее, для большей надежности рекомендуется использовать как минимум CMake 3.11, если планируется определять пользовательские типы сборки.

Другой аспект этой проблемы заключается в том, что разработчики могут добавлять свои собственные типы в CMAKE_CONFIGURATION_TYPES кэш-переменную и/или удалить некоторые из тех, которые их не интересуют. Поэтому проекты не должны делать никаких предположений о том, какие типы конфигурации определены, а какие нет.

Принимая во внимание вышеизложенные моменты, следующий шаблон показывает предпочтительный способ для проектов добавить свои собственные пользовательские типы сборки для генераторов нескольких конфигураций:

```
cmake_minimum_required(3.11)
project(Foo)

if(CMAKE_CONFIGURATION_TYPES)
    if(NOT "Profile" IN_LIST CMAKE_CONFIGURATION_TYPES)
        list(APPEND CMAKE_CONFIGURATION_TYPES Profile)
    endif()
endif()

# Set relevant Profile-specific flag variables if not already set...
```

Для генераторов с одной конфигурацией существует только один тип сборки, который задается параметром CMAKE_BUILD_TYPE кэш-переменная, которая является строкой. В графическом интерфейсе CMake она обычно представлена в виде текстового поля для редактирования, поэтому разработчик может изменить ее так, чтобы она содержала любое произвольное содержимое. Однако, как уже говорилось в разделе 9.6, "Свойства переменных кэша", переменные кэша могут иметь свойство STRINGS, определенное для хранения набора допустимых значений. Тогда приложение CMake GUI будет представлять эту переменную не в виде текстового поля редактирования, а в виде комбинированного окна, содержащего допустимые значения.

```
set_property(CACHE CMAKE_BUILD_TYPE PROPERTY
             STRINGS Debug Release Profile)
```

Свойства могут быть изменены только внутри файлов CMakeLists.txt проекта, поэтому можно смело устанавливать свойство STRINGS, не беспокоясь о сохранении изменений разработчика. Обратите внимание, однако, что установка свойства STRINGS для переменной кэша не гарантирует, что переменная кэша будет содержать одно из определенных значений, а только управляет тем, как переменная будет представлена в приложении CMake GUI. Разработчики по-прежнему могут установить CMAKE_BUILD_TYPE на любое значение в командной строке cmake или отредактировать файл CMakeCache.txt вручную. Для того чтобы строго потребовать, чтобы переменная имела одно из определенных значений, проект должен сам явно выполнить этот тест.

```
set(allowableBuildTypes Debug Release Profile)

# WARNING: This logic is not sufficient
if(NOT CMAKE_BUILD_TYPE IN_LIST allowableBuildTypes)
    message(FATAL_ERROR "${CMAKE_BUILD_TYPE} is not a defined build type")
endif()
```

Значением по умолчанию для CMAKE_BUILD_TYPE является пустая строка, поэтому приведенное выше значение приведет к фатальной ошибке как для генераторов с одной, так и с несколькими конфигурациями, если только разработчик не задаст его явно. Это нежелательно, особенно для генераторов с несколькими конфигурациями, которые даже не используют значение переменной CMAKE_BUILD_TYPE. С этим можно справиться, предоставив проекту значение по умолчанию, если CMAKE_BUILD_TYPE не была установлена. Более того, техники для генераторов с несколькими и одной конфигурацией могут и должны быть объединены, чтобы обеспечить надежное поведение для всех типов генераторов. Конечный результат будет выглядеть примерно так:

```
cmake_minimum_required(3.11)
project(Foo)

if(CMAKE_CONFIGURATION_TYPES)
    if(NOT "Profile" IN_LIST CMAKE_CONFIGURATION_TYPES)
        list(APPEND CMAKE_CONFIGURATION_TYPES Profile)
    endif()
else()
    set(allowableBuildTypes Debug Release Profile)
    set_property(CACHE CMAKE_BUILD_TYPE PROPERTY
                 STRINGS "${allowableBuildTypes}")
    if(NOT CMAKE_BUILD_TYPE)
        set(CMAKE_BUILD_TYPE Debug CACHE STRING "" FORCE)
    elseif(NOT CMAKE_BUILD_TYPE IN_LIST allowableBuildTypes)
        message(FATAL_ERROR "Invalid build type: ${CMAKE_BUILD_TYPE}")
    endif()
endif()

# Set relevant Profile-specific flag variables if not already set...
```

Все рассмотренные выше методы просто позволяют выбрать пользовательский тип сборки, они ничего не определяют в отношении этого типа сборки. По сути, когда тип сборки выбран, он определяет, какие специфические для конфигурации переменные должен использовать CMake, а также влияет на любые выражения генератора, логика которых зависит от текущей конфигурации (т.е. $<CONFIG> и $<CONFIG:...>). Эти переменные и выражения-генераторы подробно рассматриваются в следующей главе, а пока основной интерес представляют следующие два семейства переменных:

- CMAKE_<LANG>_FLAGS_<CONFIG>
- CMAKE_<TARGETTYPE>_LINKER_FLAGS_<CONFIG>

Их можно использовать для добавления дополнительных флагов компилятора и компоновщика сверх набора по умолчанию, предоставляемого одноименными переменными без суффикса _<CONFIG>. Например, флаги для пользовательского типа сборки Profile могут быть определены следующим образом:

```
set(CMAKE_C_FLAGS_PROFILE            "-p -g -O2" CACHE STRING "")
set(CMAKE_CXX_FLAGS_PROFILE          "-p -g -O2" CACHE STRING "")
set(CMAKE_EXE_LINKER_FLAGS_PROFILE   "-p -g -O2" CACHE STRING "")
set(CMAKE_SHARED_LINKER_FLAGS_PROFILE "-p -g -O2" CACHE STRING "")
set(CMAKE_STATIC_LINKER_FLAGS_PROFILE "-p -g -O2" CACHE STRING "")
set(CMAKE_MODULE_LINKER_FLAGS_PROFILE "-p -g -O2" CACHE STRING "")
```

В приведенном выше примере предполагается наличие GCC-совместимого компилятора для упрощения примера и включение профилирования, а также включение отладочных символов и большинства оптимизаций. Альтернативой может быть использование флагов компилятора и компоновщика на основе одного из других типов сборки и добавление необходимых дополнительных флагов. Это можно сделать, если это происходит после команды project(), поскольку эта команда заполняет переменные флагов компилятора и компоновщика по умолчанию. Для профилирования, тип сборки по умолчанию RelWithDebInfo является хорошим вариантом для выбора в качестве базовой конфигурации, поскольку он позволяет как отладку, так и большинство оптимизаций:

```
set(CMAKE_C_FLAGS_PROFILE
    "${CMAKE_C_FLAGS_RELWITHDEBINFO} -p" CACHE STRING "")
set(CMAKE_CXX_FLAGS_PROFILE
    "${CMAKE_CXX_FLAGS_RELWITHDEBINFO} -p" CACHE STRING "")
set(CMAKE_EXE_LINKER_FLAGS_PROFILE
    "${CMAKE_EXE_LINKER_FLAGS_RELWITHDEBINFO} -p" CACHE STRING "")
set(CMAKE_SHARED_LINKER_FLAGS_PROFILE
    "${CMAKE_SHARED_LINKER_FLAGS_RELWITHDEBINFO} -p" CACHE STRING "")
set(CMAKE_STATIC_LINKER_FLAGS_PROFILE
    "${CMAKE_STATIC_LINKER_FLAGS_RELWITHDEBINFO} -p" CACHE STRING "")
set(CMAKE_MODULE_LINKER_FLAGS_PROFILE
    "${CMAKE_MODULE_LINKER_FLAGS_RELWITHDEBINFO} -p" CACHE STRING "")
```

Для каждой пользовательской конфигурации должны быть определены соответствующие переменные флагов компилятора и компоновщика. Для некоторых типов генераторов мультиконфигураций CMake проверит наличие требуемых переменных и выдаст ошибку, если они не установлены.

Другая переменная, которая иногда может быть определена для пользовательского типа сборки, это

CMAKE_<CONFIG>_POSTFIX. Используется для инициализации свойства <CONFIG>_POSTFIX каждой целевой библиотеки, значение которого добавляется к имени файла такой цели при сборке для указанной конфигурации. Это позволяет помещать библиотеки из нескольких типов сборки в один каталог, не перезаписывая друг друга. CMAKE_DEBUG_POSTFIX часто устанавливается в значения типа d или _debug, особенно для сборок Visual Studio, где разные DLL должны использоваться для Debug и не-Debug сборок, поэтому пакеты могут включать библиотеки для обоих типов сборок. В случае пользовательского типа сборки Profile, определенного выше, примером может быть:

```
set(CMAKE_PROFILE_POSTFIX _profile)
```

При создании пакетов, содержащих несколько типов сборок, настоятельно рекомендуется установить CMAKE_<CONFIG>_POSTFIX для каждого типа сборки. По соглашению, постфикс для сборок Release обычно пуст. Обратите внимание, что свойство цели <CONFIG>_POSTFIX игнорируется на платформах Apple.

По историческим причинам элементы, передаваемые команде target_link_libraries(), могут быть снабжены ключевыми словами debug или optimized, чтобы указать, что названный элемент должен быть подключен только для отладочных или неотладочных сборок соответственно. Тип сборки считается отладочным, если он указан в глобальном свойстве DEBUG_CONFIGURATIONS, в противном случае он считается оптимизированным. Для пользовательских типов сборки их имя должно быть добавлено в это глобальное свойство, если они должны рассматриваться как отладочная сборка в данном сценарии. Например, если проект определяет свой собственный пользовательский тип сборки под названием StrictChecker и этот тип сборки должен рассматриваться как неоптимизированный тип отладочной сборки, то это можно (и нужно) пояснить следующим образом:

```
set_property(GLOBAL PROPERTY APPEND DEBUG_CONFIGURATIONS StrictChecker)
```

Новые проекты обычно предпочитают использовать выражения генератора вместо ключевых слов debug и optimized с командой target_link_libraries(). Более подробно эта область рассматривается в следующей главе.

13.4. Рекомендуемые практики

Разработчики не должны считать, что для сборки их проекта используется определенный генератор CMake. Другой разработчик того же проекта может предпочесть использовать другой генератор, потому что он лучше интегрируется с его IDE, или в будущей версии CMake может быть добавлена поддержка нового типа генератора, который может принести другие преимущества. Некоторые инструменты сборки могут содержать ошибки, которые впоследствии могут повлиять на проект, поэтому может быть полезно иметь альтернативные генераторы, на которые можно опереться, пока эти ошибки не будут исправлены. Расширение набора поддерживаемых платформ проекта также может быть затруднено, если предполагается использование определенного генератора CMake.

При использовании генераторов единой конфигурации, таких как Makefiles или Ninja, рассмотрите возможность использования нескольких каталогов сборки, по одному для каждого интересующего типа сборки. Это позволит переключаться между типами сборки без необходимости полной перекомпиляции каждый раз. Это обеспечивает поведение, схожее с тем, что предлагают генераторы нескольких конфигураций, и может быть полезным способом позволить инструментам IDE, таким как Qt Creator, имитировать функциональность нескольких конфигураций.

Для генераторов одиночной конфигурации рассмотрите возможность установки CMAKE_BUILD_TYPE в лучшее значение по умолчанию, если оно пустое. Хотя пустой тип сборки технически допустим, он также часто неправильно понимается разработчиками как сборка Debug, а не как отдельный тип сборки. Кроме того, избегайте создания логики, основанной на CMAKE_BUILD_TYPE, если предварительно не подтверждено, что используется один генератор конфигурации. Даже в этом случае такая логика, скорее всего, будет хрупкой и может быть выражена с большей общностью и надежностью с помощью выражений генератора.

Изменять переменную CMAKE_CONFIGURATION_TYPES следует только в том случае, если известно, что используется генератор нескольких конфигураций, или если эта переменная уже существует. При добавлении пользовательского типа сборки или удалении одного из типов сборки по умолчанию не изменяйте переменную кэша, а вместо этого измените обычную переменную с тем же именем (она будет иметь приоритет над переменной кэша). Также предпочитайте добавлять и удалять отдельные элементы, а не полностью заменять список. Обе эти меры помогут избежать вмешательства в изменения, внесенные в переменную кэша разработчиком.

Если требуется CMake 3.9 или более поздняя версия, используйте глобальное свойство GENERATOR_IS_MULTI_CONFIG для окончательного запроса типа генератора вместо того, чтобы полагаться на существование CMAKE_CONFIGURATION_TYPES для выполнения менее надежной проверки.

Распространенной, но неправильной практикой является запрос свойства цели LOCATION для определения имени выходного файла цели. Связанная с этим ошибка заключается в предположении определенной структуры выходного каталога сборки в пользовательских командах (см. главу 17, *Пользовательские задачи*). Эти методы не работают для всех типов сборки, поскольку LOCATION не известно во время конфигурирования для генераторов с несколькими конфигурациями, а структура выходного каталога сборки обычно отличается для различных типов генераторов CMake.
Вместо этого следует использовать выражения генераторов, такие как $<TARGET_FILE:...>, поскольку они надежно обеспечивают требуемый путь для всех генераторов, будь то одноконфигурационные или многоконфигурационные.

Глава 14. Основы компилятора и компоновщика

В предыдущей главе обсуждался тип сборки и то, как он связан с выбором определенного набора поведения компилятора и компоновщика. В этой главе рассматриваются основы управления поведением компилятора и

компоновщика. Представленный здесь материал охватывает некоторые из наиболее важных тем и методов, с которыми должен быть знаком каждый разработчик CMake.

Прежде чем углубиться в детали, важно отметить, что по мере развития CMake улучшались и доступные методы управления поведением компилятора и компоновщика. Фокус сместился с более глобального взгляда на сборку на тот, где можно контролировать требования каждой отдельной цели, а также то, как эти требования должны или не должны переноситься на другие цели, которые зависят от нее. Это важный сдвиг в мышлении, поскольку он влияет на то, как проект может наиболее эффективно определить способ сборки целей. Более зрелые функции CMake можно использовать для контроля поведения на грубом уровне за счет потери возможности определять отношения между целями. Как правило, предпочтение следует отдавать более современным функциям, ориентированным на цели, поскольку они значительно повышают надежность сборки и обеспечивают более точный контроль над поведением компилятора и компоновщика. Более новые функции также более последовательны в своем поведении и в том, как их следует использовать.

14.1. Свойства цели

В системе свойств CMake целевые свойства являются основным механизмом управления флагами компилятора и компоновщика. Некоторые свойства предоставляют возможность указать любой произвольный флаг, в то время как другие сосредоточены на конкретной возможности, что позволяет абстрагироваться от различий платформ или компиляторов. Эта глава посвящена наиболее часто используемым свойствам общего назначения, а в последующих главах будет рассмотрен ряд более специфических свойств.

14.1.1. Флаги компилятора

Наиболее фундаментальными целевыми свойствами для управления флагами компилятора являются следующие, каждое из которых содержит список элементов:

ВКЛЮЧАЕМЫЕ_КАТАЛОГИ

Это список каталогов, которые будут использоваться в качестве путей поиска заголовков, все из которых должны быть абсолютными. CMake добавит флаг компилятора для каждого пути с соответствующим префиксом (обычно -I или /I). Когда создается цель, начальное значение этого свойства цели берется из одноименного свойства каталога.

КОМПИЛЯЦИЯ_ОПРЕДЕЛЕНИЙ

Здесь содержится список определений, которые необходимо задать в командной строке компиляции. Определение имеет форму VAR или VAR=VALUE, которую CMake преобразует в соответствующую форму для используемого компилятора (обычно -DVAR... или /DVAR...). Когда цель создается, начальное значение этого свойства цели будет пустым. Существует одноименное свойство directory, но оно *не* используется для задания начального значения этого свойства target. Скорее, свойства directory и target *объединяются* в конечной командной строке компилятора.

COMPILE_OPTIONS

В этом свойстве указываются любые флаги компилятора, которые не являются ни путями поиска заголовков,ни определениями символов. Когда создается цель, начальное значение этого свойства цели берется из одноименного свойства каталога.

Старое и ныне устаревшее целевое свойство с именем COMPILE_FLAGS служило для той же цели, что и COMPILE_OPTIONS. Свойство COMPILE_FLAGS рассматривается как одна строка, которая включается непосредственно в командную строку компилятора. В результате может

> потребоваться ручное экранирование, тогда как COMPILE_OPTIONS - это список, и CMake выполняет любое необходимое экранирование или кавычки автоматически

Свойства INCLUDE_DIRECTORIES и COMPILE_DEFINITIONS на самом деле просто удобны, они заботятся о специфических флагах компилятора для наиболее распространенных вещей, которые часто хотят установить проекты. Все остальные флаги, специфичные для компилятора, предоставляются в свойстве COMPILE_OPTIONS.

Каждое из трех вышеперечисленных свойств цели имеет связанное свойство цели с тем же именем, только с приставкой INTERFACE_. Эти интерфейсные свойства делают то же самое, только вместо того, чтобы применяться к самой цели, они применяются к любой другой цели, которая ссылается непосредственно на нее. Другими словами, они используются для указания флагов компилятора, которые должны наследовать *потребляющие* цели. По этой причине их часто называют *требованиями использования*, в отличие от свойств, не относящихся к интерфейсу, которые иногда называют *требованиями сборки*. Два специальных библиотечных типа IMPORTED и INTERFACE обсуждаются далее в Главе 16, *Типы целей*. Эти специальные типы библиотек поддерживают только целевые свойства INTERFACE_..., но не свойства, не относящиеся к INTERFACE_....

В отличие от своих неинтерфейсных аналогов, ни одно из вышеперечисленных свойств INTERFACE_... не инициализируется из свойств каталога. Вместо этого все они начинаются пустыми, поскольку только проект знает, какие пути поиска заголовков, определения и флаги компилятора должны распространяться на потребляющие цели.

Все вышеперечисленные целевые свойства также поддерживают выражения генератора. Это особенно полезно для свойства COMPILE_OPTIONS, поскольку оно позволяет добавлять определенный флаг только при выполнении некоторого условия, например, только для определенного компилятора. Другим распространенным способом является получение пути, связанного с некоторой другой целью, и использование его в качестве части каталога include.

Если флагами компилятора необходимо управлять на уровне отдельных исходных файлов, целевые свойства недостаточно детализированы. Для таких случаев CMake предоставляет свойства COMPILE_DEFINITIONS, COMPILE_FLAGS и COMPILE_OPTIONS (свойство COMPILE_OPTIONS было добавлено только в CMake 3.11). Они аналогичны одноименным целевым свойствам, за исключением того, что применяются только к отдельному исходному файлу, для которого они установлены. Обратите внимание, что их поддержка выражений генератора отстает от поддержки целевых свойств: свойство исходного файла COMPILE_DEFINITIONS получило поддержку выражений генератора в CMake 3.8, а остальные - в 3.11. Более того, формат файла проекта Xcode вообще не поддерживает свойства исходного файла, специфичные для конфигурации, поэтому при работе с платформами Apple в свойствах исходного файла не следует использовать $<CONFIG> или $<CONFIG:...>. Также помните о предупреждениях, обсуждавшихся в разделе 9.5, "Свойства исходного файла", относительно деталей реализации, приводящих к проблемам производительности при использовании свойств исходного файла.

14.1.2. Флаги компоновщика

Свойства цели, связанные с флагами компоновщика, похожи на свойства флагов компилятора, но в них задействовано меньше свойств:

LINK_LIBRARIES

Это свойство содержит список всех библиотек, на которые цель должна ссылаться напрямую. Он изначально пуст при создании цели и поддерживает выражения генератора. Каждая из перечисленных библиотек может быть одной из следующих:

- Путь к библиотеке, обычно указывается как абсолютный путь.
- Просто имя библиотеки без пути, обычно также без префикса (например, lib) или суффикса (например, .a, .so, .dll) имени файла, специфичного для платформы.
- Имя целевой библиотеки CMake. CMake преобразует его в путь к собранной библиотеке при генерации команды компоновщика, включая добавление любого префикса или суффикса к имени файла, соответствующего платформе. Поскольку CMake обрабатывает все различные различия платформ и пути от имени проекта, использование имени цели CMake обычно является предпочтительным методом.

CMake будет использовать соответствующие флаги компоновщика для компоновки каждого элемента, перечисленного в свойстве LINK_LIBRARIES.

LINK_FLAGS

Здесь хранится список флагов, передаваемых компоновщику для целей, которые являются исполняемыми файлами, разделяемыми библиотеками или библиотеками модулей. Оно игнорируется для целей, собираемых как статическая библиотека. Это свойство предназначено для общих флагов компоновщика, а не для тех флагов, которые указывают другие библиотеки для компоновки. Выражения-генераторы не поддерживаются документально. При создании цели это свойство будет пустым.

STATIC_LIBRARY_FLAGS

Это аналог LINK_FLAGS, применяемый только к целям, собираемым как статическая библиотека. Он будет использоваться для инструмента библиотекаря или архиватора.

В отличие от свойств компилятора, только LINK_LIBRARIES имеет эквивалентное свойство интерфейса, INTERFACE_LINK_LIBRARIES. Не существует интерфейсного эквивалента LINK_FLAGS или STATIC_LIBRARY_FLAGS.

В старых проектах иногда можно встретить целевое свойство с именем LINK_INTERFACE_LIBRARIES, которое является более старой версией INTERFACE_LINK_LIBRARIES. Это свойство устарело с версии CMake 2.8.12, но политика CMP0022 может быть использована, чтобы дать старому свойству приоритет, если это необходимо. Новые проекты должны предпочесть использовать INTERFACE_LINK_LIBRARIES вместо него.

Свойства LINK_FLAGS и STATIC_LIBRARY_FLAGS не поддерживают выражения генератора. Однако у них есть связанные свойства, специфичные для конфигурации:

- LINK_FLAGS_<CONFIG>
- STATIC_LIBRARY_FLAGS_<CONFIG>

Эти флаги будут использоваться в дополнение к флагам, не относящимся к конфигурации, когда <CONFIG> соответствует собираемой конфигурации.

14.1.3. Команды свойств цели

Перечисленные выше свойства цели обычно не управляются напрямую. CMake предоставляет специальные функции для их изменения более удобным и надежным способом, который также способствует четкой спецификации зависимостей и переходного поведения между целями. В разделе 4.3, "Связывание целей",

была представлена команда target_link_libraries(), а также объяснение того, как выражаются межцелевые зависимости с помощью спецификаций PRIVATE, PUBLIC и INTERFACE. Это предыдущее обсуждение было сосредоточено на отношениях зависимости между целями, но после вышеприведенного обсуждения свойств целей, точное влияние этих ключевых слов теперь может быть уточнено.

```
target_link_libraries(targetName
     <PRIVATE|PUBLIC|INTERFACE> item1 [item2 ...]
    [<PRIVATE|PUBLIC|INTERFACE> item3 [item4 ...]]
    ...
)
```

PRIVATE

Элементы, перечисленные после PRIVATE, влияют только на поведение самого targetName. Изменяются только неинтерфейсные_... свойства цели (т.е. LINK_LIBRARIES, LINK_FLAGS и STATIC_LIBRARY_FLAGS).

INTERFACE

Это дополнение к PRIVATE, при этом элементы, следующие за ключевым словом INTERFACE, влияют только на цели, которые ссылаются на targetName. Изменяются только свойства цели INTERFACE_... targetName (т.е. INTERFACE_LINK_LIBRARIES).

PUBLIC

Это эквивалентно объединению эффектов PRIVATE и INTERFACE.

В большинстве случаев разработчики, вероятно, найдут объяснение в разделе 4.3, "Связывание целей", более интуитивным, но приведенное выше более точное описание может помочь объяснить поведение в более сложных проектах, где свойствами можно манипулировать необычным образом. Приведенное выше описание также очень точно соответствует поведению других команд target_...(), которые манипулируют флагами компилятора. Фактически, все они следуют одной и той же схеме и применяют ключевые слова PRIVATE, PUBLIC и INTERFACE одинаковым образом.

```
target_include_directories(targetName [BEFORE] [SYSTEM]
     <PRIVATE|PUBLIC|INTERFACE> dir1 [dir2 ...]
    [<PRIVATE|PUBLIC|INTERFACE> dir3 [dir4 ...]]
    ...
)
```

Команда target_include_directories() добавляет пути поиска заголовков в целевые свойства INCLUDE_DIRECTORIES и INTERFACE_INCLUDE_DIRECTORIES. Каталоги, следующие за ключевым словом PRIVATE, добавляются к целевому свойству INCLUDE_DIRECTORIES, а каталоги, следующие за ключевым словом INTERFACE, добавляются к целевому свойству INTERFACE_INCLUDE_DIRECTORIES. Каталоги, следующие за ключевым словом PUBLIC, добавляются в оба свойства.

Обычно при каждом вызове target_include_directories() указанные каталоги добавляются к соответствующим свойствам цели. Это упрощает добавление нескольких путей естественным, постепенным образом. При необходимости можно использовать ключевое слово BEFORE, чтобы добавить указанные каталоги к существующему содержимому целевых свойств.

Если указано ключевое слово SYSTEM, то на некоторых платформах компилятор будет рассматривать перечисленные каталоги как системные пути include. Это может привести к пропуску некоторых предупреждений компилятора или изменению способа обработки зависимостей файлов. Это также может повлиять на порядок поиска путей к заголовкам для некоторых компиляторов. Иногда у разработчиков

возникает соблазн использовать SYSTEM, чтобы заглушить предупреждения, поступающие из заголовков, вместо того, чтобы обращаться к этим предупреждениям напрямую. Если такие заголовки являются частью проекта, SYSTEM, как правило, не подходит для использования. В целом, SYSTEM предназначена для путей вне проекта, но даже в этом случае она редко бывает нужна.

Также стоит отметить, что пути, указанные свойством INTERFACE_INCLUDE_DIRECTORIES *импортированной* цели, по умолчанию будут рассматриваться потребляющими целями как пути SYSTEM. Это связано с тем, что предполагается, что импортируемые цели приходят извне проекта, и поэтому связанные с ними заголовки должны обрабатываться так же, как и другие заголовки, предоставляемые системой. Проект может отменить это поведение, установив свойство NO_SYSTEM_FROM_IMPORTED *потребляющей* цели в true, что предотвратит обработку всех потребляемых импортированных целей как SYSTEM. Импортированные цели подробно рассматриваются в главе 16, *Типы целей*.

Команда target_include_directories() предлагает еще одно преимущество по сравнению с манипулированием свойствами цели напрямую. Проекты могут указывать и относительные каталоги, а не только абсолютные. Относительные пути будут автоматически преобразованы в абсолютные, где это необходимо, при этом пути будут рассматриваться как относительные к текущему исходному каталогу.

Поскольку команда target_include_directories(), по сути, просто заполняет соответствующие свойства цели, к ней применимы все обычные свойства этих свойств. В частности, можно использовать выражения-генераторы, что становится гораздо важнее при установке целей и создании пакетов. Выражения генератора $<BUILD_INTERFACE:...> и $<INSTALL_INTERFACE:...> позволяют указывать различные пути для сборки и установки. Для устанавливаемых целей обычно используются относительные пути, и они будут интерпретироваться как относительные к базовому расположению установки, а не к исходному каталогу. В разделе 25.2.1, "Свойства интерфейса", более подробно рассматривается этот аспект указания путей поиска заголовков.

```
target_compile_definitions(targetName
     <PRIVATE|PUBLIC|INTERFACE> item1 [item2 ...]
    [<PRIVATE|PUBLIC|INTERFACE> item3 [item4 ...]]
    ...
)
```

Команда target_compile_definitions() довольно проста, каждый элемент имеет форму VAR или VAR=VALUE. Элементы PRIVATE заполняют целевое свойство COMPILE_DEFINTIONS, а элементы INTERFACE заполняют целевое свойство INTERFACE_COMPILE_DEFINTIONS. Элементы PUBLIC заполняют оба целевых свойства. Можно использовать выражения генератора, но обычно нет необходимости по-разному обрабатывать ситуации сборки и установки.

```
target_compile_options(targetName [BEFORE]
     <PRIVATE|PUBLIC|INTERFACE> item1 [item2 ...]
    [<PRIVATE|PUBLIC|INTERFACE> item3 [item4 ...]]
    ...
)
```

Команда target_compile_options() также достаточно проста. Каждый элемент рассматривается как опция компилятора, причем элементы PRIVATE заполняют целевое свойство COMPILE_OPTIONS, а элементы INTERFACE - целевое свойство INTERFACE_COMPILE_OPTIONS. Как обычно, элементы PUBLIC заполняют оба целевых свойства. Во всех случаях каждый элемент добавляется к существующим значениям целевого свойства, но ключевое слово BEFORE может быть использовано для добавления. Выражения генератора

поддерживаются во всех случаях, и обычно нет необходимости по-разному обрабатывать ситуации сборки и установки.

14.2. Свойства и команды каталога

В CMake 3.0 и более поздних версиях свойства target являются наиболее предпочтительными для указания флагов компилятора и компоновщика, поскольку они позволяют определить, как они взаимодействуют с целями, которые ссылаются друг на друга. В более ранних версиях CMake свойства цели были гораздо менее заметны, и вместо этого свойства часто задавались на уровне каталога. Этим свойствам директорий и командам, обычно используемым для работы с ними, не хватает последовательности, которую демонстрируют их эквиваленты на уровне целей, что является еще одной причиной, по которой проекты должны по возможности избегать их. Тем не менее, поскольку многие онлайн-учебники и примеры все еще используют их, разработчики должны хотя бы знать о свойствах и командах на уровне каталогов.

```
include_directories([AFTER | BEFORE] [SYSTEM] dir1 [dir2...])
```

Упрощенно, команда include_directories() добавляет пути поиска заголовков к целям, созданным в текущей области видимости каталога и ниже. По умолчанию пути добавляются к существующему списку каталогов, но это значение можно изменить, установив переменную CMAKE_INCLUDE_DIRECTORIES_BEFORE в ON. Также можно управлять параметрами BEFORE и AFTER для каждого вызова, чтобы явно указать, как должны обрабатываться пути для данного вызова. Проекты должны с осторожностью относиться к установке CMAKE_INCLUDE_DIRECTORIES_BEFORE, так как большинство разработчиков, скорее всего, предположат, что будет применяться поведение по умолчанию для добавляемых каталогов. Ключевое слово SYSTEM имеет тот же эффект, что и для команды target_include_directories().

Пути, указанные в include_directories(), могут быть относительными или абсолютными. Относительные пути автоматически преобразуются в абсолютные и рассматриваются как относительные к текущему каталогу источника. Пути могут также содержать выражения-генераторы.

Детали того, что на самом деле делает include_directories(), сложнее, чем упрощенное объяснение выше. В основном, есть два основных эффекта от вызова include_directories():

- Перечисленные пути добавляются к свойству каталога INCLUDE_DIRECTORIES текущего файла CMakeLists.txt. Это означает, что все цели, созданные в текущем каталоге и ниже, будут иметь каталоги, добавленные к свойству цели INCLUDE_DIRECTORIES.
- Любая цель, созданная в текущем файле CMakeLists.txt (точнее, в текущей области каталогов), также будет иметь пути, добавленные в свойство цели INCLUDE_DIRECTORIES, даже если эти цели были созданы до вызова include_directories(). Это относится строго только к целям, созданным в текущем файле CMakeLists.txt или других файлах, извлеченных через include(), но не к целям, созданным в родительских или дочерних областях каталогов.

Именно второй из перечисленных пунктов обычно удивляет многих разработчиков. Чтобы избежать ситуаций, которые могут привести к подобной путанице, если команда include_directories() должна быть использована, предпочитайте вызывать ее в самом начале файла CMakeLists.txt до того, как будут созданы какие-либо цели или подкаталоги будут подтянуты с помощью include() или add_subdirectory().

```
add_definitions(-DSomeSymbol /DFoo=Value ...)
remove_definitions(-DSomeSymbol /DFoo=Value ...)
```

Команды add_definitions() и remove_definitions() добавляют и удаляют записи в свойстве каталога COMPILE_DEFINITIONS. Каждая запись должна начинаться с -D или /D - двух наиболее распространенных форматов флагов, используемых подавляющим большинством компиляторов. Этот префикс флага удаляется CMake перед сохранением define в свойстве каталога COMPILE_DEFINITIONS, поэтому не имеет значения, какой префикс используется, независимо от компилятора или платформы, на которой собирается проект.

Так же, как и для include_directories(), эти две команды влияют на все цели, созданные в текущем файл CMakeLists.txt, даже те, которые были созданы до вызова add_definitions() или remove_definitions(). Цели, созданные в дочерних областях каталогов, будут затронуты только в том случае, если они были созданы *после* вызова. Это прямое следствие того, как свойство каталога COMPILE_DEFINITIONS используется в CMake.

Хотя это и не рекомендуется, с помощью этих команд также можно указывать флаги компилятора, отличные от определений. Если CMake не распознает конкретный элемент как похожий на определение компилятора, этот элемент будет добавлен в свойство каталога COMPILE_OPTIONS без изменений. Такое поведение присутствует по историческим причинам, но новые проекты должны избегать такого поведения (см. команду add_compile_options() чуть ниже для альтернативы).

Поскольку базовые свойства каталога поддерживают выражения-генераторы, то и эти две команды тоже, но с некоторыми оговорками. Выражения-генераторы следует использовать только для части значения определения, а не для части имени (т.е. только после "=" в элементе -DVAR=VALUE или вообще не использовать для элемента -DVAR). Это связано с тем, как CMake анализирует каждый элемент, чтобы проверить, является ли он определением компилятора или нет. Обратите внимание, что эти команды изменяют только свойства каталога, они не влияют на целевое свойство COMPILE_DEFINITIONS.

Команда add_definitions() имеет ряд недостатков. Требование добавлять к каждому элементу префикс -D или /D, чтобы он рассматривался как определение, не согласуется с другим поведением CMake. Тот факт, что отсутствие префикса заставляет команду рассматривать элемент как общую опцию, также противоречит интуиции, учитывая название команды. Более того, ограничение на поддержку генераторных выражений только для части VALUE определения KEY=VALUE также является прямым следствием требования префикса. В связи с этим в CMake 3.12 была введена команда add_compile_definitions() в качестве замены add_definitions():

```
add_compile_definitions(SomeSymbol Foo=Value ...)
```

Новая команда работает только с компилируемыми определениями, не требует префикса для каждого элемента и выражения генератора могут использоваться без ограничения VALUE-only. Название новой команды и обработка элементов определения соответствует аналогичной команде target_compile_definitions(). add_compile_definitions() по-прежнему влияет на все цели, созданные в той же области видимости каталога, независимо от того, были ли эти цели созданы до или после вызова add_compile_definitions(), поскольку это свойство каталога COMPILE_DEFINITIONS, которым манипулирует команда, а не сама команда.

```
add_compile_options(opt1 [opt2 ...])
```

Команда add_compile_options() используется для предоставления произвольных опций компилятора. В отличие от команд include_directories(), add_definitions(), remove_definitions() и add_compile_definitions(), ее поведение очень прямолинейно и предсказуемо. Каждая опция, переданная команде add_compile_options(), добавляется к свойству каталога COMPILE_OPTIONS. Каждая цель, впоследствии созданная в текущей области видимости каталога и ниже, наследует эти опции в своем собственном свойстве COMPILE_OPTIONS target. Любые цели, созданные до вызова, не затрагиваются. Такое поведение гораздо ближе к тому, что интуитивно ожидают разработчики, по сравнению с другими командами свойств каталога. Более того, выражения

генератора поддерживаются базовыми свойствами каталога и цели, поэтому команда add_compile_options() также поддерживает их.

```
link_libraries(item1 [item2 ...] [ [debug | optimized | general] item] ...)
link_directories(dir1 [dir2 ...])
```

В ранних версиях CMake эти две команды были основным способом указать CMake на компоновку библиотек в другие цели. Они влияют на все цели, созданные в текущем каталоге и ниже после вызова команд, но все существующие цели остаются незатронутыми (т.е. аналогично поведению команды add_compile_options()). Элементами, указанными в команде link_libraries(), могут быть цели CMake, имена библиотек, полные пути к библиотекам или даже флаги компоновщика.

Говоря в общих чертах, элемент можно сделать применимым только к типу сборки Debug, поставив перед ним ключевое слово debug, или ко всем типам сборки, кроме Debug, поставив перед ним ключевое слово optimized. Перед элементом можно поставить ключевое слово general, чтобы указать, что он применяется ко всем типам сборки, но поскольку general в любом случае используется по умолчанию, пользы от этого мало. Все три ключевых слова влияют только на один элемент, следующий за ним, а не на все элементы до следующего ключевого слова. Использовать эти ключевые слова настоятельно не рекомендуется, поскольку выражения генератора обеспечивают гораздо лучший контроль над тем, когда элемент должен быть добавлен. Для учета пользовательских типов сборки тип сборки считается отладочной конфигурацией, если он указан в глобальном свойстве DEBUG_CONFIGURATIONS.

Каталоги, добавленные функцией link_directories(), имеют эффект только в том случае, если CMake дано голое имя библиотеки для компоновки. CMake добавляет указанные пути в командную строку компоновщика и оставляет компоновщику возможность самостоятельно найти такие библиотеки. Если указан относительный путь, он будет рассматриваться как относительный к текущему каталогу с исходным кодом (очень ранние версии CMake вели себя иначе, подробности см. в документации к политике CMP0015). Обычно предпочтение отдается полному пути или имени цели CMake, поскольку они более надежны. Кроме того, после добавления каталога поиска компоновщика с помощью link_directories() у проектов нет удобного способа удалить этот путь поиска, если это необходимо. По этим причинам следует по возможности избегать добавления каталогов поиска компоновщика.

14.3. Переменные компилятора и компоновщика

Свойства - это основной способ, которым проекты должны стремиться влиять на флаги компилятора и компоновщика. Конечные пользователи не могут манипулировать свойствами напрямую, поэтому проект полностью контролирует установку свойств. Однако бывают ситуации, когда пользователь захочет добавить свои собственные флаги компилятора или компоновщика. Они могут захотеть добавить больше опций предупреждения, включить специальные функции компилятора, такие как санитары или отладочные переключатели, и так далее. Для таких ситуаций больше подходят переменные.

CMake предоставляет набор переменных, определяющих флаги компилятора и компоновщика, которые должны быть объединены с флагами, предоставляемыми различными свойствами каталогов, целевых и исходных файлов. Обычно это кэш-переменные, чтобы пользователь мог легко просматривать и изменять их, но они также могут быть заданы как обычные переменные CMake в файлах CMakeLists.txt проекта (чего проекты должны стараться избегать). CMake придает кэш-переменным подходящие значения по умолчанию при первом запуске в каталоге сборки.

Основные переменные, непосредственно влияющие на флаги компилятора, имеют следующий вид:

- CMAKE_<LANG>_FLAGS
- CMAKE_<LANG>_FLAGS_<CONFIG>

В этом семействе переменных <LANG> соответствует компилируемому языку, типичными значениями являются C, CXX, Fortran, Swift и так далее. Часть <CONFIG> представляет собой строку в верхнем регистре, соответствующую одному из типов сборки, например DEBUG, RELEASE, RELWITHDEBINFO или MINSIZEREL. Первая переменная будет применяться ко всем типам сборки, включая генераторы одиночной конфигурации с пустым CMAKE_BUILD_TYPE, в то время как вторая переменная применяется только к типу сборки, указанному <CONFIG>. Таким образом, файл C++, собираемый с конфигурацией Debug, будет иметь флаги компилятора из CMAKE_CXX_FLAGS и CMAKE_CXX_FLAGS_DEBUG.

Первая встреченная команда project() создаст для них кэш-переменные, если они еще не существуют (это некоторое упрощение, более полное объяснение дано в главе 21, *Toolchains And Cross Compiling*). Поэтому после первого запуска CMake их значения легко проверить в приложении CMake GUI. Например, для одного конкретного компилятора по умолчанию определены следующие переменные для языка C++:

CMAKE_CXX_FLAGS	
CMAKE_CXX_FLAGS_DEBUG	-g -O0
CMAKE_CXX_FLAGS_RELEASE	-O3 -DNDEBUG
CMAKE_CXX_FLAGS_RELWITHDEBINFO	-O2 -g -DNDEBUG
CMAKE_CXX_FLAGS_MINSIZEREL	-Os -DNDEBUG

Работа с флагами компоновщика аналогична. Они управляются следующим семейством переменных:

- CMAKE_<TARGETTYPE>_LINKER_FLAGS
- CMAKE_<TARGETTYPE>_LINKER_FLAGS_<CONFIG>

Они специфичны для определенного типа целей, каждый из которых был представлен в главе 4 "*Создание простых целей*". Часть имени переменной <TARGETTYPE> должна быть одной из следующих:

EXE
: Цели, созданные с помощью add_executable().

SHARED
: Цели, созданные с помощью add_library(name SHARED ...) или эквивалентной команды, например, без ключевого слова SHARED, но с переменной BUILD_SHARED_LIBS, установленной в true.

STATIC
: Цели, созданные с помощью add_library(name STATIC ...) или эквивалентной команды, например, без ключевого слова STATIC, но с переменной BUILD_SHARED_LIBS, установленной в false или не определенной.

MODULE
: Цели, созданные с помощью add_library(name MODULE ...).

Как и флаги компилятора, флаги CMAKE_<TARGETTYPE>_LINKER_FLAGS используются при компоновке любой конфигурации сборки, тогда как флаги CMAKE_<TARGETTYPE>_LINKER_FLAGS_<CONFIG> добавляются только

для соответствующей CONFIG. Нередко на некоторых платформах некоторые или все флаги компоновщика являются пустыми строками.

В руководствах и примерах кода CMake часто используются вышеуказанные переменные для управления флагами компилятора и компоновщика. Это было довольно распространенной практикой в эпоху до CMake 3.0, но со смещением акцента на модель, ориентированную на цель, в CMake 3.0 и последующих версиях такие примеры больше не являются хорошей моделью для подражания. Они часто приводят к ряду очень распространенных ошибок, некоторые из которых представлены ниже.

Переменные компилятора/линкера - это отдельные строки, а не списки

Если необходимо установить несколько флагов компилятора, их следует указывать в виде одной строки, а не в виде списка. CMake не будет правильно обрабатывать переменные флагов, если их содержимое содержит точки с запятой, в которые превращается список, если он указан в проекте.

```
# Wrong, list used instead of a string
set(CMAKE_CXX_FLAGS -Wall -Werror)

# Correct, but see later sections for why appending would be preferred
set(CMAKE_CXX_FLAGS "-Wall -Werror")

# Appending to existing flags the correct way (two methods)
set(CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS} -Wall -Werror")
string(APPEND CMAKE_CXX_FLAGS " -Wall -Werror")
```

Различать кэш- и некэш-переменные

Все упомянутые выше переменные являются переменными *кэша*. Можно определить одноименные некэш-переменные, и они будут переопределять кэш-переменные для текущей области видимости каталога и его дочерних элементов (т.е. созданных функцией add_subdirectory()). Однако могут возникнуть проблемы, когда проект пытается принудительно обновить переменную кэша вместо локальной переменной. Код, подобный приведенному ниже, обычно затрудняет работу с проектами и может привести к тому, что разработчики будут чувствовать, что они борются с проектом, когда хотят изменить флаги для собственной сборки через приложение CMake GUI или аналогичное:

```
# Case 1: Only has an effect if the variable isn't already in the cache
set(CMAKE_CXX_FLAGS "-Wall -Werror" CACHE STRING "C++ flags")

# Case 2: Using FORCE to always update the cache variable, but this overwrites
#         any changes a developer might make to the cache
set(CMAKE_CXX_FLAGS "-Wall -Werror" CACHE STRING "C++ flags" FORCE)

# Case 3: FORCE + append = recipe for disaster (see discussion below)
set(CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS} -Wall -Werror" CACHE STRING "C++ flags" FORCE)
```

Первый приведенный выше случай подчеркивает распространенную оплошность, допускаемую разработчиками-новичками в CMake. Без ключевого слова FORCE команда set() обновляет кэш-переменную, только если она еще не определена. Поэтому при первом запуске CMake может оказаться, что команда set() делает то, что задумал разработчик (если она помещена перед любой командой project()), но если строка будет изменена, чтобы указать что-то другое для флагов, это изменение не будет применено к существующей сборке,

потому что переменная уже будет в кэше на тот момент. Обычной реакцией на это является использование FORCE для обеспечения постоянного обновления переменной кэша, как показано во втором случае, но это создает другую проблему. Кэш является основным средством для разработчиков изменять переменные локально, без необходимости редактировать файлы проекта. Если проект использует FORCE для односторонней установки переменных кэша таким образом, то все изменения, внесенные разработчиком в эту переменную кэша, будут потеряны. Третий случай еще более проблематичен, поскольку при каждом запуске CMake флаги будут добавляться заново, что приведет к постоянно растущему и повторяющемуся набору флагов. Использование FORCE для обновления кэша флагов компилятора и компоновщика в таком виде редко бывает хорошей идеей.

Вместо того чтобы просто удалить ключевое слово FORCE, правильным поведением будет установка не кэш-переменной, а не кэш-переменной. В этом случае безопасно добавлять флаги к текущему значению, поскольку кэш-переменная остается нетронутой, поэтому каждый запуск CMake начинается с одним и тем же набором флагов из кэш-переменной, независимо от того, как часто вызывается CMake. Любые изменения, которые разработчик решит внести в кэш-переменную, также будут сохранены.

```
# Preserves the cache variable contents, appends new flags safely
set(CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS} -Wall -Werror")
```

Предпочитайте флаги добавления, а не замены

Как уже говорилось выше, у разработчиков иногда возникает соблазн в одностороннем порядке установить флаги компилятора в файлах CMakeLists.txt, например, так:

```
# Not ideal, discards any developer settings from cache
set(CMAKE_CXX_FLAGS "-Wall -Werror")
```

Поскольку при этом отбрасывается любое значение, установленное переменной кэша, разработчики теряют возможность легко вводить свои собственные флаги. Замена существующих флагов подобным образом заставляет разработчиков копаться в файлах проекта, чтобы найти, где и как изменить любые строки, которые изменяют соответствующие флаги. Для сложного проекта с большим количеством подкаталогов это может быть довольно утомительно. Там, где это возможно, проекты должны предпочитать добавлять флаги к существующему значению.

Разумным исключением из этого правила может быть ситуация, когда проект должен обеспечить обязательный набор флагов компилятора или компоновщика. В таких случаях приемлемым компромиссом может быть установка значений переменных в файле CMakeLists.txt верхнего уровня как можно раньше, в идеале - в самом верху, сразу после команды cmake_minimum_required() (а еще лучше - в файле цепочки инструментов, если она используется - подробнее см. главу 21, *Цепочки инструментов и перекрестная компиляция*). Однако следует помнить, что со временем проект может стать дочерним для другого проекта, и тогда он перестанет быть верхним уровнем сборки, и пригодность этого компромисса может снизиться.

Понять, когда используются переменные значения

Одним из наиболее непонятных аспектов переменных флагов компилятора и компоновщика является момент в процессе сборки, когда их значение фактически используется. Можно вполне обоснованно ожидать, что следующий код будет вести себя так, как указано во встроенных комментариях:

```
# Save the original set of flags so we can restore them later
set(oldCxxFlags "${CMAKE_CXX_FLAGS}")

# This library has stringent build requirements, so enforce them just for it alone
# WARNING: This doesn't do what it may appear to do!
set(CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS} -Wall -Werror")
add_library(strictReq STATIC ...)

# Less strict requirements from here, so restore the original set of compiler flags
set(CMAKE_CXX_FLAGS "${oldCxxFlags}")
add_library(relaxedReq STATIC ...)
```

Возможно, вас удивит тот факт, что при вышеописанном расположении библиотека strictReq *не* будет собрана с флагами -Wall -Werror. Интуитивно можно предположить, что CMake использует значение переменной на момент вызова add_library(), но на самом деле используется значение переменной в конце обработки для данной области каталогов. Другими словами, важно значение переменной в конце файла CMakeLists.txt для этого каталога. Это может привести к неожиданным результатам в различных ситуациях для неосведомленных.

Один из основных способов, которым разработчики попадают в ловушку такого поведения, заключается в том, что переменные компилятора и компоновщика рассматриваются так, как будто они немедленно применяются к любым создаваемым целям. Другая связанная с этим ловушка - когда include() используется после создания целей, и включаемый файл(ы) изменяет(ют) переменные компилятора или компоновщика. Это также изменит флаги компилятора и компоновщика для уже определенных целей в текущей области каталогов. Из-за этой отложенной природы переменных компилятора и компоновщика работа с ними может быть нестабильной. В идеале, проект должен изменять их только на ранних стадиях в файле CMakeLists.txt верхнего уровня, если это вообще возможно, чтобы свести к минимуму возможность неправильного использования и неожиданностей для разработчиков.

14.4. Рекомендуемые практики

В этой главе мы рассмотрели те области CMake, которые претерпели наиболее значительные улучшения по сравнению с предыдущими версиями. Читатель может встретить множество примеров и руководств в Интернете и других местах, в которых по-прежнему рекомендуются схемы и подходы, использующие старые методы с использованием переменных и команд свойств каталога, но следует понимать, что команды target_...() должны быть предпочтительным подходом в эпоху CMake 3.0+.

Проекты должны стремиться определять все зависимости между целями с помощью команды target_link_libraries(). Это четко выражает природу отношений между целями и однозначно сообщает всем разработчикам проекта о том, как связаны цели. Команду target_link_libraries() следует предпочесть команде link_libraries() или манипулированию свойствами цели или каталога напрямую. Аналогично, другие команды target_...() предлагают более чистый, более последовательный и более надежный способ манипулирования флагами компилятора и компоновщика, чем переменные, команды свойств каталога или прямое манипулирование свойствами. Следующее общее руководство может оказаться полезным:

- По возможности предпочитайте использовать команды target_...() для описания отношений между целями и для изменения поведения компилятора и компоновщика.
- В целом, предпочитайте избегать команд свойств каталога. Хотя они могут быть удобны в некоторых специфических обстоятельствах, последовательное использование команд target_...() вместо них установит

четкие шаблоны, которым смогут следовать все разработчики в проекте. Если команды свойств каталога необходимо использовать, делайте это как можно раньше в файле CMakeLists.txt, чтобы избежать некоторых менее интуитивных действий, описанных в предыдущих разделах.

- Избегайте прямых манипуляций со свойствами цели и каталога, которые влияют на поведение компилятора и компоновщика. Поймите, что делают свойства и как различные команды манипулируют ими, но по возможности предпочитайте использовать более специализированные команды, специфичные для целей и каталогов. Однако запрос свойств цели может быть полезен время от времени при исследовании неожиданных флагов командной строки компилятора или компоновщика.
- Предпочитайте избегать изменения различных переменных CMAKE_..._FLAGS и их конфигурационных аналогов. Считайте, что они предназначены для разработчика, который может захотеть изменить их локально по своему усмотрению. Если изменения должны быть применены на основе всего проекта, рассмотрите возможность использования нескольких стратегических команд свойств каталога на верхнем уровне проекта, но подумайте, действительно ли такие настройки должны применяться в одностороннем порядке. Частичным исключением являются файлы цепочек инструментов, где могут быть определены начальные значения по умолчанию (подробное обсуждение этой области см. в главе 21, *Цепочки инструментов и кросс-компиляция*).

Разработчики должны ознакомиться с концепциями отношений PRIVATE, PUBLIC и INTERFACE. Они являются важной частью набора команд target_...() и становятся еще более важными на этапах установки и упаковки проекта. Думайте о PRIVATE как о значении для самой цели, INTERFACE - для вещей, которые связываются с целью, а PUBLIC - это оба поведения вместе. Хотя может возникнуть соблазн просто пометить все как PUBLIC, это может привести к ненужному раскрытию зависимостей, выходящих за рамки нужных целей. Это может повлиять на время сборки, а частные зависимости могут быть навязаны другим целям, которые не должны о них знать. Это, в свою очередь, сильно влияет на другие области, такие как видимость символов (подробно обсуждается в разделе 20.5, "Видимость символов"). Поэтому лучше начинать работу с зависимостью как с ПРИВАТНОЙ и делать ее ПУБЛИЧНОЙ только тогда, когда становится ясно, что зависимость необходима тем, кто связывает ее с целью.

Ключевое слово INTERFACE, как правило, используется в основном для импортированных или интерфейсных библиотечных целей, или иногда для добавления недостающих зависимостей к цели, определенной в части проекта, которую разработчику не разрешается изменять. Примерами могут служить части проекта, которые были написаны для старых версий CMake и поэтому не используют команды target_...(), или внешние библиотеки с импортированными целями, в которых отсутствуют некоторые важные флаги, необходимые для связывающихся с ними целей. Для всех команд target_...(), кроме target_link_libraries(), указанная цель может быть определена в любом месте проекта, единственным требованием является то, что цель была создана в какой-то момент до вызова команды target_...(). Таким образом, прикрепление дополнительных зависимостей интерфейса компилятора к цели может быть сделано из любой части проекта, но прикрепление зависимостей интерфейса компоновщика может быть сделано только из той же области каталогов, в которой создана цель. Это ограничение активно обсуждается разработчиками CMake и может быть снято в одном из будущих выпусков.

Глава 15. Требования к языку

С постоянным развитием языков C и C++ от разработчиков все чаще требуется понимание флагов компилятора и компоновщика, которые обеспечивают поддержку той версии C и/или C++, которую использует их код. Разные компиляторы используют разные флаги, но даже при использовании одного и того же компилятора и компоновщика флаги могут использоваться для выбора различных реализаций стандартной библиотеки.

В те времена, когда поддержка C++11 была относительно новой, в CMake не было прямой поддержки выбора стандарта, поэтому проектам приходилось самостоятельно определять необходимые флаги. В CMake 3.1 были введены функции, позволяющие выбирать стандарт C и C++ последовательным и удобным способом, абстрагируясь от различных различий компиляторов и компоновщиков. Эта поддержка была расширена в последующих версиях и начиная с CMake 3.6 охватывает большинство распространенных компиляторов (CMake 3.2 добавил большую часть поддержки компиляторов, 3.6 - компилятор Intel).

CMake предоставляет два основных метода для определения требований к языку. Первый заключается в непосредственном задании стандарта языка, а второй - в том, чтобы позволить проектам указать необходимые им языковые особенности и позволить CMake выбрать соответствующий стандарт языка. Хотя функциональность в основном определяется языками C и C++, другие языки и псевдоязыки, такие как CUDA, также поддерживаются.

15.1. Прямая установка языкового стандарта

Самый простой способ для проекта контролировать языковые стандарты, используемые сборкой, - задавать их напрямую. При таком подходе разработчикам не нужно знать или указывать отдельные языковые особенности, используемые в коде, им достаточно задать одно число, указывающее на стандарт, который, по мнению кода, поддерживается. Это не только просто для понимания и использования, но также имеет то преимущество, что относительно просто обеспечить использование одного и того же стандарта в рамках всего проекта. Это становится важным на этапе компоновки, когда последовательная стандартная библиотека должна использоваться во всех компонуемых библиотеках и объектных файлах.

Как обычно в CMake, свойства цели управляют тем, какой стандарт будет использоваться при сборке исходных текстов этой цели и при компоновке конечного исполняемого файла или разделяемой библиотеки. Для данного языка есть три целевых свойства, связанных с указанием стандарта (<LANG> должен быть одним из C или CXX, а для более новых версий CMake также возможен вариант CUDA):

<LANG>_STANDARD

Определяет стандарт языка, который проект хочет использовать для указанной цели. Начиная с первой версии CMake, поддерживающей эту возможность, допустимыми значениями для C_STANDARD являются 90, 99 или 11, а для CXX_STANDARD допустимыми значениями являются 98, 11 или 14. Начиная с CMake 3.8, также поддерживается значение 17, а начиная с CMake 3.12, можно использовать значение 20. Можно предположить, что последующие версии CMake будут добавлять поддержку других языковых стандартов по мере их развития. CMake 3.8 также поддерживает CUDA_STANDARD со значениями 98 или 11, что является специфичной для CUDA версией того, что обычно контролирует CXX_STANDARD. При создании цели начальное значение этого свойства берется из переменной CMAKE_<LANG>_STANDARD.

<LANG>_STANDARD_REQUIRED

В то время как свойство <LANG>_STANDARD определяет языковой стандарт, который нужен проекту, <LANG>_STANDARD_REQUIRED определяет, будет ли этот языковой стандарт рассматриваться как минимальное требование или просто как рекомендация "использовать, если доступно". Интуитивно можно ожидать, что <LANG>_STANDARD будет требованием по умолчанию, но, к лучшему или худшему, свойства

<LANG>_STANDARD_REQUIRED по умолчанию выключены. Если запрошенный стандарт не поддерживается компилятором, CMake не останавливается с ошибкой, а распадается на более ранний стандарт. Такое поведение при распаде часто неожиданно для начинающих разработчиков и на практике может стать причиной путаницы. Таким образом, для большинства проектов при указании свойства <LANG>_STANDARD соответствующее свойство <LANG>_STANDARD_REQUIRED почти всегда должно быть установлено в true, чтобы гарантировать, что конкретный запрошенный стандарт будет рассматриваться как обязательное требование. При создании цели начальное значение этого свойства берется из переменной CMAKE_<LANG>_STANDARD_REQUIRED.

<LANG>_РАСШИРЕНИЯ

Многие компиляторы поддерживают свои собственные расширения стандарта языка, и для включения или отключения этих расширений обычно предусмотрен флаг компилятора и/или компоновщика. Свойство цели <LANG>_EXTENSIONS контролирует, включены ли эти расширения для данной конкретной цели. Для некоторых компиляторов/ компоновщиков этот параметр может изменить стандартную библиотеку, с которой скомпонована цель (см. примеры ниже). Имейте в виду, что для многих компиляторов/линкеров один и тот же флаг используется для управления стандартом языка и включением или невключением расширений. Одним из следствий этого является то, что если проект устанавливает свойство <LANG>_EXTENSIONS, он также должен установить свойство <LANG>_STANDARD, иначе <LANG>_EXTENSIONS может быть фактически проигнорирован. Когда создается цель, начальное значение свойства <LANG>_EXTENSIONS берется из переменной CMAKE_<LANG>_EXTENSIONS.

На практике проекты чаще всего устанавливают переменные, которые обеспечивают значения по умолчанию для вышеуказанных свойств цели, а не устанавливают свойства цели напрямую. Это гарантирует, что все цели в проекте будут построены согласованным образом с совместимыми настройками. Более того, настоятельно рекомендуется, чтобы проекты устанавливали все три свойства/переменные, а не только некоторые из них. Значения по умолчанию для <LANG>_STANDARD_REQUIRED и <LANG>_EXTENSIONS оказались относительно неинтуитивными для многих разработчиков, поэтому, явно задавая их, проект дает понять, какого стандартного поведения он ожидает. Несколько примеров помогут продемонстрировать типичное использование.

```
# Require C++11 and disable extensions for all targets
set(CMAKE_CXX_STANDARD          11)
set(CMAKE_CXX_STANDARD_REQUIRED ON)
set(CMAKE_CXX_EXTENSIONS        OFF)
```

При использовании GCC или Clang, вышеупомянутое обычно добавляет флаг -std=c++11. Он также может добавить флаг компоновщика, например -stdlib=libc++, в зависимости от платформы. Для компиляторов Visual Studio до VS2015 Update 3 никакие флаги не добавляются, так как компилятор либо поддерживает C++11 по умолчанию, либо вообще не поддерживает C++11. Обратите внимание, что начиная с Visual Studio 15 Update 3, компилятор поддерживает указание стандарта C++, но только для C++14 и более поздних версий, и C++14 является настройкой по умолчанию.

Для сравнения, следующий пример запрашивает более позднюю версию C++ и включает расширения компилятора, в результате чего вместо флага компилятора GCC/Clang появляется флаг -std=gnu++14. Компиляторы Visual Studio опять же могут поддерживать запрашиваемый стандарт по умолчанию или нет, в зависимости от версии компилятора. Если используемый компилятор не поддерживает требуемый стандарт C++, CMake настроит компилятор на использование самого последнего стандарта C++, который он поддерживает.

```
# Use C++14 if available and allow compiler extensions for all targets
set(CMAKE_CXX_STANDARD          14)
set(CMAKE_CXX_STANDARD_REQUIRED OFF)
set(CMAKE_CXX_EXTENSIONS        ON)
```

Ситуация для языка C очень похожа. В следующем примере показано, как установить детали стандарта C, на этот раз только для конкретной цели:

```
# Build target foo with C99, no compiler extensions
set_target_properties(foo PROPERTIES
    C_STANDARD          99
    C_STANDARD_REQUIRED ON
    C_EXTENSIONS        OFF
)
```

Следует отметить, что <LANG>_STANDARD технически определяет минимальный стандарт, а не обязательно точное требование. В некоторых ситуациях CMake может выбрать более новый стандарт из-за требований к компиляционным возможностям (об этом речь пойдет далее).

15.2. Установка языкового стандарта по требованиям к функциям

Прямая установка языкового стандарта для цели или для всего проекта - самый простой способ управления стандартными требованиями. Это наиболее подходящий подход, когда разработчики проекта знают, какая версия языка предоставляет возможности, используемые в коде проекта. Это особенно удобно, когда используется большое количество функций языка, поскольку каждую функцию не нужно явно указывать. В некоторых случаях, однако, разработчики могут предпочесть указать, какие языковые возможности использует их код, и предоставить CMake выбрать соответствующий языковой стандарт. Это имеет то преимущество, что, в отличие от прямого указания стандарта, требования к компилируемым возможностям могут быть частью интерфейса цели и, следовательно, могут быть применены к другим целям, связывающимся с ней.

Требования к функциям компиляции контролируются параметрами COMPILE_FEATURES и INTERFACE_COMPILE_FEATURES целевые свойства, но эти свойства обычно заполняются с помощью команды target_compile_features(), а не управляются напрямую. Эта команда очень похожа на другие команды target_...(), предоставляемые CMake:

```
target_compile_features(targetName
     <PRIVATE|PUBLIC|INTERFACE> feature1 [feature2 ...]
    [<PRIVATE|PUBLIC|INTERFACE> feature3 [feature4 ...]]
    ...
)
```

Ключевые слова PRIVATE, PUBLIC и INTERFACE имеют свои обычные значения, управляя тем, как перечисленные свойства должны применяться. Функции PRIVATE заполняют свойство COMPILE_FEATURES, которое применяется к самой цели. Функции, указанные с ключевым словом INTERFACE, заполняют свойство INTERFACE_COMPILE_FEATURES, которое применяется к любой цели, связанной с targetName.

Характеристики, указанные как PUBLIC, будут добавлены к обоим свойствам и поэтому будут применяться как к самой цели, так и к любой другой цели, которая на нее ссылается.

Каждая функция должна быть одной из функций, поддерживаемых базовым компилятором. CMake предоставляет два списка известных возможностей: CMAKE_<LANG>_KNOWN_FEATURES, который содержит все известные функции для данного языка, и CMAKE_<LANG>_COMPILE_FEATURES, который содержит только те функции, которые поддерживаются компилятором. Если запрашиваемая функция не поддерживается компилятором, CMake сообщит об ошибке. Разработчики могут счесть документацию CMake для переменных CMAKE_<LANG>_KNOWN_FEATURES особенно полезным ресурсом, поскольку в ней не только перечислены функции, поддерживаемые данной версией CMake, но и содержатся ссылки на стандартные документы, относящиеся к каждой функции. Обратите внимание, что не все функциональные возможности, предоставляемые конкретной версией языка, могут быть явно указаны с помощью функций компиляции. Например, новые типы C++ STL, функции и т.д. не имеют соответствующей возможности.

Начиная с CMake 3.8, доступна мета-функция для каждого языка, указывающая на определенный стандарт языка, а не на конкретную функцию компиляции. Эти мета-функции имеют форму <lang>_std_<value>, и если они указаны как необходимая функция компиляции, CMake обеспечит использование флагов компилятора, которые поддерживают этот стандарт языка. Например, чтобы добавить функцию компиляции, которая гарантирует, что цель и все, что ссылается на нее, имеет поддержку C++14, можно использовать следующее:

```
target_compile_features(targetName PUBLIC cxx_std_14)
```

Если проект должен поддерживать версии CMake более ранние, чем 3.8, то вышеуказанная мета-функция будет недоступна. В таких случаях каждую функцию компиляции придется перечислять отдельно, что может быть непрактично и, скорее всего, будет неполным. Это, как правило, ограничивает полезность функций компиляции в целом, и проекты часто предпочитают устанавливать стандарт языка через целевые свойства, описанные в предыдущем разделе.

В ситуациях, когда у цели задано свойство <LANG>_STANDARD и указаны особенности компиляции (напрямую или транзитивно, как результат особенностей INTERFACE от того, на что она ссылается), CMake будет применять более сильное требование стандарта. В следующем примере foo будет собран на C++14, bar - на C++17, а guff - на C++14:

```
set_target_properties(foo PROPERTIES CXX_STANDARD 11)
target_compile_features(foo PUBLIC cxx_std_14)

set_target_properties(bar PROPERTIES CXX_STANDARD 17)
target_compile_features(bar PRIVATE cxx_std_11)

set_target_properties(guff PROPERTIES CXX_STANDARD 11)
target_link_libraries(guff PRIVATE foo)
```

Обратите внимание, что это может означать использование более нового стандарта языка, чем тот, на который рассчитывал проект, что в некоторых случаях может привести к ошибкам компиляции. Например, в C++17 удален std::auto_ptr, поэтому если код ожидает компиляции с более старым стандартом языка и все еще использует std::auto_ptr, он может не скомпилироваться, если инструментарий строго придерживается этого удаления.

15.2.1. Обнаружение и использование дополнительных языковых функций

Некоторые проекты имеют возможность обрабатывать определенные функции языка, которые поддерживаются или не поддерживаются. Например, они могут предоставлять реализацию с отступлением или определять перегрузки определенных функций только в том случае, если они поддерживаются компилятором. Проект может поддерживать некоторые функции компилятора как необязательные, например, ключевые слова, предназначенные для руководства разработчика или для повышения способности компилятора отлавливать распространенные ошибки. Такие ключевые слова C++, как final и override, являются распространенными примерами этого.

CMake предоставляет несколько способов обработки описанных выше сценариев. Один из подходов заключается в использовании выражений генератора для условной установки определений компилятора или каталогов include на основе наличия определенной функции компилятора. Эти выражения могут быть немного многословными, но они обеспечивают большую гибкость и поддерживают очень точную обработку функциональности, основанной на возможностях. Рассмотрим следующий пример:

```
add_library(foo ...)

# Make override a feature requirement only if available
target_compile_features(foo PUBLIC
    $<$<COMPILE_FEATURES:cxx_override>:cxx_override>
)

# Define the foo_OVERRIDE symbol so it provides the
# override keyword if available or empty otherwise
target_compile_definitions(foo PUBLIC
    $<$<COMPILE_FEATURES:cxx_override>:-Dfoo_OVERRIDE=override>
    $<$<NOT:$<COMPILE_FEATURES:cxx_override>>:-Dfoo_OVERRIDE>
)
```

Вышеуказанное позволит компилировать код, подобный следующему, для любого компилятора C++, независимо от того, поддерживает ли он ключевое слово override или нет:

```
class MyClass : public Base
{
public:
    void  func() foo_OVERRIDE;
    ...
};
```

В дополнение к ключевому слову override, ряд других функций также может иметь аналогичный условно определенный символ, используемый примерно таким же образом. Такие ключевые слова C++, как final, constexpr, noexcept и другие, могут быть использованы, если они доступны, или опущены, если не поддерживаются компилятором, и при этом выдавать корректный и правильный код. Другие ключевые слова, такие как nullptr и static_assert, имеют альтернативные реализации, которые можно использовать, если ключевое слово не поддерживается компилятором. Определение выражений генератора для каждой функции, чтобы охватить поддерживаемые и неподдерживаемые случаи, было бы утомительным и потенциально более хрупким, но более удобный механизм предоставляет CMake через свою систему модулей. Модуль WriteCompilerDetectionHeader определяет функцию write_compiler_detection_header(), которая автоматизирует такую обработку. Она создает заголовочный файл, который исходные тексты проекта могут

#include, чтобы подхватить соответствующие определения компилятора. Упрощенная версия этой функции, показывающая только обязательные опции, может быть описана следующим образом.

```
write_compiler_detection_header(
    FILE      fileName
    PREFIX    prefix
    COMPILERS compiler1 [compiler2 ...]
    FEATURES  feature1 [feature2 ...]
)
```

Функция выписывает в указанный fileName заголовок на языке C/C++, содержимое которого будет содержать соответствующие макросы, определенные для каждой перечисленной функции. Каждая функция будет иметь макрос вида prefix_COMPILER_UPPERCASEFEATURE, значение которого будет равно 1 или 0 в зависимости от того, поддерживается или нет данная функция используемым компилятором. Некоторые функции могут также иметь макрос вида prefix_UPPERCASEFEATURE, который обеспечивает наиболее подходящую реализацию этой функции для каждого из названных компиляторов, включая различные версии компилятора, где это уместно.

Это лучше всего продемонстрировать на примере. Рассмотрим проект на C++, который может использовать ключевые слова override, final и nullptr, если они доступны, и который нацелен на поддержку компиляторов GNU, Clang, Visual Studio и Intel на любой из платформ, поддерживаемых этими компиляторами. Проект также определит конструкторы перемещений, если компилятор поддерживает ссылки rvalue. В каталоге сборки будет записан один заголовок foo_compiler_detection.h, и каждое имя макроса будет начинаться со строки foo_:

```
include(WriteCompilerDetectionHeader)
write_compiler_detection_header(
    FILE      foo_compiler_detection.h
    PREFIX    foo
    COMPILERS GNU Clang MSVC Intel
    FEATURES  cxx_override
              cxx_final
              cxx_nullptr
              cxx_rvalue_references
)
```

Пример кода на C++, использующего макросы, определенные выше, может выглядеть следующим образом:

```
#include "foo_compiler_detection.h"

class MyClass foo_FINAL : public Base
{
public:
#if foo_COMPILER_CXX_RVALUE_REFERENCES
    MyClass(MyClass&& c);
#endif
    void  func1() foo_OVERRIDE;
    void  func2(int* p = foo_NULLPTR);
};
```

Цель, потребляющая вышеупомянутый исходный файл, все равно должна иметь соответствующий языковой стандарт, но в данном случае, поскольку доступны реализации с отступлением, желаемый стандарт может быть указан, но не обязателен:

```
set(CMAKE_CXX_STANDARD          11)
set(CMAKE_CXX_STANDARD_REQUIRED OFF)
set(CMAKE_CXX_EXTENSIONS        OFF)

add_library(foo MyClass.cpp)

# The header is written to the build directory
# so ensure we add that to the header search path
target_include_directories(foo
    PUBLIC "${CMAKE_CURRENT_BINARY_DIR}"
)
```

В CMake предусмотрены запасные варианты реализации довольно многих функций, все они описаны в документации модуля WriteCompilerDetectionHeader. Команда write_compiler_detection_header() также принимает ряд дополнительных аргументов, не упомянутых здесь, которые позволяют контролировать структуру и расположение генерируемых заголовочных файлов и добавлять произвольное содержимое в начало и конец генерируемого заголовка. Заинтересованный читатель должен обратиться к документации модуля CMake для получения подробной информации.

Прежде чем приступать к использованию модуля WriteCompilerDetectionHeader, проектам следует тщательно обдумать, стоит ли использование заголовка обнаружения компилятора таких сложностей. Он может стать отличным инструментом для расширения диапазона компиляторов, которые может поддерживать проект. В частности, долгоживущие проекты могут счесть его полезной ступенькой в обновлении своей кодовой базы до более современных возможностей языка, в то время как на некоторых платформах им все еще необходимо поддерживать старые компиляторы. Одним из основных недостатков использования модуля является потенциальное снижение читаемости исходного кода. Также может быть трудно добиться того, чтобы во всех исходных файлах использовались альтернативные (генерируемые) имена символов вместо стандартных ключевых слов языка, так как для некоторых разработчиков это может показаться не совсем естественным.

15.3. Рекомендуемые практики

В проектах следует избегать прямой установки флагов компилятора и компоновщика для управления используемым стандартом языка. Необходимые флаги варьируются от компилятора к компилятору, поэтому надежнее, удобнее и проще использовать возможности CMake и позволить ему заполнить флаги соответствующим образом. Файл CMakeLists.txt также будет более четко выражать намерения, поскольку вместо часто загадочных необработанных флагов компилятора и компоновщика используются человекочитаемые переменные и свойства.

Самый простой метод управления требованиями языкового стандарта - это использование переменных CMAKE_<LANG>_STANDARD, CMAKE_<LANG>_STANDARD_REQUIRED и CMAKE_<LANG>_EXTENSIONS. Они могут использоваться для установки стандартного поведения языка для всего проекта, обеспечивая последовательное использование во всех целях. Это поможет избежать проблем с линковкой несовместимых стандартных библиотек и других проблем с линковкой. В идеале эти переменные должны быть установлены сразу после первой команды project() в файле CMakeLists.txt верхнего уровня. В проектах всегда следует задавать все три переменные вместе, чтобы было понятно, как должны выполняться требования стандарта языка и разрешены ли расширения компилятора. Отсутствие CMAKE_<LANG>_STANDARD_REQUIRED или

CMAKE_<LANG>_EXTENSIONS часто может привести к неожиданному поведению, поскольку значения по умолчанию могут быть не такими, как интуитивно ожидают некоторые разработчики.

Если языковой стандарт должен соблюдаться только для некоторых целей, а не для других, то

Свойства целей <LANG>_STANDARD, <LANG>_STANDARD_REQUIRED и <LANG>_EXTENSIONS могут быть установлены для отдельных целей, а не для всего проекта. Эти свойства ведут себя так, как если бы они были ПРИВАТНЫМИ, то есть они определяют требования только для данной цели, а не для всего, что на нее ссылается. Таким образом, на проект ложится большее бремя по обеспечению того, что все цели имеют правильно указанные детали языкового стандарта. На практике обычно проще и надежнее использовать переменные для задания языковых требований в масштабах всего проекта, чем использовать свойства для каждой цели. Предпочтительнее использовать переменные, если в проекте нет необходимости в различном поведении языкового стандарта для разных целей.

Если используется CMake 3.8 или более поздняя версия, функции компиляции могут быть использованы для указания желаемого стандарта языка для каждой цели. Команда target_compile_features() упрощает эту задачу и четко определяет, являются ли такие требования PRIVATE, PUBLIC или INTERFACE. Основное преимущество указания языкового требования таким образом заключается в том, что оно может быть транзитивно применено к другим целям через отношения PUBLIC и INTERFACE. Заметим, однако, что предоставляются только эквиваленты целевых свойств <LANG>_STANDARD и <LANG>_STANDARD_REQUIRED, поэтому целевое свойство <LANG>_EXTENSIONS или переменная CMAKE_<LANG>_EXTENSIONS все еще должны использоваться для контроля того, разрешены ли расширения компилятора. Эти свойства/переменные ...EXTENSIONS часто вступают в силу, только если соответствующий <LANG>_STANDARD также установлен из-за того, что компиляторы и компоновщики часто объединяют эти два параметра в один флаг, поэтому в конечном итоге трудно избежать необходимости указывать <LANG>_STANDARD даже при использовании возможностей компиляции. В результате, проекты могут счесть более простым и надежным предпочесть использование общепроектных переменных.

Указание отдельных функций компиляции обеспечивает тонкий контроль над требованиями к языку на уровне каждой цели. На практике разработчикам трудно обеспечить явное указание всех функций, используемых целью, поэтому всегда будет существовать вопрос о том, правильно ли определены требования к языку. Кроме того, они могут легко устаревать по мере развития кода. В большинстве проектов определение требований к языку таким способом, вероятно, покажется утомительным и хрупким, поэтому их следует использовать только в том случае, если ситуация явно того требует. Работа с очень недавними дополнениями к языку, например, при использовании предлагаемых возможностей для предстоящего выпуска языка, является одним из сценариев, когда компилируемые возможности могут быть полезным подходом, если CMake поддерживает такие возможности. В целом, однако, проекты должны предпочитать использовать переменные или свойства для задания требований к языку на более высоком уровне для лучшей сопровождаемости и надежности. В качестве альтернативы, установка стандарта через мета-функцию компиляции, например, cxx_std_11, также позволяет избежать многих проблем, связанных с установкой отдельных функций. Для более современных стандартов языка CMake отходит от определения отдельных функций и вместо этого просто предоставляет мета-функцию.

Проекты могут определять доступные функции компиляции и предоставлять реализацию того, доступна ли функция или нет. CMake даже предоставляет некоторые удобные макросы с помощью функции
модуль WriteCompilerDetectionHeader, которые облегчают эту задачу. Как правило, проекты должны рассматривать использование этих функций только в качестве переходного пути при обновлении старой базы кода для использования новых возможностей языка, поскольку они, как правило, менее естественны для разработчиков и могут ухудшить читаемость кода. Заметным исключением из этого являются проекты, предназначенные для широкого спектра компиляторов, где поддержка языковых стандартов может варьироваться. В этом случае опциональная поддержка определенных языковых особенностей может помочь уменьшить количество предупреждений компилятора и выявить ошибки кодирования при использовании

более современных компиляторов. Выгоду следует сопоставить с повышенной сложностью и потенциальным снижением читабельности и менее естественным стилем для большинства разработчиков.

Глава 16. Типы целей

CMake поддерживает широкий спектр типов целей, а не только простые исполняемые файлы и библиотеки, представленные в главе 4 "*Сборка простых целей*". Можно определить различные целевые типы, которые действуют как ссылки на другие объекты, а не собираются сами. Они могут быть использованы для сбора вместе переходных свойств и зависимостей без создания собственных двоичных файлов, или они даже могут быть разновидностью библиотеки, которая является просто коллекцией объектных файлов, а не традиционной статической или разделяемой библиотекой. Многие вещи могут быть абстрагированы в качестве цели, чтобы скрыть сложности, связанные с различиями платформ, расположением в файловой системе, именами файлов и так далее. В этой главе рассматриваются все эти различные типы целей и обсуждаются их применения.

Еще одна категория целей - утилиты или пользовательские цели. Они могут использоваться для выполнения произвольных команд и определения пользовательских правил сборки, позволяя проектам реализовать практически любое необходимое поведение. Они имеют свои собственные специальные команды и уникальное поведение и подробно рассматриваются в следующей главе.

16.1. Исполняемые файлы

Команда add_executable() имеет не только форму, представленную в главе 4 "*Создание простых целей*". Существуют также две другие формы, которые можно использовать для определения исполняемых целей, ссылающихся на другие объекты. Полный набор поддерживаемых форм следующий:

```
add_executable(targetName [WIN32] [MACOSX_BUNDLE]
               [EXCLUDE_FROM_ALL]
               source1 [source2 ...])
add_executable(targetName IMPORTED [GLOBAL])
add_executable(aliasName ALIAS targetName)
```

Форма IMPORTED может быть использована для создания цели CMake для существующего исполняемого файла, а не для того, который собран проектом. Создав цель для исполняемого файла, другие части проекта могут обращаться с ним так же, как и с любым другим исполняемым файлом, созданным самим проектом (с некоторыми ограничениями). Наиболее существенным преимуществом является то, что его можно использовать в контекстах, где CMake автоматически заменяет имя цели ее расположением на диске, например, при выполнении команд для тестов или пользовательских задач (и то, и другое рассматривается в последующих главах). Одно из немногих отличий от обычной цели заключается в том, что импортированные цели нельзя установить, о чем рассказывается в главе 25, *Установка*.

При определении импортируемой исполняемой цели необходимо установить определенные свойства цели, прежде чем она сможет быть полезной. Большинство соответствующих свойств для любой импортируемой цели имеют имена, начинающиеся с
IMPORTED, но для исполняемых файлов наиболее важными являются IMPORTED_LOCATION и IMPORTED_LOCATION_<CONFIG>. Когда необходимо определить местоположение импортируемого исполняемого файла, CMake сначала посмотрит на свойство, специфичное для конфигурации, и только если оно не установлено, посмотрит на более общее свойство IMPORTED_LOCATION. Как правило, расположение не обязательно должно быть специфичным для конфигурации, поэтому очень часто задается только IMPORTED_LOCATION.

При определении без ключевого слова GLOBAL импортированная цель будет видна только в области видимости текущего каталога и ниже, но добавление GLOBAL делает цель видимой везде. В отличие от этого, обычные

исполняемые цели, создаваемые проектом, всегда глобальны. Причины этого и некоторые последствия уменьшения видимости цели рассматриваются в разделе 16.3, "Продвижение импортированных целей".

Цель ALIAS - это просто способ ссылки на другую цель в CMake, доступный только для чтения. Он не создает новую цель сборки с именем псевдонима. Псевдонимы могут указывать только на реальные цели (т.е. псевдоним псевдонима не поддерживается), их нельзя устанавливать или экспортировать (оба эти вопроса рассматриваются в главе 25, *Установка*). До версии CMake 3.11 импортированные цели также не могли быть псевдонимами, но CMake 3.11 ослабил некоторые ограничения, разрешив псевдонимы импортированных целей, но только тех импортированных целей, которые имеют глобальную видимость.

16.2. Библиотеки

Команда add_library() также имеет несколько различных форм. Базовая форма, представленная в главе 4 *"Создание простых целей"*, может быть использована для определения обычных типов библиотек, с которыми знакомы большинство разработчиков, а также может быть использована для определения объектных библиотек, которые представляют собой просто набор объектных файлов, не объединенных в единый архив или общую библиотеку. Расширенная базовая форма команды выглядит следующим образом:

```
add_library(targetName [STATIC | SHARED | MODULE | OBJECT]
            [EXCLUDE_FROM_ALL]
            source1 [source2 ...])
```

До версии CMake 3.12 объектные библиотеки не могли быть связаны, как другие типы библиотек (т.е. их нельзя было использовать с target_link_libraries()), они требовали использования выражения генератора вида $<TARGET_OBJECTS:objLib> как части списка источников другого исполняемого файла или библиотеки. Поскольку они не могут быть связаны, они не обеспечивают переходных зависимостей к целям, к которым они добавляются в качестве объектов/источников. Это может сделать их менее удобными, чем другие типы библиотек, поскольку пути поиска заголовков, определения компилятора и т.д. должны быть вручную перенесены на цели, к которым они добавляются.

В CMake 3.12 появились возможности, благодаря которым объектные библиотеки ведут себя более похоже на другие типы библиотек, но с некоторыми оговорками. Начиная с CMake 3.12, объектные библиотеки *можно* использовать с помощью target_link_libraries(), либо в качестве объекта, к которому добавляется библиотека (т.е. первый аргумент команды), либо в качестве одной из добавляемых библиотек. Но поскольку они добавляют объектные файлы, а не реальные библиотеки, их переходная природа более ограничена, чтобы предотвратить многократное добавление объектных файлов к потребляющим целям. Упрощенное объяснение заключается в том, что объектные файлы добавляются только к цели, которая ссылается *непосредственно* на объектную библиотеку, а не транзитивно за ее пределы. Однако требования к использованию объектной библиотеки распространяются транзитивно точно так же, как и обычной библиотеки.

Некоторые разработчики могут посчитать объектные библиотеки более естественными, если они имеют опыт работы в проектах без CMake, где цели определялись на основе исходных текстов или объектных файлов, а не связанного набора статических библиотек. В целом, однако, при наличии выбора, статические библиотеки обычно являются более удобным выбором в проектах CMake. Прежде чем полагаться на расширенные возможности объектных библиотек в версии 3.12, подумайте, не является ли обычная статическая библиотека более подходящей и, в конечном счете, более простой в использовании.

Как и исполняемые файлы, библиотеки также могут быть определены как импортируемые цели. Они активно используются в конфигурационных файлах, создаваемых при упаковке, или в реализации модулей Find (рассматриваются в главе 23, *Finding Things* и главе 25, *Installing*), но имеют ограниченное применение вне этих

контекстов. Они не определяют библиотеку, которая должна быть собрана проектом, скорее они действуют как ссылка на библиотеку, которая предоставляется извне (например, она уже существует в системе, собрана каким-то процессом вне текущего проекта CMake или предоставлена пакетом, частью которого является файл конфигурации).

```
add_library(targetName (STATIC | SHARED | MODULE | OBJECT | UNKNOWN)
            IMPORTED [GLOBAL]
)
```

Тип библиотеки должен быть указан сразу после targetName. Если тип библиотеки, на которую будет ссылаться новая цель, известен, его следует указать как таковой. Это позволит CMake в различных ситуациях рассматривать импортированную цель как обычную библиотечную цель названного типа. Тип может быть установлен в OBJECT только в CMake 3.9 или более поздней версии (до этой версии импортируемые объектные библиотеки не поддерживались). Если тип библиотеки неизвестен, следует указать тип UNKNOWN, в этом случае CMake просто использует полный путь к библиотеке без дальнейшей интерпретации в таких местах, как командные строки компоновщика. Это означает меньшее количество проверок и, в случае сборок под Windows, отсутствие обработки импорта библиотек DLL.

За исключением библиотек OBJECT, расположение в файловой системе, которое представляет импортируемая цель, должно быть указано свойствами IMPORTED_LOCATION и/или IMPORTED_LOCATION_<CONFIG> (т.е. так же, как и для импортируемых исполняемых файлов). В случае платформ Windows необходимо задать два свойства: IMPORTED_LOCATION - расположение DLL и IMPORTED_IMPLIB - расположение соответствующей импортируемой библиотеки, которая обычно имеет расширение файла .lib (варианты ..._<CONFIG> этих свойств также могут быть установлены и будут иметь приоритет). Для объектных библиотек вместо вышеуказанных свойств расположения свойство IMPORTED_OBJECTS должно быть установлено в список объектных файлов, которые представляет импортируемая цель.

Импортируемые библиотеки также поддерживают ряд других целевых свойств, большинство из которых обычно можно оставить без внимания или они автоматически устанавливаются CMake. Разработчикам, которым необходимо вручную писать конфигурационные пакеты, следует обратиться к справочной документации CMake, чтобы узнать о других целевых свойствах IMPORTED_..., которые могут иметь отношение к их ситуации. Однако большинство проектов полагаются на то, что CMake будет генерировать такие файлы для них, поэтому необходимость в этом должна быть довольно редкой.

По умолчанию импортированные библиотеки определяются как локальные цели, то есть они видны только в области видимости текущего каталога и ниже. Ключевое слово GLOBAL может быть использовано для того, чтобы сделать их видимыми во всем мире, как и другие обычные цели. Изначально библиотека может быть создана без ключевого слова GLOBAL, но позже она может быть переведена в глобальную видимость. Эта тема подробно рассматривается в разделе 16.3 "Перемещение импортированных целей" ниже.

```
# Windows-specific example of imported library
add_library(myWindowsLib SHARED IMPORTED)
set_target_properties(myWindowsLib PROPERTIES
    IMPORTED_LOCATION /some/path/bin/foo.dll
    IMPORTED_IMPLIB   /some/path/lib/foo.lib
)
```

```
# Assume FOO_LIB holds the location of the library but its type is unknown
add_library(mysteryLib UNKNOWN IMPORTED)
set_target_properties(mysteryLib PROPERTIES
    IMPORTED_LOCATION ${FOO_LIB}
)
```

```
# Imported object library, Windows example shown
add_library(myObjLib OBJECT IMPORTED)
set_target_properties(myObjLib PROPERTIES
    IMPORTED_OBJECTS /some/path/obj1.obj    # These .obj files would be .o
                     /some/path/obj2.obj    # on most other platforms
)

# Regular executable target using imported object library.
# Platform differences are already handled by myObjLib.
add_executable(myExe $<TARGET_SOURCES:myObjLib>)
```

Другая форма команды add_library() позволяет определять интерфейсные библиотеки. Обычно они не представляют собой физическую библиотеку, вместо этого они служат для сбора требований к использованию и зависимостей, которые будут применяться ко всему, что на них ссылается. Популярный пример их использования - библиотеки только для заголовков, когда нет физической библиотеки, которую нужно линковать, но пути поиска заголовков, определения компилятора и т.д. должны быть перенесены на все, что использует заголовки.

```
add_library(targetName INTERFACE [IMPORTED [GLOBAL]])
```

Все различные команды target_...() можно использовать вместе с их ключевыми словами INTERFACE для определения требований к использованию библиотеки интерфейсов. Можно также задать соответствующие свойства INTERFACE_... напрямую с помощью set_property() или set_target_properties(), но команды target_...() безопаснее и проще в использовании.

```
add_library(myHeaderOnlyToolkit INTERFACE)
target_include_directories(myHeaderOnlyToolkit
    INTERFACE /some/path/include
)
target_compile_definitions(myHeaderOnlyToolkit
    INTERFACE COOL_FEATURE=1
              $<$<COMPILE_FEATURES:cxx_std_11>:HAVE_CXX11>
)
add_executable(myApp ...)
target_link_libraries(myApp PRIVATE myHeaderOnlyToolkit)
```

В приведенном выше примере цель myApp ссылается на интерфейсную библиотеку myHeaderOnlyToolkit. Когда исходные тексты myApp компилируются, они будут иметь /some/path/include в качестве пути поиска заголовков, а также будут иметь определение компилятора COOL_FEATURE=1, предоставленное в командной строке компилятора. Если цель myApp собирается с включенной поддержкой C++11, то для нее также будет определен символ HAVE_CXX11. Заголовки в myHeaderOnlyToolkit могут использовать этот символ для определения того, что они объявляют и определяют, а не полагаться на символ __cplusplus, предусмотренный стандартом C++, значение которого часто ненадежно для ряда компиляторов.

Другое применение интерфейсных библиотек - это обеспечение удобства для компоновки большего набора библиотек, возможно, инкапсулируя логику, которая выбирает, какие библиотеки должны быть в этом наборе. Например:

```
# Regular library targets
add_library(algo_fast ...)
add_library(algo_accurate ...)
add_library(algo_beta ...)

# Convenience interface library
add_library(algo_all INTERFACE)
target_link_libraries(algo_all INTERFACE
    algo_fast
    algo_accurate
    $<$<BOOL:${ENABLE_ALGO_BETA}>:algo_beta>
)

# Other targets link to the interface library
# instead of each of the real libraries
add_executable(myApp ...)
target_link_libraries(myApp PRIVATE algo_all)
```

Вышеуказанное включит algo_beta в список библиотек для линковки только в том случае, если переменная опции CMake ENABLE_ALGO_BETA равна true. Тогда другие цели просто ссылаются на algo_all, а условное связывание algo_beta обрабатывается интерфейсной библиотекой. Это пример использования интерфейсной библиотеки для абстрагирования от деталей того, что на самом деле должно быть связано, определено и т.д., так что цели, ссылающиеся на них, не должны реализовывать эти детали самостоятельно. Это можно использовать для того, чтобы абстрагироваться от совершенно разных структур библиотек на разных платформах, переключать реализации библиотек на основе некоторых условий (переменные, выражения генератора и т.д.), предоставлять старое имя цели библиотеки, если структура библиотеки была рефакторингом (например, разделена на отдельные библиотеки) и т.д.

Хотя сценарии использования библиотек INTERFACE в целом хорошо понятны, добавление

Ключевое слово IMPORTED для получения библиотеки INTERFACE IMPORTED иногда может стать причиной путаницы. Такая комбинация обычно возникает, когда библиотека INTERFACE экспортируется или устанавливается для использования вне проекта. При использовании в другом проекте она по-прежнему служит целям библиотеки INTERFACE, но часть IMPORTED добавляется, чтобы указать, что библиотека взята откуда-то еще. В результате этого видимость библиотеки по умолчанию ограничивается текущей областью каталога, а не глобальной. За одним исключением, описанным ниже, добавление ключевого слова GLOBAL для получения комбинации ключевых слов INTERFACE IMPORTED GLOBAL приводит к библиотеке, имеющей мало практических различий по сравнению только с INTERFACE. Библиотека INTERFACE IMPORTED не обязана (и даже не имеет права) устанавливать IMPORTED_LOCATION.

До CMake 3.11 ни одна из команд target_...() не могла быть использована для установки свойств INTERFACE_... для любой библиотеки типа IMPORTED. Однако эти свойства *можно было* установить с помощью set_property() или set_target_properties(). В CMake 3.11 снято ограничение на использование команд target_...() для установки этих свойств, поэтому если раньше INTERFACE IMPORTED были очень похожи на обычные библиотеки IMPORTED, то в CMake 3.11 они стали гораздо ближе к обычным библиотекам INTERFACE по набору ограничений.

В следующей таблице приведена информация о том, что поддерживают различные комбинации ключевых слов:

Keywords	Visibility	Imported Location	Set Interface Properties	Installable
INTERFACE	Global	Prohibited	Any method	Yes
IMPORTED	Local	Required	Restricted*	No
IMPORTED GLOBAL	Global	Required	Restricted*	No
INTERFACE IMPORTED	Local	Prohibited	Restricted*	No
INTERFACE IMPORTED GLOBAL	Global	Prohibited	Restricted*	No

* Различные команды target_...() можно использовать для установки свойств INTERFACE_... только при использовании CMake

3.11 или более поздней версии. Свойства INTERFACE_... могут быть установлены с помощью set_property() или set_target_properties() в любой версии CMake.

Можно подумать, что количество различных комбинаций интерфейса и импортируемых библиотек слишком сложно и запутанно. Однако для большинства разработчиков импортированные цели обычно создаются за кулисами и ведут себя более или менее как обычные цели. Из всех комбинаций, приведенных в таблице выше, только обычные цели INTERFACE обычно определяются непосредственно в проекте. Глава 25, *Установка,* охватывает большую часть мотивации и механики других комбинаций.

Последняя форма команды add_library() предназначена для определения библиотеки псевдонимов:

```
add_library(aliasName ALIAS otherTarget)
```

Псевдоним библиотеки в основном аналогичен псевдониму исполняемого файла. Он действует как способ ссылки на другую библиотеку только для чтения, но не создает новую цель сборки. Библиотечные псевдонимы не могут быть установлены и не могут быть определены как псевдоним другого псевдонима. До CMake 3.11 библиотеки-псевдонимы нельзя было создавать для импортируемых целей, но, как и другие изменения, внесенные в CMake 3.11 для импортируемых целей, это ограничение было ослаблено, и стало возможным создавать псевдонимы для глобально видимых импортируемых целей.

Существует особенно распространенное использование псевдонимов библиотек, связанное с важной функцией, представленной в CMake 3.0. Для каждой библиотеки, которая будет установлена или упакована, обычно создается соответствующий псевдоним библиотеки с именем вида projNamespace::originalTargetName. Все такие псевдонимы в проекте обычно имеют одно и то же пространство имен projNamespace. Например:

```
# Any sort of real library (SHARED, STATIC, MODULE
# or possibly OBJECT)
add_library(myRealThings SHARED src1.cpp ...)
add_library(otherThings STATIC srcA.cpp ...)

# Aliases to the above with special names
add_library(BagOfBeans::myRealThings ALIAS myRealThings)
add_library(BagOfBeans::otherThings ALIAS otherThings)
```

В самом проекте другие цели будут ссылаться либо на настоящие цели, либо на цели с именем (оба варианта имеют одинаковый эффект). Мотивация для псевдонимов возникает, когда проект установлен и что-то другое ссылается на импортированные цели, созданные установленными/упакованными конфигурационными файлами. Эти конфигурационные файлы определяют импортируемые библиотеки с именами, а не с пустыми

оригинальными именами. Тогда потребляющий проект будет линковаться с именами с разнесенными именами. Например:

```
# Pull in imported targets from an installed package.
# See details in Chapter 23: Finding Things
find_package(BagOfBeans REQUIRED)

# Define an executable that links to the imported
# library from the installed package
add_executable(eatLunch main.cpp ...)
target_link_libraries(eatLunch PRIVATE
    BagOfBeans::myRealThings
)
```

Если в какой-то момент вышеупомянутый проект захочет включить проект BagOfBeans непосредственно в свою сборку вместо того, чтобы найти установленный пакет, он сможет сделать это без изменения связи, поскольку проект BagOfBeans предоставляет псевдоним для имени, разделенного по именам:

```
# Add BagOfBeans directly to this project, making
# all of its targets directly available
add_subdirectory(BagOfBeans)

# Same definition of linking relationship still works
add_executable(eatLunch main.cpp ...)
target_link_libraries(eatLunch PRIVATE
    BagOfBeans::myRealThings
)
```

Еще один важный аспект имен с двойной точкой с запятой (::) заключается в том, что CMake всегда будет рассматривать их как имя псевдонима или импортированной цели. Любая попытка использовать такое имя для другого типа цели приведет к ошибке. Возможно, более полезно то, что когда имя цели используется как часть вызова target_link_library(), если CMake не знает цели с таким именем, он выдаст ошибку во время генерации. Сравните это с обычным именем, которое CMake будет рассматривать как библиотеку, предположительно предоставляемую системой, если он не знает о цели с таким именем. Это может привести к тому, что ошибка станет очевидной лишь много позже, во время сборки.

```
add_executable(main main.cpp)
add_library(bar STATIC ...)
add_library(foo::bar ALIAS bar)

# Typo in name being linked to, CMake will assume a
# library called "bart" will be provided by the
# system at link time and won't issue an error.
target_link_libraries(main PRIVATE bart)

# Typo in name being linked to, CMake flags an error
# at generation time because a namespaced name must
# be a CMake target.
target_link_libraries(main PRIVATE foo::bart)
```

Поэтому надежнее ссылаться на имена с разнесенными именами, если они доступны. Проектам настоятельно рекомендуется определять псевдонимы с разнесенными именами, по крайней мере, для всех целей, которые предполагается устанавливать/паковать. Такие псевдонимы с разнесенными именами могут использоваться

даже в самом проекте, а не только в других проектах, использующих его в качестве предварительно собранного пакета или дочернего проекта.

16.3. Продвижение импортированных целей

Когда цели определены без ключевого слова GLOBAL, импортированные цели видны только в области видимости каталога, в котором они созданы или ниже. Такое поведение обусловлено их основным предполагаемым использованием, которое является частью модуля Find или файла конфигурации пакета. Все, что определяется модулем Find или файлом конфигурации пакета, как правило, должно иметь локальную видимость, поэтому они не должны добавлять глобально видимые цели. Это позволяет различным частям иерархии проекта извлекать одни и те же пакеты и модули с различными настройками, но не мешать друг другу.

Тем не менее, бывают ситуации, когда импортируемые цели должны быть созданы с глобальной видимостью, например, чтобы гарантировать, что одна и та же версия или экземпляр определенного пакета используется последовательно во всем проекте. Добавление ключевого слова GLOBAL при создании импортируемой библиотеки позволяет добиться этого, но проект может не контролировать команду, выполняющую создание. Чтобы предоставить проектам возможность решить эту проблему, в CMake 3.11 появилась возможность перевести импортируемую цель в глобальную видимость, установив свойство IMPORTED_GLOBAL цели в true. Обратите внимание, что это односторонний переход, невозможно перевести глобальную цель обратно в локальную видимость.

```
# Imported library created with local visibility.
# This could be in an external file brought in
# by an include() call rather than in the same
# file as the lines further below.
add_library(builtElsewhere STATIC IMPORTED)
set_target_properties(builtElsewhere PROPERTIES
    IMPORTED_LOCATION /path/to/libSomething.a
)

# Promote the imported target to global visibility
set_target_properties(builtElsewhere PROPERTIES
    IMPORTED_GLOBAL TRUE
)
```

Важно отметить, что импортированная цель может быть продвинута, только если она определена в той же области видимости, что и продвигаемая. Импортированная цель, определенная в родительской или дочерней области видимости, не может быть продвинута. Команда include() не вводит новую область видимости каталога, как и вызов find_package(), поэтому импортированные цели, определенные в файлах, принесенных в сборку таким образом, *могут* быть продвинуты. Фактически, это основной случай использования, для которого была создана возможность продвижения импортированных целей. Следует также отметить, что после того, как импортированная цель получила глобальную видимость, она может поддерживать создание псевдонима, ссылающегося на нее.

16.4. Рекомендуемые практики

Версия 3.0 CMake привнесла значительные изменения в рекомендуемый способ управления зависимостями и требованиями между целями. Вместо того чтобы указывать большинство вещей через переменные, которыми проект должен был управлять вручную, или через команды уровня каталога, которые применялись ко всем

целям в каталоге и ниже без особой дискриминации, каждая цель получила возможность нести всю необходимую информацию в своих собственных свойствах. Смещение акцента в сторону модели, ориентированной на цели, также привело к появлению семейства псевдоцелевых типов, которые позволяют более гибко и точно выражать межцелевые отношения. Разработчикам следует познакомиться с библиотеками интерфейсов, поскольку они открывают целый ряд методов для фиксации и выражения отношений без необходимости создавать или ссылаться на физический файл. Они могут быть полезны для представления деталей библиотек, содержащих только заголовки, коллекций ресурсов и многих других сценариев, и им следует отдать предпочтение перед попытками достичь того же результата только с помощью переменных или команд уровня каталога.

Импортированные цели часто встречаются, когда проекты начинают использовать пакеты, созданные извне, или ссылаются на инструменты из файловой системы, найденные с помощью модулей Find. Разработчикам должно быть удобно использовать импортированные цели, но понимание всех тонкостей их определения обычно не требуется, если только они не пишут модули Find или не создают вручную конфигурационные файлы для пакета. Некоторые конкретные случаи обсуждаются в главе 25, *Установка*, когда разработчики могут столкнуться с определенными ограничениями импортируемых целей, но такие сценарии не так уж часто встречаются.

В ряде старых модулей CMake для ссылки на импортируемые объекты использовались только переменные. Начиная с CMake 3.0, эти модули постепенно обновляются, чтобы также предоставлять импортируемые цели, где это необходимо. В ситуациях, когда проекту необходимо обратиться к внешнему инструменту или библиотеке, предпочтительнее делать это через импортированную цель, если таковая имеется. Они обычно лучше абстрагируются от таких вещей, как различия платформ, выбор инструмента в зависимости от опций и т.д., но, что более важно, требования к использованию надежно обрабатываются CMake. Если есть выбор между использованием импортированной библиотеки и переменной для ссылки на одну и ту же вещь, предпочитайте использовать импортированную библиотеку, когда это возможно.

Предпочтительнее определять статические библиотеки, чем объектные. Статические библиотеки проще, имеют более полную и надежную поддержку из ранних версий CMake, и они хорошо понятны большинству разработчиков. Объектные библиотеки имеют свое применение, но они также менее гибкие, чем статические библиотеки. В частности, объектные библиотеки не могут быть связаны (до CMake 3.12) и поэтому не поддерживают транзитивные зависимости. Это вынуждает проекты самостоятельно вручную применять такие зависимости, что увеличивает возможность ошибок и упущений. Это также снижает инкапсуляцию, которую обычно обеспечивает целевая библиотека. Даже само название может вызвать некоторую путаницу среди разработчиков, поскольку объектная библиотека - это не настоящая библиотека, а скорее просто набор некомбинированных объектных файлов, но разработчики иногда ожидают, что она будет вести себя как настоящая библиотека. Изменения в версии 3.12 стирают это различие, но оставшиеся различия все еще оставляют место для неожиданных результатов.

Когда дело доходит до именования целей, не используйте слишком общие имена целей. Глобально видимые имена целей должны быть уникальными, а при использовании в более крупной иерархической структуре имена могут конфликтовать с целями из других проектов. Кроме того, рассмотрите возможность добавления псевдонима namespace::... target для каждой цели, которая не является приватной для проекта (т.е. для каждой цели, которая в конечном итоге может быть установлена или упакована). Это позволит потребляющим проектам ссылаться на имя цели в пространстве имен вместо реального имени цели, что позволит потребляющим проектам относительно легко переключаться между самостоятельной сборкой дочернего проекта и использованием предварительно собранного установленного проекта. Хотя поначалу это может показаться лишней работой, не дающей особого выигрыша, в сообществе CMake это становится ожидаемой стандартной практикой, особенно для тех проектов, сборка которых занимает нетривиальное количество времени. Этот шаблон обсуждается далее в разделе 25.3, "Установка экспорта".

Неизбежно, в какой-то момент может возникнуть желание переименовать или рефакторить библиотеку, но могут существовать внешние проекты, которые ожидают, что существующие цели библиотеки будут доступны для ссылки. В таких ситуациях используйте псевдоним цели, чтобы обеспечить старое имя для переименованной цели, чтобы внешние проекты могли продолжать сборку и обновление в удобное для них время. При разделении библиотеки определите интерфейсную библиотеку со старым именем цели, и пусть она определяет зависимости ссылок на новые разделенные библиотеки. Например:

```
# Old library previously defined like this:
add_library(deepCompute SHARED ...)
```

```
# Now the library has been split in two, so define
# an interface library with the old name to effectively
# forward on the link dependency to the new libraries
add_library(computeAlgoA SHARED ...)
add_library(computeAlgoB SHARED ...)

add_library(deepCompute INTERFACE)
target_link_libraries(deepCompute INTERFACE
    computeAlgoA
    computeAlgoB
)
```

Глава 17. Пользовательские задачи

Ни один инструмент сборки не может надеяться реализовать все функции, которые могут понадобиться в том или ином проекте. В какой-то момент разработчикам потребуется выполнить задачу, которая выходит за рамки непосредственно поддерживаемой функциональности. Например, может потребоваться запуск специального инструмента для создания исходных файлов или для постобработки цели после ее сборки. Файлы могут нуждаться в копировании, проверке или вычислении хэш-значения. Артефакты сборки может потребоваться архивировать или связаться со службой уведомлений. Эти и другие задачи не всегда укладываются в предсказуемую схему, которая позволяет легко предоставить их в качестве общей возможности системы сборки.

CMake поддерживает такие задачи с помощью пользовательских команд и пользовательских целей. Они позволяют выполнять любую команду или набор команд во время сборки для выполнения любых произвольных задач, требуемых проектом. CMake также поддерживает выполнение задач во время конфигурирования, что позволяет использовать различные техники, которые полагаются на выполнение задач до этапа сборки или даже до обработки последующих частей текущего файла CMakeLists.txt.

17.1. Пользовательские цели

Библиотечные и исполняемые цели - не единственные виды целей, поддерживаемые CMake. Проекты также могут определять свои собственные пользовательские цели, которые выполняют произвольные задачи, определенные как последовательность команд, выполняемых во время сборки. Эти пользовательские цели определяются с помощью команды add_custom_target():

```
add_custom_target(targetName [ALL]
                  [command1 [args1...]]
                  [COMMAND command2 [args2...]]
                  [DEPENDS depends1...]
                  [BYPRODUCTS [files...]]
                  [WORKING_DIRECTORY dir]
                  [COMMENT comment]
                  [VERBATIM]
                  [USES_TERMINAL]
                  [SOURCES source1 [source2...]]
)
```

Новая цель с указанным именем targetName будет доступна для сборки. Опция ALL делает *всю* цель зависимой от этой новой пользовательской цели (различные генераторы называют цель *all* немного по-разному, но обычно это что-то вроде all, ALL или что-то подобное). Если опция ALL не указана, то цель будет построена только в том случае, если она явно запрошена или если строится какая-то другая цель, зависящая от нее. Пользовательская цель всегда считается устаревшей, поэтому обновление любой цели, которая зависит от нее, приведет к выполнению команд.

Когда пользовательская цель построена, указанные команды будут выполнены в указанном порядке, причем каждая команда может иметь любое количество аргументов. Для улучшения читабельности аргументы могут быть разделены на несколько строк. Первая команда не обязательно должна иметь ключевое слово COMMAND перед собой, но для ясности рекомендуется всегда включать ключевое слово COMMAND даже для первой команды. Это особенно актуально при указании нескольких команд, поскольку это заставляет каждую команду использовать последовательную форму.

Команды могут быть определены для выполнения любых действий, которые могут быть выполнены на хост-платформе. Типичные команды включают запуск скрипта или исполняемого файла, предоставляемого системой, но они также могут запускать исполняемые цели, созданные в процессе сборки. Если в качестве команды для выполнения указано имя другой исполняемой цели, CMake автоматически подставит местоположение собранного исполняемого файла этой другой цели. Это работает независимо от используемой платформы или генератора CMake, освобождая тем самым проект от необходимости разбираться с различными различиями между платформами и генераторами, которые приводят к различным структурам выходных каталогов, именам файлов и т. д. Если в качестве аргумента одной из команд необходимо использовать другую цель, CMake не будет автоматически выполнять такую замену, но получить эквивалентную замену с помощью выражения генератора TARGET_FILE очень просто. Проекты должны использовать эти возможности, чтобы позволить CMake предоставлять местоположение целей, а не прописывать пути вручную, поскольку это позволяет проекту обеспечить надежную поддержку всех платформ и типов генераторов с минимальными усилиями. В следующем примере показано, как определить пользовательскую цель, которая использует две другие цели в качестве части команды и списка аргументов:

```
add_executable(hasher hasher.cpp)
add_library(myLib api.cpp)

add_custom_target(createHash
    COMMAND hasher $<TARGET_FILE:myLib>
)
```

Когда цель используется в качестве команды для выполнения, CMake автоматически создает зависимость от этой исполняемой цели, чтобы обеспечить ее сборку перед пользовательской целью. Аналогично, если цель упоминается в выражении генератора в любом месте команды или ее аргументов, зависимость автоматически создается и для этой цели. Если необходимо указать зависимость от какой-либо другой цели, для определения этой зависимости можно использовать команду add_dependencies(). Если зависимость существует не от цели, а от *файла,* ключевое слово DEPENDS может быть использовано для указания этой зависимости непосредственно в вызове add_custom_target(). Обратите внимание, что DEPENDS не следует использовать для целевых зависимостей, только для файловых. Ключевое слово DEPENDS особенно полезно, когда указанный файл генерируется другой пользовательской командой (см. раздел 17.3, "Команды, генерирующие файлы", далее), где CMake установит необходимые зависимости, чтобы другие пользовательские команды выполнялись раньше команд этой пользовательской цели. Всегда используйте абсолютный путь для DEPENDS, так как относительные пути могут дать неожиданные результаты из-за унаследованной функции, позволяющей сопоставлять пути в нескольких местах.

Если указано несколько команд, каждая из них будет выполняться в указанном порядке. Однако проект не должен предполагать какого-либо определенного поведения оболочки, поскольку каждая команда может выполняться в своей собственной оболочке или вообще без какого-либо окружения оболочки. Пользовательские команды должны быть определены так, как будто они выполняются изолированно и без каких-либо возможностей оболочки, таких как перенаправление, подстановка переменных и т.д., с соблюдением только порядка команд. Хотя некоторые из этих функций могут работать на некоторых платформах, они не поддерживаются повсеместно. Кроме того, поскольку не гарантируется определенное поведение оболочки, экранирование в именах исполняемых файлов или их аргументов может обрабатываться по-разному на разных платформах. Чтобы уменьшить эти различия, можно использовать опцию VERBATIM, чтобы гарантировать, что единственное экранирование выполняется самим CMake при разборе файла CMakeLists.txt. Никакого дополнительного экранирования платформа не выполняет, поэтому разработчик может быть уверен в том, как команда в конечном итоге будет сконструирована для выполнения. Если есть вероятность того, что экранирование будет уместным, рекомендуется использовать ключевое слово VERBATIM.

По умолчанию каталог, в котором выполняются команды, является текущим двоичным каталогом. Это можно изменить с помощью параметра WORKING_DIRECTORY, который может быть абсолютным путем или относительным путем, причем последний является относительным к текущему двоичному каталогу. Это означает, что использование
${CMAKE_CURRENT_BINARY_DIR} как часть рабочего каталога не должно быть необходимым, поскольку относительный путь уже подразумевает это.

Параметр BYPRODUCTS можно использовать для перечисления других файлов, которые создаются в процессе выполнения команды (команд). Если используется генератор Ninja, этот параметр необходим, если другая цель зависит от любого из файлов, созданных как побочный продукт выполнения этого набора пользовательских команд. Файлы, перечисленные как BYPRODUCTS, помечаются как GENERATED (для всех типов генераторов, не только Ninja), что гарантирует, что инструмент сборки знает, как правильно обрабатывать данные зависимости, связанные с файлами побочных продуктов. Для случаев, когда пользовательская цель генерирует файлы как побочный продукт, подумайте, не будет ли add_custom_command() более подходящим способом определения команд и того, что они выводят (см. раздел 17.3, "Команды, генерирующие файлы").

Если команды не выдают никаких результатов на консоль, иногда бывает полезно указать короткое сообщение с помощью опции COMMENT. Указанное сообщение записывается в журнал непосредственно перед выполнением команд, поэтому, если команды по какой-то причине не выполняются, комментарий может быть полезным маркером, указывающим на место сбоя сборки. Заметим, однако, что в некоторых генераторах комментарий не будет показан, поэтому этот механизм нельзя считать надежным, но он все же может быть полезен для тех генераторов, которые его поддерживают. Универсально поддерживаемая альтернатива представлена ниже в разделе 17.5, "Платформонезависимые команды".

USES_TERMINAL - это еще один параметр, связанный с консолью, который указывает CMake предоставить команде прямой доступ к терминалу, если это возможно. При использовании генератора Ninja это приводит к тому, что команда помещается в пул консоли. Это может привести к улучшению буферизации вывода в некоторых ситуациях, например, помочь средам IDE более своевременно перехватывать и представлять вывод сборки. Это также может быть полезно, если интерактивный ввод требуется для сборок без IDE. Опция USES_TERMINAL поддерживается в CMake 3.2 и более поздних версиях.

Опция SOURCES позволяет перечислить произвольные файлы, которые затем будут связаны с пользовательской целью. Эти файлы могут использоваться командами, или это могут быть просто дополнительные файлы, слабо связанные с целью, например, документация и т.д. Перечисление файлов в SOURCES никак не влияет на сборку или отношения зависимости, оно служит исключительно для ассоциации этих файлов с целью, чтобы проекты IDE могли показать их в соответствующем контексте. Этой возможностью иногда пользуются, определяя фиктивную пользовательскую цель и перечисляя исходники без команд только для того, чтобы они отображались в проектах IDE. Хотя это работает, у этого есть недостаток - создание цели сборки, не имеющей реального значения. Многие проекты считают это приемлемым компромиссом, в то время как некоторые разработчики считают это нежелательным или даже анти-паттерном.

17.2. Добавление шагов сборки к существующей цели

Иногда пользовательские команды не требуют определения новой цели, они могут указывать дополнительные шаги, которые необходимо выполнить при построении существующей цели. В этом случае следует использовать add_custom_command() с ключевым словом TARGET следующим образом:

```
add_custom_command(TARGET targetName buildStage
                   COMMAND command1 [args1...]
                   [COMMAND command2 [args2...]]
                   [WORKING_DIRECTORY dir]
                   [BYPRODUCTS files...]
                   [COMMENT comment]
                   [VERBATIM]
                   [USES_TERMINAL]
)
```

Большинство опций очень похожи на опции add_custom_target(), но вместо определения новой цели, приведенная выше форма присоединяет команды к существующей цели. Эта существующая цель может быть исполняемой или библиотечной целью, или даже пользовательской целью (с некоторыми ограничениями). Команды будут выполнены как часть сборки targetName, причем аргумент buildStage должен быть одним из следующих:

PRE_BUILD

Команды должны выполняться перед любыми другими правилами для указанной цели. Имейте в виду, что только генератор Visual Studio поддерживает эту опцию и только для Visual Studio 7 или более поздней версии. Все остальные генераторы CMake будут рассматривать эту опцию как PRE_LINK. Учитывая ограниченную поддержку этой опции, проекты должны стремиться к структуре, которая не требует пользовательской команды PRE_BUILD.

PRE_LINK

Команды будут выполняться после компиляции исходных текстов, но до их компоновки. Для статических библиотечных целей команды будут выполняться до инструмента архиватора библиотек. Для пользовательских целей PRE_LINK не поддерживается.

POST_BUILD

Команды будут выполняться после всех остальных правил для указанной цели. Все типы целей и генераторы поддерживают эту опцию, что делает ее предпочтительным этапом сборки, когда есть выбор.

Задачи POST_BUILD встречаются относительно часто, но PRE_LINK и PRE_BUILD нужны редко, поскольку их обычно можно избежать, используя вместо этого OUTPUT форму add_custom_command() (см. следующий раздел).

Можно выполнить несколько вызовов add_custom_command(), чтобы добавить несколько наборов пользовательских команд к определенной цели. Это может быть полезно, например, для того, чтобы одни команды выполнялись из одного рабочего каталога, а другие - из другого.

```
add_executable(myExe main.cpp)

add_custom_command(TARGET myExe POST_BUILD
    COMMAND script1 $<TARGET_FILE:myExe>
)

# Additional command which will run after the above from a different directory
add_custom_command(TARGET myExe POST_BUILD
    COMMAND writeHash $<TARGET_FILE:myExe>
    BYPRODUCTS ${CMAKE_BINARY_DIR}/verify/myExe.md5
    WORKING_DIRECTORY ${CMAKE_BINARY_DIR}/verify
)
```

17.3. Команды, генерирующие файлы

Определение команд как дополнительных шагов сборки для цели охватывает многие распространенные случаи использования. Однако иногда проекту требуется создать один или несколько файлов, выполнив команду или серию команд, и создание такого файла не относится к какой-либо существующей цели. В этом случае можно использовать OUTPUT-форму add_custom_command(). Она реализует все те же опции, что и форма TARGET, а также некоторые дополнительные опции, связанные с обработкой зависимостей и добавлением к предыдущему набору команд OUTPUT.

```
add_custom_command(OUTPUT output1 [output2...]
                   COMMAND command1 [args1...]
                   [COMMAND command2 [args2...]]
                   [WORKING_DIRECTORY dir]
                   [BYPRODUCTS files...]
                   [COMMENT comment]
                   [VERBATIM]
                   [USES_TERMINAL]
                   [APPEND]
                   [DEPENDS [depends1...]
                   [MAIN_DEPENDENCY depend]
                   [IMPLICIT_DEPENDS <lang1> depend1
                                     [<lang2> depend2...]]
                   [DEPFILE depfile]
)
```

Вместо указания цели и стадии сборки до/после сборки, эта форма требует указания одного или нескольких имен выходных файлов после ключевого слова OUTPUT. Затем CMake интерпретирует команды как рецепт для создания названных выходных файлов. Если выходные файлы указаны без пути или с относительным путем, то они будут относительными по отношению к текущему двоичному каталогу.

Сама по себе эта форма не приведет к сборке выходных файлов, поскольку не определена цель. Однако если какая-то другая цель, определенная в той же области каталогов, зависит от любого из выходных файлов, CMake автоматически создаст отношения зависимости, которые обеспечат генерацию выходных файлов перед целью, которой они нужны. Эта цель может быть обычным исполняемым файлом, библиотекой или даже пользовательской целью. На самом деле, довольно часто пользовательская цель определяется просто для того, чтобы разработчик мог вызвать пользовательскую команду. Следующая вариация на тему хэширования из предыдущего раздела демонстрирует эту технику:

```
add_executable(myExe main.cpp)

# Output file with relative path, generated in the build directory
add_custom_command(OUTPUT myExe.md5
    COMMAND writeHash $<TARGET_FILE:myExe>
)

# Absolute path needed for DEPENDS, otherwise relative to source directory
add_custom_target(computeHash
    DEPENDS ${CMAKE_CURRENT_BINARY_DIR}/myExe.md5
)
```

При таком определении сборка цели *myExe* не приведет к запуску шага хэширования, в отличие от предыдущего примера, в котором команда хэширования была добавлена как шаг POST_BUILD цели *myExe*. Вместо этого хэширование будет выполняться только в том случае, если разработчик явно запросит его в качестве цели сборки. Это позволяет определять дополнительные шаги и вызывать их по мере необходимости, а не запускать всегда, что может быть весьма полезно, если дополнительные шаги отнимают много времени или не всегда уместны.

Конечно, add_custom_command() можно также использовать для генерации файлов, потребляемых существующими целями, например, для генерации исходных файлов. В следующем примере исполняемый файл, собранный проектом, используется для генерации исходного файла, который затем компилируется как часть другого исполняемого файла.

```
add_executable(generator generator.cpp)

add_custom_command(OUTPUT onTheFly.cpp
    COMMAND generator
)

add_executable(myExe ${CMAKE_CURRENT_BINARY_DIR}/onTheFly.cpp)
```

CMake автоматически распознает, что myExe нужен исходный файл, сгенерированный пользовательской командой, которая, в свою очередь, требует исполняемый файл генератора. Запрос на сборку цели myExe приведет к тому, что генератор и сгенерированный исходный файл будут собраны до сборки myExe. Обратите внимание, однако, что эта зависимость имеет ограничения. Рассмотрим следующий сценарий:

- Изначально файл onTheFly.cpp не существует.
- Постройте цель myExe, что приведет к следующей последовательности действий:
  - Цель генератора приведена в актуальное состояние.
  - Выполняется пользовательская команда для создания файла onTheFly.cpp.
  - Цель myExe построена.
- Теперь измените файл generator.cpp.
- Снова постройте цель myExe, что на этот раз приведет к следующей последовательности действий:
  - Цель генератора обновляется. Это приведет к перестройке исполняемого файла генератора, поскольку его исходный файл был изменен.
  - Пользовательская команда НЕ выполняется, поскольку файл onTheFly.cpp уже существует.

◦ Цель myExe НЕ перестраивается, поскольку ее исходный файл остается неизменным.

Интуитивно можно предположить, что если цель генератора перестраивается, то пользовательская команда также должна быть запущена заново. Зависимость, которую автоматически создает CMake, не обеспечивает этого, она создает более слабую зависимость, которая обеспечивает обновление генератора, но пользовательская команда выполняется только в том случае, если выходной файл полностью отсутствует. Чтобы заставить пользовательскую команду запускаться заново, если цель генератора перестраивается, необходимо указать явную зависимость, а не полагаться на зависимость, которую CMake создает автоматически.

Зависимости можно указать вручную с помощью опции DEPENDS. Элементы, перечисленные с помощью DEPENDS, могут быть целями CMake или файлами (сравните это с опцией DEPENDS для add_custom_target(), которая может перечислять только файлы). Если в списке указана цель, она будет обновляться каждый раз, когда потребуется обновить выходные файлы пользовательской команды. Аналогично, если перечисленный файл будет изменен, то пользовательская команда будет выполнена, если потребуется изменить какой-либо из выходных файлов пользовательской команды. Кроме того, если какой-либо из перечисленных файлов является выходным файлом другой пользовательской команды в той же области каталогов, то сначала будет выполнена другая пользовательская команда. Что касается add_custom_target(), всегда используйте абсолютный путь, если перечисляете файл для DEPENDS, чтобы избежать неоднозначного унаследованного поведения.

Хотя автоматические зависимости CMake могут показаться удобными, на практике проекту все равно обычно требуется перечислить все необходимые цели и файлы в разделе DEPENDS, чтобы убедиться, что все отношения зависимостей указаны адекватно. Раздел DEPENDS легко опустить по ошибке, поскольку при первой сборке будет запущена пользовательская команда для создания недостающих выходных файлов, и сборка будет вести себя корректно. Последующие сборки не будут повторно выполнять пользовательскую команду, пока не будет удален выходной файл, даже если будут перестроены все автоматически обнаруженные цели зависимостей. Это может быть легко пропустить, часто в сложных проектах это остается незамеченным в течение длительного времени, пока разработчик не столкнется с ситуацией и не попытается выяснить, почему что-то не пересобирается, когда это должно было произойти. Поэтому разработчикам следует ожидать, что раздел DEPENDS обычно необходим, если только пользовательская команда не требует ничего, созданного сборкой или исходными файлами проекта.

Другая распространенная ошибка - не создать зависимость от файла, который необходим пользовательской команде, но не указан в командной строке для выполнения. Такие файлы должны появиться в разделе DEPENDS, чтобы сборка считалась надежной.

Есть еще несколько опций, связанных с зависимостями, поддерживаемых функцией add_custom_command(). Опция MAIN_DEPENDENCY предназначена для определения исходного файла, который следует считать основной зависимостью пользовательской команды. В основном она имеет тот же эффект, что и DEPENDS для указанного файла, но некоторые генераторы могут применять дополнительную логику, например, определять, куда поместить пользовательскую команду в проекте IDE. Важно отметить, что если исходный файл указан как MAIN_DEPENDENCY, то пользовательская команда становится заменой того, как этот исходный файл обычно компилируется. Это может привести к неожиданным результатам. Рассмотрим следующий пример:

```
add_custom_command(OUTPUT transformed.cpp
    COMMAND transform
            ${CMAKE_CURRENT_SOURCE_DIR}/original.cpp
            transformed.cpp
    MAIN_DEPENDENCY ${CMAKE_CURRENT_SOURCE_DIR}/original.cpp
)
add_executable(original    original.cpp)
add_executable(transformed transformed.cpp)
```

Вышеописанное приведет к ошибке компоновщика для исходной цели, поскольку original.cpp не будет скомпилирован в объектный файл, поэтому объектных файлов вообще не будет (и, следовательно, не будет функции main()). Вместо этого инструмент сборки будет рассматривать original.cpp как входной файл, используемый для создания transformed.cpp. Проблему можно решить, используя DEPENDS вместо MAIN_DEPENDENCY, так как это сохранит ту же зависимость, но не приведет к замене правила компиляции по умолчанию для исходного файла original.cpp.

Две другие опции, связанные с зависимостями, IMPLICIT_DEPENDS и DEPFILE, не поддерживаются большинством генераторов проектов. IMPLICIT_DEPENDS игнорируется для всех генераторов, кроме Makefile, а использование DEPFILE приводит к ошибке, если используется не генератор Ninja. IMPLICIT_DEPENDS направляет CMake на вызов сканера C или C++ для определения зависимостей перечисленных файлов, а DEPFILE можно использовать для предоставления специфичного для Ninja файла зависимостей .d. Проекты, как правило, стараются избегать этих двух опций из-за крайне ограниченного числа генераторов проектов, которые их поддерживают.

Формы OUTPUT и TARGET также имеют немного разное поведение, когда речь идет о добавлении большего количества зависимостей или команд в один и тот же выходной файл или цель. Для формы OUTPUT должно быть указано ключевое слово APPEND, а первый указанный файл OUTPUT должен быть одинаковым для первого и последующих вызовов add_custom_command(). Только COMMAND и DEPENDS могут быть использованы для второго и последующих вызовов для одного и того же выходного файла, остальные опции, такие как MAIN_DEPENDENCY, WORKING_DIRECTORY и COMMENT, игнорируются при наличии ключевого слова APPEND. Для формы TARGET, напротив, ключевое слово APPEND не требуется для второго и последующих вызовов add_custom_command() для одной и той же цели. Опции COMMENT и WORKING_DIRECTORY также могут быть указаны для каждого вызова, и они вступят в силу для команд, добавляемых в этом вызове.

17.4. Настройка временных задач

И add_custom_target(), и add_custom_command() определяют команды, которые будут выполняться на этапе сборки. Обычно пользовательские команды выполняются именно в это время, но бывают ситуации, когда пользовательская задача должна быть выполнена на этапе конфигурирования, например:

- Выполнение внешних команд для получения информации, которая будет использоваться при конфигурировании.
- Запись или касание файлов, которые необходимо обновлять при каждом повторном запуске CMake.
- Генерация CMakeLists.txt или других файлов, которые должны быть включены или обработаны как часть текущего шага configure.

CMake предоставляет команду execute_process() для запуска подобных задач на этапе конфигурирования:

```
execute_process(COMMAND command1 [args1...]
                [COMMAND command2 [args2...]]
                [WORKING_DIRECTORY directory]
                [RESULT_VARIABLE resultVar]
                [RESULTS_VARIABLE resultsVar]
                [OUTPUT_VARIABLE outputVar]
                [ERROR_VARIABLE errorVar]
                [OUTPUT_STRIP_TRAILING_WHITESPACE]
                [ERROR_STRIP_TRAILING_WHITESPACE]
                [INPUT_FILE inFile]
                [OUTPUT_FILE outFile]
                [ERROR_FILE errorFile]
                [OUTPUT_QUIET]
                [ERROR_QUIET]
                [TIMEOUT seconds]
)
```

Подобно add_custom_command() и add_custom_target(), один или несколько разделов COMMAND определяют задачи, которые должны быть выполнены, а опция WORKING_DIRECTORY может быть использована для управления местом выполнения этих команд. Команды передаются операционной системе для выполнения как есть, без промежуточного окружения оболочки. Поэтому такие функции, как перенаправление ввода/вывода и переменные окружения, не поддерживаются. Команды выполняются немедленно.

Если задано несколько команд, они выполняются по порядку, но вместо того, чтобы быть полностью независимыми друг от друга, стандартный вывод одной команды передается на вход следующей. При отсутствии других опций вывод *последней* команды отправляется на вывод самого процесса CMake, но стандартная ошибка *каждой* команды отправляется в поток стандартных ошибок процесса CMake.

Потоки стандартного вывода и стандартной ошибки могут быть захвачены и сохранены в переменных вместо того, чтобы быть отправленными в стандартные трубы. Вывод последней команды в наборе команд можно захватить, указав имя переменной для хранения в ней с помощью опции OUTPUT_VARIABLE. Аналогично, стандартные потоки ошибок всех команд могут быть сохранены в переменной, названной опцией ERROR_VARIABLE. Передача одного и того же имени переменной обеим этим опциям приведет к объединению стандартного вывода и стандартной ошибки, как при выводе на терминал, с сохранением объединенного результата в названной переменной. Если присутствует опция OUTPUT_STRIP_TRAILING_WHITESPACE, то все пробельные символы в конце будут опущены из содержимого, хранящегося в выходной переменной, а опция ERROR_STRIP_TRAILING_WHITESPACE делает то же самое для содержимого, хранящегося в переменной ошибок. При использовании содержимого выходных переменных или переменных ошибок для сравнения строк часто возникает проблема, связанная с неучетом пробельных символов, поэтому их удаление часто желательно.

Вместо того чтобы записывать потоки вывода и ошибок в переменную, их можно отправить в файлы.

Опции OUTPUT_FILE и ERROR_FILE могут быть использованы для указания имен файлов, в которые будут отправляться потоки. Как и в случае с опциями переменной-фокуса, указание одного и того же имени файла для обоих приводит к объединению потоков. Кроме того, с помощью опции INPUT_FILE можно указать файл для входного потока первой команды. Обратите внимание, однако, что опции OUTPUT_STRIP_TRAILING_WHITESPACE и ERROR_STRIP_TRAILING_WHITESPACE не влияют на содержимое, отправляемое в файлы.

Один и тот же поток не может быть одновременно захвачен в переменную и отправлен в файл. Однако можно отправлять разные потоки в разные места, например, выходной поток в переменную, а поток ошибок в файл или наоборот. Также можно молча отбросить содержимое потока с помощью опций OUTPUT_QUIET и ERROR_QUIET. Эти опции могут быть полезны, если интерес представляет только успех или неудача команды.

Успех или неудачу набора команд можно зафиксировать с помощью опции RESULT_VARIABLE. Результат выполнения команд будет сохранен в именованной переменной в виде либо целочисленного кода возврата последней команды, либо строки, содержащей какое-либо сообщение об ошибке. Команда if() удобно обрабатывает непустые строки ошибок и целые значения, отличные от 0, как boolean true (если только проекту не повезло иметь строку ошибок, удовлетворяющую одному из особых случаев, см. раздел 6.1.1, "Основные выражения"). Поэтому проверка успешности вызова execute_process() обычно относительно проста:

```
execute_process(COMMAND runSomeScript
                RESULT_VARIABLE result)
if(result)
    message(FATAL_ERROR "runSomeScript failed: ${result})
endif()
```

Начиная с CMake 3.10, если требуется результат каждой отдельной команды, а не только последней, вместо него можно использовать опцию RESULTS_VARIABLE. Эта опция сохраняет результат каждой команды в переменной с именем resultsVar в виде списка.

Параметр TIMEOUT можно использовать для обработки команд, которые могут выполняться дольше, чем ожидается, или которые, возможно, никогда не завершатся. Это гарантирует, что шаг configure не будет блокироваться бесконечно, и позволяет рассматривать неожиданно длинный шаг configure как ошибку. Обратите внимание, однако, что опция TIMEOUT сама по себе не заставит CMake остановиться и сообщить об ошибке. Результат команды все равно должен быть зафиксирован с помощью RESULT_VARIABLE, а затем эта переменная должна быть проверена, как в предыдущем примере. Если команда выполняется дольше порога таймаута, в переменной result будет содержаться строка ошибки, указывающая на то, что команда была завершена из-за таймаута, поэтому рекомендуется печатать переменную result.

Когда CMake выполняет команды, дочерний процесс в основном наследует то же окружение, что и главный процесс. Важным исключением является то, что при первом запуске CMake в проекте переменные окружения CC и CXX дочернего процесса явно устанавливаются на компиляторы C и C++, используемые основной сборкой (если в основном проекте включены языки C и C++). При последующих запусках CMake переменные окружения CC и CXX *не* заменяются таким образом, что может привести к неожиданным результатам, если команды выполняют действия, которые зависят от того, что CC и/или CXX имеют одинаковые значения при каждом вызове execute_process(). Это недокументированное поведение существовало с ранних версий CMake, даже в ныне устаревшей команде exec_program(), которую заменила execute_process(). Она была добавлена для того, чтобы дочерние процессы могли конфигурировать и запускать субсборки с теми же компиляторами, что и основной проект. Однако в некоторых случаях дочерний процесс может не захотеть сохранять компилятор, например, когда основная сборка кросс-компилируется, но дочерний процесс должен использовать компиляторы хоста по умолчанию. В таких случаях проекты могут установить переменную CMAKE_GENERATOR_NO_COMPILER_ENV в булево значение true, и тогда CMake не будет устанавливать CC и CXX для любого вызова execute_process(), даже для начального вызова.

17.5. Команды, независимые от платформы

Команды add_custom_command(), add_custom_target() и execute_process() предоставляют проектам большую свободу. Любая задача, которая еще не поддерживается CMake напрямую, может быть реализована с помощью команд, предоставляемых операционной системой хоста. Эти пользовательские команды по своей природе специфичны для конкретной платформы, что противоречит одной из основных причин, по которой многие проекты используют CMake, т.е. абстрагироваться от различий платформ или, по крайней мере, поддерживать ряд платформ с минимальными усилиями.

Значительная часть пользовательских задач связана с манипуляциями с файловой системой. Создание, удаление, переименование или перемещение файлов и каталогов составляют основную часть этих задач, но команды для их выполнения различаются в разных операционных системах. В результате проекты часто заканчиваются использованием условий if-else для определения версий одной и той же команды для разных платформ, или, что еще хуже, они пытаются реализовать команды только для некоторых платформ. Многие разработчики не знают, что сама команда cmake предоставляет командный режим, который абстрагирует многие из этих задач, специфичных для конкретной платформы:

```
cmake -E cmd [args...]
```

Полный набор поддерживаемых команд можно перечислить с помощью команды cmake -E help, но некоторые из них являются наиболее часто используемыми:

- сравнение_файлов
- копия
- копия_каталога
- copy_if_different
- echo
- env
- make_directory
- md5sum
- удалить
- удалить_каталог
- переименовать
- tar
- время
- прикоснуться к

Рассмотрим пример пользовательской задачи по удалению определенного каталога и всего его содержимого:

```
set(discardDir "${CMAKE_CURRENT_BINARY_DIR}/private")

# Naive platform specific implementation (not robust)
if(WIN32)
    add_custom_target(myCleanup
        COMMAND rmdir /S /Q "${discardDir}"
    )
elseif(UNIX)
    add_custom_target(myCleanup
        COMMAND rm -rf "${discardDir}"
    )
else()
    message(FATAL_ERROR "Unsupported platform")
endif()

# Platform independent equivalent
add_custom_target(myCleanup
    COMMAND "${CMAKE_COMMAND}" -E remove_directory "${discardDir}"
)
```

Реализация для конкретной платформы показывает, как проекты обычно пытаются реализовать подобный сценарий, но условия if-else проверяют *целевую* платформу, а не платформу *хоста*. В сценарии кросс-компиляции это может привести к тому, что будет использована команда не той платформы. Платформонезависимая версия, однако, не имеет такого недостатка. Она всегда выбирает правильную команду для хост-платформы.

В примере также показано, как правильно вызывать команду cmake. Переменная CMAKE_COMMAND заполняется CMake и содержит полный путь к исполняемому файлу cmake, используемому в основной сборке. Использование CMAKE_COMMAND таким образом гарантирует, что та же версия CMake используется и для пользовательской команды. Исполняемый файл cmake не обязательно должен быть в текущем PATH, и если установлено несколько версий CMake, всегда будет использоваться правильная версия, независимо от того, какая из них могла бы быть выбрана на основе PATH пользователя. Это также гарантирует, что на этапе сборки используется та же версия CMake, что и на этапе конфигурирования, даже если переменная окружения PATH пользователя изменится.

Ранее в этой главе было отмечено, что опция COMMENT для add_custom_target() и add_custom_command() не всегда надежна. Вместо использования COMMENT проекты могут использовать команду -E echo, чтобы вставить комментарии в любом месте последовательности пользовательских команд:

```
set(discardDir "${CMAKE_CURRENT_BINARY_DIR}/private")
add_custom_target(myCleanup
    COMMAND ${CMAKE_COMMAND} -E echo "Removing ${discardDir}"
    COMMAND ${CMAKE_COMMAND} -E remove_directory "${discardDir}"
    COMMAND ${CMAKE_COMMAND} -E echo "Recreating ${discardDir}"
    COMMAND ${CMAKE_COMMAND} -E make_directory "${discardDir}"
)
```

Командный режим CMake - это очень полезный способ выполнения ряда общих задач независимо от платформы. Однако иногда требуется более сложная логика, и такие пользовательские задачи часто реализуются с помощью сценариев оболочки, специфичных для конкретной платформы. Альтернативой является использование самого CMake в качестве механизма создания сценариев, предоставляющего

независимый от платформы язык, на котором можно выразить произвольную логику. Опция -P в команде cmake переводит CMake в режим обработки сценариев:

```
cmake [options] -P filename
```

Аргумент filename - это имя файла сценария CMake для выполнения. Поддерживается обычный синтаксис CMakeLists.txt, но нет шага configure или generate, и файл CMakeCache.txt не обновляется. Файл сценария по сути обрабатывается как набор команд, а не как проект, поэтому любые команды, относящиеся к целям сборки или функциям уровня проекта, не поддерживаются. Тем не менее, режим сценариев позволяет реализовать сложную логику и имеет то преимущество, что не требует установки дополнительного интерпретатора оболочки.

Хотя режим сценариев не поддерживает опции командной строки, как обычные оболочки или командные интерпретаторы, он поддерживает передачу переменных с опциями -D, как и обычные вызовы cmake. Поскольку в режиме сценариев не обновляется файл CMakeCache.txt, опции -D можно использовать свободно, не влияя на кэш основной сборки. Такие опции должны быть размещены перед -P.

```
cmake -DOPTION_A=1 -DOPTION_B=foo -P myCustomScript.cmake
```

17.6. Объединение различных подходов

Следующий пример демонстрирует многие из возможностей, представленных в этой главе. В частности, он показывает, как различные способы задания пользовательских задач могут быть использованы вместе для выполнения нетривиальных задач без необходимости прибегать к командам или функциональности, специфичным для платформы.

CMakeLists.txt

```
cmake_minimum_required(VERSION 3.0)
project(Example)

# Define an executable which generates various files in a
# directory passed as a command line argument
add_program(generateFiles generateFiles.cpp)

# Create a custom target which invokes the above executable
# after creating an empty output directory for it to populate,
# then invoke a script to archive that directory's contents
# and print the MD5 checksum of that archive
set(outDir "foo")
add_custom_target(archiver
    COMMAND ${CMAKE_COMMAND} -E echo "Archiving generated files"
    COMMAND ${CMAKE_COMMAND} -E remove_directory "${outDir}"
    COMMAND ${CMAKE_COMMAND} -E make_directory "${outDir}"
    COMMAND generateFiles "${outDir}"
    COMMAND ${CMAKE_COMMAND} "-DTAR_DIR=${outDir}"
            -P "${CMAKE_CURRENT_SOURCE_DIR}/archiver.cmake"
)
```

archiver.cmake

```
cmake_minimum_required(VERSION 3.0)

if(NOT TAR_DIR)
    message(FATAL_ERROR "TAR_DIR must be set")
endif()

# Create an archive of the directory
set(archive archive.tar)
execute_process(COMMAND ${CMAKE_COMMAND} -E tar cf ${archive} "${TAR_DIR}"
    RESULT_VARIABLE result
)
if(result)
    message(FATAL_ERROR "Archiving ${TAR_DIR} failed: ${result}")
endif()

# Compute MD5 checksum of the archive
execute_process(COMMAND ${CMAKE_COMMAND} -E md5sum ${archive}
    OUTPUT_VARIABLE md5output
    RESULT_VARIABLE result
)
if(result)
    message(FATAL_ERROR "Unable to compute md5 of archive: ${result}")
endif()

# Extract just the checksum from the output
string(REGEX MATCH "^ *[^ ]*" md5sum "${md5output}")
message("Archive MD5 checksum: ${md5sum}")
```

17.7. Рекомендуемые практики

Когда необходимо выполнить пользовательские задачи, предпочтительнее, чтобы они выполнялись на этапе сборки, а не на этапе конфигурирования. Быстрый этап configure важен, поскольку он может быть вызван автоматически при изменении некоторых файлов (например, любого файла CMakeLists.txt в проекте, любого файла, включенного в файл CMakeLists.txt или любого файла, указанного в качестве источника команды configure_file(), о чем будет рассказано в следующей главе). По этой причине вместо execute_process() лучше использовать add_custom_target() или add_custom_command(), если есть выбор.

Относительно часто можно встретить команды, специфичные для конкретной платформы, используемые в add_custom_command(), add_custom_target() и execute_process(). Однако довольно часто такие команды можно выразить в независимом от платформы виде с помощью командного режима CMake (-E). Там, где это возможно, следует предпочесть использование независимых от платформы команд. Кроме того, CMake можно использовать как платформонезависимый язык сценариев, обрабатывая файл как последовательность команд CMake при вызове с опцией -P. Использование скриптов CMake вместо специфической для платформы оболочки или отдельно устанавливаемого скриптового механизма может снизить сложность проекта и уменьшить количество дополнительных зависимостей, необходимых для его сборки. В частности, подумайте, будет ли режим сценариев CMake лучшим выбором, чем использование сценария оболочки Unix или пакетного файла Windows, или даже сценария для таких языков, как Python, Perl и т.д., которые могут быть доступны по умолчанию не на всех платформах. В следующей главе показано, как работать с файлами непосредственно с помощью CMake, не прибегая к таким инструментам и методам.

При внедрении пользовательских задач старайтесь избегать тех функций, которые не имеют универсальной поддержки на всех платформах.

- Предпочтительнее использовать командный режим -E echo вместо ключевого слова COMMENT в add_custom_command() и add_custom_target().
- Старайтесь избегать использования PRE_BUILD с TARGET формой add_custom_command().
- Подумайте, стоит ли использование опций IMPLICIT_DEPENDS или DEPFILE с add_custom_command() специфичного для генератора поведения.
- Избегайте перечисления исходного файла как MAIN_DEPENDENCY в add_custom_command(), если только не предполагается заменить правило сборки по умолчанию для этого исходного файла.

Уделите особое внимание зависимостям для входов и выходов пользовательских задач. Убедитесь, что *все* файлы, созданные командой add_custom_command(), перечислены как OUTPUT-файлы. При перечислении целей сборки в качестве команды или аргументов в вызове add_custom_command() или add_custom_target(), предпочитайте явно перечислять их как элементы DEPENDS, а не полагаться на автоматическую обработку зависимостей целей в CMake. Более слабые автоматические зависимости могут не обеспечивать все связи, которые интуитивно ожидают разработчики. Если вы указываете файл в DEPENDS для add_custom_target() или add_custom_command(), всегда используйте абсолютный путь, чтобы избежать нестабильного поведения сопоставления унаследованных путей.

При вызове execute_process() в большинстве случаев успешность выполнения команды следует проверять путем захвата результата с помощью RESULT_VARIABLE и проверки его с помощью команды if(). Это относится и к случаям, когда используется опция TIMEOUT, поскольку TIMEOUT сам по себе не генерирует ошибку, а только гарантирует, что команда не будет выполняться дольше заданного периода таймаута.

Глава 18. Работа с файлами

Многие проекты нуждаются в манипуляциях с файлами и каталогами в процессе сборки. Хотя такие манипуляции варьируются от тривиальных до довольно сложных, наиболее распространенные задачи включают:

- Построение путей или извлечение компонентов пути.
- Получение списка файлов из каталога.
- Копирование файлов.
- Генерация файла из содержимого строки.
- Генерация файла из содержимого другого файла.
- Чтение содержимого файла.
- Вычисление контрольной суммы или хэша файла.

CMake предоставляет множество функций, связанных с работой с файлами и каталогами. В некоторых случаях для достижения одной и той же цели может быть несколько способов, поэтому полезно знать о различных вариантах и понимать, как их эффективно использовать. Некоторые из этих функций часто используются неправильно, в том числе из-за того, что они широко распространены в интерактивных руководствах и примерах, что приводит к убеждению, что это правильный путь. Некоторые из наиболее проблемных антипаттернов обсуждаются в этой главе.

Большая часть функциональности CMake, связанной с файлами, обеспечивается командой file(), а некоторые другие команды предлагают альтернативы, лучше подходящие для определенных ситуаций, или предоставляют соответствующие вспомогательные возможности. Командный режим CMake, который был представлен в предыдущей главе, также предоставляет множество функций, связанных с файлами, которые во многом пересекаются с функциями команды file(), но он охватывает дополнительный набор сценариев для file(), а не является альтернативой в большинстве случаев.

18.1. Манипулирование путями

Одной из самых основных частей работы с файлами является манипуляция именами и путями к файлам. В проектах часто требуется извлечь имена файлов, суффиксы файлов и т.д. из полных путей или преобразовать абсолютные и относительные пути. Основным методом для выполнения таких операций является команда get_filename_component(), которая имеет три различных формы. Первая форма позволяет извлекать различные части пути или имени файла:

```
get_filename_component(outVar input component [CACHE])
```

Результат вызова сохраняется в переменной с именем outVar. Компонент для извлечения из входных данных задается компонентом, который должен быть одним из следующих:

DIRECTORY
: Извлечение части пути из входных данных без имени файла. До CMake 2.8.12 эта опция имела значение PATH, которое по-прежнему принимается как синоним DIRECTORY для сохранения совместимости со старыми версиями.

NAME

Извлечение полного имени файла, включая любое расширение. По сути, это просто отбрасывает часть ввода, связанную с каталогом.

NAME_WE

Извлечь только базовое имя файла. Это аналогично NAME, только извлекается только часть имени файла до первого ".", но не включая его.

EXT

Это дополнение к NAME_WE. Он извлекает только часть расширения имени файла, начиная с первого "." и далее.

Ключевое слово CACHE является необязательным. Если оно присутствует, то результат хранится как переменная кэша, а не как обычная переменная. Как правило, хранить результат в кэше нежелательно, поэтому ключевое слово CACHE часто не требуется.

```
set(input /some/path/foo.bar.txt)

get_filename_component(path1     ${input} DIRECTORY)    # /some/path
get_filename_component(path2     ${input} PATH)         # /some/path
get_filename_component(fullName  ${input} NAME)         # foo.bar.txt
get_filename_component(baseName  ${input} NAME_WE)      # foo
get_filename_component(extension ${input} EXT)          # .bar.txt
```

Вторая форма get_filename_component() используется для получения абсолютного пути:

```
get_filename_component(outVar input component [BASE_DIR dir] [CACHE])
```

В этой форме входные данные могут быть относительным путем или абсолютным путем. Если указан BASE_DIR, относительные пути интерпретируются как относительные к dir, а не к текущему каталогу источника (т.е. CMAKE_CURRENT_SOURCE_DIR). BASE_DIR будет проигнорирован, если входные данные уже являются абсолютным путем.

компонент определяет, как должны обрабатываться символические ссылки при вычислении пути, который будет храниться в outVar:

ABSOLUTE

Вычислить абсолютный путь к вводимым данным без разрешения символических ссылок.

REALPATH

Вычислить абсолютный путь входных данных с разрешенными символическими ссылками.

Команда file() обеспечивает обратную операцию, преобразуя абсолютный путь в относительный:

```
file(RELATIVE_PATH outVar relativeToDir input)
```

Следующий пример демонстрирует его использование:

```
set(basePath   /base)
set(fooBarPath /base/foo/bar)
set(otherPath  /other/place)

file(RELATIVE_PATH fooBar ${basePath} ${fooBarPath})
file(RELATIVE_PATH other  ${basePath} ${otherPath})

# The variables now have the following values:
#   fooBar = foo/bar
#   other  = ../other/place
```

Третья форма команды get_filename_component() удобна для извлечения частей полной командной строки:

```
get_filename_component(progVar input PROGRAM
    [PROGRAM_ARGS argVar] [CACHE])
```

В этой форме входные данные принимаются за командную строку, которая может содержать аргументы. CMake извлекает полный путь к исполняемому файлу, который будет вызван указанной командной строкой, при необходимости определяя его местоположение с помощью переменной окружения PATH, и сохраняет результат в progVar. Если указано PROGRAM_ARGS, набор аргументов командной строки также сохраняется в виде списка в переменной с именем argVar. Ключевое слово CACHE имеет то же значение, что и другие формы функции get_filename_component().

При работе с файлами в CMake в большинстве случаев проект может использовать прямые косые черты для разделителей каталогов на всех платформах, и CMake будет поступать правильно, преобразуя в родные пути по мере необходимости от имени проекта. Однако иногда проекту может потребоваться явное преобразование между CMake и родными путями, например, при работе с пользовательскими командами и необходимости передать путь скрипту, который требует родных путей. Для таких ситуаций команда file() предлагает еще две формы, которые помогают преобразовать пути между родными форматами платформы и CMake:

```
file(TO_NATIVE_PATH input outVar)
file(TO_CMAKE_PATH  input outVar)
```

Форма TO_NATIVE_PATH преобразует вводимые данные в родной путь для хост-платформы. Это сводится к обеспечению использования правильного разделителя каталогов (обратный слеш в Windows, прямой слеш везде). Форма TO_CMAKE_PATH преобразует все разделители каталогов во входных данных в прямые слэши. Именно такое представление используется CMake для путей на всех платформах. Входные данные могут также представлять собой список путей, заданных в форме, совместимой с переменной окружения платформы PATH. Все разделители двоеточий заменяются на полутоны, тем самым преобразуя входные данные, подобные PATH, в список путей CMake.

```
# Unix example
set(customPath /usr/local/bin:/usr/bin:/bin)

file(TO_CMAKE_PATH ${customPath} outVar)
# outVar = /usr/local/bin;/usr/bin;/bin
```

18.2. Копирование файлов

Необходимость скопировать файл на этапе конфигурирования или во время самой сборки возникает довольно часто. Поскольку копирование файла является привычной задачей для большинства пользователей, для новых разработчиков CMake естественно реализовать копирование файла в терминах тех же методов, которые они уже знают. К сожалению, это часто приводит к использованию специфичных для конкретной платформы команд оболочки add_custom_target() и add_custom_command(), а иногда и к проблемам с зависимостями, которые требуют от разработчиков многократного запуска CMake и/или ручной сборки целей в определенной последовательности. Почти во всех случаях CMake предлагает лучшие альтернативы таким подходам.

В этом разделе представлен ряд методов копирования файлов. Некоторые из них направлены на удовлетворение конкретных потребностей, в то время как другие являются более общими и могут быть использованы в различных ситуациях. Все представленные методы работают одинаково на всех платформах.

Одна из самых полезных команд для копирования файлов во время конфигурирования, к сожалению, имеет менее интуитивно понятное название. Команда configure_file() позволяет скопировать один файл из одного места в другое, выполняя по пути подстановку переменных CMake. Копирование выполняется немедленно, поэтому это операция времени конфигурирования. В несколько сокращенном виде команда выглядит следующим образом:

```
configure_file(source destination [COPYONLY | @ONLY] [ESCAPE_QUOTES])
```

Источник должен быть существующим файлом и может быть абсолютным или относительным путем, причем последний является относительным к текущему каталогу источника (т.е. CMAKE_CURRENT_SOURCE_DIR). Место назначения не может быть просто каталогом, в который нужно скопировать файл, это должно быть имя файла, опционально с путем, который может быть абсолютным или относительным. Если путь назначения не абсолютный, он интерпретируется как относительный к текущему двоичному каталогу (т.е. CMAKE_CURRENT_BINARY_DIR). Если какая-либо часть пути назначения не существует, CMake попытается создать недостающие каталоги в рамках вызова. Обратите внимание, что в проектах нередко встречается включение CMAKE_CURRENT_SOURCE_DIR или CMAKE_CURRENT_BINARY_DIR как части пути с источником и назначением соответственно, но это только добавляет ненужный беспорядок и этого следует избегать.

Если исходный файл будет изменен, сборка посчитает конечный файл устаревшим и автоматически запустит cmake заново. Если время конфигурирования и генерации нетривиально, а исходный файл часто изменяется, это может стать источником разочарования для разработчиков. По этой причине configure_file() лучше использовать только для файлов, которые не нужно часто изменять.

При выполнении копирования configure_file() имеет возможность подставлять переменные CMake. Без опций COPYONLY или @ONLY все в исходном файле, что похоже на использование переменной CMake (т.е. имеет вид ${someVar}), будет заменено значением этой переменной. Если переменной с таким именем не существует, будет заменена пустая строка. Таким же образом подставляются строки вида @someVar@. Ниже показан ряд примеров подстановки:

CMakeLists.txt

```
set(FOO "String with spaces")
configure_file(various.txt.in various.txt)
```

various.txt.in

```
CMake version: ${CMAKE_VERSION}
Substitution works inside quotes too: "${FOO}"
No substitution without the $ and {}: FOO
Empty ${} specifier gets removed
Escaping has no effect: \${FOO}
@-syntax also supported: @FOO@
```

various.txt

```
CMake version: 3.7.0
Substitution works inside quotes too: "String with spaces"
No substitution without the $ and {}: FOO
Empty  specifier gets removed
Escaping has no effect: \String with spaces
@-syntax also supported: String with spaces
```

Ключевое слово ESCAPE_QUOTES может быть использовано для того, чтобы перед любыми подставляемыми кавычками ставился обратный слеш.

CMakeLists.txt

```
set(BAR "Some \"quoted\" value")
configure_file(quoting.txt.in quoting.txt)
configure_file(quoting.txt.in quoting_escaped.txt ESCAPE_QUOTES)
```

quoting.txt.in

```
A: @BAR@
B: "@BAR@"
```

quoting.txt

```
A: Some "quoted" value
B: "Some "quoted" value"
```

quoting_escaped.txt

```
A: Some \"quoted\" value
B: "Some \"quoted\" value"
```

Как показано в примере выше, опция ESCAPE_QUOTES приводит к экранированию всех кавычек независимо от их контекста. Поэтому необходимо соблюдать определенную осторожность, если копируемый файл чувствителен к пробелам и кавычкам в любых подстановках, которые могут быть выполнены.

Некоторые типы файлов должны сохранять форму ${someVar} без замены. Классический пример этого - когда копируемый файл представляет собой сценарий оболочки Unix, где ${someVar} является правильным и обычным способом ссылки на переменную оболочки. В таких случаях подстановка может быть ограничена только формой @someVar@ с помощью ключевого слова @ONLY:

CMakeLists.txt

```
set(USER_FILE whoami.txt)
configure_file(whoami.sh.in whoami.sh @ONLY)
```

whoami.sh.in

```
#!/bin/sh

echo ${USER} > "@USER_FILE@"
```

whoami.sh

```
#!/bin/sh

echo ${USER} > "whoami.txt"
```

Подстановку также можно полностью отключить с помощью ключевого слова COPYONLY. Если известно, что подстановка не нужна, указание COPYONLY является хорошей практикой, поскольку предотвращает ненужную обработку и любые неожиданные подстановки.

При использовании configure_file() и подстановке имен файлов или путей распространенной ошибкой является неправильное обращение с пробелами и кавычками. В исходном файле может потребоваться заключить подставляемую переменную в кавычки, если она должна рассматриваться как один путь или имя файла. Вот почему в исходном файле в приведенном выше примере в качестве имени файла для записи вывода использовалось "@USER_FILE@", а не @USER_FILE@.

Замена переменных CMake с помощью формы ${someVar} или @someVar@ может также выполняться для строк, а не только для файлов. Команда string() имеет форму CONFIGURE, которая обеспечивает ту же функциональность:

```
string(CONFIGURE input outVar [@ONLY] [ESCAPE_QUOTES])
```

Опции имеют то же значение, что и для configure_file(). Эта форма может быть полезна, если копируемое содержимое требует более сложных действий, чем простая замена, пример которой приведен в следующем разделе.

Если подстановка не требуется, другой альтернативой является использование команды file() с формой COPY или INSTALL, которые поддерживают одинаковый набор опций:

```
file(<COPY|INSTALL> fileOrDir1 [fileOrDir2...]
    DESTINATION dir
    [NO_SOURCE_PERMISSIONS | USE_SOURCE_PERMISSIONS |
     [FILE_PERMISSIONS permissions...]
     [DIRECTORY_PERMISSIONS permissions...]]
    [FILES_MATCHING]
    [PATTERN pattern | REGEX regex] [EXCLUDE]
    [PERMISSIONS permissions...]
    [...]
)
```

Несколько файлов или даже целые иерархии каталогов могут быть скопированы в выбранный каталог, даже с сохранением симлинков, если они присутствуют. Любые исходные файлы или каталоги, указанные без абсолютного пути, рассматриваются как относительные к текущему исходному каталогу. Аналогично, если каталог назначения не является абсолютным, он будет интерпретироваться как относительный к текущему двоичному каталогу. Структура каталога назначения создается по мере необходимости.

Если источником является имя каталога, оно будет скопировано в место назначения. Чтобы скопировать содержимое каталога в место назначения, добавьте к исходному каталогу прямую косую черту (/), как показано ниже:

```
file(COPY base/srcDir  DESTINATION destDir)  # --> destDir/srcDir
file(COPY base/srcDir/ DESTINATION destDir)  # --> destDir
```

По умолчанию форма COPY приводит к тому, что все файлы и каталоги сохраняют те же разрешения, что и источник, из которого они копируются, в то время как форма INSTALL не сохраняет исходные разрешения. Опции NO_SOURCE_PERMISSIONS и USE_SOURCE_PERMISSIONS могут быть использованы для отмены этих значений по умолчанию, или разрешения могут быть явно указаны с помощью опций FILE_PERMISSIONS и DIRECTORY_PERMISSIONS. Значения разрешений основаны на тех, которые поддерживаются системами Unix:

OWNER_READ	OWNER_WRITE	OWNER_EXECUTE
GROUP_READ	GROUP_WRITE	GROUP_EXECUTE
WORLD_READ	WORLD_WRITE	WORLD_EXECUTE
SETUID	SETGID	

Если определенное разрешение не понятно для данной платформы, оно просто игнорируется. Несколько разрешений могут быть (и обычно так и происходит) перечислены вместе. Например, сценарий оболочки Unix может быть скопирован в текущий двоичный каталог следующим образом:

```
file(COPY whoami.sh
    DESTINATION .
    FILE_PERMISSIONS OWNER_READ OWNER_WRITE OWNER_EXECUTE
                     GROUP_READ GROUP_EXECUTE
                     WORLD_READ WORLD_WRITE
)
```

Подписи COPY и INSTALL также сохраняют временные метки копируемых файлов и каталогов. Более того, если исходный файл уже присутствует в месте назначения с той же временной меткой, копирование этого файла будет считаться уже выполненным и будет пропущено. Единственное отличие между COPY и INSTALL, кроме разрешений по умолчанию, заключается в том, что форма INSTALL печатает сообщения о состоянии для каждого скопированного элемента, в то время как COPY этого не делает. Это различие связано с тем, что форма INSTALL обычно используется как часть сценариев CMake, выполняемых в режиме сценария для установки файлов, где обычным поведением является вывод имени каждого установленного файла.

И COPY, и INSTALL также поддерживают применение определенной логики к файлам, которые соответствуют или не соответствуют определенному шаблону подстановки или регулярному выражению. Это может быть использовано для ограничения копируемых файлов и для отмены разрешений только для совпадающих файлов. В одной команде file() может быть задано несколько шаблонов и регулярных выражений. Использование лучше всего продемонстрировать на примере.

Следующая команда копирует все файлы заголовков (.h) и сценариев (.sh) из someDir, за исключением заголовков, имя файла которых заканчивается на _private.h. Структура каталогов источника сохраняется. Заголовки получают те же разрешения, что и их источник, в то время как сценарии получают разрешения владельца на чтение, запись и выполнение.

```
file(COPY someDir
    DESTINATION .
    FILES_MATCHING
        REGEX .*_private\\.h EXCLUDE
        PATTERN *.h
        PATTERN *.sh
            PERMISSIONS OWNER_READ OWNER_WRITE OWNER_EXECUTE
)
```

Если нужно скопировать весь источник, но разрешения должны быть отменены только для подмножества совпадающих файлов, ключевое слово FILES_MATCHING можно опустить, и шаблоны и регулярные выражения будут использоваться только для применения отмены разрешений.

```
file(COPY someDir
    DESTINATION .
    # Make Unix shell scripts executable by everyone
    PATTERN *.sh PERMISSIONS
        OWNER_READ OWNER_WRITE OWNER_EXECUTE
        GROUP_READ GROUP_EXECUTE
        WORLD_READ WORLD_EXECUTE
    # Ensure only owner can read/write private key files
    REGEX _dsa\$|_rsa\$ PERMISSIONS
        OWNER_READ OWNER_WRITE
)
```

CMake предлагает третий вариант копирования файлов и каталогов. Если configure_file() и file() предназначены для использования во время конфигурирования или, возможно, как часть сценария CMake во время установки, то командный режим CMake можно использовать для копирования файлов и каталогов во время сборки. Командный режим является предпочтительным способом копирования содержимого в рамках правил add_custom_target() и add_custom_command(), поскольку он обеспечивает независимость от платформы (см. раздел 17.5, "Платформонезависимые команды"). Существует три команды, связанные с копированием, первая из которых используется для копирования отдельных файлов:

```
cmake -E copy file1 [file2...] destination
```

Если указан только один исходный файл, то destination интерпретируется как имя файла для копирования, если только он не является именем существующего каталога. Если местом назначения является существующий каталог, исходный файл будет скопирован в него. Такое поведение соответствует поведению родных команд копирования большинства операционных систем, но это также означает, что поведение зависит от состояния файловой системы перед операцией копирования. По этой причине надежнее всегда явно указывать имя целевого файла при копировании одного файла, если только не гарантируется, что местом назначения является каталог, который уже существует.

В качестве удобства, если destination включает путь (относительный или абсолютный), CMake будет пытаться создать путь назначения по мере необходимости при копировании только одного исходного файла. Это означает, что при копировании отдельных файлов команда copy не требует предыдущего шага для

обеспечения существования каталога назначения. Однако если в списке указано более одного исходного файла, каталог назначения должен ссылаться на существующий каталог. И снова для этого можно использовать командный режим CMake, используя команду make_directory, которая создает названный каталог, если он еще не существует, включая при необходимости все родительские каталоги. Ниже показано, как безопасно объединить эти команды командного режима:

```
add_custom_target(copyOne
    COMMAND ${CMAKE_COMMAND} -E copy a.txt output/textfiles/a.txt
)
add_custom_target(copyTwo
    COMMAND ${CMAKE_COMMAND} -E make_directory output/textfiles
    COMMAND ${CMAKE_COMMAND} -E copy a.txt b.txt output/textfiles
)
```

Команда copy всегда будет копировать источник в место назначения, даже если место назначения уже идентично источнику. В результате целевые временные метки всегда обновляются, что иногда может быть нежелательно. Если временные метки не должны обновляться, если файлы уже совпадают, то команда copy_if_different может быть более подходящей:

```
cmake -E copy_if_different file1 [file2...] destination
```

Эта команда работает точно так же, как команда copy, за исключением того, что если исходный файл уже существует в месте назначения и является тем же самым, что и источник, копирование не выполняется, а временная метка цели остается неизменной.

Вместо копирования отдельных файлов в командном режиме можно копировать целые каталоги:

```
cmake -E copy_directory dir1 [dir2...] destination
```

В отличие от команд копирования, связанных с файлами, каталог назначения создается при необходимости, включая любой промежуточный путь. Обратите внимание, что copy_directory копирует *содержимое* исходных каталогов в каталог назначения, а не сами исходные каталоги. Например, предположим, что каталог myDir содержит файл someFile.txt и была выдана следующая команда:

```
cmake -E copy_directory myDir targetDir
```

В результате выполнения этой команды targetDir будет содержать файл someFile.txt, а не myDir/someFile.txt.

Вообще говоря, configure_file() и file() лучше всего подходят для копирования файлов во время конфигурирования, в то время как командный режим CMake является предпочтительным способом копирования во время сборки. Хотя можно использовать командный режим в сочетании с execute_process() для копирования файлов во время конфигурирования, нет особых причин делать это, поскольку configure_file() и file() являются более прямыми и имеют дополнительное преимущество - они автоматически останавливаются при любой ошибке.

18.3. Чтение и запись файлов напрямую

CMake предлагает не только возможность копирования файлов, но и ряд команд для чтения и записи содержимого файлов. Команда file() обеспечивает большую часть функциональности, а самыми простыми являются формы, которые записывают непосредственно в файл:

```
file(WRITE fileName content)
file(APPEND fileName content)
```

Обе эти команды записывают указанное содержимое в названный файл, с той лишь разницей, что если fileName уже существует, то APPEND добавит существующее содержимое, а WRITE отбросит существующее содержимое перед записью. Содержимое является таким же, как и любой другой аргумент функции, и может быть содержимым переменной или строки.

```
set(msg "Hello world")
file(WRITE hello.txt ${msg})
file(APPEND hello.txt " from CMake")
```

В результате приведенного выше примера файл hello.txt будет содержать одну строку текста Hello world от CMake. Обратите внимание, что новые строки не добавляются автоматически, поэтому текст из строки APPEND в приведенном выше примере продолжается непосредственно после текста строки WRITE без разрыва. Чтобы новая строка была записана, она должна быть включена в содержимое, передаваемое команде file(). Один из способов - использовать заключенное в кавычки значение, расположенное в нескольких строках:

```
file(WRITE multi.txt "First line
Second line
")
```

При использовании CMake 3.0 или более поздней версии синтаксис скобок, вдохновленный lua, представленный в разделе 5.1, "Основы переменных", иногда может быть более удобным, поскольку он предотвращает любую подстановку переменных в содержимое.

```
file(WRITE multi.txt [[
First line
Second line
]])

file(WRITE userCheck.sh [=[
#!/bin/bash

[[ -n "${USER}" ]] && echo "Have USER"
]=])
```

В приведенном выше примере содержимое, записываемое в файл multi.txt, состоит только из простого текста без специальных символов, поэтому достаточно использовать простейший синтаксис скобок, в котором символы = можно опустить, оставив только пару квадратных скобок для обозначения начала и конца содержимого. Обратите внимание, как игнорирование первой новой строки сразу после открывающей скобки делает команду более читабельной.

Содержимое файла userCheck.sh гораздо интереснее и подчеркивает особенности синтаксиса скобок. Без синтаксиса скобок CMake увидел бы часть ${USER} и расценил бы ее как подстановку переменной CMake, но поскольку синтаксис скобок не производит такой подстановки, она оставлена как есть. По той же причине различные символы кавычек в содержимом также не интерпретируются как часть содержимого. Их не нужно экранировать, чтобы они не были интерпретированы как начало или конец аргумента. Кроме того, обратите внимание, что встроенное содержимое содержит пару квадратных скобок. Это как раз та ситуация, для которой

предназначено переменное количество знаков = в маркерах начала и конца, позволяющее выбирать маркеры так, чтобы они не совпадали ни с чем в содержимом, которое они окружают. При записи нескольких строк в файл, когда не нужно выполнять подстановку, синтаксис скобок часто является наиболее удобным способом указания записываемого содержимого.

Иногда проекту может потребоваться написать файл, содержимое которого зависит от типа сборки. Наивным подходом было бы предположить, что переменная CMAKE_BUILD_TYPE может быть использована в качестве замены, но это не работает для генераторов мультиконфигурации, таких как Xcode или Visual Studio. Вместо этого можно использовать команду file(GENERATE...):

```
file(GENERATE
    OUTPUT outFile
    INPUT inFile | CONTENT content
    [CONDITION expression]
)
```

Это работает примерно так же, как file(WRITE...), за исключением того, что выписывается один файл для каждого типа сборки, поддерживаемого текущим генератором CMake. Должна присутствовать одна из опций INPUT или CONTENT, но не обе. Они определяют содержимое, которое будет записано в указанный выходной файл. Все аргументы поддерживают выражения генератора, именно так имена файлов и содержимое настраиваются для каждого типа сборки. Тип сборки можно пропустить, используя опцию CONDITION и имея выражение, которое оценивается в 0 для тех типов сборки, которые должны быть пропущены, и в 1 для тех, которые должны быть сгенерированы.

Следующие примеры показывают, как использовать выражения генератора для настройки содержимого и имен файлов в зависимости от типа сборки.

```
# Generate unique files for all but Release
file(GENERATE
    OUTPUT ${CMAKE_CURRENT_BINARY_DIR}/outfile-$<CONFIG>.txt
    INPUT  ${CMAKE_CURRENT_SOURCE_DIR}/input.txt.in
    CONDITION $<NOT:$<CONFIG:Release>>
)

# Embedded content, bracket syntax does not
# prevent the use of generator expressions
file(GENERATE
    OUTPUT ${CMAKE_CURRENT_BINARY_DIR}/details-$<CONFIG>.txt
    CONTENT [[
Built as "$<CONFIG>" for platform "$<PLATFORM_ID>".
]])
```

В первом случае все выражения генератора в содержимом файла input.txt.in будут оценены при записи выходного файла. Это в некоторой степени аналогично тому, как configure_file() подставляет переменные CMake, только в этом случае подстановка производится для выражений генератора. Второй случай демонстрирует, как сочетание синтаксиса скобок со встроенным содержимым может быть особенно удобным способом определения содержимого файла в строке, даже если задействованы генераторные выражения и кавычки.

Обычно для каждого типа сборки выходной файл будет разным. Однако в некоторых ситуациях может быть желательно, чтобы выходной файл всегда был одинаковым, например, когда содержимое файла зависит не от типа сборки, а от некоторых других выражений генератора. Чтобы поддержать такие случаи, CMake позволяет

использовать один и тот же выходной файл для разных типов сборки, но только если содержимое сгенерированного файла также идентично для этих типов сборки. CMake запрещает нескольким командам file(GENERATE...) пытаться сгенерировать один и тот же выходной файл.

Как и для команды file(COPY...), команда file(GENERATE...) изменит выходной файл только в том случае, если его содержимое действительно изменится. Поэтому временная метка выходного файла также будет обновлена, только если содержимое отличается. Это полезно, когда сгенерированный файл используется в качестве входных данных цели сборки, например, сгенерированного заголовочного файла, поскольку это позволяет избежать ненужных перестроек.

Есть несколько важных отличий в поведении file(GENERATE...) по сравнению с большинством других команд CMake. Поскольку она оценивает выражения генератора, она не может сразу записать файлы. Вместо этого файлы записываются в рамках фазы генерации, которая происходит после обработки всех файлов CMakeLists.txt. Это означает, что сгенерированные файлы не будут существовать после возврата команды file(GENERATE...), поэтому файлы не могут быть использованы в качестве входных данных для чего-то другого на этапе конфигурирования. В частности, поскольку сгенерированные файлы не будут существовать до конца фазы configure, их нельзя скопировать или прочитать с помощью configure_file(), file(COPY...) и т.п. Однако их все еще можно использовать в качестве исходных данных для фазы *сборки*, например, сгенерированные исходники или заголовки.

Еще один важный момент, который следует отметить, заключается в том, что до версии CMake 3.10 file(GENERATE...) обрабатывал относительные пути иначе, чем обычно принято в CMake. Поведение относительных путей оставалось неопределенным и обычно оказывалось относительным к рабочему каталогу в момент вызова cmake. Это было ненадежно и непоследовательно, поэтому в CMake 3.10 поведение было изменено, чтобы INPUT действовал относительно текущего исходного каталога, а OUTPUT относительно текущего бинарного каталога, как и большинство других команд CMake, работающих с путями. Проекты должны считать относительные пути небезопасными для использования команды file(GENERATE...), если минимальная версия CMake не установлена на 3.10 или более позднюю.

Команда file() может не только копировать или создавать файлы, она также может использоваться для чтения содержимого файла:

```
file(READ fileName outVar
    [OFFSET offset] [LIMIT byteCount] [HEX]
)
```

Без дополнительных ключевых слов эта команда считывает все содержимое файла fileName и сохраняет его в виде одной строки в outVar. Опция OFFSET может быть использована для чтения только с указанного *смещения*, отсчитываемого в байтах от начала файла. Максимальное количество байт для чтения также может быть ограничено с помощью опции LIMIT. Если задана опция HEX, содержимое файла будет преобразовано в шестнадцатеричное представление, что может быть полезно для файлов, содержащих двоичные данные, а не текст.

Если более желательно разбить содержимое файла построчно, то форма STRINGS может быть более удобной. Вместо того чтобы хранить все содержимое файла в виде одной строки, эта форма хранит его в виде списка, где каждая строка - это один элемент списка. Следующая сокращенная форма показывает наиболее часто используемые опции:

```
file(STRINGS fileName outVar
    [LENGTH_MAXIMUM maxBytesPerLine]
    [LENGTH_MINIMUM minBytesPerLine]
    [LIMIT_INPUT maxReadBytes]
    [LIMIT_OUTPUT maxStoredBytes]
    [LIMIT_COUNT maxStoredLines]
    [REGEX regex]
)
```

Опции, не показанные выше, относятся к кодированию, преобразованию специальных типов файлов или обработке символов новой строки и в большинстве ситуаций не нужны. Обратитесь к документации CMake для получения подробной информации по этим вопросам.

Опции LENGTH_MAXIMUM и LENGTH_MINIMUM можно использовать для исключения строк длиннее или короче определенного количества байт соответственно. Общее количество прочитанных байт может быть ограничено с помощью LIMIT_INPUT, а общее количество сохраненных байт - с помощью LIMIT_OUTPUT. Однако, возможно, более полезной будет опция LIMIT_COUNT, которая ограничивает общее количество сохраненных *строк*, а не количество байт.

Опция REGEX является особенно полезным способом извлечения из файла только определенных строк, представляющих интерес. Например, следующая команда позволяет получить список всех строк в файле myStory.txt, содержащих PKG_VERSION или MODULE_VERSION.

```
file(STRINGS myStory.txt versionLines
     REGEX "(PKG|MODULE)_VERSION"
)
```

Его также можно комбинировать с LIMIT_COUNT для получения только первого совпадения. В следующем примере показано, как объединить file() и string() для извлечения части первой строки, соответствующей регулярному выражению.

```
set(regex "^ *FOO_VERSION *= *([^ ]+) *$")
file(STRINGS config.txt fooVersion
     REGEX "${regex}"
)
string(REGEX REPLACE "${regex}" "\\1" fooVersion "${fooVersion}")
```

Если бы в config.txt была строка вроде этой:

```
FOO_VERSION = 2.3.5
```

Тогда значение, хранящееся в fooVersion, будет 2.3.5.

18.4. Манипулирование файловой системой

Помимо чтения и записи файлов, CMake также поддерживает другие распространенные операции с файловой системой.

```
file(RENAME source destination)
file(REMOVE files...)
file(REMOVE_RECURSE filesOrDirs...)
file(MAKE_DIRECTORY dirs...)
```

Форма RENAME переименовывает файл или каталог, беззвучно заменяя место назначения, если оно уже существует. Источник и место назначения должны быть одного типа, т.е. оба файла или оба каталога. Не разрешается указывать файл в качестве источника и существующий каталог в качестве назначения. Чтобы переместить файл в каталог, имя файла должно быть указано в качестве части назначения. Кроме того, любой путь, являющийся частью назначения, должен уже существовать, форма RENAME не будет создавать промежуточные каталоги.

Форму REMOVE можно использовать для удаления файлов. Если какой-либо из перечисленных файлов не существует, команда file() не сообщает об ошибке. Попытка удалить каталог с помощью формы REMOVE не даст никакого результата. Чтобы удалить каталоги и все их содержимое, используйте форму REMOVE_RECURSE.

Форма MAKE_DIRECTORY обеспечит существование перечисленных каталогов, создавая при необходимости промежуточные пути и не сообщая об ошибке, если каталог уже существует.

Командный режим CMake также поддерживает очень похожий набор возможностей, которые можно использовать во время сборки, а не во время конфигурирования:

```
cmake -E rename source destination
cmake -E remove [-f] files...
cmake -E remove_directory dir
cmake -E make_directory dirs...
```

Эти команды в основном ведут себя так же, как и их аналоги, основанные на file()-командах, с небольшими отличиями. Команда remove_directory может использоваться строго только для одного каталога, в то время как file(REMOVE_RECURSE...) может удалять несколько элементов, и в списке могут быть как файлы, так и каталоги. Команда remove принимает необязательный флаг -f, который изменяет поведение при попытке удалить несуществующий файл. Без -f возвращается ненулевой код выхода, тогда как с -f возвращается нулевой код выхода. Это призвано имитировать поведение команды Unix rm -f.

CMake также поддерживает перечисление содержимого одного или нескольких каталогов с помощью рекурсивной или нерекурсивной формы globbing:

```
file(GLOB outVar
    [LIST_DIRECTORIES true|false]
    [RELATIVE path]
    [CONFIGURE_DEPENDS]   # Requires CMake 3.12 or later
    expressions...
)
```

```
file(GLOB_RECURSE outVar
    [LIST_DIRECTORIES true|false]
    [RELATIVE path]
    [FOLLOW_SYMLINKS]
    [CONFIGURE_DEPENDS]   # Requires CMake 3.12 or later
    expressions...
)
```

Эти команды находят все файлы, имена которых соответствуют любому из предоставленных выражений, которые можно рассматривать как упрощенные регулярные выражения. Возможно, проще воспринимать их как обычные подстановочные знаки с добавлением выбора подмножества символов. Для GLOB_RECURSE они также могут включать компоненты пути. Некоторые примеры должны прояснить базовое использование:

*.txt	Все файлы, имя которых заканчивается на .txt.
foo?.txt	Файлы типа foo2.txt, fooB.txt и т.д.
bar[0-9].txt	Сопоставляет все файлы вида barX.txt, где X - одна цифра.
/images/*.png	Для GLOB_RECURSE это позволит найти только те файлы с расширением .png, которые находятся в подкаталоге images. Это может быть полезным способом поиска файлов в хорошо структурированной иерархии каталогов.

Для GLOB в outVar сохраняются как файлы, так и каталоги, соответствующие выражению. Для GLOB_RECURSE, с другой стороны, имена каталогов не включаются по умолчанию, но это можно контролировать с помощью опции LIST_DIRECTORIES. Кроме того, для GLOB_RECURSE симлинки на каталоги обычно сообщаются как записи в outVar, а не спускаются в них, но опция FOLLOW_SYMLINKS направляет CMake спускаться в каталог, а не перечислять его.

Набор возвращаемых имен файлов по умолчанию будет полным абсолютным путем, независимо от используемых выражений. Опция RELATIVE может быть использована для изменения этого поведения таким образом, что сообщаемые пути будут относительными к определенному каталогу.

```
set(base /usr/share)

file(GLOB_RECURSE images
    RELATIVE ${base}
    ${base}/*/*.png
)
```

Вышеприведенное выражение найдет все изображения ниже /usr/share и включит путь к этим изображениям, только с вычеркнутой частью /usr/share. Обратите внимание на /*/ в выражении, чтобы можно было найти любой каталог ниже базовой точки.

Разработчики должны знать, что команды file(GLOB...) не так быстры, как, скажем, команда оболочки Unix find. Поэтому время выполнения может быть нетривиальным, если использовать ее для поиска частей файловой системы, содержащих много файлов.

Команды file(GLOB) и file(GLOB_RECURSE) - одни из самых неправильно используемых частей CMake. Их не следует использовать для сбора набора файлов, которые будут использоваться в качестве исходных текстов, заголовков или любого другого набора файлов, служащих входными данными для сборки. Одна из причин, по которой этого следует избегать, заключается в том, что если файлы добавляются или удаляются, CMake не запускается автоматически повторно, поэтому сборка не знает об изменениях. Это становится особенно проблематичным, если разработчики используют систему контроля версий и переключаются между ветками и т.д., где набор файлов может измениться, но не таким образом, чтобы вызвать повторный запуск CMake. Система непрерывной интеграции, выполняющая инкрементные сборки, является главным кандидатом на то, чтобы быть пойманной таким использованием. Сайт

Опция CONFIGURE_DEPENDS, добавленная в CMake 3.12, пытается устранить этот недостаток, но она сопровождается снижением производительности и работает только для некоторых генераторов проектов. Следует избегать использования этой опции.

К сожалению, очень часто в руководствах и примерах можно встретить использование file(GLOB) и file(GLOB_RECURSE) для сбора набора исходных текстов для передачи командам add_executable() и add_library(). Это явно не рекомендуется в документации CMake именно по вышеуказанным причинам. Для проектов с большим количеством файлов, разбросанных по нескольким каталогам, существуют лучшие способы сбора набора исходных файлов, которые не страдают от подобных проблем. В разделе 28.5.1, "Целевые исходники", представлены некоторые альтернативные стратегии, которые не только позволяют избежать этих проблем, но и способствуют созданию более модульной и самодостаточной структуры каталогов.

18.5. Загрузка и выгрузка

Команда file() имеет ряд других форм, которые выполняют различные задачи. Удивительно мощная пара подкоманд обеспечивает возможность загрузки файлов с URL и выгрузки файлов на URL.

```
file(DOWNLOAD url fileName [options...])
file(UPLOAD   fileName url [options...])
```

Форма DOWNLOAD загружает файл с указанного url и сохраняет его в fileName. Если задано относительное имя fileName, оно интерпретируется как относительное к текущему двоичному каталогу. Форма UPLOAD выполняет аналогичную операцию, загружая именованный файл на указанный адрес url. Для выгрузки относительный путь интерпретируется как путь к текущему исходному каталогу. И DOWNLOAD, и UPLOAD имеют ряд общих опций:

LOG outVar

Сохраните протоколируемый вывод операции в именованную переменную. Это может быть полезно для диагностики проблем при неудачной загрузке или выгрузке.

SHOW_PROGRESS

Когда эта опция присутствует, информация о ходе выполнения будет записываться в журнал в виде сообщений о состоянии. Это может вызвать довольно шумный этап конфигурирования CMake, поэтому, вероятно, лучше использовать эту опцию только для временной проверки неработающего соединения.

TIMEOUT seconds

Прервите операцию, если прошло более нескольких секунд.

INACTIVITY_TIMEOUT seconds

Это более специфический вид тайм-аута. Некоторые сетевые соединения могут быть плохого качества или просто очень медленными. Может быть желательно позволить операции продолжаться до тех пор, пока она продвигается вперед, но если она застопорится более чем на некоторое приемлемое время, операция должна завершиться неудачей. Опция INACTIVITY_TIMEOUT предоставляет такую возможность, в то время как TIMEOUT позволяет ограничить только общее время.

Форма DOWNLOAD также поддерживает несколько дополнительных опций:

EXPECTED_HASH ALGO= value

Определяет контрольную сумму загружаемого файла, чтобы CMake мог проверить его содержимое. ALGO может быть любым из алгоритмов хэширования, поддерживаемых CMake, наиболее часто используемыми являются MD5 и SHA1. Некоторые старые проекты могут использовать EXPECTED_MD5 в качестве альтернативы EXPECTED_HASH MD5=..., но новые проекты должны предпочесть форму EXPECTED_HASH.

TLS_VERIFY value

Эта опция принимает булево значение, указывающее, следует ли выполнять проверку сертификата сервера при загрузке с URL https://. Если этот параметр не указан, CMake ищет переменную CMAKE_TLS_VERIFY. Если ни опция, ни переменная не определены, то по умолчанию сертификат сервера не проверяется.

TLS_CAINFO fileName

С помощью этой опции можно указать пользовательский файл центра сертификации. Это влияет только на урлы https://.

В CMake 3.7 или более поздней версии следующие опции также доступны для DOWNLOAD и UPLOAD:

USERPWD username:password

Предоставляет данные аутентификации для операции. Помните, что жесткое кодирование паролей является проблемой безопасности и в целом его следует избегать. Если вы предоставляете пароли с помощью этой опции, содержимое должно поступать извне проекта, например, из защищенного соответствующим образом файла, считываемого с локальной машины пользователя во время конфигурирования.

HTTPHEADER header

Включает HTTP-заголовок для операции и может быть повторен несколько раз по мере необходимости для предоставления более чем одного значения заголовка. Следующий частичный пример демонстрирует один из мотивирующих случаев для этой опции:

```
file(DOWNLOAD "https://somebucket.s3.amazonaws.com/myfile.tar.gz"
    myfile.tar.gz
    EXPECTED_HASH SHA1=${myfileHash}
    HTTPHEADER "Host: somebucket.s3.amazonaws.com"
    HTTPHEADER "Date: ${timestamp}"
    HTTPHEADER "Content-Type: application/x-compressed-tar"
    HTTPHEADER "Authorization: AWS ${s3key}:${signature}"
)
```

Команды загрузки и выгрузки на основе file()-файлов чаще всего используются для установки, упаковки или составления отчетов о тестировании, но иногда они могут использоваться и для других целей. Примеры включают такие вещи, как загрузка загрузочных файлов во время конфигурирования или внесение в сборку файла, который не может или не должен храниться как часть исходных текстов проекта (например,

конфиденциальные файлы, которые должны быть доступны только определенным разработчикам, очень большие файлы и т.д.). В последующих главах приводятся конкретные сценарии, в которых эти команды используются с большим эффектом.

18.6. Рекомендуемые практики

В этой главе был представлен ряд функциональных возможностей CMake, связанных с работой с файлами. Различные методы могут быть использованы очень эффективно для выполнения ряда задач независимо от платформы, но ими также можно злоупотреблять. Создание хороших привычек и шаблонов и их последовательное применение на протяжении всего проекта поможет новым разработчикам освоить лучшие методы.

Команда configure_file() - это команда, которую часто упускают из виду новые разработчики, однако она является ключевым методом создания файла, содержимое которого может быть изменено в соответствии с переменными, определенными во время конфигурирования, или даже для простого копирования файла. Обычное соглашение об именовании заключается в том, что имя файла в исходном и целевом файлах должно быть одинаковым, за исключением того, что в исходном файле добавляется дополнительный .in. Некоторые среды IDE понимают это соглашение и по-прежнему обеспечивают соответствующую подсветку синтаксиса для исходного файла на основе расширения файла без суффикса .in. Наличие суффикса .in не только служит четким напоминанием о том, что файл необходимо преобразовать/скопировать перед использованием, но и предотвращает его случайный выбор вместо целевого файла, если CMake или компилятор ищут файлы в нескольких каталогах. Это особенно актуально, когда целевой файл является заголовком C/C++, а текущие исходный и двоичный каталоги находятся в пути поиска заголовка.

Выбор наиболее подходящей команды для копирования файлов не всегда очевиден. Следующее может послужить полезным руководством при выборе между configure_file(), file(COPY) и file(INSTALL):

- Если содержимое файла необходимо изменить, чтобы включить замены переменных CMake, configure_file() является наиболее лаконичным способом добиться этого.
- Если файл нужно просто скопировать, но его имя будет изменено, синтаксис configure_file() немного короче, чем file(COPY...), но подойдет любой из них.
- При копировании более одного файла или целой структуры каталога необходимо использовать команду file(COPY) или file(INSTALL).
- Если при копировании требуется контроль над разрешениями файлов или каталогов, необходимо использовать file(COPY) или file(INSTALL).
- file(INSTALL) обычно используется только в сценариях установки. В других ситуациях лучше использовать file(COPY).

До версии CMake 3.10 команда file(GENERATE...) по-разному обрабатывала относительные пути по сравнению с большинством других команд, предоставляемых CMake. Вместо того чтобы полагаться на то, что разработчики знают об этом отличии, проекты должны предпочитать всегда указывать абсолютный путь к файлам INPUT и OUTPUT, чтобы избежать ошибок или создания файлов в неожиданных местах.

При загрузке или выгрузке файлов с помощью команд file(DOWNLOAD...) или file(UPLOAD...) следует тщательно учитывать аспекты безопасности и эффективности. Стремитесь избегать встраивания каких-либо аутентификационных данных (имена пользователей, пароли, закрытые ключи и т.д.) в любой файл, хранящийся в системе контроля версий исходных текстов проекта. Такие данные должны поступать извне проекта, например, через переменные окружения (все еще несколько небезопасные), файлы, находящиеся в файловой системе пользователя с соответствующими разрешениями, ограничивающими доступ, или через какую-либо

связку ключей. Используйте параметр EXPECTED_HASH при загрузке, чтобы повторно использовать ранее загруженное содержимое из предыдущего запуска и избежать потенциально трудоемкой удаленной операции. Если хэш загружаемого файла не может быть известен заранее, то для обеспечения целостности содержимого настоятельно рекомендуется использовать параметр TLS_VERIFY. Также рассмотрите возможность указания TIMEOUT, INACTIVITY_TIMEOUT или обоих параметров, чтобы предотвратить блокировку выполнения конфигурации на неопределенный срок при плохом или ненадежном сетевом соединении.

Глава 19. Указание сведений о версии

Версионность - одна из тех вещей, которым часто не уделяется должного внимания. Важность того, что номер версии сообщает пользователям, часто недооценивается, что приводит к тому, что пользователи не оправдывают ожиданий или путаются в изменениях между релизами. Также неизбежны противоречия между маркетингом и тем, как стратегия версионности влияет на техническую реализацию сборок, упаковки и так далее. Заблаговременное обдумывание и определение этих вопросов ставит проект в более выгодное положение, когда приходит время выпуска первого публичного релиза. В этой главе рассматриваются способы реализации эффективной стратегии версионирования с использованием возможностей CMake для обеспечения надежного и эффективного процесса.

19.1. Версия проекта

Версия проекта часто должна быть определена в начале файла CMakeLists.txt верхнего уровня, чтобы различные части сборки могли ссылаться на нее. Исходный код может захотеть встроить версию проекта, чтобы она отображалась пользователю или записывалась в лог-файл, этапы упаковки могут нуждаться в ней для определения деталей версии релиза и так далее. Можно просто установить переменную в начале файла CMakeLists.txt для записи номера версии в любой необходимой форме, например, так:

```
cmake_minimum_required(VERSION 3.0)
project(FooBar)
set(FooBar_VERSION 2.4.7)
```

Если необходимо извлечь отдельные компоненты, может потребоваться несколько более сложный набор переменных, один из примеров которого может выглядеть следующим образом:

```
cmake_minimum_required(VERSION 3.0)
project(FooBar)
set(FooBar_VERSION_MAJOR 2)
set(FooBar_VERSION_MINOR 4)
set(FooBar_VERSION_PATCH 7)
set(FooBar_VERSION
    ${FooBar_VERSION_MAJOR}.${FooBar_VERSION_MINOR}.${FooBar_VERSION_PATCH}
)
```

В разных проектах могут использоваться разные соглашения для именования переменных. Структура номеров версий также может отличаться от проекта к проекту, и в результате отсутствие согласованности усложняет объединение многих проектов в более крупную коллекцию или суперсборку (обсуждается в разделе 28.1, "Структура суперсборки"). В CMake 3.0 появилась новая функциональность, которая упрощает указание деталей версии и вносит некоторую последовательность в нумерацию версий проектов. В команду project() было добавлено ключевое слово VERSION, которое определяет номер версии в виде *major.minor.patch.tweak* как ожидаемый формат. На основе этой информации автоматически заполняется набор переменных, чтобы сделать полную строку версии, а также каждый компонент версии по отдельности доступным для остальной части проекта. Если строка версии предоставлена с опущенными частями (например, часто опускается часть *tweak*), соответствующие переменные остаются пустыми. В следующей таблице показаны автоматически заполняемые переменные версии, когда ключевое слово VERSION используется с командой project():

PROJECT_VERSION	projectName_VERSION
PROJECT_VERSION_MAJOR	projectName_VERSION_MAJOR
PROJECT_VERSION_MINOR	projectName_VERSION_MINOR
PROJECT_VERSION_PATCH	projectName_VERSION_PATCH
PROJECT_VERSION_TWEAK	projectName_VERSION_TWEAK

Эти два набора переменных служат немного разным целям. Переменные projectName_... можно использовать для получения сведений о версии в любом месте текущего каталога или ниже. Вызов типа project(FooBar VERSION 2.7.3) приводит к появлению переменных с именами FooBar_VERSION, FooBar_VERSION_MAJOR и так далее. Поскольку два вызова команды project() не могут использовать одно и то же имя проекта, эти специфические для проекта переменные не будут перезаписаны другими вызовами команды project(). Переменные PROJECT_..., с другой стороны, обновляются при каждом вызове project(), поэтому их можно использовать для получения информации о версии последнего вызова project() в текущей области видимости или выше. Начиная с CMake 3.12, аналогичный набор переменных также предоставляет сведения о версии, установленной вызовом project() в файле CMakeLists.txt верхнего уровня. Этими переменными являются:

CMAKE_PROJECT_VERSION
CMAKE_PROJECT_VERSION_MAJOR
CMAKE_PROJECT_VERSION_MINOR
CMAKE_PROJECT_VERSION_PATCH
CMAKE_PROJECT_VERSION_TWEAK

Этой же схеме следуют и переменные для имени проекта, описания и url домашней страницы, последние две были добавлены в CMake версий 3.9 и 3.12 соответственно. В качестве общего руководства переменные PROJECT_... могут быть полезны для общего кода (особенно модулей) как способ определения разумных значений по умолчанию для таких вещей, как упаковка или детали документации. Переменные CMAKE_PROJECT_... тоже иногда используются для умолчаний, но они могут быть немного менее надежными, поскольку их использование обычно предполагает наличие определенного проекта верхнего уровня. Переменные projectName_... являются наиболее надежными, поскольку они всегда однозначно указывают, детали какого проекта они будут предоставлять.

При работе с проектами, поддерживающими CMake более ранних версий, чем 3.0, иногда возникает ситуация, когда они определяют свои собственные переменные, связанные с версиями, которые противоречат переменным, автоматически определяемым CMake 3.0 и более поздними версиями. Это может привести к появлению предупреждений политики CMP0048, которые указывают на конфликт. Ниже показан пример кода, который приводит к такому предупреждению:

```
cmake_minimum_required(VERSION 2.8.12)

set(FooBar_VERSION 2.4.7)
project(FooBar)
```

В приведенном выше примере переменная FooBar_VERSION установлена явно, но это имя переменной конфликтует с переменной, которую автоматически определит команда project(). В результате предупреждение политики призвано побудить проект либо использовать другое имя переменной, либо обновить CMake до минимальной версии 3.0 и вместо этого установить сведения о версии в команде project().

19.2. Доступ к исходному коду к сведениям о версии

После того как сведения о версии определены в файле CMakeLists.txt, очень часто возникает необходимость сделать их доступными для исходного кода, скомпилированного в рамках проекта. Для этого можно использовать несколько различных подходов, каждый из которых имеет свои достоинства и недостатки. Один из наиболее распространенных методов, используемых новичками в CMake, заключается в добавлении определения компилятора на верхнем уровне проекта:

```
cmake_minimum_required(VERSION 3.0)
project(FooBar VERSION 2.4.7)
add_definitions(-DFOOBAR_VERSION=\"${FooBar_VERSION}\")
```

Это делает версию доступной в виде необработанной строки, которую можно использовать следующим образом:

```
void printVersion()
{
    std::cout << FOOBAR_VERSION << std::endl;
}
```

Хотя этот подход довольно прост, добавление определения к компиляции каждого отдельного файла в проекте имеет некоторые недостатки. Помимо загромождения командной строки каждого компилируемого файла, это означает, что при изменении номера версии весь проект будет перестроен. Это может показаться незначительным, но разработчики, которые регулярно переключаются между различными ветками в системе контроля исходных текстов, почти наверняка будут раздражены всеми этими ненужными перекомпиляциями. Несколько лучший подход использует свойства источника для определения символа FOOBAR_VERSION только для тех файлов, где он необходим. Например:

```
cmake_minimum_required(VERSION 3.0)
project(FooBar VERSION 2.4.7)

add_executable(foobar main.cpp src1.cpp src2.cpp ...)

get_source_file_property(defs src1.cpp COMPILE_DEFINITIONS)
list(APPEND defs "FOOBAR_VERSION=\"${FooBar_VERSION}\"")
set_source_files_properties(src1.cpp PROPERTIES
    COMPILE_DEFINITIONS ${defs}
)
```

Это позволяет избежать добавления определения компилятора в каждый файл, вместо этого добавляя его только в те файлы, которые в нем нуждаются. Однако, как упоминалось в разделе 9.5, "Свойства источника", при установке отдельных свойств источника могут возникнуть негативные последствия для зависимостей сборки, что снова приведет к пересборке большего количества файлов, чем это необходимо. Поэтому такой подход может показаться улучшением, но часто это не так.

Вместо того, чтобы передавать сведения о версии в командной строке, другой распространенный подход заключается в использовании configure_file() для записи заголовочного файла, который предоставляет сведения о версии. Например:

foobar_version.h.in

```
#include <string>

inline std::string getFooBarVersion()
{
    return "@FooBar_VERSION@";
}

inline unsigned getFooBarVersionMajor()
{
    return @FooBar_VERSION_MAJOR@;
}

inline unsigned getFooBarVersionMinor()
{
    return @FooBar_VERSION_MINOR@ +0;
}

inline unsigned getFooBarVersionPatch()
{
    return @FooBar_VERSION_PATCH@ +0;
}

inline unsigned getFooBarVersionTweak()
{
    return @FooBar_VERSION_TWEAK@ +0;
}
```

main.cpp

```
#include "foobar_version.h"
#include <iostream>

int main(int argc, char* argv[])
{
    std::cout << "VERSION = " << getFooBarVersion() << "\n"
              << "MAJOR   = " << getFooBarVersionMajor() << "\n"
              << "MINOR   = " << getFooBarVersionMinor() << "\n"
              << "PATCH   = " << getFooBarVersionPatch() << "\n"
              << "TWEAK   = " << getFooBarVersionTweak()
              << std::endl;
    return 0;
}
```

CMakeLists.txt

```
cmake_minimum_required(VERSION 3.0)
project(FooBar VERSION 2.4.7)

configure_file(foobar_version.h.in foobar_version.h @ONLY)
add_executable(foobar main.cpp)
target_include_directories(foobar PRIVATE "${CMAKE_CURRENT_BINARY_DIR}")
```

Значение +0 в foobar_version.h.in необходимо для частей *minor*, *patch* и *tweak,* чтобы соответствующие переменные были пустыми в случае, если эти компоненты версии опущены.Предоставление сведений о версии через такой заголовок является улучшением по сравнению с предыдущими методами. Информация о версии не включается в командную строку при компиляции любого исходного файла, и только те файлы, которые #include заголовок foobar_version.h, будут перекомпилированы при изменении информации о версии. Предоставление всех различных компонентов версии, а не только строки версии, также не влияет на командные строки. Тем не менее, если номер версии необходим во многих различных исходных файлах, это все равно может привести к большей перекомпиляции, чем это действительно необходимо. Этот подход может быть усовершенствован путем переноса реализаций из заголовка в собственный файл .cpp и компиляции его как собственной библиотеки.

foobar_version.h

```
#include <string>

std::string getFooBarVersion();
unsigned    getFooBarVersionMajor();
unsigned    getFooBarVersionMinor();
unsigned    getFooBarVersionPatch();
unsigned    getFooBarVersionTweak();
```

foobar_version.cpp.in

```
#include "foobar_version.h"

std::string getFooBarVersion()
{
    return "@FooBar_VERSION@";
}

unsigned getFooBarVersionMajor()
{
    return @FooBar_VERSION_MAJOR@;
}

unsigned getFooBarVersionMinor()
{
    return @FooBar_VERSION_MINOR@ +0;
}

unsigned getFooBarVersionPatch()
{
    return @FooBar_VERSION_PATCH@ +0;
}

unsigned getFooBarVersionTweak()
{
    return @FooBar_VERSION_TWEAK@ +0;
}
```

CMakeLists.txt

```
cmake_minimum_required(VERSION 3.0)
project(FooBar VERSION 2.4.7)

configure_file(foobar_version.cpp.in foobar_version.cpp @ONLY)
add_library(foobar_version STATIC ${CMAKE_CURRENT_BINARY_DIR}/foobar_version.cpp)

add_executable(foobar main.cpp)
target_link_libraries(foobar PRIVATE foobar_version)

add_library(fooToolkit mylib.cpp)
target_link_libraries(fooToolkit PRIVATE foobar_version)
```

Эта схема не имеет недостатков предыдущих подходов. При изменении сведений о версии нужно перекомпилировать только один исходный файл (сгенерированный файл foobar_version.cpp) и перелинковать только цели foobar и fooToolkit. Заголовок foobar_version.h никогда не изменяется, поэтому любой файл, зависящий от него, не устареет при изменении деталей версии. Никакие опции не добавляются в командную строку компиляции любого исходного файла, поэтому никакие другие перекомпиляции не вызываются в результате изменения деталей версии.

В ситуациях, когда проект предоставляет библиотеку и заголовок как часть пакета релиза, вышеописанная схема также надежна. Заголовок не содержит сведений о версии, а библиотека содержит. Поэтому код, использующий библиотеку, может вызывать функции версии и быть уверенным в том, что полученные им данные являются теми, с которыми библиотека была собрана. Это может быть полезно в сложных средах конечных пользователей, где может быть установлено несколько версий проекта, которые не всегда структурированы так, как предполагалось проектом.

One variant of this approach is to make foobar_version an object library rather than a static library. The end result is more or less the same, but there isn't much to be gained and it may feel less natural to some developers. Making it a shared library loses some of the robustness advantages and again introduces a little more complexity for little benefit, so it would generally be recommended to make these sort of version libraries static.

19.3. Коммиты контроля исходного кода

Нередко проекты хотят записывать детали, связанные с их системой контроля исходных текстов. Это может включать ревизию или хэш коммита исходных текстов на момент сборки, имя текущей ветки или последней метки и так далее. Описанный выше подход с предоставлением информации о версии через специальный .cpp-файл хорошо подходит для добавления дополнительных функций для возврата такой информации. Например, текущий git-хэш может быть предоставлен относительно просто:

foobar_version.cpp.in

```
std::string getFooBarGitHash()
{
    return "@FooBar_GIT_HASH@";
}
// Other functions as before...
```

CMakeLists.txt

```
cmake_minimum_required(VERSION 3.0)
project(FooBar VERSION 2.4.7)

# The find_package() command is covered later in the Finding Things chapter.
# Here, it provides the GIT_EXECUTABLE variable after searching for the
# git binary in some standard/well-known locations for the current platform.
find_package(Git REQUIRED)
execute_process(
    COMMAND ${GIT_EXECUTABLE} rev-parse HEAD
    RESULT_VARIABLE result
    OUTPUT_VARIABLE FooBar_GIT_HASH
    OUTPUT_STRIP_TRAILING_WHITESPACE
)
if(result)
    message(FATAL_ERROR "Failed to get git hash: ${result}")
endif()

configure_file(foobar_version.cpp.in foobar_version.cpp @ONLY)
# Targets, etc....
```

Несколько более интересный пример - измерение количества фиксаций, произошедших с момента изменения конкретного файла. Рассмотрим встраивание версии проекта в отдельный файл, а не в файл CMakeLists.txt, где единственное, что есть в этом отдельном файле - это номер версии проекта. Тогда можно сделать разумное предположение, что файл изменяется только при изменении номера версии. В результате, измерение количества коммитов с момента изменения этого файла в текущей ветке обычно является хорошим показателем количества коммитов с момента последнего обновления версии.

Следующий пример переносит версию проекта в отдельный файл projectVersionDetails.cmake и предоставляет количество коммитов через новую функцию в сгенерированном файле foobar_version.cpp. Он демонстрирует модель, подходящую для любого проекта, где версия устанавливается вызовом project() верхнего уровня, но таким образом, чтобы не мешать родительскому проекту, если он включен в более крупную иерархию проектов (эта тема обсуждается в разделе 27.2, "FetchContent").

foobar_version.cpp.in

```
unsigned getFooBarCommitsSinceVersionChange()
{
    return @FooBar_COMMITS_SINCE_VERSION_CHANGE@;
}
// Other functions as before...
```

projectVersionDetails.cmake

```
# This file should contain nothing but the following line
# setting the project version. The variable name must not
# clash with the FooBar_VERSION* variables automatically
# defined by the project() command.
set(FooBar_VER 2.4.7)
```

CMakeLists.txt

```
cmake_minimum_required(VERSION 3.0)
include(projectVersionDetails.cmake)
project(FooBar VERSION ${FooBar_VER})

find_package(Git REQUIRED)
execute_process(
    COMMAND ${GIT_EXECUTABLE} rev-list -1 HEAD projectVersionDetails.cmake
    RESULT_VARIABLE result
    OUTPUT_VARIABLE lastChangeHash
    OUTPUT_STRIP_TRAILING_WHITESPACE
)
if(result)
    message(FATAL_ERROR "Failed to get hash of last change: ${result}")
endif()

execute_process(
    COMMAND ${GIT_EXECUTABLE} rev-list ${lastChangeHash}..HEAD
    RESULT_VARIABLE result
    OUTPUT_VARIABLE hashList
    OUTPUT_STRIP_TRAILING_WHITESPACE
)
if(result)
    message(FATAL_ERROR "Failed to get list of git hashes: ${result}")
endif()
string(REGEX REPLACE "[\n\r]+" ";" hashList "${hashList}")
list(LENGTH hashList FooBar_COMMITS_SINCE_VERSION_CHANGE)

configure_file(foobar_version.cpp.in foobar_version.cpp @ONLY)
# Targets, etc....
```

Приведенный выше подход определяет git-хэш последнего изменения в файле сведений о версии, а затем использует git rev-list для получения списка хэшей коммитов для всего репозитория с момента этого коммита. Коммиты изначально находятся в виде строки с одним хэшем в строке, которая затем преобразуется в список CMake путем замены символов новой строки на разделитель списка (;). Затем команда list() просто подсчитывает количество элементов в списке, чтобы получить число коммитов. Более простой подход - использовать git rev-list --count для получения числа напрямую, но старые версии git не поддерживают опцию --count, поэтому описанный выше метод предпочтительнее, если требуется поддержка старых версий git.

Возможны и другие вариации. Некоторые проекты используют git describe для предоставления различных деталей, включая имена ветвей, последнюю метку и т.д., но обратите внимание, что детали метки и ветви могут меняться без изменения коммитов. Если ветвь или тег перемещены или переименованы, сборка может не повториться. Если детали версии зависят только от хэшей коммитов файлов, то такой слабости не возникает. Это также дает проекту свободу в создании, переименовании или удалении тегов по мере необходимости после того, как сборки подтвердили отсутствие ошибок в коммитах (подумайте о тегах релиза, применяемых к коммитам после того, как сборки непрерывной интеграции, тестирование и т.д. подтвердили отсутствие проблем).

Системы контроля исходных текстов, такие как Subversion, представляют другие проблемы. С одной стороны, Subversion поддерживает глобальный номер ревизии для всего хранилища, поэтому нет необходимости сначала получать хэши коммитов, а затем подсчитывать их. Но Subversion также имеет ту сложность, что она

позволяет смешивать различные ревизии разных файлов. В результате подходы, подобные описанному выше для git, могут

может быть побежден, если разработчик проверяет различные ревизии файлов, но оставляет файл версии проекта в покое. Это не тот сценарий, который можно ожидать для автоматизированной системы непрерывной интеграции, но он может быть более вероятен для разработчика, работающего локально на своей машине, в зависимости от того, как ему нравится работать.

Еще одно соображение, касающееся методов, подобных приведенным выше, заключается в том, что заставляет обновлять сгенерированный файл версии .cpp. CMake обеспечивает повторное выполнение шага configure при изменении файла версии проекта, поскольку он вносится в основной файл CMakeLists.txt командой include(). Однако если фиксации будут сделаны в других файлах, CMake не будет знать об этом. Возможно, можно внедрить крючки в систему контроля версий (например, крючок post-commit в git), чтобы заставить CMake повторно запуститься, но это скорее раздражает разработчиков, чем помогает им. В конечном итоге, как правило, приходится искать компромисс между удобством и надежностью. Тем не менее, точность данных контроля исходных текстов, скорее всего, будет критична только для релизов, и должно быть достаточно просто обеспечить, чтобы процесс релиза явно вызывал CMake.

19.4. Рекомендуемые практики

Проекты не обязаны следовать какой-либо определенной системе версий, но если следовать формату *major.minor.patch.tweak, mo* определенные функциональные возможности предоставляются в CMake бесплатно, а новым разработчикам легче понять, какая система версий используется в проекте. Как будет показано в последующих главах (в частности, в главе 26, *Packaging*), формат версии более важен при создании упакованных релизов, но поскольку многие проекты сообщают свой собственный номер версии во время выполнения, формат версии влияет и на сборку.

Значение каждого из чисел, составляющих формат версии, зависит от проекта, но есть соглашения, которые часто ожидают конечные пользователи. Например, изменение значения *major* обычно означает значительный релиз, часто включающий изменения, которые не являются обратно совместимыми или представляют собой изменение направления развития проекта. Если изменяется *второстепенное* значение, пользователи склонны рассматривать это как инкрементный релиз, скорее всего, добавляющий новые возможности без нарушения существующего поведения. Если изменяется только значение *patch*, пользователи могут не воспринимать это как особенно важное изменение и ожидать, что оно будет относительно незначительным, например, исправление некоторых ошибок, но не введение новой функциональности. Значение tweak часто опускается и не имеет общепринятой интерпретации, кроме того, что оно еще менее значимо, чем *patch*. Обратите внимание, что это лишь общие наблюдения, проекты могут придавать номерам версий совершенно разные значения. Для максимальной простоты проект может использовать только один номер и ничего больше, фактически определяя каждый релиз как новую *основную* версию. Хотя это было бы легко реализовать, это также обеспечит меньшее руководство для конечных пользователей и потребует качественных примечаний к релизу для управления ожиданиями пользователей между каждой версией.

Ключевое слово VERSION команды project() является одним из примеров того, как CMake обеспечивает дополнительные удобства при использовании формата *major.minor.patch.tweak*. Проект предоставляет одну строку версии, а команда project() автоматически определяет набор переменных, делающих доступными различные части номера версии. Некоторые модули CMake могут также использовать эти переменные в качестве значений по умолчанию для определенных метаданных, поэтому обычно рекомендуется задавать версию проекта командой project() с помощью ключевого слова VERSION. Это ключевое слово было добавлено в CMake 3.0, но при поддержке более старых версий CMake эту функциональность все равно необходимо учитывать. Проекты не должны определять переменные, имена которых противоречат автоматически

определенным, иначе последующие версии CMake выдадут предупреждение. Избегайте явного задания переменных с именами вида xxx_VERSION или xxx_VERSION_yyy для предотвращения подобных предупреждений.

Определяя номер версии, подумайте о том, чтобы сделать это в отдельном специальном файле, который CMake затем подтягивает с помощью команды include(). Это позволит проекту воспользоваться преимуществами изменений в номере версии, совпадающих с изменениями в этом файле, как их видит система контроля исходных текстов проекта. Чтобы свести к минимуму ненужную перекомпиляцию при изменении версии, создайте файл .c или .cpp, содержащий функции, возвращающие сведения о версии, а не вставляйте эти сведения в генерируемый заголовок или в определения компилятора, передаваемые в командной строке. Также убедитесь, что имена, присвоенные таким функциям, содержат что-то специфическое для проекта или поместите их в специфическое для проекта пространство имен. Это позволяет воспроизводить один и тот же шаблон во многих проектах, которые впоследствии могут быть объединены в одну сборку, не вызывая столкновения имен.

Установите стратегии версионирования и модели реализации на ранних стадиях проекта. Это помогает разработчикам получить четкое представление о том, как и когда обновляются сведения о версиях, и способствует продумыванию процесса выпуска задолго до первой поставки. Это также позволяет отсеять менее эффективные подходы на ранней стадии, чтобы максимально повысить эффективность сборки перед выпуском релизов, в которых номера версий меняются, а время выполнения сборки может стать более важным.

Глава 20. Библиотеки

По сравнению с написанием обычных приложений, создание и поддержка библиотек, особенно разделяемых библиотек, обычно более сложны. Все обычные заботы о корректности и сопровождаемости кода по-прежнему актуальны, но библиотеки с общим доступом, в частности, несут в себе дополнительные соображения, связанные с согласованностью API, сохранением бинарной совместимости между релизами, видимостью символов и т.д. Кроме того, каждая платформа, как правило, имеет свой собственный набор уникальных особенностей и требований, что делает разработку кросс-платформенных библиотек сложной задачей.

Однако в большинстве случаев основной набор возможностей поддерживается всеми основными платформами, просто способы их определения и использования различны. CMake предоставляет ряд функций, которые абстрагируют эти различия, чтобы разработчики могли сосредоточиться на возможностях и оставить детали реализации на усмотрение системы сборки.

20.1. Основы построения

Основная команда для определения библиотеки была рассмотрена в предыдущих главах и имеет следующий вид:

```
add_library(targetName [STATIC | SHARED | MODULE | OBJECT]
            [EXCLUDE_FROM_ALL]
            source1 [source2 ...])
```

Общая библиотека будет создана, если указано ключевое слово SHARED или MODULE. Альтернативно, если не указано ключевое слово STATIC, SHARED, MODULE или OBJECT, будет создана разделяемая библиотека, если переменная BUILD_SHARED_LIBS имеет значение true во время вызова add_library().

Основное различие между SHARED и MODULE заключается в том, что библиотеки SHARED предназначены для компоновки с другими объектами, а библиотеки MODULE - нет. Библиотеки MODULE обычно используются для таких вещей, как плагины или другие дополнительные библиотеки, которые могут быть загружены во время выполнения. Загрузка таких библиотек часто зависит от настроек конфигурации приложения или обнаружения некоторых системных функций. Другие исполняемые файлы и библиотеки обычно не связываются с библиотекой MODULE.

На большинстве платформ на базе Unix имя файла библиотеки STATIC или SHARED по умолчанию имеет приставку lib, в то время как MODULE может этого не делать. Платформы Apple также поддерживают фреймворки и загружаемые пакеты, которые позволяют подключать к библиотеке дополнительные файлы в четко определенной структуре каталогов. Подробно об этом говорится в разделе 22.3, "Фреймворки".

На платформах Windows имена библиотек не имеют префикса lib, независимо от типа библиотеки. Цели статических библиотек создают один архив .lib, тогда как цели разделяемых библиотек создают два отдельных файла, один из которых предназначен для среды выполнения (.dll или динамически подключаемая библиотека), а другой - для компоновки во время сборки (т.е. библиотека импорта .lib). Разработчики иногда путают импортные и статические библиотеки из-за того, что для обоих используется один и тот же суффикс файла, но CMake обычно обрабатывает их правильно без какого-либо специального вмешательства.

При использовании инструментов GNU в Windows (например, с генераторами проектов MinGW или MSYS) CMake имеет возможность конвертировать библиотеки импорта GNU (.dll.a) в тот же формат, который создает

Visual Studio (.lib). Это может быть полезно при распространении общей библиотеки, созданной с помощью инструментов GNU, чтобы ее можно было связать с двоичными файлами, созданными с помощью Visual Studio. Обратите внимание, что для этого преобразования должна быть установлена Visual Studio. Преобразование включается установкой свойства GNUtoMS target в true для разделяемой библиотеки. Это целевое свойство инициализируется значением переменной CMAKE_GNUtoMS во время вызова add_library().

20.2. Связывание статических библиотек

CMake обрабатывает некоторые особые случаи, специфичные для компоновки статических библиотек. Если библиотека A указана как ПРИВЛЕКАТЕЛЬНАЯ зависимость для статической библиотеки-цели B, то A будет рассматриваться как ПУБЛИЧНАЯ зависимость в том, что касается линковки (и *только* для линковки). Это происходит потому, что частная библиотека A все равно должна быть добавлена в командную строку компоновщика всего, что компонуется с B, чтобы символы из A были найдены во время компоновки. Если бы B была разделяемой библиотекой, частную библиотеку A, от которой она зависит, не нужно было бы указывать в командной строке компоновщика. Все это прозрачно обрабатывается CMake, поэтому разработчику обычно не нужно заботиться о деталях, кроме указания зависимостей PUBLIC, PRIVATE и INTERFACE в функции target_link_libraries().

В типичных проектах статические библиотеки не содержат циклических зависимостей, когда две или более библиотек зависят друг от друга. Тем не менее, некоторые сценарии приводят к таким ситуациям, и CMake распознает и обработает циклическую зависимость, если только были указаны соответствующие отношения связывания (например, с помощью target_link_libraries()). Немного измененная версия примера из документации CMake демонстрирует такое поведение:

```
add_library(A STATIC a.cpp)
add_library(B STATIC b.cpp)
target_link_libraries(A PUBLIC B)
target_link_libraries(B PUBLIC A)
add_executable(main main.cpp)
target_link_libraries(main A)
```

В приведенном выше примере команда link для main будет содержать A B A B. Это повторение обеспечивается CMake автоматически, без вмешательства разработчика, но в некоторых патологических случаях может потребоваться более одного повторения. Хотя CMake предоставляет для этого целевое свойство LINK_INTERFACE_MULTIPLICITY, такие ситуации обычно указывают на необходимость реструктуризации проекта. Библиотеки OBJECT также могут быть полезным инструментом для решения таких глубоких взаимозависимостей, поскольку они действуют как коллекция источников, а не как реальные библиотеки. Порядок следования объектных файлов в командной строке компоновщика обычно не важен, в то время как порядок следования библиотек, безусловно, важен.

20.3. Версионирование общих библиотек

Проект CMake, который не предполагает, что его библиотеки будут использоваться за пределами самого проекта, обычно не нуждается в информации о версии для любых общих библиотек, которые он создает. Весь проект, как правило, обновляется вместе при развертывании, поэтому не возникает проблем с обеспечением бинарной совместимости между релизами и т.д. Но если проект предоставляет библиотеки, и другие программы могут ссылаться на них, версионность библиотек становится очень важной. Детали версий

библиотек повышают надежность, позволяя другим программам указывать интерфейс, который они ожидают связать с библиотеками и который будет доступен им во время выполнения.

Большинство платформ предлагают функциональность для указания номера версии разделяемой библиотеки, но способ, которым это делается, существенно различается. Платформы обычно имеют возможность кодировать информацию о версии в двоичном файле разделяемой библиотеки, и эта информация иногда используется для определения того, может ли двоичный файл использоваться другим исполняемым файлом или разделяемой библиотекой, которая ссылается на него. Некоторые платформы также имеют соглашения для установки файлов и символических ссылок с различными уровнями номера версии в их именах. В Linux, например, обычный набор файлов и символических ссылок для разделяемой библиотеки может выглядеть следующим образом:

```
libmystuff.so.2.4.3
libmystuff.so.2 --> libmystuff.so.2.4.3
libmystuff.so   --> libmystuff.so.2
```

CMake позаботится о большинстве различий между платформами в отношении обработки версий для разделяемых библиотек. При компоновке цели с разделяемой библиотекой он будет следовать соглашениям платформы при выборе имен файлов или симлинков для компоновки. При сборке разделяемой библиотеки CMake автоматизирует создание полного набора файлов и симлинков, если предоставлена информация о версии.

Информация о версии разделяемой библиотеки определяется целевыми свойствами VERSION и SOVERSION. Интерпретация этих свойств различна на разных платформах, поддерживаемых CMake, но, следуя принципам *семантического версионирования*, эти различия можно обрабатывать достаточно легко. Семантическое версионирование предполагает, что номер версии задается в форме *major.minor.patch*, где каждый компонент версии - целое число. Свойство VERSION будет установлено на полное значение *major.minor.patch*, тогда как SOVERSION будет установлено только на *основную* часть. По мере развития проекта и выпуска релизов семантическое версионирование подразумевает, что сведения о версии должны быть изменены следующим образом:

- Когда происходит несовместимое изменение API, увеличьте *основную* часть версии и сбросьте *минорную* и *патч* части на 0. Это означает, что свойство SOVERSION будет изменяться каждый раз, когда происходит нарушение API, и *только* если происходит нарушение API.
- Когда функциональность добавляется в порядке обратной совместимости, увеличьте *минорную* часть и сбросьте *патч* на 0. *Основная* часть остается неизменной.
- При исправлении ошибки с обратной совместимостью увеличьте значение *патча* и оставьте без изменений *основную* и *второстепенную* части.

Если данные о версии разделяемой библиотеки изменены в соответствии с этими принципами, проблемы несовместимости API во время выполнения будут сведены к минимуму на всех платформах. Рассмотрим следующий пример, который создает набор символических ссылок, показанный ранее для Linux:

```
add_library(mystuff SHARED source1.cpp ...)
set_target_properties(mystuff PROPERTIES
    VERSION   2.4.3
    SOVERSION 2
)
```

На платформах Apple команда otool -L может быть использована для вывода информации о версии, закодированной в полученной разделяемой библиотеке. Для библиотеки общего доступа, созданной в приведенном выше примере, информация о версии будет представлена как *версия совместимости* 2.0.0 и *текущая версия* 2.4.3.

Все, что связано с библиотекой mystuff, будет иметь закодированное имя libmystuff.2.dylib как имя библиотеки, которую следует искать во время выполнения. Платформы Linux демонстрируют аналогичную структуру в своих символических ссылках для общих библиотек, и обычной практикой является использование только *основной* части для имени библиотеки.

В Windows поведение CMake заключается в извлечении *мажорной и минорной* версий из свойства VERSION и кодировании их в DLL в качестве версии образа DLL. В Windows нет понятия soname, поэтому свойство SOVERSION не используется. Тем не менее, следование принципам семантического версионирования по крайней мере гарантирует, что версия DLL может быть использована для определения совместимости библиотеки с двоичными файлами, которые связываются с ней.

Следует отметить, что семантическое версионирование не является строго обязательным для какой-либо платформы. Скорее, оно обеспечивает четко определенную спецификацию, которая вносит некоторую определенность в управление зависимостями между общими библиотеками и использующими их объектами. Так сложилось, что она близко отражает то, как версии библиотек обычно интерпретируются на большинстве платформ на базе Unix, и CMake стремится максимально использовать целевые свойства VERSION и SOVERSION для предоставления общих библиотек, которые следуют традициям родной платформы.

Проекты должны знать, что если задано только одно из целевых свойств VERSION и SOVERSION, то недостающее значение будет считаться таким, как если бы оно имело то же значение, что и предоставленное. Это вряд ли приведет к хорошей работе с версиями, если только для номера версии не используется только одно число (т.е. нет частей minor или patch). Такая нумерация версий может быть уместна в некоторых случаях, но в целом проекты должны стремиться следовать принципам, рассмотренным выше, для более гибкого и надежного поведения во время выполнения.

20.4. Совместимость интерфейсов

Свойства цели VERSION и SOVERSION позволяют задавать версионность API более или менее независимым от платформы образом на уровне операционной системы. CMake также предоставляет другие свойства, которые можно использовать для определения требований к совместимости между целями CMake, когда они связаны друг с другом. Они могут быть использованы для описания и обеспечения соблюдения деталей, которые не может передать только нумерация версий.

Рассмотрим реалистичный пример, когда сетевая библиотека обеспечивает поддержку протокола https:// и других подобных защищенных возможностей только при наличии соответствующего набора инструментов SSL. Другим частям программы может потребоваться корректировка собственной функциональности в зависимости от того, поддерживается или нет SSL, в то время как программа в целом должна быть последовательной в отношении того, можно ли использовать функции SSL. Это можно обеспечить с помощью свойства *совместимости интерфейса*.

Можно определить несколько различных типов свойств совместимости интерфейса, но самым простым является свойство boolean. Основная идея заключается в том, что библиотеки указывают имя свойства, которое они будут использовать для объявления определенного булева состояния, а затем определяют это свойство с соответствующим значением. Если несколько соединяемых библиотек определяют одно и то же имя свойства

для совместимости интерфейса, CMake проверяет, что они указывают одно и то же значение, и выдает ошибку, если они разные. Основной пример выглядит примерно так:

```
add_library(networking net.cpp)
set_target_properties(networking PROPERTIES
    COMPATIBLE_INTERFACE_BOOL SSL_SUPPORT
    INTERFACE_SSL_SUPPORT YES
)

add_library(util util.cpp)
set_target_properties(util PROPERTIES
    COMPATIBLE_INTERFACE_BOOL SSL_SUPPORT
    INTERFACE_SSL_SUPPORT YES
)

add_executable(myApp myapp.cpp)
target_link_libraries(myApp PRIVATE networking util)
target_compile_definitions(myApp PRIVATE
    $<$<BOOL:$<TARGET_PROPERTY:SSL_SUPPORT>>:HAVE_SSL>
)
```

Обе цели библиотеки сообщают, что они определяют совместимость интерфейсов для свойства SSL_SUPPORT. Ожидается, что свойство COMPATIBLE_INTERFACE_BOOL будет содержать список имен, каждое из которых требует, чтобы на этой цели было определено ассоциированное свойство с тем же именем с приставкой INTERFACE_. Когда библиотеки используются вместе в качестве зависимой связи для myApp, CMake проверяет, что обе библиотеки определяют INTERFACE_SSL_SUPPORT с одинаковым значением. Кроме того, CMake автоматически заполнит свойство SSL_SUPPORT цели myApp тем же значением, которое затем может быть использовано как часть выражения генератора и доступно исходному коду myApp в качестве определения компиляции, как показано на рисунке. Это позволяет коду myApp адаптироваться к тому, была ли скомпилирована поддержка SSL в используемых библиотеках или нет. Продолжая пример, вместо того, чтобы myApp просто определял наличие или отсутствие поддержки SSL, он может указать требование, явно определив свойство SSL_SUPPORT, чтобы оно содержало значение, с которым должны быть совместимы библиотеки. В этом случае, вместо автоматического заполнения свойства SSL_SUPPORT в myApp, CMake сравнит значения и убедится, что библиотеки соответствуют указанному требованию.

```
# Require libraries to have SSL support
set_target_properties(myApp PROPERTIES SSL_SUPPORT YES)
```

Приведенные выше примеры несколько надуманны, поскольку те же самые ограничения могли бы быть реализованы другими способами. Реальные преимущества спецификаций совместимости интерфейсов начинают проявляться, когда проект становится сложнее, а его цели распространяются по многим каталогам или приходят из внешних проектов. Совместимость интерфейсов назначается как свойство целей, поэтому их нужно определить только в одном месте, а затем они становятся доступными везде, где цель может быть использована без дополнительных усилий. Цели-потребители не обязаны знать подробности того, как определяется совместимость интерфейсов, только окончательное решение, хранящееся в свойствах INTERFACE_... цели.

CMake также поддерживает совместимость интерфейсов, выраженную в виде строки. Они работают практически так же, как и в случае с булевыми значениями, за исключением того, что именованные свойства должны иметь абсолютно одинаковые значения и могут содержать любое произвольное содержимое.

Предыдущий пример можно изменить, чтобы потребовать от библиотек использовать одну и ту же реализацию SSL, а не просто договориться о том, поддерживают они SSL или нет:

```
add_library(networking net.cpp)
set_target_properties(networking PROPERTIES
    COMPATIBLE_INTERFACE_STRING SSL_IMPL
    INTERFACE_SSL_IMPL OpenSSL
)

add_library(util util.cpp)
set_target_properties(util PROPERTIES
    COMPATIBLE_INTERFACE_STRING SSL_IMPL
    INTERFACE_SSL_IMPL OpenSSL
)

add_executable(myApp myapp.cpp)
target_link_libraries(myApp PRIVATE networking util)
target_compile_definitions(myApp PRIVATE
    SSL_IMPL=$<TARGET_PROPERTY:SSL_IMPL>
)
```

В приведенном выше примере свойство SSL_IMPL используется как строковая интерфейсная совместимость с библиотеками, указывающими, что они используют OpenSSL в качестве реализации SSL. Как и в случае с boolean, цель myApp могла бы определить свое свойство SSL_IMPL для указания требования, а не позволять CMake заполнять его значением из библиотек.

Другой вид совместимости интерфейсов, поддерживаемый CMake, - это числовое значение. Числовая совместимость интерфейсов используется для определения *минимального* или *максимального* значения, определенного для свойства среди набора библиотек, а не для того, чтобы требовать, чтобы свойства имели *одинаковое* значение. Это ключевое различие может быть использовано для того, чтобы позволить цели определить такие вещи, как минимальная версия протокола, которую она может поддерживать, или определить наибольший размер временного буфера, необходимый среди различных библиотек, к которым она подключается.

```
add_library(bigAndFast strategy1.cpp)
set_target_properties(bigAndFast PROPERTIES
    COMPATIBLE_INTERFACE_NUMBER_MIN PROTOCOL_VER
    COMPATIBLE_INTERFACE_NUMBER_MAX TMP_BUFFERS
    INTERFACE_PROTOCOL_VER 3
    INTERFACE_TMP_BUFFERS 200
)

add_library(smallAndSlow strategy2.cpp)
set_target_properties(smallAndSlow PROPERTIES
    COMPATIBLE_INTERFACE_NUMBER_MIN PROTOCOL_VER
    COMPATIBLE_INTERFACE_NUMBER_MAX TMP_BUFFERS
    INTERFACE_PROTOCOL_VER 2
    INTERFACE_TMP_BUFFERS 15
)

add_executable(myApp myapp.cpp)
target_link_libraries(myApp PRIVATE bigAndFast smallAndSlow)
target_compile_definitions(myApp PRIVATE
    MIN_API=$<TARGET_PROPERTY:PROTOCOL_VER>
    TMP_BUFFERS=$<TARGET_PROPERTY:TMP_BUFFERS>
)
```

В приведенном выше примере PROTOCOL_VER определен как *минимальная* числовая совместимость интерфейса, так что

PROTOCOL_VER свойство myApp будет установлено в наименьшее значение, указанное для свойства INTERFACE_PROTOCOL_VER библиотек, с которыми он связан, что в данном случае равно 2. Аналогично, TMP_BUFFERS определяется как *максимальная* числовая совместимость интерфейса, и свойство TMP_BUFFERS myApp получает наибольшее значение среди свойств INTERFACE_TMP_BUFFERS связанных с ним библиотек, что равно 200.

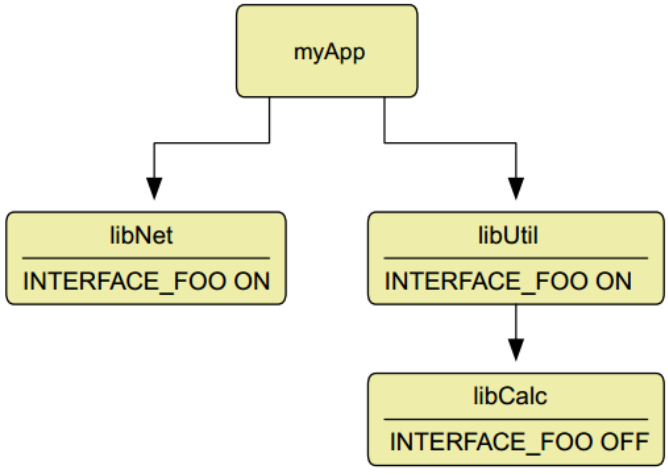
В этот момент естественно было бы подумать об использовании одного и того же свойства для минимальной и максимальной совместимости числового интерфейса, чтобы позволить определять в родителе как наименьшее, так и наибольшее значение. Это невозможно, поскольку CMake не позволяет (и не может) использовать одно и то же свойство более чем для одного типа совместимости интерфейсов. Если бы свойство использовалось для нескольких типов совместимости интерфейсов, CMake было бы невозможно узнать, какой тип следует использовать для вычисления значения, которое будет храниться в свойстве result родителя. Например, если бы PROTOCOL_VER был одновременно минимальной и максимальной совместимостью интерфейса в приведенном выше примере, CMake не смог бы определить, какое значение хранить в свойстве PROTOCOL_VER myApp - минимальное или максимальное? Вместо этого необходимо использовать отдельные свойства для достижения этой цели:

```
add_library(bigAndFast strategy1.cpp)
set_target_properties(bigAndFast PROPERTIES
    COMPATIBLE_INTERFACE_NUMBER_MIN PROTOCOL_VER_MIN
    COMPATIBLE_INTERFACE_NUMBER_MAX PROTOCOL_VER_MAX
    INTERFACE_PROTOCOL_VER_MIN 3
    INTERFACE_PROTOCOL_VER_MAX 3
)

add_library(smallAndSlow strategy2.cpp)
set_target_properties(smallAndSlow PROPERTIES
    COMPATIBLE_INTERFACE_NUMBER_MIN PROTOCOL_VER_MIN
    COMPATIBLE_INTERFACE_NUMBER_MAX PROTOCOL_VER_MAX
    INTERFACE_PROTOCOL_VER_MIN 2
    INTERFACE_PROTOCOL_VER_MAX 2
)

add_executable(myApp myapp.cpp)
target_link_libraries(myApp PRIVATE bigAndFast smallAndSlow)
target_compile_definitions(myApp PRIVATE
    PROTOCOL_VER_MIN=$<TARGET_PROPERTY:PROTOCOL_VER_MIN>
    PROTOCOL_VER_MAX=$<TARGET_PROPERTY:PROTOCOL_VER_MAX>
)
```

В результате приведенного выше примера myApp знает диапазон версий протоколов, которые ему необходимо поддерживать, на основе протоколов, используемых библиотеками, на которые он ссылается.

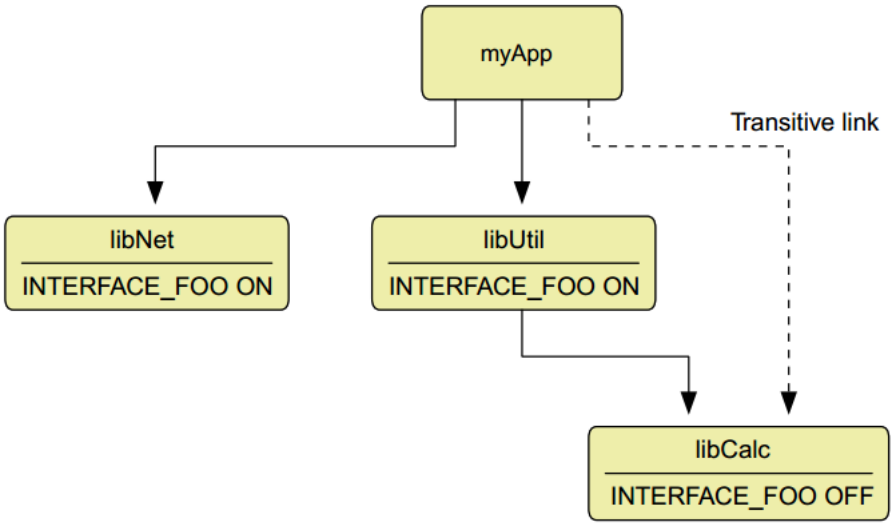
Если одна цель определяет совместимость интерфейса какого-либо конкретного типа, другие цели не обязаны определять ее также. Любая цель, которая не определяет совместимость интерфейса, просто игнорируется для этого конкретного свойства. Это гарантирует, что библиотекам нужно определять только те совместимости интерфейсов, которые имеют к ним отношение.

Когда существует несколько уровней зависимостей библиотечных ссылок, возникают некоторые тонкие сложности в обработке совместимости интерфейсов. Рассмотрим структуру, показанную на следующей диаграмме, которая содержит ряд библиотечных и исполняемых целей и их прямые ссылочные зависимости.



Если все ссылочные зависимости считаются PRIVATE, то только libNet и libUtil являются прямыми ссылочными зависимостями myApp, поэтому только эти две библиотеки должны иметь согласованные значения для своего свойства INTERFACE_FOO. Значение этого свойства в библиотеке libCalc не учитывается, поскольку она не является прямой зависимостью myApp. Более того, единственной прямой зависимостью libUtil является libCalc, поэтому свойство INTERFACE_FOO библиотеки libCalc не имеет другой библиотеки, с которой оно должно быть согласовано. Даже если и libUtil, и libCalc определяют совместимость интерфейса для одного и того же имени свойства, поскольку они оба не являются *прямыми* ссылочными зависимостями общей цели, от них не требуется иметь совместимые значения.

Теперь рассмотрим ситуацию, когда libCalc является зависимостью PUBLIC link от libUtil. В этом случае окончательные отношения связывания будут выглядеть следующим образом:



Когда libCalc является ПУБЛИЧНОЙ зависимостью от libUtil, все, что ссылается на libUtil, будет ссылаться и на libCalc. Таким образом, libCalc становится прямой зависимостью myApp и поэтому участвует в проверке совместимости интерфейсов с libNet и libUtil. Это означает, что при определении совместимости интерфейсов нужно быть очень внимательным, чтобы убедиться, что они точно выражают правильные вещи, поскольку их охват может распространяться на цели, выходящие за рамки того, что может показаться очевидным, когда задействованы ПУБЛИЧНЫЕ связи.

20.5. Видимость символов

Упрощенно библиотеку можно представить как контейнер скомпилированного исходного кода, предоставляющий различные функции и глобальные данные, которые может вызывать или использовать другой код. Для статических библиотек контейнер на самом деле является просто коллекцией объектных файлов, а инструмент, собирающий их вместе, иногда называют архиватором или библиотекарем. Общие библиотеки, с другой стороны, создаются компоновщиком, который обрабатывает объектный код, архивы и т.д. и решает, что включить в конечный двоичный файл общей библиотеки. Некоторые функции и глобальные данные могут быть скрытыми, то есть они помечены как допустимые для использования компоновщиком для разрешения внутренних зависимостей кода, но код вне разделяемой библиотеки не может вызывать или использовать их. Другие символы экспортируются, поэтому код как внутри, так и вне разделяемой библиотеки может получить к ним доступ. Это называется *видимостью* символа.

Компиляторы по-разному указывают видимость символов, и они также по-разному ведут себя по умолчанию. Некоторые делают все символы видимыми по умолчанию, в то время как другие скрывают символы по умолчанию. Компиляторы также различаются по синтаксису, используемому для обозначения отдельных функций, классов и данных как видимых или нет, что еще больше усложняет написание переносимых общих библиотек. Чтобы избежать некоторых сложностей, некоторые разработчики предпочитают просто сделать все символы видимыми и избежать необходимости явно помечать какие-либо символы для экспорта. Хотя поначалу это может показаться выигрышем, это имеет ряд отрицательных сторон:

- Это эквивалентно утверждению, что каждая функция, класс, тип, глобальная переменная и т.д. свободно доступны для использования кем угодно. Это редко желательно, но может быть приемлемо, если проект полагается на свою документацию для определения символов, которые должны считаться общедоступными.
- Делая все символы видимыми, нельзя помешать потребляющему коду использовать то, что не следует. Другой код, связывающийся с библиотекой, может полагаться на какой-то внутренний символ, что затрудняет изменение реализации или внутренней структуры разделяемой библиотеки без разрушения потребляющих проектов.
- Когда все символы должны рассматриваться как видимые, компоновщик не может знать, будет ли каждый символ использоваться чем-либо, поэтому он должен включить их все в конечную разделяемую библиотеку. Когда экспортируется только подмножество символов, компоновщик имеет возможность определить код, который никогда не будет использован видимыми символами, и, следовательно, отбросить его, что часто приводит к созданию двоичного файла гораздо меньшего размера, который впоследствии может загружаться быстрее во время выполнения.
- Такие языки, как C++, поддерживающие шаблоны, потенциально могут определять огромное количество символов. Если все символы видны по умолчанию, это может привести к тому, что таблица символов разделяемой библиотеки станет очень большой. В крайних случаях это может оказать заметное влияние на производительность запуска во время выполнения.
- Функции, используемые во внутренней реализации библиотеки, могут использовать имена, раскрывающие подробности о том, что делает библиотека или как она это делает. Это может быть связано с безопасностью в некоторых контекстах, или это может раскрыть коммерческую IP, которая не должна быть видна тем, кто получает библиотеку.

Вышеизложенные пункты подчеркивают, что видимость символов - это в равной степени и обеспечение публично-частного характера API библиотеки, и низкоуровневая механика производительности общей библиотеки и размера пакета. Очевидно, что экспорт только тех символов, которые должны считаться общедоступными, имеет свои преимущества, но то, как этого добиться, часто является существенным препятствием для многоплатформенных проектов в зависимости от компилятора и платформы. CMake значительно упрощает этот процесс, абстрагируясь от этих различий за несколькими свойствами, переменными и вспомогательным модулем.

20.5.1. Указание видимости по умолчанию

По умолчанию компиляторы Visual Studio считают все символы скрытыми, если они явно не экспортированы. Другие компиляторы, такие как GCC и Clang, наоборот, делают все символы видимыми по умолчанию и скрывают их только в том случае, если это явно указано. Если проект хочет иметь одинаковую видимость символов по умолчанию для всех компиляторов и платформ, необходимо выбрать один из этих двух подходов, но, надеюсь, что недостатки, отмеченные в предыдущем разделе, являются убедительным аргументом в пользу того, что символы должны быть скрыты по умолчанию.

Первым шагом к обеспечению скрытой видимости по умолчанию является определение набора свойств <LANG>_VISIBILITY_PRESET для цели разделяемой библиотеки. Для двух наиболее распространенных языков, в которых используется эта функциональность, свойства называются C_VISIBILITY_PRESET и CXX_VISIBILITY_PRESET для C и C++ соответственно. Значение, присваиваемое этому свойству, должно быть hidden, что изменяет видимость по умолчанию на скрытие всех символов. Другие поддерживаемые значения включают default, protected и internal, но они менее полезны для кроссплатформенных проектов. Они либо указывают на то, что уже является поведением по умолчанию, либо являются вариантами hidden с более специализированными значениями в некоторых контекстах.

Второй шаг - указать, что встроенные функции также должны быть скрыты по умолчанию. Для кода на C++, активно использующего шаблоны, это может существенно уменьшить размер конечного двоичного файла разделяемой библиотеки. Это поведение контролируется целевым свойством VISIBILITY_INLINES_HIDDEN и применимо ко всем языкам. Оно должно иметь булево значение TRUE, чтобы скрывать встроенные символы по умолчанию.

Оба свойства <LANG>_VISIBILITY_PRESET и VISIBILITY_INLINES_HIDDEN могут быть указаны для каждой цели разделяемой библиотеки, или же соответствующими переменными CMake может быть установлено значение по умолчанию. При создании цели ее свойство <LANG>_VISIBILITY_PRESET инициализируется значением CMake-переменной CMAKE_<LANG>_VISIBILITY_PRESET, а свойство VISIBILITY_INLINES_HIDDEN - переменной CMAKE_VISIBILITY_INLINES_HIDDEN. Это обычно удобнее, чем устанавливать свойства для каждой цели отдельно.

Для проектов, желающих сделать все символы видимыми по умолчанию на всех платформах, достаточно изменить поведение компиляторов Visual Studio по умолчанию. Начиная с версии 3.4, CMake предоставляет целевое свойство WINDOWS_EXPORT_ALL_SYMBOLS, которое обеспечивает такое поведение, но с оговорками. Определение этого свойства в значение true заставит CMake записать .def файл, содержащий все символы из всех объектных файлов, использованных для создания разделяемой библиотеки, и передать этот .def файл компоновщику. Это довольно грубый метод, который не позволяет исходному коду выборочно скрыть какие-либо символы, поэтому его следует использовать только в тех случаях, когда *все* символы должны быть видны. Это свойство цели инициализируется переменной CMAKE_WINDOWS_EXPORT_ALL_SYMBOLS CMake при создании цели разделяемой библиотеки.

20.5.2. Указание видимости отдельных символов

Большинство распространенных компиляторов поддерживают задание видимости отдельных символов, но способ, которым они это делают, различен. В общем случае Visual Studio использует один метод, а большинство других компиляторов следуют методу, используемому GCC. Эти два метода имеют схожую структуру, но используют разные ключевые слова. Это означает, что исходный код для таких языков, как C, C++ и их производных, может использовать общее определение препроцессора для контроля видимости, а проекты могут поручить CMake предоставить соответствующее определение.

Существует три основных случая, когда можно указать видимость символов: классы, функции и переменные. В следующем примере, который содержит объявления для каждого из этих трех случаев, обратите внимание на положение MYTOOLS_EXPORT:

```
class MYTOOLS_EXPORT SomeClass {...};   // Export non-private members of a class
MYTOOLS_EXPORT void someFunction();     // Make a free function visible
MYTOOLS_EXPORT extern int myGlobalVar;  // Make a global variable visible
```

При сборке разделяемой библиотеки, содержащей вышеперечисленные реализации, MYTOOLS_EXPORT должен быть заменен соответствующими ключевыми словами, указывающими, что символ должен быть *экспортирован* для использования другими библиотеками и исполняемыми программами. С другой стороны, если те же декларации читаются кодом, принадлежащим другой цели вне разделяемой библиотеки, то MYTOOLS_EXPORT должен быть заменен соответствующими ключевыми словами, указывающими, что символ должен быть *импортирован*. В Windows эти ключевые слова имеют форму __declspec(...), в то время как GCC и совместимые компиляторы используют __attribute__(...).

Придумать правильное содержимое для MYTOOLS_EXPORT для всех компиляторов и для обоих случаев экспорта и импорта может быть довольно сложно. Добавьте сюда то, что разработчики могут выбрать создание библиотеки как разделяемой или статической, и сложность возрастает. К счастью, CMake предоставляет модуль GenerateExportHeader, который очень удобно обрабатывает все эти детали. Этот модуль обеспечивает следующую функцию:

```
generate_export_header(target
    [BASE_NAME baseName]
    [EXPORT_FILE_NAME exportFileName]
    [EXPORT_MACRO_NAME exportMacroName]
    [DEPRECATED_MACRO_NAME deprecatedMacroName]
    [NO_EXPORT_MACRO_NAME noExportMacroName]
    [STATIC_DEFINE staticDefine]
    [NO_DEPRECATED_MACRO_NAME noDeprecatedMacroName]
    [DEFINE_NO_DEPRECATED]
    [PREFIX_NAME prefix]
    [CUSTOM_CONTENT_FROM_VARIABLE var]
)
```

Как правило, ни один из необязательных аргументов не нужен, и указывается только целевое имя разделяемой библиотеки. CMake записывает заголовочный файл в текущий двоичный каталог, используя имя цели в нижнем регистре с добавлением _export.h в качестве имени заголовочного файла. Заголовок предоставляет определение для экспорта символов с аналогично структурированным именем, на этот раз с использованием целевого имени в верхнем регистре с добавлением _EXPORT. Ниже показано типичное использование:

CMakeLists.txt

```
# Hide things by default
set(CMAKE_CXX_VISIBILITY_PRESET     hidden)
set(CMAKE_VISIBILITY_INLINES_HIDDEN YES)

# NOTE: myTools.cpp must #include myTools.h
add_library(myTools myTools.cpp)
target_include_directories(myTools PUBLIC
    "${CMAKE_CURRENT_BINARY_DIR}"
)

# Write out mytools_export.h to the current binary directory
include(GenerateExportHeader)
generate_export_header(myTools)
```

myTools.h

```
#include "mytools_export.h"

class MYTOOLS_EXPORT SomeClass
{
    // ...
};
MYTOOLS_EXPORT void someFunction();
MYTOOLS_EXPORT extern int myGlobalVar;
```

Текущий двоичный каталог не является частью пути поиска заголовков по умолчанию, поэтому его необходимо добавить в качестве ПУБЛИЧНОГО пути поиска для библиотеки, чтобы заголовок mytools_export.h мог быть найден как собственным исходным кодом библиотеки, так и любым другим кодом целей, связывающихся с разделяемой библиотекой.

Если использование целевого имени как части имени заголовочного файла или определения препроцессора нежелательно, можно использовать опцию BASE_NAME в качестве альтернативы. Он преобразуется таким же образом, преобразуется в нижний регистр с добавлением _export.h для имени файла и в верхний регистр с добавлением _EXPORT для определения препроцессора.

CMakeLists.txt

```
include(GenerateExportHeader)
generate_export_header(myTools BASE_NAME fooBar)
```

myTools.h

```
#include "foobar_export.h"

class FOOBAR_EXPORT SomeClass
{
    // ...
};
FOOBAR_EXPORT void someFunction();
FOOBAR_EXPORT extern int myGlobalVar;
```

Если для файла и определения препроцессора должно использоваться другое имя, то вместо использования BASE_NAME можно указать опции EXPORT_FILE_NAME и EXPORT_MACRO_NAME. В отличие от BASE_NAME, имена, предоставляемые этими двумя опциями, используются без каких-либо изменений.

CMakeLists.txt

```
include(GenerateExportHeader)
generate_export_header(myTools
    EXPORT_FILE_NAME  export_myTools.h
    EXPORT_MACRO_NAME API_MYTOOLS
)
```

myTools.h

```
#include "export_myTools.h"

class API_MYTOOLS SomeClass
{
    // ...
};
API_MYTOOLS void someFunction();
API_MYTOOLS extern int myGlobalVar;
```

Функция generate_export_header() предоставляет не только это одно определение препроцессора, но и другие определения препроцессора, которые можно использовать для пометки символов как устаревших или для явного указания, что символ никогда не должен экспортироваться. Последнее может быть полезно для предотвращения экспорта части класса, которая в противном случае экспортируется, например, публичной функции-члена, предназначенной для внутреннего использования в общей библиотеке, но не для кода за ее пределами. По умолчанию имя этого определения препроцессора состоит из целевого имени (или BASE_NAME, если оно конкретизировано) с добавлением _NO_EXPORT, но при желании можно указать альтернативное имя с помощью опции NO_EXPORT_MACRO_NAME.

CMakeLists.txt

```
include(GenerateExportHeader)
generate_export_header(myTools
    NO_EXPORT_MACRO_NAME REALLY_PRIVATE
)
```

myTools.h

```
#include "mytools_export.h"

class MYTOOLS_EXPORT SomeClass
{
public:
    REALLY_PRIVATE void doInternalThings();
    // ...
};
```

Поддержка обесценивания функции работает аналогичным образом, предоставляя определение препроцессора с именем цели (или BASE_NAME) в верхнем регистре, за которым следует _DEPRECATED, или позволяя указать пользовательское имя с помощью опции DEPRECATED_MACRO_NAME. Также можно указать опцию DEFINE_NO_DEPRECATED, в результате чего будет предоставлено дополнительное определение препроцессора с именем, состоящим из обычного заглавного имени target или BASE_NAME, за которым следует _NO_DEPRECATED. Как и другие определения препроцессора, это имя может быть переопределено с помощью опции NO_DEPRECATED_MACRO_NAME. В некоторых компиляторах символы, помеченные как устаревшие, могут приводить к предупреждениям во время компиляции, которые обращают внимание на их использование. Это может быть полезным механизмом, побуждающим разработчиков обновить свой код, чтобы больше не использовать устаревшие символы. Ниже показано, как можно использовать механизмы устаревания.

CMakeLists.txt

```
option(OMIT_DEPRECATED "Leave out deprecated parts of myTools")
if(OMIT_DEPRECATED)
    set(deprecatedOption "DEFINE_NO_DEPRECATED")
else()
    unset(deprecatedOption)
endif()
include(GenerateExportHeader)
generate_export_header(myTools
    NO_DEPRECATED_MACRO_NAME OMIT_DEPRECATED
    ${deprecatedOption}
)
```

myTools.h

```
#include "mytools_export.h"

class MYTOOLS_EXPORT SomeClass
{
public:
#ifndef OMIT_DEPRECATED
    MYTOOLS_DEPRECATED void oldImpl();
#endif
    // ...
};
```

myTools.cpp

```
#include "myTools.h"

#ifndef OMIT_DEPRECATED
void SomeClass::oldImpl() { ... }
#endif
```

Приведенный выше пример предоставляет переменную кэша CMake для определения того, компилировать или нет устаревшие элементы. Разработчик имеет возможность сделать этот выбор без редактирования каких-либо файлов, поэтому проверить поведение с устаревшей частью API или без нее очень просто. Это может быть особенно полезно, если сборки непрерывной интеграции были настроены на тестирование как с устаревшими частями библиотеки, так и без них. Это также может быть полезно в ситуациях, когда проект используется в качестве зависимости от другого проекта, позволяя разработчикам этого проекта проверить, использует ли их код устаревшие символы или нет, просто изменив переменную кэша CMake.

Менее распространенный, но тем не менее важный случай также заслуживает особого упоминания. Некоторые проекты могут захотеть собрать как разделяемую, так и статическую версии одной и той же библиотеки. В этом случае один и тот же набор исходных текстов должен позволять включать экспорт символов для сборки разделяемой библиотеки, но отключать его для сборки статической библиотеки (см. также следующий раздел о том, почему это не всегда так). Когда в одной сборке требуются обе формы библиотеки, они должны быть разными целями сборки, но функция generate_export_header() пишет заголовок, тесно связанный с одной целью. Чтобы поддержать этот сценарий, генерируемый заголовок включает логику проверки существования

еще одного определения препроцессора перед заполнением определения экспорта. Имя этого специального определения снова следует обычной схеме, на этот раз это заглавная цель или BASE_NAME, за которой следует _STATIC_DEFINE, или пользовательское имя, предоставляемое опцией STATIC_DEFINE. Когда определено это специальное определение препроцессора, определение экспорта принудительно расширяется до ничего, что обычно необходимо, когда цель собирается как статическая библиотека. Без специального определения препроцессора экспортное определение имеет обычное содержимое и работает так, как ожидается при сборке целевой разделяемой библиотеки.

Когда для одного и того же набора исходных файлов собираются как разделяемые, так и статические библиотеки, функции generate_export_header() должна быть присвоена цель, соответствующая разделяемой библиотеке. Тогда специальное определение препроцессора будет установлено только для статической библиотеки. Опция BASE_NAME также обычно используется для того, чтобы различные символы были интуитивно понятны для любой формы библиотеки, а не были специфичны только для разделяемой библиотеки. Ниже показана структура, необходимая для достижения желаемого результата:

```
# Same source list, different library types
add_library(myShared SHARED ${mySources})
add_library(myStatic STATIC ${mySources})

# Shared target used for generating export header
# with the name myTools_export.h, which will be suitable
# for both the shared and static targets
include(GenerateExportHeader)
generate_export_header(myShared BASE_NAME myTools)

# Static target needs special preprocessor define
# to prevent symbol import/export keywords being added
target_compile_definitions(myStatic PRIVATE
    MYTOOLS_STATIC_DEFINE
)
```

Как видно из предыдущего обсуждения, функция generate_export_header() определяет ряд различных определений препроцессора, и существует возможность того, что различные цели могут случайно попытаться использовать одинаковые имена по крайней мере для некоторых из них. Чтобы помочь уменьшить столкновения имен, опция PREFIX_NAME позволяет указать дополнительную строку, которая будет добавлена к именам каждого определения препроцессора. При использовании этой опции обычно указывается что-то, относящееся к проекту в целом, что позволяет поместить все имена препроцессоров проекта в нечто вроде пространства имен конкретного проекта.

Последняя опция, которая еще не обсуждалась, это CUSTOM_CONTENT_FROM_VARIABLE, которая была добавлена только в CMake 3.7. Эта опция позволяет вставлять произвольное содержимое в сгенерированный заголовок ближе к концу, после добавления всей разнообразной логики препроцессора. При использовании этой опции необходимо указать имя переменной, содержимое которой должно быть инжектировано, а не само содержимое.

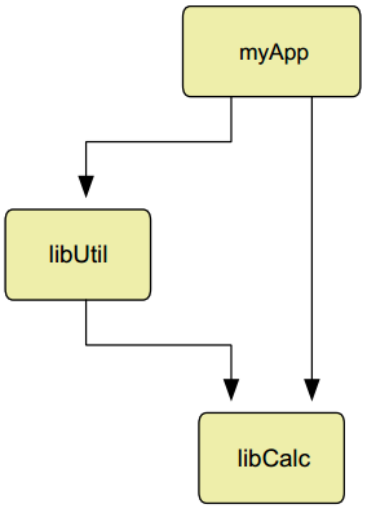
```
string(TIMESTAMP now)
set(customContents "/* Generated: ${now} */")
generate_export_header(myTools
    CUSTOM_CONTENT_FROM_VARIABLE customContents
)
```

20.6. Смешивание статических и общих библиотек

Когда проект собирает все свои библиотеки как статические, сборка может показаться более щадящей в отношении зависимостей библиотечных связей. Проект может не указать, что одна цель требует другую, но когда различные статические библиотеки соединяются в конечный исполняемый файл, отсутствующие зависимости библиотек удовлетворяются, потому что они явно перечислены для исполняемого файла в нужном порядке. Сборка проходит успешно, но, вероятно, только после периода проб и ошибок, когда компоновщик жаловался на недостающие символы, добавлял новые недостающие библиотеки или изменял порядок существующих и т. д.

Этот сценарий приводит к успеху скорее по счастливой случайности, чем благодаря продуманному дизайну, но он удивительно распространен, особенно в проектах, в которых определено много небольших библиотек. Если хотя бы для некоторых статических библиотек указаны ссылочные зависимости, CMake автоматически обрабатывает транзитивное связывание этих зависимостей, поэтому даже если природа зависимости PRIVATE/PUBLIC указана неверно, со статической библиотекой она всегда рассматривается как PUBLIC, и это иногда заставляет сборки работать, даже если ссылочная зависимость описана неточно.

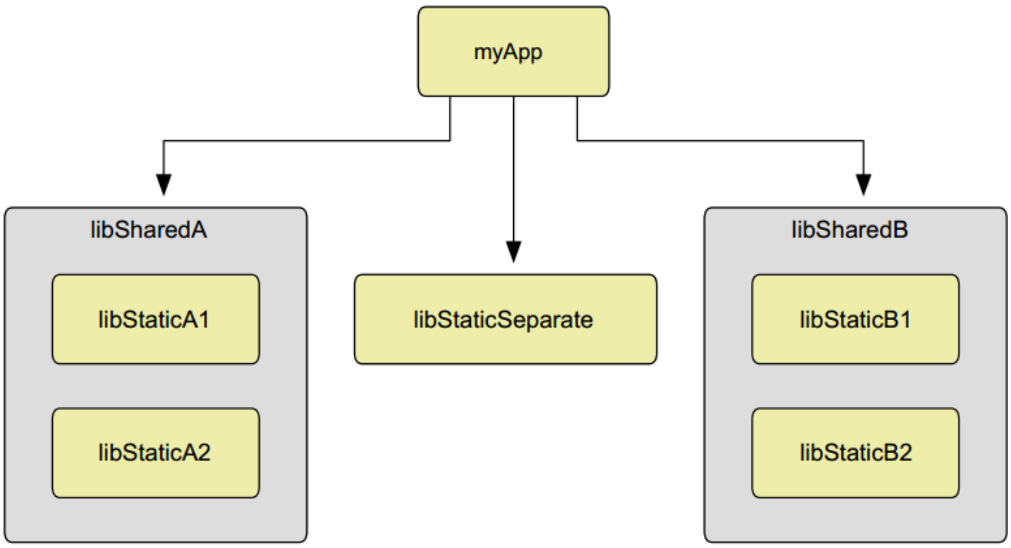
Когда цели библиотеки определены как смесь общих и статических, корректность ссылочных зависимостей становится намного важнее. Рассмотрим следующий набор целей:



Если libUtil и libCalc являются статическими библиотеками, то приведенные выше отношения зависимости ссылок безопасны. Если libUtil является разделяемой библиотекой, то приведенная выше схема зависимости ссылок открывает возможность дублирования данных, которые должны иметь только один экземпляр во всем приложении. Если libCalc определяет глобальные данные, как это обычно бывает для синглтона или статических данных класса, то может оказаться возможным, что и myApp, и libUtil будут иметь свои собственные отдельные экземпляры этих данных. Это становится возможным, поскольку и myApp, и libUtil требуют от

компоновщика разрешения символов, поэтому оба вызова могут решить, что глобальные данные необходимы, и создать их внутренний экземпляр в исполняемом файле или разделяемой библиотеке. Если глобальные данные не являются экспортируемым символом, компоновщик не увидит экземпляр, уже созданный в libUtil, когда перейдет к компоновке myApp. В итоге в myApp создается второй экземпляр, что почти наверняка вызовет трудноотслеживаемые проблемы во время выполнения. Типичным проявлением этого является волшебное изменение значения переменной при вызове функции из одной исполняемой или разделяемой библиотеки в другую разделяемую библиотеку.

Ситуации, подобные приведенному выше сценарию, могут возникать в разных формах, но в каждом случае действует один и тот же основополагающий принцип. Если статическая библиотека подключена к разделяемой библиотеке, эта разделяемая библиотека не должна объединяться с любой другой библиотекой или исполняемым файлом, который также подключается к этой статической библиотеке. В идеале, если смешиваются разделяемые и статические библиотеки, то статические библиотеки должны быть связаны только с одной разделяемой библиотекой, а все, что требует чего-то от одной из этих статических библиотек, должно связываться с разделяемой библиотекой. Общая библиотека, по сути, имеет свой собственный API, а статические библиотеки могут вносить в него свой вклад.



Использование статических библиотек для создания содержимого общих библиотек подобным образом создает свой собственный набор проблем, когда дело доходит до видимости символов. Как правило, код из статических библиотек не экспортируется, поэтому он не будет отображаться в экспортируемых символах разделяемой библиотеки. Один из способов решения этой проблемы - использовать функцию generate_export_header() в общей библиотеке как обычно, а затем заставить статическую библиотеку повторно использовать те же определения экспорта. Ключ к тому, чтобы это сработало, заключается в том, чтобы статическая библиотека имела определение компиляции для имени цели разделяемой библиотеки с добавлением _EXPORTS, именно так генерируемый заголовок определяет, собирается ли код как часть разделяемой библиотеки или нет.

CMakeLists.txt

```
add_library(myShared SHARED shared.cpp)
add_library(myStatic STATIC static.cpp)

include(GenerateExportHeader)
generate_export_header(myShared BASE_NAME mine)

target_link_libraries(myShared PRIVATE myStatic)
target_include_directories(myShared PUBLIC ${CMAKE_CURRENT_BINARY_DIR})
target_include_directories(myStatic PUBLIC ${CMAKE_CURRENT_BINARY_DIR})

# This makes the static library code appear to be part of the shared library
# library as far as the generated export header is concerned
target_compile_definitions(myStatic PRIVATE myShared_EXPORTS)
```

shared.h

```
#include "mine_export.h"

MINE_EXPORT void sharedFunc();
```

static.h

```
#include "mine_export.h"

MINE_EXPORT void staticFunc();
```

Другой фактор, который необходимо учитывать, - это то, будет ли компоновщик отбрасывать код или данные, определенные в статической библиотеке, когда дело дойдет до компоновки разделяемой библиотеки. Если он определит, что ничто не использует определенный символ, компоновщик может отбросить его в качестве оптимизации. Чтобы этого не произошло, могут потребоваться специальные меры. Один из вариантов - заставить разделяемую библиотеку явно использовать каждый символ, который должен быть сохранен из разделяемых библиотек. Преимущество этого способа в том, что он будет работать для всех компиляторов и компоновщиков, но он может оказаться невыполнимым для нетривиальных проектов. Альтернативный вариант требует добавления специфических для компоновщика флагов, таких как --whole-archive и --no-whole-archive для компоновщика ld в Unix-системах или /WHOLEARCHIVE в Visual Studio, но такая функциональность может быть доступна не во всех компоновщиках. Если обеспечение использования в разделяемой библиотеке всех символов, экспортируемых ее статическими библиотеками, нецелесообразно, возможно, стоит подумать о превращении этих статических библиотек в разделяемые.

Если разделяемая библиотека связывается со статическими библиотеками только частным образом (то есть ни один из символов статических библиотек не должен быть экспортирован), то ситуация значительно упрощается. На некоторых платформах не требуется никаких дополнительных действий, кроме простого связывания разделяемой библиотеки со статическими библиотеками. На других платформах могут возникнуть одна или две незначительные проблемы, которые необходимо решить. Например, на многих 64-разрядных Unix-системах код должен быть скомпилирован как *позиционно независимый,* если он должен попасть в разделяемую библиотеку, тогда как для статических библиотек такого требования нет. Однако если разделяемая библиотека ссылается на статическую библиотеку, то статическая библиотека должна быть собрана как позиционно независимая.

CMake предоставляет свойство цели POSITION_INDEPENDENT_CODE как способ прозрачной обработки позиционно-независимого поведения на тех платформах, которые этого требуют. Если свойство установлено в true, это приводит к тому, что код этой цели будет собираться как позиционно независимый. По умолчанию свойство включено для библиотек SHARED и MODULE и выключено для всех остальных типов целей. Значение по умолчанию можно отменить, задав переменную CMAKE_POSITION_INDEPENDENT_CODE, в этом случае она будет использоваться для инициализации свойства цели POSITION_INDEPENDENT_CODE при создании цели.

```
add_library(myShared SHARED shared.cpp)
add_library(myStatic STATIC static.cpp)
target_link_libraries(myShared PRIVATE myStatic)

set_target_properties(myStatic PROPERTIES
    POSITION_INDEPENDENT_CODE ON
)

set(CMAKE_POSITION_INDEPENDENT_CODE ON)
add_library(myOtherStatic STATIC other.cpp)
target_link_libraries(myShared PRIVATE myOtherStatic)
```

20.7. Рекомендуемые практики

Используйте библиотеки MODULE для дополнительных модулей, загружаемых по требованию, и библиотеки SHARED для компоновки с ними. Используйте разделяемые библиотеки, когда символы, которые будут открыты для потребителей библиотеки, должны строго контролироваться, либо для целей API, либо для сокрытия конфиденциальных деталей реализации. Если вы планируете поставлять библиотеку как часть пакета релиза, то в большинстве случаев предпочтительнее использовать разделяемые библиотеки, чем статические.

Если цель использует что-то из библиотеки, она всегда должна ссылаться непосредственно на эту библиотеку. Даже если библиотека уже является зависимостью от ссылки на что-то другое, на что ссылается цель, не полагайтесь на косвенную зависимость от ссылки для чего-то, что цель использует напрямую. Если эта другая цель изменит свою реализацию и перестанет ссылаться на библиотеку, основная цель перестанет собираться. Кроме того, выражайте правильный тип зависимости ссылок: PRIVATE, PUBLIC или INTERFACE. Это гарантирует, что CMake правильно обработает переходные ссылочные зависимости как для разделяемых, так и для статических библиотек. Указание всех прямых зависимостей с правильным уровнем видимости необходимо для того, чтобы CMake создал надежную командную строку компоновщика с правильным упорядочиванием библиотек.

Использование правильной видимости ссылок имеет дополнительное преимущество: потребляющим целям не нужно знать обо всех различных зависимостях библиотек, используемых внутри программы, им достаточно ссылаться на библиотеку и позволить этой библиотеке определить свои собственные зависимости. Затем CMake позаботится о том, чтобы все необходимые библиотеки были указаны в правильном порядке в командной строке конечного компоновщика. Не поддавайтесь искушению просто сделать все зависимости ссылок PUBLIC, так как это расширяет видимость закрытых библиотек туда, где это может быть нежелательно. Это становится особенно важным при упаковке проекта для выпуска или распространения.

Рассмотрите возможность использования стратегии версионирования библиотек как можно раньше. После выпуска библиотеки в свободную продажу номер версии имеет весьма специфическое значение в отношении бинарной совместимости. Используйте целевые свойства VERSION и SOVERSION для указания версии библиотеки, даже если изначально, на ранних этапах жизни проекта, они установлены на некоторые базовые значения. При отсутствии какой-либо другой стратегии, одним из разумных вариантов является начало

нумерации версий с 0.1.0, поскольку люди склонны интерпретировать 0.0.0 как значение по умолчанию или ошибочно не установленную версию, в то время как 1.0.0 иногда воспринимается как первый публичный релиз. Рассмотрите возможность принятия семантической версионности для последующей обработки изменений версий. Помните также, что изменения в версиях библиотек могут оказывать удивительно сильное влияние на такие вещи, как процессы выпуска, упаковки и т.д., и разработчикам необходимо время, чтобы узнать о последствиях номеров версий для общих библиотек задолго до того, как эти библиотеки будут выпущены публично. Подумайте также, должны ли версия проекта и версия библиотеки иметь какую-либо связь друг с другом или нет. Изменить такую связь после выпуска первого релиза может быть очень сложно, поэтому опасайтесь связывать их, если они не имеют тесной связи (проект, поставляющий согласованный набор библиотек в виде SDK, является одним из примеров тесной связи).

Некоторые проекты могут опционально предоставлять определенные функциональные возможности при наличии определенного вспомогательного инструментария, библиотеки и т.д. Чтобы позволить другим частям сборки или, более того, другим потребляющим проектам обнаружить или проверить совместимость с этой дополнительной функциональностью или функцией, можно предоставить детали совместимости интерфейса. Подумайте, должна ли рассматриваемая функция иметь видимость за пределами библиотеки, например, чтобы потребляющие цели могли определить, поддерживается ли функция или нет, или подтвердить, что выбранная реализация предоставляет все необходимые возможности. Также подумайте, приносит ли дополнительная сложность определения и использования совместимости интерфейсов достаточные преимущества, чтобы сделать это целесообразным, поскольку чем глубже становится иерархия зависимостей библиотеки, тем сложнее эффективно использовать совместимость интерфейсов.

Уделяйте внимание видимости символов как можно раньше в проекте, так как потом может быть очень трудно вернуться и изменить проект, добавив в него детали видимости символов. При создании библиотек всегда думайте о том, должен ли конкретный класс, функция или переменная быть доступной для чего-либо за пределами библиотеки. Считайте, что все, что имеет внешнюю видимость, очень трудно изменить, в то время как внутренние вещи могут быть более свободно изменены между выпусками по мере необходимости. Используйте скрытую видимость по умолчанию и явно помечайте каждую отдельную сущность для экспорта, в идеале с помощью макросов, предоставляемых функцией generate_export_header(), чтобы CMake обрабатывал различные различия платформ от имени проекта. Также рассмотрите возможность использования макросов deprecation, предоставляемых этой функцией, чтобы четко определить те части API библиотеки, которые были устаревшими и которые могут быть удалены в будущей версии.

Будьте особенно осторожны при смешивании разделяемых и статических библиотек. По возможности предпочитайте использовать одну или другую, а не обе, так как это позволяет избежать некоторых трудностей, связанных с согласованностью настроек сборки и контролем видимости символов. Там, где имеет смысл смешивать оба типа библиотек, старайтесь, чтобы статические библиотеки были связаны только с одной общей библиотекой, и никакие другие цели не связывались с этими статическими библиотеками. Рассматривайте статические библиотеки как подгруппы в общей библиотеке, при этом внешние цели должны ссылаться только на общую библиотеку. Еще лучше, если код из статических библиотек подтягивается непосредственно в общую библиотеку, избавляясь от статических библиотек вообще. Техники, представленные в разделе 28.5.1, "Исходники цели", демонстрируют, как постепенно добавлять исходники к существующей цели, позволяя удобно накапливать исходники цели в подкаталогах.

Глава 21. Цепочки инструментов и кросс-компиляция

Рассматривая процесс создания программного обеспечения и соответствующие инструменты, разработчики обычно думают о компиляторе и компоновщике. Хотя это основные инструменты, с которыми сталкиваются разработчики, существует ряд других инструментов, библиотек и вспомогательных файлов, которые также вносят свой вклад в процесс. Говоря в общем, этот более широкий набор инструментов и других файлов называется *цепочкой инструментов*.

Для настольных или традиционных серверных приложений обычно нет необходимости слишком глубоко задумываться о цепочке инструментов. В большинстве случаев решение о том, какой выпуск инструментальной цепочки преобладающей платформы использовать, не представляет особой сложности. CMake обычно находит инструментальную цепочку без особой помощи, и разработчик может приступать к написанию программного обеспечения. Однако для разработки мобильных или встроенных устройств ситуация совершенно иная. Инструментальная цепочка обычно должна быть определенным образом задана разработчиком. Это может быть простое указание имени другой целевой системы, а может быть и сложное указание путей к отдельным инструментам и целевой корневой файловой системе. Также могут потребоваться специальные флаги, чтобы инструменты создавали двоичные файлы, которые будут поддерживать нужные чипсеты, иметь необходимые характеристики производительности и так далее.

После выбора цепочки инструментов CMake выполняет довольно много внутренних операций по тестированию цепочки инструментов для определения поддерживаемых ею функций, установки различных свойств и переменных и т. д. Это происходит даже при традиционной сборке, когда используется цепочка инструментов по умолчанию, а не только при сборке с кросс-компиляцией. Результаты этих тестов можно увидеть в выводе CMake при первом запуске для данного каталога сборки. Пример для macOS выглядит следующим образом (пути компиляторов C и CXX были сокращены для краткости):

```
-- The C compiler identification is AppleClang 9.0.0.9000039
-- The CXX compiler identification is AppleClang 9.0.0.9000039
-- Check for working C compiler: /Applications/Xcode.app/.../cc
-- Check for working C compiler: /Applications/Xcode.app/.../cc -- works
-- Detecting C compiler ABI info
-- Detecting C compiler ABI info - done
-- Detecting C compile features
-- Detecting C compile features - done
-- Check for working CXX compiler: /Applications/Xcode.app/.../c++
-- Check for working CXX compiler: /Applications/Xcode.app/.../c++ -- works
-- Detecting CXX compiler ABI info
-- Detecting CXX compiler ABI info - done
-- Detecting CXX compile features
-- Detecting CXX compile features - done
```

Основная часть этой обработки обычно происходит в момент вызова первой команды project(), после чего результаты тестов цепочки инструментов кэшируются. Команда enable_language() также вызывает такую обработку, когда включает ранее не включенный язык, как и другой вызов project(), добавляющий ранее не включенный язык. После включения языка его кэшированные данные всегда будут использоваться вместо повторного тестирования цепочки инструментов, даже при последующих запусках CMake. Это имеет по крайней мере два важных последствия:

- После того, как каталог сборки был настроен на определенную цепочку инструментов, ее нельзя (безопасно) изменить. В некоторых ситуациях CMake может обнаружить, что цепочка инструментов была изменена, и отбросить свои предыдущие результаты, но при этом отбрасываются только кэшированные данные, непосредственно связанные с цепочкой инструментов. Любые другие кэшированные величины, основанные на кэшированных деталях цепочки инструментов за пределами тех, о которых известно CMake, не будут сброшены. Поэтому перед сменой цепочки инструментов следует полностью очистить каталог сборки (возможно, недостаточно просто удалить файл CMakeCache.txt, другие детали могут быть кэшированы в других местах).
- Различные цепочки инструментов не могут быть смешаны непосредственно в одном проекте. CMake принципиально рассматривает проект как использующий одну цепочку инструментов. Чтобы использовать несколько цепочек инструментов, необходимо структурировать проект так, чтобы часть сборки выполнялась как внешние подсборки (эта техника обсуждается в разделе 27.1, "ExternalProject" и разделе 28.1, "Структура суперсборки").

21.1. Файлы цепочки инструментов

Если инструментальная цепочка по умолчанию не подходит, то рекомендуемый способ указания нужных деталей инструментальной цепочки - это *файл инструментальной цепочки*. Это обычный сценарий CMake, который обычно содержит в основном команды set(...). Они определяют переменные, которые CMake использует для описания целевой платформы, расположения различных компонентов цепочки инструментов и так далее. Имя файла цепочки инструментов передается в CMake через специальную кэш-переменную CMAKE_TOOLCHAIN_FILE следующим образом:

```
cmake -DCMAKE_TOOLCHAIN_FILE=myToolchain.cmake path/to/source
```

Можно использовать полный абсолютный путь или относительный путь, как в приведенном выше примере, CMake сначала смотрит относительно вершины каталога сборки, затем, если он не найден, относительно вершины каталога исходников. Этот файл цепочки инструментов должен быть указан при первом запуске CMake для каталога сборки, он не может быть добавлен позже или изменен для указания на другую цепочку инструментов. Поскольку сама переменная кэшируется, нет необходимости повторно указывать ее при последующих запусках CMake.

Файл цепочки инструментов считывается при каждом вызове команды project(), а не только при первом. Обычно это прозрачная деталь реализации, о которой разработчику не нужно много думать, но она может привести к неожиданному поведению. Если файл цепочки инструментов устанавливает или изменяет переменные, которыми манипулирует сам проект, или если файл цепочки инструментов неверно предполагает, что он обрабатывается только один раз для всего проекта, то разработчику может показаться, что команды project() повреждают настройки цепочки инструментов или что переменные таинственным образом изменяются без какого-либо очевидного кода, вносящего эти изменения. Поэтому разработчики должны следить за тем, чтобы цепочки инструментов были минимальными, устанавливая только то, что необходимо, и делая как можно меньше предположений о том, что делает проект. Файлы цепочки инструментов в идеале должны быть полностью отделены от проекта и даже должны быть многократно используемыми в разных проектах, поскольку они должны описывать только цепочку инструментов, а не то, как они взаимодействуют с конкретным проектом.

Содержимое файла цепочки инструментов может варьироваться, но в целом есть только несколько основных вещей, которые они потенциально должны делать:

- Опишите основные детали целевой системы.
- Укажите пути к инструментам (обычно только к компиляторам).
- Установка флагов по умолчанию для инструментов (обычно только для компиляторов и, возможно, компоновщиков).
- Устанавливает расположение корневой файловой системы целевой платформы в случае кросс-компиляции.

Довольно часто в файлы инструментария включается и другая логика, особенно для влияния на поведение различных команд find_...() (см. главу 23, *Поиск вещей*). Хотя есть ситуации, когда такая логика может быть уместна, можно привести аргумент, что в большинстве случаев такая логика может и должна быть частью проекта. Только проект знает, что он пытается найти, поэтому инструментарий не должен делать предположений о том, что хочет сделать проект.

21.2. Определение целевой системы

Фундаментальными переменными, описывающими целевую систему, являются:

- CMAKE_SYSTEM_NAME
- CMAKE_SYSTEM_PROCESSOR
- CMAKE_SYSTEM_VERSION

Из них CMAKE_SYSTEM_NAME является наиболее важным. Он определяет тип платформы, на которую *направлена сборка, в* отличие от CMAKE_HOST_SYSTEM_NAME, который определяет платформу, на которой *выполняется* сборка. CMake сам всегда задает CMAKE_HOST_SYSTEM_NAME, тогда как CMAKE_SYSTEM_NAME может быть (и часто задается) файлами инструментария. Можно считать, что CMAKE_SYSTEM_NAME - это то, что CMAKE_HOST_SYSTEM_NAME было бы установлено, если бы CMake можно было запустить непосредственно на целевой платформе. Таким образом, типичные значения включают Linux, Windows, QNX, Android или Darwin, но в некоторых ситуациях (например, для встраиваемых устройств с "голым металлом") вместо этого может использоваться системное имя Generic. Существуют также вариации типичных имен платформ, которые могут быть уместны в некоторых ситуациях, например, WindowsStore и WindowsPhone. Если CMAKE_SYSTEM_NAME задано в файле цепочки инструментов, то CMake также установит переменную CMAKE_CROSSCOMPILING в true, даже если она имеет то же значение, что и CMAKE_HOST_SYSTEM_NAME. Если CMAKE_SYSTEM_NAME не задана, ей будет присвоено то же значение, что и автоматически определяемой CMAKE_HOST_SYSTEM_NAME.

CMAKE_SYSTEM_PROCESSOR предназначен для описания аппаратной архитектуры целевой платформы. Если он не указан, ему будет присвоено то же значение, что и CMAKE_HOST_SYSTEM_PROCESSOR, которое автоматически заполняется CMake. В сценариях кросс-компиляции или при сборке для 32-битной платформы на 64-битном хосте с тем же типом системы это приведет к тому, что CMAKE_SYSTEM_PROCESSOR будет неверным. Поэтому рекомендуется установить CMAKE_SYSTEM_PROCESSOR, если архитектура не соответствует хосту сборки, даже если проект, кажется, собирается нормально без него. Неправильные решения, основанные на неверном значении CMAKE_SYSTEM_PROCESSOR, могут привести к тонким проблемам, которые нелегко обнаружить или диагностировать.

Переменная CMAKE_SYSTEM_VERSION имеет различные значения в зависимости от того, какое значение установлено в CMAKE_SYSTEM_NAME. Например, при имени системы WindowsStore, WindowsPhone или WindowsCE, версия системы будет использоваться для определения того, какой Windows SDK использовать. Значения могут быть более общими, например 8.1 или 10.0, или определять очень конкретный выпуск,

например 10.0.10240.0. В качестве другого примера, если CMAKE_SYSTEM_NAME имеет значение Android, то CMAKE_SYSTEM_VERSION обычно интерпретируется как версия API Android по умолчанию и должна быть целым положительным числом. Для других имен систем нередко CMAKE_SYSTEM_VERSION устанавливается в произвольное значение, например, 1, или не устанавливается вообще. В разделе инструментальных цепочек документации CMake приведены примеры различных вариантов использования CMAKE_SYSTEM_VERSION, но смысл и набор допустимых значений переменной не всегда четко определены. По этой причине проектам рекомендуется проявлять осторожность при реализации логики, зависящей от значения CMAKE_SYSTEM_VERSION.

Обычно эти три переменные CMAKE_SYSTEM_... полностью описывают целевую систему, но бывают и исключения:

- Все платформы Apple используют Darwin для CMAKE_SYSTEM_NAME, даже для iOS, tvOS или watchOS. CMAKE_SYSTEM_PROCESSOR и CMAKE_SYSTEM_VERSION также не имеют особого смысла для платформ Apple и обычно остаются без значения. Указание целевой системы осуществляется с помощью другой переменной, CMAKE_OSX_SYSROOT, которая выбирает базовый SDK, используемый для сборки. Целевое устройство определяется на основе выбранного SDK, но разработчик все еще может выбирать между устройством и симулятором во время сборки. Это сложная тема, которая подробно рассматривается в разделе 22.5, "Настройки сборки". Разработчики CMake также активно обсуждают улучшение этой области.
- Переменная CMAKE_SYSTEM_PROCESSOR обычно не устанавливается при работе с платформами Android. Это обсуждается в разделе 21.6.3, "Android", ниже.

21.3. Выбор инструмента

Из всех инструментов, используемых в сборке, компилятор, вероятно, является самым важным с точки зрения разработчика. Путь к компилятору контролируется переменной CMAKE_<LANG>_COMPILER, которая может быть установлена в файле цепочки инструментов или в командной строке для ручного управления используемым компилятором, или может быть опущена, чтобы позволить CMake выбрать его автоматически. Если имя исполняемого файла указано вручную без пути, CMake будет искать его с помощью функции find_program() (описано в разделе 23.3, "Поиск программ"). Если указан полный путь к компилятору, он будет использован напрямую. Если компилятор не указан вручную, CMake выберет компилятор на основе внутреннего набора значений по умолчанию для целевой платформы и генератора.

Большинство языков также поддерживают настройку компилятора путем указания переменной окружения вместо установки CMAKE_<LANG>_COMPILER. Они обычно следуют общепринятым соглашениям, например CC для компилятора C, CXX для компилятора C++, FC для компилятора Fortran и так далее. Эти переменные окружения будут иметь эффект только при первом запуске CMake в каталоге сборки и только если соответствующая переменная CMAKE_<LANG>_COMPILER не установлена в файле инструментальной цепочки или в командной строке CMake.

Когда компилятор известен, CMake идентифицирует его и пытается определить его версию. Эта информация доступна через CMAKE_<LANG>_COMPILER_ID и
CMAKE_<LANG>_COMPILER_VERSION соответственно. ID компилятора - это короткая строка, используемая для отличия одного компилятора от другого, распространенные значения: GNU, Clang, AppleClang, MSVC, Intel и так далее. Полный список поддерживаемых идентификаторов приведен в документации CMake для CMAKE_<LANG>_COMPILER_ID. Если удалось определить версию компилятора, то она будет иметь обычный вид *major.minor.patch.tweak*, где не все компоненты версии должны присутствовать (например, 4.9 будет правильной версией).

В дополнение к переменным CMAKE_<LANG>_COMPILER_ID и CMAKE_<LANG>_COMPILER_VERSION также поддерживаются аналогичные выражения-генераторы без ведущей части CMAKE_. Либо переменные, либо выражения-генераторы могут быть использованы для условного добавления содержимого только для определенных компиляторов или версий компиляторов. Например, в GCC 7 появилась новая опция -fcode-hoisting, и ниже показаны оба способа ее добавления для компиляции C++, только если она доступна:

```
add_library(foo ...)

# Conditionally add -fcode-hoisting option using variables
if(CXX_COMPILER_ID STREQUAL GNU AND
   NOT CXX_COMPILER_VERSION VERSION_LESS 7)
    target_compile_options(foo PRIVATE -fcode-hoisting)
endif()

# Same thing using generator expressions instead
target_compile_options(foo PRIVATE
    $<$<AND:$<CXX_COMPILER_ID:GNU>,
            $<VERSION_GREATER_EQUAL:$<CXX_COMPILER_VERSION>,7>>:-fcode-hoisting>
)
```

ID компилятора является наиболее надежным способом идентификации используемого компилятора. Единственный случай, о котором проектам, возможно, следует знать: до CMake 3.0 компилятор Apple Clang рассматривался так же, как и восходящий Clang, и оба имели идентификатор компилятора Clang. Начиная с CMake 3.0, компилятор Apple имеет идентификатор компилятора AppleClang, чтобы его можно было отличить от вышестоящего Clang. Политика CMP0025 была добавлена, чтобы позволить использовать старое поведение для тех проектов, которые этого требуют.

После определения пути к компилятору CMake может разработать соответствующий набор флагов по умолчанию для компилятора и компоновщика. Они видны в проекте как CMAKE_<LANG>_FLAGS, CMAKE_<LANG>_FLAGS_<CONFIG> , CMAKE_<TARGETTYPE>_LINKER_FLAGS и CMAKE_<TARGETTYPE>_LINKER_FLAGS_<CONFIG> переменные, которые были рассмотрены в разделе 14.3, "Переменные компилятора и компоновщика". Разработчики могут добавить свои собственные флаги в набор значений по умолчанию для этих переменных, используя переменные с тем же именем, но с добавлением _INIT. Эти переменные ..._INIT используются только для установки начальных значений по умолчанию, они не имеют никакого эффекта после того, как CMake был запущен один раз и фактические значения были сохранены в кэше.

Частой ошибкой является установка переменных non-...INIT в файле цепочки инструментов (т.е. установка CMAKE_<LANG>_FLAGS, а не CMAKE_<LANG>_FLAGS_INIT). Это имеет нежелательный эффект отбрасывания или скрытия любых изменений, которые разработчик может внести в эти переменные в кэше. Поскольку файл toolchain также перечитывается при каждом вызове project(), он также может отбрасывать любые изменения этих переменных, сделанные самим проектом. Установка переменных ...INIT вместо этого гарантирует, что будут затронуты только начальные значения по умолчанию, а все последующие изменения переменных, не являющихся ..._INIT, с помощью любого метода будут сохранены.

В качестве примера рассмотрим файл toolchain, который разработчик может использовать для настройки своей сборки со специальными флагами компилятора для отладки (это может быть полезным способом повторного использования некоторой сложной логики, предназначенной только для разработчиков, в нескольких проектах без необходимости добавлять ее в каждый проект). Далее выбираются компиляторы GNU и добавляются флаги, включающие большинство предупреждений:

```
set(CMAKE_C_COMPILER   gcc)
set(CMAKE_CXX_COMPILER g++)

set(extraOpts "-Wall -Wextra")
set(CMAKE_C_FLAGS_DEBUG_INIT   ${extraOpts})
set(CMAKE_CXX_FLAGS_DEBUG_INIT ${extraOpts})
```

К сожалению, существуют некоторые несоответствия в том, как CMake сочетает указанные разработчиком параметры ..._INIT с параметрами по умолчанию, которые он обычно предоставляет. В большинстве случаев CMake добавляет дополнительные опции к тем, которые указаны переменными ...INIT, но в некоторых комбинациях платформы/компилятора (особенно в старых или менее часто используемых) значения ..._INIT, указанные разработчиком, могут быть отброшены. Это связано с историей этих переменных, которые раньше были предназначены только для внутреннего использования и всегда в одностороннем порядке устанавливали значения ..._INIT. Начиная с CMake 3.7, переменные ..._INIT были задокументированы для общего использования, и поведение было переключено на добавление, а не замену для широко используемых компиляторов. Поведение для очень старых или более не поддерживаемых компиляторов было оставлено без изменений.

Некоторые компиляторы действуют скорее как драйверы компилятора, то есть они ожидают аргумента командной строки для указания целевой платформы/архитектуры для компиляции. Clang и QNX qcc являются примерами компиляторов, использующих такую схему. Для тех компиляторов, которые CMake распознает как требующие таких аргументов, переменная CMAKE_<LANG>_COMPILER_TARGET может быть установлена в файле цепочки инструментов для указания цели. Там, где это поддерживается, это следует использовать вместо того, чтобы пытаться вручную добавить флаги с помощью CMAKE_<LANG>_FLAGS_INIT.

Другая, менее распространенная ситуация - когда набор инструментов компилятора не включает другие вспомогательные утилиты, такие как архиваторы или компоновщики. Такие драйверы компилятора обычно поддерживают аргумент командной строки, который можно использовать для указания места, где можно найти эти инструменты. CMake предоставляет переменную CMAKE_<LANG>_COMPILER_EXTERNAL_TOOLCHAIN, которая может использоваться для указания каталога, в котором находятся эти утилиты.

21.4. Системные корни

Во многих случаях инструментарий - это все, что нужно, но иногда проектам может потребоваться доступ к более широкому набору библиотек, заголовочных файлов и т.д., которые можно найти на целевой платформе. Обычный способ решения этой проблемы - предоставить сборке урезанную версию (или даже полную версию) корневой файловой системы целевой платформы. Это называется *системным корнем* или просто *sysroot* для краткости. По сути, sysroot - это просто корневая файловая система целевой платформы, смонтированная или скопированная на путь, доступный через файловую систему хоста. Пакеты инструментария часто предоставляют минимальный sysroot, содержащий различные библиотеки и т.д., необходимые для компиляции и компоновки.

CMake имеет довольно широкую и простую в использовании поддержку sysroot. Файлы цепочки инструментов могут устанавливать переменную CMAKE_SYSROOT в местоположение sysroot, и только на основе этой информации CMake может находить библиотеки, заголовки и т.д. преимущественно в области sysroot, а не в одноименных файлах на хосте (это подробно рассматривается в разделе 23.1.2, "Контроль кросс-компиляции"). Во многих случаях CMake также автоматически добавит необходимые флаги компилятора/линковщика в базовые инструменты, чтобы они знали об области sysroot. Для более сложных сценариев, когда для компиляции и линковки необходимо предоставить разные sysroot (например, как в Android NDK с

унифицированными заголовками), файлы цепочки инструментов могут устанавливать CMAKE_SYSROOT_COMPILE и CMAKE_SYSROOT_LINK вместо этого при использовании CMake 3.9 или более поздней версии.

В некоторых случаях разработчики могут выбрать монтирование полной целевой файловой системы в точку монтирования хоста и использовать ее в качестве sysroot. Она может быть смонтирована как доступная только для чтения, а если нет, то может быть желательно оставить ее немодифицированной при сборке. Поэтому, когда проект собран, его может потребоваться установить в другое место, а не записывать в область sysroot. CMake предоставляет переменную CMAKE_STAGING_PREFIX, которая может быть использована для задания точки постановки, ниже которой будут устанавливаться все команды установки (см. раздел 25.1.2, "Базовое место установки" для обсуждения этой области). Эта область установки может быть точкой монтирования запущенной целевой системы, и установленные двоичные файлы могут быть протестированы сразу после установки. Такое расположение особенно полезно при кросс-компиляции на быстром хосте для целевой системы, сборка на которой в противном случае была бы медленной (например, сборка на настольной машине для целевой системы Raspberry Pi). В разделе 23.1.2, "Средства управления кросс-компиляцией", также обсуждается, как CMAKE_STAGING_PREFIX влияет на способ поиска CMake библиотек, заголовков и так далее.

21.5. Проверки компилятора

Когда вызов project() или enable_language() вызывает тестирование возможностей компилятора и языка, команда try_compile() вызывается для выполнения различных проверок. Если был предоставлен файл toolchain, он считывается при каждом вызове try_compile(), поэтому тестовый проект будет настроен аналогично основной сборке. CMake автоматически передаст некоторые соответствующие переменные, такие как CMAKE_<LANG>_FLAGS, но файлы цепочки инструментов могут потребовать передачи в тестовую сборку и других переменных. Поскольку основная сборка сначала прочитает файл цепочки инструментов, сам файл цепочки инструментов может определить, какие переменные должны быть переданы в тестовую сборку. Это делается путем добавления имен переменных в переменную CMAKE_TRY_COMPILE_PLATFORM_VARIABLES (не задавайте ее в проекте, только в файле инструментальной цепочки). Используйте list(APPEND), а не set(), чтобы переменные, добавленные CMake, не были потеряны. Не имеет значения, если CMAKE_TRY_COMPILE_PLATFORM_VARIABLES будет содержать дубликаты, важно только, чтобы нужные имена переменных присутствовали.

Команда try_compile() обычно компилирует и компонует тестовый код для создания исполняемого файла. В некоторых сценариях кросс-компиляции это может представлять проблему, если запуск компоновщика требует пользовательских флагов или скриптов компоновщика, или по другим причинам нежелателен (кросс-компиляция для пустой целевой платформы может иметь такое ограничение). Если используется CMake 3.6 или более поздняя версия, команде можно указать на создание статической библиотеки, установив CMAKE_TRY_COMPILE_TARGET_TYPE в STATIC_LIBRARY. Это избавляет от необходимости использования компоновщика, но все же требует наличия инструмента архивации. CMAKE_TRY_COMPILE_TARGET_TYPE также может иметь значение EXECUTABLE, что в любом случае является поведением по умолчанию, если значение не задано. До CMake 3.6 для предотвращения вызова try_compile() использовался ныне устаревший модуль CMakeForceCompiler, но теперь CMake в значительной степени полагается на эти тесты, чтобы выяснить, какие функции поддерживаются компиляторами, поэтому использование CMakeForceCompiler теперь активно не рекомендуется.

Хотя она не вызывается при проверке компилятора, команда try_run() тесно связана с try_compile(), и на ее поведение влияет кросс-компиляция. try_run() - это фактически try_compile(), за которой следует попытка запустить только что собранный исполняемый файл. Если CMAKE_CROSSCOMPILING имеет значение true, CMake изменяет свою логику запуска тестового исполняемого файла. Если переменная

CMAKE_CROSSCOMPILING_EMULATOR установлена, CMake добавляет ее к команде, которая в противном случае использовалась бы для запуска исполняемого файла на целевой платформе, и использует ее для запуска исполняемого файла на основной платформе. Если CMAKE_CROSSCOMPILING_EMULATOR не установлена, когда CMAKE_CROSSCOMPILING равен true, CMake требует от инструментария или проекта вручную установить некоторые переменные кэша. Эти переменные предоставляют код завершения и вывод из stdout и stderr, которые были бы получены, если бы исполняемый файл мог быть запущен на целевой платформе. Необходимость задавать их вручную явно неудобна и чревата ошибками, поэтому проекты должны стараться избегать вызова try_run() в ситуациях кросс-компиляции, когда CMAKE_CROSSCOMPILING_EMULATOR не может быть установлен. В тех случаях, когда невозможно обойтись без этих переменных, определяемых вручную, документация CMake для команды try_run() предоставляет необходимые детали относительно переменных, которые необходимо установить. Дальнейшее использование CMAKE_CROSSCOMPILING_EMULATOR также обсуждается в разделе 24.6, "Кросс-компиляция и эмуляторы".

21.6. Примеры

Следующие примеры были выбраны для того, чтобы подчеркнуть концепции, обсуждаемые в этой главе. В разделе "Инструментальные цепочки" справочной документации CMake содержатся дополнительные примеры для различных целевых платформ.

21.6.1. Raspberry Pi

Перекрестная компиляция для Raspberry Pi - это хорошее введение в то, как CMake работает с перекрестной компиляцией в целом. Первым шагом является получение цепочки инструментов компилятора, наиболее распространенным способом является использование утилиты типа crosstool-NG. В дальнейшем в этом примере будет использоваться /path/to/toolchain для обозначения вершины структуры каталогов инструментария.

Типичный файл toolchain для Raspberry Pi может выглядеть примерно так:

```
set(CMAKE_SYSTEM_NAME      Linux)
set(CMAKE_SYSTEM_PROCESSOR ARM)

set(CMAKE_C_COMPILER   /path/to/toolchain/bin/armv8-rpi3-linux-gnueabihf-gcc)
set(CMAKE_CXX_COMPILER /path/to/toolchain/bin/armv8-rpi3-linux-gnueabihf-g++)

set(CMAKE_SYSROOT      /path/to/toolchain/armv8-rpi3-linux-gnueabihf/sysroot)
```

Если на хосте есть точка монтирования работающего целевого устройства, ее можно использовать для относительно простого тестирования двоичных файлов, собранных в рамках проекта. Например, предположим, что /mnt/rpiStage - это точка монтирования, прикрепленная к работающему Raspberry Pi (желательно, чтобы она указывала на какой-нибудь локальный каталог, а не на корень системы, чтобы ее можно было стереть или изменить произвольным образом без дестабилизации работающей системы). В файле инструментальной цепочки эта точка монтирования будет указана в качестве области хранения следующим образом:

```
set(CMAKE_STAGING_PREFIX /mnt/rpiStage)
```

Затем двоичные файлы проекта могут быть установлены в эту область хранения и запущены непосредственно на устройстве (см. раздел 25.1.2, "Место установки базы").

21.6.2. GCC с 32-битной целью на 64-битном хосте

GCC позволяет собирать 32-битные двоичные файлы на 64-битных хостах путем добавления флага -m32 к командам компилятора и компоновщика. Следующий пример цепочки инструментов по-прежнему позволяет находить компиляторы GCC в PATH, добавляя только дополнительный флаг к начальному набору, используемому компиляторами и компоновщиком. В зависимости от точки зрения, такая схема может рассматриваться как кросс-компиляция или нет. Поэтому установка CMAKE_SYSTEM_NAME может также рассматриваться как необязательная, поскольку ее установка заставляет CMAKE_CROSSCOMPILING иметь значение true. В любом случае, CMAKE_SYSTEM_PROCESSOR все равно должен быть установлен, так как цель этого файла инструментария - нацелить его на процессор, отличный от процессора хоста.

```
set(CMAKE_SYSTEM_NAME      Linux)
set(CMAKE_SYSTEM_PROCESSOR i686)

set(CMAKE_C_COMPILER   gcc)
set(CMAKE_CXX_COMPILER g++)

set(CMAKE_C_FLAGS_INIT   -m32)
set(CMAKE_CXX_FLAGS_INIT -m32)

set(CMAKE_EXE_LINKER_FLAGS_INIT    -m32)
set(CMAKE_SHARED_LINKER_FLAGS_INIT -m32)
set(CMAKE_MODULE_LINKER_FLAGS_INIT -m32)
```

Одним из способов подтверждения того, что сборка действительно 32-битная, является переменная CMAKE_SIZEOF_VOID_P, которая вычисляется CMake автоматически в рамках настройки инструментария. Для 64-битных сборок она будет иметь значение 8, а для 32-битных - 4.

```
math(EXPR bitness "${CMAKE_SIZEOF_VOID_P} * 8")
message("${bitness}-bit build")
```

21.6.3. Android

Кросс-компиляция для Android может быть немного сложнее, чем представленные до сих пор основные случаи, и есть некоторые различия в том, как описывается целевая система. CMAKE_SYSTEM_NAME должно быть установлено на Android, но CMAKE_SYSTEM_PROCESSOR обычно не устанавливается, а значение CMAKE_SYSTEM_VERSION часто оставляется на усмотрение CMake. Вместо того чтобы задавать пути к отдельным компиляторам и инструментам, ряд переменных, специфичных для Android, управляет конфигурацией набора инструментов. Тип используемого генератора CMake также влияет на доступные опции, причем разные генераторы поддерживают разные среды разработки. Например, при использовании генератора Visual Studio CMake требует установки NVidia Nsight Tegra Visual Studio Edition. С другой стороны, использование Ninja или одного из генераторов Makefile позволяет разработчику выбирать между использованием Android NDK и отдельной цепочки инструментов.

NDK и автономные цепочки инструментов

Когда используется Ninja или генератор Makefile, CMake использует последовательность шагов, чтобы определить, следует ли ему использовать NDK или отдельную цепочку инструментов. Эти шаги четко описаны в документации по инструментальной цепочке CMake, но может быть полезно разбить эти шаги на несколько частей (используется первое соответствие):

Прямое указание среды разработки

- Если переменная CMAKE_ANDROID_NDK установлена, то будет использоваться NDK в этом месте.
- Если переменная CMAKE_ANDROID_STANDALONE_TOOLCHAIN установлена, будет использоваться автономная цепочка инструментов в этом месте. Это место должно иметь подкаталог sysroot.

Установите CMAKE_SYSROOT

- Если CMAKE_SYSROOT установлен на каталог вида <ndk>/platforms/android-<api>/arch-<arch> , то это будет выглядеть так, как если бы CMAKE_ANDROID_NDK был установлен на часть пути <ndk>. Уровень API Android по умолчанию будет установлен на часть пути <api>, если он явно не указан в файле цепочки инструментов (см. ниже).
- Если CMAKE_SYSROOT установлен на каталог вида <someDir>/sysroot, то это будет выглядеть так, как если бы CMAKE_ANDROID_STANDALONE_TOOLCHAIN был установлен на <someDir>.

Альтернативные переменные CMake

- Если установлено значение ANDROID_NDK, это будет рассматриваться так, как если бы было установлено значение CMAKE_ANDROID_NDK. Новым проектам лучше не полагаться на это и вместо этого использовать более каноническое значение
  Переменная CMAKE_ANDROID_NDK напрямую.
- Если ANDROID_STANDALONE_TOOLCHAIN установлен, он будет рассматриваться как будто
  CMAKE_ANDROID_STANDALONE_TOOLCHAIN была установлена. Новые проекты должны предпочесть не полагаться на это и вместо этого использовать более каноническую переменную CMAKE_ANDROID_STANDALONE_TOOLCHAIN напрямую.

Переменные среды

- Если установлены переменные окружения ANDROID_NDK_ROOT или ANDROID_NDK, они будут использоваться в качестве значения переменной CMAKE_ANDROID_NDK CMake.
- Если установлена переменная окружения ANDROID_STANDALONE_TOOLCHAIN, она будет использоваться в качестве значения переменной CMAKE_ANDROID_STANDALONE_TOOLCHAIN CMake.

NDK дает разработчику немного больше гибкости, чем отдельная инструментальная цепочка. Если отдельная цепочка инструментов нацелена на одну архитектуру и уровень API, то NDK может содержать поддержку нескольких цепочек инструментов и, следовательно, различных архитектур, уровней API и т.д. Обратите внимание, что согласно дорожной карте NDK, автономные инструментальные цепочки будут выведены из употребления где-то в районе выпуска r19, после удаления всех инструментальных цепочек, кроме Clang и единственной реализации STL. Ниже приведена подборка наиболее значимых переменных для NDK и отдельных цепочек инструментов:

CMAKE_SYSTEM_VERSION

При использовании NDK этот параметр может быть установлен на уровне API Android, или его можно оставить на усмотрение CMake. Если переменная не установлена, CMake сначала проверяет, была ли установлена переменная CMAKE_ANDROID_API, и использует ее, если она доступна. В противном случае, если была установлена переменная CMAKE_SYSROOT, CMake попытается определить уровень API из структуры каталогов NDK. Если это также не удастся, будет использован последний уровень API, поддерживаемый NDK. Для автономной цепочки инструментов значение CMAKE_SYSTEM_VERSION всегда определяется автоматически из цепочки инструментов.

CMAKE_ANDROID_ARCH_ABI

Эта переменная определяет ABI Android. Для сборок NDK, если она не задана, по умолчанию будет использоваться armeabi для NDK выпусков до r16, или самый старый доступный arm ABI для более поздних выпусков.

CMAKE_ANDROID_ARCH_ABI может иметь другие значения, если NDK имеет необходимую поддержку архитектуры (например, `arm64-v8a, armeabi-v7a, armeabi-v6, mips, mips64, x86 или x86_64). Эта переменная устанавливается автоматически при использовании автономной цепочки инструментов. Значение CMAKE_ANDROID_ARCH будет получено из CMAKE_ANDROID_ARCH_ABI, чтобы обеспечить соответствующее более общее значение архитектуры, которое будет одним из arm, arm64, mips, mips64, x86 или x86_64.

CMAKE_ANDROID_ARM_MODE

Когда CMAKE_ANDROID_ARCH_ABI имеет значение одной из архитектур armeabi*, разработчики могут выбирать между 32-битными ARM и 16-битными Thumb процессорами. Если CMAKE_ANDROID_ARM_MODE имеет булево значение true, будет выбран процессор ARM, в противном случае, если установлено значение false или не установлено вообще, целевым процессором будет Thumb. Этот параметр может быть установлен как в NDK, так и в отдельном инструментарии.

CMAKE_ANDROID_ARM_NEON

Когда CMAKE_ANDROID_ARCH_ABI установлен на armeabi-v7a, CMAKE_ANDROID_ARM_NEON может быть установлен в булево значение true, чтобы включить поддержку NEON. Это значение может быть установлено как в NDK, так и в отдельном инструментарии.

CMAKE_ANDROID_NDK_TOOLCHAIN_VERSION

Эта специфическая для NDK переменная может быть использована для указания инструментальной цепочки, которую следует выбрать из NDK. Если она задана, значения должны принимать одну из следующих форм:

- X.Y - версия GCC X.Y
- clangX.Y - Clang версии X.Y
- clang - Последняя доступная версия Clang

Если эта переменная не задана, будет использоваться последняя версия GCC, доступная в NDK. Обратите внимание, что в документации NDK (r16) говорится, что GCC больше не поддерживается в NDK, а в дорожной карте NDK запланировано полное удаление GCC в r18, поэтому настоятельно рекомендуется запрашивать инструментарий Clang.

CMAKE_ANDROID_STL_TYPE

За исключением случаев использования автономного инструментария, различные реализации STL могут быть выбраны путем указания одного из поддерживаемых значений:

- none
- system
- gabi++_static
- gabi++_shared
- gnustl_static
- gnustl_shared
- c++_static
- c++_shared
- stlport_static
- stlport_shared

Если значение не указано, по умолчанию используется gnustl_static. Заметим, однако, что инструментарий GCC, к которому тесно привязаны реализации STL gnustl_*, будет недоступен начиная с NDK r18, а в старых NDK он поддерживает только C++11. Реализации stlport_* еще более старые и примитивные и даже не поддерживают C++11. Опция none не имеет поддержки C++ вообще, а опция system имеет только new и delete, но без STL.

В документации NDK r16 говорится, что типы c++_static и c++_shared STL будут единственными доступными типами в будущем выпуске NDK, а дорожная карта NDK показывает, что это произойдет в r18. Поэтому рекомендуется, чтобы проекты запрашивали одну из реализаций c++_* STL (это реализации стандартной библиотеки LLVM C++), а также использовали инструментарий Clang.

Каждая цель CMake имеет собственное свойство ANDROID_STL_TYPE, а переменная CMAKE_ANDROID_STL_TYPE используется для задания начального значения этого свойства. В большинстве случаев желательно использовать один и тот же тип STL во всей сборке, поэтому использование переменной, а не установка отдельных свойств цели, скорее всего, будет проще и надежнее.

Минимальный пример файла инструментальной цепочки для сборки NDK выглядит следующим образом:

```
set(CMAKE_SYSTEM_NAME Android)
set(CMAKE_ANDROID_NDK /path/to/android-ndk)
```

Он будет использовать последний уровень API в NDK с последним инструментарием GCC. Он будет нацелен на архитектуру armeabi (процессоры Thumb) без поддержки neon и будет использовать реализацию gnustl_static STL. Более реалистичный пример задает еще несколько таких величин:

```
set(CMAKE_SYSTEM_NAME Android)
set(CMAKE_SYSTEM_VERSION 26) # API level
set(CMAKE_ANDROID_NDK /path/to/android-ndk)
set(CMAKE_ANDROID_ARCH_ABI arm64-v8a)
set(CMAKE_ANDROID_NDK_TOOLCHAIN_VERSION clang)
set(CMAKE_ANDROID_STL_TYPE c++_shared)
```

Выше используется новейший инструментарий Clang и общая среда выполнения STL с поддержкой более современных стандартов C++.

Для сравнения, файл автономной цепочки инструментов обычно очень прост, поскольку многие решения по конфигурации предопределены самой цепочкой инструментов:

```
set(CMAKE_SYSTEM_NAME Android)
set(CMAKE_ANDROID_STANDALONE_TOOLCHAIN /path/to/android-toolchain)
```

Некоторые инструменты могут принудительно использовать свой собственный внутренний файл цепочки инструментов, что потенциально усложняет для разработчиков возможность указать любые из вышеперечисленных параметров. Android Studio является одним из таких примеров, предоставляя свой собственный файл инструментария, который отменяет большую часть собственной логики CMake. Сборки gradle настроены на создание внешней сборки CMake, которая использует генератор Ninja и NDK, предоставляемый через менеджер Android SDK. Хотя прямой доступ к файлу toolchain не разрешен, сборка gradle предоставляет ряд переменных gradle, которые переводятся в эквиваленты CMake. Разработчикам следует обратиться к документации инструмента, чтобы определить, можно ли использовать различные версии CMake и как повлиять на поведение сборки CMake. Похоже, что это активная область постоянного развития Android Studio.

Для разработчиков, использующих ndk-build (который по сути является оберткой для GNU make), а не gradle, в CMake 3.7 появилась возможность экспортировать файл Android.mk либо в процессе сборки CMake с помощью export(), либо на этапе установки с помощью install(). Форма экспорта во время сборки очень проста:

```
export(TARGETS target1 [target2...] ANDROID_MK fileName)
```

Имя файла обычно будет Android.mk с добавлением некоторого пути, чтобы поместить его в место, требуемое ndk-build. Каждая из названных целей будет включена в сгенерированный файл вместе с соответствующими требованиями по использованию, такими как флаги включения, определения компилятора и т.д. Как правило, это то, что нужно сделать проекту, если ему необходимо поддерживать участие в родительской ndk-сборке. Для случая, когда проект CMake будет упакован и хочет, чтобы его можно было легко включить в любую ndk-сборку, команда install() предоставляет необходимую функциональность (см. раздел 25.3, "Установка экспорта").

Генераторы Visual Studio

При использовании одного из генераторов Visual Studio, CMake требует установки NVidia Nsight Tegra Visual Studio Edition. Полученный в результате проект будет управлять всей сборкой, а не являться частью более крупной структуры gradle или ndk-build. Впервые поддержка была добавлена в CMake 3.1, но многие опции были добавлены только в CMake 3.4. Обычно генератор задается в командной строке CMake примерно следующим образом:

```
cmake -G "Visual Studio 12 2013 Tegra-Android" \
      -DCMAKE_TOOLCHAIN_FILE=/some/path/toolchain.cmake \
      /path/to/source
```

Минимальный файл инструментальной цепочки должен только установить CMAKE_SYSTEM_NAME в Android, но, как и в случае NDK и автономной инструментальной цепочки, могут быть установлены дополнительные переменные, влияющие на целевую архитектуру и т.д. В ряде случаев переменные, которые необходимо установить для сборок Visual Studio, отличаются от NDK, но часто связаны между собой.

В то время как сборки NDK и автономные сборки инструментария устанавливают CMAKE_ANDROID_ARCH_ABI и позволяют CMAKE_ANDROID_ARCH, чтобы быть производным от него, файлы инструментария для сборок Visual Studio устанавливают CMAKE_ANDROID_ARCH напрямую. Допустимые значения также отличаются для случая Visual Studio: armv7-a, armv7-a-hard, arm64-v8a, x86 и x86_64.

Аналогично, файлы инструментария для сборок Visual Studio будут устанавливать CMAKE_ANDROID_API, а не CMAKE_SYSTEM_VERSION, чтобы указать уровень Android API целевого устройства, при этом CMAKE_ANDROID_API будет действовать как значение по умолчанию для целевого свойства ANDROID_API. Кроме того, CMAKE_ANDROID_API_MIN может быть установлен для указания версии API, которая будет использоваться для сборки родного кода (он следует той же схеме и действует как значение по умолчанию для целевого свойства ANDROID_API_MIN). Это в некоторой степени аналогично ситуации для платформ Apple, где SDK, используемый для сборки, может быть указан отдельно для минимального уровня ОС целевого устройства (см. раздел 22.5, "Настройки сборки").

Переменная CMAKE_ANDROID_STL_TYPE может быть установлена и принимает значения, схожие с NDK, но значения c++_static и c++_shared не поддерживаются. Она используется в качестве значения по умолчанию для целевого свойства ANDROID_STL_TYPE.

Поскольку эта схема управляет всей сборкой, она должна настраивать не только собственный код, собранный CMake. Существует ряд других целевых свойств, которые относятся к частям сборки, не связанным со сборкой родного кода, например, настройки для зависимостей JAR, java-источников и т.д. Некоторые из этих целевых свойств также имеют связанные переменные CMake, которые определяют значения по умолчанию. Все эти целевые свойства имеют имена вида ANDROID_..., а переменные CMake по умолчанию имеют вид CMAKE_ANDROID_..... Эти подробности выходят за рамки рассматриваемого здесь материала, поэтому заинтересованным читателям следует обратиться к документации CMake для получения подробной информации о поддерживаемых свойствах и переменных, а затем установить их в соответствии с требованиями для неродных частей своего проекта.

21.7. Рекомендуемые практики

Файлы цепочки инструментов поначалу могут показаться немного пугающими, но во многом это происходит из-за того, что многие примеры и проекты закладывают в них слишком много логики. Файлы цепочки инструментов должны быть настолько минимальными, насколько это возможно, чтобы поддерживать необходимые инструменты, и они, как правило, должны быть многократно используемыми в различных проектах. Логика, специфичная для конкретного проекта, должна находиться в его собственных файлах CMakeLists.txt.

При написании файлов цепочки инструментов разработчики должны убедиться, что их содержимое не предполагает, что они будут выполнены только один раз. CMake может обрабатывать файл цепочки инструментов несколько раз в зависимости от того, что делает проект (например, несколько вызовов project() или enable_language()). Файл цепочки инструментов также может использоваться для временных сборок "на стороне" в рамках вызовов try_compile(), поэтому они не должны делать никаких предположений о контексте, в котором они используются.

Избегайте использования устаревшего модуля CMakeForceCompiler для установки компилятора, который будет использоваться при сборке. Этот модуль был популярен при использовании старых версий CMake, но новые версии в значительной степени полагаются на тестирование цепочки инструментов и отработку поддерживаемых ею функций. Модуль CMakeForceCompiler в основном предназначался для случаев, когда компилятор не был известен CMake, но использование таких компиляторов с последними версиями CMake,

скорее всего, приведет к нетривиальным ограничениям. Рекомендуется сотрудничать с разработчиками CMake для добавления необходимой поддержки таких компиляторов.

Будьте осторожны, чтобы не отбросить или неправильно обработать содержимое переменных, которые могут быть уже установлены к моменту обработки файла цепочки инструментов. Частой ошибкой является изменение переменных типа CMAKE_<LANG>_FLAGS, а не CMAKE_<LANG>_FLAGS_INIT, что может отбросить значения, установленные разработчиками вручную, или плохо взаимодействовать с уже установленными значениями при многократной обработке файла цепочки инструментов.

При работе с платформами Android предпочтительнее использовать простой файл toolchain с NDK и генератором Ninja или Makefile. Эта комбинация имеет наилучшую поддержку CMake и наиболее проста в использовании. Файлы цепочки инструментов могут быть очень простыми, и последние версии инструментов IDE, таких как Android Studio, переходят на использование этого подхода. Если разработчики используют собственные файлы цепочки инструментов, избегайте популярного файла цепочки инструментов *taka-no-me, на который* часто ссылаются в примерах в Интернете, поскольку он слишком сложен и имеет известные проблемы. Новые версии CMake поддерживают гораздо более простые файлы цепочки инструментов, которые работают гладко и требуют минимальных усилий.

Проекты, как правило, должны избегать использования переменной CMAKE_CROSSCOMPILING для любой своей логики. Эта переменная может вводить в заблуждение, поскольку она может быть установлена в true, даже если целевая и хост-платформа одинаковы, или в false, если они разные. Непоследовательность ее значения делает ее ненадежной. Авторы проектов также должны знать, что некоторые генераторы мультиконфигураций (например, Xcode) позволяют выбирать целевую платформу во время сборки, поэтому логика CMake, основанная на выборе кросс-компиляции или нет, должна быть написана очень тщательно для обработки различных ситуаций, в которых может быть сгенерирован проект.

Файлы цепочки инструментов часто содержат команды для изменения места поиска программ, библиотек и других файлов в CMake. Рекомендации по этой области см. в главе 23, "*Поиск вещей*".

Глава 22. Особенности Apple

Платформы Apple обладают рядом уникальных характеристик, которые напрямую влияют на способ создания программного обеспечения. Если простые приложения командной строки для macOS можно создавать аналогично другим платформам на базе Unix, то приложения с графическим пользовательским интерфейсом обычно предоставляются в специфическом для Apple формате, известном как *пакет приложений* (или просто *app bundle*). Эти пакеты представляют собой не просто один исполняемый файл, а стандартную структуру каталогов, содержащую множество файлов, связанных с приложением. Эти пакеты приложений должны быть автономными, их можно перемещать как единое целое и размещать в любом месте файловой системы пользователя.

Аналогичная ситуация существует и для библиотек. Отдельные статические и разделяемые библиотеки могут быть созданы так же, как и на других платформах на базе Unix, но они также могут быть собраны как часть *фреймворка*, который по сути является эквивалентом библиотеки в пакете приложений. Фреймворки имеют свою собственную стандартизированную структуру каталогов и могут содержать не только двоичные файлы библиотек. Они даже могут поддерживать несколько версий в рамках этой структуры каталогов. Библиотеки, предназначенные для загрузки во время выполнения, могут быть собраны в виде загружаемого пакета, что соответствует функциональности CFBundle от Apple.

Пакеты и фреймворки - это важные части механизма, используемого для создания контента для магазина приложений Apple. Другим ключевым аспектом является подписание кода - процесс, который проверяет целостность и происхождение программного обеспечения и является обязательной частью распространения в

магазине приложений. Права на код также являются неотъемлемой частью процесса сборки и определяют, какие функции Apple может использовать код. Эти права являются частью информации, запечатываемой в процессе подписания кода, и должны быть определены во время сборки, если набор прав по умолчанию (который пуст) не подходит.

В совокупности эти особенности представляют собой уникальные проблемы для проектов CMake. Следующие разделы предоставляют инструменты для понимания и решения этих проблем, а в некоторых случаях указывают на текущие ограничения поддержки CMake. Следует также отметить, что хотя CMake формально поддерживает macOS и iOS, поддержку tvOS и watchOS следует считать неполной.

22.1. Выбор генератора CMake

Технологии и инструменты, используемые для создания фреймворков и пакетов, постоянно развиваются, при этом в основных релизах Apple OS часто появляются новые функции и меняются требования к подписанию, распространению и т. д. Процессы и технологии тесно интегрированы в Xcode как основной инструмент, который Apple ожидает от разработчиков, при этом разработчики обычно обновляют Xcode до текущего выпуска, а не остаются с прошлыми основными выпусками. Такие области, как компиляция ресурсов, подписание кода и т. д., автоматически обрабатываются в рамках создания приложений и фреймворков, многие аспекты которых уникальны для экосистемы Apple.

Для проектов CMake это означает, что генератор Xcode является наиболее надежным и удобным для сборки с использованием инструментария Xcode. Другие генераторы, такие как Makefiles или Ninja, как правило, не имеют некоторой автоматизации генератора Xcode, или они могут отставать в реализации поддержки некоторых последних функций Xcode. За исключением базовых неподписанных настольных приложений, не предназначенных для распространения через магазин приложений, разработчикам придется использовать генератор Xcode, чтобы получить сборку с поддержкой необходимых функций. Также обратите внимание, что быстро меняющаяся природа платформ Apple означает, что разработчики также должны использовать достаточно свежий выпуск CMake, чтобы не отставать от изменений.

Одним из уникальных преимуществ генератора Xcode является то, что он поддерживает установку произвольных атрибутов проекта Xcode. Большинство параметров проекта можно изменить в виде ключа-значения на целевой основе с помощью целевых свойств вида XCODE_ATTRIBUTE_XXX, где XXX - имя свойства Xcode. Эти имена определены в документации Apple, но потенциально более удобный способ найти их - открыть существующий проект Xcode, перейти к настройкам сборки цели и щелкнуть на интересующей настройке сборки. В редакторе помощника *Quick Help отобразится* название настройки вместе с описанием. Умолчания для всех целей могут быть заданы соответствующими переменными CMAKE_XCODE_ATTRIBUTE_XXX, которые будут использоваться для инициализации соответствующего свойства цели при ее определении. Пример, демонстрирующий установку некоторых наиболее часто используемых атрибутов, может выглядеть следующим образом:

```
# Set the default signing identity and team ID to use for all targets
set(CMAKE_XCODE_ATTRIBUTE_CODE_SIGN_IDENTITY "iPhone Developer"
set(CMAKE_XCODE_ATTRIBUTE_DEVELOPMENT_TEAM   XYZ123ABCD)

# Some target-specific settings
set_target_properties(myiOSApp PROPERTIES
    XCODE_ATTRIBUTE_TARGETED_DEVICE_FAMILY     1,2
    XCODE_ATTRIBUTE_IPHONEOS_DEPLOYMENT_TARGET 10.0
)
```

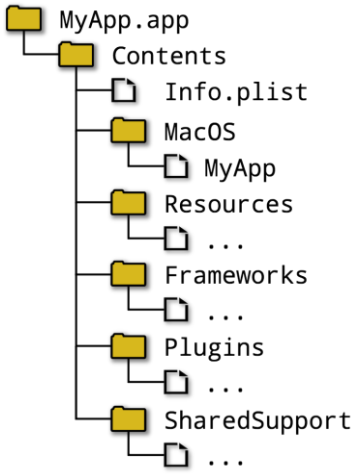
Эта возможность также может быть использована для установки атрибута Xcode только для одного определенного типа сборки путем добавления [variant=ConfigName] к имени свойства. Другие типы суффиксов также могут быть добавлены к имени свойства для еще более специфических настроек атрибута, но их использование будет необычным. Даже суффиксы [variant=...] будут нужны нечасто. Следующий пример дает представление о том, в каких случаях эта функция может быть полезна:

```
set_target_properties(myiOSApp PROPERTIES
    XCODE_ATTRIBUTE_GCC_UNROLL_LOOPS[variant=Release] YES
    XCODE_ATTRIBUTE_ENABLE_TESTABILITY[variant=Debug] YES
)
```

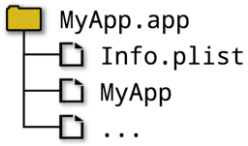
Некоторые проекты могут потребовать установки довольно большого количества атрибутов, чтобы получить желаемое поведение и возможности Xcode, в то время как другие проекты могут быть довольно простыми и требовать лишь минимального количества дополнительных настроек. Некоторые атрибуты нужны только в очень специфических обстоятельствах, тогда как другие настолько распространены, что присутствуют (или должны присутствовать) почти в каждом проекте, ориентированном на Apple. Некоторые из них обсуждаются в этой главе, в том числе и те, которые используются в приведенных выше примерах.

22.2. Пакеты приложений

Структура пакета приложений для macOS отличается от таковой для iOS, tvOS и watchOS. Структура macOS разделяет различные категории файлов по подкаталогам и выглядит примерно следующим образом (приложения могут иметь только некоторые из показанных подкаталогов):



В отличие от этого, структура пакетов для iOS, tvOS и watchOS уплощена и практически не имеет определенной структуры:



При создании пакета приложений CMake несколько абстрагируется от этих структурных различий, позволяя, по крайней мере, некоторые вещи обрабатывать одинаково независимо от того, создается ли пакет для macOS или для iOS, tvOS или watchOS. Однако разработчики должны знать, что некоторые области этой абстракции не были реализованы корректно до самых последних выпусков CMake (конкретным примером является работа с ресурсами), поэтому настоятельно рекомендуется использовать последнюю версию CMake.

Приложение идентифицируется как пакет путем добавления ключевого слова MACOSX_BUNDLE в add_executable():

```
add_executable(myApp MACOSX_BUNDLE ...)
```

Это устанавливает свойство цели MACOSX_BUNDLE в TRUE, которое платформы не-Apple просто игнорируют. Проект может альтернативно установить переменную CMAKE_MACOSX_BUNDLE в TRUE, и все последующие определенные исполняемые цели будут иметь свойство MACOSX_BUNDLE target, но более распространенным и, возможно, более понятным будет использование ключевого слова MACOSX_BUNDLE в каждой команде add_executable() (проекты обычно определяют только небольшое количество пакетов приложений, часто только один).

Несколько смущает тот факт, что MACOSX_BUNDLE применяется не только к macOS, но и к iOS, tvOS и watchOS. Ключевое слово появилось раньше, чем не настольные платформы Apple, отсюда и специфическое для настольных систем название. Вместо того чтобы создавать новые ключевые слова для каждой из других платформ, использование существующего ключевого слова было расширено, чтобы охватить все платформы Apple. Подобная схема расширения ключевых слов и переменных, специфичных для OSX, на все платформы Apple прослеживается и в ряде других случаев, но обратите внимание, что это не универсально для всех переменных и свойств, связанных с OSX. Те из них, для которых это верно, выделены в этой главе.

Каждый пакет приложений должен содержать как минимум файл Info.plist и основной исполняемый файл (MyApp в примерах структуры каталогов выше). По умолчанию CMake предоставляет базовый файл Info.plist из файла шаблона, поставляемого с самим CMake. Однако в большинстве случаев проекты захотят предоставить свой собственный Info.plist, чтобы иметь полный контроль над конфигурацией приложения. Если приложение использует файлы раскадровки или конструктора интерфейсов, то предоставление собственного Info.plist практически обязательно, чтобы присутствовали соответствующие ключевые записи, такие как NSMainStoryboardFile. Свойство MACOSX_BUNDLE_INFO_PLIST target можно установить на имя файла, который будет использоваться в качестве шаблона Info.plist (для всех платформ Apple, не только macOS). Файл шаблона по умолчанию называется MacOSXBundleInfo.plist.in и находится в каталоге модулей CMake. Он может послужить полезной отправной точкой для создания собственных шаблонов.

Независимо от того, использует ли цель шаблон Info.plist по умолчанию или предоставленный проектом, CMake скопирует файл шаблона в пакет приложения, выполняя по пути некоторые специфические замены. В файле шаблона любое содержимое вида ${XXX} будет заменено значением целевого свойства XXX, если XXX является одним из свойств в таблице ниже. Каждое из этих свойств сопоставлено с определенным ключом в стандартном файле Info.plist, поэтому если проект предоставляет свой собственный файл шаблона и использует эти переменные, он должен в целом следовать тому же сопоставлению.

Property	Info.plist Key	Example
MACOSX_BUNDLE_BUNDLE_NAME	CFBundleName	MyApp
MACOSX_BUNDLE_BUNDLE_VERSION	CFBundleVersion	2.4.7rc1
MACOSX_BUNDLE_COPYRIGHT	NSHumanReadableCopyright	© 2018 MyCompany
MACOSX_BUNDLE_GUI_IDENTIFIER	CFBundleIdentifier	com.example.myapp
MACOSX_BUNDLE_SHORT_VERSION_STRING	CFBundleShortVersionString	2.4.7
MACOSX_BUNDLE_LONG_VERSION_STRING	CFBundleLongVersionString	*see below*
MACOSX_BUNDLE_INFO_STRING	CFBundleGetInfoString	*see below*
MACOSX_BUNDLE_ICON_FILE	CFBundleIconFile	*see below*

Apple больше не документирует CFBundleLongVersionString как один из ключей Info.plist, поэтому проекты могут не предоставлять его. В их документации также говорится, что NSHumanReadableCopyright заменил CFBundleGetInfoString и что CFBundleIconFile устарел и рекомендуется использовать CFBundleIconFiles или CFBundleIcons вместо него. CFBundleIconFile все еще используется, если ни одна из других альтернатив не установлена.

Если определяется несколько целей приложения, проект может установить переменные с теми же именами, что и свойства в приведенной выше таблице, и эти переменные будут использоваться для инициализации свойств цели. Обратите внимание, что это отличается от обычной конвенции CMake, согласно которой переменные имеют префикс CMAKE_... перед целевым свойством, для которого они действуют по умолчанию.

Когда проект предоставляет свой собственный файл шаблона Info.plist, он не обязан использовать вышеуказанные целевые свойства. Вполне допустимо вместо этого жестко закодировать значения. Обратите внимание, однако, что CFBundleVersion и CFBundleShortVersionString могут потребоваться из данных о версии, указанных в файлах CMakeLists.txt, поэтому установка их через подстановки MACOSX_BUNDLE_BUNDLE_VERSION и MACOSX_BUNDLE_SHORT_VERSION_STRING может быть наиболее удобным подходом. Требования Apple к номерам версий со временем изменились, и теперь формат *major.minor.patch* *является* обязательным (за некоторыми исключениями). Ниже показано одно потенциальное отображение для обеспечения номеров версий, удовлетворяющих требованиям Apple:

```
add_executable(myApp MACOSX_BUNDLE ...)
set_target_properties(myApp PROPERTIES
    MACOSX_BUNDLE_BUNDLE_VERSION       "${PROJECT_VERSION}${BUILD_SUFFIX}"
    MACOSX_BUNDLE_SHORT_VERSION_STRING "${PROJECT_VERSION}"
)
```

В приведенном выше примере BUILD_SUFFIX может быть пустой строкой для финальных релизов или одной или несколькими буквами, за которыми следует число в диапазоне 1-255. Примерами суффиксов могут быть a17 для альфа-релиза или rc2 для второго кандидата в релизы и т.д. Пример файла Info.plist, в котором используются эти свойства, приведен ниже.

Определив соответствующий файл Info.plist, можно переходить к исходным файлам, которые должны быть скомпилированы и подключены к пакету. В дополнение к обычным исходным файлам C/C++, платформы Apple также поддерживают исходные файлы Objective C/C++. Они обычно имеют суффикс .m или .mm и могут быть указаны в качестве источников в командах add_executable() и target_sources() так же, как и обычные файлы C/C++ (подробнее об определении целевых источников см. в разделе 28.5, "Определение целей").

Большинство генераторов CMake распознают эти суффиксы файлов и компилируют их соответствующим образом, а не только генератор Xcode.

Еще одна группа исходных файлов, уникальная для платформ Apple, - это файлы, используемые для определения пользовательского интерфейса. Файлы Storyboard или interface builder похожи на исходники, но они требуют дополнительной обработки для компиляции их как ресурсов и помещения скомпилированного результата в соответствующее место в пакете приложения. Только генератор Xcode реализует эту автоматическую компиляцию и копирование в нужное место, поэтому использование генераторов Makefile или Ninja обычно не рекомендуется, если в пакете приложений есть эти файлы. Исходные тексты Storyboard и interface builder должны быть указаны как исходные тексты в add_executable() или target_sources(). Чтобы они автоматически компилировались и копировались в соответствующее место в пакете, их также нужно указать в свойстве RESOURCE target. Например:

```
set(uiFiles
    Base.lproj/Main.storyboard
    Base.lproj/LaunchScreen.storyboard
)

add_executable(MyApp MACOSX_BUNDLE
    AppDelegate.m
    AppDelegate.h
    ViewController.m
    ViewController.h
    main.m
    ${uiFiles}
)

set_target_properties(MyApp PROPERTIES
    RESOURCE "${uiFiles}"
    MACOSX_BUNDLE_INFO_PLIST "${CMAKE_CURRENT_SOURCE_DIR}/Info.plist"
)
```

Обратите внимание на то, как обрабатываются файлы конструктора интерфейсов с помощью переменной uiFiles. Значение этой переменной используется без кавычек в списке исходных файлов, передаваемом функции add_executable(). Благодаря этому файлы построителя интерфейса выглядят как еще несколько элементов в списке исходных файлов. Целевое свойство RESOURCE, с другой стороны, содержит единственное значение, и ожидается, что это значение будет списком, разделенным точкой с запятой. Поэтому свойство RESOURCE требует, чтобы значение переменной uiFiles было заключено в кавычки, тогда как вызов add_executable() требует, чтобы оно не заключалось в кавычки.

В приведенном выше примере файл Info.plist будет содержать один из ключей NSMainStoryboardFile, NSMainNibFile или UIMainStoryboardFile (подробнее о значении и правильном использовании этих ключей см. в документации Apple). Такая запись указывает операционной системе, какой элемент пользовательского интерфейса следует использовать при запуске приложения. Простой Info.plist для вышеупомянутого приложения может выглядеть следующим образом:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN"
  "http://www.apple.com/DTDs/PropertyList-1.0.dtd">
<plist version="1.0">
<dict>
    <key>CFBundleDevelopmentRegion</key>
    <string>en</string>
    <key>CFBundleExecutable</key>
    <string>$(EXECUTABLE_NAME)</string>
    <key>CFBundleIconFile</key>
    <string>${MACOSX_BUNDLE_ICON_FILE}</string>
    <key>CFBundleIdentifier</key>
    <string>${MACOSX_BUNDLE_GUI_IDENTIFIER}</string>
    <key>CFBundleInfoDictionaryVersion</key>
    <string>6.0</string>
    <key>CFBundleName</key>
    <string>${MACOSX_BUNDLE_BUNDLE_NAME}</string>
    <key>CFBundlePackageType</key>
    <string>APPL</string>
    <key>CFBundleShortVersionString</key>
    <string>${MACOSX_BUNDLE_SHORT_VERSION_STRING}</string>
    <key>CFBundleVersion</key>
    <string>${MACOSX_BUNDLE_BUNDLE_VERSION}</string>
    <key>LSMinimumSystemVersion</key>
    <string>$(MACOSX_DEPLOYMENT_TARGET)</string>
    <key>NSHumanReadableCopyright</key>
    <string>${MACOSX_BUNDLE_COPYRIGHT}</string>
    <key>NSMainStoryboardFile</key>
    <string>Main</string>
    <key>NSPrincipalClass</key>
    <string>NSApplication</string>
</dict>
</plist>
```

Поле NSMainStoryboardFile имеет значение Main, что указывает на то, что Base.lproj/Main.storyboard UI будет использоваться при запуске приложения. Обратите внимание, что значения некоторых полей предоставляются как переменные CMake с использованием синтаксиса ${}, но CFBundleExecutable и LSMinimumSystemVersion предоставляются с использованием синтаксиса подстановки переменных Xcode $(). Эти два поля будут заполнены самим Xcode на основе другой информации, предоставленной в файле проекта и собираемой схемы. Значение для LSMinimumSystemVersion будет получено из переменной CMAKE_OSX_DEPLOYMENT_TARGET в случае приложения для macOS или из переменной
XCODE_ATTRIBUTE_IPHONEOS_DEPLOYMENT_TARGET целевое свойство для iOS (но обратите внимание, что CMake 3.11 и более поздние версии могут использовать CMAKE_OSX_DEPLOYMENT_TARGET для всех платформ, см. обсуждение в разделе 22.5, "Настройки сборки" далее). Вместо этого проекты могут жестко кодировать значение непосредственно в файле Info.plist, если это удобнее.

Для файлов, которые должны быть включены в пакет приложения, но которые не являются ресурсами в обычном смысле, доступен другой механизм. Такие файлы по-прежнему должны быть указаны в качестве источников для цели, но вместо того, чтобы включать их в свойство RESOURCE цели, каждый из источников имеет свое свойство

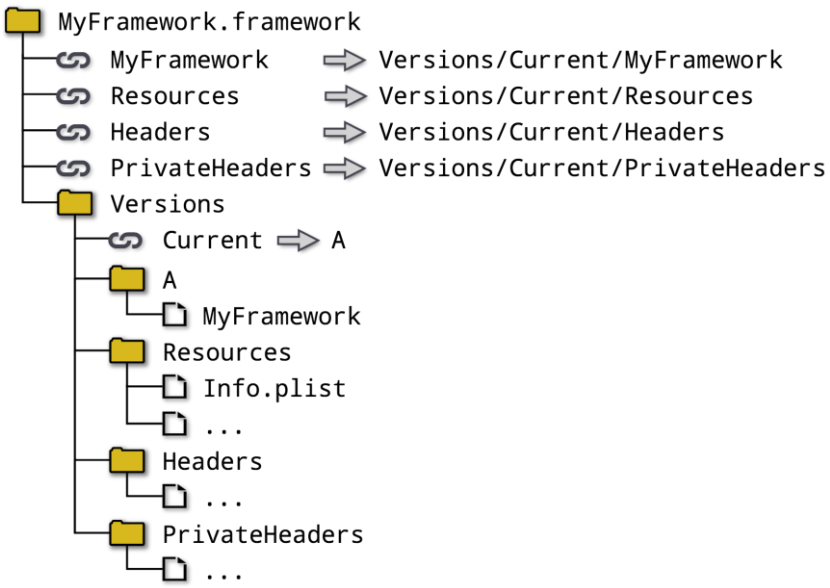
Свойство источника MACOSX_PACKAGE_LOCATION, установленное на местоположение, в которое он должен быть скопирован в пачке. Ожидается, что эти пути будут относительными по отношению к верхней части содержимого пакета. Это можно использовать для копирования файлов в места, не относящиеся к ресурсам, или для полного контроля над целевым каталогом ресурсных файлов, которые не нужно компилировать. Можно также указать каталог в качестве источника и установить его MACOSX_PACKAGE_LOCATION для копирования каталога и его содержимого в пакет, но не документировано, поддерживается ли это формально CMake (каталоги обычно не могут быть указаны в качестве источников). Некоторые примеры помогают продемонстрировать это поведение.

```
add_executable(MyApp MACOSX_BUNDLE
    AppDelegate.m
    AppDelegate.h
    ViewController.m
    ViewController.h
    main.m
    sharedConfig.xml
    nestedResource.dat
    someDir   # Directory, CMake may not formally support this
)

set_source_files_properties(sharedConfig.xml PROPERTIES
    MACOSX_PACKAGE_LOCATION SharedSupport/config
)
set_source_files_properties(nestedResource.dat PROPERTIES
    MACOSX_PACKAGE_LOCATION Resources/private/other
)

# Works, but might not be formally supported
set_source_files_properties(someDir PROPERTIES
    MACOSX_PACKAGE_LOCATION Resources/lotsOfThings
)
```

Особый случай применяется при установке параметра MACOSX_PACKAGE_LOCATION на путь, начинающийся с Resources, и если цель собирается для iOS. Поскольку пакеты приложений для iOS используют уплощенную структуру, CMake будет удалять часть пути Resources. До версии CMake 3.9 это поведение было реализовано некорректно, и не всегда удавалось поместить файл в нужное место.

22.3. Рамки

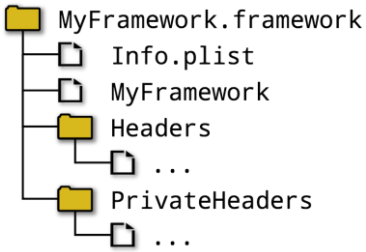
Фреймворки имеют некоторые общие черты с пакетами приложений, но у них есть и ряд уникальных особенностей. Фреймворк содержит основную библиотеку, но, в отличие от пакета приложений, в macOS может быть несколько версий библиотеки. Помимо обычных ресурсов, фреймворки также поддерживают заголовки, и в случае macOS и ресурсы, и заголовки зависят от версии. Типичный пример структуры фреймворка macOS выглядит примерно так.



Верхний уровень фреймворка всегда имеет имя, заканчивающееся на .framework, и обычно единственным содержимым этого каталога верхнего уровня, не связанным симлинками, является подкаталог Versions (исключение составляют зонтичные фреймворки, но они выходят за рамки рассматриваемой здесь поддержки фреймворков). Все остальное на этом уровне обычно является симлинком на что-то в подкаталоге текущей версии.

В каталоге Versions каждая версия библиотеки получает свой собственный подкаталог, имя которого - версия. В большинстве случаев эти каталоги называются просто A, B и т.д. Использование числовых версий является еще одним распространенным соглашением, которое более соответствует тому, как версионируются разделяемые библиотеки на других платформах. Независимо от стиля версионирования, симлинк под названием Current указывает на самую последнюю версию, и она действует как версия по умолчанию для фреймворка. Ожидается, что каждая версия будет иметь каталог Resources, содержащий, по крайней мере, файл Info.plist, в котором содержатся сведения о конфигурации данной версии (об этом подробнее ниже). Там также будет библиотека (обычно это разделяемая библиотека, но она может быть и статической) и часто подкаталоги Headers и, возможно, PrivateHeaders.

Для сравнения, структура на iOS, tvOS и watchOS более плоская и обычно не поддерживает версии:



CMake поддерживает создание фреймворков (только одноверсионных в случае macOS) и предоставляет возможности для работы с деталями версии. Также имеется поддержка файлов Info.plist, которая основана на подходе, аналогичном тому, который используется для пакетов приложений. Первым шагом является определение библиотеки обычным способом, а затем обозначение ее как фреймворка путем установки целевого свойства FRAMEWORK. Большинство фреймворков определяются как разделяемые библиотеки, но начиная с CMake 3.8 статические библиотеки также могут быть созданы как фреймворки. Свойство FRAMEWORK target игнорируется на платформах, отличных от Apple. Только для macOS версия фреймворка может быть указана с помощью целевого свойства FRAMEWORK_VERSION, а если оно опущено, то по умолчанию будет установлена версия A. На платформах Apple, отличных от macOS, свойство FRAMEWORK_VERSION будет

игнорироваться, если оно установлено, создавая ту же самую уплощенную, неверсированную структуру фреймворка, создаваемую Xcode при создании фреймворков для этих платформ.

```
add_library(MyFramework SHARED foo.cpp)
set_target_properties(MyFramework PROPERTIES
    FRAMEWORK          TRUE
    FRAMEWORK_VERSION 5
)
```

Шаблон файла Info.plist задается так же, как и для пакетов приложений, за исключением того, что целевое свойство называется MACOSX_FRAMEWORK_INFO_PLIST (это поддерживается для всех платформ Apple, а не только для macOS):

```
set_target_properties(MyFramework PROPERTIES
    MACOSX_FRAMEWORK_INFO_PLIST "${CMAKE_CURRENT_SOURCE_DIR}/Info.plist"
)
```

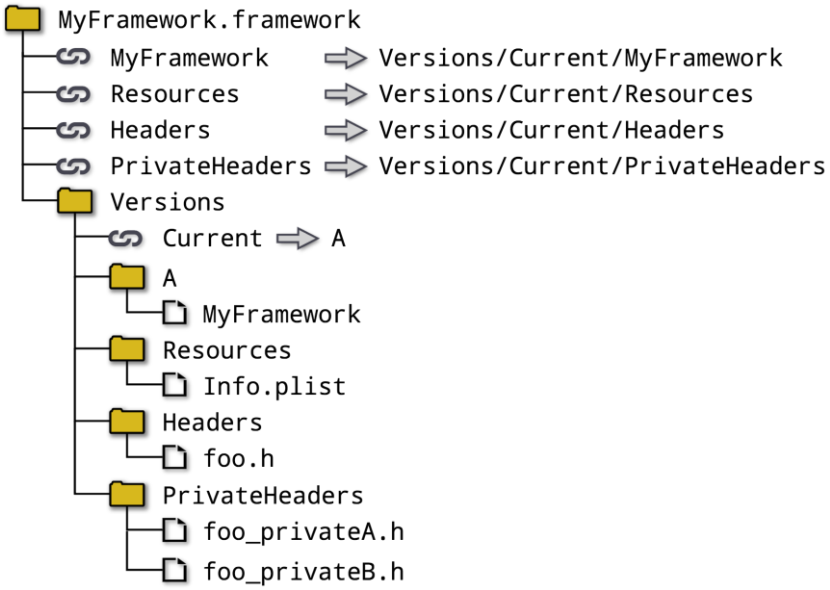
Как и для пакетов приложений, если файл Info.plist фреймворка не предоставлен явно, автоматически генерируется файл по умолчанию. Независимо от того, предоставляет ли проект свой Info.plist или полагается на файл по умолчанию, CMake при копировании его во фреймворк выполнит аналогичную замену, как и для пакетов приложений. Следующие целевые свойства будут заменены там, где на них ссылается файл Info.plist (ожидаемое имя связанного ключа в файле Info.plist также указано):

Property	Info.plist Key
MACOSX_FRAMEWORK_BUNDLE_VERSION	CFBundleVersion
MACOSX_FRAMEWORK_ICON_FILE	CFBundleIconFile
MACOSX_FRAMEWORK_IDENTIFIER	CFBundleIdentifier
MACOSX_FRAMEWORK_SHORT_VERSION_STRING	CFBundleShortVersionString

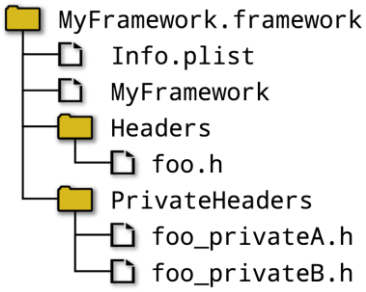
В отличие от пакетов приложений, во многих случаях файла Info.plist фреймворка по умолчанию, скорее всего, будет достаточно, поэтому в проекте обычно можно просто установить вышеуказанные четыре целевых свойства и позволить CMake предоставить соответствующий файл Info.plist.
Фреймворки часто содержат заголовки, связанные с библиотекой фреймворка. Это позволяет рассматривать фреймворк как самодостаточный пакет, на основе которого может быть построено другое программное обеспечение. Заголовки фреймворка разделяются на публичные и частные группы, причем только публичные заголовки предназначены для прямого включения или импорта в потребляющие проекты. Частные заголовки обычно нужны как детали внутренней реализации публичных заголовков, и часто фреймворки вообще не включают никаких частных заголовков. CMake поддерживает указание набора публичных и частных заголовков с помощью целевых свойств PUBLIC_HEADER и PRIVATE_HEADER соответственно. Оба свойства содержат список файлов заголовков, и все упомянутые файлы должны быть также явно указаны в качестве источников для цели. Файлы, перечисленные в PUBLIC_HEADER, будут скопированы в каталог Headers фреймворка с удалением путей, а файлы, перечисленные в PRIVATE_HEADER, будут скопированы в каталог PrivateHeaders, опять же с удалением путей. Если пути должны быть сохранены, эти целевые свойства не могут быть использованы, и заголовки должны быть добавлены с помощью таких методов, как MACOSX_PACKAGE_LOCATION, как описано в предыдущем разделе для произвольных ресурсов.

```
add_library(MyFramework SHARED
    foo.cpp
    foo.h
    foo_privateA.h
    nested/foo_privateB.h
)
set_target_properties(MyFramework PROPERTIES
    FRAMEWORK      TRUE
    PUBLIC_HEADER  foo.h
    PRIVATE_HEADER "foo_privateA.h;nested/foo_privateB.h"
)
```

Приведенный выше пример приведет к следующей структуре на macOS:



Тот же пример на iOS приведет к более плоской структуре:



Обратите внимание, что свойства цели PUBLIC_HEADER и PRIVATE_HEADER также используются при установке целей на платформах, отличных от Apple. Более подробно об этом говорится в разделе 25.2.3, "Цели, специфичные для Apple".

22.4. Загружаемые пучки

Помимо пакетов приложений и фреймворков, Apple также поддерживает загружаемые пакеты для macOS. Они часто используются в качестве плагинов или для предоставления дополнительных функций, которые могут поддерживаться или не поддерживаться во время выполнения. Структура загружаемого пакета такая же, как у пакета приложений, но каталог верхнего уровня обычно имеет расширение .bundle или .plugin (технически

допускается любое расширение). CMake поддерживает создание загружаемых пакетов с помощью библиотечного типа MODULE и целевого свойства BUNDLE. По умолчанию загружаемым пакетам присваивается расширение bundle, но это можно переопределить с помощью целевого свойства BUNDLE_EXTENSION.

```
add_library(MyBundle MODULE ...)
set_target_properties(MyBundle PROPERTIES
    BUNDLE           TRUE
    BUNDLE_EXTENSION plugin
)
```

Все целевые свойства, относящиеся к пакетам приложений, также могут быть использованы для загружаемых пакетов.

22.5. Настройки сборки

При сборке проекта для платформ Apple ряд свойств совместно определяют, для какой платформы будет производиться сборка, и задают минимальные требования к версии платформы. В отличие от других типов генераторов CMake, генератор Xcode позволяет разработчику указывать ряд этих свойств во время сборки, а не знать их во время конфигурирования, что может быть одним из наиболее сложных аспектов для корректной работы как для новичков, так и для опытных пользователей CMake.

Для генераторов с одной конфигурацией целевое устройство точно известно во время конфигурирования, но для Xcode некоторые платформы поддерживают как устройство, так и симуляторы устройства. Более того, некоторые из этих устройств имеют несколько архитектур. В случае iOS это может означать до пяти различных комбинаций целевых платформ. Разные версии Xcode также поставляются с разными версиями iOS SDK, и некоторые разработчики могут даже переносить старые SDK в новые версии Xcode и переключаться между ними. Для того чтобы разработчики могли переключаться между различными целевыми платформами устройств и SDK во время сборки, проекты CMake должны быть осторожны, чтобы не указывать слишком много или неправильно указывать эти детали.

Выбор SDK для iOS, tvOS и watchOS - одна из областей, где многие онлайн-примеры демонстрируют значительную сложность и часто приводят к блокировке разработчика от возможности переключаться между сборками для устройств и симуляторов без повторного запуска CMake. Однако в последних версиях CMake и Xcode указание SDK должно быть очень простым шагом - достаточно установить переменную CMAKE_OSX_SYSROOT в одно из значений iphoneos, appletvos или watchos. Затем Xcode выберет последний SDK для этой платформы и позволит переключаться между сборками для устройств и симуляторов без необходимости повторного запуска CMake. Более того, Xcode автоматически заполнит набор поддерживаемых архитектур на основе выбранного SDK, поэтому проекту не потребуется добавлять дополнительную логику для указания архитектур. Это дает разработчику максимальный контроль над тем, что он хочет собрать, без необходимости повторного запуска CMake. Доступные SDK можно получить, выполнив следующую команду:

```
xcodebuild -showsdks
```

В связи с тем, как CMake выполняет свои тесты компилятора, при работе с платформами Apple, отличными от macOS, необходимо установить еще несколько переменных кэша. По крайней мере, до CMake 3.12, подписание кода может вмешиваться в тесты компилятора, и эти тесты не всегда используют правильный тип цели (например, не пытаются создать пакет, когда в противном случае должны). Для решения этих проблем необходимо также установить переменные CMAKE_MACOSX_BUNDLE и

CMAKE_XCODE_ATTRIBUTE_CODE_SIGNING_REQUIRED. Для того, чтобы тесты компилятора подхватили правильные детали, CMAKE_OSX_SYSROOT, CMAKE_MACOSX_BUNDLE и CMAKE_XCODE_ATTRIBUTE_CODE_SIGNING_REQUIRED должны быть установлены очень рано на этапе конфигурирования. Лучше всего это сделать с помощью файла цепочки инструментов, минимальная версия которого выглядит следующим образом:

```
set(CMAKE_MACOSX_BUNDLE YES)
set(CMAKE_XCODE_ATTRIBUTE_CODE_SIGNING_REQUIRED NO)
set(CMAKE_OSX_SYSROOT iphoneos)
```

По умолчанию цель развертывания проекта будет установлена на самую последнюю версию, поддерживаемую SDK или хост-системой. Это часто бывает нежелательно, поскольку проекты обычно хотят оставаться совместимыми с определенной минимальной версией ОС. Для macOS свойство цели OSX_DEPLOYMENT_TARGET определяет минимальную версию macOS, которую будет поддерживать цель. Для этого целевого свойства можно задать значение по умолчанию с помощью переменной CMAKE_OSX_DEPLOYMENT_TARGET, но оно должно быть задано до вызова первой команды project(). Кроме того, CMAKE_OSX_DEPLOYMENT_TARGET должна быть кэш-переменной, если она задается непосредственно в файле CMakeLists.txt верхнего уровня, иначе она будет перезаписана проверками компилятора, выполняемыми командой project(). Альтернативной стратегией является использование файла toolchain и установка CMAKE_OSX_DEPLOYMENT_TARGET в нем, но использование файлов toolchain для сборок macOS будет довольно редким, и эта переменная должна определяться проектом, а не разработчиком. Еще одним подходом может быть установка кэш-переменной CMAKE_OSX_DEPLOYMENT_TARGET в командной строке cmake, но это также возлагает на разработчика ответственность за то, чтобы не забыть установить ее и предоставить правильное значение, что делает этот подход менее привлекательным.

До версии CMake 3.11 при выборе платформ, отличных от macOS, переменная CMAKE_OSX_DEPLOYMENT_TARGET не имеет значения. Для управления минимальной версией цели развертывания для iOS до CMake

3.11, используйте вместо него целевое свойство XCODE_ATTRIBUTE_IPHONEOS_DEPLOYMENT_TARGET. Значение по умолчанию для этого целевого свойства можно установить с помощью переменной CMAKE_XCODE_ATTRIBUTE_IPHONEOS_DEPLOYMENT_TARGET. В отличие от macOS, эта переменная может быть установлена после первого вызова project(). Из CMake

В версиях 3.11 и далее CMAKE_OSX_DEPLOYMENT_TARGET можно использовать для определения минимальной версии цели развертывания для любой из платформ Apple, а не только macOS. Если для цели определены оба свойства OSX_DEPLOYMENT_TARGET и XCODE_ATTRIBUTE_IPHONEOS_DEPLOYMENT_TARGET, последнее имеет приоритет при использовании генератора Xcode.

```
# Set the deployment target for macOS with any CMake version, or all Apple
# platforms when using CMake 3.11 or later
cmake_minimum_required(VERSION 3.9)

# Must be before first call to project()
set(CMAKE_OSX_DEPLOYMENT_TARGET 10.11)
project(AppleProject)
```

```
# Set the deployment target for iOS with any CMake version.

# Set defaults for all targets added hereafter within this directory scope or below
set(CMAKE_XCODE_ATTRIBUTE_IPHONEOS_DEPLOYMENT_TARGET 9.0)

# Build an app with the deployment target explicitly set
add_executable(MyApp MACOSX_BUNDLE ...)
set_target_properties(MyApp PROPERTIES XCODE_ATTRIBUTE_IPHONEOS_DEPLOYMENT_TARGET 10.0)
```

В случае iOS проекты, скорее всего, также захотят указать семейства устройств, на которые они нацелены. Apple обозначает устройства целочисленными значениями, указанными в атрибуте TARGETED_DEVICE_FAMILY. Для iOS допустимыми значениями являются 1 для iPhone (и, технически, iPod touch тоже) или 2 для iPad. Если приложение должно поддерживать и iPhone, и iPad, то укажите оба значения через запятую. Если этот атрибут не задан, то по умолчанию он будет равен 1. Xcode будет использовать это значение для автоматического добавления записи UIDeviceFamily в файл Info.plist приложения, поэтому избегайте установки этой записи в пользовательском Info.plist, поставляемом проектом.

```
# An app that supports only iPad
add_executable(MyiPadApp MACOSX_BUNDLE ...)
set_target_properties(MyiPadApp PROPERTIES
    XCODE_ATTRIBUTE_TARGETED_DEVICE_FAMILY 2
)

# An app that supports both iPhone and iPad
add_executable(RunEverywhereApp MACOSX_BUNDLE ...)
set_target_properties(RunEverywhereApp PROPERTIES
    XCODE_ATTRIBUTE_TARGETED_DEVICE_FAMILY 1,2
)
```

Выше перечислены основные настройки, связанные со сборкой, которые необходимо определить для большинства проектов Apple. Для простых неподписанных приложений macOS их может быть достаточно, но для большинства проектов потребуется дополнительная настройка для подписи продуктов сборки, прежде чем они смогут быть полезными.

22.6. Подписание кодекса

Функциональность Xcode, связанная с подписанием кода, значительно изменилась за последние несколько основных выпусков Xcode. В последнее время переход к автоматическому управлению подписанием и предоставлением кода упростил получение подписанных приложений, созданных с помощью CMake, однако для установки соответствующих свойств и переменных все еще требуется понимание процесса подписания.

Следует отметить, что в Xcode 8 способ работы автоматического подписания и инициализации значительно изменился, в результате чего многие примеры, демонстрирующие методы для Xcode 7 и более ранних версий, перестали отражать лучшую практику. Эта глава посвящена текущему процессу автоматической подписи и инициализации.

Чтобы автоматическое подписание и инициализация работали, приложение должно иметь действительный идентификатор пакета, а также необходимо предоставить две другие ключевые части информации: идентификатор команды разработчиков и идентификатор подписи кода. Они должны быть указаны в виде атрибутов Xcode, которые, как обычно, задаются на отдельных объектах через свойства объекта или через переменные CMake для указания значений по умолчанию для соответствующих свойств объекта. Поскольку оба параметра обычно должны быть одинаковыми для всей сборки, обычно рекомендуется задавать их как переменные в верхней части проекта, а не для каждой цели.

XCODE_ATTRIBUTE_DEVELOPMENT_TEAM целевое свойство или соответствующий

Переменная CMAKE_XCODE_ATTRIBUTE_DEVELOPMENT_TEAM должна быть установлена на ID команды разработчиков, который представляет собой короткую строку, обычно состоящую примерно из 10 символов. Наиболее удобным подходом обычно является установка переменной CMAKE_XCODE_ATTRIBUTE_DEVELOPMENT_TEAM в самом начале самого верхнего файла CMakeLists.txt, обычно сразу после первой команды project(). В зависимости от проекта, разработчику может понадобиться или не понадобиться возможность изменять это значение. Например, если проект представляет собой программное обеспечение компании, которое всегда будет создаваться сотрудником, то ID команды, скорее всего, никогда не изменится, в то время как проект с открытым исходным кодом, доступный широкой публике, почти наверняка будет создаваться разработчиками с собственным ID команды разработчиков . Если идентификатор команды никогда не должен меняться , определяя CMAKE_XCODE_ATTRIBUTE_DEVELOPMENT_TEAM в качестве обычной переменной вполне достаточно, но если предполагается, что разработчику может понадобиться ее изменить, ее следует определить как кэш-переменную, чтобы можно было задать значение по умолчанию, но разработчики могли его отменить без редактирования файла CMakeLists.txt.

XCODE_ATTRIBUTE_CODE_SIGN_IDENTITY целевое свойство или соответствующий

Переменная CMAKE_XCODE_ATTRIBUTE_CODE_SIGN_IDENTITY задает идентификатор подписи. Начиная с Xcode 8, это всегда должна быть строка Mac Developer для приложений macOS или iPhone Developer для приложений iOS, tvOS или watchOS. Эти значения направляют Xcode на выбор наиболее подходящего идентификатора подписи для указанной команды разработчиков. В необычных обстоятельствах идентификатор подписи может быть установлен в строку, которая идентифицирует определенный идентификатор подписи кода в связке ключей разработчика, но в этом случае разработчик должен убедиться, что этот идентификатор принадлежит указанной команде разработчиков.

Следующий пример показывает, как может быть структурирован CMakeLists.txt для приложения macOS, которое позволяет разработчику изменять идентификатор команды и идентификатор подписи:

```
cmake_minimum_required(VERSION 3.9)
project(macOSexample)

set(CMAKE_XCODE_ATTRIBUTE_DEVELOPMENT_TEAM   "ABC12345DE"    CACHE STRING "")
set(CMAKE_XCODE_ATTRIBUTE_CODE_SIGN_IDENTITY "Mac Developer" CACHE STRING "")
```

Для приложения iOS, в котором не предполагается изменение идентификатора команды, но разработчик может захотеть контролировать идентификатор подписи (например, для проверки другого идентификатора в связке ключей), только идентификатор должен быть переменной кэша:

```
cmake_minimum_required(VERSION 3.9)
project(iOSexample)

set(CMAKE_XCODE_ATTRIBUTE_DEVELOPMENT_TEAM   "ABC12345DE")
set(CMAKE_XCODE_ATTRIBUTE_CODE_SIGN_IDENTITY "iPhone Developer" CACHE STRING "")
```

При настройке, как описано выше, Xcode автоматически выберет подходящий профиль инициализации. Если подходящего профиля не существует, среда разработки Xcode может автоматически создать его (эта функциональность является особенностью среды разработки и недоступна для сборок из командной строки). Такая автоматическая инициализация является значительным улучшением по сравнению с предыдущими версиями Xcode, в которых профили инициализации необходимо было создавать вручную через онлайн-портал разработчика.

В предыдущем разделе CMAKE_XCODE_ATTRIBUTE_CODE_SIGNING_REQUIRED была установлена на NO в файле цепочки инструментов для iOS, но эта переменная игнорируется, когда CMAKE_XCODE_ATTRIBUTE_CODE_SIGN_IDENTITY установлена. Может возникнуть соблазн перенести детали подписи кода в файл цепочки инструментов, чтобы избежать необходимости устанавливать CMAKE_XCODE_ATTRIBUTE_CODE_SIGN_IDENTITY, но учтите, что это означает, что тест try-compile, который CMake выполняет как часть первой команды project(), потребует действительного профиля инициализации, который, в свою очередь, потребует действительного идентификатора пакета. В общем случае нежелательно, чтобы такие идентификаторы пакетов и профили инициализации создавались в командной учетной записи. Тесты try-compile не нуждаются в подписи кода, поэтому файл toolchain не следует использовать для глобального включения подписи.

Приложения Apple также имеют соответствующий набор прав доступа. Они определяют, какие функции операционная система позволит использовать приложению, например, Siri, push-уведомления и так далее. В настройках проекта в Xcode IDE пользователи могут перейти на вкладку *Capabilities* целевого приложения и включить необходимые возможности. Затем соответствующие полномочия включаются в plist-файл, который автоматически генерируется Xcode, цель связывается с любыми необходимыми фреймворками, а возможности добавляются к идентификатору приложения в учетной записи команды. При использовании проекта, созданного с помощью CMake, вкладка *Capabilities* фактически обходится стороной. Вместо этого ожидается, что проект CMake предоставит свой собственный plist-файл полномочий, если полномочий по умолчанию недостаточно. Проект должен сам обработать связывание всех необходимых фреймворков, и никаких изменений в идентификатор приложения не вносится. На практике для многих приложений это довольно мягкие ограничения, и только связывание фреймворков создает некоторые сложности.

Указание прав доступа осуществляется путем установки целевого свойства XCODE_ATTRIBUTE_CODE_SIGN_ENTITLEMENTS в имя соответствующего файла прав доступа следующим образом:

```
set_target_properties(myApp PROPERTIES
    XCODE_ATTRIBUTE_CODE_SIGN_ENTITLEMENTS
    ${CMAKE_CURRENT_LIST_DIR}/myApp.entitlements
)
```

Например, файл прав доступа, который добавляет Siri к правам доступа по умолчанию, может быть довольно простым:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN"
 "http://www.apple.com/DTDs/PropertyList-1.0.dtd">
<plist version="1.0">
<dict>
    <key>com.apple.developer.siri</key>
    <true/>
</dict>
</plist>
```

Если приложение ссылается на общие фреймворки, которые также собираются проектом, не включайте подпись кода для этих фреймворков. Рекомендуемый способ добавления таких фреймворков в пакет приложения - через этап сборки *Embed Frameworks в* Xcode с включенной опцией *Code sign on copy*, но, к сожалению, CMake не поддерживает это напрямую (см. раздел 22.8, "Ограничения" для обсуждения ограничений на поддержку фреймворков в CMake).

22.7. Создание и экспорт архивов

Чтобы распространить приложение через App Store, корпоративный портал распространения или специальное распространение, сначала необходимо создать архив. Хотя CMake не создает цель сборки для создания такого архива, инструмент xcodebuild может быть использован с проектом, созданным CMake, для выполнения этой задачи. Действие сборки архива требует всего нескольких опций, чтобы иметь возможность собрать необходимые цели для выпуска и создать архив. Есть несколько способов указать, что именно архивировать, но довольно простой подход заключается в том, чтобы назвать проект, схему и имя выходного файла:

```
xcodebuild archive \
           -project MyProject.xcodeproj \
           -scheme MyApp \
           -archivePath MyApp.xcarchive
```

CMake создает файл .xcodeproj при использовании генератора Xcode. До CMake 3.9 пользователю приходилось загружать проект в IDE Xcode, чтобы создать схемы сборки. Это представляло проблему для сборок непрерывной интеграции без головы, когда IDE недоступна, поэтому для решения этой проблемы в CMake 3.9 была введена переменная CMAKE_XCODE_GENERATE_SCHEME в качестве экспериментальной функции. Когда эта переменная имеет значение true, CMake также будет генерировать файлы схемы для сборки, что позволяет указать имя целевого приложения в опции -scheme, и задача архивирования получит всю необходимую информацию. Приведенная выше команда создаст целевой проект MyApp для конфигурации Release для всех поддерживаемых архитектур, подпишет его (по-прежнему с идентификатором подписи разработчика), а затем создаст архив с именем MyApp.archive в текущем каталоге.

Архивирование может завершиться неудачей, если некоторые атрибуты установки не установлены должным образом. Документация для разработчиков Apple содержит несколько рекомендаций по устранению неполадок, которые могут помочь справиться с наиболее распространенными ситуациями, некоторые из наиболее важных - убедиться, что атрибуты INSTALL_PATH и SKIP_INSTALL цели установлены правильно для типа цели. В проекте CMake, направленном на создание подписанного приложения для распространения, свойство XCODE_ATTRIBUTE_SKIP_INSTALL цели должно быть установлено в YES для библиотек и встроенных фреймворков и в NO для приложений. Если свойство установлено в NO, то в программе

XCODE_ATTRIBUTE_INSTALL_PATH также должен быть указан, и обычно ему должно быть присвоено значение $(LOCAL_APPS_DIR). Несоблюдение этого совета обычно приводит к тому, что шаг архивации создает общий архив, а не архив приложения.

```
# Apps must have install step enabled
set_target_properties(macOSApp PROPERTIES
    XCODE_ATTRIBUTE_SKIP_INSTALL  NO
    XCODE_ATTRIBUTE_INSTALL_PATH  "$(LOCAL_APPS_DIR)"
)
```

После создания архива приложения его необходимо экспортировать, чтобы он был готов к распространению. Это достигается еще одним вызовом инструмента xcodebuild, на этот раз с предоставлением только что созданного архива, файла options plist и места для записи выходных данных. Основная форма команды выглядит следующим образом:

```
xcodebuild -exportArchive \
           -archivePath myApp.xcarchive \
           -exportOptionsPlist exportOptions.plist \
           -exportPath Products
```

Опция -archivePath указывает на архивный файл, созданный предыдущим вызовом xcodebuild, а опция -exportPath определяет каталог, в котором будет создан конечный выходной файл. Все остальное, что касается шага экспорта, определяется plist-файлом, переданным опции -exportOptionsPlist. Полный набор поддерживаемых ключей можно найти в справочной документации инструмента (xcodebuild -help), но минимальный plist-файл может выглядеть следующим образом:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN"
 "http://www.apple.com/DTDs/PropertyList-1.0.dtd">
<plist version="1.0">
<dict>
    <key>method</key>
    <string>app-store</string>
</dict>
</plist>
```

Метод определяет предполагаемый канал распространения и, как ожидается, является одним из следующих:

- app-store
- ad-hoc
- enterprise
- development
- developer-id
- package

По умолчанию используется метод разработки, но более вероятно, что основными методами, представляющими интерес, будут app-store, enterprise или ad-hoc. При экспорте архива инструмент повторно подпишет приложение и выберет соответствующий идентификатор подписи дистрибутива на основе

выбранного метода. Предполагается, что разработчик уже создал/скачал соответствующий идентификатор подписи дистрибутива и профиль инициализации (проще всего это сделать в IDE Xcode, но можно и вручную для серверов непрерывной интеграции и т.д.).

22.8. Ограничения

По крайней мере, до версии 3.11 поддержка фреймворков в CMake имела несколько недостатков. Во многом это связано с тем, как фреймворки (и даже обычные библиотеки) включаются в проекты Xcode, где вместо определения соответствующей фазы сборки *Link Binary With Libraries*, CMake жестко кодирует связывание непосредственно в целевой атрибут *Other Linker Flags*. Это соответствует тому, как CMake связывает библиотеки и фреймворки при использовании других типов генераторов, но эти генераторы не имеют дополнительных функций обработки фреймворков и подписи кода, которые есть в Xcode. CMake действительно пытается определить, является ли библиотека, которую он хочет связать, фреймворком, используя -framework someLib вместо -lsomeLib или /path/to/someLib.dylib в командной строке компоновщика для тех, которые он идентифицирует как фреймворки, но это не делает Xcode осведомленным о фреймворке для чего-либо другого, кроме связывания.

Для статических фреймворков реализация CMake в основном работает, но для разделяемых фреймворков возникают проблемы. Встраивая детали непосредственно во флаги компоновщика, Xcode не получает полного представления о фреймворке и не сможет правильно обработать его при создании архива приложения или подписи кода. В частности, фреймворк не будет установлен вместе с целевой ссылкой на него, поскольку не определена связанная с ним фаза сборки *Copy Files*, и именно во время копирования обычно выполняется подписание кода встроенного фреймворка.

При текущем поведении CMake выбор, доступный проектам, ограничен. Можно вообще не использовать несистемные общие фреймворки, но это имеет очевидные недостатки. Проект может потребовать от разработчика выполнить некоторые изменения проекта вручную после запуска CMake, чтобы добавить фреймворки вручную, но это явно хрупко и исключает возможность сборки в безголовой среде, например, в системе непрерывной интеграции. Более жизнеспособным путем было бы определение сценария для изменения файла проекта Xcode после запуска CMake, или определение пользовательских команд или шагов после сборки в проекте CMake для имитации действий, которые обычно выполняет проект Xcode, если он знает о встроенном фреймворке(ах). Ни один из этих вариантов не приносит особого удовлетворения, и все они противоречат самой природе того, что CMake должен делать сам по себе. Даже подход с использованием пользовательских сценариев или шагов после сборки с достаточной вероятностью будет противоречить будущим улучшениям CMake, в которых эти недостатки могут быть устранены.

Работа CMake с правами доступа также довольно рудиментарна. Она не дотягивает до автоматизации, которую обеспечивает среда Xcode IDE на вкладке свойств цели *Capabilities*, где включение определенной возможности также заботится о добавлении всех необходимых фреймворков и автоматически обновляет данные идентификатора приложения по мере необходимости. Поддержка CMake по-прежнему позволяет указывать все возможности, но этот процесс полностью ручной. Проект отвечает за определение прав в формате raw plist, а также должен сам вручную подключить все необходимые фреймворки, что, как уже говорилось, не очень хорошо обрабатывается CMake. Тем не менее, работа с правами, по крайней мере, возможна без обходных путей или шагов, которые становятся слишком обременительными. Любые фреймворки, требуемые правами, предоставляются системой, поэтому их не нужно встраивать в приложение, что позволяет избежать большинства недостатков работы с фреймворками.

На более практическом, повседневном уровне следует сказать несколько слов предостережения относительно поведения CMake, которое не всегда очевидно. В генераторе Xcode, когда CMake пишет проект Xcode, он создает служебную цель под названием ZERO_CHECK. Большинство других целей в проекте зависят от

ZERO_CHECK, и его единственная цель - выяснить, нужно ли повторно запускать CMake перед выполнением остальных частей сборки. К сожалению, если CMake повторно запускается с помощью ZERO_CHECK, остальная часть сборки по-прежнему использует старые данные проекта, что может привести к тонким ошибкам, связанным со сборкой целей с устаревшими настройками. Пересборка во второй раз всегда должна гарантировать, что все такие неправильно собранные цели будут собраны правильно, но это можно легко пропустить. Разработчики могут захотеть явно собрать цель ZERO_CHECK или повторно запустить CMake после изменения файлов CMakeLists.txt или чего-либо еще, что может привести к автоматическому повторному запуску CMake, или просто выполнить сборку дважды.

Более тонкая проблема, связанная с ZERO_CHECK, существует, если проект содержит несколько вызовов команды project(). Цели, определенные ниже второго или более поздних вызовов project(), могут не иметь правильно установленной зависимости от ZERO_CHECK. Переменная CMAKE_XCODE_GENERATE_TOP_LEVEL_PROJECT_ONLY может быть установлена в true для предотвращения этой проблемы, что также будет иметь полезный побочный эффект ускорения этапа CMake, но поддержка этой переменной была добавлена только в CMake 3.11.

22.9. Рекомендуемые практики

CMake способен работать с проектами, предназначенными для платформ Apple, однако следует внимательно изучить ограничения. Если приложения должны быть подписаны, то использование любых несистемных общих фреймворков потребует ручного написания сценариев и пользовательских шагов сборки для получения желаемого конечного результата. Если общие фреймворки не нужны, то функциональности CMake должно быть достаточно, и он, как правило, автоматизирует процесс без особых усилий, если используется генератор Xcode. Другие генераторы, такие как Makefiles или Ninja, подойдут для создания неподписанного приложения macOS, но для других платформ или для подписанных приложений эти генераторы обычно не имеют некоторых функций, необходимых для легкого создания конечного пакета для распространения. За исключением разработки неподписанных приложений для macOS, настоятельно рекомендуется использовать генератор Xcode для разработки в Apple.

Большая часть информации, доступной в онлайн-учебниках и примерах, относительно устарела, когда речь идет об использовании CMake для платформ Apple. В частности, очень часто можно встретить довольно сложные файлы цепочки инструментов для iOS, но большая часть логики, содержащейся в таких файлах цепочки инструментов, либо уже не нужна, либо должна быть перенесена в сам проект. В Xcode 8 или более поздней версии проекты должны стремиться использовать автоматическую подпись и инициализацию, если это возможно, поскольку это значительно упрощает процесс подписания. Это также означает, что для минимального файла инструментария достаточно установить CMAKE_MACOSX_BUNDLE, CMAKE_XCODE_ATTRIBUTE_CODE_SIGNING_REQUIRED и CMAKE_OSX_SYSROOT, чтобы получить рабочую сборку с поддержкой подписания и распространения кода. Другая логика, связанная с настройками проекта Xcode, конфигурацией для конкретного устройства или платформы и т.д., должна находиться в самом проекте.

В руководствах и примерах часто указывается целевая архитектура путем установки переменной CMAKE_OSX_ARCHITECTURES. При использовании генератора Xcode с проектами, ориентированными на iOS, watchOS или tvOS, это нежелательно, поскольку не позволяет разработчику свободно переключаться между сборками на устройстве и симуляторе. Целевая архитектура выбирается во время сборки при работе в Xcode IDE или при сборке в командной строке. Поэтому в проектах, как правило, не следует устанавливать CMAKE_OSX_ARCHITECTURES и вместо этого позволить Xcode предоставить стандартный набор архитектур, основанный на выбранном SDK. SDK определяется CMAKE_OSX_SYSROOT, но важно, что Xcode способен распознать подходящий симулятор, когда выбран SDK устройства. Если установить CMAKE_OSX_SYSROOT на что-то вроде iphoneos, например, разработчику будут доступны сборки как устройства, так и симулятора. Более

того, хотя можно указать версию SDK как часть значения CMAKE_OSX_SYSROOT, обычно в этом нет смысла. Гораздо более вероятно, что цель развертывания должна быть задана через MACOSX_DEPLOYMENT_TARGET или XCODE_ATTRIBUTE_IPHONEOS_DEPLOYMENT_TARGET, а не версия SDK. Именно цель развертывания в конечном итоге определяет, сможет ли приложение работать на этой цели, и это не зависит от SDK, используемого для сборки (при условии, что SDK поддерживает эту цель развертывания, конечно). Поскольку по умолчанию будет использоваться последняя доступная версия SDK, требование использовать при сборке определенную версию SDK мало что дает, а может даже навредить. Если указана конкретная версия SDK, она может быть доступна не на всех машинах разработчиков, поскольку это зависит от того, какая версия Xcode используется. Некоторые разработчики переносят старые SDK на более новые версии Xcode, чтобы попытаться обойти эту проблему, но в этом нет необходимости.

В некоторых примерах CMAKE_XCODE_ATTRIBUTE_ONLY_ACTIVE_ARCH также устанавливается в true, чтобы в Xcode IDE можно было собирать только выбранную в данный момент архитектуру. Опять же, это решение, которое обычно должно быть оставлено на усмотрение разработчика во время сборки, а не навязано CMake. Это также иногда используется для скриптовых сборок, когда известно, что должна быть собрана только одна конкретная платформа, но даже в этом случае архитектура может быть указана в качестве опции командной строки, так что пользы от этого немного.

Одна из ситуаций, когда имеет смысл ограничить сборку только одной архитектурой, - это когда проект содержит цели, связанные с библиотеками или фреймворками, которые не предоставляют жирных двоичных файлов (т.е. они были собраны только для одной целевой платформы). В этом случае, поскольку эти библиотеки или фреймворки поддерживают только одну платформу, проект может быть собран только для этой платформы. Аналогично, при использовании команд find_library() или find_package() (рассматриваются в следующей главе), эти команды изначально предполагают, что они собираются для одной платформы, поэтому они не пытаются определить найденные вещи таким образом, чтобы поддерживать переключение между несколькими целевыми платформами.

Некоторые проекты могут использовать функцию установки CMake вместо того, чтобы полагать, что Xcode сделает все необходимое для дистрибутивного пакета во время сборки. Для таких случаев свойство IOS_INSTALL_COMBINED target можно установить в true, чтобы собрать обе версии цели - для устройства и симулятора - и объединить их в один жирный бинарник на этапе установки. Это может быть альтернативным путем, если использование генератора Xcode по каким-то причинам нежелательно или если проект структурирован так, чтобы следовать более платформо-агностичной модели сборки и установки CMake. Обсуждение этой темы см. в разделе 25.2.3, "Цели, специфичные для Apple".

Вывод данных сборки из Xcode может быть довольно многословным, поэтому разработчики могут использовать такой инструмент, как xcpretty, чтобы скрыть большую часть деталей (это более характерно для скриптовых сборок, чтобы уменьшить размер журнала). К сожалению, этот инструмент обычно скрывает вывод любых пользовательских шагов CMake после сборки, даже если эти шаги вызывают ошибку сборки. Когда такие пользовательские шаги не работают, бывает очень трудно определить причину сбоя, поэтому рекомендуется либо избегать использования этого инструмента, либо, по крайней мере, облегчить его отключение в сценариях, чтобы помочь диагностировать проблемы сборки. Опция -quiet в команде xcodebuild может быть альтернативой для уменьшения вывода журнала без скрытия предупреждений или ошибок, но она также может скрыть слишком много деталей.

Часть III: Большая картина

Для немногих счастливчиков проект может быть независимым от всего остального и должен удовлетворять лишь незначительным ограничениям качества или, возможно, вообще не удовлетворять их для экспериментов, проводимых на скорую руку. Более вероятный сценарий заключается в том, что в какой-то момент проекту необходимо выйти за рамки своего изолированного существования и взаимодействовать с внешними организациями. Это происходит в двух направлениях:

Зависимости

Проект может зависеть от других предоставленных извне файлов, библиотек, исполняемых файлов, пакетов и так далее.

Потребители

Другие проекты могут захотеть использовать проект различными способами. Некоторые могут захотеть включить его на уровне исходных текстов, другие могут ожидать наличия предварительно собранного бинарного пакета. Другая возможность - предположение, что проект установлен где-то в системе.

Предоставление проекта в виде отдельного пакета или для использования другими проектами также подразумевает ожидание определенного уровня качества. Автоматизированное тестирование обычно является критической частью любой надежной стратегии поставки программного обеспечения, что означает, что должно быть легко определить и выполнить тесты, а также сообщить о результатах.

Набор инструментов CMake обеспечивает помощь во всем вышеперечисленном. Он предоставляет команды, которые работают на более низком уровне для поиска отдельных файлов, библиотек и т.д., а также модули, которые основываются на этих командах для создания более высокоуровневой точки входа для управления зависимостями. Фреймворк CTest предоставляет богатый набор возможностей автоматизированного тестирования, а CPack значительно облегчает процесс создания пакетов в различных форматах. Эта часть книги охватывает эти внешне ориентированные темы, показывая, как получить максимальную отдачу от того, что предлагает CMake, а также освещая распространенные ошибки и подводные камни.

Последняя глава этой части книги возвращает читателя к размышлениям о том, как организовать проект. Для того чтобы сделать это хорошо, необходимо знать как особенности уровня сборки, так и то, как проект будет взаимодействовать с другими проектами. С использованием знаний, полученных в предыдущих главах, в ней показано, как структурировать и определить проект, чтобы он был гибким, надежным и удобным для разработчиков.

Глава 23. Поиск вещей

Проект хотя бы скромного размера, скорее всего, будет полагаться на вещи, предоставляемые чем-то за пределами самого проекта. Например, он может ожидать доступности определенной библиотеки или инструмента, или ему может понадобиться знать расположение определенного файла конфигурации или заголовка для используемой библиотеки. На более высоком уровне проект может захотеть найти полный пакет, который потенциально определяет целый ряд вещей, включая цели, функции, переменные и почти все остальное, что может определить обычный проект CMake.

Чтобы помочь в этом, CMake предоставляет множество функций, позволяющих проектам находить различные вещи и даже облегчать поиск и включение себя в другие проекты. Различные команды find_... () предоставляют возможность поиска определенных файлов, библиотек или программ, или даже целого пакета. Модули CMake также добавляют возможность использования pkg-config для предоставления информации о внешних пакетах, а другие модули облегчают написание файлов пакетов для использования другими проектами. В этой главе

рассматривается поддержка CMake для поиска чего-либо, уже имеющегося в файловой системе. Возможность загрузки недостающих зависимостей рассматривается в главе 27, "*Внешнее содержимое*", а подготовка проекта к тому, чтобы его могли найти другие проекты, рассматривается в разделе 25.7, "Написание файла конфигурационного пакета".

Основная идея поиска чего-либо относительно проста, но, как станет очевидным, детали того, как осуществляется поиск, могут быть довольно сложными. Во многих случаях поведение по умолчанию является подходящим, но понимание мест поиска и их порядка может позволить проектам адаптировать поиск для учета нестандартного поведения и необычных обстоятельств.

23.1. Поиск файлов и путей

Концептуально, самая основная задача поиска - найти конкретный файл, и самый прямой способ добиться этого - команда find_file(). Она также служит хорошим введением в целое семейство команд find_...(), поскольку все они имеют много общих опций и схожее поведение. Полный синтаксис этой команды выглядит следующим образом:

```
find_file(outVar
    name | NAMES name1 [name2...]
    [HINTS path1 [path2...] [ENV var]...]
    [PATHS path1 [path2...] [ENV var]...]
    [PATH_SUFFIXES suffix1 [suffix2 ...]]
    [NO_DEFAULT_PATH]
    [NO_PACKAGE_ROOT_PATH]
    [NO_CMAKE_PATH]
    [NO_CMAKE_ENVIRONMENT_PATH]
    [NO_SYSTEM_ENVIRONMENT_PATH]
    [NO_CMAKE_SYSTEM_PATH]
    [CMAKE_FIND_ROOT_PATH_BOTH |
     ONLY_CMAKE_FIND_ROOT_PATH |
     NO_CMAKE_FIND_ROOT_PATH]
    [DOC "description"]
)
```

Команда может искать одно имя файла или может быть задан список имен с помощью опции NAMES. Список может быть полезен, когда искомый файл может иметь несколько вариаций имени, например, разные дистрибутивы операционных систем выбирают разные соглашения об именовании, включают номера версий или нет, учитывают изменение имени файла от одного выпуска к другому и так далее. Имена должны быть перечислены в предпочтительном порядке, поскольку поиск будет остановлен на первом найденном имени (для конкретного имени проверяется весь набор мест поиска, прежде чем переходить к следующему имени). При указании имен, содержащих некоторую форму нумерации версий, документация CMake рекомендует перечислять имена, не содержащие сведений о версии, впереди тех, которые содержат такие сведения, чтобы локально собранные файлы с большей вероятностью были найдены раньше, чем файлы, предоставленные операционной системой.

Поиск будет проводиться по набору мест, проверенных в четко определенном порядке. Большинство мест имеют соответствующую опцию, которая заставляет пропустить это место, если опция присутствует, что позволяет адаптировать поиск по необходимости. Порядок поиска представлен в следующей таблице:

Location	Skip Option
Package root variables	NO_PACKAGE_ROOT_PATH
Cache variables (CMake-specific)	NO_CMAKE_PATH
Environment variables (CMake-specific)	NO_CMAKE_ENVIRONMENT_PATH
Paths specified via the HINTS option	
Environment variables (system-specific)	NO_SYSTEM_ENVIRONMENT_PATH
Cache variables (platform-specific)	NO_CMAKE_SYSTEM_PATH
Paths specified via the PATHS option	

Корневые переменные пакета

Первое место поиска применяется только в том случае, если find_file() вызывается из модуля Find (о нем речь пойдет далее в этой главе). Первоначально он был добавлен в качестве места поиска в CMake 3.9.0, но был удален в 3.9.1 из-за проблем с обратной совместимостью. Затем он был снова добавлен в CMake 3.12 с устранением проблем. Дальнейшее обсуждение этого места поиска отложено до раздела 23.5, "Поиск пакетов", где его использование более уместно.

Переменные кэша (специфические для CMake)

Расположение специфических для CMake переменных кэша происходит от переменных кэша CMAKE_PREFIX_PATH, CMAKE_INCLUDE_PATH и CMAKE_FRAMEWORK_PATH. Из них CMAKE_PREFIX_PATH, пожалуй, самая удобная, так как ее установка работает не только для find_file(), но и для всех остальных команд find_...(). Она представляет собой базовую точку, ниже которой ожидается типичная структура каталогов bin, lib, include и так далее, и каждая команда find_...() добавляет свой собственный подкаталог для построения путей поиска. В случае find_file(), для каждой записи в
CMAKE_PREFIX_PATH, будет выполняться поиск в каталоге <prefix>/include. Если переменная CMAKE_LIBRARY_ARCHITECTURE установлена, то сначала будет выполняться поиск в специфичном для архитектуры каталоге <prefix>/include/${CMAKE_LIBRARY_ARCHITECTURE}, чтобы обеспечить приоритет расположения, специфичного для архитектуры, над общим расположением. Переменная CMAKE_LIBRARY_ARCHITECTURE обычно устанавливается автоматически программой CMake, и проектам не следует пытаться установить ее самостоятельно.

Для случаев, когда необходимо найти более конкретный путь include или framework и он не является частью стандартной схемы каталога или пакета, можно использовать переменные CMAKE_INCLUDE_PATH и CMAKE_FRAMEWORK_PATH. Каждая из них предоставляет список каталогов для поиска, но в отличие от CMAKE_PREFIX_PATH, подкаталог include не добавляется. CMAKE_INCLUDE_PATH поддерживается командами find_file() и find_path(), тогда как CMAKE_FRAMEWORK_PATH поддерживается этими двумя командами и командой find_library(). В остальном, эти два набора путей обрабатываются одинаково. Дополнительные подробности см. в Разд. 23.1.1, "Поведение, специфичное для Apple" ниже.

Переменные среды (специфические для CMake)

Расположение специфических переменных окружения CMake очень похоже на расположение переменных кэша. Переменные окружения CMAKE_PREFIX_PATH, CMAKE_INCLUDE_PATH и CMAKE_FRAMEWORK_PATH обрабатываются так же, как и одноименные переменные кэша, за исключением того, что на платформах Unix каждый элемент списка будет отделен двоеточием (:) вместо точки с запятой (;). Это сделано для того, чтобы переменные окружения могли использовать специфические для платформы списки путей, определенные в том же стиле, что и другие списки путей для каждой платформы.

Переменные среды (специфические для системы)

Системными переменными окружения являются INCLUDE и PATH. Обе могут содержать список, разделенный специфическим для платформы разделителем путей (двоеточие в системах Unix, точка с запятой в Windows), причем каждый элемент добавляется к набору мест поиска (INCLUDE добавляется перед PATH).

Только в Windows (включая Cygwin) записи PATH будут обрабатываться более сложным образом. Для каждого элемента в PATH вычисляется базовый путь путем удаления из конца любого идущего следом подкаталога bin или sbin. Этот базовый путь затем используется для добавления одного или двух путей к местам поиска. Если определена CMAKE_LIBRARY_ARCHITECTURE, <base>/include/${CMAKE_LIBRARY_ARCHITECTURE} добавляется. После этого путь <base>/include добавляется в набор путей поиска независимо от того, определена ли CMAKE_LIBRARY_ARCHITECTURE. При упорядочивании путей поиска эти пути располагаются непосредственно перед самим немодифицированным элементом PATH. Например, если переменная окружения PATH установлена в C:\foo\bin;D:\bar, а CMAKE_LIBRARY_ARCHITECTURE установлена в somearch, то следующий набор путей поиска будет добавлен в указанном порядке:

- C:\foo\include\somearch
- C:\foo\include
- C:\foo\bin
- D:\bar\include\somearch
- D:\bar\include
- D:\bar

Переменные кэша (зависит от платформы)

Расположение переменных кэша для конкретной платформы очень похоже на то, что используется для переменных CMakespecific. Имена немного меняются, но схема та же. Имена переменных - CMAKE_SYSTEM_PREFIX_PATH, CMAKE_SYSTEM_INCLUDE_PATH и CMAKE_SYSTEM_FRAMEWORK_PATH. Эти специфические для платформы переменные не предназначены для установки проектом или разработчиком. Скорее, они автоматически устанавливаются CMake в процессе настройки инструментария платформы, чтобы они отражали места, специфичные для платформы и используемых компиляторов. Исключение составляют случаи, когда разработчик предоставляет свой собственный файл инструментальной цепочки, в этом случае может быть целесообразно установить эти переменные в файле инструментальной цепочки.

подсказки **и** рекомендации

Каждая из различных групп переменных, рассмотренных выше, предназначена для установки чем-то вне проекта, но опции HINTS и PATHS - это то, куда сам проект должен вводить дополнительные пути поиска. Основное различие между HINTS и PATHS заключается в том, что PATHS - это, как правило, фиксированные места, которые никогда не меняются и не зависят ни от чего другого, в то время как HINTS обычно вычисляются из других значений, таких как местоположение чего-то уже найденного ранее или путь, зависящий от значения переменной или свойства. PATHS - это последние каталоги, в которых производится поиск, но HINTS ищутся до любых мест, специфичных для платформы или системы.

И HINTS, и PATHS поддерживают задание переменных окружения, которые могут содержать список путей в собственном формате хоста (т.е. разделенных двоеточием для Unix-систем, разделенных полутоном для Windows). Это делается путем предварения имени переменной среды символом ENV, например PATHS ENV FooDirs.

Все места поиска, кроме HINTS и PATHS, имеют соответствующую опцию пропуска в форме NO_..._PATH, которую можно использовать для пропуска только этого набора мест. Кроме того, опция NO_DEFAULT_PATH может быть использована для обхода всех мест поиска, кроме HINTS и PATHS, заставляя команду искать только определенные места, контролируемые проектом.

Опция PATH_SUFFIXES может быть использована для предоставления списка дополнительных подкаталогов для проверки под каждым местом поиска. Каждое место поиска используется с каждым суффиксом по очереди, затем без суффиксов вообще, прежде чем переходить к следующему месту поиска. Используйте эту опцию с осторожностью, так как она значительно расширяет общее количество мест для поиска.

Во многих случаях в проектах требуется указать только одно имя файла для поиска, и сложность порядка поиска не представляет особого интереса. Возможно, необходимо указать лишь несколько дополнительных путей для поиска (эквивалентно опции PATHS). В таких случаях можно использовать более короткую форму команды:

```
find_file(outVar name [path1 [path2...]])
```

Независимо от того, используется ли краткая или длинная форма, порядок расположения мест поиска разработан таким образом, чтобы поиск в более конкретных местах опережал поиск в более общих местах. Хотя обычно это желаемое поведение, могут быть ситуации, когда это не так. Например, проект может захотеть всегда искать сначала конкретные пути, а затем места поиска, заданные через кэш или переменные окружения. Проекты могут установить другой приоритет, вызывая find_file() несколько раз с различными опциями, управляющими местами поиска. Как только файл найден, его местоположение кэшируется, и все последующие вызовы пропускают поиск. Именно здесь наиболее полезны различные опции NO_..._PATH. Например, следующая опция принуждает сначала искать в каталоге /opt/foo/include, и только если файл не найден, будет произведен поиск в полном наборе стандартных каталогов:

```
find_file(FOO_HEADER foo.h PATHS /opt/foo/include NO_DEFAULT_PATH)
find_file(FOO_HEADER foo.h)
```

Важным условием работы является то, что для каждого вызова должна использоваться одна и та же переменная результата. Именно эта переменная кэша устанавливается и управляет пропуском последующих вызовов после того, как файл будет найден. Опция DOC может быть использована для указания документации для кэш-переменной, хранящей результат, но в проектах ее часто опускают. Выбор имени кэш-переменной, которая является самодокументирующейся, сделает ненужным явное документирование. По соглашению, эти кэш-переменные обычно пишутся в верхнем регистре и используют подчеркивание для разделения слов.

23.1.1. Поведение, характерное для Apple

Хотя команда find_file() может быть использована для поиска любого файла, она берет свое начало в поиске заголовочных файлов. Именно поэтому некоторые пути поиска по умолчанию содержат подкаталог include. На платформах Apple фреймворки иногда содержат свои собственные заголовочные файлы (см. раздел 22.3, "Фреймворки"), и команда find_file() имеет дополнительное поведение, связанное с поиском в соответствующих подкаталогах внутри них. Для каждого места поиска команда может рассматривать это место как фреймворк, как обычный каталог или как то и другое. Поведение управляется переменной CMAKE_FIND_FRAMEWORK, которая должна иметь одно из следующих значений:

- FIRST
- LAST
- ONLY
- NEVER

FIRST означает, что место поиска рассматривается как верхний каталог фреймворка и добавляются соответствующие подкаталоги для спуска в место Headers внутри него. Если именованный файл не может быть найден там, то место поиска рассматривается как обычный каталог, а не фреймворк, и поиск выполняется снова. LAST меняет этот порядок, ONLY не будет рассматривать местоположение как обычный каталог, а NEVER пропустит шаг, рассматривающий местоположение как каркас. По умолчанию в системах Apple используется FIRST, что обычно является желаемым поведением.

23.1.2. Контроль перекрестной компиляции

В сценариях кросс-компиляции набор мест поиска становится значительно сложнее. Инструментальные цепочки кросс-компиляции часто собираются в собственной структуре каталогов, чтобы отделить их от стандартной инструментальной цепочки хоста, поэтому при поиске определенного файла, как правило, желательно сначала искать в структуре каталогов инструментальной цепочки, а не хоста, чтобы была найдена версия файла, специфичная для целевой платформы. Это особенно важно при поиске программ и библиотек, но даже при поиске файлов может случиться так, что содержимое файлов может меняться между платформами (например, заголовок конфигурации, специфичный для платформы).

Для поддержки сценариев кросс-компиляции весь набор мест поиска может быть перерожден в другую часть файловой системы. Переменная CMAKE_FIND_ROOT_PATH может быть установлена в список дополнительных каталогов, в которые следует перекоренить набор мест поиска (т.е. добавлять каждый элемент списка к каждому месту поиска). Переменная CMAKE_SYSROOT также может влиять на корень поиска подобным образом. Эта переменная предназначена для указания единственного каталога, выступающего в качестве корня системы для сценария кросс-компиляции, и ее следует задавать только в файле инструментальной цепочки, никогда в самом проекте. Она также влияет на флаги, используемые во время компиляции. Начиная с CMake 3.9, более специализированные переменные CMAKE_SYSROOT_COMPILE и CMAKE_SYSROOT_LINK также имеют аналогичный эффект. Если любое из не укорененных мест уже находится под одним из мест, указанных CMAKE_FIND_ROOT_PATH, CMAKE_SYSROOT, CMAKE_SYSROOT_COMPILE или CMAKE_SYSROOT_LINK, оно не будет повторно укоренено. Путь без корней, который находится под путем, указанным переменной CMAKE_STAGING_PREFIX, также не будет перерожден. Более того, недокументированное поведение всех команд find_...() заключается в том, чтобы не перерождать любой неперерожденный путь, начинающийся с символа ~ (это сделано для того, чтобы избежать перерождения каталогов, которые находятся под домашним каталогом пользователя).

Порядок поиска по умолчанию среди укорененных и неукорененных местоположений контролируется параметром
CMAKE_FIND_ROOT_PATH_MODE_INCLUDE. Поведение, задаваемое этой переменной, может быть также отменено на основе каждого вызова путем предоставления одной из опций CMAKE_FIND_ROOT_PATH_BOTH, ONLY_CMAKE_FIND_ROOT_PATH или NO_CMAKE_FIND_ROOT_PATH команде find_file(). В следующей таблице приведены эффекты этой переменной режима, связанные с ней опции и окончательный порядок поиска:

Find Mode	find_file() Option	Search order
BOTH	CMAKE_FIND_ROOT_PATH_BOTH	• CMAKE_FIND_ROOT_PATH • CMAKE_SYSROOT_COMPILE • CMAKE_SYSROOT_LINK • CMAKE_SYSROOT • All non-rooted locations
NEVER	NO_CMAKE_FIND_ROOT_PATH	• All non-rooted locations
ONLY	ONLY_CMAKE_FIND_ROOT_PATH	• CMAKE_FIND_ROOT_PATH • CMAKE_SYSROOT_COMPILE • CMAKE_SYSROOT_LINK • CMAKE_SYSROOT • Any non-rooted locations already under one of the re-rooted locations or under CMAKE_STAGING_PREFIX

Может оказаться желательным заставить find_file() игнорировать определенные пути, которые могут содержать подходящий файл, заведомо неподходящий. В сценарии кросс-компиляции игнорирование некоторых специфических путей к хосту может быть необходимо для того, чтобы обеспечить поиск файлов, специфичных для цели, а не для хоста. Проекты могут установить переменную CMAKE_IGNORE_PATH в список каталогов, которые нужно исключить из поиска. Эти пути не являются рекурсивными, поэтому их нельзя использовать для исключения целого раздела структуры каталогов, необходимо явно указывать каждый каталог. Переменная CMAKE_SYSTEM_IGNORE_PATH делает то же самое, но она предназначена для заполнения при настройке цепочки инструментов. Обе переменные ...IGNORE_PATH применяются независимо от кросс-компиляции или нет, но было бы необычно, если бы они были установлены при отсутствии кросс-компиляции.

Разработчики также должны знать, что find_file() может предоставить только одно местоположение, но некоторые ситуации кросс-компиляции поддерживают схемы сборки, которые могут переключаться между сборками устройства и симулятора без повторного запуска CMake. Это означает, что если результаты find_file() зависят от того, какая из двух сборок используется, то они ненадежны. Этот аспект еще более важен для поиска библиотек и более подробно рассматривается в разделе 23.4 "Поиск библиотек" ниже.

23.2. Поиск путей

В проекте может потребоваться найти каталог, содержащий определенный файл, а не сам файл. Команда find_path() обеспечивает эту функциональность и идентична find_file() во всех отношениях, за исключением того, что каталог найденного файла хранится в переменной result.

23.3. Поиск программ

Поиск программ лишь немного отличается от поиска файлов: команда find_program() принимает точно такой же набор аргументов, как и find_file(), плюс еще один необязательный аргумент, NAMES_PER_DIR. Поддерживается также краткая форма команды. Ниже описаны различия между find_program() и find_file(), и

хотя это может показаться сложным, по большей части это просто описание различий, которые можно логически ожидать, но с несколькими выделенными исключениями:

Переменные кэша (специфические для CMake)

- При поиске в CMAKE_PREFIX_PATH, find_file() добавляет include к каждому элементу. find_program() вместо этого добавляет bin и sbin в качестве проверяемых мест поиска. Переменная CMAKE_LIBRARY_ARCHITECTURE не влияет на find_program().
- CMAKE_PROGRAM_PATH заменяет CMAKE_INCLUDE_PATH, но в остальном используется точно так же. CMAKE_PROGRAM_PATH используется только функцией find_program().
- CMAKE_APPBUNDLE_PATH заменяет CMAKE_FRAMEWORK_PATH, но в остальном используется точно так же. Он используется только функциями find_program() и find_package().

Переменные среды (специфические для системы)

- Поиск мест для стандартных переменных системного окружения обрабатывается значительно проще. INCLUDE не имеет значения для find_program(), и каждый элемент в PATH проверяется без каких-либо изменений. Поведение одинаково на всех платформах.

Общий

- Обычно все места поиска проверяются для данного имени перед тем, как перейти к поиску следующего имени в списке, если опция NAMES используется для указания нескольких имен. Команда find_program() поддерживает опцию NAMES_PER_DIR, которая изменяет этот порядок, проверяя каждое имя для определенного места поиска перед переходом к следующему месту. Опция NAMES_PER_DIR была добавлена в CMake 3.4.
- В Windows (включая Cygwin и MinGW) расширения файлов .com и .exe также проверяются автоматически, поэтому нет необходимости указывать такие расширения как часть имени программы для поиска. Эти расширения проверяются в первую очередь, а затем имена без расширений. Обратите внимание, что файлы .bat и .cmd не будут искаться автоматически.
- В то время как find_file() использует CMAKE_FIND_FRAMEWORK для определения порядка поиска между путями фреймворка и не фреймворка, find_program() использует CMAKE_FIND_APPBUNDLE, которая обеспечивает аналогичный контроль между путями пучка приложений и не пучка. Поддерживаемые значения одинаковы для обеих переменных, и они имеют ожидаемое эквивалентное значение для пучков. В то время как поиск файлов будет искать в подкаталоге Headers, поиск программ будет искать в подкаталоге Contents/MacOS и установит результат на исполняемый файл в пакете приложений.
- CMAKE_FIND_ROOT_PATH_MODE_INCLUDE не влияет на find_program(), она заменяется переменной CMAKE_FIND_ROOT_PATH_MODE_PROGRAM, которая имеет эквивалентный эффект, но применяется только к find_program(). При кросс-компиляции обычно ищется инструмент хост-платформы, а не программа на целевой платформе, поэтому CMAKE_FIND_ROOT_PATH_MODE_PROGRAM часто устанавливается в NEVER.

23.4. Поиск библиотек

Поиск библиотек также похож на поиск файлов: команда find_library() поддерживает тот же набор опций, что и find_file(), плюс дополнительный параметр NAMES_PER_DIR. Существуют следующие различия:

Переменные кэша (специфические для CMake)

- При поиске в CMAKE_PREFIX_PATH find_file() добавляет include к каждому элементу, тогда как find_library() вместо этого добавляет lib. Переменная CMAKE_LIBRARY_ARCHITECTURE также учитывается так же, как и для find_file().
- CMAKE_LIBRARY_PATH заменяет CMAKE_INCLUDE_PATH, но в остальном используется точно так же. CMAKE_LIBRARY_PATH используется только функцией find_library(). Переменная CMAKE_FRAMEWORK_PATH используется точно так же, как и для find_file().

Переменные среды (специфические для системы)

- Поиск мест для стандартных системных переменных окружения выполняется аналогично find_file(). Вместо INCLUDE используется переменная окружения LIB. Более того, места поиска по PATH следуют той же сложной логике, что и для find_file(), за исключением того, что к каждому префиксу добавляется lib, а не include. Как и для find_file(), сложная логика PATH применима только в Windows.

Общий

- Опция NAMES_PER_DIR имеет точно такое же значение, как и для find_program(), и также была добавлена только в CMake 3.4.
- И find_file(), и find_library() используют CMAKE_FIND_FRAMEWORK для определения порядка поиска между путями к фреймворку и без него. В случае find_library(), если найден фреймворк, то имя каталога верхнего уровня .framework будет сохранено в переменной результата.
- CMAKE_FIND_ROOT_PATH_MODE_INCLUDE не влияет на find_library(), она заменяется переменной CMAKE_FIND_ROOT_PATH_MODE_LIBRARY, которая имеет эквивалентный эффект, но применяется исключительно к find_library(). На платформах Apple следует внимательно подумать, прежде чем устанавливать CMAKE_FIND_ROOT_PATH_MODE_LIBRARY в ONLY, поскольку библиотеки могут быть собраны в виде толстых двоичных файлов, поддерживающих несколько целевых платформ. Эти жирные двоичные файлы могут не находиться в путях, специфичных для целевой платформы, поэтому для их поиска может потребоваться поиск в путях хост-платформы.

При использовании функции find_library() возникают дополнительные различия в поведении. Платформы имеют различные соглашения для имен библиотек, например, добавление lib на большинстве платформ Unix. Расширения файлов также значительно отличаются на разных платформах, и DLL на Windows могут иметь связанную с ними библиотеку импорта с другим расширением файла. Команда find_library() делает все возможное, чтобы абстрагироваться от большинства этих различий, позволяя проектам указывать только базовое имя библиотеки в качестве имени для поиска. Если каталог содержит как статические, так и разделяемые библиотеки, будет найдена разделяемая библиотека. В большинстве случаев эта абстракция работает хорошо, но в некоторых обстоятельствах может быть полезно переопределить это поведение. Одним из распространенных случаев является предоставление приоритета статическим библиотекам перед разделяемыми библиотеками, причем потенциально только на некоторых платформах, но не на других. В следующем наивном примере статическая библиотека foobar будет предпочтительнее общей на Linux, но не на macOS или Windows:

```
# WARNING: Not robust!
find_library(FOOBAR_LIBRARY NAMES libfoobar.a foobar)
```

Помните, что переопределение приоритета применяется только к библиотекам, найденным в определенном каталоге. Если набор мест поиска таков, что каталог, содержащий только общую библиотеку, будет найден раньше, чем каталог, содержащий статическую библиотеку, то описанная выше техника не приведет к тому, что статическая библиотека будет найдена. Более надежный способ обеспечить приоритет статической библиотеки над общими библиотеками во всех местах поиска - использовать несколько вызовов find_library(), как показано ниже:

```
# Better, static library now has priority across
# all search locations
find_library(FOOBAR_LIBRARY libfoobar.a)
find_library(FOOBAR_LIBRARY foobar)
```

Обратите внимание, что такие методы нельзя использовать в Windows, поскольку статические библиотеки и библиотека импорта для разделяемых библиотек (т.е. DLL) имеют одинаковое имя файла, включая суффикс (например, foobar.lib). Поэтому имя файла не может быть использовано для различения двух типов библиотек.

Другая сложность, уникальная для работы с библиотеками, заключается в том, что многие платформы поддерживают 32- и 64-битные архитектуры, и могут существовать 32- и 64-битные версии библиотек, установленные в разных местах, но с одинаковыми именами файлов. Структура каталогов, используемая для разделения различных архитектур на таких многобитных системах, может различаться даже между дистрибутивами для одной и той же платформы. Например, некоторые дистрибутивы размещают 64-битные библиотеки в каталогах lib и 32-битные библиотеки в lib32, в то время как другие размещают 64-битные библиотеки в lib64, а 32-битные библиотеки в lib. На других платформах используется еще одна вариация - подкаталог libx32. CMake, как правило, знает об этих вариациях и при настройке платформы по умолчанию заполняет глобальные свойства FIND_LIBRARY_USE_LIB32_PATHS, FIND_LIBRARY_USE_LIB64_PATHS и FIND_LIBRARY_USE_LIBX32_PATHS соответствующими значениями, чтобы контролировать, какие каталоги, специфичные для архитектуры, следует искать первыми, если таковые имеются. Проекты могут переопределить эти значения с помощью собственного префикса, используя параметр
CMAKE_FIND_LIBRARY_CUSTOM_LIB_SUFFIX переменная, но необходимость в ней должна возникать очень редко.

Когда активен суффикс, специфичный для архитектуры (будь то из одного из вышеуказанных глобальных свойств или из переменной CMAKE_FIND_LIBRARY_CUSTOM_LIB_SUFFIX), логика, используемая для дополнения мест поиска местами, специфичными для архитектуры, нетривиальна. Любой каталог в любом месте пути поиска, который заканчивается на lib, дополняется эквивалентом, соответствующим архитектуре. Это происходит рекурсивно по всему пути, поэтому место поиска /opt/mylib/foo/lib может привести к расширенному набору мест поиска /opt/mylib64/foo/lib64, /opt/mylib64/foo/lib,
/opt/mylib/foo/lib64 и /opt/mylib/foo/lib на некоторых 64-битных системах. Даже если место поиска не заканчивается на lib, оно все равно будет дополнено местом с учетом архитектуры, поэтому поиск по месту /opt/foo может привести к поиску /opt/foo64 и /opt/foo на некоторых 64-битных системах.

Детали дополнения пути поиска, специфичного для архитектуры, обычно не являются тем, что должно волновать разработчиков. В тех ситуациях, когда обнаруживаются нежелательные библиотеки или пропускаются желаемые библиотеки, может оказаться более простым решением принудительно изменить результат с помощью переменных типа CMAKE_LIBRARY_PATH, чем пытаться манипулировать логикой, специфичной для архитектуры. Детальное знание всех тонкостей обычно не требуется, простого понимания вышеупомянутых моментов должно быть достаточно, хотя бы для того, чтобы уменьшить некоторую загадочность того, как CMake находит библиотеки в специфических для архитектуры местах.

При работе с генератором CMake, поддерживающим переключение между конфигурациями устройства и симулятора во время сборки, необходимо соблюдать особую осторожность. Любые результаты find_library() будут непригодны для использования в таких случаях, поскольку они могут найти библиотеку только для устройства или симулятора, но не для обоих. Даже если запустить CMake повторно, он сохранит свои кэшированные результаты и не обновит местоположение библиотеки, пока соответствующая запись в кэше не будет удалена вручную. Это особенно распространенная проблема в сборках Xcode, где проекты могут захотеть использовать find_library() для поиска различных фреймворков или общих библиотек, таких как zlib. В таких ситуациях у проектов нет выбора, кроме как указать флаги компоновщика напрямую без путей, оставляя компоновщику возможность найти библиотеку по своему пути поиска. Для фреймворков Apple это означает указание двух значений, поскольку фреймворки добавляются с помощью -framework <FrameworkName>. Для обычных библиотек, таких как zlib, достаточно более традиционного -lz.

23.5. Поиск пакетов

Различные команды find_...(), рассмотренные в предыдущих разделах, направлены на поиск одного конкретного элемента. Однако довольно часто эти элементы являются лишь частью большого пакета, а пакет в целом может иметь собственные характеристики, которые могут интересовать проекты, например, номер версии или поддержка определенных функций. Как правило, проекты хотят найти пакет как единое целое, а не собирать его различные части вручную.

Существует два основных способа определения пакетов в CMake: либо в виде модуля, либо через сведения о конфигурации. Детали конфигурации обычно предоставляются как часть самого пакета, и они более тесно связаны с функциональностью различных команд find_...(), рассмотренных в предыдущих разделах. Модули, с другой стороны, обычно определяются чем-то, не связанным с пакетом (обычно CMake или самими проектами), и, как следствие, их сложнее поддерживать в актуальном состоянии по мере развития пакета.

Когда загружается модуль или файл конфигурации, он обычно определяет переменные и импортируемые цели для пакета. Они могут указывать расположение программ, библиотек, флагов, которые будут использоваться потребляющими целями и так далее. Пакеты также могут определять функции и макросы, хотя обычно это делают только модули. Не существует набора требований к тому, что должно быть предоставлено, но есть некоторые соглашения, которые изложены в руководстве разработчика CMake. Авторы проектов должны обращаться к документации каждого модуля или пакета config, чтобы понять, что он предоставляет. В качестве общего руководства, старые модули обычно предоставляют переменные, которые следуют довольно последовательному шаблону, в то время как новые модули и реализации конфигураций обычно определяют импортируемые цели. В тех случаях, когда предоставляются и переменные, и импортируемые цели, проекты должны предпочесть последние из-за их большей надежности и лучшей интеграции с функциями переходных зависимостей CMake.

Проекты обычно ищут пакет с помощью команды find_package(), которая имеет короткую и длинную форму. Короткая форма обычно предпочтительнее из-за ее большей простоты и потому, что она поддерживает пакеты модулей и конфигураций, в то время как длинная форма не поддерживает модули. Однако длинная форма обеспечивает больший контроль над поиском, что делает ее более предпочтительной в некоторых ситуациях. Короткая форма имеет всего несколько вариантов:

```
find_package(packageName
    [version [EXACT]]
    [QUIET] [REQUIRED]
    [[COMPONENTS] component1 [component2...]]
    [OPTIONAL_COMPONENTS component3 [component4...]]
    [MODULE]
    [NO_POLICY_SCOPE]
)
```

Необязательный аргумент version указывает, что пакет должен быть указанной версии или выше, но если также указан параметр EXACT, то версия пакета должна точно совпадать. Пакет может быть необязательным, то есть проект может использовать его, если он доступен, или работать без него, если пакет не может быть найден или имеет неподходящую версию. Если пакет является обязательным, следует указать опцию REQUIRED, чтобы команда завершилась с ошибкой, если пакет не может быть найден или если требования к версии не могут быть выполнены. Обычно find_package() выводит сообщения, если не может найти пакет, но можно указать опцию QUIET, чтобы подавить их, что особенно полезно для необязательных пакетов, где отсутствие пакета не должно приводить к предупреждениям, которые могут запутать разработчика. QUIET также предотвращает сообщения о состоянии, которые обычно печатаются, когда пакет найден в первый раз.

Опции, связанные с компонентами, позволяют проекту указать, какие части пакета его интересуют. Не все пакеты поддерживают компоненты, от реализации модуля или конфигурации зависит, определены ли компоненты и что они собой представляют. Примером, когда компоненты могут быть полезны, является большой пакет, такой как Qt, где не все компоненты могут быть установлены. Проекту может быть недостаточно просто сказать, что ему нужен Qt, ему также может потребоваться указать, какие именно части Qt. Команда find_package() позволяет проекту указать компоненты как обязательные с помощью аргументов COMPONENTS или как необязательные с помощью аргументов OPTIONAL_COMPONENTS. Например, следующий вызов требует, чтобы был найден Qt 5.9 или более поздней версии, а также должен быть доступен компонент Gui. Модуль DBus, однако, является необязательным.

```
find_package(Qt5 5.9 REQUIRED
    COMPONENTS Gui
    OPTIONAL_COMPONENTS DBus
)
```

Когда присутствует опция REQUIRED, ключевое слово COMPONENTS может быть опущено, а обязательные компоненты размещены после REQUIRED. Это особенно часто используется, когда нет необязательных компонентов. Например:

```
find_package(Qt5 5.9 REQUIRED Gui Widgets Network)
```

Если в пакете определены компоненты, но в find_package() не указаны компоненты, то как это будет обработано, зависит от определения модуля или конфигурации. Для некоторых пакетов это может трактоваться так, как будто все компоненты были указаны, для других - как отсутствие необходимых компонентов (при этом основные детали пакета могут быть определены, например, базовые библиотеки, версия пакета и т.д.). Другая возможность заключается в том, что отсутствие компонентов может быть расценено как ошибка. Учитывая вариативность поведения, разработчикам следует обратиться к документации по пакету, который они хотят найти.

Остальные опции краткой формы используются реже. Ключевое слово NO_POLICY_SCOPE - это исторический пережиток эпохи CMake 2.6, и проектам следует избегать его использования. Ключевое слово MODULE ограничивает вызов поиском только модулей, а не пакетов конфигурации. Проектам вообще следует избегать использования этой опции, поскольку они не должны заботиться о деталях реализации того, как определяется пакет, а только о том, чтобы сформулировать требования к пакету. Когда MODULE отсутствует, краткая форма команды find_package() сначала будет искать подходящий модуль, а затем, если такой модуль не найден, будет искать конфигурационный пакет.

Впервые модули были рассмотрены в главе 11 "*Модули*". Если непакетные модули включаются в проект с помощью команды include(), то пакетные модули имеют имя файла вида Find<packageName>.cmake и предназначены для обработки вызовом find_package(). По этой причине их принято называть *модулями Find*. И include(), и find_package() учитывают переменную CMAKE_MODULE_PATH как список каталогов, в которых CMake должен искать модули, входящие в каждый выпуск CMake.

Модули find отвечают за реализацию всех аспектов вызова find_package(), включая нахождение пакета, проверку версии, выполнение требований компонента и регистрацию или отсутствие регистрации сообщений в зависимости от ситуации. Не все модули find выполняют эти обязанности, и они могут игнорировать часть или всю информацию, предоставляемую помимо имени пакета, поэтому, как всегда, обратитесь к документации модуля для подтверждения ожидаемого поведения.

Модули поиска обычно реализуются в виде вызовов различных команд find_...(). В результате на них иногда могут влиять кэш и переменные окружения, относящиеся к этим командам. Переменная CMAKE_PREFIX_PATH особенно удобна для влияния на модули поиска, поскольку каждый указанный путь действует как базовая точка, ниже которой каждая команда find_...() добавляет свои собственные подкаталоги, специфичные для команды. Для пакетов, имеющих достаточно стандартную структуру, добавления в CMAKE_PREFIX_PATH только базового места установки пакета часто бывает достаточно, чтобы модуль find нашел все необходимые ему компоненты пакета.

По сравнению с модулями find, пакеты с деталями config предлагают более богатый, более надежный способ получения информации о пакете для проектов. В режиме config доступен гораздо более широкий набор опций find_package(), а полная длинная форма команды имеет много общего с другими командами find_...():

```
find_package(packageName
    [version [EXACT]]
    [QUIET | REQUIRED]
    [[COMPONENTS] component1 [component2...]]
    [NO_MODULE | CONFIG]
    [NO_POLICY_SCOPE]
    [NAMES name1 [name2 ...]]
    [CONFIGS fileName1 [fileName2...]]
    [HINTS path1 [path2 ... ]]
    [PATHS path1 [path2 ... ]]
    [PATH_SUFFIXES suffix1 [suffix2 ...]]
    [CMAKE_FIND_ROOT_PATH_BOTH |
     ONLY_CMAKE_FIND_ROOT_PATH |
     NO_CMAKE_FIND_ROOT_PATH]
    [<skip-options>]    # See further below
)
```

Если функция find_package() вызывается с опцией, поддерживаемой только длинной формой, поиск модуля Find пропускается. Ключевые слова NO_MODULE или CONFIG позволяют рассматривать вызов, который в

противном случае соответствовал бы короткой форме, как длинную форму и, следовательно, искать только детали конфигурации (оба ключевых слова эквивалентны).

При поиске деталей конфигурации find_package() по умолчанию ищет файл с именем <packageName>Config.cmake или менее распространенный <lowercasePackageName>-config.cmake. Опция CONFIGS может быть использована для указания другого набора имен файлов для поиска, но использование этой опции должно быть редким. Нестандартные имена файлов потребуют, чтобы каждый проект, желающий найти этот пакет, знал о нестандартном имени файла.

Когда конфигурационный файл найден, find_package() также ищет связанный с ним файл версии в том же каталоге. К базовому имени файла версии добавляется Version или -version, поэтому FooConfig.cmake будет искать файл версии с именем FooConfigVersion.cmake или FooConfig-version.cmake, а foo-config.cmake будет искать foo-configVersion.cmake или foo-configversion.cmake. Пакеты не обязаны предоставлять файл версии, но обычно они это делают. Если сведения о версии включены в вызов find_package(), но файл версии для этого пакета отсутствует, считается, что требования к версии не выполнены.

Поиск местоположений осуществляется по той же схеме, что и в других командах find_...(), за исключением того, что поддерживаются также реестры пакетов. Каждое место поиска рассматривается как возможная базовая точка установки пакета, ниже которой может производиться поиск в различных подкаталогах:

```
<prefix>/
<prefix>/(cmake|CMake)/
<prefix>/<packageName>*/
<prefix>/<packageName>*/(cmake|CMake)/
<prefix>/(lib/<arch>|lib*|share)/cmake/<packageName>*/
<prefix>/(lib/<arch>|lib*|share)/<packageName>*/
<prefix>/(lib/<arch>|lib*|share)/<packageName>*/(cmake|CMake)/
<prefix>/<packageName>*/(lib/<arch>|lib*|share)/cmake/<packageName>*/
<prefix>/<packageName>*/(lib/<arch>|lib*|share)/<packageName>*/
<prefix>/<packageName>*/(lib/<arch>|lib*|share)/<packageName>*/(cmake|CMake)/

# The following are also checked on Apple platforms

<prefix>/<packageName>.framework/Resources/
<prefix>/<packageName>.framework/Resources/CMake/
<prefix>/<packageName>.framework/Versions/*/Resources/
<prefix>/<packageName>.framework/Versions/*/Resources/CMake/
<prefix>/<packageName>.app/Contents/Resources/
<prefix>/<packageName>.app/Contents/Resources/CMake/
```

В приведенном выше примере <packageName> обрабатывается нечувствительно к регистру, а подкаталоги lib/<arch> ищутся только в том случае, если установлен CMAKE_LIBRARY_ARCHITECTURE. Подкаталоги lib* представляют собой набор каталогов, которые могут включать lib64, lib32, libx32 и lib, последний из которых всегда проверяется. Если для функции find_package() задана опция NAMES, то для каждого указанного имени проверяются все вышеперечисленные каталоги.

Набор проверяемых базовых точек места поиска следует в порядке, определенном в следующей таблице, которая имеет много общего с другими командами find_...(). Большинство мест поиска можно отключить, добавив соответствующее ключевое слово NO_...:

Location	Skip Option
Package root variables	NO_PACKAGE_ROOT_PATH
Cache variables (CMake-specific)	NO_CMAKE_PATH
Environment variables (CMake-specific)	NO_CMAKE_ENVIRONMENT_PATH
Paths specified via the HINTS option	
Environment variables (system-specific)	NO_SYSTEM_ENVIRONMENT_PATH
User package registry	NO_CMAKE_PACKAGE_REGISTRY
Cache variables (platform-specific)	NO_CMAKE_SYSTEM_PATH
System package registry	NO_CMAKE_SYSTEM_PACKAGE_REGISTRY
Paths specified via the PATHS option	

Корневые переменные пакета

Как и для других команд find_...(), поддержка корневых переменных пакета была добавлена в качестве места поиска в CMake 3.9.0, удалена в 3.9.1 из-за проблем обратной совместимости и снова добавлена в CMake 3.12. При каждом вызове find_package() переменные <packageName>_ROOT CMake и переменные окружения помещаются во внутренний стек путей. Эти пути используются точно так же, как CMAKE_PREFIX_PATH, не только для текущего вызова find_package(), но и для всех команд find_...(), которые могут быть вызваны как часть обработки find_package(). На практике это означает, что если вызов find_package() загружает модуль Find, то все команды find_...(), которые вызывает модуль Find, будут использовать каждый путь в стеке, как если бы это был CMAKE_PREFIX_PATH, прежде чем проверять другие пути.

Например, вызов find_package(Foo) привел к загрузке FindFoo.cmake. Любая команда find_...() в FindFoo.cmake будет сначала искать ${Foo_ROOT} и $ENV{Foo_ROOT} (если они заданы), прежде чем переходить к проверке других мест поиска. Если FindFoo.cmake содержал вызов типа find_package(Bar), в результате которого загружался FindBar.cmake, то стек содержал бы
${Bar_ROOT}, $ENV{Bar_ROOT}, ${Foo_ROOT} и $ENV{Foo_ROOT}. Эта возможность означает, что вложенные модули Find будут сначала искать префиксные местоположения каждого из своих родительских модулей Find, так что эту информацию не нужно вручную передавать вниз через CMAKE_PREFIX_PATH или другой подобный метод. По большей части, проекты могут игнорировать эту функциональность, так как она должна работать прозрачно без каких-либо конкретных действий со стороны проекта. В основном, ее следует рассматривать как автоматическое удобство.

Переменные кэша (специфические для CMake)

Расположение специфических для CMake переменных кэша происходит от переменных кэша CMAKE_PREFIX_PATH, CMAKE_FRAMEWORK_PATH и CMAKE_APPBUNDLE_PATH. Они работают практически так же, как и другие команды find_...(), за исключением того, что записи CMAKE_PREFIX_PATH уже соответствуют базовым точкам установки пакетов, поэтому к ним не добавляются такие каталоги, как bin, lib, include и т.д.

Переменные среды (специфические для CMake)

Они имеют такое же отношение к переменным кэша, как и другие команды find_...(). Переменные окружения CMAKE_PREFIX_PATH, CMAKE_INCLUDE_PATH и CMAKE_FRAMEWORK_PATH используют специфический для платформы разделитель путей (двоеточия на платформах Unix, полуточия на Windows). Дополнительная переменная окружения <packageName>_DIR также проверяется перед тремя другими.

Переменные среды (специфические для системы)

Единственной поддерживаемой системной переменной окружения является PATH. Каждая запись используется как базовая точка установки пакета, за исключением того, что удаляются все последующие bin или sbin. Это точка, в которой в большинстве систем, скорее всего, будет производиться поиск системных мест по умолчанию, таких как /usr.

Переменные кэша (зависит от платформы)

Расположение переменных кэша, специфичных для конкретной платформы, происходит по той же схеме, что и в других командах find_...(), предоставляя ...SYSTEM... версии переменных кэша, специфичных для CMake. Имена переменных: CMAKE_SYSTEM_PREFIX_PATH, CMAKE_SYSTEM_FRAMEWORK_PATH и CMAKE_SYSTEM_APPBUNDLE_PATH и не предназначены для установки проектом.

подсказки **и** рекомендации

Они работают точно так же, как и другие команды find_...(), за исключением того, что они не поддерживают элементы формы ENV someVar.

Реестры пакетов

Уникальные для find_package(), пользовательский и системный реестры пакетов предназначены для обеспечения возможности легкого поиска пакетов без их установки в стандартных системных местах. Более подробное обсуждение см. ниже в разделе 23.5.1, "Реестры пакетов".

Различные опции NO_... работают так же, как и в других командах find_...(), позволяя обходить каждую группу мест поиска по отдельности. Ключевое слово NO_DEFAULT_PATH заставляет обходить все места поиска, кроме HINTS и PATHS. Опция PATH_SUFFIXES также имеет ожидаемый эффект, позволяя проверять дополнительные подкаталоги ниже каждого места поиска.

Команда find_package() также поддерживает ту же логику повторного рутирования поиска, что и другие команды find_...(). CMAKE_SYSROOT, CMAKE_STAGING_PREFIX и CMAKE_FIND_ROOT_PATH рассматриваются так же, как и другие команды, и значения опций CMAKE_FIND_ROOT_PATH_BOTH, ONLY_CMAKE_FIND_ROOT_PATH и NO_CMAKE_FIND_ROOT_PATH также эквивалентны. Режим перезагрузки по умолчанию, когда ни одна из этих трех опций не предоставлена, контролируется переменной CMAKE_FIND_ROOT_PATH_MODE_PACKAGE, которая имеет предсказуемый набор допустимых значений (ONLY, NEVER или BOTH).

В отличие от других команд find_...(), при поиске конфигурационного файла find_package() не обязательно останавливает поиск на первом найденном пакете, который соответствует заданным критериям. Части поиска рассматривают семейство мест поиска, и результаты поиска могут возвращать несколько совпадений для данной подветви поиска. Обычно это может произойти, если существует несколько версий пакета, установленных в некоторой общей директории, каждая из которых имеет версионный подкаталог ниже этой общей точки. В таких случаях могут быть использованы параметры CMAKE_FIND_PACKAGE_SORT_ORDER и Переменные CMAKE_FIND_PACKAGE_SORT_DIRECTION используются для сортировки кандидатов на основе данных об их версии. CMAKE_FIND_PACKAGE_SORT_DIRECTION должна иметь значение DEC или ASC для указания направления сортировки по убыванию (выбирается самый новый) или по возрастанию (выбирается самый старый) соответственно, в то время как

CMAKE_FIND_PACKAGE_SORT_ORDER управляет типом сортировки и имеет документированные значения NAME, NATURAL или NONE. Если установлено значение NONE или не установлено вообще, сортировка не выполняется, и будет использоваться первый найденный допустимый пакет. Параметр NAME сортирует лексикографически, а NATURAL сортирует путем сравнения последовательностей цифр как целых чисел. Следующая таблица демонстрирует разницу между двумя последними параметрами при сортировке в порядке убывания, который используется по умолчанию, если
CMAKE_FIND_PACKAGE_SORT_DIRECTION не установлен:

NAME	NATURAL
1.9	1.10
1.10	1.9
1.0	1.0

На практике тонкости логики поиска обычно намного превосходят уровень детализации, необходимый для эффективного использования команды find_package(). Пока пакет следует одной из наиболее распространенных схем расположения каталогов и находится в одном из базовых мест установки более высокого уровня, команда find_package() обычно находит его конфигурационный файл без дополнительной помощи.

Как только подходящий конфигурационный файл для пакета найден, кэш-переменная <packageName>_DIR будет установлена в каталог, содержащий этот файл. Последующие вызовы find_package() будут сначала искать в этом каталоге, и если файл конфигурации все еще существует, он будет использован без дальнейшего поиска. <packageName>_DIR игнорируется, если в этом месте больше нет файла конфигурации для пакета. Такое расположение гарантирует, что последующие вызовы find_package() для одного и того же пакета выполняются намного быстрее, даже от одного вызова CMake к другому, но поиск все равно выполняется, если пакет удален. Однако следует помнить, что кэширование местоположения пакета может также означать, что CMake может не получить возможности узнать о новом добавленном пакете в более предпочтительном месте. Например, в операционной системе может быть предустановлена довольно старая версия пакета. При первом запуске проекта CMake находит эту старую версию и сохраняет ее расположение в кэше. Пользователь видит, что используется старая версия, и решает установить более новую версию пакета в каком-то другом каталоге, добавляет это место в CMAKE_PREFIX_PATH и снова запускает CMake. В этом случае старая версия все равно будет использоваться, поскольку кэш по-прежнему указывает на расположение старого пакета. Запись кэша <packageName>_DIR должна быть удалена или старая версия деинсталлирована, прежде чем будет учитываться расположение новой версии.

Существует еще один способ повлиять на обработку определенных пакетов. Можно отключить каждый не требующий запроса вызов find_package() для данного имени пакета, установив переменную CMAKE_DISABLE_FIND_PACKAGE_<packageName> в true на ранней стадии проекта, в идеале на верхнем уровне или в качестве переменной кэша. Это можно рассматривать как способ отключения необязательного пакета, предотвращающий его поиск с помощью вызовов find_package(). Обратите внимание, что это не предотвратит такие вызовы, если они включают ключевое слово REQUIRED.

23.5.1. Реестры пакетов

Пакеты обычно находятся либо в стандартных системных каталогах, либо в каталогах, о которых CMake было сообщено через CMAKE_PREFIX_PATH или аналогичные. Для несистемных пакетов может быть утомительно или нежелательно указывать местоположение каждого пакета, если они не имеют общего префикса установки.

CMake поддерживает форму реестра пакетов, которая позволяет собирать ссылки на произвольные места в одном месте. Это позволяет пользователю вести реестр в рамках учетной записи или всей системы, к которому CMake будет обращаться автоматически без дополнительных указаний. Места, на которые ссылается реестр, не обязательно должны быть полной установкой пакета, это может быть и каталог в дереве сборки пакета (или любой другой каталог), если там есть необходимые файлы.

В Windows предусмотрено два реестра. Пользовательский реестр хранится в реестре Windows под ключом HKEY_CURRENT_USER, а реестр системных пакетов - под ключом HKEY_LOCAL_MACHINE:

```
HKEY_CURRENT_USER\Software\Kitware\CMake\Packages\<packageName>\
HKEY_LOCAL_MACHINE\Software\Kitware\CMake\Packages\<packageName>\
```

Для данного имени пакета (packageName) каждая запись под этим пунктом представляет собой произвольное имя, содержащее значение REG_SZ. Ожидается, что это значение будет каталогом, в котором можно найти файл конфигурации для данного пакета. На платформах Unix нет системного реестра пакетов, есть только реестр пользовательских пакетов, хранящийся в домашнем каталоге пользователя, и записи в этом пункте имеют то же значение, что и в Windows:

```
~/.cmake/packages/<packageName>/
```

CMake предоставляет очень мало помощи в создании этих записей на любой платформе. Для установленных пакетов не предусмотрено никакого автоматического механизма, но команда export() может быть использована в файлах CMakeLists.txt проекта для добавления частей дерева сборки проекта в пользовательский реестр:

```
export(PACKAGE packageName)
```

Это добавляет указанный пакет в реестр пользовательских пакетов и указывает его на текущий двоичный каталог, связанный с тем, где был вызван export(). Затем проект должен убедиться, что в этом каталоге существует соответствующий конфигурационный файл для пакета. Если такого конфигурационного файла не существует и для этого пакета в любом проекте выполняется вызов find_package(), запись в реестре будет автоматически удалена, если это позволяют разрешения. Обычно именем каждой записи в реестре пакетов является MD5-хэш пути к каталогу, на который она указывает. Это позволяет избежать коллизий имен и является стратегией именования, используемой командой export(PACKAGE).

Добавление местоположений из дерева сборки в реестр пакетов имеет свои опасности. Хотя export(PACKAGE) позволяет добавить расположение в реестр, нет соответствующего механизма для его удаления, кроме как вручную удалить запись в реестре или удалить файл конфигурации пакета из каталога сборки. Это легко забыть сделать, поэтому старое дерево сборки, оставшееся после прошлых экспериментов, может быть неожиданно поднято. Использование export(PACKAGE) также может внести хаос в системы непрерывной интеграции, заставляя проекты подхватывать деревья сборок других проектов, созданных на том же ведомом устройстве сборки. Один из способов предотвратить это - установить переменную CMAKE_EXPORT_NO_PACKAGE_REGISTRY в ON, что приведет к запрету всех вызовов export(PACKAGE). Это не позволит проектам добавлять свои собственные деревья сборки в реестр пользовательских пакетов. В дополнение к этому, проекты могут установить CMAKE_FIND_PACKAGE_NO_PACKAGE_REGISTRY или CMAKE_FIND_PACKAGE_NO_SYSTEM_PACKAGE_REGISTRY в ON, чтобы все их вызовы find_package() игнорировали реестры пакетов пользователя и системы соответственно.

На практике реестры пакетов используются нечасто. Ограниченная помощь, предоставляемая для добавления и удаления записей, означает, что ведение реестра - это в некоторой степени ручной процесс. Когда пакет устанавливается с помощью стандартной системы управления пакетами хоста, он может добавить себя в системный или пользовательский реестр, в зависимости от ситуации, а затем программа удаления пакета может удалить эту запись. Хотя места расположения пакетов хорошо определены и их определение концептуально просто, немногие пакеты утруждают себя работой по регистрации и снятию с регистрации. Различные способы, которыми пакет может попасть на машину конечного пользователя, несколько затрудняют надежную и простую реализацию таких функций регистрации и снятия с регистрации.

23.5.2. FindPkgConfig

Команда find_package() обычно является предпочтительным методом для поиска и включения пакета в проект CMake, но в некоторых случаях результаты могут быть менее чем идеальными. Некоторые модули Find еще не обновлены до более современной практики и не предоставляют цели импорта, полагаясь вместо этого на определение набора переменных, которые проекты-потребители должны обрабатывать вручную. Другие модули могут отставать от последних релизов пакетов, что приводит к несовместимости или предоставлению неверной информации.

В некоторых случаях пакет может иметь поддержку pkg-config, инструмента, который предоставляет информацию, аналогичную find_package(), но в другой форме. Если такие данные pkg-config доступны, то модуль PkgConfig Find может быть использован для чтения этой информации и предоставления ее в более удобном для CMake виде. Цели импорта могут быть созданы автоматически, освобождая проекты от необходимости обрабатывать различные переменные вручную. Детали pkg-config также, скорее всего, будут соответствовать установленной версии пакета, поскольку они обычно предоставляются самим пакетом.

Модуль FindPkgConfig находит исполняемый файл pkg-config и определяет несколько функций, которые вызывают его для поиска и извлечения подробностей о пакетах с поддержкой pkg-config. Если модуль находит исполняемый файл, он устанавливает переменную PKG_CONFIG_FOUND в true, а переменную PKG_CONFIG_EXECUTABLE - в местоположение инструмента. PKG_CONFIG_VERSION_STRING также устанавливается в версию инструмента (за исключением версий CMake до 2.8.8).

На практике проекты редко должны использовать переменную PKG_CONFIG_EXECUTABLE, так как модуль также определяет две функции, которые обертывают инструмент, чтобы обеспечить более удобный способ запроса деталей пакета. Эти две функции, pkg_check_modules() и pkg_search_module(), принимают точно такой же набор опций и имеют схожее поведение. Основное различие между ними заключается в том, что pkg_check_modules() проверяет все модули, указанные в списке аргументов, в то время как pkg_search_module() останавливается на первом найденном модуле, удовлетворяющем заданным критериям. Использование термина *модуль*, а не *пакет,* закрепилось в истории этих команд и может вызвать некоторую путаницу, но они не имеют прямого отношения к обычным модулям CMake и по сути могут рассматриваться как пакеты.

```
pkg_check_modules(prefix
    [REQUIRED] [QUIET]
    [IMPORTED_TARGET]
    [NO_CMAKE_PATH]
    [NO_CMAKE_ENVIRONMENT_PATH]
    moduleSpec1 [moduleSpec2...]
)

pkg_search_module(prefix
    [REQUIRED] [QUIET]
    [IMPORTED_TARGET]
    [NO_CMAKE_PATH]
    [NO_CMAKE_ENVIRONMENT_PATH]
    moduleSpec1 [moduleSpec2...]
)
```

Поведение этих функций имеет некоторое сходство с командой find_package(). Аргументы REQUIRED и QUIET имеют здесь такой же эффект, как и в команде find_package(). В CMake 3.1 или более поздней версии CMAKE_PREFIX_PATH, CMAKE_FRAMEWORK_PATH и CMAKE_APPBUNDLE_PATH также рассматриваются как места поиска, и ключевые слова NO_CMAKE_PATH и NO_CMAKE_ENVIRONMENT_PATH также имеют здесь то же значение. Переменная PKG_CONFIG_USE_CMAKE_PREFIX_PATH может быть установлена для изменения поведения по умолчанию в отношении того, учитываются или нет эти места поиска (она будет рассматриваться как булев переключатель для включения или выключения мест поиска), но проекты обычно должны избегать этого, если им не требуется поддержка версий CMake старше 3.1.

Опция IMPORTED_TARGET поддерживается только в CMake 3.6 или более поздней версии. При ее указании, если запрашиваемый модуль найден, то создается импортируемая цель с именем PkgConfig::<prefix>. Эта импортируемая цель будет иметь детали интерфейса, заполненные из .pc файла модуля, предоставляя такие вещи, как пути поиска заголовков, флаги компилятора и т.д. По этой причине настоятельно рекомендуется использовать эту опцию, если минимальная версия CMake, необходимая для проекта, составляет 3.6 или более позднюю.

Функции ожидают один или несколько аргументов moduleSpec для определения того, что искать. Они объединяют имя пакета/модуля, который нужно найти, с любыми требованиями к версии. Поддерживаются следующие формы:

- moduleName
- moduleName=version
- moduleName>=version
- moduleName<=version

Обратите внимание на отсутствие простых операторов сравнения > или <, единственными поддерживаемыми операторами неравенства являются >= и <=. Если требование версии не включено, принимается любая версия. По возвращении функции устанавливают ряд переменных в вызывающей области видимости, вызывая pkg-config с соответствующей опцией (опциями) для извлечения только соответствующей части сведений о пакете:

Variable	pkg-config options used
prefix_LIBRARIES	--libs-only-l
prefix_LIBRARY_DIRS	--libs-only-L
prefix_LDFLAGS	--libs
prefix_LDFLAGS_OTHER	--libs-only-other
prefix_INCLUDE_DIRS	--cflags-only-I
prefix_CFLAGS	--cflags
prefix_CFLAGS_OTHER	--cflags-only-other
prefix_STATIC_LIBRARIES	--static --libs-only-l
prefix_STATIC_LIBRARY_DIRS	--static --libs-only-L
prefix_STATIC_LDFLAGS	--static --libs
prefix_STATIC_LDFLAGS_OTHER	--static --libs-only-other
prefix_STATIC_INCLUDE_DIRS	--static --cflags-only-I
prefix_STATIC_CFLAGS	--static --cflags
prefix_STATIC_CFLAGS_OTHER	--static --cflags-only-other

Если набор опций возвращает несколько элементов (например, несколько библиотек или несколько путей поиска), соответствующая переменная будет содержать список CMake.
Перечисленные выше переменные устанавливаются только в том случае, если требования модуля удовлетворены. Каноническим способом проверки этого является использование переменных prefix_FOUND и prefix_STATIC_FOUND. Для pkg_check_modules() все требования moduleSpec должны быть удовлетворены, чтобы эти переменные имели значение true, тогда как pkg_search_module() должна найти только один подходящий moduleSpec.

Для pkg_check_modules() некоторые дополнительные переменные для каждого модуля также устанавливаются, когда модули успешно найдены. Если задан только один moduleSpec, то YYY = prefix, в противном случае YYY = prefix_moduleName.

YYY_VERSION

Версия найденного модуля, извлеченная из вывода опции --modversion.

YYY_PREFIX

Каталог префикса модуля. Он получается путем запроса переменной с именем prefix, которая обычно определяется в большинстве файлов .pc и которую pkg-config предоставляет по умолчанию в любом случае.

YYY_INCLUDEDIR

Результат запроса переменной с именем includedir. Это обычная, но не обязательная переменная.

YYY_LIBDIR

Результат запроса переменной с именем libdir. Опять же, это обычная, но не обязательная переменная.

В CMake 3.4 и более поздних версиях модуль FindPkgConfig предоставляет дополнительную функцию, которую можно использовать для извлечения произвольных переменных из файлов .pc:

```
pkg_get_variable(resultVar moduleName variableName)
```

Это используется внутри pkg_check_modules() для запроса переменных prefix, includedir и libdir, но проекты могут использовать его для запроса значения любой произвольной переменной.

Для большинства распространенных систем функции, предоставляемые модулем FindPkgConfig, работают достаточно надежно. Реализация этих функций, однако, зависит от возможностей, появившихся в pkg-config версии 0.20.0. Некоторые старые системы (например, Solaris 10) поставляются с более старыми версиями pkg-config, в результате чего все вызовы функций FindPkgConfig не могут успешно найти ни одного модуля, и в журнал не записывается сообщение об ошибке, указывающее на слишком старую версию pkg-config.

23.6. Рекомендуемые практики

Начиная с CMake 3.0, произошел сознательный переход к использованию импортируемых целей для представления внешних библиотек и программ, а не заполняемых переменных. Это позволяет рассматривать такие библиотеки и программы как единое целое, собирая вместе не только местоположение соответствующего бинарного файла, но и, в случае библиотек, связанные пути поиска заголовков, определения компилятора и другие библиотечные зависимости, которые потребуются потребляющим целям, также являются частью импортируемой цели. Это делает внешние библиотеки и программы такими же простыми для использования в проекте, как и любые другие обычные цели, определяемые проектом. Такое смещение акцентов означает, что поиск пакетов стал гораздо важнее, чем поиск отдельных файлов, путей и т.д., и проекты все больше стремятся к тому, чтобы другие проекты CMake могли использовать их как пакеты. Поиск отдельных файлов и т.д. все еще имеет свое применение, и полезно понимать, как это можно сделать, но разработчики должны рассматривать его как ступеньку к пакетам и/или импортированным целям, а не как самоцель. По возможности предпочитайте находить пакеты, а не отдельные вещи внутри пакетов.

При поиске пакетов большинство возникающих сложностей связано с ситуациями, когда несколько версий установлены в разных местах. Пользователь может не знать обо всех установленных версиях, или могут существовать ожидания относительно того, какая из них должна быть найдена раньше других. Вместо того, чтобы пытаться предсказать такие ситуации, целесообразнее не отклоняться слишком далеко от поведения поиска по умолчанию и предоставить пользователю возможность самому определять приоритеты с помощью кэша или переменных окружения. CMAKE_PREFIX_PATH обычно является наиболее удобным способом сделать это благодаря тому, что CMake автоматически ищет ряд общих компоновок каталогов под каждым указанным префиксным путем.

Все команды find_...(), кроме find_package(), работают аналогичным образом, кэшируя успешный результат, чтобы избежать повторения всей операции find в следующий раз, когда команду find_...() попросят найти то же самое. Результат кэшируется даже при нескольких вызовах CMake. Учитывая потенциально большое количество мест и записей каталогов, которые может искать каждый вызов, механизм кэширования может сэкономить нетривиальное количество времени при большом количестве таких вызовов find_...() в проекте. Однако есть, по крайней мере, два последствия этого, о которых разработчики должны знать. Во-первых, после успешного выполнения команд find_file(), find_path(), find_program() или find_library() поиск прекращается для всех последующих вызовов, даже если выполнение команды даст другой результат или если найденная ранее сущность больше не существует. Если сущность удалена, это может привести к ошибкам сборки, которые можно устранить только удалением устаревших записей из кэша. Разработчики часто просто удаляют весь кэш и собирают заново, а не пытаются выяснить, какие переменные кэша нужно удалить. Другой аспект такого поведения find, о котором следует знать разработчикам, заключается в том, что если вызов одной из этих команд find_...() не смог найти нужную сущность, поиск будет повторяться для *каждого* вызова, даже в пределах одного проекта. Неудачный вызов *не* кэшируется. Если в проекте много таких вызовов, это может замедлить работу шага configure. Поэтому разработчикам следует тщательно продумать, как проект использует команды find_...(), чтобы постараться минимизировать вероятность и количество неудачных поисков.

Ситуация с find_package() немного сложнее. Если пакет найден через модуль Find, то, скорее всего, все вышеперечисленные проблемы будут относиться и к пакету, поскольку логика, скорее всего, будет построена на других командах find_...(). Если, однако, пакет найден через режим config, то find_package() будет кэшировать успешный результат и при последующих обращениях сначала проверять это местоположение. Если пакет больше не имеет соответствующего конфигурационного файла в этом месте, команда продолжит поиск в обычном режиме. Это уникальное поведение для режима конфигурации является гораздо более надежным.

Особенно сложная ситуация, когда кэширование результатов find_...() может привести к тонким проблемам, возникает в системах непрерывной интеграции. Если используются инкрементные сборки, в которых сохраняется кэш CMake предыдущего запуска, то изменения, внесенные в проект в способ поиска, могут не отразиться в сборке. Только когда кэш CMake будет очищен, такие изменения вступят в силу. Кэширование часто также означает, что не записываются подробности о найденной сущности, поэтому результат сборки дает мало подсказок об использовании старых деталей поиска. Поэтому может возникнуть соблазн потребовать, чтобы все сборки CI собирались с нуля, но это может быть невыполнимо для длинных сборок. Стратегия, которая может помочь уменьшить проблему, заключается в том, чтобы запланировать ежедневную сборку в период низкой загрузки CI, когда дерево сборки очищается, а затем проект собирается как обычно. Это позволит сохранить инкрементное поведение в обычные часы, и обычно любые проблемы, связанные с кэшем, разрешаются сами собой в течение дня. Эффективность этой стратегии снижается в периоды, когда изменения вносятся в ветку, а CI-сборки чередуются между этой и другими ветками, но можно надеяться, что такие периоды встречаются не часто и с ними можно мириться, если разработчики будут осведомлены о возможных последствиях в это время.

К функциям реестра пакетов команды find_package() следует подходить с осторожностью. Они могут дать неожиданные результаты для систем непрерывной интеграции, где проекты могут захотеть найти пакеты, которые также собраны на одной машине. К сожалению, не существует переменной окружения, которую можно установить для отключения использования реестров, но это может быть сделано самими проектами путем установки CMAKE_FIND_PACKAGE_NO_PACKAGE_REGISTRY CMake переменной в OFF (задания CI обычно не имеют необходимых разрешений для изменения системного реестра пакетов, поэтому установка CMAKE_FIND_PACKAGE_NO_SYSTEM_PACKAGE_REGISTRY также не требуется). На практике лишь немногие проекты делают записи в реестр пакетов, поэтому, если не известно, что такой проект может использовать систему CI, необходимость добавления этой переменной CMake в каждый потенциально затрагиваемый проект невелика. Проекты также должны избегать вызовов export(PACKAGE) в заданиях CI (возможно, они вообще должны избегать таких вызовов).

Модуль FindPkgConfig следует использовать только в тех ситуациях, когда find_package() не подходит. Обычно это относится к пакетам, для которых CMake предоставляет модуль find, но этот модуль find довольно старый и не предоставляет импортируемых целей, или когда он отстает от более свежих релизов пакетов. Модуль FindPkgConfig также полезен для поиска пакетов, о которых CMake ничего не знает, и где пакет не предоставляет свой собственный файл конфигурации CMake, но предоставляет файл pkg-config (т.е. .pc).

При использовании файла цепочки инструментов для кросс-компиляции предпочтительнее устанавливать CMAKE_SYSROOT, а не CMAKE_FIND_ROOT_PATH. Хотя оба параметра одинаково влияют на пути поиска различных команд find_...(), только CMAKE_SYSROOT также гарантирует, что флаги компилятора и компоновщика будут правильно дополнены, чтобы включение заголовков и компоновка библиотек работали правильно.

В сценариях кросс-компиляции также типично, что поиск программ ожидает найти двоичные файлы, которые будут работать на хосте, в то время как поиск файлов и библиотек обычно ожидает найти вещи для цели. Поэтому очень часто в файлах цепочки инструментов можно увидеть следующее, чтобы обеспечить такое поведение по умолчанию:

```
set(CMAKE_FIND_ROOT_PATH_MODE_PROGRAM NEVER)
set(CMAKE_FIND_ROOT_PATH_MODE_LIBRARY ONLY)
set(CMAKE_FIND_ROOT_PATH_MODE_INCLUDE ONLY)
```

Можно утверждать, что это должно быть задано в проекте, а не полагаться на то, что оно задано в файле цепочки инструментов, поскольку технически разработчик волен использовать любой файл цепочки инструментов по своему усмотрению, и именно проект неявно полагается на поведение по умолчанию, которое он затем решает отменить или нет. Дополнительной сложностью здесь является то, что файлы цепочки инструментов перечитываются при каждом вызове project() или enable_language(), поэтому если проект хочет применить определенную комбинацию значений по умолчанию, ему придется делать это после каждого такого вызова. Поэтому разумный компромисс заключается в том, чтобы проекты включали приведенный выше блок перед первым вызовом project(), а авторы цепочек инструментов также включали его. Если авторы цепочки инструментов не включают такой блок, то, по крайней мере, проект все равно получает разумные значения по умолчанию, но если файл цепочки инструментов изменит значения по умолчанию на что-то другое, то, по крайней мере, они будут последовательно применяться во всем проекте. Разработчики должны быть очень осторожны с использованием настроек, отличных от тех, что показаны в приведенном выше примере, поскольку это настолько распространенная модель, что проекты часто ее используют.

В ситуациях, когда разработчик может переключаться между сборками для устройства и симулятора без повторного запуска CMake (например, при использовании Xcode для проекта iOS), избегайте вызовов find_library(). Любые результаты, полученные в результате таких вызовов, могут указывать только на одну из библиотек устройства или симулятора, но не на обе. Добавьте в таких случаях флаги компоновщика, которые связывают только по имени, а не по пути, например -framework ARKit или -lz. Если фреймворки или библиотеки не могут быть найдены по пути поиска компоновщика по умолчанию, проекту также необходимо предоставить опции компоновщика для расширения путей поиска, чтобы их можно было найти.

Довольно часто в примерах и сообщениях в блогах в Интернете можно встретить противоречивые рекомендации по поводу того, следует ли использовать CMAKE_MODULE_PATH или CMAKE_PREFIX_PATH для управления поиском в CMake. Простой способ запомнить разницу заключается в том, что CMAKE_MODULE_PATH используется CMake только при поиске файлов FindXXX.cmake. Для всего остального, включая поиск файлов конфигурации, используется CMAKE_PREFIX_PATH.

Глава 24. Тестирование

Естественным продолжением создания проекта является тестирование созданных в нем артефактов. В программный пакет CMake входит инструмент CTest, который можно использовать для автоматизации этапа тестирования или даже всего процесса конфигурирования, сборки, тестирования и даже отправки результатов на информационную панель. В этой главе мы рассмотрим более простой случай использования CMake для определения тестов и их выполнения с помощью инструмента командной строки ctest. Автоматизация всего процесса конфигурирования-сборки-тестирования использует многие из этих же знаний и обсуждается далее в главе.

24.1. Определение и выполнение простого теста

Первым шагом к определению тестов в проекте CMake является вызов enable_testing() где-нибудь в файле CMakeLists.txt верхнего уровня. Обычно это делается в самом начале, вскоре после первого вызова функции project(). Эффект этой функции заключается в том, что она направляет CMake на запись входного файла CTest в файл
CMAKE_CURRENT_BINARY_DIR с деталями всех тестов, определенных в проекте (более точно, тесты, определенные в текущем каталоге и ниже). enable_testing() может быть вызвана в подкаталоге без ошибки, но без вызова enable_testing() на верхнем уровне, входной файл CTest не будет создан в верхней части дерева сборки, где он обычно и должен быть.

Определение отдельных тестов выполняется с помощью команды add_test():

```
add_test(NAME testName
         COMMAND command [arg...]
         [CONFIGURATIONS config1 [config2...]]
         [WORKING_DIRECTORY dir]
)
```

Эта команда добавляет новый тест под названием testName, который запускает указанную команду с заданными аргументами. По умолчанию тест будет считаться пройденным, если команда вернет код выхода 0, но поддерживается более гибкая обработка прохождения/непрохождения, о которой пойдет речь в следующем разделе.

Команда может быть полным путем к исполняемому файлу или именем цели исполняемого файла, определенной в проекте. Если используется имя цели, CMake автоматически подставит реальный путь к исполняемому файлу. Это особенно полезно при использовании генераторов мультиконфигурации, таких как Xcode или Visual Studio, где расположение исполняемого файла зависит от конфигурации. Ниже показан минимальный пример проекта верхнего уровня, который использует преимущества этого поведения:

```
cmake_minimum_required(VERSION 3.0)
project(CTestExample)
enable_testing()

add_executable(testapp testapp.cpp)
add_test(NAME noArgs COMMAND testapp)
```

Автоматическая замена цели на ее реальное местоположение не распространяется на аргументы команды, такую замену поддерживает только сама команда. Если местоположение цели необходимо указать в качестве аргумента командной строки, можно использовать выражения-генераторы. Например:

```
add_executable(app1 ...)
add_executable(app2 ...)

add_test(NAME withArgs COMMAND app1 $<TARGET_FILE:app2>)
```

При запуске тестов пользователь может указать, какая конфигурация должна быть протестирована. Если в проекте используется один генератор конфигурации, конфигурация не обязательно должна соответствовать типу сборки. В частности, если конфигурация не указана, предполагается пустая конфигурация. Без дополнительного ключевого слова CONFIGURATIONS тест будет выполняться для всех конфигураций, независимо от типа сборки или от того, какая конфигурация была запрошена пользователем. Если ключевое слово CONFIGURATIONS указано, тест будет выполнен только для тех конфигураций, которые указаны в списке. Обратите внимание, что пустая конфигурация все равно считается валидной, поэтому для запуска теста в этом сценарии пустая строка должна быть одной из перечисленных CONFIGURATIONS.

Например, чтобы добавить тест, который должен выполняться только для конфигураций, содержащих отладочную информацию, можно перечислить конфигурации Debug и RelWithDebInfo. Добавление пустой строки также заставляет тест выполняться, если при запуске тестов не указана конфигурация:

```
add_test(NAME debugOnly
    COMMAND testapp
    CONFIGURATIONS Debug RelWithDebInfo ""
)
```

В большинстве случаев ключевое слово CONFIGURATIONS не нужно, и тест будет выполнен для всех конфигураций, включая пустую.

По умолчанию тест будет запущен в каталоге CMAKE_CURRENT_BINARY_DIR, но опция WORKING_DIRECTORY может быть использована, чтобы заставить тест запускаться в другом месте. Например, это может быть полезно для запуска одного и того же исполняемого файла в разных каталогах для получения различных наборов входных файлов без необходимости указывать их в качестве аргументов командной строки.

```
add_test(NAME foo
    COMMAND testapp
    WORKING_DIRECTORY ${CMAKE_CURRENT_LIST_DIR}/foo
)
add_test(NAME bar
    COMMAND testapp
    WORKING_DIRECTORY ${CMAKE_CURRENT_LIST_DIR}/bar
)
```

При указании рабочего каталога всегда используйте абсолютный путь. Если указан относительный путь, он будет интерпретирован как путь к директории, в которой был запущен сам ctest, но это может быть не вершина дерева сборки. Чтобы обеспечить предсказуемость рабочего каталога, проекты должны избегать использования относительного WORKING_DIRECTORY.

Если указанный рабочий каталог не существует на момент запуска теста, CMake версии 3.11 и более ранние не выдадут сообщения об ошибке и выполнят тест, даже если не удастся изменить рабочий каталог. CMake 3.12 и более поздние версии перехватывают ошибку и рассматривают тест как неудачный. Независимо от того, какая версия CMake используется, проект несет ответственность за то, чтобы рабочий каталог существовал и имел соответствующие разрешения.

Урезанная форма команды add_test() также поддерживается по причинам обратной совместимости:

```
add_test(testName command [args...])
```

Эту форму не следует использовать в новых проектах, поскольку она лишена некоторых возможностей полной формы NAME и COMMAND. Основные отличия заключаются в том, что выражения генераторов не поддерживаются, и если command - это имя цели, CMake не будет автоматически подставлять местоположение ее двоичного файла.

Для запуска тестов используется инструмент командной строки ctest, который обычно запускается из верхней части каталога сборки. При запуске без аргументов командной строки он выполнит все определенные тесты по одному, регистрируя сообщение о состоянии при запуске и завершении каждого теста, но скрывая весь вывод тестов. В конце будет выведено общее резюме тестов. Типичный вывод будет выглядеть следующим образом:

```
Test project /path/to/build/dir
    Start 1: fooWithBar
1/2 Test #1: fooWithBar..............   Passed    0.00 sec
    Start 2: fooWithoutBar
2/2 Test #2: fooWithoutBar...........   Passed    0.00 sec

100% tests passed, 0 tests failed out of 2

Total Test time (real) =   0.02 sec
```

При использовании генератора нескольких конфигураций, например Xcode или Visual Studio, ctest необходимо указать, какую конфигурацию он должен тестировать. Это делается путем включения опции -C configType, где configType будет одним из поддерживаемых типов сборки (Debug, Release и т.д.). Для генераторов одиночных конфигураций опция -C не обязательна, поскольку сборка может создавать только одну конфигурацию, поэтому нет никакой двусмысленности в том, где найти двоичные файлы для выполнения. Тем не менее, может быть полезно указать конфигурацию, чтобы избежать менее интуитивного поведения исключения тестов, которые определены для запуска только при определенных конфигурациях и где пустая строка не входит в список.

С помощью опции -V можно указать ctest показывать все результаты тестирования и различные другие подробности о выполнении. Опции -VV и -VVV показывают все более высокий уровень многословности, но они обычно нужны только разработчикам, работающим над самим ctest. Даже уровень многословности -V обычно показывает больше деталей, чем хотят видеть пользователи, скорее всего, интерес представляет только вывод тестов, которые не прошли. ctest можно попросить показывать только вывод неудачных тестов, передав опцию --output-on-failure. В качестве альтернативы, разработчики могут установить переменную окружения CTEST_OUTPUT_ON_FAILURE в любое значение, чтобы избежать необходимости указывать его каждый раз (значение не используется, ctest просто проверяет, установлен ли CTEST_OUTPUT_ON_FAILURE).

По умолчанию каждый тест запускается с тем же окружением, что и команда ctest. Если тест требует изменения окружения, это можно сделать с помощью свойства теста ENVIRONMENT. Это свойство должно представлять

собой список элементов NAME=VALUE, определяющих переменные окружения, которые должны быть установлены перед запуском теста. Изменения локальны только для данного теста и не влияют на другие тесты.

```
set_tests_properties(fooWithoutBar PROPERTIES
    ENVIRONMENT "FOO=bar;HAVE_BAZ=1"
)
```

Ситуации, когда переменная окружения должна модифицировать, а не заменять существующее значение, менее просты. Если окружение должно быть основано на том, в котором запускается CMake, а не команда ctest, то для получения существующих значений можно использовать форму $ENV{SOMEVAR}. Хорошим примером этого является дополнение переменной окружения PATH для того, чтобы тест мог найти разделяемые библиотеки, на которые он ссылается в Windows:

```
# In this example, algo is assumed to be a shared library defined elsewhere
# in the project and whose binary will be in a different directory to fooTest
add_executable(fooTest ...)
target_link_libraries(fooTest PRIVATE algo)

add_test(NAME fooWithAlgo COMMAND fooTest)

if(WIN32)
    set_tests_properties(fooWithAlgo PROPERTIES ENVIRONMENT
        "PATH=$<SHELL_PATH:$<TARGET_FILE_DIR:algo>>$<SEMICOLON>$ENV{PATH}"
    )
endif()
```

Модификация окружения на основе фактического окружения, используемого для вызова ctest, а не CMake, более сложна и обычно не является строго необходимой. Этого можно добиться с помощью комбинации cmake -E env, вызывающей скрипт, при этом места, предоставляемые CMake, передаются в качестве переменных в часть cmake -E env, а затем скрипт выполняет фактическую задачу дополнения среды выполнения, используя эти значения и вызывая исполняемый файл теста. Такая схема сложна, может быть хрупкой и ее следует избегать, если только нет конкретной необходимости в поддержке такого варианта использования.

В качестве удобства, прежде всего для приложений IDE, когда тестирование включено, CMake определяет пользовательскую цель сборки, которая вызывает ctest с набором аргументов по умолчанию. Для генераторов с несколькими конфигурациями, таких как Xcode и Visual Studio, эта цель будет называться RUN_TESTS и будет передавать текущий выбранный тип сборки в качестве конфигурации для ctest. Для генераторов с одной конфигурацией цель называется просто test и не указывает никакой конфигурации при вызове ctest. Нет возможности указать, какие тесты будут выполняться, или какие-либо другие пользовательские опции передать в ctest при использовании цели RUN_TESTS или тестовой сборки.

24.2. Критерии прохождения/непрохождения и другие типы результатов

Определение результата теста только по коду выхода команды теста может быть довольно ограничительным. Другой поддерживаемой альтернативой является указание регулярных выражений для сопоставления с результатами теста. Свойство теста PASS_REGULAR_EXPRESSION можно использовать для указания списка регулярных выражений, хотя бы одно из которых должно совпасть с выводом теста, чтобы тест прошел. Эти

регулярные выражения часто охватывают несколько строк. Аналогично, свойство теста FAIL_REGULAR_EXPRESSION может быть установлено в список регулярных выражений. Если любое из них совпадает с выводом теста, тест проваливается, даже если вывод также соответствует PASS_REGULAR_EXPRESSION или код выхода равен 0. Тест может иметь как PASS_REGULAR_EXPRESSION, так и FAIL_REGULAR_EXPRESSION, только одно из двух или ни одного из них. Если PASS_REGULAR_EXPRESSION установлен и не является пустым, код выхода не учитывается при определении того, прошел тест или не прошел.

```
# Ignore exit code, check output to determine the pass/fail status
set_tests_properties(fooTest PROPERTIES
PASS_REGULAR_EXPRESSION
"Checking some condition for fooTest: passed
+.*
All checks passed"
FAIL_REGULAR_EXPRESSION "warning|Warning|WARNING"
)
```

Иногда тест необходимо пропустить, возможно, по причинам, которые может определить только сам тест. Свойство теста SKIP_RETURN_CODE может быть установлено в значение, которое тест может вернуть, чтобы указать, что он был пропущен, а не провален. Тест, который завершается со значением SKIP_RETURN_CODE, отменяет любые другие критерии прохождения/непрохождения.

fooTest.cpp

```
int main(int argc, char* argv[])
{
    if (shouldSkip())
        return 2; // Skipped

    if (runTest())
        return 0; // Passed

    return 1; // Failed
}
```

CMakeLists.txt

```
add_executable(fooTest fooTest.cpp ...)
add_test(NAME foo COMMAND fooTest)

set_tests_properties(foo PROPERTIES
    SKIP_RETURN_CODE 2
)
```

Результаты вышеприведенного теста могут выглядеть следующим образом:

```
Test project /path/to/build/dir
    Start 1: foo
1/1 Test #1: foo ..................***Skipped   0.00 sec

100% tests passed, 0 tests failed out of 1

Total Test time (real) =   0.01 sec

The following tests did not run:
      1 - foo (Skipped)
```

Если хотя бы один тест не прошел или не выполняется по какой-то причине, в конце выводится сводка всех таких тестов и их статус. Тест, который своим кодом возврата указывает на то, что его следует пропустить, не считается неудачным и по-прежнему учитывается в общем количестве тестов. Тест может быть пропущен по другим причинам, которые можно считать неудачей, например, если не выполнена зависимость теста (обсуждается в разделе 24.5, "Зависимости теста", ниже).

В CMake 3.9 или более поздней версии также поддерживается свойство теста DISABLED. С его помощью можно пометить тест как временно отключенный, что позволит ему быть определенным, но не выполняться и даже не учитываться в общем количестве тестов. Это не будет считаться неудачей теста, но он все равно будет показан в результатах теста с соответствующим сообщением о состоянии. Обратите внимание, что такие тесты обычно не должны оставаться отключенными в течение длительного времени, эта функция предназначена для временного отключения проблемного или незавершенного теста до тех пор, пока он не будет исправлен.

Следующий простой пример демонстрирует поведение теста DISABLED:

```
add_test(NAME fooWithBar ...)
add_test(NAME fooWithoutBar ...)
set_tests_properties(fooWithoutBar PROPERTIES DISABLED YES)
```

Вывод ctest для приведенного выше результата может выглядеть примерно так:

```
Test project /path/to/build/dir
    Start 1: fooWithBar
1/2 Test #1: fooWithBar ..............   Passed    0.00 sec
    Start 2: fooWithoutBar
2/2 Test #2: fooWithoutBar ...........***Not Run (Disabled)   0.00 sec

100% tests passed, 0 tests failed out of 1

Total Test time (real) =   0.01 sec

The following tests did not run:
      2 - fooWithoutBar (Disabled)
```

В некоторых случаях ожидается, что тест может оказаться неудачным. Вместо того чтобы отключать тест, целесообразнее пометить его как ожидающий неудачи, чтобы он продолжал выполняться. Для этого свойство теста WILL_FAIL можно установить в true, что инвертирует результат прохождения/непрохождения. Это имеет дополнительное преимущество: если тест неожиданно начнет проходить, ctest будет считать это неудачей, и разработчик сразу же узнает об изменении поведения.

Еще один аспект статуса прохождения/непрохождения теста - это время, необходимое для его завершения. Свойство теста TIMEOUT, если оно установлено, определяет количество секунд, в течение которых тест будет выполняться, прежде чем он будет завершен и помечен как неудачный. Командная строка ctest также принимает опцию --timeout, которая имеет тот же эффект для любого теста без установленного свойства TIMEOUT (т.е. действует как таймаут по умолчанию). Кроме того, ограничение по времени может быть применено ко всему набору тестов в целом путем указания опции --stop-time в ctest. Аргумент после --stop-time должен быть реальным временем суток, а не числом секунд, причем если не указан часовой пояс, то предполагается местное время.

```
add_test(NAME t1 COMMAND ...)
add_test(NAME t2 COMMAND ...)
set_tests_properties(t2 PROPERTIES TIMEOUT 10)
```

```
ctest --timeout 30 --stop-time 13:00
```

В приведенном выше примере таймаут для каждого теста по умолчанию установлен в 30 секунд в командной строке ctest. Поскольку у t1 не установлено свойство TIMEOUT, он будет иметь таймаут 30 секунд, в то время как у t2 свойство TIMEOUT установлено на 10, что отменит значение по умолчанию, установленное в командной строке ctest. Тесты будут завершены до 13:00 по местному времени.

В некоторых обстоятельствах тесту может потребоваться подождать определенного условия, прежде чем начнется собственно тест. Может оказаться желательным применить таймаут только к части выполнения после того, как это условие будет выполнено и начнется настоящий тест. В CMake 3.6 или более поздней версии для поддержки такого поведения доступно свойство теста TIMEOUT_AFTER_MATCH. Оно ожидает список, содержащий два элемента, первый из которых - количество секунд, используемое в качестве тайм-аута после выполнения условия, а второй - регулярное выражение, которое должно быть сопоставлено с выводом теста. Когда регулярное выражение найдено, отсчет времени ожидания и время начала теста сбрасываются, а значение тайм-аута устанавливается на первый элемент списка. Например, в следующем примере для теста будет установлен общий тайм-аут в 30 секунд, но когда в выводе теста появится строка Condition met, у теста будет 10 секунд для завершения, и первоначальное условие тайм-аута в 30 секунд больше не будет применяться.

```
set_tests_properties(t2 PROPERTIES
    TIMEOUT 30
    TIMEOUT_AFTER_MATCH "10;Condition met"
)
```

Если на выполнение условия у теста ушло 25 секунд, общее время теста может составить 35 секунд, но поскольку время начала теста также сбрасывается, ctest покажет время от 0 до 10 секунд (т.е. время выполнения условия не учитывается). С другой стороны, если условие не будет выполнено в течение 30 секунд, тест покажет общее время тестирования около 30 секунд.

По возможности, следует избегать использования TIMEOUT_AFTER_MATCH в пользу других способов обработки предварительных условий. В разделе 24.5, "Зависимости тестов" и разделе 24.4, "Параллельное выполнение" ниже обсуждаются лучшие альтернативные методы.

24.3. Группировка и отбор тестов

В больших проектах довольно часто возникает желание запустить только подмножество всех определенных тестов. Разработчик может быть сосредоточен на конкретном неудачном тесте и может не интересоваться всеми остальными тестами во время работы над этой проблемой. Один из способов выполнить только определенное подмножество тестов - задать опции -R и -E для ctest. Каждая из этих опций задает регулярное выражение для сопоставления с именами тестов. Опция -R выбирает тесты для включения в набор тестов, тогда как опция -E исключает тесты. Обе опции могут быть указаны для комбинирования их эффектов.

```
add_test(NAME fooOnly    COMMAND ...)
add_test(NAME barOnly    COMMAND ...)
add_test(NAME fooWithBar COMMAND ...)
add_test(NAME fooSpecial COMMAND ...)
add_test(NAME other_foo  COMMAND ...)
```

```
ctest -R Only                    # Run just fooOnly and barOnly
ctest -E Bar                     # Run all but fooWithBar
ctest -R '^foo' -E fooSpecial    # Run all tests starting with foo except fooSpecial
ctest -R 'fooSpecial|other_foo'  # Run only fooSpecial and other_foo
```

Иногда не всегда легко разработать регулярное выражение для захвата только нужных тестов, или разработчик может просто захотеть увидеть все тесты, которые были определены, не запуская их. Опция -N указывает ctest только печатать тесты, а не запускать их, что может быть полезным способом проверки того, что регулярные выражения дают нужный набор тестов.

```
ctest -N

Test project /path/to/build/dir
  Test #1: fooOnly
  Test #2: barOnly
  Test #3: fooWithBar
  Test #4: fooSpecial
  Test #5: other_foo

Total Tests: 5
```

```
ctest -N -R 'fooSpecial|other_foo'

Test project /path/to/build/dir
Test #4: fooSpecial
Test #5: other_foo

Total Tests: 2
```

При добавлении каждого теста ему присваивается номер теста, который будет оставаться неизменным между запусками, если до него в проект не будет добавлен или удален другой тест. В выводе ctest этот номер указывается рядом с тестом. При использовании опции -N тесты перечисляются в том порядке, в котором они

были определены в проекте, но тесты не обязательно будут выполняться в этом порядке. Тесты для выполнения могут быть выбраны по номеру теста, а не по имени с помощью опции -I. Этот метод довольно хрупок, поскольку добавление или удаление одного теста может изменить номер, присвоенный любому другому тесту. Даже передача другой конфигурации через опцию -C в ctest может привести к изменению номеров тестов. В большинстве случаев предпочтительнее использовать поиск по имени.

Одна из ситуаций, когда номера тестов могут быть полезны, это когда двум тестам дано одинаковое имя. Если оба теста определены в одном каталоге, они принимаются без предупреждений. Хотя дублирования имен тестов обычно следует избегать, в иерархических проектах с тестами, предоставляемыми извне, это не всегда возможно.

Опция -I ожидает аргумент, который имеет несколько сложную форму. Самая прямая форма предполагает указание номеров тестов в командной строке, разделенных запятыми без пробелов:

```
ctest -I [start[,end[,stride[,testNum[,testNum...]]]]]
```

Чтобы указать только номера отдельных тестов, начало, конец и шаг можно оставить пустыми, как показано ниже:

```
ctest -I ,,,3,2     # Selects tests 2 and 3 only
```

Те же самые данные можно прочитать из файла вместо того, чтобы указывать их в командной строке, задав имя файла в опции -I. Это может быть полезно, если регулярно выполняется один и тот же сложный набор тестов, при этом тесты не добавляются и не удаляются:

testNumbers.txt

```
,,,3,2
```

```
ctest -I testNumbers.txt
```

Выбор тестов по имени или номеру может стать громоздким, если необходимо выполнить большой набор связанных тестов. Тестам можно присвоить произвольный список меток с помощью свойства теста LABELS, а затем выбирать тесты по этим меткам. Опции -L и -LE аналогичны опциям -R и -E соответственно, за исключением того, что они работают с метками тестов, а не с их именами. Продолжаем с теми же тестами, определенными в предыдущем примере:

```
set_tests_properties(fooOnly    PROPERTIES LABELS "foo")
set_tests_properties(barOnly    PROPERTIES LABELS "bar")
set_tests_properties(fooWithBar PROPERTIES LABELS "foo;bar;multi")
set_tests_properties(fooSpecial PROPERTIES LABELS "foo")
set_tests_properties(other_foo  PROPERTIES LABELS "foo")
```

```
ctest -L bar

Test project /path/to/build/dir
    Start 2: barOnly
1/2 Test #2: barOnly ..........................   Passed    1.52 sec
    Start 3: fooWithBar
2/2 Test #3: fooWithBar .......................   Passed    1.02 sec

100% tests passed, 0 tests failed out of 2

Label Time Summary:
bar      =   2.53 sec*proc (2 tests)
foo      =   1.02 sec*proc (1 test)
multi    =   1.02 sec*proc (1 test)

Total Test time (real) =   2.54 sec
```

Метки не только позволяют удобно группировать тесты для выполнения, они также обеспечивают группировку для получения базовой статистики времени выполнения. Как видно из приведенного выше примера, команда ctest печатает сводку по меткам, когда для любого теста из набора выполненных тестов установлено свойство LABELS. Это позволяет разработчику получить представление о том, какой вклад вносит каждая группа меток в общее время выполнения теста. Часть proc в единицах sec*proc относится к количеству процессоров, выделенных тестам (описано в разделе 24.4, "Параллельное выполнение" ниже). Тест, выполняющийся в течение 3 секунд и требующий 4 процессора, покажет значение 12. Сводка времени по меткам может быть подавлена с помощью опции --no-label-summary.

Другой распространенной необходимостью является повторный запуск только тех тестов, которые не прошли при последнем запуске ctest. Это может быть удобным способом повторной проверки только соответствующих тестов после внесения небольшого исправления или повторного запуска тестов, которые не прошли из-за временных условий окружающей среды. Команда ctest поддерживает опцию --rerun -failed, которая обеспечивает такое поведение без необходимости указания имен, номеров или меток тестов.

Иногда конкретный тест или набор тестов дает сбои только периодически, поэтому тест(ы) может потребоваться запустить много раз, чтобы попытаться воспроизвести сбой. Вместо того, чтобы запускать сам ctest снова и снова, можно задать опцию --repeat-until-fail, указав верхний предел количества повторений каждого теста. Если тест не прошел, он не будет повторно выполняться для данного вызова ctest.

```
ctest -L bar --repeat-until-fail 3

Test project /path/to/build/dir
    Start 2: barOnly
    Test #2: barOnly ..........................   Passed    1.52 sec
    Start 2: barOnly
    Test #2: barOnly ..........................***Failed    0.00 sec
    Start 3: fooWithBar
    Test #3: fooWithBar .......................   Passed    1.02 sec
    Start 3: fooWithBar
    Test #3: fooWithBar .......................   Passed    1.02 sec
    Start 3: fooWithBar
2/2 Test #3: fooWithBar .......................   Passed    1.02 sec

50% tests passed, 1 tests failed out of 2

Label Time Summary:
bar      =   1.02 sec*proc (2 tests)
foo      =   1.02 sec*proc (1 test)
multi    =   1.02 sec*proc (1 test)

Total Test time (real) =   4.59 sec

The following tests FAILED:
          2 - barOnly (Failed)
Errors while running CTest
```

Сводка по меткам не накапливает общее время для повторных тестов, она использует только время последнего выполнения теста. Однако общее время теста учитывает все повторения.

24.4. Параллельное выполнение

Максимизация пропускной способности тестов может быть важным фактором для больших проектов или там, где тесты занимают нетривиальное количество времени. Возможность параллельного запуска тестов является ключевой особенностью ctest и включается с помощью опций командной строки, которые очень похожи на стандартный инструмент make. Опция -j может быть использована для указания верхнего предела на количество одновременно выполняемых тестов. В отличие от большинства реализаций make, значение должно быть указано, иначе опция не будет иметь никакого эффекта. В качестве альтернативы, переменная окружения CTEST_PARALLEL_LEVEL может быть использована для указания количества заданий, но опция командной строки имеет приоритет, если используются обе. Такая схема особенно полезна для ведомых устройств сборки с непрерывной интеграцией, поскольку CTEST_PARALLEL_LEVEL может быть установлена на количество ядер процессора на каждом ведомом устройстве, освобождая каждый проект от необходимости самостоятельно вычислять оптимальное количество заданий. Для тех проектов, которым необходимо ограничить количество параллельных заданий, они все еще могут переопределить CTEST_PARALLEL_LEVEL с помощью опции командной строки -j.

Связанная опция -l используется для указания желаемого верхнего предела нагрузки на процессор. ctest попытается избежать запуска нового теста, если нагрузка может превысить этот предел. К сожалению, недостатки этой опции сразу же проявляются в начале тестирования. Как правило, ctest изначально запускает столько тестов, сколько позволяет лимит заданий из параметров -j или CTEST_PARALLEL_LEVEL, превышая любой лимит, заданный параметром -l. Измеренная нагрузка на процессор обычно имеет запаздывание, что

позволяет ctest запускать слишком много тестов вначале, прежде чем измеренная нагрузка увеличится. Чтобы этого не произошло, количество параллельных заданий, заданное параметрами -j или CTEST_PARALLEL_LEVEL, должно быть не более предела, налагаемого параметром -l. Если не заданы ни -j, ни CTEST_PARALLEL_LEVEL, опция -l не будет иметь никакого эффекта. Несмотря на эти ограничения, опция -l может быть полезна для снижения перегрузки процессора в системах с общим доступом, где другие процессы также могут конкурировать за ресурсы процессора.

По умолчанию ctest предполагает, что каждый тест потребляет один процессор. Для тестов, использующих более одного процессора, свойство теста PROCESSORS может быть установлено, чтобы указать, сколько процессоров они должны использовать. ctest будет использовать это значение при определении того, достаточно ли ресурсов процессора свободно перед запуском теста. Если PROCESSORS установлено в значение, превышающее предел задания, ctest будет вести себя так, как если бы оно было установлено в предел задания при определении возможности запуска теста.

Влияние этих опций можно увидеть в следующих примерах вывода, в которых используется тот же набор тестов, который был определен ранее.

```
ctest -j 5

Test project /path/to/build/dir
    Start 5: other_foo
    Start 2: barOnly
    Start 3: fooWithBar
    Start 1: fooOnly
    Start 4: fooSpecial
1/5 Test #4: fooSpecial .......................   Passed    0.12 sec
2/5 Test #1: fooOnly ..........................   Passed    0.52 sec
3/5 Test #3: fooWithBar .......................   Passed    1.01 sec
4/5 Test #2: barOnly ..........................   Passed    1.52 sec
5/5 Test #5: other_foo ........................   Passed    2.02 sec
                                                                continued...
```

```
100% tests passed, 0 tests failed out of 5

Label Time Summary:
bar      =   2.53 sec*proc (2 tests)
foo      =   1.65 sec*proc (3 tests)
multi    =   1.01 sec*proc (1 test)

Total Test time (real) =   2.03 sec
```

Было определено пять тестов, и в командной строке был задан лимит заданий - 5, поэтому ctest смог запустить все тесты немедленно. Результат каждого теста записывался по мере его завершения, а не в том порядке, в котором они были запущены. Если уменьшить лимит заданий до 2, результат будет выглядеть следующим образом:

```
ctest -j 2

Test project /path/to/build/dir
    Start 5: other_foo
    Start 2: barOnly
1/5 Test #2: barOnly ..........................   Passed    1.52 sec
    Start 3: fooWithBar
2/5 Test #5: other_foo ........................   Passed    2.01 sec
    Start 1: fooOnly
3/5 Test #1: fooOnly ..........................   Passed    0.52 sec
    Start 4: fooSpecial
4/5 Test #3: fooWithBar .......................   Passed    1.02 sec
5/5 Test #4: fooSpecial .......................   Passed    0.12 sec

100% tests passed, 0 tests failed out of 5

Label Time Summary:
bar      =   2.54 sec*proc (2 tests)
foo      =   1.65 sec*proc (3 tests)
multi    =   1.02 sec*proc (1 test)

Total Test time (real) =   2.66 sec
```

При большом количестве тестов и высоком лимите заданий может быть трудно уследить за протоколированием начала и завершения каждого отдельного теста. В этом случае гораздо важнее становится общая сводка тестов в конце выполнения, где каждый тест, который не прошел, перечисляется вместе с его результатом.

Иногда тестам необходимо убедиться, что параллельно с ними не выполняется другой тест. Они могут выполнять действие, чувствительное к другим действиям на машине, или создавать условия, которые будут мешать другим тестам. Чтобы обеспечить это ограничение, свойство RUN_SERIAL теста можно установить в true. Это довольно жесткое ограничение, которое может сильно повлиять на пропускную способность теста, поэтому его следует использовать редко. Довольно часто лучшей альтернативой является свойство теста RESOURCE_LOCK, которое используется для предоставления списка ресурсов, к которым тесту необходим эксклюзивный доступ. Эти ресурсы представляют собой произвольные строки, которые ctest никак не интерпретирует, за исключением того, что ни один другой тест, у которого в свойстве RESOURCE_LOCK указан любой из этих ресурсов, не будет запущен одновременно. Это отличный способ сериализации тестов, которым нужен эксклюзивный доступ к чему-либо (например, к базе данных, общей памяти), не блокируя тесты, которые не используют этот ресурс.

```
set_tests_properties(fooOnly fooSpecial other_foo PROPERTIES RESOURCE_LOCK foo)
set_tests_properties(barOnly                      PROPERTIES RESOURCE_LOCK bar)
set_tests_properties(fooWithBar                   PROPERTIES RESOURCE_LOCK "foo;bar")
```

Следующий пример вывода показывает, что, несмотря на то, что лимит заданий 5 позволяет выполнять все тесты одновременно, ctest задерживает запуск некоторых тестов до тех пор, пока не будут доступны необходимые им ресурсы.

```
ctest -j 5

Test project /path/to/build/dir
    Start 5: other_foo
    Start 2: barOnly
1/5 Test #2: barOnly ..........................   Passed    1.52 sec
2/5 Test #5: other_foo ........................   Passed    2.02 sec
    Start 3: fooWithBar
3/5 Test #3: fooWithBar .......................   Passed    1.01 sec
    Start 1: fooOnly
4/5 Test #1: fooOnly ..........................   Passed    0.52 sec
    Start 4: fooSpecial
5/5 Test #4: fooSpecial .......................   Passed    0.12 sec

100% tests passed, 0 tests failed out of 5

Label Time Summary:
bar      =   2.53 sec*proc (2 tests)
foo      =   1.65 sec*proc (3 tests)
multi    =   1.01 sec*proc (1 test)

Total Test time (real) =   3.67 sec
```

24.5. Зависимости при тестировании

Тесты могут использоваться не только для проверки определенных условий, но и для их выполнения. Например, одному тесту может понадобиться сервер для подключения, чтобы он мог проверить реализацию клиента. Вместо того чтобы полагаться на разработчика в вопросе обеспечения наличия такого сервера, можно создать другой тестовый пример, который обеспечит работу сервера. Тогда клиентский тест должен иметь некоторую зависимость от серверного теста, чтобы убедиться, что они выполняются в правильном порядке.

Свойство теста DEPENDS позволяет выразить это ограничение в виде списка других тестов, которые должны завершиться до запуска данного теста. Приведенный выше пример клиент/сервер можно выразить следующим образом:

```
set_tests_properties(clientTest1 clientTest2 PROPERTIES DEPENDS startServer)
set_tests_properties(stopServer              PROPERTIES DEPENDS "clientTest1;clientTest2")
```

Недостатком свойства теста DEPENDS является то, что хотя оно определяет порядок тестов, оно не учитывает, прошли или не прошли пререквизиты тестов. В приведенном выше примере, если тест startServer провалится, тесты clientTest1, clientTest2 и stopServer все равно будут запущены. Эти тесты, скорее всего, будут провалены, и результаты тестирования покажут все четыре теста как проваленные, в то время как на самом деле провален только тест startServer, а остальные следовало бы пропустить.

В CMake 3.7 добавлена поддержка тестовых приспособлений - концепции, которая позволяет выражать зависимости между тестами гораздо более строго. Тест может указать, что ему требуется определенное приспособление, перечислив имя этого приспособления в свойстве теста FIXTURES_REQUIRED. Любой другой тест с таким же именем приспособления в свойстве FIXTURES_SETUP должен успешно завершиться, прежде чем будет запущен зависимый тест. Если любой из тестов установки приспособления завершится неудачно, все

тесты, требующие этого приспособления, будут помечены как пропущенные. Аналогично, тест может указать приспособление в свойстве теста FIXTURES_CLEANUP, чтобы указать, что он должен быть запущен после любого другого теста с тем же приспособлением, указанным в свойстве FIXTURES_SETUP или FIXTURES_REQUIRED. Эти тесты очистки не требуют прохождения тестов установки или требования приспособления, поскольку очистка может потребоваться даже в том случае, если предыдущие тесты не прошли.

Все три свойства тестов, связанных с приспособлениями, принимают список имен приспособлений. Эти имена произвольны и не должны иметь отношения к именам тестов, используемым ими ресурсам или каким-либо другим свойствам. Имена приспособлений должны прояснять для разработчиков, что они представляют, поэтому, хотя это и не обязательно, они часто имеют то же значение, что и имена, используемые для свойств RESOURCE_LOCK.

Рассмотрим предыдущий пример клиент/сервер. Это можно выразить строго с помощью фикстур со следующими свойствами:

```
set_tests_properties(startServer             PROPERTIES FIXTURES_SETUP    server)
set_tests_properties(clientTest1 clientTest2 PROPERTIES FIXTURES_REQUIRED server)
set_tests_properties(stopServer              PROPERTIES FIXTURES_CLEANUP  server)
```

В приведенном выше примере сервер - это имя приспособления, тесты clientTest1 и clientTest2 будут выполняться только в случае прохождения startServer, а stopServer будет выполняться последним, независимо от результата любого из трех других тестов. Если включено параллельное выполнение, startServer будет выполняться первым, два клиентских теста будут выполняться одновременно, а stopServer будет выполняться только после того, как оба клиентских теста будут завершены или пропущены.

Еще одно преимущество фикстур можно увидеть, когда разработчик выполняет только подмножество тестов. Рассмотрим сценарий, в котором разработчик работает над тестом clientTest2 и не заинтересован в выполнении теста clientTest1. Когда зависимости между тестами выражаются с помощью DEPENDS, разработчик отвечает за то, чтобы в набор тестов были включены и необходимые тесты, а это значит, что ему необходимо понимать все соответствующие зависимости. Это приведет к командной строке ctest:

```
ctest -R "startServer|clientTest2|stopServer"
```

Когда используются фикстуры, ctest автоматически добавляет любые тесты настройки или очистки в набор тестов, которые должны быть выполнены, чтобы удовлетворить требования фикстуры. Это означает, что разработчику нужно только указать тест, на котором он хочет сосредоточиться, а все зависимости оставить на усмотрение ctest:

```
ctest -R clientTest2
```

При использовании опции --rerun-failed этот же механизм обеспечивает автоматическое добавление тестов установки и очистки в набор тестов для удовлетворения зависимостей приспособлений ранее провалившихся тестов.

Приспособление может иметь ноль или более тестов настройки и ноль или более тестов очистки. Фикстура может определять тесты установки без тестов очистки и наоборот. Хотя это не особенно полезно, приспособление может вообще не иметь тестов настройки или очистки, и в этом случае приспособление не влияет на тесты, которые будут выполняться, или на то, когда они будут выполняться. Аналогично,

приспособление может иметь связанные с ним тесты настройки и/или очистки, но не иметь тестов, требующих его использования. Такие ситуации могут возникать во время разработки, когда тесты определяются или временно отключаются. В случае, когда приспособление не имеет требующих его тестов, ошибка в CMake 3.7 позволяла тестам очистки этого приспособления выполняться раньше тестов настройки, но эта ошибка была исправлена в выпуске 3.8.0.

Более сложный пример демонстрирует, как фикстуры можно использовать для выражения более сложных зависимостей между тестами. Расширяя предыдущий пример, предположим, что один клиентский тест требует только сервера, в то время как другой требует наличия и сервера, и базы данных. Это можно лаконично выразить, определив две фикстуры: сервер и база данных. Для последнего допустимо просто проверить, доступна ли база данных, и отказать, если нет, поэтому приспособление базы данных не требует проверки на чистоту. Фикстуры сервера и базы данных не связаны между собой, поэтому им не нужны зависимости между собой. Эти ограничения могут быть выражены следующим образом:

```
# Setup/cleanup
set_tests_properties(startServer       PROPERTIES FIXTURES_SETUP    server)
set_tests_properties(stopServer        PROPERTIES FIXTURES_CLEANUP  server)
set_tests_properties(ensureDbAvailable PROPERTIES FIXTURES_SETUP    database)

# Client tests
set_tests_properties(clientNoDb        PROPERTIES FIXTURES_REQUIRED server)
set_tests_properties(clientWithDb      PROPERTIES FIXTURES_REQUIRED "server;database")
```

Хотя автоматическое добавление ctest зависимостей приспособлений в набор выполнения тестов в целом является полезной функцией, бывают случаи, когда это может быть нежелательно. Продолжая приведенный выше пример, разработчик может захотеть оставить сервер запущенным и продолжать выполнять один клиентский тест несколько раз. Он может вносить изменения, перекомпилировать код и проверять, проходит ли клиентский тест при каждом изменении. Для поддержки такого уровня контроля в CMake 3.9 появились опции -FS, -FC и -FA для ctest, каждая из которых требует регулярного выражения, которое будет сопоставлено с именами фикстур. Опция -FS используется для отключения добавления зависимостей установки приспособлений для тех приспособлений, которые соответствуют заданному регулярному выражению. -FC делает то же самое для тестов очистки, а -FA объединяет оба параметра, отключая как тесты установки, так и тесты очистки, которые совпадают. Распространенной ситуацией является запрет добавления любых зависимостей установки/очистки вообще, что можно сделать, задав регулярное выражение с одной точкой (.). Ниже показаны различные примеры этих опций и их эффекты:

Command line	Tests in execution set
`ctest -FS server -R clientNoDb`	clientNoDb, stopServer
`ctest -FC server -R clientNoDb`	clientNoDb, startServer
`ctest -FA server -R clientNoDb`	clientNoDb
`ctest -FS . -R client`	clientNoDb, clientWithDb, stopServer
`ctest -FA . -R client`	clientNoDb, clientWithDb

24.6. Кросс-компиляция и эмуляторы

Когда в качестве команды для add_test() используется цель исполняемого файла, определенная проектом, CMake автоматически подставляет местоположение собранного исполняемого файла. Для сценария кросс-компиляции это обычно не работает, поскольку хост обычно не может напрямую запускать двоичные файлы, созданные для другой платформы. Чтобы помочь в этом, CMake предоставляет свойство CROSSCOMPILING_EMULATOR target, которое может быть установлено в сценарий или исполняемый файл, используемый для запуска цели. Если это свойство установлено, CMake добавит его перед двоичным файлом цели и будет использовать его в качестве команды для запуска (т.е. реальный двоичный файл цели станет первым аргументом команды эмулятора, предоставляемой параметром
CROSSCOMPILING_EMULATOR). Это позволяет запускать тесты даже при кросс-компиляции.

CROSSCOMPILING_EMULATOR не обязательно должен быть реальным эмулятором, это просто должна быть команда, которую можно выполнить на хосте для запуска целевого исполняемого файла. Хотя эмулятор для целевой платформы является очевидным вариантом использования, можно также задать скрипт, который копирует исполняемый файл на целевую машину и запускает его удаленно (например, через SSH-соединение). Какой бы метод ни использовался, разработчики должны знать, что время запуска эмулятора или подготовки к запуску двоичного файла может быть нетривиальным и может повлиять на временные измерения теста. Это, в свою очередь, может означать необходимость пересмотра настроек тайм-аута теста.

Значение по умолчанию для свойства цели CROSSCOMPILING_EMULATOR берется из переменной CMAKE_CROSSCOMPILING_EMULATOR, что является обычным способом указания деталей эмулятора, а не установки свойства каждой цели отдельно. Переменная обычно устанавливается в файле цепочки инструментов, поскольку она влияет на такие вещи, как команды try_run(), подобно тому, как она влияет на тесты и пользовательские команды, как описано выше. Подробнее об этом аспекте влияния переменной см. обсуждение в разделе 21.5, "Проверки компилятора".

Даже если кросс-компиляция не выполняется, CMake будет учитывать непустое свойство цели CROSSCOMPILING_EMULATOR и добавлять его в командную строку для тестов и пользовательских команд, выполняющих эту цель. Это может быть весьма полезно, позволяя временно установить свойство в сценарий запуска для помощи в таких вещах, как отладка или сбор данных. Не рекомендуется использовать эту технику в качестве постоянной функции сборки проекта, но она может быть полезна в некоторых ситуациях разработки.

24.7. Режим сборки и тестирования

ctest можно использовать не только для выполнения набора тестов, но и для управления всем конвейером конфигурирования, сборки и тестирования. Для этого существует два основных метода: более простой, автономный способ и более мощный подход, тесно связанный с инструментом отчетности для приборной панели. Более простой подход заключается в вызове инструмента ctest с опцией командной строки --build-and-test, которая имеет свою собственную ожидаемую форму:

```
ctest --build-and-test sourceDir buildDir
      --build-generator generator
      [options...]
      [--test-command testCommand [args...]]
```

Без каких-либо опций вышеприведенное действие запустит CMake с указанными sourceDir и binaryDir и использует указанный генератор. Все три параметра должны быть указаны. Если запуск CMake прошел успешно, ctest соберет чистую цель, а затем соберет цель по умолчанию all. Чтобы после этапа сборки также запустить тесты, последней опцией в командной строке должна быть --test-command с соответствующей

командой testCommand и, по желанию, некоторыми аргументами. Это может быть еще один вызов ctest для запуска всех тестов.

```
ctest --build-and-test sourceDir buildDir
      --build-generator Ninja
      --test-command ctest -j 4
```

Вышеописанное выполняет полный конвейер configure-clean-build-test. Имеются различные опции, которые можно использовать для изменения того, какие части конвейера запускаются и как они запускаются. Например, --build-nocmake и --build-noclean отключают шаги configure и clean соответственно. Параметр --build
Опция -two-config будет вызывать CMake дважды, что позволяет справиться с некоторыми особыми случаями, когда для полной конфигурации проекта требуется второй проход CMake. При использовании генератора, например Visual Studio, может потребоваться указать дополнительные детали генератора с помощью опций --build-generator-platform и --build -generator-toolset, которые будут переданы в cmake на этапе configure в качестве опций -A и -T соответственно. Некоторые генераторы, например Xcode, могут потребовать указать имя проекта, чтобы он мог найти файл проекта, созданный на этапе configure, что можно сделать с помощью опции --build-project. Цель сборки на этапе сборки можно задать с помощью опции --build-target, а инструмент сборки можно переопределить, передав --build-makeprogram с альтернативным инструментом.

Как видно из вышеприведенного, все опции, относящиеся к режиму --build-and-test, начинаются с --build. Хотя большинство опций имеют интуитивно понятные названия, общий префикс --build может привести к некоторым досадным путаным аномалиям. Существует опция с именем --build-options, которая сначала может показаться связанной с этапом сборки, но на самом деле используется для передачи опций командной строки команде cmake. Она также имеет дополнительное ограничение: она должна быть последней в командной строке, если только не задана опция --test-command, в этом случае --build-options должна предшествовать --test-command. Следующий пример должен прояснить эти ограничения. Он добавляет два определения переменных кэша к вызову cmake, а также запускает полный набор тестов после этапа сборки.

```
ctest --build-and-test sourceDir buildDir
      --build-generator Ninja
      --build-options -DCMAKE_BUILD_TYPE=Debug -DBUILD_SHARED_LIBS=ON
      --test-command ctest -j 4
```

Существует еще несколько опций --build-..., но выше перечислены наиболее полезные из них. Еще одна оставшаяся опция, которую следует упомянуть, --test-timeout, которая устанавливает ограничение по времени (в секундах) на то, как долго будет выполняться команда test, прежде чем она будет принудительно завершена.

Управление всем конвейером с помощью одной команды ctest лучше или хуже, чем явное обращение к каждому из инструментов, необходимых для каждого этапа, зависит от ситуации. Последний пример, приведенный выше, может быть выполнен с помощью следующей эквивалентной последовательности команд на Unix:

```
mkdir -p buildDir
cd buildDir
cmake -G Ninja -DCMAKE_BUILD_TYPE=Debug -DBUILD_SHARED_LIBS=ON sourceDir
cmake --build . --target clean
cmake --build .
ctest -j 4
```

Вызов каждого инструмента по отдельности позволяет запускать их с полным набором опций, тогда как подход ctest --build-and-test имеет лишь очень ограниченную возможность контролировать этап сборки.

Одна из ситуаций, когда режим сборки и тестирования особенно удобен, - это когда проекту необходимо выполнить полный цикл configure-build-test в стороне, отдельно от основной сборки. Поскольку весь цикл может управляться одним вызовом ctest, его можно использовать в качестве части COMMAND в вызове add_test(), что делает процесс добавления базового проекта CMake в набор тестов основного проекта относительно простым. Сам CMake широко использует режим сборки и тестирования ctest в своем собственном наборе тестов именно таким образом.

Следующий пример показывает, как отдельная сборка может быть использована для тестирования API, предоставляемого библиотекой, собранной в основном проекте:

```
add_library(decoder foo.c bar.c)

add_test(NAME decoder.api
    COMMAND ${CMAKE_CTEST_COMMAND}
            --build-and-test  ${CMAKE_CURRENT_LIST_DIR}/test_api
                              ${CMAKE_CURRENT_BINARY_DIR}/test_api
            --build-generator ${CMAKE_GENERATOR}
            --build-options   -DDECODER_LIB=$<TARGET_FILE:decoder>
            --test-command    ${CMAKE_CTEST_COMMAND}
)
```

Исходный каталог test_api будет содержать свой собственный файл CMakeLists.txt, единственной целью которого является конфигурирование сборки, которая ссылается на библиотеку декодера, абсолютный путь к которой задается в переменной DECODER_LIB (это лишь один из нескольких способов передачи местоположения библиотеки в тестовый проект). Интересным моментом в такого рода тестах является то, что их также можно использовать для проверки того, что определенный тестовый проект *не* собирается, или для проверки того, что конфигурирование завершается с определенной фатальной ошибкой (например, отсутствующий символ). Такие ожидаемые фатальные ошибки сборки не могут быть проверены в основном проекте, поскольку это привело бы к сбою сборки основного проекта.

Другой сценарий, в котором такие тесты могут быть полезны, - это тестирование вывода генератора кода, созданного в основном проекте. Тестовые фикстуры можно использовать для установки пары тестов, один из которых генерирует код, а другой выполняет тестовую сборку с ним. Это особенно полезно, если генератор кода создает файлы, которые cmake обычно читает, например, файлы CMakeLists.txt. Например:

```
add_executable(codegen generator.cpp)

add_test(NAME generate_code COMMAND codegen)
add_test(NAME build_generated_code
    COMMAND ${CMAKE_CTEST_COMMAND}
            --build-and-test  ${CMAKE_CURRENT_LIST_DIR}/test_generation
                              ${CMAKE_CURRENT_BINARY_DIR}/test_generation
            --build-generator ${CMAKE_GENERATOR}
            --test-command    ${CMAKE_CTEST_COMMAND}
)

set_tests_properties(generate_code        PROPERTIES FIXTURES_SETUP    generator)
set_tests_properties(build_generated_code PROPERTIES FIXTURES_REQUIRED generator)
```

Режим сборки и тестирования можно также использовать для проверки скриптов утилит CMake, включив их в небольшой тестовый проект и вызывая его функциональность по мере необходимости. По сути, это обеспечивает довольно удобный способ реализации модульного тестирования сценариев CMake, позволяющий избежать необходимости помещать такие тесты в стадию configure основного проекта.

Хотя режим сборки и тестирования, безусловно, полезен для таких случаев, как упомянутые выше, ему не хватает гибкости полностью скриптового запуска, когда полный набор опций доступен для каждой отдельной команды. В следующем разделе представлен альтернативный способ вызова ctest, который предлагает более мощную обработку всего конвейера, включая некоторые полезные дополнительные возможности отчетности.

24.8. Интеграция CDash

CTest имеет долгую историю и тесную связь с другим продуктом под названием CDash, который также разработан той же компанией, что и CMake и CTest. CDash - это веб-панель, которая собирает результаты конвейера сборки и тестирования программного обеспечения, управляемого ctest. Она собирает предупреждения и ошибки с каждого этапа конвейера и показывает сводки по каждому этапу с возможностью перехода к каждому отдельному предупреждению или ошибке. История прошлых конвейеров позволяет наблюдать тенденции во времени и сравнивать запуски. Сам CMake имеет довольно обширную панель, которая отслеживает ночные сборки, сборки, связанные с запросами на слияние и так далее. Несколько минут, потраченных на изучение примера приборной панели, будут полезны для понимания материала, рассматриваемого в этом разделе:

https://open.cdash.org/index.php?project=CMake

24.8.1. Ключевые концепции CDash

Три важных понятия связывают воедино то, как CTest и CDash выполняют конвейеры и сообщают о результатах: *шаги* (иногда также называемые *действиями*), *модели* (также иногда называемые *режимами*) и *дорожки*. Шаги - это последовательность действий, которые выполняет конвейер. Основной набор определенных действий в том порядке, в котором они обычно вызываются, следующий:

- Start
- Update
- Configure
- Build
- Test

• Coverage
• MemCheck
• Submit

Не все действия должны быть выполнены, некоторые могут не поддерживаться или не требовать выполнения. Говоря грубо, каждая строка на приборной панели CDash соответствует одному конвейеру и обычно показывает сводку по каждому выполненному действию (хэш фиксации, общее количество предупреждений, ошибок, сбоев и т.д.).

Каждый конвейер должен быть связан с моделью, которая используется для определения определенного поведения, например, продолжать или нет последующие шаги после неудачи определенного шага. Модель также предоставляет набор действий по умолчанию, когда не запрашивается конкретное действие. Поддерживаются следующие модели:

Ночью

Предназначен для вызова один раз в день, обычно автоматическим заданием в то время, когда исполняющая машина менее загружена. Набор действий по умолчанию включает все перечисленные выше шаги, кроме *MemCheck*. Если шаг *Update* завершится неудачно, остальные шаги все равно будут выполнены.

Непрерывный

Очень похож на *Nightly*, за исключением того, что он предназначен для запуска несколько раз в день по мере необходимости, обычно в ответ на фиксацию изменений. Он определяет тот же набор действий по умолчанию, что и *Nightly*, но если шаг *Update* завершится неудачно, последующие шаги не будут выполнены.

Экспериментальный

Как следует из названия, эта модель предназначена для специальных экспериментов, проводимых разработчиками по мере необходимости. Ее набор действий по умолчанию включает все шаги, кроме *Update* и *MemCheck*. Если указана модель, отличная от одной из трех определенных моделей, или если модель вообще не указана, она будет рассматриваться как *экспериментальная*.

Трек контролирует, в какой группе будут показаны результаты трубопровода в результатах приборной панели. Имена дорожек могут быть любыми, какие пожелает использовать проект или разработчик, но если дорожка не указана, она будет установлена такой же, как и модель. Это привело к распространенному заблуждению, что модель управляет группировкой на приборной панели, но это делает именно трек. Действия *Coverage* и *MemCheck* - особый случай, они фактически игнорируют трек, и их результаты на приборной панели отображаются в собственных выделенных группах (*Coverage* и *Dynamic Analysis* соответственно).

24.8.2. Выполнение конвейеров и действий

Для проекта с необходимыми конфигурационными файлами (рассматривается в следующем разделе) можно вызывать целые конвейеры или отдельные шаги, используя следующую форму команды ctest:

```
ctest [-M Model] [-T Action] [--track Track] [otherOptions...]
```

Должны быть указаны хотя бы одна или обе *модели* и *действие*. Для удобства опции -M и -T можно объединить в одну опцию -D следующим образом:

```
ctest -D Model[Action] [--track Track] [otherOptions...]
```

Аргументы к -D могут опускать действие или добавлять его к *Модели*. Примеры допустимых аргументов: *Continuous*, *NightlyConfigure*, *ExperimentalBuild* и так далее. Опции -T и -D могут быть указаны несколько раз, чтобы перечислить несколько шагов в одном вызове ctest. Обратите внимание, что -D также используется для определения переменных ctest, и команда ctest будет рассматривать любую *модель* или *ModelAction, которую* она не распознает, как попытку установить переменную. Поэтому может быть безопаснее использовать опции -M и -T, а не -D.

Ночной запуск, использующий набор шагов по умолчанию и сообщающий о своих результатах в группе *Nightly* по умолчанию, тривиально вызывается как:

```
ctest -M Nightly
```

То же самое, но с результатами, представленными в другой группе под названием *Nightly Master,* будет сделано следующим образом:

```
ctest -M Nightly --track "Nightly Master"
```

Рассмотрим пользовательский *экспериментальный* конвейер, состоящий только из шагов *Configure*, *Build* и *Test* с результатами, сгруппированными в разделе *Simple Tests*. Это требует явного указания набора шагов, поскольку он отличается от набора действий по умолчанию, определенного для модели *Experimental* (не выполняется шаг *Coverage*). Это может быть сделано как последовательность вызовов ctest с одним шагом в каждом вызове, или все они могут быть перечислены вместе с использованием нескольких опций -T в одной командной строке. Для сравнения показаны обе формы:

```
# Separate commands
ctest -T Start -M Experimental --track "Simple Tests"
ctest -T Configure
ctest -T Build
ctest -T Test
ctest -T Submit

# One command
ctest -M Experimental --track "Simple Tests" \
      -T Start -T Configure -T Build -T Test -T Submit
```

Первым шагом должно быть действие *Start*, которое используется для инициализации деталей конвейера и записи имен модели и трека, которые будут использоваться в последующих шагах. Эти детали не нужно повторять ни для одного из последующих шагов, если разделить каждое действие на отдельный вызов ctest. Последним шагом будет действие *Submit*, предполагая, что целью является отправка окончательного набора результатов на приборную панель.

Все выходные данные собираются в подкаталоге Testing под каталогом, в котором вызывается ctest. Действие *Start* выписывает файл с именем TAG, который содержит как минимум две строки, первая из которых - дата-время начала выполнения в форме YYYYMMDD-hhmm, а вторая - имя трека. CMake 3.12 добавляет третью строку, содержащую имя модели. По мере выполнения каждого шага после действия *Start*, он будет создавать свой собственный выходной файл по адресу Testing/YYYYMMDD-hhmm/<Action>.xml и файл журнала по адресу

Testing/Temporary/Last<Action>_YYYYMMDD-hhmm.log (в случае шага *MemCheck* часть <Action> в именах этих файлов будет DynamicAnalysis, а не MemCheck). Действие *Submit* собирает выходные файлы XML и некоторые файлы журналов и отправляет их на назначенную приборную панель.

Чтобы прикрепить примечание о сборке ко всему конвейеру, используйте опцию -A или --add-notes с шагом *Submit,* чтобы указать имена файлов для загрузки, разделенные запятыми, если добавляется несколько файлов. Это может быть полезным способом записи дополнительных деталей о данном конкретном конвейере, например, информации от системы непрерывной интеграции, которая инициировала запуск.

```
ctest -T Submit --add-note JobNote.txt
```

Опция --extra-submit также поддерживается, но она предназначена скорее для внутреннего использования ctest. Она не является общим механизмом загрузки файлов и не должна использоваться разработчиками или проектами напрямую.

Хотя вышеописанная функциональность предназначена в первую очередь для интеграции с CDash, ее можно использовать и в других сценариях. Например, система Jenkins CI имеет плагин, который позволяет ей читать выходной файл Test.xml действия Test и записывать результаты тестирования аналогично CDash. Вместо того чтобы запускать ctest обычным способом, его можно вызвать в виде инструментальной панели, используя только действие *Test*. Затем плагину Jenkins нужно только указать, где найти файл Test.xml, и он сможет прочитать результаты тестирования. При таком использовании даже действие *Start* можно опустить, поскольку ctest молча выполнит эквивалент действия *Start* с *экспериментальной* моделью, если один из других шагов будет выполнен без предварительного действия *Start*. Проекты могут захотеть очистить предыдущее содержимое каталога Testing, чтобы гарантировать, что Jenkins получит только результаты текущего запуска.

При передаче выходного XML-файла действия инструменту, отличному от CDash, может возникнуть необходимость указать ctest не сжимать полученный вывод. По умолчанию вывод действия сжимается и записывается в XML-файл в кодировке ASCII, но это можно предотвратить, передав параметр
--no-compress-output опция для ctest. Используйте эту опцию только в случае необходимости, поскольку это приведет к увеличению размера выходных файлов.

Другая ситуация, когда действия приборной панели могут быть полезны без CDash, - это использование преимуществ поддержки покрытия кода или проверки памяти (Valgrind, Purify, различные санитайзеры и т.д.). Эти действия приборной панели могут облегчить вызов соответствующего инструмента и сбор результатов. Подробнее о том, как настроить и использовать эти инструменты, читайте в следующем разделе.

24.8.3. Конфигурация CTest

Подготовка проекта к интеграции CDash в основном осуществляется с помощью модуля CTest, предоставляемого CMake. Этот модуль должен быть включен в файл верхнего уровня CMakeLists.txt вскоре после команды project().

```
cmake_minimum_required(VERSION 3.0)
project(CDashExample)

# ... set any variables to customize CTest behavior

include(CTest)

# ... Define targets and tests as usual
```

Важно, чтобы модуль CTest был включен в файл CMakeLists.txt верхнего уровня, поскольку он записывает различные файлы в ассоциированный каталог сборки, а эти сгенерированные файлы, как правило, должны находиться в верхней части дерева сборки. Если проект впоследствии будет включен в родительский проект с помощью add_subdirectory(), родительский проект также должен поместить include(CTest) в свой CMakeLists.txt верхнего уровня, чтобы необходимые файлы генерировались в нужном месте.

Модуль CTest определяет кэш-переменную BUILD_TESTING, которая по умолчанию имеет значение true. Она используется для определения того, вызывает ли модуль enable_testing() или нет, так что проекту не нужно делать свой собственный вызов enable_testing(). Эта переменная кэша также может использоваться проектом для выполнения определенной обработки, только если тестирование включено. Если в проекте много тестов, сборка которых занимает много времени, это может быть полезным способом избежать их добавления в сборку, когда они не нужны.

```
cmake_minimum_required(VERSION 3.0)
project(CDashExample)
include(CTest)

# ... define regular targets

if(BUILD_TESTING)
    # ... define test targets and add tests
endif()
```

Модуль CTest определяет цели сборки для каждой *модели* и для каждой комбинации *ModelAction*. Эти цели выполняют ctest с опцией -D, установленной на имя цели, и предназначены для удобного выполнения всего конвейера или только одного действия приборной панели из приложения IDE. Цели не дают никаких реальных преимуществ по сравнению с прямым вызовом ctest при работе из командной строки.

Более важной задачей, выполняемой модулем CTest, является запись конфигурационного файла DartConfiguration.tcl в каталог сборки. Название этого файла является историческим, поскольку Dart - это первоначальное название проекта CDash. В этот файл записываются основные детали, такие как расположение исходного кода и каталога сборки, информация о машине, на которой выполняется сборка, используемый набор инструментов, расположение различных инструментов и другие настройки по умолчанию. В нем также будут содержаться сведения о сервере CDash, но для этого проект должен предоставить файл CTestConfig.cmake в верхней части дерева исходников с соответствующим содержимым. Подходящий файл CTestConfig.cmake можно получить из самого CDash (требуются права администратора), но обычно его несложно создать вручную. Минимальный пример выглядит следующим образом:

```
# Name used by CDash to refer to the project
set(CTEST_PROJECT_NAME "MyProject")

# Time to use for the start of each day. Used by
# CDash to group results by day, usually set to
# midnight in the local timezone of the CDash server.
set(CTEST_NIGHTLY_START_TIME "01:00:00 UTC")

# Details of the CDash server to submit to
set(CTEST_DROP_METHOD "http")
set(CTEST_DROP_SITE "my.cdash.org")
set(CTEST_DROP_LOCATION "/submit.php?project=${CTEST_PROJECT_NAME}")
set(CTEST_DROP_SITE_CDASH YES)

# Optional, but recommended so that command lines
# can be seen in the CDash logs
set(CTEST_USE_LAUNCHERS YES)
```

Файл DartConfiguration.tcl, записываемый модулем CTest, содержит определенные настраиваемые опции для каждого из действий приборной панели. Большинство из них уже установлены в соответствующие значения по умолчанию, но шаги *Coverage* и *MemCheck* имеют опции, которые могут представлять интерес для разработчиков. Они управляются переменными CMake, которые разработчик может проверить и изменить в кэше CMake или задать непосредственно в файле CMakeLists.txt до включения модуля CTest.

Предполагается, что шаг *Coverage* вызывает gcov, и модуль CTest будет искать команду с таким именем. Кэш-переменная COVERAGE_COMMAND хранит результат этого поиска, но при необходимости она может быть изменена разработчиком. Вторая кэш-переменная COVERAGE_EXTRA_FLAGS используется для хранения опций, которые должны следовать непосредственно за COVERAGE_COMMAND, так что у разработчика есть возможность контролировать как используемую команду, так и передаваемые ей опции.

Шаг *MemCheck* является более интересным. Поддерживается несколько различных средств проверки памяти, включая Valgrind, Purify, BoundsChecker и различные санитайзеры. Для первых трех их можно выбрать, установив MEMORYCHECK_COMMAND в расположение соответствующего исполняемого файла. ctest определит средство проверки по имени исполняемого файла. Для Valgrind переменная VALGRIND_COMMAND_OPTIONS также может быть установлена, чтобы переопределить опции, предоставляемые самому valgrind. Чтобы использовать один из дезинфекторов, установите MEMORYCHECK_TYPE в одну из следующих строк (MEMORYCHECK_COMMAND будет игнорироваться):

- AddressSanitizer
- LeakSanitizer
- MemorySanitizer
- ThreadSanitizer
- UndefinedBehaviorSanitizer

ctest будет запускать тестовые исполняемые файлы как обычно, но с соответствующими переменными окружения, установленными для включения требуемого санитайзера. Обратите внимание, что санитайзеры требуют сборки целей проекта с соответствующими флагами компилятора и компоновщика (обычно -fsanitize=XXX и, возможно, -fno-omit-frame-pointer). Для получения более подробной информации о

соответствующих флагах и о том, что делают различные санитайзеры, обратитесь к документации Clang или GCC.

Приведенных выше подробностей достаточно для выполнения различных действий с приборной панелью и отправки результатов на сервер CDash, но существует проблема курицы и яйца. Для получения файла DartConfiguration.tcl необходимо, чтобы шаги *Update* и *Configure* уже были выполнены. Поэтому детали этих двух шагов не могут быть получены, а в случае с шагом *Configure* результаты первого запуска cmake теряются, и их можно получить только при повторном запуске CMake в уже настроенном каталоге сборки. Тем не менее, все остальные шаги получат свои результаты, и этого может быть достаточно в некоторых ситуациях. Например, при использовании системы непрерывной интеграции, такой как Gitlab CI или Jenkins, начальное клонирование или обновление дерева исходных текстов может быть выполнено самой системой CI. Можно выполнить начальный запуск cmake, а затем запустить остальные шаги как действия панели управления. Окончательные результаты могут быть отправлены на сервер CDash или считаны непосредственно системой CI, а возможно, и то, и другое.

Чтобы получить полный конвейер, включая начальное клонирование или обновление существующего дерева исходных текстов и первый шаг конфигурирования, необходимо написать собственный сценарий ctest, чтобы установить все необходимые детали настройки и вызвать соответствующие функции ctest. Это может быть гораздо более сложным процессом и обычно не требуется, если уже используется другая система CI. Если этап клонирования/обновления не нужно захватывать, то сложность пользовательского сценария снижается. При таком использовании ctest вызывается с помощью опции -S или -SP (они одинаковы, за исключением того, что последняя создает новый процесс, а первая - нет). Ниже показан достаточно простой пример.

```
ctest -S MyCustomCTestJob.cmake
```

MyCustomCTestJob.cmake

```
# Re-use CDash server details we already have
include(${CTEST_SCRIPT_DIRECTORY}/CTestConfig.cmake)

# Basic information every run should set, values here are just examples
site_name(CTEST_SITE)
set(CTEST_BUILD_NAME          ${CMAKE_HOST_SYSTEM_NAME})
set(CTEST_SOURCE_DIRECTORY    "${CTEST_SCRIPT_DIRECTORY}")
set(CTEST_BINARY_DIRECTORY    "${CTEST_SCRIPT_DIRECTORY}/build")
set(CTEST_CMAKE_GENERATOR     Ninja)
set(CTEST_CONFIGURATION_TYPE RelWithDebInfo)

# Dashboard actions to execute, always clearing the build directory first
ctest_empty_binary_directory(${CTEST_BINARY_DIRECTORY})
ctest_start(Experimental)
ctest_configure()
ctest_build()
ctest_test()
ctest_submit()
```

Хотя приведенный выше пользовательский сценарий достаточно прост, следующий более интересный пример показывает, как пользовательские сценарии позволяют задать более гибкое поведение конвейера. Вместо того чтобы ждать до самого конца выполнения перед отправкой результатов на приборную панель, они отправляются постепенно после каждого шага (полезно, если некоторые шаги занимают много времени).

Исполняемые файлы создаются с поддержкой санитара адресов, и проверка санитара адресов выполняется вместо обычного тестирования. В конце также загружаются некоторые дополнительные файлы.

```
include(${CTEST_SCRIPT_DIRECTORY}/CTestConfig.cmake)

site_name(CTEST_SITE)
set(CTEST_BUILD_NAME          "${CMAKE_HOST_SYSTEM_NAME}-ASan")
set(CTEST_SOURCE_DIRECTORY    "${CTEST_SCRIPT_DIRECTORY}")
set(CTEST_BINARY_DIRECTORY    "${CTEST_SCRIPT_DIRECTORY}/build")
set(CTEST_CMAKE_GENERATOR     Ninja)
set(CTEST_CONFIGURATION_TYPE RelWithDebInfo)
set(CTEST_MEMORYCHECK_TYPE    AddressSanitizer)
set(configureOpts
    "-DCMAKE_CXX_FLAGS_INIT=-fsanitize=address -fno-omit-frame-pointer"
    "-DCMAKE_EXE_LINKER_FLAGS_INIT=-fsanitize=address -fno-omit-frame-pointer"
)
ctest_empty_binary_directory(${CTEST_BINARY_DIRECTORY})
ctest_start(Experimental TRACK Sanitizers)
ctest_configure(OPTIONS "${configureOpts}")
ctest_submit(PARTS Start Configure)
ctest_build()
ctest_submit(PARTS Build)
ctest_memcheck()
ctest_submit(PARTS MemCheck)
ctest_upload(FILES ${CTEST_BINARY_DIRECTORY}/mytest.log
                   ${CTEST_BINARY_DIRECTORY}/anotherFile.txt
)
ctest_submit(PARTS Upload Submit)
```

Каждая из различных команд ctest_... подробно описана в документации CMake, а также переменные CTest и CMake, которые можно использовать для настройки каждого шага или влияния на обработку различными способами. Приведенный выше сценарий должен быть хорошим базовым сценарием, который можно использовать для экспериментов с различными параметрами и переменными.

Создание сценария, который также обрабатывает клонирование/обновление проекта, усложняет задачу. Проекты часто имеют свои собственные особые способы делать это, и им обычно нужно решить, как должны быть запланированы такие вещи, как *ночные* и *непрерывные* сборки. Поддержка таких вещей, как автоматические сборки для запросов на слияние, будет сильно зависеть от возможностей хранилища, в котором находится проект. Для тех, кто заинтересован в изучении этого пути, рекомендуемый способ начать - найти проект, использующий аналогичную схему размещения репозитория, и использовать его в качестве руководства. Некоторые проекты включают пользовательский сценарий в свой репозиторий для удобства доступа (многие проекты Kitware делают это, и сценарии достаточно хорошо документированы).

24.8.4. Измерения и результаты испытаний

В приведенном выше примере было кратко показано, как загрузка файлов может быть включена в пользовательский сценарий CTest. Команда ctest_upload() предоставляет базовый механизм для записи файлов для загрузки и прикрепления к сборке в результатах CDash, при этом загрузка выполняется как часть последующего вызова ctest_submit(). Однако иногда загрузка файлов должна быть связана с конкретным тестом, а не со всем сценарием. Для этого в CMake предусмотрены свойства тестов ATTACHED_FILES и

ATTACHED_FILES_ON_FAIL. Оба содержат список файлов, которые должны быть загружены и связаны с конкретным тестом, с той лишь разницей, что последнее содержит файлы, которые будут загружены только в случае неудачи теста. Это очень полезный способ записи дополнительной информации о неудаче для дальнейшего расследования.

```
add_executable(doGen ...)
add_test(NAME generateFile COMMAND doGen)
set_tests_properties(generateFile PROPERTIES
    ATTACHED_FILES_ON_FAIL
        ${CMAKE_CURRENT_BINARY_DIR}/generated.c
        ${CMAKE_CURRENT_BINARY_DIR}/generated.h
)
```

Тесты также могут записывать одно значение измерения, которое будет записываться и отслеживаться в CDash при каждом представлении теста. Измерение обычно имеет форму key=value, хотя часть =value может быть опущена, чтобы использовать значение по умолчанию 1. Измерение записывается как свойство теста следующим образом:

```
set_tests_properties(perfRun PROPERTIES
    MEASUREMENT mySpeed=${someValue}
)
```

Поскольку значение измерения должно быть определено еще до запуска теста, это имеет ограниченную полезность. Гораздо более полезной является недокументированная возможность, активно используемая в таких проектах, как Vtk и построенных на его основе, когда измерения могут быть встроены в сам вывод теста в форме, похожей на HTML-теги. ctest сканирует вывод теста на наличие этих измерений, извлекает соответствующие данные и загружает их в CDash как часть результатов теста. Затем эти измерения отображаются в таблице результатов в верхней части страницы подробностей теста. Простейший тип измерения определяется следующей формой:

```
<DartMeasurement name="key" type="someType">value</DartMeasurement>
```

Атрибут name будет использоваться в качестве метки для измерения в таблице результатов, а атрибут type, как правило, будет что-то вроде text/string или numeric/double. Значение - это любое текстовое или числовое содержимое, которое имеет смысл для данного измерения. Для числовых значений CDash предоставляет возможность построить график истории каждого измерения по результатам последних тестов, что очень полезно для выявления изменений в поведении с течением времени.

Другая форма может быть использована для вставки файла, а не конкретного значения:

```
<DartMeasurement name="key" type="someType">value</DartMeasurement>
```

Эта вторая форма наиболее полезна для загрузки изображений, где атрибутом type будет что-то вроде image/png или image/jpeg. Значение filePath должно представлять собой полный абсолютный путь к загружаемому файлу.

CDash распознает несколько специальных названий измерений, когда речь идет об изображениях. Их можно использовать для сравнения ожидаемых и фактических изображений, причем CDash даже предоставляет

полезный интерактивный элемент пользовательского интерфейса для сравнения с наложением. Признанные атрибуты имен и их значения включают:

TestImage

Это интерпретируется как изображение, созданное тестом. Его можно рассматривать как выходной сигнал теста, который будет показан как сам по себе, так и как часть интерактивного изображения сравнения.

ValidImage

Это эквивалент ожидаемого изображения для теста. Оно должно иметь те же размеры, что и TestImage, но не обязательно должно иметь тот же формат изображения. Оно будет включено только в интерактивное изображение.

DifferenceImage2

Для создания изображения, представляющего разницу между двумя другими изображениями, можно использовать различные инструменты. Если тест предоставляет такой файл изображения, он может использовать это имя для включения его в выходные измерения теста, загруженные в CDash. Оно будет включено в интерактивное сравнительное изображение.

24.9. GoogleTest

CMake/ctest обеспечивают поддержку для создания, выполнения и определения статуса прохождения/непрохождения тестов. Проект отвечает за предоставление самого кода тестов, и именно здесь могут быть полезны такие фреймворки тестирования, как GoogleTest. Такие фреймворки дополняют возможности CMake и ctest, облегчая написание четких, хорошо структурированных тестовых примеров, которые хорошо интегрируются в работу CMake и ctest.

CMake уже давно поддерживает GoogleTest с помощью модуля FindGTest. Модуль ищет предварительно созданное местоположение GoogleTest и создает переменные, которые проекты могут использовать для включения GoogleTest в свою сборку. Начиная с CMake 3.5, также предоставляются цели импорта, которые являются более предпочтительными, чем использование переменных. Использование этих целей импорта приводит к гораздо более надежной обработке требований и свойств использования. Простой пример использования модуля в CMake 3.5 или более поздней версии будет выглядеть следующим образом:

```
add_executable(myGTestCases ...)

find_package(GTest REQUIRED)
target_link_libraries(myGTestCases PRIVATE GTest::GTest)

add_test(NAME myGTestCases COMMAND myGTestCases)
```

Цель импорта заботится о том, чтобы при сборке myGTestCases использовался соответствующий путь поиска заголовков и при необходимости подключалась соответствующая библиотека потоков. Вышеописанное работает на всех платформах, скрывая достаточно много сложностей, связанных с различными именами, временем выполнения, флагами и т.д., которые используются на различных платформах и компиляторах. Если использовать переменные, определенные модулем, вместо целей импорта, эти вещи в основном придется обрабатывать вручную, что является довольно хрупкой задачей.

Еще более надежным подходом является включение исходных текстов GoogleTest непосредственно в сборку, а не полагаться на наличие предварительно собранных двоичных файлов. Это гарантирует, что GoogleTest будет собран с теми же настройками компилятора и компоновщика, что и остальная часть проекта, что позволяет

избежать многих тонких проблем, которые могут возникнуть при использовании предварительно собранных двоичных файлов GoogleTest. Проекты могут делать это несколькими способами, каждый из которых имеет свои преимущества и недостатки. Встраивание копии исходных текстов и заголовков в проект - самый простой способ, но он отделяет проект от улучшений, которые могут быть сделаны в GoogleTest в будущем. Репозиторий git GoogleTest можно добавить в проект в качестве подмодуля git, но это тоже связано с проблемами надежности. Третий вариант загрузки исходных текстов GoogleTest в рамках шага конфигурирования подробно рассматривается в разделе 27.2, "FetchContent", и имеет мало недостатков (он также очень упрощен благодаря функциям, добавленным в CMake 3.11).

Тестовый исполняемый файл, использующий GoogleTest, обычно определяет более одного тестового случая. Обычная схема, когда исполняемый файл запускается один раз и считается, что это один тестовый пример, не совсем подходит. В идеале каждый тестовый пример GoogleTest должен быть виден в ctest, чтобы каждый из них можно было запустить и оценить по отдельности. Модуль FindGTest предоставляет функцию gtest_add_test(), которая сканирует исходный код в поисках использования соответствующих макросов GoogleTest и извлекает каждый отдельный тестовый пример в качестве собственного теста ctest. Форма этой команды традиционно была следующей:

```
gtest_add_tests(executable "extraArgs" sourceFiles..)
```

Начиная с CMake 3.1, список исходных файлов для сканирования может быть заменен ключевым словом AUTO, в этом случае список источников получается, если предположить, что исполняемый файл является целью CMake и использовать его целевое свойство SOURCES.

В CMake 3.9 было признано, что проекты могут захотеть использовать функцию gtest_add_tests() с GoogleTest, собранным самим проектом. Это означало, что проекту не нужен модуль Find, поэтому функция была перенесена в новый модуль GoogleTest, а FindGTest затем включил ее для поддержания обратной совместимости. В рамках этой работы была также добавлена улучшенная форма функции:

```
gtest_add_tests(
    TARGET target
    [SOURCES src1...]
    [EXTRA_ARGS arg1...]
    [WORKING_DIRECTORY dir]
    [TEST_PREFIX prefix]
    [TEST_SUFFIX suffix]
    [SKIP_DEPENDENCY]
    [TEST_LIST outVar]
)
```

Старая форма все еще поддерживается, но проекты должны предпочитать использовать новую форму, где это возможно, поскольку она более гибкая и надежная. Например, одна и та же цель может быть передана нескольким вызовам gtest_add_tests() с различными аргументами, при этом каждый вызов будет иметь различные TEST_PREFIX и/или TEST_SUFFIX для различения наборов генерируемых тестов. Новая форма также предоставляет набор тестов, найденных при указании опции TEST_LIST. Имея доступные имена тестов, проект может изменять свойства тестов по мере необходимости. Следующий пример демонстрирует эти различные возможности:

```
# Assume GoogleTest is already part of the build, so we don't need
# FindGTest and can reference the gtest target directly
include(GoogleTest)
add_executable(testDriver ...)
target_link_libraries(testDriver PRIVATE gtest)

# Run the testDriver twice with two different arguments
gtest_add_tests(
    TARGET      testDriver
    EXTRA_ARGS  --algo=fast
    TEST_SUFFIX .Fast
    TEST_LIST   fastTests
)
gtest_add_tests(
    TARGET      testDriver
    EXTRA_ARGS  --algo=accurate
    TEST_SUFFIX .Accurate
    TEST_LIST   accurateTests
)
set_tests_properties(${fastTests}     PROPERTIES TIMEOUT 3)
set_tests_properties(${accurateTests} PROPERTIES TIMEOUT 20)

set(betaTests ${fastTests} ${accurateTests})
list(FILTER betaTests INCLUDE REGEX Beta)
set_tests_properties(${betaTests} PROPERTIES LABELS Beta)
```

В приведенном выше примере создаются две группы тестов, а затем к ним применяются различные ограничения по времени ожидания. Имена тестов будут иметь разные суффиксы в каждой группе. Без опции TEST_SUFFIX второй вызов gtest_add_tests() будет неудачным, поскольку попытается создать тесты с тем же именем, что и при первом вызове. Пример также устанавливает метку Beta для некоторых тестов независимо от того, к какому набору тестов они принадлежат.

Хотя gtest_add_tests() хорошо работает для простых случаев и исходных файлов, не имеющих необычного форматирования, она не справляется с параметризованными тестами или тестами, определенными с помощью пользовательских макросов. Она также требует повторного запуска CMake для повторного сканирования исходных файлов при каждом изменении источников тестов. Если шаг CMake не является быстрым, работа над тестовым кодом может быть затруднена, поскольку после каждого изменения CMake будет вынужден повторно запускаться для следующей сборки. Опция SKIP_DEPENDENCY предотвращает такое поведение и полагается на то, что разработчик вручную повторно запускает CMake для обновления набора тестов, но это скорее временное обходное решение при активной работе над тестом, чем то, что следует оставлять на постоянной основе.

В CMake 3.10 была добавлена новая функция для устранения недостатков gtest_add_tests(), которая запрашивает список тестов у исполняемого файла при запуске ctest вместо сканирования исходного кода во время работы CMake. Благодаря этому CMake не нужно запускать заново при каждом изменении источника тестов, параметризованные тесты обрабатываются надежно, и нет никаких ограничений на формат или способ определения тестов. Единственный компромисс заключается в том, что список тестов недоступен во время выполнения CMake, поскольку он получается только после запуска ctest.

```
gtest_discover_tests(target
    [EXTRA_ARGS arg1...]
    [WORKING_DIRECTORY dir]
    [TEST_PREFIX prefix]
    [TEST_SUFFIX suffix]
    [NO_PRETTY_TYPES]
    [NO_PRETTY_VALUES]
    [PROPERTIES name1 value1...]
    [TEST_LIST var]
    [DISCOVERY_TIMEOUT seconds]  # See notes below
)
```

По умолчанию при генерации имен параметризованных тестов функция будет пытаться использовать имена типов или значений, а не числовой индекс. Обычно это приводит к гораздо более читабельным и полезным именам, но для тех случаев, когда это нежелательно, можно использовать опции NO_PRETTY_TYPES и NO_PRETTY_VALUES, чтобы подавить подстановку и просто использовать значения индексов.

Параметр DISCOVERY_TIMEOUT указывает на время, необходимое для запуска исполняемого файла для получения списка тестов. Значение по умолчанию 5 секунд должно быть достаточным для всех, кроме исполняемых файлов с огромным числом тестов или другим поведением, из-за которого возврат списка тестов занимает много времени. Эта конкретная опция была первоначально добавлена в CMake 3.10.1 с ключевым словом TIMEOUT, но было обнаружено, что она вызывает столкновение имен со свойством теста TIMEOUT, что приводило к неожиданному, но законному поведению. Ключевое слово было изменено на DISCOVERY_TIMEOUT в CMake 3.10.3 для предотвращения подобных сценариев.

Поскольку список тестов не возвращается вызывающей стороне, невозможно вызвать set_tests_properties() или set_property() для изменения свойств обнаруженных тестов. Вместо этого gtest_discover_tests() позволяет указать свойства и их значения в рамках вызова, которые затем записываются во входной файл ctest и применяются при запуске ctest. Несмотря на то, что эта функция не обеспечивает всей гибкости, как возможность итерационно просматривать набор обнаруженных тестов в CMake и обрабатывать их по отдельности, возможность задавать свойства обнаруженных тестов в целом - это обычно все, что нужно, и, как правило, не является существенным ограничением. Основным исключением является то, что невозможно установить свойства тестов, имена которых соответствуют ключевым словам в команде gtest_discover_tests(), или когда свойства требуют значений, представляющих собой списки. Для обработки таких случаев необходимо использовать пользовательский скрипт ctest, пример которого приведен ниже.

Опция TEST_LIST работает иначе для gtest_discover_tests(), чем для gtest_add_tests(). В этом случае имя переменной, заданное с помощью этой опции, используется во входном файле ctest, записываемом CMake, а не доступно CMake напрямую. Опция TEST_LIST нужна только в том случае, если проект добавляет свою собственную логику в сгенерированный входной файл ctest и хочет обратиться к списку сгенерированных тестов. Даже в этом случае, только если одна и та же цель используется в нескольких вызовах gtest_discover_tests(), это будет необходимо. Имя переменной по умолчанию <target>_TESTS используется, если оно не задано опцией TEST_LIST.

Пользовательский код может быть добавлен путем добавления имен файлов в список файлов, хранящихся в свойстве каталога TEST_INCLUDE_FILES. Проекты не должны перезаписывать это свойство каталога, они должны только добавлять его, так как gtest_discover_tests() использует это свойство для формирования набора файлов для чтения ctest. Следующий пример показывает, как использовать пользовательский файл для манипулирования свойствами обнаруженных тестов и реализовать ту же эквивалентную логику, что и в

предыдущем примере для gtest_add_tests(), включая обходной путь для углового случая несовпадения имен TIMEOUT:

```
gtest_discover_tests(
    testDriver
    EXTRA_ARGS  --algo=fast
    TEST_SUFFIX .Fast
    TEST_LIST   fastTests
)
gtest_discover_tests(
    testDriver
    EXTRA_ARGS  --algo=accurate
    TEST_SUFFIX .Accurate
    TEST_LIST   accurateTests
)
set_property(DIRECTORY APPEND PROPERTY
    TEST_INCLUDE_FILES ${CMAKE_CURRENT_LIST_DIR}/customTestManip.cmake
)
```

customTestManip.cmake

```
# Set here to work around the TIMEOUT keyword clash with the
# gtest_discover_tests() call, works with all CMake versions
set_tests_properties(${fastTests}     PROPERTIES TIMEOUT 3)
set_tests_properties(${accurateTests} PROPERTIES TIMEOUT 20)

set(betaTests ${fastTests} ${accurateTests})
list(FILTER betaTests INCLUDE REGEX Beta)
set_tests_properties(${betaTests} PROPERTIES LABELS Beta)
```

Использование пользовательского скрипта ctest немного усложняет проект, но позволяет полностью контролировать свойства тестов. Нет опасений по поводу столкновения имен с gtest_discover_tests(), и свойства со списочными значениями могут быть обработаны безопасно.

24.10. Рекомендуемые практики

Старайтесь, чтобы имя каждого теста было коротким, но достаточно специфичным для характера теста, чтобы можно было легко сузить набор тестов с помощью регулярных выражений с опциями -R и -E, предоставляемыми ctest. Избегайте включения test в название, поскольку это только добавляет дополнительное содержимое в вывод теста без какой-либо пользы.

Предположим, что проект однажды может быть включен в гораздо большую иерархию проектов, в которой может быть много других тестов. Может быть трудно сохранить имена тестов уникальными для всех проектов, но вместо того, чтобы включать в каждое имя теста строку, специфичную для проекта, рассмотрите возможность использования свойства теста LABELS для включения специфичной для проекта метки для каждого теста. Эти метки для каждого проекта позволят легко включать и исключать тесты с помощью регулярных выражений, задаваемых ctest с помощью опций -L и -LE. Тесты могут иметь несколько меток, поэтому это не накладывает ограничений на использование других меток, но может сделать работу с тестами более удобной.

Еще одно хорошее применение меток - определение тестов, выполнение которых, как ожидается, займет много времени. Разработчики и системы непрерывной интеграции могут захотеть запускать их реже, поэтому возможность исключить их на основе меток тестов может быть очень удобной. Рассмотрите возможность добавления метки для тестов, которые выполняются в течение нетривиального времени и которые не нужно выполнять так часто. В отсутствие каких-либо других существующих соглашений, метка LongRunning является хорошим выбором.

Помимо использования регулярных выражений для сопоставления имен и меток тестов, можно также сузить набор тестов до определенного каталога и ниже. Вместо того чтобы запускать ctest из верхней части дерева сборки, его можно запускать из подкаталогов, расположенных ниже. При этом ctest будут известны только те тесты, которые определены из связанного с этим каталогом каталога исходных текстов и ниже. Чтобы в полной мере воспользоваться этим преимуществом, не следует собирать все тесты в одном месте и определять их без структуры каталогов. Может быть полезно хранить тесты рядом с исходным кодом, который они тестируют, чтобы естественная структура каталогов исходного кода могла быть повторно использована для придания структуры тестам. Если исходный код когда-нибудь будет перемещен, такой подход также облегчит перемещение связанных с ним тестов.

Может возникнуть соблазн написать тесты, которые просто включают много протоколирования, а затем используют регулярные выражения для определения успеха. Такой подход может быть довольно хрупким, поскольку разработчики часто изменяют вывод логов, полагая, что они предназначены только для информационных целей. Добавление временных меток в регистрируемый вывод еще больше усложняет этот подход. Вместо того чтобы полагаться на соответствие вывода журнала, по возможности предпочитайте, чтобы тестовый код сам определял статус успеха или неудачи путем явного тестирования ожидаемых предварительных и последующих условий, промежуточных значений и т.д. Такие фреймворки для тестирования, как GoogleTest, значительно упрощают написание и поддержку таких тестов и настоятельно рекомендуются (какой именно фреймворк - менее важно, чем хотя бы использование *какого-то* подходящего фреймворка).

Если вы используете фреймворк GoogleTest, рассмотрите возможность использования функций gtest_add_tests() и gtest_discover_tests(), предоставляемых модулем GoogleTest. Если код теста достаточно прост, чтобы gtest_add_tests() могла найти все тесты, то это самый простой и гибкий способ манипулирования отдельными свойствами теста, но он может быть менее удобен при работе над самим кодом теста, поскольку может потребовать частого повторного запуска CMake. Если проект может потребовать CMake 3.10.3 или более поздней версии в качестве минимальной, то gtest_discover_tests() может быть более подходящей. Основной недостаток этой функции в том, что установка тестовых свойств в значения, представляющие собой списки, требует больше работы, что особенно актуально, если следовать приведенному выше совету относительно использования тестовых меток. Если требуется поддержка версий CMake до 3.9, можно использовать только gtest_add_tests() и только более простую форму команды. Проект также должен использовать модуль FindGTest, а не модуль GoogleTest, что добавляет дополнительные сложности, если GoogleTest собирается как часть самого проекта. Поэтому проектам настоятельно рекомендуется перейти на CMake 3.9 или более позднюю версию, если используется GoogleTest, а в идеале - 3.10.3 или более позднюю.

Для проектов, в которых возможна кросс-компиляция для другой целевой платформы, подумайте, можно ли написать тесты для запуска под эмулятором или для копирования и выполнения на удаленной системе с помощью скрипта. Для реализации любой из этих стратегий можно использовать переменную CMAKE_CROSSCOMPILING_EMULATOR и связанное с ней свойство CROSSCOMPILING_EMULATOR target. В идеале, CMAKE_CROSSCOMPILING_EMULATOR должна быть установлена в файле цепочки инструментов, используемой для кросс-компиляции.

Максимально используйте поддержку параллельного выполнения тестов в ctest. Если известно, что тесты используют более одного процессора, установите свойство PROCESSORS для этих тестов, чтобы обеспечить лучшее руководство для ctest по их планированию. Если тестам требуется эксклюзивный доступ к общему ресурсу, используйте свойство RESOURCE_LOCK для контроля доступа к этому ресурсу и избегайте использования свойства теста RUN_SERIAL, если нет другой альтернативы. RUN_SERIAL может оказать большое негативное влияние на производительность параллельных тестов и редко оправдано, кроме быстрых временных экспериментов разработчиков. Если машина, на которой выполняется ctest, может иметь другие процессы, способствующие загрузке процессора, рассмотрите возможность использования опции -l, чтобы помочь ограничить чрезмерную нагрузку на процессор. Это может быть особенно полезно на машинах разработчиков, где разработчики могут одновременно создавать и запускать тесты для нескольких проектов.

Если минимальная версия CMake может быть установлена на 3.7 или более позднюю, предпочитайте использовать тестовые фикстуры для определения зависимостей между тестами. Определите тестовые случаи для установки и очистки ресурсов, требуемых другими тестами, для запуска и остановки служб и т.д. При запуске с сокращенным набором тестов в результате подбора регулярных выражений или опций типа --rerun-failed, ctest автоматически добавляет в набор тестов необходимые фикстуры. Фикстуры также гарантируют, что тесты, чьи зависимости не работают, будут пропущены, в отличие от свойства теста DEPENDS, которое просто контролирует порядок тестов, не обеспечивая требования успеха. Чтобы получить более тонкий контроль над тем, какие тесты будут автоматически добавлены в набор тестов для удовлетворения зависимостей от приспособлений, используйте CMake 3.9 или более позднюю версию для опций ctest -FS, -FC и -FA, добавленных в этом выпуске. Проекты могут по-прежнему требовать только CMake 3.7 в качестве минимальной версии. Также предпочтительнее использовать фикстуры, чем свойство теста TIMEOUT_AFTER_MATCH из-за более четкой зависимости зависимостей и контроля времени.

Режим сборки и тестирования ctest может быть полезным способом включения небольших тестовых сборок в качестве тестовых примеров в набор тестов основного проекта. Это может быть особенно эффективно, когда некоторые из этих тестовых сборок должны проверить, что определенные ситуации приводят к ошибкам конфигурации или сборки. Поскольку тестовые случаи могут быть определены как ожидаемые для отказа, они могут проверять такие условия, не приводя к отказу сборки основного проекта. Рассмотрим использование режима сборки и тестирования ctest в качестве части COMMAND в вызове add_test() для определения таких тестовых случаев.

Для запуска полного конвейера конфигурирования, сборки и тестирования основного проекта рассмотрите функциональность, предлагаемую функциями интеграции CDash, вместо использования режима сборки и тестирования ctest. Они лучше справляются с захватом результатов всего конвейера и предоставляют механизмы для настройки поведения каждого этапа. Он также имеет дополнительные возможности, облегчающие использование инструментов покрытия кода и динамического анализа, таких как средства проверки памяти, санитайзеры и т.д., и эти возможности могут использоваться независимо от того, передаются результаты на сервер CDash или нет. Фактически, пользовательская функциональность сценариев ctest, которая управляет всем конвейером CDash, может использоваться без CDash, что делает ее потенциально удобным независимым от платформы способом сценариев всего конвейера сборки и тестирования и для других систем непрерывной интеграции. Сервер CDash также можно использовать совместно с другими системами CI для обеспечения более богатого набора функций для записи и сравнения истории сборок, тенденций отказов тестирования и так далее.

Глава 25. Установка

После всей тяжелой работы по разработке исходного кода проекта, созданию его различных ресурсов, обеспечению надежности сборки и внедрению автоматизированных тестов, решающим является последний шаг - предоставление программного обеспечения для распространения. Он напрямую влияет на первое впечатление конечного пользователя о проекте и, если он выполнен некачественно, может привести к тому, что программное обеспечение будет отвергнуто еще до того, как оно получит шанс на использование.

Разработчики и пользователи могут иметь разные ожидания относительно того, как проект должен быть доступен. Для некоторых достаточно просто предоставить доступ к репозиторию исходного кода и ожидать, что конечные пользователи сами проверят и соберут его. Хотя это может быть частью модели предоставления, не все конечные пользователи захотят участвовать в проекте на таком низком уровне. Вместо этого они часто ожидают получить готовый бинарный пакет, который они могут установить и использовать на своей машине, предпочтительно с помощью какой-либо уже знакомой системы управления пакетами. Учитывая разнообразие менеджеров пакетов и форматов доставки, это может представлять собой сложную задачу для сопровождающих проектов. Тем не менее, между большинством из них есть достаточно общих элементов, поэтому при разумном планировании можно поддерживать большинство популярных из них и охватить все основные платформы.

Чем раньше в жизненном цикле проекта будет рассмотрена фаза доставки, тем более гладко пройдут этапы окончательной упаковки и развертывания. Хорошая отправная точка - задать следующие вопросы до начала разработки или как можно раньше для существующих проектов:

- Какие платформы должны поддерживаться, как на начальном этапе, так и потенциально в будущем? Существуют ли минимальные требования к API платформы или версии SDK для поддержки функций проекта?
- Каковы форматы пакетов, с которыми пользователи будут знакомы на каждой платформе? Может ли проект быть предоставлен в этих форматах? Есть ли какие-то конкретные форматы пакетов, которые важнее других или являются обязательными?
- Имеются ли в любом из требуемых или желательных форматов пакетов требования к тому, как должно быть изложено, построено или доставлено программное обеспечение? Должны ли ресурсы проекта предоставляться в определенных форматах, разрешениях, местах и т.д.?
- Может ли конечный пользователь захотеть установить несколько версий программного обеспечения одновременно?
- Должно ли программное обеспечение поддерживать установку без административных привилегий?
- Можно ли сделать программное обеспечение перемещаемым, чтобы пользователи могли установить его в любом месте своей системы (включая любой диск, в случае Windows)?
- Ожидает ли проект, что один или несколько его исполняемых файлов будут доступны на машине развертывания через переменную окружения PATH пользователя? Есть ли части проекта, которые не должны быть доступны в PATH?
- Предоставляет ли проект что-либо, что другие проекты CMake могут захотеть использовать в своих собственных сборках (библиотеки, исполняемые файлы, заголовки, ресурсы и т.д.)?

Эти вопросы будут сильно влиять на то, как программное обеспечение будет размещено при установке, что, в свою очередь, повлияет на то, как исходный код должен получить доступ к своим ресурсам и так далее. Это

может даже повлиять на функциональность программного обеспечения, поэтому понимание этих вопросов на ранней стадии может сэкономить напрасные усилия в дальнейшем.

Эта глава посвящена аспектам компоновки и тому, как собрать необходимые файлы в нужных местах. Она также демонстрирует, как сделать проект удобным для использования другими проектами CMake, обеспечив поддержку пакета config. Разработчики с некоторым опытом могут отнести эти аспекты к сфере make install. Следующая глава дополняет картину, обсуждая различные форматы пакетов, которые могут создавать CMake и CPack. Реализация этой поддержки использует функциональность install, описанную здесь, для установки в чистую область хранения и последующего создания окончательных пакетов из этого содержимого.

25.1. Макет каталога

Понимание ограничений, накладываемых платформой (платформами) развертывания, является важным шагом перед принятием решения о том, как должен быть расположен установленный продукт. Только после выяснения этих деталей проект CMake может приступить к определению того, что и куда устанавливать. Можно сделать несколько наблюдений высокого уровня, которые потенциально могут оказать сильное влияние на компоновку установленного проекта.

- Форматы Apple (пакеты, фреймворки и т.д.) жестко предписаны и не обеспечивают особой гибкости, но это также позволяет четко определить, как проект должен структурировать свои результаты. Как уже говорилось в главе 22, *Особенности Apple*, CMake уже обрабатывает большую часть этого автоматически на этапе сборки, делая приложение готовым к последней части процесса, управляемого Xcode, который выполняет окончательное подписание приложения, создание пакета и отправку в магазин приложений. Если этап установки вообще будет использоваться в CMake/CPack, то в основном для того, чтобы просто упаковать пакеты, которые следуют предписанной схеме.
- Для проектов, намеревающихся поддержать включение в дистрибутив Linux, почти наверняка существуют очень конкретные рекомендации по тому, куда следует устанавливать файлы каждого типа. Стандарт иерархии файловой системы является основой для компоновки большинства дистрибутивов, и многие другие системы на базе Unix придерживаются аналогичной структуры. Даже если FHS не предназначен непосредственно для включения в дистрибутив, он все равно служит хорошим руководством по структурированию пакета для достижения гладкой и надежной установки на многих системах на базе Unix.
- Некоторые проекты могут захотеть сделать один или несколько исполняемых файлов доступными в PATH пользователя, чтобы их можно было легко вызвать из терминала или командной строки. В Windows, если установка проекта изменяет PATH, добавляя каталог, который также содержит некоторые собственные DLL, другие приложения могут подхватить эти DLL вместо ожидаемых (например, из своих личных каталогов или одного из стандартных общесистемных мест). DLL из популярных наборов инструментов, таких как Qt, регулярно становятся жертвами этого сценария в результате того, что пакеты изменяют PATH так, как не должны. Если проект хочет расширить PATH для своих собственных исполняемых файлов, он должен убедиться, что в этом каталоге нет DLL, но это напрямую противоречит необходимости размещения DLL в том же каталоге, что и исполняемые файлы, чтобы Windows могла найти их во время выполнения. Типичным решением этой проблемы является создание каталога, содержащего только сценарии запуска, которые затем можно безопасно добавить в PATH.

25.1.1. Относительная компоновка

За исключением установки на платформы Apple, существует большая степень общности (или, по крайней мере, потенциальной общности) для всех основных платформ. Место установки можно представить как состоящее из базового пути и относительного расположения под этим путем. Базовый путь может быть чем-то вроде /usr/..., /opt/... или C:\Program Files и, очевидно, сильно варьируется между платформами, но относительное расположение ниже этой базовой точки часто очень похоже. Обычно исполняемые файлы (а для Windows еще и DLL) устанавливаются в каталог bin, библиотеки - в lib или какой-либо другой, а заголовки - в каталог include. Другие типы файлов имеют несколько большую вариативность в том, куда они обычно устанавливаются, но эти три типа уже охватывают некоторые из наиболее важных типов файлов, устанавливаемых в проекте.

В Windows другой вариант - пакеты размещают исполняемые файлы и библиотеки DLL в базовом каталоге установки, а не в подкаталоге bin. Хотя это относительно распространенная практика, она может привести к довольно переполненному базовому каталогу, что затрудняет пользователям поиск других компонентов пакета. Другой вариант - размещение сценариев запуска в подкаталоге с именем cmd, что позволяет отделить их от DLL в других местах, таких как bin.

Желательно найти структуру каталогов, которая работает на большинстве платформ, поскольку это минимизирует специфическую для платформы логику, которая должна быть реализована в исходном коде проекта. Если проект использует одну и ту же относительную структуру на всех платформах, приложению легче найти нужные ему вещи во время выполнения. При отсутствии каких-либо других требований модуль GNUInstallDirs в CMake обеспечивает очень удобный способ использования стандартной компоновки каталогов. Он соответствует общим случаям, упомянутым выше, а также предоставляет различные другие стандартные расположения, которые соответствуют как стандартам кодирования GNU, так и FHS. Если отбросить части, относящиеся к пути базовой установки (о них речь пойдет в следующем разделе), эту схему можно использовать даже для установки Windows.

Использование модуля GNUInstallDirs довольно простое, он включается как любой другой модуль:

```
# Minimal inclusion, but see caveat further below
include(GNUInstallDirs)
```

Это создаст переменные кэша вида CMAKE_INSTALL_<dir>, где <dir> обозначает определенное местоположение. В документации модуля дается полная информация обо всех определенных местах, но некоторые из наиболее часто используемых и их назначение включают:

BINDIR

Исполняемые файлы, сценарии и симлинки, предназначенные для непосредственного запуска конечными пользователями. По умолчанию используется bin.

SBINDIR

Аналогичен BINDIR, но предназначен для использования системным администратором. По умолчанию используется sbin.

LIBDIR

Библиотеки и объектные файлы. По умолчанию lib или некоторые другие варианты в зависимости от платформы хоста/целевой платформы (включая, возможно, дополнительный подкаталог, специфичный для архитектуры).

LIBEXECDIR

Исполняемые файлы, которые не вызываются непосредственно пользователями, но могут быть запущены через сценарии запуска или симлинки, расположенные в BINDIR, или другими способами. По умолчанию используется libexec

INCLUDEDIR

Заголовочные файлы. По умолчанию включается.

DATAROOTDIR

Корневая точка независимых от архитектуры данных, доступных только для чтения. Обычно на него не ссылаются напрямую, за исключением, возможно, случаев, когда нужно обойти предостережения для DOCDIR.

DATADIR

Независимые от архитектуры данные, такие как изображения и другие ресурсы, доступные только для чтения. По умолчанию совпадает с DATAROOTDIR и является предпочтительным способом ссылки на местоположение произвольных данных проекта, не охваченных другими определенными местоположениями.

MANDIR

Документация в формате man. По умолчанию DATAROOTDIR/man

DOCDIR

Общая документация. По умолчанию DATAROOTDIR/doc/PROJECT_NAME (см. примечания ниже о том, почему полагаться на это значение по умолчанию относительно небезопасно).

Поскольку каждое местоположение определяется как переменная кэша, их можно переопределить при необходимости. Разработчики обычно не изменяют их, поскольку места установки должны находиться под контролем проекта. Однако даже для проекта изменение местоположений по умолчанию обычно не рекомендуется, но это может быть полезно, если проект хочет в основном следовать стандартной схеме и нуждается лишь в нескольких небольших изменениях.

Расположение DOCDIR заслуживает особого упоминания, так как по умолчанию оно принимает значение, которое включает в себя

Переменная PROJECT_NAME. PROJECT_NAME обновляется при каждом вызове project() и поэтому может меняться в иерархии проектов. Модуль GNUInstallDirs устанавливает переменные кэша, только если они еще не определены, поэтому значение CMAKE_INSTALL_DOCDIR будет определяться тем, где впервые включен модуль GNUInstallDirs. Чтобы защититься от этого и позволить директории документации по умолчанию следовать иерархии проекта, проекты могут захотеть явно задавать расположение DOCDIR каждый раз, когда модуль включается (переменная без кэша будет переопределять переменную с кэшем):

```
# Explicitly set DOCDIR location each time
include(GNUInstallDirs)
set(CMAKE_INSTALL_DOCDIR ${CMAKE_INSTALL_DATAROOTDIR}/doc/${PROJECT_NAME})
```

В оставшейся части этой главы в примерах будут использоваться переменные CMAKE_INSTALL_<dir> для большинства относительных направлений установки.

25.1.2. Расположение базовой установки

После определения относительного расположения установленных файлов необходимо определить место установки базовой версии. На это решение влияет целый ряд соображений, но, пожалуй, первый вопрос, на который нужно ответить, - должна ли установка быть перемещаемой. Это означает, что можно использовать любую базовую точку установки, и при сохранении относительной компоновки установленный проект будет работать так, как задумано. Быть перемещаемым очень желательно и должно быть целью большинства проектов, поскольку это открывает больше возможностей для использования, например:

- Одновременно можно установить несколько версий.
- Перемещаемые пакеты могут быть установлены на общие диски, которые могут иметь разные точки монтирования на машинах разных конечных пользователей.
- Набор самодостаточных перемещаемых файлов может быть легче упакован более широким спектром упаковочных систем.
- Пользователи, не являющиеся администраторами, могут установить перемещаемый проект локально под своей учетной записью.

Не все проекты можно сделать перемещаемыми, некоторые из них должны размещать свои файлы в очень специфических местах (например, пакеты ядра). Некоторые проекты могут быть перемещаемыми за исключением нескольких конфигурационных файлов, и в этом случае полезной стратегией может быть обработка этих конкретных файлов в качестве скриптового шага после установки (в следующей главе обсуждаются некоторые аспекты этого для конкретных систем упаковки).

Выбор места базовой установки тесно связан с целевой платформой, и для каждой из них существуют свои общие правила и рекомендации. В Windows местом базовой установки обычно является подкаталог C:\Program Files, в то время как в большинстве других систем это /usr/local или подкаталог /opt. CMake предоставляет ряд элементов управления расположением базовой установки, чтобы в основном абстрагироваться от этих различий между платформами. Возможно, наиболее важной является переменная CMAKE_INSTALL_PREFIX, которая управляет местом базовой установки, когда пользователь собирает цель установки (цель может называться INSTALL в некоторых типах генераторов). По умолчанию CMAKE_INSTALL_PREFIX имеет значение C:\Program Files\${PROJECT_NAME} в Windows и /usr/local на платформах Unix.

При установке в Linux значение по умолчанию не соответствует стандарту File System Hierarchy. FHS требует, чтобы системные пакеты использовали базовое расположение / или /usr, причем последний вариант является более предпочтительным. Для дополнительных пакетов они должны быть установлены в /opt/<package> или /opt/<provider>, при этом рекомендуется использовать /opt/<provider>/<package>. Если используется <provider>, формально ожидается, что это будет имя, зарегистрированное в LANANA, или просто полное доменное имя организации, предоставляющей пакет, в нижнем регистре. Это делается для того, чтобы избежать конфликтов между различными пакетами, пытающимися использовать одно и то же место базовой установки. Для большинства проектов рекомендуется явно устанавливать CMAKE_INSTALL_PREFIX для не-Windows платформ на FHS-совместимый путь /opt/..., но это обычно делается только в CMakeLists.txt верхнего уровня с соответствующей проверкой, что проект действительно является верхним уровнем дерева исходников (для поддержки иерархической организации проектов).

```
if(NOT WIN32 AND CMAKE_SOURCE_DIR STREQUAL CMAKE_CURRENT_SOURCE_DIR)
    set(CMAKE_INSTALL_PREFIX "/opt/mycompany.com/${PROJECT_NAME}")
endif()
```

Для сценариев кросс-компиляции переменная CMAKE_STAGING_PREFIX может быть определена, чтобы обеспечить переопределение места установки правила установки. Это позволяет устанавливать в альтернативную часть файловой системы, сохраняя при этом все другие эффекты CMAKE_INSTALL_PREFIX, такие как встраивание путей в устанавливаемые двоичные файлы (рассматривается в разделе 25.2.2, "RPATH", далее в этой главе). CMAKE_STAGING_PREFIX также влияет на пути поиска большинства команд find_...().

Для некоторых сценариев упаковки и для возможности тестирования процесса установки в месте, расположенном в стороне,

CMake поддерживает общую функциональность DESTDIR для платформ, отличных от Windows. DESTDIR - это не переменная CMake, а переменная, передаваемая инструменту сборки или задаваемая как переменная окружения, которую инструмент сборки должен прочитать. Она позволяет поместить местоположение базы установки в произвольное место, а не в корень файловой системы. Обычно она используется в командной строке при прямом вызове инструмента сборки, например:

```
make DESTDIR=/home/me/staging install
env DESTDIR=/home/me/staging ninja install
```

Функциональность DESTDIR концептуально похожа на CMAKE_STAGING_PREFIX, но DESTDIR указывается только во время установки и не влияет на такие вещи, как команды find_...().

CMAKE_STAGING_PREFIX сохраняется как переменная кэша, тогда как DESTDIR является переменной окружения и не сохраняется между вызовами инструмента сборки.

Сочетание CMAKE_INSTALL_PREFIX, CMAKE_STAGING_PREFIX и DESTDIR дает проекту и разработчику гибкость в настройке базового места установки по мере необходимости и позволяет выполнять тестовые установки, не затрагивая окончательное место установки. Однако следует помнить, что различные форматы упаковки могут иметь собственные базовые места установки по умолчанию и полностью игнорировать эти три переменные, отдавая предпочтение своим собственным, специфичным для конкретного пакета.

25.2. Установка мишеней

Когда структура области установки определена, можно перейти к самому устанавливаемому содержимому. Проекты используют команду install() для определения того, что устанавливать, где это должно быть расположено и так далее. Эта команда имеет несколько различных форм, каждая из которых ориентирована на определенный тип сущностей, который задается первым аргументом команды. Одна из ключевых форм - установка целей:

```
install(TARGETS targets...
        [EXPORT exportName]
        [CONFIGURATIONS configs...]
        # One or more blocks of the following
        [ [entityType]
                DESTINATION dir
                [PERMISSIONS permissions...]
                [NAMELINK_ONLY | NAMELINK_SKIP]
                [COMPONENT component]
                [NAMELINK_COMPONENT component]   # CMake 3.12 or later only
                [EXCLUDE_FROM_ALL]
                [OPTIONAL]
                [CONFIGURATIONS configs...]
        ]...
        # Special case
        [INCLUDES DESTINATION incDirs...]
)
```

Предоставляется одна или несколько целей, а затем в блоках entityType указывается, как обрабатывать установку различных частей этих целей. Каждая из целей должна быть определена в той же области каталогов, что и команда install(), а тип entityType должен быть одним из следующих:

RUNTIME

Установите исполняемые двоичные файлы. В Windows также устанавливается DLL-часть библиотечных целей. Пакеты Apple исключены.

LIBRARY

Установите разделяемые библиотеки на всех платформах, кроме Windows. Фреймворки Apple исключены.

ARCHIVE

Установите статические библиотеки (все платформы). В Windows также устанавливается часть библиотеки импорта (т.е. .lib) разделяемых библиотек. Фреймворки Apple исключены.

OBJECTS

Установите объекты, связанные с библиотеками объектов (только CMake 3.9 или более поздняя версия).

FRAMEWORK

На платформах Apple установите фреймворки (общие или статические), включая любое содержимое, которое было скопировано в них (например, с помощью пользовательских правил POST_BUILD).

BUNDLE

На платформах Apple устанавливайте пакеты, включая любое содержимое, которое было в них скопировано.

PUBLIC_HEADER

На платформах, отличных от Apple, это устанавливает файлы, перечисленные в свойстве PUBLIC_HEADER цели библиотеки фреймворка. На платформах Apple эти заголовочные файлы обрабатываются как часть типа сущности FRAMEWORK, но на платформах не Apple такие цели рассматриваются как обычные разделяемые библиотеки, и заголовки должны быть явно установлены как отдельный тип сущности.

PRIVATE_HEADER

Аналогичен типу сущности PUBLIC_HEADER, за исключением того, что затрагиваемое целевое свойство - PRIVATE_HEADER.

RESOURCE

На платформах, отличных от Apple, при этом устанавливаются файлы, перечисленные в свойстве RESOURCE цели фреймворка или пакета. На платформах Apple такие файлы включаются как часть типа сущности FRAMEWORK или BUNDLE.

После entityType могут быть перечислены различные опции, которые применяются только к этому типу сущности. Например, ниже показано, как установить библиотеки таким образом, чтобы поместить соответствующие части в ожидаемое место на всех платформах (предполагается, что это не фреймворки Apple):

```
install(TARGETS mySharedLib myStaticLib
    RUNTIME DESTINATION ${CMAKE_INSTALL_BINDIR}
    LIBRARY DESTINATION ${CMAKE_INSTALL_LIBDIR}
    ARCHIVE DESTINATION ${CMAKE_INSTALL_LIBDIR}
)
```

Приведенный выше пример показывает, как опция DESTINATION может указывать разные местоположения для разных частей одной и той же цели. Команда также достаточно гибкая для одновременной работы с несколькими целями разных типов.

- Для mySharedLib в Windows DLL будет установлена в место назначения RUNTIME, а библиотека импорта - в место назначения ARCHIVE. На других платформах общая библиотека будет установлена в место назначения LIBRARY.
- Статическая библиотека цели myStaticLib будет установлена в место назначения ARCHIVE.

CMake обычно выдает предупреждение или ошибку, если цель предоставляет определенную сущность, для которой нет соответствующего раздела entityType (например, одна из целей - статическая библиотека, но нет раздела ARCHIVE). Как исключение, entityType может быть опущен, в этом случае опции, следующие за списком целей, будут применяться ко всем типам сущностей. Обычно это делается только в тех случаях, когда очевидно, что для перечисленных целей может быть только один тип сущности:

```
# Targets are both executables, so specifying the entity type isn't needed
install(TARGETS exe1 exe2
        DESTINATION ${CMAKE_INSTALL_BINDIR}
)
```

Опции, следующие за типом сущности, могут указывать не только место назначения. Они также могут переопределять разрешения по умолчанию с помощью опции PERMISSIONS, указывая одно или несколько тех же значений, что и для команды file(COPY), описанной в разделе 18.2, "Копирование файлов":

OWNER_READ	OWNER_WRITE	OWNER_EXECUTE
GROUP_READ	GROUP_WRITE	GROUP_EXECUTE
WORLD_READ	WORLD_WRITE	WORLD_EXECUTE
SETUID	SETGID	

Как и в случае с file(COPY), разрешения, не поддерживаемые для данной платформы, будут просто проигнорированы. Обратите внимание, что CMake обычно устанавливает соответствующие разрешения для всех целей по умолчанию, поэтому обычно требуется явно указать разрешения только в том случае, если установленное место требует более строгих разрешений, чем обычно, или если необходимо добавить одно из разрешений SETUID или SETGID. Например:

```
# Intended to only be run by an administrator, so only allow the owner to have access
install(TARGETS onlyOwnerCanRunMe
        DESTINATION ${CMAKE_INSTALL_SBINDIR}
        PERMISSIONS OWNER_READ OWNER_WRITE OWNER_EXECUTE
)

# Install with set-group permission
install(TARGETS runAsGroup
        DESTINATION ${CMAKE_INSTALL_BINDIR}
        PERMISSIONS OWNER_READ OWNER_WRITE OWNER_EXECUTE
                    GROUP_READ GROUP_EXECUTE SETGID
)
```

Для типа сущности LIBRARY некоторые платформы поддерживают создание символических ссылок, если предоставлена информация о версии целевой библиотеки (см. раздел 20.3, "Версионирование общих библиотек"). Набор файлов и симлинков, которые могут существовать для разделяемой библиотеки, обычно выглядит следующим образом:

```
libmyShared.so.1.3.2                    ①
libmyShared.so.1 --> libmyShared.so.1.3.2   ②
libmyShared.so   --> libmyShared.so.1       ③
```

① Фактический бинарник с версией, собранный проектом.

② Символическая ссылка, имя которой является сонамом библиотеки. При семантическом версионировании она будет содержать в своем имени только основную часть версии.

③ Namelink без сведений о версии, встроенных в имя файла. Это необходимо для того, чтобы библиотека была найдена, когда командная строка компоновщика содержит опцию типа -lmyShared.

При установке сущностей LIBRARY можно указать опции NAMELINK_ONLY или NAMELINK_SKIP. Опция NAMELINK_ONLY приведет к установке только namelink, тогда как NAMELINK_SKIP приведет к установке всего, кроме namelink. Если целевая библиотека не имеет сведений о версии или платформа не поддерживает namelinks, поведение этих двух опций меняется. NAMELINK_ONLY не установит ничего, а NAMELINK_SKIP установит настоящую библиотеку. Эти опции особенно полезны при создании отдельных пакетов времени выполнения и разработки, при этом часть namelink будет находиться в пакете разработки, а остальные файлы/ссылки - в пакете времени выполнения. Когда задана опция NAMELINK_ONLY, CMake не будет предупреждать об отсутствии блоков типов сущностей для других частей библиотеки, не упомянутых в команде install(). Это необходимо потому, что NAMELINK_SKIP и NAMELINK_ONLY не могут быть заданы в одном вызове install(), их нужно разделить на отдельные вызовы (см. пример ниже).

В каждом разделе entityType может быть указан параметр COMPONENT. Компоненты - это логическая группировка, используемая в основном для упаковки и подробно рассматриваемая в следующей главе, а пока думайте о них как о способе разделения различных установочных наборов. Вышеупомянутый сценарий

для разделения пакетов времени выполнения и пакетов разработки может быть настроен следующим образом:

```
install(TARGETS myShared myStatic
    RUNTIME
        DESTINATION ${CMAKE_INSTALL_BINDIR}
        COMPONENT MyProj_Runtime
    LIBRARY
        DESTINATION ${CMAKE_INSTALL_LIBDIR}
        NAMELINK_SKIP
        COMPONENT MyProj_Runtime
    ARCHIVE
        DESTINATION ${CMAKE_INSTALL_LIBDIR}
        COMPONENT MyProj_Development
)
# Because NAMELINK_ONLY is given, CMake won't complain about a missing RUNTIME block
install(TARGETS myShared
    LIBRARY
        DESTINATION ${CMAKE_INSTALL_LIBDIR}
        NAMELINK_ONLY
        COMPONENT MyProj_Development
)
```

Начиная с CMake 3.12, доступен более простой способ разделения namelink на другой компонент с помощью опции NAMELINK_COMPONENT. Эта опция может использоваться вместе с COMPONENT, но только в блоке LIBRARY. Используя эту новую опцию, можно выразить вышеописанное более лаконично:

```
install(TARGETS myShared myStatic
    RUNTIME
        DESTINATION ${CMAKE_INSTALL_BINDIR}
        COMPONENT MyProj_Runtime
    LIBRARY
        DESTINATION ${CMAKE_INSTALL_LIBDIR}
        COMPONENT          MyProj_Runtime
        NAMELINK_COMPONENT MyProj_Development   # Requires CMake 3.12 or later
    ARCHIVE
        DESTINATION ${CMAKE_INSTALL_LIBDIR}
        COMPONENT MyProj_Development
)
```

Если для блока не указан COMPONENT, он ассоциируется с компонентом по умолчанию, имя которого задается переменной CMAKE_INSTALL_DEFAULT_COMPONENT_NAME. Если эта переменная не установлена, то в качестве имени компонента по умолчанию используется Unspecified. Примером, когда может быть полезно изменить имя компонента по умолчанию, является ситуация, когда дочерний проект третьей стороны не использует никаких компонентов установки. Чтобы сохранить артефакты установки этого дочернего проекта отдельно от основного проекта, имя по умолчанию можно изменить непосредственно перед вызовом add_subdirectory(), чтобы подтянуть дочерний проект к основной сборке.

Опция EXCLUDE_FROM_ALL может быть использована для ограничения блока сущностей, чтобы он устанавливался только при установке, специфичной для конкретного компонента. По умолчанию установка не является компонентно-специфичной и устанавливаются все компоненты, но реализации упаковки могут устанавливать конкретные компоненты по отдельности. В CMake 3.12 была добавлена документация,

показывающая, как это можно сделать и из командной строки. Для большинства проектов EXCLUDE_FROM_ALL вряд ли понадобится.

Ключевое слово OPTIONAL также используется редко. Если тип сущности цели должен присутствовать, но он отсутствует (например, библиотека импорта Windows DLL для раздела типа сущности ARCHIVE), CMake не будет считать это ошибкой. Используйте эту опцию с осторожностью, поскольку она может маскировать неправильную конфигурацию логики сборки/установки.

Блок типа сущности также можно сделать специфичным для конфигурации, добавив к нему опцию CONFIGURATIONS. Этот тип сущности будет установлен только в том случае, если текущий тип сборки является одним из перечисленных. Тип сущности не может быть перечислен более одного раза для одной команды install(), поэтому если для разных конфигураций требуются разные детали, необходимо несколько вызовов. В следующем примере показано, как установить Debug и Release версии статических библиотек в разные каталоги:

```
install(TARGETS myStatic
    ARCHIVE
        DESTINATION ${CMAKE_INSTALL_LIBDIR}/Debug
        CONFIGURATIONS Debug
)
install(TARGETS myStatic
    ARCHIVE
        DESTINATION ${CMAKE_INSTALL_LIBDIR}/Release
        CONFIGURATIONS Release RelWithDebInfo MinSizeRel
)
```

Ключевое слово CONFIGURATIONS также может предшествовать всем блокам сущностей и действовать как значение по умолчанию для тех, которые не предоставляют собственного переопределения конфигурации. В следующем примере все блоки устанавливаются только для сборки Release, за исключением блока ARCHIVE, который устанавливается для Debug и Release.

```
install(TARGETS myShared myStatic
    CONFIGURATIONS Release
    RUNTIME
        DESTINATION ${CMAKE_INSTALL_BINDIR}
    LIBRARY
        DESTINATION ${CMAKE_INSTALL_LIBDIR}
    ARCHIVE
        DESTINATION ${CMAKE_INSTALL_LIBDIR}
        CONFIGURATIONS Debug Release
)
```

25.2.1. Свойства интерфейса

Если цели экспортируются (об этом говорится в разделе 25.3, "Установка экспорта", далее), то у них есть возможность установить свойства интерфейса, которые будут потребляться целями других проектов. Различные свойства цели INTERFACE переносятся в экспортируемые детали установленной цели автоматически, но требуется специальная обработка для учета отличий между потребностями создания цели и потреблением установленной цели. Рассмотрим следующий пример кода:

```
add_library(foo STATIC ...)
target_include_directories(foo
    INTERFACE ${CMAKE_CURRENT_BINARY_DIR}/somewhere
              ${MyProject_BINARY_DIR}/anotherDir
)
install(TARGETS foo
    DESTINATION ...
)
```

В самой сборке все, что ссылается на foo, будет иметь абсолютные пути к somewhere и anotherDir, добавленные к пути поиска заголовка. Когда foo будет установлен, он может быть упакован и развернут на совершенно другой машине. Очевидно, что пути к somewhere и anotherDir больше не будут иметь смысла, но приведенный выше пример все равно добавит их в путь поиска заголовков потребляемых целей. Необходим способ сказать: "Используйте путь xxx при сборке и путь yyy при установке", что как раз и делают выражения генераторов BUILD_INTERFACE и INSTALL_INTERFACE:

```
include(GNUInstallDirs)
target_include_directories(foo
    INTERFACE
        $<BUILD_INTERFACE:${CMAKE_CURRENT_BINARY_DIR}/somewhere>
        $<BUILD_INTERFACE:${MyProject_BINARY_DIR}/anotherDir>
        $<INSTALL_INTERFACE:${CMAKE_INSTALL_INCLUDEDIR}>
)
```

$<BUILD_INTERFACE:xxx> будет расширяться до xxx для дерева сборки и не будет расширяться до ничего при установке, тогда как $<INSTALL_INTERFACE:yyy> делает наоборот, гарантируя, что yyy будет добавлен только для установленной цели. В случае INSTALL_INTERFACE, yyy обычно является относительным путем, который рассматривается как относительный к базовому месту установки.

Хотя путь поиска заголовков в дереве сборки может быть разным для разных целей, очень часто после установки все цели имеют один и тот же путь поиска заголовков. В приведенном выше примере используется CMAKE_INSTALL_INCLUDEDIR, который, скорее всего, будет повторяться для каждой устанавливаемой цели, но указывать его отдельно для каждой цели - не самый удобный подход. Вместо этого можно использовать опцию INCLUDES команды install(), чтобы указать ту же информацию для группы целей. Все каталоги, указанные после INCLUDES DESTINATION, добавляются к свойству INTERFACE_INCLUDE_DIRECTORIES каждой из перечисленных целей. Это приводит к более сжатому описанию деталей пути поиска заголовка.

```
add_library(myStatic STATIC ...)
add_library(myHeaderOnly INTERFACE ...)

target_include_directories(myStatic
    PUBLIC $<BUILD_INTERFACE:${CMAKE_CURRENT_BINARY_DIR}/static_exports>
)
target_include_directories(myHeaderOnly
    INTERFACE $<BUILD_INTERFACE:${CMAKE_CURRENT_LIST_DIR}>
)

install(TARGETS myStatic myHeaderOnly
    ARCHIVE
        DESTINATION ${CMAKE_INSTALL_LIBDIR}
    INCLUDES
        DESTINATION ${CMAKE_INSTALL_INCLUDEDIR}
)
```

В отличие от других блоков типа сущности, для INCLUDES DESTINATION при необходимости можно перечислить несколько каталогов, хотя на практике это, скорее всего, встречается реже. Также обратите внимание, что блок INCLUDES не поддерживает ни одну из других деталей, которые поддерживают другие блоки типа сущности, он может указывать только ключевое слово DESTINATION, за которым следует одно или несколько местоположений.

25.2.2. RPATH

Когда библиотека или исполняемый файл загружается операционной системой, она должна найти все другие общие библиотеки, с которыми был связан двоичный файл. Различные платформы используют для этого разные способы. Windows полагается на поиск всех необходимых библиотек путем поиска в переменной окружения PATH, а также в каталоге, в котором находится двоичный файл. Другие платформы используют различные переменные окружения, специально предназначенные для этой цели, такие как LD_LIBRARY_PATH или их вариации, в сочетании с другими механизмами, такими как библиотеки, перечисленные в файлах conf. Недостатком зависимости от переменных окружения является то, что она зависит от того, правильно ли человек или процесс, загружающий двоичный файл, настроил окружение.

Во многих случаях пакет, предоставляющий двоичный файл, уже знает, где можно найти многие из зависимых библиотек, поскольку они могли быть частью того же пакета. Большинство платформ, отличных от Windows, поддерживают возможность кодирования путей поиска библиотек непосредственно в самих двоичных файлах. Общее название этой возможности - поддержка *пути выполнения* или RPATH, хотя на самом деле название может иметь специфические для конкретной платформы вариации. Благодаря встроенным деталям RPATH двоичный файл может быть самодостаточным и не зависеть от путей, предоставляемых окружением или конфигурацией системы. Более того, RPATH может содержать определенные заполнители, которые позволяют эффективно определять относительные пути, которые разрешаются в абсолютные только во время выполнения. Заголовки позволяют разрешать эти пути на основе местоположения двоичного файла, поэтому перемещаемые пакеты могут определять детали RPATH, которые только жестко кодируют пути, основанные на относительном расположении пакета.

Как и в случае со свойствами интерфейса в предыдущем разделе, существуют противоречивые потребности в RPATH в дереве сборки и для установленных двоичных файлов. В дереве сборки разработчикам необходимо, чтобы двоичные файлы могли найти разделяемые библиотеки, с которыми они связаны, чтобы можно было запускать исполняемые файлы (например, для отладки, выполнения тестов и т.д.). На платформах,

поддерживающих RPATH, CMake будет встраивать необходимые пути по умолчанию, тем самым предоставляя разработчикам наиболее удобный опыт, не требующий дополнительной настройки. Однако эти детали RPATH подходят только для конкретного дерева сборки, поэтому при установке целей CMake переписывает их с заменой путей (замена по умолчанию дает пустой RPATH).

Значения RPATH по умолчанию являются разумной отправной точкой, но они вряд ли подойдут для установленных целей. Проекты захотят переопределить поведение по умолчанию, чтобы обеспечить надлежащее обслуживание как дерева сборки, так и установленных сценариев. CMake позволяет раздельно контролировать расположение RPATH сборки и установки, поэтому проекты могут реализовать стратегию, которая наилучшим образом соответствует их потребностям. Следующие целевые свойства и переменные могут быть полезны для влияния на поведение RPATH:

BUILD_RPATH

Это свойство target можно использовать для указания дополнительных путей поиска, которые будут встроены в бинарное дерево сборки. Это будет дополнение к путям, автоматически добавляемым CMake для зависимостей ссылок этого двоичного файла, поэтому следует указывать только дополнительные пути, которые CMake не может определить самостоятельно. Это свойство необходимо только в том случае, если двоичный файл загружает несвязанные библиотеки во время выполнения с помощью dlopen() или другого эквивалентного механизма, например, при загрузке дополнительных подключаемых модулей. Это свойство инициализируется значением переменной CMAKE_BUILD_RPATH в момент создания цели с помощью add_library() или add_executable(). Хотя автоматически добавляемые пути поддерживались в CMake уже давно, свойство BUILD_RPATH и переменная CMAKE_BUILD_RPATH были добавлены только в CMake 3.8.

INSTALL_RPATH

Это целевое свойство задает RPATH двоичного файла при его установке. В отличие от RPATH сборки, CMake по умолчанию не предоставляет содержимого RPATH установки, поэтому проект должен установить это свойство в список путей, отражающих установленный макет. Ниже подробно описано, как это можно сделать. Это свойство инициализируется значением переменной CMAKE_INSTALL_RPATH при создании цели.

INSTALL_RPATH_USE_LINK_PATH

Когда это свойство цели установлено в true, путь каждой библиотеки, на которую ссылается эта цель, добавляется в набор мест RPATH установки, но только если путь указывает на место за пределами исходного и двоичного каталогов проекта. Это в основном полезно для вставки абсолютных путей к внешним библиотекам, которые не являются частью проекта, но которые, как ожидается, будут находиться в одном и том же месте на всех машинах, на которых будет развернут проект. Используйте это свойство с осторожностью, поскольку такие предположения могут снизить надежность установленного пакета (пути могут измениться с будущими выпусками внешних библиотек, системные администраторы могут выбрать конфигурацию установки не по умолчанию и т.д.). Это свойство инициализируется значением переменной CMAKE_INSTALL_RPATH_USE_LINK_PATH при создании цели.

BUILD_WITH_INSTALL_RPATH

Некоторые проекты используют схему сборки, зеркально отражающую установленную схему, и в этом случае установочный RPATH может также подходить для дерева сборки. Если установить это целевое свойство в true, RPATH сборки не будет использоваться, а вместо него в двоичный файл во время сборки будет встроен RPATH установки. Обратите внимание, что это может вызвать проблемы при сборке во время линковки, если используются плейсхолдеры, поддерживаемые загрузчиком, но не линковщиком (об этом подробнее ниже). Это свойство инициализируется параметром

Переменная CMAKE_BUILD_WITH_INSTALL_RPATH при создании цели.

SKIP_BUILD_RPATH

Когда это свойство цели имеет значение true, RPATH сборки не задается. BUILD_RPATH будет игнорироваться, и CMake не будет автоматически добавлять записи RPATH для библиотек, на которые ссылается цель. Обратите внимание, что это может привести к сбою сборки, если зависимые библиотеки ссылаются на другие библиотеки, поэтому используйте его с осторожностью. Это свойство инициализируется значением переменной CMAKE_SKIP_BUILD_RPATH при создании цели. Оно также переопределяется BUILD_WITH_INSTALL_RPATH, если это свойство имеет значение true.

CMAKE_SKIP_INSTALL_RPATH

Эта переменная является эквивалентом установки CMAKE_SKIP_BUILD_RPATH. Если установить ее в true, то свойства цели INSTALL_RPATH будут игнорироваться, и, скорее всего, установленные цели не смогут найти свои зависимые библиотеки во время выполнения, поэтому ее полезность сомнительна. Обратите внимание, что не существует целевого свойства SKIP_INSTALL_RPATH, только переменная CMAKE_SKIP_INSTALL_RPATH.

CMAKE_SKIP_RPATH

Установка этой переменной в true приводит к отключению поддержки RPATH и игнорированию всех вышеперечисленных свойств и переменных. Обычно это нежелательно делать, если только проект не управляет загрузкой библиотек во время выполнения каким-либо другим способом, но в целом функциональность RPATH должна быть предпочтительнее.

Расположение RPATH при установке в идеале должно быть основано на относительных путях. Это достигается на большинстве Unix-платформ использованием местозаполнителя $ORIGIN для представления местоположения двоичного файла, в который встраивается RPATH. Например, ниже приводится обычный способ определения деталей RPATH для проектов, которые следуют схеме, аналогичной той, которую определяет модуль GNUInstallDirs по умолчанию:

```
set(CMAKE_INSTALL_RPATH $ORIGIN $ORIGIN/../lib)
```

Чтобы сделать это более надежным и учесть возможные изменения по сравнению со стандартным расположением, необходимо проделать немного больше работы. Необходимо определить относительный путь от каталога исполняемых файлов к каталогу библиотек, что можно сделать следующим образом:

```
include(GNUInstallDirs)
file(RELATIVE_PATH relDir
     ${CMAKE_CURRENT_BINARY_DIR}/${CMAKE_INSTALL_BINDIR}
     ${CMAKE_CURRENT_BINARY_DIR}/${CMAKE_INSTALL_LIBDIR})
set(CMAKE_INSTALL_RPATH $ORIGIN $ORIGIN/${relDir})
```

Все цели, определенные после вышеуказанного, будут иметь INSTALL_RPATH, который направляет загрузчик искать в том же каталоге, что и двоичный файл, а также что-то вроде ../lib или его эквивалент для платформы относительно местоположения двоичного файла. Таким образом, для исполняемых файлов, устанавливаемых в bin, и разделяемых библиотек, устанавливаемых в lib, это гарантирует, что и те и другие смогут найти любые другие библиотеки, предоставляемые проектом. Это настоятельно рекомендуется в качестве отправной точки при первом добавлении поддержки RPATH в проекты. Обратите внимание, что цели Apple работают немного по-другому и могут иметь значительно отличающуюся компоновку, поэтому вышеизложенное необходимо адаптировать для этой платформы (это обсуждается в следующем разделе).

Один недостаток, о котором следует знать, заключается в том, что в то время как загрузчики понимают $ORIGIN, компоновщик, скорее всего, не понимает. Это может привести к проблемам, когда что-то ссылается на библиотеку, которая сама ссылается на другую библиотеку. Первый уровень связывания не представляет проблемы, поскольку библиотека будет указана непосредственно в командной строке компоновщика, но второй уровень зависимости библиотек должен быть найден компоновщиком. Когда компоновщик не понимает $ORIGIN, он не может найти библиотеку второго уровня через детали RPATH. Поэтому, если путь не указан другой опцией, например -L, компоновка завершится неудачей, даже если библиотека первого уровня технически содержит всю необходимую информацию. Это известная проблема в целом, не специфичная для CMake, это просто слабость популярных компоновщиков (в частности, компоновщика GNU ld).

В зависимости от различных свойств и переменных, упомянутых выше, CMake может потребоваться изменить встроенные данные RPATH цели при ее установке. Это может быть сделано двумя способами. Если двоичный файл имеет формат ELF, то по умолчанию CMake использует внутренний инструмент для перезаписи RPATH непосредственно в устанавливаемом двоичном файле. Значение RPATH в заголовках ELF имеет фиксированный размер, но CMake гарантирует, что для установочного RPATH будет достаточно места, при необходимости заполняя RPATH сборки. Детали того, как это делается, в основном скрыты от разработчика, за исключением, возможно, некоторых странных опций в командной строке компоновщика во время сборки. Для неELF-платформ CMake перекомпонует двоичный файл во время установки, указывая вместо этого данные RPATH установки. Исторически сложилось так, что это может иногда сбивать с толку разработчиков, которые недоумевают, почему то, что уже было собрано, должно быть связано снова, но в конечном итоге повторное связывание - это прагматичный способ получить желаемый конечный результат. Повторное связывание можно принудительно использовать и для ELF-платформ, установив переменную CMAKE_NO_BUILTIN_CHRPATH в true, но обычно это не следует использовать, если только внутреннее переписывание RPATH по какой-то причине не сработало.

При перекрестной компиляции несколько других переменных могут изменять местоположения RPATH, встроенные в двоичные файлы. Любое место RPATH, начинающееся с CMAKE_STAGING_PREFIX, автоматически заменяется префиксом CMAKE_INSTALL_PREFIX. Это справедливо как для мест RPATH сборки, так и установки. В любом установочном RPATH, начинающемся с CMAKE_SYSROOT, этот префикс будет полностью удален.

25.2.3. Цели, специфичные для Apple

Загрузчик и компоновщик Apple работают немного иначе, чем на других платформах Unix. Если библиотеки на таких платформах, как Linux, кодируют только имя библиотеки в разделяемой библиотеке (т.е. *soname*), то платформы Apple кодируют полный путь к библиотеке. Этот полный путь называется имя_установки, а часть имени_установки - путь - иногда называется имя_установки_dir. Все, что ссылается на библиотеку, также кодирует полное имя install_name как библиотеку для поиска. Когда все установлено в ожидаемое место, это работает хорошо, но для перемещаемых пакетов (к которым относится большинство пакетов приложений) это слишком негибко. Чтобы решить эту проблему, Apple поддерживает относительные базовые точки, аналогичные $ORIGIN, но держатели отличаются:

@ loader_path

Это более или менее эквивалент $ORIGIN от Apple, но компоновщик способен понять его и поэтому не страдает от проблем, с которыми сталкиваются другие компоновщики, неспособные декодировать $ORIGIN.

@исполняемый_путь

Это будет заменено местоположением выполняемой программы. Для библиотек, подключаемых в качестве зависимостей от других библиотек, это менее полезно, поскольку требует от библиотек знать

местоположение любого исполняемого файла, который может их использовать. Это обычно нежелательно, поэтому @loader_path обычно является лучшим выбором.

@rpath

Это может быть использовано в качестве заполнителя для части имени_установки_дира или полностью заменить имя_установки_дира.

Комбинация @loader_path и @rpath может быть использована для достижения такого же поведения, как и на других платформах Unix, поддерживающих $ORIGIN. CMake предоставляет дополнительные элементы управления, специфичные для Apple, чтобы помочь настроить все соответствующим образом:

MACOSX_RPATH

Когда это целевое свойство установлено в true, CMake автоматически устанавливает имя_установки_dir в @rpath при сборке для платформ Apple. Это поведение по умолчанию с версии CMake 3.0 и почти всегда желательно. Его можно переопределить с помощью INSTALL_NAME_DIR. Если переменная CMAKE_MACOSX_RPATH установлена во время создания цели, она используется для инициализации значения свойства MACOSX_RPATH.

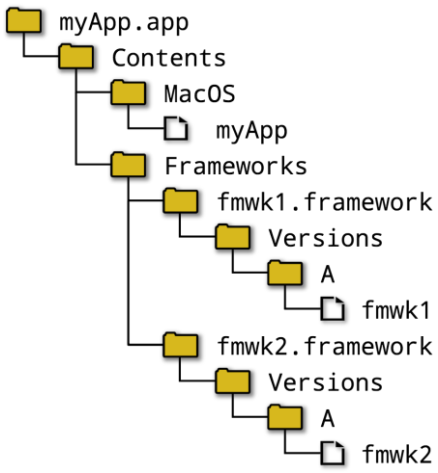
ИМЯ_УСТАНОВКИ_ДИР

Это целевое свойство используется для явного задания части install_name_dir имени_установки библиотеки. Имя_установки по умолчанию обычно имеет вид @rpath/libsomename.dylib, но для случаев, когда @rpath не подходит, INSTALL_NAME_DIR может указать альтернативу. Свойство инициализируется значением переменной CMAKE_INSTALL_NAME_DIR в момент создания. Это свойство игнорируется на платформах, отличных от Apple.

Для макетов, не относящихся к пучкам, поведение $ORIGIN можно распространить и на случай Apple:

```
if(APPLE)
    set(basePoint @loader_path)
else()
    set(basePoint $ORIGIN)
endif()
include(GNUInstallDirs)
file(RELATIVE_PATH relDir
     ${CMAKE_CURRENT_BINARY_DIR}/${CMAKE_INSTALL_BINDIR}
     ${CMAKE_CURRENT_BINARY_DIR}/${CMAKE_INSTALL_LIBDIR}
)
set(CMAKE_INSTALL_RPATH ${basePoint} ${basePoint}/${relDir})
```

Как только используются пакеты или фреймворки Apple, компоновка Apple полностью отличается от других платформ, и описанная выше стратегия оказывается бесполезной. Для таких случаев существуют различные стратегии определения путей поиска во время выполнения. Например, пакет приложений для macOS может иметь следующую структуру после установки соответствующих целей или в результате копирования фреймворков на этапе постбилда (показаны только соответствующие части структуры пакета):



Детали RPATH для вышеописанной схемы можно реализовать, установив целевое свойство INSTALL_RPATH для myApp в @executable_path/.../Frameworks, а для fmwk1 и fmwk2 - в @loader_path/.../.../.... Для поддержки сценария копирования фреймворка после сборки также используйте детали RPATH установки во время сборки. Если опустить детали для копирования фреймворка после сборки и подписи кода, такая схема может выглядеть следующим образом:

```
set(CMAKE_BUILD_WITH_INSTALL_RPATH YES)
set(CMAKE_BUILD_WITH_INSTALL_NAME_DIR YES)

add_executable(myApp MACOSX_BUNDLE ...)
add_library(fmwk1 SHARED ...)
add_library(fmwk2 SHARED ...)
target_link_libraries(myApp PRIVATE fmwk1) # Only needs fmwk1 directly...
target_link_libraries(fmwk1 PRIVATE fmwk2) # ... but fmwk1 needs fmwk2

set_target_properties(myApp PROPERTIES
    INSTALL_RPATH @executable_path/../Frameworks
)
set_target_properties(fmwk1 fmwk2 PROPERTIES
    FRAMEWORK TRUE
    INSTALL_RPATH @loader_path/../../..
)
```

Если стратегия проекта заключается в том, чтобы внедрять фреймворк только во время установки, то достаточно сделать следующее:

```
install(TARGETS fmwk1 fmwk2 myApp
    BUNDLE    DESTINATION .
    FRAMEWORK DESTINATION myApp.app/Contents/Frameworks
)
```

С другой стороны, если проект хочет внедрить фреймворк во время сборки, шаг после сборки может быть реализован относительно просто. Обратите внимание, однако, что выражения генератора TARGET_BUNDLE_DIR и TARGET_BUNDLE_CONTENT_DIR, используемые в следующем примере, доступны только в CMake 3.9 или более поздней версии:

```
add_custom_command(TARGET myApp POST_BUILD
    COMMAND rsync -a
            $<TARGET_BUNDLE_DIR:fmwk1>
            $<TARGET_BUNDLE_DIR:fmwk2>
            $<TARGET_BUNDLE_CONTENT_DIR:myApp>/Frameworks/
)
```

Приведенный выше шаг копирования имеет проблемы с надежностью, например, не удаляет старое содержимое, но для некоторых ситуаций он достаточно хорош, или, по крайней мере, является хорошей отправной точкой.

Если пакет должен быть подписан, то встраивание фреймворков в целом не очень хорошо поддерживается CMake. Как уже отмечалось в главе 22 "*Особенности Apple*", Apple предполагает, что подписание кода выполняется в Xcode в процессе сборки, а не на этапе после установки, а CMake предлагает очень мало помощи в процессе подписания. В настоящее время проекты должны реализовывать собственную логику, если они хотят подписывать пакеты со встроенными фреймворками.

Еще одна сложность возникает, если проект хочет создать универсальные двоичные файлы для iOS (иногда их также называют *толстыми* двоичными файлами). Сборка может быть как для устройства, так и для его симулятора. Обычно при установке устанавливается только одна архитектура, но CMake 3.5 и более поздние версии предлагают некоторую помощь в виде целевого свойства IOS_INSTALL_COMBINED. Если это свойство истинно, то при установке цели для сборки устройства она также создает архитектуру симулятора, устанавливает ее и объединяет в один двоичный файл. Обратное свойство также верно: установка сборки симулятора приводит к тому, что платформа устройства также собирается и устанавливается. Эта функция по-прежнему зависит от того, реализует ли проект собственную логику подписи кода, если это уместно.

Когда речь идет о встраивании заголовков во фреймворки, CMake предоставляет немного больше помощи. Как описано в разделе 22.3, "Фреймворки", цели могут указывать свои публичные и приватные заголовки в свойствах цели PUBLIC_HEADER и PRIVATE_HEADER. Затем они устанавливаются как часть установки самого фреймворка без необходимости дополнительной настройки. Когда те же цели собираются на платформах, отличных от Apple, не будет никакой структуры фреймворка для хранения заголовков (цели будут рассматриваться как обычные разделяемые библиотеки), но заголовки все равно могут быть установлены в указанное место:

```
install(TARGETS myShared
    FRAMEWORK     # Apple framework case
        DESTINATION ...
    LIBRARY       # Non-Apple case
        DESTINATION ...
    PUBLIC_HEADER
        DESTINATION ...
    PRIVATE_HEADER
        DESTINATION ...
)
```

25.3. Установка экспорта

Когда цели устанавливаются, они могут указать имя экспортного набора, к которому они принадлежат, используя опцию EXPORT в команде install(TARGETS). Затем этот экспортный набор может быть установлен с помощью другой формы команды:

```
install(EXPORT exportName
        DESTINATION dir
        [FILE name.cmake]
        [NAMESPACE namespace]
        [PERMISSIONS permissions...]
        [EXPORT_LINK_INTERFACE_LIBRARIES]
        [COMPONENT component]
        [EXCLUDE_FROM_ALL]
        [CONFIGURATIONS configs...]
)
```

Установка экспортного набора создает файл в указанном каталоге назначения с указанным именем файла name.cmake (оно должно заканчиваться на .cmake). Если параметр FILE не указан, используется имя файла по умолчанию, основанное на имени exportName. Сгенерированный файл будет содержать команды CMake, определяющие импортируемую цель для каждой цели в экспортируемом наборе. Этот файл предназначен для включения в другие проекты, чтобы они могли ссылаться на цели этого проекта и иметь полную информацию о свойствах интерфейса и межцелевых отношениях. С некоторыми ограничениями, потребляющий проект может обращаться с импортированными целями так же, как и со своими собственными обычными целями. Эти экспортные файлы обычно не включаются непосредственно в проекты, они предназначены для использования в конфигурационном пакете, который затем находят другие проекты с помощью команды find_package() (более подробно об этом говорится в разделе 25.7, "Написание файла конфигурационного пакета", далее в этой главе).

Если указан параметр NAMESPACE, то при создании соответствующей импортированной цели к ее имени будет добавлено пространство имен. Рассмотрим следующий пример:

```
add_library(myShared SHARED ...)
add_library(BagOfBeans::myShared ALIAS myShared)

install(TARGETS myShared
    EXPORT BagOfBeans
    DESTINATION ${CMAKE_INSTALL_LIBDIR}
)
install(EXPORT BagOfBeans
    DESTINATION ${CMAKE_INSTALL_LIBDIR}/cmake/BagOfBeans
    NAMESPACE BagOfBeans::
)
```

Приведенный выше пример следует советам из раздела 16.4, "Рекомендуемая практика", где каждая обычная цель также имеет связанный с ней ALIAS с пространством имен. При установке экспорта для неалиасной цели myShared используется то же пространство имен, что и для алиасной цели (т.е. BagOfBeans::). Это позволяет проектам, потребляющим экспортированные детали, ссылаться на цель так же, как этот проект может ссылаться на псевдоним (BagOfBeans::myShared). Проекты-потребители могут затем выбрать добавление этого проекта напрямую через add_subdirectory() или извлечение файла экспорта через find_package(), но при этом использовать то же самое имя цели BagOfBeans::myShared, независимо от того, какой метод был выбран. Этот важный паттерн становится довольно распространенным ожиданием от проектов в сообществе CMake, поэтому в интересах большинства проектов стараться следовать ему.

Имя экспортного набора, указанное после ключевого слова EXPORT, не обязательно должно быть связано с NAMESPACE. Пространство имен обычно тесно связано с именем проекта, но для именования экспортных наборов могут использоваться различные стратегии. Например, проект может определить несколько экспортных наборов с целями, которые разделяют одно пространство имен, и где экспортные наборы могут соответствовать логическим единицам, которые могут быть установлены как единое целое. Каждый из этих экспортных наборов может соответствовать одному установочному КОМПОНЕНТУ, или они могут объединять несколько компонентов. Ниже показаны эти случаи:

```
# Single component export
install(TARGETS algo1   EXPORT    MyProj_algoFree
        DESTINATION ... COMPONENT MyProj_free
)
install(EXPORT MyProj_algoFree
        DESTINATION ... COMPONENT MyProj_free
)
```

```
# Multi component export
install(TARGETS algo2   EXPORT    MyProj_algoPaid
        DESTINATION ... COMPONENT MyProj_licensed_A
)
install(TARGETS algo3   EXPORT    MyProj_algoPaid
        DESTINATION ... COMPONENT MyProj_licensed_B
)
install(EXPORT MyProj_algoPaid
        DESTINATION ... COMPONENT MyProj_licensed_dev
)
```

В приведенных выше примерах набор экспорта содержит только цель algo1, которая является членом компонента MyProj_free. Файл экспорта также является членом компонента MyProj_free, поэтому при установке этого компонента и библиотека, и файл экспорта будут установлены вместе. Ситуация отличается для многокомпонентного экспорта, где экспортный набор содержит algo2 из компонента MyProj_licensed_A и algo3 из компонента MyProj_licensed_B, но файл экспорта находится в отдельном компоненте. Таким образом, цели могут быть установлены с файлом экспорта или без него в зависимости от того, установлен ли компонент MyProj_licensed_dev или нет.

Приведенный выше пример экспорта нескольких компонентов подчеркивает важный аспект того, как должны устанавливаться экспортные наборы и компоненты. Ошибкой является установка файла экспорта без установки фактических целей, на которые указывает файл экспорта. Таким образом, если пользователь устанавливает компонент MyProj_licensed_dev, то компоненты MyProj_licensed_A и MyProj_licensed_B также должны быть установлены.

Из оставшихся опций команды install(EXPORT) ряд имеет схожие эффекты, как и для install(TARGETS). Опции PERMISSIONS, EXCLUDE_FROM_ALL и CONFIGURATIONS применяются к установленному экспортному файлу, а не к самим целям, но в остальном они эквивалентны. Место назначения, используемое для install(EXPORT), зависит от проекта, но есть некоторые соглашения, которые может быть полезно соблюдать. Мотивы этих соглашений связаны с основным способом использования экспортируемых файлов в составе конфигурационных пакетов, поэтому обсуждение этой темы отложено до Разд. 25.7, "Написание файла конфигурационного пакета" ниже.

Опция EXPORT_LINK_INTERFACE_LIBRARIES предназначена для поддержки старого поведения CMake до версии 3.0 и относится к библиотекам интерфейсов связей. Ее использование не рекомендуется, и проектам рекомендуется обновить CMake до минимальной версии 3.0.

Существует очень похожая форма команды install(), специально предназначенная для экспорта целей для использования в проектах Android ndk-build:

```
install(EXPORT_ANDROID_MK exportName
        DESTINATION dir
        [FILE name.mk]
        [NAMESPACE namespace]
        [PERMISSIONS permissions...]
        [EXPORT_LINK_INTERFACE_LIBRARIES]
        [COMPONENT component]
        [EXCLUDE_FROM_ALL]
        [CONFIGURATIONS configs...]
)
```

В то время как install(EXPORT) создает файл для других проектов CMake, который они могут использовать, install(EXPORT_ANDROID_MK) создает файл Android.mk, который ndk-build может включить. Файл Android.mk содержит все требования к использованию экспортируемых целей, поэтому проект ndk-build будет знать все определения компилятора, пути поиска заголовков и так далее, необходимые для компоновки. Имя экспортируемого файла может быть изменено с помощью опции FILE, но имя должно заканчиваться на .mk. Все остальные опции имеют такое же поведение, как и для формы install(EXPORT). install(EXPORT_ANDROID_MK) требует CMake 3.7 или более поздней версии, но проекты могут захотеть потребовать как минимум 3.11, чтобы избежать ошибки, которая затрагивала статические библиотеки с частными зависимостями.

В некоторых ситуациях может быть желательно иметь файл экспорта без необходимости выполнять установку. В качестве примера можно привести субсборки, компилируемые для платформы, отличной от основной сборки, или сторонние проекты, которые не могут быть добавлены в основную сборку напрямую из-за несовпадающих имен целей, неправильного использования переменных типа CMAKE_SOURCE_DIR и так далее. Для таких ситуаций CMake предоставляет команду export(), которая записывает экспортный файл непосредственно в дерево сборки:

```
export(EXPORT exportName
       [NAMESPACE namespace]
       [FILE fileName]
)
```

Вышеприведенные действия по сути эквивалентны упрощенной команде install(EXPORT), за исключением того, что файл экспорта записывается немедленно. Уменьшенный набор доступных опций имеет то же значение, что и для install(EXPORT), хотя имя файла может включать путь (он всё равно должен заканчиваться на .cmake). Некоторые другие формы команды export() позволяют экспортировать отдельные цели вместо экспортного набора, но если экспортные наборы уже определены, приведенная выше форма, вероятно, будет самой простой в использовании и поддержке.

25.4. Установка файлов и каталогов

В отличие от целей, установка отдельных файлов и каталогов менее сложна. Файлы устанавливаются с помощью следующей формы:

```
install(<FILES | PROGRAMS> files...
    DESTINATION dir
    [RENAME newName]
    [PERMISSIONS permissions...]
    [COMPONENT component]
    [EXCLUDE_FROM_ALL]
    [OPTIONAL]
    [CONFIGURATIONS configs...]
)
```

Большинство опций уже знакомы и имеют то же значение, что и для install(TARGETS). Единственное различие между install(FILES) и install(PROGRAMS) заключается в том, что последняя добавляет права на выполнение по умолчанию, если не заданы PERMISSIONS. Это предназначено для установки таких вещей, как сценарии оболочки, которые должны быть исполняемыми, но не являются целями CMake. Опция RENAME может быть задана, только если файл состоит из одного файла. Она позволяет присвоить этому файлу другое имя при установке.

В некоторых ситуациях проект может захотеть установить двоичные файлы, связанные с импортированной целью, но форма install(TARGETS) не позволяет устанавливать импортированные цели напрямую. Один из способов обойти это - установить файл(ы), связанный(ые) с импортированной целью, как обычные файлы. Все требования к использованию, связанные с целью, не будут сохранены, но это, по крайней мере, позволит установить двоичные файлы. Генераторное выражение $<TARGET_FILE:...> и другие подобные ему особенно полезны при использовании этой техники. Недостатком этого метода является то, что он перекладывает на проект ответственность за обработку всех различий платформ, что особенно проблематично для импортированных библиотечных целей.

```
# Assume myImportedExe is an imported target for an executable not built by this project
install(PROGRAMS $<TARGET_FILE:myImportedExe>
    DESTINATION ${CMAKE_INSTALL_BINDIR}
)
```

Установка каталогов происходит по той же схеме, что и файлов, но набор поддерживаемых опций расширен:

```
install(DIRECTORY dirs...
        DESTINATION dir
        [FILE_PERMISSIONS permissions... | USE_SOURCE_PERMISSIONS]
        [DIRECTORY_PERMISSIONS permissions...]
        [COMPONENT component]
        [EXCLUDE_FROM_ALL]
        [OPTIONAL]
        [CONFIGURATIONS configs...]
        [MESSAGE_NEVER]
        [FILES_MATCHING]
        # The following block can be repeated as many times as needed
        [ [PATTERN pattern | REGEX regex]
          [EXCLUDE]
          [PERMISSIONS permissions...] ]
)
```

При отсутствии любого из необязательных аргументов, для каждого местоположения dirs все дерево каталогов, начиная с этого места, устанавливается в целевой каталог dir. Если имя источника заканчивается косой чертой, то копируется содержимое исходного каталога, а не сам исходный каталог.

```
# Results in somewhere/foo/...
install(DIRECTORY foo DESTINATION somewhere)

# Results in somewhere/...
install(DIRECTORY foo/ DESTINATION somewhere)
```

Опции COMPONENT, EXCLUDE_FROM_ALL, OPTIONAL и CONFIGURATIONS имеют то же значение, что и для других команд install(). Опция MESSAGE_NEVER предотвращает сообщение журнала для каждого установленного файла, но можно утверждать, что её не следует использовать для согласованности с сообщениями для всего остального установленного содержимого.

Поддерживается несколько опций для управления разрешениями файлов и каталогов по отдельности. Если задано USE_SOURCE_PERMISSIONS, каждый установленный файл будет иметь те же разрешения, что и его источник.

FILE_PERMISSIONS отменяет это и использует указанные разрешения. Если ни одна из опций не указана, файлы будут иметь те же разрешения по умолчанию, что и при использовании команды install(FILE). Для каталогов, созданных при установке, опция DIRECTORY_PERMISSIONS может быть использована для отмены разрешений по умолчанию, которые такие же, как и для файлов, за исключением того, что добавляются разрешения на выполнение.

Остальные параметры позволяют отфильтровать набор файлов в соответствии с одним или несколькими шаблонами или регулярными выражениями. Каждый шаблон или регулярное выражение проверяется на соответствие полному пути к каждому файлу и каталогу (всегда указывается с прямыми косыми чертами, даже в Windows). Шаблоны символов подстановки должны соответствовать концу полного пути, а не только части в середине, в то время как регулярные выражения могут соответствовать любой части пути и поэтому являются более гибкими. Если за шаблоном или regex следует ключевое слово EXCLUDE, то все соответствующие файлы и каталоги не будут установлены. Это полезный способ исключить только несколько определенных вещей из дерева каталогов, но можно реализовать и обратный способ, указав ключевое слово FILES_MATCHING (один раз) перед любым блоком PATTERN или REGEX, что означает, что будут установлены только те файлы и каталоги, которые соответствуют одному из шаблонов или регексов. Если ни FILES_MATCHING, ни EXCLUDE не указаны, то единственным эффектом шаблона или регекса будет переопределение разрешений с помощью блока PERMISSIONS.

Некоторые примеры должны помочь прояснить вышеизложенные моменты. Следующий пример, слегка адаптированный из документации CMake, устанавливает все заголовки из каталога src и ниже, сохраняя структуру каталогов.

```
install(DIRECTORY src/
    DESTINATION include
    FILES_MATCHING
    PATTERN *.h
)
```

Следующий пример копирует код примера и некоторые скрипты, переопределяя разрешения последних для обеспечения их исполняемости:

```
install(DIRECTORY src/
    DESTINATION samples
    FILES_MATCHING
    REGEX "example\\.(h|c|cpp|cxx)"
    PATTERN *.txt
    PATTERN *.sh
      PERMISSIONS OWNER_READ OWNER_WRITE OWNER_EXECUTE
                  GROUP_READ GROUP_EXECUTE
                  WORLD_READ WORLD_EXECUTE
)
```

Ниже приводится установка документации, пропуская некоторые распространенные скрытые файлы:

```
install(DIRECTORY doc/ todo/ licenses
    DESTINATION doc
    FILES_MATCHING
    REGEX \\.(DS_Store|svn) EXCLUDE
)
```

В следующем примере опущены опции FILES_MATCHING или EXCLUDE, поэтому шаблоны и регексы только изменяют разрешения, а не фильтруют список файлов и каталогов:

```
install(DIRECTORY admin_scripts
    DESTINATION private
    PATTERN *.sh
      PERMISSIONS OWNER_READ OWNER_WRITE OWNER_EXECUTE
                  GROUP_READ GROUP_EXECUTE
)
```

Во всех случаях install(DIRECTORY) сохраняет структуру каталогов источника. Чтобы создать один пустой каталог в области установки, список источников может быть пустым, и DESTINATION все равно будет создан.

```
install(DIRECTORY DESTINATION somewhere/emptyDir)
```

25.5. Логика пользовательской установки

Бывают ситуации, когда простого копирования в область установки недостаточно. Может потребоваться произвольная обработка в процессе установки, например, перезапись части файла или программная генерация содержимого. Для таких случаев CMake предоставляет возможность добавить пользовательскую логику в шаг установки.

```
install(SCRIPT fileName | CODE cmakeCode
        [COMPONENT component]
        [EXCLUDE_FROM_ALL]
)
```

Форму CODE можно использовать для вставки команд CMake непосредственно в виде одной строки, тогда как форма SCRIPT будет использовать include() для чтения сценария во время установки. Обратите внимание, что

не указано, в какой момент процесса установки вызывается пользовательский код, но текущее поведение таково, что команды install() обычно обрабатываются в порядке их появления в области каталогов (но это не распространяется на вызовы install(), вложенные в подкаталоги).

Несколько блоков SCRIPT и/или CODE могут быть объединены в одной команде, и они будут выполнены в указанном порядке. Опции COMPONENT и EXCLUDE_FROM_ALL имеют свои обычные значения, но не могут быть заданы более одного раза.

```
install(CODE      [[ message("Starting custom script") ]]
        SCRIPT    myCustomLogic.cmake
        CODE      [[ message("Finished custom script") ]]
        COMPONENT MyProj_Runtime
)
```

25.6. Установка зависимостей

При создании пакетов обычно стремятся сделать их самодостаточными. Это может включать не только собственные артефакты сборки проекта, но и внешние зависимости, такие как библиотеки времени выполнения компилятора. CMake предоставляет некоторые модули, которые потенциально могут облегчить эту задачу.

Модуль InstallRequiredSystemLibraries предназначен для предоставления проектам сведений о соответствующих библиотеках времени выполнения для основных компиляторов. Это покрытие включает Intel (все основные платформы) и Visual Studio (только Windows). Использование модуля довольно простое: проекты могут либо позволить модулю определить команды install() от своего имени, либо попросить заполнить соответствующие переменные, чтобы создать необходимые команды самостоятельно. В простейшем случае проекты могут полагаться на значения по умолчанию, хотя рекомендуется установить хотя бы компонент для команд install().

```
set(CMAKE_INSTALL_SYSTEM_RUNTIME_COMPONENT MyProj_Runtime)
include(InstallRequiredSystemLibraries)
```

Местом установки по умолчанию является bin для Windows и lib для всех остальных платформ. Скорее всего, это соответствует типичной схеме установки большинства проектов, но это можно переопределить с помощью переменной CMAKE_INSTALL_SYSTEM_RUNTIME_DESTINATION:

```
include(GNUInstallDirs)
if(WIN32)
    set(CMAKE_INSTALL_SYSTEM_RUNTIME_DESTINATION ${CMAKE_INSTALL_BINDIR})
else()
    set(CMAKE_INSTALL_SYSTEM_RUNTIME_DESTINATION ${CMAKE_INSTALL_LIBDIR})
endif()
set(CMAKE_INSTALL_SYSTEM_RUNTIME_COMPONENT MyProj_Runtime)
include(InstallRequiredSystemLibraries)
```

Если проект хочет сам определять команды install(), ему необходимо установить CMAKE_INSTALL_SYSTEM_RUNTIME_LIBS_SKIP в true перед включением модуля. Затем проект может получить доступ к списку библиотек времени выполнения с помощью переменной CMAKE_INSTALL_SYSTEM_RUNTIME_LIBS:

```
set(CMAKE_INSTALL_SYSTEM_RUNTIME_LIBS_SKIP TRUE)
include(InstallRequiredSystemLibraries)
include(GNUInstallDirs)
if(WIN32)
    install(FILES ${CMAKE_INSTALL_SYSTEM_RUNTIME_LIBS}
        DESTINATION ${CMAKE_INSTALL_BINDIR}
    )
else()
    install(FILES ${CMAKE_INSTALL_SYSTEM_RUNTIME_LIBS}
        DESTINATION ${CMAKE_INSTALL_LIBDIR}
    )
endif()
```

При использовании компиляторов Intel команды install() по умолчанию устанавливают не только содержимое папки

CMAKE_INSTALL_SYSTEM_RUNTIME_LIBS. Они также устанавливают некоторые каталоги, не предоставленные проекту через какую-либо документированную переменную. Для тех разработчиков, которые заинтересованы в изучении того, желательно ли это дополнительное содержимое или нет, найдите CMAKE_INSTALL_SYSTEM_RUNTIME_DIRECTORIES в реализации модуля, чтобы увидеть, как строится это дополнительное содержимое.

Некоторые дополнительные средства контроля доступны при использовании компиляторов Visual Studio для установки различных других компонентов времени выполнения, таких как Windows Universal CRT, MFC и библиотеки OpenMP. Также можно принудительно установить отладочные версии библиотек времени выполнения. Все это подробно описано в документации модуля, поэтому заинтересованный читатель может обратиться к ней за более подробной информацией.

Другая пара модулей также может быть использована для установки зависимостей проекта во время выполнения.

Модули BundleUtilities и GetPrerequisites используют другой подход, напрямую опрашивая установленные двоичные файлы с помощью специфических для платформы инструментов и рекурсивно копируя недостающие библиотеки. Эти модули могут быть значительно сложнее в использовании и обычно не подходят для работы с зависимостями времени выполнения компилятора. Иногда они могут быть эффективны для поиска и установки зависимостей, которые могут быть не совсем предсказуемыми, например, для сложных кроссплатформенных наборов инструментов, таких как Qt (модуль DeployQt4 широко использует оба модуля). Большинству проектов лучше потратить усилия на поиск фактических зависимостей и установить их напрямую, чтобы процесс сборки был более предсказуемым и надежным, используя InstallRequiredSystemLibraries, чтобы позаботиться о зависимостях времени выполнения компилятора.

25.7. Запись файла пакета конфигурации

Предпочтительный способ для установленного проекта сделать себя доступным для использования другими проектами CMake - предоставить файл конфигурационного пакета. Этот файл находят потребляющие проекты с помощью команды find_package(), как было описано в разделе 23.5, "Поиск пакетов". Имя файла config должно соответствовать одной из двух форм:

- <packageName>Config.cmake
- <lowercasePackageName>-config.cmake

Первая из приведенных форм, возможно, встречается немного чаще и соответствует другим функциональным возможностям CMake, о которых речь пойдет ниже, но в остальном обе формы эквивалентны. Предполагается, что файл предоставляет импортированные цели для всех библиотек и исполняемых файлов, которые устанавливаемый проект хочет сделать доступными. Каталог, в который устанавливается файл конфигурации, должен быть одним из мест по умолчанию, которые будет искать find_package(), если базовая точка установки добавлена в переменную CMAKE_PREFIX_PATH. Это гарантирует, что файл конфигурации будет легко найти. Из раздела 23.5, "Поиск пакетов", полный набор мест, которые будут искаться, следующий:

```
<prefix>/
<prefix>/(cmake|CMake)/
<prefix>/<packageName>*/
<prefix>/<packageName>*/(cmake|CMake)/
<prefix>/(lib/<arch>|lib*|share)/cmake/<packageName>*/
<prefix>/(lib/<arch>|lib*|share)/<packageName>*/
<prefix>/(lib/<arch>|lib*|share)/<packageName>*/(cmake|CMake)/
<prefix>/<packageName>*/(lib/<arch>|lib*|share)/cmake/<packageName>*/
<prefix>/<packageName>*/(lib/<arch>|lib*|share)/<packageName>*/
<prefix>/<packageName>*/(lib/<arch>|lib*|share)/<packageName>*/(cmake|CMake)/
```

На платформах Apple также можно искать в следующих подкаталогах:

```
<prefix>/<packageName>.framework/Resources/
<prefix>/<packageName>.framework/Resources/CMake/
<prefix>/<packageName>.framework/Versions/*/Resources/
<prefix>/<packageName>.framework/Versions/*/Resources/CMake/
<prefix>/<packageName>.app/Contents/Resources/
<prefix>/<packageName>.app/Contents/Resources/CMake/
```

Очевидно, что это большой набор кандидатов, но лучший выбор зависит в некоторой степени от того, как предполагается устанавливать проект. При упаковке для включения в дистрибутив Linux, сам дистрибутив может иметь политику в отношении того, где должны находиться такие файлы. Вместо того, чтобы заставлять каждый дистрибутив вносить свои патчи в проект для обеспечения установки конфигурационного файла в соответствии с его политикой, проекты в идеале должны предоставлять способ передачи необходимых деталей в сборку. Переменная кэша идеально подходит для этой цели, так как проект может указать значение по умолчанию, но оно может быть отменено без необходимости изменять проект. В отсутствие каких-либо других ограничений, двумя очень простыми и часто используемыми местами являются <prefix>/cmake и <prefix>/lib/cmake/<packageName>, причем вариации последней немного более дружелюбны к многоархитектурным развертываниям (см. примеры ниже).

Для проектов, которые предоставляют файл Android.mk из команды install(EXPORT_ANDROID_MK), CMake не имеет специальных правил для его расположения. Разумным вариантом было бы использование специального каталога ndk-build в макете пакета, но в конечном счете это зависит от проекта.

25.7.1. Конфигурационные файлы для проектов CMake

Для простых проектов CMake, которые используют только один набор экспорта и не имеют зависимостей, команда install(EXPORT) может быть использована для создания базового конфигурационного файла напрямую:

```
include(GNUInstallDirs)
install(EXPORT myProj
    DESTINATION ${CMAKE_INSTALL_LIBDIR}/cmake/MyProj
    NAMESPACE MyProj::
    FILE MyProjConfig.cmake
)
```

Обратите внимание, как место назначения использует кэш-переменную CMAKE_INSTALL_LIBDIR, определенную модулем GNUInstallDirs, чтобы увеличить вероятность того, что дистрибутивам Linux не потребуется вносить никаких изменений. Модуль GNUInstallDirs уже учитывает общие случаи и, определяя переменные кэша, позволяет легко настраивать их при необходимости.

На практике файл конфигурации обычно не генерируется напрямую таким образом. Чаще подготавливается отдельный файл конфигурации, в который экспортируемые файлы добавляются с помощью команд include(). Немного расширенный пример с использованием двух наборов экспорта демонстрирует эту технику:

MyProjConfig.cmake

```
include("${CMAKE_CURRENT_LIST_DIR}/MyProj_Runtime.cmake")
include("${CMAKE_CURRENT_LIST_DIR}/MyProj_Development.cmake")
```

CMakeLists.txt

```
# Define targets, etc...

# Create two separate export sets installed to the same place
# and a manually written config file that will include them
include(GNUInstallDirs)
install(EXPORT MyProj_Runtime
    DESTINATION ${CMAKE_INSTALL_LIBDIR}/cmake/MyProj
    NAMESPACE MyProj::
    FILE MyProj_Runtime.cmake
    COMPONENT MyProj_Runtime
)
install(EXPORT MyProj_Development
    DESTINATION ${CMAKE_INSTALL_LIBDIR}/cmake/MyProj
    NAMESPACE MyProj::
    FILE MyProj_Development.cmake
    COMPONENT MyProj_Development
)
install(FILES MyProjConfig.cmake
    DESTINATION ${CMAKE_INSTALL_LIBDIR}/cmake/MyProj
)
```

Приведенный выше MyProjConfig.cmake по-прежнему очень прост. Никаких внешних зависимостей не требуется, и файл конфигурации предполагает, что всегда установлены оба компонента - и компонент выполнения, и компонент разработки. Рассмотрим сценарий, в котором компонент времени выполнения зависит от некоторого другого пакета с именем BagOfBeans. Конфигурационный файл отвечает за обеспечение доступности необходимых целей из BagOfBeans, что он обычно делает, вызывая find_package(). Для удобства макрос find_dependency() из модуля CMakeFindDependencyMacro иногда может использоваться в качестве обертки вокруг find_package() для прозрачной обработки ключевых слов QUIET и REQUIRED. Макрос

find_dependency() также имеет дополнительное поведение: если ему не удается найти запрашиваемый пакет, обработка конфигурационного файла немедленно прекращается и управление возвращается вызывающей стороне. Это как если бы вызов return() был сделан сразу после неудачного вызова find_dependency(). На практике это приводит к простой, чистой спецификации зависимостей с изящной обработкой отказов зависимостей.

MyProjConfig.cmake

```
include(CMakeFindDependencyMacro)
find_dependency(BagOfBeans)

include("${CMAKE_CURRENT_LIST_DIR}/MyProj_Runtime.cmake")
include("${CMAKE_CURRENT_LIST_DIR}/MyProj_Development.cmake")
```

Авторы проектов должны знать, что функция find_dependency() содержит оптимизацию, которая обходит вызов, если обнаруживает, что запрашиваемый пакет уже был найден ранее. Это работает хорошо, если только последующие вызовы не потребуют другого набора компонентов пакета. При первом успешном вызове find_dependency() эффективно фиксирует набор найденных компонентов. Если последующие вызовы find_dependency() передают другой набор компонентов, они игнорируются. Поэтому, если зависимость поддерживает компоненты пакета, проекты должны вместо этого вызывать find_package() напрямую и самостоятельно обрабатывать опции QUIET и REQUIRED. Эти опции передаются в конфигурационный файл как переменные ${CMAKE_FIND_PACKAGE_NAME}_FIND_QUIETLY и ${CMAKE_FIND_PACKAGE_NAME}_FIND_REQUIRED. Всегда используйте ${CMAKE_FIND_PACKAGE_NAME}, а не вводите имя пакета жестко, так как могут быть различия в верхнем и нижнем регистре.

```
unset(extraArgs)
if(${CMAKE_FIND_PACKAGE_NAME}_FIND_QUIETLY)
    list(APPEND extraArgs QUIET)
endif()
if(${CMAKE_FIND_PACKAGE_NAME}_FIND_REQUIRED)
    list(APPEND extraArgs REQUIRED)
endif()
find_package(BagOfBeans COMPONENTS Foo Bar ${extraArgs})
```

Если проект хочет поддерживать некоторые из своих собственных компонентов как необязательные, то сложность конфигурационного файла возрастает довольно значительно. Шаги, необходимые для полной поддержки такой функциональности, можно свести к следующему:

- Составьте набор компонентов проекта, которые необходимо найти. Начните с набора обязательных и необязательных компонентов из вызова find_package() и добавьте те, которые необходимы для удовлетворения зависимостей проекта.
- Разработайте набор внешних зависимостей, необходимых для данного набора компонентов проекта. Некоторые из них будут обязательными, другие могут быть необязательными, поэтому потребуется вывести два отдельных набора внешних зависимостей.
- Найдите внешние зависимости, и если какие-либо необходимые зависимости не удастся найти, операция поиска проекта также должна завершиться неудачей, а управление должно немедленно вернуться с соответствующим сообщением об ошибке. Отсутствие необязательных внешних зависимостей не должно приводить к сбою или сообщению об ошибке.

- Обновление набора компонентов проекта для удаления тех, которые зависят от отсутствующей необязательной внешней зависимости. Это может потребовать дальнейшей очистки набора компонентов проекта, если удаленные компоненты сами являются зависимостями других компонентов.
- Загрузите оставшиеся компоненты проекта.

Проекты также должны решить, что делать, если компоненты вообще не указаны. Это можно рассматривать так, как если бы все компоненты были указаны как необязательные или даже как обязательные. Другая стратегия заключается в том, чтобы загрузить минимальный набор необходимых компонентов и опустить все остальные. Наиболее подходящая стратегия будет зависеть от характера компонентов проекта. Набор запрашиваемых компонентов будет доступен в переменной ${CMAKE_FIND_PACKAGE_NAME}_FIND_COMPONENTS, и если компонент был указан как требуемый, а не необязательный, то ${CMAKE_FIND_PACKAGE_NAME}_FIND_REQUIRED_<comp> будет истинным для этого компонента.

Конфигурационные файлы не должны сообщать об ошибках с помощью message(), вместо этого они должны хранить сообщение об ошибке в переменной ${CMAKE_FIND_PACKAGE_NAME}_NOT_FOUND_MESSAGE. Затем она будет подхвачена функцией find_package(), которая обернет ее деталями о том, в каком месте проекта возникла ошибка.
${CMAKE_FIND_PACKAGE_NAME}_FOUND также должно быть установлено в false, чтобы указать на неудачу. Это позволяет find_package() правильно реализовать вызов, в котором не используется ключевое слово REQUIRED. Если бы в файле конфигурации пакета было написано message(FATAL_ERROR ...), то пакет никогда не мог бы рассматриваться вызывающей стороной как необязательный.

```
# Work out the set of components to load
if(${CMAKE_FIND_PACKAGE_NAME}_FIND_COMPONENTS)
    set(comps ${CMAKE_FIND_PACKAGE_NAME}_FIND_COMPONENTS)
    # Ensure Runtime is included if Development was specified
    if(Development IN_LIST comps AND NOT Runtime IN_LIST comps)
        list(APPEND comps Runtime)
    endif()
else()
    # No components given, look for all components
    set(comps Runtime Development)
endif()

# Find external dependencies, storing comps in a safer variable name.
# In this example, BagOfBeans is only needed by the Development component.
set(${CMAKE_FIND_PACKAGE_NAME}_comps ${comps})
if(Development IN_LIST ${comps})
    find_dependency(BagOfBeans)
endif()

# Check all required components are available before trying to load any
foreach(comp IN LISTS ${CMAKE_FIND_PACKAGE_NAME}_comps)
    if(${CMAKE_FIND_PACKAGE_NAME}_FIND_REQUIRED_${comp} AND
       NOT EXISTS ${CMAKE_CURRENT_LIST_DIR}/MyProj${comp}.cmake)
        set(${CMAKE_FIND_PACKAGE_NAME}_NOT_FOUND_MESSAGE
            "MyProj missing required dependency: ${comp}")
        set(${CMAKE_FIND_PACKAGE_NAME}_FOUND FALSE)
        return()
    endif()
endforeach()

foreach(comp IN LISTS ${CMAKE_FIND_PACKAGE_NAME}_comps)
    # All required components are known to exist. The OPTIONAL keyword
    # allows the non-required components to be missing without error.
    include(${CMAKE_CURRENT_LIST_DIR}/MyProj${comp}.cmake OPTIONAL)
endforeach()
```

Приведенный выше пример демонстрирует рекомендуемую практику не создавать никаких импортированных целей, прежде чем сначала проверить, могут ли быть удовлетворены требуемые компоненты. Это предотвращает создание импортированных целей для одних компонентов, но не для других в случае сбоя.

Близким спутником конфигурационного файла является связанный с ним файл версии. Если файл версии предоставлен, ожидается, что он будет иметь имя, соответствующее одной из двух форм, и должен находиться в том же каталоге, что и файл config:

- <packageName>ConfigVersion.cmake
- <lowercasePackageName>-config-version.cmake

Форма имени файла версии обычно соответствует форме имени связанного с ним файла конфигурации (т.е. FooConfigVersion.cmake будет идти вместе с FooConfig.cmake, тогда как foo-config-version.cmake обычно идет в паре с foo-config.cmake). Цель файла версии - сообщить find_package(), соответствует ли пакет заданным требованиям к версии. find_package() устанавливает ряд переменных до загрузки файла версии:

• PACKAGE_FIND_NAME
• PACKAGE_FIND_VERSION
• PACKAGE_FIND_VERSION_MAJOR
• PACKAGE_FIND_VERSION_MINOR
• PACKAGE_FIND_VERSION_PATCH
• PACKAGE_FIND_VERSION_TWEAK
• PACKAGE_FIND_VERSION_COUNT

Эти переменные содержат сведения о версии, указанной в качестве аргумента VERSION в find_package(). Если такой аргумент не был указан, то PACKAGE_FIND_VERSION будет пустым, а остальные переменные PACKAGE_FIND_VERSION_* будут установлены в 0. PACKAGE_FIND_VERSION_COUNT содержит счетчик того, сколько компонентов версии было указано, а остальные переменные имеют свое очевидное значение. Файл версии должен сверить запрашиваемые данные с фактической версией пакета и затем установить следующие переменные:

PACKAGE_VERSION

Это фактическая версия пакета, которая, как ожидается, будет иметь обычный формат *major.minor.patch.tweak* (не все компоненты являются обязательными).

PACKAGE_VERSION_EXACT

Устанавливается в true только в том случае, если версия пакета и запрашиваемая версия полностью совпадают.

PACKAGE_VERSION_COMPATIBLE

Устанавливается в true, только если версия пакета совместима с запрашиваемой версией. То, как пакет определяет совместимость, зависит от него самого. Для проектов, которые следуют принципам семантического версионирования, описанным в разделе 20.3, "Версионирование общих библиотек", переменная будет установлена в соответствии со следующими правилами:

- Если какой-либо компонент версии отсутствует, считайте его равным 0.
- Если *основные* компоненты версии отличаются, результат будет ложным.
- Если *основные* компоненты версии одинаковы, результат будет ложным, если *основной* компонент версии пакета меньше требуемого.
- Если компоненты *major* и *minor* версии одинаковы, результат будет ложным, если компонент *patch* версии пакета меньше требуемого.
- Если компоненты *major*, *minor* и *patch* version одинаковы, результат будет ложным, если компонент *tweak* version пакета меньше требуемого.
- Во всех остальных случаях результат устанавливается в true.

PACKAGE_VERSION_UNSUITABLE

Устанавливается в true только в том случае, если в файле версий необходимо указать, что пакет не может удовлетворить какому-либо требованию по версии (в основном пакет не имеет номера версии, поэтому любое требование по версии должно рассматриваться как отказ).

Команда find_package() будет использовать эту информацию, чтобы передать следующие переменные своему вызывающему пользователю, все из которых аналогичны тем, которые он передал в файл версии (возвращаемые значения здесь будут фактической версией пакета, а не требованиями к версии, переданными команде find_package()):

- <packageName>_VERSION
- <packageName>_VERSION_MAJOR
- <packageName>_VERSION_MINOR
- <packageName>_VERSION_PATCH
- <packageName>_VERSION_TWEAK
- <packageName>_VERSION_COUNT

Хотя проекты могут вручную создавать файл версии, гораздо более простым и, скорее всего, более надежным подходом является использование команды write_basic_package_version_file(), предоставляемой модулем CMakePackageConfigHelpers:

```
write_basic_package_version_file(outFile
    [VERSION requiredVersion]
    COMPATIBILITY compat
)
```

Если указан аргумент VERSION, то ожидается, что требуемая версия будет в обычной форме *major.minor.patch.tweak*, но обязательной является только *основная* часть. Если параметр VERSION не указан, вместо него используется переменная PROJECT_VERSION (устанавливается командой project()). Опция COMPATIBILITY задает стратегию определения совместимости. Аргумент compat должен быть одним из следующих значений (имейте в виду, что большинство названий немного вводят в заблуждение):

AnyNewerVersion

Версия пакета должна быть равна или больше указанной версии.

SameMajorVersion

Версия пакета должна быть равна или больше указанной версии, а *основная* часть номера версии пакета должна совпадать с номером в requiredVersion. Это соответствует тем же требованиям совместимости, что и семантическое версионирование.

SameMinorVersion

Версия пакета должна быть равна или больше указанной версии, а *мажорная* и *минорная* части номера версии пакета должны быть такими же, как в requiredVersion. Этот выбор поддерживается только в CMake 3.11 или более поздней версии.

ExactVersion

major, *minor* и *patch* части номера версии пакета должны быть такими же, как и в requiredVersion. Часть *tweak* игнорируется. Эта стратегия вводит в заблуждение, и в настоящее время обсуждается возможность ее отмены в пользу новой, более ясной стратегии.

Модуль CMakePackageConfigHelpers также предоставляет еще одну команду, которая иногда может быть полезной. Команда configure_package_config_file() предназначена для облегчения определения перемещаемого пакета в проектах путем предоставления некоторых удобств работы с путями. Обычно она не нужна для большинства проектов, но когда файл конфигурации пакета должен ссылаться на установленные файлы относительно базового места установки, а не местоположения самого файла конфигурации, она предоставляет более простой способ сделать это надежно. Команда имеет следующий вид:

```
configure_package_config_file(inputFile outputFile
    INSTALL_DESTINATION path
    [INSTALL_PREFIX prefix]
    [PATH_VARS var1 [var2...] ]
    [NO_SET_AND_CHECK_MACRO]
    [NO_CHECK_REQUIRED_COMPONENTS_MACRO]
)
```

Команда должна использоваться в качестве замены configure_file() для копирования файла <Проект> Файл Config.cmake.in с подстановками. Он заменит переменные вида @PACKAGE_<somevar>@ на содержимое <somevar>, преобразованное в абсолютный путь. Оригинальное содержимое будет считаться относительным к базовому расположению установки. Каждая переменная, которая должна быть преобразована таким образом, должна быть перечислена с помощью опции PATH_VARS. Чтобы эта функция работала, входной файл должен иметь @PACKAGE_INIT@ в верхней части или рядом с ней до любого использования заменяемых переменных.

INSTALL_DESTINATION - это каталог, в который будет установлен outputFile, относительно INSTALL_PREFIX. Если INSTALL_PREFIX опущен, по умолчанию используется CMAKE_INSTALL_PREFIX, что обычно является желаемым значением. INSTALL_PREFIX обычно указывается только в том случае, если файл outputFile будет использоваться непосредственно в дереве сборки, а не устанавливаться (т.е. использоваться вместе с командой export(EXPORT)).

Опции NO_SET_AND_CHECK_MACRO и NO_CHECK_REQUIRED_COMPONENTS_MACRO не позволяют @PACKAGE_INIT@ определять некоторые вспомогательные функции. До того, как импортированные цели стали предпочтительным способом предоставления целей пакета, необходимо было использовать переменные. Для этого в configure_package_config_file() был предусмотрен макрос set_and_check(), который устанавливал переменную, только если она еще не была определена. Проекты, предоставляющие импортированные цели, не должны нуждаться в этом макросе и могут добавить NO_SET_AND_CHECK_MACRO, чтобы предотвратить его определение. Аналогично, в прошлом, когда все детали предоставлялись через переменные, было принято проверять, все ли необходимые переменные были установлены в конце перед возвратом. Для этого был определен макрос check_required_components(), но проекты, предоставляющие импортированные цели, должны сами выполнять эти проверки и определять импортированные цели только в том случае, если все компоненты будут найдены. Это делает макрос check_required_components() излишним.

Пример должен помочь прояснить типичное использование этой команды:

CMakeLists.txt

```
include(GNUInstallDirs)
include(CMakePackageConfigHelpers)
set(cmakeModulesDir cmake)
configure_package_config_file(MyProjConfig.cmake.in MyProjConfig.cmake
    INSTALL_DESTINATION ${CMAKE_INSTALL_LIBDIR}/cmake/MyProj
    PATH_VARS cmakeModulesDir
    NO_SET_AND_CHECK_MACRO
    NO_CHECK_REQUIRED_COMPONENTS_MACRO
)
install(FILES ${CMAKE_CURRENT_BINARY_DIR}/MyProjConfig.cmake
    DESTINATION ${CMAKE_INSTALL_LIBDIR}/cmake/MyProj
    COMPONENT ...
)
```

MyProjConfig.cmake.in

```
@PACKAGE_INIT@

list(APPEND CMAKE_MODULE_PATH "@PACKAGE_cmakeModulesDir@")

# Include the project's export files, etc...
```

25.7.2. Конфигурационные файлы для проектов, не относящихся к CMake

Механизм конфигурационных файлов не ограничивается проектами, собранными с помощью CMake, его можно использовать и для проектов, не созданных с помощью CMake (хотя это пока не так часто встречается). Если проекты CMake могут использовать различные возможности CMake для более легкого создания необходимых файлов, то проекты, не использующие CMake, должны определять их вручную. Для таких проектов также важно, чтобы файлы были простыми, поскольку они, скорее всего, будут поддерживаться людьми, не очень хорошо знакомыми с CMake. Хороший первый шаг - изначально отказаться от поддержки компонентов и просто сделать пакет доступным в виде простого набора импортируемых целей. Для проектов, которым требуется только предоставление библиотек, ниже показан довольно минимальный

```
# Compute the base point of the install by getting the directory of this
# file and moving up the required number of directories
set(_IMPORT_PREFIX "${CMAKE_CURRENT_LIST_DIR}")
foreach(i RANGE 1 NumSubdirLevels)   ①
    get_filename_component(_IMPORT_PREFIX "${_IMPORT_PREFIX}" PATH)
    if(_IMPORT_PREFIX STREQUAL "/")
      set(_IMPORT_PREFIX "")
      break()
    endif()
endforeach()

# Use a prefix specific to this project
set(projPrefix MyProj)

# Example of defining a static library imported target
add_library(${projPrefix}::myStatic STATIC IMPORTED)
set_target_properties(${projPrefix}::myStatic PROPERTIES
    IMPORTED_LOCATION "${_IMPORT_PREFIX}/lib/libmyStatic.a"   ②
)

# Example of defining a shared library imported target with version details
add_library(${projPrefix}::myShared SHARED IMPORTED)
set_target_properties(${projPrefix}::myShared PROPERTIES
    IMPORTED_LOCATION "${_IMPORT_PREFIX}/lib/libmyShared.so.1.6.3"   ③
    IMPORTED_SONAME "libmyShared.so.1"                               ④
)

# Another example of a shared library, this time for Windows
add_library(${projPrefix}::myDLL SHARED IMPORTED)
set_target_properties(${projPrefix}::myDLL PROPERTIES
    IMPORTED_LOCATION "${_IMPORT_PREFIX}/bin/myShared.dll"
    IMPORTED_IMPLIB   "${_IMPORT_PREFIX}/lib/myShared.lib"   ⑤
)
```

①NumSubdirLevels - это количество уровней подкаталогов, в которых находится этот файл конфигурации ниже базовой точки установки. Например, если файл находится по адресу lib/cmake/Foo/FooConfig.cmake, то NumSubdirLevels должен быть равен 3.

② Укажите путь к библиотеке относительно базовой точки установки, которая была ранее найдена и сохранена в _IMPORT_PREFIX.

③ В примере показано, как номер версии разделяемой библиотеки будет помещен в конец имени файла для таких платформ, как Linux. Очевидно, что это зависит от конкретной платформы.

④ Для платформ, поддерживающих соназвания, IMPORTED_SONAME - это имя, которое будет встроено в двоичные файлы, ссылающиеся на эту цель. На платформах Apple это имя обычно имеет форму, включающую @rpath и, возможно, некоторые компоненты подкаталога.

⑤ Для Windows расположение библиотеки импорта, связанной с DLL, также должно быть указано для того, чтобы с ней можно было ссылаться. Если предполагается только предоставить DLL (например, чтобы

она была доступна во время выполнения, но не для прямого связывания), IMPORTED_IMPLIB можно не указывать, но это будет менее распространено.

Приведенный выше пример является довольно базовым, и очевидно, что различные свойства IMPORTED_... должны быть адаптированы для каждой платформы, но проект, не использующий CMake, волен использовать любые механизмы, которые он считает удобными для создания содержимого установленного конфигурационного файла. Для большей надежности каждая импортируемая библиотека будет добавлена только в том случае, если она еще не существует, как показано ниже:

```
if(NOT TARGET ${projPrefix}::myStatic)
    add_library(${projPrefix}::myStatic STATIC IMPORTED)
    set_target_properties(${projPrefix}::myStatic PROPERTIES
        IMPORTED_LOCATION "${_IMPORT_PREFIX}/lib/libmyStatic.a"
    )
endif()
```

25.8. Рекомендуемые практики

Установка - это нетривиальная тема, которая требует хорошего планирования и понимания каждой предполагаемой платформы развертывания. Обычно проект изначально ориентирован только на одну платформу или только на подмножество конечного предполагаемого набора платформ, но задержка любого планирования установки и развертывания может привести к тому, что придется иметь дело с неожиданными сложностями и различиями платформ на поздних стадиях цикла выпуска проекта. Проекты должны иметь четкое представление об установленной структуре файлов и каталогов, а также о полном наборе сценариев упаковки, которые в конечном итоге будут поддерживаться. Это может сильно повлиять на структуру проекта, включая такие фундаментальные вещи, как разделение функциональности между библиотеками и то, какие символы должны быть видны в двоичных файлах в результате.

Проекты должны предпочитать следовать стандартной компоновке пакетов, где это возможно. Использование такого модуля, как GNUInstallDirs, может значительно упростить эту задачу, даже для пакетов под Windows. Если это невозможно или нежелательно, проекты могут захотеть хотя бы рассмотреть возможность использования одинаковой структуры каталогов на разных платформах для упрощения разработки приложений.

Проектам настоятельно рекомендуется делать свои пакеты перемещаемыми. Если только пакет не должен быть установлен в очень конкретном месте, перемещаемые пакеты имеют значительные преимущества. Они обеспечивают гораздо большую гибкость для конечных пользователей, они легче поддерживают широкий спектр упаковочных систем и их легче тестировать во время разработки.

Выбор базовой точки установки по умолчанию зависит от платформы, и значения по умолчанию, предоставляемые CMake, не всегда идеальны, но при создании пакета они все равно часто переопределяются. Избегайте включения номера версии пакета в базовый путь установки, особенно для перемещаемых пакетов. Предпочтите оставить это решение на усмотрение пользователя, выполняющего установку, поскольку различные сценарии использования требуют различных структур каталогов, которые могут быть несовместимы с путями, специфичными для конкретной версии. Проекты также должны предпочитать следовать соответствующим стандартам, где это уместно, таким как Стандарт иерархии файловой системы для Linux (также подходит для большинства других платформ на базе Unix, кроме Apple).

При определении требований к использованию целей используйте выражение генератора $<BUILD_INTERFACE:...>, чтобы правильно выразить пути поиска заголовков, которые будут использоваться

сборкой. Для любой библиотечной цели, которая будет устанавливаться, предпочтите задать путь поиска заголовков с помощью секции INCLUDES DESTINATION команды install(TARGETS), а не использовать выражения генератора $<INSTALL_INTERFACE:...> на самой цели. Это может быть удобнее и лаконичнее. Убедитесь, что в INCLUDES DESTINATION используется относительный путь, который является относительным к базовой точке установки.

```
add_library(foo ...)

# Not ideal: embeds build paths in installed export files
target_include_directories(foo PUBLIC ${CMAKE_CURRENT_BINARY_DIR})

# Better: separate paths for build and install, with the latter
# added as part of the install() command rather than with the target
include(GNUInstallDirs)
target_include_directories(foo PUBLIC
    $<BUILD_INTERFACE:${CMAKE_CURRENT_BINARY_DIR}>
)
install(TARGETS foo ...
    INCLUDES DESTINATION ${CMAKE_INSTALL_INCLUDEDIR}
)
```

Всегда присваивайте КОМПОНЕНТ каждой установленной сущности и используйте имена компонентов, специфичные для проекта. Когда проект используется как часть большой иерархии проектов, это позволяет родительскому проекту контролировать обращение с дочерними компонентами. Хорошей схемой, которой следует придерживаться, будет
<ProjectName>_<ComponentName> , например MyProj_Runtime. При установке экспортных наборов используйте то же имя проекта, что и пространство имен, с добавлением двух двоеточий (например, MyProj::). Следование этим соглашениям об именовании сделает работу с установленным проектом более интуитивно понятной, но, что более важно, это также предотвратит столкновения имен с пакетами других проектов.

Если проект предоставляет библиотеки, с которыми должны компоноваться другие проекты, предпочтите определить отдельные компоненты для поддержки выполнения и для разработки. Это позволяет родительскому иерархическому проекту повторно использовать компонент времени выполнения для упаковки только общих библиотек и вещей, необходимых для выполнения, и избежать упаковки сущностей, предназначенных только для разработки, таких как статические библиотеки, заголовочные файлы и так далее. Это также потенциально сокращает работу сопровождающих пакетов (например, в дистрибутивах Linux), где пакеты часто разделяются на пакеты runtime и devel.

В файлах конфигурации пакетов всегда следите за тем, чтобы не создавались импортируемые цели, если вызов find_package() не будет успешным. Это означает, что все необходимые компоненты должны быть доступны и все необходимые целевые зависимости должны существовать до создания любых импортируемых целей. Для получения зависимостей используйте find_dependency() из модуля CMakeFindDependencyMacro, а не вызов find_package() из файла конфигурации пакета, если только зависимость не поддерживает компоненты пакета. Если вы вызываете find_package() для приведения зависимости, убедитесь, что опции QUIET и REQUIRED правильно переданы вызову find_package() зависимости. Также используйте соответствующие переменные для определения успеха/неудачи и для сообщения об ошибке исходной команде find_package() вместо вызова message(FATAL_ERROR ...) или аналогичного.

Предпочтительно использовать модуль InstallRequiredSystemLibraries для обработки установки зависимостей времени выполнения компилятора. Это позволит проекту избежать дублирования всей сложной логики поиска соответствующих файлов для различных версий Visual Studio, SDK, выбора инструментария и т.д. Если

поддержка компиляторов Intel важна, разберитесь в различных библиотеках, которые этот модуль устанавливает по умолчанию, и решите, все ли эти библиотеки нужны. Проекты, использующие OpenMP, скорее всего, захотят использовать команды установки по умолчанию, а не определять свои собственные, чтобы не определять необходимые библиотеки вручную.

Глава 26. Упаковка

Создание пакетов релизов - это область, в которой разработчики часто чувствуют себя не в своей тарелке. Различные системы упаковки, различия между платформами и соглашения могут представлять собой очень крутую кривую обучения для тех, кто хочет овладеть искусством создания надежных, хорошо представленных пакетов на различных платформах. Каждая система управления пакетами неизменно использует свою собственную уникальную форму спецификации входных данных о том, что содержит каждый пакет, как он должен быть установлен, как компоненты пакета связаны друг с другом, как интегрироваться с операционной системой и так далее. Различия между платформами и даже между различными дистрибутивами одной и той же платформы не всегда очевидны и часто становятся известны только после возникновения проблем из-за непредвиденного поведения или ограничений (ограничения длины пути в Windows и различные соглашения в Linux для имен каталогов системных библиотек являются отличными примерами этого).

Несмотря на все эти различия, существует значительная степень общности в используемых концепциях упаковки. Хотя каждая система или платформа может быть реализована по-разному, большая часть их функциональности по упаковке может быть описана достаточно общим образом. CMake и CPack используют это преимущество и предоставляют хорошо определенный интерфейс для указания этих общих аспектов, которые затем преобразуются в необходимые входные файлы и команды пакетной системы для создания пакетов в различных форматах. Это обеспечивает гораздо более короткую кривую обучения для разработчиков, что приводит к относительно быстрому пути к созданию пакетов для всех интересующих платформ.

CMake и CPack не только абстрагируют общие аспекты упаковки, но и упрощают использование многих специфических функций упаковщика. Предоставляя более простой интерфейс к этим функциям, CMake и CPack позволяют разработчикам использовать более продвинутые функции упаковки более привычным способом. По большей части это делается путем установки нескольких соответствующих переменных или вызова функций с соответствующими аргументами, все из которых определены в документации к модулю CMake каждого упаковщика.

Упаковка CPack реализуется внутри как одна или несколько установок в область постановки, которая затем используется для создания конечного пакета (пакетов). Эти установки управляются вызовами команды install(), которые были подробно рассмотрены в предыдущей главе. Теперь в этой главе представлена вторая половина этого процесса, описывающая переменные и команды, которые определяют метаданные пакета и конфигурацию самих пакетов.

26.1. Основы упаковки

Настройка и выполнение упаковки выполняется аналогично тестированию. Инструмент командной строки cpack считывает входной файл и создает соответствующий пакет(ы) на основе содержимого этого файла. Если в командной строке явно не указан входной файл, cpack будет использовать CPackConfig.cmake в текущем каталоге. Этот входной файл чаще всего создается CMake путем включения модуля CPack, подобно тому, как включение модуля CTest создает входной файл для ctest. Проекты могут настраивать содержимое генерируемого входного файла упаковки с помощью переменных и команд CMake.

Модуль CPack позволяет создавать несколько форматов пакетов по умолчанию в зависимости от целевой платформы. Набор создаваемых форматов пакетов можно переопределить в командной строке cpack с помощью опции -G. Если необходимо создать несколько форматов, их можно указать в виде списка, разделенного точкой с запятой, как показано ниже:

```
cpack -G "ZIP;RPM"
```

Если проект CMake был настроен на использование генератора с несколькими конфигурациями, например Xcode или Visual Studio, cpack должен знать, исполняемые файлы какой конфигурации ему следует упаковать. Это делается с помощью опции -C для cpack (опция -C игнорируется генераторами CMake с одной конфигурацией):

```
cpack -C Release
```

Команда cpack поддерживает несколько других опций, но -G и -C являются двумя наиболее часто используемыми. Большинство других деталей обычно предоставляются через входной файл. Отчасти это связано с тем, что вместо прямого вызова cpack разработчики могут создать цель сборки пакета, которая сначала соберет цель all по умолчанию, а затем вызовет cpack с минимальными опциями. Поэтому для проекта удобнее убедиться, что входной файл cpack определяет все необходимые параметры. CMake автоматически создаст цель сборки пакета, если в вершине дерева сборки содержится файл CPackConfig.cmake.

Самый простой способ создать входной файл cpack - это включить модуль CPack, что можно сделать только один раз для всего проекта CMake. Это включение обычно выполняется в конце или около конца файла CMakeLists.txt верхнего уровня, либо напрямую, либо через CMakeLists.txt подкаталога. Условное включение в зависимости от того, есть ли у проекта родитель, также гарантирует, что проект попытается определить упаковку, только если он является проектом верхнего уровня. Например:

```
cmake_minimum_required(VERSION 3.0)
project(MyProj)

# ...Define targets, add subdirectories, etc...

# End of the CMakeLists.txt file
if(CMAKE_SOURCE_DIR STREQUAL CMAKE_CURRENT_SOURCE_DIR)
    add_subdirectory(packaging)   # include(CPack) will happen in here
endif()
```

В момент включения модуля CPack файл CPackConfig.cmake записывается в вершину дерева сборки (т.е. в CMAKE_BINARY_DIR). Поскольку CMake проверяет наличие этого файла только после завершения обработки файлов CMakeLists.txt, он всегда будет создавать цель сборки пакета, если в него включен модуль CPack.

Хотя для большинства аспектов конфигурации упаковки предусмотрены значения по умолчанию, эти значения не всегда подходят. Большинство проектов захотят установить некоторые основные детали перед включением модуля CPack, чтобы обеспечить лучшие альтернативы. В частности, рекомендуется явно установить следующие переменные перед вызовом include(CPack):

CPACK_PACKAGE_NAME

> Имя пакета - это один из наиболее фундаментальных фрагментов метаданных. Оно используется как часть имени файла по умолчанию для пакетов, может появляться в различных местах в установщиках пользовательского интерфейса и, скорее всего, будет именем, которое конечные пользователи будут использовать для обращения к проекту. В идеале оно не должно содержать пробелов, поскольку в некоторых контекстах пробелы заменяются другими символами. Если эта переменная не задана явно, по умолчанию используется CMAKE_PROJECT_NAME.

CPACK_PACKAGE_DESCRIPTION_SUMMARY

Эта переменная содержит короткое предложение о проекте, состоящее не более чем из нескольких слов. Она должна подходить для отображения в списках пакетов, где место ограничено, и давать конечным пользователям представление о том, что представляет собой пакет. Она может быть показана пользователю и в других ситуациях и используется только в информационных целях. Начиная с CMake 3.9, значение по умолчанию берется из CMAKE_PROJECT_DESCRIPTION, тогда как в более ранних версиях CMake по умолчанию используется пустая строка.

CPACK_PACKAGE_VENDOR

Поставщик обычно используется только для информации, а не для влияния на структуру или поведение пакета, но он полезен для конечных пользователей, если установлен соответствующим образом. Значение по умолчанию Humanity, как правило, не подходит ни для чего, кроме как для использования в качестве заполнителя, пока оно не будет установлено должным образом. Предпочтительно использовать реальное название компании или организации, а не доменное имя.

CPACK_PACKAGE_VERSION_MAJOR, CPACK_PACKAGE_VERSION_MINOR, CPACK_PACKAGE_VERSION_PATCH

Они используются для построения общей версии пакета и могут появляться как часть имен файлов пакета, в метаданных пакета и в пользовательских интерфейсах программы установки. Информация о версии - это критическая часть упаковки, которую проекты всегда должны задавать явно. Значения по умолчанию 0, 1 и 1 соответственно полезны только в качестве временных меток и никогда не должны использоваться для официальных релизных пакетов. Удобным шаблоном является использование информации о версии, предоставляемой команде project():

```
set(CPACK_PACKAGE_VERSION_MAJOR ${PROJECT_VERSION_MAJOR})
set(CPACK_PACKAGE_VERSION_MINOR ${PROJECT_VERSION_MINOR})
set(CPACK_PACKAGE_VERSION_PATCH ${PROJECT_VERSION_PATCH})
```

cpack автоматически заполнит CPACK_PACKAGE_VERSION на основе этих трех переменных, но это произойдет только при запуске cpack, поэтому CPACK_PACKAGE_VERSION еще не будет заполнена во время обработки CMake. Начиная с CMake 3.12, значения по умолчанию для этих переменных берутся из недавно добавленных CMAKE_PROJECT_VERSION_MAJOR, CMAKE_PROJECT_VERSION_MINOR и CMAKE_PROJECT_VERSION_PATCH. Эти переменные задаются параметром VERSION команды project() в файле верхнего уровня CMakeLists.txt, поэтому они с гораздо большей вероятностью обеспечивают разумные значения по умолчанию, чем довольно произвольные до версии 3.12 значения 0, 1 и 1. Однако, полагаясь на эти переменные для обеспечения значений по умолчанию, предполагается, что проект всегда является проектом верхнего уровня, что может быть не всегда так. Поэтому безопаснее всегда явно устанавливать их в то значение, которое действительно нужно проекту.

CPACK_PACKAGE_INSTALL_DIRECTORY

Некоторые упаковщики добавляют его к базовой точке установки, чтобы создать каталог для конкретного пакета. Значение по умолчанию может варьироваться, но может включать имя пакета и версию. Наличие номера версии в значении по умолчанию часто нежелательно, например, для программ установки, которые могут обновлять проект на месте. Чтобы обеспечить лучшее поведение по умолчанию, проекты могут захотеть установить это значение таким же, как CPACK_PACKAGE_NAME.

CPACK_VERBATIM_VARIABLES

Эта переменная всегда должна быть явно установлена в true. Она гарантирует, что все содержимое, записываемое в конфигурационный файл cpack, будет правильно экранировано. Значение по умолчанию

false используется только для сохранения обратной совместимости с предыдущими версиями CMake, но старое поведение может привести к неправильному оформлению конфигурационного файла и не должно использоваться.

Часто для улучшения работы конечного пользователя, особенно в установщиках пользовательского интерфейса, необходимо установить больше переменных:

CPACK_PACKAGE_DESCRIPTION_FILE

Это имя текстового файла, содержащего более подробное описание проекта. Содержимое файла может быть показано на вводных экранах программы установки или добавлено в метаданные пакета. Всегда используйте абсолютный путь к файлу. В качестве альтернативы описание может быть предоставлено непосредственно как содержимое переменной с именем CPACK_PACKAGE_DESCRIPTION. Хотя этот способ не был документирован для CMake 3.11 и более ранних версий, он поддерживался даже в ранних версиях CMake.

CPACK_RESOURCE_FILE_WELCOME

Некоторые программы установки показывают приветственное сообщение на начальном экране. Эта переменная определяет имя файла, содержимое которого должно быть показано в таких случаях. Если она не задана, то для тех программ установки, которые показывают приветственное сообщение, CPack предоставляет значение по умолчанию, которое действует как заполнитель, но это относительно плохая замена, не подходящая для официальных релизов. Проекты должны всегда устанавливать это значение, если распространяют программу установки, показывающую экран приветствия. Всегда используйте абсолютный путь к файлу.

CPACK_RESOURCE_FILE_LICENSE

Большинство программ установки пользовательского интерфейса представляют пользователю страницу лицензии и могут попросить его принять лицензию перед продолжением работы. Текст, отображаемый для лицензии, берется из файла, названного этой переменной. Если переменная не задана, используется общий текст-заполнитель, но проектам настоятельно рекомендуется предоставить свои собственные, более подходящие детали лицензии. Всегда используйте абсолютный путь к файлу.

CPACK_RESOURCE_FILE_README

Некоторые программы установки пользовательского интерфейса предоставляют отдельную страницу, показывающую содержимое файла, названного этой переменной. Она служит для того, чтобы дать пользователю некоторую информацию, прежде чем он продолжит установку, и по умолчанию содержит общий, но обычно неподходящий текст. Проекты должны предпочитать указывать файл с более подходящим содержимым через эту переменную, если они собираются создавать программы установки, показывающие такие страницы. Всегда используйте абсолютный путь к файлу.

CPACK_PACKAGE_ICON

Эта переменная также может быть обычно задана, но имейте в виду, что большинство генераторов пакетов имеют свои собственные требования к формату и использованию пиктограмм в пакете и связанных с ним местах. Некоторые генераторы игнорируют эту переменную, другие используют ее по-разному.

Пример, соответствующий вышеуказанным рекомендациям, может выглядеть следующим образом:

```
set(CPACK_PACKAGE_NAME                MyProj)
set(CPACK_PACKAGE_VENDOR              MyCompany)
set(CPACK_PACKAGE_DESCRIPTION_SUMMARY "CPack example project")
set(CPACK_PACKAGE_INSTALL_DIRECTORY   ${CPACK_PACKAGE_NAME})
set(CPACK_PACKAGE_VERSION_MAJOR       ${PROJECT_VERSION_MAJOR})
set(CPACK_PACKAGE_VERSION_MINOR       ${PROJECT_VERSION_MINOR})
set(CPACK_PACKAGE_VERSION_PATCH       ${PROJECT_VERSION_PATCH})
set(CPACK_VERBATIM_VARIABLES          YES)
set(CPACK_PACKAGE_DESCRIPTION_FILE    ${CMAKE_CURRENT_LIST_DIR}/Description.txt)
set(CPACK_RESOURCE_FILE_WELCOME       ${CMAKE_CURRENT_LIST_DIR}/Welcome.txt)
set(CPACK_RESOURCE_FILE_LICENSE       ${CMAKE_CURRENT_LIST_DIR}/License.txt)
set(CPACK_RESOURCE_FILE_README        ${CMAKE_CURRENT_LIST_DIR}/Readme.txt)
include(CPack)
```

Чтобы облегчить запуск cpack без аргументов и использование цели сборки пакета, переменная CPACK_GENERATOR должна быть установлена на желаемые форматы пакетов. Если она не установлена, будет использоваться довольно консервативный набор генераторов по умолчанию. Поскольку не все форматы поддерживаются или подходят для всех платформ, установка этой переменной требует, чтобы логика указывала только те форматы, которые имеют смысл. В следующем примере выбирается один общий формат архива и один собственный формат пакета для целевой платформы (если она определена):

```
if(WIN32)
    set(CPACK_GENERATOR ZIP WIX)
elseif(APPLE)
    set(CPACK_GENERATOR TGZ productbuild)
elseif(CMAKE_SYSTEM_NAME STREQUAL "Linux")
    set(CPACK_GENERATOR TGZ RPM)
else()
    set(CPACK_GENERATOR TGZ)
endif()
```

Модуль CPack также определяет необходимые детали, позволяющие создать исходный пакет. Он создает файл CPackSourceConfig.cmake, который можно использовать вместо CPackConfig.cmake, а когда проект настроен на использование Makefile или Ninja-генератора, определяется также цель сборки package_source. Создание исходного пакета относительно простое, для этого достаточно выполнить две следующие команды.

```
# All build generators
cpack -G TGZ --config CPackSourceConfig.cmake

# Makefile and Ninja build generators only
cmake --build . --target package_source
```

Исходный пакет содержит все дерево каталогов исходного кода. Переменная CPACK_SOURCE_IGNORE_FILES может быть использована для фильтрации частей дерева исходного кода, содержащего список регулярных выражений, с которыми будет сравниваться каждый полный путь к файлу. Все совпадающие файлы будут исключены из исходного пакета. Значение этой переменной по умолчанию игнорирует каталоги репозитория, такие как .git, .svn и т.д., а также некоторые обычные временные файлы. Если проект переопределяет CPACK_SOURCE_IGNORE_FILES, ему необходимо убедиться, что он также указал все подобные шаблоны. Чтобы избежать проблем с экранированием и цитированием в регулярных выражениях, настоятельно рекомендуется установить CPACK_VERBATIM_VARIABLES в true.

```
set(CPACK_VERBATIM_VARIABLES YES)
set(CPACK_SOURCE_IGNORE_FILES
    /\\.git/
    \\.swp
    \\.orig
    /CMakeLists\\.txt\\.user
    /privateDir/
)
```

26.2. Компоненты

Если проект не определяет компоненты ни в одной из своих команд install(), то все генераторы пакетов будут создавать единый монолитный пакет, содержащий все установленное содержимое. Когда проект определяет компоненты, это обеспечивает большую гибкость при упаковке. Между компонентами также могут быть указаны связи, что позволяет определять иерархические структуры компонентов и устанавливать зависимости между ними во время установки. Каждый генератор пакетов использует эти детали компонентов по-разному, некоторые создают отдельные пакеты для разных компонентов, другие представляют выбираемые пользователем компоненты в единой программе установки с пользовательским интерфейсом. Некоторые программы установки даже поддерживают загрузку отдельных компонентов по требованию во время установки.

В предыдущей главе было показано, как определять компоненты в командах install(). Эти команды только назначают содержимое компонентам, они не определяют никаких других деталей компонента. Отношения между компонентами задаются с помощью команд из раздела
модуль CPackComponents, который автоматически включается как часть включения модуля CPack. Эти команды также предоставляют дополнительные метаданные для компонентов, которые некоторые программы установки используют для представления информации пользователю во время установки.

Наиболее важной командой из модуля CPackComponents является cpack_add_component(), которая описывает один компонент:

```
cpack_add_component(componentName
    [DISPLAY_NAME name]
    [DESCRIPTION description]
    [DEPENDS comp1 [comp2...] ]
    [GROUP group]
    [REQUIRED | DISABLED]
    [HIDDEN]
    [INSTALL_TYPES type1 [type2...] ]
    [DOWNLOADED]
    [ARCHIVE_FILE archiveFileName]
    [PLIST plistFileName]
)
```

Хотя все ключевые слова необязательны, DISPLAY_NAME и DESCRIPTION должны, по крайней мере, быть указаны, чтобы пользователю были представлены значимые детали во время установки и чтобы программы установки без пользовательского интерфейса имели достаточно метаданных, чтобы пользователи могли понять, для чего предназначен пакет. Если компонент должен быть установлен только при условии установки одного или нескольких других компонентов, эти компоненты должны быть перечислены с помощью опции DEPENDS. Обратите внимание, что не все типы пакетов полностью обеспечивают соблюдение этих

зависимостей. Компонент может быть отнесен к определенной группе с помощью опции GROUP, которая может быть описана с помощью команды cpack_add_component_group() (подробнее об этом ниже).

Если компонент должен быть установлен всегда, следует указать ключевое слово REQUIRED. Тогда пользователь не сможет отключить этот компонент через пользовательский интерфейс программы установки. Без этого ключевого слова компонент может быть включен или отключен пользователем, при этом начальное состояние по умолчанию будет включенным. Чтобы изменить это состояние по умолчанию на отключенное, добавьте ключевое слово DISABLED. Независимо от того, требуется компонент или нет, его также можно скрыть из пользовательского интерфейса программы установки, добавив ключевое слово HIDDEN. Не требуемый, но скрытый компонент обычно также отключается, и программа установки будет устанавливать его только в том случае, если от него зависит другой включенный компонент.

Остальные опции имеют более специализированные эффекты, которые применяются только к небольшому числу генераторов пакетов. Тип установки - это предустановленный набор компонентов, который может быть использован для упрощения выбора, который пользователь должен сделать во время установки. Компонент может быть отнесен к любому количеству типов установки с помощью опции INSTALL_TYPES, где каждый тип - это имя, которое определяется отдельно командой cpack_add_install_type() следующим образом:

```
cpack_add_install_type(typeName [DISPLAY_NAME uiName])
```

Опцию DISPLAY_NAME можно опустить, если typeName уже достаточно описательно, но для типов установки, которые должны отображаться с помощью нескольких слов, необходимо использовать DISPLAY_NAME, а uiName будет строкой в кавычках. Предопределённых типов установки не существует, но часто встречаются пакеты, предоставляющие типы установки с именами типа Full, Minimal или Default. Из активно поддерживаемых генераторов пакетов, предоставляемых CMake, только NSIS поддерживает функцию типов установки.

Для тех генераторов, которые поддерживают загружаемые компоненты, добавление ключевого слова DOWNLOADED в cpack_add_component() заставляет компонент загружаться по требованию, а не включаться в пакет напрямую. Опция ARCHIVE_FILE может быть использована для настройки имени файла загружаемого компонента. Единственным активно поддерживаемым генератором, предоставляемым CMake, который поддерживает загружаемые компоненты, является IFW, поэтому обсуждение этой возможности отложено до Раздела 26.4.2, "Qt Installer Framework (IFW)". Аналогично, опция PLIST (доступная только в CMake 3.9 или более поздней версии) используется исключительно генератором пакетов productbuild (см. раздел 26.4.6, "productbuild").

Если компоненты не определены с деталями GROUP, компоненты будут действовать как простой плоский список в большинстве программ установки пользовательского интерфейса. Когда используется группировка, это позволяет определить произвольно глубокую иерархическую структуру, где группы могут содержать компоненты и другие группы. Группа определяется с помощью следующей команды из модуля CPackComponents:

```
cpack_add_component_group(groupName
    [DISPLAY_NAME name]
    [DESCRIPTION description]
    [PARENT_GROUP parent]
    [EXPANDED]
    [BOLD_TITLE]
)
```

Эта команда может появляться до или после вызовов cpack_add_component(), которые ссылаются на groupName. Опции DISPLAY_NAME и DESCRIPTION служат тем же целям, что и их аналоги в команде cpack_add_component(). PARENT_GROUP - это эквивалент опции GROUP, позволяющий помещать группу под другую группу для поддержки произвольной иерархии групп. Когда указано ключевое слово EXPANDED, группа будет изначально расширена в пользовательском интерфейсе программы установки, а наличие ключевого слова BOLD_TITLE заставит эту группу отображаться жирным шрифтом.

Имена компонентов в идеале должны быть специфичными для проекта, чтобы иерархические структуры проекта могли эффективно выбирать, какие компоненты упаковывать и как представлять их в инсталляторах (или, в случае не-UI инсталляторов, как структурировать специфичные для компонентов пакеты). Имена групп менее ограничительны, поскольку они могут содержать компоненты и группы из разных проектов. Имя группы не может совпадать с именем любого компонента.

Действие cpack_add_component() и cpack_add_component_group() заключается в определении ряда переменных, специфичных для компонента, в текущей области видимости. В документации CPackComponent перечислены некоторые из этих переменных и предлагается установить переменные напрямую, но это не рекомендуется. Команды предлагают более надежный и более читабельный способ определения деталей компонента и группы, поэтому их следует предпочесть. Они также должны вызываться в той же области видимости, что и вызов include(CPack), в идеале - сразу после него. Технически это ограничение не так строго, но определение деталей компонента в другой области видимости может быть более хрупким.

Пример должен помочь закрепить некоторые из приведенных выше концепций и обсуждений.

```
set(CPACK_PACKAGE_NAME ...)
# ... set other variables as per earlier example

include(CPack)

cpack_add_component(MyProj_Runtime
    DISPLAY_NAME  Runtime
    DESCRIPTION   "Shared libraries and executables"
    REQUIRED
    INSTALL_TYPES Full Developer Minimal
)
cpack_add_component(MyProj_Development
    DISPLAY_NAME  "Developer pre-requisites"
    DESCRIPTION   "Static libraries and headers needed for building apps"
    DEPENDS       MyProj_Runtime
    GROUP         MyProj_SDK
    INSTALL_TYPES Full Developer
)
cpack_add_component(MyProj_Samples
    DISPLAY_NAME  "Code samples"
    GROUP         MyProj_DevHelp
    INSTALL_TYPES Full Developer
    DISABLED
)
cpack_add_component(MyProj_ApiDocs
    DISPLAY_NAME  "API documentation"
    GROUP         MyProj_DevHelp
    INSTALL_TYPES Full Developer
    DISABLED
)
cpack_add_component_group(MyProj_SDK
    DISPLAY_NAME  SDK
    DESCRIPTION   "Developer tools, libraries, etc."
)
cpack_add_component_group(MyProj_DevHelp
    DISPLAY_NAME  Documentation
    DESCRIPTION   "Code samples and API docs"
    PARENT_GROUP  MyProj_SDK
)
cpack_add_install_type(Full)
cpack_add_install_type(Minimal)
cpack_add_install_type(Developer DISPLAY_NAME "SDK Development")
```

Генераторам проектов может быть предложено обрабатывать компоненты одним из трех способов, выбор контролируется переменной CPACK_COMPONENTS_GROUPING, которая может быть установлена в одно из следующих значений:

ALL_COMPONENTS_IN_ONE

Будет создан один пакет со всеми запрошенными компонентами. Структура компонентов и групп игнорируется.

ONE_PER_GROUP

Каждая группа компонентов верхнего уровня должна создать свой пакет. Те компоненты, которые не входят в группу, также создают свой собственный пакет. Если CPACK_COMPONENTS_GROUPING не задан, то по умолчанию это является желательным вариантом, но для некоторых программ установки пользовательского интерфейса это скрывает компоненты, которые проекты могут предпочесть показать.

IGNORE

Каждый компонент создает свой собственный пакет независимо от групп компонентов. Этот параметр может быть более подходящим для некоторых программ установки пользовательского интерфейса, чтобы гарантировать, что никакие компоненты не будут скрыты, если они явно не настроены.

Еще две переменные также влияют на то, как генераторы интерпретируют компоненты. Если CPACK_MONOLITHIC_INSTALL имеет значение true, компоненты полностью отключаются, а все компоненты устанавливаются и собираются в единый пакет. Это довольно жестокий переключатель, поэтому тщательно протестируйте результаты на всех соответствующих платформах, уделяя особое внимание поиску любых неожиданных файлов. По причинам, связанным с наследием, каждый генератор также имеет свою собственную настройку того, поддерживаются ли компоненты по умолчанию. Эта настройка может быть переопределена для каждого генератора с помощью переменной CPACK_<GENNAME>_COMPONENT_INSTALL, которая может быть установлена в true или false в зависимости от необходимости.

При выполнении установки на основе компонентов проекты не обязаны включать все компоненты в конечный пакет(ы). Набор компонентов, которые будут включены, контролируется переменной CPACK_COMPONENTS_ALL, которая должна быть установлена перед вызовом include(CPack). Если переменная не установлена, cpack упаковывает все компоненты, но проект может явно установить эту переменную, чтобы перечислить только те компоненты, которые он хочет упаковать. Например, если проект хочет контролировать, должны ли быть упакованы документация и примеры кода, это можно сделать следующим образом:

```
if(NOT MYPROJ_PACKAGE_HELP)
    set(CPACK_COMPONENTS_ALL
        MyProj_Runtime
        MyProj_Development
    )
endif()
include(CPack)
```

Вместо явного перечисления всех компонентов, которые необходимо упаковать, проект может захотеть установить все компоненты, кроме нескольких определенных. Полный набор компонентов доступен только для чтения в псевдосвойстве COMPONENTS, которое можно получить только с помощью команды get_cmake_property(). Проект может начать с этого списка компонентов, а затем удалить ненужные записи.

```
if(NOT MYPROJ_PACKAGE_HELP)
    get_cmake_property(CPACK_COMPONENTS_ALL COMPONENTS)
    list(REMOVE_ITEM CPACK_COMPONENTS_ALL
        MyProj_Samples
        MyProj_ApiDocs
    )
endif()
include(CPack)
```

Выбор того, какой набор компонентов установить и как эти компоненты должны обрабатываться, сначала может показаться немного сложным. На практике основная область, вызывающая затруднения, - это понимание того, как каждый генератор пакетов обрабатывает различные значения CPACK_COMPONENTS_GROUPING. В последующих разделах этой главы объясняется поведение каждого типа генераторов, но некоторые быстрые эксперименты на тестовом проекте часто могут быть столь же поучительными для понимания влияния различных настроек.

26.3. Мультиконфигурационные пакеты

CPack по своей сути ориентирован на создание пакетов для одной конфигурации сборки. В большинстве случаев пакеты создаются для типа сборки Release, но для таких вещей, как SDK-проекты, может быть желательно включить как отладочные, так и релизные версии библиотек. Чтобы иметь возможность собирать и упаковывать обе конфигурации в один набор пакетов, требуется немного больше работы.

CPack предоставляет расширенную переменную CPACK_INSTALL_CMAKE_PROJECTS, которая может быть использована для включения нескольких деревьев сборки в один прогон упаковки. Предполагается, что она будет содержать один или более квадруплов, где каждый квадрупл состоит из:

- Каталог сборки.
- Имя проекта (важно только для генераторов с несколькими конфигурациями).
- Компонент для установки. Специальное значение ALL означает установку компонентов, перечисленных в поле

  переменная CMAKE_COMPONENTS_ALL. Другие значения требуют определения аналогичной переменной CMAKE_COMPONENTS_XXX, которая содержит только одно имя компонента. Например, если устанавливаемый компонент называется Runtime, то необходимо определить переменную CMAKE_COMPONENTS_RUNTIME, которая будет иметь значение Runtime.
- Относительное расположение в пакете для установки. Единственным безопасным значением для этого параметра является один прямой слэш (/) из-за того, как его используют различные генераторы пакетов.

В проекте могут быть определены наборы четверок, одна из которых предназначена для сборки релиза, а остальные - для отладочной сборки. Каталог сборки для релизной сборки может быть просто CMAKE_BINARY_DIR, но для отладочной сборки должен быть создан и собран второй отдельный каталог сборки.

В отладочные четверки нужно будет добавить только те компоненты, которые отличаются между двумя конфигурациями сборки, но независимо от того, используется ли компонент ALL по умолчанию или специальные компоненты, необходимо соблюдать особую осторожность, чтобы установленные файлы неожиданно не перезаписали друг друга. Перечисление компонента выпуска последним гарантирует, что все файлы, имеющие одинаковые имена и место установки, при упаковке попадут в версию выпуска.

```
set(CPACK_COMPONENTS_MYPROJ_RUNTIME     MyProj_Runtime)
set(CPACK_COMPONENTS_MYPROJ_DEVELOPMENT MyProj_Development)

unset(CPACK_INSTALL_CMAKE_PROJECTS)
if(MYPROJ_DEBUG_BUILD_DIR)
    list(APPEND CPACK_INSTALL_CMAKE_PROJECTS
        ${MYPROJ_DEBUG_BUILD_DIR} ${CMAKE_PROJECT_NAME} MyProj_Runtime     /
        ${MYPROJ_DEBUG_BUILD_DIR} ${CMAKE_PROJECT_NAME} MyProj_Development /
    )
endif()
list(APPEND CPACK_INSTALL_CMAKE_PROJECTS
    ${CMAKE_BINARY_DIR} ${CMAKE_PROJECT_NAME} ALL /
)
include(CPack)
```

При использовании генераторов мультиконфигурации, таких как Xcode или Visual Studio, каталог MYPROJ_DEBUG_BUILD_DIR в приведенном выше примере должен быть настроен на поддержку только типа сборки Debug, а не обычного набора по умолчанию. Это единственный способ заставить его установить отладочную сборку вместо той же конфигурации, что и в основном проекте. При запуске cmake в каталоге отладочной сборки явно установите кэш-переменную CMAKE_CONFIGURATION_TYPES в значение Debug, чтобы получить необходимую компоновку.

Хотя возможно использовать только один каталог сборки для генераторов с несколькими конфигурациями, техника для этого более хрупкая и сложная. В отличие от этого, описанная выше техника работает для всех типов генераторов сборок и пакетов. Более того, она может быть расширена для включения сборок для различных архитектур или даже совершенно отдельных проектов в один единый пакет или набор пакетов.

26.4. Генераторы пакетов

CPack поддерживает создание пакетов в различных форматах, каждый из которых можно отнести к одной из следующих категорий:

Простые архивы

Архивы могут иметь различные форматы, такие как zip, tarball, bz2 и так далее. Это самый простой из всех форматов пакетов, поскольку они представляют собой просто архив файлов, который пользователь должен распаковать где-то в своей файловой системе. Они являются наиболее широко поддерживаемыми из всех форматов пакетов, и с ними проще всего работать, когда конечный пользователь хочет иметь несколько различных версий проекта, доступных или установленных одновременно.

Установщики пользовательского интерфейса

Как правило, они имеют глубокую интеграцию с целевой платформой, которую поддерживают, предоставляя такие возможности, как добавление и удаление компонентов после установки, интеграция с меню рабочего стола и так далее. Они обычно предоставляют пользователю некоторые средства выбора компонентов для установки и обычно очень интуитивно понятны, поэтому начинающие пользователи предпочитают их. CMake поддерживает программы установки NSIS и WIX в Windows, DragNDrop (т.е. DMG) и productbuild в Mac, а также Qt Installer Framework (IFW) в Windows, Mac и Linux. На Mac некоторые старые типы инсталляторов все еще поддерживаются, но их следует считать фактически устаревшими, и в последующих разделах они упоминаются лишь вскользь.

Пакеты, не относящиеся к UI

Они нацелены на конкретный менеджер пакетов. RPM и DEB очень популярны в Linux, пакеты FreeBSD и Cygwin также поддерживаются для соответствующих платформ.

Пакеты для конкретного продукта

В CMake 3.12 добавлена начальная поддержка формата пакетов NuGet для .NET.

Независимо от того, какой тип пакета должен быть сгенерирован, в каждом случае используется один и тот же файл CPackConfig.cmake. Обычно это не представляет проблемы, так как конфигурация для конкретного генератора обычно осуществляется с помощью переменных для конкретного генератора, где это необходимо. Если определенная логика должна быть добавлена только для конкретного генератора и существующих переменных, предлагаемых CMake и CPack, недостаточно, переменная CPACK_PROJECT_CONFIG_FILE может быть установлена в имя файла, который будет включаться один раз для каждого вызываемого генератора пакетов. При каждом чтении переменная CPACK_GENERATOR будет содержать имя обрабатываемого генератора, а не весь список генераторов. Это позволяет этому файлу переопределять настройки, сделанные в CPackConfig.cmake, только для тех генераторов, которые в этом нуждаются. Полный запуск cpack в общих чертах повторяет шаги следующего псевдокода:

```
include(CPackConfig.cmake)

function(generate CPACK_GENERATOR)
    # CPACK_GENERATOR is a single generator local to this function scope
    if(CPACK_PROJECT_CONFIG_FILE)
        include(${CPACK_PROJECT_CONFIG_FILE})
    endif()

    # ...invoke package generator
endfunction()

# Here CPACK_GENERATOR is the list of generators to be processed,
# as set by CPackConfig.cmake or on the cpack command line
foreach(generator IN LISTS CPACK_GENERATOR)
    generate(${generator})
endforeach()
```

Примером того, как это может быть полезно, является установка CPACK_PACKAGE_ICON в значение, специфичное для генератора, поскольку разные генераторы ожидают, что эта иконка будет в разных форматах, и поэтому имя файла должно быть специфичным для генератора.

В оставшейся части этой главы рассматривается каждый из активно поддерживаемых генераторов пакетов, предоставляемых CMake/CPack.

26.4.1. Простые архивы

CPack поддерживает создание архивов в нескольких различных форматах. Наиболее широко поддерживаются ZIP и TGZ, первый распространен на платформах Windows, а второй создает gzipped tarballs (.tar.gz или .tgz), которые поддерживаются практически везде. Другие доступные форматы архивов включают TBZ2 (.tar.bz2), TXZ (.tar.xz), TZ (.tar.Z) и 7Z (архивы 7zip, .7z). Для максимальной переносимости, как правило, следует предпочесть ZIP и TGZ, но некоторые из других форматов могут создавать архивы меньшего размера и могут подойти для платформ, где эти форматы обычно поддерживаются.

Формат самораспаковывающегося архива также поддерживается cpack. Его можно запросить с помощью генератора с именем STGZ, который создает сценарий оболочки Unix с архивом, встроенным в конец этого сценария. Это можно рассматривать как разновидность консольного UI-инсталлятора, но на практике он предлагает только очень базовую функциональность, и пользователи могут предпочесть простой архив, который они могут распаковать самостоятельно.

По унаследованным причинам в генераторах архивов по умолчанию отключены компоненты. Чтобы включить архивы, основанные на компонентах, CPACK_ARCHIVE_COMPONENT_INSTALL должен быть установлен в true, а затем
CPACK_COMPONENTS_GROUPING определит набор архивных файлов, которые будут сгенерированы.

При установке без компонентов имя конечного файла пакета можно контролировать с помощью переменной CPACK_ARCHIVE_FILE_NAME. При компонентной установке имя пакета каждого компонента контролируется переменной CPACK_ARCHIVE_<COMP>_FILE_NAME, где <COMP> - это имя компонента или группы, написанное с заглавной буквы. К указанному имени файла будет добавлено соответствующее расширение архива (т.е. .tar.gz, .zip и т.д.).

Общим соглашением для архивных файлов является то, что верхний уровень структуры извлеченного каталога должен быть таким же, как имя архивного файла без расширения файла (т.е. таким же, как CPACK_PACKAGE_FILE_NAME). Для некомпонентных установок это уже поведение по умолчанию для генераторов архивов, но для многокомпонентных пакетов по умолчанию не используется каталог верхнего уровня. Проекты могут принудительно использовать общий каталог верхнего уровня для архивов компонентов, установив CPACK_COMPONENT_INCLUDE_TOPLEVEL_DIRECTORY в true. Поскольку эта переменная является общей для всех генераторов пакетов, наиболее подходящим способом будет переопределение для конкретного генератора:

CMakeLists.txt

```
set(CPACK_PROJECT_CONFIG_FILE
    ${CMAKE_CURRENT_LIST_DIR}/cpackGeneratorOverrides.cmake
)
```

cpackGeneratorOverrides.cmake

```
if(CPACK_GENERATOR MATCHES "^(7Z|TBZ2|TGZ|TXZ|TZ|ZIP)$")
    set(CPACK_COMPONENT_INCLUDE_TOPLEVEL_DIRECTORY YES)
endif()
```

Разработчики должны учитывать, что некоторые форматы архивов, платформы и файловые системы имеют ограничения на длину имен файлов и путей. Например, POSIX.2 требует, чтобы имена файлов были не более 100 символов, а пути - не более 255 символов для расширенного формата обмена tar, в то время как старые форматы tar могут ограничивать весь путь 100 символами или менее. При распаковке архива в файловую систему eCryptFS имена файлов имеют эмпирически выведенное ограничение около 140 символов. При распаковке в Windows ограничение длины пути может составлять 260 символов, в зависимости от определенных настроек и версии ОС. Имена файлов и пути в формате UTF-8 еще больше усложняют картину и могут еще больше сократить эффективные ограничения на количество символов.

Учитывая эти ограничения, проекты должны избегать использования длинных путей и имен файлов в содержимом установленных пакетов. Эти ограничения наиболее очевидны для архивных типов пакетов, но поскольку другие неархивные форматы также используют архивы внутри и развертываются на системах с такими ограничениями, в целом следует предпочесть более короткие пути и имена файлов.

26.4.2. Qt Installer Framework (IFW)

Генератор пакетов IFW обеспечивает самую широкую поддержку платформ среди всех форматов пакетов на основе пользовательского интерфейса, предоставляемых CPack. Установщики могут быть созданы для Windows, Mac и Linux из одних и тех же деталей конфигурации, что делает его хорошим выбором, когда проект хочет иметь согласованный UI установщик на всех основных настольных платформах. Он также имеет простую в использовании локализацию имен и описаний компонентов и групп, а также широкие возможности настройки.

Значения по умолчанию для внешнего вида пользовательского интерфейса и значков программы установки часто достаточно, но некоторые проекты могут захотеть настроить некоторые аспекты для улучшения брендинга, особенно в части использования значков. Сайт
Переменная CPACK_PACKAGE_ICON игнорируется для этого генератора, который вместо этого полагается на три отдельные IFW-специфические переменные для управления иконками для различных контекстов:

• CPACK_IFW_PACKAGE_ICON (.ico for Windows, .icns for Mac, ignored for Linux)
• CPACK_IFW_PACKAGE_WINDOW_ICON (always .png)
• CPACK_IFW_PACKAGE_LOGO (preferably .png)

К сожалению, эти переменные не обрабатываются последовательно на разных платформах, поэтому может быть трудно установить их правильно. Для простоты может быть предпочтительнее установить все три параметра для одного и того же изображения, хотя и в разных форматах и/или размерах. Рекомендуется провести тестирование на каждой интересующей платформе, чтобы убедиться, что программа установки отображается так, как ожидается. В следующем примере показано, как можно задать такую конфигурацию:

```
# Define generic setup for all generator types...

# IFW-specific configuration
if(WIN32)
    set(CPACK_IFW_PACKAGE_ICON ${CMAKE_CURRENT_LIST_DIR}/Logo.ico)
elseif(APPLE)
    set(CPACK_IFW_PACKAGE_ICON ${CMAKE_CURRENT_LIST_DIR}/Logo.icns)
endif()
set(CPACK_IFW_PACKAGE_WINDOW_ICON ${CMAKE_CURRENT_LIST_DIR}/Logo.png)
set(CPACK_IFW_PACKAGE_LOGO        ${CMAKE_CURRENT_LIST_DIR}/Logo.png)

include(CPack)
include(CPackIFW)
# Define components and component groups...
```

Установка на основе компонентов включена по умолчанию для генератора IFW. Всегда создаётся один инсталлятор, но CPACK_COMPONENTS_GROUPING контролирует, какая часть иерархии компонентов будет показана пользователю:

ALL_COMPONENTS_IN_ONE

Иерархия компонентов не отображается, всегда будут установлены включенные по умолчанию компоненты.

ONE_PER_GROUP

Отображается только первый уровень групп, а также все компоненты, которые не принадлежат ни к одной группе. Подгруппы и компоненты под любой группой будут скрыты.

IGNORE

Все компоненты, которые явно не скрыты, будут показаны, независимо от того, где они находятся в иерархии группы. Скорее всего, большинство проектов захотят использовать именно этот вариант.

Компоненты и группы могут быть настроены более детально, чем это предусмотрено общими командами:

```
cpack_ifw_configure_component(componentName
    [NAME componentNameId]
    [DISPLAY_NAME displayName...]
    [DESCRIPTION description...]
    [VERSION <version>]
    [DEPENDS compId1 [compId2...] ]
    [REPLACES compId3 [compId4...] ]
    # Other options not shown
)

# The cpack_ifw_configure_component_group() command supports all
# of the above options too
```

DISPLAY_NAME и DESCRIPTION каждого компонента или группы компонентов могут иметь альтернативное содержание для различных языков и локалей. Эти две опции принимают список пар, где первое значение пары - это идентификатор языка или локали, а второе - текст для этого языка. Самое первое значение в списке может быть задано без предшествующего идентификатора языка или локали, и оно будет использовано в качестве текста по умолчанию, если ни один из языков или идентификаторов локали не соответствует текущим настройкам пользователя во время установки.

```
cpack_ifw_configure_component(MyProj_Docs
    DISPLAY_NAME Documentation
                 de Dokumentation
                 pl Dokumentacja
)

cpack_ifw_configure_component_group(MyProj_Colors
    DISPLAY_NAME en    Colors
                 en_AU Colours
    DESCRIPTION  en    "Available color palettes"
                 en_AU "Available colour palettes"
)
```

Опция VERSION позволяет указывать номера версий для каждого компонента и каждой группы. Это используется программами установки онлайн для определения доступности обновления (см. далее). Если параметр VERSION не указан, по умолчанию он принимает значение CPACK_PACKAGE_VERSION.

Опция DEPENDS аналогична той же опции в cpack_add_component(), за исключением того, что форма записей compId1... отличается. Они должны соответствовать стилю QtIFW, который представляет собой иерархическую строку, а не необработанное имя компонента. Каждый уровень сгруппированной иерархии разделяется точками, как показано в следующем примере:

```
include(CPack)
include(CPackIFW)

cpack_add_component(foo GROUP groupA)
cpack_add_component(bar GROUP groupB)

cpack_add_component_group(groupA)
cpack_add_component_group(groupB)

cpack_ifw_configure_component(bar DEPENDS groupA.foo)
```

Имя, используемое внутри программы установки для компонента, может быть переопределено с помощью опции NAME. Это имя будет использоваться для идентификации компонента в аргументах DEPENDS, а также при проверке наличия более новой версии компонента. Имя группы верхнего уровня можно задать с помощью переменной CPACK_IFW_PACKAGE_GROUP, и часто оно устанавливается на обратное доменное имя, чтобы имена компонентов не перекликались в больших многовендорных инсталляторах. Это имя группы верхнего уровня должно быть включено в список зависимостей с помощью опции DEPENDS, как показано в следующей модификации приведенного выше примера:

```
set(CPACK_IFW_PACKAGE_GROUP com.examplecompany.product)

include(CPack)
include(CPackIFW)

cpack_add_component(foo GROUP groupA)
cpack_add_component(bar GROUP groupB)

cpack_add_component_group(groupA)
cpack_add_component_group(groupB)

cpack_ifw_configure_component(bar
    DEPENDS com.examplecompany.product.groupA.foo
)
```

CPACK_IFW_PACKAGE_GROUP - это лишь один пример большого количества дополнительных переменных, которые могут быть установлены для обеспечения специфической для IFW конфигурации. Такие переменные должны быть установлены до вызова include(CPackIFW) и могут изменять внешний вид и поведение программы установки различными способами. Документация модуля CPackIFW содержит полный список всех поддерживаемых переменных и их эффектов, многие из которых имеют аналогичные настройки в собственных настройках конфигурации продукта QtIFW. Большинство из этих переменных имеют разумные значения по умолчанию и должны рассматриваться скорее как возможности для настройки, чем как вещи, которые необходимо устанавливать. Исключением являются переменные, относящиеся к имени инструмента обслуживания, установленного вместе с остальным продуктом, который позволяет пользователю изменить набор установленных компонентов или полностью удалить продукт. По умолчанию этому инструменту присваивается имя maintenancetool, но это не дает никакого указания на то, к чему относится инструмент. На некоторых платформах имя инструмента может отображаться в меню рабочего стола или приложений, и имя по умолчанию может сбить пользователей с толку. Поэтому проекты должны предоставлять более конкретное имя, что можно сделать следующим образом:

```
set(CPACK_IFW_PACKAGE_MAINTENANCE_TOOL_NAME ${PROJECT_NAME}_MaintenanceTool)
set(CPACK_IFW_PACKAGE_MAINTENANCE_TOOL_INI_FILE
    ${CPACK_IFW_PACKAGE_MAINTENANCE_TOOL_NAME}.ini)
include(CPackIFW)
```

Файл .ini используется программой установки для сохранения информации о состоянии между вызовами. Задание имени .ini файла необязательно, но предпочтительнее, чтобы имя соответствовало самой программе установки. При указанных выше настройках пользователь увидит имя, относящееся к проекту, если инструмент обслуживания появится на его рабочем столе или в меню приложений.

Важной особенностью генератора IFW является его способность создавать онлайновые инсталляторы. Некоторые или все компоненты могут быть загружены по требованию вместо того, чтобы включать их в состав программы установки. Это особенно удобно, если некоторые дополнительные компоненты имеют большой размер. Дополнительным преимуществом онлайн-установщика является то, что отдельные компоненты можно обновлять, если новые версии доступны в онлайн-репозиториях, что обеспечивает очень удобный путь обновления. Пользователи запускают инструмент обслуживания, который связывается с набором онлайн-репозиториев для определения доступных компонентов и их версий. Затем отдельные компоненты могут быть добавлены, удалены или обновлены по желанию.

Первый шаг в настройке проекта для поддержки загружаемых компонентов - указать, откуда программа установки будет их загружать. Основной репозиторий по умолчанию задается с помощью общей команды cpack_configure_downloads():

```
cpack_configure_downloads(baseUrl
    [ALL]
    [ADD_REMOVE | NO_ADD_REMOVE]
    [UPLOAD_DIRECTORY dir]
)
```

BaseUrl - это место, где программа установки будет искать загружаемые компоненты. Программа установки ожидает найти файл Updates.xml в этом месте. Если присутствует ключевое слово ALL, все компоненты считаются загружаемыми, независимо от того, были ли они явно помечены как загружаемые или нет. Это удобный способ создания полностью онлайновой программы установки без встроенных пакетов, что позволяет получить минимально возможную программу установки.

Ключевое слово ADD_REMOVE направляет программу установки сделать пакет доступным для функции Windows Add/Remove Programs, которая затем запустит инструмент обслуживания, когда пользователь решит изменить пакет через эту часть системных настроек Windows. Ключевое слово ALL подразумевает ADD_REMOVE, но указание NO_ADD_REMOVE отменяет это поведение.

Опция UPLOAD_DIRECTORY используется другими типами генераторов CPack, которые поддерживают загружаемые компоненты (хотя ни один из них активно не поддерживается), но игнорируется генератором IFW. Когда запускается cpack, он создает загружаемые пакеты в отдельном каталоге, чтобы содержимое всего каталога можно было загрузить в местоположение baseUrl (что необходимо сделать вручную). Опция UPLOAD_DIRECTORY предназначена для того, чтобы позволить проекту переопределить местоположение этого отдельного каталога, но генератор IFW всегда создает каталог под названием repository, расположенный на несколько уровней глубже базового каталога _CPack_Packages.

Генератор IFW позволяет проектам указывать дополнительные репозитории для доступа инструмента обслуживания и программы установки. Это может быть полезно, если различные компоненты поставляются разными поставщиками или если график выпуска одних компонентов отличается от графика выпуска других.

```
cpack_ifw_add_repository(repoName
    URL baseUrl
    [DISPLAY_NAME displayName]
    [DISABLED]
    [USERNAME username]
    [PASSWORD password]
)
```

repoName - это внутреннее имя отслеживания, а baseUrl имеет такое же значение, как и для cpack_configure_downloads(). Опция DISPLAY_NAME обычно должна использоваться для указания значимого имени, иначе baseUrl будет отображаться как имя хранилища, что, как правило, менее удобно для пользователя. Если хранилищу требуется имя пользователя и пароль, их можно указать, но имейте в виду, что пароль будет храниться в незашифрованном виде и должен считаться небезопасным. Ключевое слово DISABLED указывает, что хранилище должно быть отключено по умолчанию, но пользователь может включить его в программе установки или в пользовательском интерфейсе инструмента обслуживания.

Пример основного хранилища для пакетов релиза и вторичного хранилища для пакетов предварительного просмотра (отключенного по умолчанию) может быть настроен следующим образом:

```
include(CPack)
include(CPackIFW)

cpack_configure_downloads(https://example.com/packages/product/release ALL)
cpack_ifw_add_repository(secondaryRepo
    DISPLAY_NAME "Preview features"
    URL          https://example.com/packages/product/preview
    DISABLED
)
```

К сожалению, команда cpack_configure_downloads() в настоящее время не поддерживает указание отображаемого имени, поэтому основной URL, который она предоставляет, всегда будет отображаться как пустой URL, а не как более удобное имя.

Одним из недостатков этого генератора пакетов является то, что создаваемая программа установки не предоставляет пользователям простой способ запуска установки без участия командной строки. Это ограничение самого Qt Installer Framework, а не CMake или CPack. Программа установки также имеет дополнительные накладные расходы по сравнению с большинством других типов генераторов, поскольку она включает поддержку Qt, необходимую для интерфейса программы установки, работы с сетью и так далее. Это может сделать размер даже тривиального инсталлятора 18 Мб или более, по сравнению с несколькими сотнями кб для других типов генераторов.

Выше были рассмотрены только основные аспекты генератора IFW, существует гораздо больше возможностей, которые позволяют проектам значительно расширить настройки программы установки и инструмента сопровождения. Для многих проектов описанные выше функциональные возможности уже позволяют создавать гибкие, надежные и кроссплатформенные программы установки. Если потребуется дальнейшая доработка, представленные возможности послужат надежной основой для расширения.

26.4.3. WIX

Генератор пакетов WIX создает инсталляторы .msi для Windows с помощью набора инструментов WiX. По сравнению с генератором пакетов IFW, он имеет схожую степень настраиваемости пользовательского интерфейса и обладает следующими преимуществами:

- Установка с помощью командной строки (т.е. без участия пользователя) поддерживается непосредственно через опцию к инструменту msiexec.
- Установщики тесно интегрированы в функцию Add/Remove в Windows.
- Внешний вид по умолчанию должен быть знаком большинству пользователей.

С другой стороны, он имеет следующие недостатки по сравнению с IFW:

- Нет простого, прямого способа предоставления локализованных имен и описаний компонентов.
- CPACK_WIX_COMPONENT_INSTALL и CPACK_COMPONENTS_GROUPING игнорируются (см. ниже).
- Нет поддержки загружаемых компонентов.
- Несколько версий с одним и тем же GUID обновления не могут быть установлены одновременно (см. ниже). Каждая установка заменяет предыдущую, даже если она находится в совершенно другом каталоге.

По умолчанию генератор WIX создает пакет на основе компонентов, который всегда будет представлен в пользовательском интерфейсе так, как если бы CPACK_COMPONENTS_GROUPING был установлен в IGNORE. Если пакет, основанный на компонентах, нежелателен, можно установить CPACK_MONOLITHIC_INSTALL в true, но тогда все определенные компоненты всегда будут установлены. Невозможно включить в монолитный установщик только некоторые компоненты, и если установлено значение CPACK_COMPONENTS_ALL, CMake выдаст предупреждение и проигнорирует CPACK_COMPONENTS_ALL.

Ключевой частью программы установки WIX является то, что она содержит GUID продукта и GUID обновления. Если любой другой установленный пакет имеет такой же GUID обновления, то этот пакет будет обновлен, а не установлен новый пакет как отдельный продукт. Если GUID обновления одинаковый, а GUID продукта разный, то обновление считается крупным обновлением, и новая программа установки полностью заменит старый пакет. Если GUID продукта также совпадает, то новая программа установки должна быть способна выполнить незначительное обновление, если только программа установки сообщает более новый номер версии, чем у установленного в данный момент пакета. Пакеты обновлений являются примером, когда GUID продукта сохраняется таким же, как и у базовой версии, к которой они применяются. Если вы не создаете достаточно продвинутую стратегию установки или упаковки, проекты обычно должны изменять GUID продукта с каждым выпуском, поскольку ограничения самой Windows на сохранение одного и того же GUID продукта от одного пакета к другому достаточно строги.

CPack обеспечивает прямую поддержку для установки GUID продукта и обновления. Переменные CPACK_WIX_PRODUCT_GUID и CPACK_WIX_UPGRADE_GUID могут быть установлены перед вызовом include(CPack), чтобы управлять ими вручную, или их можно не устанавливать, чтобы позволить cpack генерировать новые значения при каждом вызове. Для GUID продукта такая автоматическая генерация, вероятно, будет желательным поведением, но GUID обновления в идеале не должен меняться в течение всего срока службы продукта. Проекты должны получить GUID и установить CPACK_WIX_UPGRADE_GUID в это значение, а затем, в идеале, никогда больше не изменять его. Это гарантирует, что все будущие релизы смогут беспрепятственно обновлять старые релизы. Фактический GUID можно получить различными способами, например, с помощью инструментов командной строки, веб-генераторов UUID или даже в самом CMake с

помощью команды string(UUID). Для некоторых продуктов может иметь смысл менять GUID обновления с каждым основным выпуском, чтобы позволить более старому основному выпуску сосуществовать с более новым, тем самым облегчая пользователям путь миграции.

Одним из критериев того, когда GUID продукта должен измениться, является изменение имени файла .msi. Поскольку имя файла программы установки обычно включает некоторые сведения о версии, это означает, что каждый выпуск будет считаться крупным обновлением. Если пользователь установит новую версию, она полностью заменит все ранее установленные версии. Новая версия может быть установлена в другой каталог, а старая будет удалена. Может возникнуть соблазн использовать каталог установки по умолчанию (управляемый CPACK_PACKAGE_INSTALL_DIRECTORY), который включает номер версии, но пользователи, скорее всего, предпочтут, чтобы каталог по умолчанию оставался неизменным при всех обновлениях. В идеале каталог по умолчанию должен меняться только при изменении GUID обновления, поскольку именно этот идентификатор обеспечивает преемственность от одной версии к другой.

При установке нового пакета, когда уже установлен другой пакет с тем же GUID обновления, производится проверка между версиями. Только если новый пакет имеет более позднюю версию, обновление будет разрешено. При такой проверке учитываются только первые три компонента номера версии, поэтому версии 2.7.4.3 и 2.7.4.9 будут считаться одной и той же версией с точки зрения обновления. Поэтому проектам, планирующим использовать генератор WIX, следует избегать использования более трех компонентов номера версии. Если разрешить автоматическую установку CPACK_PACKAGE_VERSION из отдельных частей версии CPACK_PACKAGE_VERSION_xxx, это уже будет соблюдаться.

Большинство настроек пользовательского интерфейса по умолчанию приемлемы для базового пакета WIX. Проекты могут захотеть предоставить значок продукта вместо стандартного значка установщика MSI для улучшения брендинга в области Add/Remove, но в остальном значения по умолчанию вполне приемлемы. В следующем примере показана базовая конфигурация программы установки WIX.

```
# Define generic setup for all generator types...

# WIX-specific configuration
set(CPACK_WIX_PRODUCT_ICON ${CMAKE_CURRENT_LIST_DIR}/Logo.ico)
set(CPACK_WIX_UPGRADE_GUID XXXXXXXX-XXXX-XXXX-XXXX-XXXXXXXXXXXX)

include(CPack)
# Define components and component groups...
```

26.4.4. НСИС

Генератор пакетов NSIS создает исполняемые файлы программы установки для Windows с помощью системы Nullsoft Scriptable Install System. Он имеет ряд общих характеристик с генераторами IFW и WIX, включая возможность настройки пользовательского интерфейса и поддержку иерархий компонентов. К преимуществам генератора NSIS относятся:

- Исполняемый файл программы установки напрямую поддерживает неуправляемую установку с помощью специальной опции командной строки.
- Это единственный активно поддерживаемый генератор CPack, который поддерживает типы установки.
- Команды предварительной/послеустановочной и предварительной деинсталляции поддерживаются напрямую, хотя они должны быть реализованы как команды NSIS.

Генератор NSIS имеет несколько недостатков:

- CPACK_NSIS_COMPONENT_INSTALL и CPACK_COMPONENTS_GROUPING игнорируются. В этом отношении генератор NSIS имеет те же ограничения, что и генератор WIX.
- Нет поддержки загружаемых компонентов.
- После установки продукта пользователи не могут изменить набор установленных компонентов без повторной установки.
- Поддерживается только базовая настройка пользовательского интерфейса и нет прямой поддержки локализации любого содержимого пользовательского интерфейса. Это ограничения генератора CPack, а не самой NSIS, которая предлагает некоторые возможности с помощью собственного языка сценариев.
- Хотя можно установить разные версии в разные места, они используют общие данные реестра и поэтому не полностью изолированы друг от друга. Только одна версия будет отображаться в области Add/Remove в настройках Windows.

По умолчанию эти программы установки будут выполнять обновление существующей установки только в том случае, если новый пакет установлен в тот же каталог, что и старый. Проекты могут установить значение
Переменная CPACK_NSIS_ENABLE_UNINSTALL_BEFORE_INSTALL имеет значение true, чтобы заставить программу установки сначала проверить реестр на наличие существующей установки пакета. Эта проверка не зависит от места установки, поэтому это более надёжный способ проверки существующей установки для обновления. Поэтому для большинства проектов рекомендуется устанавливать значение переменной true.

Установщики NSIS выигрывают от переопределения внешнего вида по умолчанию в ряде областей. Необходимо установить значки, используемые для программы установки, деинсталляции и самого продукта в области Add/Remove, поскольку по умолчанию они либо низкого качества, либо выдают пустые поля. Имя, отображаемое для продукта, также должно быть явно задано, чтобы избежать несоответствующего текста по умолчанию, поставляемого CPack. В следующем примере показана базовая конфигурация с переопределениями, чтобы избежать значений по умолчанию, которые большинство проектов сочтут неприемлемыми.

```
# Define generic setup for all generator types...

# NSIS-specific configuration
set(CPACK_NSIS_MUI_ICON    ${CMAKE_CURRENT_LIST_DIR}/InstallerIcon.ico)   ①
set(CPACK_NSIS_MUI_UNIICON ${CMAKE_CURRENT_LIST_DIR}/UninstallerIcon.ico) ②
set(CPACK_NSIS_INSTALLED_ICON_NAME bin/MainApp.exe)       ③
set(CPACK_NSIS_DISPLAY_NAME        "My Project Suite")    ④
set(CPACK_NSIS_PACKAGE_NAME        "My Project")          ⑤

set(CPACK_NSIS_ENABLE_UNINSTALL_BEFORE_INSTALL YES)

include(CPack)
# Define components and component groups...
```

① Значок, используемый для самой программы установки. Windows может наложить дополнительное содержимое, чтобы указать, что для работы программы установки требуются привилегии администратора. Используйте абсолютный путь, чтобы NSIS могла найти значок при создании программы установки.

② Значок, используемый для деинсталлятора, который будет скопирован в каталог установки. Опять же, используйте абсолютный путь к значку.

③ Этот параметр определяет значок, используемый для продукта в области добавления/удаления. Это должен быть путь либо к файлу значка (.ico), либо к исполняемому файлу со встроенным значком приложения. Путь должен указывать на место *установки*, относительно базовой точки установки.

④ Имя, отображаемое для пакета только в области добавления/удаления.

⑤ Название, используемое во многих местах пользовательского интерфейса программы установки, а также в строке заголовка во время установки. В некоторых контекстах к нему может добавляться слово Setup.

26.4.5. DragNDrop

На Mac продукты обычно распространяются в виде файла .dmg. Они действуют как образ диска и могут содержать что угодно - от одного приложения до целого набора приложений, ссылки на документацию и т. д. Симлинк на область /Applications часто предоставляется как часть образа, чтобы пользователи могли легко перетащить приложения на него для установки, отсюда и название DragNDrop для этого типа генератора. Конфигурационные переменные, специфичные для этого типа генератора, используют в своем имени DMG, а не DRAGNDROP, но учтите, что cpack распознает DragNDrop только как имя самого генератора.

Формат .dmg ближе к архиву, чем к программе установки с пользовательским интерфейсом. Компоненты используются для управления созданием одного или нескольких файлов .dmg и тем, что содержит каждый файл .dmg, но пользовательского интерфейса для выбора компонентов при установке нет. Пользователь должен открыть .dmg файл(ы) и перетащить содержимое в нужное место для установки. CPACK_COMPONENTS_ALL контролирует, какие компоненты будут установлены, а переменная CPACK_COMPONENTS_GROUPING контролирует, как эти компоненты распределяются между .dmg файлами следующим образом:

ALL_COMPONENTS_IN_ONE

Все компоненты будут включены в один файл .dmg.

ONE_PER_GROUP

Каждая группа компонентов верхнего уровня и каждый компонент, не входящий в группу, будут помещены в отдельный файл .dmg.

IGNORE

Каждый компонент будет помещен в свой отдельный .dmg файл, а все группы компонентов будут проигнорированы.

Этот тип генератора практически не требует настройки, выходящей за рамки стандартных параметров. Размер и расположение

Окно Finder, отображаемое при открытии образа диска, можно контролировать, предоставив пользовательский файл .DS_Store. Проект должен будет либо подготовить такой файл вручную, используя папку-пример, содержащую то же самое, что и конечный образ диска, либо он может быть создан программно на языке AppleScript. Переменная CPACK_DMG_DS_STORE может быть использована для названия заранее

подготовленного файла .DS_Store или CPACK_DMG_DS_STORE_SETUP_SCRIPT может указывать на файл AppleScript, который будет запущен во время генерации пакета. В любом случае, при желании можно задать фоновое изображение с помощью переменной CPACK_DMG_BACKGROUND_IMAGE, но относительно часто фон оставляют пустым по умолчанию. Для случаев, когда образ диска не должен предоставлять сим-ссылку на папку /Applications, проект должен установить CPACK_DMG_DISABLE_APPLICATIONS_SYMLINK в true.

Для образа диска можно указать значок, установив CPACK_PACKAGE_ICON на значок в формате .icns. Этот значок используется только для представления файла .dmg при монтировании, а не для самого файла .dmg. Указанный значок может отображаться в строке заголовка Finder или в некоторых видах Finder, но в остальном он не является активно отображаемым значком.

Ограниченная языковая локализация обеспечивается через CPACK_DMG_SLA_DIR и Переменные CPACK_DMG_SLA_LANGUAGES. Они могут быть использованы для предоставления конкретных фраз, используемых на этапе открытия образа диска в лицензионном соглашении, а также для предоставления локализованной версии самого лицензионного соглашения. Как используются эти две переменные и требования к языковым файлам, которые должны быть предоставлены, смотрите в документации модуля CPackDMG.

Тип генератора Bundle связан с генератором DragNDrop. Он использует тот же набор переменных DMG, плюс некоторые собственные. Генератор Bundle изначально предназначался для создания одного пакета приложений, потенциально предназначенного для отправки в Apple App Store. В настоящее время такие пакеты приложений лучше готовить во время сборки с помощью генератора CMake в Xcode, поскольку это более точно соответствует процессу, ожидаемому Apple. Рекомендуемый способ подготовки таких пакетов приложений вместо использования генератора типа CPack Bundle см. в главе 22 "*Возможности Apple*".

26.4.6. productbuild

Альтернативой генератору DragNDrop является productbuild. Вместо создания образа диска .dmg он создаёт пакет .pkg для использования с приложением macOS Installer. CPACK_COMPONENTS_GROUPING игнорируется, и программа установки всегда ведёт себя так, как если бы эта переменная была установлена в IGNORE. CPACK_MONOLITHIC_INSTALL не должен быть установлен в true с этим генератором, так как это может привести к созданию неработающих инсталляторов. Типы инсталляторов не поддерживаются, и возможности настройки пользовательского интерфейса очень ограничены, хотя значения по умолчанию, как правило, достаточны.

По сравнению с генератором IFW, основным преимуществом productbuild является возможность подписывать программу установки. Это легко настраивается установкой CPACK_PRODUCTBUILD_IDENTITY_NAME (а также CPACK_PRODUCTBUILD_KEYCHAIN_PATH, если требуется) в детали подписи. Часто достаточно просто указать идентификатор по умолчанию, что можно сделать следующим образом:

```
set(CPACK_PRODUCTBUILD_IDENTITY_NAME "Developer ID Installer")
include(CPack)
```

В генераторе productbuild отсутствует поддержка загружаемых компонентов, поэтому создание онлайн-установщиков невозможно. Обновления выполняются путем замены предыдущего содержимого существующей установки. Как и в установщиках NSIS, набор установленных компонентов не может быть изменен без переустановки продукта. Также обычно невозможно установить несколько версий одновременно в разные каталоги.

Установщики, создаваемые генератором productbuild, по умолчанию являются перемещаемыми. Это означает, что при установке пакета на машину конечного пользователя, если ОС знает о существовании пакета

приложений с тем же именем, что и одно из приложений, предоставляемых пакетом, программа установки перезапишет существующее приложение, независимо от того, где оно находится в файловой системе. В таких случаях приложение не будет установлено в область /Applications по умолчанию, что обычно означает, что оно также не появится в тех местах, где его ожидает увидеть пользователь. Такая ситуация обычно возникает у разработчиков на машине, которую они используют для сборки и тестирования пакетов. Пакет приложений, созданный при сборке, известен ОС, поэтому при установке пакета пакет из дерева сборки используется в качестве места установки этого приложения вместо ожидаемого места в /Applications. Кроме того, в каталоге _CPack_Packages staging дерева сборки будет находиться еще одна копия приложения, что может привести к аналогичному поведению. Чтобы правильно протестировать программу установки, все копии устанавливаемых пакетов приложений необходимо сначала удалить с машины разработчика перед запуском программы установки.

Одним из обходных путей решения вышеупомянутой проблемы перемещения является пометка компонентов как неперемещаемых. Это не позволит программе установки выбрать местоположение существующего пакета приложений, но при этом пользователь не сможет перемещать пакеты приложений по своему усмотрению. Чтобы сделать компонент неперемещаемым, для каждого компонента необходимо предоставить собственный plist-файл с помощью опции PLIST в команде cpack_add_component(). Файл plist должен быть получен с помощью опции --analyze команды pkgbuild, другие опции которой можно найти, посмотрев подробный вывод команды cpack для проекта:

```
cpack -G productbuild -V
```

Типичный plist-файл может выглядеть примерно так:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN"
    "http://www.apple.com/DTDs/PropertyList-1.0.dtd">
<plist version="1.0">
<array>
    <dict>
        <key>BundleHasStrictIdentifier</key>
        <true/>
        <key>BundleIsRelocatable</key>
        <true/>
        <key>BundleIsVersionChecked</key>
        <true/>
        <key>BundleOverwriteAction</key>
        <string>upgrade</string>
        <key>RootRelativeBundlePath</key>
        <string>Applications/MyApp.app</string>
    </dict>
</array>
</plist>
```

Измените значение элемента словаря BundleIsRelocatable на false, чтобы предотвратить перемещение приложения в ОС при установке. Для каждого пакета приложений в компоненте будет одна секция <dict></dict>. После того как plist-файл будет создан и обновлен, его можно использовать следующим образом:

```
cpack_add_component(MyProj_Runtime
    ...   # Other options
    PLIST runtime.plist
)
```

Генератор productbuild следует рассматривать как замену устаревшему и более не поддерживаемому генератору PackageMaker. Apple больше не предоставляет приложение PackageMaker, поэтому разработчики, использующие новые версии macOS, должны использовать productbuild.

26.4.7. RPM

В системах Linux RPM является одним из двух доминирующих форматов управления пакетами. Пакеты RPM не имеют собственного пользовательского интерфейса, по сути, это просто архивы с довольно обширным набором метаданных и некоторыми функциями сценариев. Менеджер пакетов системы использует их для управления зависимостями между пакетами, предоставления информации пользователю, запуска сценариев предварительной и последующей установки и удаления и так далее.

Поскольку сам пакет не имеет функций пользовательского интерфейса, в этой области нет необходимости в настройке, но генератор RPM предоставляет широкие возможности настройки метаданных с помощью большого количества переменных. Многие из этих переменных не нужно задавать явно, поскольку большинство значений по умолчанию подходят для проектов, которым не нужно делать ничего сложного. Для пакетов, которым не нужно вызывать сценарии предварительной/послеустановочной или деинсталляционной установки и для которых межпакетные зависимости могут быть автоматически определены базовым инструментом создания пакетов, объем настройки аналогичен другим генераторам пакетов.

Генератор RPM поддерживает установку компонентов, но по умолчанию компоненты отключены. Когда компоненты отключены, создается только один .rpm, и поведение будет таким, как если бы CPACK_MONOLITHIC_INSTALL был установлен в true. В таких случаях все компоненты включаются в пакет. Если компоненты включены, то CPACK_COMPONENTS_GROUPING имеет свое обычное значение, и будет создано несколько файлов .rpm. Компоненты включаются установкой CPACK_RPM_COMPONENT_INSTALL в true, а набор установленных компонентов контролируется CPACK_COMPONENTS_ALL, как обычно.

```
# Define generic setup for all generator types...
set(CPACK_COMPONENTS_GROUPING ONE_PER_GROUP)

# RPM-specific configuration
set(CPACK_RPM_COMPONENT_INSTALL YES)

include(CPack)
# Define components and component groups...
```

Имена компонентов или групп могут не подходить для использования в качестве имен пакетов, которые обычно видны пользователю как часть имени файла .rpm, в приложениях пользовательского интерфейса менеджера пакетов RPM и т.д. Эти имена можно задать для каждого компонента с помощью команды CPACK_RPM_<COMP>_PACKAGE_NAME, где <COMP> - имя компонента в верхнем регистре. При создании пакета с отключенными компонентами имя единого монолитного пакета можно отменить, задав вместо него CPACK_RPM_PACKAGE_NAME.

```
add_executable(sometool ...)
install(TARGETS sometool ... COMPONENT MyProjUtils)

set(CPACK_RPM_MYPROJUTILS_PACKAGE_NAME myproj-tools)
include(CPack)
```

Имя .rpm-файлов также может быть изменено, и вполне вероятно, что проекты захотят это сделать. Имя .rpm-файла каждого компонента контролируется переменной CPACK_RPM_<COMP>_FILE_NAME (или просто CPACK_RPM_FILE_NAME для некомпонентной упаковки). Значение по умолчанию для этих переменных соответствует следующей схеме:

```
<CPACK_PACKAGE_FILE_NAME>[-<component>].rpm
```

Часть <component> - это оригинальное имя компонента (т.е. без изменений в верхнем/нижнем регистре). Недостатком этого имени файла по умолчанию является то, что оно не включает никаких сведений о версии или архитектуре, но такая информация обычно требуется (или, по крайней мере, желательна). Обычно предпочтительнее указать cpack, чтобы инструмент создания пакета по умолчанию выбрал лучшее имя пакета по умолчанию, что можно сделать, установив CPACK_RPM_<COMP>_FILE_NAME в специальную строку RPM-DEFAULT. Ниже приведены примеры типичных имен файлов, получаемых при такой схеме.

Имя файла пакета RPM-DEFAULT автоматически включает архитектуру. Если архитектура должна быть указана явно, например, чтобы пометить пакет как noarch, чтобы указать, что он не зависит от архитектуры, переменная CPACK_RPM_<COMP>_PACKAGE_ARCHITECTURE для каждого компонента может быть установлена в требуемое значение или CPACK_RPM_PACKAGE_ARCHITECTURE может быть установлена в качестве значения по умолчанию, если не задано переопределение для конкретного компонента (она также используется для монолитных пакетов). Значение по умолчанию для архитектуры вычисляется cpack как результат uname -m, но при сборке 32-битного пакета на 64-битном хосте это будет неверно, поэтому в проекте необходимо явно установить значение архитектуры.

Файлы RPM должны содержать информацию о версии . Генератор RPM будет использовать

CPACK_PACKAGE_VERSION по умолчанию, но номер версии, специфичный для RPM, также может быть задан с помощью CPACK_RPM_PACKAGE_VERSION, если это необходимо (но такая необходимость должна возникать редко). Обратите внимание, что в настоящее время невозможно указать версии для каждого компонента, генератор CPack RPM в настоящее время ограничен использованием одной и той же версии для всех компонентов. В дополнение к версии пакета, пакеты RPM также имеют отдельный номер выпуска, который указывается с помощью CPACK_RPM_PACKAGE_RELEASE. Этот номер выпуска является выпуском самого пакета, а не продукта, поэтому версия пакета обычно остается неизменной, если номер выпуска увеличивается (например, для устранения проблемы с упаковкой). Если версия пакета меняется, номер релиза обычно сбрасывается до 1, что является значением по умолчанию, если CPACK_RPM_PACKAGE_RELEASE не указан. Необязательная эпоха также может быть указана CPACK_RPM_PACKAGE_EPOCH, и ее использование может быть более распространено в некоторых системах или репозиториях, чем в других. Полная версия имеет формат E:X.Y.Z-R, где E - эпоха и должна быть числом, если оно указано. Если эпоха не задана, полная версия имеет формат X.Y.Z-R. Если не известно, что требуется значение эпохи, в проектах, как правило, эпоха не задается.

Если проект явно не переопределит CPACK_PACKAGE_VERSION и CPACK_RPM_PACKAGE_ARCHITECTURE, их значения не будут доступны в файлах CMakeLists.txt, поскольку значения по умолчанию для этих переменных вычисляются только при обработке cpack входного файла, а не при запуске CMake. Это означает, что гораздо

сложнее задать имя файла пакета напрямую, чем использовать RPM-DEFAULT. В следующем примере показано, как использовать функцию RPM-DEFAULT:

```
set(CPACK_RPM_PACKAGE_RELEASE 5)   # Optional, default of 1 is often okay
if(CMAKE_SIZEOF_VOID_P EQUAL 4)
    set(CPACK_RPM_PACKAGE_ARCHITECTURE i686)
endif()

set(CPACK_RPM_MYPROJUTILS_PACKAGE_NAME myproj-tools)
set(CPACK_RPM_MYPROJUTILS_FILE_NAME    RPM-DEFAULT)
include(CPack)
```

Если предположить, что CPACK_PACKAGE_VERSION оценивается как строка вида X.Y.Z, то в этом примере имена файлов пакетов будут иметь вид:

```
myproj-tools-X.Y.Z-5.i686.rpm
myproj-tools-X.Y.Z-5.x86_64.rpm
```

Как уже говорилось в предыдущей главе, базовая точка установки по умолчанию вряд ли будет желательна в системах Linux, и это распространяется на создание пакетов RPM. На самом деле, для всех систем, кроме Windows, более подходящая базовая точка должна быть установлена и для упаковки тоже, путем явной установки переменной CPACK_PACKAGING_INSTALL_PREFIX. Расширяя пример из предыдущей главы, проект может захотеть сделать что-то вроде следующего:

```
if(NOT WIN32 AND CMAKE_SOURCE_DIR STREQUAL CMAKE_CURRENT_SOURCE_DIR)
    set(CMAKE_INSTALL_PREFIX "/opt/mycompany.com/${PROJECT_NAME}")
    set(CPACK_PACKAGING_INSTALL_PREFIX ${CMAKE_INSTALL_PREFIX})
endif()
```

Уникальной особенностью пакетов RPM является то, что они могут включать пути перемещения. Пакеты могут указывать один или несколько префиксов путей, которые пользователь может выбрать для перемещения в другую часть файловой системы во время установки. Для поддержки этой возможности переменная CPACK_RPM_PACKAGE_RELOCATABLE должна быть установлена в true, а затем CPACK_RPM_RELOCATION_PATHS может содержать список префиксов путей, которые пользователь может перемещать. При использовании этой функции разработчикам следует обратиться к документации модуля CPackRPM, чтобы понять, как обрабатываются относительные пути и различные отступления по умолчанию, которые применяются к обеим этим переменным. Обратите внимание, что если проект включен в состав дистрибутива Linux, то его сопровождающим, скорее всего, придется переопределять как переменные префикса установки, так и каталоги перемещения, поэтому лучше не усложнять ситуацию.

Обычно ожидается, что инструмент создания пакетов RPM будет удалять из исполняемых файлов и разделяемых библиотек все отладочные символы перед добавлением их в пакет. Это объясняется тем, что размер двоичных файлов релиза должен быть минимальным, и они обычно скрывают детали реализации и не предоставляют средств отладки. Обычно зачистка контролируется переменной CPACK_STRIP_FILES, которая определяет, выполняется ли зачистка в рамках поэтапной установки при упаковке, но в случае с генератором RPM инструмент создания пакетов RPM часто выполняет собственную зачистку по умолчанию. Поэтому, даже если CPACK_STRIP_FILES имеет значение false или не установлен, зачистка все равно может произойти.

Основная проблема заключается в том, что инструмент создания пакетов rpmbuild обычно имеет секцию post staging install, которая удаляет двоичные файлы и выполняет другие задачи перед созданием окончательного пакета .rpm. Традиционно, обходным решением, предлагаемым cpack, является отмена такого поведения путем установки переменной CPACK_RPM_SPEC_INSTALL_POST, обычно на что-то вроде /bin/true. Этот подход устарел в пользу использования CPACK_RPM_SPEC_MORE_DEFINE:

```
# Prevent stripping and other post-install steps during package creation
set(CPACK_RPM_SPEC_MORE_DEFINE "%define __spec_install_post /bin/true")
```

Хотя описанная выше техника предотвращения зачистки работает, она также отбрасывает все другие операции, которые обычно применяются (например, автоматическая компиляция байт-кода для файлов python, пост-обработка с учетом особенностей архитектуры). Потенциально лучшей альтернативой является разрешение зачистки двоичных файлов в .rpm и создание отдельного пакета отладочной информации. Первоначальная поддержка создания пакетов отладочной информации была добавлена в CMake 3.7 и была улучшена в 3.8 и 3.9. Для включения этой функции обычно достаточно установить CPACK_RPM_DEBUGINFO_PACKAGE или эквивалент CPACK_RPM_<COMP>_DEBUGINFO_PACKAGE для конкретного компонента в true. Создаваемые пакеты debuginfo будут содержать исходные файлы, а также отладочную информацию. По умолчанию исходные файлы берутся из CMAKE_SOURCE_DIR и CMAKE_BINARY_DIR, но при необходимости это можно переопределить с помощью переменной CPACK_BUILD_SOURCE_DIRS. Части иерархии исходных каталогов могут быть исключены с помощью переменных CPACK_RPM_DEBUGINFO_EXCLUDE_DIRS и CPACK_RPM_DEBUGINFO_EXCLUDE_DIRS_ADDITION, хотя проекты, вероятно, захотят установить только последнюю. Первая обычно используется для исключения системных каталогов и имеет соответствующее значение по умолчанию. Сопровождающие дистрибутива могут захотеть переопределить CPACK_RPM_DEBUGINFO_EXCLUDE_DIRS независимо от того, что проект установит в CPACK_RPM_DEBUGINFO_EXCLUDE_DIRS_ADDITION, отсюда и использование двух отдельных переменных.

При создании пакетов debuginfo иногда может возникнуть ошибка, подобная следующей:

```
CPackRPM: source dir path '/path/to/source/dir' is shorter
than debuginfo sources dir path
'/usr/src/debug/SomeProject-X.Y.Z-Linux/src_0'! Source dir path must be
longer than debuginfo sources dir path.  Set
CPACK_RPM_BUILD_SOURCE_DIRS_PREFIX variable to a shorter value or make
source dir path longer.  Required for debuginfo packaging.  See
documentation of CPACK_RPM_DEBUGINFO_PACKAGE variable for details.
```

Из-за того, что пути переписываются в процессе обработки отладочной информации, путь к дереву исходников должен быть длиннее, чем предполагаемое место установки исходников. Обратите внимание, что это может повлиять на системы непрерывной интеграции, где расположение дерева исходников обычно фиксировано. Необходимость в большей длине пути может противоречить другим ограничениям, при которых длина пути должна быть минимизирована, поэтому внимательно рассмотрите, могут ли такие ограничения применяться к проекту.

Исходные RPM также могут быть созданы генератором RPM. Они похожи на пакеты debuginfo, но содержат только исходные тексты и никакой отладочной информации. Они создаются так же, как исходные пакеты для других генераторов пакетов, а документация модуля CPackRPM содержит базовые инструкции, показывающие, как собрать двоичный RPM из исходного RPM, что может быть полезным шагом проверки.

```
# Create source RPM
cpack -G RPM --config CPackSourceConfig.cmake

# Verify that a binary RPM can be produced from it
mkdir -p build_dir/{BUILD,BUILDROOT,RPMS,SOURCES,SPECS,SRPMS}
rpmbuild --define "_topdir build_dir" --rebuild <source-RPM-filename>
```

Модуль CPackRPM предоставляет гораздо больше переменных, чем рассмотренные выше. Можно указать подробную информацию о том, что пакеты предоставляют или требуют, или направить инструмент создания пакетов на автоматическое вычисление этих параметров. Если пакет заменяет или конфликтует с другими пакетами, это также может быть указано. Можно указать сценарии, которые будут выполняться до или после установки и удаления пакета, или, если необходим полный контроль, проект может предоставить свой собственный шаблон файла .spec вместо того, чтобы использовать шаблон по умолчанию, предоставляемый cpack (хотя этого следует избегать, если это возможно, поскольку это сводит на нет большую часть функциональности, уже предоставляемой cpack).

26.4.8. DEB

Формат DEB - это другой доминирующий формат пакетов для систем Linux, и как DEB, так и RPM имеют много схожих характеристик. Пакеты DEB - это, по сути, архивы с соответствующими метаданными, которые менеджер пакетов системы использует для внедрения зависимостей, запуска сценариев и так далее.

Разница между DEB и RPM заключается в том, что для подготовки пакетов DEB не требуется специальный инструмент, в отличие от пакетов RPM. Это позволяет создавать DEB-пакеты на системах, которые сами не используют формат DEB, что означает возможность создания как RPM, так и DEB-пакетов на системах, основанных на RPM, таких как RedHat, SuSE и т.д. Основная оговорка заключается в том, что при создании DEB-пакетов на системах, не использующих формат DEB, такие инструменты, как dpkg-shlibdeps, недоступны, поэтому такие вещи, как автоматические зависимости, не могут быть вычислены.

Работа с компонентами очень похожа на работу с RPM и имеет аналогичные конфигурационные переменные. Компоненты включаются установкой CPACK_DEB_COMPONENT_INSTALL в true (эта переменная не соответствует именованию, используемому для всех других переменных, специфичных для DEB, которые имеют имя с префиксом CPACK_DEBIAN_, а не CPACK_DEB_). Имена пакетов имеют аналогичные переменные CPACK_DEBIAN_PACKAGE_NAME и CPACK_DEBIAN_<COMP>_PACKAGE_NAME, а имена файлов контролируются переменными
CPACK_DEBIAN_FILE_NAME и CPACK_DEBIAN_<COMP>_FILE_NAME. Для DEB применимы те же вопросы именования файлов, что и для RPM, за исключением того, что вместо RPM-DEFAULT следует использовать специальное значение DEB-DEFAULT. Если указано любое другое значение, имя файла должно заканчиваться на .deb или .ipk. Версионность для DEB также обрабатывается аналогично RPM, как и указание архитектуры. Предоставляются эквивалентные переменные DEB, при этом DEBIAN заменяет RPM в именах переменных.

Генератор пакетов DEB имеет меньше переменных, влияющих на обработку зависимостей, по сравнению с RPM. Если упаковка выполняется на хосте на базе DEB, где доступен инструмент dpkg-shlibdeps, зависимости разделяемых библиотек могут быть автоматически вычислены путем установки значения
CPACK_DEBIAN_PACKAGE_SHLIBDEPS или переменные CPACK_DEBIAN_<COMP>_PACKAGE_SHLIBDEPS, специфичные для компонента, в true. Вручную указанные зависимости могут быть предоставлены через переменные CPACK_DEBIAN_PACKAGE_DEPENDS и CPACK_DEBIAN_<COMP>_PACKAGE_DEPENDS и будут объединены с автоматически определенными, если используются как ручные, так и автоматические зависимости. Обратите внимание, однако, что если установлена переменная зависимости, специфичная для

компонента, то переменная, не относящаяся к компоненту, не будет использоваться для этого компонента. Если включен автоматический расчет зависимостей, он заполняет специфические для компонента переменные, поэтому если в проекте задана только CPACK_DEBIAN_PACKAGE_DEPENDS, она будет игнорироваться для тех компонентов, для которых заполняются автоматические зависимости. Поэтому может быть более надежным всегда заполнять CPACK_DEBIAN_<COMP>_PACKAGE_DEPENDS, а не CPACK_DEBIAN_PACKAGE_DEPENDS, когда автоматические зависимости включены. Проекты также должны устанавливать CPACK_DEBIAN_ENABLE_COMPONENT_DEPENDS в true, если межкомпонентные зависимости указаны через опцию DEPENDS в cpack_add_component(), что приведет к внедрению этих зависимостей в генерируемые пакеты компонентов.

В связи с вышесказанным, каждый пакет может также указать разделяемые библиотеки, которые он требует. На платформах, предоставляющих инструмент readelf, эти библиотечные зависимости могут быть определены автоматически путем установки CPACK_DEBIAN_PACKAGE_GENERATE_SHLIBS в true. Затем инструмент readelf используется для определения разделяемых библиотек, необходимых каждому разделяемому объекту, и эта информация добавляется в пакет. Переменная CPACK_DEBIAN_PACKAGE_GENERATE_SHLIBS_POLICY определяет, будут ли применяться точные (=) или минимальные (>=) требования.

В документации к модулю CPackDeb подробно описан ряд других переменных, специфичных для DEB, не упомянутых выше. В частности, некоторые переменные могут использоваться для указания того, что пакет(ы) требует, предоставляет, заменяет и так далее. Также могут быть заданы некоторые элементы метаданных, специфичные для DEB, такие как данные о сопровождающем, группа или категория пакета и т.д. Разработчикам следует обратиться к документации модуля для получения полного набора поддерживаемых переменных.

26.4.9. FreeBSD

Генератор пакетов FreeBSD относительно незрелый, он был добавлен только в CMake 3.10. Он не поддерживает компоненты и всегда создает один файл .pkg. Некоторые переменные, специфичные для FreeBSD, могут быть установлены для определения основных метаданных пакета, а некоторые возвращаются к переменным, специфичным для DEB или RPM. Большая часть конфигурации пакета может быть задана общими переменными CPACK_..., а не специфическими переменными генератора, поэтому конфигурация этого генератора может быть достаточно простой. Разработчикам проекта рекомендуется обратиться к документации модуля CPackFreeBSD для получения информации о доступных возможностях и ограничениях.

26.4.10. Cygwin

Еще более простой генератор пакетов - это генератор для Cygwin. По сути, это просто обертка вокруг архива BZip2 и не предлагает практически никакой конфигурации, кроме общих переменных. Проекты могут рассмотреть возможность использования одного из простых форматов архивов.

26.4.11. NuGet

В CMake 3.12 добавлена поддержка формата пакетов NuGet. Новый модуль поддерживает ряд опций, аналогичных другим генераторам пакетов, все они перечислены в документации модуля. В основном эти опции должны быть достаточно понятны, и они следуют тем же шаблонам, что и другие генераторы, уже рассмотренные выше. Разработчикам следует помнить, что поскольку этот генератор пакетов является новым для CMake, он еще не прошел широкого публичного тестирования, поэтому рекомендуется проверить наличие исправлений и обновлений в последующих выпусках CMake.

26.5. Рекомендуемые практики

Одним из первых решений, которое необходимо принять в отношении упаковки, является то, какие форматы пакетов будет предоставлять проект для своих релизов. Хорошей отправной точкой является рассмотрение возможности предоставления по крайней мере одного простого архивного формата, а затем одного "родного" формата для каждой целевой платформы. Архивный формат удобен, когда конечные пользователи хотят установить несколько версий продукта одновременно, поскольку в этом случае они могут просто распаковать архивы релизов в разные каталоги. При условии, что пакеты полностью перемещаемы, это простая и эффективная стратегия. Для наиболее широкой совместимости рекомендуется использовать архивы ZIP для Windows и TGZ для систем на базе Unix.

Различные неархивные форматы подходят в зависимости от целевой платформы. Если для всех платформ подходит программа установки пользовательского интерфейса, то рассмотрите возможность использования генератора IFW для обеспечения единообразной работы конечного пользователя независимо от платформы. Эти инсталляторы также обеспечивают наибольшую настраиваемость, локализацию и возможность загрузки компонентов. Если предпочтение отдается более родным программам установки, то выбор зависит от того, что проект считает более важным. Для Windows могут подойти WIX или NSIS, их возможности довольно схожи. Для Mac многокомпонентный проект может предпочесть генератор productbuild для более чистой установки, но генератор DragNDrop, скорее всего, будет предпочтительнее для конечных пользователей некомпонентных проектов, поскольку он предлагает большую простоту и гибкость. В Linux, если не используется генератор IFW для обеспечения кросс-платформенной совместимости, рассмотрите возможность предоставления пакетов RPM и DEB для наиболее широкого принятия конечными пользователями.

Уделите особое внимание тому, должны ли конечные пользователи иметь возможность установить продукт на систему без головы. Это напрямую влияет как на выбор форматов пакетов, так и на то, как должны быть определены и упакованы компоненты. Для безголовой системы должен быть доступен метод установки без пользовательского интерфейса, а пакеты не должны требовать зависимостей, связанных с пользовательским интерфейсом. Это означает, что компоненты пользовательского интерфейса должны быть отделены от компонентов, не относящихся к пользовательскому интерфейсу. Это особенно важно для форматов пакетов RPM и DEB, где межпакетные зависимости обычно обеспечиваются менеджером пакетов, поэтому пакет компонентов, требующий зависимостей от пользовательского интерфейса, потенциально потянет за собой большое количество нежелательных пакетов, связанных с пользовательским интерфейсом, для безголовой системы.

При определении имен компонентов учитывайте возможность того, что проект может использоваться в качестве дочернего элемента более крупной иерархии проектов. Включите имя проекта в имя компонента, чтобы предотвратить столкновение имен между проектами. Имена компонентов, отображаемые пользователям в установщиках пользовательского интерфейса, именах файлов пакетов и т.д., могут быть заданы по-другому, а не полагаться на имя компонента, используемое внутри проекта CMake. На самом деле, установка пользовательских отображаемых имен и описаний для компонентов поощряется, включая предоставление локализованных значений, если формат пакета поддерживает это.

При задании деталей компонента лучше использовать команды, определенные соответствующими модулями CMake, а не задавать переменные напрямую. Такие команды, как cpack_add_component(), cpack_add_component_group() и т.д., используют именованные аргументы, что делает установку различных опций очень читабельной и более простой в сопровождении. Они также более надежны, поскольку любая ошибка в именах аргументов будет поймана командой, в то время как прямая установка переменных останется незамеченной, если имена переменных будут написаны неправильно.

При настройке деталей для различных генераторов потенциально большое количество переменных может повлиять на способ упаковки содержимого. Во многих случаях значения по умолчанию приемлемы, но некоторые детали всегда должны быть заданы проектом. Проекты должны явно установить все три переменные CPACK_PACKAGE_VERSION_MAJOR, CPACK_PACKAGE_VERSION_MINOR и CPACK_PACKAGE_VERSION_PATCH, так как детали версии по умолчанию редко подходят или не всегда могут быть надежными. Имя пакета, описание и сведения о поставщике также всегда должны быть заданы. Чтобы обеспечить надежную экранировку значений переменных в генерируемых входных файлах, всегда явно устанавливайте CPACK_VERBATIM_VARIABLES в true.

В большинстве случаев проекты хотят избежать включения номера версии в имя каталога установки по умолчанию. Некоторые программы установки поддерживают обновление существующей установки на месте, поэтому любой номер версии в имени каталога будет неуместен после обновления продукта. Пользователи также могут предпочесть, чтобы имя каталога оставалось неизменным при всех обновлениях, чтобы они могли писать сценарии-обёртки, программы запуска и т.д., работающие в разных версиях. Простые архивные пакеты являются исключением из этого правила, поэтому поведение по умолчанию при создании некомпонентных архивов в основном соответствует общепринятой традиции размещения извлеченного содержимого в подкаталоге с соответствующим именем, включающим имя пакета и версию. Для пакетов, основанных на компонентах, проекты захотят установить CPACK_COMPONENT_INCLUDE_TOPLEVEL_DIRECTORY в true, чтобы получить аналогичное поведение.

Пакеты RPM и DEB должны предпочитать устанавливать имена файлов пакетов в RPM-DEFAULT и DEB-DEFAULT соответственно. Это гарантирует, что имена файлов пакетов соответствуют общепринятым соглашениям об именовании, а также является более простым способом включения информации о версии и архитектуре пакета в имена файлов пакетов. Не полагайтесь на имена файлов пакетов RPM и DEB по умолчанию, предоставляемые CPack, поскольку они не содержат сведений о версии и архитектуре.

Если при использовании генератора RPM отладочная информация должна сохраняться для релизных пакетов, рассмотрите возможность использования функциональности debuginfo, а не предотвращения этапа удаления при создании пакета. Предотвращение зачистки требует отключения других потенциально желательных аспектов создания пакетов и требует раскрытия отладочной информации как части пакета релиза. Функциональность debuginfo позволяет предоставить надлежащий пакет выпуска, с отладочными деталями, собранными в отдельном пакете, который может быть распространен или не распространен среди конечных пользователей.

Если из одного вызова cpack необходимо создать пакеты с несколькими архитектурами или отладочные и релизные пакеты, используйте переменную CPACK_INSTALL_CMAKE_PROJECTS для включения компонентов из нескольких деревьев сборки. При использовании такой схемы, всегда устанавливайте компоненты релиза последними, на случай, если компоненты отладки и релиза установят артефакты в одно и то же имя файла и каталог. В идеале этого не должно происходить, но в случаях, когда это может иметь смысл, предпочтительнее будет установить артефакт релиза.

Изучите и поймите возможности настройки пользовательского интерфейса, предоставляемые каждой программой установки пользовательского интерфейса, которую будет поддерживать проект. Настоятельно рекомендуется определить соответствующие значки продукта, чтобы обеспечить профессиональный внешний вид и ощущение. Проекты также должны всегда предоставлять свои собственные readme, приветствия и сведения о лицензии, чтобы текст-заполнитель, предоставляемый CPack, не использовался ни одним из инсталляторов или метаданных пакетов.

Глава 27. Внешнее содержание

Любой проект умеренной сложности, скорее всего, будет опираться на одну или несколько внешних зависимостей. Это могут быть общедоступные наборы инструментов, такие как zlib, OpenSSL, Boost и т.д., частные проекты той же организации или контент для использования в качестве ресурсов, тестовые данные и так далее. В некоторых ситуациях проект может ожидать, что операционная система предоставит все необходимые зависимости. Это будет уместно, например, если проект распространяется как часть этой операционной системы. Для самостоятельных проектов более вероятно, что проект должен контролировать точную версию своих зависимостей, чтобы обеспечить повторяемость сборок и известное происхождение пакетов релизов. Это особенно важно при создании на основе систем непрерывной интеграции, используемых совместно с другими проектами, которые могут иметь другие требования к зависимостям.

CMake предоставляет несколько вариантов того, как привнести внешнее содержимое в сборку. На довольно грубом уровне команда file(DOWNLOAD) может использоваться для получения определенного файла либо на этапе конфигурирования, либо как часть обработки файла CMake в режиме сценария (т.е. cmake -P). Несмотря на то, что это имеет свои преимущества, обычно это не соответствует уровню функциональности, необходимому для включения целых проектов. Для загрузки и сборки целой зависимости традиционным подходом в CMake было использование модуля ExternalProject. Он был частью CMake в течение долгого времени и имеет множество применений, помимо простой загрузки и сборки. Модуль FetchContent, добавленный в CMake 3.11, построен поверх ExternalProject и открывает множество новых возможностей, включая работу с зависимостями, разделяемыми между проектами, и поддержку целых иерархий проектов в одной сборке. Модуль ExternalData предлагает еще одну альтернативу для работы с внешним содержимым во время сборки, уделяя особое внимание данным для тестовых примеров.

27.1. Внешний проект

Основная цель модуля ExternalProject - обеспечить загрузку и сборку внешних проектов, которые не могут быть легко сделаны частью основного проекта напрямую. Внешний проект добавляется как отдельная дочерняя сборка, фактически изолированная от основного проекта и рассматриваемая более или менее как "черный ящик". Это означает, что его можно использовать для сборки проектов для другой архитектуры, других настроек сборки или даже для сборки проекта с системой сборки, отличной от CMake. Его также можно использовать для работы с проектом, который определяет цели или устанавливает компоненты, несовместимые с целями основного проекта.

ExternalProject работает путем определения набора целей сборки в основном проекте, которые представляют отдельные этапы получения и сборки внешнего проекта. Затем они собираются под одной целью CMake, которая представляет всю последовательность действий. Временные метки используются для отслеживания того, какие этапы уже были выполнены и их не нужно повторять, если не изменились соответствующие детали. По умолчанию набор этапов следующий:

Скачать

Для получения исходного кода внешнего проекта можно использовать различные методы. Они включают загрузку архива с URL и его автоматическую распаковку, или клонирование/выборку из репозитория исходного кода, такого как git, subversion, mercurial или CVS. Кроме того, проекты могут определить свои собственные команды, если ни один из поддерживаемых вариантов загрузки не подходит.

Обновление/патч

После загрузки исходного кода к нему может быть применен патч (в случае загрузки архива) или он может быть обновлен (для хранилищ исходного кода). При необходимости можно предоставить пользовательские команды, чтобы отменить поведение по умолчанию.

Настроить

Если внешний проект использует CMake в качестве системы сборки, этот шаг выполняет cmake на загруженном исходном коде. Некоторая информация передается из основной сборки, чтобы сделать конфигурирование внешних проектов CMake достаточно простым. Для внешних проектов, не использующих CMake, можно задать собственную команду для выполнения эквивалентных шагов, например, запустить сценарий configure с соответствующими параметрами.

Построить

По умолчанию сконфигурированный внешний проект собирается с помощью того же инструмента сборки, что и основной проект, если для настройки сборки использовался CMake. На этапе сборки можно задать пользовательские команды для использования другого инструмента сборки или для выполнения другой задачи.

Установите

Внешний проект может быть установлен в локальный каталог, обычно в дерево сборки основного проекта. Тогда основной проект знает, где ожидать артефакты сборки внешнего проекта, и может включить их в свою сборку. Поведение по умолчанию зависит от того, предполагал ли этап configure вызов сборки CMake или нет.

Тест

Внешний проект может поставляться с собственным набором тестов, которые основной проект может выполнять, а может и не выполнять. Модуль ExternalProject обеспечивает гибкость в том, нужно ли запускать этап тестирования (по умолчанию нет) и должен ли он быть до или после этапа установки. Если этап тестирования включен, предполагается, что во внешнем проекте существует цель тестирования по умолчанию, но можно задать пользовательские команды, чтобы обеспечить полный контроль над тем, что делает этап тестирования.

Модуль позволяет определять другие пользовательские этапы и вставлять их в любой момент вышеописанного рабочего процесса, но набор этапов по умолчанию обычно достаточен для большинства проектов. Детали для этапов по умолчанию задаются основной функцией модуля, ExternalProject_Add(). Эта функция принимает множество опций, все из которых подробно описаны в документации модуля. Ниже приведены наиболее часто используемые из них и некоторые типичные сценарии, чтобы помочь читателю сориентироваться в том, как максимально использовать возможности ExternalProject.

27.1.1. Обзор основных функций

В простейшем случае необходимо загрузить архив с исходным кодом по URL и собрать его как проект CMake. Минимальная информация, необходимая для этого, - только URL, который предоставляется следующим образом:

```
include(ExternalProject)
ExternalProject_Add(someExtProj
    URL http://somecompany.com/releases/myproj_1.2.3.tar.gz
)
```

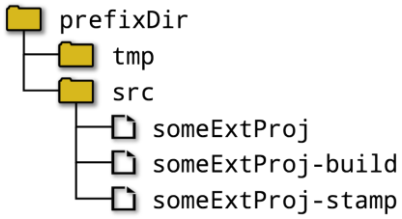
Первым аргументом функции всегда является имя цели сборки, которая должна быть создана в основном проекте. Эта цель будет использоваться для обращения ко всему процессу сборки внешнего проекта. По умолчанию она добавляется к цели all основного проекта, но это можно отключить, добавив обычную опцию EXCLUDE_FROM_ALL, которая имеет тот же эффект, что и для команд add_executable(), add_custom_target() и т.д. В приведенном выше примере сборка цели someExtProj приведет к тому, что на этапе сборки основного проекта будет выполнено следующее:

- Загрузите tarball и распакуйте его.
- Запустите cmake с параметрами по умолчанию на основе основной сборки.
- Вызовите тот же инструмент сборки, что и в основном проекте, для цели по умолчанию.
- Соберите установочную цель внешнего проекта.

Все эти шаги используют отдельный набор каталогов, созданных в каталоге сборки для хранения исходных текстов, результатов сборки, временных меток и других временных файлов, связанных со сборкой внешнего проекта. Структура этих каталогов зависит от нескольких различных факторов, и документация модуля предоставляет подробное объяснение того, как выбирается структура каталогов. Проще начать с того, чтобы показать, как основной проект может управлять расположением каталогов, а не полагаться на значения по умолчанию. Базовое расположение каталогов может быть задано с помощью опции PREFIX.

```
ExternalProject_Add(someExtProj
    PREFIX prefixDir
    URL    http://somecompany.com/releases/myproj_1.2.3.tar.gz
)
```

При использовании этого способа расположение каталогов будет основано на prefixDir, который обычно должен быть представлен в виде абсолютного пути и обычно находится где-то в области сборки основного проекта. Ниже показана относительная схема каталогов по умолчанию, созданная в этом месте. Распакованный архив будет находиться в prefixDir/src/someExtProj, а сборка CMake будет использовать prefixDir/src/someExtProj-build в качестве каталога сборки.



Свойства каталогов EP_PREFIX и EP_BASE могут быть установлены для влияния на вышеуказанную схему, подробнее см. в документации по ExternalProject. Префикс и эти свойства каталога обеспечивают только грубый контроль над структурой каталога. Для тех случаев, когда это необходимо,
ExternalProject_Add() позволяет напрямую задать некоторые или все отдельные каталоги:

```
ExternalProject_Add(someExtProj
    DOWNLOAD_DIR downloadDir
    SOURCE_DIR   sourceDir
    BINARY_DIR   binaryDir
    INSTALL_DIR  installDir
    TMP_DIR      tmpDir
    STAMP_DIR    stampDir
    URL          http://somecompany.com/releases/myproj_1.2.3.tar.gz
)
```

На практике TMP_DIR и STAMP_DIR редко используются, но остальные имеют более прямое отношение к основному проекту и иногда указываются. Место установки по умолчанию будет зависеть от внешнего проекта, который обычно является общесистемным, поэтому очень часто указывается INSTALL_DIR, чтобы облегчить сбор всех конечных артефактов внешних проектов в одном месте в каталоге сборки (чтобы заставить внешние проекты использовать указанный INSTALL_DIR, требуются дополнительные шаги, как будет показано в последующих примерах).

Другая полезная техника - указать SOURCE_DIR и указать местоположение существующего каталога, который уже заполнен. При использовании этого способа не нужно указывать метод загрузки, в этом случае команда просто использует существующее содержимое указанного каталога исходников. Это может быть очень удобным способом сборки части дерева исходных текстов основного проекта для другой платформы. Например:

```
ExternalProject_Add(firmware
    SOURCE_DIR  ${CMAKE_CURRENT_LIST_DIR}/firmware
    INSTALL_DIR ${CMAKE_CURRENT_BINARY_DIR}/firmware-artifacts
    #... other options to configure differently
)
```

Если внешний проект также использует CMake в качестве системы сборки, может оказаться желательным добавить опции командной строки cmake, чтобы повлиять на его конфигурацию. Самый прямой способ добиться этого - использовать опцию CMAKE_ARGS, за которой должны следовать аргументы, передаваемые команде cmake внешнего проекта. Приведенный выше пример может быть расширен для использования файла toolchain, настройки сборки релиза и использования назначенного каталога установки следующим образом:

```
ExternalProject_Add(firmware
    SOURCE_DIR  ${CMAKE_CURRENT_LIST_DIR}/firmware
    INSTALL_DIR ${CMAKE_CURRENT_BINARY_DIR}/firmware-artifacts
    CMAKE_ARGS  -D CMAKE_TOOLCHAIN_FILE=${CMAKE_CURRENT_LIST_DIR}/fwtoolchain.cmake
                -D CMAKE_BUILD_TYPE=Release
                -D CMAKE_INSTALL_PREFIX=<INSTALL_DIR>   # See further below
)
```

Если необходимо задать более пары опций CMake, длина генерируемой командной строки cmake может стать проблемой. Альтернативным вариантом является указание переменных кэша, которые должны быть определены с помощью CMAKE_CACHE_ARGS, вместо их определения через CMAKE_ARGS. Ожидается, что эти аргументы будут иметь вид -Dvariable:TYPE=value и будут преобразованы в файл, содержащий команды вида set(значение переменной CACHE TYPE "" FORCE). Этот файл затем передается в командную строку cmake с

опцией -C. Эффект будет таким же, как если бы переменные были заданы непосредственно в командной строке cmake с помощью опции -D. Существуют и другие опции, которые можно использовать для изменения генератора CMake и некоторых других менее распространенных аспектов вызова CMake, но они используются реже. За более подробной информацией обращайтесь к документации модуля.

Если внешний проект не использует CMake в качестве системы сборки, опция CONFIGURE_COMMAND может быть задана для предоставления альтернативной пользовательской команды, которая будет выполняться вместо запуска cmake. Например, многие проекты предоставляют сценарий configure, который может быть настроен следующим образом:

```
ExternalProject_Add(someAutotoolsProj
    URL               someUrl
    CONFIGURE_COMMAND <SOURCE_DIR>/configure
    ...
)
```

Команда configure выполняется в каталоге сборки, но сценарий configure будет находиться в каталоге исходных текстов. Вместо того, чтобы явно определять расположение каталогов для внешнего проекта, выше продемонстрирована альтернативная стратегия, при которой используется структура по умолчанию, но поддержка команды *placeholder* обеспечивает расположение исходного каталога. В предыдущем примере также использовался заполнитель для каталога установки, переданный в качестве значения для CMAKE_INSTALL_PREFIX. Замещающее устройство - это просто имя опции для определенного каталога, окруженное угловыми скобками. Наиболее часто используются <SOURCE_DIR>, <BINARY_DIR> и <INSTALL_DIR>. <DOWNLOAD_DIR> также доступно в CMake 3.11 или более поздней версии. Полный список заполнителей приведен в документации к модулю.

Если опция CONFIGURE_COMMAND не используется, предполагается, что проект является сборкой CMake, и шаг сборки внешнего проекта будет использовать тот же инструмент сборки, что и основной проект. Для таких случаев подходит поведение шага сборки по умолчанию, и никакой специальной обработки не требуется. Если указано CONFIGURE_COMMAND, предполагается, что инструментом сборки по умолчанию является make, а командой сборки по умолчанию является вызов make без явного указания цели. Если вместо этого должна быть собрана цель не по умолчанию или требуется инструмент сборки, отличный от make, необходимо указать пользовательскую команду сборки. Например:

```
find_program(MAKE_EXECUTABLE NAMES nmake gmake make)
ExternalProject_Add(someAutotoolsProj
    URL               someUrl
    CONFIGURE_COMMAND <SOURCE_DIR>/configure
    BUILD_COMMAND     ${MAKE_EXECUTABLE} specialTool
)
```

Пользовательская команда сборки может делать что угодно, это не обязательно должен быть известный инструмент сборки. Она даже может быть установлена в пустую строку, чтобы эффективно обойти этап сборки. Предсказуемо, та же картина продолжается и на этапе установки. Для проектов CMake инструмент сборки основного проекта будет вызван для сборки цели под названием install по умолчанию, в то время как для проектов, не относящихся к CMake, командой по умолчанию будет просто make install. Опция INSTALL_COMMAND может быть использована для задания пользовательской команды установки или может быть установлена в пустую строку для полного отключения этапа установки. Это часто используется, когда основной проект может использовать результаты этапа сборки без необходимости дальнейшей установки.

```
ExternalProject_Add(someAutotoolsProj
    URL               someUrl
    CONFIGURE_COMMAND <SOURCE_DIR>/configure
    BUILD_COMMAND     ${MAKE_EXECUTABLE} specialTool
    INSTALL_COMMAND   ""   # Effectively disable the install stage
)
```

Следует позаботиться о том, чтобы правильно обработать этап установки. Если внешний проект использует CMake в качестве системы сборки, назначение правила установки по умолчанию контролируется кэш-переменной CMAKE_INSTALL_PREFIX. Если эта переменная не установлена, будет использоваться местоположение по умолчанию, что обычно приводит к установке внешнего проекта в общесистемное местоположение, что обычно не является желаемым результатом (конечно, если проект собирается в системе непрерывной интеграции). Аналогично, если внешний проект использует систему сборки, отличную от CMake, то командой установки по умолчанию будет make install, которая опять же попытается установить проект в общесистемное расположение. В случае с CMake, установка переменной cache через CMAKE_ARGS, как показано в предыдущем примере, решает эту ситуацию, в то время как для проекта, основанного на Makefile, обычно подходит что-то вроде следующего:

```
ExternalProject_Add(otherProj
    URL               ...
    INSTALL_DIR       ${CMAKE_CURRENT_BINARY_DIR}/otherProj-install
    CONFIGURE_COMMAND <SOURCE_DIR>/configure
    INSTALL_COMMAND   ${MAKE_EXECUTABLE} DESTDIR=<INSTALL_DIR> install
)
```

Опция INSTALL_DIR не делает ничего, кроме определения значения для заполнителя <INSTALL_DIR>. Вызывающая сторона должна использовать местодержатель <INSTALL_DIR> для передачи этой информации туда, где она необходима. Проекты должны использовать INSTALL_DIR для определения местоположения и затем использовать местодержатель <INSTALL_DIR>, а не встраивать путь непосредственно в опции типа INSTALL_COMMAND. Это гарантирует, что местоположение может быть запрошено позже, если потребуется, как описано ниже в Разд. 27.1.3, "Различные возможности".

Этап тестирования обрабатывается несколько иначе и по умолчанию ничего не делает. Чтобы включить его, необходимо указать одну из специфических для тестирования опций, например, TEST_BEFORE_INSTALL YES или TEST_AFTER_INSTALL YES. После включения, схема работы такая же, как и на этапах сборки и установки: соответствующий инструмент сборки по умолчанию вызывает тестовую цель, но для обеспечения альтернативного поведения может быть задана команда TEST_COMMAND.

Конечно, ExternalProject имеет значительно большую поддержку загрузки, чем просто базовый URL для загрузки. Для архивов он поддерживает передачу хэша загружаемого файла в основной проект. Это не только имеет очевидное преимущество проверки загружаемого содержимого, но и позволяет модулю проверить файл, который он мог загрузить ранее, и избежать повторной загрузки, если он знает, что у него уже есть файл с правильным хэшем. Хеш-значение может быть любым алгоритмом, который поддерживает команда file(), но обычно это MD5 или SHA1. Хеш задается с помощью параметра URL_HASH, как показано в следующем примере:

```
ExternalProject_Add(someAutotoolsProj
    URL      someUrl
    URL_HASH MD5=b4a78fe5c9f2ef73cd3a6b07e79f2283
    #... other options
)
```

Настоятельно рекомендуется указывать хэш. CMake выдаст предупреждение, если опция URL используется без сопутствующей опции URL_HASH (в качестве особого случая для поддержания обратной совместимости со старыми версиями CMake можно использовать опцию URL_MD5 для предоставления хэша MD5, но проекты должны избегать ее в пользу более гибкой опции URL_HASH).

Также можно указать более одного URL-адреса и позволить проекту поочередно пробовать каждый из них, пока один не увенчается успехом. Это может быть полезно, когда доступные серверы для подключения могут меняться в зависимости от сетевого подключения, настроек VPN и т.д. или для того, чтобы попробовать локальные серверы перед потенциально более медленными удаленными серверами. Эта функция не может быть использована с урлами file://.

```
ExternalProject_Add(someProj
    URL      http://mirrors.mycompany.com/releases/someproj-1.2.3.tar.gz
             https://somewhereelse.com/artifacts/someproj-1.2.3.tar.gz
    URL_HASH MD5=b4a78fe5c9f2ef73cd3a6b07e79f2283
    #... other options
)
```

При загрузке архивов формат архива определяется на основе содержимого файла после загрузки, и архив распаковывается автоматически. При необходимости автоматическую распаковку можно отключить, а также управлять различными аспектами настройки самой загрузки. Подробности о соответствующих опциях для этих менее распространенных сценариев см. в документации к модулю.

Загружаемое содержимое не обязательно должно быть из архива, модуль также может работать непосредственно с репозиториями исходного кода git, subversion, mercurial или CVS. Каждый из них требует, чтобы репозиторий был назван с помощью опции <REPOTYPE>_REPOSITORY, а затем могут быть заданы другие опции, специфичные для репозитория.

```
ExternalProject_Add(myProj
    GIT_REPOSITORY git@somecompany.com/git/myproj.git
    GIT_TAG        3a281711d1243351190bdee50a40d81694aa630a
)
```

В приведенном выше примере показана типичная информация, необходимая для клонирования git-репозитория и проверки определенного коммита. Если опция GIT_TAG опущена, будет использован последний коммит в ветке по умолчанию (обычно master). Имя тега или ветки также может быть задано с помощью GIT_TAG вместо хэша коммита. Хотя GIT_TAG поддерживает эти различные варианты, следует отметить, что только хэш коммита является действительно однозначным. В git коммит, на который ссылается имя ветки или тега, может меняться со временем, поэтому их использование не гарантирует повторяемость сборки. Аналогично, полное отсутствие GIT_TAG равносильно указанию имени ветви по умолчанию, поэтому оно тоже не всегда будет указывать на один и тот же коммит.

Есть еще одна причина использовать хэши коммитов только с GIT_TAG. Поскольку имя тега или ветки может меняться со временем, ExternalProject_Add() должен будет обращаться к удалённой части при каждом запуске CMake, даже если у него уже есть клонированный и проверенный тег или ветка. Это происходит потому, что она не может быть уверена, что тег или ветвь не переместились без получения данных с удаленного узла. Это путешествие туда-сюда при каждом повторном запуске CMake может быть дорогостоящим, особенно если проект использует много внешних проектов. Если вместо этого используется хэш коммита, то ExternalProject_Add() может определить, есть ли у него уже локальный коммит, без необходимости обращаться к удаленному. Таким образом, после успешного получения коммита не требуется дальнейшее сетевое соединение для последующих запусков CMake.

Другие опции могут быть использованы для настройки поведения git, включая указание другого удаленного имени по умолчанию, управление подмодулями git, неглубокими клонами и произвольными опциями конфигурации git. За более подробной информацией обращайтесь к документации модуля.

Проверка из репозитория subversion довольно похожа на git:

```
ExternalProject_Add(myProj
    SVN_REPOSITORY svn+ssh@somecompany.com/svn/myproj/trunk
    SVN_REVISION   -r31227
)
```

Опция SVN_REVISION определяет опцию командной строки svn, которая, как ожидается, указывает коммит для проверки. Часто это будет глобальный номер ревизии, указанный с помощью опции -r, как показано выше, но технически это может быть любая допустимая опция командной строки. Если SVN_REVISION опущен, используется последняя ревизия, но проекты должны стремиться всегда предоставлять эту опцию для обеспечения повторяемости сборки. Несколько других опций subversion, связанных с безопасностью, поддерживаются следующими параметрами
ExternalProject_Add(), например, для аутентификации в хранилище и указания параметров доверия сертификату. Обратитесь к документации модуля ExternalProject для получения подробной информации об этих менее часто используемых опциях.

Для сравнения, поддержка Mercurial и CVS очень базовая. В случае Mercurial можно указать только репозиторий и тег, а для CVS требуется еще и модуль:

```
ExternalProject_Add(myProjHg
    HG_REPOSITORY  http://somecompany.com/hg/myproj
    HG_TAG         dd2ce38a6b8a
)
ExternalProject_Add(myProjCVS
    CVS_REPOSITORY http://somecompany.com/cvs/myproj
    CVS_MODULE     someModule
    CVS_TAG        -rsomeTag
)
```

Опция CVS_TAG аналогична опции SVN_REVISION в том, что она размещается непосредственно в командной строке cvs, поэтому она должна включать любой требуемый префикс командной опции, как показано выше.

27.1.2. Управление шагами

Иногда бывает полезно или даже необходимо обратиться к одному из шагов в последовательности ExternalProject, например, чтобы добавить зависимость от другой цели CMake, которая обеспечивает вход для конкретного шага. Опция STEP_TARGETS может быть передана ExternalProject_Add(), чтобы указать ей создать цели CMake для указанного набора шагов. Эти цели имеют имена вида mainName-step, где mainName - имя цели, заданное в качестве первого аргумента ExternalProject_Add(), а step - шаг, который представляет цель. Например, в результате следующих действий будут определены цели с именами myProjconfigure и myProj-install:

```
ExternalProject_Add(myProj
    GIT_REPOSITORY git@somecompany.com/git/myproj.git
    GIT_TAG        3a281711d1243351190bdee50a40d81694aa630a
    STEP_TARGETS   configure install
)
```

Добавление зависимостей для этих пошаговых целей требует немного большей осторожности. Чтобы сделать цель шага зависимой от какой-то другой цели CMake, проект должен использовать функцию ExternalProject_Add_StepDependencies(), предоставляемую модулем, а не вызывать add_dependencies(). Это гарантирует, что такие вещи, как временные метки шагов, будут обработаны правильно. Эта команда выглядит следующим образом:

```
ExternalProject_Add_StepDependencies(mainName step otherTarget1...)
```

В следующем примере показано, как использовать эту функцию, чтобы шаг configure предыдущего примера зависел от исполняемого файла, собранного главным проектом:

```
add_executable(preConfigure ...)
ExternalProject_Add_StepDependencies(myProj configure preConfigure)
```

Чтобы сделать обычную цель CMake зависимой от цели step, достаточно использовать add_dependencies():

```
add_executable(postInstall ...)
add_dependencies(postInstall myProj-install)
```

Если конкретный шаг одного внешнего проекта должен зависеть от шага другого внешнего проекта, необходимо снова использовать ExternalProject_Add_StepDependencies():

```
ExternalProject_Add(earlier
    STEP_TARGETS build
    ...
)
ExternalProject_Add(later
    STEP_TARGETS build
    ...
)
ExternalProject_Add_StepDependencies(later build earlier-build)
```

Вышеописанная схема может быть полезна, если более ранний проект определяет тесты, выполнение которых требует много времени, но при параллельной сборке более позднему проекту не нужно ждать этих тестов, а только сборки более раннего.

Когда один и тот же набор целей шага должен быть определен для нескольких внешних проектов, вместо того, чтобы повторять их каждый раз, их можно сделать по умолчанию, установив вместо этого свойство каталога EP_STEP_TARGETS.

```
set_property(DIRECTORY PROPERTY EP_STEP_TARGETS build)
ExternalProject_Add(earlier ...)
ExternalProject_Add(later ...)
ExternalProject_Add_StepDependencies(later build earlier-build)
```

Однако для многих проектов такая детализация зависимостей дает лишь ограниченный выигрыш, и сложность может того не стоить. Весь внешний проект можно сделать зависимым от другой цели с помощью опции DEPENDS в ExternalProject_Add(), что гораздо проще:

```
add_executable(preConfigure ...)
ExternalProject_Add(myProj
    DEPENDS preConfigure
    ...
)
```

Опция DEPENDS заботится о том, чтобы все зависимости шага обрабатывались правильно, как это делает ExternalProject_Add_StepDependencies() при установке более детальных зависимостей.

Проекты не ограничены только шагами по умолчанию, они могут создавать свои собственные пользовательские шаги и вставлять их в рабочий процесс шагов с любыми зависимостями, которые они требуют. Функция ExternalProject_Add_Step() предоставляет такую возможность:

```
ExternalProject_Add_Step(mainName step
    [COMMAND           command [args...] ]
    [COMMENT           comment]
    [WORKING_DIRECTORY dir]
    [DEPENDS           filesWeDependOn...]
    [DEPENDEES         stepsWeDependOn...]
    [DEPENDERS         stepsDependOnUs...]
    [BYPRODUCTS        byproducts...]
    [ALWAYS            bool]
    [EXCLUDE_FROM_MAIN bool]
    [LOG               bool]
    [USES_TERMINAL     bool]
)
```

COMMAND используется для определения действия, которое будет предпринято при выполнении шага. Она аналогична пользовательской команде, которая может быть задана для каждого из шагов по умолчанию. COMMENT может быть задан, чтобы предоставить пользовательское сообщение при выполнении шага, но, как отмечалось в разделе 17.1, "Пользовательские цели", такие комментарии не всегда отображаются, поэтому

считайте их полезными, но не обязательными. Параметр WORKING_DIRECTORY имеет то же значение, что и для команды add_custom_target().

Исчерпывающая информация о зависимостях может быть предоставлена с помощью пользовательского шага. Если команда зависит от определенного файла или набора файлов, их следует перечислить с помощью параметра DEPENDS. Для файлов, создаваемых в процессе сборки, они должны быть созданы пользовательскими командами, созданными в той же области каталогов. Опции DEPENDEES и DEPENDERS определяют место данного пользовательского шага в рабочем процессе шагов внешнего проекта. Необходимо полностью указать все прямые зависимости, иначе пользовательский шаг может выполняться не по порядку. Опция BYPRODUCTS также должна использоваться, если пользовательский шаг создает файл, от которого зависит что-то еще во внешнем или основном проекте. Без этого генератор Ninja, скорее всего, будет жаловаться на отсутствие правила сборки.

Пользовательский шаг можно заставить всегда отображаться устаревшим, установив опцию ALWAYS в true. Обычно проекты должны делать это только в том случае, если от него не зависит ни один другой шаг, поскольку все, что зависит от него, будет всегда считаться устаревшим, а это может привести к тому, что сборки будут выполнять больше работы, чем нужно. Если пользовательский шаг предназначен для сборки только по требованию, то установка ALWAYS и EXCLUDE_FROM_MAIN в true обычно является желательной комбинацией. Остальные опции LOG и USES_TERMINAL обсуждаются в следующем разделе.

Все шаги по умолчанию создаются вызовами ExternalProject_Add_Step() внутри ExternalProject_Add(). Проекты не должны пытаться переопределять их, что означает, что пользовательские шаги не могут иметь имена mkdir, download, update, skip-update, patch, configure, build, install или test.

Действия и межэтапные зависимости определяются функцией ExternalProject_Add_Step(), но для того, чтобы создать цель для пользовательского этапа, необходимо также вызвать функцию ExternalProject_Add_StepTargets(). Эта функция также вызывается ExternalProject_Add() для создания целей для шагов, перечисленных в опции STEP_TARGETS или установленных через свойство каталога EP_STEP_TARGETS.

```
ExternalProject_Add_StepTargets(mainName [NO_DEPENDS] steps...)
```

Опция NO_DEPENDS редко бывает желательной и не рекомендуется для большинства сценариев (более подробно см. обсуждение этой опции в документации модуля). Следующий пример демонстрирует, как определить пользовательский шаг пакета, который зависит от шага сборки, но выполняется только при явном запросе.

```
ExternalProject_Add_Step(myProj package
    COMMAND           ${CMAKE_COMMAND} --build <BINARY_DIR> --target package
    DEPENDEES         build
    ALWAYS            YES
    EXCLUDE_FROM_MAIN YES
)
ExternalProject_Add_StepTargets(myProj package)
```

27.1.3. Различные функции

Для любого из шагов по умолчанию или пользовательских шагов можно указать пользовательскую команду. Для

ExternalProject_Add(), соответствующими опциями являются те, которые заканчиваются на _COMMAND, а для External_Project_Add_Step() именно опция COMMAND обеспечивает выполнение пользовательской команды.

Обе эти функции позволяют задать более одной команды, добавляя дополнительные опции COMMAND, которые следуют непосредственно за первой. Каждая команда выполняется по порядку для данного шага.

```
ExternalProject_Add(myProj
    CONFIGURE_COMMAND ${CMAKE_COMMAND} -E echo "Starting custom configure"
              COMMAND ./configure
              COMMAND ${CMAKE_COMMAND} -E echo "Custom configure completed"
    BUILD_COMMAND     ${CMAKE_COMMAND} -E echo "Starting custom build"
          COMMAND     ${MAKE_EXECUTABLE} mySpecialTarget
          COMMAND     ${CMAKE_COMMAND} -E echo "Custom build completed"
)
ExternalProject_Add_Step(myProj package
    COMMAND           ${CMAKE_COMMAND} -E echo "Starting packaging step"
    COMMAND           ${CMAKE_COMMAND} --build <BINARY_DIR> --target package
    COMMAND           ${CMAKE_COMMAND} -E echo "Packaging completed"
    DEPENDEES         build
    ALWAYS            YES
    EXCLUDE_FROM_MAIN YES
)
ExternalProject_Add_StepTargets(myProj package)
```

Еще одна функция для команд - возможность предоставить им доступ к терминалу, что может быть важно для таких вещей, как доступ к репозиторию, который может потребовать от пользователя предоставления пароля для закрытого ключа и т.д. Хотя это не подходит для сборок непрерывной интеграции, в которых нет терминала, это иногда полезно для разработчиков в их повседневной деятельности. Для шагов по умолчанию доступ к терминалу контролируется на каждом шаге с помощью опций ExternalProject_Add() вида USES_TERMINAL_<STEP> , где <STEP> - это имя шага, написанное с заглавной буквы, а значение опции - это константа true или false. Для пользовательских шагов опция USES_TERMINAL для команды ExternalProject_Add_Step() имеет тот же эффект. Если для загрузки используется репозиторий git или subversion, то рекомендуется предоставить шагам загрузки и обновления доступ к терминалу.

```
ExternalProject_Add(myProj
    GIT_REPOSITORY         git@somecompany.com/git/myproj.git
    GIT_TAG                3a281711d1243351190bdee50a40d81694aa630a
    USES_TERMINAL_DOWNLOAD YES
    USES_TERMINAL_UPDATE   YES
)
```

Шагам следует предоставлять доступ к терминалу только в том случае, если это необходимо. Эффект от этого в основном актуален для генератора Ninja, где пользовательский шаг будет помещен в пул консольных заданий. Все цели, выделенные в консольный пул, будут принудительно выполняться последовательно, а вывод любых заданий, выполняемых параллельно в других пулах заданий, будет буферизироваться до завершения текущего консольного задания. Будьте особенно осторожны и не предоставляйте шагу сборки доступ к терминалу без крайней необходимости, поскольку это может оказать значительное негативное влияние на общую производительность сборки проекта.

В некоторых случаях может быть полезно записывать вывод отдельных шагов в файл, а не отправлять его в терминал (или туда, куда он был перенаправлен). Это особенно полезно, когда имеется большой объем вывода, который будет представлять интерес только в случае ошибки или другой неожиданной проблемы. Чтобы перенаправить вывод шага в файл, установите опцию LOG_<STEP> для ExternalProject_Add() или опцию

LOG для ExternalProject_Add_Step() в значение true. Тогда на выходе терминала будет отображаться только короткое сообщение, указывающее, был ли шаг успешным или нет, и где можно найти файлы журнала, которые будут находиться в каталоге временных меток (т.е. STAMP_DIR).

```
ExternalProject_Add(myProj
    GIT_REPOSITORY git@somecompany.com/git/myproj.git
    GIT_TAG        3a281711d1243351190bdee50a40d81694aa630a
    LOG_BUILD      YES
    LOG_TEST       YES
)
```

В некоторых ситуациях проект может столкнуться с необходимостью узнать, была ли определенная опция передана в ExternalProject_Add() или каково эффективное значение определенной опции. Задатчики, такие как <SOURCE_DIR> и так далее, покрывают многие из распространенных сценариев, когда детали должны быть указаны в вызове ExternalProject_Add(), но для других случаев модуль предоставляет команду ExternalProject_Get_Property(). Ее синтаксис значительно отличается от других команд поиска свойств, таких как get_property():

```
ExternalProject_Get_Property(mainName propertyName...)
```

Имя выходной переменной не задается, вместо этого создается переменная, соответствующая имени извлекаемого свойства. Это позволяет получить несколько свойств за один вызов.

```
ExternalProject_Get_Property(myProj SOURCE_DIR LOG_BUILD)
set(msg "myProj source will be in ${SOURCE_DIR}")
if(LOG_BUILD)
    string(APPEND msg " and its build output will be redirected to log files")
endif()
message(STATUS "${msg}")
```

27.1.4. Общие вопросы

Модуль ExternalProject является мощным и эффективным при правильном использовании, но иногда он также может привести к проблемам, которые трудно отследить. Одна из наиболее часто встречающихся проблем возникает при попытке установить несколько внешних проектов, когда один проект хочет иметь возможность использовать результаты сборки другого. Обычно это требует, чтобы основной проект сделал две вещи:

- Укажите отношения зависимости между двумя проектами.
- Предоставьте проекту "Зависимый" информацию, необходимую для поиска зависимого.

Первый пункт достаточно легко установить, создав зависимость для шага configure Depender от основной цели зависимого объекта. Второй момент требует понимания того, как Depender хочет узнать о местоположении зависимого объекта. Например, если он использует find_package(), find_library() и т.д. для поиска зависимого объекта, то установки его CMAKE_PREFIX_PATH может быть достаточно. Следующий рабочий пример демонстрирует эту технику, собирая и zlib, и libpng как внешние проекты и устанавливая их в один каталог. Поскольку libpng требует zlib, указание общей области установки в CMAKE_PREFIX_PATH позволяет ему найти zlib. Пример гарантирует, что zlib будет установлен до того, как libpng выполнит свой шаг configure, а именно тогда libpng будет использовать CMAKE_PREFIX_PATH.

```
cmake_minimum_required(VERSION 3.0)
project(ExtProjDeps)
include(ExternalProject)

set(installDir ${CMAKE_CURRENT_BINARY_DIR}/install)

ExternalProject_Add(zlib
    INSTALL_DIR ${installDir}
    URL         https://zlib.net/zlib-1.2.11.tar.gz
    URL_HASH    SHA256=c3e5e9fdd5004dcb542feda5ee4f0ff0744628baf8ed2dd5d66f8ca1197cb1a1
    CMAKE_ARGS  -DCMAKE_INSTALL_PREFIX:PATH=<INSTALL_DIR>
)
ExternalProject_Add(libpng
    INSTALL_DIR ${installDir}
    URL         ftp://ftp-osl.osuosl.org/pub/libpng/src/libpng16/libpng-1.6.34.tar.gz
    URL_HASH    MD5=03fbc5134830240104e96d3cda648e71
    CMAKE_ARGS  -DCMAKE_INSTALL_PREFIX:PATH=<INSTALL_DIR>
                -DCMAKE_PREFIX_PATH:PATH=<INSTALL_DIR>
)
ExternalProject_Add_StepDependencies(libpng configure zlib)
```

Вышеописанная схема, когда главный проект не делает ничего, кроме определения набора внешних проектов, часто называется *супербилдом*, тема которого обсуждается в следующей главе.

Еще одна проблема, связанная с зависимостями, которая может возникнуть при использовании генератора Ninja, - это жалобы Ninja на то, что он не знает, как собрать определенный файл, который должен быть предоставлен внешним проектом. Следующий пример демонстрирует такую ситуацию.

```
ExternalProject_Add(myProj
    # Relevant options to download and build a library "someLib"
    ...
)

ExternalProject_Get_Property(myProj INSTALL_DIR)

add_library(MyProj::someLib STATIC IMPORTED)

set_target_properties(MyProj::someLib PROPERTIES
    # Platform-specific due to hard-coded library location and file name
    IMPORTED_LOCATION ${INSTALL_DIR}/lib/libsomeLib.a
)

add_dependencies(MyProj::someLib myProj)
```

Генератор Ninja попытается найти libsomeLib.a в ожидаемом месте, но он еще не будет существовать до того, как внешний проект myProj будет собран в первый раз. Тогда Ninja остановится с ошибкой, говорящей о том, что он не знает, как собрать отсутствующую зависимость. Другие генераторы могут быть более расслабленными в своей проверке зависимостей и не жаловаться, но это не должно считаться подтверждением правильно указанных зависимостей. Решением проблемы является добавление опции BUILD_BYPRODUCTS к вызову ExternalProject_Add() для указания выходов сборки (доступно в CMake 3.2 или более поздней версии). Тогда Ninja будет иметь всю необходимую информацию для удовлетворения своих зависимостей.

```
ExternalProject_Add(myProj
    BUILD_BYPRODUCTS <INSTALL_DIR>/lib/libsomeLib.a
    # Relevant options to download and build the above library
    ...
)
```

Приведенная выше ситуация является примером проблем, возникающих при смешивании ExternalProject с целями, определенными в основном проекте. Это трудно сделать надежно, и обычно приходится вручную указывать детали, специфичные для платформы, которые CMake обычно обрабатывает от имени проекта (например, имена и расположение библиотек). Проектам следует подумать о том, не будет ли более уместным использование суперсборки, и не пытаться создавать собственные цели сборки при использовании ExternalProject.

Проблемы с зависимостями могут возникать и в других ситуациях. Рассмотрим предыдущий пример, в котором ExternalProject использовался для обеспечения возможности сборки артефактов прошивки с инструментарием, отличным от основного.

```
ExternalProject_Add(firmware
    SOURCE_DIR  ${CMAKE_CURRENT_LIST_DIR}/firmware
    INSTALL_DIR ${CMAKE_CURRENT_BINARY_DIR}/firmware-artifacts
    CMAKE_ARGS  -D CMAKE_TOOLCHAIN_FILE=${CMAKE_CURRENT_LIST_DIR}/fwtoolchain.cmake
                -D CMAKE_BUILD_TYPE=Release
                -D CMAKE_INSTALL_PREFIX=<INSTALL_DIR>
)
```

Вышеупомянутая сборка прошла бы успешно, и все было бы в порядке. Если разработчик затем пойдет и внесет изменения в исходные тексты в каталоге исходных текстов прошивки, основной проект не будет пересобирать цели прошивки. Это происходит потому, что ExternalProject использует временные метки для регистрации успешного завершения шагов, поэтому, если что-то не изменится в способе вычисления зависимостей, основной проект будет считать, что проект прошивки все еще актуален. Эту проблему можно решить, заставив цель сборки микропрограммы всегда считаться устаревшей с помощью опции BUILD_ALWAYS:

```
ExternalProject_Add(firmware
    SOURCE_DIR   ${CMAKE_CURRENT_LIST_DIR}/firmware
    INSTALL_DIR  ${CMAKE_CURRENT_BINARY_DIR}/firmware-artifacts
    CMAKE_ARGS   -D CMAKE_TOOLCHAIN_FILE=${CMAKE_CURRENT_LIST_DIR}/fwtoolchain.cmake
                 -D CMAKE_BUILD_TYPE=Release
                 -D CMAKE_INSTALL_PREFIX=<INSTALL_DIR>
    BUILD_ALWAYS YES
)
```

Это приведет к тому, что инструмент сборки проекта микропрограммы будет вызываться каждый раз при сборке основного проекта. Если в проекте микропрограммы ничего не изменилось, его шаг сборки не будет выполняться, но если произошли изменения, то все, что устарело, будет перестроено, как и ожидалось.

27.2. FetchContent

Некоторые из сильных сторон ExternalProject также являются его слабыми сторонами. Он позволяет сборкам внешнего проекта быть полностью изолированными от основного проекта, поэтому он может использовать другой инструментарий, ориентироваться на другую платформу, использовать другой тип сборки или даже совершенно другую систему сборки. Цена этих преимуществ заключается в том, что основной проект ничего не знает о том, что производит внешний проект. Эта информация должна быть предоставлена основной сборке вручную, если что-то в основной сборке должно ссылаться на результаты внешнего проекта. Именно такие вещи должен делать CMake от имени проекта, поэтому использование ExternalProject таким образом может быть обратным шагом.

Для внешних проектов, которые также используют CMake в качестве системы сборки, гибкость сборки с настройками, отличными от настроек основного проекта, часто не нужна. На самом деле, чаще всего внешний проект должен быть собран с теми же настройками, что и основной проект, но это не так просто сделать с помощью ExternalProject. Часто гораздо удобнее добавить его в основную сборку непосредственно с помощью add_subdirectory(), как если бы он был частью собственных исходников основного проекта. Это невозможно сделать при традиционном использовании ExternalProject, поскольку исходники не загружаются до момента сборки. Для решения этой проблемы проекты могут использовать альтернативные стратегии, такие как git-подмодули, но и они не лишены своих недостатков.

Модуль FetchContent был добавлен в CMake 3.11 для решения проблем, подобных упомянутым выше. Он использует ExternalProject для создания подсборки, которая загружает и обновляет внешнее содержимое, но делает это на этапе конфигурирования. Это означает, что загруженный контент доступен немедленно, так что основной проект может затем принести его в основную сборку через add_subdirectory(), использовать его в качестве ресурсов и так далее.

В проектах, которые зависят от многих внешних проектов, иногда бывает так, что эти внешние проекты, в свою очередь, имеют некоторые общие зависимости. Было бы нежелательно загружать и собирать эти общие зависимости несколько раз, но ExternalProject сам по себе не предоставляет прямого способа справиться с этим. Модуль FetchContent предлагает решение и для этого сценария. Он позволяет определять детали зависимостей внешних проектов отдельно от команды, которая используется для инициирования загрузки. При первом определении деталей загрузки для данной зависимости они сохраняются внутри проекта, и все последующие попытки определить их игнорируются. Когда проект просят заполнить зависимость, он использует эти сохраненные данные, и любые другие части проекта могут просто повторно использовать это содержимое вместо того, чтобы загружать его снова. Этот подход "первый установивший побеждает" означает, что родительский проект может переопределять детали зависимостей внешних дочерних проектов, подтягиваемых через add_subdirectory().

Каноническая схема использования модуля FetchContent продемонстрирована в следующем примере:

```
include(FetchContent)
FetchContent_Declare(googletest  ①
    GIT_REPOSITORY https://github.com/google/googletest.git
    GIT_TAG        ec44c6c1675c25b9827aacd08c02433cccde7780  # release-1.8.0
)

FetchContent_GetProperties(googletest)  ②
if(NOT googletest_POPULATED)
    FetchContent_Populate(googletest)
    add_subdirectory(${googletest_SOURCE_DIR} ${googletest_BINARY_DIR})  ③
endif()
```

① Запишите сведения о том, откуда следует получить GoogleTest. Если где-то в проекте это уже сделано, заявленные здесь сведения будут проигнорированы.

② Заполните содержимое GoogleTest, но только если это еще не сделала какая-то другая часть проекта.

③ Всегда указывайте xxx_SOURCE_DIR и xxx_BINARY_DIR в add_subdirectory(). Если xxx_SOURCE_DIR указывает на местоположение не в текущем бинарном каталоге (что обычно и происходит), add_subdirectory() требует, чтобы был указан и соответствующий бинарный каталог.

Команда FetchContent_Declare() требует, чтобы ее первым аргументом было имя объявляемой зависимости (это имя обрабатывается без учета регистра). Аргументы, следующие за именем, должны быть любыми из опций, поддерживаемых ExternalProject_Add(), за исключением тех, которые относятся к шагам configure, build, install или test. На практике обычно указываются только те опции, которые определяют метод загрузки, например, детали git в примере GoogleTest выше.

Команда FetchContent_GetProperties() позволяет проекту проверить, была ли уже заполнена определенная зависимость, а также получить некоторые сведения о каталоге. Полная форма команды выглядит следующим образом:

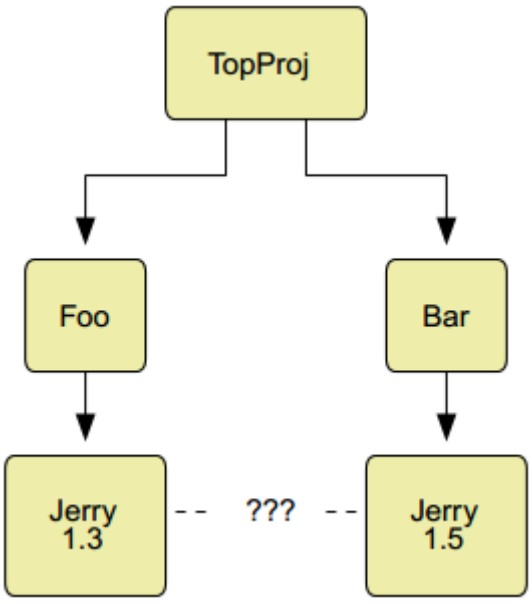
```
FetchContent_GetProperties(name
    [SOURCE_DIR srcDirVar]
    [BINARY_DIR binDirVar]
    [POPULATED  doneVar]
)
```

Опции SOURCE_DIR, BINARY_DIR и POPULATED могут быть использованы для указания имени переменной, в которой будет храниться связанное свойство для зависимости имени. Если ни одна из этих опций не задана, то команда устанавливает переменные <lcname>_SOURCE_DIR, <lcname>_BINARY_DIR и <lcname>_POPULATED в области видимости вызывающей стороны, где <lcname> - имя, преобразованное в нижний регистр. Дополнительные аргументы не нужны, если используется канонический шаблон.

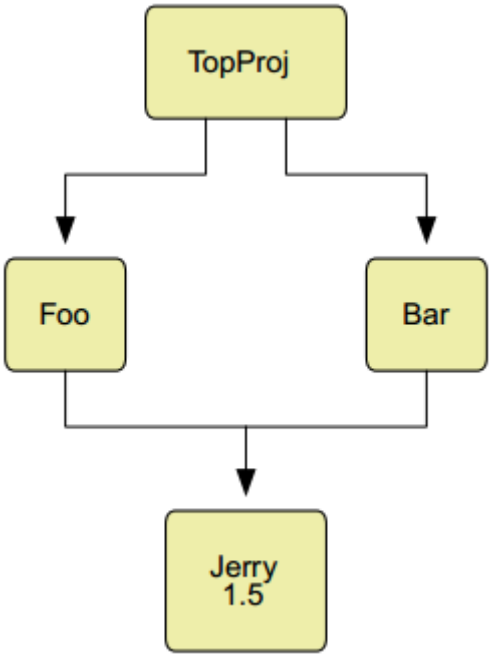
Свойство POPULATED будет истинным, если FetchContent_Populate() уже была вызвана где-то в проекте для указанного имени. Если оно истинно, то свойство SOURCE_DIR указывает, где можно найти загруженное содержимое. Поскольку загруженное содержимое может не быть непосредственным подкаталогом в месте вызова FetchContent_Populate(), свойство BINARY_DIR также почти всегда необходимо для вызова add_subdirectory().

Если FetchContent_GetProperties() подтвердила, что указанное содержимое еще не заполнено, можно вызвать команду FetchContent_Populate(), чтобы выполнить заполнение содержимого. При использовании в составе проекта с канонической формой, показанной выше, она принимает только один аргумент - имя зависимости, которую нужно заполнить. Используя сохраненные данные, объявленные ранее, содержимое будет заполнено, если оно еще не было заполнено предыдущим запуском cmake.
Переменные <lcname>_POPULATED, <lcname>_SOURCE_DIR и <lcname>_BINARY_DIR также будут установлены в области видимости вызывающей стороны точно так же, как и при вызове FetchContent_GetProperties(name).

Следующий пример показывает, как модуль FetchContent позволяет проекту верхнего уровня переопределять детали, установленные зависимостями нижнего уровня. Рассмотрим проект верхнего уровня TopProj, который зависит от внешних проектов Foo и Bar. И Foo, и Bar, в свою очередь, зависят от другого внешнего проекта, Jerry, но каждому из них нужны немного разные версии.



Необходимо загрузить и собрать только одну копию Jerry, которую затем будут использовать Foo и Bar. Когда эти проекты объединяются в одну сборку, выбранная версия Jerry должна переопределять версию, обычно используемую Foo или Bar, или, возможно, даже обе. Проект верхнего уровня отвечает за обеспечение выбора правильной версии, чтобы Foo и Bar могли собирать на ее основе. В этом примере предполагается, что хотя Foo использует версию 1.3 при самостоятельной сборке, он может безопасно использовать более позднюю версию. Желаемая схема и пример, реализующий ее, выглядят следующим образом:



TopProj CMakeLists.txt

```
# Declare the direct dependencies
include(FetchContent)
FetchContent_Declare(foo GIT_REPOSITORY ... GIT_TAG ...)
FetchContent_Declare(bar GIT_REPOSITORY ... GIT_TAG ...)

# Override the Jerry dependency to ensure we get what we want
FetchContent_Declare(jerry
    URL      https://somecompany.com/releases/jerry-1.5.tar.gz
    URL_HASH ...
)

# Populate the direct dependencies but leave Jerry to be populated by foo
FetchContent_GetProperties(foo)
if(NOT foo_POPULATED)
    FetchContent_Populate(foo)
    add_subdirectory(${foo_SOURCE_DIR} ${foo_BINARY_DIR})
endif()
FetchContent_GetProperties(bar)
if(NOT bar_POPULATED)
    FetchContent_Populate(bar)
    add_subdirectory(${bar_SOURCE_DIR} ${bar_BINARY_DIR})
endif()
```

Foo CMakeLists.txt

```
include(FetchContent)
FetchContent_Declare(jerry
    URL      https://somecompany.com/releases/jerry-1.3.tar.gz
    URL_HASH ...
)
FetchContent_GetProperties(jerry)
if(NOT jerry_POPULATED)
    FetchContent_Populate(jerry)
    add_subdirectory(${jerry_SOURCE_DIR} ${jerry_BINARY_DIR})
endif()
```

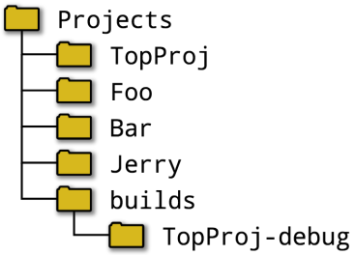
Файл CMakeLists.txt для Bar будет идентичен файлу Foo, за исключением того, что в URL будет указан jerry1.5.tar.gz вместо jerry-1.3.tar.gz. Приведенный выше пример позволяет собирать Foo и Bar как самостоятельные проекты, или они могут быть включены в другой проект, например TopProj, и при этом иметь необходимую гибкость для разрешения общих зависимостей.

27.2.1. Переопределения разработчика

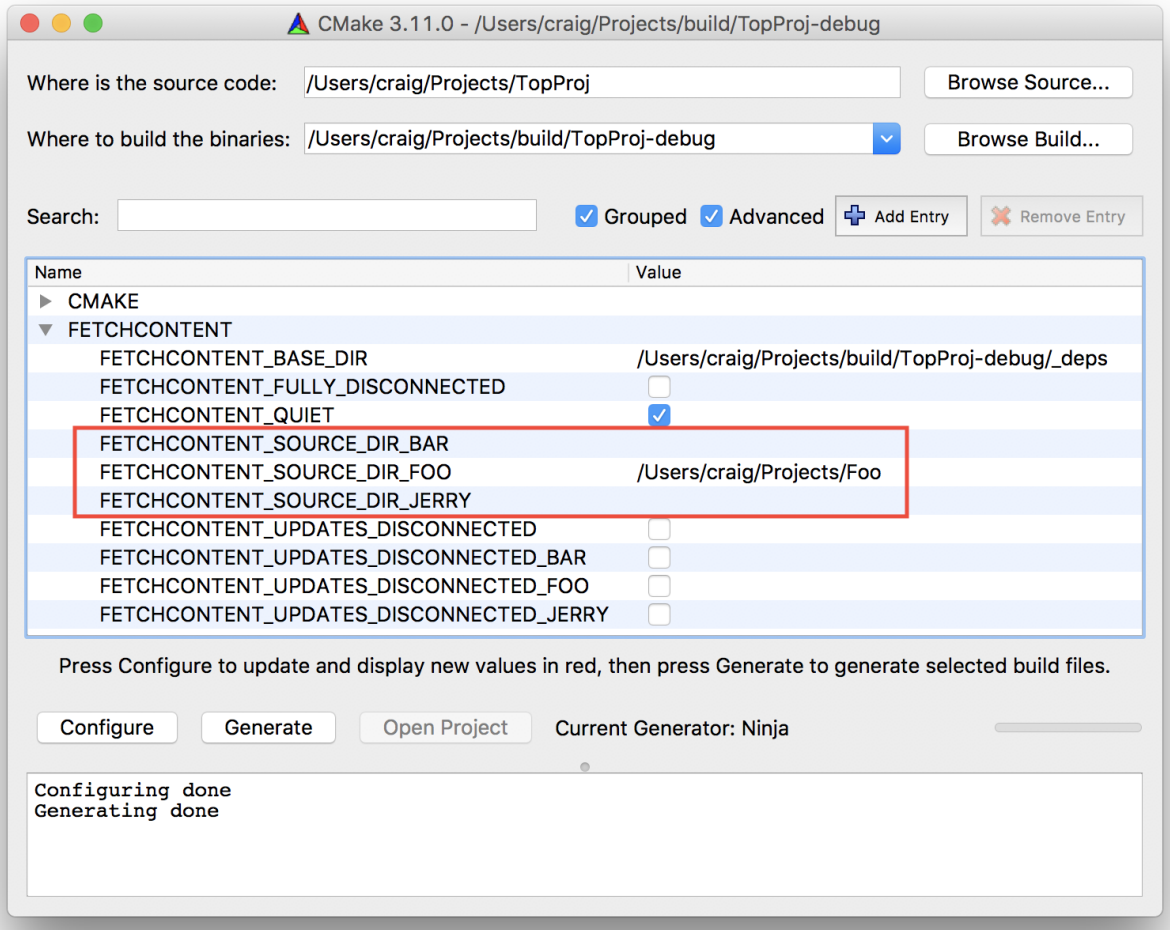
Бывают случаи, когда разработчик хочет работать над несколькими проектами одновременно, внося изменения как в основной проект, так и в его зависимости, или в несколько зависимостей и т.д. При изменении частей внешнего проекта разработчик захочет работать с локальной копией, а не обновлять каждый раз удаленное место, с которого она загружается. Модуль FetchContent предлагает прямую поддержку такого режима работы, позволяя переопределить исходный каталог любой внешней зависимости с помощью переменной кэша CMake. Эти переменные имеют имена вида FETCHCONTENT_SOURCE_DIR_<DEPNAME>, где <DEPNAME> - имя зависимости в верхнем регистре.

В предыдущем примере рассмотрим ситуацию, когда разработчик хочет внести изменения в Foo и посмотреть, как это повлияет на основной проект. Он может создать отдельный клон Foo вне основного проекта и затем установить FETCHCONTENT_SOURCE_DIR_FOO в это место. Проект TopProj будет использовать исходный текст этой локальной копии и никак не модифицировать его, но он все равно будет использовать тот же каталог сборки для него в своей собственной области сборки TopProj. Единственное различие будет в том, откуда берется исходный текст, и, установив FETCHCONTENT_SOURCE_DIR_FOO, разработчик получит контроль над содержимым. Они могут свободно изменять что угодно в своей локальной копии, делать дальнейшие коммиты, переключать ветки или что-либо еще, что может потребоваться, а затем пересобирать основной проект TopProj без необходимости изменять TopProj вообще.

Схема, которая хорошо подходит для вышеупомянутого использования, заключается в наличии общего каталога, под которым разработчик проверяет различные проекты, с которыми он хочет работать. Затем основной проект может быть направлен на эти локальные проверки, когда это необходимо, но при этом использовать загруженное по умолчанию содержимое. Такая схема может выглядеть следующим образом для приведенного выше примера:



Если разработчик хотел внести некоторые изменения в Foo и протестировать его с помощью сборки TopProj, он мог установить FETCHCONTENT_SOURCE_DIR_FOO в /.../Projects/Foo, но весь вывод сборки из зависимости Foo по-прежнему будет находиться в Projects/builds/TopProj-debug. Если значение FETCHCONTENT_SOURCE_DIR_BAR не было установлено, то Bar все равно будет загружен, а не использована локальная проверка в Projects/Bar. Разработчик может переключиться на локальную проверку так же легко, установив FETCHCONTENT_SOURCE_DIR_BAR в любое время. Поскольку все соответствующие переменные кэша имеют один и тот же префикс, их легко найти в графическом интерфейсе CMake или в инструменте ccmake. Это, в свою очередь, позволяет легко определить, какие проекты в настоящее время используют локальную копию вместо загруженного по умолчанию содержимого.



Существенным преимуществом вышеописанного сценария является то, что он хорошо интегрируется с функциями IDE, такими как инструменты рефакторинга кода и т. д. IDE видит весь проект, включая его зависимости, поэтому при использовании локальных проверок зависимостей рефакторинг может быть выполнен прозрачно для нескольких проектов так же легко, как если бы все они были частью одного проекта. Даже если не используются локальные проверки зависимостей, IDE имеет больше возможностей для построения более полной модели кода для автодополнения, следования символам и так далее.

27.2.2. Другие способы использования FetchContent

FetchContent позволяет не только загружать исходный код внешнего проекта и добавлять его в основной проект с помощью функции add_subdirectory(). Другой вариант использования - собрать часто используемые модули CMake в центральном репозитории и повторно использовать их во многих проектах. С помощью этого механизма можно привлекать несколько коллекций, что позволяет относительно просто включать полезные скрипты CMake из других проектов без необходимости встраивать копии в исходные тексты основного проекта. Ниже показан пример, когда загружается внешний git-репозиторий и его подкаталог cmake добавляется в путь поиска модулей CMake основного проекта.

```
include(FetchContent)
FetchContent_Declare(JoeSmithUtils GIT_REPOSITORY ... GIT_TAG ...)
FetchContent_GetProperties(JoeSmithUtils)
if(NOT joesmithutils_POPULATED)
    FetchContent_Populate(JoeSmithUtils)
    list(APPEND CMAKE_MODULE_PATH ${joesmithutils_SOURCE_DIR}/cmake)
endif()
```

Другой сценарий использует тот факт, что модуль FetchContent может быть использован еще до первого вызова project(). Эта возможность позволяет модулю предоставлять файлы инструментария, которые разработчик может затем использовать для основного проекта.

```
cmake_minimum_required(VERSION 3.11)

include(FetchContent)
FetchContent_Declare(CompanyXToolchains
    GIT_REPOSITORY ...
    GIT_TAG        ...
    SOURCE_DIR     ${CMAKE_BINARY_DIR}/toolchains
)
FetchContent_GetProperties(CompanyXToolchains)
if(NOT companyxtoolchains_POPULATED)
    FetchContent_Populate(CompanyXToolchains)
endif()

project(MyProj)
```

```
cmake -DCMAKE_TOOLCHAIN_FILE=toolchains/toolchain_betacxx.cmake ...
```

В приведенном выше примере каталог, в который загружаются цепочки инструментов, явно переопределяется с помощью опции SOURCE_DIR. Если предположить, что проект CompanyXToolchains представляет собой простую коллекцию файлов цепочек инструментов без подкаталогов, это делает их расположение предсказуемым и удобным для разработчиков. Там, где организации работают с очень специфическими цепочками инструментов, которые должны всегда устанавливаться в одно и то же место, это может быть очень эффективным способом облегчить всей команде использование общих настроек сборки. Эта техника может быть даже расширена для загрузки самих цепочек инструментов.

27.2.3. Ограничения

По большей части, модуль FetchContent имеет ряд серьезных преимуществ, но есть и некоторые ограничения, о которых следует знать. Основным ограничением является то, что имена целей CMake должны быть уникальными для всего набора объединяемых проектов, поэтому если два внешних проекта определяют цель с одинаковым именем, они не могут быть добавлены через add_subdirectory(). Если проекты следуют соглашению об именовании, в котором используется префикс для конкретного проекта или что-то подобное, то это ограничение довольно легко обойти. Трудности, как правило, возникают в проектах, которые никогда не ожидали, что их будут использовать подобным образом, и которые используют достаточно общие имена, которые, вероятно, будут встречаться повсеместно. Для тех проектов, которые используют специфические для

проекта имена целей, имя создаваемого бинарного файла все равно можно контролировать отдельно с помощью свойства цели OUTPUT_NAME. Например:

```
add_library(BagOfBeans_varieties ...)
set_target_properties(BagOfBeans_varieties PROPERTIES
    OUTPUT_NAME beantypes
)

add_executable(BagOfBeans_planter )
set_target_properties(BagOfBeans_planter PROPERTIES
    OUTPUT_NAME planter
)
```

Свойство OUTPUT_NAME и другие связанные с ним свойства более подробно рассматриваются в разделе 28.5.2, "Целевые выходы".

Аналогичное, но чуть менее строгое ограничение действует для компонентов установки. В идеале каждый проект должен называть свои компоненты установки префиксом, специфичным для проекта, или чем-то столь же уникальным. Это позволяет родительскому проекту выбирать только те компоненты, которые он хочет включить в свою упаковку. Если две или более внешние зависимости проекта используют одинаковые имена компонентов установки, то родительский проект не сможет их разделить и будет рассматривать их как одно целое. Важно ли это или нет, зависит от ситуации, но, опять же, этого можно легко избежать, если убедиться, что проекты используют хорошие соглашения об именах для своих установочных компонентов.

Практика встраивания внешней зависимости в более крупную родительскую сборку с помощью add_subdirectory() еще не так широко распространена. Многие проекты никогда не рассматривали такой вариант использования, и нередко встречаются шаблоны, которые затрудняют внедрение проекта таким образом. Частым примером является ситуация, когда проект предполагает, что он является проектом верхнего уровня, и использует переменные типа
CMAKE_SOURCE_DIR и CMAKE_BINARY_DIR, когда альтернативы, такие как CMAKE_CURRENT_SOURCE_DIR и CMAKE_CURRENT_BINARY_DIR, могут быть более подходящими. Такие проблемы обычно легко исправить, но для этого требуется доступ на запись к проекту, согласие сопровождающих проекта на изменения, предоставление копии проекта, в которой может быть сделано соответствующее исправление, или другие подобные меры.

27.3. ExternalData

Другой модуль под названием ExternalData предоставляет альтернативный способ управления файлами, загружаемыми во время сборки. Основное внимание в этом модуле уделяется загрузке тестовых данных при сборке определенной цели, представляющей эти данные. В некоторых отношениях он похож на работу ExternalProject, но способ определения загружаемого содержимого в этих двух модулях значительно отличается. Модуль ExternalProject позволяет явно определить детали загрузки и поддерживает различные методы. Модуль ExternalData использует другой подход, ожидая, что отдельные файлы будут доступны в одном из наборов определенных проектом базовых URL, с путями и именами файлов, закодированными с помощью определенного метода хэширования. Фактический файл представлен в дереве исходных текстов проекта файлом-заместителем с тем же именем, но с именем алгоритма хэширования, добавленного в качестве суффикса имени файла. Модуль предоставляет функцию для преобразования строковых аргументов специальной формы в их окончательное загружаемое местоположение и имя, а также обертку вокруг функции add_test(), чтобы облегчить передачу этих разрешенных местоположений тестовым командам.

На практике шаги, связанные с установкой необходимой поддержки для ExternalData, делают его менее привлекательным. Сервер, с которого загружаются данные, должен использовать определенную структуру и обрабатывать каждый файл отдельно. Каждый раз, когда добавляется новый файл или обновляется существующий, его необходимо вручную хэшировать и загружать по пути и имени файла, которые соответствуют хэшу. Если файл большой, но имеет лишь небольшое отличие от предыдущей итерации, его все равно приходится полностью копировать. Для сравнения, модуль ExternalProject может добиться того же самого с помощью одного из своих методов загрузки, основанных на репозитории, но при этом соответствующие шаги просты и знакомы большинству разработчиков. Выбор подходящего метода на основе репозитория также позволяет эффективно обрабатывать небольшие изменения в больших файлах.

Одна из причин рассмотреть использование ExternalData - это поддержка серии файлов, а не только отдельного файла. Это скорее нишевый сценарий, который обычно возникает для тестов, обрабатывающих последовательность файлов. Даже в этом случае можно реализовать аналогичную функциональность с помощью ExternalProject и цикла foreach(), что может быть проще в настройке, чем довольно жесткая структура, которую требует ExternalData. Если в проекте есть тесты, которые сильно сфокусированы на данных временных рядов или других подобных последовательных наборах данных, то стоит хотя бы оценить, не предлагает ли ExternalData более предпочтительный способ получения этих данных по запросу во время сборки. Обратитесь к документации модуля для получения справочной информации, а для более практического введения в тему может быть полезна статья на эту тему, доступная на том же сайте, что и эта книга.

27.4. Рекомендуемые практики

ExternalProject и FetchContent предоставляют способы включения внешнего содержимого в родительский проект. ExternalProject хорошо подходит для привлечения внешних проектов, которые являются зрелыми, имеют хорошую упаковку и предоставляют хорошо определенные конфигурационные файлы, которые find_package() может использовать для импорта соответствующих целей. Преимущество этого метода заключается в том, что внешние зависимости загружаются только в том случае, если они нужны сборке, и загрузка может выполняться параллельно с другими задачами сборки. Это может быть менее удобно, когда разработчикам приходится работать в нескольких проектах и вносить изменения, особенно если речь идет о скромном объеме рефакторинга. Поскольку ExternalProject уже давно является частью CMake, для него в сети уже есть готовый материал, но, несмотря на это, часто можно видеть, как разработчики испытывают трудности с его надежной настройкой. Особенно распространенным недостатком является жесткое кодирование путей и имен файлов библиотек с учетом особенностей платформы в результате смешивания ExternalProject с другими целями в основном проекте вместо классической суперсборки. Тщательно продумайте зрелость и качество упаковки внешних зависимостей, а также то, может ли основной проект использовать суперсборку, прежде чем использовать ExternalProject. Предпочтите не использовать его, если основной проект не может быть преобразован в суперсборку.

Модуль FetchContent - это хороший выбор, когда необходимо добавить в сборку другие проекты таким образом, чтобы над ними можно было работать одновременно. Он позволяет разработчикам работать над разными проектами и временно переключаться на локальные проверки, менять ветки, тестировать с разными версиями релизов и использовать различные другие возможности беспрепятственно. Это также удобно для инструментов IDE, поскольку вся сборка выглядит как единый проект, поэтому такие вещи, как завершение кода и т.д., часто дают более глубокое понимание и могут быть более надежными, чем если бы проекты загружались отдельно. При добавлении зависимостей к существующему зрелому проекту FetchContent может быть гораздо менее разрушительным, чем ExternalProject, поскольку он не требует реструктуризации основного проекта. Он также хорошо подходит для включения внешних проектов, которые являются относительно незрелыми и в которых еще не реализованы компоненты установки и упаковки. Еще одним преимуществом FetchContent является то, что он по своей природе приводит к использованию одного и того

же компилятора и настроек во всей иерархии проекта. Если минимальная версия CMake 3.11 или выше, подумайте, не является ли FetchContent более удобным и естественным решением для проекта, чем ExternalProject. Также настоятельно рекомендуется ознакомиться с такими инструментами, как ccache, для ускорения сборки, поскольку преимущества особенно заметны при использовании FetchContent.

При использовании ExternalProject или FetchContent, если детали загрузки определяются для git-репозитория, предпочтите установить GIT_TAG на хэш коммита, а не на имя ветки или тега. Это более эффективно, так как позволяет избежать сетевого соединения, если локальный клон уже имеет этот коммит.

Если проект хочет загружать тестовые данные по требованию, проверьте, является ли модуль ExternalData подходящим выбором. Модуль ExternalProject может быть проще в использовании и, скорее всего, будет лучше понятен большинству разработчиков, но в конкретных случаях, таких как работа с последовательностями файлов, ExternalData потенциально может быть проще. Если вы сомневаетесь, отдайте предпочтение ExternalProject за его более простой интерфейс и потенциально более эффективную обработку небольших изменений в больших наборах данных.

Работая над проектом, всегда предполагайте, что когда-нибудь он будет использоваться как дочерний проект другого родительского проекта. Это обеспечивает наибольшую гибкость в отношении того, как проект может быть использован в будущем. Общие проблемы, на которые следует обратить внимание, включают:

- Не предполагайте, что проект является проектом верхнего уровня. Используйте переменные CMAKE_CURRENT_SOURCE_DIR и CMAKE_CURRENT_BINARY_DIR, а не CMAKE_SOURCE_DIR и CMAKE_BINARY_DIR, когда ссылаетесь на расположение относительно собственной структуры каталогов проекта.
- Используйте имена целей, специфичные для конкретного проекта. Избегайте общих имен, даже для внутренних служебных целей, поскольку CMake требует глобально уникальных имен целей для всех импортируемых целей, кроме неглобальных, во всей иерархии проекта.
- Аналогичным образом, используйте имена компонентов установки, привязанные к конкретному проекту, и избегайте общих имен.
- Предпочтительно обеспечить некоторую детализацию набора компонентов установки, определяемых проектом, чтобы родительские проекты могли выбирать, какие части они хотят установить. Рассмотрите различные способы развертывания проекта, полностью или частично, и убедитесь, что компоненты установки позволяют фиксировать различные комбинации установленного содержимого.
- При связывании всегда используйте имя цели со сглаженным пространством имен, если доступен такой псевдоним цели (например, предпочитайте связывать с MyProj::mpfoo, а не просто mpfoo). Это позволяет использовать проект в сценариях ExternalProject и FetchContent.
- Избегайте принудительной установки переменных кэша. Вместо этого предпочитайте использовать обычные переменные CMake для переопределения любой потенциальной переменной кэша в текущей области видимости и ниже. Еще лучше использовать свойства target или directory, если они обеспечивают необходимое поведение.

Глава 28. Организация проекта

Факторы, способствующие созданию эффективной структуры проекта, многочисленны и разнообразны. То, что работает для одного проекта, может не работать для другого, но, как правило, есть некоторые общие черты.

Выбор гибкой, но предсказуемой структуры каталога на ранней стадии проекта позволяет ему развиваться с минимальными трениями и реорганизацией.

Одним из наиболее важных решений является вопрос о том, следует ли структурировать проект как суперстройку или как обычный проект. Эти два типа принципиально отличаются друг от друга и имеют свои сильные и слабые стороны. Решение в основном сводится к тому, как проект хочет относиться к своим зависимостям и есть ли желание и возможность поглотить их напрямую или держать их изолированными в своих собственных суб-билдах. Для проектов без зависимостей (и, что важно, без перспектив появления зависимостей в будущем) обычный проект является очевидным выбором. Но когда зависимости есть, правильная структура проекта может стать разницей между борьбой со сборкой и ее бесперебойной работой.

Одна из наиболее распространенных тем, возникающих в списках рассылки, на сайтах отслеживания проблем и вопросов и ответов, связана с проблемами, возникающими при попытке использовать одну структуру проекта, но ожидая, что она будет обладать возможностями другой. Во многих случаях это происходит потому, что проект начинается с определенной структуры, но затем, по мере добавления зависимостей, эта структура перестает поддерживать то, что разработчик хочет, чтобы проект мог делать. Участники проекта привыкли работать с существующей структурой, поэтому ее изменение, скорее всего, будет очень разрушительным и часто встретит значительное сопротивление. Чем старше проект, тем труднее его изменить. Поэтому решайте, как следует работать с зависимостями, на ранней стадии проекта, с учетом будущих ожиданий.

28.1. Структура суперпостройки

Там, где зависимости не используют CMake в качестве системы сборки, предпочтительной структурой является суперсборка. При этом каждая зависимость рассматривается как отдельная сборка, а главный проект определяет общую последовательность и способ передачи деталей от одной сборки зависимости к другой. Каждая отдельная сборка добавляется к основной сборке с помощью ExternalProject. Такая схема позволяет CMake просматривать то, что получается в каждой сборке, и автоматически обнаруживать информацию, которая затем может быть передана другим зависимостям, что позволяет избежать необходимости вручную вводить эту информацию в основную сборку. Даже если все зависимости используют CMake, суперсборка все равно может быть предпочтительнее по другим причинам, например, чтобы избежать столкновения имен целей или проблем с проектами, которые считают, что они всегда являются проектом верхнего уровня.

Суперсборка позволяет точно контролировать последовательность сборки отдельных зависимостей. Например, можно потребовать, чтобы одна или несколько зависимостей полностью завершили свою сборку, включая этап установки, прежде чем другие зависимости запустят свой этап конфигурации. В таком случае последующие этапы конфигурации могут видеть установленные артефакты и автоматически определять соответствующие имена файлов, расположение и т.д. Это невозможно при обычной сборке.

Суперсборки могут быть реализованы с помощью файла CMakeLists.txt верхнего уровня, который следует довольно предсказуемой схеме. В одном варианте используется общая область установки для всех зависимостей, а в другом каждая зависимость устанавливается в свою отдельную область установки. Хотя оба варианта похожи, использование общей области установки немного проще в определении:

```
cmake_minimum_required(VERSION 3.0)
project(SuperbuildExample)
include(ExternalProject)

set(installDir ${CMAKE_CURRENT_BINARY_DIR}/install)

ExternalProject_Add(someDep1      ①
    ...
    INSTALL_DIR ${installDir}
    CMAKE_ARGS  -DCMAKE_INSTALL_PREFIX:PATH=<INSTALL_DIR>
)
ExternalProject_Add(someDep2
    ...
    INSTALL_DIR ${installDir}
    CMAKE_ARGS  -DCMAKE_INSTALL_PREFIX:PATH=<INSTALL_DIR>
                -DCMAKE_PREFIX_PATH:PATH=<INSTALL_DIR>              ②
)
ExternalProject_Add_StepDependencies(someDep2 configure someDep1)   ③
```

① По крайней мере, одна зависимость не должна требовать других.

② Для других зависимостей, которые используют find_package() для поиска своих зависимостей, обычно достаточно установить CMAKE_PREFIX_PATH в общий каталог установки.

③ Зависимость шага добавляется для того, чтобы шаг configure выполнялся только после установки других необходимых зависимостей.

Если каждая зависимость должна устанавливаться в свою собственную установочную область, единственное отличие от вышеописанного заключается в том, что CMAKE_PREFIX_PATH, передаваемый последующим зависимостям, может быть списком всех установочных директорий предыдущих зависимостей вместо одной общей установочной директории.

Если зависимость не использует CMake в качестве системы сборки, общая структура не меняется, меняется только способ определения деталей сборки зависимости. Например, зависимость, использующая систему сборки типа autotools, может быть задана следующим образом:

```
ExternalProject_Add(someDep3
    INSTALL_DIR ${installDir}
    CONFIGURE_COMMAND <SOURCE_DIR>/configure --prefix <INSTALL_DIR>
    ...
)
```

Другие параметры также могут быть переданы такому сценарию конфигурации, чтобы указать ему более конкретные способы поиска зависимостей. Это будет зависеть от возможностей конфигурации зависимостей.

В суперсборках упаковка немного менее проста. В некоторых отношениях каждая зависимость действительно контролирует свою собственную упаковку, поэтому проект верхнего уровня вряд ли будет что-то упаковывать. Вместо этого, один или несколько вызовов ExternalProject_Add(), скорее всего, получат пользовательский шаг упаковки, если, конечно, упаковка вообще должна поддерживаться. В предыдущей главе было показано, как реализовать это с помощью функции ExternalProject_Add_Step() следующим образом (аналогичный подход можно использовать для подпроектов не-CMake):

```
ExternalProject_Add_Step(myProj package
    COMMAND           ${CMAKE_COMMAND} --build <BINARY_DIR> --target package
    DEPENDEES         build
    ALWAYS            YES
    EXCLUDE_FROM_MAIN YES
)
ExternalProject_Add_StepTargets(myProj package)
```

В целом, главное помнить, что суперсборки хорошо работают, когда они только объединяют другие внешние проекты. Обычно они полагаются на то, что все внешние проекты имеют четко определенные правила установки, и каждый из проектов в идеале должен уметь находить свои собственные зависимости, если ему известно местоположение других внешних проектов. Если что-то из этого не соответствует действительности, то проект верхнего уровня неизбежно окажется перед необходимостью жесткого кодирования специфических для платформы деталей одного или нескольких проектов, и тогда преимущества суперсборки начинают уменьшаться.

28.2. Несуществующее сооружение

Если у проекта нет зависимостей или если зависимости вносятся в основную сборку с помощью FetchContent или такого механизма, как git submodules, то некоторое перспективное планирование поможет избежать трудностей в дальнейшем. Практика, которая действительно помогает проекту оставаться легким для понимания и работы, заключается в том, чтобы рассматривать его верхний уровень CMakeLists.txt как оглавление. Структуру можно разделить на следующие разделы:

Преамбула

Сюда входит самая базовая настройка, например, вызовы cmake_minimum_required() и project(). Он также может включать использование модуля FetchContent для получения таких вещей, как файлы цепочки инструментов и вспомогательные репозитории CMake. Как правило, этот раздел должен быть довольно коротким.

Настройка всего проекта

В этом разделе высокого уровня можно задать некоторые глобальные свойства и переменные по умолчанию, возможно, определить некоторые опции сборки в кэше CMake и включить небольшое количество логики для выработки некоторых вещей, необходимых для всей сборки. Установка языковых стандартов по умолчанию, типов сборки и различных путей поиска - обычное дело для этого раздела.

Зависимости

Внесите внешние зависимости так, чтобы они были доступны для остальной части проекта. Вместо того, чтобы определять их в файле CMakeLists.txt верхнего уровня, поместите их в специальный каталог - это чище и имеет преимущества в плане надежности.

Основные цели строительства

В идеале этот раздел должен состоять из одного или нескольких вызовов add_subdirectory().

Тесты

В то время как модульные тесты могут быть встроены в ту же структуру каталогов, что и основные исходные тексты, интеграционные тесты могут находиться вне этой структуры в своей собственной отдельной области. Они будут добавлены после основных целей сборки.

Упаковка

Как правило, это должно быть последнее, что определяет проект, опять же в идеале в собственном подкаталоге, чтобы не загромождать верхний уровень.

Из всего вышесказанного следует, что, кроме преамбулы и настройки всего проекта, большинство вещей лучше всего определять в подкаталогах, добавляемых с помощью функции add_subdirectory(). Это не только делает файл CMakeLists.txt верхнего уровня легким для чтения и понимания, но и позволяет каждому подкаталогу сосредоточиться на определенной области. Это облегчает поиск, а также позволяет использовать области каталогов для минимизации раскрытия переменных из несвязанных областей тем, кому не нужно о них знать. Пример простого CMakeLists.txt верхнего уровня, который следует вышеуказанным рекомендациям, может выглядеть следующим образом:

```
# Preamble
cmake_minimum_required(VERSION 3.1)
project(MyProj)
enable_testing()

# Project wide setup
list(APPEND CMAKE_MODULE_PATH ${CMAKE_CURRENT_LIST_DIR}/cmake)
set(CMAKE_CXX_STANDARD 11)
set(CMAKE_CXX_STANDARD_REQUIRED YES)
set(CMAKE_CXX_EXTENSIONS NO)

# Externally provided content
add_subdirectory(dependencies)

# Main targets built by this project
add_subdirectory(src)

# Things typically only needed if we are the top level project
if(CMAKE_SOURCE_DIR STREQUAL CMAKE_CURRENT_SOURCE_DIR)
    add_subdirectory(tests)
    add_subdirectory(packaging)
endif()
```

На практике общая установка проекта, скорее всего, будет содержать больше, чем показано выше, и могут существовать другие каталоги для вещей, созданных проектом (например, для документации, добавления другого устанавливаемого содержимого, такого как скрипты, изображения и так далее).

Если следовать приведенному выше совету об определении большинства вещей в подкаталогах, то верхний каталог дерева исходных текстов проекта будет содержать, как правило, только административные файлы. Это может быть файл readme, детали лицензии, инструкции по внесению вклада и так далее. Системы непрерывной интеграции также часто ищут определенное имя файла в каталоге верхнего уровня. Не допуская попадания исходных файлов в каталог верхнего уровня, можно добиться того, чтобы он оставался сфокусированным на высокоуровневом описании проекта.

Делегирование обработки зависимостей в собственный подкаталог позволяет добиться нескольких важных вещей. Во-первых, это гарантирует, что ни одна зависимость никогда не сможет воспринимать CMAKE_SOURCE_DIR и CMAKE_CURRENT_SOURCE_DIR как равные, поэтому они могут полагаться на сравнение этих двух переменных, чтобы определить, включены ли они в большую структуру проекта или собираются отдельно. Простой пример выше показывает, как это часто используется, чтобы избежать определения тестов и деталей упаковки, когда проект не является проектом верхнего уровня. Размещение всей обработки зависимостей в собственном подкаталоге также гарантирует, что никакие некэшируемые переменные, используемые для установки зависимостей, не могут случайно попасть в другие части сборки. Полезным следствием этого является то, что это также способствует использованию целей CMake, а не переменных, как средства, с помощью которого остальная часть проекта использует зависимости.

Пример использования модуля FetchContent для включения зависимостей проекта в сборку может выглядеть следующим образом:

dependencies/CMakeLists.txt

```
include(FetchContent)

# Declare all the dependency details first in case any dependency wants
# to pull in some of the same ones (this keeps us in control)
FetchContent_Declare(jerry ...)
FetchContent_Declare(foo ...)
FetchContent_Declare(bar ...)

# Add each dependency if not already part of the build
FetchContent_GetProperties(foo)
if(NOT foo_POPULATED)
    FetchContent_Populate(foo)
    add_subdirectory(${foo_SOURCE_DIR} ${foo_BINARY_DIR})
endif()

FetchContent_GetProperties(bar)
if(NOT bar_POPULATED)
    FetchContent_Populate(bar)
    add_subdirectory(${bar_SOURCE_DIR} ${bar_BINARY_DIR})
endif()
```

Иногда зависимость может потребовать установки определенных переменных перед вызовом add_subdirectory(). В идеале это должно быть сделано в собственной области видимости, чтобы не повлиять на другие зависимости, добавленные позже в той же области видимости. Поэтому может быть полезно поместить каждую зависимость в свой собственный подкаталог, в результате чего приведенный выше пример будет выглядеть так:

dependencies/CMakeLists.txt

```
include(FetchContent)

FetchContent_Declare(jerry ...)
FetchContent_Declare(foo ...)
FetchContent_Declare(bar ...)

add_subdirectory(foo)
add_subdirectory(bar)
```

Тогда подкаталоги будут выглядеть следующим образом:

dependencies/foo/CMakeLists.txt

```
FetchContent_GetProperties(foo)
if(NOT foo_POPULATED)
    FetchContent_Populate(foo)

    # Add any customizations needed before actually pulling in the dependency.
    # For example, build static libs by default and only build those targets
    # that another target depends on.
    set(BUILD_SHARED_LIBS NO)
    set_directory_properties(PROPERTIES EXCLUDE_FROM_ALL YES)

    # Now add the dependency
    add_subdirectory(${foo_SOURCE_DIR} ${foo_BINARY_DIR})
endif()
```

Подкаталог bar будет иметь аналогичную структуру. Вышеописанное можно даже расширить для работы с предварительно собранным бинарным пакетом или пакетом исходных текстов:

```
FetchContent_GetProperties(foo)
if(NOT foo_POPULATED)
    FetchContent_Populate(foo)

    if(EXISTS ${foo_SOURCE_DIR}/CMakeLists.txt)
        # Probably source, but could still be a binary package that
        # provides itself through a top level CMakeLists.txt file
        add_subdirectory(${foo_SOURCE_DIR} ${foo_BINARY_DIR})
    else()
        # Must be a binary package, assume it provides a config file in a
        # standard location within its directory layout
        find_package(foo REQUIRED
            NO_DEFAULT_PATH
            PATHS ${foo_SOURCE_DIR}
        )
        # For this to be useful, imported targets must be promoted to global
        # so that other parts of the project can access them
        set_target_properties(foo::foo PROPERTIES IMPORTED_GLOBAL TRUE)
    endif()
endif()
```

Модуль FetchContent и свойство цели IMPORTED_GLOBAL доступны только начиная с версии CMake 3.11. Добавление зависимостей без этих свойств намного сложнее и требует компромисса в отношении некоторых рекомендуемых принципов или отказа от возможности добавления предварительно собранных бинарных пакетов. Без возможности перевода локальных целей в глобальные, альтернативные методы обычно основаны на передаче деталей обратно в основную сборку через переменные, или глобальные цели должны быть определены, чтобы действовать как прокси для локальных импортированных целей. Менее желательный подход добавляет зависимости непосредственно из файла CMakeLists.txt верхнего уровня, но это затрудняет включение проекта в более крупную иерархию проектов. Если не требуется поддержка предварительно созданных бинарных пакетов, то целевое свойство IMPORTED_GLOBAL не нужно, и обычно можно обойтись без этих альтернативных методов. Для поддержки CMake версий до 3.11 альтернативой модулю FetchContent могут быть такие методы, как git-подмодули или file(DOWNLOAD).

Из других подкаталогов верхнего уровня основного проекта, добавление тестов и упаковки не требует ничего особенного, они просто должны следовать рекомендациям, уже рассмотренным в предыдущих главах. Содержание и структура подкаталога tests зависит от конкретного проекта, в то время как для packaging обычно требуется только файл CMakeLists.txt и, возможно, несколько других файлов, которые должны быть сконфигурированы в каталог сборки для использования cpack. Он также может содержать ресурсы, используемые некоторыми генераторами пакетов. Структура каталога src - это отдельная тема, которая рассматривается в разделе 28.5, "Определение целей", далее ниже.

28.3. Общие подкаталоги верхнего уровня

В предыдущем разделе уже упоминались некоторые имена каталогов, обычно встречающиеся в качестве подкаталогов непосредственно под вершиной дерева исходников. Если расширить этот список и включить в него другие часто используемые каталоги, то получится следующее:

- cmake
- dependencies
- doc

- src
- tests
- packaging

В отсутствие каких-либо других существующих соглашений, проектам рекомендуется использовать эти же имена каталогов. Сбор вспомогательных скриптов CMake в подкаталоге cmake облегчает их поиск, позволяя разработчикам просматривать содержимое этого каталога и обнаруживать полезные утилиты, о которых они могли не знать. Один вызов list(APPEND CMAKE_MODULE_PATH ...) в разделе настроек всего проекта на верхнем уровне CMakeLists.txt также делает их доступными для всего проекта. Подкаталог doc может быть удобным местом для сбора документации, что может быть полезно, если используются форматы типа Markdown или Asciidoc и файлы содержат относительные ссылки друг на друга.

Существует несколько имен подкаталогов, которых проектам следует избегать. По умолчанию вызов add_subdirectory() только с одним аргументом приведет к появлению соответствующего каталога с тем же именем в каталоге сборки. Проекту следует избегать использования имени исходного каталога, которое может привести к столкновению с одним из предопределенных каталогов, созданных в области сборки. К числу имен, которых следует избегать, относятся следующие:

- Testing
- CMakeFiles
- CMakeScripts
- Любой из типов сборки по умолчанию (т.е. любое из значений CMAKE_CONFIGURATION_TYPES).
- Любое имя каталога, начинающееся с символа подчеркивания.

Поскольку некоторые файловые системы могут быть нечувствительны к регистру, все вышеперечисленные имена не должны использоваться в любом сочетании верхнего и нижнего регистра. Другие распространенные имена каталогов, используемые в качестве мест установки, также могут появляться в каталоге сборки в зависимости от стратегии, используемой для размещения собранных двоичных файлов (обсуждается ниже в разделе 28.5.2, "Целевые выходы"). Поэтому целесообразно избегать таких имен исходных каталогов, как bin, lib, share, man и так далее.

Некоторые проекты предпочитают определять каталог include верхнего уровня и собирать публичные заголовки там, а не хранить их рядом с соответствующими файлами реализации. Имейте в виду, что некоторые инструменты IDE могут не найти заголовки автоматически, если они разделены подобным образом, поэтому такое расположение может быть менее удобным для некоторых разработчиков. Кроме того, это делает изменения для конкретной функции или исправления ошибки менее локализованными. С другой стороны, выделенный каталог include четко указывает, какие заголовки должны быть общедоступными, и они могут иметь ту же структуру каталогов, что и при установке. Оба подхода имеют свои достоинства, но хранение заголовков вместе с соответствующими файлами реализации, возможно, немного проще для начинающих разработчиков.

28.4. Проекты IDE

При использовании генераторов проектов, таких как Xcode или Visual Studio, файл проекта или решения создается в верхней части каталога сборки. Его можно открыть в IDE, как и любой другой файл проекта для данного приложения, но он по-прежнему находится под контролем CMake. Важно отметить, что эти файлы проекта создаются как часть сборки, поэтому их не следует проверять в системе контроля версий. Также

обратите внимание, что изменения, внесенные в проект из среды IDE, будут потеряны при следующем запуске CMake.

Поскольку файлы проекта Xcode или Visual Studio генерируются CMake, это означает, что способ представления целей и файлов проекта в иерархии проекта или файловом дереве также находится под контролем проекта CMake. CMake предоставляет ряд свойств, которые могут повлиять на то, как цели и файлы группируются и маркируются в некоторых средах IDE. Первый уровень группировки - для целей, который можно включить, установив глобальное свойство USE_FOLDERS в true. Затем местоположение каждой цели можно указать с помощью свойства FOLDER target, которое содержит чувствительное к регистру имя, под которым следует поместить эту цель. Для создания древовидной иерархии можно использовать прямые косые черты для разделения уровней вложенности. Если свойство FOLDER пустое или не установлено, цель остается без группировки на верхнем уровне проекта. Генераторы Xcode и Visual Studio учитывают свойство цели FOLDER.

```
set_property(GLOBAL PROPERTY USE_FOLDERS YES)

add_executable(foo ...)
add_executable(bar ...)
add_executable(test_foo ...)
add_executable(test_bar ...)

set_target_properties(foo bar           PROPERTIES FOLDER "Main apps")
set_target_properties(test_foo test_bar PROPERTIES FOLDER "Main apps/Tests")
```

До CMake 3.11 целевой проперт FOLDER по умолчанию был пуст, а начиная с CMake 3.12 он инициализируется значением переменной CMAKE_FOLDER.

Имя, отображаемое для самой цели в IDE, по умолчанию соответствует имени цели, которое использует CMake. Генераторы Visual Studio позволяют переопределить это отображаемое имя, установив свойство цели PROJECT_LABEL, но генератор Xcode не поддерживает эту настройку.

```
set_target_properties(foo PROPERTIES PROJECT_LABEL "Foo Tastic")
```

Некоторые цели создаются самим CMake, например, для установки, упаковки, запуска тестов и так далее.

Для Xcode большинство из них не отображается в дереве файлов/целей, но для Visual Studio они по умолчанию группируются в папке CMakePredefinedTargets. Это можно переопределить с помощью глобального свойства PREDEFINED_TARGETS_FOLDER, но обычно для этого нет особых причин.

Группировка отдельных файлов под каждой целью также может контролироваться проектом CMake. Это делается с помощью команды source_group() и не зависит от группировки целевых папок (то есть поддерживается всегда, даже если глобальное свойство USE_FOLDERS ложно или не установлено). Команда имеет две формы, первая из которых используется для определения одной группы:

```
source_group(group
    [FILES src...]
    [REGULAR_EXPRESSION regex]
)
```

Группа может быть простым именем, под которым будут сгруппированы соответствующие файлы, или она может задавать иерархию, аналогичную иерархии для целей. Однако имейте в виду, что по историческим

причинам уровни вложенности определяются обратными, а не прямыми косыми чертами. Для правильного разбора CMake обратные косые черты должны быть экранированы, поэтому группа foo с вложенным под ней bar будет задана следующим образом:

```
source_group(foo\\bar ...)
```

Отдельные файлы можно указать с помощью аргумента FILES, при этом относительные пути будут считаться относительными к CMAKE_CURRENT_SOURCE_DIR. Поскольку команда не является специфичной для цели, этот параметр позволяет гарантировать, что группировка затронет только конкретные файлы. Если проект хочет определить структуру группировки, которая должна применяться более широко, то более подходящей будет опция REGULAR_EXPRESSION. Он может быть использован для эффективной настройки правил группировки, которые будут применяться ко всем целям в проекте. Если конкретный файл может соответствовать более чем одной группировке, запись FILES имеет приоритет над REGULAR_EXPRESSION, а группы REGULAR_EXPRESSION, определенные позже, имеют приоритет над теми, которые определены раньше, когда файл соответствует нескольким регулярным выражениям.

Следующий пример устанавливает общие правила для всех целей таким образом, что файлы с широко используемыми расширениями исходных и заголовочных файлов будут сгруппированы в Sources. Тестовые исходники и заголовки будут отменять эту группировку и помещаться в группу Tests, а специальный случай special.cxx будет помещен в свою собственную выделенную подгруппу под Sources.

```
source_group(Sources REGULAR_EXPRESSION "\\.(c(xx|pp)?|hh?)$")
source_group(Tests   REGULAR_EXPRESSION "test.*")  # Overrides the above
source_group(Sources\\Special FILES special.cxx)   # Overrides both of the above
```

CMake предоставляет группы по умолчанию Source Files для исходных текстов и Header Files для заголовков, но их можно легко переопределить, как показано в примере выше. Также определены другие группы по умолчанию, такие как Resources и Object Files.

Вторая форма команды source_group() позволяет иерархии групп следовать структуре каталогов для определенных файлов. Она доступна в CMake 3.8 или более поздней версии.

```
source_group(TREE    root
             [PREFIX prefix]
             [FILES  src...]
)
```

Опция TREE направляет команду на группировку указанных файлов в соответствии с их собственной структурой каталогов ниже корня. Опция PREFIX может быть использована для размещения этой структуры группировки под префиксом родительской группы или иерархии групп. Это можно очень эффективно использовать в сочетании со свойством цели SOURCES для воспроизведения структуры каталогов всех источников, составляющих цель, но только если все эти источники находятся ниже общей точки (например, нет сгенерированных источников из каталога сборки). Многие цели удовлетворяют этим условиям, поэтому следующий пример часто можно использовать для быстрого и простого придания определенной структуры тому, как цель представлена в IDE.

```
# Only suitable if SOURCES does not contain generated files in this example
get_target_property(sources someTarget SOURCES)

source_group(TREE   ${CMAKE_CURRENT_SOURCE_DIR}
             PREFIX "Magic\\Sources"
             FILES  ${sources}
)
```

В IDE обычно отображаются только те файлы, которые явно добавлены в качестве источников цели. Если цель определена и в качестве источников добавлены только файлы ее реализации, ее заголовки обычно не появляются в списках файлов IDE. Поэтому обычной практикой является явное указание заголовков, даже если они фактически не будут компилироваться. CMake будет фактически просто игнорировать их, за исключением добавления в списки исходных файлов IDE. Это относится не только к заголовочным файлам, но также может быть использовано для добавления других некомпилируемых файлов, таких как изображения, скрипты и другие ресурсы. Некоторые функции, например, связанные со свойством источника MACOSX_PACKAGE_LOCATION, требуют, чтобы файл был указан как исходный файл, чтобы иметь какой-либо эффект.

В некоторых ситуациях может быть желательно, чтобы исходный файл появлялся в списках файлов IDE, но не компилировался. Примером могут служить файлы, специфичные для конкретной платформы, которые должны компилироваться и компоноваться только на других целевых платформах. Чтобы CMake не пытался скомпилировать определенный файл, свойство исходного файла HEADER_FILE_ONLY можно установить в true (пусть вас не смущает название свойства, оно может использоваться не только для заголовков).

```
add_executable(myApp main.cpp net.cpp net_win.cpp)

if(NOT WIN32)
    # Don't need to compile this file for non-Windows platforms
    set_source_files_properties(net_win.cpp PROPERTIES
        HEADER_FILE_ONLY YES
    )
endif()
```

28.5. Определение целевых показателей

В предыдущих главах был представлен ряд функций CMake, позволяющих детально определить цель. Это включает исходные тексты и другие файлы, из которых собирается цель, как цель должна быть собрана и как цель взаимодействует с другими целями. Этот раздел посвящен демонстрации того, как использовать эти методы таким образом, чтобы проект был прост для понимания, обеспечивал надежную сборку, гибкость и удобство сопровождения.

Для простых проектов количество исходных файлов и целей, скорее всего, будет небольшим, и в этом случае все необходимые детали можно будет указать в одном файле CMakeLists.txt. Если следовать структуре каталогов проекта, рекомендованной ранее в этой главе, это означает, что подкаталог src не будет иметь других подкаталогов, а его файл CMakeLists.txt будет определять все, что необходимо. Изначально он может выглядеть просто:

src/CMakeLists.txt

```
add_executable(planter main.cpp soy.cpp coffee.cpp)

target_compile_definitions(planter PUBLIC COFFEE_FAMILY=Robusta)

add_test(NAME NoArgs   COMMAND planter)
add_test(NAME WithArgs COMMAND planter beanType=soy)
```

Это делает ряд предположений о том, как проект будет использоваться, но, возможно, самые большие из них заключаются в том, что проект не будет установлен или упакован и что он не будет поглощен в иерархию более крупного проекта каким-либо другим проектом. Это ограничения, которых можно и нужно избегать. К конкретным недостаткам приведенного выше простого примера относятся:

- Имя цели не является специфическим для проекта, поэтому, если этот проект позже был включен в более крупный родительский проект, имя цели может конфликтовать с целями, определенными в других местах. Использование специфического для проекта префикса в имени цели - простой способ устранить этот недостаток.
- Правила установки отсутствуют, поэтому цель не может быть легко установлена или включена в пакет.
- Псевдоним цели не определен, поэтому даже если команда install() будет добавлена позже и будет реализована упаковка, другим проектам придется использовать разные имена целей для сборки бинарных файлов и включения исходных текстов.
- Имена тестов не являются специфическими для проекта, поэтому они могут вступать в противоречие с именами тестов других проектов, если этот проект будет включен в иерархию более крупных проектов. Опять же, включение названия проекта или другой такой же уникальной строки в названия тестов поможет решить эту проблему.
- Тесты добавляются всегда, даже если это не проект верхнего уровня. Для больших проектов с большим количеством тестов это может заметно и неоправданно увеличить время сборки.
- Заголовки не указываются в качестве источников, поэтому они не будут отображаться в некоторых IDE.

С учетом вышеизложенных моментов и следуя рекомендациям предыдущих глав, пример выглядит примерно так:

src/CMakeLists.txt

```
#============================
# Define targets
#============================
add_executable(BagOfBeans_planter main.cpp soy.cpp soy.h coffee.cpp coffee.h)
add_executable(BagOfBeans::BagOfBeans_planter ALIAS BagOfBeans_planter)
set_target_properties(BagOfBeans_planter PROPERTIES OUTPUT_NAME planter)
target_compile_definitions(BagOfBeans_planter PUBLIC COFFEE_FAMILY=Robusta)

#============================
# Testing
#============================
add_test(NAME BagOfBeans.planter.NoArgs   COMMAND BagOfBeans_planter)
add_test(NAME BagOfBeans.planter.WithArgs COMMAND BagOfBeans_planter beanType=soy)

#============================
# Packaging
#============================
include(GNUInstallDirs)
install(TARGETS     BagOfBeans_planter
        EXPORT      BagOfBeans_apps
        DESTINATION ${CMAKE_INSTALL_BINDIR}
        COMPONENT   BagOfBeans_apps
)
```

Возможно, это удивительно много деталей для довольно простого исполняемого файла, но это подчеркивает, что для реальных проектов необходимо учитывать больше, чем просто сборка двоичного файла в отдельности. Дополнительные сложности в основном связаны с более длинными именами для уменьшения вероятности столкновений. Добавление логики упаковки также имеет тенденцию добавлять достаточно много деталей, с которыми неопытный разработчик, вероятно, не сталкивался. Добавление четких разделов в файл, как показано выше, может облегчить понимание для начинающих разработчиков, а также сохранить его организованность по мере развития проекта.

28.5.1. Целевые источники

Когда количество исходных файлов увеличивается, их размещение в одном каталоге может усложнить работу с ними. Обычно это решается путем размещения их в подкаталогах, сгруппированных по функциональности, что дает и ряд других преимуществ. Это не только помогает избежать загромождения, но и позволяет легко включать и выключать определенные функции на основе опций кэша CMake или другой логики времени конфигурирования. Например:

```
add_executable(BagOfBeans_planter main.cpp)

option(BAGOFBEANS_SOY    "Support planting soy beans"    ON)
option(BAGOFBEANS_COFFEE "Support planting coffee beans" ON)
if(BAGOFBEANS_SOY)
    add_subdirectory(soy)
endif()
if(BAGOFBEANS_COFFEE)
    add_subdirectory(coffee)
endif()
```

Во всех предыдущих главах исполняемые файлы и библиотеки всегда определялись в одном каталоге, поэтому полный список файлов мог быть предоставлен непосредственно вызову add_executable() или add_library(). В приведенной выше схеме подкаталоги добавляют исходники к цели после того, как она была определена с помощью команды target_sources(), которая доступна в CMake 3.1 или более поздней версии. Эта команда работает так же, как и другие команды target_...(), и имеет очень похожую форму:

```
target_sources(targetName
    <PRIVATE|PUBLIC|INTERFACE> src...
    # Repeat with more sections as needed
    ...
)
```

Предоставляется одна или несколько секций PRIVATE, PUBLIC или INTERFACE, в каждой из которых перечислены исходные файлы, которые должны быть добавлены к соответствующей цели. Источники PRIVATE добавляются в свойство SOURCES targetName, а источники INTERFACE - в свойство INTERFACE_SOURCES. Источник PUBLIC добавляется в оба свойства. На практике это выглядит так: PRIVATE-источники компилируются в targetName, INTERFACE-источники добавляются ко всему, что связано с targetName, а PUBLIC-источники добавляются в оба свойства.

На практике любое другое значение, кроме PRIVATE, будет необычным, поскольку добавление исходного файла ко всем объектам, связывающим с targetName, будет иметь ограниченную полезность. Его можно использовать для добавления ресурсов, которые для работы должны быть частью одной и той же единицы перевода, или для встраивания чего-то, что не должно быть открыто через какой-либо межбиблиотечный интерфейс, но такие ситуации встречаются нечасто.

Особенностью команды target_sources() является то, что если источник указан с относительным путем, этот путь предполагается относительным к каталогу источника цели, к которой он добавляется. Это создает ряд проблем. Первая заключается в том, что если бы источник был добавлен как источник INTERFACE, то путь считался бы относительным к другой цели, а не к targetName. Очевидно, что это может привести к созданию неправильных путей, поэтому любой неприватный источник должен быть указан с абсолютным путем. Вторая проблема заключается в том, что относительные пути ведут себя неинтуитивно, когда target_sources() вызывается из каталога, отличного от того, в котором было определено targetName. Рассмотрим, как может быть указан файл CMakeLists.txt для одной из директорий предыдущего примера:

src/coffee/CMakeLists.txt

```
target_sources(BagOfBeans_planter
    PRIVATE
        # WARNING: These will be wrong
        coffee.cpp
        coffee.h
)
...
```

Вышеописанное предназначено для добавления источников из того же каталога, но они будут интерпретироваться как относительные к src, а не src/coffee. Наиболее надежным способом решения этой проблемы является префикс CMAKE_CURRENT_SOURCE_DIR или CMAKE_CURRENT_LIST_DIR, чтобы гарантировать, что они всегда используют правильный путь.

src/coffee/CMakeLists.txt

```
target_sources(BagOfBeans_planter
    PRIVATE
        ${CMAKE_CURRENT_LIST_DIR}/coffee.cpp
        ${CMAKE_CURRENT_LIST_DIR}/coffee.h
)

target_compile_definitions(BagOfBeans_planter
    PUBLIC COFFEE_FAMILY=Robusta
)

target_include_directories(BagOfBeans_planter
    PUBLIC $<BUILD_INTERFACE:${CMAKE_CURRENT_LIST_DIR}>
)
```

Префикс каждого исходного файла с ${CMAKE_CURRENT_LIST_DIR} или ${CMAKE_CURRENT_LIST_DIR} в подобных случаях неудобен и не особенно интуитивен. Улучшение в этой области вероятно в ближайшем выпуске (необходимые изменения уже сделаны).

Выше также показано, как другие команды target_...() могут быть перемещены в подкаталоги, а не только target_sources(). Это помогает сохранить локальность кода, к которому они относятся. Например, определения компиляции, флаги компилятора и пути поиска заголовков, специфичные для конкретной функции, могут быть добавлены только в том случае, если эта функция включена. Если потребуется реорганизовать структуру каталогов и перенести этот каталог в другое место, то в этом файле ничего не нужно будет менять, а другие исходники в целевом файле, содержащие #include "coffee.h", продолжат работать без изменений.

Единственным исключением из этой локализации деталей является target_link_libraries(), которая может использоваться только для цели, определенной в том же каталоге. Это означает, что если подкаталогу нужно сделать целевую ссылку на что-то, он не сможет сделать это из этого подкаталога. Вызов target_link_libraries() должен быть сделан в том же каталоге, где была вызвана add_executable() или add_library(). Если, например, цель BagOfBeans_planter должна была скомпоноваться с библиотекой weather, ей пришлось бы добавить вызов в src/CMakeLists.txt, а не в src/coffee/CMakeLists.txt. В результате получилось бы что-то вроде следующего:

```
option(BAGOFBEANS_COFFEE "Support planting coffee beans" ON)

if(BAGOFBEANS_COFFEE)
    add_subdirectory(coffee)
    target_link_libraries(BagOfBeans_planter PRIVATE weather)
endif()
```

Это ограничение активно обсуждается разработчиками CMake и может быть снято или, по крайней мере, значительно смягчено в будущей версии CMake. В версиях CMake от 3.1 до 3.12 подкаталоги могут быть полностью самодостаточными, за исключением добавления библиотек, на которые должна ссылаться цель. До версии CMake 3.1 требовался совершенно другой подход, который заключался в создании списков источников в переменной и создании цели только после добавления всех подкаталогов. Такая схема могла выглядеть следующим образом:

```
# Pre-CMake 3.1 method, avoid using this approach
unset(planterSources)
unset(planterDefines)
unset(planterOptions)
unset(planterLinkLibs)

# Subdirs are expected to add to the above variables using PARENT_SCOPE
option(BAGOFBEANS_SOY    "Support planting soy beans"    ON)
option(BAGOFBEANS_COFFEE "Support planting coffee beans" ON)
if(BAGOFBEANS_SOY)
    add_subdirectory(soy)
endif()
if(BAGOFBEANS_COFFEE)
    add_subdirectory(coffee)
endif()

# Lastly define the target and its other details. All variables
# are assumed to name PRIVATE items.
add_executable(BagOfBeans_planter ${planterSources})
target_compile_definitions(BagOfBeans_planter PRIVATE ${planterDefines})
target_compile_options(BagOfBeans_planter PRIVATE ${planterOptions})
target_link_libraries(BagOfBeans_planter PRIVATE ${planterLinkLibs})
```

Вышеописанное станет еще сложнее, если некоторые элементы должны быть не PRIVATE. Использование переменных подобным образом является хрупким, так как оно полагается на то, что в подкаталогах не будут использоваться одни и те же переменные для разных целей, а опечатки в именах переменных не будут восприниматься CMake как ошибка. Это также усиливает связь между родительскими и дочерними каталогами, поскольку каждый дочерний подкаталог должен будет передавать все соответствующие переменные обратно своему родителю с помощью set(... PARENT_SCOPE). Для глубоко вложенных каталогов это быстро становится утомительным и чревато ошибками.

28.5.2. Целевые результаты

Когда библиотека или исполняемый файл собраны, их местоположение по умолчанию будет либо CMAKE_CURRENT_BINARY_DIR, либо подкаталог ниже, в зависимости от типа используемого генератора проектов. Для проектов с большим количеством подкаталогов или глубоко вложенных иерархий это может быть неудобно для разработчика. Для таких случаев CMake предоставляет ряд свойств цели, которые дают проекту определенную степень контроля над местом вывода собранного двоичного файла каждой цели:

RUNTIME_OUTPUT_DIRECTORY

Используется для исполняемых файлов на всех платформах и для DLL на Windows.

LIBRARY_OUTPUT_DIRECTORY

Используется для разделяемых библиотек на платформах, отличных от Windows.

ARCHIVE_OUTPUT_DIRECTORY

Используется для статических библиотек на всех платформах и для библиотек импорта, связанных с библиотеками DLL на Windows.

Для всех трех вышеперечисленных свойств генераторы мультиконфигурации, такие как Visual Studio и Xcode, автоматически добавят специфичный для конфигурации подкаталог к каждому значению, если оно не содержит выражения генератора. Связанные свойства конфигурации с добавлением _<CONFIG> также поддерживаются по историческим причинам, но их следует избегать в пользу использования выражений генератора там, где требуется поведение, специфичное для конфигурации.

Обычно эти целевые свойства используются для того, чтобы собрать библиотеки и исполняемые файлы в структуру каталогов, аналогичную той, которую они имеют при установке. Это полезно, если приложения ожидают, что различные ресурсы будут расположены в определенном месте относительно двоичного файла исполняемого файла. В Windows это также может упростить отладку, поскольку исполняемые файлы и библиотеки DLL могут быть собраны в одном каталоге, что позволяет исполняемым файлам автоматически находить свои зависимости DLL (на других платформах это не требуется, поскольку поддержка RPATH встраивает необходимые местоположения в сами двоичные файлы).

Следуя обычному шаблону, эти свойства цели инициализируются одноименной переменной CMake с приставкой CMAKE_. Когда все цели должны использовать одинаковые последовательные места вывода, эти переменные можно установить в верхней части проекта, чтобы не устанавливать свойства для каждой цели отдельно. Чтобы проект можно было включить в иерархию более крупных проектов, эти переменные следует устанавливать только в том случае, если они еще не установлены, чтобы родительские проекты могли переопределять местоположение вывода. Они также должны использовать расположение относительно CMAKE_CURRENT_BINARY_DIR, а не CMAKE_BINARY_DIR. Следующий пример показывает, как безопасно собирать двоичные файлы в подкаталоге stage текущего каталога двоичных файлов, если родительский проект не отменит это.

```
include(GNUInstallDirs)
if(NOT CMAKE_RUNTIME_OUTPUT_DIRECTORY)
    set(CMAKE_RUNTIME_OUTPUT_DIRECTORY
        ${CMAKE_CURRENT_BINARY_DIR}/stage/${CMAKE_INSTALL_BINDIR})
endif()
if(NOT CMAKE_LIBRARY_OUTPUT_DIRECTORY)
    set(CMAKE_LIBRARY_OUTPUT_DIRECTORY
        ${CMAKE_CURRENT_BINARY_DIR}/stage/${CMAKE_INSTALL_LIBDIR})
endif()
if(NOT CMAKE_ARCHIVE_OUTPUT_DIRECTORY)
    set(CMAKE_ARCHIVE_OUTPUT_DIRECTORY
        ${CMAKE_CURRENT_BINARY_DIR}/stage/${CMAKE_INSTALL_LIBDIR})
endif()
```

Избегайте создания CMAKE_..._OUTPUT_DIRECTORY в качестве переменных кэша, так как они не должны находиться под контролем разработчика. Они должны контролироваться проектом, поскольку различные части проекта могут делать предположения о взаимном расположении двоичных файлов. Более того, если оставить их как обычные переменные, они могут быть отменены в подкаталогах, где определены тестовые исполняемые файлы, что позволит им не собираться вместе с другими основными двоичными файлами и не загромождать эту область.

Имя собранного двоичного файла также может контролироваться проектом. По умолчанию базовое имя двоичного файла совпадает с именем цели. Когда имена целей следуют соглашению о включении имени проекта (чтобы помочь сохранить их уникальность в рамках более крупной иерархии проектов), имя цели может не подходить в качестве базового имени двоичного файла, поэтому это значение по умолчанию может потребоваться отменить. Свойство цели OUTPUT_NAME может быть установлено на базовое имя, используемое для двоичного файла, или для менее типичных ситуаций могут быть установлены более специфические

свойства RUNTIME_OUTPUT_NAME, LIBRARY_OUTPUT_NAME и ARCHIVE_OUTPUT_NAME. В большинстве случаев OUTPUT_NAME является достаточным и предпочтительным.

```
add_executable(BagOfBeans_planter ...)
set_target_properties(BagOfBeans_planter PROPERTIES OUTPUT_NAME planter)
```

Варианты, специфичные для конфигурации, такие как OUTPUT_NAME_<CONFIG>, также поддерживаются по историческим причинам, но проекты должны предпочесть использовать выражения генератора вместо этого.

Старые проекты иногда пытаются прочитать целевое свойство LOCATION для определения выходного местоположения и имени бинарного файла и использовать его в таких местах, как пользовательские целевые команды или другая подобная логика. Как уже отмечалось в разделе 13.4, "Рекомендуемая практика", это проблематично для генераторов с несколькими конфигурациями, поскольку расположение зависит от конфигурации, но это не учитывается целевым свойством LOCATION. CMake 3.0 и более поздние версии будут предупреждать, если проект попытается установить это целевое свойство. Вместо этого проектам следует использовать выражения генератора типа $<TARGET_FILE:...>.

28.5.3. Проблемы, специфичные для Windows

Отсутствие в Windows поддержки RPATH создает ряд проблем для разработчиков. При запуске исполняемого файла во время разработки все DLL, которые требует исполняемый файл, должны находиться либо в том же каталоге, либо в одном из каталогов, перечисленных в переменной окружения PATH. Для основных двоичных файлов проекта можно использовать различные свойства ..._OUTPUT_PATH для размещения исполняемых файлов и библиотек в одном месте, но эта техника менее удобна для тестовых исполняемых файлов, поскольку их может быть много, и располагать их все в одном каталоге вывода может быть сложнее.

Для тестов, выполняемых через ctest, свойство теста ENVIRONMENT можно использовать для добавления необходимых каталогов DLL в PATH следующим образом:

```
add_executable(fooTest ...)
target_link_libraries(fooTest PRIVATE algo)
add_test(NAME fooWithAlgo COMMAND fooTest)
if(WIN32)
    set_tests_properties(fooWithAlgo PROPERTIES ENVIRONMENT
        "PATH=$<SHELL_PATH:$<TARGET_FILE_DIR:algo>>$<SEMICOLON>$ENV{PATH}"
    )
endif()
```

Это не поможет разрешить запуск тестового исполняемого файла в Visual Studio IDE под отладчиком. Для этого нужны более сложные меры. В CMake 3.8 добавлена поддержка целевого свойства VS_USER_PROPS, которое можно использовать для переопределения расположения файла свойств пользователя для каждой цели. Пользовательский файл свойств может быть создан с параметром LocalDebuggerEnvironment, установленным в дополнительные записи PATH, которые будут объединены с PATH по умолчанию. Если все DLL-библиотеки, необходимые тестам, будут собраны в небольшом количестве мест, то можно создать один файл пользовательских свойств и повторно использовать его для каждого теста (но при необходимости можно создать и использовать собственный файл пользовательских свойств для каждой цели). Команда configure_file() может быть использована для автоматического заполнения выходного каталога.

```
file(TO_NATIVE_PATH ${CMAKE_RUNTIME_OUTPUT_DIRECTORY} baseDir)
configure_file(user.props.in user.props @ONLY)
```

Файлы свойств пользователя могут быть немного сложными, но пример достаточно базового файла, который можно использовать с вышеупомянутым, может выглядеть следующим образом:

user.props.in

```
<?xml version="1.0" encoding="utf-8"?>
<Project DefaultTargets="Build" ToolsVersion="15.0"
  xmlns="http://schemas.microsoft.com/developer/msbuild/2003">
  <PropertyGroup Condition="'$(Configuration)|$(Platform)'=='Debug|Win32'">
    <LocalDebuggerEnvironment>PATH=@baseDir@\Debug</LocalDebuggerEnvironment>
  </PropertyGroup>
  <PropertyGroup Condition="'$(Configuration)|$(Platform)'=='Release|Win32'">
    <LocalDebuggerEnvironment>PATH=@baseDir@\Release</LocalDebuggerEnvironment>
  </PropertyGroup>
</Project>
```

Файлы свойств пользователя можно использовать не только для настройки среды отладчика, но вышеизложенное, по крайней мере, является отправной точкой для тех, кто хочет изучить эту технику дальше.

Для исполняемых файлов Windows и DLL типично создание файла PDB (базы данных программы), чтобы отладочная информация была доступна во время разработки. Существует два вида PDB-файлов, и CMake предоставляет возможности для обоих. Для разделяемых библиотек и исполняемых файлов целевые свойства PDB_NAME и PDB_NAME_<CONFIG>, специфичные для конфигурации, могут использоваться для переопределения базового имени файла PDB. Обычно наиболее подходящим является имя по умолчанию, поскольку оно совпадает с именем DLL или исполняемого файла, но имеет суффикс .pdb, а не .dll или .exe. По умолчанию PDB-файл помещается в тот же каталог, что и DLL или исполняемый файл, но это можно переопределить с помощью целевых свойств PDB_OUTPUT_DIRECTORY и PDB_OUTPUT_DIRECTORY_<CONFIG>. Обратите внимание, что в отличие от других свойств ..._OUTPUT_DIRECTORY, PDB_OUTPUT_DIRECTORY не поддерживает выражения генератора в CMake 3.11 или более ранней версии.

Также создается второй тип PDB-файла, который содержит информацию для отдельных объектных файлов, собираемых для цели. Этот PDB-файл менее полезен во время разработки, за исключением, возможно, статических библиотек. Для C++ этот последний PDB-файл по умолчанию имеет имя VCxx.pdb, где xx означает версию используемого Visual C++ (например, VC14.pdb). Поскольку имя по умолчанию не зависит от цели, в некоторых ситуациях легко ошибиться и перепутать PDB для разных целей. CMake позволяет управлять именем объектного PDB-файла каждой цели с помощью свойств цели COMPILE_PDB и конфигурации COMPILE_PDB_<CONFIG>. Расположение этих объектных PDB-файлов также может быть переопределено с помощью целевых свойств COMPILE_PDB_OUTPUT_DIRECTORY и COMPILE_PDB_OUTPUT_DIRECTORY_<CONFIG>. Обратите внимание, что эти объектные PDB-файлы малопригодны для DLL и исполняемых целей, поскольку основной PDB уже содержит всю необходимую отладочную информацию.

28.6. Различные характеристики проекта

Генераторы проектов обычно предоставляют некую чистую цель, которую можно использовать для удаления всех сгенерированных файлов, выводов сборки и т. д. Это иногда используется инструментами IDE для обеспечения базовой функции пересборки в виде очистки, за которой следует сборка, или разработчиками для простого удаления выводов сборки, чтобы заставить все перестроить при следующей попытке сборки. Иногда

проект определяет пользовательское правило таким образом, что оно создает файлы, о которых CMake не знает, поэтому они не включаются в шаг очистки и могут повлиять на следующую сборку. Проекты могут сообщить CMake об этих файлах, добавив их в свойство каталога ADDITIONAL_MAKE_CLEAN_FILES, которое содержит список файлов для данного диапазона каталогов, которые должны быть частью цели очистки. Это поддерживается только семейством генераторов Makefile. Генератор Ninja не поддерживает это свойство, но он предоставляет более надежную альтернативу с помощью опции BYPRODUCTS, предоставляемой командам add_custom_command() и add_custom_target(). Перечисляя такие файлы как побочные продукты, Ninja знает, что их нужно удалить при сборке чистой цели. Другие генераторы проектов не имеют эквивалентной функциональности.

Некоторые более сложные методы могут потребовать повторного запуска CMake при изменении конкретного файла. Обычно CMake хорошо справляется с автоматическим отслеживанием зависимостей для контролируемых им объектов, таких как копирование файлов с помощью команды configure_file(), но пользовательские команды и другие задачи могут полагаться на файлы, для которых CMake не знает о зависимости. Такие файлы можно добавить в свойство каталога CMAKE_CONFIGURE_DEPENDS, и если какой-либо из перечисленных файлов изменится, CMake будет перезапущен перед следующей сборкой. Если файл указан с относительным путем, он будет считаться относительным к исходному каталогу, связанному со свойством directory. В большинстве проектов обычно не требуется использовать свойство каталога CMAKE_CONFIGURE_DEPENDS, но его можно и нужно использовать в тех случаях, когда CMake не имеет возможности узнать о файлах, которые служат входом для шагов configure или generation. Большинство файловых зависимостей являются зависимостями во время сборки, а не во время конфигурирования или генерации, поэтому перед использованием этого свойства проверьте, действительно ли проекту требуется повторный запуск CMake, а не просто перекомпиляция исходного файла или цели в рамках обычной сборки.

Неизбежно наступит момент, когда проект из какого-либо внешнего источника необходимо добавить в сборку, но у него есть какая-то проблема, которая не позволяет ему работать должным образом. Среди распространенных примеров - неустановка переменных или свойств, которые должны быть установлены. Это особенно часто встречается при работе с проектами, которые поддерживают очень старые версии CMake и не были обновлены для работы с более новыми функциями и проверками CMake. Для некоторых из этих проблем можно внедрить код CMake без необходимости изменять внешний проект и обойти проблему. Команда project() имеет возможность проверить наличие переменной с именем CMAKE_PROJECT_<PROJNAME>_INCLUDE, где <PROJNAME> - имя проекта, заданное командой project(). Если эта переменная определена, предполагается, что она содержит имя файла, который CMake должен включить в проект в качестве последнего действия команды project() перед возвращением. По сути, команда project() работает следующим образом:

```
project(SomeProj)
if(CMAKE_PROJECT_SomeProj_INCLUDE)
    include(${CMAKE_PROJECT_SomeProj_INCLUDE})
endif()
```

Поскольку такое поведение поддерживается для каждого вызова project(), каждый вызов project() становится потенциальной точкой инъекции кода CMake. Его можно использовать для изменения значений по умолчанию для целевых свойств в проекте, а также для добавления дополнительных флагов компилятора или компоновщика и так далее. Еще одно особенно удобное применение этой функции - безопасная установка опций для сборок непрерывной интеграции без необходимости сохранять их в кэше CMake. Это означает, что инкрементные сборки будут менее подвержены влиянию старых опций кэша CMake, которые удаляются или больше не устанавливаются после последующих изменений в проекте.

Например, рассмотрим разработчика, работающего над интеграционной веткой, в которой необходимо временно включить дополнительные проверки. Наивным подходом было бы явное задание переменных типа CMAKE_C_FLAGS или CMAKE_CXX_FLAGS, но поскольку CI-скрипты не должны изменять сам проект, единственным вариантом было бы задать их как опции кэша. Когда ветка будет объединена, эти опции кэша будут продолжать присутствовать для инкрементных сборок, но они больше не должны применяться. Тогда единственным вариантом действий будет очистка кэша, что, скорее всего, приведет к полной пересборке. Лучшей альтернативой является использование CMAKE_PROJECT_<PROJNAME>_INCLUDE для обработки CI-специфического файла в конце самого верхнего вызова project(). Этот файл будет находиться под контролем исходного кода так же, как и остальная часть проекта. Перед слиянием ветви этот файл будет восстановлен к своему обычному содержимому, и сборка не будет сохранять временные флаги.

CMakeLists.txt

```
cmake_minimum_required(VERSION 3.0)
project(MyProj)
...
```

CMake будет вызываться так системой CI:

```
cmake -D CMAKE_PROJECT_MyProj_INCLUDE:FILEPATH=path/to/ciOptions.cmake ...
```

Обычно файл ciOptions.cmake может быть пустым или содержать только несколько общих настроек, таких как включение дополнительных функций. Для ветки он может содержать следующее:

ciOptions.cmake

```
compile_definitions(DO_EXTRA_CI_CHECKS=1)
set(ENABLE_SANITIZERS YES)
```

Подобная инъекция файлов в команды project() не должна быть частью нормальной разработки проекта. Это имеет специфическое применение для устранения недостатков в старых проектах и в очень контролируемых ситуациях, таких как сборки непрерывной интеграции, но вне этих случаев разработчики обычно предпочитают добавлять или изменять файлы CMakeLists.txt проекта напрямую.

28.7. Рекомендуемые практики

Способы структурирования и использования проектов могут существенно различаться. Некоторые вещи, которые раньше были обычным делом, теперь считаются плохой практикой, поскольку новые возможности и извлеченные уроки позволяют заменить старые методы новыми, более надежными, более гибкими и позволяющими то, что раньше было невозможно. Инструменты обновляются, языки развиваются, зависимости меняются - все это означает, что проекты также должны адаптироваться со временем. В частности, для проектов CMake, которые продолжают использовать старые версии CMake до 3.0, все чаще приходится сталкиваться с неровным путем. Сейчас наблюдается сильное движение в сторону модели, ориентированной на цели, и большая часть разработки CMake направлена в этом направлении. Поэтому лучше установить минимальную версию CMake, которая позволит проекту использовать эти возможности. Все, что меньше CMake 3.1, скорее всего, будет слишком ограничительным, поэтому по возможности используйте CMake 3.7 из-за обновленной поддержки языка и новых возможностей. Если вы работаете с новыми инструментами, такими как CUDA, или с очень новым стандартом языка, настоятельно рекомендуется использовать последнюю версию

CMake. Новые выпуски Visual Studio или Xcode также обычно требуют последних версий CMake, чтобы получить исправления и дополнения для изменений в этих инструментальных цепочках.

Фундаментальный выбор, который должен сделать каждый проект, - это выбор структуры суперсборки или обычной сборки. Если проект может установить минимальную версию CMake 3.11, то не суперсборка имеет более мощные возможности для управления зависимостями, которые могут сделать ненужной суперсборку. Подумайте, может ли модуль FetchContent и продвижение локальных импортированных целей до глобального масштаба обеспечить большую гибкость и лучший опыт для разработчиков. Если все зависимости проекта относительно зрелые и имеют четко определенные правила установки, суперсборка все еще может быть подходящей альтернативой и имеет то преимущество, что ее можно использовать с гораздо более старыми версиями CMake. Оба метода имеют свое место, но чем раньше в жизни проекта будет принято решение об использовании суперсборки, тем больше вероятность того, что проект сможет избежать масштабной разрушительной реструктуризации в дальнейшем.

Независимо от того, является ли проект суперсборкой или нет, стремитесь к тому, чтобы верхний уровень проекта был сосредоточен на деталях более высокого уровня. Считайте, что файл CMakeLists.txt верхнего уровня - это скорее оглавление проекта. Каталог верхнего уровня должен в основном содержать только административные файлы и набор подкаталогов, каждый из которых сосредоточен на определенной области. Избегайте имен подкаталогов, которые могут вызвать конфликты с теми, которые создаются в каталоге сборки автоматически. Предпочтительнее использовать достаточно стандартные имена, если только нет существующей конвенции, которой необходимо следовать.

Для обычных проектов старайтесь, чтобы файл CMakeLists.txt верхнего уровня соответствовал общепринятой схеме разделов:

- Преамбула
- Настройка всего проекта - Зависимости
- Основные цели строительства
- Тесты
- Упаковка

Четкое разграничение каждого раздела с помощью блоков комментариев поможет стимулировать разработчиков, работающих над проектом, поддерживать эту структуру. Установление такой схемы в разных проектах поможет усилить внимание к тому, чтобы файл CMakeLists.txt верхнего уровня был упорядочен и выполнял роль высокоуровневого обзора.

При определении целей сборки, источники которых распределены по каталогам, предпочтите сначала создать цель, а затем в каждом подкаталоге добавить к ней источники с помощью target_sources(). При необходимости группируйте подкаталоги по функциональности или возможностям, чтобы их можно было легко перемещать или включать/выключать как единое целое. Во многих случаях другие команды, ориентированные на цель (например.
target_compile_definitions(), target_compile_options() и target_include_directories()) могут также использоваться локально в пределах подкаталога, к которому они относятся. Это помогает сохранить информацию в том месте, где она актуальна, а не распространять ее по каталогам. Избегайте использования переменных для создания списков источников, которые будут передаваться обратно вверх по иерархии каталогов и в конечном итоге использоваться для создания цели, определения флагов компилятора и т.д. Использование переменных вместо непосредственного оперирования целями гораздо более хрупко, более многословно и с меньшей вероятностью приведет к тому, что CMake обнаружит опечатки или другие ошибки.

В продолжение вышесказанного и повторяя одну из рекомендаций из главы 4, *Строительство Простые цели*, избегайте слишком распространенной практики излишнего использования переменной для хранения имени цели или проекта. В частности, следует избегать следующего шаблона:

```
set(projectName ...)
project(${projectName})
add_executable(${projectName} ...)
```

Приведенный выше пример связывает вещи, которые не должны быть так сильно связаны. Имя проекта должно меняться редко. Укажите имя проекта непосредственно в команде project() и используйте стандартные переменные, предоставляемые CMake, если на него нужно ссылаться в других местах проекта. Для целей имя цели используется настолько широко, что пытаться переносить его в переменную громоздко и чревато ошибками. Дайте цели имя и используйте его последовательно во всем проекте. Даже если во всем проекте только одна цель, она не обязательно должна совпадать с названием проекта, и их следует рассматривать отдельно, а не связывать вместе.

При добавлении тестов старайтесь держать тестовый код рядом с тестируемым кодом. Это поможет сохранить логически связанный код вместе и побудит разработчиков поддерживать тесты в актуальном состоянии. Тесты, распределенные по другим частям иерархии каталога исходных текстов, могут быть легко забыты. Для тестов, использующих несколько областей, таких как интеграционные тесты, принцип локальности не так силен, поэтому может быть целесообразно собирать эти тесты более высокого уровня в одном месте. Подкаталог тестов верхнего уровня предназначен для подобных ситуаций.

Для больших проектов подумайте, стоит ли организовывать способ представления проекта в инструментах IDE. Если целей много, работать с проектом может быть сложно, если не добавить некоторую структуру с помощью свойства цели FOLDER. Для целей с большим количеством источников их тоже можно организовать с помощью команды source_group(), которую можно использовать для определения иерархии групп по любым концепциям или функциям, имеющим смысл.

Особое внимание следует уделить проектам, которые предполагается создавать под Windows, особенно там, где разработчики могут использовать IDE Visual Studio. Отсутствие поддержки RPATH означает, что исполняемые файлы полагаются на возможность найти свои DLL-зависимости либо в том же каталоге, либо через переменную окружения PATH. Это влияет как на тестовые программы, запускаемые через ctest, так и на способность разработчика запускать исполняемые файлы из Visual Studio IDE. Одним из решений этой проблемы является принудительное помещение всех исполняемых файлов и DLL в один выходной каталог, что возможно благодаря различным целевым свойствам ...OUTPUT_DIRECTORY и связанным с ними переменным CMAKE_...OUTPUT_DIRECTORY. Они часто используются для создания схемы каталогов, зеркально отражающей ту, которая используется при установке проекта. Избегайте копирования DLL в правилах постсборки или пользовательских задачах, чтобы поместить их в несколько мест, чтобы другие исполняемые файлы могли их найти. Это хрупко и может привести к ошибочному использованию устаревших DLL.

Тестовые программы в идеале не должны собираться в том же месте, что и основные программы и DLL. Некоторым тестовым программам может потребоваться найти другие файлы относительно их собственного местоположения, поэтому их раздельное хранение может быть даже требованием. Используйте свойство теста ENVIRONMENT для указания соответствующего PATH, чтобы тесты могли найти свои DLL при запуске через ctest. Также рассмотрите возможность использования CMake 3.8 или более поздней версии и определения файла свойств пользователя, о котором тестовые цели могут быть осведомлены с помощью целевого свойства VS_USER_PROPS. Это может быть использовано для расширения среды отладчика, чтобы тесты можно было запускать непосредственно из Visual Studio IDE.

При использовании генератора Visual Studio предпочитайте оставлять настройки PDB по умолчанию. Это обычно приводит к тому, что PDB-файлы появляются в том месте, которое ожидают разработчики, и с именем, соответствующим исполняемому файлу или библиотеке, которым они соответствуют. Попытка изменить выходной каталог PDB-файлов имеет сложности реализации при использовании выражений генератора, и в некоторых случаях может быть трудно поместить PDB-файлы в нужный каталог.

Index

