Оптимизация приложений на платформе .NET

Голдштейн С.
Зурбалев Д.
Флатов И.

ДМК ПРЕСС
издательство
Москв , 2017

Pro .NET Performance

Sasha Goldshtein
Dima Zurbalev
Ido Flatov

Apress®

Голдштейн С.
Зурб лев Д.
Фл тов И.

Оптимизация приложений на платформе .NET

УДК 004.438.NET
ББК 32.973.26-018.2
Г79

Г79 Голдштейн С., Зурб лев Д., Фл тов И. и др.
Оптимиз ция приложений н пл тформе .NET. – Пер. с нгл. Киселев А. Н. – М.: ДМК Пресс, 2017. – 524 с.: ил.

ISBN 978-5-97060-487-8

Увеличение производительности лгоритмов и приложений является чрезвыч йно в жным спектом р зр ботки и может д ть в м преимущество перед конкурент ми, в шим пользов телям обеспечить низкую стоимость вл дения и удовольствие от использов ния быстрых и отзывчивых приложений. Д нн я книг описыв ет внутренние особенности ОС Windows, среды выполнения CLR и пп р тного обеспечения, влияющие н производительность приложений, т кже д ет в м зн ния и инструменты для измерения производительности в шего код в изоляции от внешних ф кторов.

Книг н полнен пример ми код н C# и рекоменд циями, которые помогут в м выж ть м ксимум возможного из в шего приложения – низкое потребление п мяти, согл сов нную н грузку н процессор и миним льное количество опер ций ввод /вывод с сетью и диском.

Изд ние предн зн чено для прогр ммистов, зн комых с языком C# и пл тформой .NET.

УДК 004.438.NET
ББК 32.973.26-018.2

ISBN 978-143-024-458-5 (нгл.)
ISBN 978-5-97060-487-8 (рус.)

Моей любимой жене Дине (Dina), наполнившей мою жизнь солнечным светом.

Моим родителям, Борису (Boris) и Марине (Marina), жертвовавшим всем, чтобы я ни в чем не нуждался.

– Саша

Моей очаровательной жене Ефрат (Efrat), напомнившей мне, что муза не пропала – она просто играет в прятки.

– Идо

ОГЛАВЛЕНИЕ

ГЛАВА 8.
Небезопасный код и взаимодействие с ним ... 364

ГЛАВА 9.
Оптимизация алгоритмов ... 400

ПРЕДИСЛОВИЕ

Платформа Desktop .NET Framework недавно (в феврале 2012) отметила свое 10-летие. Я входил в состав команды с самого ее образования и больше половины этого времени занимался проблемами оптимизации платформы, поэтому 10-й день рождения навел меня на мысли о том, где была платформа .NET, куда она пришла, и какие направления в улучшении производительности можно считать «правильными». Возможность написать предисловие к книге, посвященной производительности приложений на платформе .NET, дала мне шанс изложить свои мысли.

Продуктивность программиста всегда была и будет главной ценностью платформы .NET. Механизм сборки мусора является самой важной особенностью, обеспечивающей продуктивность. Но не только потому, что он устраняет целый класс ошибок (при работе с памятью), но и потому, что позволяет писать библиотеки классов, не загромождая их различными соглашениями о выделении ресурсов (требующими передавать временные буферы или устанавливающими правила о том, кто должен освобождать память). Еще одной важной особенностью является строгий контроль типов (которому теперь подчинены и обобщенные типы (Generics)), позволяющий выявлять намерения программиста и находить многие распространенные ошибки еще до запуска программы. Он также обеспечивает строгое соблюдение контрактов между программными компонентами, что очень важно для библиотек классов и больших проектов. Отсутствие контроля типов в таких языках как JavaScript всегда будет расцениваться как недостаток. Поверх этих двух особенностей мы создали библиотеку классов (весьма однородную, с простыми интерфейсами, непротиворечивыми соглашениями об именовании и так далее), простота использования которой была не последней целью. Я очень горжусь полученным результатом – мы создали систему, дающую возможность разрабатывать программный код с высочайшей продуктивностью.

Однако, одной продуктивности программиста недостаточно, особенно для такой зрелой платформы, как среда выполнения .NET. Нам

также необходима высочайшая производительность приложений, о чем и рассказывает данная книга.

Как нельзя ожидать, что программа будет работать без ошибок с самого первого запуска, так же нельзя ожидать, что программа «просто так» начнет показывать высокую производительность. Существуют не только инструменты (сборщик мусора и контроль типов) и приемы (проверки, интерфейсы), снижающие вероятность появления программных ошибок, но и инструменты (различные профилировщики) и приемы (планирование, создание прототипов, отложенная инициализация), снижающие вероятность появления проблем с производительностью.

Но все не так плохо – производительность приложений подчиняется правилу 90%/10%. Обычно более 90% программного кода не оказывает существенного влияния на производительность приложения в целом и может создаваться программистом с максимальной продуктивностью (в виде коротких строк простейшего кода). Однако другие 10% требуют пристального внимания. Для этого нужен план, и этот план должен быть составлен еще до того как код будет написан. Именно об этом рассказывается в главе 1. Для подобного планирования нужны исходные данные (сведения о скорости выполнения различных операций и библиотечных вызовов), а для этого необходимы инструменты измерения (профилировщики), о которых рассказывается в главе 2. И то, и другое является основой любого проекта высокопроизводительного программного обеспечения. Уделите этим главам особое внимание. Если вы примете их близко к сердцу, это поможет вам писать весьма эффективные программы.

Остальная часть книги посвящена деталям, знание которых потребуется при составлении плана (главы 3, 4 и 5). Но они не заменят базовых знаний характеристик производительности используемой платформы. Если в ходе работы над прототипом выяснится, что простая и прямолинейная реализация не дает желаемой производительности, следует заняться усовершенствованием алгоритма (глава 9) или подумать о возможности включения параллельной обработки (глава 6), потому что они позволяют получить самый большой эффект. Если и после этого производительность оказывается неудовлетворительной, можно прибегнуть к некоторым хитростям .NET (глава 10). Если все вышеперечисленное не дало желаемого эффекта, остается только пожертвовать малой толикой продуктивности программиста и переписать наиболее критичные фрагменты приложения в виде небезопасного и неуправляемого кода (глава 8).

Ключевым моментом, который я хотел бы подчеркнуть, является план (глава 1), потому что именно с него все начинается. В этом плане вы определите наиболее критичные участки программы и выделите дополнительное время на тщательное исследование прототипов этих участков. Затем, вооруженные результатами исследований прототипов и информацией из этой книги, вы без особого труда сможете добиться желаемой производительности. Иногда для этого достаточно будет просто избежать распространенных ловушек, а иногда – применив простейшие оптимизации. Реже вам может понадобиться организовать параллельную обработку данных или написать небезопасный, неуправляемый код. Как бы то ни было, в результате вы сможете поднять производительность 10% кода (потратив дополнительные усилия) и добиться высокой продуктивности, создавая остальные 90%. Это и есть главная цель платформы .NET: высокая продуктивность программиста и высокая производительность кода.

Итак, помните, что высокая производительность не дается бесплатно – для ее достижения требуется составить план, но это не так сложно, и взяв в руки данную книгу вы уже сделали первый и самый важный шаг на пути к созданию *высокопроизводительных приложений* для .NET.

Удачи, и получайте удовольствие от книги. Саша, Дима и Идо постарались для этого.

Вэнс Моррисон (Vance Morrison)
Архитектор производительности, .NET Runtime

ОБ АВТОРАХ

Саша Голдштейн (Sasha Goldshtein) – наиболее ценный специалист (MVP) в Microsoft Visual C# и директор SELA Group. Руководит направлением производительности и отладки в SELA Technology Center, и консультирует по различным темам, включая подготовку приложений к промышленной эксплуатации, решение проблем производительности и создание распределенных архитектур. Наибольший опыт имеет в области разработки приложений на C# и C++, и создания архитектур масштабируемых и высокопроизводительных систем. Часто выступает на конференциях Microsoft и является автором множества курсов, таких как: «Производительность .NET», «Отладка приложений для .NET», «Внутреннее устройство Windows» и многих других.

Блог: http://blog.sashag.net

Twitter: @goldshtn

Дима Зурбалев (Dima Zurbalev) – старший консультант в группе экстренной помощи по вопросам производительности и отладки в SELA Group. Владеет навыками оптимизации производительности и отладки, позволяющими ему решать почти неразрешимые проблемы своих клиентов. Обладает глубокими знаниями внутреннего устройства механизмов CLR и Windows. Большая часть его опыта лежит в плоскости разработки приложений для .NET и инфраструктуры проектов на C++. Является участником ряда открытых проектов на CodePlex.

Блог: http://blogs.microsoft.co.il/blogs/dimaz

Идо Флатов (Ido Flatow) – наиболее ценный специалист (MVP) в Microsoft Connected Systems и старший архитектор в SELA Group. Обладает более чем 15-летним опытом и является одним из экспертов по Windows Azure и веб-технологиям в SELA. Специализируется на таких технологиях, как WCF, ASP.NET, Silverlight и IIS. Является сертифицированным преподавателем Microsoft, соавтор официального курса обучения Microsoft WCF 4.0 (10263A). Часто выступает на конференциях по всему миру.

Блог: http://blogs.microsoft.co.il/blogs/idof

Twitter: @IdoFlatow

О НАУЧНЫХ РЕДАКТОРАХ

Тодд Мейстер (Todd Meister) работает в индустрии информационных технологий уже более 15 лет. Исполнял обязанности научного редактора при создании более 75 изданий, посвященных самым разным технологиям, от SQL Server до .NET Framework. Занимает пост старшего архитектора в области информационных технологий в университете Ball State University, в городе Манси, штат Индиана, США. Живет в центре Индианы со своей женой Кимберли (Kimberly) и пятью детьми.

Фабио Клаудио Феррачиати (Fabio Claudio Ferracchiati) – писатель и научный редактор книг, посвященных самым современным технологиям. Участвовал в создании множества книг на такие темы, как .NET, C#, Visual Basic, SQL Server, Silverlight и ASP.NET. Является сертифицированным разработчиком решений на основе продуктов Microsoft для .NET. Живет в Риме, Италия. Работает в компании Brain Force.

БЛАГОДАРНОСТИ

Создание такой книги, как эта – тяжелый труд, и нам потребовался бы еще как минимум год, если бы не поддержка наших родных и близких, друзей и коллег.

Наш руководитель в SELA Group, Дэвид Басса (David Bassa), оказал нам всю возможную помощь, чтобы мы могли закончить эту книгу, и отнесся с большим пониманием, когда работа над другими проектами затормозилась на заключительных этапах работы над книгой.

Коллектив издательства Apress старался максимально облегчить нам труд и терпел наш скверный английский, не родной для нас. Гвенн (Gwenan), Кевин (Kevin) и Корбин (Corbin): спасибо вам за ваш профессионализм и долготерпение.

И напоследок, но не в последнюю очередь, мы хотим сказать огромное спасибо нашим родным, положившим бесчисленные часы на алтарь этой книги. Без вашей поддержки эта книга никогда не увидела бы свет.

ВВЕДЕНИЕ

Эта книга появилась на свет, потому что на наш взгляд отсутствовало достаточно авторитетное издание, охватывающее все три области, имеющие отношение к производительности приложений на платформе .NET:

- определение показателей производительности и способы их измерения, чтобы можно было проверить, насколько приложение соответствует им или превосходит их;
- приемы улучшения производительности приложений в терминах оптимизации управления памятью, операций ввода/вывода, многопоточного выполнения и так далее;
- полное представление о внутреннем устройстве CLR и .NET для эффективного проектирования высокопроизводительных приложений и исправления проблем с производительностью по мере их появления.

Мы полагаем, что разработчики приложений на платформе .NET не могут создавать высокопроизводительные программные решения, не имея полного понимания во всех трех областях. Например, управление памятью в .NET – чрезвычайно сложная тема (хотя и облегчаемая механизмом автоматической сборки мусора в CLR) и является важнейшей причиной появления проблем с производительностью, таких как утечки памяти и длинные паузы, вызванные работой механизма сборки мусора. Без понимания особенностей действия этого механизма высокопроизводительное управление памятью в .NET просто невозможно. Точно так же, чтобы осознанно выбрать наиболее подходящий класс коллекций из множества, что может предложить платформа .NET Framework, или принять решение о создании собственного класса, требуется полное понимание механики работы кешей центрального процессора, среды выполнения и средств синхронизации.

Главы в этой книге упорядочены так, чтобы читать их последовательно, но вы можете свободно перемещаться между ними взад и впе-

ред, восполняя пробелы, когда это необходимо. Главы организованы в следующие логические части.

- Главы 1 и 2 описывают показатели производительности и способы их измерения. В них будут представлены инструменты оценки производительности приложения.
- Главы 3 и 4 подробно описывают внутренние особенности общеязыковой среды выполнения (Common Language Runtime, CLR). Основное внимание в них уделяется внутренней организации типов и реализации механизма сборки мусора – двум важнейшим темам, знание которых поможет улучшить производительность приложений, где управление памятью имеет особое значение.
- В главах 5, 6, 7, 8 и 11 обсуждаются конкретные темы, касающиеся платформы .NET Framework и CLR, знание которых дает дополнительные возможности оптимизации – правильное использование коллекций, организация параллельного выполнения кода, оптимизация операций ввода/вывода, эффективное применение механизмов взаимодействий и увеличение производительности веб-приложений.
- Глава 9 является кратким введением в теорию сложности и алгоритмы. Ее цель – дать представление об особенностях оптимизации алгоритмов.
- Глава 10 рассматривает самые разные темы, не укладывающиеся в другие главы, включая приемы оптимизации времени запуска приложения, применение исключений и механизма рефлексии .NET Reflection.

Для лучшего понимания некоторых из этих тем необходимо обладать определенными знаниями. В этой книге мы будем полагать, что читатель обладает существенным опытом разработки приложений на языке C# для платформы .NET Framework, а также знаком с основными понятиями, включая:

- Windows: потоки выполнения, механизмы синхронизации и виртуальная память;
- общеязыковая среда выполнения (Common Language Runtime, CLR): динамический компилятор (Just-In-Time, JIT), промежуточный язык Microsoft (Microsoft Intermediate Language, MSIL), сборка мусора;
- устройство компьютера: основная память, кеш, диск, графическая карта, сетевой интерфейс.

В этой книге приводится достаточно много примеров программ, фрагментов кода и эталонных тестов. В интересах экономии места мы часто будем включать лишь часть кода, однако на веб-сайте книги вы можете получить полные исходные тексты примеров.

В некоторых главах мы используем фрагменты на языке ассемблера для микропроцессоров серии x86, чтобы проиллюстрировать некоторые особенности работы CLR или более полно объяснить те или иные приемы оптимизации. Хотя эти пояснения не являются критически важными для понимания обсуждаемых тем, тем не менее, мы рекомендуем читателям найти время и познакомиться с основами языка ассемблера x86. Отличным ресурсом в этом отношении может служить книга Рэндалла Хайда (Randall Hyde) «The Art of Assembly Language Programming», имеющаяся в свободном доступе (http://www.artofasm.com/Windows/index.html).

Наконец, в этой книге описывается множество инструментов измерения производительности, дается масса советов и рекомендаций по улучшению потребительских качеств и скорости выполнения приложений, теоретических обоснований, лежащих в основе механизмов CLR, практических примеров кода и случаев из практики авторов. В течение почти десяти лет мы занимались оптимизацией приложений для наших клиентов и проектированием высокопроизводительных систем, что называется «с нуля». За эти годы мы научили сотни разработчиков думать о производительности на каждом этапе разработки программ и активно искать возможности повышения их потребительских качеств. Прочитав эту книгу, вы присоединитесь к разработчикам высокопроизводительных программ для платформы .NET и исследователей в области оптимизации существующих приложений.

Саша Голдштейн (Sasha Goldshtein)
Дима Зурбалев (Dima Zurbalev)
Идо Фламов (Ido Flatow)

ГЛАВА 1. Характеристики производительности

Прежде чем приступать к исследованию проблем производительности в мире .NET, необходимо понять, какие характеристики производительности существуют и в чем заключаются требования к производительности. В главе 2 мы исследуем более десятка профилировщиков и инструментов мониторинга, однако, чтобы эффективно использовать их, необходимо знать, какие показатели представляют интерес.

К разным приложениям предъявляются разные требования в смысле производительности, которые определяются различными потребностями. В одних случаях характеристики производительности явно определяются архитектурой приложения: например, чтобы веб-сервер мог обслуживать миллионы пользователей, необходимо создать распределенную систему из нескольких серверов, обеспечивающую поддержку кеширования и распределения нагрузки. В других, результаты тестирования производительности могут потребовать изменения самой архитектуры приложения: нам не раз приходилось видеть системы, перепроектировавшиеся до самого основания после проведения нагрузочных испытаний, как и системы, не выдерживавшие эксплуатационной нагрузки.

По своему опыту можем сказать, что выяснение требуемых параметров производительности и ограничений среды выполнения часто составляют половину успеха. Ниже приводится несколько примеров проблем, которые мы смогли выявить и исправить за последние несколько лет.

- Исследуя серьезные проблемы производительности веб-сервера в одном из центров обработки данных, мы обнаружили, что они были вызваны использованием для тестирования 4 Мбитного канала с низкой задержкой. Не понимая, какой из показателей производительности играет наиболее важную

роль, инженеры потратили несколько десятков дней на оптимизацию самого веб-сервера, который и без того показывал отличную производительность.

- Мы сумели улучшить скорость прокрутки в приложении с графическим интерфейсом за счет тонкой настройки механизма сборки мусора – программного компонента, не имеющего очевидной связи с проблемой. Точный хронометраж операций распределения памяти и настройка поведения сборщика мусора помогли установить и устранить причины задержки в работе графического интерфейса, так раздражавшие пользователей.
- Нам удалось увеличить скорость компиляции в десять раз, подключив жесткий диск к порту SATA и тем самым устранив влияние ошибки в драйвере SCSI-дисков от Microsoft.
- Мы смогли уменьшить размеры сообщений для обмена данными со службой WCF на 90%, заметно улучшив ее масштабируемость и использование CPU за счет настройки механизма сериализации WCF.
- Мы сократили время запуска крупного приложения, насчитывающего 300 сборок, с 35 до 12 секунд на устаревшем оборудовании, за счет сжатия кода приложения и тщательного исследования его зависимостей, с целью выявления тех, что не требуются немедленно на этапе запуска.

Эти примеры наглядно иллюстрируют, что любые системы, от мобильных устройств с низким энергопотреблением до высокопроизводительных рабочих станций с мощными графическими возможностями и распределенных систем, обладают уникальными характеристиками производительности, на которые влияет бесчисленное множество разнообразных факторов. В этой главе мы коротко исследуем разные характеристики производительности и требования к ней в типичном современном программном обеспечении. В следующей главе мы расскажем, как точно измерять эти характеристики, а в остальной части книги – как повышать их.

Требования к производительности

Требования к производительности во многом зависят от назначения приложения и его архитектуры. Выяснив круг общих требований к

приложению, необходимо определить требования к производительности. В зависимости от процесса разработки программного обеспечения, эти требования возможно придется скорректировать с учетом изменения прежних и появления новых потребностей. Мы приведем несколько примеров и рекомендаций по определению требований к производительности, но вы должны понимать, что эти рекомендации, как и все, что касается производительности, необходимо адаптировать к конкретным условиям, с учетом назначения приложения.

Для начала перечислим несколько примеров *неправильно* сформулированных требований:

- приложение должно сохранять высокую отзывчивость при одновременном доступе нескольких пользователей;
- приложение не должно использовать большой объем памяти при небольшом количестве посетителей;
- единственный сервер базы данных должен быстро обслуживать запросы, даже при наличии нескольких серверов приложений, действующих под высокой нагрузкой.

Основная проблема этих требований в том, что они являются слишком общими и субъективными. Если попытаться вникнуть в их суть, можно обнаружить, что в разных системах отсчета они могут интерпретироваться по-разному. Бизнес-аналитик может считать, что 100 000 одновременно работающих пользователей – это «немного», а технический специалист может с уверенностью сказать, что такое количество пользователей невозможно обслужить, имея единственную машину. Аналогично, разработчик может считать пользовательский интерфейс с задержками 500 мсек вполне «отзывчивым», а эксперт в пользовательских интерфейсах оценивать это как серьезный недостаток.

Требования к производительности должны выражаться *в показателях, которые можно измерить* некоторым способом. Кроме того, требования к производительности должны содержать некоторую информацию об *окружении* – общую или конкретную для этой цели. Ниже перечислены некоторые примеры правильно сформулированных требований:

- приложение должно обслуживать каждую страницу в категории «Важные» не дольше 300 мсек (не включая задержки в сети), при условии одновременного обслуживания не более 5000 пользователей;
- приложение должно потреблять не более 4 Кбайт памяти на каждый неактивный сеанс с пользователем;

- нагрузка на CPU и используемый объем жесткого диска на сервере баз данных не должны превышать 70%, а время обработки запросов из категории «Общие» не должно превышать 75 мсек, при условии одновременного обслуживания не более 10 серверов приложений.

Примечание. *В этих примерах предполагается, что категория страниц «Важные» и категория запросов «Общие» четко определены бизнес-аналитиками и архитекторами приложения. Определение требований к производительности для каждого аспекта приложения часто неразумно и не стоит затраченных усилий.*

Теперь рассмотрим несколько примеров требований к производительности для типичных приложений (табл. 1.1). Этот список не является исчерпывающим и не может служить контрольным списком или шаблоном для определения ваших собственных требований – это всего лишь общий обзор различий в требованиях, предъявляемых к приложениям разных типов.

Таблица 1.1. Примеры требований к производительности для типичных приложений

Тип системы	Требование к производительности	Ограничения окружения
Внешний веб-сервер	Время от момента получения запроса до момента создания полного ответа не должно превышать 300 мсек	При условии одновременной обработки не более 300 запросов
Внешний веб-сервер	Объем используемой виртуальной памяти (включая кеш) не должен превышать 1.3 Гбайт	При условии одновременной обработки не более 300 запросов и не более 5000 открытых сеансов пользователей
Сервер приложений	Нагрузка на CPU не должна превышать 75%	При условии не более 1000 одновременно обслуживаемых запросов к API
Сервер приложений	Частота следования ошибок обращений к страницам диска не должна превышать 2 ошибок в сек	При условии не более 1000 одновременно обслуживаемых запросов к API
Клиентское приложение	Время от двойного щелчка на ярлыке до появления полного списка клиентов не должно превышать 1500 мсек	–

Таблица 1.1. (окончание)

Тип системы	Требование к производительности	Ограничения окружения
Клиентское приложение	Нагрузка на CPU в режиме простоя приложения не должна превышать 1%	–
Веб-страница	Время фильтрации и сортировки списка входящих электронных писем не должно превышать 750 мсек, включая время, затрачиваемое на воспроизведение анимационных эффектов	При отображении не более 200 электронных писем
Веб-страница	Объем памяти для кеширования объектов JavaScript в окне не должен превышать 2.5 Мбайт	–
Служба мониторинга	Время от возникновения ошибки до вывода предупреждения не должно превышать 25 мсек	–
Служба мониторинга	Частота следования операций дискового ввода/вывода в отсутствие активных предупреждений должна быть равна 0	–

Примечание. *Параметры аппаратного обеспечения, на котором выполняется приложение, играют ключевую роль в определении ограничений среды. Например, требование к скорости запуска клиентского приложения может потребовать использования твердотельного накопителя или жесткого диска со скоростью вращения шпинделя не менее 7200 об/мин, по меньшей мере 2 Гбайт системной памяти и процессора с тактовой частотой 1.2 Ггц или выше, поддерживающего набор инструкций SSE3. Эти требования к аппаратному обеспечению не стоит повторять в описании каждого требования к производительности, но о них следует помнить.*

Когда требования к производительности сформулированы, тестирование производительности, нагрузочное тестирование и последующий процесс оптимизации становятся тривиальным делом. Проверка таких требований, как «частота следования ошибок обращений к страницам диска не должна превышать 2 ошибок в сек, при условии не более 1000 одновременно обслуживаемых запросов к API» часто может потребовать применения инструментов нагрузочного тестирования и подходящего аппаратного обеспечения. В следующей главе рассматриваются приемы измерения производительности приложений, позволяющие определить степень соответствия предъявляемым требованиям.

Для формулировки требований к производительности нередко необходимо иметь знакомство с характеристиками производительности, о которых рассказывается ниже.

Характеристики производительности

В отличие от требований к производительности, характеристики производительности не связаны с каким-то конкретным типом приложения или окружением. Характеристики производительности – это числовое выражение особенностей поведения приложения. Они могут измеряться на любом аппаратном обеспечении и в любом окружении, независимо от количества активных пользователей, запросов или сеансов. В процессе разработки вы выбираете характеристики для измерения и на их основе определяете требования к производительности.

Некоторые приложения имеют характеристики, специфичные для своей области. Мы не будем пытаться идентифицировать их здесь. Вместо этого перечислим в табл. 1.2 характеристики, наиболее важные для большинства приложений, наряду с главами в этой книге, где они обсуждаются. (Нагрузка на CPU и время выполнения являются настолько важными характеристиками, что упоминаются в каждой главе.)

Таблица 1.2. Список характеристик производительности (частичный)

Характеристика	Единицы измерения	Главы в книге, где обсуждается
Нагрузка на CPU	Процент	Все главы
Использование физической/виртуальной памяти	Байты, килобайты, мегабайты, гигабайты	Глава 4 – Сборка мусора Глава 5 – Коллекции и обобщенные типы
Попадания в кеш	Количество, частота попаданий в секунду	Глава 5 – Коллекции и обобщенные типы Глава 6 – Конкуренция и параллелизм
Ошибки обращений к страницам диска	Количество, частота следования в секунду	–

Таблица 1.2. (окончание)

Характеристика	Единицы измерения	Главы в книге, где обсуждается
Обращения к базе данных, количество/время	Количество, частота следования в секунду, миллисекунды	–
Выделение блоков памяти	Количество байтов, количество объектов, частота следования в секунду	Глава 3 – Внутреннее устройство типов Глава 4 – Сборка мусора
Время выполнения	Миллисекунды	Все главы
Сетевые операции	Количество, частота следования в секунду	Глава 7 – Сети, ввод/вывод и сериализация Глава 11 – Веб-приложения
Дисковые операции	Количество, частота следования в секунду	Глава 7 – Сети, ввод/вывод и сериализация
Время отклика	Миллисекунды	Глава 11 – Веб-приложения
Сборка мусора	Количество, частота следования в секунду, продолжительность, % от общего времени выполнения	Глава 4 – Сборка мусора
Исключительные ситуации	Количество, частота следования в секунду	Глава 10 – Приемы оптимизации
Время запуска	Миллисекунды	Глава 10 – Приемы оптимизации
Конкуренция	Количество, частота следования в секунду	Глава 6 – Конкуренция и параллелизм

Одни характеристики более важны для одних типов приложений, другие – для других. Например, время доступа к базе данных не является характеристикой, которую можно измерить на стороне клиента. В число наиболее типичных комбинаций характеристик производительности входят:

- для клиентских приложений может оказаться важным время запуска, объем используемой памяти и нагрузка на CPU;
- для серверных приложений, составляющих основу системы, обычно большое значение имеют нагрузка на CPU, попадание в кеш, количество соединений и интенсивность операций с динамической памятью, а также характеристики работы сборщика мусора;

- в веб-приложениях основное внимание, как правило, уделяется объему используемой памяти, операциям доступа к базе данных, сетевым и дисковым операциям и времени отклика.

И последнее наблюдение, касающееся характеристик производительности, – уровень детализации измерений часто может изменяться существенно, что, впрочем, никак не влияет на важность самих характеристик. Например, время выделения памяти или время выполнения могут измеряться на уровне системы, на уровне единственного приложения или даже на уровне отдельных методов или строк. Время выполнения одного конкретного метода может оказаться гораздо важнее, чем общая нагрузка на CPU или время выполнения процесса в целом. К сожалению, увеличение уровня детализации измерений часто влечет за собой дополнительные накладные расходы, как будет показано в следующей главе при обсуждении различных инструментов профилирования.

Место производительности в цикле разработки

Куда в цикл разработки следует вставить решение вопросов, связанных с производительностью? Этот, казалось бы невинный вопрос, влечет за собой необходимость *модернизации* всего процесса разработки. Для этих целей можно было бы выделить отдельный этап. Однако правильнее уделять внимание проблемам производительности на каждом этапе разработки приложения: сначала следует выявить наиболее важные характеристики и определить требования к производительности; затем выяснить, насколько полно приложение отвечает этим требованиям; и, наконец, на этапе дальнейшего развития и эксплуатации следить, как изменяются показатели производительности.

1. На этапе определения требований к приложению начинайте одновременно выяснять параметры производительности, которые было бы желательно получить.
2. На этапе проектирования очертите круг наиболее важных характеристик производительности для вашего приложения и определите конкретные их значения.
3. На этапе разработки как можно чаще выполняйте тестирование производительности прототипов или незавершенных функций, чтобы убедиться, что их производительность находится в установленных границах.
4. На этапе тестирования выполняйте всеобъемлющие нагрузочные испытания и тестирование производительности, чтобы убедиться, что производительность системы в целом соответствует установленным требованиям.
5. В дальнейшем, во время разработки и сопровождения, выполняйте дополнительные нагрузочные испытания и тестирование производительности перед выпуском каждой новой версии (желательно еже-

дневно или хотя бы раз в неделю), чтобы быстро выявить снижение производительности, вызванное добавлением новых или изменением существующих функций.

Создание комплектов нагрузочных тестов и тестов производительности, настройка изолированного окружения для их выполнения, и тщательный анализ результатов, чтобы гарантировать отсутствие снижения производительности, часто отнимает достаточно много времени. Однако преимущества, которые вы получаете взамен, в виде гарантий, что снижение производительности не окажется незамеченным, стоят дополнительных усилий и затрат времени в процессе разработки.

В заключение

Эта глава играет роль введения в мир показателей производительности и требований к ней. Знание, какие показатели и критерии являются наиболее значительными в каждом конкретном случае, может быть даже важнее самих *приемов измерения* производительности, о которых пойдет речь в следующей главе. В оставшейся части книги мы покажем, как измерять производительность с применением самых разных инструментов и дадим рекомендации по улучшению и оптимизации приложений.

ГЛАВА 2. Измерение производительности

Эта книга посвящена вопросам повышения производительности приложений для .NET. Следует понимать, что нельзя улучшить какие-либо характеристики, не измерив их предварительно. Именно поэтому первая наиболее существенная глава посвящена инструментам и приемам измерения производительности. Построение догадок и преждевременных выводов об узких местах в приложении – это самое худшее, что может сделать разработчик. Как было показано в главе 1, существует множество любопытных характеристик производительности, которые могут играть важную роль в оценке производительности приложения в целом, а в этой главе мы узнаем, как измерять их.

Подходы к измерению производительности

Измерение производительности приложений может выполняться разными способами, во многом зависящих от контекста, сложности приложения, типа требуемой информации и точности получаемых результатов.

Один из подходов к тестированию небольших программ или библиотечных методов называется *тестированием по принципу «стеклянного ящика»*. Этот подход основан на исследовании исходного кода, анализе его сложности, изменении и добавлении кода, выполняющего измерения. Этот подход, который иногда называется *микрохронометраж* (microbenchmarking), будет обсуждаться ближе к концу этой главы; он особенно ценен – и часто незаменим – когда требуется высочайшая точность результатов хронометража неболь-

ших фрагментов кода, где каждая машинная инструкция на счету, но требует больших трудозатрат при оценке больших приложений. Кроме того, не зная наперед, какие фрагменты программ следует тестировать, выявление узких мест без применения автоматизированных инструментов может оказаться чрезвычайно трудной задачей.

Для больших программ чаще используется подход, называемый *тестированием по принципу «черного ящика»*, когда параметры производительности определяются человеком и затем измеряются с применением инструментов. Применяя этот подход, разработчик не должен строить какие-либо гипотезы об узких местах в приложении. В этой главе мы познакомимся с большим количеством инструментов, автоматически анализирующих производительность приложения и предоставляющих результаты измерений в простом и понятном виде. В числе этих инструментов будут упомянуты *счетчики производительности* (performance counters), *механизм трассировки событий для Windows* (Event Tracing for Windows, ETW) и коммерческие *профилировщики*.

В процессе чтения этой главы помните, что инструменты измерения производительности сами могут отрицательно влиять на производительность. Лишь немногие из них способны дать точную информацию, не добавляя свои накладные расходы. Перемещаясь от одного инструмента к другому, всегда помните, что точность инструмента часто искажается накладными расходами, которые они вносят при применении к приложению.

Встроенные инструменты Windows

Прежде, чем обратиться к коммерческим инструментам, требующим предварительной установки, познакомимся сначала с инструментами, которые может предложить Windows «из коробки». Счетчики производительности (performance counters) являются составной частью Windows вот уже почти два десятилетия. Не так давно (в 2006 г.) в Windows Vista появился еще один инструмент хронометража – механизм трассировки событий для Windows (Event Tracing for Windows). Оба входят в состав всех разновидностей Windows и могут использоваться для оценки производительности с минимальными накладными расходами.

Счетчики производительности

Счетчики производительности – это встроенный механизм Windows, позволяющий оценивать производительность и состояние системы в целом. С помощью счетчиков производительности пользователи и администраторы могут исследовать работу различных компонентов, включая ядро Windows, драйверы, базы данных и CLR. Как дополнительное преимущество, счетчики производительности для подавляющего большинства компонентов системы уже включены по умолчанию, поэтому вам не придется вносить поправки на дополнительные накладные расходы, связанные с их использованием.

Прочитать информацию из счетчиков производительности в локальной или удаленной системе чрезвычайно просто. Встроенный инструмент *Performance Monitor* (Системный монитор) (*perfmon.exe*) может отображать все счетчики производительности, доступные в системе и сохранять информацию в файле для последующего изучения, а также автоматически генерировать оповещения, когда значение того или иного счетчика производительности превышает некоторый установленный порог. Инструмент Performance Monitor (Системный монитор) может также подключаться к удаленным системам, если вы обладаете привилегиями администратора и возможностью подключения к удаленной системе по локальной сети.

Информация о производительности имеет иерархическую организацию, как описывается ниже.

Категории счетчиков производительности (или *объектов производительности*) представляют наборы отдельных счетчиков для определенных компонентов системы. В качестве примеров категорий можно привести: **.NET CLR Memory** (Память CLR .NET), **Processor Information** (Процессор), **TCPv4** и **PhysicalDisk** (Физический диск)[1].

- *Счетчики производительности* – это отдельные числовые свойства в категориях. Обычно принято указывать категорию и название счетчика производительности, разделяя их обратным слешем, например, **Process\Private Bytes** (Процесс\Байт исключительного пользования). Счетчики производительности могут иметь разные типы, включая простые числовые значения (**Process\Thread Count** (Процесс\Счетчик потоков)), скорости следования событий (**Print Queue\Bytes Printed/sec** (Очередь печати\Печатаемых байт/сек)), проценты

1 Здесь и далее перевод пунктов взят из русифицированной версии Windows.

(**PhysicalDisk\%Idle Time** (Физический диск\% активности диска)) и средние значения (**ServiceModelOperation 3.0.0.0\ Calls Duration** (Показатели работы служб 3.0.0.0\Продолжительность вызова)).

- *Экземпляры категорий счетчиков производительности* (вхождения) используются с целью создания разных наборов счетчиков для разных экземпляров компонентов системы. Например, в системе может иметься несколько процессоров, поэтому для каждого из них имеется свой экземпляр в категории **Processor Information** (Сведения о процессоре), а также общий экземпляр `_Total`). Одни категории счетчиков производительности могут иметь несколько экземпляров (таковых большинство), другие – единственный экземпляр (например, категория **Memory** (Память)).

Исследовав полный список счетчиков производительности, предоставляемых типичной системой Windows, где выполняются приложения для .NET, легко понять, что многие проблемы производительности могут быть выявлены без использования других инструментов. По крайней мере, счетчики производительности позволяют получить общее представление о причинах низкой производительности, а исследование файлов журналов поможет понять, насколько поведение системы отличается от нормального.

Ниже перечислены ситуации, когда системный администратор или разработчик, занимающийся проблемами производительности, сможет получить представление о том, где находится узкое место в приложении, прежде чем применить более мощные инструменты.

- Если в приложении присутствуют утечки памяти, счетчики производительности помогут выяснить, какие операции выделения памяти – низкоуровневые или управляемые – являются источником этих утечек. Для этого достаточно сопоставить счетчик **Process\Private Bytes** (Процесс\Байт исключительного пользования) со счетчиком **.NET CLR Memory\# Bytes in All Heaps** (Память CLR .NET\Байт во всех кучах). Первый подсчитывает объем всей памяти, выделенной для процесса (включая кучу сборщика мусора), а второй – только объем управляемой памяти. (См. рис. 2.1.)
- Если приложение ASP.NET начинает проявлять необычное поведение, счетчики из категории **ASP.NET Applications** (Приложения ASP.NET) позволят уточнить, что именно пошло не так. Например, счетчики **Requests/Sec** (Запросов/сек),

Requests Timed Out (Запросов с истекшим временем ожидания), **Request Wait Time** (Запросов в очереди приложений) и **Requests Executing** (Выполняется запросов) помогут выявить состояния пиковых нагрузок. Счетчик **Errors Total/Sec** (Общее число ошибок/сек) покажет, не столкнулось ли приложение с необычно большим количеством исключений. А различные счетчики, имеющие отношение к механизму кеширования покажут, насколько эффективно работает этот механизм.

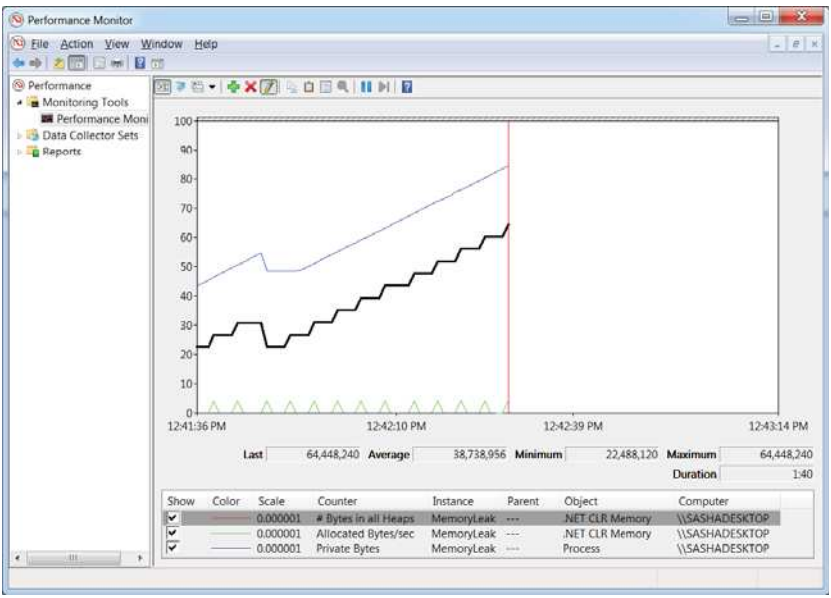

Рис. 2.1. Главное окно программы Performance Monitor (Системный монитор) с тремя счетчиками для определенного процесса. Верхняя линия на графике – значения счетчика Process\Private Bytes (Процесс\Байт исключительного пользования), средняя линия – значения счетчика .NET CLR Memory\# Bytes in all Heaps (Память CLR .NET\Байт во всех кучах), и нижняя линия – значения счетчика .NET CLR Memory\Allocated Bytes/sec (Память CLR .NET\Выделено байт/сек). На основании этого графика можно заключить, что в приложении имеется утечка памяти в куче сборщика мусора.

- Если WCF-служба, опирающаяся на взаимодействия с базами данных и распределенные транзакции, оказывается не в состоянии справиться с текущей нагрузкой, уточнить источник проблем поможет категория **Service Model Service**

(Показатели работы служб) – счетчики **Calls Outstanding** (Текущих вызовов), **Calls Per Second** (Вызовов/сек) и **Calls Failed Per Second** (Неудачных вызовов/сек) помогут идентифицировать состояние тяжелой нагрузки, счетчик **Transactions Flowed Per Second** (Транзакций в данной операции/сек) покажет частоту транзакций, выполняемых службой. А счетчики из категорий, имеющих отношение к SQL Server, таких как **MSSQL$INSTANCENAME:Transactions** и **MSSQL$INSTANCENAME:Locks** укажут на проблемы выполнения транзакций, чрезмерное количество блокировок и даже взаимоблокировок.

Мониторинг использования памяти с применением счетчиков производительности

В этом коротком эксперименте предлагается провести мониторинг использования памяти демонстрационным приложением и с помощью программы Performance Monitor (Системный монитор) выявить наличие утечек памяти, как описывалось выше.

1. Запустите программу Performance Monitor (Системный монитор) – это можно сделать, отыскав пункт **Performance Monitor** (Системный монитор) в меню **Start** (Пуск) или запустив программу *perfmon.exe* непосредственно.
2. Запустите приложение *MemoryLeak.exe* из папки с примерами для этой главы.
3. Щелкните на пункте **Performance Monitor** (Системный монитор) в панели слева и затем щелкните на кнопке с изображением зеленого плюса (+).
4. В категории **.NET CLR Memory** (Память CLR .NET) выберите счетчики **# Bytes in all Heaps** (Байт во всех кучах) и **Allocated Bytes/sec** (Выделено байт/сек), в списке справа выберите экземпляр **MemoryLeak** и щелкните на кнопке **Add >>** (Добавить >>).
5. В категории **Process** (Процесс) выберите счетчик **Private Bytes** (Байт исключительного пользования), в списке справа выберите экземпляр **MemoryLeak** и щелкните на кнопке **Add >>** (Добавить).
6. Щелкните на кнопке **OK**, чтобы подтвердить свой выбор и понаблюдайте за изменениями на графике.
7. Вам может понадобиться щелкнуть правой кнопкой мыши на строке со счетчиком в таблице, находящейся внизу окна и выбрать пункт контекстного меню **Scale selected counters** (Масштабировать выделенные счетчики), чтобы линии появились на графике.

Вы должны увидеть, что линии, соответствующие счетчикам **Private Bytes** (Байт исключительного пользования) и **# Bytes in all Heaps** (Байт во всех кучах) изменяются синхронно (как на рис. 2.1). Это указывает на утечки памяти в управляемой куче. Мы еще вернемся к данному примеру в главе 4 и раскроем причину утечки.

Совет. *В типичной системе Windows существуют, буквально, тысячи счетчиков производительности. И ни один, даже самый опытный разработчик, не в состоянии запомнить назначение их всех. Поэтому в диалоге «Add Counters» (Добавить счетчики) есть возможность отметить флажок «Show description» (Отображать описание). Когда флажок установлен, в нижней части окна будет отображаться дополнительное описание, которое сообщит, например, что счетчик «System\Processor Queue Length» (Система\Длина очереди процессора) – это количество потоков выполнения, ожидающих своей очереди, или, что счетчик «.NET CLR Locks And Threads\Contention Rate/sec» (Блокировки и потоки .NET CLR\Частота конфликтов/сек) – это количество неудачных попыток (в секунду) предпринятых потоками выполнения, чтобы получить управляемую блокировку.*

Журналы и оповещения производительности

Добавить сохранение в журнал значений счетчиков производительности очень просто, и есть даже возможность передать системным администраторам XML-шаблон, чтобы они с его помощью могли добавить автоматическую запись счетчиков без необходимости делать это вручную. После записи данных, журнал можно открыть на любом компьютере и проиграть его, как если это были оперативные данные. (Есть даже некоторые встроенные наборы счетчиков, которые не требуется настраивать вручную для записи в журнал.)

Инструмент Performance Monitor (Системный монитор) позволяет также определять настройки оповещений – выполнения определенных заданий при превышении указанными счетчиками установленных пороговых значений. Данную возможность можно использовать для создания упрощенной инфраструктуры мониторинга, способной отправлять электронные письма или сообщения системному администратору при нарушении ограничений производительности. Например, оповещение можно настроить так, что оно автоматически будет перезапускать процесс при достижении опасного предела используемого объема памяти, или когда система исчерпает все свободное пространство на диске. Мы настоятельно рекомендуем поэкспериментировать с системным монитором, чтобы поближе познакомиться с предлагаемыми им возможностями.

Настройка записи значений счетчиков в журнал

Чтобы настроить запись значений счетчиков в журнал, откройте Performance Monitor (Системный монитор) и выполните описываемые ниже действия. (Здесь предполагается, что вы пользуетесь Windows 7 или Windows Server 2008 R2. В предыдущих версиях операционной системы системный монитор имел несколько иной интерфейс – если вы пользуетесь такими версиями, обращайтесь к документации за более подробными инструкциями.)

1. В дереве слева разверните ветку **Data Collector Sets** (Группы сборщиков данных).
2. Щелкните правой кнопкой мыши на пункте **User Defined** (Определяемые пользователем) и выберите пункт **New** → **Data Collector Set** (Создать → Группа сборщиков данных) контекстного меню.
3. Введите имя группы, выберите радиокнопку **Create manually (Advanced)** (Создать вручную (для опытных)) и щелкните на кнопке **Next** (Далее).
4. Выберите радиокнопку **Create data logs** (Создать журналы данных), отметьте флажок **Performance counter** (Счетчик производительности) и щелкните на кнопке **Next** (Далее).
5. Щелкните на кнопке **Add** (Добавить) и добавьте счетчики производительности (в открывшемся стандартном диалоге **Add Counters** (Добавить счетчики)). Закончив добавление, настройте значение в поле **Sample Interval** (Интервал выборки) (по умолчанию замеры производятся один раз в 15 секунд) и щелкните на кнопке **Next** (Далее).
6. Укажите каталог, где будут сохраняться журналы и щелкните на кнопке **Next** (Далее).
7. Выберите радиокнопку **Open properties for this data collector set** (Открыть свойства группы сборщиков данных) и щелкните на кнопке **Finish** (Готово).
8. В открывшемся диалоге выполните настройки на разных вкладках – здесь можно определить расписание для автоматического запуска, условия останова (например, после сбора определенного объема данных) и задание, которое следует запустить после прекращения сбора данных (например, выгрузить результаты в централизованное хранилище). Завершив настройки, щелкните на кнопке **OK**.
9. Разверните ветку дерева **User Defined** (Определяемые пользователем), щелкните правой кнопкой мыши на вновь созданной группе сборщиков данных и выберите пункт контекстного меню **Start** (Пуск).
10. В результате начнется накопление данных в журнале, хранящемся в выбранном вами каталоге. Сбор данных можно остановить в любой момент, щелкнув на группе правой кнопкой мыши и выбрав пункт контекстного меню **Stop** (Стоп).

Когда после завершения сбора данных вам потребуется исследовать их с помощью системного монитора, выполните следующие действия:

1. Разверните ветку дерева **User Defined** (Определяемые пользователем).
2. Щелкните правой кнопкой мыши на группе сборщиков данных и выберите пункт контекстного меню **Latest Report** (Последний отчет).
3. В появившемся окне вы сможете добавить или удалить счетчики из списка в журнале, настроить диапазон изменения времени и масштаб изменения данных, щелкнув правой кнопкой мыши на графике и выбрав пункт контекстного меню **Properties** (Свойства).

Наконец, чтобы проанализировать данные на другом компьютере, необходимо скопировать каталог с журналами на этот компьютер, открыть ветку дерева Performance Monitor (Системный монитор) и щелкнуть на второй кнопке слева в панели инструментов (или нажать комбинацию клавиш **Ctrl** + **L**). В появившемся диалоге выберите радиокнопку **Log files** (Файлы журнала) и добавьте файлы с помощью кнопки **Add** (Добавить...).

Собственные счетчики производительности

Системный монитор – чрезвычайно удобный инструмент, однако значения счетчиков производительности можно читать из любого приложения для .NET, с помощью класса `System.Diagnostics.PerformanceCounter`. Более того, можно даже создавать собственные счетчики производительности и добавлять их к множеству уже имеющихся.

Ниже описаны некоторые ситуации, когда может пригодиться создание собственных категорий счетчиков:

- При разработке библиотеки, используемой как часть большой системы. Посредством счетчиков библиотека может сообщать информацию о производительности, что часто намного проще для разработчиков и системных администраторов, чем копаться в файлах журналов или в исходном коде.
- При разработке серверной системы, принимающей нестандартные запросы, обрабатывающей их и возвращающей ответы (например, нестандартного веб-сервера или веб-службы), может потребоваться оценить скорость обработки запросов, количество встречающихся ошибок и другие похожие параметры. (Дополнительные подсказки можно найти в категории счетчиков производительности ASP.NET.)
- При разработке высоконадежной службы Windows, которая выполняется без контроля со стороны человека и обменивается данными с нестандартным оборудованием. С помощью счетчиков производительности служба может сообщать о состоянии этого оборудования, о частоте следования операций обмена с ним и другие параметры.

Следующий фрагмент кода – это все, что необходимо, чтобы экспортировать из приложения категорию счетчиков производительности, имеющую единственный экземпляр, и обновлять их периодически. Предполагается, что класс `AttendanceSystem` хранит информацию о количестве зарегистрировавшихся к настоящему моменту пользователей, и вы хотите экспортировать эту информацию в виде счетчика производительности. (Чтобы скомпилировать этот фрагмент, потребуется добавить пространство имен `System.Diagnostics`.)

```
public static void CreateCategory() {
    if (PerformanceCounterCategory.Exists(“Attendance”)) {
        PerformanceCounterCategory.Delete(“Attendance”);
    }
    CounterCreationDataCollection counters = new CounterCreation-
```

```
DataCollection();
    CounterCreationData employeesAtWork = new CounterCreationData(
       "# Employees at Work", "The number of employees currently checked in.",
        PerformanceCounterType.NumberOfItems32);
    PerformanceCounterCategory.Create(
        "Attendance", "Attendance information for Litware, Inc.",
        PerformanceCounterCategoryType.SingleInstance, counters);
}

public static void StartUpdatingCounters() {
    PerformanceCounter employeesAtWork = new PerformanceCounter(
        "Attendance", "# Employees at Work", readOnly: false);
    updateTimer = new Timer(_ = > {
        employeesAtWork.RawValue = AttendanceSystem.Current.EmployeeCount;
    }, null, TimeSpan.Zero, TimeSpan.FromSeconds(1));
}
```

Как видите, нужно совсем немного усилий, чтобы определить собственные счетчики производительности, и они могут предоставлять весьма важную информацию. Корреляции системного и пользовательского счетчиков производительности часто бывает достаточно, чтобы понять причины, вызывающие проблемы с производительностью или настройками.

> **Примечание.** *Системный монитор можно использовать и для сбора другой информации, не имеющей отношения к счетчикам производительности. Например, его можно применять для сбора информации о системных настройках – значений ключей из реестра, свойств объектов WMI и даже содержимого файлов на диске. Поддерживается также возможность захватывать данные, поставляемые провайдерами механизма ETW (о котором рассказывается далее) для последующего анализа. Используя XML-шаблоны, администраторы могут создавать группы сборщиков данных на других компьютерах и генерировать отчеты, выполнив всего несколько простых операций по настройке.*

Счетчики производительности позволяют получить массу интересной информации о производительности, но они не могут использоваться как высокопроизводительная инфраструктура мониторинга и журналирования. Не существует системных компонентов, способных обновлять счетчики производительности чаще нескольких раз в секунду, а сама программа Performance Monitor (Системны монитор) не позволяет читать значения счетчиков чаще, чем один раз в секунду. Если для анализа потребуется выполнять замеры каких-либо характеристик тысячи раз в секунду, счетчики производительности окажутся непригодными для этого. Теперь обратим внимание на механизм трассировки событий для Windows (Event Tracing for Windows,

ETW), специально спроектированный для высокоскоростного сбора данных самых разных типов (не только числовых).

Механизм трассировки событий для Windows

Механизм трассировки событий для Windows (Event Tracing for Windows, ETW) – это высокопроизводительный фреймворк регистрации событий, встроенный в Windows. По аналогии со счетчиками производительности, многие компоненты системы и инфраструктура поддержки приложений, включая ядро Windows и CLR, определяют *механизмы отправки событий* – информации о внутреннем состоянии компонентов. В отличие от счетчиков производительности, которые всегда активны, механизм ETW можно включать и выключать во время выполнения, чтобы накладные расходы на сбор и отправку информации оказывали влияние на производительность, только когда это действительно необходимо.

Одним из богатейших источников информации является *провайдер ядра* (kernel provider), который генерирует события в моменты запуска процессов и потоков, загрузки DLL, распределения блоков памяти, сетевых операций ввода/вывода и при выполнении трассировки стека. В табл. 2.1 приводится перечень некоторых наиболее интересных событий, сообщаемых ETW-провайдерами ядра и CLR. Механизм ETW можно использовать для исследования общего поведения системы, например, чтобы выяснить, какой из процессов потребляет большую часть вычислительной мощности CPU, проанализировать узкие места в операциях ввода/вывода, получить статистику, касающуюся работы сборщика мусора и использования памяти управляемыми процессами, и во многих других случаях, обсуждаемых далее в этом разделе.

События ETW несут в себе точное время их возникновения, могут содержать дополнительную пользовательскую информацию, а также состояние стека на момент их появления. Информация о состоянии стека может использоваться для выявления источников различных проблем. Например, провайдер CLR может посылать события в начале и в конце каждого цикла сборки мусора. Эти события в комплексе с информацией о состоянии стека вызовов можно использовать для выявления частей программы чаще других вызывающих сборку мусора. (За дополнительной информацией о сборке мусора и событиях, ее вызывающих, обращайтесь к главе 4.)

Таблица 2.1. Неполный список событий ETW в ядре Windows и CLR

Провайдер	Флаг/ключевое слово	Описание	События
Ядро	PROC_THREAD	Запуск и завершение процессов и потоков	–
Ядро	LOADER	Загрузка и выгрузка образов (библиотек DLL, драйверов, выполняемых файлов)	–
Ядро	SYSCALL	Системные вызовы	–
Ядро	DISK_IO	Дисковые операции чтения и записи (включая позиционирование головок)	–
Ядро	HARD_FAULTS	Ошибки обращения к страницам диска (которые были вытеснены из кеша в оперативной памяти)	–
Ядро	PROFILE	Дискретное событие – сохранение информации о состоянии стека для всех процессоров выполняется через каждую 1 мсек	–
CLR	GCKeyword	Статистика и информация о работе механизма сборки мусора	Запуск сборки, конец сборки, запуск процедур завершения, выделение блока памяти ~100 Кбайт
CLR	ContentionKeyword	Конфликт между потоками выполнения при попытке приобрести разделяемую блокировку	Начало конфликта (поток переведен в режим ожидания), конец конфликта
CLR	JITTracingKeyword	Информация о состоянии динамического компилятора (Just in Time, JIT)	Успешная попытка встраивания метода, неудачная попытка встраивания метода
CLR	ExceptionKeyword	Возбужденное исключение	–

Для доступа к этой детализированной информации требуется специализированный инструмент и приложение, способное читать события ETW и выполнять простейший анализ. На момент написания этих строк существовало два инструмента, способных решать обе задачи: Windows Performance Toolkit (WPT, также известный как XPerf), распространяемый в составе Windows SDK, и PerfMonitor (не путайте с Windows Performance Monitor!) – открытый проект, разрабатываемый командой CLR в Microsoft.

Windows Performance Toolkit (WPT)

Windows Performance Toolkit (WPT) – это комплект утилит для управления сеансами ETW, сохранения событий ETW в файлах журналов и их обработки для последующего отображения на экране. Может генерировать графики и диаграммы событий ETW, сводные таблицы, включающие информацию о состоянии стека, и файлы CSV для автоматизированной обработки. Чтобы установить WPT, загрузите дистрибутив Windows SDK на странице http://msdn.microsoft.com/en-us/performance/cc752957.aspx, запустите мастер установки и выберите только **Common Utilities → Windows Performance Toolkit** (Общие утилиты → Windows Performance Toolkit). После установки перейдите в подкаталог *Redist\Windows Performance Toolkit* в каталоге установки SDK и запустите мастер установки для своей аппаратной архитектуры (*Xperf_x86.msi* – для 32-разрядных систем, *Xperf_x64.msi* – для 64-разрядных систем).

> **Примечание.** *В 64-разрядной версии Windows для поддержки возможности трассировки стека необходимо изменить настройки в реестре, запрещающие выгрузку страниц с кодом из оперативной памяти в файл подкачки (для самого ядра Windows и для всех драйверов). Это может увеличить потребление оперативной памяти системой на несколько мегабайт. Чтобы изменить настройки, найдите в реестре ключ* `HKLM\System\CurrentControlSet\Control\Session Manager\Memory Management`*, установите параметр* `DisablePagingExecutive` *типа* `DWORD` *в значение* `0x1` *и перезагрузите систему.*

Для перехвата и анализа событий ETW используются инструменты *XPerf.exe* и *XPerfView.exe*. Оба должны запускаться с привилегиями администратора. Утилита *XPerf.exe* имеет несколько ключей командной строки, с помощью которых можно указать, какие провайдеры должны включаться, размеры используемых буферов, имя файла для сохранения информации о событиях и множество других параметров. Утилита *XPerfView.exe* анализирует исходную информацию и генерирует графические отчеты на основе информации в файле журнала.

Вместе с событиями может сохраняться также информация о состоянии стека вызовов, что часто помогает выявить дополнительные грани проблем производительности. Однако, чтобы получить информацию о состоянии стека совсем необязательно включать прием событий от какого-то определенного провайдера – флаг `SysProfile` позволяет получать эту информацию от всех процессоров с интервалом 1 мсек. Это упрощенный способ понять суть событий, протекающих в системе на уровне методов. (Мы еще вернемся к этому режиму, далее в этой главе, когда будем знакомится с дискретными профилировщиками.)

Захват и анализ событий ядра с помощью XPerf

В этом разделе предлагается выполнить трассировку событий ядра с помощью *XPerf.exe* и проанализировать полученные результаты с помощью *XPerfView.exe*. Данный эксперимент планировался для проведения в версии Windows Vista или выше. (Для его проведения требуется также настроить две системные переменные окружения: щелкните правой кнопкой мыши на ярлыке **Computer** (Компьютер), выберите пункт контекстного меню **Properties** (Свойства), щелкните на пункте **Advanced system settings** (Дополнительные параметры системы) в панели слева и в открывшемся диалоге – на кнопке **Environment Variables** (Переменные среды) внизу.)

1. Создайте системную переменную окружения `_NT_SYMBOL_PATH` со значением, включающим путь к общедоступному серверу символов и локальному кешу символов, например: `srv*C:\Temp\Symbols*http://msdl.microsoft.com/download/symbols`.
2. Создайте системную переменную окружения `_NT_SYMCACHE_PATH` со значением, включающим путь к локальному каталогу на диске – это должен быть другой каталог, отличный от того, что был указан в качестве локального кеша символов в предыдущем пункте.
3. Запустите с правами администратора программу **Command Prompt** (Командная строка) и перейдите в каталог установки WPT (например, *C:\Program Files\Windows Kits\8.0\Windows Performance Toolkit*).
4. Запустите прием событий из группы Base ядра, содержащие флаги `PROC_THREAD`, `LOADER`, `DISK_IO`, `HARD_FAULTS`, `PROFILE`, `MEMINFO` и `MEMINFO_WS` (см. табл. 2.1). Для этого выполните команду: `xperf -on Base`.
5. Сымитируйте некоторую активность в системе: запустите несколько приложений, попереключайтесь между окнами, попробуйте открыть какие-нибудь файлы – хотя бы несколько секунд. (Все это будет приводить к созданию отслеживаемых событий.)
6. Остановите прием событий и сохраните результаты в файл, выполнив команду: `xperf -d KernelTrace.etl`.
7. Запустите инструмент анализа, выполнив команду: `xperfview KernelTrace.etl`.

8. В появившемся окне вы увидите несколько графиков, по одному для каждого события ETW. Выбор графиков для отображения выполняется в панели слева. Обычно флажки, соответствующие графикам нагрузки на процессоры, находятся в самом верху, а ниже – флажки выбора графиков дисковых операций ввода/вывода, использования памяти и других статистик.
9. Щелкните правой кнопкой мыши на графике нагрузки на процессор и выберите пункт контекстного меню **Load Symbols** (Загрузить символы). Щелкните правой кнопкой мыши на графике еще раз и выберите пункт контекстного меню **Simple Summary Table** (Простая сводная таблица). В результате должна появиться таблица со списком методов во всех процессах, проявлявших активность в процессе сбора информации. (Загрузка символов с сервера Microsoft в первый раз может занять продолжительное время.)

Инструмент WPT способен на большее, чем было показано в этом эксперименте. Вам следует заняться самостоятельными исследованиями пользовательского интерфейса и попробовать принять и проанализировать другие группы событий ядра или даже события от собственных провайдеров ETW. (Создание собственных провайдеров рассматривается далее в этой главе.)

Инструмент WPT может пригодиться в самых ситуациях, поможет вникнуть в поведение системы и отдельных процессов. Ниже представлено несколько скриншотов и описаний примеров подобных ситуаций:

- WPT может перехватывать все события дисковых операций ввода/вывода в системе и выводить информацию с привязкой к карте физического диска. Это дает возможность выявить наиболее дорогостоящие операции ввода/вывода, в частности операции, требующие значительных перемещений головок жесткого диска. (См. рис. 2.2.)
- WPT может предоставить информацию о состоянии стеков вызовов для всех процессоров в системе. Он группирует стеки вызовов по процессам, модулям и функциям, что позволяет визуально оценить, где система (или конкретное приложение) проводит больше всего времени. Обратите внимание, что управляемые кадры стеков не поддерживаются – к этой проблеме мы еще вернемся ниже, когда будем знакомиться с инструментом PerfMonitor. (См. рис. 2.3.)
- WPT может отображать сводные графики событий разных типов, чтобы проще было выявить корреляцию, например, между операциями ввода/вывода, использованием памяти, нагрузкой на процессор и другими характеристиками. (См. рис. 2.4.)

- WPT может отображать сводную информацию о состоянии стеков вызовов (когда утилита приема событий запускалась с ключом `-stackwalk`) – это дает возможность получить полную информацию о стеках вызовов на момент создания определенны событий. (См. рис. 2.5.)

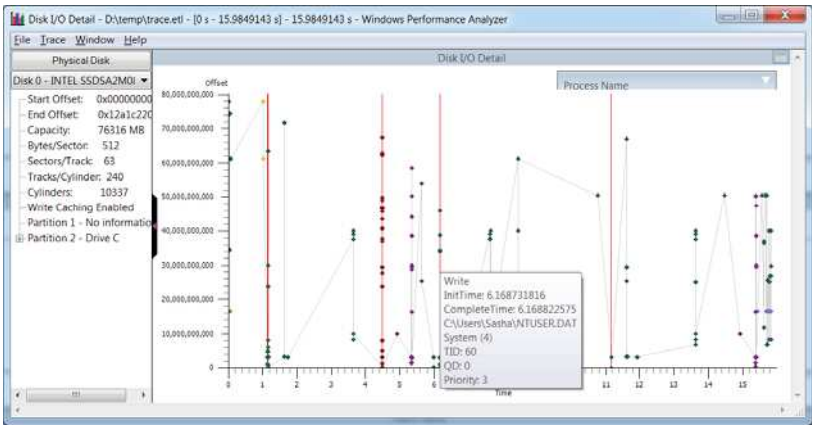

Рис. 2.2. Дисковые операции ввода/вывода, нанесенные на карту физического диска. Информация об операциях ввода/вывода и дополнительных деталях предоставляется в виде всплывающих подсказок.

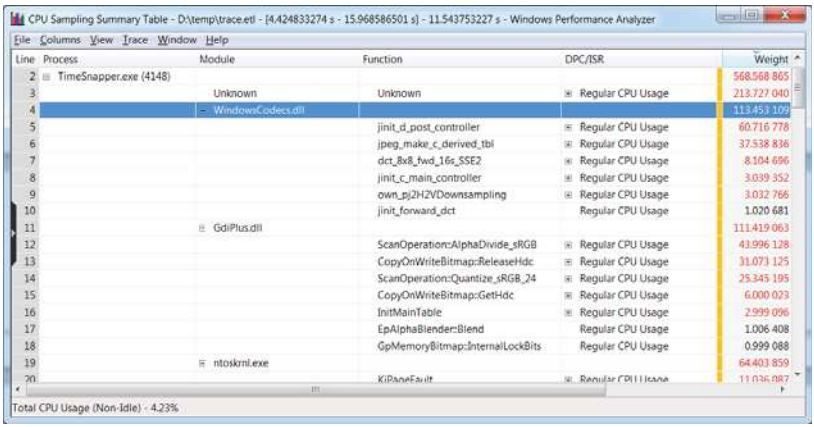

Рис. 2.3. Детальная информация о кадрах стека вызовов для единственного процесса (TimeSnapper.exe). Колонка Weight (Вес) показывает (примерно), как долго продолжалось выполнение в этом кадре.

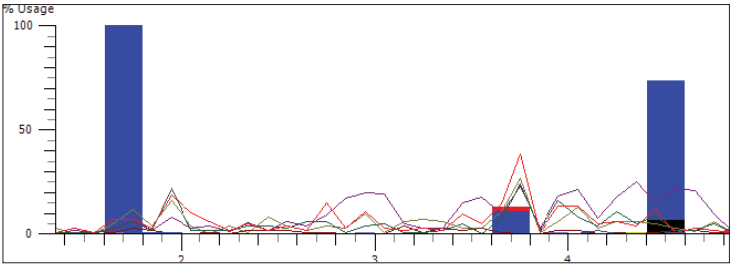

Рис. 2.4. Сводная диаграмма нагрузки на процессор (линии, каждая из которых соответствует отдельному процессору) и дисковых операций ввода/вывода (столбики). Никакой явной корреляции между нагрузкой на процессор и интенсивностью операций ввода/вывода здесь не просматривается.

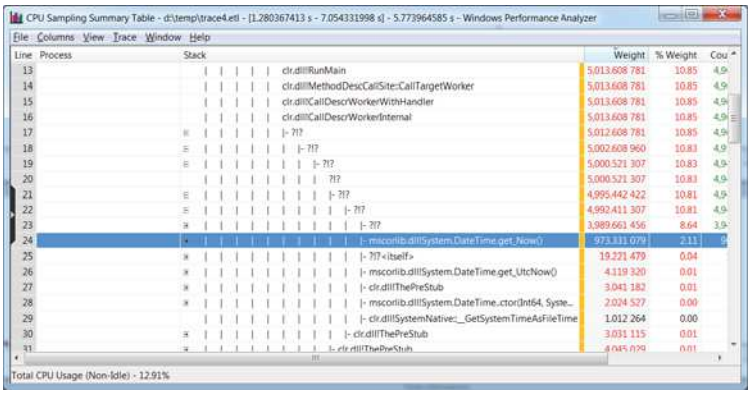

Рис. 2.5. Сводная информация о состоянии стеков вызовов. Обратите внимание, что управляемые кадры отображаются лишь частично – кадры «?!?» не могут быть идентифицированы инструментом. Кадры стека, имеющие отношение к mscorlib.dll (например, System.DateTime.get_Now()) были успешно идентифицированы, потому что код компилировался с помощью NGen а не динамическим JIT-компилятором во время выполнения.

Примечание. *Последняя версия Windows SDK (версия 8.0) включает два новых инструмента: Windows Performance Recorder (wpr.exe) и Windows Performance Analyzer (wpa.exe), созданные с целью постепенно заменить инструменты XPerf и XPerfView, описанные выше. Например, команда* `wpr -start CPU` *является примерным эквивалентом команды* `xperf -on Diag`, *а команда* `wpr -stop reportfile` *– примерным эквивалентом команды* `xperf -d reportfile`. *Пользовательский интерфейс инструмента WPA несколько отличается, но предоставляет практически те*

же возможности, что и XPerfView. За дополнительной информацией по новым инструментам обращайтесь по адресу: http://msdn.microsoft.com/en-us/library/hh162962.aspx.

Инструмент XPerfView обладает достаточно широкими возможностями в отображении данных, поставляемых провайдером ядра, в виде удобных графиков и таблиц, но он не обладает столь же широкой поддержкой других провайдеров. Например, мы можем обеспечить трассировку событий от провайдера CLR ETW, но XPerfView не будет создавать графики для различных событий – нам придется самим разбираться в исходных данных трассировки, опираясь на список ключевых слов и событий, перечисленных в документации (полный перечень ключевых слов и событий провайдера CLR ETW можно найти в документации на сайте MSDN: http://msdn.microsoft.com/ru-ru/library/ff357720.aspx).

Если запустить XPerf с провайдером CLR ETW (`e13c0d23-ccbc-4e12-931b-d9cc2eee27e4`) для сбора событий с ключевым словом `GCKeyword` (`0x00000001`) и уровнем детализации `Verbose` (`0x5`), эта утилита послушно будет перехватывать все события, генерируемые провайдером. Сохранив всю полученную информацию в файл CSV или открыв ее с помощью XPerfView, мы сможем (хотя и с трудом) идентифицировать события механизма сборки мусора в нашем приложении. На рис. 2.6 показан пример отчета, созданного утилитой XPerfView, где время между событиями `GC /Start` и `GC /Stop` соответствует протяженности одного цикла работы механизма сборки мусора.

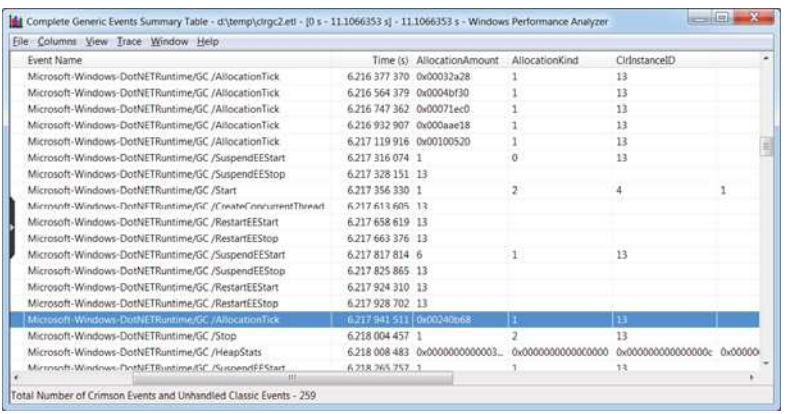

Рис. 2.6. Отчет о событиях механизма сборки мусора в CLR. Выделенная строка – это событие GCAllocationTick_V1, генерируемое после распределения каждых 100 Кбайт памяти.

К счастью, этот недостаток был замечен разработчиками базовой библиотеки классов (Base Class Library, BCL) в Microsoft, и они создали открытую библиотеку и инструмент PerfMonitor для анализа трассировочной информации от CLR ETW. Мы рассмотрим этот инструмент далее.

PerfMonitor

PerfMonitor.exe – это открытый инструмент командной строки, созданный командой разработчиков BCL в Microsoft, и доступный на сайте CodePlex. На момент написания этих строк самой последней была версия PerfMonitor 1.5: http://bcl.codeplex.com/releases/. Главное преимущество PerfMonitor перед WPT заключается в полной поддержке событий CLR и способности выводить информацию о них не только в табличном виде. PerfMonitor способен анализировать события, генерируемые сборщиком мусора и JIT-компилятором, выполнять трассировку управляемого стека и определять нагрузку на процессор, оказываемую различными частями приложения.

Для опытных пользователей в состав PerfMonitor включена библиотека с именем TraceEvent, обеспечивающая программный доступ к событиям CLR ETW и позволяющая автоматизировать анализ событий. Библиотеку TraceEvent можно использовать в собственных приложениях мониторинга для автоматического исследования и регистрации событий, протекающих в ходе эксплуатации системы.

Инструмент PerfMonitor можно использовать для сбора событий ядра или даже событий собственного провайдера ETW (запуская его с ключами `/KernelEvents` и `/Provider`), но обычно он применяется для анализа поведения управляемых приложений с использованием встроенных провайдеров CLR. С помощью ключа `runAnalyze` ему можно указать любое приложение для трассировки, по завершении которого PerfMonitor сгенерирует подробный отчет в формате HTML и откроет его в браузере по умолчанию. (Чтобы создать отчеты, похожие на те, что представлены в этом разделе, вам следует ознакомиться с руководством для PerfMonitor – хотя бы с разделом «Quick Start». Для этого выполните команду `PerfMonitor usersguide`.)

Когда выполняется запуск PerfMonitor с целью выполнить приложение и сгенерировать отчет, он выводит в окне терминала следующие строки. (В процессе чтения этого раздела вы можете сами поэкспериментировать с инструментом, запуская с его помощью приложение *JackCompiler.exe* из папки с примерами к этой главе.)

```
C:\PerfMonitor > perfmonitor runAnalyze JackCompiler.exe

Starting kernel tracing. Output file: PerfMonitorOutput.kernel.etl
Starting user model tracing. Output file: PerfMonitorOutput.etl
Starting at 4/7/2012 12:33:40 PM
Current Directory C:\PerfMonitor
Executing: JackCompiler.exe {

} Stopping at 4/7/2012 12:33:42 PM = 1.724 sec
Stopping tracing for sessions 'NT Kernel Logger' and 'PerfMonitorSession'.
Analyzing data in C:\PerfMonitor\PerfMonitorOutput.etlx
GC Time HTML Report in C:\PerfMonitor\PerfMonitorOutput.GCTime.html
JIT Time HTML Report in C:\PerfMonitor\PerfMonitorOutput.jitTime.html
Filtering to process JackCompiler (1372). Started at 1372.000 msec.
Filtering to Time region [0.000, 1391.346] msec
CPU Time HTML report in C:\PerfMonitor\PerfMonitorOutput.cpuTime.html
Filtering to process JackCompiler (1372). Started at 1372.000 msec.
Perf Analysis HTML report in C:\PerfMonitor\PerfMonitorOutput.
analyze.html
PerfMonitor processing time: 7.172 secs.
```

Различные HTML-файлы отчетов, сгенерированные инструментом PerfMonitor, содержат уже обработанную информацию, но вы всегда можете открыть исходные ETL-файлы с помощью XPerfView или любого другого инструмента, способного читать двоичные файлы журналов с событиями ETW. Сводный отчет для примера выше включает следующую информацию (при выполнении эксперимента на вашем компьютере фактические значения могут отличаться):

- статистика использования CPU – на выполнение приложения было потрачено 917 мсек процессорного времени, а средняя нагрузка составила 56.6%. Остальное время было потрачено на ожидание каких-то событий;
- статистика сборщика мусора – общее время работы сборщика мусора составило 20 мсек, максимальный размер кучи сборщика мусора составил 4.5 Мбайт, максимальная скорость выделения памяти составила 1496.1 Мбайт/сек, а средняя пауза между циклами сборки мусора составила 0.1 мсек.
- статистика JIT-компилятора – за время выполнения JIT-компилятором было скомпилировано 159 методов с общим объемом машинного кода 30493 байт.

Погружаясь на более глубокие уровни отчетов можно получить огромное количество полезной информации. В число детальных отчетов использования CPU входят: отчет о методах, на выполнение которых было потрачено больше всего процессорного времени (*вос-*

ходящий анализ), отчет о деревьях вызовов, где было потрачено больше всего процессорного времени (*нисходящий анализ*), и отдельные отчеты «вызывающий-вызываемый» для каждого метода. Чтобы предотвратить разбухание отчетов, из них исключается информация со значениями ниже определенного порога (1% – для восходящего анализа, и 5% – для нисходящего анализа). На рис. 2.7 представлен пример отчета восходящего анализа, где видно, что тремя наиболее активно используемыми методами являются `System.String.Concat`, `JackCompiler.Tokenizer.Advance` и `System.Linq.Enumerable.Contains`. На рис. 2.8 представлен пример отчета нисходящего анализа, где видно, что 84.2% процессорного времени было потрачено методом `JackCompiler.Parser.Parse`, который вызывает метод `ParseClass`, `ParseSubDecls`, `ParseSubDecl`, `ParseSubBody` и так далее.

Name	Exc %	Exc MSec	Inc %	Inc MSec	CPU Utilization	First	Last
mscorlib!System.String.Concat(String,String,String)	15.0	138	15.0	138			
JackCompiler.Compiler!JackCompiler.Tokenizer.Advance()	12.5	115	19.3	177			
System.Core!System.Linq.Enumerable.Contains(IEnumerable`1<!!0>,!!0)	8.1	74	8.1	74			
JackCompiler.Compiler!JackCompiler.Parser.ParseTerm()	7.0	64	25.8	237			
JackCompiler.Compiler!JackCompiler.Parser.ParseAddExpression()	4.4	40	36.8	337			
kernel32!?	3.7	34	3.7	34			
mscorlib!System.IO.TextWriter.WriteLine(String,Object)	3.6	33	9.9	91			
JackCompiler.Compiler!JackCompiler.Tokenizer.EatWhile(Predicate`1<wchar>)	2.9	27	3.5	32			
JackCompiler.Compiler!JackCompiler.Parser.ParseMulExpression()	2.5	23	33.5	307		535.779	1388.363
JackCompiler.Compiler!JackCompiler.Parser.ParseExpression()	2.2	20	43.8	402		534.779	1388.363
JackCompiler.Compiler!JackCompiler.Parser.IsNextTokenMulOp()	2.2	20	5.3	49		582.152	1375.159
JackCompiler.Compiler!JackCompiler.Parser.ParseLetStatement()	2.1	19	39.3	360		533.779	1388.363
ntdll!?	2.1	19	98.3	901		297.866	1390.272
JackCompiler.Compiler!JackCompiler.Parser.ParseSubCall(Token)	2.1	19	21.6	198		547.779	1389.374
mscorwks!?	2.0	18	2.0	18		399.791	519.773
JackCompiler.Compiler!JackCompiler.CCodeGenerator.Assignment(Token,bool)	1.9	17	5.6	51		555.843	1365.522
JackCompiler.Compiler!JackCompiler.Parser.ParseSubBody()	1.9	17	78.5	720		529.779	1389.374
JackCompiler.Compiler!JackCompiler.Parser.ParseRelationalExpression()	1.9	17	40.9	375		534.779	1388.363
System.Core!System.Linq.Enumerable.Contains(IEnumerable`1<!!0>,!!0)	1.5	14	1.5	14		547.779	1348.705
JackCompiler.Compiler!JackCompiler.Tokenizer.Next()	1.2	11	19.3	177		547.779	1389.374
JackCompiler.Compiler!JackCompiler.Tokenizer.EatWhitespace()	1.2	11	1.7	16		515.773	1382.273

CPU Utilization over time.
Total time is broken into 32 buckets and each digit represents its CPU
_ = no CPU used
0 = 0-10% CPU use
1 = 10-20% use
...
9 = 90-100% use
A = 100-110% use.
...
Z = 340-350% use.
* = greater than 350% use.

Рис. 2.7. Восходящий анализ. Колонка «Exc %» – оценка процессорного времени, затраченного только на выполнение данного метода; колонка «Inc %» – оценка процессорного времени, затраченного на выполнение данного метода и всех других методов, которые он вызывает (его поддерево вызовов).

Детальный отчет для механизма сборки мусора содержит таблицу со статистической информацией о его работе (счетчики, времена) для каждого цикла, а также информация об отдельных событиях, включая продолжительность пауз, объем освобожденной памяти и многое другое. Некоторые из этих сведений пригодятся нам в главе 4, при обсуждении внутреннего устройства сборщика мусора и его влияния на производительность. На рис. 2.9 показано несколько строк из отчета об отдельных событиях сборщика мусора.

Name	Inc %	Inc MSec	Exc %	Exc MSec	CPU Utilization
ROOT	100.0	917	0.0	0	0311599A9999AAA9A991_
+Process JackCompiler (1372)	100.0	917	0.0	0	0311599A9999AAA9A991_
+Thread (4636)	100.0	917	0.2	2	0311599A9999AAA9A991_
+ntdll!?	98.3	901	2.1	19	0211599A9999AA99A991_
|+JackCompiler!JackCompiler.CompilerDriver.Main(String[])	95.3	874	0.2	2	1599A9999AA99A991_
||+JackCompiler!JackCompiler.CompilerDriver.DriveCompilerWithCCodeGenerator(String[])	95.1	872	0.5	5	0599A99999A99A991_
|| +JackCompiler.Compiler!JackCompiler.Parser.Parse()	84.2	772	0.0	0	2989898988998991_
|| |+JackCompiler.Compiler!JackCompiler.Parser.ParseClass()	84.2	772	0.9	8	2989898988998991_
|| | +JackCompiler.Compiler!JackCompiler.Parser.ParseSubDecls()	82.0	752	0.1	1	2979898888988881_
|| | |+JackCompiler.Compiler!JackCompiler.Parser.ParseSubDecl()	81.9	751	0.5	5	2979898888988881_
|| | | +JackCompiler.Compiler!JackCompiler.Parser.ParseSubBody()	78.5	720	1.9	17	1878798878888881_
|| | | |+JackCompiler.Compiler!JackCompiler.Parser.ParseStatements()	75.4	691	0.3	3	1878788878888881_
|| | | ||+JackCompiler.Compiler!JackCompiler.Parser.ParseStatement()	75.0	688	0.5	5	1877788878888881_
|| | | || +JackCompiler.Compiler!JackCompiler.Parser.ParseLetStatement()	30.2	277	1.2	11	0433433333222320_
|| | | || |+JackCompiler.Compiler!JackCompiler.Parser.ParseExpression()	20.3	186	0.0	0	0223212222111210_
|| | | || ||+JackCompiler.Compiler!JackCompiler.Parser.ParseRelationalExpression()	19.5	179	0.3	3	0222212212111210_
|| | | || |||+JackCompiler.Compiler!JackCompiler.Parser.ParseAddExpression()	17.7	162	0.1	1	0222211212111110_
|| | | || ||||+JackCompiler.Compiler!JackCompiler.Parser.ParseMulExpression()	16.8	154	0.5	5	0222211211111110_
|| | | || |||||+JackCompiler.Compiler!JackCompiler.Parser.ParseTerm()	12.9	118	0.8	7	212111111100100_
|| | | || +JackCompiler.Compiler!JackCompiler.Parser.ParseDoStatement()	22.0	202	0.1	1	1221222222223220_
|| | | || |+JackCompiler.Compiler!JackCompiler.Parser.ParseSubCall(Token)	17.3	159	0.1	1	0110211122222210_
|| | | || ||+JackCompiler.Compiler!JackCompiler.Parser.ParseExpressionList()	8.1	74	0.0	0	0101100001111O_
|| | | || |||+JackCompiler.Compiler!JackCompiler.Parser.ParseExpression()	8.1	74	0.3	3	0101100001111O_

Рис. 2.8. Нисходящий анализ.

GC Events by Time

All times are in msec. Start time is msec from trace start.

Start Time	GC Num	Gen	Pause	Alloc Rate MB/sec	Alloc MB	MSec GC/ Alloc MB	MSec GC/ Kept MB	Before MB	After MB	Ratio Before/After	Reclaimed	Suspend Time	Type	Reason
551.053	1	0	0.20	7.53	4.15	0.046	2.56	4.15	0.07	55.82	4.07	0.01	NonConcurrentGC	AllocSmall
554.348	2	0	0.09	1341.22	4.15	0.020	0.84	4.23	0.10	42.74	4.13	0.01	NonConcurrentGC	AllocSmall
557.265	3	0	0.08	1465.75	4.15	0.018	0.59	4.25	0.13	33.15	4.12	0.01	NonConcurrentGC	AllocSmall
560.292	4	0	0.08	1405.49	4.15	0.019	0.49	4.27	0.16	27.49	4.12	0.01	NonConcurrentGC	AllocSmall
563.323	5	0	0.09	1406.78	4.15	0.020	0.46	4.31	0.18	23.94	4.13	0.01	NonConcurrentGC	AllocSmall
566.281	6	0	0.08	1449.91	4.16	0.017	0.33	4.34	0.21	20.79	4.14	0.01	NonConcurrentGC	AllocSmall

Рис. 2.9. Отдельные события сборщика мусора, содержащие информацию об объеме освобожденной памяти, продолжительности пауз, типе событий и другие сведения.

Наконец, отчет о работе JIT-компилятора содержит информацию о времени, затраченном на компиляцию каждого из методов приложения, а также точное время, когда они были скомпилированы. Эта информация может пригодиться, чтобы выяснить, можно ли увеличить скорость запуска приложения – если на этапе запуска приложения значительное время тратится на работу JIT-компилятора, предварительная компиляция приложения (с помощью NGen) может оказаться существенной оптимизацией. Применение NGen и другие стратегии уменьшения времени запуска приложения мы обсудим в главе 10.

Совет. *В ходе сбора информации от нескольких провайдеров ETW могут получаться очень большие файлы журналов. Например, в режиме по умолчанию PerfMonitor генерирует примерно 5 Мбайт данных в секунду. Если оставить инструмент работать на несколько дней, он наверняка исчерпает дисковое пространство даже на очень большом жестком диске. К счастью,*

оба инструмента, XPerf и PerfMonitor, поддерживают циклический режим журналирования, когда в журнале сохраняется только последние N Мбайт данных. В PerfMonitor максимальный размер файла журнала можно указать (в мегабайтах) с помощью ключа `/Circular`*, при этом все старые файлы будут автоматически удаляться при превышении указанного порогового значения.*

PerfMonitor – очень мощный инструмент, однако богатство параметров командной строки и тот факт, что он генерирует отчеты в формате HTML, несколько осложняют его использование. Следующий инструмент, который мы рассмотрим, обладает очень похожей функциональностью и может использоваться в тех же ситуациях, при этом он имеет более дружественный интерфейс и упрощает некоторые исследования проблем производительности.

PerfView

PerfView – бесплатный инструмент, разрабатываемый в корпорации Microsoft, объединяющий в себе функции сбора информации от провайдеров ETW и ее анализа, по аналогии с PerfMonitor, а также средства анализа динамической памяти, которые будут обсуждаться ниже, во время знакомства с такими инструментами, как CLR Profiler и ANTS Memory Profiler. Загрузить PerfView можно по адресу: http://www.microsoft.com/download/en/details.aspx?id=28567. Обратите внимание, что инструмент PerfView должен запускаться с привилегиями администратора, потому что требует доступа к инфраструктуре ETW.

Чтобы проанализировать информацию, полученную при выполнении определенного процесса, выберите пункт меню **Collect → Run** (Собрать → Выполнить) в PerfView (на рис. 2.10 показано главное окно программы). Для нужд анализа распределения динамической памяти, который мы вскоре выполним, в пакет с примерами к этой главе включено приложение *MemoryLeak.exe*. Это приложение будет запускаться с помощью инструмента PerfView, который сгенерирует отчеты со всей информацией, доступной в PerfMonitor, и не только, включая:

- простой список событий ETW, полученных от разных провайдеров (например, с информацией о конфликтах в CLR, дисковых операциях ввода/вывода, TCP-пакетах и ошибках чтения страниц);
- сгруппированные участки стека вызовов, где приложение провело больше всего времени, с возможностью настройки фильтров и пороговых значений;

- участки стека вызовов, соответствующие операциям загрузки образов (сборок), дисковым операциям ввода/вывода и операциям выделения памяти (для каждых ~100 Кбайт);
- статистика сборщика мусора, включая продолжительность каждого цикла сборки мусора и объем освобожденной памяти.

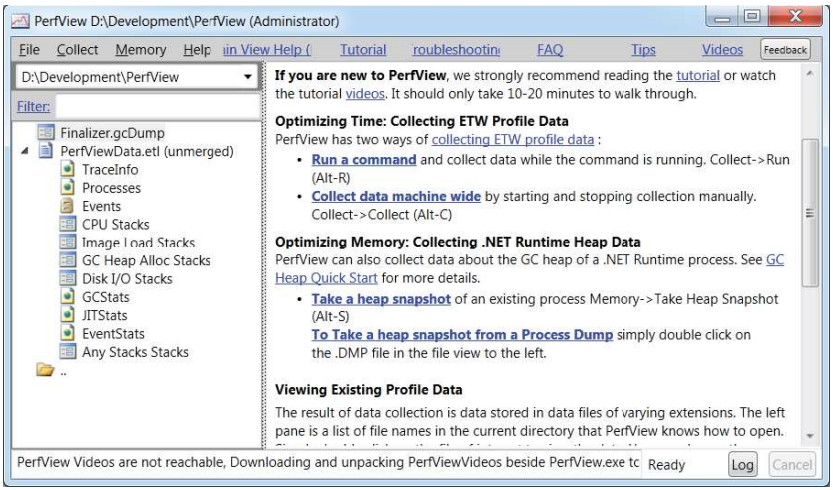

Рис. 2.10. Главное окно PerfView. В панели со списком файлов (слева) можно видеть дамп динамической памяти и файл с информацией ETW. Ссылки в центральной области окна ведут к разным командам, поддерживаемым инструментом.

Кроме того, с помощью PerfView можно сохранять срезы динамической памяти текущего выполняющегося процесса или импортировать их из файла. После импортирования, с помощью PerfView можно найти типы объектов, занимающих наибольший объем памяти и выявить цепочки ссылок, ответственные за удержание этих объектов в памяти. На рис. 2.11 изображен пример анализа ссылок на объекты класса `Schedule`, который занимают в динамической памяти 31 Мбайт. PerfView благополучно обнаружил, что ссылки на объекты `Schedule` хранятся в экземплярах класса `Employee`, а экземпляры `Employee` удерживаются в памяти очередью объектов, готовых к завершению (f-reachable queue), которая обсуждается в главе 4.

Когда ниже в этой главе будут обсуждаться профилировщики памяти, мы увидим, что возможности визуализации в PerfView все еще отстают от возможностей коммерческих инструментов. Однако

PerfView все еще остается весьма полезным и бесплатным инструментом, способным ускорить анализ проблем производительности. Более подробную информацию о нем можно получить из встроенного руководства, отображаемого в главном окне сразу после запуска, а также из видеороликов, записанных командой разработчиков BCL и демонстрирующих основные возможности инструмента.

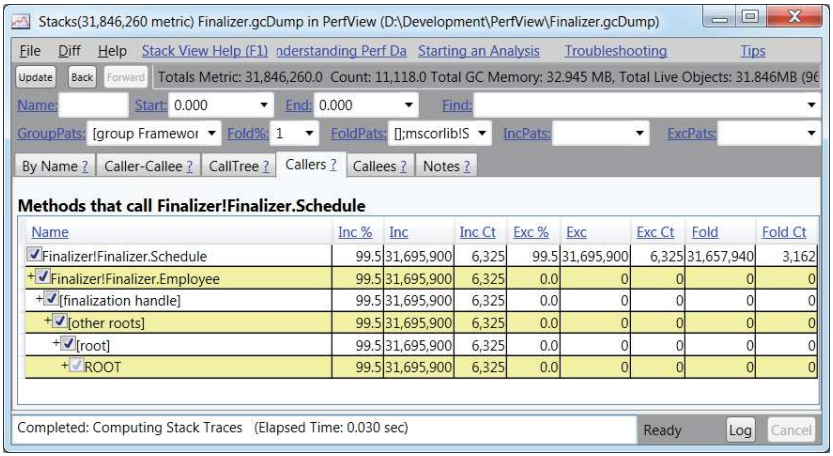

Рис. 2.11. Цепочка ссылок на объекты класса Schedule, занимающих 99.5% памяти приложения в сохраненном срезе динамической памяти.

Собственные провайдеры ETW

Как и при использовании счетчиков производительности, у вас может появиться желание внедрить в свое приложение поддержку возможности сбора и передачи информации, предлагаемой инфраструктурой ETW. До появления версии .NET 4.5, экспортирование информации ETW из управляемых приложений было весьма непростым делом. Необходимо было учитывать массу тонкостей, связанных с определением манифеста провайдера ETW для вашего приложения, созданием его во время выполнения и регистрацией событий. С выходом .NET 4.5, создание собственных провайдеров ETW стало легче некуда. Для этого достаточно унаследовать класс `System.Diagnostics.Tracing.EventSource` и вызывать метод `WriteEvent` базового класса для вывода событий ETW. Все рутинные операции по регистрации провайдера в системе и форматированию информации в событиях выполняются автоматически.

Ниже приводится пример реализации провайдера ETW в управляемом приложении (полные исходные тексты программы доступны в папке с примерами к этой главе и вы можете попробовать запустить ее с помощью PerfMonitor):

```
public class CustomEventSource : EventSource {
    public class Keywords {
        public const EventKeywords Loop = (EventKeywords)1;
        public const EventKeywords Method = (EventKeywords)2;
    }

    [Event(1, Level = EventLevel.Verbose, Keywords = Keywords.Loop,
           Message = "Loop {0} iteration {1}")]
    public void LoopIteration(string loopTitle, int iteration) {
        WriteEvent(1, loopTitle, iteration);
    }

    [Event(2, Level = EventLevel.Informational, Keywords = Keywords.Loop,
           Message = "Loop {0} done")]
    public void LoopDone(string loopTitle) {
        WriteEvent(2, loopTitle);
    }

    [Event(3, Level = EventLevel.Informational, Keywords = Keywords.Method,
           Message = "Method {0} done")]
    public void MethodDone([CallerMemberName] string methodName = null) {
        WriteEvent(3, methodName);
    }
}

class Program {
    static void Main(string[] args) {
        CustomEventSource log = new CustomEventSource();
        for (int i = 0; i < 10; ++i) {
            Thread.Sleep(50);
            log.LoopIteration("MainLoop", i);
        }
        log.LoopDone("MainLoop");
        Thread.Sleep(100);
        log.MethodDone();
    }
}
```

Для получения информации из такого приложения можно использовать инструмент PerfMonitor. Запустить с его помощью приложение, произвести сбор событий от провайдера ETW и сгенерировать отчет обо всех событиях, отправленных приложением. Например:

```
C:\PerfMonitor > perfmonitor monitorDump Ch02.exe

Starting kernel tracing. Output file: PerfMonitorOutput.kernel.etl
```

```
Starting user model tracing. Output file: PerfMonitorOutput.etl
Found Provider CustomEventSource Guid ff6a40d2-5116-5555-675b-
4468e821162e
Enabling provider ff6a40d2-5116-5555-675b-4468e821162e level:
Verbose keywords:
0xffffffffffffffff
Starting at 4/7/2012 1:44:00 PM
Current Directory C:\PerfMonitor
Executing: Ch02.exe {

} Stopping at 4/7/2012 1:44:01 PM = 0.693 sec
Stopping tracing for sessions 'NT Kernel Logger' and 'PerfMonitorSession'.
Converting C:\PerfMonitor\PerfMonitorOutput.etlx to an XML file.
Output in C:\PerfMonitor\PerfMonitorOutput.dump.xml
PerfMonitor processing time: 1.886 secs.
```

Примечание. *Существует еще один инструмент мониторинга производительности и состояния систем, который еще не упоминался: инструментарий управления Windows (Windows Management Instrumentation, WMI). WMI – инфраструктура контроля и управления (command-and-control, C&C), интегрированная в Windows, но ее рассмотрение далеко выходит за рамки этой главы. Ее можно использовать для получения информации о состоянии системы (например, о версии установленной системы, версии программного обеспечения BIOS или о свободном пространстве на диске), регистрации интересующих событий (таких как запуск и завершение процессов) и вызова управляющих методов, изменяющих состояние системы (таких как создание сетевых разделяемых ресурсов или выгрузка драйверов). За дополнительной информацией о WMI обращайтесь к документации на сайте MSDN: http://msdn.microsoft.com/en-us/library/windows/desktop/aa394582.aspx. Для тех, кому интересна тема создания провайдеров, управляемых механизмом WMI, Саша Голдштейн (Sasha Goldshtein) написал статью «WMI Provider Extensions in .NET 3.5» (http://www.codeproject.com/Articles/25783/WMI-Provider-Extensions-in-NET-3-5), которая послужит отличной отправной точкой.*

Профилировщики времени

Счетчики производительности и механизм ETW позволяет получить массу любопытной информации о производительности приложений для Windows, однако часто бывает желательно иметь инструменты – профилировщики – позволяющие оценить время выполнения на уровне методов и отдельных строк кода (что можно рассматривать, как улучшенную поддержку трассировки стека вызовов в ETW). В этом разделе будет представлено несколько коммерческих инструментов и описаны их преимущества, но имейте в виду, что более мощные и точные инструменты влекут за собой более высокие накладные расходы.

В нашем путешествии по миру профилировщиков мы не раз будем сталкиваться с коммерческими инструментами, но для большинства из них можно найти бесплатные эквиваленты. Мы не собираемся рекламировать каких то определенных производителей инструментов – продукты, демонстрируемые в данной главе являются всего лишь профилировщиками, которыми мы пользуемся чаще всего в своих исследованиях. Как и в случае с любыми другими программными инструментами, ваши предпочтения могут отличаться от наших.

Первый профилировщик, с которым мы познакомимся, является частью Visual Studio и входит в состав этой среды разработки, начиная с версии Visual Studio 2005 (редакция Team Suite). В этой главе мы будем использовать профилировщик из Visual Studio 2012, доступный в редакциях Premium и Ultimate.

Дискретный профилировщик Visual Studio

Профилировщик Visual Studio действует подобно флагу `PROFILE` механизма ETW. Он периодически прерывает работу приложения и сохраняет информацию о стеке вызовов для каждого процессора, где в настоящий момент протекает выполнение приложения или потока. В отличие от провайдера ядра ETW, этот дискретный профилировщик может прерывать работу процессов, опираясь на разные критерии, часть из которых перечислена в табл. 2.2.

Таблица 2.2. Неполный список событий дискретного профилировщика Visual Studio

Критерий	Значение	Типичный диапазон	Используется
Цикл часов	Прерывание выполняется через определенные интервалы времени по часам CPU	1M – 1000M	Для поиска методов, наиболее интенсивно использующих процессор
Ошибка чтения дисковых страниц из памяти	Прерывание выполняется в случае ошибок обращений к страницам памяти, в настоящий момент отсутствующим в ОЗУ и которые необходимо загрузить с диска (из файла подкачки)	1 – 1000	Для поиска методов, вызывающих ошибки чтения дисковых страниц из памяти

Таблица 2.2. (окончание)

Критерий	Значение	Типичный диапазон	Используется
Системные вызовы	Прерывание выполняется при попытке использовать Win32 API или класс .NET, который выполняет обращения к системным службам	1 – 10000	Для поиска методов, вызывающих переключение выполнения из режима пользователя в режим ядра, что является достаточно дорогостоящей операцией
Промахи обращения к кешу	Прерывание выполняется при попытке обратиться к данным, отсутствующим в кеше CPU, но которые могут находиться в ОЗУ	1000 – 1M	Для поиска участков кода, вызывающих промахи кеша (см. также главу 5)
Фактические инструкции	Прерывание выполняется при выполнении определенного количества машинных инструкций	500K – 100M	Подобно циклу часов

Сбор информации с использованием профилировщика Visual Studio обходится достаточно недорого, и если интервал следования событий достаточно широк (по умолчанию он составляет 10 000 000 тактов часов CPU), накладные расходы в общем времени выполнения приложения будут составлять менее 5%. Кроме того, дискретный подход к сбору информации позволяет подключиться к выполняющемуся процессу, собрать данные в течение некоторого времени и затем отключиться от процесса, чтобы проанализировать их. Из-за этой особенности, выявление проблем производительности рекомендуется начинать с поиска участков программы, оказывающих самую большую нагрузку на процессор – методов, на выполнение которых тратится больше всего процессорного времени.

По окончании сеанса сбора информации, профилировщик генерирует сводные таблицы, где каждый метод характеризуется двумя значениями: числом *исключительных попаданий* (exclusive samples), когда в момент отбора очередной порции данных на CPU выполнялся данный метод, и числом *включительных попаданий* (inclusive samples), когда выполнялся данный метод или вызванный им. Методы с большим количеством исключительных попаданий оказывают самую большую нагрузку на процессор; методы с большим количеством включительных попаданий не используют процессор непосредственно, но вызывают другие методы, суммарно занимающие значи-

тельную часть процессорного времени. (Например, в однопоточных приложениях метод `Main` будет иметь 100% включительных попаданий.)

Запуск дискретного профилировщика Visual Studio

Самый простой способ запустить дискретный профилировщик – воспользоваться интерфейсом Visual Studio, хотя (как будет показано ниже) он также может запускаться из командной строки для профилирования действующих приложений. Для проведения данного эксперимента мы рекомендуем использовать одно из ваших собственных приложений.

1. В Visual Studio выберите пункт меню **Analyze → Launch Performance Wizard** (Анализ → Запустить мастер производительности).
2. На первой странице мастера отметьте радиокнопку **CPU sampling** (Выборка циклов ЦП (рекомендуется)) и щелкните на кнопке **Next** (Далее). (Далее в этой главе мы познакомимся с другими режимами работы профилировщика, и тогда вы сможете повторить этот эксперимент.)
3. Если проект для профилирования загружен в текущем решении, щелкните на радиокнопке **One or more available projects** (Один или несколько доступных проектов) и выберите проект из списка. В противном случае щелкните на радиокнопке **An executable (.EXE file)** (Исполняемый файл (.EXE)). Щелкните на кнопке **Next** (Далее).
4. Если на предыдущем шаге вы выбрали параметр **An executable (.EXE file)** (Исполняемый файл (.EXE)), укажите профилировщику путь к выполняемому файлу и аргументы командной строки, если они необходимы, затем щелкните на кнопке **Next** (Далее). (Если у вас нет собственного подходящего приложения, используйте *JackCompiler.exe* из папки с примерами для данной главы.)
5. Оставьте флажок **Launch profiling after the wizard finishes** (Запустить профилирование после завершения работы мастера) отмеченным и щелкните на кнопке **Finish** (Готово).
6. Если среда разработки Visual Studio была запущена не с привилегиями администратора, вам будет предложено повысить привилегии профилировщика.
7. Когда приложение завершит выполнение, откроется отчет профилирования. Используйте раскрывающийся список **Current View** (Текущее представление) для навигации между представлениями, отображающими информацию, собранную в процессе профилирования.

По окончании сеанса профилирования можно также воспользоваться обозревателем производительности (Performance Explorer), чтобы запустить его, выберите пункт меню **Analyze → Windows → Performance Explorer** (Анализ → Окна → Обозреватель производительности). С его помощью можно изменить параметры профилирования (например, выбрать другой интервал опроса или установить другой критерий), изменить целевой файл и сравнить результаты разных сеансов профилирования.

На рис. 2.12 показано окно профилировщика с результатами, где можно видеть ветви в стеке вызовов, на выполнение которых было затрачено больше всего времени, и функции с наибольшим количеством исключительных попаданий. На рис. 2.13 показан подробный отчет, где перечислено несколько методов, на выполнение которых затрачено больше всего процессорного времени (имеющих наибольшее количество исключительных попаданий). Если выполнить двойной щелчок на методе в списке, откроется окно детализации со строками исходного кода в приложении, в которых обнаружено наибольшее число попаданий (рис. 2.14).

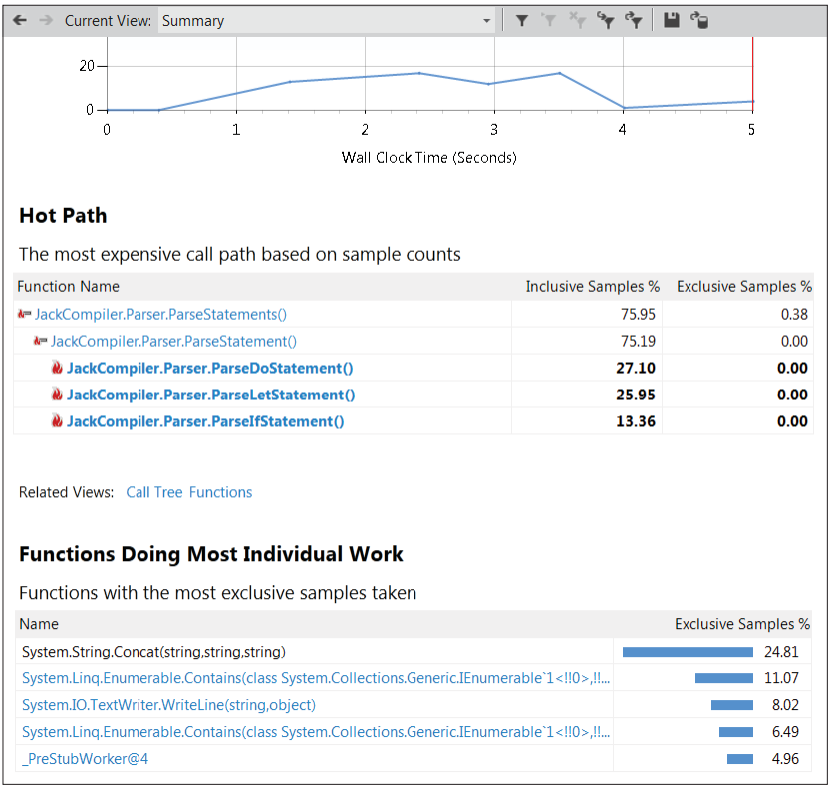

Рис. 2.12. Сводный отчет профилировщика – ветви в стеке вызовов, на выполнение которых было затрачено больше всего времени, и функции с наибольшим количеством исключительных попаданий.

Current View: Functions

Function Name	Inclusive Samp...	Exclusive Sam...
System.String.Concat(string,string,string)	65	65
System.Linq.Erumerable.Contains(class System.Collections.Generic.IEnumerable`1<!!0	29	29
System.IO.TextWriter.WriteLine(string,object)	58	21
System.Linq.Erumerable.Contains(class System.Collections.Generic.IEnumerable`1<!!0	17	17
_PreStubWorker@4	15	13
System.String.Concat(object,object)	11	11
JackCompiler.Tokenizer.NextChar()	15	6
System.IO.File.OpenText(string)	6	6
System.IO.StreamReader.Peek()	6	6
JackCompiler.Tokenizer.Advance()	80	5
System.Collections.Generic.HashSet`1.Contains(!0)	5	5

Рис. 2.13. Отчет о функциях с наибольшим количеством исключительных попаданий. Функция System.String.Concat имеет, как минимум, в два раза больше попаданий, чем любая другая функция.

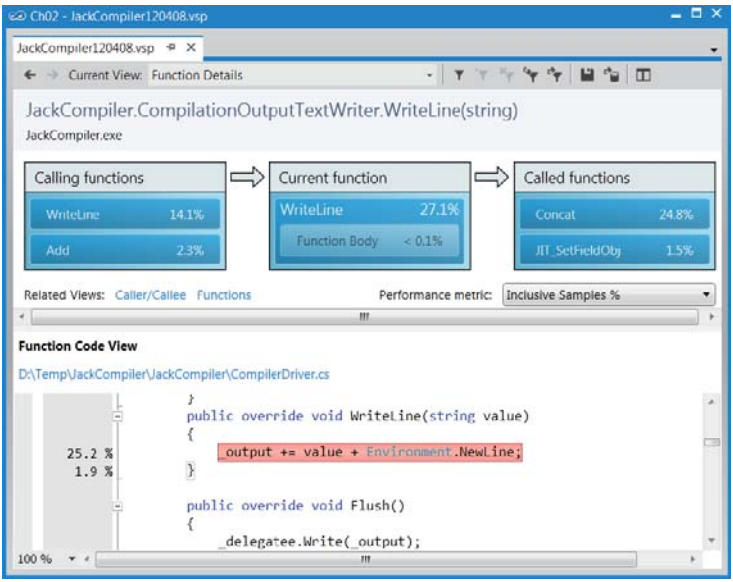

Рис. 2.14. Детализация информации о функциях – показывает функции, вызывающие и вызываемые функцией JackCompiler.CompilationOutputTextWriter.WriteLine. В коде функции выделены строки с наибольшим количеством включительных попаданий.

Внимание. *На первый взгляд кажется, что дискретный мониторинг дает достаточно точные результаты, позволяющие судить о расходовании процессорного времени. Вы могли слышать утверждения, что «если данный*

метод имеет 65% исключительных попаданий, значит на его выполнение тратится 65% процессорного времени». Из-за статистической природы дискретного мониторинга, такие рассуждения являются неоправданными и должны отметаться на практике. Есть несколько факторов, способных внести погрешность в результаты измерений: тактовая частота часов CPU может сотни раз в секунду изменяться в ходе выполнения приложения, из-за чего может искажаться корреляция между числом попаданий и фактическим количеством тактов часов процессора; метод может быть представлен недостаточно, когда моменты отбора информации не совпадают с моментами его выполнения; метод может быть представлен чрезмерно, когда моменты отбора информации наоборот совпадают с моментами его выполнения, но в действительности метод работает совсем непродолжительное время. Поэтому результаты дискретного профилировщика следует рассматривать не как точную картину распределения процессорного времени, а как общую схему, где выделены возможно узкие места.

Помимо количества исключительных/включительных попаданий для каждого метода, профилировщик Visual Studio предлагает массу другой ценной информации. Займитесь его исследованием самостоятельно – представление **Call Tree** (Дерево вызовов) отображает иерархию вызовов методов в приложении (сравните с нисходящим анализом в PerfMonitor, представленным на рис. 2.8), Представление **Lines** (Строки) отображает информацию о попаданиях на уровне строк, а представление **Modules** (Модули) группирует методы по сборкам, и может помочь быстро выяснить, в каком направлении двигаться в поисках узких мест.

Так как при дискретном мониторинге требуется, чтобы поток выполнения в приложении был активен в моменты сбора информации, нет никакого способа получить информацию о потоках, простаивавших в ожидании выполнения операций ввода/вывода или на блокировках механизмов синхронизации. Дискретный мониторинг отлично подходит для анализа приложений, занимающихся преимущественно вычислениями, но для приложений, основная работа которых заключается в выполнении операций ввода/вывода, необходимы иные подходы, опирающиеся на использование других, более глубинных механизмов профилирования.

Инструментированный профилировщик Visual Studio

Профилировщик Visual Studio поддерживает еще один режим работы, называемый *инструментированным профилированием* (instrumentation profiling), позволяющий измерить общее время выполнения, а

не только процессорное время. Он хорошо подходит для профилирования приложений, выполняющих большое количество операций ввода/вывода или интенсивно использующих механизмы синхронизации. В режиме инструментированного профилирования профилировщик изменяет целевой выполняемый файл и внедряет в него код, выполняющий измерения и сообщающий профилировщику точную информацию о времени выполнения и количестве вызовов каждого оцениваемого таким способом метода.

Например, взгляните на следующий метод:

```
public static int InstrumentedMethod(int param) {
    List< int > evens = new List < int > ();
    for (int i = 0; i < param; ++i) {
        if (i % 2 == 0) {
            evens.Add(i);
        }
    }
    return evens.Count;
}
```

Для оценки характеристик производительности этого метода, профилировщик Visual Studio изменит его. Помните, что изменению подвергается двоичный выполняемый файл – исходный код никак при этом не изменяется, но вы всегда сможете исследовать измененный двоичный файл с применением дизассемблера IL, такого как .NET Reflector. (Чтобы сэкономить место в книге, мы немного подправили результат, возвращаемый дизассемблером.)

```
public static int mmid = (int)
    Microsoft.VisualStudio.Instrumentation.g_fldMMID_2D71B909-
C28E-4fd9-A0E7-ED05264B707A;

public static int InstrumentedMethod(int param) {
    _CAP_Enter_Function_Managed(mmid, 0x600000b, 0);
    _CAP_StartProfiling_Managed(mmid, 0x600000b, 0xa000018);
    _CAP_StopProfiling_Managed(mmid, 0x600000b, 0);
    List < int > evens = new List < int > ();
    for (int i = 0; i < param; i++) {
        if (i % 2 == 0) {
            _CAP_StartProfiling_Managed(mmid, 0x600000b, 0xa000019);
            evens.Add(i);
            _CAP_StopProfiling_Managed(mmid, 0x600000b, 0);
        }
    }
    _CAP_StartProfiling_Managed(mmid, 0x600000b, 0xa00001a);
    _CAP_StopProfiling_Managed(mmid, 0x600000b, 0);
    int count = evens.Count;
```

```
    _CAP_Exit_Function_Managed(mmid, 0x600000b, 0);
    return count;
}
```

Вызовы методов, имена которых начинаются с приставки `_CAP`, – это обращения к модулю *VSPerf110.dll*, на который ссылается инструментированная сборка. Они отвечают за измерение интервалов времени и запись счетчиков вызовов методов. Так как механизм учета перехватывает все вызовы методов, выполненные из инструментированного метода, информация, получаемая в результате инструментированного профилирования, может быть очень точной.

Если то же самое приложение, представленное на рис. 2.12, 2.13 и 2.14 запустить в инструментированном режиме (вы можете сделать это с приложением *JackCompiler.exe*), профилировщик сгенерирует отчет с представлением **Summary** (Сводка), содержащим похожую информацию – о наиболее затратных ветвях в стеке вызовов и отдельные функциях, на выполнение которых тратится больше всего времени. Однако на этот раз информация в отчете будет основана не на дискретном мониторинге (позволяющем измерять только потоки, выполняющиеся на CPU), а на точном хронометраже, проводимом инструментированным кодом. На рис. 2.15 показано представление **Functions** (Функции), где доступны включительное и исключительное время, измеренные в миллисекундах, а также счетчики вызовов функций.

Current View: Functions

Function Name	Elapsed Inclusi...	Elapsed Exclus...	Number of Calls
JackCompiler.Tokenizer.NextChar()	4,163.71	2,246.36	951,000
JackCompiler.Token..ctor(valuetype JackCompiler.TokenType,string)	2,561.65	1,557.73	463,000
JackCompiler.Tokenizer.Advance()	9,195.09	1,070.54	293,000
JackCompiler.Parser.Match(class JackCompiler.Token)	7,751.63	749.13	171,000
JackCompiler.Tokenizer.EatWhile(class System.Predicate`1<char>)	2,445.82	716.53	144,000
System.Object..ctor()	627.72	627.72	483,000
System.String.op_Equality(string,string)	580.68	580.68	1,161,000
System.String.get_Chars(int32)	481.38	481.38	951,000
JackCompiler.Parser.ParseStatements()	9,734.91	456.07	6,000
System.Predicate`1.Invoke(!0)	501.06	435.24	525,000
JackCompiler.Tokenizer.get_IsAtEnd()	418.59	404.36	293,000
System.Environment.get_NewLine()	391.10	391.10	951,000
System.String.Concat(string,string)	383.15	383.15	1,091,000
System.String..ctor(char[])	342.48	342.48	463,000
JackCompiler.Parser.NextToken()	2,586.89	329.52	121,000

Рис. 2.15. Представление Functions (Функции): Метод System.String.Concat уже не выглядит узким местом в смысле производительности, теперь внимание сместилось к методам JackCompiler.Tokenizer.NextChar и JackCompiler.Token..ctor. Первый из них был вызван почти миллион раз.

Совет. *Приложение, использовавшееся для создания отчетов на рис. 2.12 и 2.15 не является исключительно вычислительным. Фактически, большую часть времени оно тратит в ожидании завершения операций ввода/вывода. Это объясняет отличия результатов дискретного мониторинга, указывающих, что наиболее дорогостоящим является метод* `System.String.Concat`*, от результатов инструментированного профилирования, согласно которым узким местом является метод* `JackCompiler.Tokenizer.NextChar`.

Инструментированное профилирование выглядит более точным, но на практике рекомендуется использовать дискретное профилирование, если приложение в основном решает вычислительные задачи. Прием инструментирования имеет ограниченную гибкость из-за необходимости изменять выполняемый файл приложения перед запуском и невозможности подключить профилировщик к уже запущенному процессу. Кроме того, инструментированное профилирование имеет немалые накладные расходы – объем выполняемого кода существенно увеличивается и в процессе выполнения часть времени затрачивается на отбор информации в точках входа и выхода из методов. (Некоторые инструментированные профилировщики предлагают режим построчного профилирования кода, когда каждая строка окружается кодом профилировщика, выполняющим измерения – такой код работает еще медленнее!)

Как всегда, будет ошибкой безоговорочно доверять результатам инструментированного профилирования. Разумеется, количество вызовов того или иного метода не изменится из-за того, что приложение подвергается инструментированному профилированию, но информация о времени все еще может существенно искажаться из-за накладных расходов, несмотря на все попытки профилировщика сместить все дорогостоящие вычисления в конец. При внимательном подходе, дискретное и инструментированное профилирование могут помочь понять, где приложение проводит больше всего времени, особенно если вы будете сопоставлять множество отчетов и обращать внимание на результаты, полученные в результате оптимизации.

Дополнительные приемы использования профилировщиков времени

Профилировщики времени прячут в своих рукавах ряд хитростей, о которых не рассказывалось в предыдущих разделах. Эта глава слишком коротка, чтобы можно было обсудить их во всех подробностях, но их стоит отметить, чтобы вы не пропустили их, когда будете пользоваться мастерами Visual Studio.

Советы по дискретному профилированию

Как было показано в разделе «Дискретный профилировщик Visual Studio», дискретный профилировщик способен собирать события различных типов, включая промахи кеша и ошибки чтения дисковых страниц. В главах 5 и 6 будет представлено еще несколько примеров приложений, производительность которых может быть существенно улучшена за счет настройки параметров доступа к памяти, и в первую очередь – за счет уменьшения количества промахов кеша. Профилировщик окажется весьма ценным инструментом для выявления участков кода, где наиболее часто возникают промахи кеша и ошибки чтения страниц в этих приложениях. (Используя прием инструментированного профилирования так же можно собирать информацию о различных счетчиках CPU, таких как промахи кеша, фактически выполненные машинные инструкции и ошибочно предсказанные ветвления. Для этого откройте диалог свойств сеанса профилирования в панели **Performance Explorer** (Обозреватель производительности) и перейдите на вкладку **CPU Counters** (Счетчики ЦПУ). Собранная информация будет доступна в представлении **Functions** (Функции) в виде дополнительных столбцов.)

Дискретное профилирование в целом обладает большей гибкостью, чем инструментированное. Например, в панели **Performance Explorer** (Обозреватель производительности) можно подключить профилировщик (в дискретном режиме) к уже запущенному процессу.

Сбор дополнительных данных при профилировании

Во всех режимах профилирования, в панели **Performance Explorer** (Обозреватель производительности) можно приостановить и возобновить сбор данных не прерывая сеанс профилирования, и сгенерировать метки, которые будут видимы в заключительном отчете профилировщика, чтобы проще было отличить различные этапы выполнения приложения. Эти метки также будут видимы в представлении **Marks** (Метки).

> **Совет.** *Профилировщик Visual Studio экспортирует свой API, который может использоваться приложениями для приостановки и возобновления профилирования программным способом. Эту возможность можно использовать, чтобы отключить сбор данных в неинтересных частях приложения и уменьшить размер файлов с данными профилировщика. За дополнительной информацией о доступных методах профилировщика обращайтесь к документации на сайте MSDN: http://msdn.microsoft.com/ru-ru/library/bb514149%28v=vs.110%29.aspx.*

Профилировщик может также собирать информацию из счетчиков производительности Windows и провайдеров событий ETW (обсуждались выше в этой главе). Для этого откройте диалог свойств сеанса профилирования в панели **Performance Explorer** (Обозреватель производительности) и перейдите на вкладку **Windows Events** (События Windows) или **Windows Counters** (Счетчики Windows). Данные ETW можно просмотреть только из командной строки, с помощью команды `VSPerfReport /summary:ETW`, тогда как счетчики производительности будут доступны в представлении **Marks** (Метки), в интерфейсе Visual Studio.

Наконец, если Visual Studio тратит слишком много времени на создание отчета с большим количеством данных, можно избежать этих затрат, щелкнув правой кнопкой мыши на отчете в панели **Performance Explorer** (Обозреватель производительности) и выбрать пункт контекстного **Save Analyzed Report** (Сохранить проанализированный отчет). Сохраненные файлы отчетов имеют расширение *.vsps* и открываются практически мгновенно.

Рекомендации профилировщика

После открытия отчета в Visual Studio можно заметить появление раздела **Profiler Guidance** (Рекомендации профилировщика), содержащего множество полезных советов по устранению типичных проблем производительности, обнаруженных при профилировании и обсуждаемых в этой книге, в том числе:

- «Consider using `StringBuilder` for string concatenations» («Рассмотрите возможность использования `StringBuilder` для конкатенации строк») – весьма полезное правило, которое поможет уменьшить количество операций, производимых сборщиком мусора, и, соответственно, сократить время его работы, как описывается в главе 4;
- «Many of your objects are being collected in generation 2 garbage collection» («Многие из ваших объектов собираются в процессе сборки мусора 2-го поколения») – феномен «кризиса среднего возраста» объектов, так же обсуждается в главе 4;
- «Override `Equals` and equality operator on value types» («Переопределять `Equals` и оператор равенства в типах значений») – важнейший прием оптимизации для часто используемых типов значений, обсуждается в главе 3;
- «You may be using Reflection excessively. It is an expensive operation» («Возможно, слишком интенсивно используется отра-

жение. Это является затратной операцией») – обсуждается в главе 10.

Дополнительные настройки профилирования

Сбор информации о производительности в промышленном окружении может оказаться сложным делом, если для этого потребуется устанавливать такие тяжеловесные инструменты, как Visual Studio. К счастью, профилировщик Visual Studio можно установить в промышленном окружении отдельно, не устанавливая всю среду разработки Visual Studio. Файлы дистрибутива профилировщика можно найти на установочном носителе Visual Studio, в каталоге *Standalone Profiler* (существуют версии для 32- и 64-разрядных систем). После установки профилировщика, следуйте инструкциям на странице http://msdn.microsoft.com/ru-ru/library/ms182401%28v=vs.110%29.aspx для запуска приложения под управлением профилировщика или для подключения профилировщика к существующему процессу с использованием инструмента *VSPerfCmd.exe*. По завершении сеанса, профилировщик сгенерирует файл *.vsp*, который можно будет открыть на другом компьютере в Visual Studio или на этом же компьютере воспользоваться инструментом *VSPerfReport.exe*, чтобы сгенерировать отчет в формате XML или CSV и исследовать их без использования Visual Studio.

При проведении инструментированного профилирования в вашем распоряжении имеется множество параметров командной строки, которые можно передать утилите *VSInstr.exe*. В частности, для запуска и приостановки профилирования в определенной функции, а также для включения/выключения инструментирования функций на основе шаблонов их имен можно использовать параметры START, SUSPEND, INCLUDE и EXCLUDE. Более подробную информацию об утилите *VSInstr.exe* можно найти на сайте MSDN: http://msdn.microsoft.com/ru-ru/library/ms182402.aspx.

Некоторые профилировщики поддерживают режим удаленного профилирования, что позволяет запускать основной пользовательский интерфейс профилировщика на одном компьютере, а собственно сеанс профилирования – на другом, без необходимости копировать отчеты о производительности вручную. Например, профилировщик dotTrace компании JetBrains поддерживает такой режим с применением небольшого удаленного агента, выполняемого на удаленном компьютере и взаимодействующего с главным пользовательским интерфейсом профилировщика. Это – отличная альтернатива установ-

ке целого комплекта файлов профилировщика на компьютер, находящийся в промышленной эксплуатации.

> **Примечание.** *В главе 6 будет показан прием использования GPU для параллельных вычислений, дающий значительное увеличение производительности (до 100 раз!). Стандартные профилировщики времени оказываются бесполезными, когда требуется выявить проблемы производительности в коде, выполняющемся на GPU. Однако существуют инструменты, способные выполнять профилирование такого кода, включая и профилировщик Visual Studio 2012. Обсуждение данной темы далеко выходит за рамки этой главы, но, если вы используете прием организации параллельных вычислений на GPU, есть смысл поближе познакомиться с инструментами, способными интегрироваться с фреймворками программирования для GPU (такими как C++ AMP, CUDA или OpenCL).*

В этом разделе мы достаточно подробно рассмотрели, как проанализировать распределение времени выполнения приложений (общего или только процессорного) с применением профилировщика Visual Studio. Другим важным фактором, влияющим на производительность управляемых приложений, является управление выделением памяти. В следующих двух разделах мы познакомимся с профилировщиками выделения памяти и профилировщиками памяти, которые способны оказать помощь в выявлении узких мест в производительности, связанных с памятью.

Профилировщики выделения памяти

Профилировщики этого типа выявляют операции выделения памяти в приложении и сообщают, какие методы выделяют больше всего памяти, какого типа объекты создаются и другие статистики, касающиеся памяти. Интенсивное выделение памяти в приложениях часто влечет за собой значительные накладные расходы на сборку мусора. Как будет показано в главе 4, выделение памяти в среде выполнения CLR является недорогой операцией, но ее освобождение может сопровождаться значительными накладными расходами. По этой причине группы небольших методов, выделяющих большие объемы памяти, могут отнимать совсем немного процессорного времени и быть практически незаметны в отчете профилировщика времени, но при этом вызывать значительные задержки на сборку мусора в случайных точках выполнения приложения. В своей практике мы встречали приложения, где память выделялась без должного внимания, и нам

удавалось увеличивать их производительность – иногда в 10 раз – оптимизировав выделение памяти и управление ею.

Для профилирования выделения памяти мы будем использовать два профилировщика – вездесущий профилировщик Visual Studio, поддерживающий режим профилирования выделения памяти, и профилировщик CLR Profiler – самостоятельный и бесплатный инструмент. К сожалению, оба инструмента часто оказывают значительное влияние на производительность приложений, интенсивно использующих динамическую память, потому что для каждой операции выделения памяти профилировщик выполняет последовательность действий по сохранению информации для последующего составления отчетов. Тем не менее, результаты могут оказаться настолько ценными, что даже 100-кратное замедление при профилировании можно потерпеть.

Профилировщик выделения памяти Visual Studio

Профилировщик Visual Studio способен собирать информацию об операциях выделения памяти и жизненном цикле объектов (которые освобождаются сборщиком мусора) в обоих режимах, дискретном и инструментированном. В дискретном режиме профилировщик собирает информацию о выделении памяти в приложении в целом. В инструментированном режиме информация собирается только из инструментированных модулей.

Для экспериментов с профилировщиком Visual Studio вы можете использовать приложение *JackCompiler.exe* из примеров к этой главе. В мастере настройки профилировщика производительности выберите радиокнопку **.NET memory allocation** (Выделение памяти .NET). В конце сеанса профилирования, в представлении **Summary** (Сводка), будут показаны функции, выделившие памяти больше всего (рис. 2.16). В представлении **Functions** (Функции) для каждого метода будет указано количество объектов и количество байтов памяти, выделенных методом (как обычно, включительные и исключительные значения). В представлении **Function Details** (Сведения о функции) будет представлена информация о вызывающих и вызываемых функциях, а также указаны строки кода с объемами выделенной ими памяти в поле слева (рис. 2.17). Но самая интересная информация содержится в представлении **Allocation** (Выделение), показывающем, какие ветви в стеке вызовов выделили памяти больше всего (рис. 2.18).

Functions Allocating Most Memory

Functions with the highest exclusive bytes allocated

Name	Bytes %
System.String.Concat(string,string,string)	89.04
System.String.CtorCharCount(char,int32)	1.66
System.IO.TextWriter.WriteLine(string,object)	1.43
System.String.Concat(object,object)	1.41
JackCompiler.Tokenizer.Advance()	1.03

Types With Most Memory Allocated

Types with the hightest total number of bytes allocated

Name	Bytes %
System.String	95.53
System.Char[]	1.47
JackCompiler.Token	0.65
System.String[]	0.48
System.Byte[]	0.45

Рис. 2.16. Представление **Summary** (Сводка) с результатами профилирования выделения памяти.

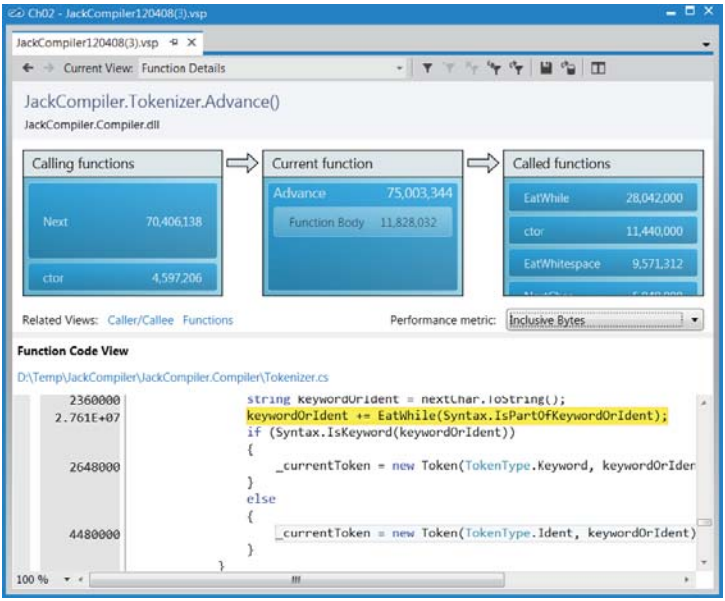

Рис. 2.17. Представление **Function Details** (Сведения о функции) с информацией для функции JackCompiler.Tokenizer.Advance, демонстрирующее вызывающую и вызываемую функции, а также строки в данной функции, выделившие память.

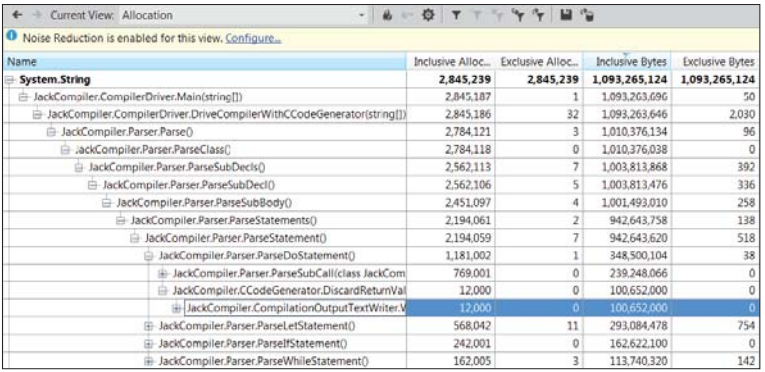

Рис. 2.18. Представление **Allocation** (Выделение), демонстрирующее ветвь в стеке вызовов, выделявшую память для объектов System.String.

В главе 4 мы узнаем, насколько важно отказаться от использования временных объектов, и обсудим феномен «кризиса среднего возраста» объектов, оказывающего существенное влияние на производительность, который проявляется в способности объектов переживать несколько циклов сборки мусора. Идентифицировать наличие этого явления в приложении можно с помощью представления **Object Lifetime** (Жизненный цикл объектов), сообщающем, в каком поколении объекты были утилизированы. Это представление поможет увидеть, имеются ли объекты, пережившие слишком много циклов сборки мусора. На рис. 2.19 можно видеть, что все объекты строк, созданные приложением (и занимающие более 1 Гбайта памяти!) были утилизированы в нулевом поколении, а это означает, что ни одному из них не удалось прожить дольше одного цикла сборки мусора.

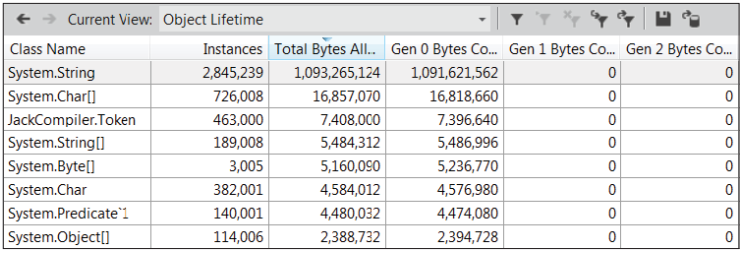

Рис. 2.19. Представление **Object Lifetime** (Жизненный цикл объектов) поможет выявить временные объекты, пережившие несколько циклов сборки мусора. В данном случае все объекты были утилизированы в нулевом поколении, что удешевило затраты на сборку мусора. (Подробности см. в главе 4.)

Хотя отчеты о выделении памяти, генерируемые профилировщиком Visual Studio, отличаются богатством информации, в них все же имеются некоторые недостатки. Например, трассировка стека вызовов с целью сгруппировать операции выделения памяти по типам объектов занимает достаточно много времени, если выделение памяти производится во множестве разных мест (что всегда верно для строк и массивов байтов). Профилировщик CLR Profiler поддерживает несколько дополнительных особенностей, делающих его ценной альтернативой профилировщику Visual Studio.

CLR Profiler

Профилировщик CLR Profiler – это отдельный инструмент профилирования, не требующий установки и занимающий менее 1 Мбайта дискового пространства. Загрузить его можно по адресу: http://www.microsoft.com/download/en/details.aspx?id=16273. Как дополнительное преимущество, он распространяется с исходными текстами, которые наверняка заинтересуют тех, кто пожелает заняться созданием собственных инструментов, использующих CLR Profiling API. Он способен подключаться к выполняющимся процессам (если используется версия CLR не ниже 4.0) или запускать выполняемые файлы, и регистрировать все операции выделения памяти и события сборки мусора.

Пользоваться профилировщиком CLR Profiler очень просто – запустите профилировщик, щелкните на кнопке **Start Application** (Запустить приложение), выберите приложение для профилирования и дождитесь появления отчета – богатство информации для кого-то может оказаться ошеломляющим. Мы рассмотрим здесь некоторые отчеты профилировщика, а полное руководство вы найдете в документе *CLRProfiler.doc*, входящем в состав загружаемого пакета. Как обычно, для экспериментов с профилировщиком вы можете использовать пример приложения *JackCompiler.exe*.

На рис. 2.20 показан главный отчет, появляющийся после завершения профилируемого приложения. Он содержит основные сведения, касающиеся выделения памяти и сборки мусора. Далее из этого отчета можно пойти в нескольких направлениях. Мы сконцентрируемся на исследовании источников выделения памяти, чтобы понять, в каком месте приложения создается больше всего объектов (этот отчет напоминает представление **Allocation** (Выделение) профилировщика Visual Studio). Мы могли бы заняться исследованием сборки мусора, чтобы узнать, какие объекты утилизируются. Наконец, можно было

бы исследовать содержимое динамической памяти, чтобы получить представление о ее распределении.

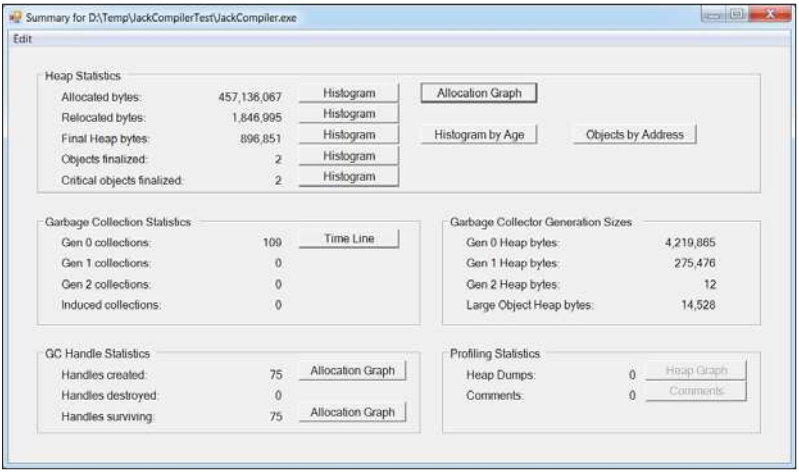

Рис. 2.20. Главный отчет профилировщика CLR Profiler, отображающий информацию о выделении памяти и сборке мусора.

Щелчок на кнопке **Histogram** (Гистограмма) рядом с полем **Allocated bytes** (Выделено байтов) или **Final heap bytes** (Конечный объем кучи в байтах) выведет гистограмму по типам объектов, сгруппированных по размерам. Эти гистограммы можно использовать для выявления больших и малых объектов, а также объектов, создаваемых чаще других. На рис. 2.21 показана гистограмма для всех объектов, создаваемых нашим примером приложения.

Щелчок на кнопке **Allocation Graph** (График выделения) (рис. 2.20) откроет отчет о выделении памяти в дереве стека вызовов для всех объектов в приложении. Информация в этом отчете сгруппирована так, чтобы легко можно было перейти от методов, выделивших больше всего памяти, к отдельным типам объектов и посмотреть, какие методы создали больше всего экземпляров этих типов. На рис. 2.22 показана малая часть графика выделения памяти, начиная от метода `Parser.ParseStatement`, выделившего (включительно) 372 Мбайт памяти и до различных методов, вызываемых им. (Кроме того, в остальных отчетах профилировщика CLR Profiler присутствует пункт контекстного меню **Show who's allocated** (Показать, для кого выделена память), открывающий отчет с графиком выделения памяти для подмножества объектов приложения.)

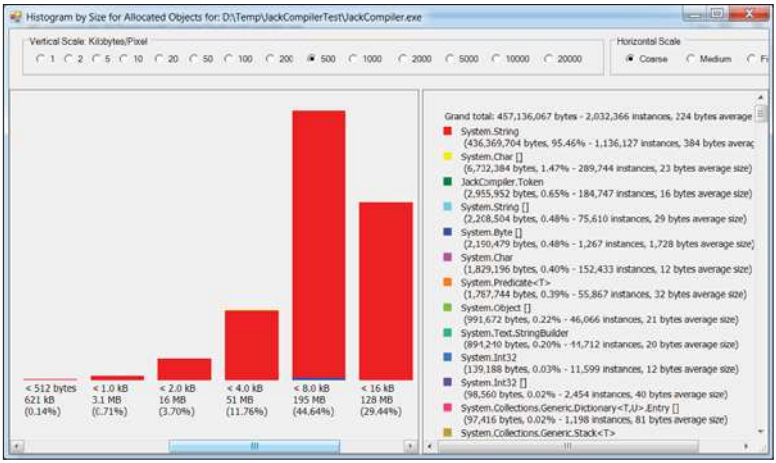

Рис. 2.21. Все объекты, созданные профилируемым приложением. Каждый столбик представляет объекты определенного размера. Легенда справа содержит общее число созданных экземпляров каждого типа и объем выделенной памяти в байтах.

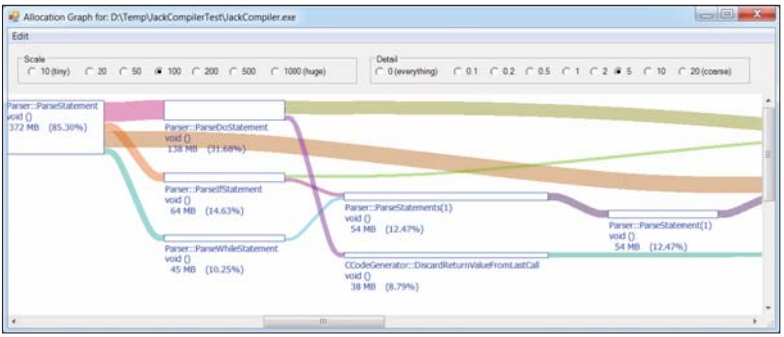

Рис. 2.22. График выделения памяти в профилируемом приложении. Здесь показаны только методы; информация о фактических типах объектов находится на графике правее.

Щелчок на кнопке **Histogram by Age** (Гистограмма по возрасту) (рис. 2.20) откроет гистограмму, где объекты сгруппированы по возрасту. Она позволяет быстро обнаружить долгоживущие и временные объекты, что может пригодиться для борьбы с явлением «кризиса среднего возраста». (Подробнее об этом рассказывается в главе 4.)

Щелчок на кнопке **Objects by Address** (Объекты по адресу) (рис. 2.20) отобразит области управляемой динамической памяти в

виде слоев; чем ниже слой, тем он старше (рис. 2.23). Подобно археологу вы можете погружаться в более глубокие слои и выяснять, какие объекты занимают память в вашем приложении. Этот отчет можно также использовать для диагностики фрагментации динамической памяти – подробнее об этом мы поговорим в главе 4.

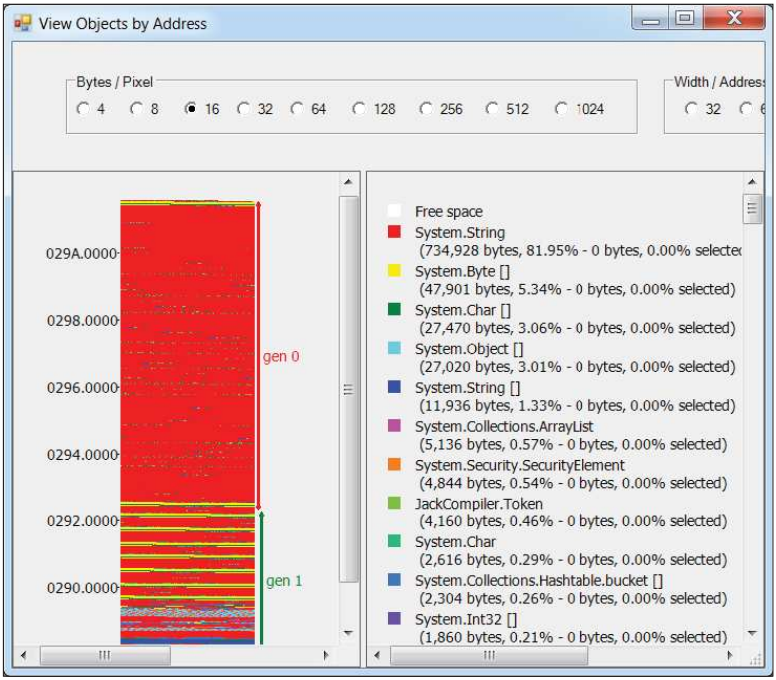

Рис. 2.23. Визуальное представление динамической памяти приложения. Метки слева – это адреса; метки «gen 0» и «gen 1» – это подразделы динамической памяти, обсуждаемые в главе 4.

Наконец, щелчок на кнопке **Time Line** (График времени) в разделе **Garbage Collection Statistics** (Статистика сборки мусора) (рис. 2.20) откроет отчет с информацией об отдельных циклах сборки мусора и их влиянии на динамическую память приложения (рис. 2.24). Этот график можно использовать для определения типов утилизируемых объектов, и получения представления о том, как изменяется распределение динамической памяти после сборки мусора. С его помощью можно также выявлять утечки памяти, когда сборщик мусора освобождает недостаточно памяти, из-за того, что приложение продолжает удерживать все увеличивающееся количество объектов.

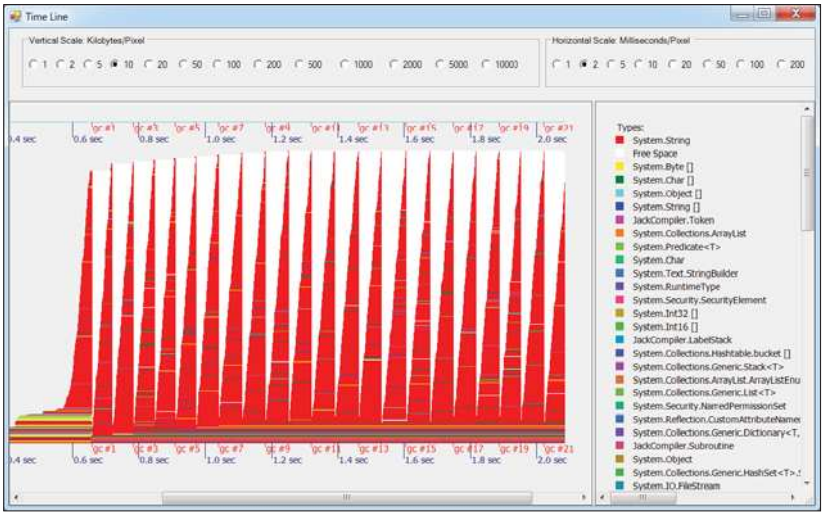

Рис. 2.24. График сборки мусора во времени. Отметки на нижней оси представляют отдельные циклы сборки мусора, а в области графика изображается состояние управляемой динамической памяти. После сборки мусора объем используемой памяти резко уменьшается, а затем постепенно возрастает, до следующего цикла. В данном случае объем используемой памяти (после сборки мусора) остается постоянным, из чего следует, что в этом приложении отсутствуют утечки памяти.

Графики и гистограммы выделения памяти – весьма полезные инструменты анализа, но иногда важнее бывает выявить ссылки между объектами, а не объемы выделяемой памяти в разных методах. Например, когда в приложении обнаруживаются утечки управляемой памяти, очень полезно пройтись по динамической памяти, чтобы найти категории объектов, занимающие наибольшие объемы памяти и узнать, какие объекты на них ссылаются, мешая сборщику мусора утилизировать их. Пока профилируемое приложение выполняется, щелкните на кнопке **Show Heap now** (Показать кучу сейчас), чтобы сгенерировать *дамп динамической памяти* для последующего исследования с целью классификации ссылок между объектами.

На рис. 2.25 показан отчет профилировщика сразу с тремя дампами динамической памяти, расположенными друг над другом, показывающий увеличение количества объектов типа `byte[]`, удерживаемых очередью объектов, готовых к завершению (f-reachable queue) (описывается в главе 4), из-за наличия ссылок на них в объектах `Employee`

и `Schedule`. На рис. 2.26 показаны результаты выбора пункта **Show New Objects** (Показать новые объекты) контекстного меню, чтобы оставить в отчете только объекты, созданные в период времени между сохранением второго и третьего дампов.

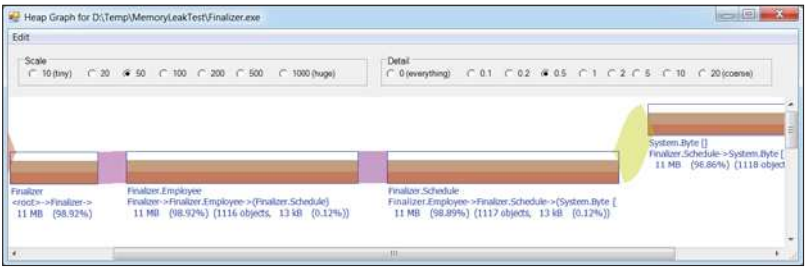

Рис. 2.25. Три дампа динамической памяти друг над другом, показывающих, что в куче удерживаются 11 Мбайт экземпляров типа byte[].

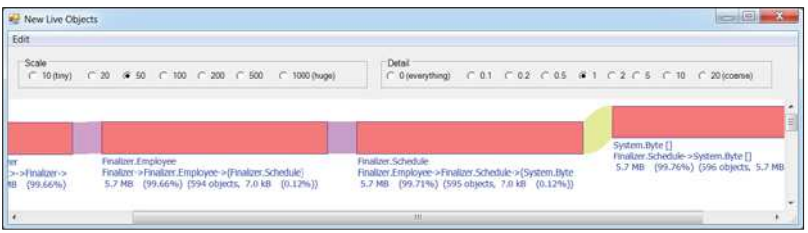

Рис. 2.26. Новые объекты, созданные между в период между сохранением последнего и предпоследнего дампов. На этом графике видно, что источником утечки памяти является цепочка ссылок из очереди объектов, готовых к завершению.

Дампы динамической памяти, созданные профилировщиком CLR Profiler, можно использовать для диагностики утечек памяти в приложениях, но средств визуализации, упрощающих это, в данном профилировщике явно недостаточно. Коммерческие инструменты, которые мы рассмотрим далее, предлагают более богатые возможности, включая автоматические средства обнаружения наиболее типичных источников утечек памяти, разнообразные фильтры и возможности более сложной группировки информации. Поскольку большинство из этих инструментов не сохраняют информацию о каждом объекте, размещенном в динамической памяти, и не сохраняют информацию о выделении памяти в разных методах, они имеют более низкие накладные расходы, что само по себе является большим преимуществом.

Профилировщики памяти

В этом разделе мы познакомимся с двумя коммерческими профилировщиками памяти, специализирующимися на визуализации состояния динамической памяти и выявлении источников утечек памяти. Поскольку эти инструменты отличаются большой сложностью, мы исследуем лишь малое подмножество их особенностей и оставим читателю возможность ознакомиться с остальными самостоятельно, прочитав соответствующие руководства.

Профилировщик памяти ANTS

Профилировщик ANTS Memory Profiler компании RedGate специализируется на анализе срезов динамической памяти. Ниже подробно описывается процесс использования ANTS Memory Profiler для диагностики утечек памяти. Если у вас есть желание самим повторить описываемые действия, загрузите пробную 14-дневную версию ANTS Memory Profiler, которую можно найти по адресу: http://www.red-gate.com/products/dotnet-development/ants-memory-profiler/, и используйте ее для профилирования собственного приложения. Описание и скриншоты, представленные ниже, относятся к версии ANTS Memory Profiler 7.3, которая была последней на момент написания данных строк.

Следуя за экспериментом, описываемым ниже, вы можете использовать пример приложения *FileExplorer.exe* из папки с примерами для этой главы. Это приложение имитирует утечку памяти, выполняя обход дерева каталогов и не освобождая объекты с информацией о непустых каталогах.

1. Запустите приложение из профилировщика. (Подобно профилировщику CLR Profiler, ANTS поддерживает возможность подключения к выполняющимся процессам, начиная с версии CLR 4.0.)
2. По завершении инициализации приложения щелкните на кнопке **Take Memory Snapshot** (Сделать снимок памяти). Этот снимок будет служить основой для последующих исследований.
3. Накопив утечки памяти, сделайте еще один снимок динамической памяти.
4. После завершения приложения сравните снимки (базовый снимок с последним или промежуточные друг с другом) что-

бы определить, какие типы объектов приводят к увеличению объема занимаемой памяти.

5. Выберите определенный тип и щелкните на кнопке **Instance Categorizer** (Классификатор экземпляров), чтобы понять, какие ссылки удерживают в памяти объекты подозреваемого в утечках типа. (На этом этапе исследуются ссылки между типами – экземпляры типа A, ссылающиеся на экземпляры типа B, будут сгруппированы по типу.)
6. Исследуйте отдельные экземпляры подозреваемых в утечках типов, щелкнув на кнопке **Instance List** (Список экземпляров). Выберите несколько наиболее представительных экземпляров и щелкните на кнопке **Instance Retention Graph** (График зависимостей экземпляров), чтобы посмотреть, почему они удерживаются в памяти. (На этом этапе исследуются ссылки между отдельными объектами, и здесь можно выяснить причину, мешающую сборщику мусора утилизировать конкретные объекты.)
7. Вернитесь в исходный код приложения и измените его так, чтобы объекты, вызывающие утечку, своевременно уничтожали ссылки на проблематичные цепочки.

К концу процесса анализа у вас должно сложиться четкое представление, почему самые тяжеловесные объекты в вашем приложении не утилизируются сборщиком мусора. Существует множество причин, вызывающих утечки памяти, и их выявление проблемных объектов из миллионов имеющихся – это целое искусство.

На рис. 2.27 показан пример сравнения двух снимков динамической памяти. Основные утечки памяти (в байтах) связаны с объектами `string`. Детальный осмотр типа `string` после щелчка на кнопке **Instance Categorizer** (Классификатор экземпляров) (рис. 2.28) наводит на мысль, что некоторое событие создает экземпляры `FileInformation` в памяти, которые в свою очередь хранят ссылки на объекты `byte[]`. Более детальное исследование конкретных экземпляров в представлении, выводимом после щелчка на кнопке **Instance Retention Graph** (График зависимостей экземпляров) (рис. 2.29) в результате указывает, что источником утечек является статическое событие `FileInformation.FileInformationNeedsRefresh`.

Рис. 2.27. Сравнение двух снимков динамической памяти. Общая разница между ними составляет +6.23 Мбайт, а наибольший объем занимают объекты типа System.String.

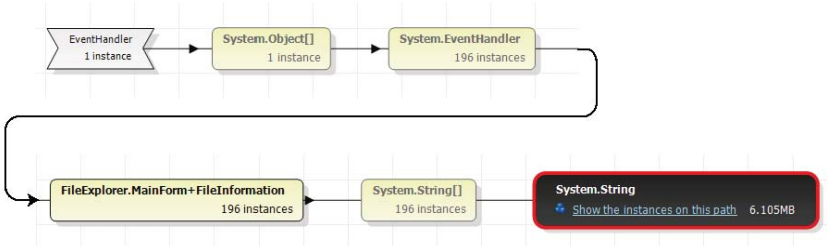

Рис. 2.28. Строки удерживаются в памяти массивами строк, которые сами удерживаются экземплярами типа FileInformation, которые в свою очередь удерживаются событием (через делегаты System.EventHandler).

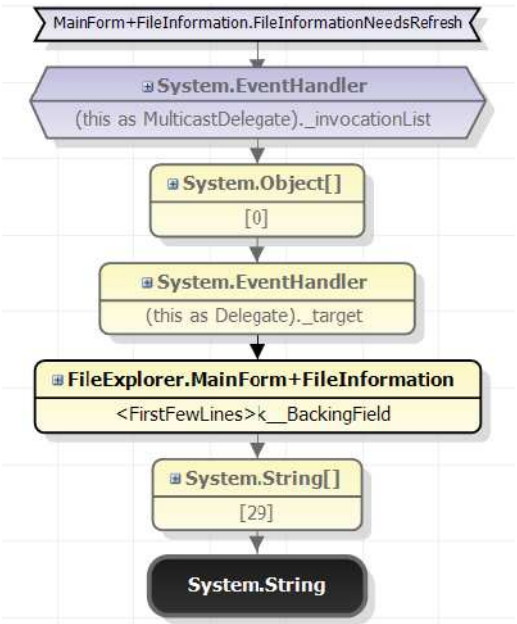

Рис. 2.29. Для исследования был выбран 29-й элемент массива строк, хранящийся в поле <FirstFewLines>k__BackingField объекта FileInformation. Ссылки приводят нас к статическому событию FileInformation.FileInformationNeedsRefresh.

Профилировщик памяти SciTech .NET

Профилировщик SciTech .NET Memory Profiler – еще один коммерческий инструмент, предназначенный для выявления утечек памяти. В общем и целом процесс анализа в этом профилировщике похож на процесс анализа с использованием ANTS Memory Profiler. Однако в отличие от последнего он способен открывать *аварийные файлы дампов памяти*, что дает возможность применять его не только для профилирования приложения, но и для анализа аварийных дампов памяти, созданных средой выполнения CLR при исчерпании доступной памяти. Эта возможность может пригодиться для диагностики проблем уже *после аварии*. Загрузить 10-дневную пробную версию можно по адресу: http://memprofiler.com/download.aspx. Описание и скриншоты, представленные ниже, относятся к версии .NET Memory Profiler 4.0, которая была последней на момент написания данных строк.

Примечание. *Профилировщик CLR Profiler не может открывать аварийные файлы дампов памяти непосредственно. Однако библиотека* `SOS.DLL` *поддерживает команду* `!TraverseHeap`*, которая генерирует файл* `.log` *в формате, поддерживаемом профилировщиком CLR Profiler. Подробнее о командах* `SOS.DLL` *с примерами будет рассказываться в главах 3 и 4. Кроме того, желающие могут ознакомиться со статьей в блоге Саши Голдштейн (Sasha Goldshtein), по адресу: http://blog.sashag.net/archive/2008/04/08/next-generation-production-debugging-demo-2-and-demo-3.aspx, где он приводит пример совместного использования библиотеки* `SOS.DLL` *и профилировщика CLR Profiler.*

Чтобы открыть файл с аварийным дампом памяти в профилировщике .NET Memory Profiler, выберите пункт меню **File → Import memory dump** (Файл → Импортировать дамп памяти) и укажите профилировщику нужный файл. Если имеется несколько файлов дампов, их все можно импортировать в один сеанс анализа и сравнить как снимки динамической памяти. Процесс импорта может занимать довольно продолжительные промежутки времени, особенно для больших дампов памяти. Чтобы ускорить работу с сеансами анализа, в состав SciTech входит отдельный инструмент, *NmpCore.exe*, который можно использовать для сохранения сеанса в промышленном окружении.

На рис. 2.30 показаны результаты сравнения двух дампов памяти в профилировщике .NET Memory Profiler. Он немедленно обнаруживает проблемные объекты и сразу же сообщает, что они удерживаются в памяти обработчиками событий и предлагает выполнить анализ объектов `FileInformation`.

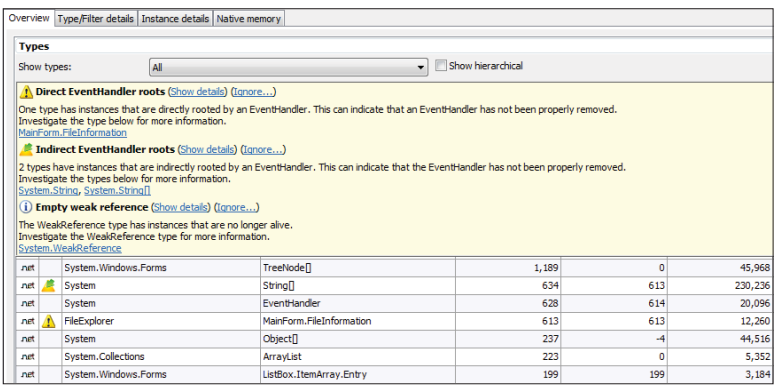

Рис. 2.30. Начальный анализ двух снимков динамической памяти. В пятом столбце списка указано число хранящихся в памяти экземпляров, а в седьмом – количество байтов, занимаемых ими. Основной объем памяти занимают объекты string, информация о которых скрыта здесь за всплывающей подсказкой.

Подробный отчет об объектах `FileInformation` показывает, что все выбранные экземпляры `FileInformation` связаны с обработчиком событий `FileInformation.FileInformationNeedsRefresh` (рис. 2.31), а отчет с информацией об отдельных экземплярах показывает ту же цепочку ссылок, которую мы видели, когда знакомились с профилировщиком ANTS Memory Profiler.

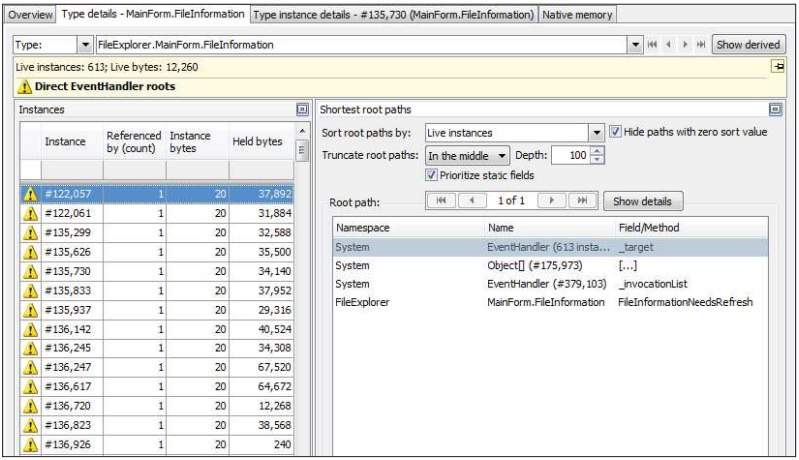

Рис. 2.31. Экземпляры FileInformation.
Колонка «Held bytes» (Занимает байт) указывает,
какой объем памяти занимает каждый экземпляр.
Справа приводится кратчайший путь к выбранному экземпляру.

Мы не будем здесь повторять инструкции по использованию, которые уже давались в описании профилировщика .NET Memory Profiler – отличное руководство по профилировщику SciTech можно найти на сайте проекта: http://memprofiler.com/OnlineDocs/. Этим инструментом завершается наш обзор инструментов и приемов выявления утечек памяти, начатый со знакомства с CLR Profiler.

Другие профилировщики

В этой главе основное внимание было уделено вопросам профилирования нагрузки на CPU, времени и памяти, потому что именно эти характеристики являются наиболее важными с точки зрения оптимизации производительности приложений. Однако существуют и другие характеристики, для измерения которых существуют отдельные инструменты; в этом разделе мы коротко упомянем некоторые из них.

Профилировщики доступа к данным и базам данных

Многие управляемые приложения в своей работе опираются на использование базы данных и немалую часть времени тратят на ожидание результатов запросов или на выполнение массивных изменений. Профилирование операций с базами данных может выполняться в двух направлениях: со стороны приложений, с помощью *профилировщиков доступа к данным*, и со стороны баз данных, с помощью *профилировщиков баз данных*.

Для работы профилировщиков баз данных часто требуется поддержка со стороны производителей, и такие профилировщики обычно используются администраторами баз данных. Мы не будем касаться профилировщиков баз данных; желающие могут поближе познакомиться с профилировщиком SQL Server Profiler, весьма мощным инструментом профилирования баз данных для Microsoft SQL Server, по адресу: http://msdn.microsoft.com/ru-ru/library/ms181091.aspx.

Профилировщики доступа к данным, напротив, предназначены для использования разработчиками приложений. Эти инструменты применяются для исследования производительности слоя доступа к данным в приложениях (Data Access Layer, DAL) и обычно сообщают следующую информацию:

- запросы, выполняемые приложением, и точную трассировку стека для каждой операции;
- список методов приложения, выполняющих операции с базой данных, и список запросов, выполняемых каждым методом;
- предупреждения о неэффективных приемах доступа к базе данных, таких как выполнение запросов, не ограничивающих возвращаемый набор результатов, извлечение всех полей из таблице, когда в приложении используются только некоторые из них, выполнение запросов со слишком большим количеством соединений таблиц или наличие одного запроса на извлечение сущности, связанной с *N* других сущностей, порождающий запросы для каждой из этих связанных сущностей (эта проблема известна под названием «проблема SELECT N + 1»).

Существует несколько коммерческих инструментов профилирования доступа к данным из приложений. Некоторые из них работают только с базами данных определенных типов (такими как Microsoft SQL Server), другие – только с определенными фреймворками доступа к данным (такими как Entity Framework или NHibernate). Ниже перечислены некоторые из них:

- RedGate ANTS Performance Profiler – способен профилировать запросы к Microsoft SQL Server;
- инструмент **Tier Interactions** (Уровневое взаимодействие) в Visual Studio – способен профилировать любые асинхронные операции доступа к данным из ADO.NET – к сожалению, он не поддерживает трассировку стека вызовов для операций с базами данных;
- семейство профилировщиков Hibernating (LINQ в SQL Profiler, Entity Framework Profiler и NHibernate Profiler) – способны профилировать все операции с конкретным фреймворком доступа к данным.

Мы не будем вдаваться в подробное обсуждение этих профилировщиков, но если вас волнует проблема производительности вашего слоя доступа к данным, рассмотрите возможность использования этих инструментов совместно с профилировщиками времени или памяти.

Профилировщики конкуренции

Все возрастающая популярность параллельного программирования подтолкнула разработчиков к созданию специализированных профилировщиков приложений, использующих большое количество потоков выполнения, которые могут выполняться на разных процессорах. В главе 6 мы исследуем некоторые ситуации, в которых немалого прироста производительности легко можно добиться лишь за счет распараллеливания операций, и этот прирост производительности лучше всего может быть оценен с помощью точных инструментов измерения.

Профилировщик Visual Studio, в режимах **Concurrency** (Данные конфликтов ресурсов (параллелизм)) и **Concurrency Visualizer** (Визуализатор параллелизма), использует события ETW для мониторинга производительности многопоточных приложений и предоставляет отчет с несколькими весьма полезными представлениями, упрощающими узкие места, отрицательно сказывающиеся на масштабируемости и производительности. Он имеет два режима работы, как показано ниже.

В режиме **Concurrency** (Данные конфликтов ресурсов (параллелизм)) выявляются ресурсы, такие как управляемые блокировки, на которых потоки выполнения в приложении останавливаются в ожидании их освобождения. Первая часть отчета описывает сами ресурсы

и потоки выполнения, которые были заблокированы на них – она поможет найти и устранить недостатки, мешающие масштабированию (рис.2.32). Другая часть отчета отображает информацию о конфликтах для конкретного потока выполнения, то есть перечень различных механизмов синхронизации, на которых поток выполнения вынужден был простаивать – она поможет уменьшить задержки на пути выполнения определенного потока. Чтобы запустить профилировщик в этом режиме, выберите пункт меню **Analyze → Launch Performance Wizard** (Анализ → Запустить мастер производительности) и затем выберите радиокнопку **Concurrency** (Данные конфликтов ресурсов (параллелизм)).

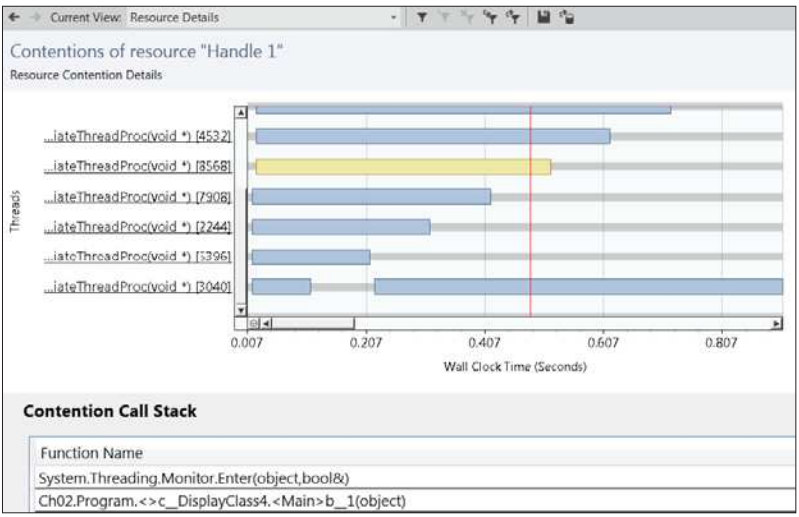

Рис. 2.32. Конфликты для конкретного ресурса – существует несколько потоков выполнения, ожидающих доступа к ресурсу. При выборе потока выполнения, в нижней части отображается его стек вызовов.

В режиме визуализации параллелизма отображается график выполнения всех потоков приложения, где цветом обозначается текущее состояние. Каждый поток имеет несколько переходных состояний – блокировка на операции ввода/вывода, ожидание на механизме синхронизации, выполнение – которые фиксируются профилировщиком, а также стек вызовов (рис. 2.33). Эти отчеты очень удобно использовать, чтобы понять особенности работы потоков выполнения и выявить причины низкой производительности, такие как чрезмерное

количество потоков выполнения, недостаточное количество потоков выполнения и чрезмерное использование механизмов синхронизации. В графике имеется также встроенная поддержка механизмов Task Parallel Library, таких как параллельные циклы (parallel loops) и механизмы синхронизации CLR. Чтобы запустить профилировщик в этом режиме выберите нужный пункт в подменю **Analyze → Concurrency Visualizer** (Анализ → Визуализатор параллелизма).

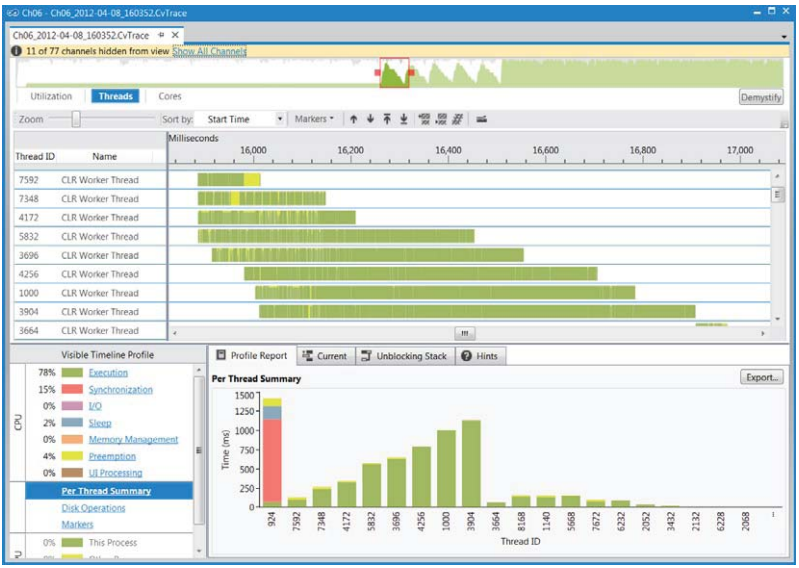

Рис. 2.33. Визуализация работы нескольких потоков выполнения в приложении (перечислены слева). На основании этого отчета можно заключить, что работа распределяется между потоками неравномерно.

> **Примечание.** *На сайте MSDN собрана коллекция антишаблонов программирования многопоточных приложений, выявленных с помощью визуализатора параллелизма, включая колонны блокировок (lock convoy), неравномерное распределение рабочей нагрузки, чрезмерное количество потоков выполнения и другие – найти описания этих антишаблонов можно по адресу: http://msdn.microsoft.com/ru-ru/library/ee329530%28v=vs.110%29.aspx. При проведении собственных испытаний, вы легко сможете идентифицировать похожие проблемы, визуально сравнив имеющиеся у вас графики с графиками на сайте MSDN.*

Профилирование и визуализация работы многопоточных приложений – весьма полезные возможности и мы вернемся к ним в последующих главах. Они могут служить еще одним ярким подтверждением

огромной гибкости механизма ETW – эта вездесущая инфраструктура мониторинга используется самыми разными инструментами профилирования.

Профилировщики ввода/вывода

Последняя характеристика, которую мы рассмотрим в этой главе – это операции ввода/вывода. События ETW могут использоваться для получения информации об операциях доступа к физическому диску, ошибках получения дисковых страниц из памяти, получении сетевых пакетов и других видах ввода/вывода.

Sysinternals Process Monitor – это бесплатный инструмент, выполняющий сбор информации об операциях с файловой системой и с сетью (рис. 2.34). Загрузить полный комплект инструментов Sysinternals или только последнюю версию Process Monitor можно на сайте проекта TechNet, по адресу: http://technet.microsoft.com/ru-ru/sysinternals/bb896645. Используя его богатые возможности фильтрации, системные администраторы и специалисты, занимающиеся вопросами производительности, смогут проанализировать ошибки, связанные с вводом/выводом (такие как отсутствие файлов или недостаточность привилегий), а также проблемы производительности (такие как попытки доступа к удаленной файловой системе или слишком частое вытеснение страниц памяти в файл подкачки).

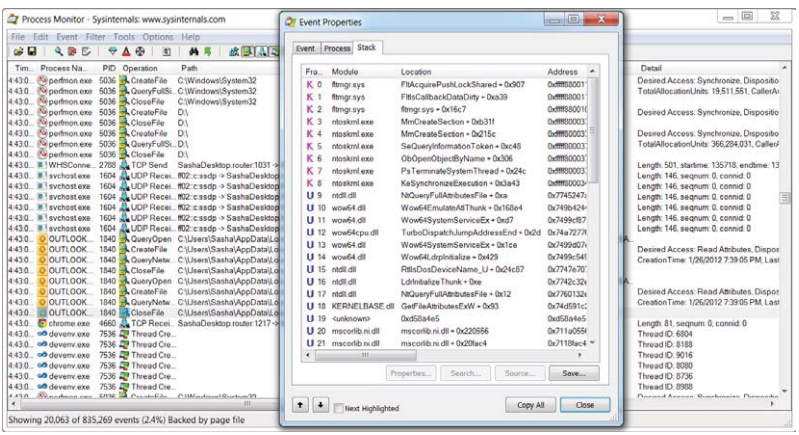

Рис. 2.34. Инструмент Process Monitor отображает в главном представлении события нескольких типов и способен выводить стек вызовов для конкретных событий в отдельном диалоговом окне. Кадры стека с номером 19 и ниже – это управляемые кадры.

Process Monitor поддерживает полную трассировку стека в режиме ядра и в режиме пользовательского приложения для каждого события, что идеально подходит для поиска избыточных или ошибочных операций ввода/вывода и их местоположений в исходном коде приложения. К сожалению, на момент написания этих строк, инструмент Process Monitor не поддерживал управляемые стеки вызовов, но он способен указать хотя бы общее направление поисков проблем в приложениях, выполняющих операции ввода/вывода.

В течение этой главы мы использовали автоматизированные инструменты измерения различных аспектов приложений, связанных с производительностью, – времени выполнения, нагрузки на CPU, операций выделения памяти и даже операций ввода/вывода. Разнообразие приемов измерений достаточно велико, что является одной из причин, почему разработчики часто предпочитают выполнять хронометраж производительности своих приложений вручную. Прежде чем завершить эту главу, мы обсудим приемы такого хронометража и потенциальные ловушки для тех, кто изберет их.

Микрохронометраж

Некоторые проблемы и вопросы производительности могут быть решены только с применением ручных способов измерения. Например, вам может потребоваться обосновать выбор в пользу `StringBuilder`, измерить производительность сторонней библиотеки, оптимизировать сложный алгоритм, разворачивая внутренние циклы или помогая JIT-компилятору поместить часто используемые данные в регистры, – и при этом может оказаться невозможным использовать профилировщики для измерений из-за их медлительности, сложности или слишком высоких накладных расходов. И хотя микрохронометраж часто полон опасностей, он пользуется большой популярностью. Если вы соберетесь использовать его, мы хотим, чтобы вы выполняли его правильно.

Пример неправильного микрохронометража

Для начала рассмотрим пример неправильно спроектированного микрохронометража, и затем будем улучшать его, пока получаемые результаты не приобретут смысл и не станут коррелировать с нашим знанием проблемы.

Цель – определить, что быстрее – применение ключевого слова `is` с последующим приведением к требуемому типу или применение ключевого слова `as` с использованием результата.

```
// Тестовый класс
class Employee {
    public void Work() {}
}

// Фрагмент 1 – выполняет безопасное приведение типа и проверяет
// результат на равенство null
static void Fragment1(object obj) {
    Employee emp = obj as Employee;
    if (emp ! = null) {
        emp.Work();
    }
}

// Фрагмент 2 – сначала проверяет тип, а затем выполняет приведение
static void Fragment2(object obj) {
    if (obj is Employee) {
        Employee emp = obj as Employee;
        emp.Work();
    }
}
```

Следующие строки выполняют простейший хронометраж:

```
static void Main() {
    object obj = new Employee();
    Stopwatch sw = Stopwatch.StartNew();
    for (int i = 0; i < 500; i++) {
        Fragment1(obj);
    }
    Console.WriteLine(sw.ElapsedTicks);
    sw = Stopwatch.StartNew();
    for (int i = 0; i < 500; i++) {
        Fragment2(obj);
    }
    Console.WriteLine(sw.ElapsedTicks);
}
```

Такой способ хронометража дает *ошибочные* результаты, хотя они и воспроизводимы от эксперимента к эксперименту. В большинстве прогонов он показывает, что продолжительность выполнения первого цикла составляет 4 такта, а продолжительность выполнения второго цикла 200–400 тактов. Отсюда можно заключить, что первый фрагмент выполняется в 50–100 раз быстрее. Однако данный способ измерений ошибочен и, соответственно, ошибочен вывод, сделанный на основе полученных результатов:

- цикл выполняется только один раз и 500 итераций недостаточно, чтобы делать далеко идущие выводы – на выполнение теста

требуется совсем немного времени, поэтому на его результаты могут влиять самые разные факторы окружающей среды;
- не было предусмотрено ничего, чтобы предотвратить оптимизацию, поэтому JIT-компилятор мог оптимизировать оба цикла;
- методы `Fragment1` и `Fragment2` измеряют не только стоимость выполнения ключевых слов `is` и `as`, но также стоимость вызова метода (и самого метода `FragmentN`!); однако может так получиться, что стоимость вызова метода окажется намного дороже стоимости измеряемой операции.

Исправим эти проблемы, как показано в следующем фрагменте, который позволяет получить более точную картину стоимости обеих операций:

```
class Employee {
    // Предотвратить оптимизацию этого метода JIT-компилятором
    [MethodImpl(MethodImplOptions.NoInlining)]
    public void Work() {}
}

static void Measure(object obj) {
    const int OUTER_ITERATIONS = 10;
    const int INNER_ITERATIONS = 100000000;
    // Внешний цикл повторяется столько раз, сколько нужно,
    // чтобы гарантировать надежность результатов
    for (int i = 0; i < OUTER_ITERATIONS; ++i) {
        Stopwatch sw = Stopwatch.StartNew();
        // Внутренний измерительный цикл повторяется столько раз,
        // чтобы гарантировать определенную продолжительность
        // выполнения операции
        for (int j = 0; j < INNER_ITERATIONS; ++j) {
            Employee emp = obj as Employee;
            if (emp ! = null)
                emp.Work();
        }
        Console.WriteLine("As - {0}ms", sw.ElapsedMilliseconds);
    }
    for (int i = 0; i < OUTER_ITERATIONS; ++i) {
        Stopwatch sw = Stopwatch.StartNew();
        for (int j = 0; j < INNER_ITERATIONS; ++j) {
            if (obj is Employee) {
                Employee emp = obj as Employee;
                emp.Work();
            }
        }
        Console.WriteLine("Is Then As - {0}ms", sw.ElapsedMilliseconds);
    }
}
```

На тестовой машине одного из авторов (после отброса первой итерации) продолжительность первого цикла составила около 410 мсек и для второго около 440 мсек. Разница в скорости выполнения надежно воспроизводилась от эксперимента к эксперименту, что могло бы служить основанием говорить о более высокой производительности ключевого слова `as`.

Однако загадки на этом не закончились. Если добавить к методу `Work` модификатор `virtual`, различия в производительности полностью исчезнут, даже при большом количестве повторений. Такое положение нельзя объяснить достоинствами или недостатками нашего способа измерений, это обусловлено особенностями предметной области. Мы не сможем объяснить такое поведение, не погрузившись на уровень машинного кода и не исследовав реализации обоих циклов, сгенерированные JIT-компилятором. Посмотрим на цикл до добавления модификатора `virtual`:

```
; Дизассемблированное тело первого цикла
mov edx,ebx
mov ecx,163780h (MT: Employee)
call clr!JIT_IsInstanceOfClass (705ecfaa)
test eax,eax
je WRONG_TYPE
mov ecx,eax
call dword ptr ds:[163774h] (Employee.Work(), mdToken: 06000001)
WRONG_TYPE:

; Дизассемблированное тело второго цикла
mov edx,ebx
mov ecx,163780h (MT: Employee)
call clr!JIT_IsInstanceOfClass (705ecfaa)
test eax,eax
je WRONG_TYPE
mov ecx,ebx
cmp dword ptr [ecx],ecx
call dword ptr ds:[163774h] (Employee.Work(), mdToken: 06000001)
WRONG_TYPE:
```

В главе 3 мы более подробно обсудим последовательности машинных инструкций, генерируемых JIT-компилятором для вызова виртуальных и невиртуальных методов. Когда вызывается невиртуальный метод, JIT-компилятор должен сгенерировать инструкции, проверяющие, что вызов не будет выполнен по нулевому адресу. Инструкция `CMP` во втором цикле решает эту задачу. В первом цикле JIT-компилятор оптимизирует эту проверку, удаляя ее за ненадобностью, потому что проверка результата приведения типа выполняется непосредственно перед вызовом (`if (emp != null) ...`). Во втором цикле,

эвристический алгоритм оптимизации JIT-компилятора не может принять решение об удалении проверки (хотя это вполне безопасно), и эта лишняя инструкция как раз и дает те самые 7–8% накладных расходов.

Однако, после добавления модификатора `virtual` JIT-компилятор генерирует абсолютно одинаковый код в обоих циклах:

```
; Дизассемблированное тело обоих циклов
mov edx,ebx
mov ecx,1A3794h (MT: Employee)
call clr!JIT_IsInstanceOfClass (6b24cfaa)
test eax,eax
je WRONG_TYPE
mov ecx,eax
mov eax,dword ptr [ecx]
mov eax,dword ptr [eax + 28h]
call dword ptr [eax + 10h]
WRONG_TYPE:
```

Причина заключается в том, что когда вызывается виртуальный метод, отпадает необходимость явно проверять ссылку – она неявно выполняется последовательностью инструкций выбора метода (как будет показано в главе 3). Когда оба цикла оказываются идентичными, и результаты хронометража тоже оказываются идентичными.

Рекомендации по проведению хронометража

Для успешного проведения хронометража старайтесь придерживаться следующих рекомендаций.

- Тестирование должно выполняться в среде, близкой по своим характеристикам окружению, в котором должно работать разрабатываемое приложение. Например, не следует производить хронометраж метода с коллекциями данных, находящимися в памяти, если в действительности он должен обрабатывать таблицы в базе данных.
- Тестовые исходные данные по своей структуре должны быть близки фактическим данным. Например, не следует измерять быстродействие сортировки списка с тремя элементами, если в действительности должна выполняться сортировка списков, содержащих миллионы элементов.
- Время выполнения кода поддержки, используемого для настройки окружения, должно быть ничтожно мало, по сравнению со временем выполнения тестируемого кода. Если это

невозможно, тогда настройка должна выполняться один раз, а тестируемый код – много раз.

- Тестируемый код должен выполняться достаточно долго, чтобы ослабить влияние случайных программных и аппаратных флуктуаций. Например, при измерении накладных расходов, связанных с упаковкой значений простых типов в экземпляры классов, одной операции будет явно недостаточно, чтобы получить значимые результаты, и необходимо выполнить большое количество итераций.
- Тестируемый код не должен оптимизироваться компилятором языка или JIT-компилятором. Такие оптимизации часто выполняются при компиляции в режиме «Release». (Мы еще вернемся к этой теме ниже.)

Когда тестируемый код будет полностью готов и достаточно надежно измеряет именно те характеристики, для измерения которых он предназначался, необходимо потратить некоторое время на подготовки окружения для хронометража.

- В процессе выполнения хронометража не должны запускаться никакие другие приложения. Сетевые операции, операции с файлами или другие типы внешней активности должны быть минимизированы (например, отключением сетевой карты или остановкой ненужных служб).
- При проведении хронометража кода, создающего большое количество объектов, желательно максимально уменьшить влияние сборщика мусора. Желательно, чтобы сборка мусора выполнялась до и после критически важных итераций, чтобы уменьшить их взаимовлияние.
- Аппаратное обеспечение тестовой системы по своим характеристикам должно быть похоже на аппаратное обеспечение промышленной системы. Например, тесты, интенсивно выполняющие дисковые операции, связанные с перемещением головок по диску, будут выполняться намного быстрее на твердотельном накопителе, чем на обычном механическом жестком диске. (То же относится к графическим картам, процессорам со специфическими возможностями, такими как поддержка набора инструкций SIMD, архитектуре памяти и другим аппаратным особенностям.)

Наконец, необходимо особое внимание уделить коду, реализующему тестирование. Ниже приводится несколько правил, которые следует помнить.

- Результаты первого измерения должны отбрасываться – оно часто отягощено накладными расходами, связанными с работой JIT-компилятора и выполнением других начальных операций. Кроме того, маловероятно, что перед выполнением первой итерации данные и инструкции окажутся в кеше процессора. (Если целью тестирования является выявление эффекта влияния кеша, это правило не должно учитываться.)
- Измерения должны повторяться много раз, при этом в качестве результата следует рассматривать не только среднее значение, но также стандартное отклонение (позволяющее оценить разброс результатов относительно среднего значения) и флуктуации между сеансами тестирования.
- Накладные расходы на выполнение измерительного цикла должны вычитаться из общих результатов хронометража. Для этого следует выполнить хронометраж пустого цикла, что не так просто, как кажется, потому что JIT-компилятор часто оптимизирует пустые циклы, удаляя их. (Одно из решений состоит в том, чтобы вручную написать пустой цикл на ассемблере.)
- Накладные расходы на выполнение измерений должны вычитаться из общих результатов. А сами измерения должны производиться наименее дорогостоящим и наиболее точным способом из доступных – обычно для этой цели можно использовать `System.Diagnostics.Stopwatch.Download`.
- Вы должны точно знать разрешающую способность и точность измерительного инструмента, используемого в хронометраже. Например, точность `Environment.TickCount` составляет всего 10-15 мсек, хотя на первый взгляд кажется, что он имеет точность 1 мсек.

Примечание. *Разрешение – это минимальный интервал времени, различаемый механизмом измерения. Если в документации сообщается, что он возвращает результат в виде целого числа, кратного 100 нсек, его разрешающая способность составляет 100 нсек. Однако его точность может быть значительно меньше – после измерения фактического интервала времени 500 нсек он может в одном случае вернуть значение 2 × 100 нсек, а в другом 7 × 100 нсек. В этой ситуации мы могли бы принять за верхнюю границу точности 300 нсек. Наконец, точность определяет погрешность механизма измерений. Если физический интервал 5000 нсек от испытания к испытанию измеряется как 5400 нсек с точностью до 100 нсек, можно говорить, что погрешность составляет +8%.*

Неудачный пример в начале этого раздела был приведен не для того, чтобы отпугнуть вас от мысли реализовать собственное тести-

рование. Однако вы должны помнить рекомендации, данные выше, и проектировать реализацию хронометража так, чтобы его результатам можно было доверять. Нет ничего хуже, когда попытки оптимизации основываются на ошибочных измерениях, и к сожалению ручное тестирование часто ведет в эту ловушку.

В заключение

Измерение производительности – непростая задача, что является одной из причин такого разнообразия измеряемых характеристик и инструментов, а также влияния инструментов на точность измерений и поведение приложений. Мы познакомились в этой главе с большим количеством инструментов, и, возможно, вы не сразу сможете ответить на вопрос – какой из них следует использовать в той или иной ситуации. Поэтому в табл. 2.3 мы перечислили наиболее важные характеристики всех инструментов, рассматривавшихся здесь.

Таблица 2.3. Инструменты измерения производительности, представленные в этой главе

Инструмент	Измеряемые характеристики	Накладные расходы	Достоинства и недостатки
Дискретный профилировщик Visual Studio	Нагрузка на CPU, промахи кеша, ошибки обращения к страницам, системные вызовы	Низкие	–
Инструментированный профилировщик Visual Studio	Время выполнения	Средние	Не поддерживает возможность подключения к действующему процессу
Профилировщик выделения памяти Visual Studio	Операции выделения памяти	Средние	–
Визуализатор параллелизма Visual Studio	Визуализация графика работы потоков выполнения, конфликты при обращении к ресурсам	Низкие	Графическое представление информации работе потоков выполнения, подробные сведения о возникающих конфликтах, разблокировка стека

Таблица 2.3. (окончание)

Инструмент	Измеряемые характеристики	Накладные расходы	Достоинства и недостатки
CLR Profiler	Операции выделения памяти, статистика работы сборщика мусора, ссылки на объекты	Высокие	Графическое представление распределения динамической памяти, график работы сборщика мусора
Performance Monitor	Числовые характеристики производительности на уровне процесса и системы	Отсутствуют	Только числовая информация, не поддерживаются измерения на уровне методов
BCL PerfMonitor	Время выполнения, информация о работе сборщика мусора, информация о работе JIT-компилятора	Очень низкие	Простота профилирования, практически полное отсутствие накладных расходов
PerfView	Время выполнения, информация о распределении динамической памяти, информация о работе сборщика мусора, информация о работе JIT-компилятора	Очень низкие	Вдобавок к возможностям PerfMonitor позволяет выполнять анализ распределения динамической памяти
Windows Performance Toolkit	События ETW от провайдеров системы и приложения	Очень низкие	–
Process Monitor	Операции ввода/вывода с файлами, реестром и сетью	Низкие	–
Entity Framework Profiler	Доступ к данным с применением классов фреймворка Entity Framework	Средние	–
ANTS Memory Profiler	Информация об использовании памяти	Средние	Большое разнообразие фильтров и способов визуализации
.NET Memory Profiler	Информация об использовании памяти	Средние	Может открывать аварийные дампы памяти

Вооруженные этими инструментами и общим пониманием характеристик управляемых приложений, мы готовы погрузиться во внутренние механизмы CLR и познакомится с практическими приемами, которые можно применить для повышения производительности управляемых приложений.

ГЛАВА 3. Внутреннее устройство типов

В этой главе рассказывается о внутреннем устройстве типов .NET, как типы значений и ссылочные типы размещаются в памяти, какой код генерирует JIT-компилятор для вызова виртуальных методов, о хитросплетениях реализации типов значений и других тонкостях. Почему мы решили остановиться на теме внутреннего устройства типов и потратить на ее обсуждение несколько десятков страниц? Как эта информация может помочь в увеличении производительности приложений? Как оказывается, типы значений и ссылочные типы отличаются структурой, способом размещения в памяти, сравнения, присваивания и имеют массу других отличий, что делает правильность выбора типов весьма важным фактором, влияющим на производительность приложения.

Пример

Рассмотрим простой тип `Point2D`, представляющий координаты точки в ограниченном двумерном пространстве. Каждая из двух координат может быть представлена значением типа `short`, а весь объект будет занимать четыре байта. Теперь допустим, что требуется сохранить в памяти массив с десятью миллионами точек. Какой объем памяти потребуется для этого? Ответ на этот вопрос в значительной степени зависит от того, является ли тип `Point2D` ссылочным типом или типом значения. Если это ссылочный тип, массив с десятью миллионами точек будет хранить десять миллионов ссылок. В 32-разрядной системе для хранения десяти миллионов ссылок потребуется почти 40 Мбайт памяти. Сами объекты займут почти такой же объем. Фактически, как будет показано чуть ниже, каждый экземпляр `Point2D` занимает не менее 12 байт памяти, что в сумме для десяти миллионов точек дает

160 Мбайт памяти! С другой стороны, если `Point2D` является типом значения, массив с десятью миллионами точек будет хранить десять миллионов точек и ни байтом больше, что в сумме составит 40 Мбайт, то есть в четыре раза меньше, чем для ссылочного типа (рис. 3.1). Такое различие в *плотности размещения информации в памяти* играет важную роль в выборе типов значений в определенных ситуациях.

> **Примечание.** *Существует еще один недостаток ссылочных типов, в сравнении с типами значениями. Если необходимо будет выполнить обход элементов такого огромного массива точек, для компилятора и аппаратной части компьютера проще будет обеспечить доступ к смежным ячейкам памяти с объектами* `Point2D`*, чем доступ через ссылки к объектам в динамической памяти, которые необязательно будут располагаться подряд. Как будет показано в главе 5, эффективное использование кеша процессора может поднять производительность приложения на порядок.*

Совершенно очевидно, что знание и понимание особенностей размещения объектов CLR в памяти, а также и отличий между ссылочными типами и типами значений очень важно для достижения высокой производительности приложений. Мы начинаем эту главу со знакомства с основными различиями между типами значений и ссылочными типами на уровне языка, а затем погрузимся в изучение особенностей их внутреннего строения.

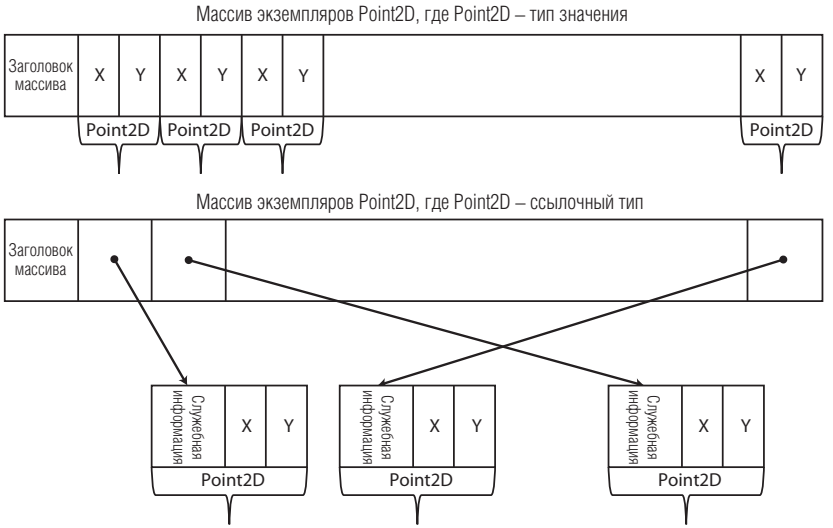

Рис. 3.1. Массив экземпляров Point2D в случаях, когда Point2D является ссылочным типом и типом значения.

Семантические отличия между ссылочными типами и типами значений

К ссылочным типам в .NET относятся классы, делегаты, интерфейсы и массивы. Тип `string` (`System.String`), один из самых вездесущих типов в .NET, также является ссылочным. К типам значений в .NET относятся структуры и перечисления. Простые типы, такие как `int`, `float`, `decimal` тоже являются типами значений, но разработчики приложений для .NET свободно могут определять собственные типы значений с помощью ключевого слова `struct`.

На уровне языка, ссылочные типы обладают семантикой ссылок, в соответствии с которой главенствующее положение имеет идентичность объектов и только потом их содержимое, тогда как типы значений обладают семантикой значений, в соответствии с которой объекты не имеют идентичности, недоступны через ссылки и интерпретируются в зависимости от их содержимого. Это находит отражение в нескольких областях языков .NET, как показано в табл. 3.1.

Таблица 3.1. Семантические отличия между ссылочными типами и типами значений

Критерий	Ссылочные типы	Типы значений
Передача объекта в вызов метода	Передается только ссылка; изменения в объекте отразятся на всех остальных ссылках на него	В параметр копируется содержимое объекта (если не используется ключевое слово `ref` или `out`); после выхода из метода изменения теряются
Присваивание одной переменной – другой	Копируется только ссылка; после присваивания обе переменные содержат ссылку на один и тот же объект	Копируется содержимое; две переменные содержат идентичные копии никак не связанных между собой данных
Сравнение двух объектов с помощью оператора ==	Сравниваются ссылки; две ссылки считаются равными, если ссылаются на один и тот же объект	Сравнивается содержимое; два объекта считаются равными, если их содержимое идентично

Эти семантические отличия определяют, как должен писаться программный код на любом из языков .NET. Однако это лишь вершина айсберга различий между ссылочными типами и типами значений. Рассмотрим сначала области памяти, где хранятся объекты, а также – как выделяется память для них и как она освобождается.

Хранение, размещение и удаление

Ссылочные типы хранятся исключительно в управляемой куче (динамической памяти) – области памяти, управляемой сборщиком мусора .NET, о котором подробно рассказывается в главе 4. Размещение объекта в управляемой динамической памяти влечет увеличение указателя, что является весьма недорогой операцией с точки зрения производительности. В многопроцессорных системах, когда код, выполняемый на разных процессорах, обращается к одной и той же динамической памяти требуется предусмотреть некоторую синхронизацию, но процедура выделения памяти все равно остается чрезвычайно дешевой, в сравнении с процедурами в неуправляемых окружениях, таких как `malloc`.

Сборщик мусора освобождает память недетерминированным способом и не дает никаких гарантий, касающихся работы его внутренних механизмов. Как будет показано в главе 4, полный процесс сборки мусора является довольно дорогостоящим, но средняя его цена в приложении, корректно использующем динамическую память, относительно невысока, если сравнивать с аналогичными механизмами в неуправляемых окружениях.

> **Примечание.** *Чтобы быть до конца точными, следует заметить, что ссылочные типы могут размещаться на стеке. Массивы значений простых типов (например, массивы целых чисел) могут размещаться на стеке при использовании контекста* `unsafe` *и ключевого слова* `stackalloc`*, или встраиванием массива фиксированного размера в собственную структуру с помощью ключевого слова* `fixed` *(обсуждается в главе 8). Однако объекты, созданные ключевыми словами* `stackalloc` *и* `fixed`*, не являются «настоящими» массивами и располагаются в памяти иначе, чем стандартные массивы.*

Автономные типы значений обычно размещаются на стеке потока выполнения. Однако типы значений могут включаться в состав ссылочных типов – в этом случае они будут храниться в динамической памяти и могут упаковываться в классы, при передаче их содержимого в динамическую память (тема упаковки будет рассматриваться ниже в этой главе). Размещение экземпляра типа значения на стеке – очень недорогая операция, которая выражается лишь в увеличении регистра указателя стека (`ESP` – в процессорах Intel x86), и имеет дополнительное преимущество, позволяя выделять память сразу для нескольких объектов. Фактически, для кода реализации *пролога метода* довольно типично единственной машинной инструкцией выде-

лять память сразу для всех локальных переменных, объявленных в охватывающем блоке.

Освобождение памяти на стеке также выполняется весьма эффективно и требует лишь восстановления прежнего значения регистра указателя стека. Из-за особенностей компиляции методов в машинный код, компилятору часто даже не требуется запоминать общий объем, занимаемый локальными переменными метода, и он может уничтожить весь кадр стека тремя стандартными инструкциями, известными как *эпилог метода*.

Ниже приводится типичный пролог и эпилог управляемого метода, скомпилированного в 32-разрядный машинный код (в действительности JIT-компилятор производит несколько иной код, производя множество оптимизаций, как описывается в главе 10). Метод имеет четыре локальные переменные, место для которых выделяется в прологе и освобождается в эпилоге:

```
int Calculation(int a, int b)
{
    int x = a + b;
    int y = a - b;
    int z = b - a;
    int w = 2 * b + 2 * a;
    return x + y + z + w;
}

; параметры передаются на стеке по адресам [esp+4] и [esp+8]
push ebp
mov ebp, esp
add esp, 16 ; выделяется место для четырех локальных переменных
mov eax, dword ptr [ebp+8]
add eax, dword ptr [ebp+12]
mov dword ptr [ebp-4], eax
; ...аналогичные манипуляции с переменными y, z, w
mov eax, dword ptr [ebp-4]
add eax, dword ptr [ebp-8]
add eax, dword ptr [ebp-12]
add eax, dword ptr [ebp-16] ; eax содержит возвращаемое значение
mov esp, ebp ; восстанавливает кадр стека – освобождает память
             ; локальных переменных
pop ebp
ret 8 ; освобождает память, занимаемую двумя параметрами
```

Примечание. *Ключевое слово* `new` *в C# и других управляемых языках не означает, что место будет выделено в динамической области. Память для типа значения может быть выделена и на стеке. Например, следующая строка разместит экземпляр* `DateTime` *на стеке, инициализировав его датой, предшествующей Новому Году (*`System.DateTime` *– это тип значения):* `DateTime newYear = new DateTime(2011, 12, 31);`

Чем отличаются динамическая память и стек?

Вопреки общепринятому мнению, между стеком и динамической памятью в процессах .NET не так много отличий. Стек и динамическая память – не более чем области адресов в виртуальной памяти и потому адреса, зарезервированные для стека определенного потока выполнения, не обладают никакими преимуществами перед адресами, зарезервированными для динамической памяти. Доступ к динамической памяти выполняется ни быстрее, ни медленнее, чем доступ к стеку. Однако, в некоторых ситуациях использование памяти на стеке может повысить производительность в целом. Это объясняется следующими причинами.

- Временные локальные значения (размещаемые близко друг от друга в смысле времени) на стеке сохраняются рядом друг с другом (в смысле адресов в памяти), а последовательное хранение на стеке обычно лучше соответствует особенностям работы кеша процессора и механизму страничной памяти системы.
- Плотность размещения на стеке обычно выше, чем в динамической памяти, из-за отсутствия накладных расходов, свойственных ссылочным типам (обсуждаются далее в этой главе). Более высокая плотность размещения в памяти обычно означает более высокую производительность, например, потому что в кеш процессора попадет больше объектов.
- Объем стека потока выполнения обычно очень невелик – в Windows по умолчанию максимальный размер стека составляет 1 Мбайт, а большинство потоков выполнения используют и того меньше – лишь несколько страниц. В современных системах стеки всех потоков в приложении могут уместиться в кеш процессора CPU, что обеспечивает чрезвычайно высокую скорость доступа к объектам, хранящимся в стеке. (Динамическая память, напротив, редко когда целиком может уместиться в кеше процессора.)

Однако, несмотря на сказанное выше, не следует стремиться все и вся размещать на стеке! Объем стека потока выполнения в Windows ограничен и его легко исчерпать в случае слишком глубокой рекурсии или попытавшись разместить на стеке слишком много объектов.

Разобравшись с различиями между типами значений и ссылочными типами, лежащими на поверхности, можно перейти к исследованию их внутренней реализации, также объясняющей огромные отличия в плотности размещения в памяти, уже несколько раз упоминавшейся выше. Но, прежде чем мы приступим, необходимо предупредить вас, что особенности, описываемые ниже, являются особенностями внутренней реализации CLR, которые могут измениться в любой момент без каких-либо уведомлений. Мы сделали все возможное, чтобы гарантировать полное соответствие информации версии .NET 4.5, но мы не можем обещать, что она останется верной в будущих версиях.

Внутреннее устройство ссылочных типов

Сначала мы разберемся со ссылочными типами, имеющих весьма сложную организацию в памяти, что оказывает существенное влияние на производительность. В дальнейшем обсуждении мы будем рассматривать пример ссылочного типа `Employee`, имеющего несколько полей (экземпляра и статических), а также несколько методов:

```
public class Employee
{
    private int _id;
    private string _name;
    private static CompanyPolicy _policy;
    public virtual void Work() {
        Console.WriteLine("Zzzz...");
    }
    public void TakeVacation(int days) {
        Console.WriteLine("Zzzz...");
    }
    public static void SetCompanyPolicy(CompanyPolicy policy) {
        _policy = policy;
    }
}
```

Теперь посмотрим, как располагается экземпляр ссылочного типа `Employee` в динамической памяти. На рис. представлена схема размещения объекта в 32-разрядном процессе .NET:

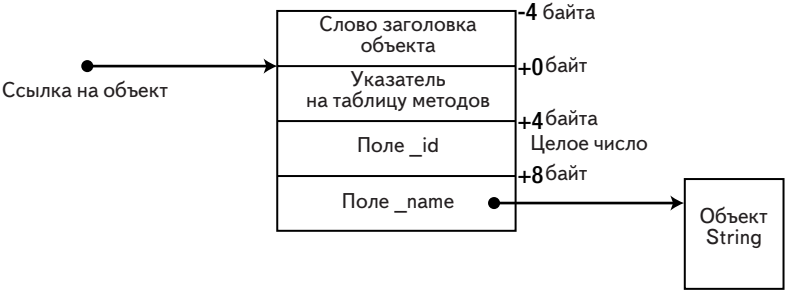

Рис. 3.2. Схема размещения экземпляра Employee в управляемой динамической памяти, включая служебную информацию.

Порядок следование полей `_id` и `_name` внутри объекта не обязательно будет таким, как на рис. 3.3 (хотя есть возможность управлять им с помощью атрибута `StructLayout`, как будет показано в разделе «Внутреннее устройство типов значений»). В начале области памяти,

занимаемой объектом, находятся четыре байта *слова заголовка объекта* (или индекса блока синхронизации), за которыми следуют четыре байта *указателя на таблицу методов* (или указатель на тип объекта). Эти поля недоступны непосредственно ни в одном из языков .NET – они используются JIT-компилятором и средой выполнения CLR. Ссылка на объект (которая фактически является всего лишь адресом в памяти) указывает на указатель таблицы методов, поэтому слово заголовка объекта находится с отрицательным смещением относительно адреса объекта.

> **Примечание.** *В 32-разрядных системах, объекты размещаются в динамической памяти по адресам, кратным 4. Это означает, что объект с единственным полем, размером 1 байт, все равно будет занимать 12 байт в памяти, из-за выравнивания (фактически, даже экземпляр класса, не имеющего ни одного поля, будет занимать 12 байт). В 64-разрядных системах имеются свои отличия. Во-первых, указатель на таблицу методов занимает 8 байт, и слово заголовка объекта так же занимает 8 байт. Во-вторых, объекты размещаются в динамической памяти по адресам, кратным 8. Это означает, что объект с единственным полем, размером 1 байт, будет занимать 24 байта в памяти. Эти цифры приведены, только чтобы показать, насколько увеличиваются накладные расходы при массовом размещении маленьких объектов ссылочных типов.*

Таблица методов

Поле указателя на таблицу методов ссылается на внутреннюю структуру CLR под названием *таблица методов* (method table), которая в свою очередь ссылается на другую внутреннюю структуру под названием EEClass (где EE, это аббревиатура от Execution Engine – механизм выполнения). Вместе, таблица методов и EEClass, содержат информацию, необходимую для выбора виртуального метода, метода интерфейса, статической переменной, определения типа объекта во время выполнения, доступа к методам базового класса и многих других целей. Таблица методов содержит часто используемую информацию, требуемую для выполнения операций такими механизмами, как механизм выбора виртуального метода, а структура EEClass содержит информацию, используемую реже, например механизмом рефлексии. Ознакомиться с содержимым обеих структур данных можно с помощью команд `!DumpMT` и `!DumpClass` библиотеки SOS, в исходных текстах Rotor реализации .NET (Shared Source Common Language Infrastructure, SSCLI), но имейте в виду, что рассматриваемые здесь детали внутренней реализации могут существенно отличаться в разных версиях CLR.

Примечание. *SOS (Son of Strike) – это DLL-библиотека, расширение отладчика, упрощающая отладку управляемых приложений с применением отладчиков Windows. Чаще всего используется совместно с WinDbg, но может также загружаться в Visual Studio с помощью «Immediate Window» (Окно команд). Команды, поддерживаемые расширением, позволяют проникнуть вглубь CLR, и именно поэтому мы часто будем использовать их в данной главе. За дополнительной информацией о библиотеке SOS обращайтесь к встроенной документации (команда* `!help`*, доступная после загрузки расширения) и к документации на сайте MSDN. Отличное описание возможностей SOS применительно к отладке управляемых приложений, можно найти в книге Марио Хъювардт (Mario Hewardt) «Advanced .NET Debugging» (Addison-Wesley, 2009).*

Размещение статических полей определяется структурой EEClass. Поля простых типов (например, целые числа) сохраняются в динамически выделенных областях, в памяти загрузчика, а экземпляры пользовательских ссылочных типов и типов значений сохраняются в виде косвенных ссылок на области в динамической памяти (через массив объектов `AppDomain`). Чтобы получить доступ к статическому полю, необязательно обращаться к таблице методов или к структуре EEClass – JIT-компилятор «жестко зашивает» адреса статических полей в генерируемый им машинный код. Массив ссылок на статические поля фиксируется так, что его адрес не может быть изменен сборщиком мусора (о котором рассказывается в главе 4), кроме того, статические поля простых типов размещаются внутри таблицы методов, которая не затрагивается сборщиком мусора. Это гарантирует, что для доступа к таким полям можно без опаски использовать жестко зашитые адреса:

```
public static void SetCompanyPolicy(CompanyPolicy policy)
{
    _policy = policy;
}
```

```
mov ecx, dword ptr [ebp+8]     ; копировать параметр в ECX
mov dword ptr [0x3543320], ecx ; копировать ECX в статическое
                               ; поле в глобальном массиве
```

Наиболее очевидной информацией, хранимой в таблице методов, является массив адресов, по одному для каждого метода, включая виртуальные методы, унаследованные от базового типа. Например, на рис. 3.3 показана возможная схема размещения в памяти экземпляра класса `Employee`, представленного выше, при этом предполагается, что он наследует только класс `System.Object`:

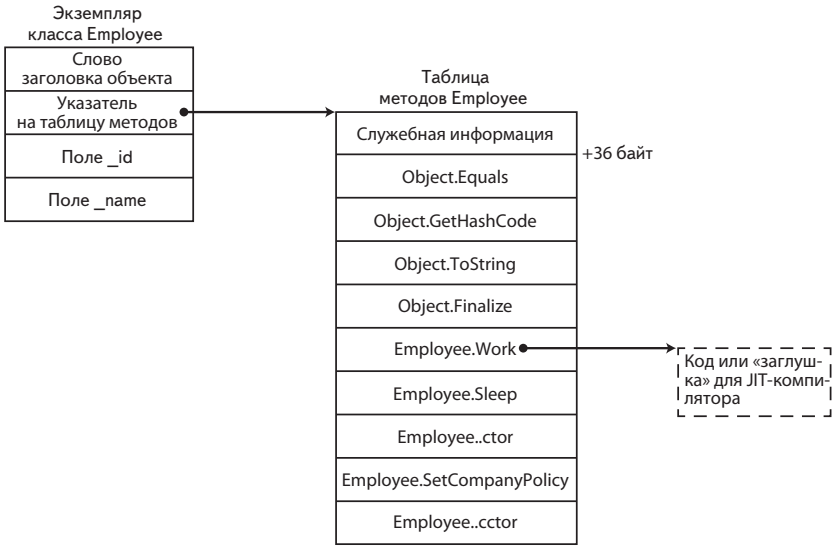

Рис. 3.3. Таблица методов класса Employee (приводится частично).

Для исследования таблиц методов можно использовать команду `!DumpMT` библиотеки SOS, применив ее к указателю на таблицу методов (его можно получить из первого поля имеющегося объекта, на который указывает ссылка на объект, или с помощью команды `!Name2EE`). Ключ `-md` выведет таблицу дескрипторов методов, содержащую адреса и дескрипторы методов. (Колонка `JIT` может иметь три значения: `PreJIT`, если метод скомпилирован с помощью NGEN; `JIT`, если метод скомпилирован JIT-компилятором во время выполнения; или `NONE`, если метод еще не был скомпилирован.)

```
0:000> r esi
esi=02774ec8
0:000> !do esi
Name:        CompanyPolicy
MethodTable: 002a3828
EEClass:     002a1350
Size:        12(0xc) bytes
File:        D:\Development\...\App.exe
Fields:
None
0:000> dd esi L1
02774ec8 002a3828
0:000> !dumpmt -md 002a3828
EEClass:          002a1350
```

```
Module:           002a2e7c
Name:             CompanyPolicy
mdToken:          02000002
File:             D:\Development\...\App.exe
BaseSize:         0xc
ComponentSize:    0x0
Slots in VTable:  5
Number of IFaces in IFaceMap: 0
--------------------------------------
MethodDesc Table
   Entry MethodDe    JIT Name
5b625450 5b3c3524 PreJIT System.Object.ToString()
5b6106b0 5b3c352c PreJIT System.Object.Equals(System.Object)
5b610270 5b3c354c PreJIT System.Object.GetHashCode()
5b610230 5b3c3560 PreJIT System.Object.Finalize()
002ac058 002a3820   NONE CompanyPolicy..ctor()
```

Примечание. *В отличие от таблиц указателей на виртуальные функции в языке C++, таблицы методов в CLR содержат адреса для всех методов, включая и невиртуальные. Порядок следования методов в таблице не определен. В настоящее время они располагаются в следующем порядке: унаследованные виртуальные методы (включая переопределяющие их – обсуждается ниже), новые виртуальные методы, невиртуальные методы экземпляра и статические методы.*

Адреса в таблице методов генерируются «на лету» – JIT-компилятор компилирует методы при первом обращении к ним, если они не были скомпилированы инструментом NGEN (обсуждается в главе 10). Однако, пользователям таблицы методов нет нужды знать такие тонкости, благодаря распространенной особенности компилятора. В момент создания таблицы методов, она заполняется специальными «заглушками», содержащими единственную инструкцию `CALL`, вызывающую процедуру компиляции соответствующего метода. После компиляции заглушка затирается инструкцией `JMP`, передающей управление вновь скомпилированному методу. Вся структура данных, хранящая заглушку и некоторую дополнительную информацию о методе, называется дескриптором метода (method descriptor, MD) и доступна для исследования с помощью команды `!DumpMD` библиотеки SOS.

До того, как метод будет скомпилирован JIT-компилятором, его дескриптор содержит следующую информацию:

```
0:000> !dumpmd 003737a8
Method Name:  Employee.Sleep()
Class:        003712fc
MethodTable:  003737c8
mdToken:      06000003
```

```
Module:       00372e7c
IsJitted:     no
CodeAddr:     ffffffff
Transparency: Critical
```

Ниже приводится пример заглушки, ответственной за обновление дескриптора метода:

```
0:000> !u 002ac035
Unmanaged code
002ac035 b002        mov    al,2
002ac037 eb08        jmp    002ac041
002ac039 b005        mov    al,5
002ac03b eb04        jmp    002ac041
002ac03d b008        mov    al,8
002ac03f eb00        jmp    002ac041
002ac041 0fb6c0      movzx  eax,al
002ac044 c1e002      shl    eax,2
002ac047 05a0372a00  add    eax,2A37A0h
002ac04c e98270ca66  jmp    clr!ThePreStub (66f530d3)
```

После компиляции метода, его дескриптор принимает следующий вид:

```
0:007> !dumpmd 003737a8
Method Name:   Employee.Sleep()
Class:         003712fc
MethodTable:   003737c8
mdToken:       06000003
Module:        00372e7c
IsJitted:      yes
CodeAddr:      00490140
Transparency:  Critical
```

В действительности таблица методов содержит больше информации, чем было показано выше. Описание некоторых дополнительных полей, играющих важную роль в выборе метода при вызове, обсуждается далее, – именно по этой причине нам необходимо внимательнее рассмотреть структуру таблицы методов экземпляра класса `Employee`. Допустим, что класс `Employee` дополнительно реализует три интерфейса: `IComparable`, `IDisposable` и `ICloneable`.

На рис. 3.4 изображено несколько дополнительных деталей в таблице методов. Во-первых, заголовок таблицы методов содержит несколько интересных флагов, позволяющих динамически исследовать ее структуру, в том числе: количество виртуальных методов и количество реализованных интерфейсов. Во-вторых, таблица методов содержит указатель на таблицу методов базового класса, указатель на модуль и указатель на структуру EEClass (которая содержит обратную ссылку на таблицу методов). В-третьих, фактическим мето-

дам предшествует список таблиц методов интерфейсов, реализуемых классом. Именно поэтому указатель на список методов в таблице методов находится со смещением 40 байт от начала таблицы методов.

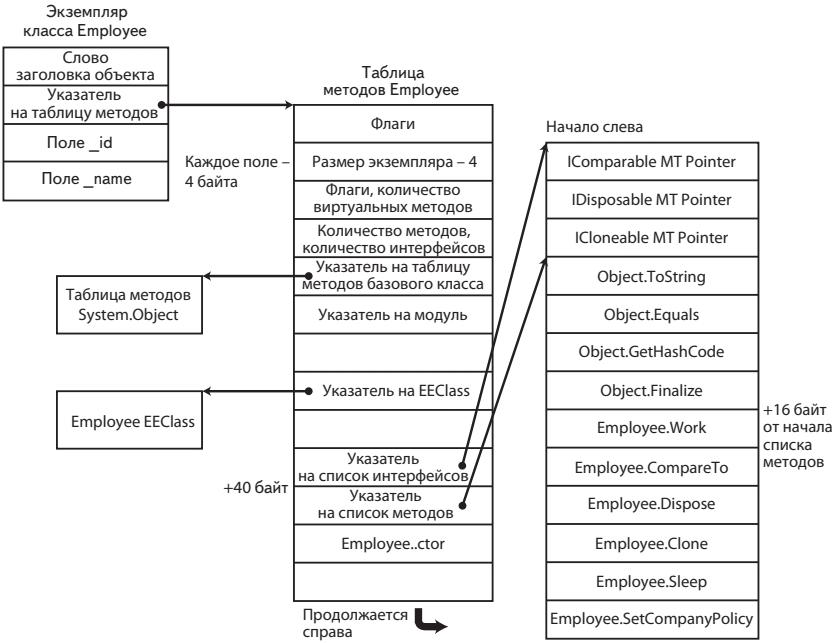

Рис. 3.4. Подробная схема таблицы методов класса Employee, включающая внутренние указатели на список интерфейсов и список методов, используемые для вызова виртуальных методов.

Примечание. *Дополнительная операция разыменования указателя, необходимая для перехода к таблице адресов методов, позволяет хранить эту таблицу отдельно от таблицы методов объекта, в другой области памяти. Например, если попробовать исследовать таблицу методов класса* `System.Object`*, можно обнаружить, что адреса методов хранятся в другом месте. Кроме того, классы с большим количеством виртуальных методов будут иметь несколько указателей на таблицы, обеспечивая повторное использование таблиц методов в классах-наследниках.*

Вызов методов экземпляров ссылочных типов

Очевидно, что таблица методов может использоваться для вызова методов произвольных экземпляров. Допустим, что на стеке, по ад-

ресу EBP-64, хранится адрес объекта Employee, схема таблицы методов которого была представлена на рис. 3.4. Тогда мы можем вызвать виртуальный метод Work, используя следующую последовательность инструкций:

```
mov ecx, dword ptr [ebp-64]
mov eax, dword ptr [ecx]     ; указатель на таблицу методов
mov eax, dword ptr [eax+40] ; указатель на фактический список
                             ; методов в таблице
call dword ptr [eax+16]      ; Work – пятое поле (четвертое, если
                             ; считать с нуля)
```

Первая инструкция копирует ссылку со стека в регистр ECX, вторая – разыменовывает регистр ECX, чтобы получить указатель на таблицу методов объекта, третья – извлекает внутренний указатель на список методов в таблице методов (который находится с постоянным смещением 40 байт), и четвертая – разыменовывает элемент списка со смещением 16, чтобы получить адрес метода Work и вызвать его. Чтобы понять, зачем необходимо использовать таблицу методов для диспетчеризации виртуальных методов, рассмотрим, как происходит привязка во время выполнения – то есть, как с помощью виртуальных методов реализуется полиморфизм.

Допустим, что имеется еще один класс Manager, наследующий класс Employee и переопределяющий его виртуальный метод Work, а также реализующий еще один интерфейс:

```
public class Manager : Employee, ISerializable
{
    private List<Employee> _reports;
    public override void Work() ...
    //...реализзация интерфейса ISerializable опущена для экономии места
}
```

Компилятору может потребоваться вызвать метод Manager.Work через ссылку на объект типа Employee, как показано в следующем фрагменте:

```
Employee employee = new Manager(...);
employee.Work();
```

В данном конкретном случае компилятор мог бы решить – используя приемы статического анализа – что должен вызываться метод Manager.Work (чего не происходит в текущих реализациях C# и CLR). В общем случае, однако, когда имеется статическая ссылка типа Employee, компилятор должен отложить привязку методов во время выполнения. В действительности, единственный способ привязать

правильный метод – определить фактический тип объекта во время выполнения и вызвать виртуальный метод, основываясь на этой информации. Именно это и позволяет сделать таблица методов.

Как изображено на рис. 3.5, поле, соответствующее методу `Work`, в таблице методов класса `Manager` содержит иной адрес, а последовательность выбора метода остается прежней. Обратите внимание, что смещение переопределенного поля с адресом метода отличается, потому что класс `Manager` реализует дополнительный интерфейс; однако поле «Указатель на список методов» имеет то же смещение и нивелирует это отличие:

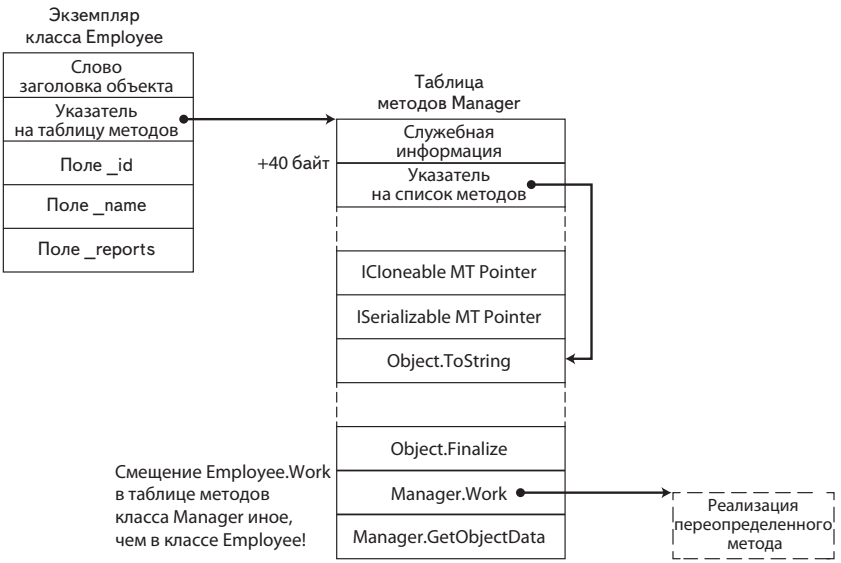

Рис. 3.5. Схема таблицы методов класса Manager. Эта таблица содержит дополнительное поле с указателем на таблицу методов нового интерфейса, что увеличивает значение поля «Указатель на список методов».

```
mov ecx, dword ptr [ebp-64]
mov eax, dword ptr [ecx]
mov eax, dword ptr [ecx+40] ; нивелирует различия в смещении метода Work
call dword ptr [eax+16]     ; абсолютное смещение от начала таблицы методов
```

Примечание. *Поддержка схемы размещения объектов, не гарантирующей, что смещение местоположения переопределенных методов от начала таблицы методов будет одинаковым в родительском и дочернем*

классах, впервые появилась CLR 4.0. В предыдущих версиях CLR список интерфейсов, реализуемых типом, хранился в конце таблицы методов, после списка адресов; это означало, что смещение адреса метода `Object.Equals` *(и всех других адресов реализаций методов) было постоянным во всех производных классах. А это, в свою очередь, означало, что последовательность вызова виртуального метода состояла всего из трех инструкций вместо четырех (третья инструкция в последовательности выше была не нужна). В старых статьях и книгах можно увидеть упоминание последовательности вызова из трех инструкций, что служит яркой демонстрацией, как может изменяться внутренняя реализация CLR.*

Вызов невиртуальных методов

Аналогичную последовательность инструкций можно также использовать для вызова невиртуальных методов. Но при обращении к невиртуальным методам нет необходимости использовать таблицу методов: адрес метода (или, по крайней мере, заглушки для JIT-компилятора) известен, когда JIT-компилятору потребуется скомпилировать вызов метода. Например, если на стеке, по адресу `EBP-64`, хранится адрес объекта `Employee`, как и в предыдущих примерах, тогда последовательность инструкций вызова метода `TakeVacation` с параметром `5` будет следующей:

```
mov edx, 5                    ; параметр передается в регистре –
                              ; в соответствии с соглашениями
mov ecx, dword ptr [ebp-64] ; все еще необходимо, потому что ECX
                              ; содержит 'this'
call dword ptr [0x004a1260]
```

Перед вызовом все еще необходимо загрузить адрес объекта в регистр `ECX` – все методы экземпляров ожидают получить этот неявный параметр в регистре `ECX`. Однако, больше не нужно разыменовать указатель на таблицу методов и получать адрес метода из нее. Но JIT-компилятору все еще требуется возможность изменить адрес вызова после вызова; поэтому выполняется косвенный вызов по адресу, хранящемуся в ячейке памяти (`0x004a1260` в этом примере), которая первоначально хранит адрес заглушки JIT-компилятора и обновляется им сразу после компиляции метода.

К сожалению, последовательность инструкций, представленная выше, страдает одним существенным недостатком. Она допускает возможность вызова метода с пустой ссылкой на объект, что может остаться незамеченным, пока метод экземпляра не попытается обратиться к полю экземпляра или вызвать виртуальный метод, что может вызвать ошибку нарушения прав доступа. В действительности

это является характерным поведением процедуры вызова методов экземпляров в языке C++ – следующий код выполнится без ошибок в большинстве окружений C++, но такая возможность заставляет разработчиков на C# почувствовать себя неуютно:

```
class Employee {
public: void Work() { } // пустой невиртуальный метод
};
Employee* pEmployee = NULL;
pEmployee->Work();      // выполнится без ошибок
```

Если взглянуть на код, сгенерированный JIT-компилятором для вызова невиртуального метода экземпляра, он будет содержать дополнительную инструкцию:

```
mov edx, 5                    ; параметр передается в регистре –
                              ; в соответствии с соглашениями
mov ecx, dword ptr [ebp-64] ; все еще необходимо, потому что ECX
                              ; содержит 'this'
cmp ecx, dword ptr [ecx]
call dword ptr [0x004a1260]
```

Напомню, что инструкция `CMP` вычитает второй операнд из первого и устанавливает флаги процессора в соответствии с результатом. Код выше не использует результат сравнения. Тогда как инструкция `CMP` сможет предотвратить вызов объекта с пустой ссылкой на него? Все просто, инструкция `CMP` попытается обратиться к ячейке памяти по адресу в регистре `ECX`, содержащем ссылку на объект. Если ссылка пустая, эта операция потерпит неудачу с ошибкой нарушения прав доступа, потому что обращение к адресу `0` всегда считается недопустимым для процессов в Windows. Среда выполнения CLR преобразует эту ошибку в исключение `NullReferenceException` и возбудит его в точке вызова. Это лучше, чем городить огород с проверкой ссылки внутри метода после его вызова. Кроме того, инструкция `CMP` занимает всего два байта в памяти и способна проверять любые недопустимые адреса, отличные от `null`.

> **Примечание.** *При вызове виртуальных методов необходимость в такой проверке отсутствует; проверка ссылки на объект выполняется косвенно, стандартной последовательностью инструкций вызова, в момент обращения к таблице методов, гарантирующей допустимость указателя на таблицу методов. Даже в вызовах невиртуальных методов не всегда можно увидеть инструкцию* `CMP`*. В последних версиях CLR JIT-компилятор способен избавляться от излишних проверок. Например, если поток выполнения программы только что вернулся из вызова виртуального метода объекта – содержащего неявную проверку ссылки – JIT-компилятор может опустить инструкцию* `CMP`.

Но причина нашего интереса к таким тонким отличиям вызовов виртуальных и невиртуальных методов вовсе не в дополнительной инструкции обращения к памяти и не в наличии или отсутствии других инструкций. Главная причина состоит в том, что виртуальные методы препятствуют оптимизации, выражающейся во *встраивании методов*, чрезвычайно важной для современных высокопроизводительных приложений. Встраивание – это очень простой трюк компилятора, когда вызов короткого или простого метода замещается его телом. Например, в следующем фрагменте вполне разумно было бы заменить вызов метода `Add` единственной операцией, выполняемой в нем:

```
int Add(int a, int b)
{
    return a + b;
}
int c = Add(10, 12);
// предположим, что c будет использоваться далее в коде
```

Неоптимизированная последовательность будет состоять почти из 10 инструкций: три – для подготовки параметров и вызова метода, две – для подготовки кадра стека метода, одна – для сложения операндов, две – для освобождения кадра стека метода и одна – для выхода из метода. Оптимизированная последовательность вызова будет состоять из единственной инструкции – догадались какой? Вполне логично предположить, что это будет инструкция `ADD`, но в действительности здесь может быть использована другая оптимизация, называемая сверткой констант, которая производит вычисление результата на этапе компиляции и присваивает переменной `c` значение `22`.

Разница в производительности встроенных и невстроенных вызовов методов может быть просто громадной, особенно когда методы просты, как в примере выше. Свойства, например, являются отличными кандидатами на встраивание, а автоматические свойства, генерируемые компилятором – тем более, потому что не содержат никакой логики, кроме прямого обращения к полю. Однако, виртуальные методы препятствуют встраиванию, потому что данная оптимизация может применяться, только когда на этапе компиляции (в случае с JIT-компилятором – на этапе JIT-компиляции) известно, какой именно метод вызывается. Когда адрес вызываемого метода определяется на этапе выполнения исходя из информации о типе, встроенной в объект, нет никакой возможности корректно выполнить встраивание виртуального метода. Если бы все методы были виртуальными по умолчанию, свойства так же были бы виртуальными, а накопленная

потеря производительности, обусловленная косвенными вызовами методов там, где иначе можно было бы применить встраивание, была бы просто ошеломляющей.

Кого-то может заинтересовать эффект влияния ключевого слова `sealed` на процедуру вызова метода, особенно теперь, когда мы познакомились с оптимизацией, выполняющей встраивание. Например, если в классе `Manager` метод `Work` объявить как `sealed`, вызов его по ссылке на объект, имеющей статический тип `Manager`, можно выполнить, как если бы он был невиртуальным методом экземпляра:

```
public class Manager : Employee
{
    public override sealed void Work() ...
}
Manager manager = ...; // может быть экземпляром Manager или
                       // производного типа
manager.Work();        // здесь возможен прямой вызов!
```

Тем не менее, на момент написания этих строк ключевое слово `sealed` не оказывало влияния на вызов методов во всех версиях CLR, протестированных нами, хотя знание, что класс или метод является конечным (sealed) позволяет использовать более эффективную последовательность инструкций вызова невиртуальных методов.

Вызов статических методов и методов интерфейсов

Для полноты картины необходимо обсудить еще две разновидности методов: статические методы и методы интерфейсов. Вызов статических методов выполняется очень просто: в этом случае нет необходимости загружать ссылку на объект и достаточно просто вызвать метод (или заглушку JIT-компилятора). Поскольку вызов выполняется без использования таблицы методов, JIT-компилятор применяет тот же трюк, что и в случае невиртуальных методов экземпляра: вызов выполняется косвенно, через специальную ячейку в памяти, которая обновляется после JIT-компиляции метода.

Однако методы интерфейсов – совершенно иное дело. Может показаться, что вызов метода интерфейса ничем не отличается от вызова виртуального метода экземпляра. Однако это не так. Интерфейсы обеспечивают разновидность полиморфизма, напоминающего классические виртуальные методы. К сожалению, нет никакой гарантии, что реализации методов какого-то определенного интерфейса окажутся в тех же позициях в таблице методов во всех классах, реализу-

ющих этот интерфейс. Взгляните на следующий фрагмент, где представлены два класса, реализующих интерфейс `IComparable`:

```
class Manager : Employee, IComparable {
    public override void Work() ...
    public void TakeVacation(int days) ...
    public static void SetCompanyPolicy(...) ...
    public int CompareTo(object other) ...
}

class BigNumber : IComparable {
    public long Part1, Part2;
    public int CompareTo(object other) ...
}
```

Очевидно, что таблицы методов в этих классах будут иметь разную структуру, и номер слота, ссылающегося на метод `CompareTo` в них, будет отличаться. Сложность иерархии объектов и наличие реализаций множества интерфейсов делают очевидной необходимость дополнительной идентификации местоположения методов интерфейсов в таблице методов.

В предыдущих версиях CLR эта информация хранилась в глобальной таблице (на уровне `AppDomain`), индексируемой числовым идентификатором интерфейса, генерируемым при первой загрузке интерфейса. В таблице методов имеется специальное поле (со смещением 12), указывающее на соответствующее место в глобальной таблице интерфейсов, а поля в глобальной таблице интерфейсов ссылаются обратно на таблицу методов, точнее на подтаблицу внутри нее, где хранятся указатели а методы. Такое решение обеспечивает возможность вызова методов в несколько этапов, как показано ниже:

```
mov ecx, dword ptr [ebp-64] ; ссылка на объект
mov eax, dword ptr [ecx]    ; указатель на таблицу методов
mov eax, dword ptr [eax+12] ; указатель на карту интерфейсов
mov eax, dword ptr [eax+48] ; смещение данного интерфейса в карте,
                            ; определяемое на этапе компиляции
call dword ptr [eax]        ; первый метод по адресу EAX,
                            ; второй - по адресу EAX+4, и т. д.
```

Выглядит очень сложно и весьма недешево! Для получения адреса реализации метода интерфейса и его вызова требуется четырежды обратиться к памяти. Для некоторых интерфейсов это может оказаться слишком дорогим удовольствием. Именно поэтому вы никогда не увидите такую последовательность инструкций, даже после отключения всех оптимизаций. JIT-компилятор использует несколько трюков, чтобы обеспечить встраивание методов интерфейсов, по крайней мере, в наиболее типичных случаях.

Анализ горячего пути (hot-path analysis) – когда JIT-компилятор обнаруживает частое использование одной и той же реализации интерфейса, он заменять конкретную процедуру вызова оптимизированным кодом и может даже встраивать часто используемые реализации интерфейсов:

```
mov ecx, dword ptr [ebp-64]
cmp dword ptr [ecx], 00385670 ; ожидается указатель на таблицу методов
jne 00a188c0                  ; холодный путь, показан ниже в псевдокоде
jmp 00a19548                  ; горячий путь, здесь может быть встроено
                              ; тело метода
    // холодный путь:
if (--wrongGuessesRemaining < 0) { ; начальное значение 100
    // заменить процедуру вызова в коде, как обсуждается ниже
} else {
    // стандартный вызов метода интерфейса, как обсуждается выше
}
```

Частотный анализ (frequency analysis) – когда JIT-компилятор обнаруживает, что выбор горячего пути не соответствует конкретному вызову (по серии нескольких вызовов), он замещает ожидаемый горячий путь новым горячим путем и продолжает выбирать между ними всякий раз, когда предположения оказываются неверными:

```
start: if (obj->MTP == expectedMTP) {
    // прямой переход к ожидаемой реализации
} else {
    expectedMTP = obj->MTP;
    goto start;
}
```

Более подробно процедура вызова методов интерфейсов описывается в статье Саши Голдштейна (Sasha Goldshtein) «JIT Optimizations» (http://www.codeproject.com/Articles/25801/JIT-Optimizations) и в статье Вэнса Моррисона (Vance Morrison) (http://blogs.msdn.com/b/vancem/archive/2006/03/13/550529.aspx). Вызов метода интерфейса – это давно и активно обсуждаемая тема; в будущих версиях CLR могут появиться другие оптимизации, не обсуждавшиеся здесь.

Блоки синхронизации и ключевое слово lock

Второе поле, встраиваемое во все экземпляры ссылочных типов – это слово заголовка объекта (или индекс блока синхронизации). В отличие от указателя на таблицу методов, это поле используется для самых разных целей, таких как синхронизация, хранение служебной

информации сборщика мусора, финализация и хранение хеш-кода. Некоторые биты этого поля определяют, какая информация хранится в нем в каждый конкретный момент.

Самым сложным является использование слова заголовка объекта механизмом мониторинга CLR для синхронизации, взаимодействие с которым в языке C# осуществляется с помощью ключевого слова `lock`. Суть заключается в следующем: несколько потоков выполнения могут попытаться одновременно выполнить фрагмент кода, защищенного инструкцией `lock`, но только одному из них будет позволено это:

```
class Counter
{
    private int _i;
    private object _syncObject = new object();
    public int Increment()
    {
        lock (_syncObject)
        {
            return ++_i; // только один поток сможет выполнить эту инструкцию
        }
    }
}
```

Однако ключевое слово `lock` – это лишь синтаксический сахар, скрывающий в себе следующую конструкцию, использующую методы `Monitor.Enter` и `Monitor.Exit`:

```
class Counter
{
    private int _i;
    private object _syncObject = new object();
    public int Increment()
    {
        bool acquired = false;
        try
        {
            Monitor.Enter(_syncObject, ref acquired);
            return ++_i;
        }
        finally
        {
            if (acquired) Monitor.Exit(_syncObject);
        }
    }
}
```

Чтобы обеспечить исключительную блокировку, механизм синхронизации можно связать с любым объектом. Поскольку иници-

ализация механизма синхронизации для каждого объекта в приложении – слишком дорогое удовольствие, она выполняется только по требованию, когда объект впервые используется для синхронизации. Когда это потребуется, среда выполнения CLR создаст структуру, которая называется *блоком синхронизации* (sync block), в глобальном массиве, называемом *таблицей блоков синхронизации* (sync block table). Блок синхронизации содержит обратную ссылку на объект, владеющий блоком (это – «слабая» ссылка, не мешающая утилизации объекта сборщиком мусора), и, кроме всего прочего, ссылку на механизм синхронизации, называемый монитором (monitor), реализация которого основана на событиях Win32. Числовой индекс созданного блока синхронизации будет сохранен в слове заголовка объекта. При последующих попытках использовать объект для нужд синхронизации, из него будет извлечен индекс существующего блока синхронизации и задействован соответствующий объект монитора.

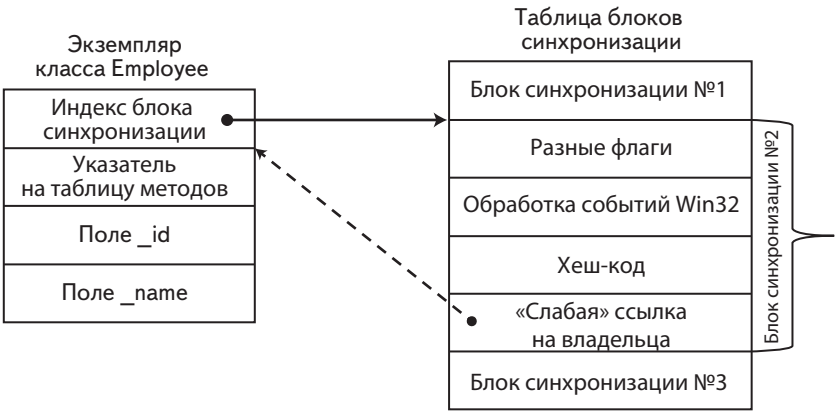

Рис. 3.6. Блок синхронизации, связанный с экземпляром объекта. Поле индекса блока синхронизации хранит только индекс в таблице блоков синхронизации, что дает среде выполнения CLR возможность свободно изменять размеры и перемещать таблицу синхронизации в памяти.

Когда блок синхронизации не используется достаточно продолжительное время, сборщик мусора утилизирует его и разрывает связь между им и его объектом-владельцем, записывая в поле индекса блока синхронизации недопустимое значение. После утилизации блок синхронизации может быть связан с другим объектом, что позволяет

сэкономить системные ресурсы, необходимые для инициализации механизма синхронизации.

С помощью команды `!SyncBlk` библиотеки SOS можно исследовать конкурирующие блоки синхронизации, то есть блоки, которыми владеют потоки выполнения, ожидающие, пока блокировка будет освобождена другим потоком (таких ожидающих потоков выполнения может быть несколько). В версии CLR 2.0 была добавлена оптимизация, откладывающая создание блока синхронизации до момента, когда он потребуется для нужд синхронизации. Пока блок синхронизации отсутствует, среда выполнения CLR может управлять состоянием синхронизации с помощью *тонких блокировок* (thin lock). Некоторые примеры приводятся ниже.

Для начала рассмотрим слово заголовка объекта, которые еще не использовался для синхронизации, но хеш-код в котором уже доступен (хеш-коды ссылочных типов мы обсудим далее в этой главе). В следующем примере регистр `EAX` указывает на объект `Employee`, имеющий хеш-код 46104728:

```
0:000> dd eax-4 L2
023d438c 0ebf8098 002a3860
0:000> ? 0n46104728
Evaluate expression: 46104728 = 02bf8098
0:000> .formats 0ebf8098
Evaluate expression:
  Hex: 0ebf8098
  Binary: 00001110 10111111 10000000 10011000
0:000> .formats 02bf8098
Evaluate expression:
  Hex: 02bf8098
  Binary: 00000010 10111111 10000000 10011000
```

Здесь отсутствует индекс блока синхронизации – имеется только хеш код и два бита установлены в 1, один из них, видимо, указывает, что слово заголовка объекта сейчас хранит хеш-код. Далее был выполнен вызов `Monitor.Enter` для объекта в одном из потоков выполнения, чтобы заблокировать его:

```
0:004> dd 02444390-4 L2
0244438c 08000001 00173868
0:000> .formats 08000001
Evaluate expression:
  Hex: 08000001
  Binary: 0000 1000 00000000 00000000 0000000 1
0:004> !syncblk
Index  SyncBlock  MonitorHeld  Recursion    Owning    Thread  Info  SyncBlock  Owner
    1   0097db4c            3          1  0092c698      1790     0  02444390   Employee
```

Объект был связан с блоком синхронизации №1, что доказывает вывод команды `!SyncBlk` (дополнительную информацию о колонках в выводе команды можно найти в документации к библиотеке SOS). Когда другой поток выполнения попытается выполнить инструкцию `lock` с тем же объектом, он попадет в стандартный цикл ожидания (допускающий возможность обработки сообщений, если этот поток обслуживает графический интерфейс пользователя). Ниже приводится дамп дна стека потока выполнения, ожидающего на мониторе:

```
0:004> kb
ChildEBP RetAddr Args to Child
04c0f404 75120bdd 00000001 04c0f454 00000001 ntdll!
NtWaitForMultipleObjects+0x15
04c0f4a0 76c61a2c 04c0f454 04c0f4c8 00000000 KERNELBASE!
WaitForMultipleObjectsEx+0x100
04c0f4e8 670f5579 00000001 7efde000 00000000 KERNEL32!
WaitForMultipleObjectsExImplementation+0xe0
04c0f538 670f52b3 00000000 ffffffff 00000001 clr!
WaitForMultipleObjectsEx_SO_TOLERANT+0x3c
04c0f5cc 670f53a5 00000001 0097db60 00000000 clr!
Thread::DoAppropriateWaitWorker+0x22f
04c0f638 670f544b 00000001 0097db60 00000000 clr!Thread::
DoAppropriateWait+0x65
04c0f684 66f5c28a ffffffff 00000001 00000000 clr!CLREventBase::
WaitEx+0x128
04c0f698 670fd055 ffffffff 00000001 00000000 clr!CLREventBase::
Wait+0x1a
04c0f724 670fd154 00939428 ffffffff f2e05698 clr!AwareLock::
EnterEpilogHelper+0xac
04c0f764 670fd24f 00939428 00939428 00050172 clr!AwareLock::
EnterEpilog+0x48
04c0f77c 670fce93 f2e05674 04c0f8b4 0097db4c clr!AwareLock::
Enter+0x4a
04c0f7ec 670fd580 ffffffff f2e05968 04c0f8b4 clr!AwareLock::
Contention+0x221
04c0f894 002e0259 02444390 00000000 00000000 clr!
JITutil_MonReliableContention+0x8a
```

Для синхронизации используется объект `25c`, являющийся дескриптором (handle) события:

```
0:004> dd 04c0f454 L1
04c0f454 0000025c
0:004> !handle 25c f
Handle 25c
  Type          Event
  Attributes    0
  GrantedAccess 0x1f0003:
         Delete,ReadControl,WriteDac,WriteOwner,Synch
         QueryState,ModifyState
```

```
  HandleCount   2
  PointerCount  4
  Name          <none>
  Object Specific Information
    Event Type Auto Reset
    Event is Waiting
```

И, наконец, если заглянуть в память блока синхронизации, связанного с этим объектом, можно будет без труда опознать хеш-код и дескриптор механизма синхронизации:

```
0:004> dd 0097db4c
0097db4c 00000003 00000001 0092c698 00000001
0097db5c 80000001 0000025c 0000000d 00000000
0097db6c 00000000 00000000 00000000 02bf8098
0097db7c 00000000 00000003 00000000 00000001
```

Обратите также внимание, что в предыдущем примере мы принудительно инициировали создание блока синхронизации, вызвав `GetHashCode` перед попыткой получить блокировку. В версии CLR 2.0 была добавлена еще одна оптимизация, позволяющая экономить время и память, – блок синхронизации не создается, если прежде объект не был связан с блоком синхронизации. Вместо этого CLR использует механизм, получивший название *тонкая блокировка* (thin lock). Когда объект блокируется первый раз и конкуренция за его обладание отсутствует (то есть, никакой другой поток выполнения не попытался заблокировать объект), CLR сохраняет в слове заголовка объекта идентификатор управляемого потока выполнения, владеющего объектом в настоящий момент. Например, ниже приводится пример слова заголовка объекта, заблокированного главным потоком приложения до того, как какой-либо другой поток попытался сделать то же самое:

```
0:004> dd 02384390-4
0238438c 00000001 00423870 00000000 00000000
```

Здесь управляемый поток выполнения с идентификатором 1 – это главный поток приложения, в чем можно убедиться, выполнив команду `!Threads`:

```
0:004> !Threads
ThreadCount:      2
UnstartedThread:  0
BackgroundThread: 1
PendingThread:    0
DeadThread:       0
Hosted Runtime:   no
```

```
                                                    Lock
  ID OSID ThreadOBJ  State  GC Mode    GC Alloc Context  Domain   Count Apt Exception
0  1 12f0 0033ce80   2a020  Preemptive 02385114:00000000 00334850 2     MTA
2  2 23bc 00348eb8   2b220  Preemptive 00000000:00000000 00334850 0     MTA (Finalizer)
```

Тонкая блокировка также обнаруживается командой `!DumpObj`, которая указывает также поток выполнения, захвативший блокировку. Аналогично, с помощью команды `!DumpHeap -thinlock` можно вывести все тонкие блокировки, присутствующие в настоящее время в управляемой динамической памяти:

```
0:004> !dumpheap -thinlock
  Address       MT     Size
02384390 00423870       12 ThinLock owner 1 (0033ce80) Recursive 0
02384758 5b70f98c       16 ThinLock owner 1 (0033ce80) Recursive 0
Found 2 objects.
0:004> !DumpObj 02384390
Name:        Employee
MethodTable: 00423870
EEClass:     004213d4
Size:        12(0xc) bytes
File:        D:\Development\...\App.exe
Fields:
      MT     Field   Offset   Type          VT    Attr    Value Name
00423970   4000001        4   CompanyPolicy  0  static 00000000 _policy
ThinLock owner 1 (0033ce80), Recursive 0
```

Когда другой поток выполнения попытается заблокировать объект, он будет приостановлен на короткое время, в ожидании снятия тонкой блокировки (то есть, когда информация о владельце исчезнет из слова заголовка объекта). Если в течение некоторого ограниченного времени блокировка не будет освобождена, она будет преобразована в блок синхронизации, индекс блока синхронизации будет сохранен в слове заголовка объекта, и с этого момента блокировкой потока выполнения будет управлять обычный механизм синхронизации Win32.

Внутреннее устройство типов значений

Теперь, получив представление об особенностях размещения ссылочных типов в памяти и назначениях полей в заголовке объекта, можно перейти к обсуждению типов значений. Для хранения типов значений в памяти используется значительно более простая схема, но она имеет некоторые ограничения, а кроме того, когда тип значения

используется там, где ожидается ссылка, выполняется его упаковка (boxing) – дорогостоящая процедура, устраняющая несовместимость. Главная причина, побуждающая использовать типы значений, как было показано в начале главы, это высокая плотность размещения в памяти и отсутствие накладных расходов.

Для дальнейшего обсуждения введем простой тип значения, который уже рассматривался в начале этой главы, – тип `Point2D`, представляющий координаты точки в двумерном пространстве:

```
public struct Point2D
{
    public int X;
    public int Y;
}
```

При сохранении экземпляра `Point2D` в памяти, инициализированного значениями координат `X=5` и `Y=7`, он имеет простой вид (рис. 3.7) и в нем отсутствуют «лишние» поля:

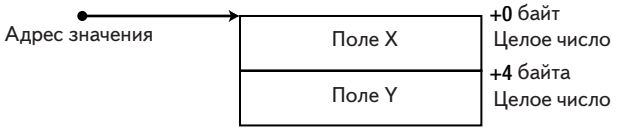

Рис. 3.7. Схема размещения в памяти экземпляра типа значения Point2D.

В некоторых редких случаях бывает желательно изменить схему размещения типов значений в памяти, например, для организации взаимодействий, когда экземпляр типа значения передается в неизменном виде неуправляемому коду. Возможность такой настройки обеспечивается двумя атрибутами, `StructLayout` и `FieldOffset`. Атрибут `StructLayout` может использоваться для определения полей объекта, которые должны размещаться последовательно, в соответствии с объявлением типа (действует по умолчанию). Атрибут `FieldOffset` позволяет явно определить смещения полей в памяти, что дает возможность создавать объединения в стиле языка C, где поля могут накладываться друг на друга. Ниже приводится простой пример типа значения, способного «преобразовывать» числа с плавающей запятой в 4-байтовое представление:

```
[StructLayout(LayoutKind.Explicit)]
public struct FloatingPointExplorer
{
    [FieldOffset(0)] public float F;
```

```
    [FieldOffset(0)] public byte B1;
    [FieldOffset(1)] public byte B2;
    [FieldOffset(2)] public byte B3;
    [FieldOffset(3)] public byte B4;
}
```

Если присвоить полю `F` объекта вещественное число, он автоматически изменит значения полей `B1`-`B4`, и наоборот. Фактически, поле `F` и поля `B1`-`B4` перекрываются в памяти, как показано на рис. 3.8:

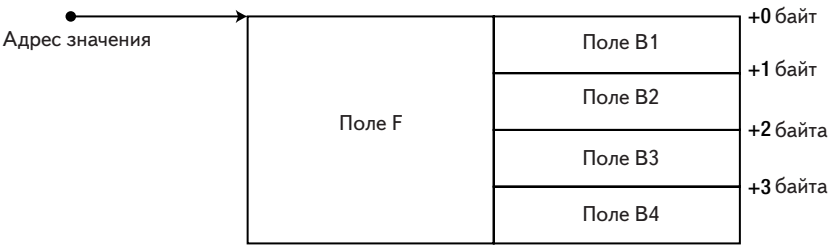

Рис. 3.8. Схема размещения в памяти экземпляра FloatingPointExplorer. Блоки, выровненные по горизонтали, перекрываются в памяти.

Поскольку экземпляры типов значений не имеют слова заголовка объекта и указателя на таблицу методов, они не обладают такой же богатой семантикой, как ссылочные типы. Теперь познакомимся, какие ограничения накладываются простой схемой размещения в памяти, и что происходит, когда разработчик пытается использовать типы значений там, где ожидаются ссылочные типы.

Ограничения типов значений

Сначала коснемся слова заголовка объекта. Если программа попытается использовать экземпляр типа значения для синхронизации, в большинстве случаев это является ошибкой (как будет показано чуть ниже). Но должна ли среда выполнения рассматривать такое использование недопустимым и возбуждать исключение? Взгляните на следующий пример, что произойдет, если метод `Increment` одного и того же экземпляра класса `Counter` будет вызван одновременно двумя потоками выполнения?

```
class Counter
{
    private int _i;
    public int Increment()
```

```
    {
        lock (_i)
        {
            return ++_i;
        }
    }
}
```

При попытке проверить это, мы наткнулись на неожиданное препятствие: компилятор C# не позволяет использовать типы значений с ключевым словом `lock`. Однако теперь мы вооружены знанием особенностей работы `lock` и можем попытаться обойти это препятствие:

```
class Counter
{
    private int _i;
    public int Increment()
    {
        bool acquired=false;
        try
        {
            Monitor.Enter(_i, ref acquired);
            return ++_i;
        }
        finally
        {
            if (acquired) Monitor.Exit(_i);
        }
    }
}
```

В результате, мы внесли в программу ошибку – оказывается, что сразу несколько потоков выполнения смогут одновременно приобрести блокировку и изменить значение поля `_i`, а кроме того, вызов `Monitor.Exit` возбудит исключение (подробнее об особенностях синхронизации доступа к целочисленной переменной рассказывается в главе 6). Проблема в том, что метод `Monitor.Enter` принимает параметр типа `System.Object`, который является ссылочным типом, а мы передаем ему экземпляр типа значения – по значению. Но, даже если бы была возможность передать значение там, где ожидается ссылка, значение, переданное методу `Monitor.Enter`, имело бы другую *идентичность* (identity), чем значение, переданное методу `Monitor.Exit`. Аналогично, значение, переданное методу `Monitor.Enter` в одном потоке, имеет иную идентичность, чем значение, переданное методу `Monitor.Enter` в другом потоке. При передаче значений (по значению!) там, где ожидаются ссылки, нет никакой возможности обеспечить правильную семантику работы блокировок.

Другой пример, объясняющий, почему семантика типов значений плохо согласуется со ссылками на объекты, – возврат экземпляров типов значений из методов. Взгляните на следующий фрагмент:

```
object GetInt()
{
    int i = 42;
    return i;
}
object obj = GetInt();
```

Метод `GetInt` возвращает экземпляр типа значения, что вполне типично для возвращаемых значений. Однако, вызывающая программа ожидает, что метод вернет ссылку на объект. Метод мог бы вернуть указатель на область памяти в стеке, где хранилась переменная `i` во время выполнения метода. Но, к сожалению, эта ссылка будет недействительной, потому что сразу после выхода из метода его кадр стека будет уничтожен. Этот пример показывает, что семантика «копирования по значению» (copy-by-value), которой по умолчанию обладают типы значений, плохо согласуется с ситуациями, когда ожидается ссылка на объект (в управляемой динамической памяти).

Виртуальные методы типов значений

Мы даже не коснулись указателя на таблицу методов, а у нас уже имеются проблемы с использованием типов значений. Теперь обратимся к виртуальным методам и методам интерфейсов. Среда выполнения CLR запрещает отношения наследования между типами значений, что делает невозможным определение новых виртуальных методов в типах значений. Однако в этом есть свои положительные стороны, потому что, если бы имелась возможность определять виртуальные методы в типах значений, для вызова этих методов потребовался бы указатель на таблицу методов, не являющийся частью экземпляров типов значений. Это не самое важное ограничение, потому что применение семантики «копирования по значению» к ссылочным типам сделало бы невозможным поддержку полиморфизма, требующей ссылку на объект.

Однако типы значений снабжаются несколькими виртуальными методами, унаследованными от `System.Object`. Вот некоторые из них: `Equals`, `GetHashCode`, `ToString` и `Finalize`. Здесь мы рассмотрим только первые два, но, все, что будет говориться далее, в значительной степени относится и к остальным методам. Начнем с исследования их сигнатур:

```
public class Object
{
    public virtual bool Equals(object obj) ...
    public virtual int GetHashCode() ...
}
```

Эти виртуальные методы реализованы для каждого типа в .NET, включая и типы значений. Это означает, что виртуальный метод можно вызвать для любого экземпляра типа значения, даже при том, что он не имеет указателя на таблицу методов! Это – третий пример, как схема размещения типов значений в памяти влияет на наши возможности выполнять даже простейшие операции с экземплярами типов значений, требующих некоторого механизма, который «превращал» бы их в нечто, что могло бы использоваться как «настоящий» объект.

Упаковка

Всякий раз, когда компилятор языка сталкивается с ситуацией, когда экземпляр типа значения требуется интерпретировать как экземпляр ссылочного типа, он вставляет в байт-код на языке IL инструкцию `box`. JIT-компилятор, в свою очередь, встретив эту инструкцию, генерирует вызов метода, который выделяет место в динамической памяти, копирует туда содержимое экземпляра типа значения и обертывает содержимое типа значения заголовком объекта, то есть, добавляет слово заголовка объекта и указатель на таблицу методов. Именно такая «упаковка» используется всякий раз, когда требуется ссылка на объект (рис. 3.9). Обратите внимание, что упакованный экземпляр никак не связан с оригинальным экземпляром типа значения – изменения в одном никак не затрагивают другой.

```
.method private hidebysig static object GetInt() cil managed
{
    .maxstack 8
    L_0000: ldc.i4.s 0x2a
    L_0002: box int32
    L_0007: ret
}
```

Упаковка – довольно дорогостоящая операция, включающая выделение памяти, копирование данных и впоследствии добавляет накладные расходы на сборку мусора, необходимую для утилизации временных упакованных экземпляров. С появлением поддержки обобщенных типов в CLR 2.0, надобность в упаковке, кроме как в механизме рефлексии и в ряде других редких ситуаций, практически от-

пала. Тем не менее, упаковка остается одной из важнейших проблем производительности во многих приложениях; как будет показано далее, «правильное применение типов значений» с целью предотвращения упаковку невозможно без представлений о том, как выполняется вызов методов типов значений.

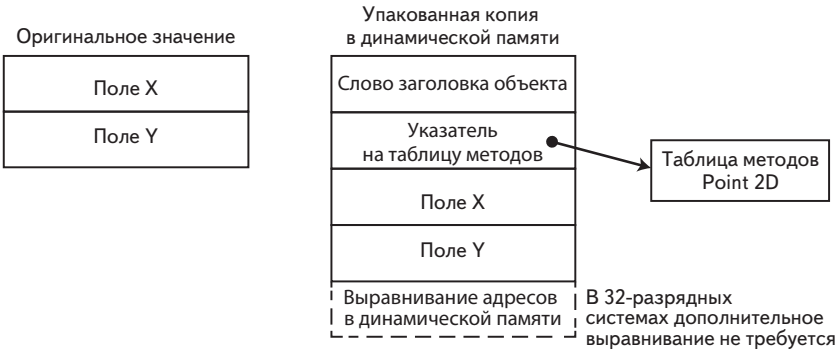

Рис. 3.9. Оригинальное значение и упакованная копия в динамической памяти. Упакованная копия имеет стандартный набор служебной информации, характерной для ссылочных типов (слово заголовка объекта и указатель на таблицу методов) и может занимать в динамической памяти дополнительное пространство из-за необходимости выравнивания адресов.

Невзирая на проблемы производительности, упаковка все же позволяет решить некоторые проблемы, с которыми мы столкнулись выше. Например, метод `GetInt` может вернуть ссылку на упакованный объект, содержащий значение 42. Этот объект продолжит существование, пока на него имеется хотя бы одна ссылка на него, и он никак не зависит от жизненного цикла локальных переменных, располагающихся на стеке метода. Аналогично, методу `Monitor.Enter`, ожидающему получить ссылку на объект, можно передать ссылку на упакованный объект, который он будет использовать для синхронизации. К сожалению, упакованные объекты, созданные на основе одного и того же экземпляра типа значения, не будут считаться идентичными. То есть, упакованный объект, переданный методу `Monitor.Exit`, будет отличаться от упакованного объекта, переданного методу `Monitor.Enter`, а упакованный объект, переданный методу `Monitor.Enter` в одном потоке выполнения, будет отличаться от упакованного объекта, переданного методу `Monitor.Enter` в другом потоке выполнения. Это означает, что использование любых типов значений для синхронизации на основе монитора ошибочно в принципе.

Еще одним важным вопросом остаются виртуальные методы, наследуемые от `System.Object`. Как оказывается, типы значений не наследуют класс `System.Object` непосредственно – они наследуют промежуточный тип `System.ValueType`.

> **Примечание.** *Как ни странно,* `System.ValueType` *является ссылочным типом – среда выполнения CLR различает типы значений и ссылочные типы по следующему критерию: типы значений наследуют* `System.ValueType`*. Согласно этому критерия System.ValueType является ссылочным типом.*

Класс `System.ValueType` переопределяет виртуальные методы `Equals` и `GetHashCode`, унаследованные от `System.Object`, и на то есть веская причина: типы значений по умолчанию используют иную семантику сравнения, и эта семантика должна быть где-то реализована. Например, переопределенная версия метода `Equals` в классе `System.ValueType` гарантирует, что сравнение типов значений будет выполняться по их содержимому, тогда как оригинальный метод `Equals` в классе `System.Object` сравнивает ссылки на объекты (их идентичность).

Не вдаваясь в подробности реализации виртуальных методов в классе `System.ValueType`, рассмотрим следующую ситуацию. У вас имеется десять миллионов объектов `Point2D` в списке `List<Point2D>`, и требуется найти единственный объект `Point2D`, используя метод `Contains`. Метод `Contains` не имеет лучшего способа поиска, как выполнить обход всех десяти миллионов объектов в списке и сравнить каждый из них с образцом.

```
List<Point2D> polygon = new List<Point2D>();
// добавление десяти миллионов точек в список
Point2D point = new Point2D { X = 5, Y = 7 };
bool contains = polygon.Contains(point);
```

Обход списка с десятью миллионами точками и сравнение каждой из них с указанным образцом требует времени, но сама по себе это довольно быстрая операция. В ходе поиска придется извлечь из памяти примерно 80000000 байт (по восемь байтов на каждый объект `Point2D`), и операция сравнения выполняется очень быстро. Досадно, но для сравнения двух объектов `Point2D` требуется вызвать виртуальный метод `Equals`:

```
Point2D a = ..., b = ...;
a.Equals(b);
```

Здесь мы столкнулись с двумя проблемами. Во-первых, метод `Equals` – даже его переопределенная версия в `System.ValueType` – принимает ссылку на экземпляр `System.Object`. Чтобы иметь возможность интерпретировать объект `Point2D` как экземпляр ссылочного типа, его необходимо упаковать, как было описано выше, то есть в данном примере объект `b` должен быть упакован. Кроме того, для вызова метода `Equals` требуется также упаковать объект `a`, чтобы получить указатель на таблицу методов!

> **Примечание.** *JIT-компилятор способен производить вычисления по короткой схеме, что могло бы обеспечить возможность непосредственного вызова метода* `Equals`*, потому что типы значений объявлены конечными (sealed) а вызываемый виртуальный метод известен уже на этапе компиляции, независимо от того, переопределяет ли тип* `Point2D` *метод* `Equals` *или нет (такую возможность допускает префикс* `constrained` *языка IL). Однако, из-за того, что* `System.ValueType` *является ссылочным типом, метод* `Equals` *может интерпретировать свой неявный параметр* `this` *как экземпляр ссылочного типа, даже при том, что для вызова метода используется экземпляр типа значения (*`Point2D a`*) – а это требует упаковки.*

В итоге, на каждый вызов `Equals` приходится две операции упаковки экземпляров `Point2D`. Для 10 000 000 вызовов `Equals` получается 20 000 000 операций упаковки, каждая из которых выделяет 16 байт памяти (в 32-разрядной системе), а в общей сложности выделяется 320 000 000 байт и копируется 160 000 000 байт. Продолжительность этих манипуляций с памятью намного превосходит время, действительно необходимое для сравнения двух точек.

Предотвращение упаковки типов значений с помощью метода Equals

Можно ли полностью избавиться от этих операций упаковки? Один из способов – переопределить метод `Equals` и предоставить реализацию, подходящую для нашего типа значения:

```
public struct Point2D
{
    public int X;
    public int Y;
    public override bool Equals(object obj)
    {
        if (!(obj is Point2D)) return false;
        Point2D other = (Point2D)obj;
        return X == other.X && Y == other.Y;
    }
}
```

Благодаря способности JIT-компилятора производить вычисления по короткой схеме, как описывалось выше, вызов `a.Equals(b)` требует упаковки только значения `b`, потому что метод принимает ссылку на объект. Чтобы избавиться от второй операции упаковки, нужно добавить перегруженную версию метода `Equals`:

```
public struct Point2D
{
    public int X;
    public int Y;
    public override bool Equals(object obj) ... // как и прежде
    public bool Equals(Point2D other)
    {
        return X == other.X && Y == other.Y;
    }
}
```

Теперь, когда компилятор встретит вызов `a.Equals(b)`, он безусловно предпочтет использовать вторую, перегруженную версию метода, потому что тип его параметра более точно соответствует типу аргумента. Пока мы не отвлеклись, заметим, что существуют и другие методы, кандидаты на перегрузку – достаточно часто мы сравниваем объекты с помощью операторов == и `!=`:

```
public struct Point2D
{
    public int X;
    public int Y;
    public override bool Equals(object obj) ... // как и прежде
    public bool Equals(Point2D other) ... // как и прежде
    public static bool operator==(Point2D a, Point2D b)
    {
        return a.Equals(b);
    }
    public static bool operator!= (Point2D a, Point2D b)
    {
        return !(a == b);
    }
}
```

Этого достаточно в большинстве случаев. Но иногда возникают крайние ситуации, связанные с особенностями реализации обобщенных типов в CLR, которые вызывают упаковку, когда `List<Point2D>` обращается к методу `Equals` для сравнения двух экземпляров `Point2D`, с типом `Point2D`, как реализацией его параметра обобщенного типа (`T`). Подробнее об этом мы поговорим в главе 5, а пока лишь отметим, что тип `Point2D` должен наследовать интерфейс

IEquatable<Point2D>, обеспечивающий более подходящее поведение в List<T>, и EqualityComparer<T>, чтобы дать возможность вызывать перегруженную версию метода Equals через интерфейс (ценой вызова виртуального метода в абстрактном методе EqualityComparer<T>.Equals). Результатом является 10-кратное увеличение производительности и полное избавление от манипуляций с динамической памятью (обусловленных процедурой упаковки) при поиске точки в списке из 10000000 экземпляров Point2D!

```
public struct Point2D : IEquatable<Point2D>
{
    public int X;
    public int Y;
    public bool Equals(Point2D other) ... // как и прежде
}
```

Сейчас самое время, чтобы поразмышлять на тему реализации интерфейсов в в типах значений. Как было показано выше, для вызова метода интерфейса необходим указатель на таблицу методов, который в свою очередь требует упаковки экземпляра типа значения. В действительности же, преобразование экземпляра типа значения в переменную с типом интерфейса требует упаковки потому, что ссылки на интерфейсы могут интерпретироваться как ссылки на объекты:

```
Point2D point = ...;
IEquatable<Point2D> equatable = point; // здесь выполняется упаковка
```

Однако, когда вызов метода интерфейса выполняется через статически типизированную переменную типа значения, упаковка не производится (здесь действуют все та же поддержка вычислений по короткой схеме, которая обеспечивается префиксом constrained в языке IL):

```
Point2D point = ..., anotherPoint = ...;
point.Equals(anotherPoint); // упаковка не выполняется, вызывается
                            // Point2D.Equals(Point2D)
```

При использовании типов значений через интерфейсы возникает потенциальная проблема, если типы значений являются изменяемыми, как тип Point2D, вокруг которого мы крутимся в этой главе. Как мы уже знаем, изменение упакованной копии никак не сказывается на оригинале, что может приводить к неожиданному поведению:

```
Point2D point = new Point2D { X = 5, Y = 7 };
Point2D anotherPoint = new Point2D { X = 6, Y = 7 };
IEquatable<Point2D> equatable = point; // здесь выполняется упаковка
equatable.Equals(anotherPoint);        // вернет false
```

```
point.X = 6;
point.Equals(anotherPoint);              // вернет true
equatable.Equals(anotherPoint);          // вернет false, упакованный
                                         // объект не изменился!
```

Это является одной из причин, почему часто рекомендуется создавать типы значений неизменяемыми, а изменения производить только за счет создания копий. (Примером такого неизменяемого типа значения может служить `System.DateTime`.)

Еще одной проблемой метода `ValueType.Equals` является его фактическая реализация. Сравнение двух экземпляров произвольных типов значений по их содержимому далеко не тривиальная задача. В результате дизассемблирования получается следующая картина (немного отредактированная для краткости):

```
public override bool Equals(object obj)
{
    if (obj == null) return false;
    RuntimeType type = (RuntimeType) base.GetType();
    RuntimeType type2 = (RuntimeType) obj.GetType();
    if (type2 != type) return false;

    object a = this;
    if (CanCompareBits(this))
    {
        return FastEqualsCheck(a, obj);
    }
    FieldInfo[] fields = type.GetFields(BindingFlags.NonPublic |
                                        BindingFlags.Public |
                                        BindingFlags.Instance);
    for (int i = 0; i < fields.Length; i++)
    {
        object obj3 = ((RtFieldInfo) fields[i]).InternalGetValue(a, false);
        object obj4 = ((RtFieldInfo) fields[i]).InternalGetValue(obj, false);
        if (obj3 == null && obj4 != null)
            return false;
        else if (!obj3.Equals(obj4))
            return false;
    }
    return true;
}
```

Проще говоря, если `CanCompareBits` вернет истинное значение, проверка равенства выполняется с помощью `FastEqualsCheck`; в противном случае метод входит в цикл, где с помощью класса `FieldInfo` рекурсивно извлекаются поля и сравниваются вызовом метода `Equals`. Разумеется, использование механизма рефлексии в цикле крайне отрицательно сказывается на производительности – механизм реф-

лексии весьма дорог в использовании и все остальные потери производительности бледнеют на его фоне. Определения методов `CanCompareBits` и `FastEqualsCheck` в CLR являются «внутренними вызовами», не реализованными на языке IL, поэтому их нельзя так просто дизассемблировать. Однако экспериментальным путем мы установили, что `CanCompareBits` возвращает истинное значение, если выполняется одно из следующих условий:

1. Тип значения содержит только поля простых типов и не переопределяет метод `Equals`.
2. Тип значения содержит только поля типов значений, для которых выполняется условие (1) и не переопределяет метод `Equals`.
3. Тип значения содержит только поля типов значений, для которых выполняется условие (2) и не переопределяет метод `Equals`.

Метод `FastEqualsCheck` тоже не менее таинственный, но в действительности он выполняет операцию `memcmp` – выполняющую побайтовое сравнение обоих экземпляров типов значений. К сожалению, оба эти метода остаются внутренней особенностью реализации, и полагаться на них, как на высокопроизводительный способ сравнения экземпляров ваших типов значений – не самая лучшая идея.

Метод GetHashCode

Последний метод, который важно переопределить – это метод `GetHashCode`. Прежде чем показать подходящую реализацию, давайте вспомним, для чего он используется. Хеш-коды часто применяются совместно с хеш-таблицами, структурами данных, которые (при определенных условиях) обеспечивают постоянное время выполнения ($O(1)$) операций добавления, поиска и удаления произвольных данных. В числе типичных классов хеш-таблиц в .NET Framework можно назвать `Dictionary<TKey,TValue>`, `Hashtable` и `HashSet<T>`. обычно хеш таблицы реализуются как динамический массив записей, каждая из которых содержит связанный список элементов. Чтобы добавить новый элемент в хеш-таблицу, сначала определяется его числовой хеш-код (вызовом метода `GetHashCode`), а затем вызывается хеш-функция, определяющая запись, куда следует поместить элемент. Наконец, элемент добавляется в связанный список выбранной записи.

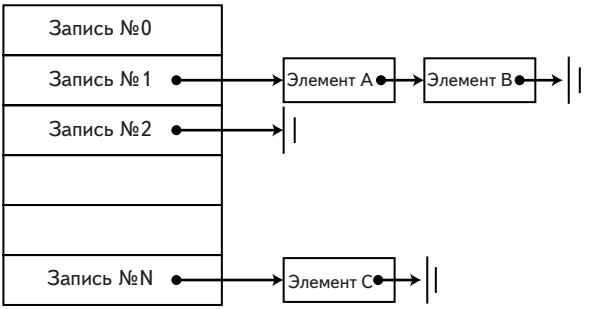

Рис. 3.10. Хеш-таблица, содержащая массив связанных списков (записей), в которых хранятся элементы. Некоторые записи могут быть пустыми, другие могут содержать значительное количество элементов.

Производительность хеш-таблиц в значительной степени зависит от хеш-функции, которая предъявляет определенные требования к методу `GetHashCode`:

1. Если два объекта равны, их хеш-коды так же должны быть равны.
2. Если два объекта не равны, вероятность равенства их хеш-кодов должна быть минимальной.
3. `GetHashCode` должен работать быстро (часто его производительность прямо пропорциональна размеру объекта).
4. Хеш-код объекта не должен изменяться.

Внимание. *Требование (2) не может быть выражено, как «если два объекта не равны, их хеш-коды не должны быть равны», потому что могут существовать типы, для которых возможное количество объектов больше количества целых чисел, когда неизбежно будут существовать объекты с одинаковыми хеш-кодами. Рассмотрим, например, значения типа* `long`*. Всего существует* 2^{64} *различных значений* `long`*, но различных целочисленных значений всего* 2^{32}*, поэтому существует, по крайней мере, одно целочисленное значение, являющееся хеш-кодом для* 2^{32} *различных значений* `long`*!*

Формально условие (2) может быть сформулировано как требование равномерного распределения хеш-кодов: для каждого объекта A функция может существовать множество $S(A)$ всех объектов B таких, что:

- B не равен A;
- хеш-код объекта B равен хеш-коду объекта A.

Условие (2) требует, чтобы размеры множеств *S(A)* для всех объектов *A* были примерно одинаковыми. (Здесь предполагается, что вероятности появления любых объектов *A* примерно одинаковы – что не всегда верно для фактических типов.)

Условия (1) и (2) подчеркивают взаимосвязь между равенством объектов и равенством их хеш-кодов. Если потребуется переопределить и перегрузить виртуальный метод `Equals`, вам придется соответственно изменить реализацию `GetHashCode`. Похоже, что типичная реализация `GetHashCode` должна каким-то образом учитывать поля объекта. Например, лучшая реализация `GetHashCode` для типа `int` должна просто возвращать целочисленное значение. Для объектов `Point2D` можно придумать некоторую линейную комбинацию двух координат или некоторых битов одной координаты с другими битами второй координаты. Проектирование хороших хеш-кодов обычно является сложной задачей, но ее рассмотрение далеко выходит за рамки этой книги.

Наконец перейдем к условию (4). За ним стоят следующие рассуждения: допустим, что имеется точка (5, 5) и ее необходимо добавить в хеш-таблицу, также допустим, что она имеет хеш-код 10. Если координаты точки изменить на (6, 6), ее хеш-код получит значение 12. В этом случае вы не сможете отыскать точку в хеш-таблице. Но это не должно вызывать беспокойства для типов значений, потому что *невозможно* изменить объекты, добавленные в хеш-таблицу – хеш-таблица хранит их *копии*, недоступные для вашего кода.

А что можно сказать о ссылочных типах? Определение равенства для ссылочных типов на основе их содержимого может порождать проблемы. Допустим, что у нас имеется следующая реализация `Employee.GetHashCode`:

```
public class Employee
{
    public string Name { get; set; }
    public override int GetHashCode()
    {
        return Name.GetHashCode();
    }
}
```

Идея формировать хеш-код на основе содержимого объекта выглядит довольно привлекательно, поэтому здесь используется `String.GetHashCode`, что избавляет нас от необходимости реализовать свою хеш-функцию для строк. Однако, взгляните, что произойдет, если изменить значение поля `Employee.Name` после добавления объекта в хеш-таблицу:

```
HashSet<Employee> employees = new HashSet<Employee>();
Employee kate = new Employee { Name = "Kate Jones" };
employees.Add(kate);
kate.Name = "Kate Jones-Smith";
employees.Contains(kate); // вернет false!
```

Хеш-код объекта изменится, потому что изменится его содержимое, и мы больше не сможем найти объект в хеш-таблице. Возможно, это ожидаемо, но проблема в том, что теперь мы не сможем *удалить* Kate из хеш-таблицы, даже при том, что у нас сохранился доступ к оригинальному объекту!

Среда выполнения CLR предоставляет для ссылочных типов реализацию `GetHashCode` по умолчанию, которая основана на идентичности объектов. Если две ссылки на объекты равны и если они ссылаются на один и тот же объект, имеет смысл сохранить хеш-код где-то в самом объекте так, чтобы исключить возможность его изменения и затруднить доступ к нему. В действительности, когда создается экземпляр ссылочного типа, среда выполнения CLR встраивает хеш-код в слово заголовка объекта (для оптимизации, это происходит при первом обращении к хеш-коду; в конце концов, множество объектов никогда не используются в качестве ключей хеш-таблиц). Чтобы вычислить хеш-код, необязательно генерировать случайные числа или учитывать содержимое объекта – достаточно будет простого счетчика.

> **Примечание.** *Как хеш-код может одновременно существовать с индексом блока синхронизации в слове заголовка объекта? Если вы помните, в большинстве объектов слово заголовка никогда не используется для хранения индекса блока синхронизации, потому что они просто не применяются для синхронизации. В редких случаях, когда объект связан с блоком синхронизации, хеш-код копируется в блок синхронизации и хранится там, пока блок синхронизации не будет отсоединен от объекта. Чтобы определить, что хранится в данный момент в слове заголовка, индекс блока синхронизации или хеш-код, один из битов в нем используется в качестве признака.*

Ссылочные типы, использующие реализации `Equals` и `GetHashCode` по умолчанию, не должны заботиться о четырех условиях, описанных выше – они удовлетворяют им бесплатно. Однако, если в вашем ссылочном типе необходимо будет переопределить понятие равенства по умолчанию (как, например, в типе `System.String`), вам следует подумать о возможности сделать свой ссылочный тип неизменяемым, если предполагается возможность использовать его в качестве ключа в хеш-таблице.

Эффективные приемы использования типов значений

Ниже перечислено несколько эффективных приемов, которые следует применять при использовании типов значений для решения некоторых задач:

- используйте типы значений, если объекты достаточно малы и предполагается, что в программе будет создаваться большое их количество;
- используйте типы значений, если требуется высокая плотность размещения их в памяти;
- переопределяйте `Equals`, определяйте перегруженные версии `Equals`, реализуйте интерфейс `IEquatable<T>`, перегружайте операторы == и != в своих типах значений;
- перегружайте `GetHashCode` в своих типах значений;
- подумайте о возможности сделать свои типы значений неизменяемыми.

В заключение

В этой главе мы рассмотрели особенности реализации ссылочных типов и типов значений, и как они влияют на производительность приложения. Типы значений обеспечивают высокую плотность размещения в памяти, что делает их превосходными кандидатами для размещения в больших коллекциях, но они не поддерживают такие особенности, свойственные объектам, как полиморфизм, возможность применения для синхронизации и семантика ссылок. Среда выполнения CLR вводит две категории типов, давая возможность выбирать между высокопроизводительными альтернативами, когда это необходимо, но все еще требует от разработчика прикладывать огромные усилия для корректной реализации типов значений.

ГЛАВА 4.
Сборка мусора

В этой главе мы займемся исследованием сборщика мусора в .NET (Garbage Collector, GC), одного из основных механизмов, оказывающих существенное влияние на производительность приложений для .NET. Освобождая разработчиков от хлопот, связанных с освобождением памяти, сборщик мусора вводит другие проблемы, которые необходимо учитывать при разработке программ, производительность которых имеет большое значение. Сначала мы познакомимся с разновидностями сборщиков мусора, доступных в CLR, и посмотрим, насколько настройка их поведения может влиять на производительность приложения. Затем мы узнаем, какое влияние на производительность сборщика мусора оказывают разные поколения объектов, и какие настройки можно выполнить в приложении. Ближе к концу главы мы исследуем прикладной программный интерфейс непосредственного управления сборщиком мусора, а также некоторые тонкости, связанные с финализацией (finalization).

Многие примеры в этой главе основаны на личном опыте авторов работы с действующими системами. Везде, где это возможно, мы будем приводить короткие примеры, и даже небольшие приложения, иллюстрирующие основные «болевые точки» производительности, с которыми вы сможете поэкспериментировать в процессе чтения данной главы. Раздел «Эффективные приемы повышения производительности сборки мусора», в конце главы, наполнен такими примерами. Однако вы должны понимать, что подобные «болевые точки» очень сложно продемонстрировать в коротких фрагментах кода и ничуть не легче в небольших примерах программ, потому что обычно разница в производительности становится особенно заметной в крупных проектах, имеющих тысячи типов и миллионы объектов в памяти.

Назначение сборщика мусора

Сборка мусора – это высокоуровневая абстракция, избавляющая разработчиков от необходимости заботиться об освобождении управляемой памяти. В окружениях, снабженных механизмом сборки мусора, выделение памяти производится в момент создания объектов, а освобождение происходит, когда в программе исчезает последняя ссылка на объект. Кроме того, сборщик мусора предоставляет интерфейс финализации для неуправляемых ресурсов, находящихся за пределами управляемой динамической памяти, благодаря чему имеется возможность обеспечить выполнение кода, когда эти ресурсы окажутся не нужны. При создании сборщика мусора в .NET преследовались две основные цели:

- избавиться от ошибок и ловушек, связанных с управлением памятью вручную;
- обеспечить производительность операций управления памятью, равную или превышающую производительность ручных механизмов.

В существующих языках программирования и фреймворках используются различные стратегии управления памятью. Мы коротко исследуем две из них: управление на основе списка свободных блоков (реализацию которой можно найти в коллекции стандартных инструментов управления памятью языка C) и сборка мусора на основе подсчета ссылок.

Управление свободным списком

Управление на основе списка свободных блоков – это механизм управления распределением памяти в стандартной библиотеке языка C, который также по умолчанию используется функциями управления памятью в C++, такими как `new` и `delete`. Это детерминированный диспетчер памяти, при использовании которого вся ответственность за выделение и освобождение памяти ложится на плечи разработчика. Свободные блоки памяти хранятся в виде связанного списка, откуда изымаются блоки памяти при выделении (рис. 4.1), и куда они возвращаются, при освобождении.

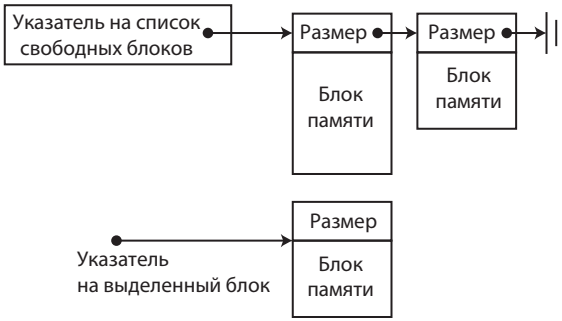

Рис. 4.1. Диспетчер памяти хранит список свободных блоков. Он изымает блоки из списка при выделении памяти и возвращает их обратно при освобождении. Приложение обычно получает блоки памяти, хранящие их размеры в служебной области.

Механизм управления памятью на основе списка не свободен от тактических и стратегических решений, влияющих на производительность приложения. Ниже перечислены некоторые из них.

- Приложение, использующее механизм управления памятью на основе списка свободных блоков, изначально получает небольшой пул свободных блоков, организованных в виде списка. Список может быть отсортирован по размеру, по времени использования и так далее.
- Когда диспетчер получает от приложения запрос на выделение памяти, он выполняет поиск соответствующего блока памяти. Соответствие может определяться по принципу «первый подходящий», «лучше подходящий» или с применением более сложных критериев.
- После исчерпания списка диспетчер запрашивает у операционной системы дополнительные свободные блоки и добавляет их в список. Когда приложение возвращает память диспетчеру, он добавляет освободившийся блок в список. На этом этапе может выполняться слияние смежных свободных блоков, дефрагментация и сокращение списка, и так далее.

Ниже перечислены основные проблемы, связанные с управлением памятью на основе списка свободных блоков.

- Высокая стоимость операции выделения: поиск блока, соответствующего параметрам запроса, требует времени, даже при использовании критерия «первый подходящий». Кроме того, блоки часто разбиваются на несколько частей. В многопроцес-

сорных системах неизбежно возникает конкуренция за список и необходимость синхронизации операций, если только не используется несколько списков. С другой стороны, наличие нескольких списков ухудшает их фрагментацию.

- Высокая стоимость освобождения: возврат блока в список требует времени, и здесь снова возникает проблема синхронизации конкурирующих операций освобождения памяти.
- Высокая стоимость управления: Чтобы избежать ситуации отсутствия блоков памяти подходящего размера при наличии большого количества маленьких блоков, необходимо выполнять дефрагментацию списка. Но эта работа должна производиться в отдельном потоке выполнения, что опять же требует применения блокировок для доступа к списку и снижает скорость операций выделения и освобождения памяти. Фрагментацию можно уменьшить, выделяя блоки фиксированного размера и поддерживая несколько списков, но при этом увеличивается количество операций по поддержанию динамической памяти в целостном состоянии и добавляет накладные расходы к каждой операции выделения и освобождения памяти.

Сборка мусора на основе подсчета ссылок

Сборщик мусора, опирающийся на подсчет ссылок, связывает каждый объект с целочисленной переменной – счетчиком ссылок. В момент создания объекта счетчик ссылок инициализируется значением 1. Когда приложение создает новую ссылку на объект, его счетчик ссылок увеличивается на 1 (рис. 4.2). Когда приложение удаляет ссылку на существующий объект, его счетчик ссылок уменьшается на 1. Когда счетчик ссылок достигает значения 0, объект можно уничтожить и освободить занимаемую им память.

Одним из примеров управления памятью на основе подсчета ссылок в экосистеме Windows является объектная модель программных компонентов (Component Object Model, COM). Объекты COM снабжаются счетчиками ссылок, определяющими продолжительность их существования. Когда значение счетчика ссылок достигает 0, объект может освободить занимаемую им память. Основное бремя подсчета ссылок ложится на плечи разработчика, в виде явного вызова методов `AddRef` и `Release`, хотя в большинстве языков имеются обертки, автоматизирующие вызовы этих методов при создании и удалении ссылок.

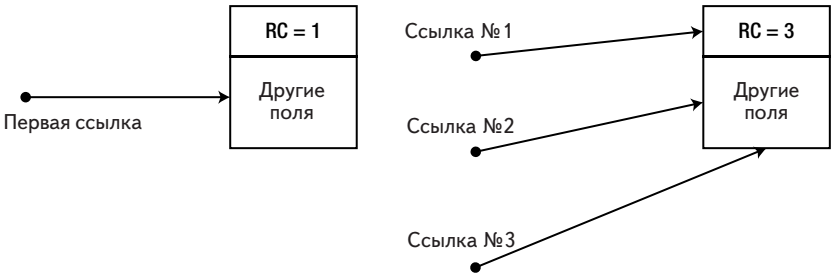

Рис. 4.2. Каждый объект содержит счетчик ссылок.

Ниже перечислены основные проблемы, связанные с управлением памятью на основе подсчета ссылок.

- Высокая стоимость управления: всякий раз, когда создается или уничтожается ссылка на объект, необходимо обновлять счетчик ссылок. Это означает, что к стоимости обновления ссылки прибавляются накладные расходы на выполнение таких тривиальных операций, как присваивание ссылки (`a = b`) или передача ссылки в функцию по значению. В многопроцессорных системах выполнение таких операций требует применения механизмов синхронизации и вызывает «пробуксовку» кеша процессора, при попытке нескольких потоков выполнения одновременно изменить счетчик ссылок. (Подробнее о проблемах кеширования в одно- и многопроцессорных системах рассказывается в главах 5 и 6.)
- Использование памяти: Счетчик ссылок на объект должен храниться в памяти объекта. Это на несколько байтов увеличивает объем памяти, занимаемой объектом, что делает подсчет ссылок нецелесообразным для легковесных объектов. (Впрочем, это не такая большая проблема для CLR, где к каждому объекту «в нагрузку» добавляется от 8 до 16 байт, как было показано в главе 3.)
- Правильность: при управлении памятью на основе подсчета ссылок возникает проблема утилизации объектов с циклическими ссылками. Если приложение больше не ссылается на некоторую пару объектов, но каждый из них продолжает хранить ссылку на другой объект (рис. 4.3), возникает утечка памяти. Эта проблема описывается в документации COM, где явно оговаривается, что такие циклические ссылки должны уничтожаться вручную. Другие платформы, такие как язык прог-

раммирования Python, поддерживают дополнительные механизмы определения циклических ссылок и их устранения, применение которых влечет за собой увеличение стоимости сборки мусора.

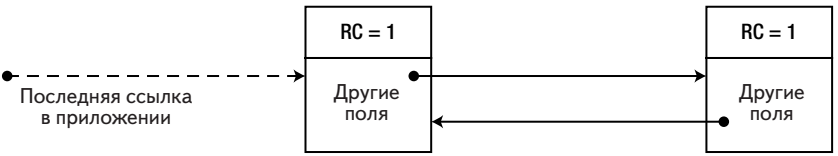

Рис. 4.3. Когда в приложении уничтожаются все ссылки на циклические объекты, их внутренние счетчики ссылок остаются равными 1 и не могут быть утилизированы сборщиком мусора, что вызывает утечку памяти. (Ссылка, обозначенная пунктиром – это уничтоженная ссылка.)

Сборка мусора на основе трассировки

Сборка мусора на основе трассировки применяется для управления динамической памятью в .NET CLR, Java VM и в других управляемых окружениях. В этих окружениях не используются счетчики ссылок, ни в какой форме. Разработчику не требуется явно освобождать память – об этом позаботится сборщик мусора. При использовании сборки мусора на основе трассировки в объекты не добавляются счетчики ссылок, и обычно освобождение памяти не выполняется, пока уровень ее использования не достигнет некоторого порога.

Когда запускается цикл сборки мусора, он начинается с *фазы маркировки* (mark phase) , в ходе которой выявляются все объекты, на которые существуют ссылки в приложении (живые объекты). После определения множества живых объектов, сборщик приступает к выполнению *фазы чистки* (sweep phase), в ходе которой он освобождает память, занимаемую неиспользуемыми объектами. В заключение выполняется *фаза сжатия* (compact phase), в которой сборщик мусора перемещает живые объекты так, чтобы они располагались в памяти непосредственно друг за другом.

В этой главе мы подробно исследуем различные проблемы, связанные с трассировочной сборкой мусора. Однако в общих чертах эти проблемы можно обозначить уже сейчас.

- Стоимость выделения памяти: стоимость выделения памяти сопоставима с выделением кадра на стеке, потому что отсутствует необходимость предусматривать сохранение какой-либо информации, которая потребуется при освобождении объекта. Выделение памяти заключается в наращивании указателя.
- Стоимость освобождения памяти: сборка мусора выполняется циклически, а не распределяется примерно равномерно по всему времени выполнения приложения. Это имеет свои преимущества и недостатки (в частности для ситуаций, когда требуется низкое время задержки), о которых будет рассказываться далее в этой главе.
- Фаза маркировки: поиск объектов по ссылкам требует значительных затрат от среды выполнения. Ссылки на объекты могут храниться в статических переменных, локальных переменных потоков выполнения, передаваемых как указатели в неуправляемый код, и так далее. Трассировка всех возможных ссылок на каждый доступный объект – далеко не тривиальная задача, и часто сопряжена с затратами на итерации через коллекции.
- Фаза чистки: перемещение объектов в памяти требует времени, что может оказаться невозможным для очень больших объектов. С другой стороны, устранение пустых, неиспользуемых пространств между объектами способствует локальности ссылок (locality of reference), потому что объекты, создававшиеся одновременно, наверняка окажутся в памяти по соседству. Кроме того, выполнение этого этапа избавляет от необходимости использовать дополнительный механизм дефрагментации, так как объекты всегда занимают непрерывную область памяти, без разрывов. Наконец, это означает, что механизм выделения памяти не должен учитывать «дырки» между объектами при поиске свободного места – можно использовать простую стратегию, основанную на увеличении указателя.

В последующих разделах мы исследуем парадигму управления памятью в .NET, начав с изучения этапов маркировки и чистки, а затем перейдем к знакомству с более сложными оптимизациями, такими как поколения объектов.

Фаза маркировки

На этапе маркировки, сборщик мусора выполняет обход графа всех объектов, на которые ссылается приложение. Для успешного обхода

графа и предотвращения ошибок первого (false positives) и второго (false negatives) рода (обсуждается ниже в этой главе), сборщику мусора необходимо множество опорных точек, откуда можно будет начать обход ссылок. Эти отправные точки называют корнями (roots). Как можно заключить из названия, они образуют корни ориентированных графов ссылок.

После определения множества корней, остальные операции, выполняемые сборщиком мусора на этапе маркировки, просты и понятны. Он просматривает каждое внутреннее ссылочное поле в каждом корне и выполняет обход графа, пока не посетит все объекты. Поскольку в приложениях для .NET допускается создавать циклические ссылки, сборщик мусора маркирует посещенные им объекты, чтобы впредь не задерживаться на них – отсюда и происходит название фаза маркировки.

Локальные корни

Наиболее типичными представителями корней являются локальные переменные; единственная локальная переменная может быть корнем целого графа объектов. Например, взгляните на следующий фрагмент кода в теле объекта `Main` приложения, который создает объект `System.Xml.XmlDocument` и вызывает его метод `Load`:

```
static void Main(string[] args) {
    XmlDocument doc = new XmlDocument();
    doc.Load("Data.xml");
    Console.WriteLine(doc.OuterXml);
}
```

Мы не контролируем момент запуска сборщика мусора и поэтому должны иметь в виду, что сборка мусора может начаться в процессе выполнения метода `Load`. В этом случае не хотелось бы, чтобы объект `XmlDocument` был утилизирован – локальная ссылка в методе `Main` является корнем графа объекта документа, которую сборщик мусора обязательно должен исследовать. Поэтому всякая локальная переменная, которая потенциально может хранить ссылку на объект (то есть, локальная переменная ссылочного типа), считается активным корнем, пока кадр стека метода не будет разрушен.

Однако не всегда требуется, чтобы ссылка оставалась активным корнем до выхода из метода. Например, после загрузки и отображения документа, нам может понадобиться добавить в тот же метод дополнительный код, не требующий сохранения документа в памяти. Этот код может выполняться очень долго и если в этом промежутке

времени начнется новый цикл сборки мусора, было бы желательно, чтобы память, занимаемая документом, была освобождена.

Поддерживает ли сборщик мусора в .NET такую возможность? Давайте рассмотрим следующий код, создающий объект `System.Threading.Timer` и инициализирующий его функцией обратного вызова, которая вызывает сборку мусора обращением к методу `GC.Collect` (ниже мы познакомимся с программным интерфейсом сборщика мусора более подробно):

```
using System;
using System.Threading;

class Program {
    static void Main(string[] args) {
        Timer timer = new Timer(OnTimer, null, 0, 1000);
        Console.ReadLine();
    }

    static void OnTimer(object state) {
        Console.WriteLine(DateTime.Now.TimeOfDay);
        GC.Collect();
    }
}
```

Если запустить этот код в отладочном режиме (скомпилировав его из командной строки, вызвав компилятор без ключа `/optimize +`), можно будет увидеть, что функция `OnTimer` вызывается каждую секунду, как и следовало ожидать, явно свидетельствуя, что объект `Timer` не утилизируется сборщиком мусора. Однако, если скомпилировать программу с ключом `/optimize +`, функция будет вызвана всего один раз! Иными словами, объект `Timer` будет утилизирован и прекратит вызывать функцию обратного вызова. Это вполне обычное (и даже желаемое) поведение, потому что локальная ссылка на объект не может больше рассматриваться как активный корень сразу после достижения инструкции вызова метода `Console.ReadLine`. По этой причине объект утилизируется сборщиком мусора, и получается результат, неожиданный для тех, кто не следил за нашим обсуждением!

Энергичная сборка мусора

Такое «нетерпение» по отношению к локальным корням в действительности поддерживается динамическим компилятором (Just-In-Time Compiler, JIT) среды выполнения.NET. Сборщик мусора понятия не имеет, когда локальная переменная выходит из употребления в пределах метода. Эта информация записывается в специальные таблицы JIT-компилятором

в процессе компиляции метода. Для каждой локальной переменной JIT-компилятор записывает в таблицу адрес самой первой и самой последней инструкции, где локальная переменная остается активным корнем. Эта таблица используется сборщиком мусора в процессе обхода ссылок на стеке. (Обратите внимание, что локальные переменные могут храниться не только на стеке, но и в регистрах процессора, и таблицы JIT-компилятора должны содержать информацию об этом.)

```
// Код на C#:
static void Main() {
    Widget a = new Widget();
    a.Use();
    //...дополнительный код
    Widget b = new Widget();
    b.Use();
    //...дополнительный код
    Foo(); //вызов статического метода
}

// Скомпилированный код на языке ассемблера x86:
      ; пролог метода опущен для краткости
      call 0x0a890a30              ; Widget..ctor
+0x14 mov esi, eax                 ; esi теперь хранит ссылку
                                   ; на объект
      mov ecx, esi
      mov eax, dword ptr [ecx]
      ; остальные инструкции, составляющие последовательность
      ; вызова функции
+0x24 mov dword ptr [ebp-12], eax ; ebp-12 теперь хранит
                                   ; ссылку на объект
      mov ecx, dword ptr [ebp-12]
      mov eax, dword ptr [ecx]
      ; остальные инструкции, составляющие последовательность
      ; вызова функции
+0x34 call 0x0a880480 ; Foo method call
      ; эпилог метода опущен для краткости

// Таблицы, сгенерированные JIT-компилятором и используемые
// сборщиком мусора:
Регистр или стек      Смещение начала      Смещение конца
     ESI                   0x14                 0x24
  EBP - 12                 0x24                 0x34
```

Этот пример показывает, что разбиение кода на небольшие методы и уменьшение количества локальных переменных не только является хорошим стилем проектирования и разработки программного обеспечения. В .NET это обеспечивает более высокую производительность, потому что в программе образуется меньше локальных корней и, соответственно, JIT-компилятору приходится выполнять меньше работы, таблицы с адресами инструкций занимают меньше места и уменьшается нагрузка на сборщика мусора, когда он выполняет обход стека.

А как быть, если потребуется явно продлить жизнь таймера до конца выполнения метода? Сделать это можно несколькими способами. Можно было бы использовать статическую переменную (которая является еще одной разновидностью корня, о которой будет рассказываться чуть ниже). Как вариант, в конец метода можно поместить инструкцию использования таймера (например, вызвать метод `timer.Dispose()`). Но наиболее очевидный путь достижения цели – вызвать метод `GC.KeepAlive`, гарантирующий сохранность ссылки на объект.

Как действует `GC.KeepAlive`? Кому-то он может показаться неким таинством, скрытым в недрах CLR. Однако в действительности все гораздо проще – его можно написать самому, что мы и сделаем. Если передать ссылку какому-либо методу, которые не может быть встроен (как описывается в главе 3), JIT-компилятор автоматически будет предполагать, что объект где-то используется. То есть, следующий метод вполне сгодится на роль `GC.KeepAlive`:

```
[MethodImpl(MethodImplOptions.NoInlining)]
static void MyKeepAlive(object obj) {
    // Преднамеренно оставлен пустым: метод не выполняет никаких операций
}
```

Статические корни

Еще одной разновидностью корней являются статические переменные. Статические члены типов создаются в момент их загрузки (мы наблюдали этот процесс в главе 3) и считаются активными корнями на протяжении всего времени выполнения приложения. Например, взгляните на следующую небольшую программу, непрерывно создающую объекты, которые регистрируют статическое событие:

```
class Button {
    public void OnClick(object sender, EventArgs e) {
        // Реализация отсутствует
    }
}

class Program {
    static event EventHandler ButtonClick;

    static void Main(string[] args) {
        while (true) {
            Button button = new Button();
            ButtonClick += button.OnClick;
        }
    }
}
```

Эта программа страдает утечкой памяти, потому что статическое событие содержит список делегатов, которые в свою очередь ссылаются на создаваемые программой объекты. Фактически, одной из наиболее типичных для .NET причин утечек памяти является сохранение ссылок на объекты в статических переменных!

Другие корни

Две категории корней, описанные выше, являются наиболее распространенными, однако существуют и другие категории. Например, дескрипторы сборщика мусора (представленные типом `System.Runtime.InteropServices.GCHandle`) также интерпретируются как корни. Очередь объектов, готовых к завершению (f-reachable queue), – еще один пример корня – объекты, ожидающие финализации, все еще считаются досягаемыми для сборщика мусора. Обе разновидности корней обсуждаются далее в этой главе; знание различных категорий корней может пригодиться при отладке утечек памяти в приложениях для .NET, потому что очень часто в отсутствие простейших (читайте: локальных и статических) корней, ссылающихся на ваши объекты, они продолжают сохраняться в памяти по какой-то другой причине.

Исследование корней с помощью SOS.DLL

Для исследования цепочек корней, удерживающих определенные объекты в памяти, можно использовать библиотеку SOS.DLL, расширение отладчика, с которой мы познакомились в главе 3. Команда `!gcroot` библиотеки позволяет получить исчерпывающую информацию о типах корней и цепочках ссылок. Ниже приводится пример вывода команды:

```
0:004> !gcroot 02512370
HandleTable:
    001513ec (pinned handle)
    -> 03513310 System.Object[]
    -> 0251237c System.EventHandler
    -> 02512370 Employee

0:004> !gcroot 0251239c
Thread 3068:
    003df31c 002900dc Program.Main(System.String[]) [d:\...\
Ch04\Program.cs @ 38]
        esi:
            -> 0251239c Employee

0:004> !gcroot 0227239c
Finalizer Queue:
    0227239c
    -> 0227239c Employee
```

Первый корень в примере выше, это, скорее всего, статическое поле – чтобы точно выяснить это, потребуется приложить некоторые усилия. Так или иначе, это связанный дескриптор сборщика мусора (дескрипторы описываются далее в этой главе). Второй корень – регистр `ESI` в потоке выполнения с числовым идентификатором 3068, где хранится локальная переменная метода `Main`. Последний корень – очередь объектов, готовых к завершению (f-reachable queue).

Вопросы производительности

Фаза маркировки в цикле сборки мусора является фазой «преимущественно для чтения», в процессе выполнения которой не производится перемещение объектов в памяти или их удаление. Тем не менее, эта фаза оказывает существенное влияние на производительность:

- В процессе маркировки сборщик мусора должен посетить каждую ссылку на объект. Это может приводить к ошибкам чтения страниц памяти, если они оказываются вытеснены в файл подкачки, а также к промахам кеша и перезагрузке кеша процессора.
- В многопроцессорной системе, из-за того, что сборщик мусора маркирует объекты, взводя флаги в их заголовках, это может приводить к недостоверности кешей других процессоров, хранящих эти объекты.
- Обработка неиспользуемых объектов оказывается менее дорогостоящей на этом этапе, поэтому производительность фазы маркировки прямо пропорциональна коэффициенту эффективности сборки мусора: отношению количества используемых и неиспользуемых объектов в памяти.
- Производительность фазы маркировки зависит также от количества объектов в графе, но не зависит от объема памяти, занимаемой этими объектами. Большие объекты, содержащие мало ссылок, влекут меньше накладных расходов. А это означает, что производительность фазы маркировки прямо пропорциональная количеству живых объектов в графе.

По завершении фазы маркировки сборщик мусора получает полный граф всех используемых объектов и ссылок на них (рис. 4.4). Теперь он может перейти к фазе чистки.

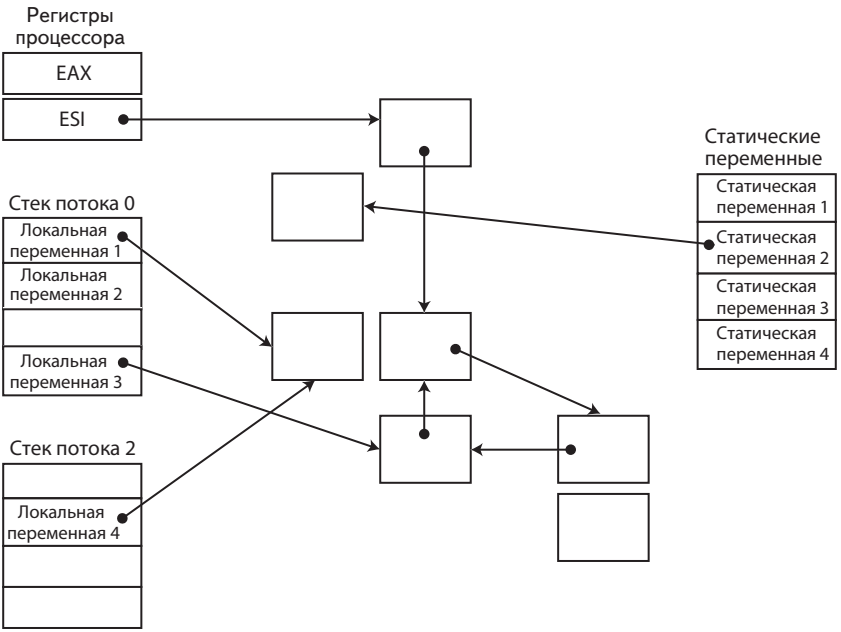

Рис. 4.4. Граф объектов с корнями различных типов. Циклические ссылки считаются допустимыми.

Фазы чистки и сжатия

В фазах чистки и сжатия сборщик мусора освобождает память, часто перемещая живые объекты так, чтобы они располагались в динамической памяти последовательно, друг за другом. Прежде чем разбираться с механикой перемещения объектов необходимо исследовать механизм выделения памяти, обеспечивающий сборщик мусора работой в фазе чистки.

В простейшей модели, исследованием которой мы сейчас занимаемся, выделение памяти выполняется простым наращиванием указателя, который всегда ссылается на следующий свободный блок памяти (рис. 4.5). Этот указатель называется указателем на следующий объект (next object pointer), или указателем на новый объект (new object pointer) и инициализируется, когда на этапе запуска приложению передается блок динамической памяти.

Выделение памяти в этой модели выполняется чрезвычайно быстро: достаточно выполнить единственную атомарную операцию уве-

личения указателя. В многопроцессорных системах наверняка будут возникать конкуренция за право доступа к этому указателю (о чем будет рассказываться далее в этой главе).

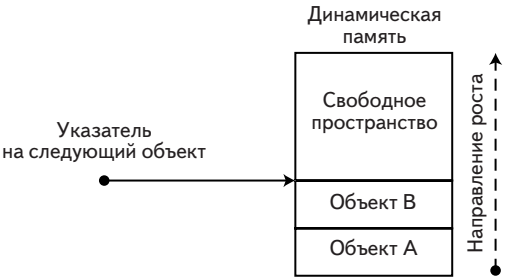

Рис. 4.5. Динамическая память и указатель на следующий объект.

Если бы память была неисчерпаемым ресурсом, запросы на выделение памяти можно было бы удовлетворять до бесконечности, просто увеличивая указатель на следующий объект. Однако в действительности в некоторый момент времени будет достигнут определенный порог, за которым последует принудительный запуск процедуры сборки мусора. Пороговые значения являются динамическими и доступны для настройки – подробнее о способах управления ими будет рассказываться далее в этой главе.

В фазе сжатия сборщик мусора перемещает живые объекты в памяти, размещая их рядом друг с другом (рис. 4.6). Это способствует локальности ссылок, потому что объекты, создававшиеся одновременно, наверняка окажутся в памяти по соседству. С другой стороны, перемещение объектов влечет за собой две проблемы производительности.

- Перемещение объектов означает необходимость копирования блоков памяти, что для крупных объектов оказывается весьма дорогим удовольствием. Даже при использовании оптимизированной операции копирования перемещение нескольких мегабайтов данных в каждом цикле сборки мусора приведет к значительным непроизводительным расходам. (Именно поэтому большие объекты обслуживаются иначе, как будет показано дальше.)
- При перемещении объектов необходимо обновлять ссылки на них соответственно их новому местоположению. Для широко используемых объектов, обновление большого количества ссылок может оказаться дорогостоящей операцией.

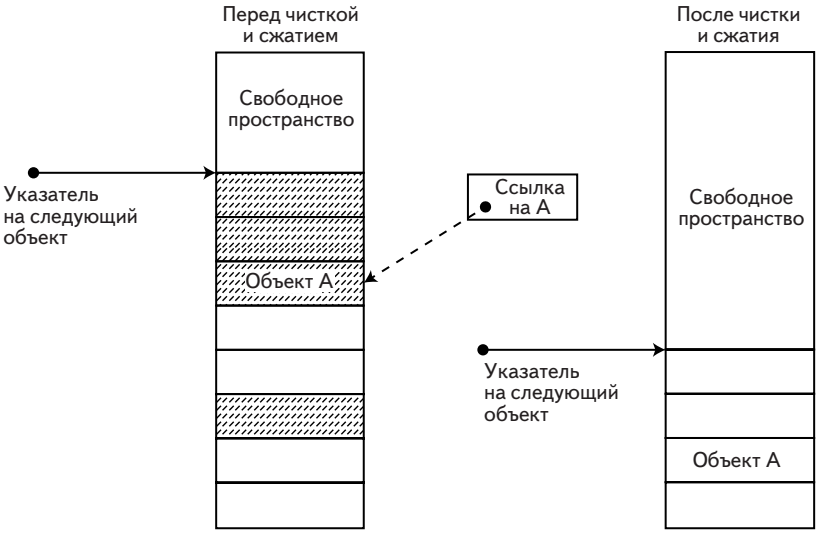

Рис. 4.6. Заштрихованные объекты (слева) продолжат существование после сборки мусора и будут сдвинуты ближе к началу пула динамической памяти. Это означает, например, что ссылка на объект А (пунктирная линия) будет обновлена. (Процесс обновления ссылок не показан здесь.)

Общая производительность фазы чистки прямо пропорциональна количеству объектов в графе и в значительной степени зависит от коэффициента эффективности сборки мусора. Если большинство составляют неиспользуемые объекты, тогда сборщику мусора придется перемещать лишь небольшое количество остающихся объектов. То же самое верно, когда большинство составляют используемые объекты, так как придется заполнить небольшое количество «дырок». С другой стороны, если неиспользуемым окажется каждый второй объект, может так получиться, что сборщику мусора придется перемещать каждый живой объект.

Примечание. *Вопреки распространенному мнению, сборщик мусора не всегда перемещает объекты (то есть, некоторые циклы сборки мусора завершаются сразу после фазы чистки, не достигая фазы сжатия), даже если они не закреплены (см. ниже) и между ними имеется свободное пространство. Существуют определенные эвристики, позволяющие оценить необходимость перемещения объектов после фазы чистки. Например, а одном из испытательных тестов, выполнявшемся в 32-разрядной системе автора, сборщик мусора решил, что объекты должны перемещаться, если свободное пространство между ними превышает 16 байт, включает более одного*

объекта, подлежащих утилизации, и с момента последнего цикла сборки мусора было выделено боле 16 Кбайт памяти. Вы не должны полагаться на эти данные, но они демонстрируют наличие некоторой оптимизации.

В описании модели маркировки и чистки, приведенном в предыдущих разделах, имеется существенный пробел, который мы восполним далее в этой главе, когда перейдем к обсуждению модели поколений. Всякий раз, когда запускается процедура сборки мусора, выполняется обход всех объектов в динамической памяти, независимо от соотношения количества живых объектов и объектов, подлежащих утилизации. Имея эту информацию, можно было бы настроить алгоритм сборки мусора и уменьшить ее дороговизну.

Закрепление

Модель сборки мусора, представленная выше, не учитывает один из типичных случаев использования управляемых объектов. В данном случае речь идет о передаче управляемых объектов неуправляемому коду. Решить эту проблему можно двумя способами:

- все объекты можно передавать неуправляемому коду по значению (то есть, копировать) и возвращать так же по значению;
- чтобы избежать копирования, объекты можно передавать по ссылке.

Копирование блоков памяти при каждом взаимодействии с неуправляемым кодом может оказаться непозволительной роскошью. Представьте программу, обрабатывающую видеоинформацию в действительном масштабе времени, которой требуется передавать изображения с высоким разрешением между управляемым и неуправляемым кодом со скоростью 30 кадров в секунду. Копирование множества мегабайтов памяти ради незначительных изменений ухудшит производительность до неприемлемого уровня.

Модель управления памятью в .NET поддерживает возможность получения адресов управляемых объектов в памяти. Однако передача этих адресов неуправляемому коду в присутствии сборщика мусора порождает серьезную проблему: представьте, что произойдет, если сборщик мусора переместит объект, пока неуправляемый код выполняет его обработку и продолжает использовать прежний адрес?

Последствия этого могут быть самыми катастрофическими – неуправляемый код легко может повредить данные в памяти. Одним из надежных решений этой проблемы является отключение сборщика мусора на время, пока неуправляемый код обладает указателем на

управляемый объект. Однако такой подход никуда не годится в ситуациях, когда объекты часто передаются между управляемым и неуправляемым кодом. Кроме того, это грозит появлением ситуации взаимоблокировки, если поток выполнения неуправляемого кода войдет в длительный цикл ожидания.

Вместо отключения сборщика мусора, любой управляемый объект, адрес которого может передаваться неуправляемому коду, следует закрепить в памяти. Закрепленные объекты не будут перемещаться сборщиком мусора до их открепления.

Сама операция закрепления стоит не очень дорого – существует несколько механизмов, выполняющих ее очень быстро. Наиболее явным способом закрепления объекта является создание дескриптора сборщика мусора с флагом `GCHandleType.Pinned`. В результате создается новый корень в таблице дескрипторов, сообщающий сборщику мусора, что объект должен оставаться прикрепленным к определенному адресу в памяти. В число альтернатив входят магическая приправа, добавляемая маршалером P/Invoke, и механизм закрепленных указателей, предоставляемый языком C# в виде ключевого слова `fixed` (или типа `pin_ptr<T>` в C++/CLI), который опирается на специальную маркировку локальных переменных. (Подробности описываются в главе 8.)

Однако потери производительности, обусловленные наличием закрепленных объектов, становятся более очевидными, если посмотреть, какое влияние они оказывают на работу сборщика мусора. Встретив закрепленный объект на этапе сжатия, сборщик мусора должен пропустить его, чтобы гарантировать постоянное местоположение в памяти. Это усложняет алгоритм сборки, но самым неприятным эффектом является фрагментация динамической памяти. Сильная фрагментация динамической памяти сводит на нет оптимизации, ускоряющие сборку: она препятствует последовательному размещению объектов в памяти (что отрицательно сказывается локальности ссылок), вносит дополнительные сложности в процедуру выделения памяти и вызывает непроизводительный расход памяти из-за невозможности заполнить «дыры» между объектами.

> **Примечание.** *Побочные эффекты, вызываемые закреплением объектов в памяти, можно наблюдать с помощью множества разных инструментов, включая профилировщик Microsoft CLR Profiler. Этот профилировщик может отображать графы объектов с адресами, отмечая свободные (фрагментированные) области, как пустое пространство. Аналогично библиотека SOS.DLL (расширение управляемого отладчика) способна отображать «дыры», образовавшиеся в результате*

фрагментации, как объекты типа «Free». Наконец, определить количество закрепленных объектов в последней области, исследованной сборщиком мусора, можно с помощью счетчика «# of Pinned Objects» («Закрепленных объектов») из категории «.NET CLR Memory» («Память CLR .NET»).

Несмотря на перечисленные недостатки, закрепление объектов является насущной необходимостью во многих приложениях. Обычно нет нужды управлять закреплением явно – для этого используется уровень абстракции (такой как P/Invoke), который автоматически выполняет все необходимые операции. Далее в этой главе мы познакомимся с некоторыми рекомендациями, позволяющими уменьшить отрицательное влияние закрепления объектов.

Итак, мы познакомились с основными этапами работы сборщика мусора. Мы также узнали, что может произойти с объектами, которые должны передаваться неуправляемому коду. На протяжении предыдущих разделов мы увидели множество мест, где проявляются различные оптимизации. Но чаще других упоминалась мысль о том, что в многопроцессорных системах конкуренция и необходимость синхронизации могут оказаться весьма важным фактором, влияющим на производительность приложений, интенсивно работающих с памятью. В последующих разделах мы исследуем различные оптимизации, в том числе и предназначенные для многопроцессорных систем.

Разновидности сборщиков мусора

Среда выполнения .NET поддерживает несколько разновидностей сборщиков мусора, даже при том, что внешне она выглядит как огромный монолит кода с ограниченными возможностями настройки. Эти разновидности предназначены для использования в разных ситуациях: в клиентских приложениях, в высокопроизводительных серверных приложениях, и так далее. Чтобы разобраться в отличительных чертах этих разновидностей, необходимо посмотреть, как сборщик мусора взаимодействует с прикладными потоками выполнения (часто называемыми потоками-мутаторами (mutator threads)).

Приостановка потоков для сборки мусора

Сборка мусора обычно запускается, когда прикладные потоки выполнения уже работают. В конце концов, запуск сборки мусора является

результатом операций выделения памяти в приложении. Сборщик мусора оказывает влияние на местоположение объектов в памяти и на ссылки на эти объекты. Перемещение объектов в памяти и изменение ссылок на них в то время, как прикладной код выполняет операции с ними, может вызывать ошибки.

С другой стороны, в некоторых ситуациях очень важно обеспечить возможность одновременного выполнения сборщика мусора с другими прикладными потоками выполнения. Представьте, например, классическое приложение с графическим интерфейсом. Пока сборка мусора выполняется в фоновом потоке, было бы желательно сохранить отзывчивость интерфейса. Даже если сама сборка будет занимать больше времени (из-за состязания за вычислительные ресурсы с потоком выполнения, обслуживающим интерфейс), пользователь будет чувствовать себя комфортнее, если приложение будет отзывчивее.

Существует две категории проблем, возникающих, когда сборщик мусора начинает конкурировать с прикладными потоками выполнения.

- Ошибка первого рода: когда объект принимается за неиспользуемый, хотя в программе имеются ссылки на него. Отладка этой проблемы может превратиться в сущий кошмар, поэтому сборщик мусора должен сделать все возможное, чтобы предотвратить ее.
- Ошибка второго рода: когда объект, пригодный для утилизации, ошибочно принимается за живой. Эта проблема, конечно, нежелательная, но если объект будет утилизирован в следующем цикле, с ней можно мириться.

Вернемся опять к фазам маркировки и чистки в работе сборщика мусора и посмотрим, можно ли обеспечить одновременную работу прикладных потоков выполнения и сборщика мусора. Обратите внимание, что к каким бы заключениям мы ни пришли, все еще существуют моменты, когда требуется приостанавливать прикладные потоки во время сборки мусора. Например, если процесс действительно исчерпал доступную ему память, его придется приостановить, пока сборщик мусора не освободит требуемый объем. Впрочем, мы будем рассматривать менее серьезные ситуации, составляющие подавляющее большинство.

Приостановка потоков на время работы сборщика мусора

Приостановка прикладных потоков выполнения на время работы сборщика мусора выполняется в *безопасных точках* (safe points). Не каждая инст-

рукция может быть прервана, чтобы дать сборщику мусора возможность выполнить свою работу. JIT-компилятор генерирует информацию о точках в выполняемом коде, где можно безопасно приостановить поток и приступить к сборке мусора, и среда выполнения CLR старается приостанавливать потоки именно в таких точках – она не будет вызывать `SuspendThread`, не убедившись, что это безопасно для потока.

В CLR 2.0 были возможны ситуации, когда управляемый поток, выполняющий массивные вычисления в глубоком цикле, мог не достигать безопасной точки длительное время, вызывая задержки в работе сборщика мусора до 1500 миллисекунд (что, в свою очередь, вызывало задержки в работе потоков, приостановленных сборщиком мусора). Эта проблема была исправлена в версии CLR 4.0; тем, кому интересны подробности, могут прочитать статью Саши Голдштейна (Sasha Goldshtein) «Garbage Collection Thread Suspension Delay» (http://blog.sashag.net/archive/2009/07/31/garbage-collection-thread-suspension-delay-250ms-multiples.aspx).

Имейте в виду, что неуправляемые потоки выполнения не могут быть приостановлены сборщиком мусора, пока не вернутся в управляемый код – об этом позаботится механизм P/Invoke.

Приостановка потоков в фазе маркировки

В фазе маркировки сборщик мусора выполняет в основном операции чтения. Тем не менее, на этом этапе могут возникать ошибки первого и второго рода.

Вновь созданный объект может быть ошибочно принят за неиспользуемый, несмотря на наличие ссылок на него в приложении. Такое возможно, когда ссылка на объект появилась в той части графа, которую сборщик мусора уже исследовал к моменту обнаружения нового объекта (рис. 4.7). Решить эту проблему можно, перехватывая операции создания новых ссылок (и объектов) и маркируя их. Это требует дополнительных затрат на синхронизацию и увеличивает стоимость выделения памяти, но позволяет другим потокам выполняться параллельно со сборкой мусора.

Объект, промаркированный сборщиком мусора, может оказаться неиспользуемым, если последняя ссылка на него была удалена в ходе выполнения фазы маркировки (рис. 4.8). Это не настолько серьезная проблема, чтобы заострять на ней наше внимание; в конце концов, если объект действительно вышел из употребления, он будет утилизирован в следующем цикле сборки мусора – нет никакой возможности вновь создать ссылку на неиспользуемый объект.

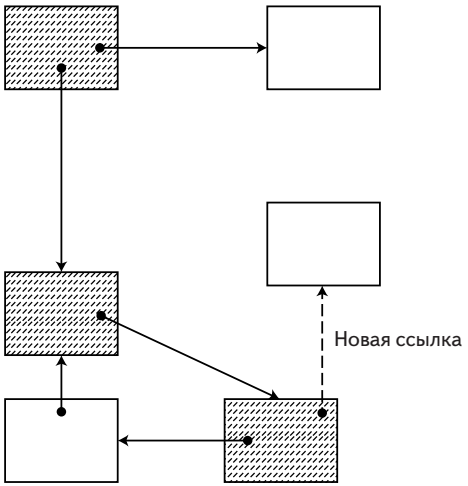

Рис. 4.7. Объект, добавленный в уже промаркированную часть графа (заштрихованными квадратиками обозначены промаркированные объекты). Из-за этого объект ошибочно может быть принят за неиспользуемый.

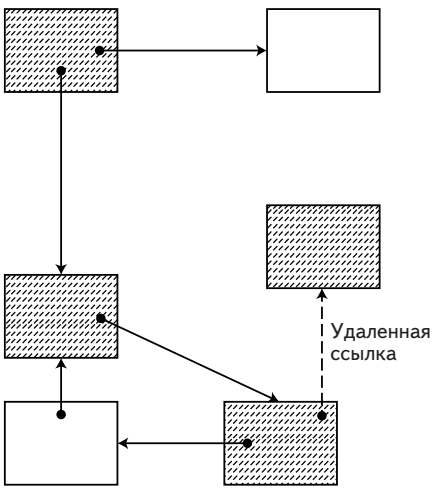

Рис. 4.8. Объект, удаленный из графа уже после того, как оно было помечено (заштрихованными квадратиками обозначены промаркированные объекты). Из-за этого объект может быть ошибочно принят за используемый.

Приостановка потоков в фазе чистки

В фазе чистки выполняется перемещение объектов в памяти и обновление ссылок. Эти обстоятельства порождают новые проблемы для прикладных потоков, выполняемых одновременно со сборкой мусора.

- Копирование объектов не является атомарной операцией. Это означает, что в ходе копирования объекта, оригинал все еще доступен приложению для изменения.
- Обновление ссылок на объекты не является атомарной операцией. Это означает, что одна часть приложения может пользоваться старой ссылкой на объект, а другая – новой.

Решения этих проблем найдены (например, эти проблемы исправлены в сборщике Azul Pauseless для JVM, http://www.azulsystems.com/zing/pgc), но они не реализованы в сборщике мусора для CLR. Гораздо проще объявить, что фаза чистки не поддерживает параллельное выполнение прикладных потоков.

> **Совет.** *Чтобы выяснить, принесет ли выгоду вашему приложению параллельное выполнение потоков и сборщика мусора, определите сначала, сколько времени обычно уходит на сборку мусора. Если приложение тратит половину всего времени на освобождение памяти, тогда перед вами открывается широкое поле для оптимизации. Если, напротив, сборка мусора выполняется раз в несколько минут, возможно вам стоит подумать о приложении своих усилий в другом направлении. Узнать, сколько времени тратится на сборку мусора, можно с помощью счетчика производительности «% Time in GC» («% времени в GC»)? находящегося в категории «.NET CLR Memory» («Память CLR .NET»).*

Теперь, когда мы узнали, как ведут себя прикладные потоки выполнения в процессе сборки мусора, можно перейти к исследованию особенностей других разновидностей сборщиков мусора.

Сборщик мусора для рабочей станции

Первая разновидность сборщика мусора, с которой мы познакомимся, называется сборщиком мусора для рабочей станции (workstation GC). Эта разновидность делится на два подвида: параллельный сборщик мусора для рабочей станции (concurrent workstation GC) и непараллельный сборщик мусора для рабочей станции (non-concurrent workstation GC) .

Этот сборщик мусора выполняется в единственном потоке – сборка мусора происходит последовательно. Обратите внимание, что есть

разница, когда сам процесс сборки выполняется параллельно на нескольких процессорах, и когда процесс сборки выполняется параллельно с прикладными потоками.

Параллельный сборщик мусора для рабочей станции

По умолчанию используется параллельный сборщик мусора для рабочей станции. Этот сборщик выполняется в отдельном потоке с приоритетом `THREAD_PRIORITY_HIGHEST`, и производит сборку мусора от начала и до конца. Кроме того, среда выполнения CLR может позволить выполнять некоторые фазы сборки параллельно с прикладными потоками (параллельно может выполняться большая часть фазы маркировки, как было показано выше). Обратите внимание, что окончательное решение принимается средой выполнения CLR – как будет показано ниже, некоторые фазы выполняются достаточно быстро, чтобы позволить полную приостановку приложения, например, сборка объектов поколения 0. Так или иначе, когда сборщик переходит к фазе чистки, все прикладные потоки приостанавливаются.

Все преимущества от использования параллельного сборщика мусора для рабочей станции могут быть утеряны, если сборка мусора вызывается в потоке выполнения, управляющим пользовательским интерфейсом. В этом случае фоновые прикладные потоки будут выполняться параллельно со сборкой мусора, а поток пользовательского интерфейса будет вынужден ждать ее завершения. Это может значительно ухудшить отзывчивость интерфейса из-за необходимости ждать завершения сборки мусора и конкуренции самого сборщика мусора за ресурсы с другими потоками выполнения (рис. 4.9).

Таким образом, приложения с графическим интерфейсом, использующие параллельный сборщик мусора, должны приложить все усилия, чтобы исключить возможность запуска из потока управления пользовательским интерфейсом. Для этого достаточно обеспечить выделение памяти только в фоновых потоках выполнения и воздерживаться от явного вызова `GC.Collect` в потоке пользовательского интерфейса.

Во всех приложениях для .NET (кроме приложений ASP.NET), независимо от того, где они выполняются, на рабочей станции или мощном многопроцессорном сервере, по умолчанию используется параллельный сборщик мусора для рабочих станций. Это умолчание редко соответствует требованиям серверных приложений, как будет

показано чуть ниже. Как мы только что видели, это умолчание иногда также не соответствует требованиям приложений с графическим интерфейсом, если они склонны вызывать сборку мусора из потока пользовательского интерфейса.

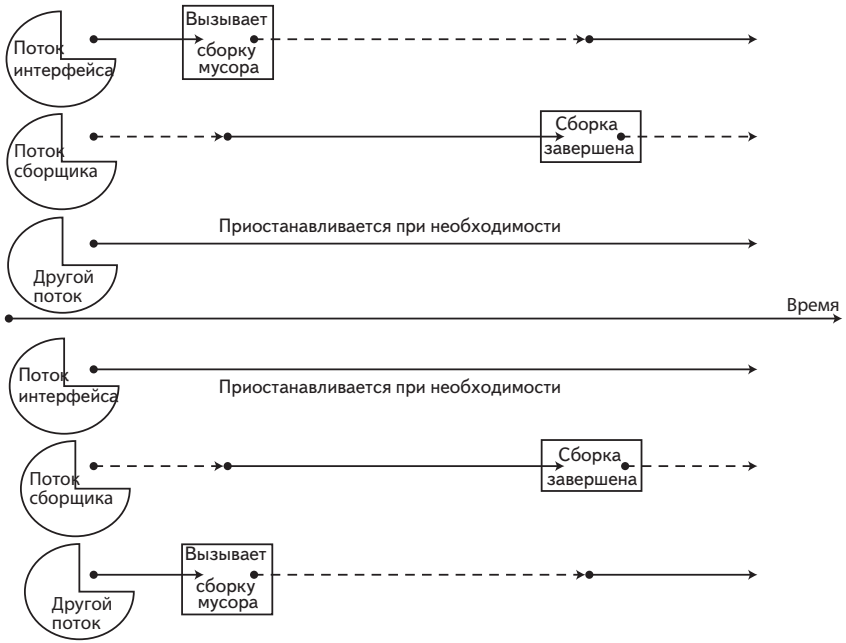

Рис. 4.9. В верхней части показан график работы параллельного сборщика мусора, когда он запускается из потока выполнения, управляющего пользовательским интерфейсом. В нижней части показан график работы параллельного сборщика мусора, когда он запускается одним из фоновых потоков выполнения. (Пунктирные линии соответствуют интервалам простоя потоков выполнения.)

Непараллельный сборщик мусора для рабочей станции

Непараллельный сборщик мусора для рабочих станций, как можно предположить из его названия, приостанавливает выполнение прикладных потоков выполнения, пока не закончатся фазы маркировки и чистки. Этот сборщик мусора предназначен для использования в ситуациях, описанных в предыдущем разделе, когда поток пользовательского интерфейса имеет тенденцию вызывать сборку

мусора. В этом случае непараллельный сборщик мусора способен обеспечить лучшую отзывчивость приложения, потому что ему не приходится конкурировать с фоновыми потоками и поток пользовательского интерфейса блокируется на более короткие интервалы времени (рис. 4.10.)

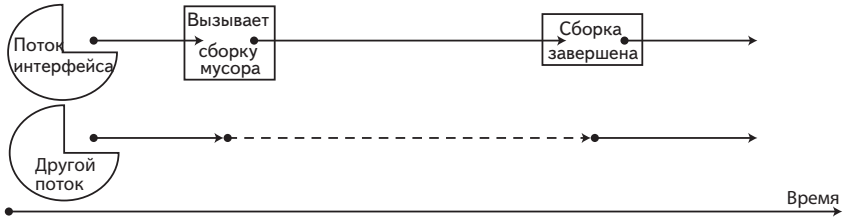

Рис. 4.10. Поток выполнения, управляющий пользовательским интерфейсом, вызывает сборку мусора, которая выполняется непараллельным сборщиком. Другие потоки не конкурируют за ресурсы в ходе сборки мусора.

Сборщик мусора для сервера

Сборщик мусора для сервера оптимизирован для использования в приложениях иного рода – в серверных приложениях. Основной задачей серверных приложений является обеспечение высокой пропускной способности (часто ценой задержки отдельных операций). Кроме того, серверные приложения обычно поддерживают возможность масштабирования на несколько процессоров, соответственно управление памятью так же должно масштабироваться на несколько процессоров.

Приложения, использующие сборщик мусора для серверов, обладают следующими характеристиками.

- В маске схожести (affinity mask) процесса .NET определяется необходимость выделения отдельного пула динамической памяти для каждого процессора. Запросы на выделение памяти из потока, выполняющегося на определенном процессоре, удовлетворяются из пула динамической памяти этого процессора. Цель такого разделения – минимизировать накладные расходы на синхронизацию при выполнении операций выделения памяти: в большинстве случаев конкуренция за обладание указателем на следующий объект не возникает и параллельно выполняющиеся потоки могут выделять память по-настоящему параллельно. Такая архитектура требует динамической

настройки размеров пулов динамической памяти и пороговых значений, чтобы устранить дискриминацию, когда приложение вручную создает рабочие потоки выполнения и назначает им маску схожести процессора. В типичных серверных приложениях, организующих обработку запросов с использованием пула рабочих потоков выполнения, весьма вероятно, что все пулы динамической памяти будут иметь примерно одинаковый размер.

- Сборка мусора не производится в потоке выполнения, вызвавшем ее. Для нужд сборки мусора на запуске приложения создается множество выделенных потоков с приоритетом `THREAD_PRIORITY_HIGHEST`, – по одному для каждого процессора в маске схожести процесса .NET. Это обеспечивает возможность одновременной сборки мусора на нескольких процессорах. Благодаря локальности ссылок, весьма вероятно, что каждый поток, занимающийся сборкой мусора, будет выполнять фазы маркировки и чистки только в собственном пуле динамической памяти, части которого гарантированно будут находиться в кеше данного процессора.
- На время выполнения обеих фаз все прикладные потоки приостанавливаются. Это дает сборщику мусора возможность быстро выполнить свою работу и позволить прикладным потокам продолжить обслуживание запросов. Такой подход увеличивает пропускную способность ценой задержки: некоторые запросы могут обслуживаться дольше из-за пауз, вызываемых сборкой мусора, но в целом приложение оказывается способным обслуживать больше запросов из-за меньшего числа переключений контекста.

Когда используется сборщик мусора для сервера, среда выполнения CLR пытается равномерно распределить операции выделения памяти по разным процессорам. До версии CLR 4.0 балансировка применялась только к куче маленьких объектов; начиная с версии CLR 4.5, балансировка была реализована и для кучи больших объектов (обсуждается ниже). Как результат, количество операций выделения памяти в единицу времени, норма заполнения и частота сборки мусора оказываются примерно одинаковыми для всех пулов динамической памяти.

Единственным ограничением, накладываемым на использование сборщика мусора для сервера, является количество физических процессоров. Если в системе имеется только один физический процессор,

приложение сможет использовать только сборщик мусора для рабочей станции. Это вполне разумный выбор, потому что при наличии единственного процессора будет создан единственный пул управляемой динамической памяти и единственный поток сборки мусора, что приведет к снижению эффективности версии сборщика мусора для сервера.

> **Примечание.** *Начиная с версии NT 6.1 (Windows 7 и Windows Server 2008 R2), операционная система Windows поддерживает более 64 логических процессоров, используя группы процессоров. Начиная с версии CLR 4.5, сборщик мусора так же поддерживает более 64 логических процессоров. Для этого требуется добавить элемент* `<GCCpuGroup enabled = "true" />` *в конфигурационный файл приложения.*

Серверные приложения, вероятнее всего, получат определенные выгоды от использования версии сборщика мусора для сервера. Однако, как было сказано выше, по умолчанию используется параллельный сборщик мусора для рабочей станции. Это верно для всех приложений, выполняющихся в среде CLR с настройками по умолчанию, – консольных приложений, Windows-приложений и Windows-служб. В других средах CLR имеется возможность выбрать другую версию сборщика мусора. Одной из таких сред выполнения является ASP.NET: она выполняет приложения, выбирая для них версию сборщика мусора для сервера, потому сервер IIS обычно устанавливается на компьютеры с несколькими процессорами (впрочем, этот выбор можно изменить настройками в файле *Web.config*).

Управление выбором версии сборщика мусора является темой следующего раздела. Было бы очень интересно провести тестирование производительности приложений, особенно интенсивно работающих с памятью, с использованием различных версий сборщика мусора, чтобы определить, какая из них обеспечивает лучшую производительность под тяжелой нагрузкой на память.

Выбор разновидности сборщика мусора

Управлять выбором разновидности сборщика мусора можно с помощью интерфейсов размещения CLR (Hosting interfaces), обсуждаемых ниже в этой главе. Однако имеется также возможность определять версию сборщика мусора в конфигурационном файле приложения (*App.config*). Ниже приводится разметка XML из конфигурационного файла приложения, с помощью которой можно реализовать выбор между версиями и подверсиями сборщика мусора:

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
    <runtime>
        <gcServer enabled="true" />
        <gcConcurrent enabled="false" />
    </runtime>
</configuration>
```

Элемент `gcServer` управляет выбором разновидности сборщика мусора – для сервера или для рабочей станции. Элемент `gcConcurrent` управляет выбором подвида сборщика мусора для рабочей станции.

В версии .NET 3.5 (включая .NET 2.0 SP1 и .NET 3.0 SP1) появился прикладной программный интерфейс, позволяющий производить выбор во время выполнения. Он реализован в виде класса `System.Runtime.GCSettings` с двумя свойствами: `IsServerGC` и `LatencyMode`.

Свойство `GCSettings.IsServerGC` доступно только для чтения и является признаком использования серверного сборщика мусора. Его нельзя применять для выбора разновидности сборщика мусора во время выполнения, оно лишь отражает настройки приложения или среды выполнения CLR.

Свойство `LatencyMode`, напротив, может принимать значения типа `GCLatencyMode`: `Batch`, `Interactive`, `LowLatency` и `SustainedLowLatency`. Значение `Batch` соответствует непараллельному сборщику мусора; значение `Interactive` – параллельному. Свойство `LatencyMode` допускается применять для переключения между параллельным и непараллельным сборщиком мусора во время выполнения.

Наибольший интерес представляют значения `LowLatency` и `SustainedLowLatency`. Они сообщают сборщику мусора, что ваш код в настоящий момент выполняет операции, чувствительные ко времени выполнения, и сборка мусора в данный момент нежелательна. Значение `LowLatency` появилось в .NET 3.5, оно поддерживается только параллельным сборщиком мусора для рабочей станции и предназначается для непродолжительных операций. Значение `SustainedLowLatency` было добавлено в CLR 4.5, поддерживается обеими разновидностями сборщика мусора, для сервера и для рабочей станции, и предназначается для выполнения продолжительных операций, в ходе которых приложение не должно приостанавливаться для полной сборки мусора. Значения `LowLatency` и `SustainedLowLatency` не предназначены для организации выполнения критически важного кода по причинам, которые будут описаны чуть ниже. Однако они могут пригодиться, например, при выполнении операций, связанных с воспроизведением

анимационных эффектов, когда сборка мусора может ухудшить впечатление, производимое на пользователя.

Установка режима сборки мусора с низкой задержкой (low latency) предписывает сборщику воздерживаться от выполнения полной сборки, если в этом нет абсолютной необходимости, то есть, когда операционная система не испытывает недостатка физической памяти (интенсивный обмен с файлом подкачки гораздо хуже, чем выполнение полной сборки мусора). Режим низкой задержки не отключает сборщик мусора; частичная сборка (о которой будет рассказываться, когда мы перейдем к поколениям объектов) все еще выполняется, но времени на сборку мусора тратится значительно меньше.

Безопасное использование режима низкой задержки

Единственный безопасный способ использовать сборщик мусора в режиме с низкой задержкой – внутри *области ограниченного выполнения* (Constrained Execution Region, CER). Область CER – это ограниченный блок кода, где CLR не сможет возбудить неожиданное исключение (например, аварийное прерывание потока выполнения), которое не позволит блоку кода выполниться полностью. Код, заключенный в область CER, должен вызывать только код, обеспечивающий гарантии надежности. Использование CER является единственным способом, гарантирующим установку режима задержки в прежнее состояние. Следующий фрагмент демонстрирует, как этого добиться (для его компиляции необходимо импортировать пространства имен `System.Runtime.CompilerServices` и `System.Runtime`):

```
GCLatencyMode oldMode = GCSettings.LatencyMode;
RuntimeHelpers.PrepareConstrainedRegions();
try
{
    GCSettings.LatencyMode = GCLatencyMode.LowLatency;
    // Произвести операции, чувствительные к продолжительности
    // выполнения
}
finally
{
    GCSettings.LatencyMode = oldMode;
}
```

Время работы в режиме с низкой задержкой должно быть сведено к минимуму, в противном случае, сразу после выхода из этого режима сборщик мусора может слишком агрессивно взяться за освобождение неиспользуемой памяти, в ущерб производительности приложения. Если вы не контролируете все элементы приложения, имеющиеся в процессе (например, если ваша программа использует расширения или запускает множество потоков выполнения, решающих независимые задачи), не забывайте, что режим низкой задержки действует для всего процесса и может вызывать нежелательные эффекты в других частях приложения.

Определение наиболее подходящей версии сборщика мусора – далеко не тривиальная задача, и зачастую выбор режима может быть сделан только в результате экспериментов. А для приложений, интенсивно работающих с памятью, такие эксперименты обязательны – мы не можем тратить 50% процессорного времени на сборку мусора, выполняемую единственным процессором, в ходе которой другие 15 процессоров будут простаивать, ожидая ее завершения.

Описанная выше модель страдает некоторыми серьезными проблемами производительности, что требует от нас дальнейших исследований. Ниже перечислены наиболее важные из этих проблем.

- Большие объекты: копирование больших объектов является весьма дорогостоящей операцией, а во время фазы чистки постоянно происходит перемещение объектов. В некоторых случаях копирование блоков памяти может составлять основной объем времени, затрачиваемого на сборку мусора. Это наводит на мысль, что объекты разного размера должны обрабатываться по-разному.
- Коэффициент эффективности сборки мусора: согласно описанной модели, каждая сборка мусора является полной, а это означает, что в каждом цикле придется платить слишком высокую цену, выполняя маркировку и чистку всей динамической памяти, даже когда число неиспользуемых объектов невелико. Чтобы повысить коэффициент эффективности сборки мусора, необходимо ввести механизм, различающий объекты по вероятности появления необходимости их утилизации: насколько вероятна необходимость утилизации в следующем цикле сборщика мусора.

Большинство аспектов этих проблем можно решить использованием механизма поколений, являющегося темой следующего раздела, где мы также коснемся некоторых других проблем производительности, которые необходимо учитывать при взаимодействии со сборщиком мусора в .NET.

Поколения

Модель поколений объектов в .NET позволяет оптимизировать производительность за счет выполнения частичной сборки мусора. Частичная сборка мусора характеризуется высоким коэффициентом эффективности, и сборщик мусора выполняет обход объектов, которые вероятнее всего придется утилизировать. Основным решающим

фактором в определении вероятности, что объект потребуется утилизировать, является его возраст – согласно данной модели, между возрастом и ожидаемой продолжительностью жизни объекта существует прямая зависимость.

Предположения в основе модели поколений

В отличие от мира животных, в .NET «юные» объекты «умирают» чаще, а «старые» – реже. Эти два предположения сделаны на основе распределения объектов по их возрасту и продолжительности жизни, как показано на рис. 4.11.

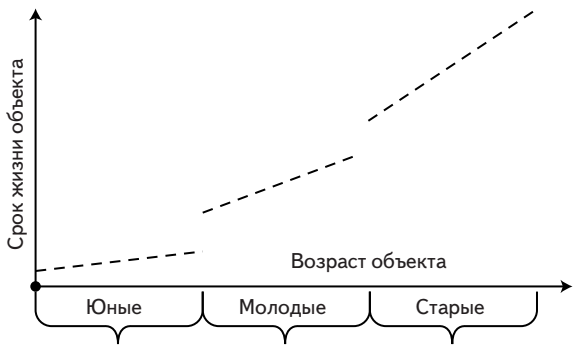

Рис. 4.11. Продолжительность жизни объекта, как функция от его возраста.

Примечание. *Определение «юный»/«старый» зависит от частоты сборки мусора в приложении. В приложении, где сборка мусора выполняется раз в минуту, объект, созданный 5 секунд назад, будет считаться «юным». В другом приложении, интенсивно работающим с памятью, где сборка мусора выполняется десятки раз в секунду, объект такого же возраста будет считаться «старым». Но как бы то ни было, в большинстве приложений временные объекты (например, локальные переменные в методах) обычно умирают «юными», а объекты, созданные в момент инициализации приложения, живут намного дольше.*

Согласно модели поколений предполагается, что в большинстве своем новые объекты являются временными, созданными для краткосрочных целей, и вскоре должны быть утилизированы сборщиком мусора. Старые объекты (например объект-одиночка (singleton) или объекты, созданные на этапе инициализации приложения), напротив, скорее всего будут жить долго.

Эти предположения верны не для всех приложений. Легко можно представить приложение, в котором временные объекты составляют большинство и переживают несколько циклов сборки мусора. Это явление, когда продолжительность жизни объекта не соответствует предсказаниям модели поколений, неформально называют «кризисом среднего возраста». Объекты, проявляющие такое поведение, сводят на нет все преимущества модели поколений. Явление «кризиса среднего возраста» мы исследуем далее в этом разделе.

Реализация поколений в .NET

Согласно модели поколений, динамическая память, обслуживаемая сборщиком мусора, делится на три области: поколение 0, поколение 1 и поколение 2. Эти области соответствуют ожидаемой продолжительности жизни объектов, находящихся в них: поколение 0 включает самые «юные» объекты, а поколение 2 – самые «старые».

Поколение 0

К поколению 0 относятся все вновь созданные объекты (далее в этом разделе будет показано, что объекты также делятся по размеру, что делает это утверждение верным лишь отчасти). Эта область динамической памяти имеет небольшой объем и не может занимать всю динамическую память, даже в небольших приложениях. Обычно для поколения 0 изначально отводится от 256 Кбайт до 4 Мбайт памяти, но этот объем может расти, в зависимости от потребностей.

> **Примечание.** *Помимо разрядности операционной системы, на размер области, выделяемой для поколения 0, также влияют размеры кешей L2 и L3, потому что к этому поколению обычно относятся наиболее часто используемые объекты, живущие в течение короткого периода времени. Размер этой области может также динамически изменяться сборщиком мусора и устанавливаться средой выполнения CLR в момент запуска приложения, исходя из настроек пороговых значений. Суммарный размер областей, выделяемых для поколений 0 и 1 не может превышать размер одного сегмента (обсуждается ниже).*

Когда новый запрос на выделение памяти не может быть удовлетворен из-за переполненности области для поколения 0, инициируется сборка мусора в этой области. В ходе ее выполнения сборщик мусора просматривает только объекты из поколения 0. Это сложная задача, потому что между конями и поколениями нет никакой связи, и всегда остается вероятность, что на объект из поколения 0 будет

ссылаться объект из другого поколения. Эту проблему мы исследуем чуть ниже.

Сборка мусора в поколении 0 является очень дешевой и эффективной операцией по следующим причинам.

- Поколение 0 весьма немногочисленно. Обход такого небольшого объема памяти занимает очень мало времени. На одном из наших тестовых компьютеров сборка мусора в поколении 0, из которого выживало только 2% объектов, занимала примерно 70 микросекунд.
- Размер области памяти для поколения 0 зависит от размера кеша процессора, что повышает вероятность попадания всех объектов из поколения 0 в этот кеш. Обход памяти, уже находящейся в кеше, выполняется существенно быстрее, чем обход обычной памяти или памяти, вытесненной в файл подкачки, как будет показано в главе 5.
- Из-за временной локальности (temporal locality) весьма вероятно, что объекты из поколения 0 будут ссылаться на другие объекты из этого же поколения. Также весьма вероятно, что эти объекты будут размещаться очень близко друг от друга. Это увеличивает эффективность обхода графа объектов в фазе маркировки из-за более частых попаданий в кеш.
- Поскольку новые объекты, как предполагается, будут существовать недолго, вероятность необходимости утилизации каждого конкретного объекта в поколении 0 чрезвычайно высока. Это, в свою очередь, означает, что большинство объектов в поколении 0 не придется перемещать – занимаемую ими память можно просто освободить для других объектов. Это также означает, что время на сборку мусора будет потрачено не напрасно и большинство объектов действительно можно утилизировать.
- Когда сборка мусора завершается, освободившуюся память можно использовать для удовлетворения новых запросов. Поскольку этап сборки мусора только что закончился, высока вероятность, что память окажется в кеше процессора, операции выделения памяти и обращения к объектам будет выполняться несколько быстрее.

Как мы установили, большинство объектов в поколении 0 будет утилизировано в первом же цикле сборки мусора. Однако некоторые объекты могут продолжить существование по тем или иным причинам:

- приложение может быть плохо спроектировано и создавать временные объекты, живущие дольше одного цикла сборки мусора;
- приложение находится на этапе инициализации, когда создаются долгоживущие объекты;
- приложение создало несколько временных, короткоживущих объектов непосредственно перед запуском сборщика мусора.

Объекты из поколения 0, пережившие сборку мусора, не перемещаются в начало области динамической памяти, выделенной для этого поколения, а переносятся в поколение 1, чтобы отразить факт увеличенной продолжительности жизни. В результате этого переноса они копируются из области памяти поколения 0 в область памяти поколения 1 (рис. 4.12). На первый взгляд такое копирование выглядит дорогостоящей операцией, но в любом случае это копирование является составной частью фазы чистки. Кроме того, так как коэффициент эффективности сборки мусора в поколении 0 очень высок, стоимость этого копирования должна быть невысока, в сравнении с разностью затрат между частичной и полной сборкой мусора.

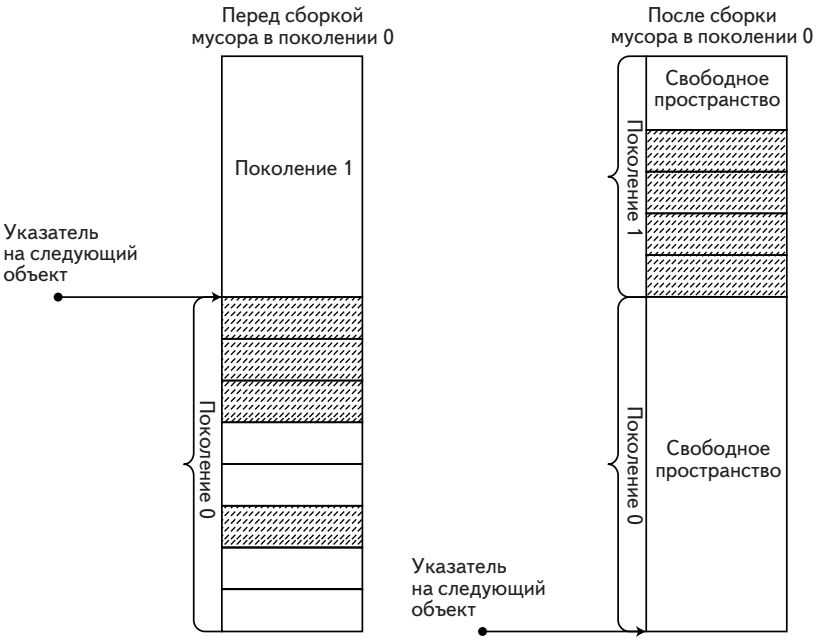

Рис. 4.12. Выжившие объекты из поколения 0 переносятся в поколение 1 по завершении сборки мусора.

Перемещение закрепленных объектов между поколениями

Закрепленные объекты не должны перемещаться сборщиком мусора. В обобщенной модели этот запрет препятствует переносу закрепленных объектов между поколениями. Наиболее существенным данное ограничение является для самых юных поколений, таких как поколение 0, потому что размер поколения 0 очень невелик. Закрепленные объекты могут приводить к увеличению фрагментации памяти внутри поколения 0 и причинять дополнительные проблемы, не видимые нами раньше, когда мы обсуждали модель закрепления объектов. К счастью, CLR обладает возможностью переноса закрепленных объектов, применяя следующий трюк: когда память поколения 0 оказывается сильно фрагментированной из-за большого количества закрепленных объектов, CLR может объявить все пространство поколения 0 следующим поколением и выделить новую область памяти для новых объектов, объявив ее поколением 0. Это достигается за счет изменения эфемерного сегмента, о чем рассказывается ниже в этой главе.

Следующий код демонстрирует, как закрепленные объекты могут переноситься из поколения в поколение с помощью метода `GC.GetGeneration`, обсуждаемого далее в этой главе:

```
static void Main(string[] args) {
    byte[] bytes = new byte[128];
    GCHandle gch = GCHandle.Alloc(bytes, GCHandleType.Pinned);

    GC.Collect();
    Console.WriteLine("Generation: " + GC.GetGeneration(bytes));

    gch.Free();
    GC.KeepAlive(bytes);
}
```

Если заглянуть в динамическую память перед сборкой мусора, поколения будут располагаться, как показано ниже:

```
Generation 0 starts at 0x02791030
Generation 1 starts at 0x02791018
Generation 2 starts at 0x02791000
```

Если заглянуть в динамическую память после сборки мусора, можно увидеть, что размещение поколений в пределах сегмента изменилось, как показано ниже:

```
Generation 0 starts at 0x02795df8
Generation 1 starts at 0x02791018
Generation 2 starts at 0x02791000
```

Адрес объекта (в данном случае, 0x02791be0) не изменился, потому что он закреплен в памяти, но за счет перемещения границ поколений средой выполнения CLR создается иллюзия переноса объекта из одного поколения в другое.

Поколение 1

Поколение 1 – это буфер между поколением 0 и поколением 2, содержащий объекты, пережившие один цикл сборки мусора. Эта область динамической памяти немного больше области поколения 0, но все еще меньше на несколько порядков объема всей доступной памяти. Обычно для поколения 1 изначально отводится от 512 Кбайт до 4 Мбайт памяти.

Когда область для поколения 1 заполняется, запускается процедура сборки мусора в поколении 1. Это все еще частичная сборка мусора; маркируются и удаляются только объекты из поколения 1. Обратите внимание, что попасть в поколение 1 объект может только из поколения 0, в результате переноса, выполняемого в ходе сборки мусора в поколении 1 (в том числе и сборки мусора, инициированной вручную).

Сборка мусора в поколении 1 все еще остается относительно недорогой операцией. Исследованию подвергается не более нескольких мегабайтов памяти. Коэффициент эффективности так же остается высоким, потому что большинство объектов в поколении 1 составляют временные, короткоживущие объекты – объекты, которые не были утилизированы в поколении 0, но которым не суждено пережить еще один цикл сборки мусора. Например, короткоживущие объекты с финализаторами (finalizers) гарантированно доживают до поколения 1. (Финализация будет рассматриваться далее в этой главе.)

Объекты, пережившие цикл сборки мусора в поколении 1, переносятся в поколение 2. Этот перенос отражает тот факт, что теперь объекты будут считаться «старыми». Одним из основных рисков модели поколений является вероятность попадания в поколение 2 временных объектов, которые вскоре после этого должны стать неиспользуемыми; это называется явлением «кризиса среднего возраста». Чрезвычайно важно воспрепятствовать попаданию временных объектов в поколение 2. Далее в этом разделе мы исследуем отрицательное влияние явления «кризиса среднего возраста» и познакомимся со средствами его диагностики и профилактики.

Поколение 2

Поколение 2 – это последняя область памяти для объектов, переживших как минимум два цикла сборки мусора (а также для очень больших объектов, как будет показано ниже). В модели поколений такие объекты считаются «старыми» и, согласно нашим предположениям, не должны выйти из употребление в ближайшем будущем.

Объем памяти для объектов поколения 2 не ограничивается искусственно. Она может занимать все пространство, выделяемое операционной системой процессу, то есть, до 2 Гбайт в 32-разрядной системе и до 8 Тбайт в 64-разрядной.

> **Примечание.** *Несмотря на огромный размер, для поколения 2 определяются пороговые значения, по достижении которых запускается сборка мусора, потому что нет смысла ждать, пока будет заполнена вся память. Если бы все приложения в системе освобождали неиспользуемую память, только после ее заполнения, интенсивное использование файла подкачки сделало бы работу просто невозможной.*

Сборка мусора в поколении 2 является полной. Это самая дорогая операция сборки мусора, для выполнения которой может потребоваться ощутимое количество времени. На одном из наших тестовых компьютеров выполнение полной сборки мусора в объеме памяти 100 Мбайт заняло примерно 30 миллисекунд – на несколько порядков больше, чем сборку в поколении 0.

Кроме того, если приложение ведет себя в соответствии с основными предположениями, сделанными в модели, коэффициент эффективности сборки мусора в поколении 2 так же должен быть очень низким, потому что большинство объектов в поколении 2 переживут множество циклов сборки мусора. Вследствие этого сборка мусора в поколении 2 должна производиться достаточно редко – она выполняется слишком медленно, в сравнении с частичной сборкой в более молодых поколениях, и неэффективно, потому что большинство объектов продолжают использоваться и память почти не освобождается.

Если все временные объекты, создаваемые приложением, будут быстро уничтожаться, они не смогут пережить несколько сборок мусора и попасть в поколение 2. В этом оптимистичном сценарии сборка мусора в поколении 2 просто не нужна, и влияние сборщика мусора на производительность приложения – минимально.

Добавив модель поколений, нам удалось решить одну из основных проблем упрощенной модели сборки мусора, описанной в предыдущих разделах: распределение объектов по вероятности появления необходимости утилизировать их. Имея возможность предсказывать сроки жизни объектов, можно выполнять дешевую частичную сборку мусора, и лишь изредка производить дорогостоящую полную сборку. Однако нерешенной остается еще одна проблема: копирование больших объектов в фазе чистки, которое может быть весьма дорогим удовольствием в смысле вычислительных ресурсов и памяти. Кроме того, остается неясным, как, согласно модели, поколение 0 может со-

держать массив из 10 000 000 целых чисел, если массив намного превышает размер области, выделенной для этого поколения.

Куча больших объектов

Куча больших объектов (Large Object Heap, LOH) – это специальная область, зарезервированная для размещения очень больших объектов. Большими считаются объекты, занимающие больше 85 Кбайт памяти. Это – пороговое значение относится к одному объекту, а не к графу объектов с корнем в данном объекте, поэтому массив из 1000 строк (по 100 символов в каждой) не считается большим объектом, так как сам массив содержит лишь 4- или 8-байтные ссылки на строки, а вот массив из 50 000 целых чисел – это большой объект.

Большие объекты помещаются непосредственно в кучу больших объектов и к ним не применяются положения модели поколений. Это устраняет накладные расходы на копирование объектов между поколениями. Однако, при сборке мусора в куче больших объектов, стандартный алгоритм фазы чистки все же может потребовать копировать объекты и оказать отрицательное влияние на производительность. Чтобы избежать непроизводительных потерь, к объектам в куче больших объектов применяется другой алгоритм чистки.

Вместо чистки и копирования больших объектов, сборщик мусора использует иную стратегию. Он поддерживает список неиспользуемых блоков памяти и удовлетворяет запросы, выделяя блоки из этого списка. Эта стратегия очень напоминает стратегию на основе списка свободных блоков памяти, обсуждавшуюся в начале главы, и ей свойственны те же недостатки, касающиеся производительности: высокая стоимость операции выделения памяти (поиск подходящего блока, дефрагментация блоков), высокая стоимость операции освобождения памяти (возврат блока памяти в список) и высокая стоимость управления памятью (объединение смежных блоков). Однако обслуживание списка свободных боков намного дешевле, чем копирование больших объектов при перемещении – и это типичный случай, когда чистота реализации нарушается в угоду лучшей производительности.

Внимание. *Поскольку большие объекты не перемещаются, может показаться, что их можно не закреплять, когда потребуется получить адрес большого объекта в памяти. Это ошибочное мнение, так как никогда не следует опираться на особенности реализации. Вы не должны предполагать, что большие объекты будут оставаться в одном и том же месте в памяти в течение всей своей жизни, к тому же пороговое значение, разде-*

ляющее обычные и большие объекты, может измениться в будущем! Тем не менее, с практической точки зрения можно смело предположить, что закрепление больших объектов будет меньше сказываться на производительности, чем закрепление маленьких и «юных» объектов. Фактически, зачастую даже предпочтительнее создать большой массив, закрепить его в памяти и передавать фрагменты этого массива, вместо того, чтобы создавать новые маленькие массивы для каждой операции, где требуется закрепленный массив.

Сборка мусора в куче больших объектов выполняется по достижении порогового значения в поколении 2. Аналогично, по достижении порогового значения в куче больших объектов выполняется сборка и в поколении 2. Таким образом, создание множества больших временных объектов вызывает те же проблемы, что и эффект «кризиса среднего возраста» – полную сборку мусора для удаления этих объектов. Еще одна потенциальная проблема – фрагментирование кучи больших объектов, потому что дырки между объектами не удаляются автоматически в фазе чистки.

Поддержка модели кучи больших объектов требует от разработчиков приложений с осторожностью подходить к распределению памяти для больших объектов. Одна из эффективных стратегий состоит в том, чтобы создать пул больших объектов и повторно использовать их, в обход сборщика мусора. Накладные расходы на управление таким пулом могут оказаться намного меньше расходов на выполнение полной сборки мусора. Другой возможный подход (вовлекающий массивы одного и того же типа) заключается в создании очень большого объекта и распределении его фрагментов для своих нужд вручную (рис. 4.13).

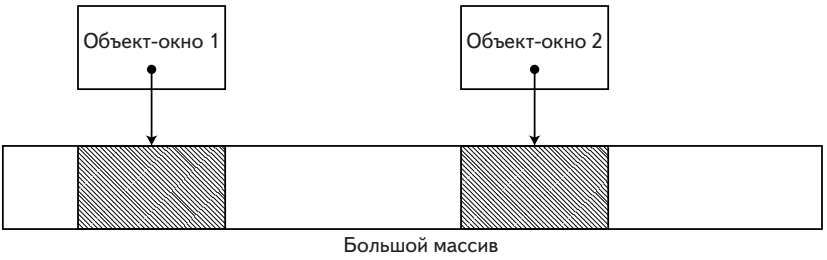

Рис. 4.13. Создание очень большого объекта и распределение его фрагментов вручную через маленькие «объекты-окна».

Ссылки между поколениями

При обсуждении модели поколений мы упустили одну важную деталь, которая может нарушить ее правильность и отрицательно сказаться на производительности. Как вы помните, частичная сборка мусора молодых поколений является недорогой операцией, потому что в фазе маркировки просматриваются только «юные» объекты. Но как сборщик мусора может быть уверен, что коснется только «юных» объектов?

Рассмотрим, как выполняется фаза маркировки в ходе сборки мусора в поколении 0. В этой фазе сборщик мусора выявляет текущие активные корни и приступает к конструированию графа всех объектов, на которые ссылаются эти корни. Из этого графа следует исключить объекты, не принадлежащие поколению 0. Однако, если исключать их после создания графа, это означает, что мы обойдем все ссылки и фаза маркировки окажется такой же затратной, как и при полной сборке мусора. Как вариант, можно было бы прекращать обход графа, достигнув первого объекта, не принадлежащего поколению 0. Но тогда есть риск пропустить объекты из поколения 0, на которые ссылаются объекты из старших поколений, как показано на рис. 4.14!

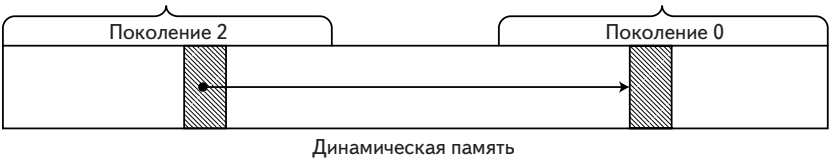

Рис. 4.14. Ссылки между поколениями могли бы быть пропущены, если бы в фазе маркировки обход объектов прекращался по достижении первого объекта старшего поколения.

Похоже, что придется искать компромисс между производительностью и правильностью. Данную проблему легко было бы решить, имея знание о наличии ссылок между поколениями. В этом случае, перед выполнением фазы маркировки сборщик мусора мог бы включить такие старые объекты в множество корней при конструировании графа, что дало бы ему возможность остановить обход, достигнув первого же объекта, не принадлежащего поколению 0.

Как оказывается, подобную информацию можно получить с помощью JIT-компилятора. Ситуация, когда объект из старшего поколения ссылается на объект из младшего поколения, может сложиться только при выполнении одной категории инструкций: присваивание

непустой ссылки полю экземпляра ссылочного типа (или запись в элемент массива).

```
class Customer {
    public Order LastOrder { get; set; }
}

class Order { }

class Program {
    static void Main(string[] args) {
        Customer customer = new Customer();
        GC.Collect();
        GC.Collect();
        // теперь объект customer находится в поколении 2
        customer.LastOrder = new Order();
    }
}
```

Когда JIT-компилятор встречает такие инструкции, он генерирует так называемый «барьер записи» (write barrier), перехватывающий запись ссылки во время выполнения и записывающий вспомогательную информацию в специальную таблицу. Барьер записи – это функция CLR, проверяющая, не присваивается ли ссылка полю объекта из поколения, более старого, чем поколение 0. В этом случае она обновляет байт в таблице, соответствующий диапазону адресов, в который попадает целевой объект операции присваивания (рис. 4.15).

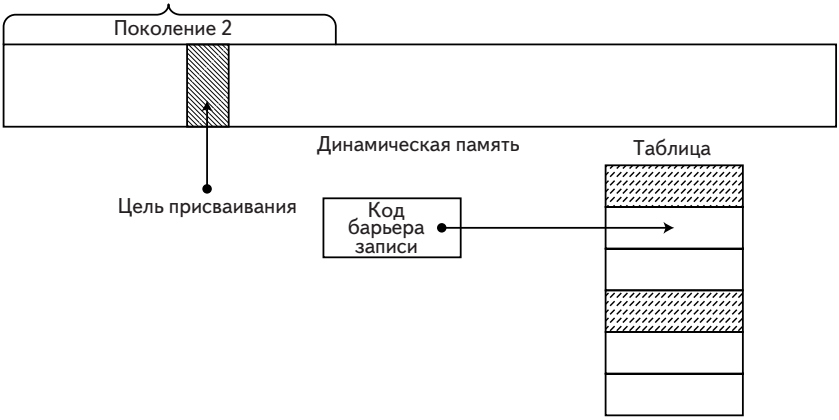

Рис. 4.15. Операция присваивания ссылки полю проходит через барьер записи, который обновляет соответствующий бит в таблице, соответствующий области памяти, где обновилась ссылка.

Трассировку барьера записи легко выполнить с помощью отладчика. Фактическая операция присваивания в методе `Main` была скомпилирована JIT-компилятором в следующий код:

```
; ESI содержит указатель на объект 'customer', ESI+4 - на 'LastOrder',
EAX - 'new Order()'
lea edx, [esi+4]
call clr!JIT_WriteBarrierEAX
```

Внутри функции, реализующей барьер записи, проверка выполняется путем сравнения адреса присваивания с нижней границей области для поколения 1 (то есть, проверяется принадлежность к поколению 1 или 2). Если проверка дает положительный результат, производится обновление таблицы, присваиванием значения `0xFF` байту со смещением, полученным сдвигом адреса объекта на 10 разрядов вправо. (Если байт уже установлен в значение `0xFF`, повторная установка не производится, чтобы предотвратить принудительное обновление кешей других процессоров; подробнее об этом рассказывается в главе 6.)

```
mov dword ptr [edx], eax          ; фактическая запись
cmp eax, 0x272237C                ; начальный адрес поколения 1
jb NoNeedToUpdate
shr edx, 0xA                      ; сдвинуть вправо на 10 разрядов
cmp byte ptr [edx+0x48639C], 0xFF ; 0x48639C - начало таблицы
jne NeedUpdate
NoNeedToUpdate:
ret
NeedUpdate:
mov byte ptr [edx+0x48639C], 0xFF ; обновить таблицу
ret
```

Сборщик мусора использует эту вспомогательную информацию в фазе маркировки. Он проверяет содержимое таблицы, чтобы определить, какие диапазоны адресов следует рассматривать как корни при сборке мусора в поколении 0. Сборщик выполняет обход объектов в этом диапазоне адресов и отыскивает в них ссылки на объекты в самом молодом поколении. Это дает возможность применить оптимизацию производительности, описанную выше, когда сборка мусора прекращается по достижении объекта, не принадлежащего поколению 0.

Каждый байт в таблице охватывает 1 Кбайт динамической памяти. То есть, таблица может занимать до ~0.1% пространства от общего объема, но она дает огромный прирост производительности при сборке мусора в поколениях молодых объектов. Если бы в таблице

можно было сохранять записи с информацией о каждой отдельной ссылке, скорость можно было бы увеличить еще больше (сборщик мусора мог бы сразу перейти к исследованию конкретного объекта, а не диапазона адресов 1024 байт), но сохранение такой информации во время выполнения – слишком дорогое удовольствие. Существующий подход отлично отражает компромисс между расходованием памяти и экономией времени выполнения.

Примечание. *Хотя такие микроскопические оптимизации редко стоят затраченных усилий, тем не менее, мы можем уменьшить накладные расходы на обновление и обход таблицы. Одно из возможных решений заключается в создании пула объектов и повторном их использовании вместо многократного создания. Это позволит уменьшить расходы на сборку мусора в целом. Другое решение – использовать типы значений везде, где только возможно, и уменьшить количество ссылок в графе объектов. При присваивании типу значения барьер записи не генерируется, потому что тип значения в динамической памяти всегда является частью некоторого ссылочного типа (или, в упакованной форме, частью самого себя).*

Фоновый сборщик мусора

Параллельный сборщик мусора для рабочей станции, появившийся в версии CLR 1.0, имеет один существенный недостаток. Несмотря на то, что во время сборки мусора в поколении 2 прикладные потоки способны продолжать создавать новые объекты, запросы на выделение памяти будут удовлетворяться, только пока не будет исчерпана вся память, выделенная для поколений 0 и 1. Как только это произойдет, прикладные потоки блокируются до окончания сборки мусора.

Фоновый сборщик мусора, появившийся в версии CLR 4.0, позволяет среде выполнения CLR выполнять сборку мусора в поколениях 0 и 1, даже когда уже выполняется полная сборка мусора. Для этого CLR запускает два потока выполнения: основной поток сборщика мусора и фоновый. Фоновый поток производит сборку мусора в поколении 2 в фоновом режиме и периодически проверяет появление запросов на выполнение быстрой сборки в поколениях 0 и 1. Получив запрос (например, когда приложение исчерпает доступную память в молодых поколениях), фоновый поток приостанавливается и возобновляет работу основного потока, который быстро освобождает память и разблокирует прикладные потоки.

В CLR 4.0 фоновый сборщик мусора используется по умолчанию в приложениях, использующих параллельный сборщик мусора для рабочей станции. Среда выполнения не давала никакой возможности

отменить выбор фонового сборщика мусора или использовать его с другими разновидностями.

В CLR 4.5 фоновый сборщик мусора был дополнен серверной версией. Кроме того, в серверный сборщик мусора была добавлена поддержка параллельной сборки мусора. При использовании параллельного сборщика мусора для сервера в процессе, выполняющемся на N логических процессорах, среда выполнения CLR создает N основных потоков сборки мусора и N фоновых потоков. Фоновые потоки обслуживают динамическую память, выделенную для поколения 2, и позволяют прикладному коду выполняться параллельно. Основные потоки сборки мусора вызываются всякий раз, когда необходимо произвести блокирующую сборку мусора, чтобы выполнить сжатие (фоновые потоки сборки мусора не производят сжатие), или собрать мусор в более юных поколениях посреди полной сборки в фоновых потоках. Итак, в версии CLR 4.5 существует четыре различные разновидности сборщика мусора, из которых можно выбирать в конфигурационном файле:

1. Параллельный сборщик мусора для рабочей станции – используется по умолчанию; имеет фоновый режим выполнения.
2. Непараллельный сборщик мусора для рабочей станции – не имеет фонового режима.
3. Непараллельный сборщик мусора для сервера – не имеет фонового режима.
4. Параллельный сборщик мусора для сервера – имеет фоновый режим.

Сегменты сборщика мусора и виртуальная память

В нашем обсуждении модели сборки мусора и модели поколений мы постоянно предполагали, что в .NET процесс узурпирует все доступное пространство памяти и использует ее в качестве динамической памяти, обслуживаемой сборщиком мусора. Совершенно очевидно, что это предположение не является верным, учитывая тот факт, что управляемые приложения не способны выполняться в изоляции от неуправляемого кода. Сама среда CLR является неуправляемой реализацией, библиотека базовых классов .NET зачастую всего лишь обертывает интерфейсы Win32 и COM, а кроме того, «управляемые»

приложения могут загружать и использовать нестандартные неуправляемые компоненты.

> **Примечание.** *Даже если бы управляемый код мог выполняться в полной изоляции, все равно не было бы никакого смысла немедленно передавать сборщику мусора все доступное адресное пространство. Передача памяти не означает, что она немедленно будет использована, тем не менее, эта операция имеет свои накладные расходы. Поэтому в общем случае желательно выделять процессу памяти немного больше, чем ему, вероятно, потребуется. Как будет показано ниже, CLR резервирует значительные области памяти на будущее, но передает их только когда это действительно необходимо и считает для себя обязательным возвращать неиспользуемую память Windows.*

Учитывая сказанное выше, нам необходимо рассмотреть возможность взаимодействия сборщика мусора с диспетчером виртуальной памяти. На запуске процессу выделяется два блока в виртуальной памяти, называемые сегментами сборщика мусора (GC segments), точнее, процесс запрашивает у среды CLR эти блоки памяти. Первый сегмент используется для поколений 0, 1 и 2 (называется эфемерным сегментом (ephemeral segment)). Второй – для кучи больших объектов. Размеры сегментов зависят от разновидности сборщика мусора и от начальных настроек. Типичный размер сегмента в 32-разрядной системе при использовании сборщика мусора для рабочей станции составляет 16 Мбайт, для сервера – в диапазоне от 16 до 64 Мбайт. В 64-разрядной системе CLR выделяет сегменты от 128 Мбайт до 2 Гбайт серверному сборщику мусора и от 128 Мбайт до 256 Мбайт – сборщик мусора для рабочей станции. (Среда выполнения CLR не передает сразу весь сегмент; она только резервирует адресное пространство и передает сегменты по частям, по мере необходимости.)

Когда сегмент заполняется и от приложения поступает дополнительный запрос на выделение памяти, CLR выделяет еще один сегмент. В каждый конкретный момент поколения 0 и 1 могут находиться только в одном сегменте. Однако, это не обязательно должен быть тот же самый сегмент! Выше мы узнали, что закрепление объектов в поколении 0 или 1 на длительное время чревато фрагментацией памяти, имеющей и без того небольшой объем. CLR решает эту проблему, объявляя другой сегмент эфемерным сегментом, в результате чего объекты, прежде находившиеся в более молодых поколениях, оказываются сразу в поколении 2 (потому что существовать может только один эфемерный сегмент).

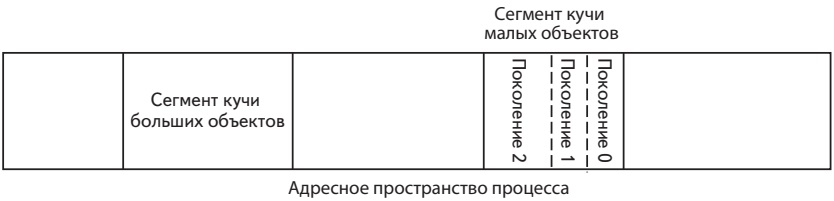

Рис. 4.16. Виртуальное адресное пространство занимают сегменты сборщика мусора. Сегмент, содержащий молодые поколения, называется эфемерным сегментом.

Когда в результате сборки мусора сегмент пустеет, CLR обычно освобождает память и возвращает ее операционной системе. Это – предпочтительное поведение для большинства приложений, особенно для приложений, нечасто запрашивающих большие объемы памяти. Однако, можно потребовать от CLR оставить пустой сегмент в резервном списке, не возвращая его операционной системе. Такое поведение называют удержанием сегмента, или удержанием виртуальной памяти, и может быть определено на этапе запуска передачей флагов функции `CorBindToRuntimeEx`. Удержание сегмента может способствовать повышению производительности приложений, очень часто выделяющему и освобождающему сегменты (приложений, интенсивно работающих с памятью с частыми пиками использования памяти), и снижению вероятности появления исключений нехватки памяти из-за фрагментации виртуальной памяти (обсуждается чуть ниже). Такое поведение используется по умолчанию в приложениях ASP.NET. В некоторых средах CLR это поведение допускает дальнейшую настройку, с использованием интерфейса `IHostMemoryManager`, что позволяет организовать выделение сегментов из пула памяти или любого другого источника.

Модель сегментов управляемого пространства памяти вводит весьма серьезную проблему, связанную с фрагментацией внешней (виртуальной) памяти. Поскольку после опустошения сегменты возвращаются операционной системе, неуправляемый код может выделить для себя память в середине области, которая когда-то была сегментом сборщика мусора. Это вызывает фрагментацию памяти, так как сегмент должен быть представлен непрерывным диапазоном адресов памяти.

Чаще всего причины фрагментации виртуальной памяти связаны с динамическими сборками (dynamic assemblies), загружаемыми на этапе выполнения (такими как сборки, реализующие сериализацию данных в формат XML, или страницы с отладочной информацией компилятора в приложениях ASP.NET), с динамически загружаемы-

ми COM-объектами и неуправляемым кодом, выполняющим выделение памяти вразброс. Фрагментация виртуальной памяти может наблюдаться, даже когда объем памяти, используемой процессом, весьма далек от предела в 2 Гбайта. Долгоживущие процессы, такие как веб-серверы, интенсивно работающие с памятью с частыми пиковыми нагрузками, обычно проявляют эту проблему через несколько часов, дней или недель выполнения. В не особенно критичных случаях, когда отказы допустимы (например, когда обслуживание выполняется фермой серверов с балансировкой нагрузки), эта проблема решается перезапуском процесса.

> **Примечание.** *Теоретически, если положить, что размер сегмента равен 64 Мбайт, точность выделения блоков виртуальной памяти равна 4 Кбайт и адресное пространство 32-разрядного процесса составляет 2 Гбайт (то есть, места хватит только для 32 сегментов), достаточно выделить всего 4 Кбайт × 32 = 128 Кбайт неуправляемой памяти, чтобы сделать невозможным выделение даже одного сегмента!*

Фрагментация виртуальной памяти легко обнаруживается с помощью утилиты Sysinternals VMMap. Она не только сообщает, какая область памяти и для каких целей используется, но также имеет возможность отображать на экране общую картину адресного пространства и упрощает визуальный поиск проблем. На рис. 4.17 показан срез адресного пространства в ситуации, когда имеется более 500 Мбайт свободного пространства, но нет возможности выделить хотя бы один блок достаточного размера, чтобы разместить сегмент сборщика мусора, размером 16 Мбайт. Белые области на скриншоте – это свободные блоки памяти.

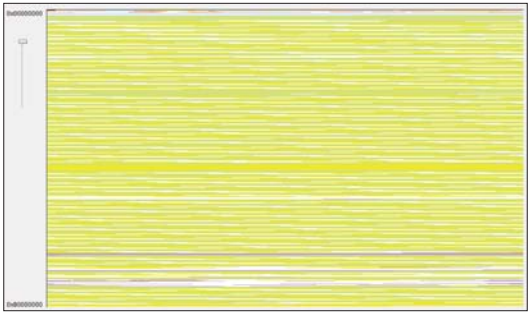

Рис. 4.17. Срез фрагментированного адресного пространства. Имеется более 500 Мбайт свободной памяти, но отсутствуют свободные блоки достаточного размера, чтобы выделить сегмент сборщика мусора.

Эксперимент с утилитой VMMAP

Вы можете попробовать использовать VMMap с примером приложения и убедиться, насколько быстро он помогает найти правильное направление. В частности, утилита VMMap позволяет легко определить, связан ли источник проблем с управляемой динамической памятью или с чем-то другим, и упрощает диагностику проблем, связанных с фрагментацией.

1. Загрузите утилиту VMMap на сайте Microsoft TechNet (http://technet.microsoft.com/en-us/sysinternals/dd535533.aspx) и созраните на своем компьютере.
2. Запустите приложение *OOM2.exe* из папки с примерами для этой главы. Приложение быстро исчерпает всю доступную память и завершится с исключением. Когда на экране появится диалог с сообщением об ошибке, не закрывайте его – оставьте приложение в текущем состоянии.
3. Запустите утилиту VMMap и выберите в открывшемся списке процесс *OOM2.exe*. Обратите внимание на объем доступной памяти (строка **Free** (Свободно) в верхней таблице). Выберите пункт меню **View** → **Fragmentation View** (Обзор → Фрагментация) чтобы оценить распределение памяти визуально. Исследуйте подробный список блоков (в нижней таблице), выбрав строку **Free** (Свободно) в верхней таблице и найдите самый большой свободный блок в адресном пространстве приложения.
4. Вы без труда заметите, что приложение исчерпало далеко не всю виртуальную память, но в ней нет свободного блока подходящего размера, чтобы можно было выделить новый сегмент динамической памяти – самый большой свободный блок из доступных имеет размер меньше 16 Мбайт.
5. Повторите шаги 2 и 3, использовав приложение *OOM3.exe* а из папки с примерами для этой главы. Теперь момент исчерпания памяти наступает позже и совсем по другой причине.

Когда вам доведется столкнуться в Windows-приложении с проблемами, связанными с памятью, утилита VMMap должна быть у вас под рукой. Она поможет отыскать точки утечки памяти, выявить фрагментацию памяти и множество других проблем. С ее помощью можно даже профилировать операции выделения памяти неуправляемым кодом: подробнее о профилировании с помощью VMMap рассказивается в статье Саши Голдштейн (Sasha Goldshtein): http://blog.sashag.net/archive/2011/08/27/vmmap-allocation-profiling-and-leak-detection.aspx.

Существует два подхода к решению проблемы фрагментации виртуальной памяти.

- Уменьшить количество динамических сборок (dynamic assemblies), уменьшить объем памяти, выделяемой неуправляемым кодом или использовать прием создания пула блоков, использовать прием удержания сегментов или выделять пулы управляемой памяти. Эти приемы обычно отдаляют проявление проблемы, но не устраняют ее полностью.

- Использовать 64-разрядную версию операционной системы и выполнять код в 64-разрядном процессе – такие процессы имеют адресное пространство объемом 8 Тбайт, что позволяет полностью избавиться от проблемы. Так как проблема связана не с объемом доступной физической памяти, а скорее с объемом виртуального адресного пространства, перехода на 64-разрядную версию будет достаточно, независимо от объема физической памяти.

На этом завершается исследование модели сегментов, определяющей взаимосвязь между управляемой и виртуальной памятью. В большинстве приложений никогда не потребуется заниматься настройкой этой связи; но если вам придется столкнуться со специфическими ситуациями, описанными выше, среда CLR готова предоставить самое полное и легко настраиваемое решение.

Финализация

До сих пор в этой главе рассматривались подробности, связанные с управлением ресурсами одного типа – управляемой памятью. Однако в реальном мире существует множество других типов ресурсов, которые все вместе можно назвать неуправляемыми ресурсами, потому что они неподконтрольны среде выполнения CLR или сборщику мусора (такие как дескрипторы объектов ядра, соединения с базами данных, неуправляемая память и так далее). Их приобретение и освобождение не подчиняется правилам, устанавливаемым сборщиком мусора, и выполнения стандартных приемов освобождения памяти, описанных выше, недостаточно для работы с ними.

Для освобождения ресурсов требуется выполнить дополнительный шаг, называемый финализацией (finalization), то есть, вызвать код в объекте, представляющем неуправляемый ресурс, когда этот объект станет ненужным. Часто этот код должен быть выполнен явно (или детерминировано), когда ресурс становится ненужным; но иногда операцию освобождения ресурса можно отложить на неопределенный срок (недетерминировано).

Детерминированная финализация вручную

Представьте фиктивный класс `File`, служащий оберткой вокруг дескриптора файла, возвращаемого функциями Win32. Класс имеет

поле типа `System.IntPtr`, хранящее сам дескриптор. Когда файл становится ненужным, должна быть вызвана функция `CloseHandle` из Win32 API, чтобы закрыть файл и освободить связанные с ним системные ресурсы.

При детерминированном подходе к финализации, в класс `File` следует добавить метод, который будет закрывать системный дескриптор. За вызов этого метода целиком и полностью будет отвечать клиент, даже в случае исключения он обязан вызвать закрыть дескриптор и освободить неуправляемый ресурс.

```
class File {
    private IntPtr handle;

    public File(string fileName) {
        handle = CreateFile(...); // P/Invoke-вызов функции CreateFile
                                  // в Win32 API
    }

    public void Close() {
        CloseHandle(handle); // P/Invoke-вызов функции CloseHandle в
                             // Win32 API
    }
}
```

Такой подход достаточно прост и с успехом применяется в неуправляемых окружениях, таких как C++, где ответственность за освобождение ресурсов несет сам клиент. Однако, разработчики приложений для .NET, приученные к практике автоматического освобождения ресурсов, могут посчитать такую модель неудобной. Они привыкли ожидать от среды CLR предоставления механизма автоматического освобождения неуправляемых ресурсов.

Автоматическая недетерминированная финализация

Автоматический механизм не может быть детерминированным (явным), так как он полагается на освобождение объектов сборщиком мусора. Недетерминированная природа сборщика мусора, в свою очередь, предполагает недетерминированность финализации. Иногда такое недетерминированное поведение является нежелательным, так как приводит к временной «утечке ресурсов» или удержанию совместно используемого ресурса немного дольше, чем это необходимо. Иногда это вполне допустимо и мы подробнее остановимся на таких ситуациях.

Любой тип может переопределить защищенный (protected) метод `Finalize`, объявленный в классе `System.Object`, чтобы обозначить, что объекты этого типа требуют автоматической финализации. Чтобы обеспечить автоматическую финализацию объектов класса `File`, необходимо реализовать метод `~File`. Этот метод называется финализатором (finalizer) и будет вызываться в момент уничтожения объекта.

> **Примечание.** *Кстати говоря, в языке C# только ссылочные типы (классы) могут определить финализатор, даже при том, что сама среда выполнения CLR не накладывает никаких ограничений. Однако, определять реализацию финализатора имеет смысл только для ссылочных типов, потому что типы значений обрабатываются механизмом сборки мусора, только когда они находятся в упакованном виде (подробнее об упаковке типов значений рассказывается в главе 3). Когда объект типа значения размещается на стеке, он никогда не добавляется в очередь финализации. Когда стек разворачивается в процессе выхода из метода или уничтожения кадра стека в результате исключения, финализаторы типов значений не вызываются.*

Когда создается объект с финализатором, ссылка на него добавляется в специальную очередь, называемую очередью финализации (finalization queue). Эта очередь воспринимается сборщиком мусора как корень, в том смысле, что даже если в приложении не останется ни одной ссылки на объект, он все равно будет удерживаться в памяти очередью финализации.

Когда объект становится ненужным в приложении, сборщик мусора обнаружит, что на объект указывает только одна ссылка – из очереди финализации – и переместит ссылку на объект в другую очередь, называемую очередью объектов, готовых к завершению (f-reachable queue). Эта очередь также воспринимается как корень, поэтому объект по-прежнему будет удерживаться в памяти.

Во время сборки мусора финализация объектов не выполняется. Эту процедуру выполняет специальный поток, называемый потоком финализации (finalizer thread) и создаваемый в ходе инициализации среды выполнения CLR (для каждого процесса создается один поток финализации, независимо от используемой разновидности сборщика мусора, который выполняется с приоритетом `THREAD_PRIORITY_HIGHEST`). Этот поток постоянно находится в ожидании появления события финализации (finalization event). Это событие посылается сборщиком мусора, если хотя бы один объект был перемещен им в очередь объектов, готовых к завершению. Поток финализации удаляет ссылку из очереди и одновременно выполняет метод-финализатор объекта. В следующем цикле сборки мусора сборщик обнаружит

отсутствие ссылок на объект и освободит занимаемую им память. На рис. 4.18 показан весь путь перемещения объекта:

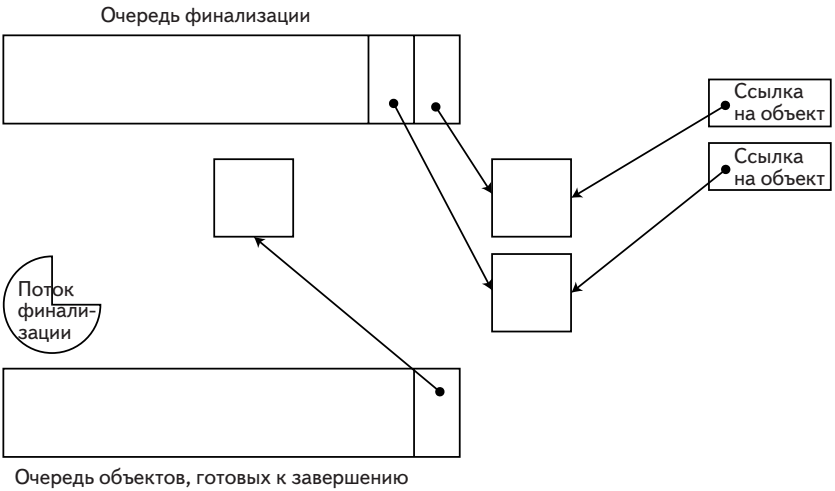

Рис. 4.18. Ссылки на объекты, имеющие метод-финализатор, сохраняются в очереди финализации. Когда объекты выходят из употребления, сборщик мусора перемещает ссылки на них в очередь объектов, готовых к завершению. Поток финализации «просыпается» и вызывает методы-финализаторы объектов, после чего они утилизируются.

Почему метод-финализатор объекта нельзя вызвать в сборщике мусора, а обязательно нужно отложить для выполнения в асинхронном потоке? Кому-то может показаться, что при таком подходе можно будет избавиться от очереди объектов, готовых к завершению, или от дополнительного потока финализации, и устранить лишние накладные расходы. Основной риск такой упрощенной схемы связан с тем, что для работы финализаторов (определяемых пользователями) может потребоваться достаточно много времени, что затормозит процесс сборки мусора и, как следствие, затормозит прикладные потоки. Кроме того, обработка исключений, возникающих в процессе сборки мусора, реализуется очень сложно, а реализовать обработку операций выделения памяти, которые могут выполняться финализаторами во время сборки мусора, еще сложнее. Поэтому, из соображений надежности, сборщик мусора не выполняет финализацию объектов, а делегирует ее специальному потоку.

Ловушки недетерминированной финализации

Модель финализации, описанная выше, влечет за собой множество накладных расходов, отрицательно сказывающихся на производительности. Некоторые из них являются незначительными, но другие обязательно должны учитываться при принятии решения о возможности применения модели финализации для освобождения ресурсов.

- Объекты с финализаторами гарантированно достигнут поколения 1, что увеличивает вероятность проявления эффекта «кризиса среднего возраста» и необходимость выполнения полной сборки мусора.
- Создание объектов с финализаторами стоит несколько дороже, потому что дополнительно они добавляются в очередь финализации. При этом возникает необходимость пользоваться механизмами синхронизации на случай выполнения в многопроцессорной среде. Впрочем, эта цена выглядит незначительной в сравнении с другими проблемами.
- Большая нагрузка на поток финализации (при большом количестве объектов, требующих финализации) может вызывать утечки памяти. Если приложение создает объекты с более высокой скоростью, чем поток финализации способен финализировать их, приложение будет постоянно терять память, занятую объектами, которые ожидают финализации.

Ниже приводится реализация прикладного потока выполнения, создающего объекты с более высокой скоростью, чем поток финализации способен их финализировать, из-за блокировок, выполняемых финализатором. Это вызывает систематическую утечку памяти.

```
class File2 {
    public File2() {
        Thread.Sleep(10);
    }
    ~File2() {
        Thread.Sleep(20);
    }
    // "Утекающие" данные:
    private byte[] data = new byte[1024];
}

class Program {
    static void Main(string[] args) {
```

```
        while (true) {
            File2 f = new File2();
        }
    }
}
```

Эксперимент с утечками, возникающими в результате финализации

В этом эксперименте используется пример приложения, страдающего утечкой памяти, которое мы исследуем прежде, чем обратиться к исходному коду. Не вдаваясь в детали отметим, что утечка связана с некорректным использованием механизма финализации, как показано в примере реализации класса `File2`.

1. Запустите приложение *MemoryLeak.exe* из папки с примерами к этой главе.
2. Запустите Performance Monitor (Системный монитор) и включите следующие счетчики производительности из категории **.NET CLR Memory** (Память CLR .NET) (подробнее о том, что такое Performance Monitor (Системный монитор) и как его запустить, описывается в главе 2): **# Bytes in All Heaps** (Байт во всех кучах), **# Gen 0 Collections** (Сборов «мусора» для поколения 0), **# Gen 1 Collections** (Сборов «мусора» для поколения 1), **# Gen 2 Collections** (Сборов «мусора» для поколения 2), **% Time in GC** (% времени в GC), **Allocated Bytes/sec** (Выделено байт/сек), **Finalization Survivors** (Объектов, оставшихся после сборки мусора), **Promoted Finalization-Memory from Gen 0** (Ожидающая выполнения операции Finalize память, наследуемая из поколения 0).
3. Выполните мониторинг этих счетчиков в течение пары минут. В результате вы должны увидеть, что в итоге значение счетчика **# Bytes in All Heaps** (Байт во всех кучах) растет, хотя иногда бывают небольшие понижения. В целом мониторинг показывает, что приложение постепенно увеличивает расход памяти, а это говорит о вероятной утечке.
4. Обратите внимание, что приложение потребляет память со средней скоростью 1 Мбайт/сек. Это не очень высокая скорость и в данной ситуации сборщику мусора не приходится прикладывать все усилия, чтобы поспеть за приложением.
5. Наконец, отметьте, что значение счетчика **Finalization Survivors** (Объектов, оставшихся после сборки мусора) постоянно остается достаточно высоким. Этот счетчик представляет число объектов, оставшихся после сборки мусора только потому, что они помечены для финализации, но их методы-финализаторы еще не были вызваны (иными словами, эти объекты находятся в очереди объектов, готовых к завершению). Счетчик **Promoted Finalization-Memory from Gen 0** (Ожидающая выполнения операции Finalize память, наследуемая из поколения 0) указывает, что этими объектами занят значительный объем памяти.

Сложив все эти элементы мозаики, легко прийти к выводу, что утечка памяти в приложении скорее всего обусловлена высокой нагрузкой на поток

финализации. Например, приложение может захватывать (и освобождать) ресурсы, требующие финализации, быстрее, чем поток финализации в состоянии справиться с их освобождением. Теперь можете заняться исследованием исходного кода приложения (используйте для этого .NET Reflector, ILSpy или любой другой декомпилятор) и убедиться, что утечка памяти действительно связана с финализацией и ее источник находится в классах `Employee` и `Schedule`.

Помимо проблем производительности, автоматическая, недетерминированная финализация также часто является источником ошибок, которые очень сложно найти и исправить. Эти ошибки обусловлены асинхронной природой и неопределенностью порядка финализации множества объектов.

Представьте объект A, хранящий ссылку на финализируемый объект B. Из-за того, что порядок финализации объектов не определен, объект A не должен строить предположения о том, когда будет вызван его финализатор, объект B может оказаться недоступным для использования к этому моменту. Например, экземпляр класса `System.IO.StreamWriter` может хранить ссылку на экземпляр класса `System.IO.FileStream`. Оба экземпляра владеют ресурсами, требующими финализации: экземпляр `StreamWriter` содержит буфер, который должен быть вытолкнут в поток, представленный экземпляром `FileStream`, а экземпляр `FileStream` содержит дескриптор файла, который следует закрыть. Если экземпляр `StreamWriter` будет финализирован первым, он вытолкнет буфер в пока еще доступный экземпляр `FileStream`, а затем будет финализирован объект `FileStream`, который благополучно закроет файл. Однако, из-за того, что порядок финализации не определен, может сложиться обратная ситуация: экземпляр `FileStream` будет финализирован первым и закроет дескриптор файла, а когда дело дойдет до финализации экземпляра `StreamWriter`, он попытается вытолкнуть буфер в уже несуществующий экземпляр `FileStream`. Это – неразрешимая проблема, поэтому в .NET Framework она «решается» за счет отсутствия метода-финализатора в классе `StreamWriter`, и осуществления только детерминированной финализации. Если клиент забудет закрыть объект `StreamWriter`, содержимое внутреннего буфера будет потеряно.

Совет. *Для пар ресурсов имеется возможность определять порядок их финализации, если один из ресурсов наследует абстрактный класс* `System.Runtime.ConstrainedExecution.CriticalFinalizerObject`, *определяющий, что его метод-финализатор является критическим. Этот специальный базовый класс гарантирует, что его метод-финализатор*

будет вызван только после некритичных методов-финализаторов. Данная возможность используется такими парами ресурсов, как `System.IO.FileStream` *и* `Microsoft.Win32.SafeHandles.SafeFileHandle`*, а также* `System.Threading.EventWaitHandle` *и* `Microsoft.Win32.SafeHandles.SafeWaitHandle`.

Существует еще одна проблема, связанная с асинхронной природой процедуры финализации, выполняемой в выделенном потоке. Метод-финализатор может попытаться приобрести блокировку, удерживаемую прикладным кодом, когда само приложение ожидает завершения финализации, инициированной вызовом `GC.WaitForPendingFinalizers()`. Единственное решение этой проблемы – приобретать блокировку с предельным временем ожидания и прекращать операцию, если она не может быть приобретена.

Третья проблема связана со стремлением сборщика мусора освободить память как можно скорее. Взгляните на следующий код, представляющий наивную реализацию класса `File` с методом-финализатором, закрывающим дескриптор файла:

```
class File3 {
    Handle handle;
    public File3(string filename) {
        handle = new Handle(filename);
    }
    public byte[] Read(int bytes) {
        return Util.InternalRead(handle, bytes);
    }
    ~File3() {
        handle.Close();
    }
}

class Program {
    static void Main() {
        File3 file = new File3("File.txt");
        byte[] data = file.Read(100);
        Console.WriteLine(Encoding.ASCII.GetString(data));
    }
}
```

Этот невинный фрагмент может проявлять очень неприятное поведение. Метод `Read` может потратить на чтение массу времени, при этом он использует только дескриптор, хранящийся в объекте, но не сам объект. Правила определения локальных переменных, как активных корней, утверждают, что локальная переменная, удерживаемая клиентом, больше не считается таковой, как только будет произве-

ден вызов метода `Read`. То есть, объект будет считаться пригодным к утилизации и его метод-финализатор может быть вызван до того, как метод `Read` вернет управление! В этом случае дескриптор файла может оказаться закрытым прямо в процессе операции чтения или даже непосредственно перед ней.

Финализатор может быть не вызван никогда

Даже при том, что процедура финализации считается «пуленепробиваемой», гарантирующей освобождение ресурсов, среда выполнения CLR не гарантирует вызов метода-финализатора во всех возможных ситуациях.

Например, финализация не выполняется, когда работа процесса грубо прерывается. Если пользователь закроет процесс с помощью **Task Manager** (Диспетчер задач) или приложение вызовет функцию `TerminateProcess`, финализаторы не получат возможность освободить ресурсы. Поэтому нельзя слепо доверять, что финализаторы освободят ресурсы, пересекающие границы процессов (например, удалят файлы на диске или запишут определенные данные в базу данных).

Менее очевидная ситуация – когда приложение столкнется с нехваткой памяти и окажется на грани аварийного завершения. Обычно мы ожидаем, что финализаторы будут выполнены даже в случае исключения, но что если финализатор некоторого класса еще ни разу не вызывался и JIT-компилятору еще предстоит скомпилировать его? Для компиляции метода-финализатора JIT-компилятору потребуется выделить память, которой уже нет. Эту проблему можно решить, воспользовавшись поддержкой предварительной компиляции в .NET (с помощью инструмента NGEN), или унаследовать класс `CriticalFinalizerObject`, гарантирующий компиляцию финализатора во время загрузки типа.

Наконец, среда выполнения CLR ограничивает время выполнения финализаторов, вызываемых на этапе завершения процесса или в ходе выгрузки `AppDomain`. В этих ситуациях (наступление которых определяется с помощью `Environment.HasShutdownStarted` или `AppDomain.IsFinalizingForUnload()`) каждому финализатору отводится примерно две секунды, а всем финализаторам в общем – около 40 секунд. При превышении любого из этих пределов финализаторы могут быть не выполнены. Наступление этой ситуации можно определить с помощью значения `BreakOnFinalizeTimeout`. За дополнительной информацией обращайтесь к статье Саши Голдштейн (Sasha Goldshtein) «Debugging Shutdown Finalization Timeout» (http://blog.sashag.net/archive/2008/08/27/debugging-shutdownfinalization-timeout.aspx, 2008).

Шаблон реализации метода Dispose

Мы познакомились с некоторыми проблемами и ограничениями реализации недетерминированной финализации. Теперь самое время

вновь вернуться к ее альтернативе – детерминированной финализации – упоминавшейся выше.

Главная проблема детерминированной финализации заключается в ответственности клиента за правильное использование объекта. Это противоречит объектно-ориентированной парадигме, согласно которой объект сам отвечает за свое состояние. Эта проблема не может быть решена полностью, потому что автоматическая финализация всегда недетерминирована. Однако мы можем ввести механизм соблюдения соглашений, который будет стараться гарантировать детерминированную финализацию, что упростит использование объектов для клиентов. В исключительных случаях мы будем предоставлять возможность автоматической финализации, несмотря на все недостатки, описанные в предыдущем разделе.

Типичные соглашения, устанавливаемые фреймворком .NET Framework, требуют, чтобы объект с детерминированной финализацией реализовал интерфейс `IDisposable`, определяющий единственный метод `Dispose`. Этот метод должен выполнять детерминированную финализацию и освобождать неуправляемые ресурсы.

Клиенты объекта, реализующего интерфейс `IDisposable`, обязаны вызвать метод `Dispose`, когда он станет не нужен. В C# этого можно добиться с помощью блока `using`, обертывающего фрагмент кода, который использует объект, конструкцией `try...finally` и вызывает `Dispose` внутри блока `finally`.

Такой согласительной модели вполне достаточно, если вы полностью доверяете клиентам. Однако часто мы не можем полагаться, что клиенты будут вызывать метод `Dispose` явно, и должны предусмотреть запасной вариант, чтобы предотвратить утечку ресурсов. Этого можно добиться, реализовав поддержку автоматической финализации, но здесь возникает новая проблема: если клиент явно вызовет метод `Dispose` и позднее будет вызван метод-финализатор, объект попытается освободить ресурс дважды. Кроме того, сама идея реализации детерминированной финализации возникла из-за стремления избежать ловушек автоматической финализации!

Нам необходим некоторый способ сообщить сборщику мусора, что неуправляемые ресурсы уже были освобождены и автоматическая финализация для данного объекта не требуется. Таким способом является вызов метода `GC.SuppressFinalize`, который запретит финализацию, установив флаг в слове заголовка объекта (подробности о заголовке объекта см. в главе 3). Объект все еще будет находиться в очереди финализации, но большая часть накладных расходов, связан-

ных с финализацией, будет устранена, потому что память, занимаемая объектом, будет освобождена в первом же цикле сборки мусора и он никогда не попадет в руки потока финализации.

Наконец, нам необходим механизм, с помощью которого можно было бы известить клиента о вызове метода-финализатора, свидетельствующем, что не был использован механизм детерминированной финализации (более эффективный, предсказуемый и надежный). Сделать это можно с помощью `System.Diagnostics.Debug.Assert` или некоторого фреймворка журналирования.

В следующем фрагменте приводится черновая версия класса-обертки для работы с неуправляемым ресурсом, которая следует всем этим соглашениям (существуют и другие проблемы, которые следует учитывать, например, если наш класс наследует другой класс, так же владеющий неуправляемым ресурсом):

```
class File3 : IDisposable {
    Handle handle;
    public File3(string filename) {
        handle = new Handle(filename);
    }
    public byte[] Read(int bytes) {
        Util.InternalRead(handle, bytes);
    }
    ~File3() {
        Debug.Assert(false, "Do not rely on finalization! Use Dispose!");
        handle.Close();
    }
    public void Dispose() {
        handle.Close();
        GC.SuppressFinalize(this);
    }
}
```

Примечание. *Шаблон финализации, описанный в этом разделе, называется шаблоном реализации метода* `Dispose` *и охватывает дополнительные области, такие как взаимодействие между производным классом и его предком, так же требующим финализации. За дополнительной информацией о шаблоне Dispose обращайтесь к документации на сайте MSDN. Так получилось, что C++/CLI уже реализует шаблон Dispose, как часть встроенного синтаксиса:* `!File` *– это финализатор C++/CLI, а* `~File` *– это реализация* `IDisposable.Dispose` *в C++/CLI. Все необходимое для вызова базового класса и подавления финализации, осуществляется компилятором автоматически.*

Гарантировать правильную реализацию шаблона Dispose не так сложно, как принудить клиента вашего класса использовать детер-

минированную финализацию вместо автоматической. Прием на основе `System.Diagnostics.Debug.Assert`, обозначенный выше, является достаточно сильным средством в этом отношении. Как вариант, для определения некорректного использования рескрсов можно использовать статический анализ кода.

Воскрешение

Механизм финализации дает объектам возможность выполнить произвольный код после того, как он выйдет из употребления. Данную возможность можно использовать для создания в приложении новой ссылки на объект, оживляя его после того, как он был объявлен мертвым. Эта возможность называется воскрешением (resurrection).

Возможность воскрешения может пригодиться во множестве ситуаций, но ее следует использовать с большой осторожностью. Основная проблема состоит в том, что другие объекты, на которые ссылался воскрешаемый объект, могут оказаться в недопустимом состоянии из-за того, что их финализаторы уже были выполнены. Эту проблему нельзя решить без полной повторной инициализации всех объектов, на которые ссылается воскрешаемый объект. Другая проблема состоит в том, что финализатор воскрешаемого объекта может быть не вызван без использования туманного метода `GC.ReRegisterForFinalize`, которому следует передать ссылку на воскрешаемый объект (обычно `this`).

Одним из типичных примеров применения механизма воскрешения является использование пула объектов. Прием использования пула объектов предполагает, что объекты будут распределяться из пула и возвращаться в пул, они не будут утилизироваться сборщиком мусора и создаваться заново. Возврат объекта в пул можно реализовать явно или в процессе отложенной финализации.

Слабые ссылки

Слабые ссылки (weak references) – это вспомогательный механизм обслуживания ссылок на управляемые объекты. Типичная ссылка на объект (также называется сильной ссылкой (strong reference)) является очень детерминированной: пока имеется ссылка на объект, он будет продолжать существовать. И это правильно, с точки зрения сборщика мусора.

Однако иногда бывает желательно сохранить невидимой строку, связанную с объектом, не мешая при этом сборщику мусора удалить

объект из памяти. Когда сборщик мусора удалит объект, строка окажется не связанной ни с каким объектом, и мы сможем определить это. Если сборщик мусора еще не прикасался к объекту, с помощью этой строки мы сможем восстановить сильную ссылку на объект и использовать его снова.

Это может пригодиться в разных ситуациях, наиболее типичные из которых перечислены ниже:

- Пользование внешними услугами без сохранения объекта. Такие услуги, как таймеры и события, могут предоставляться объектам без сохранения ссылок на них, что может способствовать устранению типичных причин утечки памяти.
- Автоматическое управление стратегией кеширования или поддержания пула объектов. Кеш может хранить слабые ссылки на недавно использовавшиеся объекты, не мешая их утилизации; пул может быть разделен на две части – основную, минимального размера, хранящую сильные ссылки, и дополнительную, содержащую слабые ссылки.
- Удержание больших объектов в надежде, что они не будут утилизированы. Приложение может хранить слабую ссылку на большой объект, для создания и инициализации которого требуется много времени. Если объект окажется уничтожен сборщиком мусора, приложение сможет повторно создать его; в противном случае его можно использовать повторно.

Механизм слабых ссылок доступен приложению в виде класса `System.WeakReference`, являющегося особым случаем типа `System.Runtime.InteropServices.GCHandle`. Слабая ссылка имеет логическое свойство `IsAlive`, сообщающее о существовании объекта ссылки, и свойство `Target`, которое можно использовать для получения ссылки на целевой объект (если он еще не был утилизирован, в противном случае возвращается значение `null`).

> **Внимание.** *Имейте в виду, что единственным безопасным способом получения строгой ссылки на объект слабой ссылки является свойство* `Target`. *Даже если свойство* `IsAlive` *вернет* `true`, *есть вероятность, что сразу после этого объект будет утилизирован. Чтобы избежать состояния гонки, сначала следует использовать свойство* `Target`, *присвоить возвращаемое им значение сильной ссылке (локальной переменной, полю и так далее) и затем сравнить полученное значение на равенство* `null`. *Свойство* `IsAlive` *следует использовать, только когда необходимо определить факт уничтожения объекта; например, чтобы удалить слабую ссылку из кеша.*

Ниже демонстрируется черновая версия реализации событий на основе слабых ссылок (рис. 4.19). Само событие не может использовать механизм делегатов .NET непосредственно, потому что делегат хранит сильную ссылку на свою цель и это является мешающим обстоятельством. Однако, можно хранить цель делегата (в виде слабой ссылки). Такой подход позволяет избавиться от одной из наиболее типичных причин утечек памяти в .NET, когда по забывчивости события продолжают оставаться зарегистрированными!

```
public class Button {
    private class WeakDelegate {
        public WeakReference Target;
        public MethodInfo Method;
    }
    private List<WeakDelegate> clickSubscribers = new List<WeakDelegate>();

    public event EventHandler Click {
        add {
            clickSubscribers.Add(new WeakDelegate {
                Target = new WeakReference(value.Target),
                Method = value.Method
            });
        }
        remove {
            //...Реализация опущена ради экономии места
        }
    }
    public void FireClick() {
        List<WeakDelegate> toRemove = new List<WeakDelegate>();
        foreach (WeakDelegate subscriber in clickSubscribers) {
            object target = subscriber.Target.Target;
            if (target == null) {
                toRemove.Add(subscriber);
            } else {
                subscriber.Method.Invoke(target, new object[] { this,
EventArgs.Empty });
            }
        }
        clickSubscribers.RemoveAll(toRemove);
    }
}
```

По умолчанию слабые ссылки не отслеживают воскрешение объектов. Чтобы включить отслеживание, используйте перегруженный конструктор, принимающий логический параметр, и передавайте в нем истинное значение, чтобы указать на необходимость слежения за воскрешением. Слабые ссылки, отслеживающие воскрешение, называют длинными слабыми ссылками (long weak references); слабые

ссылки, не отслеживающие воскрешение, называются короткими слабыми ссылками (short weak references) .

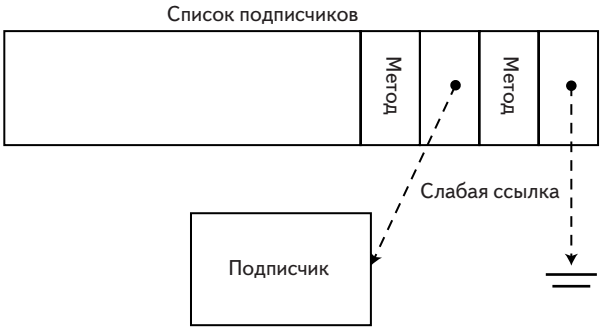

Рис. 4.19. Генератор событий хранит слабые ссылки на всех подписчиков. Если подписчик окажется недостижим для приложения, генератор событий сможет определить это по появлению значения null в слабой ссылке.

Дескрипторы сборщика мусора

Слабые ссылки – это особый случай *дескрипторов сборщика мусора*. Дескриптор сборщика мусора – это значение особого низкоуровневого типа, придающее ссылкам на объекты дополнительные возможности, позволяя:

- сохранять обычную (сильную) ссылку на объект, препятствуя его утилизации; представлено значением `GCHandleType.Normal`;
- сохранять короткую слабую ссылку на объект; представлено значением `GCHandleType.Weak`;
- сохранять длинную слабую ссылку на объект; представлено значением `GCHandleType.WeakTrackResurrection`;
- сохранять ссылку на объект, закрепляя его так, что он не может перемещаться в памяти, и дает возможность получать адрес объекта; представлено значением `GCHandleType.Pinned`;

На практике дескрипторы сборщика GC редко используются непосредственно, но они часто участвуют в результатах профилирования, как еще один тип корней, позволяющих удерживать управляемые объекты.

Взаимодействие со сборщиком мусора

До сих пор мы рассматривали приложение, как пассивный элемент, в отношении сборщика мусора. Мы изучали реализацию сборщика му-

сора и знакомились с важнейшими оптимизациями, выполняемыми автоматически и не требующими практически никаких действий с нашей стороны. В этом разделе мы исследуем доступные инструменты активного воздействия на сборщика мусора, с целью повысить производительность наших приложений и получить диагностическую информацию, недоступную иными способами.

Класс System.GC

Класс `System.GC` – это основная точка взаимодействий со сборщиком мусора .NET из управляемого кода. Он содержит множество методов, управляющих поведением сборщика мусора и возвращающих диагностическую информацию о нем.

Диагностические методы

Диагностические методы класса `System.GC` возвращают информацию о состоянии сборщика мусора. Предполагается, что эти методы будут использоваться для нужд диагностики проблем и отладки; не используйте их для принятия решений при работе приложения в обычном режиме. Информация, возвращаемая этими методами, доступна также в виде счетчиков производительности из категории **.NET CLR Memory** (Память CLR .NET).

- Метод `GC.CollectionCount` возвращает количество циклов сборки мусора, выполненных для указанного поколения с момента запуска приложения. Его можно использовать для определения факта сборки мусора в указанном поколении в некотором интервале времени.
- Метод `GC.GetTotalMemory` возвращает объем занятой динамической памяти в байтах. Если передать методу аргумент со значением `true`, предварительно будет выполнена полная сборка мусора, чтобы возвращаемое значение точно соответствовало объему памяти, которая не может быть освобождена.
- Метод `GC.GetGeneration` возвращает номер поколения, которому принадлежит указанный объект. Обратите внимание, что в текущей реализации CLR объекты не всегда перемещаются между поколениями при сборке мусора.

Уведомления

Появившийся в версии .NET 3.5 SP1 прикладной интерфейс уведомлений сборщика мусора дает приложениям возможность заранее

узнать о надвигающейся полной сборке мусора. Этот прикладной интерфейс доступен только при использовании непараллельного сборщика мусора и предназначен для использования в приложениях, в которых наблюдаются длительные паузы, вызванные сборкой мусора, и где требуется реализовать распределение работы или рассылку уведомлений, когда ожидается наступление очередной паузы.

Сначала приложение, заинтересованное в получении уведомлений от сборщика мусора, вызывает метод `GC.RegisterForFullGCNotification` и передает ему два пороговых значения (числа от 1 до 99). Эти значения указывают, как рано приложение должно извещаться о наступлении сборки мусора, и основаны на пороговых значениях поколения 2 и кучи больших объектов. Проще говоря, чем больше значение, тем больше временной разрыв между уведомлением и фактической сборкой мусора. При маленьких значениях появляется риск не получить уведомление из-за того, что разрыв времени окажется слишком маленьким и механизм уведомлений просто не успеет сработать.

Затем приложение вызывает метод `GC.WaitForFullGCApproach`, который блокируется до появления уведомления, выполняет подготовительные операции перед сборкой мусора и вызывает метод `GC.WaitForFullGCComplete`, который блокируется до окончания сборки мусора. Поскольку все методы выполняются синхронно, есть смысл вызывать их в фоновом потоке и возбуждать событие в основном потоке выполнения программы, как показано ниже:

```
public class GCWatcher {
    private Thread watcherThread;

    public event EventHandler GCApproaches;
    public event EventHandler GCComplete;

    public void Watch() {
        GC.RegisterForFullGCNotification(50, 50);
        watcherThread = new Thread(() => {
            while (true) {
                GCNotificationStatus status = GC.WaitForFullGCApproach();
                // Код обработки ошибок опущен
                if (GCApproaches != null) {
                    GCApproaches(this, EventArgs.Empty);
                }
                status = GC.WaitForFullGCComplete();
                // Код обработки ошибок опущен
                if (GCComplete != null) {
                    GCComplete(this, EventArgs.Empty);
                }
            }
        });
```

```
        watcherThread.IsBackground = true;
        watcherThread.Start();
    }

    public void Cancel() {
        GC.CancelFullGCNotification();
        watcherThread.Join();
    }
}
```

За дополнительной информацией и подробными примерами использования прикладного интерфейса уведомлений для распределения нагрузки на сервер обращайтесь к документации на сайте MSDN: http://msdn.microsoft.com/ru-ru/library/cc713687.aspx.

Управляющие методы

Метод `GC.Collect` предписывает сборщику мусора выполнить сборку в указанном поколении (включая все более молодые поколения). Начиная с версии .NET 3.5 (а также в версиях .NET 2.0 SP1 и .NET 3.0 SP1), метод `GC.Collect` был перегружен параметром типа перечисления `GCCollectionMode`. Это перечисление определяет следующие значения.

- `GCCollectionMode.Forced` – вынуждает сборщика мусора немедленно выполнить сборку, синхронно с текущим потоком выполнения. Метод возвращает управление только по завершении сборки мусора.
- `GCCollectionMode.Optimized` – позволяет сборщику мусора самому решить, насколько оправданно будет выполнить сборку в данный момент. Сборщик мусора может решить отложить сборку. Это – рекомендуемый режим, если вашей целью является помочь сборщику мусора, подсказывая, когда сборка мусора является желательной. Для диагностики или когда требуется принудительно выполнить полную сборку мусора, чтобы удалить определенный объект, используйте значений `GCCollectionMode.Forced`.
- Начиная с версии CLR 4.5 значение `GCCollectionMode.Default` действует эквивалентно значению `GCCollectionMode.Forced`.

Принудительная сборка мусора редко применяется на практике. Оптимизации, описанные в этой главе, в значительной степени опираются на динамические настройки и эвристики, хорошо протестированные в приложениях самых разных типов. Мы не рекомендуем вызывать принудительную сборку мусора и даже советовать сборщику

мусора начать ее (передачей значения `GCCollectionMode.Optimized`) без явной необходимости. А теперь, после всего сказанного выше, можно обозначить ситуации, когда можно подумать об использовании возможности выполнять принудительную сборку мусора.

- По завершении нечастых операций, требующих больших объемов памяти, эта память может быть освобождена. Если приложение не вызывает полную сборку мусора достаточно часто, занятая память может довольно долго оставаться в поколении 2 (или в куче больших объектов). В этой ситуации, когда достоверно известно, что большой объем памяти больше не будет использоваться, имеет смысл принудительно запустить сборку мусора, чтобы избежать вытеснения страниц оперативной памяти в файл подкачки.
- При использовании сборщика мусора в режиме работы с низкими задержками, желательно принудительно запустить сборку мусора в безопасной точке программы, когда известно, что все критичные ко времени операции уже выполнены и приложение может позволить себе небольшую паузу для сборки мусора. Длительная работа в режиме с низкими задержками без выполнения сборки мусора может привести к исчерпанию памяти. Вообще говоря, если приложение слишком чувствительно к выбору моментов времени на сборку мусора, разумно принудительно запускать сборку мусора в периоды простоя, чтобы избежать лишних накладных расходов в периоды активных действий.
- Когда для освобождения неуправляемых ресурсов используется недетерминированная финализация, часто бывает желательно приостановить работу, пока все такие ресурсы не будут освобождены. Этого можно добиться последовательным вызовом методов `GC.Collect` и `GC.WaitForPendingFinalizers`. В таких ситуациях всегда предпочтительнее использовать детерминированную финализацию, но зачастую у нас отсутствует возможность управлять внутренними классами, выполняющими фактическую финализацию.

Примечание. *В версии .NET 4.5 появилась еще одна перегруженная версия метода* `GC.Collect` *с дополнительным логическим параметром:* `GC.Collect(int generation, GCCollectionMode mode, bool blocking)`. *Этот параметр управляет необходимостью блокировки на время выполнения сборки мусора (по умолчанию включена, в противном случае сборка выполняется асинхронно, в фоновом потоке).*

Ниже перечислены другие методы управления сборщиком мусора.

- Методы `GC.AddMemoryPressure` и `GC.RemoveMemoryPressure` можно использовать для уведомления сборщика мусора о выделении или освобождении неуправляемой памяти в текущем процессе. Уведомление о выделении памяти сообщает сборщику мусора, какой объем неуправляемой памяти был выделен процессу. Сборщик мусора может использовать эту информацию для настройки агрессивности и частоты сборки мусора или игнорировать ее. При освобождении неуправляемой памяти можно уведомить сборщик мусора о снятии лишнего давления на память.
- Метод `GC.WaitForPendingFinalizers` блокирует выполнение текущего потока, пока не завершатся все методы-финализаторы. Этот метод следует использовать с большой осторожностью, потому что он может приводить к взаимоблокировкам. Например, если главный поток выполнения окажется заблокирован в вызове `GC.WaitForPendingFinalizers` в тот момент, когда он удерживает некоторую блокировку, и один из активных финализаторов попытается приобрести ту же самую блокировку, возникнет состояние взаимоблокировки. Так как метод `GC.WaitForPendingFinalizers` не имеет параметра, ограничивающего предельное время ожидания, финализаторы, пытающиеся приобрести блокировку, должны использовать таймаут для обработки ошибочных ситуаций.
- Методы `GC.SuppressFinalize` и `GC.ReRegisterForFinalize` используются совместно с механизмами финализации и воскрешения. Они обсуждаются в разделе с описанием механизма финализации, в этой главе.
- Начиная с версии .NET 3.5 (доступен также в версиях .NET 2.0 SP1 и .NET 3.0 SP1) появился еще один интерфейс к сборщику мусора в виде класса `GCSettings`, обсуждавшегося выше, который позволяет управлять поведением сборщика мусора и переключаться в режим с низкой задержкой.

Описание других методов и свойств класса `System.GC`, не упомянутые в этом разделе, ищите в документации на сайте MSDN.

Взаимодействие с применением интерфейсов размещения CLR

В предыдущем разделе мы исследовали методы диагностики и управления сборщиком мусора, доступные из управляемого кода. Однако

степень контроля, предлагаемая этими методами, оставляет желать лучшего. Кроме того, нет никакого механизма уведомлений, с помощью которого можно было бы узнать, когда запускается сборка мусора.

Эти проблемы невозможно решить с применением только управляемого кода. Поэтому, для реализации более полного управления сборщиком мусора необходимо выполнить размещение CLR. Этот прием обеспечивает множество механизмов управления памятью в .NET.

- Интерфейс `IHostMemoryManager` и связанный с ним интерфейс `IHostMalloc` объявляют методы обратного вызова, которые среда выполнения CLR использует для выделения сегментов сборщику мусора, для приема уведомлений о нехватке памяти, для выделения памяти, неподконтрольной сборщику мусора (например, для кода, сгенерированного JIT-компилятором) и для оценки доступного объема свободной памяти. Например, этот интерфейс можно использовать, чтобы обеспечить удовлетворение всех запросов на выделение памяти из областей ОЗУ, которые не могут вытесняться в файл подкачки. В этом заключается суть открытого проекта Non-Paged CLR Host (http://nonpagedclrhost.codeplex.com/, 2008).
- Интерфейс `ICLRGCManager` объявляет методы управления сборщиком мусора и получения статистических данных о его работе. Его можно использовать для запуска сборки мусора из размещающего кода, для получения статистики (доступной также посредством счетчиков производительности в категории **.NET CLR Memory** (Память CLR .NET)) и для инициализации начальных пороговых значений сборщика мусора, включая размер сегмента сборщика мусора и максимальный размер поколения 0.
- Интерфейс `IHostGCManager` объявляет методы для приема уведомлений о запуске и окончании сборки мусора, а также о приостановке потока выполнения, чтобы сборщик мусора мог продолжить работу.

Ниже приводится небольшой фрагмент кода из открытого проекта Non-Paged CLR Host, демонстрирующий, как можно выполнить настройку выделения сегментов по запросам размещаемой среды выполнения CLR и как обеспечить выделение страниц памяти, не вытесняемых в файл подкачки:

```
HRESULT __stdcall HostControl::VirtualAlloc(
    void* pAddress, SIZE_T dwSize, DWORD flAllocationType,
```

```
    DWORD flProtect, EMemoryCriticalLevel dwCriticalLevel, void** ppMem) {

    *ppMem = VirtualAlloc(pAddress, dwSize, flAllocationType, flProtect);
    if (*ppMem == NULL) {
        return E_OUTOFMEMORY;
    }
    if (flAllocationType & MEM_COMMIT) {
        VirtualLock(*ppMem, dwSize);
        return S_OK;
    }
}

HRESULT __stdcall HostControl::VirtualFree(
    LPVOID lpAddress, SIZE_T dwSize, DWORD dwFreeType) {

    VirtualUnlock(lpAddress, dwSize);
    if (FALSE == VirtualFree(lpAddress, dwSize, dwFreeType)) {
        return E_FAIL;
    }
    return S_OK;
}
```

За информацией о дополнительных интерфейсах размещения CLR, имеющих отношение к сборщику мусора, включая `IGCHost`, `IGCHostControl` и `IGCThreadControl`, обращайтесь к документации на сайте MSDN.

Триггеры сборщика мусора

Мы познакомились с рядом причин, вызывающих сборку мусора, но мы нигде не перечисляли их вместе. Ниже приводится список триггеров, используемых средой CLR для определения необходимости запуска сборки мусора, перечисленных в порядке убывания их значимости:

1. Заполнение области памяти для поколения 0. Это происходит всякий раз, когда приложение создает новый объект в небольшой области памяти.
2. Заполнение кучи больших объектов достигло порогового значения. Это происходит при создании больших объектов.
3. Явный вызов метода `GC.Collect`.
4. Операционная система сообщила о нехватке памяти. Для слежения за утилизацией памяти и соблюдения типичных требований, предъявляемых операционной системой, среда CLR использует Win32 API уведомлений.
5. Выгрузка экземпляра `AppDomain`.

6. Завершение процесса (или CLR). В этом случае производится вырожденная сборка мусора – ничто не интерпретируется как корень, объекты не переносятся в старшие поколения и сжатие динамической памяти не производится. Основная цель этого цикла сборки мусора – выполнить методы-финализаторы.

Эффективные приемы повышения производительности сборки мусора

В этом разделе мы познакомимся с эффективными приемами взаимодействий со сборщиком мусора .NET, а также исследуем различные ситуации, демонстрирующие разные аспекты этих приемов, и покажем, как избежать типичных ловушек.

Модель поколений

Мы познакомились с моделью поколений, как важнейшим средством оптимизации производительности в упрощенной модели сборки мусора, обсуждавшейся выше. Разделение управляемых объектов по поколениям позволяет сборщику мусора чаще уничтожать короткоживущие объекты. Кроме того, наличие отдельной кучи больших объектов решает проблему копирования значительных объемов памяти за счет использования стратегии управления списком свободных блоков.

Теперь мы можем коротко обозначить эффективные приемы взаимодействия с моделью поколений и затем рассмотреть несколько примеров.

- Временные объекты должны быть короткоживущими. Самая худшая ситуация, когда временные объекты попадают в поколение 2, потому что это будет вызывать частые полные сборки мусора.
- Большие объекты должны быть долгоживущими или память для них должна выделяться из пула. Сборка мусора в куче больших объектов эквивалентна полной сборке мусора.
- Количество ссылок между поколениями должно быть сведено к минимуму.

Далее описываются ситуации, демонстрирующие риски, связанные с явлением «кризиса среднего возраста». В приложении монито-

ринга с графическим интерфейсом, реализованном одним из наших клиентов, в главном окне постоянно отображалось 20 000 записей из журнала. Каждая запись содержала информацию об уровне важности, краткое сообщение и дополнительную контекстную информацию (иногда весьма значительного объема). Эти 20 000 записей непрерывно замещались новыми записями.

Из-за большого количества отображаемых записей, большинство их переживало два цикла сборки мусора и достигало поколения 2. Однако эти записи, по сути, являлись короткоживущими объектами, и замещались новыми вскоре после попадания в поколение 2, во всей полноте проявляя феномен «кризиса среднего возраста». В результате приложению приходилось выполнять сотни циклов полной сборки мусора в минуту, затрачивая на это примерно 50% всего времени работы кода.

Выполнив предварительный анализ, мы пришли к заключению, что отображение на экране 20 000 записей не является насущной необходимостью. Мы сократили количество отображаемых записей до 1000 и реализовали пул объектов, чтобы повторно использовать существующие объекты записей вместо создания новых. Эти меры позволили уменьшить потребление памяти приложением, но, что еще более важно, уменьшить время на сборку мусора до 0.1%, при этом частота полной сборки мусора уменьшилась до одного раза в несколько минут.

С другим проявлением «кризиса среднего возраста» мы столкнулись в реализации одного из наших веб-серверов. Система веб-серверов была разделена на несколько фронтальных серверов, принимающих запросы. Для обработки запросов эти фронтальные серверы синхронно вызывали веб-службы, выполняющиеся на внутренних серверах.

В тестовом окружении вызов веб-службы между фронтальным и внутренним сервером занимал порядка нескольких миллисекунд. Это обеспечивало непродолжительность существования объектов HTTP-запросов и быстрое их уничтожение.

В промышленном окружении вызов веб-службы часто выполнялся гораздо дольше, из-за сетевых задержек, высокой нагрузки на внутренние серверы и других факторов. Ответ на запрос все еще возвращался в течение долей секунды, и эта сторона реализации не требовала оптимизации, потому что человек все равно не почувствовал бы разницу. Однако каждую секунду в систему поступала масса запросов, вследствие чего срок жизни каждого объекта запроса и про-

тяженность графа объектов увеличились до такой степени, что эти объекты переживали несколько циклов сборки мусора и легко достигали поколения 2.

Важно заметить, что способность сервера обрабатывать запросы не страдала от того, что объекты жили чуть дольше: нагрузка на память была вполне приемлемой и клиенты не чувствовали разницы, когда ответы возвращались на доли секунды позже. Однако по способности масштабирования был нанесен серьезный удар, потому что фронтальные серверы тратили на сборку мусора до 70% времени.

Чтобы разрешить эту проблему, можно перейти на асинхронную обработку запросов или освобождать объекты запросов настолько быстро, насколько это возможно (перед выполнением синхронного обращения к службе). Применение этих двух приемов одновременно позволило сократить время на сборку мусора до 3% и повысить способность сайта к масштабированию 3 раза!

Наконец, представьте простую систему отображения двумерной графики. В подобных системах поверхность рисования является долгоживущей сущностью, постоянно перерисовывающей себя, создавая и замещая короткоживущие пиксели разного цвета и с разной степенью прозрачности.

Если эти пиксели представить в виде объектов ссылочного типа, мы не только удвоили или утроили бы расход памяти, но также получили бы ссылки между поколениями и, как результат, огромный граф объектов, представляющих пиксели. Единственный выход – использовать для представления пикселей типы значений, которые помогут уменьшить расход памяти в 2–3 раза и сэкономить время на сборку мусора в десятки раз.

Закрепление

Выше мы познакомились с приемом закрепления объектов в памяти, позволяющим гарантировать безопасную передачу адресов управляемых объектов неуправляемому коду. После закрепления объект будет оставаться в том же самом месте в памяти, препятствуя тем самым дефрагментации памяти сборщиком мусора.

Учитывая это, мы можем коротко обозначить ситуации, когда прием закрепления объектов будет наиболее эффективен.

- Закрепляйте объекты на как можно более короткий срок. Закрепление обходится достаточно дешево, если сборка мусора не производится, пока объект остается закрепленным. Если требуется передать закрепленный объект неуправляемому

коду, который может выполняться достаточно долго, подумайте о возможности копирования объекта в неуправляемую память, вместо его закрепления.

- Лучше закрепить несколько больших буферов, чем много маленьких объектов, даже если при этом вам придется организовать управление небольшими фрагментами буферов вручную. Большие объекты не перемещаются в памяти, что уменьшает фрагментацию памяти, вызываемую закреплением.
- Закрепляйте и повторно используйте старые объекты, созданные на этапе запуска приложения. Старые объекты редко перемещаются в памяти, что уменьшает фрагментацию памяти, вызываемую закреплением.
- Если приложение интенсивно использует прием закрепления, подумайте о выделении блока неуправляемой памяти. Неуправляемая память доступна неуправляемому коду непосредственно, избавляя от необходимости выполнять закрепление и платить производительностью за сборку мусора. Используя небезопасный код (указатели C#), легко можно организовать операции с блоками неуправляемой памяти из управляемого кода без копирования данных в управляемые структуры. Выделение неуправляемой памяти из управляемого кода обычно выполняется с применением класса `System.Runtime.InteropServices.Marshal`. (Подробности см. в главе 8.)

Финализация

Имея знания, полученные в разделе, описывающем процедуру финализации, совершенно очевидно, что поддержка автоматической недетерминированной финализации в .NET оставляет желать лучшего. Лучшее, что можно посоветовать, – всегда, когда это возможно, использовать детерминированную финализацию, и прибегать к недетерминированной финализации только в исключительных случаях.

Ниже перечислены наиболее эффективные приемы, касающиеся использования поддержки финализации в приложениях.

- Используйте детерминированную финализацию и реализуйте интерфейс `IDisposable`, чтобы гарантировать, что клиенты наверняка знали, чего ожидать от вашего класса. Используйте `GC.SuppressFinalize` в своих реализациях метода `Dispose`, чтобы исключить возможность вызова метода-финализатора, когда в этом нет необходимости.

- Реализуйте методы-финализаторы и используйте в них метод `Debug.Assert` или средства журналирования, чтобы клиент мог узнать о некорректном применении ваших классов.
- При реализации сложных объектов, оформляйте ресурсы, требующие финализации, в виде отдельных классов (классическим примером может служить тип `System.Runtime.InteropServices.SafeHandle`). Это гарантирует, что только данный маленький тип, обертывающий неуправляемый ресурс, переживет лишние циклы сборки мусора, а основной объект может быть безопасно удален сборщиком мусора, как только выйдет из употребления.

Разные советы и рекомендации

В этом разделе мы коротко рассмотрим разнообразные советы и рекомендации, которые не могут быть отнесены к основным темам, обсуждавшимся в этой главе.

Типы значений

Когда это возможно, отдавайте предпочтение типам значений. Мы исследовали некоторые особенности типов значений и ссылочных типов в главе 3. Но кроме них типы значений обладают еще рядом свойств, снижающих стоимость сборки мусора в приложениях.

- Типы значений практически не влекут накладных расходов на размещение в памяти, когда их экземпляры создаются на стеке, в виде локальных переменных. В этом случае выделение памяти происходит за счет расширения кадра стека, который создается при входе в метод.
- При размещении экземпляров типов значений в локальных переменных на стеке отпадает необходимость освобождения памяти (сборщиком мусора) – память освобождается автоматически, когда кадр стека разрушается и метод возвращает управление вызывающей программе.
- При встраивании типов значений в ссылочные типы уменьшается стоимость обеих фаз сборки мусора: чем больше объекты, тем меньше затрат на их маркировку, и тем большие объемы памяти приходится копировать каждый раз в фазе чистки, что снижает накладные расходы на копирование множества маленьких объектов.

- Экземпляры типов значений уменьшают расход памяти, так как размещаются в ней более компактно. Кроме того, при встраивании в ссылочные типы, они не требуют использовать ссылки для обращения к ним, что устраняет необходимость хранить дополнительные ссылки. Наконец, при встраивании в ссылочные типы, типы значений повышают локальность доступа – если объект попадет в кеш процессора, содержимое его полей, имеющих тип значения, скорее всего также окажется в кеше.
- Применение типов значений уменьшает количество ссылок между поколениями, благодаря общему уменьшению ссылок в графе объектов.

Графы объектов

Сокращение размеров графа объектов напрямую влияет на объем операций, который должен выполнить сборщик мусора. Простой граф с большими объектами обрабатывается быстрее, чем разветвленный граф с множеством маленьких объектов. Подобная ситуация уже была представлена выше.

Кроме того, сокращение количества локальных переменных ссылочных типов уменьшает размеры локальных таблиц корней, создаваемых JIT-компилятором, что увеличивает скорость компиляции и экономит небольшое количество памяти.

Использование пулов объектов

Пул объектов – это механизм управления памятью и ресурсами вручную, в обход средств, предоставляемых средой выполнения. При использовании пулов объектов, под созданием нового объекта понимается извлечение из пула неиспользуемого объекта, а под его уничтожением – возврат объекта в пул.

Применение пула может существенно повысить производительность, если затраты на выделение и освобождение памяти (исключая расходы на инициализацию и финализацию) превосходят затраты на управление пулом вручную. Например, использование пула для размещения больших объектов, вместо создания и уничтожения их с помощью сборщика мусора, может повысить производительность, так как при этом исключается необходимость выполнять полную сборку мусора.

Примечание. *Фреймворк Windows Communication Foundation (WCF) реализует пул массивов байтов и использует его для хранения и передачи*

сообщений. Фасадом пула служит абстрактный класс `System.ServiceModel.Channels.BufferManager`*, предоставляющий возможность получения массива байтов из пула и возврата его в пул. Фреймворк включает две внутренние реализации абстрактных базовых операций, использующих механизмы управления памятью на основе сборщика мусора и управлением пула буферов. Реализация пула (на момент написания этих строк) обеспечивает управление множеством пулов буферов разных размеров. Похожий прием используется в алгоритме Low-Fragmentation Heap (алгоритм организации динамической памяти с низкой фрагментацией), впервые реализованном в Windows XP.*

Для эффективной реализации пула требуется учитывать, как минимум, следующие факторы:

- количество операций синхронизации, связанных с выделением и освобождением памяти должно быть сведено к минимуму; например, для реализации пула можно было бы использовать структуры данных без блокировок (lock-free) (подробнее о неблокирующей синхронизации рассказывается в главе 6);
- размер пула не может расти до бесконечности, то есть при определенных обстоятельствах лишние объекты должны удаляться из пула с использованием сборщика мусора;
- пул не должен часто переполняться, то есть необходимо применять эвристические алгоритмы, позволяющие определять оптимальный размер пула, исходя из частоты следования запросов.

В большинстве реализаций пулов дополнительные выгоды можно получить от использования механизма выбора последних использовавшихся блоков (Least Recently Used, LRU) при распределении новых объектов, потому что последние использовавшиеся блоки с большей долей вероятности окажутся в кеше процессора.

Реализация пула в .NET должна предусматривать методы выделения и освобождения экземпляров типа, размещаемого в пуле. Мы не можем организовать управление этими операциями на уровне синтаксиса (оператор `new` не предусматривает возможность перегрузки), но мы можем использовать альтернативный API, например, определить метод `Pool.GetInstance`. Возврат объекта в пул лучше реализовать с использованием шаблона Dispose и предусматривать метод-финализатор, как запасной вариант.

В следующем примере приводится чрезвычайно упрощенная реализация пула, а также объявление типа объектов, которые могут помещаться в пул:

```
public class Pool<T> {
    private ConcurrentBag<T> pool = new ConcurrentBag<T>();
    private Func<T> objectFactory;
    public Pool(Func<T> factory) {
        objectFactory = factory;
    }
    public T GetInstance() {
        T result;
        if (!pool.TryTake(out result)) {
            result = objectFactory();
        }
        return result;
    }
    public void ReturnToPool(T instance) {
        pool.Add(instance);
    }
}

public class PoolableObjectBase<T> : IDisposable {
    private static Pool<T> pool = new Pool<T>();

    public void Dispose() {
        pool.ReturnToPool(this);
    }
    ~PoolableObjectBase() {
        GC.ReRegisterForFinalize(this);
        pool.ReturnToPool(this);
    }
}

public class MyPoolableObjectExample : PoolableObjectBase
<MyPoolableObjectExample> {
    ...
}
```

Вытеснение в файл подкачки и выделение неуправляемой памяти

Сборщик мусора в .NET автоматически освобождает неиспользуемую память. Поэтому по определению он не может обеспечить универсальное решение всех потребностей управления памятью, которые могут возникать в приложениях.

В предыдущих разделах мы исследовали множество ситуаций, когда сборщик мусора работает неэффективно и требуется выполнить дополнительные настройки, обеспечивающие приемлемую эффективность. Еще одним примером ситуации, когда приложение может быть поставлено на колени, является использование сборщика мусора при нехватке памяти для размещения всех объектов.

Представьте приложение, выполняющееся на компьютере, имеющем 8 Гбайт физической памяти (ОЗУ). Такие компьютеры скоро должны исчезнуть, но саму ситуацию легко можно распространить на любой объем памяти, пока приложение способно адресовать ее (в 64-разрядных системах это очень большой объем). Приложение выделяет 12 Гбайт памяти, из которых 8 Гбайт будут находиться в физической памяти, а остальные 4 Гбайта будут вытеснены на диск, в файл подкачки. Диспетчер памяти Windows обеспечит сохранность страниц с наиболее часто используемыми объектами в физической памяти, а страницы с редко используемыми объектами вытеснит на диск.

В нормальном режиме работы приложение может вообще не вызывать операции обмена с файлом подкачки, потому что очень редко использует объекты, хранящиеся там. Однако, когда запускается цикл полной сборки мусора, сборщик мусора должен обойти все досягаемые объекты, чтобы построить граф объектов в фазе маркировки. Для обхода всех досягаемых объектов придется прочитать с диска 4 Гбайта информации. Так как физическая память заполнена, потребуется записать на диск 4 Гбайта данных, чтобы освободить память для извлекаемых с диска объектов. Наконец, после сборки мусора приложение попытается обратиться к часто используемым объектам, которые могли быть вытеснены на диск.

На момент написания этих строк типичные жесткие диски имели скорость обмена данными, порядка 150 Мбайт/сек (даже твердотельные высокоскоростные диски имеют скорость обмена всего в два раза выше). То есть, чтобы осуществить передачу 8 Гбайт данных потребуется примерно 55 секунд. В течение этого времени приложение будет ждать завершения сборки мусора (если не используется параллельный сборщик мусора). Добавление дополнительных процессоров (то есть, использование версии сборщика мусора для сервера) не поможет в этой ситуации, потому что узким местом является диск. Наличие в системе других выполняющихся приложений еще больше снизит производительность, потому что жесткому диску придется удовлетворять конкурирующие обращения к файлу подкачки.

Единственный способ решить проблему – размещать редко используемые объекты в неуправляемой памяти. Неуправляемая память не входит в юрисдикцию сборщика мусора, и обращения к ней будут выполняться, только когда это потребуется самому приложению.

Другой пример, имеющий отношение к управлению вытеснением в файл подкачки – закрепление страниц в физической памяти.

Приложения для Windows имеют документированную возможность обратиться к системе и запросить закрепление некоторой области в физической памяти (система игнорирует такие запросы только в исключительных ситуациях). Данный механизм нельзя использовать непосредственно, в сочетании со сборщиком мусора .NET, потому что управляемые приложения не могут непосредственно управлять выделением виртуальной памяти, выполняемым средой выполнения CLR. Однако, при использовании приема размещения CLR, приложение может закреплять страницы в памяти, в ходе удовлетворения запросов на выделение виртуальной памяти, посылаемых средой CLR.

Правила статического анализа кода (FxCop)

Инструмент статического анализа кода (FxCop) для Visual Studio имеет ряд правил, позволяющих устранить типичные проблемы производительности, связанные со сборкой мусора. Мы рекомендуем следовать этим правилам, потому что они помогают устранить множество ошибок на этапе кодирования. За дополнительной информацией об анализе управляемого кода с или без применения Visual Studio, обращайтесь к электронной документации на сайте MSDN.

Ниже перечислены основные правила, имеющие отношение к сборке мусора, которые распознаются в Visual Studio 11.

- CA1001 – типы, с полями, реализующими интерфейс `IDisposable`, также должны реализовать этот интерфейс. Это правило требует использования детерминированной финализации для типа, если члены, составляющие его используют детерминированную финализацию.
- CA1049 – типы, владеющие неуправляемыми ресурсами, должны реализовать интерфейс `IDisposable`. Это правило требует, чтобы типы, обеспечивающие доступ к неуправляемым ресурсам (такие как `System.Runtime.InteropServices.HandleRef`), реализовали шаблон Dispose.
- CA1063 – интерфейс `IDisposable` должен быть реализован правильно. Это правило требует правильной реализации шаблона Dispose.
- CA1821 – удаляйте пустые финализаторы. Это правило требует отсутствия в типах пустых объявлений методов-финализаторов, наличие которых отрицательно сказывается на производительности и повышает вероятность проявления эффекта «кризиса среднего возраста».

- CA2000 – удаляйте объекты перед выходом из области видимости. Это правило требует, чтобы все объекты, реализующие интерфейс `IDisposable`, в локальных переменных удалялись до выхода из области их видимости.
- CA2006 – используйте `SafeHandle` для инкапсуляции неуправляемых ресурсов. Это правило требует использовать, когда это возможно, класс `SafeHandle` или один из его дочерних классов вместо прямого обращения (например, через вызов `System.IntPtr`) к дескриптору неуправляемого ресурса.
- CA1816 – вызывайте `GC.SuppressFinalize` правильно. Это правило требует, чтобы типы, реализующие интерфейс `IDisposable`, внутри метода `Dispose` подавляли вызов метода-финализатора сборщиком мусора, и не подавляли в других методах.
- CA2213 – типы, с полями, реализующими интерфейс `IDisposable`, должны явно освобождать их. Это правило требует, чтобы типы, реализующие интерфейс `IDisposable`, вызывали метод `Dispose` своих полей, также реализующих интерфейс `IDisposable`.
- CA2215 – методы `Dispose` должны вызывать метод `Dispose` базового класса. это правило требует правильно реализовать шаблон Dispose, включая вызов метода `Dispose` родительского класса, если он тоже реализует интерфейс `IDisposable`.
- CA2216 – типы, реализующие интерфейс `IDisposable`, должны иметь метод-финализатор. Это правило требует наличия метода-финализатора в типе, реализующем интерфейс `IDisposable`, в качестве запасного варианта на случай, если класс пользователя пренебрегает возможностью детерминированной финализации объекта.

В заключение

На протяжении всей этой главы мы знакомились с моделью и реализацией сборщика мусора в .NET, ответственного за автоматическое освобождение неиспользуемой памяти. Мы исследовали альтернативы сборке мусора на основе трассировки, включая подсчет ссылок и список свободных блоков.

В основе сборщика мусора .NET лежат следующие концепции, исследованные нами в подробностях.

- Корни являются отправной точкой на пути конструирования графа всех досягаемых объектов.
- Маркировка – это стадия, в ходе которой сборщик мусора конструирует граф всех досягаемых объектов и маркирует их как используемые. Фаза маркировки может протекать параллельно с прикладными потоками выполнения.
- Чистка – это стадия, в ходе которой сборщик мусора перемещает досягаемые объекты и обновляет ссылки на них. Фаза чистки требует приостановки всех прикладных потоков выполнения.
- Закрепление – это механизм блокировки объекта в определенном месте так, чтобы сборщик мусора не смог переместить его. Используется совместно с неуправляемым кодом, требующим передачи ему указателя на управляемый объект, и может вызывать фрагментацию памяти.
- Возможность выбора разных версий сборщика мусора обеспечивает статическую настройку поведения сборщика мусора под особенности конкретного приложения.
- Модель поколений описывает ожидания, касающиеся продолжительности жизни объектов, основанные на его текущем возрасте. Согласно им, совсем юные объекты должны выйти из употребления очень быстро; старые объекты, согласно ожиданиям, будут жить дольше.
- Поколения – это концептуальные области памяти, пересекаемые объектами в течение их существования. Модель поколений упрощает частое выполнение частичной сборки мусора в поколении, где объекты, как предполагается, будут существовать недолго, и редко – полной сборки мусора, дорогостоящей и менее эффективной.
- Куча больших объектов – это область памяти, зарезервированная для больших объектов. Куча больших объектов может фрагментироваться, но объекты в ней не перемещаются, благодаря чему снижаются накладные расходы в фазе чистки.
- Сегменты – это области виртуальной памяти, выделенные средой выполнения CLR. Виртуальная память может фрагментироваться из-за того, что сегменты имеют фиксированный размер.
- Финализация – это запасной механизм для автоматического освобождения неуправляемых ресурсов недетерминированным (неявным) способом. Всегда, когда это возможно, вмес-

то автоматической следует использовать детерминированную (явную) финализацию, но предлагать клиентам обе альтернативы.

Самые сильные оптимизации в работе сборщика мусора, как правило, сопровождаются ловушками:

- Модель поколений дает определенные выгоды правильно спроектированным приложениям, но может проявлять эффект «кризиса среднего возраста», отрицательно сказывающийся на производительности.
- Необходимость в закреплении объекта возникает всякий раз, когда его необходимо передать по ссылке неуправляемому коду, но она может приводить к фрагментации динамической памяти, даже в младших поколениях.
- Сегменты обеспечивают выделение виртуальной памяти большими блоками, но могут страдать от фрагментации, вызванной внешними причинами.
- Автоматическая финализация дает удобный способ освобождения неуправляемых ресурсов, но сопровождается высокими накладными расходами и часто приводит к эффекту «кризиса среднего возраста», утечкам памяти и к состоянию гонки.

Ниже перечислены некоторые эффективные приемы, позволяющие максимально повысить производительность сборщика мусора:

- применяйте временные объекты так, чтобы они выходили из употребления как можно скорее, но сохраняйте старые объекты на протяжении всего срока выполнения приложения;
- закрепляйте большие массивы в памяти на этапе инициализации приложения и разбивайте их на маленькие буферы по мере необходимости;
- управляете распределением памяти с применением пулов объектов или распределяя неуправляемую память;
- реализуйте поддержку детерминированной финализации и используйте автоматическую финализацию, только как запасной вариант;
- экспериментируйте с выбором версии сборщика мусора в своих приложениях, чтобы определить, какая из них лучше отвечает конкретному программному и аппаратному окружению.

Далее перечислены некоторые инструменты, которые можно использовать для диагностики проблем, связанных с памятью, и для

исследования поведения приложения с точки зрения управления памятью:

- профилировщик CLR Profiler может использоваться для диагностики внутренней фрагментации, определения участков приложения, наиболее требовательных к памяти, выявления объектов, удаляемых в каждом цикле сборки мусора, и получения общего представления о размерах и возрасте освобождаемых объектов;
- библиотека SOS.DLL может использоваться для диагностики утечек памяти, анализа внутренней и внешней фрагментации, хронометража сборки мусора, получения списка объектов в управляемой динамической памяти, исследования очереди финализации и изучения состояния потоков выполнения сборщика мусора и финализации;
- счетчики производительности CLR можно использовать для получения общего представления о работе механизма сборки мусора, включая размер каждого поколения, скорость распределения памяти, информацию о финализации, количестве закрепленных объектов и многое другое;
- прием размещения CLR можно использовать для анализа распределения сегментов, определения частоты следования циклов сборки мусора, выявления потоков выполнения, вызывающих сборку мусора, и получения информации об операциях выделения неуправляемой памяти, инициируемых размещенной средой CLR.

Вооруженные теоретическими знаниями об устройстве механизма сборки мусора, всех связанных с ним механизмов, типичных ловушек, наиболее эффективных приемах повышения производительности, а также знакомством с диагностическими инструментами, вы готовы приступить к поискам путей оптимизации использования памяти в ваших приложениях с применением надлежащих стратегий управления.

ГЛАВА 5. Коллекции и обобщенные типы

Едва ли можно встретить программу, не использующую какую-нибудь коллекцию, такую как `List<T>` или `Dictionary<K,V>`. Крупные приложения могут одновременно использовать сотни тысяч экземпляров коллекций. Правильно выбранный или написанный вами тип коллекций, наиболее полно отвечающий вашим потребностям, может значительно повысить производительность многих приложений. Поскольку, начиная с версии .NET 2.0, коллекции весьма тесно связаны с реализацией обобщенных типов (generics) в CLR, начнем обсуждение с обобщенных типов.

Обобщенные типы

Достаточно часто возникает необходимость создать класс или метод, который одинаково хорошо может работать с данными любых типов. Полиморфизм и наследование способны помочь в этом, но до определенной границы; методы, которые должны быть абсолютно универсальными, принимают параметры типа `System.Object`, что влечет две основные проблемы обобщенного программирования в .NET до версии .NET 2.0.

- *Безопасность типов*: как на этапе компиляции проверить операции с данными обобщенных типов и явно запретить то, что может потерпеть неудачу во время выполнения?
- *Отсутствие упаковки*: как избежать упаковки типов значений, если метод принимает параметры, являющиеся ссылками типа `System.Object`?

Это весьма серьезные проблемы. Чтобы убедиться в этом, рассмотрим одну из простых коллекций в .NET 1.1 – `ArrayList`. Ниже приво-

дится упрощенная ее реализация, которая, впрочем, вполне отчетливо демонстрирует проблемы, обозначенные выше:

```
public class ArrayList : IEnumerable, ICollection, IList, ... {
    private object[] items;
    private int size;
    public ArrayList(int initialCapacity) {
        items = new object[initialCapacity];
    }
    public void Add(object item) {
        if (size < items.Length - 1) {
            items[size] = item;
            ++size;
        } else {
            // Создать массив большего размера, скопировать
            // элементы и затем добавить в него 'item'
        }
    }
    public object this[int index] {
        get {
            if (index < 0 || index >= size)
                throw IndexOutOfBoundsException(index);
            return items[index];
        }
        set {
            if (index < 0 || index >= size)
                throw IndexOutOfBoundsException(index);
            items[index] = value;
        }
    }
    // Остальные методы опущены для экономии места
}
```

Мы выделили в коде все вхождения объектов типа `System.Object`, являющегося «обобщенным» типом, на котором основана коллекция. Хотя такое решение выглядит вполне допустимым, фактическое его использование оказывается далеко от идеала:

```
ArrayList employees = new ArrayList(7);
employees.Add(new Employee("Kate"));
employees.Add(new Employee("Mike"));
Employee first = (Employee)employees[0];
```

Такое неуклюжее приведение к типу `Employee` необходимо лишь потому, что тип `ArrayList` не сохраняет информацию о типах своих элементов. Кроме того, он никак не ограничивает типы элементов, вставляемые в него, чтобы хоть как-то обеспечить общность их типов. Взгляните:

```
employees.Add(42);                          // Скомпилируется и будет работать!
Employee third = (Employee)employees[2]; // Скомпилируется, но возбудит
                                            // исключение во время выполнения...
```

Действительно, число 42 не принадлежит коллекции служащих (employees), но мы нигде не определяем, что коллекция типа `ArrayList` может хранить только экземпляры определенного типа. Теоретически вполне возможно определить тип `ArrayList`, накладывающий такое ограничение, но операции с такой коллекцией оказались бы слишком дорогостоящими, зато инструкции, такие как `employees.Add(42)`, было бы невозможно скомпилировать.

Это – проблема *безопасности типов*; «обобщенные» коллекции, опирающиеся на тип `System.Object`, не могут гарантировать безопасность типов на этапе компиляции и откладывают все проверки до этапа выполнения. Тогда, может быть, такой подход снимает проблемы *производительности*? Как оказывается, такая реализация имеет проблемы и с производительностью, когда в работе участвуют типы значений. Исследуем следующий код, где используется структура `Point2D` из главы 3 (простой тип значения с целочисленными координатами X и Y точки на плоскости):

```
ArrayList line = new ArrayList(1000000);
for (int i = 0; i < 1000000; ++i) {
    line.Add(new Point2D(i, i));
}
```

Каждый экземпляр типа `Point2D`, добавляемый в массив `ArrayList`, упаковывается, потому что метод `Add` принимает параметр ссылочного типа (`System.Object`). Этот фрагмент создаст 1 000 000 объектов `Point2D` и разместит их в динамической памяти. Как было показано в главе 3, в 32-разрядной системе 1 000 000 упакованных объектов `Point2D` займет 16 000 000 байт памяти (против 8 000 000 байт, которые займут простые значения типа `Point2D`). Кроме того, в массив `ArrayList` будет добавлен 1 000 000 ссылок, которые все вместе займут еще 4 000 000 байт – а всего будет занято 20 000 000 байт (см. рис. 5.1), хотя достаточно было бы лишь 8 000 000 байт. В действительности это та самая проблема, что вынудила нас отказаться от идеи создать ссылочный тип `Point2D`; `ArrayList` вынуждает нас использовать коллекцию, которая может работать только со ссылочными типами!

Можно ли как-то улучшить ситуацию? В действительности можно было бы реализовать свою коллекцию двумерных точек, как показано ниже (хотя, при этом придется также предусмотреть специализированную реализацию интерфейсов `IEnumerable`, `ICollection` и `IList`

для точек...). Она полностью идентична «обобщенной» реализации `ArrayList`, но принимает значение типа `Point2D` везде, где раньше принимался `object`:

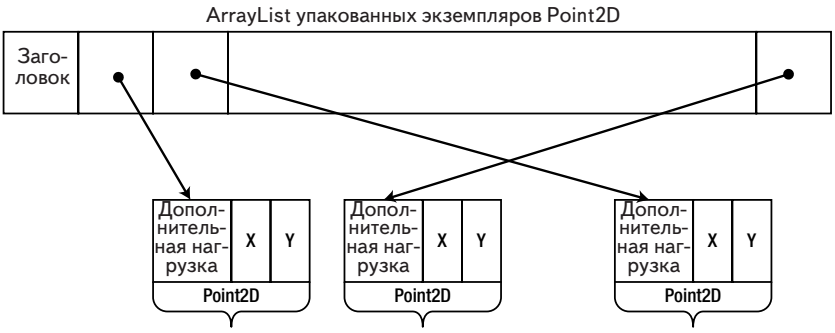

Рис. 5.1. Тип ArrayList, содержащий упакованные объекты Point2D, хранит ссылки, занимающие дополнительную память, что приводит к принудительному созданию упакованных объектов Point2D в динамической памяти и дополнительным накладным расходам.

```
public class Point2DArrayList : IPoint2DEnumerable,
IPoint2DCollection, IPoint2DList, ... {
    private Point2D[] items;
    private int size;
    public ArrayList(int initialCapacity) {
        items = new Point2D[initialCapacity];
    }
    public void Add(Point2D item) {
        if (size < items.Length - 1) {
            items[size] = item;
            ++size;
        } else {
            // Создать массив большего размера, скопировать
            // элементы и затем добавить в него 'item'
        }
    }
    public Point2D this[int index] {
        get {
            if (index < 0 || index >= size)
                throw IndexOutOfBoundsException(index);
            return items[index];
        }
        set {
            if (index < 0 || index >= size)
                throw IndexOutOfBoundsException(index);
            items[index] = value;
        }
```

```
    }
    // Остальные методы опущены для экономии места
}
```

Реализовав похожую коллекцию объектов `Employee` можно было бы решить проблему безопасности типов, описанную выше. К сожалению, создавать специализированные коллекции для каждого типа данных совершенно непрактично. Определенно, данную функцию должен выполнять компилятор языка, как это сделано в .NET 2.0 – чтобы обеспечить поддержку обобщенных типов в классах и методах, и одновременно гарантировать безопасность типов и избавиться от принудительной упаковки.

Обобщенные типы в .NET

Обобщенные классы и методы позволяют писать *по-настоящему* обобщенный код, без применения `System.Object` с одной стороны и без специализации для каждого типа данных – с другой. Ниже приводится пример реализации обобщенного типа `List<T>`, взамен предыдущей нашей реализации `ArrayList`, которая решает обе проблемы, безопасности типов и принудительной упаковки:

```
public class List<T> : IEnumerable<T>, ICollection<T>, IList<T>, ... {
    private T[] items;
    private int size;
    public List(int initialCapacity) {
        items = new T[initialCapacity];
    }
    public void Add(T item) {
        if (size < items.Length - 1) {
            items[size] = item;
            ++size;
        } else {
            // Создать массив большего размера, скопировать
            // элементы и затем добавить в него 'item'
        }
    }
    public T this[int index] {
        get {
            if (index < 0 || index >= size)
                throw IndexOutOfBoundsException(index);
            return items[index];
        }
        set {
            if (index < 0 || index >= size)
                throw IndexOutOfBoundsException(index);
            items[index] = value;
        }
```

```
    }
    // Остальные методы опущены для экономии места
}
```

Примечание. *Тем, кто не знаком с синтаксисом обобщенных типов в языке C#, рекомендуем обратиться к замечательной книге Джона Скита (Jon Skeet) «C# in Depth» (Manning, 2010). В оставшейся части главы мы будем полагать, что вы уже писали обобщенные классы или, хотя бы, пользовались такими классами.*

Если прежде вам приходилось писать обобщенные классы или методы, вы наверняка знаете, насколько просто преобразовать псевдообобщенный код на основе `System.Object` в действительно обобщенный код, задействовав *параметры обобщенного типа* (generic type parameters). Не менее просто использовать обобщенные классы и методы, просто подставляя *аргументы обобщенного типа* (generic type arguments), где это необходимо:

```
List<Employee> employees = new List<Employee>(7);
employees.Add(new Employee("Kate"));
Employee first = employees[0];   // Приведение типа не требуется
employees.Add(42);               // Не компилируется!

List<Point2D> line = new List<Point2D>(1000000);
for (int i = 0; i < 1000000; ++i) {
    line.Add(new Point2D(i, i)); // Упаковка не производится,
}                                // Сохраняется по значению
```

Как по волшебству, обобщенная коллекция обеспечивает безопасность типа (она не позволяет сохранять в ней элементы других типов) и не требует упаковки типов значений. Даже внутреннее хранилище – массив элементов – приспосабливается в соответствии с аргументом обобщенного типа: когда `T` соответствует типу `Point2D`, массив элементов получает тип `Point2D[]` и хранит значения, а не ссылки. Мы еще вернемся к этому волшебству ниже, а пока будем наслаждаться эффективным решением на уровне языка проблем обобщенного программирования.

Однако, похоже, что этого решения недостаточно, когда необходимо, чтобы обобщенные параметры обладали некоторыми дополнительными особенностями. Взгляните на метод, выполняющий поиск методом дихотомии в отсортированном массиве. Полностью обобщенная версия не в состоянии выполнить такой поиск, потому что `System.Object` не обладает механизмом сравнения:

```
public static int BinarySearch<T>(T[] array, T element) {
    // В некоторый момент потребуется выполнить сравнение:
```

```
    if (array[x] < array[y]) {
        ...
    }
}
```

Класс `System.Object` не поддерживает статический оператор `<`, из-за чего попытка скомпилировать метод потерпит неудачу! В действительности мы должны убедить компилятор, что для любого аргумента обобщенного типа, передаваемого в метод, он сможет отыскать все необходимые реализации используемых им методов (включая операторы). Теперь настало время поговорить об ограничениях обобщенных типов.

Ограничения обобщенных типов

Ограничения обобщенных типов указывают компилятору, что лишь *некоторые* фактические типы могут использоваться в качестве аргумента при использовании обобщенного типа. Всего существует пять типов ограничений:

```
// T должен реализовать интерфейс:
public int Format(T instance) where T : IFormattable {
    return instance.ToString("N", CultureInfo.CurrentUICulture);
    // OK, T должен иметь метод
    // IFormattable.ToString(string, IFormatProvider)
}

// T должен наследовать базовый класс:
public void Display<T>(T widget) where T : Widget {
    widget.Display(0, 0);
    // OK, T должен наследовать класс Widget,
    // имеющий метод Display(int, int)
}

// T должен иметь конструктор без параметров:
public T Create<T>() where T : new() {
    return new T();
    // OK, T имеет конструктор без параметров
    // Компилятор C# скомпилирует 'new T()' в не самый
    // оптимальный вызов Activator.CreateInstance<T>(), но для
    // этого ограничения отсутствует эквивалент на языке IL
}

// T должен быть ссылочным типом:
public void ReferencesOnly<T>(T reference) where T : class

// T должен быть типом значения:
public void ValuesOnly<T>(T value) where T : struct
```

Для примера реализации поиска методом дихотомии, наиболее полезным будет ограничение, требующее реализации интерфейса (и действительно, это наиболее часто используемый тип ограничений). В частности, мы можем потребовать от `T` реализовать интерфейс `IComparable` и сравнивать элементы массива с помощью метода `IComparable.CompareTo`. Однако `IComparable` не является обобщенным интерфейсом, а его метод `CompareTo` принимает параметр типа `System.Object`, что приводит к упаковке типов значений. Очевидно, что должен существовать обобщенная версия интерфейса `IComparable` – интерфейс `IComparable<T>`, который с успехом можно было бы использовать в нашем примере:

```
// Из .NET Framework:
public interface IComparable<T> {
    int CompareTo(T other);
}

public static int BinarySearch<T>(T[] array, T element) where T
: IComparable<T> {
    // В некоторый момент потребуется выполнить сравнение:
    if (array[x].CompareTo(array[y]) < 0) {
        ...
    }
}
```

Эта версия реализации поиска методом дихотомии не вызывает упаковку при сравнении экземпляров типов значений, работает с любыми типами, реализующими интерфейс `IComparable<T>` (включая все встроенные простые типы, строки и многие другие), и обеспечивает безопасность типов, не требуя определять наличие возможности сравнения во время выполнения.

Ограничение интерфейса и IEquatable<T>

В главе 3 было показано, насколько важно для производительности переопределить метод `Equals` в типах значений и реализовать интерфейс `IEquatable<T>`. Почему этот интерфейс так важен? Взгляните на следующий фрагмент:

```
public static void CallEquals<T>(T instance) {
    instance.Equals(instance);
}
```

Вызов `Equals` внутри метода в транслируется в вызов виртуального метода `Object.Equals`, который принимает параметр `System.Object`, что вызывает упаковку типов значений. Это – единственная альтернатива, которая с точки зрения компилятора C# гарантированно будет доступна

для любых типов T. Если мы хотим убедить компилятор, что T имеет метод Equals, принимающий параметр данного типа T, следует явно определить ограничение:

```
// Из the .NET Framework:
public interface IEquatable<T> {
    bool Equals(T other);
}

public static void CallEquals<T>(T instance) where T : IEquatable<T> {
    instance.Equals(instance);
}
```

Наконец, нам может потребоваться дать возможность вызывающему коду использовать любой тип T и задействовать реализацию IEquatable<T>, если тип T предоставляет ее, чтобы избежать упаковки и обеспечить более высокую эффективность. В этом отношении довольно интересным выглядит подход, применяемый в реализации List<T>. Если бы класс List<T> требовал наличия ограничения IEquatable<T> для своего параметра обобщенного типа, его нельзя было бы использовать для хранения типов, не реализующих этот интерфейс. Поэтому List<T> не имеет ограничения IEquatable<T>. Для поддержки метода Contains (и других, проверяющих объекты на равенство) List<T> опирается на *механизм проверки равенства объектов* (equality comparer) – конкретную реализацию абстрактного класса EqualityComparer<T> (который, по стечению обстоятельств, реализует интерфейс EqualityComparer<T>, используемый непосредственно некоторыми коллекциями, включая HashSet<T> и Dictionary<K,V>).

Когда методу List<T>.Contains требуется вызвать Equals для сравнения двух элементов коллекции, он использует статическое свойство EqualityComparer<T>.Default, откуда извлекает реализацию механизма проверки равенства объектов, пригодную для сравнения экземпляров типа T, и вызывает его виртуальный метод Equals(T, T). Заполнение этого поля выполняет приватный статический метод EqualityComparer<T>.CreateComparer, который создает соответствующую реализацию при первом обращении и затем сохраняет ее для последующих вызовов. Когда CreateComparer обнаруживает, что T реализует IEquatable<T>, он возвращает экземпляр GenericEqualityComparer<T>, имеющий ограничение IEquatable<T>, и вызывает Equals через интерфейс. В противном случае CreateComparer обращается к классу ObjectEqualityComparer<T>, не имеющему ограничений для типа T и вызывает виртуальный метод Equals класса Object.

Этот трюк, используемый классом List<T> для проверки равенства, может пригодиться и в других ситуациях. Когда ограничение определено, обобщенный класс или метод может использовать потенциально более эффективную реализацию, без проверки типа во время выполнения.

Совет. *Как было показано выше, для математических операторов, таких как сложение и вычитание, не существует никаких ограничений обобщенных типов. Это означает, что нельзя написать обобщенный метод,*

использующий выражения, такие как `a+b`*, с обобщенными параметрами. Стандартное решение, обеспечивающее возможность реализации обобщенных числовых алгоритмов заключается в использовании вспомогательной структуры, реализующей интерфейс* `IMath<T>` *с необходимыми арифметическими операциями, внутри обобщенного метода. За дополнительной информацией обращайтесь к статье Рудигера Кляйна (Rüdiger Klaehn) «Using generics for calculations» на сайте CodeProject: http://www.codeproject.com/Articles/8531/Using-generics-for-calculations.*

Исследовав значительную часть синтаксиса поддержки обобщенных типов в языке C#, мы можем перейти к их фактической реализации времени выполнения. Но, прежде чем углубиться в изучение этого вопроса, необходимо ответить на вопрос, – а существует ли вообще некое представление обобщенных типов во время выполнения? Как будет показано чуть ниже, шаблоны C++, похожий механизм, не имеют такого представления. На этот вопрос легко можно ответить, заглянув в реализацию Reflection API, выполняющего операции с обобщенными типами *во время выполнения*:

```
Type openList = typeof(List<>);
Type listOfInt = openList.MakeGenericType(typeof(int));
IEnumerable<int> ints = (IEnumerable<int>)Activator.CreateInstance
(listOfInt);

Dictionary<string, int> frequencies = new Dictionary<string, int>();
Type openDictionary = frequencies.GetType().GetGenericTypeDefinition();
Type dictStringToDouble = openDictionary.MakeGenericType(typeof(string),
typeof(double));
```

Как здесь показано, мы можем динамически создавать обобщенные типы из существующих обобщенных типов и параметризовать «открытый» обобщенный тип для создания экземпляра «закрытого» обобщенного типа. Данный пример демонстрирует, что обобщенные типы являются обычными объектами и имеют представление времени выполнения, с которым мы сейчас и познакомимся.

Реализация обобщенных типов в CLR

Синтаксически обобщенные типы CLR очень похожи на обобщенные типы Java и даже имеют некоторое сходство с шаблонами C++. Однако, как оказывается, их внутренняя реализация и ограничения, накладываемые на программы, использующие их, существенно отличаются от Java и C++. Чтобы понять эти отличия, необходимо получить некоторое представление об обобщенных типах Java и шаблонах C++.

Обобщенные типы Java

Обобщенный класс в языке Java может иметь параметр типа и существует даже механизм ограничений, очень похожий на тот, что предлагается фреймворком .NET (связанные параметры типов и групповые символы). Например, ниже демонстрируется первая попытка реализовать наш тип `List<T>` на языке Java:

```
public class List<E> {
    private E[] items;
    private int size;
    public List(int initialCapacity) {
        items = new E[initialCapacity];
    }
    public void Add(E item) {
        if (size < items.Length - 1) {
            items[size] = item;
            ++size;
        } else {
            // Создать массив большего размера, скопировать
            // элементы и затем добавить в него 'item'
        }
    }
    public E getAt(int index) {
        if (index < 0 || index >= size)
            throw IndexOutOfBoundsException(index);
        return items[index];
    }
    // Остальные методы опущены для экономии места
}
```

К сожалению, этот код не будет компилироваться, так как выражение `new E[initialCapacity]` в Java является недопустимым. Причина заключается в способе компиляции обобщенного кода на Java. Компилятор Java удаляет все упоминания о параметре типа и замещает их типом `java.lang.Object`. Этот процесс называется *затиранием типов* (type erasure). Как результат, во время выполнения существует только один тип – `List`, *обычный* (raw) тип – а вся информация об аргументе обобщенного типа исчезает. (Справедливости ради следует отметить, используя прием затирания типов, Java обеспечивает двоичную совместимость с библиотеками и приложениями, созданными до появления поддержки обобщенных типов, чего не предлагает .NET 2.0 в отношении кода, скомпилированного для .NET 1.1.)

Но не все так плохо. Используя массив `Object`, мы можем успокоить компилятор и получить обобщенный класс, компилирующийся и обеспечивающий безопасность типов:

```
public class List<E> {
    private Object[] items;
    private int size;
    public void List(int initialCapacity) {
        items = new Object[initialCapacity];
    }
    // Остальная часть кода осталась без изменений
}

List<Employee> employees = new List<Employee>(7);
employees.Add(new Employee("Kate"));
employees.Add(42);  // Не компилируется!
```

Однако применение такого подхода в CLR вызывает беспокойство: что будет происходить с типами значений? Одной из двух причин введения поддержки обобщенных типов было стремление избежать упаковки. Вставка типа значения в массив объектов потребует упаковки, а это для нас неприемлемо.

Шаблоны C++

В сравнении с обобщенными типами в Java, шаблоны в языке C++ выглядят намного привлекательнее. (Они обладают чрезвычайно широкими возможностями: возможно вам приходилось слышать, что механизм разрешения шаблонов сам по себе являются *полными по Тьюрингу* (Turing-complete).) Компилятор C++ не выполняет затирание типов – как раз наоборот – и не требует определения каких-либо ограничений, потому что компилятор благополучно компилирует все, что встретится на его пути. Рассмотрим для начала пример со списком, а затем разберемся, что же происходит с ограничениями:

```
template <typename T>
class list {
private:
    T* items;
    int size;
    int capacity;
public:
    list(int initialCapacity) : size(0), capacity(initialCapacity) {
        items = new T[initialCapacity];
    }
    void add(const T& item) {
        if (size < capacity) {
            items[size] = item;
            ++size;
        } else {
            // Создать массив большего размера, скопировать
            // элементы и затем добавить в него 'item'
```

```
        }
    }
    const T& operator[](int index) const {
        if (index < 0 || index >= size) throw exception("Index out of bounds");
        return items[index];
    }
    // Остальные методы опущены для экономии места
};
```

Шаблон класса `list` совершенно безопасен для типов: *для каждого* случая применения шаблона создается новый класс, использующий определение шаблона как... шаблон! Хотя все это происходит за кулисами, тем не менее, ниже показано, как мог бы выглядеть полученный код в действительности:

```
// Оригинальный код на C++:
list<int> listOfInts(14);

// Сгенерированный компилятором:
class __list__int {
private:
    int* items;
    int size;
    int capacity;
public:
    __list__int(int initialCapacity) : size(0), capacity(initialCapacity) {
        items = new int[initialCapacity];
    }
};
__list__int listOfInts(14);
```

Обратите внимание, что методы `add` и `operator[]` не были развернуты – вызываемый код не использует их, а компилятор генерирует только те части определения шаблона, которые используются конкретным экземпляром. Отметьте также, что компилятор *ничего* не генерирует из определения шаблона – он ждет определенного момента его использования, прежде чем произвести какой-либо код.

Это объясняет отсутствие необходимости ограничений для шаблонов C++. Вернемся к нашему примеру поиска методом дихотомии – следующая его реализация выглядит вполне удачной:

```
template <typename T>
int BinarySearch(T* array, int size, const T& element) {
    // В некоторый момент потребуется выполнить сравнение:
    if (array[x] < array[y]) {
        ...
    }
}
```

Здесь нет необходимости убеждать компилятор C++ в чем бы то ни было. В конце концов, определение шаблона не имеет большого значения; компилятор терпеливо ждет конкретного его использования:

```
int numbers[10];
BinarySearch(numbers, 10, 42); // Скомпилирует, тип int
                               // поддерживает оператор <
class empty {};
empty empties[10];
BinarySearch(empties, 10, empty()); // Не скомпилирует, тип empty
                                    // не поддерживает оператор <
```

Хотя такой способ определения обобщенных типов и выглядит привлекательным, шаблоны C++ имеют и отрицательные черты и ограничения, нежелательные для обобщенных типов CLR.

- Так как развертывание шаблона выполняется на этапе компиляции, нет никакого способа обеспечить возможность совместного использования его экземпляров разными скомпилированными модулями. Например, две библиотеки DLL, загруженные в один и тот же процесс, могут использовать разные скомпилированные версии `list<int>`. Компиляция шаблонов выполняется долго и требует значительных объемов памяти.
- По той же причине, почему экземпляры шаблона в разных скомпилированных модулях считаются несовместимыми, отсутствует простой механизм экспортирования экземпляров шаблона из библиотек DLL (например, очень сложно экспортировать функцию, возвращающую `list<int>`).
- Нет никакой возможности скомпилировать библиотеку так, чтобы она содержала определение шаблона. Определение шаблона существует только на уровне исходного кода, как заголовочный фай, который можно подключить к файлу с исходным кодом на C++.

Внутреннее устройство обобщенных типов

После знакомства с особенностями обобщенных типов в Java и шаблонов в C++, вам проще будет понять особенности реализации обобщенных типов в CLR, описываемой далее. Обобщенные типы – даже открытые, такие как `List<>` – являются обычными объектами времени выполнения типа `System.Type`, и имеют таблицу методов и структуру EEClass (см. главу 3). Обобщенные типы можно экспортировать из сборок (assemblies) и только одно определение обобщен-

ного типа существует на этапе компиляции. Обобщенные типы не разворачиваются во время компиляции, но, как было показано выше, компилятор убеждается, что все операции с параметром обобщенного типа совместимы с указанными ограничениями.

Когда среде выполнения CLR требуется создать экземпляр закрытого обобщенного типа, такого как `List<int>`, он создает таблицу методов и структуру EEClass, опираясь на открытый тип. Как обычно, вновь созданная таблица методов содержит указатели на методы, которые компилируются «на лету» JIT-компилятором. Однако, при этом выполняется одна важная оптимизация: скомпилированные тела методов закрытых обобщенных типов, которые ссылаются на параметр типа, могут использовать совместно. Чтобы было понятнее, рассмотрим метод `List<T>.Add` и попробуем скомпилировать его в инструкции процессора x86, когда `T` является ссылочным типом:

```
// Код на C#:
public void Add(T item) {
    if (size < items.Length - 1) {
        items[size] = item;
        ++size;
    } else {
        AllocateAndAddSlow(item);
    }
}

; код на языке ассемблера для x86, когда T является ссылочным типом
; Предполагается, что ECX содержит ссылку 'this', а EDX - ссылку
; на элемент, пролог и эпилог опущены
mov eax, dword ptr [ecx+4]          ; items
mov eax, dword ptr [eax+4]          ; items.Length
dec eax
cmp dword ptr [ecx+8], eax          ; size < items.Length - 1
jge AllocateAndAddSlow
mov eax, dword ptr [ecx+4]
mov ebx, dword ptr [ecx+8]
mov dword ptr [eax+4*ebx+4], edx    ; items[size] = item
inc dword ptr [eax+8]               ; ++size
```

Очевидно, что реализация метода в машинном коде никак не зависит от типа `T` и будет работать с любым ссылочным типом. Это обстоятельство позволяет JIT-компилятору экономить ресурсы (время и память) и совместно использовать указатель из таблицы методов для `List<T>.Add` во всех таблицах методов, где `T` является ссылочным типом.

Примечание. *Эта идея имеет продолжение, которое, впрочем, мы не будем исследовать. Например, если тело метода содержит выражение* `new T[10]`*, это могло бы потребовать создания отдельной реализации метода для каждого типа* `T` *или, по крайней мере, способа получения* `T` *во время выполнения (например, через дополнительный скрытый параметр, передаваемый методу). Кроме того, мы не коснулись особенностей влияния ограничений на генерируемый код, но у вас уже не должно быть сомнений, что вызов методов интерфейса или виртуальных методов базовых классов будет выполняться одинаково, независимо от типа.*

Все сказанное выше не относится к типам значений. Например, если в параметре `T` указать тип `long`, компилятор сгенерирует иные машинные инструкции для выражения `items[size] = item`, потому что вместо 4 необходимо скопировать 8 байт. Более крупные типы значений могут потребовать сгенерировать более одной инструкции; и так далее.

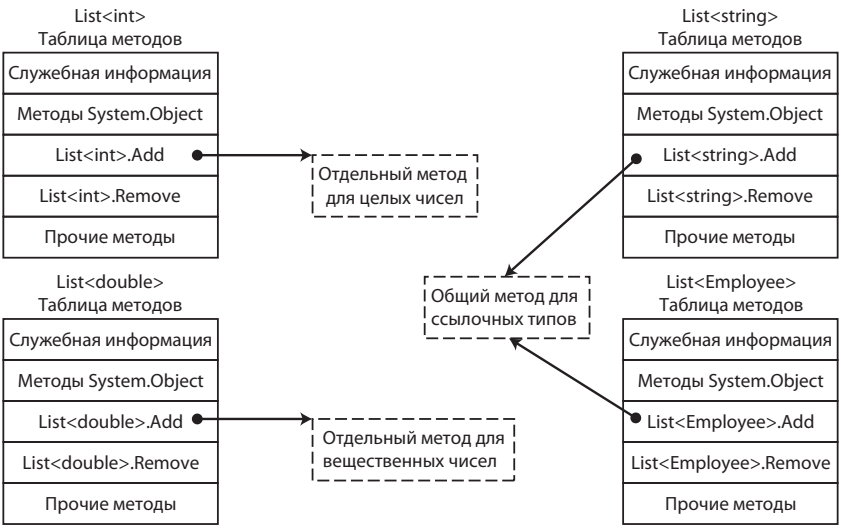

Рис. 5.2. Все ссылочные реализации List<T> совместно используют один общий метод Add, тогда как реализации типов значений имеют разные версии кода.

Для демонстрации на рис. 5.2 изображены таблицы методов закрытых обобщенных типов, полученные нами с помощью расширения SOS.DLL, которые являются реализациями одного и того же открытого обобщенного типа. Например, рассмотрим класс `BasicStack<T>` с двумя методами, `Push` и `Pop`:

```
class BasicStack<T> {
    private T[] items;
    private int topIndex;

    public BasicStack(int capacity = 42) {
        items = new T[capacity];
    }
    public void Push(T item) {
        items[topIndex++] = item;
    }
    public T Pop() {
        return items[--topIndex];
    }
}
```

Таблицы методов для `BasicStack<string>`, `BasicStack<int[]>`, `BasicStack<int>` и `BasicStack<double>` приводятся ниже. Обратите внимание, что для закрытых обобщенных типов, если в аргументе обобщенного типа был указан ссылочный тип, элементы таблиц методов (то есть адреса) совпадают, а для типов значений – нет:

```
0:004> !dumpheap -stat
...
00173b40        1     16 BasicStack`1[[System.Double, mscorlib]]
00173a98        1     16 BasicStack`1[[System.Int32, mscorlib]]
00173a04        1     16 BasicStack`1[[System.Int32[], mscorlib]]
001739b0        1     16 BasicStack`1[[System.String, mscorlib]]
...
0:004> !dumpmt -md 001739b0
EEClass:         001714e0
Module:          00172e7c
Name:            BasicStack`1[[System.String, mscorlib]]
...
MethodDesc Table
   Entry MethodDe  JIT Name
...
00260360 00173924  JIT BasicStack`1[[System.__Canon, mscorlib]].
Push(System.__Canon)
00260390 0017392c  JIT BasicStack`1[[System.__Canon, mscorlib]].
Pop()

0:004> !dumpmt -md 00173a04
EEClass:         001714e0
Module:          00172e7c
Name:            BasicStack`1[[System.Int32[], mscorlib]]
...
MethodDesc Table
   Entry MethodDe  JIT Name
...
00260360 00173924  JIT BasicStack`1[[System.__Canon, mscorlib]].
```

```
Push(System.__Canon)
00260390 0017392c  JIT BasicStack`1[[System.__Canon, mscorlib]].
Pop()

0:004> !dumpmt -md 00173a98
EEClass:         0017158c
Module:          00172e7c
Name:            BasicStack`1[[System.Int32, mscorlib]]
...
MethodDesc Table
   Entry MethodDe  JIT Name
...
002603c0 00173a7c  JIT BasicStack`1[[System.Int32, mscorlib]].
Push(Int32)
002603f0 00173a84  JIT BasicStack`1[[System.Int32, mscorlib]].
Pop()

0:004> !dumpmt -md 00173b40
EEClass:         001715ec
Module:          00172e7c
Name:            BasicStack`1[[System.Double, mscorlib]]
...
MethodDesc Table
   Entry MethodDe  JIT Name
...
00260420 00173b24  JIT BasicStack`1[[System.Double, mscorlib]].
Push(Double)
00260458 00173b2c  JIT BasicStack`1[[System.Double, mscorlib]].
Pop()
```

Наконец, если взглянуть на фактические реализации методов, станет очевидно, что версии ссылочных типов вообще не зависят от фактических типов (все, что они делают, – перемещают ссылки с места на место), а реализации методов типов значений *тесно* связаны с фактическим типом. В конце концов, копирование целого числа отличается от копирования вещественного числа двойной точности. Ниже приводятся дизассемблированные версии метода `Push`, в которых выделены строки, фактически перемещающие данные:

```
0:004> !u 00260360
Normal JIT generated code
BasicStack`1[[System.__Canon, mscorlib]].Push(System.__Canon)
00260360 57              push    edi
00260361 56              push    esi
00260362 8b7104          mov     esi,dword ptr [ecx+4]
00260365 8b7908          mov     edi,dword ptr [ecx+8]
00260368 8d4701          lea     eax,[edi+1]
0026036b 894108          mov     dword ptr [ecx+8],eax
0026036e 52              push    edx
```

```
0026036f 8bce            mov     ecx,esi
00260371 8bd7            mov     edx,edi
00260373 e8f4cb3870      call    clr!JIT_Stelem_Ref (705ecf6c)
00260378 5e              pop     esi
00260379 5f              pop     edi
0026037a c3              ret

0:004> !u 002603c0
Normal JIT generated code
BasicStack`1[[System.Int32, mscorlib]].Push(Int32)
002603c0 57              push    edi
002603c1 56              push    esi
002603c2 8b7104          mov     esi,dword ptr [ecx+4]
002603c5 8b7908          mov     edi,dword ptr [ecx+8]
002603c8 8d4701          lea     eax,[edi+1]
002603cb 894108          mov     dword ptr [ecx+8],eax
002603ce 3b7e04          cmp     edi,dword ptr [esi+4]
002603d1 7307            jae     002603da
002603d3 8954be08        mov     word ptr [esi+edi*4+8],edx
002603d7 5e              pop     esi
002603d8 5f              pop     edi
002603d9 c3              ret
002603da e877446170      call    clr!JIT_RngChkFail (70874856)
002603df cc              int 3

0:004> !u 00260420
Normal JIT generated code
BasicStack`1[[System.Double, mscorlib]].Push(Double)
00260420 56              push    esi
00260421 8b5104          mov     edx,dword ptr [ecx+4]
00260424 8b7108          mov     esi,dword ptr [ecx+8]
00260427 8d4601          lea     eax,[esi+1]
0026042a 894108          mov     dword ptr [ecx+8],eax
0026042d 3b7204          cmp     esi,dword ptr [edx+4]
00260430 730c            jae     0026043e
00260432 dd442408        fld     qword ptr [esp+8]
00260436 dd5cf208        fstp    qword ptr [edx+esi*8+8]
0026043a 5e              pop     esi
0026043b c20800          ret     8
0026043e e813446170      call    clr!JIT_RngChkFail (70874856)
00260443 cc              int     3
```

Выше уже говорилось, что реализация обобщенных типов в .NET обеспечивает полную безопасность типов на этапе компиляции. Остается только убедиться, что при использовании типов значений с коллекциями обобщенных типов не происходит их упаковка. В действительности, поскольку JIT-компилятор генерирует отдельные реализации методов для каждого закрытого обобщенного типа, если в ар-

гументе обобщенного типа был указан тип значения, необходимость в упаковке отпадает.

В заключение заметим, что обобщенные типы в .NET обладают существенными преимуществами, по сравнению с обобщенными типами в Java или шаблонами в C++. Механизм ограничений обобщенных типов несколько ограничен, в сравнении с Диким Западом языка C++, но гибкая возможность экспортирования обобщенных типов из сборок и преимущества производительности, которые дает генерация кода по требованию затмевают все недостатки.

Коллекции

В состав .NET Framework входит большое число коллекций, но мы не ставили целью рассмотреть каждую из них в этой главе – для этого лучше обратиться к электронной документации на сайте MSDN. Однако, коллекции обладают некоторыми непростыми особенностями, которые следует учитывать при выборе для использования, особенно в коде, где требуется высокая производительность. Именно исследованием этих особенностей мы и займемся в данном разделе.

> **Примечание.** *Некоторые разработчики опасаются использовать какие-либо классы коллекций и предпочитают встроенные массивы. Массивы очень неудобны: они не гибки, не позволяют динамически изменять их размер и на их основе очень сложно эффективно реализовать некоторые операции, но они, как известно, отличаются от других типов коллекций минимальными накладными расходами. Не нужно бояться использовать встроенные коллекции, пока в вашем распоряжении имеются отличные инструменты измерения производительности, как те, что рассматривались в главе 2. Внутренние особенности реализации коллекций .NET, обсуждаемые в этом разделе, также помогут вам сделать удачный выбор. Один простой пример: обход коллекции* `List<T>` *в цикле* `foreach` *выполняется лишь немногим дольше, чем в цикле* `for`*, потому что цикл* `foreach` *проверяет – не изменилась ли коллекция между итерациями.*

Для начала вспомним, какие классы коллекций входят в состав .NET 4.5 – исключая параллельные коллекции, которые будут обсуждаться отдельно – и их характеристики производительности. Сравнение производительности операций вставки, удаления и поиска – отличный способ подобрать кандидата, лучше всего подходящего для ваших потребностей. В табл. 5.1 перечислены только обобщенные коллекции (необобщенные версии были удалены в версии .NET 2.0):

Таблица 5.1. Коллекции в .NET Framework

Коллекция	Описание	Время вставки	Время удаления	Время поиска	Сортировка	Доступ по индексу
List<T>	Массив с автоматическим изменением размера	Амортизированная $O(1)^*$	$O(n)$	$O(n)$	Нет	Да
LinkedList<T>	Двусвязный список	$O(1)$	$O(1)$	$O(n)$	Нет	Нет
Dictionary<K,V>	Хеш-таблица	$O(1)$	$O(1)$	$O(1)$	Нет	Нет
HashSet <T>	Хеш-таблица	$O(1)$	$O(1)$	$O(1)$	Нет	Нет
Queue<T>	Циклический массив с автоматическим изменением размера	Амортизированная $O(1)$	$O(1)$	–		Нет
Stack<T>	Красно-черное дерево	Амортизированная $O(1)$	$O(1)$	–	Нет	Нет
SortedDictionary<K,V>	Красно-черное дерево	$O(\log n)$	$O(\log n)$	$O(\log n)$	Да (по ключам)	Нет
SortedList<K,V>	Сортированный массив с автоматическим изменением размера	$O(n)^{**}$	$O(n)$	$O(\log n)$	Да (по ключам)	Да
SortedSet<T>	Красно-черное дерево	$O(\log n)$	$O(\log n)$	$O(\log n)$	Да (по ключам)	Нет

Примечание.

* Под «амортизированной» производительностью в данном случае подразумевается, что некоторые операции могут показывать производительность $O(n)$, но большинство операций будут показывать производительность $O(1)$, поэтому средняя производительность по n операциям составит $O(1)$.

** Если данные вставляются в порядке сортировки, тогда производительность составит $O(1)$.

Существует несколько обстоятельств и особенностей реализации, которые необходимо учитывать при выборе коллекции из числа, включенных в .NET Framework.

- Требования, предъявляемые к разным коллекциям, существенно разнятся. Далее в этой главе будет показано, как внутренняя

организация коллекций влияет на работу кеша процессора с коллекциями `List<T>` и `LinkedList<T>`. Другой пример – классы `SortedSet<T>` и `List<T>`; первый реализует поиск по двоичному дереву с *n* узлами для *n* элементов, а второй – в непрерывном массиве из *n* элементов. В 32-разрядных системах для хранения *n* экземпляров типа значения размером *s* в сортированном множестве требуется $(20 + s)n$ байт, тогда как при использовании списка требуется всего sn байт.

- Некоторые коллекции предъявляют дополнительные требования к элементам для достижения удовлетворительной производительности. Например, как было показано в главе 3, любая реализация хеш-таблицы требует удачной реализации вычисления хеш-кода для элементов.
- Амортизированная производительность $O(1)$ коллекций, хорошо организованных, едва отличается от истинной производительности $O(1)$. В конце концов, немногие программисты (и программы!) опасаются, что в течение продолжительного времени использования `List<T>.Add` иногда может вызывать дополнительные задержки из-за выделения памяти, прямо пропорциональные числу элементов в списке. Анализ амортизированной производительности – отличный способ определения оптимальных границ для различных алгоритмов и коллекций.
- Извечный компромисс «пространство-время» уже заложен в архитектуру коллекций и, разумеется, влияет на выбор той или иной коллекции из числа, включенных в .NET Framework. Коллекция `SortedList<K,V>` предлагает очень компактный способ хранения элементов за счет линейного увеличения времени вставки и удаления, тогда как коллекция `SortedDictionary<K,V>` занимает больше места, но обеспечивает логарифмическое увеличение времени выполнения всех операций.

Примечание. *Сейчас самое время напомнить, что строки также являются одной из разновидностей коллекций – коллекцией символов. Внутренне класс* `System.String` *реализован как неизменяемый массив символов. Все операции над строками создают новый объект. Именно поэтому создание длинных строк путем конкатенации тысяч маленьких строк оказывается чрезвычайно неэффективной тратой времени. Эту проблему решает класс* `System.Text.StringBuilder`*, с реализацией, похожей на* `List<T>`*, удваивающей размер внутреннего хранилища при изменении содержимого. Всякий раз, когда возникает необходимость сконструировать строку из большого (или заранее неизвестного) числа маленьких строк, используйте* `StringBuilder` *для выполнения промежуточных операций.*

Такое богатство классов коллекций может показаться ошеломляющим, но иногда ни одна из встроенных коллекций не подходит для решения конкретной задачи. Ниже мы рассмотрим несколько примеров таких ситуаций. До выхода версии .NET 4.0, программисты часто ощущали нехватку поддержки конкурентного доступа во встроенных коллекциях: ни одна из коллекций, перечисленных в табл. 5.1, не поддерживает работу в многопоточном окружении. В .NET 4.0 появилось пространство имен `System.Collections.Concurrent` с несколькими новыми коллекциями, изначально предназначенных для использования в конкурентном окружении.

Параллельные коллекции

С появлением библиотеки Task Parallel Library в .NET 4.0 потребность в параллельных коллекциях встала особенно остро. В главе 6 будет представлено несколько интересных примеров организации параллельного доступа к источникам данных или выходным буферам из нескольких потоков выполнения. А пока мы сосредоточимся на доступных параллельных коллекциях и их характеристиках производительности в духе стандартных (не параллельных) коллекций, представленных выше (табл. 5.2).

Таблица 5.2. Параллельные коллекции в .NET Framework

Коллекция	Напоминает	Особенности	Синхронизация
ConcurrentStack<T>	Stack<T>	Односвязный список[1]	Свободный от блокировок алгоритм (CAS), экспоненциальное падение производительности при пробуксовке
ConcurrentQueue<T>	Queue<T>	Связанный список сегментов массива (по 32 элемента в каждом)[2]	Свободный от блокировок алгоритм (CAS), короткая пробуксовка при удалении из очереди элемента, который только что был включен в очередь

Таблица 5.2. (окончание)

Коллекция	Напоминает	Особенности	Синхронизация
ConcurrentBag<T>	–	Списки, локальные для потоков выполнения; поддерживает возможность захвата списков, принадлежащих другим потокам[3]	Обычно не требуется для локальных списков, для захвата используется `Monitor`
ConcurrentDictionary <K,V>	Dictionary<K,V>	Хеш-таблица: блоки и связанные списки[4]	Для изменения: один `Monitor` на каждую группу блоков хеш-таблицы (независимый от других блоков) Для чтения: нет

Примечания к столбцу «Особенности»:

1. *В главе 6 мы познакомимся с упрощенной реализацией стека без блокировок, с применением приема CAS (Compare-And-Swap – сравнить и заменить), и обсудим достоинства атомарных примитивов CAS.*
2. *Класс* `ConcurrentQueue<T>` *управляет связанным списком сегментов массива, что позволяет имитировать неограниченную очередь в ограниченном объеме памяти. Для добавления элементов в очередь и удаления их из очереди достаточно увеличивать указатели в сегментах массива. В некоторых случаях требуется синхронизация, например, чтобы гарантировать, что элемент не будет удален из очереди до того, как поток, добавивший этот элемент, закончит операцию добавления. Однако вся синхронизация основана на CAS.*
3. *Класс* `ConcurrentBag<T>` *управляет неупорядоченным списком элементов. Элементы хранятся в локальных (для потоков выполнения) списках; добавление элементов в локальный список и их удаление обычно не требуют синхронизации, так как добавление или удаление производится в голове списка. Когда потоку требуется получить элементы из списка другого потока, он захватывает хвост этого списка и конкурирует с другим потоком, только когда в списке содержится меньше трех элементов.*
4. *Класс* `ConcurrentDictionary<K,V>` *использует классическую реализацию хеш-таблицы (связанные списки в каждом блоке; общее описание организации хеш-таблиц см. в главе 3). Блокировки выполняются на уровне блоков – все операции с определенным блоком требуют захвата блокировки, количество которых ограничено параметром* `concurrencyLevel` *конструктора. Операции над блоками, связанными с разными блокировками, могут выполняться параллельно. Наконец, все операции чтения не требуют блокировки, потому что все операции, производящие изменения, выполняются атомарно (например, операция вставки нового элемента в список блока).*

Хотя большинство параллельных коллекций весьма похожи на свои непараллельные аналоги, они имеют несколько отличающийся набор методов, объясняемый их параллельной природой. Например, класс `ConcurrentDictionary<K,V>` имеет вспомогательные методы, существенно уменьшающие потребность в обязательных блокировках и решающие проблемы, связанные с состоянием гонки (race condition), которые могут возникать при неосторожном обращении со словарем:

```
// Этот код подвержен состоянию гонки
// между вызовами методов ContainsKey и Add:
Dictionary<string, int> expenses = ...;
if (!expenses.ContainsKey("ParisExpenses")) {
    expenses.Add("ParisExpenses", currentAmount);
} else {
    // Этот код подвержен состоянию гонки
    // при выполнении несколькими потоками:
    expenses["ParisExpenses"] += currentAmount;
}

// Следующий код использует метод AddOrUpdate, чтобы
// гарантировать корректную синхронизацию при
// добавлении нового или изменении существующего элемента:
ConcurrentDictionary<string, int> expenses = ...;
expenses.AddOrUpdate("ParisExpenses", currentAmount,
                     (key, amount) => amount + currentAmount);
```

Метод `AddOrUpdate` обеспечивает необходимую синхронизацию для составной операции «добавить или изменить». Существует похожий вспомогательный метод `GetOrAdd`, который может извлекать существующий элемент или добавлять новый элемент и возвращать его.

Проблемы, связанные с кешем

При выборе коллекции следует руководствоваться не только характеристиками производительности. Часто организация данных в памяти играет более важную роль для преимущественно вычислительных приложений, и коллекции очень разнятся в этом отношении. Основным фактором, вынуждающим очень внимательно относиться к особенностям организации коллекций в памяти, является кеш процессора.

Современные системы снабжаются большими объемами памяти. Восемь гигабайт памяти является стандартным значением для заурядной рабочей станции или игрового ноутбука. Микросхемы памяти типа DDR3 SDRAM дают задержки доступа к памяти порядка

15 наносекунд, что теоретически может обеспечить скорость передачи данных примерно 15 Гбайт/сек. С другой стороны быстрые процессоры могут выполнять миллиарды инструкций в секунду, то есть, рассуждая теоретически, задержка на 15 наносекунд в ожидании доступа к памяти – это десятки (а иногда сотни) невыполненных инструкций. Явление простоя в ожидании доступа к памяти известно под названием *удар о стену памяти* (hitting the memory wall) .

Чтобы увеличить расстояние между приложением и этой «стеной», современные процессоры снабжаются несколькими уровнями внутренней *кеш-памяти*, имеющей другие характеристики производительности, но очень дорогой и имеющей небольшой объем. Например, процессор Intel i7-860 на компьютере одного из авторов книге имеет три уровня кеша (рис. 5.3):

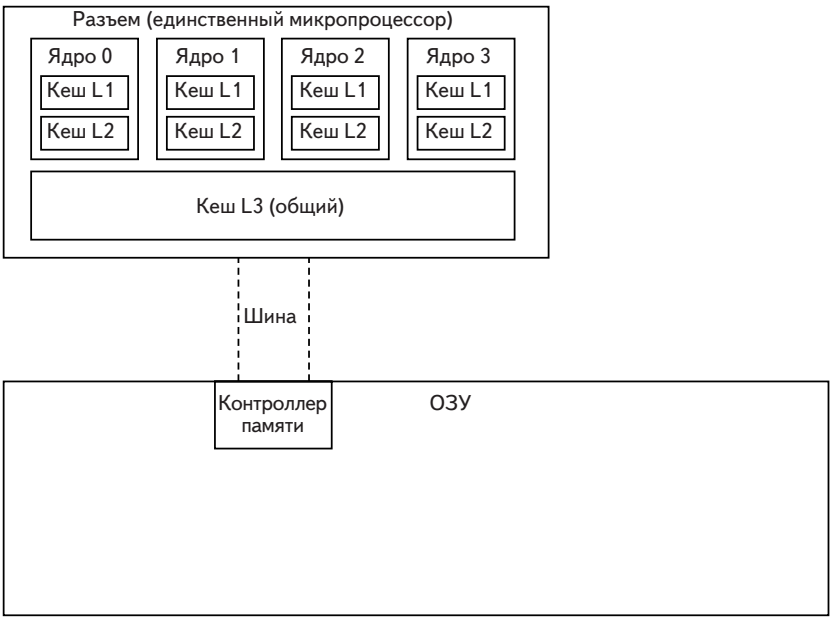

Рис. 5.3. Схема взаимосвязей между ядрами, кешем и ОЗУ в микропроцессоре Intel i7-860.

- кеш первого уровня для программных инструкций, 32 Кбайта, по одному на каждое ядро (всего 4 кеша);
- кеш первого уровня для данных, 32 Кбайта, по одному на каждое ядро (всего 4 кеша);

- кеш второго уровня для данных, 256 Кбайт, по одному на каждое ядро (всего 4 кеша);
- кеш третьего уровня для данных, 8 Мбайт, общий (всего 1 кеш).

Когда процессору требуется обратиться к ячейке памяти, он сначала проверяет, не хранятся ли необходимые данные в кеше первого уровня (L1). Если данные найдены в кеше, они извлекаются оттуда, на что тратится примерно 5 тактов процессорного времени (это называется *попаданием в кеш* (cache hit)) . В противном случае проверяется кеш второго уровня (L2); на извлечение данных из этого кеша тратится примерно 10 тактов. Аналогично выполняется проверка кеша третьего уровня (L3), на извлечение данных из которого тратится примерно 40 тактов. Наконец, если данные не найдены ни в одном из кешей, процессор обращается к главной памяти системы (это называется *промахом кеша* (cache miss)) . Когда процессор обращается к главной памяти, он читает из нее не один байт или слово, а *строку кеша* (cache line) , размер которой в современных системах составляет 32 или 64 байта. Обращение к любому слову в той же строке кеша уже не будет вызывать промах кеша, пока эта строка не будет вытеснена из кеша.

Хотя данное описание не позволяет получить полное представление об истинной сложности аппаратной реализации кеширования памяти SRAM и DRAM, тем не менее, оно дает достаточно пищи для ума и дальнейшего обсуждения влияния организации данных в памяти на высокоуровневые программные алгоритмы. Сейчас мы рассмотрим простой пример с одним ядром и одним кешем, а в главе 6 мы увидим, как многопроцессорные программы могут терять производительность из-за неаккуратного использования кешей нескольких ядер.

Допустим, что требуется реализовать обход большой коллекции целых чисел и выполнить некоторые агрегатные операции над ними, такие как вычисление суммы или среднего значения. Ниже приводятся два альтернативных решения; в одном используется `LinkedList<int>` а в другом – массив целых чисел (`int[]`), две коллекции, встроенные в .NET.

```
LinkedList<int> numbers = new LinkedList<int>(Enumerable.Range(0,
20000000));
int sum = 0;
for (LinkedListNode<int> curr = numbers.First; curr != null;
curr = curr.Next)
{
    sum += curr.Value;
```

```
}

int[] numbers = Enumerable.Range(0, 20000000).ToArray();
int sum = 0;
for (int curr = 0; curr < numbers.Length; ++curr) {
    sum += numbers[curr];
}
```

В системах, упомянутых выше, вторая версия действует *в 2 раза быстрее* первой. Это весьма существенная разница, но судя по количеству машинных инструкций, сгенерированных JIT-компилятором, такой разницы не должно быть. Обход связанного списка заключается в переходе от одного узла к другому, а обход массива – в наращивании индекса. (Фактически, без оптимизаций со стороны JIT-компилятора, доступ к массиву требует также проверки на выход за его границы.)

```
; инструкции на языке ассемблера x86 для первого цикла,
; предполагается, что "сумма" хранится в EAX, а "числа" - в ECX
xor eax, eax
mov ecx, dword ptr [ecx+4]       ; curr = numbers.First
test ecx, ecx
jz LOOP_END
LOOP_BEGIN:
add eax, dword ptr [ecx+10]      ; sum += curr.Value
mov ecx, dword ptr [ecx+8]       ; curr = curr.Next
test ecx, ecx
jnz LOOP_BEGIN                   ; всего 4 инструкции на итерацию
LOOP_END:
...

; инструкции на языке ассемблера x86 для второго цикла,
; предполагается, что "сумма" хранится в EAX, а "числа" - в ECX
mov edi, dword ptr [ecx+4]       ; numbers.Length
test edi, edi
jz LOOP_END
xor edx, edx                     ; индекс цикла
LOOP_BEGIN:
add eax, dword ptr [ecx+edx*4+8] ; sum += numbers[i], без
проверки границ
inc edx
cmp esi, edx
jg LOOP_BEGIN                    ; всего 4 инструкции на итерацию
LOOP_END:
...
```

Опираясь только на один этот код, сгенерированный компилятором для обоих циклов (при запрещенных оптимизациях, таких как использование инструкций SIMD для обхода массива, занимающего

непрерывную область памяти), сложно объяснить существенное различие в производительности. В действительности, чтобы дать достаточно обоснованное объяснение, необходимо исследовать порядок обращения к памяти.

В обоих циклах доступ к каждому целому числу осуществляется всего один раз, поэтому может показаться, что ускорение, которое дает кеш процессора, здесь ни при чем, потому что не используется преимущество попадания в кеш при повторном обращении к данным. Тем не менее, организация размещения данных в памяти оказывает существенное влияние на производительность, но не из-за того, что данные повторно используются, а из-за способа их загрузки в кеш-память. При обращении к элементу массива, вначале происходит промах кеша и загрузка строки кеша, содержащей 16 последовательно расположенных целых чисел (строка кеша = 64 байта = 16 целых чисел). Так как доступ к элементам массива выполняется последовательно, следующие 15 чисел оказываются в кеше и при обращении к ним не возникает промаха кеша. Это почти идеальный сценарий с соотношением промахов кеша 1:16. С другой стороны, при обращении к элементам связанного списка, вначале происходит промах кеша и загрузка строки кеша, содержащей *не более* 3 последовательно расположенных узлов списка, что дает в результате соотношение промахов кеша 1:3! (Узел содержит указатели на следующий и предыдущий элементы, и само целое число, что составляет 12 байт в 32-разрядной системе; заголовок ссылочного увеличивает этот объем до 20 байт на узел.)

Более высокое соотношение промахов кеша и является основной причиной, объясняющей разницу в производительности. Кроме того, мы взяли за основу идеальный сценарий, когда узлы связанного списка хранятся в памяти последовательно, без разрывов между ними, что может быть, только если память для них выделялась почти одновременно, без других операций выделения памяти между ними, а это весьма маловероятно. Если бы узлы связанного списка оказались разбросаны в памяти не так идеально, соотношение промахов кеша было бы еще выше, а производительность – хуже.

В заключение рассмотрим еще один пример, демонстрирующий другой эффект, связанный с кешем – реализацию алгоритма умножения матриц. Алгоритм умножения матриц (к которому мы еще вернемся в главе 6, когда будем обсуждать C++ AMP) – очень прост и может извлечь существенные выгоды от использования кеша процессора, потому что использует каждый элемент матриц несколько раз. Ниже приводится наивная реализация этого алгоритма «в лоб»:

```
public static int[,] MultiplyNaive(int[,] A, int[,] B) {
    int[,] C = new int[N, N];
    for (int i = 0; i < N; ++i)
        for (int j = 0; j < N; ++j)
            for (int k = 0; k < N; ++k)
                C[i, j] += A[i, k] * B[k, j];
    return C;
}
```

Основу реализации составляет внутренний цикл, вычисляющий скалярное произведение *i*-й строки в первой матрице на *j*-й столбец – во второй; он выполняет полный обход *i*-й строки *j*-го столбца. Предпосылкой, позволяющей говорить о возможности увеличить производительность за счет использования кеша, является тот факт, что при вычислении *i*-й строки матрицы результата многократно выполняется обход *j*-й строки первой матрицы. Один и тот же элемент используется многократно. При обходе первой матрицы, казалось бы, кеш используется достаточно эффективно: сначала выполняется *N* итераций по ее первой строке, затем *N* итераций по второй строке, и так далее. Однако это не так, потому что после *i*-й итерации внешнего цикла, не происходит возврат к *j*-й строке. К сожалению, обход второй матрицы полностью исключает возможность использовать кеш: сначала выполняется обход элементов ее первого *столбца N* раз, затем второго *столбца*, и так далее. (Причина отсутствия возможности использовать кеш в том, что матрица, массив типа `int[,]`, хранится в памяти построчно, как показано на рис. 5.4.)

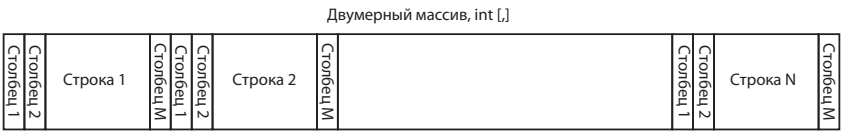

Рис. 5.4. Размещение двумерного массива (int[,]) в памяти. Матрица хранится построчно.

Если бы кеш имел достаточно большой объем, чтобы вторая матрица уместилась в нем целиком, тогда после одной итерации внешнего цикла вторая матрица оказалась бы в кеше и последующие обращения к ее элементам удовлетворялись бы из кеша. Однако, если размер второй матрицы превышает объем кеша, промахи кеша будут происходить слишком часто: промах кеша при обращении к элементу (*i*,*j*) повлечет за собой чтение строки кеша, содержащей элементы строки *i*, но другие элементы столбца *j* в кеш не попадут, то есть промахи будут происходить при каждом обращении!

Умножение матриц с разделением на блоки приводит к следующей идее. Умножение матриц можно выполнять, как показано выше, или разбив их на более маленькие матрицы (блоки), а затем умножая блоки и выполняя некоторые дополнительные арифметические операции для получения окончательного результата.

В частности, если матрицы A и B разбить на блоки, как показано на рис. 5.5, тогда матрицу $C = AB$ можно вычислить поблочно, то есть: $C_{ij} = A_{i1}B_{1j} + A_{i2}B_{2j} + ... + A_{ik}B_{kj}$. На практике это приводит к следующей реализации:

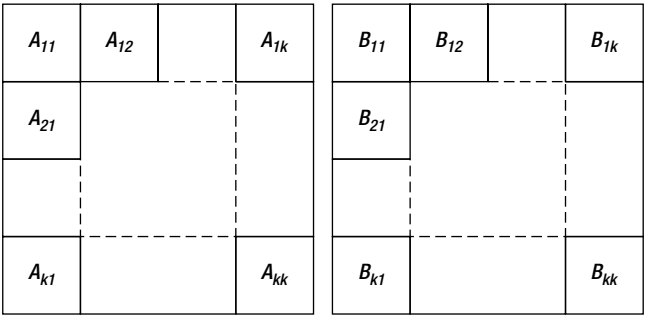

Рис. 5.5. Матрицы A и B, разбитые на блоки размером $k \times k$.

```
public static int[,] MultiplyBlocked(int[,] A, int[,] B, int bs) {
    int[,] C = new int[N, N];
    for (int ii = 0; ii < N; ii += bs)
        for (int jj = 0; jj < N; jj += bs)
            for (int kk = 0; kk < N; kk += bs)
                for (int i = ii; i < ii + bs; ++i)
                    for (int j = jj; j < jj + bs; ++j)
                        for (int k = kk; k < kk + bs; ++k)
                            C[i, j] += A[i, k] * B[k, j];
    return C;
}
```

Шесть вложенных циклов очень просты – три внутренних цикла выполняют матричное умножение двух блоков, а три внешних выполняют итерации по блокам. Чтобы проверить алгоритм поблочного умножения, мы использовали тот же компьютер, что и в предыдущем примере (имеющем кеш 3-го уровня объемом 8 Мбайт) и выполнили умножение двух матриц размером 2048 × 2048 целых чисел. Общий размер двух матриц составляет 2048 × 2048 × 4 × 2 = 32 Мбайт, что явно больше размера кеша. Результаты, полученные для блоков разного размера, показаны в табл. 5.3, где можно видеть, что разделение

на блоки может существенно повысить производительность, и что выбор оптимального размера блока может дать существенную прибавку к скорости:

Таблица 5.3. Хронометраж алгоритма поблочного умножения матриц с блоками различного размера

	Без деления на блоки	bs=4	bs=8	bs=16	bs=32	bs=64	bs=512	bs=1024
Время (сек)	178	92	81	81	79	106	117	147

Можно привести массу примеров, где учет особенностей кеша оказывает существенное влияние на производительность, при чем не только среди алгоритмов обработки коллекций. Имеются и другие, более тонкие аспекты использования кеша и организации хранения данных в памяти: взаимоотношения между кешами разных уровней, влияние ассоциативности кеша, упорядоченность данных в памяти и многие другие. Дополнительные примеры можно найти в статье Игоря Островского (Igor Ostrovsky), «Gallery of Processor Cache Effects» (http://igoro.com/archive/galleryof-processor-cache-effects/, 2010).

Собственные коллекции

Существует множество типов коллекций, хорошо известных в информатике, но не попавших в .NET Framework. Некоторые из них получили весьма широкое распространение, и ваши приложения могли бы получить определенные выгоды от их использования. Кроме того, большинство из них может быть реализовано в достаточно короткие сроки. Не смотря на то, что мы не ставим перед собой целью заняться исследованием различных типов коллекций, тем не менее, приведем два примера коллекций, существенно отличающихся от коллекций в .NET, и предлагаем рассмотреть ситуации, когда применение собственных коллекций может оказаться полезным.

Система непересекающихся множеств

Система непересекающихся множеств (disjoint-set) – это коллекция, элементы которой хранятся в виде непересекающихся множеств. Она отличается от коллекций .NET тем, что не позволяет сохранять в ней элементы. Вместо этого образуется домен элементов, в котором каждый элемент образует единственное множество, и определяется

последовательность операций по объединению в более крупные множества. Эта структура данных обеспечивает высокую эффективность двух операций::

- *объединение*: объединение двух подмножеств с целю получить общее подмножество;
- *поиск*: определение подмножества, которому принадлежит элемент (часто используется, чтобы определить, принадлежат ли два указанных элемента одному подмножеству).

Обычно операции с множествами выполняются на уровне элементов-представителей – с единственным представителем от каждого множества. Операции объединения и поиска принимают и возвращают представителей, а не целые множества.

Наивная реализация системы непересекающихся множеств вовлекает использование коллекции для представления каждого множества, и слияние коллекций при необходимости. Например, при использовании связанного списка для хранения каждого множества, время их слияния прямо пропорционально количеству множеств а операция поиска может занимать постоянное время, если каждый элемент имеет указатель на представителя множества.

Реализация алгоритма Галлера-Фишера (Galler-Fischer) имеет *намного* более высокую сложность. Множества хранятся в виде «леса» (множества деревьев); каждый узел каждого дерева хранит указатель на его родительский узел, а корнем дерева является представитель множества. Чтобы обеспечить сбалансированность получающихся деревьев, при слиянии деревьев меньшее дерево всегда присоединяется к корню большего дерева (это требует следить за глубиной дерева). Кроме того, операция поиска сжимает путь от желаемого элемента до его представителя. Ниже представлена схематическая реализация этого алгоритма:

```
public class Set<T> {
    public Set Parent;
    public int Rank;
    public T Data;
    public Set(T data) {
        Parent = this;
        Data = data;
    }

    public static Set Find(Set x) {
        if (x.Parent != x) {
            x.Parent = Find(x.Parent);
        }
        return x.Parent;
```

```
    }

    public static void Union(Set x, Set y) {
        Set xRep = Find(x);
        Set yRep = Find(y);
        if (xRep == yRep) return; // То же самое множество

        if (xRep.Rank < yRep.Rank) xRep.Parent = yRep;
        else if (xRep.Rank > yRep.Rank) yRep.Parent = xRep;
        else {
            yRep.Parent = xRep; // Объединение двух деревьев
                                // с одинаковым весом,
            ++xRep.Rank;        // поэтому вес увеличивается
        }
    }
}
```

Точное измерение производительности этой структуры данных является весьма сложной задачей. В простейшем случае верхней границей амортизированного времени операции в лесу из n элементов является $O(\log^* n)$, где $\log^* n$ (итерационное вычисление логарифма) – количество применений функции вычисления логарифма для получения результата меньше единицы, то есть, минимальное количество появлений «log» в неравенстве $\log \log \log \ldots \log n \le 1$. Для практических значений n, например, $n \le 1050$, это число не превышает 5 и является «почти постоянным».

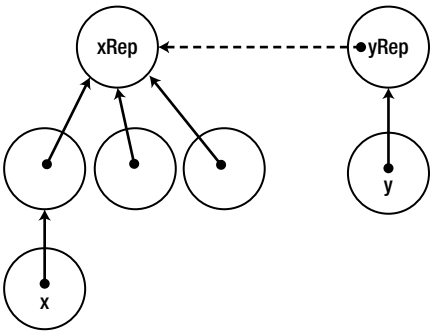

Рис. 5.6. Слияние двух множеств, *x* и *y*, где y является меньшим множеством. Пунктирная стрелка показывает результат слияния.

Список с пропусками

Список с пропусками (skip list) – это структура данных, хранящая сортированный связанный список элементов и позволяющая выполнять поиск за время $O(\log n)$, что сравнимо со временем поиска

методом дихотомии в массиве или в сбалансированном двоичном дереве. Как известно, основной проблемой реализации поиска методом дихотомии в связанном списке является невозможность произвольного обращения к его элементам по индексу. Списки с пропусками устраняют это ограничение за счет использования иерархии все более и более разреженные связанных списков: Первый связанный список связывает все узлы; второй список связывает узлы 0, 2, 4, ...; третий связывает узлы 0, 4, 8, ...; четвертый связывает узлы 0, 8, 16, ...; и так далее.

Чтобы найти элемент в списке с пропусками, сначала выполняется обход самого разреженного списка. Когда встречается элемент, больше или равный искомому, возвращается предыдущий элемент и выполняется переход к следующему списку в иерархии. Так повторяется, пока желаемый элемент не будет найден. За счет использования в иерархии списков $O(\log n)$, гарантируется время поиска $O(\log n)$.

К сожалению, задача управления элементами списка с пропусками иногда является далеко нетривиальной задачей. Если при добавлении или удалении элемента потребуется перераспределить память для всего связанного списка, список с пропусками не сможет дать какие-либо преимущества перед обычными структурами данных, такими как сортированный список `SortedList<T>`, который просто хранит сортированный массив. Типичный подход к решению этой проблемы заключается в рандомизации иерархии списков (рис. 5.7), что позволяет получить ожидаемое логарифмическое время на вставку, удаление и поиск элементов. Более точное и подробное описание особенностей списка с пропусками можно найти в статье Уильяма Пью (William Pugh), «Skip lists: a probabilistic alternative to balanced trees» (ACM, 1990).

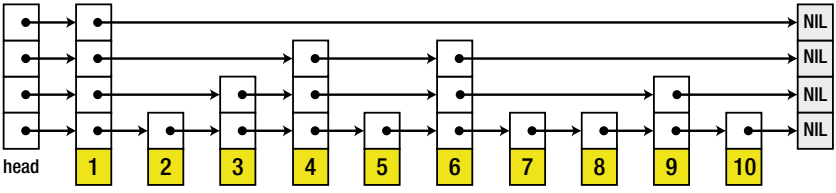

Рис. 5.7. Структура списка с пропусками с четырьмя рандомизированными списками в иерархии (изображение взято из Википедии: http://upload.wikimedia.org/wikipedia/commons/8/86/Skip_list.svg).

Одноразовые коллекции

Может так же случиться, что вы окажетесь в уникальной ситуации, когда решить поставленную задачу будет возможно только с применением собственной коллекции. Мы называем их *одноразовыми коллекциями* (one-shot collections), потому что они являются совершенно новыми изобретениями, пригодными для решения определенной задачи. Со временем вы можете обнаружить, что какие-то из ваших одноразовых коллекций вполне можно использовать повторно. И в этом разделе мы познакомимся с одной такой коллекцией.

Представьте такую ситуацию: вы создали биржевую информационную систему, снабжающую продавцов батончиков информацией о ценах на разные батончики. Основная таблица с данными хранится в памяти и содержит по одной строке для батончика каждого типа, содержащей текущую цену на данный батончик. В табл. 5.4 представлен пример этой таблицы с данными в некоторый момент времени:

Таблица 5.4. Пример таблицы с данными в информационной системе для продавцов батончиков

Батончик	Цена ($)
Twix	0.93
Mars	0.88
Snickers	1.02
Kisses	0.66

Этой системой пользуются две категории клиентов.

- Продавцы батончиков подключаются к системе через сокет TCP, и периодически запрашивают свежую информацию об определенном типе батончиков. Типичный запрос от продавца имеет вид: «Какова цена на Twix?». А типичный ответ: «$0.93». Каждую секунду в систему поступает десятки тысяч таких запросов.
- Производители батончиков подключаются к системе через сокет UDP и периодически устанавливают новую цену на свои батончики. Запросы от производителей делятся на два подтипа.
  - «Установить цену на Mars равной $0.91». Отвечать на такой запрос не требуется. Каждую секунду в систему поступают тысячи таких запросов.

- «Добавить новый батончик Snowflakes с начальной ценой $0.49». Отвечать на такой запрос не требуется. Таких запросов поступает в систему не больше нескольких десятков в день.

Известно также, что 99.9% операций чтения или обновления цены выполняется для типов батончиков, существовавших к моменту начала торгов, и только 0.1% операций выпадает на долю вновь добавленных батончиков.

Вооружившись этой информацией, вы решили спроектировать структуру данных – коллекцию – для хранения таблицы с данными в памяти. Эта структура должна поддерживать возможность использования в многопоточной среде, потому что сотни потоков выполнения могут одновременно попытаться обратиться к ней. Вам не нужно заботиться о сохранении данных в некотором постоянном хранилище – мы исследуем характеристики производительности только для случая нахождения коллекции в памяти.

Форма данных и типы запросов диктуют необходимость использовать хеш-таблицу. Синхронизация доступа к хеш-таблице является сложной задачей, которую лучше переложить на плечи `ConcurrentDictionary<K,V>`. Чтение из параллельного словаря можно выполнять вообще без синхронизации, а вот операции изменения цены и добавления нового типа батончиков требуют узконаправленной синхронизации. Хотя такое решение может быть вполне приемлемым, тем не менее, мы поднимем планку: нам хотелось бы обеспечить выполнение чтения и изменения цены вообще без синхронизации, в 99.9% операций с существующими типами батончиков.

Одним из возможных решений может служить *безопасный-небезопасный кеш* (safe-unsafe cache). Эта коллекция является множеством из двух хеш-таблиц, *безопасной таблицы* (safe table) и *небезопасной таблицы* (unsafe table). Безопасная таблица заполняется информацией о типах батончиков, существовавших к моменту начала торгов; небезопасная таблица изначально пуста. Операции с безопасной таблицей выполняются без блокировки, потому что она не изменяется; новые типы батончиков добавляются в небезопасную таблицу. Ниже представлена возможная реализация этой структуры данных с использованием `Dictionary<K,V>` и `ConcurrentDictionary<K,V>`:

```
// Предполагается, что запись TValue может выполняться атомарно,
// то есть это должен быть ссылочный тип  или достаточно
// небольшой тип значения (4 байта в 32-разрядных системах).
public class SafeUnsafeCache<TKey, TValue> {
```

```
    private Dictionary<TKey, TValue> safeTable;
    private ConcurrentDictionary<TKey, TValue> unsafeTable;

    public SafeUnsafeCache(IDictionary<TKey, TValue> initialData) {
        safeTable = new Dictionary<TKey, TValue>(initialData);
        unsafeTable = new ConcurrentDictionary<TKey, TValue>();
    }
    public bool Get(TKey key, out TValue value) {
        return safeTable.TryGetValue(key, out value)
               || unsafeTable.TryGetValue(key, out value);
    }
    public void AddOrUpdate(TKey key, TValue value) {
        if (safeTable.ContainsKey(key)) {
            safeTable[key] = value;
        } else {
            unsafeTable.AddOrUpdate(key, value, (k, v) => value);
        }
    }
}
```

Следующим шагом в развитии этой структуры данных могла бы быть периодическая приостановка торговых операций и объединение безопасной и небезопасной таблиц. Это еще больше уменьшило бы потребность в синхронизации для доступа к данным.

Реализация IEnumerable<T> и других интерфейсов

Почти любая коллекция в конечном счете реализует интерфейс `IEnumerable<T>` и, возможно, другие интерфейсы, имеющие отношение к коллекциям. Реализация этих интерфейсов дает массу преимуществ, в том числе, начиная с версии .NET 3.5, поддержку LINQ. В конце концов, любой класс, реализующий интерфейс `IEnumerable<T>`, автоматически снабжается разнообразными дополнительными методами `System.Linq` и может использоваться в выражениях C# 3.0 LINQ, наравне со встроенными коллекциями.

К сожалению, прямолинейная реализация интерфейса `IEnumerable<T>` в коллекциях вынуждает вызывающую программу платить производительностью за вызовы методов интерфейса. Взгляните на следующий фрагмент, выполняющий обход коллекции `List<int>`:

```
List<int> list = ...;
IEnumerator<int> enumerator = list.GetEnumerator();
long product = 1;
while (enumerator.MoveNext()) {
    product *= enumerator.Current;
}
```

В каждой итерации в этом примере вызываются два метода интерфейса, что влечет за собой лишние накладные расходы при попытке обойти спи-

сок и вычислить произведение его элементов. Как рассказывалось в главе 3, встраивание методов интерфейсов не самая простая задача, и если JIT-компилятору не удастся ее решить, стоимость их вызовов получится весьма высокой.

Существует несколько решений, способных помочь избежать лишних накладных расходов при вызове методов. Когда методы интерфейсов применяются непосредственно к переменной типа значения, они вызываются напрямую. То есть, если бы переменная enumerator в примере выше имела тип значения (а не `IEnumerator<T>`), стоимость вызова методов интерфейса была бы намного ниже. Если бы коллекция реализовала метод `GetEnumerator`, возвращающий непосредственно экземпляр типа значения, вызывающая программа смогла бы использовать его методы вместо методов интерфейса.

Для этого, например, класс `List<T>` явно реализует метод `IEnumerable<T>.GetEnumerator`, возвращающий `IEnumerator<T>`, и еще один общедоступный метод `GetEnumerator`, возвращающий `List<T>.Enumerator` – внутренний экземпляр типа значения:

```
public class List<T> : IEnumerable<T>, ... {
    public Enumerator GetEnumerator() {
        return new Enumerator(this);
    }
    IEnumerator<T> IEnumerable<T>.GetEnumerator() {
        return new Enumerator(this);
    }
    ...
    public struct Enumerator { ... }
}
```

Это позволяет писать такой код:

```
List<int> list = ...;
List.Enumerator<int> enumerator = list.GetEnumerator();
long product = 1;
while (enumerator.MoveNext()) {
    product *= enumerator.Current;
}
```

избавляющий от вызовов методов интерфейса.

Альтернативное решение заключается в создании итератора ссылочного типа, но использующего тот же трюк с явной реализацией интерфейса – метода `MoveNext` и свойства `Current`. Оно также позволит вызывающей программе использовать метод и свойство класса непосредственно, избежав накладных расходов на вызовы методов интерфейса.

В заключение

В этой главе мы познакомились с реализацией более чем десятка коллекций и сравнили их по разным параметрам, начиная от плотности размещения в памяти до сложности реализации, требований к организации в памяти и возможности использования в многопоточной среде. Теперь вы должны лучше ориентироваться в выборе коллекций, уметь обосновывать оптимальность своего выбора, и не должны бояться создавать свои одноразовые коллекции или использовать идеи, почерпнутые из книг по информатике.

ГЛАВА 6. Конкуренция и параллелизм

Долгие годы вычислительные мощности компьютерных систем увеличивались экспоненциально. Скорость процессоров росла с каждой новой моделью, и программы, прежде создававшиеся в расчете на высокопроизводительные и дорогостоящие рабочие станции, стали переноситься на ноутбуки и планшетные устройства. Эта эра закончилась несколько лет тому назад, и быстродействие современных процессоров увеличивается уже не экспоненциально; зато экспоненциально стало увеличиваться их количество. Создавать программы, использующие преимущества многопроцессорных архитектур было непростым делом, когда такие системы были редкими и дорогостоящими, но оно не стало простым делом и сейчас, когда даже смартфоны снабжаются двух- и четырехъядерными процессорами.

В этой главе мы бегло познакомимся с современной поддержкой параллельного программирования в .NET. В одной скромной главе невозможно описать все библиотеки, фреймворки, инструменты, ловушки, шаблоны проектирования и архитектурные модели, тем не менее, ни одна книга, посвященная проблемам производительности, не могла бы считаться полной без обсуждения одного из самых недорогих способов увеличения производительности приложений, а именно, масштабирования в многопроцессорных системах.

Перспективы и преимущества

Одной из захватывающих перспектив, с точки зрения параллельного программирования, является все возрастающее разнообразие многопроцессорных систем. Производители процессоров с гордостью говорят о возможности предложить недорогие и доступные процессоры с четырьмя или восемью ядрами для обычных персональных ком-

пьютеров, и процессоры с десятками ядер для высокопроизводительных серверов. Средние современные рабочие станции или мощные ноутбуки часто снабжаются мощными графическими процессорами (GPU), обладающими возможностью обслуживать *сотни* потоков выполнения. В дополнение к двум обычным приемам параллельного программирования появилась *инфраструктура, предоставляемая как услуга* (Infrastructure-as-a-Service, IaaS), с еженедельно снижающейся стоимостью использования, в лице которой вы в мгновение ока можете получить облачное окружение с несколькими тысячами ядер.

> **Примечание.** *Герб Саттер (Herb Sutter) дал превосходный обзор гетерогенного мира фреймворков параллельного программирования в своей статье «Welcome to the Jungle» (2011). В другой своей статье «The Free Lunch Is Over», написанной в 2005 году, он предсказал возрождение интереса к применению фреймворков параллельного программирования. Если информации о параллельном программировании, что приводится в этой главе, для вас окажется недостаточно, мы можем порекомендовать обратиться к следующим замечательным книгам, посвященным этой теме в целом и фреймворкам для платформы .NET в частности: Джо Даффи (Joe Duffy), «Concurrent Programming on Windows» (Addison-Wesley, 2008); Джозеф Албахари (Joseph Albahari), «Threading in C#» (электронная версия, 2011). Желающие вникнуть во внутренние механизмы операционной системы, обеспечивающие поддержку многопоточного выполнения и синхронизации, могут обратиться к книге Марка Руссиновича (Mark Russinovich), Дэвида Соломона (David Solomon) и Алекса Ионеску (Alex Ionescu), «Windows Internals, 5th Edition» (Microsoft Press, 2009). Наконец, отличным источником информации по библиотекам, таким как Task Parallel Library, является сайт MSDN.*

Рост производительности, получаемый за счет применения приемов параллельного программирования, достаточно велик, чтобы от него легко было отказаться. Приложения, в основном занимающиеся операциями ввода/вывода, могут получить огромные преимущества за счет переноса этих операций в отдельные потоки, и выполнения их асинхронно и параллельно, благодаря чему обеспечивается более высокая отзывчивость и масштабируемость. Преимущественно вычислительные приложения, реализующие параллельные алгоритмы, способны увеличивать свою производительность на порядок, за счет использования всех доступных процессоров, или на два порядка, за счет использования ядер графического процессора. Далее в этой главе будет показано, как можно в 130 раз ускорить простую реализацию умножения матриц лишь за счет изменения нескольких строк кода, обеспечивающих выполнение на графическом процессоре.

Как всегда, дорога к параллелизму полна ловушек – взаимоблокировки (deadlocks), состояния гонки (race conditions), «голодание» (starvation) и повреждение данных в памяти – поджидающих на каждом шагу. Новейшие параллельные фреймворки, включая библиотеки Task Parallel Library (.NET 4.0) и C++ AMP, которые мы будем использовать в этой главе, стремятся уменьшить сложность разработки параллельных приложений и дать возможность извлечь максимальную выгоду в виде высокой производительности.

Зачем использовать приемы параллельного программирования?

Существует множество причин разбиения приложений на несколько потоков выполнения. Эта книга посвящена вопросам повышения производительности программ, а большинство причин использования приемов параллельного программирования как раз связаны с производительностью. Например.

- Применение асинхронных операций ввода/вывода может повысить отзывчивость приложения. В большинстве приложений с графическим интерфейсом за все изменения в пользовательском интерфейсе отвечает единственный поток; этот поток никогда не должен блокироваться на продолжительные периоды времени, чтобы своевременно реагировать на действия пользователя.
- Распараллеливание работы между несколькими потоками выполнения позволяет полнее использовать системные ресурсы. Современные системы, построенные на многоядерных процессорах и многоядерных графических процессорах, способны на порядки увеличивать производительность простых вычислительных алгоритмов за счет их распараллеливания.
- Одновременное выполнение нескольких операций ввода/вывода (например, одновременное получение цен с нескольких сайтов туристических агентств или изменение файлов в нескольких распределенных веб-репозиториях) может увеличить общую пропускную способность, потому что в подобных ситуациях большая часть времени тратится на ожидание завершения ввода/вывода, а освободившееся время может быть потрачено на выполнение дополнительных операций ввода/вывода или обработку результатов уже завершившихся операций.

От потоков к пулам потоков и задачам

Вначале были потоки. Потоки – это наиболее элементарные средства распараллеливания и асинхронного выполнения заданий; они являются самой низкоуровневой абстракцией, доступной программам, выполняющимся с привилегиями обычного пользователя. Потоки предлагают не так много в смысле структуры и управления, а программирование потоков близко напоминает давно прошедшие дни неструктурированного программирования, до того как обрели популярность подпрограммы, объекты и агенты.

Рассмотрим простую задачу: дан большой диапазон натуральных чисел, в нем требуется найти все простые числа и сохранить их в коллекции. Это – исключительно вычислительная задача и ее легко можно разбить на несколько подзадач, решаемых параллельно. Но, для начала реализуем самое простое решение, использующее единственный поток выполнения:

```
// Возвращает все простые числа в диапазоне [start, end)
public static IEnumerable <uint> PrimesInRange(uint start, uint end) {
    List <uint> primes = new List <uint> ();
    for (uint number = start; number < end; ++number) {
        if (IsPrime(number)) {
            primes.Add(number);
        }
    }
    return primes;
}

private static bool IsPrime(uint number) {
    // Крайне неэффективный алгоритм O(n),
    // но достаточный для демонстрационных целей
    if (number == 2) return true;
    if (number % 2 == 0) return false;
    for (uint divisor = 3; divisor < number; divisor += 2) {
        if (number % divisor == 0) return false;
    }
    return true;
}
```

Можно ли здесь что-то улучшить, если допустить, что используется самый оптимальный алгоритм и в нем нечего больше улучшать? Для достаточно широкого диапазона чисел, такого как [100, 200 000], реализация выше работает в течение нескольких секунд на современном процессоре, оставляя простор для оптимизации.

У кого-то могут возникнуть серьезные сомнения в эффективности алгоритма (например, весьма тривиальная оптимизация может повысить его производительность до $O(\sqrt{n})$, вместо $O(n)$), но, независимо от оптимальности алгоритма, одного взгляда достаточно, чтобы заметить, что его легко можно распараллелить. В конце концов, проверка числа 4977 на принадлежность к категории простых чисел никак не зависит от проверки числа 3221, поэтому самый простой способ распараллелить решение задачи состоит в том, чтобы разделить диапазон чисел на фрагменты и запустить дополнительные потоки выполнения для параллельной обработки фрагментов (как показано на рис. 6.1). Разумеется, нам придется синхронизировать доступ к коллекции простых чисел, чтобы предотвратить повреждение данных. Ниже приводится простейшая реализация такого решения:

```
public static IEnumerable <uint> PrimesInRange(uint start, uint end) {
    List <uint> primes = new List <uint> ();
    uint range = end - start;
    uint numThreads = (uint)Environment.ProcessorCount; // это удачная идея?
    uint chunk = range / numThreads; // надеемся, деление будет без остатка
    Thread[] threads = new Thread[numThreads];
    for (uint i = 0; i < numThreads; ++i) {
        uint chunkStart = start + i*chunk;
        uint chunkEnd = chunkStart + chunk;
        threads[i] = new Thread(() = > {
            for (uint number = chunkStart; number < chunkEnd; ++number) {
                if (IsPrime(number)) {
                    lock(primes) {
                        primes.Add(number);
                    }
                }
            }
        });
        threads[i].Start();
    }
    foreach (Thread thread in threads) {
        thread.Join();
    }
    return primes;
}
```

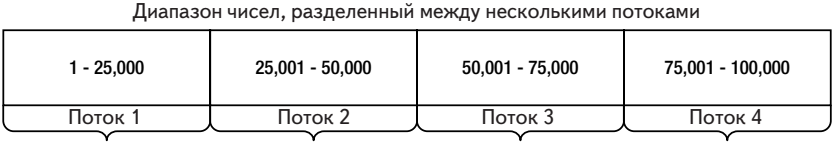

Рис. 6.1. Деление диапазона чисел для обработки несколькими потоками выполнения.

В системе на процессоре Intel i7 последовательная версия обрабатывала диапазон чисел [100, 200 000] в среднем за ~2950 миллисекунд, а параллельная версия – за ~950 миллисекунд. От системы с 8-ядерным процессором можно было бы ожидать большего. Но дело в том, что данная модель процессоров i7 использует технологию HyperThreading, а это означает, что в процессоре имеется лишь 4 физических ядра (каждое физическое ядро делится на два логических). Учитывая это вполне можно было бы ожидать 4-кратного прироста производительности, но мы получили лишь 3-кратный, что кажется недостаточным. Однако, как показывают результаты применения профилировщика параллелизма (Concurrency Profiler), изображенные на рис. 6.2 и рис. 6.3, некоторые потоки завершили работу раньше других, в результате чего общее использование процессора оказалось много ниже 100% (порядок запуска профилировщика параллелизма описывается в главе 2).

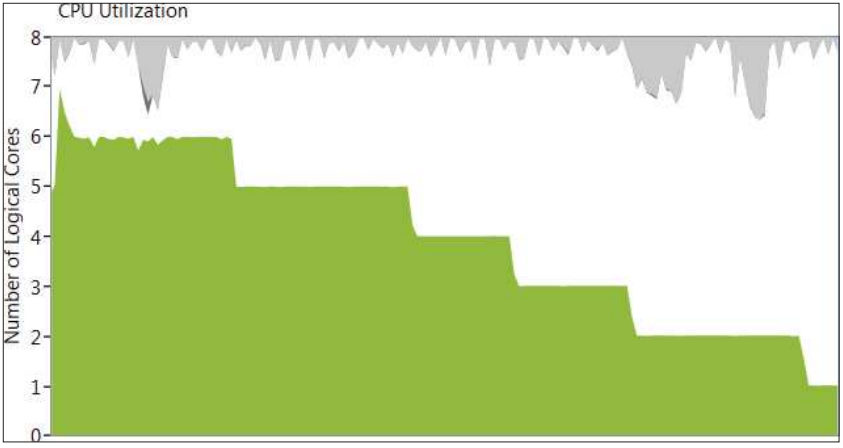

Рис. 6.2. Общее использование процессора сначала повысилось почти до 8 логических ядер (100%) а затем постепенно снижалось.

В действительности эта программа может работать намного быстрее последовательной версии (хотя масштабируемость и *не* будет линейной), особенно если задействовать большое количество ядер. Однако при этом возникает несколько вопросов.

- Какое количество потоков будет оптимальным? Следует ли в системе на процессоре с восемью ядрами создавать восемь потоков?

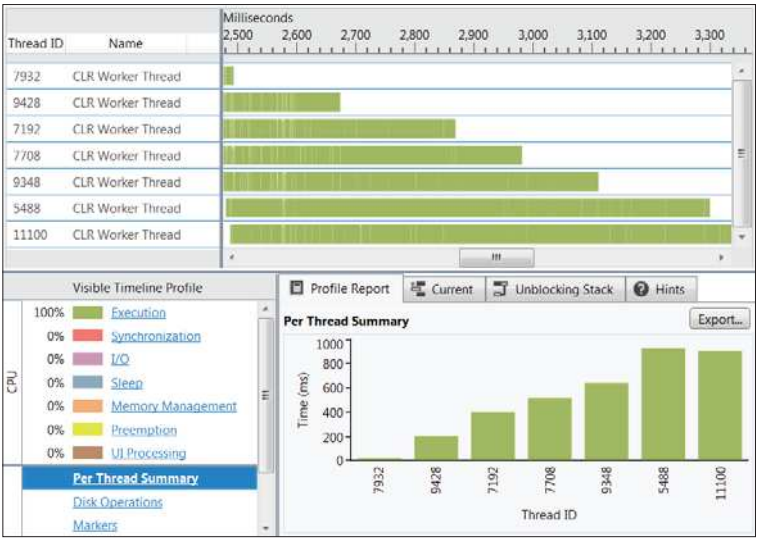

Рис. 6.3. Некоторые потоки справились со своей долей работы значительно раньше других. Поток 9428 выполнялся менее 200 мсек, а поток 5488 более 800 мсек.

- Как убедиться, что программа не монополизирует системные ресурсы и не превышает ее возможности? Например, что произойдет, если другой поток выполнения в нашем процессе попытается найти простые числа с использованием того же самого параллельного алгоритма?
- Как синхронизировать доступ потоков выполнения к коллекции с результатами? Одновременный доступ к `List<uint>` из нескольких потоков выполнения небезопасен и может повлечь повреждение данных, как будет показано в следующем разделе. Однако, приобретение блокировки перед записью каждого найденного простого числа в коллекцию (что делает решение, представленное выше) слишком дорогое удовольствие и не позволит масштабировать алгоритм с увеличением количества ядер в процессоре.
- Стоит ли для узкого диапазона чисел порождать несколько потоков выполнения или лучше выполнить всю операцию поиска синхронно, в единственном потоке? (Создание и уничтожение потоков выполнения – достаточно недорогая операция в Windows, но не настолько дешевая, чтобы с ее применением пытаться найти 20 первых простых чисел.)

- Как обеспечить равную загруженность всех потоков выполнения? Некоторые потоки выполнения могут завершаться раньше других, особенно те, которые обрабатывают начало диапазона. Для диапазона [100, 100 000], разделенного на четыре равные части, поток, обрабатывающий поддиапазон [100, 25 075], справится со своей работой более чем в два раза быстрее, чем поток, обрабатывающий поддиапазон [75 025, 100 000], потому что скорость нашего алгоритма проверки простых чисел падает с увеличением значений чисел.
- Как обрабатывать исключения, которые могут возникнуть в других потоках? В данном конкретном случае может показаться, что появление ошибок в методе `IsPrime` невозможно, но в реальном мире параллельной обработки данных можно найти массу примеров потенциальных ловушек и исключений. (По умолчанию среда выполнения CLR завершает процесс целиком, когда в каком-нибудь потоке выполнения появляется необработанное исключение, что в целом не так уж и плохо, но это не дает возможности программе, вызвавшей `PrimesInRange`, взять на себя обработку исключений.)

Ответить на эти вопросы очень непросто. Поэтому разработка фреймворка, не порождающего излишнего количества потоков, чтобы не вызвать перегрузку системы, обеспечивающего равномерную загруженность всех потоков выполнения, сообщающего об ошибках и возвращающего достоверные результаты, а также учитывающего другие источники параллелизма в процессе, была весьма непростой задачей для создателей библиотеки Task Parallel Library, которую мы будем использовать далее.

Естественным первым шагом от ручного управления потоками стал переход к использованию пулов потоков. Пул потоков – это компонент, управляющий группой потоков, доступных для выполнения некоторой работы. Вместо создания потока для выполнения какой-то определенной задачи, вы помещаете задание в очередь, а компонент извлекает его оттуда и передает первому свободному потоку для выполнения. Пулы потоков помогают решить некоторые проблемы, обозначенные выше, – они снижают затраты на создание и уничтожение потоков для очень коротких заданий, помогают избежать монополизации ресурсов и перегрузки системы, ограничивая общее количество потоков, используемых приложением, и автоматизируют принятие решения о выборе оптимального количества потоков для решения данной задачи.

В нашем конкретном случае можно попробовать разбить диапазон на большее число фрагментов (например, в каждой итерации цикла выделять один фрагмент) и передать их пулу потоков. Ниже приводится пример реализации этого подхода с размером фрагмента, равным 100:

```
public static IEnumerable <uint> PrimesInRange(uint start, uint end) {
    List <uint> primes = new List <uint> ();
    const uint ChunkSize = 100;
    int completed = 0;
    ManualResetEvent allDone = new ManualResetEvent(initialState: false);
    // Разделить диапазон на равные фрагменты
    uint chunks = (end - start) / ChunkSize;
    for (uint i = 0; i < chunks; ++i) {
        uint chunkStart = start + i*ChunkSize;
        uint chunkEnd = chunkStart + ChunkSize;
        ThreadPool.QueueUserWorkItem(_ => {
            for (uint number = chunkStart; number < chunkEnd; ++number) {
                if (IsPrime(number)) {
                    lock(primes) {
                        primes.Add(number);
                    }
                }
            }
            if (Interlocked.Increment(ref completed) == chunks) {
                allDone.Set();
            }
        });
    }
    allDone.WaitOne();
    return primes;
}
```

Эта версия имеет значительно больший запас масштабируемости и выполняется быстрее предыдущих версий. Она улучшила производительность с ~950 миллисекунд (для диапазона [100, 300 000]) простейшей многопоточной версии до ~800 миллисекунд (почти в 4 раза быстрее последовательной версии). Более того, использование процессора остается постоянно на высоком уровне, близком к 100%,, как видно из отчета профилировщика Concurrency Profiler, изображенного на рис. 6.4.

В версии CLR 4.0, в пул потоков было добавлено несколько компонентов поддержки кооперативной работы. Когда какой-то поток выполнения, не принадлежащий пулу (например, главный поток приложения), передает задания пулу потоков, они ставятся в глобальную очередь FIFO (First-In-First-Out – первым пришел, первым ушел). Каждый поток в составе пула имеет свою, локальную очередь LIFO (Last-In-First-Out – последним пришел, первым ушел), куда

пул помещает задания для потоков (рис. 6.5). Когда поток из пула освобождается, он обращается к своей очереди LIFO и извлекает из нее очередное задание, пока очередь не опустеет. Когда поток опустошит свою очередь LIFO, он попытается «захватить» задание из локальной очереди другого потока в порядке FIFO. Наконец, после опустошения всех локальных очередей потоки будут обращаться к глобальной очереди (FIFO) и выполнять задания оттуда.

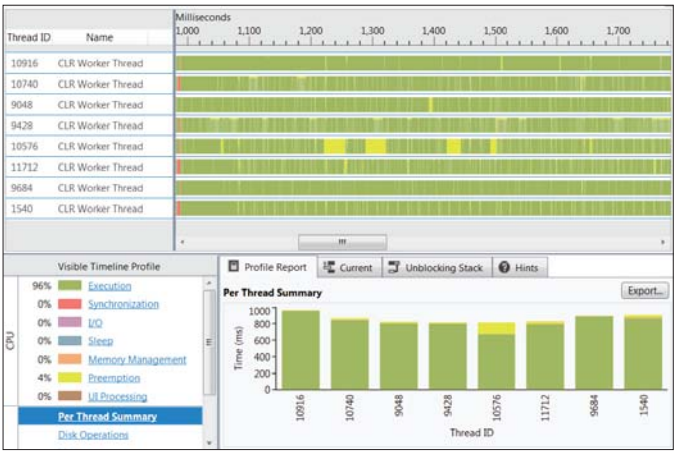

Рис. 6.4. Пул потоков в CLR содержит 8 потоков (по одному на каждое логическое ядро). Каждый поток оказался задействован практически на всем протяжении работы приложения.

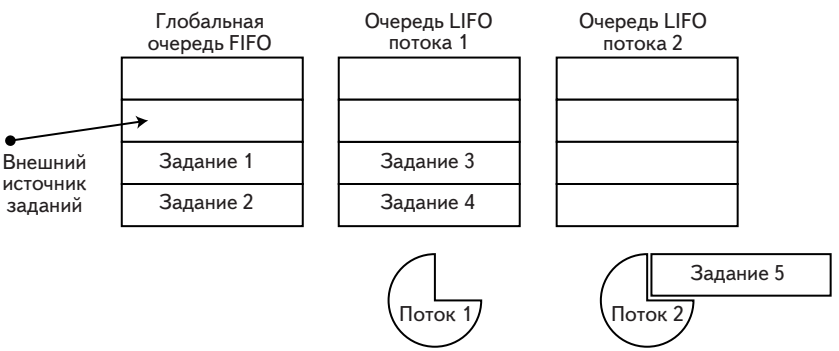

Рис. 6.5. Поток №2 в настоящий момент обрабатывает задание №5; завершив обработку, он извлечет следующее задание из глобальной очереди FIFO. Поток №1 сначала опустошит свою очередь, прежде чем приняться за другую работу.

Семантика FIFO и LIFO пула потоков

Причина странного использования очередей FIFO и LIFO заключается в следующем: когда задание добавляется в глобальную очередь, ни один из потоков не пользуется каким-либо преимущественным правом на это задание. Справедливость – единственный критерий выполнения этого задания. Именно поэтому для глобальной очереди избрана семантика FIFO. Однако, когда пул потоков перемещает задание в локальную очередь потока, весьма вероятно, что при этом будут использоваться те же данные и те же машинные инструкции, что и в текущем задании; именно поэтому для локальной очереди потока избрана семантика LIFO – очередное задание из очереди будет выполнено сразу же после текущего, что позволят максимально использовать кеши данных и инструкций.

Кроме того, доступ к заданиям в локальной очереди потока требует меньше синхронизации и вероятность конфликтов с другими потоками ниже, чем при доступе к глобальной очереди. Аналогично, когда поток захватывает задания из локальной очереди другого потока, он использует ее как очередь FIFO, чтобы поддержать оптимальное использование кешей процессора этим потоком, которое дает семантика LIFO. Такая организация пула потоков оказывается очень дружественной к заданиям с иерархической организацией, когда единственное задание, добавленное в глобальную очередь, порождает десятки дополнительных заданий, обеспечивая работой несколько потоков из пула.

Как и любая другая абстракция, пулы потоков берут на себя некоторые хлопоты по управлению жизненным циклом потоков и распределению заданий между ними, снимая это бремя с разработчика приложения. Реализация пулов потоков в CLR имеет некоторые методы, такие как `ThreadPool.SetMinThreads` и `SetMaxThreads`, позволяющие управлять количеством потоков, но в ней отсутствуют встроенные средства управления приоритетами потоков или заданий. Однако зачастую эта нехватка средств управления с лихвой компенсируется возможностью приложения автоматически масштабироваться, в зависимости от вычислительной мощности системы, и дополнительным приростом производительности из-за отсутствия необходимости создавать и уничтожать потоки для выполнения коротких заданий.

Однако механизм пула потоков с его очередью заданий не всегда удобен; задания в очереди не имеют состояния, не несут информацию об исключениях, не имеют поддержки асинхронных продолжений (continuations) и отмены (cancellation), и не предоставляют никаких механизмов для получения результатов после выполнения задания. Библиотека Task Parallel Library в .NET 4.0 вводит понятие *задачи* – мощной абстракции поверх заданий для потоков в составе

пула. Задачи являются структурированной альтернативой заданиям, почти так же, как объекты и подпрограммы стали структурированной альтернативой программированию на языке ассемблера, основанному на переходах.

Параллелизм задач

Параллелизм задач – это парадигма и набор API для разделения больших задач на более мелкие, и выполнение их с применением нескольких потоков. Библиотека Task Parallel Library (TPL) обладает превосходным API для управления миллионами задач, выполняющимися одновременно (посредством пула потоков CLR). Основу TPL составляет класс `System.Threading.Tasks.Task`, являющийся представлением задачи. Класс `Task` поддерживает следующие возможности.

- Планирование задания для выполнения некоторым независимым потоком. (Выбор конкретного потока для выполнения определенной задачи выполняется *планировщиком задач* (task scheduler); по умолчанию планировщик задач помещает задачи в очередь пула потоков CLR, но существуют и другие планировщики, передающие задачи определенным потокам, таким как поток обслуживания пользовательского интерфейса.)
- Ожидание завершения задачи и получение результатов.
- Поддержка продолжений, которые должны выполняться сразу после завершения задач. (Продолжения часто называют обратными вызовами (callbacks), но мы будем придерживаться термина *продолжения* (continuations).)
- Обработка исключений, возникающих в единственной задаче или даже в иерархии задач, в оригинальном потоке, которому данная задача была передана для выполнения, или в любом другом потоке, получающем результаты задачи.
- Отмена задач, которые еще не были запущены, и передача запросов на отмену уже выполняющихся задач.

Так как задачи можно рассматривать как высокоуровневую абстракцию, построенную поверх потоков выполнения, мы могли бы переписать пример нахождения простых чисел с использованием задач вместо потоков. Это позволило бы даже сократить код – по крайней мере, при таком подходе мы можем отказаться от счетчика завершившихся заданий и от объекта `ManualResetEvent`, следящего за выполнением заданий. Но не смотря на то, что прикладной интерфейс библиотеки TPL (с которым мы познакомимся в следующем разделе) еще

лучше подходит для распараллеливания цикла поиска простых чисел в заданном диапазоне, мы обратимся к другой проблеме.

Существует хорошо известный рекурсивный алгоритм быстрой сортировки QuickSort, легко поддающийся распараллеливанию (и имеющий среднюю производительность $O(n\log(n))$ и достаточно оптимальный – хотя можно сосчитать по пальцам все современные крупные фреймворки, использующие алгоритм QuickSort для сортировки чего бы то ни было). Алгоритм QuickSort имеет следующую реализацию:

```
public static void QuickSort <T> (T[] items) where T : IComparable <T> {
    QuickSort(items, 0, items.Length);
}

private static void QuickSort <T> (T[] items, int left, int right)
                                   where T : IComparable <T> {
    if (left == right) return;
    int pivot = Partition(items, left, right);
    QuickSort(items, left, pivot);
    QuickSort(items, pivot + 1, right);
}

private static int Partition <T> (T[] items, int left, int right)
                                 where T : IComparable <T> {
    int pivotPos = ...; // часто используется случайный индекс
    T pivotValue = items[pivotPos];
    Swap(ref items[right-1], ref items[pivotPos]);
    int store = left;
    for (int i = left; i < right - 1; ++i) {
        if (items[i].CompareTo(pivotValue) < 0) {
            Swap(ref items[i], ref items[store]);
            ++store;
        }
    }
    Swap(ref items[right-1], ref items[store]);
    return store;
}

private static void Swap < T > (ref T a, ref T b) {
    T temp = a;
    a = b;
    b = temp;
}
```

Схема на рис. 6.6 иллюстрирует единственный шаг, выполняемый методом `Partition`. В качестве точки опоры выбирается четвертый элемент (со значением 5). Сначала он перемещается в правый конец массива. Затем все элементы, больше точки опоры, сдвигаются впра-

во. Наконец, точка опоры позиционируется так, чтобы справа от нее оказались элементы, которые больше ее, а слева – которые меньше или равны.

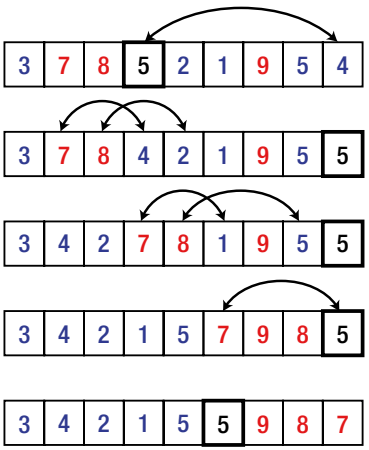

Рис. 6.6. Иллюстрация единственного вызова метода Partition.

Рекурсивные вызовы `QuickSort`, выполняемые на каждом шаге, могут служить точками распараллеливания задачи. Сортировка левой и правой частей массива выполняются независимо и не требуют синхронизации, и класс `Task` идеально подходит для этой цели. Ниже представлена первая попытка реализовать параллельную версию `QuickSort` с применением класса `Task`:

```
public static void QuickSort <T> (T[] items) where T : IComparable <T> {
    QuickSort(items, 0, items.Length);
}

private static void QuickSort <T> (T[] items, int left, int right)
where T : IComparable <T> {
    if (right - left < 2) return;
    int pivot = Partition(items, left, right);
    Task leftTask = Task.Run(() => QuickSort(items, left, pivot));
    Task rightTask = Task.Run(() => QuickSort(items, pivot + 1, right));
    Task.WaitAll(leftTask, rightTask);
}

private static int Partition <T> (T[] items, int left, int right)
where T : IComparable <T> {
    // Реализация опущена для экономии места
}
```

Метод `Task.Run` создает новую задачу (действует подобно вызову `new Task()`) и планирует ее на выполнение (подобно методу `Start` вновь созданной задачи). Статический метод `Task.WaitAll` ждет завершения обеих задач и затем возвращает управление. Обратите внимание, что нам не пришлось определять, как ждать завершения задач, когда создавать потоки и когда уничтожать их.

Существует один очень удобный вспомогательный метод с именем `Parallel.Invoke`, который выполняет указанный набор задач и возвращает управление, когда все задачи будут выполнены. Его применение позволило бы переписать ядро метода `QuickSort`, как показано ниже:

```
Parallel.Invoke(
    () => QuickSort(items, left, pivot),
    () => QuickSort(items, pivot + 1, right)
);
```

Неважно, какую версию выбрать, использующую `Parallel.Invoke` или создающую задачи вручную, если сравнить ее с последовательной версией, обнаружится, что она работает существенно медленнее, даже при том, что в ее распоряжении все доступные процессоры. И действительно, последовательная версия выполняет сортировку массива с 1 000 000 случайных целых чисел (на нашей тестовой системе) примерно за 250 миллисекунд, а параллельная версия – примерно за 650 миллисекунд!

Проблема в том, что параллельно выполняемые задачи должны быть достаточно объемными; бессмысленно пытаться распараллелить сортировку массива из трех элементов, потому что накладные расходы на создание объектов `Task`, планирование отдельных заданий и ожидание их выполнения оказываются значительно больше расходов на несколько операций сравнения.

Ограничение параллелизма в рекурсивных алгоритмах

Можно ли как-то ограничить применение параллелизма, чтобы предотвратить подобные потери? Существует несколько решений этой проблемы.

- Использовать параллельную версию, пока размер сортируемого массива не станет меньше некоторого порогового значения (например, 500 элементов), и затем переключаться на последовательную версию.

- Использовать параллельную версию, пока глубина рекурсии не превысит некоторого порогового значения, и затем переключаться на последовательную версию. (Этот вариант является частным случаем предыдущего, когда опорная точка всегда выбирается точно в середине массива.)
- Использовать параллельную версию, пока количество созданных задач (для этого придется организовать поддержку счетчика задач вручную) не превысит некоторого порогового значения, и затем переключаться на последовательную версию. (Это единственно возможный вариант, когда нет других критериев ограничения параллелизма, таких как глубина рекурсии или размер массива.)

И действительно, применение первого варианта ограничения параллелизма для массивов с числом элементов меньше 500 дает превосходные результаты. В системе автора на процессоре Intel i7 было получено 4-кратное увеличение производительности, в сравнении с последовательной версией. Чтобы добиться таких результатов потребовалось внести достаточно простые изменения в код, но имейте в виду, что в окончательной версии лучше не использовать жестко заданное пороговое значение:

```
private static void QuickSort <T> (T[] items, int left, int right)
                                   where T : IComparable <T> {
    if (right - left < 2) return;
    int pivot = Partition(items, left, right);
    if (right - left > 500) {
        Parallel.Invoke(
             () => QuickSort(items, left, pivot),
             () => QuickSort(items, pivot + 1, right)
        );
    } else {
        QuickSort(items, left, pivot);
        QuickSort(items, pivot + 1, right);
    }
}
```

Примеры рекурсивной декомпозиции

Существует множество других способов распараллеливания подобных рекурсивных алгоритмов. В действительности, почти все рекурсивные алгоритмы, разбивающие исходные данные на части, *могут* обрабатывать эти части независимо и объединять результаты. Далее в этой главе мы рассмотрим примеры, которые сложнее поддаются распараллеливанию, но сначала познакомимся с самими алгоритмами.

- Алгоритм Штрассена (Strassen) умножения матриц (http://ru.wikipedia.org/wiki/Алгоритм_Штрассена). Этот алгоритм умножения матриц обеспечивает лучшую производительность, чем обычный кубический алгоритм, который будет рассматриваться ниже в этой главе. Алгоритм Штрассена рекурсивно разлагает матрицу размера $2^n \times 2^n$ на четыре одинаковых матрицы размером $2^{n-1} \times 2^{n-1}$ и использует хитрый трюк с применением *семи* умножений вместо восьми, чтобы получить асимптотическое время выполнения $\sim O(n^{2.807})$. Как и в примере быстрой сортировки, практические реализации алгоритма Штрассена часто возвращаются к стандартному кубическому алгоритму для достаточно маленьких матриц; при распараллеливании алгоритма Штрассена с использованием приема рекурсивной декомпозиции, определение порогового значения приобретает особую важность.
- Быстрое преобразование Фурье (алгоритм Кули-Тьюки (Cooley-Tukey), http://en.wikipedia.org/wiki/Cooley%E2%80%93Tukey_FFT_algorithm). Этот алгоритм вычисляет ДПФ (дискретное преобразование Фурье) вектора с длиной 2^n с использованием рекурсивной декомпозиции вектора на два подвектора размером 2^{n-1}. Организовать параллельное выполнение этих вычислений очень просто, но здесь снова очень важно определить порогового значения, чтобы исключить параллельную обработку слишком маленьких векторов.
- Обход графа (поиск в глубину или поиск в ширину). Как было показано в главе 4, сборщик мусора CLR выполняет обход графа, в котором объекты являются вершинами, а ссылки между ними – ребрами. Обход графа в глубину или в ширину можно значительно ускорить за счет распараллеливания, как и многие другие рекурсивные алгоритмы; однако, в отличие от быстрой сортировки или быстрого преобразования Фурье, при распараллеливании обхода ветвей графа заранее сложно предсказать объем работы, который будет выполнен рекурсивным вызовом. Эта особенность требует применения некоторых эвристик при разделении пространства поиска между несколькими потоками: в главе 4 мы видели, что серверная разновидность сборщика мусора использует довольно грубое разбиение, опираясь на различные области динамической памяти, используемые разными процессорами для размещения объектов.

Желающие попрактиковаться в параллельном программировании могут также обратить внимание на алгоритм умножения Карацубы, опирающийся на рекурсивную декомпозицию при умножении n-значных чисел, и имеющий сложность $\sim O(n^{1.585})$; на алгоритм сортировки слиянием, опирающийся на рекурсивную декомпозицию при сортировке, подобно алгоритму быстрой сортировки; и многочисленные алгоритмы динамического программирования, которые часто требуют дополнительных ухищрений, таких как мемоизация, в разных ветвях параллельных вычислений (один из примеров будет представлен ниже).

Исключения и отмена

Мы пока познакомились не со всеми возможностями класса `Task`. Представьте, что нам потребовалось реализовать обработку исключений, которые могут возникать в рекурсивных вызовах `QuickSort`, и добавить поддержку отмены всей операции сортировки, если она еще не завершилась.

Среда выполнения задачи обеспечивает все необходимое для передачи исключений, возникающих в задаче, обратно любому потоку выполнения, желающему принять их. Представьте, что в одном из рекурсивных вызовов `QuickSort` в задаче возникло исключение, возможно потому, что мы были невнимательны при назначении границ массива и спровоцировали ошибку выхода за границы массива. В этом случае исключение возникнет в потоке из пула, над которым у нас нет явного контроля, и мы не можем переопределить порядок обработки исключений в нем. К счастью TPL автоматически перехватит исключение и сохранит его внутри объекта `Task` для последующей передачи дальше.

Исключение, возникшее в задаче, будет возбуждено повторно (завернутое в объект `AggregateException`), когда программа попытается перейти в режим ожидания завершения задачи (вызвав метод `Task.Wait` экземпляра) или получить ее результаты (обратившись к свойству `Task.Result`). Это дает возможность организовать автоматическую и централизованную обработку исключений в коде, создавшем задачу, и не требует передачи ошибок вручную или использования инструментов синхронизации. Следующий небольшой пример демонстрирует парадигму обработки исключений, реализованную в TPL:

```
int i = 0;
Task <int> divideTask = Task.Run(() = > { return 5/i; });
```

```
try {
    Console.WriteLine(divideTask.Result);  // обращение к свойству Result
} catch (AggregateException ex) {          // возбудит исключение
    foreach (Exception inner in ex.InnerExceptions) {
        Console.WriteLine(inner.Message);
    }
}
```

Примечание. *При создании одной задачи внутри другой, отношение между ними устанавливается с помощью значения* `TaskCreationOptions.AttachedToParent`. *Далее в этой главе будет показано, как отношение родитель-потомок влияет на отмену, продолжения и отладку задач. В продолжение темы обработки исключений, следует заметить, что под ожиданием завершения родительской задачи подразумевается ожидание завершения всех ее дочерних задач, в ходе которого все исключения, возникшие в дочерних задачах, будут переданы родительской задаче. Именно поэтому TPL возбуждает экземпляр исключения* `AggregateException`, *содержащий иерархию исключений, которая может быть порождена иерархией задач.*

Отмена выполняющихся заданий – еще одна важная тема. Представьте, что имеется иерархия задач, как, например, иерархия, которая могла бы быть создана методом `QuickSort` с помощью значения `TaskCreationOptions.AttachedToParent`. Даже при том, что одновременно могут выполняться сотни задач, у нас может появиться необходимость дать пользователю возможность отмены, например, если надобность в отсортированных данных отпала. В других случаях возможность отмены задания может быть неотъемлемой частью процесса выполнения задачи. Например, представьте параллельный алгоритм, выполняющий поиск узлов в графе в глубину или в ширину. Когда желаемый узел обнаруживается, всю иерархию выполняющихся задач поиска необходимо остановить.

Отмена задач тесно связана с использованием типов `CancellationTokenSource` и `CancellationToken`, и требует содействия со стороны отменяемой задачи. Иными словами, если задача уже выполняется, ее нельзя просто грубо оборвать с использованием механизмов отмены в библиотеке TPL. Для отмены уже выполняющегося задания требуется содействие со стороны кода, выполняющего это задание. Однако еще не запущенную задачу можно отменить немедленно, без каких-либо отрицательных последствий.

Следующий код реализует поиск в двоичном дереве, каждый узел которого содержит потенциально длинный массив, элементы которого нужно обойти; вся процедура поиска может быть отменена вызывающей программой с использованием механизмов отмены TPL.

С одной стороны, незапущенные задачи могут отменяться автоматически, самой библиотекой TPL; с другой стороны, уже запущенные задачи будут периодически проверять наличие признака отмены с инструкциями и содействовать прекращению работы, когда это необходимо.

```
public class TreeNode <T> {
    public TreeNode <T> Left, Right;
    public T[] Data;
}

public static void TreeLookup <T> (
    TreeNode <T> root, Predicate <T> condition, CancellationTokenSource cts) {
    if (root == null) {
        return;
    }
    // Запустить рекурсивные задачи, передать им признак отмены,
    // чтобы они могли останавливаться автоматически, пока не
    // запущены, и реагировать на запрос отмены в противном случае
    Task.Run(() => TreeLookup(root.Left, condition, cts), cts.Token);
    Task.Run(() => TreeLookup(root.Right, condition, cts), cts.Token);
    foreach (T element in root.Data) {
        if (cts.IsCancellationRequested) break; // удовлетворить запрос
        if (condition(element)) {
            cts.Cancel();  // отменить все запущенные задания
            // Выполнить требуемые операции с элементом element
        }
    }
}

// Пример вызывающего кода:
CancellationTokenSource cts = new CancellationTokenSource();
Task.Run(() = > TreeLookup(treeRoot, i = > i % 77 == 0, cts);
// Спустя некоторое время, например, когда пользователя перестали
// интересовать результаты операции:
cts.Cancel();
```

В вашей практике вам неизбежно будут встречаться алгоритмы, для реализации которых желательно иметь более простой способ их распараллеливания. Вспомните пример поиска простых чисел, с которого началось обсуждение этой темы. Мы могли бы вручную разбить диапазон на поддиапазоны, создать для каждого из них отдельную задачу и ждать, когда они завершатся. В действительности существует целое семейство алгоритмов обработки массивов данных, применяющих к ним определенные операции. Такие алгоритмы требуют еще более высокоуровневой абстракции, чем параллельные задачи. Давайте познакомимся с этой абстракцией.

Параллелизм данных

Парадигма параллелизма задач в первую очередь относится к задачам. Основной целью парадигмы параллелизма данных является полное устранение задач из поля зрения и их замена высокоуровневой абстракцией – параллельными циклами. Иными словами, распараллеливается не *реализация* алгоритма, а *данные*, которыми она оперирует. Библиотека Task Parallel Library предлагает несколько вариантов поддержки параллелизма данных.

Parallel.For и Parallel.ForEach

Циклы `for` и `foreach` часто являются отличными кандидатами для распараллеливания. В действительности, еще на заре развития параллельных вычислений предпринимались попытки автоматического распараллеливания таких циклов. Некоторые попытки воплотились в языковые конструкции или расширения, такие как стандарт OpenMP (описывающий такие директивы, как `#pragma omp parallel for`, обеспечивающую распараллеливание циклов `for`). Библиотека Task Parallel Library предоставляет поддержку распараллеливания циклов посредством явных методов, очень близких своим языковым эквивалентам. Речь идет о методах `Parallel.For` и `Parallel.ForEach`, максимально близко имитирующих поведение циклов `for` и `foreach`.

В примере поиска простых чисел у нас имелся цикл, который выполнял итерации по большому диапазону чисел, проверял каждое из них на принадлежность к категории простых чисел и добавлял их в коллекцию, как показано ниже:

```
for (int number = start; number < end; ++number) {
    if (IsPrime(number)) {
        primes.Add(number);
    }
}
```

Преобразование этого кода, с целью задействовать в нем метод `Parallel.For`, – по большей степени механическая задача, если не считать необходимость синхронизации доступа к коллекции простых чисел (существуют более удачные решения, такие как агрегирование, которые мы рассмотрим ниже):

```
Parallel.For(start, end, number => {
    if (IsPrime(number)) {
        lock(primes) {
```

```
            primes.Add(number);
        }
    }
});
```

Заменив языковую инструкцию цикла вызовом метода, мы автоматически обеспечили параллельное выполнение итераций цикла. Кроме того, метод `Parallel.For` — это не простой цикл, генерирующий задачи в каждой итерации или для каждого фрагмента данных определенного размера. Вместо этого `Parallel.For` не спеша приспосабливается к темпу выполнения отдельных итераций, учитывая количество задач, выполняющихся в каждый момент, и исключает вероятность дробления диапазона на слишком мелкие фрагменты, производя его деление динамически. Реализовать подобное поведение вручную весьма непросто, но вам доступны некоторые настройки (такие как управление максимальным количеством выполняемых задач), которые можно выполнить с помощью перегруженной версии метода `Parallel.For`, принимающей объект `ParallelOptions`, или используя собственный метод, выполняющий деление диапазона между задачами.

Существует похожий метод и для цикла `foreach`, который с успехом можно использовать, когда объем источника данных заранее неизвестен и может даже оказаться бесконечным. Представьте, что нам потребовалось загрузить из Интернета множество полос RSS, объявленных как `IEnumerable<string>`. В этом случае общая структура цикла могла бы иметь следующий вид:

```
IEnumerable <string> rssFeeds = ...;
WebClient webClient = new WebClient();
foreach (string url in rssFeeds) {
    Process(webClient.DownloadString(url));
}
```

Данный цикл легко можно распараллелить механической заменой инструкции `foreach` вызовом метода `Parallel.ForEach`. Обратите внимание, что источник данных (коллекция `rssFeeds`) необязательно должен быть потокобезопасным, потому что `Parallel.ForEach` автоматически синхронизирует операции обращения к нему из разных потоков.

```
IEnumerable <string> rssFeeds = ...; // не должен быть потокобезопасным
WebClient webClient = new WebClient();
Parallel.ForEach(rssFeeds, url => {
    Process(webClient.DownloadString(url));
});
```

Примечание. *Кого-то может обеспокоить упоминание бесконечных источников данных. Однако, как оказывается, подобные решения, предусматривающие возможность прерывания цикла по достижении некоторого условия, часто используются на практике. Например, представьте бесконечный источник данных, такой как множество всех натуральных чисел (в коде его можно выразить в виде метода, возвращающего* `IEnumerable<BigInteger>`*). Мы могли бы написать параллельный цикл, осуществляющий поиск числа, сумма цифр которого равна 477, но не кратного числу 133. Надеемся, что такое число существует, и наш цикл рано или поздно завершится.*

Однако, распараллелить цикл совсем не просто, как могло бы показаться из предыдущего обсуждения. Существует ряд «недостающих» особенностей, которые необходимо рассмотреть, прежде чем подвесить этот инструмент на пояс. Для начинающих отметим, что в языке C# существует ключевое слово `break`, которое может вызывать преждевременное завершение циклов. Но как завершить цикл, параллельно выполняемый несколькими потоками, когда мы даже не знаем, какая итерация в данный момент выполняется в других потоках?

Класс `ParallelLoopState` представляет состояние параллельного цикла и позволяет прервать его. Например:

```
int invitedToParty = 0;
Parallel.ForEach(customers, (customer, loopState) = > {
    if (customer.Orders.Count > 10 && customer.City == "Portland") {
        if (Interlocked.Increment(ref invitedToParty) > = 25) {
            loopState.Stop(); // прервать выполнение итераций
        }
    }
});
```

Обратите внимание: метод `Stop` не гарантирует, что итерация, вызвавшая его, будет последней – итерации, уже запущенные к этому моменту, будут выполнены до конца (если они не проверяют свойство `ParallelLoopState.ShouldExitCurrentIteration`). Но гарантирует, что никакие другие итерации не будут запланированы на выполнение.

Одним из недостатков метода `ParallelLoopState.Stop` является отсутствие гарантий, что все итерации, предшествующие данной, будут выполнены. Например, при обработке списка из 1000 заказчиков таким способом может получиться так, что заказчики с 1 по 100 будут обработаны полностью, заказчики с 101 по 110 вообще не будут обработаны, и заказчик 111 окажется последним обработанным перед вызовом `Stop`. Если необходимо обеспечить выполнение всех итераций, предшествующих данной (даже если они еще не были запущены!), следует использовать метод `ParallelLoopState.Break`.

Parallel LINQ (PLINQ)

Пожалуй, самой высокоуровневой абстракцией параллельных вычислений является возможность объявить: «Я хочу, чтобы этот фрагмент кода выполнялся параллельно», – и переложить все хлопоты на используемый фреймворк. Именно это позволяет фреймворк Parallel LINQ. Но сначала вспомним, что такое LINQ. LINQ (Language INtegrated Query – язык интегрированных запросов) – это фреймворк и набор расширений языка, появившийся в версии C# 3.0 и .NET 3.5, стирающий грань между императивным и декларативным программированием там, где требуется выполнять итерации через данные. Например, следующий запрос LINQ извлекает из источника `customers` – который может быть обычной коллекцией в памяти, таблицей в базе данных или иметь более экзотическое происхождение – имена и возраст клиентов, проживающих в штате Вашингтоне (Washington), сделавших не менее трех покупок на сумму более $10 за последние десять месяцев, и выводит эти данные в консоль:

```
var results = from customer in customers
              where customer.State == "WA"
              let custOrders = (
                      from order in orders
                      where customer.ID == order.ID
                      select new { order.Date, order.Amount })
              where custOrders.Count(
                      co => co.Amount >= 10 &&
                      co.Date >= DateTime.Now.AddMonths(-10)) >= 3
              select new { customer.Name, customer.Age };
foreach (var result in results) {
    Console.WriteLine("{0} {1}", result.Name, result.Age);
}
```

Первое, на что следует обратить внимание, – большая часть запроса определена декларативно, подобно запросу на языке SQL. Здесь не используются циклы для фильтрации объектов или группировки объектов из разных источников. Часто вам не придется даже волноваться о синхронизации итераций, выполняемых запросом, потому что запросы LINQ являются исключительно функциональными и не имеют побочных эффектов – они преобразуют одну коллекцию (`IEnumerable<T>`) в другую, не изменяя никакие объекты в процессе работы.

Чтобы распараллелить запрос, представленный выше, достаточно лишь изменить тип коллекции источника с обобщенного `IEnumerable<T>` на `ParallelQuery<T>`. Для этого можно воспользоваться методом `AsParallel` расширения и получить в результате следующий элегантный код:

```
var results = from customer in customers.AsParallel()
              where customer.State == "WA"
              let custOrders = (
                            from order in orders
                            where customer.ID == order.ID
                            select new { order.Date, order.Amount })
              where custOrders.Count(
                            co => co.Amount >= 10 &&
                            co.Date > = DateTime.Now.AddMonths(-10)) >= 3
              select new { customer.Name, customer.Age };
foreach (var result in results) {
    Console.WriteLine("{0} {1}", result.Name, result.Age);
}
```

Параллельная обработка запросов выполняется фреймворком PLINQ в три этапа, как показано на рис. 6.7. Сначала PLINQ решает, сколько потоков следует использовать для выполнения запроса. Затем рабочие потоки извлекают свои фрагменты их исходной коллекции, под защитой блокировок. Все потоки выполняют свои задания независимо и помещают результаты в свои локальные очереди. В заключение, локальные результаты объединяются в единую коллекцию, которая подается в цикл `foreach`, в примере выше.

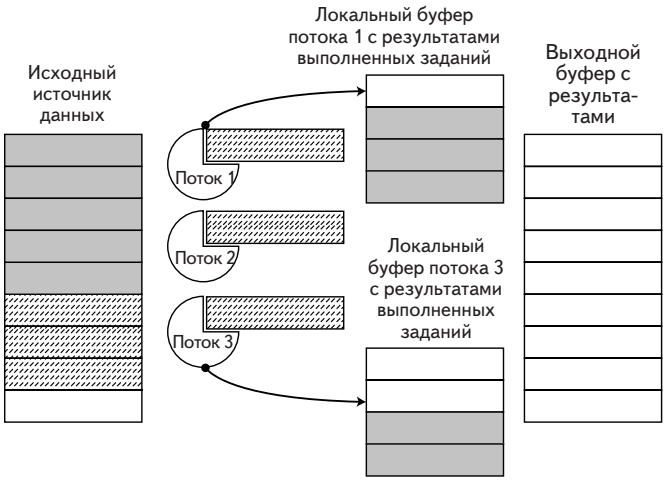

Рис. 6.7. Параллельная обработка запроса в PLINQ. Серые прямоугольники представляют результаты завершенных заданий? помещенные в локальные очереди потоков, откуда они последовательно перемещаются в общий выходной буфер, доступный вызывающей программе. Заштрихованные прямоугольники представляют задания, выполняющиеся в данный момент.

Главное преимущество PLINQ перед методом `Parallel.ForEach` заключается в автоматическом объединении результатов, полученных разными потоками. В примере поиска простых чисел с использованием `Parallel.ForEach` мы были вынуждены вручную добавлять результаты работы каждого потока в глобальную коллекцию (далее в этой главе мы рассмотрим способ оптимизации, использующий прием агрегирования). При этом необходимо было использовать механизм синхронизации и тем самым увеличивать накладные расходы. Тот же результат легко можно получить с помощью PLINQ:

```
List<int> primes = (from n in Enumerable.Range(3, 200000).AsParallel()
                    where IsPrime(n)
                      select n).ToList();
// Вместо Enumerable.Range(...).AsParallel() можно использовать
// ParallelEnumerable.Range
```

Настройка параллельных циклов и PLINQ

Параллельные циклы (`Parallel.For` и `Parallel.ForEach`) и PLINQ поддерживают несколько методов для выполнения настройки, которые делают эти инструменты чрезвычайно гибкими и близкими в богатстве и выразительности к механизму параллельных задач, обсуждавшемуся выше. Методы параллельных циклов принимают объект `ParallelOptions` с различными свойствами, определяющими дополнительные параметры, а фреймворк PLINQ – дополнительные методы объектов `ParallelQuery<T>`. В число настраиваемых параметров входят:

- ограничение степени параллелизма (максимально возможное количество задач, выполняемых параллельно);
- передача признака отмены для остановки параллельных задач;
- принудительное определение порядка получения результатов параллельных запросов;
- управление буферизацией вывода (режимом слияния) результатов параллельных запросов.

При использовании параллельных циклов чаще всего ограничивают максимально возможное количество задач, выполняемых одновременно, тогда как при использовании PLINQ обычно настраивают режим слияния результатов и порядок их вывода. За дополнительной информацией по этим параметрам настройки обращайтесь к документации на сайте MSDN.

Асинхронные методы в C# 5

До сих пор мы рассматривали приемы распараллеливания, которые могут быть выражены с использованием классов и методов из библиотеки Task Parallel Library. Однако существует еще одна среда параллельных вычислений, основанная на расширениях языка и поз-

воляющая получить еще большую выразительность там, где методы выглядят несколько неуклюже или недостаточно выразительно. В этом разделе мы познакомимся с нововведениями в языке C# 5, предназначенными для решения задач параллельного программирования и упрощающими реализацию продолжений (continuations). Но сначала познакомимся с продолжениями с точки зрения асинхронного выполнения.

Достаточно часто возникает необходимость связать с некоторой задачей *продолжение* (continuation), или обратный вызов (callback), то есть некоторый код, который должен быть выполнен по завершении задачи. Имея контроль над задачей – то есть, когда вы явно управляете ее запуском – вы можете встроить обратный вызов в саму задачу, но когда вы получаете задачу от какого-то другого метода, необходимо использовать явный API продолжения. Библиотека TPL предлагает метод экземпляра `ContinueWith` и статические методы `ContinueWhenAll`/`ContinueWhenAny` (их имена говорят сами за себя) для управления продолжениями в некоторых ситуациях. Используя класс `TaskScheduler` можно запланировать выполнение продолжения только при определенных обстоятельствах (например, только когда задача завершилась успешно или только когда в задаче возникло исключение) и в определенном потоке (группе потоков). Ниже приводится несколько примеров использования различных методов:

```
Task < string > weatherTask = DownloadWeatherInfoAsync(...);
weatherTask.ContinueWith(_ => DisplayWeather(weatherTask.Result),
                         TaskScheduler.Current);
Task left = ProcessLeftPart(...);
Task right = ProcessRightPart(...);
TaskFactory.ContinueWhenAll(
    new Task[] { left, right },
    CleanupResources
);
TaskFactory.ContinueWhenAny(
    new Task[] { left, right },
    HandleError,
    TaskContinuationOptions.OnlyOnFaulted
);
```

Продолжения – удобный способ программирования асинхронных приложений и очень ценный, при выполнении асинхронных операций ввода/вывода в приложениях с графическим интерфейсом. Например, чтобы обеспечить высокую отзывчивость приложений для Windows 8 с интерфейсом в стиле Метро (Metro), WinRT (Windows Runtime) API в Windows 8 поддерживает только асинхронные вер-

сии всех операций, длительность которых может составить больше 50 миллисекунд. При наличии множества асинхронных вызовов, выполняемых друг за другом, вложенные продолжения могут стать неудобными в использовании, как демонстрирует следующий пример:

```
// Синхронная версия:
private void updateButton_Clicked(...) {
    using (LocationService location = new LocationService())
    using (WeatherService weather   = new WeatherService()) {
        Location loc      = location.GetCurrentLocation();
        Forecast forecast = weather.GetForecast(loc.City);
        MessageDialog msg = new MessageDialog(forecast.Summary);
        msg.Display();
    }
}

// Асинхронная версия:
private void updateButton_Clicked(...) {
    TaskScheduler uiScheduler = TaskScheduler.Current;
    LocationService location = new LocationService();
    Task<Location> locTask = location.GetCurrentLocationAsync();
    locTask.ContinueWith(_ => {
        WeatherService weather = new WeatherService();
        Task<Forecast> forTask = weather.GetForecastAsync(
                                    locTask.Result.City);
        forTask.ContinueWith(__   => {
            MessageDialog message = new MessageDialog(forTask.
Result.Summary);
            Task msgTask = message.DisplayAsync();
            msgTask.ContinueWith(___ => {
                weather.Dispose();
                location.Dispose();
            });
        }, uiScheduler);
    });
}
```

Глубокая вложенность – не единственная проблема программирования на основе продолжений. Взгляните на следующий синхронный цикл, который требуется преобразовать в асинхронную версию:

```
// Синхронная версия:
private Forecast[] GetForecastForAllCities(City[] cities) {
    Forecast[] forecasts = new Forecast[cities.Length];
    using (WeatherService weather = new WeatherService()) {
        for (int i = 0; i < cities.Length; ++i) {
            forecasts[i] = weather.GetForecast(cities[i]);
        }
    }
    return forecasts;
```

```
}

// Асинхронная версия:
private Task < Forecast[] > GetForecastsForAllCitiesAsync(City[] cities) {
    if (cities.Length == 0) {
        return Task.Run(() = > new Forecast[0]);
    }
    WeatherService weather = new WeatherService();
    Forecast[] forecasts = new Forecast[cities.Length];
    return GetForecastHelper(weather, 0, cities, forecasts).ContinueWith(
       _ => forecasts);
}

private Task GetForecastHelper(
    WeatherService weather, int i, City[] cities, Forecast[] forecasts) {
    if (i >= cities.Length) return Task.Run(() => { });
    Task < Forecast > forecast = weather.GetForecastAsync(cities[i]);
    forecast.ContinueWith(task => {
        forecasts[i] = task.Result;
        GetForecastHelper(weather, i + 1, cities, forecasts);
    });
    return forecast;
}
```

Чтобы преобразовать этот цикл, пришлось полностью переписать оригинальный метод и запланировать продолжение, которое по сути выполняет следующую итерацию весьма неочевидным, рекурсивным способом. Архитекторы C# 5 решили устранить эту проблему, введением в синтаксис языка двух новых ключевых слов, `async` и `await`.

Асинхронный метод должен быть отмечен ключевым словом `async` и может возвращать значение типа `void`, `Task` или `Task<T>`. Внутри асинхронного метода можно использовать оператор `await`, чтобы реализовать продолжение без использования метода `ContinueWith`. Взгляните на следующий пример:

```
private async void updateButton_Clicked(...) {
    using (LocationService location = new LocationService()) {
        Task<Location> locTask = location.GetCurrentLocationAsync();
        Location loc = await locTask;
        cityTextBox.Text = loc.City.Name;
    }
}
```

Здесь выражение `await locTask` реализует продолжение для задачи, возвращаемой вызовом `GetCurrentLocationAsync`. Собственно продолжением является оставшаяся часть тела метода (начиная с инструкции присваивания переменной `loc`), а значением выражения `await` является результат выполнения задачи, в данном случае – объ-

ект `Location`. Кроме того, продолжение неявно планируется для выполнения в потоке управления пользовательским интерфейсом, о чем, при использовании `TaskScheduler`, необходимо было позаботиться явно.

Заботу обо всех синтаксических особенностях тела метода берет на себя компилятор C#. Например, в только что написанном методе имеется блок `try...finally`, спрятанный под покровом инструкции `using`. Компилятор переделает продолжение так, что метод `Dispose` переменной `location` будет вызван независимо от успешности завершения задачи.

Эта особенность делает замену вызовов синхронных методов их асинхронными аналогами почти тривиальным делом. Компилятор поддерживает обработку исключений, сложные циклы, рекурсивные вызовы методов – языковые конструкции, которые *плохо* сочетаются с явным механизмом продолжений. Например, ниже приводится асинхронная версия цикла, доставившего нам неприятности выше:

```
private async Task <Forecast[]> GetForecastForAllCitiesAsync(City[]
cities) {
    Forecast[] forecasts = new Forecast[cities.Length];
    using (WeatherService weather = new WeatherService()) {
        for (int i = 0; i < cities.Length; ++i) {
            forecasts[i] = await weather.GetForecastAsync(cities[i]);
        }
    }
    return forecasts;
}
```

Обратите внимание, что изменения оказались минимальными – всю заботу о переменной `forecasts` (типа `Forecast[]`), возвращаемой нашим методом, и о создании `Task<Forecast[]>` для ее поддержки взял на себя компилятор.

Всего лишь две языковые особенности (не очень сложные в реализации!) существенно снизили порог вхождения в асинхронное программирование и упростили работу с методами, возвращающими задачи и управляющими ими. Кроме того, реализация оператора `await` несовместима с библиотекой Task Parallel Library; низкоуровневый WinRT API в Windows 8 возвращает экземпляры типа `IAsyncOperation<T>`, а не задачи `Task` (которые являются управляемой концепцией), которые, тем не менее, с успехом можно передавать оператору `await`, как в следующем примере, использующем WinRT API:

```
using Windows.Devices.Geolocation;
...
```

```
private async void updateButton_Clicked(. . .) {
    Geolocator locator = new Geolocator();
    Geoposition position = await locator.GetGeopositionAsync();
    statusTextBox.Text = position.CivicAddress.ToString();
}
```

Дополнительные шаблоны в TPL

До сих пор в этой главе мы рассматривали довольно простые примеры алгоритмов, легко поддающихся распараллеливанию. В этом разделе мы коротко исследуем несколько дополнительных приемов, которые могут пригодиться вам при решении настоящих проблем; в некоторых случаях мы можем обеспечить прирост производительности в самых неожиданных местах.

Первым приемом оптимизации, который может использоваться при распараллеливании циклов с общим состоянием, является *агрегирование* (иногда называется *сверткой* (reduction)). Когда в параллельном цикле используется общее состояние, масштабируемость часто утрачивается из-за необходимости синхронизировать доступ к общим данным; чем больше ядер в процессоре оказывается доступно, тем меньше выигрыш из-за синхронизации (этот эффект является прямым следствием закона Амдала (Amdahl Law), который часто называют *законом убывающей отдачи* (The Law of Diminishing Returns)). Значительный прирост производительности часто достигается за счет создания локальных состояний потоков или задач, выполняющих параллельные итерации цикла, и их объединения в конце. Методы из библиотеки TPL, используемые для организации циклов, имеют перегруженные версии, обслуживающие такого рода локальные агрегаты.

Вернемся к примеру поиска простых чисел, реализованному выше. Одной из основных помех масштабированию в нем является необходимость добавления новых обнаруженных простых чисел в совместно используемый список, для чего требуется использовать механизм синхронизации. Вместо этого мы можем использовать в каждом потоке свой, локальный список и объединить их по завершении цикла:

```
List<int> primes = new List<int> ();
Parallel.For(3, 200000,
    () => new List < int > (),  // инициализировать локальную копию
    (i, pls, localPrimes) => {  // каждый шаг вычислений
        if (IsPrime(i)) {       // возвращает новое локальное
                                // состояние
            localPrimes.Add(i); // синхронизация не требуется
        }
```

```
        return localPrimes;
    },
    localPrimes => {   // объединить локальные списки
        lock(primes) { // синхронизация необходима
            primes.AddRange(localPrimes);
        }
    }
);
```

В примере выше количество попыток приобрести блокировку значительно меньше, чем в предыдущих примерах – блокировку требуется приобрести лишь один раз для каждого потока, а не для каждого найденного простого числа. Мы добавили накладные расходы на объединение списков, но эта цена незначительна, в сравнении с увеличившейся масштабируемостью.

Еще одно место, где можно применить оптимизацию, – итерации цикла, слишком короткие, чтобы их эффективно можно было распараллелить. Даже при том, что механизм параллелизма данных объединяет несколько итераций иногда тело цикла может выполняться настолько быстро, по скорости превосходят вызов делегата, необходимый для вызова тела цикла в каждой итерации. В этом случае можно использовать класс `Partitioner` и с его помощью вручную группировать итерации, уменьшая количество вызовов делегатов:

```
Parallel.For(
    Partitioner.Create(3, 200000), range => { // range – это
                                              // Tuple <int,int>
    for (int i = range.Item1; i < range.Item2; ++i)
        ...   // тело цикла без вызова делегата
});
```

За дополнительной информацией о разбиении циклов на фрагменты, являющемся важным способом оптимизации, обращайтесь к статье «Пользовательские разделители для PLINQ и TPL» на сайте MSDN: http://msdn.microsoft.com/ru-ru/library/dd997411.aspx.

Наконец, существуют приложения, в которых могут пригодиться собственные планировщики задач. В качестве примеров можно привести планирование заданий в потоке управления пользовательским интерфейсом (нечто похожее мы уже делали, когда использовали `TaskScheduler.Current` для организации запуска продолжений в потоке пользовательского интерфейса), назначение приоритетов задачам, планируя их с помощью высокоприоритетного планировщика, и связывание задач с определенным процессором, планируя их с помощью планировщика, использующего потоки, привязанные к определен-

ному процессору. Реализовать собственные планировщики можно путем наследования класса `TaskScheduler`. Пример такой реализации можно найти в статье «Практическое руководство. Создание планировщика заданий, ограничивающего степень параллелизма» на сайте MSDN: http://msdn.microsoft.com/ru-ru/library/ee789351.aspx.

Синхронизация

Занимаясь проблемами параллельного программирования нельзя не упомянуть хотя бы вскользь тему синхронизации. В простых примерах выше мы не раз сталкивались с ситуациями совместного использования несколькими потоками выполнения общих структур данных, будь то сложная коллекция или простая целочисленная переменная. За исключением данных, доступных только для чтения, каждое обращение к совместно используемой области памяти требует синхронизации, но не все механизмы синхронизации имеют одинаковую производительность и масштабируемость.

Прежде чем приступить к исследованиям, давайте обсудим саму необходимость синхронизации при работе с данными маленького объема. Современные процессоры способны выполнять атомарные операции чтения и записи содержимого в памяти; Например, запись 32-разрядного целого числа всегда выполняется атомарно. Это означает, что если процессор записывает в ячейку памяти значение `0xDEADBEEF`, прежде инициализированную нулем, другой процессор никогда не получит частично измененное значение в этой ячейке, такое как `0xDEAD0000` или `0x0000BEEF`. К сожалению, то же самое нельзя сказать о более объемных данных; например, даже на 64-разрядном процессоре операция записи 20 байт не является атомарной и не может быть атомарной.

Всякий раз, когда для выполнения операции с областью памяти требуется выполнить несколько инструкций, немедленно возникает проблема синхронизации. Например, операция `++i` (где `i` является переменной на стеке типа `int`) обычно транслируется в последовательность из трех машинных инструкций:

```
mov eax, dword ptr [ebp-64] ; скопировать со стека в регистр
inc eax                     ; увеличить значение в регистре
mov dword ptr [ebp-64], eax ; скопировать из регистра на стек
```

Каждая из этих инструкций выполняется атомарно, но без дополнительной синхронизации есть вероятность, что два процессора выполнят части этой последовательности инструкций конкурентно, что

приведет к *потере изменений*. Допустим, что переменная изначально имеет значение 100, и исследуем следующую историю выполнения инструкций:

Процессор 1	Процессор 2
`mov eax, dword ptr [ebp-64]`	
	`mov eax, dword ptr [ebp-64]`
	`inc eax`
`inc eax`	
`mov dword ptr [ebp-64], eax`	
	`mov dword ptr [ebp-64], eax`

В этом случае переменная получит значение 101, даже при том, что операция инкремента была выполнена *двумя* процессорами и переменная должна была получить значение 102. Это состояние гонки за ресурсами (race condition) – мы надеемся, очевидно и легко обнаруживается, – является наглядным примером ситуаций, требующих синхронизации.

Другие решения

Многие исследователи и архитекторы языков программирования не верят, что проблему управления синхронизацией доступа к совместно используемой памяти можно решить без полного изменения семантики языка программирования, применения параллельных фреймворков или определения моделей памяти процессоров. Однако в этой области есть ряд интересных решений.

- Транзакционная память, реализованная аппаратно или программно, предлагает явную или неявную модель изоляции операций с памятью и возможность отката последовательности таких операций. В настоящее время производительность подобных решений препятствует их широкому распространению в основных языках программирования и фреймворках.
- Языки программирования, основанные на использовании модели агентов, интегрируют модель конкурентного доступа глубоко в язык и требуют организации явных взаимодействий между агентами (объектами) в терминах обмена сообщениями, а не операций с совместно используемой памятью.
- Процессор передачи сообщений и архитектуры памяти образуют систему с использованием парадигмы приватной памяти, когда доступ к совместно используемым областям должен осуществляться явно, посредством передачи сообщений на аппаратном уровне.

В оставшейся части этого раздела мы будем использовать более прагматичный подход и попытаемся решить проблемы синхронизации посредством более привычных механизмов синхронизации и шаблонов. Однако авторы уверены, что в действительности приемы синхронизации сложнее,

чем могли бы быть; наш совместный опыт показывает, что основные ошибки в многопоточных программах обусловлены неправильным применением механизмов синхронизации. Мы надеемся, что через несколько лет – или десятилетий – будут придуманы более удачные альтернативы.

Код без блокировок

Один из подходов к синхронизации основан на перемещении этого бремени в операционную систему. В конце концов, операционная система предоставляет средства для создания потоков и управления ими, и принимает на себя всю ответственность за их выполнение. Поэтому вполне естественно ожидать, что система предоставит комплект примитивов синхронизации. Чуть ниже мы обсудим механизмы синхронизации, имеющиеся в Windows, но при таком подходе возникает вопрос, как операционная система *реализует* эти механизмы. Конечно, ОС Windows сама нуждается в синхронизации доступа к своим внутренним структурам данных – даже к структурам данных, представляющим другие механизмы синхронизации – и не может реализовать механизмы синхронизации рекурсивно. Кроме того, как оказывается, механизмы синхронизации Windows часто требуют выполнения системных вызовов (что связано с переходом из режима пользователя в режим ядра), в результате чего происходит переключение контекста потока, что является слишком дорогим удовольствием, если операция, требующая синхронизации, достаточно дешева (например, увеличение числового значения или добавление элемента в связанный список).

Все семейства процессоров, на которых Windows может выполняться, поддерживают *аппаратный* примитив синхронизации с названием «Сравнить-и-Заменить» (Compare-And-Swap, CAS). Примитив CAS выполняет операцию атомарно и имеет следующую семантику (в псевдокоде):

```
WORD CAS(WORD* location, WORD value, WORD comparand) {
    WORD old = *location;
    if (old == comparand) {
        *location = value;
    }
    return old;
}
```

Проще говоря, примитив CAS сравнивает содержимое памяти по адресу `location` с указанным значением `comparand`. Если в памяти

хранится это значение, оно заменяется другим значением `value`; в противном случае значение в памяти не изменяется. В любом случае возвращается прежнее содержимое памяти, хранившееся до операции.

В процессорах Intel x86 этот примитив реализован в виде инструкции `LOCK CMPXCHG`. Трансляция вызова `CAS(&a,b,c)` в инструкцию `LOCK CMPXCHG` — это чисто механическая процедура, именно поэтому в оставшейся части раздела мы будем использовать аббревиатуру CAS. В .NET Framework примитив CAS реализован в виде множества перегруженных методов `Interlocked.CompareExchange`.

```
// Код на C#:
int n = ...;
if (Interlocked.CompareExchange(ref n, 1, 0) == 0) { // заменить 0 на 1
// ...выполнить некоторые операции
}

// инструкции на языке ассемблера x86:
mov eax, 0                          ; значение для сравнения
mov edx, 1                          ; новое значение
lock cmpxchg dword ptr [ebp-64], edx ; предполагается, что n в [ebp-64]
test eax, eax                       ; if eax = 0, проверка возможности
                                    ; замены
jnz not_taken
//  ... выполнить некоторые операции
not_taken:
```

Единственной CAS-операции часто недостаточно, чтобы обеспечить надежную синхронизацию, если только не требуется выполнить единственную операцию проверки с заменой. Однако в комбинации с конструкцией цикла примитив CAS может с успехом использоваться для решения самых разных задач синхронизации. Для начала рассмотрим простой пример умножения на месте. Нам требуется атомарно выполнить операцию `x * = y`, где `x` является совместно используемой переменной, запись в которую может выполняться одновременно несколькими потоками, а `y` — это константа, недоступная для изменения. Ниже приводится метод на языке C#, решающий поставленную задачу с применением примитива CAS:

```
public static void InterlockedMultiplyInPlace(ref int x, int y) {
    int temp, mult;
    do {
        temp = x;
        mult = temp * y;
    } while(Interlocked.CompareExchange(ref x, mult, temp) ! = temp);
}
```

Каждая итерация начинается с чтения значения `x` во временную переменную на стеке, которая недоступна другим потокам. Затем вычисляется результат умножения для записи в переменную `x`. И, наконец, цикл завершается, если `CompareExchange` сообщит, что замена прежнего значения `x` результатом умножения произведена успешно, вернув оригинальное значение. Мы не можем гарантировать, что цикл завершится за некоторое ограниченное число итераций; однако весьма маловероятно, даже при наличии других процессоров, цикл будет выполнен более чем несколько раз. Но, как бы то ни было, цикл готов к такой ситуации. Взгляните на следующую историю выполнения цикла двумя процессорами со значениями `x = 3, y = 5`:

Процессор 1	**Процессор 2**
`temp = x; (3)`	
	`temp = x; (3)`
`mult = temp * y; (15)`	
	`mult = temp * y; (15)`
	`CAS(ref x, mult, temp) == 3 (== temp)`
`CAS(ref x, mult, temp) == 15` **`(!= temp)`**	

Даже в таком простом примере очень легко получить ошибочные результаты. Например, следующий цикл может потерять изменения:

```
public static void InterlockedMultiplyInPlace(ref int x, int y) {
    int temp, mult;
    do {
        temp = x;
        mult = x * y;
    } while(Interlocked.CompareExchange(ref x, mult, temp) ! = temp);
}
```

Почему? Двукратное чтение значения `x`, даже в такой быстрой последовательности, не гарантирует, что будет получено одно и то же значение! Следующая история выполнения показывает, как могут быть получены неправильные результаты на двух процессорах и начальных значениях `x = 3, y = 5` — в конце выполнения получится результат: `x = 60`!

Процессор 1	**Процессор 2**
`temp = x; (3)`	
	`x = 12;`
`mult = x * y;` **`(60!)`**	
	`x = 3;`
`CAS(ref x, mult, temp) == 3 (== temp)`	

Этот результат можно обобщить на любой алгоритм, где требуется прочитать единственное значение из памяти и заменить его новым, независимо от его сложности. Ниже приводится обобщенная версия:

```
public static void DoWithCAS <T> (ref T location, Func <T,T> generator)
                                  where T : class
{
    T temp, replace;
    do {
        temp = location;
        replace = generator(temp);
    } while (Interlocked.CompareExchange(ref location, replace,
temp) ! = temp);
}
```

Операция умножения достаточно легко выражается в терминах обобщенной версии:

```
public static void InterlockedMultiplyInPlace(ref int x, int y) {
    DoWithCAS(ref x, t => t * y);
}
```

С помощью примитива CAS можно реализовать простой механизм синхронизации, который называется *циклической блокировкой* (spin-lock). Идея состоит в следующем: как только блокировка приобретается каким-то одним потоком, все другие потоки должны терпеть неудачу при попытке захватить ее, и повторять попытки. Циклическая блокировка может быть приобретена единственным потоком, а все остальные потоки должны будут «крутиться» (spin) в ожидании ее освобождения (понапрасну тратя процессорное время):

```
public class SpinLock {
    private volatile int locked;
    public void Acquire() {
        while (Interlocked.CompareExchange(ref locked, 1, 0) ! = 0);
    }
    public void Release() {
        locked = 0;
    }
}
```

Модели памяти и изменчивые переменные

Исчерпывающее исследование проблем синхронизации должно было бы включать обсуждение моделей памяти и потребность в изменчивых переменных (volatile variables). Однако, из-за нехватки места, необходимого,

чтобы достаточно подробно охватить эти темы, мы можем предложить лишь краткий экскурс. Более подробное описание можно найти в книге Джо Даффи (Joe Duffy), «Concurrent Programming on Windows» (Addison-Wesley, 2008).

Вообще говоря, модель памяти для конкретного языка/окружения описывает, как компилятор и процессор могут переупорядочивать операции, выполняемые разными потоками, влияющими на взаимодействие потоков посредством совместно используемой памяти. Хотя в большинстве моделей принимается, что операции чтения и записи с одной и той же областью памяти не могут переупорядочиваться, существует редко встречающееся соглашение о семантике операций чтения и записи, выполняемых с *разными* участками памяти. Например, следующая программа может вывести 13 при начальных значениях `f = 0, x = 13`:

```
Процессор 1          Процессор 2
while (f == 0);      x = 42;
print(x);            f = 1;
```

Причина такого, казалось бы, неожиданного результата состоит в том, что компилятор и процессор могут изменить порядок выполнения инструкций процессором 2 так, что запись в переменную `f` будет выполнена до записи в переменную `x`, а порядок выполнения инструкций процессором 1 так, что чтение переменной `x` будет выполнено перед чтением переменной `f`. Неправильное понимание конкретной модели памяти может приводить к чрезвычайно тяжело выявляемым ошибкам.

В C# имеется несколько средств, которые можно использовать для предупреждения проблем, связанных с переупорядочением операций. Первое из них – ключевое слово `volatile`, которое препятствует переупорядочению операций с конкретной переменной компилятором и большинством процессоров. Второе – множество методов класса `Interlocked` и метод `Thread.MemoryBarrier`, устанавливающие барьер, который не может быть преодолен в каком-то одном или в обоих направлениях при попытке переупорядочить инструкции. К счастью, механизмы синхронизации ОС Windows (вовлекающие системные вызовы), а также примитивы синхронизации без блокировок (lock-free) в TPL, автоматически устанавливают барьеры, когда это необходимо. Однако, если вы собираетесь заняться реализацией собственного механизма синхронизации, вам придется потратить немало времени, чтобы разобраться в модели памяти целевой среды выполнения.

Мы не можем выразить всю сложность такого предприятия: если вы соберетесь напрямую управлять упорядочением доступа к памяти, вам совершенно необходимо будет изучить модель памяти для каждого языка и каждой комбинации аппаратных средств, которые будут использоваться для создания ваших многопоточных приложений. У вас не будет ничего, что могло бы уберечь вас от ошибок.

В нашей реализации циклической блокировки значение 0 соответствует свободной блокировке, а 1 – занятой. Наша реализация

пытается заменить внутреннее значение единицей, передавая значение 0 для сравнения – то есть, блокировка может быть приобретена, только когда она свободна. Поскольку нет никаких гарантий, что поток, получивший блокировку, быстро освободит ее, использование циклической блокировки означает, что множество других потоков могут «крутиться» на месте, в ожидании освобождения блокировки, и напрасно расходовать процессорное время. Это делает циклические блокировки непригодными для защиты таких операций, как обращения к базе данных, вывод больших объемов данных в файлы, отправка пакетов в сеть, и другие продолжительные операции. Однако циклические блокировки весьма удобны, когда защищаемые ими разделы кода выполняются очень быстро – изменение нескольких полей в объекте, увеличение значений нескольких переменных или вставка элемента в простую коллекцию.

В действительности, циклические блокировки широко используются в ядре Windows для реализации внутренних механизмов синхронизации. Структуры данных в ядре, такие как база данных планировщика, список блоков в кеше файловой системы, база данных номеров страниц памяти и другие и другие, защищаются одной или несколькими циклическими блокировками. Кроме того, в ядре Windows используется дополнительная оптимизация простой реализации циклической блокировки, описанной выше и страдающей двумя проблемами:

1. Циклическая блокировка не *справедлива* в терминах семантики FIFO. Процессор может быть последним из десяти процессоров, вызвавших метод `Acquire`, но получить блокировку первым.
2. Освобождение циклической блокировки вызывает актуализацию кешей всех процессоров, вращающихся внутри метода `Acquire`, хотя только один процессор сможет приобрести ее. (Мы вернемся к проблеме актуализации кеша далее в этой главе.)

Ядро Windows использует *циклические блокировки с очередью* (in-stack queued spinlocks); циклическая блокировка с очередью поддерживает очередь процессоров, ожидающих ее освобождения, и каждый процессор, ожидающий на блокировке, «вращается» вокруг отдельной ячейки памяти, которая не кешируется другими процессорами. Когда процессор, владевший блокировкой, освобождает ее, он находит первый процессор в очереди и устанавливает бит, появление которого ожидает данный процессор. Это гарантирует семантику

FIFO очереди и предотвращает принудительную актуализацию кешей всех процессоров, кроме того, который благополучно приобрел блокировку.

> **Примечание.** *Реализации циклических блокировок промышленного уровня могут быть еще более надежными, предусматривать возможность определения предельного времени ожидания (за счет преобразования цикла в блокирующее ожидание), следить за потоком-владельцем, чтобы обеспечить корректное приобретение и освобождение, позволять рекурсивное приобретение блокировок и поддерживать дополнительные удобства. Одной из рекомендуемых реализаций является тип* `SpinLock` *из библиотеки Task Parallel Library.*

Имея на вооружении примитив синхронизации CAS, мы можем реализовать чудо инженерной мысли – стек без блокировок. В главе 5 мы уже познакомились с некоторыми параллельными коллекциями, поэтому не будем повторять обсуждение, а реализуем тип `ConcurrentStack<T>`, оставшийся для нас тайной. Как по волшебству, стек типа `ConcurrentStack<T>` позволяет нескольким потокам вталкивать и выталкивать элементы, не требуя использовать механизмы синхронизации (которые мы рассмотрим далее).

Реализацию стека без блокировки мы выполним на основе односвязного списка. Вершиной стека будет голова списка; операции вталкивания элемента в стек и выталкивания со стека будут реализованы как замена головы списка. Для синхронизации этих операций мы используем примитив CAS; фактически мы можем использовать вспомогательный метод `DoWithCAS<T>`, представленный выше:

```
public class LockFreeStack <T> {
    private class Node {
        public T Data;
        public Node Next;
    }
    private Node head;
    public void Push(T element) {
        Node node = new Node { Data = element };
        DoWithCAS(ref head, h => {
            node.Next = h;
            return node;
        });
    }
    public bool TryPop(out T element) {
        // метод DoWithCAS здесь не подходит, потому что нам
        // нужна семантика раннего завершения
        Node node;
        do {
            node = head;
            if (node == null) {
```

```
                element = default(T);
                return false; // срочный выход – здесь нечего возвращать
            }
        } while (Interlocked.CompareExchange(ref head, node.Next,
node) ! = node);
        element = node.Data;
        return true;
    }
}
```

Метод `Push` пытается заменить голову списка новым узлом, чей указатель `Next` будет указывать на текущую голову списка. Аналогично метод `TryPop` пытается заменить голову списка узлом, на который ссылается указатель `Next` текущей головы, как показано на рис. 6.8.

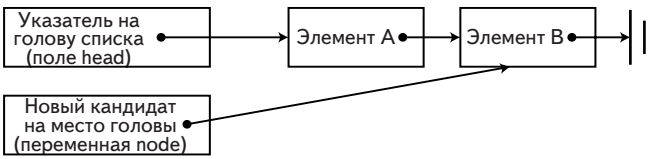

Рис. 6.8. Метод TryPop пытается заменить текущую голову списка новой.

Кому-то может показаться, что с помощью CAS и похожих примитивов синхронизации можно реализовать любую структуру данных. И действительно, мы можем привести примеры коллекций без блокировок, широко используемых в настоящее время:

- двусвязный список без блокировки;
- очередь без блокировки (с головой и хвостом);
- простая очередь с поддержкой приоритетов.

Однако существует множество разнообразных коллекций, которые не могут быть реализованы без блокировок и вынуждающих использовать блокирующие механизмы синхронизации. Кроме того, существует масса ситуаций, требующих использовать синхронизацию, но не позволяющих применять примитив CAS, потому что приходится выполнять продолжительные операции под защитой блокировки. Теперь обратимся к «настоящим» механизмам синхронизации, реализуемым операционной системой.

Механизмы синхронизации Windows

ОС Windows предлагает множество механизмов синхронизации для использования в программах, выполняющихся в пользовательском режиме, таких как события, семафоры, мьютексы и переменные ус-

ловий (condition variables). Все эти механизмы доступны программам посредством дескрипторов и функций Win32 API, которые обращаются к системным вызовам от нашего имени. Платформа .NET Framework обертывает большинство механизмов синхронизации Windows тонкими объектно-ориентированными обертками, такими как `ManualResetEvent`, `Mutex`, `Semaphore` и другими. Поверх имеющихся механизмов синхронизации .NET предлагает несколько новых, таких как `ReaderWriterLockSlim` и `Monitor`. Мы не будем подробно исследовать каждый из механизмов синхронизации и оставим эту задачу на долю документации, а займемся более важным для нас делом – изучением характеристик производительности.

Ядро Windows реализует механизмы синхронизации, обсуждаемые в данном разделе, которые блокируют выполнение потоков, пытающихся приобрести блокировку, когда она занята. Операция блокировки потока подразумевает снятие его с выполнения путем маркировки потока, как находящегося в состоянии ожидания, и запуск другого потока. При этом выполняется системный вызов, осуществляющий переход из пользовательского режима в режим ядра, переключающий контекст между двумя потоками и изменяющий небольшую структуру данных (рис. 6.9) в ядре, чтобы пометить поток как ожидающий и связать его с соответствующим механизмом синхронизации.

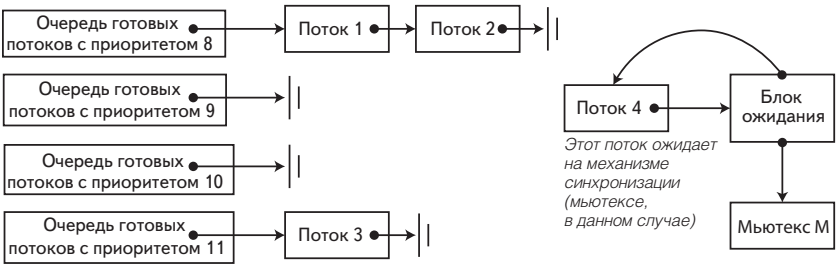

Рис. 6.9. Данные, поддерживаемые планировщиком операционной системы. Потоки, готовые к выполнению, помещаются в очередь FIFO, в порядке приоритетов. Заблокированные потоки ссылаются на соответствующие им механизмы синхронизации через внутреннюю структуру, называемую блоком ожидания.

В целом на блокировку потока может быть затрачено несколько тысяч тактов процессора, и примерно такое же количество тактов требуется для его разблокировки, когда механизм синхронизации освободится. Очевидно, что если механизм синхронизации в ядре применяется для защиты продолжительной операции, такой как вывод

большого буфера в файл или выполнение обмена по сети, накладные расходы на общем фоне будут незаметны, но если механизм синхронизации в ядре применяется для защиты такой операции, как `++i`, накладные расходы приведут к непозволительному замедлению.

Механизмы синхронизации ОС Windows и .NET доступны приложениям в первую очередь в терминах семантики приобретения и освобождения, также известной как семантика сигнального состояния (signal state) . Когда механизм синхронизации взводится в сигнальное состояние, он пробуждает поток (или группу потоков), ожидающих освобождения блокировки. В табл. 6.1 описывается семантика состояния сигнала для некоторых из механизмов синхронизации, доступных в настоящее время приложениям для .NET:

Таблица 6.1. Семантика состояния сигнала для некоторых механизмов синхронизации

Механизм синхронизации	Когда переходит в сигнальное состояние?	Какие потоки пробуждаются?
`Mutex`	Когда поток вызывает `Mutex.ReleaseMutex`	Один из потоков, ожидающих на мьютексе
`Semaphore`	Когда поток вызывает `Semaphore.Release`	Один из потоков, ожидающих на семафоре
`ManualResetEvent`	Когда поток вызывает `ManualResetEvent.Set`	Все потоки, ожидающие событие
`AutoResetEvent`	Когда поток вызывает `AutoResetEvent.Set`	Один из потоков, ожидающих событие
`Monitor`	Когда поток вызывает `Monitor.Exit`	Один из потоков, ожидающих на мониторе
`Barrier`	Когда поток вызывает `Barrier.SignalAndWait`	Все потоки, ожидающие на барьере
`ReaderWriterLock` – для чтения	Когда не остается пишущих потоков или когда последний пишущий поток освобождает блокировку для записи	Все потоки, ожидающие возможности приобрести блокировку для чтения
`ReaderWriterLock` – для записи	Когда не остается ни пишущих, ни читающих потоков	Все потоки, ожидающие возможности приобрести блокировку для записи

Помимо семантик, отличных от семантики сигнального состояния, некоторые механизмы синхронизации отличаются также внутренней реализацией. Например, критические секции Win32 и мониторы CLR реализуют оптимизированные блокировки. При попытке приобрести

такую блокировку, доступную в данный момент, поток может получить ее сразу, без обращения к системному вызову. Имеется также семейство механизмов синхронизации, реализующих блокировки для чтения/записи, различающих попытки доступа для чтения и для записи и обеспечивающих более высокую масштабируемость, когда чтение данных выполняется значительно чаще записи.

Выбор подходящего механизма синхронизации из предлагаемых ОС Windows и .NET зачастую сделать очень непросто, а иногда более высокую производительность или удобную семантику можно получить, реализовав собственный механизм синхронизации. На этом мы заканчиваем исследование механизмов синхронизации; вам решать, какой механизм синхронизации выбрать – на основе примитивов синхронизации без блокировок или на основе блокировок, и какая комбинация механизмов лучше соответствует вашему приложению.

> **Примечание.** *Никакое обсуждение механизмов синхронизации не может считаться полным без перечисления структур данных (коллекций), изначально предназначенных для использования в многопоточной среде. Такие коллекции являются потокобезопасными – могут совместно использоваться несколькими потоками – и масштабируемыми, не вызывающими непомерное снижение производительности из-за использования блокировок. Описание коллекций, изначально предназначенных для использования в многопоточной среде, можно найти в главе 5.*

Вопросы оптимального использования кеша

Выше мы уже поднимали некоторые вопросы, касающиеся кешей процессора в контексте реализации коллекций и плотности их размещения в памяти. В параллельных программах важно учитывать размер кеша и коэффициент попаданий в кеш, но еще важнее понимать, как взаимодействуют кеши процессоров в многопроцессорных системах. В этом разделе мы рассмотрим один представительный пример, демонстрирующий важность оптимизации использования кеша, и подчеркнем важность применения хороших инструментов при решении проблем оптимизации.

Для начала исследуем следующий последовательный метод. Он суммирует элементы двумерного массива целых чисел и возвращает результат.

```
public static int MatrixSumSequential(int[,] matrix) {
    int sum = 0;
    int rows = matrix.GetUpperBound(0);
```

```
    int cols = matrix.GetUpperBound(1);
    for (int i = 0; i < rows; ++i) {
        for (int j = 0; j < cols; ++j) {
            sum + = matrix[i,j];
        }
    }
    return sum;
}
```

Мы знаем немало способов распараллеливания программ такого рода. Но представьте на мгновение, что у нас нет библиотеки TPL, и нам остается только работать с потоками выполнения напрямую. Ниже приводится попытка реализовать параллельное выполнение, которая, на первый взгляд, позволяет получить выгоды от выполнения в многопроцессорной системе, и даже использует прием агрегирования, чтобы избежать необходимости синхронизировать доступ к общей переменной `sum`:

```
public static int MatrixSumParallel(int[,] matrix) {
    int sum = 0;
    int rows = matrix.GetUpperBound(0);
    int cols = matrix.GetUpperBound(1);
    const int THREADS = 4;
    int chunk = rows/THREADS; // должно делиться нацело
    int[] localSums = new int[THREADS];
    Thread[] threads = new Thread[THREADS];
    for (int i = 0; i < THREADS; ++i) {
        int start = chunk*i;
        int end = chunk*(i + 1);

        // воспрепятствовать "подъему" переменной компилятором
        // в lambda-захвате
        int threadNum = i;

        threads[i] = new Thread(() => {
            for (int row = start; row < end; ++row) {
                for (int col = 0; col < cols; ++col) {
                    localSums[threadNum] + = matrix[row,col];
                }
            }
        });
        threads[i].Start();
    }
    foreach (Thread thread in threads) {
        thread.Join();
    }
    sum = localSums.Sum();
    return sum;
}
```

Выполнив каждый из методов 25 раз в системе с процессором Intel i7, мы получили следующие результаты для матрицы a 2000 × 2000: среднее время выполнения последовательного метода составило 325 мсек, а параллельного метода 935 мсек, в три раза медленнее последовательной версии!

Понятно, что это совершенно неприемлемо, но почему так получилось? Эта ситуация не может служить еще одним примером слишком мелкого дробления задачи, потому что было запущено всего 4 потока. Если предположить, что проблема имеет некоторое отношение к кешу процессора (хотя бы потому, что пример приводится в разделе с названием «Вопросы оптимального использования кеша»), имеет смысл исследовать количество промахов кеша в каждом из двух методов. Профилировщик Visual Studio (с частотой срабатывания 2000 раз в секунду) сообщил о 963 промахах в параллельной версии и только о 659 промахах в последовательной; подавляющее большинство промахов было обнаружено в теле внутреннего цикла, выполняющем чтение элемента матрицы.

И снова, почему? Почему запись в массив `localSums` порождает промахов больше, чем запись в локальную переменную? Ответ прост: дело в том, что запись в совместно используемый массив вызывает необходимость *актуализации строк кеша в других процессорах*, что в каждой операции += приводит к промаху кеша.

Как описывалось в главе 5, кеши процессора организованы в виде последовательностей строк кеша, и ячейки памяти, располагающиеся рядом, оказываются в одной и той же строке кеша. Когда один из процессоров выполняет запись в память, прочитанную в кеш другим процессором, аппаратное окружение вызывает актуализацию кеша, помечая строку в кеше другого процессора, как недействительную. Попытка обратиться к недействительной строке кеша приводит к промаху. В примере выше весьма вероятно, что массив `localSums` целиком уместится в одну строку кеша, и одновременно в кешах всех четырех процессоров, выполняющих потоки приложения. Каждая операция записи в этот массив любым из процессоров будет вызывать необходимость актуализации строки кеша во всех четырех процессорах, приводя к постоянному обновлению кешей (рис. 6.10).

Чтобы убедиться, что проблема действительно связана только с необходимостью актуализации кеша, можно попробовать увеличить размер массива и разнести элементы, где накапливаются суммы, подальше друг от друга, или заменить массив локальны-

ми переменными, которые будут записываться в общий массив по окончании вычислений. Любое из этих усовершенствований восстановит порядок в мире и сделает параллельную версию быстрее последовательной.

Проблема актуализации кеша – очень неприятная и исключительно сложно поддается выявлению в реальных приложениях, даже с помощью мощных профилировщиков. Понимание причин, ее порождающих, и применение приемов профилактики при проектировании вычислительных алгоритмов поможет вам сэкономить массу времени позже.

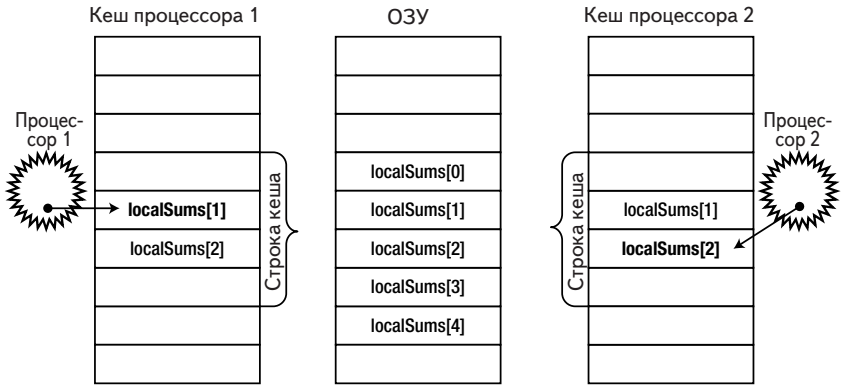

Рис. 6.10. Процессор 1 записывает значение в localSums[1], а процессор 2 записывает значение в localSums[2]. Так как оба элемента массива находятся в смежных ячейках памяти и попадают в одну строку кеша в кешах обоих процессоров, каждая операция записи вызывает необходимость актуализации кеша другого процессора.

Примечание. *Авторы столкнулись с похожей ситуацией в промышленном приложении, которое распределяло задания между двумя потоками, выполняющимися на двух процессорах, с применением общей очереди. После внесения незначительных изменений во внутреннюю структуру полей класса очереди обнаружилось существенное падение производительности (примерно 20%). В ходе скрупулезных исследований выяснилось, что падение производительности было вызвано переупорядочением полей в классе очереди; два поля, запись в которые выполняются разными потоками, оказались слишком близко друг к другу и попали в одну строку кеша. После разнесения полей, производительность очереди восстановилась до приемлемого уровня.*

Использование GPU для вычислений

До сих пор в обсуждении приемов параллельного программирования мы рассматривали только ядра процессора. Мы приобрели некоторые навыки распараллеливания программ по нескольким процессорам, синхронизации доступа к совместно используемым ресурсам и использования высокоскоростных примитивов синхронизации без применения блокировок. Однако, как отмечалось в начале этой главы, существует еще один способ распараллеливания программ – графические процессоры (GPU), обладающие большим числом ядер, чем даже высокопроизводительные процессоры. Ядра графических процессоров прекрасно подходят для реализации параллельных алгоритмов обработки данных, а большое их количество с лихвой окупает неудобства выполнения программ на них. В этом разделе мы познакомимся с одним из способов выполнения программ на графическом процессоре, с использованием комплекта расширений языка C++ под названием C++ AMP.

Примечание. *Расширения C++ AMP основаны на языке C++ и именно поэтому в данном разделе будут демонстрироваться примеры на языке C++. Однако, при умеренном использовании механизма взаимодействий в .NET, вы сможете использовать алгоритмы C++ AMP в своих программах для .NET. Но об этом мы поговорим в конце данного раздела.*

Введение в C++ AMP

По сути, графический процессор является таким же процессором, как любые другие, но с особым набором инструкций, большим количеством ядер и своим протоколом доступа к памяти. Однако между современными графическими и обычными процессорами существуют большие отличия, и их понимание является залогом создания программ, эффективно использующих вычислительные мощности графического процессора.

- Современные графические процессоры обладают очень маленьким набором инструкций. Это подразумевает некоторые ограничения: отсутствие возможности вызова функций, ограниченный набор поддерживаемых типов данных, отсутствие библиотечных функций и другие. Некоторые операции, такие как условные переходы, могут стоить значительно дороже, чем

аналогичные операции, выполняемые на обычных процессорах. Очевидно, что перенос больших объемов кода с процессора на графический процессор при таких условиях требует значительных усилий.

- Количество ядер в среднем графическом процессор значительно больше, чем в среднем обычном процессоре. Однако некоторые задачи оказываются слишком маленькими или не позволяют разбивать себя на достаточно большое количество частей, чтобы можно было извлечь выгоду от применения графического процессора.
- Поддержка синхронизации между ядрами графического процессора, выполняющими одну задачу, весьма скудна, и полностью отсутствует между ядрами графического процессора, выполняющими разные задачи. Это обстоятельство требует синхронизации графического процессора с обычным процессором.

Какие задачи подходят для решения на графическом процессоре?

Имейте в виду, что не всякий алгоритм подходит для выполнения на графическом процессоре. Например, графические процессоры не имеют доступа к устройствам ввода/вывода, поэтому у вас не получится повысить производительность программы, извлекающей ленты RSS из Веб, за счет использования графического процессора. Однако на графический процессор можно перенести многие вычислительные алгоритмы и обеспечить массовое их распараллеливание. Ниже приводится несколько примеров таких алгоритмов (этот список далеко не полон):

- увеличение и уменьшение резкости изображений, и другие преобразования;
- быстрое преобразование Фурье;
- транспонирование и умножение матриц;
- сортировка чисел;
- инверсия хеша «в лоб».

Отличным источником дополнительных примеров может служить блог Microsoft Native Concurrency (http://blogs.msdn.com/b/nativeconcurrency/), где приводятся фрагменты кода и пояснения к ним для различных алгоритмов, реализованных на C++ AMP.

C++ AMP – это фреймворк, входящий в состав Visual Studio 2012, дающий разработчикам на C++ простой способ выполнения вычислений на графическом процессоре и требующий лишь наличия драйвера DirectX 11. Корпорация Microsoft выпустила C++ AMP как открытую спецификацию (доступную по адресу: http://blogs.msdn.

com/b/nativeconcurrency/archive/2012/02/03/c-amp-open-spec-published.aspx), которую может реализовать любой производитель компиляторов. Фреймворк C++ AMP позволяет выполнять код на графических *ускорителях* (accelerators), являющихся вычислительными устройствами. С помощью драйвера DirectX 11 фреймворк C++ AMP динамически обнаруживает все ускорители. В состав C++ AMP входят также программный эмулятор ускорителя и эмулятор на базе обычного процессора, WARP, которые служит запасным вариантом в системах без графического процессора или с графическим процессором, но в отсутствие драйвера DirectX 11, и использует несколько ядер и инструкции SIMD.

А теперь приступим к исследованию алгоритма, который легко можно распараллелить для выполнения на графическом процессоре. Реализация ниже принимает два вектора одинаковой длины и вычисляет поточечный результат. Сложно представить что-либо более прямолинейное:

```
void VectorAddExpPointwise(float* first, float* second, float* result,
int length) {
    for (int i = 0; i < length; ++i) {
        result[i] = first[i] + exp(second[i]);
    }
}
```

Чтобы распараллелить этот алгоритм на *обычном* процессоре, требуется разбить диапазон итераций на несколько поддиапазонов и запустить по одному потоку выполнения для каждого из них. Выше мы посвятили достаточно много времени именно такому способу распараллеливания нашего первого примера поиска простых чисел – мы видели, как можно это сделать, создавая потоки вручную, передавая задания пулу потоков и используя `Parallel.For` для автоматического распараллеливания. Вспомните также, что при распараллеливании похожих алгоритмов на обычном процессоре мы особо заботились, чтобы не раздробить задачу на слишком мелкие задания.

Для *графического процессора* эти предупреждения не нужны. Графические процессоры имеют множество ядер, выполняющих потоки очень быстро, а стоимость переключения контекста значительно ниже, чем в обычных процессорах. Ниже приводится фрагмент, пытающийся использовать функцию `parallel_for_each` из фреймворка C++ AMP:

```
#include < amp.h>
#include < amp_math.h>
```

```
using namespace concurrency;

void VectorAddExpPointwise(float* first, float* second, float* result,
int length) {
    array_view <const float,1> avFirst (length, first);
    array_view <const float,1> avSecond(length, second);
    array_view <float,1> avResult(length, result);
    avResult.discard_data();
    parallel_for_each(avResult.extent, [=](index<1> i) restrict(amp) {
        avResult[i] = avFirst[i] + fast_math::exp(avSecond[i]);
    });
    avResult.synchronize();
}
```

Теперь исследуем каждую часть кода отдельно. Сразу заметим, что общая форма главного цикла сохранилась, но первоначально использовавшийся цикл `for` был заменен вызовом функции `parallel_for_each`. В действительности, принцип преобразования цикла в вызов функции или метода для нас не нов – выше уже демонстрировался такой прием с применением методов `Parallel.For` и `Parallel.ForEach` из библиотеки TPL.

Далее, входные данные (параметры `first`, `second` и `result`) обертываются экземплярами `array_view`. Класс `array_view` служит для обертывания данных, передаваемых графическому процессору (ускорителю). Его шаблонный параметр определяет тип данных и их размерность. Чтобы выполнить на графическом процессоре инструкции, обращающиеся к данным, первоначально обрабатываемым на обычном процессоре, кто-то или что-то должен позаботиться о копировании данных в графический процессор, потому что большинство современных графических карт являются отдельными устройствами с собственной памятью. Эту задачу решают экземпляры `array_view` – они обеспечивают копирование данных по требованию и только когда они действительно необходимы.

Когда графический процессор выполнит задание, данные копируются обратно. Создавая экземпляры `array_view` с аргументом типа `const`, мы гарантируем, что `first` и `second` будут скопированы *в* память графического процессора, но не будут копироваться *обратно*. Аналогично, вызывая `discard_data`, мы исключаем копирование `result` из памяти обычного процессора в память ускорителя, но эти данные будут копироваться в обратном направлении.

Функция `parallel_for_each` принимает объект `extent`, определяющий форму обрабатываемых данных и функцию для применения к каждому элементу в объекте `extent`. В примере выше мы исполь-

зовали лямбда-функцию, поддержка которых появилась в стандарте ISO C++ 2011 (C++11). Ключевое слово `restrict(amp)` поручает компилятору проверить возможность выполнения тела функции на графическом процессоре и отключает большую часть синтаксиса C++, который не может быть скомпилирован в инструкции графического процессора.

Параметр лямбда-функции, `index<1>` объекта, представляет одномерный индекс. Он должен соответствовать используемому объекту `extent` – если бы мы объявили объект `extent` двумерным (например, определив форму исходных данных в виде двумерной матрицы), индекс также должен был бы быть двумерным. Пример такой ситуации приводится чуть ниже.

Наконец, вызов метода `synchronize` в конце метода `VectorAddExpPointwise` гарантирует копирование результатов вычислений из `array_view avResult`, произведенных графическим процессором, обратно в массив `result`.

На этом мы заканчиваем наше первое знакомство с миром C++ AMP, и теперь мы готовы к более подробным исследованиям, а так же к более интересным примерам, демонстрирующим выгоды от использования параллельных вычислений на графическом процессоре. Сложение векторов – не самый удачный алгоритм и не самый лучший кандидат для демонстрации использования графического процессора из-за больших накладных расходов на копирование данных. В следующем подразделе будут показаны два более интересных примера.

Умножение матриц

Первый «настоящий» пример, который мы рассмотрим, – умножение матриц. Для реализации мы возьмем простой кубический алгоритм умножения матриц, а не алгоритм Штрассена, имеющий время выполнения, близкое к кубическому ($\sim O(n^{2.807})$). Для двух матриц: матрицы A размером $m \times w$ и матрицы B размером $w \times n$, следующая программа выполнит их умножение и вернет результат – матрицу C размером $m \times n$:

```
void MatrixMultiply(int* A, int m, int w, int* B, int n, int* C) {
    for (int i = 0; i < m; ++i) {
        for (int j = 0; j < n; ++j) {
            int sum = 0;
            for (int k = 0; k < w; ++k) {
                sum + = A[i*w + k] * B[k*w + j];
            }
            C[i*n + j] = sum;
```

```
        }
    }
}
```

Распараллелить эту реализацию можно несколькими способами, и при желании распараллелить этот код для выполнения на обычном процессоре правильным выбором был бы прием распараллеливания внешнего цикла. Однако графический процессор имеет достаточно большое количество ядер и распараллелив только внешний цикл, мы не сможем создать достаточное количество заданий, чтобы загрузить работой все ядра. Поэтому имеет смысл распараллелить два внешних цикла, оставив внутренний цикл нетронутым:

```
void MatrixMultiply(int* A, int m, int w, int* B, int n, int* C) {
    array_view <const int,2> avA(m, w, A);
    array_view <const int,2> avB(w, n, B);
    array_view <int,2> avC(m, n, C);
    avC.discard_data();
    parallel_for_each(avC.extent, [=](index<2> idx) restrict(amp) {
        int sum = 0;
        for (int k = 0; k < w; ++k) {
            sum + = avA(idx[0]*w, k) * avB(k*w, idx[1]);
        }
        avC[idx] = sum;
    });
}
```

Эта реализация все еще близко напоминает последовательную реализацию умножения матриц и пример сложения векторов, приводившиеся выше, за исключением индекса, который теперь является двумерным и доступен во внутреннем цикле с применением оператора `[]`. Насколько эта версия быстрее последовательной альтернативы, выполняемой на обычном процессоре? Умножение двух матриц (целых чисел) размером 1024 × 1024 последовательная версия на обычном процессоре выполняет в среднем 7350 миллисекунд, тогда как версия для графического процессора – держитесь крепче – 50 миллисекунд, в *147 раз быстрее*!

Моделирование движения частиц

Примеры решения задач на графическом процессоре, представленные выше, имеют очень простую реализацию внутреннего цикла. Понятно, что так будет не всегда. В блоге Native Concurrency, ссылка на который уже приводилась выше, демонстрируется пример моделирования гравитационных взаимодействий между частицами. Мо-

делирование включает бесконечное количество шагов; на каждом шаге вычисляются новые значения элементов вектора ускорений для каждой частицы и затем определяются их новые координаты. Здесь распараллеливанию подвергается вектор частиц – при достаточно большом количестве частиц (от нескольких тысяч и выше) можно создать достаточно большое количество заданий, чтобы загрузить работой все ядра графического процессора.

Основу алгоритма составляет реализация определения результата взаимодействий между двумя частицами, как показано ниже, которую легко можно перенести на графический процессор:

```
// здесь float4 – это векторы с четырьмя элементами,
// представляющий частицы, участвующие в операциях
void bodybody_interaction(
    float4& acceleration, const float4 p1, const float4 p2) restrict(amp) {
    float4 dist = p2 – p1;
    float absDist = dist.x*dist.x + dist.y*dist.y +
                    dist.z*dist.z; // w здесь не используется
    float invDist = 1.0f / sqrt(absDist);
    float invDistCube = invDist*invDist*invDist;
    acceleration += dist*PARTICLE_MASS*invDistCube;
}
```

Исходными данными на каждом шаге моделирования является массив с координатами и скоростями движения частиц, а в результате вычислений создается новый массив с координатами и скоростями частиц:

```
struct particle {
    float4 position, velocity;
    // реализации конструктора, конструктора копирования и
    // оператора = с restrict(amp) опущены для экономии места
};
void simulation_step(array <particle,1> & previous,
                     array <particle,1> & next, int bodies) {
    extent <1> ext(bodies);
    parallel_for_each(ext, [&](index <1> idx) restrict(amp) {
        particle p = previous[idx];
        float4 acceleration(0, 0, 0, 0);
        for (int body = 0; body < bodies; ++body) {
            bodybody_interaction(acceleration, p.position,
previous[body].position);
        }
        p.velocity + = acceleration*DELTA_TIME;
        p.position + = p.velocity*DELTA_TIME;
        next[idx] = p;
    });
}
```

С привлечением соответствующего графического интерфейса моделирование может оказаться очень интересным. Полный пример, представленный командой разработчиков C++ AMP, можно найти в блоге Native Concurrency. На системе автора с процессором Intel i7 и видеокартой ATI Radeon HD 5800, моделирование движения 10 000 частиц выполняется со скоростью ~2.5 кадра в секунду (шагов в секунду) с использованием последовательной версии, выполняющейся на обычном процессоре, и 160 кадров в секунду с использованием оптимизированной версии, выполняющейся на графическом процессоре рис. 6.11) – огромное увеличение производительности.

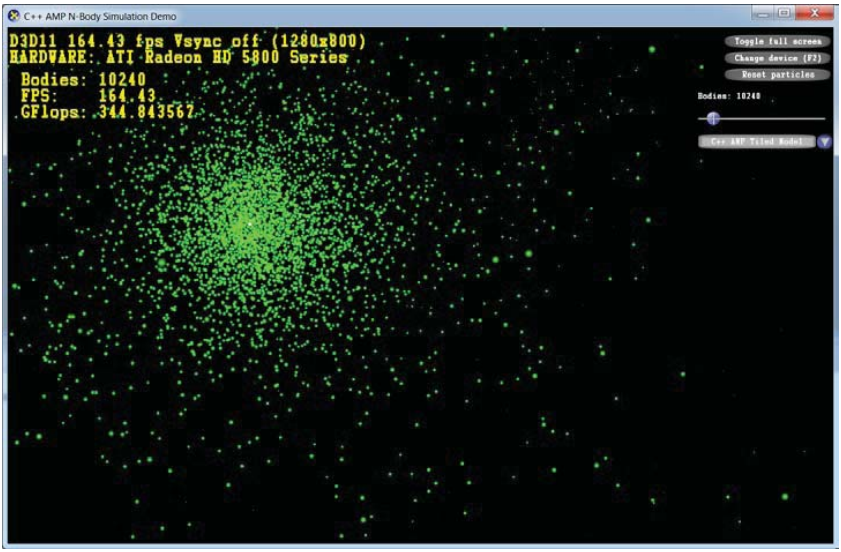

Рис. 6.11. Графический интерфейс, отображающий результаты моделирования взаимодействий 10240 частиц, показывает скорость вычислений >160 кадров в секунду (шагов моделирования) при использовании оптимизированной реализации с применением фреймворка C++ AMP.

Мозаики и разделяемая память

Прежде чем завершить этот раздел, необходимо рассказать еще об одной важной особенности фреймворка C++ AMP, которая может еще больше повысить производительность кода, выполняемого на графическом процессоре. Графические процессоры поддерживают программируемый кеш данных (часто называемый *разделяемой па-*

мятью (shared memory)). Значения, хранящиеся в этом кеше, совместно используются всеми потоками выполнения в одной мозаике (tile). Благодаря мозаичной организации памяти, программы на основе фреймворка C++ AMP могут читать данные из памяти графической карты в разделяемую память мозаики и затем обращаться к ним из *нескольких* потоков выполнения без повторного извлечения этих данных из памяти графической карты. Доступ к разделяемой памяти мозаики выполняется примерно в 10 раз быстрее, чем к памяти графической карты. Иными словами, у вас есть причины продолжить чтение.

Чтобы обеспечить выполнение мозаичной версии параллельного цикла, методу `parallel_for_each` передается домен `tiled_extent`, который делит многомерный объект `extent` на многомерные фрагменты мозаики, и лямбда-параметр `tiled_index`, определяющий глобальный и локальный идентификатор потока внутри мозаики. Например, матрицу 16 × 16 можно разделить на фрагменты мозаики размером 2 × 2 (рис. 6.12) и затем передать функции `parallel_for_each`:

```
extent <2> matrix(16,16);
tiled_extent <2,2> tiledMatrix = matrix.tile <2,2> ();
parallel_for_each(tiledMatrix, [=](tiled_index<2,2> idx)
restrict(amp) {
    ...
});
```

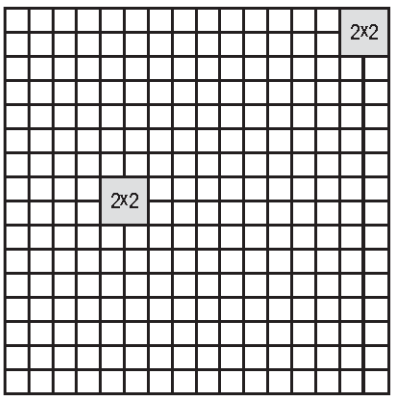

Рис. 6.12. Матрица 16 × 16 делится на блоки размером 2 × 2. Каждый из четырех потоков выполнения, принадлежащих одной и той же мозаике, могут совместно использовать данные, хранящиеся в блоке.

При выполнении операций с матрицами, в ядре графического процессора, взамен стандартного индекса `index<2>`, как в примерах выше, можно использовать `idx.global`. Грамотное использование локальной мозаичной памяти и локальных индексов может обеспечить существенный прирост производительности. Чтобы объявить мозаичную память, разделяемую всеми потоками выполнения в одной мозаике, локальные переменные можно объявить со спецификатором `tile_static`. На практике часто используется прием объявления разделяемой памяти и инициализации отдельных ее блоков в разных потоках выполнения:

```
parallel_for_each(tiledMatrix, [=](tiled_index <2,2> idx) restrict(amp) {
    tile_static int local[2][2]; // 32 байта совместно
                                 // используются всеми потоками
                                 // в блоке
    local[idx.local[0]][idx.local[1]] = 42; // присвоить значение
                                            // элементу для этого
                                            // потока выполнения
});
```

Очевидно, что какие-либо выгоды от использования разделяемой памяти можно получить только в случае синхронизации доступа к этой памяти; то есть, потоки не должны обращаться к памяти, пока она не будет инициализирована одним из них. Синхронизация потоков в мозаике выполняется с помощью объектов `tile_barrier` (напоминающего класс `Barrier` из библиотеки TPL) – они смогут продолжить выполнение только после вызова метода `tile_barrier.wait`, который вернет управление только когда все потоки вызовут `tile_barrier.wait`. Например:

```
parallel_for_each(tiledMatrix, [](tiled_index <2,2> idx) restrict(amp) {
    tile_static int local[2][2]; // 32 байта памяти, разделяемой
                                 // между всеми потоками в блоке
    local[idx.local[0]][idx.local[1]] = 42; // присвоить значение
                                            // элементу для этого
                                            // потока выполнения
    idx.barrier.wait(); // idx.barrier – экземпляр tile_barrier
    // Теперь этот поток может обращаться к массиву "local",
    // используя индексы других потоков выполнения!
});
```

Теперь самое время воплотить полученные знания в конкретный пример. Вернемся к реализации умножения матриц, выполненной без применения мозаичной организации памяти, и добавим в него описываемую оптимизацию. Допустим, что размер матрицы кратен

числу 256 – это позволит нам работать с блоками 16 × 16. Природа матриц допускает возможность поблочного их умножения, и мы можем воспользоваться этой особенностью (фактически, деление матриц на блоки является типичной оптимизацией алгоритма умножения матриц, обеспечивающей более эффективное использование кеша процессора). Суть этого приема сводится к следующему. Чтобы найти $C_{i,j}$ (элемент в строке i и в столбце j в матрице результата), нужно вычислить скалярное произведение между $A_{i,*}$ (i-я строка первой матрицы) и $B_{*,j}$ (j-й столбец во второй матрице). Однако, это эквивалентно вычислению частичных скалярных произведений строки и столбца с последующим суммированием результатов. Мы можем использовать это обстоятельство для преобразования алгоритма умножения матриц в мозаичную версию:

```
void MatrixMultiply(int* A, int m, int w, int* B, int n, int* C) {
    array_view < const int,2 > avA(m, w, A);
    array_view < const int,2 > avB(w, n, B);
    array_view < int,2> avC(m, n, C);
    avC.discard_data();
    parallel_for_each(avC.extent.tile<16,16> (),
                      [=](tiled_index<16,16> idx) restrict(amp) {
        int sum = 0;
        int localRow = idx.local[0], localCol = idx.local[1];
        for (int k = 0; k < w; k += 16) {
           tile_static int localA[16][16], localB[16][16];
           localA[localRow][localCol] = avA(idx.global[0], localCol + k);
           localB[localRow][localCol] = avB(localRow + k, idx.global[1]);
           idx.barrier.wait();
           for (int t = 0; t < 16; ++t) {
               sum + = localA[localRow][t]*localB[t][localCol];
           }
           idx.barrier.wait(); // чтобы избежать затирания
                               // разделяемой памяти в следующей
                               // итерации
        }
        avC[idx.global] = sum;
    });
}
```

Суть описываемой оптимизации в том, что каждый поток в мозаике (для блока 16 × 16 создается 256 потоков) инициализирует свой элемент в 16 × 16 локальных копиях фрагментов исходных матриц A и B (рис. 6.13). Каждому потоку в мозаике требуется только одна строка и один столбец из этих блоков, но все потоки вместе будут обращаться к каждой строке и к каждому столбцу по 16 раз. Такой подход существенно снижает количество обращений к основной памяти.

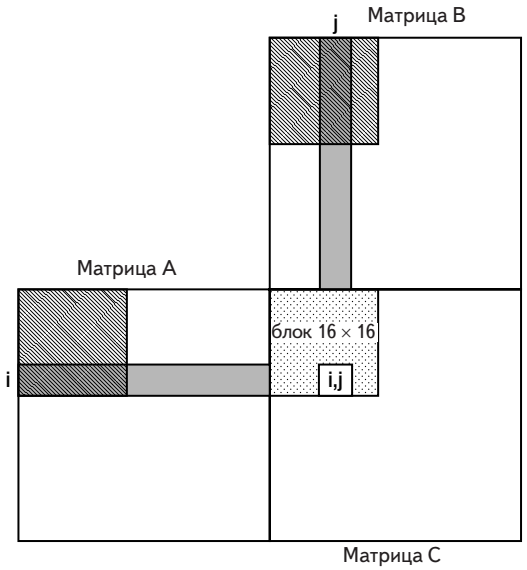

Рис. 6.13. Чтобы вычислить элемент (*i*,*j*) в матрице результата, алгоритму требуется полная *i*-я строка первой матрицы и *j*-й столбец второй матрицы. Когда потоки мозаике 16 × 16, представленные на диаграмме и *k* = 0, заштрихованные области в первой и второй матрицах будут прочитаны в разделяемую память. Поток выполнения, вычисляющий элемент (*i*,*j*) в матрице результата, вычислит частичное скалярное произведение первых *k* элементов из *i*-й строки и *j*-го столбца исходных матриц.

В данном примере применение мозаичной организации обеспечивает огромный прирост производительности. Мозаичная версия умножения матриц выполняется намного быстрее простой версии и занимает примерно 17 миллисекунд (для тех же исходных матриц размером 1024 × 1024), *что в 430 быстрее* версии, выполняемой на обычном процессоре!

Прежде чем закончить обсуждение фреймворка C++ AMP, нам хотелось бы упомянуть инструменты (в Visual Studio), имеющиеся в распоряжении разработчиков. Visual Studio 2012 предлагает отладчик для графического процессора (GPU), позволяющий устанавливать контрольные точки, исследовать стек вызовов, читать и изменять значения локальных переменных (некоторые ускорители поддерживают отладку для GPU непосредственно; для других Visual Studio использует программный симулятор), и профилировщик, да-

ющий возможность оценивать выгоды, получаемые приложением от распараллеливания операций с применением графического процессора. За дополнительной информацией о возможностях отладки в Visual Studio обращайтесь к статье «Пошаговое руководство. Отладка приложения C++ AMP» на сайте MSDN: http://msdn.microsoft.com/ru-ru/library/hh368280%28v=VS.110%29.aspx.

Альтернативы вычислений на графическом процессоре в .NET

До сих пор в этом разделе демонстрировались примеры только на языке C++, тем не менее, есть несколько способов использовать мощь графического процессора в управляемых приложениях. Один из способов – использовать инструменты взаимодействий (обсуждаемые в главе 8), позволяющие переложить работу с ядрами графического процессора на низкоуровневые компоненты C++. Это решение отлично подходит для тех, кто желает использовать фреймворк C++ AMP или имеет возможность использовать уже готовые компоненты C++ AMP в управляемых приложениях.

Другой способ – использовать библиотеку, непосредственно работающую с графическим процессором из управляемого кода. В настоящее время существует несколько таких библиотек. Например, GPU.NET и CUDAfy.NET (обе являются коммерческими предложениями). Ниже приводится пример из репозитория GPU.NET GitHub, демонстрирующий реализацию скалярного произведения двух векторов:

```
[Kernel]
public static void MultiplyAddGpu(double[] a, double[] b, double[] c) {
    int ThreadId = BlockDimension.X * BlockIndex.X + ThreadIndex.X;
    int TotalThreads = BlockDimension.X * GridDimension.X;
    for(int ElementIdx=ThreadId;ElementIdx<a.Length;ElementIdx+=TotalThreads)
    {
        c[ElementIdx] = a[ElementIdx] * b[ElementIdx];
    }
}
```

По мнению авторов этой книги, гораздо проще и эффективнее освоить расширение языка (на основе C++ AMP), чем пытаться организовывать взаимодействия на уровне библиотек или вносить существенные изменения в язык IL.

В этом разделе мы лишь вскользь коснулись возможностей, предлагаемых фреймворком C++ AMP. Мы познакомились лишь с несколькими функциями и парой примеров реализации параллельных алгоритмов. Желающим поближе познакомиться с фреймворком C++ AMP, мы настоятельно рекомендуем обратиться к книге Кейт Грегори (Kate Gregory) и Эйда Миллера (Ade Miller), «C++ AMP:

Accelerated Massive Parallelism with Microsoft Visual C++» (Microsoft Press, 2012)[1].

В заключение

К концу этой главы наверняка ни у кого не осталось сомнений, что организация параллельных вычислений является важным способом повышения производительности. Во многих серверах и рабочих станциях по всему миру остаются неиспользуемыми бесценные вычислительные мощности обычных и графических процессоров, потому что приложения просто не задействуют их. Библиотека Task Parallel Library дает нам уникальную возможность включить в работу все имеющиеся ядра центрального процессора, хотя при этом и придется решать некоторые интереснейшие проблемы синхронизации, чрезмерного дробления задач и неравного распределения работы между потоками выполнения. Фреймворк C++ AMP и другие многоцелевые библиотеки организации параллельных вычислений на графическом процессоре с успехом можно использовать для распараллеливания вычислений между сотнями ядер графического процессора. Наконец, имеется, неисследованная в этой главе, возможность получить прирост производительности от применения *облачных* технологий распределенных вычислений, превратившихся в последнее время в одно из основных направлений развития информационных технологий.

1 Кейт Г., «C++ AMP. Построение массивно параллельных программ с помощью Microsoft Visual C++», ISBN: 978-5-94074-896-0, 2013, ДМК. – *Прим. перев.*

ГЛАВА 7. Сети, ввод/вывод и сериализация

Большая часть этой книги посвящена вопросам оптимизации производительности вычислений. Мы видели множество примеров настройки процедуры сборки мусора, распараллеливания циклов и рекурсивных алгоритмов, и даже занимались оптимизацией алгоритмов, чтобы снизить накладные расходы во время выполнения.

Для некоторых приложений оптимизация вычислительных аспектов дает лишь незначительный выигрыш в производительности, потому что узким местом в них являются операции ввода/вывода, такие как передача данных по сети или доступ к диску. По своему опыту можем сказать, что значительная доля проблем производительности связана вовсе не с применением неоптимальных алгоритмов или чрезмерной нагрузкой на процессор, а с неэффективным использованием устройств ввода/вывода. Давайте рассмотрим две ситуации, когда оптимизация ввода/вывода может повысить общую производительность.

- Приложение может испытывать серьезные вычислительные перегрузки из-за неэффективности операций ввода/вывода, увеличивающих накладные расходы. Хуже того, перегрузка может быть настолько велика, что оказывается ограничивающим фактором, мешающим максимально использовать пропускную способность устройств ввода/вывода.
- Устройство ввода/вывода может быть задействовано недостаточно полно или его возможности могут растрачиваться впустую из-за применения неэффективных шаблонов программирования, как например, передача большого количества данных маленькими порциями или неиспользование всей пропускной способности канала.

В этой главе обсуждаются стратегии увеличения производительности ввода/вывода в целом и производительности сетевых операций ввода/вывода в частности. Кроме того, мы займемся проблемой производительности операций сериализации и сравним несколько способов их реализации.

Общие понятия

В этом разделе описываются общие понятия ввода/вывода и даются рекомендации по повышению производительности ввода/вывода любого типа. Эти рекомендации в равной степени применимы к сетевым приложениям, к процессам, интенсивно работающим с диском, и даже к программам, осуществляющим доступ к нестандартным, высокопроизводительным аппаратным устройствам.

Синхронный и асинхронный ввод/вывод

При выполнении в синхронном режиме, функции ввода/вывода Win32 API (например, `ReadFile`, `WriteFile` или `DeviceIoControl`) блокируют выполнение программы до завершения операции. Хотя эта модель очень удобна в использовании, она не слишком эффективна. В промежутках времени между выполнением последовательных запросов на ввод/вывод устройство может простаивать, то есть, использоваться недостаточно полно. Другая проблема синхронного режима состоит в том, что поток выполнения напрасно расходует время при выполнении любой конкурирующей операции ввода/вывода. Например, в серверном приложении, одновременно обслуживающем множество клиентов, может быть предусмотрено создание отдельного потока выполнения для каждого сеанса. Эти потоки, которые большую часть времени простаивают, понапрасну расходуют память и могут создавать ситуации *пробуксовки потоков* (thread thrashing), когда множество потоков выполнения одновременно возобновляют работу по завершении ввода/вывода и начинают бороться за процессорное время, что приводит к увеличению переключений контекста в единицу времени и снижению масштабируемости.

Подсистема ввода/вывода Windows (включая драйверы устройств) внутренне действует в асинхронном режиме – программа может продолжать выполнение одновременно с операцией ввода/вывода. Практически все современные аппаратные устройства имеют асинхронную природу и не требуют постоянно опрашивать их, чтобы передать

данные или определить момент завершения операции ввода/вывода. Большинство устройств поддерживают возможность прямого доступа к памяти (Direct Memory Access, DMA) для передачи данных между устройством и ОЗУ компьютера, не требуя участия процессора в операции, и генерируют прерывание по завершении передачи данных. Синхронный режим ввода/вывода, который внутренне является асинхронным, поддерживается только на уровне приложений Windows.

В Win32 асинхронный ввод/вывод называется *перекрывающимся вводом/выводом* (overlapped I/O) (сравнение синхронного и перекрывающегося режимов ввода/вывода приводится на рис. 7.1). Когда приложение производит асинхронный запрос на выполнение операции ввода/вывода, Windows либо выполняет эту операцию немедленно, либо возвращает код состояния, указывающий, что операция ожидает выполнения. После этого поток выполнения может запустить другие операции ввода/вывода или выполнить некоторые вычисления. Программист имеет несколько способов организовать прием извещений о завершении операций ввода/вывода.

Рис. 7.1. Сравнение синхронного и асинхронного режимов ввода/вывода.

- Событие Win32: операция, ожидающая это событие, будет выполнена по завершении ввода/вывода.
- Вызов пользовательской функции с помощью механизма асинхронного вызова процедур (Asynchronous Procedure Call, APC): поток выполнения должен находиться *в состоянии ожидания извещения* (alertable wait).
- Прием извещений через порты завершения ввода/вывода (I/O Completion Ports): это обычно наиболее эффективный механизм. Мы подробно исследуем его далее в этой главе.

Примечание. *Некоторые устройства ввода/вывода (например, файл, открытый в небуферизованном режиме) могут давать дополнительные выгоды, если приложение сумеет обеспечить постоянное присутствие небольшого количества ожидающих запросов ввода/вывода. Для этого рекомендуется предварительно произвести несколько запросов на выполнение операций ввода/вывода и для каждого выполненного запроса производить новый запрос. Это обеспечит инициализацию следующей операции драйвером устройства в самые кратчайшие сроки, не ожидая, пока приложение выполнит следующий запрос. Но не переусердствуйте с объемом передаваемых данных, потому что при этом будут потребляться ограниченные ресурсы памяти ядра.*

Порты завершения ввода/вывода

Windows поддерживает эффективный механизм извещений о завершении асинхронных операций ввода/вывода под названием *порт завершения ввода/вывода* (I/O Completion Port, IOCP). В приложениях для .NET он доступен посредством метода `ThreadPool.BindHandle`. Этот механизм используется внутренними реализациями некоторых типов в .NET, выполняющих операции ввода/вывода: `FileStream`, `Socket`, `SerialPort`, `HttpListener`, `PipeStream` и некоторые каналы .NET Remoting.

Механизм IOCP (рис. 7.2) связывается с несколькими дескрипторами ввода/вывода (сокетами, файлами и специализированными объектами драйверов устройств), открытыми в асинхронном режиме, и с определенным потоком выполнения. Как только операция ввода/вывода, связанная с таким дескриптором, завершится, Windows добавит извещение в соответствующий порт IOCP и передаст для обработки связанному с ним потоку выполнения. Использование пула потоков, обслуживающих извещения и возобновляющих выполнение потоков, инициализировавших асинхронные операции ввода/вывода, снижает количество переключений контекста в единицу времени и увеличивает использование процессора. Неудивительно, что высокопроизводительные серверы, такие как Microsoft SQL Server, используют порты завершения ввода/вывода.

Порт завершения создается вызовом функции Win32 API `CreateIoCompletionPort`, которой передается максимальное значение параллелизма (количество потоков), ключ завершения и необязательный дескриптор объекта ввода/вывода. Ключ завершения – это определяемое пользователем значение, которое служит для идентификации различных дескрипторов ввода/вывода. С одним и тем же портом IOCP можно связать несколько дескрипторов, повторно вызывая

функцию `CreateIoCompletionPort` и передавая ей дескриптор существующего порта завершения.

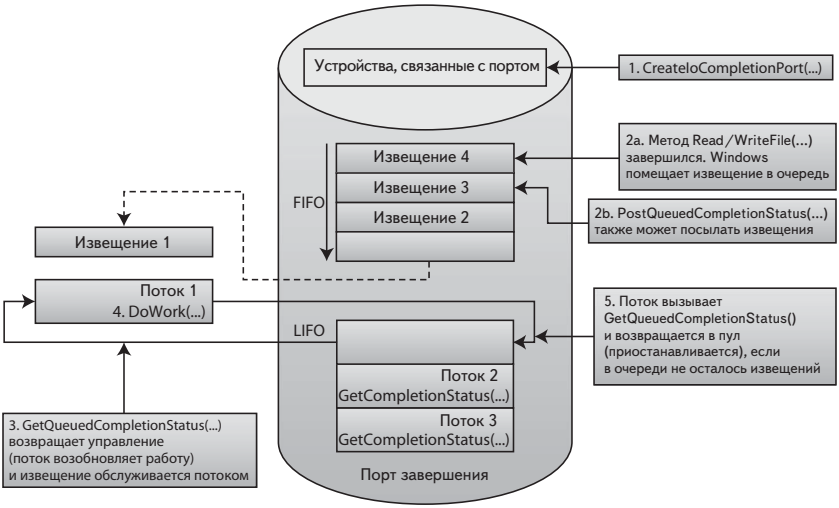

Рис. 7.2. Организация и принцип действия порта завершения ввода/вывода.

Чтобы установить связь с указанным портом IOCP, пользовательские потоки выполнения вызывают функцию `GetCompletionStatus` и ожидают ее завершения. В каждый конкретный момент времени поток выполнения может быть связан только с одним портом IOCP. Вызов функции `GetQueuedCompletionStatus` блокирует выполнение потока, пока не появится соответствующее извещение (или не истечет предельное время ожидания), после чего возвращает информацию о завершившейся операции ввода/вывода, такую как количество переданных байтов, ключ завершения и структуру асинхронной операции ввода/вывода. Если в момент появления извещения все потоки, связанные с портом ввода/вывода, окажутся заняты (то есть, не останется потоков, ожидающих в вызове `GetQueuedCompletionStatus`), механизм IOCP создаст новый поток выполнения, вплоть до максимального значения параллелизма. Если поток вызвал `GetQueuedCompletionStatus` и очередь извещений не пуста, функция немедленно вернет управление, не блокируя поток в ядре операционной системы.

Примечание. *Механизм IOCP способен определить, что какой-то из «занятых» потоков фактически выполняет синхронный ввод/вывод, и запустить дополнительный поток, возможно превысив максимальное значение*

параллелизма. Извещения можно также посылать вручную, без выполнения ввода/вывода, вызовом функции `PostQueuedCompletionStatus`.

В следующем листинге демонстрируется пример использования `ThreadPool.BindHandle` с файловым дескриптором Win32. Сначала рассмотрим метод `TestIOCP`. Здесь вызывается функция `CreateFile`, которая является функцией механизма P/Invoke, используемой для открытия или создания файла или устройства. Для выполнения операций ввода/вывода в асинхронном режиме, необходимо передать функции флаг `EFileAttributes.Overlapped`. В случае успеха функция `CreateFile` возвращает файловый дескриптор Win32, который мы связываем с портом завершения ввода/вывода вызовом `ThreadPool.BindHandle`. Далее создается объект события, используемый для временного блокирования потока, инициировавшего операцию ввода/вывода, если таких операций оказывается слишком много (предел устанавливается константой `MaxPendingIos`).

Затем начинается цикл асинхронных операций записи. В каждой итерации создается буфер с данными для записи и структура `Overlapped`, содержащая смещение внутри файла (в данном примере запись всегда выполняется со смещением 0), дескриптор события, передаваемый по завершении операции (не используется механизмом IOCP), и необязательный пользовательский объект `IAsyncResult`, который можно использовать для передачи состояния в функцию завершения. Далее вызывается метод `Overlapped.Pack`, принимающий функцию завершения и буфер с данными. Он создает эквивалентную низкоуровневую структуру операции ввода/вывода, размещая ее в неуправляемой памяти, и закрепляет буфер с данными. Освобождение неуправляемой памяти, занимаемой низкоуровневой структурой, и открепление буфера должны выполняться вручную.

Если одновременно будет выполняться не слишком много операций ввода/вывода, мы вызываем `WriteFile`, передавая ему указанную низкоуровневую структуру. В противном случае мы ждем, пока не появится событие, указывающее, что количество ожидающих операций стало меньше верхнего предела.

Функция завершения `WriteComplete` вызывается потоком из пула потоков завершения ввода/вывода, как только операция будет выполнена. Ей передается указатель на низкоуровневую структуру асинхронного ввода/вывода, которую можно распаковать и преобразовать в управляемую структуру `Overlapped`.

```
using System;
using System.Threading;
```

```
using Microsoft.Win32.SafeHandles;
using System.Runtime.InteropServices;

[DllImport("kernel32.dll", SetLastError = true, CharSet = CharSet.Auto)]
internal static extern SafeFileHandle CreateFile(
    string lpFileName,
    EFileAccess dwDesiredAccess,
    EFileShare dwShareMode,
    IntPtr lpSecurityAttributes,
    ECreationDisposition dwCreationDisposition,
    EFileAttributes dwFlagsAndAttributes,
    IntPtr hTemplateFile);

[DllImport("kernel32.dll", SetLastError = true)]
[return: MarshalAs(UnmanagedType.Bool)]
static unsafe extern bool WriteFile(SafeFileHandle hFile, byte[] lpBuffer,
    uint nNumberOfBytesToWrite, out uint lpNumberOfBytesWritten,
    System.Threading.NativeOverlapped *lpOverlapped);

[Flags]
enum EFileShare : uint {
    None = 0x00000000,
    Read = 0x00000001,
    Write = 0x00000002,
    Delete = 0x00000004
}

enum ECreationDisposition : uint {
    New = 1,
    CreateAlways = 2,
    OpenExisting = 3,
    OpenAlways = 4,
    TruncateExisting = 5
}

[Flags]
enum EFileAttributes : uint {
    // Некоторые флаги опущены для экономии места
    Normal = 0x00000080,
    Overlapped = 0x40000000,
    NoBuffering = 0x20000000,
}

[Flags]
enum EFileAccess : uint {
    // Некоторые флаги опущены для экономии места
    GenericRead = 0x80000000,
    GenericWrite = 0x40000000,
}
```

```
static long _numBytesWritten;
static AutoResetEvent _waterMarkFullEvent; // "тормоз" для потока записи
static int _pendingIosCount;

const int MaxPendingIos = 10;

// Процедура завершения, вызывается потоками завершения ввода/вывода
static unsafe void WriteComplete(uint errorCode, uint numBytes,
                                 NativeOverlapped* pOVERLAP) {
    _numBytesWritten + = numBytes;
    Overlapped ovl = Overlapped.Unpack(pOVERLAP);

    Overlapped.Free(pOVERLAP);
    // Известить поток записи, что количество ожидающих операций
    // ввода/вывода уменьшилось до допустимого предела
    if (Interlocked.Decrement(ref _pendingIosCount) < MaxPendingIos)
        _waterMarkFullEvent.Set();
    }

static unsafe void TestIOCP() {
    // Открыть файл в асинхронном режиме
    var handle = CreateFile(@"F:\largefile.bin",
        EFileAccess.GenericRead | EFileAccess.GenericWrite,
        EFileShare.Read | EFileShare.Write,
        IntPtr.Zero, ECreationDisposition.CreateAlways,
        EFileAttributes.Normal | EFileAttributes.Overlapped, IntPtr.Zero);

    _waterMarkFullEvent = new AutoResetEvent(false);
    ThreadPool.BindHandle(handle);

    for (int k = 0; k < 1000000; k++) {
        byte[] fbuffer = new byte[4096];

        // Аргументы: смещение в файле нижнее/верхнее, дескриптор
        // события объект IAsyncResult
        Overlapped ovl = new Overlapped(0, 0, IntPtr.Zero, null);
        // CLR автоматически закрепит буфер
        NativeOverlapped* pNativeOVL = ovl.Pack(WriteComplete, fbuffer);
        uint numBytesWritten;

        // Проверить количество ожидающих операций ввода/вывода
        if (Interlocked.Increment(ref _pendingIosCount) < MaxPendingIos) {
            if (WriteFile(handle, fbuffer, (uint)fbuffer.Length,
                          out numBytesWritten, pNativeOVL))
            {
                // ввод/вывод завершлся синхронно
                _numBytesWritten + = numBytesWritten;
                Interlocked.Decrement(ref _pendingIosCount);
            } else {
                if (Marshal.GetLastWin32Error() ! = ERROR_IO_PENDING) {
```

```
                    return; // Ошибка
                }
            }
        } else {
            Interlocked.Decrement(ref _pendingIosCount);
            while (_pendingIosCount > = MaxPendingIos) {
                _waterMarkFullEvent.WaitOne();
            }
        }
    }
}
```

Подводя итоги, отметим, что при работе с высокопроизводительными устройствами ввода/вывода, применяйте асинхронные операции ввода/вывода с портами завершения, либо непосредственно, создавая и используя собственный порт завершения в неуправляемой библиотеке, либо связывая дескрипторы Win32 с портом завершения в .NET с помощью метода `ThreadPool.BindHandle`.

Пул потоков в .NET

Пул потоков выполнения в .NET можно с успехом использовать в самых разных целях, для достижения каждой из которых создаются потоки разных типов. В главе 6 мы познакомились с прикладным программным интерфейсом пула потоков, где пользовались им для распараллеливания вычислительных задач. Однако пулы потоков можно использовать и для решения задач другого рода.

- *Рабочие потоки* могут обрабатывать асинхронные вызовы пользовательских делегатов (например, `BeginInvoke` или `ThreadPool.QueueUserWorkItem`).
- *Потоки завершения ввода/вывода* могут обслуживать уведомления, поступающие от глобального порта IOCP.
- *Потоки ожидания* могут обеспечивать ожидание наступления зарегистрированных событий, позволяя организовать ожидание сразу нескольких событий в одном потоке (с помощью `WaitForMultipleObjects`), вплоть до верхнего предела Windows (`MAXIMUM_WAIT_OBJECTS = 64`). Прием ожидания событий используется для организации асинхронного ввода/вывода без применение портов завершения.
- *Потоки-таймеры*, ожидающие, пока истекут сразу несколько таймеров.
- *Потоки-регуляторы* (gate threads) контролируют использование процессора потоками из пула, а также изменяют коли-

чество потоков (в установленных пределах) для достижения наивысшей производительности.

Примечание. *Имеется возможность инициировать операции ввода/вывода, которые кажутся асинхронными, но таковыми не являются. Например, вызов* `ThreadPool.QueueUserWorkItem` *делегата с последующим выполнением синхронной операции ввода/вывода не является по-настоящему асинхронной операцией и такое решение ничем не лучше выполнения той же операции в обычном потоке выполнения.*

Копирование памяти

Нередко физическое устройство ввода/вывода возвращает буфер с данными, который копируется снова и снова, пока не приложение не завершит его обработку. Подобное копирование может отнимать значительную долю вычислительной мощности процессора, поэтому его следует избегать, чтобы обеспечить максимальную пропускную способность. Далее мы рассмотрим несколько ситуаций, когда принято копировать данные, и познакомимся с приемами, позволяющими избежать этого.

Неуправляемая память

Работать с буфером, находящимся в неуправляемой памяти, в .NET работать значительно сложнее, чем с управляемым массивом `byte[]`, поэтому программисты, в поисках наиболее простого пути, часто копируют буфер в управляемую память.

Если применяемые вами функции или библиотеки позволяют явно указывать буфер в памяти или передавать им свою функцию обратного вызова для выделения буфера, распределите управляемый буфер и закрепите его в памяти, чтобы к нему можно было обращаться и по указателю, и по управляемой ссылке. Если буфер достаточно велик (>85 000 байт), он будет создан в куче больших объектов (Large Object Heap), поэтому старайтесь повторно использовать уже имеющиеся буферы. Если повторное использование буфера осложнено неопределенностью срока жизни объекта, применяйте пулы памяти, как описывается в главе 8.

В других случаях, когда функции или библиотеки сами выделяют память (неуправляемую) для буферов, вы можете обращаться к этой памяти непосредственно по указателю (из небезопасного кода) или используя классы-обертки, такие как `UnmanagedMemoryStream` и `UnmanagedMemoryAccessor`. Однако, если необходимо передать буфер

некоторому коду, который оперирует только массивами `byte[]` или строковыми объектами, копирование может оказаться неизбежным.

Даже если вам не удается избежать копирования памяти и некоторые или большинство ваших данных фильтруется на ранних этапах, излишнего копирования можно избежать, проверив необходимость данных перед их копированием.

Экспортирование части буфера

Как описывается в главе 8, программисты иногда полагают, что массивы `byte[]` содержат только необходимые данные, от начала и до конца, вынуждая вызывающий код разбивать буфер (выделять память для нового массива `byte[]` и копировать только необходимые данные). Эту ситуацию часто можно наблюдать в реализациях стеков протоколов. Эквивалентный неуправляемый код, напротив, может принимать простой указатель, не зная даже, указывает ли он на начало фактического буфера или в его середину, и параметр длины буфера, позволяющий определить, где находится конец обрабатываемых данных.

Чтобы избежать ненужного копирования памяти, организуйте прием смещения и длины везде, где принимаете параметр `byte[]`. Используйте параметр длины вместо свойства `Length` массива, а значение смещения добавляйте к текущим индексам.

Чтение вразброс и запись со слиянием

Чтение вразброс и запись со слиянием – это возможность, поддерживаемая ОС Windows, выполнять чтение в несмежные области или записывать данные из несмежных областей, как если бы они занимали непрерывный участок памяти. Данная функциональность в Win32 API предоставляется в виде функций `ReadFileScatter` и `WriteFileGather`. Библиотека сокетов Windows также поддерживает возможность чтения вразброс и записи со слиянием, предоставляя собственные функции: `WSASend`, `WSARecv` и другие.

Чтение вразброс и запись со слиянием могут пригодиться в следующих ситуациях.

- Когда в каждом пакете имеется заголовок фиксированного размера, предшествующий фактическим данным. Чтение

вразброс и запись со слиянием позволят вам избежать необходимости копировать заголовки всякий раз, когда потребуется получить непрерывный буфер.

- Когда желательно избавиться от лишних накладных расходов на обращения к системным вызовам, при выполнении ввода/вывода с несколькими буферами.

В сравнении с функциями `ReadFileScatter` и `WriteFileGather`, требующими, чтобы каждый буфер в точности соответствовал размеру одной страницы, а дескриптор был открыт в асинхронном и небуферизованном режиме (что является еще большим ограничением), функции чтения вразброс и записи со слиянием на основе сокетов выглядят более практичными, потому что не имеют этих ограничений. Фреймворк .NET Framework поддерживает чтение вразброс и запись со слиянием для сокетов посредством перегруженных методов `Socket.Send` и `Socket.Receive`, не экспортируя при этом универсальные функции чтения/записи.

Пример использования функций чтения вразброс и записи со слиянием можно найти в классе `HttpWebRequest`. Он объединяет HTTP-заголовки с фактическими данными, не прибегая к созданию непрерывного буфера для их хранения.

Файловый ввод/вывод

Обычно файловые операции ввода/вывода выполняются через кеш файловой системы, дающий некоторые выгоды с точки зрения производительности: кеширование недавно использованных данных, опережающее чтение (предварительное чтение данных с диска), отложенная запись (асинхронная запись на диск) и объединение операций записи маленьких порций данных. Подсказывая Windows ожидаемый шаблон доступа к файлам, можно получить дополнительный прирост производительности. Если ваше приложение выполняет асинхронный ввод/вывод и способно решать некоторые проблемы буферизации, тогда полный отказ от использования механизма кеширования может оказаться более эффективным решением.

Управление кешированием

Создавая или открывая файлы, программисты передают функции `CreateFile` флаги и атрибуты, часть из которых оказывает влияние на поведение механизма кеширования.

- Флаг `FILE_FLAG_SEQUENTIAL_SCAN` указывает, что обращение к файлу будет осуществляться последовательно, возможно с пропуском некоторых частей, а произвольный доступ маловероятен. В результате диспетчер кеша будет выполнять опережающее чтение с заглядыванием дальше обычного.
- Флаг `FILE_FLAG_RANDOM_ACCESS` указывает, что доступ к файлу будет осуществляться в произвольном порядке. В этом случае диспетчер кеша будет выполнять чтение с небольшим опережением, из-за снижения вероятности, что данные, прочитанные с опережением, действительно потребуются приложению.
- Флаг `FILE_ATTRIBUTE_TEMPORARY` указывает, что файл является временным, поэтому фактически операции записи на физический носитель (чтобы предотвратить потерю данных) можно отложить.

В .NET эти параметры поддерживаются (кроме последнего) с помощью перегруженного конструктора `FileStream`, принимающего параметр типа перечисления `FileOptions`.

Внимание. *Произвольный доступ отрицательно сказывается на производительности, особенно при работе с дисковыми устройствами, так как при этом возникает необходимость перемещать головки. В процессе развития технологий, пропускная способность дисков увеличивалась только за счет увеличения плотности хранения данных, но не за счет уменьшения задержек. Современные диски способны переупорядочивать выполнение запросов при произвольном доступе, чтобы уменьшить общее время, затрачиваемое на перемещение головок. Этот прием называется аппаратная установка очередности команд (Native Command Queuing, NCQ). Для большей эффективности этого приема контроллеру диска необходимо отправить сразу несколько запросов на ввод/вывод. Иными словами, если это возможно, старайтесь иметь сразу несколько ожидающих асинхронных запросов ввода/вывода.*

Небуферизованный ввод/вывод

Операции небуферизованного ввода/вывода всегда выполняются без привлечения кеша. Такой подход имеет свои достоинства и недостатки. Как и в случае использования приема управления кешем, небуферизованный режим ввода/вывода включается с помощью параметра «флагов и атрибутов» в процессе создания файла, но .NET не обеспечивает доступ к этой возможности.

- Флаг `FILE_FLAG_NO_BUFFERING` отключает кеширование операций чтения и записи, но никак не влияет на кеширование, выполняемое контроллером диска. Это позволяет избежать

копирования (из пользовательского буфера в кеш) и «загрязнения» кеша (заполнения кеша ненужными данными и вытеснение нужных). Однако небуферизованные операции чтения и записи должны придерживаться требований, касающихся выравнивания. Следующие параметры должны быть равны или кратны размеру дискового сектора: размер одной передачи, смещение в файле и адрес буфера в памяти. Обычно дисковый сектор имеет размер 512 байт. В новейших дисковых устройствах повышенной емкости размер сектора составляет 4096 байт, но они могут работать в режиме совместимости, эмулируя сектора размером 512 байт (за счет снижения производительности).

- Флаг `FILE_FLAG_WRITE_THROUGH` указывает диспетчеру кеша, что он должен сразу же выталкивать из кеша записываемые данные (если флаг `FILE_FLAG_NO_BUFFERING` не установлен) и сообщает контроллеру диска, что он должен выполнять запись на физический носитель немедленно, не сохраняя данные в промежуточном аппаратном кеше.

Опережающее чтение увеличивает производительность за счет более полного использования диска, даже когда приложение выполняет чтение в синхронном режиме с задержками между операциями. Правильное определение, какую часть файла приложение запросит в следующий раз, зависит от Windows. Отключая буферизацию, вы также отключаете опережающее чтение, и должны поддерживать высокую занятость дискового устройства выполняя несколько перекрывающихся операций ввода/вывода.

Запись с задержкой также увеличивает производительность приложений, выполняющих синхронные операции записи, создавая иллюзию, что запись на диск выполняется очень быстро. Приложение сможет улучшить использование процессора, из-за блокировок на более короткие интервалы времени. С отключенной буферизацией продолжительность операций записи будет равна полному интервалу времени, необходимому для завершения записи данных на диск. Поэтому применение асинхронного режима ввода/вывода при отключенной буферизации становится еще более важным.

Сети

Доступ к сети является одной из фундаментальных особенностью современных приложений. Серверные приложения, обрабатываю-

щие запросы от клиентов, стремятся максимально увеличить масштабируемость и пропускную способность, чтобы обслуживать как можно больше клиентов и как можно быстрее, тогда как для клиентов важно уменьшить сетевые задержки или ослабить их влияние на производительность. В этом разделе приводятся советы и рекомендации по увеличению производительности сетевых операций.

Сетевые протоколы

Особенности устройства сетевого протокола прикладного уровня (уровень 7 в модели OSI) могут оказывать существенное влияние на производительность. В этом разделе мы исследуем некоторые приемы оптимизации, обеспечивающие более полное использование пропускной способности сети и уменьшение накладных расходов.

Конвейерный режим

Если протокол не поддерживает конвейерную обработку, после отправки запроса серверу клиент вынужден ждать получения ответа, прежде чем отправить следующий запрос. Такие протоколы не дают возможность максимально использовать пропускную способность сети, потому что в промежутках между отправкой запроса и получением ответа сеть простаивает. Напротив, при наличии поддержки конвейерного режима обработки, клиент может продолжать посылать новые запросы, не дожидаясь, пока сервер обработает предыдущие. Еще лучше выглядит ситуация, когда сервер в состоянии отвечать на запросы не в порядке их поступления, отвечая на простые запросы максимально быстро и откладывая обработку более сложных и тяжелых запросов, с вычислительной точки зрения, на будущее.

Возможность конвейерной обработки играет важную роль, потому что, несмотря на увеличивающуюся пропускную способность Интернета, задержки уменьшаются не так быстро, из-за того, что скорость света имеет определенный предел.

Примером протокола с поддержкой конвейерной обработки может служить протокол HTTP 1.1, но она часто отключена по умолчанию на большинстве серверов и в веб-браузерах из-за проблем совместимости. Протокол Google SPDY (экспериментальный HTTP-подобный протокол, поддерживаемый веб-браузерами Chrome и Firefox), а также некоторые HTTP-серверы и готовящийся к выходу протокол HTTP 2.0, включают конвейерную обработку по умолчанию.

Потоковый режим

Потоковый режим может использоваться не только для передачи видео- и аудиосодержимого, но и для обмена сообщениями. При работе в потоковом режиме приложение начинает отправлять данные еще до того, как завершится их подготовка. Потоковый режим уменьшает задержки и обеспечивает более полное использование пропускной способности сети.

Например, если в ответ на запрос серверное приложение извлекает информацию из базы данных, оно может прочитать все данные сразу в один объект `DataSet` (что может потребовать выделить большой объем памяти) или извлекать их по одной записи с помощью `DataReader`. В первом случае сервер вынужден ждать получения всего объема данных, прежде чем начать отправку ответа клиенту, тогда как во втором он может начать отправку после получения первой же записи.

Объединение сообщений

Отправка данных маленькими фрагментами является напрасной тратой пропускной способности сети. Заголовки протоколов Ethernet, IP и TCP/UDP имеют довольно большие размеры, из-за чего объем полезных данных в пакете оказывается невелик, поэтому, даже при максимальном использовании пропускной способности сети, большая ее часть может отводиться на передачу заголовков, а не полезной информации. Кроме того, сама ОС Windows вносит свои накладные расходы, не зависящие или почти не зависящие от размера фрагмента данных. Протокол может смягчать этот недостаток, поддерживая возможность объединения нескольких запросов. Например, протокол службы доменных имен (Domain Name Service, DNS) дает возможность клиенту разрешать несколько доменных имен в одном запросе.

Многословные протоколы

Иногда клиент не в состоянии посылать запросы в конвейерном режиме, даже если протокол позволяет это, потому что содержимое следующего запроса может зависеть от содержимого ответа на предыдущий запрос.

Представьте сеанс использования «многословного» протокола (chatty protocol). Когда вы пытаетесь открыть веб-страницу, браузер подключается к веб-серверу по протоколу TCP, отправляет запрос HTTP GET и принимает HTML-страницу. Затем браузер анализирует

полученную страницу, выявляет в ней ссылки на сценарии JavaScript, таблицы стилей CSS и изображения, и загружает их отдельно. Далее он выполняет сценарии JavaScript, которые могут инициировать загрузку дополнительных данных. В общем случае клиент не может заранее знать, какое содержимое еще потребуется для отображения страницы.

Чтобы ослабить отрицательное влияние этой проблемы, сервер может подсказывать клиенту адреса URL, откуда потребуется загрузить дополнительную информацию, необходимую для отображения страницы, и даже отправить эту информацию, не дожидаясь, пока клиент запросит ее.

Кодирование и избыточность сообщений

Пропускная способность сети – часто ограниченный ресурс, и применение расточительных форматов сообщений приведет лишь к напрасному расходованию этого ресурса. Ниже перечислено несколько рекомендаций по оптимизации форматов сообщений.

- Не передавайте одни и те же данные снова и снова и используйте небольшие заголовки.
- Используйте экономные кодировки для представления данных. Например, строки можно передавать в кодировке UTF-8, вместо UTF-16. Двоичные форматы могут оказаться во много раз более компактными, чем текстовые форматы. Если это возможно, избегайте инкапсуляции данных в такие кодировки, как Base64.
- Используйте сжатие для данных, хорошо поддающихся сжатию, таких как текст. Не применяйте сжатие к данным, плохо поддающимся сжатию, таким как уже сжатые видео- или аудиоданные и изображения.

Сетевые сокеты

Прикладной интерфейс сокетов является стандартным инструментом для работы с сетевыми протоколами, такими как TCP и UDP. Первоначально интерфейс сокетов был реализован в операционной системе BSD UNIX и со временем превратился в стандарт практически для всех операционных систем, иногда дополненный патентованными расширениями, такими как Microsoft WinSock. В Windows поддерживается несколько способов организации ввода/вывода через сокеты: с блокировкой, без блокировки с опросом и асинхронно. Выбор

наилучшей модели ввода/вывода и параметров сокетов позволяют добиться максимальной пропускной способности, минимальных задержек и наилучшей масштабируемости. В этом разделе рассказывается о некоторых способах оптимизации операций с использованием механизма Windows Sockets.

Асинхронные сокеты

Асинхронный ввод/вывод в .NET поддерживается посредством класса `Socket`. Всего существует две группы асинхронных методов: `BeginXXX` и `XXXAsync`, где под `XXX` подразумеваются `Accept`, `Connect`, `Receive`, `Send` и другие операции. Первая группа использует механизм .NET Thread Pool для организации ожидания завершения асинхронных операций ввода/вывода, а вторая – механизм порта завершения ввода/вывода, более производительный и масштабируемый. Вторая группа методов впервые была реализована в версии .NET Framework 2.0 SP1.

Буферы сокетов

Объекты сокетов имеют свойства `ReceiveBufferSize` и `SendBufferSize`, позволяющие определять размеры буферов, выделяемых стеком протоколов TCP/IP (в пространстве памяти операционной системы). По умолчанию оба получают значение 8192 байт. Приемный буфер используется для хранения принятых данных, пока не прочитанных приложением. Выходной буфер используется для хранения данных, отправленных приложением, но получение которых еще не было подтверждено принимающей стороной. Если возникнет необходимость повторно отправить данные, они будут отправлены из выходного буфера.

Когда приложение читает данные из сокета, оно опустошает приемный буфер на объем прочитанных данных. Когда приемный буфер опустеет, вызывающая программа либо блокируется, либо запускает механизм ожидания, в зависимости от режима ввода/вывода – синхронного или асинхронного.

Когда приложение выводит данные в сокет, они записываются в выходной буфер, при этом приложение не блокируется, пока выходной буфер не заполнится или пока не заполнится приемный буфер на принимающей стороне. С каждым подтверждением приема, принимающая сторона сообщает объем свободного пространства в приемном буфере.

Для соединений с большой пропускной способностью и большими задержками, таких как спутниковые каналы связи, размеры по умолчанию буферов могут оказаться слишком маленькими. Отправляющая сторона быстро заполнит свой выходной буфер, и вынуждена будет ждать подтверждения, которое может прийти с опозданием из-за значительных задержек. Во время ожидания канал связи простаивает, и участники соединения используют лишь часть доступной пропускной способности.

В высоконадежных сетях идеальным является размер буфера, являющийся произведением пропускной способности на задержку. Например, для соединения с пропускной способностью 100 Мбит/сек с 5-миллисекундной задержкой, идеальным будет размер буфера (100 000 000 / 8) × 0.005 = 62 500 байт. При наличии потерь пакетов это значение следует уменьшить.

Алгоритм Нейгла

Как уже упоминалось выше, чем меньше размер пакетов, тем выше накладные расходы на их транспортировку, потому что объем заголовков может оказаться больше объема полезных данных. Алгоритм Нейгла (Nagle) позволяет увеличить производительность сокетов TCP посредством объединения операций записи в полноценные пакеты данных. Однако за эту услугу приходится платить задержками в отправке данных. Приложения, чувствительные к задержкам, должны отключать алгоритм Нейгла установкой свойства `Socket.NoDelay` в значение `true`. Хорошо продуманные приложения обычно посылают данные большими блоками и не получат выгод от использования алгоритма Нейгла.

Зарегистрированный ввод/вывод

Зарегистрированный ввод/вывод (Registered I/O, RIO) – это новое расширение механизма WinSock, доступное в Windows Server 2012, реализующее весьма эффективный механизм регистрации буферов и извещений. RIO устраняет наиболее существенные источники накладных расходов в подсистеме ввода/вывода Windows:

- опробование пользовательского буфера (проверка прав доступа к странице), блокировка и разблокировка (гарантирующие нахождение буферов в ОЗУ);
- поиск дескрипторов (преобразование значений типа `HANDLE` в указатели на объекты ядра);

- выполнение системных вызовов (например, чтобы извлечь из очереди извещение о завершении ввода/вывода).

Это – «налог», который приходится платить за изоляцию приложения от операционной системы и других приложений, гарантирующую безопасность и надежность. Без механизма RIO вам придется платить «налог» за каждый вызов, который при большом объеме ввода/вывода становится существенным. Напротив, используя RIO, вы платите «налог» только один раз, в процессе инициализации.

Механизм RIO требует регистрации буферов, которые запираются в физической памяти, пока не будут освобождены приложением (в процессе завершения приложения или подсистемы). Поскольку выделенные буферы постоянно находятся в ОЗУ, Windows может пропустить процедуру опробования, блокировки и разблокировки в каждом вызове.

Очереди запросов и завершения механизма RIO размещаются в памяти процесса и доступны ему, а это означает, что для проверки очереди и извлечения извещений из очереди больше не требуется обращаться к системным вызовам.

RIO поддерживает три механизма извещений:

- по опросу: имеет самую низкую задержку, но требует выделения логического процессора для опроса сетевых буферов;
- порты завершения ввода/вывода;
- события Windows.

На момент написания этих строк фреймворк .NET Framework не обеспечивал доступ к механизму RIO, но его можно использовать с применением стандартных механизмов взаимодействий .NET (обсуждаются в главе 8).

Сериализация и десериализация данных

Сериализация (serialization) – это операция преобразования объекта в формат, пригодный для записи на диск или отправки в сеть. Десериализация (de-serialization) – это операция восстановления объекта из сериализованного представления. Например, хеш-таблица может быть сериализована в массив записей ключ/значение.

Тестирование производительности средств сериализации

В состав .NET Framework входит несколько универсальных средств сериализации, с помощью которых можно выполнять сериализацию и десериализацию пользовательских типов. В этом разделе оцениваются достоинства и недостатки каждого из них и приводятся результаты тестирования производительности и компактности получаемых данных.

Для начала познакомимся с имеющимися средствами сериализации:

- `System.Xml.Serialization.XmlSerializer`:
  - сериализует текстовые и двоичные данные в формат XML;
  - сериализует дочерние объекты, но не поддерживает циклические ссылки;
  - сериализует только общедоступные поля и свойства, за исключением тех, что явно исключены из сериализации;
  - для повышения эффективности использует механизм рефлексии (Reflection) только единожды, когда генерирует код сборки сериализации, – для предварительного создания сборки сериализации можно использовать инструмент *sgen.exe*;
  - позволяет настраивать схему XML;
  - требует знания всех типов, участвующих в сериализации: эта информация выводится автоматически, за исключением случаев, когда используются унаследованные типы.
- `System.Runtime.Serialization.Formatters.Binary.BinaryFormatter`:
  - сериализует в закрытый двоичный формат, который распознается только классом `BinaryFormatter`;
  - используется механизмом .NET Remoting, но может использоваться как самостоятельный инструмент сериализации;
  - сериализует не только общедоступные поля;
  - распознает циклические ссылки;
  - не требует предварительного знания типов сериализуемых объектов;
  - определения сериализуемых типов должны быть отмечены атрибутом `[Serializable]`.
- `System.Runtime.Serialization.Formatters.Soap.SoapFormatter`:
  - своими возможностями напоминает `BinaryFormatter`, но сериализует в формат SOAP XML, более пригодный для обме-

на данными с другими системами, но менее компактный;
- не поддерживает обобщенные типы и обобщенные коллекции и, соответственно, не рекомендуется к использованию в последних версиях .NET Framework.

- `System.Runtime.Serialization.DataContractSerializer`:
  - сериализует текстовые и двоичные данные в формат XML;
  - используется механизмом WCF, но может использоваться как самостоятельный инструмент сериализации;
  - сериализует типы и поля, отмеченные атрибутами `[DataContract]` и `[DataMember]`: если класс отмечен атрибутом `[Serializable]`, сериализоваться будут все поля этого класса;
  - требует знания всех типов, участвующих в сериализации: эта информация выводится автоматически, за исключением случаев, когда используются унаследованные типы;
- `System.Runtime.Serialization.NetDataContractSerializer`:
  - напоминает `DataContractSerializer`, но встраивает в сериализованные данные информацию о типе;
  - не требует предварительного знания типов сериализуемых объектов;
  - требует доступа к сборкам, содержащим объявления сериализуемых типов.
- `System.Runtime.Serialization.DataContractJsonSerializer`:
  - напоминает `DataContractSerializer`, но сериализует в формат JSON вместо XML.

На рис. 7.3 представлены результаты тестирования производительности средств сериализации, описанных выше. Некоторые из них тестировались дважды – для сериализации в текстовый и двоичный формат. В ходе тестирования выполнялась сериализация и десериализация очень сложного графа объектов, включающего 3600 экземпляров пяти типов и имеющего древовидную структуру. Каждый тип имел поля типов `string` и `double`, а также массивы элементов этих типов. Циклические ссылки отсутствовали в тестовом наборе данных, потому что не все средства сериализации поддерживают их; однако те, что поддерживают их, обрабатывали данные существенно медленнее при их наличии. Результаты тестирования, представленные здесь, были получены в версии .NET Framework 4.5 RC, показывающей немного более высокую производительность в сравнении с версией .NET Framework 3.5 в тестах сериализации в двоичный формат XML, но в остальных тестах производительность практически ничем не отличалась.

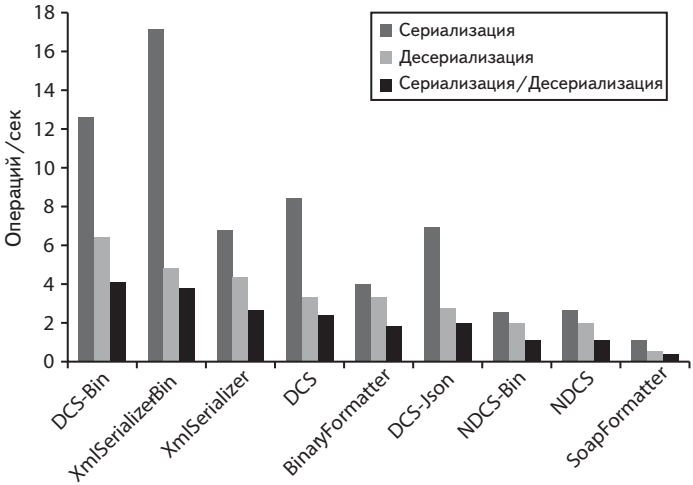

Рис. 7.3. Результаты тестирования производительности средств сериализации в операций/сек.

Из результатов тестирования можно заключить, что самыми быстрыми являются `DataContractSerializer` и `XmlSerializer`, при сериализации в двоичный формат XML.

Затем мы сравнили эти же средства по объему данных, полученных в результате сериализации (рис. 7.4). Здесь можно выделить несколько инструментов, очень близких по этому показателю. Скорее всего это обусловлено тем, что наибольший объем данных был сосредоточен в строках, которые сериализуются в одно и то же представление всеми описываемыми средствами.

Наиболее компактное представление сериализованных данных дает `DataContractJsonSerializer`. За ним близко следуют `XmlSerializer` и `DataContractSerializer`, при сериализации в двоичный формат. Самое удивительное, пожалуй, что класс `BinaryFormatter` был превзойден по производительности большинством других средств сериализации.

Сериализация объектов DataSet

Объект `DataSet` – это кеш в памяти для хранения данных, извлеченных из базы данных с помощью `DataAdapter`. Он содержит коллекцию объектов `DataTable`, определяющую структуру базы данных и записи с данными, каждая из которых содержит коллекцию сериализованных

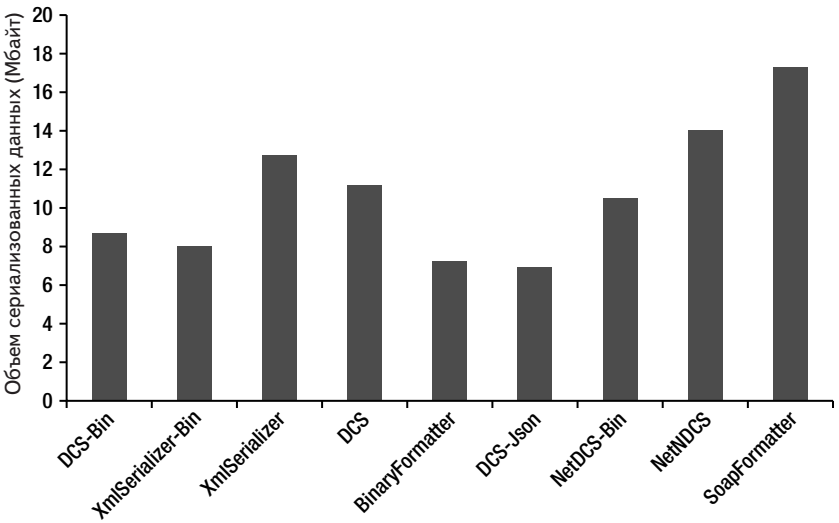

Рис. 7.4. Сравнение по объему сериализованных данных.

объектов. Объекты `DataSet` чрезвычайно сложны, занимают большие объемы памяти, и требуют значительных вычислительных затрат на сериализацию. Однако многие приложения передают их между своими уровнями. Ниже приводится несколько советов, следование которым поможет уменьшить накладные расходы, связанные с сериализацией.

- Вызывайте метод `DataSet.ApplyChanges` перед сериализацией `DataSet`. Объекты `DataSet` хранят оригинальные и измененные значения. Если вам не требуется сериализовать старые значения, вызывайте `ApplyChanges`, чтобы отбросить их.
- Сериализуйте только те объекты `DataTables`, которые действительно необходимы. Если `DataSet` содержит ненужные вам таблицы, подумайте о возможности скопировать нужные таблицы в новый объект `DataSet` и сериализовать его.
- Используйте вместо имен столбцов более короткие псевдонимы (с помощью ключевого слова `As`), чтобы уменьшить объем сериализованных данных. Например, взгляните на следующую инструкцию SQL:

```
SELECT EmployeeID As I, Name As N, Age As A
```

Windows Communication Foundation

Фреймворк Windows Communication Foundation (WCF), появившийся в версии .NET 3.0, быстро стал стандартом де-факто организации сетевых взаимодействий в приложениях для .NET. Он поддерживает огромное множество сетевых протоколов и настроек, и непрерывно расширяется с каждой новой версией .NET. В этом разделе рассказывается о приемах оптимизации фреймворка WCF.

Пороговые значения

Фреймворк WCF, в особенности до выхода версии .NET Framework 4.0, имел довольно консервативные пороговые значения. Основная их цель – обеспечить защиту от атак типа «отказ в обслуживании» (Denial of Service, DoS), но, к сожалению, в реальном мире они часто оказываются слишком строгими.

Настройки ограничений можно изменить, отредактировав раздел `system.serviceModel` либо в файле *app.config* (для обычных приложений), либо в файле *web.config* – для приложений ASP.NET:

```
<system.serviceModel>
  <behaviors>
    <serviceBehaviors>
      <behavior>
        <serviceThrottling>
          <serviceThrottling maxConcurrentCalls = "16"
            maxConcurrentSessions = "10" maxConcurrentInstances = "26" />
```

Другой способ изменить эти параметры – настроить свойства объекта `ServiceThrottling` на этапе создания службы:

```
Uri baseAddress = new Uri("http://localhost:8001/Simple");
ServiceHost serviceHost = new ServiceHost(typeof(CalculatorService),
                                          baseAddress);

serviceHost.AddServiceEndpoint(
    typeof(ICalculator),
    new WSHttpBinding(),
    "CalculatorServiceObject");

serviceHost.Open();

IChannelListener icl = serviceHost.ChannelDispatchers[0].Listener;
ChannelDispatcher dispatcher = new ChannelDispatcher(icl);
ServiceThrottle throttle = dispatcher.ServiceThrottle;
```

```
throttle.MaxConcurrentSessions = 10;
throttle.MaxConcurrentCalls = 16;
throttle.MaxConcurrentInstances = 26;
```

Давайте разберемся, что означают все эти параметры.

- `maxConcurrentSessions` ограничивает количество сообщений, одновременно обрабатываемых узлом службы `ServiceHost`. При превышении этого предела сообщения будут помещаться в очередь. В .NET 3.5 по умолчанию используется значение 10, а в .NET 4 это значение получается, как произведение числа процессоров на 100.
- `maxConcurrentCalls` ограничивает количество объектов `InstanceContext`, обрабатываемых одновременно узлом службы `ServiceHost`. Запросы на создание дополнительных экземпляров помещаются в очередь и обрабатываются, когда количество объектов `InstanceContext` оказывается ниже указанного уровня. В .NET 3.5 по умолчанию используется значение 16 for .NET 3.5, а в .NET 4 это значение получается, как произведение числа процессоров на 16.
- `maxConcurrentInstances` ограничивает количество сеансов, одновременно обслуживаемых узлом службы `ServiceHost`. Служба будет продолжать принимать соединения сверх указанного ограничения, но активными будут только каналы ниже этой границы (сообщения будут читаться из каналов). В .NET 3.5 по умолчанию используется значение 26, а в .NET 4 это значение получается, как произведение числа процессоров на 116.

Еще одно важное ограничение – количество параллельных соединений на каждый хост, которое по умолчанию равно двум. Если ваше приложение ASP.NET обращается к внешней службе WCF, это ограничение может оказаться узким местом. Ниже приводится пример конфигурации, где определяется это ограничение:

```
<system.net>
  <connectionManagement>
    <add address = "*" maxconnection = "100" />
  </connectionManagement>
</system.net>
```

Модель обработки

При разработке службы WCF вам потребуется определить ее модель активации и параллельной обработки. Сделать это можно с помощью свойств `InstanceContextMode` и `ConcurrencyMode` атрибута

ServiceBehavior, соответственно. Свойство InstanceContextMode может принимать следующие значения:

- PerCall – объект экземпляра службы создается для каждого вызова;
- PerSession (по умолчанию) – объект экземпляра службы создается для каждого сеанса; если канал не поддерживает сеансы, это значение интерпретируется как PerCall;
- Single – для всех вызовов повторно используется один и тот же объект экземпляра службы.

Свойство ConcurrencyMode может принимать следующие значения.

- Single (по умолчанию) – объект службы является однопоточным и не поддерживает реентерабельность. Если InstanceContextMode устанавливается в значение Single, когда уже идет обслуживание запроса, новые запросы будут становиться в очередь, ожидая обработки.
- Reentrant – объект службы является однопоточным, но поддерживает реентерабельность. Когда служба вызывает другую службу, она может вызываться реентерабельно. В этом случае вам придется позаботиться о сохранении целостности состояния объекта перед обращением к другой службе.
- Multiple – объект службы является многопоточным, но не гарантирует синхронизацию между потоками, поэтому служба должна предусматривать средства синхронизации, чтобы обеспечить целостность состояния.

Не используйте значения Single и Reentrant а свойстве ConcurrencyMode в сочетании со значением Single в свойстве InstanceContextMode. При использовании значения Multiple в свойстве ConcurrencyMode, используйте блокировки с высокой степенью детализации, чтобы уменьшить вероятность конкуренции между потоками.

Фреймворк WCF вызывает объекты службы из пула потоков завершения ввода/вывода, описанного выше в этой главе. Если в ходе обработки запросов понадобится выполнять синхронные операции ввода/вывода или ждать некоторого события, вам может потребоваться увеличить количество потоков, что можно сделать, отредактировав раздел system.web в конфигурационном файле приложения для ASP.NET (см. ниже) или вызвав ThreadPool.SetMinThreads и ThreadPool.SetMaxThreads в обычном приложении.

```
<system.web>
  <processModel
```

```
    ...
    enable = "true"
    autoConfig = "false"
    maxWorkerThreads ="80"
    maxIoThreads = "80"
    minWorkerThreads = "40"
    minIoThreads = "40"
  />
```

Кеширование

Фреймворк WCF не имеет встроенной поддержки кеширования. Даже если ваша служба на основе фреймворка WCF выполняется под управлением IIS, она все равно не сможет использовать его кеш по умолчанию. Чтобы включить поддержку кеширования, отметьте свой класс WCF-службы атрибутом `AspNetCompatibilityRequirements`.

```
[AspNetCompatibilityRequirements(RequirementsMode =
    AspNetCompatibilityRequirementsMode.Allowed)]
```

Кроме того, включите совместимость с ASP.NET, отредактировав файл `web.config` и добавив следующий элемент в раздел `system.serviceModel`:

```
<serviceHostingEnvironment aspNetCompatibilityEnabled = "true" />
```

Начиная с версии .NET Framework 4.0 имеется возможность использовать новые типы `System.Runtime.Caching` для реализации поддержки кеширования. Они не зависят от сборки `System.Web` и поэтому могут использоваться не только в приложениях ASP.NET.

Асинхронные клиенты и серверы WCF

Фреймворк WCF позволяет выполнять асинхронные операции, как на стороне клиента, так и на стороне сервера. Каждая сторона может принимать независимое решение об использовании синхронных и асинхронных операций.

На стороне клиента поддерживается два способа асинхронного обращения к службе: на основе событий и с применением шаблона организации асинхронных взаимодействий в .NET. Модель на основе событий не совместима с каналами, созданными с помощью `ChannelFactory`. Чтобы получить возможность применить модель на основе событий, необходимо сгенерировать прокси-объект доступа к службе с помощью инструмента *svcutil.exe*, вызвав его с ключами `/async` и `/tcv:Version35`:

```
svcutil /n:http://Microsoft.ServiceModel.Samples,Microsoft.
ServiceModel.Samples
http://localhost:8000/servicemodelsamples/service/mex /async /
tcv:Version35
```

После этого прокси-объект можно использовать, как показано ниже:

```
// Асинхронные функции обратного вызова для отображения результатов.
static void AddCallback(object sender, AddCompletedEventArgs e) {
    Console.WriteLine("Add Result: {0}", e.Result);
}

static void Main(String[] args) {
    CalculatorClient client = new CalculatorClient();
    client.AddCompleted + =
        new EventHandler <AddCompletedEventArgs> (AddCallback);
    client.AddAsync(100.0, 200.0);
}
```

В случае выбора модели на основе интерфейса `IAsyncResult` следует создать прокси-объект, вызвав *svcutil.exe* с ключом `/async`, но без ключа `/tcv:Version35`. А затем реализовать функции обратного вызова и использовать методы `BeginXXX` прокси-объекта как показано ниже:

```
static void AddCallback(IAsyncResult ar) {
    double result = ((CalculatorClient)ar.AsyncState).EndAdd(ar);
    Console.WriteLine("Add Result: {0}", result);
}

static void Main(String[] args) {
    ChannelFactory <ICalculatorChannel> factory =
        new ChannelFactory <ICalculatorChannel> ();
    ICalculatorChannel channelClient = factory.CreateChannel();
    IAsyncResult arAdd = channelClient.BeginAdd(100.0, 200.0,
                                                AddCallback,
                                                channelClient);
}
```

На сервере асинхронный режим работы можно организовать за счет создания версий `BeginXX` и `EndXX` контрактных операций. У вас не должно быть других операций с теми же именами, без префиксов `Begin/End`, потому что фреймворк WCF будет стремиться использовать их. Следуйте этим соглашениям об именовании, потому что в WCF они являются обязательными.

Метод `BeginXX` должен принимать входные параметры и возвращать значение `IAsyncResult`, а сама операция ввода/вывода должна производиться асинхронно. Метод `BeginXX` должен быть снабжен

атрибутом `OperationContract` с параметром `AsyncPattern`, имеющим значение `true`.

Метод `EndXX` должен принимать параметр типа `IAsyncResult`, возвращать необходимое значение и иметь требуемые выходные параметры. Объект `IAsyncResult` (возвращаемый методом `BeginXX`) должен содержать всю необходимую информацию о возвращаемом результате.

Кроме всего прочего, версия WCF 4.5 поддерживает новый шаблон `async/await` на основе класса `Task` для организации асинхронного ввода/вывода в серверных и клиентских приложениях. Например:

```
// Асинхронная служба на основе класса Task
public class StockQuoteService : IStockQuoteService {
    async public Task<double> GetStockPrice(string stockSymbol) {
        double price = await FetchStockPriceFromDB();
        return price;
    }
}

// Асинхронный клиент на основе класса Task
public class TestServiceClient : ClientBase <IStockQuoteService>,
        IStockQuoteService
{
    public Task<double> GetStockPriceAsync(string stockSymbol) {
        return Channel.GetStockPriceAsync();
    }
}
```

Привязки

При проектировании служб на основе фреймворка WCF, очень важно правильно выбрать механизм привязки. Каждый из этих механизмов обладает своими особенностями и характеристиками производительности. Старайтесь выбирать как можно более простые привязки и не используйте больше их особенностей, чем это необходимо для работы службы. Многие особенности, такие как надежность, безопасность и поддержка аутентификации, привносят дополнительные накладные расходы, поэтому применяйте их, только когда без этого не обойтись.

Во взаимодействиях между процессами, выполняющимися на одном компьютере, наилучшей производительностью обладает привязка Named Pipe. В двусторонних взаимодействиях между разными компьютерами наилучшей производительностью обладает привязка Net TCP. Однако этот механизм может применяться, только для работы с клиентами на основе WCF и не может использоваться для

организации взаимодействий разнородных систем. Кроме того, он не поддерживает возможность распределения нагрузки, так как связывает сеанс с определенным адресом сервера.

Чтобы получить производительность, близкую к производительности привязки TCP, и при этом сохранить совместимость со средствами распределения нагрузки, можно использовать собственную привязку HTTP, реализующую обмен данными в двоичном формате. Ниже приводится пример настройки такой привязки:

```
<bindings>
  <customBinding>
    <binding name = "NetHttpBinding">
      <reliableSession />
      <compositeDuplex />
      <oneWay />
      <binaryMessageEncoding />
      <httpTransport />
    </binding>
  </customBinding>
  <basicHttpBinding>
    <binding name = "BasicMtom" messageEncoding = "Mtom" />
  </basicHttpBinding>
  <wsHttpBinding>
    <binding name = "NoSecurityBinding">
      <security mode = "None" />
    </binding>
  </wsHttpBinding>
</bindings>
<services>
  <service name = "MyServices.CalculatorService">
    <endpoint address = " " binding = "customBinding"
              bindingConfiguration = "NetHttpBinding"
              contract = "MyServices.ICalculator" />
  </service>
</services>
```

Наконец, при возможности выбирайте простейшую привязку HTTP вместо WS-совместимой. Последняя использует не такой компактный формат сообщений.

В заключение

Как было показано в этой главе, повышая производительность операций ввода/вывода в своем приложении, можно получить значительный выигрыш и без оптимизации вычислительных операций. В этой главе мы:

- рассмотрели различия между синхронным и асинхронным вводом/выводом;
- исследовали различные механизмы рассылки извещений о завершении ввода/вывода;
- познакомились с общими рекомендациями, такими как уменьшение операций копирования буфера;
- обсудили особенности файлового ввода/вывода;
- познакомились с приемами оптимизации при работе с сокетами;
- увидели, как оптимизировать протокол, чтобы максимально использовать пропускную способность сети;
- сравнили различные инструменты сериализации, входящие в состав .NET Framework;
- рассмотрели приемы оптимизации при работе с фреймворком WCF.

ГЛАВА 8. Небезопасный код и взаимодействие с ним

Лишь немногие приложения состоят исключительно из управляемого кода. В действительности большинство приложений используют собственные или сторонние библиотеки, реализованные на низкоуровневых языках. Платформа .NET Framework предлагает несколько механизмов взаимодействий с низкоуровневым кодом, поддерживаемых многими распространенными технологиями:

- P/Invoke: обеспечивает возможность взаимодействий с функциями, экспортируемыми библиотеками DLL;
- COM Interop: позволяет обращаться к COM-объектам из управляемого кода, а также экспортировать классы .NET в виде COM-объектов для использования низкоуровневым кодом;
- язык C++/CLI: поддерживает взаимодействия с кодом на C и C++ посредством использования гибридного языка программирования.

Фактически базовая библиотека классов (Base Class Library, BCL), распространяемая в составе .NET Framework в виде набора библиотек DLL (основной из которых является *mscorlib.dll*) и содержащая реализацию встроенных типов .NET Framework, использует все вышеупомянутые механизмы. Поэтому можно смело утверждать, что любое мало-мальски сложное управляемое приложение в действительности является гибридным, в том смысле, что вызывает библиотеки, написанные на низкоуровневых языках.

Эти механизмы имеют большое значение, поэтому очень важно понять, какие проблемы производительности влечет их применение, и как уменьшить их влияние.

Небезопасный код

Управляемый код обеспечивает безопасность хранения данных в памяти и дает гарантии их защищенности, устраняя некоторые сложные в диагностике проблемы и уязвимости, распространенные в неуправляемом коде, такие как повреждение данных в динамической памяти и переполнение буфера. Эти преимущества достигаются за счет запрета прямого доступа к памяти по указателям, использования строго типизированных ссылок, проверки границ массивов и допустимости приведения типов объектов.

Однако, в некоторых случаях эти ограничения могут осложнять решение простых задач и отрицательно сказываться на производительности. Например, может возникнуть необходимость прочитать данные из файла в массив `byte[]`, но интерпретировать их как массив значений типа `double`. В C/C++ для этого достаточно привести указатель типа `char` к типу `double`. Но в безопасном коде .NET придется обернуть буфер объектом `MemoryStream` и использовать поверх его объект `BinaryReader` для чтения значений `double` из памяти; другой способ – использовать класс `BitConverter`. Оба решения вполне работоспособны, но они намного медленнее решения, доступного в неуправляемом коде. К счастью, C# и среда выполнения CLR поддерживают небезопасный доступ к памяти с помощью указателей и допускают возможность приведения типов указателей. В числе других небезопасных особенностей можно назвать выделение памяти в стеке и встраивание массивов в структуры. Недостатком небезопасного кода является снижение безопасности, что может стать причиной повреждения данных в памяти и появления уязвимостей, поэтому будьте очень осторожны при разработке небезопасного кода.

Чтобы получить возможность использовать небезопасный код, необходимо сначала включить поддержку компиляции небезопасного кода в настройках проекта C# (рис. 8.1), в результате чего компилятору C# автоматически будет передаваться параметр `/unsafe` командной строки. Затем следует выделить области, где допускается использование небезопасного кода или переменных. Такими областями могут быть целые классы или структуры, отдельные методы или фрагменты методов.

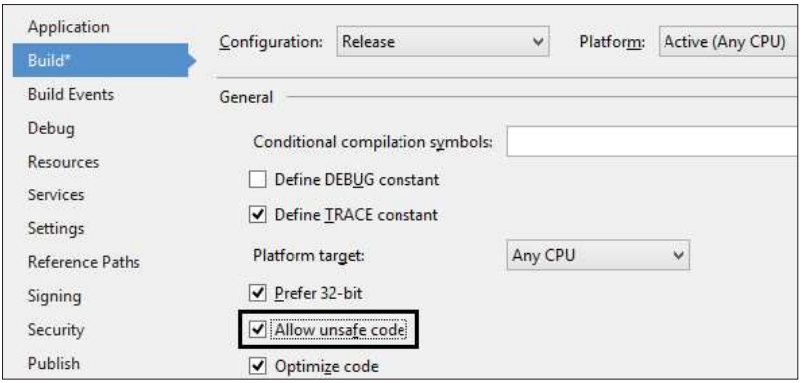

Рис. 8.1. Включение поддержки небезопасного кода в настройках проекта C# (Visual Studio 2012).

Закрепление объектов в памяти и дескрипторы сборщика мусора

Так как управляемые объекты, размещаемые в динамической памяти, могут перемещаться в процессе сборки мусора, их необходимо закреплять в памяти, прежде чем пытаться получить их адреса.

Закрепить объект в памяти можно либо за счет использования области видимости `fixed` (см. пример в листинге 8.1), либо создавая дескриптор сборщика мусора (см. листинг 8.2). Заглушки (stubs) P/Invoke, о которых рассказывается ниже, также закрепляют объекты, подобно тому, как это делает инструкция `fixed`. Используйте инструкцию `fixed`, если закрепление может быть ограничено областью видимости функции, так как это более эффективный подход, чем применение дескрипторов сборщика мусора. В противном случае используйте `GCHandle.Alloc` для создания дескриптора, закрепляющего объект на более продолжительное время (до явного вызова `GCHandle.Free`). Объекты, размещаемые на стеке (имеющие типы значений), не требуют закрепления, потому что они недоступны сборщику мусора. Указатели на такие объекты можно получать непосредственно, используя оператор получения ссылки – знак амперсанда (`&`).

Листинг 8.1. Использование инструкции fixed и приведение типа указателя на буфер с данными

```
using (var fs = new FileStream(@"C:\Dev\samples.dat", FileMode.Open)) {
    var buffer = new byte[4096];
    int bytesRead = fs.Read(buffer, 0, buffer.Length);
```

```
    unsafe {
        double sum = 0.0;
        fixed (byte* pBuff = buffer) {
            double* pDblBuff = (double*)pBuff;
            for (int i = 0; i < bytesRead / sizeof(double); i++)
                sum + = pDblBuff[i];
        }
    }
}
```

Внимание. *Указатель, полученный в области видимости инструкции* `fixed`*, нельзя использовать за ее пределами, потому что закрепленный объект открепляется после выхода из этой области. Ключевое слово fixed может применяться к массивам типов значений, к строкам и к полям управляемого класса, имеющим тип значения. Не забывайте указывать организацию памяти в структурах.*

Дескрипторы сборщика мусора – это инструмент, позволяющий ссылаться на управляемый объект, находящийся в динамической памяти сборщика мусора, с помощью неизменяемого значения дескриптора (даже если адрес объекта изменится), который можно передавать даже неуправляемому коду. Всего существует четыре разновидности дескрипторов сборщика мусора, определяемые значениями перечисления `GCHandleType`: `Weak`, `WeakTrackRessurection`, `Normal` и `Pinned`. Типы `Normal` и `Pinned` предотвращают утилизацию объекта сборщиком мусора, даже если на него не осталось ни одной ссылки. Кроме того, тип `Pinned` дает возможность не только закрепить объект, но и получить его адрес в памяти. Типы `Weak` и `WeakTrackResurrection` не препятствуют утилизации объекта, но позволяют получить обычную (сильную) ссылку на него, если объект еще не был утилизирован. Дескрипторы этих типов используются типом `WeakReference`.

Листинг 8.2. Использование GCHandle и приведение типа указателя на буфер с данными

```
using (var fs = new FileStream(@"C:\Dev\samples.dat", FileMode.Open)) {
    var buffer = new byte[4096];
    int bytesRead = fs.Read(buffer, 0, buffer.Length);
    GCHandle gch = GCHandle.Alloc(buffer, GCHandleType.Pinned);
    unsafe {
        double sum = 0.0;
        double* pDblBuff = (double *)(void *)gch.AddrOfPinnedObject();
        for (int i = 0; i < bytesRead / sizeof(double); i++)
            sum + = pDblBuff[i];
        gch.Free();
    }
}
```

Внимание. *Закрепление объектов может вызывать фрагментацию динамической памяти в ходе сборки мусора. Фрагментация приводит к напрасному расходованию памяти и снижает эффективность алгоритма сборки мусора. Чтобы уменьшить фрагментацию памяти, не удерживайте объекты закрепленными дольше, чем это необходимо.*

Управление жизненным циклом

Во многих случаях неуправляемый код продолжает удерживать неуправляемые ресурсы между вызовами функции и требует явного их освобождения. В этом случае, в дополнение к методу-финализатору, реализуйте в обертывающем управляемом классе интерфейс `IDisposable`. Это сохранит за клиентами возможность явно освобождать неуправляемые ресурсы, и обеспечит запасной вариант освобождения памяти с помощью финализатора, если вы забудете выполнить освобождение явно.

Выделение неуправляемой памяти

Управляемые объекты, занимающие в памяти более 85 000 байт (обычно массивы байтов и строки), помещаются в кучу больших объектов (Large Object Heap, LOH), которая обслуживается сборщиком мусора вместе с областью поколения 2 и требует значительных вычислительных затрат. Куча больших объектов также часто оказывается фрагментированной из-за того, что она никогда не сжимается, а свободное пространство между объектами используется, только когда это возможно. Обе эти проблемы увеличивают расход памяти и вычислительной мощности процессора. Поэтому гораздо эффективнее использовать пулы памяти или выделять подобные буферы в неуправляемой памяти (например, вызовом `Marshal.AllocHGlobal`). Если позднее потребуется получить доступ к неуправляемому буферу из управляемого кода, используйте прием на основе «потоков», копируя небольшие фрагменты из неуправляемого буфера в управляемую память и обрабатывая их по одному. Чтобы упростить работу, используйте `System.UnmanagedMemoryStream` и `System.UnmanagedMemoryAccessor`.

Использование пулов памяти

При интенсивном использовании буферов для взаимодействий с неуправляемым кодом, их можно выделять в динамической памяти сборщика мусора или в неуправляемой памяти. Первый подход недостаточно эффективен из-за высоких накладных расходов операции выделения памяти, когда буферы имеют маленький размер. Кроме

того, управляемые буферы необходимо закреплять, что увеличивает фрагментацию памяти. Второй подход также имеет свои недостатки, потому что в большинстве случаев управляемый код работает с буферами, размещенными в управляемой памяти (`byte[]`), а не с указателями. Нельзя преобразовать указатель на управляемый массив, минуя копирование, а это отрицательно сказывается на производительности.

> **Совет.** *Определить процессорное время, затрачиваемое на сборку мусора, можно с помощью счетчика производительности «% Time in GC» («% времени в GC») из категории «.NET CLR Memory» («Память CLR .NET»), но он не подскажет, какой именно код несет ответственность за «впустую» растраченное время. Прежде чем приступать к оптимизации, задействуйте профилировщик (см. главу 2) и прочитайте главу 4, где приводятся дополнительные советы по повышению производительности сборщика мусора.*

Мы предлагаем использовать решение (рис. 8.2), обеспечивающее доступ к буферу без копирования, как из управляемого, так и из неуправляемого кода, и не оказывающее отрицательного влияния на сборщик мусора. Идея состоит в том, чтобы выделить большие буферы в управляемой памяти (сегменты) в куче больших объектов. В этом случае отпадает необходимость закреплять сегменты, потому что они уже являются неперемещаемыми.

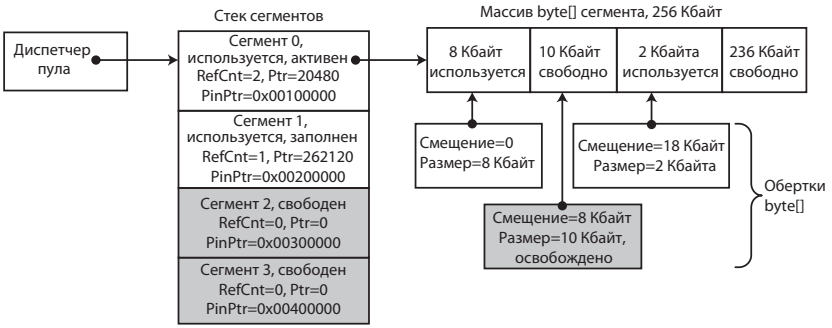

Рис. 8.2. Предлагаемая схема организации пула памяти.

Простой диспетчер пула, в котором указатель на сегмент (фактически индекс) может перемещаться только вперед с каждым запросом, выделяет буферы различных размеров (вплоть до размера сегмента) и возвращает объект, обертывающий выделенные буферы. Когда указатель приблизится к концу сегмента, и очередная попытка выделить буфер в этом сегменте потерпит неудачу, из пула сегментов выделяется новый сегмент, и попытка выделить память повторяется.

Каждый сегмент имеет счетчик ссылок, увеличивающийся с каждой операцией выделения и уменьшающийся с каждой операцией освобождения буфера. Когда счетчик ссылок достигнет нуля, сегмент может быть приведен в исходное состояние (установкой указателя в его начало, с возможным заполнением памяти нулевыми байтами), и возвращен в пул сегментов.

Объект-обертка хранит массив `byte[]` сегмента, смещение начала буфера в сегменте, размер буфера и неуправляемый указатель. Фактически, объект-обертка – это окно в сегмент. Он также ссылается на сегмент, чтобы уменьшить счетчик ссылок после его освобождения. Объект-обертка может поддерживать вспомогательные методы, например, чтобы проверить значение смещения и выход за пределы буфера.

Так как разработчики приложений для .NET привыкли считать, что данные всегда начинаются с индекса 0 и простираются до самого конца массива, вам придется изменить код, написанный исходя из таких предположений, и обеспечить учет дополнительных параметров, определяющих смещение и длину, которые должны передаваться вместе с буфером. Большинство методов в базовой библиотеке классов .NET, выполняющих операции с буферами, имеют перегруженные версии, принимающие смещение и длину явно.

Главный недостаток этого похода заключается в потере поддержки автоматического управления памятью. Чтобы освободить память, занимаемую сегментом, необходимо явно удалить объект-обертку. Реализация финализатора – не самое лучшее решение, потому что он будет более чем отрицательно сказываться на производительности.

P/Invoke

Механизм Platform Invoke, более известный как P/Invoke, позволяет вызывать из управляемого кода функции в стиле языка C, экспортируемые библиотеками DLL. Чтобы воспользоваться механизмом P/Invoke, управляемый код должен объявить статический внешний (`static extern`) метод с сигнатурой (типы параметров и тип возвращаемого значения), эквивалентной функции на языке C. Сам метод должен быть снабжен атрибутом `DllImport`, определяющим, как минимум, библиотеку DLL, экспортирующую требуемую функцию.

```
// Фактическое объявление в WinBase.h:
HMODULE WINAPI LoadLibraryW(LPCWSTR lpLibFileName);

// Объявление на языке C#:
class MyInteropFunctions {
```

```
    [DllImport("kernel32.dll", SetLastError = true)]
    public static extern IntPtr LoadLibrary(string fileName);
}
```

В предыдущем фрагменте метод `LoadLibrary` определяется как функция, принимающая значение типа `string` и возвращающая значение типа `IntPtr` – указатель, который не может быть разыменован непосредственно, то есть, оно не делает код небезопасным. Атрибут `DllImport` указывает, что функция экспортируется библиотекой *kernel32.dll* (главная библиотека Win32 API) и что последняя ошибка Win32 должна сохраняться в локальной памяти потока, чтобы она не была затерта неявными вызовами функций Win32 (например, выполняемыми средой выполнения CLR). В атрибуте `DllImport` можно также указать соглашение о вызове функции, кодировку для строк, параметры разрешения экспортируемых имен, и так далее.

Если сигнатура библиотечной функции содержит составные типы, такие как структуры языка C, в управляемом коде необходимо определить эквивалентные структуры или классы, используя эквивалентные типы для каждого поля. Порядок следования полей, типы полей и применяемые к ним правила выравнивания должны совпадать с ожидаемыми в коде на языке C. В некоторых случаях может потребоваться применить атрибут `MarshalAs` к полям в параметрах функции или в возвращаемом значении, чтобы изменить поведение механизма маршалинга по умолчанию. Например, управляемый тип `System.Boolean` (`bool`) может иметь разные представления в низкоуровневом коде: тип Win32 `BOOL` имеет размер четыре байта, а значению `true` соответствует любое ненулевое значение, тогда как в C++ значение типа `bool` занимает один байт, а значению `true` соответствует целое число 1.

В следующем фрагменте демонстрируется применение атрибута `StructLayout` к структуре `WIN32_FIND_DATA`, определяющего расположение полей в памяти; без этого среда выполнения CLR может переупорядочить поля для большей эффективности. Атрибут `MarshalAs` применяется к полям `cFileName` и `cAlternativeFileName`, чтобы указать, что строки должны передаваться как строки фиксированного размера, встроенные в структуру, а не как простые указатели на внешние строки.

```
// Фактическое объявление в WinBase.h:
typedef struct _WIN32_FIND_DATAW {
    DWORD dwFileAttributes;
    FILETIME ftCreationTime;
    FILETIME ftLastAccessTime;
    FILETIME ftLastWriteTime;
```

```
    DWORD nFileSizeHigh;
    DWORD nFileSizeLow;
    DWORD dwReserved0;
    DWORD dwReserved1;
    WCHAR cFileName[MAX_PATH];
    WCHAR cAlternateFileName[14];
} WIN32_FIND_DATAW;

HANDLE WINAPI FindFirstFileW(__in LPCWSTR lpFileName,
    __out LPWIN32_FIND_DATAW lpFindFileData);

// Объявление на языке C#:
[StructLayout(LayoutKind.Sequential, CharSet = CharSet.Auto)]
struct WIN32_FIND_DATA {
    public uint dwFileAttributes;
    public FILETIME ftCreationTime;
    public FILETIME ftLastAccessTime;
    public FILETIME ftLastWriteTime;
    public uint nFileSizeHigh;
    public uint nFileSizeLow;
    public uint dwReserved0;
    public uint dwReserved1;
    [MarshalAs(UnmanagedType.ByValTStr, SizeConst = 260)]
    public string cFileName;
    [MarshalAs(UnmanagedType.ByValTStr, SizeConst = 14)]
    public string cAlternateFileName;
}

[DllImport("kernel32.dll", CharSet = CharSet.Auto)]
static extern IntPtr FindFirstFile(string lpFileName, out WIN32_
FIND_DATA lpFindFileData);
```

Когда программа вызовет метод `FindFirstFile` из предыдущего листинга, среда выполнения CLR загрузит библиотеку DLL, которая экспортирует функцию (*kernel32.dll*), найдет в ней требуемую функцию (`FindFirstFile`) и выполнит преобразование параметров из управляемого представления в низкоуровневое (и обратно). В данном примере, входной строковый параметр `lpFileName` будет преобразован в низкоуровневую строку, а низкоуровневую структуру `WIN32_FIND_DATAW`, на которую ссылается параметр `lpFindFileData`, — в управляемую структуру `WIN32_FIND_DATA`. Все эти этапы более подробно будут описаны в следующих разделах.

PInvoke.net и P/Invoke Interop Assistant

Создание сигнатур для механизма P/Invoke может быть сложным и утомительным делом. Существует множество правил и тонкостей, которые следует помнить. Неверное определение сигнатуры может

привести к появлению сложных в диагностике ошибок. К счастью, имеются два ресурса, которые могут упростить решение этой проблемы: веб-сайт PInvoke.net и утилита P/Invoke Interop Assistant.

PInvoke.net – очень полезный веб-сайт в стиле Wiki, где можно найти или передать свои сигнатуры P/Invoke для различных функций Microsoft API. Сайт PInvoke.net был создан Адамом Натаном (Adam Nathan), ведущим инженером-программистом в Microsoft, прежде работавшим в подразделении .NET CLR Quality Assurance и написавшим объемную книгу о взаимодействии с COM-объектами. Получить доступ к сигнатурам можно также с помощью расширения для Visual Studio, распространяемого бесплатно.

P/Invoke Interop Assistant – это бесплатный инструмент, который можно получить на сайте CodePlex вместе с исходными кодами. Он содержит базу данных (XML-файл) с описаниями функций Win32, структур и констант, используемых для создания сигнатур P/Invoke. Он также способен генерировать сигнатуры P/Invoke по объявлениям функций на языке C, а также создавать объявления низкоуровневых функций обратного вызова и сигнатуры низкоуровневых интерфейсов COM-объектов для заданной управляемой сборки (assembly).

На рис. 8.3 показано, как выглядит окно утилиты P/Invole Interop Assistant с результатами поиска функции «CreateFile» слева, и сигнатурой P/Invoke, вместе с необходимыми структурами – справа. Утилиту P/Invoke Interop Assistant (и другие полезные инструменты, которые могут пригодиться для организации взаимодействий с CLR) можно получить по адресу: http://clrinterop.codeplex.com/.

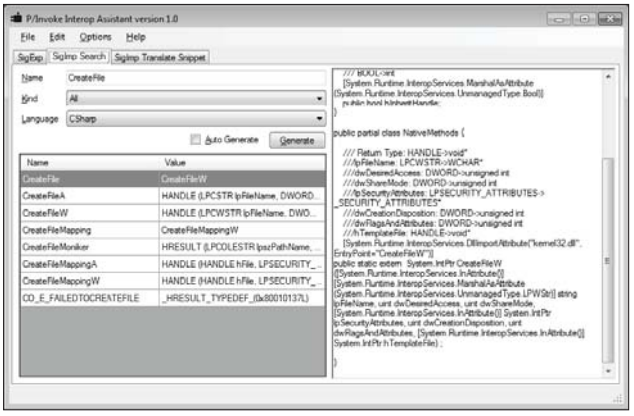

Рис. 8.3. Скриншот окна утилиты P/Invoke Interop Asssitant, демонстрирующей сигнатуру P/Invoke для CreateFile.

Привязка

При первом вызове функции с помощью механизма P/Invoke, обращением к функции `LoadLibrary` производится загрузка библиотеки DLL и всех ее зависимостей (если они еще не были загружены) в адресное пространство процесса. Затем выполняется поиск требуемой экспортируемой функции, при этом сначала предпринимается попытка найти управляемую версию функции. Порядок поиска зависит от значений полей `CharSet` и `ExactSpelling` в атрибуте `DllImport`.

- Если поле `ExactSpelling` имеет значение `true`, P/Invoke пытается найти функцию точно соответствующую указанному имени, учитывая только соглашения о вызове. В случае неудачи P/Invoke не выполняет поиск других версий с тем же именем и возбуждает исключение `EntryPointNotFoundException`.
- Если поле `ExactSpelling` имеет значение `false`, поведение механизма определяется значением свойства `CharSet`:
  - для значения `CharSet.Ansi` (по умолчанию) P/Invoke сначала пытается найти точное соответствие указанному имени, а затем – измененного имени (с окончанием «A»);
  - для значения `CharSet.Unicode` P/Invoke сначала пытается найти функцию с измененным именем (с окончанием «W»), а затем – точное соответствие.

По умолчанию поле `ExactSpelling` имеет значение `false` в C# и `True` в VB.NET. Во всех современных версиях Windows (выпущенных после Windows ME), значение `CharSet.Auto` соответствует значению `CharSet.Unicode`.

> **Совет.** *Используйте версии функций Win32 API для работы с Юникодом. Версии Windows NT и выше имеют встроенную поддержку Юникода (UTF16). Если пользоваться ANSI-версиями функций Win32 API, строки автоматически будут преобразовываться в Юникод и передаваться версиям функций для работы с Юникодом, что повлечет снижение производительности. Для внутреннего представления строк в .NET используется кодировка UTF16, поэтому маршалинг строковых параметров будет выполняться быстрее, если они уже будут в кодировке UTF16. Предусматривайте в своем коде и особенно в интерфейсах совместимость с Юникодом, что даст вам дополнительные преимущества с точки зрения глобализации. Устанавливайте* `ExactSpelling` *в значение* `true`*, это повысит производительность на начальном этапе за счет устранения поиска ненужных функций.*

Заглушки маршалера

При первом вызове функции с помощью механизма P/Invoke, сразу после загрузки библиотеки DLL, заглушки (stubs) маршалера P/Invoke генерируются по требованию, и повторно использоваться в последующих вызовах. При вызове маршалер выполняет следующие действия:

1. Проверяет наличие у вызывающего процесса права на выполнение неуправляемого кода.
2. Преобразует управляемые аргументы в их неуправляемые аналоги, возможно с выделением памяти.
3. Устанавливает вытесняющий режим работы для потока выполнения сборщика мусора, чтобы сборка мусора могла производиться, не дожидаясь, пока поток выполнения достигнет безопасной точки.
4. Вызывает библиотечную функцию.
5. Восстанавливает кооперативный режим работы потока выполнения сборщика мусора.
6. При необходимости сохраняет код ошибки Win32 в локальной переменной потока для последующего доступа с помощью метода `Marshal.GetLastWin32Error`.
7. При необходимости преобразует значение типа `HRESULT` в исключение и возбуждает его.
8. Преобразует низкоуровневое исключение, если оно было возбуждено, в управляемое исключение.
9. Преобразует возвращаемое значение и выходные параметры обратно в их управляемые аналоги.
10. Освобождает любые временные блоки динамической памяти, выделенные при вызове.

Механизм P/Invoke может также использоваться низкоуровневым кодом для вызова управляемых методов. Обратная заглушка маршалера может быть сгенерирована для делегата (через `Marshal.GetFunctionPointerForDelegate`), если передается механизму P/Invoke как параметр неуправляемой функцией. В ответ неуправляемая функция получит указатель на функцию, которую можно вызвать, чтобы обратиться к управляемому методу. Этот указатель на функцию будет ссылаться на заглушку, которая помимо параметров маршалинга знает также адрес целевого объекта (указатель `this`).

В версии .NET Framework 1.x заглушки маршалера представляли собой либо код сгенерированной сборки (для простых сигнатур),

либо код на языке ML (Marshaling Language) (для сложных сигнатур). Язык ML – это внутренний байт-код, выполняемый внутренним интерпретатором. С появлением в .NET Framework 2.0 поддержки процессоров AMD64 и Itanium, в Microsoft обнаружили, что реализация параллельной инфраструктуры ML для каждого типа процессоров – слишком обременительное занятие. Поэтому заглушки для 64-разрядных версий .NET Framework 2.0 были реализованы исключительно на языке IL. Хотя заглушки на языке IL действуют значительно быстрее заглушек на языке ML, они все равно оказываются медленнее заглушек на языке ассемблера x86, поэтому в Microsoft было решено сохранить реализацию для архитектуры x86. В .NET Framework 4.0, инфраструктура генерирования заглушек ан языке IL была значительно оптимизирована, что сделало заглушки на языке IL даже быстрее, чем заглушки на языке ассемблера x86. Это позволило Microsoft полностью удалить реализацию заглушек для аппаратной архитектуры x86 и унифицировать их для всех поддерживаемых платформ.

> **Совет.** *Вызов функции, пересекающий границу между управляемым и неуправляемым кодом, как минимум на порядок медленнее непосредственного вызова в том же окружении. Если и управляемый, и неуправляемый код находятся в вашем ведении, конструируйте интерфейсы так, чтобы уменьшить количество взаимодействий между управляемым и неуправляемым кодом. Попробуйте объединить несколько «заданий» в один вызов. Аналогично, старайтесь объединять несколько вызов простых функций (таких как функции* `Get/Set`*) в один вызов, выполняющий всю необходимую работу.*

Корпорация Microsoft предоставляет бесплатный инструмент *IL Stub Diagnostics*, который можно получить на сайте проекта CodePlex вместе с исходными кодами. Он подписывается на события CLR ETW IL создания заглушек и отображает сгенерированный код заглушек на языке IL в графическом интерфейсе.

Ниже приводится пример заглушки маршалера на языке IL с комментариями, состоящий из пяти разделов: инициализация, маршалинг входных параметров, вызов, маршалинг обратно возвращаемого значения и/или выходных параметров, и завершение. Заглушка маршалера генерируется для следующей сигнатуры:

```
// Управляемая сигнатура:
[DllImport("Server.dll")]static extern int Marshal_String_In(string s);
// Неуправляемая сигнатура:
unmanaged int __stdcall Marshal_String_In(char *s)
```

В разделе инициализации заглушка объявляет локальные переменные (на стеке), получает контекст заглушки и проверяет право на выполнение неуправляемого кода.

```
// Заглушка IL:
// Объем кода 153 (0x0099)
.maxstack 3
// Локальные переменные:
// IsSuccessful, pNativeStrPtr, SizeInBytes, pStackAllocPtr, result,
result, result
.locals (int32,native int,int32,native int,int32,int32,int32)

call native int [mscorlib] System.StubHelpers.StubHelpers::
GetStubContext()
// Проверка права на выполнение неуправляемого кода
call void [mscorlib] System.StubHelpers.StubHelpers::DemandPermission(
    native int)
```

В разделе маршалинга заглушка передает входные параметры неуправляемой функции. В данном примере выполняется маршалинг единственного входного строкового параметра. Для преобразования конкретных типов и их категорий из управляемого представления в неуправляемое, и обратно, маршалер может использовать вспомогательные типы в пространстве имен `System.StubHelpers-namespace` или обращаться к классу `System.Runtime.InteropServices.Marshal`. В данном примере для маршалинга строки вызывается метод `CSTRMarshaler::ConvertToNative`.

Здесь используется небольшая оптимизация: если управляемая строка достаточно короткая, память для нее выделяется на стеке (что намного быстрее). В противном случае для нее выделяется блок в динамической памяти.

```
    ldc.i4     0x0          // IsSuccessful = 0 [положить 0 на стек]
    stloc.0                 // [сохранить в IsSuccessful]
IL_0010:
    nop        // аргумент {
    ldc.i4     0x0          // pNativeStrPtr = null [положить 0 на стек]
    conv.i                  //   [int32 в "неуправляемый int" (указатель)]
    stloc.3                 //   [сохранить результат в pNativeStrPtr]
    ldarg.0                 // if (managedString == null)
    brfalse    IL_0042      //     goto IL_0042
    ldarg.0                 // [экземпляр managedString положить на стек]
                            // обратиться к свойству Length
                            // (получить число символов)
    call       instance int32 [mscorlib] System.String::get_Length()
    ldc.i4     0x2          // Прибавить 2 к длине, 1 - для нулевого символа
                            // в managedString, и 1 - для дополнительного
```

```
                          // нулевого символа [положить константу 2 на стек]
    add                   // [фактическое сложение, результат – на стеке]
                          // загрузить статическое поле, значение зависит от
                          // lang. для приложений, не поддерживающих Юникод
    ldsfld System.Runtime.InteropServices.Marshal::SystemMaxDBCSCharSize
    mul                   // Умножить длину на SystemMaxDBCSCharSize, чтобы
                          // получить количество байтов
    stloc.2               // Сохранить в SizeInBytes
    ldc.i4 0x105          // Сравнить SizeInBytes и 0x105, чтобы исключить
                          // возможность выделения слишком большого количества
                          // памяти на стеке [константу 0x105 на стек]
                          // CSTRMarshaler::ConvertToNative для случая
                          // pStackAllocPtr == null и вызовет CoTaskMemAlloc
                          // для выделения большего объема памяти
    ldloc.2               // [Положить SizeInBytes]
    clt                   // [If SizeInBytes > 0x105, положить 1 иначе 0]
    brtrue     IL_0042    // [If 1 goto IL_0042]
    ldloc.2               // SizeInBytes на стек (аргумент для localloc)
    localloc              // Выделить память на стеке, указатель
                          // оставить на стеке
    stloc.3               // Сохранить в pStackAllocPtr
IL_0042:
    ldc.i4     0x1        // константу 1 на стек (параметр flags)
    ldarg.0               // аргумент managedString на стек
    ldloc.3               // pStackAllocPtr на стек (this может быть null)
                          // Вызвать функцию преобразования Unicode в ANSI
    call native int [mscorlib]System.StubHelpers.
        CSTRMarshaler::ConvertToNative(int32,string, native int)
    stloc.1               // Сохранить результат в pNativeStrPtr,
                          // может быть равен pStackAllocPtr
    ldc.i4     0x1        // IsSuccessful = 1 [втолкнуть 1 на стек]
    stloc.0               // [сохранить в IsSuccessful]
    nop
    nop
    nop
```

В следующем разделе заглушка получает из своего контекста указатель на неуправляемую функцию и вызывает ее. Инструкция вызова в действительности делает больше, чем кажется. Например, она изменяет режим работы сборщика мусора, перехватывает значение, возвращаемое неуправляемой функцией, и приостанавливает выполнение управляемого кода, пока сборщик мусора выполняет фазу, требующую этого.

```
    ldloc.1               // Положить pStackAllocPtr на стек,
                          // для пользовательской функции,
                          // не для GetStubContext
    call native int [mscorlib] System.StubHelpers.StubHelpers::
GetStubContext()
```

```
    ldc.i4      0x14          // Прибавить 0x14 к указателю контекста
    add                       // [фактическое сложение, результат на стеке]
    ldind.i                   // [разыменовать указатель,
                              //  результат на стеке]
    ldind.i                   // [разыменовать указатель на функцию,
                              //  результат на стеке]
    calli unmanaged stdcall int32(native int) // Вызвать функцию
```

Следующий раздел фактически состоит из двух разделов, выполняющих обратное преобразование неуправляемых типов в управляемые для возвращаемого значения и выходных параметров, соответственно. В данном примере, неуправляемая функция возвращает значение типа `int`, не требующее преобразования (маршалинга), которое просто копируется в локальную переменную в исходном виде. Поскольку здесь нет выходных параметров, последний раздел остался пустым.

```
// UnmarshalReturn {
    nop         // возврат {
    stloc.s     0x5     // Сохранить результат функции (int) в x, y и z
    ldloc.s     0x5
    stloc.s     0x4
    ldloc.s     0x4
    nop         // } возврат
    stloc.s     0x6
// } UnmarshalReturn
// Unmarshal {
    nop         // аргумент {
    nop         // } аргумент
    leave       IL_007e       // Выход из блока try
IL_007e:
    ldloc.s     0x6           // поместить z на стек
    ret                       // Вернуть z
// } Unmarshal
```

Наконец, в разделе завершения освобождается память, выделенная временно для нужд маршалинга. Эти операции выполняются в блоке `finally` инструкции `try`, поэтому они будут выполнены, даже если неуправляемая функция возбудит исключение. Имеется также возможность выполнять некоторые действия только в случае исключения. Во взаимодействиях с COM-объектами заглушки могут преобразовывать возвращаемое значение `HRESULT`, указывающее на ошибку, в исключение.

```
// ExceptionCleanup {
IL_0081:
// } ExceptionCleanup
// Cleanup {
```

```
    ldloc.0                  // If (IsSuccessful && !pStackAllocPtr)
    ldc.i4 0x0               //     ClearNative(pNativeStrPtr)
    ble IL_0098
    ldloc.3
    brtrue IL_0098
    ldloc.1
    call void [mscorlib] System.StubHelpers.CSTRMarshaler::ClearNative(
        native int)
IL_0098:
    endfinally
IL_0099:
// } Cleanup
    .try IL_0010 to IL_007e finally handler IL_0081 to IL_0099
```

Как видите, заглушка маршалера на языке IL имеет замысловатую организацию даже для функций с такой простой сигнатурой. Для сложных сигнатур генерируются еще более длинные и медленные заглушки.

Двоично совместимые типы

Многие неуправляемые типы двоично совместимы с управляемым кодом. Эти типы, называемые двоично совместимыми (blittable), не требуют преобразования и передаются через границу между управляемым и неуправляемым кодом намного быстрее, чем двоично несовместимые (non-blittable) типы. В действительности заглушка маршалера может оптимизировать такую передачу еще больше, закрепляя управляемый объект и передавая неуправляемому коду указатель на него, исключая одну или две операции копирования (по одной для каждого направления передачи).

К двоично совместимым типам относятся:

- `System.Byte` (`byte`);
- `System.SByte` (`sbyte`);
- `System.Int16` (`short`);
- `System.UInt16` (`ushort`);
- `System.Int32` (`int`);
- `System.UInt32` (`uint`);
- `System.Int64` (`long`);
- `System.UInt64` (`ulong`);
- `System.IntPtr`;
- `System.UIntPtr`;
- `System.Single` (`float`);
- `System.Double` (`double`).

Кроме того, одномерные массивы с элементами двоично совместимых типов (где все элементы принадлежат одному и тому же типу) также являются двоично совместимыми, так же как структуры или классы, состоящие только из полей двоично совместимых типов.

Тип `System.Boolean` (`bool`) не является двоично совместимым, потому что в неуправляемом коде он может иметь 1, 2 или 4-байтное представление. Тип `System.Char` (`char`) не является двоично совместимым, потому что может представлять символ ANSI или Юникода. Тип `System.String` (`string`) не является двоично совместимым, потому что его неуправляемое представление может состоять из символов ANSI или Юникода, и может быть строкой в стиле языка C или COM BSTR, а также потому, что управляемая строка должна быть неизменяемой. Тип, содержащий поле со ссылкой на объект, не может быть двоично совместимым, даже если это ссылка на двоично совместимый тип или массив. При маршалинге двоично несовместимых типов требуется выделять память для сохранения преобразованных версий параметров, ее заполнения и последующего освобождения.

Наивысшей производительности можно добиться, реализовав маршалинг входных строковых параметров вручную (см. следующий фрагмент кода). Но при этом вызываемая неуправляемая функция должна принимать строку в кодировке UTF-16, в стиле языка C, и никогда не писать в память, занимаемую строкой, из-за чего такая оптимизация редко бывает применима. Чтобы выполнить маршалинг вручную, необходимо закрепить входную строку, изменить сигнатуру P/Invoke так, чтобы неуправляемая функция выглядела, как принимающая `IntPtr` вместо `string`, и передавать ей указатель на закрепленную строку.

```
class Win32Interop {
    [DllImport("NativeDLL.DLL", CallingConvention = CallingConvention.Cdecl)]
    public static extern void NativeFunc(IntPtr pStr); // IntPtr вместо string
}

// Управляемый код вызывает функцию P/Invoke внутри области
// видимости fixed что обеспечивает закрепление строки:
unsafe
{
    string str = "MyString";
    fixed (char *pStr = str) {
        // pStr можно использовать в нескольких вызовах.
        Win32Interop.NativeFunc((IntPtr)pStr);
    }
}
```

Преобразование неуправляемой строки UTF-16 в стиле языка C в управляемую строку также можно оптимизировать, применив конструктор `System.String`, принимающий параметр типа `char*`. Конструктор `System.String` создает копию буфера, поэтому неуправляемую память, занимаемую строкой, можно освободить сразу после создания управляемой строки. Обратите внимание, что здесь не выполняется никакой проверки, чтобы убедиться, что строка содержит только допустимые символы Юникода.

Направление маршалинга, ссылочные типы и типы значений

Как упоминалось выше, параметры функции могут передаваться заглушкой маршалера в одном или в двух направлениях. Направление передачи параметра определяется рядом факторов:

- является ли параметр ссылочным типом или типом значения:
- передается ли параметр по ссылке или по значению;
- является ли тип параметра двоично совместимым или нет;
- применяются ли к параметрам атрибуты, изменяющие направление маршалинга (`System.RuntimeInteropService.InAttribute` и `System.RuntimeInteropService.OutAttribute`).

Для дальнейшего обсуждения будем обозначать направление маршалинга из управляемого кода в неуправляемый как «in»; а направление из неуправляемого кода в управляемый – как «out». Ниже приводится список правил определения направления маршалинга по умолчанию:

- параметры, передаваемые по значению, независимо от того, являются ли они ссылочными типами или типами значений, передаются только в направлении «in»;
  - не требуется применять атрибут `In` вручную;
  - `StringBuilder` является исключением из правила и всегда передается в направлении «in/out».
- параметры, передаваемые по ссылке (с применением ключевого слова `ref` в C# или `ByRef` в VB.NET), независимо от того, являются ли они ссылочными типами или типами значений, передаются в направлении «in/out».

Добавление атрибута `OutAttribute` запрещает маршалинг в направлении «in», поэтому вызываемый неуправляемый код может не получить значение, переданное вызывающим кодом. Ключевое слов

`out` в языке C# действует подобно ключевому слову `ref`, но добавляет атрибут `OutAttribute`.

> **Совет.** *Если типы параметров не являются двоично совместимыми в вызове P/Invoke и вам требуется организовать маршалинг только в направлении «out», ненужного маршалинга в направлении «in» можно избежать, использовав ключевое слово* `out` *вместо* `ref`.

Из-за закрепления параметров двоично совместимых типов при маршалинге, как описывалось выше, для двоично совместимых ссылочных типов автоматически устанавливается направление «in/out», даже если правила выше утверждают иное. Однако не следует полагаться на эту особенность, когда требуется получить поведение «out» или «in/out» маршалинга, а вместо этого указывать направление явно, с помощью атрибутов, так как данная особенность перестанет действовать, если позднее вы добавите поле двоично несовместимого типа или если это вызов COM-объекта, пересекающий границы подразделений (apartments).

Разница между маршалингом типов значений и ссылочных типов заключается в особенностях их передачи через стек:

- типы значений, которые передаются по значению, копируются на стек, поэтому они всегда передаются в направлении «in», независимо от используемых атрибутов;
- типы значений, которые передаются по ссылке, и ссылочные типы, которые передаются по значению, передаются как указатель;
- ссылочные типы, которые передаются по ссылке, передаются как указатель на указатель.

> **Примечание.** *Передача объемных типов значений (более десятка байт) по значению стоит дороже передаче их по ссылке. То же относится к объемным возвращаемым значениям, вместо которых может оказаться предпочтительнее использовать выходные параметры.*

Code Access Security

Механизм .NET Code Access Security позволяет выполнять код, не вызывающий доверия, в изолированном окружении, называемом «песочницей» (sandbox), с ограниченным доступом к возможностям среды выполнения (например, P/Invoke) и BCL (например, доступ к файлам и реестру). Когда вызывается неуправляемый код, механизм Code Access Security требует, чтобы все сборки, чьи методы будут

вызываться, имели право `UnmanagedCode`. Заглушка маршалера будет проверять это право для каждого вызова, что влечет за собой обход стека вызовов, чтобы убедиться, что весь код в цепочке вызовов обладает данным правом.

> **Совет.** *Если вы выполняете только код, пользующийся доверием, или у вас имеются иные средства, гарантирующие безопасность, вы можете значительно увеличить производительность, добавив атрибут* `SuppressUnmanagedCodeSecurityAttribute` *в объявление метода, класса (в этом случае данный атрибут применяется ко всем методам), интерфейса или делегата.*

Взаимодействие с COM-объектами

Объектная модель программных компонентов (Component Object Model, COM) проектировалась с целью дать возможность создавать компоненты на любом языке/платформе, обладающем поддержкой этой модели, и использовать их в любом языке/платформе (другом), так же обладающем поддержкой COM. Платформа .NET не исключение и позволяет легко использовать сторонние COM-объекты и экспортировать типы .NET в виде COM-объектов.

Суть организации взаимодействий с COM-объектами та же, что и при использовании механизма P/Invoke: вы объявляете управляемое представление COM-объекта, а среда выполнения CLR создает объект-обертку, реализующий маршалинг. Существует две разновидности оберток: обертка, вызываемая средой выполнения (Runtime Callable Wrapper, RCW), которая позволяет управляемому коду использовать COM-объекты (см. рис. 8.4), и обертка, вызываемая COM-объектами (COM Callable Wrapper, CCW), дающая возможность COM-объектам вызывать управляемый код (см. рис. 8.5). Сторонние COM-объекты часто поставляются вместе с основной сборкой взаимодействий (Primary Interop Assembly, PIA), содержащей определения, одобренные производителем, подписанной и устанавливаемой в глобальный кеш сборок (Global Assembly Cache, GAC). В противном случае можно воспользоваться инструментом *tlbimp.exe*, являющийся частью Windows SDK, который автоматически генерирует сборку взаимодействий, опираясь на информацию, содержащуюся в библиотеке типов.

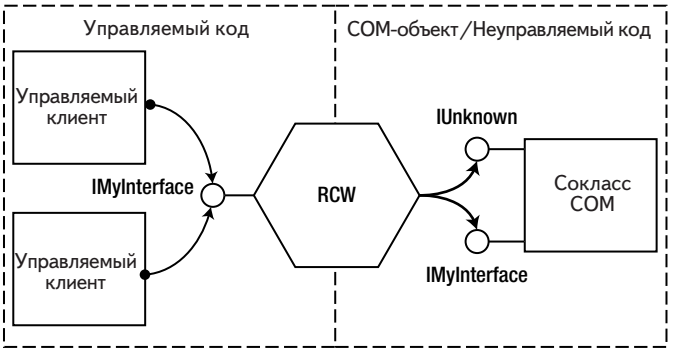

Рис. 8.4. Управляемый клиент вызывает неуправляемый COM-объект.

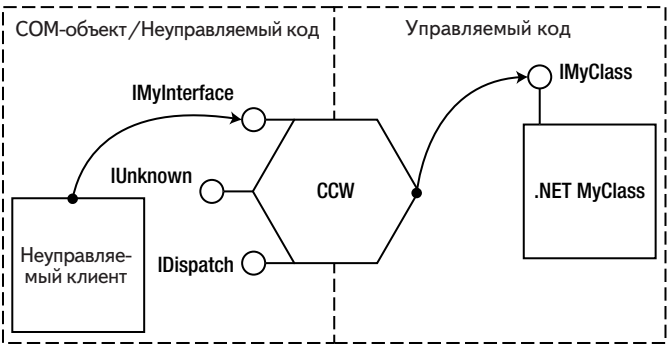

Рис. 8.5. Неуправляемый клиент вызывает управляемый COM-объект.

При взаимодействиях с COM-объектами повторно используется инфраструктура маршалинга параметров механизма P/Invoke, но с иными умолчаниями (например, по умолчанию строки преобразуются в тип `BSTR`), поэтому все советы, что были даны в этой главе относительно механизма P/Invoke, также применимы и здесь.

Модель COM имеет собственные проблемы производительности, обусловленные характерными особенностями COM, такими как многопоточная модель подразделений и несогласованность между природой COM, основанной на подсчете ссылок, и моделью сборки мусора в .NET.

Управление жизненным циклом

Получая ссылку на COM-объект в .NET, вы фактически получаете ссылку на объект-обертку RCW. Обертка RCW всегда хранит единственную ссылку на COM-объект и для каждого COM-объекта создается только один экземпляр объекта-обертки RCW. Обертка RCW поддерживает собственный счетчик ссылок, не связанный со счетчиком ссылок COM-объекта. Значение этого счетчика ссылок обычно равно 1, но может быть больше, при участии в маршалинге большего числа интерфейсов или когда к одному и тому же интерфейсу обращается несколько потоков выполнения.

Как правило, когда удаляется последняя управляемая ссылка на RCW, в следующем же цикле сборки мусора в поколении, где находится обертка RCW, вызывается финализатор RCW, который уменьшает счетчик ссылок в COM-объекте (который имеет значение 1) вызовом метода `Release` интерфейса `IUnknown` этого COM-объекта. COM-объект тут же уничтожает себя и освобождает занимаемую им память.

Поскольку сборщик мусора в .NET запускается в непредсказуемые моменты времени и не знает о блоках неуправляемой памяти, выделенных при создании оберток RCW для COM-объектов, он не может ускорить сборку мусора, вследствие чего объем занимаемой памяти может оказаться очень большим.

При необходимости можно вызвать метод `Marshal.ReleaseComObject`, чтобы явно освободить объект. Каждый вызов уменьшает счетчик ссылок в RCW и когда он достигнет нуля, автоматически уменьшается счетчик ссылок в соответствующем COM-объекте (точно так же, как при вызове финализатора RCW). После вызова метода `Marshal.ReleaseComObject` нельзя использовать обертку RCW. Если после вызова счетчик ссылок RCW может оказаться больше нуля, метод `Marshal.ReleaseComObject` следует вызывать в цикле, пока он не вернет нулевое значение. Лучше всего вызывать `Marshal.ReleaseComObject` внутри блока `finally`, чтобы гарантировать освобождение COM-объекта, даже если где-то между созданием его экземпляра и освобождением возникнет исключение

Маршалинг через границы подразделений

Модель COM реализует собственные механизмы синхронизации потоков выполнения для поддержки вызовов между разными потока-

ми, которые могут использоваться даже при работе с объектами, изначально не предназначенными для использования в многопоточной среде. Эти механизмы могут снижать производительность при неправильном их применении. Хотя эта проблема не имеет прямого отношения к взаимодействиям с COM-объектами из .NET, тем не менее, ее стоит обсудить, потому что с ней часто сталкиваются на практике, вероятно потому, что разработчики, привыкшие к типичным приемам синхронизации в .NET могут не знать, что конкретно происходит под покровом COM-объектов.

Модель COM связывает объекты и потоки выполнения с *подразделениями* (apartments), служащими границами, через которые модель COM выполняет вызовы. Всего имеется несколько типов подразделений.

- Однопоточные подразделения (Single-Threaded Apartment, STA), в каждом подразделении имеется единственный поток выполнения, но может иметься любое количество объектов. В процессе может быть несколько подразделений of STA.
- Многопоточные подразделения (Multi-Threaded Apartment, MTA), в каждом подразделении может иметься любое количество потоков выполнения и объектов, но в процессе может быть только одно подразделение MTA. Этот тип используется в .NET по умолчанию.
- Потоконезависимые подразделения (Neutral-Threaded Apartment, NTA), содержат объекты, но не потоки. В процессе может быть только одно подразделение NTA.

Связывание потока выполнения с подразделением происходит при вызове `CoInitialize` или `CoInitializeEx` для инициализации COM-объекта в этом потоке. Функция `CoInitialize` связывает поток с новым подразделением STA, а функция `CoInitializeEx` позволяет указать тип подразделения, STA или MTA. В .NET вам не придется вызывать эти функции непосредственно, вместо этого достаточно добавить атрибут `STAThread` или `MTAThread` к точке входа в поток (методу `Main`). При желании можно также вызвать метод `Thread.SetApartmentState` или установить значение в свойстве `Thread.ApartmentState` перед запуском потока выполнения. Если не указано иное, .NET инициализирует потоки (включая главный поток приложения) как принадлежащие подразделению MTA.

Связывание COM-объектов с подразделениями выполняется, исходя из параметра *ThreadingModel* в реестре, который может иметь следующие значения.

- *Single* – объект по умолчанию помещается в подразделение STA.
- *Apartment* – объект должен быть помещен в любое подразделение STA, и только потоку из этого подразделения будет позволено вызывать объект непосредственно. Другие экземпляры могут помещаться в другие подразделения STA.
- *Free* – объект помещается в подразделение MTA. Этот объект может вызываться непосредственно и одновременно из любого количества потоков в подразделении MTA. Объект должен обеспечивать поддержку использования в многопоточной среде.
- *Both* – объект помещается в подразделение создавшей его программы (STA или MTA). По сути, после создания он становится STA- или MTA-подобным объектом.
- *Neutral* – объект помещается в потоконезависимое подразделение и не требует маршалинга. Это – наиболее эффективный режим.

На рис. 8.6 изображена схема взаимоотношений между подразделениями, потоками и объектами.

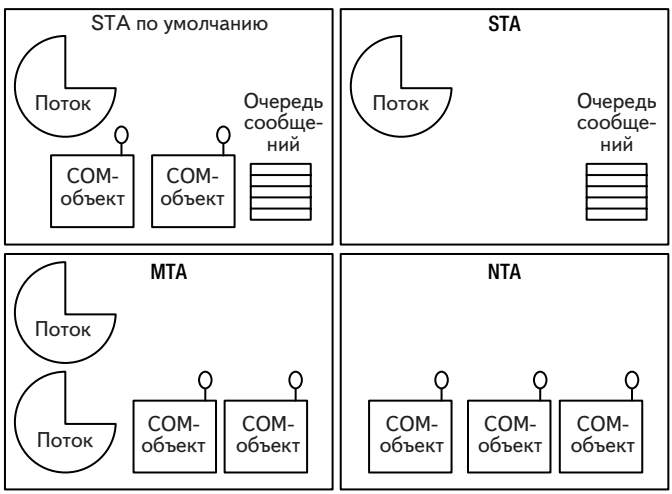

Рис. 8.6. Деление процесса на подразделения COM.

При попытке создать объект с моделью поддержки потоков, не совместимой с моделью потоков в текущем подразделении, вы получите указатель на интерфейс, который в действительности указывает на прокси-объект. Если потребуется передать интерфейс COM-объекта

другому потоку, принадлежащему другому подразделению, указатель на интерфейс должен передаваться не напрямую, а через механизм маршалинга. Инфраструктура COM вернет соответствующий прокси-объект.

В процессе маршалинга вызов функции (включая параметры) преобразуется в сообщение, которое будет послано в очередь принимающего подразделения STA. Для объектов STA очередь реализуется как скрытое окно, оконная процедура которого принимает сообщения и передает COM-объекту с помощью заглушки (stub). При таком подходе COM-объекты в подразделении STA COM всегда вызываются в одном и том же потоке выполнения, благодаря чему обеспечивается безопасность при работе в многопоточном окружении.

Если вызывающее подразделение не совместимо с подразделением COM-объекта, происходит переключение потока и выполняется маршалинг параметра между потоками.

> **Совет.** *Чтобы избежать падения производительности из-за маршалинга между потоками, старайтесь обеспечить соответствие между подразделением COM-объекта и подразделением создающего его потока. Создавайте и используйте COM-объекты STA в потоках из подразделения STA, а COM объекты из подразделения MTA – в потоках MTA. COM-объекты, помеченные, как поддерживающие оба типа подразделений, могут свободно использоваться из любых потоков выполнения без лишних накладных расходов.*

Вызов объектов STA из ASP.NET

По умолчанию среда ASP.NET выполняет страницы в потоках MTA. Если из страниц вызываются объекты, находящиеся в подразделениях STA, в дело вступает механизм маршалинга. Если основная масса используемых объектов принадлежит подразделениям STA, это приведет к деградации производительности. Эту проблему можно ликвидировать, пометив страницы атрибутом `ASPCOMPAT`, как показано ниже:

```
<%@Page Language = "vb" AspCompat = "true" %>
```

Обратите внимание, что конструкторы страниц все еще выполняются в потоке выполнения MTA, поэтому создание объектов STA следует выполнять в обработчиках событий `Page_Load` и `Page_Init`.

Импортирование библиотек типов и Code Access Security

Механизм Code Access Security выполняет те же проверки безопасности, что и P/Invoke. Вы можете добавлять ключ `/unsafe` при вызове утилиты *tlbimp.exe*, которая будет добавлять атрибут `Suppress-`

`UnmanagedCodeSecurity` к сгенерированным типам. Используйте эту возможность только в системах, пользующихся у вас безусловным доверием, так как она может порождать проблемы безопасности.

NoPIA

До выхода версии .NET Framework 4.0 приходилось вместе с приложением распространять сборки взаимодействий или основные сборки взаимодействий (Primary Interop Assemblies, PIA). Эти сборки обычно получались очень большими (даже в сравнении с кодом, использующим их) и как правило не входят в установочный комплект COM-компонентов; вместо этого их необходимо устанавливать отдельно, потому что сами они не требуются для работы самих COM-объектов. Другая причина, почему сборки PIA не включаются в установочные комплекты, состоит в том, что они устанавливаются в глобальный кеш сборок (GAC). Это вводит зависимость от .NET Framework в иначе полностью независимые приложения.

Начиная с версии .NET Framework 4.0, компиляторы C# и VB.NET могут проверить, какие COM-интерфейсы и методы используются в коде, и скопировать и встроить в вызывающую сборку только действительно необходимые определения, уменьшая размер кода и избавляя от необходимости распространять библиотеки PIA. В Microsoft эта особенность была названа NoPIA. Она действует как в отношении основных сборок взаимодействий, так и в отношении сборок взаимодействий в целом.

Сборки PIA обладают одной важной особенностью, которая называется эквивалентностью типов. Так как они имеют строгое именование и помещаются в глобальный кеш сборок, различные управляемые компоненты могут обмениваться обертками RCW и с точки зрения .NET они будут иметь эквивалентные типы. Напротив, сборки взаимодействий, сгенерированные с помощью *tlbimp.exe*, не обладают такой особенностью, так как каждый компонент в этом случае получит собственную, отдельную от других, сборку взаимодействий. С появлением поддержки особенности NoPIA отпала необходимость в строгом именовании сборок, и в Microsoft было предложено решение, позволяющее интерпретировать обертки RCW из других сборок, как принадлежащие тому же типу, если интерфейсы имеют одинаковый идентификатор GUID.

Чтобы включить поддержку NoPIA, выберите пункт **Properties** (Свойства) в контекстном меню после щелчка правой кнопкой мыши

на сборке взаимодействий в разделе **References** (Ссылки), и установите параметр **Embed Interop Types** (Внедрять типы взаимодействий) в значение **True** (см. рис. 8.7).

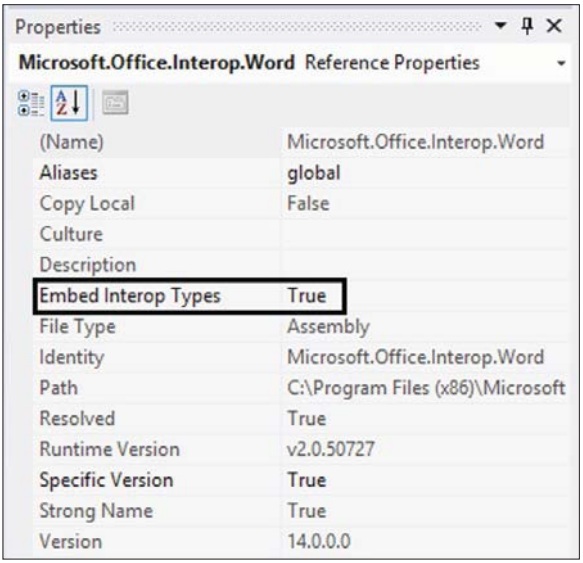

Рис. 8.7. Включение поддержки NoPIA в свойствах ссылки на сборку взаимодействий.

Исключения

Большинство методов COM-интерфейсов сообщают об успехе или неудаче, возвращая значение типа `HRESULT`. Отрицательные значения `HRESULT` (с установленным старшим битом) сообщают об ошибке, а ноль (`S_OK`) или положительные значения – об успехе. Кроме того, COM-объект может возвращать дополнительную информацию об ошибке при вызове функции `SetErrorInfo`, передавая объект `IErrorInfo`, созданный вызовом `CreateErrorInfo`. При вызове COM-метода через механизм взаимодействий с COM, заглушка маршалера преобразует значение `HRESULT` в управляемое исключение, согласно самому значению `HRESULT` и данным, содержащимся в объекте `IErrorInfo`. Поскольку возбуждение исключения является достаточно дорогостоящей операцией, функции COM-объекта, которые часто терпят неудачу, могут отрицательно сказываться на производительности. Вы можете подавить автоматическое преобразование исключений, пометив ме-

тоды атрибутом `PreserveSigAttribute`. При этом вам придется изменить управляемую сигнатуру, как возвращающую значение типа `int`, в результате чего параметр `retval` станет параметром `out`.

Расширения языка C++/CLI

C++/CLI – это набор расширений языка C++, позволяющий создавать гибридные управляемые и низкоуровневые библиотеки DLL. С применением расширений C++/CLI вы сможете определять управляемые и неуправляемые классы и функции в пределах одного файла *.cpp*, использовать управляемые и низкоуровневые типы C и C++, как в обычном программном коде на C++, то есть простым подключением заголовочного файла и связыванием с библиотекой. Эти широчайшие возможности можно использовать для создания управляемых типов-оберток, пригодных для использования в любом языке NET, а так же низкоуровневые классы-обертки и функции (доступные через файлы *.dll*, *.lib* и *.h*), пригодные для использования в программном коде на C/C++.

При использовании расширения C++/CLI маршалинг выполняется вручную, благодаря чему разработчик имеет более полный контроль и более полное представление о накладных расходах. Расширение C++/CLI с успехом можно использовать там, где механизм P/Invoke оказывается бесполезен, например, для маршалинга структур переменной длины. Еще одно преимущество расширения C++/CLI состоит в том, что оно позволяет имитировать интерфейс объединения запросов, даже если у вас нет доступа к исходным текстам вызывающего кода, многократно вызывая низкоуровневые методы без необходимости каждый раз пересекать границу между управляемым и неуправляемым кодом.

В коде, представленном ниже, мы реализовали неуправляемый класс `NativeEmployee` и управляемый класс `Employee`, служащий оберткой для первого. Управляемый код будет обращаться только к управляемому классу.

Взглянув на листинг, можно увидеть, что конструктор `Employee` демонстрирует два способа преобразования управляемых строк в неуправляемый: первый основан на выделении памяти с помощью `GlobalAlloc`, которую необходимо будет освободить явно, а второй временно закрепляет управляемую строку в памяти и возвращает прямой указатель. Второй способ выполняется быстрее, но его можно использовать, только когда неуправляемый код принимает строки,

завершающиеся нулевым символом, в кодировке UTF-16, и не записывает ничего в память по указателю. Кроме того, закрепление управляемых объектов на длительное время может привести к фрагментации памяти (см. главу 4), поэтому, если перечисленные требования не удовлетворяются, вам придется прибегнуть к копированию строк.

Метод `GetName` класса `Employee` демонстрирует три способа преобразования неуправляемых строк в управляемые: первый основан на использовании класса `System.Runtime.InteropServices.Marshal`, второй использует функцию `marshal_as template function` (будет обсуждаться ниже), объявленную в заголовочном файле `msclr/marshal.h`, и, наконец, третий использует конструктор класса `System.String`, являющийся наиболее быстрым.

Метод `DoWork` класса `Employee` принимает управляемый массив (или управляемые строки) и преобразует его в массив указателей типа `wchar_t`, указывающих на строки; фактически это массив строк в стиле языка C. Преобразование управляемых строк в неуправляемые выполняется с применением метода `marshal_as` класса объекта `marshal_context`. В отличие от глобальной функции `marshal_as`, объект `marshal_context` используется для преобразований, требующих освобождения занимаемых при этом ресурсов. Обычно для преобразования управляемых данных в неуправляемые метод `marshal_as` выделяет неуправляемую память, которую следует освободить после выполнения операции. Объект `marshal_context` содержит связанный список операций освобождения ресурсов, которые выполняются в момент уничтожения объекта.

```
#include <msclr/marshal.h>
#include <string>
#include <wchar.h>
#include <time.h>

using namespace System;
using namespace System::Runtime::InteropServices;

class NativeEmployee {
public:
    NativeEmployee(const wchar_t *employeeName, int age)
        : _employeeName(employeeName), _employeeAge(age) { }

    void DoWork(const wchar_t **tasks, int numTasks) {
        for (int i = 0; i < numTasks; i++) {
            wprintf(L"Employee %s is working on task %s\n",
                    _employeeName.c_str(), tasks[i]);
        }
```

```
    }

    int GetAge() const {
        return _employeeAge;
    }

    const wchar_t *GetName() const {
        return _employeeName.c_str();
    }

private:
    std::wstring _employeeName;
    int _employeeAge;
};

#pragma managed

namespace EmployeeLib {
    public ref class Employee {
    public:
        Employee(String ^employeeName, int age) {
          // Вариант 1:
          // IntPtr pEmployeeName =
          //   Marshal::StringToHGlobalUni(employeeName);
          // m_pEmployee = new NativeEmployee(
          //   reinterpret_cast<wchar_t *>(pEmployeeName.ToPointer()), age);
          // Marshal::FreeHGlobal(pEmployeeName);

          // Вариант 2 (прямой указатель на закрепленную
          // управляемую строку, самый быстрый):
            pin_ptr<const wchar_t> ppEmployeeName = PtrToStringChars(
                employeeName);
            _employee = new NativeEmployee(ppEmployeeName, age);
        }

        ~Employee() {
            delete _employee;
            _employee = nullptr;
        }

        int GetAge() {
            return _employee->GetAge();
        }

        String ^GetName() {
            // Вариант 1:
            // return Marshal::PtrToStringUni(
            //   (IntPtr)(void *) _employee->GetName());

            // Вариант 2:
            return msclr::interop::marshal_as<String ^>(_employee->GetName());

            // Вариант 3 (самый быстрый):
```

```
            return gcnew String(_employee->GetName());
        }

        void DoWork(array<String^>^ tasks) {
            // marshal_context - это управляемый класс, размещаемый
            // (в динамической памяти сборщика мусора)
            // с использованием семантики, напоминающей стек.
            // Его деструктор IDisposable::Dispose() будет вызван
            // после выхода из области видимости этой функции.
            msclr::interop::marshal_context ctx;
            const wchar_t **pTasks = new const wchar_t*[tasks->Length];
            for (int i = 0; i < tasks->Length; i++) {
                String ^t = tasks[i];
                pTasks[i] = ctx.marshal_as<const wchar_t *>(t);
            }
            m_pEmployee->DoWork(pTasks, tasks->Length);
            // деструктор контекста освободит неуправляемую
            // память, выделенную методом marshal_as
            delete[] pTasks;
        }

    private:
        NativeEmployee *_employee;
    };
}
```

Подводя итоги можно сказать, что расширение C++/CLI обеспечивает полный контроль над маршалингом и не требует дублирования объявлений функций, что чревато ошибками, особенно когда часто приходится изменять сигнатуры неуправляемых функций.

Вспомогательная библиотека marshal_as

В этом разделе мы остановимся на вспомогательной библиотеке `marshal_as`, входящей в состав версии Visual C++ 2008 и выше.

`marshal_as` – это библиотека шаблонов, упрощающая реализацию маршалинга управляемых типов в неуправляемые и обратно. Она способно преобразовывать многие неуправляемые строковые типы, такие как `char *`, `wchar_t *`, `std::string`, `std::wstring`, `CStringT<char>`, `CStringT<wchar_t>`, `BSTR`, `bstr_t` и `CComBSTR`, в управляемые типы и обратно. Она способна автоматически выполнять преобразование символов Юникода и ANSI, а также выделять и освобождать память.

Библиотека объявлена и реализована в файлах *marshal.h* (базовые типы), *marshal_windows.h* (типы Windows), *marshal_cppstd.h* (типы данных STL) и *marshal_atl.h* (типы данных ATL).

Имеется возможность расширять библиотеку `marshal_as` реализацией преобразований пользовательских типов. Это помогает избежать

дублирования кода, когда требуется организовать маршалинг одного и того же типа во многих местах в программе, и дает возможность обеспечить единообразие синтаксиса маршалинга разных типов.

В следующем фрагменте демонстрируется пример расширения библиотеки `marshal_as` поддержкой преобразования управляемого массива строк в эквивалентный неуправляемый массив строк.

```
namespace msclr {
    namespace interop {
        template<>
        ref class context_node<const wchar_t**, array<String^>^>
            : public context_node_base {
        private:
            const wchar_t** _tasks;
            marshal_context _context;
        public:
            context_node(const wchar_t**& toObject,
                         array<String^>^ fromObject) {

                // Здесь начинается логика преобразования
                _tasks = NULL;
                const wchar_t **pTasks =
                    new const wchar_t*[fromObject->Length];
                for (int i = 0; i < fromObject->Length; i++) {
                    String ^t = fromObject[i];
                    pTasks[i] = _context.marshal_as<const wchar_t *>(t);
                }
                toObject = _tasks = pTasks;
            }

            ~context_node() {
                this->!context_node();
            }
        protected:
            !context_node() {
                // При удалении контекста будет освобождена
                // память, выделенная для строк (и принадлежащая
                // marshal_context),поэтому массив - единственная
                // память, которую требуется освободить.
                if (_tasks != nullptr) {
                    delete[] _tasks;
                    _tasks = nullptr;
                }
            }
        };
    }
}

// Теперь можно переписать метод Employee::DoWork:
```

```
void DoWork(array<String^>^ tasks) {
    // Вся неуправляемая память освобождается автоматически,
    // как только marshal_context выйдет из области видимости.
    msclr::interop::marshal_context ctx;
    _employee->DoWork(ctx.marshal_as<const wchar_t **>(tasks),
tasks->Length);
}
```

Код на языке IL и неуправляемый код

Неуправляемый класс по умолчанию будет скомпилирован расширением C++/CLI в код на языке IL, а не в машинный код. Это может ухудшать производительность в сравнении с оптимизированным машинным кодом, потому что компилятор Visual C++ способен оптимизировать код лучше, чем JIT-компилятор.

Чтобы повлиять на процедуру компиляции можно, добавив объявление `#pragma unmanaged` или `#pragma managed` перед требуемым разделом кода. Кроме того, в проектах VC++ имеется возможность включать поддержку C++/CLI для отдельных единиц компиляции (файлов *.cpp*).

Взаимодействие со средой выполнения WinRT в Windows 8

Среда выполнения WinRT – это новая среда выполнения приложений для Windows 8 с интерфейсом в стиле Метро (Metro). WinRT имеет низкоуровневую реализацию (то есть, WinRT не использует .NET Framework), но ее можно использовать из C++/CX, языков .NET или JavaScript. Среда выполнения WinRT в значительной степени подменяет Win32 и .NET BCL, делая их недоступными. Основной упор в WinRT делается на асинхронный интерфейс, обязательный для всех операций, продолжительность которых может составлять более 50 миллисекунд. Это сделано с целью обеспечить высокую отзывчивость пользовательского интерфейса, что особенно важно для пользовательских интерфейсов в стиле Метро (Metro).

Сама среда выполнения WinRT построена поверх улучшенной версии COM. Ниже перечислены некоторые отличительные особенности WinRT:

- объекты создаются вызовом `RoCreateInstance`;
- все объекты реализуют интерфейс `IInspectable`, наследующий широко известный интерфейс `IUnknown`;

- поддерживает свойства в стиле .NET, делегаты и события (вместо объектов-приемников (sinks));
- поддерживает параметризованные интерфейсы;
- использует формат метаданных .NET (.файлы *.winmd*) вместо библиотек типов (TLB) и описаний интерфейсов на языке IDL;
- все типы наследуют `Platform::Object`.

Не смотря на заимствование множества идей из .NET, среда выполнения WinRT реализована в машинном коде, поэтому среда CLR не нужна для обращений к WinRT из языков, не относящихся к .NET.

В корпорации Microsoft были реализованы *языковые проекции* (language projections), отображающие понятия WinRT в понятия таких языков, как C++/CX, C# или JavaScript. Например, C++/CX – это новое расширение языка C++, обеспечивающее автоматический подсчет ссылок, транслирующее активацию объектов WinRT (`RoActivateInstance`) в конструкторы C++, преобразующее значения типа `HRESULT` в исключения, а аргументы «retval» в возвращаемые значения и так далее.

Когда вызывающий и вызываемый код оба являются управляемыми, среда выполнения CLR выполняет прямой вызов, минуя механизмы взаимодействий. Когда вызов осуществляется через границу управляемого и неуправляемого кода, в дело вовлекается обычный механизм взаимодействий COM. Когда обе стороны, вызывающая и вызываемая, реализованы на C++ и заголовочные файлы вызываемого кода доступны вызывающему коду, механизм взаимодействий COM не задействуется и вызов выполняется очень быстро, в противном случае используется COM-интерфейс `QueryInterface`.

Эффективные приемы взаимодействий

Ниже перечислены наиболее эффективные приемы реализации взаимодействий:

- проектируя интерфейсы, старайтесь сводить к минимуму количество переходов через границу между управляемым и неуправляемым кодом, например, объединяя задания;
- уменьшайте количество взаимодействий, совмещая несколько вызов простых функций в одном вызове;
- реализуйте интерфейс `IDisposable`, если неуправляемые ресурсы сохраняются между вызовами;

- используйте пулы памяти и выделяйте блоки неуправляемой памяти;
- используйте небезопасный код для интерпретации данных (например, в сетевых протоколах);
- явно именуйте вызываемые функции и используйте `ExactSpelling=true`;
- используйте параметры двоично совместимых типов, где это возможно;
- избегайте преобразования Юникода в ANSI, когда это возможно;
- вручную преобразуйте строки в/из `IntPtr`;
- используйте расширение C++/CLI, обеспечивающее лучшие управляемость и производительность при взаимодействиях с кодом на C/C++ и COM-объектами;
- указывайте атрибуты `[In]` и `[Out]`, чтобы избежать ненужного маршалинга;
- избегайте закрепления долгоживущих объектов;
- используйте метод `ReleaseComObject` при необходимости;
- используйте атрибут `SuppressUnmanagedCodeSecurityAttribute` в окружениях, заслуживающих доверия;
- используйте утилиту *tlbimp.exe* с ключом `/unsafe` в окружениях, заслуживающих доверия;
- избегайте или старайтесь уменьшать количество вызовов через границы подразделений COM;
- при возможности используйте атрибут `ASPCOMPAT` в приложениях ASP.NET, чтобы уменьшить количество вызовов через границы подразделений COM.

В заключение

В этой главе вы познакомились с небезопасным кодом, особенностями реализации различных механизмов взаимодействий, влиянием на производительность каждой из особенностей и узнали, как можно ослабить это влияние. Вашему вниманию были представлены наиболее эффективные приемы увеличения производительности операций взаимодействий и упрощения их реализации (например, с помощью библиотеки `marshal_as` и генератора сигнатур для P/Invoke).

ГЛАВА 9.
Оптимизация алгоритмов

Основу некоторых приложений составляют специализированные алгоритмы решения задач той или иной предметной области и на основе допущений, не являющихся универсальными. Другие приложения опираются на хорошо выверенные алгоритмы, пригодные для использования в самых разных областях и десятки лет применявшиеся при разработке программного обеспечения. Мы считаем, что любому программисту будет полезно погрузиться в исследование наиболее блистательных алгоритмов, а также категорий алгоритмов, на которых основываются программные инфраструктуры. Некоторые разделы этой главы могут показаться слишком сложными для тех, кто не имеет серьезной математической подготовки, тем не менее, ее стоит прочитать.

Эта глава лишь слегка касается некоторых основных положений информатики, знакомит с несколькими бессмертными алгоритмами и анализирует их сложность. Опираясь на эти примеры, вы более свободно сможете пользоваться существующими алгоритмами, адаптировать их под свои потребности и изобретать собственные.

> **Примечание.** *Это не учебник по исследованию алгоритмов и не введение в наиболее важные алгоритмы в современной информатике. Мы сознательно сделали эту главу максимально короткой, чтобы у вас не сложилось впечатление, что здесь вы сможете получить все необходимые сведения. Мы не будем углубляться в формальные определения. Например, наша интерпретация машин и языков Тьюринга далека от официального определения. В качестве учебных пособий по алгоритмам мы можем посоветовать книгу Кормена (Cormen), Лайзерсона (Leiserson), Ривеста (Rivest) и Штайна (Stein) «Introduction to Algorithms» (MIT Press, 2001)*[1] *и книгу Дасгупта (Dasgupta), Пападимитриу (Papadimitriou) и Вазирани (Vazirani) «Algorithms» (выход ожидается в самое ближайшее время, но черновой вариант уже доступен в Интернете).*

1 Т. Кормен, Ч. Лейзерсон, Р. Ривест, К. Штайн «Алгоритмы. Построение и анализ», ISBN: 5-8459-0857-4, Вильямс, октябрь 2011. – *Прим. перев.*

Систематизация сложности

В главе 5 немного говорилось о сложности операций, поддерживаемых коллекциями в .NET Framework и нашими собственными реализациями коллекций. В этом разделе мы подробнее расскажем, что означает «большое O» (Big-Oh) и познакомим вас с основными классами сложности, известными в информатике и теории алгоритмов.

Большое O

Обсуждая сложность операции поиска в списке `List<T>`, в главе 5, мы сказали, что он имеет сложность $O(n)$. Если говорить простым языком, это означает, что для поиска некоторого конкретного элемента в списке с 1000 элементов в самом худшем случае (когда искомый элемент отсутствует в списке) потребуется выполнить 1000 итераций. То есть, нотация «большое O» – это оценка роста времени, необходимого на выполнение алгоритма, с увеличением объема входных данных. Однако, формальное определение может показаться немного странным:

> Предположим, что функция T(A;n) выполняет несколько итераций, необходимых для вычислений по алгоритму A с входными данными, содержащими n элементов. Пусть f(n) – монотонная функция с областью определения в множестве положительных чисел. Тогда, T(A;n) – это O(f(n)), если существует такая константа c, когда для всех n выполняется соотношение $T(A;n) \le cf(n)$.

Проще говоря, сложность алгоритма обозначается как $O(f(n))$, где $f(n)$ является верхней оценкой фактического количества итераций алгоритма, когда объем входных данных равен n. Эта оценка не должна быть жесткой; например, можно сказать, что поиск по списку `List<T>` имеет сложность $O(n^4)$. Однако, применение такой свободной оценки не очень полезно, потому что не объясняет реальную возможность выполнить поиск по списку `List<T>`, даже если он содержит 1 000 000 элементов. Если бы поиск по списку `List<T>` имел жестко заданную сложность $O(n^4)$, он оказался бы крайне неэффективным даже для списков, содержащих несколько тысяч элементов.

Кроме того, верхняя оценка может быть жесткой для одних входных данных и не жесткой для других; например, если при поиске элемента в списке, который по случайному совпадению оказывается первым, количество итераций будет постоянным (равным единице!) для любого количества элементов в списке – именно поэтому в предыдущих абзацах мы говорили о худшем случае.

Ниже перечислены некоторые примеры, как эта нотация помогает оценивать время выполнения и сравнивать различные алгоритмы.

- Если один алгоритм имеет сложность $2n^3 + 4$, а другой – сложность $Sn^3–n^2$, мы можем сказать, что оба алгоритма имеют сложность $O(n^3)$ и они эквивалентны в смысле нотации большого O (попробуйте найти константу c для каждой из этих сложностей). Легко доказать, что говоря о нотации большого O, мы можем опустить любые члены, кроме наибольшего.
- Если один алгоритм имеет сложность n^2, а другой – сложность $100n + 5000$, можно смело утверждать, что первый алгоритм медленнее второго для больших объемов входных данных, потому что в нотации большого O он имеет сложность $O(n^2)$, тогда как второй – сложность $O(n)$. Действительно, уже для случая $n = 1000$ первый алгоритм оказывается намного медленнее второго.

По аналогии с верхней оценкой сложности, существует также нижняя оценка (обозначается как $W(f(n))$) и жесткая оценка (обозначается как $\Theta(f(n))$). Однако эти оценки реже используются при обсуждении сложности алгоритмов, поэтому мы не будем касаться их.

Основная теорема

Основная теорема (master theorem) – это просто результат, обеспечивающий готовое решение для анализа сложности многих рекурсивных алгоритмов, разбивающих решение большой задачи на множество маленьких подзадач. Например, взгляните на следующий код, реализующий алгоритм *сортировки слиянием* (merge sort algorithm):

```
public static List<T> MergeSort(List<T> list) where T :
IComparable<T> {
    if (list.Count <= 1) return list;
    int middle = list.Count / 2;
    List<T> left = list.Take(middle).ToList();
    List<T> right = list.Skip(middle).ToList();
    left = MergeSort(left);
    right = MergeSort(right);
    return Merge(left, right);
}

private List<T> Merge(List<T> left, List<T> right) where T :
IComparable<T> {
    List<T> result = new List<T> ();
    int i = 0, j = 0;
    while (i < left.Count || j < right.Count) {
        if (i < left.Count && j < right.Count) {
```

```
            if (left[i].CompareTo(right[j]) <= 0)
                result.Add(left[i++]);
            else
                result.Add(right[j++]);
        } else if (i < left.Count) {
            result.Add(left[i++];
        } else {
            result.Add(right++);
        }
    }
    return result;
}
```

Чтобы определить сложность этого алгоритма, необходимо решить рекуррентное соотношение относительно времени его выполнения, $T(n)$, которое задано рекурсивно, как $T(n) = 2T(n/2) + O(n)$. Это объясняется тем, что каждый вызов `MergeSort` рекурсивно вызывает `MergeSort` для обеих половин оригинального списка и тратит линейное время на слияние списков (очевидно, что вспомогательный метод `Merge` выполняет точно n операций для списка, имеющего размер n).

Один из подходов к решению рекуррентных соотношений заключается в прогнозировании результата с последующей попыткой доказать его правильность (обычно с применением математической индукции). В данном случае мы можем развернуть некоторые члены и обнаружить следующую закономерность:

```
T(n)= 2T(n/2)+O(n)=2(2T(n/4)+O(n/2))+O(n)=2(2(2T(n/8)+O(n/4))+
O(n/2))+O(n)=...
```

Основная теорема дает конечное решение этого и многих других рекуррентных соотношений. Согласно основной теореме, $T(n) = O(n \log n)$, – хорошо известное значение сложности алгоритма сортировки слиянием (фактически, оно также является оценкой сложности Θ, как и O). За дополнительной информацией об основной теореме обращайтесь по адресу: http://en.wikipedia.org/wiki/Master_theorem.

Машины Тьюринга и классы сложности

Об алгоритмах и задачах часто говорят, что они принадлежат классу «P», или «NP», или «NP-полному». Так обозначаются различные классы сложности. Классификация задач по сложности помогает программистам выделять задачи, имеющие простые решения, и отвергать сложные задачи или искать упрощенные подходы к их решению.

Машина Тьюринга (MT) – это абстрактная вычислительная машина, моделирующая машину, обрабатывающую бесконечную *ленту* с символами на ней. Головка машины может читать или записывать символы на ленту, по одному за раз, а сама машина может находиться в одном из конечного числа *состояний*. Перечень операций, выпол-

няемых машиной, полностью определяется конечным количеством *правил* (алгоритмом), таких как «когда в состоянии Q с ленты прочитан символ 'A', записать символ 'a'» или «когда в состоянии P с ленты прочитан символ 'a', переместить головку вправо и перейти в состояние S». Существует также два специальных состояния: *начальное состояние*, в котором машина находится перед выполнением любых операций, и *конечное состояние*. Когда машина достигает конечного состояния, говорят, что она попала в бесконечный цикл или просто завершила выполнение. На рис. 9.1 изображен пример определения машины Тьюринга – окружности – это состояния, а стрелками показаны переходы из одного состояния в другое в виде последовательностей операций `чтение;запись;направление_перемещения_головки`.

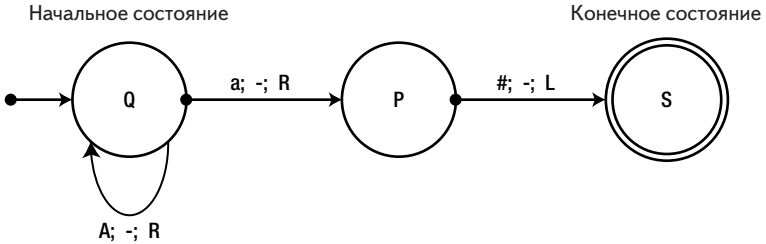

Рис. 9.1. Диаграмма состояний простой машины Тьюринга. Самая левая стрелка указывает на начальное состояние Q. Стрелка, образующая петлю, описывает ситуацию, когда операция чтения символа A в состоянии Q возвращает машину в состояние Q.

При обсуждении сложности алгоритмов на машине Тьюринга нет необходимости рассуждать в терминах абстрактных «итераций» – шагом вычислений является единственный переход из состояния в состояние (включая переходы в то же самое состояние).

Например, когда машине Тьюринга на рис. 9.1 подается на вход лента с исходными данными «AAAa#», она выполняет ровно четыре шага. Мы можем обобщить эту ситуацию и сказать, что когда на вход подается последовательность символов «A», за которой следуют символы «a#», машина выполняет $O(n)$ шагов. (В действительности, для исходных данных, имеющих объем n, машина выполняет точно $n + 2$ шагов, то есть определение $O(n)$ включает, например, константу $c = 3$.)

Моделирование настоящих вычислений в терминах машины Тьюринга является достаточно сложной задачей, Она предназначена служить примером при изучении теории автоматов, но не для прак-

тического использования. Самое удивительное, что любую программу на языке C# (а в действительности и любой алгоритм, который можно выполнить на современном компьютере) можно выразить – приложив определенные усилия – в терминах операций машины Тьюринга. С некоторыми допущениями можно утверждать, что если алгоритм, реализованный на C#, имеет сложность $O(f(n))$, его можно переложить на язык машины Тьюринга, где он будет иметь сложность $O(f^2(n))$. Из этого утверждения выводятся очень интересные следствия: если существует эффективный алгоритм решения задачи на машине Тьюринга, существует не менее эффективный алгоритм решения той же задачи на современном компьютере; если задача не имеет эффективного алгоритма решения на машине Тьюринга, для нее обычно не существует эффективного алгоритма решения на современном компьютере.

Мы могли бы назвать алгоритмы $O(n^2)$ эффективными, а все «более медленные» алгоритмы неэффективными, однако в теории сложности используется несколько иной подход. P – это множество всех задач, которые можно решить на машине Тьюринга за полиномиальное время. Иными словами, если A – это задача, принадлежащая множеству P (с объемом входных данных n), существует алгоритм решения ее на машине Тьюринга, который позволит получить желаемый результат на ленте за полиномиальное время (например, за $O(n^k)$ шагов, где k – некоторое натуральное число). В теории сложности задачи, принадлежащие множеству P, считаются простыми, а алгоритмы, позволяющие получить результат за полиномиальное время – эффективными, даже если k и, соответственно, время выполнения, могут оказаться очень большими.

С позиции этого определения, все алгоритмы, рассматривавшиеся до сих пор в этой книге, были эффективными. Однако некоторые из них являются «более» эффективными, другие – менее, что говорит о не очень четкой градации. Вы можете даже спросить: «Существуют ли задачи, не попадающие в множество P, задачи, не имеющие эффективного решения?». Ответ на этот вопрос: «Да». В действительности, с точки зрения теории, задач, не имеющих эффективного решения, *больше*, чем задач, имеющих такие решения.

Для начала рассмотрим задачу, которую нельзя решить на машине Тьюринга, независимо от эффективности алгоритма. Затем исследуем задачи, которые могут быть решены на машине Тьюринга, но не за полиномиальное время. И, наконец, обратимся к задачам, для которых *неизвестно*, существует ли машина Тьюринга, способная решить

их за полиномиальное время, и будем предполагать, что таких машин не существует.

Задача об остановке

С математической точки зрения, задач намного больше, чем машин Тьюринга (мы говорим, что машины Тьюринга поддаются «перечислению», а задачи – нет). Это означает, что существует бесконечное множество задач, которые не могут быть решены машиной Тьюринга. Такие задачи часто называют *неразрешимыми*.

Что значит «поддаются перечислению»?

В математике существует множество разных «бесконечностей». Очевидно, что количество машин Тьюринга бесконечно – в конце концов, в любую машину Тьюринга можно добавить новое фиктивное состояние, чтобы получить новую машину Тьюринга, и так до бесконечности. Точно так же очевидно, что существует бесконечное количество задач – для этого требуется создать новое формальное определение задачи («языка»), но суть от этого не меняется. Однако, совершенно не очевидно, почему задач «больше», чем машин Тьюринга, особенно если учесть, что количество тех и других бесконечно.

Говорят, что множество машин Тьюринга *поддается перечислению*, потому что между натуральными числами (1, 2, 3, ...) и машинами Тьюринга существует однозначное соответствие. Возможно, не совсем очевидно, как строится это соответствие, но оно существует, потому что машины Тьюринга описываются конечными строками, а множество *всех* конечных строк поддается перечислению.

Множество задач (языков), напротив, *не* поддается перечислению, потому что отсутствует однозначное соответствие между натуральными числами и языками. Одно из возможных доказательств приводится в следующих строках: представьте множество задач, соответствующих всем вещественным числам, где любому вещественному числу r соответствует задача вывести это число или определить, поступало ли такое число на вход раньше. Результат хорошо известен (теорема Кантора) – вещественные числа не поддаются перечислению и, соответственно, данное множество задач также не поддается перечислению.

Все это напоминает неудачное заключение. Не только существуют задачи, которые не могут быть решены машиной Тьюринга, но их намного *больше*, чем задач, которые могут быть решены таким способом. К счастью, вопреки теоретическим выкладкам, огромное количество задач *поддаются* решению машинами Тьюринга, как показало невероятное развитие компьютерной техники в 20 столетии.

Задача об остановке, которую мы сейчас рассмотрим, относится к разряду неразрешимых. Задача состоит в следующем: на вход подаются программа T (или описание машины Тьюринга) и исходные

данные *w*; на выходе возвращается TRUE, если программа *T* завершит когда-либо обработку *w*, и FALSE, если не завершится (войдет в бесконечный цикл).

Решение этой задачи можно даже выразить на языке C#, в виде метода, принимающего код программы в виде строки:

```
public static bool DoesHaltOnInput(string programCode, string input){...}
```

...или даже в виде метода, принимающего делегата и исходные данные для него:

```
public static bool DoesHaltOnInput(Action<string> program, string input){...}
```

На первый взгляд кажется, что программу можно проанализировать и определить, завершится она когда-нибудь или нет (например, исследовав в ней циклы, вызовы других и так далее), но, как оказывается, нет ни машины Тьюринга, ни программ на C#, способных решить эту задачу. Как мы пришли к такому заключению? Когда мы говорим, что существует машина Тьюринга, способная решить некоторую задачу, в доказательство достаточно просто продемонстрировать ее, но, чтобы доказать, что не существует машины Тьюринга, способной решить задачу, похоже нужно перебрать и проверить все возможные машины Тьюринга, которых бесконечное множество.

Как это распространено в математике, воспользуемся способом доказательства от противного. Допустим, что кто-то написал метод DoesHaltOnInput, выполняющий требуемую проверку. Тогда мы могли бы написать такой метод:

```
public static void Helper(string programCode, string input) {
    bool doesHalt = DoesHaltOnInput(programCode, input);
    if (doesHalt) {
        while (true) {} // Вход в бесконечный цикл
    }
}
```

Теперь добавим вызов DoesHaltOnInput во вспомогательный метод Helper. Если DoesHaltOnInput вернет true, метод Helper войдет в бесконечный цикл; если DoesHaltOnInput вернет false, метод Helper завершится как обычно. Это противоречие доказывает, что метод DoesHaltOnInput не существует.

> **Примечание.** *Попытка решить задачу об остановке приводит к удивительному выводу; она просто и наглядно доказывает ограниченность возможностей наших вычислительных устройств. В следующий раз, когда вы решите ругнуться в адрес компилятора, что он не нашел очевидно тривиальную оптимизацию, или в адрес инструмента статического анализа, что*

вывел ложные предупреждения о том, чего никогда не должно произойти, вспомните, что статический анализ программы и выбор действия в зависимости от результатов часто является неразрешимой задачей. Именно так обстоит дело с оптимизацией, ошибками при анализе кода и в определении использования переменной, а так же со многими другими задачами, которые легко решаются человеком, но оказываются не под силу машине.

Существует множество других неразрешимых задач. Другой простой пример неразрешимой задачи – задача, связанная с определением перечислимости аргумента. Множество программ на C# поддается перечислению, потому что каждая программа представлена конечным количеством символов. Однако в интервале [0,1] существует неперечислимое множество вещественных чисел. Соответственно, должно быть вещественное число, которое не может быть выведено программой.

NP-полные задачи

Даже среди множества разрешимых задач – которые могут быть решены машиной Тьюринга – существуют задачи, не имеющие эффективного решения. Сложность выбора наилучшей стратегии в игре в шахматы на доске размером $n \times n$ растет в *экспоненциальной* прогрессии от n, что выводит задачу реализации игры в шахматы за пределы класса *P*. (Если вы любите играть в шашки, но не любите, когда компьютерные программы играют в шашки лучше людей, пусть вас утешит знание, что задача реализации игры в шашки также не попадает в класс *P*.)

Однако существуют задачи, считающиеся менее сложными, но для решения которых все равно не существует полиномиальных алгоритмов. Некоторые из них с успехом могли бы использоваться в реальных приложениях:

- *задача коммивояжера*: поиск самого выгодного маршрута, проходящего через n разных городов;
- *задача о клике*: поиск наибольшего подмножества вершин в графе, в котором каждые две вершины соединены ребром;
- *задача о минимальном разрезе*: поиск способа деления графа на два подмножества узлов, когда границу подмножеств пересекает минимальное количество ребер;
- *задача пропозициональной выполнимости*: определить, может ли быть удовлетворена логическая формула определенной формы (например: «*A* и *B* или *C* и не *A*») присваиванием истинных значений ее переменным;

- *задача включения в кеш*: определить, какие данные поместить в кеш, а какие убрать из кеша, имея полную хронологию обращений к памяти из приложения.

Эти задачи принадлежат другому множеству задач – классу *NP*. Задачи из класса *NP* характеризуются следующим образом: если *A* – это задача, принадлежащая классу *NP*, тогда существует машина Тьюринга, которая может *проверить* решение задачи *A* для исходных данных объемом *n* за полиномиальное время. Например, легко можно проверить, допустимо ли присваивание истинного значения переменным и решает ли это задачу пропозициональной выполнимости, причем сложность такой проверки прямо пропорциональна количеству переменных. Аналогично, легко проверить, является ли подмножество узлов графа кликой. Иными словами, эти задачи имеют легко проверяемые решения, но не известно, являются ли эффективными эти решения.

Другая интересная особенность задач выше (и многих других), состоит в том, что если какая-либо из них имеет эффективное решение, тогда *все* они имеют эффективное решение. Объясняется это тем, что они могут быть *выражены* друг через друга. Кроме того, если *любая* из этих задач имеет эффективное решение, это означает, что задача принадлежит классу *P*, и тогда весь класс сложности *NP* целиком сжимается до размеров класса *P* так, что выполняется равенство $P = NP$. На сегодняшний день, ответ на вопрос о равенстве $P = NP$ является самой большой загадкой в теоретической информатике (большинство ученых полагают, что эти классы сложности не равны).

Задачи, проявляющие эффект свертывания класса *NP* в класс *P*, называют *NP-полными задачами*. Если задача оказывается *NP*-полной, для большинства программистов это достаточное основание, чтобы отказаться от попыток найти более эффективный алгоритм ее решения. В последующих разделах будет представлено несколько примеров *NP*-полных задач, имеющих вполне приемлемое приближенное или вероятностное решение.

Мемоизация и динамическое программирование

Мемоизация – это прием сохранения промежуточных результатов, которые могут еще раз понадобиться в ближайшее время, чтобы избежать их повторного вычисления. Этот прием можно рассматривать

как разновидность кеширования. Классическим примером применения мемоизации является вычисление последовательности чисел Фибоначчи, которое часто служит первым примером при изучении рекурсии:

```
public static ulong FibonacciNumber(uint which) {
    if (which == 1 || which == 2) return 1;
    return FibonacciNumber(which-2) + FibonacciNumber(which-1);
}
```

Этот метод выглядит просто замечательно, но его производительность хуже некуда. Даже для такого небольшого числа, как 45, этот метод тратит на вычисления несколько секунд. На этом фоне попытка отыскать 100 первых чисел Фибоначчи с помощью данного метода выглядит просто нереальной, так как его сложность растет в экспоненциальной прогрессии.

Одна из причин такой неэффективности состоит в том, что промежуточные результаты вычисляются более одного раза. Например, `FibonacciNumber(10)` вычисляется рекурсивно при попытке вычислить `FibonacciNumber(11)` и `FibonacciNumber(12)`, и затем при попытке вычислить `FibonacciNumber(12)` и `FibonacciNumber(13)`, и так далее. Сохраняя промежуточные результаты в массиве можно существенно повысить производительность этого метода:

```
public static ulong FibonacciNumberMemoization(uint which) {
    if (which == 1 || which == 2) return 1;
    ulong[] array = new ulong[which];
    array[0] = 1; array[1] = 1;
    return FibonacciNumberMemoization(which, array);
}

private static ulong FibonacciNumberMemoization(uint which, ulong[] array) {
    if (array[which-3] == 0) {
        array[which-3] = FibonacciNumberMemoization(which-2, array);
    }
    if (array[which-2] == 0) {
        array[which-2] = FibonacciNumberMemoization(which-1, array);
    }
    array[which-1] = array[which-3] + array[which-2];
    return array[which-1];
}
```

Эта версия находит 10 000-е число Фибоначчи за доли секунды и имеет линейную сложность. Кстати говоря, эту реализацию можно упростить еще больше, сохраняя только два последних вычисленных числа:

```
public static ulong FibonacciNumberIteration(ulong which) {
    if (which == 1 || which == 2) return 1;
    ulong a = 1, b = 1;
    for (ulong i = 2; i < which; ++i) {
        ulong c = a + b;
        a = b;
        b = c;
    }
    return b;
}
```

Примечание. *Следует заметить, что существует приближенная формула вычисления чисел Фибоначчи, основанная на формуле золотого сечения (http://en.wikipedia.org/wiki/Fibonacci_number#Closed-form_expression). Однако использование этой формулы для поиска точных значений может потребовать нетривиальных математических вычислений.*

Простой прием сохранения промежуточных результатов для последующих вычислений может пригодиться во многих алгоритмах, разбивающих большую задачу на множество маленьких подзадач. Его часто называют *динамическим программированием* (dynamic programming). Теперь рассмотрим два примера.

Расстояние Левенштейна

Расстояние Левенштейна между двумя строками – это количество операций с символами (удаление, вставка и замена), которое требуется выполнить, чтобы преобразовать одну строку в другую. Например, расстояние Левенштейна между строками «cat» и «hat» равно 1 (достаточно заменить «c» на «h»), а расстояние Левенштейна между «cat» и «groat» равно 3 (вставить «g», вставить «r», заменить «c» на «o»). Эффективный поиск расстояния Левенштейна между двумя строками играет важную роль во многих ситуациях, таких как правка ошибок и проверка правописания с предложением строк для замены.

Ключом к эффективности алгоритма является разбиение большой задачи на множество маленьких. Например, если известно, что расстояние Левенштейна между «cat» и «hat» равно 1, следовательно расстояние Левенштейна между «cats» и «hat» будет равно 2 – мы воспользовались результатом уже решенной подзадачи, чтобы найти решение более крупной задачи. Применять этот прием на практике следует с большой осторожностью. Для двух строк, представленных в виде массивов, *s[1 ... m]* и *t[1 ... n]*, выполняются следующие правила:

- расстояние Левенштейна между пустой строкой и строкой *t* равно *n*, а расстояние между строкой *s* и пустой строкой равно *m* (количество добавлений или удалений всех символов);
- если *s[i] = t[j]* и расстояние между *s[1 ... i-1]* и *t[1 ... j-1]* равно *k*, мы можем сохранить *i*-й символ и принять расстояние между *s[1 ... i]* и *t[1 ... j]* как равное *k*;
- если *s[i] ≠ t[j]*, расстояние между *s[1 ... i]* и *t[1 ... j]* будет не меньше, чем:
  - расстояние между *s[1 ... i]* и *t[1 ... j-1]*, +1 для вставки *t[j]*;
  - расстояние между *s[1 ... i-1]* и *t[1 ... j]*, +1 для удаления *s[i]*;
  - расстояние между *s[1 ... i-1]* и *t[1 ... j-1]*, +1 для замены *s[i]* на *t[j]*.

Следующий метод на C# находит расстояние Левенштейна между двумя строками путем конструирования таблицы расстояний для каждой подстроки, и возвращает расстояние из итоговой ячейки таблицы:

```
public static int EditDistance(string s, string t) {
    int m = s.Length, n = t.Length;
    int[,] ed = new int[m,n];

    for (int i = 0; i < m; ++i) {
        ed[i,0] = i + 1;
    }
    for (int j = 0; j < n; ++j) {
        ed[0,j] = j + 1;
    }
    for (int j = 1; j < n; ++j) {
        for (int i = 1; i < m; ++i) {
            if (s[i] == t[j]) {
                ed[i,j] = ed[i-1,j-1]; // Операция не требуется
            } else { // Минимум между удалением, вставкой и заменой
                ed[i,j] = Math.Min(ed[i-1,j] + 1, Math.Min(ed[i,j-1] + 1,
                                                           ed[i-1,j-1] + 1));
            }
        }
    }
    return ed[m-1,n-1];
}
```

Алгоритм заполняет таблицу расстояний по столбцам, то есть никогда не пытается использовать еще не вычисленные данные. На рис. 9.2 показана таблица расстояний, построенная алгоритмом для строк «stutter» и «glutton».

	g	l	u	t	t	o	n
s	1	2	3	4	5	6	7
t	2	2	3	3	4	5	6
u	3	3	2	3	4	5	6
t	4	4	3	2	3	4	5
t	5	5	4	3	2	3	4
e	6	6	5	4	3	3	4
r	7	7	6	5	4	4	4

Рис. 9.2. Заполненная таблица расстояний Левенштейна.

Этот алгоритм использует пространство памяти $O(mn)$ и имеет сложность $O(mn)$. Для сравнения, рекурсивное решение, не использующее прием мемоизации, имеет экспоненциальную сложность и показывает чрезвычайно низкую производительность даже для строк среднегоразмера.

Кратчайший путь между всеми парами вершин

Задача о *кратчайшем пути между всеми парами вершин* заключается в поиске кратчайшего расстояния для каждой пары вершин в графе. Алгоритм решения этой задачи может пригодиться для проектирования перекрытий заводских цехов, оценки расстояния поездки по нескольким городам, оценки стоимости топлива, которое потребуется для поездки, и во многих других ситуациях. Авторам довелось столкнуться с этой задачей в своей работе (см. статью в блоге Саши Голдштейна (Sasha Goldshtein) по адресу: http://blog.sashag.net/archive/2010/12/16/all-pairs-shortest-paths-algorithm-in-reallife.aspx). Далее приводится описание проблемы, полученное от нашего клиента.

- Мы реализовали службу управления несколькими физическими устройствами резервного копирования. Имеется несколько пересекающихся конвейеров и роботизированных манипуляторов для захвата кассет с резервными копиями. Служба принимает запросы вида: «переместить самую свежую ленту X из устройства 13 в устройство 89, гарантировав попутное прохождение ленты через станцию форматирования C или D».

- На запуске система вычисляет кратчайшие пути между всеми парами устройств, включая специальные требования, такие как обязательное прохождение через определенный компьютер. Эта информация хранится в огромной хеш-таблице с описаниями маршрутов в качестве индексов и маршрутами в качестве значений.
- Запуск системы для 1000 узлов и 250 пересечений занимает более 30 минут, а объем потребляемой памяти при этом достигает 5 Гбайт. Это совершенно неприемлемо.

Прежде всего мы обратили внимание, что ограничение «гарантировав попутное прохождение ленты через станцию форматирования *C* или *D*» не является серьезной дополнительной задачей. Кратчайший путь из точки *A* в точку *B* через точку *C* есть кратчайший путь из точки *A* в точку *C* плюс кратчайший путь из точки *C* в точку *B* (доказать это элементарно просто).

Алгоритм Флойда-Уоршелла (Floyd-Warshall) находит кратчайшие пути между всеми парами вершин в графе и использует прием разделения большой задачи на несколько более мелких подзадач, подобно тому, как это делалось выше. На этот раз рекурсивная формула использует то же наблюдение, сделанное выше: кратчайший путь из точки *A* в точку *B* лежит через некоторую вершину V. Тогда, чтобы найти кратчайший путь из *A* в *B*, необходимо сначала найти кратчайший путь из *A* в *V*, потом кратчайший путь из *V* в *B* и, наконец, объединить их. Так как заранее неизвестно, какое устройство будет играть роль вершины *V*, нам нужно рассмотреть все возможные промежуточные устройства. Для этого мы можем пронумеровать их от 1 до *n*.

Теперь длина кратчайшего пути (Shortest Path, SP) из вершины *i* в вершину *j* через одну из обязательных вершин 1, ..., *k* определяется следующей рекурсивной формулой, предполагающей отсутствие ребра, связывающего вершины *i* и *j*:

$$SP(i, j, k) = min \{ SP(i, j, k-1), SP(i, k, k-1) + SP(k, j, k-1) \}$$

Чтобы разобраться в ней, рассмотрим вершину *k*. Кратчайший путь из *i* в *j* либо лежит через эту вершину, либо нет. Если путь через вершину *k* не является кратчайшим, мы не должны использовать его и следует попытаться найти более короткий путь через одну из вершин 1, ..., *k*–1. Если путь через вершину *k* является кратчайшим, мы приходим к необходимости разделения задачи – кратчайший путь можно получить путем объединения кратчайшего пути из *i* в *k* (использую-

щего только вершины 1, ..., k–1) и кратчайшего пути из k в j (использующего только вершины 1, ..., k–1). Пример приводится на рис. 9.3.

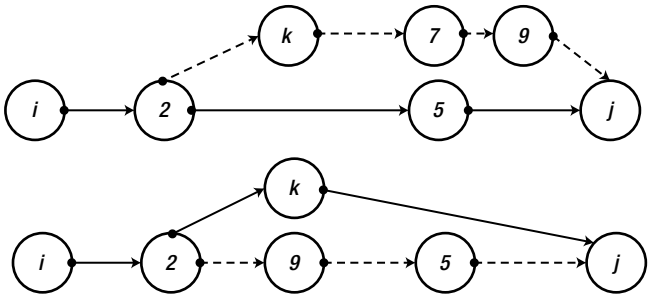

Рис. 9.3. Вверху показан кратчайший путь из i в j (через вершины 2 и 5). Так как путь через вершину k не является кратчайшим, мы можем ограничить поиск кратчайшего пути вершинами 1, ..., k–1. Внизу показан кратчайший путь из i в j, лежащий через вершину k. Этот путь из i в j можно составить из кратчайших путей из i в k и из k в j.

Чтобы избавиться от рекурсивных вызовов, будем использовать мемоизацию – на этот раз нам предстоит заполнить трехмерную таблицу, и, после поиска значений SP для всех пар вершин и всех значений k, мы получим решение задачи о кратчайшем пути между всеми парами вершин.

Мы можем еще больше уменьшить объем используемой памяти, так как для каждой пары вершин i и j нам достаточно запомнить лишь значение k, дающее кратчайший путь. В результате таблица становится двумерной, занимающей объем памяти $O(n^2)$. Однако сложность алгоритма остается равной $O(n^3)$ – слишком высокой, учитывая, что нам требуется найти кратчайшие пути между всеми вершинами в графе.

Наконец, при заполнении таблицы, в процессе поиска кратчайшего пути между каждой парой вершин i и j нам необходимо запоминать, какую следующую вершину x следует обработать. Опираясь на последнее наблюдение, эти идеи легко воплотить в код на C#:

```
static short[,] costs;
static short[,] next;

public static void AllPairsShortestPaths(short[] vertices,
bool[,] hasEdge) {
    int N = vertices.Length;
    costs = new short[N, N];
    next = new short[N, N];
```

```
    for (short i = 0; i < N; ++i) {
        for (short j = 0; j < N; ++j) {
            costs[i, j] = hasEdge[i, j] ? (short)1 : short.MaxValue;
            if (costs[i, j] == 1)
                next[i, j] = -1; // Пометить направленное ребро
        }
    }
    for (short k = 0; k < N; ++k) {
        for (short i = 0; i < N; ++i) {
            for (short j = 0; j < N; ++j) {
                if (costs[i, k] + costs[k, j] < costs[i, j]) {
                    costs[i, j] = (short)(costs[i, k] + costs[k, j]);
                    next[i, j] = k;
                }
            }
        }
    }
}

public string GetPath(short src, short dst) {
    if (costs[src, dst] == short.MaxValue) return " < no path > ";
    short intermediate = next[src, dst];
    if (intermediate == -1)
        return "- > "; // Прямой путь
    return GetPath(src, intermediate) + intermediate +
GetPath(intermediate, dst);
}
```

Этот простой алгоритм значительно увеличивает производительность приложения. Для случая с 300 узлами, когда из каждого узла в среднем выходит три ребра, создание полного набора путей занимает 3 секунды, а обработка 100 000 запросов на получение кратчайшего пути занимает 120 миллисекунд, при этом используется всего 600 Кбайт памяти.

Аппроксимация

В этом разделе рассматриваются два алгоритма, которые дают хоть и приближенное, но достаточно точное решение задачи. Алгоритм, который при поиске максимального значения некоторой функции $f(x)$ возвращает результат, являющийся произведением фактического значения (найти которое может оказаться сложнейшей задачей) на некоторый коэффициент c, называется алгоритмом с-аппроксимации (c-approximation algorithm).

Аппроксимацию особенно удобно использовать для решения *NP*-полных задач, для которых отсутствуют известные полиномиальные

алгоритмы решения. В некоторых случаях аппроксимация используется, чтобы ускорить решение задачи, когда некоторая неточность вполне допустима, а для вычисления точного решения требуется значительное время. Например, алгоритм аппроксимации со сложностью $O(\log n)$ может оказаться гораздо предпочтительнее при большом количестве исходных данных, чем точный алгоритм, имеющий сложность $O(n^3)$.

Задача коммивояжера

Чтобы выполнить формальный анализ, необходимо формализовать *задачу коммивояжера*, упоминавшуюся выше. Пусть имеется граф с весовой функцией w, которая присваивает ребрам графа положительные значения – роль такой функции, например, может играть расстояние между городами. Весовая функция удовлетворяет правилу неравенства треугольника, которое остается справедливым, если считать, что пути между городами лежат на Эвклидовой поверхности:

Для всех вершин x, y, z $\qquad w(x, y) + w(y, z) \geq w(x, z)$

Задача состоит в том, чтобы посетить каждую вершину в графе (каждый город на карте коммивояжера) только один раз и вернуться в начальную вершину (штаб-квартиру компании), чтобы сумма весов ребер, составляющих маршрут движения, была минимальной. Обозначим эту минимальную сумму как $wOPT$. (Как мы уже знаем, точное решение задачи является *NP*-полным.)

Алгоритм аппроксимации действует, как описывается далее. Сначала для графа строится минимальное связывающее дерево (Minimal Spanning Tree, MST) с общим весом $wMST$. (Минимальное связывающее дерево – это подграф, включающий все вершины, не образующий циклы и имеющий минимальный общий вес ребер из всех возможных деревьев.)

Можно смело утверждать, что $wMST \leq wOPT$, потому что $wOPT$ – это общий вес циклического пути обхода всех вершин. Удаление любого ребра из такого графа порождает связывающее дерево, а $wMST$ – это общий вес *минимального* связывающего дерева. Вооружившись этим наблюдением, мы приходим к 2-аппроксимации для $wOPT$, как описывается ниже.

1. Создание MST. Существует известный жадный алгоритм решения этой задачи со сложностью $O(n \log n)$.
2. Обход MST из корня с посещением каждого узла и возврат в корень. Общий вес этого пути составляет $2wMST \leq 2wOPT$.

3. Корректировка получившегося пути так, чтобы никакая вершина не посещалась более одного раза. Если возникнет ситуация, представленная на рис. 9.4 – вершина y посещается более одного раза – тогда путь корректируется удалением ребер (x, y) и (y, z) и заменой их ребром (x, z). Согласно правилу треугольника, такая замена может только уменьшить общий вес пути.

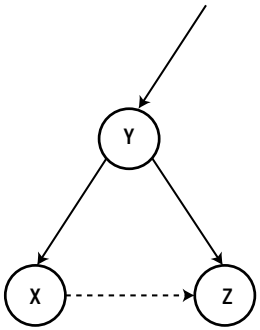

Рис. 9.4. Чтобы избежать повторного посещения вершины Y, можно заменить путь (X, Y, Z) путем (X, Z).

В результате получился алгоритм 2-аппроксимации, потому что общий вес найденного пути превосходит вес оптимального пути не более чем в два раза.

Задача о максимальном разрезе

Пусть имеется некоторый граф и его нужно *разрезать* – сгруппировать вершины в два непересекающихся множества – так, чтобы количество ребер между множествами было *максимальным*. Эта задача известна, как *задача о максимальном разрезе*, а знание алгоритма ее решения может пригодиться для решения самых разных инженерных задач.

Мы выработали очень простой и понятный алгоритм 2-аппроксимации:

1. Разделить вершины на два произвольных непересекающихся множества, A и B.
2. Найти вершину v в A, имеющую больше связанных с ней вершин в A, чем в B. Если такая вершина не найдена, поиск максимального разреза завершается.
3. Перенести вершину v из A в B и перейти к шагу 2.

Пусть A – подмножество вершин и v – вершина в множестве A. Обозначим через $\deg_A(v)$ количество вершин в A, с которыми вершина v связана ребрами (то есть, количество ее соседей в множестве A). Имеется два подмножества, A и B. Обозначим как $e(A, B)$ количество ребер между вершинами в разных подмножествах, и как $e(A)$ – количество ребер между вершинами в множестве A.

Когда поиск по данному алгоритму завершается, для каждой вершины v в A будет выполняться соотношение $\deg_B(v) \geq \deg_A(v)$? иначе алгоритм перейдет к шагу 2. Суммируя все вершины, получаем: $e(A, B) \geq \deg_B(v1) + ... + \deg_B(vk) \geq \deg_A(v1) + ... + \deg_A(vk) \geq 2e(A)$, потому что каждое ребро справа было подсчитано дважды. Аналогично, $e(A, B) \geq 2e(B)$ и, соответственно, $2e(A, B) \geq 2e(A) + 2e(B)$. Отсюда получаем $2e(A, B) \geq e(A, B) + e(A) + e(B)$, но справа находится общее количество ребер в графе. Таким образом, количество ребер, пересекающих границу двух подмножеств, будет не меньше половины всех ребер в графе. Количество ребер, пересекающих границу, не может быть больше общего количества ребер в графе, поэтому мы получили 2-аппроксимацию.

В заключение следует отметить, что алгоритм выполняет количество шагов, прямо пропорциональное количеству ребер в графе. Всякий раз, когда повторяется шаг 2, количество ребер, пересекающих границу, увеличивается как минимум на 1. А так как количество ребер, пересекающих границу, ограничено общим количеством ребер, общее количество шагов также ограничено этим числом.

Вероятностные алгоритмы

При знакомстве с алгоритмами аппроксимации мы все еще были связаны требованием предоставить детерминированное (определенное) решение. Однако в некоторых случаях введение в алгоритм некоторого элемента случайности может помочь в получении вероятного результата, правда при этом уже не приходится говорить о корректности алгоритма или об ограниченности времени его выполнения.

Вероятностное решение задачи о максимальном разрезе

Как оказывается, результат, который дает алгоритм 2-аппроксимации решения задачи о максимальном разрезе, можно получить, случайно выбирая два непересекающихся подмножества (например, для каж-

дой вершины можно подбрасывать монету, чтобы решить, к какому подмножеству ее отнести, A или B). Согласно вероятностному анализу, ожидаемое количество ребер, пересекающих границу подмножеств, будет равно половине общего количества ребер.

Чтобы показать, что ожидаемое количество ребер, пересекающих границу подмножеств, будет равно половине общего количества ребер, рассмотрим вероятность пересечения границы для конкретного ребра (u, v). Существует четыре равновероятные альтернативы (с вероятностью 0,25 каждая): ребро находится в множестве A; ребро находится в множестве B; v находится в множестве A, а u – в множестве B; и v находится в множестве B, а u – в множестве A. То есть, вероятность, что ребро пересекает границу подмножеств равна 0,5.

Для ребра e ожидаемое значение индикаторной переменной Xe (которое равно 1, если ребро пересекает границу) равно 0,5. Согласно линейности ожидания, ожидаемое количество ребер, пересекающих границу, равно половине общего количества ребер в графе.

Обратите внимание, что мы больше не можем доверять результатам единственной попытки, однако существуют приемы уменьшения случайности (такие как метод условных вероятностей), способные увеличить вероятность успеха выполнением некоторого постоянного количества попыток. Мы должны были бы доказать маловероятность того, что количество ребер, пересекающих границу, окажется меньше половины общего количества ребер в графе – сделать это можно было бы с помощью нескольких вероятностных инструментов, включая неравенство Маркова. Но мы не будем делать это здесь.

Тест простоты Ферма

Поиск простых чисел в некотором диапазоне мы уже рассматривали в главе 6, но мы не отвлекались на поиск более удачного алгоритма проверки простоты единственного числа. Эта операция играет важную роль в прикладной криптографии. Например, асимметричный алгоритм шифрования RSA, широко используемый в Интернете, опирается на поиск больших простых чисел для создания ключей шифрования.

В теории чисел существует простое следствие, известное как *малая теорема Ферма*, согласно которому, если p является простым числом, тогда для всех чисел $1 \leq a \leq p$, число a^{p-1} делится на p с остатком 1 (обозначается как $a^{p-1} \equiv 1 \pmod{p}$). Мы можем воспользоваться этой идеей и на ее основе реализовать вероятностную проверку простоты для произвольного числа n:

1. Выбрать случайное число в интервале [1, n] и посмотреть, соответствует ли оно малой теореме Ферма (то есть, получается ли остаток 1 при делении a^{p-1} на p).
2. Отвергнуть число как составное, если равенство не выполняется.
3. Принять число как простое или повторять шаг 1, пока не будет достигнут желаемый уровень доверия.

Для большинства составных чисел этому алгоритму вполне достаточно небольшого количества итераций, чтобы признать их составными и отвергнуть. Разумеется, любое простое число пройдет любое количество проверок.

К сожалению, существует бесконечное количество чисел (называются числами Кармайкла), не являющихся простыми, но способными пройти любое количество проверок для любого значения a. Хотя числа Кармайкла довольно редки, их наличие все же является веской причиной дополнить тест простоты Ферма дополнительными проверками, выявляющими числа Кармайкла. Одним из примеров может служить тест Миллера-Рабина (Miller-Rabin).

Для составных чисел, не являющихся числами Кармайкла, вероятность выбора числа, для которого равенство не выполняется, больше 0,5. То есть, вероятность принятия составного числа за простое уменьшается экспоненциально с ростом количества итераций: при достаточном количестве итераций можно уменьшить вероятность ошибки до вполне приемлемого уровня.

Индексирование и сжатие

При хранении большого объема данных, например информации об индексировании веб-страниц в поисковой системе, сжатие данных с целью экономии дискового пространства часто оказывается важнее, чем сложность выполнения. В этом разделе рассматриваются два простых примера уменьшения объема хранилища данных определенного типа, обеспечивающих при этом высокую скорость доступа.

Кодировка переменной длины

Пусть имеется коллекция 50 000 000 положительных целых чисел, которую требуется сохранить на диске, при этом мы можем гарантировать, что все значения в коллекции смогут уместиться в 32-разрядной переменной типа `int`. В простейшем случае можно просто сохранить

50 000 000 32-разрядных целых чисел на диске, заняв 200 000 000 байт. Но нам нужна более эффективная альтернатива, которая позволила бы сэкономить дисковое пространство. (Одной из причин применения сжатия может быть более высокая скорость загрузки данных в память.)

Кодировка переменной длины (variable length encoding) – это методика сжатия, пригодная для применения к последовательностям чисел, в которых присутствуют маленькие значения. Прежде чем приступать к знакомству с ней, необходимо убедиться в наличии большого количества маленьких чисел в коллекции, чего, похоже, в настоящий момент не наблюдается. Если 50 000 000 целых чисел равномерно распределены в диапазоне $[0, 2^{32}]$, тогда более 99% из них не уместятся в 3 байта и для их хранения потребуется все 4 байта. Однако, перед сохранением на диск числа можно отсортировать и вместо самих чисел сохранять на диске только пустые промежутки. Этот трюк так и называется – *сжатие промежутков* (gap compression). Он наверняка позволит уменьшить числа в сохраняемой последовательности и восстановить первоначальную последовательность.

Например, последовательность (38, 14, 77, 5, 90) сначала сортируется и получается последовательность (5, 14, 38, 77, 90), а затем кодируется с использованием приема сжатия промежутков в последовательность (5, 9, 24, 39, 13). Обратите внимание, что числа в результате получаются намного меньше и среднее количество битов, необходимое для их хранения, уменьшается значительно. В нашем случае, когда 50 000 000 целых чисел равномерно распределены в диапазоне $[0, 2^{32}]$, числа, описывающие большинство промежутков, наверняка будут укладываться в один байт.

Теперь рассмотрим основу приема кодировки переменной длины, который является лишь одним из множества приемов сжатия данных, известных в теории информации. Идея этого приема состоит в том, чтобы использовать наибольший значащий бит в каждом байте, чтобы показать, является ли этот байт последним в представлении целого числа или нет. Если бит сброшен, производится переход к следующему байту и восстановление числа продолжается; если бит установлен, чтение последовательности байтов останавливается и выполняется декодирование числа из прочитанной последовательности.

Например, число 13 кодируется как `10001101` – старший бит установлен, поэтому данный байт содержит полное целое число и остальная его часть является простым двоичным представлением числа 13. Далее, число 132 кодируется как `00000001`10000100`. В первом байте

старший бит сброшен, поэтому остальные семь бит `0000001` сохраняются, в следующем байте старший бит установлен, поэтому оставшиеся семь бит добавляются к предыдущим семи битам и получается значение `10000100`, являющееся двоичным представлением числа 132. В этом примере одно число было сохранено в одном байте, а другое – в двух байтах. Сохранение промежутков, полученных на предыдущем шаге, с использованием данного приема позволит сжать исходные данные почти в четыре раза. (Вы можете попробовать поэкспериментировать со случайными числами и подтвердить этот результат.)

Сжатие индексов

Чтобы обеспечить эффективное хранение индексов слов, встречающихся в веб-страницах – что является основой любого поискового механизма – для каждого слова необходимо сохранить номера страниц (или адреса URL), в которых они встречаются, и сжать данные, обеспечив эффективный доступ к ним. В типичных поисковых системах в оперативной памяти хранятся только словари из слов, но не номера страниц.

Сохранение на диске номеров страниц лучше выполнять с применением кодировки переменной длины, с которой мы только что познакомились. Но сохранение самого словаря является гораздо более сложной задачей. В идеале, словарь – это простой массив, содержащий встреченные слова, и смещение в дисковом файле, где хранятся номера страниц, где встречается данное слово. Чтобы повысить эффективность доступа к этим данным, они должны быть отсортированы – это гарантирует время доступа к ним $O(\log n)$.

Допустим, что каждая запись в словаре, хранящемся в памяти, является значением следующего типа значения на языке C#, а сам словарь является массивом этих записей:

```
struct DictionaryEntry {
    public string Word;
    public ulong DiskOffset;
}
DictionaryEntry[] dictionary = ...;
```

Как было показано в главе 3, массивы типов значений состоят исключительно из экземпляров этих типов. Однако каждый экземпляр содержит ссылку на строку; для n записей эти ссылки, наряду со смещениями в дисковом файле, будут занимать $16n$ байт в 64-разрядной системе. Мало того, что сам словарь слов будет занимать немалый объем ценной памяти, так еще и каждое слово в словаре будет хра-

ниться как отдельная строка, что потребует дополнительно 24 байта (16 байт на заголовок объекта + 4 байта на длину строки + 4 байта для хранения количества символов во внутреннем буфере, выделяемом для строки).

Мы можем значительно уменьшить объем памяти, занимаемой словарем, объединив все слова в одну большую строку и сохранив их смещения в этой строке в структуре `DictionaryEntry` (см. рис. 9.5). Размеры таких объединенных строк редко будет превышать 2^{24} байт = 16 Мбайт, откуда поле индекса может быть выражено 3-байтным целым числом, вместо 8-байтного адреса в памяти:

```
[StructLayout(LayoutKind.Sequential, Pack = 1, Size = 3)]
struct ThreeByteInteger {
    private byte a, b, c;
    public ThreeByteInteger() {}
    public ThreeByteInteger(uint integer) ...
    public static implicit operator int(ThreeByteInteger tbi) ...
}
struct DictionaryEntry {
    public ThreeByteInteger LongStringOffset;
    public ulong DiskOffset;
}
class Dictionary {
    public DictionaryEntry[] Entries = ...;
    public string LongString;
}
```

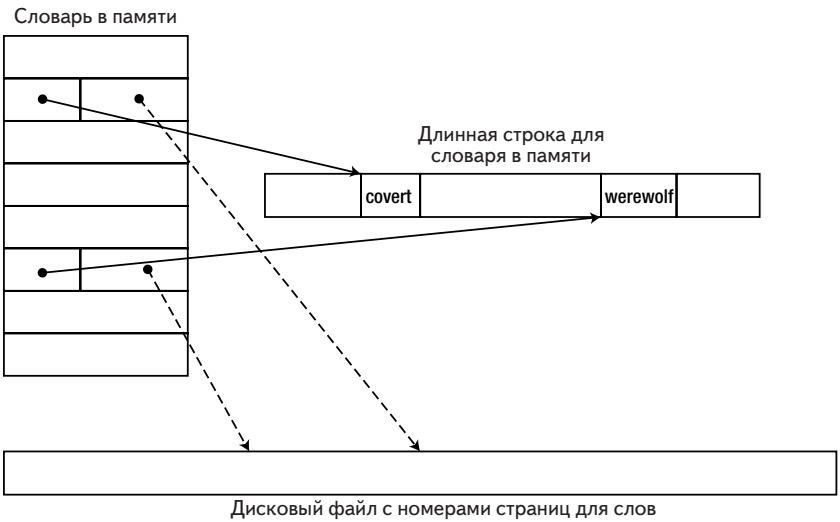

Рис. 9.5. Общая структура словаря в памяти и каталога в строке.

В результате мы получаем возможность организации поиска в массиве методом половинного деления – потому что все записи имеют одинаковый размер – а объем занимаемой памяти оказывается намного меньше. Мы сэкономили 24*n* байт на строковых объектах (так как теперь все слова хранятся в одной большой строке) и еще 5*n* байт на ссылках на строки, заменив указатели смещениями.

В заключение

В этой главе мы исследовали некоторые положения теоретической информатики, включая анализ сложности алгоритмов, их проектирование и оптимизацию. Как вы могли видеть, оптимизация алгоритмов не является чем-то, доступным только посвященным; в реальной жизни встречается множество ситуаций, когда выбор правильного алгоритма или использование приемов сжатия может существенно повысить производительность. В частности, разделы, описывающие приемы динамического программирования, хранения индексов и алгоритмов аппроксимации, содержат рецепты, которые вы сможете адаптировать для своих приложений.

Примеры в данной главе, это даже не вершина айсберга теории сложности и науки исследования алгоритмов. В первую очередь мы надеялись, что прочитав эту главу, вы познакомитесь с некоторыми идеями, лежащими в основе теоретической информатики и практических алгоритмов, используемых в действующих приложениях. Мы знаем, что многие разработчики приложений для .NET редко сталкиваются с необходимостью придумывать совершенно новые алгоритмы, но мы верим, что для любого программиста важно понимать и иметь в запасе примеры использования некоторых алгоритмов.

ГЛАВА 10. Шаблоны оптимизации производительности

В этой главе рассматриваются темы, которые не были затронуты выше. Несмотря на то, что эти темы очень короткие, они играют очень важную роль в создании высокопроизводительных приложений. Обсуждаемые здесь темы не связаны какой-то единой нитью и объединены только нашим желанием дать рекомендации по достижению наивысшей производительности в простых глубоких циклах и сложных приложениях.

Мы начнем эту главу знакомством с оптимизациями JIT-компилятора, которые совершенно необходимы для достижения высокой производительности преимущественно вычислительных приложений. Затем мы обсудим вопросы производительности при запуске приложений, важных для клиентских приложений, так как длительный запуск испытывает терпение пользователей. В заключение мы рассмотрим оптимизации, характерные для процессоров, включая распараллеливание данных и инструкций, а также ряд более мелких тем.

Оптимизации JIT-компилятора

Выше в этой книге нам уже довелось убедиться, насколько важны оптимизации, выполняемые JIT-компилятором. В частности, в главе 3 мы довольно подробно рассмотрели встраивание методов, когда исследовали последовательности инструкций вызовов виртуальных и не виртуальных методов. В этом разделе мы познакомимся с основными оптимизациями, выполняемыми JIT-компилятором, и как писать код, чтобы не мешать JIT-компилятору применять их. Оптимизации JIT-компилятора в основном предназначены для повышения производительности вычислительных приложений, но они оказывают положительное влияние и другие типы приложений.

Для исследования описываемых здесь оптимизаций, отладчик необходимо подключать к уже работающему процессу – JIT-компилятор не выполняет оптимизацию, если обнаруживает, что программа выполняется под отладчиком. В частности, подключаясь к процессу, необходимо убедиться, что исследуемые методы уже вызывались и были скомпилированы.

Если по каким-то причинам вам потребуется запретить выполнение оптимизаций JIT-компилятором, например, чтобы не сталкиваться при отладке со встроенными методами или хвостовыми вызовами (обсуждаются ниже), необязательно изменять код или использовать режим сборки **Debug**. Достаточно просто создать *.ini*-файл с тем же именем, что и выполняемый файл приложения (например, *MyApp.ini*) и добавить в него следующие три строки.

```
[.NET Framework Debugging Control]
GenerateTrackingInfo = 1
AllowOptimize = 0
```

Если поместить его в каталог с выполняемым файлом, при следующем запуске приложения JIT-компилятор обнаружит его и не будет выполнять какие-либо оптимизации.

Стандартные оптимизации

Практически все оптимизирующие компиляторы, даже самые простые, выполняют некоторые стандартные оптимизации. Например, JIT-компилятор способен сократить следующий код на C# до нескольких машинных инструкций x86:

```
// Оригинальный код на C#:
static int Add(int i, int j) {
    return i + j;
}
static void Main() {
    int i = 4;
    int j = 3*i + 11;
    Console.WriteLine(Add(i, j));
}

; Оптимизированный код на языке ассемблера
call 682789a0                ; System.Console.get_Out()
mov ecx,eax
mov edx,1Bh                  ; вызов Add(i,j) замещен
                             ; результатом, 27 (0x1B)
mov eax,dword ptr [ecx]      ; далее следует стандартная
                             ; последовательность
```

```
mov eax,dword ptr [eax + 38h] ; вызова виртуального метода
                              ; TextWriter.WriteLine
call dword ptr [eax + 14h]
```

> **Примечание.** *Эта оптимизация выполняется не компилятором C#. Если заглянуть в сгенерированный им код на языке IL, в нем будут присутствовать и локальные переменные, и вызов метода* `Add`. *Любые оптимизации выполняются JIT-компилятором.*

Эта оптимизация называется *сверткой констант* (constant folding), и существует еще множество подобных простых оптимизаций, таких как *свертка общих подвыражений* (common subexpression reduction) , например в выражениях, таких как `a + (b * a) - (a * b * c)`, значение `a * b` достаточно вычислить только один раз. JIT-компилятор способен выполнять такие стандартные оптимизации, но нередко справляется с этим значительно хуже, в сравнении с другими оптимизирующими компиляторами, такими как компилятор Microsoft Visual C++. Причина такого отставания в том, что JIT-компилятор имеет очень ограниченную среду выполнения и должен компилировать методы очень быстро, чтобы избежать значительных задержек при первом обращении к ним.

Встраивание методов

Эти оптимизации часто уменьшают объем кода и практически всегда уменьшают время выполнения, замещая последовательность инструкций вызова метода его телом. Как было показано в главе 3, виртуальные методы не встраиваются JIT-компилятором (даже при вызове конечного (sealed) метода у экземпляра производного типа); методы интерфейсов подвергаются частичному встраиванию; только статические и не виртуальные могут встраиваться всегда. Там где важна высокая производительность, например, в простых свойствах и методах часто используемых базовых классов, желательно избегать виртуальных методов и реализации интерфейсов.

Точные критерии, используемые JIT-компилятором для определения, какие методы могут встраиваться, недоступны. Однако экспериментальным путем нам удалось вскрыть некоторые из них:

- методы со сложной структурой вызовов (например, циклы) никогда не встраиваются;
- методы, включающие обработку исключений, никогда не встраиваются;
- рекурсивные методы никогда не встраиваются;

- методы, имеющие параметры составных типов значений, локальные переменные или возвращаемые значения никогда не встраиваются;
- методы, размеры тел которых превышают 32 байта на языке IL никогда не встраиваются (это ограничение можно преодолеть с помощью значения `MethodImplOptions.AggressiveInlining` атрибута `[MethodImpl]`.

В последних версиях среды выполнения CLR были убраны некоторые искусственные ограничения, препятствующие встраиванию методов. Например, начиная с версии .NET 3.5 SP1, 32-разрядный JIT-компилятор способен встраивать методы, принимающие параметры *некоторых* составных типов значений, таких как тип `Point2D`, который описывался в главе 3. В этой версии некоторые операции с составными типами значениями замещаются эквивалентными операциями с простыми типами, при определенных условиях (операции с экземпляром типа `Point2D` преобразуются в операции с двумя значениями типа `int`), что обеспечивает лучшую оптимизацию кода, выполняющего операции со структурами в целом,. Например, взгляните на следующий простой код:

```
private static void MethodThatTakesAPoint(Point2D pt) {
    pt.Y = pt.X ^ pt.Y;
    Console.WriteLine(pt.Y);
}

Point2D pt;
pt.X = 3;
pt.Y = 5;
MethodThatTakesAPoint(pt);
```

JIT-компилятор в среде выполнения CLR 4.5 весь этот фрагмент кода скомпилирует в функциональный эквивалент `Console.WriteLine(6)`, где аргумент 6 является результатом выражения `3 ^ 5`. JIT-компилятор способен выполнять встраивание и распространение констант пользовательских типов значений. В версии CLR 2.0 JIT-компилятор фактически вызывает метод без какой-либо видимой оптимизации:

```
; вызывающий код
mov  eax,3
lea  edx,[eax + 2]
push edx
push eax
call dword ptr ds:[1F3350h] (Program.MethodThatTakesAPoint(Point2D),
mdToken: 06000003)
```

```
; код метода
push ebp
mov  ebp,esp
mov  eax,dword ptr [ebp + 8]
xor  dword ptr [ebp + 0Ch],eax
call mscorlib_ni + 0x22d400 (715ed400) (System.Console.get_Out(),
mdToken: 06000773)
mov  ecx,eax
mov  edx,dword ptr [ebp + 0Ch]
mov  eax,dword ptr [ecx]
call dword ptr [eax + 0BCh]
pop  ebp
ret  8
```

Несмотря на отсутствие возможности обеспечить *принудительное* встраивание методов, когда JIT-компилятор не считает это необходимым, у нас есть механизм, позволяющий запретить встраивание. Значение `MethodImplOptions.NoInlining` атрибута `[MethodImpl]` запрещает встраивание метода, снабженного таким атрибутом – кстати говоря, это весьма полезная возможность для микрохронометража, как обсуждалось в главе 2.

Отключение проверки границ

Когда осуществляется обращение к элементам массивов, среда выполнения CLR должна гарантировать, что индекс, используемый для доступа к элементам, не окажется за пределами массива. В отсутствие такой проверки снимаются гарантии безопасности доступа к памяти; вы могли бы инициализировать объект `byte[]` и обращаться с его помощью к произвольным участкам памяти по положительным и отрицательным индексам. Несмотря на абсолютную необходимость, эта проверка имеет накладные расходы, стоимостью в несколько инструкций. Ниже показан код, сгенерированный JIT-компилятором для типичной операции доступа к массиву:

```
// Оригинальный код на C#:
uint[] array = new uint[100];
array[4] = 0xBADC0FFE;

; Сгенерированный код на языке ассемблера x86
mov  ecx,offset 67fa33aa   ; тип элемента массива
mov  edx,64h               ; размер массива
call 0036215c              ; создать новый массив (CORINFO_HELP_NEWARR_1_VC)
cmp  dword ptr [eax + 4],4 ; eax + 4 - длина массива, 4 - индекс
jbe  NOT_IN_RANGE          ; Если длина меньше или равна индексу, перейти
mov  dword ptr [eax + 18h],0BADC0FFEh ; смещение вычисляется на этапе
```

```
                                        ; JIT-компиляции (0x18 = 8 + 4*4)

; остальная часть программы, переход через метку NOT_IN_RANGE
NOT_IN_RANGE:
call clr!JIT_RngChkFail    ; возбудить исключение
```

Существуют особые ситуации, когда JIT-компилятор может отключить проверку границ при обращении к элементам массива – в цикле `for`, выполняющем обход всех элементов. Без этой оптимизации операции доступа к массивам всегда были бы медленнее, чем в неуправляемом коде, что является неприемлемой потерей производительности в приложениях, осуществляющих сложные расчеты и интенсивно работающих с памятью. Для следующего цикла JIT-компилятор отключит проверку границ:

```
// Оригинальный код на C#:
for (int k = 0; k < array.Length; ++k) {
    array[k] = (uint)k;
}

; Сгенерированный код на языке ассемблера x86 (оптимизированный)
    xor  edx,edx                        ; edx = k = 0
    mov  eax,dword ptr [esi + 4]        ; esi = array, eax = array.Length
    test eax,eax                        ; если массив пуст,
    jle  END_LOOP                       ; пропустить цикл
NEXT_ITERATION:
    mov  dword ptr [esi + edx*4 + 8],edx ; array[k] = k
    inc  edx                            ; ++k
    cmp  eax,edx                        ; пока array.Length > k,
    jg  NEXT_ITERATION                  ; перейтик следующей итерации
END_LOOP:
```

В этом цикле выполняется единственная проверка – проверка условия выхода из цикла. Обратите внимание, что проверка доступа к элементам массива внутри цикла *не* выполняется – выделенная строка выполняет запись в k-й элемент массива без проверки выхода индекса k за пределы массива.

К сожалению, эта оптимизация очень чувствительна. Некоторые, невинные на первый взгляд изменения в цикле, могут отрицательно сказаться на этой оптимизации и вынудить компилятор добавить проверку границ массивов:

```
// Проверка границ отсутствует
for (int k = 0; k < array.Length - 1; ++k) {
    array[k] = (uint)k;
}

// Проверка границ отсутствует
```

```
for (int k = 7; k < array.Length; ++k) {
    array[k] = (uint)k;
}

// Проверка границ отсутствует
// JIT-компилятор удалит -1 из проверки границ и начнет со второго элемента
for (int k = 0; k < array.Length - 1; ++k) {
    array[k + 1] = (uint)k;
}

// Проверка границ выполняется
for (int k = 0; k < array.Length / 2; ++k) {
    array[k * 2] = (uint)k;
}

// Проверка границ выполняется
staticArray = array; // "staticArray" - это статическое поле
                     // вмещающего класса
for (int k = 0; k < staticArray.Length; ++k) {
    staticArray[k] = (uint)k;
}
```

Итак, отключение проверки границ – довольно чувствительная оптимизация, но она стоит того, чтобы убедиться в ее применении в критических ко времени выполнения участках кода, даже если для этого придется исследовать ассемблерный код. Дополнительную информацию об отключении проверки границ и некоторых особых случаях можно найти в статье «Array Bounds Check Elimination in the CLR» Дейва Детлефса (Dave Detlefs), по адресу: http://blogs.msdn.com/b/clrcodegeneration/archive/2009/08/13/array-bounds-check-elimination-in-the-clr.aspx.

Хвостовые вызовы

Хвостовой вызов (tail calling) – это оптимизация, обеспечивающая повторное использование кадра стека существующего метода для вызова другого метода. Эта оптимизация будет весьма полезна во многих рекурсивных алгоритмах. Фактически, благодаря ей некоторые рекурсивные методы могут показывать столь же высокую производительность, как и аналогичные им реализации на основе циклов. Взгляните на следующий рекурсивный метод, вычисляющий наибольший общий делитель двух целых чисел:

```
public static int GCD(int a, int b) {
    if (b == 0) return a;
    return GCD(b, a % b);
}
```

Очевидно, что к рекурсивному вызову `GCD(b, a % b)` не может быть применена оптимизация встраивания тела метода – в конце концов, это рекурсивный вызов. Однако, так как кадры стеков вызывающего и вызываемого методов абсолютно совместимы, и вызывающий метод не выполняет никаких операций после рекурсивного вызова, данный метод можно оптимизировать и переписать иначе:

```
public static int GCD(int a, int b) {
START:
    if (b == 0) return a;
    int temp = a % b;
    a = b;
    b = temp;
    goto START;
}
```

Благодаря оптимизации из реализации исчезли рекурсивные вызовы – фактически рекурсивный алгоритм был преобразован в циклический. Эту оптимизацию можно выполнять вручную, когда это возможно, однако при определенных условиях JIT-компилятор способен применять ее автоматически. Ниже представлены две версии метода `GCD` – первая скомпилирована 32-разрядным JIT-компилятором из CLR 4.5, а вторая 64-разрядным JIT-компилятором из CLR 4.5:

```
; 32-разрядная версия, параметры в ECX и EDX
push ebp
mov  ebp,esp
push esi
mov  eax,ecx       ; EAX = a
mov  ecx,edx       ; ECX = b
test ecx,ecx       ; if b == 0, returning a
jne  PROCEED
pop  esi
pop  ebp
ret
PROCEED:
cdq
idiv eax,ecx       ; EAX = a / b, EDX = a % b
mov  esi,edx
test esi,esi       ; if a % b == 0, вернуть b (базовый случай рекурсии)
jne  PROCEED2
mov  eax,ecx
jmp  EXIT
PROCEED2:
mov  eax,ecx
cdq
idiv eax,esi
mov  ecx,esi       ; рекурсивный вызов в следующей строке
```

```
call dword ptr ds:[3237A0h] (Program.GCD(Int32, Int32), mdToken: 06000004)
EXIT:
pop esi
pop ebp
ret                 ; повторное использование возвращаемого
                    ; значения (в EAX) из рекурсивного вызова

; 64-разрядная версия, параметры в ECX and EDX
sub  rsp,28h        ; создание кадра стека – производится только один раз!
START:
mov  r8d,edx
test r8d,r8d        ; if b == 0, return a
jne  PROCEED
mov  eax,ecx
jmp  EXIT
PROCEED:
cmp  ecx,80000000h
jne  PROCEED2:
cmp  r8d,0FFFFFFFFh
je   OVERFLOW       ; различные проверки на переполнение
xchg ax,ax          ; два байта NOP (0x66 0x90) для выравнивания
PROCEED2:
mov  eax,ecx
cdq
idiv eax,r8d        ; EAX = a / b, EDX = a % b
mov  ecx,r8d        ; повторно инициализировать параметры
mov  r8d,edx        ; ...
jmp  START          ; и перейти в начало (без вызова функции)
xchg ax,ax          ; два байта NOP (0x66 0x90) для выравнивания
EXIT:
add  rsp,28h
ret
OVERFLOW:
call clr!JIT_Overflow
nop
```

Совершенно очевидно, что 64-разрядный JIT-компилятор использует хвостовую оптимизацию, чтобы избавиться от рекурсивных вызовов метода, а 32-разрядный – нет. Детальное исследование условий, при которых два JIT-компилятора применяют оптимизацию хвостовых вызовов, далеко выходит за рамки этой книги, поэтому приведем лишь краткий список некоторых эвристик:

- 64-разрядный JIT-компилятор более свободен в выборе оптимизации хвостовых вызовов и часто применяет ее, даже когда компилятор языка (например, компилятор C#) не предлагает использовать ее добавлением префикса `tail.` в коде на языке IL;

  - когда за вызовом следует дополнительный код (кроме инструкции возврата из метода), что препятствует оптимизации хвостовых вызовов (это ограничение несколько ослаблено в CLR 4.0);
  - когда вызываемые методы возвращают значения, типы которых отличаются от типа значения, возвращаемого вызывающим методом;
  - когда вызываемые методы имеют слишком много параметров, параметры имеют разное выравнивание или типы параметров/возвращаемого значения оказываются слишком большими типами значений (значительно ослаблено в CLR 4.0).
- 32-разрядный JIT-компилятор менее склонен к этой оптимизации и выполняет ее, только при наличии префикса `tail.` в коде на языке IL.

Примечание. *Любопытный эффект возникает, когда оптимизация хвостовых вызовов применяется к методам, образующим бесконечную рекурсию. Если вы допустили ошибку в определении базового случая прекращения рекурсии, которая может привести к бесконечной рекурсии, а JIT-компилятор смог превратить рекурсивный вызов метода в хвостовой вызов, вместо обычного в таких случаях исключения* `StackOverflowException` *вы получите бесконечный цикл!*

Более подробную информацию о префиксе `tail.` языка IL, используемого, чтобы подсказать JIT-компиляторам о возможности выполнить оптимизацию хвостового вызова, а также о критериях, используемых JIT-компилятором для оптимизации хвостовых вызовов, можно найти в Интернете:

- префикс `tail.`, который не используется компилятором C#, но часто используется компиляторами функциональных языков программирования (включая F#) описывается на сайте MSDN, в разделе с описанием класса `System.Reflection.Emit`: http://msdn.microsoft.com/ru-ru/library/system.reflection.emit.opcodes.tailcall.aspx;
- список условий, когда JIT-компилятор применяет оптимизацию хвостовых вызовов (в версиях CLR ниже 4.0), можно найти в статье Дэвида Бромана (David Broman) «Tail call JIT conditions», по адресу: http://blogs.msdn.com/b/davbr/archive/2007/06/20/tail-call-jit-conditions.aspx;
- в CLR 4.0 произошли изменения в особенностях применения оптимизации хвостовых вызовов JIT-компилятором, которые

подробно описаны в статье «Tail Call Improvements in .NET Framework 4», по адресу: http://blogs.msdn.com/b/clrcodegeneration/archive/2009/05/11/tail-call-improvements-in-net-framework-4.aspx.

Производительность на этапе запуска

Быстрый запуск клиентского приложения позволяет вызвать первые положительные впечатления у пользователя или потенциального клиента, опробующего демонстрационную версию продукта. Однако, чем сложнее приложение, тем труднее обеспечить быстрый его запуск. Очень важно отличать *холодный запуск* (cold startup), когда приложение запускается впервые после загрузки системы, и *теплый запуск* (warm startup) , когда приложение запускается (не в первый раз) после того, как система уже поработала некоторое время. Системные службы и фоновые агенты должны иметь очень короткое время холодного запуска, чтобы пользователю не пришлось слишком долго ждать возможности войти в систему. Типичные клиентские приложения, такие как веб-браузеры и клиенты электронной почты могут иметь большее время холодного запуска, но пользователи всегда ожидают, что спустя некоторое время после загрузки системы приложения будут запускаться немного быстрее. Большинство пользователей желало бы уменьшить время запуска в обоих случаях.

Существует несколько факторов, объясняющих длительное время запуска. Некоторые из них применимы только к холодному запуску; другие – к обоим типам запуска.

- Операции ввода/вывода – чтобы запустить приложение, Windows и CLR должны загрузить с диска сборки, используемые приложением, а также сборки .NET Framework, библиотеки среды выполнения CLR и библиотеки Windows. Этот фактор оказывает влияние в основном на время холодный запуск.
- JIT-компиляция – каждый метод, впервые вызываемый на этапе запуска приложения, должен быть скомпилирован JIT-компилятором. Поскольку машинный код, полученный в результате JIT-компиляции, не сохраняется после завершения приложения, этот фактор влияет на время как холодного, так и теплого запуска.

- Инициализация графического интерфейса пользователя – в зависимости от используемого фреймворка графического интерфейса (Metro, WPF, Windows Forms, и так далее), существуют определенные этапы инициализации, характерные для каждого из них, которые необходимо выполнить для отображения интерфейса приложения на экране. Этот фактор влияет на время как холодного, так и теплого запуска.
- Загрузка данных для приложения – приложению может требоваться некоторая информация, получаемая из файлов, баз данных или веб-служб, для отображения на экране сразу после запуска. Этот фактор влияет на время как холодного, так и теплого запуска, если только приложение не использует какие-либо приемы кеширования данных.

В нашем распоряжении имеется несколько измерительных инструментов, позволяющих диагностировать наиболее вероятные причины длительного времени запуска (см. главу 2). Sysinternals Process Monitor поможет выявить операции ввода/вывода, выполняемые прикладным процессом, независимо от того, кто является их инициатором – Windows, CLR или прикладной код. PerfMonitor и счетчики производительности из категории **.NET CLR JIT** могут помочь выявить чрезмерные затраты времени на JIT-компиляцию на этапе запуска приложения. Наконец, «стандартные» профилировщики (в дискретном или инструментированном режиме) способны помочь выявить участки кода, на выполнение которых тратится большая часть времени на этапе запуска.

Кроме того, с помощью несложного эксперимента можно легко определить, действительно ли узким местом являются ли операции ввода/вывода, выполняемые при запуске приложения: измерьте время холодного и теплого запуска приложения (для этого эксперимента должен использоваться компьютер, свободный от других задач, чтобы ненужные службы и другие приложения не искажали временные характеристики холодного запуска). Если теплый запуск будет выполняться значительно быстрее холодного, следовательно, главной причиной длительного холодного запуска являются операции ввода/вывода.

Ответственность за ускорение загрузки данных, необходимых приложению, полностью лежит на вас; любые рекомендации в этом направлении, какие только мы сможем дать, будут слишком общими, чтобы иметь какую-либо практическую ценность (кроме экранной заставки, вывод которой положительно сказывается на долготерпении пользователей...). Однако мы можем предложить несколько средств

увеличения скорости запуска, когда причиной является большой объем ввода/вывода и JIT-компиляции. В некоторых случаях описываемые далее приемы позволяют сократить время запуска вдвое и даже больше.

Предварительная JIT-компиляция с помощью NGen (Native Image Generator)

JIT-компилятор очень удобен и компилирует методы, только когда они действительно вызываются, однако приложению приходится порой дорого платить своей производительностью за его использование. Для решения этой проблемы платформа .NET Framework предлагает инструмент оптимизации с названием *Native Image Generator* (генератор низкоуровневых образов, *NGen.exe*), который может компилировать сборки в машинный код (native images – низкоуровневые образы) перед запуском. Если все сборки, используемые приложением, будут предварительно скомпилированы этим инструментом, отпадет необходимость загружать JIT-компилятор и использовать его в процессе запуска приложения. Даже при том, что сгенерированный низкоуровневый образ чаще оказывается больше оригинальной сборки, в большинстве случаев объем дискового ввода/вывода снижается, потому что отпадает необходимость загружать JIT-компилятор (*clrjit.dll*) и метаданные используемых сборок.

Предварительная компиляция дает еще один положительный эффект – низкоуровневые образы могут совместно использоваться процессами, в отличие от кода, который генерируется JIT-компилятором во время выполнения. Если несколько процессов на одном и том же компьютере будут использовать низкоуровневый образ одной и той же сборки, общее потребление физической памяти окажется ниже, чем при использовании JIT-компилятора. Это особенно важно в системах с открытым доступом, когда несколько пользователей подключаются к общему серверу через службу терминалов (Terminal Services) и запускают одно и то же приложение.

Чтобы скомпилировать приложение, достаточно просто указать инструменту *NGen.exe*, где находится главная сборка приложения (обычно *.exe* файл). Генератор *NGen* отыщет все статические зависимости основной сборки и скомпилирует их все в низкоуровневые образы. Получившиеся образы будут сохранены в кеше – по соседству с глобальным кешем сборок (GAC), в папках *C:\Windows\Assembly\NativeImages_** по умолчанию.

Совет. *Так как CLR и NGen управляют кешем низкоуровневых образов автоматически, вы не должны копировать низкоуровневые образы с одного компьютера на другой. Единственный доступный способ получить скомпилированные сборки в той или иной системе – воспользоваться инструментом NGen. Лучше всего это делать в процессе установки приложения (NGen поддерживает даже команду «defer» («отложить»), которая передаст компиляцию фоновой службе). Именно так поступает мастер установки .NET Framework в отношении часто используемых сборок .NET.*

Ниже приводится законченный пример использования инструмента NGen для предварительной компиляции простого приложения, состоящего из двух сборок – файла *main .exe* и вспомогательной библиотеки *.dll*. NGen благополучно определяет зависимость от этой библиотеки и создает низкоуровневые образы для обеих сборок:

```
> c:\windows\microsoft.net\framework\v4.0.30319\ngen install Ch10.exe

Microsoft (R) CLR Native Image Generator - Version 4.0.30319.17379
Copyright (c) Microsoft Corporation. All rights reserved.

Installing assembly D:\Code\Ch10.exe
1>    Compiling assembly D:\Code\Ch10.exe (CLR v4.0.30319) ...
2>    Compiling assembly HelperLibrary, ... (CLR v4.0.30319) ...
```

Во время выполнения среда CLR будет использовать низкоуровневые сборки, вообще не загружая библиотеку *clrjit.dll* JIT-компилятора (в выводе команды `lm`, что приводится ниже, библиотека *clrjit.dll* отсутствует). Таблицы методов типов (см. главу 3) также сохраняются в низкоуровневых образах и содержат указатели на скомпилированные версии внутри образов.

```
0:007 > lm
start    end        module name
01350000 01358000   Ch10       (deferred)
2f460000 2f466000   Ch10_ni    (deferred)
30b10000 30b16000   HelperLibrary_ni (deferred)
67fa0000 68eef000   mscorlib_ni   (deferred)
6b240000 6b8bf000   clr         (deferred)
6f250000 6f322000   MSVCR110_CLR0400 (deferred)
72190000 7220a000   mscoreei    (deferred)
72210000 7225a000   MSCOREE     (deferred)
74cb0000 74cbc000   CRYPTBASE   (deferred)
74cc0000 74d20000   SspiCli     (deferred)
74d20000 74d39000   sechost     (deferred)
74d40000 74d86000   KERNELBASE  (deferred)
74e50000 74f50000   USER32      (deferred)
74fb0000 7507c000   MSCTF       (deferred)
75080000 7512c000   msvcrt      (deferred)
```

```
75150000 751ed000  USP10      (deferred)
753e0000 75480000  ADVAPI32   (deferred)
75480000 75570000  RPCRT4     (deferred)
75570000 756cc000  ole32      (deferred)
75730000 75787000  SHLWAPI    (deferred)
75790000 757f0000  IMM32      (deferred)
76800000 7680a000  LPK        (deferred)
76810000 76920000  KERNEL32   (deferred)
76920000 769b0000  GDI32      (deferred)
775e0000 77760000  ntdll      (pdb symbols)

0:007 > !dumpmt -md 2f4642dc
EEClass:          2f4614c8
Module:           2f461000
Name:             Ch10.Program
mdToken:          02000002
File:             D:\Code\Ch10.exe
BaseSize:         0xc
ComponentSize:    0x0
Slots in VTable:  6
Number of IFaces in IFaceMap: 0
--------------------------------------
MethodDesc Table
   Entry MethodDe    JIT Name
68275450 68013524 PreJIT System.Object.ToString()
682606b0 6801352c PreJIT System.Object.Equals(System.Object)
68260270 6801354c PreJIT System.Object.GetHashCode()
68260230 68013560 PreJIT System.Object.Finalize()
2f464268 2f46151c PreJIT Ch10.Program..ctor()
2f462048 2f461508 PreJIT Ch10.Program.Main(System.String[])

0:007 > !dumpmt -md 30b141c0
EEClass:          30b114c4
Module:           30b11000
Name:             HelperLibrary.UtilityClass
mdToken:          02000002
File:             D:\Code\HelperLibrary.dll
BaseSize:         0xc
ComponentSize:    0x0
Slots in VTable:  6
Number of IFaces in IFaceMap: 0
--------------------------------------
MethodDesc Table
   Entry MethodDe    JIT Name
68275450 68013524 PreJIT System.Object.ToString()
682606b0 6801352c PreJIT System.Object.Equals(System.Object)
68260270 6801354c PreJIT System.Object.GetHashCode()
68260230 68013560 PreJIT System.Object.Finalize()
30b14158 30b11518 PreJIT HelperLibrary.UtilityClass..ctor()
30b12048 30b11504 PreJIT HelperLibrary.UtilityClass.SayHello()
```

Другой полезной особенностью является команда «update» («обновить»), которая вынуждает NGen повторно определить зависимости для всех низкоуровневых образов в кеше и перекомпилировать все изменившиеся сборки. Эту команду можно использовать после установки, для обновления целевой системы в процессе разработки.

Примечание. *Теоретически инструмент NGen мог бы применять совершенно иные оптимизации, отличные от тех, что применяются JIT-компилятором. В конце концов, NGen не имеет таких строгих ограничений по времени выполнения, как JIT-компилятор. Однако к моменту написания этих строк NGen не поддерживал никаких дополнительных оптимизаций, кроме тех, что предлагаются JIT-компилятором.*

При использовании CLR 4.5 в Windows 8, NGen не ждет пассивно, пока вы дадите команду скомпилировать прикладные сборки. Вместо этого CLR генерирует файлы журналов с информацией об использовании сборок, которые обрабатываются фоновым заданием поддержки инструмента NGen. Это задание решает, какие сборки можно скомпилировать, и запускает NGen с целью создания низкоуровневых образов для них. Вы все еще можете использовать команду «display» («показать»), чтобы получить перечень содержимого кеша низкоуровневых образов (или исследовать файлы в кеше из командной строки), но основное бремя принятия решения о предварительной компиляции теперь берет на себя среда выполнения CLR.

Фоновая JIT-компиляция в многопроцессорных системах

Начиная с версии CLR 4.5, появилась возможность настроить JIT-компилятор так, чтобы он генерировал перечень методов, выполняемых в ходе запуска приложения, и использовать его при последующих запусках (включая холодный запуск), чтобы обеспечить компиляцию этих методов в фоновом режиме. Иными словами, пока основной поток выполнения производит инициализацию приложения, JIT-компилятор в фоновых потоках выполняет компиляцию методов, которые понадобятся в самое ближайшее время. Перечень методов обновляется при каждом запуске, поэтому он будет оставаться свежим всегда, даже если приложение будет запускаться сотни раз в разных конфигурациях.

Примечание. *Эта особенность включена по умолчанию в приложениях для ASP.NET и Silverlight 5.*

Чтобы задействовать фоновую JIT-компиляцию, необходимо вызвать два метода класса `System.Runtime.ProfileOptimization`. Первый метод сообщает профилировщику – где хранить необходимую информацию, а второй определяет, какой сценарий запуска выполняется. Цель второго метода – обеспечить различение существенно разных сценариев запуска, чтобы в разных ситуациях использовались разные оптимизации запуска. Например, утилита архивирования может быть вызвана с параметром, сообщающим ей «показать список файлов в архиве», требующим выполнить один набор методов, или с параметром «создать архив для указанного каталога», требующим выполнить совершенно иной набор методов:

```
public static void Main(string[] args) {
    System.Runtime.ProfileOptimization.SetProfileRoot(
        Path.GetDirectoryName(Assembly.GetExecutingAssembly().Location));
    if (args[0] == "display") {
        System.Runtime.ProfileOptimization.StartProfile(
            "DisplayArchive.prof");
    } else if (args[0] == "compress") {
        System.Runtime.ProfileOptimization.StartProfile(
            "CompressDirectory.prof");
    }
    // ...Другие сценарии запуска
    // После определения сценария запуска следует остальной код приложения
}
```

Упаковщики образов

Часто для уменьшения объема ввода/вывода используют прием сжатия исходных данных. В конце концов, бессмысленно загружать 15 Гбайт установочных файлов Windows в распакованном виде, если в сжатом виде они умещаются на единственном DVD-диске. Эту идею можно распространить и на управляемые приложения, хранящиеся на диске. Распаковка приложения только после загрузки в память может существенно повысить скорость холодного запуска за счет уменьшения количества выполняемых при этом операций ввода/вывода. Сжатие – обоюдоострое оружие, потому что распаковывание сжатого кода и данных приводит к увеличению затрат процессорного времени, но иногда стоимость процессорного времени оказывается достаточно приемлемой, когда скорость запуска играет критически важную роль, как, например, в приложениях для платежных терминалов, которые должны запускаться максимально быстро после загрузки системы.

Существует несколько коммерческих и свободно распространяемых утилит сжатия для приложений (которые обычно называются

упаковщиками). Если у вас уже имеется подобный упаковщик, проверьте – поддерживает ли он сжатие приложений для .NET applications. Некоторые упаковщики могут обслуживать только двоичные файлы неуправляемых приложений. В качестве примера упаковщика, способного упаковывать приложения для .NET, можно назвать MPress, который распространяется бесплатно и доступен по адресу: http://www.matcode.com/mpress.htm. Еще одним примером может служить упаковщик Rugland Packer for .NET Executables (RPX) – свободно распространяемая утилита, доступная по адресу: http://rpx.codeplex.com/. Ниже приводится пример вывода утилиты RPX, запущенной для обработки небольшого приложения:

```
> Rpx.exe Shlook.TestServer.exe Shlook.Common.dll Shlook.Server.dll

Rugland Packer for (.Net) eXecutables 1.3.4399.43191
                                                                100.0%
Unpacked size :.............27.00 KB
Packed size   :.............13.89 KB
Compression   :.............48.55%
-------------------------------------------
Application target is the console
-------------------------------------------
Uncompressed size :.............27.00 KB
Startup overhead  :.............5.11 KB
Final size        :.............19.00 KB
-------------------------------------------
Total compression :.............29.63%
```

Управляемая оптимизация на основе профилирования

Управляемая оптимизация на основе профилирования (Managed Profile Guided Optimization, MPGO) – это инструмент, появившийся в Visual Studio 11 и CLR 4.5 и оптимизирующий размещение на диске низкоуровневых образов, созданных с помощью NGen. MPGO генерирует информацию об определенном периоде выполнения приложения и сохраняет ее в сборке. Впоследствии NGen использует эту информацию для оптимизации размещения сгенерированных образов.

Оптимизация образов с помощью MPGO выполняется двумя способами. Во-первых, MPGO обеспечивает совместное хранение часто используемого кода и данных на диске. Как результат, будет меньше ошибок доступа к страницам памяти при обращении к данным,

потому что больше часто используемых данных уместится в одной странице памяти. Во-вторых, MPGO обеспечивает совместное хранение на диске данных, которые наверняка будут изменяться. Когда страница с данными, совместно используемыми несколькими процессами, изменяется, Windows создает скрытую копию страницы для процесса, изменившего ее (этот прием называется *копированием при записи* (copy-on-write)). В результате такой оптимизации уменьшается количество изменяющихся, совместно используемых страниц, и, соответственно, уменьшается объем копирования и повышается эффективность использования памяти.

Чтобы оптимизировать приложение с помощью MPGO, необходимо передать инструменту список сборок, указать каталог для сохранения оптимизированных двоичных файлов и максимальное время ожидания, после которого следует остановить профилирование. MPGO внедряет в приложение инструментирующий код, выполняет его, анализирует результаты и создает оптимизированные низкоуровневые образы для указанных вами сборок:

```
> mpgo.exe -scenario Ch10.exe -assemblylist Ch10.exe HelperLibrary.dll
  -OutDir . -NoClean

Successfully instrumented assembly D:\Code\Ch10.exe
Successfully instrumented assembly D:\Code\HelperLibrary.dll

< вывод приложения опущен >

Successfully removed instrumented assembly D:\Code\Ch10.exe
Successfully removed instrumented assembly D:\Code\HelperLibrary.dll
Reading IBC data file: D:\Code\Ch10.ibc
The module D:\Code\Ch10-1.exe, did not contain an IBC resource
Writing profile data in module D:\Code\Ch10-1.exe
Data from one or more input files has been upgraded to a newer version.
Successfully merged profile data into new file D:\Code\Ch10-1.exe
Reading IBC data file: D:\Code\HelperLibrary.ibc
The module D:\Code\HelperLibrary-1.dll, did not contain an IBC resource
Writing profile data in module D:\Code\HelperLibrary-1.dll
Data from one or more input files has been upgraded to a newer version.
Successfully merged profile data into new file D:\Code\HelperLibrary-1.dll
```

> **Примечание.** *По завершении оптимизации необходимо вновь запустить NGen, чтобы создать окончательные низкоуровневые образы. Порядок запуска NGen обсуждался выше в этой главе.*

На момент написания этих строк не существовало планов по включению MPGO в пользовательский интерфейс Visual Studio 2012. Ко-

мандная строка – единственный доступный способ добавить все описанные оптимизации в свое приложение. Так как MPGO опирается на использование NGen, это еще одна разновидность оптимизаций, которые лучше выполнять на целевой машине уже после установки.

Различные советы по оптимизации времени запуска

У нас в запасе есть еще несколько советов, не упоминавшихся выше, которые могут сократить время запуска приложения еще на несколько секунд.

Сборки со строгими именами принадлежат глобальному кешу

Если ваши сборки имеют строгие имена, поместите их в глобальный кеш сборок (GAC). Иначе при загрузке сборки потребуется проверить цифровую подпись почти каждой страницы. Проверка строгих имен, когда сборка находится за пределами глобального кеша GAC, уменьшит выгоды, получаемые от применения NGen.

Убедитесь, что низкоуровневые образы не требуют перемещения в памяти

При использовании NGen обязательно проверяйте отсутствие конфликтов базовых адресов получаемых низкоуровневых образов, требующих перемещения кода и данных в памяти. Такое перемещение – довольно дорогостоящая операция, в ходе которой выполняется изменение адресов в коде и создаются копии страниц с кодом, которые в отсутствие конфликтов могли бы использоваться совместно. Узнать базовый адрес образа можно с помощью утилиты *dumpbin.exe*, запустив ее с флагом `/headers`, как показано ниже:

```
> dumpbin.exe /headers Ch10.ni.exe

Microsoft (R) COFF/PE Dumper Version 11.00.50214.1
Copyright (C) Microsoft Corporation. All rights reserved.

Dump of file Ch10.ni.exe

PE signature found

File Type: DLL

FILE HEADER VALUES
```

```
             14C machine (x86)
               4 number of sections
        4F842B2C time date stamp Tue Apr 10 15:44:28 2012
               0 file pointer to symbol table
               0 number of symbols
              E0 size of optional header
            2102 characteristics
                   Executable
                   32 bit word machine
                   DLL

OPTIONAL HEADER VALUES
             10B magic # (PE32)
           11.00 linker version
               0 size of code
               0 size of initialized data
               0 size of uninitialized data
               0 entry point
               0 base of code
               0 base of data
        30000000 image base (30000000 to 30005FFF)
            1000 section alignment
             200 file alignment
            5.00 operating system version
            0.00 image version
            5.00 subsystem version
               0 Win32 version
            6000 size of image
< часть вывода опущена для экономии >
```

Чтобы изменить базовый адрес низкоуровневого образа, измените базовый адрес в свойствах проекта в Visual Studio. Базовый адрес можно найти в диалоге **Advanced Build Settings** (Дополнительные параметры построения), открывающемся после щелчка на кнопке **Advanced** (Дополнительно) на вкладке **Build** (Построение) (см. рис. 10.1).

Начиная с версии .NET 3.5 SP1, NGen автоматически использует механизм рандомизации адресного пространства (Address Space Layout Randomization, ASLR), когда приложение выполняется в Windows Vista или в более новой версии Windows. При использовании механизма ASLR, по соображениям безопасности базовый адрес образов выбирается случайным образом при каждом запуске приложения. В этой ситуации проблема перемещения сборок во избежание конфликтов базовых адресов в Windows Vista и более новых версиях не имеет такого большого значения.

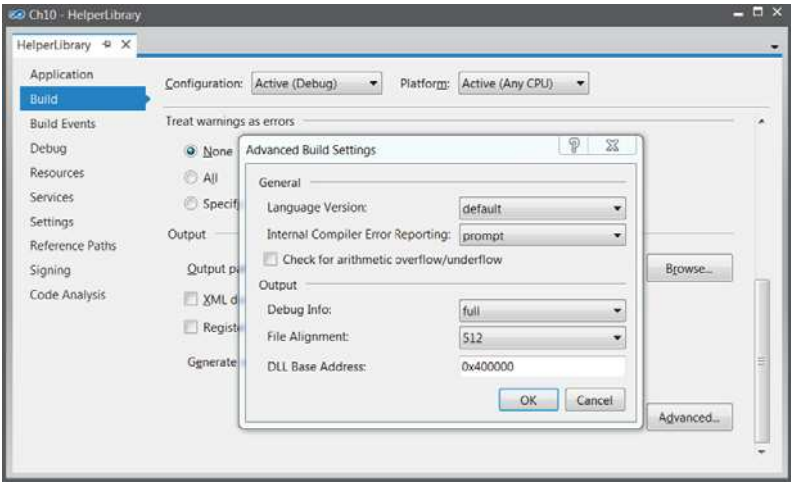

Рис. 10.1. Диалог Advanced Build Settings (Дополнительные параметры построения) в Visual Studio, где можно изменить базовый адрес низкоуровневого образа, генерируемого утилитой NGen.

Уменьшайте общее количество сборок

Уменьшайте количество сборок, загружаемых вашим приложением. Загрузка каждой сборки стоит фиксированную цену, независимо от ее размера. Кроме того, доступ к данным и вызовы методов через границы сборок могут стоить дороже. Большие приложения, загружающие сотни сборок, не так уж редки и время их загрузки можно уменьшить на несколько секунд, объединив сборки в несколько двоичных файлов.

Аппаратно-зависимые оптимизации

Теоретически, разработчиков приложений для .NET никогда не должен волновать вопрос оптимизации под конкретную аппаратную платформу. В конце концов, цель IL и JIT-компилятора состоит в том, чтобы обеспечить возможность выполнения управляемых приложений на любой аппаратной платформе, где установлен фреймворк .NET Framework, и обеспечить независимость от разрядности операционной системы, особенностей процессора и поддерживаемого им набора

машинных инструкций. Однако, чтобы выжать из управляемых приложений максимальную производительность, может потребоваться опуститься до уровня языка ассемблера, как не раз было показано на протяжении всей книги. Иногда понимание особенностей процессора является первым шагом к получению еще более существенного увеличения производительности.

В этом коротком разделе мы рассмотрим несколько примеров оптимизации под конкретные особенности процессоров, которые могут увеличивать производительность приложения на одном компьютере, и не оказывать никакого влияния на другом. В основном мы будем рассматривать процессоры Intel, в частности семейства Nehalem, Sandy Bridge и Ivy Bridge, но большая часть рекомендаций с равным успехом применимы и к процессорам AMD. Поскольку описываемые здесь оптимизации являются достаточно рискованными и могут не давать желаемого эффекта на других аппаратных платформах, вы должны рассматривать эти примеры не как безусловное руководство к действию, а только как возможность увеличить производительность своих приложений в определенных обстоятельствах.

Единственный поток команд и множество потоков данных

Архитектуры с параллельным доступом к данным также называют архитектурами с поддержкой *инструкций, обрабатывающих множество данных* (Single Instruction Multiple Data, SIMD). Эта особенность современных процессоров позволяет обрабатывать большие объемы данных (больше одного машинного слова) единственной инструкцией. Фактическим стандартом инструкций SIMD являются *потоковые SIMD-расширения* (Streaming SIMD Extensions, SSE), используемые в процессорах Intel, начиная с Pentium III. Этот набор инструкций добавляет новые 128-разрядные регистры (с префиксом `XMM`) а также инструкции, оперирующие ими. В последних процессорах Intel появилась поддержка *дополнительных векторных расширений* (Advanced Vector Extensions, AVX), расширяющих набор команд SSE поддержкой 256-разрядных регистров и дополнительными инструкциями SIMD. В качестве примеров инструкций SSE можно привести:

- целочисленная и вещественная арифметика;
- сравнение, перемешивание, преобразование типов данных (из целых в вещественные);
- поразрядные операции;

- поиск минимального и максимального значений, копирование по условию, вычисление контрольной суммы CRC32, определение количества битов, установленных в 1 (эта команда поддерживается набором инструкций SSE4 и более поздних).

Вы можете спросить, не медленнее ли такие инструкции, использующие «новые» регистры, чем стандартные, потому что если это так, любые выгоды в смысле производительности могут оказаться призрачными. К счастью это не так. В процессорах Intel i7 инструкция сложения вещественных чисел (FADD) в 32-разрядных регистрах имеет пропускную способность, равную одной инструкции за такт, и задержку 3 такта. Эквивалентная ей инструкция ADDPS, оперирующая 128-разрядными регистрами, также имеет пропускную способность, равную одной инструкции за такт, и задержку 3 такта.

Задержка и пропускная способность

Термины «задержка» и «пропускная способность» часто используются при обсуждении производительности, и особенно когда речь заходит о «скорости» выполнения инструкций процессором:

- задержка инструкции – это время (обычно измеряемое в тактах), необходимое для выполнения единственной инструкции от начала до конца;
- пропускная способность – это количество инструкций одного типа, которые могут быть выполнены в единицу времени (обычно измеряется в тактах).

Когда мы говорим, что инструкция FADD имеет задержку 3 такта, это означает, что единственная операция FADD выполняется в течение 3 тактов. Когда мы говорим, что инструкция FADD имеет пропускную способность, равную одной инструкции за такт, это означает, что при наличии нескольких инструкций FADD, выполняемых процессором параллельно, скорость выполнения может достигать одной инструкции за такт, то есть процессор может одновременно выполнять три таких инструкции.

Очень часто пропускная способность инструкций значительно выше задержки, потому что процессор может одновременно выполнять несколько инструкций (к этой теме мы еще вернемся ниже).

Использование этих инструкций в глубоком цикле может обеспечить 8-кратный прирост производительности, в сравнении с простейшими последовательными программами, обрабатывающими вещественные или целые числа по одному. Например, взгляните на следующий (специально упрощенный) код:

```
// Предполагается, что A, B и C – массивы
// вещественных чисел одинакового размера
for (int i = 0; i < A.length; ++i) {
```

```
    C[i] = A[i] + B[i];
}
```

Ниже приводится код, генерируемый JIT-компилятором в таких ситуациях:

```
; ESI – указатель на A, EDI – на B, ECX – на C,
; EDX – переменная цикла
    xor  edx,edx
    cmp  dword ptr [esi + 4],0
    jle  END_LOOP
NEXT_ITERATION:
    fld  dword ptr [esi + edx*4 + 8] ; загрузить A[i], без проверки границ
    cmp  edx,dword ptr [edi + 4]     ; проверка границ перед доступом к B[i]
    jae  OUT_OF_RANGE
    fadd dword ptr [edi + edx*4 + 8] ; прибавить B[i]
    cmp  edx,dword ptr [ecx + 4]     ; проверка границ перед доступом к C[i]
    jae  OUT_OF_RANGE
    fstp dword ptr [ecx + edx*4 + 8] ; сохранить в C[i]
    inc  edx
    cmp  dword ptr [esi + 4],edx     ; конец?
    jg NEXT_ITERATION
END_LOOP:
```

В каждой итерации цикла выполняется единственная инструкция FADD, складывающая два 32-разрядных вещественных числа. Однако, при использовании 128-разрядных инструкций SSE, одновременно можно было бы выполнять четыре итерации цикла, как показано ниже (следующий код не выполняет проверку границ и предполагает, что число итераций кратно 4):

```
    xor  edx, edx
NEXT_ITERATION:
    movups xmm1, xmmword ptr [edi + edx*4 + 8] ; 16 байт из B в xmm1
    movups xmm0, xmmword ptr [esi + edx*4 + 8] ; 16 байт из A в xmm0
    addps xmm1, xmm0                           ; сложить xmm0 и xmm1
                                               ; и сохранить результат в xmm1
    movups xmmword ptr [ecx + edx*4 + 8], xmm1 ; 16 байт из xmm1 в C
    add  edx, 4                                ; индекс цикла увеличить на 4
    cmp  edx, dword ptr [esi + 4]
    jg NEXT_ITERATION
```

На процессоре с поддержкой AVX в каждой итерации можно было бы обрабатывать еще больше данных (с использованием 256-разрядных регистров YMM*), и получить еще больший прирост производительности:

```
    xor edx, edx
NEXT_ITERATION:
    vmovups ymm1, ymmword ptr [edi + edx*4 + 8] ; 32 байта из B в ymm1
```

```
vmovups ymm0, ymmword ptr [esi + edx*4 + 8] ; 32 байта из A в ymm0
vaddps ymm1, ymm1, ymm0                     ; сложить ymm0 и ymm1
                                            ; и сохранить результат в ymm1
vmovups ymmword ptr [ecx + edx*4 + 8], ymm1 ; 32 байта из ymm1 в C
add edx, 8                                  ; индекс цикла увеличить на 8
cmp edx, dword ptr [esi + 4]
jg NEXT_ITERATION
```

Примечание. *Инструкции SIMD, используемые в этих примерах, – это лишь вершина айсберга. Современные приложения и игровые программы используют инструкции SIMD для выполнения сложных операций, включая вычисление скалярного произведения, перемещение данных между регистрами и памятью, вычисление контрольных сумм и многих других. Более подробную информацию о возможностях архитектуры AVX можно получить на портале Intel: http://software.intel.com/ru-ru/avx/.*

JIT-компилятор использует лишь малую часть инструкций SSE, даже при том, что они доступны практически во всех процессорах, выпускавшихся в последние 10 лет. В частности, JIT-компилятор использует SSE-инструкцию `MOVQ` для копирования структур среднего размера через регистры `XMM*` (при копировании больших инструкций используется `REP MOVS`), а также SSE2-инструкции для преобразования вещественных чисел в целые. Но JIT-компилятор *не выполняет* векторизацию циклов, как это было сделано вручную в примерах выше, хотя современные компиляторы C++ (включая Visual Studio 2012) поддерживают такую возможность.

К сожалению в C# отсутствуют какие-либо ключевые слова, позволяющие встраивать ассемблерный код в управляемые программы. Конечно, вы всегда можете реализовать чувствительные к производительности части приложений в виде модулей на C++ и использовать механизм взаимодействий в .NET для доступа к ним, но это довольно неудобно. Существует два других подхода к встраиванию инструкций SIMD, не прибегая к созданию отдельного модуля.

Простейший способ выполнить произвольный машинный код в управляемом приложении (с применением легковесной программной прослойки) заключается в том, чтобы динамически сгенерировать машинный код и затем вызвать его. Ключевым здесь является метод `Marshal.GetDelegateForFunctionPointer`, возвращающий управляемого делегата, указывающего на местоположение в неуправляемой памяти, где может храниться произвольный код. Следующий пример выделяет виртуальную память с флагом защиты `EXECUTE_READWRITE`, позволяющим скопировать байты кода в память и затем выполнить их. Как результат, на процессоре Intel i7-860 скорость выполнения увеличивается более чем в два раза!

```
[UnmanagedFunctionPointer(CallingConvention.StdCall)]
delegate void VectorAddDelegate(float[] C, float[] B, float[] A, int length);

[DllImport("kernel32.dll", SetLastError = true)]
static extern IntPtr VirtualAlloc(
  IntPtr lpAddress, UIntPtr dwSize, IntPtr flAllocationType,
  IntPtr flProtect);

// Следующий массив байтов был получен с помощью ассемблера
// с поддержкой SSE – это законченная функция, принимающая
// четыре параметра (три вектора и длину) и складывающая их
byte[] sseAssemblyBytes = {
    0x8b, 0x5c, 0x24, 0x10, 0x8b, 0x74, 0x24, 0x0c, 0x8b, 0x7c, 0x24,
    0x08, 0x8b, 0x4c, 0x24, 0x04, 0x31, 0xd2, 0x0f, 0x10, 0x0c, 0x97,
    0x0f, 0x10, 0x04, 0x96, 0x0f, 0x58, 0xc8, 0x0f, 0x11, 0x0c, 0x91,
    0x83, 0xc2, 0x04, 0x39, 0xda, 0x7f, 0xea, 0xc2, 0x10, 0x00 };

IntPtr codeBuffer = VirtualAlloc(
    IntPtr.Zero,
    new UIntPtr((uint)sseAssemblyBytes.Length),
    0x1000 | 0x2000, // MEM_COMMIT | MEM_RESERVE
    0x40             // EXECUTE_READWRITE
);
Marshal.Copy(sseAssemblyBytes, 0, codeBuffer, sseAssemblyBytes.Length);
VectorAddDelegate addVectors = (VectorAddDelegate)
    Marshal.GetDelegateForFunctionPointer(codeBuffer,
                                          typeof(VectorAddDelegate));
// Теперь для сложения векторов можно использовать 'addVectors'!
```

Совершенно иной подход, который, к сожалению, не поддерживается средой выполнения Microsoft CLR, заключается в расширении JIT-компилятора и добавлении в него возможности генерировать инструкции SIMD. Этот подход поддерживается сборкой *Mono.Simd*. Разработчики управляемых приложений, использующие среду выполнения Mono .NET, могут подключить сборку Mono.Simd и использовать поддержку JIT-компилятора, преобразующего операции с такими типами, как `Vector16b` и `Vector4f` в соответствующие инструкции SSE. За дополнительной информацией о Mono.Simd обращайтесь к официальной документации, по адресу: http://docs.go-mono.com/index.aspx?link=N:Mono.Simd.

Распараллеливание инструкций

В отличие от параллельного доступа к данным, когда для работы с более крупными фрагментами данных используются специализированные инструкции, *распараллеливание на уровне инструкций* (Instruction-Level Parallelism, ILP) – это механизм, позволяющий

одновременно выполнять несколько инструкций на одном и том же процессоре. Современные процессоры обладают емкими конвейерами с несколькими типами исполнительных устройств, таких как устройство доступа к памяти, устройство для выполнения арифметических операций и устройство для декодирования инструкций процессора. Конвейеры позволяют одновременно выполнять несколько инструкций, при условии, что они не конкурируют за обладание одними и теми же участками конвейера, и между ними нет никаких *зависимостей по данным*. Зависимости по данным возникают, когда одной инструкции требуются результаты выполнения предыдущей инструкции; например, когда инструкция читает из ячейки в памяти, куда выполняется запись предыдущей инструкцией.

> **Примечание.** *Распараллеливание инструкций не имеет никакого отношения к параллельному программированию, обсуждавшемуся в главе 6. Применение приемов параллельного программирования заключается в создании функций, выполняющихся в нескольких потоках на нескольких процессорах. Распараллеливание инструкций обеспечивает одновременное выполнение нескольких инструкций из одного потока выполнения на единственном процессоре. В отличие от параллельного программирования, ILP гораздо сложнее поддается управлению и сильно зависит от оптимизаций, примененных в программе.*

В дополнение к конвейерам, процессоры поддерживают также *суперскалярное выполнение* (superscalar execution), когда на одном и том же процессоре одновременно используется несколько не занятых устройств для выполнения однотипных операций. Кроме того, чтобы уменьшить влияние зависимости по данным на одновременное выполнение инструкций, процессоры будут выполнять инструкции в оригинальном порядке, пока не будут нарушены какие-либо зависимости по данным. Применяя приемы *интеллектуального выполнения* (в первую очередь, пытаясь угадать, какая ветвь условной инструкции будет выполняться), процессор сможет выполнить дополнительные инструкции, даже когда следующая инструкция в оригинальной последовательности не сможет быть выполнена из-за зависимости по данным.

Оптимизирующие компиляторы известны своей способностью изменять порядок выполнения инструкций, чтобы максимально использовать возможность процессора одновременно выполнять несколько инструкций. JIT-компилятор не обладает какими-то выдающимися возможностями, но способность современных процессоров выполнять инструкции не по порядку может компенсировать это. Однако,

не совсем удачные программы могут оказывать существенное отрицательное влияние на производительность, вводя нежелательную зависимость между инструкциями – особенно в циклах – ограничивая тем самым возможность параллельного выполнения инструкций.

Взгляните на следующие три цикла:

```
for (int k = 1; k < 100; ++k) {
    first[k] = a * second[k] + third[k];
}
for (int k = 1; k < 100; ++k) {
    first[k] = a * second[k] + first[k - 1];
}
for (int k = 1; k < 100; ++k) {
    first[k] = a * first[k - 1] + third[k];
}
```

Мы миллион раз выполнили эти циклы на одном из наших компьютеров с массивами из 100 целых чисел. Первый цикл выполнялся в среднем 190 миллисекунд, второй примерно 210 миллисекунд и третий – около 270 миллисекунд. Такая существенная разница объясняется особенностями распараллеливания инструкций. В первом цикле отсутствуют какие-либо зависимости между инструкциями – итерации могут выполняться в конвейере процессора в любом порядке и одновременно. Во втором цикле наблюдается зависимость – значение `first[k]` зависит от значения `first[k-1]`. Однако операция умножения (которая должна быть выполнена до сложения) лишена такой зависимости. В третьем цикле ситуация складывается еще хуже: даже инструкция умножения не может быть выполнена независимо от результатов предыдущей итерации.

Другой пример – поиск максимального значения в массиве целых чисел. В простейшей реализации каждая следующая итерация зависит от текущего максимального значения, установленного в предыдущей итерации. Как ни странно, но здесь мы можем применить ту же идею, что и в главе 6 – разделить диапазон на поддиапазоны, а затем сделать выводы по локальным результатам. В частности, поиск максимального значения в массиве можно разделить на поиск максимальных значений среди четных и нечетных элементов, а затем выполнить одну дополнительную операцию определения наибольшего из них. Ниже показаны оба подхода:

```
// Простейший алгоритм, страдающий зависимостью между итерациями
int max = arr[0];
for (int k = 1; k < 100; ++k) {
    max = Math.Max(max, arr[k]);
```

```
}

// оптимизированный алгоритм, устраняющий некоторые зависимости внутри
// итераций, две строки могут выполняться процессором параллельно
int max0 = arr[0];
int max1 = arr[1];
for (int k = 3; k < 100; k + = 2) {
    max0 = Math.Max(max0, arr[k-1]);
    max1 = Math.Max(max1, arr[k]);
}
int max = Math.Max(max0, max1);
```

К сожалению, JIT-компилятор в CLR препятствует данной конкретной оптимизации, генерируя недостаточно оптимальный машинный код для второго цикла. В первом цикле наиболее важные значения (`max` и `k`) умещаются и хранятся в регистрах. Во втором цикле JIT-компилятор не способен уместить все значения в регистрах; если `max1` или `max0` будет храниться в памяти, это приведет к значительной потере производительности. Соответствующая реализация на C++ дает ожидаемую прибавку в скорости – деление задачи поиска пополам увеличивает производительность в два раза, а еще одно деление (поиск четырех локальных максимумов) увеличивает производительность еще на 25%.

Распараллеливание инструкций можно объединить с распараллеливанием данных. Оба примера, рассматриваемые ниже (цикл, выполняющий умножение и сложение, и поиск максимального значения), могут получить дополнительные выгоды от использования инструкций SIMD. Для поиска максимального значения используется SSE4-инструкция `PMAXSD`, оперирующая двумя множествами из четырех упакованных 32-разрядных целых чисел и отыскивающая максимальное значение в каждой паре чисел в двух множествах. Следующий код (использующий встроенные функции Visual C++ из заголовочного файла `<smmintrin.h>`) выполняется в 3 раза быстрее, чем лучшая версия, представленная выше, и в 7 раз быстрее, чем простейшая реализация:

```
__m128i max0 = *(__m128i*)arr;
for (int k = 4; k < 100; k + = 4) {
    max0 = _mm_max_epi32(max0, *(__m128i*)(arr + k)); //Emits PMAXSD
}
int part0 = _mm_extract_epi32(max0, 0);
int part1 = _mm_extract_epi32(max0, 1);
int part2 = _mm_extract_epi32(max0, 2);
int part3 = _mm_extract_epi32(max0, 3);
int finalmax = max(part0, max(part1, max(part2, part3)));
```

Минимизировав зависимости между инструкциями, чтобы получить дополнительные выгоды от параллельного их выполнения, можно дополнительно получить весьма существенный прирост производительности за счет распараллеливания данных (иногда называют векторизацией (vectorization)).

Управляемый и неуправляемый код

Оппоненты .NET часто говорят, что управляемая природа среды выполнения CLR привносит дополнительные накладные расходы, что делает невозможным реализацию высокопроизводительных алгоритмов с использованием C#, .NET Framework и CLR. На протяжении всей этой книги и даже в этой главе мы видели разные проблемы производительности, о которых вы должны помнить, если хотите выжать из своих управляемых приложений максимальную производительность. К сожалению, *всегда* будут возникать ситуации, когда неуправляемый код (на C++, C или даже на языке Ассемблера) будет показывать более высокую производительность, чем его управляемый аналог.

Мы не собираемся анализировать и классифицировать каждый пример, когда реализация алгоритма на C++ обладает более высокой производительностью, чем его версия на C#. Однако все они обладают некоторыми общими чертами, проявляющимися чаще других.

- Численные алгоритмы, реализованные на C++, обычно выполняются быстрее, даже после применения оптимизаций на C#. Основными причинами, как правило, являются: проверки границ (которые JIT-компилятором устраняются лишь в некоторых случаях и только при работе с одномерными массивами), применение инструкций SIMD компиляторами C++ и другие оптимизации, доступные компиляторам C++, такие как сложное встраивание и оптимальное использование регистров.
- Некоторые шаблоны управления памятью отрицательно сказываются на производительности сборщика мусора (как было показано в главе 4). Иногда в коде на C++ можно использовать «правильные» приемы управления памятью, организуя пулы памяти или повторно используя выделенные блоки неуправляемой памяти, полученные из других источников, недоступных коду .NET.
- Код на C++ обладает непосредственным доступом к Win32 API и не требует привлечения механизмов взаимодействий, таких как маршалинг параметров и передачи их через границы потоков выполнения (о чем рассказывалось в главе 8). Высокопроизводительные приложения, использующие слишком подробный интерфейс операционной системы, в .NET могут выполняться значительно медленнее из-за наличия промежуточного уровня, обеспечивающего возможность взаимодействий.

На сайте CodeProject можно найти замечательную статью «Head-to-head benchmark: C++ vs. NET» Дэвида Пайпграсса (David Piepgrass) (доступную по адресу: http://www.codeproject.com/Articles/212856/Head-to-head-benchmark-Csharp-vs-NET), где автор разрушает некоторые ошибочные представления о производительности управляемого кода. Например, в своей статье Пайпграсс демонстрирует, что коллекции в .NET обладают

значительно более высокой производительностью, чем некоторые эквиваленты из стандартной библиотеки шаблонов C++. То же относится и к построчному чтению файлов с использованием `ifstream` и `StreamReader`. С другой стороны, некоторые из его тестов подчеркивают отсутствие некоторых возможностей в 64-разрядном JIT-компиляторе, а нехватка поддержки инструкций SIMD в CLR (о чем рассказывалось выше) является еще одним фактором, обеспечивающим преимущество C++.

Исключения

Исключения – не очень дорогостоящий механизм при правильном и экономном использовании. Есть несколько простых правил, следуя которым можно избежать рисков, связанных с возбуждением слишком большого количества исключений в ущерб производительности.

- Используйте исключения только в исключительных случаях: если предполагается, что исключения будут происходить слишком часто, подумайте о применении защитного стиля программирования вместо возбуждения исключений. Из этого правила есть исключения (преднамеренная игра слов), но когда требуется высокая производительность, если исключение может возникать более чем в 10 случаях из 100, такие ситуации не следует обрабатывать с применением исключений.
- Проверяйте исключительные условия перед вызовом метода, который может возбудить исключение. Примером такого подхода может служить реализация свойства `Stream.CanRead` и семейства статических методов `TryParse` типов значений (например, `int.TryParse`).
- Не используйте исключения, как механизм управления потоком выполнения: не возбуждайте исключение, чтобы организовать выход из цикла, прекратить чтение из файла или вернуть данные из метода.

Наиболее ощутимые накладные расходы, связанные с возбуждением и обработкой исключений, можно разделить на несколько категорий:

- чтобы сконструировать исключение, требуется выполнить обход стека вызовов (для получения трассировочной информации), причем эти накладные расходы тем больше, чем глубже оказывается стек вызовов;
- возбуждение и обработка исключений требует взаимодействий с неуправляемым кодом – механизмом обработки прог-

раммных и аппаратных исключений в ОС Windows (Windows Structured Exception Handling, SEH) – и выполнения цепочки обработчиков в SEH;

- исключения изменяют направление потока выполнения и потока данных, вызывая ошибки чтения страниц памяти и промахи кеша.

Чтобы выяснить, оказывают ли исключения отрицательное влияние на производительность, можно воспользоваться счетчиками производительности из категории **.NET CLR Exceptions** (Исключений CLR .NET) (за дополнительной информацией о счетчиках производительности обращайтесь к главе 2). В частности, счетчик **# of Exceps Thrown/sec** (Число исключений/сек) может помочь точно определить проблемы с производительностью, когда возбуждаются тысячи исключений в секунду.

Механизм рефлексии

Механизм рефлексии (Reflection) заработал плохую репутацию «убийцы производительности» во многих сложных приложениях. Отчасти такая репутация вполне заслуженна: в механизме рефлексии имеются чрезвычайно дорогостоящие операции, такие как вызов функции по имени с использованием `Type.InvokeMember` или создание экземпляра объекта вызовом `Activator.CreateInstance` с заданными параметрами. Основные накладные расходы, возникающие при вызове методов или изменении значений свойств с помощью механизма рефлексии, связаны с необходимостью выполнения операций, выполняемых в фоновом режиме – вместо строго типизированного кода, который может быть скомпилирован в машинные инструкции, операции механизма рефлексии фактически интерпретируются в последовательность вызовов дорогостоящих методов.

Например, чтобы вызвать метод с применением `Type.InvokeMember`, требуется определить вызываемый метод с помощью метаданных и поиска среди перегруженных версий, убедиться, что указанные аргументы соответствуют параметрам метода, попутно выполнив приведение типов, если это необходимо, проверить отсутствие каких-либо проблем, связанных с безопасностью, и, наконец, произвести вызов метода. Так как механизм рефлексии опирается на параметры типа `object` и возвращаемые значения, упаковка и распаковка значений могут привнести дополнительные накладные расходы.

Примечание. *Дополнительные советы по применению .NET Reflection API можно найти в статье «Dodge Common Performance Pitfalls to Craft Speedy Applications» Джоэла Побара (Joel Pobar) на сайте MSDN Magazine: http://msdn.microsoft.com/en-us/magazine/cc163759.aspx.*

Достаточно часто применение механизма рефлексии можно исключить из приложений, используя некоторую форму *генерации кода* – вместо обработки неизвестных типов и динамического вызова методов/свойств, можно сгенерировать код (для каждого типа) которые выполнит необходимые операции в строго типизированной манере.

Генерация кода

Прием генерации кода часто используется фреймворками, выполняющими сериализацию, такими как механизмы объектно-реляционного отображения (Object/Relational Mappers, ORM), динамические прокси-объекты и другими, которые вынуждены работать с объектами неизвестных типов. В .NET Framework поддерживается несколько способов динамической генерации кода, и еще больше способов поддерживается сторонними фреймворками, такими как LLBLGen и T4.

- Легковесный генератор кода (Lightweight Code Generation, LCG), он же класс `DynamicMethod`. Этот API можно использовать для генерации метода без создания типа и сборки, содержащих его. Для коротких методов – это самый эффективный механизм генерации кода. Чтобы сгенерировать код с помощью механизма LCG, следует создать класс `ILGenerator`, который работает непосредственно с инструкциями на языке IL.
- Пространство имен `System.Reflection.Emit` содержит методы, с помощью которых можно генерировать сборки, типы и методы на языке IL.
- Деревья выражений (expression trees), имеющиеся в пространстве имен `System.Linq.Expression` можно использовать для создания легковесных выражений из последовательного представления.
- Класс `CSharpCodeProvider` можно использовать для непосредственной компиляции исходного кода на C# (из строки или из файла) в сборку.

Генерация из исходного кода

Допустим, что вам требуется реализовать фреймворк, выполняющий сериализацию произвольных объектов и сохраняющий результаты в формате XML. Получение непустых, общедоступных полей с использованием Reflection API и их запись – весьма дорогостоящая операция, но ее вполне можно использовать для создания простейших реализаций:

```
// Упрощенный метод сериализации в XML - не поддерживает
// коллекции, циклические ссылки и т. д.
public static string XmlSerialize(object obj) {
    StringBuilder builder = new StringBuilder();
    Type type = obj.GetType();
    builder.AppendFormat(" <{0} Type = '{1}'"> ", type.Name,
                         type.AssemblyQualifiedName);
    if (type.IsPrimitive || type == typeof(string)) {
        builder.Append(obj.ToString());
    } else {
        foreach (FieldInfo field in type.GetFields()) {
            object value = field.GetValue(obj);
            if (value ! = null) {
                builder.AppendFormat(" <{0}> {1}</{0}> ", field.Name,
                                     XmlSerialize(value));
            }
        }
    }
    builder.AppendFormat("</{0} > ", type.Name);
    return builder.ToString();
}
```

Вместо этого можно было бы сгенерировать строго типизированный код, выполняющий сериализацию объектов конкретного типа, и вызывать этот код. Ниже представлена реализация, использующая `CSharpCodeProvider`:

```
public static string XmlSerialize<T>(T obj){
    Func<T,string> serializer = XmlSerializationCache<T>.Serializer;
    if (serializer == null){
        serializer = XmlSerializationCache<T>.GenerateSerializer();
    }
    return serializer(obj);
}

private static class XmlSerializationCache <T> {
    public static Func <T,string> Serializer;
    public static Func <T,string> GenerateSerializer() {
        StringBuilder code = new StringBuilder();
        code.AppendLine("using System;");
```

```
        code.AppendLine("using System.Text;");
        code.AppendLine("public static class SerializationHelper {");
        code.AppendFormat("public static string XmlSerialize({0} obj) {{",
                          typeof(T).FullName);
        code.AppendLine("StringBuilder result = new StringBuilder();");
        code.AppendFormat("result.Append(\" <{0} Type = '{1}'> \");",
                          typeof(T).Name, typeof(T).AssemblyQualifiedName);
        if (typeof(T).IsPrimitive || typeof(T) == typeof(string)) {
            code.AppendLine("result.AppendLine(obj.ToString());");
        } else {
            foreach (FieldInfo field in typeof(T).GetFields()) {
                code.AppendFormat("result.Append(\" < {0} > \");",
                                  field.Name);
                code.AppendFormat("result.Append(XmlSerialize(obj.{0}));",
                                  field.Name);
                code.AppendFormat("result.Append(\"</{0} > \");", field.Name);
            }
        }
        code.AppendFormat("result.Append(\"</{0} > \");", typeof(T).Name);
        code.AppendLine("return result.ToString();");
        code.AppendLine("}");
        code.AppendLine("}");

        CSharpCodeProvider compiler = new CSharpCodeProvider();
        CompilerParameters parameters = new CompilerParameters();
        parameters.ReferencedAssemblies.Add(typeof(T).Assembly.Location);
        parameters.CompilerOptions = "/optimize + ";
        CompilerResults results = compiler.CompileAssemblyFromSource(
                                      parameters, code.ToString());
        Type serializationHelper = results.CompiledAssembly.GetType(
                                       "SerializationHelper");
        MethodInfo method = serializationHelper.GetMethod("XmlSerialize");
        Serializer = (Func <T,string>)Delegate.CreateDelegate(
                                     typeof(Func <T,string>), method);
        return Serializer;
    }
}
```

Код, использующий Reflection API, выполняется только один раз при генерации строго типизированного кода – результат кешируется в статическом поле и повторно используется всякий раз, когда требуется сериализовать экземпляр данного конкретного типа. Обратите внимание, что пример, представленный выше, не прошел всестороннее тестирование; он лишь доказывает возможность реализации генератора кода. Простой хронометраж показывает, что подход на основе генерации кода более чем в два раза быстрее, чем первоначальный подход, основанный исключительно на применении Reflection API.

Генерация кода с использованием легковесного генератора кода

Рассмотрим еще один пример из области синтаксического анализа сетевого протокола. Допустим, что имеется объемный поток двоичных данных, например, состоящий из сетевых пакетов, и вам нужно выделить из пакетов заголовки и полезные данные. Например, взгляните на следующую структура заголовка пакета (это полностью искусственный пример – заголовки TCP-пакетов имеют совершенно иную структуру):

```
public struct TcpHeader {
    public uint SourceIP;
    public uint DestIP;
    public ushort SourcePort;
    public ushort DestPort;
    public uint Flags;
    public uint Checksum;
}
```

Реализация на C/C++ извлечения такой структуры из потока байтов – тривиальная задача, и для этого не требуется даже копировать данные, если использовать указатели. Фактически, извлечение любых структур из потока байтов, реализуется очень просто:

```
template <typename T>
const T* get_pointer(const unsigned char* data, int offset) {
    return (T*)(data + offset);
}

template <typename T>
const T get_value(const unsigned char* data, int offset) {
    return *get_pointer(data, offset);
}
```

В C# все оказывается гораздо сложнее. Существует множество способов чтения произвольных двоичных данных из потока. Один из них – выделить поля с помощью Reflection API и читать их отдельно от потока байтов:

```
// Поддерживаются только некоторые простые поля
public static void ReadReflectionBitConverter<T>(
                        byte[] data, int offset, out T value) {
    object box = default(T);
    int current = offset;
    foreach (FieldInfo field in typeof(T).GetFields()) {
        if (field.FieldType == typeof(int)) {
            field.SetValue(box, BitConverter.ToInt32(data, current));
```

```
            current + = 4;
        } else if (field.FieldType == typeof(uint)) {
            field.SetValue(box, BitConverter.ToUInt32(data, current));
            current + = 4;
        } else if (field.FieldType == typeof(short)) {
            field.SetValue(box, BitConverter.ToInt16(data, current));
            current + = 2;
        } else if (field.FieldType == typeof(ushort)) {
            field.SetValue(box, BitConverter.ToUInt16(data, current));
            current + = 2;
    }

    // ...множество других типов опущено для экономии места
    value = (T)box;
}
```

На одном из наших тестовых компьютеров мы выполнили анализ 1 000 000 20-байтных структур `TcpHeader` и выяснили, что в среднем метод выполняется примерно 170 миллисекунд. Скорость работы выглядит не так уж и плохо, но объем выделяемой при этом памяти операциями упаковки оказался внушительным. Кроме того, при вполне реальной скорости обмена по сети, равной 1 Гбит/сек, приложение будет получать десятки миллионов пакетов в секунду, что потребует значительных затрат вычислительных ресурсов только на чтение структур из входящих данных.

На этом фоне гораздо более удачным выглядит решение, основанное на использовании метода `Marshal.PtrToStructure`, предназначенного для преобразования фрагментов неуправляемой памяти в управляемые структуры. этот подход требует закрепления исходных данных в памяти, чтобы получить возможность извлекать их по указателю:

```
public static void ReadMarshalPtrToStructure<T>(
                        byte[] data, int offset, out T value) {
    GCHandle gch = GCHandle.Alloc(data, GCHandleType.Pinned);
    try {
        IntPtr ptr = gch.AddrOfPinnedObject();
        ptr + = offset;
        value = (T)Marshal.PtrToStructure(ptr, typeof(T));
    } finally {
        gch.Free();
    }
}
```

Эта версия показывает гораздо более высокую производительность – в среднем 39 миллисекунд на 1 000 000 пакетов. Это существенное улучшение, но `Marshal.PtrToStructure` все так же использует

динамическую память, потому что возвращает ссылку на объект, и к тому же скорость работы явно недостаточна для обслуживания десятков миллионов пакетов в секунду.

В главе 8 мы исследовали использование указателей и небезопасного кода в C#, и похоже, что данный пример – отличная возможность использовать их. В конце концов, версия на C++ настолько проста именно потому, что использует указатели. Следующий код работает намного, намного быстрее, обрабатывая 1 000 000 пакетов за 0.45 миллисекунды – невероятное улучшение!

```
public static unsafe void ReadPointer(byte[] data,
                            int offset, out TcpHeader header) {
    fixed (byte* pData = &data[offset]) {
        header = *(TcpHeader*)pData;
    }
}
```

Почему этот способ оказался таким быстрым? Потому что для копирования данных больше не используются такие ресурсоемкие API, как `Marshal.PtrToStructure` – копирование выполняет сам JIT-компилятор. Машинный код, полученный в результате компиляции этого метода, может встраиваться (в действительности 64-разрядный JIT-компилятор так и поступает) и использовать для копирования областей памяти 3–4 инструкции (например, инструкцию `MOVQ` в 32-разрядных системах, копирующую сразу 64 бита). Единственная проблема в том, что получившийся метод `ReadPointer` не так универсален, как версия на C++. Первая реакция на это замечание – реализовать универсальную версию:

```
public static unsafe void ReadPointerGeneric<T>(
                            byte[] data, int offset, out T value) {
    fixed (byte* pData = &data[offset]) {
        value = *(T*)pData;
    }
}
```

которая даже не компилируется! В частности, `T*` – это недопустимая конструкция в C#, потому что нет никакого способа гарантировать, что указатель на `T` можно будет разыменовать (к тому же, закреплять объекты и получать указатели на них можно, только если они имеют двоично совместимые типы, как описывалось в главе 8). Поскольку нет никаких обобщенных средств, чтобы выразить наши намерения, похоже, что мы должны будем написать отдельные версии `ReadPointer` для всех поддерживаемых типов, и в этом нам снова помогут генераторы кода.

Структура TypedReference и два недокументированных ключевых слова в языке C#

В отчаянных ситуациях требуются отчаянные меры, и такими отчаянными мерами являются два недокументированных ключевых слова в языке C#, `__makeref` и `__refvalue` (поддерживаемые такими же недокументированными кодами операций на языке IL). Вместе со структурой `TypedReference` эти ключевые слова используются в некоторых сценариях низкоуровневых взаимодействий с применением методов, имеющих переменное количество аргументов в стиле языка C (что требует применения еще одного недокументированного ключевого слова `__arglist`). `TypedReference` – это небольшая структура с двумя полями типа `IntPtr` – `Type` и `Value`. Поле `Value` – это указатель на значение, которое может быть ссылочного типа или типа значения, а поле `Type` – указатель на таблицу методов типа. Создавая экземпляры `TypedReference`, указывающие на экземпляры типов значений, можно обеспечить интерпретацию содержимого памяти строго типизированным образом, как того требует ситуация, и использовать JIT-компилятор для копирования памяти, как это делается в реализации метода `ReadPointer`.

```
// мы объявляем параметр со спецификатором ref, а не out,
// потому что нам нужен его адрес, а ключевое слово
// __makeref требует инициализированное значение.
public static unsafe void ReadPointerTypedRef<T>(
                            byte[] data, int offset, ref T value){
    // В действительности мы не изменяем 'value' – нам просто
    // требуется левостороннее значение
    TypedReference tr = __makeref(value);
    fixed (byte* ptr = &data[offset]) {
        // Первое поле-указатель в структуре TypedReference - это
        // адрес объекта, поэтому мы записываем в него
        // указатель на нужный элемент в массиве с данными:
        *(IntPtr*)&tr = (IntPtr)ptr;
        // __refvalue копирует указатель из TypedReference в 'value'
        value = __refvalue(tr, T);
    }
}
```

К сожалению, вся эта «магия» компилятора имеет свою цену. В частности, оператор `__makeref` компилируется JIT-компилятором в вызов `call clr!JIT_GetRefAny`, который несет дополнительные накладные расходы? в сравнении с полностью встраиваемой версией `ReadPointer`. Результатом является почти 2-кратная потеря производительности – этот метод обрабатывает 1 000 000 пакетов в среднем за 0.83 миллисекунды. Но, как ни странно, он все еще остается самым быстрым универсальным решением из всех, что будут показаны в этом разделе.

Чтобы избежать необходимости писать отдельные копии метода `ReadPointer` для каждого типа, мы воспользуемся легковесным гене-

ратором кода (классом `DynamicMethod`). Прежде всего исследуем код на языке IL, сгенерированный для метода `ReadPointer`:

```
.method public hidebysig static void ReadPointer(
uint8[] data, int32 offset, [out] valuetype TcpHeader& header) cil managed
{
    .maxstack 2
    .locals init ([0] uint8& pinned pData)
    ldarg.0
    ldarg.1
    ldelema uint8
    stloc.0
    ldarg.2
    ldloc.0
    conv.i
    ldobj TcpHeader
    stobj TcpHeader
    ldc.i4.0
    conv.u
    stloc.0
    ret
}
```

Теперь нам осталось сгенерировать код IL, заменив тип `TcpHeader` аргументом обобщенного типа. Фактически, благодаря превосходному расширению `ReflectionEmitLanguage` для утилиты .NET Reflector (доступен по адресу: http://reflectoraddins.codeplex.com/wikipage?title=ReflectionEmitLanguage), которое преобразует методы в вызовы `Reflection.Emit`, необходимые для генерации кода методов, нам даже не придется писать код вручную – хотя нам придется внести несколько небольших изменений:

```
static class DelegateHolder<T>
{
    public static ReadDelegate<T> Value;
    public static ReadDelegate<T> CreateDelegate() {
        DynamicMethod dm = new DynamicMethod("Read", null,
            new Type[] { typeof(byte[]), typeof(int),
                         typeof(T).MakeByRefType() },
            Assembly.GetExecutingAssembly().ManifestModule);
        dm.DefineParameter(1, ParameterAttributes.None, "data");
        dm.DefineParameter(2, ParameterAttributes.None, "offset");
        dm.DefineParameter(3, ParameterAttributes.Out, "value");
        ILGenerator generator = dm.GetILGenerator();
        generator.DeclareLocal(typeof(byte).MakePointerType(), pinned: true);
        generator.Emit(OpCodes.Ldarg_0);
        generator.Emit(OpCodes.Ldarg_1);
        generator.Emit(OpCodes.Ldelema, typeof(byte));
        generator.Emit(OpCodes.Stloc_0);
        generator.Emit(OpCodes.Ldarg_2);
```

```
            generator.Emit(OpCodes.Ldloc_0);
            generator.Emit(OpCodes.Conv_I);
            generator.Emit(OpCodes.Ldobj, typeof(T));
            generator.Emit(OpCodes.Stobj, typeof(T));
            generator.Emit(OpCodes.Ldc_I4_0);
            generator.Emit(OpCodes.Conv_U);
            generator.Emit(OpCodes.Stloc_0);
            generator.Emit(OpCodes.Ret);
            Value = (ReadDelegate < T>)dm.CreateDelegate(typeof(ReadDelegate<T>));
            return Value;
        }
    }

public static void ReadPointerLCG<T> (byte[] data, int offset, out T value)
{
    ReadDelegate<T> del = DelegateHolder<T>.Value;
    if (del == null) {
        del = DelegateHolder<T>.CreateDelegate();
    }
    del(data, offset, out value);
}
```

Эта версия обрабатывает 1 000 000 пакетов в среднем за 1.05 миллисекунды – более чем в два раза медленнее , чем `ReadPointer`, но все еще на два порядка быстрее оригинальной реализации на основе механизма рефлексии – еще одна победа генератора кода. (Потеря производительности в сравнении `ReadPointer` обусловлена необходимостью получения делегата из статического поля, проверки ссылки на пустое значение и вызов метода с применением делегата.)

В заключение

Различные советы и приемы оптимизации, обсуждавшиеся в этой главе, имеют критически важное значение для высокопроизводительных вычислительных алгоритмов и сложных систем. Гарантируя использование вашим кодом встроенных оптимизаций JIT-компилятора, а также специализированных инструкций процессора, максимально уменьшая время запуска клиентских приложений и воздерживаясь от использования дорогостоящих механизмов CLR, таких как Reflection API и исключения, вы сможете выжать максимальную производительность из своих управляемых программ.

В следующей и заключительной главе мы обсудим характеристики производительности веб-приложений, в первую очередь приложений ASP.NET, и познакомимся с оптимизациями, пригодными только для веб-серверов.

ГЛАВА 11. Производительность веб-приложений

Главная цель веб-приложений – обработка сотен и даже тысяч запросов в секунду. Чтобы обеспечить успех таким приложениям, очень важно определить потенциальные узкие места и сделать все возможное, чтобы избежать проблем. Но узкие места в приложениях ASP.NET могут возникать не только в вашем коде. С момента, когда веб-запрос достигнет сервера, и до момента передачи вашему приложению, запрос проходит через конвейер протокола HTTP, затем через конвейер IIS, после чего передается еще одному конвейеру – конвейеру ASP.NET, и только потом попадает в ваше приложение. А когда приложение заканчивает обработку запроса, отправляемый ответ проходит через те же конвейеры в обратном направлении, пока, наконец, не будет принят клиентским компьютером. Каждый из этих конвейеров может оказаться узким местом, поэтому увеличение производительности приложений ASP.NET фактически подразумевает увеличение производительности не только *вашего* кода, но и всех конвейеров.

Обсуждая способы увеличения производительности приложений ASP.NET, нужно смотреть дальше, за границы приложения, и исследовать все факторы, оказывающие влияние на общую производительность веб-приложения, которая складывается из производительности:

- прикладного кода;
- окружения ASP.NET;
- системного окружения (в большинстве случаев IIS);
- сети;
- клиентской части приложения (не рассматривается в этой книге).

В этой главе мы коротко обсудим инструменты измерения производительности веб-приложений и исследуем различные способы

увеличения производительности каждой из вышеупомянутых составляющих, из которых складывается общая производительность веб-приложения. Ближе к концу главы мы рассмотрим значимость способности веб-приложений к масштабированию и как узнаем, как избежать известных ловушек масштабирования.

Измерение производительности веб-приложений

Прежде чем пытаться что-то изменить в своем веб-приложении, необходимо узнать, достаточно ли быстро оно работает, то есть, соответствует ли требованиям, обозначенным в соглашении об обслуживании (Service Level Agreement, SLA)? Изменяется ли его производительность под нагрузкой? Наблюдаются ли какие-то типичные проблемы, которые можно устранить? Чтобы ответить на эти и другие вопросы, необходимо воспользоваться инструментами тестирования и мониторинга, которые помогут выявить узкие места в нашем веб-приложении.

В главе 2 мы познакомились с некоторыми универсальными инструментами анализа, помогающими выявлять проблемы производительности, такие как профилировщики Visual Studio и ANTS, однако существуют и другие инструменты тестирования и исследования «веб-части» приложений.

Это лишь краткое введение в область исследования производительности веб-приложений. более полное описание и рекомендации, как планировать, тестировать и анализировать производительность веб-приложений можно найти в статье «Performance Testing Guidance for Web Applications» на сайте MSDN (http://msdn.microsoft.com/library/bb924375).

Тестирование производительности и нагрузочное тестирование веб-приложений в среде Visual Studio

В числе средств тестирования, входящих в состав Visual Studio Ultimate, имеется инструмент тестирования веб-приложений, позволяющий оценить их время отклика и пропускную способность. С помощью этого инструмента можно записать все HTTP-запросы и ответы, генерируемые при работе с веб-приложением, как показано

на рис. 11.1. (Этот инструмент поддерживается только браузером Internet Explorer.)

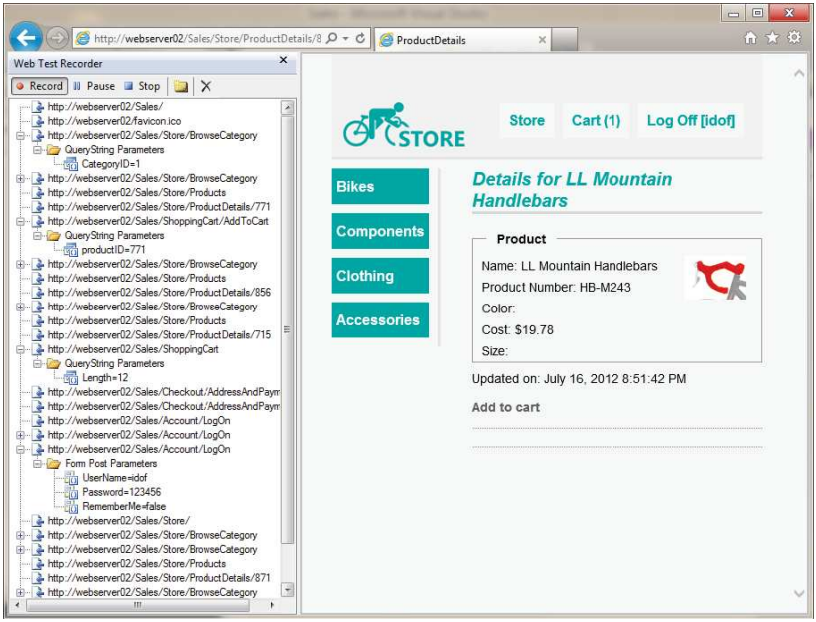

Рис. 11.1. Запись информации о работе веб-приложения с помощью Web Test Recorder.

Собрав информацию, ее можно использовать для исследования производительности веб-приложения, а также убедиться в правильной его работе, сопоставляя вновь получаемые ответы с ранее записанными ответами.

Описываемый инструмент тестирования веб-приложений позволяет настраивать поток выполнения тестирования. Вы можете изменять порядок следования запросов, добавлять новые запросы, вставлять циклы и условия, изменять заголовки и содержимое запросов, устанавливать правила проверки ответов и даже изменять тест целиком, преобразуя его в код и изменяя его вручную.

Инструмент тестирования веб-приложений ценен сам по себе, но чтобы исследовать производительность веб-приложения под нагрузкой, его следует использовать в комбинации с инструментом нагрузочного тестирования, также входящего в состав Visual Studio. Этот инструмент дает возможность сымитировать нагрузку на систему, создаваемую множеством пользователей, одновременно выполняющих

различные операции, и исследовать поведение системы, собирая различную информацию о производительности, например, из счетчиков производительности и журналов событий.

> **Внимание.** *Весьма желательно, чтобы нагрузочное тестирование производилось не на общедоступном веб-сайте, а на вашем собственном. Нагрузочное тестирование общедоступного веб-сайта может быть воспринято как атака вида «отказ в обслуживании» (Denial-Of-Service, DOS) с последующей блокировкой доступа к сайту для вашего компьютера или даже целой локальной сети, где вы находитесь.*

Совмещая нагрузочное тестирование с записью получаемой при этом информации, можно сымитировать ситуацию одновременной работы десятков и даже сотен пользователей, обращающихся к разным веб-страницам с разными параметрами в каждом запросе.

Чтобы сымитировать работу сотен пользователей, желательно использовать *тестовые агенты* (test agents). Тестовые агенты – это компьютеры, которые принимают инструкции по проведению тестирования от контроллера (управляющего компьютера), выполняют требуемые тесты, и отправляют результаты обратно контроллеру. Использование тестовых агентов позволяет уменьшить нагрузку на компьютер, выполняющий тестирование (не на тот, что подвергается испытаниям), потому что единственный компьютер, имитирующий деятельность сотен пользователей, может страдать от значительного падения производительности и давать искаженные результаты тестирования.

В ходе нагрузочного тестирования мы можем контролировать различные счетчики производительности, отражающие поведение приложения под нагрузкой, например, проверяя постановку запросов в очередь в ASP.NET, увеличение времени обслуживания запросов с течением времени, и превышение времени допустимого времени обработки запросов из-за ошибок в настройках.

Выполняя нагрузочное тестирование с различными параметрами, такими как различное количество одновременно работающих пользователей или различные типы сетей (быстрые/медленные), можно узнать очень многое о работе веб-приложения под нагрузкой, и на основе анализа записанной информации выработать решения по увеличению общей производительности.

Инструменты мониторинга HTTP

Инструменты мониторинга сети, способные перехватывать HTTP-пакеты, такие как Wireshark, NetMon, HTTP Analyzer и Fiddler, могут

помочь в выявлении проблем, связанных с передачей HTTP-запросов и ответов между веб-приложением и клиентом. С их помощью можно исследовать самые разные аспекты, оказывающие влияние на производительность веб-приложения. Например.

- **Корректное использование кеша браузера.** Просматривая HTTP-трафик, можно определить, какие ответы возвращаются с заголовками, препятствующими кешированию, и отправляются ли запросы на получение информации, которая уже должна находиться в кеше.
- **Количество и размеры сообщений.** Инструменты мониторинга позволяют увидеть каждый запрос и ответ, узнать время, затраченное на каждое сообщение и его размер, что дает возможность выявить запросы, отправляемые слишком часто, большие запросы и ответы, а также запросы, которые обрабатываются слишком долго.
- **Применение сжатия.** Просматривая запросы и ответы можно выявить запросы, отправляемые с заголовком *Accept-Encoding*, позволяющим применять GZip-сжатие, и узнать, возвращает ли веб-сервер ответы, сжатые соответствующим образом.
- **Синхронизация взаимодействий.** Некоторые инструменты мониторинга HTTP-трафика способны отображать хронологию поступления и обработки запросов, благодаря чему можно узнать, имеет ли клиентское приложение возможность отправлять сразу множество запросов, или вынуждено синхронизировать их отправку из-за нехватки исходящих соединений. Например, используя этот прием можно выяснить, сколько параллельных соединений может открыть браузер с конкретным сервером, или проверить, действует ли в клиентском приложении для .NET ограничение количества соединений, устанавливаемое классом `System.Net.ServicePointManager` и равное по умолчанию двум.

Некоторые инструменты, такие как Fiddler, могут также экспортировать записанный трафик в формате, доступном для инструмента тестирования производительности в Visual Studio, благодаря чему полученные данные можно использовать для организации нагрузочного тестирования с целью проверки веб-приложений, которые могут вызываться из клиентских приложений и браузеров, отличных от Internet Explorer. Например, можно записать HTTP-трафик взаимодействий со службой WCF из клиентского приложения для

.NET, экспортировать его и с помощью инструмента нагрузочного тестирования провести нагрузочное тестирование своей WCF-службы.

Инструменты анализа веб-взаимодействий

Для выявления проблем производительности можно также использовать инструменты анализа взаимодействий с веб-приложением, такие как Yahoo! YSlow и Google Page Speed. Инструменты анализа веб-взаимодействий не просто анализируют трафик, отыскивая заголовки, препятствующие кешированию и несжатые ответы. Они также анализируют сами HTML-страницы, выявляя проблемы, влияющие на скорость загрузки и отображения страниц, такие как:

- наличие больших и сложных структур HTML, влияющих на время отображения страницы;
- наличие содержимого в формате HTML, CSS и JavaScript, объем которого можно уменьшить, применяя приемы минификации (minification);
- наличие больших изображений, разрешение которых можно уменьшить, чтобы уменьшить их размер и обеспечить более удачное соответствие масштабу HTML-страницы;
- наличие кода на JavaScript, выполняемого после, а не во время загрузки страницы, что замедляет отображение страницы.

Увеличение производительности веб-сервера

Существует множество способов увеличения производительности приложений ASP.NET. Некоторые из них с успехом могут применяться как к обычным приложениям, так и к приложениям ASP.NET. Например, независимые асинхронные операции можно производить в параллельных потоках выполнения. Однако некоторые приемы непосредственно связаны с особенностями разработки приложений для ASP.NET, будь то приложения на основе WebForm или ASP.NET MVC Controller. Они способны обеспечить более полное использование ресурсов сервера, позволяя приложениям работать быстрее и обрабатывать большее количество параллельных запросов.

Кеширование часто используемых объектов

Обработка запросов в веб-приложениях часто связана с извлечением данных, обычно из удаленных источников, таких как базы данных или веб-службы. Выборка данных – довольно дорогостоящая операция, нередко связанная с необходимостью ожидания ответа. Вместо извлечения данных для каждой отдельной операции, их можно извлечь сразу все и сохранить в памяти, применив какой-либо механизм кеширования. После этого, для обработки вновь поступающих запросов, данные можно извлекать из кеша, не обращаясь к удаленному источнику. Алгоритм кеширования часто описывается следующим образом:

1. Если данные уже присутствуют в кеше, используются эти данные.
2. Иначе:
   a. Извлечь данные.
   b. Сохранить в кеше.
   c. Использовать.

Внимание. *Так как к одному и тому же объекту в кеше может обратиться сразу несколько запросов из разных потоков выполнения, необходимо предусмотреть непротиворечивое изменение объекта в кеше – либо интерпретировать его как неизменяемый (в этом случае, чтобы изменить объект в кеше, необходимо создать его копию, выполнить изменения в копии и сохранить измененную копию в кеше), либо использовать механизмы блокировки, чтобы исключить конфликты между разными потоками выполнения.*

Многие разработчики используют в роли кеша коллекцию `Application`, потому что она обеспечивает кеширование в памяти, доступна всем пользователям из всех сеансов. Работать с коллекцией `Application` очень просто:

```
Application["listOfCountries"] = countries; // Сохранить значение в коллекции
countries = (IEnumerable<string>)Application["listOfCountries"]; // Получить
```

При использовании коллекции `Application`, сохраняемые объекты постепенно накапливаются в памяти, что может привести к ее исчерпанию и вызвать необходимость использования файла подкачки или даже вызвать ошибку нехватки памяти. Поэтому ASP.NET предоставляет специальный механизм кеширования, поддерживающий средства управления объектами в кеше и удаляющий неиспользуемые объекты из кеша при нехватке памяти.

Кеширование в ASP.NET доступно через класс `Cache`, реализующий обширный механизм кеширования, который помимо возможности хранить объекты также позволяет.

- Определять предельное время хранения объектов в кеше, указывая либо значение типа `TimeSpan` (продолжительность), либо значение типа `DateTime` (конкретные дата и время). По истечении времени хранения объекты будут удаляться из кеша автоматически.
- Определять приоритеты кешируемых объектов. При нехватке памяти, когда требуется освободить место в кеше, наличие приоритетов помогает механизму кеша решить, какие из объектов «менее важны».
- Определять правила проверки допустимости хранения объектов в кеш, добавляя зависимости, такие как зависимости от SQL. Например, если кешируемый объект был получен в результате SQL-запроса, можно установить зависимость объекта от SQL, чтобы изменения в базе данных, влияющие на результат запроса, делали объект в кеше недействительным.
- Присоединять к объектам в кеше функции обратного вызова, которые должны вызываться при удалении объектов из кеша. Использование функций обратного вызова позволит своевременно обновлять данные в кеше, как только истекает время их хранения или они становятся недействительными.

Добавление элементов в кеш выполняется так же, как добавление элементов в словарь:

```
Cache["listOfCountries"] = listOfCountries;
```

При добавлении в кеш таким способом, элемент получит приоритет по умолчанию `Normal` и не будет иметь предельного времени хранения или зависимостей. Например, чтобы добавить элемент в кеш, который должен хранить некоторый интервал времени, используйте метод `Insert`:

```
Cache.Insert("products", productsList,
Cache.NoAbsoluteExpiration, TimeSpan.FromMinutes(60), dependencies: null);
```

> **Примечание.** *Класс* `Cache` *также предоставляет метод* `Add`*. В отличие от метода* `Insert`*, метод* `Add` *возбуждает исключение, если кеш уже содержит элемент с тем же самым ключом.*

Парадигма доступа к кешу с использованием класса `Cache` реализуется следующим образом:

```
object retrievedObject = null;

retrievedObject = Cache[“theKey”];
if (retrievedObject == null) {
    // Отыскать данные (в базе данных, веб-службе и т. д.)
    object originalData = null;
    ...
    // сохранить вновь извлеченные данные в кеше
    Cache[“theKey”] = originalData;
    retrievedObject = originalData;
}
// использовать извлеченный объект
// (либо из кеша,либо только что добавленный в кеш)
...
```

Обратите внимание, что в первой строке в этом примере извлечение объекта из кеша выполняется без предварительной проверки его наличия там. Это обусловлено тем, что объекты могут удаляться из кеша в любые моменты времени другими запросами или самим механизмом кеширования, то есть объект может быть удален между проверкой его существования и операцией извлечения.

Использование асинхронных страниц, модулей и контроллеров

Когда среда выполнения ASP.NET получает запрос от IIS, она передает его пулу потоков выполнения, после чего одному из рабочих потоков поручается выполнить задачу по обработке запроса, какой бы она ни была – простой HTTP-обработчик запроса, страница в приложении ASP.NET WebForm или контроллер в приложении ASP.NET MVC.

Поскольку количество рабочих потоков ограничено (определяется значением `maxWorkerThreads` в разделе `processModel` в файле *web.config*), среда ASP.NET также имеет ограниченное количество потоков для одновременной обработки запросов.

Максимальное количество потоков обычно достаточно велико для небольших и средних веб-приложений, обрабатывающих одновременно не более нескольких десятков запросов. Однако, если вашему приложению требуется обрабатывать одновременно сотни запросов, продолжите чтение этого раздела.

Ограниченное количество одновременно обрабатываемых запросов побуждает разработчиков уменьшать время обработки каждого запроса, но как быть, если в процессе обработки запроса требуется выполнить операции ввода/вывода, например, вызвать веб-службу

или дождаться ответа на запрос к базе данных? В этом случае время выполнения запроса сильно зависит от времени получения информации от удаленного процесса, и в течение всего этого времени рабочий поток остается связанным с обрабатываемым запросом и не может заняться обработкой другого запроса.

В конечном счете, когда количество одновременно обрабатываемых запросов превысит количество рабочих потоков, вновь поступающие запросы будут помещаться в специальную очередь ожидания. Когда количество запросов в очереди превысит установленный порог, в ответ на входящие запросы будет возвращаться ответ HTTP 503 («service unavailable» – «служба недоступна»).

> **Примечание.** *Ограничения на объем пула потоков выполнения и очереди запросов в приложениях ASP.NET определяются в разделе* `processModel`, *в файле web.config и отчасти управляются атрибутом* `processModel→autoConfig`.

В современных веб-приложениях, где операции ввода/вывода (обращения к веб-службам, запросы к базам данных, извлечение данных из сетевых хранилищ файлов и так далее) являются неотъемлемой их частью, такое поведение часто приводит к ситуации, когда большая часть потоков выполнения ожидает завершения ввода/вывода и лишь несколько потоков выполняют вычислительные операции. В результате вычислительные мощности сервера используются не полностью и при этом не могут быть задействованы для обработки других запросов, потому что не остается свободных потоков выполнения.

В веб-приложениях, где обработка многих запросов связана с получением данных от веб-служб или из баз данных, часто можно наблюдать низкое использование процессора, даже при обслуживании большого количества пользователей. Выяснить использование процессора в своем веб-приложении можно с помощью счетчиков производительности, например, с помощью комбинации счетчиков **Processor\% CPU Utilization** (Процессор\% загруженности процессора), **ASP.NET Applications\Requests/Sec** (Приложения ASP.NET\Запросов/сек) и **ASP.NET\Requests Queued** (ASP.NET\Запросов в очереди).

Если для обработки некоторые из ваших запросов может потребоваться выполнять продолжительные операции ввода/вывода, необязательно удерживать рабочий поток выполнения до полного завершения обработки. В ASP.NET имеется возможность создавать асинхронные страницы, контроллеры, обработчики и модули, позволяющие возвращать рабочие потоки обратно в пул на время ожида-

ния завершения операции ввода/вывода и захватывать их из пула для завершения обработки запросов. С точки зрения конечного пользователя, страница все еще будет выглядеть, как требующая некоторого времени для загрузки, потому что сервер возвращает ответ только после обработки запроса.

Используя асинхронный способ обработки запросов, требующих большого объема ввода/вывода, можно увеличить количество рабочих потоков, доступных для обработки запросов, требующих решения вычислительных задач, и тем самым повысить использование процессора (или процессоров), а также избежать попадания запросов в очередь ожидания.

Создание асинхронной страницы

Если у вас имеется приложение на основе ASP.NET Web Forms и вам требуется создать асинхронную страницу, для начала отметьте страницу как асинхронную:

```
<%@ Page Async = "true" ...
```

Затем создайте новый объект `PageAsyncTask` и передайте ему делегаты, осуществляющие запуск операции, обработку ее результатов и прерывания операции по тайм-ауту. После создания объекта `PageAsyncTask` вызовите метод `Page.RegisterAsyncTask`, чтобы запустить асинхронную операцию.

Следующий фрагмент кода демонстрирует, как запустить продолжительный SQL-запрос с помощью `PageAsyncTask`:

```
public partial class MyAsyncPage : System.Web.UI.Page {
    private SqlConnection _sqlConnection;
    private SqlCommand _sqlCommand;
    private SqlDataReader _sqlReader;

    IAsyncResult BeginAsyncOp(object sender, EventArgs e,
                              AsyncCallback cb, object state) {
        // Эта часть кода будет выполнена в рабочем потоке
        // выполнения, поэтому в этом методе не следует выполнять
        // продолжительные операции
        _sqlCommand = CreateSqlCommand(_sqlConnection);
        return _sqlCommand.BeginExecuteReader(cb, state);
    }

    void EndAsyncOp(IAsyncResult asyncResult) {
        _sqlReader = _sqlCommand.EndExecuteReader(asyncResult);
        // Прочитать данные и наполнить страницу содержимым
        ...
```

```
    }

    void TimeoutAsyncOp(IAsyncResult asyncResult) {
        _sqlReader = _sqlCommand.EndExecuteReader(asyncResult);
        // Прочитать данные и наполнить страницу содержимым
        ...
    }

    public override void Dispose() {
        if (_sqlConnection ! = null) {
            _sqlConnection.Close();
        }
        base.Dispose();
    }

    protected void btnClick_Click(object sender, EventArgs e) {
        PageAsyncTask task = new PageAsyncTask(
            new BeginEventHandler(BeginAsyncOp),
            new EndEventHandler(EndAsyncOp),
            new EndEventHandler(TimeoutAsyncOp),
            state:null);
        RegisterAsyncTask(task);
    }
}
```

Другой способ создания асинхронных страниц заключается в применении *событий завершения* (completion events), подобных тем, что создаются при использовании веб-служб и прокси-объектов в WCF-службах:

```
public partial class MyAsyncPage2 : System.Web.UI.Page {
    protected void btnGetData_Click(object sender, EventArgs e) {
        Services.MyService serviceProxy = new Services.MyService();
        // Подключить к событию xxCompleted службы
        serviceProxy.GetDataCompleted + = new
            Services.GetDataCompletedEventHandler(GetData_Completed);
        // Использовать асинхроннный вызов службы, выполняемый в
        // потоке, осуществляющем ввод/вывод
        serviceProxy.GetDataAsync();
    }

    void GetData_Completed (object sender,
                            Services.GetDataCompletedEventArgs e){
        // Извлечь результаты из аргументов события и
        // наполнить страницу содержимым
    }
}
```

В примере выше страница также помечается как асинхронная (`Async`), но в данном случае нет необходимости создавать объект

PageAsyncTask, так как страница автоматически получит уведомление при вызове метода xxAsync, после того, как будет сгенерировано событие xxCompleted.

> **Примечание.** *Когда страница объявляется асинхронной, ASP.NET изменяет страницу и добавляет к ней реализацию интерфейса* IHttpAsyncHandler *вместо* IHttpHandler. *Если вам потребуется создать собственный обобщенный асинхронный обработчик HTTP-запросов, создайте класс такого обработчика и реализуйте в нем интерфейс* IHttpAsyncHandler.

Создание асинхронного контроллера

Контроллеры в ASP.NET MVC также могут создаваться как асинхронные, если выполняют продолжительные операции ввода/вывод. Чтобы создать асинхронный контроллер требуется выполнить следующие шаги:

1. Создать класс контроллера, наследующий тип AsyncController.
2. Реализовать методы xxAsync и xxCompleted для каждой синхронной операции, где xx – это имя операции.
3. В методе xxAsync вызвать метод AsyncManager.OutstandingOperations.Increment, передав ему количество выполняемых асинхронных операций.
4. В коде, выполняемом по завершении каждой асинхронной операции, вызвать метод AsyncManager.OutstandingOperations.Decrement, чтобы сообщить об окончании операции.

Например, ниже представлен контроллер, выполняющий асинхронную операцию Index, которая вызывает службу, возвращающую данные для отображения в представлении:

```
public class MyController : AsyncController {
    public void IndexAsync() {
        // Известить AsyncManager, что выполняется единственная операция
        AsyncManager.OutstandingOperations.Increment();
        MyService serviceProxy = new MyService();

        // Зарегистрировать событие завершения
        serviceProxy.GetDataCompleted + = (sender, e) = > {
            AsyncManager.Parameters[“result”] = e.Value;
            AsyncManager.OutstandingOperations.Decrement();
        };
        serviceProxy.GetHeadlinesAsync();
    }

    public ActionResult IndexCompleted(MyData result) {
```

```
        return View("Index", new MyViewModel { TheData = result });
    }
}
```

Настройка окружения ASP.NET

Помимо нашего кода, каждый входящий запрос и исходящий ответ обрабатываются компонентами ASP.NET. Некоторые механизмы ASP.NET, такие как `ViewState`, были предусмотрены для нужд разработки, но они могут влиять на общую производительность приложения. При тонкой настройке приложений ASP.NET рекомендуется изменить поведение по умолчанию некоторых из этих механизмов, хотя иногда эти изменения могут потребовать изменения программного кода приложения.

Отключение механизмов трассировки и отладки в ASP.NET

Механизм трассировки ASP.NET Tracing позволяет разработчикам получать диагностическую информацию для той или иной страницы, например, время выполнения и ее путь, состояние сеанса и список HTTP-заголовков.

Хотя механизм трассировки позволяет получать бесценную информацию в процессе разработки и отладки приложений ASP.NET, он оказывает отрицательное влияние на общую производительность приложения, выполняя сбор данных о каждом запросе. То есть, если поддержка трассировки была включена на этапе разработки, отключайте ее перед развертыванием своего приложения в эксплуатационной среде, для чего достаточно просто изменить параметр `trace` в файле *web.config*:

```
<configuration>
    <system.web>
        <trace enabled = "false"/>
    </system.web>
</configuration>
```

> **Примечание.** *По умолчанию, если явно не указано иное, трассировка отключена (*`enabled = "false"`*), поэтому, отключить трассировку можно также простым удалением параметра* `trace` *из файла web.config.*

Когда создается новое веб-приложение на основе ASP.NET, в файл *web.config* автоматически добавляется конфигурационный раздел

system.web→compilation с атрибутом debug, установленным в значение true:

```
<configuration>
    <system.web>
        <compilation debug = "true" targetFramework = "4.5" />
    </system.web>
</configuration>
```

Примечание. *Этот раздел добавляется автоматически в Visual Studio 2012 или 2010. В предыдущих версиях Visual Studio параметру* debug *по умолчанию присваивается значение* false*, и когда разработчик впервые попытается отладить приложение, на экран выводится диалог, спрашивающий разрешения изменить значение этого параметра на* true.

Проблема с этим параметром в том, что разработчики часто забывают установить его в значение false при развертывании на действующем веб-сервере, или специально оставляют в значении true, чтобы иметь возможность получения более подробной информации в случае появления исключения. Это может приводить к нескольким проблемам, связанным с производительностью.

- Сценарии, загружаемые с использованием обработчика WebResources.axd, например, когда в страницах используются проверки допустимости, не будут кешироваться браузером. Когда флаг debug установлен в значение false, ответы от этого обработчика будут снабжаться заголовками, допускающими возможность кеширования, позволяя браузерам сохранять результаты в кеше для использования в будущем.
- Когда параметр debug имеет значение false, тайм-аут на обработку запросов не устанавливается. Хотя это очень удобно при отладке, такое поведение нежелательно в промышленном окружении, так как такие запросы могут привести к тому, что сервер окажется не в состоянии обслужить другие запросы, или даже вызвать чрезмерную нагрузку на процессор, увеличить потребление памяти и породить другие проблемы, связанные с распределением ресурсов.
- Установка флага debug в значение false позволит среде выполнения ASP.NET определять тайм-ауты для обработки запросов в соответствии с настройками httpRuntime→executionTimeout (значение по умолчанию 110 секунд).
- Когда параметр debug имеет значение true, JIT-компилятор не будет оптимизировать код. Эти оптимизации являются одним из важнейших преимуществ среды .NET и могут обеспечить

значительное увеличение производительности приложения ASP.NET без изменения его кода. Установка параметра `debug` в значение `false` позволит JIT-компилятору выполнить свою работу, и сделает ваше приложение более быстрым и эффективным.

- Когда параметр `debug` имеет значение `true`, процедура компиляции не использует пакетный режим. В этом случае для каждой страницы и каждого элемента управления будет создаваться отдельная сборка, в результате чего приложению потребуется загружать десятки и даже сотни сборок в процессе выполнения; загрузка такого большого количества сборок может вызвать ошибки нехватки памяти из-за фрагментации адресного пространства. Когда отладочный режим выключен, используется пакетная компиляция, когда генерируется единственная сборка для элементов управления и несколько сборок для страниц (страницы группируются в сборки по используемым элементам управления).

Изменить этот параметр очень просто: достаточно полностью удалить атрибут `debug` из файла конфигурации или присвоить ему значение `false`:

```
<configuration>
    <system.web>
        <compilation debug = "false" targetFramework = "4.5" />
    </system.web>
</configuration>
```

Если вы боитесь, что забудете изменить этот параметр при развертывании приложения на действующем сервере, можно заставить *все* приложения ASP.NET на сервере игнорировать параметр `debug`, следующий фрагмент в файл *machine.config* на сервере:

```
<configuration>
    <system.web>
        <deployment retail = "true"/>
    </system.web>
</configuration>
```

Отключение механизма ViewState

Механизм сохранения состояния представления ViewState используется в приложениях ASP.NET Web Forms для сохранения состояния страницы в отображаемую разметку HTML (приложения ASP.NET MVC не используют этот механизм). Механизм ViewState позволяет

ASP.NET сохранять состояние страницы для отправки его обратно пользователем. Данные сохраняются в формате HTML, шифруются (по умолчанию шифрование отключено), кодируются в строку Base64 и запоминаются в скрытом поле. Когда пользователь отправляет страницу обратно, содержимое поля декодируется и преобразуется обратно в ассоциативный массив. Многие средства управления сервером используют механизм ViewState для сохранения собственной информации, например, значений своих свойств.

Это очень удобный и очень мощный механизм, но он добавляет в страницу дополнительное содержимое – строку в кодировке Base64, которое может значительно увеличивать объем ответа. Например, страница, содержащая единственный компонент `GridView` с возможностью постраничного просмотра списка из 800 клиентов, генерирует разметку HTML размером 17 Кбайт, из которых 6 Кбайт приходится на скрытое поле с информацией о состоянии представления, потому что элементы `GridView` сохраняют свои исходные данные в этом поле. Кроме того, использование поддержки механизма ViewState требует выполнять сериализацию и десериализацию состояния представления для каждого запроса, что влечет дополнительные накладные расходы на обработку страницы.

> **Совет.** *Увеличение объема страниц при использовании механизма ViewState обычно остается незамеченным для клиентов, обращающихся к веб-серверу в локальной сети. Это объясняется тем, что локальные сети обычно отличаются высокой скоростью передачи и транспортировка очень больших страниц в таких сетях составляет миллисекунды (в оптимально построенной гигабитной локальной сети пропускная способность может достигать ~40–100 Мбайт/сек, в зависимости от используемого аппаратного обеспечения). Однако в более медленных глобальных сетях, таких как Интернет, увеличение размеров страниц становится более заметным.*

Если приложению не требуется использовать этот механизм, его лучше отключить. Отключить механизм ViewState можно в файле *web.config*, для всего приложения целиком:

```
<system.web>
    <pages EnableViewState = "false"/>
</system.web>
```

Если его нельзя отключить для всего приложения, можно запретить использование механизма ViewState в отдельной странице и всех ее элементах управления:

```
<%@ Page EnableViewState = "false" ... %>
```

Имеется также возможность отключить поддержку ViewState в отдельных элементах управления:

```
<asp:GridView ID = "gdvCustomers"
              runat = "server"
              DataSourceID = "mySqlDataSource"
              AllowPaging = "True"
              EnableViewState = "false"/>
```

До версии ASP.NET 4 отключение поддержки механизма ViewState в странице делало невозможным ее включение в отдельных элементах управления на странице. Начиная с версии ASP.NET 4 такая возможность была добавлена. Достигается это с помощью свойства `ViewStateMode`. Например, следующий код отключает поддержку механизма ViewState для всей страницы, кроме элемента управления `GridView`:

```
<%@ Page EnableViewState = "true" ViewStateMode = "Disabled" ... %>

<asp:GridView ID = "gdvCustomers"
              runat = "server"
              DataSourceID = "mySqlDataSource"
              AllowPaging = "True"
              ViewStateMode = "Enabled"/>
```

> **Внимание.** *Установка атрибута* `EnableViewState` *в значение false имеет более высокий приоритет, чем атрибуты* `ViewStateMode`. *То есть, если вы предполагаете использовать атрибуты* `ViewStateMode`, *обязательно установите атрибут* `EnableViewState` *в значение* `true` *или просто удалите его (по умолчанию он получает значение* `true`*).*

Кеш вывода на стороне сервера

Страницы ASP.NET считаются динамическими, в смысле информационного наполнения, тем не менее, зачастую такое динамическое содержимое страниц не меняется с течением времени. Например, страница может принимать идентификатор продукта и возвращать разметку HTML с его описанием. Сама по себе страница является динамической, потому что может возвращать разное описание для разных продуктов, но описание определенного продукта не изменяется, по крайней мере, пока оно не изменится в базе данных.

В продолжение примера с продуктом: чтобы предотвратить выполнение повторных запросов к базе данных всякий раз, когда пользователи запрашивают описание продукта, можно сохранить описание в локальном кеше и обеспечить более быстрое его извлечение, но при

этом все еще необходимо генерировать разметку HTML в ответ на каждый запрос. Вместо кеширования исходных данных, ASP.NET предоставляет другую возможность – механизм ASP.NET Output Cache, где сохраняется сама разметка HTML.

Благодаря этому механизму появляется возможность сохранять разметку HTML, которая автоматически будет возвращаться в ответ на последующие запросы, минуя этап выполнения программного кода, генерирующего страницу. Кеш вывода в ASP.NET поддерживается и для приложений на основе Web Forms, где в кеше сохраняются страницы, и для приложений на основе ASP.NET MVC, где сохраняются результаты выполнения операций контроллера.

Например, следующий код использует кеш вывода для сохранения представления, возвращаемого операцией контроллера ASP.NET MVC на срок до 30 секунд:

```
public class ProductController : Controller {
    [OutputCache(Duration = 30)]
    public ActionResult Index() {
        return View();
    }
}
```

Если бы операция `Index` в примере выше принимала параметр с идентификатором продукта и возвращала представление с информацией об определенном продукте, нам потребовалось бы кешировать несколько версий вывода для разных идентификаторов. Поэтому кеш вывода поддерживает не только возможность сохранения единственной версии выходных данных, но и разных версий данных, генерируемых одной и той же операцией, вызванной с разными параметрами. Следующий фрагмент демонстрирует, как изменить объявление операции, чтобы механизм кеширования учитывал значение параметра, передаваемое методу:

```
public class ProductController : Controller {
    [OutputCache(Duration = 30, VaryByParam = "id")]
    public ActionResult Index(int id) {
        // Извлечь информацию о продукте и настроить модель
        // соответственно
        ...
        return View();
    }
}
```

Примечание. *В дополнение к параметрам запроса механизм кеширования вывода может также учитывать HTTP-заголовки запроса, такие как*

`Accept-Encoding` *и* `Accept-Language`. *Например, если метод контроллера может возвращать содержимое на разных языках, в зависимости от HTTP-заголовка* `Accept-Language`, *вы можете настроить учет этого заголовка и сохранять в кеше разные версии вывода на разных языках.*

Если одни и те же настройки кеширования применяются к разным страницам или операциям, можно создать профиль кеширования и использовать его, вместо повторения настроек снова и снова. Профили кеширования создаются в файле *web.config*, в разделе `system.web→ caching`. Например, следующий фрагмент объявляет профиль кеширования, который предполагается использовать вместе с разными страницами:

```
<system.web>
    <caching>
        <outputCacheSettings>
            <outputCacheProfiles>
                <add name = "CacheFor30Seconds"
                     duration = "30"
                     varyByParam = "id"/>
            </outputCacheProfiles>
        </outputCacheSettings>
    </caching>
</system.web>
```

Теперь этот профиль можно применить к операции `Index`:

```
public class ProductController : Controller {
    [OutputCache(CacheProfile = "CacheFor30Seconds")]
    public ActionResult Index(int id) {
        // Извлечь информацию о продукте и настроить модель
        // соответственно
        ...
        return View();
    }
}
```

Этот же профиль кеширования можно использовать в веб-форме ASP.NET, указав его в директиве `OutputCache`:

```
<%@ OutputCache CacheProfile = "CacheEntityFor30Seconds" %>
```

Примечание. *По умолчанию механизм кеширования вывода в ASP.NET сохраняет информацию в памяти сервера. Однако начиная с версии ASP.NET 4 появилась возможность создать и использовать свой объект-провайдер. Например, можно создать провайдера, который будет сохранять кешированные данные на диске.*

Предварительная компиляция приложений ASP.NET

Когда выполняется компиляция проекта веб-приложения ASP.NET, весь код помещается в единственную сборку. Однако веб-страницы (*.aspx*) и элементы управления (*.ascx*) не компилируются, и развертываются на сервере в своем исходном виде. При первом запуске веб-приложения (когда поступает первый запрос), ASP.NET динамически компилирует веб-страницы и элементы управления, и помещает скомпилированные файлы в папку *ASP.NET Temporary Files*. Такая динамическая компиляция увеличивает время ответа на первый запрос, вызывая у пользователя ощущение, что веб-сайт загружается очень медленно.

Чтобы решить ту проблему, можно заранее скомпилировать веб-приложение, включая весь код, страницы и элементы управления, воспользовавшись инструментом компиляции ASP.NET (*Aspnet_compiler.exe*). Запуск инструмента компиляции на действующем сервере может заметно уменьшить задержку при обработке первого запроса. Для этого необходимо выполнить следующие шаги:

1. Откройте окно командной строки на сервере.
2. Перейдите в папку *%windir%\Microsoft.NET*.
3. Перейдите в папку *Framework* или *Framework64*, в соответствии с настройками пула веб-приложений на поддержку 32-разрядных или 64-разрядных приложений (в 32-разрядных операционных системах существует только папка *Framework*).
4. Перейдите в папку, соответствующую версии фреймворка .NET, используемой пулом приложений (v2.0.50727 или v4.0.30319).
5. Введите следующую команду, чтобы запустить компиляцию (замените *WebApplicationName* на виртуальный путь к своему приложению):

```
Aspnet_compiler.exe -v /WebApplicationName
```

Тонкая настройка модели процесса в ASP.NET

Когда поступает запрос к приложению ASP.NET, он передается рабочему потоку выполнения для обработки. Иногда код приложения может сам создать новый поток, например, чтобы обратиться к службе, уменьшая тем самым количество свободных рабочих потоков.

Чтобы предотвратить исчерпание пула потоков выполнения, ASP.NET автоматически корректирует некоторые настройки пула и применяет различные ограничения на количество запросов, которые могут обрабатываться одновременно. Эти настройки находятся в трех основных конфигурационных разделах: `system.web→processModel`, `system.web→httpRuntime` и `system.net→connectionManagement`.

> **Примечание.** *Разделы* `httpRuntime` *и* `connectionManagement` *можно определить в файле web.config приложения. Однако раздел* `processModel` *можно определить только в файле machine.config.*

Раздел `processModel` содержит настройки ограничений пула потоков выполнения, такие как минимальное и максимальное количество потоков, а раздел `httpRuntime` определяет ограничения, имеющие отношение к доступным потокам выполнения, такие как минимальное количество доступных потоков, необходимых для обработки входящих запросов. Раздел `connectionManagement` определяет максимальное количество исходящих HTTP-соединений на адрес.

Все настройки имеют значения по умолчанию, однако, так как некоторые из них выбраны слишком низкими, ASP.NET включает еще одну настройку, с именем `autoConfig`, корректирующую некоторые параметры для достижения более оптимальной производительности. Эта настройка, являющаяся частью раздела `processModel`, поддерживается, начиная с версии ASP.NET 2.0 и автоматически получает значение `true`.

Параметр `autoConfig` управляет регулировкой следующих настроек (значения по умолчанию, упомянутые ниже, получены из статьи KB821268 в Microsoft Knowledge Base по адресу: http://support.microsoft.com/?id=821268):

- `processModel→maxWorkerThreads`. Изменяет максимальное количество рабочих потоков с 20 потоков на ядро до 100 потоков на ядро.
- `processModel→maxIoThreads`. Изменяет максимальное количество потоков, выполняющих ввод/вывод, с 20 потоков на ядро до 100 потоков на ядро.
- `httpRuntime→minFreeThreads`. Изменяет минимальное количество доступных потоков, необходимых для обработки новых запросов с 8 потоков на ядро до 88 потоков на ядро.
- `httpRuntime→minLocalFreeThreads`. Изменяет минимальное количество доступных потоков, необходимых для обработки новых локальных запросов (поступающих с локального компьютера) с 4 потоков на ядро до 76 потоков на ядро.

- `connectionManagement→maxConnections`. Изменяет максимальное количество одновременно обрабатываемых соединений с 10 до 12 на ядро.

Хотя значения по умолчанию, описанные выше, корректируются для достижения оптимальной производительности, иногда бывает необходимо изменить их, чтобы добиться еще большей производительности, в зависимости от конкретных условий, в которых выполняется веб-приложение. Например, если приложение обращается к внешним службам, может потребоваться увеличить максимальное количество одновременно обрабатываемых соединений, чтобы большее количество запросов могло обращаться к внутренним службам. Следующий фрагмент демонстрирует, как увеличить максимальное количество соединений:

```
<configuration>
    <system.net>
        <connectionManagement>
            <add address = "*" maxconnection = "200" />
        </connectionManagement>
    </system.net>
</configuration>
```

В других ситуациях, например, когда после запуска веб-приложения получают большое количество запросов или время от времени испытывают внезапное увеличение количества запросов, может понадобиться изменить минимальное количество рабочих потоков (значение, которое вы укажете, во время выполнения будет умножено на количество ядер процессора). Чтобы изменить этот параметр, добавьте следующий фрагмент в файл *machine.config*:

```
<configuration>
    <system.web>
        <processModel autoConfig = "true" minWorkerThreads = "10"/>
    </system.web>
</configuration>
```

Прежде чем вы кинетесь увеличивать минимальное и максимальное количество потоков выполнения, подумайте о побочном эффекте этих изменений для вашего приложения: если разрешить одновременную обработку слишком большого количества запросов, это может привести к чрезмерной нагрузке на процессор и высокому потреблению памяти, и к увеличению вероятности аварийного завершения веб-приложения. Поэтому после изменения настроек необходимо выполнить нагрузочное тестирование, чтобы убедиться, что машина действительно способна выдержать большое количество запросов.

Настройка IIS

Веб-сервер IIS, как среда выполнения веб-приложения, имеет некоторое влияние на общую производительность. Например, чем короче конвейер обработки запросов в IIS, тем меньше кода будет выполняться и тем выше будет скорость работы. В IIS имеются механизмы, которые можно использовать для увеличения производительности приложения за счет снижения задержек и увеличения пропускной способности, а также некоторые механизмы, которые при правильной настройке могут увеличить общую производительность приложения.

Кеширование вывода

Мы уже знаем, что ASP.NET поддерживает собственный механизм кеширования вывода, зачем же использовать еще один подобный механизм в IIS? Ответ на этот вопрос прост: помимо страниц ASP.NET существуют и другие виды содержимого, которое можно кешировать. Например, мы можем кешировать часто запрашиваемые статические файлы изображений или вывод собственных обработчиков HTTP-запросов. Для этой цели как раз и можно использовать механизм кеширования, поддерживаемый веб-сервером IIS.

В IIS имеется два механизма кеширования: кеш в пространстве пользователя и кеш в пространстве ядра.

Кеширование в пространстве пользователя

Так же как ASP.NET, веб-сервер IIS способен кешировать ответы в памяти, чтобы ответы на последующие запросы могли отправляться из кеша в памяти, без обращения к статическим файлам на диске и без вызова программного кода на стороне сервера.

Чтобы настроить кеширование, откройте приложение IIS Manager, выберите свое веб-приложение, откройте настройку **Output Caching** (Кеширование вывода), щелкните на ссылке **Add…** (Добавить...) в панели **Actions** (Операции), чтобы добавить новое правило кеширования, или выберите существующее правило для редактирования.

Чтобы создать новое правило кеширования в пространстве пользователя, добавьте новое правило, введите расширение имен файлов, которые требуется кешировать, и отметьте флажок **User-mode caching** (Кеширование в пространстве пользователя) в диалоге **Add Cache Rule** (Добавить правило кеширования), как показано на рис. 11.2.

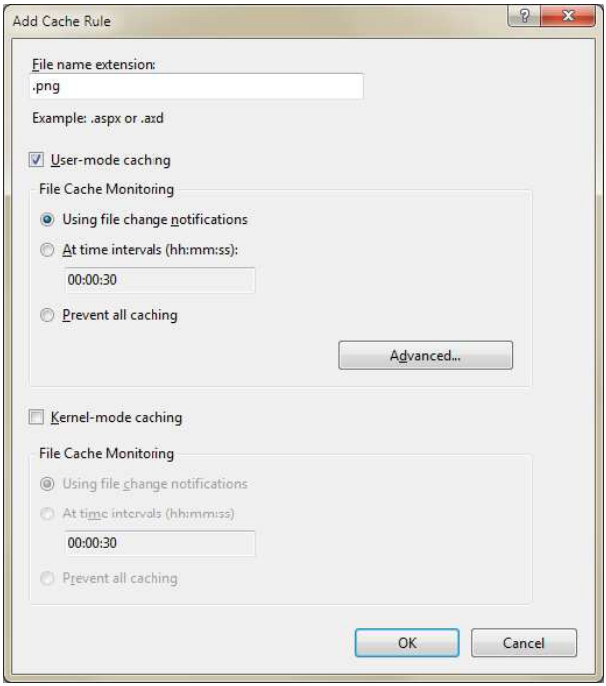

Рис. 11.2. Диалог Add Cache Rule (Добавить правило кеширования).

Отметив флажок, вы получите возможность выбирать, когда кешированный элемент будет удаляться из памяти, после обновления файла на диске или спустя некоторое время с момента кеширования. Для статических файлов больше подходит вариант удаления после обновления, тогда как определение интервала времени больше подходит для динамического содержимого. Щелкнув на кнопке **Advanced** (Дополнительно) можно получить доступ к настройкам, управляющим кешированием различных версий вывода (согласно параметрам в строке запроса или HTTP-заголовкам).

После добавления правила кеширования, настройки сохраняются в файле *web.config* приложения, в разделе `system.webServer→caching`. Например, для правила кеширования страниц *.aspx* на срок до 30 минут, с учетом HTTP-заголовка `Accept-Language`, будет сгенерирован следующий код в конфигурационном файле:

```
<system.webServer>
    <caching>
```

```
        <profiles>
            <add extension = ".aspx"
                 policy = "CacheForTimePeriod"
                 kernelCachePolicy = "DontCache"
                 duration = "00:00:30"
                 varyByHeaders = "Accept-Language" />
        </profiles>
    </caching>
</system.webServer>
```

Кеширование в пространстве ядра

В отличие от механизма кеширования в пространстве пользователя, сохраняющего информацию в памяти рабочего процесса IIS, механизм кеширования в пространстве ядра сохраняет информацию в памяти драйвера *HTTP.sys*. Кеширование в пространстве ядра обеспечивает более короткое время отклика, однако оно поддерживается не всегда. Например, кеширование в пространстве ядра нельзя использовать, когда запрос содержит строку с параметрами запроса или когда запрос не является анонимным запросом.

Настройка правил кеширования в пространстве ядра выполняется почти так же, как кеширование в пространстве пользователя. В диалоге настройки правила установите флажок **Kernel-mode caching** (Кеширование в пространстве ядра) и выберите желаемый способ кеширования.

В одном правиле допускается использовать оба режима кеширования, в пространстве ядра и в пространстве пользователя. В этом случае IIS будет сначала пытаться применить кеширование в пространстве ядра. Если попытка не увенчается успехом, например, когда запрос содержит строку с параметрами, применяется кеширование в пространстве пользователя.

> **Совет.** *Если выбран вариант кеширования на определенный интервал времени в обоих режима, в пространстве ядра и в пространстве пользователя, оба интервала должны совпадать, в противном случае в обоих режимах будет использоваться интервал, установленный для кеширования в режиме ядра.*

Настройка пула приложения

Параметры настройки пула приложения определяют, как IIS будет создавать и поддерживать рабочие процессы, в рамках которых будет выполняться наш код. В ходе установки IIS и ASP.NET создается несколько пулов приложений, в зависимости от установленной версии

.NET, к которым вы можете добавлять новые пулы, по мере развертывания дополнительных веб-приложений. При создании пула приложения, он получает некоторые параметры настройки со значениями по умолчанию, управляющие его поведением. Например, каждый пул получает тайм-аут простоя, после которого этот пул приложения останавливается.

Понимание значения некоторых из этих настроек может помочь вам настроить работу пула и тем самым более полно удовлетворить потребности приложения.

Перезапуск

Изменяя параметр настройки перезапуска можно управлять моментом, когда пул приложения будет перезапускать рабочий процесс. Например, можно организовать перезапуск рабочего процесса через каждые несколько часов или когда будет превышен некоторый предел занимаемой памяти. Если с течением времени веб-приложение начинает потреблять большие объемы памяти (например, для хранения объектов), увеличение количества перезапусков может помочь удерживать его производительность на высоком уровне. С другой стороны, если веб-приложение не проявляет никаких проблем в процессе работы, уменьшение количества перезапусков предотвратит потерю информации о состоянии.

> **Совет.** *Для проверки количества и частоты перезапусков пула приложения можно использовать счетчик производительности ASP.NET\Worker Process Restarts (ASP.NET\Перезапусков рабочего процесса). Если вы увидите слишком большое количество перезапусков без явной на то причины, попробуйте сопоставить полученные значения с потреблением памяти приложением и нагрузкой на процессор, потому что перезапуски могут быть обусловлены превышением других пределов, определяемых настройками пула приложения.*

Тайм-аут простоя

По умолчанию пул приложения прекращает работу через 20 минут простоя. Если такие перерывы а работе ожидаемы, например, когда все пользователи уходят на обед, попробуйте увеличить тайм-аут или даже вообще отменить его.

Привязка процессов к ядрам процессора

По умолчанию пул приложения настроен так, что может использовать все доступные ядра процессора. Если у вас имеется какой-либо

специализированный фоновый процесс, использующий все процессорное время, какое ему будет выделено, можете настроить привязку пула к определенным ядрам, освободив остальные для фонового процесса. Разумеется, при этом также потребуется настроить привязку фонового процесса к другим ядрам процессора, чтобы избежать конкуренции между ним и рабочим процессом за одни и те же ядра.

Веб-сад

По умолчанию пул приложения запускает один рабочий процесс, обслуживающий все запросы, поступающие приложению. Если рабочему процессу приходится одновременно обрабатывать несколько запросов, конкурирующих за обладание одним и тем же ресурсом, это может привести к задержкам при подготовке ответов. Например, если приложение использует патентованный механизм кеширования с блокировками, препятствующими одновременному добавлению элементов в кеш, при обработке запросов потребуется синхронизировать операции с ними, что приведет к задержкам, которые непросто будет выявить и устранить. Иногда можно исправить код, чтобы уменьшить использование блокировок, но такая возможность существует не всегда. Другой способ устранения конкуренции за ресурсы – запустить несколько рабочих процессов, выполняющих одно и то же приложение, каждый из которых обрабатывает собственное подмножество запросов, устраняя тем самым ненужную конкуренцию.

Другим примером, когда может пригодиться наличие нескольких процессов, выполняющих одно и то же веб-приложение, – использование 64-разрядного сервера IIS, выполняющего 32-разрядное веб-приложение. 64-разрядные серверы обычно имеют большой объем памяти, а 32-разрядное приложение может использовать не более 2 Гбайт, что часто приводит к увеличению частоты сборки мусора и, вероятно, к перезапускам пула приложения. Поддерживая два или три рабочих процесса для 32-разрядного веб-приложения, можно добиться более полного использования памяти сервера, уменьшить частоту сборки мусора и перезапусков пула приложения.

В настройках IIS пула приложения можно определить максимальное количество рабочих процессов, которое можно запустить для обслуживания запросов. Если установить этот параметр в значение больше 1 (значение по умолчанию), с ростом нагрузки на веб-приложение для него будут запускаться дополнительные рабочие процессы, вплоть до указанного максимума. Пул приложения, имеющий более одного процесса, называется «веб-садом» («Web Garden»). Каждый

раз, когда устанавливается соединение с клиентом, оно связывается с рабочим процессом, который будет обслуживать запросы от этого клиента, при этом соблюдается равномерное распределение запросов от пользователей между процессами и уменьшаются накладные расходы на конкуренцию.

Имейте в виду, что использование веб-сада имеет и недостатки. Большее количество рабочих процессов занимает больший объем памяти, исключается возможность использовать механизм по умолчанию хранения информации о сеансе в памяти процесса, при выполнении нескольких рабочих на одном компьютере, между ними может возникать конкуренция за локальные ресурсы, например, за использование общего файла журнала.

Оптимизация сети

Даже если ваш код работает быстро, а среда выполнения обеспечивает высокую пропускную способность, остается еще несколько узких мест – пропускная способность сетевых подключений клиентов, объем передаваемых данных и количество запросов, отправляемых клиентами. Существует несколько способов уменьшения количества запросов и размеров ответов, часть из которых заключается в простой настройке IIS, но другие требуют уделить более пристальное внимание программному коду приложения.

Включение HTTP-заголовков кеширования

Один из способов сэкономить трафик – обеспечить кеширование на стороне браузера всего содержимого, которое остается неизменным в течение некоторого времени. Отличными кандидатами для кеширования в браузере являются статические файлы изображений, сценариев и CSS. Однако динамическое содержимое, такое как файлы *.aspx* и *.ashx* также часто могут кешироваться, если фактическое их содержимое изменяется не слишком часто.

Настройка заголовков кеширования для статического содержимого

Статические файлы обычно отправляются клиентам с двумя заголовками кеширования:

- **ETag**. В этот HTTP-заголовок сервер IIS записывает хеш, вычисленный на основе даты последнего изменения содер-

жимого. Для статического содержимого, такого как файлы изображений и CSS, в заголовке ETag сервер IIS возвращает дату последнего изменения файла. В последующем, когда на сервер будут поступать запросы с ранее вычисленным значением ETag, IIS вычислит ETag для запрошенного файла и, если оно не совпадет с клиентским значением ETag, обратно будет отправлен запрошенный файл, а если совпадет, клиенту будет отправлен ответ HTTP 304 (Not Modified). Значение ETag, сохраненное клиентом, передается серверу в HTTP-заголовке `If-None-Match` запроса.

- **Last-Modified**. В этот HTTP-заголовок сервер IIS записывает дату последнего изменения запрошенного файла. Это – дополнительный заголовок кеширования, который может использоваться как запасной вариант, когда поддержка заголовка ETag отключена. когда на сервер будут поступать запросы, содержащие дату последнего изменения, IIS сравнит ее с датой последнего изменения файла и решит, отправить ли содержимое файла (если время последнего изменения файла изменилось) или послать ответ HTTP 304. Дата последнего изменения файла, сохраненная клиентом, передается серверу в HTTP-заголовке `If-Modified-Since` запроса.

Эти заголовки кеширования гарантируют, что это содержимое не будет посылаться клиенту повторно, если он уже имеет самую последнюю версию содержимого, но все еще требуют от клиента отправлять запросы, чтобы проверить – не изменилось ли содержимое. Если у вас имеются статические файлы, которые гарантированно не изменятся в ближайшие пару недель или даже месяцев, такие как логотип компании или сценарии, которые не изменятся до выхода следующей версии приложения, можно воспользоваться заголовками кеширования, позволяющими клиенту кешировать это содержимое и повторно использовать его, не посылая проверочные запросы на сервер. Этого можно добиться с помощью HTTP-заголовка `Cache-Control` с атрибутом `max-age` или HTTP-заголовка `Expires`. Разница между атрибутом `max-age` и заголовком `Expires` в том, что `max-age` определяет срок хранения, а заголовок `Expires` позволяет указать фиксированную дату и время, когда истекает срок хранения содержимого. Например, если установить в атрибуте `max-age` значение 3600, браузер будет хранить содержимое в кеше один час (3600 секунд = 60 минут = 1 час) и автоматически использовать его, не посылая запросы серверу. Как только срок хранения содер-

жимого истечет (независимо от использовавшегося заголовка кеширования), он будет помечено как устаревшее. Когда в следующий раз браузер обнаружит, что содержимое устарело, он отправит на сервер запрос на получение более нового содержимого.

> **Совет.** *Убедиться в отсутствии запросов на получение кешированного содержимого можно с помощью инструментов мониторинга HTTP-трафика, таких как Fiddler, и узнать, какие запросы, посылаются серверу. Если обнаружится запрос на содержимое, которое по вашему мнению должно кешироваться, проверьте наличие заголовков* `max-age/Expires` *в ответе на этот запрос.*

Использование `max-age/Expires` совместно с `ETag/Last-Modified` гарантирует, что на запрос, отправленный после того, как истечет срок хранения содержимого, будет возвращен ответ HTTP 304, если запрошенное содержимое на сервере в действительности не изменилось. Ответ в этом случае будет содержать новый HTTP-заголовок `max-age/Expires`.

В большинстве браузеров щелчок на кнопке **Refresh** (Обновить) (или нажатие клавиши **F5**) заставит браузер обновить кеш, проигнорировав заголовок `max-age/Expires`, и отправить запрос на получение кешированного содержимого, даже если срок его хранения еще не истек. Запросы все еще будут содержать заголовки `If-Modified-Since/If-None-Match`, если они применимы, чтобы сервер мог вернуть ответ HTTP 304, если содержимое не изменилось.

Чтобы включить поддержку заголовка `max-age`, добавьте следующие настройки в файл *web.config*:

```
<system.webServer>
    <staticContent>
        <clientCache cacheControlMode = "UseMaxAge"
                     cacheControlMaxAge = "0:10:00" />
    </staticContent>
</system.webServer>
```

Фрагмент с настройками выше гарантирует, что в ответ на все запросы, отправленные для получения статического содержимого, будут возвращаться ответы с HTTP-заголовком `Cache-Control`, содержащим атрибут `max-age` со значением 600 секунд.

Чтобы задействовать заголовок `Expires`, измените элемент `clientCache`, как показано ниже:

```
<system.webServer>
    <staticContent>
        <clientCache cacheControlMode = "UseExpires"
```

```
                    httpExpires = "Fri, 11 Jul 2014 6:00:00 GMT"/>
    </staticContent>
</system.webServer>
```

Фрагмент с настройками выше сообщает, что срок хранения статического содержимого выше истечет 11 июля 2014 года в 6 часов утра.

Если потребуется указать разные сроки хранения для разного содержимого, например, для файлов JavaScript – фиксированную дату, а для изображений 100-дневный срок хранения, можно добавить разделы `location` и указать разные настройки для разных компонентов приложения, как показано ниже:

```
<location path = "Scripts">
    <system.webServer>
        <staticContent>
            <clientCache cacheControlMode = "UseExpires"
                         httpExpires = "Fri, 11 Jul 2014 6:00:00 GMT" />
        </staticContent>
    </system.webServer>
</location>
<location path = "Images">
    <system.webServer>
        <staticContent>
            <clientCache cacheControlMode = "UseMaxAge"
                         cacheControlMaxAge = "100.0:00:0" />
        </staticContent>
    </system.webServer>
</location>
```

Примечание. *Для установки заголовка* `Expires` *необходимо использовать полный формат представления даты. Кроме того, согласно спецификациям протокола HTTP, дата в заголовке* `Expires` *не должна находиться в будущем далее чем на один год от текущей даты.*

Настройка заголовков кеширования для динамического содержимого

Статические файлы имеют дату последнего изменения, которую можно использовать для проверки факта изменения содержимого после сохранения в кеше. Однако динамическое содержимое не имеет такой характеристики, потому что динамическое содержимое воссоздается заново в ответ на каждый запрос и датой его последнего изменения фактически является текущая дата, поэтому такие заголовки, как `ETag` и `Last-Modified` не подходят для кеширования динамического содержимого.

С другой стороны, если изучить содержимое динамической страницы, можно найти способ выразить дату последнего ее изменения или даже рассчитать значение `ETag` для нее. Например, если в ответ на запрос информация о продукте извлекается из базы данных, таблица со списком продуктов могла бы хранить поле с датой последнего изменения, которую в свою очередь можно было бы использовать для установки заголовка `Last-Modified`. В случае отсутствия такого поля в таблице, можно попробовать вычислить контрольную сумму MD5 полей и отправлять результат в заголовке `ETag`. При получении последующих запросов сервер мог бы вычислять контрольную сумму MD5 заново и при совпадении ее со значением `ETag` возвращать ответ HTTP 304.

Например, следующий код устанавливает заголовок `Last-Modified` для динамической страницы с описанием продукта:

```
Response.Cache.SetLastModified(product.LastUpdateDate);
```

В отсутствие даты последнего изменения, можно возвращать в заголовке ETag контрольную сумму MD5, как показано ниже:

```
Response.Cache.SetCacheability(HttpCacheability.ServerAndPrivate);

// Вычислить контрольную сумму MD5
System.Security.Cryptography.MD5 md5 =
    System.Security.Cryptography.MD5.Create();
string contentForEtag = entity.PropertyA +
                        entity.NumericProperty.ToString();
byte[] checksum = md5.ComputeHash(
                    System.Text.Encoding.UTF8.GetBytes(contentForEtag));

// Создать строку ETag на основе контрольной суммы.
// Строки ETag должны заключаться в двойные кавычки, как того
// требует стандарт
string etag =
    "\"" + Convert.ToBase64String(checksum, 0, checksum.Length) + "\"";
Response.Cache.SetETag(etag);
```

> **Примечание.** *По умолчанию поддержка заголовка ETag в ASP.NET выключена. Чтобы включить ее, необходимо изменить режим кеширования в* `ServerAndPrivate`*, разрешив кеширование содержимого на стороне сервера и на стороне клиента, но не на промежуточных компьютерах, таких как прокси-серверы.*

Получив запрос с заголовком `ETag`, вы можете сравнить вычисленное значение `ETag` с полученным от браузера и, если они совпадают, послать ответ HTTP 304, как показано ниже:

```
if (Request.Headers["If-None-Match"] == calculatedETag) {
    Response.Clear();
    Response.StatusCode = (int)System.Net.HttpStatusCode.NotModified;
    Response.End();
}
```

При наличии каких-либо предположений о сроке жизни динамического содержимого, можно устанавливать заголовки `max-age` или `Expires`. Например, если предполагается, что описание снятого с производства продукта никогда не изменится, можно установить срок хранения описания этого продукта равным одному году, как в следующем фрагменте:

```
if (productIsDiscontinued)
    Response.Cache.SetExpires(DateTime.Now.AddYears(1));
```

Для той же цели можно использовать атрибут `max-age` в заголовке `Cache-Control`:

```
if (productIsDiscontinued)
    Response.Cache.SetMaxAge(TimeSpan.FromDays(365));
```

Вместо установки сроков кеширования в коде, их можно устанавливать в файлах *.aspx*, в директивах кеширования. Например, если информация о продукте, отображаемая на странице, может храниться в кеше в течение 10 минут (600 секунд) на стороне клиента, в страницу можно добавить следующую директиву:

```
<%@ Page ... %>
<%@ OutputCache Duration = "600" Location = "Client"%>
```

При использовании директивы `OutputCache` указанная продолжительность хранения содержимого в кеше передается одновременно в заголовках `max-age` и `Expires` (значение для заголовка Expires вычисляется добавление продолжительности к текущей дате).

Включение сжатия в IIS

Кроме мультимедийных файлов (аудио- и видеороликов, изображений) и двоичных файлов, таких как компоненты Silverlight и Flash, большая часть содержимого (страницы HTML, таблицы стилей CSS, сценарии JavaScript, данные в формате XML и JSON) передается веб-сервером в текстовом виде. Используя поддержку сжатия, имеющуюся в IIS, эти текстовые данные можно сжимать в размерах, что позволит уменьшить объем передаваемых данных и время его передачи. Поддержка сжатия в IIS позволяет уменьшать размеры ответов

до 50–60 процентов от их оригинального размера, а иногда и больше. Сервер IIS поддерживает два типа сжатия, статическое и динамическое. Прежде чем использовать сжатие, убедитесь сначала в наличии компонентов статического и динамического сжатия.

Статическое сжатие

При использовании статического сжатия, сжатое содержимое сохраняется на диске, поэтому в ответ на последующие обращения к ресурсам возвращается уже сжатое содержимое и операция сжатия не выполняется повторно. За хранение сжатого содержимого приходится платить дисковым пространством, но при этом экономятся вычислительные мощности и уменьшаются задержки, которые обычно возникают из-за затрат времени на сжатие.

Статическое сжатие может пригодиться для файлов, которые обычно не изменяются (то есть, являются «статическими»), таких как файлы CSS и JavaScript. Но даже если оригинальный файл изменится, IIS обнаружит это и повторно сожмет его.

Имейте в виду, что наибольшего эффекта можно получить при сжатии на текстовых файлов (**.htm*, **.txt*, **.css*) и некоторых двоичных, таких как файлы документов Microsoft Office (**.doc*, **.xsl*). Но при применении к уже сжатым файлам, таким как файлы изображений (**.jpg*, **.png*) и сжатым файлам документов Microsoft Office (*.docx*, *.xslx*) может получиться даже обратный эффект.

Динамическое сжатие

Когда используется динамическое сжатие, IIS сжимает ресурс всякий раз, когда он запрашивается и сжатое содержимое при этом не сохраняется на диске. То есть, при последующих обращениях к ресурсу оно будет сжиматься повторно перед отправкой клиенту, вызывая тем самым увеличение нагрузки на процессор и задержки из-за затрат времени на сжатие. Поэтому динамическое сжатие лучше всего подходит для часто изменяющегося содержимого, такого как страницы ASP.NET.

Поскольку динамическое сжатие увеличивает нагрузку на процессор, настоятельно рекомендуем вам проверить использование процессора после включения сжатия, чтобы убедиться, что нагрузка не получилась слишком большой.

Настройка сжатия

Первое, что следует сделать для использования сжатия, – включить динамическое или статическое сжатие, или оба сразу. Чтобы

включить сжатие в IIS, откройте приложение IIS Manager, выберите свой компьютер, щелкните на пункте **Compression** (Сжатие) и выберите параметры сжатия, как показано на рис. 11.3.

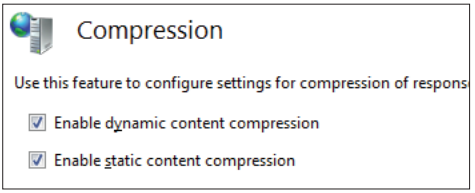

Рис. 11.3. Включение динамического и статического сжатия в приложении IIS Manager.

Можно также воспользоваться диалогом **Compression** (Сжатие) для настройки параметров статического сжатия, таких как имя папки, куда будет сохраняться сжатое содержимое, и минимальный размер файлов, подвергаемых сжатию.

После выбора типа сжатия можно пойти еще дальше и определить, какие типы MIME должны сжиматься статически, а какие динамически. К сожалению, IIS не поддерживает возможность выполнения этих настроек из приложения IIS Manager, поэтому вам придется изменить их вручную, в конфигурационном файле IIS *applicationHost.config*, находящийся в папке *%windir%\System32\inetsrv\config folder*. Откройте файл и найдите раздел `<httpCompression>`. В нем уже должно быть указано несколько типов MIME для статического и динамического сжатия. В дополнение к уже указанным типам вы можете добавить свои типы MIME содержимого, используемого вашими веб-приложениями. Например, при использовании функций AJAX, возвращающих данные в формате JSON, можно добавить динамическое сжатие для этого формата. Следующий фрагмент демонстрирует, как включить поддержку динамического сжатия для формата JSON (прочие определения были удалены для экономии места в книге):

```
<httpCompression>
    <dynamicTypes>
        <add mimeType = "application/json;
             charset = utf-8" enabled = "true" />
    </dynamicTypes>
</httpCompression>
```

Примечание. *После добавления в список новых типов MIME проверьте, действительно ли сжатие выполняется, с помощью инструментов перехвата HTTP-трафика, таких как Fiddler. Сжатые ответы должны иметь заголовок* `Content-Encoding` *со значением* `gzip` *или* `deflate`.

Сжатие и клиентские приложения

Чтобы сервер IIS мог сжимать исходящие ответы, он должен быть уверен, что клиентское приложение способно распаковывать сжатое содержимое. Поэтому, когда клиентское приложение посылает запрос серверу, оно должно добавить HTTP-заголовок `Accept-Encoding` со значением `gzip` или `deflate`.

Большинство известных браузеров автоматически добавляют этот заголовок, поэтому при работе с веб-приложением или с приложением Silverlight посредством браузера, IIS будет возвращать сжатое содержимое. Однако, если вы посылаете HTTP-запросы из своего приложения для .NET, используя для этого тип `HttpWebRequest`, заголовок `Accept-Encoding` не добавляется автоматически, поэтому вам необходимо добавить его вручную. Кроме того, `HttpWebRequest` не распаковывает сжатые ответы, если не настроить его специально. Например, при использовании объекта `HttpWebRequest` добавьте следующий код, чтобы включить возможность приема и распаковывания сжатого содержимого:

```
var request = (HttpWebRequest)HttpWebRequest.Create(uri);
request.Headers.Add(HttpRequestHeader.AcceptEncoding, "gzip,deflate");
request.AutomaticDecompression =
    DecompressionMethods.GZip | DecompressionMethods.Deflate;
```

Другие объекты, обладающие возможностью обмена по протоколу HTTP, такие как прокси-объект веб-службы ASMX или объект `WebClient`, также поддерживают сжатие, но требуют ручной настройки для отправки заголовка и распаковывания сжатого содержимого. Службы WCF на основе протокола HTTP, до версии WCF 4, не поддерживали сжатие в клиентах .NET, использующих ссылки на службы или фабрику каналов. Начиная с версии WCF 4, сжатие поддерживается автоматически (отправка заголовков и распаковывание сжатого содержимого).

Минификация и объединение

В веб-приложениях часто приходится обслуживать страницы, использующие несколько файлов JavaScript и CSS. Когда в странице имеется несколько ссылок на внешние ресурсы, она загружается дольше, и пользователь вынужден ждать пока закончится загрузка и анализ страницы, и всех связанных с ней ресурсов. При использовании внешних ресурсов мы сталкиваемся с двумя основными проблемами:

1. Количество запросов, которые браузер должен отправить и на которые должен получить ответы. Чем больше запросов приходится отправлять, тем больше времени будет тратиться на отправку всех запросов, потому что браузеры могут одновременно открывать лишь ограниченное количество соединений к единственному серверу (например, IE 9 может одновременно послать одному и тому же серверу не более 6 запросов).
2. Размеры ответов влияют на время, требуемое для их загрузки. Чем больший размер имеют ответы, тем больше времени требуется браузеру, чтобы загрузить их. Дополнительное влияние может также оказывать необходимость отправки новых запросов, если браузер достиг предела одновременно обрабатываемых запросов.

Чтобы решить эти проблемы, необходимо средство, позволяющее уменьшать размеры ответов и сокращать количество запросов (и ответов, соответственно). В ASP.NET MVC 4 и ASP.NET 4.5 это средство встроено в фреймворк и называется «Bundling and minification» (Объединение и минификация).

Под объединением понимается возможность группировки группы файлов под одним адресом URL, при обращении к которому возвращаются все файлы, объединенные в один ответ. А под минификацией понимается уменьшение размера файла стилей или сценария за счет удаления пробельных символов и, в случае сценариев, переименования переменных и функций, с присваиванием им более коротких имен.

Применение приема минификации совместно со сжатием может значительно уменьшить размеры ответов. Например, оригинальный файла библиотеки jQuery 1.6.2 имеет размер 240 Кбайт. После сжатия он уменьшается примерно до 68 Кбайт. Минифицированная версия оригинального файла имеет размер 93 Кбайт, чуть больше сжатой версии, а после сжатия минифицированной версии, размер файла уменьшается до 33 Кбайт, что составляет примерно 14 процентов от первоначального размера.

Чтобы создать пакет, объединяющий минифицированные файлы, сначала установите NuGet-пакет `Microsoft.AspNet.Web.Optimization` и добавьте ссылку на сборку `System.Web.Optimization`. После этого вы сможете использовать статический класс `BundleTable` для создания новых пакетов со сценариями и стилями. Объединение должно выполняться до загрузки страниц, поэтому весь объединенный код следует поместить в *global.asax*, в метод `Application_Start`. Например,

следующий код создает пакет (bundle) с именем *MyScripts* (доступный из виртуальной папки пакетов), содержащий три файла сценариев, которые автоматически будут минифицированы:

```
protected void Application_Start() {
    Bundle myScriptsBundle = new ScriptBundle("~/bundles/MyScripts").Include(
        "~/Scripts/myCustomJsFunctions.js",
        "~/Scripts/thirdPartyFunctions.js",
        "~/Scripts/myNewJsTypes.js");

    BundleTable.Bundles.Add(myScriptsBundle);
    BundleTable.EnableOptimizations = true;
}
```

> **Примечание.** *По умолчанию объединение и минификация выполняются, только когда режим компиляции веб-приложения установлен в значение* `release`. *Чтобы включить поддержку объединения во время отладки установите свойство* `EnableOptimizations` *в значение* `true`.

Чтобы воспользоваться созданным пакетом, добавьте в страницу следующую строку сценария:

```
<% = Scripts.Render("~/bundles/MyScripts") %>
```

Перед передачей страницы браузеру, строка выше будет заменена тегом `<script>`, ссылающимся на пакет. Например, строка выше может быть преобразована в следующий тег HTML:

```
<script
  src = "/bundles/MyScript?v = XGaE5OlO_bpMLuETD5_XmgfU5dchi8G0SSBExK294I41"
  type = "text/javascript" > </script>
```

По умолчанию механизм объединения и минификации устанавливает заголовки кеширования со сроком хранения один год, поэтому пакет будет оставаться в кеше браузера и при необходимости извлекаться оттуда. Чтобы предотвратить устаревание пакетов, каждый пакет снабжается маркером, который помещается в строку параметров URL. Если какой-либо из файлов будет удален из пакета, добавлен новый файл или файлы в пакете изменятся, изменится и маркер, поэтому при следующем же запросе к странице будет сгенерирован другой URL пакета с другим маркером, что вынудит браузер загрузить новый пакет.

Аналогично можно создать пакет для файлов CSS:

```
Bundle myStylesBundle = new StyleBundle("~/bundles/MyStyles")
       .Include("~/Styles/defaultStyle.css",
                "~/Styles/extensions.css",
```

```
                "~/Styles/someMoreStyles.js");

BundleTable.Bundles.Add(myStylesBundle);
```

И использовать его в странице с помощью директивы:

```
<% = Styles.Render("~/bundles/MyStyles") %>
```

Которая при отображении страницы превратится в элемент `<link>`:

```
<link href =
      "/bundles/MyStyles?v =
ji3nO1pdg6VLv3CVUWntxgZNf1zRciWDbm4YfW-y0RI1"
      rel = "stylesheet" type = "text/css" />
```

Механизм объединения и минификации поддерживает также пользовательские преобразования, что дает возможность создавать специализированные классы преобразований, например, выполняющие минификацию файлов JavaScript.

Использование сетей доставки содержимого (CDN)

Одной из проблем производительности, с которой часто сталкиваются веб-приложения, – это задержки, возникающие при доступе к ресурсам в сети. Когда веб-сервер и конечный пользователь находятся в одной локальной сети, время отклика обычно остается достаточно маленьким, но когда веб-приложение переносится в глобальную сеть и к нему начинают обращаться пользователи из разных уголков Земли через Интернет, удаленные пользователи, например с других континентов, наверняка будут замечать значительные задержки из-за сетевых проблем.

Одним из решений этой проблемы является географическое распространение экземпляров веб-сервера по всему миру, чтобы конечные пользователи географически всегда оказывались близки к одному из серверов. Однако, это решение создает значительные проблемы сопровождения, поскольку вам придется организовать постоянную репликацию и синхронизацию серверов, и, возможно, проинструктировать конечных пользователей о необходимости использования различных URL, в зависимости от их географического местоположения.

Здесь нам на помощь могут прийти *сети доставки содержимого* (Content Delivery Networks, CDN). Сеть доставки содержимого – это множество веб-серверов, размещенных в разных уголках Земли и

обеспечивающих географическую близость вашего веб-приложения к конечному пользователю. При использовании CDN, вы фактически используете единый адрес сети CDN, где бы ни находились, а местный сервер имен DNS будет преобразовывать этот адрес в фактический адрес ближайшего сервера CDN. Различные Интернет-компании, такие как Microsoft, Amazon и Akamai, имеют собственные сети CDN, которыми можно пользоваться за определенную плату.

Ниже описывается типичная последовательность действий по настройке сети CDN:

1. В настройках сети CDN вы указываете, где находится оригинальное содержимое.
2. При первом обращении конечного пользователя к содержимому через сеть CDN, локальный сервер CDN соединится с вашим веб-сервером, извлечет содержимое, сохранит его в своем кеше и вернет содержимое конечному пользователю.
3. В ответ на последующие запросы сервер CDN будет возвращать кешированное содержимое? не обращаясь к вашему веб-серверу, что обеспечит более быстрый отклик на запросы и, возможно, более высокую скорость передачи.

Примечание. *Помимо увеличения скорости обслуживания конечных пользователей, применение сетей CDN также позволяет уменьшить количество запросов на получение статического содержимого, что дает возможность выделить больше вычислительных ресурсов на обслуживание динамического содержимого.*

На первом шаге вам следует выбрать поставщика услуг CDN и настроить доставку своего содержимого, как описывается в инструкции поставщика. Когда у вас появится адрес CDN, просто замените ссылки на статическое содержимое (изображения, стили, сценарии) так, чтобы они ссылались на адрес CDN. Например, если вы выгрузили свое статическое содержимое в хранилище больших объектов Windows Azure и зарегистрировали его в Windows Azure CDN, вы можете изменить адреса URL в своих страницах, как показано ниже:

```
<link href = "http://az18253.vo.msecnd.net/static/Content/Site.css"
      rel = "stylesheet" type = "text/css" />
```

Для нужд отладки статический URL можно заменить переменной, что позволит управлять выбором локального адреса или адреса CDN. Например, следующий Razor-код конструирует URL, добавляя адрес CDN из параметра настройки `CdnUrl` приложения в файле *web.config*:

```
@using System.Web.Configuration
<script src =
    "@WebConfigurationManager.AppSettings["CdnUrl"]/Scripts/jquery-1.6.2.js"
    type = "text/javascript" > </script>
```

При отладке присвойте параметру `CdnUrl` пустую строку, чтобы обеспечить получение содержимого с локального веб-сервера.

Масштабирование приложений ASP.NET

Итак, вы увеличили производительность своего веб-приложения, применив все знания, полученные в этой книге, и может быть даже другие приемы, почерпнутые из других источников, и теперь ваше оптимизировано до предела. Далее вы развертываете приложение, вводите его в эксплуатацию и оно работает замечательно в течение нескольких первых недель, но с увеличением количества пользователей увеличивается и количество запросов, которые требуется обработать вашему серверу, и вдруг, ваш сервер начинает захлебываться. Сначала это может проявляться в увеличении времени обработки запросов, затем рабочий процесс начинает использовать все больше памяти и вычислительных ресурсов и в конечном итоге веб сервер просто перестает успевать обрабатывать все запросы и в файлах журналов начинают все чаще появляться сообщения HTTP 500 («Internal Server Error»).

Что случилось? Может быть снова заняться оптимизацией приложения? Однако, с увеличением количества пользователей ситуация повторится. Может быть увеличить объем памяти или добавить процессоры? Однако подобное расширение возможностей единственного компьютера имеет свои пределы. Пришло время признать тот факт, что вам необходимы дополнительные серверы.

Горизонтальное масштабирование (scaling out) веб-приложений – это естественный процесс, начинающийся в определенный момент в жизни веб-приложений. Один сервер может одновременно обслуживать десятки, сотни и даже тысячи пользователей, но он не в состоянии достаточно долго выдерживать пиковые нагрузки. Память начинает заполняться информацией о сеансах, обработка новых запросов приостанавливается из-за отсутствия свободных потоков выполнения, и переключения контекста начинают выполняться слишком часто, что ведет к увеличению задержек и снижению пропускной способности сервера.

Горизонтальное масштабирование

С архитектурной точки зрения, выполнить масштабирование совсем не сложно: достаточно приобрести еще один-два компьютера (или десять), разместить серверы за компьютером, выполняющим распределение нагрузки, и все! Но проблема в том, что обычно все не так просто.

Одной из основных проблем горизонтального масштабирования, с которыми сталкиваются разработчики – как реализовать привязку к серверу. Например, когда работает единственный веб-сервер, информация о состоянии сеансов пользователей хранится в памяти. Если добавить еще один сервер, как обеспечить для него доступ к объектам сеансов? Как синхронизировать сеансы между серверами? Некоторые веб-разработчики решают эту проблему, сохраняя информацию на сервере и связывая клиента с конкретным сервером. Как только клиент соединится с одним из серверов, находящихся за балансировщиком нагрузки, с этого момент все запросы от этого клиента будут направляться одному и тому же веб-серверу. Этот прием называется также привязкой сеанса. Привязка сеанса – это обходное решение, но оно не решает проблему, потому что не позволяет равномерно распределять нагрузку между серверами. Используя этот прием, легко попасть в ситуацию, когда один сервер будет обслуживать достаточно много пользователей, а другие будут в это время простаивать, потому что их клиенты уже закончили работу и отключились.

Поэтому настоящее решение заключается в том, чтобы не использовать память компьютера для хранения такие данные, как информация о сеансах пользователей или кеш. Но как хранение кеша в памяти определенного компьютера может помешать масштабированию? Представьте, что произойдет, когда пользователь пошлет запрос, вызывающий обновление кеша: сервер, получивший запрос, обновит свой кеш в памяти, но другие серверы не будут знать, что это необходимо сделать, и если в их кешах хранится копия того же объекта, это приведет к противоречивости данных в масштабе всего приложения. Один из способов решения этой проблемы – организовать синхронизацию объектов в кеше между серверами. Такое вполне возможно, но это усложнит общую архитектуру веб-приложения, не говоря уже о том, как вырастет объем трафика между серверами.

Механизмы масштабирования в ASP.NET

Горизонтальное масштабирование требует хранения информации о состоянии за пределами процессов. В ASP.NET имеется два механизма, обеспечивающих такой способ хранения данных.

- **Служба управления состоянием** (State Service). Служба управления состоянием – это служба Windows, поддерживающая управление состоянием для нескольких компьютеров. Эта служба устанавливается автоматически при установке .NET Framework, но она выключена по умолчанию. Вам достаточно просто выбрать, на каком сервере будет выполняться служба управления состоянием, и настроить все остальные на ее использование. Несмотря на то, что служба управления состоянием позволяет нескольким серверам использовать общее хранилище информации, она не поддерживает возможность долговременного хранения. То есть, если что-то приключится с сервером, где выполняется эта служба, вся информация о сеансах в вашей веб-ферме будет утеряна.
- **SQL Server**. ASP.NET поддерживает возможность хранения информации о состоянии в базе данных SQL Server. Этот механизм не только поддерживает те же возможности, что и служба управления состоянием, но также обеспечивает долговременное хранение данных, поэтому, даже если на веб-серверах и на сервере с базой данных SQL Server случится аварийная ситуация, информация о состоянии сохранится.

Для нужд кеширования в большинстве случаев можно с успехом использовать один из механизмов распределенного кеширования, таких как Microsoft AppFabric Cache, NCache или Memcached, последний из которых является открытой реализацией распределенного кеша. Механизм распределенного кеширования позволяет объединить память нескольких серверов в один распределенный кеш. Распределенные кеши поддерживают абстракцию местоположения, поэтому от вас не потребуется знать, где находится каждый фрагмент данных, службы уведомлений помогут оставаться в курсе – где и что изменилось, а высокая доступность гарантирует, что даже в случае аварии на одном из серверов данные не будут утеряны.

Некоторые распределенные кеши, такие как AppFabric Cache и Memcached, также имеют собственные реализации службы управления состоянием и провайдеров кеша для ASP.NET.

Ловушки горизонтального масштабирования

Хотя это и не имеет прямого отношения к производительности, все же стоит обозначить некоторые проблемы, с которыми можно столк-

нуться при масштабировании веб-приложений. Некоторые части веб-приложений требуют использования особых ключей безопасности для генерации уникальных идентификаторов, чтобы предотвратить возможность обмана веб-приложения и вторжения в него. Например, уникальный ключ используется в процедуре аутентификации Forms Authentication и при шифровании данных механизмом сохранения состояния представления. По умолчанию ключи безопасности для веб-приложений генерируются каждый раз, когда запускается пул приложения. В случае с единственным сервером это не вызывает никаких проблем, но когда веб-приложение выполняется на нескольких серверах, это может превратиться в проблему, так как каждый сервер будет иметь свой собственный уникальный ключ. Представьте такую ситуацию: клиент посылает запрос серверу *A* и получает в ответ cookie, подписанный уникальным ключом сервера *A*, затем клиент посылает новый запрос с принятым cookie, который попадает на сервер *B*. Поскольку сервер *B* имеет иной уникальный ключ, содержимое cookie признается недействительным и клиенту возвращается сообщение об ошибке.

Управлять генерацией этих ключей в ASP.NET можно путем настройки параметров в разделе `machineKey`, в файле *web.config*. Когда веб-приложение выполняется на нескольких серверах, вам необходимо настроить все серверы так, чтобы они использовали один и тот же предварительно сгенерированный ключ.

Другой проблемой, связанной с горизонтальным масштабированием и уникальными ключами, является возможность шифрования разделов в файлах *web.config*. Закрытая информация в файлах *web.config* часто шифруется, когда приложение развертывается на серверах. Например, раздел `connectionString` можно зашифровать, чтобы предотвратить утечку имени пользователя и пароля к базе данных. Вместо того, чтобы шифровать файл *web.config* на каждом сервере отдельно, усложняя процесс развертывания, можно сгенерировать один зашифрованный файл *web.config* и развернуть его на всех серверах. Для этого следует создать RSA-контейнер ключей и импортировать его на все веб-серверы.

Примечание. *Более полную информацию о создании уникальных ключей и включения их в настройки приложений можно получить в базе знаний Microsoft Knowledge Base, в документе Kb312906 (http://support.microsoft.com/?id=312906). За дополнительной информацией о создании RSA-контейнера ключей обращайтесь к статье «Importing and Exporting Protected Configuration RSA Key Containers» на сайте MSDN (http://msdn.microsoft.com/library/yxw286t2).*

В заключение

В начале этой главы мы утверждали, что общая производительность веб-приложения зависит не только от эффективности вашего кода, но также от всех компонентов, составляющих единый конвейер. Эту главу мы начали со знакомства с некоторыми инструментами тестирования и анализа, которые могут помочь в поиске узких мест в веб-приложениях. Выполняя всестороннее тестирование и применяя инструменты анализа вы легко сможете выявить проблемы и значительно повысить производительность своих веб-приложений. Затем мы прошлись через весь конвейер, указали, какие его части можно настроить, чтобы увеличить скорость работы веб-приложения или уменьшить объем посылаемых данных, чтобы они быстрее достигали места назначения. Теперь, прочитав эту главу, вы знаете, как даже самые небольшие изменения, такие как включение кеширования на стороне клиента, могут помочь уменьшить количество запросов к серверу и избавиться от некоторых проблем производительности, с которыми сталкивается большинство веб-приложений.

Позднее в этой главе мы увидели, что единственный сервер может оказаться неспособным обработать все запросы ваших клиентов. Поэтому особое значение приобретает заблаговременное планирование и применение решений проблем масштабирования, таких как распределенное кеширование и управление состоянием, которые упростят вам масштабирование в будущем, когда будет достигнут рубеж, за которым одного сервера окажется недостаточно. Наконец, в этой главе мы исследовали приемы увеличения производительности только серверной части веб-приложений и оставляем вам для самостоятельного исследования другую их часть – клиентскую.

Это – последняя глава в книге. В одиннадцати главах вы узнали, как измерять и увеличивать производительность приложений, как распараллеливать код для .NET и выполнять свои алгоритмы на графическом процессоре, как управлять сложностями системы типов .NET и сборщиком мусора, как правильно выбрать коллекцию и когда лучше реализовать свою собственную, и даже как использовать самые современные особенности процессоров, чтобы выжать дополнительную производительность. Спасибо, что следовали за нами в этом путешествии, и удачи вам в повышении производительности ваших приложений!

ПРЕДМЕТНЫЙ УКАЗАТЕЛЬ

Г

Д

З

И

К

M

Н

О

П

P

C

T

Книги издательства «ДМК Пресс» можно заказать в торгово-издательском холдинге
«Планета Альянс»наложенным платежом, выслав открытку или письмо по почтовому адресу:
115487, г. Москва, 2-й Нагатинский пр-д, д. 6А
При оформлении заказа следует указать адрес (полностью), по которому должны быть
Желательно также указать свой телефон и электронный адрес.
Эти книги вы можете заказать и в интернет-магазине: www.alians-kniga.ru.
Оптовые закупки: тел. (499) 782-38-89
Электронный адрес: books@alians-kniga.ru.

С ш Голдштейн
Дим Зурб лев
Идо Фл тов

Оптимизация приложений на платформе .NET

Гл вный ред ктор *Мовчан Д. А.*
dmkpress@gmail.com
Перевод с нглийского *Киселев А. Н.*
Корректор *Синяева Г. И.*
Верстк *Паранская Н. В.*
Диз йн обложки *Мовчан А. Г.*

Форм т 60×90 $^{1}/_{16}$.
Г рнитур «Петербург». Печ ть офсетн я.
Усл. печ. л. 32,095. Тир ж 100 экз.

Веб-с йт изд тельств : www.дмк.рф