Шакти Танвар

Параллельное программирование на C# и .NET Core

ДМК пресс издательство

DEVS UNIVERSE

Шакти Танвар

Параллельное программирование на C# и .NET Core

Hands-On Parallel Programming with C# and .NET Core

Build solid enterprise software using task parallelism and multithreading

Shakti Tanwar

Packt
BIRMINGHAM - MUMBAI

Параллельное программирование на C# и .NET Core

Создание надежного корпоративного программного обеспечения с использованием параллелизма и многопоточности

Шакти Танвар

Москва, 2022

УДК 004.438.NET
ББК 32.973.26-018.2
Т18

Танвар Ш.

Т18 Параллельное программирование на C# и .NET Core / пер. с англ. А. Д. Ворониной; ред. В. Н. Черников. – М.: ДМК Пресс, 2021. – 272 с.: ил.

ISBN 978-5-97060-851-7

Книга представляет подход к параллельному программированию с учетом современных реалий. Информация структурирована таким образом, чтобы она легко усваивалась, даже если читатель не обладает специальными знаниями. Рассматриваются общие принципы написания параллельного и асинхронного кода; реализация параллелизма данных показана на коротких и простых примерах. В конце глав приводятся вопросы для повторения пройденного.

Издание предназначено для программистов C#, которые хотят изучить концепции параллельного программирования и многопоточности, а затем использовать полученные знания для приложений, построенных на базе .NET Core. Также оно пригодится специалистам, желающим ознакомиться с принципами работы параллельного программирования на современном оборудовании.

УДК 004.438.NET
ББК 32.973.26-018.2

First published in the English language under the title 'Hands-On Parallel Programming with C# 8 and .NET Core 3 – (9781789132410)'. Russian language edition copyright © 2021 by DMK Press. All rights reserved.

Все права защищены. Любая часть этой книги не может быть воспроизведена в какой бы то ни было форме и какими бы то ни было средствами без письменного разрешения владельцев авторских прав.

ISBN 978-1-78913-241-0 (англ.)
ISBN 978-5-97060-851-7 (рус.)

© Packt Publishing, 2019
© Перевод, оформление, издание, ДМК Пресс, 2021

Посвящается моим жене и сыну,
Кирти Танвар и Шашвату Сингх Танвар.

Они мой жизненно необходимый кислород и стимул,
вдохновляющий меня на достижение выдающихся успехов

Содержание

https://t.me/it_boooks

От издательства 14
Об авторе 15
О переводе 16
О рецензентах 17
Предисловие 18

Часть I. ТЕОРЕТИЧЕСКИЕ ОСНОВЫ РАБОТЫ С ПОТОКАМИ, МНОГОЗАДАЧНОСТИ И АСИНХРОННОСТИ 22

Глава 1. Введение в параллельное программирование 23
Технические требования 24
Подготовка к многоядерным вычислениям 24
  Процессы 24
    Дополнительно об ОС 24
  Многозадачность 25
  Hyper-threading 25
    Классификация Флинна 26
  Потоки 27
    Типы потоков 27
  Многопоточность 30
    Класс Thread 31
    Класс ThreadPool 35
    BackgroundWorker 38
  Многопоточность и многозадачность 41
Сценарии, при которых полезно параллельное программирование 42
Преимущества и недостатки параллельного программирования 42
Резюме 43
Вопросы 44

Глава 2. Параллелизм задач 45
Технические требования 45
Задачи 46
Создание и запуск задачи 46
  Класс System.Threading.Tasks.Task 47
    Синтаксис лямбда-выражений 47
    Делегат Action 47
    Делегат 47

Метод System.Threading.Tasks.Task.Factory.StartNew 48
Синтаксис лямбда-выражений 48
Делегат Action 48
Делегат 48
Метод System.Threading.Tasks.Task.Run 49
Синтаксис лямбда-выражений 49
Делегат Action 49
Делегат 49
Метод System.Threading.Tasks.Task.Delay 49
Метод System.Threading.Tasks.Task.Yield 50
Метод System.Threading.Tasks.Task.FromResult<T> 52
Методы System.Threading.Tasks.Task.FromException и System.Threading.Tasks.Task.FromException<T> 53
Методы System.Threading.Tasks.Task.FromCanceled и System.Threading.Tasks.Task.FromCanceled<T> 53
Результаты выполнения задач 54
Отмена задач 55
Создание метки 55
Создание задач с использованием меток 56
Опрос состояния метки через свойство IsCancellationRequested 56
Регистрация отмены запроса с помощью делегата обратного вызова 57
Ожидание выполнения задач 58
Task.Wait 59
Task.WaitAll 59
Task.WaitAny 60
Task.WhenAll 60
Task.WhenAny 61
Обработка исключений в задачах 61
Обработка исключений из одиночных задач 62
Обработка исключений из нескольких задач 62
Обработка исключений задач с помощью обратного вызова 63
Преобразование шаблонов APM в задачи 64
Преобразование EAP в задачи 66
И еще о задачах 67
Цепочки задач 67
Продолжение выполнения задач с помощью метода Task.ContinueWith 68
Продолжение выполнения задач с помощью Task.Factory.ContinueWhenAll и Task.Factory.ContinueWhenAll<T> 69
Продолжение выполнения задач с помощью Task.Factory.ContinueWhenAny и Task.Factory.ContinueWhenAny<T> 69
Родительские и дочерние задачи 70
Создание отсоединенной задачи 70
Создание присоединенной задачи 71
Очереди с перехватом работы 72
Резюме 74

Глава 3. Реализация параллелизма данных 75
Технические требования 75
От последовательных циклов к параллельным 75
Метод Parallel.Invoke 76
Метод Parallel.For 78
Метод Parallel.ForEach 79
Степень параллелизма 80
Создание своей стратегии разделения данных 82
Разделение данных по диапазону 83
Разделение данных по блокам 83
Отмена циклов 84
Использование метода Parallel.Break 85
Использование ParallelLoopState.Stop 86
Использование CancellationToken для отмены циклов 87
Хранение данных в параллельных циклах 88
Локальная переменная потока 89
Локальная переменная блока данных 90
Резюме 91
Вопросы 91

Глава 4. Использование PLINQ 93
Технические требования 93
LINQ-провайдеры в .NET 93
Создание PLINQ-запросов 94
Знакомство с классом ParallelEnumerable 94
Наш первый запрос PLINQ 95
Сохранение порядка в PLINQ при параллельном исполнении 96
Последовательное выполнение с использованием метода AsUnOrdered() 97
Параметры объединения данных в PLINQ 98
Параметр NotBuffered 98
Параметр AutoBuffered 99
Параметр FullyBuffered 100
Отправка и обработка исключений с помощью PLINQ 102
Объединение параллельных и последовательных запросов LINQ 104
Отмена запросов PLINQ 104
Недостатки параллельного программирования с помощью PLINQ 106
Факторы, влияющие на производительность PLINQ (ускорения) 106
Степень параллелизма 107
Настройка объединения данных 107
Тип разделения данных 107
Когда нужно сохранять последовательное исполнение в PLINQ? 107
Порядок работы 108
ForAll против вызова ToArray() или ToList() 108
Принудительный параллелизм 108
Генерация последовательностей 108
Резюме 109
Вопросы 110

Часть II. СТРУКТУРЫ ДАННЫХ .NET CORE, КОТОРЫЕ ПОДДЕРЖИВАЮТ ПАРАЛЛЕЛИЗМ 111

Глава 5. Примитивы синхронизации 112

Технические требования 112
Что такое примитивы синхронизации? 113
Операции со взаимоблокировкой 113
Барьеры доступа к памяти в .NET 115
Что такое изменение порядка? 115
Типы барьеров памяти 116
Как избежать изменения порядка 117
Введение в примитивы блокировки 118
Как работает блокировка 118
Состояния потока 118
Блокировка или вращение? 119
Блокировка, мьютекс и семафор 120
Lock 120
Mutex 123
Semaphore 124
ReaderWriterLock 126
Введение в сигнальные примитивы 126
Thread.Join 126
EventWaitHandle 128
AutoResetEvent 128
ManualResetEvent 129
WaitHandles 131
Легковесные примитивы синхронизации 134
Slim locks 134
ReaderWriterLockSlim 135
SemaphoreSlim 136
ManualResetEventSlim 137
События Barrier и CountDownEvent 137
Примеры использования Barrier и CountdownEvent 138
SpinWait 140
SpinLock 141
Резюме 142
Вопросы 142

Глава 6. Использование параллельных коллекций 144

Технические требования 144
Введение в параллельные коллекции 144
Знакомство с IProducerConsumerCollection<T> 145
Использование ConcurrentQueue <T> 145
Производительность Queue<T> в сравнении с ConcurrentQueue<T> 148
Использование ConcurrentStack <T> 148
Создание параллельного стека 149

Использование ConcurrentBag<T> 150
Использование BlockingCollection<T> 151
Создание BlockingCollection<T> 151
Сценарий с несколькими производителями и потребителями 153
Использование ConcurrentDictionary<TKey,TValue> 154
Резюме 155
Вопросы 156

Глава 7. Повышение производительности с помощью отложенной инициализации 157
Технические требования 157
Что такое отложенная инициализация? 157
Введение в System.Lazy<T> 160
Логика создания объекта реализуется в конструкторе 161
Логика создания объекта передается в качестве делегата в Lazy<T> 162
Обработка исключений с помощью шаблона отложенной инициализации 163
Отсутствие исключений в ходе инициализации 163
Случайное исключение при инициализации с кешированием исключений 163
Некешируемые исключения 165
Отложенная инициализация с локальным хранилищем потоков 166
Сокращение издержек при помощи отложенной инициализации 168
Резюме 170
Вопросы 170

Часть III. АСИНХРОННОЕ ПРОГРАММИРОВАНИЕ С ИСПОЛЬЗОВАНИЕМ C# 172

Глава 8. Введение в асинхронное программирование 173
Технические требования 174
Типы выполнения программ 174
Синхронное выполнение программ 174
Асинхронное выполнение программ 176
Случаи использования асинхронного программирования 177
Написание асинхронного кода 177
Использование метода BeginInvoke класса Delegate 178
Использование класса Task 179
Использование интерфейса IAsyncResult 179
Когда не следует использовать асинхронное программирование 181
В базе данных без пула обработки подключений 181
Когда важно, чтобы код легко читался и поддерживался 181
Для простых и быстрых операций 181
Для приложений с большим количеством разделяемых данных 182
Проблемы, решаемые асинхронным кодом 182
Резюме 183
Вопросы 183

Глава 9. Основы асинхронного программирования с помощью async, await и задач 184
Технические требования 184
Введение в async и await 185
Возвращаемый тип асинхронных методов 188
Асинхронные делегаты и лямбда-выражения 189
Асинхронные шаблоны на основе задач 189
Метод компилятора с ключевым словом async 189
Ручная реализация TAP 189
Обработка исключений с помощью асинхронного кода 190
Метод, возвращающий Task и создающий исключение 190
Асинхронный метод вне блока try-catch без await 191
Вызов асинхронного метода из блока try-catch без await 192
Вызов асинхронного метода с await за пределами блока try-catch 194
Метод, возвращающий значение void 194
Асинхронность с PLINQ 195
Оценка производительности асинхронного кода 196
Рекомендации по написанию асинхронного кода 198
Не используйте async void 199
Все методы в цепочке вызовов должны быть асинхронными 199
По возможности используйте ConfigureAwait 200
Выводы 200
Вопросы 200

Часть IV. ОТЛАДКА, ДИАГНОСТИКА И МОДУЛЬНОЕ ТЕСТИРОВАНИЕ АСИНХРОННОГО КОДА 202

Глава 10. Отладка задач с Visual Studio 203
Технические требования 204
Отладка с VS 2019 204
Отладка потоков 204
Использование окон параллельных стеков 206
Отладка при помощи окон параллельных стеков 207
Представление потоков 207
Представление задач 209
Отладка с использованием окна контроля параллельных данных 209
Использование визуализатора параллелизма 211
Представление использования 212
Представление потоков 212
Представление ядер 213
Выводы 214
Вопросы 214
Дополнительные материалы для чтения 215

Глава 11. Создание модульных тестов для параллельного и асинхронного кодов 216
Технические требования 216

Модульное тестирование с .NET Core 217
Проблемы при написании модульных тестов для асинхронного кода 219
Создание модульных тестов для параллельного и асинхронного кодов 221
Проверка на успешный результат 221
Проверка результата исключения при нулевом делителе 222
Имитация обращений к реальным методам и данным с помощью Moq 222
Инструменты тестирования 224
Выводы 225
Вопросы 226
Дополнительные материалы для чтения 226

Часть V. ДОПОЛНИТЕЛЬНЫЕ СРЕДСТВА ПОДДЕРЖКИ ПАРАЛЛЕЛЬНОГО ПРОГРАММИРОВАНИЯ В .NET CORE 227

Глава 12. IIS и Kestrel в ASP.NET Core 228

Технические требования 228
Многопоточность в IIS и внутренние компоненты 229
Предотвращение нехватки ресурсов 229
Поиск восхождения к вершине 229
Многопоточность в Kestrel и внутренние компоненты 231
ASP.NET Core 1.x 232
ASP.NET Core 2.x 232
Лучшие практики использования многопоточности в микросервисах 233
Микросервисы с одним потоком и одним процессором 233
Микросервисы с одним потоком и несколькими процессорами 234
Микросервисы с несколькими потоками и одним процессором 234
Асинхронные сервисы 234
Выделенные пулы потоков 234
Введение асинхронности в ASP.NET MVC Core 235
Асинхронные потоки 238
Выводы 241
Вопросы 241

Глава 13. Шаблоны параллельного программирования 243

Технические требования 243
Шаблон MapReduce 243
Реализация MapReduce с помощью LINQ 244
Агрегация 246
Шаблон разделения/объединения 248
Шаблон спекулятивной обработки 248
Шаблон отложенной инициализации 249
Шаблон разделяемого состояния 252
Выводы 252
Вопросы 253

Глава 14. Управление распределенной памятью 254

Технические требования 255

Введение в распределенные системы......255
Модель общей и распределенной памяти......256
  Модель общей памяти......256
  Модель распределенной памяти......257
Типы коммуникационных сетей......258
  Статические коммуникационные сети......258
  Динамические коммуникационные сети......259
Свойства коммуникационных сетей......259
  Топология......260
  Алгоритмы маршрутизации......261
  Стратегия коммутации......261
  Управление потоком......261
Исследование топологий......262
  Линейная и кольцевая топологии......262
    Линейные массивы......262
    Кольцо или тор......263
  Решетки и торы......263
    Двумерные решетки......263
    2D-тор......264
Программирование устройств с распределенной памятью с использованием передачи сообщений......264
  Почему MPI?......265
  Установка MPI на Windows......265
  Пример программы с использованием MPI......265
Базовое использование отправки/приема сообщений......266
Коллективы......267
Выводы......267
Вопросы......268

Ответы на вопросы......269

Предметный указатель......270

От издательства

Отзывы и пожелания

Мы всегда рады отзывам наших читателей. Расскажите нам, что вы думаете об этой книге – что понравилось или, может быть, не понравилось. Отзывы важны для нас, чтобы выпускать книги, которые будут для вас максимально полезны.

Вы можете написать отзыв на нашем сайте www.dmkpress.com, зайдя на страницу книги и оставив комментарий в разделе «Отзывы и рецензии». Также можно послать письмо главному редактору по адресу dmkpress@gmail.com; при этом укажите название книги в теме письма.

Если вы являетесь экспертом в какой-либо области и заинтересованы в написании новой книги, заполните форму на нашем сайте по адресу http://dmkpress.com/authors/publish_book/ или напишите в издательство по адресу dmkpress@gmail.com.

Скачивание исходного кода примеров

Скачать файлы с дополнительной информацией для книг издательства «ДМК Пресс» можно на сайте www.dmkpress.com на странице с описанием соответствующей книги.

Список опечаток

Хотя мы приняли все возможные меры для того, чтобы обеспечить высокое качество наших текстов, ошибки все равно случаются. Если вы найдете ошибку в одной из наших книг, мы будем очень благодарны, если вы сообщите о ней главному редактору по адресу dmkpress@gmail.com. Сделав это, вы избавите других читателей от недопонимания и поможете нам улучшить последующие издания этой книги.

Нарушение авторских прав

Пиратство в интернете по-прежнему остается насущной проблемой. Издательства «ДМК Пресс» и Packt Publishing очень серьезно относятся к вопросам защиты авторских прав и лицензирования. Если вы столкнетесь в интернете с незаконной публикацией какой-либо из наших книг, пожалуйста, пришлите нам ссылку на интернет-ресурс, чтобы мы могли применить санкции.

Ссылку на подозрительные материалы можно прислать по адресу электронной почты dmkpress@gmail.com.

Мы высоко ценим любую помощь по защите наших авторов, благодаря которой мы можем предоставлять вам качественные материалы.

Об авторе

Шакти Танвар является генеральным директором Techpro Compsoft Pvt Ltd, глобального поставщика консалтинговых услуг в области информационных технологий. Шакти – IT-евангелист и архитектор программного обеспечения с 15-летним опытом работы в области разработки программного обеспечения и корпоративного обучения. Он также является сертифицированным преподавателем Microsoft и проводит обучение в сотрудничестве с Microsoft на Ближнем Востоке.

Шакти Танвар специализируется в таких областях, как .NET, машинное обучение в Azure, искусственный интеллект, применение чистого функционального программирования для построения отказоустойчивых систем и параллельные вычисления.

Его любовь к преподаванию привела к тому, что он запустил специальную программу «Обучение профессоров» с целью улучшения работы колледжей в Индии.

Эта книга была бы невозможна без неоценимой помощи моей жены Кирти и моего сына Шашвата, разделивших со мной все взлеты и падения. Благодаря их поддержке и желанию действовать я продолжал двигаться вперед в тяжелое время.

Я бесконечно благодарен своим родителям, братьям и сестрам, которые всегда побуждали меня к достижению новых высот.

Огромное спасибо моим друзьям, наставникам и команде Packt, которые сопровождали меня на протяжении всего пути.

О переводе

Данная книга далась нам непросто и потребовала больше года на перевод и кропотливую редактуру, но оно того стоило. Тема разработки параллельных программ сейчас актуальна как никогда, хотя и является очень непростой для понимания. В этой книге автор изложил не только сложные технические аспекты написания параллельных программ, но и объяснил механизмы реализации многопоточности в .NET, а также особенности языка C#.

Над переводом этой книги работали специалисты компании Devs Universe:

- **Алина Воронина** – переводчик, специалист по обучению разработчиков английскому языку в компании Devs Universe;
- **Вячеслав Черников** – редактор перевода, к. т. н. в области разработки ПО, основатель компании Devs Universe, автор книги «Разработка мобильных приложений на C# для iOS и Android», в прошлом – один из Microsoft MVP, Nokia Champion, Qt Certified Specialist, Qt Ambassador, автор статей для «Хабрахабра», «Хакера», Microsoft Developer Blogs, говоритель для конференций;
- **Максим Веркошанский, Дмитрий Милкин, Марина Королькова** – помощь с редактурой, специалисты компании Devs Universe.

Мы надеемся, что наши переводы помогут вам глубже понять суть современных технологий и стать суперразработчиками.

О рецензентах

Элвин Эшкрафт – разработчик, живущий недалеко от Филадельфии. Он провел свою 23-летнюю карьеру, создавая программное обеспечение при помощи C#, Visual Studio, WPF, ASP.NET и т. д. Был удостоен награды Microsoft Most Valuable Professional девять раз. Вы можете увидеть его ежедневные подборки ссылок в блоге для .Net-разработчиков *Morning Dew*. Ранее Элвин работал в софтверных компаниях, включая Oracle, сейчас он является главным специалистом по разработке ПО в Allscripts, создавая программное обеспечение для здравоохранения. Для Packt Publishing им также были написаны и другие рецензии на такие книги, как *Mastering ASP.NET Core 2.0, Mastering Entity Framework Core 2.0* и *Learning ASP.NET Core 2.0*.

> *Я хотел бы поблагодарить свою замечательную жену Стелену и трех наших очаровательных дочерей за их поддержку и понимание. Многие вечера и выходные дни были посвящены чтению и пересмотру глав этой книги, для того чтобы она смогла выйти в свет – первоклассная и полезная книга для разработчиков .NET.*

Видья Врат Агарвал – любитель книг, спикер, автор публикаций для Apress и технический редактор более чем дюжины книг Apress, Packt и O'Reilly. Он является прикладным разработчиком с 20-летним опытом в области проектирования, создания и разработки распределенных программных решений для крупных предприятий. Будучи главным архитектором в T-Mobile, Видья Врат Агарвал работал с проектами B2C и B2B. Сейчас он также продолжает сотрудничество с другими архитекторами с целью разработки решений и дорожных карт для различных проектов T-Mobile для миллионов клиентов компании. Он рассматривает разработку программного обеспечения как ремесло и является большим сторонником архитектуры программного обеспечения и практики чистого кода (clean code).

Предисловие

Прошел почти год с момента, когда издательская компания Packt впервые связалась со мной по поводу книги. Я и предположить не мог, что этот долгий путь будет таким сложным, однако за это время я смог многому научиться. Книга, которую вы сейчас держите в руках, – это результат, который стоил долгих дней трудов, и я горжусь тем, что наконец-то представляю ее вам.

Процесс создания этой книги очень много для меня значит, так как я всегда мечтал написать о языке, с которого начинал свою карьеру. Язык программирования C# развивался очень стремительно, а платформа .NET Core еще сильнее улучшила его репутацию в сообществе разработчиков.

Для того чтобы книга была полезна широкому кругу разработчиков, мы поговорим как о классическом подходе, построенном на потоках (threads), так и о разработке с использованием библиотеки TPL (Task Parallel Library). Вначале будут рассмотрены основные концепции операционных систем (ОС), которые позволяют писать многопоточный код. Затем мы проанализируем различия между классическим подходом и TPL.

В этой книге я постараюсь подойти к параллельному программированию со стороны современных реалий. Все примеры будут короткими и простыми, чтобы облегчить ваше понимание. Содержание глав построено так, чтобы информация легко усваивалась, даже если вы не обладаете специальными знаниями.

Надеюсь, вы получите такое же удовольствие от чтения этой книги, как и я от ее написания.

Целевая аудитория

Данная книга предназначена для программистов C#, которые хотят изучить концепции параллельного программирования и многопоточности, а также использовать полученные знания для своих приложений, построенных на базе .NET Core. Книга будет полезна студентам и специалистам, желающим познакомиться с принципами работы параллельного программирования на современном оборудовании.

Предполагается, что вы уже имеете представление о C# и базовые знания о том, как работают операционные системы.

Краткий обзор

Глава 1 «Введение в параллельное программирование», в которой представлены важные понятия многопоточности и параллельного программирования, включает в себя описание того, как развивались операционные системы

для поддержки современных подходов параллельного программирования.

Глава 2 «Параллелизм задач» демонстрирует возможности разделения программы на задачи (tasks) с целью эффективного использования ресурсов процессора и повышения производительности.

В главе 3 «Реализация параллелизма данных», в которой основное внимание уделяется реализации параллелизма данных с использованием параллельных циклов, также рассматриваются дополнительные методы (extension methods), помогающие в достижении параллелизма, а также разделении данных.

Глава 4 «Использование PLINQ» рассказывает о преимуществах использования PLINQ, включая отмену запросов. Также рассматриваются особенности применения PLINQ.

В главе 5 «Примитивы синхронизации» рассматриваются конструкции, доступные в C# для работы с разделяемыми (shared) ресурсами в многопоточном коде.

Глава 6 «Использование параллельных коллекций» описывает использование преимуществ параллельных коллекций, доступных в .NET Core.

Глава 7 «Повышение производительности с помощью отложенной инициализации» посвящается повышению производительности с помощью паттерна отложенной (lazy, «ленивой») инициализации.

В главе 8 «Введение в асинхронное программирование» рассматривается то, как нужно писать асинхронный код в более ранних версиях .NET.

В главе 9 «Основы асинхронного программирования с помощью async, await и задач» рассказывается о том, как использовать преимущества на новых конструкциях в .NET Core для реализации асинхронного кода.

Глава 10 «Отладка задач с Visual Studio» посвящена различным инструментам, доступным в Visual Studio 2019, которые облегчают отладку параллельных задач (tasks).

В главе 11 «Создание модульных тестов для параллельного и асинхронного кодов» рассматриваются различные способы написания тестов в Visual Studio и .NET Core.

Глава 12 «IIS и Kestrel в ASP.NET Core» расскажет, что такое IIS и Kestrel и как этим пользоваться. В этой главе также рассматривается поддержка асинхронных потоков.

Глава 13 «Шаблоны параллельного программирования» описывает различные паттерны (patterns), уже реализованные в языке C#. Она также включает в себя примеры реализации таких шаблонов.

В главе 14 «Управление распределенной памятью» рассматривается совместное использование памяти в распределенных программах.

Прежде чем начать

Вам необходимо установить Visual Studio 2019 вместе с .NET Core 3.1.

Также рекомендуется иметь базовые знания в языке C# и механизмах работы операционных систем.

Скачайте примеры исходных кодов

Вы можете скачать примеры исходных кодов для этой книги по адресу www.packtpub.com. Если вы приобрели эту книгу в другом месте, то можете перейти на сайт https://www.packtpub.com/support и получить примеры по электронной почте.

Шаги по загрузке файлов:

1. Войдите в систему или зарегистрируйтесь на сайте www.packtpub.com.
2. Выберите вкладку **Support**.
3. Нажмите на кнопку **Code Downloads**.
4. Введите название книги на английском языке «Hands-On Parallel Programming with C# 8 and .NET Core 3» в поле поиска и следуйте инструкциям.

Как только файл будет загружен, убедитесь, что вы его распаковываете, используя последнюю версию программ:

1) WinRAR/7-Zip для Windows;
2) Zipeg/iZip/UnRarX для Mac;
3) 7-Zip/PeaZip для Linux.

Примеры кода для этой книги также размещены на GitHub: https://github.com/PacktPublishing/Hands-On-Parallel-Programming-with-C-8-and-NET-core-3.

При обновлении исходного кода он тоже меняется на GitHub.

В нашем обширном каталоге книг и видео, доступных по ссылке ниже, представлены и другие примеры: https://github.com/PacktPublishing/. Обязательно их посмотрите.

Изображения

Также предоставляем вам PDF-файл с использованными в книге цветными изображениями скриншотов/диаграмм, которые вы можете скачать по ссылке https://static.packt-cdn.com/downloads/9781789132410_ColorImages.pdf.

Условные обозначения

В данной книге используется целый ряд условных обозначений.

`CodeInText`: указывает кодовые слова, например названия таблиц в базах данных, имена папок, имена файлов и их расширения, имена путей, вводимую пользователем информацию или имена пользователей Twitter. Допустим: «Подключите загруженный образ диска `WebStorm-10*.dmg` в качестве нового диска вашей системы».

Блок кода обозначается следующим образом:

```
private static void PrintNumber10Times() {
  for (int i = 0; i < 10; i++) {
```

```
    Console.Write(1);
  }
  Console.WriteLine();
}
```

Некоторые строки или элементы кода могут быть выделены жирным шрифтом с целью привлечения внимания к ним:

```
private static void PrintNumber10Times() {
  for (int i = 0; i < 10; i++) {
    Console.Write(1);
  }
  onsole.WriteLine();
}
```

Жирным шрифтом обозначается новый термин, важное слово или слова, которые вы видите в диалоговых сообщениях. Пример: «Вместо того чтобы самим находить оптимальное количество потоков, мы можем предоставить это среде **Common Language Runtime**».

Предупреждение или важная информация.

Советы и рекомендации.

Часть I

ТЕОРЕТИЧЕСКИЕ ОСНОВЫ РАБОТЫ С ПОТОКАМИ, МНОГОЗАДАЧНОСТИ И АСИНХРОННОСТИ

В данной части вы познакомитесь с понятиями потока, многозадачности и асинхронного программирования.

Содержание части I включает в себя следующие главы:

- глава 1 «Введение в параллельное программирование»;
- глава 2 «Параллелизм задач»;
- глава 3 «Реализация параллелизма данных»;
- глава 4 «Использование PLINQ».

Глава 1

Введение в параллельное программирование

https://t.me/it_boooks

Возможность использования параллельного программирования реализована в .NET с самого начала и после появления **Task Parellel Library** (TPL) в .NET Framework 4.0 получила широкое распространение.

Многопоточность (multithreading) является подмножеством параллельного программирования и выступает одной из самых сложных тем для начинающих разработчиков. Язык программирования C# заметно эволюционировал с момента своего появления и сейчас может быть использован не только для «классической» многопоточности, но и для «современного» асинхронного программирования. Многопоточность C# уходит своими корнями в версию 1.0. Язык C# является преимущественно синхронным, однако в версии 5.0 была добавлена поддержка асинхронности, которая сделала его отличным выбором для прикладных программистов. В то время как многопоточность имеет дело только с распараллеливанием внутри самих процессов, параллельное программирование также имеет дело с механизмами межпроцессного взаимодействия (inter-process communication, IPC).

До появления TPL использовались классы `Thread`, `BackgroundWorker` и `ThreadPool`, которые позволяли реализовать многопоточность. C# 1.0 опирался на потоки (threads) для отделения фоновой работы от **обработки событий пользовательского интерфейса** (user interface, или UI), что позволяло разрабатывать отзывчивые приложения. Эта модель теперь называется классической работой с потоками (classic threading). Со временем она уступила место другой модели программирования, называемой TPL, которая опирается на задачи (tasks) и скрывает от разработчика работу с потоками.

В этой главе мы познакомимся с различными концепциями, которые помогут вам научиться писать многопоточный код с нуля.

В главе 1 будут освещены следующие темы:

- основные понятия многоядерных вычислений, начиная с общих концепций и процессов **операционной системы** (ОС);
- потоки и разница между многопоточностью и многозадачностью;

- преимущества и недостатки написания параллельного кода и сценариев, в которых целесообразно использование параллельного программирования.

Технические требования

Продемонстрированные в данной книге примеры были созданы в Visual Studio 2019 при помощи C# 8. Со всеми исходниками вы можете ознакомиться на сайте: https://github.com/PacktPublishing/Hands-On-Parallel-Programming-with-C-8-and-.NET-Core-3/tree/master/Chapter01.

Подготовка к многоядерным вычислениям

В этом разделе вам будут представлены основные концепции ОС, начиная с процесса (process), внутри которого живут и исполняются потоки (threads). Далее мы рассмотрим развитие многозадачности с момента появления аппаратных возможностей, которые способствовали развитию параллельного программирования. После этого мы попытаемся разобраться в различных способах создания потоков в коде приложений.

Процессы

Говоря простым языком, слово «процесс» (process) относится к программе, которая запущена на компьютере. Однако с точки зрения ОС процесс – это адресное пространство в оперативной памяти. Каждому приложению нужны процессы для запуска, вне зависимости от того, написано ли это приложение для смартфона, Windows или веб-браузера. Процессы обеспечивают защиту одних программ от других, работающих в той же системе: выделенные одной программой данные не могут случайным образом быть доступными для другой. Кроме этого, процессы также обеспечивают изоляцию, при которой программы могут запускаться и останавливаться независимо друг от друга и от самой ОС.

Дополнительно об ОС

Производительность приложений во многом зависит от конфигурации аппаратного обеспечения, которая включает в себя следующее:

- скорость центрального процессора;
- объем оперативной памяти;
- скорость жесткого диска (HDD);
- тип диска, то есть HDD или SSD.

За последние несколько десятилетий мы стали свидетелями существенного прогресса в области аппаратных технологий. Например, микропроцес-

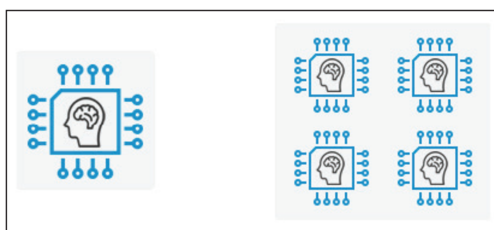
соры раньше имели одно ядро, которое представляло собой чип с одним **центральным процессором** (central processing unit, CPU). На рубеже веков мы увидели появление многоядерных процессоров, которые представляют собой чипы с двумя или более процессорами, где каждый обладает своим собственным кешем.

Многозадачность

Многозадачность – это способность компьютерной системы одновременно запускать несколько процессов или приложений. Количество процессов, которые могут выполняться системой, прямо пропорционально количеству ядер центрального процессора (ЦП). Таким образом, одноядерный процессор может выполнять только одну задачу за раз, двухъядерный процессор может одновременно выполнять две задачи, а четырехъядерный – четыре задачи. Если мы добавим к этому понятие планирования (scheduling) ЦП, то увидим, что ЦП одновременно запускает больше приложений, планируя или переключая их на основе алгоритмов планирования ЦП.

Hyper-threading

Hyper-threading (HT) – это запатентованная технология, разработанная компанией Intel, которая улучшает распараллеливание вычислений, выполняемых на процессорах x86. Впервые технология HT была представлена на серверных процессорах Xeon в 2002 году. Однопроцессорные чипы с поддержкой HT работают с двумя виртуальными (логическими) ядрами и способны выполнять две задачи одновременно. На следующей диаграмме показана разница между одноядерными и многоядерными чипами:

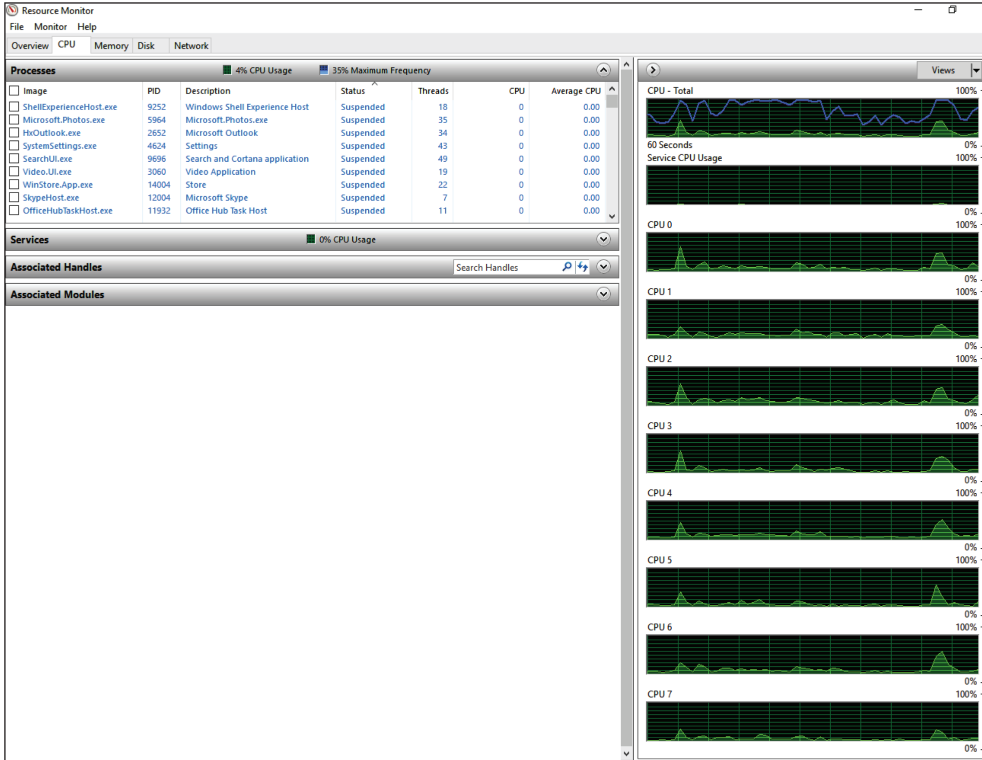


Ниже представлены примеры конфигураций процессоров и количество задач, которые они могут выполнять:

- **процессор с одноядерным чипом:** 1 задача за раз;
- **процессор с одноядерным чипом с поддержкой HT:** 2 задачи одновременно;
- **процессор с двухъядерным чипом:** 2 задачи одновременно;
- **процессор с двухъядерным чипом с поддержкой HT:** 4 задачи одновременно;

- **процессор с четырехъядерным чипом:** 4 задачи одновременно;
- **процессор с четырехъядерным чипом с поддержкой HT:** 8 задач одновременно.

На правой стороне в приложении **Resource Monitor** (Монитор ресурсов) (скриншот ниже) для четырехъядерной процессорной системы с поддержкой HT отображено восемь доступных процессоров.



Должно быть, вам интересно, насколько улучшится производительность компьютера при простом переходе от одноядерного процессора к многоядерному. На момент подготовки этой книги архитектура большинства самых быстрых суперкомпьютеров была основана на подходе «**множество команд, множество данных»** (Multiple Instruction, Multiple Data, MIMD). Данный вид архитектуры является одной из классификаций компьютерной архитектуры, предложенной Майклом Дж. Флинном в 1966 году.

Попробуем разобраться в этой классификации.

Классификация Флинна

В зависимости от количества параллельных потоков команд (или управления) и потоков данных Флинн классифицировал компьютерные архитектуры на четыре категории:

- **одиночный поток команд, одиночный поток данных** (Single Instruction, Single Data или SISD): в этой модели имеется один блок управления и один поток команд. Эти системы могут выполнять только одну команду за раз (без какой-либо параллельной обработки). Все одноядерные процессоры основаны на архитектуре SISD;
- **одиночный поток команд, множество потоков данных** (Single Instruction, Multiple Data или SIMD): в этой модели имеется только один поток команд и несколько потоков данных. Одна и та же программа параллельно применяется к нескольким наборам данных. Такая архитектура обеспечивает разделение данных на части и их параллельную обработку одним и тем же алгоритмом;
- **множество потоков команд, одиночный поток данных** (Multiple Instructions, Single Data или MISD): в этой модели множество потоков команд работают с одним потоком данных. Таким образом, несколько операций могут параллельно применяться к одним и тем же данным. Как правило, эта модель используется для обеспечения отказоустойчивости, например на ЭВМ, которые управляют полетом космических кораблей;
- **множество потоков команд, множество потоков данных** (Multiple Instructions, Multiple Data, MIMD): как видно из названия, эта модель предполагает наличие нескольких потоков команд и нескольких потоков данных. Благодаря такому подходу можно достичь истинного параллелизма, при котором каждый процессор способен выполнять несколько программ с разными наборами данных. В настоящее время большинство компьютерных систем используют этот тип архитектуры.

Теперь, когда мы разобрались с основами, давайте перейдем к обсуждению потоков.

Потоки

Поток (thread) – это единица выполнения, исполняемая внутри процесса (process). В любой момент времени программа может состоять из одного или нескольких потоков для лучшей производительности. Приложения Windows на основе графического интерфейса, такие как устаревшие **Windows Forms** (WinForms) или **Windows Presentation Foundation** (WPF), имеют выделенный поток для управления пользовательским интерфейсом и обработки действий пользователя. Этот поток также называется **потоком пользовательского интерфейса** (UI thread), или **потоком переднего плана** (foreground thread). Он владеет всеми элементами управления, которые создаются как часть пользовательского интерфейса.

Типы потоков

Существует два типа управляемых потоков: поток переднего плана и фоновый поток (background thread). Разница между ними заключается в следующем:

- **потоки переднего плана** оказывают непосредственное влияние на время жизни приложения. Приложение продолжает работать до тех пор, пока выполняется поток переднего плана;
- **фоновые потоки** не влияют на время жизни приложения. Таким образом, при закрытии приложения все фоновые потоки уничтожаются.

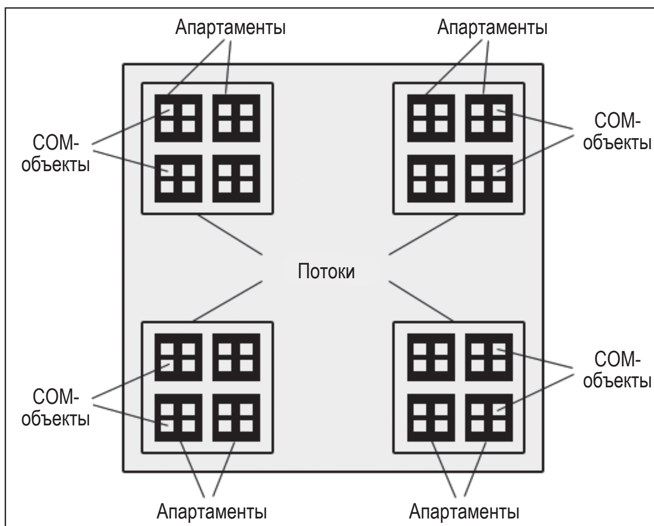
Приложение может содержать любое количество фоновых потоков и потоков переднего плана. Поток переднего плана поддерживает работу приложения, когда оно активно. Приложение полностью прекращает работу, когда последний поток переднего плана останавливается или прерывается. При выходе из приложения система останавливает все фоновые потоки.

Апартаментное состояние в модели COM

Другой немаловажной особенностью потоков выступает апартаментное состояние (apartment state). Это область внутри потока, где находятся объекты так называемой **модели компонентных объектов** (Component Object Model, COM).

COM является объектно-ориентированной системой для создания двоичного программного обеспечения, с которым может взаимодействовать пользователь. Также COM – это распределенная и кросс-платформенная система, на которой базируются технологии Microsoft OLE и ActiveX.

Как вы, возможно, знаете, все элементы управления Windows Forms содержат COM-объекты. Всякий раз, когда вы создаете приложение .NET WinForms, вы фактически размещаете на сервере провайдера эти COM-компоненты. Состояние подразделений – это отдельная область внутри прикладного процесса, где создаются COM-объекты. На следующей схеме показана связь между апартаментом потока и COM-объектами.



Из вышеуказанной схемы следует, что каждый поток имеет апартаменты, где расположены COM-объекты.

Поток может принадлежать одному из двух апартаментных состояний:

- **однопоточный апартамент** (Single-Threaded Apartment, STA): исходный COM-объект может быть доступен только через один поток;
- **многопоточный апартамент** (Multi-Threaded Apartment, MTA): исходный COM-объект может быть доступен одновременно через несколько потоков.

Ниже приведен список, содержащий важные моменты, касающиеся апартаментных состояний потоков:

- процессы могут иметь несколько потоков, среди которых – как потоки переднего плана, так и фоновые потоки;
- каждый поток может иметь один апартамент (STA либо MTA);
- каждый апартамент имеет модель параллелизма – либо однопоточную, либо многопоточную. Мы также можем программно изменять состояние потока;
- прикладной процесс может иметь более одного STA, но не MTA;
- примером приложения STA является Windows-приложение, а примером MTA – веб-приложение;
- COM-объекты создаются в апартаментах. Один COM-объект может находиться только в одном апартаменте потока, при этом апартаменты не могут быть общими.

Приложение можно принудительно запустить в режиме STA, используя атрибут STAThread в качестве основного метода. Пример метода `Main` для WinForms старого образца:

```
static class Program {
  ///<summary>
  ///Главная точка входа для приложения
  ///</summary>
  [STAThread]
  static void Main() {
    Application.EnableVisualStyles();
    Application.SetCompatibleTextRenderingDefault(false);
    Application.Run(new Form1());
  }
}
```

Атрибут `STAThread` также присутствует в WPF, однако скрыт от пользователей. Ниже приведен код скомпилированного класса `App.g.cs`, который находится в хранилище `obj/Debug` проекта WPF после компиляции:

```
///<summary>
///Приложение
///</summary>
public partial class App: System.Windows.Application {
  ///<summary>
  ///Инициализация компонента
  ///</summary>
  [System.Diagnostics.DebuggerNonUserCodeAttribute()]
```

```
  [System.CodeDom.Compiler.GeneratedCodeAttribute(
    "PresentationBuildTasks", "4.0.0.0")]

  public void InitializeComponent() {
    #line 5 "..\..\App.xaml"
    this.StartupUri = new System.Uri("MainWindow.xaml",
      System.UriKind.Relative);
    #line
    default
    #line hidden
  }
  ///<summary>
  ///Точка входа в приложение
  ///</summary>
  [System.STAThreadAttribute()]
  [System.Diagnostics.DebuggerNonUserCodeAttribute()]
  [System.CodeDom.Compiler.GeneratedCodeAttribute(
    "PresentationBuildTasks", "4.0.0.0")]

  public static void Main() {
    WpfApp1.App app = new WpfApp1.App();
    app.InitializeComponent();
    app.Run();
  }
}
```

Как вы могли заметить, метод `Main` снабжен атрибутом `STAThread`.

Многопоточность

Параллельное исполнение кода в .NET реализуется благодаря многопоточности. В зависимости от аппаратных возможностей процесс (или приложение) может использовать любое количество потоков. Любое приложение, включая консоль, существующие WinForms, WPF и даже веб-приложения, по умолчанию запускается одним потоком. Можно с легкостью достигнуть многопоточности, программно создавая больше потоков по мере необходимости.

Многопоточность обычно реализуется с помощью **планировщика потоков** (thread scheduler), отслеживающего момент, при котором активные потоки исчерпают отведенное им на выполнение время внутри процесса. Каждому созданному потоку присваивается приоритет (свойство `System.Threading.ThreadPriority`), который может иметь лишь одно из нижеуказанных допустимых значений. Приоритетом по умолчанию является значение «нормальный». Все возможные значения приоритета:

- Самый высокий (Highest);
- Выше нормы (AboveNormal);
- Нормальный (Normal);
- Ниже нормы (BelowNormal);
- Самый низкий (Lowest).

Для каждого потока внутри процесса ОС назначает временной отрезок на основе алгоритма планирования и приоритета потока. Каждая ОС имеет свой алгоритм планирования для потоков, поэтому в разных операционных системах порядок исполнения может отличаться. Это затрудняет поиск и исправление ошибок в работе потоков. Наиболее распространенный алгоритм планирования выглядит следующим образом.

1. Найти потоки с наивысшим приоритетом и запланировать их запуск.
2. Если имеется несколько потоков, обладающих наивысшим приоритетом, то каждому из них присваивается фиксированный временной диапазон (квант), в рамках которого они могут выполняться.
3. Как только потоки с наивысшим приоритетом заканчивают работу, то запускаются потоки с более низким приоритетом, для которых также выделяется определенное время.
4. Если создается новый поток с наивысшим приоритетом, то потоки с низким приоритетом снова сдвигаются назад.

Размеры временных квантов также зависят от длительности переключения между активными потоками. Кванты могут варьироваться в зависимости от конфигурации оборудования. Одноядерный процессор может одновременно запускать только один поток, поэтому планировщик проводит разделение времени между потоками. Квантование времени во многом зависит от тактовой частоты процессора, которая определяет производительность систем, однако по сравнению с многопоточностью – не в таком объеме. Более того, во время переключения контекста (загруженные в ОЗУ и кеш данные, необходимые для работы потока) появляются дополнительные издержки. Если потоку необходимо несколько временных промежутков для выполнения работы, то в таком случае поток должен быть выгружен из памяти, а затем снова загружен.

Понятие **параллелизма** в основном используется в контексте многоядерных процессоров. В основе многоядерного процессора лежит большее количество доступных процессорных модулей (ядер), как было упомянуто ранее, и поэтому различные потоки могут одновременно выполняться на разных ядрах. Чем больше будет процессоров, тем выше степень параллелизма.

Существует несколько способов создания потоков в программах:

- класс `Thread`;
- класс `ThreadPool`;
- класс `BackgroundWorker`;
- асинхронные делегаты;
- TPL.

В этой книге будут даны объяснения первым трем способам, а также подробно рассмотрены асинхронные делегаты и TPL.

Класс Thread

Самый простой и легкий способ создания потоков – через класс `Thread`, который определен в пространстве имен `System.Threading`. Этот подход использовался на платформе .NET с момента появления версии 1.0 и сейчас также применяется на .NET Core. Для создания потока необходимо предоставить

метод, который поток будет выполнять. Метод может либо содержать, либо не содержать параметры. Чтобы «обернуть» эти методы, библиотека классов предоставляет два делегата:

- `System.Threading.ThreadStart`;
- `System.Threading.ParameterizedThreadStart`.

Каждый из них будет позже продемонстрирован на примерах. Для начала мы рассмотрим, как создается поток, затем я постараюсь объяснить вам работу синхронной программы. После мы обратимся к понятию многопоточности, которое позволит понять асинхронный подход. Пример *создания потока*:

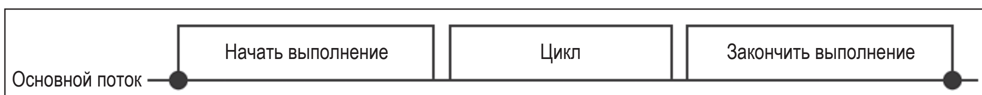
```
using System;
namespace Ch01 {
  class _1Synchronous {
    static void Main(string[] args) {
      Console.WriteLine("Start Execution!!!");
      PrintNumber10Times();
      Console.WriteLine("Finish Execution");
      Console.ReadLine();
    }
    private static void PrintNumber10Times() {
      for (int i = 0; i < 10; i++)
    }
    Console.Write(1);
  }
  Console.WriteLine();
}
```

В предыдущем примере все действия выполняются в основном потоке. Мы запросили метод `PrintNumber10Times` из Main-метода. Поскольку Main-метод вызывается основным потоком графического интерфейса, код выполняется синхронно. Если код будет долго запускаться из-за перегрузки основного потока, то это может привести к зависанию.

Так выглядит вывод в консоль:

```
C:\Program Files\dotnet\dotnet.exe
Start Execution!!!
1111111111
Finish Execution
```

Ниже приведена схема, отображающая последовательность выполнения кода в **основном потоке** (Main Thread):



На предыдущей диаграмме схематично изображено последовательное выполнение кода в основном потоке.

В данной ситуации мы можем сделать программу многопоточной, создав поток для вывода сообщения. Основной поток печатает инструкции, написанные в Main-методе:

```
using System;
namespace Ch01 {
  class _2ThreadStart {
    static void Main(string[] args) {
      Console.WriteLine("Start Execution!!!");
      //Использование потока без параметра
      CreateThreadUsingThreadClassWithoutParameter();
      Console.WriteLine("Finish Execution");
      Console.ReadLine();
    }

    private static void
    CreateThreadUsingThreadClassWithoutParameter() {
      System.Threading.Thread thread;
      thread = new System.Threading.Thread
      (new System.Threading.ThreadStart(PrintNumber10Times));
      thread.Start();
    }

    private static void PrintNumber10Times() {
      for (int i = 0; i < 10; i++) {
        Console.Write(1);
      }
      Console.WriteLine();
    }
  }
}
```

В этом коде мы делегировали выполнение `PrintNumber10Times()` новому потоку, созданному с помощью класса `Thread`. Команды `Console.WriteLine` в Main-методе по-прежнему выполняются через основной поток, однако `PrintNumber10Times` не вызывается через дочерний поток.

Ниже представлен вывод в консоль:

Далее схематично изображен представленный выше процесс. Можно заметить, что инструкция `Console.WriteLine` выполняется в **основном потоке**, а реализация самого цикла происходит в **дочернем потоке** (child thread).

Приведенная выше схема является примером многопоточной реализации.

Сравнив выходные данные, можно увидеть, что программа завершает все инструкции в основном потоке, а затем начинает печатать цифры 10 раз. В представленном примере используются простые команды, поэтому они и выполняются всегда одним и тем же образом. Однако если в основном потоке трудоемкие операторы (требующие больше времени) появятся раньше, чем будет отображено **Finish Execution**, то результаты могут отличаться. Чтобы получить более полное представление об этом, позже мы рассмотрим, как работает многопоточность и как она связана со скоростью процессора.

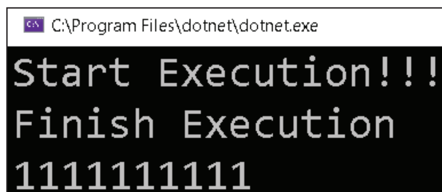
Вот другой пример, демонстрирующий передачу данных в поток с помощью делегата `System.Threading.ParameterizedThreadStart`:

```
using System;
namespace Ch01 {
  class _3ParameterizedThreadStart {
    static void Main(string[] args) {
      Console.WriteLine("Start Execution!!!");
      //Использование потока без параметра
      CreateThreadUsingThreadClassWithParameter();
      Console.WriteLine("Finish Execution");
      Console.ReadLine();
    }

    private static void CreateThreadUsingThreadClassWithParameter() {
      System.Threading.Thread thread;
      thread = new System.Threading.Thread
      new System.Threading.ParameterizedThreadStart
      PrintNumberNTimes));
      thread.Start(10);
    }

    private static void PrintNumberNTimes(object times) {
      int n = Convert.ToInt32(times);
      for (int i = 0; i < n; i++) {
        Console.Write(1);
      }
      Console.WriteLine();
    }
  }
}
```

Вывод этого кода выглядит так:



Использование класса `Thread` имеет свои преимущества и недостатки. Давайте попробуем в них разобраться.

Преимущества и недостатки потоков

Преимущества класса `Thread`:

- дочерние потоки могут быть использованы для освобождения основного потока;
- потоки можно использовать для разбиения задач на мелкие подзадачи, которые могут выполняться параллельно.

Недостатки класса `Thread`:

- при большом количестве потоков трудно отлаживать и поддерживать код;
- создание потоков нагружает систему относительно ресурсов памяти и процессора;
- необходимо выполнять обработку исключений внутри фоновых методов, так как любые необработанные исключения могут привести к сбою программы.

Класс ThreadPool

Создание потоков требует больших затрат с точки зрения как памяти, так и ресурсов ЦП. В среднем каждый поток потребляет около 1 МБ памяти и несколько сотен микросекунд процессорного времени. Производительность приложений – понятие относительное, поэтому она не всегда улучшается при большом количестве потоков. Напротив, создание большого количества потоков иногда может резко снизить производительность приложения. В приоритете – создание оптимального количества потоков в зависимости от загрузки процессора целевой системы, то есть запущенных в системе программ. Это происходит потому, что каждая программа получает свой временной квант, который затем распределяется между потоками внутри приложения. Если вы создадите слишком много потоков, они могут не успеть справиться с выполнением какого-либо конструктивного задания, прежде чем будут выгружены из памяти, чтобы другие потоки с аналогичным приоритетом могли выполниться.

Поиск оптимального количества потоков может вызвать сложности, поскольку зависит от конкретной системы, обладающей своей уникальной конфигурацией и определенным количеством одновременно работающих

приложений. То, что может быть оптимальным количеством для одной системы, может негативно сказаться для другой. Вместо самостоятельного поиска оптимального количества потоков мы можем воспользоваться средой **Common Language Runtime** (CLR). CLR обладает алгоритмом определения оптимального числа потоков на основе загрузки процессора в любой момент времени. Эта среда поддерживает пул потоков, известный как `ThreadPool`. `ThreadPool` привязан к процессу, при этом все приложения имеют свой собственный пул потоков. Преимущество пула потоков заключается в том, что он поддерживает оптимальное количество потоков и динамически связывает их с конкретными задачами (tasks). По завершении работы потоки возвращаются в пул, где могут быть привязаны к новой задаче, таким образом убираются затраты на создание и уничтожение потоков.

Оптимальное количество потоков, которые могут быть созданы в различных фреймворках внутри `ThreadPool`:

- 25 на ядро в .NET Framework 2.0;
- 250 на ядро в .NET Framework 3.5;
- 1023 в .NET Framework 4.0 в 32-битной среде;
- 32 768 в .NET Framework 4.0 и в .NET Core в 64-битной среде.

> Работая с инвестиционным банком, мы столкнулись со сценарием, при котором торговый процесс занимал почти 1800 секунд для синхронной брони 1000 сделок. Попробовав различные оптимальные числа, мы наконец переключились на `ThreadPool` и сделали процесс многопоточным. С помощью .NET Framework версии 2.0 приложение завершилось почти за 72 секунды. С версией 3.5 то же самое приложение завершилось всего за несколько секунд. Вместо того чтобы изобретать велосипед, достаточно прибегнуть к использованию фреймворка. Вы можете добиться необходимого прироста производительности, просто обновив фреймворк.

Создание потока через `ThreadPool` при помощи команды `ThreadPool.QueueUserWorkItem` показано в следующем примере.

Параллельный вызов нужного метода:

```
private static void PrintNumber10Times(object state) {
  for (int i = 0; i < 10; i++) {
    Console.Write(1);
  }
  Console.WriteLine();
}
```

Пример создания потока с помощью `ThreadPool.QueueUserWorkItem` и делегата `WaitCallback`:

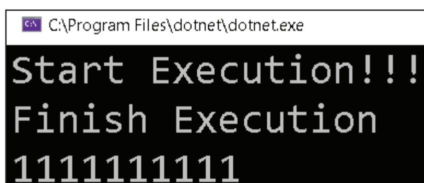
```
private static void CreateThreadUsingThreadPool() {
  ThreadPool.QueueUserWor(newWaitCallback(PrintNumber10Times));
}
```

Вызов из Main-метода:

```
using System;
using System.Threading;
namespace Ch01 {
```

```
  class _4ThreadPool {
    static void Main(string[] args) {
      Console.WriteLine("Start Execution!!!");
      CreateThreadUsingThreadPool();
      Console.WriteLine("Finish Execution");
      Console.ReadLine();
    }
  }
}
```

Вывод кода отражен на скриншоте:



Каждый пул потоков поддерживает минимальное и максимальное количества потоков. Эти значения можно изменить, вызвав следующие статические методы:

- `ThreadPool.SetMinThreads`;
- `ThreadPool.SetMaxThreads`.

> **!** Поток создается через `System.Threading`. Класс `Thread` не принадлежит к `ThreadPool`.

Далее мы рассмотрим преимущества и недостатки, связанные с использованием класса `ThreadPool`, и разберем случаи, когда следует избегать его использования.

Преимущества и недостатки ThreadPool. И когда лучше не обращаться к этому классу

Преимущества `ThreadPool`:

- дочерние потоки могут быть использованы для освобождения основного потока;
- оптимально потоки создаются и поддерживаются с помощью CLR.

Недостатки `ThreadPool`:

- при большом количестве потоков код становится трудно отлаживать и поддерживать;
- нам нужно выполнить обработку исключений внутри рабочего метода, так как любое необработанное исключение может привести к сбою программы;
- отчеты о ходе выполнения, отмена команд и логика завершения должны быть написаны с нуля.

Причины, по которым не следует обращаться к `ThreadPool`:

- когда нам нужен поток переднего плана (foreground thread);

- при необходимости установить явный приоритет потоку;
- когда у нас есть длительные или блокирующие задачи. Наличие большого количества заблокированных потоков в пуле предотвратит запуск новых задач из-за ограниченного количества потоков, доступных для `ThreadPool`;
- если нам нужны потоки STA, так как потоки в `ThreadPool` по умолчанию являются MTA;
- при необходимости привязать отдельный идентификационный номер к потоку нельзя присвоить название потоку `ThreadPool`.

BackgroundWorker

`BackgroundWorker` – это компонент, предоставляемый .NET для создания управляемых потоков в `ThreadPool`. Как вы ранее видели в примере приложения с графическим интерфейсом, Main-метод был снабжен атрибутом `STAThread`. Этот атрибут гарантирует безопасность управления, поскольку элементы пользовательского интерфейса создаются в апартаментах, принадлежащих основному потоку, и не могут быть совместно использованы другими потоками. В приложениях Windows есть основной поток, владеющий пользовательским интерфейсом и элементами управления, которые создаются при запуске приложения. Он отвечает за прием информации от пользователя и визуальное оформление (перекраску) интерфейса на основе действий пользователя. Для получения отзывчивых приложений мы должны постараться максимально освободить пользовательский интерфейс от длительных операций и делегировать все трудоемкие задачи фоновым потокам. Некоторые общие задачи, которые обычно выполняются рабочими потоками:

- загрузка изображений с сервера;
- взаимодействие с базой данных;
- взаимодействие с файловой системой;
- взаимодействие с веб-сервисами;
- сложные локальные вычисления.

Очевидно, что большинство из этих задач являются операциями **ввода-вывода** (I/O), которые выполняются центральным процессором. В момент вызова фрагмента кода, запускающего операцию ввода-вывода, выполнение передается от потока к процессору, который и выполняет задачу. Когда задача будет завершена, результат операции вернется обратно к вызывающему потоку. Период времени с момента передачи полномочий на выполнение задачи до получения результатов является периодом бездействия для потока, поскольку он просто должен ждать завершения операции процессором. Если это происходит в основном потоке, то приложение перестает отвечать на действия пользователя. Существуют и другие проблемы, которые необходимо решить для создания отзывчивых приложений. Давайте рассмотрим пример.

Сценарий:

Нам нужно получить данные от сервиса, который передает их в потоковом режиме. Затем необходимо проинформировать пользователя о проценте вы-

полнения работы. Как только работа будет завершена, нужно будет показать пользователю загруженные данные.

Проблемы:

Обращение к внешнему сервису занимает много времени, поэтому нам нужно выполнить его в фоновом потоке, чтобы избежать «заморозки» пользовательского интерфейса.

Решение:

`BackgroundWorker` – это класс, предоставленный `System.ComponentModel`. Как было упомянуто ранее, он может быть использован для создания рабочего потока на базе `ThreadPool`. `BackgroundWorker` также умеет сообщать свой статус и поддерживает отмену.

Требуемый сценарий может быть реализован при помощи следующего кода:

```
using System;
using System.ComponentModel;
using System.Text;
using System.Threading;
namespace Ch01 {
  class _5BackgroundWorker {
    static void Main(string[] args) {
      var backgroundWorker = new BackgroundWorker();
      backgroundWorker.WorkerReportsProgress = true;
      backgroundWorker.WorkerSupportsCancellation = true;
      backgroundWorker.DoWork += SimulateServiceCall;
      backgroundWorker.ProgressChanged += ProgressChanged;
      backgroundWorker.RunWorkerCompleted += RunWorkerCompleted;
      backgroundWorker.RunWorkerAsync();
      Console.WriteLine("To Cancel Worker Thread Press C.");
      while (backgroundWorker.IsBusy) {
        if (Console.ReadKey(true).KeyChar == 'C') {
          backgroundWorker.CancelAsync();
        }
      }
    }
    //Этот метод выполняется, когда завершается фоновый поток
    private static void RunWorkerCompleted(object sender, RunWorkerCompletedEventArgs e) {
      if (e.Error != null) {
        Console.WriteLine(e.Error.Message);
      } else
        Console.WriteLine($"Result from service call is {e.Result}");
    }
    //Этот метод выполняется, когда фоновый поток
    //сообщает о прогрессе
    private static void ProgressChanged(object sender, ProgressChangedEventArgs e) {
      Console.WriteLine($"{e.ProgressPercentage}% completed");
    }
    //Обращение к сервису, который мы хотим симулировать
    private static void SimulateServiceCall(object sender, DoWorkEventArgs e) {
```

```
      var worker = sender as BackgroundWorker;
      StringBuilder data = new StringBuilder();
      //Симулировать обращение к потоковому сервису
      for (int i = 1; i <= 100; i++) {
        //worker.CancellationPending будет true, если пользователь
        //нажмет C
        if (!worker.CancellationPending) {
          data.Append(i);
          worker.ReportProgress(i);
          Thread.Sleep(100);
          //Можете раскомментировать строку ниже и посмотреть на
          //исключительную ситуацию
          //throw new Exception("Some Error has occurred");
        } else {
          //Отменяет работу фонового потока
          worker.CancelAsync();
        }
      }
      e.Result = data;
    }
  }
}
```

Класс `BackgroundWorker` является абстракцией над низкоуровневыми потоками, предоставляя разработчику больше контроля и возможностей. Также `BackgroundWorker` отлично подходит для реализации асинхронной модели, основанной на событиях (**Event-Based Asynchronous Pattern**, или **EAP**), и способен более эффективно взаимодействовать с кодом, чем низкоуровневые потоки. Чтобы получать отчеты о ходе выполнения и отмене событий, необходимо установить следующие свойства в значение `true`:

```
backgroundWorker.WorkerReportsProgress = true;
backgroundWorker.WorkerSupportsCancellation = true;
```

Также вам нужно будет установить обработчик на `ProgressChanged` для получения прогресса выполнения работы; на `DoWork` – для передачи метода, выполняющего работу; `RunWorkerCompleted` – для получения результатов работы или сообщения об ошибках:

```
backgroundWorker.DoWork += SimulateServiceCall;
backgroundWorker.ProgressChanged += ProgressChanged;
backgroundWorker.RunWorkerCompleted += RunWorkerCompleted;
```

После данной настройки вы можете запустить исполнение с помощью команды

```
backgroundWorker.RunWorkerAsync();
```

У вас есть возможность в любой момент отменить выполнение потока, вызвав метод `backgroundWorker.CancelAsync()`, который устанавливает свойство `CancellationPending` в рабочем потоке. Также необходимо написать код, который будет постоянно проверять статус отмены и корректно завершать работу.

При отсутствии исключений результат выполнения потока может быть направлен обратно к вызывающему объекту со следующей установкой:

```
e.Result = data;
```

Если в программе есть какие-либо необработанные исключения, они корректно возвращаются вызывающему объекту. Это можно осуществить путем его «обертывания» в `RunWorkerCompletedEventArgs` для передачи в качестве параметра обработчику события `RunWorkerCompleted`.

В следующем разделе мы рассмотрим преимущества и недостатки использования `BackgroundWorker`.

Преимущества и недостатки использования BackgroundWorker

Преимущества использования `BackgroundWorker`:

- потоки могут быть использованы для освобождения основного потока;
- потоки создаются и должным образом поддерживаются при помощи `ThreadPool`;
- способствует корректной и автоматической обработке исключений;
- предоставляет отчетность о ходе выполнения работ, поддерживает отмену команд и логику завершения с использованием событий.

Недостатком использования `BackgroundWorker` является то, что с большим количеством потоков код становится трудно отлаживать и поддерживать.

Многопоточность и многозадачность

Мы рассмотрели работу многопоточности и многозадачности, и нами было обнаружено, что каждый из этих подходов имеет свои преимущества и недостатки. Ниже представлены сценарии, для которых лучше подходит использование многопоточности.

- **Если необходима система, которую легко настроить и остановить**: многопоточность может оказаться полезным свойством в том случае, если имеется процесс с большим ресурсопотреблением. При использовании потоков можно параллельно выполнять необходимые методы, передав в них нужные данные. Если же реализовывать параллельную обработку на основе процессов (запуск новых экземпляров одной и той же программы), то это потребует гораздо больше ресурсов.
- **Если необходимо быстро переключаться между задачами**: кеш-память процессора и программный контекст могут легко сохраняться между потоками в процессе. Если же происходит переключение процессов, то это требует перезагрузки кеша и контекста.
- **Если необходимо обмениваться данными с другими потоками**: внутри процесса все потоки совместно используют общую память, что облегчает обмен данными. Если же осуществлять обмен данными между процессами, то нужны специальные операции ввода-вывода и транспортные протоколы, что требует гораздо больших ресурсов.

В данном разделе мы обсудили основы многопоточности и многозадачности наряду с различными подходами, которые были использованы для создания потоков в более старых версиях .NET. В следующем разделе мы попытаемся разобрать некоторые сценарии, где возможно использование техник параллельного программирования.

Сценарии, при которых полезно параллельное программирование

Ниже представлены сценарии, в которых может быть использовано параллельное программирование.

- **Создание отзывчивого пользовательского интерфейса для приложений с графическим интерфейсом**: мы можем переносить ресурсозатратные задачи на рабочий поток, тем самым предоставляя возможность основному потоку обрабатывать пользовательские взаимодействия и задачи перерисовки интерфейса.
- **Обработка одновременных запросов**: при реализации серверных приложений нам необходимо обрабатывать большое количество параллельных подключений. Мы можем создавать отдельный поток для обработки каждого запроса. Например, можно использовать модель запроса ASP.NET, которая реализуется через ThreadPool (пул потоков) – под каждый поступающий на сервер запрос назначается поток из пула. Затем поток выполняет обработку запроса и отправляет ответ клиенту.
- **Эффективное использование ЦП**: при использовании многоядерных процессоров без многопоточности обычно применяется лишь одно ядро. Можно полноценно использовать ресурсы ЦП, создавая несколько потоков, каждый из которых будет работать на своем ядре. Таким образом, распределение нагрузки приводит к повышению производительности. Это важно для длительных и сложных вычислений, которые можно выполнить быстрее, используя стратегию «разделяй и властвуй».
- **Оценивающие задачи**: сценарии, включающие несколько алгоритмов, например для сортировки входного набора чисел в кратчайшие сроки. Единственный способ сделать это – передать входные данные всем алгоритмам и запустить их параллельно. Принимается тот алгоритм, который первым завершится, а остальные отменяются.

Преимущества и недостатки параллельного программирования

Многопоточность приводит к параллелизму со своими плюсами и минусами. Освоив основные понятия параллельного программирования, обратимся теперь к его преимуществам и недостаткам.

Преимущества параллельного программирования:

- **повышенная производительность**. Можно достигнуть роста производительности, поскольку задачи распределяются по работающим параллельно потокам;
- **улучшенная отзывчивость графического интерфейса**. Так как задачи производят неблокирующий ввод-вывод, поток графического интерфейса всегда может получить входные данные от пользователя. Это приводит к лучшей отзывчивости;
- **одновременный и параллельный запуск задач**. По причине того, что задачи выполняются параллельно, мы можем одновременно запускать несколько алгоритмов;
- **эффективное использование возможностей ЦП, а также кеш-памяти.** Задачи могут выполняться на разных ядрах, что приводит к увеличению пропускной способности.

Параллельное программирование также имеет следующие недостатки:

- **сложные процессы отладки и тестирования**. В связи с параллельной работой потоков их сложно отлаживать без специальных инструментов;
- **издержки на переключение контекста**. Каждый поток работает на выделенном ему отрезке времени. Как только время заканчивается, происходит переключение контекста, что также приводит к неэффективному использованию ресурсов;
- **высокая вероятность возникновения взаимной блокировки**. Если несколько потоков работают с общим ресурсом, то необходима блокировка для обеспечения потокобезопасности. Однако если несколько потоков одновременно блокируют и ждут один и тот же ресурс, это может привести к взаимоблокировкам;
- **трудности в программировании**. Написание параллельных программ может вызвать сложности (по сравнению с синхронными версиями);
- **непредсказуемые результаты**. Поскольку параллельное программирование опирается на процессорные ядра, мы можем получить разные результаты на разных конфигурациях машин.

Следует помнить, что параллельное программирование является относительным понятием. То, что подходит для одних задач, не всегда подойдет остальным, поэтому рекомендуем попробовать этот подход самостоятельно и проверить его на практике.

Резюме

В этой главе мы рассмотрели сценарии, преимущества и особенности параллельного программирования. За последние несколько десятилетий компьютерные системы эволюционировали от одноядерных к многоядерным. Аппаратное обеспечение в чипах также поддерживает Hyper Threading, что дополнительно повышает производительность современных систем.

Прежде чем перейти к параллельному программированию, полезно повторить основные понятия, связанные с ОС, такие как процессы, задачи и разница между многопоточностью и многозадачностью.

В следующей главе мы полностью сосредоточимся на TPL и связанных с ней реализациях. Однако в реальном мире существует много устаревшего кода, который все еще опирается на более старые конструкции, поэтому знание о нем не будет лишним.

Вопросы

1. Многопоточность является подмножеством параллельного программирования.
   1. Да
   2. Нет
2. Сколько ядер будет в однопроцессорной двухъядерной ЭВМ с включенным HyperThreading?
   1. 2
   2. 4
   3. 8
3. Когда приложение завершается, все потоки переднего плана также прекращают работу. Нет необходимости реализовывать логику закрытия потоков переднего плана при выходе из приложения.
   1. Да
   2. Нет
4. Какое исключение возникает, когда поток пытается получить доступ к элементам управления, которые ему не принадлежат (которых он не создавал)?
   1. `ObjectDisposedException`
   2. `InvalidOperationException`
   3. `CrossThreadException`
5. Какой из классов поддерживает отмену команд и предоставляет отчетность о ходе выполнения работ?
   1. `Thread`
   2. `BackgroundWorker`
   3. `ThreadPool`

Глава 2

Параллелизм задач

https://t.me/it_boooks

В предыдущей главе мы познакомились с понятием параллельного программирования. В этой главе мы обсудим TPL и параллелизм задач (tasks).

Одна из основных целей создания .NET была связана с необходимостью предоставить разработчикам единые API-интерфейсы для наиболее частых сценариев, тем самым упростив процесс разработки. Мы уже знаем, что первые версии .NET поддерживали потоки, но эти инструменты были очень сложными и ресурсозатратными. Для упрощения реализации Microsoft внедрила множество улучшений, облегчающих написание параллельных программ с нуля, их отладку и поддержку.

Содержание главы:

- создание и запуск задач;
- получение результата выполнения задачи;
- отмена задачи;
- ожидание выполнения задач;
- обработка исключений в задачах;
- преобразование **модели асинхронного программирования** (Asynchronous Programming Model, APM) в задачи;
- преобразование **асинхронной модели, основанной на событиях (Event-Based Asynchronous Patterns, EAP), в задачи;**
- подробнее о задачах:
  - возобновление задач;
  - родительские и дочерние задачи;
  - локальные и глобальные очереди, хранилище;
  - очереди с перехватом работы.

Технические требования

Для освоения этой главы вам пригодятся знания о C# и современных концепциях языков программирования, например таких, как делегаты (delegates).

Вы найдете исходные коды примеров из этой главы на GitHub: https://github.com/PacktPublishing/Hands-On-Parallel-Programming-with-C-8-and-.NET-Core-3/tree/master/Chapter02.

Задачи

Задачи (tasks) – это абстракции .NET, организующие асинхронный код по аналогии с «обещаниями» (promise, или «промис») JavaScript. В начальных версиях .NET нам приходилось полагаться на потоки, которые создавались либо вручную, либо при помощи класса `ThreadPool`. Хотя класс `ThreadPool` и предоставлял механизмы управления потоками, но разработчикам по-прежнему требовалось обращаться к классу `Thread`. Создавая поток через класс `Thread`, мы получаем доступ к объекту, с помощью которого можно ожидать завершения работы потока, отменять его выполнение или перемещать его на передний план либо в фон. Однако в реальности это требует написания большого объема вспомогательного кода. Также класс `Thread` создает объект низкого уровня, создавая высокую нагрузку на память и центральный процессор. Нам нужно было получить все лучшее из представленных классов, и решение было найдено в использовании задач. Задача представляет собой не что иное, как «заворачивание» потока, которое происходит при помощи `ThreadPool`. Функциональность задач отражена в их способностях ожидания, отмены и продолжения; они начинают работать после завершения других задач.

Важные особенности задач:

- задачи выполняются `TaskScheduler`, при этом планировщик по умолчанию работает на `ThreadPool`;
- мы можем возвращать значения из задач;
- вы знаете момент завершения задачи, в отличие от `ThreadPool` или потоков;
- новая задача может выполняться после завершения текущей с помощью конструкции `ContinueWith()`;
- используя `Task.Wait()`, мы ожидаем выполнения задач. Эта команда блокирует вызывающий поток до тех пор, пока задача не будет завершена;
- задачи делают код более читаемым по сравнению с традиционными потоками или `ThreadPool`. Задачи также положили начало к внедрению конструкции асинхронного программирования в C# 5.0;
- мы можем установить родительские/дочерние связи при запуске одной задачи из другой;
- мы можем передавать исключения дочерних задач родительским;
- задачу можно отменить с помощью класса `CancellationToken`.

Создание и запуск задачи

Существует множество способов для создания и запуска задачи с помощью TPL. В этом разделе мы постараемся рассмотреть все эти подходы и сравнить их по различным параметрам. Для начала вам необходимо добавить ссылку на пространство имен `System.Threading.Tasks`:

```
using System.Threading.Tasks;
```

Мы рассмотрим создание задач при помощи:

- класса `System.Threading.Tasks.Task`;
- метода `System.Threading.Tasks.Task.Factory.StartNew`;
- метода `System.Threading.Tasks.Task.Run`;
- `System.Threading.Tasks.Task.Delay`;
- `System.Threading.Tasks.Task.Yield`;
- `System.Threading.Tasks.Task.FromResult<T> Method`;
- `System.Threading.Tasks.Task.FromException` и `Task.FromException<T>`;
- `System.Threading.Tasks.Task.FromCancelled` и `Task.FromCancelled<T>`.

Класс System.Threading.Tasks.Task

Класс `Task` – способ асинхронного выполнения работы, по аналогии с потоком из `ThreadPool`, и он базируется на **асинхронном шаблоне, основанном на задачах** (Task-Based Asynchronous Pattern, TAP). Базовый класс `Task` не возвращает результаты, поэтому если нам необходимо вернуть значения из задачи, мы обращаемся к общей (generic, дженерик) версии – `Task<T>`. Задачи, созданные классом `Task`, начинают выполняться только после вызова метода `Start`.

Используя класс `Task`, мы можем создавать задачи различными способами, о которых узнаем в следующих подразделах.

Синтаксис лямбда-выражений

Благодаря конструктору класса `Task` мы можем передать лямбда-выражение с нужным кодом для создания задачи:

```
Task task = new Task(() => PrintNumber10Times());
task.Start();
```

Делегат Action

В этом коде мы создаем задачу, вызывая конструктор класса `Task` для передачи делегата, включающего исполняемый метод:

```
Task task = new Task(new Action(PrintNumber10Times));
task.Start();
```

Делегат

При помощи конструктора класса `Task` мы создаем *задачу*, передавая анонимный делегат (`delegate`) с нужным кодом:

```
Task task = new Task(delegate {
  PrintNumber10Times();
});
task.Start();
```

Во всех рассмотренных вариантах результат будет один:

```
1111111111
```

Эти способы работают одинаково, только имеют разный синтаксис.

Метод `Start` подходит лишь для тех задач, которые ранее не выполнялись. Если вам нужно повторно запустить задачу, которая была завершена, тогда нужно создать новую задачу и вызвать метод `Start` для нее.

Метод System.Threading.Tasks.Task.Factory.StartNew

Создать задачу можно также при помощи метода `StartNew` класса `TaskFactory`. При таком подходе создается задача, выполнение которой планируется внутри `ThreadPool`, а ссылка на эту задачу возвращается обратно вызывающему объекту.

Также задачу можно создать методом `Task.Factory.StartNew`. Мы разберем это чуть позже.

Синтаксис лямбда-выражений

В следующем коде мы создаем задачу, вызывая метод `StartNew()` класса `TaskFactory`, и передаем лямбда-выражение с методом, который хотим выполнить:

```
Task.Factory.StartNew(() => PrintNumber10Times());
```

Делегат Action

В данном случае мы создаем задачу при помощи метода `StartNew()` на `TaskFactory` и передаем делегат-обертку над методом, который нужно выполнить, используя класс `Action`:

```
Task.Factory.StartNew(new Action(PrintNumber10Times));
```

Делегат

В следующем коде мы создаем задачу, вызывая метод `StartNew()` и передавая анонимный делегат с нужным кодом:

```
Task.Factory.StartNew(delegate {
  PrintNumber10Times();
});
```

Данные методы работают аналогично, только используют разный синтаксис.

Метод System.Threading.Tasks.Task.Run

Мы также можем создать задачу, используя метод `Task.Run`. Этот метод подобен `StartNew` – он возвращает поток пула (`ThreadPool`).

Создание объекта `Task` можно осуществить при помощи метода `Task.Run` несколькими способами, которые мы разберем в следующих подразделах.

Синтаксис лямбда-выражений

В следующем коде мы создаем задачу, вызывая статический метод `Run()` класса `Task`, и передаем лямбда-выражение, содержащее метод, который мы хотим выполнить:

```
Task.Run(() => PrintNumber10Times());
```

Делегат Action

В данном коде мы создаем задачу с помощью класса `Action`:

```
Task.Run(new Action(PrintNumber10Times));
```

Делегат

В следующем коде мы создаем задачу при помощи анонимного делегата:

```
Task.Run(delegate {
  PrintNumber10Times();
});
```

Метод System.Threading.Tasks.Task.Delay

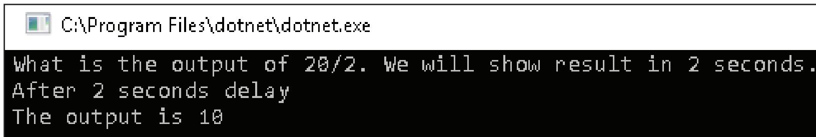
Мы можем создать задачу, которая завершится через определенное время или которую можно будет отменить в любой момент с помощью класса `CancellationToken`. Раньше использовался метод `Thread.Sleep()` класса `Thread` для создания блокирующих конструкций в очереди выполнения потоков. Однако этот метод работал синхронно и все еще использовал ресурсы центрального процессора. `Task.Delay` является лучшей альтернативой ожиданию задач, потому как работает асинхронно без использования циклов процессора.

```
Console.WriteLine("What is the output of 20/2. We will show result
  in 2 seconds.");
Task.Delay(2000);
Console.WriteLine("After 2 seconds delay");
Console.WriteLine("The output is 10");
```

Данный код задает пользователю вопрос и ждет две секунды, перед тем как отобразится ответ. С целью улучшения пользовательского интерфейса в течение этих двух секунд основной поток должен не ждать,

а выполнять другие задачи. С помощью системных часов код выполняется асинхронно, и когда заканчивается время, начинает выполняться остальная часть кода.

Так выглядит вывод предыдущего кода:



Перед тем как мы перейдем к другим методам, которые можно использовать для создания задачи, мы рассмотрим две асинхронные конструкции программирования, которые были введены в C# 5.0: `async` и `await`.

`async` и `await` являются маркерами, облегчающими написание асинхронных программ. Мы подробно рассмотрим эти дескрипторы в главе 9. Ключевое слово `await` (ожидать) позволяет нам ожидать любого асинхронного вызова. В тот момент, когда исполняющий поток видит ключевое слово `await` внутри метода, данный метод возвращается в `ThreadPool`, передает выполнение остальной части метода делегату и начинает выполнять другие задачи в очереди. Как только асинхронная задача выполняется, любой доступный поток из пула завершает оставшуюся часть метода.

Метод System.Threading.Tasks.Task.Yield

Этот метод является еще одним способом создания `Task`, при котором вызывающий код не имеет прямого доступа к задаче. Метод больше напоминает promise из языка JavaScript, чем задачу. При помощи `Task.Yield` мы можем сделать метод асинхронным, самостоятельно вернув управление операционной системе. Даже если остальная часть метода будет выполняется позже, его работа все равно может происходить как в асинхронном коде. Мы можем получить тот же результат, используя следующий код:

```
await Task.Factory.StartNew(() => {},
  CancellationToken.None,
  TaskCreationOptions.None,
  SynchronizationContext.Current != null ?
  TaskScheduler.FromCurrentSynchronizationContext() :
  TaskScheduler.Current);
```

Данный подход позволяет достигнуть лучшей отзывчивости приложений с пользовательским интерфейсом, периодически передавая контроль UI-потоку внутри длительных задач. Однако в приложениях с пользовательским интерфейсом есть и более подходящие методы, например `Application.DoEvents()` в WinForms и `Dispatcher.Yield(DispatcherPriority.ApplicationIdle)` в WPF:

```
private async static void TaskYield() {
  for (int i = 0; i < 100000; i++) {
    Console.WriteLine(i);
    if (i % 1000 == 0)
      await Task.Yield();
  }
}
```

В консольных или веб-приложениях при добавлении точки останова (brakepoint) на методе `Yeld()` мы увидим, что случайные потоки из пула переключают контекст для выполнения нашего кода. На представленных ниже скриншотах показаны потоки, управляющие исполнением нашего кода на разных этапах.

На скриншоте показаны все потоки программы. Мы видим текущий идентификатор потока – **1664**:

Если мы нажмем **F5** и позволим точке останова получить другое значение `i`, то увидим, что код теперь выполняется другим потоком с идентификатором **10244**:

Threads

Search: Search Call Stack Group by: Process ID Columns

		ID	Managed ID	Category	Name	Location
Process ID: 2284 (5 threads)						
		11872	1	Main Thread	Main Thread	System.Console.dll!System.ConsolePal.WindowsConsoleStream.ReadFileNative
		1664	6	Worker Thread	Worker Thread	<not available>
		15296	0	Worker Thread	Worker Thread	<not available>
		10244	5	Worker Thread	Worker Thread	Ch02.dll!Ch02._1CreatingAndStartingTasks.TaskYield
		12116	0	Worker Thread	Worker Thread	<not available>

1CreatingAndStartingTasks.cs

Ch02 Ch02._1CreatingAndStartingTasks

```
28              TaskYield();
29              Console.ReadLine();
30          }
31
32          private async static void TaskYield()
33          {
34              for (int i = 0; i < 100000; i++)
35              {
36                  Console.WriteLine(i);
37                  if (i % 1000 == 0)
38                      await Task.Yield();  ≤ 578ms elapsed
39              }
40          }
```

Подробнее об инструментах и методах отладки многопоточного кода мы поговорим в главе 11.

Метод System.Threading.Tasks.Task.FromResult<T>

Этот подход был представлен в .NET Framework 4.5 и все еще сильно недооценивается. С его помощью мы можем вернуть завершенную задачу с результатами:

```
static void Main(string[] args) {
  StaticTaskFromResultUsingLambda();
}

private static void StaticTaskFromResultUsingLambda() {
  Task < int > resultTask = Task.FromResult < int > (Sum(10));
  Console.WriteLine(resultTask.Result);
}

private static int Sum(int n) {
  int sum = 0;
  for (int i = 0; i < 10; i++) {
    sum += i;
  }
  return sum;
}
```

В этом коде мы преобразовали синхронный метод `Sum` так, чтобы результаты вернулись асинхронно с помощью `Task.FromResult<int>`. Данный подход часто используется в разработке на основе тестов (Test Driven Development, TDD) для эмуляции асинхронных методов, а также внутри них для возврата значений, зависящих от условий. Мы дадим объяснение этим подходам в главе 11.

Методы System.Threading.Tasks.Task.FromException и System.Threading.Tasks.Task.FromException<T>

Эти методы создают завершенные задачи с заранее определенным исключением. Также они могут быть использованы для создания исключений в асинхронных задачах и при написании тестов. Мы объясним этот подход в главе 11:

```
return Task.FromException<long>(
  new FileNotFoundException("Invalid File name."));
```

Как вы могли заметить в предыдущем коде, мы «оборачиваем» `FileNotFoundException` в задачу и возвращаем ее вызывающему объекту.

Методы System.Threading.Tasks.Task.FromCanceled и System.Threading.Tasks.Task.FromCanceled<T>

Эти методы используются для создания задач, завершенных в результате отмены с помощью `CancellationToken`:

```
CancellationTokenSource source = new CancellationTokenSource();
var token = source.Token;
source.Cancel();
Task task = Task.FromCanceled(token);
Task < int > canceledTask = Task.FromCanceled < int > (token);
```

В этом коде мы создали метку (token) отмены, используя класс `CancellationTokenSource`. Потом уже создали задачу из этой метки. Очень важно, чтобы эта метка была отменена, прежде чем мы ее используем в методе `Task.FromCanceled`.

Данный подход будет эффективен в том случае, если нам необходимо вернуть значения из асинхронных методов, а также при написании тестов.

Результаты выполнения задач

В основе TPL лежит общий (generic) вариант для всех классов, позволяющий вернуть значения из задач. С этими классами мы с вами уже познакомились ранее:

- `Task<T>`;
- `Task.Factory.StartNew<T>`;
- `Task.Run<T>`.

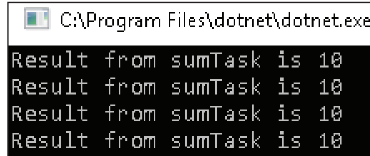
Как только задача завершится, мы сможем получить от нее результат, обратившись к свойству `Task.Result`. Давайте попробуем разобраться с этим на нескольких примерах. Сейчас мы создадим различные задачи и, когда они завершатся, попробуем получить из них значения:

```
using System;
using System.Threading.Tasks;
namespace Ch02 {
  class _2GettingResultFromTasks {
    static void Main(string[] args) {
      GetResultsFromTasks();
      Console.ReadLine();
    }

    private static void GetResultsFromTasks() {
      var sumTaskViaTaskOfInt = new Task < int > (() => Sum(5));
      sumTaskViaTaskOfInt.Start();
      Console.WriteLine($"Result from sumTask is
        {sumTaskViaTaskOfInt.Result}");
      var sumTaskViaFactory = Task.Factory.StartNew < int > (() =>
        Sum(5));
      Console.WriteLine($"Result from sumTask is
        {sumTaskViaFactory.Result}");
      var sumTaskViaTaskRun = Task.Run < int > (() => Sum(5));
      Console.WriteLine($"Result from sumTask is
        {sumTaskViaTaskRun.Result}");
      var sumTaskViaTaskResult = Task.FromResult < int > (Sum(5));
      Console.WriteLine($"Result from sumTask is
        {sumTaskViaTaskResult.Result}");
    }

    private static int Sum(int n) {
      int sum = 0;
      for (int i = 0; i < n; i++) {
        sum += i;
      }
      return sum;
    }
  }
}
```

Как показано выше, мы создали задачи с использованием generic-методов. Как только они завершились, мы смогли получить результаты, используя свойство `Result`:



В следующем разделе мы узнаем о том, как можно отменять задачи.

Отмена задач

Еще одной важной особенностью TPL является то, что она содержит готовые структуры данных для отмены запущенных задач. Для тех из вас, кто сталкивался с классической многопоточностью, знакомы сложности отмены запущенных потоков, однако теперь все иначе. Платформа .NET Framework включает два класса для отмены задач:

- `CancellationTokenSource`: этот класс отвечает за создание меток отмены и передачу всем меткам, созданным через объект этого класса, запроса на отмену;
- `CancellationToken`: используется «слушателями» (listeners) для мониторинга текущего состояния запроса.

Для создания задач, которые впоследствии можно будет отменить, необходимо следующее.

1. Создайте объект класса `System.Threading.CancellationTokenSource`, который предоставит `System.Threading.CancellationToken` через свойство `Token`.
2. При создании задачи передайте метку.
3. При необходимости вызовите метод `Cancel()` у объекта `CancellationTokenSource`.

Давайте попробуем разобраться, как создаются метки и как они передаются в задачи.

Создание метки

Метки можно создать с помощью кода:

```
CancellationTokenSource tokenSource =
  new CancellationTokenSource();
CancellationToken token = tokenSource.Token;
```

Сначала мы создали `tokenSource` при помощи конструктора `CancellationTokenSource`. Затем мы получили нашу метку, используя свойство `Token`.

Создание задач с использованием меток

Мы можем создать задачу, передав в ее конструктор объект `CancellationToken` в качестве второго аргумента, как показано ниже:

```
var sumTaskViaTaskOfInt = new Task<int>(() => Sum(5), token);
var sumTaskViaFactory = Task.Factory.StartNew<int>(() => Sum(5), token);
var sumTaskViaTaskRun = Task.Run<int>(() => Sum(5), token);
```

В классической модели многопоточности мы привыкли использовать метод `Abort()`. Однако отсутствие механизмов управления ресурсами могло привести к резкой остановке потока, вызывая утечку памяти. С помощью TPL мы можем вызвать метод `CancellationTokenSource.Cancel`, который установит на метке свойство `IsCancellationRequested`. Выполняемый задачей код должен отслеживать это свойство и, при его наличии, завершаться корректно.

Отследить, запросил ли `CancellationTokenSource` отмену, можно разными способами:

- проверка состояния свойства `IsCancellationRequested` у метки;
- регистрация обратного вызова (callback) для события отмены.

Опрос состояния метки через свойство IsCancellationRequested

Этот подход применяется в сценариях с рекурсивными методами или другими методами, содержащими длительные циклы. В этом случае мы пишем код, который проверяет `IsCancellationRequested` через конкретные оптимальные интервалы. При наличии этого свойства цикл прерывается, вызывая метод `ThrowIfCancellationRequested` у объекта `token`.

Ниже представлен пример отмены задачи путем опроса метки:

```
private static void CancelTaskViaPoll() {
  CancellationTokenSource cancellationTokenSource =
    new CancellationTokenSource();
  CancellationToken token = cancellationTokenSource.Token;
  var sumTaskViaTaskOfInt = new Task(() =>
    LongRunningSum(token), token);
  sumTaskViaTaskOfInt.Start();
  //Подождите, пока пользователь нажмет кнопку, чтобы отменить
  //задачу
  Console.ReadLine();
  cancellationTokenSource.Cancel();
}

private static void LongRunningSum(CancellationToken token) {
  for (int i = 0; i < 1000; i++) {
    //Имитация длительной работы
    Task.Delay(100);
    if (token.IsCancellationRequested)
      token.ThrowIfCancellationRequested();
  }
}
```

В предыдущем коде мы создали метку отмены с помощью класса `CancellationTokenSource`, а затем задачу, передав ей эту метку. Задача выполняет длительный метод `LongRunningSum`, который опрашивает свойство метки `IsCancellationRequested`. Оно генерирует исключение при условии, если пользователь вызвал `cancellationTokenSource.Cancel()` до завершения метода.

Данная проверка не вызывает значительных издержек в производительности, поэтому вы можете использовать ее в соответствии с вашими потребностями.

Регистрация отмены запроса с помощью делегата обратного вызова

В основе этого подхода лежит делегат обратного вызова (*callback*), который вызывается при запросе отмены. Его следует использовать вместе с операциями, при которых невозможно регулярно проверять значение `CancellationToken`.

Рассмотрим следующий код, который загружает файлы с удаленного URL-адреса:

```
private static void DownloadFileWithoutToken() {
  WebClient webClient = new WebClient();
  webClient.DownloadStringAsync(new Uri("http://www.google.com"));
  webClient.DownloadStringCompleted += (sender, e) => {
    if (!e.Cancelled)
      Console.WriteLine("Download Complete.");
    else
      Console.WriteLine("Download Cancelled.");
  };
}
```

Из предыдущего метода становится очевидно, что при вызове `DownloadStringAsync` из `WebClient` у пользователя пропадает контроль. Несмотря на то что класс `WebClient` позволяет нам отменить задачу через `webClient.CancelAsync()`, теряется контроль над его вызовом.

Предыдущий код можно модифицировать для использования делегата обратного вызова, благодаря которому пользователь получает больше контроля над отменой задачи:

```
static void Main(string[] args) {
  CancellationTokenSource cancellationTokenSource = new
  CancellationTokenSource();
  CancellationToken token = cancellationTokenSource.Token;
  DownloadFileWithToken(token);
  //Случайная задержка перед отменой метки
  Task.Delay(2000);
  cancellationTokenSource.Cancel();
  Console.ReadLine();
}

private static void DownloadFileWithToken(CancellationToken token) {
```

```
  WebClient webClient = new WebClient();
  //Здесь мы регистрируем делегат обратного вызова, который будет
  //вызываться, как только пользователь отменит метку
  token.Register(() => webClient.CancelAsync());
  webClient.DownloadStringAsync(new Uri("http://www.google.com"));
  webClient.DownloadStringCompleted += (sender, e) => {
    //Подождите 3 секунды, чтобы у нас было достаточно времени
    //для отмены задачи
    Task.Delay(3000);
    if (!e.Cancelled)
      Console.WriteLine("Download Complete.");
    else
      Console.WriteLine("Download Cancelled.");
  };
}
```

Как вы уже могли заметить, в этой измененной версии мы передали метку отмены и подписались на событие отмены путем метода `Register`.

Как только пользователь вызовет метод `cancellationTokenSource.Cancel()`, произойдет отмена загрузки посредством `webClient.CancelAsync()`.

`CancellationTokenSource` хорошо работает с устаревшим `ThreadPool.QueueUserWorkItem`.

Ниже представлен созданный `CancellationTokenSource` код, который может быть передан в `ThreadPool` для реализации отмены:

```
// Создание нового источника метки
CancellationTokenSource cts = new CancellationTokenSource();
// Передача метки в операцию, поддерживающую отмену
ThreadPool.QueueUserWorkItem(new WaitCallback(DoSomething), cts.Token );
```

В данном разделе мы рассмотрели различные способы отмены задач. Она ощутимо помогает экономить время работы процессора при наличии большого количества задач. Предположим, например, что мы создали несколько задач для сортировки списка чисел с использованием различных алгоритмов. Несмотря на то что все алгоритмы будут выдавать один и тот же результат (в нашем случае сортированный список чисел), для нас важна скорость его получения. Мы примем результат самого быстрого алгоритма и отменим остальные, улучшая тем самым производительность системы. В следующем разделе мы обсудим ожидание выполнения задач.

Ожидание выполнения задач

Ранее в примерах мы получали результат завершенной задачи из свойства `Task.Result`, которое блокировало поток до момента получения результата. Еще одним способом ожидания выполнения задач (одной или нескольких) является TPL.

В TPL представлены различные механизмы ожидания выполнения задач:

- `Task.Wait`;
- `Task.WaitAll`;
- `Task.WaitAny`;
- `Task.WhenAll`;
- `Task.WhenAny`.

Эти методы будут рассмотрены в следующих подразделах.

Task.Wait

Task.Wait – метод экземпляра Task для ожидания одной задачи. Он позволяет обозначить предельный объем времени, в рамках которого вызывающий оператор ожидает завершения задачи. Мы также можем передать методу метку отмены. Вызывающий метод будет заблокирован до тех пор, пока поток не завершится, не будет отменен или не выдаст исключение:

```
var task = Task.Factory.StartNew(() =>
  Console.WriteLine("Inside Thread"));
//Блокирует текущий поток до тех пор, пока задача не завершится.
task.Wait();
```

Существует пять перегруженных версий метода `Wait`:

- `Wait()`: бесконечно ждет завершения задачи. Вызывающий поток блокируется до тех пор, пока не завершится дочерний;
- `Wait(CancellationToken)`: бесконечное ожидание выполнения задачи или отмены с помощью метки;
- `Wait(int)`: ожидает завершения выполнения задачи в течение заданного периода времени, в миллисекундах;
- `Wait(TimeSpan)`: ожидает завершения выполнения задачи в течение заданного интервала времени;
- `Wait (int, CancellationToken)`: ожидает завершения выполнения задачи в течение заданного периода времени (в миллисекундах) или при отмене с помощью метки.

Task.WaitAll

Этот статический метод класса `Task` используется для ожидания нескольких задач. Задачи передаются методу в виде массива, и вызывающий оператор блокируется до тех пор, пока все задачи не будут выполнены. Также данный метод поддерживает метки отмены и тайм-аут. Ниже приведен пример кода метода:

```
Task taskA = Task.Factory.StartNew(() =>
  Console.WriteLine("TaskA finished"));
Task taskB = Task.Factory.StartNew(() =>
  Console.WriteLine("TaskB finished"));
Task.WaitAll(taskA, taskB);
Console.WriteLine("Calling method finishes");
```

Вывод предыдущего кода выглядит так:

```
TaskB finished
TaskA finished
Calling method finishes
```

Как видите, «Calling method finishes» выводится, когда обе задачи завершаются.

Например, мы можем использовать этот метод, когда нам нужны данные из нескольких источников (имеется одна задача для каждого источника) и мы хотим объединить данные из всех задач так, чтобы они могли отображаться в пользовательском интерфейсе.

Task.WaitAny

Это еще один статический метод, определенный в классе `Task`. Подобно `WaitAll`, `WaitAny` используется для ожидания нескольких задач, однако как только любая из задач, передаваемых в виде массива, завершит выполнение, вызывающий поток будет разблокирован. Как и другие методы, `WaitAny` поддерживает метки отмены и тайм-аут. Ниже приведен пример кода на базе этого метода.

```
Task taskA = Task.Factory.StartNew(() =>
  Console.WriteLine("TaskA finished"));
Task taskB = Task.Factory.StartNew(() =>
  Console.WriteLine("TaskB finished"));
Task.WaitAny(taskA, taskB);
Console.WriteLine("Calling method finishes");
```

В предыдущем коде мы запустили две задачи и ожидали их с помощью метода `WaitAny`, который блокировал текущий поток. При завершении любой из задач вызывающий поток разблокируется. Например, мы можем обратиться к такому способу, когда нам нужно срочно получить доступные данные из разных источников. В таком случае мы создаем задачи, которые делают к ним запросы. Как только любая задача завершается, мы разблокируем вызывающий поток и получаем результат.

Task.WhenAll

Вариация метода `WaitAll`, но без блокирования вызывающего потока. Этот метод возвращает задачу, которая запускает ожидание для всех указанных задач.

В отличие от `WaitAll`, при котором блокируется вызывающий поток, `WhenAll` может ожидать выполнения внутри асинхронного метода, освобождая вызывающий поток для выполнения других операций. Ниже приведен пример кода с использованием этого метода:

```
Task taskA = Task.Factory.StartNew(() =>
  Console.WriteLine("TaskA finished"));
Task taskB = Task.Factory.StartNew(() =>
  Console.WriteLine("TaskB finished"));
Task.WhenAll(taskA, taskB);
Console.WriteLine("Calling method finishes");
```

Работа данного кода аналогична работе `Task.WaitAll`, различаясь только в том, что вызывающий поток продолжает работу, а не блокируется.

Task.WhenAny

Это вариант `WaitAny`, но в отличие от него он не блокирует вызывающий поток. Этот метод возвращает задачу, ожидающую завершения любой из задач, переданных в качестве аргументов. Ниже приведен пример кода с использованием этого метода:

```
Task taskA = Task.Factory.StartNew(() =>
  Console.WriteLine("TaskA finished"));
Task taskB = Task.Factory.StartNew(() =>
  Console.WriteLine("TaskB finished"));
Task.WhenAny(taskA, taskB);
Console.WriteLine("Calling method finishes");
```

Работа этого кода подобна методу `Task.WaitAny`, однако отличается тем, что вызывающий поток не блокируется, а продолжает выполнение.

В этом разделе мы рассмотрели способы написания эффективного кода для работы с несколькими потоками. В следующем разделе разберем работу задач с исключениями.

Обработка исключений в задачах

Обработка исключений является одним из наиболее важных аспектов параллельного программирования. Хорошие практики чистого кода ориентированы на эффективную обработку исключений. В параллельном программировании это особенно важно, поскольку любые необработанные исключения в потоках или задачах могут привести к внезапному сбою приложения. К счастью, TPL располагает надежными и эффективными конструкциями для управления исключениями. Любые необработанные исключения откладываются и затем передаются в объединяющий поток, который их обрабатывает.

Когда внутри задачи возникает исключение, оно получает обертку из объекта класса `AggregateException` и возвращается вызывающему потоку, обрабатывающему исключения. Если вызывающий оператор ожидает выполнения единственной задачи, то внутреннее свойство `Exception` класса `AggregateException` вернет единственное исключение. Однако если вызывающий оператор ожидает выполнения нескольких задач с помощью `Task.WaitAll`, `Task.`

WhenAll, Task.WaitAny или Task.WhenAny, все исключения, возникающие в этих задачах, возвращаются вызывающему объекту в виде коллекции. Они могут быть доступны через свойство InnerExceptions.

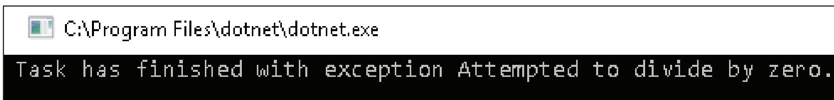
Теперь давайте рассмотрим различные способы обработки исключений внутри задач.

Обработка исключений из одиночных задач

В следующем коде мы создаем простую задачу, которая пытается разделить число на 0, тем самым вызывая исключение DivideByZeroException. Исключение возвращается вызывающему объекту и обрабатывается внутри блока catch. Потому как это единственная задача, объект исключения «оборачивается» в свойство InnerException объекта AggregateException:

```
class _4HandlingExceptions {
  static void Main(string[] args) {
    Task task = null;
    try {
      task = Task.Factory.StartNew(() => {
        int num = 0, num2 = 25;
        var result = num2 / num;
      });
      task.Wait();
    } catch (AggregateException ex) {
      Console.WriteLine($"Task has finished with exception
        {ex.InnerException.Message}");
      }
      Console.ReadLine();
    }
  }
```

Ниже представлен вывод при запуске кода:



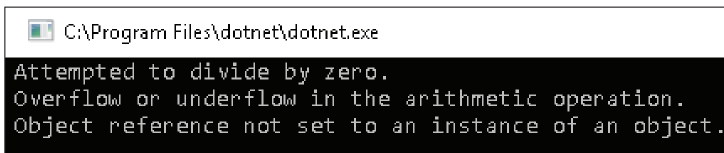
Обработка исключений из нескольких задач

Сейчас мы создадим несколько задач, а затем попытаемся сгенерировать в них исключения. После этого проверим, как можно получить все исключения из задач в вызывающем потоке:

```
static void Main(string[] args) {
  Task taskA = Task.Factory.StartNew(() =>
    throw new DivideByZeroException());
  Task taskB = Task.Factory.StartNew(() =>
    throw new ArithmeticException());
```

```
  Task taskC = Task.Factory.StartNew(() =>
    throw new NullReferenceException());
  try {
    Task.WaitAll(taskA, taskB, taskC);
  } catch (AggregateException ex) {
    foreach (Exception innerException in ex.InnerExceptions) {
      Console.WriteLine(innerException.Message);
    }
  }
  Console.ReadLine();
}
```

Вот результат, который мы получаем при запуске предыдущего кода:



В предыдущем коде были прописаны три задачи, генерирующие разные исключения, при этом их завершение ожидалось при помощи `Task.WaitAll`. Метод WaitAll вернет себе управление, когда все переданные ему задачи завершатся путем создания исключений. После этого соответствующий блок `catch` будет завершен. Выполнив проход по элементам свойства `InnerExceptions` класса `AggregateException`, мы сможем обнаружить все возникшие в задачах исключения.

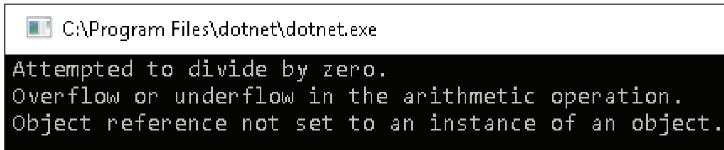
Обработка исключений задач с помощью обратного вызова

Исключения также можно обработать при помощи функции обратного вызова, которая позволяет получить доступ к исключениям в задачах.

```
static void Main(string[] args) {
  Task taskA = Task.Factory.StartNew(() =>
    throw new DivideByZeroException());
  Task taskB = Task.Factory.StartNew(() =>
    throw new ArithmeticException());
  Task taskC = Task.Factory.StartNew(() =>
    throw new NullReferenceException());
  try {
    Task.WaitAll(taskA, taskB, taskC);
  } catch (AggregateException ex) {
    ex.Handle(innerException => {
      Console.WriteLine(innerException.Message);
      return true;
    });
```

```
  }
  Console.ReadLine();
}
```

Ниже представлен результат запуска предыдущего кода в Visual Studio:



Обратившись к предыдущему коду, станет ясно, что вместо прохода по элементам `InnerExceptions` мы подписались на функцию обратного вызова дескриптора `AggregateException`. Это работает для всех задач, которые генерируют исключения. Также мы вернули `true` из нашего обработчика, указывая на то, что исключение было правильно обработано.

Преобразование шаблонов APM в задачи

Устаревшая модель асинхронного программирования (Asynchronous Programming Model, APM) использовала интерфейс IAsyncResult для создания асинхронных методов с образцом проектирования посредством двух методов: BeginXXX и EndXXX. Давайте попробуем проследить путь программы от синхронности к APM, а затем к задаче.

Ниже представлен синхронный метод, который считывает данные из текстового файла:

```
private static void ReadFileSynchronously() {
  string path = @ "Test.txt";
  //Открыть файл и прочитать его содержимое
  using(FileStream fs = File.OpenRead(path)) {
    byte[] b = new byte[1024];
    UTF8Encoding encoder = new UTF8Encoding(true);
    fs.Read(b, 0, b.Length);
    Console.WriteLine(encoder.GetString(b));
  }
}
```

В предыдущем коде нет ничего необычного. Сначала мы создали объект `FileStream` и вызвали метод `Read`, который синхронно считывает файл с диска в буфер, а затем выводит его содержимое в консоль. Также мы преобразовали буфер в строку с помощью класса `UTF8Encoding`. Однако проблема в том, что поток блокируется, пока не выполнится вызов `Read`. Операции ввода-вывода управляются ЦП с помощью циклов процессора, поэтому нет никакого смысла в ожидании завершения этих операций. Давайте попробуем разобрать, как это можно реализовать с помощью APM.

```
private static void ReadFileUsingAPMAsyncWithoutCallback() {
  string filePath = @ "Test.txt";
  //Открыть файл и прочитать его содержимое
  using(FileStream fs = new FileStream(filePath,
    FileMode.Open, FileAccess.Read, FileShare.Read,
    1024, FileOptions.Asynchronous)) {
    byte[] buffer = new byte[1024];
    UTF8Encoding encoder = new UTF8Encoding(true);
    IAsyncResult result = fs.BeginRead(buffer, 0,
      buffer.Length, null, null);
    Console.WriteLine("Do Something here");
    int numBytes = fs.EndRead(result);
    fs.Close();
    Console.WriteLine(encoder.GetString(buffer));
  }
}
```

В этом коде мы заменили метод синхронного чтения асинхронной версией BeginRead. В тот момент, когда транслятор сталкивается с BeginRead, в процессор поступает команда для начала чтения файла, при этом поток разблокируется. В рамках этого же метода могут выполняться и другие задачи, прежде чем поток будет заблокирован вызовом EndRead, который помогает дождаться завершения операции чтения и получить результат. Несмотря на блокирование потока при получении результата, данный подход к созданию адаптивных приложений является весьма эффективным. Альтернативой EndRead выступает версия метода BeginRead, принимающая ссылку на метод, который вызывается автоматически после завершения операции чтения, минуя блокировку потока. Сигнатура этого метода выглядит следующим образом:

```
public override IAsyncResult BeginRead(
  byte[] array,
  int offset,
  int numBytes,
  AsyncCallback userCallback,
  object stateObject)
```

Таким образом, мы перешли от синхронного метода к APM. Теперь преобразуем реализацию APM в задачу (пример кода ниже).

```
private static void ReadFileUsingTask() {
    string filePath = @ "Test.txt";
    //Открыть файл и прочитать его содержимое
    using(FileStream fs = new FileStream(filePath, FileMode.Open,
      FileAccess.Read, FileShare.Read, 1024,
      FileOptions.Asynchronous)) {
      byte[] buffer = new byte[1024];
      UTF8Encoding encoder = new UTF8Encoding(true);
      //Запустить задачу, которая асинхронно прочитает файл
      var task = Task < int > .Factory.FromAsync(fs.BeginRead,
        fs.EndRead, buffer, 0, buffer.Length, null);
      Console.WriteLine("Do Something while file is read
```

```
        asynchronously ");
        //Дождитесь завершения задачи
        task.Wait(); Console.WriteLine(encoder.GetString(buffer));
      }
    }
```

В этом коде мы заменили метод `BeginRead` на `Task<int>.Factory.FromAsync`. Это один из способов реализации TAP. Задача, выполняемая в фоновом режиме, возвращается методу, пока мы продолжаем выполнять другую работу. Это происходит перед блокировкой потока и получением результатов путем `task.Wait()`. Именно так вы можете легко конвертировать любой код из APM в TAP.

Преобразование EAP в задачи

Асинхронная модель на основе событий (Event-based Asynchronous Pattern, EAP) используется для создания компонентов, которые реализуют трудоемкие операции с большими накладными расходами. По этим причинам нужно сделать их асинхронными. Данный шаблон был использован в .NET Framework для создания таких компонентов, как BackgroundWorker и WebClient. Методы реализации этого шаблона выполняют длительные задачи асинхронно в фоновом режиме, но продолжают уведомлять пользователя о своем прогрессе и состоянии с помощью событий, поэтому их и называют «основанными на событиях» (event-based).

Далее показана реализация компонента на основе EAP:

```
private static void EAPImplementation() {
  var webClient = new WebClient();
  webClient.DownloadStringCompleted += (s, e) => {
    if (e.Error != null)
      Console.WriteLine(e.Error.Message);
    else if (e.Cancelled)
      Console.WriteLine("Download Cancel");
    else
      Console.WriteLine(e.Result);
  };
  webClient.DownloadStringAsync(new Uri("http://www.someurl.com"));
}
```

В представленном выше коде мы подписались на событие `DownloadString-Completed`, которое запускается после того, как `webClient` загрузит файл из URL-адреса. Используя конструкцию if-else, мы попытались прочитать различные варианты выполнения, такие как исключение, отмена и результат. По сравнению с преобразованием в APM, преобразование из EAP в TAP – это довольно сложный процесс. Для его реализации вам понадобится уверенное понимание внутренней природы компонентов EAP, ведь необходимо подключать обработчики к правильным событиям, чтобы он работал. Давайте посмотрим на такую преобразованную реализацию:

```
private static Task<string> EAPToTask() {
  var taskCompletionSource = new TaskCompletionSource<string>();
  var webClient = new WebClient();
  webClient.DownloadStringCompleted += (s, e) => {
    if (e.Error != null)
      taskCompletionSource.TrySetException(e.Error);
    else if (e.Cancelled)
      taskCompletionSource.TrySetCanceled();
    else
      taskCompletionSource.TrySetResult(e.Result);
  };
  webClient.DownloadStringAsync(new Uri("http://www.someurl.com"));
  return taskCompletionSource.Task;
}
```

Самым простым способом преобразования EAP в TAP является использование класса `TaskCompletionSource`. Мы проверили все сценарии и установили результат, исключение или отмену в экземпляр класса `TaskCompletionSource`. Затем мы вернули пользователю «обертку» реализации в виде задачи.

И еще о задачах

Давайте теперь познакомимся еще с несколькими важными понятиями в контексте задач, которые могут нам пригодиться. До сих пор мы создавали только самостоятельные задачи. Однако для более сложных решений нам иногда приходится создавать связи между задачами. Для этого мы можем создавать подзадачи, дочерние задачи, а также цепочки задач. Давайте попробуем разобраться с ними на примерах. Немного позже в этом разделе мы узнаем о хранении потоков и очередях.

Цепочки задач

Цепочки задач по своему функционированию очень похожи на промисы. Мы можем использовать их, когда нам нужно связать в цепочку несколько задач. Сразу же после завершения первой задачи начинает выполняться вторая задача, при этом результат или исключение первой задачи передается второй. Можно связывать несколько задач для создания длинной цепочки или же создать сложную выборку с помощью методов TPL.

Конструкции, которые предоставляются TPL для создания цепочек задач:

- `Task.ContinueWith`;
- `Task.Factory.ContinueWhenAll`;
- `Task.Factory.ContinueWhenAll<T>`;
- `Task.Factory.ContinueWhenAny`;
- `Task.Factory.ContinueWhenAny<T>`.

Продолжение выполнения задач с помощью метода Task.ContinueWith

Цепочку задач можно легко создать методом `ContinueWith` из библиотеки TPL.

Давайте попробуем разобраться в этом на примере простой цепочки:

```
var task = Task.Factory.StartNew < DataTable > (() => {
  Console.WriteLine("Fetching Data");
  return FetchData();
}).ContinueWith((e) => {
  var firstRow = e.Result.Rows[0];
  Console.WriteLine("Id is {0} and Name is {0}",
    firstRow["Id"], firstRow["Name"]);
});
```

В примере выше мы извлекли и отобразили данные. **Первая задача** вызывает метод `FetchData`, и когда он завершается, результат передается в качестве входных данных во **вторую задачу**, которая отвечает за печать данных.

Результат выглядит следующим образом:

```
Fetching Data
Id is 1 and Name is 1
```

Мы также можем связывать несколько задач воедино, создавая таким образом из них цепочку, как показано в примере:

```
var task = Task.Factory.StartNew < int > (() => GetData()).
  .ContinueWith((i) => GetMoreData(i.Result)).
  .ContinueWith((j) => DisplayData(j.Result)));
```

Можно осуществлять контроль выполнения последующих задач, передавая перечисление `System.Threading.Tasks.TaskContinuationOptions` в качестве параметра одного из следующих значений:

- `None`: опция по умолчанию, при которой новая задача запускается сразу же после завершения предыдущей задачи в цепочке;
- `OnlyOnRanToCompletion`: новая задача запускается только после успешного завершения предыдущей, если она не была отменена или остановлена с ошибкой;
- `NotOnRanToCompletion`: новая задача запускается только тогда, когда предыдущая задача отменяется или остановлена с ошибкой;
- `OnlyOnFaulted`: новая задача выполняется только тогда, когда предыдущая задача была остановлена с ошибкой;
- `NotOnFaulted`: новая задача будет выполняться только в том случае, если предыдущая не была остановлена с ошибкой;
- `OnlyOnCancelled`: новая задача будет выполняться только после отмены предыдущей;

- `NotOnCancelled`: новая задача будет выполняться при условии, если не была отменена предыдущая.

Продолжение выполнения задач с помощью Task.Factory.ContinueWhenAll и Task.Factory.ContinueWhenAll<T>

Можно ожидать несколько задач сразу и создавать цепочку кода, который будет выполняться только тогда, когда все задачи будут успешно завершены. Перейдем от теории к примеру.

```
private async static void ContinueWhenAll() {
  int a = 2, b = 3;
  Task<int> taskA = Task.Factory.StartNew<int>(() => a * a);
  Task<int> taskB = Task.Factory.StartNew<int>(() => b * b);
  Task<int> taskC = Task.Factory.StartNew<int>(() => 2 * a * b);
  var sum = await Task.Factory.ContinueWhenAll<int>(new Task[] {
    taskA,
    taskB,
    taskC
  }, (tasks) => tasks.Sum(t => (t as Task<int>).Result));
  Console.WriteLine(sum);
}
```

В предыдущем коде мы решаем выражение `a*a + b*b +2 *a *b`. Для этого задача разбивается на три части: `a*a`, `b*b` и `2*a*b`. Каждая из этих частей (блоков) выполняется тремя различными потоками: `TaskA`, `TaskB` и `TaskC`. После завершения выполнения всех задач мы передаем их в качестве первого параметра методу `ContinueWhenAll`. После завершения работы потоков выполняется делегат продолжения, который задается вторым параметром метода `ContinueWhenAll`. Этот делегат суммирует результаты от всех потоков и возвращает их вызывающему оператору, который печатается в следующей строке.

Продолжение выполнения задач с помощью Task.Factory.ContinueWhenAny и Task.Factory.ContinueWhenAny<T>

Эти методы делают возможным ожидание нескольких задач и цепочек в коде, который запускается при условии успешного завершения любой из задач:

```
private static void ContinueWhenAny() {
  int number = 13;
  Task<bool> taskA = Task.Factory.StartNew<bool>(() =>
    number / 2 != 0);
  Task<bool> taskB = Task.Factory.StartNew<bool>(() =>
    (number / 2) * 2 != number);
  Task<bool> taskC = Task.Factory.StartNew<bool>(() =>
    (number & 1) != 0);
  Task.Factory.ContinueWhenAny<bool>(new Task<bool>[] {
    taskA,
```

```
    taskB,
    taskC
  }, (task) => {
    Console.WriteLine((task as Task<bool>).Result);
  });
}
```

Из кода выше следует, что у нас есть три логических элемента, которые помогают проверить, является ли число нечетным. Предположим, что мы не знаем, какой из логических элементов будет самым быстрым. Для вычисления результата мы создаем три задачи и одновременно их запускаем. Каждая из этих задач реализует различную логику определения нечетных чисел. Поскольку число не может быть четным или нечетным в одно и то же время, полученный результат от потоков будет одинаковым с разницей лишь в скорости выполнения. Поэтому логичнее будет просто получить первый результат, отвергнув остальные. Именно этого мы и добились в предыдущем коде, используя метод `ContinueWhenAny`.

Родительские и дочерние задачи

Другим типом отношений между потоками является взаимосвязь «родитель – дочерний элемент». Дочерняя задача создается в качестве вложенной задачи внутри родительской. Дочерняя задача может создаваться как присоединенная (attached) или отсоединенная (detached). Оба типа задач создаются внутри родительской, однако по умолчанию все созданные задачи являются отсоединенными. Есть способ сделать задачу присоединенной, установив для нее свойство `AttachedToParent` в значении `true`. Сценарии, при которых может потребоваться создание присоединенной задачи:

- все исключения, создаваемые в дочерней задаче, должны быть переданы родительской задаче;
- статус родительской задачи зависит от дочерней задачи;
- родительская задача должна дождаться завершения дочерней.

Создание отсоединенной задачи

Код для создания отсоединенной задачи выглядит следующим образом:

```
Task parentTask = Task.Factory.StartNew(() => {
  Console.WriteLine(" Parent task started");
  Task childTask = Task.Factory.StartNew(() => {
    Console.WriteLine(" Child task started");
  });
  Console.WriteLine(" Parent task Finish");
});
//Ожидание завершения родительской задачи
parentTask.Wait();
Console.WriteLine("Work Finished");
```

В представленном выше коде мы создали еще одну задачу в теле основной задачи. Дочерняя (или вложенная) задача по умолчанию создается как отсоединенная. Мы дождались завершения родительской задачи посредством `parentTask.Wait()`. В выводе можно увидеть, что родительская задача не ждет завершения дочерней и завершается первой, а затем уже дочерняя задача начинает свое выполнение:

```
Parent task started
Parent task Finish
Work Finished
Child task started
```

Создание присоединенной задачи

Присоединенная задача создается так же, как и отсоединенная. Единственное отличие заключается в том, что мы устанавливаем свойство `AttachedParent` для задачи в значении `true`, как показано в примере:

```
Task parentTask = Task.Factory.StartNew(() => {
  Console.WriteLine("Parent task started");
  Task childTask = Task.Factory.StartNew(() => {
    Console.WriteLine("Child task started");
  }, TaskCreationOptions.AttachedToParent);
  Console.WriteLine("Parent task Finish");
});
//Ожидание завершения родительской задачи
parentTask.Wait();
Console.WriteLine("Work Finished");
```

Результат выполнения этого кода представлен ниже:

```
Parent task started
Parent task Finish
Child task started
Work Finished
```

Из примера видно, что родительская задача не завершается до тех пор, пока не завершится выполнение дочерней.

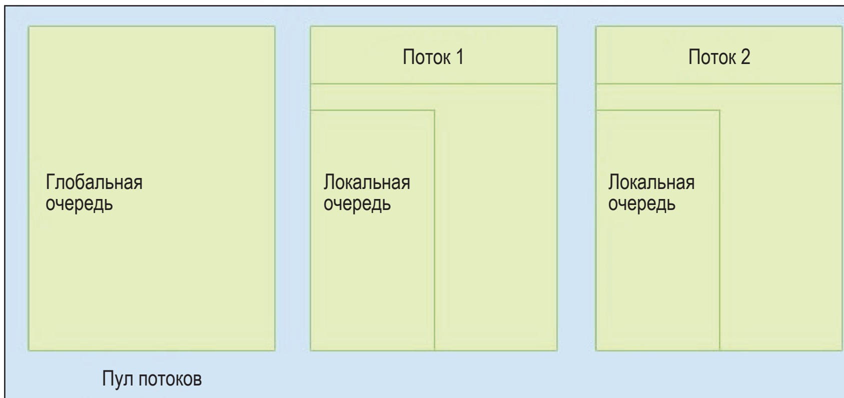
В этом разделе мы обсудили дополнительные аспекты работы с задачами, включая создание связей между ними. В следующей главе мы подробнее рассмотрим то, как задачи работают изнутри, уделяя внимание концепциям очередей и тому, как задачи справляются с ними.

Очереди с перехватом работы

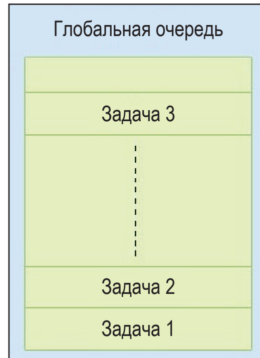
Перехват работы (work-stealing) – это техника оптимизации производительности пула потоков. Каждый пул сопровождается единой глобальной очередью задач, которые создаются внутри процесса. В главе 1 мы узнали, что каждый пул поддерживает оптимальное количество потоков для работы над задачами. Аналогично этому `ThreadPool` также поддерживает глобальную очередь потоков, выстраивая элементы в порядке очередности для их последующего выполнения доступными потоками. Так как мы имеем многопоточные сценарии и только одну очередь, то только примитивы синхронизации позволяют реализовать потокобезопасность. Если у нас будет лишь одна глобальная очередь, то это приведет к потере производительности.

Платформа .NET Framework минимизирует потери производительности, вводя понятие локальных очередей для управления потоками. У каждого потока есть доступ как к глобальной, так и к собственной локальной очереди, которая необходима для хранения рабочих элементов. Родительские задачи могут планироваться внутри глобальной очереди. В процессе выполнения задач может возникнуть необходимость в подзадачах, в таком случае сами задачи собираются в локальные очереди и, когда поток завершается, обрабатываются с помощью алгоритма FIFO (First In, First Out – первый пришел, первый ушел).

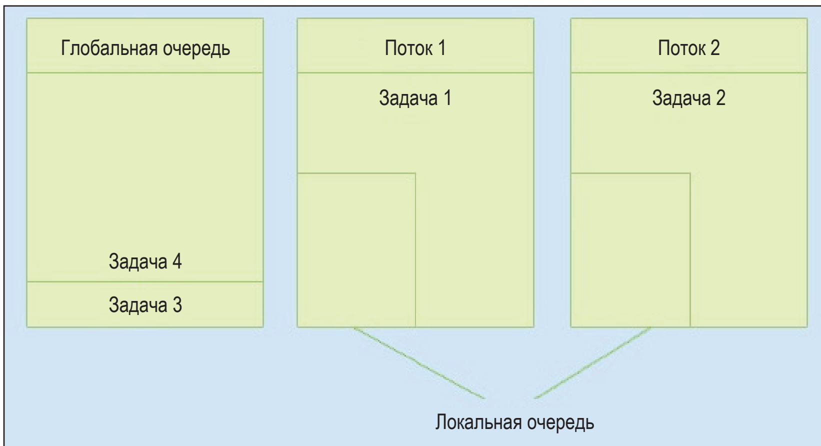
На следующей диаграмме показана связь между потоками, `ThreadPool`, глобальной и локальной очередями.



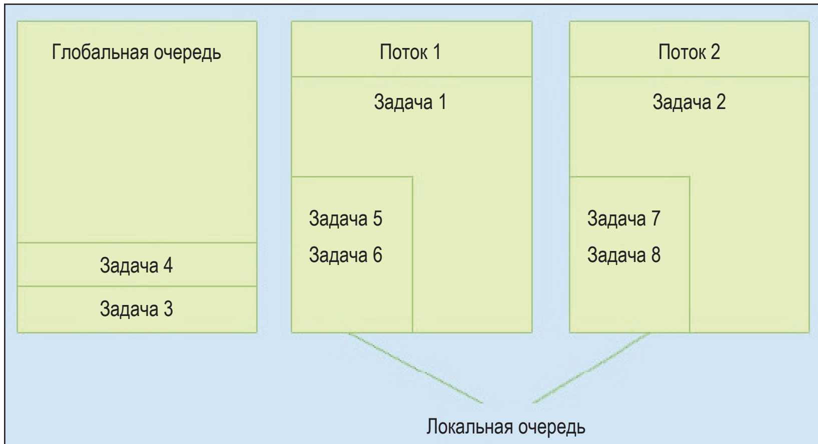
Допустим, что основной поток создает несколько задач. Все они помещаются в глобальную очередь для последующего выполнения. Ниже схематично представлена глобальная очередь с задачами.



Предположим, что **Задача 1** запланирована в **Потоке 1, Задача 2** – в **Потоке 2** и т. д., как показано ниже.



Если **Задача 1** и **Задача 2** создают дополнительные задачи, то новые задачи будут храниться в локальной очереди потоков (это изображено на диаграмме).



По аналогии с вышеуказанной схемой, при создании дочерними задачами вложенных подзадач последние будут помещены в локальную очередь, а не в глобальную. Как только **Поток 1** закончит **Задачу 1**, то он обратится к своей локальной очереди и выберет из нее последнюю задачу (LIFO, Last In, First Out – последним пришел, первым ушел). Высока вероятность того, что последняя задача располагается в кеше, поэтому ее не нужно перезагружать. Повторюсь, что это повышает производительность.

Как только у потока 1 не останется задач в локальной очереди, он обратится к глобальной очереди. Если же в ней не окажется элементов, он будет искать в локальных очередях другие потоки (скажем, поток 2). Эта техника называется перехватом работы (work-stealing) и представляет собой способ оптимизации. На сей раз поток 1 выбирает не последнюю задачу (LIFO) из потока 2, так как есть вероятность, что последний элемент находится в кеше потока 2. Вместо этого он выбирает первую задачу (FIFO). Таким алгоритмом повышается производительность, делая кешированные задачи доступными для локального потока, а задачи вне кеша – для других потоков.

Резюме

В этой главе мы обсудили разбиение задач на более мелкие блоки, для того чтобы каждый элемент мог быть независимо обработан свободным потоком. Мы также узнали о различных способах создания задач на базе `ThreadPool`. Вашему вниманию были представлены различные подходы, связанные с внутренней реализацией задач, включая понятия перехвата работы и создания/отмены задач. Мы будем возвращаться к материалам этой главы на протяжении всей книги.

В следующей главе мы познакомимся с некоторыми понятиями из области параллелизма данных, такими как работа с параллельными циклами и обработка исключений в них.

Глава 3

Реализация параллелизма данных

До сих пор мы изучали основы параллельного программирования и задачи. В этой главе мы рассмотрим еще один важный аспект параллельного программирования – параллелизм обработки данных. В то время как задача создает отдельную единицу работы для каждого задействованного потока, параллелизм обработки данных создает один поток команд, который выполняется несколькими потоками над общей коллекцией данных. Эта исходная коллекция разделена особым образом, позволяя нескольким потокам работать в ней одновременно. Поэтому понимание параллелизма данных особенно важно для получения максимальной производительности от циклов/коллекций.

В этой главе мы раскроем следующие темы:

- обработка исключений в параллельных циклах;
- создание своих стратегий разделения в параллельных циклах;
- отмена циклов;
- хранилище данных в параллельных циклах.

Технические требования

По завершении данной главы у вас сформируется хорошее представление о TPL и C#. Исходный код для главы 3 находится на сайте GitHub по адресу: https://github.com/PacktPublishing/Hands-On-Parallel-Programming-with-C-8-and-.NET-Core-3/tree/master/Chapter03.

От последовательных циклов к параллельным

TPL поддерживает параллелизм обработки данных через класс `System.Threading.Tasks.Parallel`, который обеспечивает параллельную реализацию циклов `For` и `Foreach`. Как разработчику вам не нужно беспокоиться о синхронизации или о создании задач, поскольку класс `Parallel` делает это за вас. Этот при-

ем позволяет вам легко писать параллельные циклы так же, как если бы вы писали последовательные.

Вот пример последовательного цикла `foreach`, который отправляет объект `trade` на сервер:

```
foreach(var trade in trades) {
  Book(trade);
}
```

Поскольку цикл последовательный, общее время его выполнения складывается из интервалов, необходимых для совершения каждой операции. Это означает, что время на выполнение цикла увеличивается, несмотря на то что длительность внутренних операций остается прежней. И здесь нам приходится работать с большими числами. Поскольку мы будем обрабатывать `trade` на сервере, а все современные серверы поддерживают одновременное выполнение нескольких запросов, мы можем преобразовать этот цикл из последовательного в параллельный – это даст нам значительный прирост производительности.

Предыдущий код можно преобразовать в параллельный следующим образом:

```
Parallel.ForEach(trades, trade => Book(trade));
```

При выполнении параллельного цикла TPL разбивает исходную коллекцию на части (partition) таким образом, чтобы каждая часть выполнялась одновременно и независимо. Планирование задач выполняется классом `TaskScheduler`, который учитывает системные ресурсы и нагрузку на процессор при разделении на части. Мы также можем создать **свой разделитель** (custom partitioner), или **планировщик** (scheduler). Позже мы обязательно рассмотрим это в разделе «*Создание своих стратегий разделения данных*».

Параллелизм данных лучше работает с независимыми частями данных.

Существует три способа преобразования последовательного кода в параллельный:

- при помощи метода `Parallel.Invoke`;
- при помощи метода `Parallel.For`;
- при помощи метода `Parallel.ForEach`.

Для наглядного представления о параллелизме обработки данных мы предлагаем вам разобрать различные способы использования класса `Parallel`.

Метод Parallel.Invoke

Является самым простым способом использования `Parallel`, а также основой для параллельных циклов `for` и `foreach`. Метод `Invoke` принимает массив функций в качестве параметра и выполняет их, хотя и не гарантирует полную параллельность их исполнения. Есть несколько важных моментов, о которых следует помнить при использовании `Parallel.Invoke`:

- параллельность не гарантируется. Будут действия выполняться параллельно или последовательно, решает `TaskScheduler`;
- `Parallel.Invoke` не гарантирует порядок выполнения операций;

- он также блокирует вызывающий поток до момента завершения всех операций.

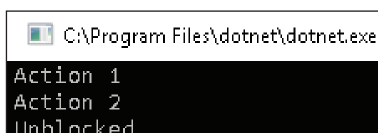
Синтаксис `Parallel.Invoke` представлен ниже.

```
public static void Invoke(
  params Action[] actions
)
```

Мы можем передать функцию или лямбда-выражение (пример ниже):

```
try {
  Parallel.Invoke(() => Console.WriteLine("Action 1"),
    new Action(() => Console.WriteLine("Action 2")));
} catch (AggregateException aggregateException) {
  foreach(var ex in aggregateException.InnerExceptions) {
    Console.WriteLine(ex.Message);
  }
}
Console.WriteLine("Unblocked");
Console.ReadLine();
```

Метод `Invoke` ведет себя как присоединенная дочерняя задача, поскольку он блокируется до тех пор, пока все действия не будут завершены. Все исключения записываются в `System.AggregateException` и выбрасываются вызывающему оператору. Поскольку исключения не было, данный код выведет следующее:



Можно достичь похожего эффекта при помощи класса `Task`, хотя на первый взгляд может показаться, что класс `Task` сложнее метода `Invoke`:

```
Task.Factory.StartNew(() => {
  Task.Factory.StartNew(() => Console.WriteLine("Action 1"),
    TaskCreationOptions.AttachedToParent);
  Task.Factory.StartNew(new Action(() => Console.WriteLine("Action 2")),
    TaskCreationOptions.AttachedToParent);
});
```

Метод `Invoke` похож на прикрепленную задачу, поскольку блокируется до тех пор, пока все действия не будут завершены. Исключения в данном случае собираются в `System.AggregateException` и возвращаются вызывающему методу.

Метод Parallel.For

Parallel.For – вариант последовательного цикла for, при котором итерации выполняются параллельно. Parallel.For возвращает экземпляр класса ParallelLoopResult, содержащий состояние цикла после его завершения. При проверке свойств IsCompleted и LowestBreakIteration ParallelLoopResult мы получаем информацию о завершении/отмене метода и узнаем, был ли он прерван пользователем. Ниже представлены возможные сценарии развития событий:

IsCompleted	LowestBreakIteration	Причина
True	N/A	Выполнять до завершения
False	Null	При выполнении конечного условия
False	Non-null integral value	Прерывание, вызванное в цикле

Базовый синтаксис метода Parallel.For:

```
public static ParallelLoopResult For {
  Int fromIncalme,
  Int toExclusiveme,
  Action < int > action
}
```

Пример вызова выглядит следующим образом:

```
Parallel.For(1, 100, (i) => Console.WriteLine(i));
```

Этот подход используется, если вам не нужно отменять, прерывать или управлять состоянием вызывающего потока, а также поддерживать какое-либо из его локальных состояний. Например, нам нужно подсчитать количество файлов в каталоге, которые были созданы сегодня. В данном случае реализация будет следующей:

```
int totalFiles = 0;
var files = Directory.GetFiles("C:\\");
Parallel.For(0, files.Length, (i) => {
  FileInfo fileInfo = new FileInfo(files[i]);
  if (fileInfo.CreationTime.Day == DateTime.Now.Day)
    Interlocked.Increment(ref totalFiles);
});
Console.WriteLine($"Total number of files in C: drive are {files.Count()} and {totalFiles}
files were created today.");
```

Этот код проверяет все файлы на диске C: и подсчитывает только те, которые были созданы сегодня. Ниже приведен пример вывода на моем компьютере:

```
Total number of files in C: drive are 91 and  0 files were created today.
```

В следующем разделе мы разберем метод `Parallel.ForEach`, реализующий параллельный вариант цикла `foreach`.

В некоторых коллекциях последовательная обработка происходит быстрее – все зависит от типа выполняемой работы.

Метод Parallel.ForEach

Представляет собой разновидность цикла `foreach`, в котором итерации могут выполняться параллельно. Исходная коллекция разбивается на части, а выполнение работы происходит в нескольких потоках. `Parallel.ForEach` работает с универсальными коллекциями, подобно циклу `for`, и возвращает результат при помощи `ParallelLoopResult`.

Основной синтаксис `Parallel.ForEach` выглядит так:

```
Parallel.ForEach < TSource > (
  IEnumerable < TSource > Source,
  Action < TSource > body
)
```

Например, у нас есть список URL-адресов, которые необходимо мониторить и обновлять для них статусы:

```
List < string > urls = new List < string > () {
  "www.google.com",
  "www.yahoo.com",
  "www.bing.com"
};
Parallel.ForEach(urls, url => {
  Ping pinger = new Ping();
  Console.WriteLine($"Ping Url {url} status is {pinger.Send(url).Status} by Task {Task.
CurrentId}");
});
```

В предыдущем коде мы обратились к классу `System.Net.NetworkInformation.Ping` для обработки отдельного элемента списка и отображения его состояния в консоли. Поскольку части независимы, эффективнее всего использовать реализацию кода, при которой не важен порядок исполнения.

На скриншоте ниже показан пример вывода этого кода:

Для одноядерных процессоров параллелизм может замедлить работу приложений. Количество задействованных ядер при параллельных операциях можно контролировать с помощью степени (degree) параллелизма.

Степень параллелизма

До сих пор мы изучали преимущество параллелизма обработки данных для эффективного использования доступных ресурсов процессора. Однако вы также должны знать о не менее важном понятии «степень параллелизма», определяющем максимальное количество задач, которые могут быть созданы вашими параллельными циклами. Вы можете установить степень параллелизма с помощью свойства `MaxDegreeOfParallelism`, которое является частью класса `ParallelOptions`. Ниже представлен синтаксис `Parallel.For`, в котором можно передать экземпляр `ParallelOptions`:

```
public static ParallelLoopResult For(
  int fromInclusive,
  int toExclusive,
  ParallelOptions parallelOptions,
  Action < int > body
)
```

Далее также приведен синтаксис метода `Parallel.ForEach`, в котором вы можете передать экземпляр `ParallelOptions`:

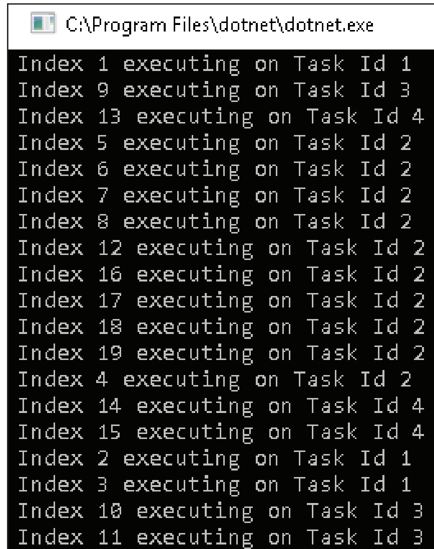
```
public static ParallelLoopResult ForEach < TSource > (
  IEnumerable < TSource > source,
  ParallelOptions parallelOptions,
  Action < TSource > body
)
```

По умолчанию значение степени параллелизма равно 64, что означает, что параллельные циклы могут использовать не более 64 одновременных задач. Но мы можем настраивать это значение. Давайте попробуем разобраться в этом на примерах.

Рассмотрим пример цикла `Parallel.For` с установленным `MaxDegreeOfParallelism` в значении 4:

```
Parallel.For(1, 20, new ParallelOptions {
  MaxDegreeOfParallelism = 4
}, index => {
  Console.WriteLine($"Index {index} executing on Task Id {Task.CurrentId}");
});
```

Результат выглядит следующим образом:

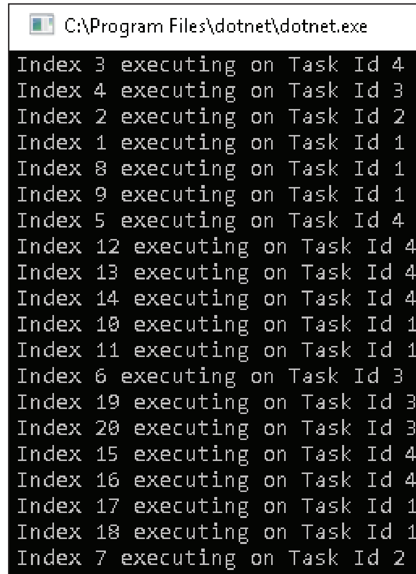


В данном случае цикл выполняется четырьмя задачами, обозначенными идентификаторами задач 1, 2, 3 и 4.

Вот пример цикла `Parallel.ForEach` с установленным `MaxDegreeOfParallelism` в значении 4:

```
var items = Enumerable.Range(1, 20);
Parallel.ForEach(items, new ParallelOptions {
  MaxDegreeOfParallelism = 4
}, item => {
  Console.WriteLine($"Index {item} executing on Task Id {Task.CurrentId}");
});
```

Так выглядит результат его работы:



Из этого следует, что цикл также выполняется четырьмя задачами с идентификаторами 1, 2, 3 и 4.

Для более сложных сценариев может потребоваться изменить эти настройки, чтобы управлять количеством используемых задач. Далее мы узнаем о том, как можно самостоятельно управлять разбиением коллекций на части при помощи стратегий разделения данных.

Создание своей стратегии разделения данных

Разделение данных – это еще одно важное понятие в параллелизме данных. Чтобы добиться параллелизма в исходной коллекции, ее необходимо разбить на небольшие части, к которым могут независимо обращаться различные потоки. Без такого разделения данных цикл будет выполняться последовательно. Выделяют два основных способа разбиения данных:

- по диапазону;
- по блокам.

Давайте подробнее поговорим о них.

Разделение данных по диапазону

Этот тип разбиения в основном используется с коллекциями, размер которых известен заранее. Название подхода говорит само за себя: каждый поток получает диапазон обрабатываемых элементов или начальные/конечные индексы исходной коллекции. Будучи простой формой разбиения данных, этот подход является очень эффективным потому, что каждый поток обрабатывает только свой диапазон. Издержки на синхронизацию отсутствуют, хотя изначально производительность немного падает из-за создания диапазонов. Этот тип разбиения лучше всего работает в сценариях с одинаковым количеством элементов в каждом диапазоне и равным временем на их обработку. При разном количестве элементов некоторые секции могут выполняться быстрее, а исполняющие их задачи простаивать, в то время как другие задачи будут продолжать свою работу.

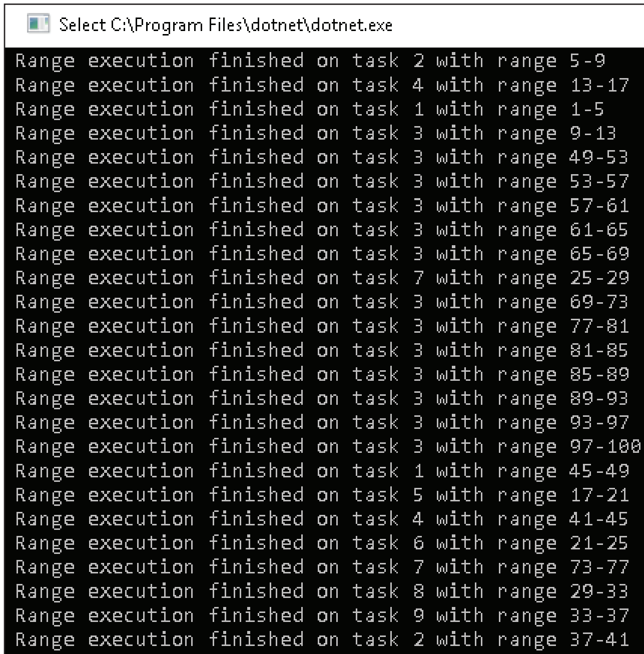
Разделение данных по блокам

Этот тип разделения данных в основном используется с такими коллекциями, как `LinkedList`, размер которых неизвестен заранее. В случае неравномерных коллекций такое разбиение на блоки (chunk) позволяет сбалансировать нагрузку. Каждый поток берет блок элементов на обработку и после завершения работы переходит к следующему блоку, который еще не успели обработать другие потоки. Размер блоков зависит от разделения, а также издержек на синхронизацию, гарантирующую отсутствие пересечений между блоками.

Мы можем в коде установить стратегию разбиения для цикла `Parallel.ForEach`:

```
var source = Enumerable.Range(1, 100).ToList();
OrderablePartitioner < Tuple < int, int >> orderablePartitioner =
  Partitioner.Create(1, 100);
Parallel.ForEach(orderablePartitioner, (range, state) => {
  var startIndex = range.Item1;
  var endIndex = range.Item2;
  Console.WriteLine($"Range execution finished on task {Task.CurrentId}with range
{startRange} - {endRange}");
});
```

Выше мы создали блочные разделители (chunked partitioners), используя класс `OrderablePartitioner` для диапазона элементов от 1 до 100. Также мы передали разделители в цикл `ForEach`, где каждый элемент передается потоку и затем выполняется. Результат выглядит следующим образом:



Теперь, когда мы понимаем принцип работы параллельных циклов, перейдем к более сложным понятиям: контроль над выполнением цикла или вынужденная остановка цикла.

Отмена циклов

До этого мы использовали конструкции `break` и `continue` в последовательных циклах. Вообще, `break` применяется для выхода из цикла, при котором он завершает текущую итерацию и пропускает следующие. `Continue`, наоборот, пропускает текущую итерацию и переходит к следующим. Эти конструкции применимы для последовательных циклов, так как они выполняются одним потоком. Говоря о параллельных циклах, здесь нельзя использовать ключевые слова `break` и `continue`, поскольку выполнение таких циклов происходит в нескольких потоках или задачах. Для прерывания параллельного цикла нам нужен класс `ParallelLoopState`, а для отмены – классы `CancellationToken` и `ParallelOptions`.

В этом разделе мы обсудим следующие параметры для отмены циклов:

- `Parallel.Break`;
- `ParallelLoopState.Stop`;
- `CancellationToken`.

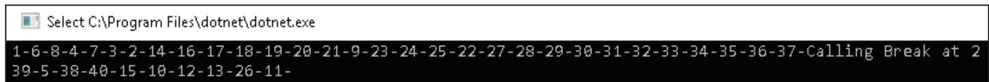
Давайте начнем!

Использование метода Parallel.Break

`Parallel.Break` имитирует работу обычного `break`. Давайте рассмотрим пример прерывания цикла. В представленном коде нам нужно отобрать определенное число из списка. Когда оно будет найдено, выполнение цикла прервется.

```
var numbers = Enumerable.Range(1, 1000);
int numToFind = 2;
Parallel.ForEach(numbers, (number, parallelLoopState) => {
  Console.Write(number + "-");
  if (number == numToFind) {
    Console.WriteLine($"Calling Break at {number}");
    parallelLoopState.Break();
  }
});
```

В соответствии с кодом цикл будет выполняться до тех пор, пока не найдет число 2. Если цикл будет последовательным, он прервется на второй итерации. Однако в параллельных циклах итерации выполняются различными задачами, поэтому числа могут выводиться вразнобой.



Для прерывания цикла мы воспользовались `parallelLoopState.Break()`, принцип работы которого аналогичен обычному `break` в последовательном цикле. Когда метод `Break()` встречается в каком-либо потоке, то он устанавливает номер итерации в свойстве `LowestBreakIteration` объекта `ParallelLoopState`. Этот номер будет считаться максимальным числом или последней выполнимой итерацией. Оставшиеся задачи будут выполняться до тех пор, пока не достигнут этого числа.

Последовательные вызовы метода `Break` путем параллельного выполнения итераций еще больше уменьшают `LowestBreakIteration`:

```
var numbers = Enumerable.Range(1, 1000);
Parallel.ForEach(numbers, (i, parallelLoopState) => {
  Console.WriteLine($"For i={i} LowestBreakIteration = {parallelLoopState.
LowestBreakIteration } and Task id = {Task.CurrentId}");
  if (i >= 10) {
    parallelLoopState.Break();
  }
});
```

При запуске предыдущего кода в Visual Studio мы получаем следующий результат:

```
C:\Program Files\dotnet\dotnet.exe
For i=3 LowestBreakIteration= and Task id =8
For i=7 LowestBreakIteration= and Task id =2
For i=5 LowestBreakIteration= and Task id =3
For i=6 LowestBreakIteration= and Task id =7
For i=1 LowestBreakIteration= and Task id =4
For i=4 LowestBreakIteration= and Task id =6
For i=8 LowestBreakIteration= and Task id =9
For i=2 LowestBreakIteration= and Task id =1
For i=9 LowestBreakIteration= and Task id =5
For i=18 LowestBreakIteration= and Task id =5
For i=19 LowestBreakIteration=17 and Task id =5
For i=11 LowestBreakIteration=17 and Task id =2
For i=20 LowestBreakIteration=10 and Task id =2
For i=13 LowestBreakIteration=10 and Task id =7
For i=10 LowestBreakIteration=10 and Task id =8
For i=22 LowestBreakIteration=9 and Task id =8
For i=12 LowestBreakIteration=9 and Task id =3
For i=17 LowestBreakIteration=9 and Task id =1
For i=14 LowestBreakIteration=9 and Task id =4
For i=16 LowestBreakIteration=9 and Task id =9
For i=15 LowestBreakIteration=9 and Task id =6
For i=21 LowestBreakIteration=9 and Task id =10
```

Здесь мы запускаем код на многоядерном процессоре. Многие итерации получают нулевое значение для `LowestBreakIteration` из-за того, что код выполняется на нескольких ядрах. На итерации 17 одно ядро вызывает метод `Break()` и устанавливает `LowestBreakIteration` со значением 17. На итерации 10 другое ядро вызывает `Break()` и уменьшает `LowestBreakIteration` до 10. Затем и для итерации 9 происходит то же самое.

Использование ParallelLoopState.Stop

Если вам не нужно имитировать последовательный `break` и вы лишь хотите поскорее выйти из параллельного цикла, то в таком случае используйте `ParallelLoopState.Stop`. Как и в случае с методом `Break()`, все параллельные итерации заканчиваются до выхода цикла.

```
var numbers = Enumerable.Range(1, 1000);
Parallel.ForEach(numbers, (i, parallelLoopState) => {
  Console.Write(i + " ");
  if (i % 4 == 0) {
    Console.WriteLine($"Loop Stopped on {i}");
    parallelLoopState.Stop();
  }
});
```

Ниже представлены результаты работы данного кода:

```
C:\Program Files\dotnet\dotnet.exe
6 7 2 5 4 8 Loop Stopped on 4
1 9 Loop Stopped on 8
11 14 13 12 Loop Stopped on 12
3 10
```

Как видите, одно ядро вызывает остановку (`Stop`) на итерации 4, другое – на итерации 8, а третье ядро – на итерации 12. Так как итерации 3 и 10 были запланированы, они продолжают выполняться.

Использование CancellationToken для отмены циклов

Для отмены циклов `Parallel.For` и `Parallel.ForEach` мы можем использовать класс `CancellationToken` по аналогии с задачами. При отмене метки (token) цикл завершает все параллельные итерации и не начинает новые. Когда они заканчиваются, параллельные циклы генерируют `OperationCanceledException`.

Давайте рассмотрим следующий пример. Для начала создадим `CancellationTokenSource`:

```
CancellationTokenSource cancellationTokenSource =
  new CancellationTokenSource();
```

Затем создадим задачу, которая отменит метку через 5 секунд:

```
Task.Factory.StartNew(() => {
  Thread.Sleep(5000);
  cancellationTokenSource.Cancel();
  Console.WriteLine("Token has been cancelled");
});
```

После этого создадим экземпляр `ParallelOptions` и передадим ему метку отмены:

```
ParallelOptions loopOptions = new ParallelOptions() {
  CancellationToken = cancellationTokenSource.Token
};
```

Далее запустим цикл с операцией, которая будет длиться более пяти секунд:

```
try {
  Parallel.For(0, Int64.MaxValue, loopOptions, index => {
    Thread.Sleep(3000);
    double result = Math.Sqrt(index);
    Console.WriteLine($"Index {index}, result {result}");
  });
} catch (OperationCanceledException) {
  Console.WriteLine("Cancellation exception caught!");
}
```

Ниже приведены результаты работы этого кода:

```
C:\Program Files\dotnet\dotnet.exe
Index 1152921504606846975, result 1073741824
Index 0, result 0
Index 2305843009213693950, result 1518500249.98802
Index 3458764513820540925, result 1859775393.37968
Index 4611686018427387900, result 2147483648
Index 5764607523034234875, result 2400959708.74862
Index 6917529027641081850, result 2630119584.2853
Index 8070450532247928825, result 2840853838.59289
Index 9223372036854775800, result 3037000499.97605
Index 1, result 1
Token has been cancelled
Index 1152921504606846976, result 1073741824
Index 1152921504606846977, result 1073741824
Index 2, result 1.4142135623731
Index 2305843009213693951, result 1518500249.98802
Index 3458764513820540926, result 1859775393.37968
Index 4611686018427387901, result 2147483648
Index 5764607523034234876, result 2400959708.74862
Index 6917529027641081851, result 2630119584.2853
Index 8070450532247928826, result 2840853838.59289
Index 2305843009213693953, result 1518500249.98802
Index 9223372036854775801, result 3037000499.97605
Index 3458764513820540928, result 1859775393.37968
Index 4, result 2
Cancellation exception caught!
```

Можно заметить, что запланированные итерации продолжают выполняться, несмотря на отмену метки. Надеюсь, мои примеры помогли вам понять, каким образом мы можем отменять параллельные циклы в различных ситуациях. Еще одним важным аспектом параллельного программирования является хранение данных, с которым мы познакомимся в следующем разделе.

Хранение данных в параллельных циклах

У всех параллельных циклов по умолчанию есть доступ к глобальным переменным. Однако существуют потери на синхронизацию при доступе к глобальным переменным, поэтому разумнее использовать локальные переменные на уровне отдельных потоков там, где это возможно. Мы можем создать либо **локальную переменную потока** (thread local variable), либо **локальную переменную блока данных** (partition local variable) для параллельных циклов.

Локальная переменная потока

Локальные переменные потока схожи с глобальными переменными, но только на уровне отдельной задачи. Они существуют до тех пор, пока задача выполняет те или иные итерации, назначенные на нее параллельным циклом.

В следующем примере мы изучим локальные переменные потока с помощью цикла `Parallel.For`. Предположим, нам нужно найти сумму 60 чисел посредством параллельного цикла.

Например, у нас есть четыре задачи, каждой из которых необходимо обработать по 15 итераций. Одним из способов решения задачи является создание глобальной переменной. После каждой итерации запущенная задача должна обновлять эту переменную, что повлечет за собой дополнительные затраты на синхронизацию. Четырем задачам будут соответствовать четыре локальные переменные потока, доступные только в самих потоках. Задача обновит переменную, а ее значение направится вызывающей программе для обновления глобальной переменной.

Этапы исполнения алгоритма:

1. Создайте коллекцию из 60 чисел, в которой каждый элемент имеет значение, равное индексу:

   ```
   var numbers = Enumerable.Range(1, 60);
   ```

2. Создайте `Action`, который будет выполняться после того, как задача завершит текущую итерацию. Метод получит конечный результат локальной переменной потока и добавит его к глобальной переменной `sumOfNumbers`:

   ```
   long sumOfNumbers = 0;
   Action < long > taskFinishedMethod = (taskResult) => {
     Console.WriteLine($"Sum at the end of all task iterations for task
   {Task.CurrentId}is {taskResult}");
     Interlocked.Add(ref sumOfNumbers, taskResult);
   };
   ```

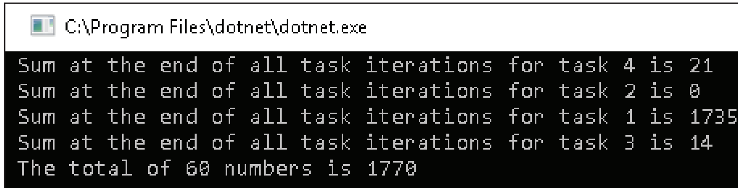
3. Создайте цикл `For`. Первыми двумя параметрами в нем будут `startIndex` и `endIndex`, а третьим – делегат, устанавливающий для локальной переменной потока начальное значение. За параметры отвечает задача, а мы лишь присваиваем значение `subtotal`, которое и является нашей локальной переменной потока.

Предположим, есть задача *TaskA*, которой поступают итерации с индексом от 1 до 5. *TaskA* сложит эти индексы (1 + 2 + 3 + 4 + 5) и полученную сумму (15) вернет как результат работы, который направится в `taskFinishedMethod` в качестве параметра:

```
Parallel.For(0, numbers.Count(),
  () => 0,
  (j, loop, subtotal) => {
```

```
    subtotal += j;
    return subtotal;
  },
  taskFinishedMethod
);
Console.WriteLine($"The total of 60 numbers is {sumOfNumbers}");
```

Если мы запустим код выше в Visual Studio, то получим следующий результат:



Помните, что результат работы может отличаться в зависимости от устройства и количества доступных ядер.

Локальная переменная блока данных

Эта переменная схожа с локальной переменной потока, только вот работает с блоками данных. Как вы уже знаете, цикл `ForEach` делит исходную коллекцию на части. Каждый такой блок имеет собственную копию локальной переменной. То есть переменная существует только во время обработки отдельного блока данных, в то время как задача может обработать несколько таких блоков.

Для начала создадим цикл `ForEach`. Первый параметр – исходная коллекция (числа). Второй параметр – делегат, который устанавливает начальное значение для локальной переменной потока. Третий параметр – выполняемое задачей действие. Нам остается лишь присвоить индекс `subtotal` для локальной переменной потока.

Разберем это на примере. Задача *TaskA* получает итерации с индексами от 1 до 5, затем их складывает (1 + 2 + 3 + 4 + 5) и получает сумму – 15. Эта сумма позже возвращается в качестве результата задачи и передается в `taskFinishedMethod` в качестве параметра.

Реализация кода:

```
Parallel.ForEach < int, long > (numbers,
  () => 0, //Метод инициализации локальной переменной
  (j, loop, subtotal) => //Действие, выполняемое на каждой итерации
  {
    subtotal += j; //Промежуточный итог - это локальная переменная
                   //потока
```

```
    return subtotal; //Значение, передаваемое на следующую итерацию
  },
  taskFinishedMethod);
Console.WriteLine($"The total of 60 numbers is {sumOfNumbers}");
```

Повторюсь, что результат работы может отличаться в зависимости от конфигурации компьютера и количества ядер процессора.

Резюме

В этой главе мы подробно разобрали реализацию параллелизма задач с помощью TPL. Мы начали с преобразования последовательных циклов в параллельные, используя некоторые встроенные методы TPL, такие как `Parallel.Invoke`, `Parallel.For` и `Parallel.ForEach`. Затем мы перешли к степени параллелизма и стратегии разделения данных на части, которые позволили нам добиться максимального использования ресурсов процессора. Далее мы обсудили способы отмены и прерывания параллельных циклов с помощью встроенных конструкций, таких как метки отмены, `Parallel.Break` и `ParallelLoopState.Stop`. В конце этой главы мы поговорили о различных вариантах хранения данных в параллельных циклах.

TPL предоставляет несколько вариантов для достижения параллелизма данных с помощью параллельной реализации циклов `For` и `ForEach`. Благодаря `ParallelOptions` и `ParallelLoopState` мы можем значительно улучшить производительность и контроль без потерь на синхронизацию.

В следующей главе мы рассмотрим еще одну интересную возможность библиотеки TPL под названием **PLINQ**.

Вопросы

1. Какой из методов не позволяет реализовать цикл `for` в TPL?
    1. `Parallel.Invoke`
    2. `Parallel.While`
    3. `Parallel.For`
    4. `Parallel.ForEach`
2. Что не является стандартной стратегией данных?
    1. Массовое разделение данных (bulk partitioning)
    2. Разделение по диапазону (range partitioning)
    3. Разделение по блокам (chunk partitioning)
3. По умолчанию значение степени параллелизма равно
    1. 1
    2. 64

4. Метод `Parallel.Break` гарантирует мгновенный выход из параллельного цикла сразу после его вызова.
   1. Верно
   2. Неверно
5. Может ли один поток видеть локальные переменные других потоков или блоков данных?
   1. Да
   2. Нет

Глава 4

Использование PLINQ

https://t.me/it_boooks

PLINQ является параллельной реализацией **Language Integrate Query** (LINQ). Впервые PLINQ был представлен в .NET Framework 4.0 и с тех пор получил множество новых возможностей. До появления LINQ у разработчиков возникали сложности с извлечением данных из различных хранилищ, таких как XML или базы данных, поскольку для работы с каждым из них нужны были специфические знания. LINQ – это синтаксис языка, который использует делегаты .NET и готовые методы для получения или изменения данных без обращения к низкоуровневым библиотекам.

Эту главу мы начнем с рассмотрения LINQ-провайдеров в .NET. Поскольку PLINQ используется все чаще, мы рассмотрим его достоинства и недостатки. И в конце мы поговорим о факторах, влияющих на производительность PLINQ.

В этой главе мы рассмотрим следующие темы:

- LINQ-провайдеры .NET;
- создание запросов PLINQ;
- сохранение порядка обработки данных в PLINQ;
- объединение данных в PLINQ;
- обработка исключений в PLINQ;
- объединение параллельных и последовательных запросов;
- недостатки PLINQ;
- оптимизация PLINQ.

Технические требования

Для освоения этой главы вам понадобится уверенное знание TPL и C#. Примеры исходного кода из главы 4 доступны на сайте GitHub по адресу: https://github.com/PacktPublishing/Hands-On-Parallel-Programming-with-C-8-and-.NET-Core-3/tree/master/Chapter04.

LINQ-провайдеры в .NET

LINQ представляет собой набор API, которые облегчают работу с XML, объектами и базами данных. LINQ также поддерживает множество поставщиков (providers) данных, среди которых чаще всего используются:

- LINQ to objects (LINQ для объектов): позволяет разработчикам запрашивать объекты из оперативной памяти (массивы, коллекции, generic-типы и т. д.). Он возвращает `IEnumerable` и поддерживает функции сортировки, фильтрации, группировки и объединения. Его функционал реализован в `System.Linq`;
- LINQ to XML (LINQ для XML) или XLINQ: дает возможность разработчикам запрашивать или изменять источники данных в формате XML. Этот провайдер определен в `System.Xml.Linq`;
- LINQ to ADO.NET (LINQ для ADO.NET): группа технологий, которая позволяет разработчикам запрашивать или изменять реляционные источники данных, такие как SQL Server, MySQL или Oracle;
- LINQ to SQL (LINQ для SQL) или DLINQ: использует **объектно-реляционное отображение** (Object Relational Mapping, ORM). DLINQ является устаревшей технологией, которая поддерживается Microsoft, но давно не развивается. Она работает только с SQL Server и позволяет пользователям сопоставлять таблицы баз данных с классами .NET. У него есть адаптер, который работает как интерфейс разработчика для базы данных;
- LINQ to datasets (LINQ для наборов данных): позволяет разработчикам запрашивать или изменять наборы данных в памяти. Может работать с любой базой данных, для которых у ADO.NET есть провайдер;
- LINQ to entities (LINQ для сущностей): самая передовая и востребованная технология. Она дает возможность работать с любыми реляционными базами данных, включая SQL Server, Oracle, IBM Db2 и MySQL. Также LINQ to entities поддерживает механизмы ORM;
- PLINQ: параллельная реализация LINQ to objects. Обычные запросы LINQ выполняются последовательно и могут медленно выполняться, если требуется множество вычислений. PLINQ поддерживает параллельное выполнение запросов, запуская их с помощью нескольких потоков и ядер процессора.

.NET поддерживает преобразования языка LINQ в язык PLINQ с помощью метода `AsParallel()`. PLINQ хорошо справляется с выполнением сложных операций. Он разделяет исходные данных на блоки, которые, в свою очередь, обрабатываются различными потоками на нескольких ядрах процессора. PLINQ также поддерживает XLINQ и LINQ to objects.

Создание PLINQ-запросов

Прежде чем перейти к запросам PLINQ, нам сначала нужно познакомиться с классом `ParallelEnumerable`. Как только у вас сформируется представление о `ParallelEnumerable`, мы изучим написание параллельных запросов.

Знакомство с классом ParallelEnumerable

Класс `ParallelEnumerable` доступен в пространстве имен `System.Linq` библиотеки `System.Core`.

Помимо поддержки многих стандартных операций запросов в LINQ, этот класс также поддерживает множество дополнительных методов параллельного исполнения:

- `AsParallel()`: преобразует последовательный LINQ-запрос в параллельный;
- `AsSequential()`: активирует последовательное выполнение параллельного запроса;
- `AsOrdered()`: PLINQ по умолчанию не сохраняет порядок выполнения задач и возврат результатов, но это можно исправить вызовом метода `AsOrdered()`;
- `AsUnordered()`: возвращает порядок выполнения задач к неупорядоченному режиму, который ранее был изменен с помощью `AsOrdered()`;
- `ForAll()`: обеспечивает параллельное выполнение запроса;
- `Aggregate()`: применяет функцию к запросу, которая объединяет результаты работы параллельных задач;
- `WithDegreesOfParallelism()`: позволяет указать максимальное количество задач, используемых для параллельного выполнения запроса;
- `WithExecutionMode()`: используя этот метод, можно принудительно выполнить запрос в параллельном режиме или позволить PLINQ решить, как он будет выполняться – последовательно или параллельно.

Чуть позже мы познакомимся ближе с представленными методами на примерах. Стоит отметить, что существует очень удобный инструмент под названием LINQPad. Он помогает нам узнать о запросах LINQ/PLINQ, так как в его арсенале более 500 доступных шаблонов, а еще он дает возможность подключаться к различным источникам данных. Вы можете скачать LINQPad на сайте https://www.linqpad.net.

Наш первый запрос PLINQ

Предположим, нам нужно найти все числа, кратные трем.

Для начала мы определим множество из 100 000 чисел:

```
var range = Enumerable.Range(1, 100000);
```

Для последовательного поиска чисел, которые делятся на три, используйте запрос LINQ:

```
var resultList = range.Where(i => i % 3 == 0).ToList();
```

Ниже приведена параллельная версия аналогичного запроса с использованием метода `AsParallel`:

```
var resultList = range.AsParallel().Where(i => i % 3 == 0).ToList();
```

Вот та же версия, использующая параметр синтаксиса запроса в LINQ:

```
var resultList = (from i in range.AsParallel() where i % 3 == 0
  select i).ToList()
```

Ниже представлен полный код:

```
var range = Enumerable.Range(1, 100000);
//Это последовательная версия
var resultList = range.Where(i => i % 3 == 0).ToList();
Console.WriteLine($"Sequential: Total items are {resultList.Count}");
//Это параллельная версия с использованием метода .AsParallel
resultList = range.AsParallel().Where(i => i % 3 == 0).ToList();
resultList = (from i in range.AsParallel() where i % 3 == 0
  select i).ToList();
Console.WriteLine($"Parallel: Total items are {resultList.Count}");
Console.WriteLine($"Parallel: Total items are {resultList.Count}");
```

Результат выглядит следующим образом:

Сохранение порядка в PLINQ при параллельном исполнении

PLINQ параллельно выполняет задачи и не заботится о сохранении их порядка для более эффективной реализации параллельных запросов. Однако иногда важно, чтобы элементы выполнялись в том же порядке, в каком они существуют в исходной коллекции. Например, вы отправляете на сервер несколько запросов для загрузки файла по частям, затем объединяете полученные кусочки в одно целое, чтобы воссоздать файл. Поскольку загрузка происходит частями, важно, чтобы каждая такая часть скачивалась и объединялась в правильном порядке. При параллельном исполнении сохранение исходного порядка в сегментах и его согласованность при объединении элементов напрямую влияют на производительность.

Мы можем отменить стандартное поведение и включить сохранение порядка, используя `AsOrdered()` в исходной коллекции. Если нам нужно будет отключить сохранение порядка, в любой момент мы можем вызвать метод `AsUnOrdered()`.

Давайте рассмотрим пример:

```
var range = Enumerable.Range(1, 10);
Console.WriteLine("Sequential Ordered");
range.ToList().ForEach(i => Console.Write(i + "-"));
```

Этот код является последовательным, поэтому при его запуске мы получим следующий результат:

```
C:\Program Files\dotnet\dotnet.exe
Sequential Ordered
1-2-3-4-5-6-7-8-9-10-
```

Также мы можем сделать его параллельным при помощи метода `AsParallel()`:

```
Console.WriteLine("Parallel Unordered");
var unordered = range.AsParallel().Select(i => i).ToList();
unordered.ForEach(i => Console.WriteLine(i));
```

Предыдущий код выполняется параллельно, поэтому порядок в нем перепутан:

```
Parallel UnOrdered
4-8-7-2-3-1-6-9-10-5-
```

Объединив самое лучшее их обоих вариантов, мы можем выполнить код параллельно с сохранением порядка:

```
var range = Enumerable.Range(1, 10);
Console.WriteLine("Parallel Ordered");
var ordered = range.AsParallel().AsOrdered().Select(i => i).ToList();
ordered.ForEach(i => Console.WriteLine(i));
```

Получаем такой результат:

```
Parallel Ordered
1-2-3-4-5-6-7-8-9-10-
```

Из примера видно, что метод `AsOrdered()` выполняет задачи параллельно, сохраняя при этом исходный порядок, в то время как при стандартных настройках порядок не сохранялся. Из-за того, что порядок проверяется на каждой итерации, `AsOrdered()` снижает производительность.

Последовательное выполнение с использованием метода AsUnOrdered()

Как только мы вызовем `AsOrdered()`, запрос начнет выполняться последовательно. Могут возникнуть ситуации, при которых на какое-то время нам понадобится чередование упорядоченной и неупорядоченной реализаций запроса для лучшей производительности.

Скажем, нам нужно сгенерировать квадраты первых 100 чисел из диапазона чисел. Сделать это параллельно можно одним из следующих способов:

```
var range = Enumerable.Range(100, 10000);
var ordered = range.AsParallel().AsOrdered()
                              .Take(100).Select(i => i*i);
```

Чтобы получить первые 100 чисел, нам нужен AsOrdered(). Проблема лишь в том, что запрос Select также будет выполняться последовательно. Мы можем повысить производительность сочетанием AsOrdered() и AsUnOrdered():

```
var range = Enumerable.Range(100, 10000);
var ordered =
  range.AsParallel().AsOrdered().Take(100).AsUnordered()
                              .Select(i => i *i).ToList();
```

Теперь первые 100 элементов будут извлекаться по порядку. После получения данных запрос выполняется без сохранения какого-либо порядка.

Параметры объединения данных в PLINQ

Ранее мы уже говорили, что при создании параллельного запроса исходная коллекция разделяется на части, которые могут одновременно обрабатываться несколькими задачами. Как только запрос завершится, результаты должны быть объединены так, чтобы они были доступны потоку-потребителю. В зависимости от параметров запроса есть разные способы объединения результатов. Мы можем указать явные параметры объединения результатов, используя перечисление ParallelMergeOperation и метод расширения WithMergeOption().

Давайте рассмотрим разные доступные нам варианты объединения данных.

Параметр NotBuffered

Результаты отдельных задач не сохраняются в буфер, а сразу возвращаются потоку-потребителю:

```
var range = ParallelEnumerable.Range(1, 100);
Stopwatch watch = null;
ParallelQuery < int > notBufferedQuery =
  range.WithMergeOptions(ParallelMergeOptions.NotBuffered)
  .Where(i => i % 10 == 0)
  .Select(x => {
    Thread.SpinWait(1000);
    return x;
  });
watch = Stopwatch.StartNew();
foreach(var item in notBufferedQuery) {
  Console.WriteLine($"{item}:{watch.ElapsedMilliseconds}");
}
```

```
Console.WriteLine($"\nNotBuffered Full Result returned
  in {watch.ElapsedMilliseconds} ms");
```

Ниже представлен результат работы:

```
C:\Program Files\dotnet\dotnet.exe
10:397
20:402
30:402
40:402
60:402
70:403
80:403
90:404
50:405
100:406

NotBuffered Full Result returned in 407 ms
```

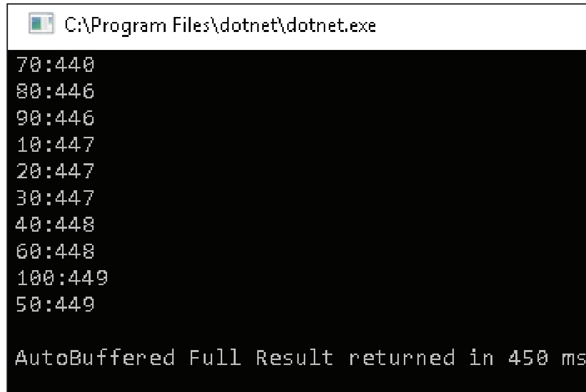
Параметр AutoBuffered

Результаты всех задач помещаются в буфер, который периодически становится доступным для потоков-потребителей. Могут возвращаться несколько буферов в зависимости от размера коллекции. При использовании этого параметра поток-потребитель дольше ожидает первый результат. Это является опцией по умолчанию.

Рассмотрим следующий код:

```
var range = ParallelEnumerable.Range(1, 100);
Stopwatch watch = null;
ParallelQuery < int > query =
  range.WithMergeOptions(ParallelMergeOptions.AutoBuffered)
  .Where(i => i % 10 == 0)
  .Select(x => {
    Thread.SpinWait(1000);
    return x;
  });
watch = Stopwatch.StartNew();
foreach(var item in query) {
  Console.WriteLine($"{item}:{watch.ElapsedMilliseconds}");
}
Console.WriteLine($"\nAutoBuffered Full Result returned in
                         {watch.ElapsedMilliseconds} ms");
watch.Stop();
```

Результат работы представленного выше кода выглядит так:



Параметр FullyBuffered

Результаты различных задач помещаются в буфер, прежде чем поток-потребитель получит к ним доступ. Это улучшает общую производительность, несмотря на то что время, необходимое для получения первого результата, возрастает.

```
var range = ParallelEnumerable.Range(1, 100);
Stopwatch watch = null;
ParallelQuery<int> fullyBufferedQuery =
  range.WithMergeOptions(ParallelMergeOptions.FullyBuffered)
  .Where(i => i % 10 == 0)
  .Select(x => {
    Thread.SpinWait(1000);
    return x;
  });
watch = Stopwatch.StartNew();
foreach(var item in fullyBufferedQuery) {
  Console.WriteLine($"{item}:{watch.ElapsedMilliseconds}");
}
Console.WriteLine($"\nFullyBuffered Full Result returned
                       in {watch.ElapsedMilliseconds} ms");
watch.Stop();
```

Результат работы представлен ниже:

```
C:\Program Files\dotnet\dotnet.exe
90:424
10:431
20:431
30:432
40:432
60:432
70:432
80:432
100:432
50:433

FullyBuffered Full Result returned in 434 ms
```

Некоторые операторы запросов не поддерживают все режимы слияния. Ниже представлен список операторов с их ограничениями по использованию:

Operator	Restrictions
AsEnumerable	None
Cast	None
Concat	Non-ordered queries that have an Array or List source only.
DefaultIfEmpty	None
OfType	None
Reverse	Non-ordered queries that have an Array or List source only.
Select	None
SelectMany	None
Skip	None
Take	None
Where	None

Данную информацию можно найти на сайте http://msdn.microsoft.com/en-us/library/dd997424(v=vs.110).aspx.

Помимо предыдущих операторов, `ForAll()` будет всегда `NotBuffered`, а `OrderBy` – всегда `FullyBuffered`. Пользовательские параметры объединения данных просто игнорируются этими операторами.

Отправка и обработка исключений с помощью PLINQ

Как и другие техники параллельной обработки, PLINQ вызывает исключение `System.AggregateException` каждый раз, когда сталкивается с ошибкой в работе. Обработка исключений во многом зависит от архитектуры приложения. Иногда вам может понадобиться, чтобы программа завершилась как можно быстрее или же все исключения были возвращены родительскому потоку.

В следующем примере мы обернем параллельный запрос в блок `try-catch`. Когда запрос сгенерирует исключение, оно передастся обратно родительскому потоку в виде `System.AggregateException`:

```
var range = ParallelEnumerable.Range(1, 20);
ParallelQuery < int > query = range.Select(i => i / (i - 10)).WithDegreeOfParallelism(2);
try {
  query.ForAll(i => Console.WriteLine(i));
} catch (AggregateException aggregateException) {
  foreach(var ex in aggregateException.InnerExceptions) {
    Console.WriteLine(ex.Message);
    if (ex is DivideByZeroException)
      Console.WriteLine("Attempt to divide by zero. Query stopped.");
  }
}
```

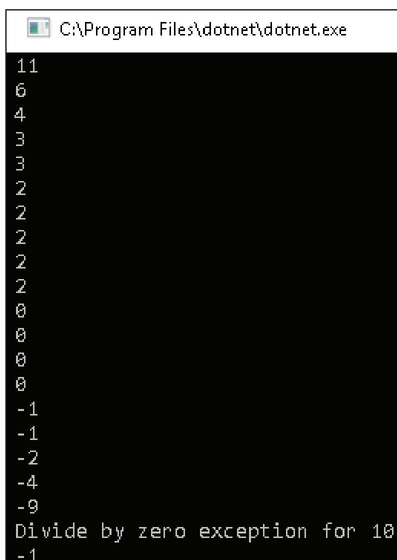
Результат будет следующим:

Мы также можем использовать `try-catch` внутри делегата, который будет своевременно предупреждать об ошибках. Он также может быть использован в сценарии, при котором нам будет нужно лишь зарегистрировать исключе-

ние и продолжить выполнение запроса, предоставив значение по умолчанию в качестве результата:

```
var range = ParallelEnumerable.Range(1, 20);
Func<int, int> selectDivision = (i) => {
  try {
    return i / (i - 10);
  } catch (DivideByZeroException ex) {
    Console.WriteLine($"Divide by zero exception for {i}");
    return -1;
  }
};
ParallelQuery <int> query = range.Select(i =>
  selectDivision(i)).WithDegreeOfParallelism(2);
try {
  query.ForAll(i => Console.WriteLine(i));
} catch (AggregateException aggregateException) {
  foreach(var ex in aggregateException.InnerExceptions) {
    Console.WriteLine(ex.Message);
    if (ex is DivideByZeroException)
      Console.WriteLine("Attempt to divide by zero. Query stopped.");
  }
}
```

Мы получим следующий результат:



Обработка исключений крайне важна для сохранения правильной работы приложения, а также для информирования пользователя об ошибках. В следующем разделе мы обсудим, как можно объединять параллельные и последовательные запросы.

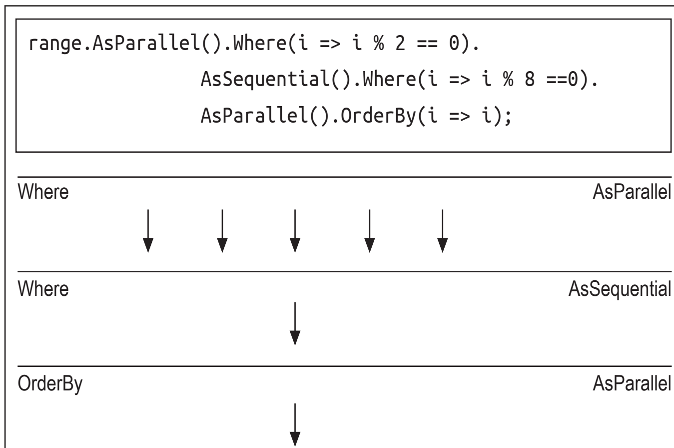
Объединение параллельных и последовательных запросов LINQ

Мы уже обсуждали использование `AsParallel()` для создания параллельных запросов. Возможно, иногда может потребоваться выполнить оперции последовательно, для этого подходит метод `AsSequential()`. Как только мы применяем этот метод к параллельному запросу, следующие операции выполняются последовательно. Рассмотрим код:

```
var range = Enumerable.Range(1, 1000);
range.AsParallel().Where(i => i % 2 == 0).AsSequential()
          .Where(i => i    % 8 == 0).AsParallel().OrderBy(i => i);
```

Показанный выше метод `Where(i => i % 2 == 0)` будет выполняться параллельно, а второй `Where(i => i % 8 == 0)` – последовательно. Вызов `OrderBy` осуществляется в параллельном режиме.

Этот процесс отображен на диаграмме:



Теперь нам должно стать понятнее, как можно объединять синхронные и параллельные запросы LINQ. В следующем разделе мы узнаем, как отменять запросы PLINQ для экономии ресурсов процессора.

Отмена запросов PLINQ

Мы можем отменить запрос PLINQ с помощью классов `CancellationTokenSource` и `CancellationToken`. Метка отмены передается в запрос PLINQ с помощью утверждения `WithCancellation`, а отменить операцию запроса можно вызовом `CancellationToken.Cancel`. Отмена запроса вызывает исключение `OperationCancelledException`.

Это делается следующим образом.

1. Создайте источник метки отмены:

```
CancellationTokenSource cs = new CancellationTokenSource();
```

2. Создайте задачу и отмените метку через 4 секунды:

```
Task cancellationTask = Task.Factory.StartNew(() => {
  Thread.Sleep(4000);
  cs.Cancel();
});
```

3. Оберните запрос PLINQ в блок `try`:

```
try {
  var result = range.AsParallel()
    .WithCancellation(cs.Token)
    .Select(number => number)
    .ToList();
}
```

4. Добавьте два блока `catch` для перехвата `OperationCanceledException` и `AggregateException`:

```
catch (OperationCanceledException ex) {
  Console.WriteLine(ex.Message);
} catch (AggregateException ex) {
  foreach(var inner in ex.InnerExceptions) {
    Console.WriteLine(inner.Message);
  }
}
```

5. Определите диапазон с наибольшим значением, выполнение которого занимает более четырех секунд:

```
var range = Enumerable.Range(1,1000000);
```

6. Запустите код. Через четыре секунды мы увидим следующий результат:

У паралелльного программирования есть свои особенности. В следующем разделе мы рассмотрим недостатки написания параллельного кода с помощью PLINQ.

Недостатки параллельного программирования с помощью PLINQ

Часто PLINQ работает быстрее своего непараллельного аналога LINQ. Однако из-за разделения данных на части и объединения результатов работы в PLINQ появляются накладные расходы, снижающие производительность. Ниже представлены особенности, которые необходимо учитывать при использовании PLINQ.

1. **Параллельная обработка не всегда происходит быстрее**: распараллеливание – это накладные расходы. Если ваша исходная коллекция небольшая и не требует длительных вычислений, целесообразнее выполнять запросы последовательно. Всегда измеряйте производительность последовательных и параллельных запросов для выбора оптимального решения.
2. **Избегайте операций ввода-вывода с атомарностью** (atomicity): избегайте в PLINQ операций ввода-вывода с участием файловой системы, базы данных, сети или разделяемой памяти. В связи с тем, что такие операции не потокобезопасны, их использование может привести к исключениям. Возможным решением могли бы стать примитивы синхронизации, но их использование существенно бы снизило производительность.
3. **Ваши запросы не всегда могут выполняться параллельно**: решение о параллельном выполнении в PLINQ принимается средой .NET CLR. Даже вызов метода `AsParallel()` в запросе не дает гарантий, что выполнение будет параллельным.

Факторы, влияющие на производительность PLINQ (ускорения)

Основной целью PLINQ является ускорение запросов путем разделения данных и параллельного исполнения задач. Однако существует множество факторов, которые могут повлиять на производительность PLINQ. К ним относят расходы на синхронизацию при разделении данных, а также планирование и сбор результатов из нескольких потоков. PLINQ лучше всего работает в «*идеально» параллельных* сценариях, где потокам не нужно синхронизироваться и беспокоиться о порядке выполнения. Однако такую совершенную, «*идеальную» параллельност*ь не всегда можно получить из-за особенностей реализуемых алгоритмов. Давайте попробуем разобраться в факторах, влияющих на производительность PLINQ.

Степень параллелизма

При большем количестве ядер можно добиться значительного повышения производительности, поскольку TPL гарантирует одновременное выполнение нескольких задач на нескольких ядрах. Это улучшение не может быть экспоненциальным. Для поиска оптимального варианта нам может потребоваться запустить разные реализации алгоритма на нескольких ядрах и сравнить результаты.

Настройка объединения данных

Мы можем существенно улучшить отзывчивость пользовательского интерфейса в сценариях, при которых результаты часто меняются, или когда пользователь хочет поскорее получить результаты. По умолчанию PLINQ буферизует результаты, а затем объединяет их и возвращает пользователю. Мы можем изменить это поведение, выбрав соответствующий вариант объединения данных.

Тип разделения данных

Мы всегда должны проверять, сбалансировано или не сбалансировано выполнение задач. В несбалансированных сценариях могут быть добавлены пользовательские разделители данных (custom partitioners) для повышения производительности.

Когда нужно сохранять последовательное исполнение в PLINQ?

Мы всегда должны вычислять затраты отдельного потока и всего запроса для принятия решения о последовательном или параллельном исполнении. Параллельные запросы не всегда могут быть быстрыми из-за дополнительных накладных расходов на разделение данных, планирование и т. д.:

> *вычислительные затраты = затраты на обработку одного блока данных * общее количество блоков данных*.

Параллельные запросы могут обеспечить значительный прирост производительности при увеличении вычислительных затрат на один блок. Однако если этот прирост окажется незначительным, тогда целесообразнее выполнять запрос последовательно.

PLINQ выбирает последовательное или параллельное выполнение в зависимости от комбинаций операторов в запросе. Проще говоря, если запрос содержит любой из представленных операторов, PLINQ может запустить его последовательно:

- Take, TakeWhile, Skip, SkipWhile, First, Last, Concat, Zip или ElementAt;
- индексированные Where и Select, они же перегрузки Where и Select соответственно.

Код ниже демонстрирует использование индексированных Where и Select:

```
IEnumerable<int> query = numbers.AsQueryable()
                  .Where((number, index) => number <= index * 10);
IEnumerable<bool> query = range.AsQueryable()
                  .Select((number, index) => number <= index * 10);
```

Порядок работы

PLINQ способствует лучшей обработке неупорядоченных коллекций, поскольку для упорядоченных требуются накладные расходы, в числе которых – разделение данных, планирование, сбор результатов, а также вызов GroupJoin и фильтров. Вам как разработчикам нужно понимать, в каких ситуациях использовать AsOrdered().

ForAll против вызова ToArray() или ToList()

Вызывая ToList(), ToArray() или проверяя результат в цикле, мы заставляем PLINQ объединять результаты всех параллельных потоков в единую структуру данных. Это заметно снижает производительность. Если мы лишь хотим выполнить действия над набором элементов, то лучше использовать метод ForAll().

Принудительный параллелизм

PLINQ не всегда осуществляет параллельное выполнение запроса. В зависимости от типа запроса PLINQ также может выбрать и последовательную реализацию. Мы можем контролировать режим исполнения с помощью WithExecutionMode. WithExecutionMode – это метод расширения, который работает с объектами типа ParallelQuery. Принимает значение из перечисления ParallelExecutionMode. Стандартное значение ParallelExecutionMode позволяет PLINQ выбрать наилучший режим выполнения. Можно реализовать параллельное исполнение с помощью ForceParallelism:

```
var range = Enumerable.Range(1, 10);
var squares = range.AsParallel().WithExecutionMode(
  ParallelExecutionMode.ForceParallelism).Select(i => i * i);
squares.ToList().ForEach(i => Console.Write(i + "-"));
```

Генерация последовательностей

На протяжении всей книги мы использовали метод Enumerable.Range() для генерации последовательности чисел. Кроме того, генерировать числа можно

и параллельно с помощью класса ParallelEnumerable. Давайте сравним класс Enumerable с ParallelEnumerable:

```
Stopwatch watch = Stopwatch.StartNew();
IEnumerable<int> parallelRange =
                  ParallelEnumerable.Range(0, 5000).Select(i => i);
watch.Stop();
Console.WriteLine($"Time elapsed {watch.ElapsedMilliseconds}");
Stopwatch watch2 = Stopwatch.StartNew();
IEnumerable<int> range = Enumerable.Range(0, 5000);
watch2.Stop();
Console.WriteLine($"Time elapsed {watch2.ElapsedMilliseconds}");
Console.ReadLine();
```

Получаем следующий результат:

```
Time elapsed using ParallelEnumerable : 3
Time elapsed using Enumerable : 16
```

Вы могли заметить, что ParallelEnumerable создает последовательность данных намного быстрее, чем Enumerable.

При схожем сценарии нам может понадобиться генерация последовательности, содержащей одно и то же число. В таком случае мы можем использовать метод ParallelEnumerable.Repeat():

```
IEnumerable<int> rangeRepeat = ParallelEnumerable.Repeat(1, 5000);
```

Вот мы и подошли к концу этой главы, в которой познакомились с PLINQ. Теперь давайте подведем итоги.

Резюме

В начале данной главы мы обсудили основные возможности LINQ, после этого учились писать параллельные запросы с помощью PLINQ. Мы узнали, что PLINQ может повышать производительность приложения, однако не стоит забывать и о его недостатках. Вам как программистам стоит взвешивать все возможные варианты, написав запросы с помощью LINQ и PLINQ, а затем сравнив их производительность.

В следующей главе мы узнаем об использовании примитивов синхронизации для сохранения согласованности и целостности данных, которые обрабатываются сразу несколькими потоками.

Вопросы

1. Какой из провайдеров LINQ лучше поддерживает реляционные объекты?
    1. LINQ to SQL
    2. LINQ to entities
2. Мы можем легко преобразовать LINQ в параллельный с помощью `AsParallel()`.
    1. Верно
    2. Неверно
3. Нельзя переключаться между упорядоченным и неупорядоченным выполнением в PLINQ.
    1. Верно
    2. Неверно
4. Какой из вариантов позволяет буферизировать результаты параллельных задач и периодически предоставлять их потоку-потребителю?
    1. `FullyBuffered`
    2. `AutoBuffered`
    3. `NotBuffered`
5. Какое вы получите исключение, если внутри задачи выполняется следующий код:

```
int i=5;
i = i/i-5;
```

    1. `AggregateException`
    2. `DivideByZeroException`

Часть II

СТРУКТУРЫ ДАННЫХ .NET CORE, КОТОРЫЕ ПОДДЕРЖИВАЮТ ПАРАЛЛЕЛИЗМ

В этой части вы глубже изучите конструкции языка и фреймворка, поддерживающие параллелизм и синхронизацию.

Содержание части представлено главами:

- глава 5 «Примитивы синхронизации»;
- глава 6 «Использование параллельных коллекций»;
- глава 7 «Повышение производительности с помощью отложенной инициализации».

Глава 5

Примитивы синхронизации

В предыдущей главе мы обсудили недостатки параллельного программирования, в числе которых были дополнительные расходы на синхронизацию. Поскольку мы разбиваем работу на паралеллельные задачи, то возникает необходимость в синхронизации результатов каждого потока. Мы обсудили понятия локальной памяти потоков и разделов, которые могут быть в некоторой степени использованы для решения проблем с синхронизацией. Однако потоки все так же необходимо синхронизировать, чтобы мы могли записывать данные в общую ячейку памяти и выполнять операции ввода-вывода.

В этой главе мы обсудим примитивы синхронизации, предоставляемые .NET Framework и TPL.

В главе 5 мы рассмотрим следующие темы:

- примитивы синхронизации (synchronization primitives);
- операции со взаимоблокировкой (interlocked operations);
- примитивы блокировки (locking primitives);
- примитивы отправки сигналов (signaling primitives);
- легковесные примитивы синхронизации (lightweight synchronization primitives);
- барьеры и события обратного отсчета (barriers and countdown events).

К концу главы у вас сформируется понимание сферы применения различных примитивов синхронизации, доступных в .NET.

Технические требования

Для освоения этой главы вам понадобится уверенное знание TPL и параллельных циклов. Исходные коды из данной главы доступны на GitHub по ссылке: https://github.com/PacktPublishing/Hands-On-Parallel-Programming-with-C-8-and-.NET-Core-3/tree/master/Chapter05.

Что такое примитивы синхронизации?

Прежде чем мы перейдем к примитивам синхронизации, нам нужно понять, что такое критическая секция. *Критическая секция* – это участок кода, который должен быть защищен от параллельного доступа для сохранения работоспособности. Например, запись данных в файл.

Примитивы синхронизации – простые программные механизмы платформы более низкого уровня (то есть операционной системы), которые помогают в реализации многопоточности на уровне ядра ОС. Внутри они используют низкоуровневые атомарные операции, а также барьеры памяти (memory barriers). Распространенными примерами примитивов являются блокировки, мьютексы (mutexes), условные переменные и семафоры (semaphores). *Монитор* – это высокоуровневое программное средство синхронизации, использующее в своей основе другие примитивы синхронизации.

Платформа .NET Framework содержит ряд примитивов синхронизации, которые нужны для взаимодействия потоков, а также предотвращения возможной «гонки», когда несколько потоков «соревнуются» за доступ к общему ресурсу. Примитивы синхронизации можно условно разделить на пять категорий:

- операции со взаимоблокировкой;
- блокирующие;
- сигнальные;
- типы с облегченной синхронизацией;
- `SpinWait`.

В следующих разделах мы обсудим каждую категорию с соответствующими низкоуровневыми примитивами.

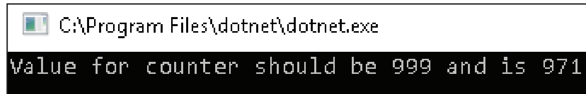
Операции со взаимоблокировкой

Класс `Interlocked` реализует операции со взаимоблокировками. Они необходимы для осуществления атомарных операций с переменными, разделяемых между потоками. Этот класс предоставляет такие методы, как `Increment`, `Decrement`, `Add`, `Exchange` и `CompareExchange`.

Рассмотрим код, который пытается увеличить переменную в параллельном цикле:

```
Parallel.For(1, 1000, i => {
  Thread.Sleep(100);
  _counter++;
});
Console.WriteLine($"Value for counter should be
                    999 and is {_counter}");
```

Если мы запустим этот код, то вывод будет следующим:

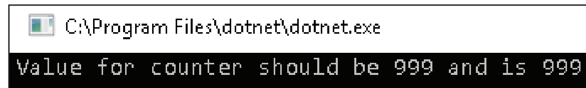


Как видите, ожидаемое значение не совпадает с фактическим. Это происходит из-за состояния «гонки» между потоками, возникающего, когда поток стремится прочитать значение переменной, которое еще не было записано в ячейку памяти.

Мы можем изменить предыдущий код с помощью класса `Interlocked` для обеспечения его потокобезопасности:

```
Parallel.For(1, 1000, i => {
  Thread.Sleep(100);
  Interlocked.Increment(ref _counter);
});
Console.WriteLine($"Value for counter should be
                          999 and is {_counter}");
```

Так выглядит ожидаемый результат:



Аналогичным образом можно использовать `Interlocked.Decrement(ref _counter)`, чтобы потокобезопасно уменьшить значение.

В представленном коде отражен полный список операций:

```
//_counter становится 1
Interlocked.Increment(ref _counter);
//_counter становится 0
Interlocked.Decrement(ref _counter);
//Добавить: _counter становится 2
Interlocked.Add(ref _counter, 2);
//Вычесть: _counter становится 0
Interlocked.Add(ref _counter, -2);
//Считываем 64-битное поле
Console.WriteLine(Interlocked.Read(ref _counter));
//Меняем местами значение _counter на 10
Console.WriteLine(Interlocked.Exchange(ref _counter, 10));
//Проверяем, равно ли _counter 10, и если да, заменяем на 100
Console.WriteLine(Interlocked.CompareExchange
                       (ref _counter, 100, 10));
//_counter становится 100
```

В .NET Framework 4.5 были также добавлены еще два новых метода: `Interlocked.MemoryBarrier()` и `Interlocked.MemoryBarrierProcessWide()`.

В следующем разделе мы узнаем больше о барьерах памяти в .NET.

Барьеры доступа к памяти в .NET

Потоковые модели по-разному работают на одноядерных и многоядерных процессорах. В одноядерных только один поток получает доступ к процессору, пока другие ждут своей очереди. Это дает нам уверенность, что при любом обращении потока к памяти (для загрузки и хранения) порядок будет сохраняться. Данная модель также известна как **модель последовательной согласованности** (sequential consistency model). В многоядерных процессорных системах несколько потоков выполняются одновременно. Есть вероятность, что последовательная согласованность в этих системах может отсутствовать, поскольку либо аппаратное обеспечение, либо компилятор во время исполнения (Just in Time, JIT) может изменить порядок работы потоков для повышения производительности. Инструкции обращения к памяти также могут быть переупорядочены в целях повышения производительности для кеширования, прогнозирования загрузки или отложенных операций сохранения.

Пример прогнозирования загрузки (load speculation) выглядит так:

```
a = b;
```

Пример операции сохранения (store operation) выглядит следующим образом:

```
c = 1;
```

Данные операторы не всегда выполняются в том же порядке, в котором они написаны в коде. Компиляторы немного изменяют порядок для повышения производительности. Давайте поближе познакомимся с изменением порядка (reordering).

Что такое изменение порядка?

Компилятор может выполнить заданную последовательность операторов либо в том же порядке, в каком они были получены, либо изменить их порядок в целях повышения производительности, но только при условии, если несколько потоков исполняют один и тот же код. Например, посмотрим на следующий код:

```
a = b;
c = 1;
```

В предыдущем примере можно поменять порядок:

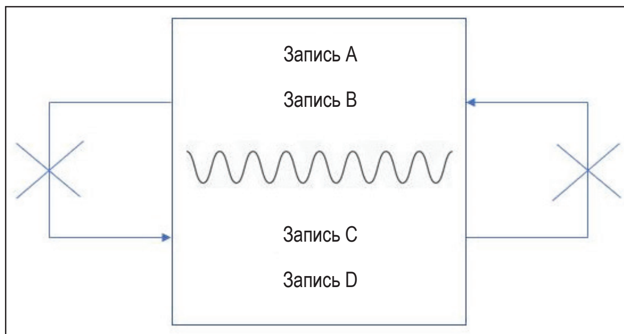
```
c = 1;
a = b;
```

Изменение порядка кода является проблемой для многоядерных процессоров со слабой памятью, таких как старенький Intel Itanium. Однако это никак не влияет на одноядерные процессоры из-за их модели последовательной согласованности. Изменение порядка кода происходит либо аппаратно, либо с помощью JIT-компилятора. Для изменения порядка в коде нам нужен определенный **барьер памяти** (memory barrier).

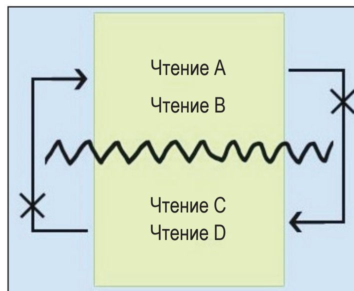
Типы барьеров памяти

Барьеры памяти гарантируют, что операторы, расположенные до и после барьера, не будут его пересекать, сохраняя порядок исполнения. Существует три типа барьера памяти:

- **барьер записи** (store barrier) не пропускает через себя операции записи значений в память. Для операций чтения он уже не подойдет, в них еще можно менять порядок. Подобный эффект достигается также инструкцией **SFENCE** центрального процессора:



- **барьер чтения** (load barrier) не позволяет операциям чтения пересекать барьер, но не ограничивает операции записи. Аналогичный эффект достигается инструкцией **LFENCE** центрального процессора:



- **полный барьер** обеспечивает сохранение порядка, не пропуская операции чтения и записи. Такой же эффект достигается с помощью ин-

струкции **MFENCE** процессора. Поведение полного барьера памяти часто реализуется конструкциями синхронизации .NET, такими как:

- Task.Start, Task.Wait и Task.Continuation;
- Thread.Sleep, Thread.Join, Thread.SpinWait, Thread.VolatileRead и Thread.VolatileWrite;
- Thread.MemoryBarrier;
- Lock, Monitor.Enter и Monitor.Exit;
- методы класса Interlocked.

Полубарьеры (half barriers) задаются ключевым словом volatile и методами класса Volatile. Платформа .NET Framework предоставляет встроенные шаблоны, использующие volatile-поля в таких классах, как Lazy<T> и LazyInitializer. Мы обсудим их далее в главе 7.

Как избежать изменения порядка

Мы можем избежать изменения порядка операторов с помощью Thread.MemoryBarrier(), как показано ниже:

```
static int a = 1, b = 2, c = 0;
private static void BarrierUsingTheadBarrier() {
  b = c;
  Thread.MemoryBarrier();
  a = 1;
}
```

Thread.MemoryBarrier создает полный барьер, который не пропускает операции чтения и записи. Он также используется внутри Interlock.MemoryBarrier, поэтому тот же код можно записать иначе:

```
private static void BarrierUsingInterlockedBarrier() {
  b = c;
  Interlocked.MemoryBarrier();
  a = 1;
}
```

Если мы хотим создать барьер на уровне процесса, то можем использовать Interlocked.MemoryBarrierProcessWide, который был впервые представлен в .NET Core 2.0. Он является оберткой над FlushProcessWriteBuffer из Windows API или sys_membarrier в ядре Linux:

```
private static void BarrierUsingInterlockedProcessWideBarrier() {
  b = c;
  Interlocked.MemoryBarrierProcessWide();
  a = 1;
}
```

Предыдущий пример показывает, как мы можем создать барьер для всего процесса. Теперь перейдем к блокирующим примитивам.

Введение в примитивы блокировки

Блокировки могут ограничивать для потоков доступ к защищенным ресурсам. Для сохранения эффективности кода нам нужно определить критические секции, которые будут защищаться блокирующими примитивами.

Как работает блокировка

Этапы при блокировке общего ресурса:

1. Поток или группа потоков получают доступ к общему ресурсу, создавая блокировку.
2. Другие потоки, которые не создавали блокировку, переходят в состояние ожидания.
3. Как только один поток сбросит блокировку, она перейдет к следующему в очереди потоку.

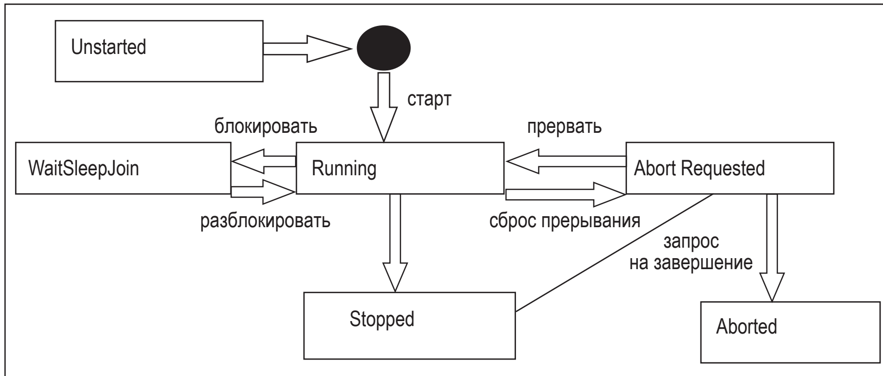
Для понимания работы примитивов блокировки требуется изучить состояния потоков, а также понятия блокировки (blocking) и вращения (spinning, спиннинг).

Состояния потока

В любой момент жизненного цикла потока мы можем запросить его состояние, используя свойство `ThreadState`. Поток может находиться в следующих состояниях:

- `Unstarted` (не запущен): поток создается средой CLR, но метод `System.Threading.Thread.Start` еще не был выполнен;
- `Running` (исполяется): поток запущен через вызов `Thread.Start` и не ожидает отложенной (pending) операции;
- `WaitSleepJoin` (ожидает-спит-объединяется): поток находится в заблокированном состоянии в результате обращения к методам `Wait()`, `Sleep()` или `Join()`;
- `StopRequested` (запрошена остановка): код запросил остановку работы потока;
- `Stoppped` (остановлен): поток перестал выполняться;
- `AbortRequested` (запрошена экстренная остановка): в потоке вызывается метод `Abort()`, но поток пока не остановлен, ожидая `ThreadAbortException` для своего завершения;
- `Aborted` (прерван): поток прерван;
- `SuspendRequested` (запрошена временная приостановка): запрашивается приостановка потока путем метода `Suspend`;
- `Suspended` (временно приостановлен): поток приостановлен;
- `Background` (фоновый режим): поток выполняется в фоновом режиме.

Давайте попробуем разобрать цикл потока от его начального состояния, `UnStarted`, до конечного, `Stopped`:



Поток, созданный CLR, находится в состоянии `Unstarted`. Он переходит от `Unstarted` к `Running`, когда внешний код вызывает метод `Thread.Start()`. Из `Running` поток может перейти в состояния:

- `WaitSleepJoin`;
- `AbortRequested`;
- `Stopped`.

Говорят, что поток блокируется, находясь в состоянии `WaitSleepJoin`. Выполнение заблокированного потока приостанавливается, так как он ожидает выполнения внешних условий, которые могут быть результатом другого потока или операции ввода-вывода, связанной с процессором. Будучи заблокированным, поток при этом получает квант (интервал для выполнения) времени процессора и не использует его до тех пор, пока не выполнится условие блокировки. После этого поток разблокируется. Поскольку процессор выполнял переключение контекста между потоками, то блокировка и разблокировка негативно скажутся на производительности.

Поток может быть разблокирован в следующих событиях:

- если условие блокировки выполнено;
- был вызван метод `Thread.Interrupt` на заблокированном потоке;
- прерыванием потока с помощью `Thread.Abort`;
- когда достигается указанный тайм-аут .

Блокировка или вращение?

Заблокированный поток освобождает квант процессорного времени на определенный период. С одной стороны, процессор становится доступен для других потоков, и это повышает производительность, а с другой – появляются накладные расходы на переключение контекста. Это хорошо работает в случае, когда поток блокируется надолго. Если же время ожидания небольшое, то лучше обратиться к ждущим циклам (spinning), не освобождая квант процессора. Например, этот код будет зациклен навечно:

```
while(!done);
```

Пустой цикл `while` проверяет значение логической (булевой) переменной. Если какой-либо внешний поток изменяет значение переменной на `false`, то ожидание заканчивается, и цикл прерывается. Несмотря на потерю процессорного времени, производительность может существенно улучшиться при условии небольшого ожидания. Платформа .NET Framework предоставляет специальные конструкции, такие как `SpinWait` и `SpinLock`, о которых мы позже поговорим в этой главе.

Давайте попробуем разобраться в некоторых примитивах блокировки на примерах кода.

Блокировка, мьютекс и семафор

Блокирующие конструкции lock и мьютекс (mutex) позволяют только одному потоку получить доступ к защищенному ресурсу. Lock – это реализация быстрого вызова (shortcut), которая использует высокоуровневый класс синхронизации под названием `Monitor`.

Семафор (semaphore) – блокирующая конструкция, которая позволяет заданному количеству потоков получить доступ к защищенному ресурсу. Lock может синхронизировать доступ только внутри процесса, но если нам нужно получить доступ к ресурсу системного уровня или общей памяти, необходимо синхронизировать доступ между несколькими процессами. Мьютекс позволяет синхронизировать доступ к ресурсам между процессами, обеспечивая блокировку на уровне ядра операционной системы.

В таблице показано сравнение характеристик этих конструкций:

Примитивы синхронизации	Выделенное количество потоков	Перекрестный процесс
Lock	1	×
Mutex	1	✓
Semaphore	Many	✓
SemaphoreSlim	Many	×

Lock и **Mutex** разрешают только однопоточный доступ к общим ресурсам, в то время как **Semaphore** и **SemaphoreSlim** могут предоставлять многопоточный доступ к ресурсам. Кроме того, там, где **Lock** и **SemaphoreSlim** работают только с потоками внутри процесса, у **Mutex** и **Semaphore** есть возможность блокировки всего процесса.

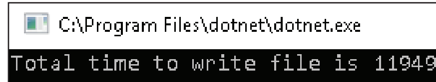
Lock

Рассмотрим код, который пытается записать число в текстовый файл:

```
vvar range = Enumerable.Range(1, 1000);
Stopwatch watch = Stopwatch.StartNew();
for (int i = 0; i < range.Count(); i++) {
  Thread.Sleep(10);
```

```
  File.AppendAllText("test.txt", i.ToString());
}
watch.Stop();
Console.WriteLine($"Total time to write file is {watch.ElapsedMilliseconds}");
```

При запуске предыдущего кода вывод выглядит следующим образом:



Задача выполняет 1000 итераций, и выполнение каждого такого элемента занимает примерно 10 миллисекунд. Время, затраченное на выполнение задачи, равно 1000 умножить на 10, что составляет 10 000 миллисекунд (10 секунд). Также нужно учитывать время, затрачиваемое на выполнение ввода-вывода, поэтому общее время будет равно 11 949 на моем компьютере.

Сейчас мы попробуем распараллелить эту задачу с помощью методов `AsParallel()` и `AsOrdered()`:

```
range.AsParallel().AsOrdered().ForAll(i => {
  Thread.Sleep(10);
  File.AppendAllText("test.txt", i.ToString());
});
```

При попытке запустить этот код мы получим исключение `System.IO.IOException: 'The process cannot access the file ...\test.txt' because it is being used by another process.'` ('Процесс не может получить доступ к файлу ...\test.txt', так как он используется другим процессом.').

В силу того, что критическая секция включает доступ к общему ресурсу в виде файла, здесь допускаются только атомарные операции. Из-за распараллеливания возникает ситуация, при которой несколько потоков пытаются одновременно записать данные в файл, и это вызывает исключение. Нам нужно убедиться, что код работает параллельно и достаточно быстро, сохраняя атомарность при записи в файл. Изменим предыдущий код с помощью оператора lock.

Сначала необходимо создать статическую переменную ссылочного типа (reference type). В нашем случае мы берем переменную типа `object`, хотя подойдет и любой другой класс. Нам нужна переменная ссылочного типа, так как блокировка применяется только к объектам из оперативной памяти:

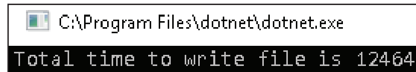
```
static object _locker = new object();
```

Затем мы модифицируем код внутри метода `ForAll()`, чтобы включить `lock`:

```
range.AsParallel().AsOrdered().ForAll(i => {
  lock(_locker) {
    Thread.Sleep(10);
```

```
    File.WriteAllText("test.txt", i.ToString());
  }
});
```

Теперь при запуске кода не возникнет никаких исключений, однако затраченное время на задачу на самом деле будет больше последовательного выполнения:



Что же пошло не так? Lock обеспечивает атомарность, гарантируя, что только у одинокого потока есть доступ к уязвимому коду, но с этим появляются и издержки на блокировку потока, ожидающего ее освобождения. Мы называем это «тупой блокировкой» (dumb lock). Мы можем немного изменить программу для блокировки только критической секции с целью повышения производительности, сохраняя при этом атомарность:

```
range.AsParallel().AsOrdered().ForAll(i => {
  Thread.Sleep(10);
  lock(_locker) {
    File.WriteAllText("test.txt", i.ToString());
  }
});
```

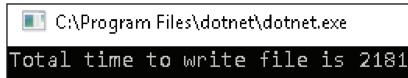
Вывод предыдущего кода представлен ниже:

Как видите, мы смогли достичь ощутимых успехов, смешивая синхронизацию с распараллеливанием. Можно добиться подобных результатов, используя другой примитив блокировки – класс `Monitor`.

На самом деле lock является сокращенным синтаксисом для использования `Monitor.Enter()` и `Monitor.Exit()`, обернутых в блок `try-catch`. Таким образом, тот же самый код может быть записан иначе:

```
range.AsParallel().AsOrdered().ForAll(i => {
  Thread.Sleep(10);
  Monitor.Enter(_locker);
  try {
    File.WriteAllText("test.txt", i.ToString());
  } finally {
    Monitor.Exit(_locker);
  }
});
```

Так выглядит вывод этого кода:



Mutex

Предыдущий код хорошо работает в приложении с одним экземпляром, поскольку задачи выполняются внутри процесса, где lock блокирует барьер памяти. Если мы запустим несколько экземпляров приложения, оба приложения будут иметь свои собственные копии статических переменных и, следовательно, будут блокировать лишь свои барьеры памяти. Это позволит лишь одному потоку в каждом процессе входить в критическую секцию для записи файла, что приведет к тому же исключению: `System.IO.IOException: 'The process cannot access the file ...\test.txt' because it is being used by another process.'`.

Для того чтобы у нас получилось провести блокировку общих ресурсов, мы можем обратиться к блокировке на уровне ядра ОС, используя класс `mutex`. Подобно `lock`, `mutex` предоставляет доступ к защищенному ресурсу только для одного потока. Но он также может работать между процессами, предоставляя одному потоку в каждом из них доступ к защищенному ресурсу, независимо от количества выполняемых процессов.

Mutex бывает именованный и безымянный. Безымянный работает как блокировка, он не может работать со всеми процессами.

Для начала создадим безымянный `Mutex`:

```
private static Mutex mutex = new Mutex();
```

Затем мы изменим предыдущий параллельный код так, чтобы можно было использовать `Mutex` в качестве блокировки:

```
range.AsParallel().AsOrdered().ForAll(i => {
  Thread.Sleep(10);
  mutex.WaitOne();
  File.AppendAllText("test.txt", i.ToString());
  mutex.ReleaseMutex();
});
```

Ниже представлен вывод предыдущего кода:

Используя `Mutex`, мы можем вызвать метод `WaitHandle.WaitOne()` для блокировки критической секции и `ReleaseMutex()` для разблокировки. Закрытие или освобождение (disposing) мьютекса автоматически его разблокирует.

Предыдущая программа прекрасно работает, но при попытке запуска нескольких экземпляров она выдаст исключение `IOException`. Для этого мы можем создать именованный мьютекс:

```
private static Mutex namedMutex = new Mutex(false, "ShaktiSinghTanwar");
```

Как вариант можно на мьютексе указать тайм-аут при вызове `WaitOne()`, чтобы тот ждал сигнала в течение заданного промежутка времени, прежде чем разблокироваться, как показано в примере:

```
namedMutex.WaitOne(3000);
```

Если мьютекс не получает сигнал, он ждет три секунды, перед тем как разблокироваться.

Блокировка и мьютекс освобождаются только из того потока, который их активировал.

Semaphore

Блокировка, мьютекс и монитор предоставляют только одному потоку доступ к защищенному ресурсу. Однако иногда необходимо, чтобы доступ к общему ресурсу был у нескольких потоков. Примерами таких задач являются объединение (pooling) ресурсов и управление узким местом (throttling). В отличие от `lock` или `mutex`, `semaphore` потоконезависим, что означает, что любой поток может активировать `semaphore`. Точно так же, как мьютекс, он работает со всеми процессами.

Обычное создание семафора выглядит так:

```
Semaphore semaphore = new Semaphore()
```

▲ 1 of 3 ▼ Semaphore(**int initialCount**, int maximumCount)
Initializes a new instance of the Semaphore class, specifying the initial number of entries and the maximum number of concurrent entries.
initialCount: *The initial number of requests for the semaphore that can be granted concurrently.*

Как видите, он принимает два параметра: `initialCount` определяет, сколько запросов из разных потоков могут использовать семафор изначально; `maximumCount` определяет максимальное количество запросов.

Скажем, у нас есть служба удаленного доступа, которая разрешает только три одновременных соединения на одного клиента и требует одну секунду для обработки запроса:

```
private static void DummyService(int i) {
  Thread.Sleep(1000);
}
```

У нас есть метод с 1000 запросов, которые должны вызывать службу и передать ей необходимые параметры. Нам нужно параллельно выполнять саму задачу, а также следить за тем, чтобы всегда было не больше трех одновременных вызовов. Это можно реализовать, создав `semaphore` с максимальным количеством 3:

```
Semaphore semaphore = new Semaphore(3,3);
```

Используя следующий semaphore, мы можем написать код, имитирующий параллельное создание 1000 запросов:

```
range.AsParallel().AsOrdered().ForAll(i => {
  semaphore.WaitOne();
  Console.WriteLine($"Index {i} making service call using
    Task {Task.CurrentId}");
  //Имитация Http-вызова
  CallService(i);
  Console.WriteLine($"Index {i} releasing semaphore using
    Task {Task.CurrentId}");
  semaphore.Release();
});
```

Так выглядит результат:

```
C:\Program Files\dotnet\dotnet.exe
Index 2 making service call using Task 4
Index 1 making service call using Task 5
Index 5 making service call using Task 6
Index 1 releasing semaphore using Task 5
Index 5 releasing semaphore using Task 6
Index 2 releasing semaphore using Task 4
Index 3 making service call using Task 9
Index 4 making service call using Task 2
Index 6 making service call using Task 3
Index 4 releasing semaphore using Task 2
Index 3 releasing semaphore using Task 9
Index 11 making service call using Task 4
Index 6 releasing semaphore using Task 3
Index 9 making service call using Task 5
Index 10 making service call using Task 6
Index 11 releasing semaphore using Task 4
Index 12 making service call using Task 2
Index 9 releasing semaphore using Task 5
Index 7 making service call using Task 7
Index 10 releasing semaphore using Task 6
Index 8 making service call using Task 8
Index 12 releasing semaphore using Task 2
Index 7 releasing semaphore using Task 7
Index 8 releasing semaphore using Task 8
```

Как видите, запускаются лишь три потока и вызывают службу, в то время как другие ждут снятия блокировки. Как только один поток сбрасывает блокировку, запускается следующий, но только при условии, что в данный момент не больше трех потоков находится в критической секции.

Существует два типа семафоров: локальные (local) и глобальные (global). Далее мы о них поговорим.

Локальный семафор

Локальный semaphore доступен лишь для приложения, в котором используется. Любой semaphore без имени создается как локальный:

```
Semaphore semaphore = new Semaphore(1,10);
```

Глобальный семафор

Глобальный semaphore создается на уровне операционной системы и доступен для всех процессов. Любой semaphore с именем создается как глобальный:

```
Semaphore semaphore = new Semaphore(1,10,"GlobalSemaphore");
```

Если мы создадим semaphore в ситуации с одним потоком, то он будет действовать как блокировка.

ReaderWriterLock

Класс ReaderWriterLock определяет блокировку, которая одновременно поддерживает несколько «читателей» (readers) и одного «писателя» (writer). Это может пригодиться в сценариях, в которых потоки регулярно считывают общий ресурс, который при этом редко обновляется. На платформе .NET Framework есть два класса блокировки reader-writer: ReaderWriterLock и ReaderWriterLockSlim. ReaderWriterLock уже почти устарел, так как он может привести к возможным взаимоблокировкам (deadlocks), сниженной производительности, сложным правилам рекурсии и другим проблемам. Позже мы обсудим ReaderWriterLockSlim более подробно в этой главе.

Введение в сигнальные примитивы

Важным аспектом в параллельном программировани является координация задач. При создании задач может возникнуть сценарий производителя/потребителя (producer/consumer), в котором поток-потребитель ждет, когда другой поток-производитель обновит общий ресурс. Поскольку потребитель точно не знает, когда будет обновление, он продолжает опрашивать общий ресурс, что может привести к ситуациям гонки. Опрос весьма неэффективен при таких сценариях. Лучше использовать сигнальные примитивы .NET Framework, благодаря которым поток-потребитель приостанавливается до тех пор, пока не получит сигнал от потока-производителя. Рассмотрим универсальные сигнальные примитивы на примере Thread.Join, WaitHandles и EventWaitHandlers.

Thread.Join

Это самый простой способ, при котором один поток может получить сигнал от другого. Примитив Thread.Join является блокирующим по своей природе. Это

значит, что вызывающий поток блокируется, пока не завершится присоединенный (joined) поток. При необходимости мы можем указать тайм-аут, по достижении которого заблокированный поток выходит из состояния блокировки.

В следующем примере мы создадим дочерний поток, имитирующий задачу с длительным временем выполнения. После ее завершения поток обновит выходные данные в локальной переменной `result`. Программа должна вывести результат `10` в консоль. Попробуем реализовать этот код:

```
int result = 0;
Thread childThread = new Thread(() => {
  Thread.Sleep(5000);
  result = 10;
});
childThread.Start();
Console.WriteLine($"Result is {result}");
```

Вывод предыдущего кода выглядит следующим образом:

Ожидаемый результат (`10`) не совпал с полученным (`0`). Это произошло из-за того, что основной поток, который должен был записать значение, выполнился раньше, чем завершился дочерний. Чтобы получить желаемую логику работы, мы можем заблокировать основной поток до тех пор, пока не завершится дочерний. Это можно сделать, вызвав `Join()` в дочернем потоке:

```
int result = 0;
Thread childThread = new Thread(() => {
  Thread.Sleep(5000);
  result = 10;
});
childThread.Start();
childThread.Join();
Console.WriteLine($"Result is {result}");
```

Повторно запустив код, мы получим ожидаемый результат через пять секунд, на которые будет заблокирован основной поток:

EventWaitHandle

`System.Threading.EventWaitHandle` предоставляет событие синхронизации для потока и служит базовым классом для `AutoResetEvent` и `ManualResetEvent`. Мы можем оповещать `EventWaitHandle`, вызывая `Set()` или `SignalAndWait()`. Класс `EventWaitHandle` не имеет привязки к потоку, поэтому о нем может сообщить любой поток. Давайте подробнее рассмотрим `AutoResetEvent` и `ManualResetEvent`.

AutoResetEvent

Он относится к классам `WaitHandle`, которые сбрасываются и перезагружаются автоматически. При сбросе они позволяют одному потоку пройти через созданный барьер. Как только поток его преодолеет, барьер снова установится, блокируя тем самым потоки до следующего сигнала.

В следующем примере мы попытаемся узнать сумму 10 чисел безопасным для потоков способом, без использования блокировок.

Для начала создайте `AutoResetEvent` с начальным состоянием non-signaled (сигнал не отправлен) или `false`. Это значит, что все потоки должны ждать сигнала. Если мы установим начальное состояние `signaled` или `true`, первый поток будет выполняться, в то время как остальные будут ждать сигнала:

```
AutoResetEvent autoResetEvent = new AutoResetEvent(false);
```

Затем создайте задачу, отправляющую сигнал 10 раз каждую секунду при помощи метода `AutoResetEvent.Set()`:

```
Task signallingTask = Task.Factory.StartNew(() => {
  for (int i = 0; i < 10; i++) {
    Thread.Sleep(1000);
    autoResetEvent.Set();
  }
});
```

Обозначьте переменную `sum` и присвойте ей значение 0:

```
int sum = 0;
```

Создайте параллельный цикл `for`, который запустит 10 задач. Каждая задача будет сразу запускаться и ждать сигнала с помощью метода `autoResetEvent.WaitOne()`. Каждую секунду первая задача будет посылать сигнал, после которого новый поток запускается и обновляет значение `sum`:

```
Parallel.For(1, 10, (i) => {
  Console.WriteLine($"Task with id {Task.CurrentId}
                        waiting for signal to enter");
```

```
  autoResetEvent.WaitOne();
  Console.WriteLine($"Task with id {Task.CurrentId}
                       received signal to enter");
  sum += i;
});
```

Результат представлен ниже:

```
C:\Program Files\dotnet\dotnet.exe
Task with id 2 waiting for signal to enter
Task with id 3 waiting for signal to enter
Task with id 4 waiting for signal to enter
Task with id 5 waiting for signal to enter
Task with id 6 waiting for signal to enter
Task with id 8 waiting for signal to enter
Task with id 7 waiting for signal to enter
Task with id 9 waiting for signal to enter
Task with id 10 waiting for signal to enter
Task with id 2 received signal to enter
Task with id 3 received signal to enter
Task with id 4 received signal to enter
Task with id 5 received signal to enter
Task with id 6 received signal to enter
Task with id 8 received signal to enter
Task with id 9 received signal to enter
Task with id 7 received signal to enter
Task with id 10 received signal to enter
```

Как видите, изначально все 10 задач блокируются, и по одной задаче в секунду разблокируется после сигнала.

ManualResetEvent

Относится к объектам, управляющим ожиданием (wait handle, дословно «переключатель, управляющий ожиданием»), сбрасывать который необходимо вручную. В отличие от `AutoResetEvent`, который пропускает только один поток при получении сигнала, `ManualResetEvent` позволяет потокам проходить через себя до тех пор, пока он не будет снова активирован. Попробуем разобраться в этом на простом примере.

В следующем примере нам нужно сделать 15 сервисных вызовов, разбив их на группы по пять в каждой. Все полученные группы будут выполняться параллельно с разницей в две секунды. При совершении вызова необходимо убедиться, что система подключена к сети. Для моделирования ее состояния создадим две задачи: одна будет оповещать об отключении сети, а другая – о включении.

Для начала создадим `ManualResetEvent` с начальным состоянием *off* (false):

```
ManualResetEvent manualResetEvent = new ManualResetEvent(false);
```

Затем создаем две задачи, имитирующие включение и выключение сети запуском события network *off* каждые две секунды (блокирует все сетевые вызовы) и запуском события network *on* каждые пять секунд (позволяет выполняться всем сетевым вызовам):

```
Task signalOffTask = Task.Factory.StartNew(() => {
  while (true) {
    Thread.Sleep(2000);
    Console.WriteLine("Network is down");
    manualResetEvent.Reset();
  }
});

Task signalOnTask = Task.Factory.StartNew(() => {
  while (true) {
    Thread.Sleep(5000);
    Console.WriteLine("Network is Up");
    manualResetEvent.Set();
  }
});
```

Из предыдущего кода видно, что каждые пять секунд мы оповещали о событии ручного сброса с помощью `ManualResetEvent.Set()`. И каждые две секунды его отключали, используя `ManualResetEvent.Reset()`. Представленный ниже код выполняет обращения к службе:

```
for (int i = 0; i < 3; i++) {
  Parallel.For(0, 5, (j) => {
    Console.WriteLine($"Task with id {Task.CurrentId} waiting
      for network to be up");
    manualResetEvent.WaitOne();
    Console.WriteLine($"Task with id {Task.CurrentId} making
      service call");
    DummyServiceCall();
  });
  Thread.Sleep(2000);
}
```

В предыдущем коде мы создали цикл `for`, который производит пять задач в каждой итерации, между которыми есть интервал сна (sleep interval) в две секунды.

До осуществления обращений к сервису мы ожидаем готовности сети, вызывая `ManualResetEvent.WaitOne()`.

Если мы запустим предыдущий код, то получим вывод:

```
C:\Program Files\dotnet\dotnet.exe
Task with id 4 waiting for network to be up
Task with id 3 waiting for network to be up
Task with id 5 waiting for network to be up
Task with id 6 waiting for network to be up
Task with id 7 waiting for network to be up
Network is down
Network is Up
Task with id 7 making service call
Task with id 5 making service call
Task with id 3 making service call
Task with id 4 making service call
Task with id 6 making service call
Network is down
Task with id 14 waiting for network to be up
Task with id 17 waiting for network to be up
Task with id 16 waiting for network to be up
Task with id 15 waiting for network to be up
Task with id 18 waiting for network to be up
Network is down
Network is Up
Task with id 18 making service call
Task with id 15 making service call
Task with id 14 making service call
Task with id 16 making service call
Task with id 17 making service call
Network is down
Task with id 25 waiting for network to be up
Task with id 26 waiting for network to be up
Task with id 27 waiting for network to be up
Task with id 28 waiting for network to be up
Task with id 29 waiting for network to be up
Network is down
Network is Up
Task with id 29 making service call
Task with id 28 making service call
Task with id 25 making service call
Task with id 26 making service call
Task with id 27 making service call
Network is down
Network is Up
Network is down
Network is down
Network is Up
```

Как видите, пять задач запускаются и моментально блокируются в связи с ожиданием включения сети. Через пять секунд, когда сеть включается, мы сообщаем об этом с помощью метода `Set()`, и все пять потоков пропускаются для обращения к сервису. И так с каждой итерацией цикла `for`.

WaitHandles

Класс `System.Threading.WaitHandle` наследуется от `MarshalByRefObject` и используется для синхронизации запущенных в приложении потоков. Блокировка и отправка сигналов применяются для синхронизации потоков с помощью управления ожиданием. Потоки могут быть заблокированы вызовом любого метода класса `WaitHandle`. Разблокируются они в зависимости от выбранного типа отправки сигналов. Методы класса `WaitHandle`:

- `WaitOne`. Блокирует вызывающий поток до тех пор, пока не получит сигнал от конкретного объекта, управляющего ожиданием;
- `WaitAll`. Блокирует вызывающий поток до тех пор, пока не получит сигнал от всех указанных объектов, управляющих ожиданием.

Ниже приведен пример, демонстрирующий работу `WaitAll`:

```
public static bool WaitAll(System.Threading.WaitHandle[]
  waitHandles, TimeSpan timeout, bool exitContext);
```

В примере используются два потока для имитации двух различных служебных вызовов. Оба потока выполняются параллельно и ожидают в `WaitHandle.WaitAll(WaitHandles)` перед выводом суммы в консоль:

```
static int _dataFromService1 = 0;
static int _dataFromService2 = 0;
private static void WaitAll() {
  List waitHandles = new List < WaitHandle > {
    new AutoResetEvent(false),
    new AutoResetEvent(false)
  };
  ThreadPool.QueueUserWorkItem(new WaitCallback(
    FetchDataFromService1), waitHandles.First());
  ThreadPool.QueueUserWorkItem(new WaitCallback
    FetchDataFromService2), waitHandles.Last());
  //Ожидает, пока все потоки (waitHandles) вызовут метод Set()
  //т. е. ждет, пока данные будут возвращены из обоих сервисов
  WaitHandle.WaitAll(waitHandles.ToArray());
  Console.WriteLine($"The Sum is
    {_dataFromService1 + _dataFromService2}");
}

private static void FetchDataFromService1(object state) {
  Thread.Sleep(1000);
  _dataFromService1 = 890;
  var autoResetEvent = state as AutoResetEvent;
  autoResetEvent.Set();
}

private static void FetchDataFromService2(object state) {
  Thread.Sleep(1000);
  _dataFromService2 = 3;
  var autoResetEvent = state as AutoResetEvent;
  autoResetEvent.Set();
}
```

Вывод предыдущего кода:

- `WaitAny`. Блокирует вызывающий поток, пока не получит сигнал от любого указанного объекта, управляющего ожиданием.
  Сигнатура метода `WaitAny`:

  ```
  public static int WaitAny(System.Threading.WaitHandle[] waitHandles);
  ```

  Вот пример, в котором используются два потока для поиска элемента. Оба потока выполняются параллельно, а программа ожидает завершения любого потока в методе `WaitHandle.WaitAny(waitHandles)` перед выводом индекса элемента в консоль.
  У нас есть два метода, бинарный поиск и линейный поиск, которые выполняют поиск с использованием соответствующих алгоритмов. Чем быстрее мы получим результат от любого из этих методов, тем лучше. Для этого мы можем передать сигнал при помощи `AutoResetEvent` и сохранить результаты в глобальных переменных `findIndex` и `winnerAlgo`:

  ```
  static int findIndex = -1;
  static string winnerAlgo = string.Empty;
  private static void BinarySearch(object state) {
    dynamic data = state;
    int[] x = data.Range;
    int valueToFind = data.ItemToFind;
    AutoResetEvent autoResetEvent = data.WaitHandle
    as AutoResetEvent;
    //Поиск элемента с помощью алгоритма двоичного поиска из .NET
    int foundIndex = Array.BinarySearch(x, valueToFind);
    //Сохранить результат глобально
    Interlocked.CompareExchange(ref findIndex, foundIndex, -1);
    Interlocked.CompareExchange(ref winnerAlgo,
      "BinarySearch", string.Empty);
    //Сигнальное событие
    autoResetEvent.Set();
  }

  public static void LinearSearch(object state) {
    dynamic data = state;
    int[] x = data.Range;
    int valueToFind = data.ItemToFind;
    AutoResetEvent autoResetEvent = data.WaitHandle as
    AutoResetEvent;
    int foundIndex = -1;
    //Поиск элемента линейно с помощью цикла for
    for (int i = 0; i < x.Length; i++) {
      if (valueToFind == x[i]) {
        foundIndex = i;
      }
    }
    //Сохранить результат глобально
    Interlocked.CompareExchange(ref findIndex, foundIndex, -1);
    Interlocked.CompareExchange(ref winnerAlgo, "LinearSearch
      string.Empty);
    //Сигнальное событие
  ```

```
    autoResetEvent.Set();
  }
```

Следующий код параллельно вызывает оба алгоритма, используя `ThreadPool`:

```
private static void AlgoSolverWaitAny() {
  WaitHandle[] waitHandles = new WaitHandle[] {
    new AutoResetEvent(false), new AutoResetEvent(false)
  };
  var itemToSearch = 15000;
  var range = Enumerable.Range(1, 100000).ToArray();
  ThreadPool.QueueUserWorkItem(new WaitCallback(LinearSearch), new {
    Range = range, ItemToFind = itemToSearch,
    WaitHandle = waitHandles[0]
  });
  ThreadPool.QueueUserWorkItem(new WaitCallback(BinarySearch),
    new {
      Range = range, ItemToFind = itemToSearch,
      WaitHandle = waitHandles[1]
    });
  WaitHandle.WaitAny(waitHandles);
  Console.WriteLine($"Item found at index {findIndex} and
                     faster algo is {winnerAlgo}");
}
```

- `SignalAndWait`. Этот метод вызывает `Set()` у одного объекта, управляющего ожиданием, и метод `WaitOne` у другого. В многопоточной среде этот метод используют для освобождения одного потока и отмены ожидания у следующего потока:

```
public static bool SignalAndWait(System.Threading.WaitHandle
toSignal, System.Threading.WaitHandle toWaitOn);
```

Легковесные примитивы синхронизации

Платформа .NET Framework предоставляет легковесные примитивы синхронизации (lightweight synchronization primitives) с более высокими показателями, чем у их «традиционных» аналогов. Они не зависят от объектов ядра ОС, поэтому работают только внутри самого процесса. Такие примитивы лучше использовать при небольшом времени ожидания. Их можно разделить на две категории, с которыми мы познакомимся в этом разделе.

Slim locks

Slim locks – это облегченные реализации устаревших примитивов синхронизации, которые способны повысить производительность за счет снижения накладных расходов.

В таблице показаны устаревшие примитивы синхронизации и их slim-аналоги:

Устаревшие	Slim
ReaderWriterLock	ReaderWriterLockSlim
Semaphore	SemaphoreSlim
ManualResetEvent	ManualResetEventSlim

Давайте подробнее рассмотрим slim locks.

ReaderWriterLockSlim

`ReaderWriterLockSlim` является упрощенной реализацией `ReaderWriterLock`. Этот примитив представляет собой блокировку, которая может использоваться для управления защищенными ресурсами, предоставляя доступ на чтение нескольким потокам и доступ на запись только одному потоку.

В следующем примере используется `ReaderWriterLockSlim`, защищающий доступ к списку, совместно используемому тремя потоками чтения и одним потоком записи:

```
static ReaderWriterLockSlim _readerWriterLockSlim = new
ReaderWriterLockSlim();
static List _list = new List();
private static void ReaderWriteLockSlim() {
  Task writerTask = Task.Factory.StartNew(WriterTask);
  for (int i = 0; i < 3; i++) {
    Task readerTask = Task.Factory.StartNew(ReaderTask);
  }
}

static void WriterTask() {
  for (int i = 0; i < 4; i++) {
    try {
      _readerWriterLockSlim.EnterWriteLock();
      Console.WriteLine($"Entered WriteLock on Task
                      {Task.CurrentId}");
      int random = new Random().Next(1, 10);
      _list.Add(random);
      Console.WriteLine($"Added {random} to list on Task
                      {Task.CurrentId}");
      Console.WriteLine($"Exiting WriteLock on Task
                      {Task.CurrentId}");
    } finally {
      _readerWriterLockSlim.ExitWriteLock();
    }
    Thread.Sleep(1000);
  }
}
```

```
static void ReaderTask() {
  for (int i = 0; i < 2; i++) {
    _readerWriterLockSlim.EnterReadLock();
    Console.WriteLine($"Entered ReadLock on Task {Task.CurrentId}");
    Console.WriteLine($"Items: {_list.Select(j=>j.ToString())
                      .Aggregate((a, b) => a + ", " + b)}
                      on Task {Task.CurrentId}");
    Console.WriteLine($"Exiting ReadLock on Task {Task.CurrentId}");
    _readerWriterLockSlim.ExitReadLock();
    Thread.Sleep(1000);
  }
}
```

Вывод этого кода:

```
Select C:\Program Files\dotnet\dotnet.exe
Entered WriteLock on Task 1
Added 9 to list on Task 1
Exiting WriteLock on Task 1
Entered ReadLock on Task 2
Entered ReadLock on Task 4
Entered ReadLock on Task 3
Items: 9 on Task 3
Exiting ReadLock on Task 3
Items: 9 on Task 4
Exiting ReadLock on Task 4
Items: 9 on Task 2
Exiting ReadLock on Task 2
Entered WriteLock on Task 1
Added 3 to list on Task 1
Exiting WriteLock on Task 1
Entered ReadLock on Task 4
Entered ReadLock on Task 3
Items: 9,3 on Task 3
Exiting ReadLock on Task 3
Items: 9,3 on Task 4
Exiting ReadLock on Task 4
Entered ReadLock on Task 2
Items: 9,3 on Task 2
Exiting ReadLock on Task 2
Entered WriteLock on Task 1
Added 9 to list on Task 1
Exiting WriteLock on Task 1
Entered WriteLock on Task 1
Added 5 to list on Task 1
Exiting WriteLock on Task 1
```

SemaphoreSlim

`SemaphoreSlim` является упрощенной реализацией `semaphore`. Он ограничивает доступ нескольким потокам к защищенному ресурсу.

Это slim-версия программы `semaphore`, которая уже демонстрировалась в данной главе:

```
private static void ThrottlerUsingSemaphoreSlim() {
  var range = Enumerable.Range(1, 12);
  SemaphoreSlim semaphore = new SemaphoreSlim(3, 3);
  range.AsParallel().AsOrdered().ForAll(i => {
    try {
      semaphore.Wait();
      Console.WriteLine($"Index {i} making service call
                          using Task {Task.CurrentId}");
      //Имитация Http-вызова
      CallService(i);
      Console.WriteLine($"Index {i} releasing semaphore
                          using Task {Task.CurrentId}");
    } finally {
      semaphore.Release();
    }
  });
}

private static void CallService(int i) {
  Thread.Sleep(1000);
}
```

Помимо замены класса `Semaphore` на `SemaphoreSlim`, разница здесь заключается в том, что теперь мы используем метод `Wait()` вместо `WaitOne()`. Данный подход более целесообразен, поскольку в подобном случае пропускается несколько потоков сразу.

Еще одним существенным отличием является то, что `SemaphoreSlim` всегда создается как локальный семафор, в отличие от обычного `semaphore`, который может также создаваться на уровне ОС.

ManualResetEventSlim

`ManualResetEventSlim` – это упрощенная реализация `ManualResetEvent`. Он работает эффективнее и потребляет меньше ресурсов, чем `ManualResetEvent`.

Используя следующий синтаксис, можно создать slim-объект, как и в `ManualResetEvent`:

```
ManualResetEventSlim manualResetEvent = new ManualResetEventSlim(false);
```

Подобно другим прототипам slim, одним из главных отличий является замена метода `WaitOne()` на `Wait()`.

Вы можете попытаться запустить код из примера `ManualResetEvent` с данными изменениями и проверить его работу.

События Barrier и CountDownEvent

Платформа .NET Framework имеет несколько встроенных примитивов на базе сигналов, которые помогают синхронизировать несколько потоков, освобождая нас от написания логики синхронизации, так как она реализуется

на уровне внутренних структур данных у этих классов. В текущем разделе мы обсудим два значимых сигнальных примитива: `CountdownEvent` и `Barrier`.

- **CountDownEvent**. `System.Threading.CountDownEvent` относится к событию, которое возникает, когда его счетчик становится равным 0.
- **Barrier**. Класс `Barrier` позволяет нескольким потокам работать без главного контролирующего потока. `Barrier` работает хорошо при параллельной и поэтапной реализации.

Примеры использования Barrier и CountdownEvent

Предположим, например, что нам необходимо получить данные от двух сервисов. Перед их извлечением из сервиса 1 его необходимо сперва запустить и уже после получения данных закрыть. Только когда сервис-1 закроется, мы сможем запустить сервис 2 и извлечь из него данные. Данные должны быть получены в кратчайшие сроки. Давайте создадим код в соответствии с требованиями такого сценария.

Создайте `Barrier` для 5 участников:

```
static Barrier serviceBarrier = new Barrier(5);
```

Создайте два события `CountdownEvents`, вызывающих запуск или закрытие сервисов при прохождении через них шести потоков. Вместе с пятью рабочими задачами в них также будет участвовать задача, управляющая запуском или закрытием сервисов:

```
static CountdownEvent serviceHost1CountdownEvent = new CountdownEvent(6);
static CountdownEvent serviceHost2CountdownEvent = new CountdownEvent(6);
```

В конце создайте еще одно событие `CountdownEvent` с числом `5`. Оно будет относиться к потокам, которые могут проходить до того, как событию поступит сигнал. `finishCountdownEvent` сработает, когда все рабочие задачи закончат свое выполнение:

```
static CountdownEvent finishCountdownEvent = new CountdownEvent(5);
```

Полученная реализация `serviceManagerTask`:

```
Task serviceManager = Task.Factory.StartNew(() => {
  //Блокируем до тех пор, пока имя сервиса не будет установлено
  //каким-нибудь из потоков
  while (string.IsNullOrEmpty(_serviceName))
    Thread.Sleep(1000);
  string serviceName = _serviceName;
  HostService(serviceName);
  //Сообщаем другим потокам о запуске сервиса 1
  serviceHost1CountdownEvent.Signal();
  // Ожидаем, пока рабочие задачи завершат вызовы сервиса 1
  serviceHost1CountdownEvent.Wait();
```

```
  //Блокируем до тех пор, пока имя сервиса не будет установлено
  //каким-нибудь из потоков
  while (_serviceName != "Service2")
    Thread.Sleep(1000);
  Console.WriteLine($"All tasks completed for service
                    {serviceName}.");
  //Закрываем текущий сервис и запускаем другой
  CloseService(serviceName);
  HostService(_serviceName);
  //Сообщаем другим потокам о запуске сервиса 2
  serviceHost2CountdownEvent.Signal();
  serviceHost2CountdownEvent.Wait();
  // Ожидаем, пока рабочие задачи завершат вызовы сервиса 2
  finishCountdownEvent.Wait();
  CloseService(_serviceName);
  Console.WriteLine($"All tasks completed for service
    {_serviceName}.");
});
```

Ниже представлен метод, который выполняется рабочими задачами:

```
private static void GetDataFromService1And2(int j) {
  _serviceName = "Service1";
  serviceHost1CountdownEvent.Signal();
  Console.WriteLine($"Task with id  {Task.CurrentId} signaled
                countdown event and waiting for service to start");
  //Ожидаем запуска сервиса
  serviceHost1CountdownEvent.Wait();
  Console.WriteLine($"Task with id {Task.CurrentId} fetching
                data from service ");
  serviceBarrier.SignalAndWait();
  // Изменяем имя сервиса
  _serviceName = "Service2";
  //Сообщаем о событии Countdown
  serviceHost2CountdownEvent.Signal();
  Console.WriteLine($"Task with id {Task.CurrentId} signaled
                    countdown event and waiting for service
                    to start");
  serviceHost2CountdownEvent.Wait();
  Console.WriteLine($"Task with id {Task.CurrentId} fetching
                    data from service ");
  serviceBarrier.SignalAndWait();
  //Сообщаем о финальном событии Countdown
  finishCountdownEvent.Signal();
}
// Наконец, создаем рабочие задачи
for (int i = 0; i < 5; ++i) {
  int j = i;
  tasks[j] = Task.Factory.StartNew(() => {
    GetDataFromService1And2(j);
  });
}
```

```
Task.WaitAll(tasks);
Console.WriteLine("Fetch completed");
```

Вывод предыдущего кода представлен ниже:

```
C:\Program Files\dotnet\dotnet.exe
Task with id 5 signaled countdown event and waiting for service to start
Task with id 6 signaled countdown event and waiting for service to start
Task with id 4 signaled countdown event and waiting for service to start
Task with id 3 signaled countdown event and waiting for service to start
Task with id 2 signaled countdown event and waiting for service to start
Service Service1 hosted
Task with id 2 fetching data from service
Task with id 5 fetching data from service
Task with id 3 fetching data from service
Task with id 6 fetching data from service
Task with id 4 fetching data from service
Task with id 6 signaled countdown event and waiting for service to start
Task with id 5 signaled countdown event and waiting for service to start
Task with id 4 signaled countdown event and waiting for service to start
Task with id 2 signaled countdown event and waiting for service to start
Task with id 3 signaled countdown event and waiting for service to start
All tasks completed for service Service1.
Service Service1 closed
Service Service2 hosted
Task with id 2 fetching data from service
Task with id 3 fetching data from service
Task with id 6 fetching data from service
Task with id 5 fetching data from service
Task with id 4 fetching data from service
Service Service2 closed
All tasks completed for service Service2.
Fetch completed
```

В этом разделе мы рассмотрели различные встроенные сигнальные примитивы, которые помогают упростить синхронизацию кода без усложнения процесса разработки. Блокировка по-прежнему влечет к снижению производительности, поскольку она требует переключения контекста. В следующем разделе мы рассмотрим некоторые spin-методы, которые могут помочь в уменьшении расходов на переключение контекста.

SpinWait

В начале этой главы мы уже упоминали, что использование вращения (spin) для короткого ожидания будет намного эффективнее, чем полноценная блокировка. Spin минимизирует накладные расходы, которые связаны с переключением контекста.

Мы можем создать объект `SpinWait` следующим образом:

```
var spin = new SpinWait();
```

Вызов данной команды позволяет нам обратиться к `spin`:

```
spin.SpinOnce();
```

SpinLock

Блокировки и блокирующие примитивы могут значительно снизить производительность при коротком времени ожидания. SpinLock предоставляет облегченную и низкоуровневую альтернативу блокировке. SpinLock работает с типами данных, которые принимают значения, поэтому если мы хотим использовать один и тот же объект в нескольких местах, нам нужно использовать ссылку на него. Для повышения производительности, даже когда SpinLock еще не получил блокировку, он выдает квант времени потоку, чтобы программа очистки памяти могла эффективно отработать. По умолчанию SpinLock не следит за тем, какие потоки были заблокированы. Однако можно активировать эту возможность. Ее рекомендуется включать только для устранения неисправностей и ошибок, так как это снижает производительность.

Создайте объект SpinLock, как показано в примере:

```
static SpinLock _spinLock = new SpinLock();
```

Создайте метод, который будет вызываться различными потоками, и обновите глобальный статический список:

```
static List<int> _itemsList = new List<int>();
private static void SpinLock(int number) {
  bool lockTaken = false;
  try {
    Console.WriteLine($"Task {Task.CurrentId} Waiting for lock");
    _spinLock.Enter(ref lockTaken);
    Console.WriteLine($"Task {Task.CurrentId} Updating list");
                     _itemsList.Add(number);
  } finally {
    if (lockTaken) {
      Console.WriteLine($"Task {Task.CurrentId} Exiting Update");
      _spinLock.Exit(false);
    }
  }
}
```

Как вы можете видеть, блокировка устанавливается с помощью _spinLock.Enter(ref lockTaken) и сбрасывается через _spinLock.Exit(false). Все, что расположено между этими двумя операторами, будет выполняться лишь одним потоком одновременно.

Теперь вызовем этот метод в параллельном цикле:

```
Parallel.For(1, 5, (i) => SpinLock(i));
```

Ниже представлен вывод, который бы мы получили, если бы использовали примитивы блокировки:

```
Task 2 Waiting for lock
Task 3 Waiting for lock
Task 4 Waiting for lock
Task 1 Waiting for lock
Task 2 Updting list
Task 2 Exiting Update
Task 4 Updting list
Task 4 Exiting Update
Task 1 Updting list
Task 1 Exiting Update
Task 3 Updting list
Task 3 Exiting Update
```

Как правило, в небольших задачах можно полностью предотвратить переключение контекста с помощью spin-метода.

Резюме

В этой главе мы познакомились с примитивами синхронизации .NET Core. Примитивы синхронизации являются обязательным условием при написании и проверке корректности параллельного кода даже тогда, когда в работе задействовано несколько потоков. Однако использование примитивов синхронизации сопровождается накладными расходами, поэтому по возможности рекомендуется использовать их slim-аналоги.

Также мы узнали о сигнальных примитивах, использование которых может помочь потокам работать с внешними событиями. Кроме этого, мы обсудили события barrier и countdown, которые позволяют избежать проблем синхронизации кода без написания дополнительной логики. И наконец, продемонстрировали spin-методы, в частности `SpinLock` и `SpinWait`, которые снижают накладные расходы, возникающие из-за блокировки кода.

В следующей главе мы познакомимся со структурами данных .NET Core, которые синхронизируются автоматически и работают параллельно.

Вопросы

1. Что из представленного можно использовать для межпроцессной синхронизации?
    1. Lock
    2. Interlocked.Increment
    3. Interlocked.MemoryBarrierProcessWide
2. Какой из барьеров памяти является некорректным?
    1. Read memory barrier (барьер чтения памяти)
    2. Half memory barrier (полубарьер памяти)
    3. Full memory barrier (полный барьер памяти)
    4. Read and execute memory barrier (барьер чтения и исполнения памяти)

3. Из какого состояния мы не можем возобновить поток?
    1. `WaitSleepJoin`
    2. `Suspended`
    3. `Aborted`
4. Где безымянный `semaphore` обеспечивает синхронизацию?
    1. Внутри процесса
    2. За пределами процесса
5. Какие из этих конструкций поддерживают отслеживание потоков?
    1. `SpinWait`
    2. `SpinLock`

Глава 6

Использование параллельных коллекций

В предыдущей главе мы рассмотрели несколько вариантов параллельного программирования, в которых у множества потоков не было доступа к одновременному использованию ресурсов. Вообще, примитивы синхронизации не так просто использовать в реальных проектах. Зачастую общий ресурс представляет собой коллекцию, которая должна быть прочитана и записана несколькими потоками.

Поскольку к коллекции можно получить доступ различными способами (например, с помощью `Enumerate`, `Read`, `Write`, `Sort` или `Filter`), сложно написать пользовательскую коллекцию с управляемой синхронизацией на основе примитивов. В связи с этим потребность в потокобезопасных коллекциях всегда оставалась актуальной.

В этой главе мы узнаем о конструкциях C#, которые способствуют разработке параллельного кода. Ниже представлены темы, которые мы рассмотрим в этой главе:

- введение в параллельные коллекции;
- множественный сценарий производителя/потребителя.

Технические требования

Для освоения этой главы вам понадобится уверенное знание TPL и C#. Исходный код для этой главы доступен на GitHub: https://github.com/PacktPublishing/Hands-On-Parallel-Programming-with-C-8-and-.NET-Core-3/tree/master/Chapter06.

Введение в параллельные коллекции

Множество потокобезопасных коллекций было добавлено в.NET, начиная с версии .NET Framework 4. В их числе было и новое пространство имен `System.Threading.Concurrent`, которое включало в себя такие конструкции, как:

- `IProducerConsumerCollection<T>`;
- `BlockingCollection<T>`;
- `ConcurrentDictionary<TKey,TValue>`.

При использовании вышеупомянутых структур необходимость в дополнительной синхронизации отпадает, а чтение и обновление в них могут выполняться атомарно.

Для коллекций понятие потокобезопасности не является абсолютно новым. Свойство `Synchronized` было доступно еще в старых коллекциях `ArrayList` и `Hashtable`, к которым оно обеспечивало потокобезопасное подключение. Однако из-за этого страдала производительность: для достижения потокобезопасности коллекция полностью блокировалась при каждой операции чтения и записи.

Параллельные коллекции используют облегченные slim-примитивы синхронизации, такие как `SpinLock`, `SpinWait`, `SemaphoreSlim` и `CountdownEvent`, и, следовательно, не задействуют ядро ОС. Как мы уже знаем, при коротких интервалах ожидания ждущий цикл работает намного эффективнее блокировки. Есть также дополнительные алгоритмы, которые автоматически задействуют блокировки на уровне ядра ОС при увеличении времени ожидания.

Знакомство с IProducerConsumerCollection<T>

Коллекции производителя (producer) и потребителя (consumer) – это коллекции, которые предоставляют эффективные альтернативы таким универсальным аналогам, как `Stack<T>` и `Queue<T>`. Любая коллекция производителя или потребителя должна разрешать пользователю добавлять и удалять элементы. В .NET Framework есть интерфейс `IProducerConsumerCollection<T>`, который представлен потокобезопасными стеками (stack), очередями (queue) и мультимножествами (bag). Ниже указаны классы, реализующие этот интерфейс:

- `ConcurrentQueue<T>`;
- `ConcurrentStack<T>`;
- `ConcurrentBag<T>`.

В интерфейсе приводятся два важных метода: `TryAdd` и `TryTake`. Для `TryAdd` синтаксис выглядит следующим образом:

```
bool TryAdd(T item);
```

Метод `TryAdd` добавляет элемент и возвращает значение `true`. Если же с добавлением элемента возникнут проблемы, то он вернет значение `false`.

А так выглядит синтаксис `TryTake`:

```
bool TryTake(out T item);
```

Метод `TryTake` удаляет элемент и возвращает значение `true`. Если же с удалением элемента будут проблемы, тогда вам вернется значение `false`.

Использование ConcurrentQueue <T>

В прикладном программировании параллельные очереди могут использоваться для решения сценариев производитель/потребитель. В шаблоне производитель/потребитель данные производятся и потребляются одним или несколькими потоками, из-за чего между ними возникает состояние гонки.

Проблему гонки потоков можно решить при помощи представленных подходов:

- использование очередей;
- использование ConcurrentQueue<T>.

В зависимости от того, какой поток (производитель/потребитель) отвечает за добавление/потребление данных, шаблоны можно классифицировать следующим образом:

- **чистый** (pure) **производитель–потребитель**, когда поток только записывает или только считывает данные, но не можете делать и то, и другое;
- **смешанный** (mixed) **производитель–потребитель**, когда любой поток может записывать и считывать данные.

Попробуем сначала решить задачу производитель–потребитель с помощью очередей.

Очереди в решении задач производитель–потребитель

В этом примере мы создадим сценарий производителя и потребителя, используя очереди, определенные в пространстве имен System.Collections. У нас есть несколько задач, которые будут считываться и записываться в очередь, а нам нужно убедиться в атомарности их чтения и записи.

1. Для начала создадим queue (очередь) и заполним ее данными:

```
Queue<int> queue = new Queue<int>();
for (int i = 0; i < 500; i++) {
  queue.Enqueue(i);
}
```

2. Определим переменную, которая будет содержать конечный результат:

```
int sum = 0;
```

3. Затем создадим параллельный цикл, который будет считывать элемент из очереди с помощью нескольких задач и потокобезопасно добавлять сумму в переменную sum, обозначенную нами ранее:

```
Parallel.For(0, 500, (i) => {
  int localSum = 0;
  int localValue;
  while (queue.TryDequeue(out localValue)) {
    Thread.Sleep(10);
    localSum += localValue;
  }
  Interlocked.Add(ref sum, localSum);
});
Console.WriteLine($"Calculated Sum is {sum} and should be
                      {Enumerable.Range(0, 500).Sum()}");
```

При запуске программы мы получаем следующий результат. Как видите, это не тот вывод, который мы ожидали получить. Это произошло из-за попытки одновременного чтения, в связи с чем между задачами возникло состояние гонки:

Чтобы сделать предыдущую программу потокобезопасной, мы можем заблокировать критическую секцию, изменив код параллельного цикла, как показано ниже:

```
Parallel.For(0, 500, (i) => {
  int localSum = 0;
  int localValue;
  Monitor.Enter(_locker);
  while (cq.TryDequeue(out localValue)) {
    Thread.Sleep(10);
    localSum += localValue;
  }
  Monitor.Exit(_locker);
  Interlocked.Add(ref sum, localSum);
});
```

Аналогичным образом нам нужно синхронизировать все операции чтения/записи в очередь, которые выполняются параллельно. Запустив предыдущий код, мы получим следующий результат:

Как видите, мы получили ожидаемое число. Тем не менее возникают и дополнительные издержки на синхронизацию, которые могут привести к взаимоблокировке при одновременных операциях чтения или записи.

Решение проблемы с использованием параллельных очередей

Решить проблему синхронизации в сценарии производитель–потребитель можно при помощи `System.Collections.Concurrent.ConcurrentQueue`, потокобезопасной версии очереди. Для этого изменим предыдущий код с помощью параллельной очереди:

```
private static void ProducerConsumerUsingConcurrentQueues() {
  // Создаем очередь
  ConcurrentQueue<int> cq = new ConcurrentQueue<int> ();
  // Заполняем очередь
  for (int i = 0; i < 500; i++) {
    cq.Enqueue(i);
  }
  int sum = 0;
  Parallel.For(0, 500, (i) => {
    int localSum = 0;
    int localValue;
```

```
    while (cq.TryDequeue(out localValue)) {
      Thread.Sleep(10);
      localSum += localValue;
    }
    Interlocked.Add(ref sum, localSum);
  });
  Console.WriteLine($"outerSum = {sum}, should be
                      {Enumerable.Range(0, 500).Sum()}");
}
```

Как видно, мы всего лишь заменили `Queue<int>` на `ConcurrentQueue<int>` в написанном ранее коде, в котором были издержки на синхронизацию. С `ConcurrentQueue` нам не нужны другие примитивы синхронизации.

Ниже показан результат, который возникает при запуске кода:

Как и `Queue<T>`, `ConcurrentQueue<T>` также работает в режиме **«первый пришел – первый ушел»** (First In, First Out; FIFO).

Производительность Queue<T> в сравнении с ConcurrentQueue<T>

Следует использовать `ConcurrentQueue` в следующих сценариях, в которых будет небольшое или значительное преимущество в производительности перед `Queue`:

- в чистом сценарии производитель–потребитель с незначительным временем обработки каждого элемента;
- в чистом сценарии производитель–потребитель, в котором будут только один поток-производитель и один поток-потребитель;
- в чистых, а также смешанных сценариях производитель–потребитель, в которых время обработки составляет 500 или более флопс (**Floating-Point Operations Per Second**, или, сокращенно, **FLOPS** – количество операций с плавающей запятой в секунду).

В целях повышения производительности лучше использовать очереди (queue) вместо параллельных очередей (concurrent queues) в смешанном сценарии производитель–потребитель, в котором время обработки каждого элемента меньше 500 FLOPS.

Использование ConcurrentStack <T>

`ConcurrentStack<T>` – это параллельная версия `Stack<T>`, реализующая интерфейс `IProducerConsumerCollection<T>`. Можно помещать или извлекать элементы из стека, который работает в формате «последний пришел – последний ушел» (Last In, First Out; LIFO). Стек не предусматривает использование

блокировки на уровне ядра, вместо этого он основывается на операциях вращения (spinning) и сравнения-переключения (compare-and-swap) во избежание гонки.

Ниже перечислены некоторые важные методы класса `ConcurrentStack<T>`:

- `Clear`: удаляет все элементы из коллекции;
- `Count`: возвращает количество элементов в коллекцию;
- `isEmpty`: возвращает значение `true` при пустой коллекции;
- `Push (T item)`: добавляет элемент в коллекцию;
- `TryPop (out T result)`: удаляет элемент из коллекции и, при его удалении, возвращает значение `true`; в противном случае вернется значение `false`;
- `PushRange (T [] items)`: добавляет диапазон элементов в коллекцию; операция выполняется атомарно;
- `TryPopRange (T [] items)`: удаляет диапазон элементов из коллекции.

Рассмотрим создание экземпляра параллельного стека.

Создание параллельного стека

Можно создать экземпляр параллельного стека и добавить элементы, как показано ниже:

```
ConcurrentStack<int> concurrentStack = new ConcurrentStack<int>();
concurrentStack.Push (1);
concurrentStack.PushRange(new[] { 1,2,3,4,5 });
```

Получить элементы из стека можно следующим образом:

```
int localValue;
concurrentStack.TryPop(out localValue)
concurrentStack.TryPopRange (new[] {1,2,3,4,5});
```

Ниже представлен полный код, который создает параллельный стек, добавляет в него элементы и параллельно выполняет по ним итерации:

```
private static void ProducerConsumerUsingConcurrentStack() {
  // Создаем очередь
  ConcurrentStack<int> concurrentStack = new ConcurrentStack<int>();
  // Заполняем очередь
  for (int i = 0; i < 500; i++) {
    concurrentStack.Push(i);
  }
  concurrentStack.PushRange(new [] {1,2,3,4,5});
  int sum = 0;
  Parallel.For(0, 500, (i) => {
    int localSum = 0;
    int localValue;
    while (concurrentStack.TryPop(out localValue)) {
      Thread.Sleep(10);
      localSum += localValue;
    }
    Interlocked.Add(ref sum, localSum);
  });
```

```
  Console.WriteLine($"outerSum = {sum}, should be 124765");
}
```

Получаем следующий результат:

Использование ConcurrentBag<T>

В отличие от `ConcurrentStack` и `ConcurrentQueues`, которые сортируют элементы при их хранении и извлечении, `ConcurrentBag<T>` является неупорядоченной коллекцией. `ConcurrentBag<T>` оптимизирован для сценариев, в которых одни и те же потоки работают как производители и как потребители. `ConcurrentBag` предусматривает алгоритм перехвата работы и поддерживает локальную очередь для каждого потока.

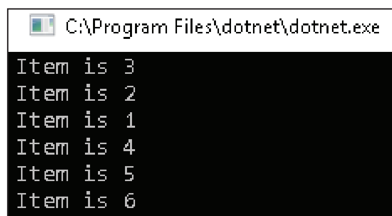
Код ниже создает `ConcurrentBag` и добавляет или получает из него элементы:

```
ConcurrentBag<int> concurrentBag = new ConcurrentBag<int>();
// Добавляем элемент в коллекцию
concurrentBag.Add(10);
int item;
// Получаем элемент из коллекции
concurrentBag.TryTake(out item);
```

Полный код выглядит следующим образом:

```
static ConcurrentBag<int> concurrentBag = new ConcurrentBag<int>();
private static void ConcurrentBackDemo() {
  ManualResetEventSlim manualResetEvent = new
                                  ManualResetEventSlim(false);
  Task producerAndConsumerTask = Task.Factory.StartNew(() => {
    for (int i = 1; i <= 3; ++i) {
      concurrentBag.Add(i);
    }
    // Разрешаем второму потоку добавлять элементы
    manualResetEvent.Wait();
    while (concurrentBag.IsEmpty == false) {
      int item;
      if (concurrentBag.TryTake(out item)) {
        Console.WriteLine($"Item is {item}");
      }
    }
  });
  Task producerTask = Task.Factory.StartNew(() => {
    for (int i = 4; i <= 6; ++i) {
      concurrentBag.Add(i);
    }
    manualResetEvent.Set();
  });
}
```

Вывод кода представлен ниже:



Нам уже известно, что у каждого потока есть своя локальная очередь. Элементы 1, 2 и 3 добавляются в локальную очередь `producerAndConsumerTask`, а элементы 4, 5 и 6 – в локальную очередь `producerTask`. После того как `producerAndConsumerTask` добавит свои элементы, мы ожидаем, пока завершится размещение других элементов в `producerTask`. Как только все элементы будут размещены, `producerAndConsumerTask` начнет их извлекать. Сначала обработаются элементы 1, 2 и 3, поскольку они были помещены в локальную очередь, затем уже начнется обработка элементов в локальной очереди `producerTask`.

Использование BlockingCollection<T>

Класс `BlockingCollection<T>` является потокобезопасной коллекцией, реализующей интерфейс `IProduceConsumerCollection<T>`. Можно одновременно добавлять в коллекцию и удалять из нее элементы, не думая при этом о синхронизации – она обеспечивается автоматически.

В нашем примере будет два потока: производитель и потребитель. Поток-производитель производит данные, при этом можно установить лимит максимального количества элементов, которые он сможет произвести, прежде чем перейдет в спящий режим и заблокируется. Поток-потребитель потребляет данные, и когда они в коллекции заканчиваются, он блокируется. Поток-производитель разблокируется, когда поток-потребитель удаляет элементы из коллекции. Когда поток-производитель добавляет в коллекцию данные, разблокируется поток-потребитель.

Существует два важных аспекта блокировки коллекций:

- **ограничение (Bounding)**: означает, что мы можем привязать коллекцию к максимальному значению, после которого новые объекты уже не смогут добавляться, при этом поток-производитель переходит в спящий режим;
- **блокировка (Blocking)**: означает, что мы можем заблокировать поток-потребитель, когда коллекция станет пустой.

Рассмотрим, как создавать блокирующие коллекции.

Создание BlockingCollection<T>

Следующий код формирует новую коллекцию `BlockingCollection`, которая создает до 10 элементов, после чего переходит в заблокированное состояние, прежде чем потоки-потребители извлекут элементы:

```
BlockingCollection<int> blockingCollection =
                                  new BlockingCollection<int>(10);
```

Элементы могут быть добавлены в коллекцию следующим образом:

```
blockingCollection.Add(1);
blockingCollection.TryAdd(3, TimeSpan.FromSeconds(1));
```

Элементы могут быть удалены из коллекции следующим образом:

```
int item = blockingCollection.Take();
blockingCollection.TryTake(out item, TimeSpan.FromSeconds(1));
```

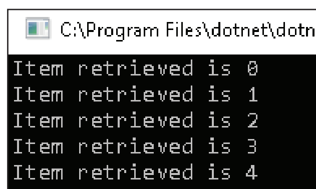
Когда элементов на добавление не остается, поток-производитель вызывает метод `CompleteAdding()`. Этот метод, в свою очередь, устанавливает свойство коллекции `IsAddingComplete` в значении `true`.

Поток-потребитель использует свойство `IsCompleted` при пустой коллекции и при установленном свойстве `IsAddingComplete` в значении `true`. Это свидетельствует о том, что все элементы обработаны и производитель больше не будет их добавлять.

Полный код представлен ниже:

```
BlockingCollection<int> blockingCollection =
                             new BlockingCollection<int>(10);
Task producerTask = Task.Factory.StartNew(() => {
  for (int i = 0; i < 5; ++i) {
    blockingCollection.Add(i);
  }
  blockingCollection.CompleteAdding();
});
Task consumerTask = Task.Factory.StartNew(() => {
  while (!blockingCollection.IsCompleted) {
    int item = blockingCollection.Take();
    Console.WriteLine($"Item retrieved is {item}");
  }
});
Task.WaitAll(producerTask, consumerTask);
```

Так выглядит вывод кода:



После того как мы познакомились с параллельными коллекциями, в следующем разделе мы попытаемся продвинуть сценарий производитель–потребитель и узнаем о специфике работы с множественными производителями/потребителями.

Сценарий с несколькими производителями и потребителями

В этом разделе мы рассмотрим, как работают блокирующие коллекции при наличии нескольких потоков-производителей и потребителей. Для наглядности мы создадим двух производителей и одного потребителя. Потоки-производители будут добавлять элементы, и как только все они вызовут метод `CompleteAdding`, потребитель начнет считывать элементы из коллекции.

1. Для начала создадим блокирующую коллекцию с несколькими производителями:

```
BlockingCollection<int>[] produceCollections =
                             new BlockingCollection<int>[2];
produceCollections[0] = new BlockingCollection<int>(5);
produceCollections[1] = new BlockingCollection<int>(5);
```

2. Затем создадим двух производителей, добавляющих элементы к коллекциям:

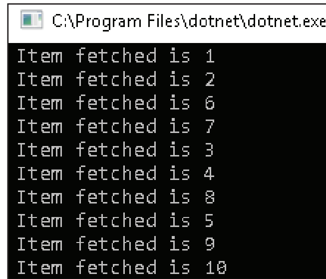
```
Task producerTask1 = Task.Factory.StartNew(() => {
  for (int i = 1; i <= 5; ++i) {
    produceCollections[0].Add(i);
    Thread.Sleep(100);
  }
  produceCollections[0].CompleteAdding();
});
Task producerTask2 = Task.Factory.StartNew(() => {
  for (int i = 6; i <= 10; ++i) {
    produceCollections[1].Add(i);
    Thread.Sleep(200);
  }
  produceCollections[1].CompleteAdding();
});
```

3. В конце напишем логику потребителя, которая будет получать добавленные элементы из обеих коллекций:

```
while (!produceCollections[0].IsCompleted ||
  !produceCollections[1].IsCompleted) {
  int item;
  BlockingCollection<int>.TryTakeFromAny(produceCollections,
    out item, TimeSpan.FromSeconds(1));
  if (item != default (int)) {
    Console.WriteLine($"Item fetched is {item}");
  }
}
```

Как вы можете видеть из предыдущего метода кода, `TryTakeFromAny` пытается прочитать элемент от нескольких производителей и вернуться, когда элемент будет доступен.

Так выглядит результат:



В программировании мы часто сталкиваемся со сценарием, при котором необходимо параллельное хранение данных в виде пары «ключ/значение». Для этого нам понадобится коллекция `ConcurrentDictionary`, с которой мы познакомимся в следующем разделе.

Использование ConcurrentDictionary<TKey,TValue>

`ConcurrentDictionary<TKey,TValue>` представляет собой потокобезопасный словарь. Он используется для хранения пар «ключ/значение», которые считываются и записываются потокобезопасным образом.

Ниже приводится создание `ConcurrentDictionary`:

```
ConcurrentDictionary<int, int> concurrentDictionary =
                            new ConcurrentDictionary<int, int>();
```

Элементы могут добавляться в словарь следующим образом:

```
concurrentDictionary.TryAdd(i, i * i);
string value = (i * i).ToString();
// Добавляем элемент, если он не существует, иначе обновляем элемент
concurrentDictionary.AddOrUpdate(i, value,
                               (key, val) => (key * key).ToString());
// Получаем элемент с ключом 5, если он существует,
// иначе добавляем ключ 5 со значением 25
concurrentDictionary.GetOrAdd(5, "25");
```

Элементы могут быть удалены из словаря, как показано ниже:

```
string value;
concurrentDictionary.TryRemove(5, out value);
```

В словаре элементы могут обновляться следующим образом:

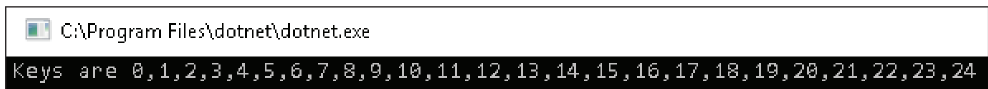
```
// Если ключ со значением 25 будет найден, то он будет обновлен до значения 30
concurrentDictionary.TryUpdate(5, "30", "25");
```

Ниже мы создадим код с двумя потоками-производителями, которые будут добавлять элементы в словарь. Производители создадут несколько копий

элементов, а словарь будет следить за их потокобезопасным добавлением, чтобы не возникли ошибки с повтором ключей. Когда потоки-производители выполнятся, потребитель прочитает все элементы при помощи свойства keys или values:

```
ConcurrentDictionary<int, string> concurrentDictionary =
                            new ConcurrentDictionary<int, string>();
Task producerTask1 = Task.Factory.StartNew(() => {
  for (int i = 0; i < 20; i++) {
    Thread.Sleep(100);
    concurrentDictionary.TryAdd(i, (i * i).ToString());
  }
});
Task producerTask2 = Task.Factory.StartNew(() => {
  for (int i = 10; i < 25; i++) {
    concurrentDictionary.TryAdd(i, (i * i).ToString());
  }
});
Task producerTask3 = Task.Factory.StartNew(() => {
  for (int i = 15; i < 20; i++) {
    Thread.Sleep(100);
    concurrentDictionary.AddOrUpdate(i, (i * i).ToString(),
      (key, value) => (key * key).ToString());
  }
});
Task.WaitAll(producerTask1, producerTask2);
Console.WriteLine("Keys are {0} ", string.Join(",", concurrentDictionary.Keys.Select(c =>
c.ToString()).ToArray()));
```

Результат выглядит следующим образом:



В этом разделе мы узнали, как могут быть использованы параллельные коллекции в сценариях производитель–потребитель. Плюсом параллельных коллекций является то, что они автоматически осуществляют синхронизации, не требуя дополнительных команд со стороны программиста.

Резюме

В этой главе мы обсудили потокобезопасные коллекции, входящие в состав .NET Framework. Параллельные коллекции доступны в пространстве имен `System.Collection.Concurrent`. Существуют также и другие коллекции для различных сценариев.

Мы также рассмотрели сценарий производителя и потребителя, в котором данные производятся одними и одновременно потребляются несколькими

потоками. Обычно в таких сценариях есть условия гонки, с которыми эффективно помогают справиться параллельные коллекции.

В следующей главе мы узнаем о повышении производительности параллельного кода с помощью шаблонов «отложенной инициализации» (lazy initialization).

Вопросы

1. Что из предложенного не является параллельной коллекцией?
   1. `ConcurrentQueue<T>`
   2. `ConcurrentBag<T>`
   3. `ConcurrentStack<T>`
   4. `ConcurrentList<T>`
2. Как называется тип, при котором один поток только производит данные, а другой – только потребляет их?
   1. Чистый производитель–потребитель
   2. Смешанный производитель–потребитель
3. Очередь будет лучше всего работать при меньшем времени обработки элементов, если речь идет о чистом сценарии «производитель–потребитель».
   1. Правда
   2. Ложь
4. Что из представленного не является методами `ConcurrentStack`?
   1. `Push`
   2. `TryPop`
   3. `TryPopRange`
   4. `TryPush`

Глава 7

Повышение производительности с помощью отложенной инициализации

В предыдущей главе мы обсуждали потокобезопасные параллельные коллекции в C#. Параллельные коллекции помогают повысить производительность параллельного кода, не заставляя разработчиков беспокоиться об издержках на синхронизацию.

В этой главе мы рассмотрим еще несколько понятий, способствующих повышению производительности кода при помощи пользовательских реализаций, а также встроенных конструкций. Ниже представлены темы, которые будут обсуждаться в этой главе:

- введение в понятие отложенной инициализации;
- введение в `System.Lazy<T>`;
- как обрабатывать исключения с помощью отложенного шаблона;
- отложенная инициализация с локальным хранилищем потоков;
- сокращение издержек при помощи отложенной инициализации.

Начнем с представления шаблона отложенной инициализации.

Технические требования

Читателям следует хорошо разбираться в TPL и C#. Исходный код этой главы доступен на GitHub по ссылке: https://github.com/PacktPublishing/-Hands-On-Parallel-Programming-with-C-8-and-.NET-Core-3/tree/master/Chapter07.

Что такое отложенная инициализация?

Отложенная загрузка является распространенным в прикладном программировании шаблоном проектирования, при котором создание объекта от-

кладывается до тех пор, пока он не потребуется в приложении. Правильное использование отложенной загрузки может значительно повысить производительность приложения.

Распространенное применение данного шаблона прослеживается в паттернах «кеш на стороне» (cache aside patterns). Мы используем шаблон «кеш на стороне» для объектов, создание которых затратно либо в плане ресурсов, либо в отношении памяти. Вместо многократного создания объектов мы создаем их один раз и затем кешируем для будущего использования. Данный шаблон становится возможным, если инициализация объекта перемещается из конструктора в отдельный метод или свойства. Объект инициализируется только тогда, когда код первый раз вызывает метод или свойство, и уже после этого объект кешируется для последующих обращений. Посмотрите на указанный ниже пример кода, который инициализирует базовый элемент данных в конструкторе:

```
class _1Eager {
  // Объявляем приватную переменную для хранения данных
  Data _cachedData;
  public _1Eager() {
    // Загружаем данные сразу после создания объекта
    _cachedData = GetDataFromDatabase();
  }
  public Data GetOrCreate() {
    return _cachedData;
  }
  // Создаем фиктивный объект данных каждый раз, когда этот метод
     // вызывается
  private Data GetDataFromDatabase() {
    // Фиктивная задержка
    Thread.Sleep(5000);
    return new Data();
  }
}
```

В предыдущем коде проблема состоит в том, что исходные данные инициализируются сразу же после создания экземпляра класса `_1Eager`, хотя доступ к данным можно получить только вызовом метода `GetOrCreate()`. В некоторых сценариях программа может даже не вызывать этот метод, поэтому память будет использоваться впустую.

Отложенная загрузка может быть полностью реализована с помощью пользовательского кода, как показано в следующем примере:

```
class _2SimpleLazy {
  // Объявляем приватную переменную для хранения данных
  Data _cachedData;
  public _2SimpleLazy() {
    // Логика инициализации удалена из конструктора
    Console.WriteLine("Constructor called");
  }
  public Data GetOrCreate() {
```

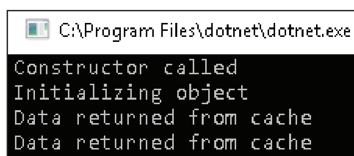
```
    // Если данные равны null, то создаем и храним их для дальнейшего
    // использования
    if (_cachedData == null) {
      Console.WriteLine("Initializing object");
      _cachedData = GetDataFromDatabase();
    }
    Console.WriteLine("Data returned from cache");
    // Возвращаем кешированные данные
    return _cachedData;
  }
  private Data GetDataFromDatabase() {
    // Фиктивная задержка
    Thread.Sleep(5000);
    return new Data();
  }
}
```

В предыдущем примере мы переместили логику инициализации из конструктора в метод `GetOrCreate()`, который проверяет наличие элемента в кеше, перед тем как осуществить его возврат вызывающей программе. В случае отсутствия данных в кеше происходит их инициализация.

Ниже представлен код, вызывающий предыдущий метод:

```
public static void Main() {
  _2SimpleLazy lazy = new _2SimpleLazy();
  var data = lazy.GetOrCreate();
  data = lazy.GetOrCreate();
}
```

Результат кода будет следующим:



Несмотря на то что предыдущий код является отложенным, у него есть потенциальная проблема, связанная с многопоточностью. Это значит, что когда метод `GetOrCreate()` вызывается одновременно несколькими потоками, может произойти многократное обращение к базе данных.

Ситуацию можно исправить путем ввода блокировки, как показано в следующем примере. Вместо шаблона «кеш на стороне» лучше использовать другой шаблон, а именно блокировку с двойной проверкой:

```
class _2ThreadSafeSimpleLazy {
  Data _cachedData;
  static object _locker = new object();
  public Data GetOrCreate() {
```

```
    // Пытаемся загрузить кешированные данные
    var data = _cachedData;
    // Если данные еще не созданы
    if (data == null) {
      // Блокируем общий ресурс
      lock(_locker) {
        // Вторая попытка загрузить данные из кеша, поскольку они
                     // могли быть созданы другим потоком, пока текущий поток
                     // ожидал блокировки
        data = _cachedData;
        // Если данные еще не добавлены в кеш
        if (data == null) {
          // Загружаем данные из базы данных и кешируем их для
                          // дальнейшего использования
          data = GetDataFromDatabase();
          _cachedData = data;
        }
      }
    }
    return _cachedData;
  }
  private Data GetDataFromDatabase() {
    // Фиктивная задержка
    Thread.Sleep(5000);
    return new Data();
  }
  public void ResetCache() {
    _cachedData = null;
  }
}
```

Предыдущий код не нуждается в пояснениях. Очевидно, что создать отложенный шаблон с нуля достаточно сложно. К счастью, .NET Framework предоставляет готовые структуры данных.

Введение в System.Lazy<T>

.NET Framework предоставляет класс `System.Lazy<T>`, который обладает всеми преимуществами отложенной инициализации и учитывает накладные расходы на синхронизацию. Объекты, созданные с помощью `System.Lazy<T>`, инициализируются при первом обращении к ним. С помощью кода отложенного получения данных, описанного в предыдущих разделах, мы помещали часть инициализации из конструктора в метод/свойство для выполнения отложенной инициализации. Благодаря `Lazy<T>` код не нужно изменять.

Существует несколько способов реализации шаблонов отложенной инициализации в C#. К ним относятся следующие:

- логика создания объекта реализуется в конструкторе;
- логика создания объекта передается в качестве делегата в `Lazy<T>`.

В последующих разделах мы постараемся подробнее разобраться в этих сценариях.

Логика создания объекта реализуется в конструкторе

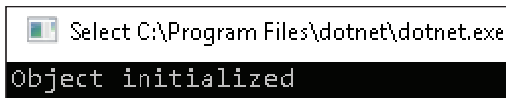
Для начала попытаемся реализовать шаблон отложенной инициализации с помощью классов, инкапсулирующих логику создания объектов в конструкторе. Допустим, есть класс `Data` и класс-обертка `DataWrapper`:

```
class DataWrapper {
  public DataWrapper() {
    CachedData = GetDataFromDatabase();
    Console.WriteLine("Object initialized");
  }
  public Data CachedData { get; set; }
  private Data GetDataFromDatabase() {
    // Фиктивная задержка
    Thread.Sleep(5000);
    return new Data();
  }
}
```

Как видите, инициализация осуществляется внутри конструктора. Если мы используем класс `Data` при помощи следующего кода, то `CachedData` инициализируется в момент создания объекта `DataWrapper`:

```
DataWrapper dataWrapper = new DataWrapper();
```

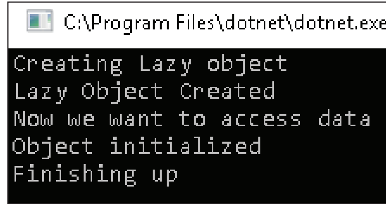
Результат представлен ниже:



Предыдущий код можно преобразовать с помощью `Lazy<T>`:

```
Console.WriteLine("Creating Lazy object");
Lazy<DataWrapper> lazyDataWrapper = new Lazy<DataWrapper>();
Console.WriteLine("Lazy Object Created");
Console.WriteLine("Now we want to access data");
var data = lazyDataWrapper.Value.CachedData;
Console.WriteLine("Finishing up");
```

Как видите, вместо создания объекта напрямую мы обернули его в класс `Lazy<T>`. Конструктор не вызовется до тех пор, пока у нас не будет доступа к свойству `Value` объекта `Lazy`, это видно из следующего вывода:



Логика создания объекта передается в качестве делегата в Lazy<T>

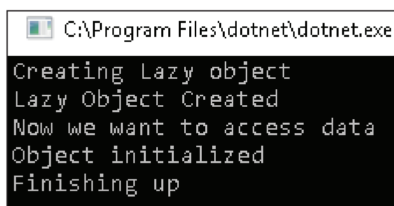
Зачастую объекты не содержат логику создания объектов, поскольку являются простыми моделями данных. Нам нужно извлечь эти данные при первом обращении к ленивым (lazy) объектам, а также передать код для их извлечения. Этого можно добиться при помощи перегрузки `System.Lazy<T>`, как показано ниже:

```
class _5LazyUsingDelegate {
  public Data CachedData { get; set; }
  static Data GetDataFromDatabase() {
    Console.WriteLine("Fetching data");
    // Фиктивная задержка
    Thread.Sleep(5000);
    return new Data();
  }
}
```

В следующем коде мы создаем ленивый объект `Lazy<Data>`, передавая делегат `Func<Data>`:

```
Console.WriteLine("Creating Lazy object");
Func<Data> dataFetchLogic = new Func<Data>(() => GetDataFromDatabase());
Lazy<Data> lazyDataWrapper = new Lazy<Data>(dataFetchLogic);
Console.WriteLine("Lazy Object Created");
Console.WriteLine("Now we want to access data");
var data = lazyDataWrapper.Value;
Console.WriteLine("Finishing up");
```

Как видите, в предыдущем коде мы передали `Func<T>` конструктору `Lazy<T>`. Логика вызывается при первом доступе к свойству `Value` экземпляра `Lazy<T>`, это показано в выводе ниже:



Но нам также нужно научиться не только создавать и использовать ленивые объекты в .NET, но и разобраться с тем, как обрабатывать исключения с помощью шаблонов отложенной инициализации! Перейдем к следующему разделу.

Обработка исключений с помощью шаблона отложенной инициализации

По своей структуре ленивые объекты неизменны. Это означает, что они всегда возвращают тот же экземпляр, с которым были инициализированы. Мы видели, что логику инициализации можно передать в `Lazy<T>`, а также что ее можно помещать в конструктор класса-обертки. Что произойдет, если логика создания/инициализации вызовет исключение? В таком сценарии поведение `Lazy<T>` зависит от значения перечисления `LazyThreadSafetyMode` и выбранного вами конструктора `Lazy<T>`. Существует множество способов обработки исключений при работе с отложенными шаблонами. Некоторые из них заключаются в следующем:

- отсутствие исключений в ходе инициализации;
- случайное исключение при инициализации с кешированием исключений;
- некешируемые исключения.

В последующих разделах мы постараемся подробнее разобрать данные сценарии.

Отсутствие исключений в ходе инициализации

Логика инициализации выполняется однократно, объект кешируется и доступен через свойство `Value`. Мы уже наблюдали такое поведение в предыдущем разделе.

Случайное исключение при инициализации с кешированием исключений

В данном случае логика инициализации выполняется при каждом вызове свойства `Value`, поскольку базовый объект не создан. Такой способ полезен в сценариях, в которых логика создания зависит от внешних факторов, таких как подключение к интернету и вызов внешних сервисов. Если на какое-то мгновение интернет отключится, вызов инициализации даст сбой, при этом последующие вызовы смогут вернуть данные. `Lazy<T>` по умолчанию кеширует исключения для всех реализаций параметризованного конструктора и не кеширует для реализаций конструктора без параметров.

Давайте попробуем понять, что же происходит, когда логика инициализации Lazy<T> вызывает случайное исключение.

1. Для начала создадим Lazy<Data> с логикой инициализации, которая задается функцией GetDataFromDatabase():

```
Func<Data> dataFetchLogic = new Func<Data>(() => GetDataFromDatabase());
Lazy<Data> lazyDataWrapper = new Lazy<Data>(dataFetchLogic);
```

2. Далее обратимся к свойству Lazy<Data>, а именно Value, которое выполнит логику инициализации и выдаст исключение, так как значение счетчика (counter) будет равно 0 (полный код будет ниже):

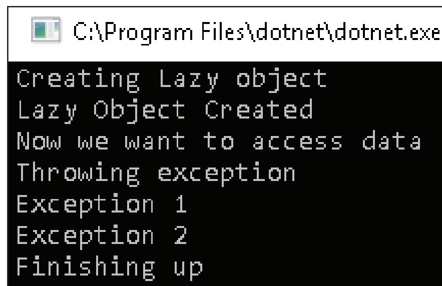
```
try {
  data = lazyDataWrapper.Value;
  Console.WriteLine("Data Fetched on Attempt 1");
} catch (Exception) {
  Console.WriteLine("Exception 1");
}
```

3. Теперь увеличим счетчик на единицу и снова попытаемся получить доступ к свойству Value. Согласно логике, на этот раз должен вернуться объект Data, но видим, что код опять выдает исключение:

```
class _6_1_ExceptionsWithLazyWithCaching {
  static int counter = 0;
  public Data CachedData { get; set; }
  static Data GetDataFromDatabase() {
    if (counter == 0) {
      Console.WriteLine("Throwing exception");
      throw new Exception("Some Error has occurred");
    } else {
      return new Data();
    }
  }
  public static void Main() {
    Console.WriteLine("Creating Lazy object");
    Func<Data>dataFetchLogic = new Func<Data>(() =>
                                       GetDataFromDatabase());
    Lazy <Data> lazyDataWrapper = new
    Lazy <Data> (dataFetchLogic);
    Console.WriteLine("Lazy Object Created");
    Console.WriteLine("Now we want to access data");
    Data data = null;
    try {
      data = lazyDataWrapper.Value;
      Console.WriteLine("Data Fetched on Attempt 1");
    } catch (Exception) {
      Console.WriteLine("Exception 1");
    }
    try {
      counter++;
      data = lazyDataWrapper.Value;
```

```
            Console.WriteLine("Data Fetched on Attempt 1");
          } catch (Exception) {
            Console.WriteLine("Exception 2");
            // обработать исключение;
          }
          Console.WriteLine("Finishing up");
          Console.ReadLine();
        }
      }
```

Как видите, исключение выбрасывается и во второй раз, даже притом, что счетчик был увеличен нами на единицу. Это происходит потому, что значение исключения было кешировано и возвращено при следующем обращении к свойству `Value`. Выходные данные показаны ниже:



Предыдущее поведение аналогично созданию `Lazy<T>` путем передачи `System.Threading.LazyThreadSafetyMode.None` в качестве второго параметра:

```
Lazy<Data> lazyDataWrapper = new
Lazy<Data>(dataFetchLogic,System.Threading.LazyThreadSafetyMode.None);
```

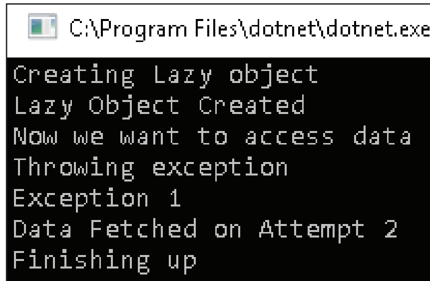
Некешируемые исключения

Изменим инициализацию `Lazy<Data>` из предыдущего кода на представленную ниже:

```
Lazy<Data> lazyDataWrapper = new
Lazy<Data>(dataFetchLogic,
  System.Threading.LazyThreadSafetyMode.PublicationOnly);
```

Это изменение позволяет разным потокам многократно выполнять логику инициализации до тех пор, пока одному из них наконец не удастся это сделать без сообщений об ошибках. Если во время процесса инициализации какой-либо поток выдает ошибку, то все экземпляры базового объекта, созданные завершенными потоками, отменяются, и исключение распространяется на текущее свойство `Value`. В однопоточном сценарии исключение возвращается, когда код повторно обращается к `Value`. Исключения не кешируются.

Получаем следующий результат:



Рассмотрев способы обработки исключений при помощи шаблона отложенной инициализации, теперь перейдем к использованию локального хранилища потоков для отложенной инициализации.

Отложенная инициализация с локальным хранилищем потоков

В многопоточном программировании мы нередко создаем локальную переменную для отдельного потока. Ее создание предполагает, что каждый поток имеет свою собственную копию данных. Это относится ко всем локальным, но не глобальным переменным, так как последние доступны всем потокам. В старых версиях .NET мы использовали атрибут `ThreadStatic`, чтобы добиться от статической переменной поведения, подобного локальной переменной потока. Однако такой способ не является надежным и плохо работает с инициализацией. Если мы инициализируем переменную `ThreadStatic`, то только первый поток получает инициализированное значение, тогда как остальные потоки – стандартное значение переменной (для целых чисел – 0). Продемонстрируем это при помощи следующего кода:

```
[ThreadStatic]
static int counter = 1;
public static void Main() {
  for (int i = 0; i < 10; i++) {
    Task.Factory.StartNew(() => Console.WriteLine(counter));
  }
  Console.ReadLine();
}
```

В предыдущем коде мы инициализировали статическую переменную счетчика (`counter`) со значением `1` и сделали ее `ThreadStatic`, для того чтобы у каждого потока могла быть своя собственная копия. В демонстрационных целях мы создали 10 задач, печатающих значение счетчика. Согласно логике, все потоки должны печатать **1**, но, как видно из вывода ниже, делает это только один поток, остальные же печатают **0**:

```
C:\Program Files\dotnet\dotnet.exe
0
0
0
0
1
0
0
0
0
0
```

.NET Framework 4 предоставляет System.Threading.ThreadLocal<T> в качестве альтернативы ThreadStatic. Его принцип работы очень схож с Lazy<T>. Используя ThreadLocal<T>, создадим локальную переменную потока, которая может быть инициализирована путем передачи функции инициализации, как показано в примере:

```
static ThreadLocal<int> counter = new ThreadLocal<int>(() => 1);
public static void Main() {
  for (int i = 0; i < 10; i++) {
    Task.Factory.StartNew(() => Console.WriteLine($"Thread with id
      {Task.CurrentId} has counter value as {counter.Value}"));
  }
  Console.ReadLine();
}
```

Ожидаемый результат:

```
C:\Program Files\dotnet\dotnet.exe
Thread with id 6 has counter value as 1
Thread with id 8 has counter value as 1
Thread with id 10 has counter value as 1
Thread with id 4 has counter value as 1
Thread with id 1 has counter value as 1
Thread with id 2 has counter value as 1
Thread with id 7 has counter value as 1
Thread with id 3 has counter value as 1
Thread with id 9 has counter value as 1
Thread with id 5 has counter value as 1
```

Различия между Lazy<T> и ThreadLocal<T>:

- каждый поток инициализирует переменную ThreadLocal, используя свои частные данные, тогда как в случае с Lazy<T> логика инициализации выполняется только один раз;
- в отличие от Lazy<T>, свойство Value в ThreadLocal<T> доступно для записи;

- при отсутствии логики инициализации стандартное значение T присваивается переменной ThreadLocal<T>.

Сокращение издержек при помощи отложенной инициализации

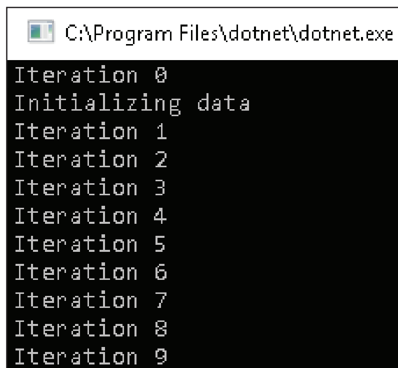
Lazy<T> использует непрямое обращение к объекту с помощью дополнительной обертки (wrapper), что может привести к проблемам с вычислениями и памятью. Чтобы не создавать лишней обертки, мы можем использовать статический вариант класса Lazy<T>, а именно класс LazyInitializer.

Можно использовать LazyInitializer.EnsureInitialized для инициализации данных, которые передаются через ссылку или функцию инициализации, как мы делали ранее с Lazy<T>.

Метод может вызываться несколькими потоками, но как только инициализируется значение, его результат становится доступен для всех потоков. В целях демонстрации я добавил вывод в консоль внутри кода инициализации. Несмотря на то что цикл выполняется 10 раз, в однопоточном исполнении инициализация осуществится лишь раз:

```
static Data _data;
public static void Main() {
  for (int i = 0; i < 10; i++) {
    Console.WriteLine($"Iteration {i}");
    // Выполняем отложенную инициализацию _data
    LazyInitializer.EnsureInitialized(ref _data, () => {
      Console.WriteLine("Initializing data");
      // Возвращаем значение, которое будет присвоено параметру
      return new Data();
    });
  }
  Console.ReadLine();
}
```

Получаем результат:

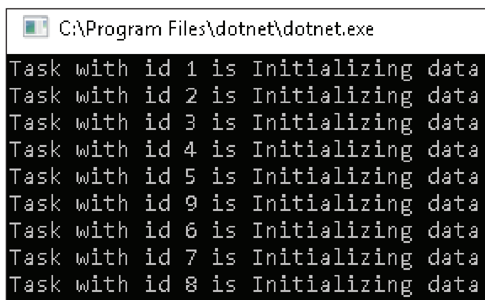


Такая реализация хороша при последовательном исполнении. Теперь попробуем изменить код и запустить его через множество потоков:

```
static Data _data;
static void Initializer() {
  LazyInitializer.EnsureInitialized(ref _data, () => {
    Console.WriteLine($"Task with id {Task.CurrentId} is
                         Initializing data");
    // Возвращаем значение, которое будет присвоено параметру
    return new Data();
  });

  public static void Main() {
    Parallel.For(0, 10, (i) => Initializer());
    Console.ReadLine();
  }
}
```

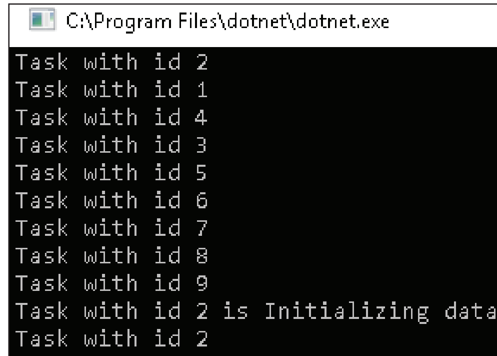
Получаем результат:



Как видно, со множеством потоков возникает состояние гонки, и в конечном итоге все потоки инициализируют данные. Исключить гонку можно, изменив программу, как показано в примере ниже:

```
static Data _data;
static bool _initialized;
static object _locker = new object();
static void Initializer() {
  Console.WriteLine("Task with id {0}", Task.CurrentId);
  LazyInitializer.EnsureInitialized(ref _data, ref _initialized,
    ref _locker, () => {
    Console.WriteLine($"Task with id {Task.CurrentId} is
                             Initializing data");
    // Возвращаем значение, которое будет присвоено параметру
    return new Data();
  });
}
public static void Main() {
  Parallel.For(0, 10, (i) => Initializer());
  Console.ReadLine();
}
```

В предыдущем коде видно, как мы использовали перегрузку метода `EnsureInitialized` и передали булеву переменную и объект блокировки в качестве параметра. Это необходимо для того, чтобы логика инициализации за раз могла выполняться только одним потоком, как показано ниже:



В данном разделе мы рассмотрели способ, при котором можно обойти проблему, связанную с накладными расходами в `Lazy<T>`, используя его другой встроенный статический вариант, так называемый класс `LazyInitializer`.

Резюме

В этой главе мы с вами обсудили различные аспекты отложенной загрузки и структуры данных, предоставляемых .NET Framework для облегчения ее реализации.

Ленивая (или отложенная) загрузка может значительно повысить производительность приложений за счет уменьшения объема памяти, а также экономии вычислительных ресурсов на повторную инициализацию. Мы можем выбрать между созданием отложенной инициализации с нуля при помощи `Lazy<T>` или же, используя статический класс `LazyInitializer`, можем вовсе избежать сложностей. При оптимальном использовании хранилищ потоков и обработке исключений они, безусловно, являются отличными инструментами для разработчиков.

В следующей главе мы начнем обсуждение подходов асинхронного программирования, которые доступны в C#.

Вопросы

1. Отложенная инициализация всегда включает в себя создание объекта в конструкторе.
   1. Да
   2. Нет

2. В шаблоне отложенной инициализации создание объекта откладывается до тех пор, пока в этом не будет необходимости.
   1. Да
   2. Нет
3. Что из перечисленного можно использовать для создания ленивых объектов, которые не кешируют исключения?
   1. `LazyThreadSafetyMode.DoNotCacheException`
   2. `LazyThreadSafetyMode.PublicationOnly`
4. Какой атрибут можно использовать для создания локальной переменной потока?
   1. `ThreadLocal`
   2. `ThreadStatic`
   3. Оба

Часть III

АСИНХРОННОЕ ПРОГРАММИРОВАНИЕ С ИСПОЛЬЗОВАНИЕМ C#

В этой части вы узнаете о еще одном важном аспекте создания производительных программ – с использованием методов асинхронного программирования. В то же время вы сможете увидеть разницу между тем, как это делалось в более ранних версиях, и тем, как это делается при помощи новейших конструкций `async` и `await`.

Данный раздел представлен следующими главами:

- глава 8 «Введение в асинхронное программирование»;
- глава 9 «Основы асинхронного программирования с помощью async, await и задач».

Глава 8

Введение в асинхронное программирование

В предыдущих главах мы уже много раз сталкивались с параллельным программированием. **Параллелизм** – это создание небольших задач, называемых единицами работы (unit of work), которые могут выполняться одновременно одним или несколькими потоками приложения. После того как потоки завершают переданный им код, они уведомляют родительский поток, поскольку их выполнение происходит внутри прикладного процесса.

В этой главе мы сначала рассмотрим различия между синхронным и асинхронным кодами. Затем обсудим случаи, при которых нужно использовать асинхронный код, и ситуации, в которых его использование лучше исключить. Также мы поговорим о том, как эволюционировали асинхронные шаблоны с течением времени. И в конце главы мы выясним, каким образом новые возможности параллельного программирования помогают нам обойти сложности асинхронного кода.

В этой главе мы рассмотрим следующие темы:

- синхронный и асинхронный коды;
- случаи использования асинхронного программирования;
- когда не следует использовать асинхронное программирование;
- проблемы, решаемые асинхронным кодом;
- асинхронные шаблоны в ранних версиях C#.

Технические требования

Для освоения материала главы вам понадобится уверенное знание TPL и C#. Исходный код главы доступен на GitHub по ссылке: https://github.com/PacktPublishing/-Hands-On-Parallel-Programming-with-C-8-and-.NET-Core-3/tree/master/Chapter08.

Типы выполнения программ

В любой отдельно взятый промежуток времени программный поток может быть либо синхронным, либо асинхронным. Синхронный код легко пишется и поддерживается, однако его использование не задействует все возможности современных многоядерных процессов и влечет к задержкам в работе пользовательского интерфейса. Асинхронный код, напротив, в целом повышает производительность и отзывчивость приложения, но, в свою очередь, его сложнее писать, отлаживать и поддерживать.

Позже мы подробнее разберем синхронный и асинхронный способы исполнения программ.

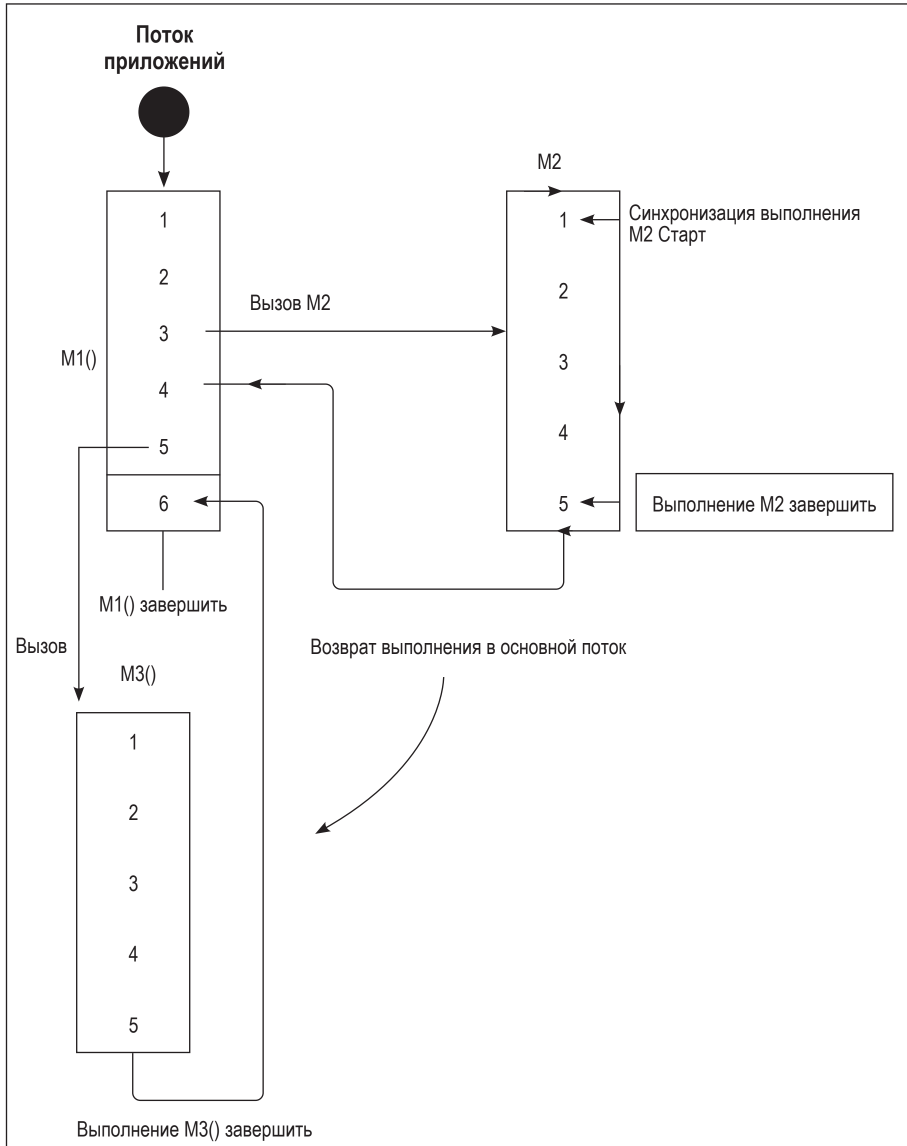
Синхронное выполнение программ

При синхронном исполнении метод никогда не выходит из вызывающего потока. Код выполняется построчно, и при вызове функции вызывающий поток сперва дожидается завершения этой функции, а уже потом переходит к выполнению следующей строки кода. Синхронное программирование – наиболее распространенный подход, который хорошо работает благодаря росту производительности процессоров, наблюдаемой нами за последние десятилетия. В более быстрых процессорах код будет быстрее завершаться.

Мы уже видели, что благодаря параллельному программированию мы создаем несколько одновременно работающих потоков. Можно не только запустить множество потоков, но и обратно сделать основной поток синхронным, обратившись к `Thread.Join` и `Task.Wait`. Рассмотрим пример синхронного кода.

1. Запустим поток приложения, вызвав метод `M1()`.
2. В строке 3 `M1()` синхронно вызывает `M3()`.
3. В момент вызова метода `M2()` управление переходит к методу `M1()`.
4. Как только вызываемый метод (`M2`) завершается, управление возвращается основному потоку, который выполняет остальную часть кода в `M1()`, то есть в строках 4 и 5.
5. То же самое происходит в строке 5 с вызовом `M2()`. Строка 6 начнет выполняться, когда завершится `M2()`.

Ниже представлена схема синхронного выполнения кода:

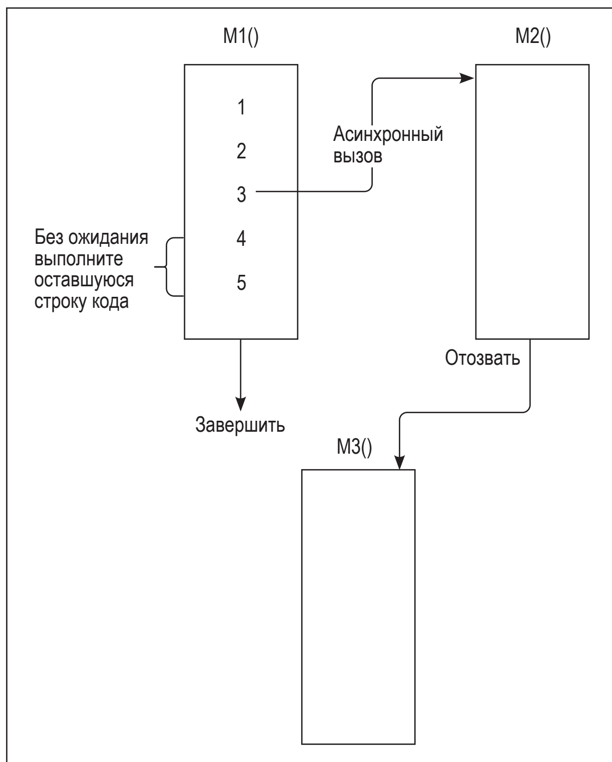


В следующем разделе мы попытаемся подробно разобрать написание асинхронного кода. С помощью полученных знаний вы сможете сравнить эти два программных потока.

Асинхронное выполнение программ

Асинхронная модель позволяет одновременно выполнять несколько задач. При асинхронном вызове метод выполняется в фоновом режиме, в то время как вызываемый поток сразу же возвращается и выполняет следующую строку кода. В зависимости от типа задачи, с которой мы имеем дело, асинхронный метод может и не создать поток. Когда асинхронный метод завершается, он возвращает результат программе через обратный вызов (callback). Если асинхронный метод будет пустым, тогда нет необходимости указывать методы для обратного вызова.

Ниже представлена схема, показывающая, как вызывающий поток выполняет метод `M1()`, который, в свою очередь, вызывает асинхронный метод `M2()`:



В отличие от предыдущего подхода, вызывающий поток не ждет завершения `M2()`. При наличии результата, который должен быть использован из `M2()`, его необходимо поместить в другой метод, например `M3()`. В этом случае происходит следующее.

1. При выполнении `M1()` вызывающий поток выполняет `M2()` в асинхронном режиме.

2. Вызывающий поток предоставляет функцию обратного вызова, скажем `M3()`, при вызове `M2()`.
3. Вызывающий поток не ждет завершения `M2()`, а вместо этого заканчивает остальную часть кода в `M1()`, если это необходимо.
4. `M2()` будет либо сразу выполняться процессором в отдельном потоке, либо позже.
5. Как только `M2()` завершит работу, вызывается `M3()`, который получает выходные данные от `M2()` и обрабатывает их.

Как видите, выполнение синхронной программы проще понять по сравнению с асинхронным кодом. В главе 9 мы узнаем, как облегчить выполнение асинхронного кода при помощи ключевых слов `async` и `await`.

Случаи использования асинхронного программирования

Существует множество ситуаций, в которых **прямой доступ к памяти** (Direct Memory Access, иначе DMA) используется для выполнения операций ввода-вывода (к файлам, базам данных или доступу к сети), где обработка выполняется операционной системой и напрямую процессором, а не кодом приложения. В таких сценариях вызывающий поток выполняет ввод-вывод с помощью специального API и в заблокированном состоянии ожидает завершения этой операции. Когда процессор выполнит задачу, поток разблокируется и выполнит остальную часть метода.

При помощи асинхронных методов можно улучшить производительность и отзывчивость приложений. К тому же выполнение метода возможно в другом потоке.

Написание асинхронного кода

Асинхронное программирование не является чем-то новым для C#. В более ранних версиях C# асинхронный код писался с помощью метода `BeginInvoke` класса `Delegate` и реализаций интерфейса `IAsyncResult`. С появлением TPL асинхронный код начали писать, используя класс `Task`. С момента появления в C# 5.0 предпочтительным вариантом для разработчиков в написании асинхронного кода стали ключевые слова `async` и `await`.

Написать асинхронный код можно при помощи:

- метода `Delegate.BeginInvoke()`;
- класса `Task`;
- интерфейса `IAsyncResult`;
- ключевых слов `async` и `await`.

Далее мы подробно рассмотрим каждый из способов, кроме ключевых слов `async` и `await` – им посвящена целая глава 9!

Использование метода BeginInvoke класса Delegate

В этом подразделе мы рассмотрим использование `Delegate.BeginInvoke` для обратной совместимости с более ранними версиями .NET, так как в .NET Core он уже не поддерживается.

Мы можем использовать `Delegate.BeginInvoke` для асинхронного вызова любого метода. Это делается для повышения производительности пользовательского интерфейса, если часть задач можно переместить в фоновый режим.

Рассмотрим в качестве примера метод `Log`. Синхронный код, представленный ниже, записывает журнал операций. Для наглядности мы удалили код записи и заменили его 5-секундной задержкой, после которой метод `Log` выводит строку в консоль:

Ниже представлен фиктивный метод `Log`, завершение которого занимает 5 секунд:

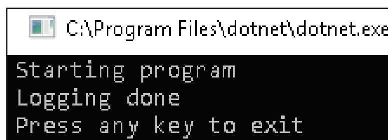
```
private static void Log(string message)
{
  // Симулируем долго выполняющийся метод
  Thread.Sleep(5000);
  // Записываем сообщение в базу данных или сервис
  Console.WriteLine("Logging done");
}
```

Так выглядит вызов `Log` из метода `Main`:

```
static void Main(string[] args)
{
  Console.WriteLine("Starting program");
  Log("this information need to be logged");
  Console.WriteLine("Press any key to exit");
  Console.ReadLine();
}
```

Очевидно, что для записи журнала задержка в 5 секунд – это слишком долго. Поскольку мы не ждем результаты работы от метода `Log` (запись в консоль выполнена в демонстрационных целях), лучше выполнить его асинхронно и сразу вернуть управление вызывающей программе.

Ниже приведен вывод в консоль:



Мы можем добавить вызов `Log` к предыдущему методу, а затем обернуть его с помощью делегата `Action` и вызвать метод `BeginInvoke` у этого делегата:

```
//Log("this information need to be logged");
Action logAction = new Action(
```

```
  () => Log("this information need to be logged"));
logAction.BeginInvoke(null, null);
```

В предыдущих версиях .NET мы увидим асинхронное поведение. Однако в .NET Core код прерывается со следующим сообщением об ошибке:

`System.PlatformNotSupportedException: 'Operation is not supported on this platform.'` (Данная операция не поддерживается платформой.)

В .NET Core перенос синхронных методов в асинхронные делегаты больше не поддерживается по двум основным причинам:

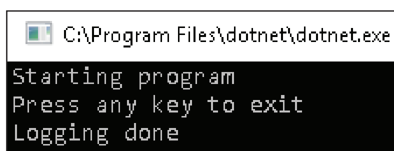
- асинхронные делегаты используют асинхронный шаблон на основе `IAsyncResult`, который не поддерживается библиотеками базовых классов .NET Core;
- асинхронные делегаты невозможны без `System.Runtime.Remoting`, которое также не поддерживается в .NET Core.

Использование класса Task

Другим способом реализации асинхронного программирования в .NET Core является использование класса `System.Threading.Tasks.Task`, как уже упоминалось ранее. Предыдущий код может быть изменен на представленный ниже:

```
// Log("this information need to be logged");
Task.Factory.StartNew(()=> Log("this information need to be logged"));
```

Эта реализация даст нам необходимый вывод без значительных изменений текущего кода:



Ранее мы обсуждали `Task` в главе 2. Класс `Task` предоставляет эффективный способ реализации асинхронных шаблонов на основе задач.

Использование интерфейса IAsyncResult

Интерфейс `IAsyncResult` использовался для реализации асинхронного программирования в старых версиях C#. Ниже приведен пример кода, который хорошо работает в ранних версиях .NET.

1. Во-первых, создадим `AsyncCallback`, который выполнится по завершении асинхронного метода:

   ```
   AsyncCallback callback = new AsyncCallback(MyCallback);
   ```

2. Затем создадим делегат, который выполнит метод `Add` с параметрами. После своего завершения он запустит метод обратного вызова `MyCallback`, обернутый в `AsyncCallback`:

```
SumDelegate d = new SumDelegate(Add);
d.BeginInvoke(100, 200, callback, state);
```

3. Метод `MyCallback` вернет экземпляр `IAsyncResult`. Чтобы получить результат работы асинхронного кода, состояние потока и ссылку на метод обратного вызова, нам нужно привести экземпляр `IAsyncResult` к `AsyncResult`:

```
AsyncResult ar = (AsyncResult)result;
```

4. Как только у нас появится `AsyncResult`, мы можем вызвать `EndInvoke`, чтобы получить результат работы метода `Add`:

```
int i = d.EndInvoke(result);
```

Ниже представлен полный код:

```
using System.Runtime.Remoting.Messaging;
public delegate int SumDelegate(int x, int y);

static void Main(string[] args) {
  AsyncCallback callback = new AsyncCallback(MyCallback);
  int state = 1000;
  SumDelegate d = new SumDelegate(Add);
  d.BeginInvoke(100, 200, callback, state);
  Console.WriteLine("Press any key to exit");
  Console.ReadLine();
}
public static int Add(int a, int b) {
  return a + b;
}
public static void MyCallback(IAsyncResult result) {
  AsyncResult ar = (AsyncResult) result;
  SumDelegate d = (SumDelegate) ar.AsyncDelegate;
  int state = (int) ar.AsyncState;
  int i = d.EndInvoke(result);
  Console.WriteLine(i);
  Console.WriteLine(state);
  Console.ReadLine();
}
```

К сожалению, .NET Core не поддерживает `System.Runtime.Remoting`, и поэтому предыдущий код не будет работать в .NET Core. Мы можем использовать только асинхронные шаблоны на основе задач для сценариев с `IAsyncResult`:

```
FileInfo fi = new FileInfo("test.txt");
byte[] data = new byte[fi.Length];
FileStream fs = new FileStream("test.txt", FileMode.Open,  FileAccess.Read, FileShare.Read,
data.Length, true);
// Мы по-прежнему передаем null в качестве последнего параметра,
// потому что переменная состояния видна для делегата продолжения
Task<int> task = Task<int>.Factory.FromAsync(fs.BeginRead, fs.EndRead,
  data, 0, data.Length, null);
```

```
int result = task.Result;
Console.WriteLine(result);
```

Предыдущий код считывает данные из файла с помощью класса `FileStream`. `FileStream` реализует `IAsyncResult` и поэтому поддерживает методы `BeginRead` и `EndRead`. Также был использован метод `Task.Factory.FromAsync` для работы с `IAsyncResult` и получения данных.

Когда не следует использовать асинхронное программирование

Важность асинхронного программирования не требует подтверждения при создании отзывчивого пользовательского интерфейса и повышении производительности приложений. Однако существуют сценарии, при которых асинхронное программирование может снизить производительность и увеличить сложность кода. В следующих подразделах мы рассмотрим несколько ситуаций, в которых стоит отказаться от использования асинхронного программирования.

В базе данных без пула обработки подключений

От использования асинхронного программирования не будет пользы при наличии одного сервера базы данных с отключенным пулом обработки подключений. При длительных и множественных запросах и будут возникать узкие места в работе системы, независимо от того, как выполняются вызовы: синхронно или асинхронно.

Когда важно, чтобы код легко читался и поддерживался

При использовании интерфейса `IAsyncResult` мы должны разбить исходный метод на два: `BeginMethodName` и `EndMethodName`. Такое изменение логики может потребовать много времени и сил, а также затруднит чтение, отладку и обслуживание кода.

Для простых и быстрых операций

Необходимо учитывать время, которое уходит у кода на синхронную работу. Если затраченное время невелико, тогда лучше сохранить синхронность, поскольку асинхронность требует небольших накладных расходов, что невыгодно для коротких операций.

Для приложений с большим количеством разделяемых данных

Если в приложении используется множество разделяемых ресурсов, таких как глобальные переменные или системные файлы, лучше сохранить синхронность кода; в противном случае результатом может стать снижение производительности. Для доступа к общим ресурсам мы используем примитивы синхронизации, которые могут снизить производительность при работе с несколькими потоками. Иногда однопоточные приложения оказываются более производительными, чем их многопоточные аналоги.

Проблемы, решаемые асинхронным кодом

Рассмотрим несколько ситуаций, в которых асинхронное программирование помогает повысить отзывчивость приложения и улучшить его производительность.

- Ведение журнала и аудит: ведение журнала и аудит используются во всех модулях приложений. Если вдруг вы пишете свой собственный код для ведения журнала и аудита, то в таком случае запросы к серверу замедлятся, так как им нужно еще будет и записаться в журнал. Можно сделать ведение журнала и аудит асинхронными.
- Запросы к внешним сервисам: запросы к веб-серверу и базе данных могут выполняться асинхронно. Как только мы обращаемся к серверу/базе данных, управление переходит к операционной системе, выполняющей обмен данными по сети. На это время вызывающий поток блокируется и ждет события от операционной системы. При завершении сетевой операции вызывающий поток разблокируется и продолжит работу.
- Создание отзывчивых пользовательских интерфейсов: в программах могут быть сценарии, при которых пользователь нажимает кнопку для сохранения данных. Само сохранение может включать в себя несколько небольших операций: получение данных от пользовательского интерфейса, подключение к базе данных (БД) и обновление данных. На работу с БД может уйти много времени, и если эти вызовы выполняются синхронно в потоке пользовательского интерфейса, то он блокируется до тех пор, пока сохранение не завершится. Это значит, что пользователь не сможет работать с приложением, пока не завершится работа с БД. Однако мы можем улучшить пользовательский опыт, используя асинхронные вызовы.
- Приложения, требовательные к ЦП: новые технологии и возможности .NET сегодня позволяют нам писать код с алгоритмами машинного обучения, выполнять операции ETL и добывать криптовалюту. Эти задачи требуют множества вычислений, поэтому лучше сделать такие программы асинхронными.

Асинхронные шаблоны в ранних версиях C#

В ранних версиях .NET поддерживались два шаблона для выполнения операций ввода-вывода и вычислений:

- модель асинхронного программирования (Asynchronous Programming Model, APM);
- асинхронная модель на основе событий (Event-Based Asynchronous Pattern, EAP).

Мы подробно обсудили оба этих подхода в главе 2. Мы также узнали, как можно преобразовывать эти устаревшие реализации в асинхронные шаблоны на основе задач.

Подведем итоги главы.

Резюме

В этой главе мы познакомились с понятием асинхронного программирования и преимуществами асинхронного кода. Также обсудили сценарии, в которых можно реализовать асинхронное программирование и где его лучше не применять. Наконец, мы рассмотрели различные асинхронные шаблоны, реализованные в TPL.

При правильном использовании асинхронное программирование помогает повысить производительность серверных приложений за счет эффективного использования потоков. Также оно способствует улучшению отзывчивости десктопных/мобильных приложений.

В следующей главе мы обсудим примитивы асинхронного программирования, предоставляемые платформой .NET Framework.

Вопросы

1. Какой код проще писать, отлаживать и поддерживать?
   1. Синхронный
   2. Асинхронный
2. В каком сценарии следует использовать асинхронное программирование?
   1. Файловый ввод-вывод
   2. База данных с пулом соединений
   3. Сетевой ввод-вывод
   4. База данных без пула соединений
3. Какой подход можно использовать для написания асинхронного кода?
   1. `Delegate.BeginInvoke`
   2. `Task`
   3. `IAsyncResult`
4. Что из этого нельзя использовать для написания асинхронного кода в .NET Core?
   1. `IAsyncResult`
   2. `Task`

Глава 9

Основы асинхронного программирования с помощью async, await и задач

В предыдущей главе мы познакомились с методами асинхронного программирования и решениями, которые были доступны в C# еще до появления .NET Core. Рассмотрели сценарии, при которых от использования асинхронного программирования есть польза, и случаи, в которых его использование следует исключить.

В этой главе мы углубимся в асинхронное программирование и введем два ключевых слова, которые значительно облегчают написание асинхронного кода. Здесь рассмотрим следующие темы:

- введение в `async` и `await`;
- асинхронные делегаты и лямбда-выражения;
- **асинхронный шаблон на основе задач (Task-Based Asynchronous Pattern, TAP)**;
- обработка исключений в асинхронном коде;
- асинхронность с PLINQ;
- измерение производительности асинхронного кода;
- рекомендации по использованию асинхронного кода.

Начнем с ключевых слов `async` и `await`, которые были впервые представлены в C# 5.0 и приняты в .NET Core.

Технические требования

Читателям следует иметь хорошее представление о **TPL** и C#. Исходный код главы доступен на GitHub по ссылке: https://github.com/PacktPublishing/Hands-On-Parallel-Programming-with-C-8-and-.NET-Core-3/tree/master/Chapter09.

Введение в async и await

`async` и `await` являются популярными ключевыми словами, которые используются разработчиками .NET Core для написания асинхронного кода с помощью новых асинхронных API, предоставляемых .NET Framework. Они помечают код при вызове асинхронных операций. В предыдущей главе мы уже обсуждали проблемы преобразования синхронного метода в асинхронный. Ранее мы также разбивали метод на `BeginMethodName` и `EndMethodName`, вызов которых может быть асинхронным. Такой подход создает неудобства и трудности при написании, отладке и обслуживании кода. Однако с помощью ключевых слов `async` и `await` код будет мало отличаться от синхронной реализации. Вся сложная работа по разбиению метода и исполнению асинхронного кода передается компилятору.

Новые API ввода-вывода в .NET Framework поддерживают асинхронность на основе задач, о которой уже говорилось в предыдущей главе. Теперь попробуем разобрать несколько сценариев, использующих операции ввода-вывода, в которых мы используем преимущества ключевых слов `async` и `await`. Допустим, мы хотим загрузить данные в формате JSON из внешнего онлайн-сервиса. В более поздних версиях C# можно написать синхронный код при помощи класса `WebClient`, доступного в пространстве имен `System.Net`, как показано ниже.

Для начала добавьте ссылку на сборку `System.Net`:

```
WebClient client = new WebClient();
string reply = client.DownloadString("http://www.aspnet.com");
Console.WriteLine(reply);
```

Затем создайте объект класса `WebClient` и вызовите метод `DownloadString`, передав URL-адрес страницы для загрузки. Метод будет работать синхронно, и вызывающий поток заблокируется, пока операция загрузки не завершится. Это может привести к снижению производительности сервера (при использовании в коде на стороне сервера) и отзывчивости приложения (при использовании в коде приложения Windows).

Для повышения производительности и отзывчивости мы можем использовать асинхронную версию метода `DownloadString`, которая появилась намного позже.

Ниже представлен метод, который создает запрос на загрузку http://www.aspnet.com и подписывается на событие `DownloadStringCompleted` без ожидания завершения загрузки:

```
private static void DownloadAsynchronously() {
  WebClient client = new WebClient();
  client.DownloadStringCompleted +=
    new DownloadStringCompletedEventHandler(DownloadComplete);
  client.DownloadStringAsync(new Uri("http://www.aspnet.com"));
}
```

Далее вы видите обработчик событий DownloadComplete, который срабатывает при завершении загрузки:

```
private static void DownloadComplete(object sender,
  DownloadStringCompletedEventArgs e) {
  if (e.Error != null) {
    Console.WriteLine("Some error has occurred.");
    return;
  }
  Console.WriteLine(e.Result);
  Console.ReadLine();
}
```

В предыдущем коде мы использовали **асинхронный шаблон на основе событий** (Event-Based Asynchronous Pattern или EAP). Как видите, мы подписались на событие DownloadCompleted, которое вызывается классом WebClient после завершения загрузки. Затем выполнили асинхронный вызов с помощью DownloadStringAsync, который не блокирует поток. Когда загрузка завершается в фоновом режиме, вызывается метод DownloadComplete. При использовании свойства e.Error мы получаем ошибку, а при свойстве e.Result метода DownloadStringCompletedEventArgs – данные.

Запустив данный код в приложении Windows, мы получим ожидаемый результат, при этом ответ всегда будет приниматься выполняемым в фоновом режиме рабочим потоком, а не основным. Разрабатывая приложения Windows, необходимо помнить о том, что из метода DownloadComplete нельзя обновить элементы пользовательского интерфейса. Все подобные вызовы следует делегировать обратно в основной поток пользовательского интерфейса с использованием таких методов, как Invoke в Windows Forms или Dispatcher в **Windows Presentation Foundation** (WPF). При применении методов Invoke/Dispatcher основной поток не блокируется, в связи с чем приложение в целом быстрее реагирует на действия пользователя.

В приведенных в книге примерах кода мы включили сценарии как для Windows Forms, так и для WPF.

Попробуем запустить предыдущий код в консольном приложении .NET Core из основного потока:

```
public static void Main() {
  DownloadAsynchronously();
}
```

Мы можем изменить метод DownloadComplete, добавив Console.ReadLine:

```
private static void DownloadComplete(object sender,
  DownloadStringCompletedEventArgs e) {
  ...
  ...
  ...
  Console.ReadLine(); //Добавим эту строку
}
```

Программа должна осуществить асинхронную загрузку страницы, отобразить результат и перед завершением дождаться от пользователя нажатия на кнопку **Enter**. При запуске данного кода мы увидим, что ничего не изменилось, поэтому давайте разберемся подробнее.

Как уже говорилось ранее, основной поток разблокируется сразу после вызова метода `DownloadStringAsync`. Предполагается, что, подобно основному потоку, асинхронные методы не будут ожидать, пока завершится обратный вызов (callback). А поскольку основной поток выполнил свою работу, вызвав метод `DownloadStringAsync`, приложение завершается.

Разрабатывая веб-приложения, вы можете столкнуться с той же проблемой, если будете использовать предыдущий код в серверном приложении, например на базе ASP.NET MVC. При асинхронном вызове метода родительский поток IIS, который выполняет запрос, не будет ожидать завершения загрузки. В связи с чем результат становится непредсказуем: в веб-приложении код отобразит на консоль вывод и не учтет наличие оператора `Console.ReadLine`. Допустим, написанная вами программа должна вернуть веб-страницу при запросе. Этого можно добиться с помощью синхронного использования класса `WebClient` с ASP.NET MVC, как показано ниже:

```
public IActionResult Index() {
  WebClient client = new WebClient();
  string content = client.DownloadString(
    new Uri("http://www.aspnet.com"));
  return Content(content, "text/html");
}
```

Данный код блокирует поток, тем самым влияя на производительность сервера и приводя к уязвимости для атак типа **«Отказ в обслуживании»** (Denial-of-Service, DoS), которая возникает при одновременном обращении большого числа пользователей к приложению. Когда у сервера не останется свободных потоков для обработки клиентских запросов, он начнет ставить их в очередь. Когда в очереди также не останется места, сервер начнет выдавать ошибку «503: Служба недоступна» (503: Service Unavailable).

Использовать метод `DownloadStringAsync` мы не можем, поскольку при его вызове поток вернет клиенту ответ раньше, чем завершится метод-обработчик `DownloadComplete`. Поэтому необходим способ, при котором поток сервера будет ожидать завершения, не блокируясь при этом. В этом нам помогают `async` и `await`, помимо прочего, способствующие созданию чистого кода, который проще писать, отлаживать и поддерживать.

Для демонстрации `async` и `await` мы воспользуемся еще одним значимым классом .NET Core, `HttpClient`, который доступен в пространстве имен `System.Net.Http`. Данный класс лучше `WebClient`, так как он полностью поддерживает асинхронные операции на основе задач, существенно повышает производительность и поддерживает HTTP-методы, такие как GET, POST, PUT и DELETE.

Ниже представлен асинхронный вариант предыдущего кода с использованием класса `HttpClient` и ключевыми словами `async` и `await`:

```
public async Task < IActionResult > Index() {
  HttpClient client = new HttpClient();
```

```
  HttpResponseMessage response = await
  client.GetAsync("http://www.aspnet.com");
  string content = await response.Content.ReadAsStringAsync();
  return Content(content, "text/html");
}
```

Для начала нужно изменить сигнатуру метода, чтобы добавить в нее ключевое слово `async`. Компилятору направляется инструкция о том, что метод по возможности будет исполняться асинхронно. Затем мы оборачиваем возвращаемый методом тип данных в `Task<T>`. Этот момент очень важен, поскольку .NET Framework поддерживает асинхронные операции на основе задач, а все асинхронные методы должны возвращать `Task`.

Создадим экземпляр класса `HttpClient` и вызовем метод `GetAsync()`, передав URL сайта, который нужно загрузить. В отличие от шаблона EAP, задействующего обратные вызовы, здесь мы всего лишь указываем ключевое слово `await` вместе с вызовом. Это обеспечивает следующее:

- метод выполняется асинхронно;
- вызывающий поток разблокируется, возвращается в пул и обрабатывает другие клиентские запросы. Это повышает производительность сервера;
- когда загрузка заканчивается, `ThreadPool` получает сигнал прерывания от процессора, который начинает извлекать из пула свободный поток. Может оказаться, что извлекаемый поток уже работал с запросом;
- получив ответ, поток из `ThreadPool` начинает выполнять остальную часть метода.

После завершения загрузки получить загруженные данные можно с помощью другой асинхронной операции – `ReadAsStringAsync()`. В этом разделе мы увидели, что написание асинхронных методов, напоминающих их синхронные аналоги, – это несложный и удобный процесс.

Возвращаемый тип асинхронных методов

В предыдущем примере мы изменили тип возвращаемого метода с `IAsyncResult` на `Task<IAsyncResult>`. Существует три типа возвращаемых значений из асинхронных методов:

- `void`;
- `Task`;
- `Task<T>`.

Асинхронные методы должны возвращать тип `Task`, чтобы их можно было ожидать с помощью оператора `await`. При вызове результаты работы метода вернутся не сразу, так как, скорее всего, будут асинхронно выполнять длительную задачу.

`void` можно использовать с асинхронными методами, если вызывающий поток не будет ожидать их выполнения. Такие методы могут выполнять любые фоновые операции, которые не являются частью ответа, возвращаемого пользователю. Например, можно сделать ведение журнала и аудит асинхронными, выполнив эти операции внутри асинхронных `void`-методов. Вызывающий поток сразу продолжит работу при вызове такого метода, а операции

ведения журнала и аудита выполнятся позже в фоновом режиме. Поэтому из асинхронных методов рекомендуется возвращать не `void`, а именно `Task`.

Асинхронные делегаты и лямбда-выражения

Ключевое слово `async` также используется для создания асинхронных делегатов и лямбда-выражений.

Ниже представлен синхронный делегат, который возвращает квадрат числа:

```
Func<int, int> square = (x) => {return x * x;};
```

Можно сделать предыдущий делегат асинхронным, добавив ключевое слово `async`:

```
Func<int, Task<int>> square = async (x) => {return x * x;};
```

Аналогичным образом преобразуются и лямбда-выражения:

```
Func<int, Task<int>> square = async (x) => x * x;
```

Асинхронные методы работают в цепочке. Преобразовав один метод в асинхронный, остальные методы, вызывающие его, также нужно сделать асинхронными, тем самым образуется большая цепочка асинхронных методов.

Асинхронные шаблоны на основе задач

В главе 2 уже обсуждалось, как обеспечивается TAP при помощи класса `Task`. Существует два способа реализации данного шаблона:

- метод компилятора с ключевым словом `async`;
- ручной метод.

Далее мы рассмотрим эти методы в действии.

Метод компилятора с ключевым словом async

При использовании `async` компилятор выполняет необходимую оптимизацию для асинхронного исполнения метода, внутренне используя TAP. Асинхронный метод должен возвращать `Sysem.Threading.Task` или `System.Threading.Task<T>`. Компилятор асинхронно исполняет метод и возвращает результаты или исключения.

Ручная реализация TAP

Ранее мы уже показывали ручную реализацию TAP в EAP и в **модели асинхронного программирования** (Asynchronous Programming Model, APM). Реализация этого шаблона предоставляет больше контроля над общей реали-

зацией метода. Мы можем создать класс TaskCompletionSource<TResult>, а затем выполнить асинхронную операцию. После завершения асинхронной операции мы можем вернуть результат вызывающему потоку с помощью методов SetResult, SetException или SetCanceled класса TaskCompletionSource<TResult>, как показано в примере:

```
public static Task<int> ReadFromFileTask(this FileStream stream,
  byte[] buffer, int offset, int count, object state) {
  var taskCompletionSource = new TaskCompletionSource<int>();
  stream.BeginRead(buffer, offset, count, ar => {
    try {
      taskCompletionSource.SetResult(stream.EndRead(ar));
    } catch (Exception exc) {
      taskCompletionSource.SetException(exc);
    }
  }, state);
  return taskCompletionSource.Task;
}
```

В этом коде мы создали метод, возвращающий Task<int>, который работает в качестве метода-расширения (extension method) с объектами System.IO.FileStream. Внутри метода мы также создали объект TaskCompletionSource<int>, а затем вызвали асинхронную операцию из класса FileStream для считывания файла в массив байтов. При успешном чтении результаты вернутся вызывающему объекту при помощи метода SetResult; в противном случае мы возвращаем исключения с помощью метода SetException. И в конце метод возвращает исходную задачу с результатом выполнения.

Обработка исключений с помощью асинхронного кода

В синхронном коде все исключения находятся в верхней части стека до тех пор, пока их не обработает блок try-catch либо пока они не приведут к остановке кода. При ожидании асинхронного метода стек вызовов не останется прежним, поскольку поток во время ожидания переходит в пул потоков, а затем возвращается обратно. C# облегчает обработку исключений, изменяя поведение исключений для асинхронных методов. Все асинхронные методы возвращают либо Task, либо void. Рассмотрим эти сценарии на примерах, а также изучим поведение программ.

Метод, возвращающий Task и создающий исключение

Скажем, есть метод void, из которого мы возвращаем Task:

```
private static Task DoSomethingFaulty() {
  Task.Delay(2000);
```

```
  throw new Exception("This is custom exception.");
}
```

Метод выдает исключение после двухсекундной задержки.

Мы попытаемся вызвать этот метод, используя различные методы, чтобы понять, как обрабатываются исключения для асинхронных методов. В этом разделе содержатся следующие сценарии:

- вызов асинхронного метода за пределами блока try-catch без `await`;
- вызов асинхронного метода из блока try-catch без `await`;
- вызов асинхронного метода с `await` за пределами блока try-catch;
- методы, возвращающие `void`.

Подробнее мы рассмотрим эти методы в следующих разделах.

Асинхронный метод вне блока try-catch без await

Ниже приведен пример асинхронного метода, возвращающего `Task`. В свою очередь, метод вызывает `DoSomethingFaulty()`, который выдает исключение.

Наша реализация метода `DoSomethingFaulty()`:

```
private static Task DoSomethingFaulty() {
  Task.Delay(2000);
  throw new Exception("This is custom exception.");
}
```

Ниже представлен код для метода `AsyncReturningTaskExample()`:

```
private async static Task AsyncReturningTaskExample() {
  Task<string> task = DoSomethingFaulty();
  Console.WriteLine("This should not execute");
  try {
    task.ContinueWith((s) => {
      Console.WriteLine(s);
    });
  } catch (Exception ex) {
    Console.WriteLine(ex.Message);
    Console.WriteLine(ex.StackTrace);
  }
}
```

Ниже представлен вызов `AsyncReturningTaskExample()` из метода `Main()`:

```
public static void Main() {
  Console.WriteLine("Main Method Starts");
  var task = AsyncReturningTaskExample();
  Console.WriteLine("In Main Method After calling method");
  Console.ReadLine();
}
```

Async Main появился в C# 7.1, правда, его можно было полноценно использовать только в .NET Core 3.0 и выше.

Как видите, программа вызывает асинхронный метод `AsyncReturning-TaskExample()` без использования `await`.

После чего `AsyncReturningTaskExample()` вызывает метод `DoSomethingFaulty()`, создающий исключение. При запуске этого кода мы получим следующий вывод:

```
C:\Program Files\dotnet\dotnet.exe
Main Method Startes
In Main Method After calling method
```

В синхронном программировании программа бы выдала необработанное исключение и аварийно завершилась. Но в нашем случае программа продолжает свою работу. Это связано с тем, что объекты класса `Task` исполняются фреймворком в фоне. При этом объект task доступен в вызывающей программе в состоянии «завершено с ошибкой» (**faulted**), как показано на скриншоте:

```
10      public static void Main()
11      {
12          // AsyncReturningValueExample();
13          Console.WriteLine("Main Method Startes");
14          var task = AsyncReturningTaskExample();
15          Console.WriteLine("In Main Method After calling method");
16          Console.ReadLine();  ≤ 31ms elapsed
```

Watch 1

Name	Value
task	Id = 2, Status = Faulted, Method = "{null}", Result = "{Not yet computed}"
AsyncState	null
CancellationPending	false
CreationOptions	None
Exception	Count = 1
Id	2
Result	{System.Threading.Tasks.VoidTaskResult}
Status	Faulted
Raw View	

Более подходящим решением станет проверка статуса задачи и ручная обработка исключений:

```
var task = AsyncReturningTaskExample();
if (task.IsFaulted)
  Console.WriteLine(task.Exception.Flatten().Message.ToString());
```

В главе 2 мы уже видели, что эта задача возвращает экземпляр `AggregateExceptions`. Для получения внутренних исключений мы можем использовать метод `Flatten()`, как показано выше.

Вызов асинхронного метода из блока try-catch без await

Изменим метод таким образом, чтобы можно было переместить вызов асинхронного метода `GetSomethingFaulty()` внутри блока try-catch, и вызовем его из метода `Main()`.

Ниже представлен `Main`-метод:

```
public static void Main() {
  Console.WriteLine("Main Method Started");
  var task = Scenario2CallAsyncWithoutAwaitFromInsideTryCatch();
  if (task.IsFaulted)
    Console.WriteLine(task.Exception.Flatten().Message.ToString());
  Console.WriteLine("In Main Method After calling method");
  Console.ReadLine();
}
```

А также метод `Scenario2CallAsyncWithoutAwaitFromInsideTryCatch()`:

```
private async static Task Scenario2CallAsyncWithoutAwaitFromInsideTryCatch() {
  try {
    var task = DoSomethingFaulty();
    Console.WriteLine("This should not execute");
    task.ContinueWith((s) => {
      Console.WriteLine(s);
    });
  } catch (Exception ex) {
    Console.WriteLine(ex.Message);
    Console.WriteLine(ex.StackTrace);
  }
}
```

В коде выше мы видим, что создается исключение, которое принимается блоком catch, после чего программа продолжает работу в обычном режиме.

Обратите внимание на значение объекта `Task` в методе `Main`:

```
29        var task = Scenario2CallAsyncWithoutAwaitFromInsideTryCatch();
30        if (task.IsFaulted)
31            Console.WriteLine(task.Exception.Flatten().Message.ToString());
32
33        Console.WriteLine("In Main Method After calling method");
34        Console.ReadLine();
35    }
36    private async static Task Scenario3CallAsyncWithAwaitFromOutsideTryCatch()
37    {
38        await DoSomethingFaulty();
39        Console.WriteLine("This should not execute");
40    }
```

Watch 1

Name	Value
task	Id = 2, Status = RanToCompletion, Method = "{null}", Result = "System.Threading.Tasks.VoidTaskResult"
AsyncState	null
CancellationPending	false
CreationOptions	None
Exception	null
Id	2
Result	{System.Threading.Tasks.VoidTaskResult}
Status	RanToCompletion
Raw View	

Если задачи не создаются в блоке try-catch, то появляются необработанные исключения. Есть вероятность того, что логика будет работать не так, как планировалось, из-за чего могут возникнуть проблемы. Наилучшим решением станет создание задачи внутри блока try-catch.

Как видите, после обработки исключения выполнение асинхронного метода продолжилось в нормальном режиме. Статус возвращаемой задачи принимает значение `RanToCompletion`.

Вызов асинхронного метода с await за пределами блока try-catch

Ниже показан код метода, вызывающий неисправный метод `DoSomethingFaulty()` и ожидающий его завершения с помощью ключевого слова `await`:

```
private async static Task Scenario3CallAsyncWithAwaitFromOutsideTryCatch() {
  await DoSomethingFaulty();
  Console.WriteLine("This should not execute");
}
```

Вызов из метода `Main`:

```
public static void Main() {
  Console.WriteLine("Main Method Starts");
  var task = Scenario3CallAsyncWithAwaitFromOutsideTryCatch();
  if (task.IsFaulted)
    Console.WriteLine(task.Exception.Flatten().Message.ToString());
  Console.WriteLine("In Main Method After calling method");
  Console.ReadLine();
}
```

Поведение программы в этом случае будет подобно поведению в первом сценарии.

Метод, возвращающий значение void

Если вместо `Task` методы вернут `void`, то программа завершится аварийно. Можно попробовать запустить следующий код.

Ниже представлен метод, возвращающий `void` вместо `Task`:

```
private async static void
Scenario4CallAsyncWithoutAwaitFromOutsideTryCatch() {
  Task task = DoSomethingFaulty();
  Console.WriteLine("This should not execute");
}
```

Ниже вы также увидите вызов из метода `Main`:

```
public static void Main() {
  Console.WriteLine("Main Method Started");
  Scenario4CallAsyncWithoutAwaitFromOutsideTryCatch();
  Console.WriteLine("In Main Method After calling method");
  Console.ReadLine();
}
```

Вывода в консоль не будет, поскольку программа завершится аварийно.

Возвращать void из асинхронных методов все же не стоит, хотя такое и может случиться по ошибке. Наша задача – написать код таким образом, чтобы он не приводил к сбоям в работе, или же сделать так, чтобы код смог корректно завершить свою работу, обработав исключения.

Для обработки фоновых исключений на уровне всего приложения можно реализовать обработчики для двух глобальных событий:

```
AppDomain.CurrentDomain.UnhandledException += (s, e) =>
  Console.WriteLine("Program Crashed", "Unhandled Exception
    Occurred");
TaskScheduler.UnobservedTaskException += (s, e) =>
  Console.WriteLine("Program Crashed", "Unhandled Exception
    Occurred");
```

Предыдущий код перехватывает все необработанные исключения в программе и делает процесс управляемым. Вообще, программа не должна произвольно завершаться, и если подобное все-таки происходит, она должна зафиксировать информацию об этом и очистить все используемые ресурсы.

Асинхронность с PLINQ

Инструмент PLINQ позволяет разработчикам повысить производительность приложений за счет параллельного выполнения множества задач. Однако если задачи используют блокировки, то в таком случае приложение завершится созданием множества блокирующих потоков и в определенный момент перестанет отвечать на запросы. В особенности это касается задач, выполняющих операции ввода-вывода. Ниже представлен метод, который должен как можно быстрее загрузить 100 страниц из интернета:

```
public async static void Main() {
  var urls = Enumerable.Repeat("http://www.dummyurl.com", 100);
  foreach(var url in urls) {
    HttpClient client = new HttpClient();
    HttpResponseMessage response =
      await client.GetAsync("http://www.aspnet.com");
    string content = await response.Content.ReadAsStringAsync();
    Console.WriteLine();
  }
}
```

Как видите, у представленного выше синхронного кода сложность порядка $O(n)$ (количество операций линейно зависит от количества шагов n). Если выполнение одного запроса занимает одну секунду, то в целом метод займет не менее 100 секунд (n = 100). Чтобы ускорить загрузку (при условии хорошей конфигурации сервера), нужно сделать этот метод параллельным.

Это можно сделать при помощи `Parallel.ForEach`, как показано в примере:

```
Parallel.ForEach(urls, url => {
  HttpClient client = new HttpClient();
```

```
  HttpResponseMessage response =
    await client.GetAsync("http://www.aspnet.com");
  string content = await response.Content.ReadAsStringAsync();
});
```

Для данного кода компилятор отобразит ошибку:

> *Оператор 'await' может использоваться только в асинхронном лямбда-выражении. Сделайте лямбда-выражение асинхронным, используя модификатор 'async'. (The 'await' operator can only be used within an async lambda expression. Consider marking this lambda expression with the 'async' modifier.)*

Причина в том, что для корректной работы необходимо асинхронное лямбда-выражение, как в примере:

```
Parallel.ForEach(urls, async url => {
  HttpClient client = new HttpClient();
  HttpResponseMessage response =
    await client.GetAsync("http://www.aspnet.com");
  string content = await response.Content.ReadAsStringAsync();
});
```

Теперь код будет компилироваться, а также корректно и намного эффективнее работать.

О производительности мы поговорим в следующем разделе и углубимся в вопрос оценки производительности асинхронного кода.

Оценка производительности асинхронного кода

С помощью асинхронного кода можно повысить производительность и отзывчивость приложений, однако здесь есть свои особенности. Если в приложениях с графическим интерфейсом, таких как Windows Forms или WPF, на выполнение отдельных методов требуется много времени, то лучше обратиться к асинхронности. Однако в серверных приложениях нужно учитывать дополнительные расходы памяти, используемой заблокированными потоками, и затраты процессора на переключение контекстов.

Рассмотрим следующий код, создающий три задачи, каждая из которых по цепочке асинхронно выполняется. Завершившись, метод приступает к асинхронному выполнению другой задачи. Общее время на выполнение метода можно рассчитать с помощью `Stopwatch`:

```
public static void Main(string[] args) {
  MainAsync(args).GetAwaiter().GetResult();
  Console.ReadLine();
}
public static async Task MainAsync(string[] args) {
  Stopwatch stopwatch = Stopwatch.StartNew();
```

```
  var value1 = await Task1();
  var value2 = await Task2();
  var value3 = await Task3();
  stopwatch.Stop();
  Console.WriteLine($"Total time taken is
    {stopwatch.ElapsedMilliseconds}");
}
public static async Task<int> Task1() {
  await Task.Delay(2000);
  return 100;
}
public static async Task<int> Task2() {
  await Task.Delay(2000);
  return 200;
}
public static async Task<int> Task3() {
  await Task.Delay(2000);
  return 300;
}
```

Вывод кода представлен следующим образом:

Это фактически и есть синхронный код. Преимущество лишь в том, что поток не блокируется, однако общая производительность приложения значительно снизилась, поскольку код теперь является синхронным. Для улучшения производительности код можно изменить:

```
Stopwatch stopwatch = Stopwatch.StartNew();
await Task.WhenAll(Task1(), Task2(), Task3());
stopwatch.Stop();
Console.WriteLine($"Total time taken is
  {stopwatch.ElapsedMilliseconds}");
```

Как видите, это отличное использование параллельности и асинхронности, которое повышает производительность:

Чтобы лучше понять асинхронность, нам также необходимо разобраться с тем, в каких потоках выполняется код. Поскольку новые асинхронные API работают с классом `Task`, то вызовы выполняются потоком из `ThreadPool`. Когда мы выполняем асинхронные вызовы (например, для извлечения данных из

сети), то управление передается потоку ввода-вывода, который управляется ОС. Обычно этот поток существует в единственном экземпляре и является общим для всех сетевых запросов. Когда запрос ввода-вывода завершается, ОС создает сигнал прерывания, который обрабатывается потоком ввода-вывода отдельного приложения. В серверных приложениях, обычно работающих в режиме **многопоточного апартамента** (Multi-Threaded Apartment, MTA), любой поток может запустить асинхронный запрос, и любой другой может его получить.

Однако в приложениях Windows (включая WinForms и WPF), которые работают в режиме **однопоточного апартамента** (Single-Threaded Apartment, STA), важно, чтобы асинхронный вызов возвращался в родительский контекст (обычно это поток пользовательского интерфейса). У каждого потока пользовательского интерфейса в приложении Windows есть контекст синхронизации (`SynchronizationContext`), обеспечивающий выполнение кода правильным потоком. Во избежание проблем с переключением контекста только один поток может изменять значения элементов пользовательского интерфейса. Наиболее значимым методом класса `SynchronizationContext` является `Post`, который делает так, чтобы переданный в него код выполнился в правильном контексте.

При ожидании результатов выполнения задачи текущий `SynchronizationContext` захватывается. Затем ключевое слово `await` использует (в фоне) метод `Post` для возобновления метода в удерживаемом `SynchronizationContext`. Вызов метода `Post` очень накладен, однако для этого внутри фреймворка существует встроенная оптимизация производительности. Метод `Post` не вызывается, если захваченный `SynchronizationContext` совпадает с `SynchronizationContext` у того потока, который возвращает результат своей работы.

Если мы пишем библиотеку классов и нам не важно, в какой `SynchronizationContext` будет возвращаться вызов метода, то мы можем вовсе отключить метод `Post`. Это делается путем вызова метода `ConfigureAwait()`:

```
HttpClient client = new HttpClient();
HttpResponseMessage response = await
client.GetAsync(url).ConfigureAwait(false);
```

Теперь, когда мы уже знакомы с основами асинхронного программирования, перейдем к рекомендациям по использованию асинхронного кода!

Рекомендации по написанию асинхронного кода

Все рекомендации по написанию асинхронного кода можно свести к следующим ключевым пунктам:

- не используйте async void;
- все методы в цепочке вызовов должны быть асинхронными;
- по возможности используйте `ConfigureAwait`.

Подробнее об этом мы поговорим в следующих разделах.

Не используйте async void

Ранее на примерах мы уже видели, как возврат void из асинхронных методов влияет на обработку исключений. Чтобы исключения не оставались необработанными, асинхронные методы должны возвращать Task или Task<T>.

Все методы в цепочке вызовов должны быть асинхронными

Одновременное использование асинхронных и блокирующих методов скажется на производительности. Если мы захотим сделать метод асинхронным, то в таком случае нужно также изменить методы, которые будут в нем вызываться. Игнорирование данного условия может привести к взаимоблокировке, как показано в примере:

```
private async Task DelayAsync() {
  await Task.Delay(2000);
}
public void Deadlock() {
  var task = DelayAsync();
  task.Wait();
}
```

Вызов метода Deadlock()из приложений с графическим интерфейсом или ASP.NET приводит к блокировке, однако в консольном приложении тот же самый код исправно работает. При вызове метода DelayAsync() захватывается текущий SynchronizationContext, и если он равен null, то захватывается TaskScheduler. Когда метод DelayAsync() завершается, то он пытается выполнить оставшуюся часть кода с захваченным контекстом. Проблема в том, что уже есть поток, синхронно ожидающий завершения асинхронного метода. Получается, что оба потока ожидают завершения третьего потока, что и приводит к взаимоблокировке. Данная проблема характерна только для приложений с графическим интерфейсом и ASP.NET, так как они используют SynchronizationContext, который за раз выполняет только один фрагмент кода. Консольные приложения, напротив, используют ThreadPool. При завершении ожидания часть асинхронного метода планируется в поток ThreadPool. Взаимоблокировки в данном случае не возникнет, так как метод завершается в отдельном потоке.

Создав примеры кодов async/await в консольном приложении, не пытайтесь их скопировать и вставить в приложения с графическим интерфейсом или ASP.NET, так как у них разные модели выполнения асинхронного кода.

По возможности используйте ConfigureAwait

Взаимоблокировку можно предотвратить, исключив `SynchronizationContext`:

```
private async Task DelayAsync() {
  await Task.Delay(2000);
}
public void Deadlock() {
  var task = DelayAsync().ConfigureAwait(false);
  task.Wait();
}
```

При использовании `ConfigureAwait(false)` метод находится в режиме ожидания. Когда ожидание завершается, процессор выполняет остальную часть асинхронного метода в контексте пула потоков. Из-за отсутствия контекста блокировки метод спокойно завершается. Он выполняет возвращенную задачу, и взаимоблокировка не возникает.

Вот мы и подошли к концу данной главы. Время подводить итоги!

Выводы

В этой главе мы рассмотрели две важные конструкции, существенно облегчающие написание асинхронного кода. Основную работу выполняет компилятор, при этом код напоминает синхронный аналог. Мы также обсудили, в каких потоках выполняется код, когда мы делаем методы асинхронными, и узнали о снижении производительности при использовании `SynchronizationContext`. И в конце рассмотрели пример отключения `SynchronizationContext` для повышения производительности.

В следующей главе мы познакомимся с инструментами отладки параллельного кода, используя Visual Studio.

Вопросы

1. Какое ключевое слово используется для разблокировки потока в асинхронных методах?
    1. `async`
    2. `await`
    3. `Thread.Sleep`
    4. `Task`
2. Какие из типов возвращаемых данных допустимы в асинхронных методах?
    1. `void`
    2. `Task`

   3. Task<T>
   4. IAsyncResult

3. TaskCompletionSource<T> может использоваться для ручной реализации асинхронного шаблона на основе задач.
   1. Верно
   2. Неверно

4. Могут ли Main-методы быть асинхронными?
   1. Да
   2. Нет

5. Какое свойство класса Task можно использовать для проверки того, вызывается ли исключение асинхронным методом?
   1. IsException
   2. IsFaulted

6. void всегда используют в качестве возвращаемого типа для асинхронных методов.
   1. Верно
   2. Неверно

Часть IV

ОТЛАДКА, ДИАГНОСТИКА И МОДУЛЬНОЕ ТЕСТИРОВАНИЕ АСИНХРОННОГО КОДА

В этой части мы расскажем о методах и инструментах отладки в Visual Studio. Основное внимание будет уделено таким функциям среды разработки, как интерфейсы для работы с параллельными задачами, потоками, параллельными стеками и визуализации параллелизма. Рассмотрим создание модульных тестов для кода на основе TPL и асинхронного программирования, написание заглушек для модульных тестов, а также дадим несколько советов и рекомендаций по избеганию проблем при написании модульных тестов для ORM.

Данная часть представлена следующими главами:

- глава 10 «Отладка задач с Visual Studio»;
- глава 11 «Создание модульных тестов для параллельного и асинхронного кодов».

Глава 10

Отладка задач с Visual Studio

При использовании параллельного программирования обычно повышаются производительность и отзывчивость приложений, однако бывают и исключения. Распространенными проблемами параллельного/асинхронного кода являются производительность и корректность.

Говоря о производительности, мы подразумеваем снижение скорости выполнения, а о корректности – их непредсказуемость (например, из-за «гонки»). Взаимоблокировка также является значимой проблемой при работе с несколькими параллельными задачами. Отладка многопоточного кода – процесс, требующий особых усилий, поскольку в ходе отладки потоки не прекращают переключаться. При работе с приложениями с графическим интерфейсом важно понимать, какой поток выполняет код.

В этой главе мы научимся отлаживать потоки с помощью некоторых инструментов Visual Studio – окна потоков, окна задач и визуализатора параллелизма.

Глава содержит следующие темы:

- отладка с VS 2019;
- отладка потоков;
- использование окон параллельных задач (Parallel Tasks);
- отладка с использованием окон параллельных стеков (Parallel Stacks);
- использование визуализатора параллелизма (Concurrency Visualizer).

Технические требования

Перед началом этой главы ознакомьтесь с потоками, задачами, Visual Studio и параллельным программированием.

Вы можете посмотреть исходный код на GitHub по ссылке: https:// github.com/PacktPublishing/-Hands-On-Parallel-Programming-with-C-8-and-.NET-Core-3/tree/master/Chapter10.

Отладка с VS 2019

Visual Studio предоставляет множество встроенных инструментов для отладки и устранения неполадок. Инструменты, которые мы обсудим в этой главе, представлены ниже:

- окно потока (Thread window);
- окно параллельных стеков (Parallel Stacks window);
- окно контроля параллельных данных (Parallel Watch window);
- адресная панель отладки (Debug Location toolbar);
- визуализатор параллелизма (Concurrency Visualizer, доступно в VS 2017 и выше);
- окно потока графического процессора (GPU thread window).

В следующих разделах мы подробнее рассмотрим эти инструменты.

Отладка потоков

Работая со множеством потоков, мы должны понимать, когда и какой поток выполняется. Благодаря этому мы избежим проблем перекрестных потоков и гонок. Окно `Threads` позволяет нам работать с потоками в процессе отладки. Достигнув точки останова во время отладки кода в среде Visual Studio, окно потока выводит таблицу, содержащую информацию об активных потоках.

Теперь рассмотрим отладку потоков с помощью Visual Studio.

1. Напишите следующий код вVisual Studio:

```
for (int i = 0; i < 10; i++) {
  Task task = new TaskFactory().StartNew(() => {
    Console.WriteLine($"Thread with Id
      {Thread.CurrentThread.ManagedThreadId}");
  });
}
```

2. Создайте точку останова, нажав клавишу **F9** на вызове `Console.Writeline`.
3. Запустите приложение в режиме отладки, нажав клавишу **F5**. Приложение создаст потоки и начнет выполняться. После точки останова на панели инструментов откройте окно **Потоки** из окна **Отладка** (Debug) | **Окна** (Windows) | **Потоки** (Threads):

Threads

Search: | Search Call Stack | Group by: Process ID | Columns

Process ID: 10076 (9 threads)

ID	Managed ID	Category	Name	Location
16652	1	Main Thread	Main Thread	System.Console.dll!System.Console.ReadLine
3784	3	Worker Thread	Worker Thread	Ch10.dll!Ch10.Program.Main.AnonymousMethod__0_0
12692	4	Worker Thread	Worker Thread	Ch10.dll!Ch10.Program.Main.AnonymousMethod__0_0
15824	5	Worker Thread	Worker Thread	Ch10.dll!Ch10.Program.Main.AnonymousMethod__0_0
576	7	Worker Thread	Worker Thread	Ch10.dll!Ch10.Program.Main.AnonymousMethod__0_0
15252	6	Worker Thread	Worker Thread	Ch10.dll!Ch10.Program.Main.AnonymousMethod__0_0
10220	10	Worker Thread	Worker Thread	Ch10.dll!Ch10.Program.Main.AnonymousMethod__0_0
15972	8	Worker Thread	Worker Thread	Ch10.dll!Ch10.Program.Main.AnonymousMethod__0_0
9548	9	Worker Thread	Worker Thread	Ch10.dll!Ch10.Program.Main.AnonymousMethod__0_0

```
10      {
11          for (int i = 0; i < 10; i++)
12          {
13              Task task = new TaskFactory().StartNew(() =>
14              {
15                  Console.WriteLine("Thread with Id ", Thread.CurrentThread.ManagedThreadId);
```

Среда .NET фиксирует большой объем данных по потокам, отображаемый в столбцах. Желтая стрелка указывает на текущий поток, который сейчас выполняется.

Столбцы таблицы включают следующие значения:

- **Флаг** (Flag): мы можем пометить конкретный поток для его отслеживания, нажав на значок флага;
- **ID**: показывает уникальный идентификатор потока;
- **Управляемый идентификатор** (Managed ID): отображает управляемый идентификатор, присвоенный каждому потоку;
- **Категория** (Category): потокам присваивается уникальная категория, при помощи которой можно определить тип потока (основной поток – поток графического интерфейса, или рабочий поток);
- **Имя** (Name): показывает имя каждого потока или отображает <Без имени>;
- **Расположение** (Location): определяет место выполнения потоков. Также используется для отображения завершенного стека вызовов (call stack).

Нажав на значок флажка, мы помечаем потоки для их мониторинга. Чтобы посмотреть помеченные потоки, можно использовать параметр **Показывать только помеченные потоки** (Show Flagged Threads Only) в окне **Threads**:

Threads

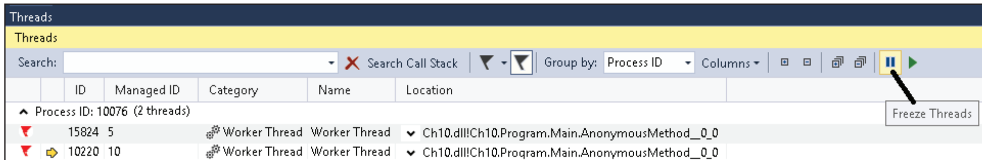
Search: | Search Call Stack | Group by: Process ID | Columns

Show Flagged Threads Only

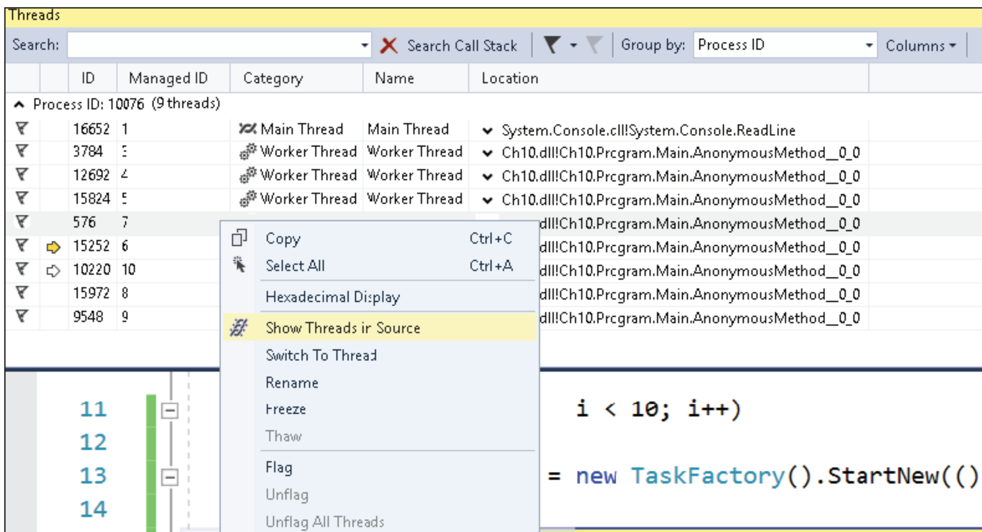
Process ID: 10076 (2 threads)

ID	Managed ID	Category	Name	Location
15824	5	Worker Thread	Worker Thread	Ch10.dll!Ch10.Program.Main.AnonymousMethod__0_0
10220	10	Worker Thread	Worker Thread	Ch10.dll!Ch10.Program.Main.AnonymousMethod__0_0

Еще одной особенностью окна **Threads** является заморозка потоков (freeze threads), которые, на наш взгляд, могут привести к проблемам во время отладки, направленной на отслеживание поведения приложения. Система не выполняет замороженные потоки, даже если для этого есть все необходимое. Во время заморозки поток приостанавливает свою работу:



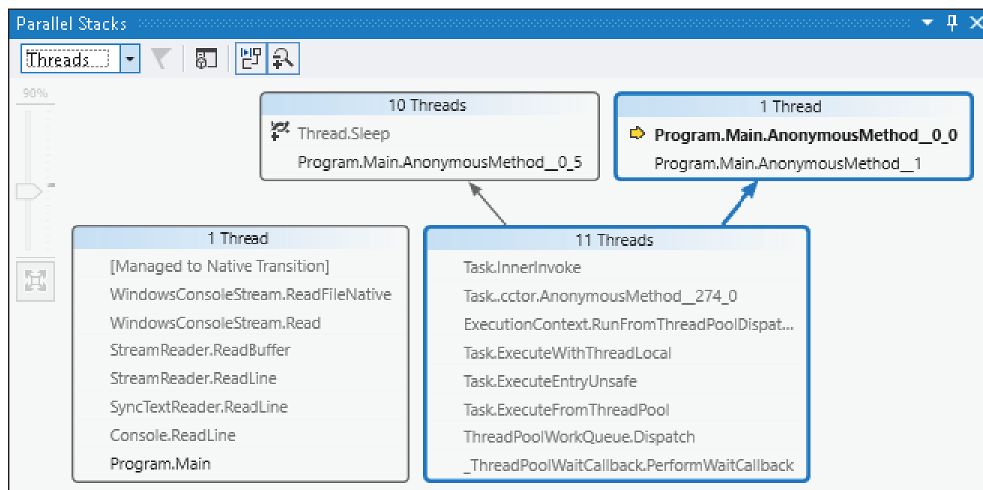
Во время отладки можно переключать выполнение с одного потока на другой нажатием правой кнопкой мыши на поток в окне **Threads** или же двойным нажатием:



Visual Studio поддерживает задачи отладки с использованием параллельных стеков, о которых мы поговорим в следующем разделе.

Использование окон параллельных стеков

Окно **Parallel Stacks** (Параллельные стеки) – хороший инструмент для отладки потоков и задач. Впервые он появился в более поздних версиях Visual Studio. Открыть окно **Parallel Stacks** во время отладки можно, перейдя в раздел **Debug** (Отладка) | **Windows** (Окна) | **Parallel Stacks** (Параллельные стеки):



На скриншоте выше показаны различные представления, между которыми можно переключаться во время работы с окном **параллельных стеков**. Далее мы познакомимся с представлениями и с процессом отладки при помощи параллельных стеков.

Отладка при помощи окон параллельных стеков

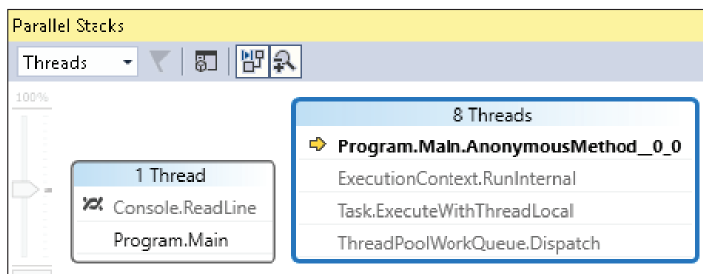
Окна **параллельных стеков** имеют раскрывающееся меню с двумя вариантами. Мы можем переключаться между этими параметрами для получения нескольких представлений в окне **параллельных стеков**. Существуют следующие представления:

- представление **потоков**;
- представление **задач**.

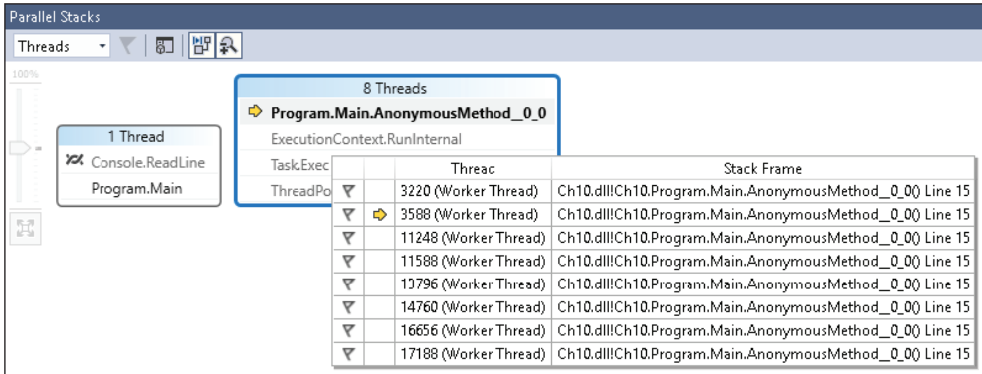
В следующих разделах мы подробнее рассмотрим данные представления.

Представление потоков

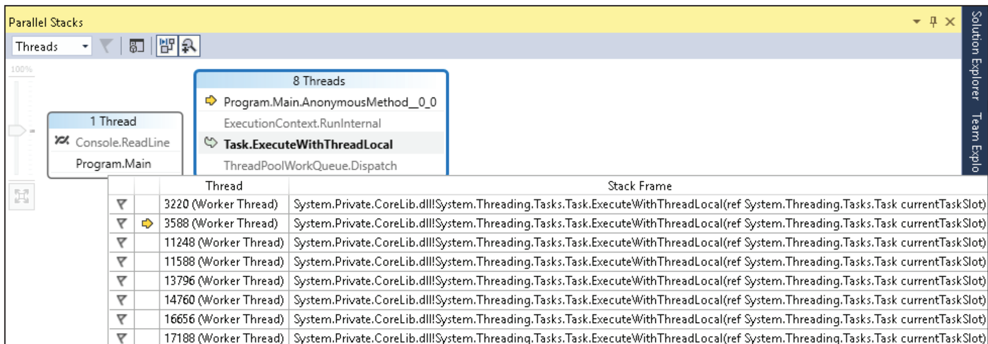
В представлении **потоков** отображаются стеки вызовов для всех потоков, запущенных во время отладки приложения:



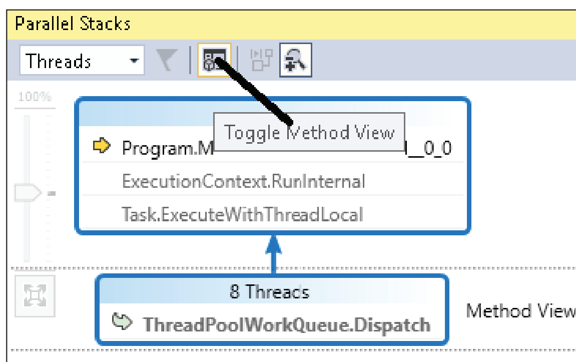
Желтая стрелка указывает на текущее местоположение, в котором выполняется код. При наведении курсора на любой метод из окна **Параллельные стеки** открывается окно **Потоки** с информацией о выполняемом в данный момент потоке:



Двойным щелчком можно переключаться между методами:



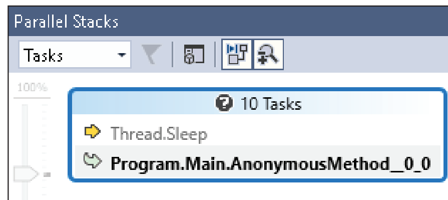
Также можно переключиться на **представление метода** (Method View) для отображения полного стека вызовов:



Представление метода помогает в отладке стека вызовов для отображения переданных методу значений.

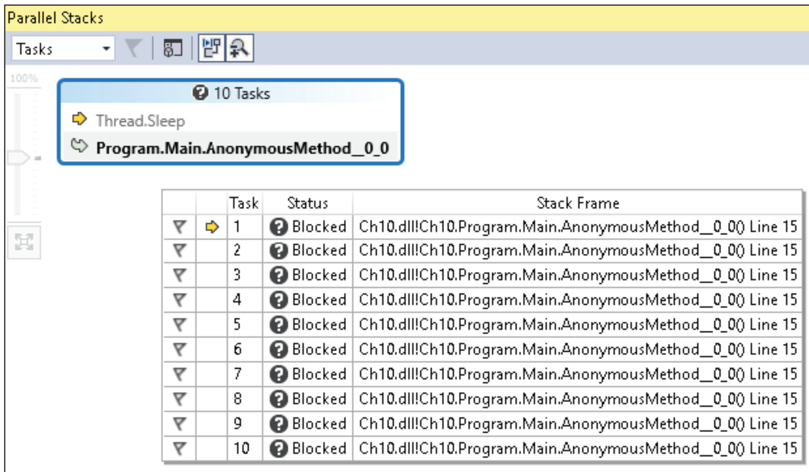
Представление задач

Представление **задач** необходимо при использовании TPL (Task Parallel Library) для создания объектов `System.Threading.Tasks.Task` в коде:



На скриншоте ниже, в строках исполнения, показаны 10 текущих задач.

Состояние всех запущенных задач отображается при наведении указателя мыши на метод:



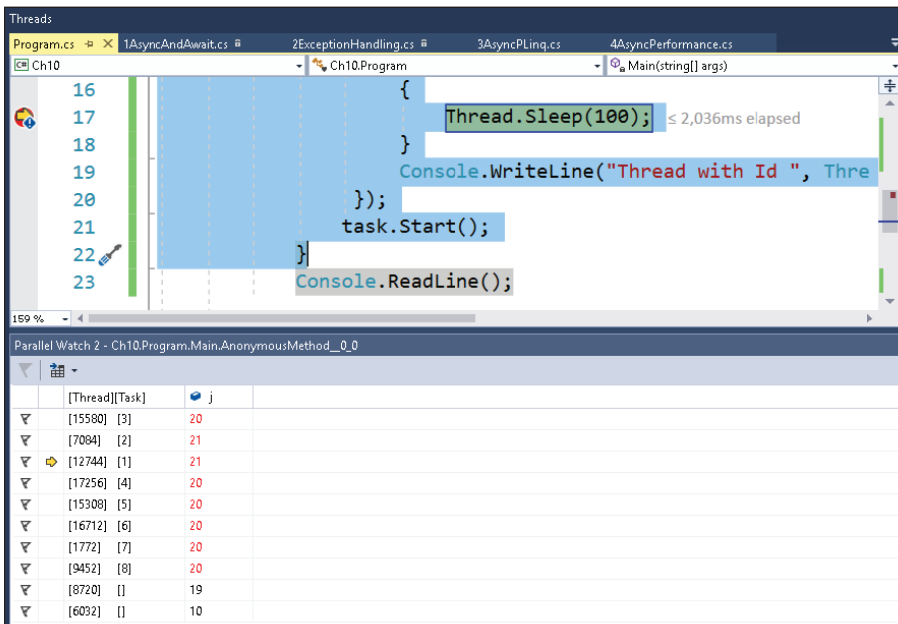
Окно **задач** помогает в анализе проблем с производительностью приложения, возникающих из-за медленных вызовов методов или взаимоблокировки.

Отладка с использованием окна контроля параллельных данных

Можно использовать **окна контроля параллельных данных** (Parallel Watch) для отображения значения переменной в разных потоках. Рассмотрим следующий код:

```
for (int i = 0; i < 10; i++) {
  Task task = new Task(() => {
    for (int j = 0; j < 100; j++) {
      Thread.Sleep(100);
    }
    Console.WriteLine($"Thread with Id
      {Thread.CurrentThread.ManagedThreadId}");
  });
  task.Start();
}
```

Он создает несколько задач, каждая из которых выполняет цикл `for` за 100 итераций. В каждой итерации поток бездействует в течение 100 мс. Мы даем коду поработать какое-то время, затем достигаем точки останова. Используя параллельное окно **Parallel Watch**, мы можем это увидеть на практике. Окно контроля параллельных данных можно открыть следующим образом: **Debug** (Отладка) | **Windows** (Окна) | **Parallel Watch** (Контроль параллельных данных). Можно открыть четыре таких окна, где каждое сможет одновременно отслеживать только одно значение переменной для разных задач:



Как видите, нам нужно отследить значение `j`, поэтому мы указываем его в заголовке третьего столбца и нажимаем **Enter**. В окно добавляется `j`, и мы видим его во всех потоках/задачах.

Использование визуализатора параллелизма

Визуализатор параллелизма (Concurrency Visualizer) – удобное дополнительное расширение в коллекции инструментов Visual Studio. По умолчанию в Visual Studio визуализатора нет, однако его всегда можно скачать из Visual Studio Marketplace по ссылке: https://marketplace.visualstudio.com/.

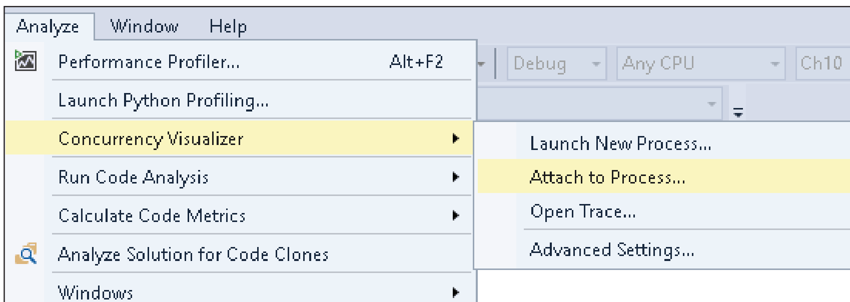
Этот современный инструмент решает сложные проблемы потоков: например, устраняет узкие места производительности системы, противоречия потоков, проверяет загрузку центрального процессора, устраняет перекрестную миграцию потоков и дублирующиеся операции ввода-вывода.

Визуализатор параллелизма поддерживает только проекты Windows/консоли и недоступен для веб-проектов. Ниже мы рассмотрим код в консольном приложении:

```
Action computeAction = () => {
  int i = 0;
  while (true) {
    i = 1 * 1;
  }
};
Task.Run(() => computeAction());
Task.Run(() => computeAction());
Task.Run(() => computeAction());
Task.Run(() => computeAction());
```

Выше мы создали четыре задачи, которые запускают вычислительную задачу, например 1*1, и так до бесконечности. Затем поместили точку останова в цикл `while` и открыли визуализатор параллелизма.

Сейчас мы запустим предыдущий код из Visual Studio и, пока код выполняется, нажмем на **Attach to Process...** (Прикрепить к процессу...), как показано ниже:



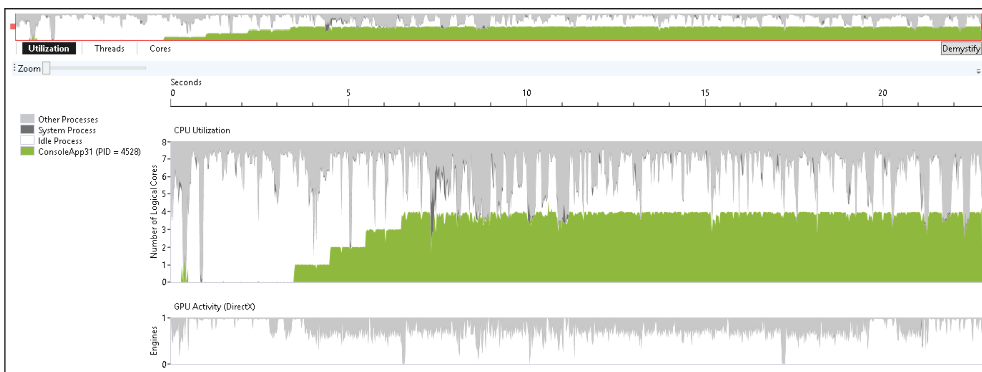
Для начала вам понадобится визуализатор параллелизма нужной версии Visual Studio. Для Visual Studio 2017 его можно скачать по ссылке ниже: https://marketplace.visualstudio.com/items?itemName=Diagnostics.ConcurrencyVisualizer2017#overview.

Включенный визуализатор останавливает профилирование. Нужно дать приложению время на сбор данных для проведения анализа, а затем уже остановить профилировщик.

В арсенале визуализатора параллелизма имеется три вида представлений: представление использования (Utilization), **потоков** (Threads) и **ядер** (Cores). По умолчанию профилировщик открывает представление **использования**, которое мы рассмотрим в следующем разделе.

Представление использования

Представление **использования** отображает активность системы по всем процессорам. На следующем скриншоте представлен момент остановки профилировщика параллелизма.



Зеленым цветом в скриншоте выше выделены четыре ядра, полностью загружающих центральный процессор. Зачастую данное представление используют для подробного отображения состояния параллелизма.

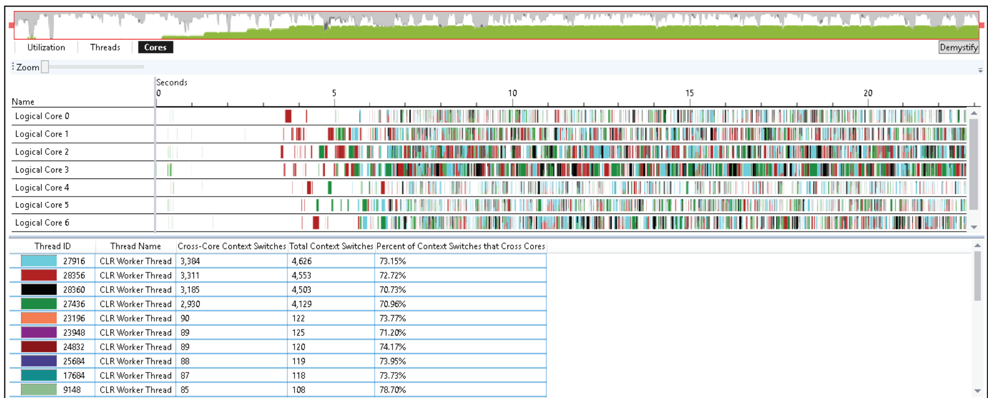
Представление потоков

Благодаря представлению **потоков** можно получить довольно подробный анализ текущего состояния системы, а также определить, выполняются потоки или блокируются, например из-за проблем ввода-вывода или синхронизации:

Это представление используют при выявлении и устранении узких мест производительности в системе. Таким образом, оно позволяет нам определить точное время на фактическое исполнение и на решение проблем синхронизации.

Представление ядер

Представление **ядер** можно использовать для определения количества переключений ядер потоками:



На диаграмме показано, что большую часть времени четыре потока с идентификаторами **12112**, **1604**, **16928** и **4928** выполняют переключение контекста между ядрами.

Рассмотрев последний раздел, посвященный представлениям визуализатора параллелизма, мы подошли к концу этой главы. Время подводить итоги.

Выводы

В этой главе мы поговорили об отладке многопоточных приложений с использованием окон **потоков** для мониторинга больших объемов информации, фиксируемых в .NET. Мы также рассмотрели маркирование потоков, переключение между ними, окна представления **потоков** и **задач** в окнах **параллельных стеков**, работу с несколькими окнами **контроля параллельных данных**, а также отслеживание значения единственной переменной для разных задач в каждый отдельно взятый момент времени. Все это помогает нам лучше понять особенности приложения.

Помимо этого, мы познакомились с таким передовым инструментом, как визуализатор параллелизма, который используют для устранения сложных сценариев многопоточности, свойственных только проектам для Windows и консольным приложениям.

В следующей главе мы научимся создавать модульные тесты для параллельного и асинхронного кодов, а также рассмотрим их недостатки. Кроме того, мы поговорим о проблемах настройки макетных (mock) объектов и применении.

Вопросы

1. Что из перечисленного не является окном для отладки потоков в Visual Studio?
    1. Параллельные потоки
    2. Параллельный стек
    3. Поток графического интерфейса
    4. Параллельное, контрольное значение
2. Пометив флажком определенный поток, его можно отслеживать во время отладки.
    1. Верно
    2. Неверно
3. Что из этого не является представлением в параллельных окнах контрольного значения?
    1. Задачи
    2. Процесс
    3. Потоки
4. Каким способом можно проверить стек вызовов потоков?
    1. Представлением метода
    2. Представлением задач
5. Какое из представлений отсутствует в визуализаторе параллелизма?
    1. Представление потоков
    2. Представление ядер
    3. Представление процесса

Дополнительные материалы для чтения

О параллельном программировании и методах отладки:

- https://www.packtpub.com/application-development/c-multithreaded-and-parallel-programming;
- https://www.packtpub.com/application-development/net-45-parallel-extensions-cookbook.

Глава 11

Создание модульных тестов для параллельного и асинхронного кодов

В этой главе мы расскажем, как пишутся модульные тесты для параллельного и асинхронного кодов. Модульные тесты являются важной частью при создании надежного кода, который можно легко поддерживать при совместной работе в больших командах.

С современными платформами CI/CD проще сделать запуск модульных тестов частью процесса сборки. Это помогает выявить проблемы на ранних этапах. Также оправдывает себя и написание интеграционных тестов, которые помогают оценить корректность совместной работы различных компонентов. Несмотря на многочисленность функций в версиях Visual Studio Community и Professional, только Enterprise позволяет анализировать покрытие кода модульными тестами.

Содержание главы:

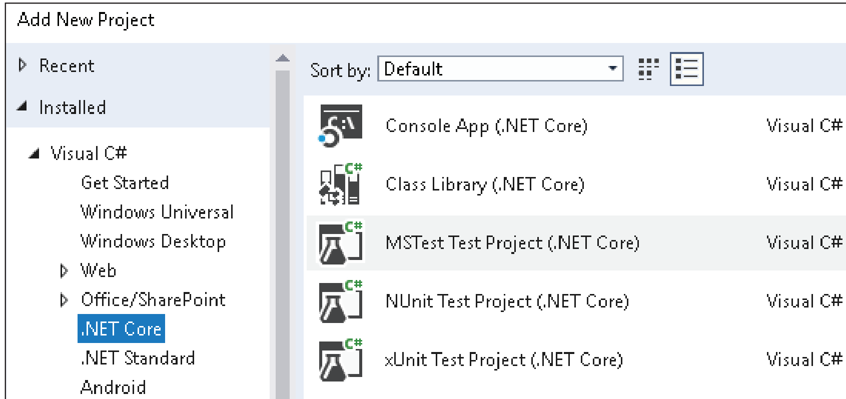
- проблемы при написании модульных тестов для асинхронного кода;
- создание модульных тестов для параллельного и асинхронного кодов;
- имитация обращений к реальным методам и данным с помощью Moq;
- использование инструментов тестирования.

Технические требования

Для написания модульных тестов при помощи фреймворков, поддерживаемых Visual Studio, вам необходимо обладать базовыми знаниями о модульном тестировании и C#. Исходный код данной главы расположен на GitHub по следующей ссылке: https://github.com/PacktPublishing/Hands-On-Parallel-Programming-with-C-8-and-.NET-Core-3/tree/master/Chapter11.

Модульное тестирование с .NET Core

.NET Core обладает тремя фреймворками для создания модульных тестов: **MSTest**, **NUnit** и **xUnit**. Они показаны ниже:



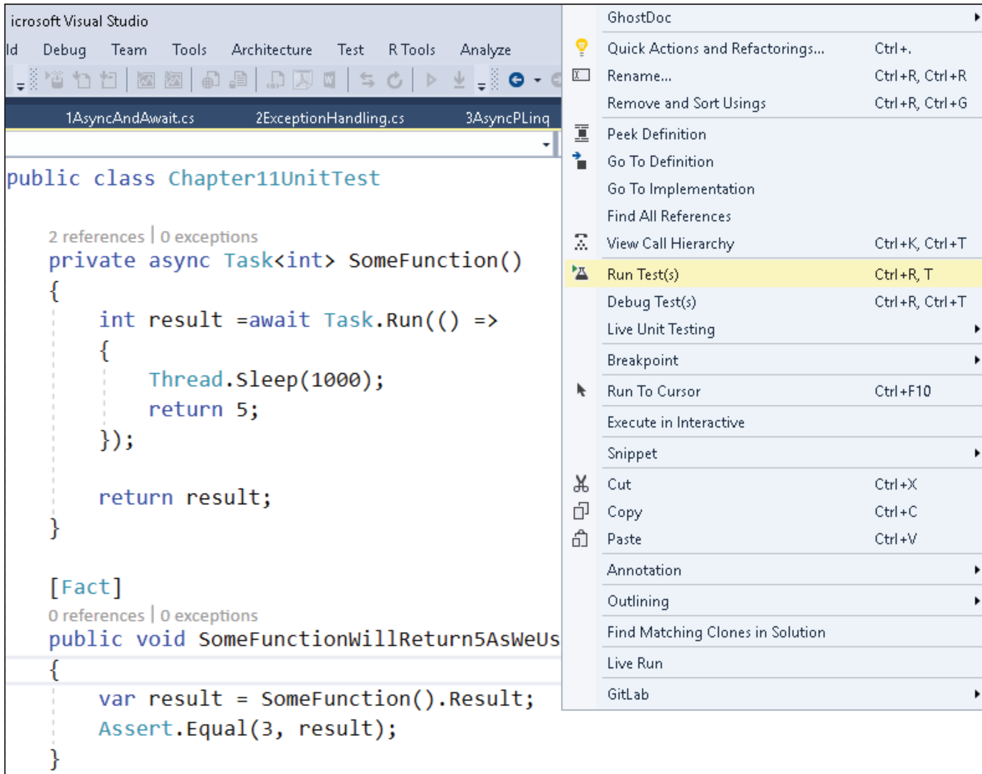
Изначально для создания модульных тестов использовался фреймворк **NUnit**. Затем в Visual Studio добавили **MSTest**, задолго до появления **xUnit** в .NET Core. По сравнению с **NUnit**, версия **xUnit** экономичнее, она помогает пользователям создавать чистые тесты и использовать преимущества новых возможностей .NET Core. Некоторые преимущества **xUnit**:

- легковесность;
- новые возможности;
- улучшенная изоляция тестов;
- xUnit является инструментом, используемым в Microsoft;
- атрибуты `Setup` и `TearDown` были заменены конструктором и `System.IDisposable`, что вынуждает разработчиков писать чистый код.

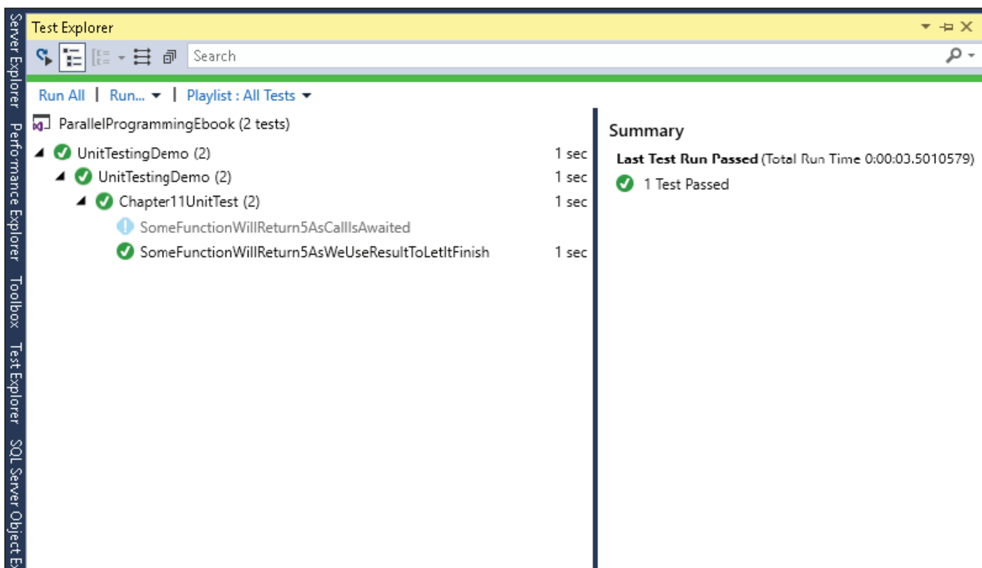
Модульный тест является простой функцией, которая возвращает значение `void`. Она используется для тестирования логики метода и проверки выходных данных по заранее заданным входным данным. Чтобы функция распознавалась как тестовый сценарий, в ней должен присутствовать атрибут `[Fact]`, как показано в примере ниже:

```
[Fact]
public void SomeFunctionWillReturn5AsWeUseResultToLetItFinish() {
  var result = SomeFunction().Result;
  Assert.Equal(5, result);
}
```

Для запуска модульного теста нужно щелкнуть правой кнопкой мыши на метод в коде и выбрать **Выполнить тест(ы)** (Run Test(s)) или **Выполнить отладку теста(ов)** (Debug Test(s)):



Результаты запуска теста отображены в окне **Обозреватель тестов (Test Explorer)**:



И хотя само по себе написание модульных тестов является достаточно простым, тестировние параллельного и асинхронного кодов имеет свои особенности. Более подробно о возникающих сложностях мы поговорим далее.

Проблемы при написании модульных тестов для асинхронного кода

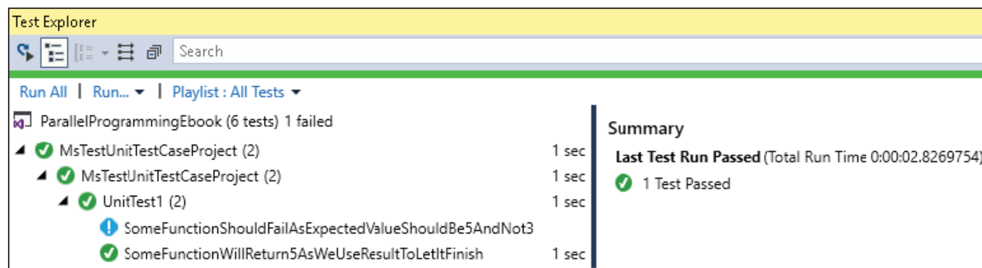
Асинхронные методы возвращают `Task`, ожидание которой необходимо для получения результатов. При отсутствии ожидания метод незамедлительно перейдет к следующему оператору, не дожидаясь завершения асинхронной задачи. Рассмотрим метод для написания модульного теста при помощи xUnit:

```
private async Task<int> SomeFunction() {
  int result = await Task.Run(() => {
    Thread.Sleep(1000);
    return 5;
  });
  return result;
}
```

Метод возвращает постоянное значение 5 с задержкой в 1 секунду. Поскольку метод использовал `Task`, мы применили ключевые слова `async` и `await` для получения результата. Ниже представлен простой тестовый сценарий, который позволяет тестировать данный метод с помощью MSTest:

```
[TestMethod]
public async void SomeFunctionShouldFailAsExpectedValueShouldBe5AndNot3() {
  var result = await SomeFunction();
  Assert.AreEqual(3, result);
}
```

Как видите, метод завершился с ошибкой: ожидаемое значение должно равняться 3, а вместо этого метод возвращает 5. Однако при запуске теста все работает:



Это просходит из-за того, что метод обозначен как асинхронный, и как только он видит ключевое слово `await`, то сразу же прекращает работу. Когда

запускается фоновая задача, ее выполнение откладывается, но поскольку тестовый сценарий завершился без ошибок, тестовая платформа его пропускает. Поэтому есть опасение, что тесты будут проходить, даже в том случае, если фоновая задача будет вызывать исключения.

Предыдущий тестовый случай можно немного изменить, чтобы он работал с MSTest:

```
[TestMethod]
public void SomeFunctionWillReturn5AsWeUseResultToLetItFinish() {
  var result = SomeFunction().Result;
  Assert.AreEqual(3, result);
}
```

Ниже показано, как можно записать тот же самый модульный тест в xUnit:

```
[Fact]
public void SomeFunctionWillReturn5AsWeUseResultToLetItFinish() {
  var result = SomeFunction().Result;
  Assert.Equal(5, result);
}
```

При запуске тестового сценария xUnit он успешно выполняется. Однако проблема кода в том, что это блокирующий вызов фоновой задачи, который может значительно повлиять на скорость выполнения тестов. Лучшим вариантом стал бы следующий:

```
[Fact]
public async void SomeFunctionWillReturn5AsCallIsAwaited() {
  var result = await SomeFunction();
  Assert.Equal(5, result);
}
```

Изначально асинхронные тесты не поддерживались платформами модульного тестирования, например MSTest. Однако сейчас поддерживаются xUnit и NUnit. Тестовый сценарий, описанный выше, также успешно выполняется.

Этот же модульный тест можно записать, используя NUnit:

```
[Test]
public async void SomeFunctionWillReturn5AsCallIsAwaited() {
  var result = await SomeFunction();
  Assert.AreEqual(3, result);
}
```

Этот код отличается от предыдущего. Так, атрибут `[Fact]` заменяется на `[Test]`, в то время как `Assert.Equal` – на `Assert.AreEqual`. Основным отличием кода будет то, что при попытке запуска предыдущего тестового сценария в Visual Studio вы получите ошибку: «Message: Async test method must have non-void return type» (у асинхронного метода должен быть непустой возвращаемый тип). Итак, для NUnit метод нужно изменить, как показано ниже:

```
[Test]
public async Task SomeFunctionWillReturn5AsCallIsAwaited() {
  var result = await SomeFunction();
  Assert.AreEqual(3, result);
}
```

Единственное отличие в том, что void заменяется на Task.

В этом разделе мы рассмотрели возможные проблемы, возникающие при использовании разных фреймворков для модульного тестирования. Теперь перейдем к эффективным способам создания модульных тестов.

Создание модульных тестов для параллельного и асинхронного кодов

В предыдущем разделе мы познакомились со способами создания модульных тестов для асинхронного кода. В этом разделе мы поговорим о создании модульных тестов для сценариев с исключениями. Рассмотрим следующий метод:

```
private async Task<float> GetDivisionAsync(int number, int divisor) {
  if (divisor == 0) {
    throw new DivideByZeroException();
  }
  int result = await Task.Run(() => {
    Thread.Sleep(1000);
    return number / divisor;
  });
  return result;
}
```

Предыдущий метод асинхронно возвращает результат деления двух чисел. Если делитель равен 0, то метод создаст исключение DivideByZero. Для двух сценариев нужны два типа тестов:

- проверка на успешный результат;
- проверка результата исключения при нулевом делителе.

Проверка на успешный результат

Так выглядит тестовый сценарий:

```
[Test]
public async Task GetDivisionAsyncShouldReturnSuccessIfDivisorIsNotZero() {
  int number = 20;
  int divisor = 4;
  var result = await GetDivisionAsync(number, divisor);
  Assert.AreEqual(result, 5);
}
```

Как видите, ожидаемым результатом будет 5. Когда мы запустим тест, он отобразится как успешный в **Обозревателе тестов** (Test Explorer).

Проверка результата исключения при нулевом делителе

Мы можем написать тестовый случай для метода, создающего исключение с помощью метода `Assert.ThrowsAsync<>`:

```
[Test]
public void GetDivisionAsyncShouldCheckForExceptionIfDivisorIsNotZero() {
  int number = 20;
  int divisor = 0;
  Assert.ThrowsAsync<DivideByZeroException>(async () =>
    await GetDivisionAsync(number, divisor));
}
```

Как видите, мы проверили утверждение с помощью `Assert.ThrowsAsync<DivideByZeroException>` при асинхронном вызове метода `GetDivisionAsync`. Поскольку `divisor` (делитель) передается как `0`, метод вызывает исключение, и утверждение выполняется.

Имитация обращений к реальным методам и данным с помощью Moq

Имитация объектов является важным аспектом модульного тестирования. Как вы, возможно, уже знаете, модульное тестирование – это поочередное тестирование модулей; предполагается, что любая внешняя зависимость работает исправно.

Существуют разные имитационные фреймворки, доступные для .NET. Некоторые из них:

- NSubstitute (не поддерживается в .NET core);
- Rhino Mocks (не поддерживается в .NET core);
- Moq (поддерживается в .NET core);
- NMock3 (не поддерживается в .NET core).

Далее мы будем использовать Moq для имитации обслуживаемых компонентов.

В этом разделе мы создадим простую службу с асинхронными методами, а затем напишем модульные тесты для функций, которые ее будут вызывать. Рассмотрим следующий интерфейс службы:

```
public interface IService {
  Task<string> GetDataAsync();
}
```

Мы видим, что в интерфейсе есть метод `GetDataAsync()`, который асинхронно извлекает данные. В следующем фрагменте кода показан класс контроллера, который использует внедрение зависимостей (dependency injection) для получения доступа к экземпляру службы:

```
class Controller {
  public Controller(IService service) {
    Service = service;
  }
  public IService Service { get; }
  public async Task DisplayData() {
    var data = await Service.GetDataAsync();
    Console.WriteLine(data);
  }
}
```

Класс `Controller` также предоставляет асинхронный метод `DisplayData()`, который извлекает данные из службы и выводит их в консоль. При попытке написания модульного теста для предыдущего метода первое, с чем мы столкнемся, – это с невозможностью создания экземпляра службы `IService` без конкретной реализации. Даже при наличии конкретной реализации мы должны избегать вызова реального метода службы, поскольку он больше подходит для интеграционного, а не модульного тестового сценария. Тогда мы обращаемся к нашему помощнику – имитации.

Давайте напишем модульный тест для предыдущего метода с использованием Moq.

1. Установите `Moq` в виде пакета NuGet.
2. Добавьте ссылку на него, как показано в примере:

   ```
   using Moq;
   ```

3. Создайте имитацию объекта:

   ```
   var serviceMock = new Mock<IService>();
   ```

4. Настройте имитацию объекта, который возвращает фиктивные данные. Это можно сделать с помощью метода `Task.FromResult`:

   ```
   serviceMock.Setup(s => s.GetDataAsync())
     .Returns(Task.FromResult("Some Dummy Value"));
   ```

5. Далее нужно создать объект контроллера, передав только что созданную имитацию объекта:

   ```
   var controller = new Controller(serviceMock.Object);
   ```

Ниже приведен простой пример тестового сценария для метода `Display-Data()`:

```
[Test]
public async System.Threading.Tasks.Task DisplayDataTestAsync() {
  var serviceMock = new Mock<IService>();
  serviceMock.Setup(s => s.GetDataAsync())
    .Returns(Task.FromResult("Some Dummy Value"));
  var controller = new Controller(serviceMock.Object);
  await controller.DisplayData();
}
```

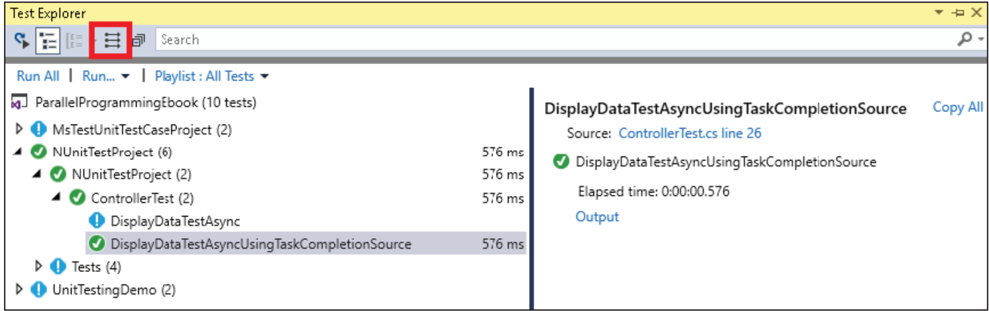
Таким образом можно настроить данные для имитируемых объектов. Еще одним способом настройки является использование класса `TaskCompletionSource`:

```
[Test]
public async Task DisplayDataTestAsyncUsingTaskCompletionSource() {
  // Создаем имитацию сервиса
  var serviceMock = new Mock<IService>();
  string data = "Some Dummy Value";
  // Создаем объект TaskCompletionSource
  var tcs = new TaskCompletionSource<string>();
  // Возвращаем тестовые данные
  tcs.SetResult(data);
  // Настраиваем объект имитации сервиса на возврат Task из tcs,
  // когда метод GetDataAsync сервиса будет вызван
  serviceMock.Setup(s => s.GetDataAsync()).Returns(tcs.Task);
  // Передаем экземпляр имитации службы контроллеру
  var controller = new Controller(serviceMock.Object);
  // Вызываем асинхронно метод DisplayData контроллера
  await controller.DisplayData();
}
```

В корпоративном проекте количество тестов может увеличиваться, поэтому очень важно, чтобы все они корректно выполнялись. В следующем разделе мы поговорим о некоторых общих инструментах тестирования Visual Studio, которые помогают при написании и отладке тестов.

Инструменты тестирования

Обозреватель тестов (Test Explorer) является одним из основных инструментов Visual Studio для запуска тестов и просмотра результатов их выполнения. О нем мы уже говорили в начале этой главы. Отличительной особенностью **Обозревателя** является параллельный запуск тестовых сценариев. Для системы с несколькими ядрами можно использовать параллелизм, который ускорит выполнение тестов. Это делается нажатием кнопки со значком параллельных прямых на панели инструментов и кнопки **Запустить тесты** (Run Tests) в **Обозревателе тестов**:



Для некоторых версий Visual Studio корпорация Microsoft предоставляет дополнительные возможности. Одним из эффективных средств является **Intellitest** с возможностью автоматического создания модульных тестов. Он анализирует исходный код и автоматически создает тестовые сценарии, тестовые данные и наборы тестов. На момент написания книги Intellitest не поддерживался для .NET Core, но поддерживался для других версий .NET Framework.

Выводы

В этой главе мы научились создавать модульные тесты для асинхронных методов. Они способствуют созданию надежного кода, подходят для работы в больших командах и адаптируются к новым платформам CI/CD, благодаря чему ошибки выявляются на ранних этапах. Мы начали с нескольких проблем, которые могут возникнуть при написании модульных тестов для параллельного и асинхронного кодов, и рассмотрели способы минимизации их влияния при помощи правильных подходов к написанию кода. Затем перешли к весьма важному аспекту модульного тестирования – имитации.

Мы также узнали, что Moq поддерживается на .NET Core, которая стремительно развивается; вскоре все главные имитационные платформы будут ею поддерживаться. Далее мы показали вам поэтапное создание тестовых сценариев, вместе с установкой Moq в виде пакета NuGet и настройкой данных для имитируемых объектов.

И наконец, мы рассмотрели функциональные возможности **Обозревателя тестов**, который можно использовать для выполнения тестовых сценариев. А также мы познакомились со способами распараллеливания модульных тестов для увеличения скорости их выполнения.

В следующей главе мы поговорим о концепциях и ролях IIS и Kestrel в среде разработки веб-приложений на базе .NET Core.

Вопросы

1. Что из представленного не поддерживается платформой модульного тестирования в Visual Studio?
   1. JUnit
   2. NUnit
   3. xUnit
   4. MSTest
2. С помощью чего можно проверить результаты модульного теста?
   1. С помощью окна **Обозревателя задач**
   2. С помощью окна **Обозревателя тестов**
3. Какие атрибуты можно применить к методу тестирования, если средой тестирования является xUnit?
   1. Fact
   2. TestMethod
   3. Test
4. Как можно проверить успешность тестового сценария, который вызывает исключение?
   1. `Assert.AreEqual(ex, typeof(Exception)`
   2. `Assert.IsException`
   3. `Assert.ThrowAsync<T>`
5. Какая из имитационных платформ поддерживается в .NET Core?
   1. NSubstitute
   2. Moq
   3. Rhino Mocks
   4. NMock

Дополнительные материалы для чтения

О параллельном программировании и методах модульного тестирования:

- https://www.packtpub.com/application-development/c-multithreaded-and-parallel-programming;
- https://www.packtpub.com/application-development/net-45-parallel-extensions-cookbook.

Часть V

ДОПОЛНИТЕЛЬНЫЕ СРЕДСТВА ПОДДЕРЖКИ ПАРАЛЛЕЛЬНОГО ПРОГРАММИРОВАНИЯ В .NET CORE

В данной части вы познакомитесь с новыми возможностями .NET Core, поддерживающими параллельное программирование.

Содержание части:

- глава 12 «IIS и Kestrel в ASP.NET Core»;
- глава 13 «Шаблоны параллельного программирования»;
- глава 14 «Управление распределенной памятью».

Глава 12

IIS и Kestrel в ASP.NET Core

В предыдущей главе мы обсуждали случаи написания модульных тестов для параллельного и асинхронного кодов, а также некоторые платформы модульного тестирования, доступные в Visual Studio: MSUnit, NUnit и xUnit.

В этой главе мы расскажем о том, как работает модель многопоточности с Internet Information Services (IIS) и Kestrel. Также мы рассмотрим различные параметры настройки, применение которых позволяет максимально использовать ресурсы на сервере. Вы сможете познакомиться с моделью работы Kestrel и с возможностями использования преимуществ параллельного программирования при создании микросервисов.

В данной главе мы рассмотрим следующие темы:

- многопоточность в IIS и внутренние компоненты;
- многопоточность в Kestrel и внутренние компоненты;
- лучшие практики использования многопоточности в микросервисах;
- асинхронность в ASP.NET MVC Core;
- асинхронные потоки данных (новое в .NET Core 3.0).

Приступим.

Технические требования

Вам нужно иметь хорошее представление о работе серверов. Прежде чем вы приступите к данной главе, ознакомьтесь с моделями многопоточности. Исходный код главы доступен на GitHub по ссылке: https://github.com/PacktPublishing/-Hands-On-Parallel-Programming-with-C-8-and-.NET-Core-3/tree/master/Chapter12.

Многопоточность в IIS и внутренние компоненты

Как следует из названия, IIS – это служба Windows для выполнения веб-приложений и обработки запросов к ним через интернет по таким протоколам, как HTTP, TCP, веб-сокеты (web sockets) и др.

В этом разделе мы обсудим **многопоточность в IIS**. В основе IIS лежит **пул потоков CLR** (Common Language Runtime, общеязыковая исполняющая среда). Для того чтобы понять, как ISS обслуживает пользовательские запросы, нужно сначала разобраться в том, как CLR добавляет и удаляет потоки в пул.

Каждому приложению в IIS присваивается уникальный рабочий процесс, который имеет два пула потоков: **пул рабочих потоков** и пул потоков **IOCP** (**I/O completion port**, порт завершения ввода-вывода):

- при создании нового потока с устаревшим `ThreadPool.QueueUserWorkItem` или **TPL** среда выполнения ASP.NET использует рабочие потоки для обработки;
- при выполнении операций ввода-вывода, то есть запросов к базе данных, чтения/записи файлов или сетевых вызовов другой веб-службы, среда выполнения ASP.NET использует потоки IOCP.

На каждый процессор по умолчанию приходится по одному рабочему потоку и одному потоку IOCP. Таким образом, у двухъядерного процессора будет два рабочих и два IOCP-потока. В зависимости от нагрузки `ThreadPool` продолжает добавлять и удалять потоки. IIS назначает поток каждому полученному запросу, из-за чего у каждого из них появляется свой собственный контекст. Поток обслуживает запросы, а также создает и отправляет ответ клиенту.

Если количество доступных потоков в пуле меньше, чем количество полученных сервером запросов, запросы начнут помещаться в очередь. Позже пул создает потоки при помощи одного из двух алгоритмов, известных как *Поиск восхождения к вершине* (Hill Climbing) и *Предотвращение нехватки ресурсов* (Starvation Avoidance). Создание потоков – процесс небыстрый. Он обычно занимает до 500 миллисекунд с момента, когда `ThreadPool` узнает о нехватке потоков. Давайте попробуем разобрать эти алгоритмы.

Предотвращение нехватки ресурсов

В данном алгоритме `ThreadPool` продолжает отслеживать очередь, и если она не двигается, то он продолжает загружать в нее новые потоки.

Поиск восхождения к вершине

В этом алгоритме `ThreadPool` пытается максимизировать пропускную способность, используя минимальное количество потоков. Запуск IIS со стандартными настройками сильно скажется на производительности, поскольку по

умолчанию на процессор приходится только один рабочий поток. Можно увеличить данный параметр, изменив элемент конфигурации в файле `machine.config`:

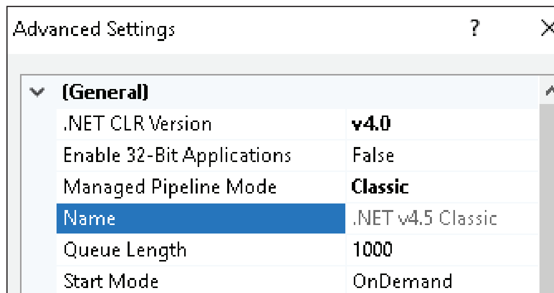
```
<configuration>
  <system.web>
    <processModel minWorkerThreads="25" minIoThreads="25" />
  </system.web>
</configuration>
```

Как видите, минимальное количество рабочих потоков и потоков IOCP было увеличено до 25. По мере поступления новых запросов будут создаваться дополнительные потоки. Здесь важно отметить, что поскольку каждому запросу присваивается свой уникальный поток, код должен выполняться без блокировок. В блокирующем коде свободные потоки отсутствуют. Когда пул потоков себя исчерпает, запросы начнут помещаться в очереди. IIS может ставить в очередь только до 1000 запросов на один пул приложения. Мы можем изменить это значение с помощью параметра `requestQueueLimit` в файле `machine.config`.

Чтобы изменить настройки для всех пулов приложений, нам нужно добавить элемент `applicationPool` с необходимыми значениями:

```
<system.web>
  <applicationPool
   maxConcurrentRequestPerCPU="5000"
   maxConcurrentThreadsPerCPU="0"
   requestQueueLimit="5000" />
</system.web>
```

Для изменения параметров одного пула приложений нужно перейти к **Расширенным настройкам** (Advanced Settings) пула приложений в IIS. Мы можем изменить свойство **Длина очереди** (Queue Length) для настройки количества запросов в очереди для каждого пула приложений (см. скриншот ниже):



Чтобы уменьшить проблемы с гонкой и не получить очереди на сервере, нужно использовать ключевые слова `async`/`await` для любого блокирующего кода ввода-вывода. Такая методика уменьшит проблемы гонки на сервере,

так как потоки не будут блокироваться и возвращаться в пул потоков для обслуживания других запросов.

Многопоточность в Kestrel и внутренние компоненты

Раньше наиболее используемым сервером для размещения приложений .NET был IIS, который привязывался к операционной системе Windows. С появлением других облачных провайдеров и существенным снижением стоимости облачных хостингов, отличных от Windows, возникла необходимость в кросс-платформенной веб-службе. Microsoft представила Kestrel в качестве кросс-платформенной среды для запуска приложений ASP.NET Core. У Kestrel открытый исходный код, и он использует асинхронный сервер ввода-вывода на основе событий. Так как он является лишь средой выполнения приложений ASP.NET Core, а не полнофункциональным веб-сервером, то его рекомендуется использовать совместно с IIS или Nginx.

Первоначально Kestrel основывался на библиотеке `libuv`, которая также является проектом с открытым исходным кодом. Использование `libuv` в .NET началось с ASP.NET 5. `libuv` специально создавалась для асинхронных операций ввода-вывода, в ее основе лежит однопоточная модель циклической обработки событий. Библиотека также поддерживает кросс-платформенные асинхронные сокеты в Windows, macOS и Linux. С ходом развития библиотеки вы можете ознакомиться на GitHub по ссылке https://github.com/libuv/libuv.

В Kestrel `libuv` использовалась только для поддержки асинхронного ввода-вывода. Кроме операций ввода-вывода, всю остальную работу в Kestrel по-прежнему выполняют рабочие потоки .NET с помощью управляемого кода. В первую очередь Kestrel создавалась с целью повышения производительности серверов. Стек очень устойчив к ошибкам и расширяем. `libuv` используется только в качестве транспортного уровня в Kestrel, и благодаря хорошей абстракции его можно заменить другими сетевыми реализациями. Kestrel также поддерживает запуск нескольких циклов событий, создавая таким образом преимущество перед Node.js. Количество используемых циклов обработки событий зависит от количества логических процессоров компьютера, так как на один поток приходится по одному циклу событий. Мы можем настроить это значение в коде приложения. Ниже представлен кусочек файла `Program.cs`, который есть во всех проектах ASP.NET Core:

```
public class Program {
  public static void Main(string[] args) {
    CreateWebHostBuilder(args).Build().Run();
  }
  public static IWebHostBuilder CreateWebHostBuilder(string[] args) => WebHost.
CreateDefaultBuilder(args).UseStartup<Startup>();
}
```

Вы вскоре увидите, что сервер Kestrel основан на шаблоне «строитель» (builder), и можно расширить его функциональность с помощью внешних пакетов и методов. В следующих разделах мы узнаем, как можно менять настройки Kestrel для различных версий .NET Core.

ASP.NET Core 1.x

Мы можем использовать метод расширения `UseLibuv` для установки количества потоков. Это можно сделать при помощи свойства `ThreadCount`, как показано в следующем коде:

```
public static IWebHostBuilder CreateWebHostBuilder(string[] args) =>
  WebHost.CreateDefaultBuilder(args)
    .UseLibuv(opts => opts.ThreadCount = 4).UseStartup<Startup>();
```

`WebHost` заменили общим хостом в .NET Core 3.0. Ниже вы видите фрагмент кода для ASP.NET Core 3.0:

```
public static IHostBuilder CreateHostBuilder(string[] args) =>
  Host.CreateDefaultBuilder(args)
    .ConfigureWebHostDefaults(webBuilder => {
    webBuilder.UseStartup<Startup>();
  });
```

ASP.NET Core 2.x

Начиная с версии ASP.NET 2.1 в Kestrel заменили реализацию транспортного протокола от `libuv` на управляемые сокеты. Так что если вы обновляете свой проект с ASP.NET Core 1.x на ASP.NET 2.x или 3.x и хотите при этом использовать `libuv`, то вам нужно добавить пакет NuGet `Microsoft.AspNetCore.Server.Kestrel.Transport.Libuv`, чтобы ваш код работал.

В настоящее время в Kestrel поддерживаются следующие сценарии:

- HTTPS;
- непрозрачные обновления сессий, которые используются для поддержки веб-сокетов (https://github.com/aspnet/websockets);
- сокеты Unix в Nginx для высокой производительности;
- HTTP/2 (в настоящее время не поддерживаются на macOS).

Поскольку Kestrel построен на сокетах, можно настроить ограничение на число подключений с помощью метода `ConfigureLimits` в `Host`:

```
Host.CreateDefaultBuilder(args)
  .ConfigureKestrel((context, options) => {
     options.Limits.MaxConcurrentConnections = 100;
     options.Limits.MaxConcurrentUpgradedConnections = 100;
}
```

По умолчанию ограничений нет, и значение `MaxConcurrentConnections` равно 0.

Лучшие практики использования многопоточности в микросервисах

В последнее время микросервисы являются наиболее популярными шаблонами проектирования для создания производительных и масштабируемых серверных приложений. Вместо создания одной большой службы для всего приложения создается несколько связанных сервисов, каждый из которых отвечает за свой блок задач. В зависимости от нагрузки эти сервисы могут динамически масштабироваться. Следовательно, при проектировании микросервисов очень важно выбрать правильную модель многопоточности.

Микросервисы могут сохранять свое состояние во время работы или не делать этого. Выбор того или иного вида микросервисов сказывается на производительности. В службах без сохранения состояния (stateless) запросы могут обслуживаться в любом порядке, независимо от того, что было до и после текущего запроса, в то время как в службах с сохранением состояния (stateful) все запросы должны обрабатываться в определенном порядке, например в порядке очереди. При использовании асинхронности в микросервисах нам необходима дополнительная логика обработки запросов в правильной последовательности. Микросервисы могут быть также однопоточными или многопоточными. Сохранение состояния и изменение количества параллельных потоков заметно влияет на производительность, поэтому нужно хорошо продумать этот момент при проектировании сервисов.

Классификация подходов к проектированию микросервисов:

- микросервисы с одним потоком и одним процессором;
- микросервисы с одним потоком и несколькими процессорами;
- микросервисы с несколькими потоками и одним процессором.

Мы рассмотрим эти подходы более подробно в следующих подразделах.

Микросервисы с одним потоком и одним процессором

Это самый простой подход к проектированию микросервисов. Микросервис работает в одном потоке и на одном ядре центрального процессора. Для каждого нового запроса клиента создается новый поток, порождающий новый процесс. Для этого сценария кеширование пула соединений становится невозможным. Например, при работе с базой данных каждый новый процесс будет создавать новый пул соединений. Кроме того, одновременно может создаваться только один процесс, из-за чего возможно обслуживание лишь одного клиента.

К недостаткам таких микросервисов относится бесполезная трата ресурсов, а также то, что пропускная способность службы не увеличивается при увеличении нагрузки.

Микросервисы с одним потоком и несколькими процессорами

Этот вид микросервисов работает в одном потоке и способен порождать несколько процессов с хорошей пропускной способностью. Поскольку для каждого клиента создается новый процесс, воспользоваться преимуществами пула соединений при подключении к базам данных мы не можем. Такие сторонние среды, как Zend, OpCache и APC, предоставляют кросс-процессные кеши кодов операций (opcodes).

Плюсом таких микросервисов является то, что они повышают пропускную способность при нагрузке, ограничивая лишь возможность использования пула соединений.

Микросервисы с несколькими потоками и одним процессором

Микросервис использует несколько потоков, при этом существует лишь один процесс. Это позволяет использовать пул соединений при работе с базой данных, а также ограничивать количество подключений по мере необходимости. Проблема одного процессора заключается в том, что всеми потоками будет использоваться общий ресурс, а следовательно, могут возникнуть «гонки».

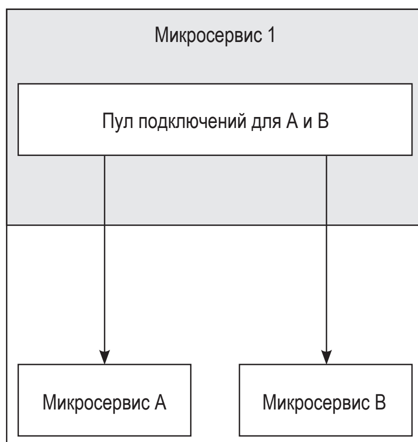
Плюсом данного подхода является повышение производительности сервисов без сохранения состояния. Однако могут возникнуть проблемы с синхронизацией при доступе к общему ресурсу.

Асинхронные сервисы

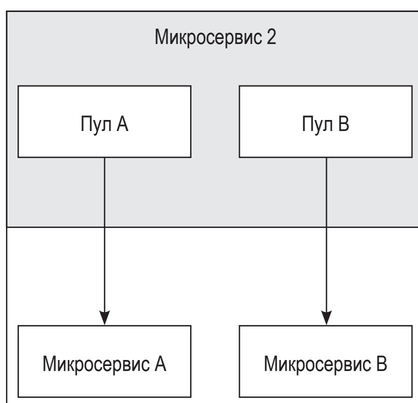
Для избежания проблем с производительностью при интеграции различных компонентов приложения рекомендуется уменьшать количество связей между микросервисами. Для этого необходимо, чтобы создание микросервисов было асинхронным.

Выделенные пулы потоков

Если для работы приложения требуется подключение к нескольким микросервисам, то лучше создать выделенный пул потоков для таких задач. При использовании единственного пула потоков возможна ситуация, в которой проблемы в коде могут привести к исчерпанию доступных потоков, что скажется на производительности. Чтобы избежать таких проблем в работе микросервиса, можно применить **Шаблон отсеков** (Bulkhead). На следующей диаграмме показаны два микросервиса с общим пулом. Как видите, оба микросервиса используют общий пул подключений:



На следующей диаграмме показаны два микросервиса с выделенными пулами потоков:



Далее мы расскажем о возможностях использования асинхронности в ASP.NET MVC Core.

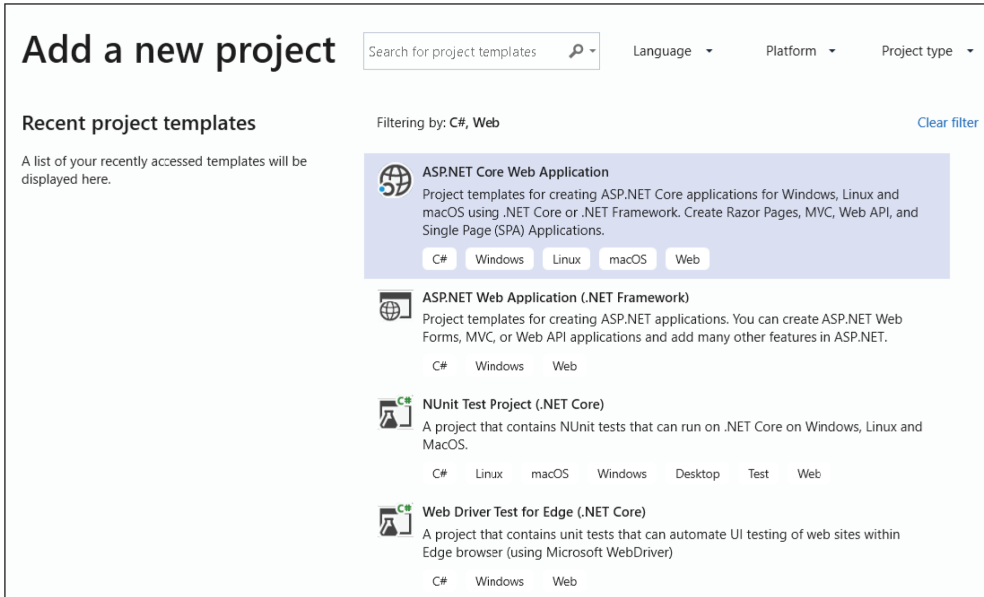
Введение асинхронности в ASP.NET MVC Core

Как вы помните, `async` и `await` являются ключевыми словами, которые используются при написании асинхронного кода с применением TPL. Они помогают писать асинхронный код в такой же манере, как и синхронный.

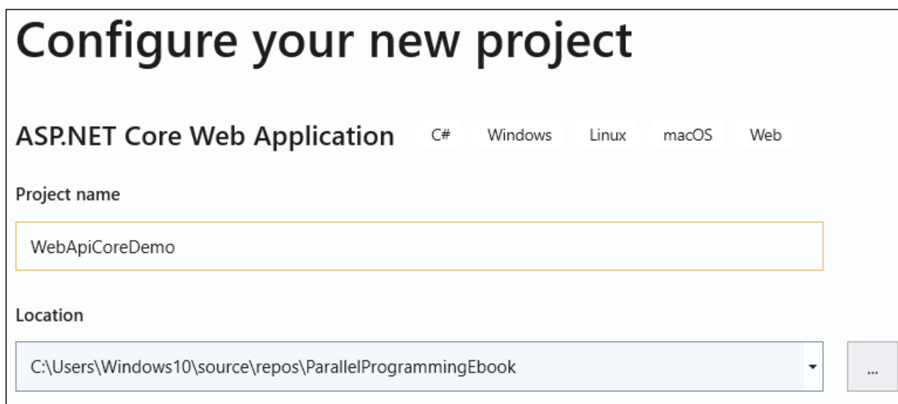
Мы уже говорили об `async` и `await` в главе 9 «Основы асинхронного программирования с помощью async, await и задач».

Попробуем создать асинхронный API с помощью ASP.NET Core 3.0 и версии VS 2019. API будет считывать файл с сервера.

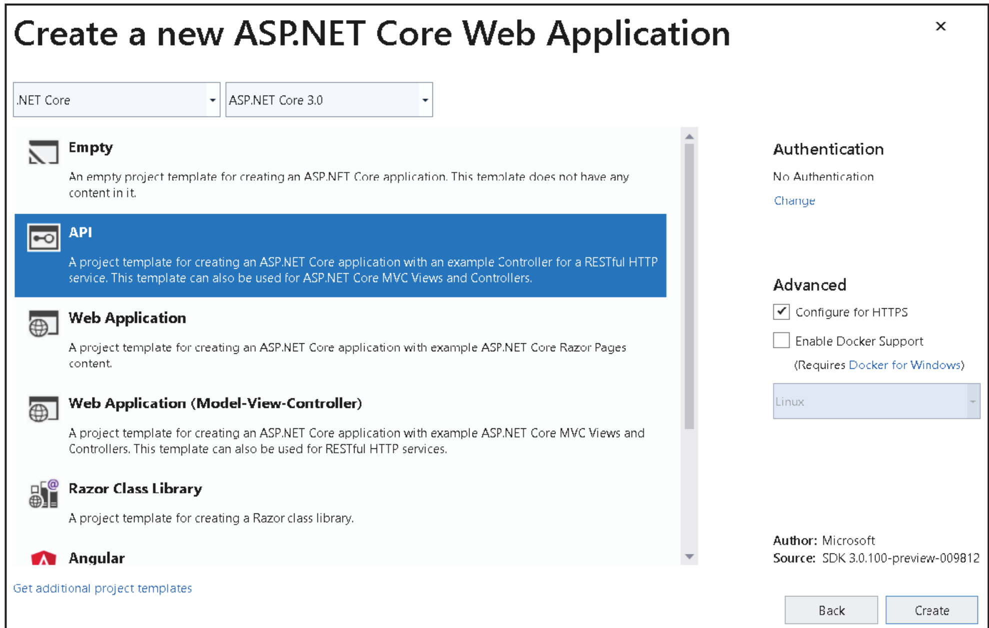
1. Откройте Visual Studio 2019 с экраном добавления проекта. Создайте новый проект **ASP.NET Core Web Application** в VS 2019, как показано ниже:



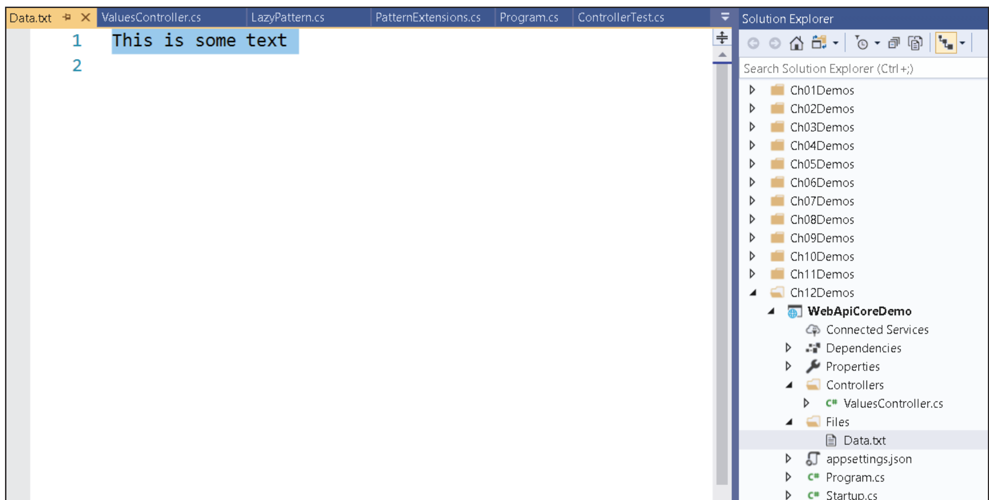
2. Обозначьте имя проекта и место, в котором вы его создаете:



3. Выберите тип проекта, в нашем случае это **API**, и нажмите **Create** (Создать):



4. Затем создайте новую папку в проекте под названием `Files` и добавьте файл с именем `data.txt` со следующим содержанием:



5. Измените метод `Get` в `ValuesController.cs`, как показано в примере:

```
[HttpGet]
public ActionResult<IEnumerable<string>> Get() {
  var filePath = System.IO.Path.Combine(
    HostingEnvironment.ContentRootPath, "Files", "data.txt");
```

```
        var text = System.IO.File.ReadAllText(filePath);
        return Content(text);
    }
```

Этот метод считывает файл с сервера и возвращает содержимое пользователю в виде строки. Проблема его заключается в том, что при вызове `File.ReadAllText` вызывающий поток блокируется до тех пор, пока файл полностью не прочитается. Мы можем легко сделать данный метод асинхронным:

```
[HttpGet]
public async Task <ActionResult<IEnumerable<string>>> GetAsync() {
  var filePath = System.IO.Path.Combine(
    HostingEnvironment.ContentRootPath, "Files", "data.txt");
  var text = await System.IO.File.ReadAllTextAsync(filePath);
  return Content(text);
}
```

ASP.NET Core Web API поддерживает новые возможности параллельного программирования, включая асинхронность, как мы уже видели ранее.

Асинхронные потоки

.NET Core 3.0 поддерживает асинхронные потоки. `IAsyncEnumerable<T>` является асинхронной версией `IEnumerable<T>`. Она позволяет ожидать новые элементы в цикле `foreach` для множества типа `IAsyncEnumerable<T>` с помощью фоновых задач. Ключевое слово `yield` возвращает новое значение множества из таких фоновых задач.

Это важно для сценариев с асинхронным перебором элементов и выполнением вычислительных операций над ними. Поскольку сейчас все чаще приходится работать с большими объемами данных, то стоит сделать выбор в пользу *асинхронных* потоков. Благодаря эффективной реализации многопоточности сервисы становятся более отзывчивыми.

Поддержка асинхронных потоков реализована с помощью двух новых интерфейсов:

```
public interface IAsyncEnumerable<T> {
  public IAsyncEnumerator <T> GetEnumerator();
}

public interface IAsyncEnumerator<out T> {
  public T Current { get; }
  public Task<bool> MoveNextAsync();
}
```

Из определения `IAsyncEnumerator` видно, что `MoveNext` сделали асинхронным, получив при этом следующие преимущества:

- кешировать `Task<bool>` вместо `Task<T>` стало проще, благодаря чему сокращается объем необходимой памяти;
- существующим коллекциям нужно просто добавить один дополнительный метод для асинхронности.

Попробуем в этом разобраться на примере кода, который асинхронно возвращает числа с нечетными индексами.

Ниже представлен пользовательский перечислитель (enumerator):

```
class OddIndexEnumerator: IAsyncEnumerator<int> {
  List <int> _numbers;
  int _currentIndex = 1;
  public OddIndexEnumerator(IEnumerable<int> numbers) {
    _numbers = numbers.ToList();
  }
  public int Current {
    get {
      Task.Delay(2000);
      return _numbers[_currentIndex];
    }
  }
  public ValueTask DisposeAsync() {
    return new ValueTask(Task.CompletedTask);
  }
  public ValueTask<bool> MoveNextAsync() {
    Task.Delay(2000);
    if (_currentIndex < _numbers.Count() - 2) {
      _currentIndex += 2;
      return new ValueTask<bool>(Task.FromResult<bool>(true));
    }
    return new ValueTask<bool>(Task.FromResult<bool>(false));
  }
}
```

Как видно из реализации `MoveNextAsync()`, этот метод начинает с нечетного индекса (то есть 1) и продолжает считывать элементы с нечетными индексами.

Ниже представлена коллекция с созданной нами ранее пользовательской логикой перечисления. Она реализует метод `GetAsyncEnumerator()` интерфейса `IAsyncEnumerable<T>` для возврата созданного нами `OddIndexEnumerator`:

```
class CustomAsyncIntegerCollection: IAsyncEnumerable<int> {
  List<int> _numbers;
  public CustomAsyncIntegerCollection(IEnumerable<int> numbers) {
    _numbers = numbers.ToList();
  }
  public IAsyncEnumerator<int> GetAsyncEnumerator(
    CancellationToken cancellationToken = default) {
    return new OddIndexEnumerator(_numbers);
  }
}
```

Так мы получили «волшебный» метод расширения, который и преобразует нашу коллекцию в `AsyncEnumerable`. Как видите, он может работать с любой коллекцией, реализующей `IEnumerable<int>`, а также способен оборачивать базовую коллекцию с помощью `CustomAsyncIntegerCollection`, которая, в свою очередь, реализует `IAsyncEnumerable<T>`:

```
public static class CollectionExtensions {
  public static IAsyncEnumerable<int> AsEnumerable(
    this IEnumerable<int> source) =>
      new CustomAsyncIntegerCollection(source);
}
```

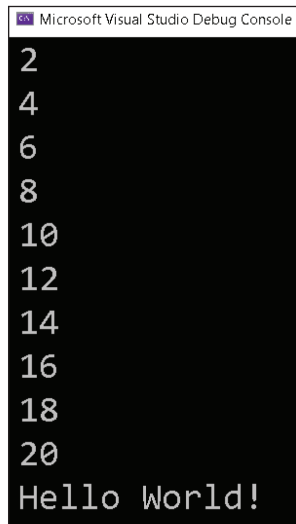
Как только все будет на своих местах, мы сможем создать метод, возвращающий асинхронный поток данных. При помощи ключевого слова `yield` мы возвращаем элементы коллекции:

```
static async IAsyncEnumerable<int> GetBigResultsAsync() {
  var list = Enumerable.Range(1, 20);
  await foreach(var item in list.AsEnumerable()) {
    yield
    return item;
  }
}
```

Следующий код обращается к нашему потоку данных. В этом случае вызывается метод `GetBigResultsAsync()`, возвращающий `IAsyncEnumerable<int>` в цикле `foreach`:

```
async static Task Main(string[] args) {
  await foreach(var dataPoint in GetBigResultsAsync()) {
    Console.WriteLine(dataPoint);
  }
  Console.WriteLine("Hello World!");
}
```

Ниже вы видите вывод предыдущего кода. Можно заметить, что генерация чисел в коллекции была по нечетным индексам:



В этом разделе мы познакомились с асинхронными потоками, способствующими эффективному выполнению параллельных циклов по коллекции без блокировки вызывающего кода, чего так не хватало еще с момента появления TPL.

Подведем итоги данной главы.

Выводы

В этой главе мы поговорили о многопоточности в IIS, познакомились с реализацией сервисов на .NET Core, начиная от использования `libuv` и заканчивая .NET Core 2.0 для управления сокетами из .NET Core 2.1 и выше. Также обсуждали способы повышения производительности IIS, Kestrel и некоторых алгоритмов пула потоков, таких как поиск восхождения к вершине и предотвращения нехватки ресурсов. Вам были представлены концепции микросервисов и различные шаблоны многопоточности: микросервисы с одним потоком и одним процессором, микросервисы с одним потоком и несколькими процессорами, микросервисы с несколькими потоками и одним процессором.

Мы также рассмотрели использование асинхронности в ASP.NET MVC Core 3.0 и познакомились с новой концепцией асинхронных потоков данных в .NET Core 3.0. Удобство асинхронных потоков данных становится заметным в сценариях с большими объемами данных, в которых из-за большого объема информации появляется большая нагрузка на серверы.

В следующей главе мы поговорим о некоторых шаблонах, которые часто используются в параллельном и асинхронном программировании. Эти шаблоны помогут нам лучше разобраться в параллельном программировании.

Вопросы

1. Что из нижеперечисленного используется для размещения веб-приложений?
    1. `IWebHostBuilder`
    2. `IHostBuilder`
2. Какой из алгоритмов `ThreadPool` пытается максимизировать пропускную способность, используя меньшее число потоков?
    1. Поиск максимума
    2. Предотвращение зависания процессора

3. Какой подход к проектированию микросервисов ложный?
    1. Один поток – один процесс
    2. Один поток – несколько процессов
    3. Несколько потоков – один процесс
    4. Несколько потоков – несколько процессов

4. В новых версиях .NET Core допустимо ожидание циклов `foreach`.
    1. Верно
    2. Неверно

Глава 13

Шаблоны параллельного программирования

В предыдущей главе мы рассмотрели многопоточность в IIS и Kestrel вместе со способами их оптимизации для повышения производительности, а также новые возможности для асинхронного кода в .NET Core 3.0.

В этой главе мы познакомимся с шаблонами параллельного программирования и сосредоточимся на проблемных сценариях параллельного кода и их решениях путем методов параллельного программирования/асинхронности.

В методах параллельного программирования используется множество разных шаблонов, однако мы выделим наиболее важные.

В этой главе мы рассмотрим следующие шаблоны:

- `MapReduce`;
- Агрегирование (Aggregation);
- Разделить/объединить (Fork/join);
- Спекулятивная обработка (Speculative processing);
- Отложенность (Laziness);
- Разделяемое состояние (Shared state).

Технические требования

Вам нужно иметь хорошее представление о C# и параллельном программировании. Исходный код главы доступен на GitHub по ссылке https://github.com/PacktPublishing/-Hands-On-Parallel-Programming-with-C-8-and-.NET-Core-3/tree/master/Chapter13.

Шаблон MapReduce

Шаблон `MapReduce` был создан для решения проблем с большими объемами данных, в число которых входит параллельная обработка массива данных на вычислительном кластере. Шаблон также можно использовать на одноядерных процессорах.

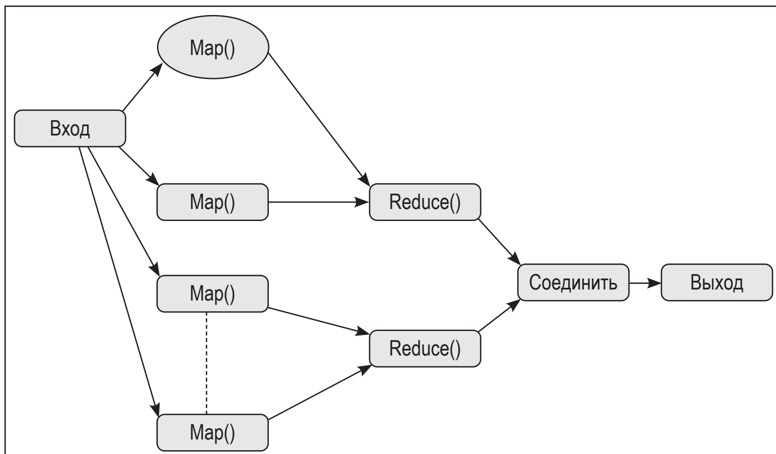
Программа `MapReduce` состоит из двух задач: **map** и **reduce**. Входные и выходные данные для программы `MapReduce` передаются и принимаются в виде коллекций ключ/значение.

Для реализации шаблона нужно сначала написать функцию `map`, которая будет принимать данные (пара ключ/значение) в виде единого входного значения и преобразовывать их в набор промежуточных данных (также в формате ключ/значение). Затем разработчик пишет функцию `reduce`, которая принимает выходные данные от `map` (пары ключ/значение) и объединяет их в небольшие блоки данных, состоящих из нескольких записей.

Давайте рассмотрим реализацию базового шаблона `MapReduce` с помощью LINQ и затем преобразуем ее в реализацию на основе PLINQ.

Реализация MapReduce с помощью LINQ

Ниже показано классическое графическое представление шаблона `MapReduce`. Входные данные проходят через несколько функций `Map()`, каждая из которых возвращает набор выходных данных. Затем эти данные группируются и объединяются при помощи функций `Reduce()` для создания конечного результата:



Для реализации шаблона `MapReduce` с помощью LINQ выполните следующие шаги.

1. Сначала реализуем функцию `map` с одним входным значением, возвращающую набор преобразованных значений. Для этого можно использовать функцию LINQ `SelectMany`.
2. Затем нужно сгруппировать данные в соответствии с промежуточным ключом. Это можно сделать при помощи метода LINQ `GroupBy`.
3. Вызовем метод `reduce`, который будет принимать промежуточный ключ в качестве входных данных, а также получать для этого соответствующий набор значений и производить вывод. Для этого можно использовать `SelectMany`.

4. Окончательная реализация MapReduce примет следующий вид:

```
public static IEnumerable<TResult> MapReduce<TSource, TMapped,
  TKey, TResult>(IEnumerable<TSource> source,
    Func<TSource, IEnumerable<TMapped>> map,
    Func<TMapped, TKey> keySelector,
    Func<IGrouping<TKey, TMapped>, IEnumerable<TResult>> reduce) {
  return source.SelectMany(map).GroupBy(keySelector)
    .SelectMany(reduce);
}
```

5. Поменяем ввод и вывод, чтобы шаблон работал с ParallelQuery<T>, а не с IEnumerable<T>, как показано в примере:

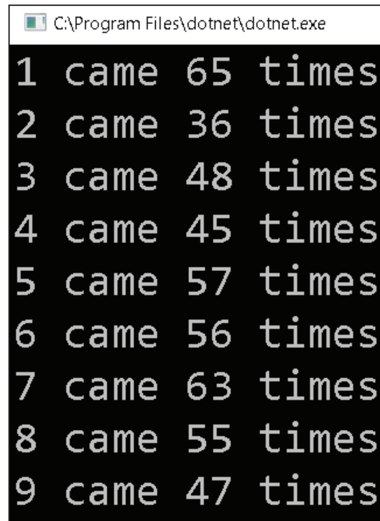
```
public static ParallelQuery<TResult> MapReduce<TSource, TMapped,
  TKey, TResult> (this ParallelQuery<TSource> source,
    Func<TSource, IEnumerable<TMapped>> map,
    Func<TMapped, TKey> keySelector,
    Func<IGrouping<TKey, TMapped>, IEnumerable<TResult>> reduce) {
  return source.SelectMany(map).GroupBy(keySelector)
    .SelectMany(reduce);
}
```

Ниже представлен пример использования пользовательской реализации MapReduce в .NET Core. Программа генерирует положительные и отрицательные случайные числа в целочисленном списке. Затем применяется схема распределения для фильтрации положительных чисел и их группировки по номерам. Наконец, применяется функция reduce для возврата списка чисел и их количества:

```
private static void MapReduceTest() {
  // Обрабатываем только положительные числа из списка
  Func<int, IEnumerable<int>> mapPositiveNumbers = number => {
    IList<int> positiveNumbers = new List<int>();
    if (number > 0)
      positiveNumbers.Add(number);
    return positiveNumbers;
  };
  // Группируем результаты вместе
  Func<int, int> groupNumbers = value => value;
  // Метод Reduce, который считает количество вхождений каждого числа
  Func<IGrouping<int, int>, IEnumerable<KeyValuePair<int,int>>>
    reduceNumbers = grouping => new [] {
    new KeyValuePair<int, int> (grouping.Key, grouping.Count())
  };
  // Создаем список случайных чисел от -10 до 10
  IList<int> sourceData = new List<int>();
  var rand = new Random();
  for (int i = 0; i < 1000; i++) {
    sourceData.Add(rand.Next(-10, 10));
  }
  // Используем метод MapReduce
  var result = sourceData.AsParallel()
```

```
    .MapReduce(mapPositiveNumbers, groupNumbers, reduceNumbers);
  // Обрабатываем результат
  foreach(var item in result) {
    Console.WriteLine($"{item.Key} came {item.Value} times");
  }
}
```

Ниже представлен вывод результатов работы, которые мы получаем при запуске этого программного кода в Visual Studio. Как видите, код повторяет предоставленный список и находит количество числовых повторений:



Далее мы познакомимся еще с одним распространенным и важным шаблоном параллельного проектирования – агрегацией. Работа шаблона `MapReduce` напоминает фильтр, а вот агрегация объединяет входные данные и преобразует их в другой формат.

Агрегация

Агрегация – распространенный в параллельных приложениях шаблон проектирования. В параллельных программах данные делятся на блоки (units), которые обрабатываются несколькими потоками на нескольких ядрах процессора. В агрегации возникает необходимость, когда нужно объединить данные из нескольких источников перед их отправкой пользователю.

Зачем же нужна агрегация, и что предоставляет PLINQ для ее реализации? Узнаем далее.

Обычное использование агрегации представлено ниже. В этом примере нужно повторить набор значений, выполнить операции и вернуть результат вызывающему объекту:

```
var output = new List<int>();
var input = Enumerable.Range(1, 50);
Func<int, int> action = (i) => i * i;
foreach(var item in input) {
  var result = action(item);
  output.Add(result);
}
```

Проблема кода в том, что вывод данных не потокобезопасен. Поэтому во избежание проблем перекрестных потоков воспользуемся примитивами синхронизации:

```
var output = new List<int>();
var input = Enumerable.Range(1, 50);
Func<int, int> action = (i) => i * i;
Parallel.ForEach(input, item => {
  var result = action(item);
  lock(output)
  output.Add(result);
});
```

Данный код хорошо работает при условии небольших вычислений, выполняемых для каждого элемента. Однако с увеличением вычислений возрастут и затраты на обработку, а также повысятся и затраты на саму блокировку, что приведет к снижению производительности. Параллельные коллекции, о которых мы говорили в главе 6 «Использование параллельных коллекций», помогают этого избежать. Благодаря параллельным коллекциям можно забыть о синхронизации. В следующем фрагменте кода как раз показано использование параллельных коллекций:

```
var input = Enumerable.Range(1, 50);
Func<int, int> action = (i) => i * i;
var output = new ConcurrentBag<int> ();
Parallel.ForEach(input, item => {
  var result = action(item);
  output.Add(result);
});
```

PLINQ определяет методы для агрегации и обрабатывает синхронизацию. В число этих методов входят такие, как `ToArray`, `ToList`, `ToDictionary` и `ToLookup`:

```
var input = Enumerable.Range(1, 50);
Func<int, int> action = (i) => i * i;
var output = input.AsParallel().Select(item =>
                                 action(item)).ToList();
```

В этом коде метод `ToList()` агрегирует все данные и осуществляет синхронизацию. TPL обладает шаблонами реализации, встроенными в языки программирования. Одним из них является шаблон разделения/объединения (fork/join), о котором мы как раз сейчас и поговорим.

Шаблон разделения/объединения

В шаблонах разделения/объединения (fork/join) работа *разбивается (fork)* на набор задач, которые могут асинхронно выполняться. Затем отдельные задачи объединяются в том же или другом порядке в соответствии с требованиями и областью распараллеливания. С общими примерами шаблонов fork/join мы уже познакомились, когда говорили об идеальных параллельных циклах. Некоторыми реализациями fork/join являются:

- `Parallel.For`;
- `Parallel.ForEach`;
- `Parallel.Invoke`;
- `System.Threading.CountdownEvent`.

Использование этих методов, предоставляемых платформой, помогает ускорить разработку без дополнительных расходов на синхронизацию. Благодаря этим шаблонам мы получаем высокую пропускную способность. Для повышения производительности и сокращения задержек также широко используют шаблон спекулятивной обработки.

Шаблон спекулятивной обработки

Шаблон спекулятивной обработки – это еще один шаблон параллельного программирования с высокой пропускной способностью для сокращения задержек. Этот шаблон помогает распознать самый быстрый способ получения результатов в сценариях со множеством вариантов выполнения задач. Создается задача для каждого возможного метода, которая впоследствии выполняется на разных процессорах. Задача, завершившаяся первой, используется для получения результатов, при этом остальные задачи уже не учитываются (даже если они успешно завершатся, ведь важна лишь скорость их завершения).

Ниже представлено традиционное представление `SpeculativeInvoke`. Массив `Func<T>` задается в качестве аргумента и выполняется параллельно, пока одна из функций не завершится:

```
public static T SpeculativeInvoke<T> (params Func<T>[] functions) {
  return SpeculativeForEach(functions, function => function ());
}
```

Следующий метод параллельно выполняет каждое переданное действие и выходит из параллельного цикла с помощью вызова метода `ParallelLoopState.Stop()`, как только одна из вызванных реализаций успешно выполнится:

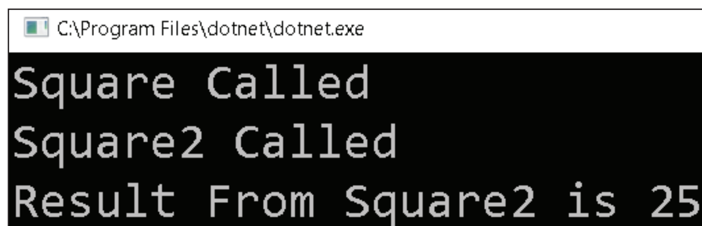
```
public static TResult SpeculativeForEach<TSource, TResult>(
  IEnumerable<TSource> source, Func<TSource, TResult> body) {
  object result = null;
  Parallel.ForEach(source, (item, loopState) => {
    result = body(item);
```

```
    loopState.Stop();
  });
  return (TResult) result;
}
```

Представленный далее код использует две различные реализации для вычисления квадрата числа 5. Передав обе функции методу `SpeculativeInvoke`, мы сразу же выведем `result`:

```
Func<string> Square = () => {
  Console.WriteLine("Square Called");
  return $ "Result From Square is {5 * 5}";
};
Func<string> Square2 = () => {
  Console.WriteLine("Square2 Called");
  var square = 0;
  for (int j = 0; j < 5; j++) {
    square += 5;
  }
  return $ "Result From Square2 is {square}";
};
string result = SpeculativeInvoke(Square, Square2);
Console.WriteLine(result);
```

А это вывод, получаемый по завершении кода:



Оба метода завершаются, но вызывающей программе возвращается только результат первого завершенного выполнения. Создание большого количества задач может негативно сказаться на системной памяти, поскольку увеличивается число переменных, которые нужно выделять и хранить. Поэтому крайне важно выделять объекты только по мере необходимости. Именно это и помогает делать следующий вариант шаблона.

Шаблон отложенной инициализации

Шаблон отложенной инициализации является еще одним шаблоном программирования, который используется разработчиками для повышения производительности приложений. Этот шаблон откладывает вычисления до тех пор, пока они не понадобятся. В лучшем случае вычисления могут и не

потребоваться, в связи с чем сэкономятся вычислительные ресурсы и, таким образом, повысится производительность системы. Ленивые вычисления не являются новыми для высокопроизводительной обработки данных, и LINQ интенсивно использует *ленивую загрузку*. LINQ следует модели отложенного выполнения, в которой запросы не выполняются, пока при помощи функций итератора не будет вызван `MoveNext()`.

Ниже представлен пример потокобезопасного ленивого одноэлементного шаблона. Для создания он использует сложные вычислительные операции и поэтому откладывается:

```
public class LazySingleton<T> where T: class {
  static object _syncObj = new object();
  static T _value;
  private LazySingleton() {}
  public static T Value {
    get {
      if (_value == null) {
        lock(_syncObj) {
          if (_value == null)
            _value = SomeHeavyCompute();
        }
      }
      return _value;
    }
  }
  private static T SomeHeavyCompute() {
    return default (T);
  }
}
```

Ленивый объект создается путем вызова свойства `Value` класса `LazySingleton<T>`. Отложенная инициализация гарантирует, что объект будет создан после вызова свойства `Value`. Создание одноэлементной реализации предполагает, что один и тот же объект возвратится при последующих вызовах. Проверка `_value` на `null` позволяет избежать возникновения блокировок при последующих вызовах, тем самым сохраняя некоторые операции ввода-вывода и повышая производительность.

Этот же код можно сократить при помощи `System.Lazy<T>`, как показано в примере:

```
public class MyLazySingleton<T> {
  // Объявляем экземпляр Lazy<T> с функцией инициализации
  // (SomeHeavyCompute)
  static Lazy<T> _value = new Lazy<T> ();
  // Свойство Value для возврата значения экземпляра Lazy, когда оно
  // действительно требуется по коду
  public T Value {
    get {
      return _value.Value;
    }
  }
```

```
  // Функция инициализации
  private static T SomeHeavyCompute() {
    return default (T);
  }
}
```

В асинхронном программировании взаимодействие `Lazy<T>` с TPL может принести существенные результаты.

Ниже вы видите пример использования `Lazy<T>` и `Task<T>` для реализации ленивого и асинхронного поведений:

```
var data = new Lazy<Task<T>>(() =>
  Task<T>.Factory.StartNew(SomeHeavyCompute));
```

Получить доступ к исходной `Task` можно также через свойство `data.Value`. Благодаря исходной ленивой реализации один и тот же экземпляр задачи будет каждый раз возвращаться независимо от количества вызовов свойства `data.Value`. Такой принцип работы будет полезен в сценариях, в которых для асинхронной обработки вам будет достаточно запуска одного потока вместо нескольких.

Рассмотрим следующий фрагмент кода. Здесь происходит извлечение данных из службы и их сохранение в файлы Excel или CSV из двух различных реализаций потоков:

```
public static string GetDataFromService() {
  Console.WriteLine("Service called");
  return "Some Dummy Data";
}
```

В следующих примерах использована логика, которую можно сохранить в виде текста или в формате CSV:

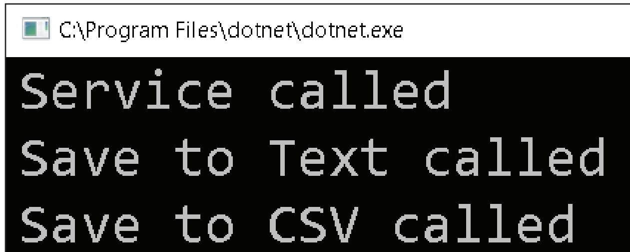
```
public static void SaveToText(string data) {
  Console.WriteLine("Save to Text called");
  // Сохраняем в текстовый файл
}

public static void SaveToCsv(string data) {
  Console.WriteLine("Save to CSV called");
  // Сохраняем в формате CVS
}
```

В примере ниже показано, как оборачивается вызов службы внутри `lazy`. Это помогает нам убедиться в том, что вызов службы выполняется единожды, в то время как выходные данные асинхронно используются. Как видите, метод отложенной инициализации обернули в задачу с помощью `Task.Factory.StartNew(GetDataFromService)`:

```
Lazy<Task<string>> lazy = new Lazy<Task<string>>
(Task.Factory.StartNew(GetDataFromService));
lazy.Value.ContinueWith((s) => SaveToText(s.Result));
lazy.Value.ContinueWith((s) => SaveToCsv(s.Result));
```

Вывод этого кода будет выглядеть следующим образом:



Как видите, вызов службы однократен. Использование шаблонной инициализации представляется целесообразным при создании объектов. При создании множества задач мы сталкиваемся с проблемами синхронизации ресурсов. В таком случае на помощь приходят шаблоны разделяемого состояния.

Шаблон разделяемого состояния

С реализацией данного шаблона мы уже познакомились в главе 5 «Примитивы синхронизации».

У параллельных приложений обычно есть проблемы с разделяемыми данными. Элементы данных приложения следует обезопасить при их использовании в многопоточном окружении. Существует множество способов для решения проблем с общим состоянием, например использование `Synchronization` (синхронизации), `Isolation` (изоляции) и `Immutability` (неизменяемости). Благодаря имеющимся в .NET Framework примитивам можно получить синхронизацию, обеспечивающую эксклюзивный доступ к общему элементу данных. Неизменность элемента данных означает, что он может находиться лишь в одном состоянии и никогда не изменяется, поэтому может быть без проблем использован в нескольких потоках. Изоляция предполагает наличие у всех потоков своих собственных копий элемента данных.

Подведем итоги этой главы.

Выводы

В данной главе мы познакомились с дополнительными шаблонами параллельного программирования и рассмотрели примеры их использования. Список шаблонов довольно многообразен. Представленные нами шаблоны могут послужить хорошей отправной точкой для разработчиков параллельных программ.

Вкратце мы поговорили о шаблоне `MapReduce`, шаблоне спекулятивной обработки, шаблоне отложенной инициализации и агрегации. Также мы с вами

рассмотрели такие шаблоны, как разделение/объединение и разделяемое состояние, используемые в библиотеках .NET Framework для параллельного программирования.

В следующей главе мы познакомим вас с управлением распределенной памятью и сфокусируемся на моделях разделяемой и распределенной памяти; рассмотрим различные типы коммуникационных сетей, а также посмотрим на примеры их реализации.

Вопросы

1. Что из перечисленного не является реализацией шаблона fork/join?
   1. `System.Threading.Barrier`
   2. `System.Threading.Countdown`
   3. `Parallel.For`
   4. `Parallel.ForEach`
2. Что из перечисленного является реализацией шаблона отложенной инициализации в TPL?
   1. `Lazy<T>`
   2. `LazySingleton`
   3. `LazyInitializer`
3. В основе какого шаблона лежит достижение высокой пропускной способности для уменьшения задержки?
   1. Отложенного
   2. Общего состояния
   3. Спекулятивной обработки
4. При помощи какого шаблона вы можете отфильтровать данные из списка и вернуть единственный результат?
   1. Агрегация
   2. `MapReduce`

Глава 14

Управление распределенной памятью

За последние два десятилетия в отрасли произошел сдвиг парадигмы в сторону больших массивов данных и архитектур машинного обучения, оперативно обрабатывающих терабайты/петабайты данных. По мере того как дешевела вычислительная мощность, стали использовать несколько процессоров, заметно ускоряющих обработку. Подобное новаторство привело к появлению распределенных вычислений, которые структурируют компьютерные системы, подключенные через сетевое или промежуточное программное обеспечение. Объединенные системы имеют общие ресурсы и координируются этим типом ПО для создания образа единой системы. В связи с большими размерами современных приложений и предъявляемыми к ним требованиями по производительности возникает необходимость в распределительных вычислениях. Ниже представлены некоторые распространенные сценарии, в которых предъявляются высокие требования по производительности:

- Google выполняет не менее 1,5 триллиона поисковых запросов в год;
- устройства интернета вещей отправляют несколько терабайт данных в концентраторы событий;
- хранилища данных получают и вычисляют терабайты записей за короткие сроки.

В этой главе мы поговорим об управлении распределенной памятью и о необходимости в распределенных вычислениях. Также мы познакомимся с классификацией сетей связи и рассмотрим принципы передачи сообщений по сетям связи для распределенных систем.

Данная глава представлена следующими темами:

- преимущества распределенных систем;
- модель общей памяти и модель распределенной памяти;
- классификация сетей связи;
- свойства сетей связи;
- изучение топологий;

- программирование устройств с распределенной памятью при передаче сообщений;
- совместное использование ресурсов.

Технические требования

В рамках данной главы вам нужно разбираться в программировании на C и использовании Windows API для C#.

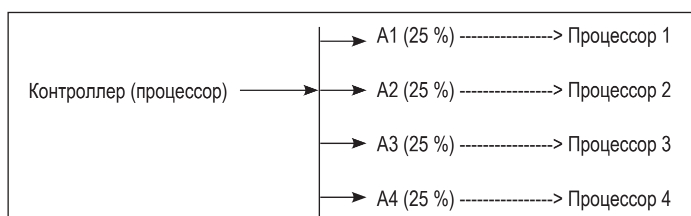
Введение в распределенные системы

Ранее мы уже говорили о работе распределенных вычислений. В этом разделе мы попытаемся разобрать распределенные вычисления на примере, работающем с массивами.

Допустим, нам нужно найти сумму всех чисел массива, состоящего из 1040 элементов:

```
a = [1,2,3,4....,n];
```

Если общее время на добавление чисел (скажем, многозначных) равно *x*, а нам срочно нужно найти их сумму, мы можем воспользоваться распределенными вычислениями. Разделив массив на несколько частей (на 4 массива, например), в каждой из которых будет 25 % исходного количества элементов, мы бы отправили каждый такой массив отдельному процессору для вычисления суммы (см. пример ниже):



В данном случае время на сложение всех чисел уменьшается до (x/4 + d), или (x/количество процессоров + d), где d – время, затраченное на объединение сумм всех процессоров и их сложение для получения результата.

Преимущества распределенных систем:

- системы масштабируются до любого уровня без аппаратных ограничений;
- отсутствие единой точки отказа, способствующее отказоустойчивости систем;
- доступность;
- эффективность при решении проблем с большими данными.

Распределенные системы часто путают с параллельными, однако между ними есть небольшая разница. **Параллельные системы** представляют собой систему из нескольких процессоров, которые размещаются в основном в одном и иногда в нескольких расположенных рядом хранилищах. **Распределенные системы** состоят из нескольких процессоров (у каждого имеется собственная память и устройства ввода-вывода), которые соединяются через сеть, обеспечивающую обмен данными.

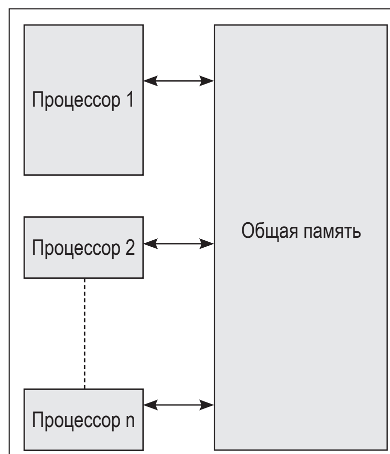
Модель общей и распределенной памяти

Для высокой производительности были разработаны **многопроцессорные** и **многокомпьютерные** архитектуры. В многопроцессорной архитектуре несколько процессоров совместно используют общую память и взаимодействуют друг с другом, считывая/записывая данные в общую память. В многокомпьютерной архитектуре у компьютеров нет общей физической памяти для взаимодействия друг с другом при передаче сообщения. **Распределенная общая память** (Distributed Shared Memory, DSM) имеет дело с общим доступом к памяти в физической, неразделяемой (распределенной) архитектуре.

Рассмотрим эти архитектуры и обсудим их различия.

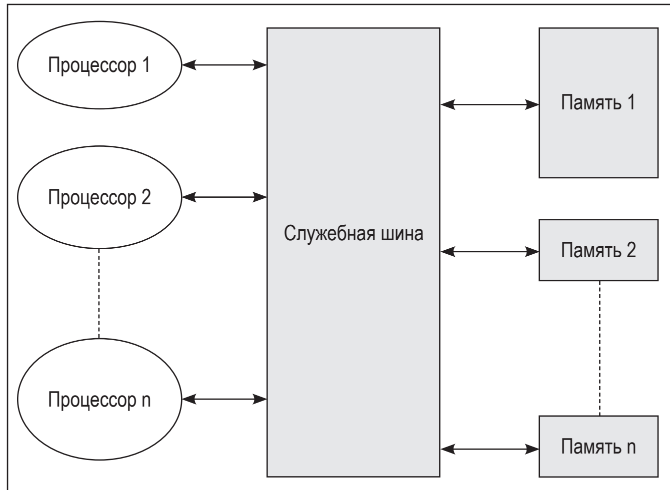
Модель общей памяти

В этой модели несколькими процессами используется одно общее пространство памяти. Поскольку это пространство памяти едино для всех процессов, возникает необходимость в мерах по синхронизации во избежание повреждения данных и возникновения условий гонки. Как упоминалось ранее, синхронизация влечет за собой издержки на производительность. Ниже изображен пример модели общей памяти. В данном случае устройство имеет количество процессоров – **n**, у которых есть доступ к общему блоку памяти:



Особенности модели общей памяти:

- у процессоров есть доступ ко всему блоку памяти. Блоком памяти может быть и отдельный фрагмент памяти, состоящий из модулей памяти (схема ниже):

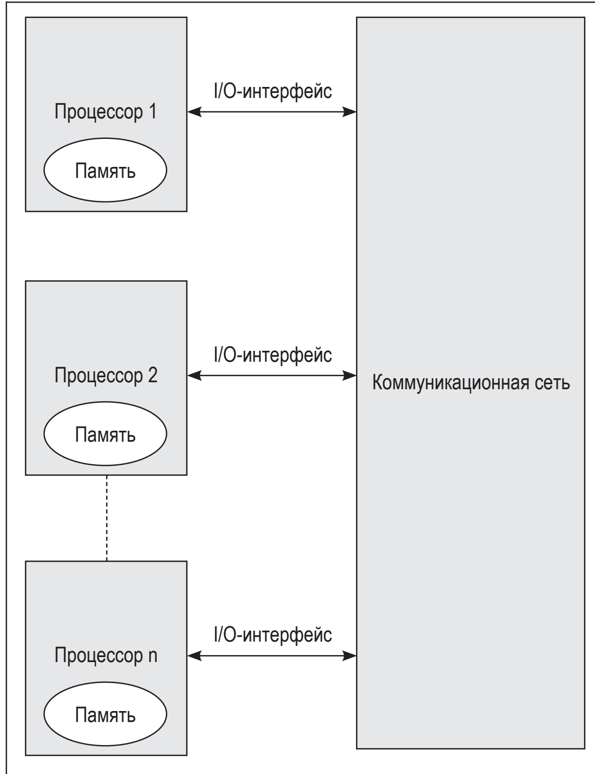


- процессоры взаимодействуют друг с другом, создавая общие переменные в основной памяти;
- эффективность распараллеливания во многом зависит от скорости сервисной шины (service bus);
- из-за скорости сервисной шины система масштабируется только до n-го количества процессоров.

Модели общей памяти известны как модели симметричной многопроцессорной обработки (symmetric multiprocessing, SMP), поскольку процессоры обладают доступом ко всем доступным блокам памяти.

Модель распределенной памяти

В модели распределенной памяти процессоры не обладают общей физической памятью и могут располагаться удаленно. У каждого процессора есть свое личное пространство памяти и устройства ввода-вывода. Хранение данных происходит не в одном компьютере, а в разных. Каждый процессор может работать со своими локальными данными, но для доступа к данным, хранящимся в памяти других процессоров, ему необходимо подключиться через коммуникационную сеть. Передача данных между процессорами происходит посредством **передачи сообщений** с использованием инструкций *отправки сообщения* (send message) и *получения сообщения* (receive message). Ниже представлена схема модели распределенной памяти:



На диаграмме отображены процессоры со своим пространством памяти, а также показано их взаимодействие с **коммуникационными сетями** через интерфейсы ввода-вывода. Разберем разные типы коммуникационных сетей, используемых в распределенных системах.

Типы коммуникационных сетей

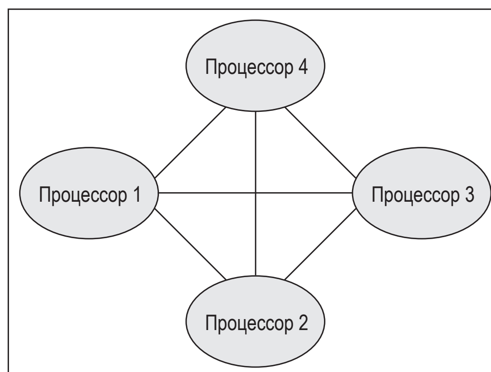
Коммуникационные сети представляют собой связи между двумя или более узлами в традиционной компьютерной сети. Они подразделяются на две категории:

- статические коммуникационные сети;
- динамические коммуникационные сети.

Рассмотрим эти топологии.

Статические коммуникационные сети

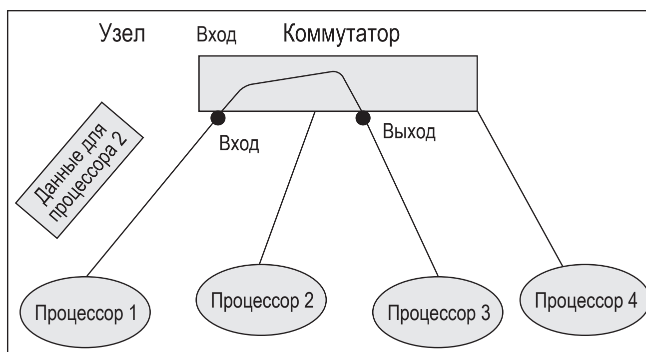
Статические коммуникационные сети состоят из связей, как показано на диаграмме:



Связи используются для соединения узлов, создавая полную коммуникационную сеть, в которой узлы взаимодействуют между собой.

Динамические коммуникационные сети

Динамические коммуникационные сети содержат каналы связи и коммутаторы, как показано на диаграмме:



Коммутаторы – устройства с портами ввода/вывода, перенаправляющие входные данные в порты вывода. Это означает, что пути создаются динамически. Отправка данных одного процессора другому должна быть выполнена через коммутатор (см. диаграмму выше).

Свойства коммуникационных сетей

При проектировании коммуникационной сети необходимо учитывать следующие характеристики:

- топологию;

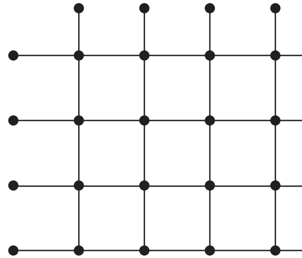
- алгоритм маршрутизации;
- стратегию коммутации;
- управление потоком.

Далее мы подробнее познакомимся с каждой из характеристик.

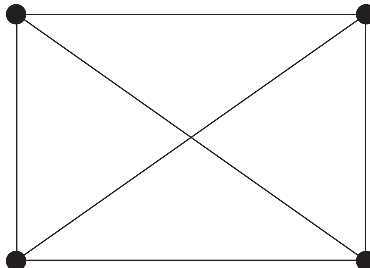
Топология

Топология показывает, как происходит подключение узлов (мосты, коммутаторы и устройства инфраструктуры). Некоторые распространенные топологии: топология перекрестной коммутации (crossbar), кольцо (ring), двумерная решетка (2D mesh), трехмерная решетка (3D mesh), многомерная решетка (xD mesh), двухмерный тор (2D torus), трехмерный тор (3D torus), многомерный тор (xD torus), гиперкуб (hypercube), дерево (tree), «бабочка» (butterfly), идеальное тасование (perfect shuffle) и «стрекоза» (dragonfly).

В топологии перекрестной коммутации все узлы в сети подключены друг к другу (могут быть подключены не напрямую). Таким образом, сообщения могут передаваться по нескольким маршрутам во избежание конфликтов. Ниже представлена стандартная топология перекрестной коммутации:



В топологии решетки, или, как ее обычно называют, ячеистой (meshnet), узлы соединяются друг с другом напрямую, не завися от других узлов в сети. Таким образом, все узлы могут передавать информацию независимо друг от друга. Решетка бывает частично или полностью связанной. Ниже представлена полностью связанная решетка:



В разделе «Исследование топологий» мы детальнее разберем эту топологию.

Алгоритмы маршрутизации

Маршрутизация представляет собой процесс передачи отдельного пакета данных до определенного узла по сети. Маршрутизация бывает адаптивной и неадаптивной. Адаптивная реагирует на изменения в топологии сети, непрерывно принимая информацию от смежных узлов. При неадаптивной маршрутизации узлы статичны, в них маршрутная информация загружается при запуске сети. Необходимо использовать правильные алгоритмы маршрутизации, чтобы не возникало взаимоблокировок. Так, например, в 2D-торе пути перемещаются с востока на запад и с севера на юг во избежание тупиковых маршрутов. Подробнее мы рассмотрим 2D-тор немного позже.

Стратегия коммутации

Правильно выбранная стратегия коммутации может повысить производительность сети. Двумя наиболее известными стратегиями коммутации являются:

- **коммутация каналов**: при коммутации каналов полный путь закрепляется за целым сообщением, например как в телефонной связи. Чтобы начать вызов в телефонной сети, необходимо установить выделенный канал между вызывающим и вызываемым абонентами, и этот канал сохраняется в течение всего времени вызова;
- **коммутация пакетов**: при коммутации пакетов сообщение разбивается на отдельные маршрутизируемые пакеты, например на группы связанных маршрутизаторами сетей, функционирующих как отдельные большие виртуальные сети. По сравнению с коммутацией каналов, коммутация пакетов более экономична, поскольку расходы на связь распределяются между пользователями. Этот тип коммутации в основном используется для асинхронных сценариев, таких как отправка электронной почты или передача файлов.

Управление потоком

Управление потоком представляет собой процесс, с помощью которого сеть обеспечивает надежную и эффективную передачу пакетов. В контексте топологии сети, скорости отправителя и получателя могут варьироваться, из-за чего возникает риск появления узких мест или потери пакетов. Управление потоком предоставляет нам несколько вариантов решений при перегрузке сети. Так, в число возможных сценариев поведения входят временное хранение данных в буфере, их перенаправление на другие узлы, принудительная приостановка узлов-источников, удаление данных и т. д. Ниже приведены общие алгоритмы по управлению потоком.

- **Остановка и ожидание**: сообщение разбивается на части. Отправитель отправляет часть сообщения получателю и ждет подтверждения в течение заданного времени (тайм-аут). Получив подтверждение, он передает следующую часть сообщения.
- **Скользящее окно**: получатель назначает отправителю окно передачи для отправки сообщений. Когда окно будет заполнено, отправитель приостановит передачу, позволяя получателю обработать сообщения и объявить следующее окно передачи. Такой подход наиболее эффективен, когда приемник хранит данные в буфере, и из этого следует, что он может принимать столько данных, сколько вмещает буфер.

Исследование топологий

Ранее мы рассматривали полные коммуникационные сети, в которых каждый процессор напрямую взаимодействует с другими без помощи коммутаторов. Данный способ работы эффективен при небольшом количестве процессоров, однако при увеличении их числа он может стать еще той проблемой. При измерении производительности графа в топологии необходимо учитывать:

- **диаметр графа**: самый длинный путь между узлами;
- **пропускную способность сечения**: пропускная способность через наименьший срез, разделяющий сеть на две равные половины. Это важно для сетей, в которых каждый процессор взаимодействует с другими.

Ниже представлены примеры некоторых сетевых топологий.

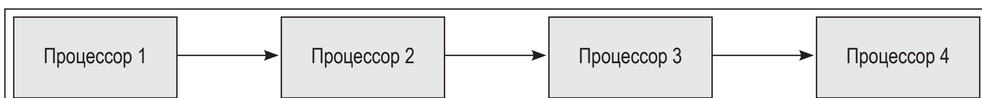
Линейная и кольцевая топологии

Эти топологии хорошо работают с одномерными массивами. В линейной топологии процессоры находятся в линейном расположении с одним входным и выходным потоком, в то время как в кольцевой процессоры закольцовываются.

Давайте подробнее рассмотрим данные топологии.

Линейные массивы

Процессоры линейно располагаются, как показано на диаграмме:

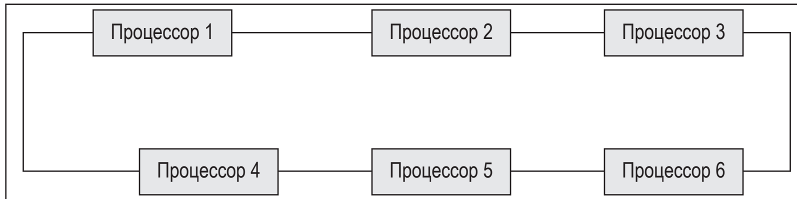


Это расположение имеет следующие значения:

- диаметр = $n - 1$, где n – количество процессоров;
- пропускная способность сечения = 1.

Кольцо или тор

Процессоры закольцовываются, при этом информация поступает от одного процессора к другому, делая петлю обратной связи к исходному процессору. Создается кольцо, как показано на диаграмме ниже:



У данного расположения будут следующие значения:

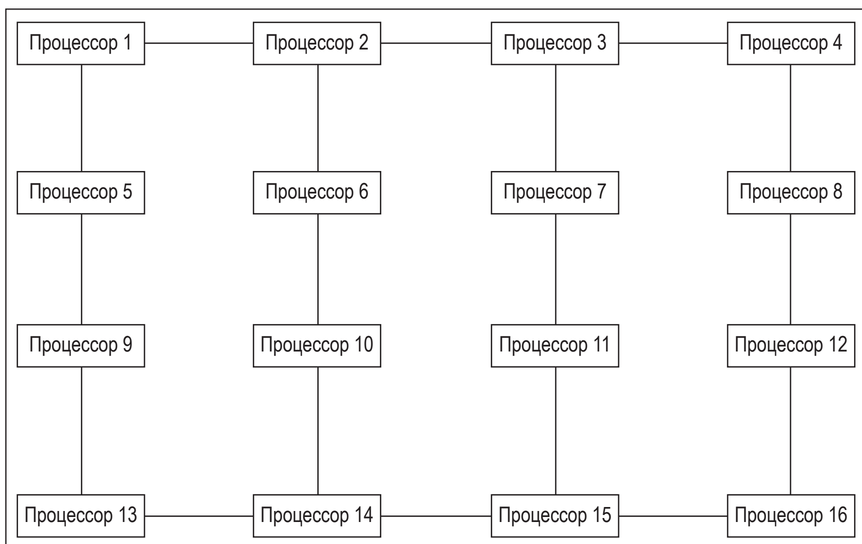
- диаметр = $n/2$, где n – количество процессоров;
- пропускная способность сечения = 2.

Решетки и торы

Эти топологии хорошо работают с 2D- и 3D-массивами. Давайте рассмотрим их более подробно.

Двумерные решетки

Здесь узлы соединяются друг с другом напрямую без зависимости от других узлов в сети. В двумерной решетке узлы имеют следующее расположение (см. диаграмму ниже):

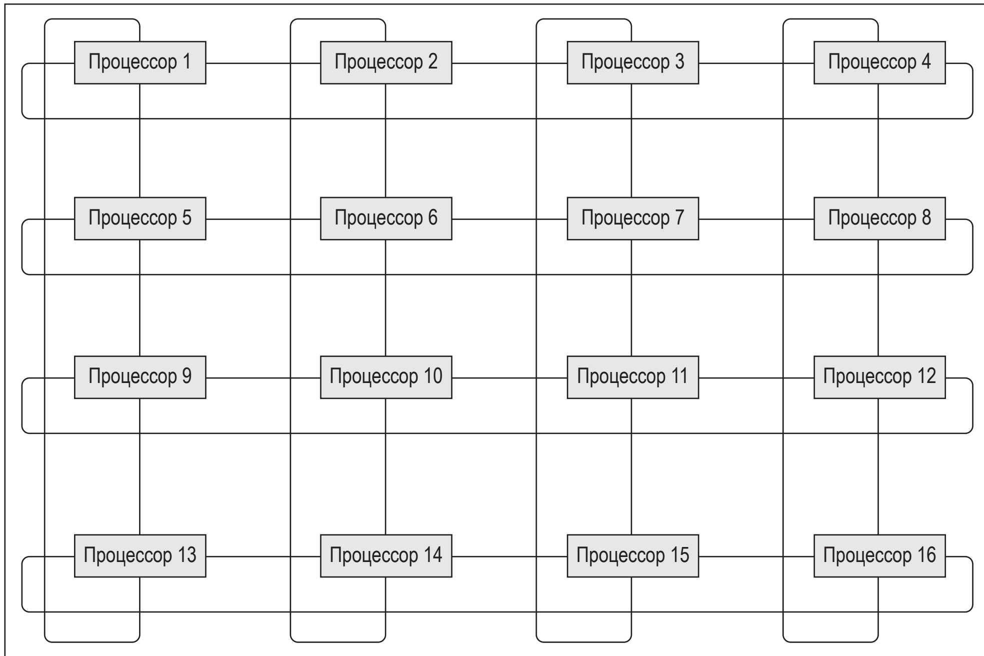


У данного расположения будут следующие значения:

- диаметр = 2 * (sqrt(*n*) – 1), где *n* – количество процессоров;
- пропускная способность сечения = sqrt(*n*).

2D-тор

На диаграмме показано расположение процессоров в 2D-торе:



У данного расположения будут следующие значения:

- диаметр = sqrt(*n*), где *n* – количество процессоров;
- пропускная способность сечения = 2 * sqrt(*n*).

Программирование устройств с распределенной памятью с использованием передачи сообщений

В этом разделе мы обсудим, как программировать машины с распределенной памятью с помощью **интерфейса передачи сообщений** (message passing interface, MPI) в реализации Microsoft.

MPI – это стандарт портируемой системы, разработанный для распределенных и параллельных систем. Он определяет базовый набор функций,

которые используются поставщиками параллельного оборудования для поддержки связи с распределенной памятью. В следующих разделах мы обсудим преимущества применения MPI по сравнению со старыми библиотеками обмена сообщениями и объясним, как установить и запустить простую программу MPI.

Почему MPI?

Преимущество MPI заключается в том, что механизмы MPI могут вызываться из различных языков, таких как C, C++, C#, Java, Python и др. MPI более платформонезависим в сравнении со старыми библиотеками обмена сообщениями, и инструменты MPI оптимизированы по скорости для каждого поддерживаемого ими аппаратного обеспечения.

Установка MPI на Windows

MPI можно загрузить и установить в виде ZIP-файла по ссылке: https://www.open-mpi.org/software/ompi/v1.10/.

Также вы можете загрузить версию MPI от Microsoft с https://github.com/Microsoft/Microsoft-MPI/releases.

Пример программы с использованием MPI

Ниже представлена простая программа `HelloWorld`, которую запускает MPI. Программа печатает номер процессора, на котором выполняется код, с задержкой в две секунды. Один и тот же код может выполняться на нескольких процессорах (количество процессоров варьируется).

Давайте создадим новый проект консольного приложения в Visual Studio и запишем следующий код в файл `Program.cs`:

```
[DllImport("Kernel32.dll"), SuppressUnmanagedCodeSecurity]
public static extern int GetCurrentProcessorNumber();

static void Main(string[] args) {
  Thread.Sleep(2000);
  Console.WriteLine($"Hello {GetCurrentProcessorNumber()} Id");
}
```

`GetCurrentProcessorNumber()` является служебной функцией, выдающей номер процессора, на котором выполняется код. В предыдущем коде нет ничего удивительного – работает как один поток и печатает `Hello` вместе с текущим номером процессора.

Установим `msmpisetup.exe` по ссылке Microsoft MPI, которую мы расссмотрели ранее в подразделе «Установка MPI на Windows». После установки выполним следующую команду из командной строки (Command Prompt):

```
C:\Program Files\Microsoft MPI\Bin>mpiexec.exe -n 5 path_to_executable
```

Здесь n – количество процессоров, на которых запустится программа, а path_to_executable – путь к исполняемому файлу (exe) вашей программы.

Ниже представлен вывод предыдущего кода:

```
Hello 3 Id
Hello 7 Id
Hello 5 Id
Hello 6 Id
Hello 6 Id
```

Как видите, MPI позволяет запустить одну и ту же программу на нескольких процессорах.

Базовое использование отправки/приема сообщений

MPI реализован на C++, поэтому большая часть документации на сайте Microsoft доступна только для C++. Однако вы можете легко использовать библиотеки для C++ в вашем приложении для .NET. Существуют и другие сторонние реализации MPI для .NET, но, к сожалению, на момент написания книги они не поддерживали реализацию .NET Core.

Ниже представлен синтаксис функции MPI_Send, которая отправляет данные другому процессору:

```
int MPIAPI MPI_Send(
  _In_opt_ void *buf, // Указатель на буфер, содержащий данные для отправки
  int          count, // Количество элементов в буфере
  MPI_Datatype datatype, // Тип данных элемента буфера
  int          dest, // Ранг процесса-получателя
  int          tag, // Тег для идентификации сообщений
  MPI_Comm     comm // Указатель (дескриптор) на коммуникатор
);
```

Метод завершится, когда буфер для отправки данных снова можно будет безопасно использовать.

А это синтаксис функции MPU_Recv, которая будет получать данные от другого процессора:

```
int MPIAPI MPI_Recv(
  _In_opt_ void    *buf,
  int              count,
  MPI_Datatype     datatype,
  int              source,
  int              tag,
  MPI_Comm         comm,
  _Out_ MPI_Status *status // Возвращает MPI_SUCCESS или код ошибки
);
```

Этот метод завершится только после получения данных в буфер.

Ниже представлен пример использования функций отправки и получения:

```
#include “mpi.h”
#include <iostream>
int main(int argc, char * argv[]) {
  int rank, buffer;
  MPI::Init(argv, argc);
  rank = MPI::COMM_WORLD.Get_rank();
  // Процесс 0 отправляет данные из buffer
  // Процесс 1 получает данные в buffer
  if (rank == 0) {
    buffer = 999999;
    MPI::COMM_WORLD.Send( & buffer, 1, MPI::INT, 1, 0);
  } else if (rank == 1) {
    MPI::COMM_WORLD.Recv( & buffer, 1, MPI::INT, 0, 0);
    std::cout << “Data Received“ << buf << “\n”;
  }
  MPI::Finalize();
  return 0;
}
```

При работе через MPI программа отправляет данные, которые затем получит функция приема в другом процессоре.

Коллективы

Коллективы (collectives) – это метод связи, который задействует все процессоры в сети. Двумя основными методами коллективов являются:

- `MPI_BCAST`: **распределяет** данные от одного (корневого) процесса к другому процессору в сети;
- `MPI_REDUCE`: **объединяет** данные всех процессоров в сети и возвращает их в корневой процесс.

Теперь, со знанием о коллективах, мы подошли к концу этой главы и к концу всей книги.

Пришло время посмотреть, чему мы научились!

Выводы

В этой главе мы обсудили реализации управления распределенной памятью, узнали о ее моделях (общая память и процессоры распределенной памяти) и поговорили об этих реализациях. В конце главы мы познакомились с MPI и его использованием. Мы также обсудили коммуникационные сети и различные варианты их разработки для эффективной работы. Теперь вы сможете легко разбираться в топологиях сетей, алгоритмах маршрутизации, стратегиях коммутации и управлении потоками.

В этой книге мы рассмотрели различные конструкции программирования .NET Core 3.1 для реализации параллельного программирования. При правильном использовании параллельное программирование может значительно повысить производительность и отзывчивость приложений. Новые функции и синтаксисы, доступные в .NET Core 3.1, действительно облегчают написание/отладку и поддержку параллельного кода. Для сравнения мы с вами рассмотрели создание многопоточного кода еще до появления TPL.

С помощью новых конструкций асинхронного программирования (async и await) мы научились по максимуму использовать преимущества неблокирующего ввода-вывода при последовательном выполнении программы. Затем мы поговорили о новых функциях, а именно об асинхронных потоках и основных методах асинхронности, облегчающих создание асинхронного кода. Мы также поговорили о поддержке параллельных инструментов в Visual Studio, способствующих отладке кода. И наконец, мы рассмотрели создание модульных тестов для параллельного кода, повышающих его устойчивость к ошибкам.

В завершение мы познакомили вас с методами распределенного программирования и их использованием в .NET Core.

Вопросы

1. ___________ – это расположение нескольких процессоров в основном в отдельных контейнерах, но иногда и в нескольких контейнерах, в непосредственной близости друг от друга.
2. В динамической коммуникационной сети любой узел может отправлять данные на любой другой узел.
    1. Верно
    2. Неверно
3. Что из перечисленного является характеристиками сети связи?
    1. Топология
    2. Стратегия переключения
    3. Управление потоком
    4. Общая память
4. В модели распределенной памяти общее пространство памяти разделяется между процессорами.
    1. Верно
    2. Неверно
5. Коммутация цепей может использоваться в асинхронных сценариях.
    1. Верно
    2. Неверно

Ответы на вопросы

Глава 1

1. 2
2. 2
3. 2
4. 2
5. 2

Глава 3

1. 2
2. 1
3. 2
4. 2
5. 2

Глава 4

1. 2
2. 1
3. 2
4. 2
5. 1

Глава 5

1. 3
2. 4
3. 3
4. 1
5. 1

Глава 6

1. 4
2. 1
3. 1
4. 4

Глава 7

1. 2
2. 1
3. 2
4. 3

Глава 8

1. 1
2. 1, 2, 3
3. 1, 2
4. 1

Глава 9

1. 2
2. 1, 2, 3
3. 1
4. 1
5. 1
6. 2

Глава 10

1. 3
2. 1
3. 2
4. 2
5. 3

Глава 11

1. 1
2. 2
3. 1
4. 3
5. 2

Глава 12

1. 1
2. 1
3. 4
4. 1

Глава 13

1. 1
2. 2
3. 3
4. 2

Глава 14

1. Параллельные системы
2. 2
3. 4
4. 2

Предметный указатель

Латиница
async, 185
AutoResetEvent, 128
await, 185
BackgroundWorker, 38
Common Language Runtime, 36
ConcurrentDictionary<TKey,TValue>, 154
Hyper-threading, 25
IProducerConsumerCollection<T>, интерфейс, 145
Kestrel, многопоточность, 231
Lock, 120
ManualResetEvent, 129
Mutex, 123
Semaphore, 124
Slim locks, 134
SpinLock, 141
SpinWait, 140

А
Асинхронные делегаты, 189
Асинхронный код, 174
  обработка исключений, 190
  оценка производительности, 196
  рекомендации по написанию, 198
Асинхронный шаблон
  на основе задач, 189
  на основе событий, 186

Б
Барьер памяти, 116
Блокировка, 151

В
Визуализатор параллелизма, 211

Д
Делегат обратного вызова, 57

З
Задачи, 46
  обработка исключений, 61
    задач с помощью обратного вызова, 63
    из нескольких задач, 62
    из одиночных задач, 62
  ожидание выполнения, 58
  отмена, 55
  родительские и дочерние, 70
  цепочки, 67

И
Интерфейс передачи сообщений Microsoft (MPI), 264

К
Класс
  BlockingCollection<T>, 151
  EventWaitHandle, 128
  ParallelEnumerable, 94
  ReaderWriterLock, 126
  System.Lazy<T>, 160
  System.Threading.Tasks.Task, 47
  Thread, 31
  ThreadPool, 35
  WaitHandles, 131
Классификация Флинна, 26
Классическая работа с потоками, 23
Коллективы, 267
Коммуникационные сети, 258
  свойства, 259
Критическая секция, 113

Л
Локальная переменная блока данных, 90
Локальная переменная потока, 89
Лямбда-выражения, 189

М
Метод
  AsParallel(), 97
  AsSequential(), 104
  AsUnOrdered(), 97
  CancellationToken, 87
  Parallel.Break, 85
  Parallel.For, 78
  Parallel.ForEach, 79
  Parallel.Invoke, 76
  ParallelLoopState.Stop, 86
  System.Threading.Tasks.Task.Delay, 49
  System.Threading.Tasks.Task.Factory.StartNew, 48
  System.Threading.Tasks.Task.FromCanceled, 53
  System.Threading.Tasks.Task.FromCanceled<T>, 53
  System.Threading.Tasks.Task.FromExceptio, 53
  System.Threading.Tasks.Task.FromException<T>, 53
  System.Threading.Tasks.Task.

FromResult<T>, 52
System.Threading.Tasks.Task.Run, 49
System.Threading.Tasks.Task.Yield, 50
Task.ContinueWith, 68
Task.Factory.ContinueWhenAll, 69
Task.Factory.ContinueWhenAll<T>, 69
Task.Factory.ContinueWhenAny, 69
Task.Factory.ContinueWhenAny<T>, 69
Task.Wait, 59
Task.WaitAll, 59
Task.WaitAny, 60
Task.WhenAll, 60
Task.WhenAny, 61
Микросервисы, 233
Многозадачность, 25
Многопоточность, 23
Многопоточность в IIS, 229
Многопоточный апартамент, 198
Модель
общей памяти, 256
распределенной памяти, 257
Модель компонентных объектов, 28
Модель последовательной согласованности, 115
Модульное тестирование, 217
Модульные тесты, 221
Монитор, 113
Мьютекс, 120

О

Ограничение, 151
Однопотоковый апартамент, 198
Операции ввода-вывода, 38
Отладка
потоков, 204
при помощи окон параллельных стеков, 207
с использованием окна контроля параллельных данных, 209
с VS 2019, 204
Отложенная инициализация, 157
обработка исключений, 163
с локальным хранилищем потоков, 166
сокращение издержек, 168

П

Параллелизм, 31
принудительный, 108
степень, 80
Параллельные стеки, 206
Параметр
AutoBuffered, 99
FullyBuffered, 100
NotBuffered, 98
Перехват работы, 72
Планировщик, 76
Планировщик потоков, 30
Поток, 27
дочерний, 34
основной, 32
переднего плана, 28
фоновый, 28
Преобразование
шаблонов APM в задачи, 64
EAP в задачи, 66
Примитив
блокировки, 118
сигнальный, 126
Barrier, 137
CountDownEvent, 137
синхронизации, 113
легковесный, 134
ManualResetEventSlim, 137
ReaderWriterLockSlim, 135
SemaphoreSlim, 136
Thread.Join, 126
Процесс, 24
Прямой доступ к памяти, 177
Пул потоков CLR, 229

Р

Разделение данных, 82
Разделитель, 76
Распределенные системы, 255

С

Семафор, 120
Синхронный код, 174
Создание потока (пример), 32

Т

Топология, 260

Ш

Шаблон параллельного программирования
агрегация, 246
отложенной инициализации, 249
разделения/объединения, 248
разделяемого состояния, 252
спекулятивной обработки, 248
MapReduce, 243

Книги издательства «ДМК ПРЕСС»
можно купить оптом и в розницу
в книготорговой компании «Галактика»
(представляет интересы издательств
«ДМК ПРЕСС», «СОЛОН ПРЕСС», «КТК Галактика»).
Адрес: г. Москва, пр. Андропова, 38;
тел.: **(499) 782-38-89**, электронная почта: **books@alians-kniga.ru**.
При оформлении заказа следует указать адрес (полностью),
по которому должны быть высланы книги;
фамилию, имя и отчество получателя.
Желательно также указать свой телефон и электронный адрес.
Эти книги вы можете заказать и в интернет-магазине: **http://www.galaktika-dmk.com/**.

Шакти Танвар

Параллельное программирование на C# и .NET Core

Главный редактор *Мовчан Д. А.*
dmkpress@gmail.com
Зам. главного редактора *Сенченкова Е. А.*
Редактор *Черников В. Н.*
Перевод *Воронина А. Д.*
Корректор *Синяева Г. И.*
Верстка *Чаннова А. А.*
Дизайн обложки *Мовчан А. Г.*

Гарнитура PT Serif. Печать цифровая.
Усл. печ. л. 19,53. Тираж 200 экз.

Веб-сайт издательства: **www.dmkpress.com**

Создание надежного корпоративного программного обеспечения с использованием параллелизма и многопоточности

Современные компьютеры работают на процессорах с несколькими ядрами. Однако, если при создании программы не использовалось параллельное программирование, то она не сможет в полной мере задействовать ресурсы оборудования. Из этой книги вы узнаете, как создавать современное программное обеспечение на высокопроизводительной платформе .NET Core с использованием C#.

В этой книге рассказано о создании многопоточных, параллельных и оптимизированных приложений, использующих мощность многоядерных процессоров. Познакомившись с основами многопоточности и параллельности, вы перейдете к структурам данных в .NET Core, поддерживающих параллелизм.

Прочитав книгу, вы освоите асинхронное программирование на C#, сможете диагностировать и успешно устранять ошибки в параллельном коде. Также овладеете навыками работы с новым сервером Kestrel и сможете различать модели работы IIS и Kestrel. Кроме того, познакомитесь с передовым опытом по применению разработки на основе тестирования и научитесь запускать модульные тесты для параллельного кода.

К концу чтения вы будете хорошо разбираться в основных понятиях параллельности и асинхронности, используемых для создания отзывчивых приложений.

Вы научитесь:

- анализировать задачи по распараллеливанию кода;
- работать с современными шаблонами разработки, а также преобразовывать унаследованный программный код с помощью задач (Tasks);
- применять методы работы с большими объемами данных;
- создавать очереди PLINQ и исследовать ограничения, которые влияют на их производительность;
- решать проблемы в параллельных программах, вызванные ситуациями гонки производитель-потребитель;
- подбирать нужные примитивы синхронизации .NET Core;
- понимать принцип работы многопоточности в IIS и Kestrel;
- использовать ресурсы сервера по максимуму.

Интернет-магазин:
www.dmkpress.com

Оптовая продажа:
КТК «Галактика»
books@alians-kniga.ru

Packt>

ДМК ПРЕСС издательство
www.дмк.рф

ISBN 978-5-97060-851-7