

C# Concurrency

Нир Добовицки

Асинхронное программирование и многопоточность



C# Concurrency

ASYNCHRONOUS AND MULTITHREADED
PROGRAMMING

NIR DOBOVIZKI



MANNING
SHELTER ISLAND

C# Concurrency

АСИНХРОННОЕ ПРОГРАММИРОВАНИЕ
И МНОГОПОТОЧНОСТЬ

Нир Добовицки



Санкт-Петербург · Москва · Минск

2025

ББК 32.973.2-018.1

УДК 004.438

Д55

Добавицки Нир

Д55 C# Concurrency. Асинхронное программирование и многопоточность. — СПб.: Питер, 2025. — 272 с.: ил. — (Серия «Библиотека программиста»).
ISBN 978-5-4461-4371-9

Асинхронные и многопоточные программы могут выполнять несколько задач одновременно, не теряя скорости или надежности. Но правильная организация параллельного выполнения может вызвать затруднения даже у опытных разработчиков. Эта практическая книга научит вас создавать параллельные приложения на C#, работающие с максимальной скоростью и не имеющие взаимоблокировок и других проблем синхронизации, которые ухудшают производительность и требуют огромных усилий для их обнаружения.

16+ (В соответствии с Федеральным законом от 29 декабря 2010 г. № 436-ФЗ.)

ББК 32.973.2-018.1

УДК 004.438

Права на издание получены по соглашению с Manning Publications. Все права защищены. Никакая часть данной книги не может быть воспроизведена в какой бы то ни было форме без письменного разрешения владельцев авторских прав.

Информация, содержащаяся в данной книге, получена из источников, рассматриваемых издательством как надежные. Тем не менее, имея в виду возможные человеческие или технические ошибки, издательство не может гарантировать абсолютную точность и полноту приводимых сведений и не несет ответственности за возможные ошибки, связанные с использованием книги. В книге возможны упоминания организаций, деятельность которых запрещена на территории Российской Федерации, таких как Meta Platforms Inc., Facebook, Instagram и др. Издательство не несет ответственности за доступность материалов, ссылки на которые вы можете найти в этой книге. На момент подготовки книги к изданию все ссылки на интернет-ресурсы были действующими.

ISBN 9781633438651 англ.

Authorized translation of the English edition © 2025 Manning Publications.

This translation is published and sold by permission of Manning Publications, the owner of all rights to publish and sell the same.

ISBN 978-5-4461-4371-9

© Перевод на русский язык ООО «Прогресс книга», 2025

© Издание на русском языке, оформление ООО «Прогресс книга», 2025

2025

© Серия «Библиотека программиста», 2025

Краткое содержание

Часть I

Основы асинхронного программирования и многопоточности

Глава 1. Асинхронное программирование и многопоточность	22
Глава 2. Компилятор переписывает ваш код	34
Глава 3. Ключевые слова async и await	45
Глава 4. Основы многопоточности	65
Глава 5. async/await и многопоточность.....	88
Глава 6. Когда использовать async/await	101
Глава 7. Классические ловушки многопоточности и как их избежать	115

Часть II

Продвинутое использование async/await и многопоточности

Глава 8. Обработка последовательности элементов в фоновом режиме	140
Глава 9. Отмена фоновых заданий	160
Глава 10. Ожидаем собственные события.....	174
Глава 11. Выбор потока для выполнения асинхронного кода	188
Глава 12. async/await и исключения	211
Глава 13. Потокобезопасные коллекции	219
Глава 14. Асинхронная генерация коллекций / await foreach и IAsyncEnumerable.....	255

Оглавление

Предисловие	12
Благодарности.....	14
О книге.....	16
Кому адресована эта книга	16
Структура издания.....	16
О примерах программного кода.....	17
От издательства.....	18
Об авторе.....	19
Иллюстрация на обложке	20
Вступительное слово от сообщества DotNetRu	21

Часть I **Основы асинхронного программирования и многопоточности**

Глава 1. Асинхронное программирование и многопоточность	24
1.1. Что такое многопоточность.....	25
1.2. Многоядерные процессоры.....	28
1.3. Асинхронное программирование	31
1.4. Совместное использование многопоточности и асинхронного программирования.....	33
1.5. Эффективность программного обеспечения и облачные вычисления.....	34
Итоги главы.....	35
Глава 2. Компилятор переписывает ваш код	36
2.1. Лямбда-функции	37
2.2. Инструкция yield return	39
Итоги главы.....	44

Глава 3. Ключевые слова async и await	45
3.1. Сложность асинхронного кода.....	46
3.2. Знакомство с Task и Task<T>.....	48
3.2.1. Мы уже приехали?	50
3.2.2. Разбуди меня, когда приедем	51
3.2.3. Синхронный вариант	52
3.2.4. После завершения задачи	53
3.3. Как работают async/await	55
3.4. Асинхронные методы void	60
3.5. ValueTask и ValueTask<T>	62
3.6. А что насчет многопоточности?.....	63
Итоги главы.....	63
Глава 4. Основы многопоточности	65
4.1. Способы запуска кода в другом потоке.....	66
4.1.1. Thread.Start.....	66
4.1.2. Пул потоков.....	70
4.1.3. Task.Run.....	72
4.2. Доступ к одним и тем же переменным из нескольких потоков.....	75
4.2.1. Отказ от общих данных.....	77
4.2.2. Неизменяемые общие данные.....	77
4.2.3. Использование блокировок и мьютексов	78
4.2.4. Взаимоблокировки	80
4.3. Замечания о приложениях с графическим пользовательским интерфейсом.....	81
4.4. Ожидание другого потока.....	82
4.5. Другие методы синхронизации.....	83
4.6. Параметры потока	84
4.6.1. Режим выполнения потока в фоне.....	85
4.6.2. Язык и локаль	85
4.6.3. СОМ-апартаменты	85
4.6.4. Текущий пользователь	86
4.6.5. Приоритет потока.....	86
Итоги главы.....	86
Глава 5. async/await и многопоточность.....	88
5.1. Асинхронное программирование и многопоточность	89
5.2. Где выполняется код после вызова await	92
5.3. Блокировки и async/await	95
5.4. Потоки пользовательского интерфейса.....	98
Итоги главы.....	100

8 Оглавление

Глава 6. Когда использовать async/await	101
6.1. Преимущества асинхронности на серверах.....	102
6.2. Преимущества асинхронности в нативных клиентских приложениях....	108
6.3. Недостатки async/await.....	109
6.3.1. Асинхронное программирование заразно.....	109
6.3.2. Асинхронное программирование имеет больше пограничных случаев.....	111
6.3.3. Многопоточность имеет еще больше пограничных случаев.....	112
6.3.4. async/await – это дорого.....	112
6.4. Когда использовать async/await	113
Итоги главы.....	114
Глава 7. Классические ловушки многопоточности и как их избежать	115
7.1. Частичное обновление	116
7.2. Изменение порядка доступа к памяти при компиляции	119
7.3. Взаимоблокировки.....	123
7.4. Состояние гонки.....	130
7.5. Синхронизация.....	133
7.6. Голодание	135
Итоги главы.....	138

Часть II

Продвинутое использование async/await и многопоточности

Глава 8. Обработка последовательности элементов в фоновом режиме	140
8.1. Параллельная обработка элементов.....	141
8.1.1. Параллельная обработка элементов с помощью класса Thread.....	142
8.1.2. Параллельная обработка элементов с использованием пула потоков.....	144
8.1.3. Асинхронная параллельная обработка элементов.....	146
8.1.4. Класс Parallel.....	148
8.2. Последовательная обработка элементов в фоновом режиме	152
8.2.1. Последовательная обработка элементов в фоновом режиме с помощью класса Thread	152
8.2.2. Паттерн очереди заданий и BlockingCollection	154
8.2.3. Обработка важных элементов с помощью постоянных очередей	157
Итоги главы.....	158

Глава 9. Отмена фоновых заданий	160
9.1. Введение в CancellationToken	160
9.2. Отмена с использованием исключения	168
9.3. Получение обратного вызова, когда вызывающий код отменяет операцию	169
9.4. Реализация тайм-аутов	170
9.5. Комбинирование методов отмены	171
9.6. Специальные токены отмены	172
Итоги главы	172
Глава 10. Ожидаем собственные события	174
10.1. Знакомство с TaskCompletionSource	175
10.2. Выбор потока для запуска продолжения	179
10.3. Пример: ожидание инициализации	180
10.4. Пример: адаптация старых API	182
10.5. Асинхронные операции в старом стиле (BeginXXX, EndXXX)	183
10.6. Пример: асинхронные структуры данных	184
Итоги главы	187
Глава 11. Выбор потока для выполнения асинхронного кода	188
11.1. Поведение await в многопоточной среде	189
11.1.1. await в потоках пользовательского интерфейса	190
11.1.2. await в потоках, отличных от потока пользовательского интерфейса	191
11.2. Контексты синхронизации	193
11.3. Переключение потоков – ConfigureAwait(false)	197
11.4. Дополнительные параметры ConfigureAwait	204
11.5. Task.Yield: возможность выполнить другой код	205
11.6. Планировщики задач	207
Итоги главы	209
Глава 12. async/await и исключения	211
12.1. Исключения и асинхронный код	211
12.2. await и AggregateException	215
12.3. Потеря исключений	216
12.4. Исключения и методы async void	217
Итоги главы	217

10 Оглавление

Глава 13. Потокобезопасные коллекции	219
13.1. Проблемы использования обычных коллекций	220
13.2. Потокобезопасные коллекции	225
13.2.1. ConcurrentDictionary<TKey, TValue>	225
13.2.2. BlockingCollection<T>.....	229
13.2.3. Асинхронные альтернативы BlockingCollection.....	232
13.2.4. ConcurrentQueue<T> и ConcurrentStack<T>.....	233
13.2.5. ConcurrentBag<T>	234
13.2.6. Когда следует использовать потокобезопасные коллекции	235
13.2.7. Когда не следует использовать потокобезопасные коллекции	235
13.3. Неизменяемые коллекции.....	235
13.3.1. Как работают неизменяемые коллекции	236
13.3.2. Как использовать неизменяемые коллекции	242
13.3.3. ImmutableList.....	244
13.3.4. ImmutableDictionary<TKey,TValue>	245
13.3.5. ImmutableHashSet<T> и ImmutableSortedSet<T>.....	246
13.3.6. ImmutableList<T>	247
13.3.7. ImmutableQueue<T> и ImmutableStack<T>	248
13.3.8. ImmutableList<T>	248
13.3.9. Когда использовать неизменяемые коллекции	250
13.4. Замороженные коллекции	250
13.4.1. Когда использовать замороженные коллекции.....	252
Итоги главы.....	252
Глава 14. Асинхронная генерация коллекций / await foreach и IAsyncEnumerable.....	255
14.1. Итерации по асинхронной коллекции	256
14.2. Создание асинхронной коллекции.....	258
14.3. Отмена итерации по асинхронной коллекции.....	261
14.4. Другие варианты.....	264
14.5. IAsyncEnumerable<T> и LINQ.....	264
14.6. Пример: итерации по данным, получаемым асинхронно	265
14.7. Пример: асинхронная очередь, подобная BlockingCollection<T>	266
Итоги главы	269

Посвящается моей замечательной супруге Аналии.

Предисловие

Я занимаюсь разработкой ПО уже более 30 лет: с конца 1990-х годов специализировался на создании высокопроизводительных серверов с применением многопоточности и асинхронного программирования, а с 2003 года начал программировать на C#. Последние десять с небольшим лет я работал консультантом, приходя в проект на непродолжительный период времени и помогая решить конкретную проблему. За это десятилетие мне посчастливилось посетить множество компаний и удалось оказать помощь в разработке многих проектов.

Конечно, каждый проект уникален, и каждая компания изобретает свою инновационную, неповторимую и прорывную технологию, но, поработав со множеством проектов, вы начинаете видеть некоторые сходства. Так и я, посещая разные компании, снова и снова наблюдал одни и те же проблемы, обусловленные неправильным использованием многопоточного и асинхронного программирования.

Многопоточность — это простая концепция: она подразумевает одновременное выполнение нескольких задач. Известно, что правильно реализовать ее очень сложно, но, несмотря на эти трудности, она уже давно широко используется. Разработчики вроде вас, которые уделяют время изучению многопоточности по книгам, могут эффективно ее использовать.

Асинхронное программирование существует с момента изобретения микропроцессора и уже давно используется в высокопроизводительных серверах. Однако оно получило особенно большую популярность среди рядовых разработчиков, когда в 2012 году в C# добавили поддержку операторов `async/await`. (В JavaScript она появилась намного раньше, но в ограниченном виде.) Основываясь на наблюдениях за различными проектами и опыте собеседований при приеме на работу, я обнаружил, что лишь немногие понимают, как работают `async/await`.

Проблемы, возникающие из-за недостатка знаний в области многопоточности и асинхронного программирования, вполне очевидны. Всего за месяц до того, как начались переговоры с издательством Manning о публикации этой книги, я провел курсы по многопоточности и асинхронному программированию в трех разных компаниях.

Вот так и родилась идея этой книги. А затем последовали два года очень глубокого погружения в многопоточное и асинхронное программирование на C#. За это время я многому научился. И уверяю вас, что нет лучшего способа чему-то научиться, чем преподавание, и я надеюсь, что чтение этой книги будет для вас столь же полезным, как для меня ее написание.

Благодарности

Я искренне верю, что это очень хорошая книга, но я писал ее не один. Работа над книгой — это командная работа, требующая участия многих людей. Без их помощи эта книга едва ли получилась бы такой хорошей и, скорее всего, ее бы вообще не было.

Прежде всего хочу поблагодарить моего редактора по развитию в Manning Дуга Раддера (Doug Rudder), у которого хватило терпения научить меня, начинающего автора, писать технические книги. Помощник издателя Майк Стивенс (Mike Stephens), заинтересовавшийся идеей публикации моей книги, помог с поддержкой и получением отзывов. Использование аналогии с приготовлением блюд в первой главе — его идея. Технический редактор Пол Гребенц (Paul Grebenc) был первой линией обороны от технических ошибок. Пол — главный разработчик ПО в OpenText. Он имеет опыт профессиональной разработки ПО более 25 лет и в основном программирует на C# и Java. Его основные интересы — системы, использующие многопоточность, асинхронное программирование и сетевые взаимодействия.

Я также хочу поблагодарить всех рецензентов, вычитывавших черновики этой книги, и всех, кто оставил отзывы, пока книга находилась в программе раннего доступа MEAP: ваши комментарии были бесценны и помогли значительно улучшить книгу. Спасибо всем вам: Альдо Биондо (Aldo Biondo), Александр Сантос Коста (Alexandre Santos Costa), Аллан Табилог (Allan Tabilog), Амрах Умудлу (Amrah U mudlu), Андрий Стосик (Andriy Stosyk), Барри Уоллис (Barry Wallis), Крисс Барнارد (Chriss Barnard), Дэвид Пакку (David Paccoud), Дастин Мецгар (Dustin Metzgar), Герт Ван Летем (Geert Van Laethem), Джейсон Даун (Jason Down), Джейсон Хейлз (Jason Hales), Жан-Поль Малерб (Jean-Paul Malherbe), Джекф Шергалис (Jeff Shergalis), Джереми Кейни (Jeremy Caney), Джим Уэлч (Jim Welch), Иржи Чинчура (Jiří Činčura), Джо Куэвас (Joe Cuevas), Джонатан Блэр (Jonathan Blair), Йорт Роденбург (Jort Rodenburg), Хоце Антонио Мартинес (Jose Antonio Martinez), Жюльен Пои (Julien Pohie), Кришна Чайтанья Анипинди

(Krishna Chaitanya Anipindi), Marek Petak (Marek Petak), Mark Элстон (Mark Elston), Markus Wolff (Markus Wolff), Mikkel Arentoft (Mikkel Arentoft), Milorad Imbra (Milorad Imbra), Oliver Korten (Oliver Korten), Onofreй Джордж (Onofreй George), Sachin Handikar (Sachin Handikar), Simon Seyag (Simon Seyag), Stefan Turalski (Stefan Turalski), Sumit Singh (Sumit Singh) и Vincent Delcoigne (Vincent Delcoigne) — ваши предложения помогли сделать эту книгу лучше.

Я также хочу выразить свою личную благодарность всем, кто приобрел эту книгу по программе раннего доступа. Радость от того, что люди достаточно заинтересованы, чтобы потратить свои честно заработанные деньги на приобретение написанной мной книги, — это прекрасное чувство, которое стало одним из важнейших мотивов начатое до конца.

И наконец, самое важное: я хочу поблагодарить свою семью и особенно жену, мирившуюся со всеми моими глупыми выходками в целом и с тем, что большую часть свободного времени я проводил в своем офисе за написанием книги, в частности.

О книге

Цель этой книги — помочь разработчикам на C# писать безопасный и эффективный многопоточный и асинхронный код приложений. Она фокусируется на практических приемах и особенностях, с которыми вы наверняка столкнетесь в своей повседневной практике.

В книге подробно описывается все, что вам нужно знать, чтобы писать и отлаживать многопоточный и асинхронный код. В ней вы не найдете экзотические и замысловатые приемы, применимые, например, только когда нужно создать свой сервер базы данных, которые слишком сложны для кода обычного приложения и способны создать проблемы, если использовать их в обычном коде, потому что многопоточность и без того достаточно сложна.

Кому адресована эта книга

Эта книга адресована всем разработчикам на C#, желающим получить дополнительные знания о многопоточном и асинхронном программировании. Информация в этой книге применима к любой версии .NET, .NET Core и .NET Framework, выпущенной после 2012 года, а также к Windows и Linux (но только для .NET Core и .NET 5 и более поздних версий, потому что более ранние версии не поддерживались в Linux).

Основное внимание в издании уделяется разработке серверного программного кода, но кое-где затрагиваются также вопросы, касающиеся разработки приложений с графическим интерфейсом.

Структура издания

Эта книга состоит из двух частей, которые разбиты на 14 глав.

Часть I охватывает основы многопоточности и `async/await` в C#:

- глава 1 знакомит с понятиями и терминологией многопоточного и асинхронного программирования;

- глава 2 посвящена методам, которые использует компилятор .NET для реализации расширенной функциональности;
- глава 3 – это глубокое погружение в работу `async/await`;
- глава 4 объясняет суть многопоточности;
- глава 5 связывает вместе все, о чем рассказывалось в главах 3 и 4, и демонстрирует связь `async/await` и многопоточности;
- глава 6 рассказывает о том, когда следует использовать операторы `async/await`, потому что сам факт возможности их использования не означает, что они должны использоваться везде;
- глава 7 завершает первую часть описанием распространенных ошибок в многопоточном программировании и, что еще важнее, приемов, помогающих их избежать.

Часть II посвящена практическому использованию информации, которую вы узнали в части I:

- глава 8 рассказывает об обработке данных в фоновом режиме;
- глава 9 посвящена прерыванию обработки в фоновом режиме;
- глава 10 учит создавать сложные асинхронные компоненты, которые делают больше, чем просто объединяют встроенные асинхронные операции;
- в главе 11 обсуждаются продвинутые варианты использования `async/await` и потоков;
- глава 12 поможет устранить проблемы с исключениями в асинхронном коде;
- глава 13 посвящена потокобезопасным коллекциям;
- глава 14 показывает, как создавать классы, работающие подобно асинхронным коллекциям.

О примерах программного кода

Эта книга содержит множество примеров исходного кода как в пронумерованных листингах, так и внутри текста. В обоих случаях исходный код оформлен шрифтом фиксированной ширины, чтобы отделить его от обычного текста. Местами код выделен **жирным шрифтом**, чтобы подчеркнуть изменения по сравнению с предыдущими шагами в главе, например, когда новая функциональность добавляется в существующую строку кода.

Во многих случаях исходный код был переформатирован; мы добавили разрывы строк и переделали отступы, чтобы уместить строки кода по ширине страницы. В редких случаях даже этого было недостаточно, и мы включили в листинги маркеры продолжения строки (➡). Кроме того, мы удалили комментарии из исходного кода, если он подробно описывается в тексте. Примечания в листингах дополняют описание в тексте, помогая выделить важные понятия.

Примеры кода для этой книги можно найти на GitHub по адресу <https://github.com/nirdobovizki/AsynchronousAndMultithreadedProgrammingInCSharp> и на веб-сайте автора по адресу <https://nirdobovizki.com>. Кроме того, код всех примеров доступен на сайте издательства Manning по адресу <https://www.manning.com/books/csharp-concurrency>.

От издательства

Ваши замечания, предложения, вопросы отправляйте по адресу comp@piter.com (издательство «Питер», компьютерная редакция).

Мы будем рады узнать ваше мнение!

На веб-сайте издательства www.piter.com вы найдете подробную информацию о наших книгах.

Об авторе



Нир Добовицки (Nir Dobovizki) — архитектор программного обеспечения и старший консультант. Занимается разработкой параллельных и асинхронных систем (в основном для высокопроизводительных серверов) с конца 1990-х. Хорошо знаком с языками, компилирующимися в машинный код, а с момента появления .NET 1.1 в 2003 году использует .NET и язык C#. Работал с несколькими компаниями в медицинской, оборонной и промышленной отраслях, которым помогал решать проблемы, возникающие из-за неправильного использования многопоточного и асинхронного программирования.

Иллюстрация на обложке

На обложке книги изображена иллюстрация «Татарин из Тобольска» из коллекции Жака Грассе де Сен-Совера, опубликованной в 1788 году. Каждая иллюстрация в коллекции тщательно прорисована и раскрашена вручную.

В те времена по одежде легко было определить место жительства, профессию и социальное положение. Издательство Manning прославляется изобретательность и инициативу компьютерного бизнеса с помощью обложек книг, основанных на богатом разнообразии региональной культуры многовековой давности,озвращенных к жизни иллюстрациями из таких коллекций, как эта.

Вступительное слово от сообщества DotNetRu

Перед вами русскоязычная версия книги, посвященной многопоточности и асинхронности в .NET. Темы, поднятые здесь, подробно раскрывают, почему так важно грамотно использовать механизмы потоков и асинхронности для ускорения ваших программ, а также помогают избежать распространенных ошибок. В нашем мире повсеместной цифровизации и бесконечно растущих нагрузок, когда приложения должны обрабатывать тысячи и миллионы запросов в секунду, эта книга как никогда актуальна.

Адаптация этого труда на русский язык — результат совместной работы команды энтузиастов, которые постарались не только локализовать книгу, но и сохранить техническую точность, избегая двусмысленности в формулировках. Мы стремились преодолеть языковой барьер и сделать информацию доступной для всех русскоязычных разработчиков, так или иначе связанных с .NET.

Книга раскрывает не только TPL, взаимодействие с потоками и `async/await`, но и современные Frozen Collections. Мы постарались максимально сохранить, а где-то и расширить контекст рассматриваемых вопросов, чтобы улучшить ваш читательский опыт.

Надеемся, что эта книга станет для вас ценным ресурсом — будь вы начинающий разработчик в процессе знакомства с concurrency или опытный инженер в поисках глубоких знаний и понимания этой области .NET.

Российское сообщество .NET-разработчиков DotNetRu

22 Вступительное слово

Над переводом работали представители сообщества DotNetRu:

- Игорь Лабутин;
- Сергей Бензенко;
- Евгений Бестфатор;
- Максим Кушнирук;
- Ренат Тазиев;
- Степан Филиппов;
- Дмитрий Павлов;
- Евгений Асташев;
- Рустам Сафин;
- Кирилл Вишневский;
- Константин Игнаков;
- Владимир Майоров;
- Азамат Сулейманов;
- Радмир Тагиров;
- Сергей Мозговой;
- Максим Молоканов;
- Иван Курилов;
- Александр Заозерский;
- Алексей Синютин;
- Василий Иващенко;
- Антон Шевяков;
- Никита Бандурин;
- Андрей Брижак;
- Анатолий Кулаков.

Часть I

Основы асинхронного программирования и многопоточности

Первая часть этой книги охватывает асинхронное программирование и многопоточность на C#, объясняя, что это такое и как правильно их использовать. В этой части освещаются распространенные подводные камни и рассказывается, как их избежать.

Сначала мы рассмотрим понятия и терминологию многопоточности и асинхронного программирования, используемые в информатике в целом и в C# в частности (глава 1). Затем погрузимся в особенности асинхронного программирования с применением `async/await` на C# (главы 2 и 3). Потом обсудим многопоточность в C# (глава 4) и как многопоточность и асинхронное программирование работают вместе (глава 5). Наконец, мы поговорим о том, когда следует использовать `async/await` (глава 6) и как правильно использовать многопоточность (глава 7).

К концу части 1 вы научитесь писать корректный многопоточный код и правильно использовать `async/await`.

1

Асинхронное программирование и многопоточность

В этой главе

- ✓ Введение в многопоточность.
- ✓ Введение в асинхронное программирование.
- ✓ Совместное использование асинхронного программирования и многопоточности.

Как разработчики программного обеспечения мы часто стремимся сделать наши приложения более быстрыми, отзывчивыми и эффективными. Один из способов достичь этого — позволить компьютеру выполнять несколько задач одновременно, максимально используя существующие ядра процессора. Многопоточность и асинхронное программирование — это два подхода, которые обычно применяются для решения этой задачи.

Многопоточность позволяет компьютеру создавать видимость одновременного выполнения нескольких задач, даже когда их количество превышает количество ядер процессора. Асинхронное программирование, напротив, фокусируется на оптимизации использования процессора во время операций, которые обычно блокируют его, что гарантирует эффективное использование вычислительных ресурсов.

Возможность компьютера выполнять несколько задач одновременно чрезвычайно полезна. Она помогает поддерживать высокую отзывчивость приложений во время их работы и играет важную роль в работе высокопроизводительных

серверов, которые благодаря этой возможности получают способность взаимодействовать со множеством клиентов одновременно.

Оба подхода можно использовать для создания отзывчивых клиентских приложений или серверов, работающих с несколькими клиентами. А их сочетание может значительно повысить производительность и позволить серверам обрабатывать тысячи клиентов одновременно.

Эта глава познакомит вас с многопоточностью и асинхронным программированием и покажет, почему они важны. В оставшейся части книги будем говорить о том, как правильно их использовать в .NET и C#, и основное внимание уделим операторам `async/await` в C#. Вы узнаете, как работают эти подходы, научитесь правильно их использовать и обходить распространенные подводные камни.

1.1. Что такое многопоточность

Прежде чем начать рассматривать `async/await`, нам нужно понять, что такое многопоточность и асинхронное программирование. Для этого мы немного поговорим о веб-серверах и приготовлении пиццы. Начнем с пиццы (потому что она вкуснее веб-сервера).

В общих чертах процесс приготовления пиццы в заведениях, продающих еду навынос, обычно выглядит так.

1. Повар получает заказ.
2. Берет готовое тесто, формует его, добавляет соус, сыр и начинку.
3. Ставит пиццу в духовку и ждет, пока она приготовится (это самый долгий этап).
4. Затем повар делает еще несколько дел — достает пиццу из духовки, разрезает ее и кладет в коробку.
5. И наконец, передает пиццу доставщику.

Это не поваренная книга, поэтому наш алгоритм выпечки пиццы — метафора для одного из самых простых видов серверов — веб-сервера, обслуживающего статические файлы. В общих чертах работа простого веб-сервера выглядит так.

1. Сервер получает веб-запрос.
2. Исследует запрос, чтобы выяснить, что нужно сделать.
3. Читает файл (это самый долгий этап).
4. Выполняет дополнительную обработку (например, упаковывает содержимое файла).
5. Отправляет содержимое файла обратно браузеру.

В большей части главы мы будем игнорировать первый и последний шаги, потому что в большинстве веб-фреймворков (включая ASP.NET и ASP.NET Core) они обрабатываются самим фреймворком, а не нашим кодом. Мы кратко поговорим о них ближе к концу этой главы. Схема на рис. 1.1 иллюстрирует процесс обработки веб-запроса.

А теперь вернемся к пицце. В простейшем случае повар будет выполнять шаги по порядку, полностью заканчивая приготовление одной пиццы, прежде чем приступить к следующей. Пока пицца выпекается, повар будет просто стоять и ничего не делать (это полностью синхронная однопоточная версия процесса).

В мире веб-серверов в роли повара выступает центральный процессор. Код этого простого однопоточного веб-сервера последовательно выполняет необходимые операции, обрабатывая веб-запрос, и пока файл читается с диска, процессор простояивает и ничего не делает. В действительности операционная система приостанавливает ожидающий поток и передает процессор другой программе, но с точки зрения нашей программы это выглядит так, будто процессор простояивает.

Такая версия процесса имеет свои преимущества — она простая и легкая для понимания. Вы можете посмотреть на текущий шаг и с полной уверенностью сказать, на каком этапе процесса находитесь. Поскольку никакие два этапа не выполняются одновременно, они не могут мешать друг другу. Наконец, эта версия требует наименьшего объема памяти и потребляет меньше ресурсов, так как в каждый момент времени обрабатывает только один веб-запрос (или готовит только одну пиццу).

В то же время эта однопоточная синхронная версия процесса довольно расточительна, потому что большую часть времени повар/процессор ничего не делает и ждет, пока пицца приготовится в духовке (или файл загрузится с диска), и если наша пиццерия не обанкротится, то в какой-то момент мы начнем получать новые заказы быстрее, чем сможем их выполнять.

По этой причине было бы желательно, чтобы повар мог делать несколько пицц одновременно. Для этого можно использовать таймер, который будет подавать



Рис. 1.1. Синхронная однопоточная обработка запроса

звуковой сигнал каждые несколько секунд. Каждый раз, когда таймер подает сигнал, повар прекращает делать то, что он делал, записывая на бумажке, на чем он остановился, и начинает готовить новую пиццу или продолжает готовить предыдущую (игнорируя пиццы, выпекающиеся в духовке), пока таймер снова не подаст сигнал.

В этой версии повар пытается делать несколько дел одновременно, и каждое из этих дел называется *потоком*. Каждый поток — это последовательность операций, которые могут выполняться параллельно с другими похожими или отличающимися последовательностями.

Этот пример может показаться слишком наивным, потому что он явно неэффективен, и наш повар будет тратить слишком много времени, чтобы записать на бумажке, на чем он остановился, или прочитать с бумажки, чтобы понять, с какого места продолжить работу. Однако именно так работает многопоточность. Внутри процессора есть таймер, который сигнализирует, когда он должен переключиться на следующий поток, и при каждом переключении процессор сохраняет информацию о том, что он делал, и загружает состояние другого потока (это называется *переключением контекста*).

Например, когда код читает файл, поток не может продолжить работу, пока данные не извлечены из файла на диске. До этого момента мы говорим, что поток *заблокирован*. Выделение процессорного времени заблокированному потоку, очевидно, было бы расточительством, поэтому, когда поток начинает читать файл, операционная система переводит его в заблокированное состояние. Войдя в это состояние, поток немедленно освобождает процессор, чтобы им мог воспользоваться другой поток (возможно, в другой программе), и операционная система не будет выделять потоку процессорное время, пока он находится в заблокированном состоянии. Когда система закончит читать файл, она выведет поток из заблокированного состояния и тот снова сможет получать свои кванты процессорного времени.

Операции, которые могут привести к блокировке потока, называются *блокирующими*. Таковыми являются все операции доступа к файлам и сети. Блокирующими также являются все операции, взаимодействующие с чем-либо еще, кроме процессора и памяти, и все операции, ожидающие наступления некоторого события в другом потоке.

Вернемся в нашу пиццерию. В дополнение ко времени, необходимому на переключение между пиццами, повару нужно время, чтобы прочитать информацию и вернуться в то же самое место, где он был до переключения. Каждый поток в наших программах, даже если он не запущен, занимает некоторый объем памяти, поэтому создание очень большого количества потоков, каждый из которых выполняет блокирующую операцию (и большую часть времени пребывает в заблокированном состоянии, не потребляя процессорного времени),

приведет к расточительному расходованию памяти. С увеличением числа потоков программа будет замедляться все больше из-за необходимости тратить все больше времени на управление потоками. В какой-то момент программа начнет тратить так много времени на управление потоками, что на выполнение полезной работы не останется времени или просто закончится память и произойдет сбой.

Даже при всей этой неэффективности многопоточный повар, прыгающий от одной пиццы к другой как сумасшедший, приготовит больше пицц за то же время, если только не окажется перегружен сверх меры и не даст сбой (я знаю, что повар не может сломаться, но это всего лишь метафора). Это объясняется тем, что однопоточный повар до этого большую часть своего времени проводил в ожидании, пока пицца испечется в духовке.

Как показано на рис. 1.2, поскольку у нас в процессоре есть только одно ядро (я знаю, что сейчас практически все компьютеры оснащены многоядерными процессорами, и мы скоро поговорим о них), мы не можем делать два дела одновременно. Все этапы приготовления выполняются один за другим и в действительности это не происходит параллельно, однако процессор может ожидать одновременно сколько угодно задач. Именно поэтому наша многопоточная версия смогла бы обработать три запроса за значительно меньшее время, чем потребовалось бы однопоточной версии для обработки двух.

Если внимательно посмотреть на рис. 1.2, то можно увидеть, что, хотя однопоточная версия обработала первый запрос быстрее, многопоточная версия завершила обработку всех трех до того, как однопоточная версия смогла завершить второй запрос. Этот пример иллюстрирует большое преимущество многопоточности, которое заключается в более эффективном использовании процессора в сценариях, где возможны периоды ожидания, а также показывает ее цену — немного дополнительных накладных расходов на каждом этапе пути.

До сих пор мы говорили об одноядерных процессорах, но все современные процессоры имеют несколько ядер. Как это меняет ситуацию?

1.2. Многоядерные процессоры

Многоядерные процессоры концептуально просты — это лишь группа одноядерных процессоров, размещенных на одном физическом кристалле.

Наличие восьмиядерного процессора эквивалентно наличию восьми поваров в нашей пиццерии. В предыдущем примере у нас был один повар, который мог делать только одно дело за раз, но притворялся, что делает несколько дел, быстро переключаясь между ними. Теперь у нас восемь поваров, каждый из которых может делать одно дело за раз (а все вместе — восемь дел одновременно), и каждый создает видимость, что делает несколько дел, быстро переключаясь между ними.

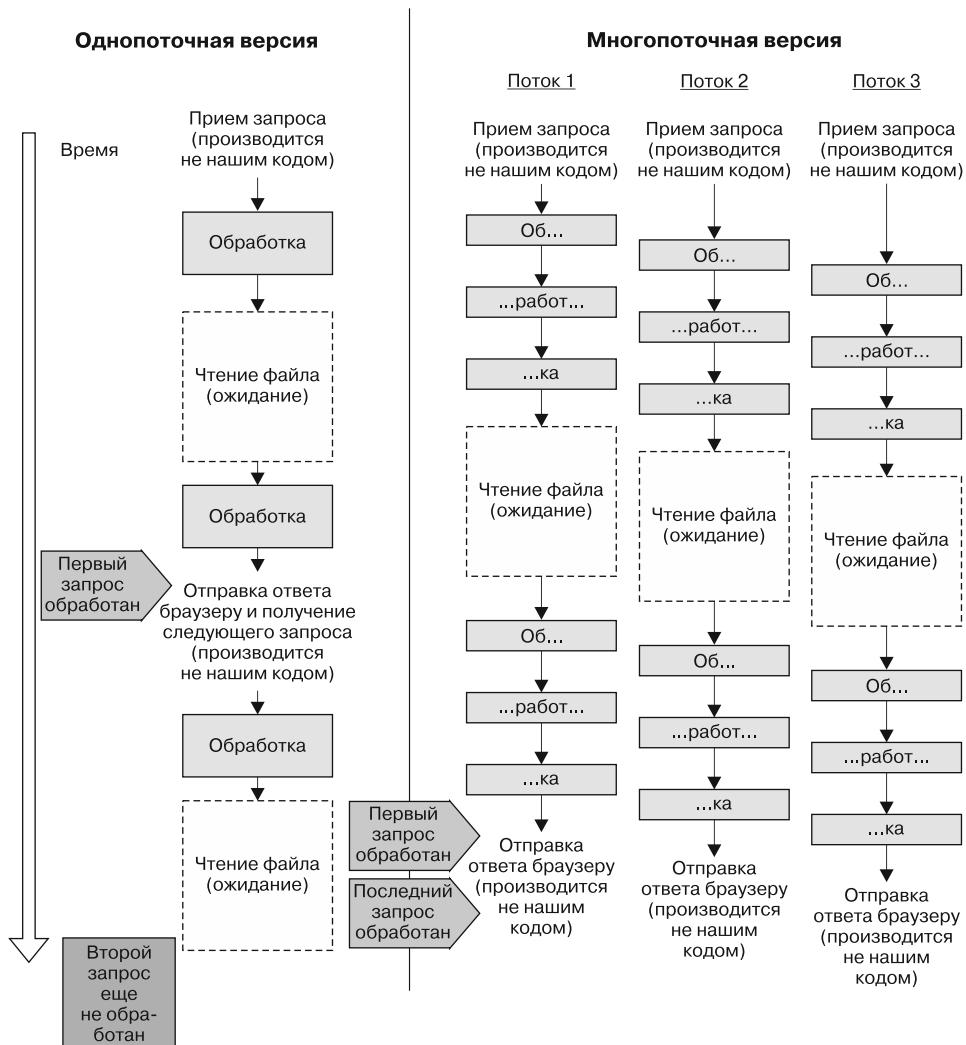


Рис. 1.2. Пример однопоточной и многопоточной обработки нескольких запросов

С точки зрения программного обеспечения при наличии многоядерного процессора, действительно, можно иметь несколько потоков, работающих по-настоящему одновременно. Когда у нас был одноядерный процессор, мы нарезали процесс на крошечные этапы и чередовали их, чтобы казалось, что они выполняются одновременно (хотя на самом деле в каждый конкретный момент времени выполнялся только один этап). Теперь, имея восьмиядерный процессор, мы по-прежнему нарезаем процесс на крошечные этапы и чередуем их, но при этом можем одновременно выполнять восемь таких этапов.

Теоретически восемь поваров способны приготовить больше пиццы, чем один, однако несколько поваров могут непреднамеренно мешать друг другу работать. Например, они могут попытаться одновременно поставить пиццу в духовку или им может понадобиться использовать один и тот же нож для нарезания пиццы — чем больше поваров, тем выше вероятность, что это произойдет.

На рис. 1.3 показано, как выполняется та же многопоточная работа, что и на рис. 1.2, но на этот раз на двухъядерном процессоре (здесь показаны только два ядра, потому что схема с восемью ядрами была бы слишком большой, чтобы уместить ее на книжной странице).

Многопоточная двухъядерная версия

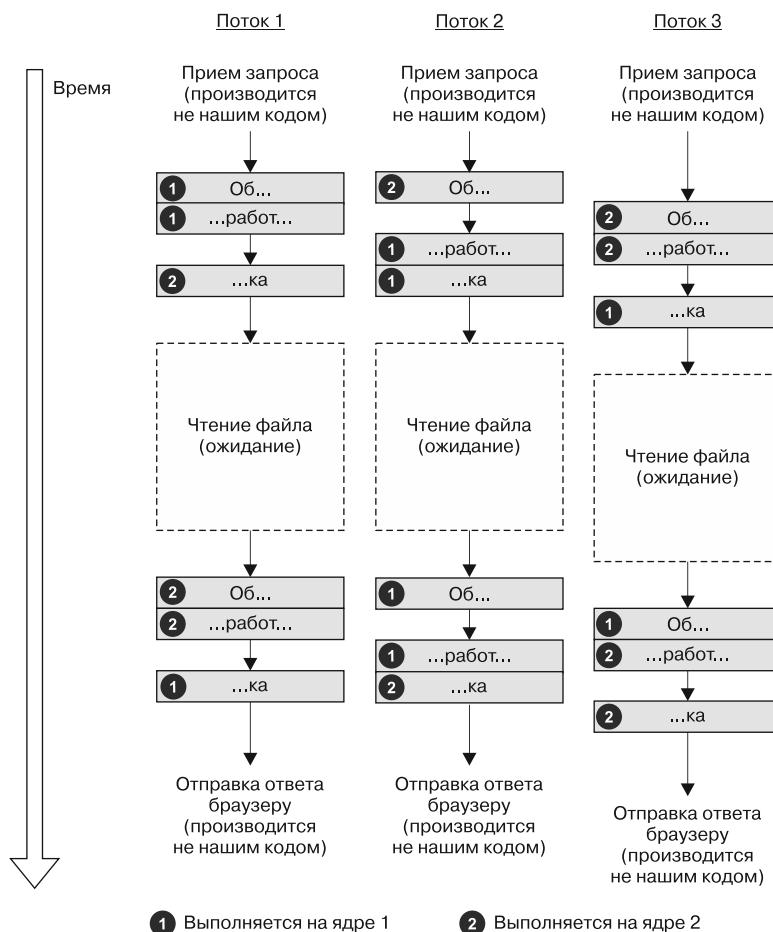


Рис. 1.3. Пример обработки трех запросов на двухъядерном процессоре

Обратите внимание, что по умолчанию между потоками и ядрами нет постоянной связи. Поток может произвольно перемещаться с одного ядра на другое (однако есть возможность привязать потоки для выполнения на определенных ядрах; этот прием так и называется — «привязка потоков», но пользоваться им следует только в особых случаях).

Двухъядерный процессор позволил вдвое сократить время обработки, по сравнению с одноядерной версией, но не повлиял на время, затраченное на ожидание. Так что, хотя мы и получили значительное ускорение, общее время сократилось менее чем вдвое. До сих пор основное улучшение производительности мы получали за счет выполнения других действий, ожидая, пока жесткий диск прочитает файл, но платили за это дополнительными накладными расходами и сложностью многопоточного программирования. Возможно, мы сможем сократить эти накладные расходы.

1.3. Асинхронное программирование

Вернемся в пиццерию и рассмотрим рациональное решение, которое раньше игнорировали: повар готовит одну пиццу, не останавливаясь и не переключаясь на другие, но, поставив пиццу в духовку, он может начать готовить следующую пиццу, а не просто сидеть и ждать. Позже, закончив очередной этап, повар может проверить, готова ли пицца в духовке, и если да, то вынуть ее, нарезать, положить в коробку и передать курьеру.

Это пример асинхронного программирования. Всякий раз, когда процессору нужно сделать что-то, что происходит вне процессора (например, прочитать файл), он отправляет задание внешнему компоненту (например, контроллеру диска) и просит этот компонент уведомить процессор, когда задание будет выполнено.

Асинхронная (или неблокирующая) версия обработки файла просто ставит операцию в очередь с помощью операционной системы (которая затем поставит ее в очередь с помощью контроллера диска) и немедленно возвращает управление, позволяя тому же потоку сделать что-то еще (рис. 1.4). Позже мы можем проверить, была ли операция завершена, и получить доступ к прочитанным данным.

Сравнив все схемы в этой главе, можно заметить, что эта однопоточная и асинхронная версия — самая быстрая из всех. Она завершает обработку первого запроса почти так же быстро, как однопоточная версия, а обработку последнего запроса — почти так же быстро, как двухъядерная многопоточная версия (и при этом не использует второе ядро), что делает ее самой производительной версией на данный момент.

Теперь взгляните на рис. 1.4. Схема на нем выглядит более беспорядочной, чем предыдущие, и ее сложнее читать, и это еще без указания того, что шаги

«последующей обработки» зависят от завершения операций чтения. Сложность чтения этой схемы объясняется тем, что на ней больше не виден весь процесс целиком. Обработка каждого запроса разбита на этапы, и, в отличие от примера с потоками, эти этапы не связаны друг с другом.

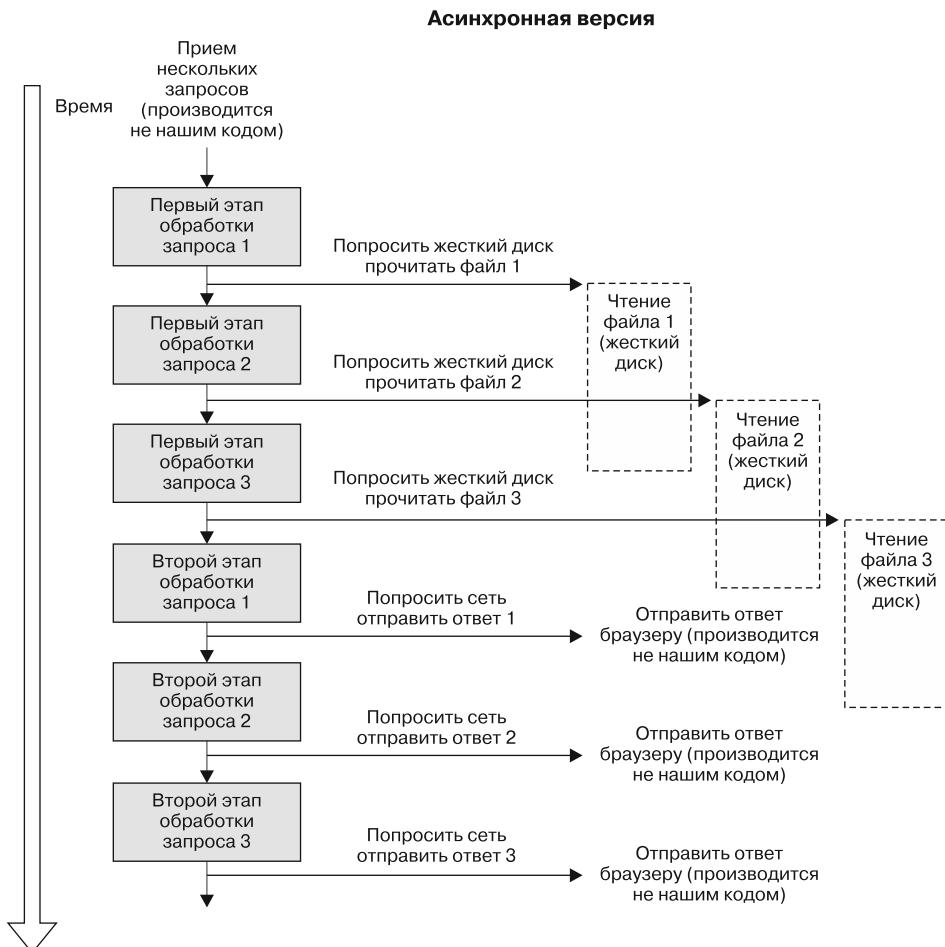


Рис. 1.4. Обработка трех запросов однопоточным асинхронным веб-сервером

Вот почему, несмотря на широкое использование многопоточности, асинхронное программирование до появления `async/await` выбирали только те, кто создавал высокопроизводительные серверы (или использовал среды, где не было другого выбора, например JavaScript). Как показано на рис. 1.4, код приходилось разбивать на части, которые пишутся по отдельности, что затрудняло создание кода

и еще больше — его понимание. Так было до тех пор, пока в C# не появились операторы `async/await`, позволяющие писать асинхронный код так, как если бы это был обычный синхронный код.

На рис. 1.4 также показано, что я использую такие же асинхронные техники как для чтения файла, так и для отправки ответа браузеру. Это потому, что первый и последний этапы в последовательности обработки веб-запроса — «прием веб-запроса» и «отправка ответа браузеру» — выполняются в основном сетевой картой, а не процессором, подобно тому как чтение файла производится жестким диском, и потому они могут выполняться асинхронно, не заставляя процессор ждать.

Даже с использованием одной лишь многопоточности, без применения приемов асинхронного программирования, вполне можно написать серверы, способные обрабатывать низкие и средние нагрузки, запуская отдельный поток для обслуживания каждого соединения. Однако, если вам нужно создать сервер, который может обрабатывать тысячи соединений одновременно, накладные расходы на работу такого количества потоков замедлят сервер до такой степени, что он не сможет обрабатывать запросы или вообще выйдет из строя.

Выше мы говорили об асинхронном программировании как о способе избежать многопоточности, но без многопоточности невозможно использовать всю мощь многоядерных процессоров. Давайте посмотрим, можно ли использовать многопоточность и асинхронное программирование вместе, чтобы добиться еще большей производительности.

1.4. Совместное использование многопоточности и асинхронного программирования

В последний раз вернемся к примеру с пиццерией. Мы можем еще больше усовершенствовать процесс приготовления пиццы: не заставлять повара периодически проверять духовку, а оснастить духовку звуковым сигналом, который будет подаваться по завершении выпекания пиццы. Когда духовка подаст звуковой сигнал, повар сможет прервать свою текущую работу, вынуть пиццу, положить ее в коробку, передать курьеру, а затем вернуться к прерванной работе.

Программный эквивалент — при запуске асинхронной операции попросить операционную систему уведомить программу запуском функции обратного вызова, зарегистрированной при запуске асинхронной операции. Эта функция обратного вызова должна запускаться в новом потоке (на самом деле в потоке из пула; о пулах потоков мы поговорим далее в этой книге), потому что сам вызывающий поток не ждет и в данный момент делает что-то другое. Вот почему асинхронность и многопоточность хорошо работают вместе.

1.5. Эффективность программного обеспечения и облачные вычисления

В настоящее время мы без труда можем использовать «бессерверные» вычисления, предлагаемые многими облачными провайдерами, и запустить 10 000 копий нашего однопоточного кода, работающих в одно и то же время. Так нужно ли нам беспокоиться обо всем этом многопоточном и асинхронном коде?

Теоретически можно легко и просто бросить огромный объем вычислительной мощности на решение задачи. С современными облачными предложениями можно в любой момент получить неограниченную вычислительную мощность, но за нее придется платить. А поскольку плата взимается ровно за тот объем вычислительных ресурсов, который был потреблен, любое увеличение эффективности экономит вам деньги.

В эпоху до облачных вычислений вы бы купили сервер, и пока не выжали бы из него максимум, эффективность кода не имела бы для вас особого значения. На сегодняшний день сокращение времени обработки каждого запроса на доли секунды на высоконагруженном сайте может сэкономить значительную сумму денег.

Раньше процессоры становились все быстрее, сохранялась тенденция, при которой скорость процессора удваивалась каждые два года. Это означало, что вы могли «починить» медленное программное обеспечение, просто немного подождав и купив новый компьютер. К сожалению, ситуация изменилась, потому что современный процессор настолько приблизился к физическому пределу на количество транзисторов, которые можно разместить на кристалле определенной площади, что в принципе стало невозможно существенно ускорить одноядерный процессор. По этим причинам однопоточная производительность процессоров теперь растет довольно медленно, и у нас осталась единственная возможность повышения производительности — использовать больше ядер процессора (есть очень интересная статья под названием *The Free Lunch Is Over* Герба Саттера (Herb Sutter), посвященная этой теме; она доступна по адресу www.gotw.ca/publications/concurrency-ddj.htm).

Тем не менее современный процессор все еще чрезвычайно быстр, быстрее других компьютерных компонентов и, очевидно, намного быстрее любого человека. Поэтому типичный процессор проводит большую часть своего времени в ожидании. Иногда он ждет ввода пользователя, а иногда завершения операции жестким диском, но все равно ждет. Многопоточность и асинхронность позволяют использовать время ожидания для выполнения полезной работы.

Итоги главы

- Многопоточность — это настолько быстрое переключение между несколькими потоками, которые выполняются на одном и том же процессоре, что создается ощущение, будто все они выполняются одновременно.
- Поток — это последовательность команд, которые могут обрабатываться в один и тот же момент времени только одним ядром процессора.
- Поток имеет значительные накладные расходы.
- Переключение между потоками называется *переключением контекста* и тоже имеет накладные расходы.
- При выполнении операций за пределами процессора, например при чтении файла или использовании сети, поток должен дождаться завершения операции, чтобы получить результат и продолжить работу. Такие операции называются *блокирующими*.
- Асинхронное программирование избавляет поток от необходимости ждать завершения блокирующей операции, запрашивая у системы отправку уведомления по ее завершении. Такие операции называются *неблокирующими*. Позже, когда данные станут доступны, программа сможет продолжить обработку, обычно в другом потоке.
- Асинхронные и многопоточные подходы к реализации программ совершенно необходимы, поскольку сложность программного обеспечения растет быстрее, чем однопоточная производительность процессоров.
- Поскольку в облачных вычислениях мы платим только за использованные ресурсы, асинхронные и многопоточные методы вычислений, повышающие эффективность, могут сэкономить нам немного денег.



Компилятор переписывает ваш код

В этой главе

- ✓ Как компилятор C# поддерживает возможности, отсутствующие в среде выполнения .NET.
- ✓ Реализация лямбда-функций компилятором.
- ✓ Реализация `yield return` компилятором.

Компилятор изменяет ваш код, то есть результат компиляции не является прямым представлением исходного кода. Это объясняется двумя основными причинами: необходимостью уменьшить объем шаблонного кода, вводимого руками, и добавить возможности, не поддерживаемые нижележащей платформой. Одной из таких возможностей являются операторы `async/await`, которые реализуются компилятором C#, а не средой выполнения .NET. Чтобы написать правильный асинхронный код, избежать потенциальных ошибок и, что особенно важно, отладить этот код, необходимо понимать, как компилятор преобразует исходный код, то есть что происходит при запуске вашего кода.

В этой главе мы поговорим о том, как компилятор C# переписывает ваш код во время компиляции. Однако, поскольку операторы `async/await`, вероятно, являются самыми сложными преобразованиями в текущей версии C#, мы для начала познакомимся с лямбда-функциями и инструкцией `yield return`, которые реализуются с применением тех же подходов, что и `async/await`. Начав

с более простых возможностей компилятора, мы сможем изучить концепции, лежащие в основе `async/await`, не погружаясь в сложности асинхронного программирования и многопоточности. В следующей главе будет показано, как все это транслируется в `async/await`.

А сейчас посмотрим, как компилятор C# добавляет расширенные возможности, не поддерживаемые нижележащей средой выполнения .NET, начав с лямбда-функций (обратите внимание, что лямбда-функции в C# не имеют ничего общего с сервисом Amazon AWS Lambda).

2.1. Лямбда-функции

Начнем с одной из самых простых возможностей C#, реализуемых компилятором — лямбда-функций. Это блоки кода, записываемые в одну строку внутри более крупного метода, которые можно задействовать как отдельные методы. Лямбда-функции позволяют взять код, который по техническим причинам должен быть другим методом, и записать его в строку там, где он используется, что упрощает чтение и понимание кода. Лямбда-функции также могут использовать локальные переменные, объявленные в методах, где определяются эти функции.

Однако среда выполнения .NET не поддерживает встраиваемые функции — весь код в .NET должен быть в форме методов, являющихся частью классов. Так как же работают лямбда-функции? Рассмотрим очень простой пример: создадим таймер, настроим его вызов через 1 с, а затем выведем в консоль строку "Выполнено" (листинг 2.1).

Листинг 2.1. Использование лямбда-функций

```
public class LambdaDemo1
{
    private System.Timers.Timer? _timer;

    public void InitTimer()
    {
        _timer = new System.Timers.Timer(1000);
        _timer.Elapsed += (sender, args) => Console.WriteLine("Выполнено");
        _timer.Enabled = true;
    }
}
```

Если запустить этот пример, то программа выведет "Выполнено" через 1 с. В этом примере я хочу сосредоточить ваше внимание на строке, которая устанавливает свойство `_timer.Elapsed`. Здесь определяется лямбда-функция и передается в свойство `Elapsed`.

38 Глава 2. Компилятор переписывает ваш код

Выше я сказал, что в .NET весь код должен быть в методах классов, так как же такое возможно? Дело в том, что компилятор C# преобразует лямбда-функцию в обычный метод. Если посмотреть на вывод компилятора, то вы увидите такой код:

```
public class LambdaDemo2
{
    private System.Timers.Timer? _timer;

    private void HiddenMethodForLambda(          ←
        object? sender, System.Timers.ElapsedEventArgs args)
    {
        Console.WriteLine("Выполнено");
    }

    public void InitTimer()
    {
        _timer = new System.Timers.Timer(1000);
        _timer.Elapsed += HiddenMethodForLambda;
        _timer.Enabled = true;
    }
}
```

Лямбда-функция превращается
в обычный метод

Компилятор реорганизовал наш код и переместил тело лямбда-функции в новый метод. То есть мы можем записать код в строку, а среда выполнения будет рассматривать его как обычный метод.

Но лямбда-функция может также использовать локальные переменные, объявленные в методе, где определена лямбда-функция. Добавим в метод `InitTimer` локальную переменную и используем ее внутри лямбда-функции (листинг 2.2).

Листинг 2.2. Лямбда-функция, использующая локальные переменные

```
public class LambdaDemo3
{
    private System.Timers.Timer? _timer;

    public void InitTimer()
    {
        int aVariable = 5;          ←
        _timer = new System.Timers.Timer(1000);
        _timer.Elapsed += (sender,args) => Console.WriteLine(aVariable);
        _timer.Enabled = true;
    }
}
```

Новая
переменная

Если мы попробуем применить к этому коду такие же трансформации, как в предыдущем примере, то снова получим два метода, которые совместно используют локальную переменную. Очевидно, что это не поддерживается и даже не имеет смысла. Но как компилятор справляется с этим? Прежде всего ему нужно что-то, что может хранить данные, к которым можно получить доступ

из двух мест, и в .NET это «что-то» есть: классы. Так что компилятор создает класс для хранения «локальной» переменной:

```
public class LambdaDemo4
{
    private System.Timers.Timer? _timer;
    private class HiddenClassForLambda
    {
        public int aVariable;
        public void HiddenMethodForLambda(
            object? sender,
            System.Timers.ElapsedEventArgs args)
        {
            Console.WriteLine(aVariable);
        }
    }

    public void InitTimer()
    {
        var hiddenObject = new HiddenClassForLambda();
        hiddenObject.aVariable = 5;
        _timer = new System.Timers.Timer(1000);
        _timer.Elapsed += hiddenObject.HiddenMethodForLambda;
        _timer.Enabled = true;
    }
}
```

Здесь компилятор создал новый метод и совершенно новый класс. Локальная переменная была преобразована в член этого нового класса, а метод `InitTimer` и лямбда-функция начали ссылаться на этот член нового класса. В результате меняется способ доступа к локальной переменной, объявленной вне лямбда-функции, — после добавления лямбда-функции некоторые операции, использующие только локальные переменные, могут превратиться в операции, обращающиеся к членам класса. Если в одном и том же методе определить несколько лямбда-функций, все они будут помещены в один класс, чтобы они могли совместно использовать локальные переменные. Важно отметить, что здесь нет никакой магии — компилятор просто добавляет код, который мы можем написать сами, потому что фактически мы имеем такой же доступ к функциональности среды выполнения, как и компилятор.

Теперь, рассмотрев преобразование лямбда-функций в методы, разберем более сложный пример.

2.2. Инструкция `yield return`

В инструкции `yield return` используются те же трюки, что были показаны в примерах с лямбда-функциями, чтобы реализовать нечто более сложное. Она немного похожа на `async/await`, но лишена сложностей, присущих многопоточности и асинхронному коду, поэтому она послужит нам хорошей основой для изучения `async/await`.

40 Глава 2. Компилятор переписывает ваш код

Что же такое `yield return`? По сути, эта инструкция позволяет писать функции, генерирующие последовательность значений, которые можно задействовать в циклах `foreach` напрямую без использования коллекций, таких как списки или массивы. Каждое значение может быть использовано без необходимости генерации полной последовательности. Давайте напишем очень простой метод, возвращающий коллекцию с двумя элементами, числами 1 и 2. Листинг 2.3 показывает, как это выглядит без `yield return`.

Листинг 2.3. Использование списка

```
private IEnumerable<int> NoYieldDemo()
{
    var result = new List<int>();

    result.Add(1);
    result.Add(2);
    return result;
}

public void UseNoYieldDemo()
{
    foreach(var current in NoYieldDemo())
    {
        Console.WriteLine($"Получено {current}");
    }
}
```

Как и следовало ожидать, этот код выведет две строки: Получено 1 и Получено 2. Листинг 2.4 демонстрирует аналогичную реализацию, но с применением `yield return`.

Листинг 2.4. Использование yield return

```
private IEnumerable<int> YieldDemo()
{
    yield return 1;
    yield return 2;
}

public void UseYieldDemo()
{
    foreach(var current in YieldDemo())
    {
        Console.WriteLine($"Получено {current}");
    }
}
```

Код выглядит очень похожим и выдает те же результаты. Так в чем же разница? В первом примере все значения были сначала сгенерированы, а затем использованы, тогда как во втором примере каждое значение генерируется, только когда оно необходимо, как показано на рис. 2.1.



Рис. 2.1. Использование коллекции и инструкции yield return

В версии без `yield return` код работает линейно. Метод `NoYieldDemo` запускается, выполняет какие-то операции и затем возвращает результаты. Однако метод `YieldDemo` ведет себя иначе — при каждом вызове он приостанавливается и возвращает очередной результат, выполняя минимальный объем кода, необходимый для получения очередного значения (до следующей инструкции `yield return`). Но .NET не предусматривает возможности приостанавливать и возобновлять выполнение метода. Что это за волшебство?

Очевидно, что никакого волшебства нет, потому что в информатике нет магии. Как и в случае с лямбда-функциями, которые мы видели выше, компилятор просто переписал наш код.

Код, который может приостанавливаться, возобновляться и потенциально возвращать несколько значений, называется *сопрограммой*¹. В C# методы с `yield return` называются *методами-итераторами*, а методы с `async/await` — *асинхронными методами*. В этой книге мы будем использовать терминологию, принятую в C#. Интерфейс `IEnumerable<T>`, который я использовал в методе `YieldDemo`

¹ Можно также встретить название «корутина», от англ. coroutine. — Примеч. пер.

в качестве типа возвращаемого значения, является базовым для всего, что можно рассматривать как коллекции или последовательности элементов (включая все, что можно использовать в `foreach`). Все обобщенные коллекции в .NET реализуют этот интерфейс (старые классы коллекций, существовавшие до появления обобщенных классов в .NET 2.0, используют необобщенный интерфейс `IEnumerable`). Он определяет только один метод, возвращающий экземпляр `IEnumerator<T>`, который и выполняет всю работу. Экземпляр `IEnumerator<T>` может делать всего две вещи: возвращать текущее значение и переходить к следующему.

Интерфейс `IEnumerator<T>` играет важную роль, позволяя нам (и компилятору) писать код, который обрабатывает последовательность элементов, ничего не зная об устройстве этой последовательности. Все коллекции в .NET реализуют `IEnumerable<T>`, поэтому конструкциям, которые работают с последовательностями (например, цикл `foreach`), не требуется знать, как работать с каждым типом коллекций — им достаточно знать, как работать с `IEnumerable<T>`. Обратное тоже верно — все, что реализует `IEnumerable<T>`, автоматически считается последовательностью элементов, которую можно использовать в циклах `foreach` и в других подобных конструкциях, поддерживаемых в .NET и C#.

Как и в примере с лямбда-функциями, компилятор переписывает метод `YieldDemo` в класс, но на этот раз класс реализует `IEnumerator<int>`, поэтому цикл `foreach` знает, как с ним работать. Давайте теперь сами напишем код, чтобы получить тот же результат.

`YieldDemo` возвращает `IEnumerable<int>`, поэтому нам нужно объявить класс, реализующий этот интерфейс, чтобы его экземпляр можно было вернуть из `YieldDemo`. Как я уже говорил, единственное, что делает `IEnumerable<int>`, — возвращает `IEnumerator<int>` (по историческим причинам, чтобы обеспечить совместимость с кодом, написанным до появления .NET 2.0, помимо `IEnumerator<int>`, нам также необходимо предоставить реализацию необобщенного интерфейса `IEnumerator`, и в этом примере мы реализуем оба интерфейса в одном классе):

```
public class YieldDemo_Enumerable : IEnumerable<int> ← Реализует интерфейс
{
    public I IEnumerator<int> GetEnumerator()
    {
        return new YieldDemo_ Enumerator(); ← Возвращает экземпляр
    }                                         I Enumerator<int>
    IEnumerator IEnumerable.GetEnumerator()
    {
        return new YieldDemo_ Enumerator(); ←
    }
}
```

Теперь нужно написать наш `IEnumerator<int>`, который будет выполнять всю работу:

```
public class YieldDemo_Enumerator : IEnumerator<int>
{
```

Нам понадобится свойство `Current` для хранения текущего значения:

```
public int Current { get; private set; }
```

Далее следует самая важная часть. Разделим наш исходный код на фрагменты сразу после каждой инструкции `yield return` и заменим `yield return` на `Current =:`:

```
private void Step0()
{
    Current = 1;
}
private void Step1()
{
    Current = 2;
}
```

Следующая часть — метод `MoveNext`. Этот метод вызывает нужный фрагмент из предыдущего абзаца для обновления свойства `Current`. Он использует поле `_step`, чтобы запомнить, какой шаг должен выполняться, и после выполнения всех шагов возвращает `false`, чтобы сообщить, что все значения сгенерированы (если вы изучали информатику, то без труда узнаете здесь простую реализацию конечного автомата):

```
private int _step = 0; ← Переменная для хранения текущего шага
public bool MoveNext()
{
    switch(_step)
    {
        case 0:
            Step0();
            ++_step;
            break;
        case 1:
            Step1();
            ++_step;
            break;
        case 2: ← Генерация значений завершена;
            return false; возвращаем false
    }
    return true;
}
```

Теперь немного технического кода, не относящегося к данному примеру:

```
object IEnumarator.Current => Current;
public void Dispose() { }
public void Reset() { }
}
```

И наконец, обернем сгенерированные нами классы в метод, который можно вызвать:

```
public IEnumerable<int> YieldDemo()
{
    return new YieldDemo_Enumerable();
}
```

Фактический код, сгенерированный компилятором, длиннее и сложнее в основном потому, что в нашем примере я проигнорировал проверку ошибок. Однако концептуально это то, что создает компилятор. Компилятор переписывает наш код, разбивая его на фрагменты, и вызывает каждый фрагмент по очереди, когда это необходимо, создавая иллюзию приостановки и возобновления кода.

Для имитации работы `yield return` нам необходимо:

- преобразовать код, разделив его на части, и смоделировать один метод, который можно приостанавливать и возобновлять;
- стандартное представление, похожее на коллекцию (`IEnumerable<T>`), чтобы дать возможность использовать результаты этого преобразования.

Мы практически вплотную подошли к инструкциям `async/await` и классу `Task`, которые рассмотрим в следующей главе.

Итоги главы

- Компилятор C# реорганизует и переписывает ваш код, чтобы добавить возможности, отсутствующие в .NET.
- Для реализации лямбда-функций компилятор перемещает их код в новый метод, а общие данные — в новый класс.
- Для реализации `yield return` компилятор делит исходный код на фрагменты и заключает их в класс, который запускает требуемый фрагмент в нужное время, чтобы имитировать функцию, которую можно приостанавливать и возобновлять.

3

Ключевые слова `async` и `await`

В этой главе

- ✓ Использование Task и Task<T> для проверки завершения операции.
- ✓ Применение Task и Task<T> для уведомления вашего кода о завершении операции.
- ✓ Использование Task и Task<T> в синхронном коде.
- ✓ Как работает `async/await`.

В предыдущей главе вы увидели, как компилятор преобразует наш код, чтобы добавить новые возможности, поддерживаемые языком. В этой главе вы узнаете, как это применяется к `async/await`.

`async/await` — это инструкции, позволяющие писать асинхронный код, как если бы это был обычный синхронный код. В асинхронном программировании, когда выполняется операция, которая может заставить процессор ожидать некоторого события (обычно это получение данных с какого-то устройства, например чтение файла), вместо просто ожидания процессор делает что-то еще. Сделать асинхронный код похожим на обычный — это сама по себе большая проблема, потому что раньше каждую последовательность операций приходилось разбивать на небольшие части (по границам асинхронных операций) и вызывать необходимую часть в нужное время. Неудивительно, что это приводит к путанице при написании кода.

3.1. Сложность асинхронного кода

Для демонстрации сложности я поместил рис. 1.1 и 1.4 рядом (рис. 3.1).

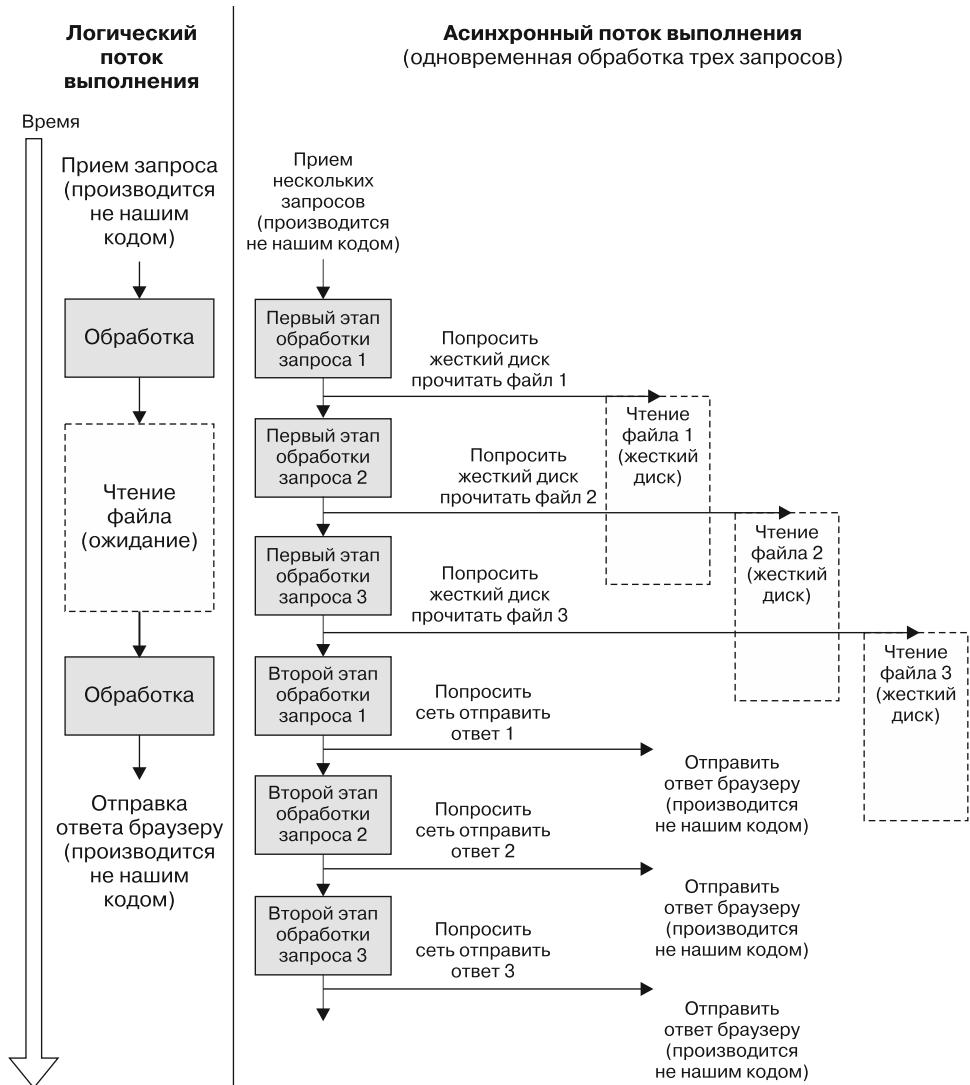


Рис. 3.1. Логический поток выполнения и код, работающий асинхронно

Очевидно, что левая часть, описывающая логический поток, проста, линейна и понятна, тогда как правая часть, описывающая работу асинхронной версии, весьма запутанна (и к тому же сложна в отладке).

Традиционно асинхронное программирование требует проектировать и писать код для схемы справа, разбив его на фрагменты, не отражающие логический поток выполнения. Кроме того, необходим дополнительный код, который будет управлять всем этим месивом и решать, что и когда запускать.

Инструкции `async/await` позволяют создавать код, описывающий логический поток, а компилятор преобразует его в код, способный выполняться асинхронно. Это позволяет писать код как на схеме слева, который будет выполняться как на схеме справа.

Проиллюстрируем сказанное на простом примере: напишем метод, который читает ширину изображения (в пикселях) из файла формата BMP. Я выбрал BMP, потому что, в отличие от более современных форматов, все данные в файле BMP находятся в фиксированном месте, что упрощает их извлечение. Мы прочитаем ширину изображения в два этапа.

1. Сначала проверим, действительно ли это файл изображения BMP. Для этого просмотрим начало файла: файлы изображений BMP начинаются с символов «BM».
2. Затем перейдем к восемнадцатому байту в файле, где хранится ширина в виде 32-битного (4 байта) целого числа.

Наш метод вернет ширину изображения в пикселях или выбросит исключение, если файл не является изображением BMP или возникнут какие-то другие ошибки. Поскольку мы еще не знаем, как писать асинхронный код, реализуем первую версию в виде старого доброго синхронного кода (листинг 3.1).

Листинг 3.1. Чтение ширины BMP, синхронная версия

```
int GetBitmapWidth(string path)
{
    using (var file = new FileStream(path, FileMode.Open, FileAccess.Read))
    {
        var fileId = new byte[2];
        var read = file.Read(fileId, 0, 2);
        if (read != 2 || fileId[0] != 'B' || fileId[1] != 'M')
            throw new Exception("Этот файл не BMP");

        file.Seek(18, SeekOrigin.Begin);
        var widthBuffer = new byte[4];
        read = file.Read(widthBuffer, 0, 4);
        if (read != 4) throw new Exception("Этот файл не BMP");
        return BitConverter.ToInt32(widthBuffer, 0);
    }
}
```

Файл должен начинаться с символов «BM»

Чтение ширины начиная с 18-го байта

Как видите, код довольно прост. Он читает первые два байта и проверяет, содержатся ли в них символы «BM». Затем выполняется переход к восемнадцатому байту и читается ширина изображения.

3.2. Знакомство с `Task` и `Task<T>`

Теперь сделаем этот код асинхронным. Для этого есть две веские причины:

- первая и самая важная — это книга об асинхронном программировании;
- вторая причина состоит в том, что основная операция, выполняемая нашим кодом, — это чтение файла, а чтение файла, как известно, является блокирующей операцией, которая заставляет поток ждать поступления данных с жесткого диска. Это означает, что мы можем повысить эффективность, используя поток для выполнения других задач во время ожидания, вместо того, чтобы заставлять операционную систему переключаться на другой поток (или другой процесс).

Основная операция, выполняемая нашим методом, — это чтение файла с помощью метода `Stream.Read`, и, к счастью, существует асинхронная версия метода `Stream.Read` под названием `Stream.ReadAsync`. Рассмотрим отличия в сигнтурах этих двух методов:

```
public int Read(byte[] buffer, int offset, int count);  
  
public Task<int> ReadAsync(byte[] buffer, int offset, int count,  
    CancellationToken cancellationToken);
```

Мы можем видеть следующие два различия в сигнатуре метода.

- Метод `Read` возвращает `int`, а `ReadAsync` — `Task<int>`. Классы `Task` и `Task<T>` являются важными элементами современного асинхронного программирования на C#, и мы рассмотрим их использование в этой главе.
- Метод `ReadAsync` также принимает параметр `CancellationToken`, но мы пока проигнорируем его, поскольку далее ему будет посвящена целая глава.

Ранее в этой главе я писал, что в случае асинхронного кода мы должны разделить его на части, а также нам нужна система для управления выполнением этих частей. `Task` — это класс, который помогает взаимодействовать с такой системой. `Task` выполняет несколько функций: представляет текущую асинхронную операцию, позволяет запланировать выполнение кода по завершении асинхронной операции (мы поговорим об этом далее) и дает возможность создавать и компоновать асинхронные операции (о них мы поговорим позже в книге).

В главе 2 мы познакомились с интерфейсом `IEnumerable<T>`, который поддерживает инструкцию `yield return`. Классы `Task` и `Task<T>` — это аналоги `IEnumerable<T>` в асинхронном программировании. Они стандартным способом представляют асинхронные операции, поэтому все асинхронные конструкции знают, как с ними работать.

Имя класса `Task` может показаться странным; слово `task` («задача») подразумевает, что есть некоторая выполняемая операция, но это не единственное его предназначение. `Task` представляет событие, которое может произойти в будущем, а `Task<T>` представляет значение, которое может быть доступно в будущем. Эти события и значения могут быть или не быть результатами чего-то, что мы описываем с помощью английского слова *task*. В информатике эти концепции часто называют *future* (будущее), *promise* (обещание) или *deferred value* (отложенное значение), но в этой книге мы будем ссылаться на них, используя термин `Task` из .NET/C#.

Важно отметить, что, хотя обычной практикой является создание `Task` или `Task<T>` для кода, который запускается в фоне (как будет показано в следующей главе), некоторые классы и методы в .NET используют слово `task`, ссылаясь на такой код или для управления контекстной информацией, связанной с ним. Объекты `Task` и `Task<T>` сами по себе не позволяют управлять фоновой операцией и не несут контекст, связанный с ней. `Task` просто в какой-то момент сообщает, что фоновая операция завершилась (объект `Task` представляет событие завершения фоновой операции), а `Task<T>` добавляет возможность получить результат этой операции (`Task<T>` представляет значение, полученное в результате фоновой операции). `Task` не является потоком или фоновой операцией, но иногда используется для передачи ее результатов.

В терминологии .NET/C# мы говорим, что задача завершена, когда происходит событие, представляемое объектом `Task`, или доступно значение, представляемое объектом `Task<T>`. Операция, которую представляет `Task`, также считается завершенной, если она отмечена как отмененная или завершилась ошибкой.

Например, вызывая `Task.Delay(1000)`, мы получаем объект, представляющий событие, которое произойдет через 1 с, но не имеет соответствующего потока выполнения и ничего не делает. Аналогично, когда мы вызываем `File.ReadAllBytesAsync` и при этом, например, нет потока, выполняющего фоновое чтение, система попросит контроллер диска (аппаратное устройство, отличное от процессора) загрузить данные и вызовет указанный нами код по завершении, поэтому мы получаем обратно объект `Task<byte[]>`, представляющий данные, которые будут получены с диска в будущем.

Метод `Read`, использованный в нашем примере, заполняет указанный нами буфер и возвращает количество успешно прочитанных байтов. По соображениям совместимости и производительности метод `ReadAsync` действует аналогично, но возвращает `Task<int>` вместо `int`. Возвращаемый объект `Task<int>` представляет количество успешно прочитанных байтов, которые будут доступны после завершения операции. Обратите внимание, что мы не должны обращаться к буферу, который передали в вызов `ReadAsync`, пока операция не будет завершена.

Итак, объекты `Task` или `Task<T>` представляют событие или значение соответственно, которые могут произойти или стать доступными в будущем. Если мы хотим узнать, произошло ли событие и доступно ли значение, `Task` и `Task<T>` поддерживают два асинхронных подхода — если использовать метафору с путешествием — модель «Мы уже приехали?» и модель «Разбуди меня, когда прибудем». Есть также синхронный подход на тот случай, если нет возможности или желания использовать асинхронный подход.

3.2.1. Мы уже приехали?

В модели «Мы уже приехали?» мы должны спросить `Task`, завершилась ли задача. Обычно это делается в цикле, который выполняет другие действия между проверками (это называется *опросом*), путем чтения свойства `IsCompleted`. Обратите внимание, что `IsCompleted` получает значение `true`, даже если задача завершилась ошибкой или была отменена.

`Task` также имеет свойство `Status`, которое мы можем использовать. Задача считается завершенной, если свойство `Status` получает значение `RanToCompletion`, `Canceled` или `Faulted`. Использовать свойство `IsCompleted` проще, чем свойство `Status`, потому что проверка одного условия вместо трех лаконичнее и менее подвержена ошибкам (мы поговорим об отмененных и потерпевших неудачу задачах позже в этой книге).

Не следует проверять `IsCompleted` или `Status` в цикле, если не требуется выполнять какую-то другую работу между проверками. Если вы просто ждете завершения задачи, то, выполняя проверку в цикле, вы не только используете поток для ожидания, полностью сводя на нет преимущества асинхронных методов, но и расходуете циклы процессора, понапрасну тратя ресурсы, которые другой код, выполняющийся на компьютере (включая саму операцию, завершения которой вы ждете), мог бы использовать для чего-то более полезного.

Эту проверку можно сравнить с вопросом «Мы уже приехали?», который задают сидя в машине. Если делать это слишком часто, вы будете мешать остальным пассажирам и можете даже приехать позже, если будете раздражать водителя.

Вот пример использования `IsCompleted` в цикле для проверки завершения задачи:

```
var readCompleted = File.ReadAllBytesAsync("example.bin");
while(!readCompleted.IsCompleted)
{
    UpdateCounter();
}
var bytes = readCompleted.Result;
// обработать полученные байты
```

В этом примере программа должна постоянно обновлять счетчик, ожидая получения данных с диска. Поэтому она в цикле обновляет счетчик и проверяет,

завершилось ли чтение. Как только данные станут доступны, она выйдет из цикла, чтобы обработать только что полученные данные.

Чаще всего нам ничего не нужно делать, ожидая, пока `IsCompleted` получит значение `true`, поэтому данная модель используется редко. В большинстве случаев (и на протяжении большей части этой книги) мы позволяем среде выполнения .NET планировать и запускать наши задачи и не используем модель «Мы уже приехали?». Эта модель полезна, только когда есть что делать во время ожидания, то есть по какой-то причине нежелательно возвращаться и освобождать поток (далее в книге мы рассмотрим пример взаимодействия с потоками пользовательского интерфейса).

3.2.2. Разбуди меня, когда приедем

В модели «Разбуди меня, когда приедем» задаче передается метод обратного вызова, который она вызовет, когда завершится (или будет отменена, или завершится ошибкой). Это реализуется путем передачи метода обратного вызова в метод `ContinueWith`.

По завершении задача будет передана методу обратного вызова в качестве параметра и ее можно использовать для проверки успешности выполнения операции, а в случае `Task<T>` — для получения значения результата:

```
var readCompleted = File.ReadAllBytesAsync("example.bin");
readCompleted.ContinueWith( t =>
{
    if(t.IsCompletedSuccessfully)
    {
        byte[] bytes = t.Result;
        // обработать полученные байты
    }
});
```

В отличие от предыдущей модели, эта очень хорошо подходит для нашего примера. Если посмотреть на код непосредственно рядом с первым вызовом `Read`, то он изменится с

```
var fileId = new byte[2];
var read = file.Read(fileId, 0, 2);
if (read != 2 || fileId[0] != 'B' || fileId[1] != 'M')
...
на
var fileId = new byte[2];
var read = file.ReadAsync(fileId, 0, 2, CancellationToken.None).
    ContinueWith(t=>
{
    if (t.Result != 2 || fileId[0] != 'B' || fileId[1] != 'M')
...
...
```

52 Глава 3. Ключевые слова `async` и `await`

В данном случае мы заменили `Read` на `ReadAsync` и весь код, следующий за вызовом `Read`, переместили в `ContinueWith` в виде лямбда-функции (дополнительно необходимо внести еще некоторые изменения, касающиеся использования `using` и `throw`, но, к счастью, в нашем примере они не влияют на эти три строки кода — мы поговорим об этом позже в главе).

С технической точки зрения можно сделать несколько асинхронных вызовов, вложив друг в друга вызовы `ContinueWith` с лямбда-функциями, как показано в следующем примере, однако такой код трудно читать и строки получаются слишком длинными. Например, чтение трех байтов из файла по одному байту за раз будет выглядеть так:

```
f.ReadAsync(buffer, 0, 1, CancellationToken.None).
    ContinueWith( t1 =>
{
    f.ReadAsync(buffer, 1, 1, CancellationToken.None).
        ContinueWith( t1 =>
{
    f.ReadAsync(buffer, 2, 1, CancellationToken.None).
        ContinueWith( t1 =>
{
    // конец чтения трех байтов!
});
});
});
```

Этот код трудно читать, и каждый вызов `ContinueWith` сдвигает код вправо все дальше и дальше. Если бы я захотел изменить этот пример, чтобы таким же способом прочитать четыре или более байтов, то он бы не поместился по ширине книжной страницы. (К слову: далее в этой главе вы увидите, как `async/await` решают эту проблему.)

3.2.3. Синхронный вариант

Кроме того, у вас может появиться желание дождаться выполнения задачи синхронным способом. Например, когда пишется синхронный код, использующий API, в котором есть только один асинхронный метод на основе `Task`, то лучшим решением будет вызов метода `Task.Wait` или чтение свойства `Task<T>.Result`. Вызов метода `Wait` или чтение свойства `Result` заблокирует текущий поток до завершения задачи и выбросит исключение, если задача будет отменена или завершится с ошибкой. Обратите внимание, что решение использовать метод `Wait` или свойство `Result` для ожидания завершения задачи довольно неэффективно и в первую очередь сводит на нет преимущества использования асинхронного подхода. В некоторых случаях это также может стать причиной взаимных

блокировок (взаимные блокировки приводят к зависанию программы, и мы подробно поговорим о них далее в книге):

```
var readCompleted = File.ReadAllBytesAsync("example.bin");
var bytes = readCompleted.Result; ←
// обработать полученные байты
```

Этот код будет ждать окончания
чтения файла

Этот подход следует использовать, только если нет другого выбора (в основном при интеграции синхронного и асинхронного кода).

3.2.4. После завершения задачи

После завершения задачи нужно проверить, была она выполнена успешно или нет. Оба класса, `Task` и `Task<T>`, содержат свойства `IsFaulted`, `IsCanceled` и `IsCompletedSuccessfully`, имена которых точно отражают их предназначение. Их можно использовать после завершения задачи для проверки ее статуса. (К ним можно обращаться и до завершения задачи; в этом случае они просто возвращают `false`.) Если `IsFaulted` имеет значение `true`, то можно прочитать свойство `Exception`, чтобы узнать, что пошло не так.

Если задача завершилась с ошибкой, то самый простой способ обработать исключение, выброшенное внутри задачи, — вызвать метод `Wait`, помещенный внутри блока `try-catch`. Вызов `Wait()` после завершения задачи не несет никакой опасности и не блокирует поток (потому что ожидаемое событие уже произошло). Он просто немедленно вернет управление, если задача завершилась успешно, или выбросит исключение, если задача была отменена или завершилась ошибкой. Благодаря такому поведению вам даже не нужно проверять, находится задача в состоянии ошибки или отмены (исключение будет выброшено, если задача не завершилась успехом).

Итак, если вы решите проверить, произошла ли ошибка при выполнении задачи, и обработать исключение без его повторного выброса, то используйте такой прием:

```
if(task.IsFaulted)
    HandleError(task.Exception);
```

Но если вы решите проверить, произошла ли ошибка при выполнении задачи, и сгенерировать исключение *только после ее завершения*, то можете использовать такой способ:

```
task.Wait();
```

Этот способ работает, потому что, как уже говорилось, вызов `Task.Wait` после завершения задачи либо ничего не делает и немедленно возвращает управление,

либо генерирует исключение. Обратите внимание, что последние два фрагмента кода поведут себя по-разному, если задача была отменена (далее в книге есть целая глава, рассказывающая об отмене).

Исключение в свойстве `Task.Exception` (а также исключение, сгенерированное вызовом метода `Wait` или обращением к свойству `Result`, если задача находится в состоянии ошибки) будет экземпляром `AggregateException`, содержащим исходное исключение в своем свойстве `InnerExceptions` (обратите внимание на множественное число — символ `s` в конце имени), которое не следует путать со свойством `InnerException` (единственное число), унаследованным от `Exception` и не используемым в этом случае.

`AggregateException` используется здесь для поддержки ситуаций, когда задача представляет комбинацию нескольких операций.

Если вы знаете, что может возникнуть только одно исключение, и хотите получить доступ к нему, а не к `AggregateException`, то используйте примерно такой код:

```
If(task.IsFaulted)
    HandleError(task.Exception.InnerException[0]);
```

`Task<T>` (но не `Task`) также имеет свойство `Result`, которое используется для получения значения, хранимого в задаче. Обычно обращение к свойству `Result` производится только после завершения задачи, при этом `IsCompleted` имеет значение `true` или вызывается `ContinueWith`. Если попытаться прочитать свойство `Result` до завершения задачи, то поток блокируется до завершения задачи. Это эквивалентно вызову `Wait` и чревато теми же опасностями и потерей эффективности, о которых говорилось выше. Если задача находится в состоянии ошибки или отмены, то чтение `Result` вызовет исключение.

Итак, при использовании задач без `async/await` можно обращаться к свойствам `IsCompleted` или `Status`, чтобы спросить: «Мы уже приехали?» И, так же как при поездке в машине, не стоит спрашивать слишком часто. Другой вариант — использовать `ContinueWith`, чтобы задача сообщила вам, когда она завершится («Разбуди меня, когда приедем»). Наконец, можно вызвать `Wait` или обратиться к `Result`, чтобы сделать задачу синхронной, но это неэффективно и опасно, потому что такой вызов блокирует поток до завершения задачи (вызов `Wait` или обращение к `Result` после завершения задачи совершенно безопасны и эффективны, потому что результат уже доступен и в блокировке нет необходимости).

Теперь, узнав, как работают `Task` и `Task<T>`, поговорим о том, как `async/await` могут упростить нам жизнь.

3.3. Как работают `async/await`

Мы узнали, что `Task` и `Task<T>` — это все, что нужно, чтобы писать асинхронный код, но разработка любого нетривиального кода с использованием `ContinueWith` и лямбда-функций (как в примере «Разбуди меня, когда приедем») быстро становится утомительной задачей, и сам код получается трудночитаемым. Скопируем только часть примера, которая «читает из файла ширину BMP», и изменим ее, задействовав `ReadAsync` и `ContinueWith`.

Для этого выполним элементарное механическое преобразование: каждый вызов `Read` заменим вызовом `ReadAsync`, а следующую за ним часть кода просто передадим в вызов `ContinueWith` в форме лямбда-функции:

```
file.ReadAsync(fileId, 0, 2, CancellationToken.None).
    ContinueWith(firstReadTask =>
{
    int read = firstReadTask.Result;
    if (read != 2 || fileId[0] != 'B' || fileId[1] != 'M')
    {
        // каким-то образом вернуть ошибку вызывающему коду
    }
    file.Seek(18, SeekOrigin.Begin);
    var widthBuffer = new byte[4];
    file.ReadAsync(widthBuffer, 0, 4, CancellationToken.None).
        ContinueWith(secondReadTask =>
{
    read = secondReadTask.Result;
    if(read != 4) throw new Exception("Этот файл не BMP");
    var result = BitConverter.ToInt32(widthBuffer, 0);
    // каким-то образом передать результат вызывающему коду
});
});
```

Ну и ну! Прежде простой и легко читаемый метод превратился в нечто неудобоваримое. Код тяжело читать, потому что он разделен асинхронными вызовами и не отражает логики нашего алгоритма. Но самое неприятное — наше преобразование даже не является корректным! В исходном коде имелась инструкция `using`, закрывающая файл по завершении или при исключении, поэтому, чтобы получить то же поведение, нужно заключить все в блок `try-catch` и реализовать нужные манипуляции самостоятельно (я не стал добавлять этот код в пример, потому что его и без того трудно читать). Нам также нужно передать исключение и результат вызывающему коду, и поскольку лямбда-функция вызывается асинхронно, мы не можем использовать `return` и `throw` для передачи информации вызывающей стороне. К счастью, у нас есть `async/await`, которые позаботятся обо всем этом.

56 Глава 3. Ключевые слова `async` и `await`

Чтобы переписать этот пример с использованием `async/await` и `ReadAsync`, необходимо внести следующие изменения.

- Нужно отметить наш метод ключевым словом `async`, и, как будет показано чуть ниже, само по себе это ничего не дает.
- Наш асинхронный метод больше не может возвращать `int`, потому что он немедленно вернет управление, а инициированная им операция завершится позже. Мы не можем вернуть `int`, потому что на момент возврата из метода результат еще неизвестен! К счастью, мы можем вернуть «`int`, который будет доступен в будущем», — `Task<int>`.
- Наконец, нужно добавить ключевое слово `await` перед каждым вызовом `ReadAsync`. Ключевое слово `await` сообщает компилятору, что код в этой точке необходимо приостановить и возобновить, когда завершится асинхронная операция.

Наш измененный метод, использующий `async/await`, показан в листинге 3.2. Изменения, отличающие его от оригинальной синхронной версии, выделены жирным.

Листинг 3.2. Чтение ширины BMP-файла, асинхронная версия

```
public async Task<int> GetBitmapWidth(string path)
{
    using (var file = new FileStream(path, FileMode.Open, FileAccess.Read))
    {
        var fileId = new byte[2];
        var read = await file.ReadAsync(fileId, 0, 2);
        if (read != 2 || fileId[0] != 'B' || fileId[1] != 'M')
            throw new Exception("Этот файл не BMP");

        file.Seek(18, SeekOrigin.Begin);
        var widthBuffer = new byte[4];
        read = await file.ReadAsync(widthBuffer, 0, 4);
        if (read != 4) throw new Exception("Этот файл не BMP");
        return BitConverter.ToInt32(widthBuffer, 0);
    }
}
```

Выглядит практически как оригинальная синхронная версия, только с добавлением ключевых слов `async` и `await`, но в действительности это совершенно другой код. Посмотрим, что он делает на самом деле.

Обратите внимание, что код в листинге 3.3 дает лишь концептуальное представление о том, что создает компилятор. Фактический код, сгенерированный компилятором, сильно отличается и гораздо сложнее. Я использую эту упрощенную версию, потому что она понятнее и при этом достаточно наглядно демонстрирует, что делает компилятор. В конце этого раздела я перечислю

основные различия между моей версией и фактическим кодом, сгенерированным компилятором.

`async/await` используют модель «Разбуди меня, когда приедем». Они разбивают код на части (подобно инструкции `yield return` из предыдущей главы) и используют метод `ContinueWith` задачи для запуска каждой части в нужное время.

Посмотрим, как компилятор переписал наш код. Но прежде внесем одно небольшое изменение: в примере `async/await` мы возвращаем `Task<int>`, но мы пока не обсуждали приемы создания `Task` (не волнуйтесь, этому будет посвящена целая глава). Поэтому мы передадим нашему методу два обратных вызова: `setResult`, который будет вызываться в случае успешного завершения нашего метода, и `setException`, который будет вызываться в случае исключения.

Компилятор выделяет код после `await` в другой метод (подобно тому, как мы делали это с `yield return` в предыдущей главе) и передает его в вызов `ContinueWith` задачи. Чтобы иметь возможность совместно использовать переменные, локальные переменные перемещаются в класс, как мы делали это в примере с лямбда-функциями.

Листинг 3.3. Чтение ширины BMP-файла, асинхронная версия только с `async` и `ContinueWith`

```
public void GetBitmapWidth(string path,
                           Action<int> setResult, Action<Exception> setException)
{
    var data = new ClassForGetBitmapWidth();
    data.setResult = setResult;
    data.setException = setException;
    data.file = new FileStream(path, FileMode.Open, FileAccess.Read); ← Код из листинга 3.2
    try
    {
        data.fileId = new byte[2];
        var read = data.file.ReadAsync(data.fileId, 0, 2). ContinueWith(data.GetBitmapWidthStep2); ←
    } ←
    catch(Exception ex)
    {
        data.file.Dispose(); ← Код, добавленный
        setException(ex); ← для имитации поведения
    } ←
}
```

Здесь показан преобразованный код до первой инструкции `await`. Обратите внимание, что наши изменения не заставляют эту часть кода работать асинхронно. Все, что предшествует первой инструкции `await`, выполняется как обычный синхронный код. И если в коде встретится вызов `async`-метода без `await`, то он будет выполнен так, будто и не был объявлен асинхронным (хотя возвращаемое значение все равно будет обернуто в `Task`).

58 Глава 3. Ключевые слова async и await

Нам пришлось заменить оператор `using` на `try-catch`, чтобы гарантировать корректное закрытие файла в случае исключения (обратите внимание, что конструкция `try-finally` здесь не подходит, потому что в случае успешного выполнения этой части кода файл должен оставаться открытым, пока не завершится следующая часть).

Теперь обратим свое внимание на класс, используемый для хранения «локальных» переменных:

```
private class ClassForGetBitmapWidth
{
    public Stream file;
    public byte[] fileId;
    public byte[] widthBuffer;
    public Action<int> setResult;
    public Action<Exception> setException;
```

Вот как в этом классе выглядит код, находящийся между первой и второй инструкциями `await`:

```
public void GetBitmapWidthStep2(Task<int> task)
{
    try
    {
        var read = task.Result;
        if (read != 2 || fileId[0] != 'B' || fileId[1] != 'M')
            throw new Exception("Этот файл не BMP");

        file.Seek(18, SeekOrigin.Begin);
        widthBuffer = new byte[4];
        file.ReadAsync(widthBuffer, 0, 4).
            ContinueWith(GetBitmapWidthStep3);
    }
    catch(Exception ex)
    {
        file.Dispose();
        setException(ex);
    }
}
```

Код из листинга 3.2

Код, добавленный для имитации поведения инструкции using

Выглядит так, будто мы не проверили результат предыдущей операции. Мы не прочитали ни свойство `Task.IsCompletedSuccessfully`, ни свойство `Task.Status`. А значит, не знаем, была ли ошибка. Однако попытка прочитать свойство `Task.Result` приведет к исключению, если задача завершилась неудачей, поэтому нет необходимости писать код, явно проверяющий ошибки. Обратите также внимание, что этот метод вызывается из `ContinueWith`, а значит, к моменту вызова задача уже завершена, и мы можем без опаски прочитать свойство `Result`, потому что чтение свойства `Result` в этом месте — безопасная и неблокирующая операция.

Для части кода, следующей за последней инструкцией `await`, есть метод:

```
public void GetBitmapWidthStep3(Task<int> task)
{
    try
    {
        var read = task.Result;
        if(read != 4) throw new Exception("Этот файл не BMP");
        file.Dispose();
        var result = BitConverter.ToInt32(widthBuffer, 0);
        setResult(result); ← Вместо инструкции return
    }
    catch(Exception ex)
    {
        file.Dispose();
        setException(ex);
    }
}
```

Код из листинга 3.2

Код, добавленный для имитации поведения инструкции `using`

Точно так же, как в примере с `yield return` в главе 2, компилятор разделил нашу функцию на части и добавил код для их вызова в нужное время. Мы увидели, что первая часть до первой инструкции `await` выполняется в момент вызова метода. Добавление ключевого слова `async` в объявление метода не делает его асинхронным. Это просто флаг, сообщающий компилятору, что тот должен отыскать в методе ключевые слова `await` и разделить его на части. Точно так же инструкция `await` ничего не ждет — она фактически завершает выполнение текущей части и возвращает управление вызывающему коду.

А теперь, как и обещал, я перечислю основные различия между кодом, о котором мы только что говорили, и кодом, который компилятор генерирует в действительности.

- Компилятор не делит код на разные методы, а создает единый метод, реализующий конечный автомат, который отслеживает текущее состояние с помощью переменной и использует большой оператор `switch` для запуска нужного фрагмента кода.
- Вместо `ContinueWith` компилятор использует внутренний объект, так называемый `awaiter`. Я же использовал метод `ContinueWith` в примере, потому что он концептуально похож, и если вы не собираетесь писать свой компилятор или свою поддержку асинхронного выполнения для .NET, то помнить об этом совершенно необязательно.
- На самом деле инструкция `await` делает гораздо больше, чем `ContinueWith`, — она не только запускает обратный вызов после завершения задачи, как это делает `ContinueWith`, но и выполняет ряд дополнительных полезных действий, о которых мы поговорим позже в этой книге.

3.4. Асинхронные методы void

Представим, что мы пишем приложение WinForms и хотим добавить функцию, которая копирует текст из одного файла в другой, когда пользователь нажимает кнопку. Допустим также, что мы знаем, что файлы имеют небольшой размер и их можно целиком загрузить в память. Код этой функции будет выглядеть примерно так, как в листинге 3.4.

Листинг 3.4. Асинхронный обработчик событий

```
private async void Button1_Click(object sender, EventArgs ea)
{
    var text = await File.ReadAllTextAsync("source.txt");
    await File.WriteAllTextAsync("dest.txt", text);
}
```

Этот метод просто асинхронно загружает содержимое файла в переменную, а затем асинхронно записывает содержимое переменной в другой файл. Теперь воспользуемся новыми знаниями, полученными в этой главе, и преобразуем этот код так же, как преобразовали метод `GetBitmapWidth` в листинге 3.3, только на этот раз сохраним сигнатуру обработчика событий (листинг 3.5). Мы не можем добавить параметры `setResult` и `setException` (точно так же как в асинхронной версии мы должны были вернуть `void` и не могли вернуть `Task`).

Листинг 3.5. Код асинхронного обработчика событий, преобразованный компилятором

```
private void Button1_Click(object sender, EventArgs ea)
{
    var data = new ClassForButton1_Click();
    File.ReadAllTextAsync("source.txt").
        ContinueWith(data.Button1_ClickStep2);
}
private class ClassForButton1_Click
{
    public void Button1_ClickStep2(Task<string> task)
    {
        try
        {
            var text = task.Result;
            File.WriteAllTextAsync("dest.txt", text).
                ContinueWith(Button1_ClickStep3);
        }
        catch
        {
            // ? ← | У нас нет возможности уведомить
            // о возникшем исключении
        }
    }
}
```

```

public void Button1_ClickStep3(Task task)
{
    if(task.IsFaulted)
    {
        // ? ← | У нас нет возможности уведомить
        // о возникшем исключении (снова)
    }
    else
    {
        // ? ← | У нас нет возможности уведомить
        // об успешном завершении
    }
}

```

Поскольку этот метод прост, его преобразованная версия тоже выглядит просто (хотя писать ее было немного утомительно). Нам даже не пришлось перемещать локальные переменные в отдельный класс. Однако у нас есть проблема: отсутствие всякой возможности уведомить остальную часть программы о том, что выполнение операции завершилось. Хуже того, мы не можем даже уведомить программу, если вдруг возникнет какая-то ошибка. Мы добавили три комментария с вопросительными знаками и не знаем, что там написать.

Это характерно для асинхронных методов с типом возвращаемого значения `void`. Поскольку `Task` отсутствует, вызывающий код не сможет узнать, когда метод завершит выполнение (все способы, о которых мы говорили — `await`, `Wait`, `IsCompleted` и даже `ContinueWith` — требуют объекта `Task`). В данном случае это не проблема, поскольку обработчики событий обычно являются операциями, действующими по принципу «запустил и забыл», когда вызывающему коду неважно, что делает обработчик или когда он завершит работу (лишь бы он быстро возвращал управление вызывающему коду, что и делает наш обработчик).

Кроме того, нет способа сообщить об исключении (так как нет доступа к свойству `Task.Exception` и нет никакой другой возможности передать исключение из-за отсутствия `Task`), но, в отличие от случая успешного выполнения, это реальная проблема, потому что подобный код может выбросить неожиданное исключение и аварийно завершиться. Подробнее об исключениях мы поговорим в отдельной главе, а пока отмечу, что решение состоит в том, чтобы просто не позволять методам `async void` генерировать исключения, — в своем методе `async void` вы должны перехватывать все исключения и самостоятельно их обрабатывать.

Итак, если такой подход настолько проблематичен, то почему вообще существует возможность объявлять методы `async void`? Причина в том, что такие методы часто играют роль обработчиков событий. По соглашению, как и в нашем примере, обработчики событий всегда возвращают тип `void`, поэтому, если бы методы `async` не поддерживали возврат `void`, мы не смогли бы использовать `async/await` в обработчиках событий.

Так постепенно мы пришли к официальному правилу, касающемуся методов `async void`: эти методы должны использоваться только как обработчики событий и не должны генерировать никаких исключений. В листинге 3.6 показан правильный вариант обработчика событий из листинга 3.4.

Листинг 3.6. Асинхронный обработчик событий с обработкой ошибок

```
private async void Button1_Click(object sender, EventArgs ea)
{
    try
    {
        var text = await File.ReadAllTextAsync("source.txt");
        await File.WriteAllTextAsync("dest.txt", text);
    }
    catch(Exception ex)
    {
        // Обработать исключение
    }
}
```

3.5. `ValueTask` и `ValueTask<T>`

Некоторые методы не всегда действуют асинхронно. Например, предположим, что у нас есть метод, выполняющий асинхронную операцию, только если он не может вернуть ответ из кэша:

```
public async Task<int> GetValue(string request)
{
    if(_cache.TryGetValue(request, out var fromCache))
    {
        return fromCache; ← | Если возможно, возвращает
    }                                | значение из кэша
    int newValue = await GetValueFromServer(request); ← | Иначе выполняет
    return newValue;                  | асинхронную операцию
}
```

Обратите внимание, что здесь `_cache` не является словарем (`Dictionary`). `Dictionary` не является потокобезопасным и не может использоваться в асинхронных методах. О потокобезопасных структурах данных, которые можно использовать для создания потокобезопасного кэша, мы поговорим в главе 13.

Метод `GetValue` сначала проверяет, имеется ли требуемое значение в кэше. Если да, то возвращает значение до выполнения первой инструкции `await`. Как мы уже видели выше, код, предшествующий первой инструкции `await`, выполняется синхронно, поэтому если искомое значение присутствует в кэше, то оно будет возвращено немедленно, вследствие чего возвращаемое значение `Task<int>` станет просто очень сложной оберткой для `int`.

Создание целого объекта `Task<int>`, когда он не требуется, — это расточительство, и было бы лучше, если бы имелась возможность немедленно вернуть `int`, когда значение доступно в кэше, и возвращать объект `Task`, только когда нужно выполнить асинхронную операцию. Именно такую возможность предоставляет `ValueTask<T>` — структура, которая содержит непосредственное значение, если оно доступно немедленно, и ссылку на `Task<T>` в противном случае. Необобщенный тип `ValueTask` действует аналогично, но содержит только флаг, указывающий на завершение операции, а не значение.

Объекты `ValueTask` и `ValueTask<T>` можно использовать точно так же, как `Task` и `Task<T>`. Они имеют многие свойства, которые есть в `Task` и `Task<T>`. А если понадобится использовать какую-то особенность `Task`, недоступную в `ValueTask` (например, `Wait`), то можно задействовать метод `ValueTask.AsTask()`, чтобы получить задачу, хранящуюся внутри `ValueTask`.

`ValueTask` и `ValueTask<T>` менее эффективны, чем `Task` и `Task<T>`, когда выполняется асинхронная операция, но намного эффективнее, если результат доступен немедленно. Возвращайте `ValueTask` из методов, которые чаще возвращают значение без выполнения асинхронной операции, особенно если эти методы вызываются часто.

3.6. А что насчет многопоточности?

В главе 1 я сказал, что асинхронность и многопоточность прекрасно работают вместе. Однако в этой главе мы вообще не затронули тему многопоточности. Также я сказал, что обратный вызов, который передается в `ContinueWith`, будет запущен позже, но мы полностью проигнорировали, как и где будет запущен обратный вызов. Это подводит нас к следующей главе, которая посвящена многопоточности.

Итоги главы

- `Task` представляет событие, которое может произойти в будущем.
- `Task<T>` представляет значение, которое может быть доступно в будущем.
- Когда событие происходит или значение становится доступно, мы говорим, что задача завершилась.
- Свойства `IsCompleted` и `Status` можно использовать для проверки завершения задачи.
- Используйте `ContinueWith`, чтобы задача уведомила вас по ее завершении.
- Вы можете вызвать `Wait` или `Result`, чтобы выполнить задачу синхронно, но это неэффективно и опасно.

64 Глава 3. Ключевые слова `async` и `await`

- Вызов `Wait` или `Result` после завершения задачи совершенно безопасен и эффективен.
- `async` — это просто флаг, сообщающий компилятору, что метод нужно разбить на части, если в нем обнаружится ключевое слово `await`.
- Ключевое слово `async` не заставляет код работать в фоновом режиме. Без `await` он ничего не делает (кроме того, что заставляет компилятор сконструировать огромный объем шаблонного кода).
- Компилятор разбивает метод на части после каждой инструкции `await` и (концептуально) передает следующую часть в вызов `ContinueWith`.
- `await` ничего не ждет, а завершает текущую часть и возвращает управление вызывающему коду.
- Методы `async` могут возвращать тип `void`, но в таких случаях нет возможности узнать, когда метод завершился, и все исключения необходимо перехватывать и обрабатывать внутри метода.
- Если `async`-метод часто немедленно возвращает результат, не выполняя асинхронных операций, то его эффективность можно повысить, используя `ValueTask` или `ValueTask<T>` вместо `Task` или `Task<T>`.

4

Основы многопоточности

В этой главе

- ✓ Основы потоков.
- ✓ Запуск потоков.
- ✓ Ожидание завершения потоков.
- ✓ Доступ к общим данным и использование блокировок.
- ✓ Основы взаимных блокировок.

В главе 1 говорилось, что система может выполнять параллельно сразу несколько фрагментов кода — намного больше, чем число ядер в процессоре, — быстро переключаясь между ними. Это возможно благодаря аппаратному таймеру внутри процессора. Каждый раз, когда таймер отмеряет очередной квант времени, операционная система может приостановить код, выполняющийся в данный момент, и переключиться на выполнение другого кода. Если переключение происходит достаточно быстро, то создается иллюзия, что все потоки работают одновременно.

В этой главе мы поговорим о том, как использовать потоки для параллельного выполнения разных заданий, и обсудим ключевые аспекты параллельного программирования. В следующей главе мы объединим эти темы с инструкциями `async/await`.

Сразу после запуска процесса в нем выполняется только один поток, который запускает метод `Main` (правда, есть еще несколько потоков, контролируемых системой, но мы пока отложим их в сторону). Этот начальный поток называется главным. Теперь давайте посмотрим, как использовать дополнительные потоки, чтобы позволить нескольким фрагментам кода выполнятся одновременно.

4.1. Способы запуска кода в другом потоке

Теперь, когда мы решили, что хотим запускать код параллельно, поговорим о том, как это сделать. В этом разделе рассматриваются три наиболее распространенных способа запуска кода в другом потоке, имеющихся в языке C#. Начнем с самого старого и гибкого варианта — создания собственного потока.

4.1.1. `Thread.Start`

В .NET поток представлен классом с соответствующим именем `System.Threading.Thread`. Этот класс позволяет контролировать существующие потоки и создавать новые.

Чтобы запустить новый поток, сначала нужно создать объект `Thread`, передав конструктору функцию обратного вызова с кодом, который должен выполняться в новом потоке. После этого можно настроить параметры потока и затем вызвать `Thread.Start`, чтобы запустить его. В листинге 4.1 показано, как создать и настроить поток.

Листинг 4.1. Создание потока

```
public void RunInBackground()
{
    var newThread = new Thread(CodeToRunInBackgroundThread); ← Создает
    newThread.IsBackground = true; ← объект потока
    newThread.Start(); ← Настраивает поток
} ← Запускает поток
private void CodeToRunInBackgroundThread() ← Код, который будет выполняться
{
    Console.WriteLine("Делаем что-то");
}
```

Как видите, этот код точно следует описанным шагам.

1. Создает поток, передав конструктору метод, который должен быть запущен в потоке.
2. Настраивает параметры потока, в данном случае делает его фоновым (о фоновых потоках мы поговорим позже в этой книге). Этот шаг необязателен.
3. Запускает поток вызовом `Thread.Start`.

Конструктор класса `Thread` имеет две версии, принимающие разные делегаты. Простая версия, использованная в этом примере, принимает метод `void` без параметров. Более сложная параметризованная версия принимает метод `void` с одним параметром типа `object`.

Метод `Thread.Start` тоже имеет две версии: одна не содержит параметров, а вторая принимает параметр типа `object`. Второй версии `Thread.Start` можно передать любой объект, который использует код потока.

Это позволяет написать один метод для потоков, выполняющих немного разные действия, и передавать разные значения параметра каждому потоку, чтобы различать их. Например, создадим 100 потоков и передадим каждому разное число (листинг 4.2).

Листинг 4.2. Создание потока с параметром

```
public void RunLotsOfThreads()
{
    var threads = new Thread[100];
    for(int i=0;i<100;++i)
    {
        threads[i] = new Thread(MyThread); ← Передает каждому
        threads[i].Start(i);             ← потоку свое значение
    }
}
private void MyThread(object? parameter)
{
    Console.WriteLine($"Привет из потока {parameter}"); ← Использует переданное
}                                                 ← значение
```

В листинге 4.2 мы просто передаем индекс цикла в `Thread.Start`, а тот передается нашему методу `MyThread` в начале выполнения в новом потоке.

Передача метода без параметров конструктору `Thread` и последующее использование параметризованной версии `Thread.Start` или наоборот не имеет особого смысла, но поддерживается. Если использовать делегат с параметрами и `Thread.Start` без параметров, то методу будет передан параметр со значением `null`. Если использовать делегат без параметров и параметризованный `Thread.Start`, то заданное вами значение будет проигнорировано.

Класс `Thread` также содержит метод `Join`, который будет ждать, пока поток завершит работу. *Join* (присоединение) — это стандартный термин в информатике, означающий ожидание завершения потока. Мне попадались противоречивые истории о происхождении этого термина, и во всех них используются метафоры, которые я не хочу здесь повторять, потому что они не очень удачные. Нам просто придется принять, что в контексте потоков *join* (присоединение) означает *wait* (ожидание).

Метод `Join` удобно использовать в случаях, когда требуется запустить несколько потоков параллельно, а после их завершения сделать что-то еще, например обединить вычисленные ими результаты. `Thread.Join` немедленно вернет значение, если поток уже завершился. Код в листинге 4.3 запускает три потока и ждет, пока они все завершатся, после чего уведомляет пользователя, что работа выполнена.

Листинг 4.3. Ожидание завершения потоков

```
public void RunAndWait()
{
    var threads = new Thread[3];
    for(int i=0;i<3;++i)           ← Запускает
    {
        threads[i] = new Thread(DoWork);
        threads[i].Start();
    }
    foreach(var current in threads) ← Ждет завершения
    {
        current.Join();
    }
    Console.WriteLine("Выполнено");
}

private void DoWork()
{
    Console.WriteLine("Выполняется работа");
}
```

Код в листинге 4.3 запускает три потока в первом цикле, а затем ждет их завершения во втором. Важно, чтобы это были два отдельных цикла, потому что по условию требуется сначала запустить все потоки и только потом ждать их завершения. Если запускать потоки и ждать их завершения в одном цикле, то это приведет к последовательному выполнению с избыточными накладными расходами на создание и запуск потоков (эта проблема называется *синхронизацией*, и мы обсудим ее в главе 7).

Второй цикл выглядит так, будто он зависит от порядка потоков в списке, но на самом деле это не так. Неважно, в каком порядке программа будет ждать завершения потоков. Если самый долго выполняющийся поток окажется первым, то код будет ждать только его завершения, а последующие вызовы `Join` для других уже завершившихся потоков будут возвращать управление немедленно. Если самый долго выполняющийся поток является последним, то цикл будет ждать завершения первого потока, потом второго и т. д., пока не доберется до последнего. В обоих случаях цикл будет ждать, пока не завершится поток, выполняющийся дальше других.

Класс `Thread` имеет ряд методов, позволяющих управлять потоками: `Suspend`, `Resume` и `Abort`. На первый взгляд они выглядят довольно удобными, но

в действительности чрезвычайно опасны, и вы никогда не должны их использовать. О причинах я расскажу позже в этой главе.

Использование класса `Thread` и метода `Thread.Start` — единственный способ получить поток, находящийся под вашим полным контролем. С таким потоком можно сделать все что угодно без риска помешать другому коду, работающему в вашем приложении.

Когда использовать `Thread.Start`

Используйте `Thread.Start`:

- ◆ для долго выполняющегося кода;
- ◆ если нужно изменить свойства потока, такие как информация о языке и локали, признак выполнения в фоне, СОМ-апартаменты и т. д. (обо всех настройках потока мы поговорим ближе к концу этой главы).

Не используйте `Thread.Start` для:

- ◆ асинхронного кода;
- ◆ коротких задач.

Создание и уничтожение потоков требует значительных вычислительных ресурсов, и если создать много потоков, каждый из которых выполняет лишь небольшой объем работы, то приложение может потратить больше времени на управление потоками, чем на выполнение полезной работы.

Это влияет на эффективность асинхронного кода, потому что такой код, даже требующий довольно много времени для выполнения, обычно состоит из множества коротких частей. Например, следующий метод выполняет две асинхронные операции:

```
private async Task SoSomethingComplicated()
{
    await DoFirstPart();
    await DoSecondPart();
}
```

После запуска этот метод доберется до вызова `DoFirstPart` и вернет управление вызывающему коду, как только `DoFirstPart` запустит асинхронную операцию (вызывающий код, скорее всего, тоже будет использовать `await`, как и код, вызывающий этот вызывающий код, и т. д., пока выполнение не доберется до вершины иерархии вызовов `await`, после чего поток будет освобожден). Когда асинхронная операция завершится, метод возобновит работу, требуя от потока выполниться ровно столько времени, сколько нужно, чтобы добраться до вызова `DoSecondPart`, где поток снова будет освобожден. Позже, когда `DoSecondPart` завершится, метод возобновит работу, снова потребовав поток. Если эта последовательность

действий будет включать создание и уничтожение потоков, то она потребует создания и уничтожения двух потоков.

Короткие задачи имеют ту же проблему. Если поток запускается, только чтобы выполнить быстрые, короткие вычисления, то может случиться так, что на создание и уничтожение потоков тратится больше времени, чем на выполнение полезной работы. И это подводит нас к нашей следующей теме — пул потоков.

4.1.2. Пул потоков

Пул потоков — это решение, направленное на уменьшение накладных расходов на создание и уничтожение потоков, о которых мы говорили. Используя пул потоков, система поддерживает небольшое количество потоков в фоновом режиме и всякий раз, когда нужно что-то запустить, позволяет использовать один из этих уже существующих потоков. Система автоматически управляет этими потоками и создает новые по мере необходимости (регулируя их количество между минимальным и максимальным количествами, которыми управляете вы).

Пул потоков — оптимальное решение для выполнения коротких задач, когда один и тот же поток может выполнять несколько задач одну за другой. Используя пул потоков для выполнения длительной задачи, вы фактически выведете поток из ротации на долгое время, и когда все потоки в пуле окажутся заняты, новая задача должна будет ждать, пока не освободится один из потоков.

Кроме того, получая поток «взаймы», вы не должны изменять никакие его свойства, поскольку любое изменение повлияет на код, который будет выполняться в том же потоке в будущем (точно так же вы не станете переставлять мебель в чужой квартире, прийдя в гости). Если вам нужен поток с особыми свойствами, то создайте его с помощью класса `Thread`.

Пул потоков управляется классом с соответствующим именем `System.Threading.ThreadPool`. Для запуска задачи в пуле потоков используется метод с менее говорящим именем `QueueUserWorkItem` (листинг 4.4).

Листинг 4.4. Работа с пулом потоков

```
public void RunInBackground()
{
    ThreadPool.QueueUserWorkItem(RunInPool); ← Ставит код в очередь
}                                         на выполнение в пуле потоков

private void RunInPool(object? state) ← Код для выполнения
{                                         в потоке
    Console.WriteLine("Делаем что-то");
}
```

Этот код аналогичен коду в листинге 4.1, но здесь мы не можем изменить конфигурацию потока (потому что заимствуем существующий поток) и нам не нужно запускать поток вручную (потому что поток уже запущен).

Как и `Thread.Start`, метод `QueueUserWorkItem` тоже имеет параметризованную и непараметризованную версии. Но, в отличие от класса `Thread`, метод, который выполняется в пуле потоков, всегда принимает параметр типа `object`. Если использовать непараметризованную версию `QueueUserWorkItem`, то код получит в параметре значение `null`. Перепишем код из листинга 4.2 так, чтобы он использовал пул потоков (листинг 4.5).

Листинг 4.5. Запуск в пуле потоков с параметром

```
public void RunInBackground()
{
    for(int i=0;i<10;++i)
    {
        ThreadPool.QueueUserWorkItem(RunInPool,i); ← Передача значения
    }                                                 в поток
}
private void RunInPool(object? parameter) ← Значение принимается
{
    Console.WriteLine($"Привет из потока {parameter}");
}
```

Код в этом примере не содержит ничего нового — он точно такой же, как и в листинге 4.2, за исключением того, что в нем не требуется запускать поток вручную.

В отличие от класса `Thread` с его методом `Join`, пул потоков не имеет встроенного способа дождаться завершения выполняемого кода. Позже в этой главе мы увидим, как можно реализовать свой способ ожидания завершения фонового кода.

Когда использовать `ThreadPool.QueueUserWorkItem`

Используйте `ThreadPool.QueueUserWorkItem` для:

- ◆ коротких задач.

Не используйте `ThreadPool.QueueUserWorkItem`:

- ◆ для долго выполняющихся задач;
- ◆ когда нужно изменить свойства потока;
- ◆ с асинхронными операциями на основе `Task`;
- ◆ с инструкциями `async/await`.

В этой главе рассказывается о пуле потоков, поэтому стоит отметить, что существует один пул потоков, создаваемый фреймворком автоматически, и все примеры здесь используют его. Однако есть возможность создать свой собственный пул потоков, но в этом нет большого смысла и потому поступать так, вероятно, не следует.

Интерфейс пула потоков устарел, не отличается гибкостью (например, мы вынуждены использовать метод с именем `QueueUserWorkItem`) и плохо работает с задачами `Task` и инструкциями `async/await` (потому что он старше их на десятилетие). По этим причинам был создан `Task.Run`.

4.1.3. Task.Run

Мы узнали, что пул потоков оптимизирован для запуска множества коротких задач и что асинхронные задачи на самом деле являются последовательностью коротких задач, поэтому пул потоков идеально подходит для запуска асинхронного кода, но проблема в том, что метод `QueueUserWorkItem` не использует класс `Task` (потому что он старше `async/await` и `Task` примерно на десятилетие). Именно поэтому был реализован метод `Task.Run`.

`Task.Run` запускает код в пуле потоков, подобно `ThreadPool.QueueUserWorkItem`, но имеет более удобный интерфейс, поддерживающий `async/await`. В простых сценариях он работает практически так же, как `ThreadPool.QueueUserWorkItem` в предыдущем примере (листинг 4.6).

Листинг 4.6. Запуск задания в пуле потоков с помощью `Task.Run`

```
public void RunInBackground()
{
    Task.Run(RunInPool);           ← Ставит задание в очередь
}                                ← для выполнения в пуле потоков
                                    Код для выполнения

private void RunInPool()
{
    Console.WriteLine("Делаем что-то");
}
```

Это тот же код, что и в примере использования пула потоков в листинге 4.4, но вместо `ThreadPool.QueueUserWorkItem` задействует `Task.Run`. В отличие от класса `ThreadPool`, метод `Task.Run` очень хорошо работает с `async/await` (и другими методами, возвращающими `Task`) (листинг 4.7).

Как видите, этот код просто работает. Нам не пришлось делать ничего особенного, чтобы запустить метод `async` с помощью `Task.Run`.

Кроме того, `Task.Run` возвращает сам объект `Task`, и его можно использовать, чтобы узнать, когда завершится код, запущенный в пуле потоков, чего так

не хватает в классе `ThreadPool`. В листинге 4.8 приводится адаптация примера из листинга 4.3, в котором код создает несколько потоков с помощью класса `Thread` и ждет, пока все они завершатся.

Листинг 4.7. Запуск кода `async` с помощью `Task.Run`

```
public void RunInBackground()
{
    Task.Run(RunInPool);           ← Ставит задание в очередь
}                                     для выполнения в пуле потоков

private async Task RunInPool() ← Код для выполнения
{
    await Task.Delay(500);
    Console.WriteLine("Делал что-то асинхронное");
}
```

Листинг 4.8. Ожидание завершения задач с помощью `Task.Run`

```
public async Task RunInBackground()
{
    var tasks = new Task[10];
    for(int i=0;i<10;++i)
    {
        tasks[i] = Task.Run(RunInPool);           ← Ставит задачу в очередь
    }                                         для выполнения в пуле потоков
    await Task.WhenAll(tasks);               ← Ждет завершения
    Console.WriteLine("Все выполнено");      задач
}

private void RunInPool()
{
    Console.WriteLine("Делаем что-то");
}
```

Дождаться завершения задачи можно с помощью метода `Task.WhenAll`, который выглядит гораздо элегантнее, чем цикл с `Thread.Join`. Он не только не требует цикла, но и ждет асинхронно.

Обратите внимание, что в случаях, когда метод `Task.Run` используется без `await`, компилятор выдает предупреждение, но добавление `await` почти всегда является неверным решением. Добавляя `await` перед вызовом `Task.Run`, вы сообщаете компилятору, что нужно дождаться завершения задачи, прежде чем переходить к следующей строке кода, по сути заставляя код выполнятся синхронно, что сводит на нет смысл использования `Task.Run`. Используя `Task.Run` с `await`, вы берете на себя накладные расходы по управлению различными задачами, не получая никаких преимуществ — эффективнее просто запустить код без `Task.Run`. Исключением из этого правила является поток пользовательского интерфейса, и мы поговорим о нем ближе к концу этой главы.

Чтобы избавиться от предупреждения, можно присвоить объект `Task`, возвращаемый методом `Task.Run`, в специальную переменную-заполнитель:

```
Task.Run(MethodToRunInBackground); ← Может выдать предупреждение
_ = Task.Run(MethodToRunInBackground); ← Нет предупреждения
```

`Task.Run` не имеет параметризованной версии, как `Thread.Start` и `ThreadPool.QueueUserWorkItem`, но для его имитации и передачи данных в запускаемый код можно использовать лямбда-функции (листинг 4.9).

Листинг 4.9. Использование лямбда-функции для параметризации `Task.Run`

```
public void RunInBackground()
{
    for(int i=0;i<10;++i)
    {
        var icopy = i;
        Task.Run(()=>
        {
            Console.WriteLine($"Привет из потока {icopy}");
        });
    }
}
```

Здесь мы использовали лямбда-функцию, чтобы захватить локальные переменные и передать уникальное значение каждой задаче. Обратите внимание, что нам пришлось объявить переменную `icopy` внутри цикла, поскольку иначе все потоки использовали бы ту же переменную `i`, что и цикл `for`, а поскольку для запуска задачи требуется время, к моменту начала их выполнения цикл завершится и все задачи получат конечное значение `i` (в данном случае 10).

Когда использовать `Task.Run`

Используйте `Task.Run` для:

- ◆ кода, использующего `async/await`;
- ◆ коротких задач.

Не используйте `Task.Run` для:

- ◆ синхронных задач, выполняющихся продолжительное время.

В этом примере мы смогли создать отдельную копию `i` для каждого потока, но нередко требуется иметь общие данные, доступные нескольким потокам, поэтому далее поговорим о доступе к одним и тем же переменным из нескольких потоков.

4.2. Доступ к одним и тем же переменным из нескольких потоков

Теперь, зная, как запускать код параллельно, разберемся с возможными последствиями. Большинство программ манипулируют данными в памяти, и проблема с такими данными в многопоточной программе заключается в том, что манипуляции с данными часто не являются одной непрерывной операцией, даже если это всего лишь одна строка кода или даже один оператор.

Возьмем самую простую операцию, которую я смог придумать, — увеличение целого числа на 1:

```
int n=0;  
++n;
```

Строка `++n` выглядит так, будто она выполняет одну операцию. Это всего лишь одна переменная и один оператор, и код, реализующий ее, занимает всего три символа. Сколько различных операций мы можем реализовать всего тремя символами? Однако, как оказывается, это три разные операции.

1. Чтение значения из области памяти, выделенной для переменной `n`, в центральный процессор.
2. Увеличение (инкремент) значения внутри процессора.
3. Запись нового значения из процессора обратно в область памяти, выделенную для переменной `n`.

В однопоточной программе эта последовательность действий выглядит как одна операция, потому что просто невозможно случайно нарушить эту последовательность и внедрить какой-то код, который будет выполняться в середине последовательности. Однако в многопоточной программе такое возможно.

Операции, которые нельзя прервать в многопоточном приложении, обычно потому, что они представляют собой одну аппаратную операцию, называют атомарными. На рис. 4.1 приводится сравнение двукратного увеличения переменной на единицу 1) в однопоточном приложении, где все работает так, как ожидается; 2) в многопоточном приложении на одноядерном процессоре, где система, имитирующая многопоточность, может и будет приостанавливать потоки в самое неподходящее время; 3) и наконец, в многопоточном приложении на многоядерном процессоре, где все действия действительно выполняются параллельно.

Как показано на рис. 4.1, только в однопоточных приложениях или приложениях, которые не имеют общих данных, используемых несколькими потоками, наша простая операция действует как операция, которую нельзя прервать. Во всех других случаях может произойти все что угодно.

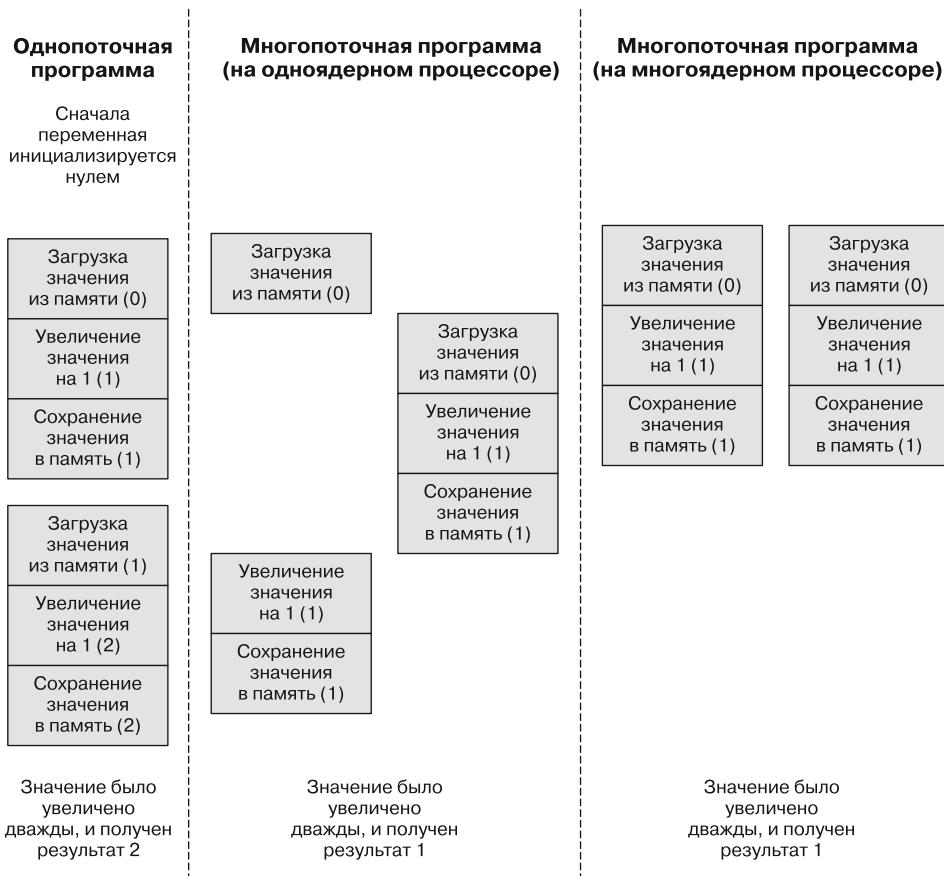


Рис. 4.1. Операции могут прерываться в самое неподходящее время и давать неправильные результаты

Давайте напишем код, демонстрирующий этот аспект: создадим два потока и увеличим в них одну и ту же переменную, а чтобы проблема не осталась незамеченной, в каждом потоке увеличим переменную пять миллионов раз (листинг 4.10).

Листинг 4.10. Получение неверного результата при доступе к общим данным без блокировки

```
public void GetIncorrectValue()
{
    int theValue = 0;

    var threads = new Thread[2];
    for(int i=0;i<2;++i)
```

```
{  
    threads[i] = new Thread(()=>  
    {  
        for(int j=0;j<5000000;++j)  
            ++theValue;  
    });  
    threads[i].Start();  
}  
  
foreach(var current in threads)  
{  
    current.Join();  
}  
Console.WriteLine(theValue);  
}
```

Запуская этот код, мы ожидаем, что он выведет 10 000 000, но, прочитав, что я написал перед этим примером, вы уже знаете, что это не так. Фактически результат будет меняться при каждом новом запуске кода, но чаще всего вы будете видеть значение где-то около 6 000 000.

Как же решить эту проблему?

4.2.1. Отказ от общих данных

Самое простое решение — никогда не использовать общие данные, доступные нескольким потокам. Если каждый поток имеет свой набор переменных, доступных только этому потоку, то у нас никогда не получится прочитать или записать переменную из другого потока, и мы в безопасности.

Такое иногда возможно. Например, если мы пишем сервер, принимающий данные от клиента, вычисляющий что-то исключительно на основе этих данных и возвращающий результаты, то каждый поток может работать, не прикасаясь ни к каким значениям, доступным другим потокам. Однако чаще это невозможно, потому что приложения обычно манипулируют некоторыми общими данными.

А можно ли обойти проблему другим способом, например, не изменяя общих данных?

4.2.2. Неизменяемые общие данные

Если проблема заключается в том, что один поток может обратиться к данным в тот момент, когда другой их изменяет, то проблему можно полностью устранить, если просто никогда не изменять общие данные. Типичным примером может служить веб-сервер, использующий статические данные, которые никогда не изменяются. Такие данные можно читать из параллельных потоков столько раз, сколько потребуется, и это не вызовет никаких проблем.

Для большинства приложений это непросто, но такое решение стало стандартом в функциональных языках и может быть реализовано в C#. Правда, стиль написания кода в этом случае будет отличаться от привычного.

Неизменяемость общих данных может показаться непрактичной для разработчиков, не привыкших к функциональному программированию, но на самом деле это не только возможно, но и является хорошим техническим решением. Единственная проблема — вам придется писать код в совершенно непривычном стиле, отличающемся от обычного стиля, принятого при программировании на C#. Но я не буду продолжать обсуждение этого решения, потому что ему можно посвятить целую книгу (на самом деле такая книга была выпущена в Manning — *Concurrency in .NET*), и вы все равно не будете использовать этот подход, потому что он будет казаться вам чуждым. Однако в .NET есть некоторые встроенные неизменяемые структуры данных, которые мы обсудим в главе 13.

Так мы плавно подходим к стандартному решению — блокировкам и мьютексам.

4.2.3. Использование блокировок и мьютексов

Осталось еще одно решение — синхронизация доступа к общему состоянию: всякий раз, когда потоку требуется получить доступ к общему состоянию, он «блокирует» его, а закончив работу с данными, «снимает» блокировку. Если другой поток попытается заблокировать данные, которые уже были заблокированы, то ему придется подождать, пока данные не будут разблокированы.

В информатике это называется *мьютекс* (*mutex*, сокращенно от *mutual exclusion* — «взаимоисключающий»). В C# есть класс *Mutex*, реализующий поддержку мьютексов операционной системы, и инструкция *lock*, основанная на внутренней реализации .NET. Инструкция *lock* проще в использовании и быстрее (потому что не требует применения системных вызовов), поэтому мы будем использовать ее. Давайте перепишем нашу программу и задействуем в ней блокировки (листинг 4.11).

За инструкцией *lock* следует блок кода, по выходе из которого блокировка автоматически снимается. Благодаря такому решению мы не сможем по случайности забыть снять блокировку. (Блокировка также будет снята, если выход произойдет из-за исключения.)

Кроме того, инструкция *lock* принимает объект. В принципе можно передать любой объект .NET, однако лучшей практикой является использование внутреннего объекта, который предназначен исключительно для целей блокировки и доступен только коду, которому он нужен. Обычно на эту роль назначается приватный член класса.

¹ Терrell Р. Конкурентность и параллелизм на платформе .NET. — СПб.: Питер, 2019.

Листинг 4.11. Добавление блокировок, чтобы избежать проблем из-за одновременного доступа

```
public void GetCorrectValue()
{
    int theValue = 0;
    object theLock = new Object();

    var threads = new Thread[2];
    for(int i=0;i<2;++i)
    {
        threads[i] = new Thread(()=>
        {
            for(int j=0;j<50000000;++j)
            {
                lock(theLock)
                {
                    ++theValue;
                }
            }
        });
        threads[i].Start();
    }

    foreach(var current in threads)
    {
        current.Join();
    }
    Console.WriteLine(theValue);
}
```

Блокирует доступ к данным на время их изменения

Инструкция `lock` в этом примере помогает программе получить правильный результат. Примитивы синхронизации, такие как `lock` и `Mutex`, являются неотъемлемой частью многопоточного программирования. Однако те же примитивы влекут за собой некоторые новые виды ошибок, самой серьезной из которых является взаимная блокировка.

Зачем инструкции `lock` нужен объект

В .NET 8 и более ранних версиях лучше использовать объект типа `Object` (который также может быть получен с помощью ключевого слова `object`), потому что обычно такой объект не используется ни для чего другого и тип `Object` имеет самые низкие накладные расходы среди всех объектов ссылочного типа.

В .NET 9 и более поздних версиях лучше использовать объект `System.Threading.Lock`. Новый класс `Lock` в инструкции `lock` выглядит более естественно (его имя прямо говорит, что это блокировка), а его реализация в новых версиях может оказаться эффективнее, чем `Object`.

Использование инструкции `lock` с объектом `Object` по-прежнему поддерживается в .NET 9 и последующих версиях. В этой книге для обратной совместимости во всех примерах будут использоваться объекты `Object`, а не `Lock`.

4.2.4. Взаимоблокировки

Взаимоблокировка — это ситуация, когда поток или несколько потоков блокируются в ожидании чего-то, что никогда не произойдет. Самый простой пример — когда один поток заблокировал мьютекс A и ждет, когда освободится мьютекс B, в это время второй поток заблокировал мьютекс B и ждет, когда освободится мьютекс A. В результате поток A ждет завершения потока B, а тот ждет завершения потока A, который, как мы уже отметили, ждет завершения потока B, который, как мы сказали, ждет завершения потока A, и т. д. — оба потока останавливаются навечно (рис. 4.2).

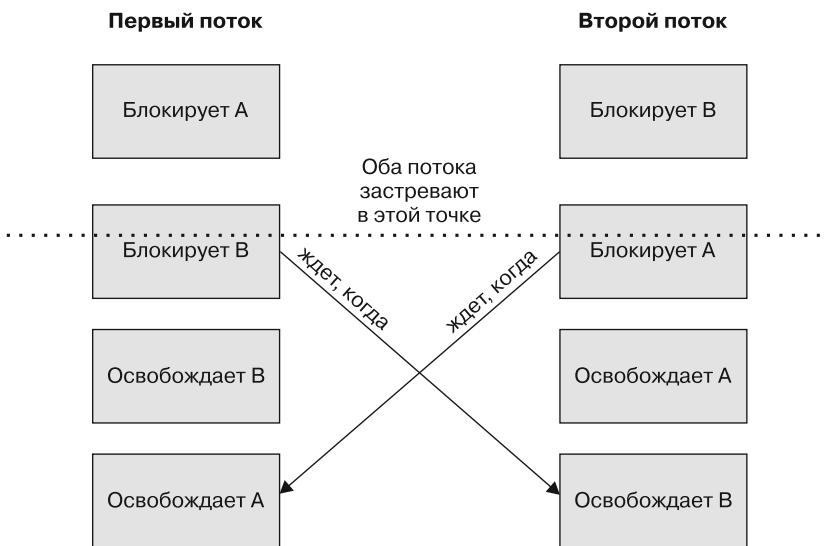


Рис. 4.2. Взаимоблокировка возникает, когда один поток заблокировал мьютекс A и ждет освобождения мьютекса B, в то время как другой поток заблокировал мьютекс B и ждет освобождения мьютекса A. Создается ситуация, когда поток A ждет завершения потока B, а поток B ждет завершения потока A, что приводит к бесконечному циклу ожидания

Теперь вы понимаете, почему лучше всего в инструкции `lock` использовать приватный объект, доступный только коду, которому он нужен. В противном случае сторонний код мог бы случайно или намеренно заблокировать тот же объект в неожиданный для нас момент и вызвать состояние взаимоблокировки. При использовании приватного объекта все равно есть риск вызвать взаимоблокировку, но, когда все стороны контролируются нами, у нас есть методы предотвращения подобных ситуаций. В этой книге есть целая глава, посвященная взаимным блокировкам и другим типичным проблемам многопоточности, а также методам их предотвращения.

Вот почему я сказал ранее, что `Thread.Suspend`, `Thread.Resume` и `Thread.Abort` очень опасны. Представьте, что вы написали очень умную систему управления работой вашей программы, которая использует `Suspend` и `Resume` для управления потоками. С точки зрения потока вызов `Suspend` может произойти в любой момент, например, когда поток находится в середине операции выделения памяти и удерживает блокировку внутри диспетчера памяти. Обычно эта блокировка полностью безопасна, потому что быстро освобождается, и код никогда ничего не ждет, удерживая блокировку, но теперь вы можете по неосторожности заставить код диспетчера памяти ждать, пока не будет вызван метод `Resume`. В этот период никакой другой поток не сможет выделить память, включая поток, который должен вызвать `Resume`. Если этот поток попытается выделить память (очень распространенная операция), то возникнет состояние взаимоблокировки.

Взаимоблокировка может возникнуть даже без использования блокировок или мьютексов, например, когда два потока совместно используют другие ресурсы, которыми могут быть и сами потоки. Это особенно часто встречается в потоках специального назначения. Наиболее распространенным потоком специального назначения является поток пользовательского интерфейса приложения.

4.3. Замечания о приложениях с графическим пользовательским интерфейсом

Во всех технологиях настольных приложений для Windows (WinForms, WPF, UWP и WinAPI) доступ к окнам и элементам управления пользовательского интерфейса возможен только из потока, который их создал. Попытка обратиться к элементам пользовательского интерфейса из другого потока может привести к неверным результатам, ошибкам и исключениям, в зависимости от используемой технологии. При запуске программы вы настраиваете главное окно, а затем вызываете `Dispatcher.Run` или `Application.Run` (в зависимости от технологии). Обычно этот вызов производится в шаблонном коде, сгенерированном средой Visual Studio. После вызова `Run` поток входит в цикл ожидания событий пользовательского интерфейса и, при необходимости, вызывает ваш код для их обработки. Если заблокировать поток или выполнять какие-либо продолжительные вычисления в обработчиках событий пользовательского интерфейса, то вы лишите поток возможности обрабатывать события пользовательского интерфейса, и он зависнет, пока ваш обработчик не завершится.

Вы можете воспользоваться тем фактом, что поток пользовательского интерфейса ожидает и обрабатывает события, и внедрить свои события. Это позволяет другим потокам запрашивать выполнение кода в потоке пользовательского интерфейса, что очень полезно, потому что иначе было бы сложно обновить в пользовательском интерфейсе результаты вычислений, выполняемых в фоновых потоках (поскольку единственный поток, который имеет прямой доступ

к пользовательскому интерфейсу, — это поток пользовательского интерфейса). Это можно сделать с помощью методов `Control.BeginInvoke` или `Control.Invoke` в WinForms, `Dispatcher.BeginInvoke` или `Dispatcher.Invoke` в WPF или с помощью обобщенного интерфейса `SynchronizationContext`.

В типичном обработчике событий, который вызывается в ответ на событие пользовательского интерфейса, например в ответ на нажатие кнопки, вы можете использовать класс `Thread` или пул потоков для запуска кода в фоновом режиме, который по завершении вычислений вызовет `BeginInvoke`, чтобы поток пользовательского интерфейса обновил результаты в пользовательском интерфейсе.

Но будьте осторожны с кодом, который выполняется в потоке пользовательского интерфейса, так как есть риск попасть в ситуацию, когда код может блокироваться в ожидании какого-то события, происходящего в ответ на вызов `Invoke`/`BeginInvoke` (или в некоторых случаях в ответ на выполнение инструкции `await`, как будет показано в следующей главе), из-за того что код, переданный в `Invoke/BeginInvoke`, никогда не запустится, потому что поток заблокирован в ожидании. Это создаст ситуацию взаимоблокировки и зависания пользовательского интерфейса.

4.4. Ожидание другого потока

Иногда бывает нужно, чтобы один поток дождался, когда другой поток что-то сделает. Если нужно дождаться, когда созданный нами поток завершил работу, то можно использовать `Thread.Join` (как обсуждалось ранее в этой главе). Но если другой поток должен продолжить работу после вычисления каких-то результатов, то мы не сможем использовать `Thread.Join` и должны выбрать какой-то механизм передачи сигналов между потоками. Этот механизм называется `ManualResetEventSlim` и является более новой, быстрой и простой реализацией `ManualResetEvent`. Как и все классы, заканчивающиеся на `Slim`, он лишен некоторой функциональности, в основном поддержки межпроцессных взаимодействий, ради улучшенной производительности.

`ManualResetEventSlim` работает как шлюз. Когда событие отсутствует, шлюз закрыт, и любой поток, вызывающий его метод `Wait`, будет ждать. Когда событие происходит, шлюз открывается, и метод `Wait` немедленно возвращает управление любому потоку, ожидающему в данный момент, и всем потокам, которые вызовут этот метод в будущем (пока событие не будет сброшено).

Конструктор `ManualResetEventSlim` принимает один параметр. Значение `false` в параметре создает закрытый шлюз (событие не произошло), а значение `true` — открытый шлюз (событие произошло). Метод `Set` сообщает шлюзу, что событие произошло, и открывает его, а метод `Reset` сбрасывает событие и закрывает шлюз. Напишем код, в котором один поток ожидает события от другого потока с помощью `ManualResetEventSlim` (листинг 4.12).

Листинг 4.12. Ожидание другого потока

```

var myEvent = new ManualResetEventSlim(false); ← Создает событие
var threadWeWaitFor = new Thread(()=>
{
    Console.WriteLine("Делаем что-то");
    Thread.Sleep(5000);
    Console.WriteLine("Выполнено");
    myEvent.Set(); ← Открывает шлюз
});
var waitingThread = new Thread(()=>
{
    Console.WriteLine("Ожидаем другой поток, чтобы что-то сделать");
    myEvent.Wait(); ← Ждет, когда шлюз откроется
    Console.WriteLine("Другой поток закончил выполнение, продолжаем");
});
threadWeWaitFor.Start();
waitingThread.Start();

```

В этом примере мы создаем два потока. Первый имитирует выполнение длительной операции, приостанавливаясь на время, а затем устанавливает событие. Второй поток использует событие для ожидания завершения вычислений в первом потоке.

4.5. Другие методы синхронизации

В дополнение к двум методам синхронизации потоков, обсуждавшимся выше в этой главе, — инструкции `lock` и классу `ManualResetEventSlim` — в .NET имеется обширная коллекция других примитивов синхронизации. Каждый из них был написан не просто так и чрезвычайно полезен в конкретных обстоятельствах. И все они имеют одну общую черту: в большинстве случаев их следует избегать.

Инструкция `lock` — самый простой и безопасный примитив синхронизации в .NET, хотя и при ее использовании может возникнуть проблема взаимоблокировки, о которой мы говорили выше, и множество других проблем, которые рассмотрим в главе 7. По этой причине я рекомендую придерживаться инструкции `lock`, а более продвинутые и более опасные механизмы использовать только в случае крайней необходимости, то есть только после того, как будет проведено профилирование кода и обнаружится, что узкое место можно устраниить, только переключившись на другой способ синхронизации потоков.

Например, возьмем широко используемый инструмент синхронизации потоков — класс `Interlocked`, предлагающий потокобезопасные операции, которые не требуют блокировки. Казалось бы, этот волшебный класс может решить все наши проблемы, однако у него тоже есть свои нюансы.

- Он поддерживает только ограниченный набор операций, а именно: приращение, уменьшение и сложение, а также побитовые операции И и ИЛИ для некоторых целочисленных типов (`int`, `uint`, `long` и `ulong`).

- Все остальные операции могут быть реализованы с применением методов `Exchange` и `CompareExchange`, которые должны использоваться особым образом (несколько примеров вы найдете в главе 13).
- Он защищает операцию, а не переменную. Если кто-то обращается к переменной, «защищенной» блокирующей операцией, с помощью других инструментов, кроме членов класса `Interlocked`, то все гарантии отменяются и код перестает быть потокобезопасным, в том числе тот, что использует класс `Interlocked`. Даже если вам просто нужно прочитать переменную, не изменяя ее, то вы должны использовать `Interlocked.Read` и не обращаться к переменной напрямую.
- Значение, которое возвращают методы `Interlocked`, гарантированно является правильным на момент вызова метода, но к моменту использования этого значения, даже если это происходит в той же строке кода, значение может устареть.
- Потокобезопасным является только один вызов метода класса `Interlocked`. Если, к примеру, дважды вызвать `Interlocked.Increment`, чтобы увеличить две переменные, то другой поток (даже тот, который тоже использует класс `Interlocked`) может прочитать или изменить любую из переменных между этими двумя операциями. Это особый случай проблемы, известной как «комбинирование потокобезопасных операций» (редко приводит к созданию потокобезопасной операции), которую мы обсудим в главе 7.
- Члены класса `Interlocked` работают быстрее, чем инструкция `lock`, но они могут оказаться значительно медленнее обычных операций (таких как `+=`, `++`, `--`, `&` и `|`).

Учитывая все вышесказанное (и не только), проще и безопаснее использовать инструкцию `lock` и прибегать к классу `Interlocked` (с большой осторожностью) только в коде, который очень чувствителен к производительности. То же касается всех остальных примитивов синхронизации, которые упоминались, но не обсуждались, — используйте их только в случае крайней необходимости, потому что они сложны и не так безопасны и просты в использовании, как инструкция `lock`.

4.6. Параметры потока

Рассказывая о классе `Thread`, я говорил, что изменять параметры потоков допускается только в том случае, если эти потоки создаются вашим кодом с помощью класса `Thread` (также допускается изменять настройки главного потока, если вы пишете приложение, а не библиотеку). Никогда не следует изменять настройки пула потоков или потока, созданного другим компонентом, потому что компонент, создавший поток, может полагаться на свои настройки потока и их изменение помешает работе компонента.

Обратите внимание, что в асинхронном коде бессмысленно изменять настройки, потому что всякий раз, когда выполняется `await` (или `ContinueWith`), выполнение кода может продолжиться в другом потоке с иными настройками. Далее мы рассмотрим настройки, которые можно изменить с помощью класса `Thread`, в порядке их полезности.

4.6.1. Режим выполнения потока в фоне

Это наиболее часто используемая настройка. Она устанавливается с помощью свойства `Thread.IsBackground` и влияет на выбор момента, когда приложение сможет завершить работу. Это произойдет, когда завершатся все нефоновые потоки. То есть если с помощью класса `Thread` запустить поток (и не установить его свойство `IsBackground`), то, когда метод `Main` завершится, приложение продолжит работать, пока не завершится поток. Если созданный вами поток не должен задерживать завершение приложения, то просто присвойте его свойству `IsBackground` значение `true`.

Это свойство должно устанавливаться до вызова `Thread.Start`. Оно никак не влияет на принудительное завершение приложения (например, вызовом `Environment.Exit` или `Environment.FailFast`) или если приложение завершается из-за необработанного исключения.

4.6.2. Язык и локаль

Изменить язык и локаль потока можно с помощью свойства `Thread.CurrentCulture`, которое влияет на форматирование значений, в основном чисел и дат (если в метод форматирования не передается конкретный объект `CultureInfo`). Оно также влияет на выбор языковых ресурсов в приложениях с графическим интерфейсом. По умолчанию заданы язык и форматирование, используемые операционной системой.

Используйте это свойство, только если ваше приложение позволяет пользователю менять язык. В иных случаях предпочтительнее опираться на настройки операционной системы.

4.6.3. СОМ-апартаменты

Для управления типом СОМ-апартаментов потока можно задействовать методы `Thread.SetApartmentState` и `Thread.TrySetApartmentState`. Это относится только к приложениям, использующим СОМ-компоненты в потоках, созданных с помощью класса `Thread` (для настройки главного потока следует использовать атрибут `STAThread` или `MTAThread` метода `Main`).

СОМ-компоненты — обширная тема, обсуждение которой выходит за рамки этой книги. Но для читателей, которым повезло не использовать СОМ-компоненты,

кратко поясню: дело в том, что в COM есть концепция *типа апартаментов* и большинство объектов COM могут работать только с определенным типом апартаментов. Чтение или настройка COM-апартаментов поддерживается только в Windows, поскольку другие операционные системы не используют COM.

4.6.4. Текущий пользователь

Это свойство фактически (неофициально) устарело. Для присоединения идентификатора и разрешений к потоку можно использовать статическое свойство `Thread.CurrentPrincipal`. Однако изменение этого свойства не изменяет разрешений потока на уровне операционной системы. Это просто место, где вы (или библиотеки, которые вы используете) можете хранить информацию о пользователе для вашей собственной системы разрешений.

В ASP.NET Classic (в .NET Framework 4.x и в более ранних версиях) при использовании встроенной системы аутентификации текущая информация о веб-пользователе сохранялась в свойстве `CurrentPrincipal`. В ASP.NET Core (.NET Core и .NET 5 и более поздних версиях) ситуация изменилась: в новой версии информация о текущем пользователе хранится только в `HttpContext.User`.

4.6.5. Приоритет потока

Изменение приоритета потока довольно опасно, и его следует избегать. Если не проявить должной осторожности, то изменение приоритета потока может привести к снижению производительности и/или взаимоблокировкам.

Проблема в том, что слишком легко попасть в ситуацию, когда поток с высоким приоритетом ждет освобождения ресурса, удерживаемого потоком с низким приоритетом, но этот низкоприоритетный поток не может освободить ресурс, потому что все процессорное время отдается высокоприоритетному потоку.

Управление приоритетом обычно требуется для некоторых видов системного программирования, но в обычных приложениях никогда не следует изменять приоритеты потоков. Приоритет контролируется свойством `Thread.Priority`. Системе разрешено игнорировать установленный вами приоритет.

Итоги главы

- Вы можете запускать несколько задач параллельно. Каждая из этих задач называется потоком.
- В начальный момент программа запускает один поток, в котором выполняется метод `Main`. Этот поток называется главным потоком.
- Вы можете создавать свои потоки, полностью подконтрольные вам. Для этого с помощью класса `Thread` нужно создать объект `Thread`, изменив его параметры при необходимости, и вызвать метод `Thread.Start`, чтобы запустить его.

- Пул потоков — это коллекция потоков, управляемых системой, которые можно использовать для параллельного запуска своего кода. Пул потоков оптимизирован для выполнения коротких задач и может создавать новые потоки по мере необходимости.
- Традиционно запуск кода в пуле потоков производится с помощью метода `ThreadPool.QueueUserWorkItem`.
- Более простой и дружественный к `async/await` способ запуска кода в пуле потоков — вызов метода `Task.Run`.
- Главный поток и потоки, созданные вами с помощью класса `Thread`, — это единственные потоки, которые полностью находятся под вашим контролем и могут настраиваться любым желаемым вами способом. Вы никогда не должны перенастраивать потоки, созданные другими компонентами, особенно потоки, управляемые пулом потоков.
- При доступе к общим данным из нескольких потоков необходимо использовать блокировку, в противном случае разные потоки могут искажать данные, записанные другими потоками, и приводить к неверным результатам.
- То же самое относится и к чтению общих данных. Если не использовать блокировки для синхронизации операций чтения и записи, то есть риск получить устаревшие данные и даже промежуточные результаты операций записи.
- В приложениях с графическим пользовательским интерфейсом имеется отдельный поток, управляющий пользовательским интерфейсом, который так и называют — «поток пользовательского интерфейса». Обычно это главный поток, но при необходимости это может быть другой поток. Поток пользовательского интерфейса — единственный, который может обращаться к окнам и другим элементам управления пользовательского интерфейса.
- Избегайте блокировок потока пользовательского интерфейса, потому что это влечет зависание пользовательского интерфейса.

async/await и многопоточность

В этой главе

- ✓ Совместное использование `async/await` и многопоточности.
- ✓ Выполнение кода после `await`.
- ✓ Использование блокировок с `async/await`.

Асинхронное программирование — это выполнение действий (например, чтение файла или ожидание получения данных по сети), которые не требуют участия процессора и происходят в фоновом режиме, при этом процессор используется для выполнения каких-то других действий. Многопоточность — это выполнение действий с использованием или без использования процессора в фоновом режиме, при котором процессор задействуется для выполнения каких-то других действий. Эти две вещи очень похожи, и для работы с ними мы применяем одни и те же инструменты.

В главе 3 мы говорили об `async/await`, но не упоминали потоки. В частности, мы намеренно умолчали о том, где выполняется обратный вызов, переданный в `ContinueWith`. В главе 4 мы также говорили о многопоточности и почти не упоминали `async/await`. В этой главе мы свяжем эти два понятия вместе.

5.1. Асинхронное программирование и многопоточность

Демонстрацию связи между асинхронным программированием и многопоточностью мы начнем с метода, который параллельно читает десять файлов, используя асинхронные операции, а затем ждет завершения всех операций чтения. Ради интереса не будем делать метод асинхронным, а просто используем асинхронные операции (листинг 5.1).

Листинг 5.1. Чтение десяти файлов

```
public void Read10Files()
{
    var tasks = new Task[10];
    for(int i=0;i<10;++i)
    {
        tasks[i] = File.ReadAllBytesAsync($"{i}.txt");
    }
    Task.WaitAll(tasks);
}
```

Очевидно, что это асинхронное программирование. Чтение файла — это пример работы из учебников, основная часть которой выполняется без участия процессора (и да, здесь я просто проигнорировал данные, полученные из файла, потому что главная цель примера — демонстрация механики работы `Task` и асинхронных операций). Но как бы выглядел код параллельного вычисления десяти значений? Для простоты будем выводить текст, что значение вычисляется и ожидается получение результатов. Используем метод `Task.Run`, который запустит наш код в пуле потоков (см. главу 4) (листинг 5.2).

Листинг 5.2. Вычисление десяти значений

```
public void Compute10Values()
{
    var tasks = new Task[10];
    for(int i=0;i<10;++i)
    {
        tasks[i] = Task.Run(()=>Console.WriteLine("Вычисляем..."));
    }
    Task.WaitAll(tasks);
}
```

Здесь изменена буквально одна строка, да и то не полностью. Этот пример показывает, что те же инструменты, которые применяются для запуска асинхронных операций, можно использовать и для организации многопоточного выполнения. Сделаем еще один шаг и добавим многопоточность для параллельного чтения файлов (листинг 5.3).

Листинг 5.3. Чтение десяти файлов с использованием многопоточности

```
public void Read10Files()
{
    var tasks = new Task[10];
    for(int i=0;i<10;++i)
    {
        var icopy = i;
        tasks[i] = Task.Run(()=>File.ReadAllBytes($"'{icopy}'.txt"));
    }
    Task.WaitAll(tasks);
}
```

Здесь потребовалось объявить локальную переменную внутри цикла. Иначе все потоки использовали бы одну и ту же переменную *i* и к моменту их запуска цикл уже завершился бы, а переменная *i* имела бы конечное значение 10. Это заставило бы все потоки предпринять попытку прочитать файл 10.txt, которая потерпела бы неудачу, потому что доступны только файлы 0.txt–9.txt.

В остальном код выглядит почти так же, как в листинге 5.1, и делает то же самое, но при этом использует больше вычислительных ресурсов, потому что запускает до десяти отдельных потоков (в зависимости от того, насколько быстро система может читать файлы). Более того, каждый из потоков приостанавливается в ожидании получения файла с жесткого диска, тогда как код в листинге 5.1 использует только один поток, ожидающий чтения всех файлов.

Но настоящая программа не будет читать файлы и игнорировать данные, как в нашем примере. Настоящая программа будет не только читать файл, но и что-то делать с его содержимым. Изменим последний пример, чтобы он тоже что-то делал (в данном случае просто выведем в консоль сообщение, уведомляющее о том, что что-то делается с содержимым файла) (листинг 5.4).

Листинг 5.4. Чтение десяти файлов и обработка данных

```
public void Process10Files()
{
    var tasks = new Task[10];
    for(int i=0;i<10;++i)
    {
        var icopy = i;
        tasks[i] = Task.Run(()=>
        {
            File.ReadAllBytes($"'{icopy}'.txt");
            Console.WriteLine("Выполнение действий с содержимым файла");
        });
    }
    Task.WaitAll(tasks);
}
```

Этот код будет использовать до десяти потоков из пула потоков (потому что, напомню, `Task.Run` использует пул потоков), возможно, создавая новые потоки, если в пуле потоков их окажется недостаточно, и сразу же их блокируя в ожидании получения данных, прочитанных с жесткого диска. Посмотрим, что получится, если то же самое запрограммировать с применением только асинхронных операций (листинг 5.5).

Листинг 5.5. Асинхронное чтение десяти файлов и обработка данных

```
public void Process10Files()
{
    var tasks = new Task[10];
    for(int i=0;i<10;++i)
    {
        var icopy = i;
        tasks[i] = Task.Run(async ()=>
        {
            await File.ReadAllBytesAsync($"{icopy}.txt");
            Console.WriteLine("Выполнение действий с содержимым файла");
        });
    }
    Task.WaitAll(tasks);
}
```

Этот код выглядит точно так же, как и предыдущий, только вместо `File.ReadAllText` вызывается `File.ReadAllBytesAsync`, да еще добавились ключевые слова `async` и `await`, однако происходящее во время выполнения сильно отличается. Вместо того чтобы получить десять потоков и сразу заблокировать их на все время выполнения чтения, этот код возьмет поток из пула, запустит в нем асинхронную операцию чтения, затем освободит поток и использует механизм обратного вызова, о котором рассказывалось в главе 3. То есть программа будет использовать небольшое количество потоков для запуска операций чтения (возможно, даже один; это зависит от текущей нагрузки на компьютер и состояния пула потоков), а затем вообще не будет использовать потоки во время ожидания. И только после поступления данных она начнет использовать десять потоков (то есть только когда для них появится работа).

На самом деле это решение заметно лучше предыдущего, потому что файлы читаются с одного и того же жесткого диска и код будет получать содержимое файлов по очереди, а не всех сразу (так как имеется только один жесткий диск с одним подключением к шине данных на материнской плате), и каждая задача подхватит поток только после того, как ее данные станут доступны. Это означает, что, скорее всего, программа никогда не будет использовать десять потоков одновременно (но мы не можем утверждать этого, потому что многопоточное программирование по своей сути непредсказуемо).

92 Глава 5. async/await и многопоточность

Двумя абзацами выше я немного склонил, сказав, что программа вообще не будет использовать потоки во время ожидания чтения файлов. На самом деле она задействует один поток — поток, который вызвал `Process10Files` и ждет завершения обработки всех файлов. Мы можем это исправить, просто сделав сам `Process10Files` асинхронным (листинг 5.6).

Листинг 5.6. Объявление вызывающей функции async

```
public async Task Process10Files() ← Метод объявляется асинхронным
{
    var tasks = new Task[10];
    for(int i=0;i<10;++i)
    {
        var icopy = i;
        tasks[i] = Task.Run(async ()=>
        {
            await File.ReadAllBytesAsync($"{icopy}.txt");
            Console.WriteLine("Выполнение действий с содержимым файла");
        });
    }
    await Task.WhenAll(tasks); ← Замена вызова WaitAll на WhenAll
}
```

В этом случае программа освободит поток, вызвавший `Process10Files`, и вообще не будет использовать потоки, пока не будет прочитан хотя бы один файл.

Освободив все потоки на время ожидания получения данных, мы должны продолжить работу, когда данные наконец поступят, но мы не сможем этого сделать, потому что освободили поток. Так где же выполнится наш код после вызова `await`?

5.2. Где выполняется код после вызова await

В главе 2 я говорил, что использование ключевого слова `await` эквивалентно вызову `Task.ContinueWith`, поэтому код

```
var buffer = await File.ReadAllBytesAsync("somefile.bin");
Console.WriteLine(buffer[0]);
```

будет преобразован компилятором в следующий:

```
File.ReadAllBytesAsync("somefile.bin").ContinueWith(task=>
{
    var buffer = task.Result;
    Console.WriteLine(buffer[0]);
});
```

Я также упоминал, что это упрощенное представление и что `await` немного сложнее. Теперь пришло время посмотреть, что именно делает `await`.

`ContinueWith` запускает обратный вызов в пуле потоков, как и `Task.Run`. Технически `ContinueWith` имеет версию с параметрами, определяющими, как запустить переданный ему обратный вызов, но я не буду вдаваться в подробности, потому что `await` делает это автоматически, а `ContinueWith` очень редко используется напрямую в приложениях.

`await` пытается запустить код в том же потоке, который его вызвал, поэтому в большинстве случаев нет необходимости задумываться о возможности смены потока. Если `await` не может использовать тот же поток, то вместо этого он будет использовать пул потоков. Ниже перечислены конкретные правила, которым он при этом следует.

- Если `await` используется в потоке пользовательского интерфейса приложения WinForms, WPF или UWP, то код после `await` будет выполняться в том же потоке.
- Если `await` используется при обработке запроса в приложении ASP.NET Classic (.NET Framework 4.8 и в более ранних версиях), то код после `await` будет выполняться в том же потоке.
- Если код или используемый фреймворк изменяют `SynchronizationContext` или `TaskFactory` текущего потока (мы поговорим о них позже в этой книге), то `await` будет использовать их. Именно так фреймворки, упомянутые в предыдущих пунктах, управляют поведением `await`, но за исключением фреймворков пользовательского интерфейса это случается крайне редко.
- Во всех остальных случаях код после `await` будет запущен в пуле потоков. Вот несколько распространенных примеров:
 - если код, вызывающий `await`, выполняется в потоке из пула, то код после `await` тоже будет запущен в потоке из пула, однако он может быть запущен в другом потоке;
 - это также относится к приложениям ASP.NET Core (.NET Core или .NET 5.0 и в более поздних версиях), потому что ASP.NET Core всегда использует пул потоков;
 - если `await` вызывается в главном потоке в приложении без пользовательского интерфейса, то код после `await` тоже будет запущен в пуле потоков, а не в главном потоке; система будет поддерживать главный поток (и все приложение) активным до тех пор, пока не завершится задача;
 - если `await` вызывается в потоке, созданном с помощью класса `Thread`, то поток, выполняющий метод, который был передан конструктору `Thread`, завершится, а код после `await` будет запущен с использованием пула потоков.

Эти правила применяются, только если `await` должен ждать завершения операции. Если к моменту вызова `await` операция уже завершилась, то почти

94 Глава 5. async/await и многопоточность

во всех случаях код просто продолжит работу в текущем потоке. Наиболее распространенная ситуация, когда такое происходит, — если ожидается завершение метода, не содержащего асинхронных операций. Например, метод в листинге 5.7 посылает запрос на сервер для получения результата и использует очень простой кэш, чтобы избежать выполнения повторных долгостоящих сетевых вызовов, если результат уже был получен ранее.

Листинг 5.7. Получение значения с сервера с кэшированием; небезопасно в многопоточной среде

```
// Этот метод не является потокобезопасным, корректную версию вы увидите далее
private Dictionary<string, string> _cache = new();
public async Task<string> GetResult(string query)
{
    if(_cache.TryGetValue(query, out var cacheResult))
        return cacheResult; ← Если результат доступен в кэше, вернуть его
    var http = new HttpClient();
    var result = await http.GetStringAsync( | Обращается к серверу,
        "https://example.com?"+query); | чтобы получить результат
    _cache[query] = result; ← Сохраняет результат
    return result;
}
```

Этот метод сначала проверяет, присутствует ли строка запроса в кэше. Если такой запрос уже отправлялся, то возвращается результат из кэша без выполнения асинхронной операции. В противном случае выполняется асинхронный HTTP-запрос, чтобы получить результат с сервера. Затем результат сохраняется в кэше.

При первом вызове метода для данного запроса он вернет объект задачи `Task`, завершения которой нужно дождаться, но при последующих попытках выполнить тот же запрос метод вернет объект `Task`, представляющий уже завершившуюся задачу, и асинхронная операция не будет выполнена. Для демонстрации напишем код, который вызывает этот метод из потока, созданного с помощью класса `Thread` (листинг 5.8).

Листинг 5.8. Вызов `GetResult` из потока, созданного с помощью класса `Thread`

```
var thread = new Thread(async () =>
{
    var result = await GetResult("my query");
    DoSomething(result); ← В каком потоке будет
}); | выполняться этот код?
thread.Name = "Query thread";
thread.Start();
```

Код создает поток, который вызывает метод `GetResult` из листинга 5.7, а затем что-то делает с результатом. Одной из причин, почему здесь используется класс

`Thread`, является возможность изменить свойства потока. В данном случае я изменил имя потока. Имя потока — это просто строка, прикрепленная к потоку. Ее можно увидеть в окне со списком потоков в Visual Studio, чтобы быстро идентифицировать поток и понять его назначение. Это свойство не влияет на работу потока.

Если этот код выполняется первый раз, то он вызовет `GetResult("my query")`. Поток завершится, потому что, как рассказывалось в главе 3, `await` регистрирует код для последующего выполнения, а затем возвращает управление вызывающей стороне подобно оператору `return`, а позже, когда сервер вернет ответ, `DoSomething` запустится в потоке из пула потоков, а не в нашем именованном потоке. Напротив, если результат для "my query" уже имеется в кэше, то код продолжит выполняться в именованном потоке, как будто `await` там не было.

Теперь посмотрим, как сделать метод `GetResult` из листинга 5.7 потокобезопасным.

5.3. Блокировки и `async/await`

Проблема метода `GetResult` из листинга 5.7 заключается в том, что он, скорее всего, будет работать в многопоточной среде (о чем говорит инструкция `await`), но не является потокобезопасным, потому что небезопасно обращаться к словарю `Dictionary< TKey, TValue >` (как для изменения, так и для чтения), пока он изменяется другим потоком. Код в листинге 5.7 изменяет словарь, не защищая его от параллельного доступа. К счастью, в предыдущей главе мы познакомились с инструкцией `lock`. Но, к сожалению, если просто заключить все тело метода в инструкцию `lock`, он просто не скомпилируется (листинг 5.9).

Листинг 5.9. Получение значения с сервера с кэшированием под защитой блокировки

```
private Dictionary<string, string> _cache = new();
private object _cacheLock = new();
public async Task<string> GetResult(string query)
{
    lock(_cacheLock)
    {
        if(_cache.TryGetValue(query, out var cacheResult))
            return cacheResult;
        var http = new HttpClient();
        var result =
            await http.GetStringAsync(
                "https://example.com?"+query); | Ошибка
                | компилятора
            _cache[query] = result;
    }
    return result;
}
```

Этот код не компилируется, потому что инструкцию `await` запрещено использовать внутри блока `lock`. Существуют две проблемы — одна концептуальная и одна практическая.

- Концептуальная проблема заключается в том, что вызов `await` освобождает поток и возвращает его в пул, поэтому мы даже не знаем, какой код будет запущен в этом потоке. Это проблема, потому что, как говорилось в предыдущей главе, запуск кода, который не контролируется нами, при удерживаемой блокировке может привести к состоянию взаимоблокировки.
- Практическая проблема состоит в том, что код после `await` может выполняться в другом потоке, тогда как реализация инструкции `lock` позволяет снять блокировку, только если это происходит в том же потоке, который ее установил.

Как решить эту проблему? Достаточно реорганизовать код так, чтобы вызов `await` производился вне блока `lock`. В этом примере нет необходимости удерживать блокировку во время выполнения HTTP-запроса, достаточно защитить доступ к кэшу до и после вызова (листинг 5.10).

Листинг 5.10. Снятие блокировки на время ожидания

```
private Dictionary<string, string> _cache = new();
private object _cacheLock = new();
public async Task<string> GetResult(string query)
{
    lock(_cacheLock) ← | Установить блокировку
    {
        if(_cache.TryGetValue(query, out var cacheResult))
            return cacheResult;
    }
    var http = new HttpClient();
    var result = await http.GetStringAsync( | Асинхронный
        "https://example.com?"+query); | HTTP-запрос
    lock(_cacheLock) ← | Установить блокировку
    {
        _cache[query] = result;
    }
    return result;
}
```

В этом коде мы решили проблему с ошибкой компиляции, поместив инструкцию `lock` только код доступа к кэшу, но нам пришлось снять блокировку в середине метода. Попытка одновременно запустить метод, который полностью защищен блокировкой, из двух потоков привела бы к последовательности, показанной на рис. 5.1.

Один из параллельных вызовов запускается первым. Он проверяет кэш, не находит кэшированного результата, выполняет HTTP-запрос и обновляет кэш, все это время находясь внутри блока `lock`. Другой вызов запустится вторым после

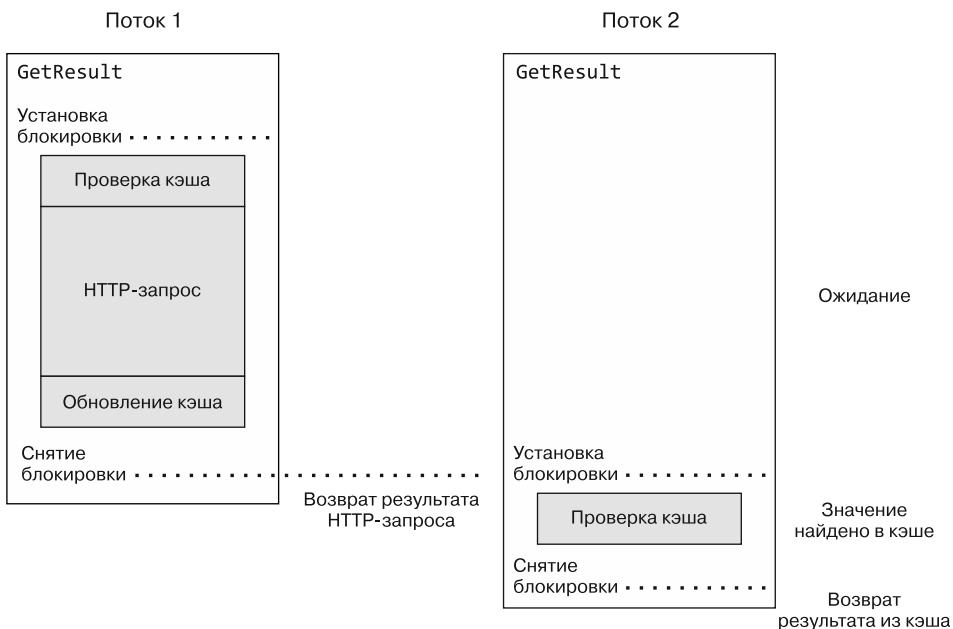


Рис. 5.1. Последовательность выполнения двух вызовов метода, все тело которого защищено блокировкой

обновления кэша и вернет кэшированное значение. Но та же попытка запустить версию метода, которая фактически компилируется, дает последовательность, показанную на рис. 5.2.

Один из параллельных вызовов запускается первым, внутри первого блока `lock` проверяет кэш и не находит кэшированного результата. Затем он снимает блокировку, и в этот момент запускается другой вызов, который тоже проверяет кэш и тоже не находит кэшированного результата, потому что первый HTTP-запрос еще не завершился. В какой-то момент первый HTTP-запрос завершится, и первый поток обновит кэш. Чуть позже завершится второй HTTP-запрос, и второй поток перезапишет значение в кэше (вот почему для обновления кэша используется оператор `[]`, а не `Add` — `Add` выдал бы исключение).

Это простая форма кэширования, которая очень хорошо подходит для любого длительного процесса (как синхронного, так и асинхронного), если для одних и тех же входных данных всегда возвращается одно и то же значение. В этом случае мы готовы смириться с потенциальным падением производительности при многократном запуске длительного процесса (HTTP-запроса в этом примере) до того, как первый запуск завершится и кэш будет заполнен. Однако этот подход к кэшированию не подойдет в случаях, если по каким-то причинам потенциальное падение производительности неприемлемо.

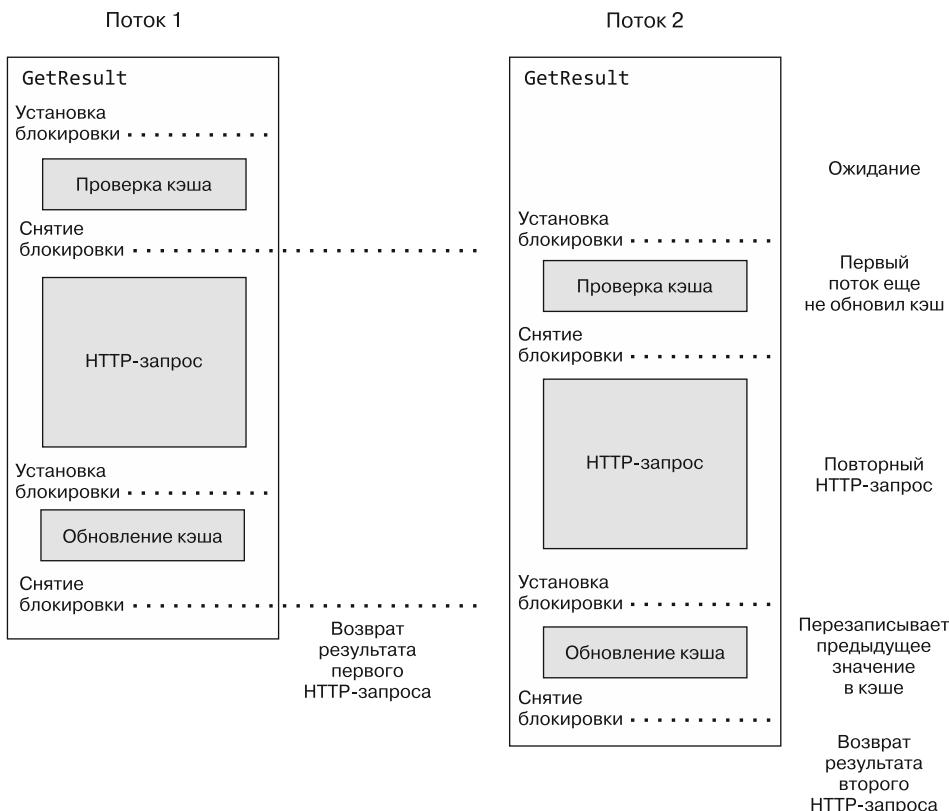


Рис. 5.2. Последовательность выполнения двух вызовов метода, в котором блокировкой защищены только обращения к словарю

5.4. Потоки пользовательского интерфейса

В наборе правил, которыми руководствуется среда выполнения, запуская код после `await`, поток пользовательского интерфейса нативных приложений стоит особняком. Давайте посмотрим почему. Чтобы продемонстрировать, какую проблему это решает, напишем обработчик события нажатия кнопки для WinForms, который выполняет продолжительные вычисления и обновляет пользовательский интерфейс в соответствии с результатом (не волнуйтесь, вам не нужно знать WinForms, чтобы понять код) (листинг 5.11).

В ответ на нажатие кнопки этот код вызывает `LongCalculation`, а затем отображает результат в элементе управления `Label1`. Однако в этом коде есть проблема: поток будет занят на протяжении всего времени выполнения `LongCalculation`, и пользовательский интерфейс приложения подвиснет. Но мы можем исправить эту проблему с помощью многопоточности (листинг 5.12).

Листинг 5.11. Продолжительные вычисления, замораживающие пользовательский интерфейс

```
private void MyButtonClick(object sender, EventArgs ea)
{
    int result = 0;
    result = LongCalculation(); ← Заморозит пользовательский интерфейс
    MyLabel.Text = result.ToString();
}
```

Листинг 5.12. Продолжительные вычисления, которые не замораживают пользовательский интерфейс, но выдают исключение

```
private void MyButtonClick(object sender, EventArgs ea)
{
    Task.Run(()=>
    {
        int result = 0;
        result = LongCalculation();
        MyLabel.Text = result.ToString(); ← Исключение
    });
}
```

Мы просто использовали `Task.Run`, чтобы переместить вычисления в фоновый поток и позволить потоку пользовательского интерфейса свободно обрабатывать события, чтобы пользовательский интерфейс не зависал. Мы решили предыдущую проблему, но создали новую. Теперь попытка отобразить результат из другого потока вызывает исключение и аварийно завершает программу. Нам нужен способ вернуть результат в поток пользовательского интерфейса после завершения фонового потока. К счастью, `Task.Run` возвращает задачу, которую мы и используем. В частности, мы можем проверить ее, чтобы узнать, когда результат будет готов (листинг 5.13).

Листинг 5.13. Корректный запуск вычислений в фоновом режиме из потока пользовательского интерфейса

```
private async void MyButtonClick(object sender, EventArgs ea)
{
    int result = 0;
    await Task.Run(()=>
    {
        result = LongCalculation(); ← Запустится в фоновом потоке;
    });                                пользовательский интерфейс
    MyLabel.Text = result.ToString(); ← не подвиснет
}                                     Этот код выполняется в потоке
                                         пользовательского интерфейса
```

Здесь мы использовали `Task.Run` для запуска продолжительных вычислений в фоновом режиме и `await` для освобождения потока пользовательского интерфейса до завершения вычисления. Поскольку `await` вызывается в потоке пользовательского интерфейса, после завершения вычислений наш код снова

запустится в потоке пользовательского интерфейса и мы сможем изменить текст метки, не вызвав исключений.

Теперь вы понимаете, почему конструкция `await Task.Run` особенно полезна, когда нужно запустить фоновый поток из потока пользовательского интерфейса, в отличие от почти всех других случаев (см. предыдущую главу).

Итоги главы

- Инструменты для использования асинхронных операций также можно применять для запуска многопоточных операций.
- Высокопроизводительный код, выигрывающий от многопоточности, вероятно, также выиграет от использования асинхронных операций.
- В приложениях с графическим пользовательским интерфейсом код после вызова `await` в потоке пользовательского интерфейса тоже будет выполняться в потоке пользовательского интерфейса.
- Во всех остальных случаях код после вызова `await` будет выполняться в пуле потоков (за исключением случаев, когда такое поведение переопределяется с помощью `SynchronizationContext` или `TaskFactory`).
- Если код,зывающий `await`, выполняется в пуле потоков, то код после `await` может быть запущен в другом потоке из того же пула потоков.
- Внутри блока `lock` нельзя вызывать `await`. Лучшим решением будет реорганизовать код и вынести `await` за пределы блока `lock`.

Когда использовать async/await

В этой главе

- ✓ Важность `async/await`.
- ✓ Недостатки `async/await`.
- ✓ Когда следует использовать `async/await`, а когда избегать.

Думаю, вас не удивит, что я, автор книги об асинхронном программировании и многопоточности, считаю эти приемы важными и полезными и что каждый программист должен знать их. Однако следует признать, что они подходят не для каждой ситуации, и если их применять ненадлежащим образом, то они только усложняют ваш программный код и создадут проблемы, которые трудно будет обнаружить.

В начале этой главы мы поговорим о повышении производительности многопоточности и асинхронного программирования, а также о том, как иногда асинхронное программирование может дать обратный эффект и сделать нашу жизнь невыносимой. В оставшейся части этой главы я расскажу о концепциях асинхронного программирования и инструкциях `async/await`, как если бы они были синонимами. На самом деле это не одно и то же, хотя `async/await` — это, безусловно, самый простой способ создания асинхронных программ на C#. Если вы хотите писать асинхронные программы на C#, то должны использовать `async/await`, и наоборот, если вы не собираетесь писать асинхронные программы, то `async/await` будут по большей части бесполезны.

Для начала рассмотрим сценарии, в которых `async/await` действительно проявляют себя во всей красе.

6.1. Преимущества асинхронности на серверах

Однопоточные синхронные серверы — это нонсенс. Такой сервер смог бы обслуживать клиентов только по одному за раз, а максимальная нагрузка будет определяться потреблением вычислительных ресурсов, необходимых для обработки одного подключения, плюс еще нескольких, ожидающих своей очереди в приемном буфере, в зависимости от того, какую длину входной очереди мы настроим.

Однопоточные асинхронные серверы широко распространены, но почти исключительно в языках, не поддерживающих многопоточность, таких как `node.js` и `Python`. Хорошо продуманные асинхронные серверы могут быть довольно эффективными, особенно если они выполняют в основном операции ввода-вывода и почти не производят тяжелых вычислений (например, обслуживают статические файлы или выполняют запросы к базе данных). Но если сервер должен выполнять какие-то нетривиальные действия, то выгодно иметь возможность использовать в реализации сервера дорогой многоядерный процессор.

Чтобы продемонстрировать преимущество производительности при добавлении асинхронных техник в многопоточный сервер, мы создадим два почти идентичных сервера: классический сервер с одним потоком на запрос, который можно найти в учебниках по сетевому программированию, и асинхронный сервер. Это будут простые серверы, предоставляющие по запросу статический файл. Их задача — дождаться подключения клиента и затем прочитать файл с диска и отправить его клиенту.

Но для тестирования наших серверов нам нужен клиент для тестирования нагрузки. Мы также сделаем наш клиент асинхронным, потому что это более эффективно (как мы увидим, запустив тесты далее в этой главе), и дополнительно нам нужно минимизировать влияние клиента на производительность, чтобы точнее измерить производительность серверов. Кроме того, клиент послужит хорошим примером асинхронной программы.

Программа клиента начинает работу с получения количества подключений из командной строки. Это позволит нам запускать тесты с различными нагрузками. Затем вызываем метод `RunTest`, который фактически подключается к серверу. Мы измерим, сколько времени потребуется для завершения всех экземпляров `RunTest`, используя класс `System.Diagnostics.Stopwatch`, а также подсчитаем количество неудачных попыток подключения, так как это даст нам представление о максимальном количестве подключений, которое может обработать сервер (листинг 6.1).

Листинг 6.1. Асинхронный клиент для нагрузочного тестирования

```

using System.Diagnostics;
using System.Net;
using System.Net.Sockets;

int count = int.Parse(args[0]);
Console.WriteLine($"Запущен с {count} соединениями");

var tasks = new Task[count];
int failCount = 0;
var failCountLock = new object();
Stopwatch sw = Stopwatch.StartNew(); ← Запуск секундомера

for (int i = 0; i < count; ++i)
{
    tasks[i] = RunTest(i); ← Запуск отдельного теста
}
Task.WaitAll(tasks); ← Ожидание завершения всех тестов
sw.Stop(); ← Остановка секундомера

lock(failCountLock)
    if (failCount > 0) Console.WriteLine($"Неудачных попыток: {failCount}");
Console.WriteLine($"время: {sw.ElapsedMilliseconds}мс");

Task RunTest(int currentTask)
{
    return Task.Run(()=> ← Запуск параллельных тестов
    {
        var rng = new Random(currentTask);
        await Task.Delay(rng.Next(2*count));
        using var clientSocket =
            new Socket(SocketType.Stream, ProtocolType.Tcp);
        try
        {
            await clientSocket.ConnectAsync( ← Подключение
                new IPEndPoint(IPAddress.Loopback, 7777)); ← к серверу
            var buffer = new byte[1024 * 1024];
            while (clientSocket.Connected)
            {
                int read = await clientSocket.ReceiveAsync(
                    buffer, SocketFlags.None); ← Чтение
                if (read == 0) break; ← данных
            }
        }
        catch
        {
            lock (failCountLock)
                ++failCount; ← Подсчет неудач
        }
    });
}

```

Обратите внимание, что в цикле, когда вызывается метод `RunTest`, который фактически подключается к серверу, нет ожидания завершения операции, потому что нам нужно, чтобы все задачи работали параллельно. Как рассказывалось в главе 3, вызов асинхронного метода не запускает его сразу в фоновом режиме — метод работает синхронно до первого вызова `await`.

Внутри метода `RunTest` мы запускаем код в фоновом режиме с помощью `Task.Run`. Поскольку `RunTest` вызывает только `Task.Run`, мы можем просто вернуть объект `Task`, который получили из вызова `Task.Run`, вместо того чтобы объявить `RunTest` асинхронным и использовать `await`. Это повышает эффективность, так как компилятору не нужно выполнять преобразование `async`, а также создавать и управлять объектом `Task`, который будет всего лишь зеркаливать `Task`, возвращаемый из `Task.Run`.

Внутри теста мы добавляем небольшую случайную задержку перед подключением, потому что в реальных сценариях почти не бывает ситуаций, когда все клиенты пытаются подключиться одновременно. Затем мы подключаемся к серверу и читаем всю информацию, которую отправляет сервер. Мы используем сокеты, потому что это самая низкозатратная технология сетевых взаимодействий, доступная нам.

Взаимодействие через сокет

Мы использовали в клиенте и на сервере взаимодействие через сокет. Поскольку эта книга не о сетевом программировании, я не буду вдаваться в подробности, но для любопытствующих кратко объясню работу сетевых вызовов, которые мы использовали.

На сервере мы сначала должны вызвать метод `Bind`, чтобы взять под контроль сетевой порт, а затем вызвать `Listen`, чтобы сообщить сетевой подсистеме, что ждем подключений от клиентов. Фактическое ожидание подключений осуществляется вызовом `Accept`. Когда клиент подключается к серверу, `Accept` возвращает новый сокет, представляющий подключение. `AcceptAsync` — это асинхронная версия `Accept`. Вместо ожидания она возвращает объект задачи `Task<Socket>`, которая завершится, когда клиент подключится.

Чтобы подключиться к серверу, мы вызываем `ConnectAsync` на стороне клиента. В качестве адреса сервера используется `IPAddress.Loopback` — специальный адрес, который всегда назначается текущему компьютеру. В большинстве сетевых инструментов он более известен как `localhost`.

`Send` отправляет данные и возвращает управление сразу после передачи данных сетевому стеку внутри компьютера-отправителя (не после того, как данные будут фактически отправлены в сеть, и не после того, как данные будут получены другой стороной; мы не можем знать, когда это произойдет). `Send` возвращает количество байтов, фактически принятых сетевым стеком, которое почти всегда будет равно размеру отправляемого буфера. `SendAsync` — асинхронная версия. Она возвращает

управление немедленно вместе с задачей `Task<int>`, которая завершится, когда синхронный метод `Send` вернет управление.

`ReceiveAsync` читает данные в переданный ему массив и возвращает `Task<int>` с количеством полученных байтов. Если результат в этой задаче `Task` равен 0, то это означает, что все данные получены и, вероятно, сервер закрыл соединение.

Наконец, `Shutdown` корректно завершает соединение, уведомляя другую сторону, что соединение закрыто. Он также освобождает все ресурсы, удерживаемые соединением.

Это весь код тестового клиента. Теперь перейдем к серверу. Первый сервер – классическая реализация из учебников, создающая один поток на каждый запрос (листинг 6.2).

Листинг 6.2. Сервер, создающий один поток на запрос

Это классический многопоточный синхронный сервер. Давайте протестируем его.

Я запустил сервер, а затем на стороне клиента создал 50 соединений. Тест успешно завершился менее чем за 8 с. Я посмотрел на количество потоков, которые запустил сервер, и увидел, что он использовал 56 потоков. Из них 50 — это потоки, созданные для обработки запросов. Кроме этого потока есть также главный поток и еще пять фоновых потоков, созданных средой выполнения .NET.

Я повторил тест со 100 соединениями. Тест также успешно завершился, на этот раз примерно за 16 с. Проверив количество потоков на стороне сервера, я обнаружил 106 потоков: 100 потоков для обработки соединений вместо 50 в предыдущем teste и такое же количество вспомогательных потоков.

Воодушевленный успехом, я удвоил количество подключений, доведя его до 200. На этот раз тест провалился: в 61 случае файл не был получен (время, которое потребовалось тесту для завершения, не имеет значения, поскольку он не выполнил всю работу). Сбой вызван тем, что программа слишком медленно принимает новые подключения, потому что занята обработкой более ранних подключений. В результате очередь ожидающих подключений переполнилась и сетевой стек отказался принимать любые последующие подключения.

Теперь, увидев ограничения первого сервера, напишем асинхронную версию. А чтобы все было максимально честно, мы просто заменим синхронные блокирующие вызовы их асинхронными версиями (листинг 6.3).

Листинг 6.3. Асинхронный сервер

```
using System.Net;
using System.Net.Sockets;

var listenSocket = new Socket(SocketType.Stream, ProtocolType.Tcp);
listenSocket.Bind(new IPEndPoint(IPAddress.Any, 7777));
listenSocket.Listen(100);

while(true)
{
    var connection = await listenSocket.AcceptAsync(); ←
    Task.Run(async () => ←
    {
        using var file = new FileStream("somefile.bin",
            FileMode.Open, FileAccess.Read);
        var buffer = new byte[1024*1024];
        while (true)
        {
            int read = await file.ReadAsync(buffer,
                0, buffer.Length)); ←
            if (read != 0) ←
                Вызов File.ReadAsync
                вместо Read
    });
}
```

```

    {
        await connection.SendAsync(
            new ArraySegment<byte>(buffer, 0, read),
            SocketFlags.None); ← Вызов SendAsync
        // вместо Send
    }
    else
    {
        connection.Shutdown(SocketShutdown.Both);
        connection.Dispose();
        return;
    }
}
});
```

Этот код идентичен коду в листинге 6.2, за исключением замены `Accept` на `AcceptAsync`, `Send` на `SendAsync`, `new Thread` на `Task.Run` и, конечно же, добавления ключевых слов `async` и `await` там, где это необходимо.

В этой версии сервера, в отличие от предыдущей, вызовы `AcceptAsync`, `SendAsync` и `ReadAsync` будут освобождать поток, а не блокировать его. Это означает, что серверу понадобится меньшее число потоков для обработки того же количества параллельных подключений. Вместо создания нового потока для каждого подключения `Task.Run` будет использовать пул потоков.

Теперь мы можем наконец-то сравнить производительность синхронной и асинхронной версий (табл. 6.1). Как и в случае с первым сервером, я сначала запустил тест с 50 соединениями. Тест успешно завершился за 8 с — то же количество времени, которое потребовалось синхронной версии. Однако эта версия использовала всего 27 потоков вместо 56 (20 потоков в пуле, выполняющих всю работу, главный поток и шесть потоков, созданных фреймворком или операционной системой, — на один поток больше, чем в синхронной версии).

Таблица 6.1. Различия между синхронной и асинхронной обработкой соединений

Количество соединений	Синхронная версия			Асинхронная версия		
	Неудачных попыток	Время, мс	Количество потоков	Неудачных попыток	Время, мс	Количество потоков
50	0	8398	56	0	8534	27
100	0	16 538	106	0	16 310	27
200	61	—	—	0	32 132	27
300	168	—	—	0	48 229	27
400	265	—	—	72	—	—

После первого успешного теста я удвоил количество подключений до 100, и тест успешно завершился примерно за 16 с — снова то же количество времени, что и в синхронной версии. Количество потоков снова оказалось равным 27, как и в teste с 50 соединениями (сравните со 106 потоками в синхронной версии).

Я снова удвоил количество соединений до 200 (если вы помните, в этом teste синхронная версия начала давать сбои), но асинхронная версия снова успешно справилась с задачей и использовала только 27 потоков.

Избавлю вас от всех скучных подробностей остальных тестов. На простеньком ноутбуке, на котором я это пишу, синхронная версия смогла обработать около 130 соединений, тогда как асинхронная версия обработала чуть больше 300.

Эти цифры могут показаться невпечатляющими, но имейте в виду, что редко какому серверу приходится обрабатывать такое количество подключений за столь короткое время (в среднем подключение каждые 2 мс), еще одновременно выполняя Word и Visual Studio. Кроме того, важно отметить, что эти цифры очень приблизительные; точное количество подключений будет зависеть от оборудования, особенностей приложения, от того, что еще запущено в данный момент, характера использования, версии .NET, версии операционной системы и т. д. Но одно видно ясно — асинхронная версия постоянно использует меньше ресурсов и может обрабатывать большие нагрузки. По сути, здесь мы видим, что асинхронный сервер способен обрабатывать вдвое большую нагрузку, чем синхронная версия, используя при этом существенно меньше ресурсов.

Я предполагаю, что в настоящее время большая часть разработки на C# будет выполняться на сервере, но существуют и нативные приложения.

6.2. Преимущества асинхронности в нативных клиентских приложениях

Асинхронное выполнение кода также может быть очень полезным в нативных приложениях. Конечно, они обычно не выполняют десятки тысяч действий одновременно (из-за ограниченных физических возможностей человека по сравнению с компьютером), им достаточно поддерживать поток управления пользовательским интерфейсом, чтобы тот оставался отзывчивым и доступным в любой момент.

Например, если приложение выполняет продолжительные вычисления, из-за которых пользовательский интерфейс подвисает, то, скорее всего, код, из-за которого это происходит, будет выглядеть примерно так:

```
public void button1_Click(object sender, EventArgs args)
{
    LongCalculation();
    UpdateUIWithCalculationResults();
}
```

Чтобы исправить проблему, нужно переместить `LongCalculation` в фоновый поток, но при этом мы должны сохранить `UpdateUIWithCalculationResults` в потоке пользовательского интерфейса. Это можно сделать примерно так:

```
public void button1_Click(object sender, EventArgs args)
{
    Task.Run(()=>
    {
        LongCalculation();
        BeginInvoke(()=>
        {
            UpdateUIWithCalculationResults();
        });
    });
}
```

Здесь `Task.Run` запускает вычисления в фоновом потоке, а `BeginInvoke` определяет код обратного вызова, который будет выполняться в потоке пользовательского интерфейса. Если применить `async/await`, то можно положиться на `await`, чтобы вернуть управление в нужный поток, и тогда достаточно будет написать:

```
public async void button1_Click(object sender, EventArgs args)
{
    await Task.Run(()=>
    {
        LongCalculation();
    });
    UpdateUIWithCalculationResults();
}
```

Эти три преимущества `async/await` (способность обрабатывать более высокие нагрузки, меньшее количество используемых ресурсов даже при низких нагрузках и простота использования многопоточности в сочетании с кодом, который должен выполняться в определенном потоке) довольно значительны и могут легко перевесить все недостатки. Но недостатки есть, и вы должны знать о них.

6.3. Недостатки `async/await`

До сих пор мы много говорили о преимуществах асинхронного программирования и как `async/await` делают его простым. Но, как и многое другое, асинхронное программирование имеет существенные недостатки.

6.3.1. Асинхронное программирование зарано

Любой код, вызывающий асинхронный код, сам должен быть асинхронным. Для получения результатов после вызова асинхронного кода вы должны использовать `await` или обратные вызовы. Такой код часто называют «полностью асинхронным».

Для иллюстрации рассмотрим программу, которая делает снимок с помощью камеры, подключенной к компьютеру. Следующий фрагмент кода демонстрирует, что нужно сделать, чтобы использовать камеру с помощью Windows UWP API. Для простоты я удалил все параметры и код, который ищет камеру, но структура кода правильная:

```
public void TakeSinglePicture()
{
    cameraApi.AquireCamera();
    cameraApi.TakePicture();
    cameraApi.ReleaseCamera();
}
```

Код сначала получает доступ к камере, затем выполняет съемку и, наконец, освобождает камеру и все связанные с ней ресурсы. Теперь предположим, что в новейшей версии API нашей воображаемой камеры метод `TakePicture` сделан асинхронным. Если просто переключиться на новую асинхронную версию, то мы получим:

```
public void TakeSinglePicture()
{
    cameraApi.AquireCamera();
    cameraApi.TakePictureAsync(); ← Ошибка: попытка освободить
    cameraApi.ReleaseCamera();   камеру до завершения
                                TakePictureAsync
```

Это логическая ошибка: съемка происходит асинхронно, поэтому она завершится в фоновом режиме, но в этом примере мы не ждем ее завершения и освобождаем камеру, когда она еще делает снимок. В этом случае мы должны дождаться завершения `TakePictureAsync` и только потом продолжить. К счастью, `async/await` упрощают задачу, но нужно изменить метод `TakeSinglePicture`, чтобы он тоже стал асинхронным:

```
public async Task TakeSinglePicture() ← Вместо void метод возвращает Task
{
    cameraApi.AquireCamera();
    await cameraApi.TakePictureAsync();
    cameraApi.ReleaseCamera();
}
```

Казалось бы, ошибку легко исправить, но теперь то же самое нужно сделать с кодом, который вызывает `TakeSinglePicture`, и с кодом, который вызывает его, и с кодом, который вызывает этот код, и так до точки входа из нашего кода.

Можно подумать, что проблему легко решить, использовав `Task.Wait()` и `Task.Result` и превратив асинхронный код в блокирующий, но если асинхронный код не был специально разработан для поддержки этого варианта

использования, то такое решение может привести к странным ошибкам и взаимоблокировкам. Некоторые API (например, UWP API камеры, на котором основан этот пример) сразу же дадут сбой и генерируют исключение. И даже если этот подход сработает, то, превращая асинхронный вызов в блокирующий, мы устранием все преимущества асинхронности.

6.3.2. Асинхронное программирование имеет больше пограничных случаев

Другая проблема заключается в том, что, когда мы делаем код асинхронным, добавляются новые пограничные случаи и варианты сбоев, которые просто не возникают в синхронном однопоточном коде. Возьмем простой код WinForms. Пусть есть программа, которая управляет источниками, предоставляющими данные, и этот конкретный код выбирает случайный источник и отображает имя источника и полученное из него значение (для большей реалистичности этот код также использует имена, автоматически генерированные редактором WinForms):

```
private void button1_Click(object sender, EventArgs args)
{
    var source = GetRandomSource();
    label1.Text = source.Name;
    label2.Text = source.GetValue();
}
```

Этот код довольно прост, и в нем нет ничего, что могло бы привести к сбою, кроме самих источников. Но если `GetValue` выполняется долго, то это приведет к подвисанию пользовательского интерфейса. Эту проблему можно решить, сделав методы `GetValue` и `button1_Click` асинхронными. Для этого в наш метод достаточно внести минимальные изменения, после чего пользовательский интерфейс станет отзывчивым:

```
private async void button1_Click(object sender, EventArgs args)
{
    var source = GetRandomSource();
    label1.Text = source.Name;
    label2.Text = await source.GetValue();
}
```

Это решение может показаться удивительно простым, но мы внесли ошибку. Теперь, когда пользовательский интерфейс продолжит работу во время выполнения `GetValue`, пользователь сможет нажать кнопку еще раз, и если нам не повезет, то может возникнуть ситуация, когда интерфейс будет отображать имя одного источника, полученное после первого щелчка, а значение — из другого, полученного после второго щелчка, показывая пользователю неверную

информацию. Чтобы исправить проблему, нужно заблокировать кнопку на время работы кода:

```
private async void button1_Click(object sender, EventArgs args)
{
    button1.Enabled=false;
    var source = GetRandomSource();
    label1.Text = source.Name;
    label2.Text = await source.GetValue();
    button1.Enabled=true;
}
```

Иногда может даже потребоваться заблокировать все элементы управления пользовательского интерфейса, в зависимости от особенностей приложения и связей между различными элементами.

6.3.3. Многопоточность имеет еще больше пограничных случаев

Причина, почему в предыдущей демонстрации использовался фреймворк WinForms, заключается в том, что WinForms позволяет легко писать асинхронный и однопоточный код. Но в настоящее время мы все реже пишем настольные приложения, и эта невинно выглядящая инструкция `await`, добавленная в код, может сделать его многопоточным, а вы даже не будете подозревать об этом.

Многопоточность таит в себе так много ловушек, что ей посвящена целая глава далее в книге.

6.3.4. `async/await` — это дорого

`async/await` — дорогое удовольствие по сравнению с однопоточным кодом. Они заставляют компилятор генерировать много вспомогательного кода, необходимого, чтобы сделать код асинхронным. Важно помнить, что сейчас мы говорим о сравнении асинхронного кода с однопоточным, и в большинстве случаев асинхронный код эффективнее синхронного многопоточного кода, даже с учетом всех накладных расходов.

Например, возьмем эту законченную, но бесполезную программу на C#:

```
Thread.Sleep(1000);
```

Какой фактический код будет сгенерирован для этой программы? Чтобы ответить на этот вопрос, воспользуемся IISpy — бесплатной программой, которая на основе скомпилированной программы для .NET может восстановить ее исходный код. Просматривая скомпилированный код, IISpy видит весь код, сгенерированный компилятором.

После декомпиляции программы мы получим восемь строк кода. Одна строка – это наш исходный код, и семь строк, заключающих наш код в метод `Main` и класс (потому что компилятор C# позволяет просто писать код, но среда выполнения поддерживает только код внутри классов и методов). Для эквивалентной асинхронной программы:

```
await Task.Delay(1000);
```

мы получим уже 63 строки кода! Компилятор сделал то, о чем мы говорили в главе 3: он превратил эту строку в класс, реализующий конечный автомат с двумя состояниями (до и после `await`), со вспомогательным кодом для управления им.

Итак, обсудив преимущества и недостатки, давайте попробуем ответить на вопрос: когда следует использовать `async/await`, а когда желательно их избегать?

6.4. Когда использовать `async/await`

Вот простые рекомендации, которые помогут решить, когда использовать `async/await`, а когда лучше выбрать синхронные блокирующие операции.

- Если код должен управлять большим количеством подключений одновременно, то используйте `async/await` всегда, когда это возможно.
- При работе с асинхронным API используйте `async/await` всегда, когда используете этот API.
- Если в вашем приложении с графическим пользовательским интерфейсом возникла проблема с зависанием интерфейса, используйте `async/await` для запуска кода, который выполняется продолжительное время и является причиной зависания пользовательского интерфейса.
- Если ваш код создает новый поток для обработки каждого запроса/элемента и значительная часть времени выполнения приходится на ввод-вывод (например, доступ к сети или файлам), то подумайте о возможности использовать `async/await`.
- Если вы добавляете код в приложение, широко использующее `async/await`, то указывайте `async/await` там, где это имеет смысл.
- Если вы добавляете код в приложение, не использующее `async/await`, то избегайте этих инструкций по мере возможности. Если вы решили использовать `async/await` в новом коде, то подумайте о рефакторинге хотя бы того кода, который вызывает новый код, чтобы и в нем использовать `async/await`.
- Если вы пишете код, который одновременно выполняет только одну операцию, не используйте `async/await`.
- Во всех остальных случаях внимательно исследуйте возможные компромиссы и принимайте обоснованное решение.

Эти рекомендации отсортированы в порядке убывания важности. Если ваш проект соответствует условиям более чем одной из перечисленных рекомендаций, то отдавайте предпочтение той, что стоит выше в этом списке. В любом случае если ваше приложение точнее всего соответствует последнему пункту, то постарайтесь тщательно взвесить все компромиссы и принять наиболее оптимальное решение. Я бы очень хотел дать вам простые правила, охватывающие все возможные варианты, но правда в том, что проектирование программного обеспечения — слишком сложное занятие и просто невозможно дать конкретную рекомендацию, не зная всех особенностей вашего проекта. В конце концов, если бы разработка программного обеспечения была такой простой, то вам не пришлось бы читать книги об этом.

Итоги главы

- Асинхронный код может справляться с гораздо большей нагрузкой, чем синхронный код, используя при этом значительно меньше ресурсов.
- В случаях, когда важно запустить код в определенном потоке (например, в приложениях с пользовательским интерфейсом), `async/await` упрощает использование многопоточности и асинхронных вызовов.
- Однако асинхронный код имеет некоторые недостатки:
 - код,зывающий асинхронные методы, сам должен быть асинхронным;
 - асинхронный код более подвержен ошибкам, чем синхронный однопоточный код;
 - многопоточный код более подвержен ошибкам, чем однопоточный;
 - для асинхронных методов генерируется больше кода, чем для синхронных (но они все равно быстрее и эффективнее, чем синхронный многопоточный код).
- Принимая решение об использовании или неиспользовании `async/await`, учитывайте компромиссы.

Классические ловушки многопоточности и как их избежать

В этой главе

- ✓ Классические ловушки многопоточности: частичное обновление, взаимоблокировка, состояние гонки, синхронизация и голодание.
- ✓ Переупорядочение доступа к памяти и модель памяти C#.
- ✓ Простые правила, помогающие избежать классических ловушек многопоточности.

При переходе от однопоточного к многопоточному программированию важно понимать, что многопоточность влечет за собой определенные типы ошибок, которые не возникают в однопоточных приложениях. К счастью, понимая общие категории ошибок, мы можем их избежать. В этой главе представлены простые рекомендации, следуя которым можно значительно снизить вероятность столкновения с такими проблемами.

Начнем обзор с фундаментального побочного эффекта многопоточности. В однопоточной среде каждый фрагмент кода должен выполнить свою задачу до того, как начнет выполняться следующий. Однако, когда два фрагмента кода выполняются одновременно, один может попытаться обратиться к данным, которые все еще обрабатываются другим фрагментом, что может привести к неправильному результату.

7.1. Частичное обновление

Частичное обновление происходит, когда один поток обновляет некоторые данные, а в середине этого обновления другой поток читает данные и видит смесь старых и новых значений.

Иногда эта проблема очевидна, например, в коде:

```
x = 5;
y = 7;
```

первая строка присваивает значение переменной `x`, а вторая — переменной `y`. Между этими операциями есть небольшой промежуток времени, когда `x` уже получила значение 5, а `y` еще не равна 7. Однако часто проблема не так очевидна. Например, следующий метод имеет только одно присваивание и подвержен потенциальной проблеме частичного обновления:

```
void SetGuid(Guid src)
{
    _myGuid = src;
}
```

Здесь `Guid` — это структура, и, хотя C# позволяет скопировать структуру с помощью одного оператора присваивания, внутренне компилятор генерирует код, копирующий члены структуры по одному, создавая эквивалент первого фрагмента кода.

Но ситуация может быть еще хуже. Следующий код присваивает значение переменной типа `decimal`. Тип `decimal` — это базовый тип в .NET, а не структура. Так что же здесь может пойти не так?

```
void SetPrice(decimal newPrice)
{
    _price = newPrice;
}
```

Проблема в том, что `decimal` имеет размер 128 бит, а в 64-битных процессорах доступ к памяти осуществляется блоками по 64 бита. Поэтому присваивание значения переменной типа `decimal` разбивается на две отдельные операции с памятью, что и порождает потенциальную проблему частичного обновления, как и в двух других примерах.

Однако `decimal` — это необычный базовый тип. Это базовый тип в .NET, но изначально он не поддерживается ни в одной известной мне аппаратной архитектуре, поэтому поговорим о базовом типе: `long`. Тип `long` — это 64-битное целое число, и этот тип является нативно поддерживаемым типом в 64-битных процессорах. Я уже упоминал, что доступ к памяти осуществляется 64-битными

блоками, поэтому присваивание значения переменной типа `long` должно быть безопасным, верно?

```
void SetIndex(long newIndex)
{
    _index = newIndex;
}
```

В 64-битных системах это присваивание, скорее всего, будет выполнено атомарно, но .NET по-прежнему поддерживает 32-битные операции, и если этот код будет выполняться на 32-битном компьютере (или в 32-битной операционной системе на 64-битном процессоре, или на 32-битном процессе в 64-битной ОС – вы поняли суть), то доступ к памяти будет выполняться 32-битными блоками, и возникнет та же проблема.

Решение всех этих проблем заключается в использовании механизма блокировки, а самый простой механизм блокировки в C# – это инструкция `lock`. Например, в листинге 7.1 мы используем инструкцию `lock` при каждом обращении к переменной-члену (как при чтении, так и при записи), поэтому мы защищены от проблемы частичного обновления.

Листинг 7.1. Использование оператора lock

```
private int _x;
private int _y;
private object _lock = new object();

public void SetXY(int newX, int newY)
{
    lock(_lock) ←
    {
        _x = newX;
        _y = newY;
    }
}

public (int x, int y) GetXY()
{
    lock(_lock) ←
    {
        return (_x,_y);
    }
}
```

Инструкция `lock` защищает операцию записи

Другая инструкция `lock` защищает операцию чтения

Инструкции `lock` не позволяют нескольким потокам одновременно выполнять код, который находится внутри блока `lock`. Если поток достигнет инструкции `lock`, когда другой поток уже выполняет код в блоке `lock`, то он приостановится и будет ждать, когда другой поток выйдет из блока.

Инструкция `lock` принимает параметр, который позволяет нам создавать разные блокировки для разных переменных. При достижении инструкции `lock` поток будет ждать, только если блок оператора `lock` с тем же параметром выполняется другим потоком. В листинге 7.2 выполняются операции с двумя разными значениями, `_a` и `_b`. Если одновременно вызвать `GetA` и `GetB` из разных потоков, то один из них запустится немедленно, а другой будет ждать, пока первый не выйдет из блока `lock`.

Листинг 7.2. Единая блокировка для двух переменных

```
private object _lock = new object();
private int _a;
private int _b;

public int GetA()
{
    lock(_lock)
    {
        return _a;
    }
}

public int GetB()
{
    lock(_lock)
    {
        return _b;
    }
}
```

Однако в следующем примере, где `GetA` использует `_lockA`, а `GetB` — `_lockB`, эти методы могут выполняться одновременно и будут приостановлены только в том случае, если будут вызваны одновременно с другим фрагментом кода, который использует оператор `lock` с `_lockA` или `_lockB` соответственно (листинг 7.3).

Листинг 7.3. Две блокировки для двух переменных

```
private object _lockA = new object();
private object _lockB = new object();
private int _a;
private int _b;

public int GetA()
{
    lock(_lockA)
    {
        return _a;
    }
}
```

```
public int GetB()
{
    lock(_lockB)
    {
        return _b;
    }
}
```

В инструкциях `lock` лучше всего использовать приватный член типа `object` (в .NET 9 и более поздних версиях также можно использовать объект типа `Lock`), добавленный специально для инструкции `lock` и недоступный нигде за пределами класса. Не раскрывать его за пределами вашего класса стоит для того, чтобы устраниить риск использования этого же объекта в инструкции `lock` внешним кодом, поскольку это может нарушить стратегию блокировки и привести к взаимоблокировке (как будет показано далее в этой главе). Причина использования типа `object` заключается в том, что любой другой класс, который действительно что-то делает, может использовать `lock(this)` (такое часто встречается в старом коде, написанном еще до того, как использование приватного объекта `object` стало рекомендуемой практикой), что тоже может нарушить стратегию блокировки и вызвать взаимоблокировку.

Вы могли бы подумать, что частичные обновления можно предотвратить, внимательно отслеживая порядок операций присваивания, но этому решению препятствует изменение порядка доступа к памяти.

7.2. Изменение порядка доступа к памяти при компиляции

В современных аппаратных архитектурах доступ к памяти является медленной операцией, по сравнению с обработкой данных внутри процессора, и различные шаблоны доступа к памяти могут оказывать значительное влияние на производительность. Для более эффективного использования оборудования компилятор может изменить порядок операций в вашем коде. Это может быть сделано ради уменьшения количества операций доступа к памяти и позволяет ускорить доступ к памяти.

Компьютер, который я использую для написания этой книги, имеет память DDR4 2,666 МГц. У нее задержка около 14,5 нс (то есть 0,0000000145 с), но процессор имеет 12 виртуальных ядер, работающих на частоте 2,66 ГГц, то есть на выполнение каждого такта уходит всего 0,37 нс (чтобы наглядно представить этот интервал времени, замечу, что, пока свет идет от экрана до вашего глаза, каждое ядро такого процессора успевает выполнить около семи операций). Простое деление говорит нам, что каждое ядро ЦП может выполнить примерно 40 операций за время, необходимое для извлечения одного значения из памяти, или, учитывая количество ядер, — до 480 операций (реальный мир,

конечно, сложнее, и объем работы, выполняемой процессором за один такт, может варьироваться в зависимости от того, что именно делает код; это максимальное значение). Выражаясь человеческим языком, если бы процессор выполнял одну операцию в секунду, то загрузка одного значения из памяти занимала бы 8 мин.

Рассмотрим простой пример, как компилятор может улучшить производительность, перемещая и устранивая операции доступа к памяти. Возьмем простой цикл, который увеличивает переменную в 100 раз:

```
int counter=0;
for(int i;i<100;++i)
{
    ++counter;
}
```

Теперь переведем этот код на C# в псевдомашинный код. В машинном коде каждый оператор или выражение делится на инструкции. Инструкции, которые выполняют вычисления, могут работать только с переменными внутри самого процессора. Эти переменные называются *регистрами*, и их количество ограничено. Загрузка значения из памяти в регистр или сохранение значения из регистра в память — это отдельные инструкции. Чтобы описание было более кратким и читабельным, мы не будем переводить в машинные инструкции сам цикл:

```
Записать 0 в регистр (быстро)
Сохранить значение из регистра в память в переменную "counter" (медленно)
for(int i;i<100;++i)
{
    Загрузить значение из памяти с переменной "counter" в регистр (медленно)
    Увеличить значение в регистре (быстро)
    Сохранить значение из регистра в память в переменную "counter" (медленно)
}
```

В процессе выполнения этот псевдокод выполнит 101 быструю и 201 медленную операции (не учитывая накладные расходы на сам цикл `for`). Теперь вынесем доступ к памяти за пределы цикла:

```
Записать 0 в регистр (быстро)
Сохранить значение из регистра в память в переменную "counter" (медленно)
Загрузить значение из памяти с переменной "counter" в регистр (медленно)
for(int i;i<100;++i)
{
    Увеличить значение в регистре (быстро)
}
Сохранить значение из регистра в память в переменную "counter" (медленно)
```

Новый псевдокод получит тот же результат, но при этом выполнит всего четыре медленные операции (сравните с 201 операцией в первом варианте

перевода в машинный код). Но этот код можно еще ускорить. В самом начале код сохраняет значение из регистра в переменной, а затем тут же загружает ее. Эти две операции можно опустить и получить

```
Записать 0 в регистр (быстро)
for(int i;i<100;++i)
{
    увеличить значение в регистре (быстро)
}
Сохранить значение из регистра в память в переменную "counter" (медленно)
```

Теперь код выполняет 101 быструю операцию и только 1 медленную, что существенно меньше, чем 101 быстрая и 201 медленная операция в первом варианте псевдомашинного кода. Если представить, что каждая быстрая операция занимает 1 единицу времени, а каждая медленная — 10 единиц, то первый вариант кода выполнится за 2111 единиц времени, а последний — всего за 121 единицу, то есть в 17 раз быстрее!

Общее правило заключается в том, что компилятору разрешено вносить любые изменения, которые не изменяют конечный результат в однопоточной среде; в нашем примере единственным результатом является значение переменной `counter` в конце цикла. В однопоточной среде все наши преобразования не меняют конечный результат, поскольку нет других потоков, которые могли бы наблюдать значение `counter` в середине выполнения нашего кода. В многопоточной среде ситуация иная. В исходном коде другой поток мог бы видеть, как `counter` постепенно увеличивается, тогда как в оптимизированной версии `counter` сразу получает конечное значение.

Теперь применим ту же логику к другому фрагменту кода. Попытаемся предотвратить выполнение двумя потоками одного и того же блока кода, используя флаг, который устанавливается перед запуском и сбрасывается по завершении. Перед входом в код проверим значение этого флага и прервем выполнение, если флаг установлен:

```
if(_doingSomething) return;
_doingSomething = true;
// выполнить какие-то действия
_doingSomething = false;
```

По завершении этого фрагмента флаг `_doingSomething` всегда содержит значение `false`, то есть в однопоточной среде никто и никогда не сможет увидеть значение `true`, поэтому такой код эквивалентен следующему:

```
if(_doingSomething) return;
// выполнить какие-то действия
_doingSomething = false;
```

Компилятор может свободно перемещать или удалять код, устанавливающий флаг, и тем самым полностью ликвидировать наш самодельный механизм синхронизации потоков. Как видите, оптимизация кода путем внесения изменений, которые не изменяют результата в однопоточной среде, может привести к результатам, которые явно бессмысленны в многопоточной среде.

В действительности ситуация еще хуже, потому что доступ к памяти компьютера осуществляется очень медленно. У процессора есть своя внутренняя память, меньшая по объему и более быстрая (но все равно более медленная, чем регистры). Она называется *кэш-памятью*. Процессор пытается загрузить данные из основной памяти в кэш до того, как они понадобятся (в фоновом режиме, во время выполнения других задач), поэтому, когда выполняется инструкция чтения значения из памяти, это значение уже находится в кэше. Разные ядра могут иметь свою кэш-память.

Все вместе это означает, что такой код:

```
public void Init()
{
    _value = 7;
    _initialized = true;
}
```

не гарантирует, что если `_initialized` имеет значение `true`, то переменной `_value` к этому моменту будет присвоено новое значение. Компилятору разрешено менять порядок этих операций присваивания, и даже если он этого не сделает, то ваш код может увидеть устаревшую неинициализированную версию `_value` просто потому, что она все еще находится в кэше.

Если вы прочтете только первый абзац описания ключевого слова `volatile` в C#, то у вас может сложиться ошибочное впечатление, что оно способно решить проблему. Однако семантика `volatile` в C# настолько сложна, что не гарантирует получение последнего значения и может вызвать еще больше проблем. Поэтому старайтесь не использовать `volatile` — оно делает совсем не то, что вы думаете.

Очевидно, что невозможно написать правильный многопоточный код, когда любое обращение к памяти может быть перемещено или удалено. Вот почему существуют инструменты, ограничивающие систему в вариантах перемещения операций доступа к памяти. Есть операции, которые говорят системе: «Не выполняя операцию чтения раньше этой точки». Это называется *семантикой захвата*, и все операции, которые приобретают блокировку, имеют это свойство. Есть операции, которые говорят системе: «Не выполняя операцию записи позднее этой точки». Это называется *семантикой освобождения*, и все операции, которые освобождают блокировку, имеют это свойство. На рис. 7.1 показано, как семантики захвата и освобождения влияют на способность системы перемещать операции доступа к памяти.

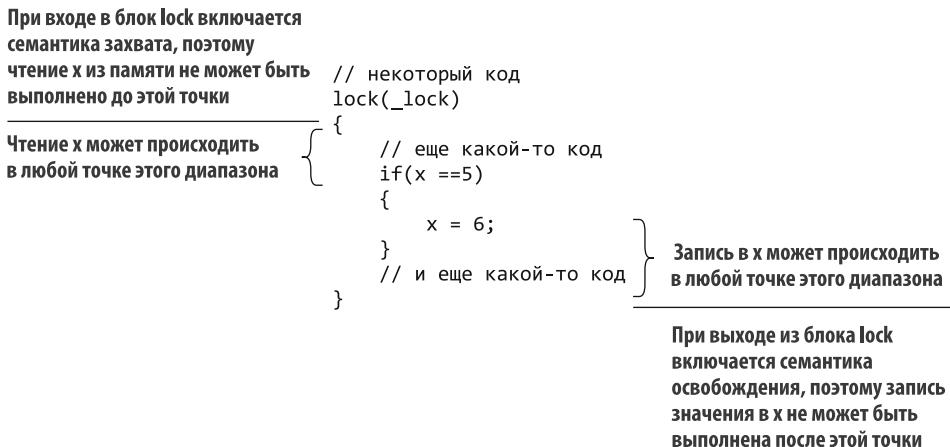


Рис. 7.1. Семантики захвата и освобождения

Существуют также операции, не позволяющие компилятору перемещать обе операции, чтения и записи, в любом направлении. Они называются *барьерами памяти*. Набор правил, как именно компилятору и процессору разрешено перемещать операции доступа к памяти, в дополнение к правилам включения семантик захвата, освобождения и барьеров памяти, называется *моделью памяти*.

В отношении переупорядочения операций с памятью и модели памяти в C# важно помнить, что при использовании блокировок для доступа к любым данным, которые могут использоваться одновременно несколькими потоками, все просто работает. Получение блокировки включает семантику захвата и позволит вам получить самые последние значения из памяти. Освобождение блокировки включает семантику освобождения, и в результате все изменения будут записаны в память и доступны другому потоку, который войдет в блок `lock`. Так мы подошли к первому правилу: всегда используйте блокировку при доступе к общим данным.

Теперь, убедившись еще раз, что всегда необходимо использовать блокировки, поговорим о самой распространенной проблеме с блокировками — взаимоблокировках.

7.3. Взаимоблокировки

Взаимоблокировка, как уже упоминалось в главе 4, — это ситуация, когда поток блокируется в ожидании ресурса, который никогда не станет доступным из-за чего-то, что уже успел сделать этот же поток. В классической взаимоблокировке один поток удерживает ресурс A, ожидая ресурс B, в то же время другой поток

удерживает ресурс В, ожидая ресурс А. Оба потока блокируются в ожидании, когда другой поток освободит блокировку. Но этого никогда не произойдет, потому что другой поток тоже блокируется, как показано на рис. 7.2.

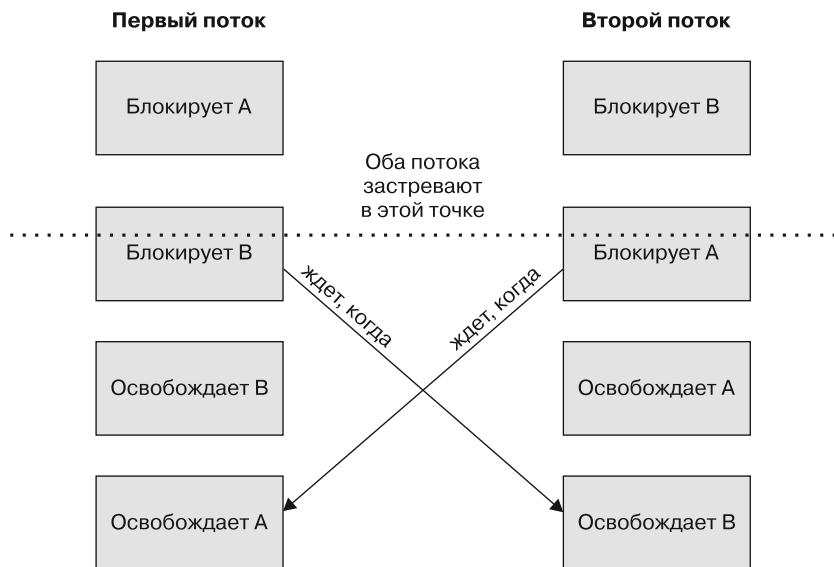


Рис. 7.2. Простая взаимоблокировка двух потоков

Это классический и наиболее распространенный пример взаимоблокировки, но взаимоблокировки могут быть и часто бывают более сложными. В кольцо взаимоблокировки может быть вовлечено любое количество потоков (поток 1, удерживающий А и ожидающий В, поток 2, удерживающий В и ожидающий С, и поток 3, удерживающий С и ожидающий А). И даже один и тот же поток может заблокировать самого себя, что возможно, когда он удерживает ресурс, пытаясь снова получить тот же ресурс.

Некоторые примитивы синхронизации, такие как инструкция `lock` или класс `Mutex`, позволяют одному и тому же потоку получить их более одного раза (они должны быть освобождены столько же раз, сколько были приобретены). Такие блокировки называются *рекурсивными*. Другие примитивы, такие как класс `Semaphore` или файлы в режиме исключительного доступа, будут рассматривать каждую попытку получить их — даже тем же потоком — как отдельную попытку и будут блокировать поток (вызывая взаимоблокировку) или завершаться неудачей, в зависимости от фактического ресурса.

Иногда проблему можно заметить, просто прочитав код. Например, в листинге 7.4 есть два метода, один из которых приобретает блокировки в обратном порядке относительно другого.

Листинг 7.4. Код с простой ошибкой взаимоблокировки

```

public int Multiply()
{
    lock(_leftOperandLock)           ← Сначала блокируется левый
    {                                операнд, затем правый
        lock(_rightOperandLock)
        {
            return _leftOperand * _rightOperand;
        }
    }
}
public int Add()
{
    lock(_rightOperandLock)          ← Сначала блокируется правый
    {                                операнд, затем левый
        lock(_leftOperandLock)
        {
            return _leftOperand + _rightOperand;
        }
    }
}

```

В этом примере программист, написавший метод `Multiply`, сначала блокирует доступ к левому операнду, потому что читает выражение слева направо, а программист, написавший метод `Add`, сначала блокирует правый операнд, просто потому что он правша. С вычислительной точки зрения порядок блокировки не имеет значения, но если запустить эти два метода одновременно, то возникнет состояние взаимоблокировки.

Так мы подошли ко второму правилу: всегда устанавливайте блокировки в одном и том же порядке. Сам порядок не имеет значения. Вы можете проанализировать код и выбрать наиболее оптимальный порядок или всегда применять блокировки в алфавитном порядке — это не имеет значения. Суть в том, чтобы блокировки всегда устанавливались в одном и том же порядке. Это называется *иерархией блокировок*.

В листинге 7.5 приводится код с исправленной ошибкой, которая имелась в листинге 7.4. Здесь блокировки устанавливаются в порядке чтения математических выражений, поэтому `_leftOperandLock` всегда устанавливается раньше `_rightOperandLock`.

Важно всегда соблюдать один и тот же порядок установки блокировок, даже если кажется, что есть веская причина изменить его. Например, добавим метод `Divide` (листинг 7.6), и поскольку деление на ноль не допускается в математике, этот метод сначала проверяет неравенство правого операнда нулю и только потом делит числа (если второе число равно нулю, то он вызовет событие `DivideByZero`). У нас может возникнуть соблазн сначала заблокировать и проверить правый операнд, прежде чем блокировать левый операнд, потому что если правый операнд равен нулю, то обращаться к левому операнду вообще не потребуется.

Листинг 7.5. Устранение взаимоблокировки с помощью иерархии блокировок

```
public int Multiply()
{
    lock(_leftOperandLock) ← Сначала блокируется левый
    {
        lock(_rightOperandLock)
        {
            return _leftOperand * _rightOperand;
        }
    }
}
public int Add()
{
    lock(_leftOperandLock) ← Тоже сначала блокируется
    {
        lock(_rightOperandLock)
        {
            return _leftOperand + _rightOperand;
        }
    }
}
```

Листинг 7.6. Создание ситуации взаимоблокировки попыткой избежать ненужной блокировки

```
public int Divide()
{
    lock(_rightOperandLock) ← Сначала блокируется правый
    {
        if(_rightOperand==0)
        {
            DivideByZeroEvent?.Invoke(this,EventArgs.Empty);
            return 0;
        }
        lock(_leftOperandLock) ← Левый операнд блокируется,
        {
            return _leftOperand/_rightOperand;
        }
    }
}
```

В этом методе, в попытке избежать ненужной блокировки, мы нарушили порядок установки блокировок и блокируем сначала правый operand, а затем левый, тем самым создав вероятность взаимоблокировки. Мы всегда должны соблюдать один и тот же порядок установки блокировок, как в листинге 7.7.

В листинге 7.7 блокировки устанавливаются в том же порядке, что и в других методах, но если правый operand равен нулю, то блокировка левого operand'a не используется. Это плохо, потому что можно затормозить выполнение другой операции, которой нужен левый operand. Если все же, исходя из специфики

задачи, требуется уменьшить количество блокировок (как в листинге 7.7), а также избежать риска попадания в состояние взаимоблокировки (листинг 7.6), то всякий раз, когда нужно установить блокировку не по порядку, для конкретной части кода, следует снять и снова установить необходимые блокировки, если потребуется, чтобы порядок остался прежним. В листинге 7.8 показан правильный способ реализации метода из листинга 7.7.

Листинг 7.7. Правильный порядок установки блокировок, но одна из блокировок может оказаться ненужной

```
public int Divide()
{
    lock(_leftOperandLock)           ← Сначала блокируется левый
    {                                операнд, даже если это не нужно
        lock(_rightOperandLock)
        {
            if(_rightOperand==0)
            {
                DivideByZeroEvent?.Invoke(this,EventArgs.Empty);
                return 0;
            }
            return _leftOperand/_rightOperand;
        }
    }
}
```

Листинг 7.8. Правильный порядок установки блокировок без установки ненужной блокировки

```
public int Divide()
{
    lock(_rightOperandLock)          ← Сначала блокируется
    {                                только правый operand
        if(_rightOperand==0)
        {
            DivideByZeroEvent?.Invoke(this,EventArgs.Empty);
            return 0;               ← Если правый operand равен нулю,
        }                                то метод завершится здесь
    }
    lock(_leftOperandLock)           | Блокировка правого operand'a
    {                                снимается, чтобы приобрести
        lock(_rightOperandLock)      | блокировки в правильном порядке
        {
            if(_rightOperand==0)
            {
                DivideByZeroEvent?.Invoke(this,EventArgs.Empty);
                return 0;
            }
            return _leftOperand/_rightOperand;
        }
    }
}
```

Повторная проверка
условия, потому
что значение могло
измениться после
освобождения
блокировки

В этом примере код блокирует правый операнд и проверяет его на равенство нулю. Затем блокировка снимается, чтобы получить обе блокировки в правильном порядке. Затем снова правый операнд проверяется на равенство нулю, поскольку он мог измениться за этот крошечный промежуток времени между моментом, когда была снята блокировка правого операнда, и моментом, когда она снова была установлена. И только после этого выполняется деление и возвращается результат.

Но это были простые случаи. Иногда выявить взаимоблокировку сложнее. Предыдущие два листинга с правильным порядком блокировки все еще могут попасть в состояние взаимоблокировки. Давайте еще раз взглянем на этот код (используем более короткий и простой код из листинга 7.7) (листинг 7.9).

Листинг 7.9. Правильный порядок установки блокировок, но взаимоблокировка все еще возможна

```
public int Divide()
{
    lock(_leftOperandLock)
    {
        lock(_rightOperandLock)
        {
            if(_rightOperand==0)
            {
                DivideByZero?.Invoke(this,EventArgs.Empty);
                return 0;
            }
            return _leftOperand/_rightOperand;
        }
    }
}
```

Этот код устанавливает обе блокировки в правильном порядке, а затем, если второй операнд равен нулю, генерирует событие `DivideByZero`, иначе делит первый операнд на второй и возвращает результат. Проблема здесь в вызове обработчика событий `DivideByZero`. Код в этом событии находится вне нашего контроля. Он мог быть написан кем-то из другой организации и выполнять различные операции в разных приложениях. Этот код мог бы, например, устанавливать свои блокировки (листинг 7.10).

Листинг 7.10. Код, вызывающий ошибку взаимоблокировки в листинге 7.9

```
public void SomeMethod()
{
    lock(_outputLock)
    {
        Console.WriteLine(_numbers.Add());
    }
}
```

```
private void Numbers_DivideByZeroEvent(object sender, EventArgs ea)
{
    lock(_outputLock)
    {
        Console.WriteLine("Деление на ноль");
    }
}
```

Этот код устанавливает блокировку перед вызовом метода `Add` из листинга 7.5 и ту же блокировку устанавливает в обработчике события, которое генерирует метод `Divide` из листинга 7.9. Этот код выглядит правильно, как и наш метод `Divide`, который тоже выглядит правильно.

Но если один поток вызовет `SomeMethod`, а другой попытается вызвать `Divide` и произвести деление на ноль, то возникнет ситуация взаимоблокировки. Первый поток сначала установит блокировку на `_outputLock` (в `SomeMethod`), а затем попытается установить `_leftOperandLock` и `_rightOperandLock` (внутри `Add`), в то время как второй поток установит `_leftOperandLock` и `_rightOperandLock` (внутри `Divide`), а затем попытается установить `_outputLock` (в `Numbers_DivideByZeroEvent`).

Первый поток будет удерживать `_outputLock`, ожидая, когда освободятся `_leftOperandLock` и `_rightOperandLock`, а в это время второй поток будет удерживать `_leftOperandLock` и `_rightOperandLock`, ожидая, когда освободится `_outputLock`. Это та же самая проблема, которую мы видели раньше, только теперь в нее вовлечено несколько файлов и ее сложнее отладить.

Так мы подошли к третьему правилу: никогда не вызывайте неподконтрольный вам код, удерживая блокировку. Если нужно вызывать любой неподконтрольный вам код, то вызывайте его после снятия блокировок. В листинге 7.11 показано, как это сделать.

Листинг 7.11. Устранение ошибки взаимоблокировки в обработчике событий

```
public int Divide()
{
    lock(_leftOperandLock)
    {
        lock(_rightOperandLock)
        {
            if(_rightOperand!=0) // Условие изменено на обратное
            {
                return _leftOperand/_rightOperand; // Если правый операнд не равен нулю, то метод завершится здесь
            }
        }
        DivideByZero?.Invoke(this,EventArgs.Empty); // Вызов события после снятия блокировки
    }
    return 0;
}
```

В этой версии, вместо того чтобы проверять, равен ли нулю правый операнд, и вызывать событие, мы убеждаемся, что правый операнд не равен нулю, и выполняем деление. Это означает, что в конце метода, после снятия всех блокировок, правый операнд будет равен нулю, и в этот момент можно безопасно вызывать события.

Вы могли бы подумать, что эту проблему можно решить, если никогда не удерживать более одной блокировки одновременно, но на самом деле это не так, потому что в этом случае может возникнуть состояние гонки.

7.4. Состояние гонки

Состояние гонки возникает, когда результат кода зависит от совпадения неконтролируемых событий. Например, предположим, что кто-то изменил метод `Add` из более раннего варианта и теперь в каждый момент времени он удерживает не более одной блокировки, и тот же разработчик добавил метод `SetOperands`, присваивающий значения обоим операндам и использующий ту же стратегию блокировки (листинг 7.12).

Листинг 7.12. Удержание не более одной блокировки за раз

```
public int Add()
{
    int leftCopy,rightCopy;
    lock(_leftOperandLock)
    {
        leftCopy = _leftOperand;
    }
    lock(_rightOperandLock)
    {
        rightCopy = _rightOperand;
    }
    return rightCopy + leftCopy;
}

public int SetOperands(int left, int right)
{
    lock(_leftOperandLock)
    {
        _leftOperand = left;
    }
    lock(_rightOperandLock)
    {
        _rightOperand = right;
    }
}
```

Метод `Add` блокирует левый операнд, копирует его значение в локальную переменную и тут же снимает блокировку. Затем повторяет ту же процедуру с правым операндом. Аналогично действует и метод `SetOperands`: блокирует один операнд, присваивает ему значение и снимает блокировку, после чего повторяет

те же действия со вторым операндом. Поскольку код не пытается установить блокировку, удерживая другую, он полностью защищен от состояния взаимоблокировки. Однако вместе эти два метода демонстрируют новую проблему. Если эти два метода вызвать одновременно, то мы не сможем с уверенностью сказать, в каком порядке будут выполняться четыре инструкции `lock`. Если повезет, то два блока в одном потоке выполняются вместе — назовем эту ситуацию «хорошим вариантом» (рис. 7.3).

Поток 1 вызывает `Add()`, и в то же время поток 2 вызывает `SetOperands(1,2)`
Оба операнда изначально имеют значение 0

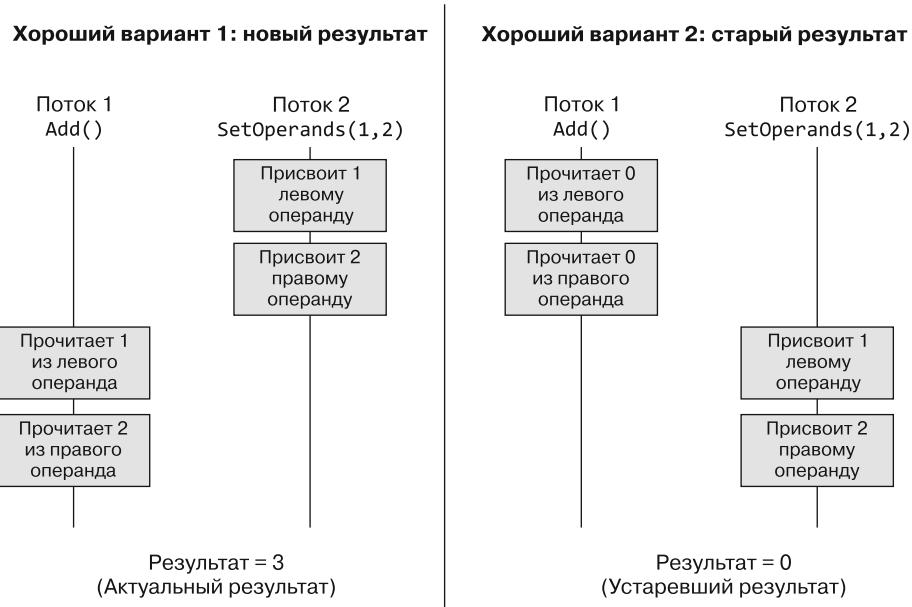


Рис. 7.3. Порядок установки блокировок, при котором получаются правильные результаты

В варианте 1 получается правильный результат: операнды получают новые значения и затем тут же вычисляется результат. В варианте 2 получается устаревший результат, но он устареет всего на миллисекунду, поэтому этот результат я тоже назову правильным. Обычно это нормально, когда результат немного устаревает (приемлемая величина этого «немного» сильно разнится в разных проектах), и чрезвычайно сложно гарантировать неизменную актуальность результата из-за законов физики. Если вы выглянете в окно и увидите снаружи солнечный свет, то можно сделать вывод, что Солнце существовало 8 мин назад (свет идет от Солнца до Земли примерно 8 мин). Возможно (но, к нашему огромному счастью, крайне маловероятно), что в течение последних 8 мин Солнце исчезло и мы просто пока не знаем об этом.

Из-за небольших отклонений в планировании потоков код может получать разные результаты, а это уже состояние гонки. Но ситуация может ухудшиться, если операции, выполняемые двумя потоками, будут чередоваться. Например, две операции в одном потоке могут выполниться между операциями в другом потоке, как показано на рис. 7.4.

Поток 1 вызывает Add(), и в то же время поток 2 вызывает SetOperands(1,2)
Оба операнда изначально имеют значение 0

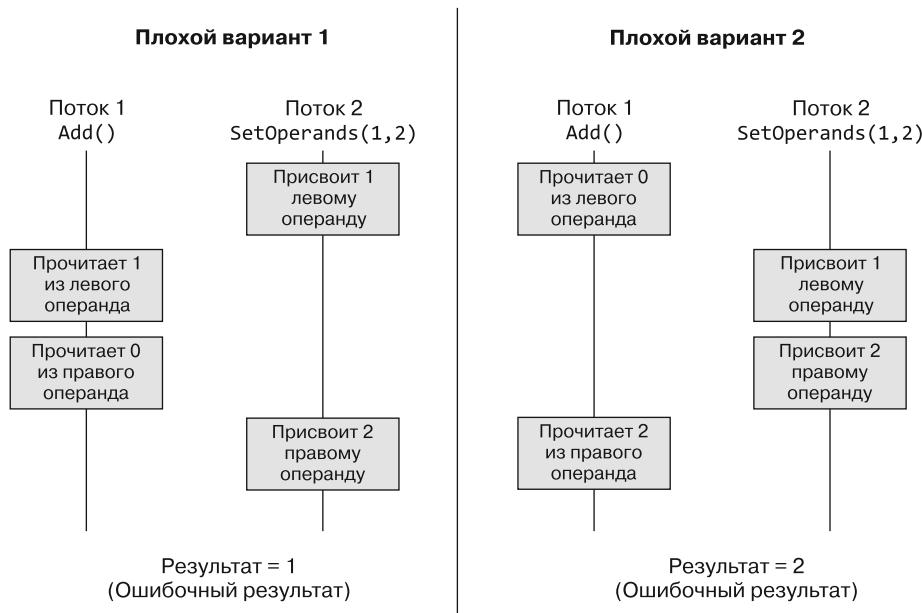


Рис. 7.4. Порядок установки блокировок, при котором получаются ошибочные результаты

В этих двух вариантах мы явно получим неверные результаты. Причина заключается в следующем: поскольку в коде используются отдельные короткие блокировки, нам удалось получить частично обновленные данные, несмотря на использование блокировок. Это связано с тем досадным фактом, что объединение нескольких потокобезопасных операций не обязательно приводит к потокобезопасной операции — каждая из двух блокировок по отдельности потокобезопасна, но две блокировки подряд могут привести к состоянию гонки.

И это подводит нас к четвертому правилу: блокировки должны удерживаться на протяжении всего выполнения операции. Возможно, сейчас вы попытаетесь возразить: «Нет, мы должны удерживать блокировки только в течение минимального периода времени!» — и будете правы, потому что слишком долгое удержание блокировок, скорее всего, приведет к синхронизации.

7.5. Синхронизация

Синхронизация — это ситуация, когда операции происходят последовательно, а не параллельно. Для демонстрации вернемся к листингу 4.11 и кратко повторим наши приключения, пережитые в главе 4. Там мы написали программу, которая считает от 0 до 10 миллионов. Чтобы ускорить счет, мы создали два потока, каждый из которых считает до пяти миллионов, но получили неправильный результат из-за проблемы частичного обновления, которая кратко упоминалась в главе 4 и более подробно обсуждалась в начале этой главы. Исправив проблему с помощью блокировки, мы получили следующее (листинг 7.13).

Листинг 7.13. Многопоточный счет до 10 миллионов

```
public void GetCorrectValue()
{
    int theValue = 0;
    object theLock = new Object();
    var threads = new Thread[2];
    for(int i=0;i<2;++i)
    {
        threads[i] = new Thread(()=>
        {
            for(int j=0;j<5000000;j++)
            {
                lock(theLock)
                {
                    ++theValue;
                }
            }
        });
        threads[i].Start();
    }

    foreach(var current in threads)
    {
        current.Join();
    }
    Console.WriteLine(theValue);
}
```

Этот код создает два потока, каждый из которых увеличивает значение в пять миллионов раз. Чтобы избежать проблемы частичных обновлений, о которой говорилось в начале этой главы, код использует блокировку. Но это не избавляет его от проблемы. Мы создали два потока, чтобы повысить производительность, однако из-за блокировок, защищающих фактические операции увеличения, само увеличение происходит последовательно, а не параллельно. На рис. 7.5 показана схема работы кода.

Первый столбец показывает, что произойдет, если программу увеличения написать как однопоточную: она запустит единственный поток, который досчитает

до 10 миллионов и завершится. Вторые два столбца показывают, что мы хотели получить: два потока, каждый из которых выполняет свою половину работы примерно за половину времени. Последние два столбца — это то, что произошло на самом деле: мы разделили работу между двумя потоками, но всякий раз, когда один из потоков выполняет полезную работу, другой должен ждать, что приводит к отсутствию реального параллелизма и уменьшению скорости по сравнению с однопоточной версией (из-за накладных расходов на синхронизацию потоков).

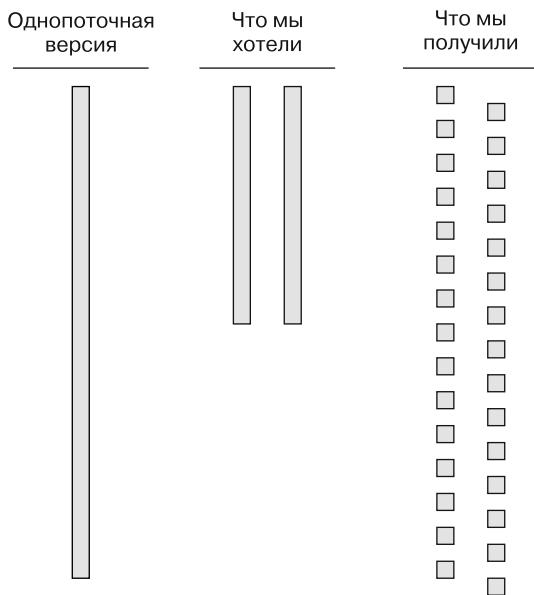


Рис. 7.5. Многопоточность не увеличила производительность из-за синхронизации потоков

Этот пример подводит нас к пятому правилу: чтобы избежать синхронизации, нужно удерживать блокировки как можно более короткое время, желательно только на время, необходимое для доступа к общему ресурсу, а не в течение выполнения всей операции. Если вы подумали, что правило «удерживать блокировки как можно более короткое время, а не в течение выполнения всей операции» противоречит четвертому правилу «удерживать блокировки в течение всей операции», то вы совершенно правы. Если блокировки слишком короткие, есть риск попасть в состояние гонки, а если блокировки слишком долгие, то возникает проблема синхронизации.

Мы должны стремиться к золотой середине, когда блокировки достаточно длительные, чтобы предотвратить состояние гонки, но достаточно короткие, чтобы избежать синхронизации. Однако золотая середина существует не всегда.

В некоторых ситуациях сокращение длительности удержания блокировки до любой величины, которая не вызывает синхронизацию, вызовет состояние гонки. В таких случаях следует помнить, что синхронизация может ухудшить производительность, а состояние гонки приведет к неправильным и неожиданным результатам. Поэтому состояние синхронизации выглядит предпочтительнее состояния гонки.

Синхронизация плоха тем, что не позволяет выполнять действия параллельно, но в некоторых случаях она желательна. Например, в банковском деле очень нежелательно, чтобы деньги переводились со счета на счет дважды, только потому что два поручения на банковский перевод поступили одновременно. Чтобы избежать этой ситуации, международная банковская система (которая является высокодецентрализованной цифровой распределенной системой, управляемой тысячами различных независимых организаций по всему миру) синхронизирует доступ к вашему банковскому счету. Взглянув на историю транзакций своего банковского счета или кредитной карты, вы увидите, что каждая следующая транзакция начинается только после завершения предыдущей.

Даже если две транзакции по кредитной карте начнутся в одно и то же время в двух разных магазинах в разных странах и магазины используют разные платежные процессоры и разные банки (куда более параллельные процессы, чем разные потоки на одном компьютере), система все равно синхронизирует их в упорядоченную последовательность и будет действовать так, будто одна из них началась после завершения другой.

Для синхронизации операций, когда это желательно, можно использовать блокировки, как в примере со счетом до 10 миллионов, или одну из более продвинутых стратегий, которые мы обсудим в следующей главе. Но синхронизация нежелательна, когда параллельные операции непреднамеренно начинают выполняться последовательно и производительность системы снижается, потому что двум или более потокам требуется исключительный доступ к одному и тому же общему ресурсу (обычно блокировке). В таких случаях каждый поток получает ресурс, выполняет небольшую часть работы и освобождает его. Однако иногда один поток может удерживать ресурс очень долго, не давая поработать другому потоку. Эта ситуация называется *голоданием*.

7.6. Голодание

Голодание — это ситуация, когда поток или группа потоков монополизирует доступ к некоторому ресурсу, не позволяя другому потоку или группе потоков выполнить какую-либо работу. Например, следующая программа создает два потока, каждый из которых устанавливает блокировку и в бесконечном цикле выводит символ в консоль (первый поток — знак «минус», а второй поток — букву x) (листинг 7.14).

Листинг 7.14. Голодание из-за блокировки

```
using System.Diagnostics;
var theLock = new object();
var thread1 = new Thread(() =>
{
    lock(theLock)
    {
        while (true)
        {
            Console.Write("-");
        }
    }
});
thread1.Start();
var thread2 = new Thread(() =>
{
    while (true)
    {
        lock (theLock)
        {
            Console.Write("x");
        }
    }
});
thread2.Start();
```

Первый поток устанавливает блокировку перед входом в цикл и снимает ее после цикла (а так как цикл бесконечный, то блокировка не будет снята никогда), а второй поток устанавливает блокировку перед выводом каждого символа и снимает ее сразу после вывода. Запустив эту программу, мы увидим, что она выводит только знаки «минус», потому что первый поток удерживает блокировку на протяжении всего времени выполнения программы, а второй поток просто не получает возможности установить блокировку.

Этот пример возвращает нас к пятому правилу: удерживайте блокировки как можно более короткое время. Выше, обсуждая это правило, я сказал, что удержание блокировки в течение слишком долгого времени может привести к синхронизации. Теперь мы видим, что в экстремальных случаях, когда блокировка удерживается слишком долго, возникает состояние голодания.

Голодание также часто вызывается некоторыми потоками, монополизирующими другие ресурсы, например процессор. Если в программе есть высокоприоритетные потоки, которые не блокируются, они могут помешать выполнению низкоприоритетных потоков. Например, программа в листинге 7.15 создает два потока: первый в бесконечном цикле выводит в консоль знаки «минус», а второй — символы x. Я повысил приоритет второго потока до `AboveNormal` и заставил всю программу использовать только первое ядро процессора (потому что иначе

пришлось бы создать много потоков, которые заняли бы все ядра, и рисковать тем, что после запуска программы компьютер перестанет откликаться на любые наши действия). Этот прием называется *привязкой к процессору*.

Листинг 7.15. Голодание из-за потока с высоким приоритетом

```
using System.Diagnostics;

Process.GetCurrentProcess().ProcessorAffinity = new IntPtr(1); ←
    Заставит
    программу
    выполняться
    только на первом
    ядре процессора

var thread1 = new Thread(() =>
{
    while (true)
    {
        Console.Write("-");
    }
});
thread1.Start();

var thread2 = new Thread(() =>
{
    while (true)
    {
        Console.Write("x");
    }
});
thread2.Priority = ThreadPriority.AboveNormal; ←
    Увеличивает приоритет
    потока
thread2.Start();
```

Если запустить эту программу в Windows, то получим экран, заполненный длинной последовательностью символов x с редкими минусами. Появление минусов объясняется работой механизма антиголодания, который Microsoft добавила в планировщик потоков Windows. В Linux мы увидим примерно одинаковое количество символов x и минусов, потому что по умолчанию Linux не позволяет изменять приоритет потока.

Запуск этой программы в двух упомянутых операционных системах демонстрирует две стороны проблемы изменения приоритета потоков или процессов и привязки к процессору:

- запуск в Windows показал, что даже небольшое повышение приоритета одного потока может существенно ограничить вычислительную мощность, которую могут использовать другие потоки;
- запуск в Linux показал, что приоритет и привязка к процессору иногда работают не так, как ожидается.

И этот пример подвел нас к последнему правилу: не меняйте приоритет или привязку к процессору.

Теперь, когда мы рассмотрели наиболее распространенные ошибки в многопоточном программировании и способы их устранения, можем поговорить о различных стратегиях написания многопоточного кода.

Итоги главы

- Компилятор и процессор могут менять порядок и даже исключать операции доступа к памяти (при условии, что в однопоточной среде конечный результат не изменится). Вы не должны рассчитывать на то, что другие потоки увидят состояние, соответствующее написанному вами коду, если не используете блокировки. Поэтому всегда используйте блокировки при доступе к общим данным.
- Всегда устанавливайте блокировки в одном и том же порядке; это называется *иерархией блокировок*.
- Никогда не вызывайте неподконтрольный вам код, пока удерживается блокировка.
- Объединение нескольких потокобезопасных операций редко дает объединенную потокобезопасную операцию.
- Удерживайте блокировки на всем протяжении выполнения операции.
- Удерживайте блокировки минимально возможное время.
- Последние два пункта противоречат друг другу. Если удерживать блокировки слишком долго, то можно получить состояние синхронизации. Если блокировки удерживаются недостаточно долго, то можно получить состояние гонки. Попробуйте найти золотую середину. Если золотой середины не существует, то выбирайте более долгие блокировки, потому что состояние гонки обычно хуже, чем состояние синхронизации.
- Синхронизация иногда желательна. Некоторые операции должны выполняться последовательно, даже если большая часть системы может работать параллельно.
- Не меняйте приоритет потоков и процессов или привязку к процессору.

Часть II

Продвинутое использование `async/await` и многопоточности

Теперь, когда вы узнали все или почти все об `async/await` и многопоточности, пришло время погрузиться глубже и увидеть, что многопоточность и асинхронное программирование — это гораздо больше, чем ключевое слово `await` и метод `Task.Run`.

Во второй части обсуждаются различные способы обработки данных в фоновом режиме (глава 8), а затем объясняется, как прерывать фоновую обработку (глава 9). После этого вы узнаете, как создавать сложные асинхронные компоненты (глава 10) и настраивать поведение потоков `async/await` (глава 11). Затем мы кратко обсудим сложности с исключениями в асинхронном программировании (глава 12) и поговорим о потокобезопасных коллекциях (глава 13). В последней главе мы обсудим вопросы создания своих асинхронных компонентов, подобных коллекциям (глава 14).

К концу второй части (и книги) вы получите все знания, необходимые для разработки многопоточных приложений. Вы научитесь понимать, как работают все части и как их комбинировать.

Обработка последовательности элементов в фоновом режиме

В этой главе

- ✓ Параллельная обработка элементов.
- ✓ Показатели производительности различных способов параллельного выполнения кода.
- ✓ Последовательная обработка элементов в фоновом режиме.
- ✓ Фоновая обработка важных заданий.

В современных приложениях многопоточность используется по трем основным причинам. Первая и самая распространенная — когда сервер приложений должен одновременно обрабатывать запросы от нескольких клиентов. Обычно обработка подобных задач нам неподконтрольна, она выполняется на уровне используемого нами фреймворка, например ASP.NET. Две другие причины — ускорение вычислений с помощью параллельной обработки по частям и отложенное выполнение некоторых фоновых задач. В обоих случаях многопоточность может значительно улучшить производительность вашей программы (или, по крайней мере, отзывчивость и воспринимаемую производительность). Давайте сначала обсудим первую из этих двух причин: ускорение вычислений за счет параллельной обработки.

8.1. Параллельная обработка элементов

Для демонстрации параллельной обработки напишем самую простую в мире программу рассылки электронных писем. Она составляет индивидуальные сообщения по шаблону для каждого получателя с подстановкой информации о получателе (листинг 8.1).

Листинг 8.1. Самая простая в мире программа рассылки писем по электронной почте

```
void MailMerge(
    string from,
    string subject,
    string text,
    (string email, string name)[] recipients)
{
    var sender = new SmtpClient("smtp.example.com");
    foreach(var current in recipients) ← Цикл по списку
    {
        try
        {
            var message = new MailMessage();
            message.From = new MailAddress(from);
            message.Subject = subject;
            message.To.Add(new MailAddress(current.email));
            message.Body = text.Replace("{name}", current.name); ← Замена шаблонного
            sender.Send(message); ← заполнителя
        }                                         конкретным
        catch                                         значением
        {
            LogFailure(current.email);
        }
    }
}
```

Этот метод принимает адрес отправителя, тему письма, текст шаблона письма, а также список имен и адресов получателей. Затем для каждого получателя подставляет вместо {name} имя получателя и отправляет письмо. Если возникает ошибка, то метод просто выводит сообщение о ней и продолжает выполнение (в реальном коде отправка электронной почты может потерпеть неудачу по многим причинам, большинство из которых носят временный характер, поэтому обычно предусматривается некоторая логика выполнения повторных попыток).

Обратите внимание, что из-за постоянных злоупотреблений со стороны спамеров провайдеры услуг электронной почты очень строго следят за соблюдением условий использования своих сервисов. Если вам понадобится организовать похожую рассылку, убедитесь, что соблюдаете условия вашего провайдера, и подумайте о возможности использовать транзакционный сервис отправки электронной

почты вместо услуг вашего провайдера. Я настоятельно рекомендую никогда не писать программы, отправляющие электронную почту в цикле, если только вы не согласовали ее применение с вашим провайдером.

При использовании этого метода в веб-приложении мы быстро столкнемся с проблемой: отправка электронной почты выполняется медленно и может занять до нескольких секунд на одно сообщение. Типичное время ожидания для веб-запроса составляет 30 с. Это означает, что мы начнем получать ошибку превышения времени ожидания и не будем возвращать результаты пользователю уже между 10 и 40 сообщениями, и это действительно очень небольшое количество сообщений для любой задачи, требующей автоматизированной персонализированной почтовой рассылки.

8.1.1. Параллельная обработка элементов с помощью класса Thread

У нас нет времени ждать, когда все сообщения будут отправлены одно за другим, поэтому очевидным выглядит решение просто отправить все сообщения параллельно. В этом случае мы должны будем ждать ровно столько времени, сколько потребуется для самой долгой отправки сообщения. Чтобы организовать параллельную отправку, можно использовать любой из вариантов, о которых рассказывалось в главе 4, например класс `Thread` (листинг 8.2).

Листинг 8.2. Рассылка электронной почты с созданием отдельного потока для каждого сообщения

```
void MailMerge(
    string from,
    string subject,
    string text,
    (string email, string name)[] recipients)
{
    var processingThreads = new Thread[recipients.Length];
    for(int i=0; i< recipients.Length; ++i)
    {
        processingThreads[i] = new Thread(()=> ←
        {
            try
            {
                var sender = new SmtpClient("smtp.example.com");
                var message = new MailMessage();
                message.From = new MailAddress(from);
                message.Subject = subject;
                message.To.Add(new MailAddress(recipients[i].email));
                message.Body = text.Replace("{name}", recipients[i].name);
                sender.Send(message);
            }
        });
    }
}
```

Каждое сообщение отправляется
в отдельном потоке

```

        catch
        {
            LogFailure(current.email);
        }
    });
    processingThreads[i].Start();
}
foreach(var current in processingThreads)
{
    current.Join();
}
}

```



Ожидаем завершения
всех потоков

Этот код создает поток для каждого отправляемого сообщения, запускает все потоки параллельно, а затем ожидает их завершения.

Эта программа может работать просто отлично, но у нее есть несколько слабых мест. Самое очевидное — отсутствие ограничения на количество потоков. Например, если у нас есть 10 пользователей, работающих одновременно, и каждый отправляет 100 сообщений (и это еще не очень много), то наш код создаст 1000 потоков, и еще неизвестно, как это повлияет на производительность программы. Однако можно попробовать написать небольшую программу, чтобы оценить этот эффект (листинг 8.3).

Листинг 8.3. Тест производительности программы, создающей отдельный поток для каждого сообщения

```

for (int j = 0; j < 5; ++j)
{
    var sw = Stopwatch.StartNew();
    var threads = new Thread[1000];
    for (int i = 0; i < 1000; i++)
    {
        threads[i] = new Thread(() => Thread.Sleep(1000));
        threads[i].Start();
    }
    foreach (var current in threads)
        current.Join();
    sw.Stop();
    Console.WriteLine(sw.ElapsedMilliseconds);
}

```

Эта программа создает 1000 потоков, каждый из которых приостанавливается на 1 с. Все это повторяется пять раз, просто чтобы убедиться, что на результаты не повлияло что-то не связанное с нашим кодом, выполняющимся параллельно.

При компиляции в конфигурации Release и без отладчика моему ноутбуку потребовалось от 1,1 до 1,2 с для выполнения каждой итерации. Это показывает,

что для современных компьютеров накладные расходы, обусловленные обслуживанием 1000 потоков, вполне приемлемы.

Увеличив количество потоков до 10 000, я получил время чуть больше 2 с, а запустив 100 000 потоков, я получил время от 15 до 20 с, и это в программе, которая ничего не делает. На реальном сервере, скорее всего, ситуация будет хуже, потому что сервер должен действительно выполнять полезную работу, а не просто играть с потоками, поэтому, пожалуйста, не создавайте слишком большое количество потоков и не думайте, что накладные расходы будут незначительными.

Обратите внимание, что если запустить программу под отладчиком, то результаты будут намного хуже, потому что отладчик отслеживает создание и уничтожение потоков. Когда я запустил программу с 1000 потоков под отладчиком, то на ее выполнение потребовалось 14 с вместо 1,2 с — будьте осторожны с тестами производительности!

8.1.2. Параллельная обработка элементов с использованием пула потоков

Создание произвольного количества потоков внутри нашего серверного процесса — это плохая идея. К счастью, именно для таких ситуаций был разработан пул потоков. Давайте перенесем обработку сообщений в пул потоков. Мы могли бы запускать наш код в пуле потоков с помощью `ThreadPool.QueueUserWorkItem`, но тогда пришлось бы написать свой механизм определения момента, когда все потоки завершат отправку сообщений. Написать этот код не так уж и сложно, но еще проще не писать его, потому что Microsoft оказалась настолько любезной, что включила эту функцию в `Task.Run` (листинг 8.4).

Листинг 8.4. Рассылка электронной почты с помощью пула потоков

```
void MailMerge(
    string from,
    string subject,
    string text,
    (string email, string name)[] recipients)
{
    var processingTasks = new Task[recipients.Length];
    for(int i=0;i< recipients.Length;++i)
    {
        processingTasks[i] = Task.Run(()=> ←
        {
            try
            {
                var sender = new SmtpClient("smtp.example.com");
                var message = new MailMessage();
                message.From = new MailAddress(from); ←
                Использует Task.Run, чтобы запустить
                отправку сообщения в пуле потоков
            }
        });
    }
}
```

```

        message.Subject = subject;
        message.To.Add(new MailAddress(recipients[i].email));
        message.Body = text.Replace("{name}", recipients[i].name);
        sender.Send(message);
    }
    catch
    {
        LogFailure(current.email);
    }
});
Task.WaitAll(processingTasks); ← Ожидает завершения
}                                     всех потоков
}

```

Это почти тот же код, что и в листинге 8.2, только вместо `new Thread` вызывается `Task.Run`, удален вызов `Thread.Start` и цикл `Join` в конце заменен одним вызовом `Task.WaitAll`.

Это решает проблему потенциального создания огромного количества потоков. Пул ограничит количество потоков разумным для системы числом и избавит от накладных расходов на создание и уничтожение потоков. Однако возникает возможность перенасыщения пула потоков. Если попытаться отправить те же 1000 сообщений, как в предыдущем примере, то пул окажется занятым до тех пор, пока все сообщения не будут отправлены, а тем временем весь остальной код, который тоже пользуется пулем потоков (например, ASP.NET), будет вынужден ждать. Это означает, что при попытке отправить слишком много сообщений сервер может прекратить обработку веб-запросов.

Изменим программу тестирования производительности и посмотрим, как повлиял на производительность переход на пул потоков (листинг 8.5).

Листинг 8.5. Тест производительности пула потоков

```

for (int j = 0; j < 5; ++j)
{
    var sw = Stopwatch.StartNew();
    var tasks = new Task[1000];
    for (int i = 0; i < 1000; i++)
    {
        tasks[i] = Task.Run(() => Thread.Sleep(1000));
    }
    Task.WaitAll(tasks);
    sw.Stop();
    Console.WriteLine(sw.ElapsedMilliseconds);
}

```

Я просто взял код из листинга 8.3 и внес те же изменения, чтобы заменить класс `Thread` пулом потоков.

Запустив этот тест, я получил 69 с для первой, 37 с для второй, 27 с для третьей, 23 с для четвертой и 20 с для пятой и последней итераций. О чём это говорит? Очевидно, что пул потоков оказался перегружен и эта версия выполнялась в 60 раз дольше предыдущей. Но в результатах наблюдается странная тенденция: каждая следующая итерация выполнялась быстрее предыдущей. Причина этого явления в том, что пул потоков всегда автоматически оптимизирует количество потоков и будет ускоряться с каждой итерацией. Это означает, что, несмотря на неприемлемо низкую скорость для программы, которая делает что-то, что редко требует большого количества потоков, в системе, где постоянно используется большое количество потоков, этот прием будет работать очень хорошо.

При разработке программы, которая, как мы знаем, потребует создания большого количества потоков в пуле, можно просто сообщить об этом системе и не ждать автоматических оптимизаций. Если сообщить пулу, что нам потребуется 1000 рабочих потоков, используя следующие две строки кода:

```
ThreadPool.GetMinThreads(out _, out var completionPortThreads);
ThreadPool.SetMinThreads(1000, completionPortThreads);
```

то первая итерация выполнится так же быстро, как и при использовании класса `Thread`, а последующие итерации будут еще быстрее — на моем компьютере на их выполнение затрачивалось в среднем 1025 мс.

8.1.3. Асинхронная параллельная обработка элементов

В предыдущем примере мы вызвали перегрузку пула потоков, добавив слишком много длительно выполняемых заданий в отсутствие достаточного количества доступных потоков для их обработки. В нашем случае потоки являются продолжительно выполняющимися из-за использования блокирующих операций. Если бы эти задания выполняли вычисления, то пул потоков был бы ничуть не медленнее любых других вариантов, потому что тогда производительность ограничивалась бы количеством ядер процессора. Но наша программа тратит большую часть времени на простое ожидание и ничего не делает. Все эти потоки просто блокируются и ничего не делают. Мы уже говорили, что это именно та ситуация, в которой особенно хороши асинхронные техники, поэтому сделаем программу рассылки асинхронной (листинг 8.6).

Мне пришлось внести лишь два небольших изменения в код из листинга 6.4: добавить ключевое слово `async`, вместо `Send` использовать `SendMailAsync` и дождаться результата (асинхронная версия `Send` называется `SendMailAsync`, а не `SendAsync`, потому что это имя уже занято более старым методом, который предшествовал появлению `async/await`).

Листинг 8.6. Асинхронная рассылка электронной почты с использованием Task.Run

```

void MailMerge(
    string from,
    string subject,
    string text,
    (string email, string name)[] recipients)
{
    var processingTasks = new Task[recipients.Length];
    for(int i=0; i< recipients.Length; ++i)
    {
        processingTasks[i] = Task.Run(async ()=> ← Добавлена
        {
            try ← инструкция async
            {
                var sender = new SmtpClient("smtp.example.com");
                var message = new MailMessage();
                message.From = new MailAddress(from);
                message.Subject = subject;
                message.To.Add(new MailAddress(recipients[i].email));
                message.Body = text.Replace("{name}", recipients[i].name);
                await sender.SendMailAsync(message); ← Добавлена
            } ← инструкция await
            catch
            {
                LogFailure(current.email);
            }
        });
    }
    Task.WaitAll(processingTasks);
}

```

И конечно, я обновлю тест производительности, чтобы и его сделать асинхронным. Для этого нужно только заменить `()=>Thread.Sleep(1000)` на `async ()=>await Task.Delay(1000)` (листинг 8.7).

Листинг 8.7. Асинхронный тест производительности

```

for (int j = 0; j < 5; ++j)
{
    var sw = Stopwatch.StartNew();
    var tasks = new Task[1000];
    for (int i = 0; i < 1000; i++)
    {
        tasks[i] = Task.Run(async () => await Task.Delay(1000));
    }
    Task.WaitAll(tasks);
    sw.Stop();
    Console.WriteLine(sw.ElapsedMilliseconds);
}

```

Как применение асинхронности повлияло на производительность? Запустив код, я получил результаты от 1001 до 1017 мс, а это означает, что в данном случае пул потоков практически не имеет накладных расходов. Важно помнить, что в программе рассылки код будет тратить большую часть времени на ожидание, но в нашем тесте производительности он тратит *все* свое время на ожидание, поэтому полученные здесь результаты не отражают поведения реальной программы (я уже говорил, что нужно быть осторожными с тестами производительности).

8.1.4. Класс Parallel

Во всех предыдущих примерах использовался цикл, который создавал потоки или добавлял задания в пул потоков, и затем программа ждала завершения объектов `Thread` или `Task`. Это как раз тот тип шаблонного кода, написание которого не доставляет никакого удовольствия. К счастью, в библиотеке .NET есть класс `Parallel`, который может сделать все это автоматически.

Класс `Parallel` имеет четыре статических метода.

- `Invoke` — принимает массив делегатов и выполняет их все, по возможности параллельно. Этот метод возвращает управление после того, как все делегаты завершат работу.
- `For` — действует как цикл `for`, но итерации выполняются параллельно. Возвращает управление после того, как все итерации завершатся.
- `ForEach` — действует как цикл `foreach`, но итерации выполняются параллельно. Возвращает управление после того, как все итерации завершатся.
- `ForEachAsync` — действует подобно `ForEach`, но роль тела цикла играет асинхронный метод. Он немедленно возвращает управление и задачу `Task`, которая завершится, когда все итерации завершатся.

В этой главе мы рассмотрим `Parallel.ForEach` и `Parallel.ForEachAsync`, так как они наиболее полезны для нашего примера рассылки. Внутренне `Invoke` и `For` используют тот же код, что и `ForEach`. Вот как будет выглядеть код, если мы используем класс `Parallel` (листинг 8.8).

Листинг 8.8. Рассылка электронной почты с помощью класса Parallel

```
void MailMerge(
    string from,
    string subject,
    string text,
    (string email, string name)[] recipients)
{
    var processingTasks = new Task[recipients.Length];
    Parallel.ForEach(recipients,
        (current, _) =>
```

```

    {
        try
        {
            var sender = new SmtpClient("smtp.example.com");
            var message = new MailMessage();
            message.From = new MailAddress(from);
            message.Subject = subject;
            message.To.Add(new MailAddress(current.email));
            message.Body = text.Replace("{name}", current.name);
            sender.Send(message);
        }
        catch
        {
            LogFailure(current.email);
        }
    });
}

```

Как видите, этот код выглядит ближе к исходному однопоточному коду из листинга 8.1. По сути, мы заменили `foreach` на `Parallel.ForEach`, что заставило наш код выполнятся параллельно. Игнорируемый параметр — это токен отмены. Мы поговорим о нем в следующей главе.

Класс `Parallel` также поддерживает отмену задания, настройку планировщика и управление максимальным количеством заданий, обрабатываемых одновременно. Отмену легко реализовать самостоятельно, и мы поговорим об этом в следующей главе. Использование планировщиков тоже широко поддерживается, и о них мы поговорим в главе 11. Управление максимальным количеством заданий, обрабатываемых параллельно, доступно не везде и, как ни странно, является самой большой ловушкой класса `Parallel`. Если перенести тест производительности на использование `Parallel.ForEach`, то получится следующее (листинг 8.9).

Листинг 8.9. Тест производительности Parallel.ForEach

```

for (int j = 0; j < 5; ++j)
{
    var items = Enumerable.Range(0, 1000).ToArray();
    var sw = Stopwatch.StartNew();
    Parallel.ForEach(items,
        (item) => Thread.Sleep(1000));
    sw.Stop();
    Console.WriteLine(sw.ElapsedMilliseconds);
}

```

Эта версия теста производительности создает массив чисел от 0 до 999 и использует `Parallel.ForEach` для их обхода, приостанавливаясь на каждом элементе на 1 с. Я ожидал, что этот код покажет точно такую же производительность, что и код с `Task.Run` в листинге 8.5, потому что это просто другой синтаксис для

выполнения тех же самых действий. Но когда я запустил его, то был удивлен. Первая итерация выполнялась 48 с — быстрее, чем почти 70 с, полученные при использовании `Task.Run`. Однако все последующие итерации выполнялись 31 с — быстрее первых двух итераций в реализации с `Task.Run`, но медленнее третьей и последующих итераций.

По умолчанию `Parallel.ForEach` ограничивает количество заданий, выполняемых параллельно, поэтому он не привел к такой сильной нагрузке на пул потоков, как код с `Task.Run`. Однако именно из-за самоограничения механизм оптимизации пула потоков не создал много потоков, и поэтому последующие итерации с `Parallel.ForEach` выполнялись медленнее.

Для проверки этой теории увеличим максимальное количество заданий, которые будут обрабатываться параллельно, заменив строку `Parallel.ForEach` на

```
Parallel.ForEach(items,
    new ParallelOptions { MaxDegreeOfParallelism = 1000 },
    (item)=>Thread.Sleep(1000));
```

Если установить параметр `MaxDegreeOfParallelism` равным длине списка (и тем самым заставить `Parallel.ForEach` выполнять все задания одновременно), мы получим точно такие же характеристики производительности, как и в примере с `Task.Run` в листинге 8.5. Это противоречит официальной документации, в которой четко говорится, что по умолчанию используются все потоки, а установка `MaxDegreeOfParallelism` может только уменьшить их количество и никогда не увеличивает его. Это означает, что `Parallel.ForEach` очень хорошо подходит для обработки коротких коллекций и когда пул потоков еще не успел создать много потоков.

Обратите внимание, что всякий раз, когда обнаруживается противоречие между наблюдаемым и задокументированным поведением, у нас возникает проблема. Очевидно, что мы не можем полагаться на документированное поведение, потому что оно не соответствует действительности. Но и надеяться на наблюдаемое поведение тоже рискованно, потому что любое обновление может исправить ошибку и заставить систему работать так, как указано в документации. Поэтому мы должны писать код так, чтобы он хорошо работал как с задокументированным, так и с наблюдаемым поведением, либо пойти на риск, что придется выпустить экстренное обновление, если это противоречие будет исправлено в будущем.

В предыдущих примерах, где использовался `Task.Run`, мы получили огромный прирост скорости, перейдя от блокирующих операций (листинги 8.4 и 8.5) к асинхронным (листинги 8.6 и 8.7). К сожалению, этого не происходит при переключении с `Parallel.ForEach` к его `async/await`-совместимому аналогу `Parallel.ForEachAsync`. В отличие от `Parallel.ForEach`, параметр `MaxDegreeOfParallelism` по умолчанию (согласно документации) получает значение, равное числу ядер,

и это логически и теоретически наиболее эффективное число потоков для асинхронного кода. Однако проблема в том, что `Parallel.ForEachAsync` использует это как количество одновременно обрабатываемых заданий, а не как количество потоков.

Например, наш код ждет асинхронно 1 с 1000 раз, а у моего ноутбука 12 ядер, поэтому `Parallel.ForEachAsync` начнет обрабатывать первые 12 элементов. Обработает их все ровно за 1 с, а затем начнет обрабатывать следующие 12 элементов. Общее время работы составит 84 с (потому что деление 1000 на 12 с округлением вверх дает 84).

Такое поведение трудно оправдать, и если оно не изменится в будущей версии .NET, то я рекомендовал бы избегать `Parallel.ForEachAsync` или, если вы будете вынуждены его использовать, выбирать подходящее значение для `MaxDegreeOfParallelism`.

Для полноты картины в листинге 8.10 приводится версия реализации с `Parallel.ForEachAsync`.

Листинг 8.10. Асинхронная рассылка электронной почты
с помощью класса Parallel

```
void MailMerge(
    string from,
    string subject,
    string text,
    (string email, string name)[] recipients)
{
    var processingTasks = new Task[recipients.Length];
    Parallel.ForEachAsync(recipients, ← Используется Parallel.ForEach
        new ParallelOptions {
            MaxDegreeOfParallelism = recipients.Length ← Не забывайте
            }, ← Превращает тело цикла
            async (current,_) => ← в асинхронную лямбда-функцию
    {
        try
        {
            var sender = new SmtpClient("smtp.example.com");
            var message = new MailMessage();
            message.From = new MailAddress(from);
            message.Subject = subject;
            message.To.Add(new MailAddress(current.email));
            message.Body = text.Replace("{name}", current.name);
            await sender.SendMailAsync(message); ← Ожидает завершения
        }
        catch
        {
            LogFailure(current.email);
        }
    }).Wait(); ← Ожидает завершения
} ← всех потоков
```

В этот код внесены следующие изменения:

- тело цикла преобразовано в асинхронную лямбда-функцию, вместо `Send` используется `SendMailAsync` и добавлено ожидание его завершения (практически такие же изменения были внесены при преобразовании примера с `Task.Run` в асинхронную версию в листинге 8.6);
- для задачи, возвращаемой из `Parallel.ForEachAsync`, вызывается метод `Task.Wait()`, чтобы дождаться завершения обработки всех заданий (в листинге 8.6 для той же цели использовался метод `Task.WaitAll`);
- наконец, параметру `MaxDegreeOfParallelism` присваивается длина списка. Возможно, это не самое оптимальное значение, но оно намного лучше значения по умолчанию.

8.2. Последовательная обработка элементов в фоновом режиме

Все предыдущие примеры ожидали отправки всех сообщений, но ничего не делали с результатом. Мы могли бы перенести операцию отправки в фоновый поток и вернуть ответ пользователю немедленно, не дождаясь результата. По сути, если не ждать завершения обработки всех сообщений, то нам безразлично, сколько времени займет их отправка.

То есть, просто переместив цикл отправки электронной почты в фоновый поток, мы решим все наши проблемы с производительностью. И, как бонус, наш провайдер услуг электронной почты будет относиться к нам лояльнее, потому что мы не пытаемся отправлять одновременно большое количество сообщений.

8.2.1. Последовательная обработка элементов в фоновом режиме с помощью класса `Thread`

Еще в начале главы, задавшись целью реализовать параллельную обработку заданий (листинг 8.2), мы первым делом использовали класс `Thread`, поэтому будет логично начать обсуждение этой темы с применения класса `Thread` (листинг 8.11).

Этот код очень похож на код в листинге 8.2. Фактически единственное отличие состоит в том, что поток создается вне цикла. У нас по-прежнему нет ограничения на количество потоков, которые можно создать, но теперь код запускает ровно один поток на каждый вызов метода, а не на каждое сообщение, поэтому влияние на производительность должно быть действительно незначительным.

Важно отметить, что если попытаться выйти из программы как обычно (чего никогда не происходит в приложениях ASP.NET, но бывает в командной строке и настольных приложениях с графическим пользовательским интерфейсом), то программа не завершится, пока поток не закончит отправку всех сообщений. Это может быть как преимуществом, так и недостатком, в зависимости от того, с какой стороны посмотреть. Если нужно, чтобы программа закончила работу, не дожидаясь завершения потока, то можно установить свойство `IsBackground` потока в значение `true`.

Листинг 8.11. Перемещение всего цикла в отдельный поток

```
void MailMerge(
    string from,
    string subject,
    string text,
    (string email, string name)[] recipients)
{
    var processingThread = new Thread(()=> ← Создает
    {
        var sender = new SmtpClient("smtp.example.com");
        foreach(var current in recipients)
        {
            try ← А не здесь
            {
                var message = new MailMessage();
                message.From = new MailAddress(from);
                message.Subject = subject;
                message.To.Add(new MailAddress(current.email));
                message.Body = text.Replace("{name}", current.name);
                sender.Send(message);
            }
            catch
            {
                LogFailure(current.email);
            }
        });
        processingThread.Start();
    }
}
```

Прием с запуском нового потока для выполнения каких-либо операций в фоновом режиме особенно хорош в однопользовательских приложениях, таких как настольные приложения с графическим интерфейсом. Работа в фоновом режиме должна выполняться лишь изредка, и если приложение действительно создает слишком много потоков и перегружает процессор, то единственный человек, который пострадает от снижения производительности, — это пользователь, заставивший приложение заняться тяжелой работой. Это не относится

к серверам. На серверах (и в других сценариях с привлечением нескольких пользователей), как правило, приходится управлять нагрузкой и не допускать, чтобы отдельный пользователь чрезмерно нагружал систему. Вот почему на серверах важно контролировать количество потоков, для чего обычно используется паттерн очереди заданий.

8.2.2. Паттерн очереди заданий и `BlockingCollection`

Если нас не волнует время, необходимое для отправки сообщений, то лучше просто использовать один поток или небольшое фиксированное количество потоков, которые займутся отправкой. Такой подход называется *паттерном очереди заданий* и реализуется путем создания очереди, в которую каждый поток может добавлять задания, и имеется выделенный набор потоков, обрабатывающих эти задания. Эти потоки реализуют простой цикл чтения следующего элемента из очереди и его обработки. Чтобы сохранить простоту наших примеров, мы будем использовать только один обрабатывающий поток.

При создании такой очереди приходится учитывать удивительно большое количество деталей, но Microsoft была настолько любезна, что большую часть этой работы выполнила за нас, предоставив класс `BlockingCollection<T>`.

`BlockingCollection<T>` можно использовать несколькими способами, например как потокобезопасный `List<T>`. Но нас интересует только сценарий использования `BlockingCollection<T>` в качестве очереди заданий, и в этом случае нам нужны только три метода.

- `Add` — как и ожидалось, этот метод добавляет новое задание в конец очереди.
- `CompleteAdding` — указывает, что добавление заданий прекращено и поток, обрабатывающий задания, может завершиться, когда обработает все задания, находящиеся в очереди.
- `GetConsumingEnumerable` — возвращает объект, который можно использовать в цикле `foreach` для перебора всех элементов в очереди. Если очередь пуста, то `foreach` заблокируется до момента, пока в очередь не будет добавлено очередное задание или не будет вызван метод `CompleteAdding`. После вызова `CompleteAdding` заблокированный метод `GetConsumingEnumerable` сообщит, что заданий в очереди больше нет, и цикл `foreach` завершится.

Поскольку этот пример получился немного длиннее предыдущих, я реализовал его как класс, а не как метод. Начнем с определения класса для всей информации, которую нужно сохранить в очереди, и самой очереди `BlockingCollection` (листинг 8.12).

Листинг 8.12. Очередь заданий на основе BlockingCollection

```
public class MailMerger
{
    private record MailInfo(
        string from,
        string subject,
        string text,
        string email);

    BlockingCollection<MailInfo> _queue = new();
```

Далее создадим поток для обработки очереди (код, выполняемый в этом потоке, находится в `BackgroundProc`):

```
public void Start()
{
    var thread = new Thread(BackgroundProc);
    thread.Start();
}
```

Добавим также метод, останавливающий обработку вызовом `CompleteAdding`. Вызов этого метода приведет к завершению фонового потока после того, как все задания в очереди будут обработаны:

```
public void Stop()
{
    _queue.CompleteAdding();
}
```

Теперь добавим метод, действующий как `MailMerge` в предыдущих примерах. В этом примере метод только добавляет задания в очередь, но не отправляет электронные письма. Мы запустим здесь цикл подготовки сообщений и добавления их в очередь. Подготовка сообщений перед добавлением в очередь или после чтения из нее в этом примере не имеет никакого значения, но она важна для постоянных очередей (мы поговорим об этом несколькими абзацами ниже):

```
public void MailMerge(
    string from,
    string subject,
    string text,
    (string email, string name)[] recipients)
{
    foreach(var current in recipients)
    {
        _queue.Add(new MailInfo(
            from,
            subject,
            text.Replace("{name}", current.name),
            current.email));
    }
}
```

И наконец, код, который вы все ждали, — код, который выполняется в отдельном потоке и отправляет сообщения. Этот метод не содержит ничего примечательного. Он просто использует `foreach` для обхода заданий, возвращаемых вызовом `BlockingCollection<T>.GetConsumingEnumerable()`, и отправляет сообщение:

```
private void BackgroundProc()
{
    var sender = new SmtpClient("smtp.example.com");
    foreach (var current in _queue.GetConsumingEnumerable())
    {
        try
        {
            var message = new MailMessage();
            message.From = new MailAddress(current.from);
            message.Subject = current.subject;
            message.To.Add(new MailAddress(current.email));
            message.Body = current.text.Replace("{name}", current.name);
            sender.Send(message);
        }
        catch
        {
            LogFailure(current.email);
        }
    }
}
```

Это полная реализация очереди заданий для программы рассылки по электронной почте. Мы использовали класс `Thread`, потому что поток работает продолжительное время — вероятно, он будет работать в течение всего времени выполнения программы — и при использовании пула потоков мы просто израсходуем один из потоков в пуле и лишим систему возможности повторно использовать его для чего-то еще после того, как мы закончим с ним работать (потому что мы никогда не закончим с ним работать). Еще в главе 5 мы видели, что класс `Thread` плохо работает с асинхронным кодом, но класс `BlockingCollection` не является асинхронным и не имеет асинхронной версии, которая не блокируется. Однако в главе 10 я покажу, как создать его асинхронную версию.

`BlockingCollection` хранится только в памяти, то есть в случае аварии или при принудительном завершении процесса другим способом (включая перезагрузку компьютера или выдергивание шнуря питания из розетки) все необработанные задания, находящиеся в очереди, будут потеряны. То есть такая программа не дает гарантий, что все задания, поставленные в очередь, будут обработаны. Чтобы получить более надежную реализацию очереди заданий, нужно использовать постоянные очереди.

8.2.3. Обработка важных элементов с помощью постоянных очередей

Во всех предыдущих примерах при аварийном завершении процесса (а в некоторых из них даже при обычном выходе из процесса) все не обработанные к этому моменту сообщения будут потеряны. Поэтому в случаях, когда это неприемлемо, следует использовать постоянные очереди (persistent queues), иногда их еще называют долговечными (durable).

Постоянные очереди — это самые обычные очереди, но хранящиеся на диске, а не в памяти, поэтому они не теряются в случае сбоя программы. Вы можете написать свою такую очередь, которая сохраняет задания в таблице базы данных, или использовать отдельный сервер очередей. Если вы работаете в облачной среде, то у вашего провайдера облачных услуг, вероятно, уже есть недорогой и простой в использовании сервис очередей (в AWS есть SQS, а в Azure — Storage Queues). Другой распространенный вариант — бесплатный сервер RabbitMQ. Однако обсуждение подробностей использования Azure, AWS или RabbitMQ выходит за рамки этой книги.

При использовании постоянной очереди чтение следующего задания и его удаление из очереди могут быть разными операциями. Это важно, поскольку позволяет определять, как будут развиваться события в случае сбоя.

Первый вариант — удалить задание из очереди только после его обработки. В этом случае если бродячий кот выдернет шнур питания из розетки сразу после завершения обработки задания, но до его удаления из очереди, то после перезагрузки компьютера мы не будем знать, что задание уже было обработано, и оно обрабатывается снова. Это называется «доставкой не менее одного раза».

Второй вариант — удалить задание из очереди до его обработки. В этом случае если наша собака, виляя хвостом от радости, что видит нас, нажмет клавишу выключения питания после удаления задания из очереди, но до его обработки, то после перезагрузки компьютера задания не окажется в очереди и оно никогда не будет обработано. Это называется «доставка не более одного раза».

На самом деле для нас предпочтительнее иметь гарантию, что каждый элемент будет обработан один и только один раз. Это называется «доставка строго один раз», и, к сожалению, реализовать такую логику работы обычно невозможно. Например, в нашей программе рассылки, даже если очередь поддерживает семантику доставки строго один раз, потеря соединения с почтовым сервером после завершения отправки сообщения, но до получения подтверждения от сервера лишит нас возможности узнать, было ли отправлено сообщение. И мы вновь оказываемся в той же ситуации, когда должны либо рискнуть отправить сообщение дважды, либо рискнуть не отправить его вообще.

Почти всегда потеря данных хуже их дублирования, поэтому чаще всего выбирается семантика доставки не менее одного раза. Но если выбирается семантика доставки не менее одного раза и в очереди появляется задание, обработка которого приводит к сбою программы, то программа застрянет в бесконечном цикле: запуск, чтение первого задания из очереди, сбой при попытке его обработать и перезапуск. Вот почему важно иметь что-то, что следит за сбоями в коде, выполняющем обработку (можно обернуть код обработки в блок `try-catch`), и, если обработка одного и того же задания несколько раз терпит неудачу, удаляет это задание из очереди.

Задания, которые всегда вызывают сбой, называются *ядовитыми (poison) сообщениями*, и лучшей практикой считается сохранение их где-то еще (часто в другой постоянной очереди), чтобы иметь возможность проверить задание и найти причину сбоя. Очереди, в которых хранятся такие задания, а также задания, которые не были обработаны по другим причинам, часто называются *очередями недоставленных сообщений*.

Важно также подумать о сбоях при проектировании заданий, сохраняемых в очереди. Вот почему последний пример в листинге 8.12 подготавливал сообщения перед их добавлением в очередь. Таким образом, сбой обработки задания повлияет только на одно задание, а не на все задания в операции рассылки.

Итоги главы

- Если есть задания, которые обрабатываются индивидуально, то, обрабатывая их параллельно, можно ускорить процесс.
- Для параллельной обработки можно использовать класс `Thread`. Это хорошее решение, но оно является ресурсоемким.
- Пул потоков можно использовать с помощью `ThreadPool.QueueUserWorkItem` или `Task.Run`. Пул потоков достаточно эффективен и способен автоматически подстраиваться под характер использования. Но может потребоваться некоторое время, чтобы достичь пиковой производительности, если переложить на него слишком много работы. Этот недостаток можно смягчить с помощью настроек пула потоков, если необходимое количество потоков известно заранее.
- Пул потоков особенно эффективен при работе с асинхронным кодом.
- Класс `Parallel` предлагает более простой синтаксис использования пула потоков, но если использовать его для обработки больших коллекций, то следует провести тестирование производительности, чтобы подобрать хорошее значение для `MaxDegreeOfParallelism`.
- Если вас не волнует, сколько времени займет выполнение операции, и нужно просто освободить текущий поток, то можно запустить последовательную обработку заданий в фоновом режиме.

- Для организации фонового потока можно использовать класс `Thread` или пул потоков. Оба варианта прекрасно справляются с задачей.
- Однако лучше всего реализовать паттерн очереди заданий, например, на основе класса `BlockingCollection`.
- Если вы не хотите терять данные в случае сбоя программы, то следует использовать постоянную очередь. Ее можно реализовать вручную, на основе базы данных, или использовать специализированное решение для очередей, такое как RabbitMQ, AWS SQS или Azure Storage Queues.
- Используя постоянные очереди, необходимо подумать о выборе между семантиками «доставка не менее одного раза» и «доставка не более одного раза». Необходимо также предусмотреть обработку отправленных сообщений.

9

Отмена фоновых задач

В этой главе

- ✓ Отмена операций.
- ✓ Классы `CancellationToken` и `CancellationTokenSource`.
- ✓ Реализация тайм-аутов.
- ✓ Объединение источников отмены.

В предыдущей главе мы говорили о том, как запустить что-то в фоновом режиме. В этой главе мы разберемся, как прервать такую работу. Библиотека .NET предоставляет стандартный механизм, с помощью которого можно сообщить фоновой операции, что она должна завершиться, — класс `CancellationToken`. Этот класс можно использовать для отмены (почти) всех операций, поддерживаемых библиотекой .NET и многими сторонними библиотеками.

9.1. Введение в `CancellationToken`

В этой главе нам понадобится пример продолжительной операции, которую мы могли бы отменять. Так что давайте напишем короткую программу, которая будет работать максимально долго — вечно (листинг 9.1).

Листинг 9.1. Фоновый поток, выполняющийся бесконечно

```
var thread = new Thread(BackgroundProc);
thread.Start();
Console.ReadKey();
void BackgroundProc()
{
    int i=0;
    while(true)
    {
        Console.WriteLine(i++);
    }
}
```

Эта программа запускает поток, который работает вечно. Затем ждет, пока пользователь нажмет любую клавишу, и когда пользователь наконец сделает это, ничего не происходит. Программа не завершится, пока не остановится второй поток, а поскольку мы не предусмотрели никакого механизма, который заставил бы его остановиться, программа продолжит выполняться вечно, точнее до того, как не будет остановлена с применением каких-либо внешних средств для завершения всего процесса (например, можно использовать диспетчер задач, команду `taskkill`, отладчик, сочетание клавиш `Ctrl+C`, перезагрузку системы и т. д.).

Самый простой способ реализовать возможность завершения программы — поставить поток как фоновый. Процесс завершается, когда завершается последний поток, не отмеченный как фоновый, поэтому, чтобы программа закончила работу после нажатия клавиши пользователем, достаточно добавить следующую строку кода перед вызовом `Thread.Start`:

```
thread.IsBackground = true;
```

В некоторых случаях такой способ может решить проблему, но у него есть два недостатка:

- его можно использовать только для отмены операции при завершении всей программы;
- фоновый поток будет остановлен на середине выполняемой им операции и не будет иметь возможности завершить операцию или сохранить свое состояние (но из-за этого программа не останется в нестабильном состоянии, потому что она перестанет выполнять).

Первая проблема уже делает этот способ непригодным для большинства сценариев. Если внимательно поискать способ остановить поток, не завершая всю программу, то можно заметить метод `Abort` в классе `Thread`, чье имя кажется многообещающим. Однако этот метод страдает второй проблемой. На самом

деле он может приводить к еще более тяжелым последствиям, чем первый способ, потому что завершение потока в середине операции может оставить всю программу в несогласованном состоянии, если, например, в момент остановки поток выделял память и обновлял внутренние структуры данных менеджера памяти.

Это делает `Abort` слишком опасным в использовании, настолько, что компания Microsoft запретила его в .NET Core, .NET 5 и более поздних версиях (теперь он просто генерирует исключение `PlatformNotSupportedException` на всех платформах).

Итак, в отсутствие встроенного способа остановить поток нам остается только реализовать свой способ. Начнем с самого простого возможного варианта — обычного флага, который сообщает потоку, когда тот должен остановиться (листиng 9.2).

Листинг 9.2. Использование флага для остановки фонового потока

```
var thread = new Thread(BackgroundProc);
bool isCancellationRequested = false; ← Переменная с флагом
thread.Start();
Console.ReadKey();
isCancellationRequested = true; ← Устанавливает флаг,
void BackgroundProc()           чтобы завершить поток
{
    int i=0;
    while(true)
    {
        if(isCancellationRequested) return; ← Если флаг установлен,
        Console.WriteLine(i++);           то завершается
    }
}
```

Этот вариант работает в текущей версии .NET и на текущем оборудовании, но его работоспособность не гарантируется. Как рассказывалось в главе 7, высокопроизводительные процессоры могут иметь отдельную кэш-память для каждого ядра, и когда главный поток устанавливает флаг, он фактически изменяет версию в кэше своего ядра. Аналогично, когда фоновый поток проверяет флаг, он может читать версию из кэша другого ядра. На компьютере, где ведется разработка, обычно меньшее количество ядер, чем на сервере, и много запущенных программ (например, среда разработки и браузер), поэтому ядрам процессора приходится часто переключать потоки и процессы, и эта проблема никогда не возникнет. Но если запустить программу на высокопроизводительном сервере с большим количеством ядер и меньшим количеством процессов, то отмена может быть задержана, потому что изменение значения флага не будет замечено фоновым потоком, пока оба ядра не актуализируют свои кэши.

Дополнительно в главе 7 обсуждалось, что компилятору разрешено переписывать ваш код, чтобы он работал быстрее, при условии, что это *не изменит результат выполнения в однопоточной среде*. А в однопоточной среде флаг не может измениться во время цикла, поэтому компилятор может посчитать, что проверку флага можно безопасно удалить. Эту проблему особенно сложно отладить, потому что она, как правило, происходит только в сборках, выполненных в конфигурации Release (отладочные сборки обычно не оптимизируются), и только в некоторых средах. То есть код может отлично работать на вашей машине для разработки и давать сбой на рабочем сервере. Более того, сегодня он может работать на сервере, но начать давать сбои в будущем, когда что-то на сервере будет обновлено.

Решение, показанное в главе 7, заключается в использовании блокировок при доступе к флагу. Существуют другие, более эффективные и быстрые способы защитить доступ к одной переменной `bool`, но для простоты я использую оператор `lock` (листинг 9.3). Не волнуйтесь. В следующих листингах мы заменим его чем-нибудь более подходящим.

Листинг 9.3. Использование блокировок для защиты флага отмены

```
var thread = new Thread(BackgroundProc);
var cancelLock = new object();
bool isCancellationRequested = false;
thread.Start();
Console.ReadKey();           ← Блокирует
lock(cancelLock)             установку флага
{
    isCancellationRequested = true;
}

void BackgroundProc()
{
    int i=0;
    while(true)
    {
        lock(cancelLock)   ← Блокирует
        {
            if(isCancellationRequested) return;
        }
        Console.WriteLine(i++);
    }
}
```

Я взял предыдущий пример и заключил все операции с флагом в инструкцию `lock`. Теперь у нас есть потокобезопасный и устойчивый способ остановить фоновый поток. Но мы создали проблему с поддержкой кода. Рано или поздно будущий член команды забудет добавить блокировку и внесет ошибку, которая случается только в промышленном окружении под нагрузкой. Это плохо, но, к счастью, объектно-ориентированное программирование уже решило эту

проблему более 50 лет назад (объектно-ориентированное программирование было впервые формализовано в 1967 году): нужно просто написать класс, инкапсулирующий флаг и контролирующий доступ к нему (листинг 9.4).

Листинг 9.4. Упаковка флага отмены в класс

```
public class CancelFlag
{
    private bool _isCancellationRequested;
    private object _lock = new();

    public void Cancel()
    {
        lock(_lock)
        {
            _isCancellationRequested = true;
        }
    }

    public bool IsCancellationRequested
    {
        get
        {
            lock(_lock)
            {
                return _isCancellationRequested;
            }
        }
    }
}
```

Этот класс прост до крайности: он имеет метод `Cancel`, позволяющий установить флаг отмены, и свойство `IsCancellationRequested`, с помощью которого можно проверить значение флага отмены. Внутри каждого из них доступ к флагу защищен блокировками.

Теперь воспользуемся классом `CancelFlag` в программе:

```
var thread = new Thread(BackgroundProc);
var shouldCancel = new CancelFlag(); ← Создает флаг отмены
thread.Start();
Console.ReadKey();
shouldCancel.Cancel(); ← Устанавливает флаг
void BackgroundProc()
{
    int i=0;
    while(true)
    {
        if(shouldCancel.IsCancellationRequested) return; ← Проверяет флаг
        Console.WriteLine(i++);
    }
}
```

Итак, мы создали потокобезопасный, устойчивый и поддерживаемый способ остановки фонового потока. Но (куда же без «но», ведь мы даже не близко к концу главы) `CancelFlag` имеет слабое место, допуская возможность злоупотребления и использования его таким образом, что это будет иметь неожиданные последствия для других частей программы. Например, если добавить еще один фоновый поток, который иногда должен останавливать сам себя, это может выглядеть примерно так:

```
void SomeOtherBackgroundProcesses()
{
    int i=0;
    while(true)
    {
        if(shouldCancel.IsCancellationRequested) return;
        Console.WriteLine(i++);
        if(i==7) shouldCancel.Cancel(); ← | Использование флага
                                            | для остановки самого себя
    }
}
```

Этот метод, похожий на `BackgroundProc` из предыдущего примера, имеет дополнительное условие выхода, и разработчик заметил, что в программе уже есть готовый способ остановить поток (наш флаг отмены), и воспользовался им. Этот способ правильно сработает для этого потока, но он также остановит другой фоновый поток, просто потому что оба потока проверяют один и тот же флаг, а это, вероятно, не то, что нам нужно. Мы можем исправить этот недостаток, разделив `CancelFlag` на два класса: один для установки флага отмены, а другой для его проверки. В результате получаем API, который выглядит как:

```
class CancelFlag
{
    public bool IsCancellationRequested {get;}
}
class CancelFlagSource
{
    public void Cancel();
    public CancelFlag Flag {get;}
}
```

Мы разделили интерфейс на два класса: `CancelFlagSource` создает и управляет флагом `CancelFlag`, а `CancelFlag` используется только для проверки флага запроса остановки потока. Код, который может отменить операцию, использует `CancelFlagSource`, а код, который можно отменить, получает только `CancelFlag`. Изменив программу и использовав в ней новый интерфейс флага отмены, получим следующее (листинг 9.5).

В этом примере отсутствует одна важная деталь: мы не реализовали классы `CancelFlagSource` и `CancelFlag`. Но в этом нет ничего страшного, потому что

Microsoft позаботилась об этом и реализовала классы `CancellationToken` и `CancellationTokenSource`, которые делают все то, о чем говорилось выше, и даже больше. Вот как могла бы выглядеть наша программа, если в ней использовать класс `CancellationToken` (листинг 9.6).

Листинг 9.5. Использование CancelFlag и CancelFlagSource

```
var thread = new Thread(BackgroundProc);
var cancelFlagSource = new CancellationTokenSource(); ← Создает флаг отмены
var shouldCancel = cancelFlagSource.Flag; ← Получает флаг
thread.Start();
Console.ReadKey();
cancelFlagSource.Cancel(); ← Устанавливает флаг

void BackgroundProc()
{
    int i=0;
    while(true)
    {
        if(shouldCancel.IsCancellationRequested) return; ← Проверяет флаг
        Console.WriteLine(i++);
    }
}
```

Листинг 9.6. Использование CancellationToken

```
var thread = new Thread(BackgroundProc);
var cancelTokenSource = new CancellationTokenSource(); ← Создает CancellationTokenSource
var shouldCancel = cancelTokenSource.Token; ← Получает токен
thread.Start();
Console.ReadKey();
cancelTokenSource.Cancel(); ← Отменяет токен

void BackgroundProc()
{
    int i=0;
    while(true)
    {
        if(shouldCancel.IsCancellationRequested) return; ← Проверяет токен
        Console.WriteLine(i++);
    }
}
```

Это тот же самый код, что и в листинге 9.5. Я лишь заменил `CancelFlag` на `CancellationToken`.

Важно помнить, что по своей сути `CancellationToken` — это просто переменная типа `bool` (обернутая в потокобезопасный и устойчивый к злоупотреблениям класс); в ней нет ничего магического, и она сама не знает, что и как будет отменяться. Если бы наша предыдущая программа делала в цикле что-то более

существенное и продолжительное, чем простой вызов `Console.WriteLine` (например, выполняла бы вычисления, занимающие одну полную минуту), то остановка потока была бы отложена до тех пор, пока не завершится это длительное вычисление (листинг 9.7).

Листинг 9.7. Отложенная остановка потока, выполняющего продолжительные вычисления

```
var thread = new Thread(BackgroundProc);
var cancelTokenSource = new CancellationTokenSource();
var shouldCancel = cancelTokenSource.Token;
thread.Start();
Console.ReadKey();
cancelTokenSource.Cancel();

void BackgroundProc()
{
    int i=0;
    while(true)
    {
        if(shouldCancel.IsCancellationRequested) return;
        ACalculationThatTakesOneMinute();
        Console.WriteLine(i++);
    }
}

void ACalculationThatTakesOneMinute()
{
    var result = 0;
    var start = DateTime.UtcNow;
    while((DateTime.UtcNow - start).TotalMinutes < 1) ←
    {
        result++; ← Сами вычисления
    }
}
```

Вычисления производятся
одну полную минуту

В этом примере основной цикл в фоновом потоке проверяет признак отмены и вызывает долго выполняющийся метод. Это означает, что поток сможет остановиться, только когда этот метод вернет управление и выполнится следующая проверка признака отмены, а поскольку время между проверками в этом примере составляет 1 мин, то с момента установки признака отмены до фактической остановки фонового потока пройдет от 0 до 1 мин (или в среднем 30 с).

При выполнении продолжительных операций внутри цикла вам придется либо смириться с тем, что остановка может занять некоторое время, либо изменить долго выполняющийся код и включить в него периодическую проверку `CancellationToken`. В частности предыдущий пример можно изменить, как показано в листинге 9.8, и проверять признак отмены внутри `ACalculationThatTakesOneMinute`.

Листинг 9.8. Проверка CancellationToken в процессе выполнения длительной операции

```

var thread = new Thread(BackgroundProc);
var cancelTokenSource = new CancellationTokenSource();
var shouldCancel = cancelTokenSource.Token;
thread.Start();
Console.ReadKey();
cancelTokenSource.Cancel();

void BackgroundProc()
{
    int i=0;
    while(true)
    {
        if(!ACalculationThatTakesOneMinute(cancelTokenSource.Token))
            return;
        Console.WriteLine(i++);
    }
}

bool ACalculationThatTakesOneMinute(CancellationToken shouldCancel)
{
    var start = DateTime.UtcNow;
    var result = 0;
    while((DateTime.UtcNow - start).TotalMinutes < 1)
    {
        if(shouldCancel.IsCancellationRequested) ← Проверка признака отмены
            return false;                           во внутреннем цикле
        result++;
    }
    return true;
}

```

В этом примере я переместил проверку признака отмены в метод `ACalculationThatTakesOneMinute` и изменил тип его возвращаемого значения на `bool`: `true` означает, что метод успешно завершился, а `false` означает, что он был отменен. Это необходимо, потому что в реальных программах в большинстве случаев полезно знать, завершилось ли вычисление штатно и можно ли использовать результат.

9.2. Отмена с использованием исключения

В предыдущих примерах мы переместили проверку отмены в метод `ACalculationThatTakesOneMinute`. Это означает, что вызов метода изменился с простого и понятного

```

ACalculationThatTakesOneMinute();
на более замысловатый
if(!ACalculationThatTakesOneMinute()) return;

```

Оператор `if` не только загромождает наш код, но и создает риск обслуживания, поскольку кто-то в будущем может изменить код и забыть добавить `if`. Код также использует значение, возвращаемое методом, поэтому если метод должен возвращать значение, то мы должны использовать кортежи или `out`-параметры.

Все эти проблемы можно решить с помощью исключения, заменив проверку отмены, в случае которой возвращается `false`:

```
if(shouldCancel.IsCancellationRequested)
    return false;
```

очень похожим кодом, выбрасывающим исключение:

```
if(shouldCancel.IsCancellationRequested)
    throw new OperationCanceledException();
```

Этот прием настолько распространен, что в Microsoft добавили метод, который делает именно это, и проверка отмены превращается в простой вызов

```
shouldCancel.ThrowIfCancellationRequested();
```

Во всех предшествующих примерах фоновые операции выполняли некоторые вычисления. Бывает, что такие операции нужно отменить, и мы могли встроить проверку отмены внутрь этих вычислений. Но что, если нужно отменить операцию, в которую невозможно вставить проверку отмены?

9.3. Получение обратного вызова, когда вызывающий код отменяет операцию

Представьте, что мы используем библиотеку, в которой действует собственная система отмены, не основанная на `CancellationToken`. Эта система может иметь, например, такой интерфейс:

```
class MyCalculation
{
    void Start();
    void Cancel();
    event Action Complete;
}
```

При нормальной работе мы вызываем `Start` и ждем события `Complete`. А если понадобится отменить текущую операцию, то вызываем метод `Cancel`. Подобные интерфейсы можно найти в коде, который вызывает удаленные серверы, использует сторонние библиотеки, отличные от .NET, или в сторонних библиотеках, написанных кем-то, кому по какой-то причине не нравится `CancellationToken`.

В таких случаях можно добавить еще один фоновый поток, который просто периодически проверяет статус `CancellationToken` и вызывает `MyCalculation.Cancel`, когда `IsCancellationRequested` получает значение `true`, но это не самое эффективное решение. По этой причине в `CancellationToken` была добавлена возможность использования метода обратного вызова в случае отмены. Использование этого решения упрощает пример класса `MyCalculation`:

```
void RunMyCalculation(CancellationToken cancel)
{
    var calc = new MyCalculation();
    cancel.Register(()=>calc.Cancel()); ← Регистрирует
    calc.Complete += CalcComplete();      обратный вызов
    calc.Start();
}
```

Метод `CancellationToken.Register` регистрирует метод обратного вызова, который `CancellationToken` должен запустить при отмене. Многократный вызов `Register` приведет к запуску всех зарегистрированных методов обратного вызова при отмене `CancellationToken`. Вызов `Register`, когда `CancellationToken` уже отменен, немедленно выполнит метод обратного вызова. `Register` возвращает объект, который можно использовать для отмены регистрации обратного вызова.

Обратите внимание, что метод обратного вызова, который передается в `Register`, будет выполнен в потоке,зывающем `Cancel`, а не в фоновом потоке, выполнение которого прерывается. Поэтому убедитесь, что все операции в обратном вызове потокобезопасные, и избегайте действий, которые могут помешатьзывающему потоку.

9.4. Реализация тайм-аутов

Очень распространенным сценарием отмены являются тайм-ауты, когда операция отменяется, если не успела завершиться в течение определенного времени. Например, если при попытке отправить сообщение по сети мы не получили ответа, то не можем с уверенностью сказать, в чем причина, — мы не получили ответа, потому что он еще не дошел до нас или потому что компьютер, которому отправили сообщение, не подключен к сети. Поэтому мы ждем определенное время, и если в течение заданного периода ожидания ответ не был получен, то предполагаем, что ответ никогда не придет, и отменяем операцию.

Мы легко можем написать код, запускающий таймер и вызывающий `CancellationTokenSource.Cancel`, когда таймер сработает, но поскольку это очень распространенный сценарий, в `CancellationTokenSource` уже имеется эта функция, реализованная в виде метода `CancelAfter`. Метод `CancelAfter` имеет две

перегруженные версии, одна из которых принимает количество миллисекунд для ожидания

```
var cancelSource = new CancellationTokenSource();
cancelSource.CancelAfter(30000);
```

а другая, более удобная и современная, принимает `TimeSpan`:

```
var cancelSource = new CancellationTokenSource();
cancelSource.CancelAfter(TimeSpan.FromSeconds(30));
```

Оба этих фрагмента кода создают `CancellationToken` (доступный как `cancelSource.Token`), который автоматически отменяется через 30 с.

Вызов `CancelAfter`, когда `CancellationToken` уже отменен, ничего не делает. Повторный вызов `CancelAfter` до отмены `CancellationToken` перезапустит таймер. Вызов `CancelAfter(-1)` до отмены `CancellationToken` отменит тайм-аут.

9.5. Комбинирование методов отмены

Иногда бывает нужно отменить операцию по двум совершенно разным причинам. Например, представьте, что у вас есть код, для которого нужно предусмотреть возможность отмены как пользователем, так и по тайм-ауту. Для демонстрации напишем код, который выполняет HTTP-запрос `GET` к серверу и возвращает результат в виде строки (листинг 9.9).

Листинг 9.9. HTTP-вызов, который может быть отменен пользователем

```
public async Task<string>
    GetTextFromServer(CancellationToken canceledByUser)
{
    using(var http = new HttpClient())
    {
        return await http.GetStringAsync("http://example.com",
            canceledByUser);
    }
}
```

Этот метод принимает экземпляр `CancellationToken` с именем `canceledByUser`, который явно указывает, что операция отменена пользователем. Теперь мы должны добавить тайм-аут, но не можем, потому что нам нужен `CancellationTokenSource`, а у нас есть только `CancellationToken`.

К счастью для нас, существует статический метод `CancellationTokenSource.CreateLinkedTokenSource`, способный создать `CancellationTokenSource` из одного или нескольких объектов `CancellationToken`. Вызвав его, мы можем

использовать новый `CancellationTokenSource` для создания `CancellationToken` и добавить в него тайм-аут (листинг 9.10).

Листинг 9.10. HTTP-вызов, который может быть отменен пользователем или по тайм-ауту

```
public async Task<string>
    GetTextFromServer(CancellationToken canceledByUser)
{
    var combined = CancellationTokenSource.CreateLinkedTokenSource(
        canceledByUser);
    combined.CancelAfter(TimeSpan.FromSeconds(10)); ← Добавляет тайм-аут
    using(var http = new HttpClient())
    {
        return await http.GetStringAsync("http://example.com",
            combined.Token); ← Использует
    }                                новый токен
}
```

В `CreateLinkedTokenSource` можно передать любое количество объектов `CancellationToken`, и токен, контролируемый новым `CancellationTokenSource`, будет автоматически отменен, если произойдет отмена любого из них. Используя новый `CancellationTokenSource`, можно добавить тайм-аут или вручную отменить его токен. Все, что будет сделано с новым `CancellationTokenSource`, не повлияет на токены, использованные для его создания.

9.6. Специальные токены отмены

На протяжении всей этой главы рассказывалось, как использовать `CancellationToken` для отмены операции. Однако иногда, несмотря на то что возможность отмены операции не нужна, используемый API все равно требует `CancellationToken`. В таких случаях можете просто передать `CancellationToken.None`. В результате вы получите `CancellationToken`, который нельзя отменить. Создание `CancellationToken` с помощью `new CancellationToken(false)` даст тот же результат, но такой код выглядит более запутанным.

Напротив, `new CancellationToken(true)` создаст уже отмененный `CancellationToken`. В обычном коде в этом нет особого смысла, но такой прием может пригодиться в модульных тестах.

Итоги главы

- `CancellationToken` — стандартный способ отмены операций в .NET.
- Класс `CancellationTokenSource` используется для создания и управления объектами `CancellationToken`.

- `CancellationTokenSource.Cancel` используется для отмены операции, а `CancellationToken.IsCancellationRequested` — для проверки отмены операции.
- `CancellationToken` — это просто флаг. Сам по себе он ничего не отменяет.
- Для вызова метода обратного вызова при отмене можно использовать `CancellationToken.Register`.
- Для реализации тайм-аутов можно использовать `CancellationTokenSource.CancelAfter`.
- `CancellationTokenSource.CreateLinkedTokenSource` позволяет создать `CancellationTokenSource` из одного или нескольких существующих объектов `CancellationToken` и управлять им.
- Если нужно передать `CancellationToken`, который никогда не отменится, используйте `CancellationToken.None`.

10

Ожидаем собственные события

В этой главе

- ✓ Создание объектов Task, которые вы можете контролировать.
- ✓ TaskCompletionSource и TaskCompletionSource<T>.
- ✓ Завершение Task с признаком успеха или ошибки, а также отмена Task.
- ✓ Адаптация старых и нестандартных асинхронных API для использования с типом Task.
- ✓ Использование TaskCompletionSource для асинхронной инициализации.
- ✓ Использование TaskCompletionSource для реализации асинхронных структур данных.

До сих пор мы говорили об использовании `async/await` при работе с асинхронными API. В этой главе мы поговорим о написании собственных асинхронных API. Обычно это бывает необходимо, чтобы адаптировать асинхронный API, основанный на механизме, отличном от `Task`, к использованию с оператором `await`, а также чтобы получить возможность использовать `await` для асинхронного ожидания событий, которые происходят в приложении, и для создания потокобезопасной структуры данных, совместимой с `async/await`, — это

лишь несколько примеров. (Спойлер: мы напишем код для этих примеров в этой главе.)

Еще в главе 3, чтобы понять, как работают ключевые слова `async` и `await`, мы написали метод, использующий `async/await`, и преобразовали его в эквивалентный метод, выполняющий точно такую же асинхронную операцию, но без использования `async` и `await`. Тогда мы не умели создавать объекты `Task`, но знали, что `await` можно реализовать с помощью обратного вызова (в частности, `Task.ContinueWith`). Поэтому вместо `Task` мы использовали обратные вызовы для возврата результатов операции. С этой целью мы изменили сигнатуру метода с

```
Task<int> GetBitmapWidth(string path)
```

на

```
void GetBitmapWidth(string path,
    Action<int> setResult,
    Action<Exception> setException)
```

Метод `ContinueWith`, встроенный в .NET механизм обратного вызова, использует один метод обратного вызова, который должен проверять, как завершилась асинхронная операция — успешно или с ошибкой. Однако мы решили использовать для каждого из этих случаев отдельные методы обратного вызова `setResult` и `setException`, потому что это упрощает код. Как побочный результат, успешно смоделировав `Task` с помощью этих двух вызовов, мы показали, что вызывать `setResult` и `setException` (при наличии возможности связать их с `Task`) более чем достаточно для управления задачей. Сюрприз! В библиотеке .NET есть класс с именем `TaskCompletionSource<T>`. Он может создавать объекты `Task<T>` и имеет методы с именами `SetResult` и `SetException`. Давайте посмотрим, как вы можете использовать его.

10.1. Знакомство с `TaskCompletionSource`

В библиотеке .NET есть класс `TaskCompletionSource`, предназначенный для создания и управления объектами `Task`, и класс `TaskCompletionSource<T>` для создания и управления объектами `Task<T>`. `TaskCompletionSource` и `TaskCompletionSource<T>` очень похожи, за исключением того, что методы `SetResult` и `TrySetResult` в `TaskCompletionSource` (без `<T>`) не принимают никаких параметров и просто завершают `Task`, не устанавливая результат (потому что, в отличие от `Task<T>`, `Task` не имеет свойства `Result`). В оставшейся части этой главы я буду говорить о `TaskCompletionSource`, не различая `TaskCompletionSource` и `TaskCompletionSource<T>`, но все сказанное будет в равной степени относиться к обоим классам.

`TaskCompletionSource` имеет свойство с именем `Task`, которое позволяет получить созданный им объект `Task`. Каждый экземпляр `TaskCompletionSource` управляет

одним объектом `Task`, и многократное чтение свойства `Task` будет возвращать один и тот же объект `Task`.

Изначально `Task` находится в состоянии `WaitingForActivation`, а свойства `IsCompleted`, `IsCompletedSuccessfully`, `IsCanceled` и `IsFaulted` имеют значение `false`. Применение инструкции `await` к новому объекту `Task` приведет к асинхронному ожиданию, а вызов `Wait` или чтение свойства `Result` заблокируется до завершения задачи с помощью `TaskCompletionSource`.

Чтобы продемонстрировать различные способы завершения задачи `Task`, воспользуемся следующим примером (листинг 10.1).

Листинг 10.1. Пример кода для демонстрации `TaskCompletionSource`

```
public class TaskCompletionSourceDemo
{
    private Task<int> BackgroundWork() ← Нет ключевого слова async
    {
        var tcs = new TaskCompletionSource<int>();
        Task.Run(()=>
        {
            ← Задача должна
            ← завершиться здесь
        });
        return tcs.Task; ← Возвращает
                           Task<int>, а не int
    }

    public async Task RunDemo()
    {
        var result = await BackgroundWork(); ← Ждет завершения задачи
        Console.WriteLine(result);
    }
}
```

Обратите внимание, что метод `BackgroundWork` не отмечен ключевым словом `async`. По этой причине компилятор не будет преобразовывать этот метод, мы не сможем использовать `await` внутри него, и компилятор не будет заключать результат в `Task`. Все это означает, что мы сами должны создать и вернуть `Task<int>`. Метод `RunDemo` (отмеченный как асинхронный) просто использует `await` для получения результата, созданного методом `BackgroundWork`.

`TaskCompletionSource` имеет три группы методов, которые можно использовать для завершения задачи.

1. `SetResult` и `TrySetResult` завершат задачу `Task`, изменят ее состояние на `RanToCompletion`, а в случае `Task<T>` сохранят результат в объекте `Task<T>` (который можно будет получить с помощью `Task.Result` или `await`) (листинг 10.2). После вызова `SetResult` или `TrySetResult` свойства `IsCompleted` и `IsCompletedSuccessfully` получат значение `true`.

Этот пример показывает, как завершить задачу `Task` с признаком успеха. Вызов `TrySetResult` (или `SetResult`) заставляет `await` продолжить работу.

Листинг 10.2. Демонстрация TaskCompletionSource.TrySetResult

```
public class TaskCompletionSourceDemo
{
    private Task<int> BackgroundWork()
    {
        var tcs = new TaskCompletionSource<int>();
        Task.Run(()=>
        {
            tcs.TrySetResult(7); ← Завершает задачу Task
            // с признаком успеха
        });
        return tcs.Task;
    }

    public async Task RunDemo()
    {
        var result = await BackgroundWork(); ← Выполнение
        Console.WriteLine(result); ← продолжится здесь
        // Выведет 7
    }
}
```

2. `SetException` и `TrySetException` завершат задачу с состоянием `Faulted` и сохранят исключение или список исключений в объекте `Task`. Исключение или список исключений будут заключены в объект `AggregateException` и сохранены в свойстве `Task.Exception` (листинг 10.3). Применение `await` к `Task`, чтение свойства `Result` или вызов `Wait()` выбросят `AggregateException`. После вызова `SetException` или `TrySetException` свойства `IsCompleted` и `IsFaulted` получат значение `true`.

В этом примере мы использовали `TrySetException` для завершения задачи `Task` и перевода ее в состояние сбоя. Оператор `await` в этом случае выбросит исключение.

Листинг 10.3. Демонстрация TaskCompletionSource.TrySetException

```
public class TaskCompletionSourceDemo
{
    private Task<int> BackgroundWork()
    {
        var tcs = new TaskCompletionSource<int>();
        Task.Run(()=>
        {
            tcs.TrySetException(new Exception("oops")); ← Завершает задачу Task
            // ошибкой
        });
        return tcs.Task;
    }

    public async Task RunDemo()
    {
        var result = await BackgroundWork(); ← Выбросит исключение
        Console.WriteLine(result);
    }
}
```

3. `SetCanceled` и `TrySetCanceled` завершат задачу `Task`, изменят ее состояние на `Canceled` и при необходимости сохранят токен отмены в объекте `Task` (листинг 10.4). Применение инструкции `await` к `Task`, чтение свойства `Result` или вызов `Wait()` выбросят исключение `TaskCanceledException`. Если передать токен отмены в `TrySetCanceled`, то он будет доступен в свойстве `TaskCanceledException.CancellationToken`. После вызова `SetCanceled` или `TrySetCanceled` свойства `IsCompleted` и `IsCanceled` получат значение `true`. Обратите внимание: несмотря на то что `await` выбросит исключение, свойство `IsFaulted` объекта `Task` будет иметь значение `false`, а свойство `Exception` — значение `null`.

Листинг 10.4. Демонстрация `TaskCompletionSource.TrySetCanceled`

```
public class TaskCompletionSourceDemo
{
    private Task<int> BackgroundWork()
    {
        var tcs = new TaskCompletionSource<int>();
        Task.Run(()=>
        {
            tcs.TrySetCanceled();           ← Завершает задачу Task,
            return tcs.Task;              ← отменяя ее
        });
    }

    public async Task RunDemo()
    {
        var result = await BackgroundWork(); ← Выбросит исключение
        Console.WriteLine(result);
    }
}
```

В этом примере мы использовали `TrySetCanceled` для отмены `Task`, а `await` выбросит исключение `TaskCanceledException`. `TaskCompletionSource` не позволяет установить статус `Task` в любое другое значение (`WaitingToRun`, `Running` или `WaitingForChildrenToComplete`).

Разница между двумя вариантами каждого метода заключается в том, что старый `SetXXX` выбрасывает исключение, если задача `Task` уже завершилась, а новый `TrySetXXX` — нет (если задача `Task` уже завершилась, то `Task` не изменится, а все параметры, переданные методу, будут проигнорированы). Причина добавления варианта `TrySetXXX` состоит в том, что старые методы могут создать состояние гонки в любой ситуации, когда производится попытка завершить задачу из двух разных потоков (например, когда один из них выполняет работу, а другой обрабатывает отмену). Предпочтительнее использовать новые версии `Try` всех методов, если только вы намеренно не полагаетесь

на исключения, которые выбрасываются, если задача Task уже завершилась. Вариант Try вернет `true`, если он завершит задачу Task, и `false`, если задача Task уже завершилась.

Например, в следующем фрагменте, моделирующем ситуацию, когда другой поток отменяет Task прямо перед завершением вычислений, вызов `SetResult` выбросит исключение:

```
var tcs = new TaskCompletionSource<int>();
tcs.SetCanceled(); ← Отменяет задачу
tcs.SetResult(7); ← Выбрасывает
                    исключение
```

В следующем фрагменте вызов `TrySetResult` будет проигнорирован и исключение не будет выброшено (но вы все равно сможете узнать, что `TrySetResult` не отработал, потому что он вернет `false`):

```
var tcs = new TaskCompletionSource<int>();
tcs.TrySetCanceled(); ← Отменяет задачу
tcs.TrySetResult(7); ← Игнорируется,
                    вернет false
```

10.2. Выбор потока для запуска продолжения

Код, выполняемый после асинхронной операции (после `await` или обратного вызова, переданного в `ContinueWith`), называется продолжением (continuation). Вызов любого метода `TaskCompletionSource`, завершающего Task, приведет к запуску продолжения (очевидно, в этом и есть весь смысл), и `TaskCompletionSource` позволяет решить — запустить ли продолжение немедленно в том же потоке, откуда был вызван метод класса `TaskCompletionSource`.

Если позволить продолжению запуститься немедленно, то оно может запуститься до возврата из `TrySetResult` (или любого другого метода). Это означает, что выполнение `TrySetResult` может занять произвольно много времени и что наш код находится в состоянии, в котором может выполняться произвольный код, безопасность которого неподконтрольна нам. Например, следующий код потенциально может попасть в состояние взаимоблокировки:

```
lock(_valueLock)
{
    _taskSource.TrySetResult(_value);
}
```

Здесь предполагается, что используемое значение (результат задачи) защищено блокировкой, поэтому мы получаем блокировку и вызываем `TrySetResult` со значением. Это может привести к тому, что код, неподконтрольный нам

(продолжение Task), будет запущен, пока мы удерживаем блокировку, и если этот код будет ждать чего-то еще и использовать ту же блокировку в другом потоке, то возникнет состояние взаимоблокировки.

Одно из решений этой проблемы — перемещение вызова TrySetResult за пределы блока lock:

```
int copyOfValue;
lock(_valueLock)
{
    copyOfValue = _value;
}
_taskSource.TrySetResult(copyOfValue);
```

Мы не можем использовать переменную `_value` вне блока `lock`, но можем скопировать ее в локальную переменную и передать копию TrySetResult вне блокировки. В этом случае неподконтрольный нам код по-прежнему может запуститься до возврата из TrySetResult, поэтому мы не можем знать, сколько времени займет TrySetResult, зато исчез риск взаимоблокировки.

Другой вариант — заставить `TaskCompletionSource` запустить код в другом потоке. Это можно сделать с помощью конструктора `TaskCompletionSource`, который принимает параметр `TaskCreationOptions`, и передать ему значение `TaskCreationOptions.RunContinuationsAsynchronously`:

```
_taskSource = new TaskCompletionSource<int>(
    TaskCreationOptions.RunContinuationsAsynchronously);
```

Нам нужно решить, должен ли `TaskCompletionSource` запускать код продолжения в другом потоке, во время создания `TaskCompletionSource`. Мы не всегда можем с помощью TrySetResult запустить продолжение в фоновом потоке. Например, мы не можем заставить `TaskCompletionSource` использовать другой поток, если в этот момент удерживается блокировка.

В этом примере я использовал TrySetResult, но все сказанное здесь относится и к другим методам, завершающим задачу (`SetResult`, `SetException`, `TrySetException`, `SetCanceled` и `TrySetCanceled`).

10.3. Пример: ожидание инициализации

Начнем с простого примера: напишем класс, инициализация которого требует много времени, из-за чего инициализация выполняется в фоновом режиме. При вызове любого метода этого класса до завершения инициализации он будет ждать, пока инициализация не завершится (листинг 10.5).

В этом примере определен класс `RequiresInit`, инициализация которого занимает много времени, из-за чего нежелательно (или невозможно) выполнять

процесс инициализации синхронно в конструкторе. Поэтому конструктор просто запускает процесс инициализации в фоновом потоке с помощью `Task.Run` и немедленно возвращает управление. Чтобы получить доступ к результату процесса инициализации, с помощью `TaskCompletionSource<int>` создается экземпляр `Task<int>` и связывается с полем `value`.

Листинг 10.5. Класс с фоновой инициализацией

```
public class RequiresInit
{
    private Task<int> _value;

    public RequiresInit ()
    {
        var tcs = new TaskCompletionSource<int>();
        _value = tcs.Task; ← Присваивает задачу перед
        Task.Run(()=>           выходом из конструктора
        {
            try
            {
                Thread.Sleep(1000); ← Имитирует продолжительный
                tcs.TrySetResult(7); ← процесс инициализации
                Устанавливает
                результат задачи
            }
            catch(Exception ex)
            {
                tcs.TrySetException(ex);
            }
        });
    }

    public async Task<int> Add1()
    {
        var actualValue = await _value; ← Ждет результата,
        return actualValue+1;           если необходимо
    }
}
```

Очевидно, что в подобных случаях результатом продолжительной инициализации, скорее всего, является сложный объект, но для простоты в этом примере процесс инициализации просто вызывает `Thread.Sleep`, а результатом всегда является число 7.

В фоновом потоке после вычисления результата вызывается `TrySetResult`, завершающий задачу и присваивающий результат вычислений. На случай, если в ходе вычислений возникнет исключение, здесь используется метод `TrySetException`, который передаст исключение в задачу.

Позже, когда потребуется использовать результат инициализации, его можно прочитать с помощью `await _value`, и если к этому моменту вычисления уже завершились, то эта инструкция немедленно вернет значение, в противном

случае она асинхронно будет ждать, когда результат станет доступен. Наконец, если в ходе вычислений возникнет ошибка, то будет выброшено исключение, сообщающее причину.

Использование такой задачи сводит получение результата, обработку сигнала о доступности результата и сообщение об ошибках инициализации (если такие имеются) в одну операцию. Это не только избавляет от необходимости писать лишний код, но и делает код более удобным для поддержки, поскольку будущие разработчики не смогут забыть дождаться, когда значение станет доступно, или проверить наличие ошибок.

10.4. Пример: адаптация старых API

Вероятно, наиболее простым применением `TaskCompletionSource` является адаптация асинхронных API, несовместимых с `await`. К счастью, это случается все реже и реже, потому что почти все асинхронные операции в библиотеке .NET и популярные сторонние компоненты были адаптированы для использования объектов `Task` и уже совместимы с `await`. В настоящее время библиотеки, не поддерживающие `async/await`, в основном являются обертками для стороннего кода, не являющегося кодом .NET, либо написаны авторами, которые почему-то недолюбливают `async/await`.

Для демонстрации воспользуемся шаблоном, который был довольно распространён до появления `async/await`, — это интерфейс, позволяющий запустить операцию и получить уведомление о ее завершении:

```
public interface IAsyncOperation
{
    void StartCalculation();
    event Action<int> CalculationComplete;
}
```

Адаптировать его с помощью `TaskCompletionSource` довольно просто, как показано в листинге 10.6.

Листинг 10.6. Адаптация нестандартных асинхронных API

```
public Task<int> CallAsyncOperation()
{
    var tcs = new TaskCompletionSource<int>();
    _asyncOperation.CalculationComplete +=
        result => tcs.TrySetResult(result);
    _asyncOperation.StartCalculation();
    return tcs.Task;
}
```

Здесь создается объект `TaskCompletionSource`, который подписывается на нестандартное уведомление о завершении асинхронной операции и вызывает `TrySetResult`, когда асинхронная операция завершается.

10.5. Асинхронные операции в старом стиле (BeginXXX, EndXXX)

Для реализации асинхронных операций в .NET до появления задач `Task` и `async/await` использовалась пара методов: один с префиксом `Begin`, который возвращает объект `IAsyncResult`, и второй с префиксом `End`. Все эти методы в библиотеке .NET и большинстве сторонних библиотек уже имеют альтернативу на основе задач `Task`, поэтому они встречаются довольно редко. Я говорю об этом только для того, чтобы вы знали, что представляют собой все эти методы `BeginXXX` и `EndXXX` и что делать, если вам встретится старая библиотека, не адаптированная для использования класса `Task`.

До появления `async/await` писать асинхронный код было сложно. Асинхронные методы использовались редко и только в случаях, когда это действительно было необходимо, поэтому асинхронная версия API была основана на синхронной версии. Асинхронные версии всегда состояли из двух методов.

- Метод с префиксом `Begin` принимает те же параметры, что и синхронная версия, и два дополнительных параметра (`callback` и `state`). Этот метод запускает асинхронную операцию. Объект `IAsyncResult`, возвращаемый методом `Begin`, представляет асинхронную операцию и, как `Task` в новых API, может использоваться для определения момента завершения операции.
- Метод с префиксом `End` принимает объект `IAsyncResult`, освобождает все ресурсы, используемые асинхронной операцией, и возвращает результат операции.

Для демонстрации адаптации возьмем старую асинхронную операцию и адаптируем ее к новому стилю на основе `Task`. Для этого примера мы будем использовать метод `Stream.Read`:

```
int Read (byte[] buffer, int offset, int count);
```

А асинхронная версия старого стиля состоит из двух методов — `Stream.BeginRead` и `Stream.EndRead`:

```
IAsyncResult BeginRead(
    byte[] buffer, int offset, int count,
    AsyncCallback? callback, object? state);
int EndRead (IAsyncResult asyncResult);
```

Нетрудно догадаться, что параметр `callback` и `TaskCompletionSource` можно использовать так же, как в предыдущем примере, но есть более простой способ. Библиотека .NET содержит метод `Task.Factory.FromAsync`, который создает `Task` из этой пары методов. Вот как он используется:

```
public Task<int> MyReadAsync(
    Stream stream, byte[] buffer, int offset, int length)
{
    return Task.Factory.FromAsync(
        (callback, state)=>stream.BeginRead(
            buffer,0,buffer.Length,callback,state), stream.EndRead, null);
}
```

Метод `Task.Factory.FromAsync` принимает три параметра:

- *лямбда-функция, вызывающая метод BeginXXX* — если методу `BeginXXX` не нужны никакие параметры, кроме `callback` и `state`, то его можно передать, не заключая в лямбда-функцию;
- *метод EndXXX* — если этот метод имеет выходные (`out`) параметры, то его необходимо заключить в лямбда-функцию и извлекать значения этих параметров;
- *параметр state* — он не требуется, если используется лямбда-функция, и всегда может иметь значение `null`.

Поскольку большинство API (включая `Stream.Read`) уже имеют версию на основе `Task`, а более новые API имеют только асинхронные версии на основе `Task` без методов `BeginXXX` и `EndXXX`, необходимость использовать этот метод встречается довольно редко. Поэтому мы не будем вдаваться в подробности.

10.6. Пример: асинхронные структуры данных

В этом примере мы напишем асинхронную очередь, которая, как и обычная, представляет собой коллекцию FIFO (first-in, first-out — первым пришел, первым вышел) с двумя операциями: постановка в очередь (`enqueue`) и извлечение из очереди (`dequeue`). Операция постановки в очередь добавляет элемент в очередь, а операция извлечения из очереди извлекает первый элемент из очереди, если она не пуста, и возвращает его. Особенность нашей очереди `AsyncQueue` состоит в том, что если в очереди нет элементов, то операция извлечения из очереди вернет задачу `Task`, которая завершится, когда позже в очередь будет добавлен новый элемент. Таким образом, `await asyncQueue.Dequeue()` немедленно вернет следующее значение, если оно уже есть в очереди, или, если очередь пуста, то асинхронно дождется, когда следующее значение станет доступно.

Наш класс очереди фактически состоит из двух очередей: одна хранит данные, ожидающие обработки, а другая хранит обработчики, ожидающие данные (листинг 10.7). По крайней мере одна из этих очередей должна быть пустой все время, потому что иначе мы упускаем возможность сопоставить элемент данных с обработчиком.

Листинг 10.7. Асинхронная очередь

```
public class AsyncQueue<T>
{
    private Queue<TaskCompletionSource<T>>
        _processorsWaitingForData = new();
    private Queue<T> _dataWaitingForProcessors = new();
    private object _lock = new object();
```

Когда обработчик становится доступным, он вызывает `Dequeue`. Если в очереди есть элемент данных, ожидающий обработки, то он немедленно передается обработчику через завершенную задачу `Task`, созданную с помощью `Task.FromResult`. Если данные недоступны, то создается новый экземпляр `TaskCompletionSource` и возвращается его свойство `Task`, а затем этот `TaskCompletionSource` ставится в очередь `_processorsWaitingForData`:

```
public Task<T> Dequeue(CancellationToken cancellationToken)
{
    lock (_lock)
    {
        if (_dataWaitingForProcessors.Count > 0)
        {
            return Task.FromResult(_dataWaitingForProcessors.Dequeue());
        }
        var tcs = new TaskCompletionSource<T>(
            TaskCreationOptions.RunContinuationsAsynchronously);
        _processorsWaitingForData.Enqueue(tcs);
    }
}
```

Поскольку тот, кто использует наш класс, скорее всего, ожидает, что операции `Enqueue` и `Dequeue` будут выполняться быстро, мы создаем объекты `TaskCompletionSource` с флагом `TaskCreationOptions.RunContinuationsAsynchronously`. Это означает, что код, обрабатывающий данные, будет запущен в другом потоке, а не внутри методов `Enqueue` и `Dequeue`. Это также позволяет вызывать `TrySetResult` и `TryCancel`, удерживая блокировку.

Мы также позволяем обработчику передавать `CancellationToken`, потому что хотим дать обработчику возможность сообщать, что он недоступен. Если `CancellationToken` отменяется, то мы отменяем задачу `Task` обработчика, но оставляем `TaskCompletionSource` в очереди, потому что так проще.

В качестве оптимизации мы регистрируем токены отмены, только если их можно отменить. Примером `CancellationToken`, который нельзя отменить, является фиктивный токен, возвращаемый свойством `CancellationToken.None`:

```
if (cancellationToken.CanBeCanceled)
{
    cancellationToken.Register(() =>
    {
        tcs.TrySetCanceled(cancellationToken);
    });
}
return tcs.Task;
}
```

Когда данные добавляются в очередь вызовом `Enqueue`, мы пытаемся передать их первому доступному обработчику, извлечь первый `TaskCompletionSource` из очереди `_processorsWaitingForData` и вызвать `TrySetResult`. Если `TrySetResult` возвращает `true`, это значит, что мы успешно завершили `Task` и отправили элемент данных обработчику, поэтому можем выйти из метода.

Если `TrySetResult` возвращает `false`, это означает, что задача `Task` уже завершена, то есть была отменена, поскольку `TaskCompletionSource` полностью находится под нашим контролем, а код отмены — единственный, написанный нами, который завершает задачу без предварительного удаления ее `TaskCompletionSource` из очереди. В этом случае мы просто переходим к следующему обработчику.

В качестве оптимизации мы обрабатываем случай отмены, только если `CancellationToken` можно отменить (например, свойство `CancellationToken.None` возвращает фиктивный токен, который нельзя отменить):

```
public void Enqueue(T value)
{
    lock (_lock)
    {
        while (_processorsWaitingForData.Count > 0)
        {
            var nextDeququer = _processorsWaitingForData.Dequeue();
            if(nextDeququer.TrySetResult(value))
            {
                return;
            }
        }
    }
}
```

Если очередь обработчиков пустая или все элементы в очереди были отменены, то мы помещаем данные в очередь `_dataWaitingForProcessors`, где они будут ждать, пока кто-нибудь вызовет `Dequeue`:

```
    _dataWaitingForProcessors.Enqueue(value);
}
}
```

Итоги главы

- `TaskCompletionSource` можно использовать для создания объектов `Task`, а `TaskCompletionSource<T>` — для создания объектов `Task<T>`.
- `TaskCompletionSource<T>.TrySetResult` завершает `Task<T>` с признаком успеха и устанавливает свойство `Result`.
- `TaskCompletionSource<T>.TrySetResult` завершает `Task` с признаком успеха, но не устанавливает результат, потому что `Task` не имеет результата.
- `TaskCompletionSource<T>.TrySetException` и `TaskCompletionSource.TrySetException` завершают задачу `Task`, записывают в ее свойство `Status` значение `Faulted` и сохраняют одно или несколько исключений в `Task<T>` или `Task`.
- `TaskCompletionSource<T>.TrySetCanceled` и `TaskCompletionSource.TrySetCanceled` завершают задачу `Task` и записывают в ее свойство `Status` значение `Canceled`.
- При использовании `await`, вызове `Wait` или чтении свойства `Result` отмененной задачи `Task` выбрасывается исключение `TaskCanceledException`. Свойство `Exception` получит значение `null`. Проверить, была ли задача отменена, в этом случае можно с помощью свойства `Status` или `IsCanceled`.
- Все методы `TrySetXXX`, упомянутые ранее, вернут значение `true`, если задача `Task` завершилась, и `false`, если задача уже была завершена к моменту вызова метода.
- Есть также вариант `SetXXX`, который выбрасывает исключение, если задача `Task` уже завершена. Вариант `TrySetXXX` предпочтительнее, потому что старый `SetXXX` может вызвать состояние гонки в некоторых многопоточных сценариях.
- По умолчанию продолжения (код после `await` или в обратных вызовах `ContinueWith`) могут запускаться немедленно внутри вызова `TrySetXXX` или `SetXXX`, что делает небезопасным их вызов при удержании блокировки. Чтобы запустить продолжение в другом потоке (и сделать безопасным его вызов при удержании блокировки), можно передать флаг `TaskCreationOptions.RunContinuationsAsynchronously` конструктору `TaskCompletionSource`.
- Если потребуется использовать асинхронную операцию старого стиля (`BeginXXX`, `EndXXX`) с задачами, воспользуйтесь методом `Task.Factory.FromAsync`.

Выбор потока для выполнения асинхронного кода

В этой главе

- ✓ Поведение `await` в многопоточной среде.
- ✓ Введение в `SynchronizationContext`.
- ✓ Когда использовать `ConfigureAwait`.
- ✓ Применение `Task.Yield`.
- ✓ Основы `TaskScheduler`.

Чаще всего нам безразлично, в каком потоке выполнится наш код. Если мы что-то вычисляем, то вычисления вернут тот же самый результат, независимо от того, в каком потоке или на каком ядре процессора они выполняются. Но некоторые операции, примеры которых перечислены ниже, работают по-разному в разных потоках.

- *Графический интерфейс* – в WinForms и WPF все элементы пользовательского интерфейса доступны только потоку, который их создал. Обычно все элементы пользовательского интерфейса создаются и доступны только одному потоку (так называемому *потоку пользовательского интерфейса*), который, как правило, является главным потоком процесса.
- *ASP.NET Classic* – в ASP.NET Classic, более старой версии, используемой в .NET Framework 4.8 и ниже, свойство `HttpContext.Current` будет возвращать правильное значение только в случае вызова из правильного потока.

(Для тех, у кого нет опыта работы с ASP.NET Classic, отмечу, что во многих распространенных сценариях требуется доступ к `HttpContext.Current`.)

- *СОМ* — правила, затрагивающие потоки и компоненты СОМ, чрезвычайно сложны, и мы не будем рассматривать их в этой книге. Но доступ к компоненту СОМ из неправильного потока может привести к сбою или значительному снижению производительности, в зависимости от обстоятельств.
- *Блокирующие операции* могут заблокировать поток на потенциально долгое время. Блокировка разных потоков может иметь самые разные последствия. Например, блокировка потока пользовательского интерфейса приведет к зависанию пользовательского интерфейса, блокировка большого количества потоков в пуле потоков может помешать серверам принимать соединения и продолжать асинхронные операции и т. д.
- *Потенциально любой сторонний код* — любой сторонний код может накладывать свои ограничения на его использование. Современный код .NET, как правило, совместим с `async/await` и не зависит от того, в каком потоке запускается. Но старый код и низкоуровневый код могут устанавливать более строгие правила, касающиеся потоков.

В предыдущих главах мы говорили, что код после `await` и обратные вызовы, передаваемые в `ContinueWith`, являются взаимозаменяемыми. Однако все, что рассказывается в этой главе, применимо только к `await`. В главе 3 мы реализовали `await` с помощью `ContinueWith`, и я сказал, что код, сгенерированный компилятором, сложнее. Вот что я имею в виду: вся сложность в этой главе обусловлена кодом, сгенерированным компилятором для инструкции `await`.

11.1. Поведение `await` в многопоточной среде

Ниже перечислены основные правила выполнения кода, следующего за инструкцией `await`.

- В приложениях с графическим интерфейсом (WinForms и WPF), когда `await` применяется в потоке пользовательского интерфейса и не используется `ConfigureAwait` (мы поговорим об этом классе далее в этой главе), код после `await` будет выполняться в том же потоке.
- В ASP.NET Classic (не ASP.NET Core), когда `await` применяется в потоке, обрабатывающем веб-запрос, и не используется `ConfigureAwait`, код после `await` будет выполняться в том же потоке.
- Во всех остальных случаях код после `await` будет выполняться в одном из потоков в пуле.

Этот короткий, простой и легко запоминающийся список раскрывает основную мотивацию — поддержку приложений с графическим интерфейсом и старых

версий ASP.NET. Однако это поведение по умолчанию для всего в .NET (по крайней мере до версии 9) можно изменить. Далее в этой главе я покажу, как это поведение реализовано и как его можно изменить.

11.1.1. await в потоках пользовательского интерфейса

Приложения с графическим интерфейсом обычно читают значения, вводимые пользователем, что-то с ними делают и отображают результат. Например, следующие методы вызываются, когда пользователь нажимает кнопку. Они читают текст, введенный пользователем в текстовое поле, передают его асинхронному методу `DoSomething` и отображают результат в метке (`label1`) на экране (листинг 11.1).

Листинг 11.1. Ожидание в обработчике событий пользовательского интерфейса

```
private async void button1_Click(object sender, EventArgs ea)
{
    label1.Text = await DoSomething(textBox1.Text);
}
```

Этот метод вызывается фреймворком пользовательского интерфейса в потоке, создавшем кнопку. Почти во всех случаях это будет поток, создавший весь пользовательский интерфейс (и единственный, который может получить к нему доступ). Чтение `textBox1.Text` происходит до `await` и выполняется в потоке пользовательского интерфейса. Запись в `label1.Text` происходит после `await` и завершится ошибкой, если будет выполнена в потоке, отличном от потока пользовательского интерфейса.

Если код, выполняющийся после `await`, запустить в другом потоке, то это нарушит одно из самых полезных свойств `await`: асинхронный код, использующий `await`, пишется так же, как синхронный код. Если воспользоваться полученными в главе 3 знаниями и преобразовать асинхронный метод, использующий `await`, в синхронный, использующий `ContinueWith`, то мы получим результат, показанный в листинге 11.2.

Листинг 11.2. Ошибка доступа к пользовательскому интерфейсу при использовании ContinueWith

```
private void button1_Click(object sender, EventArgs ea)
{
    var result = DoSomething(textBox1.Text).ContinueWith(t=> ←
    {
        label1.Text = t.Result; ← Исключение
    });
}
```

Замена await на ContinueWith

Мы просто взяли код после `await` и поместили его в лямбда-функцию, которую передали в `ContinueWith`. Но это не сработает. Если запустить этот код, то он выбросит исключение, потому что `ContinueWith` всегда запускает обратный вызов в пуле потоков (а не в потоке пользовательского интерфейса). Поэтому мы должны что-то предпринять, чтобы код, записывающий текст в метку, запускался в потоке пользовательского интерфейса. И WinForms, и WPF предоставляют такую возможность. В WinForms это делается с помощью `Control.BeginInvoke`, а в WPF – с помощью `Dispatcher.BeginInvoke` (листинг 11.3).

Листинг 11.3. Доступ к пользовательскому интерфейсу при использовании ContinueWith

```
private void button1_Click(object sender, EventArgs ea)
{
    var result = DoSomething(textBox1.Text).ContinueWith(t=>
    {
        label1.BeginInvoke((Action)((()=> ←
        {
            label1.Text = t.Result;           | Запустит код в потоке
        }));
    });
}
```

В листинге 11.3 мы, как и в листинге 11.2, использовали `ContinueWith`, но на этот раз добавили вызов `Control.BeginInvoke`, чтобы фреймворк WinForms запустил код, который записывает текст в `label1.Text` в потоке пользовательского интерфейса. Теперь доступ к пользовательскому интерфейсу осуществляется только из потока пользовательского интерфейса, и все работает.

Листинги 11.1 и 11.3 делают одно и то же, но разница между ними наглядно показывает, как поведение `await` в многопоточной среде избавляет нас от сложностей работы с потоками.

11.1.2. await в потоках, отличных от потока пользовательского интерфейса

Выше мы рассмотрели случай выполнения кода в потоке пользовательского интерфейса и увидели, почему возврат в тот же поток после `await` так важен. Теперь давайте посмотрим, почему `await` не возвращается в тот же поток, когда он используется в потоках, отличных от потока пользовательского интерфейса. Для демонстрации напишем программу, которая создает поток и выполняет в нем асинхронную операцию (листинг 11.4).

Эта программа создает поток, который асинхронно ждет полсекунды, затем выводит идентификатор потока в консоль до и после ожидания. Главный поток запускает созданный нами поток и затем ждет одну секунду, поскольку программа

завершится сразу после окончания выполнения кода в главном потоке. Мы хотим удержать программу активной, пока второй поток не выполнит свою работу.

Листинг 11.4. Асинхронная операция — это поток, созданный классом Thread

```
var thread = new Thread(async ()=>
{
    Console.WriteLine($"Поток {Thread.CurrentThread.ManagedThreadId}");
    await Task.Delay(500);
    Console.WriteLine($"Поток {Thread.CurrentThread.ManagedThreadId}");
});
thread.Start();
Thread.Sleep(1000);
```

Запустив этот код, мы увидим, что идентификатор потока до и после `await` отличается. Но почему `await` не вернула выполнение в исходный поток, как это происходит при ее вызове из потока пользовательского интерфейса?

Как говорилось в главе 3, `await` подготавливает асинхронную операцию, а затем возвращает управление, в данном случае из главного метода потока (метода, который мы передали конструктору `Thread`). Это заставит поток завершиться (успешно, так как код, запущенный в потоке, выполнил нужную нам операцию). После ожидания в полсекунды, когда наступит время запустить код после `await`, исходный поток, вызвавший `await`, перестанет существовать.

Но что произойдет, если вызвать `await` так, чтобы поток не завершился (листинг 11.5)?

Листинг 11.5. Асинхронная операция без завершения потока

```
var thread = new Thread(()=>
{
    DoSomethingAsync();
    int i=0;
    while(true) Console.Write(++i);
});
thread.IsBackground = true;
thread.Start();
Thread.Sleep(1000);

async Task DoSomethingAsync()
{
    Console.WriteLine($"Поток {Thread.CurrentThread.ManagedThreadId}");
    await Task.Delay(500);
    Console.WriteLine($"Поток {Thread.CurrentThread.ManagedThreadId}");
}
```

В листинге 11.5 код изменен так, что `await` выполняется в методе, который вызывается из главного метода потока. В главном методе потока мы игнорируем объект `Task`, возвращаемый этим методом, и не ждем его завершения. Поскольку

`await` не используется, поддержка `await` в компиляторе не включается и он не добавляет `return`. Соответственно, главный метод потока не возвращает значение и поток не завершается. После вызова асинхронного метода мы запускаем в потоке вечный отсчет, просто чтобы поддержать его в активном состоянии. Этот поток также настроен как фоновый, поэтому приложение закончит работу после завершения главного потока (через одну секунду) и не будет работать вечно.

Запустив этот код, вы увидите, что код до и после `await` выполняется в разных потоках (как и ожидалось). Также вы увидите, что созданный нами поток занят наращиванием счетчика. Даже если система захочет запустить код после `await` в том же потоке, у нее не будет такой возможности. Поток выполняет свой код, и мы не реализовали никакого способа, с помощью которого система могла бы попросить нас запустить код после `await` (например, `Control.BeginInvoke` в WinForms, который использовался в листинге 11.3).

11.2. Контексты синхронизации

Поведение потока пользовательского интерфейса не является необычным или особым случаем для компилятора, предусмотренным только для фреймворков пользовательского интерфейса, написанных Microsoft. Это поведение реализовано в .NET с помощью механизма с именем `SynchronizationContext`.

`SynchronizationContext` — это универсальный способ запуска кода в другом потоке. Представьте, что мы пишем код, который запускает фоновый поток для вычисления чего-либо, а затем использует обратный вызов, чтобы сообщить результат вызывающей стороне. По умолчанию обратный вызов будет запущен в созданном нами фоновом потоке, что неудобно при использовании в приложении с графическим интерфейсом, поскольку попытка доступа к пользовательскому интерфейсу из этого потока вызовет исключение.

Если известно, что этот код всегда будет выполняться, например, только в приложениях WinForms, то можно использовать метод `Control.BeginInvoke`, как мы сделали в листинге 11.3. Однако у этого способа есть очевидное ограничение: он работает только в WinForms. В приложениях WPF придется использовать `Dispatcher`, а в других фреймворках — другие механизмы.

Чтобы такой код работал в любой ситуации, нельзя использовать `Control.BeginInvoke` напрямую. Но можно создать абстрактный класс, представляющий код, запущенный в другом потоке, и потребовать от того, кто использует этот код, реализовать его. Класс может выглядеть так:

```
public abstract class RunInOtherThread
{
    public abstract void Run(Action codeToRun);
}
```

Тот, кто пишет приложение для WPF, реализует метод `Run` с помощью `Dispatcher.BeginInvoke`; тот, кто пишет приложение для WinForms, реализует его с помощью `Control.BeginInvoke`; и т. д. для любого фреймворка, накладывающего ограничения на многопоточное выполнение кода. Таким способом можно запускать обратные вызовы в наиболее удобном для пользователя потоке и не зависеть от какого-либо фреймворка пользовательского интерфейса. В качестве дополнительного преимущества этот прием также будет работать с будущими фреймворками, даже неизвестными нам.

`SynchronizationContext` — это реализация нашего класса `RunInOtherThread`, встроенная в .NET. Он поддерживает два способа запуска кода в целевом потоке: метод `Send`, который будет ждать, пока другие потоки закончат выполнение нашего кода, и метод `Post`, который не ждет. Инструкция `await` использует только `Post`.

Но мы ни в одном примере не передавали `SynchronizationContext` в инструкцию `await`. Как она определяет, какой поток выбрать? Для этого `SynchronizationContext` можно связать с потоком, вызвав `SynchronizationContext.SetSynchronizationContext`. После этого любой код, работающий в этом потоке, сможет прочитать `SynchronizationContext.Current`, чтобы получить его. WinForms, WPF и ASP.NET Classic реализуют класс, производный от `SynchronizationContext`, и связывают его с потоками пользовательского интерфейса или обработки запросов, поэтому любой универсальный код, который должен вернуться в правильный поток (например, `await`), сможет его использовать.

Давайте напишем класс, производный от `SynchronizationContext`, который запускает код после `await` в том же потоке. Для этого используем `BlockingCollection` и паттерн очереди заданий, о котором говорилось в главе 8 (листинг 11.6).

Листинг 11.6. Нестандартная версия `SynchronizationContext` для `async/await`

```
using System.Collections.Concurrent;

public class SingleThreadSyncContext : SynchronizationContext
{
```

Начнем с метода `Run`, который запускает очередь заданий. Поскольку мы хотим запустить все в текущем потоке, метод `Run` не возвращает управление, пока мы не закончим. Метод `Run` принимает метод для запуска, поскольку без него никто не сможет использовать наш `SynchronizationContext` (доступ к `SynchronizationContext` будет иметь только код, который мы запускаем после вызова `SetSynchronizationContext`, поэтому мы должны запустить свой код в этой точке):

```

public static void Run(Func<Task> startup)
{
    var prev = SynchronizationContext.Current;
    try
    {
        var ctxt = new SingleThreadSyncContext();
        SynchronizationContext.SetSynchronizationContext(ctxt);
        ctxt.Loop(startup); ← Связет SynchronizationContext
        ↗ с потоком
    }
    finally
    {
        SynchronizationContext.SetSynchronizationContext(prev); ←
        ↗ Восстанавливает
        ↗ оригинальный
        ↗ SynchronizationContext
    }
}

```

Теперь мы реализуем очередь заданий. Подобно тому как мы делали это в главе 8, создадим очередь делегатов, представляющих задания, которые нужно выполнить. Используя `foreach` с `GetConsumingEnumerable`, мы получаем следующий элемент из очереди или ждем, если очередь пустая, а затем вызываем делегата, полученного из очереди:

```

private BlockingCollection<(SendOrPostCallback call, object? state)>
    _queue = new();

private void Loop(Func<Task> startup) ← Прекращает работу
{                                         после завершения
    startup().ContinueWith(t => _queue.CompleteAdding()); ← первого метода
    foreach(var next in _queue.GetConsumingEnumerable())
    {
        next.call(next.state); ← Цикл очереди
    }                                     заданий
}

```

Последняя часть — метод `Post`, который добавит задание в очередь. Когда асинхронная операция завершит работу, `await` вызовет его для запуска кода после `await` (через `TaskScheduler`; о нем мы поговорим далее в этой главе):

```

public override void Post(SendOrPostCallback d, object? state)
{
    _queue.Add((d, state));
}

```

Осталось реализовать методы `SynchronizationContext`, которые не используются инструкцией `await`. В них мы просто выбросим исключение `NotImplementedException`:

```

public override void Send(SendOrPostCallback d, object? state)
{
    // не используется инструкциями async/await
    throw new NotImplementedException();
}

```

```

public override SynchronizationContext CreateCopy()
{
    // не используется инструкциями async/await
    throw new NotImplementedException();
}

public override int Wait(IntPtr[] waitHandles,
    bool waitAll, int millisecondsTimeout)
{
    // не используется инструкциями async/await
    throw new NotImplementedException();
}
}

```

Теперь просто протестируем нашу реализацию `SynchronizationContext`. Сначала запустим простую асинхронную операцию без него (листинг 11.7).

Листинг 11.7. Простая асинхронная операция без SingleThreadSyncContext

```

Console.WriteLine($"перед await {Thread.CurrentThread.ManagedThreadId}");
await Task.Delay(500);
Console.WriteLine($"после await {Thread.CurrentThread.ManagedThreadId}");

```

Этот код выводит идентификатор текущего потока, вызывает `await`, а затем снова выводит идентификатор текущего потока. Запустив этот код, мы увидим, что код после `await` выполняется в другом потоке, отличном от того, в котором выполняется код до `await`, как и ожидалось.

А теперь запустим тот же код с нашим `SingleThreadSyncContext` (листинг 11.8).

Листинг 11.8. Простая асинхронная операция с SingleThreadSyncContext

```

SingleThreadSyncContext.Run(async ()=>
{
    Console.WriteLine($"до {Thread.CurrentThread.ManagedThreadId}");
    await Task.Delay(500);
    Console.WriteLine($"после {Thread.CurrentThread.ManagedThreadId}");
});

```

Здесь код из листинга 11.7 помещен в лямбда-функцию и передан в `SingleThreadSyncContext.Run`. Запустив этот код, мы увидим, что идентификатор потока не изменился и код до и после `await` запускается в одном и том же потоке (в частности, в потоке, который вызвал `SingleThreadSyncContext.Run`).

WinForms и WPF используют свою версию `SynchronizationContext` в потоке пользовательского интерфейса, и, как мы увидели в нашем примере, любой сторонний код тоже может использовать свою версию `SynchronizationContext`, вызывая `SetSynchronizationContext`. Но если никто не вызвал `SetSynchronizationContext` в текущем потоке, как почти во всех приложениях без графического интерфейса (например, в консольных приложениях и приложениях ASP.NET Core),

а также в потоках, отличных от потока пользовательского интерфейса, то `SynchronizationContext.Current` будет иметь значение `null` и, как мы видели в листинге 11.7, код после `await` будет запущен в пуле потоков.

11.3. Переключение потоков — `ConfigureAwait(false)`

Мы обсудили, почему и как `await` возвращается в тот же поток в приложениях с графическим интерфейсом. Теперь обсудим, зачем может понадобиться блокировать такое поведение и как это сделать.

Вызов `ConfigureAwait(false)` позволяет запретить инструкции `await` использовать текущий объект `SynchronizationContext`, а также текущий объект `TaskScheduler` (который мы обсудим далее в этой главе). Сначала я хочу развенчать распространённое, но неверное представление о `ConfigureAwait(false)`, что «без `ConfigureAwait(false)` выполнение продолжится в том же потоке, а с `ConfigureAwait(false)` — в другом». Начнем с первой части этого мнения (листинг 11.9).

Листинг 11.9. Консольное приложение без `ConfigureAwait(false)`

```
Console.WriteLine($"1: {Thread.CurrentThread.ManagedThreadId}");
await Task.Delay(500);
Console.WriteLine($"2: {Thread.CurrentThread.ManagedThreadId}");
```

Если запустить эту программу как консольное приложение, то можно увидеть, что при вызове `await` без `ConfigureAwait(false)` она переключит потоки. Причина в том, что, пока `await` пытается оставаться в том же `SynchronizationContext` (не в потоке), у нас не остается ни одного потока, поэтому `await` продолжит выполнение в пуле потоков.

Теперь вторая часть: всегда ли `ConfigureAwait(false)` вызывает переключение на другой поток (листинг 11.10)?

Листинг 11.10. `ConfigureAwait(false)` и завершение задач

```
Console.WriteLine($"1: {Thread.CurrentThread.ManagedThreadId}");
await DoSomething().ConfigureAwait(false);
Console.WriteLine($"2: {Thread.CurrentThread.ManagedThreadId}");

async Task DoSomething()
{
    Console.WriteLine("что-то выполнилось");
}
```

Запустив этот код, вы увидите, что использование `ConfigureAwait(false)` не привело к переключению потоков. Так получилось, потому что `DoSomething` всегда возвращает завершенную задачу, а применение `await` к завершенной задаче просто продолжает работу в том же потоке.

Почему `DoSomething` возвращает завершенную задачу? Как уже говорилось в главе 3, упрощенная, но в основном правильная модель (за исключением контекстов синхронизации) обработки асинхронных методов компилятором заключается в том, что каждая инструкция `await` заменяется вызовом `Task.ContinueWith`. Поскольку `DoSomething` не использует `await`, компилятор ничего не изменяет. Метод остается точно таким же, каким был бы, если бы он не был асинхронным, за исключением того, что он возвращает `Task`, поэтому он преобразуется примерно в такой код:

```
Task DoSomething()
{
    Console.WriteLine("что-то выполнилось");
    return Task.CompletedTask;
}
```

Напомню, что объявление метода асинхронным не заставляет его работать в фоновом режиме, а лишь включает поддержку `await` компилятором.

Можно подумать, что наш метод `DoSomething` — редкий и нетипичный случай или даже ошибка, однако методы, которые возвращают уже завершенную задачу `Task`, совсем не редкость. Во многих библиотеках есть синхронные методы, возвращающие задачу `Task`, в основном потому, что автор хочет иметь возможность поддерживать асинхронные операции в будущем без изменения API, или потому, что операция была асинхронной в предыдущей версии.

Теперь давайте посмотрим, что на самом деле делает `ConfigureAwait(false)`. Начнем с кода WinForms в листинге 11.11.

Листинг 11.11. Обработчик событий WinForms с `ConfigureAwait`

```
private async void Button1_Click(object sender, EventArgs ea)
{
    Debug.WriteLine($" до: {Thread.CurrentThread.ManagedThreadId }");
    await Task.Delay(500).ConfigureAwait(false);
    Debug.WriteLine($" после: {Thread.CurrentThread.ManagedThreadId }");
}
```

Этот код использует `await` с `ConfigureAwait(false)`, чтобы приостановиться на полсекунды, и выводит идентификатор потока, полученный до и после вызова `await`. Мы знаем, что без `ConfigureAwait(false)` будет выведен один и тот же идентификатор потока, но, запустив этот код, мы увидим два разных идентификатора, точно так же, как в случае с кодом в листинге 11.9. Разница между листингами 11.9 и 11.11 в том, что код в листинге 11.9 является консольным приложением и как таковой не имеет `SynchronizationContext`, тогда как в листинге 11.11 показан фрагмент приложения WinForms и, следовательно, он имеет `SynchronizationContext`. Поэтому `ConfigureAwait(false)` просто игнорирует любой `SynchronizationContext`, связанный с текущим потоком.

Итак, обобщая все, о чем говорилось до сих пор, получаем следующие правила, которые определяют, где должен выполняться код после `await`:

- если задача уже завершилась, то код продолжает выполняться в том же потоке — `ConfigureAwait(false)` в этом случае не оказывает никакого влияния;
- если для текущего потока задан `SynchronizationContext` и `ConfigureAwait(false)` не задействуется, то для выполнения кода после `await` будет использоваться `SynchronizationContext`;
- во всех остальных случаях код будет выполняться в пуле потоков.

А теперь я хочу рассказать еще кое-что, что может показаться спорным для тех, кому доводилось читать списки рекомендуемых приемов применения `async/await`: не используйте `ConfigureAwait(false)` каждый раз, когда используете `await`.

Многие рекомендуют обязательно задействовать `ConfigureAwait(false)` всякий раз, когда используется `await`. Они говорят, что неправильное использование `async/await` в приложениях с графическим интерфейсом и в классических приложениях ASP.NET (не ASP.NET Core) может привести к взаимоблокировкам, а `ConfigureAwait(false)` поможет предотвратить их. Однако это неверно, и Microsoft согласна со мной и в своем официальном руководстве говорит, что `ConfigureAwait(false)` следует использовать в коде библиотек, а не приложений.

В очень многих рекомендациях упоминается часть «всегда используйте `ConfigureAwait(false)`», но опускается часть «в коде библиотек», из-за чего кажется, что простое добавление `ConfigureAwait(false)` в любом коде предотвратит взаимоблокировки и избавит разработчика от необходимости задумываться об этом или отлаживать свой код. Однако это только кажется. Я покажу проблему этого подхода и другие решения, помогающие предотвратить взаимоблокировку. Но сначала рассмотрим проблему (листинг 11.12).

Листинг 11.12. Взаимоблокировка `async/await`

```
private void button1_Click(object sender, EventArgs ea)
{
    var task = DoSomething();
    label1.Text = task.Result;
}

private async Task<string> DoSomething()
{
    await Task.Delay(500);
    return "выполнено";
}
```

Этот код объединяет асинхронные и блокирующие вызовы. Первый метод, `button1_Click`, вызывает `DoSomething` без `await`, а затем читает свойство `Task.Result` и блокирует поток, потому что `DoSomething` еще не завершился. Тем временем

внутри `DoSomething` вызов `Task.Delay` завершается, и следующая строка (`return "выполнено"`) готова к выполнению. Как мы уже видели, код после `await` будет запущен в потоке пользовательского интерфейса, но поток пользовательского интерфейса занят ожиданием `Task.Result`.

Итак, здесь поток пользовательского интерфейса ожидает завершения `DoSomething`, но `DoSomething` не может завершиться, пока поток пользовательского интерфейса не освободится — классическая взаимоблокировка. Если просто добавить `ConfigureAwait(false)` к `await`, то получится следующее (листинг 11.13).

Листинг 11.13. Предотвращение взаимоблокировки с помощью `ConfigureAwait(false)`

```
private void button1_Click(object sender, EventArgs ea)
{
    var task = DoSomething();
    label1.Text = task.Result;
}

private async Task<string> DoSomething()
{
    await Task.Delay(500).ConfigureAwait(false); ← Добавлен вызов ConfigureAwait
    return "выполнено";
}
```

Теперь в `button1_Click` поток пользовательского интерфейса все так же блокируется, ожидая завершения `DoSomething`, как и в листинге 11.12, но на этот раз код в `DoSomething` после завершения `Task.Delay` продолжит выполняться в пуле потоков, а не в потоке пользовательского интерфейса. Это означает, что `DoSomething` завершится в фоновом режиме, `Task.Result` снимет блокировку, вернет результат и код просто продолжит работу.

Итак, если простое механическое добавление `ConfigureAwait(false)` предотвращает взаимоблокировки, то почему я так против этого? В этом случае есть более простое решение: если не смешивать асинхронные и блокирующие операции и просто использовать `await`, то проблемы и не возникнет (листинг 11.14).

Листинг 11.14. Предотвращение взаимоблокировки с помощью `await`

```
private async void button1_Click(object sender, EventArgs ea)
{
    var result = await DoSomething(); ← Использована инструкция await
    label1.Text = result;
}

private async Task<string> DoSomething()
{
    await Task.Delay(500);
    return "выполнено";
}
```

Мы решили проблему, использовав `await` вместо чтения `Task.Result`. Теперь `button1_Click` не блокирует поток, пока `DoSomething` не завершится, и состояние взаимоблокировки не возникает. Обратите внимание, что не всегда есть возможность использовать `await`, такое может произойти, например, если мы изменим способ работы программы, чтобы предусмотреть обработку исключений, которые могут возникнуть в `DoSomething`. Мы рассмотрим эту проблему немного позже, но сначала посмотрим, что произойдет, если добавить `ConfigureAwait` в эту версию кода (листинг 11.15).

Листинг 11.15. Код WinForms с `ConfigureAwait(false)` повсюду

```
private async void button1_Click(object sender, EventArgs ea)
{
    var result = await DoSomething().ConfigureAwait(false); ← Добавлен вызов
    label1.Text = result; ← Исключение
}

private async Task<string> DoSomething()
{
    await Task.Delay(500).ConfigureAwait(false);
    return "выполнено";
}
```

Здесь мы добавили `ConfigureAwait(false)` к первому вызову `await`. Поэтому теперь код после `await` будет запущен в пуле потоков, а не в потоке пользовательского интерфейса, но он изменяет пользовательский интерфейс (теперь уже из неправильного потока) и в результате выбросит исключение `InvalidOperationException`.

Чтобы этот код с `ConfigureAwait(false)` заработал, нужно использовать `Control.BeginInvoke`, как это делалось в листинге 11.3, где мы вообще не использовали `await` (листинг 11.16).

Листинг 11.16. Код WinForms, который работает с `ConfigureAwait(false)` повсюду

```
private async void button1_Click(object sender, EventArgs ea)
{
    var result = await DoSomething().ConfigureAwait(false); ← ConfigureAwait
    label1.BeginInvoke((Action)(()=> ← Записывает текст
    {
        label1.Text = result)); ← в метку в потоке
    }); ← пользователяского
} ← интерфейса

private async Task<string> DoSomething()
{
    await Task.Delay(500).ConfigureAwait(false);
    return "выполнено";
}
```

В этом примере, поскольку код после `await` запускается не в потоке пользовательского интерфейса (из-за использования `ConfigureAwait(false)`), нам пришлось использовать другие способы, чтобы запустить код, обновляющий пользовательский интерфейс, в потоке пользовательского интерфейса, и результат получился очень похожим на код, который вообще не использует `await`. Этот пример показывает, что `ConfigureAwait(false)` сводит на нет все преимущества `await`.

Но `ConfigureAwait` все же не воплощение зла. Если использовать `ConfigureAwait(false)` только с `await` внутри `DoSomething`, то код будет работать исправно (листинг 11.17).

Листинг 11.17. Код WinForms с `ConfigureAwait(false)` только в методах, не работающих с пользовательским интерфейсом

```
private async void button1_Click(object sender, EventArgs ea)
{
    var result = await DoSomething(); ← Без ConfigureAwait вернется в контекст
    label1.Text = result; ← пользователяского интерфейса
}
private async Task<string> DoSomething()
{
    await Task.Delay(500).ConfigureAwait(false);
    return "выполнено";
}
```

Исключение
не выбрасывается

Здесь код внутри `DoSomething` выходит из контекста пользовательского интерфейса, и инструкция `return` будет запущена в пуле потоков. Но инструкция `await` в `button1_Click` (не использующая `ConfigureAwait(false)`) вернет выполнение в поток пользовательского интерфейса, и код, изменяющий метку, будет работать.

Обратите также внимание, что метод `DoSomething` в этом листинге выглядит точно так же, как в листинге 11.12, поэтому вызов `ConfigureAwait(false)` может быть полезен в случаях, когда неизвестно, является вызывающий код асинхронным, как в листинге 11.16, или блокирующим, как в листинге 11.13, что типично для случаев, когда мы пишем библиотеку. Вы можете видеть, что `ConfigureAwait(false)` исправляет проблему, потому что сам `DoSomething` не обращается к пользовательскому интерфейсу и потому для него неважно, в каком контексте выполняется код после `await`.

Напомню, что даже в библиотеках часто бывает нужно позаботиться о контексте (и потоке), в котором выполняется код. Очевидным примером может служить библиотека, разработанная специально для использования в приложении с графическим интерфейсом. Менее очевидным примером является библиотека, которая использует обратные вызовы для уведомления приложения, которое ее использует. Даже если библиотеке безразлично, в каком потоке она выполняется, коду внутри обратного вызова может быть не все равно.

Прежде чем закончить обсуждение `ConfigureAwait(false)`, я расскажу, как можно решить проблему с взаимоблокировкой без использования `ConfigureAwait`. Взаимоблокировка в ее общей форме — пользовательский интерфейс, который ждет завершения некоторого кода, который, в свою очередь, ждет, когда освободится поток пользовательского интерфейса, — существует с тех пор, как мы начали создавать приложения с графическим интерфейсом. Она появилась до `async/await`, до .NET и C#, и даже до добавления поддержки асинхронного ввода-вывода в Windows. Поэтому неудивительно, что стандартное решение этой проблемы уже встроено в WinForms (а также в WPF и другие известные мне фреймворки пользовательского интерфейса). Это решение заключается в том, чтобы позволить пользовательскому интерфейсу обрабатывать события (или *перекачивать сообщения* в терминологии Windows API), пока ожидается завершение фоновой задачи. В WinForms это обеспечивается вызовом метода `Application.DoEvents` (листинг 11.18).

Листинг 11.18. Предотвращение взаимоблокировки с помощью `DoEvents`

```
private void button1_Click(object sender, EventArgs ea)
{
    var task = DoSomething();
    while(!task.IsCompleted)
        Application.DoEvents(); | Продолжает обрабатывать
        label1.Text = task.Result; | сообщения в процессе ожидания
}

private async Task<string> DoSomething()
{
    await Task.Delay(500)
    return "выполнено";
}
```

Этот код работает точно так же, как мы ожидали от кода в листинге 11.11. Взаимоблокировка не возникает, метод `button1_Click` неасинхронный, и, как дополнительное преимущество, пользовательский интерфейс не зависает на время выполнения `DoSomething`.

В заключение приведу свои правила использования `ConfigureAwait(false)`:

- если вы пишете код приложения, избегайте использования `ConfigureAwait(false)`; поведение по умолчанию существует не просто так;
- если ваш код предназначен только для работы в средах, где нет `SynchronizationContext` (например, в консольных приложениях и приложениях ASP.NET Core), то не используйте `ConfigureAwait(false)`;
- если вы пишете библиотечный код, которому безразлично, в каком контексте выполняться, то используйте `ConfigureAwait(false)` для каждой инструкции `await`;

- если вы хотите покинуть текущий контекст, используйте `Task.Run` вместо `ConfigureAwait(false)`, потому что `ConfigureAwait(false)` ничего не делает, если задача `Task` уже завершилась.

Примечание о `DoEvents`

Обратите внимание, что цикл `DoEvents` нагрузит процессор (одно ядро) на 100 % на время ожидания. Это не повлияет на приложение (в частности, на поток пользовательского интерфейса), потому что цикл будет максимально быстро обрабатывать все события пользовательского интерфейса, но он расходует ресурсы, которые могли бы использоваться другим приложением, и не дает ядру процессора переключиться в режим энергосбережения. Поэтому я не рекомендую использовать `DoEvents`, и тем не менее это решение лучше, чем попасть в состояние взаимоблокировки. В этом примере допустимо использовать `DoEvents`, поскольку мы знаем, что ожидание длится всего полсекунды и влияние на систему будет минимальным, однако мы должны учитывать недостатки `DoEvents`, когда пишем циклы, вызывающие этот метод.

Есть некоторые фреймворки модульного тестирования, которые не будут работать, если ваш код не всегда использует `ConfigureAwait(false)`. Лично я считаю это ошибкой в фреймворке тестирования и призываю вас подумать, что лучше: изменить поведение потоков своего приложения, чтобы компенсировать техническую ошибку во фреймворке модульного тестирования, или использовать другой фреймворк модульного тестирования.

После обсуждения `ConfigureAwait(false)`, возможно, вам будет интересно узнать о `ConfigureAwait(true)`. `ConfigureAwait(true)` — это поведение по умолчанию, которое не оказывает никакого влияния на ваш код (за исключением отключения статических анализаторов кода, которые жалуются на неиспользование `ConfigureAwait(false)`).

11.4. Дополнительные параметры `ConfigureAwait`

В .NET 8 в `ConfigureAwait` были добавлены новые интересные параметры, которые еще больше усложняют ситуацию. Их поддержка реализована в виде перегруженной версии `ConfigureAwait`, которая принимает параметр `ConfigureAwaitOptions`. К сожалению, несмотря на определенную полезность в особых случаях, все эти параметры влекут за собой неочевидные сложности, поэтому я рекомендую не использовать их в обычных приложениях.

На тот случай, если вам доведется столкнуться с ними в коде, который нужно будет отладить, я перечислю поддерживаемые параметры, но имейте в виду, что каждый из них таит в себе подводные камни.

- `None` — вызов `ConfigureAwait(ConfigureAwaitOptions.None)` эквивалентен вызову `ConfigureAwait(false)`, то есть изменяет поведение инструкции

`await` и заставляет ее запускать продолжения в пуле потоков. Я рекомендую использовать старый добрый вызов `ConfigureAwait(false)` вместо нового `ConfigureAwait(ConfigureAwaitOptions.None)`. Новая версия не так отчетливо говорит о том, что она делает (учитывая значение слова *none* («ничего») в английском языке, я бы ожидал, что параметр `None` ничего не делает, но на самом деле он меняет поведение), а старая версия хотя бы более лаконична.

- `ContinueOnCapturedContext` — это значение сохраняет поведение `await` по умолчанию и само по себе ничего не меняет. Оно необходимо, потому что параметры `ConfigureAwaitOptions` являются флагами и их можно комбинировать вместе, а `None` — это значение по умолчанию. Все остальные параметры также включают поведение `ConfigureAwait(false)`, если не комбинировать их с параметром `ContinueOnCapturedContext`.
- `ForceYielding` — это значение заставляет `await` всегда возвращать управление и планировать запуск продолжения на более позднее время, даже если в этом нет необходимости, потому что задача `Task` уже завершилась. Этот флаг не заставляет вызываемый код работать в фоновом режиме, а просто выбирает пул потоков для выполнения кода после `await`. Использование этого флага эквивалентно использованию инструкции `await Task.Yield().ConfigureAwait(false);` в следующей строке.
- `SuppressThrowing` — этот флаг заставляет `await` игнорировать некоторые ошибки. Он предназначен для ситуаций, когда вам безразлично, завершились ли операция успешно или нет. Однако игнорироваться будут только ошибки, которые возникают после первой инструкции `await` внутри вызываемого вами метода, поэтому данный флаг не гарантирует, что не будет выброшено исключение. Кроме того, попытка использовать этот флаг с `Task<T>` приведет к исключению.

Резюме: из четырех новых параметров `None` и `ContinueOnCapturedContext` можно выразить с помощью старой версии `ConfigureAwait`, причем более компактно, параметр `ForceYielding` не особенно полезен, а `SuppressThrowing` не в полной мере соответствует своему имени. Кроме того, новая версия `ConfigureAwait` добавляет еще одну потенциальную ловушку: она не поддерживается в `ValueTask` и `ValueTask<T>`, поэтому я советую придерживаться старой версии `ConfigureAwait(bool)`.

11.5. Task.Yield: возможность выполнить другой код

Внутри системы события, генерируемые Windows (движения мыши, нажатия клавиш клавиатуры и т. д.), и задания, добавленные с помощью `Control.BeginInvoke` и `SynchronizationContext`, хранятся в так называемой очереди ввода (потому что ее основная функция — передача ввода из пользовательского интерфейса в приложение).

В листинге 11.18 было показано, что в WinForms с помощью `Application.DoEvents` можно позволить фреймворку обрабатывать события и оставаться отзывчивым. `DoEvents` просто читает ожидающие элементы из очереди ввода и возвращает управление только после обработки всех событий.

Однако существует универсальная версия `DoEvents`, совместимая с `async/await`, — `Task.Yield()`. Когда вызов `Task.Yield()` с помощью `await` выполняется в потоке пользовательского интерфейса, то код после `await` добавляется в конец входной очереди и выполняется после всех других событий, которые уже ожидают обработки. В листинге 11.19 показано, что произойдет, если создать приложение WinForms, которое вечно будет наращивать счетчик.

Листинг 11.19. Попытка запустить вечное наращивание счетчика и зависание приложения

```
private void button1_Click(object sender, EventArgs ea)
{
    int i=0;
    while(true)
    {
        label1.Text = (++i).ToString();
    }
}
```

При использовании этого метода, когда пользователь нажмет кнопку, программа войдет в бесконечный цикл, наращивая счетчик и записывая его значение в свойство `Text` метки, то есть программа просто зависнет. Поток пользовательского интерфейса будет занят постоянным увеличением счетчика и не сможет обрабатывать события из очереди ввода, такие как щелчки мыши. Кроме того, вы не увидите изменения текста метки, потому что поток пользовательского интерфейса также не будет обрабатывать запросы на перерисовку метки. Чтобы исправить эту проблему, можно использовать ту же стратегию, которая так хорошо сработала в листинге 11.18, основанную на вызове `DoEvents` (листинг 11.20).

Листинг 11.20. Вечное наращивание счетчика без зависания приложения

```
private void button1_Click(object sender, EventArgs ea)
{
    int i=0;
    while(true)
    {
        label1.Text = (++i).ToString();
        Application.DoEvents();
    }
}
```

Теперь в каждой итерации вызывается метод `DoEvents`. Он обработает все ожидающие события (включая события перерисовки метки). Приложение останется отзывчивым, а метка будет отображать изменяющееся число. Обратите внимание, что этот код нагрузит ядро процессора на все 100 %, поэтому, в отличие от листинга 11.18, он не оказывает отрицательного влияния на систему, вызывая `DoEvents`. Мы могли бы также пойти по пути `async/await` и сделать то же самое с помощью `Task.Yield` (листинг 11.21).

Листинг 11.21. Вечное наращивание счетчика без зависания приложения с использованием `await`

```
private async void button1_Click(object sender, EventArgs ea)
{
    int i=0;
    while(true)
    {
        label1.Text = (++i).ToString();
        await Task.Yield();
    }
}
```

В этой версии вызов `await Task.Yield()` в конце каждой итерации поставит метод в конец входной очереди и тут же вернет управление. Это совсем другая механика, отличная от `DoEvents`, но с точно такими же результатами.

Выбор потока для запуска кода после `await Task.Yield()` осуществляется в соответствии с теми же правилами, что перечислены в этой главе, поэтому вы можете использовать их, чтобы выяснить, в каком потоке будет выполняться ваш код в каждой конкретной ситуации.

11.6. Планировщики задач

В этой главе я неоднократно писал, что `await` использует `SynchronizationContext`. Но это еще не все: `async/await` имеют собственную инфраструктуру принятия решения о выборе потока для запуска кода. Эта инфраструктура основана на классе `TaskScheduler`. Классы, производные от `TaskScheduler`, как и классы, производные от `SynchronizationContext`, знают, как получить задачу, созданную `await`, и запланировать ее выполнение в некотором потоке когда-нибудь в будущем.

Планировщик задач по умолчанию (доступный через статическое свойство `TaskScheduler.Default`) всегда ставит код в очередь для выполнения в пуле потоков. Если вызвать `await` в потоке с `SynchronizationContext`, то компилятор создаст планировщик (вызовом `TaskScheduler.FromCurrentSynchronizationContext`),

который будет использовать свой метод `Post` для планирования кода. Текущий планировщик можно получить, прочитав `TaskScheduler.Current`.

В отличие от `SynchronizationContext`, нельзя установить текущий `TaskScheduler`, но можно установить планировщик задач при вызове `Task.Run`, `ContinueWith` или при использовании для запуска кода любого другого способа, совместимого с `async/await`. Например, следующий код запустит лямбда-функцию через полсекунды в пуле потоков:

```
Task.Delay(500).ContinueWith(t=>Console.WriteLine("Привет"));
```

Этот код использует `ContinueWith` для запуска после истечения тайм-аута, переданного в `Task.Delay`. Поскольку мы не передали `TaskScheduler` в необязательном параметре, будет использоваться планировщик по умолчанию, который запустит код в пуле потоков. Однако в потоке с `SynchronizationContext` можно создать планировщик задач, который его использует:

```
Task.Delay(500).ContinueWith(t=>Console.WriteLine("Привет"),
    TaskScheduler.FromCurrentSynchronizationContext());
```

Здесь мы передали необязательный параметр с `TaskScheduler` — в данном случае планировщик, созданный из текущего `SynchronizationContext`, поэтому лямбда-функция будет работать в этом потоке. Если текущего `SynchronizationContext` нет или `SynchronizationContext` нельзя заключить в `TaskScheduler`, то `FromCurrentSynchronizationContext` выбросит исключение. В версии .NET, которая была текущей на момент написания этой книги (версия 8), такое происходит, только если нет текущего `SynchronizationContext` (то есть `SynchronizationContext.Current` имеет значение `null`).

Как и в случае с `SynchronizationContext`, вызов `ConfigureAwait(false)` заставит `await` игнорировать текущий `TaskScheduler` и использовать планировщик по умолчанию.

В результате мы получаем правила выбора потока инструкцией `await` (выбирается первое подходящее правило).

- Если задача уже завершилась, то код продолжит выполняться немедленно в том же потоке. В этом случае `ConfigureAwait(false)` не дает никакого эффекта.
- Если в текущем потоке установлен `SynchronizationContext` и отсутствует вызов `ConfigureAwait(false)`, то будет использоваться `SynchronizationContext`.
- Если с текущей задачей связан `TaskScheduler` и отсутствует вызов `ConfigureAwait(false)`, то будет использоваться `TaskScheduler`.

- Если производится вызов `ConfigureAwait(false)` или поток не имеет `SynchronizationContext` и `TaskScheduler`, то будет использоваться планировщик задач по умолчанию, а код запустится в пуле потоков.

Итоги главы

- Вот упрощенные правила выбора потока для запуска кода после `await`:
 - в приложениях с графическим интерфейсом (WinForms и WPF), если `await` вызывается в потоке пользовательского интерфейса и отсутствует вызов `ConfigureAwait(false)`, то код после `await` будет выполняться в том же потоке;
 - в ASP.NET Classic (не ASP.NET Core), если `await` вызывается в потоке, обрабатывающем веб-запрос, и отсутствует вызов `ConfigureAwait(false)`, то код после `await` будет выполняться в том же потоке;
 - во всех остальных случаях код после `await` будет выполняться в пуле потоков.
- Однако фактически применяются следующие правила:
 - если задача уже завершилась и не используется `ConfigureAwait(ConfigureAwaitOptions.ForceYielding)`, то код продолжит выполняться немедленно в том же потоке, `ConfigureAwait(false)` в этом случае не дает никакого эффекта;
 - если в текущем потоке установлен `SynchronizationContext` и отсутствует вызов `ConfigureAwait(false)`, то будет использоваться `SynchronizationContext`;
 - если с текущей задачей связан `TaskScheduler` и отсутствует вызов `ConfigureAwait(false)`, то будет использоваться `TaskScheduler`;
 - если имеется вызов `ConfigureAwait(false)` или поток не имеет `SynchronizationContext` и `TaskScheduler`, то будет использоваться планировщик задач по умолчанию, а код продолжит выполнение в пуле потоков.
- Если вы не используете сторонние фреймворки и не реализовали свой класс `SynchronizationContext` или `TaskScheduler`, то эти два набора правил дадут одинаковые результаты.
- `ConfigureAwait(false)` заставляет `await` игнорировать текущий `SynchronizationContext` или `TaskScheduler`. Такое поведение помогает предотвратить взаимоблокировки, но может сделать использование `await` гораздо менее удобным.

- Вот правила использования `ConfigureAwait(false)`:
 - при разработке приложения не используйте `ConfigureAwait(false)`. Поведение по умолчанию существует не просто так;
 - если ваш код должен продолжить выполнение в том же потоке, например, если вы изменили настройки потока или используете локальное хранилище, то не используйте `ConfigureAwait(false)`;
 - если ваш код предназначен для работы в средах, не использующих `SynchronizationContext` (например, в консольных приложениях и в ASP.NET Core), то не используйте `ConfigureAwait(false)`;
 - если вы пишете библиотечный код и вам неважно, в каком контексте будет выполняться ваш код, то используйте `ConfigureAwait(false)` для каждого вызова `await`;
 - если вы хотите покинуть текущий контекст, то используйте `Task.Run` вместо `ConfigureAwait(false)`, потому что `ConfigureAwait(false)` ничего не делает, если задача уже завершилась.

12

async/await и исключения

В этой главе

- ✓ Как работают исключения в асинхронном коде.
- ✓ Как восстановить потерянные исключения.
- ✓ Обработка исключений в асинхронных методах `void`.

В этой главе мы поговорим об исключениях. Обсудим их работу в асинхронном коде и отличия от работы в синхронном коде. В простом синхронном коде исключения всплывают вверх по стеку вызовов. Как мы видели в главах 3, 5 и 11, в асинхронном коде постоянно происходит регистрация обратных вызовов для последующего вызова, часто в других потоках, поэтому стек вызовов больше не описывает поток выполнения кода. Понимание этой особенности и знание механизмов, которые предлагают `async/await` для смягчения данной проблемы, очень пригодятся при отладке исключений в асинхронном коде, то есть при любой ситуации, когда асинхронный код терпит сбой непрямым образом. Мы также рассмотрим некоторые ловушки, связанные с исключениями, о которых следует знать.

12.1. Исключения и асинхронный код

Исключения задействуют стек вызовов. Стек вызовов — это структура данных, используемая системой для реализации концепции методов (функций или процедур, в зависимости от языка программирования). Когда код вызывает метод, система помещает адрес следующей инструкции в стек вызовов, а когда

выполняется оператор `return`, система переходит к адресу, хранящемуся на вершине стека (это весьма грубое упрощение, однако эта книга не об аппаратной архитектуре процессоров).

Если метод выбросит исключение внутри блока `try` с соответствующей инструкцией `catch`, то управление будет передано инструкции `catch`. Если метод выбросит исключение вне блока `try` или в этом блоке `try` нет соответствующей инструкции `catch`, то исключение начнет всплывать вверх по стеку вызовов, пока не будет найдена соответствующая инструкция `catch`. Если будет достигнуто начало стека вызовов и соответствующая инструкция `catch` так и не будет найдена, то программа аварийно завершит работу.

Исключение всплывает, используя стек вызовов программы, а не структуру исходного кода. Например, давайте рассмотрим код, структура которого не соответствует поведению во время выполнения:

```
public void MyMethod()
{
    try
    {
        Win.Click += ()=>
        {
            throw new NotImplementedException();
        };
    }
    catch
    {
        Console.WriteLine("В блоке catch");
    }
}
```

В этом примере оператор `throw` выполняется не как часть этого метода, даже при том, что он находится внутри блока `try`. Компилятор выделит лямбда-функцию, вызываемую в обработчике события `Click`, в отдельный метод (как мы видели в главе 2), который выполнится за пределами блока `try` — в блоке `try` произойдет только подключение обработчика события `Click`. Код в лямбда-функции будет вызван кодом, запускающим событие `Click`, и исключение всплывет в том коде, а не в этом.

Методы `async` имеют аналогичное поведение, потому что, как мы видели в главе 3, вызов `await` эквивалентен вызову `ContinueWith`. Напишем для примера простой асинхронный метод, который генерирует исключение:

```
try
{
    await File.ReadAllBytesAsync("file.bin");
    throw new NotImplementedException();
}
```

```

catch
{
    Console.WriteLine("В блоке catch");
}

```

Этот код ждет завершения вызова `ReadAllBytesAsync` и затем генерирует исключение. Если преобразовать `await` в `ContinueWith`, то получится:

```

try
{
    File.ReadAllBytes("file.bin").ContinueWith(()=>
    {
        throw new NotImplementedException();
    });
}
catch
{
    Console.WriteLine("В блоке catch");
}

```

И этот код имеет ту же проблему, что и код в примере с обработчиком событий. Стока `throw` находится в лямбда-функции (которая передается в вызов `ContinueWith`), поэтому она выполняется за пределами блока `try`. По этой причине компилятор также продублирует `try-catch`, чтобы создать впечатление, что `await` работает без проблем с блоками `try-catch`:

```

try
{
    File.ReadAllBytesAsync("file.bin").ContinueWith(()=>
    {
        try
        {
            throw new NotImplementedException();
        }
        catch
        {
            Console.WriteLine("В блоке catch");
        }
    });
}
catch
{
    Console.WriteLine("В блоке catch");
}

```

Здесь компилятор знает, где находится инструкция `catch`, поэтому сможет выполнить все необходимые преобразования, чтобы заставить `try-catch` работать. Но что, если оператор `try` находится не в нашем методе, а в коде, который вызывает наш метод? В этом случае во время компиляции неизвестно, какую

инструкцию `catch` использовать, и компилятор не сможет скопировать ее в код продолжения. Давайте посмотрим, что происходит с простым асинхронным методом, который генерирует исключение:

```
public async Task<int> MyMethod()
{
    throw new NotImplementedException();
}
```

Это асинхронный метод, который просто генерирует исключение. Он преобразуется в такой код:

```
public Task MyMethod()
{
    throw new NotImplementedException();
}
```

Компилятор никак его не преобразовал! Напомню, что добавление спецификатора `async` не делает метод асинхронным. Это просто флаг для компилятора, чтобы тот включил обработку, необходимую для поддержки `await`. Если `await` не используется, то компилятор просто упаковывает возвращаемое значение в объект `Task`. Обратите внимание, что компилятору не потребовалось менять код, чтобы исключения вели себя как в синхронном методе. Вызов этого метода выбросит исключение подобно синхронному методу, чего мы и хотели.

Теперь рассмотрим метод, использующий `await`:

```
public async Task<int> MyMethod()
{
    await File.ReadAllTextAsync("file.bin");
    throw new NotImplementedException();
}
```

Этот метод ждет завершения вызова `ReadAllBytesAsync`, а затем выбрасывает исключение. В этом случае, чтобы передать сообщение об ошибке вызывающему коду, компилятор добавит блок `try-catch`, который перехватит исключение и сохранит его в возвращаемой задаче `Task`:

```
public async Task<int> MyMethod()
{
    var result = new TaskCompletionSource<int>();
    File.ReadAllTextAsync("file.bin").ContinueWith(t=>
    {
        try
        {
            throw new NotImplementedException();
        }
    });
    return result.Task;
}
```

```

        catch(Exception ex)
    {
        result.TrySetException(new AggregateException(ex));
    }
});

return result.Task;
}

```

Здесь компилятор добавил блок `try` внутри продолжения (код, который передается в `ContinueWith`). Обратите внимание, что, как и в предыдущем примере, компилятор не добавил блок `try` в код перед первым вызовом `await` и `ReadAllBytesAsync`.

Если ошибка возникнет до первого вызова `await`, то метод выбросит исключение как обычно. А если ошибка возникнет после первого вызова `await`, то исключение будет перехвачено кодом, который сгенерировал компилятор, и сохранено в возвращаемой задаче `Task`. Именно так работает большая часть асинхронного кода: асинхронный метод может выбрасывать исключение как обычно, а также сообщать об ошибке с помощью объекта `Task` (посредством значения `Faulted` в свойстве `Status` задачи `Task` и самого исключения в свойстве `Task.Exception`).

Если `await` задействуется при вызове метода, то обе ситуации выглядят одинаково. Но если `await` не используется (например, если вы собираете несколько задач и используете `Task.WhenAny` или `Task.WhenAll`), то вы должны обрабатывать и исключения, выбрасываемые асинхронными методами, и исключения, хранящиеся в возвращаемом объекте `Task`. Кроме того, не забывайте, что продолжение обычно запускается после возврата из метода, поэтому в случае ошибки задача `Task`, возвращаемая асинхронным методом, будет находиться в состоянии `Created`, `WaitingForActivation` или `Running`, и только позже она получит состояние `Faulted`.

Использование `await` для повторного выброса исключения и добавление свойства `Task.Exception` имеют единственное отличие, заключающееся в использовании `AggregateException`.

12.2. *await* и `AggregateException`

Свойство `Task.Exception` всегда хранит `AggregateException`. Класс `AggregateException`, как следует из названия, хранит внутри нескольких других исключений.

`Task` использует `AggregateException`, чтобы иметь возможность представлять результат нескольких операций, выполняемых параллельно (например, нескольких асинхронных операций, переданных в `Task.WhenAll`). Неудачей могут завершиться несколько фоновых операций, поэтому и нужен способ хранения более чем одного исключения.

На практике эта функция почти никогда не применяется. Фактически она встречается настолько редко, что если при использовании `await` задача `Task` потерпела неудачу, то `await` всегда будет повторно выбрасывать первое исключение из `AggregateException`, а не само исключение `AggregateException`. Если возникнет несколько исключений, то `await` все равно выбросит только первое исключение из `AggregateException` и проигнорирует остальные. Все исключения, кроме первого, вместе с информацией, хранящейся внутри них, будут потеряны. Вот код, показывающий, как `await` выбрасывает сохраненное исключение:

```
var tcs = new TaskCompletionSource();
tcs.SetException(new NotSupportedException());
Console.WriteLine("В задачах: " + tcs.Task.Exception.GetType()); ← AggregateException
try
{
    await tcs.Task; ← Выбрасывает внутреннее
    исключение
}
catch(Exception ex)
{
    Console.WriteLine("Выброшено: " + ex.GetType()); ← NotSupportedException
}
```

Эта программа сохраняет `NotSupportedException` в задаче `Task`, используя `TaskCompletionSource` (мы уже говорили о создании собственных задач с помощью `TaskCompletionSource` в главе 10). Затем она проверяет тип исключения, хранящегося внутри `Task`, и получает экземпляр `AggregateException`, обертывающий фактическое исключение. Однако затем она использует `await`, и поскольку `Task` находится в состоянии `Failed`, вызов `await` выбрасывает исключение, но это будет внутреннее исключение `NotSupportedException`, а не `AggregateException`.

12.3. Потеря исключений

Мы видели, что компилятор генерирует код для перехвата исключений и прячет их внутри `Task`, а `await` повторно выбрасывает это исключение. Но что произойдет, если по какой-то причине в коде не используется `await`?

Ответ — ничего. Код, сгенерированный компилятором, перехватит исключение и сохранит его в `Task`. И все. Если код не прочитает исключение из `Task` (используя `await` или обратившись к свойству `Task.Exception`), то исключение будет проигнорировано.

Рассмотрим следующий фрагмент кода:

```
public async Task MethodThatThrowsException()
{
    await Task.Delay(100);
    throw new NotImplementedException();
}
```

```
public async Task MethodThatCallsOtherMethod()
{
    MethodThatThrowsException();
}
```

Здесь мы имеем два метода. Первый метод, `MethodThatThrowsException`, выбрасывает исключение после `await`, которое компилятор перехватит и сохранит в возвращаемой задаче `Task`. Второй метод вызывает первый, но когда я его писал, то забыл добавить вызов `await` и не предусмотрел проверку возвращаемой задачи `Task`. Исключение будет перехвачено в первом методе кодом, сгенерированным компилятором, но проигнорировано вторым методом, потому что я не использовал `await`. Поэтому среда выполнения подумает, что мы обработали исключение (так как код, сгенерированный компилятором, все-таки перехватил его), и код продолжит выполняться, игнорируя ошибку.

Если метод, который генерирует исключение, находится в библиотеке и в отладчике включена функция «только мой код» (just my code), то вы не увидите исключения даже в отладчике. Так что, если какой-то код в вашей программе вдруг перестает работать без видимых причин, высока вероятность, что кто-то где-то забыл добавить вызов `await`.

12.4. Исключения и методы `async void`

Методы с возвращаемым значением типа `async void` не возвращают `Task` (это очевидно). Поскольку объект `Task`, в котором можно было бы сохранить исключение, отсутствует, компилятор не знает, как поступить с исключением, сгенерированном внутри метода, и поэтому не генерирует блок `try-catch`, который мы видели в предыдущем примере. В результате любое исключение, сгенерированное в коде, поднимется по стеку вызовов в контекст `SynchronizationContext`, который запустил код (о том, как работает `SynchronizationContext`, рассказывается в главе 11). Это, скорее всего, вызовет сбой программы. Поэтому всегда обрабатывайте все исключения в методах `async void` самостоятельно и никогда не позволяйте исключениям выходить за их пределы.

Итоги главы

- Исключения используют стек вызовов для поиска соответствующей инструкции `catch`, которую можно выполнить в случае ошибки, что является проблемой для асинхронного кода, потому что продолжения выполняются не в том же стеке вызовов, что и код, вызывающий асинхронный метод.
- Если вы используете `async/await`, то компилятор генерирует код, который будет выглядеть так же, как синхронный код. Этот код перехватит исключение внутри асинхронного метода и сохранит его в возвращаемой задаче `Task`.

Затем `await` выбросит исключение внутри продолжения, создав иллюзию, что оно сгенерировано инструкцией `await`.

- Каждый асинхронный метод может выбросить обычное исключение или сообщить о сбое, используя возвращаемую задачу `Task`. Применение `await` стирает различия между этими двумя режимами сбоя. Если вы не используете `await`, то должны обрабатывать оба случая самостоятельно.
- Для хранения исключений внутри `Task` используется класс `AggregateException` на тот случай, если `Task` представляет несколько асинхронных операций. Инструкция `await` игнорирует все исключения внутри `AggregateException`, кроме первого. Если вы не используете `await`, то должны обработать эту ситуацию самостоятельно. В редких случаях `Task` представляет несколько операций, и чтобы обработать несколько возможных сбоев, вы должны не полагаться на `await`, а проанализировать свойство `Task.Exception` самостоятельно.
- Если вы проигнорируете задачу `Task`, возвращаемую асинхронным методом (например, забыв использовать `await`), и внутри кода задачи возникнет исключение, то оно будет потеряно.
- Как следствие, если в коде происходит сбой и при этом не возникает исключений, то высока вероятность, что вы где-то забыли добавить `await`.
- Поддержка исключений, предлагаемая инструкциями `async/await`, основана на возвращаемой задаче `Task`. Методы `async void` не возвращают задачу `Task` и поэтому лишены этой поддержки. Никогда не выбрасывайте исключение в методах `async void` и не позволяйте исключениям всплывать из них.

13

Потокобезопасные коллекции

В этой главе

- ✓ Проблемы, возникающие при использовании обычных коллекций в многопоточной среде.
- ✓ Потокобезопасные (Concurrent) коллекции.
- ✓ Класс `BlockingCollection`.
- ✓ Асинхронные альтернативы `BlockingCollection`.
- ✓ Неизменяемые (Immutable) коллекции и особенности их использования.
- ✓ Замороженные (Frozen) коллекции.

Пространство имен `System.Collections.Generic` предоставляет много полезных коллекций, однако их нельзя просто использовать в многопоточном приложении, потому что ни одна из них не является потокобезопасной. В этой главе мы рассмотрим проблемы, связанные с самым простым подходом, помогающим сделать эти коллекции потокобезопасными, — обрачиванием в `lock` любого доступа к ним, а также поговорим о потокобезопасных альтернативах, предоставляемых библиотекой .NET.

В частности, мы рассмотрим потокобезопасные коллекции, добавленные в .NET Framework 4, обсудим неизменяемые коллекции, добавленные в .NET Core (который является основой для .NET 5 и более поздних версий), и поговорим о замороженных коллекциях, добавленных в .NET 8. Вы также узнаете, как и когда использовать коллекции каждого типа. Но сначала давайте поговорим о том, почему нельзя просто использовать обычные коллекции в многопоточном приложении.

13.1. Проблемы использования обычных коллекций

Библиотека .NET предоставляет множество удобных классов коллекций в пространстве имен `System.Collections.Generic`. Эти коллекции поддерживают множественное параллельное чтение, но могут быть повреждены и, как следствие, могут возвращать неожиданные результаты в случае одновременной записи из нескольких потоков или когда одни потоки их читают, а другие записывают.

Кроме того, согласно официальной документации, итерации по коллекциям изначально не являются потокобезопасными, поэтому, приступая к итерациям по коллекции с помощью оператора цикла, такого как `foreach`, или `Linq`-выражения, необходимо предотвратить любые возможности записи в эти коллекции из других потоков на протяжении всего цикла. Чтобы использовать эти коллекции в многопоточной программе, вам следует позаботиться о синхронизации потоков, например, с помощью блокировок, но сделать это правильно часто довольно сложно.

Например, рассмотрим очень распространенный случай. Представьте, что мы используем `Dictionary< TKey, TValue >` в качестве кэша. Когда нам нужно получить какой-то элемент данных, мы сначала проверяем, есть ли он в кэше. Если его там нет, мы создаем и инициализируем этот элемент, возможно обращаясь к внешнему сервису или выполняя какие-то вычисления (причина, почему решено использовать кэш, заключается в том, что инициализация элемента занимает много времени). Начнем с однопоточного кода, а позже добавим блокировку (листинг 13.1).

Листинг 13.1. Простой, непотокобезопасный кэш

```
if(!dictionary.TryGetValue(itemId, out var item))
{
    item = CreateAndInitializeItem(itemId);
    dictionary.Add(itemId, item);
}
```

Этот код пытается получить элемент из словаря, и если искомый элемент отсутствует, то вызывает `CreateAndInitializeItem` для создания и инициализации

элемента. После создания элемента вызывается метод `Add` для добавления элемента в словарь, чтобы он был доступен в следующий раз, когда он нам понадобится.

Это очень даже неплохой способ реализовать простой кэш внутри однопоточного приложения, но этот код совсем не потокобезопасный. Вызов `TryGetValue` из нескольких потоков одновременно, очевидно, не приведет ни к чему плохому, но одновременный вызов `Add` или вызов `Add` и `TryGetValue` может привести к неожиданным результатам или даже повредить словарь.

Давайте сделаем этот код потокобезопасным. Начнем с самого простого варианта — поместим весь блок в конструкцию `lock` (листинг 13.2).

Листинг 13.2. Потокобезопасный кэш с одной блокировкой

```
Item item;
lock(_dictLock) ← Добавлена блокировка
{
    if(!dictionary.TryGetValue(itemId, out item))
    {
        item = CreateAndInitializeItem(itemId);
        dictionary.Add(itemId, item);
    }
}
```

Это почти тот же самый код, что и в листинге 13.1, он отличается лишь использованием блокировки с помощью конструкции `lock`, чтобы предотвратить его одновременное выполнение несколькими потоками. Это делает наш код потокобезопасным, но ценой блокировки всего кэша при каждом его вызове. Если искомый элемент уже присутствует в словаре, то блокировка будет кратковременной и все будет хорошо. Однако если понадобится создать новый элемент, то весь кэш будет заблокирован на протяжении всей инициализации. Это означает, что другие потоки, работающие с совершенно разными элементами, будут ждать каждый раз, когда потребуется инициализировать новый элемент. Чтобы решить эту проблему, нужно снять блокировку на время инициализации элемента (листинг 13.3).

В этом коде есть два блока `lock`: один защищает вызов `TryGetValue`, а другой — вызов `Add`. Эти блокировки гарантируют, что обращение к `Dictionary<TKey, TValue>` никогда не произойдет одновременно из нескольких потоков, а значит, словарь не повредится. К сожалению, это не делает наш код потокобезопасным. На самом деле этот код почти гарантированно потерпит неудачу, если нескольким потокам потребуется один и тот же элемент, которого еще нет в кэше.

Это распространенная проблема, когда комбинация двух (или более) потокобезопасных операций далеко не всегда делает код потокобезопасным. Вызов

`TryGetValue` теперь безопасен, поскольку защищен блокировкой, и вызов `Add` тоже безопасен по той же причине. Но поскольку блокировка удерживается не на всем протяжении выполнения кода, другие потоки могут изменить словарь между `TryGetValue` и `Add`.

Листинг 13.3. Непотокобезопасный кэш со снятием блокировки на время инициализации

```
Item item;
bool exists;
lock(_dictLock) ← Устанавливает блокировку
{
    exists = dictionary.TryGetValue(itemId, out item);
} ← Снимает блокировку
if(!exists)
{
    item = CreateAndInitializeItem(itemId);
    lock(_dictLock) ← Снова устанавливает блокировку
    {
        dictionary.Add(itemId, item); ← Add — терпит неудачу, если данный элемент
        уже был добавлен другим потоком
    }
}
```

Если два потока одновременно вызовут этот код, чтобы получить один и тот же элемент, то первый поток выполнит `TryGetValue` и обнаружит, что элемента нет в словаре, поэтому он перейдет к созданию и инициализации элемента. Второй поток тем временем также вызовет `TryGetValue` (поскольку инициализация элемента занимает много времени — мы используем кэш). Так как первый поток еще не добавил элемент, второй поток будет вынужден начать создавать еще одну копию того же объекта.

Теперь у нас есть два потока, одновременно занятых инициализацией двух разных копий объекта `Item`, представляющих один и тот же логический элемент. Один из потоков завершит работу первым и добавит элемент в кэш, вызвав `Add`. Другой поток тоже в какой-то момент завершит инициализацию своей копии и попытается добавить ее в кэш, вызвав `Add`. Однако, поскольку первый поток уже добавил элемент, вызов `Add` во втором потоке завершится исключением `ArgumentException`. Схема на рис. 13.1 иллюстрирует работу этих двух потоков.

Чтобы сделать этот код потокобезопасным, следует пропустить вызов `Add`, если другой поток уже добавил элемент, пока текущий поток был занят его инициализацией. Проще всего заменить вызов `Add` оператором `[]` (технически это свойство `Item[]`, но я буду называть его оператором, а не свойством, потому что он используется как оператор). Оператор `[]` добавляет новый элемент в словарь, если указанный ключ отсутствует, и перезаписывает значение, если элемент с таким ключом уже существует. Этот прием решает нашу проблему с исключением, но вносит новую малозаметную ошибку (листинг 13.4).

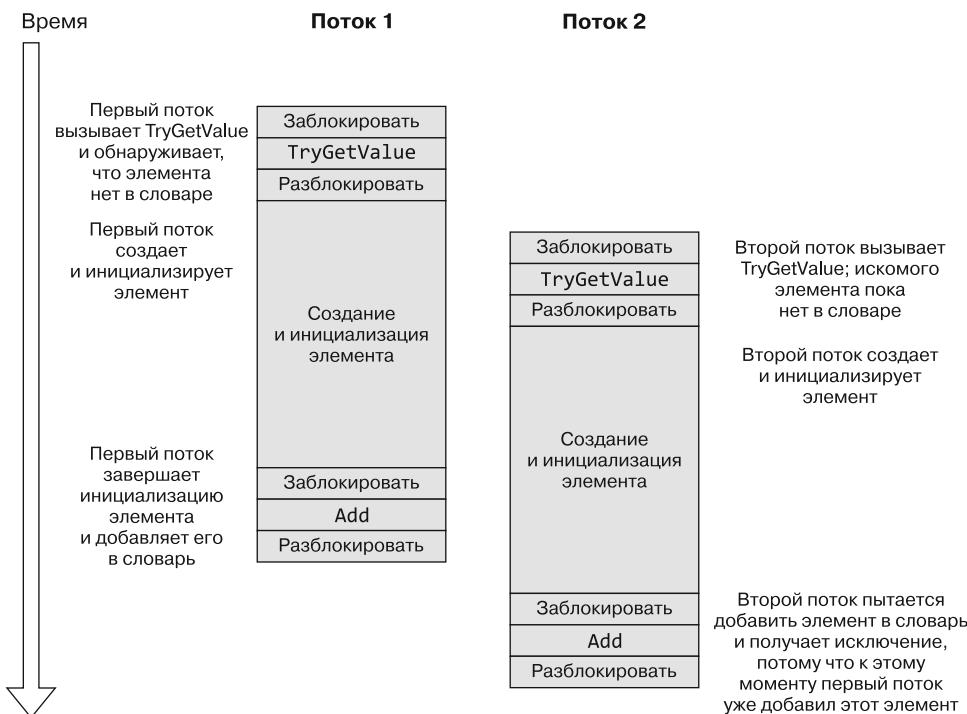


Рис. 13.1. Одновременная инициализация одного и того же элемента двумя потоками. Второй поток терпит неудачу, так как первый поток уже добавил элемент

Листинг 13.4. Потокобезопасный кэш с оператором []

```
Item item;
bool exists;
lock(_dictLock) ← Устанавливает блокировку
{
    exists = dictionary.TryGetValue(itemId, out item);
} ← Снимает блокировку
if(!exists)
{
    item = CreateAndInitializeItem(itemId);
    lock(_dictLock) ← Снова устанавливает блокировку
    {
        dictionary[itemId] = item; ← Перезаписывает изменения,
    }                                     внесенные другим потоком
}
```

В этом коде я просто заменил `dictionary.Add(itemId, item)` на `dictionary[itemId] = item`, и это решило нашу текущую проблему, поскольку оператор `[]` просто перезаписывает предыдущее значение, если оно существует. Однако

это решение может привести к ошибке. Теперь первый поток использует свою копию объекта `Item`, а второй и последующие потоки — копию, созданную вторым потоком. Если объект `Item` является неизменяемым, а инициализация всегда возвращает эквивалентный объект для одного и того же ключа, то такое решение допустимо. Но если объект `Item` изменяется первым потоком, эти изменения будут потеряны.

Чтобы гарантировать, что все потоки используют один и тот же объект `Item`, нам не остается ничего иного, кроме как перепроверить словарь после инициализации. Если другой поток уже обновил словарь, мы должны использовать значение из первого потока (листинг 13.5).

Листинг 13.5. Потокобезопасный кэш со снятием блокировки на время инициализации

```
Item item;
bool exists;
lock(_dictLock) ← Устанавливает блокировку
{
    exists = dictionary.TryGetValue(itemId, out item);
} ← Снимает
if(!exists) ← блокировку
{
    item = CreateAndInitializeItem(itemId);
    lock(_dictLock) ← Снова устанавливает блокировку
    {
        if(dictionary.TryGetValue(itemId, out var itemFromOtherThread)) | Проверяет, был ли добавлен
            | элемент другим потоком
        {
            item = itemFromOtherThread;
        }
        else
        {
            dictionary.Add(itemId, item);
        }
    }
}
```

Здесь после инициализации элемента мы еще раз проверяем словарь, вместо того чтобы бездумно вызывать метод `Add` или оператор `[]`. Если мы обнаружили, что элемент уже имеется в словаре, мы игнорируем проинициализированную копию и используем ту, что находится в словаре (так как хотим, чтобы все потоки работали с одним и тем же объектом). Добавление элемента происходит, только если его по-прежнему нет в словаре.

Как видите, код получился довольно сложным и трудным для понимания по сравнению с однопоточной версией в листинге 13.1. К счастью, в таком коде нет необходимости, потому что у нас есть потокобезопасные коллекции.

13.2. Потокобезопасные коллекции

В пространстве имен `System.Collections.Concurrent` представлены потокобезопасные версии наиболее популярных коллекций. Эти коллекции используют специализированные стратегии блокировки и приемы выполнения операций без блокировки, чтобы оставаться потокобезопасными, даже когда к коллекции одновременно обращается несколько потоков. Это означает, что все методы и свойства потокобезопасных коллекций можно одновременно использовать в разных потоках, не вызывая повреждений или неожиданного поведения.

Интерфейс потокобезопасных коллекций отличается от интерфейса обычных коллекций из пространства имен `System.Collections.Generic`. Как мы уже видели ранее в этой главе, простая реализация возможности вызова методов обычной коллекции из нескольких потоков сама по себе не делает код потокобезопасным.

В этом разделе мы рассмотрим часто используемые потокобезопасные коллекции и начнем со словаря `ConcurrentDictionary< TKey, TValue >`, который элегантно решает проблемы, описанные выше.

13.2.1. `ConcurrentDictionary< TKey, TValue >`

Как вы наверняка догадались, `ConcurrentDictionary< TKey, TValue >` — это потокобезопасная альтернатива для `Dictionary< TKey, TValue >`. Однако, учитывая проблемы, которые мы наблюдали, когда пытались использовать блокировки с `Dictionary< TKey, TValue >`, у него немного другой интерфейс, ориентированный на решение проблем в многопоточной среде. Давайте вернемся к исходному однопоточному коду, использующему кэш на основе `Dictionary< TKey, TValue >` (листинг 13.6).

Листинг 13.6. Снова непотокобезопасный кэш

```
if(!dictionary.TryGetValue(itemId, out var item))
{
    item = CreateAndInitializeItem(itemId);
    dictionary.Add(itemId, item);
}
```

Это тот же самый код, что и в листинге 13.1. Он проверяет присутствие искомого элемента в кэше, и если его там нет, то создает и инициализирует новый объект элемента и добавляет его в кэш.

Теперь просто заменим `Dictionary< TKey, TValue >` на `ConcurrentDictionary< TKey, TValue >`. Класс `ConcurrentDictionary< TKey, TValue >` имеет метод

`TryGetValue`, который работает аналогично одноименному методу в `Dictionary<TKey, TValue>`. Если указанный ключ присутствует в словаре, то он возвращает `true` и помещает значение в выходной (`out`) параметр. Если искомый ключ отсутствует, то возвращает `false`.

Но `ConcurrentDictionary<TKey, TValue>` не имеет метода `Add`. Как мы видели в листинге 13.3, в многопоточном коде всегда есть вероятность, что другой поток добавит элемент прямо перед тем, как текущий поток вызовет `Add` и тем самым выбросит исключение. По этой причине метод `Add` в `ConcurrentDictionary<TKey, TValue>` был заменен методом `TryAdd`.

Как мы видели, каким бы коротким ни был временной интервал между вызовами `TryGetValue` и `Add`, всегда есть вероятность, что другой поток успеет вклиниваться в этот промежуток и добавить элемент. Чтобы решить эту проблему, мы должны рассматривать случай присутствия ключа в словаре не как исключительную ситуацию, а как нормальное явление. Именно так и работает метод `TryAdd`.

Это различие между `Add` и `TryAdd` проявляется в небольшом изменении интерфейса метода. Там, где `Add` выбросит исключение, если ключ уже существует в словаре, `TryAdd` вернет `false`.

Замена `Add` на `TryAdd` в коде из листинга 13.6 дает простой и потокобезопасный код. Но он имеет небольшую проблему (листинг 13.7). (Подсказка: это та же проблема, которую мы наблюдали в листинге 13.4.)

Листинг 13.7. Потокобезопасный кэш с `ConcurrentDictionary.TryAdd`

```
if(!dictionary.TryGetValue(itemId, out var item))
{
    item = CreateAndInitializeItem(itemId);
    dictionary.TryAdd(itemId, item);
}
```

В этом листинге вместо `Add` вызывается `TryAdd`. И поскольку `TryAdd` не генерирует исключения, когда ключ уже имеется в словаре, этот код является потокобезопасным и надежным. Но он имеет разновидность проблемы, которую мы наблюдали в листинге 13.4. Если несколько потоков инициализируют один и тот же элемент одновременно, то каждый будет работать со своей копией объекта `Item`. Если объект `Item` не является неизменяемым или все копии неидентичны, то это может стать настоящей проблемой.

На самом деле было бы желательно иметь возможность объединить `TryGetValue` и `TryAdd` в одну атомарную операцию, чтобы устраниТЬ эту проблему. `ConcurrentDictionary<TKey, TValue>` дает такую возможность в виде метода `GetOrAdd` (листинг 13.8).

Листинг 13.8. Потокобезопасный кэш с ConcurrentDictionary.GetOrAdd

```
var item = dictionary.GetOrAdd(itemId, CreateAndInitializeItem);
```

Эта единственная строка, вызывающая `GetOrAdd`, эквивалентна 21-й строке кода в листинге 13.5. Если элемент уже есть в словаре, то вызов `GetOrAdd` вернет его. Если элемента нет, вызывается `CreateAndInitialize` для создания нового элемента. Если несколько потоков вызовут `GetOrAdd` до того, как элемент будет создан и добавлен в словарь, все они запустят `CreateAndInitialize`, но `GetOrAdd` во всех потоках вернет один и тот же объект.

`GetOrAdd` имеет версию, принимающую во втором параметре значение для добавления в словарь, а также версию, которую мы только что использовали, — она принимает делегат, создающий значение. `ConcurrentDictionary< TKey, TValue >` использует хорошо продуманную блокировку, поэтому вызовы `GetOrAdd` с разными ключами могут выполняться одновременно.

Обратите внимание, что при одновременном вызове из нескольких потоков версии `GetOrAdd`, которая принимает делегат, код создания элемента может быть запущен несколько раз. Поток, завершивший работу первым, добавит значение в словарь, а элементы, созданные другими потоками, будут проигнорированы. Вам следует учитывать, требует ли ваш объект освобождения ресурсов и возможны ли многократные выполнения кода инициализации.

Класс `ConcurrentDictionary< TKey, TValue >` также имеет метод `TryRemove`, который удалит ключ со значением и вернет `true`, если ключ существовал и был удален. Если ключ отсутствовал, то он вернет `false`. Метод `TryRemove`, заменивший метод `Remove`, решает ту же проблему состояний гонки, которую мы видели, когда говорили об `Add` и `TryAdd`.

Наконец, `ConcurrentDictionary< TKey, TValue >` имеет метод `TryUpdate`. Он решает проблему, когда один поток может перезаписать данные, внесенные другим потоком. Для демонстрации этой проблемы напишем метод, увеличивающий значение в словаре (листинг 13.9).

Листинг 13.9. Непотокобезопасное увеличение значения

```
private ConcurrentDictionary<string, int> _dictionary = new();

public void Increment(string key)
{
    int prevValue = _dictionary[key];
    _dictionary[key] = prevValue+1;
}
```

Этот метод читает значение, связанное с указанным ключом, увеличивает это значение на единицу и записывает новое значение обратно в словарь.

В этом коде есть ошибка, порождающая состояние гонки. Допустим, в настоящий момент ключу соответствует значение 1 и мы вызываем `Increment` одновременно из двух разных потоков. Ожидаемый результат равен 3 (исходное значение 1 было увеличено дважды), но если нам не повезет, то мы можем получить такую последовательность.

1. Поток 1 читает значение и получает 1.
2. Поток 2 читает значение, и, поскольку первый поток еще не записал новое значение, он также получает 1.
3. Поток 1 увеличивает и сохраняет значение; теперь в словаре хранится значение 2.
4. Поток 2 увеличивает и сохраняет значение; в словаре по-прежнему хранится значение 2.

Чтобы решить эту проблему, нужно заключить всю операцию в блокировку (`lock`) или хотя бы иметь возможность обнаружить проблему, чтобы ее исправить. Именно для таких случаев предназначен метод `TryUpdate` (листинг 13.10).

Листинг 13.10. Потокобезопасное увеличение с `ConcurrentDictionary.TryUpdate`

```
private ConcurrentDictionary<string, int> _dictionary = new();

public void Increment(string key)
{
    while(true)
    {
        int prevValue = _dictionary[key];
        if(_dictionary.TryUpdate(key, prevValue+1, prevValue))
            break;
    }
}
```

Теперь метод входит в цикл и сохраняет текущее значение в переменную `prevValue`. Затем вызывает `TryUpdate` с новым значением (`prevValue+1`) и старым значением (`prevValue`). Если текущее значение в словаре все еще равно `prevValue`, то `TryUpdate` обновит значение и вернет `true`. Это заставит наш код выйти из цикла. Но если какой-то другой поток успел изменить значение в словаре, то `TryUpdate` оставит словарь неизменным и вернет `false`, что заставит наш код повторить попытку увеличения значения. Цикл будет продолжаться до тех пор, пока операция не завершится успешно.

`ConcurrentDictionary<TKey, TValue>` не имеет асинхронных интерфейсов, но блокирует выполнение на очень короткое время и отлично работает в асинхронном коде.

13.2.2. **BlockingCollection**<T>

Класс **BlockingCollection**<T> добавляет блокирующие операции к другой коллекции и реализует паттерн «производитель — потребитель». По сути, он добавляет возможность дождаться, когда элемент коллекции станет доступным. В роли базовой коллекции для **BlockingCollection**<T> можно использовать **ConcurrentQueue**<T>, **ConcurrentStack**<T> и **ConcurrentBag**<T> (подробнее о них рассказывается в следующем разделе).

По умолчанию и намного чаще других используется **ConcurrentQueue**<T>. Коллекция **BlockingCollection**<T>, основанная на **ConcurrentQueue**<T>, сохраняет порядок элементов как в обычной очереди: первый вошедший элемент будет первым выходящим. Мы уже видели в главе 8, как это может служить основой для очень простой и эффективной очереди заданий.

Второй вариант — **ConcurrentStack**<T>. Коллекция **BlockingCollection**<T>, основанная на **ConcurrentStack**<T>, действует подобно стеку — последний добавленный элемент извлекается первым. Это полезно, когда у вас есть многопоточный алгоритм, требующий потокобезопасного стека, где потребитель ожидает, пока другой поток добавит элементы, если стек пуст.

Последний тип, **ConcurrentBag**<T>, используется очень редко. **ConcurrentBag**<T> — это специализированная коллекция, оптимизированная для сценариев, когда один и тот же поток и читает значения из коллекции, и записывает их в нее (подробнее об этой коллекции я расскажу далее в этой главе).

Помимо возможности ожидания новых элементов, **BlockingCollection**<T> также позволяет указать максимальный размер коллекции, что может пригодиться в случаях, когда скорость работы производителя намного превышает скорость работы потребителя. Эта возможность называется *ограниченной емкостью* (*bounded capacity*).

Чаще всего **BlockingCollection**<T> используется в роли очереди заданий (как в нашем примере с очередью заданий в главе 8, где мы написали реализацию очереди с одним фоновым потоком). Давайте вернемся к коду из главы 8 и реализуем очередь на основе **BlockingCollection**<T> с несколькими потоками-потребителями (листинг 13.11).

Этот код создает **BlockingCollection**<int> для хранения данных, которые нужно обработать в фоновом режиме. Затем запускает десять фоновых потоков, каждый из которых использует цикл **foreach** и метод **GetConsumingEnumerable** для обработки элементов. Имитируя обработку, потоки просто приостанавливаются на некоторое случайное время и выводят число. В очередь помещаются числа от 0 до 99, которые играют роль обрабатываемых данных.

Запустив этот код, вы увидите, что он обрабатывает каждый элемент данных ровно один раз, причем большинство элементов обрабатывается по порядку.

Причина, почему некоторые элементы обрабатываются не по порядку, хотя на самом деле `BlockingCollection<T>` возвращает элементы по порядку, заключается в синхронизации: иногда потоки могут обгонять друг друга, создавая иллюзию, что два элемента поменялись местами.

Листинг 13.11. BlockingCollection с 10 обрабатывающими потоками

```
BlockingCollection<int> blockingCollection = new BlockingCollection<int>();
Thread[] workers = new Thread[10];
for(int i=0; i<workers.Length; i++) ← Создает 10 обрабатывающих потоков
{
    workers[i] = new Thread(threadNumber =>
    {
        var rng = new Random((int)threadNumber);
        int count = 0;
        foreach (var currentValue in
            blockingCollection.GetConsumingEnumerable())
        {
            Console.WriteLine($"поток {threadNumber} значение {currentValue}");
            Thread.Sleep(rng.Next(500));
            count++;
        }
        Console.WriteLine($"поток {threadNumber}, всего {count} элементов");
    });
    workers[i].Start(i);
}
for(int i=0;i<100;i++) ← Добавляет 100 элементов
{                               для обработки
    blockingCollection.Add(i);
}
blockingCollection.CompleteAdding(); ← Уведомляет, что элементов
foreach (var currentThread in workers) ← больше не будет
    currentThread.Join();           ← Ожидает завершения всех потоков
```

Ограниченнная емкость, упомянутая ранее, в основном реализуется методом `Add`, который добавляет элемент в коллекцию. Если коллекция достигла максимальной емкости, он блокируется до тех пор, пока какой-то другой поток не извлечет элемент из коллекции. Метод `TryAdd` работает аналогично, но добавляет тайм-аут (который может быть равен нулю). Если коллекция достигла максимальной емкости, то вызов метода заблокируется, пока какой-то другой поток не извлечет элемент из коллекции или пока не истечет тайм-аут. По истечении тайм-аута `TryAdd` завершается неудачей и возвращает `false`. Если тайм-аут равен нулю, `TryAdd` возвращает управление немедленно.

Метод `Take` возвращает следующий элемент в коллекции и удаляет его в одной потокобезопасной операции. Следующим элементом будет самый старый элемент в коллекции, если она основана на `ConcurrentQueue<T>`, самый новый элемент, если она основана на `ConcurrentStack<T>`, или любой из элементов, если

коллекция создана на основе `ConcurrentBag<T>`. Если коллекция пуста, то вызов `Take` будет блокироваться, пока другой поток не добавит элемент вызовом `Add` или `TryAdd`. `TryTake` (подобно `TryAdd`) работает аналогично `Take`, но с добавлением тайм-аута. Если коллекция пуста и тайм-аут истечет до того, как в коллекции появится элемент, `TryTake` завершится неудачей и вернет `false`. Если тайм-аут равен нулю, то `TryTake` вернет управление немедленно.

Наиболее распространенный способ чтения данных из `BlockingCollection<T>` — цикл `foreach` в сочетании с вызовом метода `GetConsumingEnumerable`, как было показано в главе 8, вместо прямого вызова `Take` или `TryTake`. Метод `GetConsumingEnumerable` возвращает экземпляр `IEnumerable<T>`, который при использовании с `foreach` извлекает из коллекции текущий элемент в каждой итерации и, если коллекция пуста, блокируется до тех пор, пока другой поток не добавит элемент в коллекцию. Это, по сути, эквивалентно вызову `Take` в начале каждой итерации цикла.

При использовании `GetConsumingEnumerable` и `foreach` нам нужна возможность сообщить, что новых элементов больше не будет и можно выйти из цикла. Такую возможность реализует `CompleteAdding`. После вызова `CompleteAdding` цикл `foreach` продолжит обработку всех оставшихся элементов в коллекции, а затем завершится. Вызов `Add` или `TryAdd` после `CompleteAdding` выбросит исключение `InvalidOperationException`.

`BlockingCollection<T>` также содержит статические методы `AddToAny`, `TryAddToAny`, `TakeFromAny` и `TryTakeFromAny`. Они действуют подобно своим нестатическим аналогам, за исключением того, что принимают массив коллекций `BlockingCollection<T>` и используют одну из них в зависимости от количества элементов в каждой коллекции. Может показаться, что они предлагают хороший способ построить систему с несколькими потоками-потребителями, в которой у каждого потока есть своя коллекция `BlockingCollection<T>`, но на самом деле это не так.

`AddToAny` и `TryAddToAny` никак не балансируют нагрузку. Они оптимизированы для максимально быстрого выполнения операции `AddToAny`, поэтому всегда будут искать самый быстрый вариант добавления элемента в коллекцию. В большинстве случаев они просто добавляют элемент в первую коллекцию, не заполненную до максимальной емкости. Поэтому `AddToAny` и `TryAddToAny` будут преимущественно добавлять элементы в одну и ту же коллекцию `BlockingCollection<T>`. Если вы решите использовать их для построения системы с несколькими обрабатывающими потоками, то один из ваших потоков получит большую часть работы, а остальные будут в основном простаивать.

`BlockingCollection<T>` очень полезна, если вы управляете своими потоками, но, как следует из названия, она использует блокирующие операции и потому не соответствует модели асинхронного программирования.

13.2.3. Асинхронные альтернативы *BlockingCollection*

По состоянию на .NET версии 8 стандартная библиотека .NET не имеет асинхронных коллекций в целом и асинхронной версии *BlockingCollection*<T> в частности. Однако имеет несколько других классов, которые можно превратить в асинхронную очередь. Один из них — *Channel*<T>, потокобезопасная очередь с несколькими производителями и потребителями, разработанная для связи между программными компонентами.

Класс *Channel*<T> — это коммуникационный канал. У каждого канала есть писатель, который может добавлять сообщения, и читатель, который может извлекать сообщения из канала. Канал сохраняет порядок сообщений, что делает его эквивалентом очереди. И писатель, и читатель явно поддерживают одновременный доступ.

Код в листинге 13.11 можно изменить, использовав в нем *Channel*<T> вместо *BlockingCollection*<T>, как показано в листинге 13.12.

Листинг 13.12. Асинхронная фоновая обработка с *Channel*<T>

```
var ch = Channel.CreateUnbounded<int>();
Task[] tasks = new Task[10];
for(int i=0; i<10; ++i)    ← Запускает 10 асинхронных задач
{
    var threadNumber = i;
    tasks[i] = Task.Run(async () =>
    {
        var rng = new Random((int)threadNumber);
        int count = 0;
        while (true)
        {
            try
            {
                var currentValue = await ch.Reader.ReadAsync(); ← Ждет появления
                Console.WriteLine($"задача {threadNumber} значение {currentValue}");
                Thread.Sleep(rng.Next(500));
                count++;
            }
            catch(ChannelClosedException) ← Это исключение означает,
            {                                что данных больше не будет
                break;
            }
        }
        Console.WriteLine($"задача {threadNumber}, всего {count} элементов");
    });
}
for (int i = 0; i < 100; i++) ← Добавляет 100 элементов
                           для обработки
```

```

{
    await ch.Writer.WriteAsync(i);
}
ch.Writer.Complete();   ← Сообщает, что данных больше не будет
Task.WaitAll(tasks);  ← Ждет завершения всех задач

```

Код в листинге 13.12 создает экземпляр `Channel<T>` вместо `BlockingCollection<T>` и вместо создания потока использует `Task.Run`. Код, передаваемый в `Task.Run`, запускается в пуле потоков и тут же вызывает `await`, чтобы освободить поток. Этот шаг можно пропустить, использовав хитроумный трюк с `ContinueWith`, но это усложнит код.

Самое большое отличие от листинга 13.11 в том, что вместо `foreach` используется `while(true)` и перехватывается исключение, сообщающее о том, что можно выйти из цикла. В следующей главе я покажу, что еще можно сделать в этом случае.

13.2.4. `ConcurrentQueue<T>` и `ConcurrentStack<T>`

`ConcurrentQueue<T>` — это потокобезопасная версия `Queue<T>`, а `ConcurrentStack<T>` — потокобезопасная версия `Stack<T>`. `ConcurrentQueue<T>` — это структура данных FIFO (first in, first out — первым пришел, первым вышел), то есть, извлекая следующий элемент, вы всегда получите самый старый элемент в очереди. `ConcurrentStack<T>` — это структура данных LIFO (last in, first out — последним пришел, первым вышел), то есть следующий элемент всегда будет самым последним из добавленных.

`ConcurrentQueue<T>` и `ConcurrentStack<T>` предлагают те же методы для добавления элементов, что и их непотокобезопасные аналоги (`Enqueue` в `ConcurrentQueue<T>` и `Push` в `ConcurrentStack<T>`), и те же методы для извлечения следующего элемента (`TryDequeue` и `TryPop` соответственно). Отличия в интерфейсе от непотокобезопасной версии обусловлены теми же причинами, которые мы обсуждали, когда рассматривали `ConcurrentDictionary< TKey, TValue >`. Если бы у нас была потокобезопасная версия с тем же интерфейсом, что и `Queue<T>`, то нам нужно было бы написать такой код:

```

var queue = new Queue<int>();
// ...
if(queue.Count > 0)   ← В этом месте другой поток может успеть
{                     раньше извлечь последний элемент
    var next = queue.Dequeue();
    // использовать next
}

```

Этот код проверяет, есть ли элементы в очереди, а затем извлекает следующий элемент, однако в многопоточном коде другой поток может успеть извлечь

последний элемент из очереди между проверкой и попыткой извлечения в текущем потоке. Это означает, что, даже имея потокобезопасный класс с тем же интерфейсом, что и `Queue<T>`, нам было бы сложно использовать его для написания потокобезопасного кода. Напротив, с интерфейсом `ConcurrentQueue<T>` мы пишем код следующим образом:

```
var queue = new ConcurrentQueue<int>();
// ....
if(queue.TryDequeue(out var next)) ←
{
    // использовать next
}
```

Проверяет и одновременно
извлекает элемент

Здесь операции проверки и извлечения из очереди объединяются в вызове `TryDequeue`, что устраняет временное окно между проверкой и операцией извлечения из очереди и решает проблему.

Классы `ConcurrentQueue<T>` и `ConcurrentStack<T>` можно использовать напрямую, когда требуется потокобезопасная очередь или стек и не нужен механизм оповещения о доступности элемента для обработки. Но чаще они применяются в сочетании с `BlockingCollection<T>`.

13.2.5. `ConcurrentBag<T>`

В отличие от `ConcurrentQueue<T>` и `ConcurrentStack<T>`, класс `ConcurrentBag<T>` не имеет аналогов среди обычных непотокобезопасных коллекций. Структура данных `ConcurrentBag<T>` не располагает элементы в каком-то определенном порядке — они могут извлекаться в любом порядке. `ConcurrentBag<T>` может хранить повторяющиеся элементы. Для добавления элементов `ConcurrentBag<T>` предлагает метод `Add`, а для извлечения — метод `TryTake`.

Реализация `ConcurrentBag<T>` использует отдельные очереди для каждого потока, а `TryTake` пытается возвращать элементы, добавленные тем же потоком. Это удобно тем, что `ConcurrentBag<T>` не должен блокироваться, если два потока одновременно попытаются получить элемент. Если элемент, добавленный текущим потоком, недоступен, то `TryTake` извлечет элемент из очереди другого потока (это называется *перехватом работы — work stealing*), но такая ситуация требует синхронизации потоков, что увеличивает накладные расходы и замедляет выполнение.

По этой причине `ConcurrentBag<T>` следует использовать, только если один и тот же поток (или набор потоков) и добавляет, и извлекает элементы из коллекции. Например, применение `ConcurrentBag<T>` в качестве базовой коллекции для организации очереди заданий на основе `BlockingCollection<T>` будет эффективным, только если элементы были записаны в коллекцию тем же потоком, который их обрабатывает. Кроме того, `ConcurrentBag<T>` не следует использовать в асинхронном коде, потому что обычно вы не контролируете, какой поток обрабатывает его.

13.2.6. Когда следует использовать потокобезопасные коллекции

Класс `ConcurrentDictionary< TKey, TValue >` — очень хорошая потокобезопасная альтернатива `Dictionary< TKey, TValue >`. Мы уже видели примеры его использования в роли внутреннего кэша. Эту коллекцию можно использовать всегда, когда словарь должен быть доступен из нескольких потоков одновременно. Кроме того, его можно использовать как в синхронном, так и в асинхронном коде.

Аналогично `ConcurrentQueue< T >` и `ConcurrentStack< T >` — хорошие потокобезопасные реализации очереди и стека. Их можно использовать всегда, когда требуется параллельный доступ к очереди или стеку и не нужен механизм оповещения о доступности элементов. Они тоже могут с успехом использоваться как в синхронном, так и в асинхронном коде.

Если вам потребуется механизм оповещения о доступности элемента, чтобы блокировать поток-потребитель в отсутствие доступных задач, то используйте `BlockingCollection< T >`. Однако `BlockingCollection< T >`, особенно его методы `Take` и `GetConsumingEnumerable`, не подходят для асинхронной модели программирования.

13.2.7. Когда не следует использовать потокобезопасные коллекции

Если в коде нужно использовать блокировку, чтобы сделать его потокобезопасным (не только для защиты доступа к коллекции), то лучше использовать эту блокировку для синхронизации доступа к коду (и коллекции), и нам не нужна потокобезопасная коллекция. В этом случае код, использующий непотокобезопасные альтернативы (`Dictionary< TKey, TValue >`, `Queue< T >` и `Stack< T >`), получится проще и быстрее.

Если потокобезопасность нужна только потому, что вы передаете коллекцию какому-то (возможно, внешнему) коду, который не изменяет коллекцию и допускает, что данные коллекции могли немного устареть (то есть коллекция не получила всех последних обновлений), то подумайте о возможности использования неизменяемых коллекций.

13.3. Неизменяемые коллекции

Проблемы потокобезопасности всегда сводятся к тому, что несколько потоков изменяют данные одновременно: одни потоки читают данные, в то время как другие потоки их изменяют, или к проблемам с временем выполнения, из-за которых потоки изменяют данные в неправильном порядке. Все эти проблемы связаны с изменением данных — если данные никогда не изменяются, то ни одна из этих проблем не возникнет и ваш код по своей природе будет потокобезопасным.

В отличие от параллельных коллекций, использующих замысловатые стратегии блокировки, чтобы сделать безопасным одновременное изменение коллекции несколькими потоками, неизменяемые коллекции обеспечивают потокобезопасность, просто будучи неизменяемыми. Если их нельзя изменить, то никакие два потока не смогут изменить их одновременно.

Неизменяемые коллекции работают подобно классу `String` в .NET. Все методы, изменяющие коллекцию, на самом деле оставляют ее нетронутой и возвращают совершенно новую коллекцию — копию исходной коллекции с внесенными изменениями.

Может показаться, что копирование существенно ухудшит производительность и повлечет чрезмерное потребление памяти, но в действительности неизменяемые коллекции смягчают эту проблему, используя внутренние структуры данных, которые могут совместно использовать фрагменты данных между коллекциями. Поэтому создание измененной копии коллекции обходится довольно дешево (или, по крайней мере, дешевле, чем копирование всей коллекции).

13.3.1. Как работают неизменяемые коллекции

Если заглянуть в стандартные коллекции, то можно увидеть, что все они основаны на массивах. Причина такой организации объясняется способом обращения к памяти, вследствие чего массивы оказываются самым эффективным вариантом хранения данных. `List<T>` — это обертка вокруг массива. `Queue<T>` и `Stack<T>` тоже внутри организованы как массивы. `HashSet<T>` — это хеш-таблица, реализованная с использованием двух массивов, а `Dictionary< TKey, TValue >` реализован с использованием четырех массивов. Все коллекции основаны на массивах. Однако массивы — это непрерывные блоки памяти, и, как следствие, массивы не могут иметь общие области памяти, поэтому вместо них неизменяемые коллекции обычно используют структуры данных, поддерживающие совместное использование блоков памяти.

Почему массивы эффективнее других структур данных

Процессор работает намного быстрее, чем оперативная память компьютера. Например, для процессора с тактовой частотой 2 ГГц каждый такт длится 0,5 нс, тогда как доступ к памяти DDR5 занимает около 16,67 нс. Проведя несложные арифметические вычисления, легко увидеть, что за время, необходимое для извлечения чего-либо из памяти, процессор успеет выполнить примерно 33 внутренние операции.

Очевидно, что заставлять процессор большую часть времени ждать поступления данных из памяти было бы неэффективно, поэтому умные разработчики процессо-

ров придумали решение — добавить немного памяти в сам процессор. Эта память работает почти так же быстро, как и ядра процессора (причина, мешающая сделать всю память компьютера такой же быстрой, — чрезвычайно высокая стоимость). Эта память называется *кэш-памятью процессора*. Внутри процессора есть также аппаратный компонент, который называется *контроллером кэша*. Одна из задач, которую решает контроллер кэша, — предварительная загрузка данных из основной памяти в кэш до того, как они понадобятся.

Массивы — это непрерывные блоки памяти. Все элементы массива хранятся в памяти друг за другом. Перебирая элементы массива в цикле, мы сканируем эту память последовательно. Когда память сканируется последовательно, контроллер кэша с легкостью угадывает, какое следующее значение будет извлечено из памяти — это значение следует сразу за текущим используемым.

Другие структуры данных, такие как связные списки и деревья, не являются непрерывными в памяти. Чтобы получить адрес памяти следующего элемента в связном списке, нужно прочитать узел текущего элемента и извлечь из него поле, содержащее адрес следующего элемента. Связные списки и деревья не имеют стандартной компоновки узлов, и контроллер кэша не знает, в каком порядке будут анализироваться узлы любой структуры данных из любой библиотеки, которую может использовать ваша программа.

Вот почему при сканировании массива следующее значение, скорее всего, будет ждать своей очереди в кэше, готовое к немедленному использованию, однако при доступе к другой структуре данных процессор потратит значительное время на ожидание передачи данных из оперативной памяти компьютера.

И еще одно небольшое замечание для читателей, которые разбираются в проектировании процессоров и могут заметить, что я упустил из виду строки кэша и тактовые циклы на операцию: вы совершенно правы, но эта книга не о проектировании процессоров, и приведенные здесь объяснения достаточно корректны, чтобы объяснить высокую производительность при работе с массивами.

Чтобы понять принципы работы неизменяемых коллекций, мы реализуем неизменяемый стек. Но прежде рассмотрим обычный стек на основе массива, чтобы потом было с чем сравнить.

Мы реализуем простейший потокобезопасный стек из возможных. В нашем стеке будет всего два метода: `Push`, добавляющий элемент на вершину стека, и `TryPop`, извлекающий элемент с вершины стека или возвращающий `false`, если стек пуст. Мы также ограничим размер нашего стека на основе массива десятью элементами, потому что наша цель — понять, как работает неизменяемый стек, а не механизм изменения размера массива. Мы добьемся потокобезопасности за счет применения ключевого слова `lock` для защиты доступа ко всему стеку (листинг 13.13).

Листинг 13.13. Реализация простого стека

```
public class MyStack<T>
{
    private T?[] _data = new T[10];
    private int _top = -1;
    private object _lock = new();
    public void Push(T item)
    {
        lock(_lock)
        {
            if(_top == _data.Length-1) throw new Exception("Стек полон");
            _top++;
            _data[_top] = item;
        }
    }
    public bool TryPop(out T? item)
    {
        if(_top == -1)
        {
            item = default(T);
            return false;
        }
        item = _data[_top];
        _data[_top] = default(T);
        _top--;
        return true;
    }
}
```

Теперь посмотрим, что получится, если мы запустим следующий тестовый код (листинг 13.14).

Листинг 13.14. Код для тестирования простого стека

```
var stack = new MyStack<int>();
stack.Push(1);
stack.Push(2);
stack.TryPop(out var item);
```

Этот код создает стек, затем помещает в него два значения (1 и 2) и потом извлекает последнее значение.

Давайте посмотрим, что происходит внутри стека после запуска тестового кода. При создании стека его конструктор записывает в поле `_top` значение `-1` (которое говорит, что стек пуст), а массив `_data` инициализируется средой выполнения нулевыми значениями (рис. 13.2).

После вызова `Push(1)` значение `_top` увеличивается до нуля, заставляя его указывать на текущую вершину стека, а значение 1 сохраняется в новой вершине (`_data[0]`; рис. 13.3).

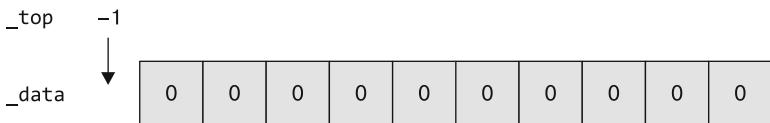


Рис. 13.2. Начальное состояние простого стека

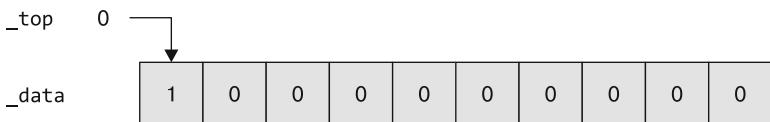


Рис. 13.3. Простой стек после добавления первого элемента

Вызов `Push(2)` снова увеличит `_top` до 1, заставляя его указывать на новую вершину стека `_data[1]`, и сохранит 2 в новой вершине (рис. 13.4).

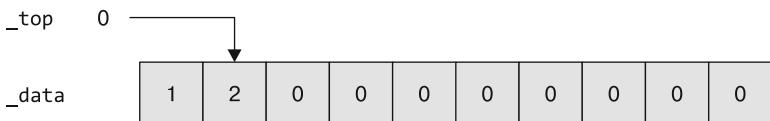


Рис. 13.4. Простой стек после добавления второго элемента

Вызов `TryPop` извлечет и вернет элемент на текущей вершине стека (`_data[1]`) в выходном параметре `item`. Он также запишет 0 в текущую вершину и уменьшит `_top` до нуля, сообщая, что новая вершина — `_data[0]`, фактически возвращая стек в то же состояние, в каком он находился до последнего вызова `Push` (рис. 13.5).

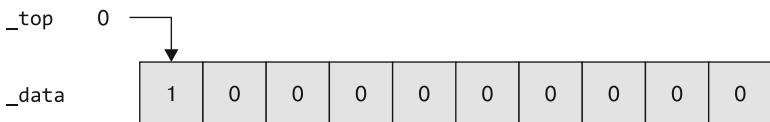


Рис. 13.5. Простой стек после извлечения элемента

Теперь, выяснив, как работает стандартный стек, реализуем неизменяемый стек. В случае неизменяемого стека мы больше не изменяем сам стек. Вместо этого каждый вызов будет возвращать новый стек. Теперь в нашем стеке будут методы `Push` и `Pop`, а также свойство `IsEmpty`. В обычной реализации нам пришлось бы объединить `Pop` и `IsEmpty` в один метод `TryPop`, потому что нет иного способа помешать другому потоку изменить стек между проверкой `IsEmpty` и вызовом `Pop`. Однако неизменяемый стек никто не сможет изменить, поэтому никто не сможет изменить стек между проверкой `IsEmpty` и вызовом `Pop`. Поскольку

стек неизменяемый, Push и Pop вместо изменения текущего будут возвращать новый стек с добавленным или удаленным элементом.

Если продолжить использовать массив, то при каждом вызове Push и Pop нам придется копировать весь массив, что, очевидно, нежелательно. К счастью, стек можно реализовать с помощью односвязного списка (листинг 13.15).

Листинг 13.15. Реализация неизменяемого стека

```
public class MyImmutableStack<T>
{
    private record class StackItem(T Value, StackItem Next);
    private readonly StackItem? _top;
    public MyImmutableStack() {}
    private MyImmutableStack(StackItem? top)
    {
        _top = top;
    }
    public MyImmutableStack<T> Push(T item)
    {
        return new MyImmutableStack<T>(new StackItem(item, _top));
    }
    public MyImmutableStack<T> Pop(out T? item)
    {
        if(_top == null)
            throw new InvalidOperationException("Стек пуст");
        item = _top.Value;
        return new MyImmutableStack<T>(_top.Next);
    }
    public bool IsEmpty => _top == null;
}
```

Теперь запустим тестовый код, эквивалентный коду в листинге 13.14 (листинг 13.16).

Листинг 13.16. Код для тестирования неизменяемого стека

```
var stack1 = new MyImmutableStack<int> ();
var stack2 = stack1.Push(1);
var stack3 = stack2.Push(2);
var stack4 = stack3.Pop(out var item);
```

Этот код создает новый пустой стек и присваивает его переменной `stack1`. Затем он вызывает Push, который создает еще один стек с новым элементом, и присваивает его переменной `stack2`. Далее он снова вызывает Push, который тоже создает новый стек с дополнительным элементом, и сохраняет его как `stack3`. Наконец, он вызывает Pop, который создает еще один стек, но на этот раз с удаленным верхним элементом, и помещает его в переменную `stack4`.

Давайте посмотрим, что происходит внутри неизменяемого стека после запуска тестового кода. Конструктор `MyImmutableStack` без параметров создаст новый стек, не содержащий элементов `StackItem` (`_top` будет иметь значение `null`; рис. 13.6).

Вызов `Push(1)` создаст новый стек, который будет содержать один элемент `StackItem` с полем `Value`, содержащим 1, и полем `Next`, ссылающимся на предыдущее значение `_top` (`null`; рис. 13.7).

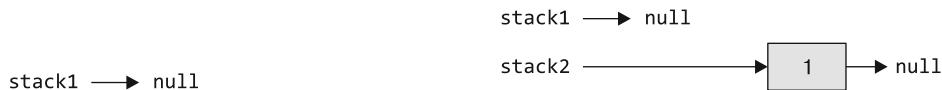


Рис. 13.6. Неизменяемый стек в начальном состоянии



Рис. 13.7. Неизменяемый стек после добавления первого элемента

Вызов `Push(2)` создаст новый стек и новый `StackItem`. Новый `StackItem` будет содержать в поле `Value` значение 2 и в поле `Next` ссылку на `_top` в `stack2` — существующий элемент `StackItem`, хранящий значение 1. Обратите внимание, что теперь два стека совместно используют первый `StackItem` (рис. 13.8).

Наконец, вызов `Pop` создаст новый стек (что неудивительно). Новый стек будет указывать на `_top.Next`, то есть на старый `StackItem` со значением 1, которое теперь будет общим для трех стеков (рис. 13.9).

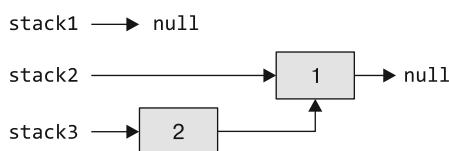


Рис. 13.8. Неизменяемый стек после добавления второго элемента

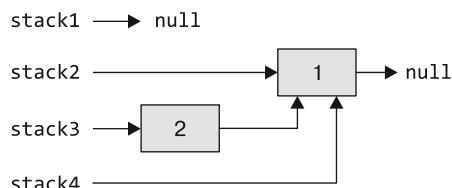


Рис. 13.9. Неизменяемый стек после удаления элемента

Обычно мы повторно использовали бы одну и ту же переменную и не создавали бы новую для каждой версии стека (у нас будет только одна переменная вместо `stack1`, `stack2`, `stack3` и `stack4`), однако это ничего не меняет (за исключением того, что с отдельными именами переменных рисунки становятся более понятными). Даже при повторном использовании переменных все старые стеки будут по-прежнему храниться в памяти до следующего запуска сборщика мусора.

Теперь вы знаете, как каждая операция с неизменяемым стеком создает новый стек, выполняя незначительный объем работы и без копирования каких-либо

элементов, уже находящихся в стеке. Это происходит за счет использования немного большего количества памяти (каждый элемент теперь имеет ссылку `Next` и все накладные расходы объекта) и распределения элементов в памяти вместо их последовательного хранения в массиве (что из-за особенностей конструкции кэша процессора замедляет доступ к ним).

Я решил продемонстрировать, как работает неизменяемый стек, потому что это самая простая из неизменяемых коллекций. Однако все они используют одну и ту же базовую тактику — размещение данных внутри узловых объектов и проектирование операций таким образом, чтобы каждая «модифицированная» копия могла использовать большинство узлов предыдущей коллекции. Как видите, стек можно реализовать всего одним связным списком. Для реализации очереди потребуются два связных списка, а в реализациях большинства других неизменяемых коллекций используется некое подобие двоичного дерева. Но мы не будем обсуждать детали реализации всех неизменяемых коллекций, так как это выходит за рамки книги.

13.3.2. Как использовать неизменяемые коллекции

Некоторые данные изменяются так редко, что хранение их в неизменяемой структуре данных не вызывает проблем. Например, список стран мира иногда меняется, но это происходит настолько редко, что мы легко можем смириться с необходимостью перезапуска нашего сервиса для обновления этого списка. Однако данные, которые действительно критичны для нашего ПО, те данные, которыми мы управляем, имеют тенденцию постоянно меняться.

Допустим, мы создаем сайт электронной коммерции, который продает книги. Очевидно, что процветание компании зависит от продажи большого количества книг, поэтому нам хотелось бы иметь возможность продавать больше одной книги одновременно. По этой причине мы решили использовать неизменяемую коллекцию для хранения каталога книг из-за присущей ей потокобезопасности. Напишем код для управления нашим каталогом (листинг 13.17).

Мы написали класс `InventoryManager` с единственным методом `TryToBuyBook`. Этот метод сначала извлекает количество копий книги, хранящихся на складе, из `ImmutableDictionary<string,int>`, на который ссылается переменная `_bookIdToQuantity`. Если искомой книги нет в каталоге или на складе нет ни одной ее копии, то метод возвращает `false`, чтобы сообщить, что клиент не может купить книгу. Если все в порядке, метод обновляет количество копий на складе, используя метод `SetItem` словаря, который создает новый словарь с обновленным количеством копий, сохраняет новый словарь в той же переменной и возвращает `true`.

Листинг 13.17. Непотокобезопасное управление стеком с помощью `ImmutableDictionary`

```
public class InventoryManager
{
    private ImmutableDictionary<string,int> _bookIdToQuantity;

    public bool TryToBuyBook(string bookId)
    {
        if(!_bookIdToQuantity.TryGetValue(bookId, out var copiesInStock)) ←
            return false;
        if(copiesInStock == 0) ←
            return false;
        _bookIdToQuantity =
            _bookIdToQuantity.SetItem(bookId, copiesInStock-1); ←
        return true;
    }
}
```

Если поразмыслить над происходящим при выполнении этого кода сразу в нескольких потоках, то можно заметить проблему: словарь не может измениться, он неизменяем, и другой поток не сможет изменить словарь. Однако, если этот словарь ссылается на обычную изменяемую переменную, другой поток сможет изменить переменную, заменив один словарь другим, и это изменение не защищено неизменяемой структурой данных (рис. 13.10).

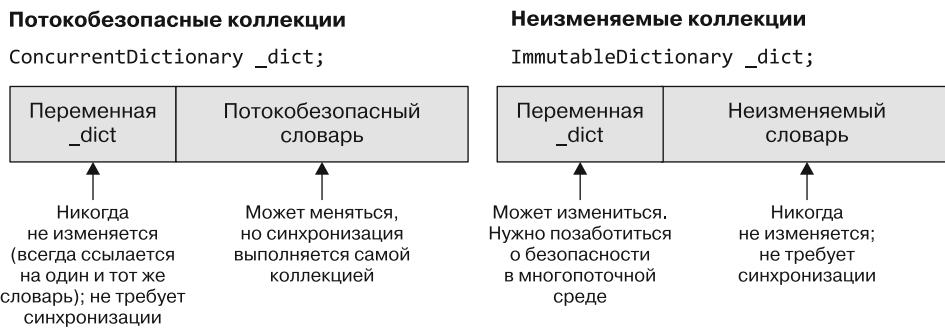


Рис. 13.10. Необходимость синхронизации при использовании неизменяемых и параллельных коллекций

Самое простое решение — заключить все тело метода в блокировку `lock`. Это решает проблему повреждения данных, но сводит на нет все преимущества потокобезопасности неизменяемой коллекции. Если мы будем использовать блокировку `lock` при каждом обращении к словарю, с тем же успехом мы можем использовать более простой непотокобезопасный `Dictionary< TKey, TValue >`. Сложное решение заключается в добавлении `ImmutableInterlocked`.

13.3.3. *ImmutableInterlocked*

Класс *ImmutableInterlocked* позволяет выполнять операции изменения переменной, ссылающейся на неизменяемую коллекцию, без блокировок. Он содержит методы, реализующие те же операции, которые мы видели в потокобезопасных коллекциях, но на этот раз для неизменяемых коллекций.

Например, для *ImmutableDictionary<TKey, TValue>* класс *ImmutableInterlocked* предоставляет методы *AddOrUpdate*, *GetOrAdd*, *TryAdd*, *TryRemove* и *TryUpdate*, которые аналогичны одноименным методам в *ConcurrentDictionary<TKey, TValue>*.

Как мы видели в листинге 13.10, *ConcurrentDictionary<TKey, TValue>.TryUpdate* проверяет, было ли обновляемое значение изменено кем-то другим. Если значение не было изменено другим потоком, то он заменит это значение в словаре. *ImmutableInterlocked.TryUpdate* делает то же самое для неизменяемых словарей: проверяет, было ли значение, которое мы пытаемся обновить, изменено кем-то другим. Если значение не изменилось, он заменит весь словарь новым словарем с одним измененным значением.

Если значение было изменено другим потоком, то он так же, как *ConcurrentDictionary<TKey, TValue>.TryUpdate*, вернет *false*, ничего не изменив, и тогда нам нужно обработать эту ситуацию, например, чтением нового значения и повторением операции до успешного выполнения (листинг 13.18).

Листинг 13.18. Потокобезопасное управление каталогом магазина с помощью *ImmutableInterlocked*

```
private ImmutableDictionary<string, int> _bookIdToQuantity = new();
public bool TryToBuyBook(string bookId)
{
    while(true)
    {
        if(!_bookIdToQuantity.TryGetValue(bookId,
            out var copiesInStock)) ← Получает предыдущее
            return false;           | количества
        if(copiesInStock == 0)
            return false;
        if(ImmutableInterlocked.TryUpdate(
            ref _bookIdToQuantity, bookId,
            copiesInStock - 1, copiesInStock)) | Устанавливает новое количество,
            return true;           | только если оно не было
                                | изменено другим потоком
    } } ← Возвращает true
        | Если значение изменено
        | другим потоком,
        | повторяет попытку
```

Это потокобезопасная версия кода из листинга 13.17. Он считывает количество копий на складе и пытается уменьшить его на 1. Если количество копий успело измениться между моментом чтения и моментом записи уменьшенного значения,

то код снова читает значение, проверяет наличие копий на складе и снова пытается уменьшить значение в словаре `_bookIdToQuantity`. Так будет продолжаться до тех пор, пока операция записи не увенчается успехом либо пока транзакции, запущенные в других потоках, не приведут к тому, что количество копий на складе уменьшится до нуля.

`ImmutableInterlocked` также предоставляет методы `Push` и `TryPop` для использования с `ImmutableStack<T>` и `Enqueue` и `TryDequeue` для `ImmutableQueue<T>`. Обратите внимание, что `ImmutableInterlocked` — это операция типа «все или ничего»: она безопасна, только если все изменения производятся с помощью `ImmutableInterlocked`. Также обратите внимание, что вы не можете использовать `ImmutableInterlocked` для потокобезопасного обновления нескольких коллекций одновременно. Если, например, нужно добавить один и тот же ключ в два словаря, то этого не получится сделать с помощью `ImmutableInterlocked`, не создав временное окно, в течение которого другой поток сможет прочитать изменение в первом словаре до того, как вы обновите второй. То же самое касается нескольких изменений в одном словаре — `ImmutableInterlocked` не получится использовать для атомарного изменения двух значений.

13.3.4. `ImmutableDictionary< TKey, TValue >`

`ImmutableDictionary< TKey, TValue >`, как нетрудно догадаться, является неизменяемой версией `Dictionary< TKey, TValue >`. Так же как `Dictionary< TKey, TValue >`, неизменяемый словарь `ImmutableDictionary< TKey, TValue >` позволяет проверить существование ключа с помощью `ContainsKey`, извлечь значение с помощью оператора `[]` и выполнить оба действия атомарно с помощью `TryGetValue`.

Он также имеет многие методы `Dictionary< TKey, TValue >` для изменения словаря (`Add`, `Remove` и т. д.), но в `ImmutableDictionary< TKey, TValue >` они не изменяют существующий, а возвращают новый словарь с изменениями. Чтобы изменить определенное значение внутри словаря, нужно вместо оператора `[]` использовать метод `UpdateValue`, потому что оператор `[]` не имеет возможности вернуть новый словарь.

Сам по себе `ImmutableDictionary< TKey, TValue >` не предлагает никаких специальных операций, имеющихся в `ConcurrentDictionary< TKey, TValue >`, таких как `GetOrAdd`, поскольку `ImmutableDictionary< TKey, TValue >` является неизменяемым и не может измениться между вызовами `Get` и `Add`.

Для атомарного добавления, удаления или изменения сразу нескольких значений `ImmutableDictionary< TKey, TValue >` предоставляет методы `AddRange`, `RemoveRange` и `UpdateItems`. Эти методы помогают повысить производительность кода, создавая только один новый `ImmutableDictionary< TKey, TValue >` со всеми изменениями вместо добавления нового словаря для каждого измененного значения.

Если вам потребуется внести несколько изменений разных типов (например, добавить одно значение и удалить другое) или ваш алгоритм не позволяет сгруппировать изменения (например, если нет возможности заменить несколько вызовов `Add` одним вызовом `AddRange`), то можно воспользоваться объектом-построителем (`builder`). Объект-строитель создается с помощью метода `ToBuilder` словаря. Постройтель не является неизменяемым, и его можно использовать для внесения нескольких изменений без создания нового словаря. Когда необходимые изменения будут выполнены, останется только вызвать метод `ToImmutable` построителя, который создаст новый `ImmutableDictionary<TKey, TValue>` со всеми изменениями.

Постройтель не является ни неизменяемым, ни потокобезопасным и позволяет быстро и эффективно создавать новые объекты `ImmutableDictionary<TKey, TValue>`, но он не предоставляет никаких средств для работы в многопоточной среде.

Как было показано выше в этой главе, переменная, содержащая актуальную версию словаря, является простой переменной и ведет себя в многопоточной среде точно так же, как любая другая переменная (то есть изменять ее в многопоточной среде без блокировки небезопасно). Также выше в этой главе было показано, что для быстрого и неблокирующего способа синхронизации доступа к этой переменной можно использовать `ImmutableInterlocked`. Этот класс предлагает методы `AddOrUpdate`, `GetOrAdd`, `TryAdd`, `TryRemove` и `TryUpdate`, которые действуют точно так же, как одноименные методы `ConcurrentDictionary<TKey, TValue>` (за исключением того, что словарь, будучи неизменяемым, никогда не изменяется и эти операции возвращают новый словарь, включающий запрошенные изменения).

Если `ImmutableInterlocked` используется в паре с `ImmutableDictionary<TKey, TValue>`, скорее всего, вы делаете это, чтобы получить поведение, аналогичное `ConcurrentDictionary<TKey, TValue>`. В таком случае, вероятно, лучше использовать `ConcurrentDictionary<TKey, TValue>`.

Помните, что для создания измененных копий можно использовать либо методы `ImmutableDictionary<TKey, TValue>` (в том числе и объект-строитель), либо `ImmutableInterlocked`. Применение обоих методов не является потокобезопасным.

13.3.5. `ImmutableHashSet<T>` и `ImmutableSortedSet<T>`

`ImmutableHashSet<T>` и `ImmutableSortedSet<T>`, как легко догадаться, — это неизменяемые версии `HashSet<T>` и `SortedSet<T>`. Оба класса представляют множества — коллекции, которые могут содержать ноль или более элементов без дубликатов и поддерживают операции из теории множеств (`Except`, `Intersect`, `IsSubsetOf`, `IsSupersetOf` и др.).

При обходе содержимого `ImmutableHashSet<T>` (например, с помощью `foreach`) порядок возвращаемых элементов полностью произвольный и не поддается контролю, но операции с этим множеством, особенно поиск, выполняются очень быстро. Напротив, при обходе содержимого `ImmutableSortedSet<T>` элементы возвращаются в порядке сортировки (которым можно управлять, передав `IComparer<T>` в вызов статического метода `ImmutableSortedSet.Create<T>`), но операции с таким множеством будут выполняться медленнее. Из-за разницы в производительности рекомендуется отдавать предпочтение `ImmutableHashSet<T>` и использовать `ImmutableSortedSet<T>`, только если для вас важен порядок элементов при перечислении.

По аналогии с другими неизменяемыми коллекциями все методы, изменяющие коллекцию, возвращают новую коллекцию с внесенными изменениями. Кроме того, подобно `ImmutableDictionary< TKey, TValue >`, коллекции `ImmutableHashSet<T>` и `ImmutableSortedSet<T>` имеют метод `ToBuilder`, возвращающий объект-построитель. Данный объект можно использовать для эффективного внесения множества изменений, а затем создать новое неизменяемое множество, содержащее все эти изменения. Напомню, что объект-построитель не является потокобезопасным.

Кроме того, по аналогии с другими неизменяемыми коллекциями если у вас есть переменная, ссылающаяся на актуальную версию коллекции, то вы должны сами синхронизировать доступ к ней (например, с помощью блокировки `lock`). В отличие от `ImmutableDictionary< TKey, TValue >`, неизменяемые классы множеств нельзя использовать в паре с `ImmutableInterlocked`.

13.3.6. *ImmutableList<T>*

`ImmutableList<T>`, как вы наверняка поняли, — это неизменяемая версия `List<T>`, которая, что совсем неудивительно, работает так же, как и предыдущие неизменяемые коллекции. Все методы, не изменяющие список, работают точно так же, как в `List<T>`. Все методы, изменяющие список, возвращают новый `ImmutableList<T>`. Чтобы добавить элемент в определенное место в списке, вместо оператора `[]` следует использовать метод `SetItem`, потому что оператор не имеет возможности вернуть новый объект `ImmutableList<T>`.

Как и все предыдущие неизменяемые коллекции, `ImmutableList<T>` имеет метод `ToBuilder`, возвращающий объект-построитель, который можно использовать для выполнения множества изменений без создания промежуточных объектов `ImmutableList<T>`. Как и в случаях с другими неизменяемыми коллекциями, объект-построитель не является потокобезопасным.

Если у вас есть переменная, содержащая объект `ImmutableList<T>`, к которой обращаются разные потоки, то вы должны самостоятельно синхронизировать доступ к ней. `ImmutableInterlocked` не поддерживает `ImmutableList<T>`.

13.3.7. *ImmutableQueue<T>* и *ImmutableStack<T>*

ImmutableQueue<T> и *ImmutableStack<T>* — это неизменяемые версии *Queue<T>* и *Stack<T>* соответственно. Очередь и стек почти всегда используются как временные хранилища, в которых элементы постоянно добавляются и удаляются, поэтому обычно лучше отдать предпочтение *ConcurrentQueue<T>* и *ConcurrentStack<T>*, за исключением случаев, когда код пишется в функциональном стиле.

Если вы все-таки используете очередь или стек, свойства *ImmutableQueue<T>.IsEmpty* и *ImmutableStack<T>.IsEmpty* сообщат вам, когда коллекция опустеет, а методы *ImmutableQueue<T>.Enqueue* и *ImmutableStack<T>.Push* создадут новую очередь или стек с добавленным элементом. Методы *ImmutableQueue<T>.Peek* и *ImmutableStack<T>.Peek* вернут следующий элемент, не удаляя его из коллекции, а *ImmutableQueue<T>.Dequeue* и *ImmutableStack<T>.Pop* возвращают новую очередь или стек с удаленным элементом и при необходимости поместят удаленный элемент в выходной параметр *out*.

Методы *Peek*, *Dequeue* и *Pop* выбросят исключение, если очередь или стек пусты. Другой поток не сможет изменить коллекцию между проверкой свойства *IsEmpty* и вызовом *Peek*, *Dequeue* или *Pop* — коллекция неизменяема, ее вообще нельзя изменить, поэтому ее не сможет изменить никакой другой поток.

Если на очередь или стек ссылается переменная, доступная другим потокам для записи, то вы должны синхронизировать доступ к этой переменной и как минимум скопировать ссылку в локальную переменную перед чтением *IsEmpty*, чтобы другой поток не мог заменить очередь или стек между чтением *IsEmpty* и вызовом *Peek*, *Dequeue* или *Pop*.

ImmutableInterlocked поддерживает как *ImmutableQueue<T>*, так и *ImmutableStack<T>*, предлагая операции, аналогичные тем, что доступны в *ConcurrentQueue<T>* и *ConcurrentStack<T>*. Однако если они вам понадобятся, то в таких случаях почти всегда лучше использовать *ConcurrentQueue<T>* или *ConcurrentStack<T>*.

13.3.8. *ImmutableArray<T>*

В начале рассказа о неизменяемых коллекциях я отметил, что все непотокобезопасные коллекции используют массивы, потому что процессор работает с ними намного быстрее, но неизменяемые коллекции не используют массивы, потому что иначе им пришлось бы копировать все данные при каждом изменении (чтобы получить новую коллекцию с изменением). *ImmutableArray<T>* — это исключение, которое, как следует из названия, действительно использует массив.

Коллекция `ImmutableArray<T>` не свободна от недостатков, свойственных массивам, а это означает, что неизменяемый массив вынужден копировать все данные при каждом изменении, что делает `ImmutableArray<T>` самой медленной неизменяемой коллекцией в операциях записи. Однако, поскольку она использует массив, она также самая быстрая в операциях чтения. Это обстоятельство делает `ImmutableArray<T>` больше похожим на замороженные коллекции (описываются далее в этой главе).

`ImmutableArray<T>` — хороший выбор в ситуациях, когда нужно передать массив, доступный только для чтения, в какой-то сторонний код. Будучи массивом, эта коллекция быстро сканируется и даже поддерживает объекты `ReadOnlyMemory<T>` и `ReadOnlySpan<T>`.

С другой стороны, `ImmutableArray<T>` — не лучший выбор в качестве основы для структуры данных, потому что операции изменения выполняются медленно и потребляют много памяти. Однако иногда, когда изменения очень редки или массив небольшой, эту коллекцию вполне можно использовать, но будьте очень осторожны с ней.

Создать новый неизменяемый массив из обычного массива или из четырех отдельных элементов данных можно с помощью `ImmutableArray.Create<T>`. Создать неизменяемый массив из любой коллекции также можно с помощью `ImmutableArray.ToImmutable<T>`, а с помощью `ImmutableArray.CreateRange` можно создать неизменяемый массив из подмножества другой коллекции.

Как всегда, если переменная, ссылающаяся на неизменяемый массив, доступна нескольким потокам, то вы должны самостоятельно синхронизировать доступ к ней. `ImmutableInterlocked` поддерживает `ImmutableArray<T>`, предлагая методы `InterlockedCompareExchange`, `InterlockedExchange` и `InterlockedInitialize`, но их лучше не использовать. Они сложны, и работа с ними чревата ошибками; намного проще использовать оператор `lock`, и если ваш код не имеет критических требований к производительности, то нет смысла тратить время и силы на преодоление сложностей.

В общем случае `ImmutableArray<T>` следует использовать так же, как и замороженные коллекции (которые описываются чуть ниже). `ImmutableArray<T>` неэффективен в создании (как по скорости, так и по потреблению памяти), но очень эффективен в использовании. Он хорошо подходит для случаев, когда нужна последовательная коллекция (например, `List<T>` или массив), потокобезопасная и доступная только для чтения. Однако, поскольку его повторное создание обходится слишком дорого, его следует использовать, только когда он вообще не будет изменяться (то есть не будут создаваться новые измененные копии).

13.3.9. Когда использовать неизменяемые коллекции

Неизменяемые коллекции очень распространены в функциональном программировании. Если вы пишете код в функциональном стиле или используете функциональные алгоритмы, то неизменяемые коллекции идеально вам подойдут.

Неизменяемые коллекции также удобны, когда нужно сохранить предыдущие состояния системы, например, для реализации возможности отмены. Обратите внимание, что неизменяемые коллекции не подходят для случаев, когда нужно сохранить состояние системы для целей аудита, потому что тогда состояние системы нужно сохранять на диске, а неизменяемые коллекции находятся только в памяти.

Неизменяемые коллекции также очень полезны, когда нужно передать коллекцию стороннему коду. В таком случае вам не придется дублировать данные для защиты и отправлять копию стороннему коду.

Но всякий раз, используя неизменяемые коллекции, вы должны помнить, что, несмотря на полную потокобезопасность самих неизменяемых коллекций, «изменение коллекции» подразумевает создание новой коллекции, а вновь созданные копии обычно присваиваются той же переменной, что и предыдущая коллекция. Эта переменная будет изменяться с каждым изменением коллекции, и поэтому доступ к ней следует синхронизировать, как и к любой другой переменной, к которой одновременно обращается несколько потоков. Такой подход часто требует использования блокировки `lock` при обновлении коллекции или `ImmutableInterlocked`, и в этих случаях код, скорее всего, будет проще и быстрее, если использовать потокобезопасные коллекции.

Наконец, в случаях, когда данные действительно никогда не меняются, можно подумать об использовании замороженных коллекций.

13.4. Замороженные коллекции

Выше мы видели, что неизменяемые коллекции никогда не изменяются. Если вы хотите их изменить, нужно создать копию коллекции с требуемыми изменениями. Мы также видели, что в качестве компромисса неизменяемые коллекции используют менее эффективные структуры данных, ускоряющие создание измененных копий. Но что, если этот компромисс неприемлем? Что, если данные действительно никогда не изменяются? Что, если мы не хотим жертвовать производительностью чтения для поддержки операций записи, которые нам не нужны? Для таких случаев есть замороженные коллекции.

Замороженные коллекции — это коллекции, доступные только для чтения и оптимизированные для чтения. Они создаются медленнее обычных,

потокобезопасных или неизменяемых коллекций, но чтение из них происходит с максимальной скоростью.

Замороженные коллекции предназначены только для чтения. Их вообще нельзя изменить, и у них даже нет методов создания измененных копий, как у неизменяемых коллекций.

В настоящее время существуют только два типа замороженных коллекций — `FrozenDictionary< TKey, TValue >` и `FrozenSet< T >` — версии `Dictionary< TKey, TValue >` и `HashSet< T >` соответственно, доступные только для чтения. Если вам нужна замороженная версия `List< T >`, то используйте `ImmutableArray< T >` (я говорил об этом выше). Замороженных очередей и стеков не существует, потому что в этом нет никакого смысла.

Создать `FrozenSet< T >` можно на основе любой коллекции, вызвав метод расширения `ToFrozenSet` (листинг 13.19).

Листинг 13.19. Инициализация FrozenSet

```
var data = new List<int> {1, 2, 3, 4};
var set = data.ToFrozenSet();
```

Этот код создает `List< int >` с некоторыми числами, а затем вызывает `ToFrozenSet`, чтобы создать `FrozenSet< int >` с тем же содержимым.

В качестве основы для создания `FrozenDictionary< TKey, TValue >` можно использовать любую коллекцию и вызвать метод расширения `ToFrozenDictionary`. Самый простой способ получить `FrozenDictionary< TKey, TValue >` — использовать `Dictionary< TKey, TValue >` (листинг 13.20).

Листинг 13.20. Инициализация FrozenDictionary из словаря

```
var numberNames = new Dictionary<int, string>
{
    {1, "one"},
    {2, "two"}
};
var frozenDict = numberNames.ToFrozenDictionary();
```

Этот код создаст `Dictionary< int, string >`, который связывает числа и их названия на английском языке (только для единицы и двойки — чтобы пример был короче), а затем использует `ToFrozenDictionary` для создания `FrozenDictionary< int, string >` с тем же содержимым. Метод `ToFrozenDictionary` имеет перегруженную версию, которая принимает два делегата: один для извлечения ключей и второй — для значений. Такой метод можно использовать для преобразования любой коллекции (листинг 13.21).

Листинг 13.21. Создание FrozenDictionary из списка

```
var data = new List<int> {1,2,3,4};
var frozenDict = data.ToFrozenDictionary(x=>x,x=>x.ToString());
```

Этот код создает `List<int>` с некоторыми числами, а затем использует `ToFrozenDictionary` для создания `FrozenDictionary<int, string>`, связывающего числа в списке и их строковые представления. Обратите внимание, что, если исходная коллекция содержит дубликаты значений (в случае `FrozenSet`) или дубликаты ключей (в случае `FrozenDictionary`), в результирующую коллекцию будет включена только последняя запись для одного и того же ключа. Это отличается от поведения `Dictionary<TKey, TValue>` и `HashSet<T>`, которые выбрасывают исключение при попытке добавить дубликат ключа.

13.4.1. Когда использовать замороженные коллекции

Замороженные коллекции следует использовать, только когда данные (почти) никогда не изменяются. Замороженные коллекции оптимизируют чтение за счет значительного замедления операции создания коллекции. Если данные часто используются, но никогда не изменяются, то такие коллекции могут повысить производительность. Напротив, если данные часто изменяются, то время, необходимое для создания замороженной коллекции после каждого изменения, может легко превысить время, сэкономленное за счет более быстрого чтения.

Итоги главы

- Обычные коллекции из пространства имен `System.Collections.Generic` можно читать сразу из нескольких потоков.
- Запись из нескольких потоков одновременно или запись из одного потока, когда другие потоки выполняют чтение, не допускается и может привести к получению неверных результатов и даже к повреждению коллекций.
- Потокобезопасные коллекции из пространства имен `System.Collections.Concurrent` полностью потокобезопасны и поддерживают одновременные чтение и запись из нескольких потоков.
- Существуют потокобезопасные версии `Dictionary<TKey, TValue>`, `Queue<T>` и `Stack<T>`, которые называются `ConcurrentDictionary<TKey, TValue>`, `ConcurrentQueue<T>` и `ConcurrentStack<T>` соответственно. Их интерфейс отличается от обычных коллекций — он объединяет часто используемые операции в одну, чтобы избежать состояния гонки.
- Коллекция `ConcurrentBag<T>` полезна в случаях, когда порядок извлечения элементов не имеет значения. Она предназначена для использования, когда одни и те же потоки одновременно читают из коллекции и записывают в нее.

- Класс `BlockingCollection<T>` добавляет поддержку сценариев «производитель — потребитель» и помогает ограничить размер коллекции. По умолчанию `BlockingCollection<T>` работает как очередь, но может также использоваться как стек.
- Коллекции `ConcurrentDictionary<TKey, TValue>`, `ConcurrentQueue<T>`, `ConcurrentStack<T>` и `ConcurrentBag<T>` можно использовать в асинхронном коде.
- Класс `BlockingCollection<T>`, как следует из его имени, является блокирующим, поэтому используйте его с осторожностью (а лучше вообще не используйте) с асинхронным кодом.
- `BlockingCollection<T>` не имеет асинхронной версии, зато есть класс `Channel<T>`, с помощью которого можно создать асинхронную версию наиболее распространенного варианта использования `BlockingCollection<T>` (подробнее об этом в следующей главе).
- Неизменяемые коллекции из пространства имен `System.Collections.Immutable` — это коллекции, которые нельзя изменить (каждое изменение оставляет коллекцию нетронутой и создает новую коллекцию). Они потокобезопасны: поскольку по определению они вообще не могут быть изменены, они также не могут быть изменены другими потоками, пока вы их используете.
- Однако если переменная, ссылающаяся на неизменяемую коллекцию, доступна нескольким потокам, то вы должны самостоятельно синхронизировать доступ к ней. В этом может помочь класс `ImmutableInterlocked` (для словаря, очереди и стека).
- Чтобы внести несколько изменений в неизменяемую коллекцию, используйте метод `ToBuilder`. Он возвращает объект-построитель, который вносит изменения без создания новых коллекций, и затем, когда вызывается его метод `ToImmutable`, создает одну новую коллекцию сразу со всеми изменениями. Объект-построитель не является потокобезопасным.
- Классы `ImmutableDictionary<TKey, TValue>`, `ImmutableHashSet<T>`, `ImmutableSortedSet<T>`, `ImmutableQueue<T>`, `ImmutableStack<T>` и `ImmutableList<T>` являются неизменяемыми версиями одноименных классов без префикса `Immutable`.
- Эти коллекции читаются медленнее обычных или потокобезопасных коллекций, зато их копии создаются быстро, что очень важно, потому что при каждом изменении коллекции создается ее копия.
- Коллекция `ImmutableArray<T>` — это неизменяемый массив. Она отличается самой высокой скоростью чтения среди всех неизменяемых коллекций, но самой низкой скоростью изменения (то есть создания измененной копии). Она также поддерживает `ReadOnlyMemory<T>` и `ReadOnlySpan<T>`.

- Замороженные коллекции оптимизированы для чтения. Они имеют низкую скорость создания, но высокую скорость чтения. Их нельзя изменить.
- Замороженные коллекции, как и неизменяемые, по своей природе потокобезопасны.
- Существуют только две замороженные коллекции: `FrozenDictionary<TKey, TValue>` и `FrozenSet<T>`.
- Обычно для создания замороженной коллекции сначала создается обычная коллекция, а затем к ней применяется метод `ToFrozenSet` или `ToFrozenDictionary`.

Асинхронная генерация коллекций / `await foreach` и `IAsyncEnumerable`

В этой главе

- ✓ Как работает `await foreach`.
- ✓ Использование `yield return` в асинхронных методах.
- ✓ Итерации по асинхронным данным с использованием `IAsyncEnumerable<T>` и `await foreach`.

Иногда бывает нужно использовать `foreach` для обхода последовательности элементов, генерируемых на лету или извлекаемых из внешнего источника без предварительного сохранения всего набора элементов в коллекцию. Например, в главах 8 и 13 мы видели, как поддержка `foreach` в коллекции `BlockingCollection<T>` упрощает ее использование в роли очереди заданий. С# помогает решить эту задачу, предоставляя инструкцию `yield return`, обсуждавшуюся в главе 2. Однако обе рассмотренные версии `yield return`, а также `BlockingCollection<T>` не поддерживают асинхронное выполнение.

В этой главе мы рассмотрим асинхронную версию `foreach` (`await foreach`) и улучшения `yield return` из C# 8, позволяющие использовать ее в асинхронном коде. И наконец, с помощью всего этого мы напишем асинхронную версию `BlockingCollection<T>` и полностью асинхронную очередь заданий.

14.1. Итерации по асинхронной коллекции

Чтобы понять, как работает асинхронная инструкция `await foreach`, сначала нужно рассмотреть старую добрую синхронную инструкцию `foreach`. Ключевое слово `foreach` — это синтаксический сахар, просто более удобный способ записи кода по сравнению с использованием базовых возможностей языка. В частности, `foreach` — это способ записи цикла `while`. Реализацию `foreach` компилятором можно представить как простую замену кода. Компилятор получает код, такой как

```
foreach(var x in collection)
{
    Console.WriteLine(x);
}
```

и преобразует его в

```
using(var enumerator = collection.GetEnumerator())
{
    while(enumerator.MoveNext())
    {
        var x = enumerator.Current;
        Console.WriteLine(x);
    }
}
```

Как видите, `foreach` преобразуется в вызов `GetEnumerator`, который извлекает `IEnumerator<T>`, и цикл `while`, использующий `MoveNext` для получения следующего элемента в каждой итерации. В более общем смысле можно сказать, что компилятор принимает код в форме

```
foreach([loop-variable-type] [loop-variable] in [collection])
{
    [loop-body]
}
```

и транслирует его в

```
using(var enumerator = [collection].GetEnumerator())
{
    while(enumerator.MoveNext())
    {
        [loop-variable-type] [loop-variable] = enumerator.Current;
        [loop-body];
    }
}
```

Очевидно, что здесь я опустил немало деталей — существует множество особых случаев и оптимизаций, которые компилятор использует для улучшения

кода, но функционально цикл `foreach` эквивалентен показанному здесь циклу `while`.

Это преобразование дает хорошие результаты в синхронном коде, но для использования коллекции в асинхронном коде, когда извлечение следующего элемента производится в асинхронной операции, придется внести некоторые изменения в работу `foreach`. А именно, нужно добавить `await` в условие цикла `while` (третья строка в предыдущем фрагменте кода), а чтобы использовать `await`, метод `MoveNext` должен возвращать `Task<bool>` вместо `bool`.

Именно это и делают инструкция `await foreach` и интерфейс `IAsyncEnumerable<T>`. Интерфейс `IAsyncEnumerable<T>` похож на `IEnumerable<T>`: он имеет только один метод `GetAsyncEnumerator` (аналог `IEnumerable<T>.GetEnumerator`), который возвращает объект, реализующий интерфейс `IAsyncEnumerator<T>` (аналог `IEnumerator<T>`). Сам этот метод не является асинхронным и должен быстро возвращать управление. Любая длительная асинхронная инициализация должна происходить при первом вызове метода `MoveNextAsync`. Метод `MoveNextAsync` интерфейса `IAsyncEnumerable<T>` действует так же, как метод `IEnumerator<T>.MoveNext`, за исключением того, что возвращает `ValueTask<bool>` вместо `bool`.

В табл. 14.1 приводится сравнение между `IEnumerable<T>` и `IAsyncEnumerable<T>`, а в табл. 14.2 – сравнение между `IEnumerator` и `IAsyncEnumerator`.

Таблица 14.1. Сравнение `IEnumerable` и `IAsyncEnumerable`

<code>IEnumerable<T></code>	<code>IAsyncEnumerable<T></code>
<code>IEnumerator<T> GetEnumerator()</code>	<code>IAsyncEnumerator<T> GetAsyncEnumerator()</code>

Таблица 14.2. Сравнение `IEnumerator` и `IAsyncEnumerator`

<code>IEnumerator<T></code>	<code>IAsyncEnumerator<T></code>
<code>T Current {get;}</code>	<code>T Current {get;}</code>
<code>bool MoveNext()</code>	<code>ValueTask<bool> MoveNextAsync()</code>
<code>void Dispose()</code>	<code>ValueTask DisposeAsync()</code>

Как показано в таблицах, синхронные и асинхронные интерфейсы почти одинаковы и отличаются лишь поддержкой `async/await`. В таблицах не упоминается поддержка старого необобщенного интерфейса `IEnumerable`, поскольку он практически никогда не используется. Кроме того, я проигнорировал параметр токена отмены `IAsyncEnumerable<T>.GetAsyncEnumerator`, поскольку мы подробно поговорим о нем позже в этой главе.

Последняя часть головоломки — это не совсем удачно названный цикл `await foreach`, который похож на `foreach`, за исключением того, что использует `IAsyncEnumerable<T>` вместо `IEnumerable<T>` и добавляет требуемый `await`. Таким образом, цикл

```
await foreach(var x in collection)
{
    Console.WriteLine(x);
}
```

транслируется в

```
await using(var enumerator = collection.GetAsyncEnumerator())
{
    while(await enumerator.MoveNext()) ← Добавляет
    {
        var x = enumerator.Current;
        Console.WriteLine(x);
    }
}
```

14.2. Создание асинхронной коллекции

Теперь мы знаем, как использовать асинхронную коллекцию. На самом деле асинхронных коллекций как таковых не существует — по крайней мере в стандартной библиотеке. Все коллекции, включенные в библиотеку .NET, хранят элементы в памяти, и нет никакой необходимости получать элементы (асинхронно или как-то иначе), которые уже хранятся в памяти.

Когда мы говорим о поддержке асинхронных коллекций, то в действительности имеем в виду возможность использовать цикл `await foreach` для обработки последовательности элементов данных, которые генерируются или извлекаются асинхронно: фактически нам нужна асинхронная версия инструкции `yield return`, о которой мы говорили в главе 2, где использовали следующий код для динамического создания значений 1 и 2 (листинг 14.1).

Листинг 14.1. Пример использования `yield return` из главы 2

```
private IEnumerable<int> YieldDemo()
{
    yield return 1;
    yield return 2;
}

public void UseYieldDemo()
{
    foreach(var current in YieldDemo())
    {
        Console.WriteLine($"Получил {current}");
    }
}
```

Теперь добавим асинхронный вызов в метод генерации значений и для простоты используем `Task.Delay` (листинг 14.2).

Листинг 14.2. Пример асинхронного кода, использующего `yield return`

```
private async IAsyncEnumerable<int> AsyncYieldDemo()
{
    yield return 1;
    await Task.Delay(1000); ← Можно использовать await
    yield return 2;
}

public async Task UseAsyncYieldDemo()
{
    await foreach(var current in AsyncYieldDemo()) ← Вместо foreach используется await foreach
    {
        Console.WriteLine($"Получил {current}");
    }
}
```

Вместо `IEnumerable<int>`
используется `async
IAsyncEnumerable<int>`

Вместо `foreach` используется
`await foreach`

Нам пришлось изменить метод генератора, чтобы он возвращал `IAsyncEnumerable<int>` вместо `IEnumerable<int>`, и объявить его асинхронным — и все. Теперь внутри него можно использовать `await` и ранее упомянутую инструкцию `await foreach` для выполнения итераций по сгенерированной последовательности.

Как мы видели в главе 2, когда `yield return` использовался в синхронном коде, при компиляции кода выше компилятор аналогично преобразует метод `AsyncYieldDemo` в классы, реализующие `IAsyncEnumerable<int>` и `IAsyncEnumerator<int>`. Если применить те же преобразования, что и в главе 2, то получится код, показанный в листинге 14.3.

Листинг 14.3. Код, сгенерированный компилятором из кода в листинге 14.2

```
public class AsyncYieldDemo_Enumerable : IAsyncEnumerable<int>
{
    public IAsyncEnumerator<int> GetAsyncEnumerator(CancellationToken _)
    {
        return new YieldDemo_Enumerator();
    }
}

public class YieldDemo_Enumerator : IAsyncEnumerator<int>
{
    public int Current { get; private set; }
    private async Task Step0()
    {
        Current = 1;
    }
    private async Task Step1()
    {
```

```

    await Task.Delay(1000);
    Current = 2;
}

private int _step = 0;
public async ValueTask<bool> MoveNextAsync()
{
    switch(_step)
    {
        case 0:
            await Step0();
            ++_step;
            break;
        case 1:
            await Step1();
            ++_step;
            break;
        case 2:
            return false;
    }
    return true;
}
public ValueTask DisposeAsync() => ValueTask.CompletedTask;
}

public IAsyncEnumerable<int> AsyncYieldDemo()
{
    return new AsyncYieldDemo_Enumerable();
}

```

Это то же самое преобразование, которое мы видели в главе 2 (за исключением добавленных `async`, `await` и иногда `Task`, где это необходимо; изменения выделены жирным шрифтом). Вы можете вернуться к листингу 2.5, где приводится полный разбор этого кода, а здесь я перечислю лишь основные моменты:

- компилятор разбивает метод на части всякий раз, когда встречает `yield return`; каждая инструкция `yield return` завершает предшествующую часть;
- инструкция `yield return` заменяется операцией `Current =`;
- компилятор генерирует метод `MoveNextAsync`, который вызывает первый фрагмент при первом вызове, второй фрагмент при втором вызове и т. д.

В этом коде мы активно использовали `await`, но, как было показано в главе 3, `await` (как и `yield return`) реализуется путем преобразования вашего кода компилятором в класс. Давайте посмотрим, какой код генерирует компилятор для асинхронных методов в листинге 14.3. Начнем с метода `Step0`:

```

private Task Step0()
{
    Current = 1;
    return Task.CompletedTask;
}

```

Здесь все просто, потому что `Step0` не выполняет никаких асинхронных операций и компилятору не нужно его менять, поэтому он просто опускает ключевое слово `async` и явно возвращает `Task.CompletedTask`. Теперь посмотрим на `Step1`:

```
private TaskCompletionSource _step1tcs = new();
private Task Step1()
{
    Task.Delay(1000).ContinueWith(Step1Part2);
    return _step1tcs.Task;
}
private void Step1Part2(Task task)
{
    Current = 2;
    _step1tcs.SetResult();
}
```

В отличие от `Step0`, метод `Step1` выполняет асинхронную операцию, а именно `Task.Delay`, поэтому, как было показано в главе 3, весь код после `await` помещается в другой метод, который передается в вызов `ContinueWith`. Метод `Step1` должен возвращать экземпляр `Task`, для создания которого используется класс `TaskCompletionSource`, о котором рассказывалось в главе 10. Чтобы не усложнять код, я опустил обработку ошибок. Аналогичным образом компилятор транслирует метод `MoveNextAsync`.

14.3. Отмена итерации по асинхронной коллекции

Асинхронные операции часто поддерживают возможность отмены, и хотелось бы иметь аналогичную возможность в операциях, вызываемых из циклов `await foreach`. Для поддержки этой возможности метод `GetAsyncEnumerator` принимает токен отмены в качестве необязательного параметра, но само по себе это не решает нашу проблему, потому что:

- код, вызывающий `GetAsyncEnumerator`, при использовании `await foreach` генерируется компилятором и у нас нет очевидного способа передать токен отмены;
- код `GetAsyncEnumerator` тоже генерируется компилятором и у нас опять нет очевидного способа получить доступ к параметру метода.

Первая проблема решается методом расширения `WithCancellation`. Этот метод можно вызвать для любого `IAsyncEnumerable<T>`, и он вернет новый объект, который тоже реализует интерфейс `IAsyncEnumerable<T>`. Метод `GetAsyncEnumerator` этого нового объекта просто вызывает `GetAsyncEnumerator` исходного объекта с переданным нами токеном отмены. Чтобы использовать `WithCancellation`, достаточно вызвать его и использовать полученный объект, например, непосредственно в инструкции `await foreach`:

```
await foreach(var item in collection.WithCancellation(token))
```

Метод `WithCancellation` довольно прост. Вы можете реализовать его самостоятельно, для чего нужно определить класс, как показано в листинге 14.4.

Листинг 14.4. Реализация WithCancellation

```
public class WithCancellation<T> : IAsyncEnumerable<T>
{
    private IAsyncEnumerable<T> _originalEnumerable; | Поля для исходного экземпляра
    private CancellationToken _cancellationToken; | IAsyncEnumerable<T> и токена отмены

    public WithCancellation(
        IAsyncEnumerable<T> originalEnumerable,
        CancellationToken cancellationToken)
    {
        _originalEnumerable = originalEnumerable; | Сохраняет исходный экземпляр
        _cancellationToken = cancellationToken; | IAsyncEnumerable<T> и токен отмены
    }

    public IAsyncEnumerator<T> GetAsyncEnumerator(
        CancellationToken dontcare)
    {
        return _originalEnumerable.
            GetAsyncEnumerator(_cancellationToken); | Вызывает исходный экземпляр
                                                | IAsyncEnumerable<T> с токеном отмены
    }
}
```

Большая часть этого кода просто сохраняет асинхронный объект `IAsyncEnumerable<T>` и токен отмены, поэтому в методе `GetAsyncEnumerator` он может вызвать метод `GetAsyncEnumerator` исходного объекта и передать токен отмены в параметре.

Этот код решает первую проблему — позволяет передать токен отмены в `GetAsyncEnumerator`, когда он используется в `await foreach`. Однако остается вторая проблема — получение токена методом, генерирующим значения.

К счастью, компилятор C# позволяет это сделать. Он передаст токен отмены в параметре методу, генерирующему последовательность, если мы просто подскажем, какой именно параметр использовать. Чтобы указать такой параметр, нужно добавить атрибут `[EnumeratorCancellation]`. Теперь мы знаем, как изменить код из листинга 14.2, чтобы задействовать токен отмены (листинг 14.5).

В этом листинге мы добавили параметр `CancellationToken` в `AsyncYieldDemo` и декорировали его атрибутом `[EnumeratorCancellation]`, чтобы иметь возможность отменять выполнение `AsyncYieldDemo`. Затем использовали `WithCancellation` для передачи токена отмены в `AsyncYieldDemo`.

Поскольку `AsyncYieldDemo` вызывается и используется в итерациях в одном и том же методе, мы могли бы передать токен отмены в `AsyncYieldDemo` напрямую, но это не всегда возможно. Код, который создает `IAsyncEnumerable<T>`, и код, который выполняет итерации, могут находиться в разных компонентах.

Более того, исходный код, который создает `IAsyncEnumerable<T>`, может быть вообще недоступен или находиться в других методах, а использование `WithCancellation` просто проще, чем передача токена отмены в код, который создал объект `IAsyncEnumerable<T>`.

Листинг 14.5. Пример использования асинхронной инструкции `yield return` с отменой

```
private async IAsyncEnumerable<int> AsyncYieldDemo(
    [EnumeratorCancellation] CancellationToken cancellationToken = default)
{
    yield return 1;
    await Task.Delay(1000, cancellationToken);
    yield return 2;
}

public async Task UseAsyncYieldDemo()
{
    var cancel = new CancellationTokenSource();
    var collection = AsyncYieldDemo();
    await foreach(var current in
        collection.WithCancellation(cancel.Token)) ← Использует WithCancellation
    {                                         для передачи токена отмены
        Console.WriteLine($"Получил {current}");
    }
}
```

Этот пример поддерживает возможность отмены, однако он фактически никогда не отменяет операцию. Посмотрим, что произойдет, если выполнить отмену. Начнем с отмены еще до начала цикла (листинг 14.6).

Листинг 14.6. Отмена итераций до начала цикла

```
private async IAsyncEnumerable<int> AsyncYieldDemo(
    [EnumeratorCancellation] CancellationToken cancellationToken=default)
{
    yield return 1;
    await Task.Delay(1000, cancellationToken);
    yield return 2;
}

public async Task UseYieldDemo()
{
    var cancel = new CancellationTokenSource();
    cancel.Cancel(); ← Отменяет итерации
    await foreach(var current in
        AsyncYieldDemo().WithCancellation(cancel.Token))
    {
        Console.WriteLine($"Получил {current}");
    }
}
```

В этом примере мы вызываем метод `Cancel` источника токена отмены до начала цикла. Если запустить этот код, то он выведет "Получил 1" и только потом выбросит исключение `TaskCanceledException`. Почему он выполнил первую итерацию, если операция уже была отменена до ее начала?

Напомню, что `CancellationToken`, как обсуждалось в главе 9, — это всего лишь потокобезопасный флаг, который можно использовать для проверки необходимости отмены операции. В листинге 14.6 мы вообще не проверяем отмену в цикле, и в методе `AsyncYieldDemo` мы генерируем первое значение без проверки флага. Первая проверка происходит внутри вызова `Task.Delay`.

14.4. Другие варианты

В дополнение к методу `WithCancellation` есть метод `ToBlockingEnumerable`. Он заключает `IAsyncEnumerable<T>` в синхронный `IEnumerable<T>`, который можно использовать в обычном цикле `foreach` в синхронном коде. Метод `ToBlockingEnumerable` позволяет использовать асинхронный API из синхронного кода. Это эквивалентно вызову `Wait()` для каждой задачи, возвращаемой методом `MoveNextAsync`.

Метод `ToBlockingEnumerable`, как и другие способы вызова `Wait()`, сводит на нет преимущества асинхронного программирования и в некоторых случаях может вызывать состояние взаимоблокировки. Его следует использовать, только когда асинхронная коллекция используется в синхронном коде и нет другого выбора.

Наконец, есть еще один метод расширения — `ConfigureAwait`. Вызов `ConfigureAwait` для объекта `IAsyncEnumerable<T>` эквивалентен вызову `ConfigureAwait` для всех задач, возвращаемых методом `MoveNextAsync`.

Метод `ConfigureAwait` позволяет решить, будет ли код после `await` выполняться в том же контексте, что и до `await`. Обычно такая потребность возникает только в локальных приложениях с графическим интерфейсом (подробности ищите в главе 11).

14.5. `IAsyncEnumerable<T>` и LINQ

LINQ — это механизм в C#, позволяющий использовать SQL-подобные операторы (например, `Select` и `Where`) для преобразования любой последовательности элементов (обычно коллекций .NET). LINQ использует интерфейс `IEnumerable<T>` для взаимодействия с последовательностью, которую вы преобразуете.

На момент написания этих строк последняя версия .NET (версия 9) не поддерживала LINQ в `IAsyncEnumerable<T>`. Однако группа разработчиков из .NET Reactive Extensions (Rx.NET) опубликовала библиотеку `System.Linq.Async` (доступную через NuGet), добавляющую поддержку всех операторов LINQ

в `IAsyncEnumerable<T>` (и, соответственно, во всех асинхронных коллекциях и последовательностях).

Если в будущем разработчики .NET добавят встроенную поддержку асинхронного механизма LINQ, то, скорее всего, они используют библиотеку `System.Linq.Async` от команды Rx.NET (сам `IAsyncEnumerable<T>` изначально был написан командой RX) или, по крайней мере, сделают встроенную поддержку LINQ совместимой с `System.Linq.Async`.

14.6. Пример: итерации по данным, получаемым асинхронно

Допустим, нам нужно обработать двоичный поток с числами. Напишем два метода. Один будет читать поток и извлекать числа (используя `yield return`), а другой — обрабатывать полученные числа. В роли потока, например, может быть файл или сетевое соединение. Для простоты начнем с синхронной версии (листинг 14.7).

Листинг 14.7. Синхронное чтение потока чисел

```
public class NumbersProcessor
{
    private I Enumerable<int> GetNumbers(Stream stream)
    {
        var buffer = new byte[4];
        while(stream.Read(buffer, 0, 4) == 4) ← Извлекает 4 байта
        {
            var number = BitConverter.ToInt32(buffer); ← Преобразует их в значение int
            yield return number; ← Возвращает
        }                                         значение int
    }

    public void ProcessStream(Stream stream)
    {
        foreach(var number in GetNumbers(stream)) ← Для каждого числа в потоке
        {
            Console.WriteLine(number); ← Обрабатывает число
        }
    }
}
```

Первый метод, `GetNumbers`, читает поток и выдает последовательность чисел. Он останавливается, когда не может получить следующее целое число. Второй метод, `ProcessStream`, использует первый, а затем что-то делает с числами (поскольку это всего лишь пример, мы просто выводим их на консоль).

Как уже говорилось ранее в этой книге, такие операции, как чтение из файла или канала связи, лучше выполнять асинхронно. Поэтому вспомним все, о чем говорилось в этой главе, и сделаем код асинхронным (листинг 14.8).

Листинг 14.8. Асинхронное чтение потока чисел

```

public class AsyncNumbersProcessor
{
    private async IAsyncEnumerable<int> GetNumbers(Stream stream)
    {
        var buffer = new byte[4];
        while(await stream.ReadAsync(buffer, 0, 4) == 4)
        {
            var number = BitConverter.ToInt32(buffer);
            yield return number;
        }
    }

    public async Task ProcessStream(Stream stream)
    {
        await foreach(var number in GetNumbers(stream))
        {
            Console.WriteLine(number);
        }
    }
}

```

Вместо `IEnumerable<int>` метод объявляется как `async IAsyncEnumerable<int>`

Вместо `stream.Read` используется `await stream.ReadAsync`

Вместо `void` метод объявляется как `async Task`

Вместо `foreach` используется `await foreach`

Асинхронный код выглядит так же, как синхронный, за исключением слов `async` и `await`, добавленных в нескольких местах. Чтобы прочитать поток асинхронно, нужно вызвать `Stream.ReadAsync` вместо `Stream.Read`, что является важным изменением. Нужно также дождаться завершения вызова `ReadAsync`, поэтому перед вызовом `ReadAsync` добавлено ключевое слово `await`. Чтобы иметь возможность использовать `await`, нужно объявить метод асинхронным, а асинхронные методы не могут возвращать `IEnumerable<int>`, поэтому мы дополнитель но меняем тип возвращаемого значения на `IAsyncEnumerable<int>`.

Теперь, закончив изменение метода `GetNumbers`, перейдем к методу `ProcessStream`. Чтобы обработать `IAsyncEnumerable<int>`, возвращаемый методом `GetNumbers`, нужно заменить `foreach` на `await foreach`. Но `await foreach` можно использовать только в асинхронных методах, поэтому объявляем метод асинхронным и меняем тип возвращаемого значения с `void` на `Task` (мы говорили о проблемах с асинхронными методами `void` в конце главы 3).

14.7. Пример: асинхронная очередь, подобная `BlockingCollection<T>`

В предыдущей главе, в листинге 13.11, мы использовали `BlockingCollection<T>` для реализации очереди заданий с десятью рабочими потоками. `BlockingCollection<T>` имеет метод `GetConsumingEnumerable`, который позволяет вызывающему его коду использовать конструкцию `foreach` и получить простой и легкочитаемый код. Однако `BlockingCollection<T>` не поддерживает асинхронные операции.

В листинге 13.12 мы использовали `Channel<T>` для реализации асинхронной версии той же программы, но интерфейс `Channel<T>` не очень хорошо подходит для таких задач. Нам пришлось использовать бесконечный цикл для чтения элементов из очереди и исключение для сигнализации о том, что работа выполнена и больше элементов не будет.

Теперь, с появлением `IAsyncEnumerable<T>`, мы можем написать класс, реализующий аналог метода `GetConsumingEnumerable` из `BlockingCollection<T>`, но на основе `Channel<T>`. В этом примере реализованы только методы `Add` и `GetConsumingEnumerable` (которых достаточно, чтобы получить очередь задачий) (листинг 14.9).

Листинг 14.9. Асинхронная версия `BlockingCollection<T>.GetConsumingEnumerable`

```
public class ChannelAsyncCollection<T>
{
    private Channel<T> _channel = Channel.CreateUnbounded<T>();
    public void Add(T item)
    {
        _channel.Writer.TryWrite(item);
    }

    public void CompleteAdding()
    {
        _channel.Writer.Complete();
    }

    public async IAsyncEnumerable<T> GetAsyncConsumingEnumerable()
    {
        while (true)
        {
            T next;
            try
            {
                next = await _channel.Reader.ReadAsync();
            }
            catch (ChannelClosedException)
            {
                yield break;
            }
            yield return next;
        }
    }
}
```

Метод `Add` просто вызывает `TryWrite` объекта, пишущего в канал. `TryWrite` не должен давать сбой на неограниченных каналах, но в промышленном коде, вероятно, следует проверить значение, возвращаемое `TryWrite`, и выбросить исключение, если он вернул `false`.

Метод `GetAsyncConsumingEnumerable` немного сложнее. По сути, это просто цикл, вызывающий метод `ReadAsync` объекта, читающего из канала:

```
public async IAsyncEnumerable<T> GetAsyncConsumingEnumerable()
{
    while (true)
    {
        yield return await _channel.Reader.ReadAsync();
    }
}
```

Но этот код не определяет момент, когда новых данных больше не будет и можно завершить цикл. Когда данные заканчиваются, `ReadAsync` выбросит исключение. Мы должны перехватить это исключение и завершить итерирование:

```
public async IAsyncEnumerable<T> GetAsyncConsumingEnumerable()
{
    while (true)
    {
        try
        {
            yield return await _channel.Reader.ReadAsync();
        }
        catch (ChannelClosedException) ←
        {
            yield break;
        }
    }
}
```

Определяет, когда следует завершить итерации

Однако эта версия `GetAsyncEnumerable` не компилируется, потому что `yield return` нельзя использовать внутри блока `try`. Мы должны переместить `yield return` за его пределы и тогда получим код из листинга 14.7:

```
public async IAsyncEnumerable<T> GetAsyncConsumingEnumerable()
{
    while (true)
    {
        T next;
        try
        {
            next = await _channel.Reader.ReadAsync();
        }
        catch (ChannelClosedException)
        {
            yield break;
        }
        yield return next;
    }
}
```

Инструкция `yield return` вынесена за пределы блока `try`

Теперь, реализовав асинхронную коллекцию на основе каналов, мы можем использовать ее для создания очереди заданий. Это асинхронная адаптация очереди на основе `BlockingCollection<T>` из листинга 13.11 (листинг 14.10).

Листинг 14.10. Асинхронная очередь заданий с десятью потоками

```
ChannelAsyncCollection <int> asyncCollection =
    new ChannelAsyncCollection <int>();
Task[] workers = new Task[10];
for(int i=0; i<workers.Length; i++) ← Создает 10 рабочих потоков
{
    var threadNumber = i;
    workers[i] = Task.Run(async () =>
    {
        var rng = new Random((int)threadNumber);
        int count = 0;
        await foreach (var currentValue in
            asyncCollection.GetAsyncConsumingEnumerable())
        {
            Console.WriteLine($"поток {threadNumber} значение {currentValue}");
            Thread.Sleep(rng.Next(500));
            count++;
        }
        Console.WriteLine($"поток {threadNumber}, всего {count} элементов");
    });
}
for(int i=0;i<100;i++) ← Добавляет 100 элементов
    for(int i=0;i<10;i++) ← для обработки
    {
        asyncCollection.Add(i);
    }
    asyncCollection.CompleteAdding(); ← Сообщает, что элементов
    await Task.WhenAll(workers); ← больше не будет
    Ждет завершения всех потоков
```

Этот код создает `ChannelAsyncCollection<int>` для хранения данных, которые нужно обработать в фоновом режиме. Затем запускает десять фоновых потоков для выполнения этой работы, и каждый поток использует `foreach` и `GetAsyncConsumingEnumerable` для извлечения элементов из очереди. Чтобы имитировать обработку, мы просто ждем небольшой случайный промежуток времени и выводим число. А для имитации обрабатываемых данных используем числа от 0 до 99.

Итоги главы

- Ключевые слова `yield return` и `yield break` можно использовать вместе с `async/await`. Для этого метод объявляется асинхронным и указывается тип возвращаемого значения `IAsyncEnumerable<T>` вместо `IEnumerable<T>`. После этого в методе, выполняющем итерации, можно использовать `await`.

- `IAsyncEnumerable<T>` и `IAsyncEnumerator<T>` являются асинхронными, `async/await`-совместимыми версиями `IEnumerable<T>` и `IEnumerator<T>`.
- Компилятор преобразует метод в класс, встретив инструкцию `yield return`, о которой рассказывалось в главе 2, и инструкцию `await`, обсуждавшуюся в главе 3.
- Для итераций по полученному объекту `IAsyncEnumerable<T>` вместо `foreach` следует использовать `await foreach`.
- Конструкция `await foreach` похожа на обычную конструкцию `foreach`, за исключением того, что выполняет `await` в каждой итерации.
- Итерации можно отменить с помощью метода расширения `WithCancellation`. Этот метод передает токен отмены в `IAsyncEnumerable<T>` (или, если `IAsyncEnumerable<T>` создан для инструкции `yield return`, то может передать его в метод, генерирующий последовательность). Не забывайте, что токен отмены — это просто флаг. Для остановки итераций должен иметься код, проверяющий статус токена и останавливающий итерирование.
- Применение метода расширения `ConfigureAwait` к `IAsyncEnumerable<T>` работает как вызов `Task.ConfigureAwait` в каждой итерации. Плюсы и минусы `ConfigureAwait` обсуждались в главе 11.
- Метод расширения `ToBlockingEnumerable` заключает `IAsyncEnumerable<T>` в `IEnumerable<T>`, который выполняет эквивалент вызова `Task.Wait` в каждой итерации. Как и `Task.Wait`, он может вызывать проблемы с производительностью и взаимоблокировкой. Его следует использовать только для вызова асинхронных API из синхронного кода и только если API поддерживает этот вариант использования.
- Асинхронные последовательности не имеют встроенной поддержки LINQ, но пакет `System.Linq.Async` из NuGet от команды Rx.NET добавляет такую поддержку.
- `yield return` и `await foreach` можно использовать в простом коде, который создает и обрабатывает последовательности асинхронно сгенерированных или полученных элементов данных (см. листинг 14.8).
- `yield return` и `await foreach` также можно использовать для создания асинхронных очередей заданий и другой многопоточной инфраструктуры (см. листинг 14.9).

Нир Добовицки

C# Concurrency. Асинхронное программирование и многопоточность

Перевел с английского А. Киселев

Руководитель дивизиона	<i>Ю. Сергиенко</i>
Руководитель проекта	<i>Н. Михеева</i>
Ведущий редактор	<i>Н. Гринчик</i>
Литературный редактор	<i>Н. Роцина</i>
Художественный редактор	<i>В. Мостапан</i>
Корректоры	<i>Т. Никифорова, Н. Терех</i>
Верстка	<i>Г. Блинов</i>

Изготовлено в России. Изготовитель: ООО «Прогресс книга».
Место нахождения и фактический адрес: 194044, Россия, г. Санкт-Петербург,
Б. Сампсониевский пр., д. 29А, пом. 52. Тел.: +78127037373.

Дата изготовления: 09.2025. Наименование: книжная продукция. Срок годности: не ограничен.

Налоговая льгота — общероссийский классификатор продукции ОК 034-2014, 58.11.12 — Книги печатные
профессиональные, технические и научные.

Импортер в Беларусь: ООО «ПИТЕР М», 220020, РБ, г. Минск, ул. Тимирязева, д. 121/3, к. 214, тел./факс: 208 80 01.

Подписано в печать 15.08.25. Формат 70×100/16. Бумага офсетная. Усл. п. л. 21,930. Тираж 700. Заказ 0000.



DotNetRu – это группа независимых сообществ .NET-разработчиков со всей России. Мы объединяем людей вокруг .NET-платформы, чтобы способствовать обмену опытом и знаниями. Мы проводим регулярные встречи, организуем конференции, записываем подкасты, выступаем консультантами и всегда рады новым участникам.



Сайт: DotNet.Ru