Асинхронное программирование в C# 5.0

Алекс Дэвис

ДМК Издательство

O'REILLY®

Алекс Дэвис

Асинхронное программирование в C# 5.0

Async in C# 5.0

Alex Davies

O'REILLY®
Beijing • Cambridge • Farnham • Köln • Sebastopol • Tokyo

Асинхронное программирование в C# 5.0

Алекс Дэвис

ДМК Пресс
издательство
Москва, 2013

УДК 004.438C#
ББК 32.973.26-018.2
Д94

Д94 Алекс Дэвис

Асинхронное программирование в C# 5.0. / Пер. с англ. Слинкин А. А. – М.: ДМК Пресс, 2013. – 120 с.: ил.

ISBN 978-5-94074-886-1

Из этого краткого руководства вы узнаете, как механизм async в C# 5.0 позволяет упростить написание асинхронного кода. Помимо ясного введения в асинхронное программирование вообще, вы найдете углубленное описание работы этого конкретного механизма и ответ на вопрос, когда и зачем использовать его в собственных приложениях.

В книге рассматриваются следующие вопросы.

- Как писать асинхронный код вручную и как механизм async скрывает неприглядные детали.
- Новые способы повышения производительности серверного кода в приложениях ASP.NET.
- Совместная работа async и WinRT в приложениях для Windows 8.
- Смысл ключевого слова await в async-методах.
- В каком потоке .NET исполняется асинхронный код в каждой точке программы.
- Написание асинхронных API, согласованных с паттерном Task-based Asynchronous Pattern (TAP).
- Распараллеливание программ для задействования возможностей современных компьютеров.
- Измерение производительности async-кода и сравнение с альтернативными подходами.

Книга рассчитана на опытных программистов на C#, но будет понятна и начинающим. Она изобилует примерами кода, который можно использовать в своих программах.

ISBN 978-1-449-33716-2 (англ.)

ISBN 978-5-94074-886-1 (рус.)

ОГЛАВЛЕНИЕ

ПРЕДИСЛОВИЕ

Средства асинхронного программирования – мощный механизм, добавленный в версию 5.0 языка программирования C#. Это произошло как раз в тот момент, когда производительность и распараллеливание вызывают всё более пристальный интерес у разработчиков программного обеспечения. При правильном использовании новые средства позволяют создавать программы с такими характеристиками производительности и параллелизма, которых раньше можно было добиться только за счет написания объемного и громоздкого кода. Однако тема эта непроста, и у нее есть масса нюансов, не очевидных с первого взгляда.

Если не считать Visual Basic .NET, в который средства асинхронного программирования были добавлены одновременно с C#, то ни в одном из других широко распространенных языков программирования ничего эквивалентного нет[1]. Рекомендации по их применению в реальных программах найти трудно, опыт еще не наработан. В этой книге я хочу поделиться своей практикой работы, а также идеями, почерпнутыми у проектировщиков C# и из теоретической информатики. Но главное – я покажу, что такое механизм async, как он работает и почему его стоит использовать.

Предполагаемая аудитория

Эта книга рассчитана на программистов, уверенно владеющих языком C#. Быть может, вы хотите понять, что такое механизм async и стоит ли его использовать в своих программах. А, быть может, уже вовсю работаете с ним, но хотели бы познакомиться с передовыми приемами и узнать о подводных камнях.

Тем не менее, знакомство с другими продвинутыми средствами C# не предполагается, так что книга может быть полезна также начинающим программистам на C# и тем, кто пишет на других языках.

На C# разрабатываются самые разные приложения, и во всех них механизм async может найти применения – для различных целей. Поэтому в этой книге мы будем рассматривать как клиентские, так

1 Это чрезмерно категоричное утверждение оставим на совести автора. – *Прим. перев.*

и серверные программы и посвятим отдельные главы технологиям ASP.NET и WinRT.

Как читать эту книгу

Предполагается, что вы будете читать книгу последовательно, от начала до конца – для того чтобы познакомиться с механизмом async. Новые идеи излагаются по порядку и иллюстрируются на примерах. В особенности это относится к первым пяти главам.

Лучший способ чему-то научиться – практиковаться, поэтому я рекомендую выполнять все примеры. Для этого вам понадобится какая-нибудь среда разработки на C#, например Microsoft Visual Studio или MonoDevelop. Не упускайте возможность развить примеры и применить полученные знания в собственных программах – так вы сможете глубже усвоить материал.

Прочитав всю книгу, вы, возможно, захотите использовать главы, начиная с шестой, как справочник по продвинутым способам работы с async. Эти главы относительно независимы.

- В главах 6 и 7 рассматриваются приемы, применяемые при программировании с помощью async.
- Главы 8 и 9 посвящены нетривиальным аспектам механизма async.
- В главах с 10 по 13 обсуждаются ситуации, в которых механизм async может быть полезен.
- В главах 14 и 15 речь идет о внутреннем устройстве async.

Принятые соглашения

В этой книге используются следующие графические выделения:

Курсив

Таким шрифтом набраны новые термины, URL-адреса, адреса электронной почты, имена и расширения файлов.

`Моноширинный шрифт`

Применяется для набора листингов, а также внутри основного текста для обозначения элементов программы: имен переменных и функций, баз данных, типов данных, переменных окружения, языковых конструкций и ключевых слов.

`Моноширинный курсив`

Так набран текст, вместо которого нужно подставить задаваемые пользователем или зависящие от контекста значения.

Таким значком обозначаются советы, предложения и замечания общего характера.

О примерах кода

Эта книга призвана помогать вам в работе. Поэтому вы можете использовать приведенный в ней код в собственных программах и в документации. Спрашивать у нас разрешение необязательно, если только вы не собираетесь воспроизводить значительную часть кода. Например, никто не возбраняет включить в свою программу несколько фрагментов кода из книги. Однако для продажи или распространения примеров на компакт-диске разрешение требуется. Цитировать книгу и примеры в ответах на вопросы можно без ограничений. Но для включения значительных объемов кода в документацию по собственному продукту нужно получить разрешение.

Мы высоко ценим, хотя и не требуем, ссылки на наши издания. В ссылке обычно указываются название книги, имя автора, издательство и ISBN, например: «Async in C# 5.0 by Alex Davies O'Reilly). Copyright 2012 Alex Davies, 978-1-449-33716-2».

Если вы полагаете, что планируемое использование кода выходит за рамки изложенной выше лицензии, пожалуйста, обратитесь к нам по адресу *permissions@oreilly.com*.

Как с нами связаться

Вопросы и замечания по поводу этой книги отправляйте в издательство:

O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472
800-998-9938 (в США и Канаде)
707-829-0515 (международный или местный)
707-829-0104 (факс)

Для этой книги имеется веб-страница, на которой публикуются списки замеченных ошибок, примеры и прочая дополнительная информация. Адрес страницы: *http://oreil.ly/Async_in_CSharp5*.

Замечания и вопросы технического характера следует отправлять по адресу *bookquestions@oreilly.com*.

Дополнительную информацию о наших книгах, конференциях и новостях вы можете найти на нашем сайте по адресу *http://www.oreilly.com*.

Читайте нас на Facebook: *http://facebook.com/oreilly*.

Следите за нашей лентой в Twitter: *http://twitter.com/oreillymedia*.

Смотрите нас на YouTube: *http://www.youtube.com/oreillymedia*.

Благодарности

Я благодарен Стивену Тоубу (Stephen Toub) не только за техническое редактирование, но и за ценные сведения о некоторых аспектах параллельных вычислений. Именно из его блога я впервые узнал о многих идеях, которые объясняю на страницах этой книги. Спасибо Хэмишу за корректуру и Кэти за то, что она подавала мне чай во время работы.

Я также благодарен Рэчел Румелиотис (Rachel Roumeliotis), моему редактору, и всему коллективу издательства O'Reilly, оказывавшему мне неоценимую помощь в работе над книгой.

Я признателен своей семье, особенно маме, которая ухаживала за мной после операции, когда и была написана большая часть текста. И наконец, благодарю своих коллег по компании Red Gate, которые создали благоприятствующую экспериментам атмосферу, в которой у меня и зародилось желание исследовать механизм async на практике.

Об авторе

Алекс Дэвис – программист, блогер и энтузиаст параллельного программирования. Проживает в Англии. В настоящее время является разработчиком и владельцем продукта в компании Red Gate, занимающейся созданием инструментария для платформы .NET. Получил ученую степень по информатике в Кэмбриджском университете и до сих пор теоретическая информатика у него в крови. В свободное время разрабатывает каркас Actors с открытым исходным кодом для .NET, цель которого – упростить написание параллельных программ.

ГЛАВА 1. Введение

Начнем с общего введения в средства асинхронного программирования (или просто *async*) в C# 5.0.

Асинхронное программирование

Код называется асинхронным, если он запускает какую-то длительную операцию, но не дожидается ее завершения. Противоположностью является блокирующий код, который ничего не делает, пока операция не завершится.

К числу таких длительных операций можно отнести:

- сетевые запросы;
- доступ к диску;
- продолжительные задержки.

Основное различие заключается в том, в каком *потоке* выполняется код. Во всех популярных языках программирования код работает в контексте какого-то потока операционной системы. Если этот поток продолжает делать что-то еще, пока выполняется длительная операция, то код асинхронный. Если поток в это время ничего не делает, значит, он заблокирован и, следовательно, вы написали блокирующий код.

Разумеется, есть и третья стратегия ожидания результата длительной операции – *опрос*. В этом случае вы периодически интересуетесь, завершилась ли операция. Для очень коротких операций такой способ иногда применяется, но в общем случае эта идея неудачна.

Вполне возможно, что вы уже применяли асинхронный код в своих программах. Всякий раз, запуская новый поток или пользуясь классом `ThreadPool`, вы пишете асинхронную программу, потому что текущий поток может продолжать заниматься другими вещами. Если вам доводилось создавать веб-страницы, из которых пользова-

тель может обращаться к другим страницам, то такой код был асинхронным, потому, что внутри веб-сервера нет потока, ожидающего, когда пользователь закончит ввод данных. Кажется очевидным, но вспомните о консольном приложении, которое запрашивает данные от пользователя с помощью метода `Console.ReadLine()`, и вам станет понятно, как мог бы выглядеть альтернативный блокирующий дизайн веб-приложений. Да, такой дизайн был бы кошмаром, но всё же он возможен.

Для асинхронного кода характерна типичная трудность: как узнать, когда операция завершилась? Ведь только после этого можно приступить к обработке ее результатов. В блокирующем коде все тривиально – следующая строка помещается сразу после вызова длительной операции. Но в асинхронном мире так сделать нельзя, потому что размещенная в этом месте строка почти наверняка будет выполнена раньше, чем асинхронная операция завершится.

Для решения этой проблемы придуман целый ряд приемов, позволяющих выполнить код по завершении фоновой операции:

- включить нужный код в состав самой операции, после кода, составляющего ее основное назначение;
- подписаться на событие, генерируемое по завершении;
- передать делегат или лямбда-функцию, которая должна быть выполнена по завершении (*обратный вызов*).

Если код, следующий за асинхронной операцией, необходимо выполнить в конкретном потоке (например, в потоке пользовательского интерфейса в программе на базе *WinForms* или *WPF*), то приходится ставить операцию в очередь этого потока. Всё это очень утомительно.

Чем так хорош асинхронный код?

Асинхронный код освобождает поток, из которого был запущен. И это очень хорошо по многим причинам. Прежде всего, потоки потребляют ресурсы компьютера, а чем меньше расходуется ресурсов, тем лучше. Часто существует лишь один поток, способный выполнить определенную задачу (например, поток пользовательского интерфейса) и, если не освободить его быстро, то приложение перестанет реагировать на действия пользователя. Мы еще вернемся к этой теме в следующей главе.

Но самым важным мне представляется тот факт, что асинхронное выполнение открывает возможность для параллельных вычислений.

Вы можете структурировать программу по-новому, реализовав мелкоструктурный параллелизм, но не жертвуя простотой и удобством сопровождения. Этот вопрос мы будем исследовать в главе 10.

Что такое async?

В версии C# 5.0 Microsoft добавила механизм, предстающий в виде двух новых ключевых слов: `async` и `await`.

Этот механизм опирается на ряд нововведений в .NET Framework 4.5, без которых был бы бесполезен.

Механизм async встроен в компилятор и без поддержки с его стороны не мог бы быть реализован в библиотеке. Компилятор преобразовывает исходный код, то есть действует примерно по тому же принципу, что в случае лямбда-выражений и итераторов в предыдущих версиях C#.

Эта возможность существенно упрощает *асинхронное* программирование, избавляя от необходимости использовать сложные приемы, как то было в предыдущих версиях языка. С ее помощью можно написать всю программу целиком в асинхронном стиле.

В этой книгу я буду называть словом **асинхронный** общий стиль программирования, упростившийся после появления в C# механизма **async**. На C# всегда можно было писать асинхронные программы, но это требовало значительных усилий со стороны программиста.

Что делает async?

Механизм async дает простой способ выразить, что должна делать программа по завершении длительной асинхронной операции. Метод, помеченный ключевым словом `async`, компилятор преобразует так, что асинхронный код выглядит очень похоже на блокирующий эквивалент. Ниже приведен простой пример блокирующего метода для загрузки веб-страницы.

```
private void DumpWebPage(string uri)
{
 WebClient webClient = new WebClient();
 string page = webClient.DownloadString(uri);
 Console.WriteLine(page);
}
```

А вот эквивалентный ему асинхронный метод.

```
private async void DumpWebPageAsync(string uri)
{
 WebClient webClient = new WebClient();
 string page = await webClient.DownloadStringTaskAsync(uri);
 Console.WriteLine(page);
}
```

Похожи, не правда ли? Но под капотом они сильно отличаются. Второй метод помечен ключевым словом `async`. Это обязательное условие для всех методов, в которых используется ключевое слово `await`. Еще мы добавили к имени метода суффикс `Async`, чтобы соблюсти общепринятое соглашение.

Наибольший интерес представляет ключевое слово `await`. Видя его, компилятор переписывает метод. Точная процедура довольно сложна, поэтому пока я приведу лишь ее не вполне корректное описание, которое, на мой взгляд, полезно для понимания простых случаев.

1. Весь код после `await` переносится в отдельный метод.
2. Новый вариант метода `DownloadString` называется `DownloadStringTaskAsync`. Он делает то же самое, что исходный, но асинхронно.
3. Это означает, что мы можем передать ему сгенерированный метод, который будет вызываться по завершении операции. Делается это с помощью некоторых магических манипуляций, о которых я расскажу ниже.
4. Когда загрузка страницы завершится, будет вызван наш код, которому передается загруженная строка `string`; в данном случае мы просто выводим ее на консоль.

```
private void DumpWebPageAsync(string uri)
{
 WebClient webClient = new WebClient();
 webClient.DownloadStringTaskAsync(uri) <- magic(SecondHalf);
}

private void SecondHalf(string awaitedResult)
{
 string page = awaitedResult;
 Console.WriteLine(page);
}
```

Что происходит в вызывающем потоке, когда он исполняет такой код? По достижении вызова `DownloadStringTaskAsync` начинается загрузка страницы. Но не в текущем потоке. В нем вызванный метод

сразу возвращает управление. Что делать дальше, решает вызывающая программа. К примеру, поток пользовательского интерфейса мог бы продолжить обработку действий пользователя. Или просто завершиться, освободив ресурсы. В любом случае мы написали асинхронный код!

Async не решает все проблемы

Механизм async намеренно спроектирован так, чтобы максимально напоминать блокирующий код. Мы можем рассматривать длительные или удаленные операции, как будто они выполняются локально и быстро, увеличив производительность за счет асинхронности.

Однако он не дает вам совсем забыть о том, что на самом деле операция выполняется в фоновом режиме и происходит обратный вызов. Необходимо иметь в виду, что многие средства языка в асинхронном режиме ведут себя по-другому, частности:

- исключения и блоки try-catch-finally;
- возвращаемые методами значения;
- потоки и контекст;
- производительность.

Не зная, что происходит в действительности, вы не сможете ни понять смысл неожиданных сообщений об ошибках, ни воспользоваться отладчиком для их исправления.

ГЛАВА 2. Зачем делать программу асинхронной

Асинхронное программирование – вещь важная и полезная, но почему именно важная, зависит от вида приложения. Некоторые преимущества проявляются всегда, но особенно значимы в приложениях, которые вы, возможно, не имели в виду. Поэтому рекомендую прочитать эту главу, чтобы лучше понимать контекст в целом.

Приложения с графическим интерфейсом пользователя для настольных компьютеров

К приложениям для настольных компьютеров предъявляется одно важное требование – они должны реагировать на действия пользователя. Исследования в области человеко-машинного интерфейса показывают, что пользователь не обращает внимания на медленную работу программы, если она откликается на его действия и – желательно – имеет индикатор хода выполнения.

Но если программа зависает, то пользователь недоволен. Обычно зависания связаны с тем, что программа перестает реагировать на действия пользователя во время выполнения длительной операции, будь то медленное вычисление или операция ввода/вывода, например обращение к сети.

Все каркасы для организации пользовательского интерфейса в C# работают в одном потоке. Это относится и к WinForms, и к WPF, и к Silverlight. Только этот поток может управлять содержимым окна, распознавать действия пользователя и реагировать на них. Если он занят или блокирован дольше нескольких десятков миллисекунд, то пользователь сочтет, что приложение «тормозит».

Асинхронный код, даже написанный вручную, позволяет потоку пользовательского интерфейса вернуться к своей основной обязанности – опросу *очереди сообщений* и реагированию на появляющиеся в ней события. Он также может анимировать ход выполнения задачи, а в последних версиях Windows еще и наведение мыши на различные объекты. То и другое служит для пользователя наглядным подтверждением того, что программа работает.

Наличие только одного потока пользовательского интерфейса позволяет упростить синхронизацию. Если бы таких потоков было много, то один мог бы попытаться получить ширину кнопки в момент, когда другой занят размещением элементов управления. Чтобы избежать конфликтов, пришлось бы повсеместно расставлять блокировки; при этом производительность оказалась бы не лучше, чем в случае единственного потока.

Аналогия с кафе

Чтобы помочь вам разобраться, прибегну к аналогии. Если вы полагаете, что и так всё понимаете, можете спокойно пропустить этот раздел.

Представьте себе небольшое кафе, которое предлагает тосты на завтрак. Функции официантки выполняет сам хозяин. Он очень серьезно относится к качеству обслуживания клиентов, но об асинхронной обработке слыхом не слыхивал.

Поток пользовательского интерфейса как раз и моделирует действия хозяина кафе. Если в компьютере вся работа выполняется потоками, то в кафе – обслуживающим персоналом. В данном случае персонал состоит из одного-единственного человека, которого можно уподобить единственному потоку пользовательского интерфейса.

Первый посетитель заказывает тост. Хозяин берет ломтик хлеба, включает тостер и ждет, пока тот поджарит хлеб. Посетитель спрашивает, где взять масло, но хозяин его игнорирует – он всецело поглощен наблюдением за тостером, иначе говоря – блокирован. Через пять минут тост готов, и хозяин подает его посетителю. К этому моменту уже скопилась очередь, а посетитель обижен, что на него не обращают внимания. Ситуация далека от идеала.

Посмотрим, нельзя ли научить хозяина кафе действовать асинхронно.

Во-первых, необходимо, чтобы сам тостер мог работать асинхронно. При написании асинхронного кода мы должны позаботиться о

том, чтобы запущенная нами длительная операция могла выполнить обратный вызов по завершении. Точно так же, у тостера должен быть таймер, а поджаренный хлеб должен выскакивать с громким звуком, так чтобы хозяин заметил это.

Теперь хозяин может включить тостер и на время забыть о нем, вернувшись к обслуживанию клиентов. Точно так же, наш асинхронный код должен возвращать управление сразу после запуска длительной операции, чтобы поток пользовательского интерфейса мог реагировать на действия пользователя. Тому есть две причины:

- у пользователя остается впечатление, что интерфейс «отзывчивый», – клиент может попросить масло, и его не проигнорируют;
- пользователь может одновременно начать другую операцию – следующий клиент может изложить свой заказ.

Теперь хозяин кафе может одновременно обслуживать нескольких клиентов; единственным ограничением является количество тостеров и время, необходимое для подачи готового теста. Однако при этом возникают новые проблемы: необходимо помнить, кому какой тост предназначен. На самом деле, поток пользовательского интерфейса, вернувшись к обслуживанию очереди событий, ничего не помнит о том, завершения каких операций он ждет.

Поэтому мы должны связать с запускаемой задачей обратный вызов, который известит нас о том, что задача завершилась. Хозяину кафе достаточно прикрепить к тосту листочек с именем клиента. Нам же может потребоваться нечто более сложное, и в общем случае хотелось бы иметь возможность задавать произвольные инструкции, что делать по завершении задачи.

Последовав нашим советам, хозяин кафе стал работать полностью асинхронно, и его дело процветает. Довольны и клиенты. Ждать приходится меньше, и их просьбы больше не игнорируются. Надеюсь, эта аналогия помогла вам лучше понять, почему асинхронность так важна в приложениях с пользовательским интерфейсом.

Серверный код веб-приложения

У ASP.NET-приложений на веб-сервере нет ограничения на единственный поток, как в случае программ с пользовательским интерфейсом. И тем не менее асинхронное выполнение может оказаться весьма полезным, так как для таких приложений характерны длительные операции, особенно запросы к базе данных.

В зависимости от версии IIS может быть ограничено либо общее число потоков, обслуживающих веб-запросы, либо общее число одновременно обрабатываемых запросов. Если большая часть времени обработки запроса уходит на обращение к базе данных, то увеличение числа одновременно обрабатываемых запросов может повысить пропускную способность сервера.

Когда поток блокирован в ожидании какого-то события, он не потребляет процессорное время. Но не следует думать, что он вообще не потребляет ресурсы сервера. На самом деле, любой поток, даже блокированный, потребляет два ценных ресурса.

Память

В Windows для каждого управляемого потока резервируется примерно один мегабайт виртуальной памяти. Если количество потоков исчисляется десятками, то это не страшно, но когда их сотни, то может превратиться в проблему. Если операционная система вынуждена выгружать память на диск, то возобновление потока резко замедляется.

Ресурсы планировщика

Планировщик операционной системы отвечает за выделение потокам процессоров. Планировщик должен рассматривать даже заблокированные потоки, иначе он не будет знать, когда они разблокируются. Это замедляет контекстное переключение, а, значит, и работу системы в целом.

В совокупности эти накладные расходы означают дополнительную нагрузку на сервер, а, стало быть, увеличивают задержку и снижают пропускную способность.

Помните: основная отличительная особенность асинхронного кода состоит в том, что поток, начавший длительную операцию, освобождается для других дел. В случае ASP.NET этот поток берется из пула потоков, поэтому после запуска длительной операции он сразу же возвращается в пул и может затем использоваться для обработки других запросов. Таким образом, для обработки одного и того же количества запросов требуется меньше потоков.

Еще одна аналогия: кухня в ресторане

Веб-сервер можно рассматривать как модель ресторана. Есть много клиентов, заказывающих еду, а кухня пытается обслужить все заказы как можно быстрее.

На нашей кухне много поваров, каждый из которых можно уподобить потоку. Повар готовит заказанные блюда, но в течение какого-то времени любое блюдо должно постоять на плите, а повару в этот момент делать нечего. Точно так же обстоит дело с веб-запросом, для обработки которого нужно обратиться к базе данных, – веб-сервер к этому отношения не имеет.

При блокирующей организации работ на кухне повар будет стоять у плиты, дожидаясь готовности блюда. Чтобы добиться точной аналогии с заблокированным потоком, которому не выделяется процессорное время, предположим, что повару ничего не платят за время, когда он ждет готовности. Быть может, в это время он читает газету.

Но даже если повару не нужно платить и для приготовления каждого блюда можно нанять нового повара, простаивающие в ожидании повара занимают место на кухне. В одну кухню не удастся впихнуть больше нескольких десятков поваров, иначе в ней будет трудно перемещаться, и вся работа станет.

Разумеется, асинхронная система работает куда лучше. Ставя блюдо на плиту или в духовку, повар помечает, что это за блюдо и на какой стадии приготовления оно находится, а затем начинает заниматься другим делом. Когда подойдет время снимать блюдо с плиты, это сможет сделать любой повар, – он и продолжит его готовить.

Этот подход может быть эффективно применен в веб-серверах. Несколько потоков могут справиться с обработкой того же количества одновременных запросов, для которого раньше требовались сотни потоков (а то и вообще не удавалось обработать). На самом деле, в некоторых каркасах для создания веб-серверов, например в *node.js*, отвергается сама идея о наличии нескольких потоков, и все запросы асинхронно обрабатываются единственным потоком. Зачастую при этом удается обработать больше запросов, чем может многопоточная, но блокирующая система. Точно так же, единственный повар в пустой кухне, хорошо организовавший свою работу, сможет приготовить больше блюд, чем сотня поваров, которые только и делают, что мешают друг другу или почитывают газетку.

Silverlight, Windows Phone и Windows 8

Проектировщики Silverlight, безусловно, знали о преимуществах асинхронного кода в приложениях с пользовательским интерфейсом, поэтому решили поощрить такой подход. Для этого они просто

изъяли из каркаса большую часть синхронных API. Так, например, веб-запросы можно отправлять только асинхронно.

Асинхронный код «заразен». Если где-то вызвать какой-нибудь асинхронный API, то и вся программа естественно становится асинхронной. Поэтому в Silverlight вы *обязаны* писать асинхронный код – альтернативы просто не существует. Возможно, и существует метод `Wait` или иной способ работать с асинхронным API синхронно, приостановив выполнение в ожидании обратного вызова. Но, поступая так, вы теряете все преимущества, о которых я говорил выше.

Silverlight для Windows Phone, как следует из названия, является разновидностью Silverlight. Правда, включены дополнительные API, небезопасные в среде браузера, например TCP-сокеты. Но и в этом случае предоставляются только асинхронные версии API, что заставляет вас писать асинхронный код. И уж для мобильного устройства, располагающего крайне ограниченными ресурсами, это вдвойне оправдано. Запуск дополнительных потоков весьма негативно отражается на времени работы аккумулятора.

И в Windows 8, технически не связанной с Silverlight, подход такой же. Количество различных API в этом случае гораздо больше, но для методов, которые могут выполняться дольше 50 мс, предоставляются только асинхронные версии.

Параллельный код

Современные компьютеры оснащаются несколькими процессорными ядрами, работающими независимо друг от друга. Желательно, чтобы программа могла задействовать имеющиеся ядра, но два ядра не могут писать в одну и ту же ячейку памяти, так как это чревато повреждением ее содержимого.

Быть может, было бы лучше применять *чистое* (иначе говоря, функциональное) программирование, при котором не существует побочных эффектов, то есть состояние памяти не изменяется. Это помогло бы полнее воспользоваться преимуществами параллелизма, но для некоторых программ неприемлемо. Пользовательским интерфейсам состояние необходимо. Базы данных сами по себе являются состоянием.

Стандартное решение предполагает использование взаимно исключающих блокировок (мьютексов) в случаях, когда несколько ядер потенциально могут обращаться к одной и той же ячейке памяти. Но тут есть свои проблемы. Часто бывает так, что программа захватывает

блокировку, а затем вызывает метод или генерирует событие, в котором захватывается другая блокировка. Иногда удерживать сразу две блокировки необязательно, но так код оказывается проще. В результате другим потокам приходится ждать освобождения блокировки, хотя они могли бы в это время заниматься полезной работой. Хуже того, иногда возникает ситуация, когда каждый поток ждет освобождения блокировки, занятой другим, а это приводит к взаимоблокировке (deadlock). Такие ошибки трудно предвидеть, воспроизводить и исправлять.

Одно из самых многообещающих решений – модель вычислений, основанная на *акторах*. При таком подходе каждый участок записываемой памяти принадлежит ровно одному актору. Единственный способ записать в эту память – отправлять актору-владельцу сообщения, которые он обрабатывает поочередно и, возможно, посылает сообщения в ответ. Но это как раз и есть асинхронное программирование. Запрос действия у актора – типичная асинхронная операция, поскольку мы можем заниматься другими вещами, пока не придет ответное сообщение. А значит, для программирования такой модели можно использовать механизм async, как мы и увидим в главе 10.

Пример

Рассмотрим пример приложения для ПК, которое откровенно нуждается в переходе на асинхронный стиль. Его исходный код можно скачать по адресу https://bitbucket.org/alexdavies74/faviconbrowser. Рекомендую сделать это (если вы не пользуетесь системой управления версиями Mercurial, то код можно скачать в виде zip-файла) и открыть решение в Visual Studio. Необходимо скачать ветвь `default`, содержащую синхронную версию.

Запустив программу, вы увидите окно с кнопкой. Нажатие этой кнопки приводит к отображению значков нескольких популярных сайтов. Для этого программа скачивает файл `favicon.ico`, присутствующий на большинстве сайтов (рис. 2.1).

Приглядимся к коду. Наиболее важная его часть – метод, который загружает значок и добавляет его на панель WrapPanel (отметим, что это приложение WPF).

```
private void AddAFavicon(string domain)
{
 WebClient webClient = new WebClient();
 byte[] bytes = webClient.DownloadData("http://"+domain+"/favicon.ico");
```

```
    Image imageControl = MakeImageControl(bytes);
    m_WrapPanel.Children.Add(imageControl);
  }
```

Рис. 2.1. Обозреватель значков сайтов

Обратите внимание, что эта реализация полностью синхронна. Поток блокируется на время скачивания значка. Вероятно, вы заметили, что в течение нескольких секунд после нажатия кнопки окно ни на что не реагирует. Как вы теперь понимаете, объясняется это тем, что поток пользовательского интерфейса блокирован и не обрабатывает события, генерируемые действиями пользователя.

В следующих главах мы преобразуем эту синхронную программу в асинхронную.

ГЛАВА 3. Написание асинхронного кода вручную

В этой главе мы поговорим о написании асинхронного кода без помощи со стороны C# 5.0 и async. В некотором смысле это обзор техники, которую вы вряд ли будете когда-либо использовать, но знать о ней надо, чтобы понимать, что происходит за кулисами. Поэтому я не стану задерживаться надолго, а только отмечу моменты, необходимые для понимания.

О некоторых асинхронных паттернах в .NET

Как я уже отмечал, Silverlight предоставляет только асинхронные версии таких API, как доступ к сети. Вот, например, как можно было бы скачать и отобразить веб-страницу:

```
private void DumpWebPage(Uri uri)
{
 WebClient webClient = new WebClient();
 webClient.DownloadStringCompleted += OnDownloadStringCompleted;
 webClient.DownloadStringAsync(uri);
}

private void OnDownloadStringCompleted(object sender,
  DownloadStringCompletedEventArgs eventArgs)
{
 m_TextBlock.Text = eventArgs.Result;
}
```

API такого вида называется *асинхронным событийно-управляемым паттерном* (Event-based Asynchronous Pattern – EAP). Идея в том, чтобы вместо одного синхронного метода, который блокирует исполнение программы до тех пор, пока не будет скачана вся стра-

ница, используется один метод и одно событие. Сам метод выглядит, как синхронный аналог, только не возвращает никакого значения. Аргументом же события является специальный подкласс `EventArgs`, который содержит скачанную строку.

Перед вызовом метода мы подписываемся на событие. Метод, будучи асинхронным, возвращает управление немедленно. Затем в какой-то момент генерируется событие, которое мы можем обработать.

Очевидно, что этот паттерн не слишком удобен и не в последнюю очередь из-за того, что приходится разбивать простую последовательность команд на два метода. Не упрощает дела и тот факт, что нужно подписаться на событие. Если вы захотите использовать тот же самый экземпляр `WebClient` для отправки другого запроса, то будете неприятно удивлены тем, что старый обработчик события все еще присоединен и будет вызван снова.

Еще один асинхронный паттерн, встречающийся в .NET, подразумевает использование интерфейса `IAsyncResult`. В качестве примера упомянем метод класса `Dns`, который ищет IP-адрес по имени хоста, – `BeginGetHostAddresses`. В этом случае предоставляются два метода: `Begin`*`MethodName`* начинает операцию, а `End`*`MethodName`* – обратный вызов, которому передается результат.

```
private void LookupHostName()
{
 object unrelatedObject = "hello";
 Dns.BeginGetHostAddresses("oreilly.com", OnHostNameResolved,
    unrelatedObject);
}

private void OnHostNameResolved(IAsyncResult ar)
{
 object unrelatedObject = ar.AsyncState;
 IPAddress[] addresses = Dns.EndGetHostAddresses(ar);
 // Обработать адрес
 ...
}
```

Этот вариант, по крайней мере, не страдает от проблем, связанных с присоединенными ранее обработчиками событий. Но неоправданная сложность API – два метода вместо одного – все равно осталась, и мне это кажется неестественным.

В обоих случаях приходится разбивать логически единую процедуру на два метода. Интерфейс `IAsyncResult` позволяет передавать данные из первого метода во второй, что я продемонстрировал на при-

мере строки "hello". Но сделано это неудобно, поскольку вы должны передать что-то, даже если это не нужно, и выполнить приведение к нужному типу из object. Паттерн EAP также поддерживает передачу объекта – и столь же неэлегантно.

Передача контекста из одного метода в другой – общая проблема, свойственная всем асинхронным паттернам. В следующем разделе мы увидим, что решение дают лямбда-функции, которые можно использовать в обоих случаях.

Простейший асинхронный паттерн

Пожалуй, проще всего реализовать асинхронное поведение без использования async, передав обратный вызов в качестве параметра метода:

```
void GetHostAddress(string hostName, Action<IPAddress> callback)
```

Мне кажется, что с точки зрения простоты он превосходит все остальные варианты.

```
private void LookupHostName()
{
 GetHostAddress("oreilly.com", OnHostNameResolved);
}

private void OnHostNameResolved(IPAddress address)
{
 // Обработать адрес
 ...
}
```

Как я уже отмечал, вместо двух методов можно использовать в качестве обратного вызова анонимный метод или лямбда-выражение. У такого подхода есть важное достоинство – возможность обращаться к переменным, объявленным в первой части метода.

```
private void LookupHostName()
{
 int aUsefulVariable = 3;
 GetHostAddress("oreilly.com", address =>
   {
     // Сделать что-то с адресом и переменной aUsefulVariable
     ...
   });
}
```

Правда, лямбда-выражения несколько сложновато читать, а если используется несколько асинхронных API, то потребуется много вложенных лямбда-выражений. Поэтому количество уровней отступа быстро возрастает, и с кодом становится трудно работать.

Недостаток этого простого подхода заключается в том, что вызывающая программа больше не получает исключений. В паттернах, используемых в .NET, вызов метода `End*MethodName*` или чтение свойства `Result` приводит к повторному возбуждению исключения, которое ваш код может обработать. В противном случае исключение возникнет в неожиданном месте или останется необработанным вовсе.

Введение в класс Task

Библиотека Task Parallel Library была включена в версию .NET Framework 4.0. Важнейшим в ней является класс `Task`, представляющий выполняемую операцию. Его универсальный вариант, `Task<T>`, играет роль обещания вернуть значение (типа `T`), когда в будущем, по завершении операции, оно станет доступно.

Как мы увидим ниже, механизм async в C# 5.0 активно пользуется классом `Task`. Но и без async классом `Task`, а особенно его вариантом `Task<T>`, можно воспользоваться при написании асинхронных программ. Для этого нужно запустить операцию, которая возвращает `Task<T>`, а затем вызвать метод `ContinueWith` для регистрации обратного вызова.

```
private void LookupHostName()
{
 Task<IPAddress[]> ipAddressesPromise =
    Dns.GetHostAddressesAsync("oreilly.com");
 ipAddressesPromise.ContinueWith(_ =>
   {
    IPAddress[] ipAddresses = ipAddressesPromise.Result;
    // Обработать адрес
    ...
   });
}
```

Достоинство `Task` в том, что теперь требуется вызвать только один метод класса `Dns`, в результате чего API становится чище. Вся логика, относящаяся к обеспечению асинхронного поведения, инкапсулирована в классе `Task` и дублировать ее в каждом асинхронном методе нет необходимости. Этот класс умеет в частности обрабатывать исключения и работать с контекстами синхронизации `Synchroniza-`

tionContext. Они, как мы увидим в главе 8, полезны, когда требуется выполнить обратный вызов в конкретном потоке (например, в потоке пользовательского интерфейса).

Сверх того, класс Task позволяет абстрагировать работу с асинхронными операциями. Мы можем воспользоваться композицией для создания вспомогательных средств, которые работают с объектами Task и предоставляют поведение, полезное во многих ситуациях. Подробнее об этом речь пойдет в главе 7.

Чем плоха реализация асинхронности вручную?

Как мы видели, есть много способов реализовать асинхронную программу. Одни получше, другие похуже. Но надеюсь, вы заметили один общий недостаток. Процедуру необходимо разбить на два метода: запускающий операцию и обратный вызов. Использование в качестве обратного вызова анонимного метода или лямбда-выражения отчасти сглаживает проблему, зато получается код с большим числом уровней отступа, который трудно читать.

Существует и еще одна проблема. Мы говорили о методах, делающих один асинхронный вызов, но что если таких вызовов несколько? Или – еще хуже – если асинхронные методы требуется вызывать в цикле? Единственный выход – рекурсивный метод, но воспринимается такая конструкция куда сложнее, чем обычный цикл.

```
private void LookupHostNames(string[] hostNames)
{
 LookUpHostNamesHelper(hostNames, 0);
}

private static void LookUpHostNamesHelper(string[] hostNames, int i)
{
 Task<IPAddress[]> ipAddressesPromise =
    Dns.GetHostAddressesAsync(hostNames[i]);
 ipAddressesPromise.ContinueWith(_ =>
   {
    IPAddress[] ipAddresses = ipAddressesPromise.Result;
    // Обработать адрес
    ...
    if (i + 1 < hostNames.Length)
    {
     LookUpHostNamesHelper(hostNames, i + 1);
    }
   });
}
```

Фу, гадость какая!

Еще одна проблема, присущая всем рассмотренным способам, – использование написанного вами асинхронного кода. Если вы написали нечто асинхронно выполняемое и хотите использовать это в другом месте программы, то должны предоставить соответствующий асинхронный API. Но если использовать асинхронный API трудно и неудобно, то реализовать его трудно вдвойне. А поскольку асинхронный код «заразен», то иметь дело с асинхронными API придется не только непосредственно вызывающему его модулю, но и тому, который вызывает его, и так далее, пока вся программа не превратится в хаос.

Переработка примера с использованием написанного вручную асинхронного кода

Напомним, что в примере из предыдущей главы мы обсуждали WPF-приложение, которое не реагировало на действия пользователя, пока скачивало значки веб-сайтов; в это время поток пользовательского интерфейса был блокирован. Теперь посмотрим, как сделать это приложение асинхронным, применяя ручную технику.

Прежде всего, нужно найти асинхронную версию метода, которым я пользовался (`WebClient.DownloadData`). Как мы уже видели, в классе `WebClient` используется асинхронный событийно-управляемый паттерн (EAP), поэтому мы можем подписаться на событие обратного вызова и начать скачивание.

```
private void AddAFavicon(string domain)
{
 WebClient webClient = new WebClient();
 webClient.DownloadDataCompleted += OnWebClientOnDownloadDataCompleted;
 webClient.DownloadDataAsync(new Uri("http://" + domain + "/favicon.ico"));
}
 private void OnWebClientOnDownloadDataCompleted(object sender,
    DownloadDataCompletedEventArgs args)
 {
   Image imageControl = MakeImageControl(args.Result);
   m_WrapPanel.Children.Add(imageControl);
}
```

Разумеется, логически неделимую логику придется разбить на два метода. При работе с паттерном EAP я предпочитаю не использовать лямбда-выражение, потому что оно появилось бы в тексте до вызова

метода, в котором производится собственно скачивание, а мне такой код представляется нечитаемым.

Эта версия примера также доступна в сетевом репозитории – в ветви `manual`. Запустив ее, вы убедитесь, что не только пользовательский интерфейс продолжает реагировать на действия пользователя, но и значки появляются постепенно. Но тем самым мы внесли ошибку – поскольку все операции закачки выполняются параллельно, а не последовательно, то порядок следования значков зависит от скорости скачивания, а не от того, в каком порядке они запрашивались. Если вы хотите проверить, хорошо ли понимаете принципы ручного написания асинхронного кода, попробуйте исправить эту ошибку. Одно из возможных решений – находящееся в ветви `orderedManual` – основано на преобразовании цикла в рекурсивный метод. Существуют и более эффективные способы.

ГЛАВА 4. Написание асинхронных методов

Теперь мы знаем, какими выдающимися достоинствами обладает асинхронный код, но насколько трудно его писать? Самое время познакомиться с механизмом async в C# 5.0. Как мы видели в разделе «Что делает async?» в главе 1, метод, помеченный ключевым словом `async`, может содержать ключевое слово `await`.

```
private async void DumpWebPageAsync(string uri)
{
 WebClient webClient = new WebClient();
 string page = await webClient.DownloadStringTaskAsync(uri);
 Console.WriteLine(page);
}
```

Выражение `await` в этом примере приводит к преобразованию метода таким образом, что он приостанавливается на время скачивания и возобновляется по его завершении. В результате метод становится асинхронным. В этой главе мы научимся писать такие асинхронные методы.

Преобразование программы скачивания значков к виду, использующему async

Сейчас мы изменим программу обозревателя значков сайта, так чтобы воспользоваться механизмом async. Если есть такая возможность, откройте первый вариант примера (из ветви `default`) и попробуйте преобразовать его самостоятельно, добавив ключевые слова `async` и `await`, и только потом продолжайте чтение.

Наиболее важен метод `AddAFavicon`, который скачивает значок и отображает его в пользовательском интерфейсе. Мы хотим сделать

этот метод асинхронным, так чтобы поток пользовательского интерфейса продолжал реагировать на действия пользователя во время скачивания. Первый шаг – пометка метода ключевым словом `async`. Оно включается в сигнатуру метода точно так же, как, например, слово `static`.

Далее мы должны дождаться завершения скачивания, воспользовавшись ключевым словом `await`. С точки зрения синтаксиса C#, `await` – это унарный оператор, такой же как оператор `!` или оператор приведения типа (*type*). Он располагается слева от выражения и означает, что нужно дождаться завершения асинхронного выполнения этого выражения.

Наконец, вместо вызова метода `DownloadData` нужно вызвать его асинхронную версию, `DownloadDataTaskAsync`.

Метод, помеченный ключевым словом async, автоматически не становится асинхронным. Async-методы лишь упрощают использование других асинхронных методов. Они начинают исполняться синхронно, и так происходит до тех пор, пока не встретится вызов асинхронного метода внутри оператора `await`. В этот момент сам вызывающий метод становится асинхронным. Если же оператор `await` не встретится, то метод так и будет выполняться синхронно до своего завершения.

```
private async void AddAFavicon(string domain)
{
 WebClient webClient = new WebClient();
 byte[] bytes = await webClient.DownloadDataTaskAsync("http://" + domain +
    "/favicon.ico");
 Image imageControl = MakeImageControl(bytes);
 m_WrapPanel.Children.Add(imageControl);
}
```

Сравните этот вариант кода с двумя предыдущими. По структуре он куда ближе к исходной синхронной версии. Никакого дополнительного метода нет, добавлено лишь немного кода в теле метода. Однако ведет он себя, как асинхронная версия, представленная в разделе «Переработка примера с использованием написанного вручную асинхронного кода».

Task и await

Рассмотрим подробнее написанное выше выражение `await`. Вот как выглядит сигнатура метода `WebClient.DownloadStringTaskAsync`:

```
Task<string> DownloadStringTaskAsync(string address)
```

Тип возвращаемого значения – Task<string>. В разделе «Введение в класс Task» я говорил, что класс Task представляет выполняемую операцию, а его подкласс Task<T> – операцию, которая в будущем вернет значение типа T. Можно считать, что Task<T> – это обещание вернуть значение типа T по завершении длительной операции.

Оба класса Task и Task<T> могут представлять асинхронные операции, и оба умеют вызывать ваш код по завершении операции. Чтобы воспользоваться этой возможностью вручную, необходимо вызвать метод ContinueWith, передав ему код, который должен быть выполнен, когда длительная операция завершится. Именно так и поступает оператор await, чтобы выполнить оставшуюся часть async-метода.

Если применить await к объекту типа Task<T>, то мы получим *выражение await*, которое само имеет тип T. Это означает, что результат оператора await можно присвоить переменной, которая используется далее в методе, что мы и видели в примерах. Однако если await применяется к объекту неуниверсального класса Task, то получается *предложение await*, которое ничему нельзя присвоить (как и результат метода типа void). Это разумно, потому что класс Task не обещает вернуть значение в качестве результата, а представляет лишь саму операцию.

```
await smtpClient.SendMailAsync(mailMessage);
```

Ничто не мешает разбить выражение await на части и обратиться к Task напрямую до того, как приступать к ожиданию.

```
Task<string> myTask = webClient.DownloadStringTaskAsync(uri);
// Здесь можно что-то сделать
string page = await myTask;
```

Важно отчетливо понимать, что при этом происходит. В первой строке вызывается метод DownloadStringTaskAsync. Он начинает исполняться синхронно в текущем потоке и, приступив к скачиванию, возвращает объект Task<string> – все еще в текущем потоке. И лишь когда мы выполняем await для этого объекта, компилятор делает нечто необычное. Это относится и к случаю, когда оператор await непосредственно предшествует вызову асинхронного метода.

Длительная операция начинается, как только вызван метод DownloadStringTaskAsync, и это позволяет очень просто организовать одновременное выполнение нескольких асинхронных операций.

Достаточно просто запустить их по очереди, сохранить все объекты `Task`, а затем ждать их завершения с помощью `await`.

```
Task<string> firstTask =
  webClient1.DownloadStringTaskAsync("http://oreilly.com");
Task<string> secondTask =
  webClient2.DownloadStringTaskAsync("http://simpletalk.com");
string firstPage = await firstTask;
string secondPage = await secondTask;
```

Такой способ запуска нескольких задач `Task` ненадежен, если задача может возбуждать исключения. Если обе операции возбудят исключения, то первый `await` передаст исключение в вызывающую программу, а до ожидания задачи `secondTask` дело не дойдет вовсе. Возбужденное ей исключение останется незамеченным и в зависимости от версии и настроек .NET может быть проигнорировано или повторно возбуждено в другом потоке, где его никто не ожидает, что приведет к завершению процесса. В главе 7 мы рассмотрим более правильные способы параллельного исполнения нескольких задач.

Тип значения, возвращаемого асинхронным методом

Метод, помеченный ключевым словом `async`, может возвращать значения трех типов:

- `void`
- `Task`
- `Task<T>`, где `T` – некоторый тип.

Никакие другие типы не допускаются, потому что в общем случае исполнение асинхронного метода не завершается в момент возврата управления. Как правило, асинхронный метод ждет завершения длительной операции, то есть управление-то он возвращает сразу, но результат в этот момент еще не получен и, стало быть, недоступен вызывающей программе.

Я провожу различие между типом *возвращаемого* методом значения (например, `Task<string>`) и типом *результата*, который нужен вызывающей программе (в данном случае `string`). В обычных, не асинхронных, методах тип возвращаемого значения совпадает с типом результата, тогда как в асинхронных методах они различны – и это очень существенно.

Очевидно, что в асинхронном случае void – вполне разумный тип возвращаемого значения. Метод, описанный как async void, можно рассматривать как операцию вида «запустил и забыл». Вызывающая программа не ждет результата и не может узнать, когда и как операция завершается. Использовать тип void следует, когда вы уверены, что вызывающей программе безразлично, когда завершится операция и завершится ли она успешно. Чаще всего методы типа async void употребляются на границе между асинхронным и прочим кодом, например, обработчик события пользовательского интерфейса должен возвращать void.

Async-методы, возвращающие тип Task, позволяют вызывающей программе ждать завершения асинхронной операции и распространяют исключения, имевшие место при ее выполнении. Если значение результата несущественно, то метод типа async Task предпочтительнее метода типа async void, потому что вызывающая программа получает возможность узнать о завершении операции, что упрощает упорядочение задач и обработку исключений.

Наконец, async-методы, возвращающие тип Task<T>, например Task<string>, используются, когда асинхронная операция возвращает некий результат.

Async, сигнатуры методов и интерфейсы

Ключевое слово async указывается в объявлении метода, как public или static. Однако спецификатор async *не* считается частью сигнатуры метода, когда речь заходит о переопределении виртуальных методов, реализации интерфейса или вызове.

Единственное назначение ключевого слова async – изменить способ компиляции соответствующего метода, на взаимодействие с окружением оно не оказывает никакого влияния. Поэтому в правилах, действующих в отношении переопределения методов и реализации интерфейсов, слово async полностью игнорируется.

```
class BaseClass
{
 public virtual async Task<int> AlexsMethod()
 {
   ...
 }
}
```

```
class SubClass : BaseClass
{
 // Переопределяет метод базового класса AlexsMethod
 public override Task<int> AlexsMethod()
 {
   ...
 }
}
```

В объявлениях методов интерфейса слово `async` запрещено просто потому, что в этом нет необходимости. Если в интерфейсе объявлен метод, возвращающий тип `Task`, то в реализующем интерфейс классе этот метод может быть помечен словом `async`, а может быть и не помечен – на усмотрение программиста.

Предложение return в асинхронных методах

В асинхронном методе предложение `return` ведет себя особым образом. Напомним, что в обычном, не асинхронном, методе правила, действующие отношении `return`, зависят от типа, возвращаемого методом.

Методы типа `void`

Предложение `return` должно содержать только `return;` и может быть опущено.

Методы, возвращающие тип `T`

Предложение `return` должно содержать выражение типа `T` (например, `return 5+x;`); при этом любой путь исполнения метода должен завершаться предложением `return`.

В методах, помеченных ключевым словом `async`, действуют другие правила.

Методы типа `void` и методы, возвращающие тип `Task`

Предложение `return` должно содержать только `return;` и может быть опущено.

Методы, возвращающие тип `Task<T>`

Предложение `return` должно содержать выражение типа `T`; при этом любой путь исполнения метода должен завершаться предложением `return`.

В async-методах тип возвращаемого значения отличается от типа выражения, указанного в предложении `return`. Выполняемое компи-

лятором преобразование можно рассматривать как обертывание возвращенного значения объектом `Task<T>` до передачи его вызывающей программе. Конечно, в действительности объект `Task<T>` создается сразу же, но результат в него записывается позже, когда длительная операция завершится.

Асинхронные методы «заразны»

Как мы видели, чтобы воспользоваться объектом `Task`, возвращенным каким-то асинхронным API, проще всего дождаться его с помощью оператора `await` в async-методе. Но при таком подходе и ваш метод будет возвращать `Task` (как правило). Чтобы получить выигрыш от асинхронности, программа, вызывающая ваш метод, не должна блокироваться в ожидании завершении задачи `Task`, и, следовательно, вызов вашего метода, вероятно, также будет производиться с помощью `await`.

Ниже приведен пример вспомогательного метода, который подсчитывает количество символов на веб-странице и асинхронно возвращает результат.

```
private async Task<int> GetPageSizeAsync(string url)
{
 WebClient webClient = new WebClient();
 string page = await webClient.DownloadStringTaskAsync(url);
 return page.Length;
}
```

Чтобы им воспользоваться, я должен написать еще один async-метод, который возвращает результат также асинхронно:

```
private async Task<string> FindLargestWebPage(string[] urls)
{
 string largest = null;
 int largestSize = 0;
 foreach (string url in urls)
 {
   int size = await GetPageSizeAsync(url);

   if (size > largestSize)
   {
     size = largestSize;
     largest = url;
   }
 }

 return largest;
}
```

В итоге получается цепочка async-методов, каждый из которых ждет следующего с помощью оператора `await`. Асинхронная модель программирования «заразна», она с легкостью распространяется по всему коду продукта. Но я не считаю это серьезной проблемой – ведь писать асинхроннные методы очень легко.

Асинхронные анонимные делегаты и лямбда-выражения

Асинхронными могут быть как обычные именованные методы, так и обе формы анонимных методов. Вот как сделать асинхронным анонимный делегат:

```
Func<Task<int>> getNumberAsync = async delegate { return 3; };
```

А вот так – лямбда-выражение:

```
Func<Task<string>> getWordAsync = async () => “hello”;
```

При этом действуют те же правила, что для обычных асинхронных методов. Асинхронные анонимные методы можно применять для сокращения размера кода или для формирования замыканий – точно так же, как не асинхронные.

ГЛАВА 5. Что в действительности делает await

Воспринимать механизм async в C# 5.0 и, в частности, ключевое слово `await` можно двумя способами:

- как языковое средство с четко определенным поведением, которое можно изучить и применять;
- как преобразование программы на этапе компиляции, то есть *синтаксическую глазурь*, скрывающую более сложный код на C#, в котором ключевое слово async не используется.

Обе точки зрения верны, это всего лишь две стороны одной медали. В этой главе нас будет интересовать первый взгляд на async. В главе 14 мы взглянем на этот механизм под другим углом зрения; это сложнее, зато мы познакомимся с деталями, которые позволят с открытыми глазами заниматься отладкой и повышением производительности.

Приостановка и возобновление метода

Когда поток исполнения программы доходит до оператора `await`, должны произойти две вещи.

- Текущий поток должен быть освобожден, чтобы поведение программы было асинхронным. С привычной, синхронной, точки зрения это означает, что метод должен вернуть управление.
- Когда задача `Task`, погруженная в оператор `await`, завершится, ваш метод должен продолжить выполнение с того места, где перед этим вернул управление, как будто этого возврата никогда не было.

Чтобы добиться такого поведения, метод должен приостановить выполнение, дойдя до `await`, и возобновить его впоследствии.

Я рассматриваю эту последовательность событий как аналог режима *гибернации* компьютера (режим S4), только в более мелком масштабе. Текущее состояние метода сохраняется, и метод возвращает управление. При переходе в режим гибернации динамическое состояние всех программ записывается на диск и компьютер выключается. Отключение питания не нанесет компьютеру, находящемуся в режиме гибернации, никакого вреда. Точно так же, ожидающий метод не потребляет никаких ресурсов, кроме разве что крохотного объема памяти, поскольку запустивший его поток освобожден.

Если продолжить аналогию, то блокирующий метод больше всего напоминает *спящий* режим компьютера (режим S3). В нем компьютер потребляет меньше ресурсов, но по существу продолжает работать.

В идеале программист не должен замечать, что имела место гибернация. Несмотря на то, что приостановка метода в середине и последующее возобновление – довольно сложная операция, C# гарантирует, что ваша программа продолжит выполнение, как будто ничего не случилось.

Состояние метода

Чтобы стало яснее, сколько работы должен выполнить компилятор C#, встретив в программе оператор `await`, я перечислю, какие именно аспекты состояния метода необходимо сохранить.

Во-первых, запоминаются все локальные переменные метода, в том числе:

- параметры метода;
- все переменные, определенные в текущей области видимости;
- все прочие переменные, например счетчики циклов;
- переменную `this`, если метод не статический. В результате после возобновления метода окажутся доступны все переменные-члены класса.

Всё это сохраняется в виде объекта в куче .NET, обслуживаемой сборщиком мусора. Таким образом, встретив `await`, компилятор выделяет память для объекта, то есть расходует ресурсы, но в большинстве случае это не приводит к потере производительности.

C# также запоминает место, где встретился оператор `await`. Эту информацию можно представить в виде числа, равного порядковому

номеру достигнутого оператора `await` среди всех прочих встречающихся в методе.

На способ использования выражений `await` не накладывается никаких ограничений. Например, это может быть часть объемлющего выражения, возможно, содержащего несколько `await`:

```
int myNum = await AlexsMethodAsync(await myTask, await StuffAsync());
```

Поэтому на время ожидания завершения операции требуется также запомнить состояние остальной части выражения. В примере выше результат выражения `await myTask` необходимо сохранить на время исполнения `await StuffAsync()`. В промежуточном языке .NET (IL) такие подвыражения сохраняются в стеке, поэтому, встретив оператор `await`, компилятор должен запомнить еще и этот стек.

Далее, когда программа доходит до первого `await` в методе, этот метод возвращает управление. Если метод не объявлен как `async void`, то в этот момент возвращается объект `Task`, чтобы вызывающая программа могла дождаться завершения. C# должен также запомнить этот объект, чтобы по завершении метода его можно было заполнить и продолжить выполнение цепочки асинхронных методов. Как именно это делается, мы рассмотрим в главе 14.

Контекст

Пытаясь сделать процесс ожидания максимально прозрачным, C# запоминает различные аспекты контекста в точке, где встретился `await`, а затем, при возобновлении метода, восстанавливает их.

Наиболее важным из всех является *контекст синхронизации*, который среди прочего позволяет возобновить выполнение метода в конкретном потоке. Это особенно важно для приложений с графическим интерфейсом, которым можно манипулировать только из одного, вполне определенного потока. Контекст синхронизации – это сложная тема, к которой мы еще вернемся в главе 8.

Есть и другие виды контекста вызывающего потока, которые также необходимо запомнить. Все они собираются в соответствующих классах, наиболее важные из которых перечислены ниже.

`ExecutionContext`

Это родительский контекст, включающий все остальные контексты. Он не имеет собственного поведения, а служит только для запоминания и передачи контекста и используется такими компонентами .NET, как класс `Task`.

`SecurityContext`

Здесь хранится информация о безопасности, обычно действующая только в пределах текущего потока. Если код должен выполняться от имени конкретного пользователя, то, вероятно, ваша программа *олицетворяет* этого пользователя, или за вас это делает ASP.NET. В таком случае сведения об олицетворении хранятся в объекте `SecurityContext`.

`CallContext`

Позволяет программисту сохранить определенные им самим данные, которые должны быть доступны на протяжении всего времени жизни логического потока. В большинстве случаев использование этой возможности не поощряется, но таким образом можно избежать передачи большого числа параметров методам в пределах одной программы, поместив их вместо этого в контекст вызова. Класс `LogicalCallContext` предназначен для той же цели, но работает через границы доменов приложений.

Следует отметить, что *поточно-локальная память*, предназначенная для той же цели, что `CallContext`, для асинхронных программ не годится, потому что на время выполнения длительной операции поток освобождается и может быть использован для других целей. Ваш метод может быть возобновлен совершенно в другом потоке.

Перед возобновлением метода C# восстанавливает все эти контексты. С восстановлением сопряжены определенные накладные расходы. Например, асинхронная программа, в которой используется олицетворение, может работать существенно медленнее. Я рекомендую не пользоваться средствами .NET, создающими контексты, если это не является насущной необходимостью.

Когда нельзя использовать await

Оператор `await` можно использовать почти в любом месте метода, помеченного ключевым словом `async`. Однако есть несколько случаев, когда использование `await` запрещено. Ниже я объясню, почему это так.

Блоки catch и finally

Оператор `await` может встречаться внутри блока `try`, но не внутри блоков `catch` или `finally`. В блоке `catch` часто, а в блоке `finally` всег-

да, исключение еще находится в фазе раскрутки стека и позже может быть возбуждено в блоке повторно. Если использовать в этой точке `await`, то стек окажется другим, и определить в этой ситуации поведение повторного возбуждения исключения было бы очень сложно.

Напомню, что `await` всегда можно поставить не внутри блока `catch`, а после него, для чего следует либо воспользоваться предложением `return`, либо завести булевскую переменную, в которой запомнить, возбуждала ли исходная операция исключение. Например, вместо такого некорректного в C# кода:

```
try
{
 page = await webClient.DownloadStringTaskAsync("http://oreilly.com");
}
 catch (WebException)
{
 page =
   await webClient.DownloadStringTaskAsync("http://oreillymirror.com");
}
```

можно было бы написать:

```
bool failed = false;
try
{
 page = await webClient.DownloadStringTaskAsync("http://oreilly.com");
}
catch (WebException)
{
 failed = true;
}

if (failed)
{
 page =
   await webClient.DownloadStringTaskAsync("http://oreillymirror.com");
}
```

Блоки lock

Ключевое слово `lock` позволяет запретить другим потокам доступ к объектам, с которыми в данный момент работает текущий поток. Поскольку асинхронный метод обычно освобождает поток, в котором начал асинхронную операцию, и через неопределенно долгое время может быть возобновлен в другом потоке, то удерживать блокировку во время выполнения `await` не имеет смысла.

В некоторых случаях важно защитить объект от одновременного доступа, но разрешить другим потокам обращаться к нему во время `await`. Тогда можно написать чуть более длинный код, в котором синхронизация явно производится дважды:

```
lock (sync)
{
 // Подготовиться к асинхронной операции
}

int myNum = await AlexsMethodAsync();

lock (sync)
{
 // Использовать результат асинхронной операции
}
```

Можно вместо этого воспользоваться библиотекой, реализующей управление одновременным доступом, например NAct (см. главу 10).

Если вы считаете, что программе все же необходимо удерживать ту или иную блокировку на протяжении асинхронной операции, попытайтесь переосмыслить ее дизайн, потому что в общем случае чрезвычайно трудно реализовать блокировку ресурсов на время асинхронного вызова, избежав при этом состояний гонки и взаимоблокировок.

Выражения LINQ-запросов

В C# имеется синтаксис для упрощения записи декларативных запросов на фильтрацию, трансформацию, упорядочение и группировку данных. Впоследствии запрос может быть применен к коллекции .NET или преобразован в форму, пригодную для применения к базе данных или другому источнику данных.

```
IEnumerable<int> transformed = from x in alexsInts
                               where x != 9
                               select x + 2;
```

В большинстве мест внутри выражения запроса употребление `await` недопустимо. Объясняется это тем, что эти места компилятор преобразует в лямбда-выражения, и, значит, такое лямбда-выражение следовало бы пометить ключевым словом `async`. Однако синтаксиса, позволяющего пометить эти неявные лямбда-выражения как `async`, просто не существует, и попытка ввести его только привела бы к недоразумениям.

При необходимости всегда можно написать эквивалентное выражение с помощью методов расширения, как это делает сам LINQ. Тогда лямбда-выражения станут явными и их можно будет пометить как `async`, чтобы использовать внутри `await`.

```
IEnumerable<Task<int>> tasks = alexsInts
  .Where(x => x != 9)
  .Select(async x => await DoSomthingAsync(x) +
                      await DoSomthingElseAsync(x));

IEnumerable<int> transformed = await Task.WhenAll(tasks);
```

Для сбора результатов я воспользовался методом `Task.WhenAll`, предназначенным для работы с коллекциями объектов `Task`. Подробно мы рассмотрим его в главе 7.

Небезопасный код

Код, помеченный ключевым словом `unsafe`, не может содержать `await`. Необходимость в небезопасном коде возникает очень редко, и обычно он содержит автономные методы, не требующие асинхронности. Да и в всё равно преобразования, выполняемые компилятором при обработке `await`, в большинстве случаев сделали бы небезопасный код неработоспособным.

Запоминание исключений

По идее, исключения в асинхронных методах должны работать практически так же, как в синхронных. Но из-за дополнительных сложностей механизма async существуют некоторые тонкие различия. В этом разделе я расскажу о том, как async упрощает обработку исключений, а некоторые подводные камни опишу более подробно в главе 9.

По завершении операции в объекте `Task` сохраняется информация о том, завершилась ли она успешно или с ошибкой. Получить к ней доступ проще всего с помощью свойства `IsFaulted`, которое равно `true`, если во время выполнения операции произошло исключение. Оператор `await` знает об этом и повторно возбуждает исключение, хранящееся в `Task`.

У читателя, знакомого с системой исключений в .NET, может возникнуть вопрос, корректно ли сохраняется первоначальная трассировка стека исключения при его повторном возбуждении. Раньше это было невозможно; каждое исключение могло быть

возбуждено только один раз. Однако в .NET 4.5 это ограничение снято благодаря новому классу `ExceptionDispatchInfo`, который взаимодействует с классом `Exception` с целью запоминания трассировки стека и воспроизведения ее при повторном возбуждении.

Async-методы также знают об исключениях. Любое исключение, возбужденное, но не перехваченное в async-методе, помещается в объект `Task`, возвращаемый вызывающей программе. Если в этот момент вызывающая программа уже ждет объекта `Task`, то исключение будет возбуждено в точке ожидания. Таким образом, исключение передается вызывающей программе вместе со сформированной виртуальной трассировкой стека – точно так же, как в синхронном коде.

Я называю это *виртуальной* трассировкой стека, потому что стек – вообще-то принадлежность потока, а в асинхронной программе реальный стек текущего потока может не иметь ничего общего с трассировкой стека в момент исключения. В исключении запоминается трассировка стека, отражающая *намерение* программиста, в ней представлены те методы, который программист вызывал сам, а не детали того, как C# исполнял части этих методов в действительности.

Асинхронные методы до поры исполняются синхронно

Выше я уже отмечал, что async-метод становится асинхронным, только встретив вызов асинхронного метода внутри оператора `await`. До этого момента он работает в том потоке, в котором вызван, как обычный синхронный метод. Иногда это приводит к вполне ощутимым последствиям, особенно если есть шанс, что вся цепочка async-методов исполняется в синхронном режиме.

Напомню, что async-метод приостанавливается, лишь дойдя до первого `await`. Но даже в этом случае бывает, что приостановка не нужна, потому что иногда задача `Task`, переданная оператору `await`, уже завершена. Так может случиться в следующих ситуациях.

- Она была завершена уже в момент создания методом `Task.FromResult`, который мы рассмотрим подробнее в главе 7.
- Она была возвращена async-методом, который так ни разу и не дошел до `await`.
- Действительно была выполнена асинхронная операция, но она уже завершилась (быть может потому, что до момента вызова `await` текущий поток делал что-то еще).

- Она была возвращена async-методом, который дошел до `await`, но задача `Task`, завершения которой тот ждал, уже завершилась.

Именно последняя возможность – когда глубоко внутри цепочки async-методов произошло ожидание уже завершившейся задачи – дает интересный эффект. Выполнение всей цепочки вполне может оказаться синхронным. Объясняется это тем, что в цепочке async-методов первым всегда вызывается `await` с самым глубоким уровнем вложенности. До остальных дело доходит только после того, как у самого глубоко вложенного метода был шанс завершиться синхронно.

Возможно, вам непонятно, зачем вообще использовать async в первом или втором из вышеупомянутых случаев. Если бы можно было гарантировать, что метод всегда исполняется синхронно, то да, было бы эффективнее сразу написать синхронный код, чем async-метод без `await`. Но дело в том, что бывают ситуации, когда метод возвращает управление синхронно лишь в *некоторых* случаях. Например, если метод кэширует результаты в памяти, то он может завершиться синхронно, когда результат уже находится в кэше, а в противном случае будет асинхронно обратиться к сети. Кроме того, иногда имеет смысл возвращать из метода `Task` или `Task<T>` с прицелом на будущее, если известно, что впоследствии он будет переписан и сделан асинхронным.

ГЛАВА 6. Паттерн TAP

Паттерн Task-based Asynchronous Pattern (TAP) – это предлагаемый Microsoft набор рекомендаций по написанию асинхронных API в .NET с помощью класса `Task`. В документе[1], написанном Стивеном Тоубом (Stephen Toub) из группы параллельного программирования в Microsoft приводятся содержательные примеры; с ним, безусловно, стоит ознакомиться.

Следование этому паттерну позволяет строить API, допускающие использование внутри `await`, и, хотя добавление ключевого слова `async` порождает методы, согласованные с TAP, иногда полезно работать непосредственно с классом `Task`. В этой главе я объясню, в чем смысл этого паттерна, и продемонстрирую технику работы с ним.

Что специфицировано в TAP?

Я буду предполагать, что вам уже известно, как проектировать хорошую сигнатуру синхронного метода в C#:

- метод должен иметь нуль или более параметров, причем параметров со спецификаторами `ref` и `out` по возможности следует избегать;
- в тех случаях, когда это необходимо, для метода должен быть задан тип возвращаемого значения, которое действительно содержит результат работы метода, а не просто является индикатором успеха, как иногда бывает в коде на C++;
- у метода должно быть имя, объясняющее его назначение и не содержащее какой-либо дополнительной нотации;
- типичные или ожидаемые ошибки должны отражаться в типе возвращаемого значения, тогда как неожиданные ошибки должны приводить к возбуждению исключения.

1 http://www.microsoft.com/en-gb/download/details.aspx?id=19957

Вот пример хорошо спроектированного синхронного метода из класса `Dns`:

```
public static IPHostEntry GetHostEntry(string hostNameOrAddress)
```

TAP содержит аналогичные рекомендации по проектирования асинхронных методов, основанные на уже имеющихся у вас знаниях о синхронных методах. Перечислим их.

- Параметры должны быть такими же, как у эквивалентного синхронного метода. Параметры со спецификаторами `ref` и `out` не допускаются.
- Метод должен возвращать значение типа `Task` или `Task<T>` в зависимости от того, возвращает что-то синхронный метод или нет. Задача должна завершаться в какой-то момент в будущем и предоставлять методу значение результата.
- Метод следует называть по образцу *`Name`*`Async`, где *`Name`* – имя эквивалентного синхронного метода.
- Исключение из-за ошибки в параметрах вызова метода можно возбуждать непосредственно. Все остальные исключения следует помещать в объект `Task`.

Ниже приведена сигнатура метода, следующая рекомендациям TAP:

```
public static Task<IPHostEntry> GetHostEntryAsync(string hostNameOrAddress)
```

Все это может показаться очевидным, но, как мы видели в разделе «О некоторых асинхронных паттернах в .NET», TAP – уже третий формальный асинхронный паттерн из имеющихся в каркасе .NET, и я уверен, что есть еще бесчисленное множество неформализованных способов написания асинхронного кода.

Основная идея TAP заключается в том, что асинхронный метод должен возвращать объект типа `Task`, инкапсулирующий обещание длительной операции завершиться в будущем. Без этого более ранним асинхронным паттернам приходилось вводить дополнительные параметры метода либо включать в интерфейс дополнительные методы или события, чтобы поддержать механизм обратных вызовов. Объект `Task` может содержать всю инфраструктуру, необходимую для обратных вызовов, не засоряя код техническими деталями.

У такого подхода есть и еще одно достоинство: поскольку механизм асинхронных вызовов теперь находится в `Task`, нет нужды дублировать его при каждом асинхронном вызове. Это в свою очередь

означает, что механизм можно сделать более сложным и мощным, в частности восстанавливать различные контексты, например контекст синхронизации, перед обратным вызовом. Наконец, TAP предлагает единый API для работы с асинхронными операциями, что позволяет реализовать такие средства, как async, на уровне компилятора; при использовании прежних паттернов это было невозможно.

Использование Task для операций, требующих большого объема вычислений

Иногда длительная операция не отправляет запросов в сеть и не обращается к диску, а просто выполняет продолжительное вычисление. Разумеется, нельзя рассчитывать на то, что при этом удастся обойтись без занятия потока, как в случае сетевого доступа, но желательно все же избежать зависания пользовательского интерфейса. Для этого мы должны вернуть управление потоку пользовательского интерфейса, чтобы он мог обрабатывать другие события, а длительное вычисление производить в отдельном потоке.

Класс `Task` позволяет это сделать, а для обновления пользовательского интерфейса по завершении вычисления мы можем, как обычно, использовать `await`:

```
Task t = Task.Run(() => MyLongComputation(a, b));
```

Метод `Task.Run` исполняет переданный ему делегат в потоке, взятом из пула `ThreadPool`. В данном случае я воспользовался лямбда-выражением, чтобы упростить передачу счетной задаче локальных переменных. Возвращенная задача `Task` запускается немедленно, и мы можем дождаться ее завершения, как любой другой задачи:

```
await Task.Run(() => MyLongComputation(a, b));
```

Это очень простой способ выполнить некоторую работу в фоновом потоке.

Если необходим более точный контроль над тем, какой поток производит вычисления или как он планируется, в классе `Task` имеется статическое свойство `Factory` типа `TaskFactory`. У него есть метод `StartNew` с различными перегруженными вариантами для управления вычислением:

```
Task t = Task.Factory.StartNew(() => MyLongComputation(a, b),
                               cancellationToken,
                               TaskCreationOptions.LongRunning,
                               taskScheduler);
```

Если вы занимаетесь разработкой библиотеки, в которой много счетных методов, то может возникнуть соблазн предоставить их асинхронные версии, которые вызывают `Task.Run` для выполнения работы в фоновом потоке. Это неудачная идея, потому что пользователь вашего API лучше вас знает о требованиях приложения к организации потоков. Например, в веб-приложениях использование пула потоков не дает никакого выигрыша; единственное, что следует оптимизировать, – это общее число потоков. Вызвать метод `Task.Run` очень просто, поэтому оставьте это решение на усмотрение пользователя.

Создание задачи-марионетки

Методы, написанные в соответствии с TAP, настолько просто использовать, что вы естественно захотите оформлять свои API именно так. Мы уже знаем, как потреблять другие TAP API с помощью async-методов. Но что, если для какой-то длительной операции еще не написан TAP API? Быть может, в реализации ее API использован какой-то другой асинхронный паттерн. А быть может, нет вообще никакого API, и вы делаете нечто асинхронное вручную.

На этот случай предусмотрен класс `TaskCompletionSource<T>`, позволяющий создать задачу `Task`, которой вы управляете как марионеткой. Вы можете в любой момент сделать эту задачу успешно завершившейся. Или записать в нее исключение и тем самым сказать, что она завершилась с ошибкой.

Рассмотрим пример. Предположим, требуется инкапсулировать показываемый пользователю вопрос в следующем методе:

```
Task<bool> GetUserPermission()
```

Вопрос представляет собой написанный вами диалог, в котором у пользователя запрашивается какое-то разрешение. Поскольку запрашивать разрешение нужно в разных местах приложения, важно, чтобы метод было просто вызывать. Идеальная ситуация для использования асинхронного метода, так как мы не хотим, чтобы этот диалог отображался в потоке пользовательского интерфейса. Однако этот метод очень далек от традиционных асинхронных методов, в которых

производится обращение к сети или еще какая-то длительная операция. В данном случае мы ждем ответа от пользователя. Рассмотрим тело метода.

```
private Task<bool> GetUserPermission()
{
 // Создать объект TaskCompletionSource, чтобы можно было вернуть
 // задачу-марионетку
 TaskCompletionSource<bool> tcs = new TaskCompletionSource<bool>();

 // Создать диалог
 PermissionDialog dialog = new PermissionDialog();

 // Когда пользователь закроет диалог, сделать задачу завершившейся
 // с помощью метода SetResult
 dialog.Closed += delegate { tcs.SetResult(dialog.PermissionGranted); };

 // Показать диалог на экране
 dialog.Show();

 // Вернуть еще не завершившуюся задачу-марионетку
 return tcs.Task;
}
```

Обратите внимание, что метод не помечен ключевым словом `async`; мы создаем объект `Task` вручную и не нуждаемся в помощи компилятора. `TaskCompletionSource<bool>` создает объект `Task` и предоставляет к нему доступ через свойство `Task`. Мы возвращаем этот объект, а позже делаем его завершившимся, вызывая метод `SetResult` объекта `TaskCompletionSource`.

Поскольку мы следовали паттерну TAP, вызывающая программа может просто ждать разрешения пользователя с помощью `await`. Код получается весьма элегантным:

```
if (await GetUserPermission())
{ ....
```

Вызывает раздражение отсутствие неуниверсальной версии класса `TaskCompletionSource<T>`. Однако `Task<T>` — подкласс `Task`, поэтому его можно использовать всюду, где требуется объект `Task`. Это в свою очередь означает, что можно воспользоваться классом `TaskCompletionSource<T>`, и рассматривать объект типа `Task<T>`, являющийся значением свойства `Task`, как объект типа `Task`. Я обычно работаю с конкретизацией `TaskCompletionSource<object>` и для ее заполнения вызываю `SetResult(null)`. При желании нетрудно создать неуниверсальную версию `TaskCompletionSource`, основываясь на универсальной.

Взаимодействие с прежними асинхронными паттернами

Разработчики .NET создали TAP-версии для всех важных асинхронных API, встречающихся в каркасе. Но на случай, если вам придется взаимодействовать с каким-нибудь написанным ранее асинхронным кодом, полезно знать, как реализовать согласованный с TAP метод на основе варианта, не согласованного с TAP. Кстати, это еще и интересный пример использования класса `TaskCompletionSource<T>`.

Обратимся к примеру поиска в DNS, который я приводил выше. В .NET 4.0 API асинхронного поиска в DNS был основан на применении паттерна `IAsyncResult`, то есть фактически состоит из пары методов – `Begin` и `End`:

```
IAsyncResult BeginGetHostEntry(string hostNameOrAddress,
                               AsyncCallback requestCallback,
                               object stateObject)

IPHostEntry EndGetHostEntry(IAsyncResult asyncResult)
```

Обычно для использования такого API применяется лямбда-выражение, внутри которого вызывается метод `End`. Так мы и поступим, только в обратном вызове не будем делать ничего содержательного, а просто завершим задачу `Task` с помощью `TaskCompletionSource<T>`.

```
public static Task<IPHostEntry> GetHostEntryAsync(string hostNameOrAddress)
{
 TaskCompletionSource<IPHostEntry> tcs =
    new TaskCompletionSource<IPHostEntry>();
 Dns.BeginGetHostEntry(hostNameOrAddress, asyncResult =>
   {
    try
    {
     IPHostEntry result = Dns.EndGetHostEntry(asyncResult);
     tcs.SetResult(result);
    }
    catch (Exception e)
    {
     tcs.SetException(e);
    }
   }, null);

 return tcs.Task;
}
```

Из-за возможности исключения этот код пришлось несколько усложнить. Если поиск в DNS завершается неудачно, то обращение к `EndGetHostEntry` возбуждает исключение. Именно по этой причине в паттерне `IAsyncResult` нельзя просто передать результат методу обратного вызова, а необходим метод `End`. Обнаружив исключение, мы должны поместить его в объект `TaskCompletionSource<T>`, чтобы вызывающая программа могла получить его, – в полном соответствии с TAP.

На самом деле, количество асинхронных API, построенных по этому образцу, настолько велико, что разработчики .NET включили вспомогательный метод для преобразования их к виду, совместимому с TAP. И мы тоже могли бы воспользоваться этим методом:

```
Task t = Task<IPHostEntry>.Factory.FromAsync<string>(Dns.BeginGetHostEntry,
                                                     Dns.EndGetHostEntry,
                                                     hostNameOrAddress,
                                                     null);
```

Этот метод принимает методы `Begin` и `End` в виде делегатов и внутри устроен очень похоже на наш вариант. Разве что чуть более эффективен.

Холодные и горячие задачи

В библиотеке Task Parallel Library, впервые появившейся в версии .NET 4.0, было понятие *холодной* задачи `Task`, которую еще только предстоит запустить, и *горячей* задачи, которая уже запущена. До сих пор мы имели дело только с горячими задачами.

В спецификации TAP оговорено, что любая задача, возвращенная из метода, должна быть горячей. К счастью, все рассмотренные выше приемы и так создают горячие объекты `Task`. Исключение составляет техника на основе класса `TaskCompletionSource<T>`, к которой понятие о горячей или холодной задаче неприменимо. Вы просто должны позаботиться о том, чтобы в какой-то момент сделать объект `Task` завершенным самостоятельно.

Предварительная работа

Мы уже знаем, что асинхронный TAP-метод, как и всякий другой, сразу после вызова начинает исполняться в текущем потоке. Разница же в том, что TAP-метод, скорее всего, не завершается в момент

возврата. Он быстро возвращает объект-задачу `Task`, которая станет завершенной, когда закончит выполнение.

Тем не менее, какая-то часть метода работает синхронно в текущем потоке. Как мы видели в разделе «Асинхронные методы до поры исполняются синхронно», в случае async-метода эта часть содержит по меньшей мере код до первого оператора `await`, включая и его операнд.

TAP рекомендует, чтобы синхронно выполняемая часть TAP-метода была сведена к минимуму. В ней можно проверить корректность аргументов и посмотреть, нет ли нужных данных в кэше (что позволит избежать длительной операции), но никаких медленных вычислений производить не следует. Гибридные методы, в которых сначала выполняется какое-то вычисление, а затем доступ к сети или нечто подобное, вполне допустимы, но рекомендуется перенести вычисление в фоновый поток, вызвав метод `Task.Run`. Вот, например, как может выглядеть функция, которая закачивает изображение на сайт, но для экономии пропускной способности сети сначала уменьшает его размер:

```
Image resized = await Task.Run(() => ResizeImage(originalImage));
await UploadImage(resized);
```

Это существенно в приложении с графическим интерфейсом, но в веб-приложениях не дает практического выигрыша. Тем не менее, мы ожидаем, что метод, следующий рекомендациям TAP, будет возвращать управление быстро. Всякий, кто захочет перенести ваш код в приложение с графическим интерфейсом, будет весьма удивлен, увидев, что медленная операция масштабирования изображения выполняется синхронно.

ГЛАВА 7. Вспомогательные средства для асинхронного кода

В дизайн паттерна TAP заложен механизм, позволяющий упростить создание вспомогательных средств для работы с задачами – объектами `Task`. Поскольку любой метод, следующий TAP, возвращает `Task`, любое специализированное поведение, реализованное для одного такого метода, можно повторно использовать и в других. В этой главе мы рассмотрим некоторые средства для работы с объектами `Task`, в том числе:

- методы, которые выглядят как TAP-методы, но сами по себе не являются асинхронными, а обладают каким-то полезным специальным поведением;
- комбинаторы, то есть методы, которые порождают из одного объекта `Task` другой, в чем-то более полезный;
- средства для отмены асинхронных операций и информирования о ходе их выполнения.

Хотя имеется немало готовых средств такого рода, полезно знать, как реализовать их самостоятельно, – на случай, если понадобится что-то такое, чего в каркасе .NET Framework нет.

Задержка на указанное время

Простейшая длительная операция – это «ничегонеделание» в течение некоторого времени, то есть аналог синхронного метода `Thread.Sleep`. Вообще-то, и реализовать ее можно, воспользовавшись `Thread.Sleep` в сочетании с `Task.Run`:

```
await Task.Run(() => Thread.Sleep(100));
```

Но такой простой подход расточителен. Мы захватываем поток только для того, чтобы заблокировать его, а это прямое расточитель-

ство ресурсов. Ведь существует способ заставить .NET вызвать ваш код по истечении заданного времени без использования дополнительных потоков – класс `System.Threading.Timer`. Поэтому гораздо эффективнее взвести таймер, а затем с помощью класса `TaskCompletionSource` создать объект `Task` и сделать его завершенным в момент срабатывания таймера:

```
private static Task Delay(int millis)
{
 TaskCompletionSource<object> tcs = new TaskCompletionSource<object>();
 Timer timer = new Timer(_ => tcs.SetResult(null), null, millis,
                                                  Timeout.Infinite);
 tcs.Task.ContinueWith(delegate { timer.Dispose(); });
 return tcs.Task;
}
```

Разумеется, это полезная утилита уже предоставляется каркасом. Она называется `Task.Delay` и без сомнения является более мощной, надежной и, вероятно, более эффективной, чем моя версия.

Ожидание завершения нескольких задач

В разделе «Task и await» выше мы видели, как просто организовать выполнение несколько параллельных асинхронных задач, – нужно запустить их по очереди, а затем ждать завершения каждой. В главе 9 мы узнаем, что необходимо дождаться завершения каждой задачи, иначе можно пропустить исключения.

Для решения этой задачи можно воспользоваться методом `Task.WhenAll`, который принимает несколько объектов `Task` и порождает агрегированную задачу, которая завершается, когда завершены все исходные задачи. Вот простейший вариант метода `WhenAll` (имеется также перегруженный вариант для коллекции универсальных объектов `Task<T>`):

```
Task WhenAll(IEnumerable<Task> tasks)
```

Основное различие между использованием `WhenAll` и самостоятельным ожиданием нескольких задач, заключается в том, что `WhenAll` корректно работает даже в случае исключений. Поэтому старайтесь всегда пользоваться методом `WhenAll`.

Универсальный вариант `WhenAll` возвращает массив, содержащий результаты отдельных поданных на вход задач `Task`. Это сделано ско-

рее для удобства, чем по необходимости, потому что доступ к исходным объектам `Task` сохраняется, и ничто не мешает опросить их свойство `Result`, так как точно известно, что все задачи уже завершены.

Обратимся снова к обозревателю значков сайтов. Напомним, что у нас уже есть версия, которая вызывает метод типа `async void`, начинающий скачивание одного значка. По завершении скачивания значок добавляется в окно. Это решение очень эффективно, потому что все операции скачивания производятся параллельно. Однако есть две проблемы:

- значки появляются в порядке, определяемом временем скачивания;
- поскольку каждый значок скачивается в отдельном экземпляре метода типа `async void`, неперехваченные исключения повторно возбуждаются в потоке пользовательского интерфейса, и корректно обработать их нелегко.

Поэтому переработаем программу так, чтобы метод, в котором перебираются значки, сам был помечен ключевым словом async. Тогда мы сможем управлять всеми асинхронными операциями, как одной группой. Вот как выглядит текущий код:

```
private async void GetButton_OnClick(object sender, RoutedEventArgs e)
{
 foreach (string domain in s_Domains)
 {
   Image image = await GetFavicon(domain);
   AddAFavicon(image);
 }
}
```

Изменим его так, чтобы все операции скачивания по-прежнему выполнялись параллельно, но значки отображались в нужном нам порядке. Сначала запустим все операции, вызвав метод `GetFavicon` и сохранив объекты `Task` в списке `List`.

```
List<Task<Image>> tasks = new List<Task<Image>>();
foreach (string domain in s_Domains)
{
 tasks.Add(GetFavicon(domain));
}
```

Или с помощью LINQ, что даже лучше:

```
IEnumerable<Task<Image>> tasks = s_Domains.Select(GetFavicon);

// Вычисление IEnumerable, являющегося результатом Select, отложенное.
```

```
// Инициируем его, чтобы запустить задачи.
tasks = tasks.ToList();
```

Имея группу задач, передадим ее методу `Task.WhenAll`, который вернет объект `Task`; этот объект станет завершенным, когда закончатся все операции скачивания, и в этот момент будет содержать все результаты.

```
Task<Image[]> allTask = Task.WhenAll(tasks);
```

Осталось только дождаться `allTask` и воспользоваться результатами:

```
Image[] images = await allTask;
foreach (Image eachImage in images)
{
 AddAFavicon(eachImage);
}
```

Итак, нам потребовалось всего несколько строчек, чтобы записать довольно сложный параллельный алгоритм. Эта версия программа находится в ветви `whenAll`.

Ожидание завершения любой задачи из нескольких

Часто возникает также ситуация, когда требуется дождаться завершения первой задачи из нескольких запущенных. Например, так бывает, когда вы запрашиваете один и тот же ресурс из нескольких источников и готовы удовлетвориться первым полученным ответом.

Для этой цели предназначен метод `Task.WhenAny`. Ниже приведен универсальный вариант, он наиболее интересен, хотя есть еще ряд перегруженных.

```
Task<Task<T>> WhenAny(IEnumerable<Task<T>> tasks)
```

Сигнатура метода `WhenAny` несколько сложнее, чем метода `WhenAll`, но тому есть причины. В ситуации, когда возможны исключения, пользоваться методом `WhenAny` следует с осторожностью. Если вы хотите знать обо всех исключениях, произошедших в программе, то необходимо ждать каждую задачу, иначе некоторые исключения могут быть потеряны. Воспользоваться методом `WhenAny` и просто забыть об остальных задачах – всё равно, что перехватить все исключения и игнорировать их. Это достойное порицания решение, которое

может впоследствии привести к тонким ошибкам и недопустимым состояниям программы.

Метод WhenAny возвращает значение типа Task<Task<T>>. Это означает, что по завершении задачи вы получаете объект типа Task<T>. Он представляет первую из завершившихся задач и поэтому гарантированно находится в состоянии «завершен». Но почему нам возвращают объект Task, а не просто значение типа T? Чтобы мы знали, какая задача завершилась первой, и могли отменить все остальные и дождаться их завершения.

```
Task<Task<Image>> anyTask = Task.WhenAny(tasks);
Task<Image> winner = await anyTask;
Image image = await winner; // Этот оператор всегда завершается синхронно

AddAFavicon(image);

foreach (Task<Image> eachTask in tasks)
{
 if (eachTask != winner)
 {
   await eachTask;
 }
}
```

Нет ничего предосудительного в том, чтобы обновить пользовательский интерфейс, как только будет получен результат первой завершившейся задачи (winner), но после этого необходимо дождаться остальных задач, как сделано в коде выше. При удачном стечении обстоятельств все они завершатся успешно, и на выполнении программы это никак не отразится. Если же какая-то задача завершится с ошибкой, то вы сможете узнать причину и исправить ошибку.

Создание собственных комбинаторов

Методы WhenAll и WhenAny называются асинхронными комбинаторами. Они возвращают объект Task, но сами по себе не являются асинхронными методами, а лишь комбинируют другие объекты Task тем или иным полезным способом. При необходимости вы можете и сами написать комбинаторы и тем самым запастись набором повторно используемых поведений.

Покажем на примере, как пишется комбинатор. Допустим, нам нужно добавить к произвольной задаче таймаут. Было бы несложно

написать такой код с нуля, но мы продемонстрируем, как удачно здесь можно применить методы `Delay` и `WhenAny`. Вообще говоря, комбинаторы проще всего реализовывать с помощью механизма async, как в данном случае, но так бывает не всегда.

```
private static async Task<T> WithTimeout<T>(Task<T> task, int time)
{
 Task delayTask = Task.Delay(time);
 Task firstToFinish = await Task.WhenAny(task, delayTask);

 if (firstToFinish == delayTask)
 {
   // Первой закончилась задача задержки - разобраться с исключениями
   task.ContinueWith(HandleException);
   throw new TimeoutException();
 }

 // Если мы дошли до этого места, исходная задача уже завершилась
 return await task;
}
```

Мой подход состоит в том, чтобы создать методом `Delay` задачу `Task`, которая завершится по истечении таймаута. Затем с помощью метода `WhenAny` я жду эту и исходную задачу, так что исполнение возобновляется в любом из двух случаев: когда завершилась исходная задача или истек таймаут. Далее я смотрю, что именно произошло и либо возбуждаю исключение `TimeoutException`, либо возвращаю результат.

Обратите внимание, что я аккуратно обрабатываю исключения в случае таймаута. С помощью метода `ContinueWith` я присоединил к исходной задаче продолжение, в котором будет обработано исключение, если таковое имело место. Я точно знаю, что задача задержки не может возбудить исключение, поэтому и пытаться перехватить его не нужно. Метод `HandleException` можно было бы реализовать следующим образом:

```
private static void HandleException<T>(Task<T> task)
{
 if (task.Exception != null)
 {
   logging.LogException(task.Exception);
 }
}
```

Что именно в нем делается, конечно, зависит от вашей стратегии обработки исключений. Присоединив этот метод с помощью

ContinueWith, я гарантирую, что в момент завершения исходной задачи, когда бы это ни случилось, будет выполнен код проверки исключения. Важно, что это никак не тормозит исполнение основной программы, которая уже сделала всё необходимое в момент, когда истек таймаут.

Отмена асинхронных операций

Согласно TAP, отмена связывается не с типом Task, а с типом CancellationToken. По соглашению, всякий TAP-метод, поддерживающий отмену, должен иметь перегруженный вариант, в котором за обычными параметрами следует параметр типа CancellationToken. В самом каркасе примером может служить тип DbCommand и его асинхронные методы для опроса базы данных. Простейший перегруженный вариант метода ExecuteNonQueryAsync вообще не имеет параметров, и ему соответствует такой вариант с параметром CancellationToken:

```
Task<int> ExecuteNonQueryAsync(CancellationToken cancellationToken)
```

Посмотрим, как отменить вызванный асинхронный метод. Для этого нам понадобится класс CancellationTokenSource, который умеет создавать объекты CancellationToken и управлять ими – примерно так же, как TaskCompletionSource создает и управляет объектами Task. Приведенный ниже код неполон, но дает представление о применяемой технике:

```
CancellationTokenSource cts = new CancellationTokenSource();
cancelButton.Click += delegate { cts.Cancel(); };
int result = await dbCommand.ExecuteNonQueryAsync(cts.Token);
```

При вызове метода Cancel объекта CancellationTokenSource тот переходит в состояние «отменен». Мы можем зарегистрировать делегат, который будет вызван в этот момент, но на практике эффективнее гораздо более простой подход, заключающийся в проверке того, что вызывающая программа хочет отменить начатую операцию. Если в асинхронном методе есть цикл и объект CancellationToken доступен, то достаточно просто вызывать на каждой итерации метод ThrowIfCancellationRequested.

```
foreach (var x in thingsToProcess)
{
 cancellationToken.ThrowIfCancellationRequested();
```

```
    // Обработать x ...
  }
```

При вызове метода ThrowIfCancellationRequested отмененного объекта CancellationToken возбуждается исключение типа OperationCanceledException. Библиотека Task Parallel Library знает, что такое исключение представляет отмену, а не ошибку, и обрабатывает его соответственно. Например, в классе Task имеется свойство IsCanceled, которое автоматически принимает значение true, если при выполнении async-метода произошло исключение OperationCanceledException.

Удобной особенностью подхода к отмене, основанного на маркерах CancellationToken, является тот факт, что один и тот же маркер можно распространить на столько частей асинхронной операции, сколько необходимо, – достаточно просто передать его всем частям. Неважно, работают они параллельно или последовательно, идет ли речь о медленном вычислении или удаленной операции, – один маркер отменяет всё.

Информирование о ходе выполнения асинхронной операции

Помимо сохранения отзывчивости интерфейса и предоставления пользователю возможности отменить операцию, часто бывает полезно сообщать, сколько времени осталось до конца медленной операции. Для этого в каркас введено еще два типа, рекомендуемых в TAP. Мы должны передать асинхронному методу объект, реализующий интерфейс IProgress<T>, который метод будет вызывать для информирования о ходе работы.

По соглашению, параметр типа IProgress<T> помещается в конец списка параметров, после CancellationToken. Вот как можно было бы добавить средства информирования о ходе выполнения в метод DownloadDataTaskAsync.

```
Task<byte[]> DownloadDataTaskAsync(Uri address,
        CancellationToken cancellationToken,
        IProgress<DownloadProgressChangedEventArgs> progress)
```

Чтобы воспользоваться таким методом, мы должны создать класс, реализующий интерфейс IProgress<T>. По счастью, каркас уже пре-

доставляет такой класс Progress<T>, который в большинстве случаев делает именно то, что нужно. Вы должны либо создать объект, передав его конструктору лямбда-выражение, либо подписаться на событие для получения информации о ходе выполнения, которую можно будет отобразить в пользовательском интерфейсе.

```
new Progress<int>(percentage => progressBar.Value = percentage);
```

Интересно, что при конструировании объекта Progress<T> запоминается контекст SynchronizationContext, который затем используется для вызова кода, обновляющего информацию о ходе выполнения, в нужном потоке. Это по существу то же самое поведение, которое демонстрирует сам объект Task в момент возобновления после await, так что нам не нужно думать о том, что IProgress<T> может вызываться из любого потока.

Если вы хотите сообщать о ходе выполнения из самого TAP-метода, то должны всего лишь вызвать метод Report интерфейса IProgress<T>:

```
progress.Report(percent);
```

Трудная часть заключается в выборе параметра-типа T. Это тип объекта, который передается методу Report, то есть того самого объекта, который передается в лямбда-выражение из вызывающей программы. Если требуется всего лишь показать процент, то можно взять тип int (как в примере выше), но иногда требуется более детальная информация. Однако будьте осторожны, потому что этот объект обычно используется не в том потоке, в котором создан. Во избежание проблем пользуйтесь неизменяемыми типами.

ГЛАВА 8. В каком потоке исполняется мой код?

Я уже говорил, что асинхронное программирование неразрывно связано с потоками. В C# это означает, что необходимо понимать, в каком потоке .NET исполняется наш код и что происходит с потоками во время выполнения длительной операции.

До первого await

В любом async-методе какой-то код предшествует первому вхождению ключевого слова `await`. И, разумеется, само ожидаемое выражение тоже содержит некий код. Весь этот код выполняется в вызывающем потоке. До первого `await` не происходит ничего интересного.

Эту часть механизма async чаще всего понимают неправильно. Async не планирует выполнение метода в фоновом потоке. Единственный способ сделать это – воспользоваться методом `Task.Run`, который специально предназначен для этой цели, или чем-то подобным.

В приложении с пользовательским интерфейсом это означает, что код до первого `await` работает в потоке пользовательского интерфейса. А в веб-приложении на базе ASP.NET – в рабочем потоке ASP.NET.

Часто бывает, что выражение в строке, содержащей первый `await`, содержит еще один async-метод. Поскольку это выражение предшествует первому `await`, оно также выполняется в вызывающем потоке. Таким образом, вызывающий поток продолжает «углубляться» в код приложения, пока не встретит метод, действительно возвращающий объект `Task`. Это может быть метод, являющийся час-

тью каркаса, или метод, создающий задачу-марионетку с помощью `TaskCompletionSource`. Именно этот метод и является источником асинхронности – все прочие async-методы просто распространяют асинхронность вверх по стеку вызовов.

Путь до первой реальной точки асинхронности может оказаться довольно длинным, и весь лежащий на этом пути код исполняется в потоке пользовательского интерфейса, а, стало быть, интерфейс не реагирует на действия пользователя. К счастью, в большинстве случаев он выполняется недолго, но важно помнить, что одно лишь наличие ключевого слова async не гарантирует отзывчивости интерфейса. Если программа отзывается медленно, подключите профилировщик и разберитесь, на что уходит время.

Во время асинхронной операции

Какой поток в действительности исполняет асинхронную операцию?

Вопрос хитрый. Это асинхронный код. Для таких типичных операций, как доступ к сети, вообще не существует потоков, заблокированных в ожидании завершения операции.

Разумеется, если механизм async используется для ожидания завершения вычисления, запущенного, к примеру, с помощью метода `Task.Run`, то занимается поток из пула.

Существует поток, ожидающий завершения сетевых запросов, но он один для всех запросов. В Windows он называется *портом завершения ввода-вывода*. Когда сетевой запрос завершается, обработчик прерывания в операционной системе добавляет задачу в очередь порта завершения. Если запущено 1000 сетевых запросов, то все полученные ответы по очереди обрабатываются единственным портом завершения ввода-вывода.

На самом деле обычно существует несколько потоков, связанных с портом завершения ввода-вывода, – столько, сколько имеется процессорных ядер. Но их число одинаково вне зависимости от того, исполняется в данный момент 10 или 1000 сетевых запросов.

Подробнее о классе SynchronizationContext

Класс `SynchronizationContext` предназначен для исполнения кода в потоке конкретного вида. В .NET имеются разные контексты синхронизации, но наиболее важны контексты потока пользовательского интерфейса, используемые в WinForms и в WPF.

Сам по себе класс `SynchronizationContext` не делает ничего полезного, интерес представляют лишь его подклассы. В классе есть статические члены, позволяющие получить *текущий* `SynchronizationContext` и управлять им. Текущий контекст `SynchronizationContext` – это свойство текущего потока. Идея в том, что всякий раз, как код исполняется в каком-то специальном потоке, мы можем получить текущий контекст синхронизации и сохранить его. Впоследствии этот контекст можно использовать для того, чтобы продолжить исполнение кода в том потоке, в котором оно было начато. Поэтому нам не нужно точно знать, в каком потоке началось исполнение, достаточно иметь соответствующий объект `SynchronizationContext`.

В классе `SynchronizationContext` есть важный метод `Post`, который гарантирует, что переданный делегат будет исполняться в правильном контексте.

Некоторые подклассы `SynchronizationContext` инкапсулируют единственный поток, например поток пользовательского интерфейса. Другие инкапсулируют потоки определенного вида, например взятые из пула потоков, но могут выбирать любой из них. Есть и такие, которые вообще не изменяют поток, в котором выполняется код, а используются только для мониторинга, например контекст синхронизации в ASP.NET.

await и SynchronizationContext

Мы знаем, что код, предшествующий первому `await`, исполняется в вызывающем потоке, но что происходит, когда исполнение вашего метода возобновляется после `await`?

На самом деле, в большинстве случаев он также исполняется в вызывающем потоке, несмотря на то, что в промежутке вызывающий поток мог делать что-то еще. Это существенно упрощает написание кода.

Для достижения такого эффекта используется класс `SynchronizationContext`. Выше в разделе «Контекст» мы видели, что в момент приостановки метода при встрече оператора `await` текущий контекст `SynchronizationContext` сохраняется. Далее, когда метод возобновляется, компилятор вставляет вызов `Post`, чтобы исполнение возобновилось в запомненном контексте.

А теперь о подводных камнях. Метод может возобновиться в потоке, отличном от того, где был начат, при выполнении следующих условий:

- если запомненный контекст `SynchronizationContext` инкапсулирует несколько потоков, например пул потоков;
- если контекст не подразумевает переключения потоков;
- если в точке, где встретился оператор `await`, вообще не было текущего контекста синхронизации, как, например, в консольном приложении;
- если объект `Task` сконфигурирован так, что при возобновлении `SynchronizationContext` не используется.

По счастью, к приложениям с графическим интерфейсом, где возобновление в том же потоке наиболее существенно, ни одно из этих условий не применимо, поэтому после `await` можно без опаски манипулировать пользовательским интерфейсом.

Жизненный цикл асинхронной операции

Попробуем на примере обозревателя значков сайтов понять, в каком потоке какой код исполняется. Я написал два async-метода:

```
async void GetButton_OnClick(...)
async Task<Image> GetFaviconAsync(...)
```

Обработчик события `GetButton_OnClick` вызывает метод `GetFaviconAsync`, который в свою очередь вызывает метод `WebClient.DownloadDataTaskAsync`. На рис. 8.1 приведена диаграмма последовательности событий, возникающих при исполнении этих методов.

1. Пользователь нажимает кнопку, обработчик `GetButton_OnClick` помещается в очередь.
2. Поток пользовательского интерфейса исполняет первую половину метода `GetButton_OnClick`, включая вызов `GetFaviconAsync`.

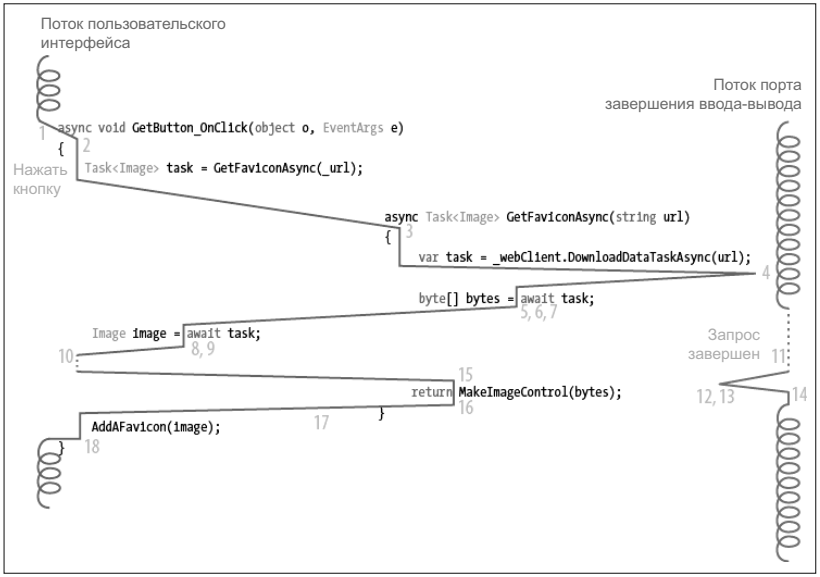

Рис. 8.1. Жизненный цикл асинхронной операции

3. Поток пользовательского интерфейса входит в метод `GetFaviconAsync` и исполняет его первую половину, включая вызов `DownloadDataTaskAsync`.
4. Поток пользовательского интерфейса входит в метод `DownloadDataTaskAsync`, который начинает скачивание и возвращает объект `Task`.
5. Поток пользовательского интерфейса покидает метод `DownloadDataTaskAsync` и доходит до оператора `await` в методе `GetFaviconAsyncAsync`.
6. Запоминается текущий контекст `SynchronizationContext` — поток пользовательского интерфейса.
7. Метод `GetFaviconAsync` приостанавливается оператором `await`, и задача `Task` из `DownloadDataTaskAsync` извещается о том, что она должна возобновиться по завершении скачивания (в запомненном контексте `SynchronizationContext`).
8. Поток пользовательского интерфейса покидает метод `GetFaviconAsync`, который вернул объект `Task`, и доходит до оператора `await` в методе `GetButton_OnClick`.
9. Как и в предыдущем случае, оператор `await` приостанавливает метод `GetButton_OnClick`.

10. Поток пользовательского интерфейса покидает метод `GetButton_OnClick` и освобождается для обработки других действий пользователя.

В этот момент мы ждем, пока скачается значок. На это может уйти несколько секунд. Отметим, что поток пользовательского интерфейса свободен и может обрабатывать другие действия пользователя, а порт завершения ввода-вывода пока не задействован. Во время исполнения операции ни один поток не заблокирован.

11. Скачивание завершается, и порт завершения ввода-вывода ставит в очередь метод `DownloadDataTaskAsync` для обработки полученного результата.
12. Поток порта завершения ввода-вывода помечает, что задача `Task`, возвращенная методом `DownloadDataTaskAsync`, завершена.
13. Поток порта завершения ввода-вывода исполняет код обработки завершения внутри `Task`; этот код вызывает метод `Post` запомненного контекста `SynchronizationContext` (поток пользовательского интерфейса) для продолжения.
14. Поток порта завершения ввода-вывода освобождается для обработки других событий ввода-вывода.
15. Поток пользовательского интерфейса находит команду, отправленную методом `Post`, и возобновляет исполнение второй половины метода `GetFaviconAsync` – до конца.
16. Покидая метод `GetFaviconAsync`, поток пользовательского интерфейса помечает, что задача `Task`, возвращенная методом `GetFaviconAsync`, завершена.
17. Поскольку на этот раз текущий контекст синхронизации совпадает с запомненным, вызывать метод `Post` не нужно, и поток пользовательского интерфейс продолжает работать синхронно.

В WPF эта логика ненадежна, потому что WPF часто создает новые объекты `SynchronizationContext`. И хотя все они эквивалентны, Task Parallel Library считает, что должна вызвать метод Post.

18. Поток пользовательского интерфейса возобновляет исполнение второй половины метода `GetButton_OnClick` – до конца.

Всё это выглядит довольно сложно, но я думаю, что детально выписать все шаги полезно. Обратите внимание, что *абсолютно все* строки моей программы исполнялись в потоке пользовательского интерфейса. Поток порта завершения ввода-вывода только вызывал метод `Post`, чтобы отправить команду потоку пользовательского интерфейса, который затем исполнял вторые половины моих методов.

Когда не следует использовать SynchronizationContext

В каждом подклассе `SynchronizationContext` метод `Post` реализован по-своему. Как правило, вызов этого метода обходится сравнительно дорого. Чтобы избежать накладных расходов, .NET не вызывает `Post`, если запомненный контекст синхронизации совпадает с текущим на момент завершения задачи. Если в этот момент посмотреть на стек вызовов в отладчике, то он окажется «перевернутым» (если не обращать внимания на вызовы самого каркаса). Тот метод, который с точки зрения программиста является самым глубоко вложенным, то есть вызывается другими методами, на самом деле вызывает остальные методы по своем завершении.

Однако если контексты синхронизации различаются, то необходим дорогостоящий вызов `Post`. Если производительность стоит на первом месте или речь идет о библиотечном коде, которому безразлично, в каком потоке выполняться, то, возможно, не имеет смысла нести такие расходы. В таком случае следует вызвать метод `ConfigureAwait` объекта `Task`, перед тем как ждать его. Тогда при возобновлении исполнения не будет вызываться метод `Post` запомненного контекста `SynchronizationContext`.

```
byte[] bytes =
  await client.DownloadDataTaskAsync(url).ConfigureAwait(false);
```

Однако метод `ConfigureAwait` не всегда делает то, что вы ожидаете. Он задуман как способ информирования .NET о том, что вам безразлично, в каком потоке будет возобновлено выполнение, а не как неукоснительный приказ. Что происходит в действительности, зависит от того, в каком потоке завершилось выполнение ожидаемой задачи. Если этот поток не очень важен, например взят из пула, то исполнение кода в нем и продолжится. Но если поток по какой-то причине важен, то .NET предпочтет освободить его для других дел,

а исполнение вашего метода продолжить в потоке, взятом из пула. Решение о том, важен поток или нет, принимается на основе анализа текущего контекста синхронизации.

Взаимодействие с синхронным кодом

Допустим, вы работаете с уже существующим приложением и, хотя написанный вами новый код является асинхронным и следует паттерну TAP, остается необходимость взаимодействовать со старым синхронным кодом. Конечно, при этом обычно теряются преимущества асинхронности, но, возможно, в будущем вы планируете переписать код в асинхронном стиле, а пока надо соорудить какую-то «времянку».

Вызвать синхронный код из асинхронного просто. Если имеется блокирующий API, то достаточно выполнить его в потоке, взятом из пула, воспользовавшись методом `Task.Run`. Правда, в таком случае захватывается поток, но это неизбежно.

```
var result = await Task.Run(() => MyOldMethod());
```

Вызов асинхронного кода из синхронного или реализация синхронного API тоже выглядит просто, но тут есть скрытые проблемы. В классе `Task` имеется свойство `Result`, обращение к которому блокирует вызывающий поток до завершения задачи. Его можно использовать в тех же местах, что `await`, но при этом не требуется, чтобы метод был помечен ключевым словом `async` или возвращал объект `Task`. И в этом случае один поток занимается – на этот раз вызывающий (то есть тот, что блокируется).

```
var result = AlexsMethodAsync().Result;
```

Хочу предупредить: эта техника не будет работать, если используется в контексте синхронизации с единственным потоком, например пользовательского интерфейса. Вдумайтесь – что мы хотим от потока пользователя интерфейса? Он блокируется в ожидании завершения задачи `Task`, возвращенной методом `AlexsMethodAsync`. `AlexsMethodAsync`, скорее всего, вызвал какой-то еще TAP-метод и ждет его завершения. Когда операция завершится, запомненный `SynchronizationContext` (поток пользовательского интерфейса) используется для отправки (методом `Post`) команды возобновления `AlexsMethodAsync`. Однако поток пользовательского интерфейса

никогда не получит это сообщение, потому что он по-прежнему заблокирован. Получилась взаимоблокировка. К счастью, эта ошибка приводит к стопроцентно воспроизводимой взаимоблокировке, так что отладить ее нетрудно.

При должной осмотрительности проблемы взаимоблокировки можно избежать, переключившись на поток из пула до запуска асинхронного кода. Тогда будет запомнен контекст синхронизации, связанный с пулом потоков, а не с потоком пользовательского интерфейса. Но это некрасивое решение, лучше потратить время на то, чтобы сделать вызывающий код асинхронным.

```
var result = Task.Run(() => AlexsMethodAsync()).Result;
```

ГЛАВА 9. Исключения в асинхронном коде

В синхронном коде исключение распространяется вверх по стеку вызовов, пока не достигнет блока `try-catch`, способного его обработать, либо не выйдет за пределы вашего кода. В асинхронном коде, особенно после возобновления кода, погруженного в `await`, текущий стек вызовов имеет мало общего с тем, что интересует программиста, и состоит по большей части из вызовов методов каркаса, обеспечивающих возобновление async-метода. В такой ситуации возникшее исключение было бы невозможно перехватить в вызывающем коде, а трассировка вызовов вообще не имела бы никакого смысла. Поэтому компилятор C# изменяет поведение исключений, делая его более полезным.

Исходный стек вызовов тем не менее можно просмотреть в отладчике.

Исключения в async-методах, возвращающих Task

Большинство написанных вами async-методов возвращают значение типа `Task` или `Task<T>`. Цепочка таких методов, в которой каждый ожидает завершения следующего, представляет собой асинхронный аналог стека вызовов в синхронном коде. Компилятор C# прилагает максимум усилий к тому, чтобы исключения, возбуждаемые в этих методах, вели себя так же, как в синхронном случае. В частности, блок `try-catch`, окружающий ожидаемый async-метод, перехватывает исключения, возникшие внутри этого метода.

```
async Task Catcher()
{
 try
 {
  await Thrower();
 }
 catch (AlexsException)
 {
  // Исключение будет обработано здесь
 }
}

async Task Thrower()
{
 await Task.Delay(100);
 throw new AlexsException();
}
```

До тех пор пока исполнение не дошло до первого `await`, синхронный стек вызовов и цепочка асинхронных методов ничем не отличаются. Поведение исключений в этой точке несколько изменено ради согласованности, но это изменение гораздо более скромное.

Для этого C# перехватывает все исключения, возникшие в вашем async-методе. Перехваченное исключение помещается в объект `Task`, который был возвращен вызывающей программе. Объект `Task` переходит в состояние *Faulted*. Если задача завершилась с ошибкой, то ожидающий ее метод не возобновится как обычно, а получит исключение, возбужденное в коде внутри `await`.

Исключение, повторно возбужденное оператором `await`, – это тот же самый объект, который был порожден в предложении `throw`. По мере того как он распространяется вверх по стеку вызовов, растет ассоциированная с ним трассировка стека. Это может показаться странным тому, кто пробовал повторно возбуждать исключение вручную, например в написанном вручную асинхронном коде. Дело в том, что здесь используется новая возможность типа .NET `Exception`.

Ниже приведен пример трассировки стека для цепочки из двух async-методов. Мой код выделен полужирным шрифтом.

```
System.NullReferenceException: Object reference not set to an instance of an object.
  at FaviconBrowser.MainWindow.<GetFavicon>d__c.MoveNext() in
MainWindow.xaml.cs:line 74
--- End of stack trace from previous location where exception was thrown ---
  at System.Runtime.CompilerServices.TaskAwaiter.ThrowForNonSuccess(Task task)
  at
```

```
System.Runtime.CompilerServices.TaskAwaiter.HandleNonSuccessAndDebuggerNotification(T
ask task)
  at System.Runtime.CompilerServices.TaskAwaiter`1.GetResult()
  at FaviconBrowser.MainWindow.<GetButton_OnClick>d__0.MoveNext() in
MainWindow.xaml.cs:line 41
--- End of stack trace from previous location where exception was thrown ---
  at System.Runtime.CompilerServices.AsyncMethodBuilderCore.<ThrowAsync>b__0(Object
state)
at ... Методы каркаса
```

Упоминание метода `MoveNext` связано с преобразованием кода, осуществляемым компилятором (см. главу 14). Мои методы перемежаются методами каркаса, но тем не менее можно получить представление о том, какая последовательность моих методов привела к исключению.

Незамеченные исключения

Одно из важных различий между синхронным и асинхронным кодом – место, где возбуждается исключение, возникшее в вызванном методе. В async-методе оно возбуждается там, где находится оператор `await`, а не в точке фактического вызова метода. Это становится очевидным, если разделить вызов и `await`.

```
// Здесь AlexsException никогда не возбуждается
Task task = Thrower();
try
{
 await task;
}
catch (AlexsException)
{
 // Исключение обрабатывается здесь
}
```

Очень легко забыть о необходимости ожидать завершения async-метода, особенно если тот возвращает объект неуниверсального класса `Task`, потому что программе не требуется значение результата. Но поступить таким образом – все равно, что включить пустой блок `catch`, который перехватывает и игнорирует все исключения. Это плохое решение, потому что может привести к некорректному состоянию программы и к тонким ошибкам, проявляющимся далеко от места возникновения. Возьмите за правило всегда ожидать завершения любого async-метода, если не хотите потом провести долгие часы в обществе отладчика.

Такое игнорирование исключений – новшество, появившееся в .NET после включения механизма async. Если вы ожидаете, что исключение, возникшее в библиотечном коде Task Parallel Library, будет повторно возбуждено в потоке финализатора, то имейте в виду – в .NET 4.5 этого уже не происходит.

Исключения в методах типа async void

Ждать с помощью `await` завершения async-метода, возвращающего `void`, невозможно, поэтому его поведение в части исключений должно быть другим. Заведомо не хотелось бы, чтобы исключения, возбуждаемые такими методами, оставались незамеченными. А хотим мы, чтобы исключение, покинувшее метод типа `async void`, было повторно возбуждено в вызывающем потоке:

- если в месте, где был вызван async-метод, существовал контекст синхронизации `SynchronizationContext`, то исключение посылается ему методом `Post`;
- в противном случае оно возбуждается в потоке, взятом из пула.

В большинстве случаев обе ситуации приводят к завершению процесса, если только к соответствующему событию не присоединен обработчик необработанных исключений. Маловероятно, что вас это устроит, и именно по этой причине методы типа `async void` следует писать, только если они будут вызываться из внешнего кода или если можно гарантировать отсутствие исключений.

Выстрелил и забыл

Редко, но бывают ситуации, когда неважно, как завершился метод, а ждать его завершения слишком сложно. В таком случае я рекомендую все же возвращать объект `Task`, но передавать его методу, способному обработать исключения. Лично мне нравится такой метод расширения:

```
public static void ForgetSafely(this Task task)
{
 task.ContinueWith(HandleException);
}
```

Здесь метод `HandleException` передает любое исключение системе протоколирования, как, например, в разделе «Создание собственных комбинаторов» выше.

AggregateException и WhenAll

В асинхронных программах приходится иметь дело с ситуацией, которая в синхронном коде вообще невозможна. Метод может возбуждать сразу несколько исключений. Так бывает, например, когда мы используем метод `Task.WhenAll` для ожидания завершения группы асинхронных операций. Ошибкой могут завершиться несколько операций, причем ни одну ошибку нельзя считать первой или самой важной.

Метод `WhenAll` – всего лишь наиболее распространенный механизм, приводящий к возникновению нескольких исключений, но существует немало других способов параллельного выполнения нескольких операций. Поэтому поддержка множественных исключений встроена непосредственно в класс `Task`. Исключение в нем представляется объектом класса `AggregateException`, а не просто `Exception`. Объект `AggregateException` содержит коллекцию других исключений.

Поскольку эта поддержка встроена в класс `Task`, то когда исключение покидает async-метод, создается объект `AggregateException` и фактическое исключение добавляется в него как внутреннее перед помещением в `Task`. Таким образом, в большинстве случаев `AggregateException` содержит только одно внутреннее исключение, но метод `WhenAll` создает `AggregateException` с несколькими исключениями.

Все это происходит вне зависимости от того, произошло ли исключение до первого `await` или позже. Исключения, имевшие место до первого `await`, легко можно было бы возбудить синхронно, но тогда они возникали бы при вызове метода, а не в результате `await` в вызывающей программе, что было бы непоследовательно.

С другой стороны, если исключение повторно возбуждается оператором `await`, нам необходим какой-то компромисс. Оператор `await` должен возбуждать исключение того же типа, что было изначально возбуждено в async-методе, а не типа `AggregateException`. Поэтому ничего не остается, кроме как возбудить первое внутреннее

исключение. Однако, перехватив его, мы можем обратиться к объекту `Task` напрямую и получить от него объект `AggregateException`, содержащий полный список исключений.

```
Task<Image[]> allTask = Task.WhenAll(tasks);
try
{
 await allTask;
}
catch
{
 foreach (Exception ex in allTask.Exception.InnerExceptions)
 {
   // Обработать исключение
 }
}
```

Синхронное возбуждение исключений

Паттерн TAP допускает синхронное возбуждение исключений, но только если исключение служит для индикации ошибок в порядке вызова метода, а не ошибок, возникающих в процессе его выполнения. Мы видели, что async-метод перехватывает любое исключение и помещает его в объект `Task`, вне зависимости от того, произошло оно до или после первого `await`. Поэтому если вы хотите возбудить исключение синхронно, то придется прибегнуть к трюку: воспользоваться синхронным методом, который проверяет ошибки перед вызовом асинхронного.

```
private Task<Image> GetFaviconAsync(string domain)
{
 if (domain == null) throw new ArgumentNullException("domain");
 return GetFaviconAsyncInternal(domain);
}

private async Task<Image> GetFaviconAsyncInternal(string domain)
{
 ...
```

Получающуюся при таком подходе трассировку стека интерпретировать немного проще. Стоит ли овчинка выделки? Лично я сомневаюсь. Тем не менее, этот пример полезен для лучшего понимания.

Блок finally в async-методах

Наконец, в async-методе разрешено использовать блок `try-finally`, и работает он в основном, как и ожидается. Гарантируется, что блок `finally` будет выполнен до того, как программа покинет содержащий его метод, неважно произошло это в ходе нормального выполнения или в результате исключения внутри блока `try`.

Но эта гарантия таит в себе скрытую опасность. В случае async-метода нет гарантии, что поток выполнения вообще когда-либо покинет метод. Легко написать метод, который доходит до `await`, приостанавливается, а затем про него забывают, и в конечном итоге он достается сборщику мусора.

```
async void AlexsMethod()
{
 try
 {
   await DelayForever();
 }
 finally
 {
   // Сюда мы никогда не попадем
 }
}

Task DelayForever()
{
 return new TaskCompletionSource<object>().Task;
}
```

С помощью `TaskCompletionSource` я создал задачу-марионетку и забыл про нее. Поскольку метод `AlexsMethod` больше не исполняется ни в каком потоке, то он не сможет ни возобновиться, ни возбудить исключение. Рано или поздно его уберет сборщик мусора.

Таким образом, в асинхронных методах гарантия, предоставляемая ключевым словом `finally`, оказывается гораздо слабее.

ГЛАВА 10. Организация параллелизма с помощью механизма async

Механизм async открывает широкие возможности для использования параллелизма, характерного для современных машин. Это языковое средство позволяет структурировать программу гораздо проще, чем было возможно раньше.

Мы уже видели, как пишется простой код, запускающий несколько длительных операций, например сетевых запросов, которые затем исполняются параллельно. Благодаря таким средствам, как `WhenAll`, асинхронный код весьма эффективен для выполнения подобных операций, не связанных с локальными вычислениями. Если же речь идет о локальных вычислениях, то сам по себе механизм async не поможет. Пока не достигнута точка асинхронности, весь код выполняется синхронно в вызывающем потоке.

await и блокировки

Простейший способ распараллелить программу состоит в том, что запланировать работу в разных потоках. Это легко сделать с помощью метода `Task.Run`, так как он возвращает объект `Task`, с которым мы можем обращаться, как с любой другой длительной операцией. Но наличие нескольких потоков сопряжено с риском небезопасного доступа к разделяемым объектам в памяти.

Традиционное решение, заключающееся в использовании ключевого слова `lock`, в асинхронном коде более сложно, о чем мы говорили в разделе «Блоки lock» выше. Поскольку в блоке `lock` нельзя использовать оператор `await`, то не существует способа предотвратить исполнение конфликтующего кода во время ожидания. На самом

деле, лучше вообще избегать удержания ресурсов на время исполнения `await`. Весь смысл асинхронности как раз и состоит в том, чтобы освобождать ресурсы на время ожидания, и программист должен понимать, что в это время может произойти всё что угодно.

```
lock (sync)
{
 // Подговиться к асинхронной операции
}

int myNum = await AlexsMethodAsync();

lock (sync)
{
 // Использовать результат асинхронной операции
}
```

Рассмотрим в качестве примера поток пользовательского интерфейса. Такой поток существует в единственном экземпляре, поэтому в некотором смысле его можно считать блокировкой. Если некий код работает в потоке пользовательского интерфейса, то в каждый момент времени заведомо выполняется только одна его команда. Но даже в этом случае во время ожидания может случиться всё что угодно. Если в ответ на нажатие пользователем кнопки вы запускаете сетевой запрос, то ничто не помешает пользователю нажать ту же кнопку еще раз, пока ваша программа ждет результата. В этом и состоит смысл асинхронности в приложениях с пользовательским интерфейсом: интерфейс остается отзывчивым и делает всё, что просит пользователь, даже если это опасно.

Но мы по крайней мере можем указать в программе точки, в которых могут происходить неприятности. Следует взять за правило ставить `await` только в местах, где это безопасно, и быть готовым к тому, что после возобновления состояние мира может измениться. Иногда это означает, что необходимо произвести вторую, кажущуюся бессмысленной, проверку, прежде чем продолжать работу.

```
if (DataInvalid())
{
 Data d = await GetNewData();

 // Внутри await могло произойти всё что угодно
 if (DataInvalid())
 {
   SetNewData(d);
 }
}
```

Акторы

Я сказал, что поток пользовательского интерфейса можно рассматривать, как блокировку, просто потому что такой поток всего один. На самом деле, было бы правильно назвать его актором. Актор – это поток, который отвечает за определенный набор данных, причем никакой другой поток не вправе обращаться к этим данным. В данном случае только поток пользовательского интерфейса имеет доступ к данным, составляющим пользовательский интерфейс. При таком подходе становится гораздо проще обеспечить безопасность кода пользовательского интерфейса, потому что единственное место, где что-то может произойти, – это `await`.

Если говорить более общо, то программы можно строить из компонентов, которые работают в одном потоке и отвечают за определенные данные. Такая модель акторов легко адаптируется к параллельным архитектурам, так как акторы могут работать на разных ядрах. Она эффективна и в программах общего вида, когда имеются компоненты, наделенные состоянием, которым требуется безопасно манипулировать.

Существуют и другие парадигмы, например программирование потоков данных (dataflow programming), чрезвычайно эффективные в *естественно параллельных* (embarrassingly parallel) задачах, когда имеется много независимых друг от друга вычислений, допускающих очевидное распараллеливание. Акторы – подходящий выбор для тех задач, где очевидного распараллеливания не существует.

На первый взгляд, между программированием с помощью акторов и с помощью блокировок очень много общего. В частности, в обоих моделях присутствует идея о том, что в каждый момент времени доступ к определенным данным разрешен только одному потоку. Разница же в том, что один поток не может принадлежать сразу нескольким акторам. Вместо того чтобы удерживать ресурсы одного актора на время выполнения кода другого актора, поток должен сделать асинхронный вызов. Во время ожидания вызывающий поток может заняться другими делами.

Модель акторов лучше масштабируется, чем модель программирования с блокировками разделяемых объектов. Модель, основанная на доступе к общему адресному пространству из нескольких ядер, постепенно отдаляется от реальности. Если вам доводилось использовать в программе блокировки, то вы знаете, как просто «нарваться» на взаимоблокировки и состояния гонки.

Использование акторов в C#

Конечно, никто не мешает программировать в стиле модели акторов вручную, но есть и библиотеки, упрощающие эту задачу. Так, библиотека NAct (http://code.google.com/p/n-act/) в полной мере задействует механизм async в C# для превращения обычных объектов в акторов, в результате чего все обращения к ним производятся в отдельном потоке. Достигается это с помощью заместителя, который обертывает объект, делая его актором.

Рассмотрим пример. Пусть требуется реализовать криптографическую службу, которой для шифрования потока данных необходима последовательность псевдослучайных чисел. Здесь имеются две задачи, требующие большого объема вычислений, и мы хотели бы решать их параллельно:

- генерация псевдослучайных чисел;
- использование их для потокового шифрования.

Мы будем рассматривать только актора, играющего роль генератора случайных чисел. Библиотеке NAct необходим интерфейс, имея который она сможет создать для нас заместителя. Реализуем этот интерфейс:

```
public interface IRndGenerator : IActor
{
 Task<int> GetNextNumber();
}
```

Этот интерфейс должен наследовать пустому маркерному интерфейсу `IActor`. Все методы интерфейса должны возвращать один из совместимых с механизмом async типов:

- `void`
- `Task`
- `Task<T>`

Теперь можно реализовать сам класс генератора.

```
class RndGenerator : IRndGenerator
{
 public async Task<int> GetNextNumber()
 {
   // Безопасная генерация случайного числа – медленная операция
   ...
   return num;
 }
}
```

Неожиданно здесь только то, что никаких неожиданностей нет. Это самый обычный класс. Чтобы им воспользоваться, мы должны сконструировать объект и передать его NAct, чтобы та обернула его, превратив в актора.

```
IRndGenerator rndActor = ActorWrapper.WrapActor(new RndGenerator());

Task<int> nextTask = rndActor.GetNextNumber();
foreach (var chunk in stream)
{
 int rndNum = await nextTask;

 // Начать генерацию следующего числа
 nextTask = rndActor.GetNextNumber();

 // Использовать rndNum для шифрования блока - медленная операция
 ...
}
```

На каждой итерации цикла я жду случайного числа, а затем запускаю процедуру генерации следующего, пока сам занимаюсь медленной операцией. Поскольку `rndActor` – актор, NAct возвращает объект `Task` немедленно, а генерацию производит в потоке `RndGenerator`. Теперь два вычисления осуществляются параллельно, что позволяет лучше использовать ресурсы процессора. Благодаря встроенному в язык механизму async трудная задача программирования выражается очень естественно.

Здесь не место вдаваться в детали работы с библиотекой NAct, но надеюсь, я рассказал достаточно для понимания того, насколько просто использовать модель акторов. У нее есть и другие возможности, в частности генерация событий в нужном потоке и интеллектуальное разделение потоков между простаивающими акторами. В общем и целом это означает, что модель хорошо масштабируется в реальных системах.

Библиотека Task Parallel Library Dataflow

Еще одно полезное средство параллельного программирования, использование которого в C# упрощается за счет применения механизма async, – это парадигма программирования потоков данных. В этой модели определяется последовательность операций, которые необходимо произвести над входными данными, а система автоматичес-

ки распараллеливает их. Microsoft предлагает для этой цели библиотеку TPL Dataflow, которую можно скачать из репозитория NuGet (https://nuget.org/packages/Microsoft.Tpl.Dataflow).

Техника программирования потоков данных особенно полезна, когда самой критичной с точки зрения производительности частью программы является преобразование данных. Ничто не мешает одновременно использовать акторов и программирование потоков данных. В этом случае один актор, на который приходится большой объем вычислений, применяет для их распараллеливания потоки данных.

Идея библиотеки TPL Dataflow заключается в передаче сообщений между *блоками*. Чтобы создать сеть потоков данных, мы сцепляем блоки, реализующие два интерфейса:

`ISourceBlock<T>`
: Нечто такое, у чего можно запросить сообщения типа `T`.

`ITargetBlock<T>`
: Нечто такое, чему можно передать сообщения.

В интерфейсе `ISourceBlock<T>` определен метод `LinkTo`, который принимает объект-получатель типа `ITargetBlock<T>` и связывает его с источником, так что каждое сообщение, порожденное объектом `ISourceBlock<T>`, передается объекту `ITargetBlock<T>`. Большинство блоков реализуют оба интерфейса, быть может, с разными параметрами-типами, так чтобы блок мог потреблять сообщения одного типа и порождать сообщения другого типа.

Интерфейсы можно реализовывать самостоятельно, но гораздо чаще используются встроенные блоки, например:

`ActionBlock<T>`
: Конструктору объекта `ActionBlock<T>` передается делегат, который вызывается для каждого сообщения. Класс `ActionBlock<T>` реализует только интерфейс `ITargetBlock<T>`.

`TransformBlock<TIn, TOut>`
: Конструктору также передается делегат, только на этот раз он является функцией, возвращающей значение. Это значение становится сообщением, которое передается следующему блоку. Класс `TransformBlock<TIn, TOut>` реализует интерфейсы `ITargetBlock<TIn>` и `ISourceBlock<TOut>`. Это параллельная версия LINQ-метода `Select`.

`JoinBlock<T1, T2,...>`
: Объединяет несколько входных потоков в один выходной поток, состоящий из кортежей.

Есть много других встроенных блоков, из которых можно конструировать вычисление конвейерного типа. Без дополнительного программирования блоки работают как параллельный конвейер, но каждый блок способен одновременно обрабатывать только одно сообщение. Этого достаточно, если все блоки работают примерно одинаковое время, но если какая-то стадия оказывается существенно медленнее всех остальных, то можно сконфигурировать объекты `ActionBlock<T>` и `TransformBlock<TIn, TOut>`, так чтобы они работали параллельно в отдельном блоке, то есть по сути дела расщепляли себя на много идентичных блоков, которые сообща трудятся над некоторой задачей.

Библиотека TPL Dataflow получает выигрыш от механизма async, потому что делегаты, передаваемые блокам `ActionBlock<T>` и `TransformBlock<TIn, TOut>`, могут быть асинхронными и возвращать значения типа `Task` или `Task<T>` соответственно. Это особенно важно, когда указанные делегаты выполняют длительные удаленные операции, потому что их можно запускать параллельно, не занимая зря потоки. Кроме того, взаимодействие с блоками потоков данных извне также полезно организовывать асинхронно, и для облегчения этой задачи существуют такие TAP-методы, как `SendAsync`, расширяющий класс `ITargetBlock<T>`.

ГЛАВА 11. Автономное тестирование асинхронного кода

Я хочу коротко остановиться на вопросе об автономном тестировании асинхронного кода. Простейший подход работает не очень хорошо, но при поддержке со стороны каркаса автономного тестирования нетрудно написать тесты, вызывающие async-методы.

Проблема автономного тестирования в асинхронном окружении

Async-методы быстро возвращают управление, обычно вместе с объектом `Task`, который станет завершенным в будущем. К такому объекту мы, как правило, применяем оператор `await`; посмотрим, как это выглядит в автономном тесте.

```
[TestMethod]
public async void AlexsTest()
{
 int x = await AlexsMethod();
 Assert.AreEqual(3, x);
}
```

Чтобы можно было использовать оператор `await`, я пометил тестовый метод ключевым словом `async`. Но у этого действия есть очень важный побочный эффект. Теперь тестовый метод также быстро возвращает управление, а завершается в будущем. Фактически оказывается, что тестовый метод возвращается, не возбуждая исключений, сразу по достижении `await`, поэтому каркас тестирования – в данном случае MSTest – помечает его как успешный.

Поскольку тестовый метод имеет тип `async void`, то все возбужденные в нем исключения повторно возбуждаются в вызывающем контексте синхронизации, где они либо игнорируются, либо приводят к отказу совершенно другого теста, запущенного позднее.

Но настоящая опасность состоит в том, что все тесты считаются успешными вне зависимости от фактического результата.

Написание работающих асинхронных тестов вручную

Один из способов решить эту проблему – не помечать тестовые методы ключевым словом `async`. Тогда мы должны синхронно ждать результата асинхронного вызова.

```
[TestMethod]
public void AlexsTest()
{
 int x = AlexsMethod().Result;
 Assert.AreEqual(3, x);
}
```

Чтение свойства `Result` приводит к блокирующему ожиданию завершения задачи `Task`. Это работает и тест благополучно не проходит, если не должен проходить. Если метод `AlexsMethod` возбудит исключение, то оно будет повторно возбуждено при обращении к `Result`, правда, в отличие от исключения, повторно возбуждаемого оператором `await`, оно будет обернуто объектом `AggregateException`.

Но теперь вы уже понимаете, насколько некрасиво выглядит использование блокирующего свойства `Result` объекта `Task`. К тому же, как мы видели в разделе «Взаимодействие с синхронным кодом», это еще и опасно, если программа работает в однопоточном контексте синхронизации. По счастью, ни один из популярных каркасов автономного тестирования по умолчанию не пользуется однопоточным `SynchronizationContext`. И всё равно при таком решении впустую расходуется один поток, так что оно не оптимально.

Поддержка со стороны каркаса автономного тестирования

Некоторые каркасы автономного тестирования явно поддерживают механизм async. Они позволяют создавать тестовые методы, возвра-

щающие `Task`, и, следовательно, помечать их ключевым словом async. Каркас будет ждать завершения задачи `Task` и только после этого пометит тест как прошедший и перейдет к следующему тесту.

На момент написания этой книги такой стиль поддерживали каркасы xUnit.net и MSTest. Я полагаю, что в ближайшее время поддержка будет добавлена и в другие популярные каркасы, пусть даже в виде небольшого дополнительного модуля.

```
[TestMethod]
public async Task AlexsTest()
{
 int x = await AlexsMethod();
 Assert.AreEqual(3, x);
}
```

Пожалуй, это самый чистый способ написания автономных тестов для асинхронного кода, поскольку ответственность за управление потокам перекладывается на каркас тестирования.

ГЛАВА 12. Механизм async в приложениях ASP.NET

Большинство разработчиков на платформе .NET пишут веб-приложения. Механизм async открывает новые возможности для повышения производительности серверного кода, поэтому рассмотрим эту тему подробнее.

Преимущества асинхронного веб-серверного кода

При обработке запроса веб-сервером отзывчивость не играет такой роли, как в программах с пользовательским интерфейсом. Производительность веб-сервера измеряется в терминах пропускной способности, задержки и постоянства этих характеристик.

Асинхронный код на нагруженном веб-сервере требует меньше потоков, чем синхронный, обслуживающий ту же нагрузку. Каждый поток потребляет память, а объем памяти часто является узким местом веб-сервера. Если памяти на хватает, то сборщик мусора запускается чаще и вынужден выполнять больше работы. Если недостаточно физической памяти, то начинается выгрузка страниц на диск, а в случае, когда выгруженные страницы скоро снова оказываются необходимы, работа системы существенно замедляется.

Возможность писать асинхронный веб-серверный код появилась в ASP.NET начиная с версии 2.0, но без поддержки со стороны языка это было нелегко. Большинство разработчиков считало, что проще и рентабельнее установить дополнительные серверы и балансировать нагрузку между ними. Но с выходом C# 5.0 и .NET 4.5 писать асинхронный код стало настолько просто, что каждому имеет смысл воспользоваться присущей ему эффективностью.

Использование async в ASP.NET MVC 4

Версия ASP.NET MVC 4 и более поздние при запуске на платформе .NET 4.5 и выше в полной мере поддерживают паттерн TAP, поэтому можно использовать async-методы. В приложении MVC важнейшим местом для задействования асинхронности является контроллер. Методы контроллера можно пометить ключевым словом async и возвращать из них значение типа `Task<ActionResult>`:

```
public class HomeController : Controller
{
 public async Task<ActionResult> Index()
 {
  ViewBag.Message = await GetMessageAsync();
  return View();
 }
 ...
```

Это решение опирается на тот факт, что для запросов, занимающих длительное время, желательно иметь асинхронный API. Многие системы объектно-реляционного отображения (ORM) пока не поддерживают асинхронных вызовов, но в API, основанный на классе .NET `SqlConnection`, такая поддержка встроена.

Использование async в предыдущих версиях ASP.NET MVC

В версиях младше MVC 4 поддержка асинхронных контроллеров не основана на паттерне TAP и устроена более сложно. Вот один из способов адаптировать TAP-метод контроллера в духе MVC 4 к паттерну, применявшемуся в предыдущих версиях MVC.

```
public class HomeController : AsyncController
{
 public void IndexAsync()
 {
  AsyncManager.OutstandingOperations.Increment();
  Task<ActionResult> task = IndexTaskAsync();
  task.ContinueWith(_ =>
  {
   AsyncManager.Parameters["result"] = task.Result;
```

```
      AsyncManager.OutstandingOperations.Decrement();
    });
  }

  public ActionResult IndexCompleted(ActionResult result)
  {
    return result;
  }

  private async Task<ActionResult> IndexTaskAsync()
  {
...
```

Во-первых, класс контроллера должен наследовать классу `AsyncController`, а не `Controller`, так как именно это позволяет применить асинхронный паттерн. А означает этот паттерн, что для каждого действия существуют два метода: *Action*`Async` и *Action*`Completed`. Объект `AsyncManager` управляет временем жизни асинхронного запроса. Когда свойство `OutstandingOperations` обращается в нуль, вызывается метод *Action*`Completed`. Работа с объектом `Task` организуется вручную с помощью метода `ContinueWith`, а результат передается методу *Action*`Completed` в словаре `Parameters`.

Для простоты я опустил в этом примере обработку исключений. В общем и целом, решение не блещет красотой, но, подготовив такую инфраструктуру, все же можно использовать async-методы.

Использование async в ASP.NET Web Forms

Для стандартных вариантов ASP.NET и Web Forms не существует версий, отдельных от версии каркаса .NET, с которой они поставляются. В .NET 4.5 ASP.NET в классе `Page` поддерживаются методы типа `async void`, например `Page_Load`.

```
protected async void Page_Load(object sender, EventArgs e)
{
 Title = await GetTitleAsync();
}
```

Такая реализация может показаться странной. Как ASP.NET узнает, что метод типа `async void` завершился? Было бы более естественно возвращать задачу `Task`, завершения которой ASP.NET могла бы дождаться, перед тем как приступать к отрисовке страницы, – так же, как это делается в MVC 4. Тем не менее, возможно, из соображений

обратной совместимости, требуется, чтобы методы возвращали `void`. А для решения указанной проблемы в ASP.NET используется специальный подкласс `SynchronizationContext`, который ведет учет всем асинхронным операциям и позволяет перейти к следующему этапу обработки только после того, как все они завершены.

При исполнении асинхронного кода в контексте синхронизации ASP.NET имейте в виду, что этот контекст однопоточный. Попытка организовать блокирующее ожидание `Task`, например путем чтения свойства `Result`, скорее всего, приведет к взаимоблокировке, так как операторы `await` с более глубоким уровнем вложенности не смогут воспользоваться контекстом `SynchronizationContext` для возобновления.

ГЛАВА 13. Механизм async в приложениях WinRT

Для читателей, не знакомых с WinRT, приведу краткий обзор этой технологии.

Что такое WinRT?

WinRT (или Windows Runtime) – это группа API, используемых в приложениях, работающих на платформах Windows 8 и Windows RT для процессоров ARM. Одна из проектных целей WinRT API – обеспечить отзывчивость за счет асинхронного программирования. Все методы, для выполнения которых может потребоваться более 50 мс, асинхронны.

При проектировании WinRT ставилась задача обеспечить единообразный доступ из трех разных технологий: .NET, JavaScript и машинный код (обычно на C++). Для этого все API определены в едином формате метаданных, который называется WinMD. Программу на любом из перечисленных языков можно откомпилировать вместе с WinMD-определением API, не прибегая к зависящим от языка оберткам. Этот подход называется проецированием – каждый компилятор или интерпретатор проецирует тип WinRT на тип своего языка.

Формат WinMD основан на формате метаданных в сборках .NET, поэтому имеющиеся конструкции очень похожи на применяемые в .NET: классы, интерфейсы, методы, свойства, атрибуты и т. д. Но есть и различия; например, универсальные типы допустимы, а универсальные методы – нет.

Большая часть WinRT реализована на машинном языке, но можно писать WinRT-компоненты также на C#, и эти компоненты будут доступны из любого поддерживаемого языка.

Поскольку интерфейсы WinRT не ориентированы специально на .NET, в API написанного вами WinRT-компонента не могут использоваться многие типы .NET. Многие интерфейсы коллекций, например IList<T>, проецируются автоматически. Но не класс Task, в котором слишком много специфичного для .NET поведения.

Интерфейсы IAsyncAction и IAsyncOperation<T>

Эти два интерфейса WinRT эквивалентны соответственно Task и Task<T>. В асинхронных методах WinRT используется паттерн, аналогичный TAP, только методы должны возвращать значения типа IAsyncAction или IAsyncOperation<T>. Оба интерфейса очень похожи, поэтому далее в этой главе, упоминая любой из них, я буду иметь в виду оба.

Ниже приведен пример метода из класса WinRT SyndicationClient, который читает RSS-ленту.

```
IAsyncOperation<SyndicationFeed> RetrieveFeedAsync(Uri uri)
```

Напомню, что IAsyncAction и IAsyncOperation<T> – интерфейсы WinMD, а не .NET. Это различие несколько сбивает с толку, потому что те и другие можно использовать в программах на C#, как обычные интерфейсы .NET.

Как и TAP-методы, методы WinRT такого вида сразу же возвращают значение типа IAsyncOperation<T>, которое трактуется как обещание вернуть объект SyndicationFeed в будущем. Ключевое слово await можно использовать для объектов типа IAsyncOperation<T> точно так же, как для объектов Task, то есть метод RetrieveFeedAsync можно вызвать следующим образом:

```
SyndicationFeed feed = await rssClient.RetrieveFeedAsync(url);
```

Оператор await применим к любому типу, обладающему методами специального вида, которые обеспечивают требуемое поведение. В классе Task такие методы есть, а в интерфейсе IAsyncOperation<T> – нет. Однако удовлетворить требованиям паттерна можно и за счет методов расширения IAsyncOperation<T>, которые .NET предоставляет для согласования с await.

Иногда необходим доступ к объекту Task, представляющему асинхронный вызов WinRT; например, для передачи комбинатору Task.WhenAll или для использования метода ConfigureAwait.

Для создания такого объекта служит еще один метод расширения IAsyncOperation<T> – AsTask:

```
Task<SyndicationFeed> task = rssClient.RetrieveFeedAsync(url).AsTask();
```

Метод AsTask возвращает обычный объект Task, который можно использовать как угодно.

Отмена

В версии TAP для WinRT выбран другой подход к отмене операции. Если в .NET TAP в качестве дополнительного параметра методу передается объект CancellationToken, то в WinRT механизм отмены встроен в возвращаемый объект типа IAsyncOperation<T>.

```
IAsyncOperation<SyndicationFeed> op = rssClient.RetrieveFeedAsync(url);
op.Cancel();
```

Благодаря этому все асинхронные методы в WinRT допускают возможность отмены. Действительно ли они прекращают работу при вызове Cancel – это другой вопрос; я полагаю, что не все.

У такого решения по сравнению с CancellationToken есть плюсы и минусы, поэтому не удивительно, что для TAP было выбрано одно, а для WinRT – другое. Использование CancellationToken упрощает применение одного маркера к нескольким методам, тогда как встраивание отмены в тип возвращенного обещания делает API чище.

Впрочем, использовать метод Cancel напрямую не следует, потому что у метода расширения AsTask имеется перегруженный вариант, которые принимает стандартный тип .NET CancellationToken и организует все необходимые действия автоматически:

```
... = await rssClient.RetrieveFeedAsync(url).AsTask(cancellationToken);
```

Теперь можно использовать класс CancellationTokenSource, как обычно.

Информирование о ходе выполнения

И в этом случае в асинхронных методах WinRT применен иной подход, нежели в TAP. В WinRT механизм информирования о ходе вы-

полнения встроен в тип возвращаемого обещания. Но поскольку эта возможность факультативна, методы, информирующие о ходе выполнения, возвращают специализированные интерфейсы:

- `IAsyncActionWithProgress<TProgress>`
- `IAsyncOperationWithProgress<T, TProgress>`.

Они очевидным образом соответствуют интерфейсам `IAsyncAction` и `IAsyncOperation<T>` и добавляют к ним лишь событие, которое генерируется, когда информация о ходе выполнения изменяется.

Подписаться на это событие проще всего, воспользовавшись еще одним перегруженным вариантом `AsTask`, который принимает значение стандартного типа .NET `IProgress<T>` и организует всю необходимую обвязку:

```
... = await rssClient.RetrieveFeedAsync(url).AsTask(progress);
```

Разумеется, существует также перегруженный вариант, который принимает одновременно `CancellationToken` и `IProgress<T>`.

Реализация асинхронных методов в компоненте WinRT

Своей эффективностью WinRT обязана тому, что библиотеки одинаково просто использовать из любого поддерживаемого языка. При создании собственных библиотек для работы на платформе WinRT вы можете обеспечить такую же гибкость, откомпилировав библиотеку как компонент WinRT, а не как сборку .NET.

В C# сделать это очень просто с одним ограничением: в открытом интерфейсе компонента должны использоваться только типы WinMD или автоматически проецируемые компилятором на типы WinMD. Повторю, что тип `Task` не является ни тем, ни другим, поэтому необходимо вместо него возвращать `IAsyncOperation<T>`.

```
public IAsyncOperation<int> GetTheIntAsync()
{
 return GetTheIntTaskAsync().AsAsyncOperation();
}

private async Task<int> GetTheIntTaskAsync()
{
...
```

И снова задача упрощается благодаря методам расширения, предоставляемым .NET. В простых случаях метод `AsAsyncOperation` делает именно то, что нужно, – преобразует `Task<T>` в `IAsyncOperation<T>`. Соответственно `AsAsyncAction` преобразует `Task` в `IAsyncAction`.

Кстати говоря, методы `AsTask` и `AsAsyncOperation` знают друг о друге и могут определить, написаны ли владелец и потребитель WinMD-метода на .NET. Если это так, то исходный объект `Task` возвращается непосредственно, что повышает производительность.

Но если требуется отмена или информирование о ходе выполнения, то метода `AsAsyncOperation` недостаточно. TAP-методы требуют задания параметра типа `CancellationToken` или `IProgress<T>`, поэтому метод расширения класса `Task` бессилен. Для преобразования из одной модели в другую придется прибегнуть к более сложному средству – методу `AsyncInfo.Run`.

```
public IAsyncOperation<int> GetTheIntAsync()
{
 return AsyncInfo.Run(cancellationToken =>
    GetTheIntTaskAsync(cancellationToken));
}

private async Task<int> GetTheIntTaskAsync(CancellationToken ct)
{
...
```

`AsyncInfo.Run` принимает делегат, который позволяет передать лямбда-выражению объект `CancellationToken`, `IProgress<T>` или тот и другой. Затем эти объекты можно передать TAP-методу. Метод `AsAsyncOperation` можно рассматривать как синоним простейшего перегруженного варианта метода `AsyncInfo.Run`, для которого делегат не принимает никаких параметров.

ГЛАВА 14. Подробно о преобразовании асинхронного кода, осуществляемом компилятором

Механизм async реализован в компиляторе C# при поддержке со стороны библиотек базовых классов .NET. В саму исполняющую среду не пришлось вносить никаких изменений. Это означает, что ключевое слово `await` реализовано путем преобразования к виду, который мы могли бы написать и сами в предыдущих версиях C#. Для изучения генерируемого кода можно воспользоваться декомпилятором, например .NET Reflector.

Это не только интересно, но и полезно для отладки, анализа производительности и других видов диагностики асинхронного кода.

Метод-заглушка

Метод, помеченный ключевым словом async, подменяется заглушкой. При вызове async-метода работает именно эта заглушка. Рассмотрим для примера следующий простой async-метод:

```
public async Task<int> AlexsMethod()
{
 int foo = 3;
 await Task.Delay(500);
 return foo;
}
```

Сгенерированный компилятором метод-заглушка выглядит следующим образом:

```
public Task<int> AlexsMethod()
{
 <AlexsMethod>d__0 stateMachine = new <AlexsMethod>d__0();
 stateMachine.<>4__this = this;
 stateMachine.<>t__builder = AsyncTaskMethodBuilder<int>.Create();
 stateMachine.<>1__state = -1;
 stateMachine.<>t__builder.Start<<AlexsMethod>d__0>(ref stateMachine);
 return stateMachine.<>t__builder.Task;
}
```

Я немного изменил имена переменных, чтобы было проще понять код.

В разделе «Async, сигнатуры методов и интерфейсы» выше мы видели, что ключевое слово `async` не оказывает влияние на то, как метод используется извне. Это следует из того факта, что сигнатура метода-заглушки совпадает с сигнатурой исходного метода, но без слова `async`.

Обратите внимание, что в заглушке нет и следов моего оригинального кода. Большую часть заглушки составляет инициализация полей структуры с именем `<AlexsMethod>d__0`. Эта структура представляет собой конечный автомат, в котором и производится вся трудная работа. Заглушка вызывает метод `Start`, а затем возвращает объект `Task`. Чтобы понять, что происходит, нам придется заглянуть внутрь самой структуры `stateMachine`.

Структура конечного автомата

Компилятор генерирует структуру, которая играет роль конечного автомата и содержит весь код моего оригинального метода. Делается это для того, чтобы получить объект, способный представить состояние метода, который можно было бы сохранить, когда программа дойдет до `await`. Напомню, что при достижении `await` сохраняется вся информация о том, в каком месте метода находится программа, чтобы при возобновлении ее можно было восстановить.

Компилятор мог бы, конечно, сохранить все локальные переменные метода в момент приостановки, но для этого потребовалось бы сгенерировать очень много кода. Лучше поступить по-другому – преобразовать все локальные переменные вашего метода в переменные-члены некоторого типа, тогда при сохранении экземпляра этого типа автоматически будут сохранены и все локальные переменные. Вот для этого и предназначена сгенерированная структура.

Конечный автомат объявлен как структура, а не класс, из соображений производительности. Благодаря этому при синхронном завершении асинхронного метода не приходится выделять память для объекта из кучи. К сожалению, из-за того, что конечный автомат – структура, рассуждать о нем становится сложнее.

Конечный автомат генерируется в виде структуры, вложенной в тип, содержащий асинхронный метод. Так проще понять, из какого метода он был сгенерирован, но основная причина в том, чтобы предоставить автомату доступ к закрытым членам вашего типа.

Рассмотрим, какая структура конечного автомата (`<AlexsMethod>d__0`) сгенерирована для нашего примера. Сначала – переменные-члены:

```
public int <>1__state;
public int <foo>5__1;
public AlexsClass <>4__this;
public AsyncTaskMethodBuilder<int> <>t__builder;
private object <>t__stack;
private TaskAwaiter <>u__$awaiter2;
```

Имена всех переменных содержат угловые скобки, показывающие, что имена сгенерированы компилятором. Это нужно для того, чтобы сгенерированный компилятором код не вступал в конфликт с пользовательским, – ведь в корректной программе на C# имена переменных не могут содержать угловых скобок. В данном случае это не так важно.

В первой переменной, `<>1__state`, сохраняется номер достигнутого оператора `await`. Пока не встретился никакой `await`, значение этой переменной равно `-1`. Все операторы `await` в оригинальном методе пронумерованы, и в момент приостановки в переменную `state` заносится номер `await`, после которого нужно будет возобновить исполнение.

Следующая переменная, `<foo>5__1`, служит для хранения значения моей оригинальной переменной `foo`. Как мы скоро увидим, все обращения к `foo` заменены обращениями к этой переменной-члену.

Далее следует переменная `<>4__this`. Она встречается только в конечных автоматах для нестатических асинхронных методов и содержит объект, от имени которого этот метод вызывался. В каком-то смысле `this` – это просто еще одна локальная переменная метода, только используется она специальным образом – для доступа к другим членам того же объекта. В процессе преобразования async-метода

ее необходимо сохранить и использовать явно, потому что код оригинального объекта перенесен в структуру конечного автомата.

`AsyncTaskMethodBuilder` — это вспомогательный тип, в котором инкапсулирована логика, общая для всех конечных автоматов. Именно этот тип создает объект `Task`, возвращаемый заглушкой. На самом деле, он очень похож на класс `TaskCompletionSource` в том смысле, что создает задачу-марионетку, которую сможет сделать завершенной позже. Отличие от `TaskCompletionSource` заключается в том, что `AsyncTaskMethodBuilder` оптимизирован для async-методов и ради повышения производительности является структурой, а не классом.

Async-методы, возвращающие `void`, пользуются вспомогательным типом `AsyncVoidMethodBuilder`, а async-методы, возвращающие `Task<T>`, – универсальным вариантом типа, `AsyncTaskMethodBuilder<T>`.

Переменная `<>t__stack` применяется для тех операторов `await`, которые входят в более сложное выражение. Промежуточный язык .NET (IL) является стековым, поэтому сложные выражения строятся из небольших команд, которые манипулируют стеком значений. Если `await` встречается в середине такого сложного выражения, то находящиеся в стеке значения помещаются в эту переменную, причем если значений несколько, то она на самом деле является кортежем `Tuple`.

Наконец, в переменной `TaskAwaiter` хранится временный объект, который помогает оператору `await` подписаться на уведомление о завершении задачи `Task`.

Метод MoveNext

В конечном автомате всегда имеется метод `MoveNext`, в котором находится весь наш оригинальный код. Этот метод вызывается как при первом входе в наш метод, так и при возобновлении после `await`. Даже в случае простейшего async-метода код `MoveNext` на удивление сложен, поэтому я попытаюсь описать преобразование в виде последовательности шагов. Кроме того, я опускаю малосущественные детали, поэтому во многих местах мое описание не вполне точно.

Этот метод назван `MoveNext` из-за сходства с методами `MoveNext`, которые генерировались блоками итераторов в предыдущих версиях C#. Эти блоки позволяют реализовать интерфейс `IEnumerable` в одном методе с помощью ключевого слова `yield return`. Применяемый для этой цели конечный автомат во многом напоминает асинхронный автомат, только проще.

Наш код

Первым делом необходимо скопировать наш код в метод `MoveNext`. Напомним, что все обращения к локальным переменным следует заменить обращениями к переменным-членам конечного автомата. На месте `await` я пока оставлю пропуск, который заполню позже.

```
<foo>5__1 = 3;
Task t = Task.Delay(500);
Здесь будет код, относящийся к await t
return <foo>5__1;
```

Преобразование предложений return в код завершения

Каждое предложение `return` в оригинальном коде следует преобразовать в код, завершающий задачу `Task`, возвращенную методом-заглушкой. На самом деле, метод `MoveNext` возвращает `void`, поэтому предложение `return foo;` вообще недопустимо.

```
<>t__builder.SetResult(<foo>5__1);
return;
```

Разумеется, сделав задачу завершенной, мы выходим из `MoveNext` с помощью `return;`.

Переход в нужное место метода

Поскольку `MoveNext` вызывается как для возобновления после каждого `await`, так и при первом входе в метод, мы должны уметь переходить в нужное место метода. Для этого генерируется примерно такой же IL-код, как для предложения `switch`, как если бы мы ветвились по переменной `state`.

```
switch (<>1__state)
{
 case -1: // При первом входе в метод
   <foo>5__1 = 3;
   Task t = Task.Delay(500);
   Здесь будет код, относящийся к await t
 case 0: // Есть только один await, его номер равен 0
   <>t__builder.SetResult(<foo>5__1);
   return;
}
```

Приостановка метода в месте встречи await

Именно здесь мы используем объект `TaskAwaiter` для подписки на уведомление о завершении задачи `Task`, которую мы ждем. Чтобы возобновиться с нужного места, необходимо изменить переменную `state`. Всё подготовив, мы возвращаем управление, освобождая поток для других дел, как и полагается приличному асинхронному методу.

```
...
  <foo>5__1 = 3;
  <>u__$awaiter2 = Task.Delay(500).GetAwaiter();
  <>1__state = 0;
  <>t__builder.AwaitUnsafeOnCompleted(<>u__$awaiter2, this);
  return;
 case 0:
...
```

В процедуре подписки на уведомление участвует также объект `AsyncTaskMethodBuilder`, и в целом она весьма сложна. Именно здесь реализуются дополнительные возможности `await`, в том числе запоминание контекста синхронизации, который нужно будет восстановить при возобновлении. Но конечный результат понятен. Когда задача `Task` завершится, метод `MoveNext` будет вызван снова.

Возобновление после await

По завершении ожидаемой задачи мы оказываемся в нужном месте метода `MoveNext`, но перед исполнением оригинального кода должны еще получить результат задачи. В данном примере используется неуниверсальный класс `Task`, поэтому в переменную читать нечего. Тем не менее задача могла завершиться ошибкой, и в таком случае мы должны возбудить исключение. Всё это делает метод `GetResult` объекта `TaskAwaiter`.

```
...
 case 0:
  <>u__$awaiter2.GetResult();
  <>t__builder.SetResult(<foo>5__1);
...
```

Синхронное завершение

Напомню, что если `await` используется для ожидания задачи, которая уже завершилась синхронно, то не следует приостанавливать и

возобновлять метод. Для этого мы должны перед возвратом проверить, является ли задача `Task` завершенной. Если да, то мы просто переходим в нужное место с помощью предложения `goto case`.

```
...
<>u__$awaiter2 = Task.Delay(500).GetAwaiter();
if (<>u__$awaiter2.IsCompleted)
{
 goto case 0;
}
<>1__state = 0;
...
```

Сгенерированный компилятором код хорош тем, что его никто не должен сопровождать, поэтому употреблять `goto` можно сколько душе угодно. Раньше я даже не знал о существовании конструкции `goto case`, и, наверное, это к лучшему.

Перехват исключений

Если при исполнении нашего async-метода возникло и не было перехвачено в блоке `try-catch` исключение, то его должен перехватить сгенерированный компилятором код. Это нужно для того, чтобы перевести возвращенный объект `Task` в состояние «ошибка», не позволив исключению покинуть метод. Напомню, что метод `MoveNext` может быть вызван как из того места, где вызывался наш оригинальный async-метод, так и из завершившейся ожидаемой задачи, возможно, через контекст синхронизации. Ни в том, ни в другом случае программа не ожидает исключений.

```
try
{
 ... Весь метод
}
catch (Exception e)
{
 <>t__builder.SetException(<>t__ex);
 return;
}
```

Более сложный код

Мой пример был очень прост. Метод `MoveNext` становится гораздо сложнее при наличии в оригинальном коде следующих конструкций:

- блоков `try-catch-finally`;
- ветвлений (`if` и `switch`);
- циклов;
- операторов `await` в середине выражения.

Компилятор корректно преобразует все эти конструкции, поэтому программисту не приходится задумываться о сложности кода.

Призываю вас воспользоваться декомпилятором и посмотреть, как выглядит метод `MoveNext` для какого-нибудь из ваших собственных async-методов. Попробуйте найти места, которые я упростил в своем описании, и понять, как преобразуется более сложный код.

Разработка типов, допускающих ожидание

Тип `Task` допускает ожидание, то есть к нему можно применить оператор `await`. В разделе «Интерфейсы IAsyncAction и IAsyncOperation<T>» мы видели, что есть и другие допускающие ожидание типы, например тип WinRT `IAsyncAction`. На самом деле, можно и самостоятельно написать такого рода типы, хотя вряд ли в этом возникнет необходимость.

Чтобы тип допускал ожидание, он должен предоставлять средства, используемые в рассмотренном выше методе `MoveNext`. Прежде всего, в нем должен быть определен метод `GetAwaiter`:

```
class MyAwaitableClass
{
 public AlexsAwaiter GetAwaiter()
 {
...
```

`GetAwaiter` может быть и методом расширения, что дает дополнительную гибкость. Например, в интерфейсе `IAsyncAction` нет метода `GetAwaiter`, потому что он входит в WinRT, а в WinRT нет понятия типа, допускающего ожидание. `IAsyncAction` наделяется такой возможностью только благодаря методу расширения `GetAwaiter`, предоставляемому .NET.

Далее, тип, возвращаемый методом `GetAwaiter`, должен обладает определенными свойствами, чтобы класс `MyAwaitableClass` мог считаться допускающим ожидание. Минимальные требования таковы:

- он должен реализовывать интерфейс `INotifyCompletion`, то есть содержать метод `void OnCompleted(Action handler)`, который подписывается на уведомление о завершении;

- он должен содержать свойство `bool IsCompleted { get; }`, которое служит для проверки синхронного завершения;
- он должен содержать метод `T GetResult()`, который возвращает результат операции и возбуждает исключения.

Тип `T`, возвращаемый методом `GetResult`, может быть `void`, как то имеет место в случае `Task`. Но может быть и настоящим типом, как в случае `Task<T>`. И лишь во втором случае компилятор позволит использовать `await` как выражение, например в правой части оператора присваивания.

Вот как мог бы выглядеть тип `AlexsAwaiter`:

```
class AlexsAwaiter : INotifyCompletion
{
 public bool IsCompleted
 {
   get
   {
     ...
   }
 }

 public void OnCompleted(Action continuation)
 {
   ...
 }

 public void GetResult()
 {
   ...
 }
}
```

Важно помнить о существовании класса `TaskCompletionSource` и о том, что обычно гораздо лучше воспользоваться им, когда требуется преобразовать нечто асинхронное в нечто допускающее ожидание. В классе `Task` есть немало полезных функций, и пренебречь ими было бы непростительным легкомыслием.

Взаимодействие с отладчиком

Возможно, вы думаете, что после того как компилятор так перелопатил ваш код, встроенный в Visual Studio отладчик не сможет разобраться в том, что происходит. Но на самом деле с отладкой всё в порядке. Достигается это в основном за счет того, что компилятор связывает строки написанного вами исходного кода с соответствую-

щими им частями сгенерированного метода `MoveNext`. Это соответствие хранится в PDB-файле и гарантирует правильную работу следующих функций отладчика:

- расстановка точек прерывания;
- пошаговое исполнение строк, не содержащих `await`;
- просмотр строки, в которой было возбуждено исключение.

Однако если внимательнее присмотреться к коду, остановившись в точке прерывания после оператора `await` в async-методе, то можно заметить некоторые признаки, свидетельствующие о том, что компилятор действительно преобразовал код.

- В нескольких местах имя текущего метода отображается как `MoveNext`. В окне трассировки стека оно заменяется именем оригинального метода, но Intellitrace этого не делает.
- В окне трассировки стека видны кадры, отражающие инфраструктуру TPL, за которыми следует строка *[Resuming Async Method]* и имя вашего метода.

Но настоящее волшебство творится в режиме пошагового исполнения кода. Отладчик Visual Studio предпринимает героические усилия, чтобы корректно перешагнуть (команда Step Over – F10) через оператор `await`, несмотря на то, что метод продолжается через неопределенное время в заранее неизвестном потоке. Следы необходимой для этого инфраструктуры можно наблюдать в классе `AsyncTaskMethodBuilder`, где имеется свойство `ObjectIdForDebugger`. Отладчик умеет также выходить из async-метода (команда Step Out – Shift+F11), оказываясь при этом в точке после `await`, где ждет завершения задачи.

ГЛАВА 15. Производительность асинхронного кода

Решая воспользоваться асинхронным кодом, вы, вероятно, задумывались о производительности. Каким бы ни был побудительный мотив – отзывчивость пользовательского интерфейса, пропускная способность сервера или распараллеливание с помощью акторов – необходима уверенность, что изменение действительно оправдает себя.

Рассуждая о производительности асинхронного кода, следует сравнить его с имеющимися в конкретной ситуации альтернативами. В этой главе мы будем рассматривать:

- ситуации, когда имеется длительная операция, которую потенциально можно выполнить асинхронно;
- ситуации, когда не существует длительной операции и возможностей асинхронного исполнения не видно;
- сравнение асинхронного кода с обычным, блокирующим программу на время выполнения длительной операции;
- сравнение механизма async с асинхронным кодом, написанным вручную.

Мы также обсудим некоторые оптимизации, полезные в случае, когда оказывается, что сопряженные с механизмом async накладные расходы приводят к проблемам с производительностью приложения.

Измерение накладных расходов механизма async

На реализацию механизма async неизбежно затрачивается дополнительное по сравнению с синхронным кодом время, а переключения

между потоками увеличивают задержку. Невозможно точно измерить накладные расходы на реализацию асинхронности. Производительность приложения зависит от того, чем занимаются потоки, от поведения кэша и от других непредсказуемых факторов. Кроме того, есть различие между использованием процессора и дополнительной задержкой, поскольку время операции в асинхронной системе может возрастать и без потребления ЦП – из-за того, что запрос ожидает своей очереди. Поэтому мой анализ дает лишь порядок величины. В качестве эталона для сравнения я возьму стоимость обычного вызова метода. На моем ноутбуке за одну секунду эталонный метод можно вызвать примерно 100 миллионов раз.

Async и блокирующая длительная операция

Обычно к механизму async прибегают, когда имеется длительная операция, которую можно выполнить асинхронно, освободив тем самым ресурсы. В программах с пользовательским интерфейсом асинхронность позволяет обеспечить отзывчивость интерфейса (если, конечно, операция не выполняется мгновенно). В серверном коде компромисс не столь очевиден, так как мы должны выбирать между памятью, занятой заблокированными потоками, и дополнительным процессорным временем, затрачиваемым на выполнение асинхронных методов.

Накладные расходы на действительно асинхронное выполнение async-метода всецело зависят от того, необходимо ли переключение потоков с помощью метода `SynchronizationContext.Post`. Если это так, то подавляющую долю накладных расходов составляет именно переключение потоков в момент возобновления метода. Это означает, что текущий контекст синхронизации играет очень важную роль. Я замерял накладные расходы, выполняя метод, который не делает ничего, кроме `await Task.Yield`, то есть всегда завершается асинхронно:

```
async Task AlexsMethod()
{
 await Task.Yield();
}
```

Таблица 15.1. Накладные расходы на исполнение и возобновление async-метода

SynchronizationContext	Стоимость (по сравнению с пустым методом)
`Post` не нужен	100
Пул потоков	100
Windows forms	1000
WPF	1000
ASP.NET	1000

Придется ли платить за переключение потоков, зависит как от контекста синхронизации вызывающего потока, так и от контекста синхронизации потока, в котором завершилась задача.

- Если эти потоки совпадают, то вызывать метод `Post` исходного контекста `SynchronizationContext` не нужно, и метод можно возобновить в потоке, где завершилась задача, синхронно – как часть процедуры завершения.
- Если в вызывающем потоке был контекст синхронизации, но не тот, что в потоке, где произошло завершение, то требуется вызвать метод `Post`, что приведет к высоким накладным расходам, отраженным в таблице. То же самое имеет место, когда в потоке завершения нет контекста синхронизации.
- Если в вызывающем потоке не было контекста синхронизации, как, например, в консольном приложении, то ситуация определяется контекстом синхронизации потока завершения. Если он существует, то .NET предполагает, что этот поток важен и планирует возобновление метода в потоке из пула. Если же контекста синхронизации в потоке завершения нет или это поток, взятый из пула, то метод возобновляется в том же потоке, синхронно.

В действительности пул потоков в .NET работает настолько быстро, что накладные расходы на переключение потоков даже не отразились на порядке величины сравнительно с возобновлением метода в том же потоке. Таким образом, о контексте синхронизации потока завершения можно вообще не думать.

Эти правила означают, что цепочка async-методов приведет к одному дорогостоящему переключению потоков – при возобновлении

метода с наибольшим уровнем вложенности. После этого контекст синхронизации уже не меняется, и возобновление остальных методов обходится дешево. Переключение потоков в контексте пользовательского интерфейса оказывается одной из самых дорогих операций. Однако в приложении с пользовательским интерфейсом синхронное выполнение длительных операций выглядит настолько безобразно, что выбора всё равно нет. Если сетевой запрос занимает 500 мс, то имеет смысл пожертвовать еще одну миллисекунду на обеспечение отзывчивости интерфейса.

К сожалению, WPF часто пересоздает объекты `SynchronizationContext`, поэтому в контексте WPF метод `Post` вызывается при возобновлении каждого вложенного async-метода. Приложения для Windows Forms и Windows 8 этой болезнью не страдают.

В серверном коде, например в приложениях ASP.NET, выбор компромисса требует более тщательного анализа. Имеет ли смысл переходить на асинхронный код, зависит прежде всего от того, хватает ли серверу оперативной памяти, потому что именно памятью приходится расплачиваться за использование большого числа потоков. Есть целый ряд факторов, из-за которых синхронное приложение может потреблять память быстрее, чем процессорное время.

- Вызываются длительные операции, занимающие относительно много времени.
- Длительные операции распараллеливаются за счет использования дополнительных потоков.
- Есть много запросов, которые требуют запуска длительных операций и не могут быть обслужены из кэша в памяти.
- Для порождения ответа не требуется много процессорного времени.

Единственный способ уяснить, как обстоит дело, – замерять потребление памяти сервером. Если это действительно проблема и память выделяется чрезмерно большому количеству потоков, то переход к асинхронному выполнению может оказаться неплохим решением. При этом будет потребляться немного больше процессорного времени, но если серверу не хватает памяти, а процессорных мощностей в избытке, то с этим легко смириться.

Напомню, что хотя async-методы всегда потребляют больше процессорного времени, чем синхронные, разница на самом деле совсем невелика и может оказаться незаметна на фоне других задач, решаемых приложением.

Оптимизация асинхронного кода для длительной операции

Если async-метод действительно работает асинхронно, то, как мы видели, львиная доля накладных расходов приходится на обращение к методу `Post` вызывающего контекста `SynchronizationContext`, в результате которого происходит переключение потоков. В разделе «Когда не следует использовать SynchronizationContext» мы говорили, что метод `ConfigureAwait` позволяет подавить вызов `Post` и тем самым не платить за переключение потоков, когда без него можно обойтись. Если ваш код вызывается из потока пользовательского интерфейса в WPF, то такое решение позволит избежать повторных вызовов `Post`.

Источником накладных расходов при написании async-методов служит также контекст исполнения вызывающего потока – `ExecutionContext`. В разделе «Контекст» мы видели, что .NET запоминает и восстанавливает `ExecutionContext` при каждом `await`. Если вы не используете `ExecutionContext`, то запоминание и восстановление контекста по умолчанию хорошо оптимизировано и обходится очень дешево. В противном случае процедура становится куда более дорогой. Поэтому для повышения производительности старайтесь не пользоваться контекстами `CallContext`, `LogicalCallContext` или олицетворением.

Async-методы и написанный вручную асинхронный код

В старой программе с пользовательским интерфейсом проблема отзывчивости, вероятно, уже решена за счет какой-то ручной асинхронной техники. Способов несколько, в том числе:

- создание нового потока;
- использование метода `ThreadPool.QueueUserWorkItem` для выполнения длительной операции в фоновом потоке;
- использование класса `BackgroundWorker`;
- ручное использование асинхронного API.

При любом из этих подходов необходим хотя бы один возврат в поток пользовательского интерфейса для представления результата пользователю, то есть то же самое, что async-метод делает автоматически. Иногда это осуществляется неявно (например, с помощью

события `RunWorkerCompleted` в классе `BackgroundWorker`), а иногда требуется явно вызывать метод `BeginInvoke`.

По скорости все эти подходы различаются незначительно, исключение составляет создание нового потока, которое выполняется гораздо медленнее. Механизм async как минимум не уступает в скорости любому из перечисленных решений, если удается избежать использования `ExecutionContext`. На самом деле, в моих измерениях async оказывался даже на несколько процентов быстрее. И так как наличествует небольшой выигрыш в скорости, а код выглядит понятнее, то я предпочитаю async любой другой технике.

Async и блокирование без длительной операции

Весьма типична ситуация, когда некоторый метод иногда выполняется долго, но в 99 процентах случаев работает очень быстро. Примером может служить кэшируемый сетевой запрос, когда большинство запросов обслуживаются из кэша. Использовать ли в таких случаях асинхронный код, часто зависит от накладных расходов в типичном случае, когда метод завершается синхронно, а не от расходов в одном проценте случаев, когда действительно требуется асинхронная сетевая операция.

Напомню, что ключевое слово `await` не приостанавливает метод без необходимости, когда задача уже завершена. Метод, содержащий `await`, в этом случае также завершается синхронно и возвращает уже завершенный объект `Task`. Следовательно, вся цепочка async-методов отрабатывает синхронно.

Async-методы, даже когда они исполняются синхронно, неизбежно оказываются медленнее эквивалентных синхронных методов. И в данном случае мы не получаем никакого выигрыша от освобождения ресурсов. Так называемый async-метод не является асинхронным, а так называемый блокирующий метод ничего не блокирует. Тем не менее преимущества, которые асинхронность дает в одном проценте случаев, когда запрос нельзя обслужить из кэша, могут быть настолько велики, что написание асинхронного кода оправдано.

Всё зависит от того, насколько асинхронный код медленнее обычного, когда тот и другой синхронно возвращают результат из кэша.

Но точное измерение затруднительно, так как зависит от слишком многих факторов. Я обнаружил, что вызов пустого async-метода в 10 раз медленнее вызова пустого синхронного метода.

Да, асинхронный код медленнее, но напомню, что это лишь накладные расходы. Почти всегда они поглощаются реальной работой. Например, поиск в словаре `Dictionary<string, string>` также обходится примерно в 10 раз медленнее, чем вызов пустого метода.

Оптимизация асинхронного кода без длительной операции

Накладные расходы на вызов синхронно завершающегося async-метода, которые примерно в 10 раз превышают стоимость вызова пустого синхронного метода, проистекают из нескольких источников. Большая их часть неизбежна – например, исполнение сгенерированного компилятором кода, вызовы каркаса и невозможность различных оптимизаций из-за способа обработки исключений, возникающих в async-методах.

Из тех расходов, которых можно избежать, основная часть приходится на выделение памяти для объектов из кучи. Собственно выделение памяти – очень дешевая операция. Но когда таких объектов много, приходится чаще запускать сборщик мусора, а, объект, все еще используемый во время сборки мусора, обходится дорого.

Механизм async спроектирован так, чтобы память из кучи выделялась как можно реже. Именно поэтому конечной автомат является структурой, равно как и типы `AsyncTaskMethodBuilder`. Они перемещаются в кучу, только если async-метод приостанавливается.

Но `Task` – не структура, поэтому всегда выделяется из кучи. По этой причине в .NET заранее выделено несколько объектов `Task`, которые используются, когда async-метод завершается синхронно и возвращает одно из следующих типичных значений:

- неуниверсальный, успешно завершенный объект `Task`;
- объект типа `Task<bool>`, содержащий `true` или `false`;
- объект типа `Task<int>`, содержащий небольшое целое число;
- объект типа `Task<T>`, содержащий `null`.

Если вы разрабатываете кэш, который должен обладать очень высокой производительностью, и ни один из этих случаев неприменим, то избежать выделения памяти из кучи можно путем кэширования завершенного объекта `Task`, а не просто значения. Впрочем, это редко бывает оправдано, поскольку вы, скорее всего, всё равно выделяете память для объектов из кучи в других местах программы.

В заключение отметим, что async-методы, завершающиеся синхронно, уже работают очень быстро и дальнейшая оптимизация зат-

руднительна. Тратить силы на кэширование объектов `Task` имеет смысл, только если приложение работает не так быстро, как вам хотелось бы, а причиной является именно сборка мусора.

Резюме

Хотя async-код всегда потребляет больше процессорного времени, чем эквивалентный синхронный код, разница обычно мала по сравнению со временем выполнения операции, которую вы хотите сделать асинхронной. В серверном коде накладные расходы следует соизмерять с памятью, отводимой под дополнительные потоки. В программах с пользовательским интерфейсом и при распараллеливании с помощью акторов async-код быстрее и чище, чем реализация асинхронных паттернов вручную, поэтому лучше отдать ему предпочтение.

Наконец, если операция обычно завершается немедленно, то использование async-кода не принесет никакого вреда, так как он лишь чуть медленнее эквивалентного кода, написанного без использования async.

Книги издательства «ДМК Пресс» можно заказать в торгово-издательском холдинге «АЛЬЯНС БУКС» наложенным платежом, выслав открытку или письмо по почтовому адресу: **123242, Москва, а/я 20** или по электронному адресу: **orders@alians-kniga.ru**.

При оформлении заказа следует указать адрес (полностью), по которому должны быть высланы книги; фамилию, имя и отчество получателя. Желательно также указать свой телефон и электронный адрес.

Эти книги вы можете заказать и в Интернет-магазине: **www.alians-kniga.ru**

Оптовые закупки: тел. **(499) 725-54-09, 725-50-27**.

Электронный адрес: **books@alians-kniga.ru**

Алекс Дэвис

Асинхронное программирование в C# 5.0

Главный редактор *Мовчан Д. А.*
dm@dmk-press.ru
Перевод с английского *Слинкин А. А.*
Корректор *Синяева Г. И.*
Верстка *Паранская Н. В.*
Дизайн обложки *Мовчан А. Г.*

Подписано в печать 10.11.2012. Формат 60×90 $^{1}/_{16}$.
Гарнитура «Петербург». Печать офсетная.
Усл. печ. л. 7,35. Тираж 500 экз.
заказ №

Веб-сайт издательства: www.dmk-press.ru

Асинхронное программирование в C# 5.0

Из этого краткого руководства вы узнаете, как механизм async в C# 5.0 позволяет упростить написание асинхронного кода. Помимо ясного введения в асинхронное программирование вообще, вы найдете углубленное описание работы этого конкретного механизма и ответ на вопрос, когда и зачем использовать его в собственных приложениях.

Книга рассчитана на опытных программистов на C#, но будет понятна и начинающим. Она изобилует примерами кода, который можно использовать в своих программах.

Рассматриваются следующие вопросы:

- Как писать асинхронный код вручную и как механизм async скрывает неприглядные детали;
- Новые способы повышения производительности серверного кода в приложениях ASP.NET;
- Совместная работа async и WinRT в приложениях для Windows 8;
- Смысл ключевого слова await в async-методах;
- В каком потоке .NET исполняется асинхронный код в каждой точке программы;
- Написание асинхронных API, согласованных с паттерном Task-based Asynchronous Pattern (TAP);
- Распараллеливание программ для задействования возможностей современных компьютеров;
- Измерение производительности async-кода и сравнение с альтернативными подходами.

Интернет-магазин:
www.dmk-press.ru
Книга – почтой:
orders@alians-kniga.ru
Оптовая продажа:
“Альянс-книга”
тел.(499)725-54-09
books@alians-kniga.ru

O’REILLY®

ДМК
ИЗДАТЕЛЬСТВО
www.дмк.рф

ISBN 978-5-94074-886-1
9 785940 748861 >