Асинхронное программирование в C# 5.0

Алекс Дэвис

ДМК Издательство

O'REILLY®

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

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».