

# Асинхронные функции в C#

<b>Введение.....</b>	<b>2</b>
<b>Ожидание.....</b>	<b>2</b>
Захват локального состояния.....	5
Ожидание в пользовательском интерфейсе.....	6
Сравнение с крупномодульным параллелизмом.....	10
<b>Написание асинхронных функций.....</b>	<b>11</b>
Возврат Task<TResult>.....	13
Выполнение графа асинхронных вызовов.....	15
Параллелизм.....	16
<b>Асинхронные лямбда-выражения.....</b>	<b>17</b>
<b>Асинхронные потоки данных.....</b>	<b>19</b>
Запрашивание IEnumerable<T>.....	21
IEnumerable<T> в ASP.NET Core.....	22
<b>Асинхронные методы в WinRT.....</b>	<b>22</b>
<b>Асинхронность и контексты синхронизации.....</b>	<b>23</b>
Отправка исключений.....	23
OperationStarted и OperationCompleted.....	25
<b>Оптимизация.....</b>	<b>26</b>
Синхронное завершение.....	26
ValueTask<T>.....	29
Меры предосторожности при использовании ValueTask<T>.....	30
Избегание чрезмерных возвратов.....	31

## Введение

Ключевые слова `async` и `await` позволяют писать асинхронный код, который обладает той же самой структурой и простотой, что и синхронный код, а также устранять необходимость во вспомогательном коде, присущем асинхронному программированию.

## Ожидание

Ключевое слово `await` упрощает присоединение признаков продолжения. Рассмотрим базовый сценарий. Приведенные ниже строки:

```
var результат = await выражение;  
оператор(ы);
```

компилятор развернет в следующий функциональный эквивалент:

```
var awaiter = выражение.GetAwaiter();  
awaiter.OnCompleted(() =>  
{  
    var результат = awaiter.GetResult();  
    оператор(ы);  
});
```

❖ Компилятор также выпускает код для замыкания продолжения в случае синхронного завершения (см. пункт “Оптимизация” далее) и для обработки разнообразных нюансов, которые мы затронем в последующих разделах.

В целях демонстрации вернемся к ранее написанному асинхронному методу, который вычисляет и подсчитывает простые числа:

```
Task<int> GetPrimesCountAsync(int start, int count)  
{  
    return Task.Run(() =>  
        ParallelEnumerable.Range(start, count).Count(n =>  
            Enumerable.Range(2, (int)Math.Sqrt(n) - 1)  
                .All(i => n % i > 0)));  
}
```

Используя ключевое слово *await*, его можно вызвать следующим образом:

```
int result = await GetPrimesCountAsync(2, 1000000);  
Console.WriteLine(result);
```

Чтобы код скомпилировался, к содержащему такой вызов методу понадобится добавить модификатор *async*:

```
async void DisplayPrimesCount()  
{  
    int result = await GetPrimesCountAsync(2, 1000000);  
    Console.WriteLine(result);  
}
```

Модификатор *async* сообщает компилятору о необходимости трактовать *await* как ключевое слово, а не идентификатор, что привело бы к неоднозначности внутри данного метода (это гарантирует успешную компиляцию кода, написанного до выхода версии C# 5, где слово *await* использовалось в качестве идентификатора). Модификатор *async* может применяться только к методам (и лямбда-выражениям), которые возвращают *void* либо (как позже будет показано) тип *Task* или *Task<TResult>*.

❖ Модификатор *async* подобен модификатору *unsafe* в том, что он не дает никакого эффекта на сигнатуре или открытых метаданных метода, а воздействует, только когда находится *внутри* метода. По этой причине не имеет смысла использовать *async* в интерфейсе. Однако вполне законно, например, вводить *async* при переопределении виртуального метода, не являющегося асинхронным, при условии сохранения сигнатуры метода в неизменном виде.

Методы с модификатором *async* называются *асинхронными функциями*, т.к. сами они обычно асинхронны. Чтобы увидеть почему, давайте посмотрим, каким образом процесс выполнения проходит через асинхронную функцию.

Встретив выражение *await*, процесс выполнения (обычно) производит возврат в вызывающий код – почти как оператор *yield return* в итераторе. Но перед возвратом исполняющая среда присоединяет к ожидающей задаче признак продолжения, который гарантирует, что когда задача завершится, управление перейдет обратно в метод и продолжит с места, где оно его оставило. Если задача отказывает, тогда ее исключение генерируется повторно, а в противном случае выражению *await* присваивается возвращаемое значение задачи. Все сказанное можно резюмировать, просмотрев логическое расширение только что рассмотренного асинхронного метода:

```
void DisplayPrimesCount()
{
    var awaiter = GetPrimesCountAsync(2, 1000000).GetAwaiter();
    awaiter.OnCompleted(() =>
    {
        int result = awaiter.GetResult();
        Console.WriteLine(result);
    });
}
```

Выражение, к которому применяется *await*, обычно является задачей; тем не менее, компилятор устроит любой объект с методом *GetAwaiter*, который возвращает объект ожидания (реализующий метод *INotifyCompletion.OnCompleted* и имеющий должным образом типизированный метод *GetResult* и свойство *bool IsCompleted*).

Обратите внимание, что выражение *await* оценивается как относящееся к типу *int*; причина в том, что ожидаемым выражением было *Task<int>* (метод *GetAwaiter().GetResult* которого возвращает тип *int*).

Ожидание необобщенной задачи вполне законно и генерирует выражение *void*:

```
await Task.Delay(5000);
Console.WriteLine("Five seconds passed!");
```

## Захват локального состояния

Реальная мощь выражений *await* заключается в том, что они могут находиться практически в любом месте кода. В частности, выражение *await* может присутствовать на месте любого выражения (внутри асинхронной функции) кроме выражения *lock* или контекста *unsafe*.

В следующем примере *await* располагается внутри цикла:

```
async void DisplayPrimeCounts()
{
    for (int i = 0; i < 10; i++)
        Console.WriteLine(await GetPrimesCountAsync(i * 1000000 + 2,
1000000));
}
```

При первом выполнении метода *GetPrimesCountAsync* управление возвращается вызывающему коду из-за выражения *await*. Когда метод завершается (или отказывает), выполнение возобновляется с места, в котором оно его покинуло, с сохраненными значениями локальных переменных и счетчиков циклов.

В возобновлении выполнения после выражения *await* компилятор полагается на признаки продолжения (согласно шаблону объектов ожидания). Это значит, что в случае запуска в потоке пользовательского интерфейса контекст синхронизации гарантирует, что выполнение будет возобновлено в том же самом потоке. В противном случае выполнение возобновляется в любом потоке, где задача была завершена. Смена потока не оказывает влияния на порядок выполнения и несущественна, если только вы каким-то образом не зависите от родства потоков, возможно, из-за использования локального хранилища потока. Здесь уместна аналогия с ситуацией, когда вы ловите такси, чтобы добраться из одного места в другое. При наличии контекста синхронизации в вашем распоряжении всегда будет один и тот же таксомотор, а без контекста синхронизации таксомоторы каждый раз, возможно,

окажутся разными. Хотя путешествие в любом случае будет в основном тем же самым.

## Ожидание в пользовательском интерфейсе

Мы можем продемонстрировать асинхронные функции в более практичном контексте, реализовав простой пользовательский интерфейс, который остается отзывчивым во время вызова метода с интенсивными вычислениями. Давайте начнем с синхронного решения:

```
class TestUI: Window
{
    Button _button = new Button { Content = "Go" };
    TextBlock _results = new TextBlock();
    public TestUI()
    {
        var panel = new StackPanel();
        panel.Children.Add(_button);
        panel.Children.Add(_results);
        Content = panel;
        _button.Click += (sender, args) => Go();
    }
    void Go()
    {
        for (int i = 1; i < 5; i++)
            _results.Text += GetPrimesCount(i * 1000000, 1000000) +
                " primes between " + (i * 1000000) + " and " + ((i + 1) *
1000000 - 1) +
                Environment.NewLine;
    }
    int GetPrimesCount(int start, int count)
    {
        return ParallelEnumerable.Range(start, count).Count(n =>
            Enumerable.Range(2, (int)Math.Sqrt(n) - 1)
                .All(i => n % i > 0));
    }
}
```

После щелчка на кнопке Go (Запуск) приложение перестает быть отзывчивым на время, необходимое для выполнения кода с интенсивными вычислениями. Решение превращается в асинхронное за два шага. Первый шаг связан с переключением на асинхронную версию метода *GetPrimesCount*, который применялся в предыдущем примере:

```
Task<int> GetPrimesCountAsync(int start, int count)
{
    return Task.Run(() =>
        ParallelEnumerable.Range(start, count).Count(n =>
            Enumerable.Range(2, (int)Math.Sqrt(n) - 1).All(i => n % i >
0))) );
}
```

Второй шаг предусматривает изменение метода Go для вызова метода *GetPrimesCountAsync*:

```
async void Go()
{
    _button.IsEnabled = false;
    for (int i = 1; i < 5; i++)
        _results.Text += await GetPrimesCountAsync(i * 1000000,
1000000) +
            " primes between " + (i * 1000000) + " and " + ((i + 1) *
1000000 - 1) +
                Environment.NewLine;
    _button.IsEnabled = true;
}
```

Приведенный выше код демонстрирует простоту программирования с использованием асинхронных функций: все делается как при синхронном подходе, но вместо блокирования функций и их ожидания посредством *await* производится вызов асинхронных функций. В рабочем потоке запускается только код внутри метода *GetPrimesCountAsync*; код в методе Go “арендует” время у потока пользовательского интерфейса. Можно было бы сказать, что метод Go выполняется *псевдопараллельно* циклу сообщений (т.е. его выполнение пересекается с другими событиями, которые обрабатывает поток пользовательского интерфейса). Благодаря такому псевдопараллелизму единственной точкой, где может возникнуть вытеснение, является выполнение *await*. В итоге обеспечение безопасности к потокам упрощается: в данном случае может возникнуть только одна проблема – *реентерабельность* (из-за повторного щелчка на кнопке во время выполнения метода Go, чего мы избегаем, делая кнопку недоступной). Подлинный параллелизм происходит ниже в

стеке вызовов, внутри кода, вызываемого методом *Task.Run*. Чтобы извлечь преимущества из такой модели, по-настоящему параллельный код избегает доступа к разделяемому состоянию или элементам управления пользовательского интерфейса.

Рассмотрим еще один пример, в котором вместо вычисления простых чисел загружается несколько веб-страниц с суммированием их длин. В *.NET* доступно множество асинхронных методов, возвращающих задачи, один из которых определен в классе *WebClient* внутри пространства имен *System.Net*. Метод *DownloadDataTaskAsync* асинхронно загружает URI в байтовый массив, возвращая объект *Task<byte[]>*, так что в результате ожидания можно получить массив *byte[]*. Давайте перепишем метод *Go*:

```
async void Go()
{
    _button.IsEnabled = false;
    string[] urls = "www.albahari.com www.oreilly.com
www.lingpad.net".Split();
    int totalLength = 0;
    try
    {
        foreach (string url in urls)
        {
            var uri = new Uri("http://" + url);
            byte[] data = await new WebClient().DownloadDataTaskAsync(uri);
            _results.Text += "Length of " + url + " is " + data.Length +
Environment.NewLine;    // длина загруженных данных
            totalLength += data.Length;
        }
        _results.Text += "Total length: " + totalLength; // общая длина
    }
    catch (WebException ex)
    {
        _results.Text += "Error: " + ex.Message; // ошибка
    }
    finally { _button.IsEnabled = true; }
}
```

И снова код отражает то, как бы мы реализовали его синхронным образом, включая применение блоков *catch* и *finally*. Хотя после первого *await* управление возвращается вызывающему коду, блок *finally* не выполняется вплоть до логического завершения метода (после выполнения всего его кода либо по причине раннего



возвращения из-за оператора *return* или возникновения необработанного исключения).

Полезно посмотреть, что в точности происходит. Для начала необходимо вернуться к псевдокоду, который выполняет цикл сообщений в потоке пользовательского интерфейса:

```
Установить для этого потока контекст синхронизации WPF
while (приложение не завершено)
{
    Ожидать появления чего-нибудь в очереди сообщений
    Что-то получено: к какому виду сообщений оно относится?
    - Сообщение клавиатуры / мыши->запустить обработчик событий
    - Пользовательское сообщение (BeginInvoke / Invoke)->выполнить делегат
}
```

Обработчики событий, присоединяемые к элементам пользовательского интерфейса, выполняются через такой цикл сообщений. Когда запускается наш метод *Go*, выполнение продолжается до выражения *await*, после чего управление возвращается в цикл сообщений (освобождая пользовательский интерфейс для реагирования на дальнейшие события). Однако расширение компилятором выражения *await* гарантирует, что перед возвращением продолжение настроено так, чтобы выполнение возобновлялось там, где оно было прекращено до завершения задачи. И поскольку ожидание с помощью *await* происходит в потоке пользовательского интерфейса, то признак продолжения отправляется контексту синхронизации, который выполняет его через цикл сообщений, сохраняя выполнение всего метода *Go* псевдопараллельным с потоком пользовательского интерфейса. Подлинный параллелизм (с интенсивным вводом-выводом) происходит внутри реализации метода *DownloadDataTaskAsync*.

## Сравнение с крупномодульным параллелизмом

До выхода версии C# 5 асинхронное программирование было затруднено не только из-за отсутствия языковой поддержки, но и потому, что асинхронная функциональность в .NET Framework открывалась через неуклюжие шаблоны EAP и APM, а не через методы, возвращающие объекты задач.

Популярным обходным путем являлся крупномодульный параллелизм (для чего даже был предусмотрен тип по имени *BackgroundWorker*). Мы можем продемонстрировать крупномодульную асинхронность на исходном *синхронном* примере с методом *GetPrimesCount*, изменив обработчик событий кнопки, как показано ниже:

```
button.Click += (sender, args) =>
{
    _button.IsEnabled = false;
    Task.Run(() => Go());
};
```

(Мы решили использовать метод *Task.Run* вместо класса *BackgroundWorker* из-за того, что *BackgroundWorker* никак бы не упростил данный конкретный пример.) В любом случае конечный результат состоит в том, что целый граф синхронных вызовов (*Go* и *GetPrimesCount*) выполняется в рабочем потоке. И поскольку метод *Go* обновляет элементы пользовательского интерфейса, в код придется добавить вызовы *Dispatcher.BeginInvoke*:

```
void Go()
{
    for (int i = 1; i < 5; i++)
    {
        int result = GetPrimesCount(i * 1000000, 1000000);
        Dispatcher.BeginInvoke(new Action(() =>
            _results.Text += result + " primes between " + (i * 1000000) + "
and " + ((i + 1) * 1000000 - 1) + Environment.NewLine));
    }
    Dispatcher.BeginInvoke(new Action(() => _button.IsEnabled = true));
}
```

В отличие от асинхронной версии цикл сам выполняется в рабочем потоке. Это может казаться безобидным, но даже в таком простом случае применение многопоточности привело к возникновению условия состязаний. (Смогли его заметить? Если нет, тогда запустите программу: условие состязаний почти наверняка станет очевидным.) Реализация отмены и сообщения о ходе работ создает больше возможностей для ошибок, связанных с нарушением безопасности к потокам, как делает любой дополнительный код в методе. Например, предположим, что верхний предел для цикла не закодирован жестко, а поступает из вызова метода:

```
for (int i = 1; i < GetUpperBound(); i++)
```

Далее представим, что метод *GetUpperBound* читает значение из конфигурационного файла, который ленивым образом загружается с диска при первом вызове. Весь этот код теперь выполняется в рабочем потоке – код, который весьма вероятно не является безопасным к потокам. В том и заключается опасность за пуска рабочих потоков на высоких уровнях внутри графа вызовов.

## Написание асинхронных функций

Что касается любой асинхронной функции, то возвращаемый тип *void* можно заменить типом *Task*, чтобы сделать сам метод *пригодным* для асинхронного выполнения (и ожидания с помощью *await*). Никаких других изменений вносить не придется:

```
async Task PrintAnswerToLife() // Можно возвращать Task вместо
{
    await Task.Delay(5000);
    int answer = 21 * 2;
    Console.WriteLine(answer);
}
```

Обратите внимание, что в теле метода мы не возвращаем объект задачи явным образом. Компилятор произведет задачу, которая будет отправлять сигнал о за вершении данного метода (или о

возникновении необработанного исключения). В итоге упрощается создание цепочек асинхронных вызовов:

```
async Task Go()
{
    await PrintAnswerToLife();
    Console.WriteLine("Done");
}
```

Так как метод `Go` объявлен с возвращаемым типом `Task`, сам `Go` допускает ожидание посредством `await`.

Компилятор расширяет асинхронные функции, возвращающие задачи, в код, использующий класс `TaskCompletionSource` для создания задачи, которая затем отправляет сигнал о завершении или отказе.

Оставив в стороне нюансы, мы можем развернуть метод `PrintAnswerToLife` в такой функциональный эквивалент:

```
Task PrintAnswerToLife()
{
    var tcs = new TaskCompletionSource<object>();
    var awaiter = Task.Delay(5000).GetAwaiter();
    awaiter.OnCompleted(() =>
    {
        try
        {
            awaiter.GetResult(); // Сгенерировать повторно любые
            // исключения
            int answer = 21 * 2;
            Console.WriteLine(answer);
            tcs.SetResult(null);
        }
        catch (Exception ex) { tcs.SetException(ex); }
    });
    return tcs.Task;
}
```

Следовательно, всякий раз, когда возвращающий задачу асинхронный метод завершается, управление переходит обратно к месту его ожидания (благодаря признаку продолжения).

❖ В сценарии с обогащенным клиентом управление перемещается с этой точки обратно в поток пользовательского интерфейса (если оно еще в нем не находится). В противном случае выполнение продолжается в любом потоке, куда был направлен признак продолжения, что означает отсутствие задержки при подъеме по графу асинхронных вызовов кроме первого “прыжка”, если он был инициирован потоком пользовательского интерфейса.

## Возврат *Task<TResult>*

Возвращать объект *Task<Tresult>* можно, если в теле метода возвращается тип *TResult*:

```
async Task<int> GetAnswerToLife()
{
    await Task.Delay(5000);
    int answer = 21 * 2;
    return answer; // Метод имеет возвращаемый тип Task<int>,
                  // поэтому вернуть int
}
```

Внутренне это приводит к тому, что объекту *TaskCompletionSource* отправляется сигнал со значением, отличающимся от *null*. Мы можем продемонстрировать работу метода *GetAnswerToLife*, вызвав его из метода *PrintAnswerToLife* (который в свою очередь вызывается из *Go*):

```
async Task Go()
{
    await PrintAnswerToLife();
    Console.WriteLine("Done");
}
async Task PrintAnswerToLife()
{
    int answer = await GetAnswerToLife();
    Console.WriteLine(answer);
}
async Task<int> GetAnswerToLife()
{
    await Task.Delay(5000);
    int answer = 21 * 2;
    return answer;
}
```

В сущности, мы преобразовали исходный метод *PrintAnswerToLife* в два метода – с той же легкостью, как если бы программировали синхронным образом. Сходство с синхронным программированием является умышленным; вот синхронный эквивалент нашего графа вызовов, для которого вызов метода *Go* дает тот же самый результат после блокирования на протяжении пяти секунд:

```
void Go()
{
    PrintAnswerToLife();
    Console.WriteLine("Done");
}
void PrintAnswerToLife()
{
    int answer = GetAnswerToLife();
    Console.WriteLine(answer);
}
int GetAnswerToLife()
{
    Thread.Sleep(5000);
    int answer = 21 * 2;
    return answer;
}
```

❖ Тем самым также иллюстрируется базовый принцип проектирования с применением асинхронных функций в C#.

1. Напишите синхронные версии своих методов.
2. Замените вызовы *синхронных* методов вызовами *асинхронных* методов и примените к ним *await*.
3. За исключением методов “верхнего уровня” (обычно обработчиков событий для элементов управления пользовательского интерфейса) поменяйте возвращаемые типы асинхронных методов на *Task* или *Task<TResult>*, чтобы они поддерживали *await*.

Способность компилятора производить задачи для асинхронных функций означает, что явно создавать объект *TaskCompletionSource* придется главным образом только в (относительно редком) случае методов нижнего уровня, которые иницируют параллелизм с интенсивным вводом-выводом. (Для методов, иницирующих параллелизм с интенсивными вычислениями, создается задача с помощью метода *Task.Run*.)

## Выполнение графа асинхронных вызовов

Чтобы ясно увидеть, как все выполняется, полезно реорганизовать код следующим образом:

```
async Task Go()
{
    var task = PrintAnswerToLife();
    await task; Console.WriteLine("Done");
}
async Task PrintAnswerToLife()
{
    var task = GetAnswerToLife();
    int answer = await task; Console.WriteLine(answer);
}
async Task<int> GetAnswerToLife()
{
    var task = Task.Delay(5000);
    await task; int answer = 21 * 2; return answer;
}
```

Метод *Go* вызывает метод *PrintAnswerToLife*, который вызывает метод *GetAnswerToLife*, а тот в свою очередь вызывает метод *Delay* и затем ожидает. Наличие *await* приводит к тому, что управление возвращается методу *PrintAnswerToLife*, который сам ожидает, возвращая управление методу *Go*, который также ожидает и возвращает управление вызывающему коду. Все происходит синхронно в потоке, вызвавшем метод *Go*; это короткая *синхронная* фаза выполнения.

Спустя пять секунд запускается продолжение на *Delay* и управление возвращается методу *GetAnswerToLife* в потоке из пула. (Если мы начинали в потоке пользовательского интерфейса, то управление возвратится в него.) Затем выполняются оставшиеся операторы в методе *GetAnswerToLife*, после чего задача *Task<int>* данного метода завершается с результатом 42 и иницируется продолжение в методе *PrintAnswerToLife*, что приведет к выполнению оставшихся операторов в этом методе. Процесс продолжается до тех пор, пока задача метода *Go* не выдаст сигнал о своем завершении.

Поток выполнения соответствует показанному ранее графу синхронных вызовов, т.к. мы следуем шаблону, при котором к каждому асинхронному методу сразу после вызова применяется `await`. В результате создается последовательный поток без параллелизма или перекрывающегося выполнения внутри графа вызовов. Каждое выражение `await` образует “брешь” в выполнении, после которой программа возобновляет работу с того места, где она ее оставила.

## Параллелизм

Вызов асинхронного метода без его ожидания позволяет коду, который за ним следует, выполняться параллельно. В приведенных ранее примерах вы могли отметить наличие кнопки, обработчик события которой вызывал метод `Go` так, как показано ниже:

```
_button.Click += (sender, args) => Go();
```

Несмотря на то что `Go` является асинхронным методом, мы не можем применить к нему `await`, и это действительно то, что содействует параллелизму, необходимому для поддержки отзывчивого пользовательского интерфейса.

Тот же самый принцип можно использовать для запуска двух асинхронных операций параллельно:

```
var task1 = PrintAnswerToLife();  
var task2 = PrintAnswerToLife();  
await task1; await task2;
```

(За счет ожидания обеих операций впоследствии мы “заканчиваем” параллелизм в данной точке. Позже мы покажем, как комбинатор задач `WhenAll` помогает в реализации такого шаблона.)



Параллелизм, организованный подобным образом, происходит независимо от того, инициированы ли операции в потоке пользовательского интерфейса, хотя существует отличие в том, как он проявляется. В обоих случаях мы получаем тот же самый “подлинный” параллелизм в операциях нижнего уровня, которые его иницируют (таких как *Task.Delay* или код, предоставленный методу *Task.Run*). Методы, находящиеся выше в стеке вызовов, будут по-настоящему параллельными, только если операция была иницирована без наличия контекста синхронизации; иначе они окажутся псевдопараллельными (и упростят обеспечение безопасности к потокам), согласно чему единственным местом, где может произойти вытеснение, является оператор *await*. Это позволяет, например, определить разделяемое поле *\_x* и инкрементировать его в методе *GetAnswerToLife* без блокирования:

```
async Task<int> GetAnswerToLife()
{
    _x++;
    await Task.Delay(5000);
    return 21 * 2;
}
```

(Тем не менее, мы не можем предполагать, что *\_x* имеет одно и то же значение до и после *await*.)

## Асинхронные лямбда-выражения

Точно так же как обычные *именованные* методы могут быть асинхронными:

```
async Task NamedMethod()
{
    await Task.Delay(1000);
    Console.WriteLine("Foo");
}
```

асинхронными способны быть и *неименованные* методы (лямбда-выражения и анонимные методы), если они предварены ключевым словом *async*:

```
Func<Task> unnamed = async () =>
{
    await Task.Delay(1000);
    Console.WriteLine("Foo");
};
```

Вызывать и применять *await* к ним можно одинаково:

```
await NamedMethod();
await unnamed();
```

Асинхронные лямбда-выражения можно использовать при подключении обработчиков событий :

```
myButton.Click += async (sender, args) =>
{
    await Task.Delay(1000);
    myButton.Content = "Done";
};
```

Это более лаконично, чем приведенный ниже код, который обеспечивает тот же самый результат:

```
myButton.Click += ButtonHandler;
...
async void ButtonHandler(object sender, EventArgs args)
{
    await Task.Delay(1000);
    myButton.Content = "Done";
}
```

Асинхронные лямбда-выражения могут также возвращать тип `Task<TResult>`:

```
Func<Task<int>> unnamed = async () =>
{
    await Task.Delay(1000);
    return 123;
};
int answer = await unnamed();
```

## Асинхронные потоки данных

С помощью *yield return* вы можете писать итератор, а с помощью *await* – асинхронную функцию. Асинхронные потоки (появившиеся в C# 8) объединяют эти концепции и позволяют реализовать итератор, который ожидает, выдавая элементы асинхронным образом. Такая поддержка основана на следующей паре интерфейсов, которые представляют собой асинхронные аналоги интерфейсов перечисления:

```
public interface IAsyncEnumerable<out T>
{
    IAsyncEnumerator<T> GetAsyncEnumerator(...);
}
public interface IAsyncEnumerator<out T> : IAsyncDisposable
{
    T Current { get; }
    ValueTask<bool> MoveNextAsync();
}
```

`ValueTask<T>` – это структура, которая является оболочкой для `Task<T>` и по поведению похожа на `Task<T>`, но обеспечивает более эффективное выполнение, когда задача завершается синхронно (что часто может происходить при перечислении последовательности). Обсуждение отличий ищите в разделе "`ValueTask<T>`" далее в главе. Интерфейс `IAsyncDisposable` представляет собой асинхронную версию `IDisposable`; он предоставляет возможность выполнения очистки, если вы решите вручную реализовывать интерфейсы:

```
public interface IAsyncDisposable
{
    ValueTask DisposeAsync();
}
```

❖ Действие по извлечению каждого элемента из последовательности (*MoveNextAsync*) является асинхронной операцией, поэтому асинхронные потоки подходят, когда элементы поступают постепенно (например, как при обработке данных из видеопотока). Напротив, следующий тип лучше подходит, когда задерживается последовательность как единое целое, но когда элементы поступают, то поступают все вместе: *Task<IEnumerable<T>>*

Чтобы сгенерировать асинхронный поток, необходимо написать метод, в котором объединены принципы итераторов и асинхронных методов. Другими словами, метод должен включать и *yield return*, и *await*, а также возвращать *IAsyncEnumerable<T>*:

```
async IAsyncEnumerable<int> RangeAsync(
    int start, int count, int delay)
{
    for (int i = start; i < start + count; i++)
    {
        await Task.Delay(delay);
        yield return i;
    }
}
```

Для применения асинхронного потока нужно использовать оператор *await foreach*:

```
await foreach (var number in RangeAsync(0, 10, 500))
    Console.WriteLine(number);
```

Обратите внимание, что данные поступают постоянно каждые 500 миллисекунд (или в реальности, когда они становятся доступными). Сравните это с похожей конструкцией, использующей *Task<IEnumerable<T>>*, для которой данные не возвращаются до тех пор, пока не будет доступной последняя порция данных:

```
static async Task<IEnumerable<int>> RangeTaskAsync(int start, int
count, int delay)
{
    List<int> data = new List<int>();
    for (int i = start; i < start + count; i++)
    {
        await Task.Delay(delay);
        data.Add(i);
    }
    return data;
}
```

А вот как потреблять асинхронный поток посредством оператора *foreach*:

```
foreach (var data in await RangeTaskAsync(0, 10, 500))
    Console.WriteLine(data);
```

## Запрашивание *IAsyncEnumerable<T>*

В NuGet-пакете *System.Linq.Async* определены операции LINQ, работаю щие с *IAsyncEnumerable<T>*, которые позволяют писать запросы во многом подобно тому, как делалось бы с *IEnumerable<T>*.

Например, мы можем написать запрос LINQ для метода *RangeAsync*, определенного в предыдущем разделе, следующим образом:

```
IAsyncEnumerable<int> query =
    from i in RangeAsync(0, 10, 500)
    where i % 2 == 0 // Только четные числа.
    select i * 10;    // Умножить на 10.

await foreach (var number in query)
    Console.WriteLine(number);
```

В результате выводятся числа 0, 20, 40 и т.д.

❖ Если вы знакомы с библиотекой Reactive Extensions, то можете также извлечь преимущества из ее (более мощных) операций запросов, вызывая расширяющий метод `ToObservable`, который преобразует реализацию `IAsyncEnumerable<T>` в `IObservable<T>`. Кроме того, доступен расширяющий метод `ToAsyncEnumerable`, предназначенный для преобразования в обратном направлении.

## ***IAsyncEnumerable<T>* в ASP.NET Core**

Действия контроллера ASP.NET Core теперь могут возвращать *IAsyncEnumerable<T>*. Такие методы должны быть помечены как `async`. Например:

```
[HttpGet]
public async IAsyncEnumerable<string> Get()
{
    using var dbContext = new BookContext();
    await foreach (var title in dbContext.Books
        .Select(b => b.Title)
        .AsAsyncEnumerable())
        yield return title;
}
```

## **Асинхронные методы в WinRT**

В случае разработки приложений UWP вам придется иметь дело с типами WinRT, определенными в ОС. Эквивалентом `Task` в WinRT является тип `IAsyncAction`, а эквивалентом `Task<TResult>` – тип `IAsyncOperation<TResult>`.

Для операций, которые сообщают о ходе работ, эквивалентами будут `IAsyncActionWithProgress<TProgress>` и `IAsyncOperationWithProgress<TResult, TProgress>`. Все они определены в пространстве имени `Windows.Foundation`.

Выполнять преобразование в тип `Task` или `Task<TResult>` либо из него можно с помощью расширяющего метода `AsTask`:

```
Task<StorageFile> fileTask =  
KnownFolders.DocumentsLibrary.CreateFileAsync  
("test.txt").AsTask();
```

Или же можно реализовать ожидание напрямую:

```
StorageFile file = await KnownFolders.DocumentsLibrary.CreateFileAsync  
("test.txt");
```

❖ Из-за ограничений системы типов COM интерфейсы *IAsyncActionWithProgress<TProgress>* и *IAsyncOperationWithProgress<TResult, TProgress>* не основаны на *IAsyncAction*, как можно было бы ожидать. Взамен оба интерфейса унаследованы от общего базового типа по имени *IAsyncInfo*.

Метод *AsTask* также перегружен для приема признака отмены. Он также принимает объект *IProgress<T>*, когда соединяется в цепочку с вариантами *WithProgress*.

## Асинхронность и контексты синхронизации

Ранее уже было показано, что наличие контекста синхронизации играет важную роль в смысле отправки признаков продолжения. В случае асинхронных функций, возвращающих *void*, существует пара других, более тонких способов взаимодействия с контекстами синхронизации. Они являются не прямым результатом расширений, производимых компилятором C#, а функцией типов *Async\*MethodBuilder* из пространства имен *System.CompilerServices*, которое компилятор использует при расширении асинхронных функций.

### Отправка исключений

В обогащенных клиентских приложениях общепринято полагаться на событие централизованной обработки исключений (*Application.DispatcherUnhandledException* в WPF) для учета необработанных исключений, сгенерированных в потоке

пользовательского интерфейса. В приложениях ASP.NET Core похожую работу делает специальный фильтр *ExceptionHandlerAttribute* в методе *ConfigureServices* из *Startup.cs*. Внутренне они функционируют, иницируя события пользовательского интерфейса (или конвейера методов обработки страниц в случае ASP.NET Core) в собственном блоке *try/catch*.

Асинхронные функции верхнего уровня затрудняют это. Рассмотрим следующий обработчик события щелчка на кнопке:

```
async void ButtonClick(object sender, RoutedEventArgs args)
{
    await Task.Delay(1000);
    throw new Exception("Will this be ignored?"); //Будет ли это
//проигнорировано?
}
```

Когда на кнопке осуществляется щелчок и обработчик событий запускается, после оператора *await* управление обычно возвращается в цикл сообщений, и сгенерированное секунду спустя исключение не может быть перехвачено блоком *catch* в цикле сообщений.

Чтобы устранить проблему, структура *AsyncVoidMethodBuilder* перехватывает необработанные исключения (в асинхронных функциях, возвращающих *void*) и отправляет их контексту синхронизации, если он присутствует, гарантируя то, что события глобальной обработки исключений по-прежнему иницируются.

❖ Компилятор применяет упомянутую логику только к асинхронным функциям, возвращающим *void*. Следовательно, если изменить *ButtonClick* для возвращения типа *Task* вместо *void*, тогда необработанное исключение приведет к отказу результирующей задачи, поскольку ему некуда больше двигаться (в результате давая *необнаруженное* исключение).

Интересный нюанс связан с тем, что нет никакой разницы, где генерируется исключение – до или после *await*. Таким образом, в



следующем примере исключение отправляется контексту синхронизации (если он существует), но не вызывающему коду:

```
async void Foo() { throw null; await Task.Delay(1000); }
```

(При отсутствии контекста синхронизации исключение будет распространяться в пуле потоков и приведет к завершению работы приложения.)

Причина, по которой исключение не возвращается обратно вызывающему компоненту, связана с обеспечением предсказуемости и согласованности. В следующем примере исключение *InvalidOperationException* всегда будет иметь один и тот же эффект отказа результирующей задачи независимо от того, как выглядит какое-то-условие:

```
async Task Foo()
{
    if (какое-то-условие) await Task.Delay(100);
    throw new InvalidOperationException();
}
```

Итераторы работают аналогично:

```
IEnumerable<int> Foo() { throw null; yield return 123; }
```

В приведенном примере исключение никогда не возвращается прямо вызывающему коду: с исключением имеет дело только перечисляемая последовательность.

### ***OperationStarted* и *OperationCompleted***

Если контекст синхронизации присутствует, то асинхронные функции, возвращающие *void*, также вызывают его метод *OperationStarted* при входе в функцию и метод *OperationCompleted*, когда функция завершается.

Переопределение таких методов удобно при написании специального контекста синхронизации для проведения модульного тестирования асинхронных методов, возвращающих *void*. Данная тема обсуждается в блоге, посвященном параллельному программированию в .NET (<https://devblogs.microsoft.com/pfxteam/>).

## Оптимизация

### Синхронное завершение

Возврат из асинхронной функции может произойти *перед* организацией ожидания. Рассмотрим следующий метод, который обеспечивает кэширование в процессе загрузки веб-страниц:

```
static Dictionary<string, string> _cache = new
Dictionary<string, string>();
async Task<string> GetWebPageAsync(string uri)
{
    string html;
    if (_cache.TryGetValue(uri, out html)) return html;
    return _cache[uri] =
        await new WebClient().DownloadStringTaskAsync(uri);
}
```

Если URI присутствует в кеше, тогда управление возвращается вызывающему коду безо всякого ожидания, и метод возвращает объект задачи, которая уже *сигнализирована*. Это называется синхронным завершением.

Когда производится ожидание синхронно завершенной задачи, управление не возвращается вызывающему коду и не переходит обратно через признак продолжения – взамен выполнение продолжается со следующего оператора. Компилятор реализует такую оптимизацию, проверяя свойство *IsCompleted* объекта ожидания; другими словами, всякий раз, когда производится ожидание:

```
Console.WriteLine (await GetWebPageAsync ("http://oreilly.com"));
```

Компилятор выпускает код для короткого замыкания продолжения в случае синхронного завершения:

```
var awaiter = GetWebPageAsync().GetAwaiter();  
if (awaiter.IsCompleted)  
    Console.WriteLine(awaiter.GetResult());  
else  
    awaiter.OnCompleted(() =>  
        Console.WriteLine(awaiter.GetResult()));
```

❖ Ожидание асинхронной функции, которая завершается синхронно, все равно связано с (крайне) небольшими накладными расходами – примерно 20 нс. на современных компьютерах. Напротив, переход в другие потоки вызывает переключение контекста – возможно одну или две микросекунды, а переход в цикл обработки сообщений пользовательского интерфейса – минимум в десять раз больше (и ещё больше, если пользовательский интерфейс занят).

Вполне законно даже писать асинхронные методы, для которых *никогда* не производится ожидание, хотя компилятор сгенерирует предупреждение:

```
async Task<string> Foo() { return "abc"; }
```

Такие методы могут быть полезны при переопределении виртуальных/абстрактных методов, если случится так, что ваша реализация не потребует асинхронности. (Примером могут служить методы *ReadAsync/WriteAsync* класса *MemoryStream*.) Другой способ достичь того же результата предусматривает применение метода *Task.FromResult*, который возвращает уже сигнализированную задачу:

```
Task<string> Foo() { return Task.FromResult ("abc"); }
```

Если наш метод *GetWebPageAsync* вызывается из потока пользовательского интерфейса, то он является неявно безопасным

к потокам в смысле том, что его можно было бы вызвать несколько раз подряд (инициируя тем самым множество параллельных загрузок) без необходимости в каком-либо блокировании с целью защиты кэша. Однако если бы последовательность обращений относилась к одному и тому же URI, то инициировалось бы множество избыточных загрузок, которые все в конечном итоге обновляли бы одну и ту же запись в кеше (в выигрыше окажется последняя загрузка). Хотя это и не ошибка, но более эффективно было бы сделать так, чтобы последующие обращения к тому же самому URI взамен (асинхронно) ожидали результата выполняющегося запроса.

Существует простой способ достичь указанной цели, не прибегая к блокировкам или сигнализирующим конструкциям. Вместо кеша строк мы создаем кеш “будущего” (*Task<string>*):

```
static readonly ConcurrentDictionary<string, Task<string>> _cache
= new();
Task<string> GetWebPageAsync(string uri)
{
    if (_cache.TryGetValue(uri, out var downloadTask)) return
downloadTask;
    return _cache[uri] = new
WebClient().DownloadStringTaskAsync(uri);
}
```

(Обратите внимание, что мы не помечаем метод как *async*, поскольку напрямую возвращаем объект задачи, полученный в результате вызова метода класса *WebClient*.)

Теперь при повторяющихся вызовах метода *GetWebPageAsync* с тем же самым URI мы гарантируем получение одного и того же объекта *Task<string>*. (Это обеспечивает и дополнительное преимущество минимизации нагрузки на сборщик мусора.) И если задача завершена, то ожидание не требует больших затрат благодаря описанной выше оптимизации, которую предпринимает компилятор.

Мы могли бы и дальше расширять пример, чтобы сделать его безопасным к потокам без защиты со стороны контекста синхронизации, для чего необходимо блокировать все тело метода:

```
lock (_cache)
    if (_cache.TryGetValue(uri, out var downloadTask))
        return downloadTask;
    else
        return _cache[uri] = new WebClient().DownloadStringTaskAsync(uri);
```

Код работает из-за того, что мы производим блокировку не на время загрузки страницы (это нанесло бы ущерб параллелизму), а на небольшой промежуток времени, пока проверяется кэш и при необходимости запускается новая задача, которая обновляет кэш.

## ValueTask<T>

❖ Тип `ValueTask<T>` предназначен для сценариев микро-оптимизации и вам, возможно, никогда не придется писать методы, которые возвращают этот тип. Тем не менее, все равно нужно знать об изложенных в следующем разделе мерах предосторожности, поскольку некоторые методы .NET возвращают `ValueTask<T>`, а `IAsyncEnumerable<T>` тоже его использует.

Мы только что описали, каким образом компилятор оптимизирует выражение `await` для синхронно завершаемой задачи – путем замыкания накоротко продолжения и немедленного перехода к следующему оператору. Если синхронное завершение происходит из-за кеширования, то мы выяснили, что кеширование самой задачи может дать элегантное и эффективное решение.

Однако кешировать задачу во всех сценариях синхронного завершения нецелесообразно. Иногда должна создаваться новая задача, что порождает (крошечную) потенциальную неэффективность. Дело в том, что `Task` и `Task<T>` являются ссылочными типами, а потому создание их экземпляров требует выделения памяти в куче и последующей сборки мусора. Экстремальная форма оптимизации предусматривает написание кода без выделения памяти; другими словами, код не создает

экземпляры каких-либо ссылочных типов, не увеличивая нагрузку на сборщик мусора. Для поддержки такого шаблона были введены структуры `ValueTask` и `ValueTask<T>`, которые компилятор разрешает помещать вместо `Task` и `Task<T>`:

```
async ValueTask<int> Foo() { ... }
```

Ожидание `ValueTask<T>` свободно от выделения памяти, если операция завершается синхронно:

```
int answer = await Foo(); // (Потенциально) свободно от выделения памяти
```

Если операция не завершается синхронно, тогда `ValueTask<T>` создает "за кулисами" обыкновенный экземпляр `Task<T>` (которому передает ожидание) и никакого преимущества не достигается.

Экземпляр `ValueTask<T>` можно преобразовать в обыкновенный экземпляр `Task<T>`, вызвав метод `AsTask`.

Кроме того, существует необобщенная версия – `ValueTask`, которая похожа на `Task`.

## Меры предосторожности при использовании `ValueTask<T>`

Тип `ValueTask<T>` относительно необычен в том, что он определен как структура исключительно по соображениям производительности. Таким образом, он обременен *неподходящей* семантикой типа значения, что может приводить к неожиданностям. Чтобы избежать некорректного поведения, вы должны не допускать следующих действий :

- организовывать ожидание одного и того же экземпляра `ValueTask<T>` много раз;
- вызывать `.GetAwaiter().GetResult()`, когда операция не завершена.

Если вам необходимо выполнить указанные действия, тогда вызовите `.AsTask ()` и работайте с результирующим экземпляром `Task`.

❖ Избежать описанных ловушек проще всего, применяя `await` напрямую к вызову метода, например:

```
await Foo(); // Безопасно
```

Дверь к ошибочному поведению открывается, когда вы присваиваете значение задачи `ValueTask` какой-то переменной:

```
ValueTask valueTask = Foo(); //осторожно  
// Использование переменной valueTask теперь может приводить к ошибкам
```

что можно смягчить, преобразуя непосредственно в обыкновенную задачу:

```
Task task = Foo().AsTask(); // Безопасно  
// С переменной task работать безопасно
```

## Избегание чрезмерных возвратов

В методах, которые многократно вызываются в цикле, можно избежать накладных расходов, связанных с повторяющимся возвратом в цикл сообщений пользовательского интерфейса, за счет вызова метода `ConfigureAwait`. В результате задача не передает признаки продолжения контексту синхронизации, сокращая накладные расходы до затрат на переключение контекста (или намного меньших, если метод, для которого осуществляется ожидание, завершается синхронно):

```
async void A() { ... await B(); ... }  
async Task B()  
{  
    for (int i = 0; i < 1000; i++)  
        await C().ConfigureAwait(false);  
}  
async Task C() { ... }
```

Это означает, что для методов `B` и `C` мы аннулируем простую модель безопасности к потокам в приложениях с пользовательским интерфейсом, согласно которой код выполняется в потоке пользовательского интерфейса и может

быть вытеснен только во время выполнения оператора *await*. Однако метод *A* не затрагивается и останется в потоке пользовательского интерфейса, если он в нем был запущен.

Такая оптимизация особенно уместна при написании библиотек: преимущество упрощенной безопасности потоков не требуется, потому что код библиотеки обычно не разделяет состояние с вызывающим кодом и не обращается к элементам управления пользовательского интерфейса. (В приведенном выше примере также имеет смысл реализовать синхронное завершение метода *C*, если известно, что операция, скорее всего, окажется кратковременной.)