# Сценарии асинхронного программирования

Статья • 13.10.2023

Для решения задач, связанных с вводом-выводом (например, запрос данных из сети, доступ к базе данных или чтение и запись в файловой системе), желательно использовать асинхронное программирование. Если у вас есть код, ограниченный ресурсами процессора, например выполняющий сложные вычисления, то это также подходящий сценарий для асинхронного программирования.

В C# есть модель асинхронного программирования, реализованная на уровне языка, которая позволяет легко писать асинхронный код, не прибегая к обратным вызовам или библиотекам, которые поддерживают асинхронность. Она строится на принципах асинхронной модели на основе задач (TAP).

# Обзор асинхронной модели

В основе асинхронного программирования лежат объекты `Task` и `Task<T>`, которые моделируют асинхронные операции. Они поддерживаются ключевыми словами `async` и `await`. В большинстве случаев модель достаточно проста.

- В коде, ограниченном производительностью ввода-вывода, выполняйте await для операции, которая возвращает `Task` или `Task<T>`, внутри метода `async`.
- В коде, ограниченном ресурсами процессора, выполняйте await для операции, которая запускается в фоновом потоке методом Task.Run.

Именно с помощью ключевого слова `await` творится вся магия. Оно передает управление вызывающему объекту метода, который выполнил `await`, позволяя, таким образом, пользовательскому интерфейсу или службе отвечать на запросы. Хотя существуют и другие способы реализации асинхронного кода, кроме `async` и `await`, в этой статье рассматриваются только конструкции уровня языка.

> **Примечание**
>
> В некоторых из следующих примеров класс используется для скачивания некоторых System.Net.Http.HttpClient данных из веб-службы. Объект, `s_httpClient` используемый в этих примерах, является статическим полем `Program` класса (проверьте полный пример):
>
> ```
> private static readonly HttpClient s_httpClient = new();
> ```

## Пример привязки ввода-вывода: скачивание данных из веб-службы

Предположим, вам нужно скачать некоторые данные из веб-службы по нажатию кнопки, не блокируя поток пользовательского интерфейса. Это можно сделать так:

C#

```
s_downloadButton.Clicked += async (o, e) =>
{
    // This line will yield control to the UI as the request
    // from the web service is happening.
    //
    // The UI thread is now free to perform other work.
    var stringData = await s_httpClient.GetStringAsync(URL);
    DoSomethingWithData(stringData);
};
```

В этом коде намерение (скачивание данных в асинхронном режиме) выражается без запутанных операций с объектами `Task`.

## Пример, привязанный к ЦП: выполнение вычисления для игры

Предположим, вы разрабатываете игру для мобильных устройств, в которой при нажатии кнопки может наноситься урон множеству противников на экране. Расчет урона может потреблять много ресурсов. Если производить его в потоке пользовательского интерфейса, то на это время игра может приостанавливаться!

Оптимальный способ — запустить фоновый поток, который выполняет задачу с помощью `Task.Run`, а затем ожидать ее результат с помощью `await`. Это обеспечит плавность работы пользовательского интерфейса в процессе вычисления.

C#

```
static DamageResult CalculateDamageDone()
{
    return new DamageResult()
    {
        // Code omitted:
        //
        // Does an expensive calculation and returns
        // the result of that calculation.
    };
}

s_calculateButton.Clicked += async (o, e) =>
{
    // This line will yield control to the UI while CalculateDamageDone()
    // performs its work. The UI thread is free to perform other work.
    var damageResult = await Task.Run(() => CalculateDamageDone());
    DisplayDamage(damageResult);
};
```

В этом коде четко выражается назначение события нажатия кнопки, фоновым потоком не требуется управлять вручную, и он выполняется без блокировки.

### Что происходит на внутреннем уровне

С точки зрения C#, компилятор преобразовывает код в конечный автомат, который контролирует такие моменты, как передача выполнения при достижении `await` и возобновление выполнения после завершения фонового задания.

Если вас интересует теория, это реализация модели асинхронности на основе обещаний.

## Ключевые моменты для понимания

- Асинхронный код можно использовать как при ограниченной производительности ввода-вывода, так и при ограниченных ресурсах процессора, но по-разному в каждом случае.
- В асинхронном коде используются конструкции `Task<T>` и `Task`, которые служат для моделирования задач, выполняемых в фоновом режиме.
- Ключевое слово `async` делает метод асинхронным, что позволяет использовать в его теле ключевое слово `await`.
- Когда применяется ключевое слово `await`, оно приостанавливает выполнение вызывающего метода и передает управление обратно вызывающему объекту, пока не будет завершена ожидаемая задача.
- `await` можно использовать только внутри асинхронного метода.

## Различия задач, ограниченных ресурсами процессора и производительностью ввода-вывода

В первых двух примерах этого руководства было показано, как можно использовать `async` и `await` для выполнения задач, ограниченных производительностью ввода-вывода и ресурсами процессора. Крайне важно уметь идентифицировать такие задачи, так как они могут существенно повлиять на производительность кода и привести к неправильному использованию некоторых конструкций.

Перед написанием любого кода нужно ответить на два вопроса.

1. Будет ли код "ожидать" чего-либо, например данных из базы данных?

   Если ответ утвердительный, то ваша задача **ограничена производительностью ввода-вывода**.

2. Будет ли код выполнять сложные вычисления?

   Если ответ утвердительный, то задача **ограничена ресурсами процессора**.

Если ваша задача **ограничена производительностью ввода-вывода**, используйте `async` и `await` *без* `Task.Run`. Библиотеку параллельных задач использовать *не следует*.

Если ваша задача **ограничена ресурсами процессора** и вам важна скорость реагирования, используйте `async` и `await`, но перенесите выполнение задачи в дополнительный поток, у которого *есть* `Task.Run`. Если к задаче применим параллелизм, рассмотрите возможность использования библиотеки параллельных задач.

Кроме того, всегда следует оценивать выполнение кода. Например, затраты на выполнение задачи, ограниченной ресурсами процессора, могут оказаться не столь высокими, как накладные расходы, связанные с переключениями контекста при многопоточности. Каждый вариант имеет свои недостатки, поэтому следует выбрать наиболее компромиссный вариант в вашей ситуации.

# Дополнительные примеры

В приведенных ниже примерах демонстрируются различные способы написания асинхронного кода на C#. Они охватывают несколько сценариев, с которыми вы можете столкнуться.

## Извлечение данных из сети

Этот фрагмент скачивает HTML-код из заданного URL-адреса и подсчитывает количество случаев, когда строка .NET возникает в HTML. С помощью ASP.NET он определяет метод контроллера веб-API, который выполняет эту задачу и возвращает число.

> ⓘ **Примечание**
>
> Если вы планируете проанализировать HTML в рабочем коде, не используйте регулярные выражения. Используйте библиотеку анализа.

C#

```
[HttpGet, Route("DotNetCount")]
static public async Task<int> GetDotNetCount(string URL)
{
    // Suspends GetDotNetCount() to allow the caller (the web server)
    // to accept another request, rather than blocking on this one.
    var html = await s_httpClient.GetStringAsync(URL);
    return Regex.Matches(html, @"\.NET").Count;
}
```

Вот аналогичный код для универсального приложения для Windows, который выполняет ту же задачу при нажатии кнопки:

C#

```
private readonly HttpClient _httpClient = new HttpClient();

private async void OnSeeTheDotNetsButtonClick(object sender, RoutedEventArgs e)
{
    // Capture the task handle here so we can await the background task later.
    var getDotNetFoundationHtmlTask = _httpClient.GetStringAsync("https://dotnetfoundation.org");

    // Any other work on the UI thread can be done here, such as enabling a Progress Bar.
    // This is important to do here, before the "await" call, so that the user
    // sees the progress bar before execution of this method is yielded.
    NetworkProgressBar.IsEnabled = true;
    NetworkProgressBar.Visibility = Visibility.Visible;

    // The await operator suspends OnSeeTheDotNetsButtonClick(), returning control to its caller.
    // This is what allows the app to be responsive and not block the UI thread.
    var html = await getDotNetFoundationHtmlTask;
    int count = Regex.Matches(html, @"\.NET").Count;
```

```
    DotNetCountLabel.Text = $"Number of .NETs on dotnetfoundation.org: {count}";

    NetworkProgressBar.IsEnabled = false;
    NetworkProgressBar.Visibility = Visibility.Collapsed;
}
```

## Ожидание выполнения нескольких задач

Может возникнуть ситуация, когда несколько фрагментов данных должны извлекаться одновременно. API-интерфейс `Task` содержит два метода, Task.WhenAll и Task.WhenAny, которые позволяют писать асинхронный код, выполняющий неблокирующее ожидание нескольких фоновых заданий.

В этом примере показано, как можно получить данные `User` для набора `userId`.

C#

```
private static async Task<User> GetUserAsync(int userId)
{
    // Code omitted:
    //
    // Given a user Id {userId}, retrieves a User object corresponding
    // to the entry in the database with {userId} as its Id.

    return await Task.FromResult(new User() { id = userId });
}

private static async Task<IEnumerable<User>> GetUsersAsync(IEnumerable<int> userIds)
{
    var getUserTasks = new List<Task<User>>();
    foreach (int userId in userIds)
    {
        getUserTasks.Add(GetUserAsync(userId));
    }

    return await Task.WhenAll(getUserTasks);
}
```

Вот более лаконичный вариант этого кода, написанный с использованием LINQ:

C#

```
private static async Task<User[]> GetUsersAsyncByLINQ(IEnumerable<int> userIds)
{
    var getUserTasks = userIds.Select(id => GetUserAsync(id)).ToArray();
    return await Task.WhenAll(getUserTasks);
}
```

Хотя размер кода будет меньше, соблюдайте осторожность при использовании LINQ в сочетании с асинхронным кодом. Так как в LINQ используется отложенное выполнение, асинхронные вызовы будут выполняться не немедленно, как в цикле `foreach`, если только вы не производите принудительную итерацию созданной последовательности с помощью вызова `.ToList()` или `.ToArray()`. Приведенный выше пример использует Enumerable.ToArray для выполнения запроса с нетерпением и хранения результатов в массиве. Это заставляет код `id => GetUserAsync(id)` запускать и запускать задачу.

# Важные сведения и советы

При создании асинхронного кода необходимо учитывать ряд моментов, которые позволят избежать непредвиденного поведения.

- **В методах `async` должно присутствовать ключевое слово `await`. В противном случае результат не будет получен.**

  Это важно помнить. Если в теле метода `async` не используется ключевое слово `await`, компилятор C# выдаст предупреждение, но код скомпилируется и будет выполняться, как обычный метод. Это крайне неэффективно, так как созданный компилятором C# конечный автомат для асинхронного метода не будет выполнять никакой работы.

- **К имени каждого создаваемого асинхронного метода следует добавлять суффикс Async.**

Это соглашение применяется в .NET для удобной дифференциации синхронных и асинхронных методов. Это не всегда применимо к некоторым методам, которые не вызываются в коде явным образом (например, к обработчикам событий или методам веб-контроллеров). Так как они не вызываются в коде явно, требования к их именованию не так строги.

- **`async void` следует использовать только для обработчиков событий.**

  `async void` — это единственный способ обеспечить работу асинхронных обработчиков событий, так как у событий нет типов возвращаемых значений (поэтому они не могут использовать `Task` и `Task<T>`). Любые иные способы применения `async void` не предусмотрены моделью TAP и могут создавать указанные ниже проблемы.
  - Исключения, вызываемые в методе `async void`, невозможно перехватывать вне этого метода.
  - Методы `async void` очень трудно тестировать.
  - Методы `async void` могут иметь негативные побочные эффекты, если вызывающий объект не ожидает, что они будут асинхронными.

- **Будьте осторожны при использовании асинхронных лямбда-выражений в выражениях LINQ**

  Для лямбда-выражений в LINQ применяется отложенное выполнение. Это означает, что код может выполняться в произвольный момент, когда вы этого не ожидаете. Неправильное использование блокирующих задач при этом может привести к взаимоблокировке. Кроме того, вложение такого асинхронного кода может усложнить анализ выполнения кода. Асинхронное выполнение и LINQ — эффективные средства, но использовать их следует с максимальной осторожностью и ясным пониманием того, что вы делаете.

- **При написании кода ожидание задач следует реализовывать без блокирования**

  Блокирование текущего потока для ожидания завершения `Task` может привести к взаимоблокировкам и блокированию потоков контекста, что потребует более сложной обработки ошибок. В приведенной ниже таблице даются рекомендации по реализации ожидания задач без блокировки.

  Развернуть таблицу

| Используется... | Нерекомендуемый способ | Задача |
|---|---|---|
| `await` | `Task.Wait` или `Task.Result` | Получение результата фоновой задачи |
| `await Task.WhenAny` | `Task.WaitAny` | Ожидание завершения выполнения любой задачи |
| `await Task.WhenAll` | `Task.WaitAll` | Ожидание завершения выполнения всех задач |
| `await Task.Delay` | `Thread.Sleep` | Ожидание в течение заданного времени |

- **Рекомендуется использовать `ValueTask` во всех возможных случаях**

  В некоторых случаях возврат объекта `Task` из асинхронных методов может вызывать сложности. `Task` — это тип ссылки, поэтому его применение означает распределение объекта. В случаях, когда метод с модификатором `async` возвращает кэшированный результат или завершается синхронно, лишние распределения могут вызывать серьезные потери времени при выполнении фрагментов кода, зависящих от производительности. Эта проблема встает серьезно, если распределения происходят в коротких циклах. Дополнительные сведения см. в разделе Обобщенные асинхронные типы возвращаемых значений.

- **Рекомендуется использовать `ConfigureAwait(false)`**

  Часто возникает вопрос: "Когда же нужно использовать метод Task.ConfigureAwait(Boolean)?" Этот метод позволяет экземпляру `Task` настроить ожидающий объект. Это важный элемент, неправильная настройка которого может привести к снижению производительности и даже к взаимоблокировкам. Дополнительные сведения о `ConfigureAwait` см. в статье с вопросами и ответами по ConfigureAwait.

- **Пишите код с менее строгим отслеживанием состояния**

  Старайтесь, чтобы выполнение кода не зависело от состояния глобальных объектов или выполнения определенных методов. Оно должно зависеть только от возвращаемых методами значений. Почему?
  - Код будет проще анализировать.
  - Код будет проще тестировать.
  - Гораздо проще будет сочетать асинхронный и синхронный код.

- Как правило, можно полностью избежать состояний гонки.
- Зависимость от возвращаемых значений упрощает согласование асинхронного кода.
- Дополнительным преимуществом является то, что такой код хорошо работает с внедрением зависимостей.

Следует стремиться к достижению полной или почти полной ссылочной прозрачности в коде. Результатом будет предсказуемость базы кода, а также ее пригодность для тестирования и обслуживания.

# Полный пример

Приведенный ниже код — это полный текст файла *Program.cs* для примера.

C#

```
using System.Text.RegularExpressions;
using System.Windows;
using Microsoft.AspNetCore.Mvc;

class Button
{
    public Func<object, object, Task>? Clicked
    {
        get;
        internal set;
    }
}

class DamageResult
{
    public int Damage
    {
        get { return 0; }
    }
}

class User
{
    public bool isEnabled
    {
        get;
        set;
    }

    public int id
    {
        get;
        set;
    }
}

public class Program
{
    private static readonly Button s_downloadButton = new();
    private static readonly Button s_calculateButton = new();

    private static readonly HttpClient s_httpClient = new();

    private static readonly IEnumerable<string> s_urlList = new string[]
    {
            "https://learn.microsoft.com",
            "https://learn.microsoft.com/aspnet/core",
            "https://learn.microsoft.com/azure",
            "https://learn.microsoft.com/azure/devops",
            "https://learn.microsoft.com/dotnet",
            "https://learn.microsoft.com/dotnet/desktop/wpf/get-started/create-app-visual-studio",
            "https://learn.microsoft.com/education",
            "https://learn.microsoft.com/shows/net-core-101/what-is-net",
            "https://learn.microsoft.com/enterprise-mobility-security",
            "https://learn.microsoft.com/gaming",
            "https://learn.microsoft.com/graph",
            "https://learn.microsoft.com/microsoft-365",
            "https://learn.microsoft.com/office",
            "https://learn.microsoft.com/powershell",
            "https://learn.microsoft.com/sql",
            "https://learn.microsoft.com/surface",
```

```
        "https://dotnetfoundation.org",
        "https://learn.microsoft.com/visualstudio",
        "https://learn.microsoft.com/windows",
        "https://learn.microsoft.com/maui"
};

private static void Calculate()
{
    // <PerformGameCalculation>
    static DamageResult CalculateDamageDone()
    {
        return new DamageResult()
        {
            // Code omitted:
            //
            // Does an expensive calculation and returns
            // the result of that calculation.
        };
    }

    s_calculateButton.Clicked += async (o, e) =>
    {
        // This line will yield control to the UI while CalculateDamageDone()
        // performs its work. The UI thread is free to perform other work.
        var damageResult = await Task.Run(() => CalculateDamageDone());
        DisplayDamage(damageResult);
    };
    // </PerformGameCalculation>
}

private static void DisplayDamage(DamageResult damage)
{
    Console.WriteLine(damage.Damage);
}

private static void Download(string URL)
{
    // <UnblockingDownload>
    s_downloadButton.Clicked += async (o, e) =>
    {
        // This line will yield control to the UI as the request
        // from the web service is happening.
        //
        // The UI thread is now free to perform other work.
        var stringData = await s_httpClient.GetStringAsync(URL);
        DoSomethingWithData(stringData);
    };
    // </UnblockingDownload>
}

private static void DoSomethingWithData(object stringData)
{
    Console.WriteLine("Displaying data: ", stringData);
}

// <GetUsersForDataset>
private static async Task<User> GetUserAsync(int userId)
{
    // Code omitted:
    //
    // Given a user Id {userId}, retrieves a User object corresponding
    // to the entry in the database with {userId} as its Id.

    return await Task.FromResult(new User() { id = userId });
}

private static async Task<IEnumerable<User>> GetUsersAsync(IEnumerable<int> userIds)
{
    var getUserTasks = new List<Task<User>>();
    foreach (int userId in userIds)
    {
        getUserTasks.Add(GetUserAsync(userId));
    }

    return await Task.WhenAll(getUserTasks);
}
// </GetUsersForDataset>
```

```
    // <GetUsersForDatasetByLINQ>
    private static async Task<User[]> GetUsersAsyncByLINQ(IEnumerable<int> userIds)
    {
        var getUserTasks = userIds.Select(id => GetUserAsync(id)).ToArray();
        return await Task.WhenAll(getUserTasks);
    }
    // </GetUsersForDatasetByLINQ>

    // <ExtractDataFromNetwork>
    [HttpGet, Route("DotNetCount")]
    static public async Task<int> GetDotNetCount(string URL)
    {
        // Suspends GetDotNetCount() to allow the caller (the web server)
        // to accept another request, rather than blocking on this one.
        var html = await s_httpClient.GetStringAsync(URL);
        return Regex.Matches(html, @"\.NET").Count;
    }
    // </ExtractDataFromNetwork>

    static async Task Main()
    {
        Console.WriteLine("Application started.");

        Console.WriteLine("Counting '.NET' phrase in websites...");
        int total = 0;
        foreach (string url in s_urlList)
        {
            var result = await GetDotNetCount(url);
            Console.WriteLine($"{url}: {result}");
            total += result;
        }
        Console.WriteLine("Total: " + total);

        Console.WriteLine("Retrieving User objects with list of IDs...");
        IEnumerable<int> ids = new int[] { 1, 2, 3, 4, 5, 6, 7, 8, 9, 0 };
        var users = await GetUsersAsync(ids);
        foreach (User? user in users)
        {
            Console.WriteLine($"{user.id}: isEnabled={user.isEnabled}");
        }

        Console.WriteLine("Application ending.");
    }
}

// Example output:
//
// Application started.
// Counting '.NET' phrase in websites...
// https://learn.microsoft.com: 0
// https://learn.microsoft.com/aspnet/core: 57
// https://learn.microsoft.com/azure: 1
// https://learn.microsoft.com/azure/devops: 2
// https://learn.microsoft.com/dotnet: 83
// https://learn.microsoft.com/dotnet/desktop/wpf/get-started/create-app-visual-studio: 31
// https://learn.microsoft.com/education: 0
// https://learn.microsoft.com/shows/net-core-101/what-is-net: 42
// https://learn.microsoft.com/enterprise-mobility-security: 0
// https://learn.microsoft.com/gaming: 0
// https://learn.microsoft.com/graph: 0
// https://learn.microsoft.com/microsoft-365: 0
// https://learn.microsoft.com/office: 0
// https://learn.microsoft.com/powershell: 0
// https://learn.microsoft.com/sql: 0
// https://learn.microsoft.com/surface: 0
// https://dotnetfoundation.org: 16
// https://learn.microsoft.com/visualstudio: 0
// https://learn.microsoft.com/windows: 0
// https://learn.microsoft.com/maui: 6
// Total: 238
// Retrieving User objects with list of IDs...
// 1: isEnabled= False
// 2: isEnabled= False
// 3: isEnabled= False
// 4: isEnabled= False
// 5: isEnabled= False
// 6: isEnabled= False
// 7: isEnabled= False
```

```
// 8: isEnabled= False
// 9: isEnabled= False
// 0: isEnabled= False
// Application ending.
```

# Другие ресурсы

- Асинхронная модель программирования задач (C#).

**Совместная работа с нами на GitHub**

Источник этого содержимого можно найти на GitHub, где также можно создавать и просматривать проблемы и запросы на вытягивание. Дополнительные сведения см. в нашем руководстве для участников.

## Отзыв о .NET

.NET — это проект с открытым исходным кодом. Выберите ссылку, чтобы оставить отзыв:

Открыть проблему с документацией

Отзыв о продукте