ASP.NET Core. Веб-приложения

Асинхронное программирование



# На этом уроке

1. Изучим понятие асинхронности
2. Узнаем, зачем нужны ключевые слова async/await
3. Познакомимся с возвращаемыми типами асинхронных методов
4. Узнаем, что такое State Machine и как это связано с асинхронностью в .NET
5. Научимся обрабатывать ошибки в асинхронных методах
6. Взглянем на методы WaitAll, WaitAny, ConfigureAwait

Оглавление

[На этом уроке](#_4wfp8dsrmxhk)

[Понятие асинхронности](#_vdn9odz8azdj)

[Асинхронное программирование](#_izu5p8jlbpe6)

[Асинхронность в .NET](#_f7546oqdmq4x)

[State Machine](#_nv31z0oof9kz)

[Обработка ошибок асинхронных методов](#_5vd965ygxa29)

[Методы WaitAll, WaitAny](#_i642jj826xk5)

[Глоссарий](#_tyjcwt)

[Дополнительные материалы](#_44sinio)

[Используемые источники](#_z337ya)

# Понятие асинхронности

Асинхронность (греч.) — несовпадение с чем-либо во времени; неодновременность.

В качестве примера представим алгоритм вашей утренней рутины. Вы проснулись, теперь необходимо посетить ванную комнату, приготовить завтрак и собрать рюкзак в офис. Список дел, которые придется выполнить утром, можно представить следующим образом:

1. Сходить в ванную комнату.
2. Приготовить кофе и бутерброд.
3. Собрать рюкзак.

Теперь составим примерный алгоритм действий, который нужно выполнить, чтобы выполнить утренний список задач:

1. Умыть лицо
2. Почистить зубы
3. Принять душ
4. Включить чайник
5. Заварить кофе
6. Сделать бутерброд
7. Съесть бутерброд
8. Выпить кофе
9. Собрать рюкзак

Получился довольно большой набор. Тем не менее, вы уже наверняка отметили, что оптимальным с точки зрения временных затрат будет некая синхронизация нескольких задач сразу. Допустим, вы пришли на кухню, включили чайник, а затем ушли умывать лицо и чистить зубы. После того, как услышали сигнал о том, что вода вскипела, вы вернулись на кухню, взяли банку с кофе, добавили его в чашку и заполнили кипятком. Для того, чтобы кофе заварился, требуется время, а вы тем временем снова ушли в душ. После душа вы снова на кухне и уже поместили в тостер ломтики хлеба; тем временем вы зашли в комнату, взяли рюкзак и сложили в него вещи. По возвращении на кухню вы достали сыр из холодильника, бросили его на уже готовые тосты и начали есть готовый бутерброд, запивая кофе.

Нетрудно предположить, что если выполнять все действия синхронно (сначала посетить душ, а после вернуться на кухню и ждать, пока вскипит вода в чайнике) то потребуется больше времени на всю утреннюю рутину в целом. К счастью, многие из нас, сами того не подразумевая, оптимизируют свою рутину асинхронно.

# Асинхронное программирование

Асинхронность в программировании позволяет оптимизировать исполнение отдельных блоков задач программы, тем самым помогая повысить ее производительность и отзывчивость.

Подход асинхронного программирования заключается в выделении отдельной задачи из основного потока в некие асинхронные блоки, которые и позволят не блокировать основной поток выполнения программы.

Для примера, мы можем представить форму WPF-приложения, в котором есть кнопка “Соединение”. Предположим, что по событию нажатия данной кнопки происходит HTTP-запрос на некий ресурс, от которого мы получаем ответ и отображаем результат на экране.

В синхронном варианте выполнения приложение, выполняя HTTP-запрос, попросту перестанет отвечать на действия пользователя (обработка перемещения мышки, анимации и так далее). Окно программы попросту будет неактивно и это повлияет на пользовательский опыт использования.

Если же выполнить HTTP-запрос асинхронно, то основной поток приложения, который связан с интерфейсом пользователя, продолжит свою корректную работу, ожидая завершения HTTP-запроса.

В действительности, применение асинхронных подходов выходит далеко за рамки построения приложений с графическим интерфейсом. Асинхронность в целом полезна при любых ресурсоемких задачах (HTTP-запросы, сложные вычисления, соединения с базами данных и т.д.).

# Асинхронность в .NET

Как мы уже выяснили, асинхронный подход в разработке помогает избежать проблем, связанных с производительностью. Тем не менее, традиционные возможности написания асинхронного кода оказываются довольно трудоемкими.

В версии языка C# 5 добавлена поддержка ключевых слов async/await, которые вывели асинхронное программирование на качественно новый уровень.

Ключевое слово async используется в роли модификатора метода, то есть оно отмечает метод в качестве асинхронного. Ключевое слово await используется в качестве оператора ожидания вызова метода. То есть при запуске мы говорим компилятору ожидать выполнения чего-то.

|  |
| --- |
| static async Task<Coffee> MakeCoffee(int amountOfGroundCoffee) {   Console.WriteLine("Start making coffee...");  Console.WriteLine("Put ground coffee into the cup & add boiled water");  await Task.Delay(3000);  Console.WriteLine("Coffee is ready!");  return new Coffee(); } |

Данные ключевые слова позволили сделать казалось бы невозможное - асинхронный код можно писать в привычном любому разработчику синхронном виде. Компилятор при наличии слов async/await возьмет всю работу на себя. О том, произойдет под капотом async/await мы поговорим в следующем разделе.

# State Machine

Теперь следует ознакомиться с тем, что находится под капотом абстракций async/await. Для этого возьмем исходный код метода приготовления выше (из предыдущего раздела) и посмотрим на IL-код, который сгенерировал компилятор.

|  |
| --- |
| [CompilerGenerated] private sealed class <MakeCoffee>d\_\_1 : IAsyncStateMachine {  public int <>1\_\_state;   public AsyncTaskMethodBuilder<Coffee> <>t\_\_builder;   public int amountOfGroundCoffee;   private TaskAwaiter <>u\_\_1;   private void MoveNext()  {  int num = <>1\_\_state;  Coffee result;  try  {  TaskAwaiter awaiter;  if (num != 0)  {  Console.WriteLine("Start making coffee...");  Console.WriteLine("Put ground coffee into the cup & add boiled water");  awaiter = Task.Delay(3000).GetAwaiter();  if (!awaiter.IsCompleted)  {  num = (<>1\_\_state = 0);  <>u\_\_1 = awaiter;  <MakeCoffee>d\_\_1 stateMachine = this;  <>t\_\_builder.AwaitUnsafeOnCompleted(ref awaiter, ref stateMachine);  return;  }  }  else  {  awaiter = <>u\_\_1;  <>u\_\_1 = default(TaskAwaiter);  num = (<>1\_\_state = -1);  }  awaiter.GetResult();  Console.WriteLine("Coffee is ready!");  result = new Coffee();  }  catch (Exception exception)  {  <>1\_\_state = -2;  <>t\_\_builder.SetException(exception);  return;  }  <>1\_\_state = -2;  <>t\_\_builder.SetResult(result);  }   void IAsyncStateMachine.MoveNext()  {  //ILSpy generated this explicit interface implementation from .override directive in MoveNext  this.MoveNext();  }   [DebuggerHidden]  private void SetStateMachine(IAsyncStateMachine stateMachine)  {  }   void IAsyncStateMachine.SetStateMachine(IAsyncStateMachine stateMachine)  {  //ILSpy generated this explicit interface implementation from .override directive in SetStateMachine  this.SetStateMachine(stateMachine);  } }  public static void Main(string[] args) { }  [AsyncStateMachine(typeof(<MakeCoffee>d\_\_1))] [DebuggerStepThrough] private static Task<Coffee> MakeCoffee(int amountOfGroundCoffee) {  <MakeCoffee>d\_\_1 stateMachine = new <MakeCoffee>d\_\_1();  stateMachine.<>t\_\_builder = AsyncTaskMethodBuilder<Coffee>.Create();  stateMachine.amountOfGroundCoffee = amountOfGroundCoffee;  stateMachine.<>1\_\_state = -1;  stateMachine.<>t\_\_builder.Start(ref stateMachine);  return stateMachine.<>t\_\_builder.Task; } |

На первый взгляд это очень странный и запутанный код. Но, стоит помнить, что это автоматически сгенерированный код, поэтому он не является удобным для восприятия. Попробуем выделить главное из фрагмента выше:

IAsyncStateMachine - интерфейс конечного автомата. Содержит методы MoveNext() и SetStateMachine(IAsyncStateMachine). Первый перемещает конечный автомат в его следующее состояние а второй настраивает конечный автомат.

Стоит отметить, что сам по себе конечный автомат это некая абстракция, которая имеет конечное число состояний. Иными словами, это устройство, в которое можно что-то передать и обратно получить. Конечные автоматы очень полезны, когда следует скрыть сложную логику.

Возвращаясь к реализации автомата в .NET, суть его работы сводится к вызову метода MoveNext(). Взглянем на него еще раз:

|  |
| --- |
| private void MoveNext()  {  int num = <>1\_\_state;  Coffee result;  try  {  TaskAwaiter awaiter;  if (num != 0)  {  Console.WriteLine("Start making coffee...");  Console.WriteLine("Put ground coffee into the cup & add boiled water");  awaiter = Task.Delay(3000).GetAwaiter();  if (!awaiter.IsCompleted)  {  num = (<>1\_\_state = 0);  <>u\_\_1 = awaiter;  <MakeCoffee>d\_\_1 stateMachine = this;  <>t\_\_builder.AwaitUnsafeOnCompleted(ref awaiter, ref stateMachine); return;  }  }  else  {  awaiter = <>u\_\_1;  <>u\_\_1 = default(TaskAwaiter);  num = (<>1\_\_state = -1);  }  awaiter.GetResult();  Console.WriteLine("Coffee is ready!");  result = new Coffee();  }  catch (Exception exception)  {  <>1\_\_state = -2;  <>t\_\_builder.SetException(exception);  return;  }  <>1\_\_state = -2;  <>t\_\_builder.SetResult(result);  } |

Мы можем заметить, что кусочки нашего асинхронного метода MakeCoffee() просто скопированы в данный метод автомата. Еще раз вспомним как выглядит метод MakeCoffee():

|  |
| --- |
| static async Task<Coffee> MakeCoffee(int amountOfGroundCoffee) {   Console.WriteLine("Start making coffee...");  Console.WriteLine("Put ground coffee into the cup & add boiled water");  await Task.Delay(3000);  Console.WriteLine("Coffee is ready!");  return new Coffee(); } |

При этом, код автомата как бы разделился на две части: сначала синхронно выполняется все то, что было до вызова await Task.Delay(3000); , а затем создается некий “ожидатель” awaiter и собственно ожидает выполнения метода. В нашем случае это задержка на 3000 миллисекунд. При этом каждый раз происходит проверка выполнен ли метод if (!awaiter.IsCompleted) . Если он завершил работу, то блок с условием просто пропускается и мы переходим ко второй части метода MakeCoffee(), где собственно и возвращается объект кофе.

# Отмена задач

Асинхронное выполнение задачи может отнять много времени. В некоторых случаях, возникает потребность в прерывании задачи. В .NET для этого существует токен отмены операции CancellationToken.

Представим, что мы выполняем HTTP-запрос к стороннему сервису. Мы можем инициализировать токен отмены и передать его асинхронный метод. В данном случае мы реализуем отмену по таймауту:

|  |
| --- |
| private static readonly CancellationTokenSource cts = new CancellationTokenSource(); private static readonly HttpClient client = new HttpClient();  public static async Task Main(string[] args) {  try  {  cts.CancelAfter(10000);  await MakeACallToDocumentation();  }  catch (TaskCanceledException)  {  Console.WriteLine("Fetching docs is cancelled...");  }  finally  {  cts.Dispose();  } }  static async Task MakeACallToDocumentation() {  var response = await client.GetAsync("https://docs.microsoft.com", cts.Token);  var content = await response.Content.ReadAsStringAsync(cts.Token);  Console.WriteLine(content); } |

C помощью cts.CancelAfter(10000); мы определяем таймаут ожидания в виде 10000 мс. Если по истечении заданного периода задача не будет выполнена мы получим в консоли сообщение "Fetching docs is cancelled...". В ином случае выведется HTML-текст полученной страницы.

# Обработка ошибок асинхронных методов

Обработка ошибок асинхронных методов концептуально схожа с таковой в синхронных методах. Выражения с await помещаются в блоки try, в блоках catch происходит обработка исключения.

|  |
| --- |
| public static async Task Main(string[] args) {  try  {  var coffee2 = await MakeCoffee(11);  var coffee1 = await MakeCoffee(-1);  }  catch (Exception ex)  {  Console.WriteLine(ex.Message);  } }  static async Task<Coffee> MakeCoffee(int amountOfGroundCoffee) {  if (amountOfGroundCoffee < 0)  throw new ArgumentException($"Parameter {nameof(amountOfGroundCoffee)} cannot be less than zero.");    Console.WriteLine("Start making coffee...");  Console.WriteLine("Put ground coffee into the cup & add boiled water");  await Task.Delay(3000);  Console.WriteLine("Coffee is ready!");  return new Coffee(); } |

Стоит заметить, что использование слова await в блоках try-catch-finally появилось в языке C# 6.

Рекомендуется также избегать async void методов (т.е. методов, не возвращающих ничего). Дело в том, что обработка ошибка в подобных случаях может обернуться коварными последствиями: исключения async void-методов нельзя захватывать с помощью catch.

Поэтому следует использовать в качестве возвращаемых значений асинхронных методов либо Task, либо Task<T>.

# Методы WaitAll, WaitAny

Теперь подумаем над тем случаем, когда у нас есть много задач на выполнение. Когда нам необходимо дождаться выполнения первой задачи, мы используем метод Task.WhenAny:

|  |
| --- |
| var task1 = MakeCoffee(11); var task2 = MakeCoffee(12); var task3 = MakeCoffee(1);  // Здесь мы ожидаем завершения выполнения одной из задач var finishedTask = await Task.WhenAny(task1, task2, task3);  // И далее получаем наш кофе var coffee = await finishedTask; |

В случае же если нам важно дождаться выполнения всех задач мы используем Task.WhenAll:

|  |
| --- |
| // Здесь создаем набор задач на приготовление 3 чашек кофе var tasks = new List<Task<Coffee>>(); tasks.AddRange(new[]{task1, task2, task3});  // Ждем, пока все 3 чашки будут готовы await Task.WhenAll(tasks);  // И далее работаем с результатом tasks.ForEach(t => { Console.WriteLine(t); }); |

# Глоссарий

**State Machine** - метод конечных автоматов; позволяет моделировать процессы с помощью принципов конечного автомата.

**Асинхронность** (греч.) — несовпадение с чем-либо во времени; неодновременность.

# Практическое задание

Представим, что мы работаем с api системы блогов. Перед нами поставлена задача получить несколько постов сразу, обработать содержимое ответа и записать результат в текстовый файл.

Метод получения поста представлен следующим образом: **GET /posts/{id},** где id - идентификатор поста.

Формат ответа:

|  |
| --- |
| {  "userId": 2,  "id": 14,  "title": "voluptatem eligendi optio",  "body": "fuga et accusamus dolorum perferendis illo" } |

Необходимо написать консольное приложение, которое выполнит асинхронно 10 HTTP-запросов на получение постов из блога, по целочисленному идентификатору, начиная с id=4 по id=13. Для выполнения запросов необходимо использовать HttpClient (см. <https://docs.microsoft.com/ru-ru/dotnet/api/system.net.http.httpclient?view=net-5.0>).

В результате работы программы должен получиться текстовый файл под названием result.txt, в котором сохранена полученная информация о постах в следующем формате:

userId

id

title

body

Между постами присутствует пробел. Так, содержимое файла может выглядеть следующим образом:

|  |
| --- |
| 1  4  eum et est occaecati  ullam et saepe reiciendis voluptatem adipisci\nsit amet autem assumenda provident rerum culpa\nquis hic commodi nesciunt rem tenetur doloremque ipsam iure\nquis sunt voluptatem rerum illo velit  1  5  nesciunt quas odio  repudiandae veniam quaerat sunt sed\nalias aut fugiat sit autem sed est\nvoluptatem omnis possimus esse voluptatibus quis\nest aut tenetur dolor neque |

Всего в файле должно оказаться 10 постов.

Документация к api: <https://jsonplaceholder.typicode.com/>

Метод получения поста по id: <https://jsonplaceholder.typicode.com/posts/1>

# Дополнительные материалы

1. <https://docs.microsoft.com/ru-ru/dotnet/csharp/async>

# Используемые источники

1. <https://docs.microsoft.com/ru-ru/dotnet/csharp/async>
2. <https://docs.microsoft.com/ru-ru/archive/msdn-magazine/2013/march/async-await-best-practices-in-asynchronous-programming>