**№ 16 Платформа параллельных вычислений**

**Курносенко Софья**

**Параллельное программирование и библиотека TPL**

В эпоху многоядерных машин, которые позволяют параллельно выполнять сразу несколько процессов, стандартных средств работы с потоками в .NET уже оказалось недостаточно. Поэтому во фреймворк .NET была добавлена библиотека параллельных задач TPL (Task Parallel Library), основной функционал которой располагается в пространстве имен **System.Threading.Tasks**. Данная библиотека позволяет распараллелить задачи и выполнять их сразу на нескольких процессорах, если на целевом компьютере имеется несколько ядер. Кроме того, упрощается сама работа по созданию новых потоков. Поэтому начиная с .NET 4.0. рекомендуется использовать именно TPL и ее классы для создания многопоточных приложений, хотя стандартные средства и класс Thread по-прежнему находят широкое применение.

## Задачи и класс Task

В основе библиотеки TPL лежит концепция задач, каждая из которых описывает отдельную продолжительную операцию. В библиотеке классов .NET задача представлена специальным классом - классом **Task**, который находится в пространстве имен **System.Threading.Tasks**. Данный класс описывает отдельную задачу, которая запускается асинхронно в одном из потоков из пула потоков. Хотя ее также можно запускать синхронно в текущем потоке.

Для определения и запуска задачи можно использовать различные способы. Первый способ создание объекта Task и вызов у него метода Start:

using System;

using System.Threading.Tasks;

namespace ConsoleApplication1

{

class Program

{

static void Main(string[] args)

{

Task task = new Task(() => Console.WriteLine("Hello Task!"));

task.Start();

}

}

}

В качестве параметра объект Task принимает делегат Action, то есть мы можем передать любое действие, которое соответствует данному делегату, например, лямбда-выражение, как в данном случае, или ссылку на какой-либо метод. То есть в данном случае при выполнении задачи на консоль будет выводиться строка "Hello Task!".

А метод Start() собственно запускает задачу.

Второй способ заключается в использовании статического метода **Task.Factory.StartNew()**. Этот метод также в качестве параметра принимает делегат Action, который указывает, какое действие будет выполняться. При этом этот метод сразу же запускает задачу:

Task task = Task.Factory.StartNew(() => Console.WriteLine("Hello Task!"));

В качестве результата метод возвращает запущенную задачу.

Третий способ определения и запуска задач представляет использование статического метода **Task.Run()**:

Task task = Task.Run(() => Console.WriteLine("Hello Task!"));

Метод Task.Run() также в качестве параметра может принимать делегат Action - выполняемое действие и возвращает объект Task.

Определим небольшую программу, где используем все эти способы:

using System;

using System.Threading.Tasks;

namespace HelloApp

{

class Program

{

static void Main(string[] args)

{

Task task1 = new Task(() => Console.WriteLine("Task1 is executed"));

task1.Start();

Task task2 = Task.Factory.StartNew(() => Console.WriteLine("Task2 is executed"));

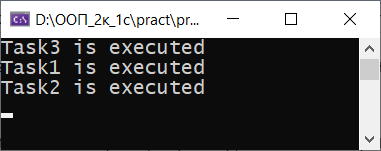
Task task3 = Task.Run(() => Console.WriteLine("Task3 is executed"));

Console.ReadLine();

}

}

}



Чтобы в консоли появились результаты выполнения задач нужно использовать: Console.ReadLine();

### Ожидание задачи

Важно понимать, что задачи не выполняются последовательно. Первая запущенная задача может завершить свое выполнение после последней задачи.

Или рассмотрим еще один пример:

using System;

using System.Threading.Tasks;

namespace TaskApp

{

class Program

{

static void Main(string[] args)

{

Task task = new Task(Display);

task.Start();

Console.WriteLine("Завершение метода Main");

Console.ReadLine();

}

static void Display()

{

Console.WriteLine("Начало работы метода Display");

Console.WriteLine("Завершение работы метода Display");

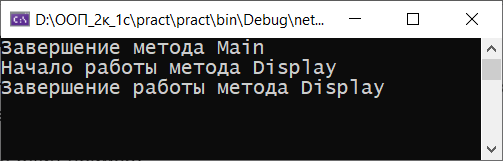
}

}

}

Класс Task в качестве параметра принимает метод Display, который соответствует делегату Action. Далее чтобы запустить задачу, вызываем метод Start: task.Start(), и после этого метод Display начнет выполняться во вторичном потоке. В конце метода Main выводит некоторый маркер-строку, что метод Main завершился.

Однако в данном случае консольный вывод может выглядеть следующим образом:



То есть мы видим, что даже когда основной код в методе Main уже отработал, запущенная ранее задача еще не завершилась.

Чтобы указать, что метод Main должен подождать до конца выполнения задачи, нам надо использовать метод Wait:

static void Main(string[] args)

{

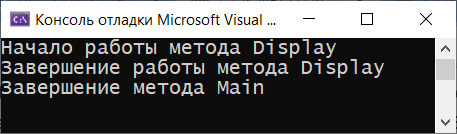
Task task = new Task(Display);

task.Start();

task.Wait();

Console.WriteLine("Завершение метода Main");

}



### Свойства класса Task

Класс Task имеет ряд свойств, с помощью которых мы можем получить информацию об объекте. Некоторые из них:

* **Id**: возвращает идентификатор текущей задачи
* **Status**: возвращает статус задачи

# ► Stopwatch Class

Namespace: [System.Diagnostics](https://docs.microsoft.com/en-us/dotnet/api/system.diagnostics?view=net-6.0)

Предоставляет набор методов и свойств, которые можно использовать для точного измерения прошедшего времени.

В следующем примере показано, как использовать класс [Stopwatch](https://docs.microsoft.com/en-us/dotnet/api/system.diagnostics.stopwatch?view=net-6.0) для определения времени выполнения приложения.

using System;

using System.Diagnostics;

using System.Threading;

class Program

{

static void Main(string[] args)

{

Stopwatch stopWatch = new Stopwatch();

stopWatch.Start();

Thread.Sleep(10000);

stopWatch.Stop();

// Get the elapsed time as a TimeSpan value.

TimeSpan ts = stopWatch.Elapsed;

// Format and display the TimeSpan value.

string elapsedTime = String.Format("{0:00}:{1:00}:{2:00}.{3:00}",

ts.Hours, ts.Minutes, ts.Seconds,

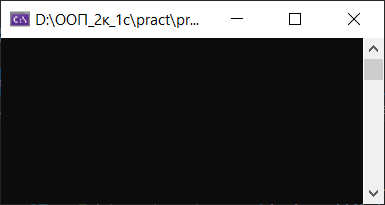
ts.Milliseconds / 10);

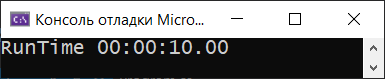
Console.WriteLine("RunTime " + elapsedTime);

}

}

Консоль сразу после запуска (поток спит):

  
Через 10 секунд:



**Свойства**

**Elapsed** получает общее прошедшее время, измеренное текущим экземпляром.

**Методы**

**Start()** начинает или возобновляет измерение прошедшего времени в течение определенного интервала.

**Stop()** прекращает измерение прошедшего времени за интервал.

### Массив задач

Также как и с потоками, мы можем создать и запустить массив задач. Можно определить все задачи в массиве непосредственно через объект Task:

Task[] tasks1 = new Task[3]

{

new Task(() => Console.WriteLine("First Task")),

new Task(() => Console.WriteLine("Second Task")),

new Task(() => Console.WriteLine("Third Task"))

};

// запуск задач в массиве

foreach (var t in tasks1)

t.Start();

Либо также можно использовать методы Task.Factory.StartNew или Task.Run и сразу запускать все задачи:

Task[] tasks2 = new Task[3];

int j = 1;

for (int i = 0; i < tasks2.Length; i++)

tasks2[i] = Task.Factory.StartNew(() => Console.WriteLine($"Task {j++}"));

Но в любом случае мы опять же можем столкнуться с тем, что все задачи из массива могут завершиться после того, как отработает метод Main, в котором запускаются эти задачи:

using System;

using System.Threading.Tasks;

class Program

{

static void Main(string[] args)

{

Task[] tasks1 = new Task[3]

{

new Task(() => Console.WriteLine("First Task")),

new Task(() => Console.WriteLine("Second Task")),

new Task(() => Console.WriteLine("Third Task"))

};

foreach (var t in tasks1)

t.Start();

Task[] tasks2 = new Task[3];

int j = 1;

for (int i = 0; i < tasks2.Length; i++)

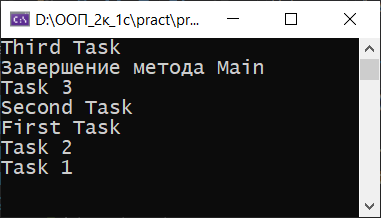
tasks2[i] = Task.Factory.StartNew(() => Console.WriteLine($"Task {j++}"));

Console.WriteLine("Завершение метода Main");

Console.ReadLine();

}

}



Если необходимо выполнять некоторый код лишь после того, как все задачи из массива завершатся, то применяется метод **Task.WaitAll(tasks)**:

static void Main(string[] args)

{

Task[] tasks1 = new Task[3]

{

new Task(() => Console.WriteLine("First Task")),

new Task(() => Console.WriteLine("Second Task")),

new Task(() => Console.WriteLine("Third Task"))

};

foreach (var t in tasks1)

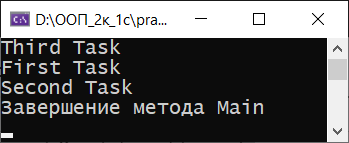
t.Start();

Task.WaitAll(tasks1); // ожидаем завершения задач

Console.WriteLine("Завершение метода Main");

Console.ReadLine();

}



Также мы можем применять метод **Task.WaitAny(tasks)**. Он ждет, пока завершится хотя бы одна задача из массива.

### Возвращение результатов из задач

Задачи могут не только выполняться как процедуры, но и возвращать определенные результаты:

using System;

using System.Threading.Tasks;

class Program

{

static void Main(string[] args)

{

Task<int> task1 = new Task<int>(() => Factorial(5));

task1.Start();

Console.WriteLine($"Факториал числа 5 равен {task1.Result}");

Task<Book> task2 = new Task<Book>(() =>

{

return new Book { Title = "Война и мир", Author = "Л. Толстой" };

});

task2.Start();

Book b = task2.Result;  // ожидаем получение результата

        Console.WriteLine($"Название книги: {b.Title}, автор: {b.Author}");

Console.ReadLine();

}

static int Factorial(int x)

{

int result = 1;

for (int i = 1; i <= x; i++)

{

result \*= i;

}

return result;

}

}

public class Book

{

public string Title { get; set; }

public string Author { get; set; }

}

Во-первых, чтобы задать возвращаемый из задачи тип объекта, мы должны типизировать Task. Например, Task<int> - в данном случае задача будет возвращать объект int.

И, во-вторых, в качестве задачи должен выполняться метод, возвращающий данный тип объекта. Например, в первом случае у нас в качестве задачи выполняется функция Factorial, которая принимает числовой параметр и также на выходе возвращает число.

Возвращаемое число будет храниться в свойстве Result: task1.Result. Нам не надо его приводить к типу int, оно уже само по себе будет представлять число.

То же самое и со второй задачей task2. В этом случае в лямбда-выражении возвращается объект Book. И также мы его получаем с помощью task2.Result

При этом при обращении к свойству Result программа текущий поток останавливает выполнение и ждет, когда будет получен результат из выполняемой задачи.

## Задачи продолжения

Задачи продолжения, или **continuation task**, позволяют определить задачи, которые выполняются после завершения других задач. Благодаря этому мы можем вызвать после выполнения одной задачи несколько других, определить условия их вызова, передать из предыдущей задачи в следующую некоторые данные.

Задачи продолжения похожи на методы обратного вызова, но фактически являются обычными задачами Task. Посмотрим на примере:

using System;

using System.Threading;

using System.Threading.Tasks;

namespace TaskApp

{

class Program

{

static void Main(string[] args)

{

Task task1 = new Task(() => {

Console.WriteLine($"Id задачи: {Task.CurrentId}");

});

// задача продолжения

Task task2 = task1.ContinueWith(Display);

task1.Start();

// ждем окончания второй задачи

task2.Wait();

Console.WriteLine("Выполняется работа метода Main");

Console.ReadLine();

}

static void Display(Task t)

{

Console.WriteLine($"\* Id задачи: {Task.CurrentId}");

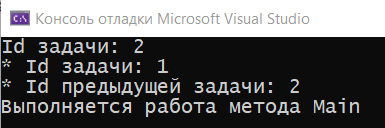
Console.WriteLine($"\* Id предыдущей задачи: {t.Id}");

Thread.Sleep(3000);

}

}

}

****

Первая задача задается с помощью лямбда-выражения, которое просто выводит id этой задачи. Вторая задача - задача продолжения задается с помощью метода **ContinueWith**, который в качестве параметра принимает делегат Action<Task>. То есть метод Display, который передается в данный метод в качестве значения параметра, должен принимать параметр типа Task.

Благодаря передачи в метод параметра Task, мы можем получить различные свойства предыдущей задачи, как например, в данном случае получает ее Id.

И после завершения задачи task1 сразу будет вызываться задача task2.

Также мы можем передавать конкретный результат работы предыдущей задачи:

using System;

using System.Threading.Tasks;

namespace HelloApp

{

class Program

{

static void Main(string[] args)

{

Task<int> task1 = new Task<int>(() => Sum(4, 5));

// задача продолжения

Task task2 = task1.ContinueWith(sum => Display(sum.Result));

task1.Start();

// ждем окончания второй задачи

task2.Wait();

Console.WriteLine("End of Main");

Console.ReadLine();

}

static int Sum(int a, int b) => a + b;

static void Display(int sum)

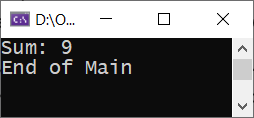
{

Console.WriteLine($"Sum: {sum}");

}

}

}



Подобным образом можно построить целую цепочку последовательно выполняющихся задач:

using System;

using System.Threading.Tasks;

namespace HelloApp

{

class Program

{

static void Main(string[] args)

{

Task task1 = new Task(() => {

Console.WriteLine($"Id задачи 1: {Task.CurrentId}");

});

// задача продолжения

Task task2 = task1.ContinueWith(Display);

Task task3 = task1.ContinueWith((Task t) =>

{

Console.WriteLine($"Id задачи 3: {Task.CurrentId}");

});

Task task4 = task2.ContinueWith((Task t) =>

{

Console.WriteLine($"Id задачи 4: {Task.CurrentId}");

});

task1.Start();

Console.ReadLine();

}

static void Display(Task t)

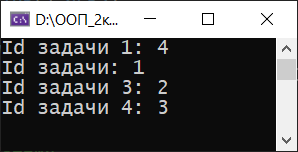
{

Console.WriteLine($"Id задачи: {Task.CurrentId}");

}

}

}



## Класс Parallel

Класс Parallel также является частью TPL и предназначен для упрощения параллельного выполнения кода. Parallel имеет ряд методов, которые позволяют распараллелить задачу.

Одним из методов, позволяющих параллельное выполнение задач, является метод **Invoke**:

using System;

using System.Threading.Tasks;

using System.Threading;

namespace HelloApp

{

class Program

{

static void Main(string[] args)

{

Parallel.Invoke(

Display,

() => {

Console.WriteLine($"\*\*Выполняется задача {Task.CurrentId}");

Thread.Sleep(3000);

},

() => Factorial(5)

);

Console.ReadLine();

}

static void Display()

{

Console.WriteLine($"\*Выполняется задача {Task.CurrentId}");

Thread.Sleep(3000);

}

static void Factorial(int x)

{

int result = 1;

for (int i = 1; i <= x; i++)

{

result \*= i;

}

Console.WriteLine($"\*\*\*Выполняется задача {Task.CurrentId}");

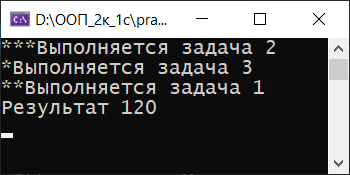
Thread.Sleep(3000);

Console.WriteLine($"Результат {result}");

}

}

}



Метод Parallel.Invoke в качестве параметра принимает массив объектов Action, то есть мы можем передать в данный метод набор методов, которые будут вызываться при его выполнении. Количество методов может быть различным, но в данном случае мы определяем выполнение трех методов. Опять же, как и в случае с классом Task мы можем передать либо название метода, либо лямбда-выражение.

И таким образом, при наличии нескольких ядер на целевой машине данные методы будут выполняться параллельно на различных ядрах.

### Parallel.For

Метод Parallel.For позволяет выполнять итерации цикла параллельно. Он имеет следующее определение: **For(int, int, Action<int>)**, где первый параметр задает начальный индекс элемента в цикле, а второй параметр - конечный индекс. Третий параметр - делегат Action - указывает на метод, который будет выполняться один раз за итерацию. Данный **метод должен принимать параметр типа int!**

using System;

using System.Threading.Tasks;

using System.Threading;

namespace HelloApp

{

class Program

{

static void Main(string[] args)

{

Parallel.For(1, 10, Factorial);

Console.ReadLine();

}

static void Factorial(int x)

{

int result = 1;

for (int i = 1; i <= x; i++)

{

result \*= i;

}

Console.WriteLine($"Выполняется задача {Task.CurrentId}");

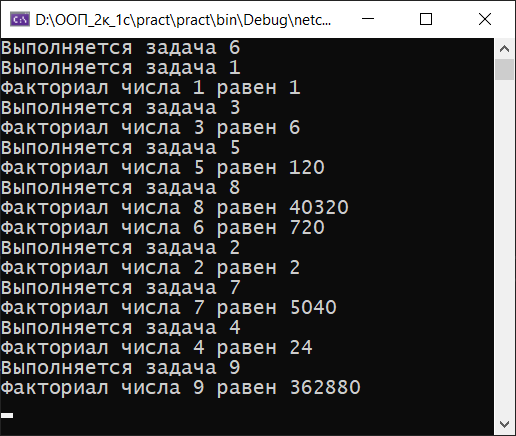
Console.WriteLine($"Факториал числа {x} равен {result}");

Thread.Sleep(3000);

}

}

}



В данном случае в качестве первого параметра в метод Parallel.For передается число 1, а в качестве второго - число 10. Таким образом, метод будет вести итерацию с 1 до 9 включительно. Третий параметр представляет метод, подсчитывающий факториал числа. Так как этот параметр представляет тип Action<int>, то этот метод в качестве параметра должен принимать объект int.

А в качестве значения параметра в этот метод передается счетчик, который проходит в цикле от 1 до 9 включительно. И метод Factorial, таким образом, вызывается 9 раз.

### Parallel.ForEach

Метод Parallel.ForEach осуществляет итерацию по коллекции, реализующей интерфейс IEnumerable, подобно циклу foreach, только осуществляет параллельное выполнение перебора. Он имеет следующее определение: ParallelLoopResult ForEach<TSource>(IEnumerable<TSource> source,Action<TSource> body), где первый параметр представляет перебираемую коллекцию, а второй параметр - делегат, выполняющийся один раз за итерацию для каждого перебираемого элемента коллекции.

На выходе метод возвращает структуру ParallelLoopResult, которая содержит информацию о выполнении цикла.

using System;

using System.Threading.Tasks;

using System.Threading;

using System.Collections.Generic;

namespace HelloApp

{

class Program

{

static void Main(string[] args)

{

ParallelLoopResult result = Parallel.ForEach<int>(new List<int>() { 1, 3, 5, 8 },

Factorial);

Console.ReadLine();

}

static void Factorial(int x)

{

int result = 1;

for (int i = 1; i <= x; i++)

{

result \*= i;

}

Console.WriteLine($"Выполняется задача {Task.CurrentId}");

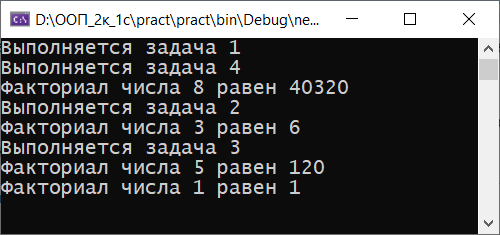
Console.WriteLine($"Факториал числа {x} равен {result}");

Thread.Sleep(3000);

}

}

}



В данном случае поскольку мы используем коллекцию объектов int, то и метод, который мы передаем в качестве второго параметра, должен в качестве параметра принимать значение int.

### Выход из цикла

В стандартных циклах for и foreach предусмотрен преждевременный выход из цикла с помощью оператора break. В методах Parallel.ForEach и Parallel.For мы также можем, не дожидаясь окончания цикла, выйти из него:

using System;

using System.Threading.Tasks;

using System.Threading;

using System.Collections.Generic;

namespace HelloApp

{

class Program

{

static void Main(string[] args)

{

ParallelLoopResult result = Parallel.For(1, 10, Factorial);

if (!result.IsCompleted)

Console.WriteLine($"Выполнение цикла завершено на итерации {result.LowestBreakIteration}");

Console.ReadLine();

}

static void Factorial(int x, ParallelLoopState pls)

{

int result = 1;

for (int i = 1; i <= x; i++)

{

result \*= i;

if (i == 5)

pls.Break();

}

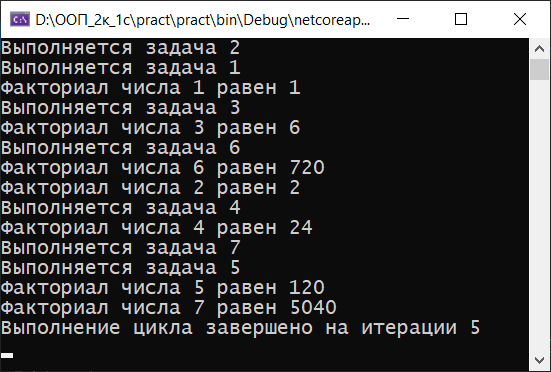
Console.WriteLine($"Выполняется задача {Task.CurrentId}");

Console.WriteLine($"Факториал числа {x} равен {result}");

}

}

}



Здесь метод Factorial, обрабатывающий каждую итерацию, принимает дополнительный параметр - объект **ParallelLoopState**. И если счетчик в цикле достигнет значения 5, вызывается метод Break. Благодаря чему система осуществит выход и прекратит выполнение метода Parallel.For при первом удобном случае.

Методы Parallel.ForEach и Parallel.For возвращают объект **ParallelLoopResult**, наиболее значимыми свойствами которого являются два следующих:

* **IsCompleted**: определяет, завершилось ли полное выполнение параллельного цикла
* **LowestBreakIteration**: возвращает индекс, на котором произошло прерывание работы цикла

Так как у нас на индексе равном 5 происходит прерывание, то свойство IsCompleted будет иметь значение false, а LowestBreakIteration будет равно 5.

## Отмена задач и параллельных операций. CancellationToken

Параллельное выполнение задач может занимать много времени. И иногда может возникнуть необходимость прервать выполняемую задачу. Для этого .NET предоставляет класс CancellationToken:

using System;

using System.Threading.Tasks;

using System.Threading;

using System.Collections.Generic;

namespace HelloApp

{

class Program

{

static void Main(string[] args)

{

CancellationTokenSource cancelTokenSource = new CancellationTokenSource();

CancellationToken token = cancelTokenSource.Token;

int number = 6;

Task task1 = new Task(() =>

{

int result = 1;

for (int i = 1; i <= number; i++)

{

if (token.IsCancellationRequested)

{

Console.WriteLine("Операция прервана");

return;

}

result \*= i;

Console.WriteLine($"Факториал числа {number} равен {result}");

Thread.Sleep(5000);

}

});

task1.Start();

Console.WriteLine("Введите Y для отмены операции или другой символ для ее продолжения:");

string s = Console.ReadLine();

if (s == "Y")

cancelTokenSource.Cancel();

Console.Read();

}

}

}

Для отмены операции нам надо создать и использовать токен. Вначале создается объект CancellationTokenSource:

|  |  |
| --- | --- |
| 1 | CancellationTokenSource cancelTokenSource = new CancellationTokenSource(); |

Затем из него получаем сам токен:

|  |  |
| --- | --- |
| 1 | CancellationToken token = cancelTokenSource.Token; |

Чтобы отменить операцию, необходимо вызвать метод Cancel() у объекта CancellationTokenSource:

|  |  |
| --- | --- |
| 1 | cancelTokenSource.Cancel(); |

В самой операции мы можем отловить выставление токена с помощью условной конструкции:

|  |  |
| --- | --- |
| 1  2  3  4  5 | if (token.IsCancellationRequested)  {      Console.WriteLine("Операция прервана токеном");      return;  } |

Если был вызван метод cancelTokenSource.Cancel(), то выражение token.IsCancellationRequested возвращает true.

Если операция представляет внешний метод, то ему надо передавать в качестве одного из параметров токен:

using System;

using System.Threading.Tasks;

using System.Threading;

using System.Collections.Generic;

namespace HelloApp

{

class Program

{

static void Main(string[] args)

{

CancellationTokenSource cancelTokenSource = new CancellationTokenSource();

CancellationToken token = cancelTokenSource.Token;

Task task1 = new Task(() => Factorial(5, token));

task1.Start();

Console.WriteLine("Введите Y для отмены операции или любой другой символ для ее продолжения:");

string s = Console.ReadLine();

if (s == "Y")

cancelTokenSource.Cancel();

Console.ReadLine();

}

static void Factorial(int x, CancellationToken token)

{

int result = 1;

for (int i = 1; i <= x; i++)

{

if (token.IsCancellationRequested)

{

Console.WriteLine("Операция прервана токеном");

return;

}

result \*= i;

Console.WriteLine($"Факториал числа {x} равен {result}");

Thread.Sleep(5000);

}

}

}

}

### Отмена параллельных операций Parallel

Для отмены выполнения параллельных операций, запущенных с помощью методов **Parallel.For()** и **Parallel.ForEach()**, можно использовать перегруженные версии данных методов, которые принимают в качестве параметра объект **ParallelOptions**. Данный объект позволяет установить токен:

using System;

using System.Threading.Tasks;

using System.Threading;

using System.Collections.Generic;

namespace HelloApp

{

class Program

{

static void Main(string[] args)

{

CancellationTokenSource cancelTokenSource = new CancellationTokenSource();

CancellationToken token = cancelTokenSource.Token;

new Task(() =>

{

Thread.Sleep(400);

cancelTokenSource.Cancel();

}).Start();

try

{

Parallel.ForEach<int>(new List<int>() { 1, 2, 3, 4, 5, 6, 7, 8 },

new ParallelOptions { CancellationToken = token }, Factorial);

// или так

//Parallel.For(1, 8, new ParallelOptions { CancellationToken = token }, Factorial);

}

catch (OperationCanceledException)

{

Console.WriteLine("Операция прервана");

}

finally

{

cancelTokenSource.Dispose();

}

Console.ReadLine();

}

static void Factorial(int x)

{

int result = 1;

for (int i = 1; i <= x; i++)

{

result \*= i;

}

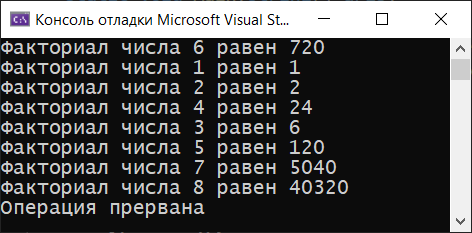
Console.WriteLine($"Факториал числа {x} равен {result}");

Thread.Sleep(3000);

}

}

}



В параллельной запущеной задаче через 400 миллисекунд происходит вызов cancelTokenSource.Cancel(), в результате программа выбрасывает исключение OperationCanceledException, и выполнение параллельных операций прекращается.

**► Параллельные коллекции**

В версию 4.0 среды .NET Framework добавлено новое пространство имен **System.Collections.Concurrent**. Оно содержит коллекции, которые являются потокобезопасными и специально предназначены для параллельного программирования. Это означает, что они могут безопасно использоваться в многопоточной программе, где возможен одновременный доступ к коллекции со стороны двух или больше параллельно исполняемых потоков.

Для безопасного в отношении потоков доступа к коллекциям определен интерфейс IProducerConsumerCollection<T>. Наиболее важными методами этого интерфейса являются TryAdd() и TryTake(). Метод TryAdd() пытается добавить элемент в коллекцию, но это может не получиться, если коллекция заблокирована от добавления элементов. Метод возвращает булевское значение, сообщающее об успехе или неудаче операции.

TryTake() работает аналогичным образом, информируя вызывающий код об успехе или неудаче, и в случае успеха возвращает элемент из коллекции.

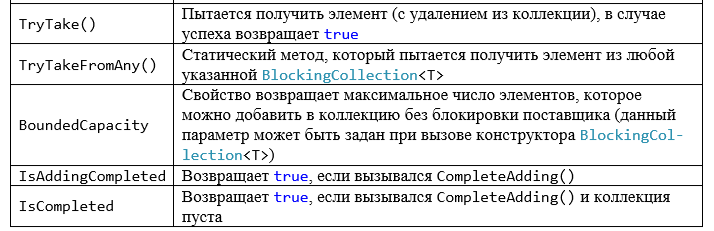
**BlockingCollection<T>** ─ параллельная коллекция, которая осуществляет блокировку и ожидает, пока не появится возможность выполнить действие по добавлению или извлечению элемента. BlockingCollection<T> предлагает интерфейс для добавления и извлечения элементов методами **Add**() и **Take**(). Эти методы блокируют поток и затем ожидают, пока не появится возможность выполнить задачу.

Метод Add() имеет перегрузку, которой можно также передать CancellationToken. Эта лексема всегда отменяет блокирующий вызов.

Если не нужно, чтобы поток ожидал бесконечное время, и не хотите отменять вызов извне, доступны также методы TryAdd() и TryTake(). В них можно указать значение таймаута — максимального периода времени, в течение которого вы готовы блокировать поток и ждать, пока вызов не даст сбой.

Класс BlockingCollection <T> ─ реализация шаблона «поставщик потребитель».





Давайте рассмотрим пример применения параллельных коллекций:

using System;

using System.Collections.Concurrent;

using System.Threading.Tasks;

namespace ConsoleApplication1

{

class Program

{

static BlockingCollection<int> bc;

static void producer()

{

for (int i = 0; i < 100; i++)

{

bc.Add(i \* i);

Console.WriteLine("Производится число " + i \* i);

}

bc.CompleteAdding();

}

static void consumer()

{

int i;

while (!bc.IsCompleted)

{

if (bc.TryTake(out i))

Console.WriteLine("Потребляется число: " + i);

}

}

static void Main()

{

bc = new BlockingCollection<int>(4);

// Создадим задачи поставщика и потребителя

Task Pr = new Task(producer);

Task Cn = new Task(consumer);

// Запустим задачи

Pr.Start();

Cn.Start();

try

{

Task.WaitAll(Cn, Pr);

}

catch (Exception ex)

{

Console.WriteLine(ex);

}

finally

{

Cn.Dispose();

Pr.Dispose();

bc.Dispose();

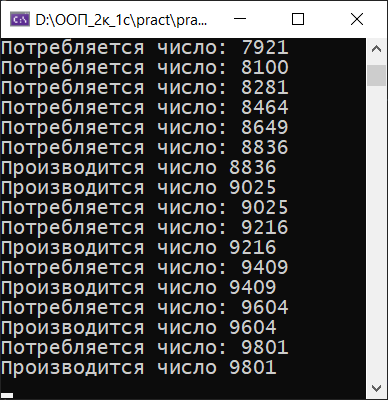
}

Console.ReadLine();

}

}

}



[BlockingCollection<T>(Int32)](https://docs.microsoft.com/en-us/dotnet/api/system.collections.concurrent.blockingcollection-1.-ctor?view=net-6.0#System_Collections_Concurrent_BlockingCollection_1__ctor_System_Int32_) инициализирует новый экземпляр класса BlockingCollection <T> с указанной верхней границей.

# Aсинхронное программирование

## Асинхронные методы, async и await

**Асинхронность** позволяет вынести отдельные задачи из основного потока в специальные асинхронные методы или блоки кода. Особенно это актуально в графических программах, где продолжительные задачи могу блокировать интерфейс пользователя. И чтобы этого не произошло, нужно задействовать асинхронность. Также асинхронность несет выгоды в веб-приложениях при обработке запросов от пользователей, при обращении к базам данных или сетевым ресурсам. При больших запросах к базе данных асинхронный метод просто уснет на время, пока не получит данные от БД, а основной поток сможет продолжить свою работу. В синхронном же приложении, если бы код получения данных находился в основном потоке, этот поток просто бы блокировался на время получения данных.

Ключевыми для работы с асинхронными вызовами в C# являются два ключевых слова: **async** и **await**, цель которых - упростить написание асинхронного кода. Они используются вместе для создания асинхронного метода.

**Асинхонный метод** обладает следующими признаками:

* В заголовке метода используется модификатор **async**
* Метод содержит одно или несколько выражений **await**
* В качестве возвращаемого типа используется один из следующих:
  + void
  + Task
  + Task<T>
  + ValueTask<T>

Асинхронный метод, как и обычный, может использовать любое количество параметров или не использовать их вообще. Однако асинхронный метод не может определять параметры с модификаторами **out** и **ref**.

Также стоит отметить, что слово **async**, которое указывается в определении метода, не делает автоматически метод асинхронным. Оно лишь указывает, что данный метод может содержать одно или несколько выражений **await**.

Рассмотрим пример асинхронного метода:

using System;

using System.Threading;

using System.Threading.Tasks;

namespace HelloApp

{

class Program

{

static void Factorial()

{

int result = 1;

for (int i = 1; i <= 6; i++)

{

result \*= i;

}

Thread.Sleep(8000);

Console.WriteLine($"Факториал равен {result}");

}

        // определение асинхронного метода

        static async void FactorialAsync()

{

Console.WriteLine("Начало метода FactorialAsync"); // выполняется синхронно

            await Task.Run(() => Factorial());                // выполняется асинхронно

            Console.WriteLine("Конец метода FactorialAsync");

}

static void Main(string[] args)

{

FactorialAsync(); // вызов асинхронного метода

Console.WriteLine("Введите число: ");

int n = Int32.Parse(Console.ReadLine());

Console.WriteLine($"Квадрат числа равен {n \* n}");

Console.Read();

}

}

}

Здесь прежде всего определен обычный метод подсчета факториала. Для имитации долгой работы в нем используется задержка на 8 секунд с помощью метода Thread.Sleep(). Условно это некоторый метод, который выполняет некоторую работу продолжительное время. Но для упрощения понимания он просто подсчитывает факториал числа 6.

Также здесь определен асинхронный метод FactorialAsync(). Асинхронным он является потому, что имеет в определении перед возвращаемым типом модификатор **async**, его возвращаемым типом является void, и в теле метода определено выражение **await**.

Выражение **await** определяет задачу, которая будет выполняться асинхронно. В данном случае подобная задача представляет выполнение функции факториала:

|  |  |
| --- | --- |
| 1 | await Task.Run(()=>Factorial()); |

По негласным правилам в названии асинхроннных методов принято использовать суффикс *Async* - Factorial**Async**(), хотя в принципе это необязательно делать.

Сам факториал мы получаем в асинхронном методе FactorialAsync. Асинхронным он является потому, что он объявлен с модификатором async и содержит использование ключевого слова await.

И в методе Main мы вызываем этот асинхронный метод.

Посмотрим, какой у программы будет консольный вывод:

Начало метода FactorialAsync

Введите число:

7

Квадрат числа равен 49

Конец метода Main

Факториал равен 720

Окончание метода FactorialAsync

Разберем поэтапно, что здесь происходит:

1. Запускается метод Main, в котором вызывается асинхронный метод FactorialAsync.
2. Метод FactorialAsync начинает выполняться **синхронно** вплоть до выражения await.
3. Выражение await запускает асинхронную задачу Task.Run(()=>Factorial())
4. Пока выполняется асинхронная задача Task.Run(()=>Factorial()) (а она может выполняться довольно продолжительное время), выполнение кода возвращается в вызывающий метод - то есть в метод Main. В методе Main нам будет предложено ввести число для вычисления квадрата числа.

В этом и преимущество асинхронных методов - асинхронная задача, которая может выполняться довольно долгое время, не блокирует метод Main, и мы можем продолжать работу с ним, например, вводить и обрабатывать данные.

1. Когда асинхронная задача завершила свое выполнение (в случае выше - подсчитала факториал числа), продолжает работу асинхронный метод FactorialAsync, который вызвал асинхронную задачу.

Функция факториала, возможно, представляет не самый показательный пример, так как в реальности в данном случае нет смысла делать ее асинхронной. Но рассмотрим другой пример - чтение-запись файла:

using System;

using System.Threading;

using System.Threading.Tasks;

using System.IO;

namespace HelloApp

{

class Program

{

static async void ReadWriteAsync()

{

string s = "Hello world! One step at a time";

// hello.txt - файл, который будет записываться и считываться

using (StreamWriter writer = new StreamWriter("hello.txt", false))

{

await writer.WriteLineAsync(s);  // асинхронная запись в файл

            }

using (StreamReader reader = new StreamReader("hello.txt"))

{

string result = await reader.ReadToEndAsync();  // асинхронное чтение из файла

                Console.WriteLine(result);

}

}

static void Main(string[] args)

{

ReadWriteAsync();

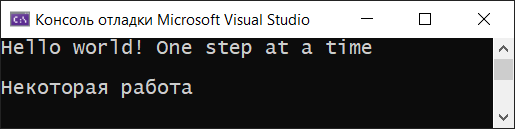
Console.WriteLine("Некоторая работа");

Console.Read();

}

}

}



Асинхронный метод ReadWriteAsync() выполняет запись в файл некоторой строки и затем считывает записанный файл. Подобные операции могут занимать продолжительное время, особенно при больших объемах данных, поэтому такие операции лучше делать асинхронными.

Фреймворк .NET уже имеет встроенную поддержку таких операций. Например, в классе StreamWriter определен метод **WriteLineAsync()**. По сути он уже представляет асинхронную операцию и принимает в качестве параметра некоторую строку, которую надо записать в файл. Поскольку этот метод представляет асинхронную операцию, то вызов этого метода мы можем оформить в выражение **await**:

|  |  |
| --- | --- |
| 1 | await writer.WriteLineAsync(s);  // асинхронная запись в файл |

Аналогично в классе StreamReader определен метод **ReadToEndAsync()**, который также представляет асинхронную операцию и который возвращает весь считанный текст.

Во фреймворке .NET Core определено много подобных методов. Как правило, они связаны с работой с файлами, отправкой сетевых запросов или запросов к базе данных. Их легко узнать по суффиксу **Async**. То есть если метод имеет подобный суффикс в названии, то с большей степенью вероятности его можно использовать в выражении await.

Далее в методе Main вызывается асинхронный метод ReadWriteAsync:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | static void Main(string[] args)  {      ReadWriteAsync();        Console.WriteLine("Некоторая работа");      Console.Read();  } |

И опять же, когда выполнение в методе ReadWriteAsync доходит до первого выражения await, управление возвращается в метод Main, и мы можем продолжать с ним работу. Запись в файл и считывание файла будут производиться параллельно и не будут блокировать работу метода Main.

### Определение асинхронной операции

Как выше уже было сказано, фреймворк .NET Core имеет много встроенных методов, которые представляют асинхронную операцию. Они заканчиваются на суффикс Async. И перед вызывами подобных методов мы можем указывать оператор **await**. Например:

|  |  |
| --- | --- |
| 1  2 | StreamWriter writer = new StreamWriter("hello.txt", false);  await writer.WriteLineAsync("Hello");  // асинхронная запись в файл |

Либо мы сами можем определить асинхронную операцию, используя метод **Task.Run()**:

static void Factorial()

{

int result = 1;

for (int i = 1; i <= 6; i++)

{

result \*= i;

}

Thread.Sleep(8000);

Console.WriteLine($"Факториал равен {result}");

}

// определение асинхронного метода

static async void FactorialAsync()

{

await Task.Run(() => Factorial()); // вызов асинхронной операции

}

Можно определить асинхронную операцию с помощью лямбда-выражения:

static async void FactorialAsync()

{

await Task.Run(() =>

{

int result = 1;

for (int i = 1; i <= 6; i++)

{

result \*= i;

}

Thread.Sleep(8000);

Console.WriteLine($"Факториал равен {result}");

});

}

### Передача параметров в асинхронную операцию

Выше вычислялся факториал 6, но, допустим, мы хотим вычислять факториалы разных чисел:

using System;

using System.Threading;

using System.Threading.Tasks;

namespace HelloApp

{

class Program

{

static void Factorial(int n)

{

int result = 1;

for (int i = 1; i <= n; i++)

{

result \*= i;

}

Thread.Sleep(5000);

Console.WriteLine($"Факториал равен {result}");

}

        // определение асинхронного метода

        static async void FactorialAsync(int n)

{

await Task.Run(() => Factorial(n));

}

static void Main(string[] args)

{

FactorialAsync(5);

FactorialAsync(6);

Console.WriteLine("Некоторая работа");

Console.Read();

}

}

}

### Получение результата из асинхронной операции

Асинхронная операция может возвращать некоторый результат, получить который мы можем так же, как и при вызове обычного метода:

using System;

using System.Threading;

using System.Threading.Tasks;

namespace HelloApp

{

class Program

{

static int Factorial(int n)

{

int result = 1;

for (int i = 1; i <= n; i++)

{

result \*= i;

}

return result;

}

        // определение асинхронного метода

        static async void FactorialAsync(int n)

{

int x = await Task.Run(() => Factorial(n));

Console.WriteLine($"Факториал равен {x}");

}

static void Main(string[] args)

{

FactorialAsync(5);

FactorialAsync(6);

Console.Read();

}

}

}

Метод Factorial возвращает значение типа int, это значение мы можем получить, просто присвоив результат асинхронной операции переменной данного типа: int x = await Task.Run(()=>Factorial(n));