

Введение в асинхронные задачи

Класс Task

Позволяет упростить написание параллельного кода без необходимости работы непосредственно с потоками или пулом потоков.

Задачи создаются в виде объектов класса **Task** различными способами:

- с использованием делегата **Action** и именного метода;
- с использованием анонимного делегата;
- с использованием лямбда-выражения и именного метода;
- с использованием лямбда-выражения и анонимного метода.

Для создания объектов класса **Task** используется *конструктор*:

public Task(Action действие)

где **действие** – точка входа в код, представляющий задачу, **Action** – делегат, определенный в пространстве имен *System*.

Пример:

Task t = new Task (new Action(Print));

где **Print** – метод программы, который будет выполнен в отдельной задаче.

Пример. Создать задачи разными способами

```
using System;
using System.Threading.Tasks;
class Program
{
    static void Hello()
    {
        Console.WriteLine("Hello!");
    }
    static void Main()
    {
        // Используем обычный метод
        Task t1 = new Task(Hello);
        // Используем делегат Action
        Task t2 = new Task(new Action(Hello));
    }
}
```

```
// Используем безымянный делегат
Task t3 = new Task(delegate
{
    Hello();
});
// Используем лямбда-выражение
Task t4 = new Task(() => Hello());
Task t5 = new Task(() =>
{
    Hello();
});
Task t6 = new Task(() =>
{
    Console.WriteLine("Hello, world!");
});
```

```
}  
    // Запускаем задачи  
    t1.Start(); t2.Start(); t3.Start();  
    t4.Start(); t5.Start(); t6.Start();  
    // Дожидаемся завершения задач  
    Task.WaitAll(t1, t2, t3, t4, t5, t6);  
}
```

```
Hello!  
Hello!  
Hello!  
Hello!  
Hello!  
Hello, world!
```

```
Hello!  
Hello!  
Hello!  
Hello, world!  
Hello!  
Hello!
```

```
Hello!  
Hello!  
Hello!  
Hello!  
Hello, world!  
Hello!
```

```
Hello, world!  
Hello!  
Hello!  
Hello!  
Hello!  
Hello!
```

Работа с задачами:

- объявление задачи,
- добавление задачи в *очередь* готовых задач,
- ожидание завершения выполнения задачи.

```
Task task1 = new Task(SomeWork);  
task1.Start();  
task1.Wait();
```

Вызов метода **Start** для задачи не создает новый поток, а помещает задачу в *очередь готовых задач* – *пул потоков*. *Планировщик* (TaskScheduler) в соответствии со своими правилами распределяет готовые задачи по рабочим потокам. Действия планировщика можно корректировать с помощью параметров задач. Момент фактического запуска задачи в общем случае не определен и зависит от загруженности пула потоков.

Класс TaskFactory

Класс **TaskFactory** кодирует некоторые распространенные шаблоны класса **Task** в методы, которые получают параметры по умолчанию, настраиваемые посредством своих конструкторов. В классе **TaskFactory** предоставляются различные методы, упрощающие создание задач и управление ими. По умолчанию объект класса **TaskFactory** может быть получен из свойства **Factory**, доступного только для чтения в классе **Task**, используя это свойство, можно вызвать любые методы класса **TaskFactory**.

Метод StartNew()

Форма вызова:

```
public Task StartNew(Action действие)
```

где **действие** – точка входа в исполняемую задачу.

Сначала в методе `StartNew()` автоматически создается экземпляр класса `Task` для действия, определяемого параметром **действие**, а затем планируется запуск задачи на исполнение.

Т.о. нет необходимости использовать метод `Start()`.

Способы запуска задач с использованием **TaskFactory**:

- Использование фабрики задач

```
TaskFactory tf = new TaskFactory();
```

```
Task t = tf.StartNew(Действие);
```

- Использование фабрики задач через задачу

```
Task t = Task.Factory.StartNew(Действие);
```

- Использование конструктора Task

```
Task t = new Task(Действие);
```

```
t.Start();
```

- Вызов именного метода с помощью лямбда выражения

```
Task t = Task.Factory.StartNew(() => DoSom());
```

- Вызов анонимного метода с помощью лямбда выражения

```
Task t = new Task.Factory.StartNew(() =>  
{ Console.WriteLine("Hello"); });
```

Ожидание завершения конкретной задачи осуществляется с помощью метода **Wait**.

Методы ожидания **WaitAll** и **WaitAny** могут принимать в качестве аргументов, как массив задач (один параметр), так и сами задачи (произвольное число параметров).

Вызов **WaitAll** блокирует текущий поток до завершения всех указанных задач.

Вызов **WaitAny** блокирует текущий поток до завершения хотя бы одной из указанных задач и возвращает номер первой завершенной задачи.

Пример.

```
using System;
using System.Threading.Tasks;
class HelloWorld
{
    static void Hello()
    {
        Console.WriteLine("Hello!");
    }
    static void Main()
    {
        Task t1 = Task.Factory.StartNew(Hello);
        Task t2 = Task.Factory.StartNew(Hello);
        Task t3 = Task.Factory.StartNew(Hello);
    }
}
```

```
// Дожидаемся завершения хотя бы одной задачи
int firstTask = Task.WaitAny(t1, t2, t3);
Console.WriteLine("завершение задачи "+firstTask);
// Дожидаемся завершения всех задач
Task.WaitAll(t1, t2, t3);
}
}
```

```
Hello!
Hello!
Hello!
завершение задачи 2
```

```
Hello!
завершение задачи 1
Hello!
Hello!
```

```
Hello!
Hello!
завершение задачи 2
Hello!
```

```
Hello!
Hello!
Hello!
завершение задачи 0
```

Работа с данными в задаче

В задаче можно оперировать всеми переменными, находящимися в области видимости. Если рабочий элемент это *метод класса*, то работать можно с переменными этого класса. Если рабочим элементом является *лямбда-выражение*, то работать можно со всеми локальными переменными метода, порождающего задачу.

Пример.

```
using System;
using System.Threading;
using System.Threading.Tasks;
class HelloWorld
{
    static string prName;
    static void Info(object taskName)
    {
        Console.WriteLine("Task name: "+taskName);
        Console.WriteLine("Task ID: "+Task.CurrentId);
        Console.WriteLine("Thread id: " +
Thread.CurrentThread.ManagedThreadId);
        Console.WriteLine("Program name: "+prName);
        Console.WriteLine("-----");
    }
}
```

```
static void Main()
{
    prName = "Working with data";
    Task t1 = Task.Factory.StartNew(new Action
<object>(Info), "First worker");
    Task t2 = Task.Factory.StartNew(o => Info(o),
"Second worker");
    string t3Name = "Third worker";
    Task t3 = Task.Factory.StartNew(() =>
Info(t3Name));
    Task.WaitAll(t1, t2, t3);
}
}
```



```
Task name: Second worker
Task ID: 1
Thread id: 5
Program name: Working with data
-----
Task name: First worker
Task ID: 2
Thread id: 4
Program name: Working with data
-----
Task name: Third worker
Task ID: 3
Thread id: 6
Program name: Working with data
-----
```

```
Task name: First worker
Task name: Third worker
Task name: Second worker
Task ID: 1
Task ID: 2
Thread id: 6
Program name: Working with data
Thread id: 4
Program name: Working with data
Task ID: 3
Thread id: 5
Program name: Working with data
-----
-----
-----
```

```
Task name: First worker
Task name: Second worker
Task name: Third worker
Task ID: 3
Task ID: 1
Thread id: 4
Program name: Working with data
-----
Task ID: 2
Thread id: 5
Program name: Working with data
-----
Thread id: 6
Program name: Working with data
-----
```

Для получения результата работы задачи существует специальный тип `Task<T>`.

Свойство `Result` содержит результат задачи. Обращение к свойству блокирует поток до завершения задачи.

Пример. Описать задачу, возвращающую квадратный корень из числа.

```
using System;
using System.Threading.Tasks;
class Program
{
    static void Main()
    {
        long N = 256;
    }
}
```

```
Task <double> Task1 = Task.Factory.StartNew((obj) =>  
{ return Math.Sqrt((long)obj); }, N);  
    // Дожидаемся завершения вычислений  
    // без явной блокировки  
    double R = Task1.Result;  
    Console.WriteLine($"Sqrt({N}) = {R}");  
}
```

```
Sqrt(256) = 16
```

Вложенные задачи

В коде задачи можно запускать вложенные задачи, которые могут быть *дочерними* и *недочерними*.

Недочерние задачи обладают независимостью от родительской задачи: родитель не дожидается завершения вложенной задачи, статусы задач не взаимосвязаны.

Дочерняя задача действительно является вложенной – родитель дожидается завершения дочерней задачи, статусы задач при исключениях взаимосвязаны.

Пример. Создать задачу, в которой запустить две вложенные задачи (не дочернюю и дочернюю), выводящие сообщения о своем запуске и завершении.

```
using System;
using System.Threading;
using System.Threading.Tasks;
class Program
{
    static void Main()
    {
        Task Parent = Task.Factory.StartNew( () =>
        {
            Console.WriteLine("Родительская задача запущена");
            Task t1 = Task.Factory.StartNew( () =>
            Console.WriteLine("Вложенная задача запущена"));
            Thread.SpinWait(500);    // задержка
            Console.WriteLine("Вложенная задача выполнена");
        });
    }
}
```

```
        Task t2 = Task.Factory.StartNew( () =>
Console.WriteLine("Дочерняя задача запущена"),
TaskCreationOptions.AttachedToParent);
        Thread.SpinWait(50000);
        Console.WriteLine("Дочерняя задача выполнена");
        Console.WriteLine("Родительская задача завершена");
    });
    Parent.Wait();
    Console.WriteLine("Родительская задача действительно завершена");
}
}
```

Вывод результатов определяется загруженностью системы и пула потоков.

Вызов **Parent.Wait()** завершится только после вывода вложенной дочерней задачи. Вывод вложенной не дочерней задачи может быть и после завершения родительской

Задачи

Родительская задача запущена
Вложенная задача выполнена
Дочерняя задача выполнена
Родительская задача завершена
Дочерняя задача запущена
Родительская задача действительно завершена
Вложенная задача запущена

Родительская задача запущена
Вложенная задача выполнена
Вложенная задача запущена
Дочерняя задача выполнена
Родительская задача завершена
Дочерняя задача запущена
Родительская задача действительно завершена

Родительская задача запущена
Вложенная задача выполнена
Дочерняя задача выполнена
Родительская задача завершена
Дочерняя задача запущена
Вложенная задача запущена
Родительская задача действительно завершена

Родительская задача запущена
Вложенная задача запущена
Вложенная задача выполнена
Дочерняя задача запущена
Дочерняя задача выполнена
Родительская задача завершена
Родительская задача действительно завершена

Ожидание задач

Два способа ожидания выполнения задач:

- С помощью метода `Wait()`.

`public Wait(TimeSpan timeout)`

где `timeout` – время ожидания в миллисекундах.

- С помощью обращения к свойству `Result` (свойство класса `Task<TResult>`):

`public TResult Result;`

Ожидание завершения одновременно нескольких задач :

`public static void WaitAll(Task[]);` – ожидает завершения всех указанных задач

`public static void WaitAny(Task[]);` – ожидает завершения какой-либо задачи

где `Task[]` – массив экземпляров класса `Task`.

Пример. В предыдущем примере

Родительская задача запущена
Вложенная задача выполнена
Дочерняя задача запущена
Дочерняя задача выполнена
Родительская задача завершена
Вложенная задача запущена
Родительская задача действительно завершена

Родительская задача запущена
Вложенная задача выполнена
Вложенная задача запущена
Дочерняя задача выполнена
Родительская задача завершена
Дочерняя задача запущена
Родительская задача действительно завершена

задачи `t1.Wait();`

задачи `t2.Wait();`

Родительская задача запущена
Вложенная задача выполнена
Вложенная задача запущена
Дочерняя задача запущена
Дочерняя задача выполнена
Родительская задача завершена
Родительская задача действительно завершена

Обработка ошибок в задачах

При ожидании завершения задачи с помощью метода `Wait()`, или свойства `Result`, любое необработанное *исключение* будет передано вызывающему коду, а именно объекту `AggregationException`, что делает не обязательным обработку исключений внутри самой задачи:

Но все же необходимо обрабатывать исключения автономных задач (т.е задач, которые не являются родительскими для других задач, и завершения, выполнения которых никто не ожидает).

Пример. Обработать исключение деления на 0 при вычислении выражения

`x = 0`

`Attempted to divide by zero.`

`x = 5`

`10/5=2`

...

```
static void Main()
{
    Console.Write ("x = ");
    int x =Convert.ToInt32(Console.ReadLine());
    Task<int> z = Task.Factory.StartNew (() => 10 / x);
    try
    {
        if (x!=0)
            Console.WriteLine ($"10/{x}="+z.Result);
        else
            Console.WriteLine (z.Result);
    }
    catch (AggregateException aex)
    {
        Console.Write (aex.InnerException.Message);    }
}
```

Отмена выполнения задач

Отмена выполнения заданий осуществляется с помощью специальных маркеров отмены (*token*).

Структура **CancellationToken** – распространяет уведомление о том, что операции следует отменить. При запуске задачи можно передать маркер отмены, что позволит отменить выполнение задачи.

Пусть источник ресурса для маркера отмены **S**, задача **Z**. Тогда для отмены задачи используется следующий код:

```
CancellationTokenSource S= new CancellationTokenSource();
CancellationToken token = S.Token;
Task Z=Task.Factory.StartNew (() =>
{ token.ThrowIfCancellationRequested(); // Проверка отмены
}, token);
```

...

```
S.Cancel();
```

Для отмененной задачи вызывается *исключение* – **AggregateException**, которое нужно обработать :

```
try
{ Z.Wait();
}
catch (AggregateException ex)
{ if (ex.InnerException is OperationCanceledException)
    Console.Write ("Задание отменено");
}
```

Пример

```
using System;
using System.Threading.Tasks;
class Program
{ static void Main()
    { CancellationTokenSource Source = new
        CancellationTokenSource();
        CancellationToken token = Source.Token;
        Task task = Task.Factory.StartNew (() =>
        { token.ThrowIfCancellationRequested(); }, token);
        Console.WriteLine ("---");
        Console.WriteLine ("+++");
        Source.Cancel();
    }
```

```
try
{
    task.Wait();
}
catch (AggregateException ex)
{
    if (ex.InnerException is OperationCanceledException)
    {
        Console.WriteLine ("Задание отменено");
    }
}
}
```

```
---
+++
Задание отменено
```

```
---
+++
```

Задачи-продолжения

Для того что бы упорядочить выполнение задач в программе, т.е. к примеру чтобы Задача 3 выполнялась сразу же после выполнения Задачи 2, а Задача 2 после завершения Задачи 1, необходимо использовать метод `ContinueWith()` класса `Task`:

```
public Task ContinueWith(Action<Task>)
```

Пример

```
Task t1 = Task.Factory.StartNew (() => Console.Write ("Hello "));  
Task t2 = t1.ContinueWith (main => Console.Write ("World!"));  
t2.Wait();
```

где `main` – аргумент, который передается в задачу-продолжение в лямбда-выражении, является ссылкой на родительскую задачу.

```
Hello World!
```


В классе `Task<TResult>`, также можно использовать продолжение и возвращать некоторые данные, с использованием последовательности вычисления.

Пример

```
Task t1 =Task.Factory.StartNew<int>(() => 10)
    .ContinueWith(main => main.Result + 6)
    .ContinueWith(main => Math.Sqrt(main.Result))
    .ContinueWith(main
=>Console.WriteLine(main.Result));
t1.Wait();
```

Продолжения предыдущих задач

Метод `ContinueWhenAll()` используется для того, чтобы выполнить продолжение конкретной задачи, после выполнения нескольких предыдущих задач.

Пример. Запустить 2 задачи, получающих с помощью лямбда-выражения числа 200 и 100, соответственно. Выполнить продолжение задачи, вычисляющей сумму двух чисел, взятых из первых двух задач.

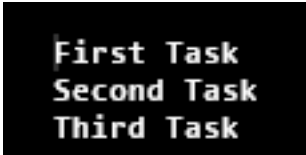
```
using System;
using System.Threading.Tasks;
using System.Linq;
class Program
{
    static void Main()
    {
        Task<int> task1 = Task.Factory.StartNew
            (() => 200);
        Task<int> task2 = Task.Factory.StartNew
            (() => 100);
        Task<int> task3 = Task<int>.Factory.ContinueWhenAll
            (new[] { task1, task2 }, tasks => tasks.Sum (t => t.Result));
        Console.WriteLine ("sum="+task3.Result);
    }
}
```

sum=300

Массив задач

Можно определить все задачи в массиве непосредственно через объект Task:

```
Task[] t1 = 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 t1)
    t.Start();
// завершение задач в массиве
foreach (var t in t1)
    t.Wait();
```



```
First Task
Second Task
Third Task
```

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

```
Task[] t2 = new Task[3];
int j = 1;
for (int i = 0; i < t2.Length; i++)
    t2[i] = Task.Factory.StartNew(() =>
        Console.WriteLine($"Task {j++}"));
foreach (var t in t2)
    t.Wait();
// или Task.WaitAll(t2);
```

```
Task 1
Task 3
Task 2
```