

# Работа с задачами

---

Задачи (tasks) являются основным строительным блоком библиотеки Task Parallel Library. Задачи представляют собой рабочие элементы, которые могут выполняться параллельно. В качестве рабочих элементов могут использоваться: методы, делегаты, лямбда-выражения.

```
static void HelloWorld()
{
    Console.WriteLine("Hello, world!");
}
static void Main()
{
    // Используем обычный метод
    Task t1 = new Task(HelloWorld);
    // Используем делегат Action
    Task t2 = new Task(new Action(HelloWorld));
    // Используем безымянный делегат
    Task t3 = new Task(delegate
    {
        HelloWorld();
    });
    // Используем лямбда-выражение
    Task t4 = new Task(() => HelloWorld());
    // Используем лямбда-выражение
    Task t5 = new Task(() =>
    {
        HelloWorld();
    });
    Task t6 = new Task(() =>
    {
        Console.WriteLine("Hello, world!");
    });

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

Работа с задачами, как правило, включает три основные операции: объявление задачи, добавление задачи в очередь готовых задач, ожидание завершения выполнения задачи.

Работа с потоками:

```
Thread threadOne = new Thread(SomeWork);
threadOne.Start();
```

```
threadOne.Join();
```

Работа с задачами:

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

Важное отличие заключается в том, что 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.

Для более лаконичного «запуска» задачи существует шаблон, замещающий этапы объявления и добавления задачи в пул потоков:

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

Ожидание завершения конкретной задачи осуществляется с помощью метода `Wait`. Для ожидания завершения нескольких задач существуют статические методы класса `Task`, принимающие в качестве аргумента массив задач:

```
Task t1 = Task.Factory.StartNew(DoWork1);  
Task t2 = Task.Factory.StartNew(DoWork2);  
Task t3 = Task.Factory.StartNew(DoWork3);  
// Дожидаемся завершения хотя бы одной задачи  
int firstTask = Task.WaitAny(t1, t2, t3);  
// Дожидаемся завершения всех задач  
Task.WaitAll(t1, t2, t3);
```

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.

## Статусы задачи

Задача может находиться в нескольких состояниях: `Created`, `Running`, `WaitingToRun`, `Faulted`, `Canceled`, `RanToCompletion`, `WaitingForActivation`, `WaitingForChildrenToComplete`. Переходы между основными состояниями изображены на рисунке.



```

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

```

Для получения результата работы задачи существует специальный тип `Task<T>`. Свойство `Result` содержит результат задачи. Обращение к свойству блокирует поток до завершения задачи.

```

static void Main()
{
    long lNumber = 123456789;
    Task<double> SqrtTask = Task.Factory.StartNew((obj) =>
    {
        return Math.Sqrt((long) obj);
    }, lNumber);
    // Дожидаемся завершения вычислений без явной блокировки
    double sqrtNumber = SqrtTask.Result;
    Console.WriteLine("Result: {0}", sqrtNumber);
}

```

При объявлении задачи или при использовании шаблона `StartNew` можно указать ряд важных параметров:

- токен отмены (`CancellationToken`), с помощью которого можно реализовать «согласованную» отмену задачи;
- опции задачи (`TaskCreationOptions`), позволяющие скорректировать действия планировщика при обработке задачи;
- параметр задачи - аргумент метода, ассоциированного с задачей;
- планировщик, который будет использоваться для обработки задачи.

## Вложенные задачи

В коде задачи можно запускать вложенные задачи, которые могут быть дочерними и недочерными. Недочерные 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.

```
Task tParent = Task.Factory.StartNew( () =>
```

```

        {
            Console.WriteLine("Parent task starts");
            Task t1 = Task.Factory.StartNew( () =>
                Console.WriteLine("Inner task"));
            Task t2 = Task.Factory.StartNew( () =>
                Console.WriteLine("Child task"),
                TaskCreationOptions.AttachedToParent);
            Console.WriteLine("Parent task ends");
        });
        tParent.Wait();
        Console.WriteLine("Parent task really ends");

```

Вывод приведенного фрагмента определяется загруженностью системы и пула потоков. Тем не менее, можно быть уверенным, что вызов `tParent.Wait()` завершится только после вывода вложенной дочерней задачи. Вывод вложенной недочерней задачи может быть и после завершения родительской задачи.

```

Parent task starts
Parent task ends
Child task
Parent task really ends
Inner task

```

## Механизм отмены задач

Встроенный механизм согласованной отмены задач позволяет унифицированным образом реализовывать корректное досрочное завершение выполнения задач.

Для реализации механизма отмены необходимо выполнить следующие шаги:

1. Создать объект `CancellationTokenSource` в области видимости метода, который порождает и запускает задачу.
2. Получить объект `CancellationToken`, через который осуществляется взаимодействие с задачей.
3. Передать объект `CancellationToken` при запуске задачи.
4. Реализовать в задаче процедуру отмены
5. При необходимости отмены вызвать метод `Cancel()`.

Обработчик отмены задачи можно реализовать либо в самой задаче, либо назначить отдельный делегат, который будет вызываться при возникновении сигнала об отмене задачи.

```

static void Main()
{
    var cts = new CancellationTokenSource();
    var token = cts.Token;
    Task t1 = new Task ( () =>
        {
            while (true)

```

```

        {
            DoSomeWork();
            if(token.IsCancellationRequested)
            {
                // Код обработки отмены
                SaveSomeData();
                break;
            }
        }
    }, token);

    t1.Start();
    Thread.Sleep(1000);
    cts.Cancel();
    t1.Wait();

    var cts2 = new CancellationTokenSource();
    var token2 = cts2.Token;
    // Регистрируем обработчик отмены
    token2.Register(() => {
        Console.WriteLine("Task #2 was cancelled");
        throw new TaskCanceledException();
    });
    Task t2 = new Task(() => {
        while(true)
            DoSomeWork();
    }, token2);

    t2.Start();
    Thread.Sleep(1000);
    cts2.Cancel();
    t2.Wait();
}

```

В первой задаче сигнал отмены контролируется в рабочем цикле с помощью свойства `IsCancellationRequested`. Во второй задаче отмена обрабатывается с помощью делегата, который регистрируется методом `Register`.

## Исключения в задачах

Исключения, которые могут возникнуть при выполнении задач, обрабатываются в коде метода, который объявляет, запускает и ожидает завершения задач.

```

void SomeMethod()
{
    Task t1 = Task.Factory.StartNew(WorkOne);
    Task t2 = Task.Factory.StartNew(WorkTwo);
    try
    {
        Task.WaitAll(t1, t2);
    }
    catch (AggregateException ae)
    {
    }
}

```

```
{  
    // Обработка ошибок  
}  
}
```

Исключения могут возникнуть в нескольких задачах. Поэтому для обработки исключений параллельного кода используется объект типа `AggregateException`, который агрегирует все возникнувшие исключения. Список единичных исключений можно получить с помощью свойства `InnerExceptions`.

```
catch(AggregateException ae)  
{  
    foreach(Exception exc in ae.InnerExceptions)  
        Console.WriteLine(exc.Message);  
}
```

Если у задачи есть вложенные дочерние задачи, то объект `exc` представляет собой опять тип `AggregateException` и для обработки исключений дочерних задач необходимо во вложенном цикле обрабатывать элементы `exc.InnerExceptions`.

Метод `Flatten()` объекта `AggregateException` возвращает все исключения, возникнувшие в задачах и вложенных задачах в одном списке, делая более удобной обработку.

```
Task[] tasks = new Task[N];  
// объявляем и запускаем задачи  
..  
// Обработка исключений  
try {  
    Task.WaitAll(tasks);  
}  
catch(AggregateException ae)  
{  
    foreach(Exception e in ae.Flatten().InnerExceptions)  
        Console.WriteLine("Message:{0}", e.Message);  
}
```

Метод `Handle` позволяет назначить делегаты-обработчики для конкретных исключений:

```
Task t1 = new Task(() => {  
    throw new OutOfMemoryException(); });  
Task t2 = new Task(() => {  
    throw new DivideByZeroException(); });  
t1.Start(); t2.Start();  
try {  
    Task.WaitAll(t1, t2);  
} catch (AggregateException ae) {  
    ae.Handle((inner) => {  
        if (inner is OperationCancelledException) {
```

```

        // обработчик исключения
        return true;
    } else if(inner is DivideByZeroException) {
        // обработчик исключения
        return true;
    } else
        return false;
    });
}

```

Делегат, определяемый в методе `Handle`, возвращает `true`, если исключение обработано, и `false`, если исключение не обработано.

## Задачи-продолжения

Задачи-продолжения предназначены для планирования запуска задач после завершения предшествующих задач с тем или иным статусом завершения: `OnlyOnRanToCompletion`, `OnlyOnCanceled`, `OnlyOnFaulted`, `NotOnCancelled`, `NotOnRanToCompletion`.

```

// Основная задача, выполняющая расчёт
Task t1 = Task<int>.Factory.StartNew(() => FindDecision());
// Вывод результатов в отдельной задаче
Task t2 = t1.ContinueWith((prev) =>
    Console.WriteLine("Result: {0}", prev.Result),
    TaskContinuationOptions.OnlyOnRanToCompletion);
// Обработчик ошибок
Task t3 = t1.ContinueWith((prev) =>
    Console.WriteLine("Error: {0}",
        prev.Exception.InnerException.Message),
    TaskContinuationOptions.OnlyOnFaulted);
// Задача была отменена
Task t4 = t1.ContinueWith((prev) =>
    Console.WriteLine("Task was cancelled"),
    TaskContinuationOptions.OnlyOnCanceled);

```

Первая задача осуществляет основной расчет. Следующие задачи выполняются в зависимости от статуса завершения первой задачи. Вторая задача выполняется при успешном завершении. Третья задача выполняется при возникновении необработанного исключения. Четвертая задача выполняется только в случае отмены первой.

Задачи-продолжения позволяют без дополнительных средств синхронизации реализовать критическую секцию и конструкцию барьера:

```

// Объявляем задачи, которые могут выполняться параллельно
Task[] tasks = new Task[3];
tasks[0] = new Task(Work1);
tasks[1] = new Task(Work2);
tasks[2] = new Task(Work3);
// Планируем выполнение критической секции

```



```
Task tCr = Task.Factory.ContinueWhenAll(tasks, (tt) => {  
    // Критическая секция  
});  
// Параллельные задачи  
Task t5 = tCr.ContinueWith(Work5);  
Task t6 = tCr.ContinueWith(Work6);  
// Запускаем задачи  
tasks[0].Start(); tasks[1].Start(); tasks[2].Start();  
// Ожидаем завершения последней задачи  
t6.Wait();
```