Параллельное программирование и библиотека TPL

Задачи и класс Task

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 упрощает работу с многопроцессорными, многоядерными системами. Кроме того, она упрощает работу по созданию новых потоков. Поэтому обычно рекомендуется использовать именно TPL и ее классы для создания многопоточных приложений, хотя стандартные средства и класс Thread по-прежнему находят широкое применение.

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:

```
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:

```
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"));
```

Итак, в данном коде задачи создаются и запускаются, но при выполнении приложения на консоли мы можем не увидеть ничего. Почему? Потому что когда поток задачи запускается из основного потока программы - потока метода Main, приложение может завершить выполнение до того, как все три или даже хотя бы одна из трех задач начнет выполнение. Чтобы этого не произошло, мы можем программным образом ожидать завершения задачи.

Ожидание завершения задачи

Чтобы приложение ожидало завершения задачи, можно использовать метод Wait() объекта Task:

```
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"));

task1.Wait();   // ожидаем завершения задачи task1
task2.Wait();   // ожидаем завершения задачи task2
task3.Wait();   // ожидаем завершения задачи task3
```

Возможный консольный вывод программы:

```
Task3 is executed
Task2 is executed
Task1 is executed
```

Консольный вывод не детерминирован, поскольку задачи не выполняются последовательно. Первая запущенная задача может завершить свое выполнение после последней задачи.

Стоит отметить, что метод Wait() блокирует вызывающий поток, в котором запущена задача, пока эта задача не завершит свое выполнение. Например:

```
Console.WriteLine("Main Starts");
// создаем задачу
Task task1 = new Task(() =>
{
    Console.WriteLine("Task Starts");
    Thread.Sleep(1000);     // задержка на 1 секунду - имитация долгой работы
    Console.WriteLine("Task Ends");
 });
task1.Start();  // запускаем задачу
task1.Wait();   // ожидаем выполнения задачи
Console.WriteLine("Main Ends");
```

Для эмуляции долговременной работы здесь в задаче task1 устанавливается задержка на 1 секунду. В итоге, когда выполнение дойдет до вызова task1.Wait() основной поток остановит свое выполнение и будет ждать завершения задачи. И мы получим следующий консольный вывод:

```
Main Starts
Task Starts
Task Ends
Main Ends
```

Если подобное поведение не принципиально, то ожидание завершения задачи можно поместить в конец метода Main:

```
Console.WriteLine("Main Starts");
// создаем задачу
Task task1 = new Task(() =>
{
    Console.WriteLine("Task Starts");
    Thread.Sleep(1000);     // задержка на 1 секунду - имитация долгой работы
    Console.WriteLine("Task Ends");
 });
task1.Start();  // запускаем задачу
Console.WriteLine("Main Ends");
task1.Wait();   // ожидаем выполнения задачи
```

В этом случае приложение все равно будет ждать завершения задачи, однако другие синхронные действия в основном потоке не будут блокироваться и ожидать завершения задачи.

Синхронный запуск задачи

По умолчанию задачи запускаются асинхронно. Однако с помощью метода RunSynchronously() можно запускать синхронно:

```
Console.WriteLine("Main Starts");
// создаем задачу
Task task1 = new Task(() =>
{
    Console.WriteLine("Task Starts");
    Thread.Sleep(1000);
    Console.WriteLine("Task Ends");
 });
task1.RunSynchronously(); // запускаем задачу синхронно
Console.WriteLine("Main Ends"); // этот вызов ждет завершения задачи task1
```

Свойства класса Task

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

- AsyncState: возвращает объект состояния задачи
- CurrentId: возвращает идентификатор текущей задачи (статическое свойство)
- Id: возвращает идентификатор текущей задачи
- Exception: возвращает объект исключения, возникшего при выполнении задачи
- Status: возвращает статус задачи. Представляет перечисление System.Threading.Tasks.TaskStatus, которое имеет следующие значения:

  Canceled: задача отменена

  Created: задача создана, но еще не запущена

  Faulted: в процессе работы задачи произошло исключение

  RanToCompletion: задача успешно завершена

  Running: задача запущена, но еще не завершена

  WaitingForActivation: задача ожидает активации и постановки в график выполнения

  WaitingForChildrenToComplete: задача завершена и теперь ожидает завершения прикрепленных к ней дочерних задач

  WaitingToRun: задача поставлена в график выполнения, но еще не начала свое выполнение
- IsCompleted: возвращает true, если задача завершена
- IsCanceled: возвращает true, если задача была отменена

- IsFaulted: возвращает true, если задача завершилась при возникновении исключения
- IsCompletedSuccessfully: возвращает true, если задача завершилась успешно

Используем некоторые из этих свойств:

```
Task task1 = new Task(() =>
{
    Console.WriteLine($"Task{Task.CurrentId} Starts");
    Thread.Sleep(1000);
    Console.WriteLine($"Task{Task.CurrentId} Ends");
 });
task1.Start(); //запускаем задачу

// получаем информацию о задаче
Console.WriteLine($"task1 Id: {task1.Id}");
Console.WriteLine($"task1 is Completed: {task1.IsCompleted}");
Console.WriteLine($"task1 Status: {task1.Status}");

task1.Wait(); // ожидаем завершения задачи
```

Пример консольного вывода:

```
task1 Id: 1
Task1 Starts
task1 is Completed: False
task1 Status: Running
Task1 Ends
```

Работа с классом Task

Вложенные задачи

Одна задача может запускать другую - вложенную задачу. При этом эти задачи выполняются независимо друг от друга. Например:

```
var outer = Task.Factory.StartNew(() =>      // внешняя задача
{
    Console.WriteLine("Outer task starting...");
    var inner = Task.Factory.StartNew(() =>  // вложенная задача
    {
        Console.WriteLine("Inner task starting...");
        Thread.Sleep(2000);
        Console.WriteLine("Inner task finished.");
    });
});
```

```
outer.Wait(); // ожидаем выполнения внешней задачи
Console.WriteLine("End of Main");
```

Несмотря на то, что здесь мы ожидаем выполнения внешней задачи, но вложенная задача может завершить выполнение даже после завершения метода Main:

```
Outer task starting...
End of Main
```

При этом внутренняя задача может даже не начать свое выполнение к завершению работы основного потока программы. То есть в данном случае внешняя и вложенная задачи выполняются независимо друг от друга.

Если необходимо, чтобы вложенная задача выполнялась как часть внешней, необходимо использовать значение TaskCreationOptions.AttachedToParent:

```
var outer = Task.Factory.StartNew(() =>      // внешняя задача
{
    Console.WriteLine("Outer task starting...");
    var inner = Task.Factory.StartNew(() =>  // вложенная задача
    {
        Console.WriteLine("Inner task starting...");
        Thread.Sleep(2000);
        Console.WriteLine("Inner task finished.");
    }, TaskCreationOptions.AttachedToParent);
});
outer.Wait(); // ожидаем выполнения внешней задачи
Console.WriteLine("End of Main");
```

Консольный вывод:

```
Outer task starting...
Inner task starting...
Inner task finished.
End of Main
```

В данном случае вложенная задача прикреплена к внешней и выполняется как часть внешней задачи. И внешняя задача завершится только когда завершатся все прикрепленные к ней вложенные задачи.

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:

```
Task[] tasks = new Task[3];
for(var i = 0; i < tasks.Length; i++)
{
    tasks[i] = new Task(() =>
    {
        Thread.Sleep(1000); // эмуляция долгой работы
        Console.WriteLine($"Task{i} finished");
    });
    tasks[i].Start();   // запускаем задачу
}
Console.WriteLine("Завершение метода Main");
```

Один из возможных консольных выводов программы:

```
Завершение метода Main
```

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

```
Task[] tasks = new Task[3];
for(var i = 0; i < tasks.Length; i++)
{
    tasks[i] = new Task(() =>
```

```
    {
        Thread.Sleep(1000); // эмуляция долгой работы
        Console.WriteLine($"Task{i} finished");
    });
    tasks[i].Start();   // запускаем задачу
}
Console.WriteLine("Завершение метода Main");

Task.WaitAll(tasks); // ожидаем завершения всех задач
```

В этом случае сначала завершатся все задачи, и лишь только потом будет выполняться последующий код из метода Main:

```
Завершение метода Main
Task3 finished
Task3 finished
Task3 finished
```

В то же время порядок выполнения самих задач в массиве также недетерминирован.

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

Возвращение результатов из задач

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

```
int n1 =4, n2 = 5;
Task<int> sumTask = new Task<int>(() => Sum(n1, n2));
sumTask.Start();

int result = sumTask.Result;
Console.WriteLine($"{n1} + {n2} = {result}"); // 4 + 5 = 9

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

Во-первых, чтобы получать из задачи не который результат, необходимо типизировать объект Task тем типом, объект которого мы хотим получить из задачи. Например, в примере выше мы ожидаем из задачи sumTask получить число типа int, соответственно типизируем объект Task данным типом - Task.

И, во-вторых, в качестве задачи должен выполняться метод, который возвращает данный тип объекта. Так, в данном случае у нас в качестве задачи выполняется метод Sum, которая принимаетдва числа и на выходе возвращает их сумму - значение типа int.

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

```
int result = sumTask.Result;
```

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

Другой пример:

```
Task<Person> defaultPersonTask = new Task<Person>(() => new Person("Tom", 37));
defaultPersonTask.Start();

Person defaultPerson = defaultPersonTask.Result;
Console.WriteLine($"{defaultPerson.Name} - {defaultPerson.Age}"); // Tom - 37

record class Person(string Name, int Age);
```

В данном случае задача defaultPersonTask возвращает объект типа Person, который мы можем получить из свойства Result.

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:

```
Task task1 = new Task(() =>
{
    Console.WriteLine($"Id задачи: {Task.CurrentId}");
});

// задача продолжения - task2 выполняется после task1
Task task2 = task1.ContinueWith(PrintTask);

task1.Start();

// ждем окончания второй задачи
task2.Wait();
Console.WriteLine("Конец метода Main");


void PrintTask(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. То есть метод PrintTask, который передается в вызов ContinueWith, 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. Консольный вывод программы:

```
Id задачи: 1
Id задачи: 2
Id предыдущей задачи: 1
Конец метода Main
```

Также мы можем передавать конкретный результат работы предыдущей задачи:

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

// задача продолжения
Task printTask = sumTask.ContinueWith(task => PrintResult(task.Result));

sumTask.Start();

// ждем окончания второй задачи
printTask.Wait();
Console.WriteLine("Конец метода Main");

int Sum(int a, int b) => a + b;
void PrintResult(int sum) => Console.WriteLine($"Sum: {sum}");
```

В данном случае задача sumTask выполняет метод Sum и возвращает его результат. Задача printTask является задачей продолжения, выполняется сразу после sumTask и получает ее результат. Так, в вызове

```
Task printTask = sumTask.ContinueWith(task => PrintResult(task.Result));
```

Параметр task в лямбда-выражении фактически представляет задачу sumTask, из которой извлекается результат.

Подобным образом можно построить целую цепочку последовательно выполняющихся задач:

```
Task task1 = new Task(() => Console.WriteLine($"Current Task: {Task.CurrentId}"));

// задача продолжения
Task task2 = task1.ContinueWith(t =>
    Console.WriteLine($"Current Task: {Task.CurrentId}  Previous Task: {t.Id}"));

Task task3 = task2.ContinueWith(t =>
    Console.WriteLine($"Current Task: {Task.CurrentId}  Previous Task: {t.Id}"));


Task task4 = task3.ContinueWith(t =>
    Console.WriteLine($"Current Task: {Task.CurrentId}  Previous Task: {t.Id}"));

task1.Start();

task4.Wait();   //  ждем завершения последней задачи
Console.WriteLine("Конец метода Main");
```

здесь друг за другом выполняются задачи task1, task2, task3, task4. Консольный вывод программы:

```
Current Task: 1
Current Task: 2  Previous Task: 1
Current Task: 3  Previous Task: 2
Current Task: 4  Previous Task: 3
Конец метода Main
```

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:

```
// метод Parallel.Invoke выполняет три метода
Parallel.Invoke(
    Print,
    () =>
    {
        Console.WriteLine($"Выполняется задача {Task.CurrentId}");
        Thread.Sleep(3000);
    },
    () => Square(5)
);

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

```
    Thread.Sleep(3000);
}
// вычисляем квадрат числа
void Square(int n)
{
    Console.WriteLine($"Выполняется задача {Task.CurrentId}");
    Thread.Sleep(3000);
    Console.WriteLine($"Результат {n * n}");
}
```

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. Пример консольного вывода программы:

```
Выполняется задача 1
Выполняется задача 3
Выполняется задача 2
Результат 25
```

Parallel.For

Метод Parallel.For позволяет выполнять итерации цикла параллельно. Он имеет следующее определение:

```
For(int, int, Action<int>)
```

Первый параметр метода 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:

```
Parallel.For(1, 5, Square);

// вычисляем квадрат числа
void Square(int n)
{
    Console.WriteLine($"Выполняется задача {Task.CurrentId}");
    Console.WriteLine($"Квадрат числа {n} равен {n * n}");
    Thread.Sleep(2000);
}
```

В данном случае в качестве первого параметра в метод Parallel.For передается число 1, а в качестве второго - число 5. Таким образом, метод будет вести итерацию с 1 до 4 включительно. Третий параметр представляет метод, который вычисляет квадрат числа. Так как этот параметр представляет тип Action, 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 4 включительно. И метод Square, таким образом, вызывается 4 раза. Пример консольного вывода:

```
Выполняется задача 1
Выполняется задача 2
Квадрат числа 4 равен 16
Выполняется задача 4
Квадрат числа 1 равен 1
Выполняется задача 3
Квадрат числа 3 равен 9
Квадрат числа 2 равен 4
```

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.

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

// вычисляем квадрат числа
void Square(int n)
{
    Console.WriteLine($"Выполняется задача {Task.CurrentId}");
    Console.WriteLine($"Квадрат числа {n} равен {n * n}");
    Thread.Sleep(2000);
}
```

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. Пример консольного вывода:

```
Выполняется задача 1
Выполняется задача 3
Квадрат числа 8 равен 64
Выполняется задача 4
Квадрат числа 3 равен 9
Выполняется задача 2
Квадрат числа 5 равен 25
Квадрат числа 1 равен 1
```

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:

```
ParallelLoopResult result = Parallel.For(1, 10, Square);
if (!result.IsCompleted)
    Console.WriteLine($"Выполнение цикла завершено на итерации
{result.LowestBreakIteration}");

// вычисляем квадрат числа
void Square(int n, ParallelLoopState pls)
{
    if (n == 5) pls.Break();    // если передано 5, выходим из цикла

    Console.WriteLine($"Квадрат числа {n} равен {n * n}");
    Thread.Sleep(2000);
}
```

Здесь метод Square, который обрабатывает 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 платформа .NET предоставляет структуру CancellationToken из пространства имен System.Threading.

Общий алгоритм отмены задачи обычно предусматривает следующий порядок действий:

1. Создание объекта CancellationTokenSource, который управляет и посылает уведомление об отмене токену.

2. С помощью свойства CancellationTokenSource.Token получаем собственно токен - объект структуры CancellationToken и передаем его в задачу, которая может быть отменена.

```
CancellationTokenSource cancelTokenSource = new CancellationTokenSource();
CancellationToken token = cancelTokenSource.Token;
```

Для передачи токена в задачу можно применять один из конструкторов класса Task:

```
CancellationTokenSource cancelTokenSource = new CancellationTokenSource();
CancellationToken token = cancelTokenSource.Token;
Task task = new Task(() => { выполняемые_действия}, token); //
```

3. Определяем в задаче действия на случай ее отмены. Вызываем метод CancellationTokenSource.Cancel(), который устанавливает для свойства CancellationToken.IsCancellationRequested значение true. Стоит понимать, что сам по себе метод CancellationTokenSource.Cancel() не отменяет задачу, он лишь посылает уведомление об отмене через установку свойства CancellationToken.IsCancellationRequested. Каким образом будет происходить выход из задачи, это решает сам разработчик.

4. Вызываем метод CancellationTokenSource.Cancel(), который устанавливает для свойства CancellationToken.IsCancellationRequested значение true. Стоит понимать, что сам по себе метод CancellationTokenSource.Cancel() не отменяет задачу, он лишь посылает уведомление об отмене через установку свойства CancellationToken.IsCancellationRequested. Каким образом будет происходить выход из задачи, это решает сам разработчик.

5. Класс CancellationTokenSource реализует интерфейс IDisposable. И когда работа с объектом CancellationTokenSource завершена, у него следует вызвать метод Dispose для освобождения всех связанных с ним используемых ресурсов. (Вместо явного вызова метода Dispose можно использовать конструкцию using).

Теперь касательно третьего пункта - определения действий отмены задачи. Как именно завершить задачу? Конкретные действия на лежат целиком на разработчике, тем не менее есть два общих варианта выхода:

- При получении сигнала отмены выйти из метода задачи, например, с помощью оператора return или построив логику метода соответствующим образом. Но следует учитывать, что в этом случае задача перейдет в состояние TaskStatus.RanToCompletion, а не в состояние TaskStatus.Canceled.
- При получении сигнала отмены сгенерировать исключение OperationCanceledException, вызвав у токена метод ThrowIfCancellationRequested(). После этого задача перейдет в состояние TaskStatus.Canceled.

Мягкий выход из задачи без исключения OperationCanceledException

Сначала рассмотрим первый - "мягкий" вариант завершения:

```
CancellationTokenSource cancelTokenSource = new CancellationTokenSource();
CancellationToken token = cancelTokenSource.Token;

// задача вычисляет квадраты чисел
Task task = new Task(() =>
{
    for (int i = 1; i < 10; i++)
    {
        if (token.IsCancellationRequested)  // проверяем наличие сигнала отмены
задачи
        {
            Console.WriteLine("Операция прервана");
            return;     //  выходим из метода и тем самым завершаем задачу
        }
        Console.WriteLine($"Квадрат числа {i} равен {i * i}");
        Thread.Sleep(200);
    }
}, token);
task.Start();

Thread.Sleep(1000);
// после задержки по времени отменяем выполнение задачи
cancelTokenSource.Cancel();
// ожидаем завершения задачи
Thread.Sleep(1000);
//  проверяем статус задачи
Console.WriteLine($"Task Status: {task.Status}");
cancelTokenSource.Dispose(); // освобождаем ресурсы
```

В данном случае задача task вычисляет и выводит на консоль квадраты чисел от 1 до 9. Для отмены задачи нам надо создать и использовать токен. Вначале создается объект CancellationTokenSource:

```
CancellationTokenSource cancelTokenSource = new CancellationTokenSource();
```

Затем из него получаем сам токен:

```
CancellationToken token = cancelTokenSource.Token;
```

Затем из него получаем сам токен:

```
CancellationToken token = cancelTokenSource.Token;
```

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

```
cancelTokenSource.Cancel();
```

В данном случае отмена задачи вызывается через секунду, чтобы задача произвела некоторые действия.

В самом методе задачи в цикле мы можем отловить сигнал отмены с помощью проверки свойства token.IsCancellationRequested:

```
if (token.IsCancellationRequested)
{
    Console.WriteLine("Операция прервана");
    return;
}
```

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

После завершения задачи проверяем ее статус:

```
Console.WriteLine($"Task Status: {task.Status}");
```

Поскольку задача успешно завершена, у задачи должен быть статус RanToCompletion

И в конце у объекта CancellationTokenSource вызываем метод Dispose:

```
cancelTokenSource.Dispose();
```

Консольный вывод программы:

```
Квадрат числа 1 равен 1
Квадрат числа 2 равен 4
Квадрат числа 3 равен 9
Квадрат числа 4 равен 16
Квадрат числа 5 равен 25
Операция прервана
Task Status: RanToCompletion
```

Отмена задачи с помощью генерации исключения

Второй способ завершения задачи представляет генерация исключения OperationCanceledException. Для этого применяется метод ThrowIfCancellationRequested() объекта CancellationToken:

```
CancellationTokenSource cancelTokenSource = new CancellationTokenSource();
CancellationToken token = cancelTokenSource.Token;

Task task = new Task(() =>
{
    for (int i = 1; i < 10; i++)
    {
        if (token.IsCancellationRequested)
            token.ThrowIfCancellationRequested(); // генерируем исключение

        Console.WriteLine($"Квадрат числа {i} равен {i * i}");
        Thread.Sleep(200);
    }
}, token);
try
{
    task.Start();
    Thread.Sleep(1000);
    // после задержки по времени отменяем выполнение задачи
    cancelTokenSource.Cancel();

    task.Wait(); // ожидаем завершения задачи
}
catch (AggregateException ae)
{
    foreach (Exception e in ae.InnerExceptions)
    {
        if (e is TaskCanceledException)
            Console.WriteLine("Операция прервана");
        else
            Console.WriteLine(e.Message);
    }
}
finally
{
    cancelTokenSource.Dispose();
```

```
}

//  проверяем статус задачи
Console.WriteLine($"Task Status: {task.Status}");
```

Здесь опять же проверяем значение свойства IsCancellationRequested, и если оно равно true, генерируем исключение:

```
if (token.IsCancellationRequested)
    token.ThrowIfCancellationRequested(); // генерируем исключение
```

Чтобы обработать исключение, помещаем весь код работы с задачей в конструкцию try..catch и также с помощью вызова cancelTokenSource.Cancel() посылаем сообщение об отмене задачи.

Стоит отметить, что генерируемое исключение будет спрятано в объекте AggregateException, который по сути представляет набор исключений. Если причина исключения состояла в отмене задачи, то мы можем найти в этом наборе исключений исключение типа TaskCanceledException

```
catch (AggregateException ae)
{
    foreach (Exception e in ae.InnerExceptions)
    {
        if (e is TaskCanceledException)
            Console.WriteLine("Операция прервана");
        else
            Console.WriteLine(e.Message);
    }
}
```

Класс TaskCanceledException является производным от OperationCanceledException. Исключение типа TaskCanceledException возникает, если для задачи устанавливается статус Canceled.

Консольный вывод программы:

```
Квадрат числа 1 равен 1
Квадрат числа 2 равен 4
Квадрат числа 3 равен 9
Квадрат числа 4 равен 16
Квадрат числа 5 равен 25
Операция прервана
Task Status: Canceled
```

Стоит отметить, что исключение возникает только тогда, когда мы останавливаем текущий поток и ожидаем завершения задачи с помощью методов Wait или WaitAll. Если эти методы не используются

для ожидания задачи, то для нее просто устанавливается состояние Canceled. Например, в следующем случае исключение не возникнет:

```
CancellationTokenSource cancelTokenSource = new CancellationTokenSource();
CancellationToken token = cancelTokenSource.Token;

Task task = new Task(() =>
{
    for (int i = 1; i < 10; i++)
    {
        if (token.IsCancellationRequested)
            token.ThrowIfCancellationRequested(); // генерируем исключение

        Console.WriteLine($"Квадрат числа {i} равен {i * i}");
        Thread.Sleep(200);
    }
}, token);
try
{
    task.Start();
    Thread.Sleep(1000);
    // после задержки по времени отменяем выполнение задачи
    cancelTokenSource.Cancel();

    // ожидаем завершения задачи
    Thread.Sleep(1000);
}
catch (AggregateException ae)
{
    foreach (Exception e in ae.InnerExceptions)
    {
        if (e is TaskCanceledException)
            Console.WriteLine("Операция прервана");
        else
            Console.WriteLine(e.Message);
    }
}
finally
{
    cancelTokenSource.Dispose();
}

//  проверяем статус задачи
Console.WriteLine($"Task Status: {task.Status}");
```

Консольный вывод программы:

```
Квадрат числа 1 равен 1
Квадрат числа 2 равен 4
Квадрат числа 3 равен 9
```

```
Квадрат числа 4 равен 16
Квадрат числа 5 равен 25
Task Status: Canceled
```

Регистрация обработчика отмены задачи

Выше для проверки сигнала отмены применялось свойство IsCancellationRequested. Но есть и другой способ узнать о том, что был послан сигнал отмены задачи. Метод Register() позволяет зарегистрировать обработчик отмены задачи в виде делегата Action:

```
CancellationTokenSource cancelTokenSource = new CancellationTokenSource();
CancellationToken token = cancelTokenSource.Token;

// задача вычисляет квадраты чисел
Task task = new Task(() =>
{
    int i = 1;
    token.Register(() =>
    {
        Console.WriteLine("Операция прервана");
        i = 10;
    });
    for (; i < 10; i++)
    {
        Console.WriteLine($"Квадрат числа {i} равен {i * i}");
        Thread.Sleep(400);
    }
}, token);
task.Start();

Thread.Sleep(1000);
// после задержки по времени отменяем выполнение задачи
cancelTokenSource.Cancel();
// ожидаем завершения задачи
Thread.Sleep(1000);
//  проверяем статус задачи
Console.WriteLine($"Task Status: {task.Status}");
cancelTokenSource.Dispose(); // освобождаем ресурсы
```

Здесь обработчик отмены представлен лямбда-выражением:

```
token.Register(() =>
{
    Console.WriteLine("Операция прервана");
    i = 10;
});
```

Поскольку действие задачи представляет цикл, который выполняется при значении i меньше 10, то установка этой переменной в обработчике отмены приведет к выходу из цикла и соответственно завершению задачи.

Передача токена во внешний метод

Если операция, которая выполняется в задаче, представляет внешний метод, то ему можно передавать в качестве одного из параметров:

```
CancellationTokenSource cancelTokenSource = new CancellationTokenSource();
CancellationToken token = cancelTokenSource.Token;

Task task = new Task(() =>PrintSquares(token), token);
try
{
    task.Start();
    Thread.Sleep(1000);
    // после задержки по времени отменяем выполнение задачи
    cancelTokenSource.Cancel();

    // ожидаем завершения задачи
    task.Wait();
}
catch (AggregateException ae)
{
    foreach (Exception e in ae.InnerExceptions)
    {
        if (e is TaskCanceledException)
            Console.WriteLine("Операция прервана");
        else
            Console.WriteLine(e.Message);
    }
}
finally
{
    cancelTokenSource.Dispose();
}

//  проверяем статус задачи
Console.WriteLine($"Task Status: {task.Status}");


void PrintSquares(CancellationToken token)
{
    for (int i = 1; i < 10; i++)
    {
        if (token.IsCancellationRequested)
            token.ThrowIfCancellationRequested(); // генерируем исключение

        Console.WriteLine($"Квадрат числа {i} равен {i * i}");
        Thread.Sleep(200);
```

```
    }
}
```

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:

```
CancellationTokenSource cancelTokenSource = new CancellationTokenSource();
CancellationToken token = cancelTokenSource.Token;

// в другой задаче посылаем сигнал отмены
new Task(() =>
{
    Thread.Sleep(400);
    cancelTokenSource.Cancel();
}).Start();

try
{
    Parallel.ForEach<int>(new List<int>() { 1, 2, 3, 4, 5},
                                new ParallelOptions { CancellationToken = token },
Square);
    // или так
    //Parallel.For(1, 5, new ParallelOptions { CancellationToken = token },
Square);
}
catch (OperationCanceledException)
{
    Console.WriteLine("Операция прервана");
}
finally
{
    cancelTokenSource.Dispose();
}

void Square(int n)
{
    Thread.Sleep(3000);
    Console.WriteLine($"Квадрат числа {n} равен {n * n}");
}
```

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