HW6 2150276 沈卓成

Part 1 基于任务的编程

1 线程池 WorkItem 到 Task

Code:

```
using System;

class Program
{
    static void WorkItem(object o)
    {
        Console.WriteLine("Here is the object: {0}", o);
    }
    static void Main(String[] args)
    {
        ThreadPool.QueueUserWorkItem(WorkItem, 100);
        Console.WriteLine("at main1");
        Thread.Sleep(1000);

        Task t = new Task(WorkItem, 200);
        t.Start();
        Console.WriteLine("at main2");
        t.Wait();
        Console.WriteLine("Main thread is done.");
    }
}
```

Ans:

```
(base) PS D:\dotnet\C-\HW6> dotnet run
D:\dotnet\C-\HW6\1.cs(9,43): warning CS8622: “void Program.WorkItem
WaitCallback”不匹配(
可能是由于为 Null 性特性)。 [D:\dotnet\C-\HW6\HW6.csproj]
D:\dotnet\C-\HW6\1.cs(13,27): warning CS8622: “void Program.WorkIte
“Action<object?>”
不匹配(可能是由于为 Null 性特性)。 [D:\dotnet\C-\HW6\HW6.csproj]
at main1
Here is the object: 100
at main2
Here is the object: 200
Main thread is done.
```

2 在任务完成时可以安排自动启动另一个新的任务

Code:

```
using System;

class Program
```

```
{
    static int WorkItem1(object o)
    {
        Console.WriteLine("Here1 is the object: {0}", o);
        return 1;
    }
    static void WorkItem2(Task<int> t)
    {   Console.WriteLine("Here2 is the object: {0}", t.Result);

    }
    static void Main(String[] args)
    {
        Task<int> ta = new Task<int>(WorkItem1, "Hello");
        Task tb = ta.ContinueWith(t=>WorkItem2(t));
        ta.Start();
        tb.Wait();
        Console.WriteLine("Main done");
    }
}
```

Ans:

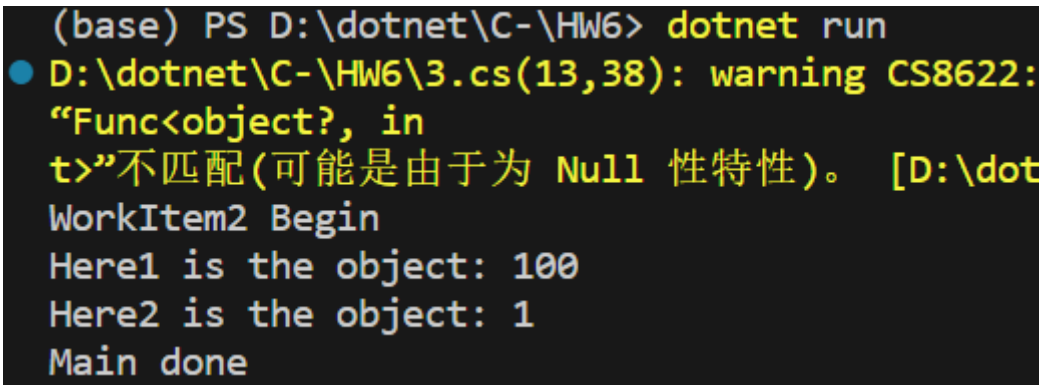
```
(base) PS D:\dotnet\C-\HW6> dotnet run
D:\dotnet\C-\HW6\2.cs(16,38): warning CS8622: “int Program.WorkItem1(o
“Func<object?, in
t>”不匹配(可能是由于为 Null 性特性)。 [D:\dotnet\C-\HW6\HW6.csproj]
Here1 is the object: Hello
Here2 is the object: 1
Main done
```

3 使用 async 和 await 进行异步编程

Code:

```
using System;

class Program
{
    static int WorkItem1(object o)
    {
        Console.WriteLine("Here1 is the object: {0}", o);
        return 1;
    }
    static async Task WorkItemAsync()
    {
        Console.WriteLine("WorkItem2 Begin");
        Task<int> ta = new Task<int>(WorkItem1, 100);
        ta.Start();
        var result = await ta;
        Console.WriteLine("Here2 is the object: {0}", result);
    }
    static void Main(String[] args)
    {
        Task t = WorkItemAsync();
        t.Wait();
        Console.WriteLine("Main done");
    }
}
```

Ans:



Q1 调用 async 修饰的异步方法，该方法在哪里执行?

Ans:

调用async修饰的异步方法会在所在的线程上开始执行，直到它达到await第一个未完成的任务或完成。在示例中，`WorkItemAsync`方法将开始在`Main`方法调用它的同一个线程上执行。

当`WorkItemAsync`到达`await`表达式时，它会异步等待`new Task<int>(WorkItem1, 100);`任务完成，而不会阻塞主线程。这意味着主线程可以继续执行其他工作，如果是UI应用程序，则保持响应。

一旦被await的任务(例如`ta`)完成，它将在一个线程池线程上恢复`WorkItemAsync`方法的执行。在这个例子中，之后执行`Console.WriteLine("Here2 is the object: {0}", result);`的代码位置则取决于具体的SynchronizationContext和TaskScheduler设置，默认情况下，它可能在不同的线程上执行。完成后，因为`Main`方法中调用了`t.Wait()`，主线程将等待`WorkItemAsync`方法完成才继续执行，执行`Console.WriteLine("Main done");`。

Q2 await 是否会阻塞当前线程?

Ans:

`await`关键字本身不会阻塞当前线程。它是用于异步编程的一种机制，当你在`await`一个异步操作时，当前方法的执行会在这一点暂停，直到所等待的任务完成，但它不会阻塞当前线程。

在等待异步操作期间，当前线程可以返回并执行其他任务，这就是使得`await`特别有用的原因：它允许你编写看起来像是同步代码的异步操作，但是没有线程的阻塞和资源费用。实际上，`await`后面的代码在异步操作完成后通常是在一个线程池线程上执行的，而不是在原始的调用线程上。

Part 2 TPL

1 创建任务

Code:

```
using System;
using System.Threading.Tasks;
using static System.Console;
using static System.Threading.Thread;

class Program
{
    static void TaskMethod(string name)
    {
        WriteLine($"Task {name} is running on a thread id " +
```

```
                    $"{CurrentThread.ManagedThreadId}. Is thread pool thread: " +
                    $"{CurrentThread.IsThreadPoolThread}");
    }

    static string TaskMethod2(string name)
    {
        return name + 500.ToString();
    }

    static void Main(string[] args)
    {
        //方式1: 创建任务对象并且启动,如果不调用Start，任务不会被执行
        var t1 = new Task(() => TaskMethod("Task 1"));
        var t2 = new Task(() => TaskMethod("Task 2"));
        t2.Start();
        t1.Start();

        //方式2: 创建任务并自动立即开始执行，可以提供一个参数
        Task.Factory.StartNew(() => TaskMethod("Task 3"));

        //标记为长时间运行的任务将不会使用线程池，而在单独的线程中运行
        Task.Factory.StartNew(() => TaskMethod("Task 4"),
TaskCreationOptions.LongRunning);
        //方式3: Task.Run是Task.Factory.StartNew的一个快捷方式
        Task.Run(() => TaskMethod("Task 5"));

        Task<string> t3 = Task.Run(() => TaskMethod2("Task 6 "));
        //t3.Wait();
        WriteLine($"TaskMethod2 result: {t3.Result}");

        Sleep(TimeSpan.FromSeconds(1));
    }
}
```

Ans:

```
(base) PS D:\dotnet\C-\HW6> dotnet run
TaskMethod2 result: Task 6 500
Task Task 2 is running on a thread id 4. Is thread pool thread: True
Task Task 3 is running on a thread id 8. Is thread pool thread: True
Task Task 4 is running on a thread id 6. Is thread pool thread: False
Task Task 5 is running on a thread id 9. Is thread pool thread: True
Task Task 1 is running on a thread id 7. Is thread pool thread: True
(base) PS D:\dotnet\C-\HW6>
```

代码分析:

程序中定义了两个方法：`TaskMethod` 和 `TaskMethod2`。`TaskMethod` 打印出任务的名字和执行该任务的线程ID，还显示该线程是否为线程池中的线程。`TaskMethod2` 接受一个名字字符串，返回这个字符串连接上 "500"。

在 `Main` 方法中，程序通过不同的方式来启动和运行这些任务：

1. 使用 `new Task()` 构造函数来创建任务 `t1` 和 `t2`，但这些任务默认并不启动，需要显示调用 `Start()` 方法来执行。在这个例子中，任务 `t1` 和 `t2` 都调用了同一个方法 `TaskMethod`，但使用了不同的参数来区分它们。
2. 使用 `Task.Factory.StartNew()` 创建并启动任务 `Task 3` 和 `Task 4`。这个方法创建了一个任务并且自动开始执行。`Task 4` 设置了 `TaskCreationOptions.LongRunning` 选项，表明它是一

个需长时间运行的任务，不应使用线程池里的线程。

3. 使用 `Task.Run()` 方法创建并启动任务 `Task 5`，这是 `Task.Factory.StartNew()` 的简化版，更直接地启动任务。
4. 使用 `Task.Run()` 方法执行 `TaskMethod2` 并返回一个 `Task<string>` 对象 `t3`。这允许程序获取异步操作的结果。在这个例子中，虽然有代码 `//t3.Wait();` 被注释了，表示可以等待任务结束，但程序显示了通过 `.Result` 属性立即获取任务 `t3` 的结果，这会导致主线程阻塞直到 `TaskMethod2` 完成并返回结果。

主线程休眠一秒钟，这是为了给其他启动的任务足够的时间执行，否则程序可能在它们完成前退出。

此外，代码的最后还演示了如何获取异步任务的返回值。通过 `t3.Result` 可以获取 `TaskMethod2` 返回的字符串，因为 `t3` 是一个泛型任务 `Task<string>`，其 `Result` 属性会返回任务的返回值。

主线程通过调用 `Sleep(TimeSpan.FromSeconds(1))` 方法暂停了一秒钟，为任务的执行提供额外的时间。如果没有足够的睡眠时间，那么主线程可能会在任务结束前退出，这将导致程序终止，使得一些任务可能无法正常完成。

2 运行任务并得到结果

Code:

```
using System;
using System.Threading.Tasks;
using static System.Console;
using static System.Threading.Thread;

class Program
{
    static int TaskMethod(string name)
    {
        WriteLine($"Task 【{name}】 is running on a thread id " +
                  $"{CurrentThread.ManagedThreadId}. Is thread pool thread: " +
                  $"{CurrentThread.IsThreadPoolThread}");
        Sleep(TimeSpan.FromSeconds(2));
        return 42;
    }
    static Task<int> CreateTask(string name)
    {
        return new Task<int>(() => TaskMethod(name));
    }
    static void Main(string[] args)
    {        //(1)在主线程直接调用方法
        TaskMethod("Main Thread Task");
        WriteLine("");

        //(2)创建一个任务,然后正常执行，得到结果
        Task<int> task = CreateTask("Task 1");
        task.Start();
        int result = task.Result;
        WriteLine($"Result is: {result}\r\n");
        //return;

        //(3)创建任务,使用RunSynchronously进行执行,得到结果
        task = CreateTask("Task 2");
        task.RunSynchronously();
        result = task.Result;
        WriteLine($"Result is: {result}\r\n");
```

```
        //(4)观察任务的运行状态
        task = CreateTask("Task 3");
        WriteLine(task.Status);
        task.Start();
        while (!task.IsCompleted)
        {
            WriteLine(task.Status);
            Sleep(TimeSpan.FromSeconds(0.5));
        }
        WriteLine(task.Status);
        result = task.Result;
        WriteLine($"Result is: {result}");
    }
}
```

Ans:

```
(base) PS D:\dotnet\C-\HW6> dotnet run
Task 【Main Thread Task】 is running on a thread id 1. Is thread pool thread: False

Task 【Task 1】 is running on a thread id 4. Is thread pool thread: True
Result is: 42

Task 【Task 2】 is running on a thread id 1. Is thread pool thread: False
Result is: 42

Created
Running
Task 【Task 3】 is running on a thread id 4. Is thread pool thread: True
Running
Running
Running
RanToCompletion
Result is: 42
(base) PS D:\dotnet\C-\HW6>
```

代码分析：

1. 首先，`TaskMethod`被直接在主线程中调用，运行`"Main Thread Task"`任务。
2. 接着，使用`CreateTask`方法创建了一个名为`"Task 1"`的新任务，通过调用`Start`方法启动这个任务，然后通过访问`Result`属性等待任务完成并获取结果。由于`Result`属性会阻塞调用线程直至任务完成，所以这里不需要使用`Wait`方法。
3. 然后，又创建了另一个名为`"Task 2"`的新任务，但是这次使用了`RunSynchronously`方法来运行任务。这会导致任务在调用它的当前线程上同步执行。然后同样通过`Result`属性获取执行结果。
4. 最后，展示了如何检测任务的状态，创建了名为`"Task 3"`的任务并在启动任务前打印其状态。任务启动后，程序进入一个循环，不断打印任务的状态，直到任务完成。循环后再次打印任务状态，然后通过`Result`获取结果。

3 组合任务

Code:

```
using System;
using System.Threading.Tasks;
using static System.Console;
using static System.Threading.Thread;

class Program
```

```
{
    static int TaskMethod(string name, int seconds)
    {
        WriteLine(
            $"Task {name} is running on a thread id " +
            $"{CurrentThread.ManagedThreadId}. Is thread pool thread: " +
            $"{CurrentThread.IsThreadPoolThread}");
        Sleep(TimeSpan.FromSeconds(seconds));

        //throw new Exception("err");

        return 42 * seconds;
    }

    static void Main(string[] args)
    {
        var firstTask  = new Task<int>(() => TaskMethod("First Task", 3));
        var secondTask = new Task<int>(() => TaskMethod("Second Task", 2));

        //使用ContinueWith组合任务
        firstTask.ContinueWith(
            t => WriteLine(
                $"The first answer is {t.Result}. Thread id " +
                $"{CurrentThread.ManagedThreadId}, is thread pool thread: " +
                $"{CurrentThread.IsThreadPoolThread}"),
            TaskContinuationOptions.OnlyOnRanToCompletion);
        // `TaskContinuationOptions.OnlyOnRanToCompletion` 的作用是确保延续操作只在前一个任务成功完成时(即没有抛出异常)执行。

        firstTask.Start();
        secondTask.Start();

        Sleep(TimeSpan.FromSeconds(4));



        //firstTask: Faulted
        //secondTask: Faulted
        //firstTask: RanToCompletion
        //secondTask: RanToCompletion
        WriteLine($"firstTask: {firstTask.Status}");
        WriteLine($"secondTask: {secondTask.Status}");


        Task continuation = secondTask.ContinueWith(
            t => WriteLine(
                $"The second answer is {t.Result}. Thread id " +
                $"{CurrentThread.ManagedThreadId}, is thread pool thread: " +
                $"{CurrentThread.IsThreadPoolThread}"),
            TaskContinuationOptions.OnlyOnRanToCompletion
            | TaskContinuationOptions.ExecuteSynchronously);

        Sleep(TimeSpan.FromSeconds(2));
        WriteLine("------------------------------------------------------");


        //return;
```

```
        firstTask = new Task<int>(() =>
        {
            var innerTask = Task.Factory.StartNew(() => TaskMethod("Second
Task", 5),
                TaskCreationOptions.AttachedToParent);

            innerTask.ContinueWith(t => TaskMethod("Third Task", 2),
                TaskContinuationOptions.AttachedToParent);

            return TaskMethod("First Task", 2);
        });

        firstTask.Start();

        while (!firstTask.IsCompleted)
        {
            WriteLine(firstTask.Status);
            Sleep(TimeSpan.FromSeconds(0.5));
        }
        WriteLine(firstTask.Status);

        Sleep(TimeSpan.FromSeconds(10));
    }
}
```

Ans:

```
(base) PS D:\dotnet\C-\HW6> dotnet run
Task Second Task is running on a thread id 6. Is thread pool thread: True
Task First Task is running on a thread id 4. Is thread pool thread: True
The first answer is 126. Thread id 4, is thread pool thread: True
firstTask: RanToCompletion
secondTask: RanToCompletion
The second answer is 84. Thread id 1, is thread pool thread: False
--------------------------------------------------
WaitingToRun
Task Second Task is running on a thread id 6. Is thread pool thread: True
Task First Task is running on a thread id 4. Is thread pool thread: True
Running
Running
Running
WaitingForChildrenToComplete
WaitingForChildrenToComplete
WaitingForChildrenToComplete
WaitingForChildrenToComplete
WaitingForChildrenToComplete
WaitingForChildrenToComplete
Task Third Task is running on a thread id 6. Is thread pool thread: True
WaitingForChildrenToComplete
WaitingForChildrenToComplete
WaitingForChildrenToComplete
WaitingForChildrenToComplete
RanToCompletion
```

1. 定义并启动两个任务`firstTask`和`secondTask`。每个任务都使用`TaskMethod`来运行一个作业，分别延迟3秒和2秒。
2. 通过`ContinueWith`方法创建了一个延续任务。当`firstTask`完成时，这个延续任务将执行并打印出`firstTask`的结果，以及它自己运行的线程信息。
   `TaskContinuationOptions.OnlyOnRanToCompletion`是一个标志，表示只有在`firstTask`成功运行并返回（未抛出任何异常）时，才会运行该延续任务。

3. 使用`Thread.Sleep`让主线程休眠4秒。
4. 之后打印出`firstTask`和`secondTask`的状态。
5. 对`secondTask`同样创建了一个延续任务，与`firstTask`的延续任务类似，但是这次在执行选项中增加了`TaskContinuationOptions.ExecuteSynchronously`，这表示延续任务将在`secondTask`所在的线程上同步执行。
6. 最后，定义了一个新的`firstTask`，这次在其中启动了一个嵌套任务`innerTask`，这个嵌套任务也有自己的延续任务。这些任务通过`TaskCreationOptions.AttachedToParent`和`TaskContinuationOptions.AttachedToParent`与父任务关联。主线程会等待`firstTask`完成，并不断打印`firstTask`的状态。

若取消注释：

```
Task Second Task is running on a thread id 6. Is thread pool thread: True
Task First Task is running on a thread id 4. Is thread pool thread: True
firstTask: Faulted
secondTask: Faulted
(base) PS D:\dotnet\C-\HW6>
```

多了Faulted的线程状态。

4 APM To Task

Code：

```
using System;
using System.Threading.Tasks;
using static System.Console;
using static System.Threading.Thread;

class Program
{
    delegate string AsynchronousTask(string threadName);
    delegate string IncompatibleAsynchronousTask(out int threadId);

    static void Callback(IAsyncResult ar)
    {
        WriteLine("Starting a callback...");
        WriteLine($"State passed to a callbak: {ar.AsyncState}");
        WriteLine($"Is thread pool thread: {CurrentThread.IsThreadPoolThread}");
        WriteLine($"Thread pool worker thread id:
{CurrentThread.ManagedThreadId}");
    }

    static string Test(string threadName)
    {
        WriteLine("Starting...");
        WriteLine($"Is thread pool thread: {CurrentThread.IsThreadPoolThread}");
        Sleep(TimeSpan.FromSeconds(2));
        CurrentThread.Name = threadName;
        return $"Thread name: {CurrentThread.Name}";
    }

    static string Test(out int threadId)
    {
        WriteLine("Starting...");
        WriteLine($"Is thread pool thread: {CurrentThread.IsThreadPoolThread}");
        Sleep(TimeSpan.FromSeconds(2));
        threadId = CurrentThread.ManagedThreadId;
```

```
        return $"Thread pool worker thread id was: {threadId}";
    }

    static void Main(string[] args)
    {
        int threadId;
        AsynchronousTask d             = Test;
        IncompatibleAsynchronousTask e = Test;

        WriteLine("Option 1");

        //FromAsync(IAsyncResult, Action<IAsyncResult>)
        //创建一个 Task，它在指定的 IAsyncResult 完成时执行一个结束方法操作
        Task<string> task = Task<string>.Factory.FromAsync(
            d.BeginInvoke("AsyncTaskThread", Callback,
            "a delegate asynchronous call"), d.EndInvoke);

        task.ContinueWith(t => WriteLine(
            $"Callback is finished, now running a continuation! Result:
{t.Result}"));

        while (!task.IsCompleted)
        {
            WriteLine(task.Status);
            Sleep(TimeSpan.FromSeconds(0.5));
        }
        WriteLine(task.Status);
        Sleep(TimeSpan.FromSeconds(1));


        WriteLine("------------------------------------------------");
        WriteLine();
        WriteLine("Option 2");

        task = Task<string>.Factory.FromAsync(
            d.BeginInvoke, d.EndInvoke, "AsyncTaskThread", "a delegate
asynchronous call");

        task.ContinueWith(t => WriteLine(
            $"Task is completed, now running a continuation! Result:
{t.Result}"));
        while (!task.IsCompleted)
        {
            WriteLine(task.Status);
            Sleep(TimeSpan.FromSeconds(0.5));
        }
        WriteLine(task.Status);
        Sleep(TimeSpan.FromSeconds(1));

        WriteLine("------------------------------------------------");
        WriteLine();
        WriteLine("Option 3");


        IAsyncResult ar = e.BeginInvoke(out threadId, Callback, "a delegate
asynchronous call");
        task = Task<string>.Factory.FromAsync(ar, _ => e.EndInvoke(out threadId,
ar));
```

```
        task.ContinueWith(t =>
            WriteLine(
                $"Task is completed, now running a continuation! " +
                $"Result: {t.Result}, ThreadId: {threadId}"));

        while (!task.IsCompleted)
        {
            WriteLine(task.Status);
            Sleep(TimeSpan.FromSeconds(0.5));
        }
        WriteLine(task.Status);

        Sleep(TimeSpan.FromSeconds(1));
    }
}
```

Ans:

```
(base) PS D:\dotnet\C-\HW6> dotnet run
Option 1
Unhandled exception. System.PlatformNotSupportedException: Operation is not supported on this platform.
   at Program.AsynchronousTask.BeginInvoke(String threadName, AsyncCallback callback, Object object)
   at Program.Main(String[] args) in D:\dotnet\C-\HW6\7.cs:line 47
```

APM在.NET 8已经不支持相关函数！！！

5 取消任务执行

Code:

```
using System;
using System.Threading;
using System.Threading.Tasks;
using static System.Console;
using static System.Threading.Thread;

class Program
{
    static int TaskMethod(string name, int seconds, CancellationToken token)
    {
        WriteLine(
            $"Task {name} is running on a thread id " +
            $"{CurrentThread.ManagedThreadId}. Is thread pool thread: " +
            $"{CurrentThread.IsThreadPoolThread}");

        for (int i = 0; i < seconds; i ++)
        {
            Sleep(TimeSpan.FromSeconds(1));
            if (token.IsCancellationRequested)
            {
                WriteLine("Cancel Here.");
                return -1;
            }
        }
        return 42*seconds;
    }

    static void Main(string[] args)
    {
```

```
        //参数cts.Token传递两次的原因:如果任务没有开始就被取消，需要有TPL基础设施处理取消操
作

        var cts = new CancellationTokenSource();
        var longTask = new Task<int>(() => TaskMethod("Task 1", 10, cts.Token),
cts.Token);
        WriteLine(longTask.Status);
        cts.Cancel();
        WriteLine(longTask.Status);
        WriteLine("First task has been cancelled before execution");

        cts = new CancellationTokenSource();
        longTask = new Task<int>(() => TaskMethod("Task 2", 10, cts.Token),
cts.Token);
        longTask.Start();
        for (int i = 0; i < 5; i++ )
        {
            Sleep(TimeSpan.FromSeconds(0.5));
            WriteLine(longTask.Status);
        }
        cts.Cancel();
        for (int i = 0; i < 5; i++)
        {
            Sleep(TimeSpan.FromSeconds(0.5));
            WriteLine(longTask.Status);
        }
        WriteLine($"A task has been completed with result {longTask.Result}.");
    }
}
```

Ans:

```
(base) PS D:\dotnet\C-\HW6> dotnet run
Created
Canceled
First task has been cancelled before execution
Task Task 2 is running on a thread id 4. Is thread pool thread: True
Running
Running
Running
Running
Running
Cancel Here.
RanToCompletion
RanToCompletion
RanToCompletion
RanToCompletion
RanToCompletion
A task has been completed with result -1.
```

定义了一个`TaskMethod`方法，它接收一个任务名称、一个执行时长（秒）和一个`CancellationToken`。该方法运行时会在控制台输出执行的任务信息，并每秒检查一次是否有取消请求。如果有取消请求，则输出相应信息并立即返回-1；如果没有被取消，任务正常完成，返回42乘以秒数。定时输出，显示主线程ID和是否为线程池线程的信息。实现一个循环，该循环通过休眠（`Sleep`）模拟长时间执行操作，并检查取消令牌以确定是否应该中止执行。`Main`方法是程序的入口点。首先创建一个`CancellationTokenSource`并声明一个`longTask`任务（此任务尚未启动）。任务将执行`TaskMethod`。输出`longTask`的状态（应该是`Created`，因为此时任务还未启动）。调用`cts.Cancel()`取消任务。再次输出`longTask`的状态（应该仍然是`Created`）。输出提示信息，表示任务在开始执行前已被取消。创建新的`CancellationTokenSource`和`longTask`以供第二个任务使

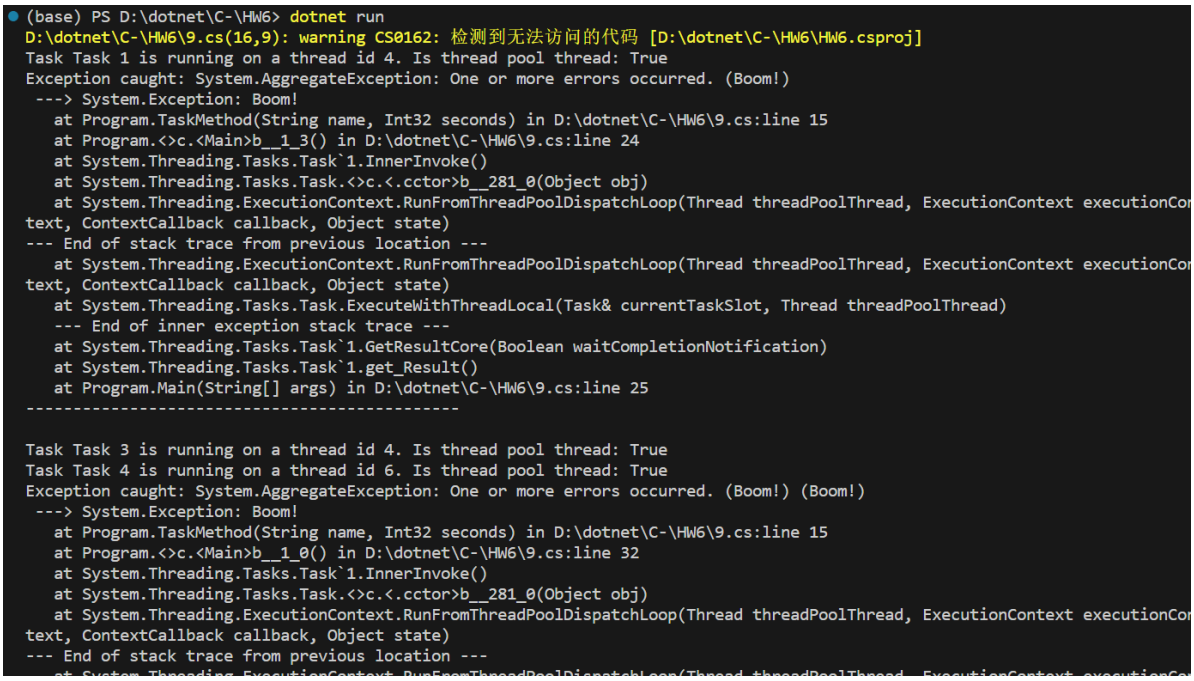
用。调用`Start`方法启动任务。执行一个循环，每隔半秒输出当前任务的状态。循环五次以等待一些执行时间。调用`Cancel`方法尝试取消任务。再次执行循环，继续监视和输出任务的状态。尝试获取任务的结果，如果任务已经完成，它会输出任务返回的结果。

6 任务异常处理

Code:

```
using System;
using System.Threading.Tasks;
using static System.Console;
using static System.Threading.Thread;

class Program
{
    static int TaskMethod(string name, int seconds)
    {        WriteLine(
            $"Task {name} is running on a thread id " +
            $"{CurrentThread.ManagedThreadId}. Is thread pool thread: " +
            $"{CurrentThread.IsThreadPoolThread}");

        Sleep(TimeSpan.FromSeconds(seconds));
        throw new Exception("Boom!");
        return 42 * seconds;
    }

    static void Main(string[] args)
    {        //这个这样捕捉到任务产生的异常
        Task<int> task;
        try
        {
            task = Task.Run(() => TaskMethod("Task 1", 2));
            int result = task.Result;
            WriteLine($"Result: {result}");
        }        catch (Exception ex)
        {            WriteLine($"Exception caught: {ex}");
        }        WriteLine("-----------------------------------------------");
        WriteLine();

       var t1 = new Task<int>(() => TaskMethod("Task 3", 3));
        var t2 = new Task<int>(() => TaskMethod("Task 4", 2));
        var complexTask = Task.WhenAll(t1, t2); // Task.WhenAll 方法会等待所有的任务都完成后才会继续执行
        var exceptionHandler = complexTask.ContinueWith(t =>
                WriteLine($"Exception caught: {t.Exception}"),
TaskContinuationOptions.OnlyOnFaulted // exceptionHandler 任务会在 complexTask 发生异常时执行
        );
        t1.Start();
        t2.Start();

        Sleep(TimeSpan.FromSeconds(5));
    }}
```

Ans:



1. 在第一部分中，通过创建一个名为 `TaskMethod` 的方法来模拟一个耗时任务。在这个方法中，打印了一些有关当前执行的线程信息，并让线程休眠了指定的秒数。在任务完成后，这个方法将会抛出一个异常。
2. 在 `Main` 方法中，通过 `Task.Run` 创建并启动了一个任务，然后尝试获取任务的结果。因为 `TaskMethod` 中会抛出异常，所以尝试获取任务返回值的时候就会触发这个异常，然后在 `catch` 块中打印出这个异常信息。

然后又创建了两个新任务 `t1` 和 `t2`，然后通过 `Task.WhenAll(t1, t2)` 来创建一个新任务 `complexTask`，这个新任务将会在 `t1` 和 `t2` 都完成的时候才会完成。然后通过 `complexTask.ContinueWith` 方法添加一个后续任务 `exceptionHandler`，这个后续任务将会在 `complexTask` 失败的时候触发，并打印出异常信息。

最后启动了 `t1` 和 `t2` 这两个任务，然后让主线程休眠了5秒钟以等待这两个任务完成。

7 等待任务完成

Code:

```
using System;
using System.Collections.Generic;
using System.Threading.Tasks;
using static System.Console;
using static System.Threading.Thread;

class Program
{
    static int TaskMethod(string name, int seconds)
    {
        WriteLine(
            $"Task {name} is running on a thread id " +
            $"{CurrentThread.ManagedThreadId}. Is thread pool thread: " +
            $"{CurrentThread.IsThreadPoolThread}");

        Sleep(TimeSpan.FromSeconds(seconds));
        return 42 * seconds;
    }
```

```
    static void Main(string[] args)
    {
        var firstTask = new Task<int>(() => TaskMethod("First Task", 3));
        var secondTask = new Task<int>(() => TaskMethod("Second Task", 2));

        //(1)使用WhenAll等待所有任务完成

        var whenAllTask = Task.WhenAll(firstTask, secondTask);

        whenAllTask.ContinueWith(t =>
            WriteLine($"The first answer is {t.Result[0]}, the second is
{t.Result[1]}"),
            TaskContinuationOptions.OnlyOnRanToCompletion);

        firstTask.Start();
        secondTask.Start();

        Sleep(TimeSpan.FromSeconds(4));

        //(2)使用WhenAll等待所有任务完成
        var tasks = new List<Task<int>>();
        for (int i = 1; i < 4; i++)
        {
            int counter = i;
            var task = new Task<int>(() => TaskMethod($"Task {counter}",
counter));
            tasks.Add(task);
            task.Start();
        }

        while (tasks.Count > 0)
        {
            var completedTask = Task.WhenAny(tasks).Result;
            tasks.Remove(completedTask);
            WriteLine($"A task has been completed with result
{completedTask.Result}.");
        }

        Sleep(TimeSpan.FromSeconds(1));
    }
}
```

Ans:

```
(base) PS D:\dotnet\C-\HW6> dotnet run
Task Second Task is running on a thread id 6. Is thread pool thread: True
Task First Task is running on a thread id 4. Is thread pool thread: True
The first answer is 126, the second is 84
Task Task 1 is running on a thread id 4. Is thread pool thread: True
Task Task 2 is running on a thread id 6. Is thread pool thread: True
Task Task 3 is running on a thread id 7. Is thread pool thread: True
A task has been completed with result 42.
A task has been completed with result 84.
A task has been completed with result 126.
```

1. 首先，它定义了一个`TaskMethod`方法，这个方法用于模拟一个长时间运行的任务，它会接收一个任务名和一个暂停的秒数，然后在控制台中输出任务的信息，并让当前线程暂停指定的秒数，最后返回一个计算结果。

2. 在`Main`方法中，首先创建了两个任务（firstTask 和 secondTask），但并没有启动它们。
3. 然后使用了`Task.WhenAll`方法来创建一个新的任务，这个新任务会在firstTask 和 secondTask这两个任务都完成的时候完成，并在任务完成后输出它们的执行结果。
4. 调用这两个任务的`Start`方法来启动它们。
5. 之后，`Main`方法主线程暂停了4秒等待这两个任务完成。
6. 然后又创建了三个任务并启动，这次使用`Task.WhenAny`方法来等待任何一个任务完成，并输出完成任务的执行结果，这个过程会持续到所有任务都完成。
7. 最后，`Main`方法主线程再次暂停了1秒等待所有任务完成。