创建任务时，给任务绑定一个CancellationToken，那么我们就可以在外界取消task；

task处于未曾占用过CPU，未曾进入过Running状态：

1. 引用已经返回，但是还未调用Start()；
2. 调用Start()，但是调用后的极短时间
3. Task.Factory.StartNew（）后的极短时间

这样才能“取消task”，source.Cancel（）才有作用；当任务进入Running状态后，source.Cancel（）无效，和不存在一样，就当CancellationToken不曾与任务绑定过；

1. 创建CancellationTokenSource对象

static CancellationTokenSource cancelTokenSource = new CancellationTokenSource();

1. 创建任务时，将CancellationTokenSource对象的属性Token(CancellationToken对象)与任务绑定；

Task.Factory.StartNew(MyTask, cancelTokenSource.Token);

1. 取消任务：调用CancellationTokenSource对象的Cancel()；（仅任务未进入过Running状态才有作用）

cancelTokenSource.Cancel();

1. 检查任务是否被成功取消

if(task.Status == Canceled)

{

//自定义操作

}

补充说明：

if(!cancelTokenSource.IsCancellationRequested) 或 if(!cancelToken.IsCancellationRequested)

{

//自定义操作

}

这样并不能检查任务是否被取消执行了，这只，当于方法的逻辑检查static bool endFlag是不是置成true了，从而改变任务内方法的执行流；

CancellationToken是ConcellationTokenSoure的一个属性，实现取消任务的是CancellationToken，ConcellationTokenSoure只是封装一下CancellationToken；

看懂下面三个示例，就明白了token的作用了

class Program

{

public static CancellationTokenSource source = new CancellationTokenSource();

public static CancellationToken token = source.Token;

static void Main(string[] args)

{

Task task = new Task(() => { TaskMethod(); }, token);

task.Start();

source.Cancel();

if(token.IsCancellationRequested == true)

{

Console.WriteLine("任务被取消了，没有执行");

}

Console.WriteLine(task.Status);

Console.Read();

}

static void TaskMethod()

{

for(int i = 0;i < 10; ++i)

{

Console.WriteLine("任务执行了，并没有被取消");

Thread.Sleep(200);

}

}

}

输出结果：

任务被取消了，没有执行

Canceled

示例二

class Program

{

public static CancellationTokenSource source = new CancellationTokenSource();

public static CancellationToken token = source.Token;

static void Main(string[] args)

{

Task task = new Task(() => { TaskMethod(); }, token);

task.Start();

Thread.Sleep(100);

source.Cancel();

if(token.IsCancellationRequested == true)

{

Console.WriteLine("任务被取消了，没有执行");

}

Console.WriteLine(task.Status);

Console.Read();

}

static void TaskMethod()

{

for(int i = 0;i < 10; ++i)

{

Console.WriteLine("任务执行了，并没有被取消");

Thread.Sleep(200);

}

}

}

输出结果：

任务执行了，并没有被取消

任务被取消了，没有执行

Running

任务执行了，并没有被取消

任务执行了，并没有被取消

任务执行了，并没有被取消

任务执行了，并没有被取消

任务执行了，并没有被取消

任务执行了，并没有被取消

任务执行了，并没有被取消

任务执行了，并没有被取消

任务执行了，并没有被取消

示例三

class Program

{

public static CancellationTokenSource source = new CancellationTokenSource();

public static CancellationToken token = source.Token;

static void Main(string[] args)

{

Task task = new Task(() => { TaskMethod(); }, token);

task.Start();

source.Cancel();

Thread.Sleep(100);

if (token.IsCancellationRequested == true)

{

Console.WriteLine("任务被取消了，没有执行");

}

Console.WriteLine(task.Status);

Console.Read();

}

static void TaskMethod()

{

for(int i = 0;i < 10; ++i)

{

Console.WriteLine("任务执行了，并没有被取消");

Thread.Sleep(200);

}

}

输出结果：

任务被取消了，没有执行

Canceled

任务取消的扩展

static CancellationTokenSource source = new CancellationTokenSource();

static CancellationToken token = source.Token;

Task task = new Task(()=> { Fun(token);},token);

两个token都能影响task的状态；

第二个token的作用就是“取消线程”，“取消线程”仅仅能在任务未跑起来前取消，一旦跑起来就没有“取消线程”这种说法了；后者token起作用的时间段：在线程还未跑起来，即Start()之前，Start()之后的极短时间，Task.Factory.Startwith()开启任务的极短时间内还在等待调度器选择，这些时间段或时间点任务都未曾占用CPU进入Running状态； 一旦任务进入Running状态后，第二个token不再起任何作用；一旦线程跑起来，即使source.Cancel()，任务的状态也不可能是Canceled，仅仅可能是Faulted和RanToCompletion;

任务跑起来后，线程的流程完全由第一个token从业务逻辑上控制，间接影响线程的状态,相当于在线程的内部埋下了一个开关量，随时打开或关闭开关，影响方法的执行流程。

示例一：

class Program

{

public static CancellationTokenSource source = new CancellationTokenSource();

public static CancellationToken token = source.Token;

static void Main(string[] args)

{

Task task = new Task(() => { TaskMethod(token); }, token);

Console.WriteLine(task.Status);

task.Start();

source.Cancel();

Console.WriteLine(task.Status);

for (int i=0;i < 5; ++i)

{

Console.WriteLine(task.Status);

Thread.Sleep(3000);

}

Console.Read();

}

static void TaskMethod(CancellationToken token)

{

if (token.IsCancellationRequested == true) //而不是if (source.IsCancellationRequested == true)

{

Console.WriteLine("方法内逻辑结束方法，从而线程运行结束");

return;

}

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

{

Thread.Sleep(1000);

if (token.IsCancellationRequested == true) //token.IsCancellationRequested == true

{

Console.WriteLine("方法内逻辑结束方法，从而线程运行结束");

return;

}

//检查到标志位没置否，进行其他一些操作

Console.WriteLine("方法内逻辑没让方法返回，所以线程没运行结束");

}

}

}

输出结果：

Created

WaitingToRun

WaitingToRun

Canceled

Canceled

Canceled

Canceled

1. 任务还未Start()，状态时Created
2. 还未被CPU调度，状态是WaitingToRun
3. 等调度时，发现已经被取消，状态是Canceled，不再执行该任务

示例二：

class Program

{

public static CancellationTokenSource source = new CancellationTokenSource();

public static CancellationToken token = source.Token;

static void Main(string[] args)

{

Task task = new Task(() => { TaskMethod(token); }, token);

Console.WriteLine(task.Status);

task.Start();

Console.WriteLine(task.Status);

for (int i=0;i < 5; ++i)

{

Console.WriteLine(task.Status);

Thread.Sleep(2000);

if(i == 2)

{

source.Cancel();

}

}

Console.Read();

}

static void TaskMethod(CancellationToken token)

{

if (token.IsCancellationRequested == true)

{

Console.WriteLine("方法内逻辑结束方法，从而线程运行结束");

return;

}

for (int i = 0; i < 10; ++i)

{

Thread.Sleep(1000);

if (token.IsCancellationRequested == true) //token.IsCancellationRequested == true

{

Console.WriteLine("方法内逻辑结束方法，从而线程运行结束");

return;

}

//检查到标志位没置否，进行其他一些操作

Console.WriteLine("方法内逻辑没让方法返回，所以线程没运行结束");

}

}

}

输出结果：

Created

WaitingToRun

WaitingToRun

方法内逻辑没让方法返回，所以线程没运行结束

Running

方法内逻辑没让方法返回，所以线程没运行结束

方法内逻辑没让方法返回，所以线程没运行结束

Running

方法内逻辑没让方法返回，所以线程没运行结束

方法内逻辑没让方法返回，所以线程没运行结束

Running

方法内逻辑结束方法，从而线程运行结束

RanToCompletion

1. 任务还未Start()，状态时Created
2. 还未被CPU调度，状态是WaitingToRun
3. 任务进入Running状态，第二个Cancel无效；

任务和主线程轮流占用CPU，当主线程的i等于2时，任务返回，任务结束，此时任务的状态成RanToCompletion;(而不是Canceled)

namespace ConsoleApp1

{

/\*分析：

\* TaskMethod()方法的参数token，是一个普通的变量，与Task的状态什么的完全无关，仅仅用于控制方法的流程；

\* 等效于下面代码中的全局静态变量endFlag；（开关变量）

\* source.Cancel();相当于 endFlag = true；

class Program

{

public static bool endFlag = false;

// public static CancellationTokenSource source = new CancellationTokenSource();

//public static CancellationToken token = source.Token;

static void Main(string[] args)

{

Task task = new Task(() => { TaskMethod(); });

task.Start();

Thread.Sleep(5000);

endFlag = true;

//source.Cancel();

Console.Read();

}

static void TaskMethod()

{

if (endFlag == true) //token.IsCancellationRequested == true

{

Console.WriteLine("方法内逻辑取消线程");

return;

}

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

{

Thread.Sleep(1000);

if (endFlag == true) //token.IsCancellationRequested == true

{

Console.WriteLine("方法内逻辑取消线程");

return;

}

//检查到标志位没置否，进行其他一些操作

Console.WriteLine("方法内逻辑没取消线程，方法继续执行");

}

}

}

\*

\*/

class Program

{

public static CancellationTokenSource source = new CancellationTokenSource();

public static CancellationToken token = source.Token;

static void Main(string[] args)

{

//开启一个线程

Task task = new Task(()=> { TaskMethod(token); });

task.Start();

Thread.Sleep(5000);

//从方法逻辑上结束线程

source.Cancel();

Console.Read();

}

static void TaskMethod(CancellationToken token)

{

//逻辑上检查是否结束线程，它与业务逻辑紧紧的绑定在一起

if (token.IsCancellationRequested == true)

{

Console.WriteLine("方法内逻辑取消线程");

return;

}

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

{

Thread.Sleep(1000);

if (token.IsCancellationRequested == true)

{

Console.WriteLine("方法内逻辑取消线程");

return;

}

//检查到标志位没置否，进行其他一些操作

Console.WriteLine("方法内逻辑没取消线程，方法继续执行");

}

}

}

}

定时取消任务

var cancelTokenSource = new CancellationTokenSource(3000);

3秒后，自动调用cancelTokenSource.Cancel()取消任务的执行；当然，3秒后，如果任务已经正在执行或者执行完毕，就没有用；如果任务3秒后还没开始，那就真的自动取消执行了。

任务取消的回调函数

            // create cancellationTokenSource, which decide when to cancel(abort) thread  
            CancellationTokenSource cts = new CancellationTokenSource();  
           cts.Token.Register(() => Console.WriteLine("CancellationToken register callback function invoked..."));

当任务真的被取消了，就会执行该回调函数；

关联的CancellationTokenSource

还有一种 link 的方式创建 TokenSource，这种方式使得如果关联的 TokenSource，cancel 了，那么将导致这个 TokenSource 也会 cancel            CancellationTokenSource one = new CancellationTokenSource();

            one.Token.Register(() => Console.WriteLine("One register function invoked..."));  
            //create another new cancellationTokenSource  
            CancellationTokenSource two = new CancellationTokenSource();  
            two.Token.Register(() => Console.WriteLine("Two register function invoked..."));  
            //create a linked cancelationTokenSource  
            CancellationTokenSource three = CancellationTokenSource.CreateLinkedTokenSource(one.Token, two.Token);  
            three.Token.Register(() => Console.WriteLine("Three register function invoked..."));  
            Console.WriteLine("One's IsCancellationRequested: {0} \nTwo's IsCancellationRequested: {1} \nThree'sIsCancellationRequested: {2}",  
                one.IsCancellationRequested, two.IsCancellationRequested, three.IsCancellationRequested);  
            two.Cancel();  
            Console.WriteLine("One's IsCancellationRequested: {0} \nTwo's IsCancellationRequested: {1} \nThree'sIsCancellationRequested: {2}",  
                one.IsCancellationRequested, two.IsCancellationRequested, three.IsCancellationRequested);  
            one.Cancel();  
            Console.WriteLine("One's IsCancellationRequested: {0} \nTwo's IsCancellationRequested: {1} \nThree'sIsCancellationRequested: {2}",  
                one.IsCancellationRequested, two.IsCancellationRequested, three.IsCancellationRequested);

虽然暂时没有想到用在哪里，但是感觉还是很有用处的。

可以看到，CancellationToken 是真正起作用的关键点，CancellationTokenSource 是对它的一个包装，在内部，定义了 CancellationToken 对象， 它是一个结构体，重载了!= == 操作符。如果想在最开始的 Demo 中不让线程由 CancellationToken 控制而退出，那可以在传参 CancellationToken.None