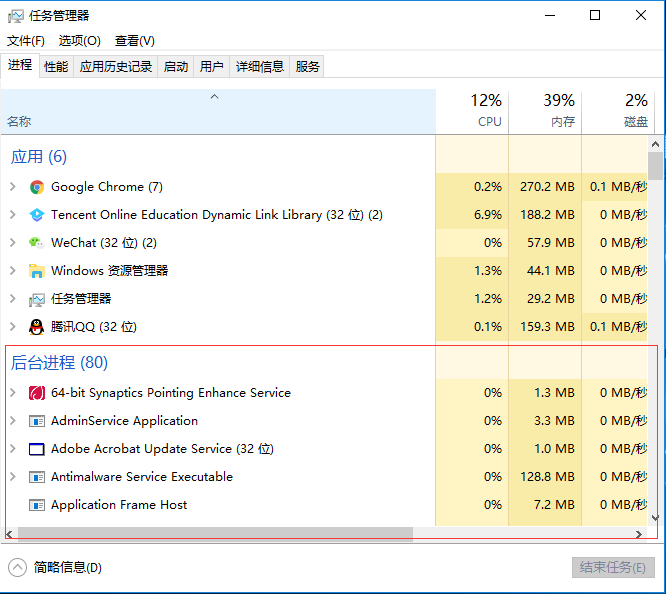
# [C#多线程](https://www.cnblogs.com/dotnet261010/p/6159984.html)

一、基本概念

1、进程

首先打开任务管理器，查看当前运行的进程：

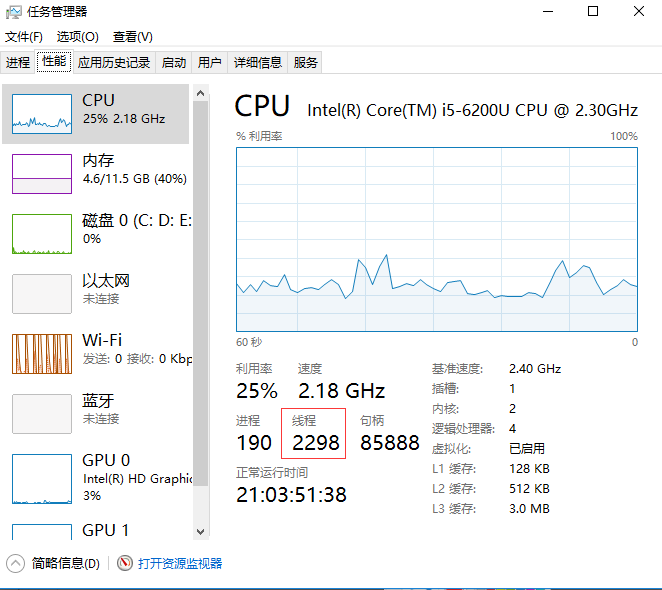


从任务管理器里面可以看到当前所有正在运行的进程。那么究竟什么是进程呢？

进程（Process）是Windows系统中的一个基本概念，它包含着一个运行程序所需要的资源。一个正在运行的应用程序在操作系统中被视为一个进程，进程可以包括一个或多个线程。线程是操作系统分配处理器时间的基本单元，在进程中可以有多个线程同时执行代码。进程之间是相对独立的，一个进程无法访问另一个进程的数据（除非利用分布式计算方式），一个进程运行的失败也不会影响其他进程的运行，Windows系统就是利用进程把工作划分为多个独立的区域的。进程可以理解为一个程序的基本边界。是应用程序的一个运行例程，是应用程序的一次动态执行过程。

二、线程

在任务管理器里面查询当前总共运行的线程数：



线程（Thread）是进程中的基本执行单元，是操作系统分配CPU时间的基本单位，一个进程可以包含若干个线程，在进程入口执行的第一个线程被视为这个进程的主线程。在.NET应用程序中，都是以Main()方法作为入口的，当调用此方法时系统就会自动创建一个主线程。线程主要是由CPU寄存器、调用栈和线程本地存储器（Thread Local Storage，TLS）组成的。CPU寄存器主要记录当前所执行线程的状态，调用栈主要用于维护线程所调用到的内存与数据，TLS主要用于存放线程的状态信息。

二、多线程

多线程的优点：可以同时完成多个任务；可以使程序的响应速度更快；可以让占用大量处理时间的任务或当前没有进行处理的任务定期将处理时间让给别的任务；可以随时停止任务；可以设置每个任务的优先级以优化程序性能。

那么可能有人会问：为什么可以多线程执行呢？总结起来有下面两方面的原因：

1、CPU运行速度太快，硬件处理速度跟不上，所以操作系统进行分时间片管理。这样，从宏观角度来说是多线程并发的，因为CPU速度太快，察觉不到，看起来是同一时刻执行了不同的操作。但是从微观角度来讲，同一时刻只能有一个线程在处理。

2、目前电脑都是多核多CPU的，一个CPU在同一时刻只能运行一个线程，但是多个CPU在同一时刻就可以运行多个线程。

然而，多线程虽然有很多优点，但是也必须认识到多线程可能存在影响系统性能的不利方面，才能正确使用线程。不利方面主要有如下几点：

（1）线程也是程序，所以线程需要占用内存，线程越多，占用内存也越多。

（2）多线程需要协调和管理，所以需要占用CPU时间以便跟踪线程。

（3）线程之间对共享资源的访问会相互影响，必须解决争用共享资源的问题。

（4）线程太多会导致控制太复杂，最终可能造成很多程序缺陷。

当启动一个可执行程序时，将创建一个主线程。在默认的情况下，C#程序具有一个线程，此线程执行程序中以Main方法开始和结束的代码，Main（）方法直接或间接执行的每一个命令都有默认线程（主线程）执行，当Main（）方法返回时此线程也将终止。

一个进程可以创建一个或多个线程以执行与该进程关联的部分程序代码。在C#中，线程是使用Thread类处理的，该类在System.Threading命名空间中。使用Thread类创建线程时，只需要提供线程入口，线程入口告诉程序让这个线程做什么。通过实例化一个Thread类的对象就可以创建一个线程。创建新的Thread对象时，将创建新的托管线程。Thread类接收一个ThreadStart委托或ParameterizedThreadStart委托的构造函数，该委托包装了调用Start方法时由新线程调用的方法，示例代码如下：

Thread thread=new Thread(new ThreadStart(method));//创建线程

thread.Start();                                                           //启动线程

上面代码实例化了一个Thread对象，并指明将要调用的方法method()，然后启动线程。ThreadStart委托中作为参数的方法不需要参数，并且没有返回值。ParameterizedThreadStart委托一个对象作为参数，利用这个参数可以很方便地向线程传递参数，示例代码如下：

Thread thread=new Thread(new ParameterizedThreadStart(method));//创建线程

thread.Start(3);                                                                             //启动线程

创建多线程的步骤：  
1、编写线程所要执行的方法  
2、实例化Thread类，并传入一个指向线程所要执行方法的委托。（这时线程已经产生，但还没有运行）  
3、调用Thread实例的Start方法，标记该线程可以被CPU执行了，但具体执行时间由CPU决定

2.1 System.Threading.Thread类

###### Thread类是是控制线程的基础类，位于System.Threading命名空间下，具有4个重载的构造函数：

[Thread(ParameterizedThreadStart)](https://msdn.microsoft.com/zh-cn/library/1h2f2459(v=vs.110).aspx):初始化 Thread 类的新实例，指定允许对象在线程启动时传递给线程的委托。要执行的方法是有参的。

[Thread(ParameterizedThreadStart, Int32)](https://msdn.microsoft.com/zh-cn/library/ms149581(v=vs.110).aspx):初始化 Thread 类的新实例，指定允许对象在线程启动时传递给线程的委托，并指定线程的最大堆栈大小

[Thread(ThreadStart)](https://msdn.microsoft.com/zh-cn/library/xx3ezzs2(v=vs.110).aspx):初始化 Thread 类的新实例。要执行的方法是无参的。

[Thread(ThreadStart, Int32)](https://msdn.microsoft.com/zh-cn/library/5cykbwz4(v=vs.110).aspx):初始化 Thread 类的新实例，指定线程的最大堆栈大小。

ThreadStart是一个无参的、返回值为void的委托。委托定义如下：

public delegate void ThreadStart()

ParameterizedThreadStart是一个有参的、返回值为void的委托，定义如下：

public delegate void ParameterizedThreadStart(Object obj)

注意：ParameterizedThreadStart委托的参数类型必须是Object的。如果使用的是不带参数的委托，不能使用带参数的Start方法运行线程，否则系统会抛出异常。但使用带参数的委托，可以使用thread.Start()来运行线程，这时所传递的参数值为null。

2.2 线程的常用属性

CurrentContext:获取线程正在其中执行的当前上下文。

CurrentThread:获取当前正在运行的线程

ExecutionContext:获取一个 ExecutionContext 对象，该对象包含有关当前线程的各种上下文的信息。

IsAlive:获取一个值，该值指示当前线程的执行状态

IsBackground:获取或设置一个值，该值指示某个线程是否为后台线程。

IsThreadPoolThread:获取一个值，该值指示线程是否属于托管线程池。

ManagedThreadId:获取当前托管线程的唯一标识符

Name:获取或设置线程的名称。

Priority:获取或设置一个值，该值指示线程的调度优先级

ThreadState:获取一个值，该值包含当前线程的状态

**2.2.1 线程的标识符**

ManagedThreadId是确认线程的唯一标识符，程序在大部分情况下都是通过Thread.ManagedThreadId来辨别线程的。而Name是一个可变值，在默认时候，Name为一个空值 Null，开发人员可以通过程序设置线程的名称，但这只是一个辅助功能。

**2.2.2 线程的优先级别**

当线程之间争夺CPU时间时，CPU按照线程的优先级给予服务。高优先级的线程可以完全阻止低优先级的线程执行。.NET为线程设置了Priority属性来定义线程执行的优先级别，里面包含5个选项，其中Normal是默认值。除非系统有特殊要求，否则不应该随便设置线程的优先级别。随意设置线程优先级也会导致线程阻塞的问题。

Lowest：可以将 Thread 安排在具有任何其他优先级的线程之后。

BelowNormal：可以将 Thread 安排在具有 Normal 优先级的线程之后，在具有 Lowest 优先级的线程之前。

Normal：默认选择。可以将 Thread 安排在具有 AboveNormal 优先级的线程之后，在具有 BelowNormal 优先级的线程之前。

AboveNormal：可以将 Thread 安排在具有 Highest 优先级的线程之后，在具有 Normal 优先级的线程之前。

Highest：可以将 Thread 安排在具有任何其他优先级的线程之前。

**2.2.3 线程的状态**

通过ThreadState可以检测线程是处于Unstarted、Sleeping、Running 等等状态，它比 IsAlive 属性能提供更多的特定信息。

前面说过，一个应用程序域中可能包括多个上下文，而通过CurrentContext可以获取线程当前的上下文。

CurrentThread是最常用的一个属性，它是用于获取当前运行的线程。

**2.2.4 System.Threading.Thread的方法**

Thread 中包括了多个方法来控制线程的创建、挂起、停止、销毁，以后来的例子中会经常使用。

Abort()：终止本线程。

GetDomain()：返回当前线程正在其中运行的当前域。

GetDomainId()：返回当前线程正在其中运行的当前域Id。

Interrupt()：中断处于 WaitSleepJoin 线程状态的线程

Join()：已重载。 阻塞调用线程，直到某个线程终止时为止。

Resume()：继续运行已挂起的线程。

Start()：执行本线程

Suspend()：挂起当前线程，如果当前线程已属于挂起状态则此不起作用

Sleep()：把正在运行的线程挂起一段时间

2.3 前台线程和后台线程

前台线程：只有所有的前台线程都结束，应用程序才能结束。默认情况下创建的线程  
              都是前台线程  
后台线程：只要所有的前台线程结束，后台线程自动结束。通过Thread.IsBackground设置后台线程。必须在调用Start方法之前设置线程的类型，否则一旦线程运行，将无法改变其类型。

后台线程一般用于处理不重要的事情，应用程序结束时，后台线程是否执行完成对整个应用程序没有影响。如果要执行的事情很重要，需要将线程设置为前台线程。

2.4 线程同步

所谓同步：是指在某一时刻只有一个线程可以访问变量。  
如果不能确保对变量的访问是同步的，就会产生错误。  
c#为同步访问变量提供了一个非常简单的方式，即使用c#语言的关键字Lock，它可以把一段代码定义为互斥段，互斥段在一个时刻内只允许一个线程进入执行，而其他线程必须等待。在c#中，关键字Lock定义如下：  
Lock(expression)  
{  
   statement\_block  
}

expression代表你希望跟踪的对象：  
           如果你想保护一个类的实例，一般地，你可以使用this；  
           如果你想保护一个静态变量（如互斥代码段在一个静态方法内部），一般使用类名就可以了  
而statement\_block就算互斥段的代码，这段代码在一个时刻内只可能被一个线程执行。

2.5 跨线程访问

使用窗体应用程序创建项目，创建按钮和文本框，使用线程thread来给文本框赋值，会发现报错：

System.InvalidOperationException:“线程间操作无效: 从不是创建控件“textBox1”的线程访问它。”

产生错误的原因：textBox1是由主线程创建的，thread线程是另外创建的一个线程，在.NET上执行的是托管代码，C#强制要求这些代码必须是线程安全的，即不允许跨线程访问Windows窗体的控件。

解决方案：

1、在窗体的加载事件中，将C#内置控件(Control)类的CheckForIllegalCrossThreadCalls属性设置为false，屏蔽掉C#编译器对跨线程调用的检查。(也就是在构造函数中 InitializeComponent完成后设置)

使用上述的方法虽然可以保证程序正常运行并实现应用的功能，但是在实际的软件开发中，做如此设置是不安全的（不符合.NET的安全规范），在产品软件的开发中，此类情况是不允许的。如果要在遵守.NET安全标准的前提下，实现从一个线程成功地访问另一个线程创建的空间，要使用C#的方法回调机制。

2、使用回调函数

回调实现的一般过程：

 C#的方法回调机制，也是建立在委托基础上的，下面给出它的典型实现过程。

（1）、定义、声明回调。

//定义回调

private delegate void DoSomeCallBack(Type para);

//声明回调

DoSomeCallBack doSomaCallBack

可以看出，这里定义声明的“回调”（doSomaCallBack）其实就是一个委托。

（2）、初始化回调方法。

doSomeCallBack=new DoSomeCallBack(DoSomeMethod);

所谓“初始化回调方法”实际上就是实例化刚刚定义了的委托，这里作为参数的DoSomeMethod称为“回调方法”，它封装了对另一个线程中目标对象（窗体控件或其他类）的操作代码。

（3）、触发对象动作

Opt  obj.Invoke(doSomeCallBack,arg);

其中Opt obj为目标操作对象，在此假设它是某控件，故调用其Invoke方法。Invoke方法签名为：

object  Control.Invoke(Delegate  method,params  object[] args);

它的第一个参数为委托类型，可见“触发对象动作”的本质，就是把委托doSomeCallBack作为参数传递给控件的Invoke方法，这与委托的使用方式是一模一样的。

最终作用于对象Opt obj的代码是置于回调方法体DoSomeMethod()中的，如下所示：

private void DoSomeMethod(type para)

{

     //方法体

    Opt obj.someMethod(para);

}

如果不用回调，而是直接在程序中使用“Opt obj.someMethod(para);”，则当对象Opt obj不在本线程（跨线程访问）时就会发生上面所示的错误。

从以上回调实现的一般过程可知：C#的回调机制，实质上是委托的一种应用。在C#网络编程中，回调的应用是非常普遍的，有了方法回调，就可以在.NET上写出线程安全的代码了。

 2.6 终止线程

若想终止正在运行的线程，可以使用Abort（）方法。

三、同步和异步

同步和异步是对方法执行顺序的描述。

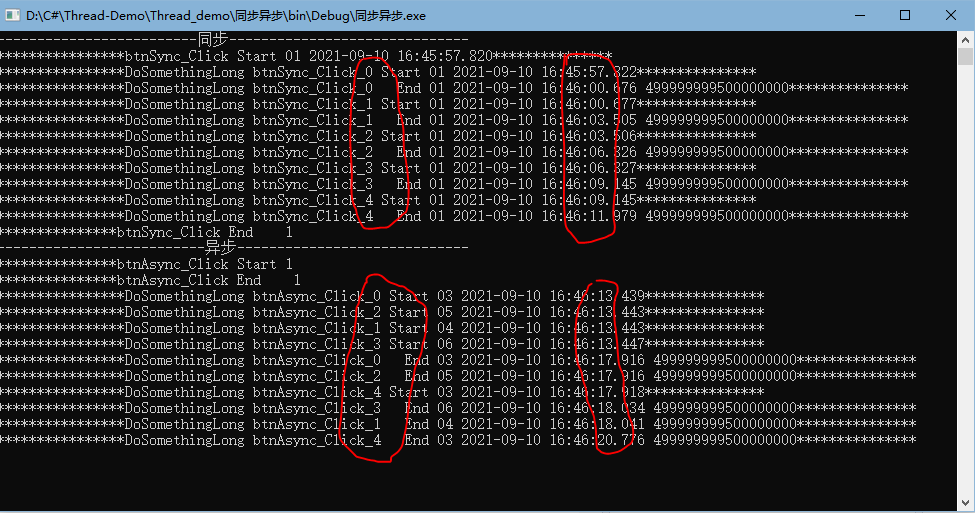
同步：等待上一行完成计算之后，才会进入下一行。

例如：请同事吃饭，同事说很忙，然后就等着同事忙完，然后一起去吃饭。

异步：不会等待方法的完成，会直接进入下一行，是非阻塞的。

例如：请同事吃饭，同事说很忙，那同事先忙，自己去吃饭，同事忙完了他自己去吃饭。

同步方法执行是有序的，异步方法执行是无序的。异步方法无序包括启动无序和结束无序。启动无序是因为同一时刻向操作系统申请线程，操作系统收到申请以后，返回执行的顺序是无序的，所以启动是无序的。结束无序是因为虽然线程执行的是同样的操作，但是每个线程的耗时是不同的，所以结束的时候不一定是先启动的线程就先结束。从上面同步方法中可以清晰的看出：btnSync\_Click\_0执行时间耗时不到3秒，而btnSync\_Click\_1执行时间耗时超过了3秒。可以想象体育比赛中的跑步，每位运动员听到发令枪起跑的顺序不同，每位运动员花费的时间不同，最终到达终点的顺序也不同。



从上图可以看出，同步执行方法有序，但是所用时间长，效率低，不能随意拖动窗体

而异步执行方法无序，所用时间短，效率高，且在异步方法执行时可以随意拖动窗体

总结一下同步方法和异步方法的区别：

1、同步方法由于主线程忙于计算，所以会卡住界面。

      异步方法由于主线程执行完了，其他计算任务交给子线程去执行，所以不会卡住界面，用户体验性好。

2、同步方法由于只有一个线程在计算，所以执行速度慢。

      异步方法由多个线程并发运算，所以执行速度快，但并不是线性增长的（资源可能不够）。多线程也不是越多越好，只有多个独立的任务同时运行，才能加快速度。

3、同步方法是有序的。

      异步多线程是无序的：启动无序，执行时间不确定，所以结束也是无序的。一定不要通过等待几毫秒的形式来控制线程启动/执行时间/结束。所以：异步不需要控制sleep来让线程等待。

异步执行比较快，效率高，但是有个缺点就是无序的，怎么样才能让它有序呢？答：使用回调

1. 回调
2. 既然异步多线程是无序的，那我们有没有什么办法可以解决无序的问题呢？办法当然是有的，那就是使用回调，.NET框架已经帮我们实现了回调：
3. BeginInvoke的第二个参数就是一个回调，那么AsyncCallback究竟是什么呢？F12查看AsyncCallback的定义：发现AsyncCallback就是一个委托，参数类型是IAsyncResult
4. 程序执行到BeginInvoke的时候，会申请一个基于线程池的线程，这个线程会完成委托的执行(在这里就是执行DoSomethingLong()方法)，在委托执行完以后，这个线程又会去执行callback回调的委托，执行callback委托需要一个IAsyncResult类型的参数，这个IAsyncResult类型的参数是如何来的呢？鼠标右键放到BeginInvoke上面，查看返回值：

发现BeginInvoke的返回值就是IAsyncResult类型的。那么这个返回值是不是就是callback委托的参数呢？

这里可以看出BeginInvoke的返回值就是callback委托的参数。

现在我们可以使用回调解决异步多线程无序的问题了。

4、获取委托异步调用的返回值

使用EndInvoke可以获取委托异步调用的返回值

|  |
| --- |
| IAsyncResult BeginInvoke(AsyncCallback callback,@object object)：callback.AsyncState = object |
| delegate void AsyncCallback(IAsyncResult asyncResult):入参就是BeginInvoke的返回值(定义、执行) |
| Func<T> EndInvoke(IAsyncResult asyncResult):入参就是BeginInvoke的返回值(得到、获取) |

**注意Action和Func它两都是委托类型方法，只不过Action不具有返回值且只有一个参数而Func是不具有参数同时返回值是由TResult来决定的也就是Func<T>！**

**提问1：简要说明同步和异步的区别？**

同步：去食堂排队买饭，顺序执行

异步：去食堂不排队，谁去的早谁先买（抢资源）。

#### 提问2：保证线程安全的几种方法和机制？

加lock，锁机制，保证当前方法或实例只能有一个线程进入，等该线程执行完毕下个线程才能访问资源。还有线程的优先级也会引发阻塞，所以一般不要设置优先级除非有特殊需求。除此之外还有async和await。

如果你的代码所在的进程中有多个线程在同时运行，而这些线程可能会同时运行这段代码。如果每次运行结果和单线程运行的结果是一样的，而且其他的变量的值也和预期的是一样的。 线程安全问题都是由全局变量及静态变量引起的

为了保证多线程情况下，访问静态变量的安全，可以用锁机制来保证Lock

需要注意的是，Lock只能锁住一个引用类型的对象。另外，除了锁机制外，高版本的C#中加入了**async**和**await**方法来保证线程安全

若每个线程中对全局变量、静态变量只有读操作，而无写操作，一般来说，这个全局变量是线程安全的；若有多个线程同时对一个变量执行读写操作，一般都需要考虑线程同步，否则就可能影响线程安全

线程的优先级：当线程之间争夺CPU时间时，CPU按照线程的优先级给予服务。高优先级的线程可以完全阻止低优先级的线程执行。.NET为线程设置了Priority属性来定义线程执行的优先级别，里面包含5个选项，其中Normal是默认值。除非系统有特殊要求，否则不应该随便设置线程的优先级别。随意设置线程优先级也会导致线程阻塞的问题。

async 和 await

如果使用 [async](https://docs.microsoft.com/zh-cn/dotnet/csharp/language-reference/keywords/async) 修饰符将某种方法指定为异步方法，即启用以下两种功能。

标记的异步方法可以使用 [await](https://docs.microsoft.com/zh-cn/dotnet/csharp/language-reference/operators/await) 来指定暂停点。 await 运算符通知编译器异步方法：在等待的异步过程完成后才能继续通过该点。 同时，控制返回至异步方法的调用方。

异步方法在 await 表达式执行时暂停并不构成方法退出，只会导致 finally 代码块不运行。

标记的异步方法本身可以通过调用它的方法等待。

异步方法通常包含 await 运算符的一个或多个实例，但缺少 await 表达式也不会导致生成编译器错误。 如果异步方法未使用 await 运算符标记暂停点，则该方法会作为同步方法执行，即使有 async 修饰符，也不例外。 编译器将为此类方法发布一个警告。