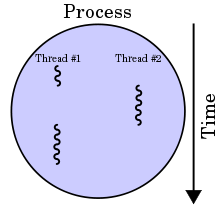
c# 温故而知新： 线程篇(一)

**Thread**

目录：

* 目录：
* [1 线程基础的简单介绍](https://www.cnblogs.com/JimmyZheng/archive/2012/06/10/2543143.html" \l "no1)
* [2 线程同步与线程异步的简单介绍](https://www.cnblogs.com/JimmyZheng/archive/2012/06/10/2543143.html" \l "no2)
* [3 前台线程与后台线程的简单介绍](https://www.cnblogs.com/JimmyZheng/archive/2012/06/10/2543143.html" \l "no3)
* [4 细说下Thread 最为关键的构造函数](https://www.cnblogs.com/JimmyZheng/archive/2012/06/10/2543143.html" \l "no4)
* [5 细说下Thread 的 Sleep方法](https://www.cnblogs.com/JimmyZheng/archive/2012/06/10/2543143.html" \l "no5)
* [6 细说下Thread 的 join 方法](https://www.cnblogs.com/JimmyZheng/archive/2012/06/10/2543143.html" \l "no6)
* [7 细说下Thread 的 Abort和 Interrupt方法](https://www.cnblogs.com/JimmyZheng/archive/2012/06/10/2543143.html" \l "no7)
* [8 细说下Thread 的 Suspend，Resume方法](https://www.cnblogs.com/JimmyZheng/archive/2012/06/10/2543143.html" \l "no8)
* [9 简单了解下Thread 的 一些重要属性](https://www.cnblogs.com/JimmyZheng/archive/2012/06/10/2543143.html" \l "no9)
* [10  简单示例](https://www.cnblogs.com/JimmyZheng/archive/2012/06/10/2543143.html" \l "no10)
* [多线程从一个图片中截取部分图片](https://www.cnblogs.com/JimmyZheng/archive/2012/06/10/2543143.html" \l "no10)
* [11 本章总结](https://www.cnblogs.com/JimmyZheng/archive/2012/06/10/2543143.html" \l "n11)

1 线程基础的简单介绍



首先让我们翻开书本来了解下线程的一些基础知识：

|  |
| --- |
| 1 线程有时被称为轻量级进程，是程序执行流的最小单元  2 线程时由线程ID，当前指令指针(PC），寄存器集合和堆栈组成。  3 线程自身不能拥有系统资源，但是可以使用线程所属进程所占有的系统资源  4 线程可以创建和撤销另一个线程  5 线程可以拥有自身的状态，例如 运行状态，挂起状态，销毁释放状态等等  6 线程具有优先级，每个线程都分配了0-31 级别的其中一个优先级，数字越大，优先级越高，然而手动分配优先级过于复杂，  所以微软为我们的Thread类提供一个优先级的枚举，ThreadPriority枚举便是优先级枚举，我们可以利用thread.Priority属性来进行设置  7 线程开销，这个是个复杂的话题，希望有机会的话能够单独写一遍文章解释下 |

那么多线程有什么实际好处呢？

首先让我们了解下多线程的概念：一个程序或者进程中同时运行多个线程完成不同的工作

从概念中我们便可知道多线程的优点了

|  |
| --- |
| 1 能够实现并行操作，也就是说多个线程可以同时进行工作  2 利用多线程后许多复杂的业务或者是计算可以交给后台线程去完成，从而提高整体程序的性能  3 类似于第一条利用多线程可以达到异步的作用（注意，实现异步的一种方式是多线程） |

当然多线程也有一定的问题需要注意，那就是线程同步问题，关于这个问题我会今后的文章中详细说明

2 线程同步与线程异步的简单介绍

\*1 线程同步

关于线程同步的概念最简单的理解就是

同步方法调用在程序继续执行之前，需要等待同步方法执行完毕返回结果

|  |
| --- |
| 很有可能多个线程都会对一个资源进行访问,从而导致资源被破坏，所以必须采用线程的同步机制，例如为共享资源加锁  ，当其中一个线程占有了锁之后，其余线程均不能使用共享资源，只有等其释放锁之后，接下来的其中一个线程会占有该  锁，本系列会从Thread类开始讲起，以后多章都会讨论线程同步机制，例如锁机制，临界区，互斥，信号量 同步事件等待句柄； 等等 |

\*2 线程异步

线程异步指的是一个调用请求发送给被调用者，而调用者不用等待其结果的返回，一般异步执行的任务都需要比较长的时间，

所以为了不影响主线程的工作，可以使用多线程或者新开辟一个线程来实现异步，同样，异步和线程池也有着非常紧密的联系，

这点我会在今后有关线程池的文章中详细叙述，线程池和异步线程将在第二章中详细阐述下

3 前台线程与后台线程的简单介绍

前台线程：

诸如我们Console程序的主线程，wpf或者sliverlight的 界面线程等等，都属于前台线程，一旦前台线程奔溃或者终止，相应的后台

线程都会终止，本章中通过Thread类产生的线程默认都是前台线程，当然我们可以设置Thread的属性让该对象成为后台线程，必须

注意的是，一旦前台线程全部运行完毕，应用程序的进程也会释放，但是假设Console程序中main函数运行完毕，但是其中几个前台

线程还处在运行之中，那么这个Console程序的进程是不会释放的，仍然处于运行之中，直到所有的前台线程都释放为止

后台线程：

和前台线程唯一的区别是，后台线程更加默默无闻，甚至后台线程因某种情况，释放销毁时不会影响到进程，也就是说后台线程释放时

不会导致进程的释放

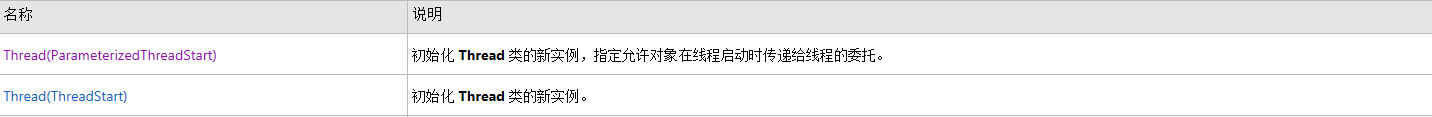
|  |
| --- |
| 用一个例子再来说明下前后台线程的区别：  有时我们打开outlook 后接受邮件时，程序会失去响应或被卡住，这时候我们去点击outlook时系统会提示 outlook 失去响应，是否等待或者关闭，  当我们点击关闭时，其实在程序中关于outlook的所有运行的前台线程被终止，导致了outlook被关闭了，其进程也随之释放消失。但是，当我们在  outlook中点击更新邮件时，后台线程会去收取邮件的工作，我们可以在此期间关闭 outlook接受新邮件的后台线程，而不会导致整个outlook的关闭 |

4 细说下Thread 最为关键的构造函数

相信大家再看过前几章对于线程的介绍后，对线程应该有一个温故的感觉，那么让我们开始对thread这个线程类进行深层次的研究下，

首先要启动一个线程必须将该线程将要做的任务告诉该线程，否则，线程会不知道干什么事导致线程无意义的开启，浪费系统资源，果然，

Thread类的构造函数提供了以下的版本



ThreadStart 和 ParameterThreadStart 参数都是委托，所以可以看出委托其实就是方法的抽象，前者用于不带参数的并且无返回值的

方法的抽象，后者是带object参数的方法的抽象，大家通过以下简单的方法注意下线程如何调用带参数的方法

[IMG_258](https://www.cnblogs.com/JimmyZheng/archive/2012/06/10/javascript:void(0);)

public class ThreadStartTest

{

//无参数的构造函数

Thread thread = new Thread(new ThreadStart(ThreadMethod));

//带有object参数的构造函数

Thread thread2 = new Thread(new ParameterizedThreadStart(ThreadMethodWithPara));

public ThreadStartTest()

{

//启动线程1 thread.Start();

//启动线程2

thread2.Start(new Parameter { paraName="Test" });

}

static void ThreadMethod()

{

//.... }

static void ThreadMethodWithPara(object o)

{

if (o is Parameter)

{

// (o as Parameter).paraName............. }

}

}

public class Parameter

{

public string paraName { get; set; }

}

[IMG_259](https://www.cnblogs.com/JimmyZheng/archive/2012/06/10/javascript:void(0);)

不带参数的方法似乎很简单的能被调用，只要通过第一个构造函数便行，对于带参数的方法，大家注意下参数是如何传入线程所调用的方法，

当启动线程时，参数通过thread.Start方法传入，于是我们便成功启动了thread线程，大伙可千万不要小看基础啊，往往在复杂的项目中很多

就是因为一些基础导致，所以一定不要忽视它。。。

5 细说下Thread 的 Sleep方法

  话说微软对Thread.Sleep方法的解释过于简单，导致许多人会误认为这个方法并不重要，其实这是错误的，其实线程是非常复杂的，

  而且我们围绕这个方法来温故下windows系统对于CPU竞争的策略：

  所谓抢占式操作系统，就是说如果一个进程得到了 CPU 时间，除非它自己放弃使用 CPU ，否则将完全霸占 CPU 。因此可以看出，

  在抢占式操作系统中，操作系统假设所有的进程都是“人品很好”的，会主动退出 CPU 。

  发现写到这里貌似真的已经比较复杂了，由于本人对操作系统底层的知识比较匮乏，决定还是引用下别人的理解，顺便自己也学习下

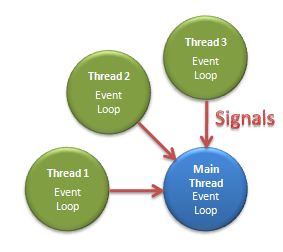
引用：

|  |
| --- |
| 假设有源源不断的蛋糕（源源不断的时间），一副刀叉（一个CPU），10个等待吃蛋糕的人（10 个进程）。如果是 Unix 操作系统来负责分蛋糕，  那么他会这样定规矩：每个人上来吃 1 分钟，时间到了换下一个。最后一个人吃完了就再从头开始。于是，不管这10个人是不是优先级不同、饥饿  程度不同、饭量不同，每个人上来的时候都可以吃 1 分钟。当然，如果有人本来不太饿，或者饭量小，吃了30秒钟之后就吃饱了，那么他可以跟操  作系统说：我已经吃饱了（挂起）。于是操作系统就会让下一个人接 着来。如果是 Windows 操作系统来负责分蛋糕的，那么场面就很有意思了。  他会这样定规矩：我会根据你们的优先级、饥饿程度去给你们每个人计算一个优先级。优先级最高的那个人，可 以上来吃蛋糕——吃到你不想吃为止。  等这个人吃完了，我再重新根据优先级、饥饿程度来计算每个人的优先级，然后再分给优先级最高的那个人。这样看来，这个 场面就有意思了——  可能有些人是PPMM，因此具有高优先级，于是她就可以经常来吃蛋糕。可能另外一个人的优先级特别低，于是好半天了才轮到他一次（因为 随着时间  的推移，他会越来越饥饿，因此算出来的总优先级就会越来越高，因此总有一天会轮到他的）。而且，如果一不小心让一个大胖子得到了刀叉，因为他  饭量 大，可能他会霸占着蛋糕连续吃很久很久，导致旁边的人在那里咽口水。。。而且，还可能会有这种情况出现：操作系统现在计算出来的结果，是  5号PPMM总优 先级最高——高出别人一大截。因此就叫5号来吃蛋糕。5号吃了一小会儿，觉得没那么饿了，于是说“我不吃了”（挂起）。因此操作系  统就会重新计算所有人的 优先级。因为5号刚刚吃过，因此她的饥饿程度变小了，于是总优先级变小了；而其他人因为多等了一会儿，饥饿程度都变大了，  所以总优先级也变大了。不过这时 候仍然有可能5号的优先级比别的都高，只不过现在只比其他的高一点点——但她仍然是总优先级最高的啊。因此操作  系统就会说：5号mm上来吃蛋糕……（5号 mm心里郁闷，这不刚吃过嘛……人家要减肥……谁叫你长那么漂亮，获得了那么高的优先级）。那么，  Thread.Sleep 函数是干吗的呢？还用刚才的分蛋糕的场景来描述。上面的场景里面，5号MM在吃了一次蛋糕之后，觉得已经有8分饱了，她觉得在未来  的半个小时之内都不想再 来吃蛋糕了，那么她就会跟操作系统说：在未来的半个小时之内不要再叫我上来吃蛋糕了。这样，操作系统在随后的半个小时  里面重新计算所有人总优先级的时候， 就会忽略5号mm。Sleep函数就是干这事的，他告诉操作系统“在未来的多少毫秒内我不参与CPU竞争”。 |

6 细说下Thread 的 join 方法

为什么我要把Thread.Join()方法单独细说下，个人认为join方法非常重要，

在细说前我想再次强调下主线程和子线程的区别:



首先大家肯定知道在Console程序中，主线程自上而下着运行着main函数，假如我们在main函数中新增一个线程thread对象的话，

也就是说，在主线程中再开启一个子线程，同时子线程和主线程可以同时工作（前提是子线程使用Start方法），同理，假如我在这

个子线程中再开辟一个属于这个子线程的子线程，同理这3个爷爷，父亲，儿子线程也可以使用Start()方法一起工作，假如在主线

程中添加2个thread对象并开启，那么这2 线程便属于同一层次的线程（兄弟线程）（和优先级无关,只同一位置层次上的兄弟），

有可能上述的让你觉得郁闷或者难以理解？没关系看简单例子就能够理解了

[IMG_261](https://www.cnblogs.com/JimmyZheng/archive/2012/06/10/javascript:void(0);)

public static void ShowFatherAndSonThread(Thread grandFatherThread)

{

Console.WriteLine("爷爷主线程名：{0}", grandFatherThread.Name);

Thread brotherThread = new Thread(new ThreadStart(() => { Console.WriteLine("兄弟线程名：{0}", Thread.CurrentThread.Name); }));

Thread fatherThread = new Thread(new ThreadStart(

() =>

{

Console.WriteLine("父亲线程名：{0}", Thread.CurrentThread.Name);

Thread sonThread = new Thread(new ThreadStart(() =>

{

Console.WriteLine("儿子线程名：{0}", Thread.CurrentThread.Name);

}));

sonThread.Name = "SonThread";

sonThread.Start();

}

));

fatherThread.Name = "FatherThread";

brotherThread.Name="BrotherThread";

fatherThread.Start();

brotherThread.Start();

}

[IMG_262](https://www.cnblogs.com/JimmyZheng/archive/2012/06/10/javascript:void(0);)

言归正传让我们温故下Jion方法,先看msdn中是怎么解释的：

继续执行标准的 COM 和 SendMessage 消息泵处理期间，阻塞调用线程，直到某个线程终止为止。

大家把注意力移到后面红色的部分，什么是“调用线程”呢？如果你理解上述线程关系的话,可能已经理解了，主线程（爷爷辈）的调用了父亲线程，

父亲线程调用了儿子线程，假设现在我们有一个奇怪的需求，必须开启爷爷辈和父亲辈的线程但是，爷爷辈线程必须等待父亲线程结束后再进行,

这该怎么办？ 这时候Join方法上场了，我们的目标是阻塞爷爷线程，那么后面的工作就明确了，让父亲线程（thread）对象去调用join方法就行

一下是个很简单的例子，让大家再深入理解下。

[IMG_263](https://www.cnblogs.com/JimmyZheng/archive/2012/06/10/javascript:void(0);)

public static void ThreadJoin()

{

Console.WriteLine("我是爷爷辈线程,子线程马上要来工作了我得准备下让个位给他。");

Thread t1 = new Thread(

new ThreadStart

(

() =>

{

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

{

if (i == 0)

Console.WriteLine("我是父亲线层{0}, 完成计数任务后我会把工作权交换给主线程", Thread.CurrentThread.Name);

else

{

Console.WriteLine("我是父亲线层{0}, 计数值:{1}", Thread.CurrentThread.Name, i);

}

Thread.Sleep(1000);

}

}

)

);

t1.Name = "线程1";

t1.Start();

//调用join后调用线程被阻塞  
 t1.Join();

Console.WriteLine("终于轮到爷爷辈主线程干活了");

}

[IMG_264](https://www.cnblogs.com/JimmyZheng/archive/2012/06/10/javascript:void(0);)

代码中当父亲线程启动后会立即进入Jion方法，这时候调用该线程爷爷辈线程被阻塞，直到父亲线程中的方法执行完毕为止，最后父亲线程将控制

权再次还给爷爷辈线程，输出最后的语句。聪明的你肯定会问：兄弟线程怎么保证先后顺序呢？很明显如果不使用join，一并开启兄弟线程后结果

是随机的不可预测的（暂时不考虑线程优先级），但是我们不能在兄弟线程全都开启后使用join，这样阻塞了父亲线程，而对兄弟线程是无效的，

其实我们可以变通一下，看以下一个很简单的例子：

[IMG_265](https://www.cnblogs.com/JimmyZheng/archive/2012/06/10/javascript:void(0);)

public static void ThreadJoin2()

{

IList<Thread> threads = new List<Thread>();

for (int i = 0; i < 3; i++)

{

Thread t = new Thread(

new ThreadStart(

() =>

{

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

{

if (j == 0)

Console.WriteLine("我是线层{0}, 完成计数任务后我会把工作权交换给其他线程", Thread.CurrentThread.Name);

else

{

Console.WriteLine("我是线层{0}, 计数值:{1}", Thread.CurrentThread.Name, j);

}

Thread.Sleep(1000);

}

}));

t.Name = "线程" + i;

//将线程加入集合 threads.Add(t);

}

foreach (var thread in threads)

{

thread.Start();

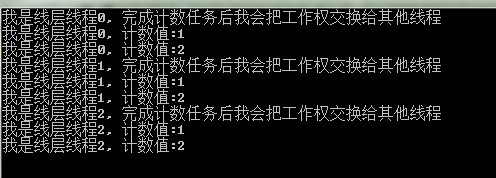
//每次按次序阻塞调用次方法的线程 thread.Join();

}

}

[IMG_266](https://www.cnblogs.com/JimmyZheng/archive/2012/06/10/javascript:void(0);)

 输出结果：



但是这样我们即便能达到这种效果，也会发现其中存在着不少缺陷：

1：必须要指定顺序

2：一旦一个运行了很久，后续的线程会一直等待很久

3: 很容易产生死锁

从前面2个例子能够看出 jion是利用阻塞调用线程的方式进行工作，我们可以根据需求的需要而灵活改变线程的运行顺序，但是在复杂的项目或业务中

对于jion方法的调试和纠错也是比较困难的。

7 细说下Thread 的 Abort和 Interrupt方法

Abort 方法：

其实 Abort 方法并没有像字面上的那么简单，释放并终止调用线程，其实当一个线程调用 Abort方法时，会在调用此方法的线程上引发一个异常：

ThreadAbortException ,让我们一步步深入下对这个方法的理解：

      1 首先我们尝试对主线程终止释放

[IMG_268](https://www.cnblogs.com/JimmyZheng/archive/2012/06/10/javascript:void(0);)

static void Main(string[] args)

{

try

{

Thread.CurrentThread.Abort();

}

catch

{

//Thread.ResetAbort();

Console.WriteLine("主线程接受到被释放销毁的信号");

Console.WriteLine( "主线程的状态:{0}",Thread.CurrentThread.ThreadState);

}

finally

{

Console.WriteLine("主线程最终被被释放销毁");

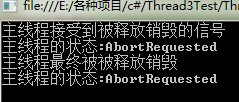
Console.WriteLine("主线程的状态:{0}", Thread.CurrentThread.ThreadState);

Console.ReadKey();

}

}

[IMG_269](https://www.cnblogs.com/JimmyZheng/archive/2012/06/10/javascript:void(0);)



从运行结果上看很容易看出当主线程被终止时其实报出了一个ThreadAbortException, 从中我们可以进行捕获，但是注意的是，主线程直到finally语

句块执行完毕之后才真正结束（可以仔细看下主线程的状态一直处于AbortRequest），如果你在finally语句块中执行很复杂的逻辑或者计算的话，那

么只有等待直到运行完毕才真正销毁主线程(也就是说主线程的状态会变成Aborted,但是由于是主线程所以无法看出).

2 尝试终止一个子线程

同样先看下代码：

[IMG_271](https://www.cnblogs.com/JimmyZheng/archive/2012/06/10/javascript:void(0);)

static void TestAbort()

{

try

{

Thread.Sleep(10000);

}

catch

{

Console.WriteLine("线程{0}接受到被释放销毁的信号",Thread.CurrentThread.Name);

Console.WriteLine("捕获到异常时线程{0}主线程的状态:{1}", Thread.CurrentThread.Name,Thread.CurrentThread.ThreadState);

}

finally

{

Console.WriteLine("进入finally语句块后线程{0}主线程的状态:{1}", Thread.CurrentThread.Name, Thread.CurrentThread.ThreadState);

}

}

Main:static void Main(string[] args)

{

Thread thread1 = new Thread(TestAbort);

thread1.Name = "Thread1";

thread1.Start();

Thread.Sleep(1000);

thread1.Abort();

thread1.Join();

Console.WriteLine("finally语句块后，线程{0}主线程的状态:{1}", thread1.Name, thread1.ThreadState);

Console.ReadKey();

}

[IMG_272](https://www.cnblogs.com/JimmyZheng/archive/2012/06/10/javascript:void(0);)



了解了主线程的销毁释放后，再来看下子线程的销毁释放的过程（Start->abortRequested->Aborted->Stop）,从最后输出的状态变化来看，

子线程thread1 的状态变化是十分清楚的，几乎和主线程的例子一致，唯一的区别是我们在 main方法中故意让主线程阻塞这样能看见thread 1

在 finally语句块后的状态

3，尝试对尚未启动的线程调用Abort

如果对一个尚未启动的线程调用Abort的话，一旦该线程启动就被停止了

4  尝试对一个挂起的线程调用Abort

如果在已挂起的线程上调用 Abort，则将在调用 Abort 的线程中引发 ThreadStateException，并将 AbortRequested 添加到被中止的线程的

ThreadState 属性中。直到调用 Resume 后，才在挂起的线程中引发 ThreadAbortException。如果在正在执行非托管代码的托管线程上调用 Abort，

则直到线程返回到托管代码才引发 ThreadAbortException。

 Interrupt 方法：

Interrupt 方法将当前的调用该方法的线程处于挂起状态，同样在调用此方法的线程上引发一个异常：ThreadInterruptedException，和Abort方法不同的是，

被挂起的线程可以唤醒

[IMG_274](https://www.cnblogs.com/JimmyZheng/archive/2012/06/10/javascript:void(0);)

static void Main(string[] args)

{

Thread thread1 = new Thread(TestInterrupt);

thread1.Name = "Thread1";

thread1.Start();

Thread.Sleep(1000);

thread1.Interrupt();

thread1.Join();

Console.WriteLine("finally语句块后，线程{0}主线程的状态:{1}", thread1.Name, thread1.ThreadState);

Console.ReadKey();

}

static void TestInterrupt()

{

try

{

Thread.Sleep(3000);

}

catch (ThreadInterruptedException e)

{

Console.WriteLine("线程{0}接受到被Interrupt的信号", Thread.CurrentThread.Name);

Console.WriteLine("捕获到Interrupt异常时线程{0}的状态:{1}", Thread.CurrentThread.Name, Thread.CurrentThread.ThreadState);

}

finally

{

Console.WriteLine("进入finally语句块后线程{0}的状态:{1}", Thread.CurrentThread.Name, Thread.CurrentThread.ThreadState);

}

}

[IMG_275](https://www.cnblogs.com/JimmyZheng/archive/2012/06/10/javascript:void(0);)



从代码中可以看出，当线程调用Interrupted后，它的状态是已中断的.这个状态对于正在执行join,sleep的线程,却改变了线程的运行结果

.因为它正在某一对象的休息室中,这时如果它的中断状态被改变,那么它就会抛出ThreadInterruptedException异常,意思就是这个线程不能再等待了,其意义就等同于唤醒它了。

让我们想象一下我们将一个线程设置了其长达1星期的睡眠时间，有时后必须唤醒它，上述方法就能实现这点

8 细说下Thread 的 Suspend，Resume方法

Suspend 和Resume方法很奥妙,前者将当前运行的线程挂起，后者能够恢复当钱被挂起的线程

[IMG_277](https://www.cnblogs.com/JimmyZheng/archive/2012/06/10/javascript:void(0);)

Thread thread1 = new Thread(TestSuspend);

Thread thread2 = new Thread(TestSuspend);

thread1.Name = "Thread1";

thread2.Name = "Thread2";

thread1.Start();

thread2.Start();

//假设在做一些事情

Thread.Sleep(1000);

Console.WriteLine("需要主线程帮忙了");  
 // throw new NullReferenceException("error!"); thread1.Resume();

thread2.Resume();

static void TestSuspend()

{

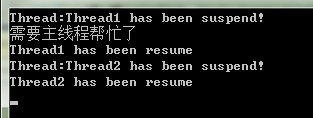
Console.WriteLine("Thread:{0} has been suspend!",Thread.CurrentThread.Name);

//这里讲当前线程挂起 Thread.CurrentThread.Suspend();

Console.WriteLine("{0} has been resume", Thread.CurrentThread.Name);

}

[IMG_278](https://www.cnblogs.com/JimmyZheng/archive/2012/06/10/javascript:void(0);)



如上代码，我们制造两个线程来实现Suspend和Resume的测试，(暂时不考虑临界区共享同步的问题)，TestSuspend方法便是两个线程的共用方法，

方法中我们获取当前运行该方法的线程，然后将其挂起操作，那么假设线程1先挂起了，线程1被中止当前的工作，面壁思过去了，可是这并不影响线程

2的工作，于是线程2也急匆匆的闯了进来，结果和线程1一样的悲剧，聪明的你肯定会问，谁能让线程1和线程2恢复工作？其实有很多方法能让他们恢

复工作，但是个人认为，在不创建新线程的条件下，被我们忽视的主线程做不住了，看到自己的兄弟面壁，心里肯定不好受，于是做完他自己的一系列

事情之后，他便去召唤这2个兄弟回来工作了，可是也许会有这种情况，主线程迫于自己的事情太多太杂而甚至报出了异常, 那么完蛋了，这两个线程永

远无法继续干活了，或者直接被回收。。。

这样这次把他们共享区上锁，上面部分的代码保持不变，这样会发生什么情况呢？

[IMG_280](https://www.cnblogs.com/JimmyZheng/archive/2012/06/10/javascript:void(0);)

static void TestSuspend()

{

lock (lockObj)

{

。。。。

}

}

[IMG_281](https://www.cnblogs.com/JimmyZheng/archive/2012/06/10/javascript:void(0);)

 （由于在TestSuspend方法中加入了锁，所以每次只允许一个线程工作，大伙不必在本文中深究锁机制，后续章节会给大家详细温故下）

尽然在thread2.resume()方法上报错了，仔细分析后发现在thread1离开共享区（testSuspend）方法之后刹那间，thread2进来了，与此同时，主线程

跑的太快了，导致thread2被挂起前去唤醒thread2，悲剧就这么发生了，其实修改这个bug很容易，只要判断下线程的状态，或者主线程中加一个Thread.Sleep()等等，

但是这种错误非常的严重，往往在很复杂的业务里让你发狂，所以微软决定放弃这两个方法，将他们归为过时方法，最后让大家看下微软那个深奥的解释，

相信看完上述例子后大家都能理解这个含义了

9 简单了解下Thread 的 一些常用的重要属性

1 CurrentThread

   获取到当前线程的对象

2 IsAlive

   判断线程是否处于激活状态

3 IsBackground

   设置该线程是否是后台线程，一旦设置true 的话，该线程就被标示为后台线程

再次强调下后台线程的终止不会导致进程的终止

4 IsThreadPoolThread

    只读属性标示该线程是否属于线程池的托管线程，一般我通过线程池创建的线程该属性都是true

5 Name

    获取到线程的名字，我们可以根据业务或者逻辑来自定义线程的名字

6 Priority

这个属性表示线程的优先级，我们可以用ThreadPriority这个枚举来设置这个属性

ThreadPriority包含有5个优先级大家了解下就行

10  Thread的简单示例

 在WPF中实现多线程从一个图片中截取部分图片

[IMG_282](https://www.cnblogs.com/JimmyZheng/archive/2012/06/10/javascript:void(0);)

using System;using System.Collections.Generic;using System.Linq;using System.Text;using System.Windows;using System.Windows.Controls;using System.Windows.Data;using System.Windows.Documents;using System.Windows.Input;using System.Windows.Media;using System.Windows.Media.Imaging;using System.Windows.Navigation;using System.Windows.Shapes;using System.Drawing;using System.Windows.Interop;using System.Threading;

namespace ImageFlip

{

/// <summary>

/// WPF 多线程将图片分割

/// </summary>

public partial class MainWindow : Window

{

BitmapSource source;

private object lockObj = new object();

public MainWindow()

{

InitializeComponent();

//首先获取图片

Bitmap orginalImage = new Bitmap(@"G:\Picture\Tamriel\_4E.png");

//创建线程1

Thread t1 = new Thread(new ParameterizedThreadStart

(

obj =>

{

//WPF中使用多线程的话最后一定要返回UI线程，否则操作界面控件时会报错

//BeginInvoke方法便是返回UI线程的方法

this.Dispatcher.BeginInvoke((Action)(() =>

{

//通过Parameter类的属性裁剪图片 ClipImageAndBind(obj);

//图片的部分绑定到页面控件

this.TestImage1.Source = source;

}));

}

));

//创建线程2

Thread t2 = new Thread(new ParameterizedThreadStart

(

obj =>

{

//WPF中使用多线程的话最后一定要返回UI线程，否则操作界面控件时会报错

//BeginInvoke方法便是返回UI线程的方法

this.Dispatcher.BeginInvoke((Action)(() =>

{

//通过Parameter类的属性裁剪图片 ClipImageAndBind(obj);

//图片的部分绑定到页面控件

this.TestImage2.Source = source;

//尝试将线程1的启动逻辑放在线程2所持有的方法中

// t1.Start(new Parameter { OrginalImage = orginalImage, ClipHeight = 500, ClipWidth = 500, StartX = 0, StartY = 0 }); }));

}

));

t2.Start(new Parameter { OrginalImage = orginalImage, ClipHeight = 500, ClipWidth = 500, StartX = orginalImage.Width - 500, StartY = orginalImage.Height - 500 });

//尝试下注释掉t2.join方法后是什么情况,其实注释掉之后，两个线程会一起工作，

//去掉注释后，界面一直到两个图片部分都绑定完成后才出现

//t2.Join();

t1.Start(new Parameter { OrginalImage = orginalImage, ClipHeight = 500, ClipWidth = 500, StartX = 0, StartY = 0 });

}

/// <summary>

/// 根据参数类进行剪裁图片，加锁防止共享资源被破坏

/// </summary>

/// <param name="para">Parameter类对象</param>

private void ClipImageAndBind(object para)

{

lock (lockObj)

{

Parameter paraObject = (para as Parameter);

source = this.ClipPartOfImage(paraObject);

Thread.Sleep(5000);

}

}

/// <summary>

/// 具体裁剪图片，大家不必在意这个方法，关键是线程的使用

/// </summary>

/// <param name="para">Parameter</param>

/// <returns>部分图片</returns>

private BitmapSource ClipPartOfImage(Parameter para)

{

if (para == null) { throw new NullReferenceException("para 不能为空"); }

if (para.OrginalImage == null) { throw new NullReferenceException("OrginalImage 不能为空"); }

System.Drawing.Rectangle rect = new System.Drawing.Rectangle(para.StartX, para.StartY, para.ClipWidth, para.ClipHeight);

var bitmap2 = para.OrginalImage.Clone(rect, para.OrginalImage.PixelFormat) as Bitmap;

return ChangeBitmapToBitmapSource(bitmap2);

}

private BitmapSource ChangeBitmapToBitmapSource(Bitmap bmp)

{

BitmapSource returnSource;

try

{

returnSource = Imaging.CreateBitmapSourceFromHBitmap(bmp.GetHbitmap(), IntPtr.Zero, Int32Rect.Empty, BitmapSizeOptions.FromEmptyOptions());

}

catch

{

returnSource = null;

}

return returnSource;

}

}

/// <summary>

/// 参数类

/// </summary>

public class Parameter

{

public Bitmap OrginalImage { get; set; }

public int StartX { get; set; }

public int StartY { get; set; }

public int ClipWidth { get; set; }

public int ClipHeight { get; set; }

}

}

[IMG_283](https://www.cnblogs.com/JimmyZheng/archive/2012/06/10/javascript:void(0);)

前台界面：

[IMG_284](https://www.cnblogs.com/JimmyZheng/archive/2012/06/10/javascript:void(0);)

<Window x:Class="ImageFlip.MainWindow"

xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"

xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"

Title="MainWindow" Height="350" Width="525">

<Grid>

<Grid.ColumnDefinitions>

<ColumnDefinition></ColumnDefinition>

<ColumnDefinition></ColumnDefinition>

</Grid.ColumnDefinitions>

<Image x:Name="TestImage1" Grid.Column="0"></Image>

<Image x:Name="TestImage2" Grid.Column="1"></Image>

</Grid></Window>

[IMG_285](https://www.cnblogs.com/JimmyZheng/archive/2012/06/10/javascript:void(0);)

11 本章总结

 本章介绍了线程一些简单的基础知识和对Thread类进行了详细的介绍，在以后的章节中我会逐步向大家介绍线程同步，异步线程等等有关线程的知识，

文中估计会有错误的地方也请大家海涵并且帮助指出，马上欧锦赛荷兰的比赛开始了，祝大家多多鼓励和关注！