

.NET 基础拾遗（5）：多线程开发基础 - 文章 - 伯乐在线



一、多线程编程的基本概念

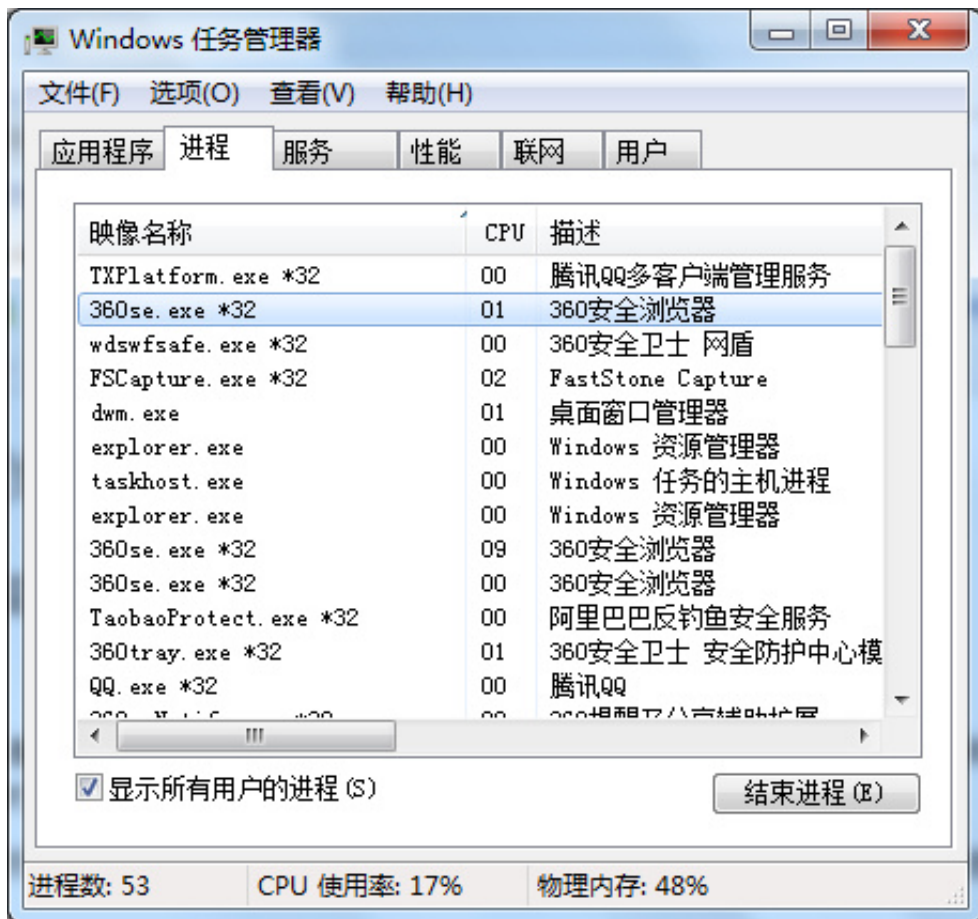
下面的一些基本概念可能和.NET的联系并不大，但对于掌握.NET中的多线程开发来说却十分重要。我们在开始尝试多线程开发前，应该对这些基础知识有所掌握，并且能够在操作系统层面理解多线程的运行方式。

1.1 操作系统层面的进程和线程

（1）进程

进程代表了操作系统上运行着的一个应用程序。进程拥有自己的程序块，拥有独占的资源 and 数据，并且可以被操作系统调度。But，即使是同一个应用程序，当被强制启动多次时，也会被安放到不同的进程之中单独运行。

直观地理解进程最好的方式就是通过进程管理器浏览，其中每条记录就代表了一个活动着的进程：

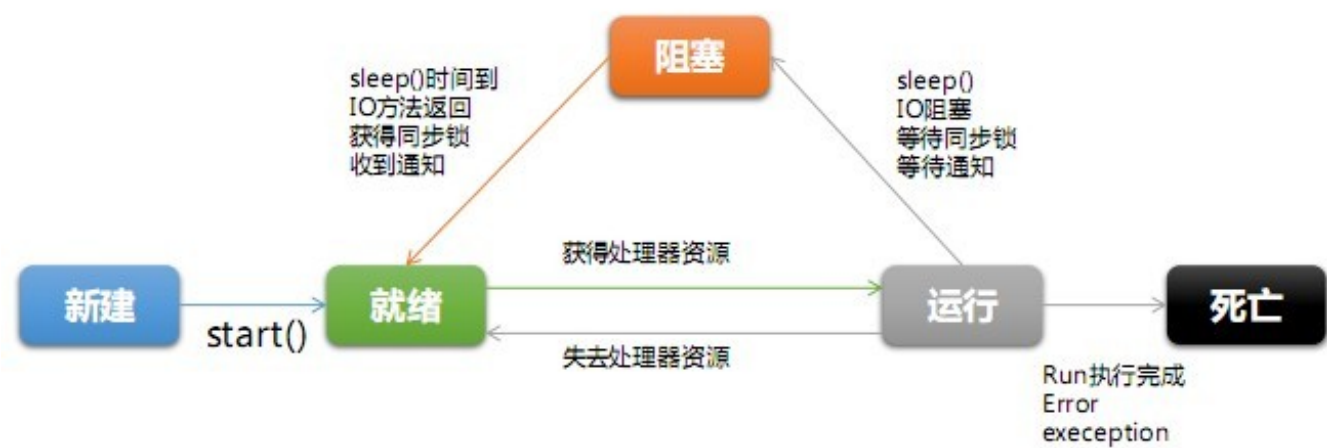


（2）线程

线程有时候也被称为轻量级进程，它的概念和进程十分相似，是一个可以被调度的单元，并且维护自己

的堆栈和上下文环境。线程是附属于进程的，一个进程可以包含1个或多个线程，并且同一进程内的多个线程共享一块内存块和资源。

由此看来，一个线程是一个操作系统可调度的基本单元，但是它的调度受限于该线程所属的进程，也就是说操作系统首先决定执行下一个执行的进程，进而才会调度该进程内的线程。一个线程的基本生命周期如下图所示：



（3）进程和线程的区别

最大的区别在于隔离性，每个进程都会被单独隔离（进程拥有自己的内存、资源和运行数据，一个进程的崩溃不会影响到其他进程，因此进程间的交互也相对困难），而同一进程内的所有线程则共享内存和资源，并且一个线程可以访问和结束同一进程内的其他线程。

1.2 多线程程序在操作系统中是并行执行的吗？

（1）线程的调度

在计算机系统发展的早期，操作系统层面不存在并行的概念，所有的应用程序都在排队等候一个单线程的队列之中，每个程序都必须等到前面的程序都安全执行完毕之后才能获得执行的权利，一个小小的错误将会导致操作系统上的所有程序的阻塞。在后来的操作系统中，逐渐产生了分时和进程、线程的概念。

多个线程由操作系统进行调度控制，决定何时运行哪个线程。所谓线程调度，是指操作系统决定如何安排线程执行顺序的算法。按常规分类，线程调度可以分为以下两种：

①抢占式调度

抢占式调度是指每个线程都只有极少的运行时间（在Windows NT内核模式下这个时间不会超过20ms），而当时间片用完时该线程就会被强制暂停，保存上下文并把运行权利交给下一个线程。这样调度的结果就是：所有的线程都在被不停地快速切换运行，使得用户感觉所有的线程都在并行运行。

②非抢占式调度

非抢占式调度是指某个线程在运行时不会被操作系统强制暂停，它可以持续地运行直到运行告一段落并

主动交出运行权。在这样的调度方式之下，线程的运行就是单队列的，并且可能产生恶意程序长期霸占运行权的情况。

PS：现在很多的操作系统（包括Windows在内），都同时采用了抢占式和非抢占式模式。对于那些优先级较高的线程，OS采用非抢占式来给予充分的时间运行，而对于普通的线程，则采用抢占式模式来快速地切换执行。

（2）线程的并行问题

在单核单CPU的硬件架构上，线程的并行运行完全是用户的主观体验。事实上，在任一时刻只可能存在一个处于运行状态的线程。但在多CPU或多核的架构上，情况则略有不同。多CPU多核的架构则允许系统完全并行地运行两个或多个无其他资源争用的线程，理论上这样的架构可以使运行性能整数倍地提高。

PS：微软公司曾经提出超线程技术，简单说来这是一种逻辑上模拟多CPU的技术，但实际上它们却共享物理处理器和缓存，超线程对性能的提高相当有限。

1.3 神马是纤程？

（1）纤程的概念

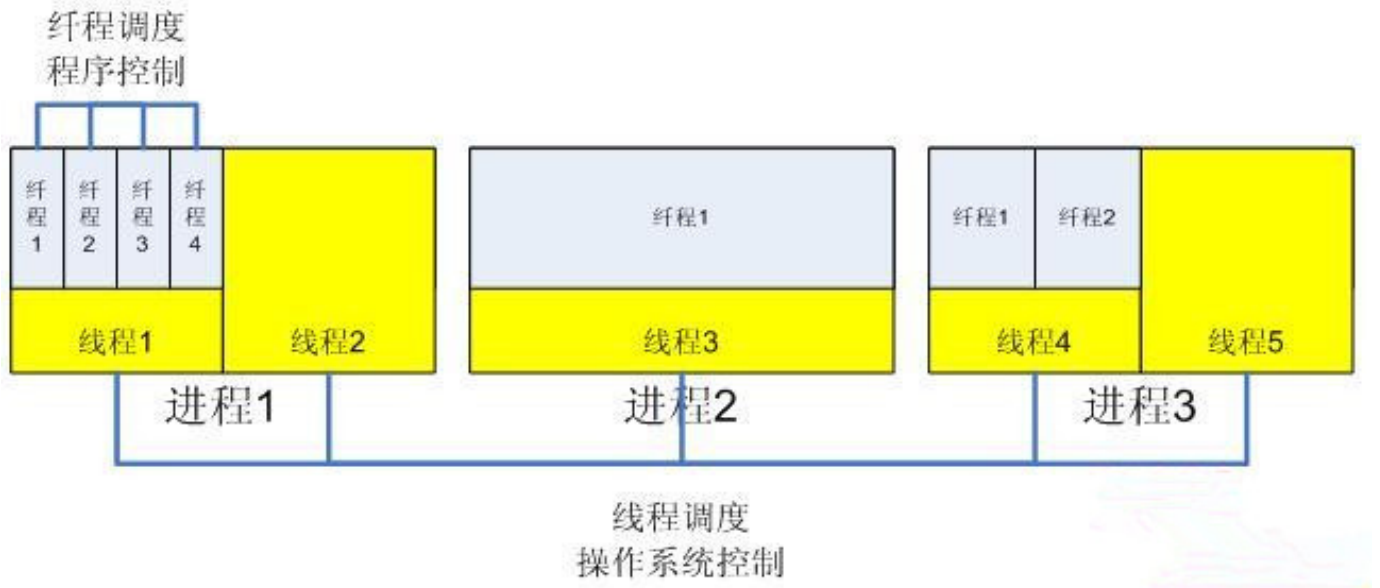
纤程是微软公司在Windows上提出的一个概念，其设计目的是用来方便地移植其他操作系统上的应用程序。一个线程可以拥有0个或多个纤程，一个纤程可以视为一个轻量级的线程，它拥有自己的栈和上下文状态。But，纤程的调度是由程序员编码控制的，当一个纤程所在线程得到运行时，程序员需要手动地决定运行哪一个纤程。

PS：事实上，Windows操作系统内核是不知道纤程的存在的，它只负责调度所有的线程，而纤程之所以成为操作系统的概念，是因为Windows提供了关于线程操作的Win32函数，能够方便地帮助程序员进行线程编程。

（2）纤程和线程的区别

纤程和线程最大的区别在于：线程的调度受操作系统的管理，程序员无法进行完全干涉。但纤程却完全受控于程序员本身，允许程序员对多任务进行自定义的调度和控制，因此纤程带给程序员很大的灵活性。

下图展示了进程、线程以及纤程三者之间的关系：



（3）纤程在.NET中的地位

需要谨记的是：.NET运行框架没有做出关于线程真实性的保证！也就是说，我们在.NET程序中新建的线程并不一定是操作系统层面上产生的一个真正线程。在.NET框架寄宿的情况下，一个程序中的线程很可能对应某个纤程。

PS：所谓CLR寄宿，就是指CLR运行在某个应用程序而非操作系统内。常见的寄宿例子是微软公司的SQL Server 2005。

二、.NET中的多线程编程

.NET为多线程编程提供了丰富的类型和机制，程序员需要做的就是掌握这些类型和机制的使用方法和运行原理。

2.1 如何在.NET程序中手动控制多个线程？

.NET中提供了多种实现多线程程序的方法，但最直接且灵活性最大的，莫过于主动创建、运行、结束所有线程。

（1）第一个多线程程序

.NET提供了非常直接的控制线程类型的类型：System.Threading.Thread类。使用该类型可以直观地创建、控制和结束线程。下面是一个简单的多线程程序：

C#

```
class Program
{
    static void Main(string[] args)
    {
        Console.WriteLine("进入多线程工作模式：");
        for (int i = 0; i < 10; i++)
        {
            Thread newThread = new Thread(Work);
            // 开启新线程
        }
    }
}
```

```
        newThread.Start();  
    }  
    Console.ReadKey();  
}  
  
static void Work()  
{  
    Console.WriteLine("线程开始");  
    // 模拟做了一些工作，耗费1s时间  
    Thread.Sleep(1000);  
    Console.WriteLine("线程结束");  
}
```

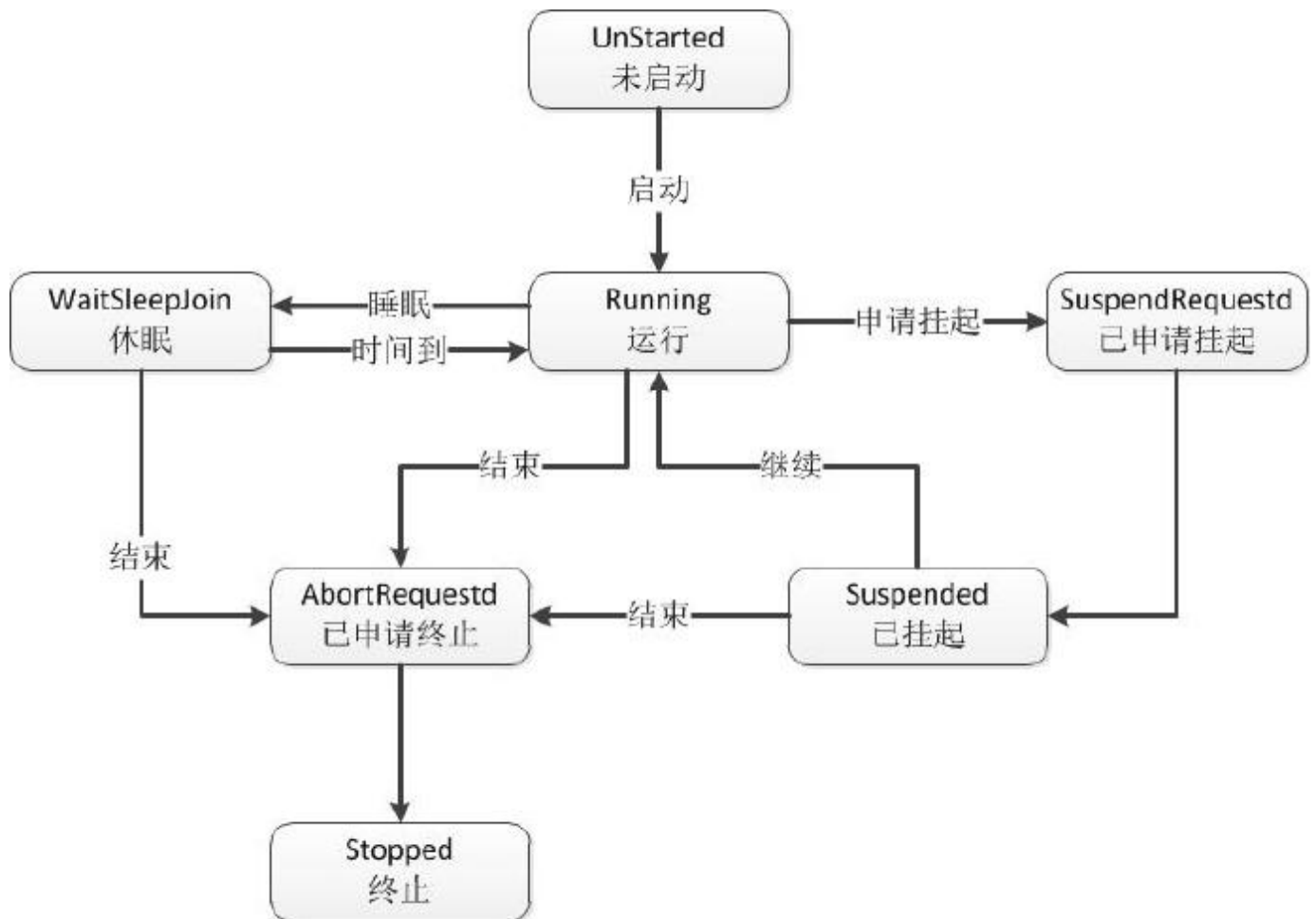
在主线程中，该代码创建了10个新的线程，这个10个线程的工作互不干扰，宏观上来看它们应该是并行运行的，执行的结果也证实了这一点：



PS：这里再次强调一点，当new了一个Thread类型对象并不意味着生成了一个线程，事实上线程的生成是在调用Thread的Start方法的时候。另外在之前的介绍中，这里的线程并不一定是操作系统层面上产生的一个真正线程！

(2) 控制线程的状态

很多时候，我们需要主动关心线程当前所处的状态。在任意时刻，.NET中的线程都会处于如下图所示的几个状态中的某一个状态上，该图也直观地展示了一个线程可能经过的状态转换过程（该图并没有列出所有的状态转换途径/原因）：



下面的示例代码则展示了我们如何手动地查看和控制一个线程的状态：

C#

```
class Program
```

```
{
```

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

```
    {
```

```
        Console.WriteLine("开始测试线程1");
```

```
        // 初始化一个线程 thread1
```

```
        Thread thread1 = new Thread(Work1);
```

```
        // 这时状态: UnStarted
```

```
        PrintState(thread1);
```

```
        // 启动线程
```

```
        Console.WriteLine("现在启动线程");
```

```
        thread1.Start();
```

```
        // 这时状态: Running
```

```
        PrintState(thread1);
```

```
        // 让线程飞一会 3s
```

```
        Thread.Sleep(3 * 1000);
```

```
        // 让线程挂起
```

```
        Console.WriteLine("现在挂起线程");
```

```
        thread1.Suspend();
```

```
        // 给线程足够的时间来挂起, 否则状态可能是SuspendRequested
```

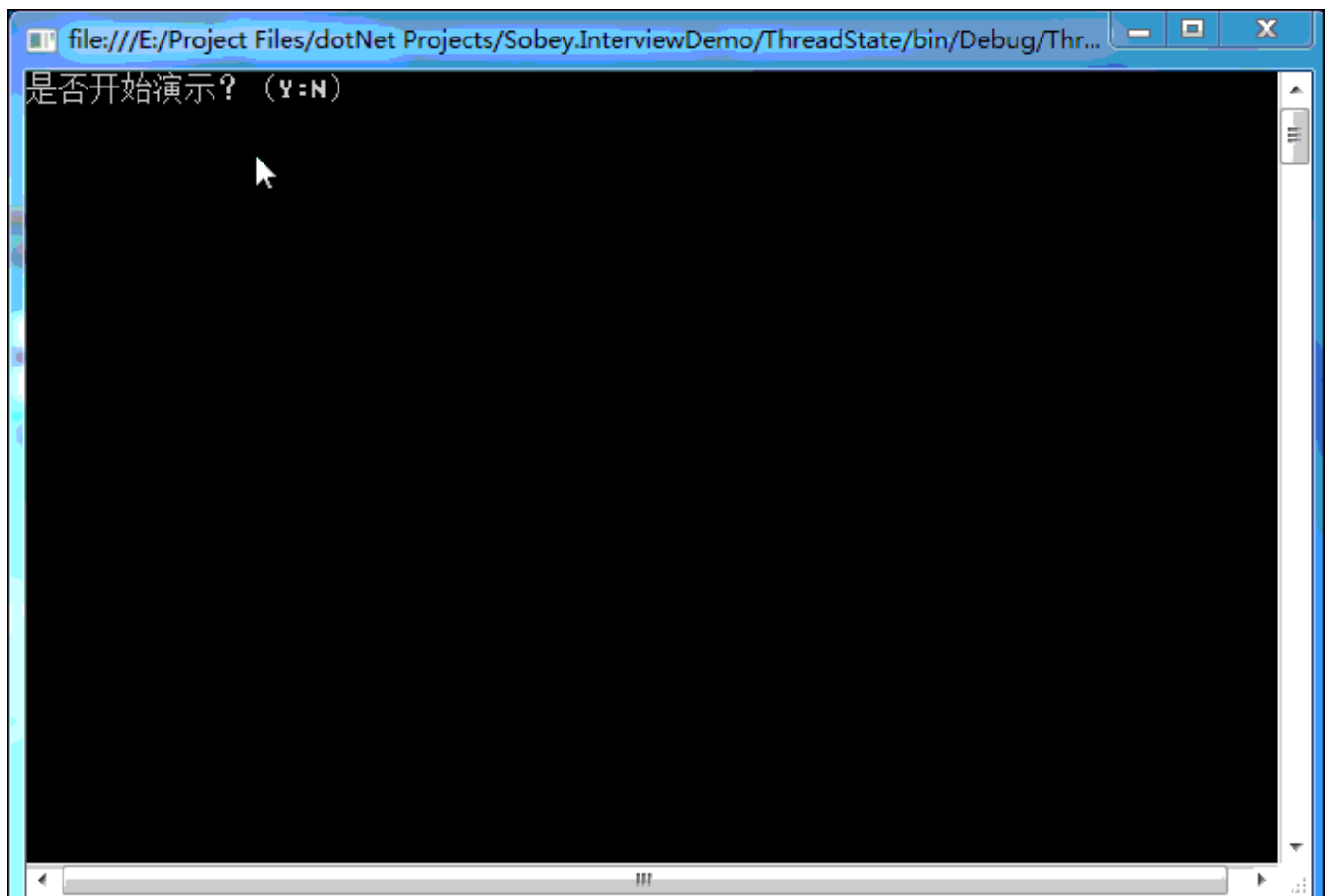
```
Thread.Sleep(1000);
// 这时状态: Suspend
PrintState(thread1);
// 继续线程
Console.WriteLine("现在继续线程");
thread1.Resume();
// 这时状态: Running
PrintState(thread1);
// 停止线程
Console.WriteLine("现在停止线程");
thread1.Abort();
// 给线程足够的时间来终止, 否则的话可能是AbortRequested
Thread.Sleep(1000);
// 这时状态: Stopped
PrintState(thread1);
Console.WriteLine("-----");
Console.WriteLine("开始测试线程2");
// 初始化一个线程 thread2
Thread thread2 = new Thread(Work2);
// 这时状态: UnStarted
PrintState(thread2);
// 启动线程
thread2.Start();
Thread.Sleep(2 * 1000);
// 这时状态: WaitSleepJoin
PrintState(thread2);
// 给线程足够的时间结束
Thread.Sleep(10 * 1000);
// 这时状态: Stopped
PrintState(thread2);
Console.ReadKey();
}

// 普通线程方法: 一直在运行从未被超越
private static void Work1()
{
    Console.WriteLine("线程运行中...");
    // 模拟线程运行, 但不改变线程状态
    // 采用忙等状态
    while (true) { }
}

// 文艺线程方法: 运行10s就结束
private static void Work2()
```

```
{  
    Console.WriteLine("线程开始睡眠：");  
    // 睡眠10s  
    Thread.Sleep(10 * 1000);  
    Console.WriteLine("线程恢复运行");  
}  
// 打印线程的状态  
private static void PrintState(Thread thread)  
{  
    Console.WriteLine("线程的状态是：{0}",  
thread.ThreadState.ToString());  
}  
}
```

上述代码的执行结果如下图所示：



PS：为了演示方便，上述代码刻意地使线程处于各个状态并打印出来。在.NET Framework 4.0 及之后的版本中，已经不再鼓励使用线程的挂起状态，以及Suspend和Resume方法了。

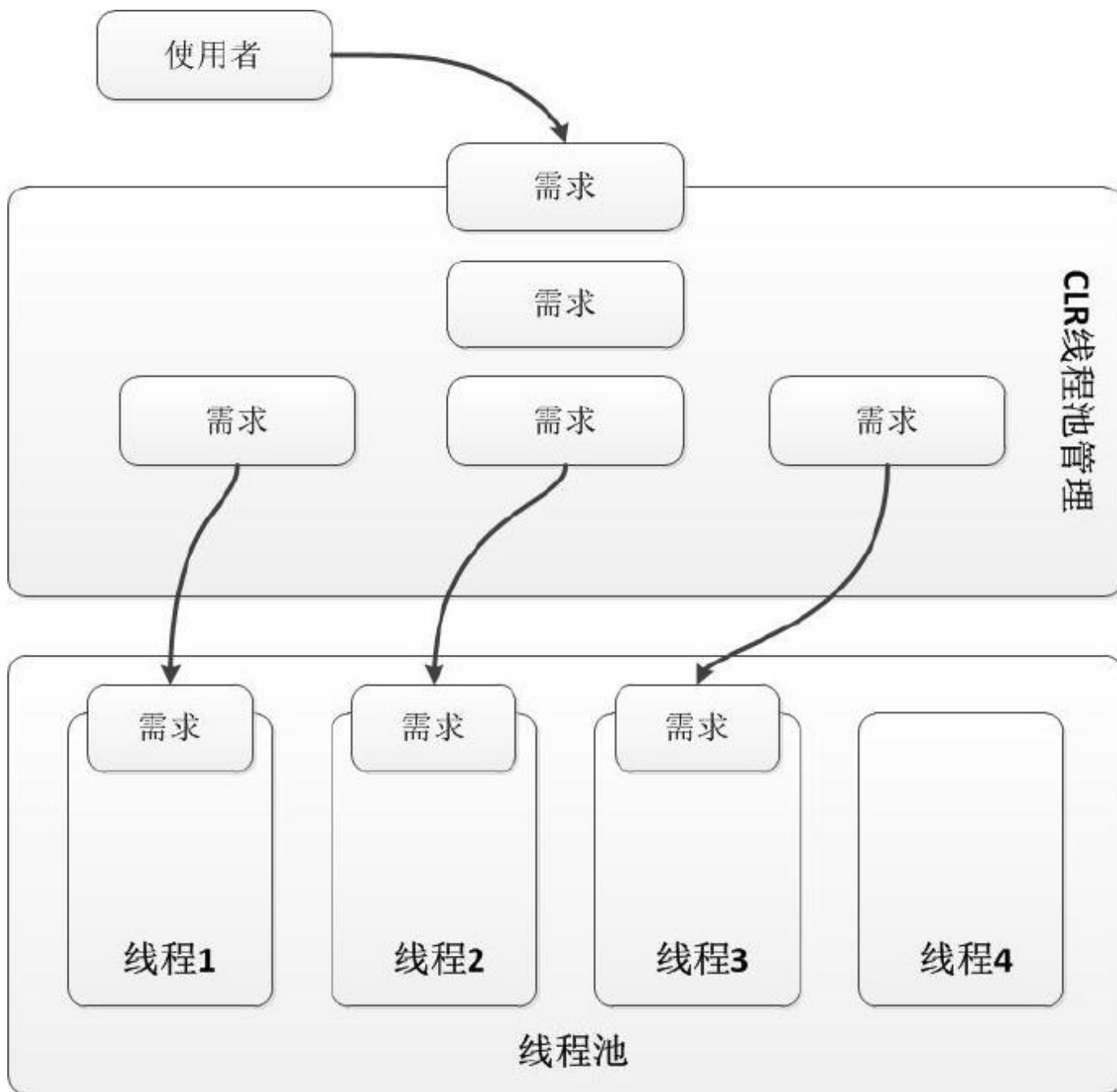
2.2 如何使用.NET中的线程池？

(1) .NET中的线程池是神马

我们都知道，线程的创建和销毁需要很大的性能开销，在Windows NT内核的操作系统中，每个进程都会包含一个线程池。而在.NET中呢，也有自己的线程池，它是由CLR负责管理的。

线程池相当于一个缓存的概念，在该池中已经存在了一些没有被销毁的线程，而当应用程序需要一个新的线程时，就可以从线程池中直接获取一个已经存在的线程。相对应的，当一个线程被使用完毕后并不会立刻被销毁，而是放入线程池中等待下一次使用。

.NET中的线程池由CLR管理，管理的策略是灵活可变的，因此线程池中的线程数量也是可变的，使用者只需向线程池提交需求即可，下图则直观地展示了CLR是如何处理线程池需求的：



PS：线程池中运行的线程均为后台线程（即线程的 `IsBackground` 属性被设为true），所谓的后台线程是指这些线程的运行不会阻碍应用程序的结束。相反的，应用程序的结束则必须等待所有前台线程结束后才能退出。

（2）在.NET中使用线程池

在.NET中通过 `System.Threading.ThreadPool` 类型来提供关于线程池的操作，`ThreadPool` 类型提供了几个静态方法，来允许使用者插入一个工作线程的需求。常用的有以下三个静态方法：

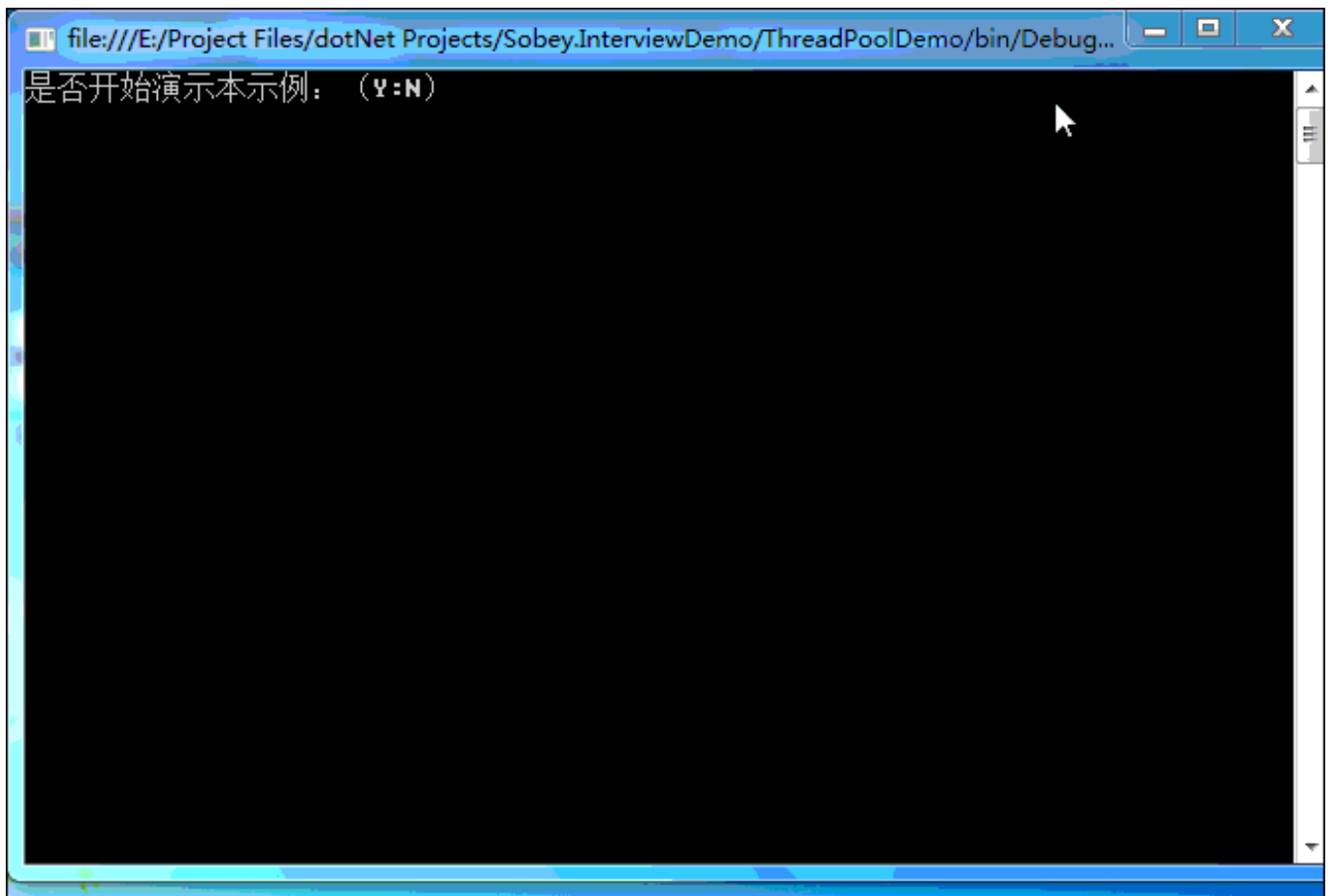
- ① `static bool QueueUserWorkItem(WaitCallback callback)`
- ② `static bool QueueUserWorkItem(WaitCallback callback, Object state)`
- ③ `static bool UnsafeQueueUserWorkItem(WaitCallback callback, Object state)`

有了这几个方法，我们只需要将线程要处理的方法作为参数传入上述方法即可，随后的工作都由CLR的线程池管理程序来完成。其中，`WaitCallback` 是一个委托类型，该委托方法接受一个`Object`类型的参数，并且没有返回值。下面的代码展示了如何使用线程池来编写多线程的程序：

```
C#
class Program
{
    static void Main(string[] args)
    {
        string taskInfo = "运行10秒";
        // 插入一个新的请求到线程池
        bool result = ThreadPool.QueueUserWorkItem(DoWork, taskInfo);
        // 分配线程有可能会失败
        if (!result)
        {
            Console.WriteLine("分配线程失败");
        }
        else
        {
            Console.WriteLine("按回车键结束程序");
        }
        Console.ReadKey();
    }

    private static void DoWork(object state)
    {
        // 模拟做了一些操作，耗时10s
        for (int i = 0; i < 10; i++)
        {
            Console.WriteLine("工作者线程的任务是：{0}", state);
            Thread.Sleep(1000);
        }
    }
}
```

上述代码执行后，如果不输入任何字符，那么会得到如下图所示的执行结果：



PS：事实上，UnsafeQueueWorkItem方法实现了完全相同的功能，二者的差别在于 UnsafeQueueWorkItem方法不会将调用线程的堆栈传递给辅助线程，这就意味着主线程的权限限制不会传递给辅助线程。UnsafeQueueWorkItem由于不进行这样的传递，因此会得到更高的运行效率，但是潜在地提升了辅助线程的权限，也就有可能会成为一个潜在的安全漏洞。

2.3 如何查看和设置线程池的上下限？

线程池的线程数是有限制的，通常情况下，我们无需修改默认的配置。但在一些场合，我们可能需要了解线程池的上下限和剩余的线程数。线程池作为一个缓冲池，有着其上下限。在通常情况下，当线程池中的线程数小于线程池设置的下限时，线程池会设法创建新的线程，而当线程池中的线程数大于线程池设置的上限时，线程池将销毁多余的线程。

PS：在 .NET Framework 4.0 中，每个 CPU 默认的工作者线程数量最大值为 250 个，最小值为 2 个。而 IO 线程的默认最大值为 1000 个，最小值为 2 个。

在 .NET 中，通过 ThreadPool 类型提供的 5 个静态方法可以获取和设置线程池的上限和下限，同时它还额外地提供了一个方法来让程序员获知当前可用的线程数量，下面是这五个方法的签名：

- ① `static void GetMaxThreads(out int workerThreads, out int completionPortThreads)`
- ② `static void GetMinThreads(out int workerThreads, out int completionPortThreads)`
- ③ `static bool SetMaxThreads(int workerThreads, int completionPortThreads)`
- ④ `static bool SetMinThreads(int workerThreads, int completionPortThreads)`

⑤ `static void GetAvailableThreads(out int workerThreads, out int completionPortThreads)`

下面的代码示例演示了如何查询线程池的上下限阈值和可用线程数量：

C#

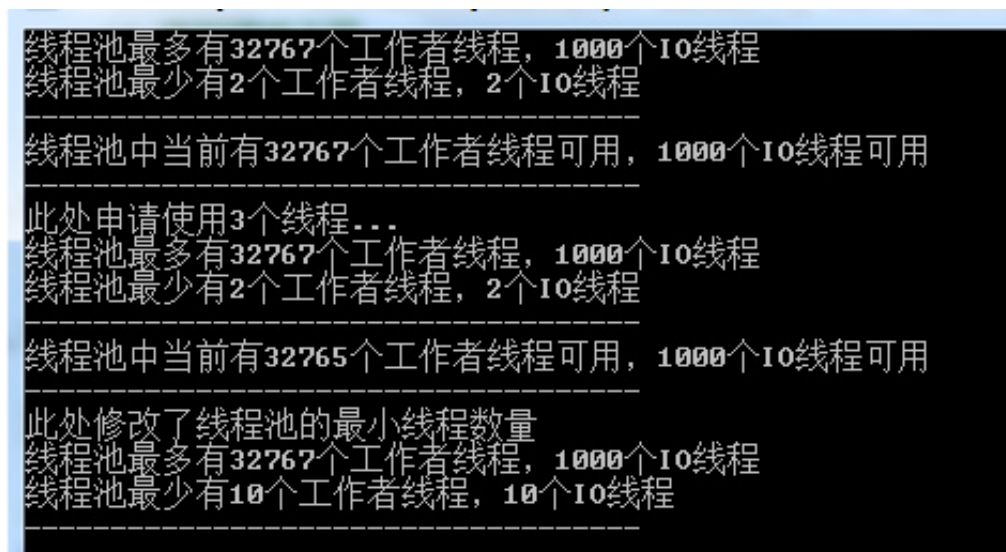
```
class Program
{
    static void Main(string[] args)
    {
        // 打印阈值和可用数量
        GetLimitation();
        GetAvailable();
        // 使用掉其中三个线程
        Console.WriteLine("此处申请使用3个线程...");
        ThreadPool.QueueUserWorkItem(Work);
        ThreadPool.QueueUserWorkItem(Work);
        ThreadPool.QueueUserWorkItem(Work);
        Thread.Sleep(1000);
        // 打印阈值和可用数量
        GetLimitation();
        GetAvailable();
        // 设置最小值
        Console.WriteLine("此处修改了线程池的最小线程数量");
        ThreadPool.SetMinThreads(10, 10);
        // 打印阈值
        GetLimitation();
        Console.ReadKey();
    }
    // 运行10s的方法
    private static void Work(object o)
    {
        Thread.Sleep(10 * 1000);
    }
    // 打印线程池的上下限阈值
    private static void GetLimitation()
    {
        int maxWork, minWork, maxIO, minIO;
        // 得到阈值上限
        ThreadPool.GetMaxThreads(out maxWork, out maxIO);
        // 得到阈值下限
        ThreadPool.GetMinThreads(out minWork, out minIO);
        // 打印阈值上限
        Console.WriteLine("线程池最多有 {0} 个工作者线程, {1} 个IO线程",
```

```

maxWork.ToString(), maxIO.ToString());
        // 打印阈值下限
        Console.WriteLine("线程池最少有 {0} 个工作者线程, {1} 个IO线程",
minWork.ToString(), minIO.ToString());
        Console.WriteLine("-----");
    }
    // 打印可用线程数量
    private static void GetAvailable()
    {
        int remainWork, remainIO;
        // 得到当前可用线程数量
        ThreadPool.GetAvailableThreads(out remainWork, out remainIO);
        // 打印可用线程数量
        Console.WriteLine("线程池中当前有 {0} 个工作者线程可用, {1} 个IO线程可
用", remainWork.ToString(), remainIO.ToString());
        Console.WriteLine("-----");
    }
}

```

该实例的执行结果如下图所示：



```

线程池最多有32767个工作者线程, 1000个IO线程
线程池最少有2个工作者线程, 2个IO线程
-----
线程池中当前有32767个工作者线程可用, 1000个IO线程可用
-----
此处申请使用3个线程...
线程池最多有32767个工作者线程, 1000个IO线程
线程池最少有2个工作者线程, 2个IO线程
-----
线程池中当前有32765个工作者线程可用, 1000个IO线程可用
-----
此处修改了线程池的最小线程数量
线程池最多有32767个工作者线程, 1000个IO线程
线程池最少有10个工作者线程, 10个IO线程
-----

```

PS：上面代码示例在不同的计算机上运行可能会得到不同的结果，线程池中的可用数码不会再初始时达到最大值，事实上CLR会尝试以一定的时间间隔来逐一地创建新线程，但这个时间间隔非常短。

2.4 如何定义线程独享的全局数据？

线程和进程最大的一个区别就在于线程间可以共享数据和资源，而进程则充分地隔离。在很多场合，即使同一进程的多个线程之间拥有相同的内存空间，也需要在逻辑上为某些线程分配独享的数据。例如，在实际开发中往往会针对一些ORM如EF一类的上下文实体做线程内唯一实例的设置，这时就需要用到下面提到的技术。

(1) 线程本地存储 (Thread Local Storage, TLS)

很多时候，程序员可能会希望拥有线程内可见的变量，而不希望其他线程对其进行访问和修改（传统方

式中的静态变量是对整个应用程序域可见的)，这就需要用到TLS的概念。所谓的线程本地存储（TLS）是指存储在线程环境块内的一个结构，用来存放该线程内独享的数据。进程内的线程不能访问不属于自己的TLS，这就保证了TLS内的数据在线程内是全局共享的，而对于线程外确实不可见的。

（2）定义和使用TLS变量

在.NET中提供了下列连个方法来存取线程独享的数据，它们都定义在System.Threading.Thread类型中：

① object GetData(LocalDataStoreSlot slot)

② void SetData(LocalDataStoreSlot slot, object data)

下面的代码示例则展示了这个机制的使用方法：

```
C#
class Program
{
    static void Main(string[] args)
    {
        Console.WriteLine("开始测试数据插槽：");
        // 创建五个线程来同时运行，但是这里不适合用线程池，
        // 因为线程池内的线程会被反复使用导致线程ID一致
        for (int i = 0; i < 5; i++)
        {
            Thread thread = new Thread(ThreadDataSlot.Work);
            thread.Start();
        }
        Console.ReadKey();
    }
}

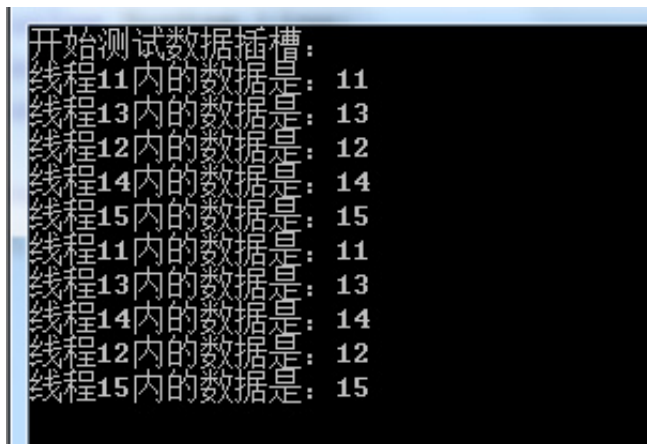
/// <summary>
/// 包含线程方法和数据插槽
/// </summary>
public class ThreadDataSlot
{
    // 分配一个数据插槽，注意插槽本身是全局可见的，因为这里的分配是在所有线程
    // 的TLS内创建数据块
    private static LocalDataStoreSlot localSlot = Thread.AllocateDataSlot();
    // 线程要执行的方法，操作数据插槽来存放数据
    public static void Work()
    {
        // 将线程ID注册到数据插槽中，一个应用程序内线程ID不会重复
        Thread.SetData(localSlot, Thread.CurrentThread.ManagedThreadId);
        // 查看一下刚刚插入的数据
    }
}
```

```

        Console.WriteLine("线程{0}内的数据是：
{1}", Thread.CurrentThread.ManagedThreadId.ToString(), Thread.GetData(localSlot).ToString());
        // 这里线程休眠1秒
        Thread.Sleep(1000);
        // 查看其他线程的运行是否干扰了当前线程数据插槽内的数据
        Console.WriteLine("线程{0}内的数据是：{1}",
Thread.CurrentThread.ManagedThreadId.ToString(), Thread.GetData(localSlot).ToString());
    }
}

```

该实例的执行结果如下图所示，从下图可以看出多线程的并行运行并没有破坏每个线程插槽内的数据，这就是TLS所提供的功能。



PS：LocalDataStoreSlot对象本身并不是线程共享的，初始化一个LocalDataStoreSlot对象意味着在应用程序域内的每个线程上都分配了一个数据插槽。

(3) ThreadStaticAttribute特性的使用

除了使用上面说到的数据槽之外，我们还有另一种方式，即ThreadStaticAttribute特性。申明了该特性的变量，会被.NET作为线程独享的数据来使用。我们可以将其理解为一种被.NET封装了的TLS机制，本质上，它仍然使用了线程环境块来存放数据。

下面的示例代码展示了ThreadStaticAttribute特性的使用：

```

C#
class Program
{
    static void Main(string[] args)
    {
        Console.WriteLine("开始测试数据插槽：");
        // 创建五个线程来同时运行，但是这里不适合用线程池，
        // 因为线程池内的线程会被反复使用导致线程ID一致
        for (int i = 0; i < 5; i++)
        {
            Thread thread = new Thread(ThreadStatic.Work);

```

```
        thread.Start();
    }
    Console.ReadKey();
}
}

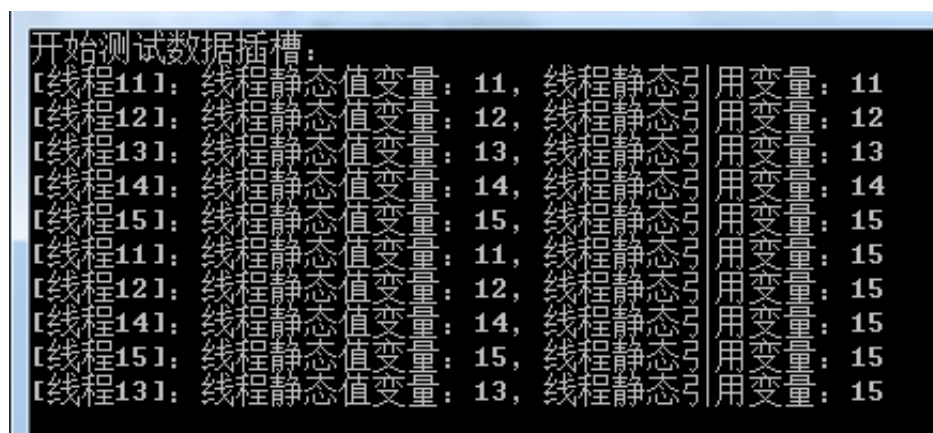
/// <summary>
/// 包含线程静态数据
/// </summary>
public class ThreadStatic
{
    // 值类型的线程静态数据
    [ThreadStatic]
    private static int threadId = 0;
    // 引用类型的线程静态数据
    private static Ref refThreadId = new Ref();
    /// <summary>
    /// 线程执行的方法，操作线程静态数据
    /// </summary>
    public static void Work()
    {
        // 存储线程ID，一个应用程序域内线程ID不会重复
        threadId = Thread.CurrentThread.ManagedThreadId;
        refThreadId.Id = Thread.CurrentThread.ManagedThreadId;
        // 查看一下刚刚插入的数据
        Console.WriteLine("[线程{0}]: 线程静态值变量: {1}, 线程静态引用变
量: {2}", Thread.CurrentThread.ManagedThreadId.ToString(), threadId,
refThreadId.Id.ToString());
        // 睡眠1s
        Thread.Sleep(1000);
        // 查看其他线程的运行是否干扰了当前线程静态数据
        Console.WriteLine("[线程{0}]: 线程静态值变量: {1}, 线程静态引用变
量: {2}", Thread.CurrentThread.ManagedThreadId.ToString(), threadId,
refThreadId.Id.ToString());
    }
}

/// <summary>
/// 简单引用类型
/// </summary>
public class Ref
{
    private int id;
    public int Id
```



```
{  
    get  
    {  
        return id;  
    }  
    set  
    {  
        id = value;  
    }  
}
```

该实例的执行结果如下图所示，正如我们所看到的，对于使用了ThreadStatic特性的字段，.NET会将其作为线程独享的数据来处理，当某个线程对一个使用了ThreadStatic特性的字段进行赋值后，这个值只有这个线程自己可以看到并访问修改，该值对于其他线程时不可见的。相反，没有标记该特性的，则会被多个线程所共享。



2.5 如何使用异步模式读取一个文件？

异步模式是在处理流类型时经常采用的一种方式，其应用的领域相当广阔，包括读写文件、网络传输、读写数据库，甚至可以采用异步模式来做任何计算工作。相对于手动编写线程代码，异步模式是一个高效的编程模式。

(1) 所谓异步模式是个什么鬼？

所谓的异步模式，是指在启动一个操作之后可以继续执行其他工作而不会发生阻塞。以读取文件为例，在同步模式下，当程序执行到Read方法时，需要等到读取动作结束后才能继续往下执行。而异步模式则可以简单地通知开始读取任务之后，继续其他的操作。异步模式的优点就在于不需要使当前线程等待，而可以充分地利用CPU时间。

PS：异步模式区别于线程池机制的地方在于其允许程序查看操作的执行状态，而如果利用线程池的后台线程，则无法确切地知道操作的进行状态以及其是否已经结束。

使用异步模式可以通过一些异步聚集技巧来查看异步操作的结果，所谓的聚集技巧是指查看操作是否结束的方法，常用的方式是：在调用BeingXXX方法时传入操作结束后需要执行的方法（又称为回调方法），同时把执行异步操作的对象传入以便执行EndXXX方法。

(2) 使用异步模式读取一个文件

下面的示例代码中:

① 主线程中负责开始异步读取并传入聚集时需要使用的方法和状态对象:

```
C#
partial class Program
{
    // 测试文件
    private const string testFile = @"C:\AsyncReadTest.txt";
    private const int bufferSize = 1024;
    static void Main(string[] args)
    {
        // 删除已存在文件
        if (File.Exists(testFile))
        {
            File.Delete(testFile);
        }
        // 写入一些东西以便后面读取
        using (FileStream stream = File.Create(testFile))
        {
            string content = "我是文件具体内容, 我是不是帅得掉渣? ";
            byte[] contentByte = Encoding.UTF8.GetBytes(content);
            stream.Write(contentByte, 0, contentByte.Length);
        }
        // 开始异步读取文件具体内容
        using (FileStream stream = new FileStream(testFile, FileMode.Open,
            FileAccess.Read, FileShare.Read, bufferSize, FileOptions.Asynchronous))
        {
            byte[] data = new byte[bufferSize];
            // 将自定义类型对象实例作为参数
            ReadFileClass rfc = new ReadFileClass(stream, data);
            // 开始异步读取
            IAsyncResult result = stream.BeginRead(data, 0,
data.Length, FinshCallBack, rfc);
            // 模拟做了一些其他的操作
            Thread.Sleep(3 * 1000);
            Console.WriteLine("主线程执行完毕, 按回车键退出程序");
        }
        Console.ReadKey();
    }
}
```

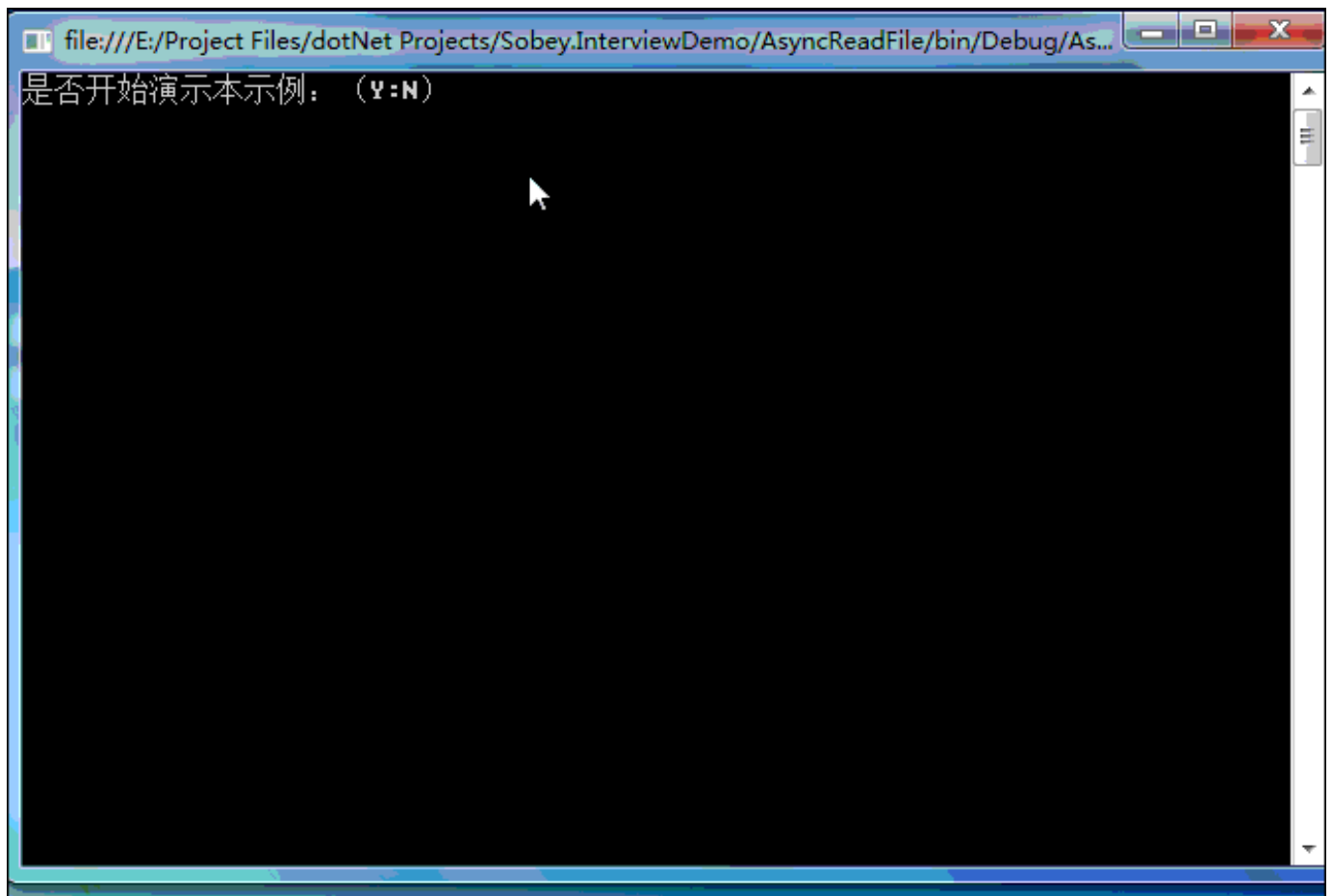
② 定义了完成异步操作读取之后需要调用的方法，其逻辑是简单地打印出文件的内容：

```
C#
partial class Program
{
    /// <summary>
    /// 完成异步操作后的回调方法
    /// </summary>
    /// <param name="result">状态对象</param>
    private static void FinshCallBack(IAsyncResult result)
    {
        ReadFileClass rfc = result.AsyncState as ReadFileClass;
        if (rfc != null)
        {
            // 必须的步骤：让异步读取占用的资源被释放掉
            int length = rfc.stream.EndRead(result);
            // 获取读取到的文件内容
            byte[] fileData = new byte[length];
            Array.Copy(rfc.data, 0, fileData, 0, fileData.Length);
            string content = Encoding.UTF8.GetString(fileData);
            // 打印读取到的文件基本信息
            Console.WriteLine("读取文件结束：文件长度为[{0}]，文件内容
为[{1}]", length.ToString(), content);
        }
    }
}
```

③ 定义了作为状态对象传递的类型，这个类型对所有需要传递的数据包进行打包：

```
C#/// <summary>
/// 传递给异步操作的回调方法
/// </summary>
public class ReadFileClass
{
    // 以便回调方法中释放异步读取的文件流
    public FileStream stream;
    // 文件内容
    public byte[] data;
    public ReadFileClass(FileStream stream, byte[] data)
    {
        this.stream = stream;
        this.data = data;
    }
}
```

下图展示了该实例的执行结果：



如上面的实例，使用回调方法的异步模式需要花费一点额外的代码量，因为它需要将异步操作的对象及操作的结果数据都打包到一个类型里以便能够传递回给回调的委托方法，这样在委托方法中才能够有机会处理操作的结果，并且调用EndXXX方法以释放资源。

2.6 如何阻止线程执行上下文的传递？

（1）何为线程的执行上下文

在.NET中，每一个线程都会包含一个执行上下文，执行上下文是指线程运行中某时刻的上下文概念，类似于一个动态过程的快照（Snapshot）。在.NET中，System.Threading中的ExecutionContext类型代表了一个执行上下文，该执行上下文会包含：安全上下文、调用上下文、本地化上下文、事务上下文和CLR宿主上下文等等。通常情况下，我们将所有这些综合成为线程的上下文。

（2）执行上下文的流动

当程序中新建一个线程时，执行上下文会自动地从当前线程流入到新建的线程之中，这样做可以保证新建的线程天生就就有和主线程相同的安全设置和文化等设置。下面的示例代码通过修改安全上下文来展示线程上下文的流动性，主要使用到ExecutionContext类的Capture方法来捕获当前想成的执行上下文。

① 首先定义一些辅助犯法，封装了文件的创建、删除和文件访问权限检查：

```
C#  
partial class Program  
{
```

```
private static void CreateTestFile()
{
    if (!File.Exists(testFile))
    {
        FileStream stream = File.Create(testFile);
        stream.Dispose();
    }
}

private static void DeleteTestFile()
{
    if (File.Exists(testFile))
    {
        File.Delete(testFile);
    }
}

// 尝试访问测试文件来测试安全上下文
private static void JudgePermission(object state)
{
    try
    {
        // 尝试访问文件
        File.GetCreationTime(testFile);
        // 如果没有异常则测试通过
        Console.WriteLine("权限测试通过");
    }
    catch (SecurityException)
    {
        // 如果出现异常则测试通过
        Console.WriteLine("权限测试没有通过");
    }
    finally
    {
        Console.WriteLine("-----");
    }
}
```

② 其次在入口方法中使主线程和创建的子线程访问指定文件来查看权限上下文流动到子线程中的情况：（这里需要注意的是由于在 .NET 4.0 及以上版本中 `FileIOPermission` 的 `Deny` 方法已过时，为了方便测试，将程序的 .NET 版本调整为了 3.5）

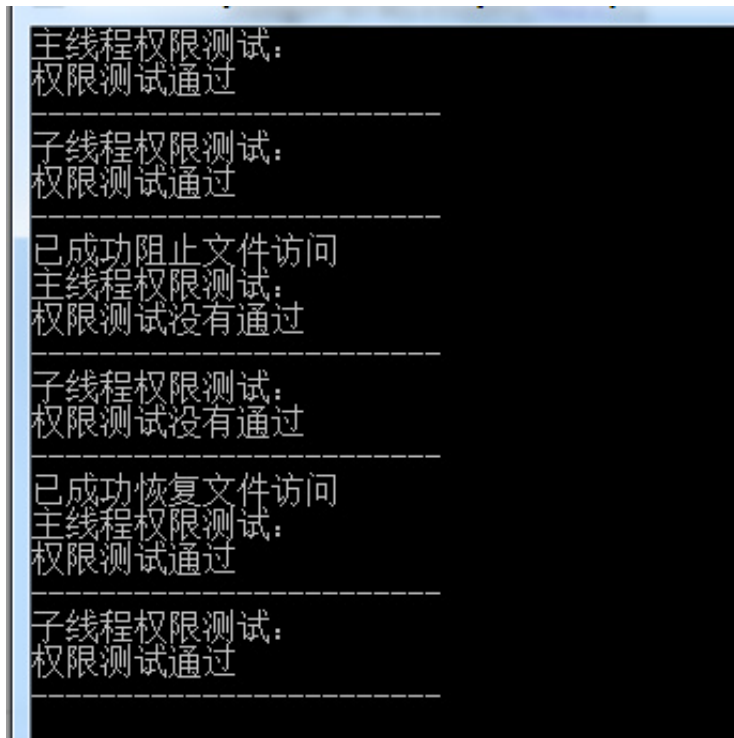
C#

```
partial class Program
```

```
{  
  
    private const string testFile = @"C:\TestContext.txt";  
    static void Main(string[] args)  
    {  
  
        try  
        {  
  
            CreateTestFile();  
            // 测试当前线程的安全上下文  
            Console.WriteLine("主线程权限测试: ");  
            JudgePermission(null);  
            // 创建一个子线程 subThread1  
            Console.WriteLine("子线程权限测试: ");  
            Thread subThread1 = new Thread(JudgePermission);  
            subThread1.Start();  
            subThread1.Join();  
            // 现在修改安全上下文, 阻止文件访问  
            FileIOPermission fip = new  
FileIOPermission(FileIOPermissionAccess.AllAccess, testFile);  
            fip.Deny();  
            Console.WriteLine("已成功阻止文件访问");  
            // 测试当前线程的安全上下文  
            Console.WriteLine("主线程权限测试: ");  
            JudgePermission(null);  
            // 创建一个子线程 subThread2  
            Console.WriteLine("子线程权限测试: ");  
            Thread subThread2 = new Thread(JudgePermission);  
            subThread2.Start();  
            subThread2.Join();  
            // 现在修改安全上下文, 允许文件访问  
            SecurityPermission.RevertDeny();  
            Console.WriteLine("已成功恢复文件访问");  
            // 测试当前线程安全上下文  
            Console.WriteLine("主线程权限测试: ");  
            JudgePermission(null);  
            // 创建一个子线程 subThread3  
            Console.WriteLine("子线程权限测试: ");  
            Thread subThread3 = new Thread(JudgePermission);  
            subThread3.Start();  
            subThread3.Join();  
            Console.ReadKey();  
        }  
  
        finally
```

```
        {  
            DeleteTestFile();  
        }  
    }  
}
```

该实例的执行结果如下图所示，从图中可以看出程序中通过FileIOPermission对象来控制对主线程对文件的访问权限，并且通过新建子线程来查看主线程的安全上下文的改变是否会影响子线程。



正如刚刚说到，主线程的安全上下文将作为执行上下文的一部分由主线程传递给子线程。

(3) 阻止上下文的流动

有的时候，系统需要子线程拥有新的上下文。抛开功能上的需求，执行上下文的流动确实使得程序的执行效率下降很多，线程上下文的包装是一个成本较高的工作，而有的时候这样的包装并不是必须的。在这种情况下，我们如果需要手动地防止线程上下文的流动，常用的有下列两种方法：

- ① System.Threading.ThreadPool类中的UnsafeQueueUserWorkItem方法
- ② ExecutionContext类中的SuppressFlow方法

下面的代码示例展示了如何使用上面两种方法阻止执行上下文的流动：

```
C#  
partial class Program  
{  
    private const string testFile = @"C:\TestContext.txt";  
    static void Main(string[] args)  
    {  
        try
```

```
{  
    CreateTestFile();  
    // 现在修改安全上下文, 阻止文件访问  
    FileIOPermission fip = new  
FileIOPermission(FileIOPermissionAccess.AllAccess, testFile);  
    fip.Deny();  
    Console.WriteLine("已成功阻止文件访问");  
    // 主线程权限测试  
    Console.WriteLine("主线程权限测试: ");  
    JudgePermission(null);  
    // 使用UnsafeQueueUserWorkItem方法创建一个子线程  
    Console.WriteLine("子线程权限测试: ");  
    ThreadPool.UnsafeQueueUserWorkItem(JudgePermission, null);  
    Thread.Sleep(1000);  
    // 使用SuppressFlow方法  
    using (var afc = ExecutionContext.SuppressFlow())  
    {  
        // 测试当前线程安全上下文  
        Console.WriteLine("主线程权限测试: ");  
        JudgePermission(null);  
        // 创建一个子线程 subThread1  
        Console.WriteLine("子线程权限测试: ");  
        Thread subThread1 = new Thread(JudgePermission);  
        subThread1.Start();  
        subThread1.Join();  
    }  
    // 现在修改安全上下文, 允许文件访问  
    SecurityPermission.RevertDeny();  
    Console.WriteLine("已成功恢复文件访问");  
    // 测试当前线程安全上下文  
    Console.WriteLine("主线程权限测试: ");  
    JudgePermission(null);  
    // 创建一个子线程 subThread2  
    Console.WriteLine("子线程权限测试: ");  
    Thread subThread2 = new Thread(JudgePermission);  
    subThread2.Start();  
    subThread2.Join();  
    Console.ReadKey();  
}  
finally  
{  
    DeleteTestFile();  
}
```

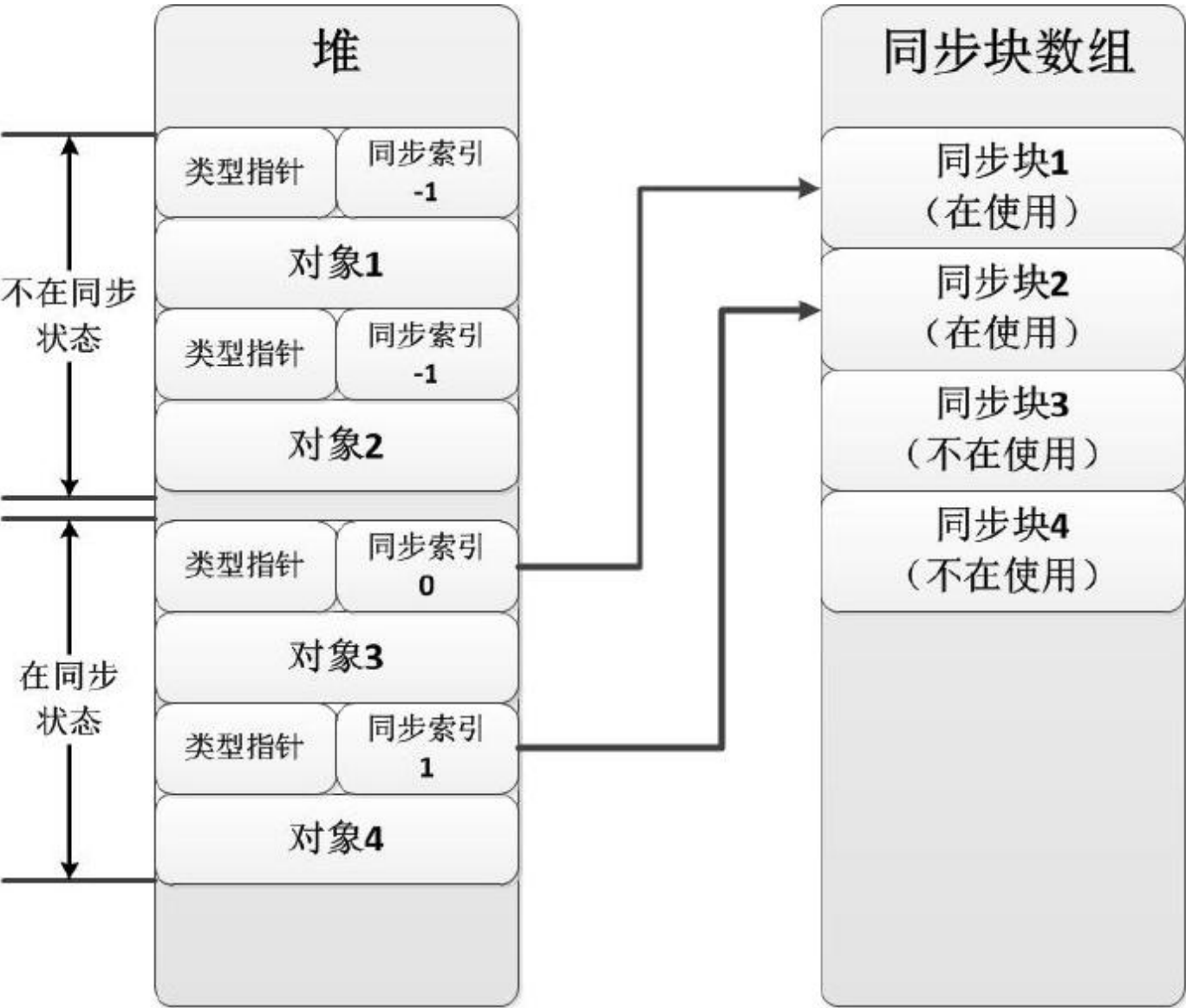


```
    }  
}  
}
```

该实例的执行结果如下图所示，可以看出，通过前面的两种方式有效地阻止了主线程的执行上下文流动到新建的线程之中，这样的机制对于性能的提高有一定的帮助。

三、多线程编程中的线程同步3.1 理解同步块和同步块索引

同步块是.NET中解决对象同步问题的基本机制，该机制为每个堆内的对象（即引用类型对象实例）分配一个同步索引，该索引中只保存一个表明数组内索引的整数。具体过程是：.NET在加载时就会新建一个同步块数组，当某个对象需要被同步时，.NET会为其分配一个同步块，并且把该同步块在同步块数组中的索引加入该对象的同步块索引中。下图展现了这一机制的实现：



同步块机制包含以下几点：

- ① 在.NET被加载时初始化同步块数组；
- ② 每一个被分配在堆上的对象都会包含两个额外的字段，其中一个存储类型指针，而另外一个就是同步块索引，初始时被赋值为-1；
- ③ 当一个线程试图使用该对象进入同步时，会检查该对象的同步索引：

如果同步索引为负数，则会在同步块数组中新建一个同步块，并且将该同步块的索引值写入该对象的同步索引中；

如果同步索引不为负数，则找到该对象的同步块并检查是否有其他线程在使用该同步块，如果有则进入等待状态，如果没有则申明使用该同步块；

④ 当一个对象退出同步时，该对象的同步索引被修改为-1，并且相应的同步块数组中的同步块被视为不再使用。

3.2 C#中的lock关键字有啥作用？

lock关键字可能是我们在遇到线程同步的需求时最常用的方式，但lock只是一个语法糖，为什么这么说呢，下面慢慢道来。

(1) lock的等效代码其实是Monitor类的Enter和Exit两个方法

C#

```
private object locker = new object();
public void Work()
{
    // 避免直接使用私有成员locker（直接使用有可能导致线程不安全）
    object temp = locker;
    Monitor.Enter(temp);
    try
    {
        // 做一些需要线程同步的工作
    }
    finally
    {
        Monitor.Exit(temp);
    }
}
```

(2) System.Threading.Monitor类型的作用和使用

Monitor类型的Enter和Exit方法用来实现进入和退出对象的同步，当Enter方法被调用时，对象的同步索引将被检查，并且.NET将负责一系列的后续工作来保证对象访问时的线程同步，而Exit方法的调用则保证了当前线程释放该对象的同步块。

下面的代码示例演示了如何使用lock关键字来实现线程同步：

C#

```
class Program
{
    static void Main(string[] args)
    {
        // 多线程测试静态方法的同步
    }
}
```

```

Console.WriteLine("开始测试静态方法的同步：");
for (int i = 0; i < 5; i++)
{
    Thread thread = new Thread(Lock.StaticIncrement);
    thread.Start();
}
// 这里等待线程执行结束
Thread.Sleep(5 * 1000);
Console.WriteLine("-----");
// 多线程测试实例方法的同步
Console.WriteLine("开始测试实例方法的同步：");
Lock l = new Lock();
for (int i = 0; i < 6; i++)
{
    Thread thread = new Thread(l.InstanceIncrement);
    thread.Start();
}
Console.ReadKey();
}
}

public class Lock
{
    // 静态方法同步锁
    private static object staticLocker = new object();
    // 实例方法同步锁
    private object instanceLocker = new object();
    // 成员变量
    private static int staticNumber = 0;
    private int instanceNumber = 0;
    // 测试静态方法的同步
    public static void StaticIncrement(object state)
    {
        lock (staticLocker)
        {
            Console.WriteLine("当前线程ID: {0}",
Thread.CurrentThread.ManagedThreadId.ToString());
            Console.WriteLine("staticNumber的值为: {0}",
staticNumber.ToString());

            // 这里可以制造线程并行执行的机会，来检查同步的功能
            Thread.Sleep(200);
            staticNumber++;
            Console.WriteLine("staticNumber自增后为: {0}",

```

```
staticNumber.ToString());
    }
}

// 测试实例方法的同步
public void InstanceIncrement(object state)
{
    lock (instanceLocker)
    {
        Console.WriteLine("当前线程ID:
{0}", Thread.CurrentThread.ManagedThreadId.ToString());
        Console.WriteLine("instanceNumber的值为: {0}",
instanceNumber.ToString());

        // 这里可以制造线程并行执行的机会，来检查同步的功能
        Thread.Sleep(200);
        instanceNumber++;
        Console.WriteLine("instanceNumber自增后为: {0}",
instanceNumber.ToString());
    }
}
}
```

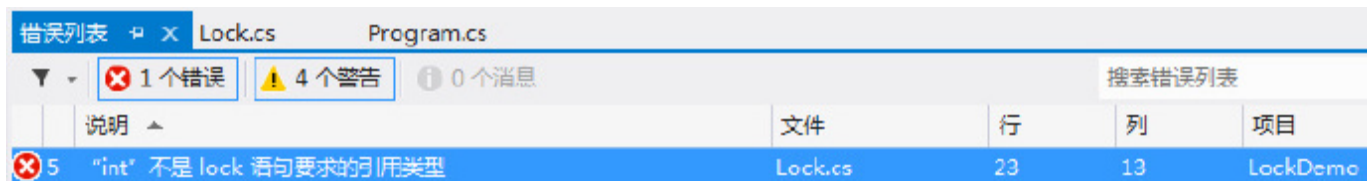
下图是该实例的执行结果：

PS：线程同步本身违反了多线程并行运行的原则，所以我们在使用线程同步时应该尽量做到将lock加在最小的程序块上。对于静态方法的同步，一般采用静态私有的引用对象成员，而对于实例方法的同步，一般采用私有的引用对象成员。

3.3 可否使用值类型对象来实现线程同步吗？

前面已经说到，在.NET中每个堆内的对象都会有一个同步索引字段，用以指向同步块的位置。但是，对于值类型来说，它们的对象是分配在堆栈上的，也就是说值类型是没有同步索引这一字段的，所以直接使用值类型对象无法实现线程同步。

如果在程序中对于lock关键字使用了值类型对象，会直接导致一个编译错误：



3.4 可否使用引用类型对象自身进行同步？

引用类型的对象是分配在堆上的，必然会包含同步索引，也可以分配同步块，所以原则上可以在对象的方法内对自身进行同步。而事实上，这样的代码也确实能有效地保证线程同步。But，这样的代码健壮性存在一定问题。

(1) lock(this)

回顾lock(this)的设计，就可以看出问题来：this代表了执行代码的当前对象，可以预见该对象可以被任何使用者访问，这就导致了不仅对象内部的代码在争用同步块，连类型的使用者也可以有意无意地进入到争用的队伍中→这显然不符合设计意图。

下面通过一个代码示例展示了一个恶意的使用者是如何导致类型死锁的：

C#

```
class Program
{
    static void Main(string[] args)
    {
        Console.WriteLine("开始使用");
        SynchroThis st = new SynchroThis();
        // 模拟恶意的使用者
        Monitor.Enter(st);
        // 正常的使用者会收到恶意使用者的影响
        // 下面的代码完全正确，但却被死锁
        Thread thread = new Thread(st.Work);
        thread.Start();
        thread.Join();
        // 程序不会执行到这里
        Console.WriteLine("使用结束");
        Console.ReadKey();
    }
}

public class SynchroThis
{
    private int number = 0;
    public void Work(object state)
    {
        lock (this)
        {
            Console.WriteLine("number现在的值为: {0}",
number.ToString());

            number++;
            // 模拟做了其他工作
            Thread.Sleep(200);
            Console.WriteLine("number自增后值为: {0}",
number.ToString());
        }
    }
}
```

运行这个示例，我们发现程序完全被死锁，这是因为一个恶意的使用者在使用了同步块之后却没有对其

进行释放，导致了SynchroThis类型的方法被组织。

(2) lock(typeof(类型名))

这样的设计有时候会被用来在静态方法中实现线程同步，因为静态方法的访问需要通过类型来进行，但它也和lock(this)一样，缺乏健壮性。下面展示了常见的错误使用代码示例：

C#

```
class Program
{
    static void Main(string[] args)
    {
        Console.WriteLine("开始使用");
        SynchroThis st = new SynchroThis();
        // 模拟恶意的使用者
        Monitor.Enter(typeof(SynchroThis));
        // 正常的使用者会收到恶意使用者的影响
        // 下面的代码完全正确，但却被死锁
        Thread thread = new Thread(SynchroThis.Work);
        thread.Start();
        thread.Join();
        // 程序不会执行到这里
        Console.WriteLine("使用结束");
        Console.ReadKey();
    }
}

public class SynchroThis
{
    private static int number = 0;
    public static void Work(object state)
    {
        lock (typeof(SynchroThis))
        {
            Console.WriteLine("number现在的值为: {0}",
number.ToString());

            number++;
            // 模拟做了其他工作
            Thread.Sleep(200);
            Console.WriteLine("number自增后值为: {0}",
number.ToString());
        }
    }
}
```

可以发现，当一个恶意的使用者对type对象进行同步时，也会造成所有的使用者被死锁。

PS：应该完全避免使用this对象和当前类型对象作为同步对象，而应该在类型中定义私有的同步对象，同时应该使用lock而不是Monitor类型，这样可以有效地减少同步块不被释放的情况。

3.5 互斥体是个什么鬼？Mutex和Monitor两个类型的功能有啥区别？

(1) 什么是互斥体？

在操作系统中，互斥体（Mutex）是指某些代码片段在任意时间内只允许一个线程进入。例如，正在进行一盘棋，任意时刻只允许一个棋手往棋盘上落子，这和线程同步的概念基本一致。

(2) .NET中的互斥体

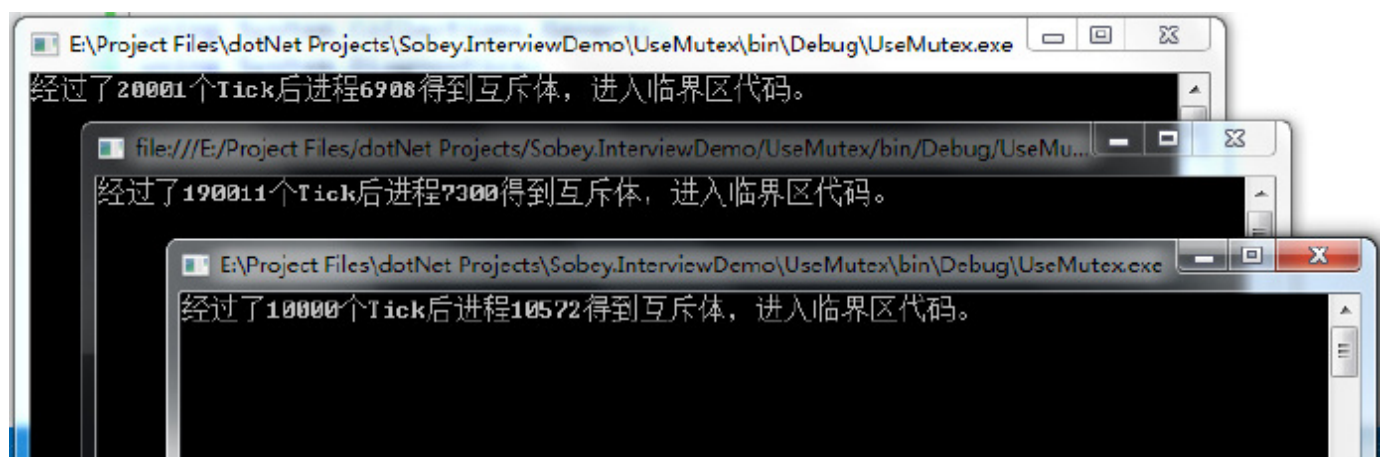
Mutex类是.NET中为我们封装的一个互斥体类型，和Mutex类似的还有Semaphore（信号量）等类型。下面的示例代码展示了Mutex类型的使用：

```
C#
class Program
{
    const string testFile = "C:\\\\TestMutex.txt";
    /// <summary>
    /// 这个互斥体保证所有的进程都能得到同步
    /// </summary>
    static Mutex mutex = new Mutex(false, "TestMutex");
    static void Main(string[] args)
    {
        //留出时间来启动其他进程
        Thread.Sleep(3000);
        DoWork();
        mutex.Close();
        Console.ReadKey();
    }
    /// <summary>
    /// 往文件里写连续的内容
    /// </summary>
    static void DoWork()
    {
        long d1 = DateTime.Now.Ticks;
        mutex.WaitOne();
        long d2 = DateTime.Now.Ticks;
        Console.WriteLine("经过了{0}个Tick后进程{1}得到互斥体，进入临界区代码。", (d2 - d1).ToString(), Process.GetCurrentProcess().Id.ToString());
        try
        {

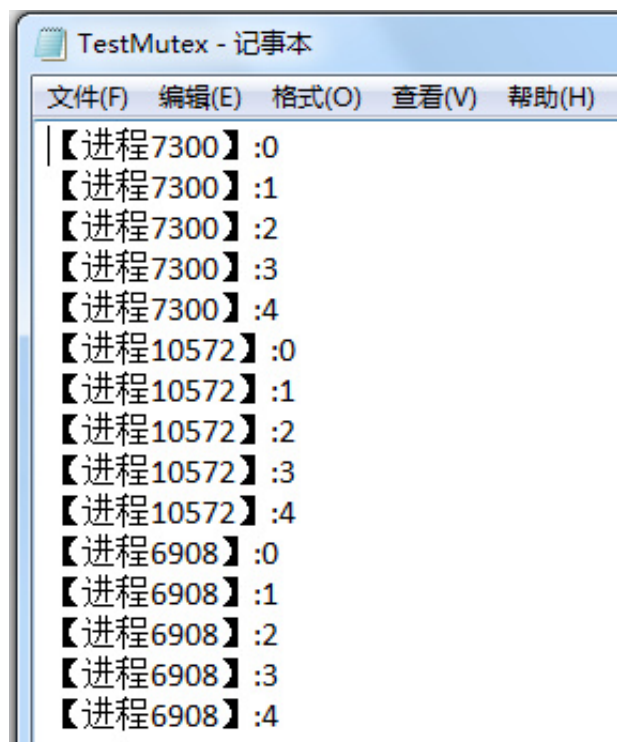
```

```
if (!File.Exists(testFile))
{
    FileStream fs = File.Create(testFile);
    fs.Dispose();
}
for (int i = 0; i < 5; i++)
{
    // 每次都保证文件被关闭再重新打开
    // 确定有mutex来同步，而不是IO机制
    using (FileStream fs = File.Open(testFile,
        FileMode.Append))
    {
        string content = "【进程” +
        Process.GetCurrentProcess().Id.ToString() +
        ”】:” + i.ToString() + ”\r\n”;
        Byte[] data =
        Encoding.Default.GetBytes(content);
        fs.Write(data, 0, data.Length);
    }
    // 模拟做了其他工作
    Thread.Sleep(300);
}
}
finally
{
    mutex.ReleaseMutex();
}
}
```

模拟多个用户，执行上述代码，下图就是在我的计算机上的执行结果：



现在打开C盘目录下的TestMutext.txt文件，将看到如下图所示的结果：



```
TestMutex - 记事本
文件(F) 编辑(E) 格式(O) 查看(V) 帮助(H)

【进程7300】:0
【进程7300】:1
【进程7300】:2
【进程7300】:3
【进程7300】:4
【进程10572】:0
【进程10572】:1
【进程10572】:2
【进程10572】:3
【进程10572】:4
【进程6908】:0
【进程6908】:1
【进程6908】:2
【进程6908】:3
【进程6908】:4
```

(3) Mutex和Monitor的区别

这两者虽然都用来进行同步的功能，但实现方法不同，其最显著的两个差别如下：

- ① Mutex使用的是操作系统的内核对象，而Monitor类型的同步机制则完全在.NET框架之下实现，这就导致了Mutex类型的效率要比Monitor类型要低很多；
- ② Monitor类型只能同步同一应用程序域中的线程，而Mutex类型却可以跨越应用程序域和进程。

3.6 如何使用信号量Semaphore？

这里首先借用阮一峰的《[进程与线程的一个简单解释](#)》中的介绍来说一下Mutex和Semaphore：



一个防止他人进入的简单方法，就是门口加一把锁。先到的人锁上门，后到的人看到上锁，就在门口排队，等锁打开再进去。这就叫”互斥锁”（Mutual exclusion，缩写 Mutex），防止多个线程同时读写某一块内存区域。

还有些房间，可以同时容纳 n 个人，比如厨房。也就是说，如果人数大于 n ，多出来的人只能在外面等着。这好比某些内存区域，只能供给固定数目的线程使用。



这时的解决方法，就是在门口挂 n 把钥匙。进去的人就取一把钥匙，出来时再把钥匙挂回原处。后到的人发现钥匙架空了，就知道必须在门口排队等着了。这种做法叫做”信号量”（Semaphore），用来保证多个线程不会互相冲突。

不难看出，mutex是semaphore的一种特殊情况（ $n=1$ 时）。也就是说，完全可以用后者替代前者。但是，

因为mutex较为简单，且效率高，所以在必须保证资源独占的情况下，还是采用这种设计。

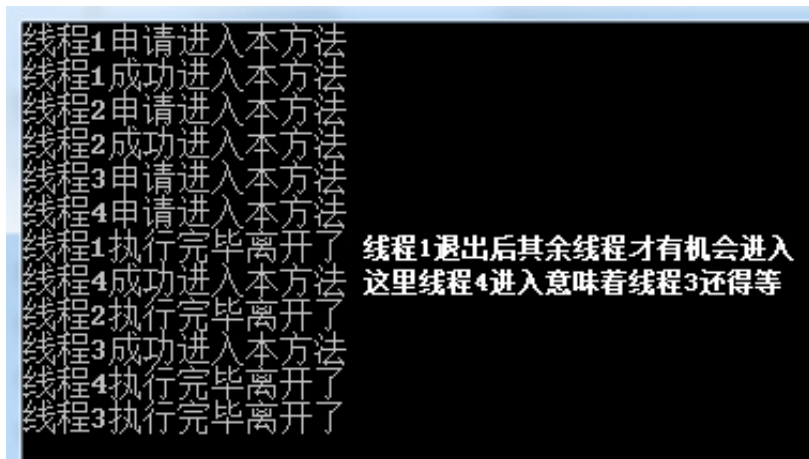
现在我们知道了Semaphore是干啥的了，再把目光放到.NET中的Semaphore上。Semaphore 继承自WaitHandle (Mutex也继承自WaitHandle)，它用于锁机制，与Mutex不同的是，它允许指定数量的线程同时访问资源，在线程超过数量以后，则进行排队等待，直到之前的线程退出。Semaphore很适合应用于Web服务器这样的高并发场景，可以限制对资源访问的线程数。此外，Semaphore不需要一个锁的持有者，通常也将Semaphore声明为静态的。

下面的示例代码演示了4条线程想要同时执行ThreadEntry()方法，但同时只允许2条线程进入：

C#

```
class Program
{
    // 第一个参数指定当前有多少个“空位”（允许多少条线程进入）
    // 第二个参数指定一共有多少个“座位”（最多允许多少个线程同时进入）
    static Semaphore sem = new Semaphore(2, 2);
    const int threadSize = 4;
    static void Main(string[] args)
    {
        for (int i = 0; i < threadSize; i++)
        {
            Thread thread = new Thread(ThreadEntry);
            thread.Start(i + 1);
        }
        Console.ReadKey();
    }
    static void ThreadEntry(object id)
    {
        Console.WriteLine("线程{0}申请进入本方法", id);
        // WaitOne:如果还有“空位”，则占位，如果没有空位，则等待；
        sem.WaitOne();
        Console.WriteLine("线程{0}成功进入本方法", id);
        // 模拟线程执行了一些操作
        Thread.Sleep(100);
        Console.WriteLine("线程{0}执行完毕离开了", id);
        // Release:释放一个“空位”
        sem.Release();
    }
}
```

上面示例的执行结果如下图所示：



如果将资源比作“座位”，Semaphore接收的两个参数中：第一个参数指定当前有多少个“空位”（允许多少条线程进入），第二个参数则指定一共有多少个“座位”（最多允许多少个线程同时进入）。

WaitOne()方法则表示如果还有“空位”，则占位，如果没有空位，则等待；Release()方法则表示释放一个“空位”。

感叹一下：人生中有很多人在你的城堡中进进出出，城中的人想出去，城外的人想冲进来。But，一个人身边的位置只有那么多，你能给的也只有那么多，在这个狭小的圈子里，有些人要进来，就有一些人不得不离开。

参考资料

- (1) 朱毅，《进入IT企业必读的200个.NET面试题》
- (2) 张子阳，《.NET之美：.NET关键技术深入解析》
- (3) 王涛，《你必须知道的.NET》
- (4) 阮一峰，《进程与线程的一个简单解释》

加入伯乐在线专栏作者。扩大知名度，还能得赞赏！详见《[招募专栏作者](#)》

1 赞 2 收藏 [评论](#)



合作联系

Email: bd@jobbole.com

QQ: 2302462408 （加好友请注明来意）

更多频道

- [小组](#) - 好的话题、有启发的回复、值得信赖的圈子
- [头条](#) - 分享和发现有价值的内容与观点
- [相亲](#) - 为IT单身男女服务的征婚传播平台
- [资源](#) - 优秀的工具资源导航

- [翻译](#) - 翻译传播优秀的外文文章
- [文章](#) - 国内外的精选文章
- [设计](#) - UI, 网页, 交互和用户体验
- [iOS](#) - 专注iOS技术分享
- [安卓](#) - 专注Android技术分享
- [前端](#) - JavaScript, HTML5, CSS
- [Java](#) - 专注Java技术分享
- [Python](#) - 专注Python技术分享