C# 不安全代码

C# 多线程

线程 被定义为程序的执行路径。每个线程都定义了一个独特的控制流。如果您的应用程序涉及到复杂的和耗时的操作，那么设置不同的线程执行路径往往是有益的，每个线程执行特定的工作。

线程是**轻量级进程**。一个使用线程的常见实例是现代操作系统中并行编程的实现。使用线程节省了 CPU 周期的浪费，同时提高了应用程序的效率。

到目前为止我们编写的程序是一个单线程作为应用程序的运行实例的单一的过程运行的。但是，这样子应用程序同时只能执行一个任务。为了同时执行多个任务，它可以被划分为更小的线程。

线程生命周期

线程生命周期开始于 System.Threading.Thread 类的对象被创建时，结束于线程被终止或完成执行时。

下面列出了线程生命周期中的各种状态：

- **未启动状态**：当线程实例被创建但 Start 方法未被调用时的状况。
- **就绪状态**：当线程准备好运行并等待 CPU 周期时的状况。
- **不可运行状态**：下面的几种情况下线程是不可运行的：
  - 已经调用 Sleep 方法
  - 已经调用 Wait 方法
  - 通过 I/O 操作阻塞
- **死亡状态**：当线程已完成执行或已中止时的状况。

主线程

在 C# 中，**System.Threading.Thread** 类用于线程的工作。它允许创建并访问多线程应用程序中的单个线程。进程中第一个被执行的线程称为**主线程**。

当 C# 程序开始执行时，主线程自动创建。使用 **Thread** 类创建的线程被主线程的子线程调用。您可以使用 Thread 类的 **CurrentThread** 属性访问线程。

下面的程序演示了主线程的执行：

实例

```
using System;
using System.Threading;

namespace MultithreadingApplication
{
    class MainThreadProgram
    {
        static void Main(string[] args)
        {
```

```
            Thread th = Thread.CurrentThread;
            th.Name = "MainThread";
            Console.WriteLine("This is {0}", th.Name);
            Console.ReadKey();
        }
    }
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
This is MainThread
```

Thread 类常用的属性和方法

下表列出了 **Thread** 类的一些常用的 **属性**：

属性	描述
CurrentContext	获取线程正在其中执行的当前上下文。
CurrentCulture	获取或设置当前线程的区域性。
CurrentPrinciple	获取或设置线程的当前负责人（对基于角色的安全性而言）。
CurrentThread	获取当前正在运行的线程。
CurrentUICulture	获取或设置资源管理器使用的当前区域性以便在运行时查找区域性特定的资源。
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。

下表列出了 **Thread** 类的一些常用的 **方法**：

序号	方法名 & 描述
1	**public void Abort()**

	在调用此方法的线程上引发 ThreadAbortException，以开始终止此线程的过程。调用此方法通常会终止线程。
2	**public static LocalDataStoreSlot AllocateDataSlot()** 在所有的线程上分配未命名的数据槽。为了获得更好的性能，请改用以 ThreadStaticAttribute 属性标记的字段。
3	**public static LocalDataStoreSlot AllocateNamedDataSlot(string name)** 在所有线程上分配已命名的数据槽。为了获得更好的性能，请改用以 ThreadStaticAttribute 属性标记的字段。
4	**public static void BeginCriticalRegion()** 通知主机执行将要进入一个代码区域，在该代码区域内线程中止或未经处理的异常的影响可能会危害应用程序域中的其他任务。
5	**public static void BeginThreadAffinity()** 通知主机托管代码将要执行依赖于当前物理操作系统线程的标识的指令。
6	**public static void EndCriticalRegion()** 通知主机执行将要进入一个代码区域，在该代码区域内线程中止或未经处理的异常仅影响当前任务。
7	**public static void EndThreadAffinity()** 通知主机托管代码已执行完依赖于当前物理操作系统线程的标识的指令。
8	**public static void FreeNamedDataSlot(string name)** 为进程中的所有线程消除名称与槽之间的关联。为了获得更好的性能，请改用以 ThreadStaticAttribute 属性标记的字段。
9	**public static Object GetData(LocalDataStoreSlot slot)** 在当前线程的当前域中从当前线程上指定的槽中检索值。为了获得更好的性能，请改用以 ThreadStaticAttribute 属性标记的字段。
10	**public static AppDomain GetDomain()** 返回当前线程正在其中运行的当前域。
11	**public static AppDomain GetDomainID()** 返回唯一的应用程序域标识符。
12	**public static LocalDataStoreSlot GetNamedDataSlot(string name)** 查找已命名的数据槽。为了获得更好的性能，请改用以 ThreadStaticAttribute 属性标记的字段。
13	**public void Interrupt()** 中断处于 WaitSleepJoin 线程状态的线程。
14	**public void Join()** 在继续执行标准的 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。此方法有不同的

	重载形式。
15	**public static void MemoryBarrier()** 按如下方式同步内存存取：执行当前线程的处理器在对指令重新排序时，不能采用先执行 MemoryBarrier 调用之后的内存存取，再执行 MemoryBarrier 调用之前的内存存取的方式。
16	**public static void ResetAbort()** 取消为当前线程请求的 Abort。
17	**public static void SetData(LocalDataStoreSlot slot, Object data)** 在当前正在运行的线程上为此线程的当前域在指定槽中设置数据。为了获得更好的性能，请改用以 ThreadStaticAttribute 属性标记的字段。
18	**public void Start()** 开始一个线程。
19	**public static void Sleep(int millisecondsTimeout)** 让线程暂停一段时间。
20	**public static void SpinWait(int iterations)** 导致线程等待由 iterations 参数定义的时间量。
21	**public static byte VolatileRead(ref byte address)** **public static double VolatileRead(ref double address)** **public static int VolatileRead(ref int address)** **public static Object VolatileRead(ref Object address)** 读取字段值。无论处理器的数目或处理器缓存的状态如何，该值都是由计算机的任何处理器写入的最新值。此方法有不同的重载形式。这里只给出了一些形式。
22	**public static void VolatileWrite(ref byte address, byte value)** **public static void VolatileWrite(ref double address, double value)** **public static void VolatileWrite(ref int address, int value)** **public static void VolatileWrite(ref Object address, Object value)** 立即向字段写入一个值，以使该值对计算机中的所有处理器都可见。此方法有不同的重载形式。这里只给出了一些形式。
23	**public static bool Yield()** 导致调用线程执行准备好在当前处理器上运行的另一个线程。由操作系统选择要执行的线程。

创建线程

线程是通过扩展 Thread 类创建的。扩展的 Thread 类调用 **Start()** 方法来开始子线程的执行。

下面的程序演示了这个概念：

实例

```
using System;
using System.Threading;

namespace MultithreadingApplication
{
    class ThreadCreationProgram
    {
        public static void CallToChildThread()
        {
            Console.WriteLine("Child thread starts");
        }

        static void Main(string[] args)
        {
            ThreadStart childref = new ThreadStart(CallToChildThread);
            Console.WriteLine("In Main: Creating the Child thread");
            Thread childThread = new Thread(childref);
            childThread.Start();
            Console.ReadKey();
        }
    }
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
In Main: Creating the Child thread
Child thread starts
```

管理线程

Thread 类提供了各种管理线程的方法。

下面的实例演示了 **sleep()** 方法的使用，用于在一个特定的时间暂停线程。

实例

```
using System;
using System.Threading;

namespace MultithreadingApplication
{
    class ThreadCreationProgram
    {
        public static void CallToChildThread()
        {
            Console.WriteLine("Child thread starts");
            // 线程暂停 5000 毫秒
            int sleepfor = 5000;
```

```
            Console.WriteLine("Child Thread Paused for {0} seconds",
                              sleepfor / 1000);
            Thread.Sleep(sleepfor);
            Console.WriteLine("Child thread resumes");
        }

        static void Main(string[] args)
        {
            ThreadStart childref = new ThreadStart(CallToChildThread);
            Console.WriteLine("In Main: Creating the Child thread");
            Thread childThread = new Thread(childref);
            childThread.Start();
            Console.ReadKey();
        }
    }
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
In Main: Creating the Child thread
Child thread starts
Child Thread Paused for 5 seconds
Child thread resumes
```

销毁线程

Abort() 方法用于销毁线程。

通过抛出 **threadabortexception** 在运行时中止线程。这个异常不能被捕获，如果有 *finally* 块，控制会被送至 *finally* 块。

下面的程序说明了这点：

实例

```
using System;
using System.Threading;

namespace MultithreadingApplication
{
    class ThreadCreationProgram
    {
        public static void CallToChildThread()
        {
            try
            {

                Console.WriteLine("Child thread starts");
                // 计数到 10
                for (int counter = 0; counter <= 10; counter++)
                {
                    Thread.Sleep(500);
                    Console.WriteLine(counter);
                }
```

```
                Console.WriteLine("Child Thread Completed");

            }
            catch (ThreadAbortException e)
            {
                Console.WriteLine("Thread Abort Exception");
            }
            finally
            {
                Console.WriteLine("Couldn't catch the Thread Exception");
            }

        }

        static void Main(string[] args)
        {
            ThreadStart childref = new ThreadStart(CallToChildThread);
            Console.WriteLine("In Main: Creating the Child thread");
            Thread childThread = new Thread(childref);
            childThread.Start();
            // 停止主线程一段时间
            Thread.Sleep(2000);
            // 现在中止子线程
            Console.WriteLine("In Main: Aborting the Child thread");
            childThread.Abort();
            Console.ReadKey();
        }
    }
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
In Main: Creating the Child thread
Child thread starts
0
1
2
In Main: Aborting the Child thread
Thread Abort Exception
Couldn't catch the Thread Exception
```

C# 不安全代码