C#多线程编程

一：线程与进程的概念

1 概念

进程是具有一定独立功能的程序关于某个数据集合上的一次运行活动,进程是系统进行资源分配和调度的一个独立单位.

线程是进程的一个实体,是CPU调度和分派的基本单位,它是比进程更小的能独立运行的基本单位.线程自己基本上不拥有系统资源,只拥有一点在运行中必不可少的资源(如程序计数器,一组寄存器和栈),但是它可与同属一个进程的其他的线程共享进程所拥有的全部资源.

2.关系

一个线程可以创建和撤销另一个线程;同一个进程中的多个线程之间可以并发执行.

相对进程而言，线程是一个更加接近于执行体的概念，它可以与同进程中的其他线程共享数据，但拥有自己的栈空间，拥有独立的执行序列。

3.区别

进程和线程的主要差别在于它们是不同的操作系统资源管理方式。进程有独立的地址空间，一个进程崩溃后，在保护模式下不会对其它进程产生影响，而线程只是一个进程中的不同执行路径。线程有自己的堆栈和局部变量，但线程之间没有单独的地址空间，一个线程死掉就等于整个进程死掉，所以多进程的程序要比多线程的程序健壮，但在进程切换时，耗费资源较大，效率要差一些。但对于一些要求同时进行并且又要共享某些变量的并发操作，只能用线程，不能用进程。

1）简而言之,一个程序至少有一个进程,一个进程至少有一个线程.
2）线程的划分尺度小于进程，使得多线程程序的并发性高。
3）另外，进程在执行过程中拥有独立的内存单元，而多个线程共享内存，从而极大地提高了程序的运行效率。
4）线程在执行过程中与进程还是有区别的。每个独立的线程有一个程序运行的入口、顺序执行序列和程序的出口。但是线程不能够独立执行，必须依存在应用程序中，由应用程序提供多个线程执行控制。
5）从逻辑角度来看，多线程的意义在于一个应用程序中，有多个执行部分可以同时执行。但操作系统并没有将多个线程看做多个独立的应用，来实现进程的调度和管理以及资源分配。这就是进程和线程的重要区别。

二：前台线程与后台线程

.Net的公用语言运行时（Common Language Runtime，CLR）能区分两种不同类型的线程：前台线程和后台线程。这两者的区别就是：应用程序必须运行完所有的前台线程才可以退出；而对于后台线程，应用程序则可以不考虑其是否已经运行完毕而直接退出，所有的后台线程在应用程序退出时都会自动结束。

.Net环境使用Thread建立的线程默认情况下是前台线程，即线程属性IsBackground=false，在进程中，只要有一个前台线程未退出，进程就不会终止。主线程就是一个前台线程。
而后台线程不管线程是否结束，只要所有的前台线程都退出（包括正常退出和异常退出）后，进程就会自动终止。一般后台线程用于处理时间较短的任务，如在一个Web服务器中可以利用后台线程来处理客户端发过来的请求信息。而前台线程一般用于处理需要长时间等待的

任务，如在Web服务器中的监听客户端请求的程序，或是定时对某些系统资源进行扫描的程序。

```
class MainClass
{
     public static void Main (string[] args)
     {
          Console.WriteLine ("Hello World!");

          Thread backThread = new Thread (Worker);
          backThread.Start ();
          backThread.IsBackground = true;
          Console.WriteLine ("从主线程退出");
     }
     public static void Worker(){
          Thread.Sleep (1000);
          Console.WriteLine ("后台线程退出");
     }
}
```

首先通过Thread类创建了一个线程对象，然后通过IsBackground属性指明该线程为后台线程，如果不设置，默认为前台线程。

由于前台线程执行完毕后CLR会无条件的中止后台线程的运行，所以在前面的代码中，若是启动了后台线程，主线程将会继续执行。

主线程运行完毕之后就会中止后台线程，然后使整个程序结束运行。所以Worker不会执行。

修改方案如下：

第一种是将创建的线程设置为非后台线程，只需要注释backThread.IsBackground = true即可；

另一种方式就是使主线程在后台线程执行完毕之后再执行，即使主线程也进入睡眠，且使睡眠时间比后台线程更长，修改后的代码如下。

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

     {
```

```
        Console.WriteLine ("Hello World!");

        Thread backThread = new Thread (Worker);
        backThread.Start ();
        backThread.IsBackground = true;
        Thread.Sleep (2000);
        Console.WriteLine ("从主线程退出");
    }
    public static void Worker(){
        Thread.Sleep (1000);
        Console.WriteLine ("后台线程退出");

    }
```

此外还可以使用函数Join来实现，确保主线程会在后台线程结束后才开始运行。

修改之后的代码如下：

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

    {
        Console.WriteLine ("Hello World!");

        Thread backThread = new Thread (Worker);
        backThread.Start ();
        backThread.IsBackground = true;
        backThread.Join ();
        Console.WriteLine ("从主线程退出");
    }
    public static void Worker(){
        Thread.Sleep (1000);
        Console.WriteLine ("后台线程退出");

    }
```

使用Join的时候，主线程会等待后台线程结束之后才能继续执行。

前面的代码中，我们使用了Thread构造函数来创建线程对象。除此之外，还可以使用Thread(ParameterizedThreadStart)

```
    Thread paramThread = new Thread (new
ParameterizedThreadStart(Worker1));
                paramThread.Name = "线程1";
                paramThread.Start ("123");
}

    public static void Worker1(object data){
                Thread.Sleep (1000);
                Console.WriteLine ("传入的参数 "+data.ToString());
                Console.WriteLine ("后台线程退出");
          }
```

三：线程的容器——线程池

用来存放应用程序使用的线程的集合，可以理解为存放线程的地方，这种集中存放的方式有利于管理线程。

当应用程序想要执行一个异步操作时，需要调用QueueUserWorkItem方法将对应的任务添加到线程池中。线程池会从队列中提取任务，并且将其委派给线程池中的线程执行。

如果线程池中没有空闲的线程，就会创建一个新的线程去提取新的任务。当线程完成某个任务时候，线程也不会被销毁，而是返回到线程池中，等待另外一个请求。

默认都是后台线程。

下面我们通过代码学习线程池的基本使用

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

```
            Console.WriteLine ("主线程
ID={ 0}",Thread.CurrentThread.ManagedThreadId);
            ThreadPool.QueueUserWorkItem (CallBackWorkItem);
            ThreadPool.QueueUserWorkItem (CallBackWorkItem, "work");
            Thread.Sleep (3000);
            Console.WriteLine ("主线程退出");
            }

      public static void CallBackWorkItem(object state){
                  Console.WriteLine ("子线程执行");
                  if (state != null) {
                        Console.WriteLine ("ID={ 0}:
{ 1}",Thread.CurrentThread.ManagedThreadId,
                              state.ToString());
                  }  else {
                        Console.WriteLine
("Id={ 0}",Thread.CurrentThread.ManagedThreadId);
                  }
            }
```

四 线程同步技术

线程同步技术是指多线程程序中，为了保证后者线程，只有等待前者线程完成之后才能继续执行。就好比买票，前面的人没买到票之前，后面的人必须等待。

对线程技术存在的隐患：

当我们去访问一个共享资源的时候，有可能会同时去访问一个共享资源，这会破坏资源中的数据。下面我们模拟一下卖票的程序。

```
                  Thread thread1 = new Thread (SaleTicketThread1);
                  Thread thread2 = new Thread (SaleTicketThread2);
                  thread1.Start ();
                  thread2.Start ();

                  Thread.Sleep (4000);
            }
            private static void SaleTicketThread1(){
                  while (true) {
                        if (ticket > 0) {
                              Console.WriteLine ("线程1出票" + ticket--);
                        }  else {
                              break;
                        }
```

```
        }
    }

    private static void SaleTicketThread2(){
        while (true) {
            if (ticket > 0) {
                Console.WriteLine ("线程2出票" + ticket--);
            }  else {
                break;
            }
        }

    }
```

上面的代码存在问题，号码不连续，为了避免这种情况的发生，我们需要对多个线程进行同步处理，保证在同一个时刻只有一个线程访问共享资源，以及保证前面的线程售票完成之后，后面的才会访问资源。

4.1 使用监视器对象实现线程同步

监视器对象能够确保线程拥有对共享资源的互斥访问权，C#通过lock关键字提供简化的语法，下面我们使用线程同步技术来修改前面的代码。

```
private static void SaleTicketThread1(){
            while (true) {
                try{
                    Monitor.Enter(globalObj);
                    Thread.Sleep(1);
                if (ticket > 0) {
                    Console.WriteLine ("线程1出票" + ticket--);
                }  else {
                    break;
                }
                }finally{
                    Monitor.Exit(globalObj);
                }
        }
    }

    private static void SaleTicketThread2(){
        while (true) {
```

```
                    try{
                         Monitor.Enter(globalObj);
                         Thread.Sleep(1);
                         if (ticket > 0) {
                              Console.WriteLine ("线程2出票" +
ticket--);
                         }  else {
                              break;
                         }
                    }finally{
                         Monitor.Exit(globalObj);
                    }
               }

          }
```

修改代码，将try..finally去掉，查看最终结果分析原因。

线程同步技术带来的问题

- 使用繁琐，必须包装共享的数据
- 使用线程同步会影响性能
- 每次只允许一个线程访问，会阻塞线程。

所以在实际开发者，除非有必要，尽量不使用线程同步技术，避免使用共享数据。8

```
               int x = 0;
               const int iterationNumber = 5000000;
               Stopwatch sw = Stopwatch.StartNew();
               for (int i = 0; i < iterationNumber; i++) {
                    x++;
               }
               Console.WriteLine ("lost
time={ 0}",sw.ElapsedMilliseconds);


               sw.Restart ();
               for (int i = 0; i < iterationNumber; i++) {
                    Interlocked.Increment (ref x);
               }
               Console.WriteLine ("lost
time={ 0}",sw.ElapsedMilliseconds);
```