**一、使用线程的理由**

1、可以使用线程将代码同其他代码隔离，提高应用程序的可靠性。

2、可以使用线程来简化编码。

3、可以使用线程来实现并发执行。

**二、基本知识**

1、进程与线程：进程作为操作系统执行程序的基本单位，拥有应用程序的资源，进程包含线程，进程的资源被线程共享，线程不拥有资源。

2、前台线程和后台线程：通过Thread类新建线程默认为前台线程。当所有前台线程关闭时，所有的后台线程也会被直接终止，不会抛出异常。

3、挂起（Suspend）和唤醒（Resume）：由于线程的执行顺序和程序的执行情况不可预知，所以使用挂起和唤醒容易发生死锁的情况，在实际应用中应该尽量少用。

4、阻塞线程：Join，阻塞调用线程，直到该线程终止。

5、终止线程：Abort：抛出 ThreadAbortException 异常让线程终止，终止后的线程不可唤醒。Interrupt：抛出 ThreadInterruptException 异常让线程终止，通过捕获异常可以继续执行。

6、线程优先级：AboveNormal BelowNormal Highest Lowest Normal，默认为Normal。

**三、线程的使用**

线程函数通过委托传递，可以不带参数，也可以带参数（只能有一个参数），可以用一个类或结构体封装参数。

[复制代码](javascript:void(0);)

namespace Test

{

class Program

{

static void Main(string[] args)

{

Thread t1 = new Thread(new ThreadStart(TestMethod));

Thread t2 = new Thread(new ParameterizedThreadStart(TestMethod));

t1.IsBackground = true;

t2.IsBackground = true;

t1.Start();

t2.Start("hello");

Console.ReadKey();

}

public static void TestMethod()

{

Console.WriteLine("不带参数的线程函数");

}

public static void TestMethod(object data)

{

string datastr = data as string;

Console.WriteLine("带参数的线程函数，参数为：{0}", datastr);

}

}

}

[复制代码](javascript:void(0);)

**四、线程池**

由于线程的创建和销毁需要耗费一定的开销，过多的使用线程会造成内存资源的浪费，出于对性能的考虑，于是引入了线程池的概念。线程池维护一个请求队列，线程池的代码从队列提取任务，然后委派给线程池的一个线程执行，线程执行完不会被立即销毁，这样既可以在后台执行任务，又可以减少线程创建和销毁所带来的开销。

线程池线程默认为后台线程（IsBackground）。

[复制代码](javascript:void(0);)

namespace Test

{

class Program

{

static void Main(string[] args)

{

//将工作项加入到线程池队列中，这里可以传递一个线程参数

ThreadPool.QueueUserWorkItem(TestMethod, "Hello");

Console.ReadKey();

}

public static void TestMethod(object data)

{

string datastr = data as string;

Console.WriteLine(datastr);

}

}

}

[复制代码](javascript:void(0);)

**五、Task类**

使用ThreadPool的QueueUserWorkItem()方法发起一次异步的线程执行很简单，但是该方法最大的问题是没有一个内建的机制让你知道操作什么时候完成，有没有一个内建的机制在操作完成后获得一个返回值。为此，可以使用System.Threading.Tasks中的Task类。

构造一个Task<TResult>对象，并为泛型TResult参数传递一个操作的返回类型。

[复制代码](javascript:void(0);)

namespace Test

{

class Program

{

static void Main(string[] args)

{

Task<Int32> t = new Task<Int32>(n => Sum((Int32)n), 1000);

t.Start();

t.Wait();

Console.WriteLine(t.Result);

Console.ReadKey();

}

private static Int32 Sum(Int32 n)

{

Int32 sum = 0;

for (; n > 0; --n)

checked{ sum += n;} //结果太大，抛出异常

return sum;

}

}

}

[复制代码](javascript:void(0);)

一个任务完成时，自动启动一个新任务。  
一个任务完成后，它可以启动另一个任务，下面重写了前面的代码，不阻塞任何线程。

[复制代码](javascript:void(0);)

namespace Test

{

class Program

{

static void Main(string[] args)

{

Task<Int32> t = new Task<Int32>(n => Sum((Int32)n), 1000);

t.Start();

//t.Wait();

Task cwt = t.ContinueWith(task => Console.WriteLine("The result is {0}",t.Result));

Console.ReadKey();

}

private static Int32 Sum(Int32 n)

{

Int32 sum = 0;

for (; n > 0; --n)

checked{ sum += n;} //结果溢出，抛出异常

return sum;

}

}

}

[复制代码](javascript:void(0);)

**六、委托异步执行**

委托的异步调用：BeginInvoke() 和 EndInvoke()

[复制代码](javascript:void(0);)

namespace Test

{

public delegate string MyDelegate(object data);

class Program

{

static void Main(string[] args)

{

MyDelegate mydelegate = new MyDelegate(TestMethod);

IAsyncResult result = mydelegate.BeginInvoke("Thread Param", TestCallback, "Callback Param");

//异步执行完成

string resultstr = mydelegate.EndInvoke(result);

}

//线程函数

public static string TestMethod(object data)

{

string datastr = data as string;

return datastr;

}

//异步回调函数

public static void TestCallback(IAsyncResult data)

{

Console.WriteLine(data.AsyncState);

}

}

}

[复制代码](javascript:void(0);)

**七、线程同步**

　　1）原子操作（Interlocked）：所有方法都是执行一次原子读取或一次写入操作。

　　2）lock()语句：避免锁定public类型，否则实例将超出代码控制的范围，定义private对象来锁定。

　　3）Monitor实现线程同步

　　　　通过Monitor.Enter() 和 Monitor.Exit()实现排它锁的获取和释放，获取之后独占资源，不允许其他线程访问。

　　　　还有一个TryEnter方法，请求不到资源时不会阻塞等待，可以设置超时时间，获取不到直接返回false。

　　4）ReaderWriterLock

　　　　当对资源操作读多写少的时候，为了提高资源的利用率，让读操作锁为共享锁，多个线程可以并发读取资源，而写操作为独占锁，只允许一个线程操作。

　　5）事件（Event）类实现同步

　　　　事件类有两种状态，终止状态和非终止状态，终止状态时调用WaitOne可以请求成功，通过Set将时间状态设置为终止状态。

　　　　1）AutoResetEvent（自动重置事件）

　　　　2）ManualResetEvent（手动重置事件）

　　6）信号量（Semaphore）

　　　　　　信号量是由内核对象维护的int变量，为0时，线程阻塞，大于0时解除阻塞，当一个信号量上的等待线程解除阻塞后，信号量计数+1。

　　　　　　线程通过WaitOne将信号量减1，通过Release将信号量加1，使用很简单。

　　7）互斥体（Mutex）

　　　　　　独占资源，用法与Semaphore相似。

 　 8）跨进程间的同步

　　　　　　通过设置同步对象的名称就可以实现系统级的同步，不同应用程序通过同步对象的名称识别不同同步对象。