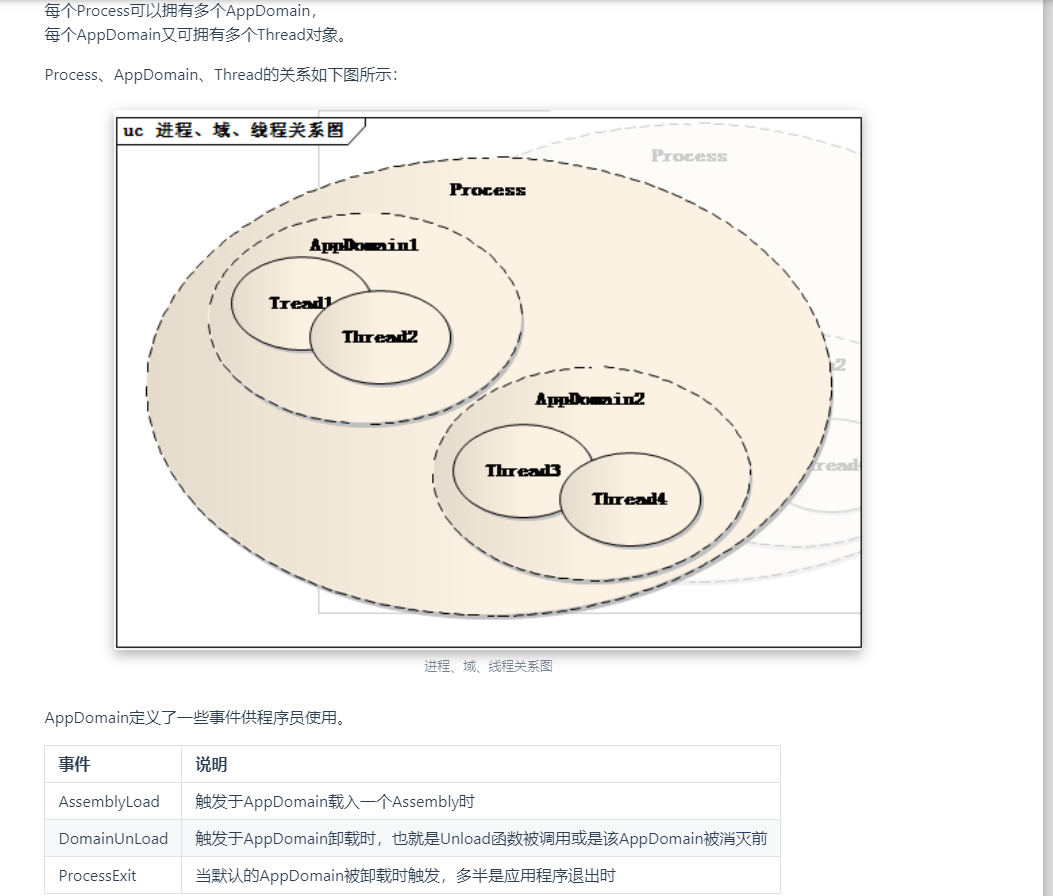
1. 进程/线程
   1. 进程：指一个应用程序所运行的操作系统单元（一个应用程序运行时所占用系统资源的总和）
      1. 进程在运行时创建的资源随着进程的终止而死亡
      2. 进程间获得专用数据或内存的唯一途径就是通过协议来共享内存块，这是一种策略
      3. 一个进程可以创建多个线程及子进程
      4. 一个进程内部的线程可以共享该进程所分配的资源
   2. 线程：进程可以分为若干个独立的执行流，这些执行流被称为线程
      1. 线程指进程内的一个执行单元，也是进程内的可调度实体
      2. 线程是进程的一个实体，是cpu调度和分配时间的基本单位
      3. 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间可以并发执行
      4. 线程提供了多任务的处理能力
   3. 线程与进程的异同
      1. 地址空间
         1. 进程拥有自己独立的内存地址空间
         2. 线程共享进程的地址空间
      2. 资源拥有
         1. 进程是资源分配和拥有的单位
         2. 同一进程内的线程共享进程的资源
      3. 系统粒度
         1. 进程是分配资源的基本单位
         2. 线程则是系统（处理器）调度的基本单位
      4. 执行过程
         1. 每个独立的进程都有一个程序运行入口、顺序执行序列和程序的出口
         2. 线程不能独立运行，必须依存于进程中
      5. 系统开销
         1. 创建或撤销进程时，系统都要为之分配或回收资源（如内存空间、IO设备）
         2. 进程间的切换的消耗远大于线程切换的开销
2. 并发/并行
   1. 在单cpu系统中，系统调度在某一时刻只能让一个线程运行，要通过不断切换需要运行的线程，这种运行模式称为并发
   2. 在多cpu系统中，可以让两个以上的线程同时运行，这种运行模式称为并行。
3. 同步/异步
   1. 同步，就是调用某个东西，调用方法等待这个调用返回结果之后才能继续执行
   2. 异步，和同步相反，调用发出后调用者可继续执行后续操作，被调用者通过状态来通知调用者，或者通过回调函数来处理掉这个调用。
4. 任务管理器
   1. 进程并不拥有独立于其所属实例的映射名称：5个记事本，会显示5个记事本的进程
   2. 他们是根据进程ID进行区分的，该进程ID由系统维护，并且可以循环使用
   3. CPU列：它是进程中线程所占用的CPU时间百分比
5. 线程是如何工作的
   1. 线程被一个线程协调程序管理着--一个CLR委托给操作系统的函数
   2. 线程协调程序确保所有活动的线程被分配适当的执行时间。
6. 线程安全
   1. 当使用线程的时候，需要注意线程同步的问题
   2. 当线程执行同一块资源的时候，需要上锁，并且在某一时刻，只能有一个线程执行这个操作，其余线程等待其执行完毕后，方可执行。
7. 使用多线程的情况分析
   1. 为什么要使用多线程
      1. 并发需要
         1. 服务端需处理来自不同终端的并发请求
      2. 提高应用程序的响应速度
         1. 使用多线程可将耗时的操作置于一个新的线程，此时程序仍能响应用户的其他操作
      3. 提高cpu利用率
         1. 在多cpu体系中，操作系统会保证当线程数不大于cpu数目时，不同的线程运行于不同的cpu上
      4. 改善程序结构
         1. 一个既长又复杂的进程可以考虑分为多个线程，称为几个独立或半独立的运行部分，这样的程序会利于理解和修改
      5. 花销小、切换快
         1. 线程间的切换消耗时间很小，可以忽略不计
      6. 方便的通信机制
         1. 线程间共享内存，互相交换数据很简单
   2. 何时使用多线程
      1. 多线程程序一般被用来在后台执行耗时的任务：主线程保持运行，而工作线程执行后台工作
      2. 远程服务器或webservice或asp.net程序将别无选择，必须使用多线程，需要关心的是提供适当锁机制的静态变量问题
   3. 何时不用多线程
      1. 当用户频繁的分配和切换线程时，多线程会带来增加资源和cpu的开销
      2. 在某些情况下，太多的I/O操作是非常棘手的，当只有一个或两个工作线程要比有众多线程在相同时间执行任务快得多。
8. C#中的多线程和异步
   1. 多线程适用场景：计算密集型的应用，视频图像处理，科学计算
   2. 异步适用场景：IO密集的应用，网络请求，读取文件资源，数据库访问
      1. 
      2. 各AppDomain之间互不影响。
   3. c#实现现成的方法
      1. BackgroundWorker类
      2. 线程池
      3. Threading Timer
9. 简介
   1. Thread
   2. ThreadPool
   3. Parallel
   4. Task
   5. BackgroundWorker组件
10. Thread类
    1. 使用Thread类通过ThreadStart（无参数）或ParameterizedThreadStart（一个输入参数）类型的委托创建一个Thread对象，开启一个新线程，执行该委托传递的任务，此时线程尚未处于运行状态
    2. 调用Start（）函数启动线程，当前线程继续执行
    3. 调用join（）函数可以阻塞当前线程，直到调用join（）的线程终止
    4. 调用Abort（）方法，如需中止线程，在调用该方法的线程上抛出ThreadAbrotException异常，以结束该线程
    5. 可以通过Thread，ResetAbort（）方法阻止线程的中止
    6. 属性表
       1. Name线程的名称
       2. Priority线程的优先级
       3. ThreadState当前线程的状态
       4. IsAlive当前线程是否处于启动状态
       5. IsBackground当前线程是否为后台线程
       6. CurrentThread当前正在运行的线程
    7. 方法表
       1. Start 启动线程
       2. Sleep 睡眠的毫秒数
       3. Join 阻塞调用线程，直到某个线程终止为止
       4. 中断当前线程
    8. 开启线程
       1. 首先new申请Thread对象，然后对象调用Start（）方法启用线程
       2. Thread thread = new Thread（方法名（）） thread.start（）
       3. 用Lambda表达式代替函数调用，也能达到相同的效果
       4. Thread thread = new Thread（（）=>
       5. {
       6. 方法体
       7. }）
       8. Thread.Start()
    9. 传递参数
       1. Start（）函数传参

Thread thread = new Thread(方法名)

Thread.Start(参数)

* + 1. 对象构造函数 a = new 对象构造函数（参数）

Thread thread = new Thread(方法)

Thread.Start()

* + 1. 匿名方法

static voidMain() {  
 stringtext = "Before";  
 Threadt = new Thread(delegate() { WriteText (text); });  
 text = "After";  
 t.Start();  
 }  
 static void WriteText (stringtext) { Console.WriteLine (text); }

* 1. 线程命名
     1. 通过他的name属性命名
  2. 前台线程和后台线程 IsBackground
     1. 前台线程（用户界面线程）
        1. 只要存在有一个前台线程在运行，应用程序就在运行
     2. 后台线程（工作线程）
        1. 应用程序关闭时，固若后台线程还没有执行完，会被强制性的关闭
  3. 注意事项
     1. Thread类创建的线程默认为前台线程，可以通过IsBackground属性设置其为前台或后台线程
     2. 通过Priority属性设置线程的优先级
     3. 线程内部可以通过Try catch捕获异常 在catch模块中进行一些必要的处理 如释放持有的锁和文件资源等
     4. 慎用Abort方法
        1. 如果在当前县城中抛出该异常，其结果是可预测的
        2. 但是对于其他线程，它会中断任何正在执行的代码，有可能中断静态对象的生成，造成不可预测的结果

1. 线程池
   1. 概述
      1. ThreadPool类维护一个线程的列表 提供给用户以执行不同的小任务 减少频繁创建线程的开销
      2. 该线程池可用于执行任务、发送工作项、处理异步IO、代表其他线程等待以及处理计时器
      3. 线程池其实就是一个存放线程对象的池子，他提供了一些基本方法 pool中最小最大线程数量、把要执行的方法排入队列。Threadpool是一个静态类，可以直接使用，不用创建对象。
   2. 优点
      1. 每新建一个线程都需要占用内存空间和其他资源，于是就提出了线程池的概念
      2. 线程池中的线程执行完方法后不会自动消除，而是以挂起的状态返回线程池，如果应用城区再次调用，则不需要重新创建线程，这就节约了很多开销。只有当线程数达到最大线程数量，系统才会自动销毁线程。
      3. 因此，使用线程池可以避免大量的创建和销毁的开支，具有更好的性能和稳定性。其次，开发人员把线程交给系统管理，可以集中精力处理其他任务。
   3. 使用
      1. 设置线程池最大最小
         1. ThreadPool.SetMaxThreads（int workerThreads,int completionPortThreads）
         2. 设置可以同时处于活动状态的线程池的请求数目，所有大于此数目的请求保持排队状态，直到线程池线程变为可用。还可以设置最小线程数
      2. 线程池中的线程均为后台线程，并且不能设置为前台线程
         1. 不能给入池的线程设置优先级或名称
         2. 对于com对象，入池的所有线程都是多线程单元（mta）线程，许多com对象都需要单线程单元（sta）线程
         3. 入池的线程只适合时间较短的任务，如果线程需要长时间运行，应使用Thraed类创建线程或使用task的longrunning选项
         4. .net下线程池最小默认允许4个工作线程，最大允许2048个工作线程。并发线程启动后，瞬间会启动4个线程，而剩下的会依据环境每0.5秒或1秒启动一个

如果同时运行的线程达到max工作线程，那么剩下的就会挂起

直到线程池中的线程又空闲得了，才会去执行。

1. Parallel类
   1. Parallel和task都位于system.Threading.Task命名空间中，是对Thread和ThreadPool类更高级的抽象。
      1. Parrallel类又For、ForEach、Invoke三个方法
      2. Invoke实现任务的并行性 允许同时调用不同的方法
      3. For和foreach实现数据并行性 在每次迭代中调用相同的代码
2. Task类
   1. 相比于Thread类，Task类为控制线程提供了更大的灵活性
      1. Task类可以获取线程的返回值
      2. 可以定义连续的任务，在一个任务结束后开启下一个任务
      3. 可以在层次结构中安排任务，在父任务中可以创建子任务，这样就创建了一种依赖关系，如果父任务被取消，子任务也随之取消
      4. Task类默认使用线程池中的线程，如果该任务需长期运行，应使用TaskCreationOptions.LongRunning属性告诉任务管理器创建一个新的线程，而不是使用线程池中的线程
   2. 任务task和线程thread的区别
      1. 任务是架构在线程之上的
      2. 任务和线程不是一对一的关系
      3. Task和Thread一样，位于System.Threading命名空间下
      4. Task是后台线程且不可设置为前台线程，Thread默认是前台线程且可以设置为后台线程
   3. Task的生存周期与状态
      1. Created 表示默认初始化任务，但是“工厂创建的”实例直接跳过
      2. Waiting 这种状态表示等待任务调度器分配线程给任务执行
      3. RanToCompletion 任务执行完毕
   4. Task的使用方法
      1. 启动任务
         1. 实例化后手动start
            1. Var task1 = new Task(()=>{代码段}); task1.Start()
         2. 使用task工厂对象创建新任务并执行
            1. TaskFactory tf = new TaskFactory();
            2. Task t1 = tf.StartNew(TaskMethod.DoTask,”using a task factory”)
         3. 工厂创建 直接执行
            1. Task t2 = Task.Factory.StartNew(TaskMethod.DoTask,”factory via a task”)
      2. 任务控制
         1. Task.Wait 就是等待任务执行（task1）完成，task1的状态变为Completed
         2. Task.WaitAll 等待所有的任务都执行完成
         3. Task.WaitAny 同Task.waitAll 等待任何一个任务完成就继续向下执行，将上面的代码WaitAll替换为WaitAny
         4. Task.ContinueWith 在第一个Task完成后自动启动下一个Task 实现Task的延续
         5. RunSunchronously 用于实现同步调用，直接在当前线程上调用该任务
      3. 任务取消
         1. 当我们启动了一个task，出现异常或者用户点击取消等等，我们可以取消这个任务
         2. 我们通过cancellation的tokens来取消一个task
         3. 在很多task的body里面包含循环，我们可以在轮询的时候判断IsCancellationRequested属性是否为True
         4. 如果是true的话就return或者抛出异常
      4. 接收任务的返回值
         1. 对于任务有返回值的情况，可使用task泛型类，TResult定义了返回值的类型
   5. 任务的层次结构
      1. 如果在一个Task内部创建了另一个任务，这两者间就存在父/子的层次结构，当父任务被取消时，子任务也会被取消。如果不希望使用该层次的结构，可在创建子任务时选择TaskCreationOptions.DetachedFromParent
3. BackgroundWorker控件
   1. 概述
      1. c#提供了BackgroundWorker控件帮助用户更简单、安全地实现多线程运算
   2. 属性表
      1. WorkerReportsProgress bool类型，指示BackgroundWorker是否可以报告进度更新
         1. True时，可以成功调用ReportProgress方法
         2. 否则将引发InvalidOperationException异常
      2. WorkerSupportsCancellation bool类型 指示BackgroundWorker是否支持异步取消操作
         1. True时，将可以成功调用CancelAsync方法
         2. 否则将引发InvalidOperationException异常
      3. CancellationPending bool类型 指示应用程序是否已请求取消后台操作
         1. 此属性农场放在用户执行的异步操作内部 用来判断用户是否取消执行异步操作
         2. 当执行BackgroundWorker.CancelAsync方法时，该属性值将变为true
      4. IsBusy
         1. bool类型，指示BackgroundWorker是否正在执行一个异步操作
         2. 此属性通常放在BackgroundWorker.RunworkerAsync方法之前，避免多次调用RunWorkerAsync方法引发异常
         3. 当执行BackgroundWorker.RunWorkerAsync方法时，该属性值将变为true
   3. 方法表
      1. RunWorkerAsync
         1. 开始执行一个后台操作 bgWorker.RunworkerAsync(“hello”)
      2. ReportProgress(Int percentProgress)
         1. 报告操作进度
      3. CancelAsync()
         1. 请求取消当前正在执行的异步操作
   4. 事件表
      1. DoWork 用于承载异步操作，当调用BackgroundWorker.RunWorkerAsync时触发
      2. ProgressChanged 当调用BackgroundWorker.ReportProgress(int percentProgress)方式时触发该事件
      3. RunWorkerCompleted 异步操作完成或取消时执行的操作，当调用DoWork事件执行完成时触发
   5. 案例

using System;  
using System.ComponentModel;  
using System.Threading;  
using System.Windows.Forms;  
   
namespace bcworker  
{  
 public partial class Form1 : Form  
 {  
 //后台工作  
 private BackgroundWorker bw = new BackgroundWorker();  
   
 public Form1()  
 {  
 InitializeComponent();  
 //后台工作初始化  
 bw.WorkerReportsProgress = true;//报告进度  
 bw.WorkerSupportsCancellation = true;//支持取消  
 bw.DoWork += new DoWorkEventHandler(bgWorker\_DoWork);//开始工作  
 bw.ProgressChanged += new ProgressChangedEventHandler(bgWorker\_ProgessChanged);//进度改变事件  
 bw.RunWorkerCompleted += new RunWorkerCompletedEventHandler(bgWorker\_WorkerCompleted);//进度完成事件  
 }  
   
 private void btnStart\_Click(object sender, EventArgs e)  
 {  
 //后台工作运行中，避免重入  
 if (bw.IsBusy) return;  
 bw.RunWorkerAsync("参数");//触发DoWork事件并异步执行，IsBusy置为True  
 }  
 //后台工作将异步执行  
 public void bgWorker\_DoWork(object sender, DoWorkEventArgs e)  
 {  
 //(string)e.Argument == "参数";  
 for (int i = 0; i <= 100; i++)  
 {  
   
 if (bw.CancellationPending)  
 {//用户取消了工作  
 e.Cancel = true;  
 return;  
 }  
 else  
 {  
 bw.ReportProgress(i, "Working");//报告进度，触发ProgressChanged事件  
 Thread.Sleep(10);//模拟工作  
 }  
 }  
 }  
 //进度改变事件  
 public void bgWorker\_ProgessChanged(object sender, ProgressChangedEventArgs e)  
 {  
 //(string)e.UserState=="Working"  
 progressBar1.Value = e.ProgressPercentage;//取得进度更新控件，不用Invoke了  
 }  
 //后台工作执行完毕,IsBusy置为False  
 public void bgWorker\_WorkerCompleted(object sender, RunWorkerCompletedEventArgs e)  
 {  
 //e.Error == null 是否发生错误  
 //e.Cancelled 完成是由于取消还是正常完成  
 }  
   
 private void btnCancel\_Click(object sender, EventArgs e)  
 {  
 if (bw.IsBusy) bw.CancelAsync();//设置CancellationPending属性为True  
 }  
   
 }  
}