  系列文章目录地址：

1. 描述线程与进程的区别？

2. 为什么GUI不支持跨线程访问控件？一般如何解决这个问题？

3. 简述后台线程和前台线程的区别？

4. 说说常用的锁，lock是一种什么样的锁？

5. lock为什么要锁定一个参数，可不可锁定一个值类型？这个参数有什么要求？

6. 多线程和异步有什么关系和区别？

7. 线程池的优点有哪些？又有哪些不足？

8. Mutex和lock有何不同？一般用哪一个作为锁使用更好？

9. 下面的代码，调用方法DeadLockTest（20），是否会引起死锁？并说明理由。

Public void DeadLockTest(int i)

{

lock (this) //或者lock一个静态object变量

{

if (i > 10)

{

Console.WriteLine(i--);

DeadLockTest(i);

}

}

}

10. 用双检锁实现一个单例模式Singleton。

11.下面代码输出结果是什么？为什么？如何改进她？

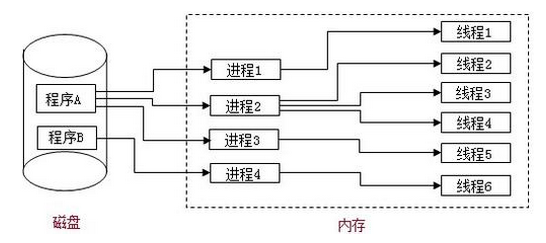
int a = 0;

System.Threading.Tasks.Parallel.For(0, 100000, (i) =>{ a++; });Console.Write(a);

**线程基础**

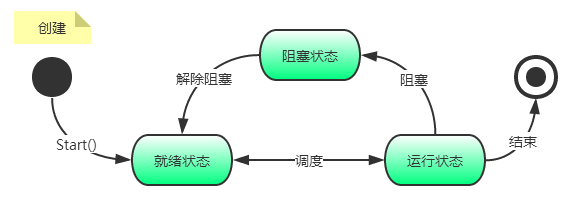
**进程与线程**

我们运行一个exe，就是一个进程实例，系统中有很多个进程。每一个进程都有自己的内存地址空间，每个进程相当于一个独立的边界，有自己的独占的资源，进程之间不能共享代码和数据空间。



image

每一个进程有一个或多个线程，进程内多个线程可以共享所属进程的资源和数据，**线程是操作系统调度的基本单元**。线程是由操作系统来调度和执行的，她的基本状态如下图。

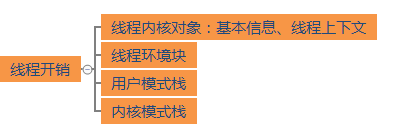


image

**线程的开销及调度**

当我们创建了一个线程后，线程里面到底有些什么东西呢？主要包括**线程内核对象**、**线程环境块**、**1M大小的用户模式栈**、**内核模式栈**。其中用户模式栈对于普通的系统线程那1M是预留的，在需要的时候才会分配，但是对于CLR线程，那1M是一开始就分类了内存空间的。

补充一句，CLR线程是直接对应于一个Windows线程的。



image

还记得以前学校里学习计算机课程里讲到，计算机的核心计算资源就是CPU核心和CPU寄存器，这也就是线程运行的主要战场。操作系统中那么多线程（一般都有上千个线程，大部分都处于休眠状态），对于单核CPU，一次只能有一个线程被调度执行，那么多线程怎么分配的呢？Windows系统采用时间轮询机制，CPU计算资源以时间片(大约30ms)的形式分配给执行线程。

计算机资源（CPU核心和CPU寄存器）一次只能调度一个线程，具体的调度流程：

1. 把CPU寄存器内的数据保存到当前线程内部（线程上下文等地方），给下一个线程腾地方；
2. 线程调度：在线程集合里取出一个需要执行的线程；
3. 加载新线程的上下文数据到CPU寄存器；
4. 新线程执行，享受她自己的CPU时间片（大约30ms），完了之后继续回到第一步，继续轮回；

上面线程调度的过程，就是一次线程切换，一次切换就涉及到线程上下文等数据的搬入搬出，性能开销是很大的。因此线程不可滥用，线程的创建和消费也是很昂贵的，这也是为什么建议尽量使用线程池的一个主要原因。

对于Thread的使用太简单了，这里就不重复了，**总结一下线程的主要几点性能影响**：

1. 线程的创建、销毁都是很昂贵的；
2. 线程上下文切换有极大的性能开销，当然假如需要调度的新线程与当前是同一线程的话，就不需要线程上下文切换了，效率要快很多；
3. 这一点需要注意，**GC执行回收时，首先要（安全的）挂起所有线程，遍历所有线程栈（根），GC回收后更新所有线程的根地址，再恢复线程调用，线程越多，GC要干的活就越多；**

当然现在硬件的发展，CPU的核心越来越多，多线程技术可以极大提高应用程序的效率。但这也必须在合理利用多线程技术的前提下，了线程的基本原理，然后根据实际需求，还要注意相关资源环境，如磁盘IO、网络等情况综合考虑。

**多线程**

单线程的使用这里就略过了，那太easy了。上面总结了线程的诸多不足，因此微软提供了可供多线程编程的各种技术，如线程池、任务、并行等等。

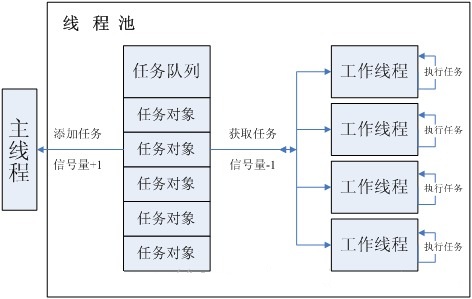
**线程池ThreadPool**

线程池的使用是非常简单的，如下面的代码，把需要执行的代码提交到线程池，线程池内部会安排一个空闲的线程来执行你的代码，完全不用管理内部是如何进行线程调度的。

ThreadPool.QueueUserWorkItem(t => Console.WriteLine("Hello thread pool"));

每个CLR都有一个线程池，线程池在CLR内可以多个AppDomain共享，**线程池是CLR内部管理的一个线程集合**，初始是没有线程的，在需要的时候才会创建。线程池的主要结构图如下图所示，基本流程如下：

1. 线程池内部维护一个请求列队，用于缓存用户请求需要执行的代码任务，就是ThreadPool.QueueUserWorkItem提交的请求；
2. 有新任务后，线程池使用空闲线程或新线程来执行队列请求；
3. 任务执行完后线程不会销毁，留着重复使用；
4. 线程池自己负责维护线程的创建和销毁，当线程池中有大量闲置的线程时，线程池会自动结束一部分多余的线程来释放资源；



线程池是有一个容量的，因为他是一个池子嘛，可以设置线程池的最大活跃线程数，调用方法ThreadPool.SetMaxThreads可以设置相关参数。但很多编程实践里都不建议程序猿们自己去设置这些参数，其实微软为了提高线程池性能，做了大量的优化，线程池可以很智能的确定是否要创建或是消费线程，大多数情况都可以满足需求了。

线程池使得线程可以充分有效地被利用，减少了任务启动的延迟，也不用大量的去创建线程，避免了大量线程的创建和销毁对性能的极大影响。

上面了解了线程的基本原理和诸多优点后，如果你是一个爱思考的猿类，应该会很容易发现很多疑问，比如把任务添加到线程池队列后，怎么取消或挂起呢？如何知道她执行完了呢？下面来总结一下线程池的不足：

1. 线程池内的线程不支持线程的挂起、取消等操作，如想要取消线程里的任务，.NET支持一种**协作式**方式取消，使用起来也不是很方便，而且有些场景并不满足需求；
2. 线程内的任务没有返回值，也不知道何时执行完成；
3. 不支持设置线程的优先级，还包括其他类似需要对线程有更多的控制的需求都不支持；

因此微软为我们提供了另外一个东西叫做Task来补充线程池的某些不足。

**任务Task与并行Parallel**

任务Task与并行Parallel**本质上内部都是使用的线程池**，提供了更丰富的并行编程的方式。任务Task基于线程池，可支持返回值，支持比较强大的任务执行计划定制等功能，下面是一个简单的示例。Task提供了很多方法和属性，通过这些方法和属性能够对Task的执行进行控制，并且能够获得其状态信息。Task的创建和执行都是独立的，因此可以对关联操作的执行拥有完全的控制权。

//创建一个任务

Task<int> t1 = new Task<int>(n =>{ System.Threading.Thread.Sleep(1000); return (int)n;}, 1000);

//定制一个延续任务计划

t1.ContinueWith(task =>{ Console.WriteLine("end" + t1.Result);},TaskContinuationOptions.AttachedToParent);

t1.Start();

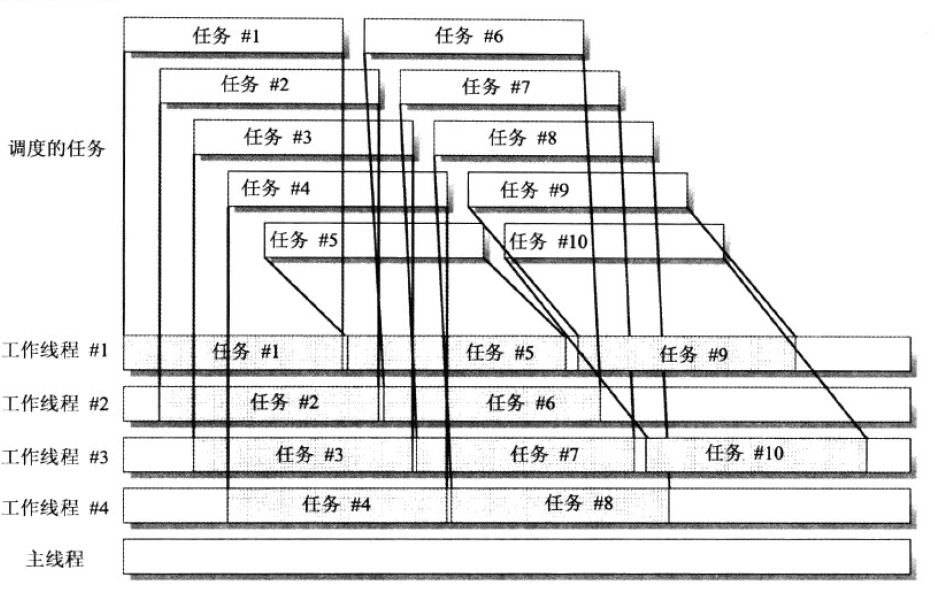
//使用Task.Factory创建并启动一个任务

var t2 = System.Threading.Tasks.Task.Factory.StartNew(() =>{ Console.WriteLine("t1:" + t1.Status);});

Task.WaitAll();

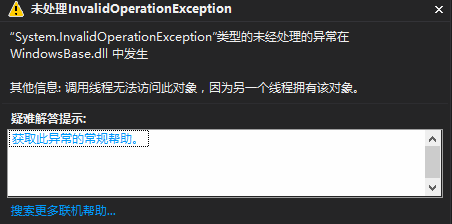
Console.WriteLine(t1.Result);

并行Parallel内部其实使用的是Task对象（TPL会在内部创建System.Threading.Tasks.Task的实例），所有并行任务完成后才会返回。少量短时间任务建议就不要使用并行Parallel了，并行Parallel本身也是有性能开销的，而且还要进行并行任务调度、创建调用方法的委托等等。



**GUI线程处理模型**

这是很多开发C/S客户端应用程序会遇到的问题，GUI程序的界面控件不允许跨线程访问，如果在其他线程中访问了界面控件，运行时就会抛出一个异常，就像下面的图示，是不是很熟悉！这其中的罪魁祸首就是，就是“GUI的线程处理模型”。

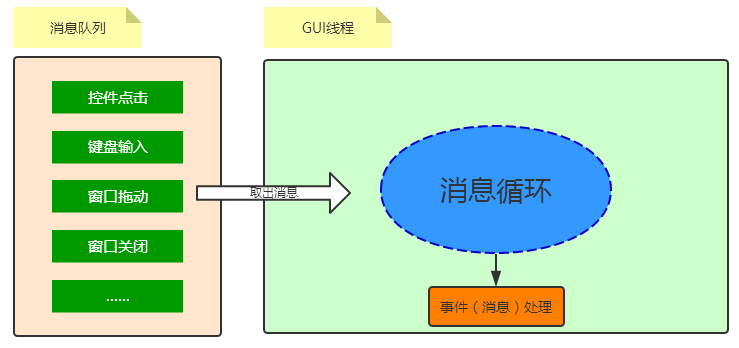


image

.NET支持多种不同应用程序模型，大多数的线程都是可以做任何事情（他们可能没有引入线程模型），但GUI应用程序（主要是Winform、WPF）引入了一个特殊线程处理模型，UI控件元素只能由创建它的线程访问或修改，微软这样处理是为了保证UI控件的线程安全。

为什么在UI线程中执行一个耗时的计算操作，会导致UI假死呢？这个问题要追溯到Windows的消息机制了。

因为Windows是基于消息机制的，我们在UI上所有的键盘、鼠标操作都是以消息的形式发送给各个应用程序的。GUI线程内部就有一个消息队列，GUI线程不断的循环处理这些消息，并根据消息更新UI的呈现。如果这个时候，你让GUI线程去处理一个耗时的操作（比如花10秒去下载一个文件），那GUI线程就没办法处理消息队列了，UI界面就处于假死的状态。



image

那我们该怎么办呢？不难想到使用线程，那在线程里处理事件完成后，需要更新UI控件的状态，又该怎么办呢？常用几种方式：

**① 使用GUI控件提供的方法，Winform是控件的Invoke方法，WPF中是控件的Dispatcher.Invoke方法**

//1.Winform：Invoke方法和BeginInvokethis.label.Invoke(method, null);

//2.WPF：Dispatcher.Invokethis.label.Dispatcher.Invoke(method, null);

**② 使用.NET中提供的BackgroundWorker执行耗时计算操作，在其任务完成事件RunWorkerCompleted 中更新UI控件**

using (BackgroundWorker bw = new BackgroundWorker()){

bw.RunWorkerCompleted += new RunWorkerCompletedEventHandler((ojb,arg) => { this.label.Text = "anidng"; });

bw.RunWorkerAsync();}

**③ 看上去很高大上的方法：使用GUI线程处理模型的同步上下文来送封UI控件修改操作，这样可以不需要调用UI控件元素**

.NET中提供一个用于同步上下文的类SynchronizationContext，利用它可以把应用程序模型链接到他的线程处理模型，其实它的本质还是调用的第一步**①**中的方法。

实现代码分为三步，第一步定义一个静态类，用于GUI线程的UI元素访问封装：

Public static class GUIThreadHelper

{

Public static System.Threading.SynchronizationContext GUISyncContext

{

get { return \_GUISyncContext; }

set { \_GUISyncContext = value; }

}

Private static System.Threading.SynchronizationContext \_GUISyncContext = System.Threading.SynchronizationContext.Current;

///

/// 主要用于GUI线程的同步回调

///

///

Public static void SyncContextCallback(Action callback)

{

if (callback == null)

return;

if (GUISyncContext == null)

{

callback();

return;

}

GUISyncContext.Post(result => callback(), null);

}

///

/// 支持APM异步编程模型的GUI线程的同步回调

///

Public static AsyncCallback SyncContextCallback(AsyncCallback callback)

{

if (callback == null)

return callback;

if (GUISyncContext == null)

return callback;

return asynresult => GUISyncContext.Post(result => callback(result as IAsyncResult), asynresult);

}

}

第二步，在主窗口注册当前SynchronizationContext：

publicpartialclass MainWindow : Window

{

public MainWindow()

{

InitializeComponent();

CLRTest.ConsoleTest.GUIThreadHelper.GUISyncContext = System.Threading.SynchronizationContext.Current;

}

}

第三步，就是使用了，可以在任何地方使用

GUIThreadHelper.SyncContextCallback(() =>{ this.txtMessage.Text = res.ToString(); this.btnTest.Content = "DoTest"; this.btnTest.IsEnabled = true;});

**线程同步构造**

多线程编程中很常用、也很重要的一点就是线程同步问题，掌握线程同步对临界资源正确使用、线程性能有至关重要的作用！基本思路是很简单的，就是加锁嘛，在临界资源的门口加一把锁，来控制多个线程对临界资源的访问。但在实际开发中，根据资源类型不同、线程访问方式的不同，有多种锁的方式或控制机制（基元**用户**模式构造和基元**内核**模式构造）。.NET提供了两种线程同步的构造模式，需要理解其基本原理和使用方式。

基元线程同步构造分为：基元**用户**模式构造和基元**内核**模式构造，两种同步构造方式各有优缺点，而混合构造（如lock）就是综合两种构造模式的优点。

**用户模式构造**

基元用户模式比基元内核模式速度要快，她使用特殊的cpu指令来协调线程，在硬件中发生，速度很快。但也因此Windows操作系统永远检测不到一个线程在一个用户模式构造上阻塞了。举个例子来模拟一下用户模式构造的同步方式：

1. 线程1请求了临界资源，并在资源门口使用了用户模式构造的锁；
2. 线程2请求临界资源时，发现有锁，因此就在门口等待，并不停的去询问资源是否可用；
3. 线程1如果使用资源时间较长，则线程2会一直运行，并且占用CPU时间。占用CPU干什么呢？她会不停的轮询锁的状态，直到资源可用，这就是所谓的活锁；

缺点有没有发现？线程2会一直使用CPU时间（假如当前系统只有这两个线程在运行），**也就意味着不仅浪费了CPU时间，而且还会有频繁的线程上下文切换，对性能影响是很严重的**。

当然她的优点是效率高，适合哪种对资源占用时间很短的线程同步。.NET中为我们提供了两种原子性操作，利用原子操作可以实现一些简单的用户模式锁（如自旋锁）。

**System.Threading.Interlocked**：易失构造，它在包含一个简单数据类型的变量上执行原子性的读**或**写操作。

**Thread.VolatileRead 和 Thread.VolatileWrite**：互锁构造，它在包含一个简单数据类型的变量上执行原子性的读**和**写操作。

以上两种原子性操作的具体内涵这里就细说了（有兴趣可以去研究文末给出的参考书籍或资料），针对题目11，来看一下题目代码：

int a = 0;System.Threading.Tasks.Parallel.For(0, 100000, (i) =>{ a++; });Console.Write(a);

上面代码是通过并行（多线程）来更新共享变量a的值，结果肯定是小于等于100000的，具体多少是不稳定的。解决方法，可以使用我们常用的Lock，还有更有效的就是使用**System.Threading.Interlocked**提供的原子性操作，保证对a的值操作每一次都是原子性的：

System.Threading.Interlocked.Add(ref a, 1);//正确

下面的图是一个简单的性能验证测试，分别使用Interlocked、不用锁、使用lock锁三种方式来测试。不用锁的结果是95，这答案肯定不是你想要的，另外两种结果都是对的，性能差别却很大。

DD4708DB3FEE42DD9FE9E32B17AC633D.png

image

为了模拟耗时操作，对代码稍作了修改，如下，所有的循环里面加了代码Thread.Sleep(20);。如果没有Thread.Sleep(20);他们的执行时间是差不多的。

System.Threading.Tasks.Parallel.For(0, 100, (i) =>{ lock (\_obj) { a++; //不正确 Thread.Sleep(20); }});

**内核模式构造**

这是针对用户模式的一个补充，先模拟一个内核模式构造的同步流程来理解她的工作方式：

1. 线程1请求了临界资源，并在资源门口使用了内核模式构造的锁；
2. 线程2请求临界资源时，发现有锁，就会被系统要求睡眠（阻塞），线程2就不会被执行了，也就不会浪费CPU和线程上下文切换了；
3. 等待线程1使用完资源后，解锁后会发送一个通知，然后操作系统会把线程2唤醒。假如有多个线程在临界资源门口等待，则会挑选一个唤醒；

看上去是不是非常棒！彻底解决了用户模式构造的缺点，但内核模式也有缺点的：将线程从用户模式切换到内核模式（或相反）导致巨大性能损失。调用线程将从托管代码转换为内核代码，再转回来，会浪费大量CPU时间，同时还伴随着线程上下文切换，因此尽量不要让线程从用户模式转到内核模式。

她的优点就是阻塞线程，不浪费CPU时间，适合那种需要长时间占用资源的线程同步。

内核模式构造的主要有两种方式，以及基于这两种方式的常见的锁：

1. **基于事件**：如AutoResetEvent、ManualResetEvent
2. **基于信号量**：如Semaphore

**混合线程同步**

既然内核模式和用户模式都有优缺点，混合构造就是把两者结合，充分利用两者的优点，把性能损失降到最低。大概的思路很好理解，就是如果是在没有资源竞争，或线程使用资源的时间很短，就是用用户模式构造同步，否则就升级到内核模式构造同步，其中最典型的代表就是Lock了。

常用的混合锁还不少呢！如SemaphoreSlim、ManualResetEventSlim、Monitor、ReadWriteLockSlim，这些锁各有特点和锁使用的场景。这里主要就使用最多的lock来详细了解下。

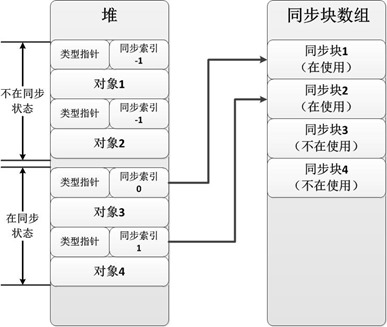
lock的本质就是使用的Monitor，lock只是一种简化的语法形式，实质的语法形式如下：

bool lockTaken = false;try{ Monitor.Enter(obj, ref lockTaken); //...}finally{ if (lockTaken) Monitor.Exit(obj);}

那lock或Monitor需要锁定的那个对象是什么呢？注意这个对象才是锁的关键，在此之前，需要先回顾一下引用对象的同步索引块（AsynBlockIndex），这是前面文章中提到过的引用对象的标准配置之一（还有一个是类型对象指针TypeHandle），它的作用就在这里了。

同步索引块是.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，她其实是一个地址指针，初始值为-1不指向任何地址。

1. 创建一个锁对象Object obj，obj的同步索引块（地址）为-1，不指向任何地址；
2. Monitor.Enter（obj），创建或使用一个空闲的同步索引块（如下图中的同步块1），（[图片来源](http://www.cnblogs.com/edisonchou/p/4848131.html)），这个才是真正的同步索引块，其内部结构就是一个混合锁的结构，包含线程ID、递归计数、等待线程统计、内核对象等，类似一个混合锁AnotherHybridLock。obj对象（同步索引块AsynBlockIndex）指向该同步块1；
3. Exit时，重置为-1，那个同步索引块1可以被重复利用；



381412-20150930224247574-1653709348

因此，锁对象要求必须为一个引用对象（在堆上）。

**多线程使用及线程同步总结**

首先还是尽量避免线程同步，不管使用什么方式都有不小的性能损失。一般情况下，大多使用Lock，这个锁是比较综合的，适应大部分场景。在性能要求高的地方，或者根据不同的使用场景，可以选择更符合要求的锁。

在使用Lock时，关键点就是锁对象了，需要注意以下几个方面：

1. 这个对象肯定要是引用类型，值类型可不可呢？值类型可以装箱啊！你觉得可不可以？但也不要用值类型，因为值类型多次装箱后的对象是不同的，会导致无法锁定；
2. 不要锁定this，尽量使用一个没有意义的Object对象来锁；
3. 不要锁定一个类型对象，因类型对象是全局的；
4. 不要锁定一个字符串，因为字符串可能被驻留，不同字符对象可能指向同一个字符串；
5. 不要使用[System.Runtime.CompilerServices.MethodImpl(MethodImplOptions.Synchronized)]，这个可以使用在方法上面，保证方法同一时刻只能被一个线程调用。她实质上是使用lock的，如果是实例方法，会锁定this，如果是静态方法，则会锁定类型对象；

**题目答案解析:**

**1. 描述线程与进程的区别？**

1. 一个应用程序实例是一个进程，一个进程内包含一个或多个线程，线程是进程的一部分；
2. 进程之间是相互独立的，他们有各自的私有内存空间和资源，进程内的线程可以共享其所属进程的所有资源；

**2. 为什么GUI不支持跨线程访问控件？一般如何解决这个问题？**

因为GUI应用程序引入了一个特殊的线程处理模型，为了保证UI控件的线程安全，这个线程处理模型不允许其他子线程跨线程访问UI元素。解决方法还是比较多的，如：

1. 利用UI控件提供的方法，Winform是控件的Invoke方法，WPF中是控件的Dispatcher.Invoke方法；
2. 使用BackgroundWorker；
3. 使用GUI线程处理模型的同步上下文SynchronizationContext来提交UI更新操作

上面几个方式在文中已详细给出。

**3. 简述后台线程和前台线程的区别？**

应用程序必须运行完所有的前台线程才可以退出，或者主动结束前台线程，不管后台线程是否还在运行，应用程序都会结束；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。

通过将 Thread.IsBackground 设置为 true，就可以将线程指定为后台线程，主线程就是一个前台线程。

**4. 说说常用的锁，lock是一种什么样的锁？**

常用的如如SemaphoreSlim、ManualResetEventSlim、Monitor、ReadWriteLockSlim，lock是一个混合锁，其实质是Monitor['mɒnɪtə]。

**5. lock为什么要锁定一个参数，可不可锁定一个值类型？这个参数有什么要求？**

lock的锁对象要求为一个引用类型。她可以锁定值类型，但值类型会被装箱，每次装箱后的对象都不一样，会导致锁定无效。

对于lock锁，锁定的这个对象参数才是关键，这个参数的同步索引块指针会指向一个真正的锁（同步块），这个锁（同步块）会被复用。

**6. 多线程和异步有什么关系和区别？**

多线程是实现异步的主要方式之一，异步并不等同于多线程。实现异步的方式还有很多，比如利用硬件的特性、使用进程或纤程等。在.NET中就有很多的异步编程支持，比如很多地方都有Begin\*\*\*、End\*\*\*的方法，就是一种异步编程支持，她内部有些是利用多线程，有些是利用硬件的特性来实现的异步编程。

**7. 线程池的优点有哪些？又有哪些不足？**

优点：减小线程创建和销毁的开销，可以复用线程；也从而减少了线程上下文切换的性能损失；在GC回收时，较少的线程更有利于GC的回收效率。

缺点：线程池无法对一个线程有更多的精确的控制，如了解其运行状态等；不能设置线程的优先级；加入到线程池的任务（方法）不能有返回值；对于需要长期运行的任务就不适合线程池。

**8. Mutex和lock有何不同？一般用哪一个作为锁使用更好？**

Mutex是一个基于内核模式的互斥锁，支持锁的递归调用，而Lock是一个混合锁，一般建议使用Lock更好，因为lock的性能更好。

**9. 下面的代码，调用方法DeadLockTest（20），是否会引起死锁？并说明理由。**

Public void DeadLockTest(int i)

{

lock (this) //或者lock一个静态object变量

{

if (i > 10)

{

Console.WriteLine(i--);

DeadLockTest(i);

}

}

}

不会的，因为lock是一个混合锁，支持锁的递归调用，如果你使用一个ManualResetEvent或AutoResetEvent可能就会发生死锁。

**10. 用双检锁实现一个单例模式Singleton。**

Public static class Singleton where T : class,new()

{

privatestatic T \_Instance;

privatestaticobject \_lockObj = newobject();

///

/// 获取单例对象的实例

///

publicstatic T GetInstance()

{

if (\_Instance != null)

return \_Instance;

lock (\_lockObj)

{

if (\_Instance == null)

{

var temp = Activator.CreateInstance(); System.Threading.Interlocked.Exchange(ref \_Instance, temp);

}

}

return \_Instance;

}

}

**11.下面代码输出结果是什么？为什么？如何改进她？**

int a = 0;

System.Threading.Tasks.Parallel.For(0, 100000, (i) =>{ a++; });Console.Write(a);

输出结果不稳定，小于等于100000。因为多线程访问，没有使用锁机制，会导致有更新丢失。具体原因和改进在文中已经详细的给出了。

**参考资料:**