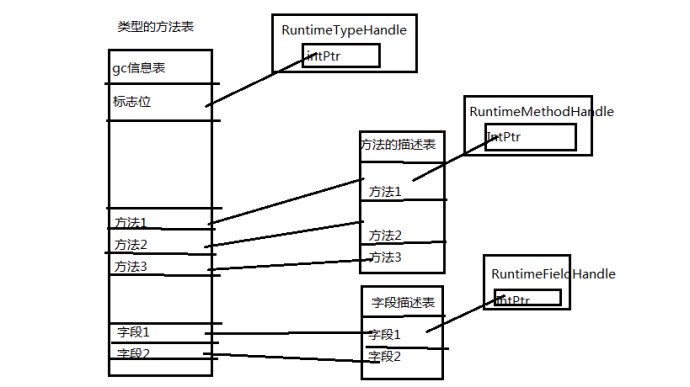
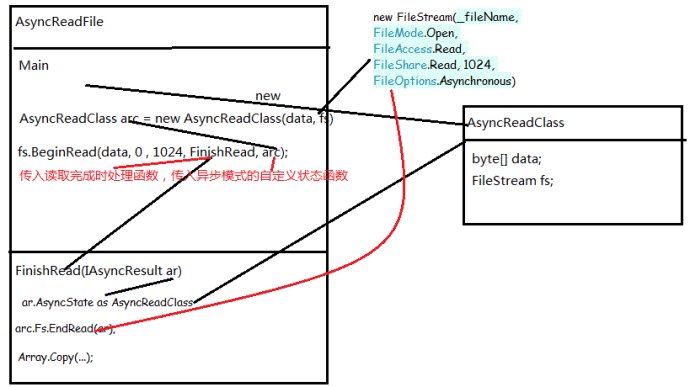
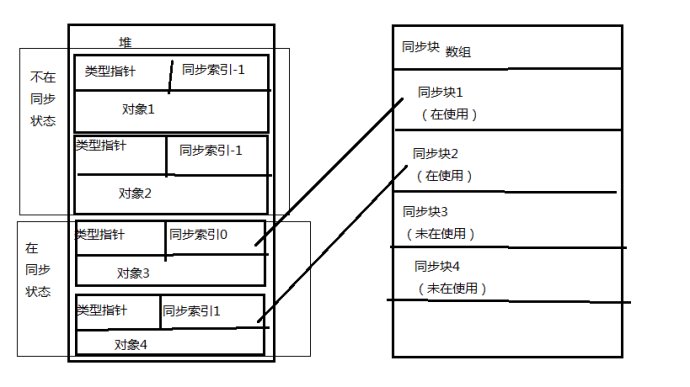
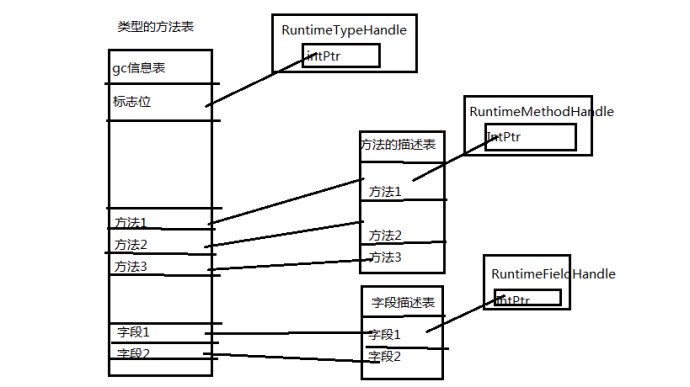
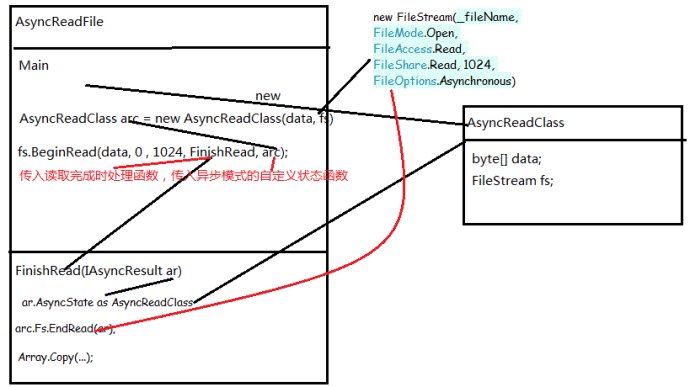
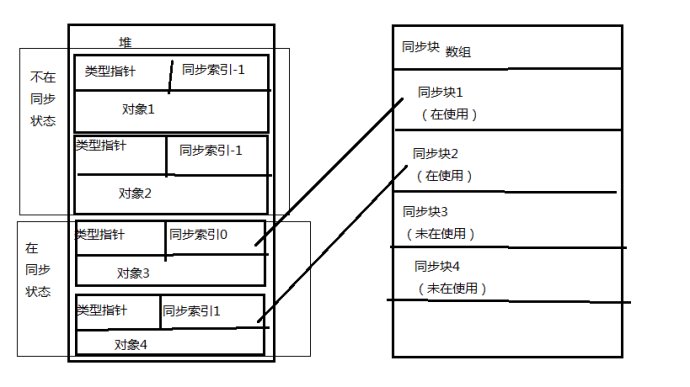
高级特性，多线程编程，单元测试;  
  
第一部分 .net高级特性  
1.委托:提供安全的函数回调机制;  
\*基本原理:与C++中的函数指针相似;安全--它和其他所有.net成员一样是一种类型，任何委托都是System.Delegate的某个派生类对象;  
System.Object->SystemDelegate(ISerializable,ICloneable)->System.MultiCastDelegate  
->Delegate T;  
public delegate void TestDelegate(int i)定义一个委托，内部包含invoke方法(由编译器自动完成);委托的调用其实就是执行类定义委托时生成的Invoke方法;  
总结:每个委托至少包含一个指向回调方法的指针，该方法可以是实例方法，也可以是静态方法。委托实现回调方法的机制，方便使用;  
\*委托回调静态方法和实例方法的区别;  
当委托绑定静态方法时，target为null;绑定实例方法是，target指向该实例方法所属类型的一个实例对象;  
\*什么是链式委托:指一个由委托组成的链表，而不是指另一个特殊的委托;prev，当一个委托被调用时，所有链表上的该委托的后续委托均被顺序执行;  
\*链式委托的执行顺序:按照委托链上的顺序从当前委托开始依次向后执行。如果有需要，可以通过GetInvocationList()来获得委托链上所有需要执行的委托，并且按照任何希望的顺序去执行;  
\*可否定义有返回值的方法的委托链:委托可以是带有返回值的方法，但多于一个待返回值的方法被添加到委托链是，程序员需要手动调用每一个委托链上的方法;  
\*委托的应用场合:任务的执行者把细节工作进行再分配，执行者确切地知道什么工作将要被执行，但却把执行细节委托给其他组件，方法或程序集;一些简单的重点记录:(给个程序集中有多个模块的例子)  
  
2.事件(Event)  
定义:是一种使对象或类能够提供通知的成员，客户端可以通过提供事件处理程序为相应的事件添加可执行代码;  
event标记\_\_委托类型(如EventHandler)\_\_声明的事件对象;  
使用事件的步骤:  
如果需要，定义一个派生自System.EventArgs的参数类型;  ConsoleEventArgs  
在事件的管理类型中定义事件的私有成员;                ConsoleManagsder.ConsoleEvent  
通知事件订阅者;                                     ConsoleManager.SendConsoleEvent  
事件使用客户端订阅/取消订阅事件;                     Log  
补充StreamWriter,StreamReader,各种Stream:  
using(FileStream fs = File.Create(...)){}  
using(StreamWriter sw =info.AppendText()){}  
\*事件和委托有何联系?  
实际上，事件就是就是一个委托类型，当程序员定义一个事件的时候，实际上是定义了一个特殊的委托成员。它没有返回值，有固定的两个参数 sender,EventArgs;而事件的使用者订阅事件时，本质就是把事件处理方法加入到委托链表中;声明event时默认会添加一对针对特定委托的 add/remove方法;  
\*如何设计一个带很多事件的类型？使用EventHandlerList(System.ComponentModel);  
注意:考虑线程同步措施;每个事件定义一套成员--事件的委托原型，事件的订阅和取消订阅方法，定义事件的专用参数类型;减少多事件类型的大小，但代码量增加;  
\*用代码表示如下情形:猫叫，老鼠逃跑,主人惊醒<==>猫叫被老鼠和主人订阅;  
  
3.反射  
\*反射的原理和实现它的基石:反射是一种动态分析程序集，模块，类型，字段等目标对象的机制，它的实现依托于元数据(基石)。元数据是存储在PE文件中的数据块，他详细记录了程序集或模块内部的结构，引用类型，程序集清单.TypeDef,TypeRef&AssemblyRef,Assembly；  
\*.net提供哪些类型来实现反 射:Assembly,AssemblyName,EventInfo,LocalVariableInfo,ManifestResourceInfo,MemberInfo,MethodBase,MethodBody,Module,ParameterInfo,PropertyInfo;  
\*如何动态的发射程序集:动态的生成一个程序集。而不仅仅是分析程序经济或者生成程序集对象;在System.Reflection.Emit命名空间 下，定义了一系列用于动态发射程序集，模块，类型，方法等元素的中间代码。这些类型的主要使用者是编译器，特殊反射工具或者脚本解释器.  
AssemblyBuilder,ConstructorBuilder,CustomerAttributeBuilder,EnumBuilder,EventBuilder,FieldBuilder,GenericTypeParameterBuilder,ILGenerator,LocalBuilder,MethodBuilder,OpCodes,ParameterBuilder,PropertyBuilder,TypeBuilder;  
//步骤：  
在当前appDomain中定义新程序集: AppDomain myDomain = Thread.GetDomain();  
定义模块:                     assemblyBuilder.DefineDynamicModule();  
定义模块中的类型:              Typebuilder addClass=assemblyModule.DefineType();  
定义公共构造方法及其参数       Type[] ctorParams = new Type[]{typeof(long),typeof(long)};  
            构造方法中间代码   ctorIL.Emit(OpCodes.Ldarg\_0);  
动态发送程序集:  
object  ptInstance=Activator.CreateInstance(type, ctorParams);  
MethodInfo info = type.GetMethod("GetResult", new Type[0]);//通过方法名和返回值类型得到方法;  
OpCodes类型包含大多数需要使用的中间代码指令;  
注意:新的程序集发射是直接把中间代码在内存中生成一种机制，而不是在物理硬盘上生成代码的机制;  
\*利用反射来实现工程模式,动手;  
'  
    //针对每一个product做相应的Attribute,针对product系列做Attribute，还是需要参考  
    [AttributeUsage(AttributeTargets.Class)]  
    public class ProductAttribute : Attribute  
    {RoomPart \_myRP = new RoomPart();  
        public ProductAttribute(RoomPart rp)  
        {\_myRP = rp;  }  
        public RoomPart MyRoomPart  
        {get{ return \_myRP;}  
        } }  
  
    [AttributeUsage(AttributeTargets.Interface)]  
    public class ProductListAttribute : Attribute  
    {Type[] \_mylist;  
        public ProductListAttribute(Type[] productList)  
        {\_mylist = productList;}  
        public Type[] MyProductList  
        {get { return \_mylist; } }  
    }  
public IProduct  Produce(RoomPart rp)  
{//通过反射，从IProduct接口获得属性，从而获得所有的产品零件列表  
    ProductListAttribute pla = (ProductListAttribute)Attribute.GetCustomAttribute(typeof(IProduct), typeof(ProductListAttribute));  
    //遍历所有的实现产品零件类型  
    foreach (Type type in pla.MyProductList) {  
        ProductAttribute pa = (ProductAttribute)Attribute.GetCustomAttribute(type, typeof(ProductAttribute));  
        if (rp == pa.MyRoomPart) {  
           object pro =  Assembly.GetExecutingAssembly().CreateInstance(type.FullName);  
           return pro as IProduct;  
        }  
        return null;}}  
\*用较小内存保存Type,Field,Method信息:  
System.RuntimeTypeHandle和System.Type的转换;Type.GetTypeHandle(type),Type.GetTypeFromHandle(typeHandle);  
System.RuntimeMethodHandle和System.Reflection.MethodInfo的转换;  
System.RuntimeFeildHandle和System.Reflection.FieldInfo的转换;  
[](http://blog.photo.sina.com.cn/showpic.html#url=http://album.sina.com.cn/pic/0024gZI6gy6HAkDF9BB67)  
  
  
  
  
4.特性  
特性机制帮助程序员以声明的方式进行编程，而不再需要考虑实现的细节;这样的机制有点类似AOP的编程概念;  
\*什么是特性，如何自定义一个特性？  
特性是一种有别于普通命令式编程的编程方式，通常称为声明式编程方法。所谓声明式编程就是程序员秩序声明某个模块会具有怎么样的特性，而无需关心实现;特 性在被编译时，和传统的命令式代码不同，它会被以二进制数据的方式写入模块文件的元数据中，而在运行时被解读使用。特性也是经常被反射机制应用的元素，因 为它本身是以元数据形式存放的.  
自定义特性:本质就是定义一个继承自System.Attribute类的类型;  
使用需要注意:特性名称用Attribute结尾;为了方便，使用特性时可以省略最后的Attribute;特性类型自身也可以添加其他特性，如[AttributeUsage()].  
\*.net特性可以在那些元素上应用?  
Assembly,Module,Class,Struct,Enum,Constructor,Method,Property,Field,Event,Interface,Parameter,Delegate,ReturnValue,GenericParameter;  
对于类型，结构等元素，特性的使用可以添加在其定义上方，而对于程序集、模块等元素的特性来说，则需要显示地告诉编译器这些特性的作用目标;如:[assembly:MyAttribute];  
可以使用AttributeUsage(AttributeTargets..)来限定特性的使用范围;  
\*获取元素已经申明特性的方法:  
System.Attribute.IsDefined;  
System.Attribute.GetCustomerAttribute()/GetCustomerAttributes;会查找指定特性以及其派生特性，并且会实例化.若声明多次，则报AmbiguousMatchException异常;  
System.Reflection.CustomAttributeData--GetCustomAttributes;该类型使用不会导致特性的实例化，适合安全性要求高的系统;  
注意:读者使用这些方法时，需要注意是否需要实例化特性，因为这意味着元数据中的字节流将被执行，这可能是一个安全隐患。(未理解)  
\*一个元素能否重复声明同一特性?使用AttributeUsageAttribute的AllowMultiple属性,设置为True;  
  
5.名企面试真题  
\*.什么是反射?一种动态分析程序集，模块，类型，方法，字段等目标对象的机制，它依托于元数据;  
\*.在什么情况下使用过委托(答的都不是很准确)?需要由使用者而不是设计者提供回调方法时使用。  
任务的执行者把细节任务进行再分配，执行者确切知道什么工作将要被执行，但却把执行细节委托给其他的组件，方法或程序集。  
\*.请概述事件与委托有什么不同?事件是一种指定格式的委托，要求它没有返回值，参数固定为object-sender,EventArgs-args,它自带add/remove方法，由于在委托链上添加和删除回调方法(事件处理方法);  
\*.你最常用的特性有哪些？首先是特性的特性:[AttributeUsage],[Serializable]等;  
\*.介绍一个你设计过的自定义特性，为什么要使用特性？  
如[target.class]NameAttribute，使用特性有很大的灵活性，比如对工厂设计模式的优化，达到解耦的作用;同时特性是一种声明式的编程方式;  
\*.反射机制的性能如何，你会在什么情况下使用反射?反射是一种动态的分析程序集，模块，类型，方法等目标对象的机制，它的基石是元数据；其实他就是通过调用方法对元数据进行操作，使用它会使程序性能下降。  
当我需要对暂时未知的程序集，类型等目标对象进行操作时，我会使用。因为此时是没用方法名称，字段名称等内容的。在一些特殊情况下，我还会有Reflection.Emit动态的在内存中创建程序集;  
\*.请问动态的发射代码有何作用?可以直接在内存中创建程序集，不用驻留在硬盘;它一般使用在编译器，特殊反射工具，脚本解释器中;  
\*.请用代码描述肯德基排队购买场景;自己做一个简版的;假设一个收银,来人加入队列，买好离开队列;不涉及多线程,算法选择等内容;  
\*.请介绍程序集元数据包含哪些内容?TypeDef,TypeRef&AssemblyRef,Assembly清单;  
  
  
第二部分 .net多线程编程  
1.多线程编程的基本概念;  
\*解释操作系统层面上的进程和线程  
进程:拥有自己的程序块，独占的资源和数据，并且可以被操作系统调用;  
线程:是一个可以被调度的单元，并且维护自己的堆栈和上下文环境;  
简单来说进程代表了一个正在运行的应用程序实体，而进程可以包含一个或多个线程;  
线程和进程最大的区别在于隔离性问题,每个进程被单独地隔离，拥有自己的内存块，独占资源和运行数据，进程间的交互也是相当困难的.而同一进程内的所有线程共享资源和内存块，并且一个线程可以访问，结束同一进程内的其他线程;  
\*多线程程序在OS中是并行执行的么  
线程调度:抢占式和非抢占式，例如Windows--属于同时采用抢占式和非抢占式模式。对于那些优先级高的线程，采用非抢占，对于普通线程，采用抢占模式快速切换;  
在单个CPU的架构上，任何时候只能存在一个运行的线程，OS用过快速的调度轮换让使用者感觉多线程同时执行。而在多CPU架构上，则可能存在并行运行的线程，这取决与线程间是否争用资源;（windows提出一个超线程的概念，就是虚拟CPU，多通道(Intel)？）  
\*什么是迁程?可以视为一个轻量级线程，拥有自己的栈和上下文(寄存器)状态，调度由程序员编码控制;  
在.net运行框架中新建Thread，并不一定保证在OS层面上产生了一个真正的线程;想想（os线程，用户线程）;  
实际上，.net中的线程可能是一个线程，一个迁程甚至一个.net自定义的结构;  
补充:所谓CLR寄宿，指CLR框架运行在某个应用程序上而非字节在操作系统上。常见的有asp.net,sqlserver2005.  
  
2.net中的多线程编程;  
\*如何在.net中手动控制多个线程;创建一个Thread类型对象并不意味着生成一个线程，需要调用Start才生成;  
控制线程的状态:  
  
\*如何使用.net线程池;  
所谓的.net线程池，是指由CLR管理的线程池，而不是指线程池是由.net框架引入的;CLR管理代码负责整理并处理线程的需求，策略可变，投递需求较多时，可能多个线程同时运行处理需求，反之，只创建单线程。  
线程池中运行的线程都是后台线程,IsBackground为true;所谓后台线程指这些线程的运行不会阻碍应用程序的结束;  
System.Threading.ThreadPool:每个进程都拥有一个线程池，.net提供管理机制,用户只需要把线程需求插入到线程池即可;  
static bool QueueUserWorkItem(WaitCallBack--委托类型，接受Object参数，无返回值--callback)  
static bool QueueUserWorkItem(WaitCallBack callback, Object state)  
static bool UnsafeQueueUserWorkItem(WaitCallBack callback, Object state):不会将主线程权限限制传递给辅助线程，可能会提升辅助线程的权限，产生安全漏洞;  
\*如何查看和设置线程池的上下限;一般不需要修改  
ThreadPool.Get/Set Max/Min/Available Threads;  
\*如何定义线程独享的全局数据;  
TLS:本地线程存储;静态变量扮演了全局(appDomain)可见的数据角色，一个static变量同一appDomain的线程均可访问，若希望只有当前线程可对其访问修改的变量，就需要TLS的概念;  
方式一，使用LocalDataStoreSlot：它本身不是线程独显的，但初始化一个该对象意味着在应用程序域内的每个线程上均分配一个数据插槽;  
LocalDataStoreSlot ldss  = thread.AllocateDataSlot();  
Thread.SetData(ldss, Thread.CurrentThread.ManagedThreadId);  
Thread.GetData(ldss);  
方式二，ThreadStaticAttribute使用  
\*如何使用异步模式读取一个文件;  
异步模式:是一种处理流类型时经常用到的模式，读写文件，网络传输，读写数据库，甚至可以异步模式来做任何计算工作。相对于收到编写线程代码，异步模式是一种高效的编程模式;  
指启动一个操作后可以继续执行其他工作，而不必等待操作的结束。  
在.net中，很多类型都支持异步模式编程,以下为4个步骤:  
调用一个形似BeginXXX的方法，表明开始异步执行某操作;  
在调用了BeginXXX方法后，主线程可以继续执行任意的代码，而无需关心异步操作情况;  
以异步聚集技巧来查看异步操作的结果;  
调用EndXXX来表示一个异步操作结果;  
异步模式区别于线程池机制的地方:  
直接调用EndXXX方法，如果异步操作还未执行，主线程会被阻止直到一步操作结束;  
查看调用BeginXXX后得到的IAsyncResult对象IsCompleted属性;  
在调用BeginXXX时传入操作结束后需要执行的方法，同时把执行异步操作的对象传入以便执行EndXXX方法;(未理解)  
尽量使用第三种技巧:主线程负责开始异步读取并且传入狙击时需要的方法和状态对象;  
 using (FileStream fs = File.Create(\_fileName))  
{string content = @"宝山是个SB，哈哈！";  
  byte[] contentByte = Encoding.Default.GetBytes(content);  
  fs.Write(contentByte, 0, contentByte.Length);}  
//开始异步读取文件内容,注意这儿Fs的生命周期是有限的。  
using (FileStream fs = new FileStream(\_fileName, FileMode.Open, FileAccess.Read, FileShare.Read, 1024, FileOptions.Asynchronous))  
                {byte[] data = new byte[1024];  
                    AsyncReadClass arc = new AsyncReadClass(data, fs);  
                    fs.BeginRead(data, 0, 1024, FinishRead, arc);  
//主线程执行一些其他操作  
                    Thread.Sleep(1000 \* 3);  
                    Console.Read();}  
 private static void FinishRead(IAsyncResult ar)  
        {  
            AsyncReadClass arc = ar.AsyncState as AsyncReadClass;  
            //让异步读取占用的资源释放  
            int length = arc.Fs.EndRead(ar);//注意对象的生命周期}[](http://blog.photo.sina.com.cn/showpic.html#url=http://album.sina.com.cn/pic/0024gZI6gy6HAkwBMTy0c)  
  
\*如何阻止线程执行上下文的传递;  
同一进程中线程虽然共享资源和内存块，但仍然拥有自己的上线问，在.net中，线程的上下文有流动的特性;  
线程执行上下文的内容:安全上下文，调用上下文，同步上下文，本地化上下文，事务上下文,CLR宿主上下文;  
上下文的流动:Thread thread;thread.Start();thread.join()//阻塞当前线程;  
如何阻止上下文的流动:线程执行的上下文是所有线程的一个包装，在通常情况下，当前线程的执行上下文会流动到新建线程之中。程序员可以使用定义在 System.Threading.ThreadPool类型的UnsafeQueueUserWorkItem方法和定义在 ExecutionContext类型中的SuppressFlow方法来阻止这种流动。注意这样虽然可以提高效率，但会降低安全性;  
  
3.多线程程序的线程同步  
\*什么是同步快和同步快索引;  
[](http://blog.photo.sina.com.cn/showpic.html#url=http://album.sina.com.cn/pic/0024gZI6gy6HAyJlkm00c)  
.net团队在设计基本框架时已经考虑了线程同步的问题，采用了折中的方式:为每个对内存对象分配一个索引，该索引中只存在一个表明数组内索引的整数。 在.net加载时会新建一个同步块数组，每当某个对象需要被同步时，.net会为其分配一个同步块，并且把该同步块在同步数组中的索引加入对象的同步块索 引中;  
当一个线程试图使用该对象进入同步时，会检查该对象的同步索引，如果索引为负数则会在同步块数组中寻找或者新建一个同步块，并且把同步块索引值放入该对象 的同步块索引中，如果不为负值，则找到该同步块，并且检查是否被其他线程使用，如果有进入等待状态，如果没用则申明使用该同步块.  
进入和退出同步:System.Threading.Monitor.Enter/Exit;  
\*C#中的lock关键字有何作用;  
lock等价于Monitor.Entry/Exit; 在通常情况下，lock一个私用引用成员变量来完成成员方法的线程同步，使用一个私有静态引用变量来完成静态方法的线程同步;  
\*是否可以使用值类型对象来实现线程同步？不能，会出现严重错误,Monitor相关方法使用时会出现拆装箱，每一次堆内的对象均会改变，出现严重错误。所以应该使用lock，而不要使用Monitor.Entrr/Exit；  
\*可否对引用类型自身进行同步;可以，但这样的程序缺乏健壮性（lock(this),lock(Typeof(...))），当某个类型使用者恶意地长期占用对象的同步块时，所有的其他对象会死锁;  
\*什么是互斥体,Mutex类型，Monitor类型的功能有何区别;（WaitHandle(所有封装的内核同步对象的的抽象基类)，类似的还有Semaphone, EventWaitHandler);  
Mutex.WaitOne();Mutex.Close();Mutex.ReleaseMutex();  
Mutex使用OS内核对象，Monitor在.net框架实现，mutex效率低下(10倍,用户态->系统态);  
Monitor只能同步一个AppDomian中的线程，而Mutex可以跨越Process;  
4.名企真题;  
\*进程与线程如何理解?操作系统中的进程拥有自己独立的内存空间(包含数据块，程序块)，如Win32(分配4G的虚拟内存空间)，进程可以被操作系统调 度,简单来说，一个进程代表了一个正在运行的应用程序实体，可以包含一个或多个线程;线程是一个可以被调度的单元，维护自己独立的堆栈和上下文环境;  
进程与线程的最大区别是隔离性，每个进程独立的运行，拥有自己的内存块，独占资源，相互间的交互困难；而一个进程中的多个线程可以共享数据和内存块，一个线程可以访问，结束同一进程中另外的线程;  
\*根据线程安全的相关知识，分析当调用test时，i>10是否会引起死锁:  
public void test(int i){  
lock(this){  
if(i>10){  
i--;  
test(i);  
}}}//首先不建议使用this之中lock方式;//不会死锁，因为传的是值类型;（未OK）  
\*后台线程与一般线程有何区别:  
前台线程能阻止应用程序的终结，一直到所有的前台线程终止后，CLR才能关闭应用程序。后台线程又被称为守护线程，它被CLR认为是程序执行中可以做出牺牲的途径，即任何时候都可以被忽略，因此，如果所有的前台线程终止，应用程序卸载时，所有的后台线程也会被自动终止.  
\*一共有几种方法在多线程间共享数据?（这儿的共享其实就是指的如何同步）lock,mutex;  
\*使用lock和mutex的区别:效率上lock高很多,但mutex因为是os内核对象所以可以实现多进程间的同步;  
\*是否可以对值类型使用lock？不能，CLR的机制是在所有的堆对象中分配一个同步块索引，值类型没有;  
\*你会在什么时候考虑使用多线程?比如考虑到相应速度，不希望因为资源的阻塞而影响用户的使用;  
\*使用Thread类型新建的线程是否来自线程池?必须不是，ThreadPool.QueueUseWorkItem();  
  
第三部分 .net单元测试  
1.单元测试的基本概念;  
\*简述单元测试概念和优点;  
\*举例说明TDD开发方式流程;  
\*编程阶乘功能模块测试用例;  
  
2.使用NUnit进行单元测试;  
\*如何使用NUNIT进行单元测试;  
\*如何对测试用例进行分类;  
\*解释SetUp，TearDown;  
  
3.名企面试真题; 高级特性，多线程编程，单元测试;  
  
第一部分 .net高级特性  
1.委托:提供安全的函数回调机制;  
\*基本原理:与C++中的函数指针相似;安全--它和其他所有.net成员一样是一种类型，任何委托都是System.Delegate的某个派生类对象;  
System.Object->SystemDelegate(ISerializable,ICloneable)->System.MultiCastDelegate  
->Delegate T;  
public delegate void TestDelegate(int i)定义一个委托，内部包含invoke方法(由编译器自动完成);委托的调用其实就是执行类定义委托时生成的Invoke方法;  
总结:每个委托至少包含一个指向回调方法的指针，该方法可以是实例方法，也可以是静态方法。委托实现回调方法的机制，方便使用;  
\*委托回调静态方法和实例方法的区别;  
当委托绑定静态方法时，target为null;绑定实例方法是，target指向该实例方法所属类型的一个实例对象;  
\*什么是链式委托:指一个由委托组成的链表，而不是指另一个特殊的委托;prev，当一个委托被调用时，所有链表上的该委托的后续委托均被顺序执行;  
\*链式委托的执行顺序:按照委托链上的顺序从当前委托开始依次向后执行。如果有需要，可以通过GetInvocationList()来获得委托链上所有需要执行的委托，并且按照任何希望的顺序去执行;  
\*可否定义有返回值的方法的委托链:委托可以是带有返回值的方法，但多于一个待返回值的方法被添加到委托链是，程序员需要手动调用每一个委托链上的方法;  
\*委托的应用场合:任务的执行者把细节工作进行再分配，执行者确切地知道什么工作将要被执行，但却把执行细节委托给其他组件，方法或程序集;一些简单的重点记录:(给个程序集中有多个模块的例子)  
  
2.事件(Event)  
定义:是一种使对象或类能够提供通知的成员，客户端可以通过提供事件处理程序为相应的事件添加可执行代码;  
event标记\_\_委托类型(如EventHandler)\_\_声明的事件对象;  
使用事件的步骤:  
如果需要，定义一个派生自System.EventArgs的参数类型;  ConsoleEventArgs  
在事件的管理类型中定义事件的私有成员;                ConsoleManagsder.ConsoleEvent  
通知事件订阅者;                                     ConsoleManager.SendConsoleEvent  
事件使用客户端订阅/取消订阅事件;                     Log  
补充StreamWriter,StreamReader,各种Stream:  
using(FileStream fs = File.Create(...)){}  
using(StreamWriter sw =info.AppendText()){}  
\*事件和委托有何联系?  
实际上，事件就是就是一个委托类型，当程序员定义一个事件的时候，实际上是定义了一个特殊的委托成员。它没有返回值，有固定的两个参数 sender,EventArgs;而事件的使用者订阅事件时，本质就是把事件处理方法加入到委托链表中;声明event时默认会添加一对针对特定委托的 add/remove方法;  
\*如何设计一个带很多事件的类型？使用EventHandlerList(System.ComponentModel);  
注意:考虑线程同步措施;每个事件定义一套成员--事件的委托原型，事件的订阅和取消订阅方法，定义事件的专用参数类型;减少多事件类型的大小，但代码量增加;  
\*用代码表示如下情形:猫叫，老鼠逃跑,主人惊醒<==>猫叫被老鼠和主人订阅;  
  
3.反射  
\*反射的原理和实现它的基石:反射是一种动态分析程序集，模块，类型，字段等目标对象的机制，它的实现依托于元数据(基石)。元数据是存储在PE文件中的数据块，他详细记录了程序集或模块内部的结构，引用类型，程序集清单.TypeDef,TypeRef&AssemblyRef,Assembly；  
\*.net提供哪些类型来实现反 射:Assembly,AssemblyName,EventInfo,LocalVariableInfo,ManifestResourceInfo,MemberInfo,MethodBase,MethodBody,Module,ParameterInfo,PropertyInfo;  
\*如何动态的发射程序集:动态的生成一个程序集。而不仅仅是分析程序经济或者生成程序集对象;在System.Reflection.Emit命名空间 下，定义了一系列用于动态发射程序集，模块，类型，方法等元素的中间代码。这些类型的主要使用者是编译器，特殊反射工具或者脚本解释器.  
AssemblyBuilder,ConstructorBuilder,CustomerAttributeBuilder,EnumBuilder,EventBuilder,FieldBuilder,GenericTypeParameterBuilder,ILGenerator,LocalBuilder,MethodBuilder,OpCodes,ParameterBuilder,PropertyBuilder,TypeBuilder;  
//步骤：  
在当前appDomain中定义新程序集: AppDomain myDomain = Thread.GetDomain();  
定义模块:                     assemblyBuilder.DefineDynamicModule();  
定义模块中的类型:              Typebuilder addClass=assemblyModule.DefineType();  
定义公共构造方法及其参数       Type[] ctorParams = new Type[]{typeof(long),typeof(long)};  
            构造方法中间代码   ctorIL.Emit(OpCodes.Ldarg\_0);  
动态发送程序集:  
object  ptInstance=Activator.CreateInstance(type, ctorParams);  
MethodInfo info = type.GetMethod("GetResult", new Type[0]);//通过方法名和返回值类型得到方法;  
OpCodes类型包含大多数需要使用的中间代码指令;  
注意:新的程序集发射是直接把中间代码在内存中生成一种机制，而不是在物理硬盘上生成代码的机制;  
\*利用反射来实现工程模式,动手;  
'  
    //针对每一个product做相应的Attribute,针对product系列做Attribute，还是需要参考  
    [AttributeUsage(AttributeTargets.Class)]  
    public class ProductAttribute : Attribute  
    {RoomPart \_myRP = new RoomPart();  
        public ProductAttribute(RoomPart rp)  
        {\_myRP = rp;  }  
        public RoomPart MyRoomPart  
        {get{ return \_myRP;}  
        } }  
  
    [AttributeUsage(AttributeTargets.Interface)]  
    public class ProductListAttribute : Attribute  
    {Type[] \_mylist;  
        public ProductListAttribute(Type[] productList)  
        {\_mylist = productList;}  
        public Type[] MyProductList  
        {get { return \_mylist; } }  
    }  
public IProduct  Produce(RoomPart rp)  
{//通过反射，从IProduct接口获得属性，从而获得所有的产品零件列表  
    ProductListAttribute pla = (ProductListAttribute)Attribute.GetCustomAttribute(typeof(IProduct), typeof(ProductListAttribute));  
    //遍历所有的实现产品零件类型  
    foreach (Type type in pla.MyProductList) {  
        ProductAttribute pa = (ProductAttribute)Attribute.GetCustomAttribute(type, typeof(ProductAttribute));  
        if (rp == pa.MyRoomPart) {  
           object pro =  Assembly.GetExecutingAssembly().CreateInstance(type.FullName);  
           return pro as IProduct;  
        }  
        return null;}}  
\*用较小内存保存Type,Field,Method信息:  
System.RuntimeTypeHandle和System.Type的转换;Type.GetTypeHandle(type),Type.GetTypeFromHandle(typeHandle);  
System.RuntimeMethodHandle和System.Reflection.MethodInfo的转换;  
System.RuntimeFeildHandle和System.Reflection.FieldInfo的转换;  
[](http://blog.photo.sina.com.cn/showpic.html#url=http://album.sina.com.cn/pic/0024gZI6gy6HAkDF9BB67)  
  
  
  
  
4.特性  
特性机制帮助程序员以声明的方式进行编程，而不再需要考虑实现的细节;这样的机制有点类似AOP的编程概念;  
\*什么是特性，如何自定义一个特性？  
特性是一种有别于普通命令式编程的编程方式，通常称为声明式编程方法。所谓声明式编程就是程序员秩序声明某个模块会具有怎么样的特性，而无需关心实现;特 性在被编译时，和传统的命令式代码不同，它会被以二进制数据的方式写入模块文件的元数据中，而在运行时被解读使用。特性也是经常被反射机制应用的元素，因 为它本身是以元数据形式存放的.  
自定义特性:本质就是定义一个继承自System.Attribute类的类型;  
使用需要注意:特性名称用Attribute结尾;为了方便，使用特性时可以省略最后的Attribute;特性类型自身也可以添加其他特性，如[AttributeUsage()].  
\*.net特性可以在那些元素上应用?  
Assembly,Module,Class,Struct,Enum,Constructor,Method,Property,Field,Event,Interface,Parameter,Delegate,ReturnValue,GenericParameter;  
对于类型，结构等元素，特性的使用可以添加在其定义上方，而对于程序集、模块等元素的特性来说，则需要显示地告诉编译器这些特性的作用目标;如:[assembly:MyAttribute];  
可以使用AttributeUsage(AttributeTargets..)来限定特性的使用范围;  
\*获取元素已经申明特性的方法:  
System.Attribute.IsDefined;  
System.Attribute.GetCustomerAttribute()/GetCustomerAttributes;会查找指定特性以及其派生特性，并且会实例化.若声明多次，则报AmbiguousMatchException异常;  
System.Reflection.CustomAttributeData--GetCustomAttributes;该类型使用不会导致特性的实例化，适合安全性要求高的系统;  
注意:读者使用这些方法时，需要注意是否需要实例化特性，因为这意味着元数据中的字节流将被执行，这可能是一个安全隐患。(未理解)  
\*一个元素能否重复声明同一特性?使用AttributeUsageAttribute的AllowMultiple属性,设置为True;  
  
5.名企面试真题  
\*.什么是反射?一种动态分析程序集，模块，类型，方法，字段等目标对象的机制，它依托于元数据;  
\*.在什么情况下使用过委托(答的都不是很准确)?需要由使用者而不是设计者提供回调方法时使用。  
任务的执行者把细节任务进行再分配，执行者确切知道什么工作将要被执行，但却把执行细节委托给其他的组件，方法或程序集。  
\*.请概述事件与委托有什么不同?事件是一种指定格式的委托，要求它没有返回值，参数固定为object-sender,EventArgs-args,它自带add/remove方法，由于在委托链上添加和删除回调方法(事件处理方法);  
\*.你最常用的特性有哪些？首先是特性的特性:[AttributeUsage],[Serializable]等;  
\*.介绍一个你设计过的自定义特性，为什么要使用特性？  
如[target.class]NameAttribute，使用特性有很大的灵活性，比如对工厂设计模式的优化，达到解耦的作用;同时特性是一种声明式的编程方式;  
\*.反射机制的性能如何，你会在什么情况下使用反射?反射是一种动态的分析程序集，模块，类型，方法等目标对象的机制，它的基石是元数据；其实他就是通过调用方法对元数据进行操作，使用它会使程序性能下降。  
当我需要对暂时未知的程序集，类型等目标对象进行操作时，我会使用。因为此时是没用方法名称，字段名称等内容的。在一些特殊情况下，我还会有Reflection.Emit动态的在内存中创建程序集;  
\*.请问动态的发射代码有何作用?可以直接在内存中创建程序集，不用驻留在硬盘;它一般使用在编译器，特殊反射工具，脚本解释器中;  
\*.请用代码描述肯德基排队购买场景;自己做一个简版的;假设一个收银,来人加入队列，买好离开队列;不涉及多线程,算法选择等内容;  
\*.请介绍程序集元数据包含哪些内容?TypeDef,TypeRef&AssemblyRef,Assembly清单;  
  
  
第二部分 .net多线程编程  
1.多线程编程的基本概念;  
\*解释操作系统层面上的进程和线程  
进程:拥有自己的程序块，独占的资源和数据，并且可以被操作系统调用;  
线程:是一个可以被调度的单元，并且维护自己的堆栈和上下文环境;  
简单来说进程代表了一个正在运行的应用程序实体，而进程可以包含一个或多个线程;  
线程和进程最大的区别在于隔离性问题,每个进程被单独地隔离，拥有自己的内存块，独占资源和运行数据，进程间的交互也是相当困难的.而同一进程内的所有线程共享资源和内存块，并且一个线程可以访问，结束同一进程内的其他线程;  
\*多线程程序在OS中是并行执行的么  
线程调度:抢占式和非抢占式，例如Windows--属于同时采用抢占式和非抢占式模式。对于那些优先级高的线程，采用非抢占，对于普通线程，采用抢占模式快速切换;  
在单个CPU的架构上，任何时候只能存在一个运行的线程，OS用过快速的调度轮换让使用者感觉多线程同时执行。而在多CPU架构上，则可能存在并行运行的线程，这取决与线程间是否争用资源;（windows提出一个超线程的概念，就是虚拟CPU，多通道(Intel)？）  
\*什么是迁程?可以视为一个轻量级线程，拥有自己的栈和上下文(寄存器)状态，调度由程序员编码控制;  
在.net运行框架中新建Thread，并不一定保证在OS层面上产生了一个真正的线程;想想（os线程，用户线程）;  
实际上，.net中的线程可能是一个线程，一个迁程甚至一个.net自定义的结构;  
补充:所谓CLR寄宿，指CLR框架运行在某个应用程序上而非字节在操作系统上。常见的有asp.net,sqlserver2005.  
  
2.net中的多线程编程;  
\*如何在.net中手动控制多个线程;创建一个Thread类型对象并不意味着生成一个线程，需要调用Start才生成;  
控制线程的状态:  
  
\*如何使用.net线程池;  
所谓的.net线程池，是指由CLR管理的线程池，而不是指线程池是由.net框架引入的;CLR管理代码负责整理并处理线程的需求，策略可变，投递需求较多时，可能多个线程同时运行处理需求，反之，只创建单线程。  
线程池中运行的线程都是后台线程,IsBackground为true;所谓后台线程指这些线程的运行不会阻碍应用程序的结束;  
System.Threading.ThreadPool:每个进程都拥有一个线程池，.net提供管理机制,用户只需要把线程需求插入到线程池即可;  
static bool QueueUserWorkItem(WaitCallBack--委托类型，接受Object参数，无返回值--callback)  
static bool QueueUserWorkItem(WaitCallBack callback, Object state)  
static bool UnsafeQueueUserWorkItem(WaitCallBack callback, Object state):不会将主线程权限限制传递给辅助线程，可能会提升辅助线程的权限，产生安全漏洞;  
\*如何查看和设置线程池的上下限;一般不需要修改  
ThreadPool.Get/Set Max/Min/Available Threads;  
\*如何定义线程独享的全局数据;  
TLS:本地线程存储;静态变量扮演了全局(appDomain)可见的数据角色，一个static变量同一appDomain的线程均可访问，若希望只有当前线程可对其访问修改的变量，就需要TLS的概念;  
方式一，使用LocalDataStoreSlot：它本身不是线程独显的，但初始化一个该对象意味着在应用程序域内的每个线程上均分配一个数据插槽;  
LocalDataStoreSlot ldss  = thread.AllocateDataSlot();  
Thread.SetData(ldss, Thread.CurrentThread.ManagedThreadId);  
Thread.GetData(ldss);  
方式二，ThreadStaticAttribute使用  
\*如何使用异步模式读取一个文件;  
异步模式:是一种处理流类型时经常用到的模式，读写文件，网络传输，读写数据库，甚至可以异步模式来做任何计算工作。相对于收到编写线程代码，异步模式是一种高效的编程模式;  
指启动一个操作后可以继续执行其他工作，而不必等待操作的结束。  
在.net中，很多类型都支持异步模式编程,以下为4个步骤:  
调用一个形似BeginXXX的方法，表明开始异步执行某操作;  
在调用了BeginXXX方法后，主线程可以继续执行任意的代码，而无需关心异步操作情况;  
以异步聚集技巧来查看异步操作的结果;  
调用EndXXX来表示一个异步操作结果;  
异步模式区别于线程池机制的地方:  
直接调用EndXXX方法，如果异步操作还未执行，主线程会被阻止直到一步操作结束;  
查看调用BeginXXX后得到的IAsyncResult对象IsCompleted属性;  
在调用BeginXXX时传入操作结束后需要执行的方法，同时把执行异步操作的对象传入以便执行EndXXX方法;(未理解)  
尽量使用第三种技巧:主线程负责开始异步读取并且传入狙击时需要的方法和状态对象;  
 using (FileStream fs = File.Create(\_fileName))  
{string content = @"宝山是个SB，哈哈！";  
  byte[] contentByte = Encoding.Default.GetBytes(content);  
  fs.Write(contentByte, 0, contentByte.Length);}  
//开始异步读取文件内容,注意这儿Fs的生命周期是有限的。  
using (FileStream fs = new FileStream(\_fileName, FileMode.Open, FileAccess.Read, FileShare.Read, 1024, FileOptions.Asynchronous))  
                {byte[] data = new byte[1024];  
                    AsyncReadClass arc = new AsyncReadClass(data, fs);  
                    fs.BeginRead(data, 0, 1024, FinishRead, arc);  
//主线程执行一些其他操作  
                    Thread.Sleep(1000 \* 3);  
                    Console.Read();}  
 private static void FinishRead(IAsyncResult ar)  
        {  
            AsyncReadClass arc = ar.AsyncState as AsyncReadClass;  
            //让异步读取占用的资源释放  
            int length = arc.Fs.EndRead(ar);//注意对象的生命周期}[](http://blog.photo.sina.com.cn/showpic.html#url=http://album.sina.com.cn/pic/0024gZI6gy6HAkwBMTy0c)  
  
\*如何阻止线程执行上下文的传递;  
同一进程中线程虽然共享资源和内存块，但仍然拥有自己的上线问，在.net中，线程的上下文有流动的特性;  
线程执行上下文的内容:安全上下文，调用上下文，同步上下文，本地化上下文，事务上下文,CLR宿主上下文;  
上下文的流动:Thread thread;thread.Start();thread.join()//阻塞当前线程;  
如何阻止上下文的流动:线程执行的上下文是所有线程的一个包装，在通常情况下，当前线程的执行上下文会流动到新建线程之中。程序员可以使用定义在 System.Threading.ThreadPool类型的UnsafeQueueUserWorkItem方法和定义在 ExecutionContext类型中的SuppressFlow方法来阻止这种流动。注意这样虽然可以提高效率，但会降低安全性;  
  
3.多线程程序的线程同步  
\*什么是同步快和同步快索引;  
[](http://blog.photo.sina.com.cn/showpic.html#url=http://album.sina.com.cn/pic/0024gZI6gy6HAyJlkm00c)  
.net团队在设计基本框架时已经考虑了线程同步的问题，采用了折中的方式:为每个对内存对象分配一个索引，该索引中只存在一个表明数组内索引的整数。 在.net加载时会新建一个同步块数组，每当某个对象需要被同步时，.net会为其分配一个同步块，并且把该同步块在同步数组中的索引加入对象的同步块索 引中;  
当一个线程试图使用该对象进入同步时，会检查该对象的同步索引，如果索引为负数则会在同步块数组中寻找或者新建一个同步块，并且把同步块索引值放入该对象 的同步块索引中，如果不为负值，则找到该同步块，并且检查是否被其他线程使用，如果有进入等待状态，如果没用则申明使用该同步块.  
进入和退出同步:System.Threading.Monitor.Enter/Exit;  
\*C#中的lock关键字有何作用;  
lock等价于Monitor.Entry/Exit; 在通常情况下，lock一个私用引用成员变量来完成成员方法的线程同步，使用一个私有静态引用变量来完成静态方法的线程同步;  
\*是否可以使用值类型对象来实现线程同步？不能，会出现严重错误,Monitor相关方法使用时会出现拆装箱，每一次堆内的对象均会改变，出现严重错误。所以应该使用lock，而不要使用Monitor.Entrr/Exit；  
\*可否对引用类型自身进行同步;可以，但这样的程序缺乏健壮性（lock(this),lock(Typeof(...))），当某个类型使用者恶意地长期占用对象的同步块时，所有的其他对象会死锁;  
\*什么是互斥体,Mutex类型，Monitor类型的功能有何区别;（WaitHandle(所有封装的内核同步对象的的抽象基类)，类似的还有Semaphone, EventWaitHandler);  
Mutex.WaitOne();Mutex.Close();Mutex.ReleaseMutex();  
Mutex使用OS内核对象，Monitor在.net框架实现，mutex效率低下(10倍,用户态->系统态);  
Monitor只能同步一个AppDomian中的线程，而Mutex可以跨越Process;  
4.名企真题;  
\*进程与线程如何理解?操作系统中的进程拥有自己独立的内存空间(包含数据块，程序块)，如Win32(分配4G的虚拟内存空间)，进程可以被操作系统调 度,简单来说，一个进程代表了一个正在运行的应用程序实体，可以包含一个或多个线程;线程是一个可以被调度的单元，维护自己独立的堆栈和上下文环境;  
进程与线程的最大区别是隔离性，每个进程独立的运行，拥有自己的内存块，独占资源，相互间的交互困难；而一个进程中的多个线程可以共享数据和内存块，一个线程可以访问，结束同一进程中另外的线程;  
\*根据线程安全的相关知识，分析当调用test时，i>10是否会引起死锁:  
public void test(int i){  
lock(this){  
if(i>10){  
i--;  
test(i);  
}}}//首先不建议使用this之中lock方式;//不会死锁，因为传的是值类型;（未OK）  
\*后台线程与一般线程有何区别:  
前台线程能阻止应用程序的终结，一直到所有的前台线程终止后，CLR才能关闭应用程序。后台线程又被称为守护线程，它被CLR认为是程序执行中可以做出牺牲的途径，即任何时候都可以被忽略，因此，如果所有的前台线程终止，应用程序卸载时，所有的后台线程也会被自动终止.  
\*一共有几种方法在多线程间共享数据?（这儿的共享其实就是指的如何同步）lock,mutex;  
\*使用lock和mutex的区别:效率上lock高很多,但mutex因为是os内核对象所以可以实现多进程间的同步;  
\*是否可以对值类型使用lock？不能，CLR的机制是在所有的堆对象中分配一个同步块索引，值类型没有;  
\*你会在什么时候考虑使用多线程?比如考虑到相应速度，不希望因为资源的阻塞而影响用户的使用;  
\*使用Thread类型新建的线程是否来自线程池?必须不是，ThreadPool.QueueUseWorkItem();  
  
第三部分 .net单元测试  
1.单元测试的基本概念;  
\*简述单元测试概念和优点;  
\*举例说明TDD开发方式流程;  
\*编程阶乘功能模块测试用例;  
  
2.使用NUnit进行单元测试;  
\*如何使用NUNIT进行单元测试;  
\*如何对测试用例进行分类;  
\*解释SetUp，TearDown;  
  
3.名企面试真题;