[T]深刻理解C#中资源释放

今天我的一个朋友看到我写的那篇《C#中用AJAX验证用户登录》时，给我指出了点小毛病。就是在用户登录时，如果用户登录失败，在下面这段代码中，都会new出来一个User对象，如果连续登录失败多次，就会生成多个User对象，而它们在登录失败后已经无用了，依然占据着内存，就算是C#有垃圾回收机制，但不确定什么时候对这些对象进行回收。  
然后去网上找了一篇C#资源释放的文章，讲的很透彻，和大家分享一下。http://www.cnblogs.com/Images/OutliningIndicators/ExpandedBlockStart.gif  
http://www.cnblogs.com/Images/OutliningIndicators/None.gifusing System;  
http://www.cnblogs.com/Images/OutliningIndicators/None.gifusing System.Collections.Generic;  
http://www.cnblogs.com/Images/OutliningIndicators/None.gifusing System.Linq;  
http://www.cnblogs.com/Images/OutliningIndicators/None.gifusing System.Web;  
http://www.cnblogs.com/Images/OutliningIndicators/None.gifusing System.Web.UI;  
http://www.cnblogs.com/Images/OutliningIndicators/None.gifusing System.Web.UI.WebControls;  
http://www.cnblogs.com/Images/OutliningIndicators/None.gifusing Model;  
http://www.cnblogs.com/Images/OutliningIndicators/None.gifusing BLL;  
http://www.cnblogs.com/Images/OutliningIndicators/None.gifpublic partial class AJAX\_CheckUser : System.Web.UI.Page  
http://www.cnblogs.com/Images/OutliningIndicators/ExpandedBlockStart.gif{  
http://www.cnblogs.com/Images/OutliningIndicators/InBlock.gif    protected void Page\_Load(object sender, EventArgs e)  
http://www.cnblogs.com/Images/OutliningIndicators/ExpandedSubBlockStart.gif    {  
http://www.cnblogs.com/Images/OutliningIndicators/InBlock.gif        string username = Request.Params["\_name"];  
http://www.cnblogs.com/Images/OutliningIndicators/InBlock.gif        string password = Request.Params["\_pwd"];  
http://www.cnblogs.com/Images/OutliningIndicators/InBlock.gif        User user = new User();  
http://www.cnblogs.com/Images/OutliningIndicators/InBlock.gif        user.Username = username;  
http://www.cnblogs.com/Images/OutliningIndicators/InBlock.gif        user.Password = password;  
http://www.cnblogs.com/Images/OutliningIndicators/InBlock.gif        string check = BLLCheckUser.Check(user);  
http://www.cnblogs.com/Images/OutliningIndicators/InBlock.gif        Response.Write(check);  
http://www.cnblogs.com/Images/OutliningIndicators/ExpandedSubBlockEnd.gif    }  
http://www.cnblogs.com/Images/OutliningIndicators/ExpandedBlockEnd.gif}

      首先，我们需要明确2个概念。

第一个就是很多人用.Net写程序，会谈到托管这个概念。那么.Net所指的资源托管到底是什么意思，是相对于所有资源，还是只限于某一方面资源？很多人对此不是很了解，**其实.Net所指的托管只是针对内存这一个方面，并不是对于所有的资源；因此对于Stream，数据库的连接，GDI+的相关对象，还有Com对象等等，这些资源并不是受到.Net管理而统称为非托管资源。而对于内存的释放和回收，系统提供了GC-Garbage Collector，而至于其他资源则需要手动进行释放。**  
   
那么第二个概念就是什么是垃圾，通过我以前的文章，会了解到.Net类型分为两大类，一个就是值类型，另一个就是引用类型。前者是分配在栈上，并不需要GC回收；后者是分配在堆上，因此它的内存释放和回收需要通过GC来完成。GC的全称为“Garbage Collector”,顾名思义就是垃圾回收器，那么只有被称为垃圾的对象才能被GC回收。也就是说，**一个引用类型对象所占用的内存需要被GC回收，需要先成为垃圾。那么.Net如何判定一个引用类型对象是垃圾呢，.Net的判断很简单，只要判定此对象或者其包含的子对象没有任何引用是有效的，那么系统就认为它是垃圾。**  
   
明确了这两个基本概念，接下来说说GC的运作方式以及其的功能。内存的释放和回收需要伴随着程序的运行，因此系统为GC安排了独立的线程。那么GC的工作大致是，查询内存中对象是否成为垃圾，然后对垃圾进行释放和回收。那么对于GC对于内存回收采取了一定的优先算法进行轮循回收内存资源。其次，**对于内存中的垃圾分为两种，一种是需要调用对象的析构函数，另一种是不需要调用的。GC对于前者的回收需要通过两步完成，第一步是调用对象的析构函数，第二步是回收内存，但是要注意这两步不是在GC一次轮循完成，即需要两次轮循；相对于后者，则只是回收内存而已。**  
   
很明显得知，对于某个具体的资源，无法确切知道，对象析构函数什么时候被调用，以及GC什么时候会去释放和回收它所占用的内存。那么对于从C、C++之类语言转换过来的程序员来说，这里需要转变观念。  
   
那么对于程序资源来说，我们应该做些什么，以及如何去做，才能使程序效率最高，同时占用资源能尽快的释放。前面也说了，资源分为两种，托管的内存资源，这是不需要我们操心的，系统已经为我们进行管理了；那么对于非托管的资源，这里再重申一下，就是Stream，数据库的连接，GDI+的相关对象，还有Com对象等等这些资源，需要我们手动去释放。  
   
如何去释放，应该把这些操作放到哪里比较好呢。.Net提供了三种方法，也是最常见的三种，大致如下：  
<!--[if !supportLists]-->1． <!--[endif]-->析构函数；  
<!--[if !supportLists]-->2． <!--[endif]-->继承IDisposable接口，实现Dispose方法；  
<!--[if !supportLists]-->3． <!--[endif]-->提供Close方法。  
   
**经过前面的介绍，可以知道析构函数只能被GC来调用的，那么无法确定它什么时候被调用，因此用它作为资源的释放并不是很合理，因为资源释放不及时；但是为了防止资源泄漏，毕竟它会被GC调用，因此析构函数可以作为一个补救方法。而Close与Dispose这两种方法的区别在于，调用完了对象的Close方法后，此对象有可能被重新进行使用；而Dispose方法来说，此对象所占有的资源需要被标记为无用了，也就是此对象被销毁了，不能再被使用。**例如，常见SqlConnection这个类，当调用完Close方法后，可以通过Open重新打开数据库连接，当彻底不用这个对象了就可以调用Dispose方法来标记此对象无用，等待GC回收。明白了这两种方法的意思后，大家在往自己的类中添加的接口时候，不要歪曲了这两者意思。

接下来说说这三个函数的调用时机，我用几个试验结果来进行说明，可能会使大家的印象更深。  
首先是这三种方法的实现，大致如下：

http://www.cnblogs.com/Images/OutliningIndicators/ExpandedBlockStart.gif  
http://www.cnblogs.com/Images/OutliningIndicators/ExpandedBlockStart.gif///<summary>  
http://www.cnblogs.com/Images/OutliningIndicators/InBlock.gif    /// The class to show three disposal function  
http://www.cnblogs.com/Images/OutliningIndicators/ExpandedBlockEnd.gif    ///</summary>   
http://www.cnblogs.com/Images/OutliningIndicators/None.gif    public class DisposeClass:IDisposable  
http://www.cnblogs.com/Images/OutliningIndicators/ExpandedBlockStart.gif    {  
http://www.cnblogs.com/Images/OutliningIndicators/InBlock.gif        public void Close()  
http://www.cnblogs.com/Images/OutliningIndicators/ExpandedSubBlockStart.gif        {  
http://www.cnblogs.com/Images/OutliningIndicators/InBlock.gif            Debug.WriteLine( "Close called!" );  
http://www.cnblogs.com/Images/OutliningIndicators/ExpandedSubBlockEnd.gif        }  
http://www.cnblogs.com/Images/OutliningIndicators/InBlock.gif   
http://www.cnblogs.com/Images/OutliningIndicators/InBlock.gif        ~DisposeClass()  
http://www.cnblogs.com/Images/OutliningIndicators/ExpandedSubBlockStart.gif        {  
http://www.cnblogs.com/Images/OutliningIndicators/InBlock.gif            Debug.WriteLine( "Destructor called!" );  
http://www.cnblogs.com/Images/OutliningIndicators/ExpandedSubBlockEnd.gif        }  
http://www.cnblogs.com/Images/OutliningIndicators/InBlock.gif   
http://www.cnblogs.com/Images/OutliningIndicators/ExpandedSubBlockStart.gif        #region IDisposable Members  
http://www.cnblogs.com/Images/OutliningIndicators/InBlock.gif   
http://www.cnblogs.com/Images/OutliningIndicators/InBlock.gif        public void Dispose()  
http://www.cnblogs.com/Images/OutliningIndicators/ExpandedSubBlockStart.gif        {  
http://www.cnblogs.com/Images/OutliningIndicators/InBlock.gif            // TODO: Add DisposeClass.Dispose implementation  
http://www.cnblogs.com/Images/OutliningIndicators/InBlock.gif            Debug.WriteLin  
http://www.cnblogs.com/Images/OutliningIndicators/InBlock.gife( "Dispose called!" );  
http://www.cnblogs.com/Images/OutliningIndicators/ExpandedSubBlockEnd.gif        }  
http://www.cnblogs.com/Images/OutliningIndicators/InBlock.gif   
http://www.cnblogs.com/Images/OutliningIndicators/ExpandedSubBlockEnd.gif        #endregion  
http://www.cnblogs.com/Images/OutliningIndicators/ExpandedBlockEnd.gif    }  
http://www.cnblogs.com/Images/OutliningIndicators/None.gif

      对于Close来说不属于真正意义上的释放，除了注意它需要显示被调用外，我在此对它不多说了。而对于析构函数而言，不是在对象离开作用域后立刻被执行，只有在关闭进程或者调用GC.Collect方法的时候才被调用，参看如下的代码运行结果。

http://www.cnblogs.com/Images/OutliningIndicators/ExpandedBlockStart.gif  
http://www.cnblogs.com/Images/OutliningIndicators/None.gif        private void Create()  
http://www.cnblogs.com/Images/OutliningIndicators/ExpandedBlockStart.gif        {  
http://www.cnblogs.com/Images/OutliningIndicators/InBlock.gif            DisposeClass myClass = new DisposeClass();  
http://www.cnblogs.com/Images/OutliningIndicators/ExpandedBlockEnd.gif        }  
http://www.cnblogs.com/Images/OutliningIndicators/None.gif   
http://www.cnblogs.com/Images/OutliningIndicators/None.gif        private void CallGC()  
http://www.cnblogs.com/Images/OutliningIndicators/ExpandedBlockStart.gif        {  
http://www.cnblogs.com/Images/OutliningIndicators/InBlock.gif            GC.Collect();  
http://www.cnblogs.com/Images/OutliningIndicators/ExpandedBlockEnd.gif        }  
http://www.cnblogs.com/Images/OutliningIndicators/None.gif   
http://www.cnblogs.com/Images/OutliningIndicators/None.gif        // Show destructor  
http://www.cnblogs.com/Images/OutliningIndicators/None.gif        Create();  
http://www.cnblogs.com/Images/OutliningIndicators/None.gif        Debug.WriteLine( "After created!" );  
http://www.cnblogs.com/Images/OutliningIndicators/None.gif        CallGC();

运行的结果为：  
**After created!**  
**Destructor called!**

显然在出了Create函数外，myClass对象的析构函数没有被立刻调用，而是等显示调用GC.Collect才被调用。  
   
对于Dispose来说，也需要显示的调用，但是对于继承了IDisposable的类型对象可以使用using这个关键字，这样对象的Dispose方法在出了using范围后会被自动调用。例如：

http://www.cnblogs.com/Images/OutliningIndicators/None.gif    using( DisposeClass myClass = new DisposeClass() )  
http://www.cnblogs.com/Images/OutliningIndicators/ExpandedBlockStart.gif    {  
http://www.cnblogs.com/Images/OutliningIndicators/InBlock.gif        //other operation here  
http://www.cnblogs.com/Images/OutliningIndicators/ExpandedBlockEnd.gif    }  
http://www.cnblogs.com/Images/OutliningIndicators/None.gif

如上运行的结果如下：  
**Dispose called!**

      那么对于如上DisposeClass类型的Dispose实现来说，事实上GC还需要调用对象的析构函数，**按照前面的GC流程来说，GC对于需要调用析构函数的对象来说，至少经过两个步骤，即首先调用对象的析构函数，其次回收内存。也就是说，按照上面所写的Dispose函数，虽说被执行了，但是GC还是需要执行析构函数，那么一个完整的Dispose函数，应该通过调用GC.SuppressFinalize(this )来告诉GC，让它不用再调用对象的析构函数中。那么改写后的DisposeClass如下：**

http://www.cnblogs.com/Images/OutliningIndicators/ExpandedBlockStart.gif  
http://www.cnblogs.com/Images/OutliningIndicators/ExpandedBlockStart.gif    ///<summary>  
http://www.cnblogs.com/Images/OutliningIndicators/InBlock.gif    /// The class to show three disposal function  
http://www.cnblogs.com/Images/OutliningIndicators/ExpandedBlockEnd.gif    ///</summary>   
http://www.cnblogs.com/Images/OutliningIndicators/None.gif    public class DisposeClass:IDisposable  
http://www.cnblogs.com/Images/OutliningIndicators/ExpandedBlockStart.gif    {  
http://www.cnblogs.com/Images/OutliningIndicators/InBlock.gif        public void Close()  
http://www.cnblogs.com/Images/OutliningIndicators/ExpandedSubBlockStart.gif        {  
http://www.cnblogs.com/Images/OutliningIndicators/InBlock.gif            Debug.WriteLine( "Close called!" );  
http://www.cnblogs.com/Images/OutliningIndicators/ExpandedSubBlockEnd.gif        }  
http://www.cnblogs.com/Images/OutliningIndicators/InBlock.gif   
http://www.cnblogs.com/Images/OutliningIndicators/InBlock.gif        ~DisposeClass()  
http://www.cnblogs.com/Images/OutliningIndicators/ExpandedSubBlockStart.gif        {  
http://www.cnblogs.com/Images/OutliningIndicators/InBlock.gif            Debug.WriteLine( "Destructor called!" );  
http://www.cnblogs.com/Images/OutliningIndicators/ExpandedSubBlockEnd.gif        }  
http://www.cnblogs.com/Images/OutliningIndicators/InBlock.gif   
http://www.cnblogs.com/Images/OutliningIndicators/ExpandedSubBlockStart.gif        #region IDisposable Members  
http://www.cnblogs.com/Images/OutliningIndicators/InBlock.gif   
http://www.cnblogs.com/Images/OutliningIndicators/InBlock.gif        public void Dispose()  
http://www.cnblogs.com/Images/OutliningIndicators/ExpandedSubBlockStart.gif        {  
http://www.cnblogs.com/Images/OutliningIndicators/InBlock.gif            // TODO: Add DisposeClass.Dispose implementation  
http://www.cnblogs.com/Images/OutliningIndicators/InBlock.gif            Debug.WriteLine( "Dispose called!" );  
http://www.cnblogs.com/Images/OutliningIndicators/InBlock.gif            GC.SuppressFinalize( this );  
http://www.cnblogs.com/Images/OutliningIndicators/ExpandedSubBlockEnd.gif        }  
http://www.cnblogs.com/Images/OutliningIndicators/InBlock.gif   
http://www.cnblogs.com/Images/OutliningIndicators/ExpandedSubBlockEnd.gif        #endregion  
http://www.cnblogs.com/Images/OutliningIndicators/ExpandedBlockEnd.gif    }  
http://www.cnblogs.com/Images/OutliningIndicators/None.gif  
http://www.cnblogs.com/Images/OutliningIndicators/None.gif

      通过如下的代码进行测试。

http://www.cnblogs.com/Images/OutliningIndicators/ExpandedBlockStart.gif  
http://www.cnblogs.com/Images/OutliningIndicators/None.gif        private void Run()  
http://www.cnblogs.com/Images/OutliningIndicators/ExpandedBlockStart.gif        {  
http://www.cnblogs.com/Images/OutliningIndicators/InBlock.gif            using( DisposeClass myClass = new DisposeClass() )  
http://www.cnblogs.com/Images/OutliningIndicators/ExpandedSubBlockStart.gif            {  
http://www.cnblogs.com/Images/OutliningIndicators/InBlock.gif                //other operation here  
http://www.cnblogs.com/Images/OutliningIndicators/ExpandedSubBlockEnd.gif            }  
http://www.cnblogs.com/Images/OutliningIndicators/ExpandedBlockEnd.gif        }  
http://www.cnblogs.com/Images/OutliningIndicators/None.gif   
http://www.cnblogs.com/Images/OutliningIndicators/None.gif        private void CallGC()  
http://www.cnblogs.com/Images/OutliningIndicators/ExpandedBlockStart.gif        {  
http://www.cnblogs.com/Images/OutliningIndicators/InBlock.gif            GC.Collect();  
http://www.cnblogs.com/Images/OutliningIndicators/ExpandedBlockEnd.gif        }  
http://www.cnblogs.com/Images/OutliningIndicators/None.gif   
http://www.cnblogs.com/Images/OutliningIndicators/None.gif        // Show destructor  
http://www.cnblogs.com/Images/OutliningIndicators/None.gif        Run();  
http://www.cnblogs.com/Images/OutliningIndicators/None.gif        Debug.WriteLine( "After Run!" );  
http://www.cnblogs.com/Images/OutliningIndicators/None.gif        CallGC();  
http://www.cnblogs.com/Images/OutliningIndicators/None.gif

运行的结果如下：  
**Dispose called!**  
**After Run!**

显然对象的析构函数没有被调用。通过如上的实验以及文字说明，大家会得到如下的一个对比表格。 **析构函数Dispose方法Close方法意义销毁对象销毁对象关闭对象资源调用方式不能被显示调用，会被GC调用需要显示调用或者通过using语句需要显示调用调用时机不确定确定，在显示调用或者离开using程序块确定，在显示调用时** 那么在定义一个类型的时候，是否一定要给出这三个函数地实现呢。 **我的建议大致如下。**<!--[if !supportLists]-->**1．**<!--[endif]-->**提供析构函数，避免资源未被释放，主要是指非内存资源；**<!--[if !supportLists]-->**2．**<!--[endif]-->**对于Dispose和Close方法来说，需要看所定义的类型所使用的资源（参看前面所说），而决定是否去定义这两个函数；**<!--[if !supportLists]-->**3．**<!--[endif]-->**在实现Dispose方法的时候，一定要加上“GC.SuppressFinalize( this )”语句，避免再让GC调用对象的析构函数。** C#程序所使用的内存是受托管的，但不意味着滥用，好地编程习惯有利于提高代码的质量以及程序的运行效率。