**C# 装箱与拆箱**

知识点   
 值类型。   
   值类型是在栈中分配内存，在声明时初始化才能使用，不能为null。   
   值类型超出作用范围系统自动释放内存。   
   主要由两类组成：结构，枚举（enum），结构分为以下几类：   
    1、整型（Sbyte、Byte、Char、Short、Ushort、Int、Uint、Long、Ulong）   
    2、浮点型（Float、Double）   
    3、decimal   
    4、bool   
    5、用户定义的结构（struct）   
 引用类型。   
   引用类型在堆中分配内存，初始化时默认为null。   
   引用类型是通过垃圾回收机制进行回收。   
   包括类、接口、委托、数组以及内置引用类型object与string。   
概念  
   由于C#中所有的数据类型都是由基类System.Object继承而来的，所以值类型和引用类型的值可以通过显式或隐式）操作相互转换，而这转换过程也就是装箱（boxing）和拆箱（unboxing）过程。

装箱   是值类型到 object 类型或到此值类型所实现的任何接口类型的隐式转换。对值类型装箱会在堆中分配一

个对象实例，并将该值复制到新的对象中。  
拆箱   是从 object 类型到值类型或从接口类型到实现该接口的值类型的显式转换。

-------------------

为何需要装箱？   
一种最普通的场景是，调用一个含类型为Object的参数的方法，该Object可支持任意为型，以便通用。当你需

要将一个值类型(如Int32)传入时，需要装箱。   
另一种用法是，一个非泛型的容器，同样是为了保证通用，而将元素类型定义为Object。于是，要将值类型数据

加入容器时，需要装箱。

装箱的内部操作。   
装箱： 对值类型在堆中分配一个对象实例，并将该值复制到新的对象中。按三步进行。   
  第一步：新分配托管堆内存(大小为值类型实例大小加上一个方法表指针和一个SyncBlockIndex)。   
  第二步：将值类型的实例字段拷贝到新分配的内存中。   
  第三步：返回托管堆中新分配对象的地址。这个地址就是一个指向对象的引用了。   
拆箱：检查对象实例，确保它是给定值类型的一个装箱值。将该值从实例复制到值类型变量

中。

装箱/拆箱对执行效率的影响(如何优化效率)  
装箱时,生成的是全新的引用对象,这会有时间损耗,也就是造成效率降低。 那该如何做呢？

避免装箱的方法：  
  1、通过重载函数来避免。  
  2、通过泛型来避免。   
  凡事并不能绝对，假设你想改造的代码为第三方程序集，你无法更改，那你只能是装箱了

。 对于装箱/拆箱代码的优化，由于C#中对装箱和拆箱都是隐式的，所以，根本的方法是对

代码进行分析，而分析最直接的方式是了解原理结何查看反编译的IL代码。比如：在循环体

中可能存在多余的装箱，你可以简单采用提前装箱方式进行优化。

对装箱/拆箱更进一步的了解 装箱/拆箱并不如上面所讲那么简单明了，  
比如：装箱时,变为引用对象,会多出一个方法表指针,这会有何用处呢？ 通过示例来进一步探讨。   
例子:   
  Struct A : ICloneable   
  {   
    public Int32 x;   
    public override String ToString()   
    {   
      return String.Format(”{0}”,x);   
    }  
    public object Clone()   
    {   
      return MemberwiseClone();   
    }   
  }   
  static void main()   
  {   
    A a;   
    a.x = 100;   
    Console.WriteLine(a.ToString());   
    Console.WriteLine(a.GetType());   
    A a2 = (A)a.Clone();   
    ICloneable c = a2; Ojbect o = c.Clone();   
  }  
 ：a.ToString()。编译器发现A重写了ToString方法，会直接调用ToString的指令。因为A是值类型，编译器不会出现多态行为。因此，直接调用，不装箱。(注：ToString是A的基类System.ValueType的方法)   
 ：a.GetType()，GetType是继承于System.ValueType的方法，要调用它，需要一个方法表指针，于是a将被装箱，从而生成方法表指针，调用基类的System.ValueType。(补一句，所有的值类型都是继承于System.ValueType的)。   
 ：a.Clone()，因为A实现了Clone方法，所以无需装箱。 5.3：ICloneable转型：当a2为转为接口类型时，必须装箱，因为接口是一种引用类型。   
 ：c.Clone()。无需装箱，在托管堆中对上一步已装箱的对象进行调用。   
附：其实上面的基于一个根本的原理，因为未装箱的值类型没有方法表指针，所以，不能通过值类型来调用其上继承的虚方法。另外，接口类型是一个引用类型。对此，我的理解，该方法表指针类似C++的虚函数表指针，它是用来实现引用对象的多态机制的重要依据。

如何更改已装箱的对象?  
  对于已装箱的对象，因为无法直接调用其指定方法，所以必须先拆箱，再调用方法，但再次拆箱，会生成新的栈

实例，而无法修改装箱对象。有点晕吧，感觉在说绕口令。还是举个例子来说：  
(在上例中追加change方法)   
   public void Change(Int32 x)   
   {  
     this.x = x;   
   }   
//调用：   
   A a = new A();   
   a.x = 100;   
   Object o = a; //装箱成o，下面，想改变o的值。   
   ((A)o).Change(200); //改掉了吗？没改掉。 没改掉的原因是o在拆箱时，生成的是临时的栈实例A，所以，改

动是基于临时A的，并未改到装箱对象。   
(附：在托管C++中，允许直接取加拆箱时第一步得到的实例引用，而直接更改，但C#不行。) 那该如何是好？ 嗯，

通过接口方式，可以达到相同的效果。 实现如下：   
  interface IChange   
  {   
    void Change(Int32 x);   
  }   
  struct A : IChange   
  {   
   …   
  }   
//调用：  
  ((IChange)o).Change(200);//改掉了吗？改掉了。 为啥现在可以改？ 在将o转型为IChange时，这里不会进行再

次装箱，当然更不会拆箱，因为o已经是引用类型，再因为它是IChange类型，所以可以直接调用Change，于是，更

改的也就是已装箱对象中的字段了，达到期望的效果。

-------------------------------------------------------------------------------------------------------------

-------------------------------------------------------------------------------------------------------------

-------------------------------------------------------------------------------------------------------------

**附录：(转自**<http://www.cnblogs.com/hunts/archive/2007/01/19/boxing_unboxing.html>和

<http://www.cnblogs.com/cry/archive/2009/03/13/1410903.html>**）** a、装箱  
     一个很简单的例子。新建一个控制台程序，在Main()里面就写两句话。      
       
     int i = 13;  
     object ob = i;  
   
     编译。然后用.net 提供的工具ILDASM.exe（MSIL Disassembler ）查看新生产这个程序的配件代码（Microsoft intermediate language ，MSIL。顺带说一句.net framework SDK除了这个MSIL的反汇编工具，当然还提供了汇编工具ILASM.exe，可以使用MSIL编写程序，当然。。谁也不会没事这么干。那个反汇编工具倒是挺有用，可以了解一些底层机制）  
   
     用那个工具查看一下编译后程序的Main(string[] args)方法，显示如下（我现在用的时.net framework 2.0可能MSIL代码显示出来的和原来的1.0或者1.1稍有不同，不过没关系核心没变）：  
.method private hidebysig static void  Main(string[] args) cil managed  
{  
  .entrypoint  
  // Code size       12 (0xc)  
  .maxstack  1  
  .locals init ([0] int32 i,  
           [1] object ob)  
  IL\_0000:  nop  
  IL\_0001:  ldc.i4.s   13  
  IL\_0003:  stloc.0  
  IL\_0004:  ldloc.0  
  IL\_0005:  box        [mscorlib]System.Int32  
  IL\_000a:  stloc.1  
  IL\_000b:  ret  
} // end of method Program::Main  
稍微解释一下：  
 (1)先注意  .locals ，定义了两个类型分别为int32 和object 的局部变量  
 (2)然后看   IL\_0001处，ldc是个指令，后面的i4.s指出作为32位（4个字节）整数被压入堆栈。而压入的值就是13     
 (3)下面的stloc把上面的值从堆栈弹出给局部变量i,这里的.0是指弹出给到第一个局部变量中，也就是i了  
 (4)这个值（13），被弹出后，就被装载回堆栈，也就是后面IL\_0004行的ldloc命令做的事情  
 (5)然后使用CIL（Common Language Infrastructure ）box将这个值转换为引用类型。装箱喽～  
 (6)stloc.1根据（3）的解释就好理解了，就是把box返回值弹出给第二个局部变量ob中。  
   
  但是这个box指令内部又发生了什么呢？有牛人告诉了我们。  
  (1)在堆上分配内存。因为值类型最终有一个对象代表，所有堆上分配的内存量必须是值类型的大小加上容纳此对象及其内部结构（比如虚拟方法表）所需的内存量。  
  (2)值类型的值被复制到新近分配的内存中  
  (3)新近分配的对象地址被放到堆栈上，现在它指向一个引用类型  
   
   b、拆箱  
   在刚才程序的基础上，再加一句话变成，编译：  
            int i = 13;  
            object ob = i;  
            int j = (int)ob;  
   在装箱的时候，并不需要显示类型转换。但在拆箱时需要类型转换。这是因为在拆箱时对象可以被转换为任何类型。看看MSIL代码变成这德行了：  
.method private hidebysig static void  Main(string[] args) cil managed  
{  
  .entrypoint  
  // Code size       19 (0x13)  
  .maxstack  1  
  .locals init ([0] int32 i,  
           [1] object ob,  
           [2] int32 j)  
  IL\_0000:  nop  
  IL\_0001:  ldc.i4.s   13  
  IL\_0003:  stloc.0  
  IL\_0004:  ldloc.0  
  IL\_0005:  box        [mscorlib]System.Int32  
  IL\_000a:  stloc.1  
  IL\_000b:  ldloc.1  
  IL\_000c:  unbox.any  [mscorlib]System.Int32  
  IL\_0011:  stloc.2  
  IL\_0012:  ret  
} // end of method Program::Main  
   
    整个流程就不再重复叙述了，参照前面的解释现在这个过程应该能看明白。  
    说说拆箱unbox的内部过程：  
   (1)因为一个对象将被转换，所以编译器必须先判断堆栈上指向合法对象的地址，以及这个对象类型是否可以转换为MSL unbox指令调用中指定的值类型。如果检查失败就抛出InvalidCastException异常。  
   (2)校验通过后，就返回指向对象内的值的指针。可以看出，装箱操作会创建转换类型的副本，而拆箱就不会。不过注意一下，在我们装箱的时候是先把变量i的值复制了一份赋给ob的，所变量j拿到的是ob这个变量的引用。也就是后面再改变i的值并不会影响j的值，但是改变ob的值就会。  
   
   c、再来一个稍微复杂点的例子，有如下代码：  
            int i = 13;  
            object ob = i;  
            Console.WriteLine(i + "," + (Int32)ob);  
    这里做了几次装箱和拆箱操作呢？我开始想当然的以为是1次装1次拆箱操作了，可实际上确是3次装箱1次拆箱操作！先看看MSIL代码：  
.method private hidebysig static void  Main(string[] args) cil managed  
{  
  .entrypoint  
  // Code size       45 (0x2d)  
  .maxstack  3  
  .locals init ([0] int32 i,  
           [1] object ob)  
  IL\_0000:  nop  
  IL\_0001:  ldc.i4.s   13  
  IL\_0003:  stloc.0  
  IL\_0004:  ldloc.0  
  IL\_0005:  box        [mscorlib]System.Int32  
  IL\_000a:  stloc.1  
  IL\_000b:  ldloc.0  
  IL\_000c:  box        [mscorlib]System.Int32  
  IL\_0011:  ldstr      ","  
  IL\_0016:  ldloc.1  
  IL\_0017:  unbox.any  [mscorlib]System.Int32  
  IL\_001c:  box        [mscorlib]System.Int32  
  IL\_0021:  call       string [mscorlib]System.String::Concat(object,  
                                                              object,  
                                                              object)  
  IL\_0026:  call       void [mscorlib]System.Console::WriteLine(string)  
  IL\_002b:  nop  
  IL\_002c:  ret  
} // end of method Program::Main  
   
    (1)前面好说，跟前面一样  object ob = i;引起了一次装箱操作也就是 IL\_0005处代码。  
    (2)后面可以看出Console.WriteLine方法调用的是单个String作为参数的版本。因此上面调用了String.Concat方法将i + "," + (Int32)ob这3个值连接产生单个String再传给WriteLine。  
    (3)String.Concat的重载版本里面找到最匹配的就是Concat(object, object,object)。这样为了匹配这3个参数：  
        (3.1) IL\_000c处代码，第一个参数i被装箱   
        (3.2)IL\_0011 处ldstr      "," 就是将字符串'，'压入堆栈  
        (3.3)然后 IL\_0017 (int32)ob引起了一次拆箱操作  
        (3.4)我们可怜的(int32)ob，又为了匹配Concat的参数，再次被装箱(IL\_001c)  
   明显后面那个(int32)ob造成了一次不必要的拆箱和装箱操作！所以正因为.net的自动类型处理能力，还是小心地注意一下写法，否则就会引起不必有的性能损失。  
   
   下面举类似的小例子  
    还是个那个控制台代码写成这样  
        static ArrayList al;  
        static void Main(string[] args)  
        {  
            int i = 13;  
            al = new ArrayList();  
            al.Add(i);  
            Console.WriteLine("{0}", i);  
              
        }   
  MSIL命令如下：  
.method private hidebysig static void  Main(string[] args) cil managed  
{  
  .entrypoint  
  // Code size       49 (0x31)  
  .maxstack  2  
  .locals init ([0] int32 i)  
  IL\_0000:  nop  
  IL\_0001:  ldc.i4.s   13  
  IL\_0003:  stloc.0  
  IL\_0004:  newobj     instance void [mscorlib]System.Collections.ArrayList::.ctor()  
  IL\_0009:  stsfld     class [mscorlib]System.Collections.ArrayList ConsoleApplication1.Program::al  
  IL\_000e:  ldsfld     class [mscorlib]System.Collections.ArrayList ConsoleApplication1.Program::al  
  IL\_0013:  ldloc.0  
  IL\_0014:  box        [mscorlib]System.Int32  
  IL\_0019:  callvirt   instance int32 [mscorlib]System.Collections.ArrayList::Add(object)  
  IL\_001e:  pop  
  IL\_001f:  ldstr      "{0}"  
  IL\_0024:  ldloc.0  
  IL\_0025:  box        [mscorlib]System.Int32  
  IL\_002a:  call       void [mscorlib]System.Console::WriteLine(string,  
                                                                object)  
  IL\_002f:  nop  
  IL\_0030:  ret  
} // end of method Program::Main  
   
其他的都不用管，看懂了前面我说的，那么这里就知道因为ArrayList.Add(object)做了一次装箱和Console.WriteLine(string,object)又做了一次装箱。如果我们换一种写法，把程序改成这样：  
   
        static ArrayList al;  
        static void Main(string[] args)  
        {  
            int i = 13;  
            object ob = i;  
            al = new ArrayList();  
            al.Add(ob);  
            Console.WriteLine("{0}", ob);  
              
        }  
   
MSIL就变成：  
.method private hidebysig static void  Main(string[] args) cil managed  
{  
  .entrypoint  
  // Code size       46 (0x2e)  
  .maxstack  2  
  .locals init ([0] int32 i,  
           [1] object ob)  
  IL\_0000:  nop  
  IL\_0001:  ldc.i4.s   13  
  IL\_0003:  stloc.0  
  IL\_0004:  ldloc.0  
  IL\_0005:  box        [mscorlib]System.Int32  
  IL\_000a:  stloc.1  
  IL\_000b:  newobj     instance void [mscorlib]System.Collections.ArrayList::.ctor()  
  IL\_0010:  stsfld     class [mscorlib]System.Collections.ArrayList ConsoleApplication1.Program::al  
  IL\_0015:  ldsfld     class [mscorlib]System.Collections.ArrayList ConsoleApplication1.Program::al  
  IL\_001a:  ldloc.1  
  IL\_001b:  callvirt   instance int32 [mscorlib]System.Collections.ArrayList::Add(object)  
  IL\_0020:  pop  
  IL\_0021:  ldstr      "{0}"  
  IL\_0026:  ldloc.1  
  IL\_0027:  call       void [mscorlib]System.Console::WriteLine(string,  
                                                                object)  
  IL\_002c:  nop  
  IL\_002d:  ret  
} // end of method Program::Main