

C# 类型基础

C# Type Fundamentals

张子阳

www.tracefact.net

jimmy_dev@163.com

引言

本文之初的目的是讲述设计模式中的 Prototype(原型)模式,但是如果较清楚地弄明白这个模式,需要了解对象克隆(Object Clone), Clone 其实也就是对象复制。复制又分为了浅度复制(Shallow Copy)和 深度复制(Deep Copy), 浅度复制 和 深度复制又是以 如何复制引用类型成员来划分的。由此又引出了 引用类型 和 值类型, 以及相关的对象判等、装箱、拆箱等基础知识。

于是我干脆新起一篇,从最基础的类型开始自底向上写起了。我仅仅想对于这个主题的理解表述出来,一是总结和复习,二是交流经验,或许有地方我理解的有偏差,希望指正。如果前面基础的内容对你来说过于简单,可以跳跃阅读。

值类型 和 引用类型

我们先简单回顾一下 C#中的类型系统。C# 中的类型一共分为两类,一类是值类型(Value Type),一类是引用类型(Reference Type)。值类型 和 引用类型是以它们在计算机内存中是如何被分配的来划分的。值类型包括 结构和枚举,引用类型包括 类、接口、委托 等。还有一种特殊的值类型,称为简单类型(Simple Type),比如 byte, int 等,这些简单类型实际上是 FCL 类库类型的别名,比如声明一个 int 类型,实际上是声明一个 System.Int32 结构类型。因此,在 Int32 类型中定义的操作,都可以应用在 int 类型上,比如 “123.Equals(2)”。

所有的 值类型 都隐式地继承自 System.ValueType 类型(注意 System.ValueType 本身是一个类类型), System.ValueType 和所有的引用类型都继承自 System.Object 基类。你不能显示地让结构继承一个类,因为 C#不支持多重继承,而结构已经隐式继承自 ValueType。

NOTE: 堆栈(stack)是一种后进先出的数据结构,在内存中,变量会被分配在堆栈上来进行操作。堆(heap)是用于为类型实例(对象)分配空间的内存区域,在堆上创建一个对象,会将对象的地址传给堆栈上的变量(反过来叫变量指向此对象,或者变量引用此对象)。

1. 值类型

当声明一个值类型的变量(Variable)的时候,变量本身包含了值类型的全部字段,该变量会被分配在线程堆栈(Thread Stack)上。

假如我们有这样一个值类型,它代表了直线上的一点:

```
public struct ValPoint {
    public int x;

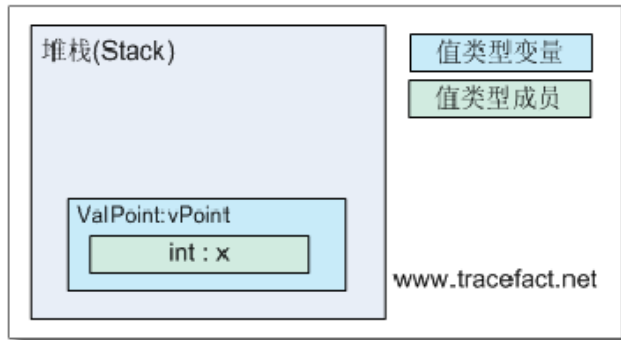
    public ValPoint(int x) {
        this.x = x;
    }
}
```

```
}
```

当我们在程序中写下这样的一条变量的声明语句时：

```
ValPoint vPoint1;
```

实际产生的效果是声明了 vPoint1 变量，变量本身包含了值类型的所有字段(即你想要的所有数据)。



NOTE: 如果观察 MSIL 代码，会发现此时变量还没有被压到栈上，因为 .maxstack(最高栈数) 为 0。并且没有看到入栈的指令，这说明只有对变量进行操作，才会进行入栈。

因为变量已经包含了值类型的所有字段，所以，此时你已经可以对它进行操作了(对变量进行操作，实际上是一系列的入栈、出栈操作)。

```
vPoint1.x = 10;  
Console.WriteLine(vPoint.x); // 输出 10
```

NOTE: 如果 vPoint1 是一个引用类型(比如 class)，在运行时抛出 NullReferenceException 异常。因为 vPoint 是一个值类型，不存在引用，所以永远也不会抛出 NullReferenceException。

如果你不对 vPoint.x 进行赋值，直接写 Console.WriteLine(vPoint.x)，则会出现编译错误：使用了未赋值的局部变量。产生这个错误是因为 .Net 的一个约束：所有的元素使用前都必须初始化。比如这样的语句也会引发这个错误：

```
int i;  
Console.WriteLine(i);
```

解决这个问题我们可以通过这样一种方式：编译器隐式地会为结构类型创建了无参数构造函数。在这个构造函数中会对结构成员进行初始化，所有的值类型成员被赋予 0 或相当于 0 的值(针对 Char 类型)，所有的引用类型被赋予 null 值。(因此，Struct 类型不可以自行声明无参数的构造函数)。所以，我们可以通过隐式声明的构造函数去创建一个 ValPoint 类型变量：

```
ValPoint vPoint1 = new ValPoint();  
Console.WriteLine(vPoint.x); // 输出为 0
```

我们将上面代码第一句的表达式由“=”分隔拆成两部分来看：

- 左边 `ValPoint vPoint1`，在堆栈上创建一个 `ValPoint` 类型的变量 `vPoint`，结构的所有成员均未赋值。在进行 `new ValPoint()` 之前，将 `vPoint` 压到栈上。
- 右边 `new ValPoint()`，`new` 操作符不会分配内存，它仅仅调用 `ValPoint` 结构的默认构造函数，根据构造函数去初始化 `vPoint` 结构的所有字段。

注意上面这句，**new 操作符不会分配内存，仅仅调用 `ValPoint` 结构的默认构造函数去初始化 `vPoint` 的所有字段。**那如果我这样做，又如何解释呢？

```
Console.WriteLine((new ValPoint()).x); // 正常，输出为 0
```

在这种情况下，会创建一个临时变量，然后使用结构的默认构造函数对此临时变量进行初始化。我知道我这样很没有说服力，所以我们来看下 MS IL 代码，为了节省篇幅，我只节选了部分：

```
.locals init ([0] valuetype Prototype.ValPoint CS$0$0000) // 声明临时变量
IL_0000: nop
IL_0001: ldloca.s CS$0$0000 // 将临时变量压栈
IL_0003: initobj Prototype.ValPoint // 初始化此变量
```

而对于 `ValPoint vPoint = new ValPoint()`；这种情况，其 MSIL 代码是：

```
.locals init ([0] valuetype Prototype.ValPoint vPoint) // 声明 vPoint
IL_0000: nop
IL_0001: ldloca.s vPoint // 将 vPoint 压栈
IL_0003: initobj Prototype.ValPoint // 使用 initobj 初始化此变量
```

那么当我们使用自定义的构造函数时，`ValPoint vPoint = new ValPoint(10)`，又会怎么样呢？通过下面的代码我们可以看出，实际上会使用 `call` 指令(instruction)调用我们自定义的构造函数，并传递 10 到参数列表中。

```
.locals init ([0] valuetype Prototype.ValPoint vPoint)
IL_0000: nop
IL_0001: ldloca.s vPoint // 将 vPoint 压栈
IL_0003: ldc.i4.s 10 // 将 10 压栈
// 调用构造函数，传递参数
IL_0005: call instance void Prototype.ValPoint::.ctor(int32)
```

对于上面的 MSIL 代码不清楚不要紧，有的时候知道结果就已经够用了。关于 MSIL 代码，有空了我会为大家翻译一些好的文章。

2. 引用类型

当声明一个引用类型变量的时候，该引用类型的变量会被分配到堆上，这个变量将用于保存

位于堆上的该引用类型的实例 的内存地址，变量本身不包含对象的数据。此时，如果仅仅声明这样一个变量，由于在堆上还没有创建类型的实例，因此，变量值为 null，意思是不指向任何类型实例(堆上的对象)。对于变量的类型声明，用于限制此变量可以保存的类型实例的地址。

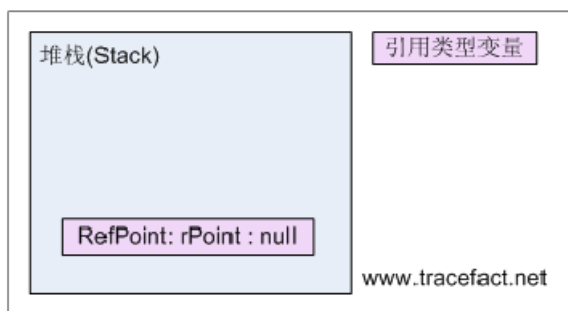
如果我们有一个这样的类，它依然代表直线上的一点：

```
public class RefPoint {  
    public int x;  
  
    public RefPoint(int x) {  
        this.x = x;  
    }  
    public RefPoint() {}  
}
```

当我们仅仅写下一条声明语句：

```
RefPoint rPoint1;
```

它的效果就向下图一样，仅仅在堆栈上创建一个不包含任何数据，也不指向任何对象(不包含创建再堆上的对象的地址)的变量。

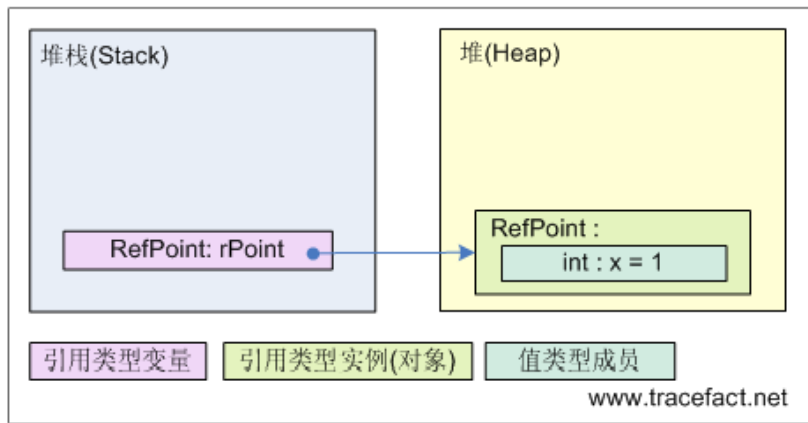


而当我们使用 new 操作符时：

```
rPoint1= new RefPoint(1);
```

会发生这样的事：

1. 在应用程序堆 (Heap) 上创建一个引用类型 (Type) 的实例 (Instance) 或者叫对象 (Object)，并为它分配内存地址。
2. 自动传递该实例的引用给构造函数。(正因为如此，你才可以在构造函数中使用 this 来访问这个实例。)
3. 调用该类型的构造函数。
4. 返回该实例的引用(内存地址)，赋值给 rPoint 变量。



3. 关于简单类型

很多文章和书籍中在讲述这类问题的时候，总是喜欢用一个 `int` 类型作为 值类型 和一个 `Object` 类型 作为引用类型来作说明。本文将采用自定义的一个 结构 和 类 分别作值类型和引用类型的说明。这是因为简单类型(比如 `int`)有一些 CLR 实现了的行为，这些行为会让我们对一些操作产生误解。

举个例子，如果我们想比较两个 `int` 类型是否相等，我们会通常这样：

```
int i = 3;
int j = 3;
if(i==j) Console.WriteLine("i equals to j");
```

但是，对于自定义的值类型，比如结构，就不能用 “`==`” 来判断它们是否相等，而需要在变量上使用 `Equals()` 方法来完成。

再举个例子，大家知道 `string` 是一个引用类型，而我们比较它们是否相等，通常会这样做：

```
string a = "123456"; string b = "123456";
if(a == b) Console.WriteLine("a Equals to b");
```

实际上，在后面我们就会看到，当使用 “`==`” 对引用类型变量进行比较的时候，比较的是它们是否指向的堆上同一个对象。而上面 `a`、`b` 指向的显然是不同的对象，只是对象包含的值相同，所以可见，对于 `string` 类型，CLR 对它们的比较实际上比较的是值，而不是引用。

为了避免上面这些引起的混淆，在对象判等部分将采用自定义的结构和类来分别说明。

装箱 和 拆箱

这部分内容可深可浅，本文只简要地作一个回顾。简单来说，装箱 就是 将一个值类型转换

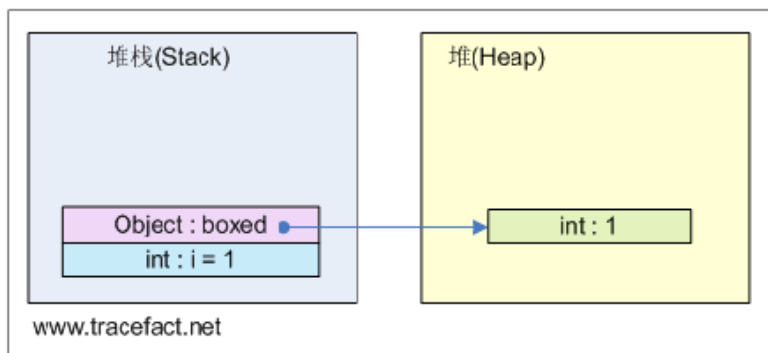
成等值的引用类型。它的过程分为这样几步：

1. 在堆上为新生成的对象(该对象包含数据，对象本身没有名称)分配内存。
2. 将 堆栈上 值类型变量的值拷贝到 堆上的对象 中。
3. 将堆上创建的对象地址返回给引用类型变量(从程序员角度看，这个变量的名称就好像堆上对象的名称一样)。

当我们运行这样的代码时：

```
int i = 1;
Object boxed = i;
Console.WriteLine("Boxed Point: " + boxed);
```

效果图是这样的：



MSIL 代码是这样的：

```
.method private hidebysig static void Main(string[] args) cil managed
{
    .entrypoint
    // 代码大小      19 (0x13)
    .maxstack 1 // 最高栈数是 1, 装箱操作后 i 会出栈
    .locals init ([0] int32 i, // 声明变量 i(第 1 个变量, 索引为 0)
                 [1] object boxed) // 声明变量 boxed (第 2 个变量, 索引为 1)
    IL_0000: nop
    IL_0001: ldc.i4.s 10 // #1 将 10 压栈
    IL_0003: stloc.0 // #2 10 出栈, 将值赋给 i
    IL_0004: ldloc.0 // #3 将 i 压栈
    IL_0005: box [mscorlib]System.Int32 // #4 i 出栈, 对 i 装箱(复制值到堆, 返回地址)
    IL_000a: stloc.1 // #5 将返回值赋给变量 boxed
    IL_000b: ldloc.1 // 将 boxed 压栈
    // 调用 WriteLine()方法
    IL_000c: call void [mscorlib]System.Console::WriteLine(object)
    IL_0011: nop
}
```

```
IL_0012: ret
} // end of method Program::Main
```

而拆箱则是将一个 **已装箱的引用类型** 转换为值类型：

```
int i = 1;
Object boxed = i;
int j;
j = (int)boxed;           // 显示声明 拆箱后的类型
Console.WriteLine("UnBoxed Point: " + j);
```

需要注意的是：UnBox 操作需要显示声明拆箱后转换的类型。它分为两步来完成：

1. 获取已装箱的对象的地址。
2. 将值从堆上的对象中拷贝到堆栈上的值变量中。

对象判等

因为我们要提到对象克隆(复制)，那么，我们应该有办法知道复制前后的两个对象是否相等。所以，在进行下面的章节前，我们有必要先了解如何进行对象判等。

NOTE: 有机会较深入地研究这部分内容，需要感谢 微软的开源 以及 VS2008 的FCL调试功能。关于如何调试 FCL 代码，请参考 [Configuring Visual Studio to Debug .NET Framework Source Code](#)。

我们先定义用作范例的两个类型，它们代表直线上的一点，唯一区别一个是引用类型 class，一个是值类型 struct：

```
public class RefPoint {    // 定义一个引用类型
    public int x;
    public RefPoint(int x) {
        this.x = x;
    }
}

public struct ValPoint {  // 定义一个值类型
    public int x;
    public ValPoint(int x) {
        this.x = x;
    }
}
```

1. 引用类型判等

我们先进行引用类型对象的判等，我们知道在 `System.Object` 基类型中，定义了实例方法 `Equals(object obj)`，静态方法 `Equals(object objA, object objB)`，静态方法 `ReferenceEquals(object objA, object objB)` 来进行对象的判等。

我们先看看这三个方法，注意我在代码中用 `#number` 标识的地方，后文中我会直接引用：

```
public static bool ReferenceEquals (Object objA, Object objB)
{
    return objA == objB;          // #1
}

public virtual bool Equals(Object obj)
{
    return InternalEquals(this, obj); // #2
}

public static bool Equals(Object objA, Object objB) {
    if (objA==objB) {           // #3
        return true;
    }

    if (objA==null || objB==null) {
        return false;
    }

    return objA.Equals(objB); // #4
}
```

我们先看 `ReferenceEquals(object objA, object objB)` 方法，它实际上简单地返回 `objA == objB`，所以，在后文中，除非必要，我们统一使用 `objA == objB` (省去了 `ReferenceEquals` 方法)。另外，为了范例简单，我们不考虑对象为 `null` 的情况。

我们来看第一段代码：

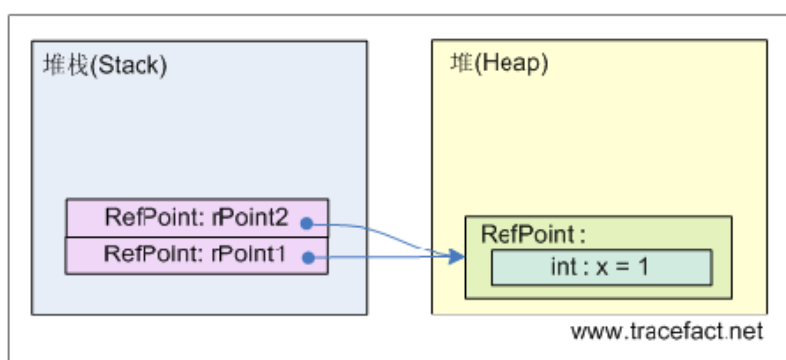
```
// 复制对象引用
bool result;
RefPoint rPoint1 = new RefPoint(1);
RefPoint rPoint2 = rPoint1;

result = (rPoint1 == rPoint2); // 返回 true;
Console.WriteLine(result);
```

```
result = rPoint1.Equals(rPoint2); // #2 返回 true;
Console.WriteLine(result);
```

在阅读本文中，应该时刻在脑子里构思一个堆栈，一个堆，并思考着每条语句会在这两种结构上产生怎么样的效果。在这段代码中，产生的效果是：在堆上创建了一个新的 RefPoint 类型的实例(对象)，并将它的 x 字段初始化为 1；在堆栈上创建变量 rPoint1，rPoint1 保存堆上这个对象的地址；将 rPoint1 赋值给 rPoint2 时，此时并没有在堆上创建一个新的对象，而是将之前创建的对象地址复制到了 rPoint2。此时，rPoint1 和 rPoint2 指向了堆上同一个对象。

从 ReferenceEquals() 这个方法名就可以看出，它判断两个引用变量是不是指向了同一个变量，如果是，那么就返回 true。这种相等叫做 **引用相等** (rPoint1 == rPoint2 等效于 ReferenceEquals)。因为它们指向的是同一个对象，所以对 rPoint1 的操作将会影响 rPoint2：



注意 System.Object 静态的 Equals(Object objA, Object objB) 方法，在 #3 处，如果两个变量引用相等，那么将直接返回 true。所以，可以预见我们上面的代码 rPoint1.Equals(rPoint2); 在 #3 就会返回 true。但是我们没有调用静态 Equals()，直接调用了实体方法，最后调用了 #2 的 InternalEquals()，返回 true。(InternalEquals() 无资料可查，仅通过调试测得)。

我们再看引用类型的第二种情况：

```
//创建新引用类型的对象，其成员的值相等
RefPoint rPoint1 = new RefPoint(1);
RefPoint rPoint2 = new RefPoint(1);

result = (rPoint1 == rPoint2);
Console.WriteLine(result); // 返回 false;

result = rPoint1.Equals(rPoint2);
Console.WriteLine(result); // #2 返回 false
```

上面的代码在堆上创建了两个类型实例，并用同样的值初始化它们；然后将它们的地址分别赋值给堆上的变量 rPoint1 和 rPoint2。此时 #2 返回了 false，可以看到，**对于引用类型，即使类型的实例(对象)包含的值相等，如果变量指向的是不同的对象，那么也不相等。**

2. 简单值类型判等

注意本节的标题：简单值类型判等，这个简单是如何定义的呢？如果值类型的成员仅包含值类型，那么我们暂且管它叫简单值类型，如果值类型的成员包含引用类型，我们管它叫复杂值类型。（注意，这只是本文中为了说明我个人作的定义。）

应该还记得我们之前提过，值类型都会隐式地继承自 `System.ValueType` 类型，而 `ValueType` 类型覆盖了基类 `System.Object` 类型的 `Equals()` 方法，在值类型上调用 `Equals()` 方法，会调用 `ValueType` 的 `Equals()`。所以，我们看看这个方法是什么样的，依然用 `#number` 标识后面会引用的地方。

```
public override bool Equals (Object obj) {
    if (null==obj) {
        return false;
    }
    RuntimeType thisType = (RuntimeType)this.GetType();
    RuntimeType thatType = (RuntimeType)obj.GetType();

    if (thatType!=thisType) { // 如果两个对象不是一个类型，直接返回 false
        return false;
    }

    Object thisObj = (Object)this;
    Object thisResult, thatResult;

    if (CanCompareBits(this)) // #5
        return FastEqualsCheck(thisObj, obj); // #6

    // 利用反射获取值类型所有字段
    FieldInfo[] thisFields = thisType.GetFields(BindingFlags.Instance |
BindingFlags.Public | BindingFlags.NonPublic);
    // 遍历字段，进行字段对字段比较
    for (int i=0; i<thisFields.Length; i++) {
        thisResult = ((RtFieldInfo)thisFields[i]).InternalGetValue(thisObj, false);
        thatResult = ((RtFieldInfo)thisFields[i]).InternalGetValue(obj, false);

        if (thisResult == null) {
            if (thatResult != null)
                return false;
        }
        else
            if (!thisResult.Equals(thatResult)) { // #7
                return false;
            }
    }
}
```

```
}  
  
return true;  
}
```

我们先来看看第一段代码：

```
// 复制结构变量  
ValPoint vPoint1 = new ValPoint(1);  
ValPoint vPoint2 = vPoint1;  
  
result = (vPoint1 == vPoint2); //编译错误：不能在 ValPoint 上应用 "==" 操作符  
Console.WriteLine(result);  
  
result = Object.ReferenceEquals(vPoint1, vPoint2); // 隐式装箱，指向了堆上的不同对象  
Console.WriteLine(result); // 返回 false
```

我们先在堆栈上创建了一个变量 vPoint1，变量本身已经包含了所有字段和数据。然后在堆栈上复制了 vPoint1 的一份拷贝给了 vPoint2，从常理思维上来讲，我们认为它应该是相等的。接下来我们就试着去比较它们，可以看到，我们不能用“==”直接去判断，这样会返回一个编译错误。如果我们调用 System.Object 基类的静态方法 ReferenceEquals()，有意思的事情发生了：它返回了 false。为什么呢？我们看下 ReferenceEquals()方法的签名就可以了，它接受的是 Object 类型，也就是引用类型，而当我们传递 vPoint1 和 vPoint2 这两个值类型的时候，会进行一个隐式的装箱，效果相当于下面的语句：

```
Object boxPoint1 = vPoint1;  
Object boxPoint2 = vPoint2;  
result = (boxPoint1 == boxPoint2); // 返回 false  
Console.WriteLine(result);
```

而装箱的过程，我们在前面已经讲述过，上面的操作等于是在堆上创建了两个对象，对象包含的内容相同(地址不同)，然后将对象地址分别返回给堆栈上的 boxPoint1 和 boxPoint2，再去比较 boxPoint1 和 boxPoint2 是否指向同一个对象，显然不是，所以返回 false。

我们继续，添加下面这段代码：

```
result = vPoint1.Equals(vPoint2); // #5 返回 true; #6 返回 true;  
Console.WriteLine(result); // 输出 true
```

因为它们均继承自 ValueType 类型，所以此时会调用 ValueType 上的 Equals()方法，在方法体内部，#5 CanCompareBits(this) 返回了 true，CanCompareBits(this)这个方法，按微软的注释，意识是说：如果对象的成员中存在对于堆上的引用，那么返回 false，如果不存在，返回 true。按照 ValPoint 的定义，它仅包含一个 int 类型的字段 x，自然不存在对堆上其他对象的引用，所以返回了 true。从#5 的名字 CanCompareBits，可以看出是判断是否可以按位比较，那么返回了 true 以后，#6 自然是进行按位比较了。

接下来，我们对 vPoint2 做点改动，看看会发生什么：

```
vPoint2.x = 2;
result = vPoint1.Equals(vPoint2); // #5 返回 true; #6 返回 false;
Console.WriteLine(result);
```

3. 复杂值类型判等

到现在，上面的这些方法，我们还没有走到的位置，就是 CanCompareBits 返回 false 以后的部分了。前面我们已经推测出了 CanCompareBits 返回 false 的条件（值类型的成员包含引用类型），现在只要实现下就可以了。我们定义一个新的结构 Line，它代表直线上的线段，我们让它的一个成员为值类型 ValPoint，一个成员为引用类型 RefPoint，然后去作比较。

```
/* 结构类型 ValLine 的定义，
public struct ValLine {
    public RefPoint rPoint; // 引用类型成员
    public ValPoint vPoint; // 值类型成员
    public Line(RefPoint rPoint, ValPoint vPoint) {
        this.rPoint = rPoint;
        this.vPoint = vPoint;
    }
}
*/

RefPoint rPoint = new RefPoint(1);
ValPoint vPoint = new ValPoint(1);

ValLine line1 = new ValLine (rPoint, vPoint);
ValLine line2 = line1;

result = line1.Equals(line2); // 此时已经存在一个装箱操作，调用 ValueType.Equals()
Console.WriteLine(result); // 返回 True
```

这个例子的过程要复杂得多。在开始前，我们先思考一下，当我们写下 line1.Equals(line2) 时，已经进行了一个装箱的操作。如果要进一步判等，显然不能去判断变量是否引用的堆上同一个对象，这样的话就没有意义了，因为总是会返回 false（装箱后堆上创建了两个对象）。那么应该如何判断呢？对 堆上对象 的成员（字段）进行一对一的比较，而成员又分为两种类型，一种是值类型，一种是引用类型。对于引用类型，去判断是否引用相等；对于值类型，如果是简单值类型，那么如同前一节讲述的去判断；如果是复杂类型，那么当然是递归调用了；最终直到要么是引用类型要么是简单值类型。

NOTE: 进行字段对字段的一对一比较，需要用到反射，如果不了解反射，可以参看 [.Net 中的反射](#) 系列文章。

好了，我们现在看看实际的过程，是不是如同我们料想的那样，为了避免频繁的拖动滚动条查看 ValueType 的 Equals() 方法，我拷贝了部分下来：

```
public override bool Equals (Object obj) {

    if (CanCompareBits(this))                // #5
        return FastEqualsCheck(thisObj, obj); // #6
    // 利用反射获取类型的所有字段(或者叫类型成员)
    FieldInfo[] thisFields = thisType.GetFields(BindingFlags.Instance |
BindingFlags.Public | BindingFlags.NonPublic);
    // 遍历字段进行比较
    for (int i=0; i<thisFields.Length; i++) {
        thisResult = ((RtFieldInfo)thisFields[i]).InternalGetValue(thisObj, false);
        thatResult = ((RtFieldInfo)thisFields[i]).InternalGetValue(obj, false);

        if (thisResult == null) {
            if (thatResult != null)
                return false;
        }
        else
            if (!thisResult.Equals(thatResult)) { #7
                return false;
            }
        }

    return true;
}
```

1. 进入 ValueType 上的 Equals() 方法，#5 处返回了 false；
2. 进入 for 循环，遍历字段。
3. 第一个字段是 RefPoint 引用类型，#7 处，调用 System.Object 的 Equals() 方法，到达#2，返回 true。
4. 第二个字段是 ValPoint 值类型，#7 处，调用 System.ValType 的 Equals() 方法，也就是当前方法本身。此处递归调用。
5. 再次进入 ValueType 的 Equals() 方法，因为 ValPoint 为简单值类型，所以 #5 CanCompareBits 返回了 true，接着 #6 FastEqualsCheck 返回了 true。
6. 里层 Equals() 方法返回 true。
7. 退出 for 循环。
8. 外层 Equals() 方法返回 true。

对象复制

有的时候，创建一个对象可能会非常耗时，比如对象需要从远程数据库中获取数据来填充，又或者创建对象需要读取硬盘文件。此时，如果已经有了一个对象，再创建新对象时，可能会采用复制现有对象的方法，而不是重新建一个新的对象。本节就讨论如何进行对象的复制。

1. 浅度复制

浅度复制 和 深度复制 是以如何复制对象的成员(member)来划分的。一个对象的成员有可能是值类型，有可能是引用类型。当我们对对象进行一个浅度复制的时候，对于值类型成员，会复制其本身(值类型变量本身包含了所有数据，复制时进行按位拷贝)；对于引用类型成员(注意它会引用另一个对象)，仅仅复制引用，而不创建其引用的对象。结果就是：新对象的引用成员 和 复制对象的引用成员 指向了同一个对象。

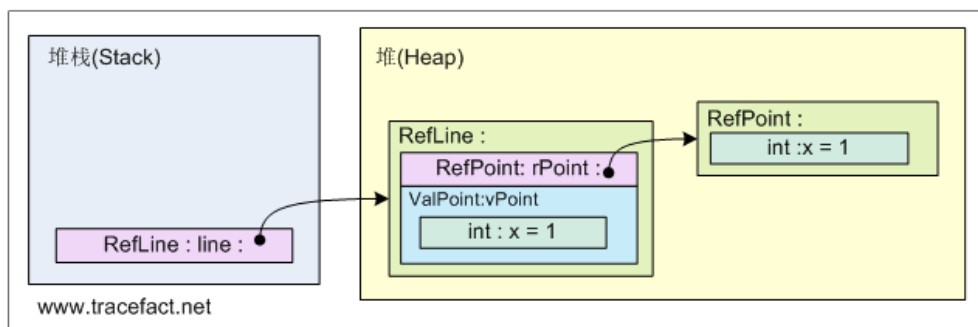
继续我们上面的例子，如果我们想要进行复制的对象(RefLine)是这样定义的，(为了避免look up，我在这里把代码再贴过来)：

```
// 将要进行 浅度复制 的对象，注意为 引用类型
public class RefLine {
    public RefPoint rPoint;
    public ValPoint vPoint;
    public Line(RefPoint rPoint,ValPoint vPoint){
        this.rPoint = rPoint;
        this.vPoint = vPoint;
    }
}
// 定义一个引用类型成员
public class RefPoint {
    public int x;
    public RefPoint(int x) {
        this.x = x;
    }
}
// 定义一个值类型成员
public struct ValPoint {
    public int x;
    public ValPoint(int x) {
        this.x = x;
    }
}
```

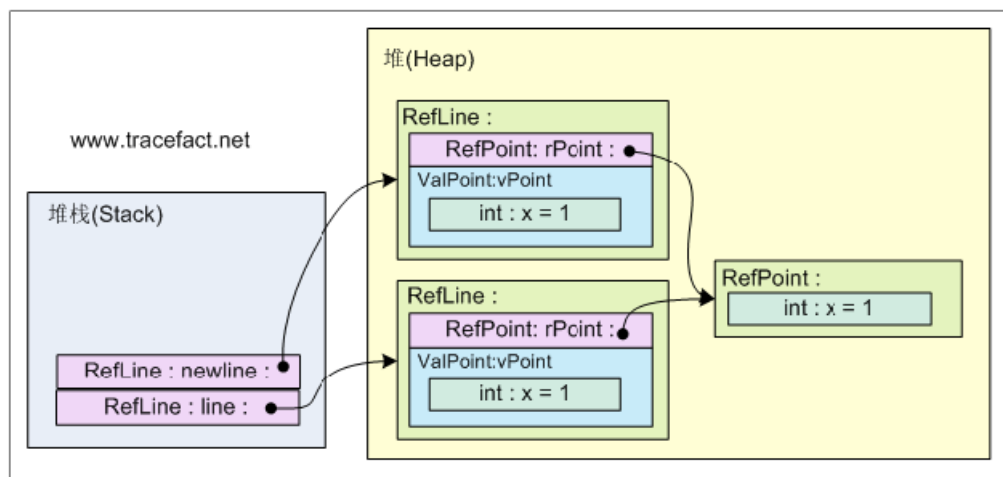
我们先创建一个想要复制的对象：

```
RefPoint rPoint = new RefPoint(1);
ValPoint vPoint = new ValPoint(1);
RefLine line = new RefLine(rPoint, vPoint);
```

它所产生的实际效果是(堆栈上仅考虑 line 部分):



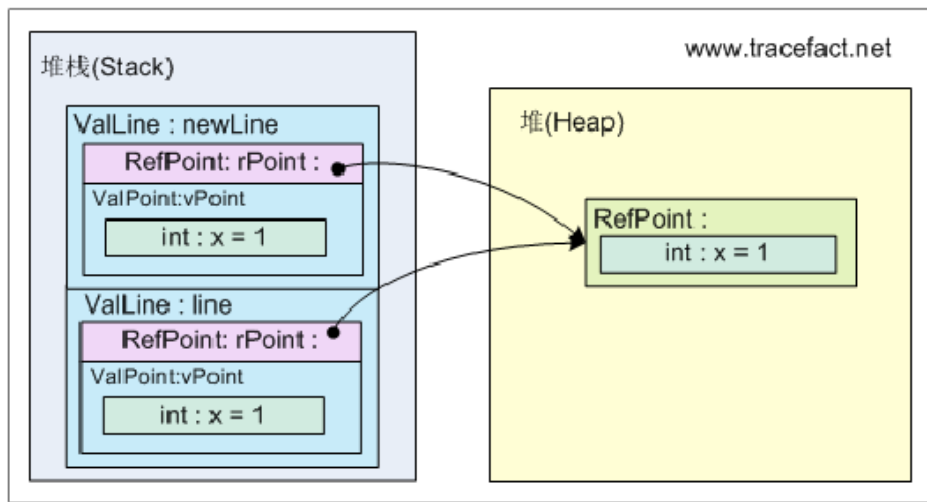
那么当我们对它复制时, 就会像这样(newLine 是指向新拷贝的对象的指针, 在代码中体现为一个引用类型的变量):



按照这个定义, 再回忆上面我们讲到的内容, 可以推出这样一个结论: 当复制一个结构类型成员的时候, 直接创建一个新的结构类型变量, 然后对它赋值, 就相当于进行了一个浅度复制, 也可以认为结构类型隐式地实现了浅度复制。如果我们将上面的 RefLine 定义为一个结构 (Struct), 结构类型叫 ValLine, 而不是一个类, 那么对它进行浅度复制就可以这样:

```
ValLine newLine = line;
```

实际的效果图是这样:



现在你已经已经搞清楚什么是浅度复制，知道了如何对结构浅度复制。那么如何对一个引用类型实现浅度复制呢？在 .Net Framework 中，有一个 `ICloneable` 接口，我们可以实现这个接口来进行浅度复制(也可以是深度复制，这里有争议，国外一些人认为 `ICloneable` 应该被标识为过时(Obsolete)的，并且提供 `IShallowCloneable` 和 `IDeepCloneable` 来替代)。这个接口只要求实现一个方法 `Clone()`，它返回当前对象的副本。我们并不需要自己实现这个方法(当然完全可以)，在 `System.Object` 基类中，有一个保护的 `MemberwiseClone()` 方法，它便用于进行浅度复制。所以，对于引用类型，如果想要实现浅度复制时，只需要调用这个方法就可以了：

```
public object Clone() {
    return MemberwiseClone();
}
```

现在我们来做一个测试：

```
class Program {
    static void Main(string[] args) {

        RefPoint rPoint = new RefPoint(1);
        ValPoint vPoint = new ValPoint(1);
        RefLine line = new RefLine(rPoint, vPoint);

        RefLine newLine = (RefLine)line.Clone();
        Console.WriteLine("Original: line.rPoint.x = {0}, line.vPoint.x = {1}",
            line.rPoint.x, line.vPoint.x);
        Console.WriteLine("Cloned: newLine.rPoint.x = {0}, newLine.vPoint.x = {1}",
            newLine.rPoint.x, newLine.vPoint.x);

        line.rPoint.x = 10;    // 修改原先的 line 的 引用类型成员 rPoint
        line.vPoint.x = 10;    // 修改原先的 line 的 值类型 成员 vPoint
        Console.WriteLine("Original: line.rPoint.x = {0}, line.vPoint.x = {1}",
            line.rPoint.x, line.vPoint.x);
    }
}
```

```
        Console.WriteLine("Cloned: newLine.rPoint.x = {0}, newLine.vPoint.x = {1}",
newLine.rPoint.x, newLine.vPoint.x);

    }
}
```

输出为:

```
Original: line.rPoint.x = 1, line.vPoint.x = 1
Cloned: newLine.rPoint.x = 1, newLine.vPoint.x = 1
Original: line.rPoint.x = 10, line.vPoint.x = 10
Cloned: newLine.rPoint.x = 10, newLine.vPoint.x = 1
```

可见,复制后的对象和原先对象成了连体婴,它们的引用成员字段依然引用堆上的同一个对象。

2. 深度复制

其实到现在你可能已经想到什么时深度复制了,深度复制就是将引用成员指向的对象也进行复制。实际的过程是创建新的引用成员指向的对象,然后复制对象包含的数据。

深度复制可能会变得非常复杂,因为引用成员指向的对象可能包含另一个引用类型成员,最简单的例子就是一个线性链表。

如果一个对象的成员包含了对于线性链表结构的一个引用,浅度复制 只复制了对头结点的引用,深度复制 则会复制链表本身,并复制每个结点上的数据。

考虑我们之前的例子,如果我们期望进行一个深度复制,我们的 Clone() 方法应该如何实现呢?

```
public object Clone(){           // 深度复制
    RefPoint rPoint = new RefPoint();           // 对于引用类型,创建新对象
    rPoint.x = this.rPoint.x;                 // 复制当前引用类型成员的值 到 新对象
    ValPoint vPoint = this.vPoint;           // 值类型,直接赋值
    RefLine newLine = new RefLine(rPoint, vPoint);
    return newLine;
}
```

可以看到,如果每个对象都要这样去进行深度复制的话就太麻烦了,我们可以利用串行化/反串行化来对对象进行深度复制:先把对象串行化(Serialize)到内存中,然后再进行反串行化,通过这种方式来进行对象的深度复制:

```
public object Clone() {
    BinaryFormatter bf = new BinaryFormatter();
```

```
MemoryStream ms = new MemoryStream();
bf.Serialize(ms, this);
ms.Position = 0;

return (bf.Deserialize(ms)); ;
}
```

我们来做一个测试:

```
class Program {
    static void Main(string[] args) {
        RefPoint rPoint = new RefPoint(1);
        ValPoint vPoint = new ValPoint(2);

        RefLine line = new RefLine(rPoint, vPoint);
        RefLine newLine = (RefLine)line.Clone();

        Console.WriteLine("Original line.rPoint.x = {0}", line.rPoint.x);
        Console.WriteLine("Cloned newLine.rPoint.x = {0}", newLine.rPoint.x);

        line.rPoint.x = 10; // 改变原对象 引用成员 的值
        Console.WriteLine("Original line.rPoint.x = {0}", line.rPoint.x);
        Console.WriteLine("Cloned newLine.rPoint.x = {0}", newLine.rPoint.x);
    }
}
```

输出为:

```
Original line.rPoint.x = 1
Cloned newLine.rPoint.x = 1
Original line.rPoint.x = 10
Cloned newLine.rPoint.x = 1
```

可见,两个对象的引用成员已经分离,改变原对象的引用对象的值,并不影响复制后的对象。

这里需要注意:如果想将对象进行序列化,那么对象本身,及其所有的自定义成员(类、结构),都必须使用 `Serializable` 特性进行标记。所以,如果想让上面的代码运行,我们之前定义的类都需要进行这样的标记:

```
[Serializable()]
public class RefPoint { /*略*/ }
```

NOTE: 关于特性(Attribute),可以参考 [.Net 中的反射\(反射特性\)](#) 一文。

总结

本文简单地对 C# 中的类型作了一个回顾。

我们首先讨论了 C# 中的两种类型——值类型和引用类型，随后简要回顾了 装箱/拆箱 操作。接着，详细讨论了 C# 中的对象判等。最后，我们讨论了浅度复制 和 深度复制，并比较了它们之间不同。

希望这篇文章能给你带来帮助！