

# [翻译]深入探察相等操作符

疯狂代码 <http://CrazyCoder.cn/> <http://CrazyCoder.cn/Translate/Article1783.html>

转自infoq阅读英文原文：A Detailed look at Overriding the Equality Operator重写相等操作符是非常容易出错的。不仅因为相等操作符有许多内涵，而且目前有很多指导文档有瑕疵，甚至在MSDN网站上有些指导文档也有瑕疵。我们将分别对支持相等操作的引用类型和值类型给出系统的分析，来澄清事实。为了清晰起见，这里将类称作引用类型而结构称作值类型。通常在结构中操作符重载比在类中有意义，所以我们先来展示在结构中的情况。类和结构的主要区别是，类需要检查空值，而在结构中你需要意识到可能存在的类型装箱。这一点将在后面说明。类签名结构的签名是直接了当的。你仅仅需要用System.IEquatable接口来标识该结构。请注意这个接口没有非泛型的版本，泛型的版本的角色由基础类Object承担。

C#

```
struct PointStruct : System.IEquatable
```

VB

```
Public Structure PointStruct
```

Implements IEquatable(Of PointStruct)类的签名本质上和结构签名是一样的。类的继承会破坏相等性，这会造成问题。如果a是一个基类而b是一个重写了Equals方法的子类，那么 a.Equals(b)会返回与b.Equals(a)不同的返回值。后面我们通过封闭（sealing）的Equals方法来解决这个问题。

C#

```
class PointClass : System.IEquatable
```

VB

```
Public Class PointClass Implements IEquatable(Of PointClass)
```

成员变量和属性任何用于相等性比较的成员变量必须是不可变的。通常，这意味着类中所有的属性是只读的或者类有一个类似于数据库主键的唯一标识符。在使用任何依赖哈希的东西的时候这条规则都是至关重要的。这样的例子包括Hashtable、Dictionary、HashSet和 KeyedCollection。这些类都使用哈希码作查找和存储。如果对象的哈希码变化了，它会被放在错误的槽中而且集合不能再正确的工作。最常见的故障是不能找到以前放在集合中的对象。为了确保成员变量是不可变的，它们被标记为只读的。由于成员变量可以在构造器中设置，所以改变成员变量有点象写错名字。但是一旦初始化完成了，没有方法可以直接改变成员变量的值。

C#

```
    readonly int _X;
    readonly int _Y;

    public PointStruct (int x, int y)
    {
        _X = x;
        _Y = y;
    }
```

```
int X
{
    get { return _X; }
}
```

```
int Y
{
    get { return _Y; }
}
```

VB

```
Private ReadOnly m_X As Integer
Private ReadOnly m_Y As Integer
```

```
Public Sub New(ByVal x As Integer, ByVal y As Integer)
    m_X = x
    m_Y = y
End Sub
```

```
Public ReadOnly Property X() As Integer
    Get
        Return m_X
    End Get
End Property
```

```
Public ReadOnly Property Y() As Integer
    Get
        Return m_Y
    End Get
```

End Property由于类版本的代码与上面的代码几乎是相同的，且在VB中是完全相同的，所以这里不给出类版本代码。类型安全的相等方法我们实现的第一个方法是类型安全的相等方法，在IEquatable接口中使用。

C#

```
public bool Equals(PointStruct other)
{
    return (this._X == other._X) && (this._Y == other._Y);
}
```

```
VB Public Overloads Function Equals(ByVal other As PointStruct) As Boolean _  
    Implements System.IEquatable(Of PointStruct).Equals  
    Return m_X = other.m_X AndAlso m_Y = other.m_Y
```

End Function 对于类，需要额外检查空值。按照惯例，所有非空值被认为与空值不相等。你会注意到我们没有使用地道的C#代码来检查空值。这是由于C#和VB处理相等性的方式有一处不同。Visual Basic在处理引用相等和值相等上有明确的区别。前者使用Is操作符，后者使用=操作符。C#缺乏这种区别，对两者都使用==操作符。由于我们会重写==操作符，所以不想使用==，不得不转而使用一个后门。这个后门是Object.ReferenceEquals方法。由于类总是与自己相等，所以在进行潜在的更昂贵的相等性检查之前，我们首先作这个检查。在下面代码中我们比较了私有成员变量，也可以使用属性来作比较。

```
C#  
public bool Equals(PointClass other)  
{  
    if (Object.ReferenceEquals(other, null))  
    {  
        return false;  
    }  
    if (Object.ReferenceEquals(other, this))  
    {  
        return true;  
    }  
    return (this._X == other._X) && (this._Y == other._Y);  
}
```

```
VB  
Public Overloads Function Equals(ByVal other As PointClass) As Boolean  
Implements System.IEquatable(Of PointClass).Equals  
    If other Is Nothing Then Return False  
    If other Is Me Then Return True  
    Return m_X = other.m_X AndAlso m_Y = other.m_Y
```

End Function 哈希码下一步是产生哈希码。最简单的方法是将所有用于相等性比较的成员变量的哈希码作异或运算。

```
C#  
public override int GetHashCode()  
{  
    return _X.GetHashCode() ^ _Y.GetHashCode();  
}
```

```
}
```

VB

```
Public Overrides Function GetHashCode() As Integer  
    Return m_X.GetHashCode Xor m_Y.GetHashCode
```

End Function如果你确实决定要从头写自己的哈希码，你必须确保对于一套给定的值，你总是能返回相同的哈希码。换言之，如果a等于b，那么它们的哈希码也相等。哈希码不必是唯一的，不同的值可以有相同的哈希码。但是它们应该有一个良好的分布。对于每一个哈希码都返回42在技术上是合法的，但是任何使用该算法的应用在性能上都会很糟糕。哈希码应该以非常快的速度计算出来。由于计算哈希值可能成为瓶颈，所以宁可选用一个快速的有合理良好分布的哈希码算法，而不选择一个慢的，复杂的有着完美的均匀分布的算法。相等（对象）重写基类的Equals方法是一个基础工作，该方法被Object.Equals(Object, Object)函数和其他方法调用。你应该注意到由于类型转换做了两次，所以可能存在一点性能问题：一次是看它是否有效，第二次是真正的执行它。不幸的是，在结构中是无法避免这样作的。

C#

```
public override bool Equals(object obj)  
{  
    if (obj is PointStruct)  
    {  
        return this.Equals((PointStruct)obj);  
    }  
    return false;  
}
```

VB

```
Public Overrides Function Equals(ByVal obj As Object) As Boolean  
    If TypeOf obj Is PointStruct Then Return CType(obj, PointStruct) = Me
```

End Function对于类可以只用一次类型转换。在处理步骤中，我们可以早点检测空值然后跳过对Equals(PointClass)方法的调用。C#必须用ReferenceEquals函数来检查空值。为了防止子类破坏相等性，我们封闭（Seal）了方法。

C#

```
public sealed override bool Equals(object obj)  
{  
    var temp = obj as PointClass;  
    if (!Object.ReferenceEquals(temp, null))  
    {  
        return this.Equals(temp);  
    }  
}
```

```
    }  
    return false;  
}
```

VB

```
Public NotOverridable Overrides Function Equals(ByVal obj As Object) As Boolean  
    Dim temp = TryCast(obj, PointClass)  
    If temp IsNot Nothing Then Return Me.Equals(temp)  
End Function
```

操作符重载所有的难题都被攻克了，我们现在可以进行操作符的重写了。这里和调用类型安全的Equals方法一样简单。

C#

```
public static bool operator ==(PointStruct point1 PointStruct point2)  
{  
    return point1.Equals(point2);  
}  
  
public static bool operator !=(PointStruct point1, PointStruct point2)  
{  
    return !(point1 == point2);  
}
```

VB

```
Public Shared Operator =(ByVal point1 As PointStruct, ByVal point2 As PointStruct) As Boolean  
    Return point1.Equals(point2)  
End Operator  
  
Public Shared Operator <>(ByVal point1 As PointStruct,  
    ByVal point2 As PointStruct) As Boolean  
    Return Not (point1 = point2)  
End Operator
```

对于类，需要检查空值。幸运的是，Object.Equals(object, object)为你处理了这种情况。然后调用已经被重写的Object.Equals(Object)方法。

C#

```
public static bool operator ==(PointClass point1, PointClass point2)  
{  
    return Object.Equals(point1, point2);  
}
```

```
public static bool operator !=(PointClass point1, PointClass point2)
{
    return !(point1 == point2);
}
```

VB

```
Public Shared Operator =(ByVal point1 As PointClass, ByVal point2 As PointClass) As Boolean
    Return Object.Equals(point1 ,point2)
End Operator
```

```
Public Shared Operator <>(ByVal point1 As PointClass, ByVal point2 As PointClass) As Boolean
    Return Not (point1 = point2)
```

End Operator

性能你会注意到每个调用链都有点长，尤其是不相等操作符。如果需要考虑性能问题，你可以分别在每个方法中实现比较逻辑来提高速度。这样很容易出错而且使得维护工作比较棘手，所以仅仅当你使用性能检测工具证明了必须这样作之后，才应该这样作。测试本文使用了下面的测试。它使用了列表的形式使得你可以方便的将它们翻译到你最喜欢的单元测试框架中。因为相等性很容易被破坏，所以这是单元测试的首要的测试目标。请注意测试不是全面的。你应该测试左右值部分相等的情况，例如PointStruct(1, 2) and PointStruct(1, 5)。

;

; 2008-6-13 13:15:24  
疯狂代码 <http://CrazyCoder.cn/>