C++程序员转向C#时的十大陷阱

C# 和 C++的语法很相像, 但很多语法上的改变是小而琐细的。有一些改变衬于粗心的C++ 程序员来说是潜在的陷阱。本文将集中阐述十个最大的危险。

关键词: C++, C#, 陷阱

【译序：C#入门文章。请注意：所有程序调试环境为Microsoft Visual Studio.NET 7.0 Beta2和 Microsoft .net framework SDK Beta2。限于译者时间和能力，文中倘有讹误，当以英文原版为准】

在最近发表于《MSDN Magazine》（2001年7月刊）上的一篇文章里，我讲了“从C++转移到C#，你应该了解些什么？”。【译注：这篇文章的中文版可查阅《程序员》杂志.NET专刊】。在那篇文章里，我说过C#和C++的语法很象，转移过程中的困难并非来自语言自身，而是对受管制的.NET环境的适应和对庞大的.NET框架的理解。

我已经编辑了一个C++和C#语法不同点的列表（可在我的web站点上找到这个列表。在站点上，点击Books可以浏览《Programming C#》，也可以点击FAQ看看）。正如你所意料的，很多语法上的改变是小而琐细的。有一些改变对于粗心的C++程序员来说是潜在的陷阱，本文将集中阐述十个最大的危险。

我已经编辑了一个C++和C#语法不同点的列表（可在 [我的web站点](https://cloud.tencent.com/developer/tools/blog-entry?target=http%3A%2F%2Fwww.libertyassociates.com%2F&objectId=1796883&objectType=1)上找到这个列表。在站点上，点击Books可以浏览《Programming C#》，也可以点击FAQ看看）。正如你所意料的，很多语法上的改变是小而琐细的。有一些改变对于粗心的C++程序员来说是潜在的陷阱，本文将集中阐述十个最大的危险。

## 陷阱1.非确定终结和C#析构器

理所当然，对于大多数C++程序员来说，C#中最大的不同是垃圾收集。这就意味着你不必再担心内存泄漏以及确保删除指针对象的问题。当然，你也就失去了对何时销毁对象的精确控制。实际上，C#中没有显式的析构器。

如果你在处理一个未受管制的资源，当你用完时，你需要显式地释放那些资源。资源的隐式控制可通过提供一个Finalize方法（称为终结器），当对象被销毁时，它将被垃圾收集器调用。

终结器只应该释放对象携带的未受管制的资源，而且也不应该引用别的对象。注意：如果你只有一些受管制的对象引用那你用不着也不应该实现Finalize方法—它仅在需处理未受管制的资源时使用。因为使用终结器要付出代价，所以，你只应该在需要的方法上实现（也就是说，在使用代价昂贵的、未受管制的资源的方法上实现）。

永远不要直接调用Finalize方法（除了在你自己类的Finalize里调用基类的Finalize方法外），垃圾收集器会帮你调用它。

C#的析构器在句法上酷似C++的析构器，但它们本质不同。C#析构器仅仅是声明Finalize方法并链锁到其基类的一个捷径【译注：这句话的意思是，当一个对象被销毁时，从最派生层次的最底层到最顶层，析构器将依次被调用，请参见后面给出的完整例子】。因此，以下写法：

~MyClass()

{

//do work here

}

和如下写法具有同样效果：

MyClass.Finalize()

{

// do work here

base.Finalize();//

}

【译注：上面这段代码显然是错误的，首先应该写为：

class MyClass

{

void Finalize()

{

// do work here

base.Finalize(); //这样也不对！编译器会告诉你不能直接调用基类的Finalize方法，它将从析构函数中自动调用。关于原因，请参见本小节后面的例子和陷阱二的有关译注！

}

}

下面给出一个完整的例子：

using System;

class RyTestParCls

{

~RyTestParCls()

{

Console.WriteLine("RyTestParCls's Destructor");

}

}

class RyTestChldCls: RyTestParCls

{

~RyTestChldCls()

{

Console.WriteLine("RyTestChldCls's Destructor");

}

}

public class RyTestDstrcApp

{

public static void Main()

{

RyTestChldCls rtcc = new RyTestChldCls();

rtcc = null;

GC.Collect();//强制垃圾收集

GC.WaitForPendingFinalizers();//挂起当前线程，直至处理终结器队列的线程清空了该队列

Console.WriteLine("GC Completed!");

}

}

以上程序输出结果为：

RyTestChldCls's Destructor

RyTestParCls's Destructor

GC Completed!

**注意：**在CLR中，是通过重载System.object的虚方法Finalize()来实现虚方法的，在C#中，不允许重载该方法或直接调用它，如下写法是错误的：

class RyTestFinalClass

{

override protected void Finalize() {}//错误！不要重载System.Object方法。

}

同样，如下写法也是错误的：

class RyTestFinalClass

{

public void SelfFinalize() //注意！这个名字是自己取的，不是Finalize

{

this.Finalize()//错误！不能直接调用Finalize()

base.Finalize()//错误！不能直接调用基类Finalize()

}

}

class RyTestFinalClass

{

protected void Finalize() //注意！这个名字和上面不一样，同时，它也不是override的，这是可以的，这样，你就隐藏了基类的Finalize。

{

this.Finalize()//自己调自己，当然可以了，但这是个递归调用你想要的吗？J

base.Finalize()//错误！不能直接调用基类Finalize()

}

}

对这个主题的完整理解请参照陷阱二。】

## 陷阱2.Finalize 和 Dispose

显式调用终结器是非法的，Finalize方法应该由垃圾收集器调用。如果是处理有限的、未受管制的资源（比如文件句柄），你或许想尽可能快地关闭和释放它，那你应该实现IDisposable接口。这个接口有一个Dispose方法，由它执行清除动作。类的客户负责显式调用该Dispose方法。Dispose方式等于是你的客户说“不要等Finalize了，现在就干吧！”。

如果你提供了Dispose方法，你应该禁止垃圾收集器调用对象的Finalize方法—既然要显式进行清除了。为了做到这一点，你应该调用静态方法GC.Suppressfinalize，并传入对象的this指针，你的Finalize方法就能够调用你的Dispose方法。

你可能会这么写：

public void Dispose()

{

// 执行清除动作

// 告诉垃圾收集器不要调用Finalize

GC.SuppressFinalize(this);

}

public override void Finalize()

{

Dispose();

base.Finalize();

}

【译注：以上这段代码是有问题的，请参照我在陷阱一中给的例子。微软站点上有一篇很不错的文章（“SCOl/html/deepc10192000.asp">Gozer the Destructor），说法和这儿基本一致，但其代码示例在Microsoft Visual Studio.NET 7.0 Beta2和 Microsoft .NET Framework SDK Beta2都过不了，由于手头没有Beta1比对，所以，现在还不能确定是文章的笔误，还是因为Beta1和Beta2的不同而导致。比如下面这个例子（来自[Gozer the Destructor](https://cloud.tencent.com/developer/tools/blog-entry?target=http%3A%2F%2Fmsdn.microsoft.com%2Flibrary%2Fdefault.asp%3Furl%3D%2Flibrary%2Fen-us%2Fdncscol%2Fhtml%2Fdeepc10192000.asp&objectId=1796883&objectType=1)）在Beta2环境下无法通过：

class X

{

public X(int n)

{

this.n = n;

}

~X()

{

System.Console.WriteLine("~X() {0}", n);

}

public void Dispose()

{

Finalize();//此行代码在Beta2环境中出错！编译器提示，不能调用Finalize，可考虑调用Idisposable.Dispose（如可用）

System.GC.SuppressFinalize(this);

}

private int n;

};

class main

{

static void f()

{

X x1 = new X(1);

X x2 = new X(2);

x1.Dispose();

}

static void Main()

{

f();

System.GC.Collect();

System.GC.WaitForPendingFinalizers();

}

};

而在该文章里，则声称会有如下输出：

~X() 1

~X() 2

why？

对于某些对象来说，你可能宁愿让你的客户调用Close方法（例如，对于文件对象来说，Close比Dispose更有意义）。那你可以通过创建一个private的Dispose方法和一个public的Close方法，并且在Close里调用Dispose。

因为你并不能肯定客户将调用Dispose，并且终结器是不确定的（你无法控制什么时候运行GC），C#提供了using语句以确保尽可能早地调用Dispose。这个语句用于声明你正在使用什么对象，并且用花括号为这些对象创建一个作用域。当到达“}”J时，对象的Dispose方法将被自动调用：

using System.Drawing;

class Tester

{

public static void Main()

{

using (Font theFont = new Font("Arial", 10.0f))

{

// 使用theFont

} // 编译器为theFont调用Dispose

Font anotherFont = new Font("Courier",12.0f);

using (anotherFont)

{

// 使用 anotherFont

} // 编译器为anotherFont调用Dispose

}

}

在上例的第一部份，theFont对象在using语句内创建。当using语句的作用域结束，theFont对象的Dispose方法被调用。例子第二部份，在using语句外创建了一个anotherFont对象，当你决定使用anotherFont对象时，可将其放在using语句内，当到达using语句的作用域尾部时，对象的Dispose方法同样被调用。

using 语句还可保护你处理未曾意料的异常，不管控制是如何离开using语句的，Dispose都会被调用，就好像那儿有个隐式的try-catch-finally程序块。

## 陷阱3.C#区分值类型和引用类型

和C++一样，C#是一个强类型的语言。并且象C++一样，C#把类型划分为两类：语言提供的固有（内建）类型和程序员定义的用户定义类型【译注：即所谓的UDT】。

除了区分固有类型和用户自定义类型外，C#还区分值类型和引用类型。就象C++里的变量一样，值类型在栈上保存值，除非是嵌在对象中的值类型。引用类型变量本身位于栈上，但它们所指向的对象则位于堆上，这很象C++里的指针【译注：这其实更象C++里的引用J】。当被传递给方法时，值类型是传值（做了一个拷贝）而引用类型则按引用高效传递。

类和接口创建引用类型，但要谨记（参见陷阱五）：和所有固有类型一样，结构也是值类型。

【译注：可参见陷阱五的例子】

## 陷阱4.警惕隐式装箱

装箱和拆箱是使值类型（如整型等）能够象引用类型一样被处理的过程。值被装箱进一个对象，随后的拆箱则是将其还原为值类型。C#里的每一种类型包括固有类型都是从object派生下来并可以被隐式转换为object。装箱一个值相当于创建一个object的实例，并将该值拷贝入该对象。

装箱是隐式进行的，因此，当需要一个引用类型而你提供了一个值类型时，该值将会被隐式装箱。装箱带来了一些执行负担，因此，要尽可能地避免装箱，特别是在一个大的集合里。

如果要把被装箱的对象转换回值类型，必须将其显式拆箱。拆箱动作分为两步：首先检查对象实例以确保它是一个将被转换的值类型的装箱对象，如果是，则将值从该实例拷贝入目标值类型变量。若想成功拆箱，被拆箱的对象必须是目标值类型的装箱对象的引用。

using System;

public class UnboxingTest

{

public static void Main()

{

int i = 123;

//装箱

object o = i;

// 拆箱 (必须显式进行)

int j = (int) o;

Console.WriteLine("j: {0}", j);

}

}

如果被拆箱的对象为null或是一个不同于目标类型的装箱对象的引用，那将抛出一个InvalidCastException异常。【译注：此处说法有误，如果正被拆箱的对象为null，将抛出一个System.NullReferenceException而不是System.InvalidCastExcepiton】

【译注：关于这个问题，我在另一篇译文（[A Comparative Overview of C#中文版(上篇)](https://cloud.tencent.com/developer/tools/blog-entry?target=http%3A%2F%2Fwww.csdn.net%2FDevelop%2FArticle%2F11%2F11685.shtm&objectId=1796883&objectType=1)）里有更精彩的描述J】

## 陷阱5.C#中结构是大不相同的

C++中的结构几乎和类差不多。在C++中，唯一的区别是结构【译注：指成员】缺省来说具有public访问（而不是private）级别并且继承缺省也是public（同样，不是private）的。有些C++程序员把结构当成只有数据成员的对象，但这并不是语言本身支持的约定，而且这种做法也是很多OO设计者所不鼓励的。

在C#中，结构是一个简单的用户自定义类型，一个非常不同于类的轻量级的可选物。尽管结构支持属性、方法、字段和操作符，但结构并不支持继承或析构器之类的东西。

更重要的是，类是引用类型，而结构是值类型（参见陷阱三）。因此，结构对表现不需要引用语义的对象就非常有用。在数组中使用结构，在内存上会更有效率些，但若用在集合里，就不是那么有效率了。集合需要引用类型，因此，若在集合中使用结构，它就必须被装箱（参见陷阱四），而装箱和拆箱需要额外的负担，因此，在大的集合里，类可能会更有效。

【译注：下面是一个完整的例子，它同时还演示了隐式类型转换，请观察一下程序及其运行结果J

using System;

class RyTestCls

{

public RyTestCls(int AInt)

{

this.IntField = AInt;

}

public static implicit operator RyTestCls(RyTestStt rts)

{

return new RyTestCls(rts.IntField);

}

private int IntField;

public int IntProperty

{

get

{

return this.IntField;

}

set

{

this.IntField = value;

}

}

}

struct RyTestStt

{

public RyTestStt(int AInt)

{

this.IntField = AInt;

}

public int IntField;

}

class RyClsSttTestApp

{

public static void ProcessCls(RyTestCls rtc)

{

rtc.IntProperty = 100;

}

public static void ProcessStt(RyTestStt rts)

{

rts.IntField = 100;

}

public static void Main()

{

RyTestCls rtc = new RyTestCls(0);

rtc.IntProperty = 200;

ProcessCls(rtc);

Console.WriteLine("rtc.IntProperty = {0}", rtc.IntProperty);

RyTestStt rts = new RyTestStt(0);

rts.IntField = 200;

ProcessStt(rts);

Console.WriteLine("rts.IntField = {0}", rts.IntField);

RyTestStt rts2= new RyTestStt(0);

rts2.IntField = 200;

ProcessCls(rts2);

Console.WriteLine("rts2.IntField = {0}", rts2.IntField);

}

}

以上程序运行结果为：

rtc.IntProperty = 100

rtc.IntField = 200

rts2.IntField = 200

## 陷阱6.虚方法必须被显式重载

在C#中，如果程序员决定重载一个虚方法，他（她）必须显式使用overr[ide](https://cloud.tencent.com/developer/tools/blog-entry?target=http%3A%2F%2Fblog.itpub.net%2F10797429%2Fviewspace-101630%2Ftag-351-1.html&objectId=1796883&objectType=1)关键字。

让我们考察一下这样做的好处。假定公司A写了一个Window类，公司B购买了公司A的Window类的一个拷贝作为基类。公司B的程序员从中派生【译注：原文为...using...，从下文来看，显然是“派生”之意。事实上，使用类的方式还有“组合”（也有说为“嵌入”或“包容”（COM语义）等等），后者不存在下文所描述的问题】出ListBox和RadioButton类。公司B的程序员不知道或不能控制Window类的设计，包括公司A将来对Window类可能做的修改。

现在假定公司B的程序员决定为ListBox加入一个Sort方法：

public class ListBox : Window

{

public virtual void Sort() {"}

}

这是没有问题的—直到公司A的Window类作者发布了Window类的版本2，公司A的程序员向Window类也加入了一个public的Sort方法：

public class Window

{

//

public virtual void Sort() {"}

}

在C++中，Window类新的虚方法Sort将会作为ListBox虚方法的基类方法。当你试图调用Window的Sort时，实际上调用的是ListBox的Sort。C#中虚方法【译注：原文写成virtual function】永远被认为是虚拟调度的根。这就是说，只要C#找到了一个虚方法，它就不会再沿着继承层次进一步寻找了，如果一个新的Sort虚方法被引入Window，ListBox的运行时行为不会被改变。当ListBox再次被编译时，编译器会发出如下警告：

"class1.cs(54,24): warning CS0114: 'ListBox.Sort()' hides inherited member 'Window.Sort()'.

如果要使当前成员重载实现，可加入override关键字。否则，加上new关键字。

如果想要移去这个警告，程序员必须明确指明他的意图。可以将ListBox的Sort方法标为new，以指明它不是对Window的虚方法的重载：

public class ListBox : Window

{

public new virtual void Sort() {"}

}

这样编译器就不会再警告。另一方面，如果程序员想重载Window的方法，只要显式加上override关键字即可。

## 陷阱7：不可以在头部进行初始化

C#里的初始化不同于C++。假定你有一个类Person，它有一个私有成员变量age；一个派生类Employee，它有一个私有成员变量salaryLeverl。在C++中，你可以在Employee构造器的成员初始化列表部分初始化salaryLevel：

Employee::Employee(int theAge, int theSalaryLevel):

Person(theAge) // 初始化基类

salaryLevel(theSalaryLevel) // 初始化成员变量

{

// 构造器体

}

在C#中，这个构造器是非法的。尽管你仍可以如此初始化基类，但对成员变量的初始化将导致一个编译时错误。当然，你可以在成员变量声明处对其赋初始值：

Class Employee : public Person

{

// 在这儿声明

private salaryLevel = 3; //初始化

}

【译注：以上代码有误。C#中，正确写法如下：

class Employee: Person

{

private int salaryLevel = 3;

}

】

注意：你不需要在每一个类声明的后面都加上一个分号，每一个成员都必须要有显式的访问级别声明。

## 陷阱8.不能把布尔值转换为整型值

在C#中，布尔值（true、false）不同于整型值。因此，不能这么写：

if(someFuncWhichReturnsAValue() )//【译注：假定这个方法不返回布尔值】

也不能指望如果someFuncWhichReturnsAValue返回一个0它将等于false，否则为true。一个好消息是误用赋值操作符而不是相等操作符的老毛病不会再犯了。因此，如果这么写：

if ( x = 5 )

将会得到一个编译时错误，因为x = 5的结果为5，而它不是布尔值。

【译注：以下是C++里一不小心会出的逻辑错误，编译器不会有任何提示L运行得很顺畅，不过结果不是你想要的：

**C++：**

代码语言：javascript

复制

#include "stdafx.h"

int main(int argc, char\* argv[])

{

int n = 0;

if (n = 1)//编译器啥都没说L一般推荐写为1 == n，万一写成1 = n编译器都不同意J

{

printf("1 ");

}

else

{

printf("0 ");

}

return 0;

}

以上运行结果为1，这未必是你想要的。

C#：

using System;

public class RyTestBoolApp

{

public static void Main()

{

int n = 0;

if (n = 1)//编译器不同意J无法将int转换成bool

{

Console.WriteLine("1");

}

else

{

Console.WriteLine("0");

}

}

}

但是，如果是这种情况：

bool b = false;

if (b = true)

...

不管是C++还是C#都没招L

】

【译注：C++程序员一般是喜欢这种自由的写法：

if (MyRef)

if (MyInt)

但在C#里，必须写成：

if (MyRef == null)//或if (null == MyRef)

if (MyInt == 0)//或if (0 == MyInt)

等。

】

## 陷阱9.switch语句不会“贯穿”

在C#中，case语句不会贯穿到下一句—如果在该case里有代码的话。因此，尽管下面的代码在C++里是合法的，但在C#里则不然：

switch (i)

{

case 4:

CallFuncOne();

case 5: // 错误，不可以贯穿

CallSomeFunc();

}

为了达到这个目的，你需要显式地使用goto语句：

switch (i)

{

case 4:

CallFuncOne();

goto case 5;

case 5:

CallSomeFunc();

}

如果case语句没做任何事（里面没有代码）你就可以贯穿：

switch (i)

{

case 4: // 可以贯穿

case 5: // 可以贯穿

case 6:

CallSomeFunc();

}

【译注：以下是使用switch的完整例子，它还说明了switch语句支配的类型可以是字符串，并演示了属性的使用方法。

using System;

class RySwitchTest

{

public RySwitchTest(string AStr)

{

this.StrProperty = AStr;

}

protected string StrField;

public string StrProperty

{

get

{

return this.StrField;

}

set

{

this.StrField = value;

}

}

public void SwitchStrProperty()

{

switch (this.StrProperty)

{

case ("ry01"):

Console.WriteLine("ry01");

break;

case ("ry02"):

Console.WriteLine("ry02");

break; //如果这一行注释掉，编译器会报控制不能从一个case标签（case "ry02":）贯穿到另一个标签，如果你确实需要，可以这么写：goto case ("ry03");或goto default。

case ("ry03"):

Console.WriteLine("ry03");

break;

default:

Console.WriteLine("default");

break;

}

}

}

class RySwitchTestApp

{

public static void Main()

{

RySwitchTest rst = new RySwitchTest("ry02");

rst.SwitchStrProperty();

}

}

】

## 陷阱10.C#需要明确的赋值操作

C#要求必须进行明确地赋值操作，这就意味所有的变量在被使用前必须被赋值。因此，尽管你可以声明未初始化的变量，但在它拥有值之前是不可以被传递到方法的。

这就引出了一个问题—若你仅仅想把变量按引用传递给方法，就象一个“out”参数。例如，假定你有个方法，返回当前的小时、分钟和秒。如果这么写：

int theHour;

int theMinute;

int theSecond;

timeobject.GetTime( ref theHour, ref theMinute, ref theSecond);

将会得到编译错误，因为在使用theHour、theMinute和theSecond前，它们没有被初始化：

Use of unassigned local variable 'theHour'

Use of unassigned local variable 'theMinute'

Use of unassigned local variable 'theSecond'

可以将它们初始化为0或者其它什么无伤大雅的值以让讨厌的编译器安静下来：

int theHour = 0;

int theMinute = 0;

int theSecond = 0;

timeObject.GetTime( ref theHour, ref theMinute, ref theSecond);

但是这种写法太愚蠢！我们的本意不过是想把这些变量按引用传递到GetTime，在其中改变它们的值。为了解决这个问题，C#提供了out参数修饰符。out修饰符避免了对引用参数也需要初始化的需求。例如，为GetTime提供的参数没有提供给方法任何信息，它们仅仅是要从方法里取得信息的机制。因此，把这三个都标记为out参数，就避免了在方法外初始化它们的需要。当从被传入的方法返回时，out参数必须被赋值。这儿是改变后的GetTime参数声明：

public void GetTime(out int h, out int m, out int s)

{

h = Hour;

m = Minute;

s = Second;

}

这儿是对GetTime方法的新的调用：

timeObject.GetTime( out theHour, out theMinute, out theSecond);

【译注：完整示例如下：

**C#：[例1：使用ref修饰的方法参数]**

using System;

class RyRefTest

{

public RyRefTest()

{

this.IntField = 1;

this.StrField = "StrField";

}

protected int IntField;

protected string StrField;

public void GetFields(ref int AInt, ref string AStr)

{

AInt = this.IntField;

AStr = this.StrField;

}

}

class RyRefTestApp

{

public static void Main()

{

RyRefTest rrt = new RyRefTest();

int IntVar = 0;//如果是int IntVar; 编译器会报使用了未赋值的变量IntVar

string StrVar = "0";//如果是string StrVar; 编译器会报使用了未赋值的变量StrVar

rrt.GetFields(ref IntVar, ref StrVar);

Console.WriteLine("IntVar = {0}, StrVar = {1}", IntVar, StrVar);

}

}

**C#：[例2：使用out修饰的方法参数]**

using System;

class RyRefTest

{

public RyRefTest()

{

this.IntField = 1;

this.StrField = "StrField";

}

protected int IntField;

protected string StrField;

public void GetFields(out int AInt, out string AStr)

{

AInt = this.IntField;

AStr = this.StrField;

}

}

class RyRefTestApp

{

public static void Main()

{

RyRefTest rrt = new RyRefTest();

int IntVar;//这样就可以了，如果写成int IntVar = 0;当然也没问题J

string StrVar; //这样就可以了，如果写成string StrVar = "0";当然也没问题J

rrt.GetFields(out IntVar, out StrVar);

Console.WriteLine("IntVar = {0}, StrVar = {1}", IntVar, StrVar);

}

}

【译注：如欲了解更多，请参阅[A Comparative Overview of C#中文版(上篇)](https://cloud.tencent.com/developer/tools/blog-entry?target=http%3A%2F%2Fwww.csdn.net%2FDevelop%2FArticle%2F11%2F11685.shtm&objectId=1796883&objectType=1)、[A Comparative Overview of C#中文版(下篇)](https://cloud.tencent.com/developer/tools/blog-entry?target=http%3A%2F%2Fwww.csdn.net%2FDevelop%2FArticle%2F11%2F11686.shtm&objectId=1796883&objectType=1)、[C#首席设计师Anders Hejlsberg专访](https://cloud.tencent.com/developer/tools/blog-entry?target=http%3A%2F%2Fwww.csdn.net%2Fdevelop%2Fread_article.asp%3Fid%3D11580&objectId=1796883&objectType=1)。】

**来源:** [https://web.archive.org/web/20030621085815/http://www.ondotnet.com/pub/a/dotnet/2002/02/11/csharp\_traps.html](https://cloud.tencent.com/developer/tools/blog-entry?target=https%3A%2F%2Fweb.archive.org%2Fweb%2F20030621085815%2Fhttp%3A%2F%2Fwww.ondotnet.com%2Fpub%2Fa%2Fdotnet%2F2002%2F02%2F11%2Fcsharp_traps.html&objectId=1796883&objectType=1)

[http://blog.sina.com.cn/s/blog\_4b3419a2010119j2.html](https://cloud.tencent.com/developer/tools/blog-entry?target=http%3A%2F%2Fblog.sina.com.cn%2Fs%2Fblog_4b3419a2010119j2.html&objectId=1796883&objectType=1)

[C#面试题](https://cloud.tencent.com/developer/article/2097317)

[面向对象编程](https://cloud.tencent.com/developer/tag/10705)[存储](https://cloud.tencent.com/developer/tag/10665)[.net](https://cloud.tencent.com/developer/tag/10180)[编程算法](https://cloud.tencent.com/developer/tag/10663)

[值类型包括简单类型、结构体类型和枚举类型，引用类型包括自定义类、数组、接口、委托等。 1、赋值方式：将一个值类型变量赋给另一个值类型变量时，将复制包含的值。这与引用类型变量的赋值不同，引用类型变量的赋值只复制对象的引用（即内存地址，类似C++中的指针），而不复制对象本身。 2、继承：值类型不可能派生出新的类型，所有的值类型均隐式派生自 System.ValueType。但与引用类型相同的是，结构也可以实现接口。 3、null：与引用类型不同，值类型不可能包含 null 值。然而，可空类型功能允许将 null 赋给值类型。 4、每种值类型均有一个隐式的默认构造函数来初始化该类型的默认值，值类型初始会默认为0，引用类型默认为null。 5、值类型存储在栈中，引用类型存储在托管堆中。](https://cloud.tencent.com/developer/article/2097317)

全栈程序员站长

2022/09/07

7730

[.NET基础拾遗（1）类型语法基础和内存管理基础](https://cloud.tencent.com/developer/article/1190703)

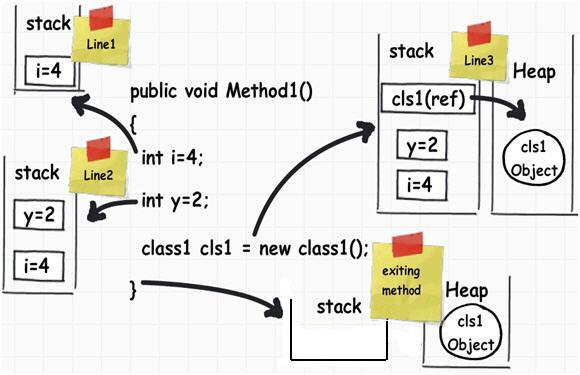
[其他](https://cloud.tencent.com/developer/tag/125)

[在.NET中所有的内建类型都继承自System.Object类型。在C#中，不需要显示地定义类型继承自System.Object，编译器将自动地自动地为类型添加上这个继承申明，以下两行代码的作用完全一致：](https://cloud.tencent.com/developer/article/1190703)

Edison Zhou

2018/08/20

6490



[[C#1] 11-接口](https://cloud.tencent.com/developer/article/1025185)

[数据库](https://cloud.tencent.com/developer/tag/10244)[c#](https://cloud.tencent.com/developer/tag/10171)

[接口与继承 CLR规定一个类型只能有一个基类型，这种继承成为单继承； 接口继承是指一个类型继承的是接口中的方法签名，而非方法实现，通常称为实现接口； 接口仅仅是含有一组虚方法的抽象类型，不含有任何实现。CLR允许接口包含静态方法、静态字段、常量、以及静态构造器， 但是CLS兼容的接口类型是不允许有任何静态成员的，因为一些编程语言不能定义或者访问它们。 C#语言就是如此，C#编译器不允许接口中有任何静态成员。 约定接口名称第一个字母是大写的I；接口可以多继承，实际上实现了多个接口的类型允许我们将它的对象看作这](https://cloud.tencent.com/developer/article/1025185)

[c#面试题汇总](https://cloud.tencent.com/developer/article/1341901)

[c#](https://cloud.tencent.com/developer/tag/10171)

[下面的参考解答只是帮助大家理解，不用背，面试题、笔试题千变万化，不要梦想着把题覆盖了，下面的题是供大家查漏补缺用的，真正的把这些题搞懂了，才能“以不变应万变”。回答问题的时候能联系做过项目的例子是最好的，有的问题后面我已经补充联系到项目中的对应的案例了。](https://cloud.tencent.com/developer/article/1341901)

进击的猫思

2018/09/18

5.1K0

[C# 学习笔记（10）—— 可空类型、匿名方法、迭代器](https://cloud.tencent.com/developer/article/2347878)

[c#](https://cloud.tencent.com/developer/tag/10171)[编译器](https://cloud.tencent.com/developer/tag/17191)[对象](https://cloud.tencent.com/developer/tag/17248)[接口](https://cloud.tencent.com/developer/tag/17329)[学习笔记](https://cloud.tencent.com/developer/tag/17534)

[泛型是 C# 2 众多特性的其中之一，C# 2 还提出了可空类型、匿名方法和迭代器三个重要特性](https://cloud.tencent.com/developer/article/2347878)

[[读书笔记]C#学习笔记三: C#类型详解..](https://cloud.tencent.com/developer/article/1129280)

[c#](https://cloud.tencent.com/developer/tag/10171)

[前言 这次分享的主要内容有五个, 分别是值类型和引用类型, 装箱与拆箱,常量与变量,运算符重载,static字段和static构造函数. 后期的分享会针对于C#2.0 3.0 4.0 等新特性进行. 再会有三篇博客  这个系列的就会结束了. 也算是自己对园子中@Learning Hard出版的<<C#学习笔记>>的一个总结了. 博客内容基本上都是白天抽空在公司写好的了, 但是由于公司内部网络不能登录博客园所以只能够夜晚拿回来修改,  写的不好或者不对的地方也请各位大神指出. 在下感激不尽了.  1，值类型和](https://cloud.tencent.com/developer/article/1129280)

[深入.NET平台和C#编程](https://cloud.tencent.com/developer/article/1347987)

[【深入浅出C#】章节10: 最佳实践和性能优化：内存管理和资源释放](https://cloud.tencent.com/developer/article/2327723)

[c#](https://cloud.tencent.com/developer/tag/10171)[内存](https://cloud.tencent.com/developer/tag/17382)[内存管理](https://cloud.tencent.com/developer/tag/17383)[性能优化](https://cloud.tencent.com/developer/tag/17528)[最佳实践](https://cloud.tencent.com/developer/tag/17607)

[C#对象池示例代码： 以下是一个简单的C#对象池示例，用于管理字符串对象。注意，这只是一个示例，实际应用中可以根据需要自定义更复杂的对象池。](https://cloud.tencent.com/developer/article/2327723)

喵叔

2023/09/11

1.3K0

[C#核心概念--装箱和拆箱(什么是装箱和拆箱)](https://cloud.tencent.com/developer/article/1745824)

[c#](https://cloud.tencent.com/developer/tag/10171)[编程算法](https://cloud.tencent.com/developer/tag/10663)

[装箱（boxing）和拆箱（unboxing）是C#类型系统的核心概念.是不同于C与C++的新概念！,通过装箱和拆箱操作，能够在值类型和引用类型中架起一做桥梁.换言之,可以轻松的实现值类型与引用类型的互相转换,装箱和拆箱能够统一考察系统,任何类型的值最终都可以按照对象进行处理. C#语言中的所有类型都是由基类System.Object继承过来的，包括最常用的基础类型：int, byte, short，bool等等，就是说所有的事物都是对象。如果申明这些类型得时候都在堆(HEAP)中分配内存，会造成极低的效率！(个中原因以及关于堆和栈得区别会在另一篇里单独得说说！) .NET如何解决这个问题得了？正是通过将类型分成值型(value)和引用型(regerencetype)，C#中定义的值类型包括原类型（Sbyte、Byte、Short、Ushort、Int、Uint、Long、Ulong、Char、Float、Double、Bool、Decimal）、枚举(enum)、结构(struct)，引用类型包括：类、数组、接口、委托、字符串等。 值型就是在栈中分配内存，在申明的同时就初始化，以确保数据不为NULL； 引用型是在堆中分配内存，初始化为null，引用型是需要GARBAGE COLLECTION来回收内存的，值型不用，超出了作用范围，系统就会自动释放！ 下面就来说装箱和拆箱的定义！ 装箱就是隐式的将一个值型转换为引用型对象。比如： int i=0; Syste.Object obj=i; 这个过程就是装箱！就是将i装箱！ 拆箱就是将一个引用型对象转换成任意值型！比如： int i=0; System.Object obj=i; int j=(int)obj; 这个过程前2句是将i装箱，后一句是将obj拆箱！ 再写个代码，看看进行了几次装拆箱！ int i=0; System.Object obj=i; Console.WriteLine(i+","+(int)obj); 其中共发生了3次装箱和一次拆箱！^\_^，看出来了吧？！ 第一次是将i装箱，第2次是输出的时候将i转换成string类型，而string类型为引用类型，即又是装箱，第三次装箱就是(int)obj的转换成string类型，装箱！ 拆箱就是(int)obj，将obj拆箱！！ 在C#中，将类和数组等都归为了引用型的，那么值类型和引用型有什么区别呢？](https://cloud.tencent.com/developer/article/1745824)

[C# 学习笔记（8）—— 深入理解类型](https://cloud.tencent.com/developer/article/2347876)

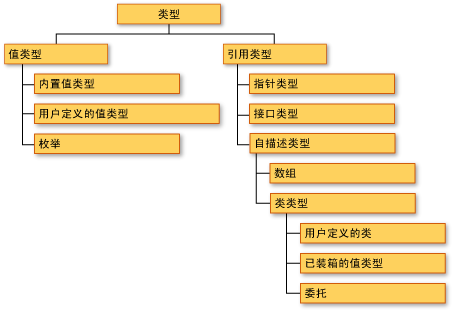
[c#](https://cloud.tencent.com/developer/tag/10171)[变量](https://cloud.tencent.com/developer/tag/17188)[内存](https://cloud.tencent.com/developer/tag/17382)[数据](https://cloud.tencent.com/developer/tag/17440)[学习笔记](https://cloud.tencent.com/developer/tag/17534)

[C# 中的类型可以分为两种——值类型和引用类型，本文详细分析两种类型，并讨论它们之间的类型转换方法](https://cloud.tencent.com/developer/article/2347876)

[C#/.NET Web 部分复习总结（面试常问）](https://cloud.tencent.com/developer/article/1552627)

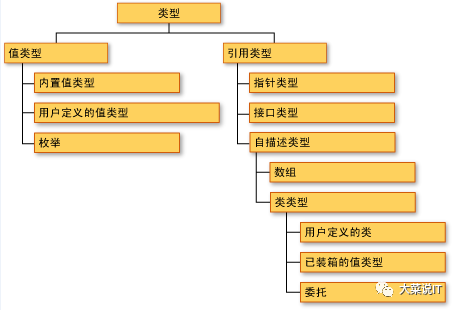
[.net](https://cloud.tencent.com/developer/tag/10180)[网站](https://cloud.tencent.com/developer/tag/10548)[c#](https://cloud.tencent.com/developer/tag/10171)[html](https://cloud.tencent.com/developer/tag/10205)[sql](https://cloud.tencent.com/developer/tag/10253)

[.NET一般指 .NET FrameWork框架，它是一种平台，一种技术。理论上，各种语言都可以编译成它的IL，在上面运行。.NET开发，就是把它当作目标平台的开发。](https://cloud.tencent.com/developer/article/1552627)



[c#基础系列1---值类型和引用类型](https://cloud.tencent.com/developer/article/1469968)

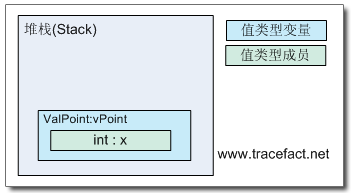
7550



[C#类型基础](https://cloud.tencent.com/developer/article/1350561)

[其他](https://cloud.tencent.com/developer/tag/125)

[本文之初的目的是讲述设计模式中的 Prototype(原型)模式，但是如果想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。](https://cloud.tencent.com/developer/article/1350561)



[C#理解泛型(译)](https://cloud.tencent.com/developer/article/1347483)

[其他](https://cloud.tencent.com/developer/tag/125)

[Visual C# 2.0 的一个最受期待的(或许也是最让人畏惧)的一个特性就是对于泛型的支持。这篇文章将告诉你泛型用来解决什么样的问题，以及如何使用它们来提高你的代码质量，还有你不必恐惧泛型的原因。](https://cloud.tencent.com/developer/article/1347483)

[C# 进行AI工程开发-基础篇](https://cloud.tencent.com/developer/article/2319303)

[c#](https://cloud.tencent.com/developer/tag/10171)[基础](https://cloud.tencent.com/developer/tag/17302)[开发](https://cloud.tencent.com/developer/tag/17337)[内存](https://cloud.tencent.com/developer/tag/17382)[语法](https://cloud.tencent.com/developer/tag/17559)

[c#中的多态学习总结](https://cloud.tencent.com/developer/article/1692802)

[面向对象编程](https://cloud.tencent.com/developer/tag/10705)[c++](https://cloud.tencent.com/developer/tag/10166)[c#](https://cloud.tencent.com/developer/tag/10171)

[c#的多台方法，大体上和c++的类似，但是有点区别的，我这里刚刚初学，因此把重点记录下。](https://cloud.tencent.com/developer/article/1692802)

用户4645519

2020/09/06

8200

[[译]聊聊C＃中的泛型的使用（新手勿入）](https://cloud.tencent.com/developer/article/1367110)

[那些年~~~我们的C#笔试内测题目](https://cloud.tencent.com/developer/article/1060210)

[c#](https://cloud.tencent.com/developer/tag/10171)[.net](https://cloud.tencent.com/developer/tag/10180)

[《深入.NET平台和C#编程》内部测试题-笔试试卷 一 选择题 1) 以下关于序列化和反序列化的描述错误的是（ C）。 a) 序列化是将对象的状态存储到特定存储介质中的过程 b) 二进制格式化器的Serialize()和Deserialize()方法可以分别用来实现序列化和反序列 化过程 解析：BinaryFormatter c) 如果一个类可序列化，则它的子类和包含的各成员对象也一定可序列化 问题出在了子类，如果子类压根不能进行序列化操作，则会抛出异常 d) 标识一个类可以序列化要使用[Serializ](https://cloud.tencent.com/developer/article/1060210)

房上的猫

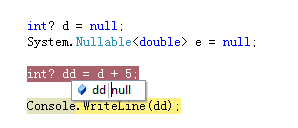
2018/03/14

2.4K0

[C#基础知识系列二（值类型和引用类型、可空类型、堆和栈、装箱和拆箱）](https://cloud.tencent.com/developer/article/1199922)

[其他](https://cloud.tencent.com/developer/tag/125)

[之前对几个没什么理解，只是简单的用过可空类型，也是知道怎么用，至于为什么，还真不太清楚，通过整理本文章学到了很多知识，也许对于以后的各种代码优化都有好处。](https://cloud.tencent.com/developer/article/1199922)



[C#基础深入学习02](https://cloud.tencent.com/developer/article/2366068)

[c#](https://cloud.tencent.com/developer/tag/10171)[存储](https://cloud.tencent.com/developer/tag/10665)[基础](https://cloud.tencent.com/developer/tag/17302)[集合](https://cloud.tencent.com/developer/tag/17303)[数组](https://cloud.tencent.com/developer/tag/17457)

[Hashtable 类代表了一系列基于键的哈希代码组织起来的键/值对。它使用键来访问集合中的元素。](https://cloud.tencent.com/developer/article/2366068)