

[首页](#)  
[最新文章](#)  
[经典回顾](#)  
[开发](#)  
[设计](#)  
[IT技术](#)  
[职场](#)  
[业界](#)  
[极客](#)  
[创业](#)  
[访谈](#)  
[在国外](#)

- 导航条 - ▾

[伯乐在线](#) > [首页](#) > [所有文章](#) > [其他](#) > .NET基础拾遗（1）：类型语法基础和内存管理基础

# .NET基础拾遗（1）：类型语法基础和内存管理基础

2015/10/13 • [其他](#), [开发](#) • [1 评论](#) • [.Net](#)

分享到: 3

[Android-节日短信送祝福（功能篇）](#)  
[Android-Volley详解](#)  
[Android实现卫星菜单](#)  
[懒人必备之Android效率开发框架](#)

原文出处: [周旭龙 \(@周旭龙EdisonChou\)](#)

## 一、基础类型和语法

### 1.1 .NET中所有类型的基类是什么？

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：

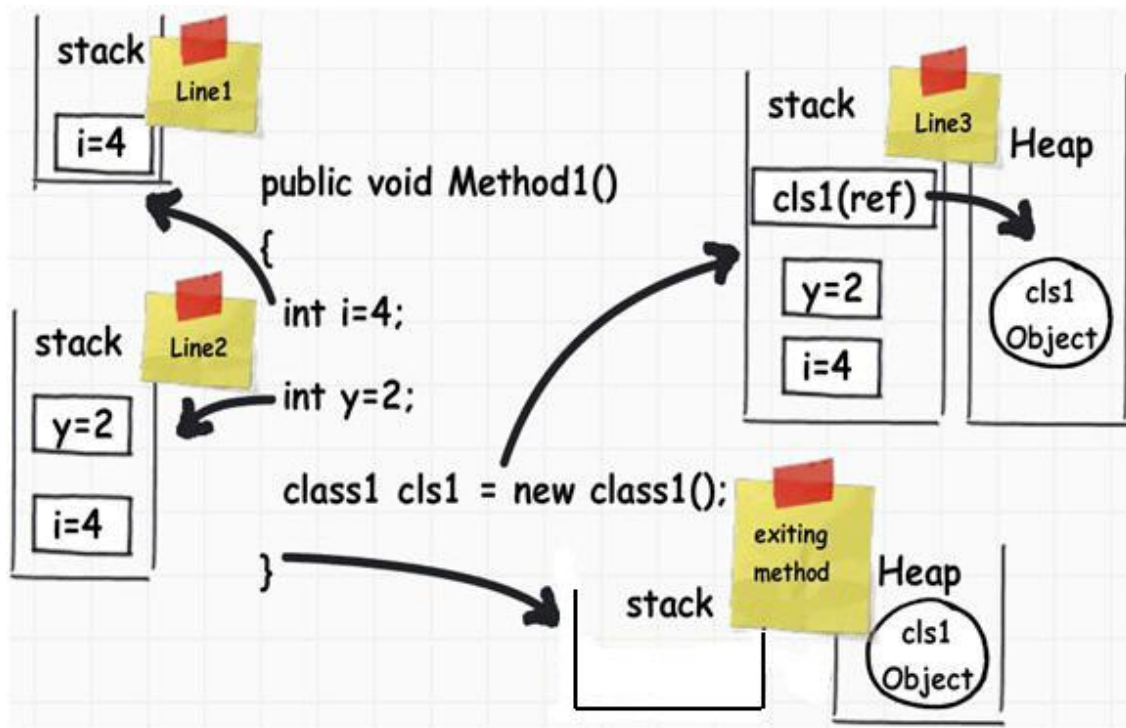
```
1 public class A { }  
2 public class A : System.Object { }
```

### 1.2 值类型和引用类型的区别？

在.NET中的类型分为值类型和引用类型，它们各有特点，其共同点是都继承自System.Object，但最明显的区分标准却是是否继承自System.ValueType（System.ValueType继承自System.Object），也就是

说所有继承自System.ValueType的类型是值类型，而其他类型都是引用类型。常用的值类型包括：结构、枚举、整数型、浮点型、布尔型等等；而在C#中所有以class关键字定义的类型都是引用类型。

PS：严格来讲，System.Object作为所有内建类型的基类，本身并没有值类型和引用类型之分。但是System.Object的对象，具有引用类型的特点。这也是值类型在某些场合需要装箱和拆箱操作的原因。



### （1）赋值时的区别

这是值类型与引用类型最显著的一个区别：值类型的变量直接将获得一个真实的数据副本，而对引用类型的赋值仅仅是把对象的引用赋给变量，这样就可能导致多个变量引用到一个对象实例上。

### （2）内存分配的区别

引用类型的对象将会在堆上分配内存，而值类型的对象则会在堆栈上分配内存。堆栈空间相对有限，但是运行效率却比堆高很多。

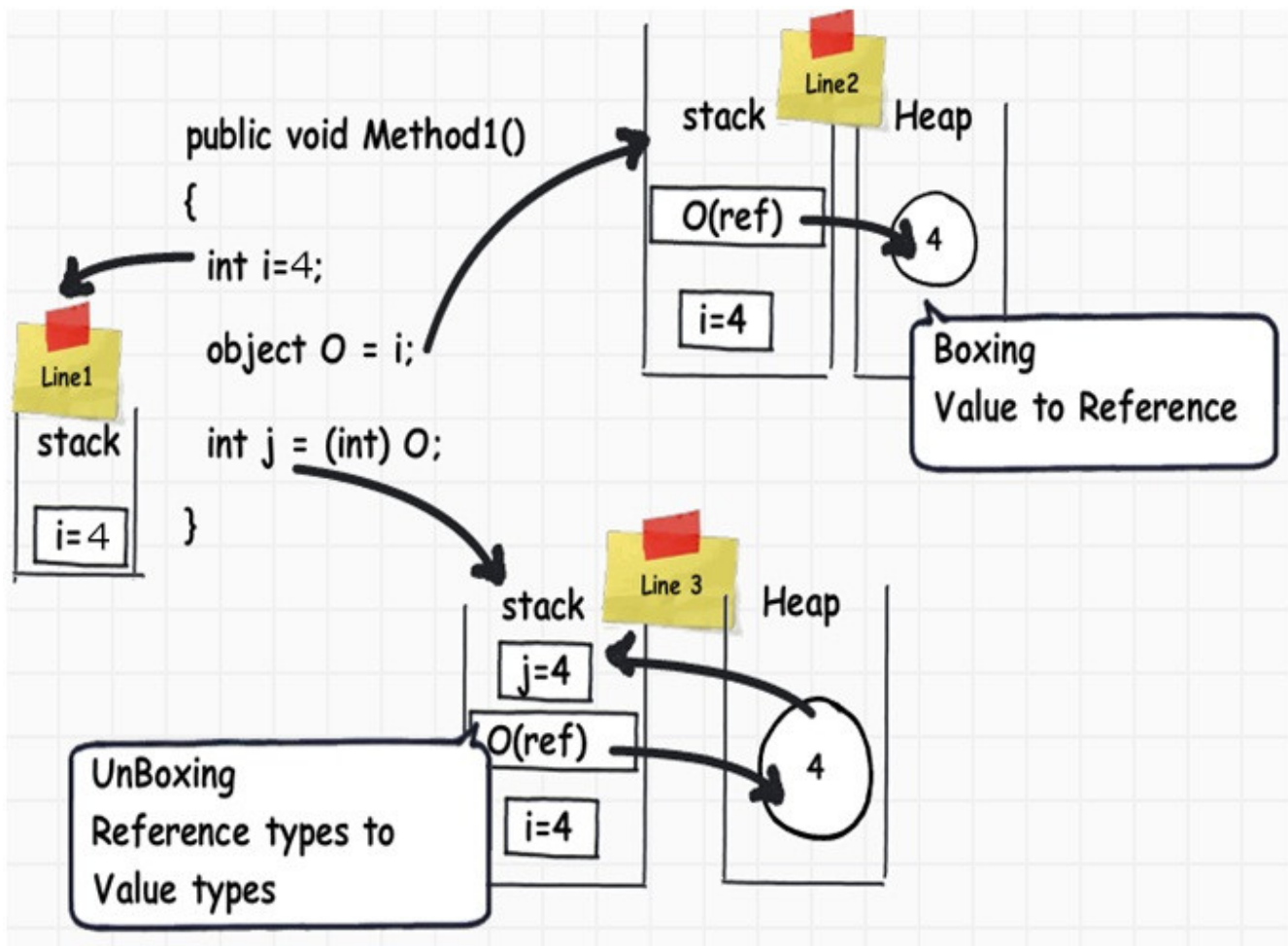
### （3）继承结构的区别

由于所有的值类型都有一个共同的基类System.ValueType，因此值类型具有了一些引用类型所不具有的共同性质，比较重要的一点就是值类型的比较方法：Equals。所有的值类型已经实现了内容的比较（而不再是引用地址的比较），而引用类型没有重写Equals方法还是采用引用比较。

## 1.3 装箱和拆箱的原理？

（1）装箱：CLR需要做额外的工作把堆栈上的值类型移动到堆上，这个操作就被称为装箱。

（2）拆箱：装箱操作的反操作，把堆中的对象复制到堆栈中，并且返回其值。



装箱和拆箱都意味着堆和堆栈空间的一系列操作，毫无疑问，这些操作的性能代价是很大的，尤其对于堆上空间的操作，速度相对于堆栈的操作慢得多，并且可能引发垃圾回收，这些都将大规模地影响系统的性能。因此，我们应该避免任何没有必要的装箱和拆箱操作。

如何避免呢，首先分析装箱和拆箱经常发生的场合：

①值类型的格式化输出

②System.Object类型的容器

对于第①种情况，我们可以通过下面的改动示例来避免：

```

1 int i = 10;
2 Console.WriteLine("The value is {0}", i.ToString());

```

对于第②种情况，则可以使用泛型技术来避免使用针对System.Object类型的容器，有效避免大规模地使用装箱和拆箱：

```

1 ArrayList arrList = new ArrayList();
2 arrList.Add(0);
3 arrList.Add("1");
4 // 使用泛型数据结构代替ArrayList
5 List<int> intList = new List<int>();
6 intList.Add(1);
7 intList.Add(2);

```

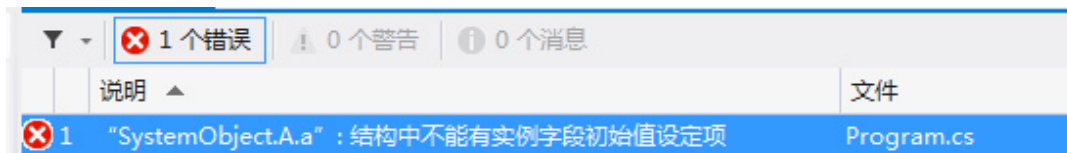
1.4 struct和class的区别，struct适用于哪些场合？

首先，struct（结构）是值类型，而class（类）是引用类型，所有的结构对象都分配在堆栈上，而所有的类对象都分配在堆上。

其次，struct与class相比，不具备继承的特性，struct虽然可以重写定义在System.Object中的虚方法，但不能定义新的虚方法和抽象方法。

最后，struct不能有无参数的构造方法（class默认就有），也不能为成员变量定义初始值。

```
1 public struct A
2 {
3     public int a = 1; // 这里不能编译通过
4 }
```



结构对象在构造时必须被初始化为0，构造一个全0的对象是指在内存中为对象分配一个合适的空间，并且把该控件置为0。

如何使用struct or class? 当一个类型仅仅是原始数据的集合，而不需要复杂的操作时，就应该设计为struct，否则就应该设计为一个class。

### 1.5 C#中方法的参数传递有哪几种方式？

（1）ref关键字：引用传递参数，需要在传递前初始化；（ref 要求参数在传入前被初始化）

（2）out关键字：引用传递参数，需要在返回前初始化；（out 要求参数在方法返回前被初始化）

ref和out这两个关键字的功能极其类似，都用来说明该参数以引用方式进行传递。大家都知道，.NET的类型分为引用类型和值类型，当一个方法参数是引用类型时，传递的本质就是对象的引用。所以，这两个关键字的作用都发生在值类型上。

（3）params关键字：允许方法在定义时不确定参数的数量，这种形式非常类似数组参数，但形式更加简洁易懂。

But，params关键字的使用也有一定局限：当一个方法申明了一个params参数后，就不允许在其后面再有任何其他参数。

例如下面一段代码，定义了两个完全相等的方法：NotParams和UseParams，使用由params修饰参数的方法时，可以直接把所有变量集合传入而无须先申明一个数组对象。

```
1 class Program
2 {
3     static void Main(string[] args)
4     {
5         // params
6         string s = "I am a string";
7         int i = 10;
8         double f = 2.3;
9         object[] par = new object[3] { s, i, f };
10        // not use params
11        NotParams(par);
12        // use params
13        UseParams(s, i, f);
14
15        Console.ReadKey();
16    }
17
18    // Not use params
```

```
19 public static void NotParams(object[] par)
20 {
21     foreach (var obj in par)
22     {
23         Console.WriteLine(obj);
24     }
25 }
26
27 // Use params
28 public static void UseParams(params object[] par)
29 {
30     foreach (var obj in par)
31     {
32         Console.WriteLine(obj);
33     }
34 }
35 }
```

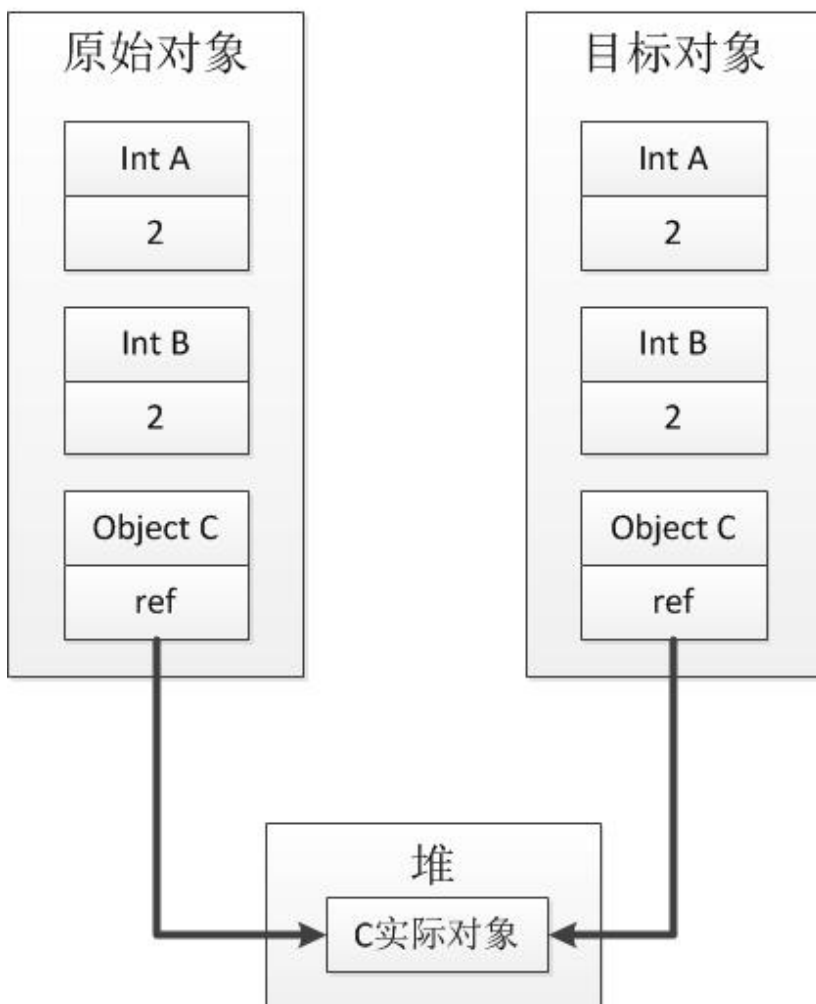
## 1.6 浅复制和深复制的区别？

（1）浅复制：复制一个对象的时候，仅仅复制原始对象中所有的非静态类型成员和所有的引用类型成员的引用。（新对象和原对象将共享所有引用类型成员的实际对象）

（2）深复制：复制一个对象的时候，不仅复制所有非静态类型成员，还要复制所有引用类型成员的实际对象。

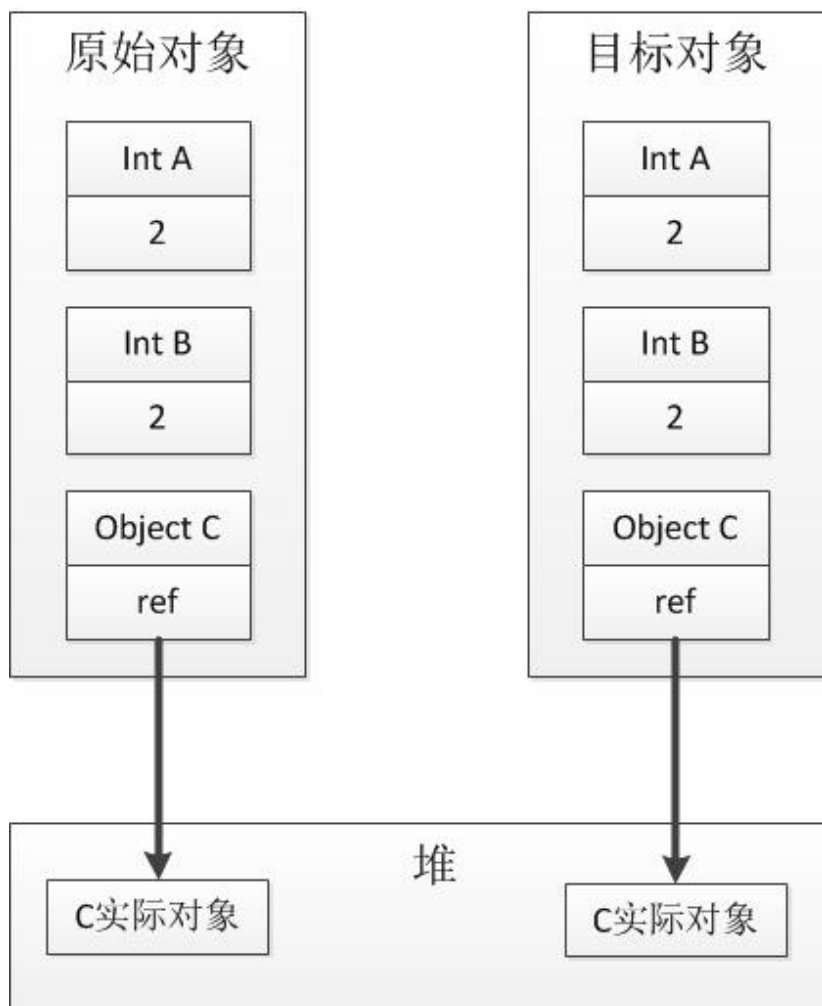
下图展示了浅复制和深复制的区别：

### 浅复制





## 深复制



在.NET中，基类System.Object已经为所有类型都实现了浅复制，类型所要做的就是公开一个复制的接口，而通常的，这个接口会由ICloneable接口来实现。ICloneable只包含一个方法Clone，该方法既可以被实现为浅复制也可以被实现为深复制，具体如何取舍则根据具体类型的需求决定。此外，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，要想实现浅度复制时，只需要调用这个方法就可以了：

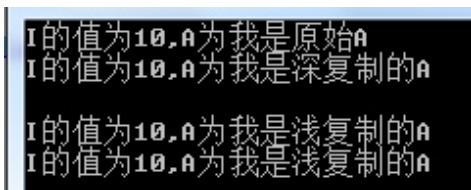
```
1 public object Clone()  
2 {  
3     return MemberwiseClone();  
4 }
```

下面的代码展示了一个使用ICloneable接口提供深复制的简单示例：

```
1 public class DeepCopy : ICloneable  
2 {  
3     public int i = 0;  
4     public A a = new A();  
5  
6     public object Clone()  
7     {  
8         // 实现深复制-方式1: 依次赋值和实例化  
9         DeepCopy newObj = new DeepCopy();  
10        newObj.a = new A();  
11        newObj.a.message = this.a.message;  
12    }  
13 }
```

```
12         newObj.i = this.i;
13
14         return newObj;
15     }
16
17     public new object MemberwiseClone()
18     {
19         // 实现浅复制
20         return base.MemberwiseClone();
21     }
22
23     public override string ToString()
24     {
25         string result = string.Format("I的值为{0},A为{1}", this.i.ToString(), this.a.
26         return result;
27     }
28 }
29
30 public class A
31 {
32     public string message = "我是原始A";
33 }
34
35 public class Program
36 {
37     static void Main(string[] args)
38     {
39         DeepCopy dc = new DeepCopy();
40         dc.i = 10;
41         dc.a = new A();
42
43         DeepCopy deepClone = dc.Clone() as DeepCopy;
44         DeepCopy shadowClone = dc.MemberwiseClone() as DeepCopy;
45
46         // 深复制的目标对象将拥有自己的引用类型成员对象
47         deepClone.a.message = "我是深复制的A";
48         Console.WriteLine(dc);
49         Console.WriteLine(deepClone);
50         Console.WriteLine();
51         // 浅复制的目标对象将和原始对象共享引用类型成员对象
52         shadowClone.a.message = "我是浅复制的A";
53         Console.WriteLine(dc);
54         Console.WriteLine(shadowClone);
55
56         Console.ReadKey();
57     }
58 }
```

其执行结果如下图所示，可以清楚地看到对深复制对象的属性的赋值不会影响原始对象，而浅复制则相反。



从上面的代码中可以看到，在深复制的实现中，如果每个对象都要这样去进行深度复制就太麻烦了，可以利用序列化/反序列化来对对象进行深度复制：先把对象序列化（Serialize）到内存中，然后再进行反序列化，通过这种方式来进行对象的深度复制：

```
1 [Serializable]
2 public class DeepCopy : ICloneable
3 {
4     .....
```

```
5
6     public object Clone()
7     {
8         // 实现深复制-方式1: 依次赋值和实例化
9         // DeepCopy newObj = new DeepCopy();
10        // newObj.a = new A();
11        // newObj.a.message = this.a.message;
12        // newObj.i = this.i;
13
14        //return newObj;
15        // 实现深复制-方式2: 序列化/反序列化
16        BinaryFormatter bf = new BinaryFormatter();
17        MemoryStream ms = new MemoryStream();
18        bf.Serialize(ms, this);
19        ms.Position = 0;
20        return bf.Deserialize(ms);
21    }
22
23    .....
24 }
25 [Serializable]
26 public class A
27 {
28     public string message = "我是原始A";
29 }
```

PS：一般可被继承的类型应该避免实现ICloneable接口，因为这样做将强制所有的子类型都需要实现ICloneable接口，否则将使类型的深复制不能覆盖子类的新成员。

## 二、内存管理和垃圾回收

### 2.1 .NET中栈和堆的差异？

每一个.NET应用程序最终都会运行在一个OS进程中，假设这个OS的传统的32位系统，那么每个.NET应用程序都可以拥有一个4GB的虚拟内存。.NET会在这个4GB的虚拟内存块中开辟三块内存作为 堆栈、托管堆 以及 非托管堆。

#### （1）.NET中的堆栈

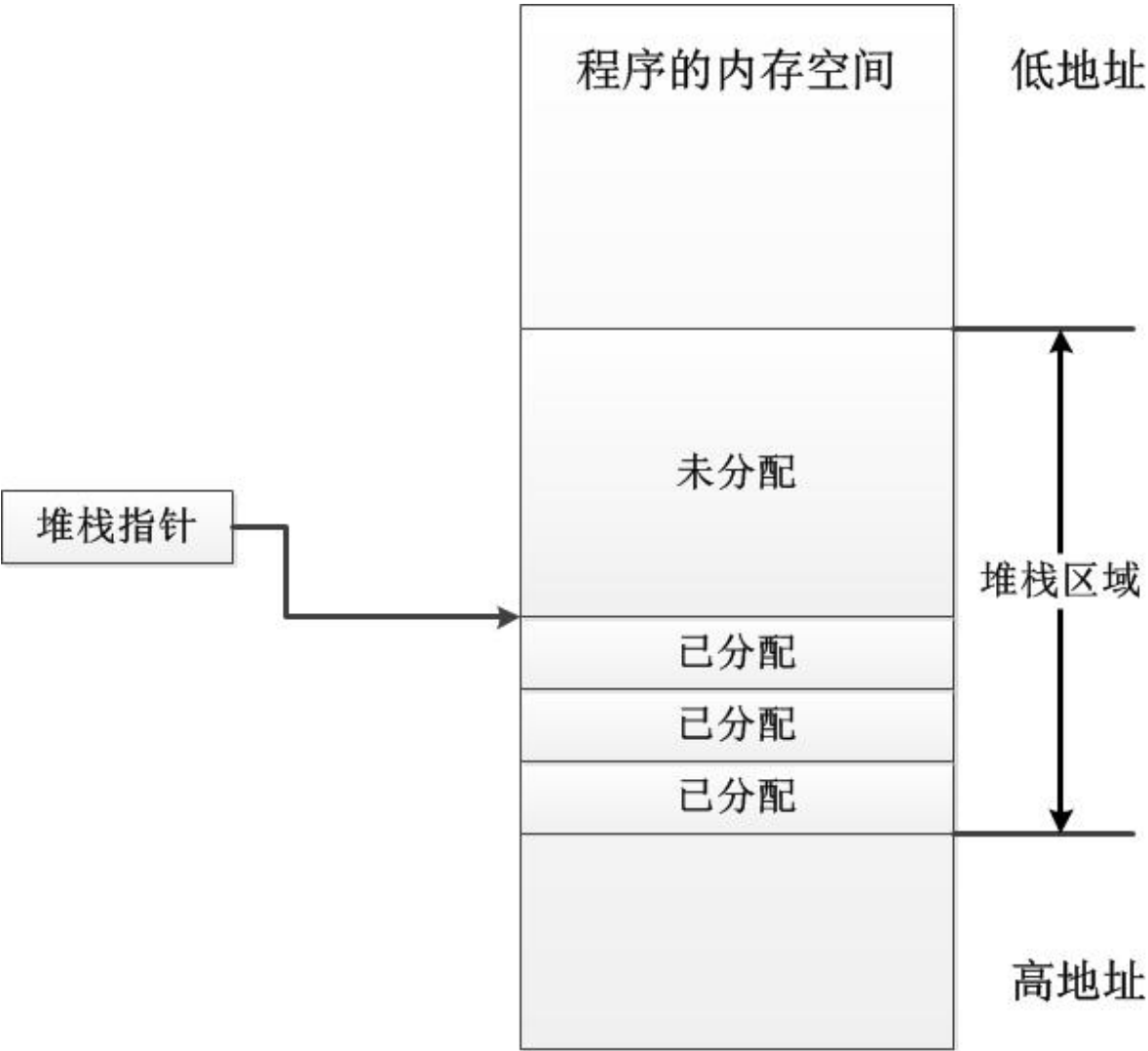
堆栈用来存储值类型的对象和引用类型对象的引用（地址），其分配的是一块连续的地址，如下图所示，在.NET应用程序中，堆栈上的地址从高位向低位分配内存，.NET只需要保存一个指针指向下一个未分配内存的内存地址即可。

对于所有需要分配的对象，会依次分配到堆栈中，其释放也会严格按照栈的逻辑（FILO，先进后出）依次进行退栈。（这里的“依次”是指按照变量的作用域进行的），假设有以下一段代码：

```
1 TempClass a = new TempClass();
2 a.numA = 1;
3 a.numB = 2;
```

其在堆栈中的内存图如下图所示：



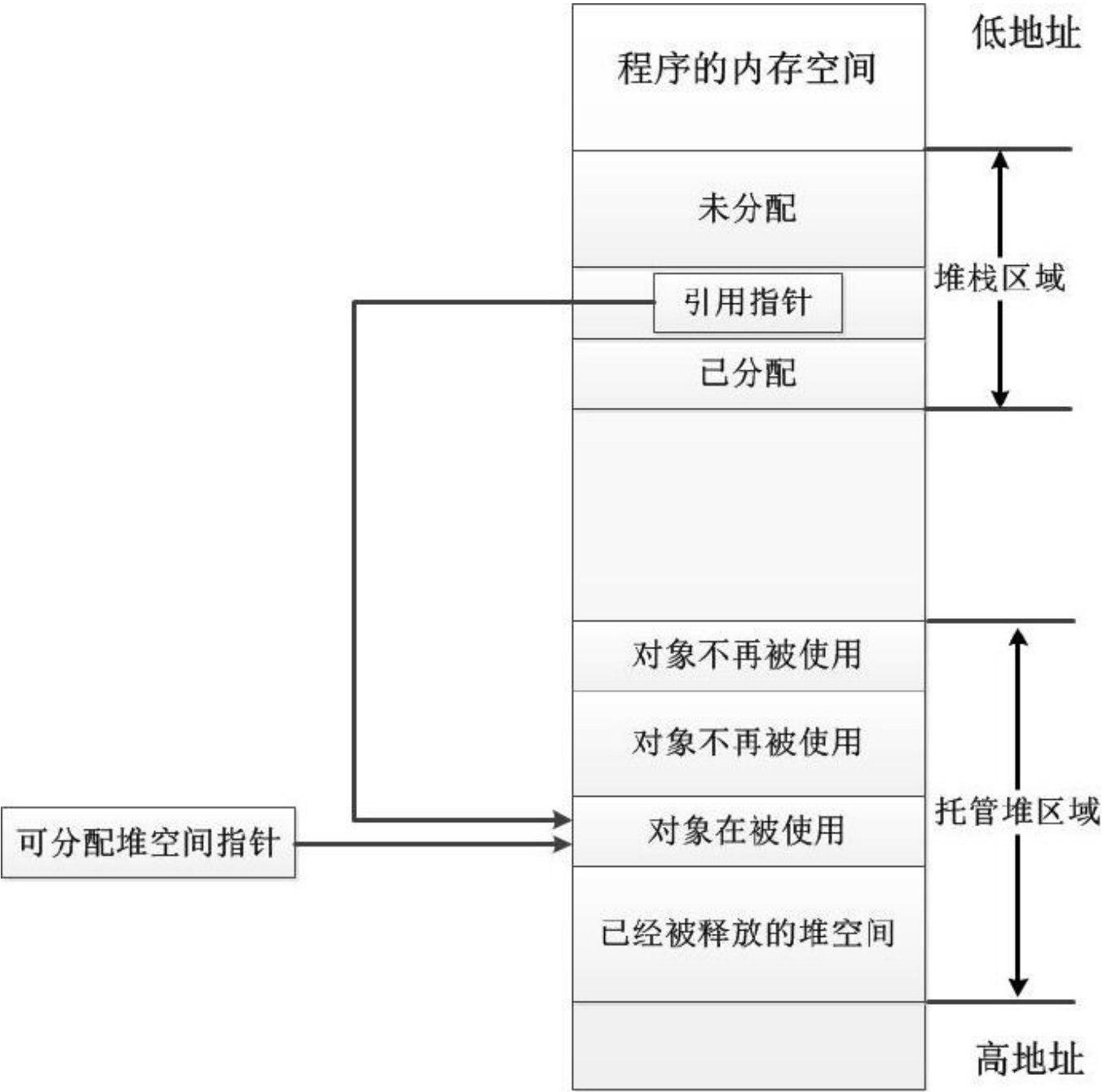


这里TempClass是一个引用类型，拥有两个整型的int成员，在栈中依次需要分配的是a的引用，a.numA和a.numB。当a的作用域结束之后，这三个会按照a.numB→a.numA→a的顺序依次退栈。

（2）.NET中的托管堆

众所周知，.NET中的引用类型对象时分配在托管堆上的，和堆栈一样，托管堆也是进程内存空间中的一块区域。But，托管堆的内存分配却和堆栈有很大区别。受益于.NET内存管理机制，托管堆的分配也是连续的（从低位到高位），但是堆中却存在着暂时不能被分配却已经无用的对象内存块。

当一个引用类型对象被初始时，会通过指向堆上可用空间的指针分配一块连续的内存，然后使堆栈上的引用指向堆上刚刚分配的这块内存块。下图展示了托管堆的内存分配方式：



如上图所示，.NET程序通过分配在堆栈中的引用来找到分配在托管堆的对象实例。当堆栈中的引用退出作用域时，这时仅仅就断开和实际对象实例的引用联系。而当托管堆中的内存不够时，.NET会开始执行GC（垃圾回收）机制。GC是一个非常复杂的过程，它不仅涉及托管堆中对象的释放，而且需要移动合并托管堆中的内存块。当GC之后，堆中不再被使用的对象实例才会被部分释放（注意并不是完全释放），而在这之前，它们在堆中是暂时不可用的。在C/C++中，由于没有GC，因此可以直接free/delete来释放内存。

（3）.NET中的非托管堆

.NET程序还包含了非托管堆，所有需要分配堆内存的非托管资源将会被分配到非托管堆上。非托管的堆需要程序员用指针手动地分配和释放内存，.NET中的GC和内存管理不适用于非托管堆，其内存块也不会被合并移动，所以非托管堆的内存分配是按块的、不连续的。因此，这也解释了我们为何在使用非托管资源（如:文件流、数据库连接等）需要手动地调用Dispose()方法进行内存释放的原因。

2.2 执行string abc=" aaa" + " bbb" + " ccc" 共分配了多少内存？

这是一个经典的基础知识题目，它涉及了字符串的类型、堆栈和堆的内存分配机制，因此被很多人拿来考核开发者的基础知识功底。首先，我们都知道，判断值类型的标准是查看该类型是否会继承自System.ValueType，通过查看和分析，string直接继承于System.Object，因此string是引用类型，其内存分配会遵照引用类型的规范，也就是说如下的代码将会在堆栈上分配一块存储引用的内存，然后再

在堆上分配一块存储字符串实例对象的内存。

```
1 string a = "edc";
```

现在再来看看string abc="aaa"+"bbb"+"ccc"，按照常规的思路，字符串具有不可变性，大部分人会认为这里的表达式会涉及很多临时变量的生成，可能C#编译器会先执行"aaa"+"bbb"，并且把结果值赋给一个临时变量，再执行临时变量和"ccc"相加，最后把相加的结果再赋值给abc。But，其实C#编译器比想象中要聪明得多，以下的C#代码和IL代码可以充分说明C#编译器的智能：

```
1 // The first format
2 string first = "aaa" + "bbb" + "ccc";
3 // The second format
4 string second = "aaabbbccc";
5 // Display string
6 Console.WriteLine(first);
7 Console.WriteLine(second);
```

该C#代码的IL代码如下图所示：

```
.method private hidebysig static void Main(string[] args) cil managed
{
    .entrypoint
    .maxstack 1
    .locals init (
        [0] string first,
        [1] string second)
    L_0000: nop
    L_0001: ldstr "aaabbbccc" // 第一种赋值方式，C#编译器将其看成一个完整的字符串对象，
    L_0006: stloc.0           而不是执行某些拼接方法
    L_0007: ldstr "aaabbbccc"
    L_000c: stloc.1
    L_000d: ldloc.0
    L_000e: call void [mscorlib]System.Console::WriteLine(string)
    L_0013: nop
    L_0014: ldloc.1
    L_0015: call void [mscorlib]System.Console::WriteLine(string)
    L_001a: nop
    L_001b: call valuetype [mscorlib]System.ConsoleKeyInfo [mscorlib]System.Console::ReadKey()
    L_0020: pop
    L_0021: ret
}
```

正如我们所看到的，string abc="aaa"+"bbb"+"ccc"；这样的表达式被C#编译器看成一个完整的字符串"aaabbbccc"，而不是执行某些拼接方法，可以将其看作是C#编译器的优化，所以在本次内存分配中只是在栈中分配了一个存储字符串引用的内存块，以及在托管堆分配了一块存储"aaabbbccc"字符串对象的内存块。

那么，我们的常规思路在.NET程序中又是怎么体现的呢？我们来看下一段代码：

```
1 int num = 1;
2 string str = "aaa" + num.ToString();
3 Console.WriteLine(str);
```

这里我们首先初始化了一个int类型的变量，其次初始化了一个string类型的字符串，并执行 + 操作，这时我们来看看其对应的IL代码：

```
.method private hidebysig static void Main(string[] args) cil managed
{
    .entrypoint
    .maxstack 2
    .locals init (
        [0] int32 num,
        [1] string str)
    L_0000: nop
    L_0001: ldc.i4.1
    L_0002: stloc.0
    L_0003: ldstr "aaa"
    L_0008: ldloc.s num
    L_000a: call instance string [mscorlib]System.Int32::ToString()
    L_000f: call string [mscorlib]System.String::Concat(string, string)
    L_0014: stloc.1
    L_0015: ldloc.1
    L_0016: call void [mscorlib]System.Console::WriteLine(string)
    L_001b: nop
    L_001c: call valuetype [mscorlib]System.ConsoleKeyInfo [mscorlib]System.Console::ReadKey()
    L_0021: pop
    L_0022: ret
}
```

产生了两个string类型，即说明这样的代码不能被C#识别并优化，会产生临时变量

如上图所示，在这段代码中执行 + 操作，会调用String的Concat方法，该方法需要传入两个string类型的参数，也就产生了另一个string类型的临时变量。换句话说，在此次内存分配中，堆栈中会分配一个存储字符串引用的内存块，在托管堆则分配了两块内存块，分别存储了存储”aaa”字符串对象和”1”字符串对象。

可能这段代码还是不熟悉，我们再来看看下面一段代码，我们就感觉十分亲切熟悉了：

```
1 string str = "aaa";
2 str += "bbb";
3 str += "ccc";
4 Console.WriteLine(str);
```

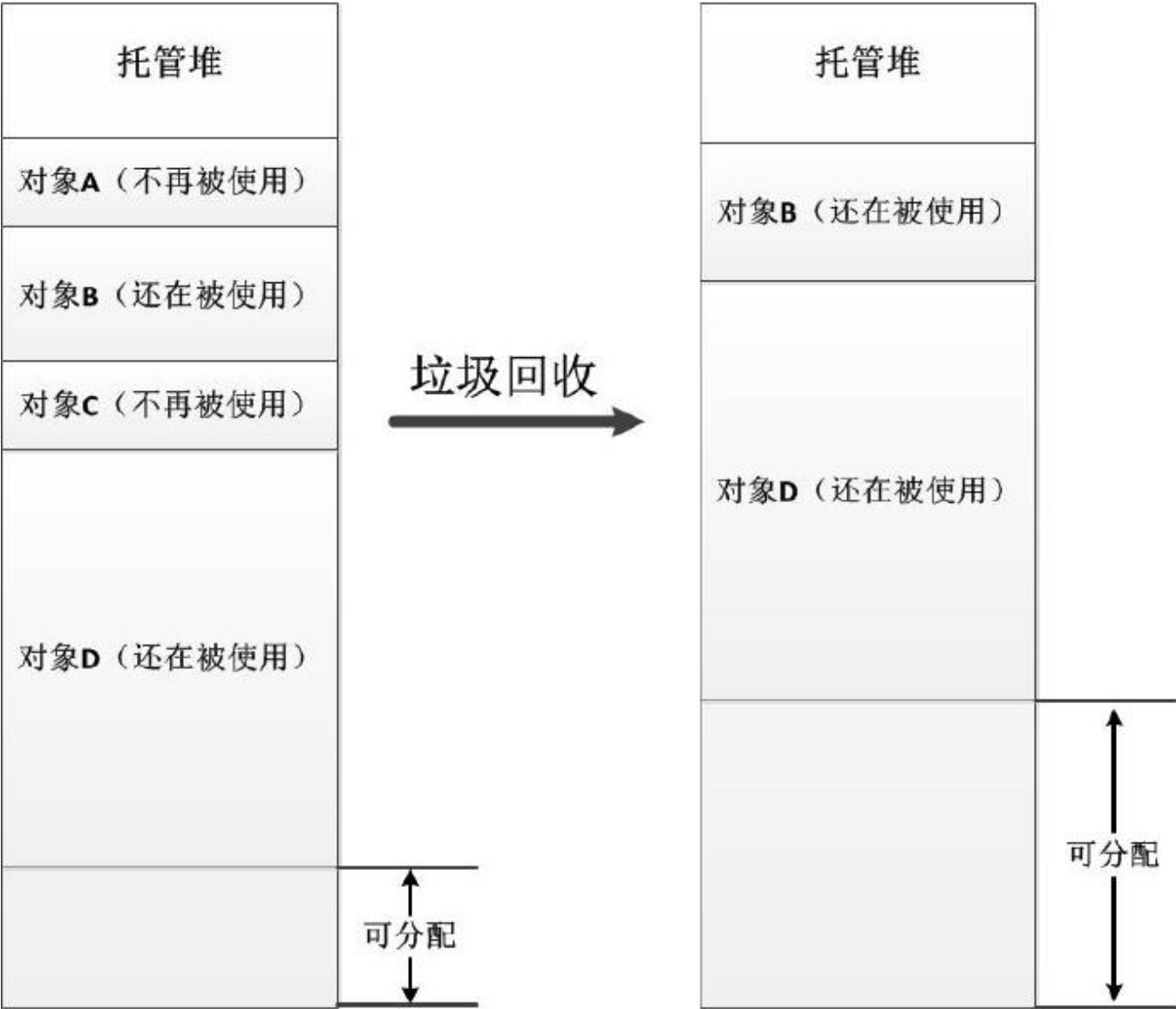
其对应的IL代码如下图所示：

```
.method private hidebysig static void Main(string[] args) cil managed
{
    .entrypoint
    .maxstack 2
    .locals init (
        [0] string str)
    L_0000: nop
    L_0001: ldstr "aaa"
    L_0006: stloc.0
    L_0007: ldloc.0
    L_0008: ldstr "bbb" 临时字符串变量1
    L_000d: call string [mscorlib]System.String::Concat(string, string)
    L_0012: stloc.0
    L_0013: ldloc.0
    L_0014: ldstr "ccc" 临时字符串变量2
    L_0019: call string [mscorlib]System.String::Concat(string, string)
    L_001e: stloc.0
    L_001f: ldloc.0
    L_0020: call void [mscorlib]System.Console::WriteLine(string)
    L_0025: nop
    L_0026: call valueType [mscorlib]System.ConsoleKeyInfo [mscorlib]System.Console::ReadKey()
    L_002b: pop
    L_002c: ret
}
```

如图可以看出，在拼接过程中产生了两个临时字符串对象，并调用了两次String.Concat方法进行拼接，就不用多解释了。

### 2.3 简要说说.NET中GC的运行机制

GC是垃圾回收（Garbage Collect）的缩写，它是.NET众多机制中最为重要的一部分，也是对我们的代码书写方式影响最大的机制之一。.NET中的垃圾回收是指清理托管堆上不会再被使用的对象内存，并且移动仍在被使用的对象使它们紧靠托管堆的一边。下图展示了一次垃圾回收之后托管堆上的变化（这里仅仅为了说明，简化了GC的执行过程，省略了包含Finalize方法对象的处理以及大对象分配的特殊性）：



如上图所示，我们可以知道GC的执行过程分为两个基本动作：

- （1）一是找到所有不再被使用的对象：对象A和对象C，并标记为垃圾；
- （2）二是移动仍在被使用的对象：对象B和对象D。

这样之后，对象A和对象C所占用的内存空间就被腾空出来，以备下次分配的时候使用。

PS：通常情况下，我们不需要手动干预垃圾回收的执行，不过CLR仍然提供了一个手动执行垃圾回收的方法：`GC.Collect()`。当我们需要在某一对象不再使用并且及时释放内存的时候可以调用该方法来实现。But，垃圾回收的运行成本较高（涉及到了对象块的移动、遍历找到不再被使用的对象、很多状态变量的设置以及`Finalize`方法的调用等等），对性能影响也较大，因此我们在编写程序时，应该避免不必要的内存分配，也尽量减少或避免使用`GC.Collect()`来执行垃圾回收。

2.4 Dispose和Finalize方法在何时被调用？

由于有了垃圾回收机制的支持，对象的析构（或释放）和C++有了很大的不同，这就需要在设计类型的时候，充分理解.NET的机制，明确怎样利用`Dispose`方法和`Finalize`方法来保证一个对象正确而高效地被析构。

（1）Dispose方法

```
1 // 摘要：  
2 //      定义一种释放分配的资源的方法。
```



```
3 [ComVisible(true)]
4 public interface IDisposable
5 {
6     // 摘要:
7     //     执行与释放或重置非托管资源相关的应用程序定义的任务。
8     void Dispose();
9 }
```

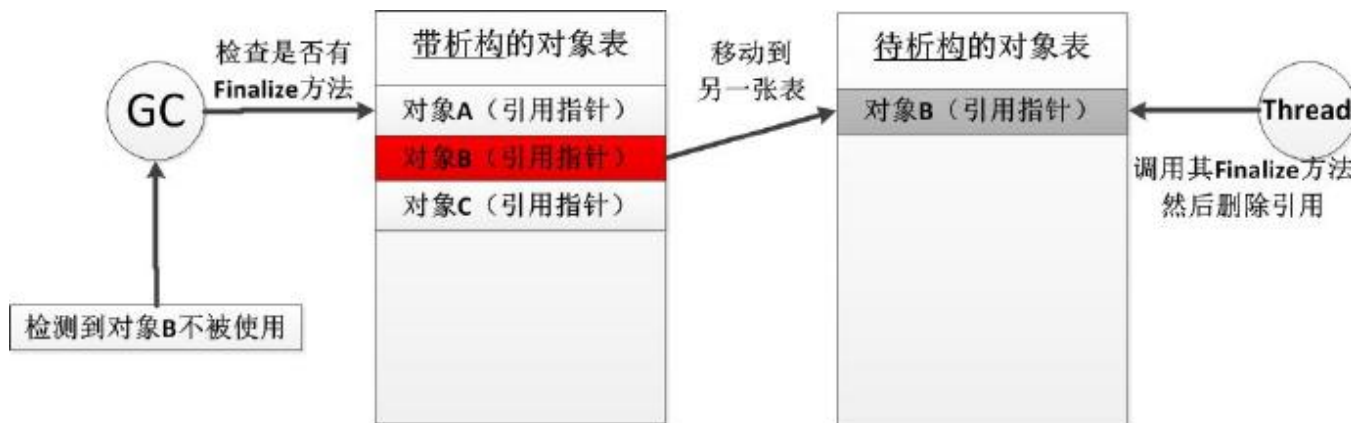
Microsoft考虑到很多情况下程序员仍然希望在对象不再被使用时进行一些清理工作，所以.NET提供了IDisposable接口并且在其中定义了Dispose方法。通常我们会在Dispose方法中实现一些托管对象和非托管对象的释放以及业务逻辑的结束工作等等。

But，即使我们实现了Dispose方法，也不能得到任何有关释放的保证，Dispose方法的调用依赖于类型的使用者，当类型被不恰当地使用，Dispose方法将不会被调用。因此，我们一般会借助using等语法来帮助Dispose方法被正确调用。

## （2）Finalize方法

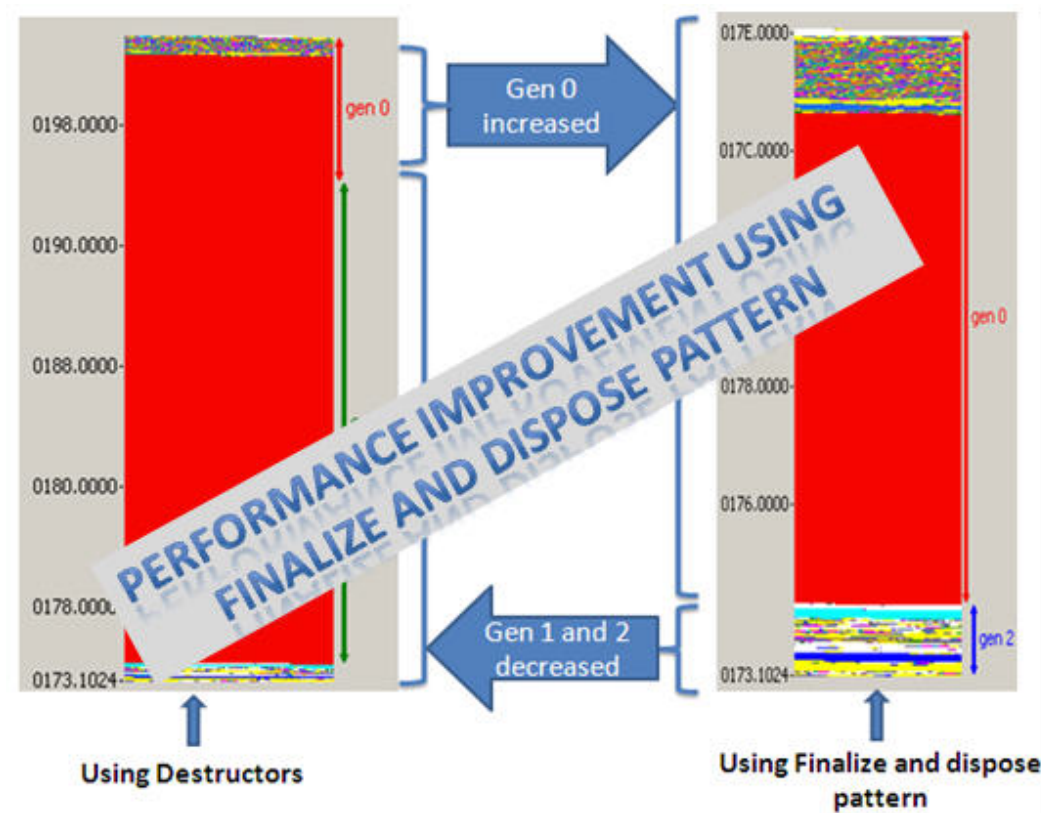
刚刚提到Dispose方法的调用依赖于类型的使用者，为了弥补这一缺陷，.NET还提供了Finalize方法。Finalize方法类似于C++中的析构函数(方法)，但又和C++的析构函数不同。Finalize在GC执行垃圾回收时被调用，其具体机制如下：

- ①当每个包含Finalize方法的类型的实例对象被分配时，.NET会在一张特定的表结构中添加一个引用并且指向这个实例对象，暂且称该表为“带析构方法的对象表”；
- ②当GC执行并且检测到一个不被使用的对象时，需要进一步检查“带析构方法的对象表”来查询该对象类型是否含有Finalize方法，如果没有则将该对象视为垃圾，如果存在则将该对象的引用移动到另外一张表，暂且称其为“待析构的对象表”，并且该对象实例仍然被视为在被使用。
- ③CLR将有一个单独的线程负责处理“待析构的对象表”，其执行方法内部就是依次通过调用其中每个对象的Finalize方法，然后删除引用，这时托管堆中的对象实例就被视为不再被使用。
- ④下一个GC执行时，将释放已经被调用Finalize方法的那些对象实例。



## （3）结合使用Dispose和Finalize方法：标准Dispose模式

Finalize方法由于有CLR保证调用，因此比Dispose方法更加安全（这里的安全是相对的，Dispose需要类型使用者的及时调用），但在性能方面Finalize方法却要差很多。因此，我们在类型设计时一般都会使用标准Dispose模式：Finalize方法作为Dispose方法的后备，只有在使用者没有调用Dispose方法的情况下，Finalize方法才被视为需要执行。这一模式保证了对象能够被高效和安全地释放，已经被广泛使用。



下面的代码则是实现这种标准Dispose模式的一个模板：

```

1 public class BaseTemplate : IDisposable
2 {
3     // 标记对象是否已经被释放
4     private bool isDisposed = false;
5     // Finalize方法
6     ~BaseTemplate()
7     {
8         Dispose(false);
9     }
10    // 实现IDisposable接口的Dispose方法
11    public void Dispose()
12    {
13        Dispose(true);
14        // 告诉GC此对象的Finalize方法不再需要被调用
15        GC.SuppressFinalize(this);
16    }
17    // 虚方法的Dispose方法做实际的析构工作
18    protected virtual void Dispose(bool isDisposing)
19    {
20        // 当对象已经被析构，则不必再继续执行
21        if(isDisposed)
22        {
23            return;
24        }
25
26        if(isDisposing)
27        {
28            // Step1:在这里释放托管资源
29        }
30
31        // Step2:在这里释放非托管资源
32
33        // Step3:最后标记对象已被释放
34        isDisposed = true;
35    }

```

```
36
37     public void MethodA()
38     {
39         if(isDisposed)
40         {
41             throw new ObjectDisposedException("对象已经释放");
42         }
43
44         // Put the logic code of MethodA
45     }
46
47     public void MethodB()
48     {
49         if (isDisposed)
50         {
51             throw new ObjectDisposedException("对象已经释放");
52         }
53
54         // Put the logic code of MethodB
55     }
56 }
57
58 public sealed class SubTemplate : BaseTemplate
59 {
60     // 标记子类对象是否已经被释放
61     private bool disposed = false;
62
63     protected override void Dispose(bool isDisposing)
64     {
65         // 验证是否已被释放，确保只被释放一次
66         if(disposed)
67         {
68             return;
69         }
70
71         if(isDisposing)
72         {
73             // Step1:在这里释放托管的并且在这个子类型中声明的资源
74         }
75
76         // Step2:在这里释放非托管的并且这个子类型中声明的资源
77
78         // Step3:调用父类的Dispose方法来释放父类中的资源
79         base.Dispose(isDisposing);
80         // Step4:设置子类的释放标识
81         disposed = true;
82     }
83 }
```

真正做释放工作的只是受保护的虚方法Dispose，它接收一个bool参数，主要用于区分调用者是类型的使用者还是.NET的GC机制。两者的区别在于通过Finalize方法释放资源时不能再释放或使用对象中的托管资源，这是因为这时的对象已经处于不被使用的状态，很有可能其中的托管资源已经被释放掉了。在Dispose方法中GC.SuppressFinalize(this)告诉GC此对象在被回收时不需要调用Finalize方法，这一句是改善性能的关键，记住实现Dispose方法的本质目的就在于避免所有释放工作在Finalize方法中进行。

## 2.5 GC中代（Generation）是什么，分为几代？

在.NET的GC执行垃圾回收时，并不是每次都扫描托管堆内的所有对象实例，这样做太耗费时间而且没有必要。相反，GC会把所有托管堆内的对象按照其已经不再被使用的可能性分为三类，并且从最有可能不被使用的类别开始扫描，.NET对这样的分类类别有一个称呼：代（Generation）。

GC会把所有的托管堆内的对象分为0代、1代和2代：

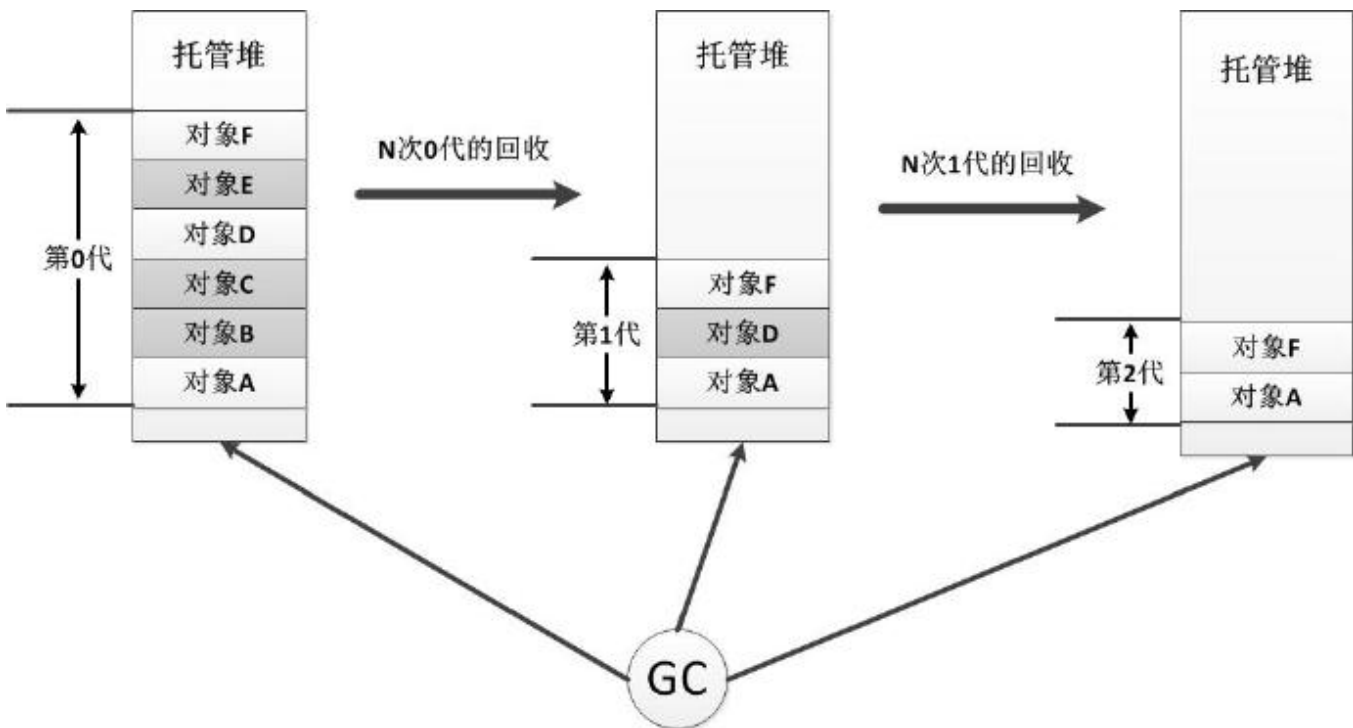
第0代，新近分配在堆上的对象，从来没有被垃圾收集过。任何一个新对象，当它第一次被分配在托管堆上时，就是第0代。

第1代，经历过一次垃圾回收后，依然保留在堆上的对象。

第2代，经历过两次或以上垃圾回收后，依然保留在堆上的对象。如果第2代对象在进行完垃圾回收后空间仍然不够用，则会抛出OutOfMemoryException异常。

对于这三代，我们需要知道的是并不是每次垃圾回收都会同时回收3个代的所有对象，越小的代拥有着越多被释放的机会。

CLR对于代的基本算法是：每执行N次0代的回收，才会执行一次1代的回收，而每执行N次1代的回收，才会执行一次2代的回收。当某个对象实例在GC执行时被发现仍然在被使用，它将被移动到下一个代中上，下图简单展示了GC对三个代的回收操作。



根据.NET的垃圾回收机制，0代、1代和2代的初始分配空间分别为256KB、2M和10M。说完分代的垃圾回收设计，也许我们会有疑问，为什么要这样弄？其实分代并不是空穴来风的设计，而是参考了这样一个事实：

一个对象实例存活的时间越长，那么它就具有更大的机率去存活更长的时间。换句话说，最有可能马上就不被使用的对象实例，往往是那些刚刚被分配的对象实例，而且新分配的对象实例通常都会被马上大量地使用。这也解释了为什么0代对象拥有最多被释放的机会，并且.NET也只为0代分配了一块只有256KB的小块逻辑内存，以使得0代对象有机会被全部放入处理器的缓存中去，这样做的结果就是使用频率最高并且最有可能马上可以被释放的对象实例拥有了最高的使用效率和最快的释放速度。

因为一次GC回收之后仍然被使用的对象会被移动到更高的代上，因此我们需要避免保留已经不再被使用的对象引用，将对象的引用置为null是告诉.NET该对象不需要再使用的最直接的方法。

在前面我们提到Finalize方法会大幅影响性能，通过结合对代的理解，我们可以知道：在带有Finalize方法的对象被回收时，该对象会被视为正在被使用从而被留在托管堆中，且至少要等一个GC循环才能被释放（为什么是至少一个？因为这取决于执行Finalize方法的线程的执行速度）。很明显，需要执行Finalize方法的那些对象实例，被真正释放时最乐观的情况下也已经位于1代的位置上了，而如果它们是在1代上才开始释放或者执行Finalize方法的线程运行得慢了一点，那该对象就在第2代上才被释放，相对于0代，这样的对象实例在堆中存留的时间将长很多。

## 2.6 GC机制中如何判断一个对象仍然在被使用？

在.NET中引用类型对象实例通常通过引用来访问，而GC判断堆中的对象是否仍然在被使用的依据也是引用。简单地说：当没有任何引用指向堆中的某个对象实例时，这个对象就被视为不再使用。

在GC执行垃圾回收时，会把引用分为以下两类：

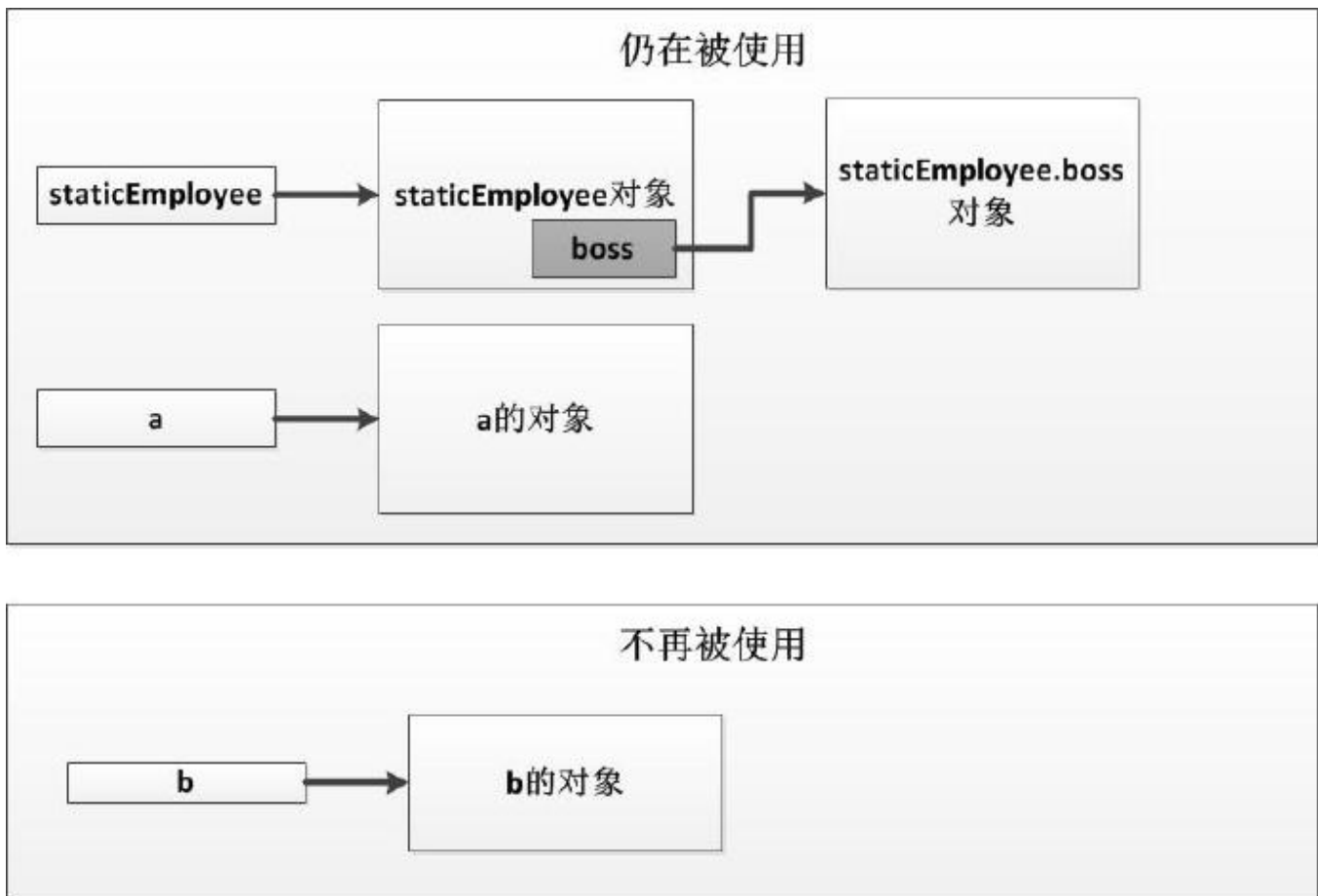
(1) 根引用：往往指那些静态字段的引用，或者存活的局部变量的引用；

(2) 非根引用：指那些不属于根引用的引用，往往是对象实例中的字段。

垃圾回收时，GC从所有仍在被使用的根引用出发遍历所有的对象实例，那些不能被遍历到的对象将被视为不再被使用而进行回收。我们可以通过下面的一段代码来直观地理解根引用和非根引用：

```
1 class Program
2 {
3     public static Employee staticEmployee;
4
5     static void Main(string[] args)
6     {
7         staticEmployee = new Employee(); // 静态变量
8         Employee a = new Employee();    // 局部变量
9         Employee b = new Employee();    // 局部变量
10        staticEmployee.boss = new Employee(); // 实例成员
11
12        Console.ReadKey();
13        Console.WriteLine(a);
14    }
15 }
16
17 public class Employee
18 {
19     public Employee boss;
20
21     public override string ToString()
22     {
23         if(boss == null)
24         {
25             return "No boss";
26         }
27
28         return "One boss";
29     }
30 }
```

上述代码中一共有两个局部变量和一个静态变量，这些引用都是根引用。而其中一个局部变量 `a` 拥有一个成员实例对象，这个引用就是一个非跟引用。下图展示了代码执行到 `Console.ReadKey()` 这行代码时运行垃圾回收时的情况。



从上图可以看出，在执行到`Console.ReadKey()`时，存活的根引用有`staticEmployee`和`a`，前者因为它是一个公共静态变量，而后者则因为后续代码还会使用到`a`。通过这两个存活的根引用，GC会找到一个非跟引用`staticEmployee.boss`，并且发现三个仍然存活的对象。而`b`的对象则将被视为不再使用从而被释放。（更简单地确保`b`对象不再被视为在被使用的方法时把`b`的引用置为`null`，即`b=null`；）

此外，当一个从根引用触发的遍历抵达一个已经被视为在使用的对象时，将结束这一个分支的遍历，这样做可以避免陷入死循环。

## 2.7 .NET中的托管堆中是否可能出现内存泄露的现象？

首先，必须明确一点：即使在拥有垃圾回收机制的.NET托管堆上，仍然是有可能发生内存泄露现象的。

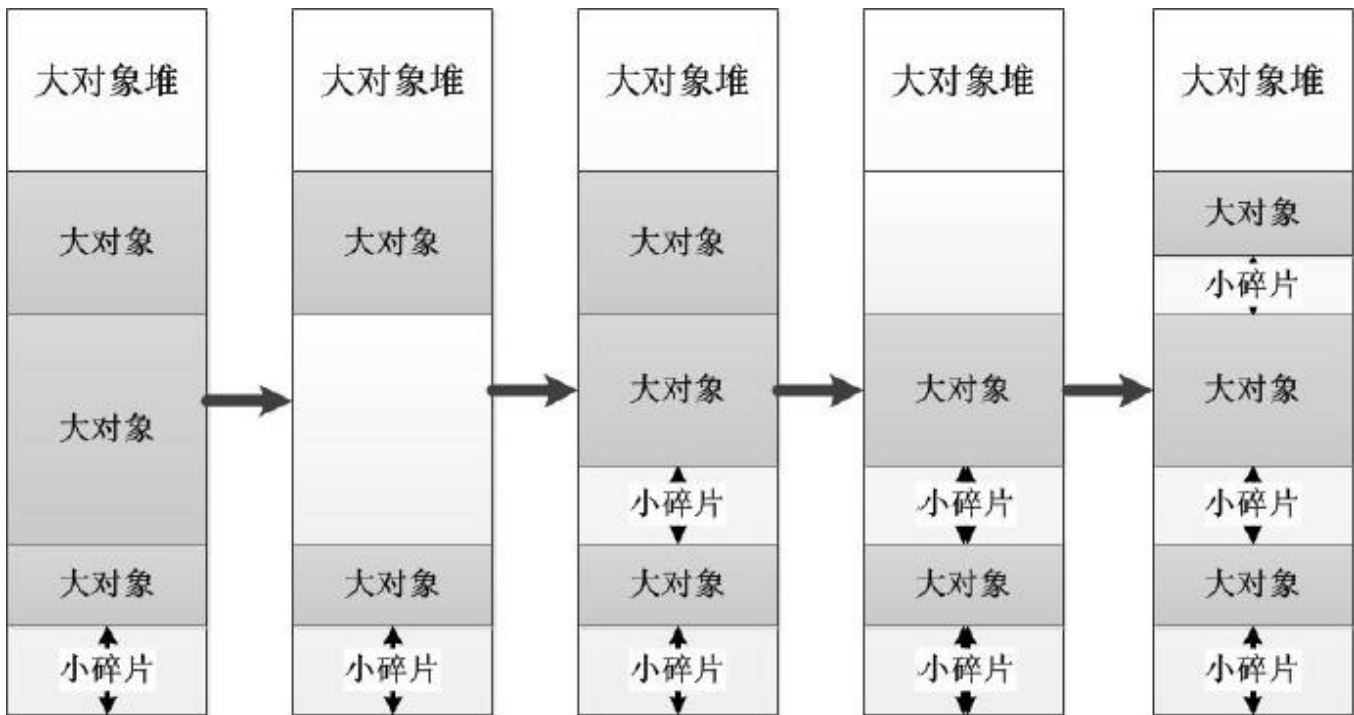
其次，什么是内存泄露？内存泄露是指内存空间上产生了不再被实际使用却又不能被分配的内存空间，其意义很广泛，像内存碎片、不彻底的对象释放等都属于内存泄露现象。内存泄露将导致主机的内存随着程序的运行而逐渐减少，无论其表现形式怎样，它的危害是很大的，因此我们需要努力地避免。

按照内存泄露的定义，我们可以知道在大部分的时候.NET中的托管堆中存在着短暂的内存泄露情况，因为对象一旦不再被使用，需要等到下一个GC时才会被释放。这里列举几个在.NET中常见的几种对系统危害较大的内存泄露情况，我们在实际开发中需要极力避免：

### （1）大对象的分配

.NET中的所有的大对象（这里主要是指对象的大小超过指定数值[85000字节]）将分配在托管堆内一个特殊的区域内，暂且将其称为“大对象堆”（这也算是CLR对于GC的一个优化策略）。大对象堆中最重要一个特点就是：没有代级的概念，所有对象都被视为第2代。在回收大对象堆内的对象时，其他的大对象不会被移动，这是考虑到大规模地移动对象需要耗费过多的资源。这样，在程序过多地分配和释放大对象之后，就会产生很多内存碎片。下图解释了这一过程：





如图所示可以看出，随着对象的分配和释放不断进行，在不进行对象移动的大对象堆内，将不可避免地产生小的内存碎片。我们所需要的就是尽量减少大对象的分配次数，尤其是那些作为局部变量的，将被大规模分配和释放的大对象，典型的例子就是String类型。

## （2）不恰当地保存根引用

最简单的一个错误例子就是不恰当地把一个对象申明为公共静态变量，一个公共的静态变量将一直被GC视为一个在使用的根引用。更糟糕的是：当这个对象内部还包含更多的对象引用时，这些对象同样不会被释放。例如下面一段代码：

```
1 public class Program
2 {
3     // 公共静态大对象
4     public static RefRoot bigObject = new RefRoot("test");
5
6     public static void Main(string[] args)
7     {
8         Console.ReadKey();
9     }
10 }
11
12 public class RefRoot
13 {
14     // 这是一个占用大量内存的成员
15     public string[] BigMember;
16
17     public RefRoot(string content)
18     {
19         // 初始化大对象
20         BigMember = new string[1000];
21         for (int i = 0; i < 1000; i++)
22         {
23             BigMember[i] = content;
24         }
25     }
26 }
27
```

在代码中，定义了一个公共静态的大对象，这个对象将直到程序运行结束后才会被GC释放掉。如果在整个程序中各个类型不断地使用这个静态成员，那这样的设计有助于减少大对象堆内的内存碎片，但是如果整个程序极少地甚至只有一次使用了这个成员，那考虑到它占用的内存会影响整体系统性能，设计时

则应该考虑设计成实例变量，以便GC能够及时释放它。

### （3）不正确的Finalize方法

前面已经介绍了Finalize方法时由GC的一个专用的线程进行调用，抛开Microsoft怎样实现的这个具体的调度算法，有一点可以肯定的是：不正确的Finalize方法将导致Finalize方法不能被正确执行。如果系统中所有的Finalize方法不能被正确执行，包含它们的对象也只能驻留在托管堆内不能被释放，这样的情况将会导致严重的后果。

那么，什么是不正确的Finalize方法？Finalize方法应该只致力于快速而简单地释放非托管资源，并且尽可能快地返回。相反，不正确的Finalize方法则可能包含以下这样的一些代码：

- ①没有保护地写文件日志；
- ②访问数据库；
- ③访问网络；
- ④把当前对象赋给某个存活的引用；

例如，当Finalize方法试图访问文件系统、数据库或者网络时，将会有资源争用和等待的潜在危险。试想一个不断尝试访问离线数据库的Finalize方法，将会在长时间内不会返回，这不仅影响了对对象的释放，也使得排在Finalize方法队列中的所有后续对象得不到释放，这个连锁反应将会导致很快地造成内存耗尽。此外，如果在Finalize方法中把对象自身又赋给了另外一个存活的引用，这时对象内的一部分资源已经被释放掉了，而另外一部分还没有，当这样一个对象被激活后，将导致不可预知的后果。

## 参考资料

- （1）朱毅，《进入IT企业必读的200个.NET面试题》
- （2）张子阳，《.NET之美：.NET关键技术深入解析》
- （3）王涛，《你必须知道的.NET》

拿高薪，还能扩大业界知名度！优秀的开发工程师看过来 -> 《[高薪招募讲师](#)》

👍 1 赞

🔖 11 收藏

💬 1 评论



## 相关文章

- [StackOverflow 这么大，它的架构是怎么样的？](#)
- [.NET 基础拾遗（3）：字符串、集合和流](#)
- [ASP.NET MVC随想录（3）：创建自定义的Middleware中间件](#)
- [WebApi接口返回值不困惑：返回值类型详解](#)
- [C# 泛型的协变和逆变](#)
- [.NET之全平台一体化的体验](#)
- [C#进阶系列——AOP？AOP！](#)
- [C#中的Lambda表达式和表达式树](#)
- [Net分布式系统之一：系统整体框架介绍](#)
- [.NET委托解析](#)

可能感兴趣的话题

- [项目实施部人手不够，就来研发部把我借走了，如果是你会闹情绪吗？](#)
- [微信公众号“小乐喵”，程序媛的生活指南，欢迎关注](#) • [Q\\_6](#)
- [妹子六月中去青岛逛逛，一周时间，有朋友有兴趣吗？](#) • [Q\\_94](#)
- [搜狐2017校园招聘实习生笔试题\(概率问题\)](#) • [Q\\_6](#)
- [房租占个人收入多大比例才比较合理？](#) • [Q\\_34](#)
- [程序员和妹子聊什么、怎么聊不会冷场？](#) • [Q\\_50](#)
- [boss选人是不是歧视女程序员？脑洞再开点，是不是虽然面试女程序员，但只是...](#) • [Q\\_28](#)
- [来公司一年了，我的工作量翻了一倍，工资一分都没加……](#) • [Q\\_21](#)
- [手机UI软件程序设计，底层程序和一些app程序设计是一回事嘛？](#) • [Q\\_13](#)
- [最近比较闲，求接 iOS 外包。免费和付费都可以哦~](#) • [Q\\_17](#)


« .NET基础拾遗（2）：面向对象的实现和异常的处理基础  
如何配置一个高效的 Mac 工作环境 »


[登录后评论](#)

[新用户注册](#)

直接登录     



最新评论



[TroubleMaker](#) (  1 )  
互联网/020

2015/10/25

很想知道装箱拆箱的那张图使用什么工具画的？

 1 赞 [回复](#) 

文章 ▼

输入搜索关键字

搜索



- [本周热门文章](#)
- [本月热门文章](#)
- [热门标签](#)

- 0 [高效 Windows 工作环境 &&...](#)
- 1 [一个程序猿必须会玩的游戏](#)
- 2 [谷歌胜诉，Android 是正当使用 Ja...](#)
- 3 [如果你很忙，你一定在什么地方做错了！](#)
- 4 [招程序员的最佳方式是这样的？](#)
- 5 [我的开源项目从0到1024的过程](#)

6 [一个 Java 程序员眼中的 Go 语言](#)

7 [为什么给类、方法、变量命名这么难？](#)

8 [计算机科学一百年](#)

9 [高效编程之道：好好休息](#)



## 业界热点资讯

[更多 »](#)



[极度简约，最小 Linux 发行版 Tiny Core Linux 7...](#)

1 天前 •  14



[即将进入稳定阶段 苹果发布 Swift 3.0 首个预览版](#)

1 天前 •  7



[IBM和思科开展合作：将沃森AI整合到边缘路由器](#)

5 小时前 •  2



[93%的钓鱼邮件旨在传播勒索软件](#)

5 小时前 •  2



[谷歌最新软件AnyPixel.js 用户自己创造交互展示](#)

7 小时前 •  3





EChart：高度可扩展，简单易用的图表库



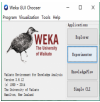
JRebel：热部署解决方案



TEAChart：一个简单直接的iOS图表库






HPPC：高效的Java集合库



Weka：一个开源的机器学习和数据挖掘软件



最新评论

- Re: [如果让我完善几年前的一个项目](#)  
学习了 非常感谢分享这些经验
- Re: [我遇到一个Bug，金额大于一千万...](#)  
仅用作格式的校验，金融交易不会的。
- Re: [我遇到一个Bug，金额大于一千万...](#)  
电脑确实比人靠谱，你和他们打交道时一旦有问题，肯定是你的问题。不像人，即使他们有问题也不轻易承认。说...



Re: [程序员提高效率的一些建议](#)

嗯，原文中的禁食是比较「奇葩」的建议，已删除咯。



Re: [一个 Java 程序员眼中的 Go...](#)

现在可以跟Java抗衡的还真没有，Java乘着android快车，所向披靡~但是Go这两年国内发展也...



Re: [我遇到一个Bug，金额大于一千万...](#)

金额也真有用double而不用decimal表示的。。。



Re: [PHP 底层的运行机制与原理](#)

就需要这样的文章, 好!1



Re: [高效 Windows 工作环境 &&a...](#)

你的建议很好，已经记录，以后会更新。

## emoi便携音响灯

音乐随身 畅听感受



### 关于伯乐在线博客

在这个信息爆炸的时代，人们已然被大量、快速并且简短的信息所包围。然而，我们相信：过多“快餐”式的阅读只会令人“虚胖”，缺乏实质的内涵。伯乐在线内容团队正试图以我们微薄的力量，把优秀的原创文章和译文分享给读者，为“快餐”添加一些“营养”元素。

快速链接

[问题反馈与求助](#) »

[加入伯乐翻译小组](#) »

[加入专栏作者](#) »

### 关注我们

新浪微博: [@伯乐在线官方微博](#)

RSS: [订阅地址](#)

推荐微信号



程序员的那些事



UI设计达人



极客范

合作联系

Email: [bd@jobbole.com](mailto:bd@jobbole.com)

QQ: 2302462408 (加好友请注明来意)

### 更多频道

[小组](#) - 好的话题、有启发的回复、值得信赖的圈子

[头条](#) - 分享和发现有价值的内容与观点



- [相亲](#) - 为IT单身男女服务的征婚传播平台
- [资源](#) - 优秀的工具资源导航
- [翻译](#) - 翻译传播优秀的外文文章
- [文章](#) - 国内外的精选文章
- [设计](#) - UI, 网页, 交互和用户体验
- [iOS](#) - 专注iOS技术分享
- [安卓](#) - 专注Android技术分享
- [前端](#) - JavaScript, HTML5, CSS
- [Java](#) - 专注Java技术分享
- [Python](#) - 专注Python技术分享

© 2016 伯乐在线

[首页](#)[博客](#)[资源](#)[小组](#)[相亲](#)[📢 反馈](#)