《你必须知道的.NET》

作者简介：王涛 微软 C# MVP，高级软件工程师，机械工程硕士，主要研究方向为.NET 底层架构和企业级系统应用。现就职于某软件公司负责架构设计、软件开发和项目管理方面的工作。作者对.NET 基础架构和 CLR 底层运行机制有浓厚的研究兴趣和造诣，熟悉 ASP.NET、XML、SQL Server

第 1 部分 渊源——.NET 与面向对象

第 1 章 OO 大智慧

1.1 对象的旅行

2 1.1 对象的旅行

3 本节将介绍以下内容：

4 — 面向对象的基本概念

5 — .NET 基本概念评述

6 — 通用类型系统

7 1.1.1 引言

8 提起面向对象，每个程序设计者都有自己的理解，有的深入肌理，有的剑走偏锋。但是无论所长，几个基本的概念总会得到大家的重视，它们是：类、对象、继承、封装和多态。很对，差不多就是这些元素构成了面向对象设计开发的基本逻辑，成为数以千万计程序设计者不懈努力去深入理解和实践的根本。而实际上，理解面向对象一个重要的方法就是以实际的生活来类比对象世界，对象世界的逻辑和我们生活的逻辑形成对比的时候，这种体验将会更有亲切感，深入程度自然也就不同以往。

9 本节就从对象这一最基本元素开始，进行一次深度的对象旅行，把.NET 面向对象世界中的主角来一次遍历式曝光。把对象的世界和人类的世界进行一些深度类比，以人类的角度戏说对象，同时也以对象的逻辑反思人类。究竟这种旅程，会有什么样的洞悉，且看本文的演义。

10 对象和人，两个世界，一样情怀。

11 1.1.2 出生

12 对象就像个体的人，生而入世，死而离世。

13 我们的故事就从对象之生开始吧。首先，看看一个对象是如何出生的：

14 Person aPerson = new Person("小王", 27);

15 那么一个人又是如何出生呢？每个婴儿随着一声啼哭来到这个世界，鼻子是鼻子、嘴巴是嘴巴，已经成为一个活生生的独立的个体。而母亲的怀胎十月是人在母体内的成长过程，母亲为胎儿提供了所有的养分和舒适的环境，这个过程就是一次实实在在的生物化构造。同样的道理，对象的出生，也是一次完整的构造过程：首先会在内存中分配一定的存储空间；然后初始化其附加成员，就像给人取个

具有标识作用的姓名一样；最后，再调用构造函数执行初始化，这样一个对象实体就完成了其出生的过程，例如上例中我们为 aPerson 对象初始化了姓名和年龄。

正如人出生之时，一身赤裸没有任何的附加品，其余的一切将随需而生，生不带来就是这个意思。对象的出生也只是完成了对必要字段的初始化操作，其他数据要通过后面的操作来完成。例如对属性赋值，通过方法获取必要的信息等。

1.1.3 旅程

婴儿一出世，由 it 成为 he or she，就意味着从此融入了复杂的社会关系，经历一次在人类伦理与社会规则的双重标准中生活，开始了为人的旅程。同理，对象也一样。

作为个体的人，首先是有类型之分的，农民、工人、学者、公务员等，所形成的社会规则就是农民在田间务农，工人在工厂生产，学者探讨知识，公务员管理国家。

对象也一样是有类型的，例如整型、字符型等等。当然，分类的标准不同，产生的类别也就不同。但是常见的分类就是值类型和引用类型两种。其依据是对象在运行时在内存中的位置，值类型位于线程的堆栈，而引用类型位于托管堆。正如农民可以进城务工，工人也可以回乡务农，值类型和引用类型的角色也会发生转变，这个过程在面向对象中称为装箱与拆箱。这一点倒是与刚刚的例子很贴切，农民进城，工人回乡，不都得把行李装进箱子里折腾嘛。

作为人，我们都是有属性的，例如你的名字、年龄、籍贯等，用来描述你的状态信息，同时每个人也用不同的行为来操作自己的属性，实现了与外界的交互。对象的字段、属性就是我们自己的标签，而方法就是操作这些标签的行为。人的名字来自于长辈，是每个人在出生之时构造的，这和对象产生时给字段赋值一样。但是每个人都有随时更名的权力，这种操作名称的行为，我们称之为方法。在面向对象中，可以像这样来完成：

```
aPerson.ChangeName("Apple Boy");
```

所以，对象的旅行过程，在某种程度上就是外界通过方法与对象交互，从而达到改变对象状态信息的过程，这也和人的生存之道暗合。

人与人之间通过语言交流。人一出生，就必然和这个世界的其他人进行沟通，形成种种相互的关系，融入这个完整的社会群体。在对象的世界里，你得绝对相信对象之间也是相互关联的，不同的对象之间发生着不同的交互性操作，那么对象的交互是通过什么方式呢？对象的交互方式被记录在一本称为“设计模式”的魔法书中，当你不解以什么样的方式建立对象与对象之间的关系时，学习前人的经验，往往是最好的选择。

下面，我们简要地分析一下对象到底旅行在什么样的世界里？

对象的生存环境是 CLR，而人的生存环境是社会。CLR 提供了对象赖以生存的托管环境，制定一系列的规则，称之为语法，例如类型、继承、多态、垃圾回收等，在对象世界里建立了真正的法制秩序；而社会提供了人行走江湖的秩序，例如法律、规范、道德等，帮助我们制约个体，维护社会。

人类社会就是系统架构，也是分层的。上层建筑代表政治和思想，通过社会契约和法律规范为经济基础服务，在对象世界中，这被称为接口。面向接口的编程就是以接口方式来抽象变化，从而形成体系。正如人类以法律手段来维系社会体系的运作和秩序一样。

由此可见，对象的旅行就是这样一个过程，在一定的约定与规则下，通过方法进行彼此的交互操作，从而达到改变本身状态的目的。从最简单的方式理解实际情况，这些体会与人的旅程如此接近，给我们的启示更加感同身受。

1.1.4 插曲

接下来，我们以与人类世界的诸多相似之处，来进一步阐释对象世界的几个最熟悉的概念。

关于继承。人的社会中，继承一般发生在有血缘关系的族群中。最直接的例子一般是，儿子继承父亲，包括姓氏、基因、财产和一切可以遗留的东西。但并不代表可以继承所有，因为父亲隐私的那一部分属于父亲独有，不可继承。当然，也可能是继承于族群的其他人，视实情而定。而在面向对象中，继承无处不在，子类继承父类，以访问权限来实现不同的控制规则，称为访问级别，如表 1-1 所示。

表 1-1 访问修改符

访问修饰符	访问权限
pubic	对访问成员没有限制，属于最高级别访问权限
protected	访问包含类或者从包含类派生的类
internal	访问仅限于程序集
protected internal	访问仅限于从包含类派生的当前程序集或类型。也就是同一个程序集的对象，或者该类及其子类可以访问
private	访问仅限于包含类型

这些规则可以以公司的体制来举例说明，将公司职权的层级与面向对象的访问权限层级做类比，应该是这样：

— public，具有最高的访问权限，就像是公司的董事会具有最高的决策权与管理权，因此 public 开放性最大，不管是否同一个程序集或者不管是否继承，都可以访问。

— protected，类似于公司业务部门经理的职责，具有对本部门的直接管辖权，在面向对象中就体现为子类继承这种纵向关系的访问约定，也就是只要继承了该类，则其对象就有访问父类的权限，而不管这两个具有继承关系的类是否在同一个程序集中。

— internal，具有类比意义的就是 internal 类似于公司的职能部门的职责，不管是否具有上下级的隶属关系，人力资源部都能管辖所有其他部门的员工考勤。这是一种横向的职责关系，在面向对象中用来

表示同一程序集的访问权限，只要是隶属于同一程序集，对象即可访问其属性，而不管是否存在隶属关系。

— protected internal，可以看做是 protected internal 的并集，就像公司中掌管职能部门的副总经理，从横向到纵向都有管理权。

— private，具有最低的访问权限，就像公司的一般员工，管好自己就行了。因此，对应于面向对象的开放性最小。

另外，对象中继承的目的是提高软件复用，而人类中的继承，不也是现实中的复用吗？

而关于多态，人的世界中，我们常常在不同的环境中表现为不同的角色，并且遵守不同的规则。例如在学校我们是学生，回到家里是儿女，而在车上又是乘客，同一个人在不同的情况下，代表了不同的身份，在家里你可以撒娇但是在学校你不可以，在学校你可以打球但在车上你不可以。所以这种身份的不同，带来的是规则的差异。在面向对象中，我们该如何表达这种复杂的人类社会学呢？

```
interface IPerson
{
    string Name
    {
        get;
        set;
    }
    Int32 Age
    {
        get;
        set;
    }
    void DoWork();
}
class PersonAtHome : IPerson
{
}
class PersonAtSchool : IPerson
{
}
class PersonOnBus : IPerson
{
```

63 }

64 显然，我们让不同角色的 Person 继承同一个接口：IPerson。然后将不同的实现交给不同角色的人自行负责，不同的是 PersonAtHome 在实现时可能是 CanBeSpoil()，而 PersonOnBus 可能是 BuyTicket()。不同的角色实现不同的规则，也就是接口协定。在使用上的规则是这个样子：

65 IPerson aPerson = new PersonAtHome();

66 aPerson.DoWork();

67 另一个角色又是这个样子：

68 IPerson bPerson = new PersonOnBus();

69 bPerson.DoWork();

70 由此带来的好处是显而易见的，我们以 IPerson 代表了不同角色的人，在不同的情况下实现了不同的操作，而把决定权交给系统自行处理。这就是多态的魅力，其乐无穷中，带来的是面向对象中最为重要的特性体验。记住，很重要的一点是，DoWork 在不同的实现类中体现为同一命名，不同的只是实现的内部逻辑。

71 这和我们的规则多么一致呀！

72 当然，有必要补充的是对象中的多态主要包括以下两种情况：

73 — 接口实现多态，就像上例所示。

74 — 抽象类实现多态，就是以抽象类来实现。

75 其细节我们将在 1.4 节“多态的艺术”中加以详细讨论。

76 由此可见，以我们自己的角度来阐释技术问题，有时候会有意想不到的收获，否则你将被淹没在诸如“为什么以这种方式来实现复用”的叫喊中不能自拔。换一个角度，眼界与思路都会更加开阔。

77 **1.1.5 消亡**

78 对象和人，有生必然有死。在对象的世界里，它的生命是由 GC 控制的，而在人的世界里我们把 GC 称为自然规律。进入死循环的对象，是违反规则的，必然无法逃脱被 Kill 的命运，就如同没有长生不死的人一样。

79 在这一部分，我们首先观察对象之死，以此反思和体味人类入世的哲学，两者相比较，也会给我们更多关于自己的启示。对象的生命周期由 GC 控制，其规则大概是这样：GC 管理所有的托管堆对象，当内存回收执行时，GC 检查托管堆中不再被使用的对象，并执行内存回收操作。不被应用程序使用的对象，指的是对象没有任何引用。关于如何回收、回收的时刻，以及遍历可回收对象的算法，是较为复杂的问题，我们将在 5.3 节“垃圾回收”中进行深度探讨。不过，这个回收的过程，同样使我们感慨。大自然就是那个看不见的 GC，造物而又终将万物回收，无法改变。我们所能做到的是，将生命的周期拓宽、延长、书写得更加精彩。

80 **1.1.6** 结论

81 程序世界其实和人类世界有很多相似的地方，本节就以这种类比的方式来诠释这两个世界的主角：对象和人。以演化推进的手法来描述面向对象程序世界的主角对象由生而死的全过程，好似复杂的人生。而其实，人也可以是简单的。这是一种相互的较量，也是一种相互的借鉴。

对象创建始末（上）

本文将介绍以下内容：

- 对象的创建过程
- 内存分配分析
- 内存布局研究

1. 引言

了解.NET 的内存管理机制，首先应该从内存分配开始，也就是对象的创建环节。对象的创建，是个复杂的过程，主要包括内存分配和初始化两个环节。例如，对象的创建过程可以表示为：

```
FileStream fs = new FileStream(@"C:"temp.txt", FileMode.Create);
```

通过 new 关键字操作，即完成了对 FileStream 类型对象的创建过程，这一看似简单的操作背后，却经历着相当复杂的过程和周折。

本篇全文，正是对这一操作背后过程的详细讨论，从中了解.NET 的内存分配是如何实现的？

2. 内存分配

关于内存的分配，首先应该了解分配在哪里的问题。CLR 管理内存的区域，主要有三块，分别为：

- 线程的堆栈，用于分配值类型实例。堆栈主要由操作系统管理，而不受垃圾收集器的控制，当值类型实例所在方法结束时，其存储单位自动释放。栈的执行效率高，但存储容量有限。
- GC 堆，用于分配小对象实例。如果引用类型对象的实例大小小于 85000 字节，实例将被分配在 GC 堆上，当有内存分配或者回收时，垃圾收集器可能会对 GC 堆进行压缩，详情见后文讲述。

- LOH（Large Object Heap）堆，用于分配大对象实例。如果引用类型对象的实例大小不小于 85000 字节时，该实例将被分配到 LOH 堆上，而 LOH 堆不会被压缩，而且只在完全 GC 回收时被回收。

本文讨论的重点是.NET 的内存分配机制，因此下文将不加说明的以 GC 堆上的分配为例来展开。关于值类型和引用类型的论述，请参见[第八回：品味类型---值类型与引用类型（上）－内存有理]。

了解了内存分配的区域，接着我们看看有哪些操作将导致对象创建和内存分配的发生，关于实例创建有多个 IL 指令解析，主要包括：

- newobj，用于创建引用类型对象。
- ldstr，用于创建 string 类型对象。
- newarr，用于分配新的数组对象。
- box，在值类型转换为引用类型对象时，将值类型字段拷贝到托管堆上发生的内存分配。

在上述论述的基础上，下面从堆栈的内存分配和托管堆的内存分配两个方面来分别论述.NET 的内存分配机制。

2.1 堆栈的内存分配机制

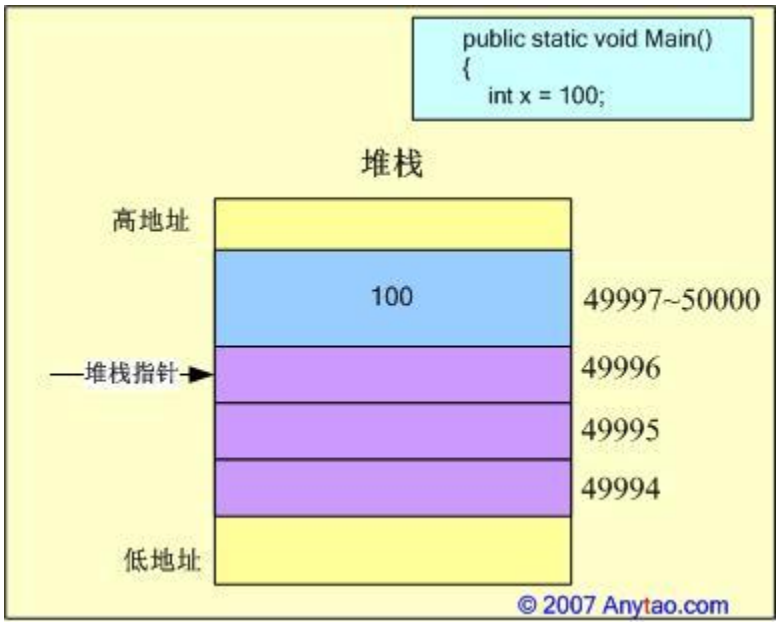
对于值类型来说，一般创建在线程的堆栈上。但并非所有的值类型都创建在线程的堆栈上，例如作为类的字段时，值类型作为实例成员的一部分也被创建在托管堆上；装箱发生时，值类型字段也会拷贝在托管堆上。

对于分配在堆栈上的局部变量来说，操作系统维护着一个堆栈指针来指向下一个自由空间的地址，并且堆栈的内存地址是由高位到低位向下填充。以下例而言：

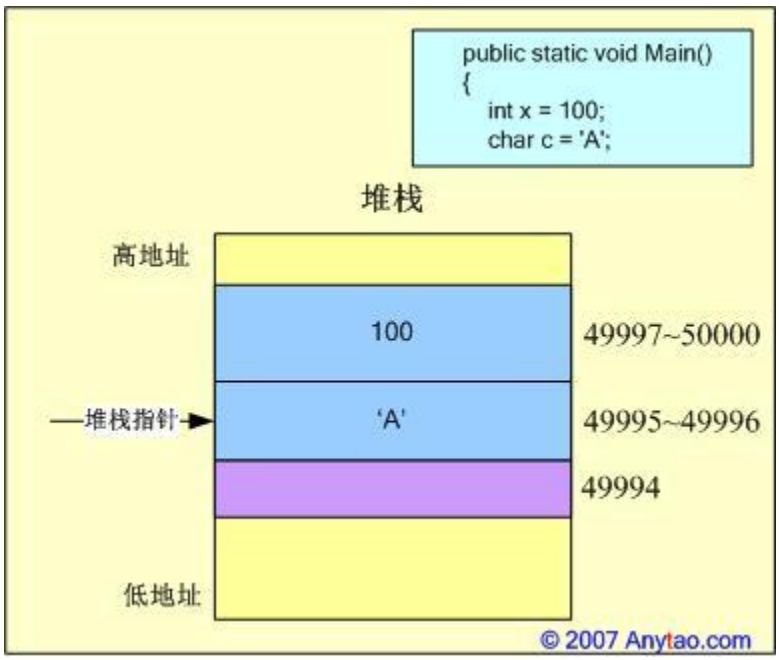
```
public static void Main()
{
    int x = 100;
    char c = 'A';
}
```

假设线程栈的初始化地址为 50000，因此堆栈指针首先指向 50000 地址空间。代码由入口函数 Main 开始执行，首先进入作用域的是整型局部变量 x，它将在栈上分配 4Byte 的内存空间，因此堆栈指针向下移动 4 个字节，则值 100 将保存在 49997~50000 单位，而堆栈指针表示的下一个自由空间地址为

49996，如图所示：

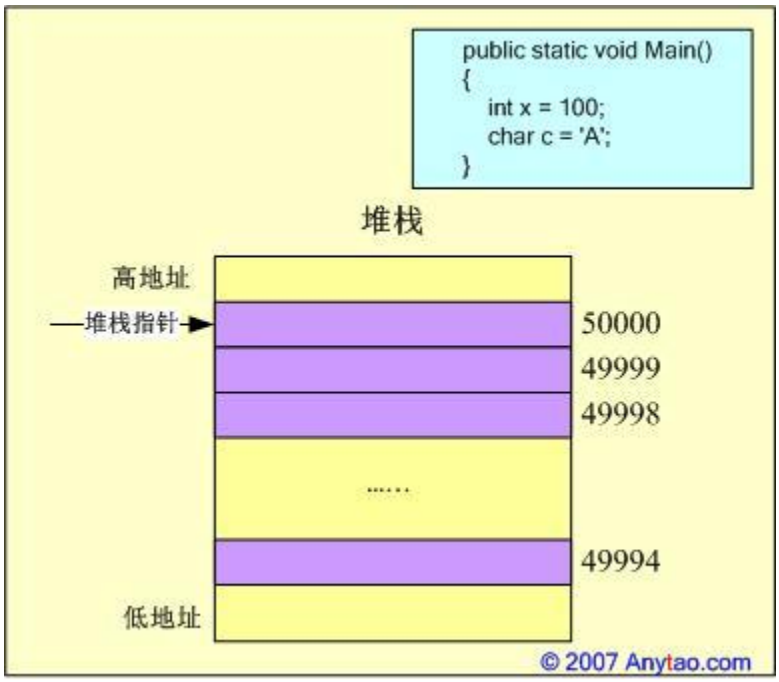


接着进入下一行代码，将为字符型变量 c 分配 2Byte 的内存空间，堆栈指针向下移动 2 个字节至 49994 单位，值'A'会保存在 49995~49996 单位，地址的分配如图：



最后，执行到 Main 方法的右括号，方法体执行结束，变量 x 和 c 的作用域也随之结束，需要删除变量 x 和 c 在堆栈内存中的值，其释放过程和分配过程刚好相反：首先删除 c 的内存，堆栈指针向上递增 2 个字节，然后删除 x 的内存，堆栈指针继续向上递增 4 个字节，程序执行结

束，此时的内存状况为：



其他较复杂的分配过程，可能在作用域和分配大小上有所不同，但是基本过程大同小异。栈上的内存分配，效率较高，但是内存容量不大，同时变量的生存周期随着方法的结束而消亡。

未完待续：托管堆的内存分配机制和必要的补充说明，近期发布，敬请关注。

第十九回：对象创建始末（下）

本文将介绍以下内容：

- 对象的创建过程
- 内存分配分析
- 内存布局研究

接上回[第十八回：对象创建始末（上）]，继续对对象创建话题的讨论>>>

2.2 托管堆的内存分配机制

引用类型的实例分配于托管堆上，而线程栈却是对象生命周期开始的地方。对 32 位处理器来说，应用程序完成进程初始化后，CLR 将在进程的可用地址空间上分配一块保留的地址空间，它是进程（每个进程可使用 4GB）中可用地址空间上的一块内存区域，但并不对应于任何物理内存，这块地址空间即是托管堆。

托管堆又根据存储信息的不同划分为多个区域，其中最重要的是垃圾回收堆（GC Heap）和加载堆（Loader Heap），GC Heap 用于存储对象实例，受 GC 管理；Loader Heap 又分为 High-Frequency Heap、Low-Frequency Heap 和 Stub Heap，不同的堆上又存储不同的信息。Loader Heap 最重要的信息就是元数据相关的信息，也就是 Type 对象，每个 Type 在 Loader Heap 上体现为一个 Method Table（方法表），而 Method Table 中则记录了存储的元数据信息，例如基类型、静态字段、实现的接口、所有的方法等等。Loader Heap 不受 GC 控制，其生命周期为从创建到 AppDomain 卸载。

在进入实际的内存分配分析之前，有必要对几个基本概念做以交代，以便更好的在接下来的分析中展开讨论。

- TypeHandle，类型句柄，指向对应实例的方法表，每个对象创建时都包含该附加成员，并且占用 4 个字节的内存空间。我们知道，每个类型都对应于一个方法表，方法表创建于编译时，主要包含了类型的特征信息、实现的接口数目、方法表的 slot 数目等。
- SyncBlockIndex，用于线程同步，每个对象创建时也包含该附加成员，它指向一块被称为 Synchronization Block 的内存块，用于管理对象同步，同样占用 4 个字节的内存空间。
- NextObjPtr，由托管堆维护的一个指针，用于标识下一个新建对象分配时在托管堆中所处的位置。CLR 初始化时，NextObjPtr 位于托管堆的基地址。

因此，我们对引用类型分配过程应该有个基本的了解，由于本篇示例中 FileStream 类型的继承关系相对复杂，在此本文实现一个相对简单的类型来做说明：

```
//@ 2007 Anytao.com
//http://www.anytao.com
    public class UserInfo
    {
        private Int32 age = -1;
```

```
        private char level = 'A';
    }

    public class User
    {
        private Int32 id;
        private UserInfo user;
    }

    public class VIPUser : User
    {
        public bool isVip;

        public bool IsVipUser()
        {
            return isVip;
        }

        public static void Main()
        {
            VIPUser aUser;
            aUser = new VIPUser();
            aUser.isVip = true;
            Console.WriteLine(aUser.IsVipUser());
        }
    }
```

将上述实例的执行过程，反编译为 IL 语言可知：new 关键字被编译为 newobj 指令来完成对象创建工作，进而调用类型的构造器来完成其初始化操作，在此我们详细的描述其执行的具体过程：

- 首先，将声明一个引用类型变量 aUser：

```
        VIPUser aUser;
```

它仅是一个引用（指针），保存在线程的堆栈上，占用 4Byte 的内存空间，将用于保存 VIPUser 对象的有效地址，其执行过程正是上文描述的在线程栈上的分配过程。此时 aUser 未指向任何有效的实例，因此被自行初始化为 null，试图对 aUser 的任何操作将抛出 NullReferenceException 异常。

- 接着，通过 new 操作执行对象创建：

```
aUser = new VIPUser();
```

如上文所言，该操作对应于执行 newobj 指令，其执行过程又可细分为以下几步：

（a）CLR 按照其继承层次进行搜索，计算类型及其所有父类的字段，该搜索将一直递归到 System.Object 类型，并返回字节总数，以本例而言类型 VIPUser 需要的字节总数为 15Byte，具体计算为：VIPUser 类型本身字段 isVip（bool 型）为 1Byte；父类 User 类型的字段 id（Int32 型）为 4Byte，字段 user 保存了指向 UserInfo 型的引用，因此占 4Byte，而同时还要为 UserInfo 分配 6Byte 字节的内存。

实例对象所占的字节总数还要加上对象附加成员所需的字节总数，其中附加成员包括 TypeHandle 和 SyncBlockIndex，共计 8 字节（在 32 位 CPU 平台下）。因此，需要在托管堆上分配的字节总数为 23 字节，而堆上的内存块总是按照 4Byte 的倍数进行分配，因此本例中将分配 24 字节的地址空间。

（c）CLR 在当前 AppDomain 对应的托管堆上搜索，找到一个未使用的 20 字节的连续空间，并为其分配该内存地址。事实上，GC 使用了非常高效的算法来满足该请求，NextObjPtr 指针只需要向前推进 20 个字节，并清零原 NextObjPtr 指针和当前 NextObjPtr 指针之间的字节，然后返回原 NextObjPtr 指针地址即可，该地址正是新创建对象的托管堆地址，也就是 aUser 引用指向的实例地址。而此时的 NextObjPtr 仍指向下一个新建对象的位置。注意，栈的分配是向低地址扩展，而堆的分配是向高地址扩展。

另外，实例字段的存储是有顺序的，由上到下依次排列，父类在前子类在后，详细的分析请参见[第十五回：继承本质论]。

在上述操作时，如果试图分配所需空间而发现内存不足时，GC 将启动垃圾收集操作来回收垃圾对象所占的内存，我们将以后对此做详细的分析。

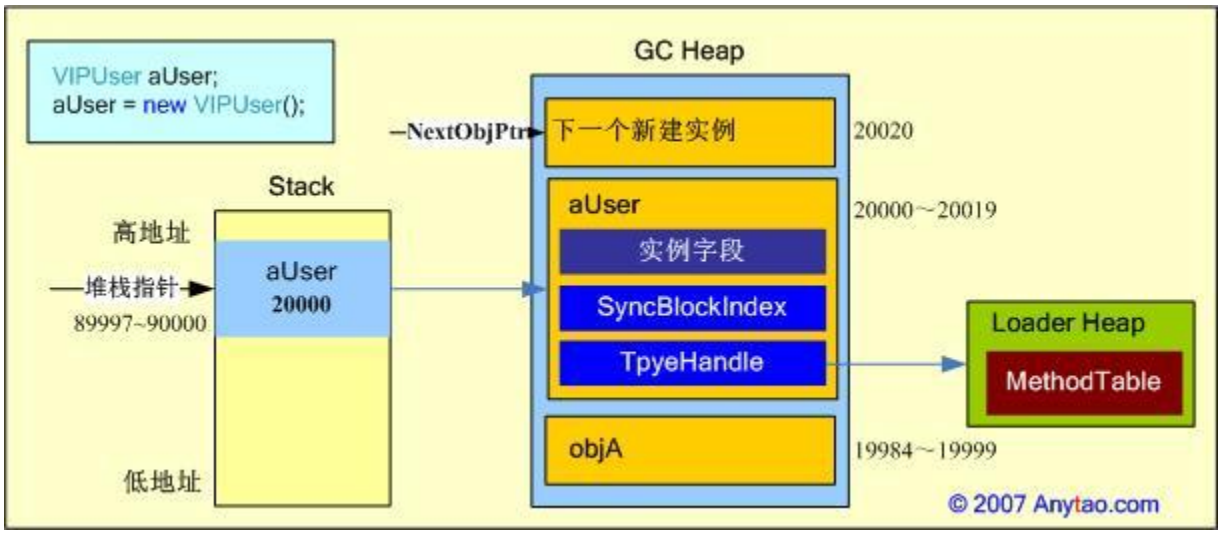
- 最后，调用对象构造器，进行对象初始化操作，完成创建过程。该构造过程，又可细分为以下几个环节：

（a）构造 VIPUser 类型的 Type 对象，主要包括静态字段、方法表、实现的接口等，并将其分配在上文提到托管堆的 Loader Heap 上。

（b）初始化 aUser 的两个附加成员：TypeHandle 和 SyncBlockIndex。将 TypeHandle 指针指向 Loader Heap 上的 MethodTable，CLR 将根据 TypeHandle 来定位具体的 Type；将 SyncBlockIndex 指针指向 Synchronization Block 的内存块，用于在多线程环境下对实例对象的同步操作。

（c）调用 VIPUser 的构造器，进行实例字段的初始化。实例初始化时，会首先向上递归执行父类初始化，直到完成 System.Object 类型的初始化，然后再返回执行子类的初始化，直到执行 VIPUser 类为止。以本例而言，初始化过程为首先执行 System.Object 类，再执行 User 类，最后才是 VIPUser 类。最终，newobj 分配的托管堆的内存地址，被传递给 VIPUser 的 this 参数，并将其引用传给栈上声明的 aUser。

上述过程，基本完成了一个引用类型创建、内存分配和初始化的整个流程，然而该过程只能看作是一个简化的描述，实际的执行过程更加复杂，涉及到一系列细化的过程和操作。对象创建并初始化之后，内存的布局，可以表示为：



由上文的分析可知，在托管堆中增加新的实例对象，只是将 NextObjPtr 指针增加一

定的数值，再次新增的对象将分配在当前 NextObjPtr 指向的内存空间，因此在托管堆栈中，连续分配的对象在内存中一定是连续的，这种分配机制非常高效。

2.3 必要的补充

有了对象创建的基本流程概念，下面的几个问题时常引起大家的思考，在此本文一并做以探索：

- 值类型中的引用类型字段和引用类型中的值类型字段，其分配情况又是如何？

这一思考其实是一个问题的两个方面：对于值类型嵌套引用类型的情况，引用类型变量作为值类型的成员变量，在堆栈上保存该成员的引用，而实际的引用类型仍然保存在 GC 堆上；对于引用类型嵌套值类型的情况，则该值类型字段将作为引用类型实例的一部分保存在 GC 堆上。在[第八回：品味类型---值类型与引用类型（上）－内存有理]一文对这种嵌套结构，有较详细的分析。对于值类型，你只要记着它总是分配在声明它的地方。

- 方法保存在 Loader Heap 的 MethodTable 中，那么方法调用时又是怎么样的过程？

如上文所言，MethodTable 中包含了类型的元数据信息，类在加载时会在 Loader Heap 上创建这些信息，一个类型在内存中对应一份 MethodTable，其中包含了所有的方法、静态字段和实现的接口信息等。对象实例的 TypeHandle 在实例创建时，将指向 MethodTable 开始位置的偏移处（默认偏移 12Byte），通过对象实例调用某个方法时，CLR 根据 TypeHandle 可以找到对应的 MethodTable，进而可以定位到具体的方法，再通过 JIT Compiler 将 IL 指令编译为本地 CPU 指令，该指令将保存在一个动态内存中，然后在该内存地址上执行该方法，同时该 CPU 指令被保存起来用于下一次的执行。

在 MethodTable 中，包含一个 Method Slot Table，称为方法槽表，该表是一个基于方法实现的线性链表，并按照以下顺序排列：继承的虚方法，引入的虚方法，实例方法和静态方法。方法表在创建时，将按照继承层次向上搜索父类，直到 System.Object 类型，如果子类覆写了父类方法，则将会以子类方法覆盖父类虚方法。关于方法表的创建过程，可以参考[第十五回：继承本质论]中的描述。

- 静态字段的内存分配和释放，又有何不同？

静态字段也保存在方法表中，位于方法表的槽数组后，其生命周期为从创建到 AppDomain 卸载。因此一个类型无论创建多少个对象，其静态字段在内存中也只有一份。静态字段只能由静态构造函数进行初始化，静态构造函数确保在类型任何对象创建前，或者在任何静态字段或方法被引用前执行，其详细的执行顺序请参考相关讨论。

3. 结论

对象创建过程的了解，是从底层接触 CLR 运行机制的入口，也是认识.NET 自动内存管理的关键。通过本文的详细论述，关于对象的创建、内存分配、初始化过程和方法调用等技术都会建立一个相对全面的理解，同时也清楚的把握了线程栈和托管堆的执行机制。

对象总是有生有灭，本文简述其生，这是个伟大的开始。

1.2 什么是继承

本节将介绍以下内容：

— 什么是继承？

— 继承的实现本质

— 继承的分类与规则

— 继承与聚合

— 继承的局限

1.2.1 引言

继承，一个熟悉而容易产生误解的话题。这是大部分人对继承最直观的感受。说它熟悉，是因为作为面向对象的三大要素之一的继承，每个技术研究者都会在职业生涯中不断地重复关于继承的话题；说它容易产生误解，是因为它总是和封装、多态交织在一起，形成复杂的局面。以继

承为例，如何理清多层继承的机制，如何了解实现继承与接口继承的异同，如何体会继承与多态的关系，似乎都不是件简单的事情。

本节希望将继承中最为头疼，最为复杂的问题统统拿出来晒一晒，以防时间久了，不知不觉在使用者那里发霉生虫。

本节不会花太多笔墨做系统性的论述，如有需要请参考其他技术专著上更详细的分析。我们将从关于继承的热点出发，逐个击破，最后总结规律，期望用这种方式实现对继承全面的了解，让你掌握什么才是继承。

1.2.2 基础为上

正如引言所述，继承是个容易产生误解的技术话题。那么，对于继承，就应该着手从这些容易误解与引起争论的话题来寻找关于全面认识和了解继承的答案。一点一滴摆出来，最后再对分析的要点做归纳，形成一种系统化认识。这是一种探索问题的方式，用于剖析继承这一话题真是再恰当不过了。

不过，解密之前，我们还是按照技术分析的惯例，从基本出发，以简洁的方式来快速了解关于继承最基本的概念。首先，认识一张比较简单的动物分类图（图 1-1），以便引入我们对继承概念的介绍。

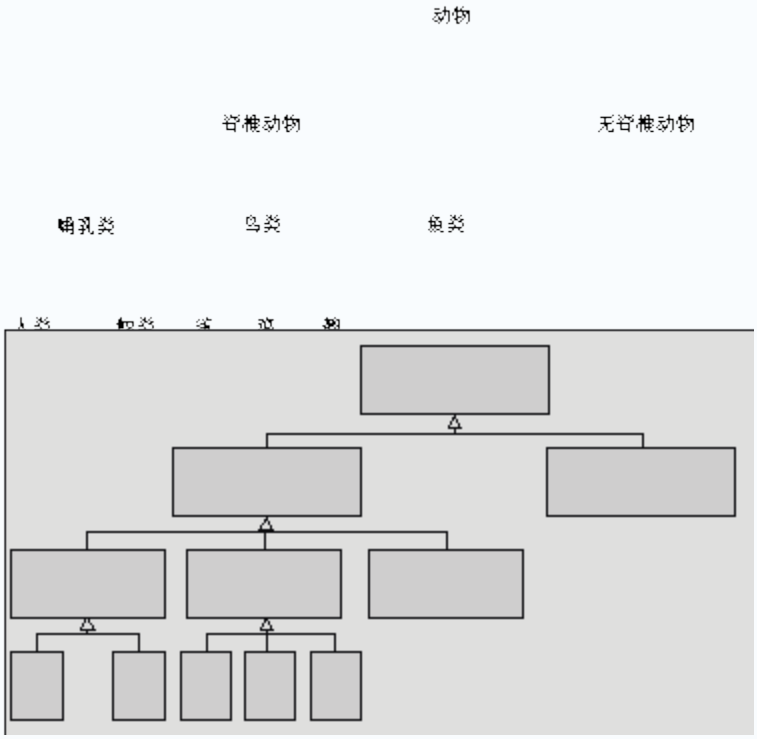


图 1-1 继承关系图

从图 1-1 中，我们可以获得的信息包括：

— 动物继承关系是以一定的分类规则进行的，将相同属性和特征的动物及其类别抽象为一类，类别与类别之间的关系反映为对相似或者对不相似的某种抽象关系，例如鸟类一般都能飞，而鱼类一般都生活在水中。

— 位于继承图下层的类别继承了上层所有类别的特性，形成一种 IS-A 的关系，例如我们可以说，人类 IS-A 哺乳类、人类 IS-A 脊椎类。但是这种关系是单向的，所以我们不能说鸟类 IS-A 鸡。

— 动物继承图自上而下是一种逐层具体化过程，而自下而上是一种逐层抽象化过程，这种抽象化关系反映为上下层之间的继承关系。例如，最高层的动物具有最普遍的特征，而最低层的人则具有较具体的特征。

— 下层类型只能从上层类型中的某一个类别继承，例如鲸类的上层只能是哺乳类一种，因此是一种单继承形式。

— 这种继承关系中，层与层的特性是向下传递的，例如鸟类具有脊椎类的特征，鹤类也具有脊椎类的特征，而所有的类都具有动物的特征，因此说动物是这个层次关系的根。

我们将这种现实世界的对象抽象化，就形成了面向对象世界的继承机制。因此，关于继承，我们可以定义为：

继承，就是面向对象中类与类之间的一种关系。继承的类称为子类、派生类，而被继承类称为父类、基类或超类。通过继承，使得子类具有父类的属性和方法，同时子类也可以通过加入新的属性和方法或者修改父类的属性和方法建立新的类层次。

继承机制体现了面向对象技术中的复用性、扩展性和安全性。为面向对象软件开发与模块化软件架构提供了最基本的技术基础。

在.NET 中，继承按照其实现方式的不同，一般分类如下。

— 实现继承：派生类继承了基类的所有属性和方法，并且只能有一个基类，在.NET 中 System.Object 是所有类型的最终基类，这种继承方式称为实现继承。

— 接口继承：派生类继承了接口的方法签名。不同于实现继承的是，接口继承允许多继承，同时派生类只继承了方法签名而没有方法实现，具体的实现必须在派生类中完成。因此，确切地说，这种继承方式应该称为接口实现。

CLR 支持实现单继承和接口多继承。本节重点关注对象的实现继承，关于接口继承，我们将在 1.5 节“玩转接口”中做详细论述。另外，值得关注的是继承的可见性问题，.NET 通过访问权限来实现不同的控制规则，这些访问修饰符主要包括：public、protected、internal 和 private。

下面，我们就以动物继承情况为例，实现一个最简单的继承实例，如图 1-2 所示。

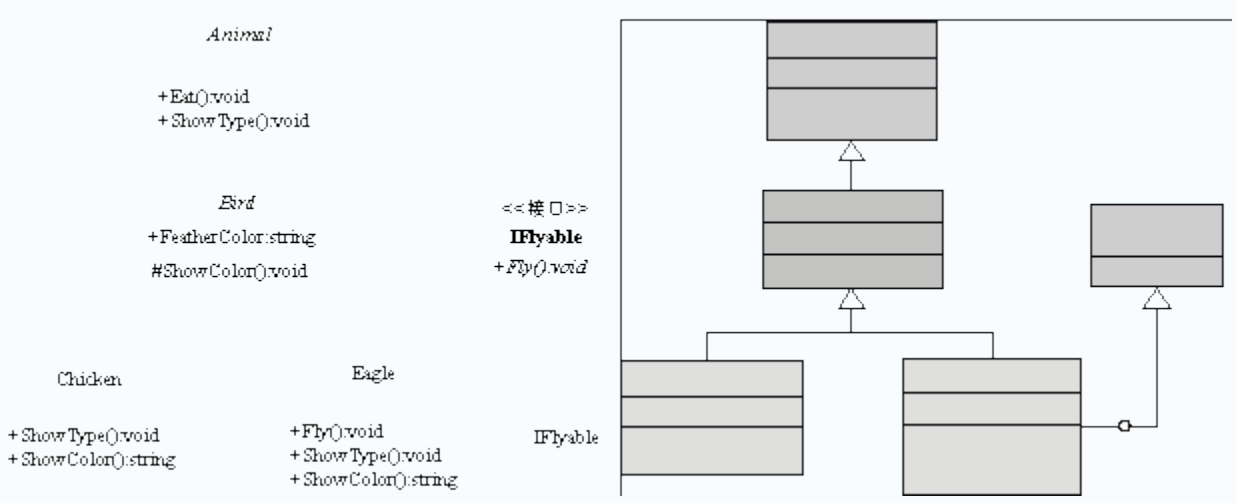


图 1-2 动物系统 UML

在这个继承体系中，我们实现了一个简单的三层继承层次，Animal 类是所有类型的基类，在此将其构造为抽象类，抽象了所有类型的普遍特征行为：Eat 方法和 ShowType 方法，其中 ShowType 方法为虚函数，其具体实现在子类 Chicken 和 Eagle 中给出。这种在子类中实现虚函数的方式，称为方法的动态绑定，是实现面向对象另一特性：多态的基本机制。另外，Eagle 类实现了接口继承，使得 Eagle 实例可以实现 Fly 这一特性，接口继承的优点是显而易见的：通过 IFlyable 接口，实现了对象与行为的分离，这样我们无需担心因为继承不当而使 Chicken 有 Fly 的能力，保护了系统的完整性。

从图 1-2 所示的 UML 图中可知，通过继承我们轻而易举地实现了代码的复用和扩展，同时通过重载（overload）、覆写（override）、接口实现等方式实现了封装变化，隐藏私有信息等面向对象的基本规则。通过继承，轻易地实现了子类对父类共性的继承，例如，Animal 类中实现了方法 Eat()，那么它的所有子类就都具有了 Eat()特性。同时，子类也可以实现对基类的扩展和改写，主要有两种方式：一是通过在子类中添加新方法，例如 Bird 类中就添加了新方法 ShowColor 用于现实鸟类的毛色；二是通过对父类方法的重新改写，在.NET 中称为覆写，例如 Eagle 类中的 ShowColor()方法。

1.2.3 继承本质论

了解了关于继承的基本概念，我们回归本质，从编译器运行的角度来揭示.NET 继承中的运行本源，来发现子类对象如何实现对父类成员与方法的继承，以简单的示例揭示继承的实质，来阐述继承机制是如何被执行的。

```
public abstract class Animal
{
    public abstract void ShowType();
    public void Eat()
    {
        Console.WriteLine("Animal always eat.");
    }
}
public class Bird: Animal
{
    private string type = "Bird";
    public override void ShowType()
    {
        Console.WriteLine("Type is {0}", type);
    }
    private string color;
    public string Color
    {
        get { return color; }
        set { color = value; }
    }
}
public class Chicken : Bird
{
    private string type = "Chicken";
    public override void ShowType()
    {
        Console.WriteLine("Type is {0}", type);
```

```
    }
    public void ShowColor()
    {
        Console.WriteLine("Color is {0}", Color);
    }
}
```

然后，在测试类中创建各个类对象，由于 Animal 为抽象类，我们只创建 Bird 对象和 Chicken 对象。

```
public class TestInheritance
{
    public static void Main()
    {
        Bird bird = new Bird();
        Chicken chicken = new Chicken();
    }
}
```

下面我们从编译角度对这一简单的继承示例进行深入分析，从而了解.NET 内部是如何实现我们强调的继承机制的。

（1）我们简要地分析一下对象的创建过程：

```
Bird bird = new Bird();
```

Bird bird 创建的是一个 Bird 类型的引用，而 new Bird()完成的是创建 Bird 对象，分配内存空间和初始化操作，然后将这个对象引用赋给 bird 变量，也就是建立 bird 变量与 Bird 对象的关联。

（2）我们从继承的角度来分析 CLR 在运行时如何执行对象的创建过程，因为继承的本质正体现于对象的创建过程中。

在此我们以 Chicken 对象的创建为例，首先是字段，对象一经创建，会首先找到其父类 Bird，并为其字段分配存储空间，而 Bird 也会继续找到其父类 Animal，为其分配存储空间，依次类推直到递归结束，也就是完成 System.Object 内存分配为止。我们可以在编译器中用单步执行的方法来大致了解其分配的过程和顺序，因此，对象的创建过程是按照顺序完成了对整个父类及其本身字

段的内存创建，并且字段的存储顺序是由上到下排列，最高层类的字段排在最前面。其原因是如果父类和子类出现了同名字段，则在子类对象创建时，编译器会自动认为这是两个不同的字段而加以区别。

然后，是方法表的创建，必须明确的一点是方法表的创建是类第一次加载到 AppDomain 时完成的，在对象创建时只是将其附加成员 TypeHandle 指向方法列表在 Loader Heap 上的地址，将对象与其动态方法列表相关联起来，因此方法表是先于对象而存在的。类似于字段的创建过程，方法表的创建也是父类在先子类在后，原因是显而易见的，类 Chicken 生成方法列表时，首先将 Bird 的所有方法复制一份，然后和 Chicken 本身的方法列表做对比，如果有覆写的虚方法则以子类方法覆盖同名的父类方法，同时添加子类的新方法，从而创建完成 Chicken 的方法列表。这种创建过程也是逐层递归到 Object 类，并且方法列表中也是按照顺序排列的，父类在前子类在后，其原因和字段大同小异，留待读者自己体味。不言而喻，任何类型方法表中，开始的 4 个方法总是继承自 System.Object 类型的虚方法，它们是：ToString、Equals、GetHashCode 和 Finalize，详见 8.1 节“万物归宗：System.Object”所述。

结合我们的分析过程，现在将对象创建的过程以图例来揭示其在内存中的分配情形，如图 1-3 所示。

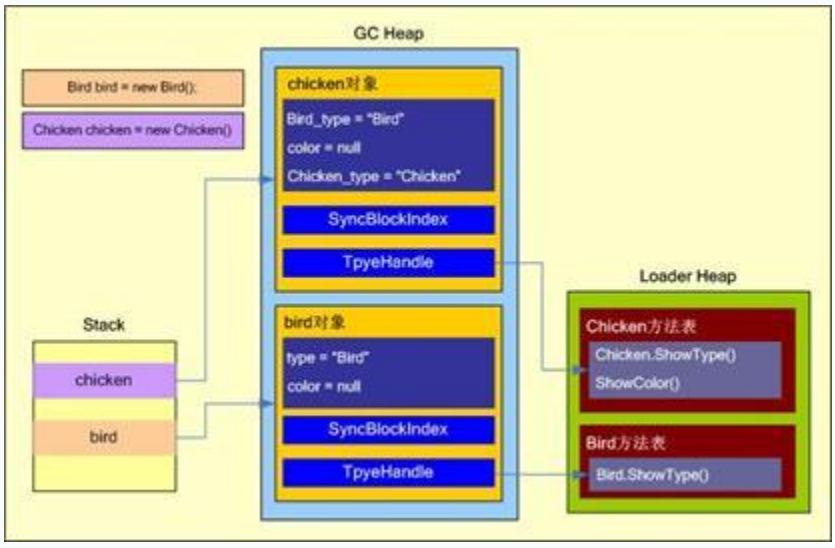


图 1-3 对象创建内存概括

从我们的分析和上面的对象创建过程中，我们应对继承的本质有了以下更明确的认识：

— 继承是可传递的，子类是对父类的扩展，必须继承父类方法，同时可以添加新方法。

— 子类可以调用父类方法和字段，而父类不能调用子类方法和字段。

— 虚方法如何实现覆写操作，使得父类指针可以指向子类对象成员。

— 子类不光继承父类的公有成员，同时继承了父类的私有成员，只是在子类中不被访问。

— new 关键字在虚方法继承中的阻断作用。

你是否已经找到了理解继承、理解动态编译的不二法门？

通过上面的讲述与分析，我们基本上对.NET 在编译期的实现原理有了大致的了解，但是还有以下的问题，可能会引起疑惑，那就是：

```
Bird bird2 = new Chicken();
```

这种情况下，bird2.ShowType 应该返回什么值呢？而 bird2.type 又该是什么值呢？有两个原则，是.NET 专门用于解决这一问题的。

— 关注对象原则：调用子类还是父类的方法，取决于创建的对象是子类对象还是父类对象，而不是它的引用类型。例如 Bird bird2 = new Chicken()时，我们关注的是其创建对象为 Chicken 类型，因此子类将继承父类的字段和方法，或者覆写父类的虚方法，而不用关注 bird2 的引用类型是否为 Bird。引用类型的区别决定了不同的对象在方法表中不同的访问权限。


注意

根据关注对象原则，下面的两种情况又该如何区别呢？

```
Bird bird2 = new Chicken();
Chicken chicken = new Chicken();
```

根据上文的分析，bird2 对象和 chicken 对象在内存布局上是一样的，差别就在于其引用指针的类型不同：bird2 为 Bird 类型指针，而 chicken 为 Chicken 类型指针。以方法调用为例，不同的类型指针在虚拟方法表中有不同的附加信息作为标志来区别其访问的地址区域，称为 offset。不同类型的指针只能在其特定地址区域内执行，子类覆盖父类时会保证其访问地址区域的一致性，从而解决了不同的类型访问具有不同的访问权限问题。

— 执行就近原则：对于同名字段或者方法，编译器是按照其顺序查找来引用的，也就是首先访问离它创建最近的字段或者方法，例如上例中的 bird2，是 Bird 类型，因此会首先访问 Bird_type（注意编译器是不会重新命名的，在此是为区分起见），如果 type 类型设为 public，则在此将返回"Bird"值。这也就是为什么在对象创建时必须将字段按顺序排列，而父类要先于子类编译的原因了。


思考

1．上面我们分析到 bird2.type 的值是"Bird"，那么 bird2.ShowType()会显示什么值呢？答案是"Type is Chicken"，根据上面的分析，想想到底为什么？

2．关于 new 关键字在虚方法动态调用中的阻断作用，也有了更明确的理论基础。在子类方法中，如果标记 new 关键字，则意味着隐藏基类实现，其实就是创建了与父类同名的另一个方法，在编译中这两个方法处于动态方法表的不同地址位置，父类方法排在前面，子类方法排在后面。

1.2.4 密境追踪

通过对继承的基本内容的讨论和本质揭示，是时候将我们的眼光转移到继承应用中的热点问题了，主要是从面向对象的角度对继承进行讨论，就像追踪继承中的密境，在迷失的森林中寻找出口。

1．实现继承与接口继承

实现继承通常情况下表现为对抽象类的继承，而其与接口继承在规则上有以下几点归纳：

— 抽象类适合于有族层概念的类间关系，而接口最适合为不同的类提供通用功能。

— 接口着重于 CAN-DO 关系类型，而抽象类则偏重于 IS-A 式的关系。

— 接口多定义对象的行为；抽象类多定义对象的属性。

— 如果预计会出现版本问题，可以创建"抽象类"。例如，创建了狗（Dog）、鸡（Chicken）和鸭（Duck），那么应该考虑抽象出动物（Animal）来应对以后可能出现马和牛的事情。而向接口中添加新成员则会强制要求修改所有派生类，并重新编译，所以版本式的问题最好以抽象类来实现。

— 因为值类型是密封的，所以只能实现接口，而不能继承类。

关于实现继承与接口继承的更详细的讨论与规则，请参见 7.4 节“面向抽象编程：接口和抽象类”。

2. 聚合还是继承，这是个问题。

类与类的关系，通常有以下几种情况，我们分别以两个简单类 Class1 和 Class2 的 UML 图来表示如下。

（1）继承

如图 1-4 所示，Class2 继承自 Class1，任何对基类 Class1 的更改都有可能影响到子类 Class2，继承关系的耦合度较高。

（2）聚合

如图 1-5 所示。

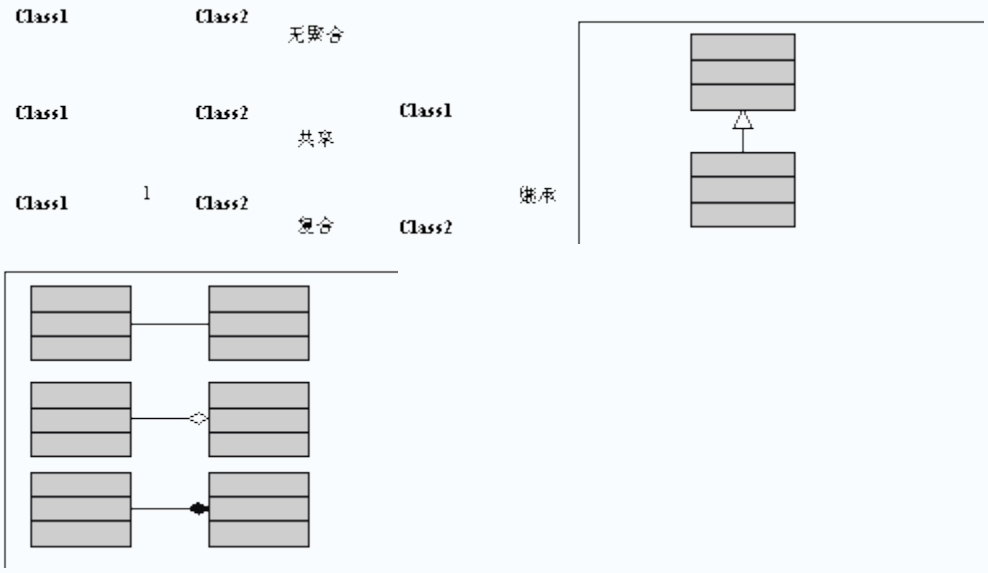


图 1-4 继承关系　　图 1-5 聚合关系

聚合分为三种类型，依次为无、共享和复合，其耦合度逐级递增。无聚合类型关系，类的双方彼此不受影响；共享型关系，Class2 不需要对 Class1 负责；而复合型关系，Class1 会受控于 Class2 的更改，因此耦合度更高。总之，聚合关系是一种 HAS-A 式的关系，耦合度没有继承关系高。

（3）依赖

依赖关系表明，如果 Class2 被修改，则 Class1 会受到影响，如图 1-6 所示。

图 1-6 依赖关系

通过上述三类关系的比较，我们知道类与类之间的关系，通常以耦合度来描述，也就是表示类与类之间的依赖关系程度。没有耦合关系的系统是根本不存在的，因为类与类、模块与模块、系统与系统之间或多或少要发生相互交互，设计应力求将类与类之间的耦合关系降到最低。而面向对象的基本原则之一就是实现低耦合、高内聚的耦合关系，在 2.1 节“OO 原则综述”中所述的合成/聚合复用原则正是对这一思想的直接体现。

显然，将耦合的概念应用到继承机制上，通常情况下子类都会对父类产生紧密的耦合，对基类的修改往往会对子类产生一系列的不良反应。继承之毒瘤主要体现在：

— 继承可能造成子类的无限膨胀，不利于类体系的维护和安全。

— 继承的子类对象确定于编译期，无法满足需要运行期才确定的情况，而类聚合很好地解决了这一问题。

— 随着继承层次的复杂化和子类的多样化，不可避免地会出现对父类的无效继承或者有害继承。子类部分的继承父类的方法或者属性，更能适应实际的设计需求。

那么，通过上面的分析，我们深知继承机制在满足更加柔性的需求方面有一些弊端，从而可能造成系统设计的漏洞与失衡。解决问题的办法当然是多种多样的，根据不同的需求进行不同的设计变更，例如将对象与行为分离抽象出接口实现来避免大基类设计，以聚合代替继承实现更柔性的子类需求等等。

面向对象的基本原则

多聚合，少继承。

低耦合，高内聚。

聚合与继承通常体现在设计模式的伟大思想中，在此以 Adapter 模式的两种方式为例来比较继承和聚合的适应场合与柔性较量。首先对 Adapter 模式进行简单的介绍。Adapter 模式主要用于

将一个类的接口转换为另外一个接口，通常情况下在改变原有体系的条件下应对新的需求变化，通过引入新的适配器类来完成对既存体系的扩展和改造。Adapter 模式就其实现方式主要包括：

— 类的 Adapter 模式。通过引入新的类型来继承原有类型，同时实现新加入的接口方法。其缺点是耦合度高，需要引入过多的新类型。

— 对象的 Adapter 模式。通过聚合而非继承的方式来实现对原有系统的扩展，松散耦合，较少的新类型。

下面，我们回到动物体系中，为鸟儿加上鸣叫 ToTweet 这一行为，为自然界点缀更多美丽的声音。当然不同的鸟叫声是不同的，鸡鸣鹰嘶，各有各的范儿。因此，在 Bird 类的子类都应该对 ToTweet 有不同的实现。现在我们的要求是在不破坏原有设计的基础上来为 Bird 实现 ITweetable 接口，理所当然，以 Adapter 模式来实现这一需求，通过类的 Adapter 模式和对象的 Adapter 模式两种方式来感受其差别。

首先是类的 Adpater 模式，其设计 UML 图表示为图 1-7。

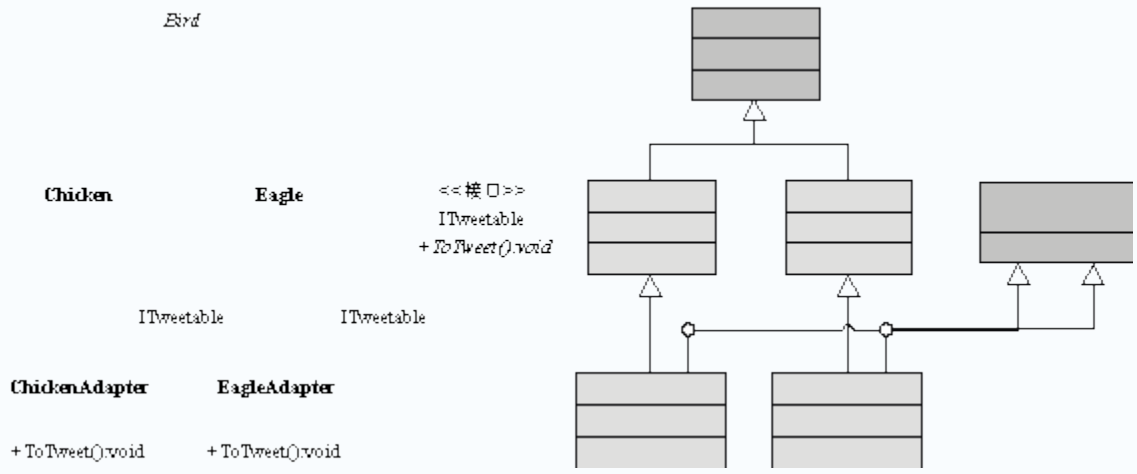


图 1-7 类的 Adapter 模式

在这一新设计体系中，两个新类型 ChickenAdapter 和 EagleAdapter 就是类的 Adapter 模式中新添加的类，它们分别继承自原有的类，从而保留原有类型特性与行为，并实现添加 ITweetable 接口的新行为 ToTweet()。我们没有破坏原有的 Bird 体系，同时添加了新的行为，这是继承的魔力在 Adapter 模式中的应用。我们在客户端应用新的类型来为 Chicken 调用新的方法，如图 1-8 所见，原有继承体系中的方法和新的方法对对象 ca 都是可见的。

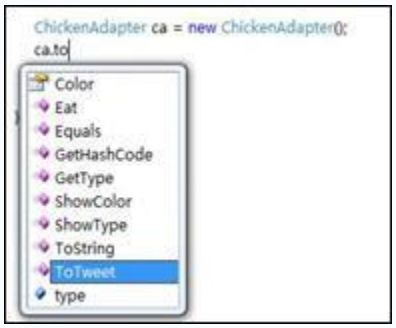


图 1-8 ToTweet 方法的智能感知

我们轻松地完成了这一难题，是否该轻松一下？不。事实上还早着呢，要知道自然界里的鸟儿们都有美丽的歌喉，我们只为 Chicken 和 Eagle 配上了鸣叫的行为，那其他成千上万的鸟儿们都有意见了。怎么办呢？以目前的实现方式我们不得不为每个继承自 Bird 类的子类提供相应的适配类，这样太累了，有没有更好的方式呢？

答案是当然有，这就是对象的 Adapter 模式。类的 Adapter 模式以继承方式来实现，而对象的 Adapter 模式则以聚合的方式来完成，详情如图 1-9 所示。

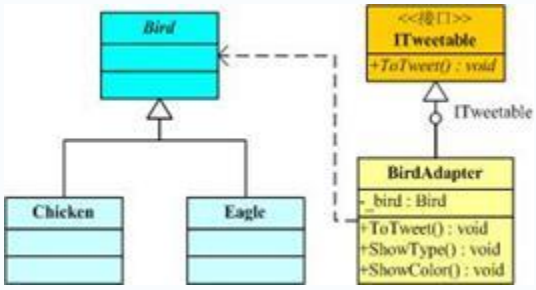


图 1-9 对象的 Adapter 模式

具体的实现细节为：

```
interface ITweetable
{
   void ToTweet();
}
public class BirdAdapter : ITweetable
{
   private Bird _bird;
   public BirdAdapter(Bird bird)
   {
      _bird = bird;
   }
```

```
    public void ShowType()
    {
        _bird.ShowType();
    }
    ……部分省略……
    public void ToTweet()
    {
        //为不同的子类实现不同的 ToTweet 行为
    }
}
```

客户端调用为：

```
public class TestInheritance
{
    public static void Main()
    {
        BirdAdapter ba = new BirdAdapter(new Chicken());
        ba.ShowType();
        ba.ToTweet();
    }
}
```

现在可以松口气了，我们以聚合的方式按照对象的 Adapter 模式思路来解决为 Bird 类及其子类加入 ToTweet()行为的操作，在没有添加过多新类型的基础上十分轻松地解决了这一问题。看起来一切都很完美，新的 BirdAdapter 类与 Bird 类型之间只有松散的耦合关系而不是紧耦合。

至此，我们以一个几乎完整的动物体系类设计，基本完成了对继承与组合问题的探讨，系统设计是一个复杂、兼顾、重构的过程，不管是继承还是聚合，都是系统设计过程中必不可少的技术基础，采取什么样的方式来实现完全取决于具体的需求情况。根据面向对象多组合、少继承的原则，对象的 Adapter 模式更能体现松散的耦合关系，应用更灵活。

1.2.5 规则制胜

根据本节的所有讨论，行文至此，我们很有必要对继承进行归纳总结，将继承概念中的重点内容和重点规则做系统地梳理，对我们来说这些规则条款是掌握继承的金科玉律，主要包括：

— 密封类不可以被继承。

— 继承关系中，我们更多的是关注其共性而不是特性，因为共性是层次复用的基础，而特性是系统扩展的基点。

— 实现单继承，接口多继承。

— 从宏观来看，继承多关注于共通性；而多态多着眼于差异性。

— 继承的层次应该有所控制，否则类型之间的关系维护会消耗更多的精力。

— 面向对象原则：多组合，少继承；低耦合，高内聚。

1.2.6 结论

在.NET 中，如果创建一个类，则该类总是在继承。这缘于.NET 的面向对象特性，所有的类型都最终继承自共同的根 System.Object 类。可见，继承是.NET 运行机制的基础技术之一，一切皆为对象，一切皆于继承。对于什么是继承这个话题，希望每个人能从中寻求自己的答案，理解继承、关注封装、品味多态、玩转接口是理解面向对象的起点，也希望本节是这一旅程的起点。

第十五回：继承本质论

本文将介绍以下内容：

- 什么是继承？
- 继承的实现本质

1. 引言

关于继承，你是否驾熟就轻，关于继承，你是否了如指掌。

本文不讨论继承的基本概念，我们回归本质，从编译器运行的角度来揭示.NET 继承中的运行本源，来发现子类对象是如何实现了对父类成员与方法的继承，以最为简陋的示例来揭示继承的实质，阐述继承机制是如何被执行的，这对于更好的理解继承，是必要且必然的。

2. 分析

下面首先以一个简单的动物继承体系为例，来进行说明：

```
public abstract class Animal

{
```

```
    public abstract void ShowType();
    public void Eat()
    {
        Console.WriteLine("Animal always eat.");
    }
}
public class Bird: Animal
{
    private string type = "Bird";
    public override void ShowType()
    {
        Console.WriteLine("Type is {0}", type);
    }
    private string color;
    public string Color
    {
        get { return color; }
        set { color = value; }
    }
}
public class Chicken : Bird
{
```

```
    private string type = "Chicken";
    public override void ShowType()
    {
        Console.WriteLine("Type is {0}", type);
    }
    public void ShowColor()
    {
        Console.WriteLine("Color is {0}", Color);
    }
}
```

然后，在测试类中创建各个类对象，由于 Animal 为抽象类，我们只创建 Bird 对象和 Chicken 对象。

```
public class TestInheritance
{
    public static void Main()
    {
        Bird bird = new Bird();
        Chicken chicken = new Chicken();
    }
}
```

下面我们从编译角度对这一简单的继承示例进行深入分析，从而了解.NET 内部是如何实现我们强调的继承机制。

（1）我们简要的分析一下对象的创建过程：

```
        Bird animal = new Bird();
```

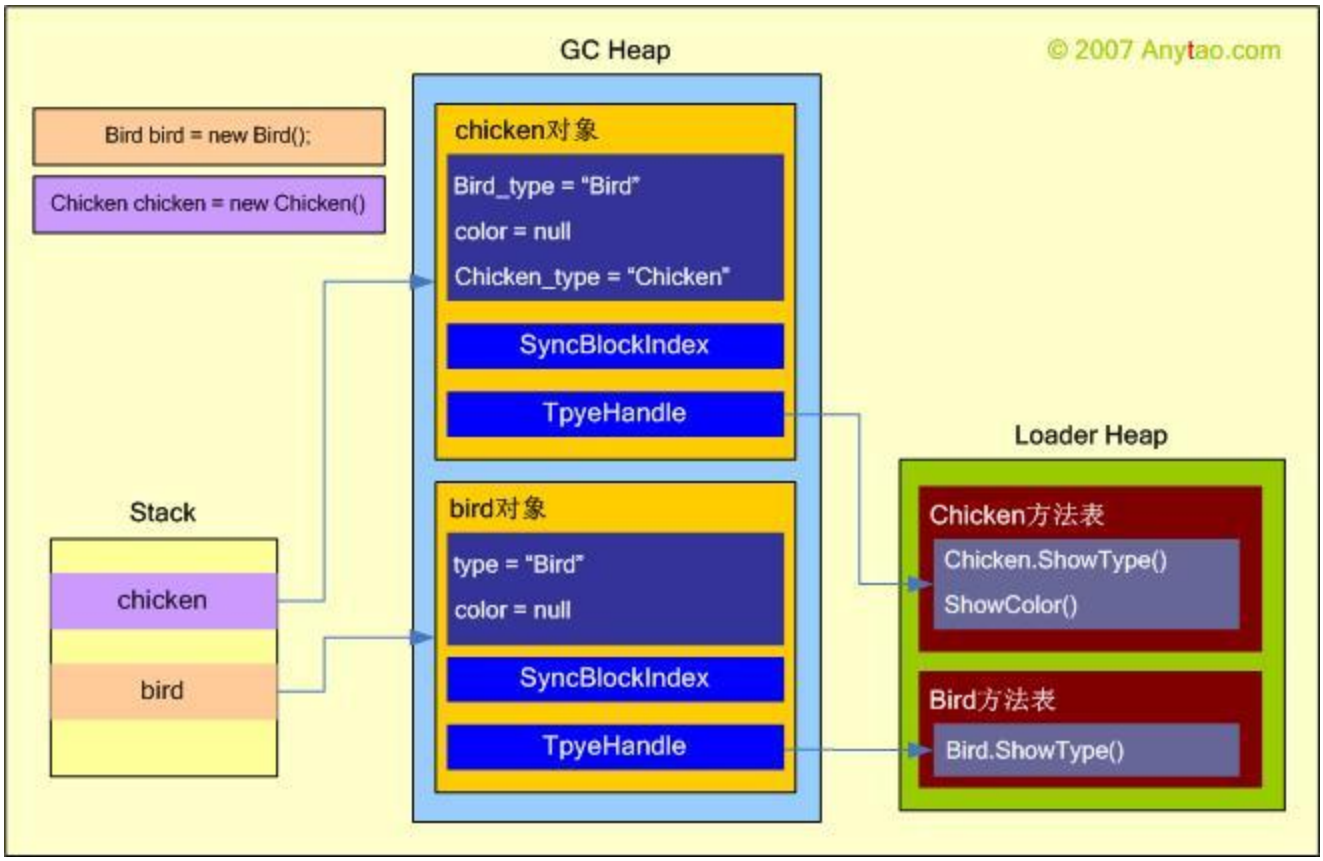
Bird bird 创建的是一个 Bird 类型的引用，而 new Bird()完成的是创建 Bird 对象，分配内存空间和初始化操作，然后将这个对象赋给 bird 引用，也就是建立 bird 引用与 Bird 对象的关联。

（2）我们从继承的角度来分析在编译器编译期是如何执行对象的创建过程，因为继承的本质就体现于对象的创建过程。

在此我们以 Chicken 对象的创建为例，首先是字段，对象一经创建，会首先找到其父类 Bird，并为其字段分配存储空间，而 Bird 也会继续找到其父类 Animal，为其分配存储空间，依次类推直到递归结束，也就是完成 System.Object 内存分配为止。我们可以在编译器中单步执行的方法来大致了解其分配的过程和顺序，因此，对象的创建过程是按照顺序完成了对整个父类及其本身字段的内存创建，并且字段的存储顺序是由上到下排列，object 类的字段排在最前面，其原因是如果父类和子类出现了同名字段，则在子类对象创建时，编译器会自动认为这是两个不同的字段而加以区别。

然后，是方法表的创建，必须明确的一点是方法表的创建是类第一次加载到 CLR 时完成的，在对象创建时只是将其附加成员 TypeHandle 指向方法列表在 Loader Heap 上的地址，将对象与其动态方法列表相关联起来，因此方法表是先于对象而存在的。类似于字段的创建过程，方法表的创建也是父类在先子类在后，原因是显而易见的，类 Chicken 生成方法列表时，首先将 Bird 的所有方法拷贝一份，然后和 Chicken 本身的方法列表做以对比，如果有覆写的虚方法则以子类方法覆盖同名的父类方法，同时添加子类的新方法，从而创建完成 Chicken 的方法列表。这种创建过程也是逐层递归到 Object 类，并且方法列表中也是按照顺序排列的，父类在前子类在后，其原因和字段大同小异，留待读者自己体味。

结合我们的分析过程，现在将对象创建的过程以简单的图例来揭示其在内存中的分配情形，如下：



从我们的分析，和上面的对象创建过程可见，对继承的本质我们有了更明确的认识，对于以下的问题就有了清晰明白的答案：

- 继承是可传递的，子类是对父类的扩展，必须继承父类方法，同时可以添加新方法。
- 子类可以调用父类方法和字段，而父类不能调用子类方法和字段。
- 虚方法如何实现覆写操作，使得父类指针可以指向子类对象成员。
- new 关键字在虚方法继承中的阻断作用。

你是否已经找到了理解继承、理解动态编译的不二法门。

3. 思考

通过上面的讲述与分析，我们基本上对.NET 在编译期的实现原理有了大致的了解，但是还有以下的问题，一定会引起一定的疑惑，那就是：

```
            Bird bird2 = new Chicken();
```

这种情况下，bird2.ShowType 应该返回什么值呢？而 bird2.type 有该是什么值呢？有两个原则，是.NET 专门用于解决这一问题的：

- 关注对象原则：调用子类还是父类的方法，取决于创建的对象是子类对象还是父类对象，而不是它的引用类型。例如 Bird bird2 = new Chicken()时，我们关注的是其创建对象为 Chicken 类型，因此子类将继承父类的字段和方法，或者覆写父类的虚方法，而不用关注 bird2 的引用类型是否为 Bird。引用类型不同的区别决定了不同的对象在方法表中不同的访问权限。

注意

根据关注对象原则，那么下面的两种情况又该如何区别呢？

```
        Bird bird2 = new Chicken();

        Chicken chicken = new Chicken();
```

根据我们上文的分析，bird2 对象和 chicken 对象在内存布局上是一样的，差别就在于其引用指针的类型不同：bird2 为 Bird 类型指针，而 chicken 为 Chicken 类型指针。以方法调用为例，不同的类型指针在虚拟方法表中有不同的附加信息作为标志来区别其访问的地址区域，称为 offset。不同类型的指针只能在其特定地址区域内进行执行，子类覆盖父类时会保证其访问地址区域的一致性，从而解决了不同的类型访问具有不同的访问权限问题。

- 执行就近原则：对于同名字段或者方法，编译器是按照其顺序查找来引用的，也就是首先访问离它创建最近的字段或者方法，例如上例中的 bird2，是 Bird 类型，因此会首先访问 Bird_type（注意编译器是不会重新命名的，在此是为区分起见），如果 type 类型设为 public，则在此将返回“Bird”值。这也就是为什么在对象创建时必须将字段按顺序排列，而父类要先于子类编译的原因了。

思考

1. 上面我们分析到 bird2.type 的值是“Bird”，那么 bird2.ShowType()会显示什么值呢？答案是“Type is Chicken”，根据本文上面的分析，想想到底为什么？

2. 关于 new 关键字在虚方法动态调用中的阻断作用，也有了更明确的理论基础。在子类方法中，如果标记 new 关键字，则意味着隐藏基类实现，其实就是创建了与父类同名的另一个方法，在编译中这两个方法处于动态方法表的不同地址位置，父类方法排在前面，子类方法排在后面。

4. 结论

在.NET 中，如果创建一个类，则该类总是在继承。这缘于.NET 的面向对象特性，所有的类型都最终继承自共同的根 System.Object 类。可见，继承是.NET 运行机制的基础技术之一，一切皆为对象，一切皆于继承。本文从基础出发，深入本质探索本源，分析疑难比较鉴别。对于什么是继承这个话题，希望每个人能从中寻求自己的答案，理解继承、关注封装、玩转多态是理解面向对象的起点，希望本文是这一旅程的起点。

[祝福]
仅以此篇献给我的老师们：汤文海老师，陈桦老师。

1.3 封装的秘密

本节将介绍以下内容：

— 面向对象的封装特性

— 字段赏析

— 属性赏析

1.3.1 引言

在面向对象三要素中，封装特性为程序设计提供了系统与系统、模块与模块、类与类之间交互的实现手段。封装为软件设计与开发带来前所未有的革命，成为构成面向对象技术最为重要的基础之一。在.NET 中，一切看起来都已经被包装在.NET Framework 这一复杂的网络中，提供给最终开发人员的是成千上万的类型、方法和接口，而 Framework 内部一切已经做好了封装。例如，如果你想对文件进行必要的操作，那么使用 System.IO.File 基本就能够满足多变的需求，因为.NET Framwork 已经把对文件的重要操作都封装在 System.IO.File 等一些基本类中，用户不需要关心具体的实现。

1.3.2 让 ATM 告诉你，什么是封装

那么，封装究竟是什么？

首先，我们考察一个常见的生活实例来进行说明，例如每当发工资的日子小王都来到 ATM 机前，用工资卡取走一笔钱为女朋友买礼物，从这个很帅的动作，可以得出以下的结论：

— 小王和 ATM 机之间，以银行卡进行交互。要取钱，请交卡。

— 小王并不知道 ATM 机将钱放在什么地方，取款机如何计算钱款，又如何通过银行卡返回小王所要数目的钱。对小王来说，ATM 就是一个黑匣子，只能等着取钱；而对银行来说，ATM 机就像银行自己的一份子，是安全、可靠、健壮的员工。

— 小王要想取到自己的钱，必须遵守 ATM 机的对外约定。他的任何违反约定的行为都被视为不轨，例如欲以砖头砸开取钱，用公交卡冒名取钱，盗卡取钱都将面临法律风险，所以小王只能安分守己地过着月光族的日子。

那么小王和 ATM 机的故事，能给我们什么样的启示？对应上面的 3 条结论，我们的分析如下：

— 小王以工资卡和 ATM 机交互信息，ATM 机的入卡口就是 ATM 机提供的对外接口，砖头是塞不进去的，公交卡放进去也没有用。

— ATM 机在内部完成身份验证、余额查询、计算取款等各项服务，具体的操作对用户小王是不可见的，对银行来说这种封闭的操作带来了安全性和可靠性保障。

— 小王和 ATM 机之间遵守了银行规定、国家法律这样的协约。这些协约和法律，就挂在 ATM 机旁边的墙上。

结合前面的示例，再来分析封装吧。具体来说，封装隐藏了类内部的具体实现细节，对外则提供统一访问接口，来操作内部数据成员。这样实现的好处是实现了 UI 分离，程序员不需要知道类内部的具体实现，只需按照接口协议进行控制即可。同时对类内部来说，封装保证了类内部成员的安全性和可靠性。在上例中，ATM 机可以看做封装了各种取款操作的类，取款、验证的操作对类 ATM 来说，都在内部完成。而 ATM 类还提供了与小王交互的统一接口，并以文档形式——法律法规，规定了接口的规范与协定来保证服务的正常运行。以面向对象的语言来表达，类似于下面的样子：

```
namespace InsideDotNet.OOThink.Encapsulation
{
    /// <summary>
    /// ATM 类
    /// </summary>
    public class ATM
    {
        #region 定义私有方法，隐藏具体实现

        private Client GetUser(string  userID) {}
        private bool IsValidUser(Client user) {}
        private int GetCash(int money) {}
        #endregion
        #region 定义公有方法，提供对外接口
        public void CashProcess(string userID, int money)
        {
            Client tmpUser = GetUser(userID);
            if (IsValidUser(tmpUser))
            {
                GetCash(money);
            }
            else
            {
                Console.Write("你不是合法用户，是不是想被发配南极？");
            }
        }
        #endregion
    }
    /// <summary>
    /// 用户类
    /// </summary>
```

```
    public class Client
    {
    }
}
```

在.NET 应用中，Framework 封装了你能想到的各种常见的操作，就像微软提供给我们一个又一个功能不同的 ATM 机一样，而程序员手中筹码就是根据.NET 规范进行开发，是否能取出自己的钱，要看你的卡是否合法。

那么，如果你是银行的主管，又该如何设计自己的 ATM 呢？该以什么样的技术来保证自己的 ATM 在内部隐藏实现，对外提供接口呢？

1.3.3 秘密何处：字段、属性和方法

字段、属性和方法，是面向对象的基本概念之一，其基本的概念介绍不是本书的范畴，任何一本关于语言和面向对象的著作中都有相关的详细解释。本书关注的是在类设计之初应该基于什么样的思路，来实现类的功能要求与交互要求？每个设计者，是以什么角度来完成对类架构的设计与规划呢？在我看来，下面的问题是应该首先被列入讨论的选项：

— 类的功能是什么？

— 哪些是字段，哪些是属性，哪些是方法？

— 对外提供的公有方法有哪些，对内隐藏的私有变量有哪些？

— 类与类之间的关系是继承还是聚合？

这些看似简单的问题，却往往是困扰我们进行有效设计的关键因素，通常系统需求描述的核心名词，可以抽象为类，而对这些名词驱动的动作，可以对应地抽象为方法。当然，具体的设计思路要根据具体的需求情况，在整体架构目标的基础上进行有效的筛选、剥离和抽象。取舍之间，彰显 OO 智慧与设计模式的魅力。

那么，了解这些选项与原则，我们就不难理解关于字段、属性和方法的实现思路了，这些规则可以从对字段、属性和方法的探索中找到痕迹，然后从反方向来完善我们对于如何设计的思考与理解。

1. 字段

字段（field）通常定义为 private，表示类的状态信息。CLR 支持只读和读写字段。值得注意的是，大部分情况下字段都是可读可写的，只读字段只能在构造函数中被赋值，其他方法不能改变只读字段。常见的字段定义为：

```
public class Client
{
   private string name;          //用户姓名
   private int age;              //用户年龄
   private string password;      //用户密码
}
```

如果以 public 表示类的状态信息，则我们就可以以类实例访问和改变这些字段内容，例如：

```
public static void Main()
{
   Client xiaoWang = new Client();
   xiaoWang.name = "Xiao Wang";
   xiaoWang.age = 27;
   xiaoWang.password = "123456"
}
```

这样看起来并没有带来什么问题，Client 实例通过操作公有字段很容易达到存取状态信息的目的，然而封装原则告诉我们：类的字段信息最好以私有方式提供给类的外部，而不是以公有方式来实现，否则不适当的操作将造成不必要的错误方式，破坏对象的状态信息，数据安全性和可靠性无法保证。例如：

```
xiaoWang.age = 1000;
xiaoWang.password = "5&@@Ld;afk99";
```

显然，小王的年龄不可能是 1000 岁，他是人不是怪物；小王的密码也不可能是“@&;”这些特殊符号，因为 ATM 机上根本没有这样的按键，而且密码必须是 6 位。所以对字段公有化的操作，会引起对数据安全性与可靠性的破坏，封装的第一个原则就是：将字段定义为 private。

那么，如上文所言，将字段设置为 private 后，对对象状态信息的控制又该如何实现呢？小王的状态信息必须以另外的方式提供给类外部访问或者改变。同时我们也期望除了实现对数据的访问，最好能加入一定的操作，达到数据控制的目的。因此，面向对象引入了另一个重量级的概念：属性。

2. 属性

属性（property）通常定义为 public，表示类的对外成员。属性具有可读、可写属性，通过 get 和 set 访问器来实现其读写控制。例如上文中 Client 类的字段，我们可以相应地封装其为属性：

```
public class Client
{
    private string name;        //用户姓名
    public string Name
    {
        get { return name; }
        set
        {
            name = value == null ? String.Empty : value;
        }
    }
    private int age;            //用户年龄
    public int Age
    {
        get { return age; }
        set
        {
            if ((value > 0) && (value < 150))
            {
```

```
            age = value;
        }
        else
        {
            throw new ArgumentOutOfRangeException ("年龄信息不正确。");
        }
    }
  }
}
```

当我们再次以

```
xiaoWang.Age = 1000;
```

这样的方式来实现对小王的年龄进行写控制时，自然会弹出异常提示，从而达到了保护数据完整性的目的。

那么，属性的 get 和 set 访问器怎么实现对对象属性的读写控制呢？我们打开 ILDASM 工具查看 client 类反编译后的情况时，会发现如图 1-10 所示的情形。

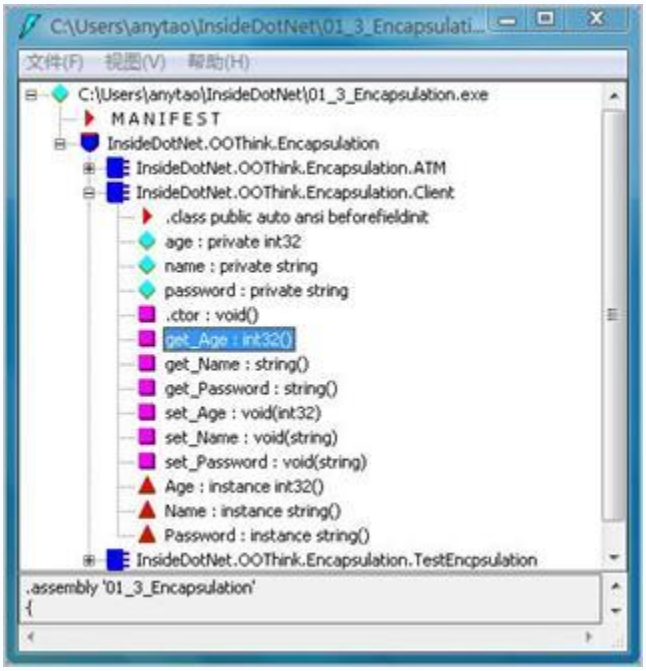


图 1-10 Client 类的 IL 结构

由图 1-10 可见，IL 中不存在 get 和 set 方法，而是分别出现了 get_Age、set_Age 这样的方法，打开其中的任意方法分析会发现，编译器的执行逻辑是：如果发现一个属性，并且查看该属性中实现了 get 还是 set，就对应地生成 get_属性名、set_属性名两个方法。因此，我们可以说，属性的实质其实就是在编译时分别将 get 和 set 访问器实现为对外方法，从而达到控制属性的目的，而对属性的读写行为伴随的实际是一个相应方法的调用，它以一种简单的形式实现了方法。

所以我们也可以定义自己的 get 和 set 访问器，例如：

```
public string get_Password()
{
   return password;
}
public string set_Password(string value)
{
   if (value.Length < 6)
      password = value;
}
```

事实上，这种实现方法正是 Java 语言所采用的机制，而这样的方式显然没有实现 get 和 set 访问器来得轻便，而且对属性的操作也带来多余的麻烦，所以我们推荐的还是下面的方式：

```
public string Password
{
   get { return password; }
   set
   {
      if (value.Length < 6)
         password = value;
   }
}
```

另外，get 和 set 对属性的读写控制，是通过实现 get 和 set 的组合来实现的，如果属性为只读，则只实现 get 访问器即可；如果属性为可写，则实现 set 访问器即可。

通过对公共属性的访问来实现对类状态信息的读写控制，主要有两点好处：一是避免了对数据安全的访问限制，包含内部数据的可靠性；二是避免了类扩展或者修改带来的变量连锁反应。

至于修改变量带来的连锁反应，表现在对类的状态信息的需求信息发生变化时，如何来减少代码重构基础上，实现最小的损失和最大的补救。例如，如果对 client 的用户姓名由原来的简单 name 来标识，换成以 firstName 和 secondName 来实现，如果不是属性封装了字段而带来的隐藏内部细节的特点，那么我们在代码中就要拼命地替换原来 xiaoWang.name 这样的实现了。例如：

```
private string firstName;
private string secondName;
public string Name
{
   get { return firstName + secondName; }
}
```

这样带来的好处是，我们只需要更改属性定义中的实现细节，而原来程序 xiaoWang.name 这样的实现就不需要做任何修改即可适应新的需求。你看，这就是封装的强大力量使然。

还有一种含参属性，在 C＃中称为索引器（indexer），对 CLR 来说并没有含不含参数的区别，它只是负责将相应的访问器实现为对应的方法，不同的是含参属性中加入了对参数的处理过程罢了。

3. 方法

方法（method）封装了类的行为，提供了类的对外表现。用于将封装的内部细节以公有方法提供对外接口，从而实现与外部的交互与响应。例如，从上面属性的分析我们可知，实际上对属性的读写就是通过方法来实现的。因此，对外交互的方法，通常实现为 public。

当然不是所有的方法都被实现为 public，否则类内部的实现岂不是全部暴露在外。必须对对外的行为与内部操作行为加以区分。因此，通常将在内部的操作全部以 private 方式来实现，而将需要与外部交互的方法实现为 public，这样既保证了对内部数据的隐藏与保护，又实现了类的对外交互。例如在 ATM 类中，对钱的计算、用户验证这些方法涉及银行的关键数据与安全数据的保护问题，必须以 private 方法来实现，以隐藏对用户不透明的操作，而只提供返回钱款这一 public 方法接口即可。在封装原则中，有效地保护内部数据和有效地暴露外部行为一样关键。

那么这个过程应该如何来实施呢？还是回到 ATM 类的实例中，我们首先关注两个方法：IsValidUser()和 CashProcess()，其中 IsValidUser()用于验证用户的合法性，而 CashProcess()用于提供用户操作接口。显然，验证用户是银行本身的事情，外部用户无权访问，它主要用于在内部进行验证处理操作，例如 CashProcess()中就以 IsValidUser()作为方法的进入条件，因此很容易知道 IsValidUser()被实现为 private。而 CashProcess()用于和外部客户进行交互操作，这正是我们反复强调的外部接口方法，显然应该实现为 public。其他的方法 GetUser()、GetCash()也是从这一主线出发来确定其对外封装权限的，自然就能找到合理的定位。从这个过程中我们发现，谁为公有、谁为私有，取决于需求和设计双重因素，在职责单一原则下为类型设计方法，应该广泛考虑的是类本身的功能性，从开发者与设计者两个角度出发，分清访问权限就会水到渠成。

1.3.4 封装的意义

通过对字段、属性与方法在封装性这一点上的分析，我们可以更加明确地了解到封装特性作为面向对象的三大特性之一，表现出来的无与伦比的重要性与必要性，对于深入地理解系统设计与类设计提供了绝好的切入点。

下面，我们针对上文的分析进行小结，以便更好地理解我们对于封装所提出的思考，主要包括：

（1）字段通常定义为 private，属性通常实现为 public，而方法在内部实现为 private，对外部实现为 public，从而保证对内部数据的可靠性读写控制，保护了数据的安全和可靠，同时又提供了与外部接口的有效交互。这是类得以有效封装的基础机制。

（2）通常情况下的理解正如我们上面提到的规则，但是具体的操作还要根据实际的设计需求而定，例如有些时候将属性实现为 private，也将方法实现为 private 是更好的选择。例如在 ATM 类中，可能需要提供计数器来记录更新或者选择的次数，而该次数对用户而言是不必要的状态信息，因此只需在 ATM 类内部实现为 private 即可；同理，类型中的某些方法是对内部数据的操作，因此也以 private 方式来提供，从而达到数据安全的目的。

（3）从内存和数据持久性角度上来看，有一个很重要但常常被忽视的事实是，封装属性提供了数据持久化的有效手段。因为，对象的属性和对象一样在内存期间是常驻的，只要对象不被垃圾回收，其属性值也将一直存在，并且记录最近一次对其更改的数据。

（4）在面向对象中，封装的意义还远不止类设计层面对字段、属性和方法的控制，更重要的是其广义层面。我们理解的封装，应该是以实现 UI 分离为目的的软件设计方法，一个系统或者软件开发之后，从维护和升级的目的考虑，一定要保证对外接口部分的绝对稳定。不管系统内部的功能性实现如何多变，保证接口稳定是保证软件兼容、稳定、健壮的根本。所以 OO 智慧中的封装性旨在保证：

— 隐藏系统实现的细节，保证系统的安全性和可靠性。

— 提供稳定不变的对外接口。因此，系统中相对稳定部分常被抽象为接口。

— 封装保证了代码模块化，提高了软件的复用和功能分离。

1.3.5 封装规则

现在，我们对封装特性的规则做一个总结，这些规则就是在平常的实践中提炼与完善出的良药，我们在进行实际的开发和设计工作时，应尽量遵守规则，而不是盲目地寻求方法。

— 尽可能地调用类的访问器，而不是成员，即使在类的内部。其目的在我们的示例中已有说明，例如 Client 类中的 Name 属性就可以避免由于需求变化带来的代码更改问题。

— 内部私有部分可以任意更改，但是一定要在保证外部接口稳定的前提下。

— 将对字段的读写控制实现为属性，而不是方法，否则舍近而求远，非明智之选。

— 类封装是由访问权限来保证的，对内实现为 private，对外实现为 public。再结合继承特性，还要对 protected，internal 有较深的理解，详细的情况参见 1.1 节“对象的旅行”。

— 封装的精华是封装变化。张逸在《软件设计精要与模式》一书中指出，封装变化是面向对象思想的核心，他提到开发者应从设计角度和使用角度两方面来分析封装。因此，我们将系统中变化频繁的部分封装为独立的部分，这种隔离选择有利于充分的软件复用和系统柔性。

1.3.6 结论

封装是什么？横扫全文，我们的结论是：封装就是一个包装，将包装的内外分为两个空间，对内实现数据私有，对外实现方法调用，保证了数据的完整性和安全性。

我们从封装的意义谈起，然后逐层深入到对字段、属性和方法在定义和实现上的规则，这是一次自上而下的探求方式，也是一次反其道而行的揭密旅程。关于封装，远不是本节所能全面展现的话题，关于封装的技巧和更多深入的探求，来自于面向对象，来自于设计模式，也来自于软件工程。因此，要想全面而准确地认识封装，除了本节打下的基础之外，不断的在实际学习中完善和总结是不可缺少的，这在.NET 学习中也是至关重要的。

1.4 多态的艺术

本节将介绍以下内容：

— 什么是多态？

— 动态绑定

— 品味多态和面向对象

1.4.1 引言

翻开大部头的《韦氏大词典》，关于多态（Polymorphisn）的定义为：可以呈现不同形式的能力或状态。这一术语来源于生物系统，意指同族生物具有的相同特征。而在.NET 中，多态指同一操作作用于不同的实例，产生不同运行结果的机制。继承、封装和多态构成面向对象三要素，成就了面向对象编程模式的基础技术机制。

在本节，我们以入情入理的小故事为线索，来展开一次关于多态的循序渐进之旅，在故事的情节中思考多态和面向对象的艺术品质。

1.4.2 问题的抛出

故事开始。

小王的爷爷，开始着迷于电脑这个新鲜玩意儿了，但是老人家面对陌生的屏幕却总是摸不着头脑，各种各样的文件和资料眼花缭乱，老人家却不知道如何打开，这可急坏了身为光荣程序员的小王。为了让爷爷享受高科技带来的便捷与震撼，小王决定自己开发一个万能程序，用来一键式打开常见的计算机资料，例如文档、图片和影音文件等，只需安装一个程序就可以免了其他应用文件的管理，并且使用方便，就暂且称之为万能加载器（FileLoader）吧。

既然是个独立的应用系统，小王就分析了万能加载器应有的几个功能点，小结如下：

— 自动加载各种资料，一站式搜索系统常见资料。

— 能够打开常见文档类资料，例如 txt 文件、Word 文件、PDF 文件、Visio 文件等。

— 能够打开常见图片资料，例如 jpg 格式文件、gif 格式文件、png 格式文件等。

— 能够打开常见音频资料和视频资料，例如 avi 文件、mp3 文件等。

— 支持简单可用的类型扩展接口，易于实现更多文件类型的加载。

这可真是一个不小的挑战，小王决定利用业余时间逐步地来实现这一伟大的构想，就当成是送给爷爷 60 岁的寿礼。有了一个令人兴奋的念头，小王怎么都睡不着，半夜按捺不住爬起来，构思了一个基本的系统流程框架，如图 1-11 所示。

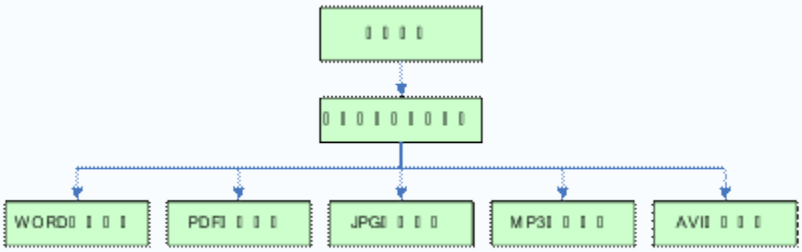


图 1-11 万能加载器系统框架图

1.4.3 最初的实现

说干就干，小王按照构思的系统框架，首先构思了可能打开的最常用的文件，并将其设计为一个枚举，这样就可以统一来管理文件的类型了，实现如下：

```
//可支持文件类型，以文件扩展名划分
enum FileType
{
    doc,   //Word 文档
    pdf,   //PDF 文档
    txt,   //文本文档
    ppt,   //Powerpoint 文档
    jpg,   //jpg 格式图片
    gif,   //gif 格式图片
    mp3,   //mp3 音频文件
    avi    //avi 视频文件
}
```

看着这个初步设想的文件类型枚举，小王暗暗觉得真不少，如果再增加一些常用的文件类型，这个枚举还真是气魄不小呀。

有了要支持的文件类型，小王首先想到的就是实现一个文件类，来代表不同类型的文件资料，具体如下：

```
class Files
```

```
{
    private FileType fileType;
    public FileType FileType
    {
        get { return fileType; }
    }
}
```

接着小王按照既定思路构建了一个打开文件的管理类，为每种文件实现其具体的打开方式，例如：

```
class FileManager
{
    //打开 Word 文档
    public void OpenDocFile()
    {
        Console.WriteLine("Alibaba, Open the Word file.");
    }
    //打开 PDF 文档
    public void OpenPdfFile()
    {
        Console.WriteLine("Alibaba, Open the PDF File.");
    }
    //打开 Jpg 文档
    public void OpenJpgFile()
    {
        Console.WriteLine("Alibaba, Open the Jpg File.");
    }
    //打开 MP3 文档
    public void OpenMp3File()
    {
        Console.WriteLine("Alibaba, Open the MP3 File.");
    }
```

```
}
```

哎呀，这个长长的单子还在继续往下写：OpenJpgFile、OpenGifFile、OpenMp3File、OpenAviFile……不知到什么时候。

上一步着实让小王步履维艰，下一步的实现更让小王濒临崩溃了，在系统调用端，小王实现的文件加载器是被这样实现的：

```
class FileClient
{
   public static void Main()
   {
      //首先启动文件管理器
      FileManager fm = new FileManager();
      //看到一堆一堆的电脑资料
      IList<Files> files = new List<Files>();
      //当前的万能加载器该如何完成工作呢？
      foreach (Files file in files)
      {
         switch(file.FileType)
         {
            case FileType.doc:
               fm.OpenDocFile();
               break;
            case FileType.pdf:
               fm.OpenPdfFile();
               break;
            case FileType.jpg:
               fm.OpenJpgFile();
               break;
            case FileType.mp3:
               fm.OpenMp3File();
               break;
```

```
            //.....部分省略.....
        }
    }
  }
}
```

完成了文件打开的调用端，一切都好像上了轨道，小王的万能文档器也有了基本的架子，剩下再根据实际需求做些调整即可。小王兴冲冲地将自己的作品拿给爷爷试手，却发现爷爷正在想打开一段 rm 格式的京剧听听。但是小王的系统还没有支持这一文件格式，没办法只好回去继续修改了。

等到要添加支持新类型的时候，拿着半成品的小王，突然发现自己的系统好像很难再插进一脚，除了添加新的文件支持类型，修改打开文件操作代码，还得在管理类中添加新的支持代码，最后在客户端还要修改相应的操作。小王发现添加新的文件类型，好像把原来的系统整个做了一次大装修，那么下次爷爷那里有了新需求呢，号称万能加载器的作品，应该怎么应付下一次的需求变化呢？这真是噩梦，气喘吁吁的小王，忍不住回头看了看一天的作品，才发现自己好像掉进了深渊，无法回头。勇于探索的小王经过一番深入的分析发现了当前设计的几个重要问题，主要包括：

— 需要深度调整客户端，为系统维护带来麻烦，况且我们应该尽量保持客户端的相对稳定。

— Word、PDF、MP3 等，都是可以实现的独立对象，整个系统除了有文档管理类，几乎没有面向对象的影子，全部是面向结构和过程的开发方式。

— 在实现打开文件程序时，小王发现其实 OpenDocFile 方法、OpenPDFFile 方法以及 OpenTxtFile 方法有很多可复用的代码，而 OpenJpgFile 方法和 OpenGifFile 方法也有很多重复构造的地方。

— 由于系统之间没有分割、没有规划，整个系统就像一堆乱麻，几乎不可能完成任何简单的扩展和维护。

— 任何修改都会将整个系统洗礼一次，修改遍布全系统的整个代码，并且全部重新编译才行。

— 需求变更是结构化设计的大敌，无法轻松完成起码的系统扩展和变更，例如在打开这一操作之外，如果实现删除、重命名等其他操作，对当前的系统来说将是致命的打击。在发生需求多变的今天，必须实现能够灵活扩展和简单变更的设计构思，面向对象是灵活设计的有效手段之一。

1.4.4 多态，救命的稻草

看着经不起考验的系统，经过了短期的郁闷和摸索，小王终于找到了阿里巴巴念动芝麻之门打开的魔咒，这就是：多态。

没错！就是多态，就是面向对象。这是小王痛定思痛后，发出的由衷感慨。小王再接再厉，颠覆了原来的构思，一个新的设计框架应运而生，如图 1-12。

结合新的框架，比较之前的蹩脚设计，小王提出了新系统的新气象，主要包括以下几个修改：

— 将 Word、PDF、TXT、JPG、AVI 等业务实体抽象为对象，并在每个相应的对象内部来处理本对象类型的文件打开工作，这样各个类型之间的交互操作就被分离出来，这样很好地体现了职责单一原则的目标。

— 将各个对象的属性和行为相分离，将文件打开这一行为封装为接口，再由其他类来实现这一接口，有利于系统的扩展同时减少了类与类的依赖。

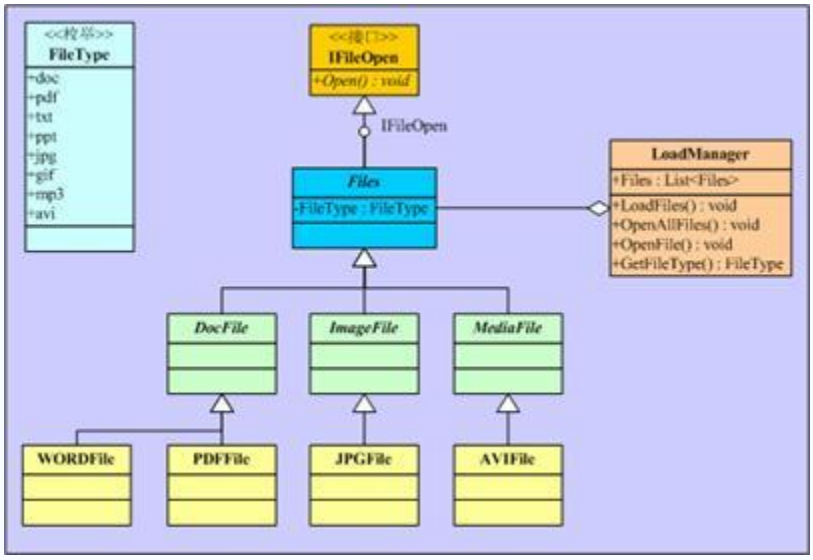


图 1-12 万能加载器系统设计

— 将相似的类抽象出公共基类，在基类中实现具有共性的特征，并由子类继承父类的特征，例如 Word、PDF、TXT 的基类可以抽象为 DocLoader；而 JPG 和 GIF 的基类可以抽象为 ImageLoader，这种实现体现的是面向对象的开放封闭原则：对扩展开放，对修改关闭。如果有新的类型需要扩展，则只需继承合适的基类成员，实现新类型的特征代码即可。

— 实现可柔性扩展的接口机制，能够更加简单的实现增加新的文件类型加载程序，也能够很好的扩展打开文件之外的其他操作，例如删除、重命名等修改操作。

— 实现在不需要调整原系统，或者很少调整原系统的情况下，进行功能扩展和优化，甚至是无需编译的插件式系统。

下面是具体的实现，首先是通用的接口定义：

```
interface IFileOpen
{
   void Open();
}
```

接着定义所有文件类型的公共基类，因为公共的文件基类是不可以实例化的，在此处理为抽象类实现会更好，详细为：

```
abstract class Files: IFileOpen
{
   private FileType fileType = FileType.doc;
   public FileType FileType
   {
      get { return fileType; }
   }
   public abstract void Open();
}
```

基类 Files 实现了 IFileOpen 接口，不过在此仍然定义方法为抽象方法。除了文件打开抽象方法，还可以实现其他的通用文件处理操作，例如文件删除 Delete、文件重命名 ReName 和获取文件路径等。有了文件类型的公共基类，是时候实现其派生类了。经过一定的分析和设计，小王没有马上提供具体的资料类型类，而是对派生类型做了归档，初步实现文件类型、图片类型和媒体类型三个大类，将具体的文件类型进一步做了抽象：

```
   abstract class DocFile: Files
   {
      public int GetPageCount()
      {
         //计算文档页数
```

```
    }
  }
  abstract class ImageFile : Files
  {
    public void ZoomIn()
    {
      //放大比例
    }
    public void ZoomOut()
    {
      //缩小比例
    }
}
```

终于是实现具体资料类的时候了，在此以 Word 类型为例来说明具体的实现：

```
class WORDFile : DocFile
{
  public override void Open()
  {
    Console.WriteLine("Open the WORD file.");
  }
}
```

其他类型的实现类似于此，不同之处在于不同的类型有不同 Open 实现规则，以应对不同资料的打开操作。

小王根据架构的设计，同时提供了一个资料管理类来进行资料的统一管理：

```
class LoadManager
{
  private IList<Files> files = new List<Files>();
  public IList<Files> Files
  {
    get { return files; }
  }
  public void LoadFiles(Files file)
  {
    files.Add(file);
```

```
    }
    //打开所有资料
    public void OpenAllFiles()
    {
        foreach(IFileOpen file in files)
        {
            file.Open();
        }
    }
    //打开单个资料
    public void OpenFile(IFileOpen file)
    {
        file.Open();
    }
    //获取文件类型
    public FileType GetFileType(string fileName)
    {
        //根据指定路径文件返回文件类型
        FileInfo fi = new FileInfo(fileName);
        return (FileType)Enum.Parse(typeof(FileType), fi.Extension);
    }
}
```

最后，小王实现了简单的客户端，并根据所需进行文件的加载：

```
class FileClient
{
    public static void Main()
    {
        //首先启动文件加载器
        LoadManager lm = new LoadManager();
        //添加要处理的文件
        lm.LoadFiles(new WORDFile());
        lm.LoadFiles(new PDFFile());
        lm.LoadFiles(new JPGFile());
        lm.LoadFiles(new AVIFile());
        foreach (Files file in lm.Files)
```

```
    {
        if (file is 爷爷选择的)      //伪代码
        {
            lm.OpenFile(file);
        }
    }
  }
}
```

当然，现在的 FileLoader 客户端还有很多要完善的工作要做，例如关于文件加载的类型，完全可以定义在配置文件中，并通过抽象工厂模式和反射于运行期动态获取，以避免耦合在客户端。不过基本的文件处理部分已经能够满足小王的预期。

1.4.5 随需而变的业务

爷爷机子上的资料又增加了新的视频文件 MPEG，原来的 AVI 文件都太大了。可是这回根本就没有难倒小王的万能加载器。在电脑前轻松地折腾 30 分钟后，万能加载器就可以适应新的需求，图 1-13 所示的是修改的框架设计。

按照这个新的设计，小王对系统只需做如下的简单调整，首先是增加处理 MPEG 文件的类型 MPEGFile，并让它继承自 MediaFile，实现具体的 Open 方法即可。

```
class MPEGFile : MediaFile
{
  public override void Open()
  {
    Console.WriteLine("Open the MPEG file.");
  }
}
```

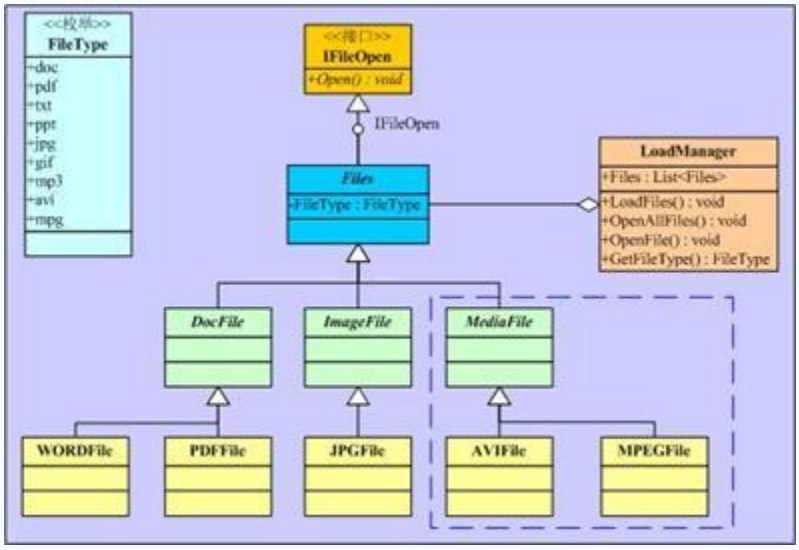


图 1-13 万能加载器架构设计调整

接着就是添加处理新文件的加载操作，如下：

```
lm.LoadFiles(new MPEGFile());
```

OK。添加新类型的操作就此完成，在没有对原系统进行修改的继承上，只需加入简单的类型和操作即可完成原来看似复杂的操作，结果证明新架构经得起考验，爷爷也为小王竖起了大拇指。事实证明，只要有更合理的设计与架构，在基于面向对象和.NET 框架的基础上，完全可以实现类似于插件的可扩展系统，并且无需编译即可更新扩展。

这一切是如何神奇般地实现了呢？回顾从设计到实现的各个环节，小王深知这都是源于多态机制的神奇力量，那么究竟什么是多态，.NET 中如何来实现多态呢？

1.4.6 多态的类型、本质和规则

从小王一系列大刀阔斧的改革中，我们不难发现是多态、是面向对象技术成就了 FileLoader 的强大与灵活。回过头来，结合 FileLoader 系统的实现分析，我们也可以从技术的角度来进一步探讨关于多态的话题。

1. 多态的分类

多态有多种分类的方式，Luca Cardelli 在《On Understanding Types, Data Abstraction, and Polymorphism》中将多态分为四类：强制的、重载的、参数的和包含的。本节可以理解为包含的多态，从面向对象的角度来看，根据其实现的方式我们可以进一步分为基类继承式多态和接口实现式多态。

（1）基类继承式多态

基类继承多态的关键是继承体系的设计与实现，在 FileLoader 系统中 File 类作为所有资料类型的基类，然后根据需求进行逐层设计，我们从架构设计图中可以清楚地了解继承体系关系。在客户端调用时，多态是以这种方式体现的：

```
Files myFile = new WORDFile();
myFile.Open();
```

myFile 是一个父类 Files 变量，保持了指向子类 WORDFile 实例的引用，然后调用一个虚方法 Open，而具体的调用则决定于运行时而非编译时。从设计模式角度看，基类继承式多态体现了一种 IS-A 方式，例如 WORDFile IS-A Files 就体现在这种继承关系中。

（2）接口实现式多态

多态并非仅仅体现在基于基类继承的机制中，接口的应用同样能体现多态的特性。区别于基类的继承方式，这种多态通过实现接口的方法约定形成继承体系，具有更高的灵活性。从设计模式的角度来看，接口实现式多态体现了一种 CAN-DO 关系。同样，在万能加载器的客户端调用时，也可以是这样的实现方式：

```
IFileOpen myFile = new WORDFile();
myFile.Open();
```

当然，很多时候这两种方式都是混合应用的，就像本节的 FileLoader 系统的实现方式。

2. 多态的运行机制

从技术实现角度来看，是.NET 的动态绑定机制成就了面向对象的多态特性。那么什么是动态绑定，.NET 又是如何实现动态绑定呢？这就是本节关于多态的运行机制所要探讨的问题。

动态绑定，又叫晚期绑定，是区别与静态绑定而言的。静态绑定在编译期就可以确定关联，一般是以方法重载来实现的；而动态绑定则在运行期通过检查虚拟方法表来确定动态关联覆写的方法，一般以继承和虚方法来实现。在.NET 中，虚方法以 virtual 关键字来标记，在子类中覆写的虚方法则以 override 关键字标记。从设计角度考量，通常将子类中共有的但却容易变化的特征抽取为虚函数在父类中定义，而在子类中通过覆写来重新实现其操作。


注意

严格来讲，.NET 中并不存在静态绑定。所有的.NET 源文件都首先被编译为 IL 代码和元数据，在方法执行时，IL 代码才被 JIT 编译器即时转换为本地 CPU 指令。JIT 编译发生于运行时，因此也就不存在完全在编译期建立的关联关系，静态绑定的概念也就无从谈起。本文此处仅是参照 C++等传统语言的绑定概念，读者应区别其本质。

关于.NET 通过什么方式来实现虚函数的动态绑定机制，详细情况请参阅本章 2.2 节“什么是继承”的详细描述。在此，我们提取万能加载器 FileLoader 中的部分代码，来深入分析通过虚方法进行动态绑定的一般过程：

```
abstract class Files: IFileOpen
{
   public abstract void Open();
   public void Delete()
   {
      //实现对文件的删除处理
   }
}
abstract class DocFile: Files
{
   public int GetPageCount()
   {
      //计算文档页数
   }
}
```

```
class WORDFile : DocFile
{
    public override void Open()
    {
        Console.WriteLine("Open the WORD file.");
    }
}
```

在继承体系的实现基础上，接着是客户端的实现部分：

```
Files myFile = new WORDFile();
myFile.Open();
```

针对上述示例，具体的调用过程，可以小结为：

编译器首先检查 myFile 的声明类型为 Files，然后查看 myFile 调用方法是否被实现为虚方法。如果不是虚方法，则直接执行即可；如果是虚方法，则会检查实现类型 WORDFile 是否重写该方法 Open，如果重写则调用 WORDFile 类中覆写的方法，例如本例中就将执行 WORDFile 类中覆写过的方法；如果没有重写，则向上递归遍历其父类，查找是否覆写该方法，直到找到第一个覆写方法调用才结束。

3. 多态的规则和意义

— 多态提供了对同一类对象的差异化处理方式，实现了对变化和共性的有效封装和继承，体现了“一个接口，多种方法”的思想，使方法抽象机制成为可能。

— 在.NET 中，默认情况下方法是非虚的，以 C#为例必须显式地通过 virtual 或者 abstract 标记为虚方法或者抽象方法，以便在子类中覆写父类方法。

— 在面向对象的基本要素中，多态和继承、多态和重载存在紧密的联系，正如前文所述多态的基础就是建立有效的继承体系，因此继承和重载是多态的实现基础。

1.4.7 结论

在爷爷大寿之际，小王终于完成了送给爷爷的生日礼物：万能加载器。看到爷爷轻松地玩着电脑，小王笑开了花，原来幸福是面向对象的。

在本节中，花了大量的笔墨来诠释设计架构和面向对象，或多或少有些喧宾夺主。然而，深入地了解多态及其应用，正是体现在设计模式、软件架构和面向对象的思想中；另一方面，也正是多态、继承和封装从技术角度成就了面向对象和设计模式，所以深入的理解多态就离不开大肆渲染以消化设计，这正是多态带来的艺术之美。

1.5 玩转接口

本节将介绍以下内容：

— 什么是接口

— 接口映射本质

— 面向接口编程

— 典型的.NET 接口

1.5.1 引言

接口，是面向对象设计中的重要元素，也是打开设计模式精要之门的钥匙。玩转接口，就意味着紧握这把钥匙，打开面向对象的抽象之门，成全设计原则、成就设计模式，实现集优雅和灵活于一身的代码艺术。

本节，从接口由来讲起，通过概念阐述、面向接口编程的分析以及.NET 框架中的典型接口实例，勾画一个理解接口的框架蓝图，通过这一蓝图将会了解玩转接口的学习曲线。

1.5.2 什么是接口

所谓接口，就是契约，用于规定一种规则由大家遵守。所以，.NET 中很多的接口都以 able 为命名后缀，例如 INullable、ICloneable、IEnumerable、IComparable 等，意指能够为空、能够克隆、能够枚举、能够对比，其实正是对契约的一种遵守寓意，只有实现了 ICloneable 接口的类型，才

允许其实例对象被拷贝。以社会契约而言，只有司机，才能够驾驶，人们必须遵守这种约定，无照驾驶将被视为犯罪而不被允许，这是社会契约的表现。由此来理解接口，才是对面向接口编程及其精髓的把握，例如：

```
interface IDriveable
{
   void Drive();
}
```

面向接口编程就意味着，在自定义类中想要有驾驶这种特性，就必须遵守这种契约，因此必须让自定义类实现 IDriveable 接口，从而才使其具有了“合法”的驾驶能力。例如：

```
public class BusDriver : IDriveable
{
   public void Drive()
   {
      Console.WriteLine("有经验的司机可以驾驶公共汽车。");
   }
}
```

没有实现 IDriveable 接口的类型，则不被允许具有 Drive 这一行为特性，所以接口是一组行为规范。例如要使用 foreach 语句迭代，其前提是操作类型必须实现 IEnumerable 接口，这也是一种契约。

实现接口还意味着，同样的方法对不同的对象表现为不同的行为。如果使司机具有驾驶拖拉机的能力，也必须实现 IDriveable 接口，并提供不同的行为方式，例如：

```
public class TractorDriver: IDriveable
{
   public void Drive()
   {
      Console.WriteLine("拖拉机司机驾驶拖拉机。");
   }
}
```

在面向对象世界里，接口是实现抽象机制的重要手段，通过接口实现可以部分的弥补继承和多态在纵向关系上的不足，具体的讨论可以参见 1.4 节“多态的艺术”和 7.4 节“面向抽象编程：接口和抽象类”。接口在抽象机制上，表现为基于接口的多态性，例如：

```
public static void Main()
{
   IList<IDriveable> drivers = new List<IDriveable>();
   drivers.Add(new BusDriver());
   drivers.Add(new CarDriver());
   drivers.Add(new TractorDriver());
   foreach (IDriveable driver in drivers)
   {
      driver.Drive();
   }
}
```

通过接口实现，同一个对象可以有不同的身份，这种设计的思想与实现，广泛存在于.NET 框架类库中，正是这种基于接口的设计成就了面向对象思想中很多了不起的设计模式。

1.5.3 .NET 中的接口

1. 接口多继承

在.NET 中，CLR 支持单实现继承和多接口继承。这意味着同一个对象可以代表多个不同的身份，以 DateTime 为例，其定义为：

```
public struct DateTime : IComparable, IFormattable, IConvertible, ISerializable,
                         IComparable<DateTime>, IEquatable<DateTime>
```

因此，可以通过 DateTime 实例代表多个身份，不同的身份具有不同的行为，例如：

```
public static void Main()
{
   DateTime dt = DateTime.Today;
   int result = ((IComparable)dt).CompareTo(DateTime.MaxValue);
```

```
    DateTime dt2 = ((IConvertible)dt).ToDateTime(new
                                System.Globalization.DateTimeFormatInfo());
}
```

2. 接口的本质

从概念上理解了接口，还应进一步从本质上揭示其映射机制，在.NET 中基于接口的多态究竟是如何被实现的呢？这是值得思考的话题，根据下面的示例，及其 IL 分析，我们对此进行一定的探讨：

```
interface IMyInterface
{
    void MyMethod();
}
```

该定义在 Reflector 中的 IL 为：

```
.class private interface abstract auto ansi IMyInterface
{
    .method public hidebysig newslot abstract virtual instance void MyMethod() cil managed
    {
    }
}
```

根据 IL 分析可知，IMyInterface 接口本质上仍然被标记为.class，同时提供了 abstract virtual 方法 MyMethod，因此接口其实本质上可以看作是一个定义了抽象方法的类，该类仅提供了方法的定义，而没有方法的实现，其功能由接口的实现类来完成，例如：

```
class MyClass : IMyInterface
{
    void IMyInterface.MyMethod()
    {
    }
}
```

其对应的 IL 代码为：

```
.class private auto ansi beforefieldinit MyClass
    extends [mscorlib]System.Object
    implements InsideDotNet.OOThink.Interface.IMyInterface
{
    .method public hidebysig specialname rtspecialname instance void .ctor() cil managed
    {
    }
    .method private hidebysig newslot virtual final instance void InsideDotNet.OOThink.Interface.IMyI
nterface.MyMethod() cil managed
    {
        .override InsideDotNet.OOThink.Interface.IMyInterface::MyMethod
    }
}
```

由此可见，实现了接口的类方法在 IL 标记为 override，表示覆写了接口方法实现，因此接口的抽象机制仍然是多态来完成的。接口在本质上，仍旧是一个不能实例化的类，但是又区别于一般意义上的类，例如不能实例化、允许多继承、可以作用于值类型等。

那么在 CLR 内部，接口的方法分派是如何被完成的呢？在托管堆中 CLR 维护着一个接口虚表来完成方法分派，该表基于方法表内的接口图信息创建，主要保存了接口实现的索引记录。以 IMyInterface 为例，在 MyClass 第一次加载时，CLR 检查到 MyClass 实现了 IMyInterface 的 MyMethod 方法，则会在接口虚表中创建一条记录信息，用于保存 MyClass 方法表中实现了 MyMethod 方法的引用地址，其他实现了 IMyInterface 的类型都会在接口虚表中创建相应的记录。因此，接口的方法调用是基于接口虚表进行的。

3. 由 string 所想到的：框架类库的典型接口

在.NET 框架类库中，存在大量的接口，以典型的 System.String 类型为例，就可知接口在 FCL 设计中的重要性：

public sealed class String : IComparable, ICloneable, IConvertible, Icomparable <string>, IEnumerable<char>, IEnumerable, IEquatable<string>

其中 IComparable<string>、IEnumerable<char>和 IEquatable<string>为泛型接口，具体的讨论可以参见 10.3 节“深入泛型”。

表 1.2 对几个典型的接口进行简要的分析，以便在 FCL 的探索中不会感觉陌生，同时也有助于熟悉框架类库。

表 1-2 FCL 的典型接口

接口名称	接口定义	功能说明

IComparable	public interface IComparable
	{
	int CompareTo(object obj);
	}
		提供了方法 CompareTo，用于对单个对象进行比较，实现 IComparable 接口的类需要自行提供排序比较函数。值类型比较会引起装箱与拆箱操作，IComparable<T>是它的泛型版本

IComparer	public interface IComparer
	{
	int Compare(object x, object y);
	}
		定义了为集合元素排序的方法 Compare，支持排序比较，因此实现 IComparer 接口的类型不需要自行实现排序操作。IComparer 接口同样存在装箱与拆箱问题，IComparer<T>是其泛型版本

IConvertible	public interface IConvertible
	{
	TypeCode GetTypeCode();
	bool ToBoolean(IFormatProvider provider);
	byte ToByte(IFormatProvider provider);
	char ToChar(IFormatProvider provider);
	int ToInt32(IFormatProvider provider);
	string ToString(IFormatProvider provider);
	object ToType(Type conversionType, IFormatProvider provider);
	//部分省略
	}
		提供了将类型的实例值转换为 CLR 标准类型的多个方法，在.NET 中，类 Convert 提供了公开的 IConvertible 方法，常用于类型的转换

ICloneable	public interface ICloneable
	{
	object Clone();
	}
		支持对象克隆，既可以实现浅拷贝，也可以实现深复制

IEnumerable	public interface IEnumerable
	{
	IEnumerator GetEnumerator();
	}
		公开枚举数，支持 foreach 语句，方法 GetEnumerator 用于返回 IEnumerator 枚举，IEnumerable<T>是它的泛型版本

续表

接口名称	接口定义	功能说明

IEnumerator	public interface IEnumerator
	{
	bool MoveNext();
	object Current { get; }
	void Reset();
	}
		是所有非泛型集合的枚举数基接口，可用于支持非泛型集合的迭代，IEnumerator<T>是它的泛型版本

IFormattable	public interface IFormattable
	{
	string ToString(string format, IFormatProvider formatProvider);
	}
		提供了将对象的值转化为字符串的形式

ISerializable	public interface ISerializable
	{
	[SecurityPermission(SecurityAction. LinkDemand, Flags = SecurityPermissionFlag. SerializationFormatter)]
	void GetObjectData(SerializationInfo info, StreamingContext context);
	}
		实现自定义序列化和反序列化控制方式，方法 GetObjectData 用于将对象进行序列化的数据存入 SerializationInfo 对象

IDisposable	public interface IDisposable
	{
	void Dispose();
	}
		对于非托管资源的释放，.NET 提供了两种模式：一种是终止化操作方式，一种是 Dispose 模式。实现 Dispose 模式的类型，必须实现 IDisposable 接口，用于显示的释放非托管资源

关于框架类库的接口讨论，在本书的各个部分均有所涉及，例如关于集合的若干接口 IList、ICollection、IDictionary 等在 7.9 节“集合通论”中有详细的讨论，在本书的学习过程中将会逐渐有所收获，在此仅做简要介绍。

1.5.4 面向接口的编程

设计模式的师祖 GoF，有句名言：Program to an interface, not an implementation，表示对接口编程而不要对实现编程，更通俗的说法是对抽象编程而不要对具体编程。关于面向对象和设计原则，将始终强调对抽象编程的重要性，这源于抽象代表了系统中相对稳定并又能够通过多态特性对其扩展，这很好地符合了高内聚、低耦合的设计思想。

下面，就以著名的 Petshop 4.0 中一个简单的面向对象设计片段为例，来诠释面向接口编程的奥秘。

在 Petshop 4.0 的数据访问层设计上，微软设计师将较为基础的增删改查操作封装为接口，由具体的实体操作类来实现。抽象出的单独接口模块，使得对于数据的操作和业务逻辑对象相分离。借鉴这种设计思路实现一个简单的用户操作数据访问层，其设计如图 1-14 所示。

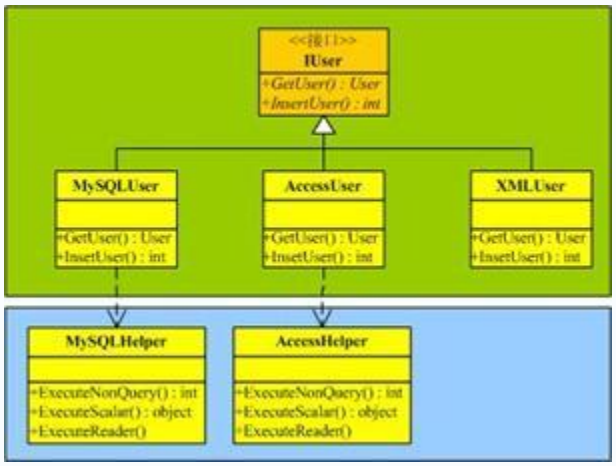


图 1-14 基于 Petshop 的数据访问层设计

从上述设计可见，通过接口将增删改查封装起来，再由具体的 MySQLUser、AccessUser 和 XMLUser 来实现，Helper 类则提供了操作数据的通用方法。基于接口的数据访问层和具体的数据操作

实现彻底隔离，对数据的操作规则的变更不会影响实体类对象的行为，体现了职责分离的设计原则，而这种机制是通过接口来完成的。

同时，能够以 IUser 接口来统一处理用户操作，例如在具体的实例创建时，可以借助反射机制，通过依赖注入来设计实现：

```
public sealed class DataAccessFactory
{
   private static readonly string assemblyPath = ConfigurationManager.AppSettings ["AssemblyPath
"];
   private static readonly string accessPath = ConfigurationManager.AppSettings ["AccessPath"];
   public static IUser CreateUser()
   {
      string className = accessPath + ".User";
      return (IUser)Assembly.Load(assemblyPath).CreateInstance(className);
   }
}
```

你看，通过抽象可以将未知的对象表现出来，通过读取配置文件的相关信息可以很容易创建具体的对象，当有新的类型增加时不需要对原来的系统做任何修改只要在配置文件中增加相应的类型全路径即可。这种方式体现了面向接口编程的另一个好处：对修改封闭而对扩展开放。

正是基于这种设计才形成了数据访问层、业务逻辑层和表现层三层架构的良好设计。而数据访问层是实现这一架构的基础，在业务逻辑层，将只有实体对象的相互操作，而不必关心具体的数据库操作实现，甚至看不到任何 SQL 语句执行的痕迹，例如：

```
public class BLL
{
   private static readonly IUser user = DataAccessFactory.CreateUser();
   private static User userInfo = new User();
   public static void HandleUserInfo(string ID)
   {
      userInfo = user.GetUser(ID);
      //对 userInfo 实体对象进行操作
```

```
    }
}
```

另外，按照接口隔离原则，接口应该被实现为具有单一功能的多个小接口，而不是具有多个功能的大接口。通过多个接口的不同组合，客户端按需实现不同的接口，从而避免出现接口污染的问题。

1.5.5 接口之规则

关于接口的规则，可以有以下的归纳：

- 接口隔离原则强调接口应该被实现为具有单一功能的小接口，而不要实现为具有多个功能的胖接口，类对于类的依赖应建立在最小的接口之上。
- 接口支持多继承，既可以作用于值类型，也可以作用于引用类型。
- 禁止为已经发布的接口，添加新的成员，这意味着你必须重新修改所有实现了该接口的类型，在实际的应用中，这往往是不可能完成的事情。
- 接口不能被实例化，没有构造函数，接口成员被隐式声明为 public。
- 接口可以作用于值类型和引用类型，并且支持多继承。

1.5.6 结论

通常而言，良好的设计必然是面向抽象的，接口是实现这一思想的完美手段之一。通过面向接口编程，保证了系统的职责清晰分离，实体与实体之间保持相对合适的耦合度，尤其是高层模块不再依赖于底层模块，而依赖于比较稳定的抽象，使得底层的更改不会波及到高层，实现了良好的设计架构。

透彻地了解接口，认识对接口编程，体会面向对象的设计原则，是培养一个良好设计习惯的开端。关于接口，是否玩的过瘾，就看如何体会本节强调的在概念上的契约，在设计上的抽象。

第 2 部分 本质——.NET 深入浅出

第 3 章 一切从 IL 开始

从 Hello, world 开始认识 IL

本文将介绍以下内容：

- IL 代码分析方法
- Hello, world 历史
- .NET 学习方法论

1. 引言

1988 年 Brian W. Kernighan 和 Dennis M. Ritchie 合著了软件史上的经典巨著《The C programming Language》，我推荐所有的程序人都有机会重温这本历史上的经典之作。从那时起，Hello, world 示例就作为了几乎所有实践型程序设计书籍的开篇代码，一直延续至今，除了表达对巨人与历史的尊重，本文也以 Hello, world 示例作为我们扣开 IL 语言的起点，开始我们循序渐进的 IL 认识之旅。

2. 从 Hello, world 开始

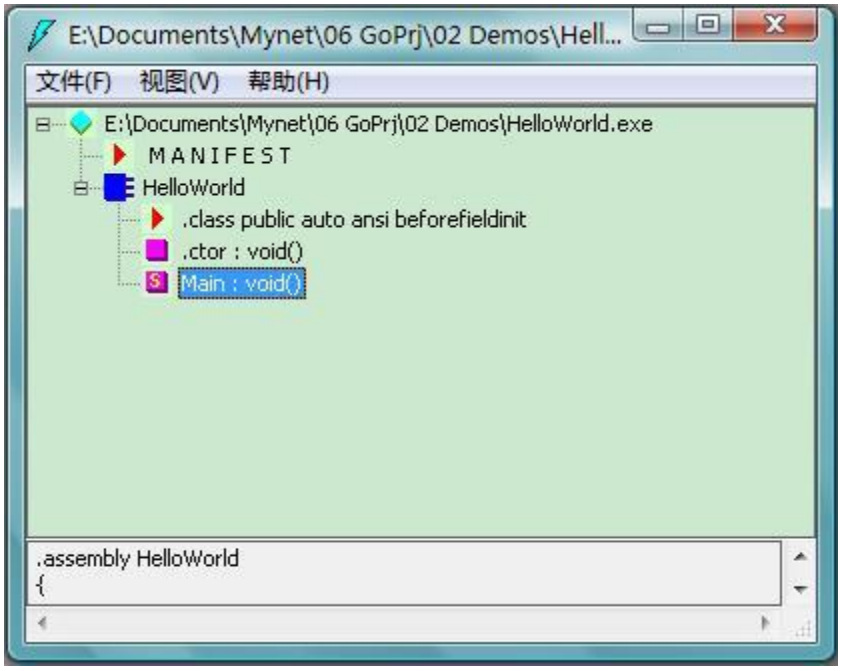
首先，当然是展示我们的 Hello, world 代码，开始一段有益的分享。

```
using System;
using System.Data;

public class HelloWorld
{
   public static void Main()
   {
      Console.WriteLine("Hello, world.");
   }
}
```

这段代码执行了最简单的过程，向陌生的世界打了一个招呼，那么运行在高级语言背后真相又是什么呢，下面开始我们基于上述示例的 IL 代码分析。

3. IL 体验中心

对编译后的可执行文件 HelloWorld.exe 应用 ILDasm.exe 反编译工具，还原 HelloWorld 的为文本 MSIL 编码，至于其工作原理我们期望在系列的后续文章中做以交代，我们查看其截图为：



由上图可知，编译后的 IL 结构中，包含了 MANIFEST 和 HelloWorld 类，其中 MANIFEST 是个附加信息列表，主要包含了程序集的一些属性，例如程序集名称、版本号、哈希算法、程序集模块等，以及对外部引用程序集的引用项；而 HelloWorld 类则是我们下面介绍的主角。

3.1 MANIFEST 清单分析

打开 MANIFEST 清单，我们可以看到

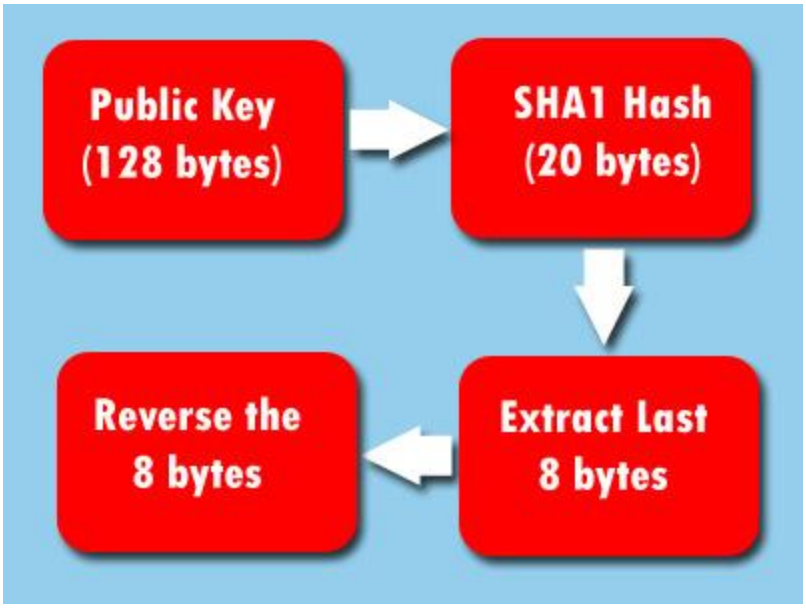
```
// Metadata version: v2.0.50727
.assembly extern mscorlib
{
  .publickeytoken = (B7 7A 5C 56 19 34 E0 89 )                    // .z\V.4..
  .ver 2:0:0:0
}
.assembly HelloWorld
{
  .custom instance void [mscorlib]System.Runtime.CompilerServices.CompilationRelaxationsAttribute::.ctor(int32) = ( 01 00 08 00 00 00 0
  .custom instance void [mscorlib]System.Runtime.CompilerServices.RuntimeCompatibilityAttribute::.ctor() = ( 01 00 01 00 54 02 16 57 72
                                                                                                             63 65 70 74 69 6F 6E 54 68 72 6F 77 73 01 )       // ceptionThrows.
  .hash algorithm 0x00008004
  .ver 0:0:0:0
}
.module HelloWorld.exe
// MVID: {4E594BE7-8736-4520-8BD0-FC43F60E2FBA}
.imagebase 0x00400000
.file alignment 0x00000200
.stackreserve 0x00100000
.subsystem 0x0003       // WINDOWS_CUI
.corflags 0x00000001    // ILONLY
// Image base: 0x01080000
```

从这段 IL 代码中，我们的分析如下：

- .assembly 指令用于定义编译目标或者加载外部库。在 IL 清单中可见，.assembly extern mscorlib 表示外部加载了外部核心库 mscorlib，而.assembly HelloWorld 则表示了定义的编译目标。值得注意的是，.assembly 将只显示程序中实际应用到的程序集列表，而对于加入 using 引用的程序集，如果并未在程序中引用，则编译器会忽略多加载的程序集，例如 System.Data 将被忽略，这样就有效避免了过度加载引起的代码膨胀。

- 我们知道 mscorlib.dll 程序集定义 managed code 依赖的核心数据类型，属于必须加载项。 例如接下来要分析的.ctor 指令表示构造函数，从代码中我们知道没有为 HelloWord 类提供任何显示的构造函数，因此可以肯定其继承自基类 System.Object，而这个 System.Object 就包含在 mscorlib 程序集中。

- 在外部指令中还会指明了引用版本（.ver）；应用程序实际公钥标记（.publickeytoken），公钥 Token 是 SHA1 哈希码的低 8 位字节的反序（如下图所示），用于唯一的确定程序集；还包括其他信息如语言文化等。



- HelloWorld 程序集中包括了.hash algorithm 指令，表示实现安全性所使用的哈希算法，系统缺省为 0x00008004，表明为 SHA1 算法；.ver 则表示了 HelloWorld 程序集的版本号；
- 程序集由模块组成， .module 为程序集指令，表明定义的模块的元数据，以指定当前模块。
- 其他的指令还有：imagebase 为影像基地址；.file alignment 为文件对齐数值；.subsystem 为连接系统类型，0x0003 表示从控制台运行；.corflags 为设置运行库头文件标志，默认为 1；这些指令不是我们研究的重点，详细的信息请参考 MSDN 相关信息。

3.2 HelloWorld 类分析

首先是 HelloWorld 类，代码为：

```
.class public auto ansi beforefieldinit HelloWorld
       extends [mscorlib]System.Object
{
} // end of class HelloWorld
```

- .class 表明了 HelloWorld 是一个 public 类，该类继承自外部程序集 mscorlib 的 System.Object 类。
- public 为访问控制权限，这点很容易理解。
- auto 表明程序加载时内存的布局是由 CLR 决定的，而不是程序本身

- ansi 属性则为了在没有被管理和被管理代码间实现无缝转换。没有被管理的代码，指的是没有运行在 CLR 运行库之上的代码，例如原来的 C，C++代码等。

- beforefieldinit 属性为 HelloWorld 提供了一个附加信息，用于标记运行库可以在任何时候执行类型构造函数方法，只要该方法在第一次访问其静态字段之前执行即可。如果没有 beforefieldinit 则运行库必须在某个精确时间执行类型构造函数方法，从而影响性能优化，详细的情况可以参与 MSDN 相关内容。

然后是.ctor 方法，代码为：

```
.method public hidebysig specialname rtspecialname
        instance void  .ctor() cil managed
{
 // 代码大小       7 (0x7)
 .maxstack  8
 IL_0000:  ldarg.0
 IL_0001:  call       instance void [mscorlib]System.Object::.ctor()
 IL_0006:  ret
} // end of method HelloWorld::.ctor
```

- cil managed 说明方法体中为 IL 代码，指示编译器编译为托管代码。

- .maxstack 表明执行构造函数.ctor 期间的评估堆栈（Evaluation Stack）可容纳数据项的最大个数。关于评估堆栈，其用于保存方法所需变量的值，并在方法执行结束时清空，或者存储一个返回值。

- IL_0000，是一个标记代码行开头，一般来说，IL_之前的部分为变量的声明和初始化。

- ldarg.0 表示装载第一个成员参数，在实例方法中指的是当前实例的引用，该引用将用于在基类构造函数中调用。

- call 指令一般用于调用静态方法，因为静态方法是在编译期指定的，而在此调用的是构造函数.ctor()也是在编译期指定的；而另一个指令 callvirt 则表示调用实例方法，它的调用过程有异于 call，函数的调用是在运行时确定的，首先会检查被调用函数是否为虚函数，如果不是就直接调用，如果是则向下检查子类是否有重写，如果有就调用重写实现，如果没有还调用原来的函数，依次类推直到找到最新的重写实现。

- ret 表示执行完毕，返回。

最后是 Main 方法，代码为：

```
.method public hidebysig static void  Main() cil managed
{
 .entrypoint
 // 代码大小       11 (0xb)
 .maxstack  8
 IL_0000:  ldstr      "Hello, world."
 IL_0005:  call       void [mscorlib]System.Console::WriteLine(string)
 IL_000a:  ret
} // end of method HelloWorld::Main
```

- .entrypoint 指令表明了 CLR 加载程序 HelloWorld.exe 时，是首先从.entrypoint 方法开始执行的，也就是表明 Main 方法将作为程序的入口函数。每个托管程序必须有并且只有一个入口点。这区别于将 Main 函数作为程序入口标志。

- ldstr 指令表示将字符串压栈，"Hello, world."字符串将被移到 stack 顶部。CLR 通过从元数据表中获得文字常量来构造 string 对象，值得注意的是，在此构造 string 对象并未出现在《第五回：深入浅出关键字---把 new 说透》中提到的 newobj 指令，对于这一点的解释我们将在下一回中做简要分析。

- hidebysig 属性用于表示如果当前类作为父类时，类中的方法不会被子类继承，因此 HelloWorld 子类中不会看到 Main 方法。

接下来的一点补充：

- 关于注释，IL 代码中的注释和 C#等高级语言的注释相同，其实编译器在编译 IL 代码时已经将所有的注释去掉，所以任何对程序的注释在 IL 代码中是看不见的。

3.3 回归简洁

去粗取精，我们的 IL 代码可以简化，下面的代码是基于上面的分析，并去处不重要的信息，以更简洁的方式来展现的 HelloWorld 版 IL 代码，详细的分析就以注释来展开吧。

```
// FileName    : Anytao.net.My_Must_net(HelloWorldIL.cs)
// Description : .NET IL analyse
// Release      : 2007/07/21 1.0
// Copyright   : (C)2007 Anytao.com  http://www.anytao.com

//加载外部程序集
.assembly extern mscorlib
//指定编译目标程序集
.assembly HelloWorld

.class HelloWorld extends [mscorlib]System.Object
{
  .method public instanct void .ctor() cil managed
  {
    .maxstack 8

    //调研父类构造函数
    ldarg.0
    call instance void [mscorlib]System.Object::.ctor()
    //执行完毕,返回
    ret
  }

  .method static void Main() cil managed
  {
    //表明程序入口点
    .entrypoint
    .maxstack 8
    //装载string对象
    ldstr "Hello, world."
    //调用WriteLine方法
    call void [mscorlib]System.Console::WriteLine(string)
    ret
  }
}
```

4. 结论

结束本文，我们从一个点的角度和 IL 来了一次接触，除了了解几个重要的指令含义，更重要的是已经走进了 IL 的世界。通过一站式的扫描 HelloWorld 的 IL 编码，我们还不足以从全局来了解 IL，不过第一次的亲密接触至少让我们太陌生，而且随着系列文章的深入我们将逐渐建立起这种认知，从而提高我们掌握了解.NET 底层的有效工具。本系列也将在后续的文章中，逐渐建立起这种使用工具的方法，敬请关注。

3.2 教你认识 IL 代码---从基础到工具

本文将介绍以下内容：

- IL 代码分析方法

- IL 命令解析
- .NET 学习方法论

1. 引言

自从『你必须知道.NET』系列开篇以来，受到大家很多的关注和支持，给予了 anytao 巨大的鼓励和动力。俱往昔，我发现很多的园友都把目光和焦点注意在如何理解 IL 代码这个问题上。对我来说，这真是个莫大的好消息，因为很明显我们的思路慢慢的从应用向底层发生着转变，技巧性的东西是一个方面的积累，底层的探索在我认为也是必不可少的修炼。如果我们选择了来关注这项修炼，那么我们就应该选择如何来着手这项修炼，首先关注 anytao 的『你必须知道的.NET』系列可以给你提供一个捷径，少花一些功夫；其次对大师级的作品也应有更深入的了解，如《Applied Microsoft .NET Framework Programming》、《.NET 本质论》；再次，就是像我一样从博客园和 MSDN 的知识库中不断的成长。呵呵，除了给自己做了个广告之外，我认为不管是何种途径，了解和认识 IL 代码，对于我们更深刻的理解.NET 和.NET 应用之上的本质绝对有不一样的收获，这也就是本文研究和分享的理由。

那么，我们要了解 IL 代码，就要知道了解 IL 的好处，时间对每个程序设计师来说都是宝贵的，你必须清楚自己投资的价值再决定投入的资本。对于.NET 程序员来说，IL 代码意味着：

- 通用的语言基础是.NET 运行的基础，当我们对程序运行的结果有异议的时候，如何透过本质看表面，需要我们从本质入手来探索，这时 IL 是你必须知道的基础；
- 元数据和 IL 语言是 CLR 的基础，了解必要的中间语言是深入认识 CLR 的捷径；
- 大量的事例分析是以 IL 来揭密的，因此了解 IL 是读懂他人代码的必备基础，可以给自己更多收获。

很明显这些优越性足以诱惑我们花时间和精力涉猎其中。然而，了解了 IL 的好处，并不意味着我们应该过分的来关注 IL，有人甚至可以洋洋洒洒的写一堆 IL 代码来实现一个简单 Hello world 程序，但是正如我们知道的那样，程序设计已经走过了几十年的发展，如果纯粹的陶醉在历史中，除了脑子不好，没有其他的解释。不然看见任何代码都以 IL 的角度来分析，又将走进另一个误区，我们的宗旨是追求但不过分。

因此，有了上述了应该了解的理由和不应该过分的基线，在摆正心态的前提下，本文开始以作者认为的方式来展开对 IL 代码的认识，作者期望通过本文的阐述与分析使得大

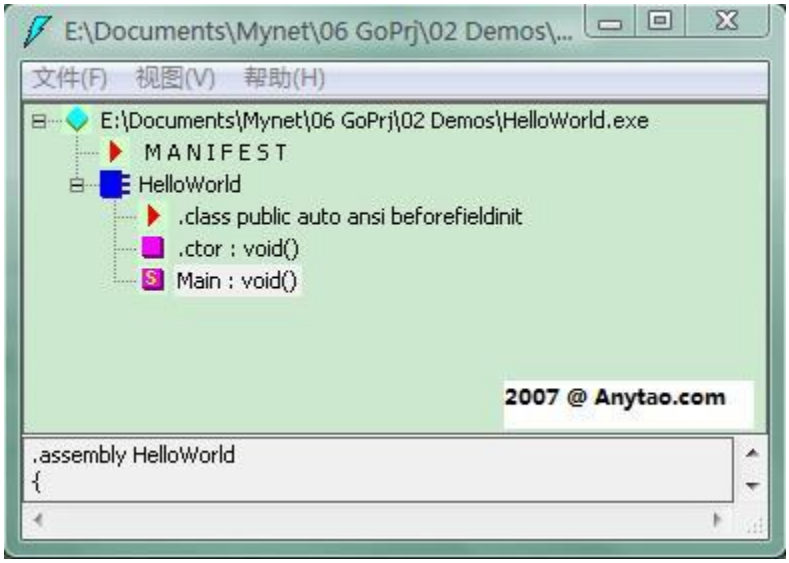
家都能对 IL 有个概观之解，并在平时的项目实践中使用这种方法通过了解自己的代码来了解.NET。我想，这种方法应该是值得提倡和发挥的最佳实践，不知你信不信呢？呵呵。

2. 使用工具

俗话说，工欲善其事，必先利其器。IL 的器主要就是 ILadsm.exe 和 reflector.exe，这两个工具都是了解 IL 的基础，其原理都是通过反射机制来查看 IL 代码。

- ILadsm.exe

打开.NET Framework SKD 命令提示行，输入 ildasm 回车即可打开，如图所示：



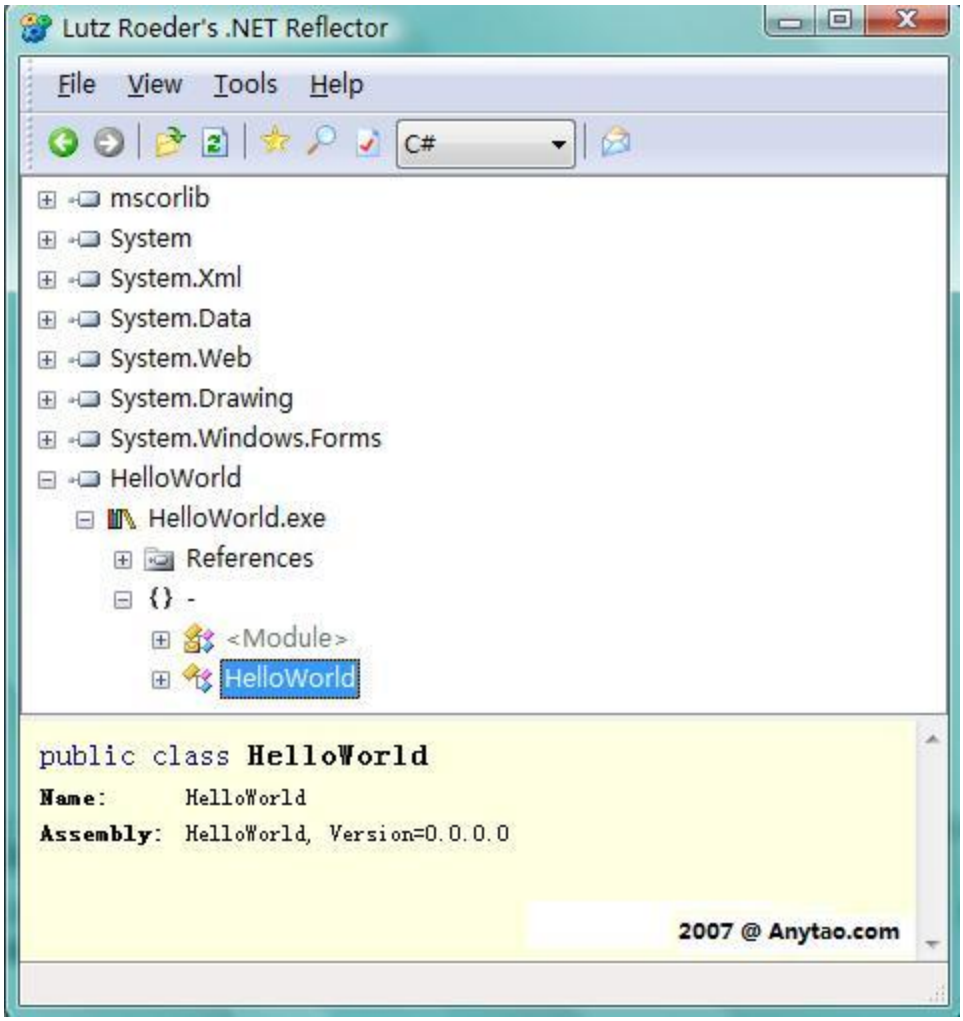
上图是我们熟悉的《第十三回：从 Hello, world 开始认识 IL》中的示例，其中的树形符号代表的意思，可以从 MSDN 的一张经典帮助示例来解释，如下图所示：

符号	含义
	更多信息
	命名空间
	类
	接口
	值类
	枚举
	方法
	静态方法
	字段
	静态字段
	事件
	属性
	清单或类信息项

（图表来源：MSDN）

- reflector.exe【下载】

Reflector 是 Lutz Roeder 开发的一个让人兴奋的反编译利器，目前的版本是 Version 5.0.35.0，可以支持.NET3.0，其功能也相当强大，在使用上也较 ILDASM 更加灵活，如图所示：



Reflector 可以方便的反编译为 IL、C#、VB、Delphi 等多种语言，是深入了解 IL 的最佳利器。

在本文中我们以最简单的 ILadsm.exe 为说明工具。

3. 分析结构

分析 IL 结构，就参阅《第十三回：从 Hello, world 开始认识 IL》 ，已经有了大致的介绍，在此不需要进行过多的笔墨，实际上 IL 的本身的结构也不是很复杂，了解了大致的体系即可。

4. 解析常用命令

我们在了解了 IL 文件结构的基础上，通过学习常用的 IL 命令，就可以基本上对 IL 达到了了解不过分的标准，因此对 IL 常用命令的分析就是本文的重点和要点。我们通过对常用命令的解释、示例与分析，逐步了解你陌生的语言世界原来也很简单。

IL 指令集包括了基础指令集和对象模型指令集大概有近 200 多个，对我们来说消化这么多的陌生指令显然不是明智的办法，就行高级语言的关键字一样，我们只取其一瓢独饮，抓大放小的革命传统同样是有效的学习办法，详细的指令集解释请下载[MSIL 指令速查手册]。

4.1 newobj 和 initobj

newobj 和 intiobj 指令就像两个兄弟，常常让我们迷惑在其然而不知其所以然，虽然认识但是不怎么清楚，这种感觉很郁闷，下面就让我们看看他们的究竟：

代码引入

```
// FileName     : Anytao.net.My_Must_net
// Description  : The .NET what you should know of arguments.
// Release      : 2007/08/21 1.0
// Copyright    : (C)2007 Anytao.com http://www.anytao.com
using System;
namespace Anytao.net.My_Must_net.IL
{
    struct MyStruct
    {    }
    class MyClass
    {    }
    class ILDemo
    {
        public static void Main()
        {
            MyStruct ms = new MyStruct();
            MyClass mc = new MyClass();
            Console.WriteLine("IL Keywords.");
            Console.Read();
        }
    }
}
```

指令说明

```
  .method public hidebysig static void  Main() cil managed
{
  .entrypoint
  // 代码大小              33 (0x21)
  .maxstack  1
  .locals init (valuetype Anytao.net.My_Must_net.IL.MyStruct V_0,
                class Anytao.net.My_Must_net.IL.MyClass V_1)
  IL_0000:  nop
  IL_0001:  ldloca.s    V_0
  IL_0003:  initobj     Anytao.net.My_Must_net.IL.MyStruct
  IL_0009:  newobj      instance void
Anytao.net.My_Must_net.IL.MyClass::.ctor()
  IL_000e:  stloc.1
  IL_000f:  ldstr       "IL Keywords."
  IL_0014:  call        void
[mscorlib]System.Console::WriteLine(string)
  IL_0019:  nop
  IL_001a:  call        int32 [mscorlib]System.Console::Read()
  IL_001f:  pop
  IL_0020:  ret
} // end of method ILDemo::Main
```

深入分析

从上面的代码中，我们可以得出哪些值得推敲的结论呢？

MSDN 给出的解释是：newobj 用于分配和初始化对象；而 initobj 用于初始化值类型。

那么 newobj 又是如何分配内存，完成对象初始化；而 initobj 又如何完成对值类型的初始化呢？

显然，关于 newobj 指令，在《**第五回：深入浅出关键字---把 NEW 说透**》中，已经有了一定的介绍，简单说来关于 newobj 我们有如下结论：

- 从托管堆分配指定类型所需要的全部内存空间。
- 在调用执行构造函数初始化之前，首先初始化对象附加成员：一个是指向该类型方法表的指针；一个是 SyncBlockIndex，用于进行线程同步。所有的对象都包含这两个附加成员，用于管理对象。
- 最后才是调用构造函数 ctor，进行初始化操作。并返回新建对象的引用地址。

而 initobj 的作用又可以小结为：

- 构造新的值类型，完成值类型初始化。值得关注的是，这种构造不需要调用值类型的构造函数。具体的执行过程呢？以上例来说，initobj MyStruct 的执行结果是，将 MyStruct 中的引用类型初时化为 null，而基元类型则置为 0。

因此，值类型的初始化可以是：

```
//initobj 方式初始化值类型

initobj      Anytao.net.My_Must_net.IL.MyStruct
```

同时，也可以直接显示调用构造函数来完成初始化，具体为

```
MyStruct ms = new MyStruct(123);
```

对应于 IL 则是对构造函数 cto 的调用。

```
//调用构造函数方式初始化值类型

call          instance void Anytao.net.My_Must_net.IL.MyStruct::.ctor(int32)
```

- Initobj 还用于完成设定对指定存储单元的指针置空（null）。这一操作虽不常见，但是应该引起注意。

由此可见，newobj 和 initobj，都具有完成实例初始化的功能，但是针对的类型不同，执行的过程有异。其区别主要包括：

- newobj 用于分配和初始化对象；而 initobj 用于初始化值类型。因此，可以说，newobj 在堆中分配内存，并完成初始化；而 initobj 则是对栈上已经分配好的内存，进行初始化即可，因此值类型在编译期已经在栈上分配好了内存。
- newobj 在初始化过程中会调用构造函数；而 initobj 不会调用构造函数，而是直接对实例置空。
- newobj 有内存分配的过程；而 initobj 则只完成数据初始化操作。

关于对象的创建，还有其他的情况值得注意，例如：

- Newarr 指令用来创建一维从零起始的数组；而多维或非从零起始的一维数组，则仍由 newobj 指令创建。
- String 类型的创建由 ldstr 指令来完成，具体的讨论我们在下文来展开。

4.2 call、callvirt 和 calli

call、callvirt 和 calli 指令用于完成方法调用，这些正是我们在 IL 中再熟悉不过的几个朋友。那么，同样是作为方法调用，这几位又有何区别呢？我们首先对其做以概括性的描述，再来通过代码与实例，进入深入分析层面。

- call 使用静态调度，也就是根据引用类型的静态类型来调度方法。
- callvirt 使用虚拟调度，也就是根据引用类型的动态类型来调度方法；
- calli 又称间接调用，是通过函数指针来执行方法调用；对应的直接调用当然就是前面的：call 和 callvirt。

然而，虽然有以上的通用性结论，但是对于 call 和 callvirt 不可一概而论。call 在某种情况下可以调用虚方法，而 callvirt 也可以调用非虚方法。具体的分析我们在以后的文章中来展开，暂不做过多分析。

5. 结论

本文从几个重点的 IL 指令开始，力求通过对比性的分析和深入来逐步揭开 IL 的神秘与迷惑，正如我们在开始强调的那样，本文只是个开始也许也是个阶段，对 IL 的探求正如我自己的脚步一样，也在继续着，为的是在.NET 的技术世界能够有更多的领悟。作者期望通过不断的努力逐渐和大家一起从 IL 世界探求.NET 世界，在以后的讨论中我们间或的继续这个主题的不断成长。

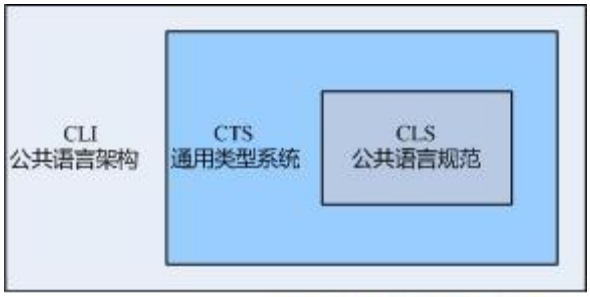
第 4 章 品味类型

4．1 品味类型---从通用类型系统开始

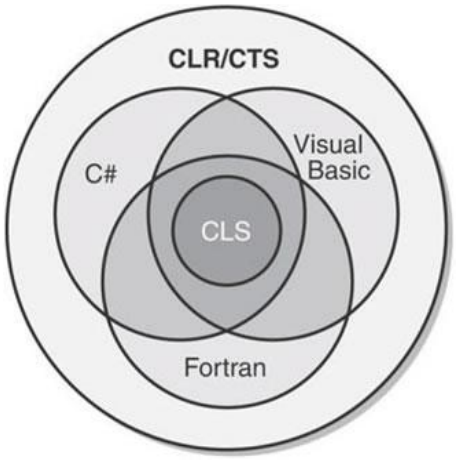
本文将介绍以下内容：

- .NET 基础架构概念
- 类型基础
- 通用类型系统
- CLI、CTS、CLS 的关系简述

1. 引言



©2007 Anytao.com



本文不是连环画，之所以在开篇以图形的形式来展示本文主题，其实就是想更加特别的强调这几个概念的重要性和关注度，同时希望从剖析其关系和联系的角度来讲述.NET Framework 背后的故事。因为，在作者看来想要深入的了解.NET，必须首先从了解类型开始，因为 CLR 技术就是基于类型而展开的。而了解类型则有必要把焦点放在.NET 类型体系的公共基础架构上，这就是：通用类型系统（Common Type System, CTS）。

我之所以将最基本的内容以独立的章节来大加笔墨，除了为后面几篇关于对类型这一话题深入讨论做以铺垫之外，更重要的是从论坛上、博客间，我发现有很多同行对.NET Framework 基础架构的几个重要体系的理解有所偏差，因此很有必要补上这一课，必备我们在深入探索知识的过程中，能够游刃有余。

2. 基本概念

还是老套路，首先引入 MSDN 对通用类型系统的定义，通用类型系统定义了如何在运行库中声明、使用和管理类型，同时也是运行库支持跨语言集成的一个重要组成部分。通用类型系统执行以下功能：

- 建立一个支持跨语言集成、类型安全和高性能代码执行的框架。

- 提供一个支持完整实现多种编程语言的面向对象的模型。
- 定义各语言必须遵守的规则，有助于确保用不同语言编写的对象能够交互作用。

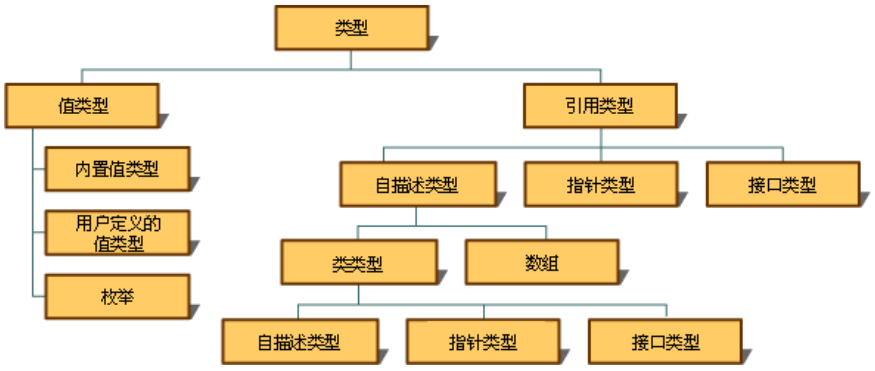
那么我们如何来理解呢？

还是一个现实的场景来引入讨论吧。小王以前是个 VB 迷，写了一堆的 VB.NET 代码，现在他变心了，就投靠 C#的阵营，因为流行嘛。所以当然就想在当前的基于 C#开发的项目中，应用原来 VB.NET 现成的东西，省点事儿:-)。那么 CLR 是如何来实现类型的转换的，例如 Dim i as Single 变量 i，编译器会自动的实现将 i 由 Single 到 float 的映射，当然其原因是所有的.NET 编译器都是基于 CLS 实现的。具体的过程为：CTS 定义了在 MSIL 中使用的预定义数据类型，.NET 语言最终都要编译为 IL 代码，也就是所有的类型最终都要基于这些预定义的类型，例如应用 ILDasm.exe 分析可知，VB.NET 中 Single 类型映射为 IL 类型就是 float32，而 C#中 float 类型也映射为 float32，由此就可以建立起 VB.NET 和 C#的类型关系，为互操作打下基础。

```
.method public hidebysig static void Main(string[] args) cil managed
{
.entrypoint
// 代码大小 15 (0xf)
.maxstack 1
.locals init (float32 V_0)
IL_0000: nop
IL_0001: ldc.r4 1.
IL_0006: stloc.0
IL_0007: ldloc.0
IL_0008: call void [mscorlib]System.Console::WriteLine(float32)
IL_000d: nop
IL_000e: ret
} // end of method BaseCts::Main
```

过去，由于各个语言在类型定义方面的不一致，造成跨语言编程实现的难度，基于这一问题，.NET 中引入 CTS 来解决各个编程语言类型不一致的问题，类型机制使得多语言的代码可以无缝集成。因此 CTS 也成为.NET 跨语言编程的基础规范，为多语言的互操作提供了便捷之道。可以简单的说，基于.NET 的语言共同使用一个类型系统，这就是 CTS。

进一步的探讨通用类型系统的内容，我们知道 CTS 支持两种基本的类型，每种类型又可以细分出其下级子类，可以以下图来表示：

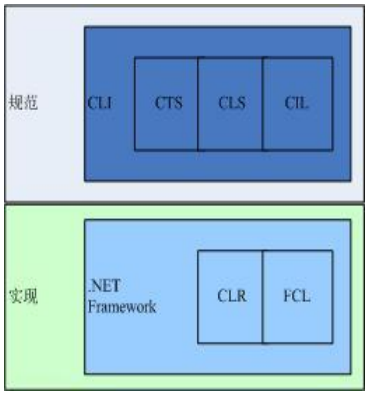


.NET 提供了丰富的类型层次结构，从上图中也可以看出该层次结构是基于单继承层次实现的，反映了.NET 面向对象原则中实现单继承、接口多继承的特点。关于值类型和引用类型，是之后要探讨的重点内容，也是『品味类型』子系列的重中之重，在此不作进一步探讨，但是上面的这张图有必要清楚的印在心中，因为没有什么比这个更基础的了。

3. 位置与关系

位置强调的是 CTS 在.NET 技术框架中的位置和作用，作者期望以这种方式来自然的引出.NET 技术架构的其他基本内容，从而在各个技术要点的层次中，来讲明白各个技术要点的些细联系，从大局的角度来对其有个基本的把握。我想，这样也可以更好的理解 CTS 本身，因为技术从来都不是孤立存在的。

.NET 技术可以以规范和实现两部分来划分，而我们经常强调和提起的.NET Framwork，主要包括公共语言运行时（Common Language Runtime, CLR）和.NET 框架类库（Framework Class Library, FCL），其实是对.NET 规范的实现。而另外一部分：规范，我们称之为公共语言架构（Common Language Infrastructure, CLI），主要包括通用类型系统（CTS），公共语言规范（Common Language Specification, CLS）和通用中间语言（Common Intermediate Language, CIL）。我们以图的形式来看看 CTS 在.NET 技术阵营中的位置，再来简要的介绍新登场的各个明星。



- CLI，.NET 技术规范，已经得到 ECMA（欧洲计算机制造商协会）组织的批准实现了标注化。

- CTS，本文主题，此不冗述。
- CLS，定义了 CTS 的子集，开发基于 CTS 的编译器，则必须遵守 CLS 规则，由本文开头的图中就可以看出 CLS 是面向.NET 的开发语言必须支持的最小集合。
- CIL，是一种基于堆栈的语言，是任何.NET 语言编译产生的中间代码，我们可以理解为 IL 就是 CLR 的汇编语言。IL 定义了一套与处理器无关的虚拟指令集，与 CLR/CTS 的规则进行映射，执行 IL 都会翻译为本地机器语言来执行。常见的指令有：add, box, call, newobj, unbox。另外，IL 很类似于 Java 世界里的字节码（Bytecode），当然也完全不是一回事，最主要的区别是 IL 是即时编译（Just in time, JIT）方式，而 Bytecode 是解释性编译，显然效率上更胜一筹。
- .NET Framework，可以说是 CLI 在 windows 平台的实现，运行与 windows 平台之上。
- CLR，.NET 框架核心，也是本系列的核心。类似于 Java 世界的 JVM，主要的功能是：管理代码执行，提供 CTS 和基础性服务。对 CLR 的探讨，将伴随着这个系列的成长来慢慢展开，在此就不多说了。
- FCL，提供了一整套的标准类型，以命名空间组织成树状形式，树的根是 System。对程序设计人员来说，学习和熟悉 FCL 是突破设计水平的必经之路，因为其中数以万计的类帮助我们完成了程序设计绝大部分的基础性工作，重要的是我们要知道如何去使用。

可见，这些基本内容相互联系，以简单的笔墨来澄清其概念、联系和功能，显然还不够力度。然而在此我们以抛砖引玉的方式来引入对这些知识的探求，目的是给一个入口，从此来进行更深入的探索是每个设计人员的成长的关键，就像对 FCL 的认识，需要实践，需要时间，需要心思。

4. 通用规则

- .NET 中，所有的类型都继承自 System.Object 类。
- 类型转换，通常有 is 和 as 两种方式，具体的探讨可以参考我的另一拙作《第一回：恩怨情仇：is 和 as》。另外，还有另外的几个类型转换的方式：（typename）valuename，是通用方法；Convert 类提供了灵活的类型转换封装；Parse 方法，适用于向数字类型的转换。
- 可以给类型创建别名，例如，using mynet = Anytao.net.MyClass，其好处是当需要有两个命名空间的同名类型时，可以清楚的做以区别，例如：

```
using AClass = Anytao.net.MyClass;
using BClass = Anytao.com.MyClass;
```

其实，我们常用的 int、char、string 对应的是 System.Int32、System.Char、System.String 的别名。

- 一个对象获得类型的办法是：obj.GetType()。
- Typeof 操作符，则常在反射时，获得自定义类型的 Type 对象，从而获取关于该类型的方法、属性等。
- 可以使用 CLSCompliantAttribute 将程序集、模块、类型和成员标记为符合 CLS 或不符合 CLS。

- IL 中使用/checked+开关来进行基元类型的溢出检查，在 C#中实现这一功能的是 checked 和 unchecked 操作符。
- 命名空间是从功能角度对类型的划分，是一组类型在逻辑上的集合。

5. 结论

类型的话题，是个老掉牙的囫囵觉，但也是个永不言退的革命党。在实际的程序设计中，我们经常要吃这一亏。因为，很多异常的产生，很多性能的损耗，很多冗余的设计都和类型解下不解之缘，所以清晰、清楚的了解类型，没有什么不可以。重要的是，我们以什么角度来了解和化解，内功的修炼还是要从内力开始。本系列不求包罗万象，但求以更新鲜、更全面的角度，清楚、干净、深入的把某个问题说透，此足尹。

品味类型，就从 CTS 开始了。

4.2 品味类型——品味类型---值类型与引用类型（上）－内存有理

本文将介绍以下内容：

- 类型的基本概念
- 值类型深入
- 引用类型深入
- 值类型与引用类型的比较及应用

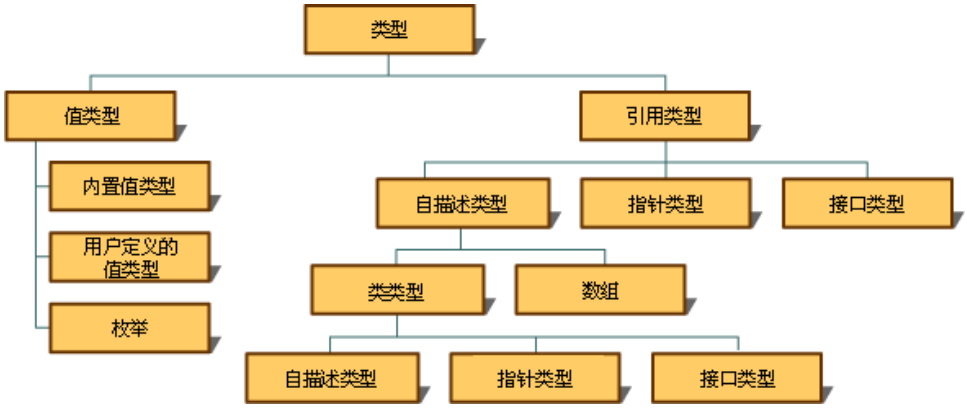
1. 引言

买了新本本，忙了好几天系统，终于开始了对值类型和引用类型做个全面的讲述了，本系列开篇之时就是因为想写这个主题，才有了写个系列的想法。所以对值类型和引用类型的分析，是我最想成文的一篇，其原因是过去的学习过程中我就是从这个主题开始，喜欢以 IL 语言来分析执行，也喜好从底层的过程来深入了解。这对我来说，似乎是一件找到了有效提高的方法，所以想写的冲动就没有停过，旨在以有效的方式来分享所得。同时，我也认为，对值类型和引用类型的把握，是理解语言基础环节的关键主题，有必要花力气来了解和深入。

2. 一切从内存开始

2.1 基本概念

从上回《第七回：品味类型---从通用类型系统开始》我们知道，CLR 支持两种基本类型：**值类型**和**引用类型**。因此，还是把 MSDN 这张经典视图拿出来做个铺垫。



值类型（Value Type），值类型实例通常分配在线程的堆栈（stack）上，并且不包含任何指向实例数据的指针，因为变量本身就包含了其实例数据。其在 MSDN 的定义为值类型直接包含它们的数据，值类型的实例要么在堆栈上，要么内联在结构中。我们由上图可知，值类型主要包括简单类型、结构体类型和枚举类型等。通常声明为以下类型：int、char、float、long、bool、double、struct、enum、short、byte、decimal、sbyte、uint、ulong、ushort 等时，该变量即为值类型。

引用类型（Reference Type），引用类型实例分配在托管堆（managed heap）上，变量保存了实例数据的内存引用。其在 MSDN 中的定义为引用类型存储对值的内存地址的引用，位于堆上。我们由上图可知，引用类型可以是自描述类型、指针类型或接口类型。而自描述类型进一步细分成数组和类类型。类类型是则可以是用户定义的类、装箱的值类型和委托。通常声明为以下类型：class、interface、delegate、object、string 以及其他的自定义引用类型时，该变量即为引用类型。

下面简单的列出我们类型的进一步细分，数据来自 MSDN，为的是给我们的概念中有清晰的类型概念，这是最基础也是最必须的内容。

类　别		描　述
值类型	简单类型	有符号整型：sbyte,short,int,long
		无符号整型：byte,ushort,uint,ulong
		Unicode字符：char
		IEEE浮点型：float,double
		高精度小数：decimal
		布尔型：bool
	枚举类型	用户自定义类型：enum
	结构类型	用户自定义类型：struct
引用类型	类类型	所有其他类型的最终基类：object
		Unicode字符串：string
		用户自定义类型：class
	接口类型	用户自定义类型：interface
	数组类型	单维与多维数组，例如，int[]与int[,]
	委托类型	用户自定义类型：delegate
www.anytao.com		

2.2 内存深入

2.2.1. 内存机制

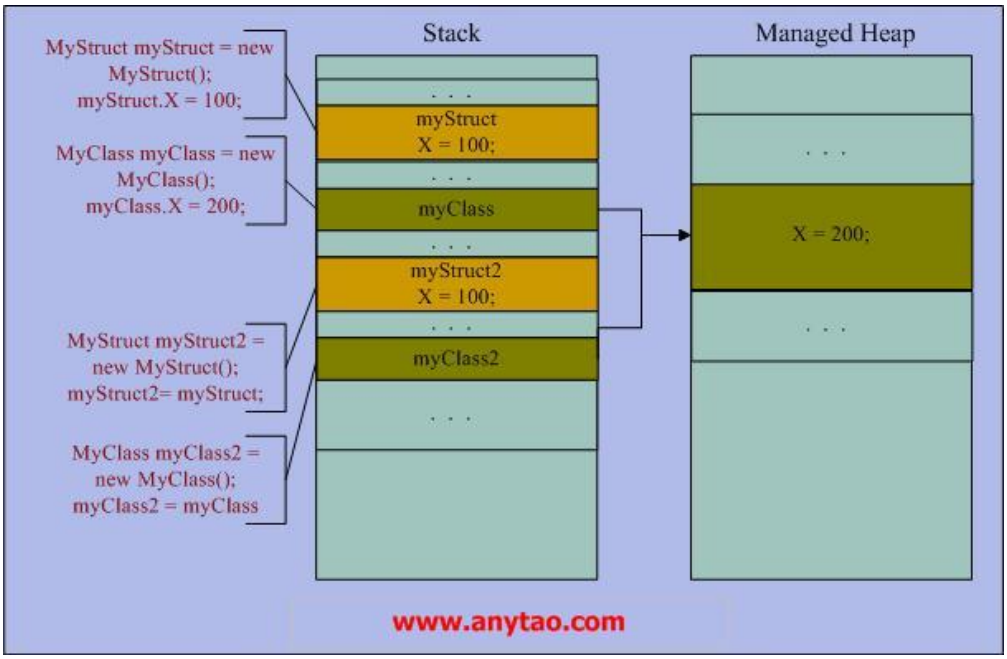
那么.NET 的内存分配机制如何呢？

数据在内存中的分配位置，取决于该变量的数据类型。由上可知，值类型通常分配在线程的堆栈上，而引用类型通常分配在托管堆上，由 GC 来控制其回收。例如，现在有 MyStruct 和 MyClass 分别代表一个结构体和一个类，如下：

```
using System;

public class Test
{
   static void Main()
   {
      //定义值类型和引用类型，并完成初始化
      MyStruct myStruct = new MyStruct();
```

```
        MyClass myClass = new MyClass();

        //定义另一个值类型和引用类型，
        //以便了解其内存区别
        MyStruct myStruct2 = new MyStruct();
        myStruct2 = myStruct;

        MyClass myClass2 = new MyClass();
        myClass2 = myClass;
    }
}
```

在上述的过程中，我们分别定义了值类型变量 myStruct 和引用类型变量 myClass，并使用 new 操作符完成内存分配和初始化操作，此处 new 的区别可以详见《第五回：深入浅出关键字---把 new 说透》 的论述，在此不做进一步描述。而我们在此强调的是 myStruct 和 myClass 两个变量在内存分配方面的区别，还是以一个简明的图来展示一下：



我们知道，每个变量或者程序都有其堆栈，不同的变量不能共有同一个堆栈地址，因此 myStruct 和 myStruct2 在堆栈中一定占用了不同的堆栈地址，尽管经过了变量的传递，实际的内存还是分配在不同的地址上，如果我们再对 myStruct2 变量改变时，显然不会影响到 myStruct 的数据。从图中我们还可以显而易见的看出，myStruct 在堆栈中包含其实例数据，而 myClass 在堆栈中只是保存了其实例数据的引用地址，实际的数据保存在托管堆中。因此，就有可能不同的变量保存了同一地址的数据引用，当数据从一个引用类型变量传递到另一个相同类型的引用类型变量时，传递的是其引用地址

而不是实际的数据，因此一个变量的改变会影响另一个变量的值。从上面的分析就可以明白的知道这样一个简单的道理：值类型和引用类型在内存中的分配区别是决定其应用不同的根本原因，由此我们就可以很容易的解释为什么参数传递时，按值传递不会改变形参值，而按址传递会改变行参的值，道理正在于此。

对于内存分配的更详细位置，可以描述如下：

- 值类型变量做为局部变量时，该实例将被创建在堆栈上；而如果值类型变量作为类型的成员变量时，它将作为类型实例数据的一部分，同该类型的其他字段都保存在托管堆上，这点我们将在接下来的嵌套结构部分来详细说明。
- 引用类型变量数据保存在托管堆上，但是根据实例的大小有所区别，如下：如果实例的大小小于 85000Byte 时，则该实例将创建在 GC 堆上；而当实例大小大于等于 85000byte 时，则该实例创建在 LOH（Large Object Heap）堆上。

更详细的分析，我推荐《类型实例的创建位置、托管对象在托管堆上的结构》。

2.2.2. 嵌套结构

嵌套结构就是在值类型中嵌套定义了引用类型，或者在引用类型变量中嵌套定义了值类型，相信园子中关于这一话题的论述和关注都不是很多。因此我们很有必要发挥一下，在此就顺藤摸瓜，从上文对.NET 的内存机制着手来理解会水到渠成。

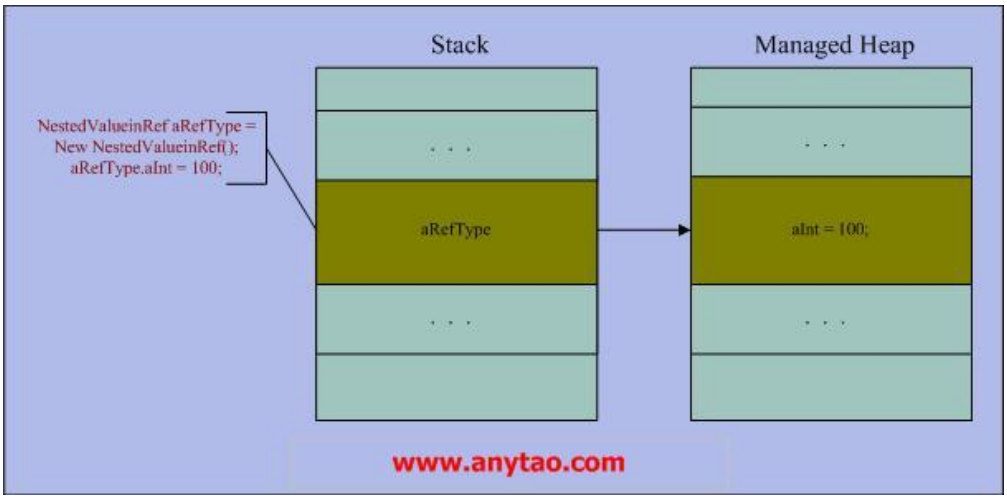
- 引用类型嵌套值类型

值类型如果嵌套在引用类型时，也就是值类型在内联的结构中时，其内存分配是什么样子呢？ 其实很简单，例如类的私有字段如果为值类型，那它作为引用类型实例的一部分，也分配在托管堆上。例如：

```
public class NestedValueinRef
{
 //aInt 做为引用类型的一部分将分配在托管堆上
 private int aInt;
 public NestedValueinRef
 {
  //aChar 则分配在该段代码的线程栈上
   char achar = 'a';
```

```
 }
}
```

其内存分配图可以表示为：

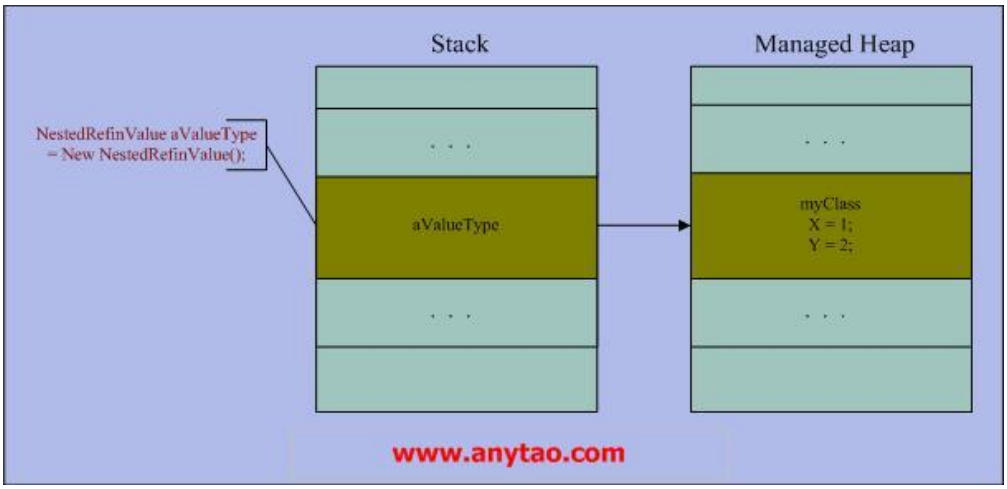


- 值类型嵌套引用类型

引用类型嵌套在值类型时，内存的分配情况为：该引用类型将作为值类型的成员变量，堆栈上将保存该成员的引用，而成员的实际数据还是保存在托管堆中。例如：

```
public struct NestedRefinValue
{
   public MyClass myClass;
   public NestedRefinValue
   {
      myClass.X = 1;
      myClass.Y = 2;
   }
}
```

其内存分配图可以表示为：



2.2.3. 一个简单的讨论

通过上面的分析，如果我们现在有如下的执行时：

```
AType[] myType = new AType[10];
```

试问：如果 AType 是值类型，则分配了多少内存；而如果 AType 是引用类型时，又分配了多少内存？

我们的分析如下：根据 CRL 的内存机制，我们知道如果 ATpye 为 Int32 类型，则表示其元素是值类型，而数组本身为引用类型，myType 将保存指向托管堆中的一块大小为 4×10byte 的内存地址，并且将所有的元素赋值为 0；而如果 AType 为自定义的引用类型，则会只做一次内存分配，在线程的堆栈创建了一个指向托管堆的引用，而所有的元素被设置为 null 值，表示为空。

未完，下回即将发布。。。

参考文献

（USA）Jeffrey Richter, Applied Microsoft .NET Framework Programming

（USA）David Chappell， Understanding .NET

广而告之

本文有些长，因此分两回来展开。我们已经分析了类型的内存机制，接下来就该着重于类型的实际应用领域了，因此在下回中我们会从[通用规则与区别]、[实例分析]、[应用场合]、[类型比较]等几个方面来着重展开，希望给大家以帮助，对于表达有谬或者理解有误的地方还望不吝赐教，本人将不胜感激。

To be continue soon ...

温故知新

品味类型---值类型与引用类型（中）－规则无边

接上回[第八回：品味类型---值类型与引用类型（上）－内存有理]的探讨，继续我们关注值类型和引用类型的话题。

本文将介绍以下内容：

- 类型的基本概念
- 值类型深入
- 引用类型深入
- 值类型与引用类型的比较及应用

1. 引言

上回[第八回：品味类型---值类型与引用类型（上）－内存有理]的发布，受到大家的不少关注，我们从内存的角度了解了值类型和引用类型的所以然，留下的任务当然是如何应用类型的不同特点在系统设计、性能优化等方面发挥其作用。因此，本回是对上回有力的补充，同时应朋友的希望，我们尽力从内存调试的角度来着眼一些设计的分析，这样就有助于对这一主题进行透彻和全面的理解，当然这也是下一回的重点。

从内存角度来讨论值类型和引用类型是有理有据的， 而从规则的角度来了解值类型和引用类型是无边无际的。本文旨在从上文呼应的角度，来把这个主题彻底的融会贯通，无边无迹的应用，还是来自反复无常的实践，因此对应用我只能说以一个角度来阐释观点，但是肯定不可能力求全局。因此，我们从以下几个角度来完成对值类型与引用类型应用领域的讨论。

2. 通用规则与比较

通用有规则：

- string 类型是个特殊的引用类型，它继承自 System.Object 肯定是个引用类型，但是在应用表现上又凸现出值类型的特点，那么究竟是什么原因呢？例如有如下的一段执行：

```
using System;

namespace Anytao.net.My_Must_net
{
    public class TestString
    {
        public static void Main(string[] args)
        {
            //定义一个string类型
            string myMsg = "Anytao is anytao.";
            //改变string值，并显示
            ChangeMsg(myMsg);
            //当前的myMsg是否发生了改变？
            ShowMsg(myMsg);
        }

        public static void ChangeMsg(string str)
        {
            //改变参数内容，实际是生成了新的string对象
            //执行初始化，但是函数外部无法取得变化
            str = "Anytao is not anytao.";
            Console.WriteLine(str);
        }

        public static void ShowMsg(string str)
        {
            Console.WriteLine(str);
        }
    }
}
©2007 Anytao.com
```

简单的说是由于 string 的 immutable 特性，因此每次对 string 的改变都会在托管堆中产生一个新的 string 变量，上述 string 作为参数传递时，实际上执行了 s=s 操作，在托管堆中会产生一个新的空间，并执行数据拷贝，所以才有了类似于按值传递的结果。但是根据我们的内存分析可知，string 在本质上还是一个引用类型，在参数传递时发生的还是按址传递，不过由于其特殊的恒定特性，在函数内部新建了一个 string 对象并完成初始化，但是函数外部取不到这个变化的结果，因此对外表现的特性就类似于按值传递。至于 string 类型的特殊性解释，我推荐 Artech 的大作《深入理解 string 和如何高效地使用 string》。

另外，string 类型重载了==操作符，在类型比较是比较的是实际的字符串，而不是引用地址，因此有以下的执行结果：

```
        string aString = "123";
        string bString = "123";
        Console.WriteLine((aString == bString)); //显示为 true，等价于 aString.Equals(bStrin
g)；
        string cString = bString;
        cString = "456";
        Console.WriteLine((bString == cString)); //显示为 false，等价于 bString.Equals(cStri
ng)；
```

- 通常可以使用 Type.IsValueType 来判断一个变量的类型是否为值类型，典型的操作为：

```
public struct MyStructTester
{ }

public class isValueType_Test
{
   public static void Main()
   {
      MyStructTester aStruct = new MyStructTester();
      Type type = aStruct.GetType();
      if (type.IsValueType)
      {
         Console.WriteLine("{0} belongs to value type.", aStruct.ToString());
      }

   }
}
```

- .NET 中以操作符 ref 和 out 来标识值类型按引用类型方式传递，其中区别是：ref 在参数传递之前必须初始化；而 out 则在传递前不必初始化，且在传递时必须显式赋值。
- 值类型与引用类型之间的转换过程称为装箱与拆箱，这值得我们以专门的篇幅来讨论，因此留待后文详细讨论这一主题。

- sizeof()运算符用于获取值类型的大小，但是不适用于引用类型。
- 值类型使用 new 操作符完成初始化，例如：MyStruct aTest = new MyStruct(); 而单纯的定义没有完成初始化动作，此时对成员的引用将不能通过编译，例如：

```
MyStruct aTest;
Console.WriteLine(aTest.X);
```

- 引用类型在性能上欠于值类型主要是因为以下几个方面：引用类型变量要分配于托管堆上；内存释放则由 GC 完成，造成一定的 CG 堆压力；同时必须完成对其附加成员的内存分配过程；以及对象访问问题。因此，.NET 系统不能由纯粹的引用类型来统治，性能和空间更加优越和易于管理的值类型有其一席之地，这样我们就不会因为一个简单的 byte 类型而进行复杂的内存分配和释放工作。Richter 就称值类型为“轻量级”类型，简直恰如其分，处理数据较小的情况时，应该优先考虑值类型。
- 值类型都继承自 System.ValueType，而 System.ValueType 又继承自 System.Object，其主要区别是 ValueType 重写了 Equals 方法，实现对值类型按照实例值比较而不是引用地址来比较，具体为：

```
char a = 'c';
char b = 'c';
Console.WriteLine((a.Equals(b))); //会返回 true;
```

- 基元类型，是指编译器直接支持的类型，其概念其实是针对具体编程语言而言的，例如 C#或者 VB.NET，通常对应用.NET Framework 定义的内置值类型。这是概念上的界限，不可混淆。例如：int 对应于 System.Int32，float 对应于 System.Single。

比较出真知：

- 值类型继承自 ValueType（注意：而 System.ValueType 又继承自 System.Object）；而引用类型继承自 System.Object。
- 值类型变量包含其实例数据，每个变量保存了其本身的数据拷贝（副本），因此在默认情况下，值类型的参数传递不会影响参数本身；而引用类型变量保存了其数据的引用地址，因此以引用方式进行参数传递时会影响到参数本身，因为两个变量会引用了内存中的同一块地址。
- 值类型有两种表示：装箱与拆箱；引用类型只有装箱一种形式。我会在下节以专门的篇幅来深入讨论这个话题。

- 典型的值类型为：struct，enum 以及大量的内置值类型；而能称为类的都可以说是引用类型。 struct 和 class 主要的区别可以参见我的拙作《第四回：后来居上：class 和 struct》来详细了解，也是对值类型和引用类型在应用方面的有力补充。

- 值类型的内存不由 GC（垃圾回收，Gabage Collection）控制，作用域结束时，值类型会自行释放，减少了托管堆的压力，因此具有性能上的优势。例如，通常 struct 比 class 更高效；而引用类型的内存回收，由 GC 来完成，微软甚至建议用户最好不要自行释放内存。

- 值类型是密封的（sealed），因此值类型不能作为其他任何类型的基类，但是可以单继承或者多继承接口；而引用类型一般都有继承性。

- 值类型不具有多态性；而引用类型有多态性。

- 值类型变量不可为 null 值，值类型都会自行初始化为 0 值；而引用类型变量默认情况下，创建为 null 值，表示没有指向任何托管堆的引用地址。对值为 null 的引用类型的任何操作，都会抛出 NullReferenceException 异常。

- 值类型有两种状态：装箱和未装箱，运行库提供了所有值类型的已装箱形式；而引用类型通常只有一种形式：装箱。

3. 对症下药－应用场合与注意事项

现在，在内存机制了解和通用规则熟悉的基础上，我们就可以很好的总结出值类型和引用类型在系统设计时，如何作出选择？当然我们的重点是告诉你，如何去选择使用值类型，因为引用类型才是.NET 的主体，不必花太多的关照就可以赢得市场。

3.1 值类型的应用场合

- MSDN 中建议以类型的大小作为选择值类型或者引用类型的决定性因素。数据较小的场合，最好考虑以值类型来实现可以改善系统性能；

- 结构简单，不必多态的情况下，值类型是较好的选择；

- 类型的性质不表现出行为时，不必以类来实现，那么用以存储数据为主要目的的情况下，值类型是优先的选择；

- 参数传递时，值类型默认情况下传递的是实例数据，而不是内存地址，因此数据传递情况下的选择，取决于函数内部的实现逻辑。值类型可以有高效的内存支持，并且在不暴露内部结构的情况下返回

实例数据的副本，从安全性上可以考虑值类型，但是过多的值传递也会损伤性能的优化，应适当选择；

- 值类型没有继承性，如果类型的选择没有子类继承的必要，优先考虑值类型；
- 在可能会引起装箱与拆箱操作的集合或者队列中，值类型不是很好的选择，因为会引起对值类型的装箱操作，导致额外内存的分配，例如在 Hashtable。关于这点我将在后续的主题中重点讨论。

3.2 引用类型的应用场合

- 可以简单的说，引用类型是.NET 世界的全值杀手，我们可以说.NET 世界就是由类构成的，类是面向对象的基本概念，也是程序框架的基本要素，因此灵活的数据封装特性使得引用类型成为主流；
- 引用类型适用于结构复杂，有继承、有多态，突出行为的场合；
- 参数传递情况也是考虑的必要因素；

4. 再论类型判等

类型的比较通常有 Equals()、ReferenceEquals()和==/!=三种常见的方法，其中核心的方法是 Equals。我们知道 Equals 是 System.Object 提供的虚方法，用于比较两个对象是否指向相同的引用地址，.NET Framework 的很多类型都实现了对 Equals 方法的重写，例如值类型的“始祖”System.ValueType 就重载了 Equal 方法，以实现对实例数据的判等。因此，类型的判等也要从重写或者重载 Equals 等不同的情况具体分析，对值类型和引用类型判等，这三个方法各有区别，应多加注意。

4.1 值类型判等

- Equals，System.ValueType 重载了 System.Object 的 Equals 方法，用于实现对实例数据的判等。
- ReferenceEquals，对值类型应用 ReferenceEquals 将永远返回 false。
- ==，未重载的==的值类型，将比较两个值是否“按位”相等。

4.2 引用类型判等

- Equals，主要有两种方法，如下

```
public virtual bool Equals(object obj);
public static bool Equals(object objA, object objB);
```

一种是虚方法，默认为引用地址比较；而静态方法，如果 objA 是与 objB 相同的实例，或者如果两者均为空引用，或者如果 objA.Equals(objB)返回 true，则为 true；否则为 false。.NET 的大部分类都重写了 Equals 方法，因此判等的返回值要根据具体的重写情况决定。

- ReferenceEquals，静态方法，只能用于引用类型，用于比较两个实例对象是否指向同一引用地址。

- ==，默认为引用地址比较，通常进行实现了==的重载，未重载==的引用类型将比较两个对象是否引用地址，等同于引用类型的 Equals 方法。因此，很多的.NET 类实现了对==操作符的重载，例如 System.String 的==操作符就是比较两个字符串是否相同。而==和 equals 方法的主要区别，在于多态表现上，==是被重载，而 Equals 是重写。

有必要在自定义的类型中，实现对 Equals 和==的重写或者重载，以提高性能和针对性分析。

5. 再论类型转换

类型转换是引起系统异常一个重要的因素之一，因此在有必要在这个主题里做以简单的总结，我们不力求照顾全面，但是追去提纲挈领。常见的类型转换包括：

- 隐式转换：由低级类型项高级类型的转换过程。主要包括：值类型的隐式转换，主要是数值类型等基本类型的隐式转换；引用类型的隐式转换，主要是派生类向基类的转换；值类型和引用类型的隐士转换，主要指装箱和拆箱转换。

- 显示转换：也叫强制类型转换。但是转换过程不能保证数据的完整性，可能引起一定的精度损失或者引起不可知的异常发生。转换的格式为，

```
(type)(变量、表达式)
```

例如：int a = (int)(b + 2.02);

- 值类型与引用类型的装箱与拆箱是.NET 中最重要的类型转换，不恰当的转换操作会引起性能的极大损耗，因此我们将以专门的主题来讨论。

- 以 is 和 as 操作符进行类型的安全转换，详见本人拙作《第一回：恩怨情仇：is 和 as》。

- System.Convert 类定义了完成基本类型转换的便捷实现。

- 除了 string 以外的其他类型都有 Parse 方法，用于将字符串类型转换为对应的基本类型；

- 使用 explicit 或者 implicit 进行用户自定义类型转换，主要给用户提高自定义的类型转换实现方式，以实现更有目的的转换操作，转换格式为，

```
static 访问修饰操作符 转换修饰操作符 operator 类型(参数列表);
```

例如：

```
public Student
{
   //...

   static public explicite opertator Student(string name, int age)
   {
      return new Student(name, age);
   }

   //...
}
```

其中，所有的转换都必须是 static 的。

6. 结论

现在，我们从几个角度延伸了上回对值类型和引用类型的分析，正如本文开头所言，对类型的把握还有很多可以挖掘的要点，但是以偏求全的办法我认为还是可取的，尤其是在技术探求的过程中，力求面面俱到的做法并不是好事。以上的几个角度，我认为是对值类型和引用类型把握的必经之路，否则在实际的系统开发中常常会在细小的地方栽跟头，摸不着头脑。

品味类型，我们以应用为要点撬开值类型和引用类型的规矩与方圆。

品味类型，我们将以示例为导航，开动一个层面的深入分析，下回《第十回：品味类型---值类型与引用类型（下）－应用征途》我们再见。

参考文献

（USA）Jeffrey Richter, Applied Microsoft .NET Framework Programming

（USA）David Chappell， Understanding .NET

品味类型---值类型与引用类型（下）－应用征途

本文将介绍以下内容：

- 类型的基本概念
- 值类型深入
- 引用类型深入
- 值类型与引用类型的比较及应用

[下载]：[类型示例代码]

1. 引言

值类型与引用类型的话题经过了两个回合（[第八回：品味类型---值类型与引用类型（上）－内存有理]和[第九回：品味类型---值类型与引用类型（中）－规则无边]）的讨论和切磋，我们就基本的理解层面来说已经差不多了，但是对这一部分的进一步把握和更深刻的理解还要继续和深化，因为我自己就在两篇发布之际，我就得到装配脑袋兄的不倦指导，之后又查阅了很多的资料发现类型在.NET或者说语言基础中何其重要的内涵和深度，因此关于这个话题的讨论还没有停止，以后我将继续分享自己的所得与所感。

不过作为一个阶段，本文将值类型和引用类型的讨论从应用示例角度来进一步做以延伸，可以看作是对前两回的补充性探讨。我们从类型定义、实例创建、参数传递、类型判等、垃圾回收等几个方面来简要的对上两回的内容做以剖析，并以一定的 IL 语言和内存机制来说明，期望进一步加深我们的理解和分析。

2. 以代码剖析

下面，我们以一个经典的值类型和引用类型对比的示例来剖析，其区别和实质。在剖析的过程中，我们主要以执行分析（主要是代码注释）、内存分析（主要是图例说明）和 IL 分析（主要是 IL 代码简析）三个方面来逐知识点解析，最后再做以总结描述，这样就可以有更深的理解。

2.1 类型定义

定义简单的值类型 MyStruct 和引用类型 MyClass，在后面的示例中将逐渐完善，完整的代码可以点击下载[类型示例代码]。我们的讨论现在开始，

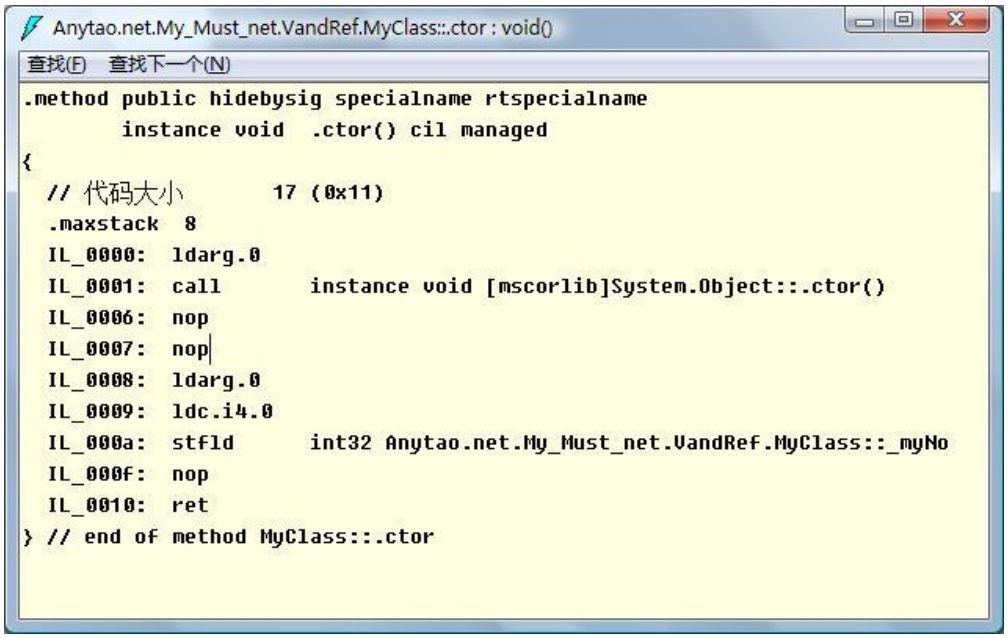
- 代码演示

```
// 01 定义值类型
public struct MyStruct
{
    private int _myNo;

    public int MyNo
    {
        get { return _myNo; }
        set { _myNo = value; }
    }

    public MyStruct(int myNo)
    {
        _myNo = myNo;
    }

    public void ShowNo()
    {
        Console.WriteLine(_myNo);
    }

}
```

```
// 02 定义引用类型
public class MyClass
{
    private int _myNo;

    public int MyNo
    {
        get { return _myNo; }
        set { _myNo = value; }
    }

    public MyClass()
```

```
    {
        _myNo = 0;
    }

    public MyClass(int myNo)
    {
        _myNo = myNo;
    }


    public void ShowNo()
    {
        Console.WriteLine(_myNo);
    }
}
```

- IL 分析



分析 IL 代码可知，静态方法.ctor 用来表示实现构造方法的定义，其中该段 IL 代码表示将 0 赋给字段_myNo。

2.2 创建实例、初始化及赋值

接下来，我们完成实例创建和初始化，和简单的赋值操作，然后在内存和 IL 分析中发现其实质。

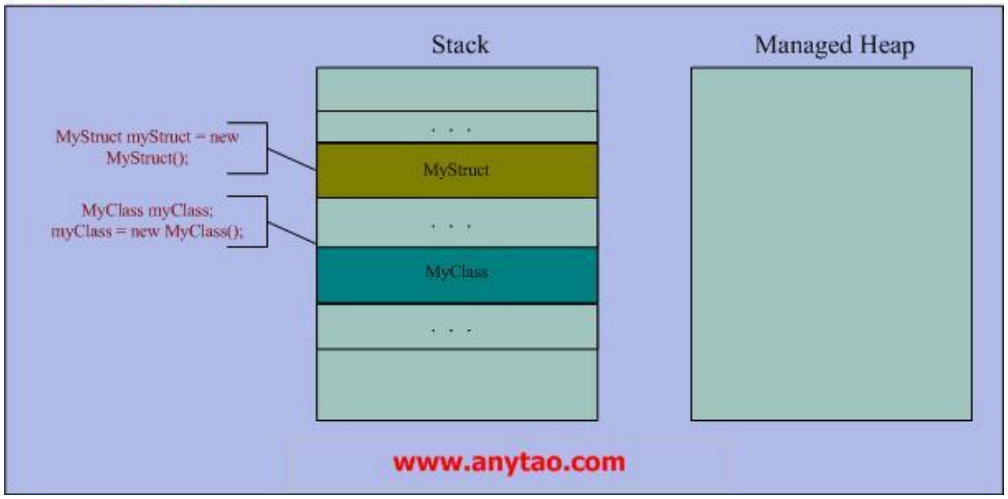
- 代码演示

```
创建实例、初始化及赋值
```

- 内存实况

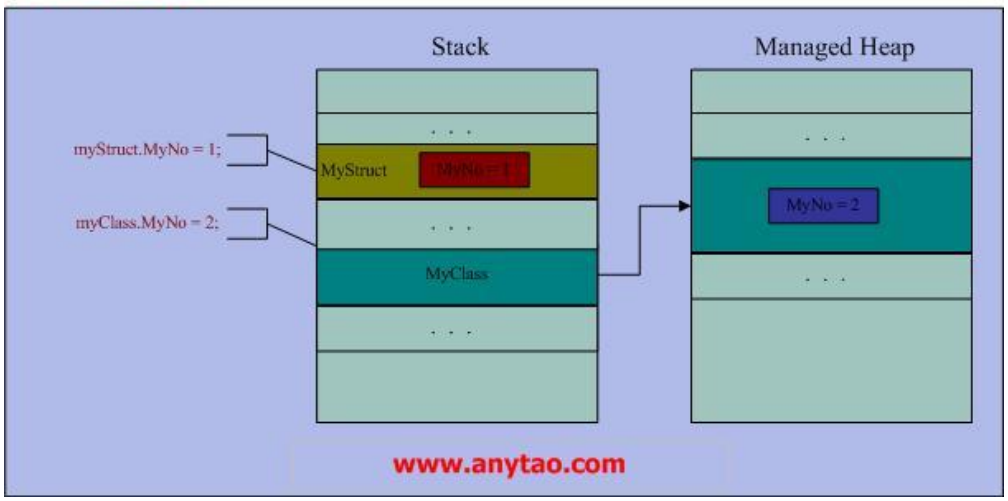
首先是值类型和引用类型的定义，这是一切面向对象的开始，

01 类型定义时



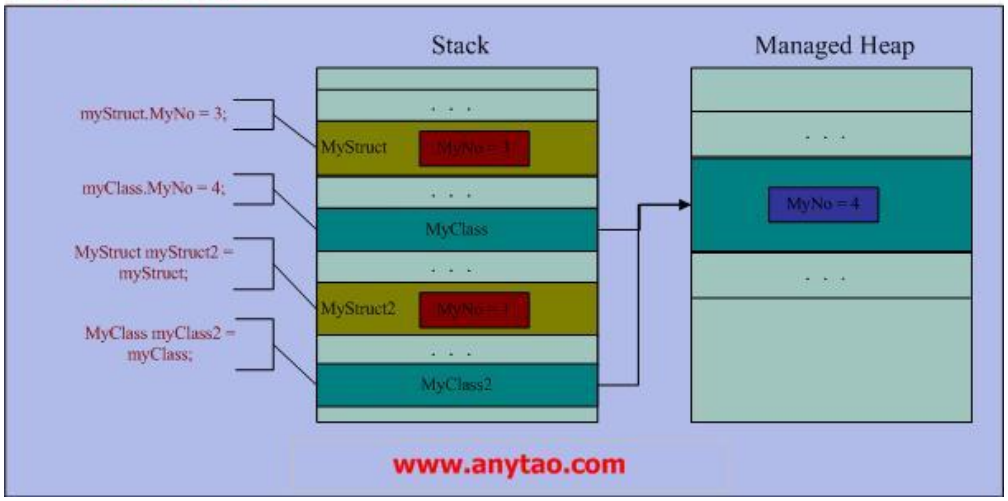
然后是初始化过程，

02 类型初始化时



简单的赋值和拷贝，是最基本的内存操作，不妨看看，

03 完成赋值与拷贝后



2.3 参数传递

- 代码演示

⊞ ⊟参数传递

不必多说，就是一个简要阐释，对于参数的传递作者将计划以更多的笔墨来在后面的系列中做以澄清和深入。

2.4 类型转换

类型转换的演示，包括很多个方面，在此我们只以自定义类型转换为例来做以说明，更详细的类型转换可以参考[第九回：品味类型---值类型与引用类型（中）－规则无边]的[再论类型转换部分]。

- 代码演示

首先是值类型的自定义类型转换，

```
public struct MyStruct
{
    // 01.2 自定义类型转：整形->MyStruct 型
    static public explicit operator MyStruct(int myNo)
```

```
    {
        return new MyStruct(myNo);
    }

}
```

然后是引用类型的自定义类型转换，

```
public class MyClass
{
    // 02.2 自定义类型转换：MyClass->string 型
    static public implicit operator string(MyClass mc)
    {
        return mc.ToString();
    }

    public override string ToString()
    {
        return _myNo.ToString();
    }
}
```

最后，我们对自定义的类型做以测试，

```
    public static void Main(string[] args)
    {
        #region 03. 类型转换
        MyStruct MyNum;
        int i = 100;
        MyNum = (MyStruct)i;
        Console.WriteLine("整形显式转换为 MyStruct 型---");
        Console.WriteLine(i);

        MyClass MyCls = new MyClass(200);
        string str = MyCls;
```

```
        Console.WriteLine("MyClass 型隐式转换为 string 型---");
        Console.WriteLine(str);
        #endregion
            }
```

2.5 类型判等

类型判等主要包括：ReferenceEquals()、Equals()虚方法和静态方法、==操作符等方面，同时注意在值类型和引用类型判等时的不同之处，可以参考[第九回：品味类型---值类型与引用类型（中）—规则无边]的[4. 再论类型判等]的简述。

- 代码演示

```
// 01 定义值类型
public struct MyStruct
{

// 01.1 值类型的类型判等
public override bool Equals(object obj)
{
return base.Equals(obj);
}

}

public class MyClass
{

// 02.1 引用类型的类型判等
public override bool Equals(object obj)
{
return base.Equals(obj);
}

}

public static void Main(string[] args)

{
```

```
#region 05 类型判等
Console.WriteLine("类型判等---");
// 05.1 ReferenceEquals 判等
//值类型总是返回 false，经过两次装箱的 myStruct 不可能指向同一地址
Console.WriteLine(ReferenceEquals(myStruct, myStruct));
//同一引用类型对象，将指向同样的内存地址
Console.WriteLine(ReferenceEquals(myClass, myClass));
//RefenceEquals 认为 null 等于 null，因此返回 true
Console.WriteLine(ReferenceEquals(null, null));

// 05.2 Equals 判等
//重载的值类型判等方法，成员大小不同
Console.WriteLine(myStruct.Equals(myStruct2)) ;

//重载的引用类型判等方法，指向引用相同
Console.WriteLine(myClass.Equals(myClass2));

#endregion

}
```

2.6 垃圾回收

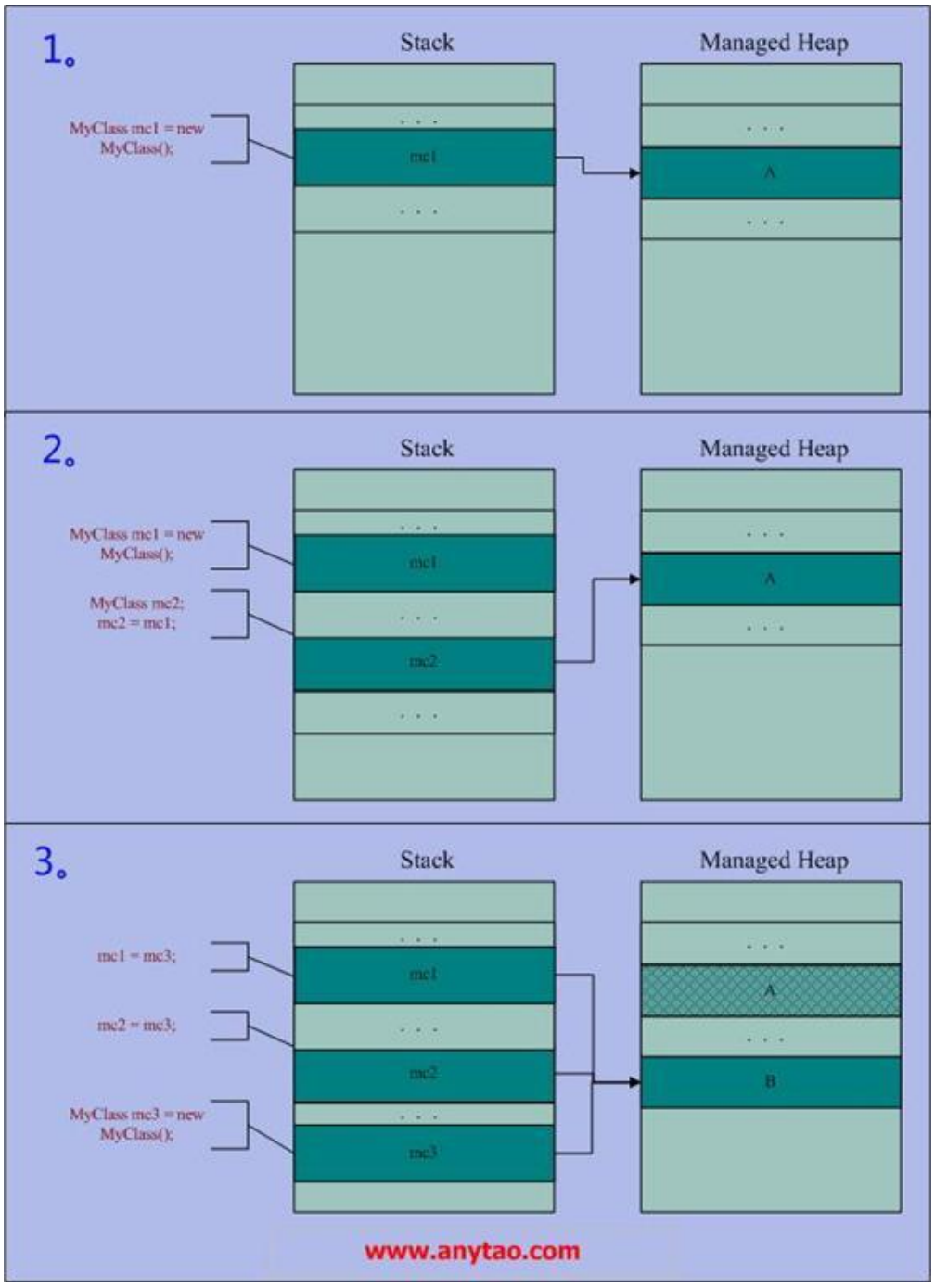
首先，垃圾回收机制，绝对不是三言两语就能交代清楚，分析明白的。因此，本示例只是从最简单的说明出发，对垃圾回收机制做以简单的分析，目的是有始有终的交代实例由创建到消亡的全过程。

- 代码演示

```
public static void Main(string[] args)
{

#region 06 垃圾回收的简单阐释
//实例定义及初始化
MyClass mc1 = new MyClass();
//声明但不实体化
MyClass mc2;
//拷贝引用，mc2 和 mc1 指向同一托管地址
mc2 = mc1;
```

```
//定义另一实例，并完成初始化
MyClass mc3 = new MyClass();
//引用拷贝，mc1、mc2 指向了新的托管地址
//那么原来的地址成为 GC 回收的对象，在
mc1 = mc3;
mc2 = mc3;
#endregion

}
```

- 内存实况



GC 执行时，会遍历所有的托管堆对象，按照一定的递归遍历算法找出所有的可达对象和不可访问对象，显然本示例中的托管堆 A 对象没有被任何引用访问，属于不可访问对象，将被列入执行垃圾收集的目标。对象由 newobj 指令产生，到被 GC 回收是一个复杂的过程，我们期望在系列的后期对此做以深入浅出的理解。

2.7 总结陈述

这些示例主要从从基础的方向入手来剖析前前两回中的探讨，不求能够全面而深邃，但求能够一点而及面的展开，技术的魅力正在于千变万化，技术追求者的力求却是从变化中寻求不变，不然我们实质太累了，我想这就是好方法，本系列希望的就是提供一个入口，打开一个方法。示例的详细分析可以下载[类型示例代码]，简单的分析希望能带来丝丝惬意。

3. 结论

值类型和引用类型，要说的，要做的，还有很多。此篇只是一个阶段，更多的深入和探讨我相信还在继续，同时广泛的关注技术力量的成长，是每个人应该进取的空间和道路。

品味类型，为应用之路开辟技术基础。

品味类型，继续探讨还会更多精彩。

4.3 参数之惑——参数之惑---传递的艺术（上）

本文将介绍以下内容：

- 按值传递与按引用传递深论
- ref 和 out 比较
- 参数应用浅析

1. 引言

接上回《第九回：品味类型---值类型与引用类型（中）－规则无边》中，对值类型和引用类型的讨论，其中关于 string 类型的参数传递示例和解释，引起园友的关注和讨论，可谓一石激起千层浪。受教于装配脑袋的深切指正，对这一概念有了相当进一步的了解，事实证明是我错了，在此向朋友们致歉，同时非常感谢大家的参与，尤其是装配脑袋的不倦相告。

因此，本文就以更为清晰的角度，把我理解有误的雷区作做以深入的讨论与分析，希望通过我的一点点努力和探讨至少对如下几个问题能有清晰的概念：

- 什么是按值传递？什么是按引用传递？
- 按引用传递和按引用类型参数传递的区别？
- ref 与 out 在按引用传递中的比较与应用如何？

- param 修饰符在参数传递中的作用是什么？

2. 参数基础论

简单的来说，参数实现了不同方法间的数据传递，也就是信息交换。Thinking in Java 的作者有过一句名言：一切皆为对象。在.NET 语言中也是如此，一切数据都最终抽象于类中封装，因此参数一般用于方法间的数据传递。例如典型的 Main 入口函数就有一个 string 数组参数，args 是函数命令行参数。通常参数按照调用方式可以分为：形参和实参。形参就是被调用方法的参数，而实参就是调用方法的参数。例如：

```
using System;

public class Arguments
{
  public static void Main(string [] args)
  {
    string myString = "This is your argument.";
    //myString 是实际参数
    ShowString(myString);
  }

  private void ShowString(string astr)
  {
    Console.WriteLine(astr);
  }
}
```

由上例可以得出以下几个关于参数的基本语法：

- 形参和实参必须类型、个数与顺序对应匹配；
- 参数可以为空；
- 解析 Main(string [] args)，Main 函数的参数可以为空，也可以为 string 数组类，其作用是接受命令行参数，例如在命令行下运行程序时，args 提供了输入命令行参数的入口。
- 另外，值得一提的是，虽然 CLR 支持参数默认值，但是 C#中却不能设置参数默认值，这一点让我很郁闷，不知为何？不过可以通过重载来变相实现，具体如下：

```
static void JudgeKind(string name, string kind)
{
   Console.WriteLine("{0} is a {1}", name, kind);
}

static void JudgeKind(string name)
{
   //伪代码
   if(name is person)
   {
      Console.WriteLine(name, "People");
   }
}
```

这种方法可以扩展，可以实现更多个默认参数实现，不过，说实话有些多此一举，不够灵活，不爽不爽。

3. 传递的基础

接下来，我们接上面的示例讨论，重点将参数传递的基础做以交代，以便对参数之惑有一个从简入繁的演化过程。我们以基本概念的形式来一一列出这些基本概念，先混个脸儿熟，关于形参、实参、参数默认值的概念就不多做交代，参数传递是本文的核心内容，将在后文以大量的笔墨来阐述。所以接下来的概念，我们就做以简单的引入不花大量的精力来讨论，主要包括：

3.1 泛型类型参数

泛型类型参数，可以是静态的，例如 MyGeneric<int>；也可以是动态的，此时它其实就是一个占位符，例如 MyGeneric<T>中的 T 可以是任何类型的变量，在运行期动态替换为相应的类型参数。泛型类型参数一般也以 T 开头来命名。

3.2 可变数目参数

一般来说参数个数都是固定的，定义为集群类型的参数可以实现可变数目参数的目的，但是.NET 提供了更灵活的机制来实现可变数目参数，这就是使用 param 修饰符。可变数目参数的好处就是在某些情况下可以方便的提供对于参数个数不确定情况的实现，例如计算任意数字的加权和，连接任意字符串为一个字符串等。我们以一个简单的示例来展开对这个问题的论述，为：

```
// FileName    : Anytao.net.My_Must_net
// Description : The .NET what you should know of arguments.
// Release     : 2007/07/05 1.0

// Copyright   : (C)2007 Anytao.com  http://www.anytao.com
using System;

namespace Anytao.net.My_Must_net
{
    class ParamArrayAttributeTest
    {
        static void Main()
        {
            //编译器找到相应的方法就先调用不包含
            //param特性的方法
            ShowAgeSum("Anycall", 1, 2, 3);
            //编译器找不到五个证型参数的方法，故调用包含
            //param特性的方法，同时构造这5个整数为
            //一维整形数组
            ShowAgeSum("Anytao", 1, 2, 3, 4, 5);
        }

        static void ShowAgeSum(string team, int i, int j, int k)
        {
            Console.WriteLine("The No Param team {0}'s age sum is {1}", team, i + j + k);
        }

        static void ShowAgeSum(string team, params int[] ages)
        {
            int ageSum = 0;
            for (int i = 0; i < ages.Length; i++)
                ageSum += ages[i];
            Console.WriteLine("The Param team {0}'s age sum is {1}", team, ageSum);
        }
    }
}
```

在此基础上，我们将使用 param 关键字实现可变数目参数的规则和使用做以小结为：

- param 关键字的实质是：param 是定制特性 ParamArrayAttribute 的缩写（关于定制特性的详细论述请参见第三回：历史纠葛：特性和属性），该特性用于指示编译器的执行过程大概可以简化为：编译器检查到方法调用时，首先调用不包含 ParamArrayAttribute 特性的方法，如果存在这种方法就施行调用，如果不存在才调用包含 ParamArrayAttribute 特性的方法，同时应用方法中的元素来填充一个数组，同时将该数组作为参数传入调用的方法体。总之就是 param 就是提示编译器实现对参数进行数组封装，将可变数目的控制由编译器来完成，我们可以很方便的从上述示例中得到启示。例如：

```
static void ShowAgeSum(string team, params int[] ages){...}
```

实质上是这样子：

```
static void ShowAgeSum(string team, [ParamArrayAttribute] int[] ages){...}
```

- param 修饰的参数必须为一维数组，事实上通常就是以群集方式来实现多个或者任意多个参数的控制的，所以数组是最简单的选择；
- param 修饰的参数数组，可是是任何类型。因此，如果需要接受任何类型的参数时，只要设置数组类型为 object 即可；
- param 必须在参数列表的最后一个，并且只能使用一次。

4. 深入讨论，传递的艺术

默认情况下，CRL 中的方法都是按值传递的，但是在具体情况会根据传递的参数情况的不同而有不同的表现，我们在深入讨论传递艺术的要求下，就是将不同的传递情况和不同的表现情况做以小结，从中剥离出参数传递复杂表现之内的实质所在。从而为开篇的几个问题给出清晰的答案。

4.1 值类型参数的按值传递

首先，参数传递根据参数类型分为按值传递和按引用传递，默认情况下都是按值传递的。按值传递主要包括值类型参数的按值传递和引用类型参数的按值传递。值类型实例传递的是该值类型实例的一个拷贝，因此被调用方法操作的是属于自己本身的实例拷贝，因此不影响原来调用方法中的实例值。以例为证：

```
// FileName    : Anytao.net.My_Must_net
// Description : The .NET what you should know of arguments.
// Release     : 2007/07/01 1.0

// Copyright   : (C)2007 Anytao.com  http://www.anytao.com

using System;

namespace Anytao.net.My_Must_net
{
   class Args
   {
```

```
        public static void Main()
        {
            int a = 10;
            Add(a);
            Console.WriteLine(a);
        }

        private static void Add(int i)
        {
            i = i + 10;
            Console.WriteLine(i);
        }
    }
}
```

参数之惑---传递的艺术（下）

本文将介绍以下内容：

- 按值传递与按引用传递深论
- ref 和 out 比较
- 参数应用浅析

接上篇继续，『第十一回：参数之惑---传递的艺术（上）』

4.2 引用类型参数的按值传递

当传递的参数为引用类型时，传递和操作的是指向对象的引用，这意味着方法操作可以改变原来的对象，但是值得思考的是该引用或者说指针本身还是按值传递的。因此，我们在此必须清楚的了解以下两个最根本的问题：

- 引用类型参数的按值传递和按引用传递的区别？

- string 类型作为特殊的引用类型，在按值传递时表现的特殊性又如何解释？

首先，我们从基本的理解入手来了解引用类型参数按值传递的本质所在，简单的说对象作为参数传递时，执行的是对对象地址的拷贝，操作的是该拷贝地址。这在本质上和值类型参数按值传递是相同的，都是按值传递。不同的是值类型的“值”为类型实例，而引用类型的“值”为引用地址。因此，如果参数为引用类型时，在调用方代码中，可以改变引用的指向， 从而使得原对象的指向发生改变，如例所示：

```
⊞⊟引用类型参数的按值传递
// FileName    : Anytao.net.My_Must_net
// Description : The .NET what you should know of arguments.
// Release     : 2007/07/01 1.0

// Copyright   : (C)2007 Anytao.com  http://www.anytao.com

using System;

namespace Anytao.net.My_Must_net
{
   class Args
   {
      public static void Main()
      {
         ArgsByRef abf = new ArgsByRef();
         AddRef(abf);
         Console.WriteLine(abf.i);
      }

      private static void AddRef(ArgsByRef abf)
      {
         abf.i = 20;
         Console.WriteLine(abf.i);
      }

   }
```

```
    class ArgsByRef
    {
        public int i = 10;
    }
}
```

因此，我们进一步可以总结为：按值传递的实质的是传递值，不同的是这个值在值类型和引用类型的表现是不同的：参数为值类型时，“值”为实例本身，因此传递的是实例拷贝，不会对原来的实例产生影响；参数为引用类型时，“值”为对象引用，因此传递的是引用地址拷贝，会改变原来对象的引用指向，这是二者在统一概念上的表现区别，理解了本质也就抓住了根源。关于值类型和引用类型的概念可以参考《第八回：品味类型---值类型与引用类型（上）－内存有理》《第九回：品味类型---值类型与引用类型（中）－规则无边》《第十回：品味类型---值类型与引用类型（下）－应用征途》，相信可以通过对系列中的值类型与引用类型的 3 篇的理解，加深对参数传递之惑的昭雪。

了解了引用类型参数按值传递的实质，我们有必要再引入另一个参数传递的概念，那就是：按引用传递，通常称为引用参数。这二者的本质区别可以小结为：

- 引用类型参数的按值传递，传递的是参数本身的值，也就是上面提到的对象的引用；
- 按引用传递，传递的不是参数本身的值，而是参数的地址。如果参数为值类型，则传递的是该值类型的地址；如果参数为引用类型，则传递的是对象引用的地址。

关于引用参数的详细概念，我们马上就展开来讨论，不过还是先分析一下 string 类型的特殊性，究竟特殊在哪里？

关于 string 的讨论，在本人拙作《第九回：品味类型---值类型与引用类型（中）－规则无边》已经有了讨论，也就是开篇陈述的本文成文的历史，所以在上述分析的基础上，我认为应该更能对第九回的问题，做以更正。

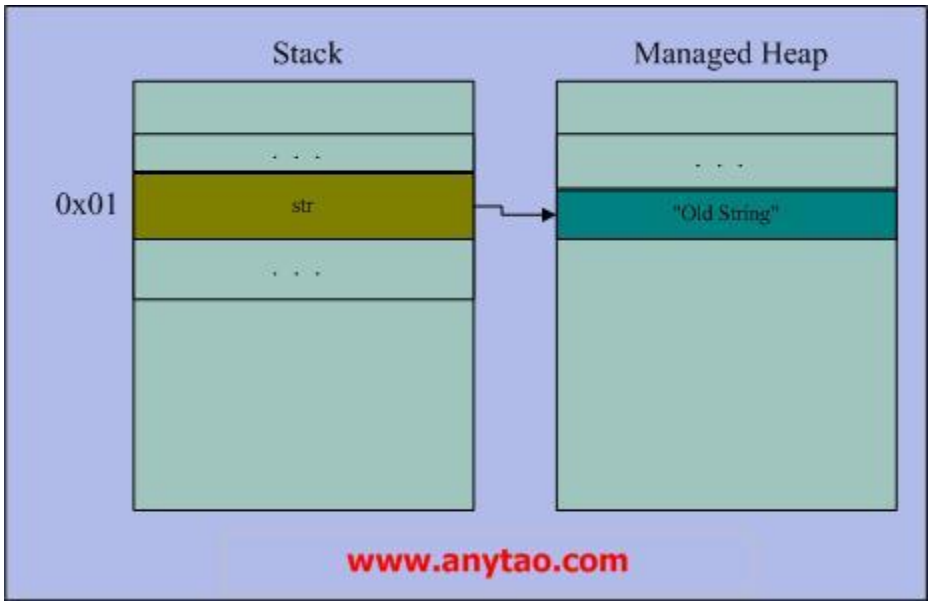
string 本身为引用类型，因此从本文的分析中可知，对于形如

static void ShowInfo(string aStr){...}

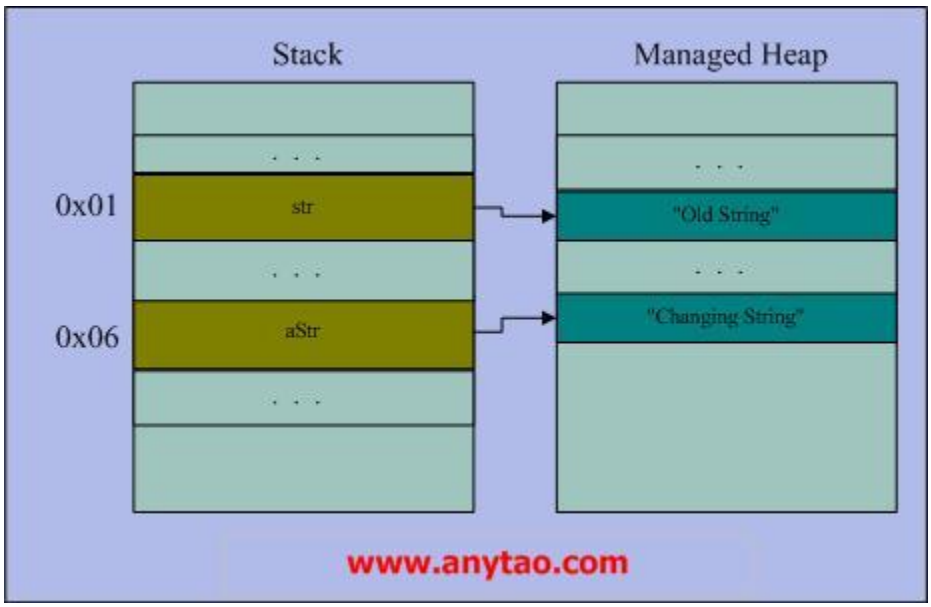
的传递形式，可以清楚的知道这是按值传递，也就是本文总结的引用类型参数的按值传递。因此，传递的是 aStr 对象的值，也就是 aStr 引用指针。接下来我们看看下面的示例来分析，为什么 string 类型在传递时表现出特殊性及其产生的原因？

```
// FileName    : Anytao.net.My_Must_net
// Description : The .NET what you should know of arguments.
// Release     : 2007/07/05 1.0

// Copyright   : (C)2007 Anytao.com  http://www.anytao.com
using System;

namespace Anytao.net.My_Must_net
{
   class how2str
   {
      static void Main()
      {
         string str = "Old String";
         ChangeStr(str);
         Console.WriteLine(str);
      }

      static void ChangeStr(string aStr)
      {
         aStr = "Changing String";
         Console.WriteLine(aStr);
      }
   }
}
```

下面对上述示例的执行过程简要分析一下：首先，string str = "Old String"产生了一个新的 string 对象，如图表示：



然后执行 ChangeStr(aStr)，也就是进行引用类型参数的按值传递，我们强调说这里传递的是引用类型的引用值，也就是地址指针；然后调用 ChangeStr 方法，过程 aStr = "Changing String"完成了以下的操作，先在新的一个地址生成一个 string 对象，该新对象的值为"Changing String"，引用地址为 0x06 赋给参数 aStr，因此会改变 aStr 的指向，但是并没有改变原来方法外 str 的引用地址，执行过程可以表示为：



因此执行结果就可想而知，我们从分析过程就可以发现 string 作为引用类型，在按值传递过程中和其他引用类型是一样的。如果需要完成 ChangeStr()调用后，改变原来 str 的值，就必须使用 ref 或者 out 修饰符，按照按引用传递的方式来进行就可以了，届时 aStr = "Changing String"改变的是 str

的引用，也就改变了 str 的指向，具体的分析希望大家通过接下来的按引用传递的揭密之后，可以自行分析。

4.3 按引用传递之 ref 和 out

不管是值类型还是引用类型，按引用传递必须以 ref 或者 out 关键字来修饰，其规则是：

- 方法定义和方法调用必须同时显示的使用 ref 或者 out，否则将导致编译错误；
- CRL 允许通过 out 或者 ref 参数来重载方法，例如：

```
// FileName    : Anytao.net.My_Must_net
// Description : The .NET what you should know of arguments.
// Release     : 2007/07/03 1.0

// Copyright   : (C)2007 Anytao.com  http://www.anytao.com

using System;
namespace Anytao.net.My_Must_net._11_Args
{
   class TestRefAndOut
   {
      static void ShowInfo(string str)
      {
         Console.WriteLine(str);
      }
      static void ShowInfo(ref string str)
      {
         Console.WriteLine(str);
      }
   }
}
```

当然，按引用传递时，不管参数是值类型还是引用类型，在本质上也是相同的，这就是：ref 和 out 关键字将告诉编译器，方法传递的是参数地址，而不是参数本身。理解了这一点也就抓住了按引用传递的本质，因此根据这一本质结论我们可以得出以下更明白的说法，这就是：

- 不管参数本身是值类型还是引用类型，按引用传递时，传递的是参数的地址，也就是实例的指针。

- 如果参数是值类型，则按引用传递时，传递的是值类型变量的引用，因此在效果上类似于引用类型参数的按值传递方式，其实质可以分析为：值类型的按引用传递方式，实现的是对值类型参数实例的直接操作，方法调用方为该实例分配内存，而被调用方法操作该内存，也就是值类型的地址；而引用类型参数的按值传递方式，实现的是对引用类型的“值”引用指针的操作。例如：

```
// FileName    : Anytao.net.My_Must_net
// Description : The .NET what you should know of arguments.
// Release     : 2007/07/06 1.0

// Copyright   : (C)2007 Anytao.com  http://www.anytao.com
using System;

namespace Anytao.net.My_Must_net
{
   class TestArgs
   {
      static void Main(string[] args)
      {
         int i = 100;
         string str = "One";
         ChangeByValue(ref i);
         ChangeByRef(ref str);
         Console.WriteLine(i);
         Console.WriteLine(str);
      }

      static void ChangeByValue(ref int iVlaue)
      {
         iVlaue = 200;
      }

      static void ChangeByRef(ref string sValue)
      {
         sValue = "One more.";
      }
```

```
    }
}
```

如果参数是引用类型，则按引用传递时，传递的是引用的引用而不是引用本身，类似于指针的指针概念。示例只需将上述 string 传递示例中的 ChangeStr 加上 ref 修饰即可。

下面我们再进一步对 ref 和 out 的区别做以交代，就基本阐述清楚了按引用传递的精要所在，可以总结为：

- 相同点：从 CRL 角度来说，ref 和 out 都是指示编译器传递实例指针，在表现行为上是相同的。最能证明的示例是，CRL 允许通过 ref 和 out 来实现方法重载，但是又不允许通过区分 ref 和 out 来实现方法重载，因此从编译角度来看，不管是 ref 还是 out，编译之后的代码是完全相同的。例如：

```
// FileName    : Anytao.net.My_Must_net
// Description : The .NET what you should know of arguments.
// Release     : 2007/07/03 1.0

// Copyright   : (C)2007 Anytao.com  http://www.anytao.com

using System;
namespace Anytao.net.My_Must_net._11_Args
{
    class TestRefAndOut
    {
        static void ShowInfo(string str)
        {
            Console.WriteLine(str);
        }
        static void ShowInfo(ref string str)
        {
            Console.WriteLine(str);
        }
        static void ShowInfo(out string str)
        {
            str = "Hello, anytao.";
            Console.WriteLine(str);
```

```
    }
  }
}
```

编译器将提示：“ShowInfo”不能定义仅在 ref 和 out 上有差别的重载方法。

- 不同点：使用的机制不同。ref 要求传递之前的参数必须首先显示初始化，而 out 不需要。也就是说，使用 ref 的参数必须是一个实际的对象，而不能指向 null；而使用 out 的参数可以接受指向 null 的对象，然后在调用方法内部必须完成对象的实体化。

5. 结论

完成了对值类型与引用类型的论述，在这些知识积累的基础上，本文期望通过深入的论述来进一步的分享参数传递的艺术，解开层层疑惑的面纱。从探讨问题的角度来说，参数传递的种种误区其实根植与对值类型和引用类型的本质理解上，因此完成了对类型问题的探讨再进入参数传递的迷宫，我们才会更加游刃有余。我想，这种探讨问题的方式，也正是我们追逐问题的方式，深入进入.NET 的高级殿堂是绕不开这一选择的。

参考文献

（USA）Jeffrey Richter, Applied Microsoft .NET Framework Programming

（USA）David Chappell， Understanding .NET

第 5 章 内存天下

5.1 内存管理概要

5.1.1 引言

提及内存管理，始终是 C++程序员最为头疼的问题，而这一切在.NET 托管平台下将变得容易，对象的创建、生存期管理及资源回收都由 CLR 负责，大大解放了开发者的精力，可以将更多的脑细胞投入到业务逻辑的实现上。

那么，使得这一切如此轻松的技术，又来自哪里？答案是.NET 自动内存管理（Automatic Memory Management）。CLR 引入垃圾收集器（GC，Garbage Collection）来负责执行

内存的清理工作，GC 通过对托管堆的管理，能有效解决 C++程序中类似于内存泄漏、访问不可达对象等问题。然而，必须明确的是垃圾回收并不能解决所有资源的清理，对于非托管资源，例如：数据库链接、文件句柄、COM 对象等，仍然需要开发者自行清理，.NET 又是如何处理呢？

总结起来，.NET 的自动内存管理，主要包括以下几个方面：

- 对象创建时的内存分配。
- 垃圾回收。
- 非托管资源释放。

本节，首先对这几个方面作以简单的介绍，而详细的论述在本章的其他部分逐一展开。

5.1.2 内存管理概观要论

本书在 1.1 节“对象的旅行”一节，从宏观的角度对对象生命周期做了一番调侃，而宏观之外对象的整个周期又是如何呢？下面，首先从一个典型的示例开始，以内存管理的角度对对象的生命周期做以梳理：

```
class MemoryProcess
{
   public static void Main()
   {
      //创建对象，分配内存，并初始化
      FileStream fs = new FileStream(@"C:\temp.txt", FileMode.Create);
      try
      {
         //对象成员的操作和应用
         byte[] txts = new UTF8Encoding(true).GetBytes("Hello, world.");
         fs.Write(txts, 0, txts.Length);
      }
```

```
        finally
        {
            //执行资源清理
            if (fs != null) fs.Close();
        }
    }
}
```

上述示例完成了一个简单的文件写入操作，我们要关注的是 FileStream 类型对象从创建到消亡的整个过程，针对上述示例总结起来各个阶段主要包括：

- 对象的创建及内存分配。

通过 new 关键字执行对象创建并分配内存，对应于 IL 中的 newobj 指令，除了这种创建方式，.NET 还提供了其他的对象创建方式与内存分配，在本章 5.2 节“对象创建始末”中，将对.NET 的内存分配及管理作以详细的讨论与分析。

- 对象初始化。

通过调用构造函数，完成对象成员的初始化，在本例 FileStream 对象的初始化过程中，必然发生对文件句柄的初始化操作，以便执行读写文件等应用。.NET 提供了 15 个不同的 FileStream 构造函数来完成对不同情况下的初始化处理，详细的分析见本章 5.2 节“对象创建始末”。

- 对象的应用和操作。

完成了内存分配和资源的初始化操作，就可以使用这些资源进行一定的操作和应用，例如本例中 fs.Write 通过调用文件句柄进行文件写入操作。

- 资源清理。

应用完成后，必须对对象访问的资源进行清理，本例中通过 Close 方法来释放文件句柄，关于非托管资源的释放及其清理方式，详见描述可参见 5.3 节“垃圾回收”。

- 垃圾回收。

在.NET 中，内存资源的释放由 GC 负责，这是.NET 技术中最闪亮的技术之一。CLR 完全代替开发人员管理内存，从分配到回收都有相应的机制来完成，原来熟悉的 free 和 delete 命令早已不复存在，在本章 5.3 节“垃圾回收”中，将对垃圾回收机制作以详细的讨论与分析。

5.1.3 结论

虽然，CLR 已经不需要开发者做太多的事情了，但是适度的探索可以帮助我们实现更好的驾驭，避免很多不必要的错误。本章的重点正是关于内存管理，对象创建、垃圾回收及性能优化等.NET 核心问题的探讨。本节可以看作一个起点，在接下来的各篇中我们将逐一领略.NET 自动内存管理的各个方面。

5.2 对象创建始末

5.2.1 引言

了解.NET 的内存管理机制，首先应该从内存分配开始，也就是对象的创建环节。对象的创建，是个复杂的过程，主要包括内存分配和初始化两个环节。在本章开篇的示例中，对象的创建过程为：

```
FileStream fs = new FileStream(@"C:\temp.txt", FileMode.Create);
```

通过 new 关键字操作，即完成了对 FileStream 类型对象的创建过程，这一看似简单的操作背后，却经历着相当复杂的过程和波折。

本篇全文，正是对这一操作背后过程的详细讨论，从中了解.NET 的内存分配是如何实现的。

5.2.2 内存分配

关于内存的分配，首先应该了解分配在哪里的问题。CLR 管理内存的区域，主要有三块，分别为：

- 线程的堆栈，用于分配值类型实例。堆栈主要由操作系统管理，而不受垃圾收集器的控制，当值类型实例所在方法结束时，其存储单位自动释放。栈的执行效率高，但存储容量有限。
- GC 堆，用于分配小对象实例。如果引用类型对象的实例大小小于 85000 字节，实例将被分配在 GC 堆上，当有内存分配或者回收时，垃圾收集器可能会对 GC 堆进行压缩，详见后文讲述。
- LOH（Large Object Heap）堆，用于分配大对象实例。如果引用类型对象的实例大小不小于 85000 字节时，该实例将被分配到 LOH 堆上，而 LOH 堆不会被压缩，而且只在完全 GC 回收时被回收。这种设计方案是对垃圾回收性能的优化考虑。

本节讨论的重点是.NET 的内存分配机制，因此下文将不加说明的以 GC 堆上的分配为例来展开。关于值类型和引用类型的论述，请参见本书 4.2 节“品味类型——值类型与引用类型”。

了解了内存分配的区域，接着我们看看有哪些操作将导致对象创建和内存分配的发生，在本书 3.4 节“经典指令解析之实例创建”一节中，详细描述了关于实例创建的多个 IL 指令解析，主要包括：

- newobj，用于创建引用类型对象。
- ldstr，用于创建 string 类型对象。
- newarr，用于分配新的数组对象。
- box，在值类型转换为引用类型对象时，将值类型字段拷贝到托管堆上发生的内存分配。

在上述论述的基础上，我们将从堆栈的内存分配和托管堆的内存分配两个方面来分别论述.NET 的内存分配机制。

1. 堆栈的内存分配机制

对于值类型来说，一般创建在线程的堆栈上。但并非所有的值类型都创建在线程的堆栈上，例如作为类的字段时，值类型作为实例成员的一部分也被创建在托管堆上；装箱发生时，值类型字段也会拷贝在托管堆上。

对于分配在堆栈上的局部变量来说，操作系统维护着一个堆栈指针来指向下一个自由空间的地址，并且堆栈的内存地址是由高位到低位向下填充，也就表示入栈时栈顶向低地址扩展，出栈时，栈顶向高地址回退。以下例而言：

```
public void MyCall()
{
```

```
    int x = 100;
    char c = 'A';
}
```

当程序执行至 MyCall 方法时，假设此时线程栈的初始地址为 50000，因此堆栈指针开始指向 50000 地址空间。方法调用时，首先入栈的是返回地址，也就是方法执行之后的下一条可执行语句的地址，用于方法返回之后程序继续执行，如图 5-1 所示。

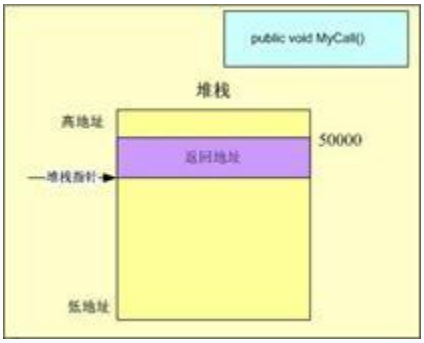


图 5-1 栈上的内存分配

然后是整型局部变量 x，它将在栈上分配 4Byte 的内存空间，因此堆栈指针继续向下移动 4 个字节，并将值 100 保存在相应的地址空间，同时堆栈指针指向下一个自由空间，如图 5-2 所示。

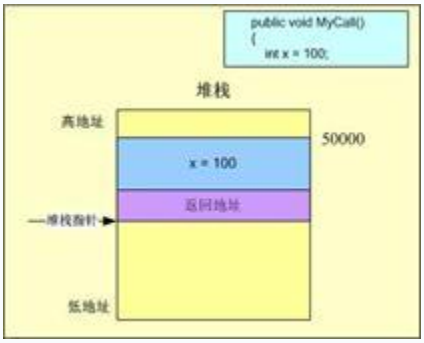


图 5-2 栈上的内存分配

接着是字符型变量 c，在堆栈上分配 2Byte 的内存空间，因此堆栈指针向下移动 2 个字节，值‘A’会保存在新分配的栈上空间，内存的分配如图 5-3 所示。

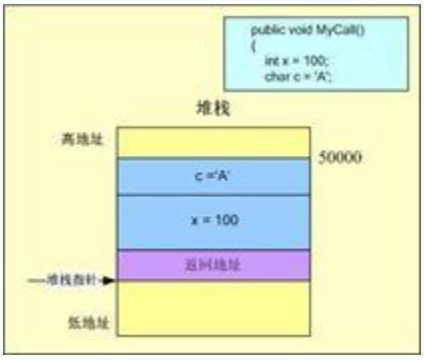


图 5-3 栈上的内存分配

最后，MyCall 方法开始执行，直到方法体执行结束，执行结果被返回，栈上的存储单元也被自行释放。其释放过程和分配过程刚好相反：首先删除 c 的内存，堆栈指针向上递增 2 个字节，然后删除 x 的内存，堆栈指针继续向上递增 4 个字节，最终的内存状况如图 5-4 所示，程序又将回到栈上最初的方法调用地址，继续向下执行。

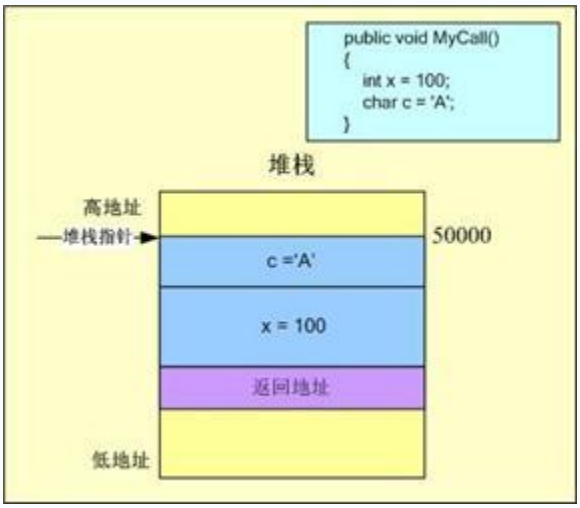


图 5-4 栈上的内存分配

其实，实际的分配情况是个非常复杂的分配过程，同时还包括方法参数，堆引用等多种情形的发生，但是本例演示的简单过程基本阐释了栈上分配的操作方式和过程。通过内置于处理器的特殊指令，栈上的内存分配，效率较高，但是内存容量不大，同时栈上变量的生存周期由系统自行管理。

注意

上述执行过程，只是一个简单的模拟情况，实际上在方法调用时都会在栈中创建一个活动记录（包含参数、返回值地址和局部变量），并分配相应的内存空间，这种分配是一次性完成的。方法执行结束返回时，活动记录清空，内存被一次性解除。而数据的压栈和出栈是有顺序的，栈内是先进先出（FILO）的形式。具体而言：首先入栈的是返回地址；然后是参数，一般以由右向左的顺序入栈；最后是局部变量，依次入栈。方法执行之后，出栈的顺序正好相反，首先是局部变量，再是参数，最后是那个地址指针。

2. 托管堆的内存分配机制

引用类型的实例分配于托管堆上，而线程栈却是对象生命周期开始的地方。对 32 位处理器来说，应用程序完成进程初始化后，CLR 将在进程的可用地址空间上分配一块保留的地址空间，它是进程（每个进程可使用 4GB）中可用地址空间上的一块内存区域，但并不对应于任何物理内存，这块地址空间即是托管堆。

托管堆又根据存储信息的不同划分为多个区域，其中最重要的是垃圾回收堆（GC Heap）和加载堆（Loader Heap），GC Heap 用于存储对象实例，受 GC 管理；Loader Heap 用于存储类型系统，又分为 High-Frequency Heap、Low-Frequency Heap 和 Stub Heap，不同的堆上存储不同的信息。Loader Heap 最重要的信息就是元数据相关的信息，也就是 Type 对象，每个 Type 在 Loader Heap 上体现为一个 Method Table（方法表），而 Method Table 中则记录了存储的元数据信息，例如基类型、静态字段、实现的接口、所有的方法等等。Loader Heap 不受 GC 控制，其生命周期为从创建到 AppDomain 卸载。

在进入实际的内存分配分析之前，有必要对几个基本概念做个交代，以便更好地在接下来的分析中展开讨论。

TypeHandle，类型句柄，指向对应实例的方法表，每个对象创建时都包含该附加成员，并且占用 4 个字节的内存空间。我们知道，每个类型都对应于一个方法表，方法表创建于编译时，主要包含了类型的特征信息、实现的接口数目、方法表的 slot 数目等。

SyncBlockIndex，用于线程同步，每个对象创建时也包含该附加成员，它指向一块被称为 Synchronization Block 的内存块，用于管理对象同步，同样占用 4 个字节的内存空间。

NextObjPtr，由托管堆维护的一个指针，用于标识下一个新建对象分配时在托管堆中所处的位置。CLR 初始化时，NextObjPtr 位于托管堆的基地址。

因此，我们对引用类型分配过程应该有个基本的了解，由于本篇示例中 FileStream 类型的继承关系相对复杂，在此本节实现一个相对简单的类型来做说明：

```
public class UserInfo
{
   private Int32 age = -1;
   private char level = 'A';
```

```
}
public class User
{
    private Int32 id;
    private UserInfo user;
}
public class VIPUser : User
{
    public bool isVip;
    public bool IsVipUser()
    {
        return isVip;
    }
    public static void Main()
    {
        VIPUser aUser;
        aUser = new VIPUser();
        aUser.isVip = true;
        Console.WriteLine(aUser.IsVipUser());
    }
}
```

将上述实例的执行过程，反编译为 IL 语言可知：new 关键字被编译为 newobj 指令来完成对象创建工作，进而调用类型的构造器来完成其初始化操作，在此我们详细的描述其执行的具体过程。

首先，将声明一个引用类型变量 aUser：

```
VIPUser aUser;
```

它仅是一个引用（指针），保存在线程的堆栈上，占用 4Byte 的内存空间，将用于保存 VIPUser 对象的有效地址，其执行过程正是上文描述的在线程栈上的分配过程。此时 aUser 未指向任何有效的实例，因此被自行初始化为 null，试图对 aUser 的任何操作将抛出 NullReferenceException 异常。

接着，通过 new 操作执行对象创建：

```
aUser = new VIPUser();
```

如上文所言，该操作对应于执行 newobj 指令，其执行过程又可细分为以下几步：

（a）CLR 按照其继承层次进行搜索，计算类型及其所有父类的字段，该搜索将一直递归到 System.Object 类型，并返回字节总数，以本例而言类型 VIPUser 需要的字节总数为 15Byte，具体计算为：VIPUser 类型本身字段 isVip（bool 型）为 1Byte；父类 User 类型的字段 id（Int32 型）为 4Byte，字段 user 保存了指向 UserInfo 型的引用，因此占 4Byte，而同时还要为 UserInfo 分配 6Byte 字节的内存。

（b）实例对象所占的字节总数还要加上对象附加成员所需的字节总数，其中附加成员包括 TypeHandle 和 SyncBlockIndex，共计 8 字节（在 32 位 CPU 平台下）。因此，需要在托管堆上分配的字节总数为 23 字节，而堆上的内存块总是按照 4Byte 的倍数进行分配，因此本例中将分配 24 字节的地址空间。

（c）CLR 在当前 AppDomain 对应的托管堆上搜索，找到一个未使用的 24 字节的连续空间，并为其分配该内存地址。事实上，GC 使用了非常高效的算法来满足该请求，NextObjPtr 指针只需要向前推进 24 个字节，并清零原 NextObjPtr 指针和当前 NextObjPtr 指针之间的字节，然后返回原 NextObjPtr 指针地址即可，该地址正是新创建对象的托管堆地址，也就是 aUser 引用指向的实例地址。而此时的 NextObjPtr 仍指向下一个新建对象的位置。注意，栈的分配是向低地址扩展，而堆的分配是向高地址扩展。

另外，实例字段的存储是有顺序的，由上到下依次排列，父类在前子类在后，详细的分析请参见 1.2 节“什么是继承”。

在上述操作时，如果试图分配所需空间而发现内存不足时，GC 将启动垃圾收集操作来回收垃圾对象所占的内存，我们将在下一节对此做详细的分析。

最后，调用对象构造器，进行对象初始化操作，完成创建过程。该构造过程，又可细分为以下几个环节：

（a）构造 VIPUser 类型的 Type 对象，主要包括静态字段、方法描述、实现的接口等，并将其分配在上文提到托管堆的 Loader Heap 上。

（b）初始化 aUser 的两个附加成员：TypeHandle 和 SyncBlockIndex。将 TypeHandle 指针指向 Loader Heap 上的 MethodTable，CLR 将根据 TypeHandle 来定位具体的 Type；将 SyncBlockIndex 指针指向 Synchronization Block 的内存块，用于在多线程环境下对实例对象的同步操作。

（c）调用 VIPUser 的构造器，进行实例字段的初始化。实例初始化时，会首先向上递归执行父类初始化，直到完成 System.Object 类型的初始化，然后再返回执行子类的初始化，直到执行 VIPUser 类为止。以本例而言，初始化过程首先执行 System.Object 类，再执行 User 类，最后才是 VIPUser 类。最终，newobj 分配的托管堆的内存地址，被传递给 VIPUser 的 this 参数，并将其引用传给栈上声明的 aUser。

关于构造函数的执行顺序，本书在 7.8 节“动静之间：静态和非静态”一节有较为详细的论述。

上述过程，基本完成了一个引用类型创建、内存分配和初始化的整个流程，然而该过程只能看作是一个简化的描述，实际的执行过程更加复杂，涉及一系列细化的过程和操作。对象创建并初始化之后，内存的布局，可以表示为图 5-5。

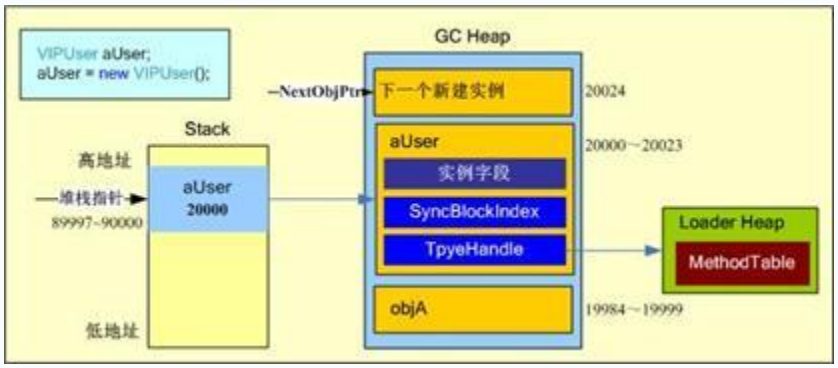


图 5-5 堆上的内存分配

由上面的分析可知，在托管堆中增加新的实例对象，只是将 NextObjPtr 指针增加一定的数值，再次新增的对象将分配在当前 NextObjPtr 指向的内存空间，因此在托管堆栈中，连续分配的对象在内存中一定是连续的，这种分配机制非常高效。

3. 必要的补充

有了对象创建的基本流程概念，下面的几个问题时常引起大家的思考，在此本节一并做以探索：

l 值类型中的引用类型字段和引用类型中的值类型字段，其分配情况又是如何？

这一思考其实是一个问题的两个方面：对于值类型嵌套引用类型的情况，引用类型变量作为值类型的成员变量，在堆栈上保存该成员的引用，而实际的引用类型仍然保存在 GC 堆上；对于引用类型嵌套值类型的情况，则该值类型字段将作为引用类型实例的一部分保存在 GC 堆上。本书在 4.2 节“品味类型——值类型与引用类型”一节对这种嵌套结构，有较详细的分析。

l 方法保存在 Loader Heap 的 MethodTable 中，那么方法调用时又是怎样的过程呢？

如上所言，MethodTable 中包含了类型的元数据信息，类在加载时会在 Loader Heap 上创建这些信息，一个类型在内存中对应一份 MethodTable，其中包含了所有的方法、静态字段和实现的接口信息等。对象实例的 TypeHandle 在实例创建时，将指向 MethodTable 开始位置的偏移处（默认偏移 12Byte）。通过对象实例调用某个方法时，CLR 根据 TypeHandle 可以找到对应的 MethodTable，进而可以定位到具体的方法，再通过 JIT Compiler 将 IL 指令编译为本地 CPU 指令，该指令将保存在一个动态内存中，然后在该内存地址上执行该方法，同时该 CPU 指令被保存起来用于下一次的执行。

在 MethodTable 中，包含一个 Method Slot Table，称为方法槽表，该表是一个基于方法实现的线性链表，并按照以下顺序排列：继承的虚方法、引入的虚方法、实例方法和静态方法。方法表在创建时，将按照继承层次向上搜索父类，直到 System.Object 类型，如果子类覆写了父类方法，则将会以子类方法覆盖父类虚方法。关于方法表的创建过程，可以参考 2.2 节“什么是继承”中的描述。

l 静态字段的内存分配和释放，又有何不同？

静态字段也保存在方法表中，位于方法表的槽数组后，其生命周期为从创建到 AppDomain 卸载。因此一个类型无论创建多少个对象，其静态字段在内存中也只有一份。静态字段只能由静态构造函数进行初始化，静态构造函数确保在任何对象创建前，或者在任何静态字段或方法被引用前执行，其详细的执行顺序在 7.8 节“动静之间：静态和非静态”有所讨论。

5.2.3 结论

对象创建过程的了解，是从底层接触 CLR 运行机制的入口，也是认识.NET 自动内存管理的关键。通过本节的详细论述，关于对象的创建、内存分配、初始化过程和方法调用等技术都会建立一个相对全面的理解，同时也清楚地把握了线程栈和托管堆的执行机制。

对象总是有生有灭，本节简述其生，下一节讨论其亡。继续本章对自动内存管理技术的认识，下一个重要的内容就是：垃圾回收机制。

5.3 垃圾回收

本节将介绍以下内容：

— .NET 垃圾回收机制

— 非托管资源的清理

5.3.1 引言

.NET 自动内存管理将开发人员从内存错误的泥潭中解放出来，这一切都归功于垃圾回收（GC，Garbage Collection）机制。

通过对对象创建全过程的讲述，我们理解了 CLR 执行对象内存分配的基本面貌。一个分配了内存空间和完成初始化的对象实例，就是一个 CLR 世界中的新生命体，其生命周期大概可以概括为：对象在系统中进行一定的操作和应用，到一定阶段它将不被系统中任何对象引用或操作，则表示该对象不会再被使用。因此，对象符合了可以销毁的条件，而 CLR 可能不会马上执行销毁操作，而是在适当的时间执行该对象的内存销毁。一旦被执行销毁，对象及其成员将不可在运行时使用，最后由垃圾收集器释放其内存资源，完成一个对象由生而灭的全过程。

由此可见，在.NET 中自动内存管理是由垃圾回收器来执行的，GC 自动完成对托管堆的全权管理，然而一股脑将所有事情交给 GC，并非万全保障。基于性能与安全的考虑，很有必要对 GC 的工作机理、执行过程，以及对非托管资源的清理做一个讨论。

5.3.2 垃圾回收

顾名思义，垃圾回收就是清理内存中的垃圾，因此了解垃圾回收机制就应从以下几个方面着手：

- 什么样的对象被 GC 认为是垃圾呢？
- 如何回收？
- 何时回收？
- 回收之后，又执行哪些操作？

清楚地回答上述几个问题，也就基本了解.NET 的垃圾回收机制。下面本节就逐一揭开这几个问题的答案。

- 什么样的对象被 GC 认为是垃圾呢？

简单地说，一个对象成为“垃圾”就表示该对象不被任何其他对象所引用。因此，GC 必须采用一定的算法在托管堆中遍历所有对象，最终形成一个可达对象图，而不可达的对象将成为被释放的垃圾对象等待收集。

- 如何回收？

每个应用程序有一组根（指针），根指向托管堆中的存储位置，由 JIT 编译器和 CLR 运行时维护根指针列表，主要包括全局变量、静态变量、局部变量和寄存器指针等。下面以一个简单的示例来说明，GC 执行垃圾收集的具体过程。

```
class A
{
   private B objB;
   public A(B o)
   {
      objB = o;
   }
   ～A()
   {
      Console.WriteLine("Destory A.");
   }
```

```
}
class B
{
    private C objC;
    public B(C o)
    {
        objC = o;
    }
    ～B()
    {
        Console.WriteLine("Destory B.");
    }
}
class C
{
    ～C()
    {
        Console.WriteLine("Destory C.");
    }
}
public class Test_GCRun
{
    public static void Main()
    {
        A a = new A(new B(new C()));
        //强制执行垃圾回收
        GC.Collect(0);
        GC.WaitForPendingFinalizers();
    }
}
```

在上述执行中，当创建类型 A 的对象 a 时，在托管堆中将新建类型 B 的实例（假设表示为 objB）和类型 C 的实例（假设表示为 objC），并且这几个对象之间保存着一定的联系。而局部变量 a 则相当于一个应用程序的根，假设其在托管堆中对应的实例表示为 objA，则当前的引用关系可以表示为图 5-6。

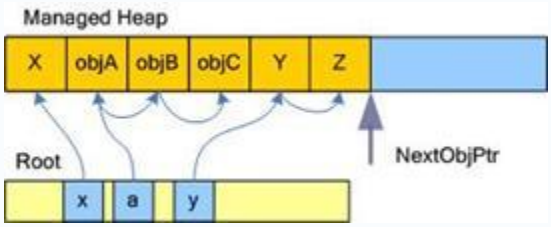


图 5-6 垃圾收集执行前的托管堆

垃圾收集器正是通过根指针列表来获得托管堆中的对象图，其中定义了应用程序根引用的托管堆中的对象，当垃圾收集器启动时，它假设所有对象都是可回收的垃圾，并开始遍历所有的根，将根引用的对象标记为可达对象添加到可达对象图中，在遍历过程中，如果根引用的对象还引用着其他对象，则该对象也被添加到可达对象图中，依次类推，垃圾收集器通过根列表的递归遍历，将能找到所有可达对象，并形成一个可达对象图。同时那些不可达对象则被认为是可回收对象，垃圾收集器接着运行垃圾收集进程来释放垃圾对象的内存空间。通常，将这种收集算法称为：标记和清除收集算法。

在上例中，a 可以看出是应用程序的一个根，它在托管堆中对应的对象 objA 就是一个可达对象，而对象 objA 依次关联的 objB、objC 都是可达对象，被添加到可达对象图中。当 Main 方法运行结束时，a 不再被引用，则其不再是一个根，此时通过 GC.Collect 强制启动垃圾收集器，a 对应的 objA，以及相关联的 objB 和 objC 将成为不可达对象，我们从执行结果中可以看出类型 A、B、C 的析构方法被分别调用，由此可以分析垃圾回收执行了对 objA、objB、objC 实例的内存回收。

l 何时回收？

垃圾收集器周期性的执行内存清理工作，一般在以下情况出现时垃圾收集器将会启动：

（1）内存不足溢出时，更确切地应该说是第 0 代对象充满时。

（2）调用 GC.Collect 方法强制执行垃圾回收。

（3）Windows 报告内存不足时，CLR 将强制执行垃圾回收。

（4）CLR 卸载 AppDomain 时，GC 将对所有代龄的对象执行垃圾回收。

（5）其他情况，例如物理内存不足，超出短期存活代的内存段门限，运行主机拒绝分配内存等等。

作为开发人员，我们无需实现任何代码来管理应用程序中各个对象的生命周期，CLR 知道何时去执行垃圾收集工作来满足应用程序的内存需求。当上述情况发生时，GC 将着手进行内存清理，当内存释放之前 GC 会首先检查终止化链表中是否有记录来决定在释放内存之前执行非托管资源的清理工作，然后才执行内存释放。

同时，微软强烈建议不要通过 GC.Collect 方法来强制执行垃圾收集，因为那会妨碍 GC 本身的工作方式，通过 Collect 会使对象代龄不断提升，扰乱应用程序的内存使用。只有在明确知道有大量对象停止引用时，才考虑使用 GC.Collect 方法来调用收集器。

l 回收之后，又执行哪些操作？

GC 在垃圾回收之后，堆上将出现多个被收集对象的“空洞”，为避免托管堆的内存碎片，会重新分配内存，压缩托管堆，此时 GC 可以看出是一个紧缩收集器，其具体操作为：GC 找到一块较大的连续区域，然后将未被回收的对象转移到这块连续区域，同时还要对这些对象重定位，修改应用程序的根以及发生引用的对象指针，来更新复制后的对象位置。因此，势必影响 GC 回收的系统性能，而 CLR 垃圾收集器使用了 Generation 的概念来提升性能，还有其他一些优化策略，如并发收集、大对象策略等，来减少垃圾收集对性能的影响。例如，上例中执行后的托管堆的内存状况可以表示为图 5-7。

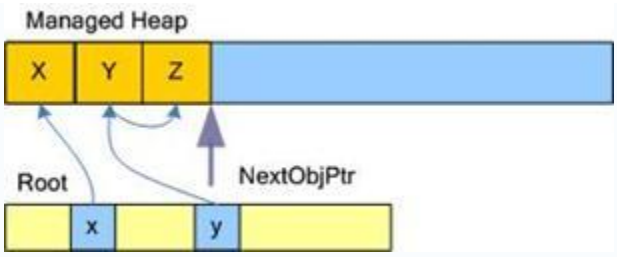


图 5-7 垃圾收集执行后的托管堆

CLR 提供了两种收集器：工作站垃圾收集器（Workstation GC，包含在 mscorwks.dll）和服务器垃圾收集器（Server GC，包含在 mscorsvr.dll），分别为不同的处理机而设计，默认情况为工作站收集器。工作站收集器主要应用于单处理器系统，工作站收集器尽可能地通过减少垃圾回收过程中程序的暂停次数来提高性能；服务器收集器，专为具有多处理器的服务器系统而设计，采用并行算法，每个 CPU 都具有一个 GC 线程。在 CLR 加载到进程时，可以通过 CorBindToRuntimeEx()函数来选择执行哪种收集器，选择合适的收集器也是有效、高效管理的关键。

关于代龄（Generation）

接下来对文中多次提到的代龄概念做以解释，来理解 GC 在性能优化方面的策略机制。

垃圾收集器将托管堆中的对象分为三代，分别为：0、1 和 2。在 CLR 初始化时，会选择为三代设置不同的阈值容量，一般分配为：第 0 代大约 256KB，第 1 代 2MB，第 2 代 10MB，可表示为如图 5-8 所示。显然，容量越大效率越低，而 GC 收集器会自动调节其阈值容量来提升执行效率，第 0 代对象的回收效率肯定是最高的。

图 5-8 代龄的阈值容量

在 CLR 初始化后，首先被添加到托管堆中的对象都被定为第 0 代，如图 5-9 所示。当有垃圾回收执行时，未被回收的对象代龄将提升一级，变成第 1 代对象，而后新建的对象仍为第 0 代对象。也就是说，代龄越小，表示对象越新，通常情况下其生命周期也最短，因此垃圾收集器总是首先收集第 0 代的不可达对象内存。

随着对象的不断创建，垃圾收集再次启动时则只会检查 0 代对象，并回收 0 代垃圾对象。而 1 代对象由于未达到预定的 1 代容量阈值，则不会进行垃圾回收操作，从而有效的提高了垃圾收集的效率，这就是代龄机制在垃圾回收中的性能优化作用。

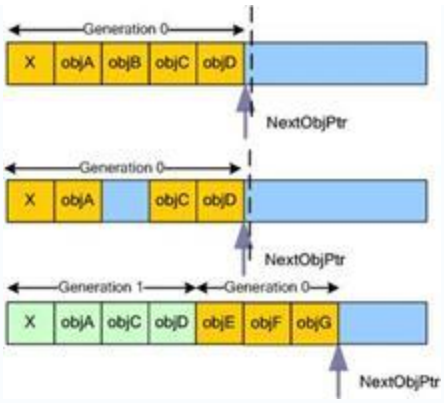


图 5-9 初次执行垃圾回收

那么，垃圾收集器在什么情况下，才执行对第 1 代对象的收集呢？答案是仅当第 0 代对象释放的内存不足以创建新的对象，同时 1 代对象的体积也超出了容量阈值时，垃圾收集器将同时对 0 代和 1 代对象进行垃圾回收。回收之后，未被回收的 1 代对象升级为 2 代对象，未被回收的 0 代对象升级为 1 代对象，而后新建的对象仍为第 0 代对象，如图 5-10 所示。垃圾收集正是对上述过程的不断重复，利用分代机制提高执行效率。

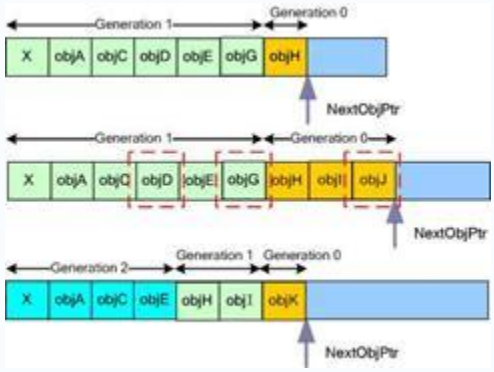


图 5-10 执行 1 代对象垃圾回收

通过 GC.Collect 方法可以指定对从第 0 代到指定代的对象进行回收，通过 GC. MaxGeneration 来获取框架版本支持的代龄的最大有效值。

规则小结

关于垃圾回收，对其有以下几点小结：

- CLR 提供了一种分代式、标记清除型 GC，利用标记清除算法来对不同代龄的对象进行垃圾收集和内存紧缩，保证了运算效率和执行优化。

l 一个对象没有被其他任何对象引用，则该对象被认为是可以回收的对象。

l 最好不要通过调用 GC.Collect 来强制执行垃圾收集。

l 垃圾对象并非立即被执行内存清理，GC 可以在任何时候执行垃圾收集。

l 对“胖”对象考虑使用弱引用，以提高性能，详见 5.4 节“性能优化的多方探讨”。

5.3.3 非托管资源清理

对于大部分的类型来说，只存在内存资源的分配与回收问题，因此 CLR 的处理已经能够满足这种需求，然而还有部分的类型不可避免的涉及访问其他非托管资源。常见的非托管资源包括数据库链接、文件句柄、网络链接、互斥体、COM 对象、套接字、位图和 GDI+对象等。

GC 全权负责了对托管堆的内存管理，而内存之外的资源，又该由谁打理？在.NET 中，非托管资源的清理，主要有两种方式：Finalize 方法和 Dispose 方法，这两种方法提供了在垃圾收集执行前进行资源清理的方法。Finalize 方式，又称为终止化操作，其大致的原理为：通过对自定义类型实现一个 Finalize 方法来释放非托管资源，而终止化操作在对象的内存回收之前通过调用 Finalize 方法来释放资源；Dispose 模式，指的是在类中实现 IDisposable 接口，该接口中的 Dispose 方法定义了显式释放由对象引用的所有非托管资源。因此，Dispose 方法提供了更加精确的控制方式，在使用上更加的灵活。

1. 终止化操作

对 C++程序员来说，提起资源释放，会首先想到析构器。不过，在.NET 世界里，没落的析构器已经被终结器取而代之，.NET 在语法上选择了类似的实现策略，例如你可以有如下定义：

```
class GCApp: Object
{
   ~GCApp()
   {
      //执行资源清理
   }
}
```

将上述代码编译为 IL：

```
.method family hidebysig virtual instance void
        Finalize() cil managed
{
 // 代码大小       14 (0xe)
 .maxstack  1
 .try
 {
   IL_0000:  nop
   IL_0001:  nop
   IL_0002:  leave.s    IL_000c
 }  // end .try
 finally
 {
   IL_0004:  ldarg.0
   IL_0005:  call       instance void [mscorlib]System.Object::Finalize()
   IL_000a:  nop
   IL_000b:  endfinally
 }  // end handler
 IL_000c:  nop
 IL_000d:  ret
} // end of method GCApp::Finalize
```

可见，编译器将～GCApp 方法编译为托管模块元数据中一个 Finalize 方法，由于示例本身没有实现任何资源清理代码，上述 Finalize 方法只是简单调用了 Object.Finalize 方法。可以通过重写基类的 Finalize 方法实现资源清理操作，注意：自.NET 2.0 起，C#编译器认为 Finalize 方法是一个特殊的方法，对其调用或重写必须使用析构函数语法来实现，不可以通过显式非覆写 Finalize 方法来实现。因此在自定义类型中重写 Finalize 方法将等效于：

```
protected override void Finalize()
{
   try
```

```
    {
        //执行自定义资源清理操作
    }
    finally
    {
        base.Finalize();
    }
}
```

由此可见，在继承链中所有实例将递归调用 base.Finalize 方法，也就是意味调用终结器释放资源时，将释放所有的资源，包括父类对象引用的资源。因此，在 C#中，也无需调用或重写 Object. Finalize 方法，事实上显示的重写会引发编译时错误，只需实现虚构函数即可。

在具体操作上，终结器的工作原理是这样的：在 Systm.Object 中，Finalize 方法被实现为一个受保护的虚方法，GC 要求任何需要释放非托管资源的类型都要重写该方法，如果一个类型及其父类均未重写 Systm.Object 的 Finalize 方法，则 GC 认为该类型及其父类不需要执行终止化操作，当对象变成不可达对象时，将不会执行任何资源清理操作；而如果只有父类重写了 Finalize 方法，则父类会执行终止化操作。因此，对于在类中重写了 Finalize 的方法（在 C#中实现析构函数），当 GC 启动时，对于判定为可回收的垃圾对象，GC 会自动执行其 Finalize 方法来清理非托管资源。例如通常情况下，对于 Window 资源的释放，是通过调用 Win32API 的 CloseHandle 函数来实现关闭打开的对象句柄。

对于重写了 Finalize 方法的类型来说，可以通过 GC. SuppressFinalize 来免除终结。

对于 Finalize 方式来说，存在如下几个弊端，因此一般情况下在自定义类型中应避免重写 Final ize 方法，这些弊端主要包括：

- 终止化操作的时间无法控制，执行顺序也不能保证。因此，在资源清理上不够灵活，也可能由于执行顺序的不确定而访问已经执行了清理的对象。
- Finalize 方法会极大地损伤性能，GC 使用一个终止化队列的内部结构来跟踪具有 Finalize 方法的对象。当重写了 Finalize 方法的类型在创建时，要将其指针添加到该终止化队列中，由此对性能产生影响；另外，垃圾回收时调用 Finalize 方法将同时清理所有的资源，包括其父类对象的资源，也是影响性能的一个因素。

- 重写了 Finalize 方法的类型对象，其引用类型对象的代龄将被提升，从而带来内存压力。
- Finalize 方法在某些情况下可能不被执行，例如可能某个终结器被无限期的阻止，则其他终结器得不到调用。因此，应该确保重写的 Finalize 方法尽快被执行。

基于以上原因，应该避免重写 Finalize 方法，而实现 Dispose 模式来完成对非托管资源的清理操作，具体实现见下文描述。

对于 Finalize 方法，有以下规则值得总结：

- 在 C#中无法显示的重写 Finalize 方法，只能通过析构函数语法形式来实现。
- struct 中不允许定义析构函数，只有 class 中才可以，并且只能有一个。
- Finalize 方法不能被继承或重载。
- 析构函数不能加任何修饰符，不能带参数，也不能被显示调用，唯一的例外是在子类重写时，通过 base 调用父类 Finalize 方法，而且这种方式也被隐式封装在析构函数中。
- 执行垃圾回收之前系统会自动执行终止化操作。
- Finalize 方法中，可以实现使得被清理对象复活的机制，不过这种操作相当危险，而且没有什么实际意义，仅作参考，不推荐使用：

```
public class ReLife
{
   ～ReLife()
   {
      //对象重新被一个根引用
      Test_ReLife.Instance = this;
      //重新将对象添加到终止化队列
      GC.ReRegisterForFinalize(this);
   }
   public void ShowInfo()
   {
      Console.WriteLine("对象又复活了。");
   }
```

```
}
public class Test_ReLife
{
    public static ReLife Instance;
    public static void Main()
    {
        Instance = new ReLife();
        Instance = null;
        GC.Collect();
        GC.WaitForPendingFinalizers();
        //对象又复活了
        Instance.ShowInfo();
    }
}
```

2. Dispose 模式

另一种非托管资源的清理方式是 Dispose 模式，其原理是定义的类型必须实现 System.IDisposable 接口，该接口中定义了一个公有无参的 Dispose 方法，用户可以在该方法中实现对非托管资源的清理操作。在此，我们实现一个典型的 Dispose 模式：

```
class MyDispose : IDisposable
{
    //定义一个访问外部资源的句柄
    private IntPtr _handle;
    //标记 Dispose 是否被调用
    private bool disposed = false;
    //实现 IDisposable 接口
    public void Dispose()
    {
        Dispose(true);
        //阻止 GC 调用 Finalize 方法
        GC.SuppressFinalize(this);
```

```
    }
    //实现一个处理资源清理的具体方法
    protected virtual void Dispose(bool disposing)
    {
        if (! disposed)
        {
            if (disposing)
            {
                //清理托管资源
            }
            //清理非托管资源
            if (_handle != IntPtr.Zero)
            {
                //执行资源清理，在此为关闭对象句柄
                CloseHandle(_handle);
                _handle = IntPtr.Zero;
            }
        }
        disposed = true;
    }
    public void Close()
    {
        //在内部调用 Dispose 来实现
        Dispose();
    }
}
```

在上述实现 Dispose 模式的典型操作中，有几点说明：

- Dispose 方法中，应该使用 GC. SuppressFinalize 防止 GC 调用 Finalize 方法，因为显式调用 Dispose 显然是较佳选择。
- 公有 Dispose 方法不能实现为虚方法，以禁止在派生类中重写。

- 在该模式中，公有 Dispose 方法通过调用重载虚方法 Dispose（bool disposing）方法来实现，具体的资源清理操作实现于虚方法中。两种策略的区别是：disposing 参数为真时，Dispose 方法由用户代码调用，可释放托管或者非托管资源；disposing 参数为假时，Dispose 方法由 Finalize 调用，并且只能释放非托管资源。
- disposed 字段，保证了两次调用 Dispose 方法不会抛出异常，值得推荐。
- 派生类中实现 Dispose 模式，应该重写基类的受保护 Dispose 方法，并且通过 base 调用基类的 Dispose 方法，以确保释放继承链上所有对象的引用资源，在整个继承层次中传播 Dispose 模式。

```
protected override void Dispose(bool disposing)
{
   if (!disposed)
   {
      try
      {
         //子类资源清理
         //.....
         disposed = true;
      }
      finally
      {
         base.Dispose(disposing);
      }
   }
}
```

- 另外，基于编程习惯的考虑，一般在实现 Dispose 方法时，会附加实现一个 Close 方法来达到同样的资源清理目的，而 Close 内部其实也是通过调用 Dispose 来实现的。

3. 最佳策略

最佳的资源清理策略，应该是同时实现 Finalize 方式和 Dispose 方式。一方面，Dispose 方法可以克服 Finalize 方法在性能上的诸多弊端；另一方面，Finalize 方法又能够确保没有显式调用 Dispo

se 方法时，也自行回收使用的所有资源。事实上，.NET 框架类库的很多类型正是同时实现了这两种方式，例如 FileStream 等。因此，任何重写了 Finalize 方法的类型都应实现 Dispose 方法，来实现更加灵活的资源清理控制。

因此，我们模拟一个简化版的文件处理类 FileDealer，其中涉及对文件句柄的访问，以此来说明在自定义类型中对非托管资源的清理操作，在此同时应用 Finalize 方法和 Dispose 方法来实现：

```
class FileDealer: IDisposable
{
   //定义一个访问文件资源的 Win32 句柄
   private IntPtr fileHandle;
   //定义引用的托管资源
   private ManagedRes managedRes;
   //定义构造器，初始化托管资源和非托管资源
   public FileDealer(IntPtr handle, ManagedRes res)
   {
      fileHandle = handle;
      managedRes = res;
   }
   //实现终结器，定义 Finalize
   ～FileDealer()
   {
      if(fileHandle != IntPtr.Zero)
      {
         Dispose(false);
      }
   }
   //实现 IDisposable 接口
   public void Dispose()
   {
      Dispose(true);
      //阻止 GC 调用 Finalize 方法
```

```
        GC.SuppressFinalize(this);
    }
    //实现一个处理资源清理的具体方法
    protected virtual void Dispose(bool disposing)
    {
        if (disposing)
        {
            //清理托管资源
            managedRes.Dispose();
        }
        //执行资源清理，在此为关闭对象句柄
        if (fileHandle != IntPtr.Zero)
        {
            CloseHandle(fileHandle);
            fileHandle = IntPtr.Zero;
        }
    }
    public void Close()
    {
        //在内部调用 Dispose 来实现
        Dispose();
    }
    //实现对文件句柄的其他应用方法
    public void Write() { }
    public void Read() { }
    //引入外部 Win32API
    [DllImport("Kernel32")]
    private extern static Boolean CloseHandle(IntPtr handle);
}
```

注意，本例只是一个简单化的演示，并非专门的设计文件操作类型。在.NET 框架中的 FileStream 类中，文件句柄被封装到一个 SafeFileHandle 的类中实现，该类间接继承于 SafeHandle 抽象类。

其中 SafeHandle 类型是一个对操作系统句柄的包装类，实现了对本地资源的封装，因此对于大部分的资源访问应用来说，以 SafeHandle 的派生类作为操作系统资源的访问方式，是安全而可信的，例如 FileStream 中的 SafeFileHandle 类，就是对文件句柄的有效包装。

4. using 语句

using 语句简化了资源清理代码实现，并且能够确保 Dispose 方法得到调用，因此值得推荐。凡是实现了 Dispose 模式的类型，均可以 using 语句来定义其引用范围。关于 using 语句的详细描述，请参考 6.3 节“using 的多重身份”，在此我们将演示引用 using 语句实现对上述 FileDealer 类的访问：

```
public static void Main()
{
   using(FileDealer fd = new FileDealer(new IntPtr(), new ManagedRes()))
   {
      fd.Read();
   }
}
```

上述执行，等效于实现了一个 try/finally 块，并将资源清理代码置于 finally 块中：

```
public static void Main()
{
   FileDealer fd = null;
   try
   {
      fd = new FileDealer(new IntPtr(), new ManagedRes());
      fd.Read();
   }
   finally
   {
      if(fd != null)
         fd.Dispose();
   }
```

```
}
```

5. 规则所在

对于 Finalize 方法和 Dispose 方法，有如下的规则，留作参考：

- 对于非托管资源的清理，Finalize 由 GC 自行调用，而 Dispose 由开发者强制执行调用。
- 尽量避免使用 Finalize 方式来清理资源，必须实现 Finalize 时，也应一并实现 Dispose 方法，来提供显式调用的控制权限。
- 通过 GC. SuppressFinalize 可以免除终结。
- 垃圾回收时，执行终结器的准确时间是不确定的，除非显式的调用 Dispose 或者 Close 方法。
- 强烈建议不要重写 Finalize 方法，同时强烈建议在任何有非托管资源访问的类中同时实现终止化操作和 Dispose 模式。
- Finalize 方法和 Dispose 方法，只能清理非托管资源，释放内存的工作仍由 GC 负责。
- 对象使用完毕应该立即释放其资源，最好显式调用 Dispose 方法来实现。

5.3.4 结论

.NET 自动内存管理，是 CLR 提供的最为重要的基础服务之一。通过本节对垃圾回收和非托管资源的管理分析，可以基本了解 CLR 对系统资源管理回收方面的操作本质。对于开发人员来说，GC 全权负责了对内存的管理、监控与回收，我们应将更多的努力关注于非托管资源的清理方式的理解和应用上，以提升系统资源管理的性能和安全。

5.4 性能优化的多方探讨

本节将介绍以下内容：

— .NET 性能优化的策略探讨

— 多种性能优化分析

5.4.1 引言

什么才算良好的软件产品？业务流程、用户体验、安全性还有性能，一个都不能少。因此，良好的系统性能，是用户评价产品的重要指标之一。交易所里数以万亿计的数据要想保证全球股市交易的畅通无阻，稳定运行和高效的性能缺一不可。而小型系统的性能，同样会受到关注，因为谁也不想访问一个蜗牛般的软件系统。

因此，性能是系统设计的重要因素，然而影响系统性能的要素又是多种多样，例如硬件环境、数据库设计以及软件设计等等。本节将关注集中在.NET 中最常见的性能杀手，并以条款的方式来一一展现，某些可能是规则，某些可能是习惯，而某些可能是语法。

本节在分析了.NET 自动内存管理机制的基础上，来总结.NET 开发中值得关注的性能策略，并以这些策略作为选择的依据和平衡的杠杆。同时，本节的优化条款主要针对.NET 基础展开，而不针对专门的应用环节，例如网站性能优化、数据库优化等。

孰优孰劣，比较应用中自有体现。

5.4.2 性能条款

¡ Item1：推荐以 Dispose 模式来代替 Finalize 方式。

在本章中关于非托管资源的清理，主要有终止化操作和 Dispose 模式两种，其中 Finalize 方式存在执行时间不确定，运行顺序不确定，同时对垃圾回收的性能有极大的损伤。因此强烈建议以 Dispose 模式来代替 Finalize 方式，在带来性能提升的同时，实现了更加灵活的控制权。

对于二者的详细比较，请参见 5.3 节“垃圾回收”的讨论。

¡ Item2：选择合适的垃圾收集器：工作站 GC 和服务期 GC。

.NET CLR 实现了两种垃圾收集器，不同的垃圾收集器应用不同的算法，分别为不同的处理机而设计：工作站 GC 主要应用于单处理器系统，而服务器收集器专为多处理器的服务器系统设计，默认情况为工作站收集器。因此，在多处理器系统中如果使用工作站收

集器，将大大降低系统的性能，无法适应高吞吐量的并行操作模式，为不同主机选择合适的垃圾收集器是有效提高性能的关键之一。

¡ Item3：在适当的情况下对对象实现弱引用。

为对象实现弱引用，是有效提高性能的手段之一。弱引用是对象引用的一种“中间态”，实现了对象既可以通过 GC 回收其内存，又可被应用程序访问的机制。这种看似矛盾的解释，的确对胖对象的内存性能带来提升，因为胖对象需要大量的内存来创建，弱引用机制保证了胖对象在内存不足时 GC 可以回收，而不影响内存使用，在没有被 GC 回收前又可以再次引用该对象，从而达到空间与时间的双重节约。

在.NET 中，WeakReference 类用于表示弱引用，通过其 Target 属性来表示要追踪的对象，通过其值赋给变量来创建目标对象的强引用，例如：

```
public void WeakRef()
{
   MyClass mc = new MyClass();
   //创建弱引用
   WeakReference wr = new WeakReference(mc);
   //移除强引用
   mc = null;
   if (wr.IsAlive)
   {
      //弱引用转换为强引用，对象可以再次使用
      mc = wr.Target as MyClass;
   }
   else
   {
      //对象已经被回收，重新创建
      mc = new MyClass();
   }
}
```

关于弱引用的相关讨论，参见 5.3 节“垃圾回收”。

¡ Item4：尽可能以 using 来执行资源清理。

以 using 语句来执行实现了 Dispose 模式的对象，是较好的资源清理选择，简洁优雅的代码实现，同时能够保证自动执行 Dispose 方法来销毁非托管资源，在本章已做详细讨论，因此值得推荐。

¡ Item5：推荐使用泛型集合来代替非泛型集合。

泛型实现了一种类型安全的算法重用，其最直接的应用正是在集合类中的性能与安全的良好体现，因此我们建议以泛型集合来代替非泛型集合，以 List<T>和 ArrayList 为例来做以说明：

```
public static void Main()
{
   //List<T>性能测试
   List<Int32> list = new List<Int32>();
   for (Int32 i = 0; i < 10000; i++)
      //未发生装箱
      list.Add(i);
   //ArrayList 性能测试
   ArrayList al = new ArrayList();
   for (Int32 j = 0; j < 10000; j++)
      //发生装箱
      al.Add(j);
}
```

上述示例，仅仅给出了泛型集合和非泛型集合在装箱操作上引起的差别，同样的拆箱操作也伴随了这两种不同集合的取值操作。同时，大量的装箱操作会带来频繁的垃圾回收，类型转换时的安全检查，都不同程度的影响着性能，而这些弊端在泛型集合中荡然无存。

必须明确的是，泛型集合并不能完全代替非泛型集合的应用，.NET 框架类库中有大量的集合类用以完成不同的集合操作，例如 ArrayList 中包含的很多静态方法是 List<T>所没有的，而这些方法又能为集合操作带来许多便利。因此，恰当地做出选择是非常重要的。

注意，这种性能差别对值类型的影响较大，而引用类型不存在装箱与拆箱问题，因此性能影响不是很明显。关于集合和泛型的讨论，详见 7.9 节“集合通论”和第 10 章“接触泛型”中的讨论。

¡ Item6：初始化时最好为集合对象指定大小。

长度动态增加的集合类，例如 ArrayList、Queue 的等。可以无需指定其容量，集合本身能够根据需求自动增加集合大小，为程序设计带来方便。然而，过分依赖这种特性并非好的选择，因为集合动态增加的过程是一个内存重新分配和集合元素复制的过程，对性能造成一定的影响，所以有必要在集合初始化时指定一个适当的容量。例如：

```
public static void Main()
{
   ArrayList al = new ArrayList(2);
      al.Add("One");
      al.Add("Two");
      //容量动态增加一倍
      al.Add("Three");
      Console.WriteLine(al.Capacity);
   }
```

¡ Item7：特定类型的 Array 性能优于 ArrayList。

ArrayList 只接受 Object 类型的元素，向 ArrayList 添加其他值类型元素会发生装箱与拆箱操作，因此在性能上使用 Array 更具优势，当然 object 类型的数组除外。不过，ArrayList 更容易操作和使用，所以这种选择同样存在权衡与比较。

¡ Item8：字符串驻留机制，是 CLR 为 String 类型实现的特殊设计。

String 类型无疑是程序设计中使用最频繁、应用最广泛的基元类型，因此 CLR 在设计上为了提升 String 类型性能考虑，实现了一种称为“字符串驻留”的机制，从而实现了相同字符串可能共享内存空间。同时，字符串驻留是进程级的，垃圾回收不能释放 CLR 内部

哈希表维护的字符串对象，只有进程结束时才释放。这些机制均为 String 类型的性能提升和内存优化提供了良好的基础。

关于 String 类型及其字符串驻留机制的理解，详见 8.3“如此特殊：大话 string”。

¡ Item9：合理使用 System.String 和 System.Text.StringBuilder。

在简单的字符串操作中使用 String，在复杂的字符串操作中使用 StringBuilder。简单地说，StringBuilder 对象的创建代价较大，在字符串连接目标较少的情况下，应优先使用 String 类型；而在有大量字符串连接操作的情况下，应优先考虑 StringBuilder。

同时，StringBuilder 在使用上，最好指定合适的容量值，否则由于默认容量的不足而频繁进行内存分配的操作会影响系统性能。

关于 String 和 StringBuilder 的性能比较，详见 8.3“如此特殊：大话 string”的讨论。

¡ Item10：尽量在子类中重写 ToString 方法。

ToString 方法是 System.Object 提供的一个公有的虚方法，.NET 中任何类型都可继承 System.Object 类型提供的实现方法，默认为返回类型全路径名称。在自定义类或结构中重写 ToString 方法，除了可以有效控制输出结果，还能在一定程度上减少装箱操作的发生。

```
public struct User
{
   public string Name;
   public Int32 Age;
   //避免方法调用时的装箱
   public override string ToString()
   {
      return "Name: " + Name + ", Age:" + Age.ToString();
   }
}
```

关于 ToString 方法的讨论，可以参考 8.1 节“万物归宗：System.Object”。

¡ Item11：其他推荐的字符串操作。

字符串比较，常常习惯的做法是：

```
public bool StringCompare(string str1, string str2)
```

```
{
   return str1 == str2;
}
```

而较好的实现应该是：

```
public int StringCompare(string str1, string str2)
{
   return String.Compare(str1, str2);
}
```

二者的差别是：前者调用 String.Equals 方法操作，而后者调用 String. Compare 方法来实现。String.Equals 方法实质是在内部调用一个 EqualsHelper 辅助方法来实施比较，内部处理相对复杂。因此，建议使用 String.Compare 方式进行比较，尤其是非大小写敏感字符串的比较，在性能上更加有效。

类似的操作包含字符串判空的操作，推荐的用法以 Length 属性来判断，例如：

```
public bool IsEmpty(string str)
{
   return str.Length == 0;
}
```

¡ Item12：for 和 foreach 的选择。

推荐选择 foreach 来处理可枚举集合的循环结构，原因如下：

- .NET 2.0 以后编译器对 foreach 进行了很大程度的改善，在性能上 foreach 和 for 实际差别不大。
- foreach 语句能够迭代多维数组，能够自动检测数组的上下限。
- foreach 语句能够自动适应不同的类型转换。
- foreach 语句代码更简洁、优雅，可读性更强。

```
public static void Main()
{
   ArrayList al = new ArrayList(3);
   al.Add(100);
   al.Add("Hello, world.");
   al.Add(new char[] { 'A', 'B', 'C' });
   foreach (object o in al)
      Console.WriteLine(o.ToString());
```

```
    for (Int32 i = 0; i < al.Count; i++)
        Console.WriteLine(al[i].ToString());
}
```

¡ Item13：以多线程处理应对系统设计。

毫无疑问，多线程技术是轻松应对多任务处理的最强大技术，一方面能够适应用户的响应，一方面能在后台完成相应的数据处理，这是典型的多线程应用。在.NET 中，基于托管环境的多个线程可以在一个或多个应用程序域中运行，而应用多个线程来处理不同的任务也造成一定的线程同步问题，同时过多的线程有时因为占用大量的处理器时间而影响性能。

推荐在多线程编程中使用线程池，.NET 提供了 System.Threading.ThreadPool 类来提供对线程池的封装，一个进程对应一个 ThreadPool，可以被多个 AppDomain 共享，能够完成异步 I/O 操作、发送工作项、处理计时器等操作，.NET 内部很多异步方法都使用 ThreadPool 来完成。在此做以简单的演示：

```
class ThreadHandle
{
    public static void Main()
    {
        ThreadHandle th = new ThreadHandle();
        //将方法排入线程池队列执行
        ThreadPool.QueueUserWorkItem(new WaitCallback(th.MyProcOne), "线程 1");
        Thread.Sleep(1000);
        ThreadPool.QueueUserWorkItem(new WaitCallback(th.MyProcTwo), "线程 2");
        //实现阻塞主线程
        Console.Read();
    }
    //在不同的线程执行不同的回调操作
    public void MyProcOne(object stateInfo)
    {
        Console.WriteLine(stateInfo.ToString());
        Console.WriteLine("起床了。");
    }
    public void MyProcTwo(object stateInfo)
    {
        Console.WriteLine(stateInfo.ToString());
        Console.WriteLine("刷牙了。");
    }
}
```

然而，多线程编程将使代码控制相对复杂化，不当的线程同步可能造成对共享资源的访问冲突等待，在实际的应用中应该引起足够的重视。

¡ Item14：尽可能少地抛出异常，禁止将异常处理放在循环内。

异常的发生必然造成系统流程的中断，同时过多的异常处理也会对性能造成影响，应该尽量用逻辑流程控制来代替异常处理。对于例行发生的事件，可以通过编程检查方式来判断其情况，而不是一并交给异常处理，例如：

```
Console.WriteLine(obj == null ? String.Empty : obj.ToString());
```

不仅简洁，而且性能表现更好，优于以异常方式的处理：

```
try
{
   Console.WriteLine(obj.ToString());
}
catch (NullReferenceException ex)
{
   Console.WriteLine(ex.Message);
}
```

当然，大部分情况下以异常机制来解决异常信息是值得肯定的，能够保证系统安全稳定的面对不可意料的错误问题。例如不可预计的溢出操作、索引越界、访问已关闭资源等操作，则应以异常机制来处理。

关于异常机制及其性能的讨论话题，详见 8.6 节“直面异常”的分析。

¡ Item15：捕获异常时，catch 块中尽量指定具体的异常筛选器，多个 catch 块应该保证异常由特殊到一般的排列顺序。

指定具体的异常，可以节约 CLR 搜索异常的时间；而 CLR 是按照自上而下的顺序搜索异常，因此将特定程度较高的排在前面，而将特定程度较低的排在后面，否则将导致编译错误。

¡ Item16：struct 和 class 的性能比较。

基于性能的考虑，在特殊情况下，以 struct 来实现对轻量数据的封装是较好的选择。这是因为，struct 是值类型，数据分配于线程的堆栈上，因此具有较好的性能表现。在本章中，已经对值类型对象和引用类型对象的分配进行了详细讨论，由此可以看出在线程栈上进行内存分配具有较高的执行效率。

当然，绝大部分情况下，class 都具有不可代替的地位，在面向对象程序世界里更是如此。关于 strcut 和 class 的比较，详见 7.2 节“后来居上：class 和 struct”。

¡ Item17：以 is/as 模式进行类型兼容性检查。

以 is 和 as 操作符可以用于判断对象类型的兼容性，以 is 来实现类型判断，以 as 实现安全的类型转换，是值得推荐的方法。这样能够避免不必要的异常抛出，从而实现一种安全、灵活的转换控制。例如：

```
public static void Main()
{
   MyClass mc = new MyClass();
   if (mc is MyClass)
   {
      Console.WriteLine("mc is a MyClass object.");
   }
   object o = new object();
   MyClass mc2 = o as MyClass;
   if (mc2 != null)
   {
      //对转换类型对象执行操作
   }
}
```

详细的论述，请参见 7.5“恩怨情仇：is 和 as”。

¡ Item18：const 和 static readonly 的权衡。

const 是编译时常量，readonly 是运行时常量，所以 const 高效，readonly 灵活。在实际的应用中，推荐以 static readonly 来代替 const，以解决 const 可能引起的程序集引用不一致问题，还有带来的较多灵活性控制。

关于 const 和 readonly 的讨论，详细参见 7.1 节“什么才是不变：const 和 readonly”。

¡ Item19：尽量避免不当的装箱和拆箱，选择合适的代替方案。

通过本节多个条款的性能讨论，我们不难发现很多情况下影响性能的正是装箱和拆箱，例如非泛型集合操作，类型转换等，因此选择合适的替代方案是很有必要的。可以使用泛型集合来代替非泛型集合，可以实现多个重载方法以接受不同类型的参数来减少装箱，可以在子类中重写 ToString 方法来避免装箱等等。

关于装箱和拆箱的详细讨论，参见 4.4 节“皆有可能——装箱与拆箱”的深入分析。

¡ Item20：尽量使用一维零基数组。

CLR 对一维零基数组使用了特殊的 IL 操作指令 newarr，在访问数组时不需要通过索引减去偏移量来完成，而且 JIT 也只需执行一次范围检查，可以大大提升访问性能。在各种数组中其性能最好、访问效率最高，因此值得推荐。

关于一维零基数组的讨论，参加 3.4 节“经典指令解析之实例创建”的分析。

¡ Item21：以 FxCop 工具，检查你的代码。

FxCop 是微软开发的一个针对.NET 托管环境的代码分析工具，如图 5-11 所示，可以帮助我们检查分析现存托管程序在设计、本地化、命名规范、性能和安全性几个方面是否规范。

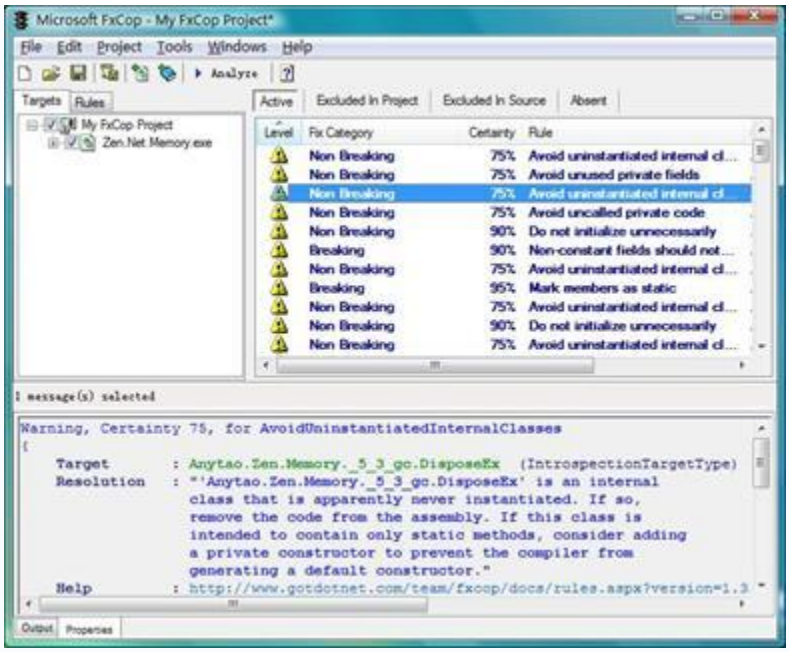


图 5-11 FxCop 代码分析工具

尤其是在性能的检查方面，FxCop 能给我们很多有益的启示，最重要的是 FxCop 简单易用，而且免费，在改善软件质量，重构既有代码时，FxCop 是个不错的选择工具。

5.4.3 结论

性能条款就是系统开发过程中的杠杆，在平衡功能与性能之间做出恰当的选择，本节的 21 条选择策略仅从最普遍意义的选择角度进行了分析，这些条款应该作为开发人员软件设计的参照坐标，并应用于实际的代码编写中。

通读所有条款，你可能会发现本节在一定程度上对本书很多内容做了一次梳理，个中条款以简单的方式呈现，渗透了大师们对于.NET 开发的智慧和经验，作者有幸作为一个归纳梳理的后辈，从中受益匪浅。

6.1 把 new 说透

本文将介绍以下内容：

- 面向对象基本概念
- new 关键字深入浅出
- 对象创建的内存管理

1. 引言

园子里好像没有或者很少把 new 关键字拿出来说的，那我就占个先机吧，呵呵。那么，我们到底有必要将一个关键字拿出来长篇大论吗？看来是个问题。回答的关键是：你真的理解了 new 吗？如果是，那请不要浪费时间，如果不是，那请继续本文的循序之旅。

下面几个 问题可以大概的考察你对 new 的掌握，开篇之前，希望大家做个检验，如果通过了，直接关掉本页即可。如果没有通过，希望本文的阐述能帮你找出答案。

- new 一个 class 对象和 new 一个 struct 或者 enum 有什么不同？
- new 在.NET 中有几个用途，除了创建对象实例，还能做什么？
- new 运算符，可以重载吗？
- 范型中，new 有什么作用？
- new 一个继承下来的方法和 override 一个继承方法有何区别？
- int i 和 int i = new int()有什么不同？

2. 基本概念

一般说来，new 关键字在.NET 中用于以下几个场合，这是 MSDN 的典型解释：

- 作为运算符， 用于创建对象和调用构造函数。

本文的重点内容，本文在下一节来重点考虑。

- 作为修饰符，用于向基类成员隐藏继承成员。

作为修饰符，基本的规则可以总结为：实现派生类中隐藏方法，则基类方法必须定义为 virtual；new 作为修饰符，实现隐藏基类成员时，不可和 override 共存，原因是这两者语义相斥：new 用于实现创建一个新成员，同时隐藏基类的同名成员；而 override 用于实现对基类成员的扩展。

另外，如果在子类中隐藏了基类的数据成员，那么对基类原数据成员的访问，可以通过 base 修饰符来完成。

例如：

new 作为修饰符

- 作为约束，用于在泛型声明中约束可能用作类型参数的参数的类型。

MSDN 中的定义是：**new** 约束指定泛型类声明中的任何类型参数都必须有公共的无参数构造函数。当泛型类创建类型的新实例时，将此约束应用于类型参数。

注意：new 作为约束和其他约束共存时，必须在最后指定。

其定义方式为：

```
class Genericer<T> where T : new()
{
   public T GetItem()
   {
      return new T();
   }
}
```

实现方式为：

```
class MyCls
{
   private string _name;

   public string Name
   {
      get { return _name; }
      set { _name = value; }
```

```
    }

    public MyCls()
    {
        _name = "Emma";
    }
}
```

```
class MyGenericTester
{
    public static void Main(string[] args)
    {
        Genericer<MyCls> MyGen = new Genericer<MyCls>();
        Console.WriteLine(MyGen.GetItem().Name);
    }
}
```

- 使用 new 实现多态。 这不是我熟悉的话题，详细的内容可以参见 《多态与 new [C#]》，这里有较详细的论述。

3. 深入浅出

作为修饰符和约束的情况，不是很难理解的话题，正如我们看到本文开篇提出的问题，也大多集中在 new 作为运算符的情况，因此我们研究的重点就是揭开 new 作为运算符的前世今生。

Jeffrey Richter 在其著作中，极力推荐读者使用 ILDASM 工具查看 IL 语言细节，从而提高对.NET 的深入探究，在我认为这真是一条不错的建议，也给了自己很多提高的空间挖掘。因此，以下是本人的一点建议，我将在后续的系列中，关于学习方法论的讨论中深入探讨，这里只是顺便小议，希望有益于大家。
1 不断的学习代码；
2 经常看看 IL 语言的运行细节，对于提供.NET 的认识非常有效。

文归正题，new 运算符用于返回一个引用，指向系统分配的托管堆的内存地址。因此，在此我们以 Reflector 工具，来了解以下 new 操作符执行的背后，隐藏着什么玄机。

首先我们实现一段最简单的代码，然后分析其元数据的实现细节，来探求 new 在创建对象时到做了什么？

new 作为运算符

使用 Reflector 工具反编译产生的 IL 代码如下为：

IL 元数据分析

从而可以得出以下结论：

- new 一个 class 时，new 完成了以下两个方面的内容：一是调用 newobj 命令来为实例在托管堆中分配内存；二是调用构造函数来实现对象初始化。
- new 一个 struct 时，new 运算符用于调用其带构造函数，完成实例的初始化。
- new 一个 int 时，new 运算符用于初始化其值为 0。
- 另外必须清楚，值类型和引用类型在分配内存时是不同的，值类型分配于线程的堆栈（stack）上，并变量本身就保存其实值，因此也不受 GC 的控制，；而引用类型变量，包含了指向托管堆的引用，内存分配于托管堆（managed heap）上，内存收集由 GC 完成。

另外还有以下规则要多加注意：

- new 运算符不可重载。
- new 分配内存失败，将引发 OutOfMemoryException 异常。

对于基本类型来说，使用 new 操作符来进行初始化的好处是，某些构造函数可以完成更优越的初始化操作，而避免了不高明的选择，例如：

```
string str = new string('*', 100);


string str = new string(new char[] {'a', 'b', 'c'});
```

而不是

```
string str = "**************************************";
```

4. 结论

我能说的就这么多了，至于透了没透，作者的能量也就这么多了。希望园子的大牛们常来扔块砖头，对我也是一种莫大的促进。但是作为基本的原理和应用，我想对大部分的需求是满足了。希望这种力求深入浅出的介绍，能给你分享 new 关键字和其本质的来龙去脉能有所帮助。

言归正传，开篇的几个题目，不知读者是否有了各自的答案，我们不妨畅所欲言，做更深入的讨论，以便揭开其真实的面纱。

参考文献

（USA）Stanley B.Lippman, C# Primer

（USA）David Chappell Understanding .NET

广而告之

[预告]

另外鉴于前几个主题的讨论中，不管是类型、关键字等都涉及到引用类型和值类型的话题，我将于近期发表相关内容的探讨，同时还有其他的关键字值得研究，这是本系列近期动向，给自己做个广告。祝各位愉快。

[声明]

本文的关键字 new 指的是 C#中的关键字概念，并非一般意义上的.NET CRL 范畴，之所以将这个主题加入本系列，是基于在.NET 体系下开发的我们，何言能逃得过基本语言的只是要点。所以大可不必追究什么是.NET，什么是 C#的话题，希望大家理清概念，有的放肆。

6.2 base 和 this

本文将介绍以下内容：

- 面向对象基本概念
- base 关键字深入浅出
- this 关键字深入浅出

1. 引言

new 关键字引起了大家的不少关注，尤其感谢 Anders Liu 的补充，让我感觉博客园赋予的交流平台真的无所不在。所以，我们就有必要继续这个话题，把我认为最值得关注的关键字开展下去，本文的重点是访问关键字（Access Keywords）：base 和 this。虽然访问关键字不是很难理解的话题，我们还是有可以深入讨论的地方来理清思路。还是老办法，我的问题先列出来，您是否做好了准备。

- 是否可以在静态方法中使用 base 和 this，为什么？
- base 常用于哪些方面？this 常用于哪些方面？
- 可以 base 访问基类的一切成员吗？
- 如果有三层或者更多继承，那么最下级派生类的 base 指向那一层呢？例如.NET 体系中，如果以 base 访问，则应该是直接父类实例呢，还是最高层类实例呢？
- 以 base 和 this 应用于构造函数时，继承类对象实例化的执行顺序如何？

2. 基本概念

base 和 this 在 C#中被归于访问关键字，顾名思义，就是用于实现继承机制的访问操作，来满足对对象成员的访问，从而为多态机制提供更加灵活的处理方式。

2.1 base 关键字

其用于在派生类中实现对基类公有或者受保护成员的访问，但是只局限在构造函数、实例方法和实例属性访问器中，MSDN 中小结的具体功能包括：

- 调用基类上已被其他方法重写的方法。
- 指定创建派生类实例时应调用的基类构造函数。

2.2 this 关键字

其用于引用类的当前实例，也包括继承而来的方法，通常可以隐藏 this，MSDN 中的小结功能主要包括：

- 限定被相似的名称隐藏的成员
- 将对象作为参数传递到其他方法
- 声明索引器

3. 深入浅出

3.1 示例为上

下面以一个小示例来综合的说明，base 和 this 在访问操作中的应用，从而对其有个概要了解，更详细的规则和深入我们接着阐述。本示例没有完全的设计概念，主要用来阐述 base 和 this 关键字的使用要点和难点阐述，具体的如下：

base 和 this 示例

3.2 示例说明

上面的示例基本包括了 base 和 this 使用的所有基本功能演示，具体的说明可以从注释中得到解释，下面的说明是对注释的进一步阐述和补充，来说明在应用方面的几个要点：

- base 常用于，在派生类对象初始化时和基类进行通信。
- base 可以访问基类的公有成员和受保护成员，私有成员是不可访问的。
- this 指代类对象本身，用于访问本类的所有常量、字段、属性和方法成员，而且不管访问元素是任何访问级别。因为，this 仅仅局限于对象内部，对象外部是无法看到的，这就是 this 的基本思想。另外，静态成员不是对象的一部分，因此不能在静态方法中引用 this。
- 在多层继承中，base 可以指向的父类的方法有两种情况：一是有重载存在的情况下，base 将指向直接继承的父类成员的方法，例如 Audi 类中的 ShowResult 方法中，使用 base 访问的将是 Car.ShowResult()方法，而不能访问 Vehicle.ShowResult()方法；而是没有重载存在的情况下，base 可以指向任何上级父类的公有或者受保护方法，例如 Audi 类中，可以使用 base 访问基类 Vehicle.Run()方法。这些我们可以使用 ILDasm.exe，从 IL 代码中得到答案。

```
.method public hidebysig virtual instance void
        ShowResult() cil managed
{
 // 代码大小       27 (0x1b)
 .maxstack  8
 IL_0000:  nop
 IL_0001:  ldarg.0
 //base 调用父类成员
 IL_0002:  call       instance void Anytao.net.My_Must_net.Car::ShowResult()
 IL_0007:  nop
```

```
  IL_0008:  ldarg.0
  //base 调用父类成员，因为没有实现 Car.Run(),所以指向更高级父类
  IL_0009:  call       instance void Anytao.net.My_Must_net.Vehicle::Run()
  IL_000e:  nop
  IL_000f:  ldstr      "It's audi's result."
  IL_0014:  call       void [mscorlib]System.Console::WriteLine(string)
  IL_0019:  nop
  IL_001a:  ret
} // end of method Audi::ShowResult
```

3.3 深入剖析

如果有三次或者更多继承，那么最下级派生类的 base 指向那一层呢？例如.NET 体系中，如果以 base 访问，则应该是直接父类实例呢，还是最高层类实例呢？

首先我们有必要了解类创建过程中的实例化顺序，才能进一步了解 base 机制的详细执行过程。一般来说，实例化过程首先要先实例化其基类，并且依此类推，一直到实例化 System.Object 为止。因此，类实例化，总是从调用 System.Object.Object()开始。因此示例中的类 Audi 的实例化过程大概可以小结为以下顺序执行，详细可以参考示例代码分析。

- 执行 System.Object.Object();
- 执行 Vehicle.Vehicle(string name, int speed);
- 执行 Car.Car();
- 执行 Car.Car(string name, int speed);
- 执行 Audi.Audi();
- 执行 Audi.Audi(string name, int speed)。

我们在充分了解其实例化顺序的基础上就可以顺利的把握 base 和 this 在作用于构造函数时的执行情况，并进一步了解其基本功能细节。

下面更重要的分析则是，以 ILDASM.exe 工具为基础来分析 IL 反编译代码，以便更深层次的了解执行在 base 和 this 背后的应用实质，只有这样我们才能说对技术有了基本的剖析。

Main 方法的执行情况为：

⊞ ⊟IL 分析 base 和 this 执行

因此，对重写父类方法，最终指向了最高级父类的方法成员。

4. 通用规则

- 尽量少用或者不用 base 和 this。除了决议子类的名称冲突和在一个构造函数中调用其他的构造函数之外，base 和 this 的使用容易引起不必要的结果。
- 在静态成员中使用 base 和 this 都是不允许的。原因是，base 和 this 访问的都是类的实例，也就是对象，而静态成员只能由类来访问，不能由对象来访问。
- base 是为了实现多态而设计的。
- 使用 this 或 base 关键字只能指定一个构造函数，也就是说不可同时将 this 和 base 作用在一个构造函数上。
- 简单的来说，base 用于在派生类中访问重写的基类成员；而 this 用于访问本类的成员，当然也包括继承而来公有和保护成员。
- 除了 base，访问基类成员的另外一种方式是：显示的类型转换来实现。只是该方法不能为静态方法。

5. 结论

base 和 this 关键字，不是特别难于理解的内容，本文之所以将其作为系列的主题，除了对其应用规则做以小结之外，更重要的是在关注其执行细节的基础上，对语言背景建立更清晰的把握和分析，这些才是学习和技术应用的根本所在，也是.NET 技术框架中本质诉求。对学习者来说，只有从本质上来把握概念，才能在变化非凡的应用中，一眼找到答案。

言归正传，开篇的几个题目，不知读者是否有了各自的答案，我们不妨畅所欲言，做更深入的讨论，以便揭开其真实的面纱。

参考文献

（USA）Stanley B.Lippman, C# Primer

（USA）David Chappell， Understanding .NET

（Cnblog）Bear-Study-Hard，C#学习笔记（二）：构造函数的执行序列

广而告之

[预告]

另外鉴于前几个主题的讨论中，不管是类型、关键字等都涉及到引用类型和值类型的话题，我将于近期发表相关内容的探讨，主要包括 3 个方面的内容，这是本系列近期动向，给自己做个广告。祝各位愉快。

[声明]

本文的关键字指的是 C#中的关键字概念，并非一般意义上的.NET CRL 范畴，之所以将这个主题加入本系列，是基于在.NET 体系下开发的我们，何言能逃得过基本语言的只是要点。所以大可不必追究什么是.NET，什么是 C#的话题，希望大家理清概念，有的放肆。

6.3 深入浅出关键字---using 全接触

本文将介绍以下内容：

- using 指令的多种用法
- using 语句在 Dispose 模式中的应用

1. 引言

在.NET 大家庭中，有不少的关键字承担了多种角色，例如 new 关键字就身兼数职，除了能够创建对象，在继承体系中隐藏基类成员，还在泛型声明中约束可能用作类型参数的参数，在[第五回：深入浅出关键字---把 new 说透]我们对此都有详细的论述。本文，将把目光转移到另外一个身兼数职的明星关键字，这就是 using 关键字，在详细讨论 using 的多重身份的基础上来了解.NET 在语言机制上的简便与深邃。

那么，using 的多重身份都体现在哪些方面呢，我们先一睹为快吧：

- 引入命名空间
- 创建别名
- 强制资源清理

下面，本文将从这几个角度来阐述 using 的多彩应用。

2. 引入命名空间

using 作为引入命名空间指令的用法规则为：

```
using Namespace;
```

在.NET 程序中，最常见的代码莫过于在程序文件的开头引入 System 命名空间，其原因在于 System 命名空间中封装了很多最基本最常用的操作，下面的代码对我们来说最为熟悉不过：

```
using System;
```

这样，我们在程序中就可以直接使用命名空间中的类型，而不必指定详细的类型名称。using 指令可以访问嵌套命名空间。

关于：命名空间

命名空间是.NET 程序在逻辑上的组织结构，而并非实际的物理结构，是一种避免类名冲突的方法，用于将不同的数据类型组合划分的方式。例如，在.NET 中很多的基本类型都位于 System 命名空间，数据操作类型位于 System.Data 命名空间，

误区：

- using 类似于 Java 语言的 import 指令，都是引入命名空间（Java 中称作包）这种逻辑结构；而不同于 C 语言中的#include 指令，用于引入实际的类库，
- using 引入命名空间，并不等于编译器编译时加载该命名空间所在的程序集，程序集的加载决定于程序中对该程序集是否存在调用操作，如果代码中不存在任何调用

操作则编译器将不会加载 using 引入命名空间所在程序集。因此，在源文件开头，引入多个命名空间，并非加载多个程序集，不会造成“过度引用”的弊端。

3. 创建别名

using 为命名空间创建别名的用法规则为：

```
using alias = namespace | type;
```

其中 namespace 表示创建命名空间的别名；而 type 表示创建类型别名。例如，在.NET Office 应用中，常常会引入 Microsoft.Office.Interop.Word.dll 程序集，在引入命名空间时为了避免繁琐的类型输入，我们通常为其创建别名如下：

```
using MSWord = Microsoft.Office.Interop.Word;
```

这样，就可以在程序中以 MSWord 来代替 Microsoft.Office.Interop.Word 前缀，如果要创建 Application 对象，则可以是这样，

```
private static MSWord.Application ooo = new MSWord.Application();
```

同样，也可以创建类型的别名，用法为：

```
using MyConsole = System.Console;

class UsingEx

{

   public static void Main()

   {

      MyConsole.WriteLine("应用了类的别名。");

   }

}
```

而创建别名的另一个重要的原因在于同一 cs 文件中引入的不同命名空间中包括了相同名称的类型，为了避免出现名称冲突可以通过设定别名来解决，例如：

```
namespace Anytao.Essential.Demos
{
    using BoyPlayer = Boyspace.Player;
    using GirlPlayer = Girlspace.Player;
    class UsingEx
    {
        public static void Main()
        {
            BoyPlayer.Play();
            GirlPlayer.Play();
        }
    }
}
```

2007 @ Anytao.com

```
namespace Boyspace
{
    public class Player
    {
        public static void Play()
        {
            System.Console.WriteLine("Boys play football.");
        }
    }
}
namespace Girlspace
{
    public class Player
```

```
    {
        public static void Play()
        {
            System.Console.WriteLine("Girls play violin.");
        }
    }
}
```

以 using 创建别名，有效的解决了这种可能的命名冲突，尽管我们可以通过类型全名称来加以区分，但是这显然不是最佳的解决方案，using 使得这一问题迎刃而解，不费丝毫功夫，同时在编码规范上看来也更加的符合编码要求。

4. 强制资源清理

4.1 由来

要理解清楚使用 using 语句强制清理资源，就首先从了解 Dispose 模式说起，而要了解 Dispose 模式，则应首先了解.NET 的垃圾回收机制。这些显然不是本文所能完成的宏论，我们只需要首先明确的是.NET 提供了 Dispose 模式来实现显式释放和关闭对象的能力。

Dispose 模式

Dispose 模式是.NET 提供的一种显式清理对象资源的约定方式，用于在.NET 中释放对象封装的非托管资源。因为非托管资源不受 GC 控制，对象必须调用自己的 Dispose()方法来释放，这就是所谓的 Dispose 模式。从概念角度来看，Dispose 模式就是一种强制资源清理所要遵守的约定；从实现角度来看，Dispose 模式就是让要一个类型实现 IDisposable 接口，从而使得该类型提供一个公有的 Dispose 方法。

本文不再讨论如何让一个类型实现 Dispose 模式来提供显示清理非托管资源的方式，而将注意集中在如何以 using 语句来简便的应用这种实现了 Dispose 模式的类型的资源清理方式。我们在内存管理与垃圾回收章节将有详细的讨论。

using 语句提供了强制清理对象资源的便捷操作方式，允许指定何时释放对象的资源，其典型应用为：

```
        using (Font f = new Font("Verdana", 12, FontStyle.Regular))
        {
           //执行文本绘制操作
           Graphics g = e.Graphics;
           Rectangle rect = new Rectangle(10, 10, 200, 200);
           g.DrawString("Try finally dispose font.", f, Brushes.Black, rect);
        }//运行结束，释放 f 对象资源
```

在上述典型应用中，using 语句在结束时会自动调用欲被清除对象的 Dispose()方法。因此，该 Font 对象必须实现 IDispose 接口，才能使用 using 语句强制对象清理资源。我们查看其类型定义可知：

```
public sealed class Font : MarshalByRefObject, ICloneable, ISerializable, IDisposable
```

Font 类型的确实现了 IDisposeable 接口，也就具有了显示回收资源的能力。然而，我们并未从上述代码中，看出任何使用 Dispose 方法的蛛丝马迹，这正式 using 语句带来的简便之处，其实质究竟怎样呢？

4.2 实质

要想了解 using 语句的执行本质，了解编译器在背后做了哪些手脚，就必须回归到 IL 代码中来揭密才行：

```
.method public hidebysig static void Main() cil managed

{

.entrypoint

// 代码大小       40 (0x28)

.maxstack 4

.locals init ([0] class [System.Drawing]System.Drawing.Font f,

         [1] bool CS$4$0000)

IL_0000: nop

IL_0001: ldstr      "Verdana"

IL_0006: ldc.r4     12.

IL_000b: ldc.i4.0

IL_000c: newobj     instance void [System.Drawing]System.Drawing.Font::.ctor(string,float32,

                                                valuetype [System.Drawing]System.Drawing.FontStyle)

IL_0011: stloc.0

.try

{

      ......部分省略.....

} // end .try

finally

{

      ......部分省略.....

  IL_001f: callvirt   instance void [mscorlib]System.IDisposable::Dispose()
```

```
  IL_0024: nop

  IL_0025: endfinally

} // end handler

IL_0026: nop

IL_0027: ret

} // end of method UsingDispose::Main
```

显然，编译器在自动将 using 生成为 try-finally 语句，并在 finally 块中调用对象的 Dispose 方法，来清理资源。

在.NET 规范中，微软建议开放人员在调用一个类型的 Dispose()或者 Close()方法时，将其放在异常处理的 finally 块中。根据上面的分析我们可知，using 语句正是隐式的调用了类型的 Dispose 方法，因此以下的代码和上面的示例是完全等效的：

```
        Font f2 = new Font("Arial", 10, FontStyle.Bold);

        try

        {

              //执行文本绘制操作

           Graphics g = new Graphics();

           Rectangle rect = new Rectangle(10, 10, 200, 200);

           g.DrawString("Try finally dispose font.", f2, Brushes.Black, rect);

        }

        finally

        {

           if (f2 != null)
```

```
            ((IDisposable)f2).Dispose();

        }
```

4.3 规则

- using 只能用于实现了 IDisposable 接口的类型，禁止为不支持 IDisposable 接口的类型使用 using 语句，否则会出现编译时错误；
- using 语句适用于清理单个非托管资源的情况，而多个非托管对象的清理最好以 try-finnaly 来实现，因为嵌套的 using 语句可能存在隐藏的 Bug。内层 using 块引发异常时，将不能释放外层 using 块的对象资源。
- using 语句支持初始化多个变量，但前提是这些变量的类型必须相同，例如：

```
        using(Pen p1 = new Pen(Brushes.Black), p2 = new Pen(Brushes.Blue))

        {

          //

        }
```

否则，编译将不可通过。不过，还是有变通的办法来解决这一问题，原因就是应用 using 语句的类型必然实现了 IDisposable 接口，那么就可以以下面的方式来完成初始化操作，

```
        using (IDisposable font = new Font("Verdana", 12, FontStyle.Regular), pen = new Pen(Brushes.Bla
ck))

        {

          float size = (font as Font).Size;

          Brush brush = (pen as Pen).Brush;

        }
```

另一种办法就是以使用 try-finally 来完成，不管初始化的对象类型是否一致。

- Dispose 方法用于清理对象封装的非托管资源，而不是释放对象的内存，对象的内存依然由垃圾回收器控制。
- 程序在达到 using 语句末尾时退出 using 块，而如果到达语句末尾之前引入异常则有可能提前退出。
- using 中初始化的对象，可以在 using 语句之前声明，例如：

```
Font f3 = new Font("Verdana", 9, FontStyle.Regular);
using (f3)
{
   //执行文本绘制操作
}
```

5. 结论

一个简单的关键字，多种不同的应用场合。本文从比较全面的角度，诠释了 using 关键字在.NET 中的多种用法，值得指出的是这种用法并非实现于.NET 的所有高级语言，本文的情况主要局限在 C#中。

第 7 章 巅峰对决——走出误区

7.2 后来居上：class 和 struct

本文将介绍以下内容：

- 面向对象基本概念
- 类和结构体简介
- 引用类型和值类型区别

1. 引言

提起 class 和 struct，我们首先的感觉是语法几乎相同，待遇却翻天复地。历史将接力棒由面向过程编程传到面向对象编程，class 和 struct 也背负着各自的命运前行。在我认为，struct 英雄迟暮，class 天下独行，最本质的区别是 class 是引用类型，而 struct 是值类型，它们在内存中的分配情况有所区别。由此产生的一系列差异性，本文将做以全面讨论。

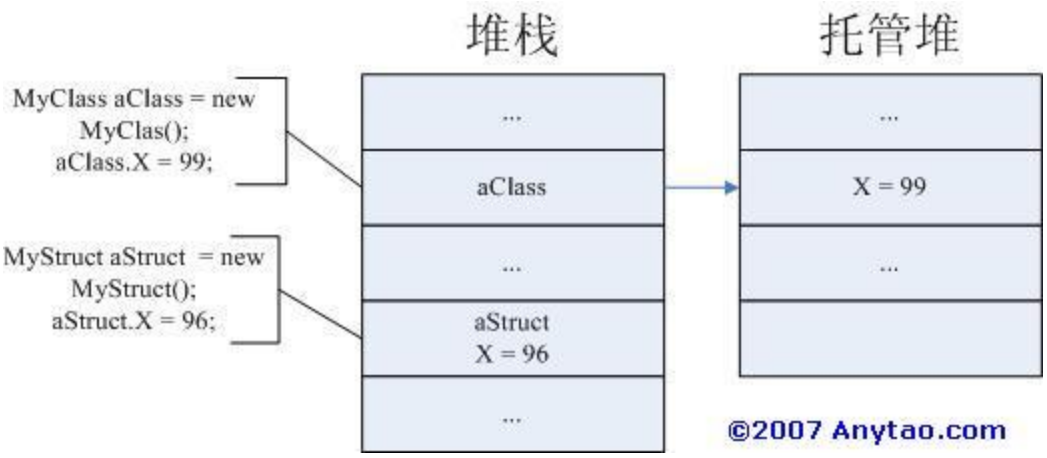
2. 基本概念

2.1. 什么是 class?

class（类）是面向对象编程的基本概念，是一种自定义数据结构类型，通常包含字段、属性、方法、属性、构造函数、索引器、操作符等。因为是基本的概念，所以不必在此详细描述，读者可以查询相关概念了解。我们重点强调的是.NET 中，所有的类都最终继承自 System.Object 类，因此是一种引用类型，也就是说，new 一个类的实例时，对象保存了该实例实际数据的引用地址，而对象的值保存在托管堆（managed heap）中。

2.2. 什么是 struct?

struct（结构）是一种值类型，用于将一组相关的信息变量组织为一个单一的变量实体 。所有的结构都继承自 System.ValueType 类，因此是一种值类型，也就是说，struct 实例分配在线程的堆栈（stack）上，它本身存储了值，而不包含指向该值的指针。所以在使用 struct 时，我们可以将其当作 int、char 这样的基本类型类对待。



3. 相同点，不同点

相同点：语法类似。

不同点：

- class 是引用类型，继承自 System.Object 类；struct 是值类型，继承自 System.ValueType 类，因此不具多态性。但是注意，System.ValueType 是个引用类型。

- 从职能观点来看，class 表现为行为；而 struct 常用于存储数据。
- class 支持继承，可以继承自类和接口；而 struct 没有继承性，struct 不能从 class 继承，也不能作为 class 的基类，但 struct 支持接口继承（记得吗，《第二回：对抽象编程：接口和抽象类》也做过讨论）
- class 可以声明无参构造函数，可以声明析构函数；而 struct 只能声明带参数构造函数，且不能声明析构函数。因此，struct 没有自定义的默认无参构造函数，默认无参构造器只是简单地把所有值初始化为它们的 0 等价值
- 实例化时，class 要使用 new 关键字；而 struct 可以不使用 new 关键字，如果不以 new 来实例化 struct，则其所有的字段将处于未分配状态，直到所有字段完成初始化，否则引用未赋值的字段会导致编译错误。
- class 可以实抽象类（abstract），可以声明抽象函数；而 struct 为抽象，也不能声明抽象函数。
- class 可以声明 protected 成员、virtual 成员、sealed 成员和 override 成员；而 struct 不可以，但是值得注意的是，struct 可以重载 System.Object 的 3 个虚方法，Equals()、ToString()和 GetHashTable()。
- class 的对象复制分为浅拷贝和深拷贝（该主题我们在本系列以后的主题中将重点讲述，本文不作详述），必须经过特别的方法来完成复制；而 struct 创建的对象复制简单，可以直接以等号连接即可。
- class 实例由垃圾回收机制来保证内存的回收处理；而 struct 变量使用完后立即自动解除内存分配。
- 作为参数传递时，class 变量是以按址方式传递；而 struct 变量是以按值方式传递的。

我们可以简单的理解，class 是一个可以动的机器，有行为，有多态，有继承；而 struct 就是个零件箱，组合了不同结构的零件。其实，class 和 struct 最本质的区别就在于 class 是引用类型，内存分配于托管堆；而 struct 是值类型，内存分配于线程的堆栈上。由此差异，导致了上述所有的不同点，所以只有深刻的理解内存分配的相关内容，才能更好的驾驭。本系列将再以后的内容中，将引用类型和值类型做以深入的比较和探讨，敬请关注。当然正如本文标题描述的一样，使用 class 基本可以替代 struct 的任何场合，class 后来居上。虽然在某些方面 struct 有性能方面的优势，但是在面向对象编程里，基本是 class 横行的天下。

那么，有人不免会提出，既然 class 几乎可以完全替代 struct 来实现所有的功能，那么 struct 还有存在的必要吗？答案是，至少在以下情况下，鉴于性能上的考虑，我们应该考虑使用 struct 来代替 class：

- 实现一个主要用于存储数据的结构时，可以考虑 struct。
- struct 变量占有堆栈的空间，因此只适用于数据量相对小的场合。
- 结构数组具有更高的效率。
- 提供某些和非托管代码通信的兼容性。

所有这些是 struct 有一席之地的理由，当然也许还有其他的更多说法，只是我不知道罢了:-)

4. 经典示例

4.1 小菜一碟

下面以示例为说明，来阐述本文的基本规则，详细见注释内容。
（1）定义接口

```
interface IPerson
{
   void GetSex();
}
```

（2）定义类

```
public class Person
{
   public Person()
   {
   }

   public Person(string name, int age)
   {
      _name = name;
      _age = age;
   }
```

```
    private string _name;

    public string Name
    {
      get { return _name; }
      set { _name = value; }
    }

    private int _age;

    public int Age
    {
      get { return _age; }
      set { _age = value; }
    }
 }
```

（3）定义结构

```
 //可以继承自接口，不可继承类或结构
  struct Family: IPerson
  {
    public string name;
    public int age;
    public bool sex;
    public string country;
    public Person person;

    //不可以包含显式的无参构造函数和析构函数
    public Family(string name, int age, bool sex, string country, Person person)
    {
       this.name = name;
       this.age = age;
       this.sex = sex;
       this.country = country;
       this.person = person;
    }
```

```
    //不可以实现 protected、virtual、sealed 和 override 成员
    public void GetSex()
    {
        if (sex)
            Console.WriteLine(person.Name + " is a boy.");
        else
            Console.WriteLine(person.Name + " is a girl.");
    }

    public void ShowPerson()
    {
        Console.WriteLine("This is {0} from {1}", new Person(name, 22).Name, country);
    }

    //可以重载 ToString 虚方法
    public override string ToString()
    {
        return String.Format("{0} is {1}, {2} from {3}", person.Name, age, sex ? "Boy
" : "Girl", country);
    }
  }
```

（4）测试结构和类

```
class MyTest
{
    static void Main(string[] args)
    {
        //不使用new来生成结构，其内部成员将初始化为0
        Family newFamily;
        newFamily.name = "Anytao Family";
        newFamily.sex = true;
        Console.WriteLine(newFamily.name);
        //以new来生成结构，调用带参数构造器
        Family myFamily = new Family("Anytao Family", 26, true, "China", new Person("Anyta
        Person person = new Person();
        person.Name = "Anytao";
        //按值传递参数
        ShowFamily(myFamily);
        //按引用传递参数
        ShowPerson(person);
        myFamily.GetSex();
        myFamily.ShowPerson();
        Console.WriteLine("I'm {0}", myFamily.name);
        Console.WriteLine("I'm {0}", person.Name);
        Console.WriteLine(myFamily.ToString());
    }

    public static void ShowPerson(Person person)
    {
        person.Name = "Emma";
        Console.WriteLine("This is {0}", person.Name);
    }

    public static void ShowFamily(Family family)
    {
        family.name = "Aeor";
        Console.WriteLine("This is {0}", family.name);
    }
}
```

猜猜运行结果如何，可以顺便检查检查对这个概念的认识。

4.2 .NET 研究

在.NET 框架中，System.Drawing 命名空间中的有些元素，如 System.Drawing.Point 就是实现为 struct，而不是 class。其原因也正在于以上介绍的各方面的权衡，大家可以就此研究研究，可以体会更多。另外，还有以 struct 实现的 System.Guid。

5. 结论

对基本概念的把握，是我们进行技术深入探索的必经之路，本系列的主旨也是能够从基本框架中，提供给大家一个通向高级技术的必修课程。本文关于 class 和 struct 的讨论就是如此，在.NET 框架中，关于 class 和 struct 的讨论将涉及到对引用类型和值类型的认识，并且进一步将触角伸向变量内存分配这一高级主题，所以我们有必要来了解其运行机制，把握区别和应用场合，以便在平常的系统设计中把握好对这一概念层次的把握。

另外，请大家就以下问题进行讨论，希望能够更加清晰本文的拓展：

- struct 还主要应用在哪些方面？
- C++和 C#中，关于 struct 的应用又有所不同，这些不同又有哪些区别？

7.3 历史纠葛：特性和属性

本文将介绍以下内容：

- 定制特性的基本概念和用法
- 属性与特性的区别比较
- 反射的简单介绍

1. 引言

attribute 是.NET 框架引入的有一技术亮点，因此我们有必要花点时间来了解本文的内容，走进一个发现 attribute 登堂入室的入口。因为.NET Framework 中使用了大量的定制特性来完成代码约定，[Serializable]、[Flags]、[DllImport]、[AttributeUsage]这些的构造，相信我们都见过吧，那么你是否了解其背后的技术。

提起特性，由于高级语言发展的历史原因，不免让人想起另一个耳熟能详的名字：属性。特性和属性，往往给初学者或者从 C++转移到 C#的人混淆的概念冲击。那么，什么是属性，什么是特性，二者的概念和区别，用法与示例，将在本文做以概括性的总结和比较，希望给你的理解带来收获。另外本文的主题以特性的介绍为主，属性的论述重点突出在二者的比较上，关于属性的更多论述将在另一篇主题中详细讨论，敬请关注。

2. 概念引入

2.1. 什么是特性？

MADN 的定义为：公共语言运行时允许添加类似关键字的描述声明，叫做 attributes, 它对程序中的元素进行标注，如类型、字段、方法和属性等。Attributes 和 Microsoft .NET Framework 文件的元数据保存在一起，可以用来向运行时描述你的代码，或者在程序运行的时候影响应用程序的行为。

我们简单的总结为：定制特性 attribute，本质上是一个类，其为目标元素提供关联附加信息，并在运行期以反射的方式来获取附加信息。具体的特性实现方法，在接下来的讨论中继续深入。

2.2. 什么是属性？

属性是面向对象编程的基本概念，提供了对私有字段的访问封装，在 C#中以 get 和 set 访问器方法实现对可读可写属性的操作，提供了安全和灵活的数据访问封装。关于属性的概念，不是本文的重点，而且相信大部分的技术人员应该对属性有清晰的概念。以下是简单的属性示例：

```
public class MyProperty
{
    //定义字段
    private string _name;
    private int _age;

    //定义属性，实现对_name 字段的封装
    public string Name
    {
        get { return (_name == null) ? string.Empty : _name; }
        set { _name = value; }
    }

    //定义属性，实现对_age 字段的封装
    //加入对字段的范围控制
    public int Age
    {
        get { return _age; }
        set
        {
            if ((value > 0) && (value < 150))
            {
```

```
            _age = value;
        }
        else
        {
            throw new Exception("Not a real age");
        }
    }
  }
}

public class MyTest
{
    public static void Main(string[] args)
    {
        MyProperty myProperty = new MyProperty();
        //触发 set 访问器
        myProperty.Name = "Anytao";
        //触发 get 访问器
        Console.WriteLine(myProperty.Name);
        myProperty.Age = 66;
        Console.WriteLine(myProperty.Age.ToString());
        Console.ReadLine();
    }
}
```

2.3. 区别与比较

通过对概念的澄清和历史的回溯，我们知道特性和属性只是在名称上有过纠葛，在 MSDN 上关于 attribute 的中文解释甚至还是属性，但是我同意更通常的称呼：特性。在功能上和应用上，二者其实没有太多模糊的概念交叉，因此也没有必要来比较其应用的异同点。本文则以特性的概念为重点，来讨论其应用的场合和规则。

我理解的定制特性，就是为目标元素，可以是数据集、模块、类、属性、方法、甚至函数参数等加入附加信息，类似于注释，但是可以在运行期以反射的方式获得。定制特性主要应用在序列化、编译器指令、设计模式等方面。

3. 通用规则

- 定制特性可以应用的目标元素可以为：程序集(assembly)、模块(module)、类型(type)、属性(property)、事件(event)、字段(field)、方法(method)、参数(param)、返回值(return)，应该全了。
- 定制特性以[,]形式展现，放在紧挨着的元素上，多个特性可以应用于同一元素，特性间以逗号隔开，以下表达规则有效：[AttributeUsage][Flags]、[AttributeUsage, Flags]、[Flags, AttibuteUsageAttribute]、[AttributeUsage(), FlagesAttribute()]
- attibute 实例，是在编译期进行初始化，而不是运行期。
- C#允许以指定的前缀来表示特性所应用的目标元素，建议这样来处理，因为显式处理可以消除可能带来的二义性。例如：

```
using System;

namespace Anytao.net
{
   [assembly: MyAttribute(1)]         //应用于程序集
   [moduel: MyAttribute(2)]          //应用于模块
   pubic class Attribute_how2do
   {
      //
   }
}
```

- 定制特性类型，必须直接或者间接的继承自 System.Attribute 类，而且该类型必须有公有构造函数来创建其实例。
- 所有自定义的特性名称都应该有个 Attribute 后缀，这是习惯性约定。
- 定制特性也可以应用在其他定制特性上，这点也很好理解，因为定制特性本身也是一个类，遵守类的公有规则。例如很多时候我们的自定义定制特性会应用 AttributeUsageAttribute 特性，来控制如何应用新定义的特性。

```
[AttributeUsageAttribute(AttributeTarget.All),
AllowMultiple = true,
Inherited = true]
class MyNewAttribute: System.Attribute
{
//
}
```

- 定制特性不会影响应用元素的任何功能，只是约定了该元素具有的特质。
- 所有非抽象特性必须具有 public 访问限制。
- 特性常用于编译器指令，突破#define, #undefine, #if, #endif 的限制，而且更加灵活。
- 定制特性常用于在运行期获得代码注释信息，以附加信息来优化调试。
- 定制特性可以应用在某些设计模式中，如工厂模式。
- 定制特性还常用于位标记，非托管函数标记、方法废弃标记等其他方面。

4. 特性的应用

4.1. 常用特性

常用特性，也就是.NET 已经提供的固有特性，事实上在.NET 框架中已经提供了丰富的固有特性由我们发挥，以下精选出我认为最常用、最典型的固有特性做以简单讨论，当然这只是我的一家之言，亦不足道。我想了解特性，还是从这里做为起点，从.NET 提供的经典开始，或许是一种求知的捷径，希望能给大家以启示。

- **AttributeUsage**

  AttributeUsage 特性用于控制如何应用自定义特性到目标元素。关于 **AttributeTargets**、**AllowMultiple**、**Inherited**、**ValidOn**，请参阅示例说明和其他文档。我们已经做了相当的介绍和示例说明，我们还是在实践中自己体会更多吧。

- **Flags**

  以 Flags 特性来将枚举数值看作位标记，而非单独的数值，例如：

```
enum Animal
{
   Dog     = 0x0001,
   Cat     = 0x0002,
   Duck    = 0x0004,
 Chicken = 0x0008
}
```

因此，以下实现就相当轻松，

```
Animal animals = Animal.Dog | Animal.Cat;
Console.WriteLine(animals.ToString());
```

请猜测结果是什么，答案是:"Dog, Cat"。如果没有 Flags 特别，这里的结果将是"3"。关于位标记，也将在本系列的后续章回中有所交代，在此只做以探讨止步。

- **DllImport**

DllImport 特性，可以让我们调用非托管代码，所以我们可以使用 DllImport 特性引入对 Win32 API 函数的调用，对于习惯了非托管代码的程序员来说，这一特性无疑是救命的稻草。

```
using System;
using System.Runtime.InteropServices;

namespace Anytao.net
{
   class MainClass
   {
     [DllImport("User32.dll")]
     public static extern int MessageBox(int hParent, string msg, string caption, int typ
e);

     static int Main()
     {
       return MessageBox(0, "How to use attribute in .NET", "Anytao_net", 0);
    }
   }
}
```

- **Serializable**

  Serializable 特性表明了应用的元素可以被序列化(serializated)，序列化和反序列化是另一个可以深入讨论的话题，在此我们只是提出概念，深入的研究有待以专门的主题来呈现，限于篇幅，此不赘述。

- **Conditional**

  Conditional 特性，用于条件编译，在调试时使用。注意：Conditional 不可应用于数据成员和属性。

还有其他的重要特性，包括：**Description**、**DefaultValue**、**Category**、**ReadOnly**、**BrowerAble** 等，有时间可以深入研究。

4.2. 自定义特性

既然 attribute，本质上就是一个类，那么我们就可以自定义更特定的 attribute 来满足个性化要求，只要遵守上述的 12 条规则，实现一个自定义特性其实是很容易的，典型的实现方法为：

- 定义特性

```
[AttributeUsage(AttributeTargets.Class |
   AttributeTargets.Method,
   Inherited = true)]
public class TestAttribute : System.Attribute
{
   public TestAttribute(string message)
   {
      Console.WriteLine(message);
   }
   public void RunTest()
   {
      Console.WriteLine("TestAttribute here.");
   }
}
```

- 应用目标元素

```
    [Test("Error Here.")]
    public void CannotRun()
    {
        //
    }
```

- 获取元素附加信息

  如果没有什么机制来在运行期来获取 Attribute 的附加信息，那么 attribute 就没有什么存在的意义。因此，.NET 中以反射机制来实现在运行期获取 attribute 信息，实现方法如下：

```
    public static void Main()
    {
        Tester t = new Tester();
        t.CannotRun();

        Type tp = typeof(Tester);
        MethodInfo mInfo = tp.GetMethod("CannotRun");
        TestAttribute myAtt = (TestAttribute)Attribute.GetCustomAttribute(mInfo, type
of(TestAttribute));
        myAtt.RunTest();
    }
```

5. 经典示例

5.1 小菜一碟

啥也不说了，看注释吧。

```
using System;
using System.Reflection;                              //应用反射技术获得特性信息

namespace Anytao.net
{
   //定制特性也可以应用在其他定制特性上，
   //应用 AttributeUsage，来控制如何应用新定义的特性
   [AttributeUsageAttribute(AttributeTargets.All,      //可应用任何元素
      AllowMultiple = true,                            //允许应用多次
      Inherited = false)]                              //不继承到派生类
   //特性也是一个类，
   //必须继承自 System.Attribute 类，
   //命名规范为："类名"+Attribute。
   public class MyselfAttribute : System.Attribute
   {
      //定义字段
      private string _name;
      private int _age;
      private string _memo;

      //必须定义其构造函数，如果不定义有编译器提供无参默认构造函数
      public MyselfAttribute()
      {
      }
      public MyselfAttribute(string name, int age)
      {
         _name = name;
         _age = age;
      }

      //定义属性
      //显然特性和属性不是一回事儿
      public string Name
      {
         get { return _name == null ? string.Empty : _name; }
      }
```

```
    public int Age
    {
        get { return _age; }
    }

    public string Memo
    {
        get { return _memo; }
        set { _memo = value; }
    }

    //定义方法
    public void ShowName()
    {
        Console.WriteLine("Hello, {0}", _name == null ? "world." : _name);
    }
}

//应用自定义特性
//可以以 Myself 或者 MyselfAttribute 作为特性名
//可以给属性 Memo 赋值
[Myself("Emma", 25, Memo = "Emma is my good girl.")]
public class Mytest
{
    public void SayHello()
    {
        Console.WriteLine("Hello, my.net world.");
    }
}

public class Myrun
{
    public static void Main(string[] args)
    {
        //如何以反射确定特性信息
```

```
            Type tp = typeof(Mytest);
            MemberInfo info = tp;
            MyselfAttribute myAttribute =
                (MyselfAttribute)Attribute.GetCustomAttribute(info, typeof(MyselfAttribute));
            if (myAttribute != null)
            {
                //嘿嘿，在运行时查看注释内容，是不是很爽
                Console.WriteLine("Name: {0}", myAttribute.Name);
                Console.WriteLine("Age: {0}", myAttribute.Age);
                Console.WriteLine("Memo of {0} is {1}", myAttribute.Name, myAttribute.Memo);
                myAttribute.ShowName();
            }

            //多点反射
            object obj = Activator.CreateInstance(typeof(Mytest));

            MethodInfo mi = tp.GetMethod("SayHello");
            mi.Invoke(obj, null);
            Console.ReadLine();
        }
    }
}
```

啥也别想了，自己做一下试试。

5.2 他山之石

- MSDN 认为，特性 (Attribute) 描述如何将数据序列化，指定用于强制安全性的特性，并限制实时 (JIT) 编译器的优化，从而使代码易于调试。属性 (Attribute) 还可以记录文件名或代码作者，或在窗体开发阶段控制控件和成员的可见性。
- dudu Boss 收藏的系列文章《Attribute 在.net 编程中的应用》，给你应用方面的启示会很多，值得研究。
- 亚历山大同志 的系列文章《手把手教你写 ORM（六）》中，也有很好的诠释。
- idior 的文章《Remoting 基本原理及其扩展机制》也有收获，因此补充。

6. 结论

Attribute 是.NET 引入的一大特色技术，但在博客园中讨论的不是很多，所以拿出自己的体会来分享，希望就这一技术要点进行一番登堂入室的引导。更深层次的应用，例如序列化、程序安全性、设计模式多方面都可以挖掘出闪耀的金子，这就是.NET 在技术领域带来的百变魅力吧。希望大家畅所欲言，来完善和补充作者在这方面的不全面和认知上的不深入，那将是作者最大的鼓励和动力。

7.4：对抽象编程：接口和抽象类

本文将介绍以下内容：

- 面向对象思想：多态
- 接口
- 抽象类

1. 引言

在我之前的一篇 post《抽象类和接口的谁是谁非》中，和同事管伟的讨论，得到很多朋友的关注，因为是不成体系的论道，所以给大家了解造成不便，同时关于这个主题的系统性理论，我认为也有必要做以总结，因此才有了本篇的新鲜出炉。同时，我将把上贴中的问题顺便也在此做以交代。

2. 概念引入

- 什么是接口？

接口是包含一组虚方法的抽象类型，其中每一种方法都有其名称、参数和返回值。接口方法不能包含任何实现，CLR 允许接口可以包含事件、属性、索引器、静态方法、静态字段、静态构造函数以及常数。但是注意：C#中不能包含任何静态成员。一个类可以实现多个接口，当一个类继承某个接口时，它不仅要实现该接口定义的所有方法，还要实现该接口从其他接口中继承的所有方法。

定义方法为：

```
public interface System.IComparable
{
   int CompareTo(object o);
}
```

```
public class TestCls: IComparable
{
    public TestCls()
    {
    }

    private int _value;
    public int Value
    {
        get { return _value; }
        set { _value = value; }
    }

    public int CompareTo(object o)
    {

        //使用 as 模式进行转型判断
        TestCls aCls = o as TestCls;
        if (aCls != null)
        {

        //实现抽象方法
        return _value.CompareTo(aCls._value);
        }
    }
}
```

- 什么是抽象类？

抽象类提供多个派生类共享基类的公共定义，它既可以提供抽象方法，也可以提供非抽象方法。抽象类不能实例化，必须通过继承由派生类实现其抽象方法，因此对抽象类不能使用 new 关键字，也不能被密封。如果派生类没有实现所有的抽象方法，则该派生类也必须声明为抽象类。另外，实现抽象方法由 overriding 方法来实现。

定义方法为：

```
/// <summary>
/// 定义抽象类
/// </summary>
abstract public class Animal
{
   //定义静态字段
   protected int _id;

   //定义属性
   public abstract int Id
   {
      get;
      set;
   }

   //定义方法
   public abstract void Eat();

   //定义索引器
   public string this[int i]
   {
      get;
      set;
   }
}

/// <summary>
/// 实现抽象类
/// </summary>
public class Dog: Animal
{
   public override int Id
   {
      get {return _id;}
      set {_id = value;}
```

```
    }

    public override void Eat()
    {
        Console.Write("Dog Eats.")
    }
}
```

3. 相同点和不同点

3.1 相同点

- 都不能被直接实例化，都可以通过继承实现其抽象方法。
- 都是面向抽象编程的技术基础，实现了诸多的设计模式。

3.2 不同点

- 接口支持多继承；抽象类不能实现多继承。
- 接口只能定义抽象规则；抽象类既可以定义规则，还可能提供已实现的成员。
- 接口是一组行为规范；抽象类是一个不完全的类，着重族的概念。
- 接口可以用于支持回调；抽象类不能实现回调，因为继承不支持。
- 接口只包含方法、属性、索引器、事件的签名，但不能定义字段和包含实现的方法；抽象类可以定义字段、属性、包含有实现的方法。
- 接口可以作用于值类型和引用类型；抽象类只能作用于引用类型。例如，Struct 就可以继承接口，而不能继承类。

通过相同与不同的比较，我们只能说接口和抽象类，各有所长，但无优略。在实际的编程实践中，我们要视具体情况来酌情量才，但是以下的经验和积累，或许能给大家一些启示，除了我的一些积累之外，很多都来源于经典，我相信经得起考验。所以在规则与场合中，我们学习这些经典，最重要的是学以致用，当然我将以一家之言博大家之笑，看官请继续。

3.3 规则与场合

- 请记住，面向对象思想的一个最重要的原则就是：面向接口编程。

- 借助接口和抽象类，23 个设计模式中的很多思想被巧妙的实现了，我认为其精髓简单说来就是：面向抽象编程。
- 抽象类应主要用于关系密切的对象，而接口最适合为不相关的类提供通用功能。
- 接口着重于 CAN-DO 关系类型，而抽象类则偏重于 IS-A 式的关系；
- 接口多定义对象的行为；抽象类多定义对象的属性；
- 接口定义可以使用 public、protected、internal 和 private 修饰符，但是几乎所有的接口都定义为 public，原因就不必多说了。
- “接口不变”，是应该考虑的重要因素。所以，在由接口增加扩展时，应该增加新的接口，而不能更改现有接口。
- 尽量将接口设计成功能单一的功能块，以.NET Framework 为例，IDisposable、IDisposable、IComparable、IEquatable、IEnumerable 等都只包含一个公共方法。
- 接口名称前面的大写字母“I”是一个约定，正如字段名以下划线开头一样，请坚持这些原则。
- 在接口中，所有的方法都默认为 public。
- 如果预计会出现版本问题，可以创建“抽象类”。例如，创建了狗（Dog）、鸡（Chicken）和鸭（Duck），那么应该考虑抽象出动物（Animal）来应对以后可能出现风马牛的事情。而向接口中添加新成员则会强制要求修改所有派生类，并重新编译，所以版本式的问题最好以抽象类来实现。
- 从抽象类派生的非抽象类必须包括继承的所有抽象方法和抽象访问器的实实现。
- 对抽象类不能使用 new 关键字，也不能被密封，原因是抽象类不能被实例化。
- 在抽象方法声明中不能使用 static 或 virtual 修饰符。

以上的规则，我就厚颜无耻的暂定为 T14 条吧，写的这么累，就当一时的奖赏吧。大家也可以互通有无，我将及时修订。

4. 经典示例

4.1 绝对经典

.NET Framework 是学习的最好资源，有意识的研究 FCL 是每个.NET 程序员的必修课，关于接口和抽象类在 FCL 中的使用，我有以下的建议：

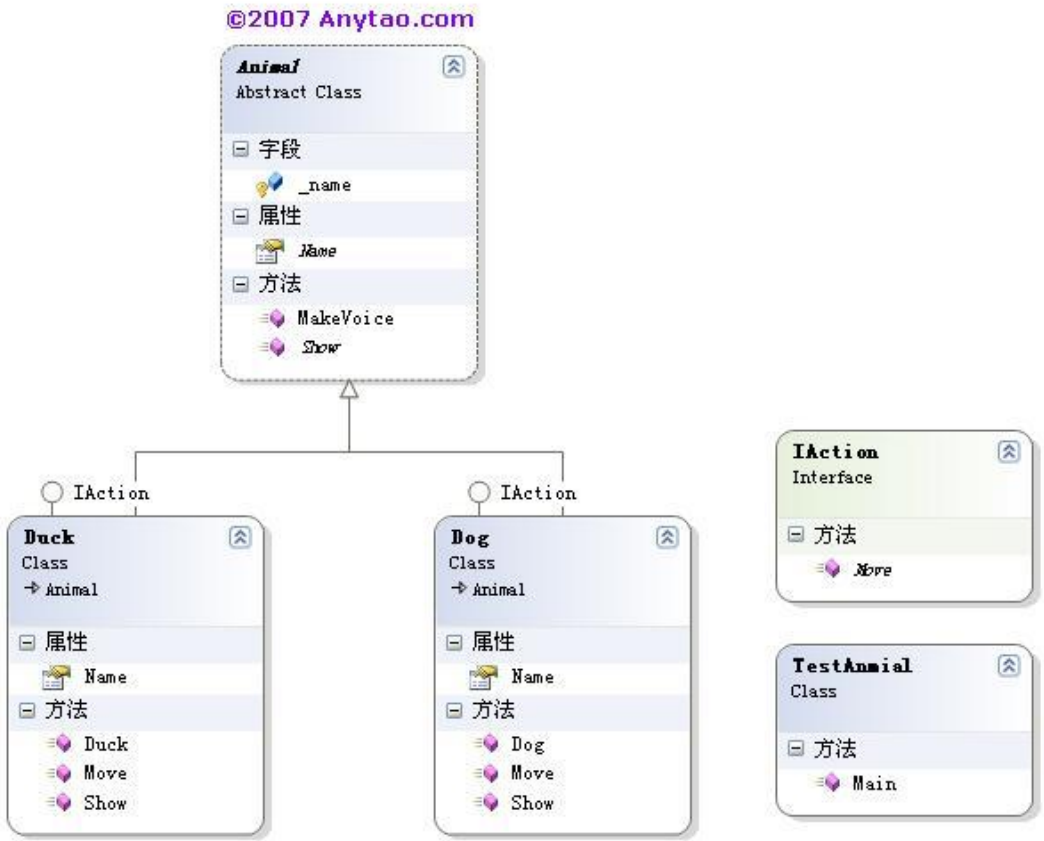
- FCL 对集合类使用了基于接口的设计，所以请关注 System.Collections 中关于接口的设计实现；

- FCL 对数据流相关类使用了基于抽象类的设计，所以请关注 System.IO.Stream 类的抽象类设计机制。

4.2 别样小菜

下面的实例，因为是我的理解，因此给经典打上“相对”的记号，至于什么时候晋升为“绝对”，就看我在.NET 追求的路上，是否能够一如既往的如此执着，因此我将把相对重构到绝对为止（呵呵）。 本示例没有阐述抽象类和接口在设计模式中的应用，因为那将是另一篇有讨论价值的文本，本文着眼与概念和原则的把握，但是真正的应用来自于具体的需求规范。

设计结构如图所示：



1. 定义抽象类

```
public abstract class Animal
{
    protected string _name;

    //声明抽象属性
    public abstract string Name
    {
        get;
    }

    //声明抽象方法
    public abstract void Show();

    //实现一般方法
    public void MakeVoice()
    {
        Console.WriteLine("All animals can make voice!");
    }
}
```

2. 定义接口

```
public interface IAction
{
    //定义公共方法标签
    void Move();
}
```

3. 实现抽象类和接口

```


public class Duck : Animal, IAction
{
    public Duck(string name)
    {
```

```
        _name = name;
    }

    //重载抽象方法
    public override void Show()
    {
        Console.WriteLine(_name + " is showing for you.");
    }

    //重载抽象属性
    public override string Name
    {
        get { return _name;}
    }

    //实现接口方法
    public void Move()
    {
        Console.WriteLine("Duck also can swim.");
    }

}

public class Dog : Animal, IAction
{
    public Dog(string name)
    {
        _name = name;
    }

    public override void Show()
    {
        Console.WriteLine(_name + " is showing for you.");
    }

    public override string Name
```

```
    {
        get { return _name; }
    }

    public void Move()
    {
        Console.WriteLine(_name + " also can run.");
    }

}
```

4. 客户端实现

```
public class TestAnmial
{
    public static void Main(string [] args)
    {
        Animal duck = new Duck("Duck");
        duck.MakeVoice();
        duck.Show();

        Animal dog = new Dog("Dog");
        dog.MakeVoice();
        dog.Show();

        IAction dogAction = new Dog("A big dog");
        dogAction.Move();
    }
}
```

5. 他山之石

正所谓真理是大家看出来的，所以将园子里有创新性的观点潜列于此，一是感谢大家的共享，二是完善一家之言的不足，希望能够将领域形成知识，受用于我，受用于众。

- dunai 认为：抽象类是提取具体类的公因式，而接口是为了将一些不相关的类"杂凑"成一个共同的群体。至于他们在各个语言中的句法，语言细节并不是我关心的重点。
- 桦山涧的收藏也很不错。
- Artech 认为：所代码共用和可扩展性考虑，尽量使用 Abstract Class。当然接口在其他方面的优势，我认为也不可忽视。
- shenfx 认为：当在差异较大的对象间寻求功能上的共性时，使用接口；当在共性较多的对象间寻求功能上的差异时，使用抽象基类。

最后，MSDN 的建议是：

- 如果预计要创建组件的多个版本，则创建抽象类。抽象类提供简单易行的方法来控制组件版本。通过更新基类，所有继承类都随更改自动更新。另一方面，接口一旦创建就不能更改。如果需要接口的新版本，必须创建一个全新的接口。
- 如果创建的功能将在大范围的全异对象间使用，则使用接口。抽象类应主要用于关系密切的对象，而接口最适合为不相关的类提供通用功能。
- 如果要设计小而简练的功能块，则使用接口。如果要设计大的功能单元，则使用抽象类。
- 如果要在组件的所有实现间提供通用的已实现功能，则使用抽象类。抽象类允许部分实现类，而接口不包含任何成员的实现。

6. 结论

接口和抽象类，是论坛上、课堂间讨论最多的话题之一，之所以将这个老话题拿出来再议，是因为从我的体会来说，深刻的理解这两个面向对象的基本内容，对于盘活面向对象的抽象化编程思想至关重要。本文基本概况了接口和抽象类的概念、异同和使用规则，从学习的观点来看，我认为这些总结已经足以表达其核心。但是，对于面向对象和软件设计的深入理解，还是建立在不断实践的基础上，Scott 说自己每天坚持一个小时用来写 Demo，那么我们是不是更应该勤于键盘呢。对于接口和抽象类，请多用而知其然，多想而知其奥吧。

7.5：恩怨情仇：is 和 as

本文将介绍以下内容：

- 类型转换

- is/as 操作符小议

1. 引言

类型安全是.NET 设计之初重点考虑的内容之一，对于程序设计者来说，完全把握系统数据的类型安全，经常是力不从心的问题。现在，这一切已经在微软大牛们的设计框架中为你解决了。在.NET 中，一切类型都必须集成自 System.Object 类型，因此我们可以很容易的获得对象的准确类型，方法是：GetType()方法。那么.NET 中的类型转换，应该考虑的地方有那些呢？

2. 概念引入

类型转换包括显示转换和隐式转换，在.NET 中类型转换的基本规则如下：

- 任何类型都可以安全的转换为其基类类型，可以由隐式转换来完成；
- 任何类型转换为其派生类型时，必须进行显示转换，转换的规则是：（类型名）对象名；
- 使用 GetType 可以取得任何对象的精确类型；
- 基本类型可以使用 Covert 类实现类型转换；
- 除了 string 以外的其他类型都有 Parse 方法，用于将字符串类型转换为对应的基本类型；
- 值类型和引用类型的转换机制称为装箱（boxing）和拆箱（unboxing）。

3. 原理与示例说明

浅谈了类型转换的几个普遍关注的方面，该将主要精力放在 is、as 操作符的恩怨情仇上了。类型转换将是个较大的话题，留于适当的时机讨论。

is/as 操作符，是 C#中用于类型转换的，提供了对类型兼容性的判断，从而使得类型转换控制在安全的范畴，提供了灵活的类型转换控制。

is 的规则如下：

- 检查对象类型的兼容性，并返回结果，true 或者 false；
- 不会抛出异常；
- 如果对象为 null，则返回值永远为 false。

其典型用法为：

```
object o = new object();

class A

{

}

if (o is A)  //执行第一次类型兼容检查

{

  A a = (A) o;  //执行第二次类型兼容检查

}


```

as 的规则如下：

- 检查对象类型的兼容性，并返回结果，如果不兼容就返回 null；
- 不会抛出异常；
- 如果结果判断为空，则强制执行类型转换将抛出 NullReferenceException 异常。

其典型用法为：

```
object o = new object();

class B

{

}

```

```
 9  B b = o as B;  //执行一次类型兼容检查
10
11  if (b != null)
12
13 {
14
15     MessageBox.Show("b is B's instance.");
16
17 }
18
19
```

4. 结论

纵上比较，is/as 操作符，提供了更加灵活的类型转型方式，但是 as 操作符在执行效率上更胜一筹，我们在实际的编程中应该体会其异同，酌情量才。

7.6：貌合神离：覆写和重载

本文将介绍以下内容：

- 什么是覆写，什么是重载
- 覆写与重载的区别
- 覆写与重载在多态特性中的应用

1. 引言

覆写（override）与重载（overload），是成就.NET 面向对象多态特性的基本技术之一，两个貌似相似而实则不然的概念，常常带给我们很多的误解，因此有必要以专题来讨论清楚其区别，而更重要的是关注其在多态中的应用。

在系列中，我们先后都有关于这一话题的点滴论述，本文以专题的形式再次做以深度讨论，相关的内容请对前文做以参考。

2. 认识覆写和重载

从一个示例开始来认识什么是覆写，什么是重载？

```
abstract class  Base
{
   //定义虚方法
   public virtual void MyFunc()
   {
   }

   //参数列表不同，virtual 不足以区分
   public virtual void MyFunc(string str)
   {
   }

   //参数列表不同，返回值不同
   public bool MyFunc(string str, int id)
   {
      Console.WriteLine("AAA");
      return true;
   }

   //参数列表不同表现为个数不同，或者相同位置的参数类型不同
   public bool MyFunc(int id, string str)
   {
      Console.WriteLine("BBB");
      return false;
   }

   //泛型重载，允许参数列表相同
   public bool MyFunc<T>(string str, int id)
   {
      return true;
```

```
    }

    //定义抽象方法
    public abstract void Func();
  }

  class Derived: Base
  {
    //阻隔父类成员
    public new void MyFunc()
    {
    }

    //覆写基类成员
    public override void MyFunc(string str)
    {
      //在子类中访问父类成员
      base.MyFunc(str);
    }

    //覆写基类抽象方法
    public override void Func()
    {
      //实现覆写方法
    }
}
```

2.1 覆写基础篇

覆写，又称重写，就是在子类中重复定义父类方法，提供不同实现，存在于有继承关系的父子关系。当子类重写父类的虚函数后，父类对象就可以根据根据赋予它的不同子类指针动态的调用子类的方法。从示例的分析，总结覆写的基本特征包括：

- 在.NET 中只有以 virtual 和 abstract 标记的虚方法和抽象方法才能被直接覆写。
- 覆写以关键字 override 标记，强调继承关系中对基类方法的重写。

- 覆写方法要求具有相同的方法签名，包括：相同的方法名、相同的参数列表和相同的返回值类型。

概念：虚方法

虚方法就是以 virtual 关键字修饰并在一个或多个派生类中实现的方法，子类重写的虚方法则以 override 关键字标记。虚方法调用，是在运行时确定根据其调用对象的类型来确定调用适当的覆写方法。.NET 默认是非虚方法，如果一个方法被 virtual 标记，则不可再被 static、abstrcat 和 override 修饰。

概念：抽象方法

抽象方法就是以 abstract 关键字修饰的方法，抽象方法可以看作是没有实现体的虚方法，并且必须在派生类中被覆写，如果一个类包括抽象方法，则该类就是一个抽象类。因此，抽象方法其实隐含为虚方法，只是在声明和调用语法上有所不同。abstract 和 virtual 一起使用是错误的。

2.2 重载基础篇

重载，就是在同一个类中存在多个同名的方法，而这些方法的参数列表和返回值类型不同。值得注意的是，重载的概念并非面向对象编程的范畴，从编译器角度理解，不同的参数列表、不同的返回值类型，就意味着不同的方法名。也就是说，方法的地址，在编译期就已经确定，是这一种静态绑定。从示例中，我们总结重载的基本特征包括：

- 重载存在于同一个类中。
- 重载方法要求具有相同的方法名，不同的参数列表，返回值类型可以相同也可以不同（通过 operator implicit 可以实现一定程度的返回值重载，不过不值得推荐）。
- .NET 2.0 引入泛型技术，使得相同的参数列表、相同的返回值类型的情况也可以构成重载。

3. 在多态中的应用

多态性，简单的说就是“一个接口，多个方法”，具体表现为相同的方法签名代表不同的方法实现，同一操作作用于不同的对象，产生不同的执行结果。在.NET 中，覆写实现了运行时的多态性，而重载实现了编译时的多态性。

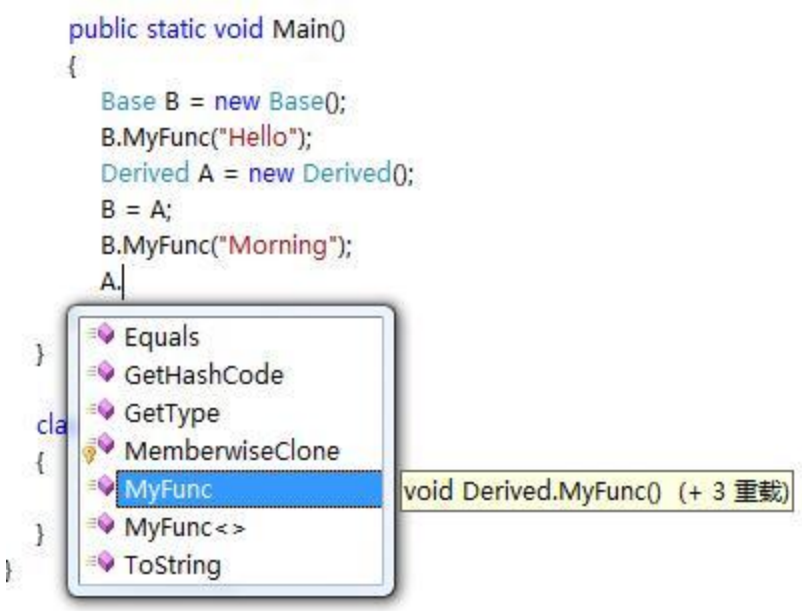
运行时的多态性，又称为动态联编，通过虚方法的动态调度，在运行时根据实际的调用实例类型决定调用的方法实现，从而产生不同的执行结果。

```
class Base
{
```

```
    public virtual void MyFunc(string str)
    {
        Console.WriteLine("{0} in Base", str);
    }
}

class Derived: Base
{
    //覆写基类成员
    public override void MyFunc(string str)
    {
        Console.WriteLine("{0} in Derived", str);
    }

    public static void Main()
    {
        Base B = new Base();
        B.MyFunc("Hello");
        Derived A = new Derived();
        B = A;
        B.MyFunc("Morning");
    }
}
```

从结果中可知，对象 B 两次执行 B.MyFunc 调用了不同的方法，第一次调用基类方法 MyFunc，而第二次调用了派生类方法 MyFunc。在执行过程中，对象 B 先后指向了不同的类的实例，从而动态调用了不同的实例方法，显然这一执行操作并非确定于编译时，而是在运行时根据对象 B 执行的不同类型来确定的。我们在此不分析虚拟方法的动态调度机制，而只关注通过虚方法覆写而实现的多态特性，详细的实现机制请参考本系列的其它内容。

编译时的多态性，又称为静态联编，一般包括方法重载和运算符重载。对于非虚方法来说，在编译时通过方法的参数列表和返回值类型决定不同操作，实现编译时的多态性。例如，在实际的开发过程中，.NET 开发工具 Visual Studio 的智能感知功能就很好的为方法重载提供了很好的交互手段，例如：



从智能感知中可知方法 MyFunc 在派生类 Derived 中有三次重载，调用哪种方法由程序开发者根据其参数、返回值的不同而决定。由此可见，方法重载是一种编译时的多态，对象 A 调用哪种方法在编译时就已经确定。

4. 比较，还是规则

- 如果基访问引用的是一个抽象方法，则将导致编译错误。

```
abstract class  Base
{
   public abstract void Func();
}

class Derived: Base
{
   //覆写基类抽象方法
   public override void Func()
   {
      base.Func();
   }
}
```

- 虚方法不能是静态的、密封的。

- 覆写实现的多态确定于运行时，因此更加的灵活和抽象；重载实现的多态确定于编译时，因此更加的简单和高效。二者各有特点与应用，不可替代。

在下表中，将覆写与重载做以总结性的对比，主要包括：

规则	覆写（override）	重载（overload）
存在位置	存在于有继承关系的不同类中	存在于同一个类中
调用机制	运行时确定	编译时确定
方法名	必须相同	必须相同
参数列表	必须相同	必须不同
返回值类型	必须相同	可以不相同
泛型方法	可以覆写	可以重载

注：参数列表相同表示参数的个数相同，并且相同位置的参数类型也相同。

5. 结论

深入的理解覆写和重载，是对多态特性和面向对象机制的有力补充，本文从基本概念到应用领域将两个概念进行一一梳理，通过对比整理区别，还覆写和重载以更全面的认知角度，同时也更能从侧面深入的了解运行时多态与编译时多态的不同情况。

再谈重载与覆写

昨天我在新手区发了一篇《重载还是覆写？》的随笔，后来我发现我犯了一个严重的错误，没有具体说明是.NET 1.1 还是 2.0，在.NET2.0 中由于泛型的出现，对重载和覆写有时候就不能按照 1.1 下那几个特征去要求。

1．重载（Overload）

在.NET1.1 下，我们定义重载：类中定义的方法可能有不同的版本，它具有如下的特征：

I. 方法名必须相同

II. 参数列表必须不相同，与参数列表的顺序无关

III. 返回值类型可以不相同

示意代码：

```
public class MyClass
{
    public void Write(string _str)
    {
        // … …
    }
    public void Write(string _str, string _target)
    {
        // … …
    }
    public bool Write(string _str, string _target, bool _flag)
    {
        // … …
    }
}
```

在.NET2.0 下，由于泛型的出现，我们就不能再用这三个特征来判断重载，如下面的两个方法，它们具有相同的方法名，相同的参数列表，相同的返回值，但是它们却可以构成重载：

```
public class MyClass
{
    public void Write<T>(string _str)
    {
        // … …
    }
    public void Write(string _str)
```

```
    {
        //... ...
    }
}
```

再看下面这两个方法，它们不能构成重载，因为如果 T，U 如果实例化时传入相同的类型，则这两个方法就具有相同的签名：

```
public class MyClass8<T,U>
{
    public T MyMothed(T a, U b)
    {
        return a;
    }
    public U MyMothed(U a, T b)
    {
        return b;
    }
}
```

但是当我们再添加另外一个方法后，这个类却可以编译通过：

```
public class MyClass8<T,U>
{
    public T MyMothed(T a, U b)
    {
        return a;
    }
    public U MyMothed(U a, T b)
    {
        return b;
    }
    public int MyMothed(int a, int b)
```

```
    {
        return a + b;
    }
}
```

通过调用可以发现，优先匹配的方法是一般方法，而非泛型方法。总之，构成重载的第二个特征参数列表必须不同，其实是让方法具有不同的签名，调用程序可以区分，在有泛型时要特别注意，而第一点和第三点仍然适用。

2．覆写（Override）

在.NET1.1 下，对覆写我们的定义是：子类中为满足自己的需要来重复定义某个方法的不同实现，它具有如下特征：

I. 通过使用关键字 Override 来覆写

II. 只有虚方法和抽象方法直接可以被覆写

III. 相同的方法名

IV. 相同的参数列表

V. 相同的返回值类型

示意代码：

```
public abstract class BaseClass
{
    public abstract void Write(string _str);
}
public class SubClass : BaseClass
{
    public override void Write(string _str)
    {
        // … …
    }
}
```

在.NET2.0 中，泛型方法的覆写，除了要遵守以上几点外，还应该注意：

在重写定义了泛型参数的虚拟方法时，子类方法必须重新定义该方法特定的泛型参数：

```
public class MyBaseClass
{
    public virtual void MyMothed<T>(T t)
    {
        // … …
    }
}
public class MySubClass : MyBaseClass
{
    public override void MyMothed<T>(T t)//重新定义泛型参数 T
    {
        // … …
    }
}
```

在重写定义了泛型参数的虚拟方法时，子类实现不能重复在基类方法级别出现的约束：

```
public class MyBaseClass
{
    public virtual void MyMothed<T>(T t) where T : new()
    {
        // … …
    }
}
public class MySubClass:MyBaseClass
{
    public override void MyMothed<T>(T t)//不能重复任何约束
    {
        // … …
```

```
    }
}
```

第 8 章 本来面目——框架诠释

8.1 万物归宗：System.Object

本节将介绍以下内容：

— System.Object 类型解析

— Object 类型的常用方法及其应用

8.1.1 引言

正如标题所示，System.Object 是所有类型的基类，任何类型都直接或间接继承自 System.Object 类。没有指定基类的类型都默认继承于 System.Object，从而具有 Object 的基本特性，这些特性主要包括：

- 通过 GetType 方法，获取对象类型信息。
- 通过 Equals、ReferenceEquals 和＝＝，实现对象判等。
- 通过 ToString 方法，获取对象字符串信息 ，默认返回对象类型全名。
- 通过 MemberwiseClone 方法，实现对象实例的浅拷贝。
- 通过 GetHashCode 方法，获取对象的值的散列码。
- 通过 Finalize 方法，在垃圾回收时进行资源清理。

接下来，我们和这些公共特性一一过招，来了解其作用和意义，深入其功能和应用。

8.1.2 初识

有了对 Object 类型的初步认识，我们使用 Reflector 工具加载 mscorlib 程序集来反编译 Sytem.Object 的实现情况，首先不关注具体的实现细节，将注意力放在基本的类型定义上：

```
public class Object
{
   //构造函数
   public Object() { }
   public virtual int GetHashCode() { }
   //获取对象类型信息
   public System.Type GetType() { }
   //虚方法，返回对象的字符串表示方式
   public virtual string ToString() { }
   //几种对象判等方法
   public virtual bool Equals(object obj) { }
   public static bool Equals(object objA, object objB) { }
   public static bool ReferenceEquals(object objA, object objB) { }
   //执行对象的浅拷贝
   protected object MemberwiseClone() { }
   //析构函数
   protected virtual void Finalize() { }
}
```

从反编译代码中可知，System.Object 主要包括了 4 个公用方法和 2 个受保护方法，其具体的应用和实现在后文表述。

8.1.3 分解

下面，我们选择 Object 的几个主要的方法来分析其实现，以便从整体上把握对 Object 的认知。

1. ToString 解析

ToString 是一个虚方法，用于返回对象的字符串表示，在 Object 类型的实现可以表示为：

```
public virtual string ToString()
```

```
{
    return this.GetType().FullName.ToString();
}
```

可见，默认情况下，对象调用 ToString 方法将返回类型全名称，也就是命名空间加类型名全称。在通常的情况下，ToString 方法提供了在子类中重新覆写基类方法而获取对象当前值的字符串信息的合理途径。例如，下面的类型 MyLocation 将通过 ToString 方法来获取其坐标信息：

```
class MyLocation
{
    private int x = 0;
    private int y = 0;
    public override string ToString()
    {
        return String.Format("The location is ({0}, {1}).", x, y);
    }
}
```

而.NET 框架中的很多类型也实现了对 ToString 方法的覆写，例如 Boolean 类型通过覆写 ToString 来返回真或者假特征：

```
public override string ToString()
{
    if (!this)
    {
        return "False";
    }
    return "True";
}
```

ToString 方法，可以在调试期快速获取对象信息，但是 Object 类型中实现的 ToString 方法还是具有一些局限性，例如在格式化、语言文化方面 Object.ToString 方法就没有更多的选择。解决的办法就是实现 IFormattable 接口，其定义为：

```
public interface IFormattable
```

```
{
    string ToString(string format, System.IFormatProvider formatProvider);
}
```

其中，参数 format 表明要格式化的方式，而参数 formatProvider 则提供了特定语言文化的信息。事实上，.NET 基本类型都实现了 IFormattable 接口，以实现更灵活的字符串信息选择。以 DateTime 类型的 ToString 方法为例，其实现细节可表示为：

```
public struct DateTime : IFormattable
{
    public string ToString(string format, IFormatProvider provider)
    {
        return DateTimeFormat.Format(this, format,
                                     DateTimeFormatInfo.GetInstance(provider));
    }
}
```

我们可以通过控制 format 参数和 provider 参数来实现特定的字符串信息返回，例如要想获取当前线程的区域性长格式日期时，可以以下面的方式实现：

```
DateTime dt = DateTime.Now;
string time = dt.ToString("D", DateTimeFormatInfo.CurrentInfo);
```

而想要获取固定区域性短格式日期时，则以另外的设定来实现：

```
DateTime dt = DateTime.Now;
string time = dt.ToString("d", DateTimeFormatInfo.InvariantInfo);
```

关于 ToString 方法，还应指出的是 System.String 类型中并没有实现 IFormattable 接口，System.String.ToString 方法用来返回当前对象的一个引用，也就是 this。

2. GetType 解析

GetType 方法为非虚的，用于在运行时通过查询对象元数据来获取对象的运行时类型。因为子类无法通过覆写 GetType 而篡改类型信息，从而保证类型安全。例如在下面的示例中：

```
class MyType
```

```
{
}
class Test_GetType
{
   public static void Main()
   {
      MyType mt = new MyType();
      //使用 Object.GetType 返回 Type 实例
      Type tp = mt.GetType();
      //返回类型全名称
      Console.WriteLine(tp.ToString());
      //仅返回类型名
      Console.WriteLine(tp.Name.ToString());
   }
}
//执行结果
//InsideDotNet.Framework.Object.MyType
//MyType
```

GetType 返回的是一个 System.Type 或其派生类的实例。而该实例对象可以通过反射获取类型的元数据信息。从可以提供所属类型的很多信息，例如字段、属性和方法等，例如：

```
class MyType
{
   private int number = 0;
   private string name = null;
   public static void ShowType(string type, string info)
   {
      Console.WriteLine("This type is MyType.");
   }
   private void ShowNumber()
   {
      Console.WriteLine(number.ToString());
```

```
    }
}
class Test_GetType
{
    public static void Main()
    {
        MyType mt = new MyType();
        //根据 Type 实例查找类型成员
        foreach (MemberInfo info in tp.GetMembers())
        {
            Console.WriteLine("The member is {0}, {1}", info.Name, info.DeclaringType);
        }
        //根据 Type 实例查找类型方法
        foreach (MethodInfo mi in tp.GetMethods())
        {
            Console.WriteLine("The method is {0}", mi.ToString());
            //查找方法参数信息
            ParameterInfo[] pis = mi.GetParameters();
            foreach (ParameterInfo pi in pis)
            {
                Console.WriteLine("{0}'s member is {1}", mi.ToString(), pi.ToString());
            }
        }
    }
}
```

通过反射机制，就可以根据 GetType 方法返回的 Type 对象在运行期枚举出元数据表中定义的所有类型的信息，并根据 System.Reflection 空间中的方法获取类型的信息，包括：字段、属性、方法、参数、事件等，例如上例中就是根据 System.Reflection 中定义的相关方法来完成获取对象信息的处理过程。在晚期绑定的应用场合中，这种处理尤为常见。

.NET 中，用于在运行期获取类型 Type 实例的方法并非只有 Object.GetType 方法，Type.GetType 静态方法和 typeof 运算符也能完成同样的操作，不过在应用上有些区别，主要是：

- Type.GetType 是非强类型方法；而 typeof 运算符支持强类型。

```
Type tp = Type.GetType("InsideDotNet.Framework.Object.MyType");
Type tp = typeof(InsideDotNet.Framework.Object.MyType);
```

- Type.GetType 支持运行时跨程序集反射，以解决动态引用；而 typeof 只能支持静态引用。

```
Assembly ass = Assembly.LoadFrom(@"C:\Anytao.Utility.exe");
Type tpd = ass.GetType("Anytao.Utility.Message.AnyMsg");
Console.WriteLine(tpd.ToString());
```

注意：Type.GetType 必须使用完全限定名，以避免模块依赖或循环引用问题。

另外，对于在运行期获取 Type 实例的方法，还可参考以下几种常见的方式，主要包括：

- 利用 System.Reflection.Assembly 的非静态方法 GetType 或 GetTypes。
- 利用 System.Reflection.Module 的非静态方法 GetType 或 GetTypes。

通过 Assembly 或 Module 实例来获取 Type 实例，也是程序设计中常见的技巧之一。

3. 其他

- Equals 静态方法、虚方法和 ReferenceEquals 方法用于对象判等，详细的应用请参考 8.2 节“规则而定：对象判等”。
- GetHashCode 方法，用于在类型中提供哈希值，以应用于哈希算法或哈希表，不过值得注意的是对 Equals 方法和 GetHashCode 方法的覆写要保持统一，因为两个对象的值相等，其哈希码也应该相等，否则仅覆写 Equals 而不改变 GetHashCode，会导致编译器抛出警告信息。
- Memberwise 方法，用于在对象克隆时实现对象的浅拷贝，详细应用请参考 7.7 节“有深有浅的克隆：浅拷贝和深拷贝”。
- Finalize 方法，用于在垃圾回收时实现资源清理，详细应用请参考 5.3 节“垃圾回收”。

8.1.4 意义

l 实现自上而下的单根继承。

l System.Object 是一切类型的最终基类，也就意味着.NET 的任何变量都是 System.Object 的实例，这种机制提供了不同类型之间进行交互通信的可能。也赋予了所有.NET 基本类型的最小化功能方法，例如 ToString 方法、GetHashCode 方法和 Equals 方法等。

8.1.5 结论

通过本节的论述，我们基本了解了 System.Object 类型的设计思路和实现细节，从框架设计的角度来看，我们应该了解和学习 System.Object 在设计与实现上的可取之道，一方面.NET 框架提供了最小功能特征在子类中继承，另一方面则分别将不同的特征方法实现为不同的访问级别和虚方法，这些思路和技巧正是值得我们借鉴和深思的精华所在。

8.2 规则而定：对象判等

本节将介绍以下内容：

— 四种判等方法解析

— 实现自定义 Equals 方法

— 判等规则

8.2.1 引言

了解.NET 的对象判等，有必要从了解几个相关的基本概念开始：

l 值相等。表示比较的两个对象的数据成员按内存位分别相等，即两个对象类型相同，并且具有相等和相同的字段。

l 引用相等。表示两个引用指向同一对象实例，也就是同一内存地址。因此，可以由引用相等推出其值相等，反之则不然。

关于对象的判等，涉及了对相等这一概念的理解。其实这是一个典型的数学论题，所以数学上的等价原则也同样适用于对象判等时的规则，主要是：

- 自反性，就是 a==a 总是为 true。
- 对称性，就是如果 a==b 成立，则 b==a 也成立。
- 传递性，就是如果 a==b，b==c 成立，则 a==c 也成立。

了解了对象判断的类型和原则，接下来就认识一下 System.Object 类中实现的几个对象判等方法，它们是：

- public virtual bool Equals（object obj）虚方法，比较对象实例是否相等。
- public static bool Equals（object objA，object objB）静态方法，比较对象实例是否相等。
- public static bool ReferenceEquals（object objA，object objB）静态方法，比较两个引用是否指向同一个对象。

同时在.NET 中，还有一个“==”操作符提供了更简洁的语义来表达对象的判等，所以.NET 的对象判等方法就包括了这四种类型，下面一一展开介绍。

8.2.2 本质分析

1. Equals 静态方法

Equals 静态方法实现了对两个对象的相等性判别，其在 System.Object 类型中实现过程可以表示为：

```
public static bool Equals(object objA, object objB)
{
   if (objA == objB)
   {
      return true;
   }
   if ((objA != null) && (objB != null))
   {
      return objA.Equals(objB);
   }
```

```
    return false;
}
```

对以上过程，可以小结为：首先比较两个类型是否为同一实例，如果是则返回 true；否则将进一步判断两个对象是否都为 null，如果是则返回 true；如果不是则返回 objA 对象的 Equals 虚方法的执行结果。所以，Equals 静态方法的执行结果，依次取决于三个条件：

- 是否为同一实例。
- 是否都为 null。
- 第一个参数的 Equals 实现。

因此，通常情况下 Equals 静态方法的执行结果常常受到判等对象的影响，例如有下面的测试过程：

```
class MyClassA
{
    public override bool Equals(object obj)
    {
        return true;
    }
}
class MyClassB
{
    public override bool Equals(object obj)
    {
        return false;
    }
}
class Test_Equals
{
    public static void Main()
    {
        MyClassA objA = new MyClassA();
```

```
        MyClassB objB = new MyClassB();
        Console.WriteLine(Equals(objA, objB));
        Console.WriteLine(Equals(objB, objA));
    }
}
//执行结果
True
False
```

由执行结果可知，静态 Equals 的执行取决于==操作符和 Equals 虚方法这两个因素。因此，决议静态 Equals 方法的执行，就要在自定义类型中覆写 Equals 方法和重载==操作符。

还应注意到，.NET 提供了 Equals 静态方法可以解决两个值为 null 对象的判等问题，而使用 objA.Equals（object objB）来判断两个 null 对象会抛出 NullReferenceException 异常，例如：

```
public static void Main()
{
    object o = null;
    o.Equals(null);
}
```

2. ReferenceEquals 静态方法

ReferenceEquals 方法为静态方法，因此不能在继承类中重写该方法，所以只能使用 System.Object 的实现代码，具体为：

```
public static bool ReferenceEquals(object objA, object objB)
{
    return (objA == objB);
}
```

可见，ReferenceEquals 方法用于判断两个引用是否指向同一个对象，也就是前文强调的引用相等。因此以 ReferenceEquals 方法比较同一个类型的两个对象实例将返回 fasle，而.NET 认为 null 等于 null，因此下面的实例就能很容易理解得出的结果：

```
public static void Main()
```

```
{
    MyClass mc1 = new MyClass();
    MyClass mc2 = new MyClass();
    //mc1 和 mc3 指向同一对象实例
    MyClass mc3 = mc1;
    //显示：False
    Console.WriteLine(ReferenceEquals(mc1, mc2));
    //显示：True
    Console.WriteLine(ReferenceEquals(mc1, mc3));
    //显示：True
    Console.WriteLine(ReferenceEquals(null, null));
    //显示：False
    Console.WriteLine(ReferenceEquals(mc1, null));
}
```

因此，ReferenceEquals 方法，只能用于比较两个引用类型，而以 ReferenceEquals 方法比较值类型，必然伴随着装箱操作的执行，分配在不同地址的两个装箱的实例对象，肯定返回 false 结果，关于装箱详见 4.4 节“皆有可能——装箱与拆箱”。例如：

```
public static void Main()
{
    Console.WriteLine(ReferenceEquals(1, 1));
}
//执行结果：False
```

另外，应该关注.NET 某些特殊类型的“意外”规则，例如下面的实现将突破常规，除了深刻地了解 ReferenceEquals 的实现规则，也应理解某些特殊情况背后的秘密：

```
public static void Main()
{
    string strA = "ABCDEF";
    string strB = "ABCDEF";
    Console.WriteLine(ReferenceEquals(strA, strB));
```

```
}
//执行结果：True
```

从结果分析可知两次创建的 string 类型实例不仅内容相同，而且分享共同的内存空间，事实上的确如此，这缘于 System.String 类型的字符串驻留机制，详细的讨论见 8.3 节“为什么特殊：string 类型解析”，在此我们必须明确 ReferenceEquals 判断引用相等的实质是不容置疑的。

3. Equals 虚方法

Equals 虚方法用于比较两个类型实例是否相等，也就是判断两个对象是否具有相同的“值”，在 System.Object 中其实现代码，可以表示为：

```
public virtual bool Equals(object obj)
{
   return InternalEquals(this, obj);
}
```

其中 InternalEquals 为一个静态外部引用方法，其实现的操作可以表示成：

```
if (this == obj)
   return true;
else
   return false;
```

可见，默认情况下，Equals 方法和 ReferenceEquals 方法是一样的，Object 类中的 Equals 虚方法仅仅提供了最简单的比较策略：如果两个引用指向同一个对象，则返回 true；否则将返回 false，也就是判断是否引用相等。然而这种方法并未达到 Equals 比较两个对象值相等的目标，因此 System.Object 将这个任务交给其派生类型去重新实现，可以说 Equals 的比较结果取决于类的创建者是如何实现的，而非统一性约定。

事实上，.NET 框架类库中有很多的引用类型实现了 Equals 方法用于比较值相等，例如比较两个 System.String 类型对象是否相等，肯定关注其内容是否相等，判断的是值相等语义：

```
public static void Main()
{
   string str1 = "acb";
```

```
    string str2 = "acb";
    Console.WriteLine(str1 == str2);
}
```

4. ==操作符

在.NET 中，默认情况下，操作符“==”在值类型情况下表示是否值相等，由值类型的根类 System.ValueType 提供了实现；而在引用类型情况下表示是否引用相等，而“!=”操作符与“==”语义类似。当然也有例外，System.String 类型则以“==”来处理值相等。因此，对于自定义值类型，如果重载 Equals 方法，则应该保持和“==”在语义上的一致，以返回值相等结果；而对于引用类型，如果以覆写来处理值相等规则时，则不应该再重载“==”运行符号，因为保持其缺省语义为判断引用相等才是恰当的处理规则。

Equals 虚方法与==操作符的主要区别在于多态表现：Equals 通过虚方法覆写来实现，而==操作符则是通过运算符重载来实现，覆写和重载的区别请参考 1.4 节“多态的艺术”。

8.2.3 覆写 Equals 方法

经过对四种不同类型判等方法的讨论，我们不难发现不管是 Equals 静态方法、Equals 虚方法抑或==操作符的执行结果，都可能受到覆写 Equals 方法的影响。因此研究对象判等就必须将注意力集中在自定义类型中如何实现 Equals 方法，以及实现怎样的 Equals 方法。因为，不同的类型，对于“相等”的理解会有所偏差，你甚至可以在自定义类型中实现一个总是相等的类型，例如：

```
class AlwaysEquals
{
    public override bool Equals(object obj)
    {
        return true;
    }
}
```

因此，Euqls 方法的执行结果取决于自定义类型的具体实现规则，而.NET 又为什么提供这种机制来实现对象判等策略呢？首先，对象判等决定于需求，没有必要为所有.NET 类型完成逻辑判等，

System.Object 基类也无法提供满足各种需求的判等方法；其次，对象判等包括值判等和引用判等两个方面，不同的类型对判等的处理又有所不同，通过多态机制在派生类中处理各自的判等实现显然是更加明智与可取的选择。

接下来，我们开始研究如何通过覆写 Equals 方法实现对象的判等。覆写 Equals 往往并非易事，要综合考虑到对值类型字段和引用类型字段的分别判等处理，同时还要兼顾父类覆写所带来的影响。不适当的覆写会引发意想不到的问题，所以必须遵循三个等价原则：自反、传递和对称，这是实现 Equals 的通用契约。那么又如何为自定义类型实现 Equals 方法呢？

最好的参考资源当然来自于.NET 框架类库的实现，事实上，关于 Equals 的覆写在.NET 中已经有很多的基本类型完成了这一实现。从值类型和引用类型两个角度来看：

- 对于值类型，基类 System.ValueType 通过反射机制覆写了 Equals 方法来比较两个对象的值相等，但是这种方式并不高效，更明智的办法是在自定义值类型时有针对性的覆写 Equals 方法，来提供更灵活、高效的处理机制。
- 对于引用类型，覆写 Equals 方法意味着要改变 System.Object 类型提供的引用相等语义。那么，覆写 Equals 要根据类型本身的特点来实现，在.NET 框架类库中就有很多典型的引用类型实现了值相等语义。例如 System.String 类型的两个变量相等意味着其包含了相等的内容，System.Version 类型的两个变量相等也意味着其 Version 信息的各个指标分别相等。

因此对 Equals 方法的覆写主要包括对值类型的覆写和对引用类型的覆写，同时也要区别基类是否已经有过覆写和不曾覆写两种情况，并以等价原则为前提，进行判断。在此，我们仅提供较为标准的实现方法，具体的实现取决于不同的类型定义和语义需求。

```
class EqualsEx
{
   //定义值类型成员 ms
   private MyStruct ms;
   //定义引用类型成员 mc
   private MyClass mc;
   public override bool Equals(object obj)
   {
      //为 null，则必不相等
```

```
        if (obj == null) return false;
        //引用判等为真，则二者必定相等
        if (ReferenceEquals(this, obj)) return true;
        //类型判断
        EqualsEx objEx = obj as EqualsEx;
        if (objEx == null) return false;
        //最后是成员判断，分值类型成员和引用类型成员
        //通常可以提供强类型的判等方法来单独处理对各个成员的判等
        return EqualsHelper(this, objEx);
    }
    private static bool EqualsHelper(EqualsEx objA, EqualsEx objB)
    {
        //值类型成员判断
        if (!objA.ms.Equals(objA.ms)) return false;
        //引用类型成员判断
        if (!Equals(objA.mc, objB.mc)) return false;
        //最后，才可以判定两个对象是相等的
        return true;
    }
}
```

上述示例只是从标准化的角度来阐释 Equals 覆写的简单实现，而实际应用时又会有所不同，然而总结起来实现 Equals 方法我们应该着力于以下几点：首先，检测 obj 是否为 null，如果是则必然不相等；然后，以 ReferenceEquals 来判等是否引用相等，这种办法比较高效，因为引用相等即可以推出值相等；然后，再进行类型判断，不同类型的对象一定不相等；最后，也是最复杂的一个过程，即对对象的各个成员进行比较，引用类型进行恒定性判断，值类型进行恒等性判断。在本例中我们将成员判断封装为一个专门的处理方法 EqualsHelper，以隔离对类成员的判断实现，主要有以下几个好处：

- 符合 Extract Method 原则，以隔离相对变化的操作。
- 提供了强类型版本的 Equals 实现，对于值类型成员来说还可以避免不必要的装箱操作。

l 为==操作符提供了重载实现的安全版本。

在.NET 框架中，System.String 类型的 Equals 覆写方法就提供了 EqualsHelper 方法来实现。

8.2.4 与 GetHashCode 方法同步

GetHashCode 方法，用于获取对象的哈希值，以应用于哈希算法、加密和校验等操作中。相同的对象必然具有相同的哈希值，因此 GetHashCode 的行为依赖于 Equals 方法进行判断，在覆写 Equals 方法时，也必须覆写 GetHashCode，以同步二者在语义上的统一。例如：

```
public class Person
{
   //每个人有唯一的身份证号，因此可以作为 Person 的标识码
   private string id = null;
   private string name = null;
   //以 id 作为哈希码是可靠的, 而 name 则有可能相同
   public override int GetHashCode()
   {
      return id.GetHashCode();
   }
   public override bool Equals(object obj)
   {
      if(ReferenceEquals(this, obj)) return true;
      Person person = obj as Person;
      if(person == null) return false;
      //Equals 也以用户身份证号作为判等依据
      if(this.id == person.id) return true;
      return false;
   }
}
```

二者的关系可以表达为：如果 x.Equals(y)为 true 成立，则必有 x.GetHashCode() == y.GetHashCode()成立。如果覆写了 Equals 而没有实现 GetHashCode，C#编译器会给出没有覆写 GetHashCode 的警告。

8.2.5 规则

- 值相等还是引用相等决定于具体的需求，Equals 方法的覆写实现也决定于类型想要实现的判等逻辑。
- 几个判等方法相互引用，所以对某个方法的覆写可能会影响其他方法的执行结果。
- 如果覆写了 Equals 虚方法，则必须重新实现 GetHashCode 方法，使二者保持同步。
- 禁止从 Equals 方法或者"=="操作符抛出异常，应该在 Equals 内部首先避免 null 引用异常，要么相等要么不等。
- ReferenceEquals 方法主要用于判别两个对象的唯一性，比较两个值类型则一定返回 false。
- ReferenceEquals 方法比较两个 System.String 类型的唯一性时，要注意 String 类型的特殊性：字符串驻留。
- 实现 ICompare 接口的类型必须重新实现 Equals 方法。
- 值类型最好重新实现 Equals 方法和重载==操作符，因为默认情况下实现的是引用相等。

8.2.6 结论

四种判等方法，各有用途又相互关联。这是 CLR 提供给我们关于对象等值性和唯一性的执行机制。分，我们以不同角度来了解其本质；合，我们以规则来阐释其关联。在本质和关联之上，充分体会.NET 这种抽象而又灵活的判等机制，留下更多的思考来认识这种精妙的设计。

8.3 如此特殊：大话 String

本节将介绍以下内容：

— String 类型解析

— 字符串恒定与字符串驻留

— StringBuilder 应用与对比

8.3.1 引言

String 类型很特殊，算是.NET 大家庭中少有的异类，它是如此的与众不同，使我们无法忽视它的存在。本节就是这样一篇关于 String 类型及其特殊性讨论的话题，通过逐层解析来解密 System.String 类型。

那么，String 究竟特殊在哪里？

- 创建特殊性：String 对象不以 newobj 指令创建，而是 ldstr 指令创建。在实现机制上，CLR 给了特殊照顾来优化其性能。
- String 类型是.NET 中不变模式的经典应用，在 CLR 内部由特定的控制器来专门处理 String 对象。
- 应用上，String 类型表现为值类型语义；内存上，String 类型实现为引用类型，存储在托管堆中。
- 两次创建内容相同的 String 对象可以指向相同的内存地址。
- String 类型被实现为密封类，不可在子类中继承。
- String 类型是跨应用程序域的，可以在不同的应用程序域中访问同一 String 对象。

然而，将 String 类型认清看透并非易事，根据上面的特殊问题，我们给出具体的答案，为 String 类型的各个难点解惑，最后再给出应用的常见方法和典型操作。

8.3.2 字符串创建

string 类型是 C#基元类型，对应于 FCL 中的 System.String 类型，是.NET 中使用最频繁，应用最广泛的基本类型之一。其创建与实例化过程非常简单，在操作方式上类似与其他基元类型 int、char 等，例如：

```
string mystr = "Hello";
```

分析 IL 可知，CLR 使用 ldstr 指令从元数据中获取文本常量来加载字符串，而以典型的 new 方式来创建：

```
String mystr2 = new String("Hello");
```

会导致编译错误。因为 System.String 只提供了数个接受 Char*、Char[]类型的构造函数，例如：

```
Char[] cs = {'a', 'b', 'c'};
String strArr = new String(cs);
```

在.NET 中很少使用构造器方式来创建 string 对象，更多的方式还是以加载字符常量的方式来完成，关于 String 类型的创建，我们在 3.4 节“经典指令解析之实例创建”中已有详细的本质分析，详细情况请参阅。

8.3.3 字符串恒定性

字符串恒定性（Immutability），是指字符串一经创建，就不可改变。这是 String 对象最为重要的特性之一，是 CLR 高度集成 String 以提高其性能的考虑。具体而言，字符串一旦创建，就会在托管堆上分配一块连续的内存空间，我们对其的任何改变都不会影响到原 String 对象，而是重新创建出新的 String 对象，例如：

```
public static void Main()
{
   string str = "This is a test about immutablitity of string type.";
   Console.WriteLine(str.Insert(0, "Hi, ").Substring(19).ToUpper());
   Console.WriteLine(str);
}
```

在上例中，我们对 str 对象完成一系列的修改：增加、取子串和大写格式改变等操作，从结果输出上来看 str 依然保持原来的值不变。而 Insert、Substring 和 ToUpper 方法都会创建出新的临时字符串，而这些新对象不被其他代码所引用，因此成为下次垃圾回收的目标，从而造成了性能上的损失。

之所以特殊化处理 String 具有恒定性的特点，源于 CLR 对其的处理机制：String 类型是不变模式在.NET 中的典型应用，String 对象从应用角度体现了值类型语义，而从内存角度实现为引用类型存储，位于托管堆。

对象恒定性，为程序设计带来了极大的好处，主要包括为：

- 保证对 String 对象的任意操作不会改变原字符串。
- 恒定性还意味着操作字符串不会出现线程同步问题。
- 恒定性一定程度上，成就了字符串驻留。

对象恒定性，还意味着 String 类型必须为密封类，例如 String 类型的定义为：

```
public sealed class String : IComparable, ICloneable, IConvertible, Icomparable <string>, IEnumerable<char>, IEnumerable, IEquatable<string>
```

如果可以在子类中继承 String 类型，则必然有可能破坏 CLR 对 String 类型的特殊处理机制，也会破坏 String 类型的恒定性。

8.3.4 字符串驻留

关于字符串驻留，我们以一个简单的示例开始：

```
class StringInterning
{
   public static void Main()
   {
      string strA = "abcdef";
      string strB = "abcdef";
      Console.WriteLine(ReferenceEquals(strA, strB));
      string strC = "abc";
      string strD = strC + "def";
      Console.WriteLine(ReferenceEquals(strA, strD));
      strD = String.Intern(strD);
      Console.WriteLine(ReferenceEquals(strA, strD));
```

```
    }
}
//执行结果：
//True
//False
//True
```

上述示例，会给我们三个意外，也是关于执行结果的意外：首先，strA 和 strB 为两个不同的 String 对象，按照一般的分析两次创建的不同对象，CLR 将为其在托管堆分配不同的内存块，而 ReferenceEquals 方法用于判断两个引用是否指向同一对象实例，从结果来看 strA 和 strB 显然指向了同一内存地址；其次，strD 和 strA 在内容上也是一样的，然而其 ReferenceEquals 方法返回的结果为 False，显然 strA 和 strD 并没有指向相同的内存块；最后，以静态方法 Intern 操作 strD 后，二者又指向了相同的对象，ReferenceEquals 方法又返回 True。

要想解释以上疑惑，只有请字符串驻留（String Interning）登场了。下面我们通过对字符串驻留技术的分析，来一步一步解开上述示例的种种疑惑。

缘起

String 类型区别于其他类型的最大特点是其恒定性。对字符串的任何操作，包括字符串比较，字符串链接，字符串格式化等会创建新的字符串，从而伴随着性能与内存的双重损耗。而 String 类型本身又是.NET 中使用最频繁、应用最广泛的基本类型，因此 CLR 有必要有针对性的对其性能问题，采取特殊的解决办法。

事实上，CLR 以字符串驻留机制来解决这一问题：对于相同的字符串，CLR 不会为其分别分配内存空间，而是共享同一内存。因此，有两个问题显得尤为重要：

- 一方面，CLR 必须提供特殊的处理结构，来维护对相同字符串共享内存的机制。
- 另一方面，CLR 必须通过查找来添加新构造的字符串对象到其特定结构中。

的确如此，CLR 内部维护了一个哈希表（Hash Table）来管理其创建的大部分 string 对象。其中，Key 为 string 本身，而 Value 为分配给对应的 string 的内存地址。我们以一个简单的图例（图 8-1）来说明这一问题。

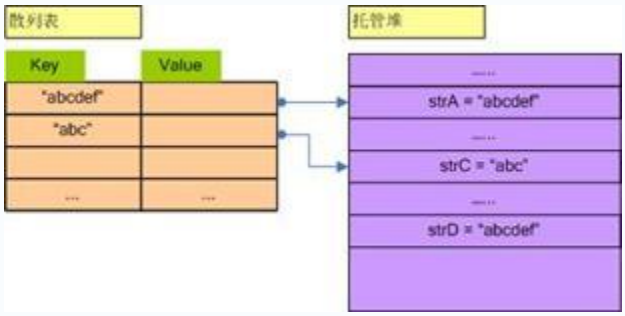


图 8-1 string 的内存概况

细节

我们一步一步分析上述示例的执行过程，然后才能从总体上对字符串驻留机制有所了解。

```
string strA = "abcdef";
```

CLR 初始化时，会创建一个空哈希表，当 JIT 编译方法时，会首先在哈希表中查找每一个字符串常量，显然第一次它不会找到任何“abcdef”常量，因此会在托管堆中创建一个新的 string 对象 strA，并在哈希表中创建一个 Key-Value 对，将“abcdef”串赋给 Key，而将 strA 对象的引用赋给 Value，也就是说 Value 内保持了指向“abcdef”字符串在托管堆中的引用地址。这样就完成了第一次字符串的创建过程。

```
string strB = "abcdef";
```

程序接着运行，JIT 根据“abcdef”在哈希表中逐个查找，结果找到了该字符串，所以 JIT 不会执行任何操作，只是把找到的 Key-Value 对的 Value 值赋给 strB 对象。由此可知，strA 和 strB 具有相同的内存引用，所以 ReferenceEquals 方法当然返回 true。

```
string strC = "abc";
string strD = strC + "def";
```

接着，JIT 以类似的过程来向哈希表中添加了“abc”字符串，并将引用返回给 strC 对象；但是 strD 对象的创建过程又有所区别，因为 strD 是动态生成的字符串，这样的字符串是不会被添加到哈希表中维护的，因此以 ReferenceEquals 来比较 strA 和 strD 会返回 false。

对于动态生成的字符串，因为没有添加到 CLR 内部维护的哈希表而使字符串驻留机制失效。但是，当我们需要高效的比较两个字符串是否相等时，可以手工启用字符串驻留机制，这就是调用 String 类型的两个静态方法，它们是：

```
public static string Intern(string str);
public static string IsInterned(string str);
```

二者的处理机制都是在哈希表中查找是否存在 str 参数字符串，如果找到就返回已存在的 String 对象的引用，否则 Intern 方法将该 str 字符串添加到哈希表中，并返回引用；而 IsInterned 方法则不会向哈希表中添加字符串，而只是返回 null。例如，

```
strD = String.Intern(strD);
Console.WriteLine(ReferenceEquals(strA, strD));
```

我们就很容易解释上述代码的执行结果了。

补充

综上所述，当一个引用字符串的方法被编译时，所有的字符串常量都会被以这种方式添加到该哈希表中，但是动态生成的字符串并未执行字符串驻留机制。值得注意的是，下面的代码执行结果又会有所不同：

```
public static void Main()
{
   string strA = "abcdef";
   string strC = "abc";
   string strD = strC + "def";
   Console.WriteLine(ReferenceEquals(strA, strD));
   string strE = "abc" + "def";
   Console.WriteLine(ReferenceEquals(strA, strE));
}
```

由结果可知，strA 和 strD 指向不同的对象；而 strA 与 strE 指向相同的对象。我们将上述代码翻译为 IL 代码：

```
 IL_0001:  ldstr      "abcdef"
```

```
IL_0006:  stloc.0
IL_0007:  ldstr      "abc"
IL_000c:  stloc.1
IL_000d:  ldloc.1
IL_000e:  ldstr      "def"
IL_0013:  call       string [mscorlib]System.String::Concat(string,
                                                            string)
```

.....部分省略.....

```
IL_0026:  ldstr      "abcdef"
IL_002b:  stloc.3
```

由 IL 分析可知，动态生成字符串时，CLR 调用了 System::Concat 来执行字符串链接；而直接赋值 strE = “abc” + “def”的操作，编译器会自动将其连接为一个文本常量加载，因此会添加到内部哈希表中，这也是为什么最后 strA 和 strE 指向同一对象的原因了。

最后，需要特别指出的是：字符串驻留是进程级的，可以跨应用程序域（AppDomain）而存在。垃圾回收不能释放哈希表中引用的字符串对象，只有进程结束这些对象才会被释放。因此，String 类型的特殊性还表现在同一个字符串对象可以在不同的应用程序域中被访问，从而突破了 AppDomain 的隔离机制，其原因还是源于字符串的恒定性，因为是不可变的，所以根本没有必要再隔离。

8.3.5 字符串操作典籍

本节从几个相对孤立的角度来描述 String 类型，包括了不同操作、常用方法和典型问题几个方面。

1. 字符串类型与其他基元类型的转换

String 类型可以与其他基本类型直接进行转换，在此以 System.Double 类型与 System.String 类型的转换为例，来简要说明二者转换的几个简单的方法及其区别。

Double 类型转换为 String 类型：

```
Double num = 123.456;
```

```
string str = num.ToString();
```

Double 类型覆写了 ToString 方法用于返回对象的值。

String 类型转换为 Double 类型，有多种方法可供选择：

```
string str = "123.456";
Double num= 0.0;
num = Double.Parse(str);
Double.TryParse(str, out num);
num = Convert.ToDouble(str);
```

这三种方法的区别主要是对异常的处理机制上：如果转换失败，则 Parse 方法总会抛出异常，主要包括 ArgumentNullException、OverflowException、FormatException 等；TryParse 则不会抛出任何异常，而返回 false 标志解析失败；Convert 方法在 str 为 null 时不会抛出异常，而是返回 0。

其他的基元类型，例如 Int32、Char、Byte、Boolean、Single 等均提供了上述方法实现与 String 类型进行一定程度的转换，同时对于特定的格式化转换可以参考上述方法的各个重载版本，限于篇幅，此不赘述。

2. 转义字符和字面字符串

l 使用转义字符来实现特定格式字符串

对于在 C++等语言中熟悉的转义字符串，在.NET 中同样适用，例如 C＃语言提供了相应的实现版本：

```
string strName = "Name:\n\t\"小雨\"";
```

上述示例实现了回车和 Tab 空格操作，并为“小雨”添加了双引号。

l 在文件和目录路径、数据库连接字符串和正则表达式中广泛应用的字面字符串（verbatim string），为 C#提供了声明字符串的特殊方式，用于将引号之间的所有字符视为字符串的一部分，例如：

```
string strPath = @"C:\Program Files \Mynet.exe";
```

上述代码，完全等效于：

```
string strPath = "C:\\Program Files \\Mynet.exe";
```

而以下代码则导致被提示“无法识别的转义序列”的编译错误：

```
string strPath = "C:\Program Files \Mynet.exe";
```

显然，以@实现的字面字符串更具可读性，克服了转义字符串带来的阅读障碍。

3. 关于 string 和 System.String

string 与 System.String 常常使很多初学者感到困惑。实际上，string 和 System.String 编译为 IL 代码时，会生成完全相同的代码。那么关于 string 和 System.String 我们应该了解的是其概念上的细微差别。

- string 为 C#语言的基元类型，类似于 int、char 和 long 等其他 C#基元类型，基元类型简化了语言代码，带来简便的可读性，不同高级语言对同一基元类型的标识符可能有所不同。
- System.String 是框架类库（FCL）的基本类型，string 和 System.String 有直接的映射关系。
- 从 IL 角度来看，string 和 System.String 之间没有任何不同。同样的情况，还存在于其他的基元类型，例如：int 和 System.Int32，long 和 System.Int64，float 和 System.Single，以及 object 和 System.Object 等。

4. String 类型参数的传递问题

有一个足以引起关注的问题是，String 类型作为参数传递时，以按值传递和按引用传递时所表现的不同：

```
class StringArgument
{
   public static void Main()
   {
      string strA = "String A";
      string strB = "String B";
      //参数为 String 类型的按值传递（strA）和按引用传递（strB）
      ChangeString(strA, ref strB);
      Console.WriteLine(strA);
```

```
        Console.WriteLine(strB);
    }
    private static void ChangeString(string stra, ref string strb)
    {
        stra = "Changing String A";
        strb = "Changing String B";
    }
}
//执行结果
//String A
//Changing String B
```

String 作为典型的引用类型，其作为参数传递也代表了典型的引用类型按值传递和按引用传递的区别，可以小结为：

- 默认情况为按值传递，strA 参数所示，传递 strA 的值，也就是指向“String A”的引用；
- ref 标识了按引用传递，strB 参数所示，传递的是原引用的引用，也就是传递一个到 strB 本身的引用，这区别于到“String B”的引用这个概念，二者不是相同的概念。

因此，默认情况下，string 类型也是按值传递的，只是这个“值”是指向字符串实例的引用而已，关于参数传递的详细描述请参考 4.3 节“参数之惑---传递的艺术”。

5. 其他常用方法

表 8-1 对 System.String 的常用方法做以简单说明，而不以示例展开，这些方法广泛的应用在平常的字符串处理操作中，因此有必要做以说明。

表 8-1 System.String 类型的常用方法

常用方法	方法说明
ToString	ToString 方法是 System.Object 提供的虚方法，用于返回对象的字符串表达形式，可以获取格式化或者带有语言文化信息的实例信息
SubString	用于获取子字符串，FCL 提供了两个重载版本，可以指定起始位置和长度
Split	返回包含此实例中由指定 Char 或者 String 元素隔开的子字符串的 String 数组
StartsWith、EndsWith	StartsWith 用于判断字符串是否以指定内容开始；而 EndsWith 用于判断字符串是否以指定内容结尾
ToUpper、ToLower	ToUpper 用于返回实例的大写版本；而 ToLower 用于返回实例的小写版本
IndexOf、LastIndexOf	IndexOf 用于返回匹配项的第一个的索引位置；LastIndexOf 用于返回匹配项的最后一个索引位置
Insert、Remove	Insert 用于向指定位置插入指定的字符串；Remove 用于从实例中删除指定个数的字符串
Trim、TrimStart、TrimEnd	Trim 方法用于从实例开始和末尾位置，移除指定字符的所有匹配项；TrimStart 用于从实例开始位置，移除指定字符的所有匹配项；TrimEnd 用于从实例

	结束位置，移除指定字符的所有匹配项
Copy、CopyTo	Copy 为静态方法，CopyTo 为实例方法，都是用于拷贝实例内容给新的 String 对象。其中 CopyTo 方法可以指定起始位置，拷贝个数等信息
Compare、CompareOrdinal、CompareTo	Compare 为静态方法，用于返回两个字符串间的排序情况，并且允许指定语言文化信息；CompareOrdinal 为静态方法，按照字符串中的码值比较字符集，并返回比较结果，为 0 表示结果相等，为负表示第一个字符串小，为正表示第一个字符串大；而 CompareTo 是实例方法，用于返回两个字符串的排序，不允许指定语言文化信息，因为该方法总是使用当前线程相关联的语言文化信息
Concat、Join	均为静态方法。Concat 用于连接一个或者多个字符串；Join 用于以指定分隔符来串联 String 数组的各个元素，并返回新的 String 实例
Format	静态方法。用于格式化 String 对象为指定的格式或语言文化信息

8.3.6 补充的礼物：StringBuilder

String 对象是恒定不变的，而 System.Text.StringBuilder 对象表示的字符串是可变的。StringBuilder 是.NET 提供的动态创建 String 对象的高效方式，以克服 String 对象恒定性带来的性能影响，克服了对 String 对象进行多次修改带来的创建大量 String 对象的问题。因此，我们首先将二者的执行性能做以简单的比较：

```
public static void Main()
{
    #region 性能比较
    Stopwatch sw = Stopwatch.StartNew();
    //String 性能测试
    string str = "";
    for (int i = 0; i < 10000; i++)
        str += i.ToString();
    sw.Stop();
    Console.WriteLine(sw.ElapsedMilliseconds);
    //StringBuilder 性能测试
    sw.Reset();
    sw.Start();
    StringBuilder sb = new StringBuilder();
    for (int i = 0; i < 10000; i++)
        sb.Append(i.ToString());
    sw.Stop();
    Console.WriteLine(sw.ElapsedMilliseconds);
    #endregion
```

```
}
//执行结果
//422
//3
```

创建同样的字符串过程，执行结果有百倍之多的性能差别，而且这种差别会随着累加次数的增加而增加。因此，基于性能的考虑，我们应该尽可能使用 StringBuilder 来动态创建字符串，然后以 ToString 方法将其转换为 String 对象应用。StringBuilder 内部有一个指向 Char 数值的字段，StringBuilder 正是通过操作该字符数组而实现高效的处理机制。

1. 构造 StringBuilder

StringBuilder 对象的实例化没有什么特殊可言，与其他对象实例化一样，典型的构造方式为：

```
StringBuilder sb = new StringBuilder("Hello, word.", 20);
```

其中，第二个参数表示容量，也就是 StringBuilder 所维护的字符数组的长度，默认为 16，可以设定其为合适的长度来避免不必要的垃圾回收；还有一个概念为最大容量，表示字符串所能容纳字符的最大个数，默认为 Int32.MaxValue，对象创建时一经设定就不可更改；字符串长度表示当前 StringBuilder 对象的字符数组长度，可以使用 Length 属性来获取和设定当前的 StringBuilder 长度。

2. StringBuilder 的常用方法

（1）ToString 方法

返回一个 StringBuilder 中字符数组字段的 String，因为不必拷贝字符数组，所以执行效率很高，是最常用的方法之一。不过，值得注意的是，在调用了 StringBuilder 的 ToString 方法之后，都会导致 StringBuilder 重新分配和创建新的字符数组，因为 ToString 方法返回的 String 必须是恒定的。

（2）Append/AppendFormat 方法

用于将文本或者对象字符串添加到当前 StringBuilder 字符数组中，例如：

```
StringBuilder sbs = new StringBuilder("Hello, ");
sbs.Append("Word.");
Console.WriteLine(sbs);
```

```
//执行结果
//Hello, Word.
```

而 AppendFormat 方法进一步实现了 IFormattable 接口，可接受 IFormatProvider 类型参数来实现可格式化的字符串信息，例如：

```
StringBuilder formatStr = new StringBuilder("The price is ");
formatStr.AppendFormat("{0:C}", 22);
formatStr.AppendFormat("\r\nThe Date is {0:D}", DateTime.Now.Date);
Console.WriteLine(formatStr);
```

（3）Insert 方法

用于将文本或字符串对象添加到指定位置，例如：

```
StringBuilder mysb = new StringBuilder("My name XiaoWang");
mysb.Insert(8, "is ");
Console.WriteLine(mysb);
//执行结果
//My name is XiaoWang
```

（4）Replace 方法

Replace 方法是一种重要的字符串操作方法，用来将字符串数组中的一个字符或字符串替换为另外一个字符或字符串，例如：

```
StringBuilder sb = new StringBuilder("I love game.");
sb.Replace("game", ".NET");
Console.WriteLine(sb);
//执行结果
//I love .NET.
```

限于篇幅，我们不再列举其他方法，例如 Remove、Equals、AppendLine 等，留于读者自己来探索 StringBuilder 带来的快捷操作。

3. 再论性能

StringBuilder 有诸多的好处，是否可以代替 String 呢？基于这个问题我们有如下的对比性分析：

- String 是恒定的；而 StringBuilder 是可变的。
- 对于简单的字符串连接操作，在性能上 StringBuilder 不一定总是优于 String。因为 StringBuilder 对象的创建代价较大，在字符串连接目标较少的情况下，过度滥用 StringBuilder 会导致性能的浪费而非节约。只有大量的或者无法预知次数的字符串操作，才考虑以 StringBuilder 来实现。事实上，本节开始的示例如果将连接次数设置为一百次以内，就根本看不出二者的性能差别。
- String 类型的“＋”连接操作，实际上是重载操作符“＋”调用 String.Concat 来操作，而编译器则会优化这种连接操作的处理，编译器根据其传入参数的个数，一次性分配相应的内存，并依次拷入相应的字符串。
- StringBuilder 在使用上，最好指定合适的容量值，否则由于默认容量不足而频繁的进行内存分配操作，是不妥的实现方法。
- 通常情况下，进行简单字符串连接时，应该优先考虑使用 String.Concat 和 String.Join 等操作来完成字符串的连接，但是应该留意 String.Concat 可能存在的装箱操作。

8.3.7 结论

最后，回答为什么特殊？

String 类型是所有系统中使用最频繁的类型，以致于 CLR 必须考虑为其实现特定的实现方式，例如 System.Object 基类就提供了 ToString 虚方法，一切.NET 类型都可以使用 ToString 方法来获取对象的字符串表达。因此，String 类型紧密地集成于 CLR，CLR 可以直接访问 String 类型的内存布局，以一系列解决方案来优化其执行。

8.4 简易不简单：认识枚举

本节将介绍以下内容：

— 枚举类型全解

— 位标记应用

— 枚举应用规则

8.4.1 引言

在哪里可以看到枚举？打开每个文件的属性，我们会看到只读、隐藏的选项；操作一个文件时，你可以采用只读、可写、追加等模式；设置系统级别时，你可能会选择紧急、普通和不紧急来定义。这些各式各样的信息中，一个共同的特点是信息的状态分类相对稳定，在.NET 中可以选择以类的静态字段来表达这种简单的分类结构，但是更明智的选择显然是：枚举。

事实上，在.NET 中有大量的枚举来表达这种简单而稳定的结构，FCL 中对文件属性的定义为 System.IO.FileAttributes 枚举，对字体风格的定义为 System.Drawing.FontStyle 枚举，对文化类型定义为 System.Globlization.CultureType 枚举。除了良好的可读性、易于维护、强类型的优点之外，性能的考虑也占了一席之地。

关于枚举，在本节会给出详细而全面的理解，认识枚举，从一点一滴开始。

8.4.2 枚举类型解析

1. 类型本质

所有枚举类型都隐式而且只能隐式地继承自 System.Enum 类型，System.Enum 类型是继承自 System.ValueType 类型唯一不为值类型的引用类型。该类型的定义为：

```
public abstract class Enum : ValueType, IComparable, IFormattable, IConvertible
```

从该定义中，我们可以得出以下结论：

- System.Enum 类型是引用类型，并且是一个抽象类。
- System.Enum 类型继承自 System.ValueType 类型，而 ValueType 类型是一切值类型的根类，但是显然 System.Enum 并非值类型，这是 ValueType 唯一的特例。
- System.Enum 类型实现了 IComparable、IFormattable 和 IConvertible 接口，因此枚举类型可以与这三个接口实现类型转换。

.NET 之所以在 ValueType 之下实现一个 Enum 类型，主要是实现对枚举类型公共成员与公共方法的抽象，任何枚举类型都自动继承了 Enum 中实现的方法。关于枚举类型与 Enum 类型的关

系，可以表述为：枚举类型是值类型，分配于线程的堆栈上，自动继承于 Enum 类型，但是本身不能被继承；Enum 类型是引用类型，分配于托管堆上，Enum 类型本身不是枚举类型，但是提供了操作枚举类型的共用方法。

下面我们根据一个枚举的定义和操作来分析其 IL，以从中获取关于枚举的更多认识：

```
enum LogLevel
{
   Trace,
   Debug,
   Information,
   Warnning,
   Error,
   Fatal
}
```

将上述枚举定义用 Reflector 工具翻译为 IL 代码，对应为：

```
.class private auto ansi sealed LogLevel
   extends [mscorlib]System.Enum
{
   .field public static literal valuetype InsideDotNet.Framework.EnumEx.LogLevel Debug = int32(1)
   .field public static literal valuetype InsideDotNet.Framework.EnumEx.LogLevel Error = int32(4)
   .field public static literal valuetype InsideDotNet.Framework.EnumEx.LogLevel Fatal = int32(5)
   .field public static literal valuetype InsideDotNet.Framework.EnumEx.LogLevel Information = int3
2(2)
   .field public static literal valuetype InsideDotNet.Framework.EnumEx.LogLevel Trace = int32(0)
   .field public specialname rtspecialname int32 value__
   .field public static literal valuetype InsideDotNet.Framework.EnumEx.LogLevel Warnning = int32
(3)
}
```

从上述 IL 代码中，LogLevel 枚举类型的确继承自 System.Enum 类型，并且编译器自动为各个成员映射一个常数值，默认从 0 开始，逐个加 1。因此，在本质上枚举就是一个常数集合，各个成员常量相当于类的静态字段。

然后，我们对该枚举类型进行简单的操作，以了解其运行时信息，例如：

```
public static void Main()
{
   LogLevel logger = LogLevel.Information;
   Console.WriteLine("The log level is {0}.", logger);
}
```

该过程实例化了一个枚举变量，并将它输出到控制台，对应的 IL 为：

```
.method public hidebysig static void Main() cil managed
{
   .entrypoint
   .maxstack 2
   .locals init (
      [0] valuetype InsideDotNet.Framework.EnumEx.LogLevel logger)
   L_0000: nop
   L_0001: ldc.i4.2
   L_0002: stloc.0
   L_0003: ldstr "The log level is {0}."
   L_0008: ldloc.0
   L_0009: box InsideDotNet.Framework.EnumEx.LogLevel
   L_000e: call void [mscorlib]System.Console::WriteLine(string, object)
   L_0013: nop
   L_0014: ret
}
```

分析 IL 可知，首先将 2 赋值给 logger，然后执行装箱操作（L_0009），再调用 WriteLine 方法将结果输出到控制台。

2. 枚举规则

讨论了枚举的本质，我们再回过头来，看看枚举类型的定义及其规则，例如下面的枚举定义略有不同：

```
enum Week: int
{
   Sun = 7,
   Mon = 1,
   Tue,
   Wed,
   Thur,
   Fri,
   Sat,
   Weekend = Sun
}
```

根据以上定义，我们了解关于枚举的种种规则，这些规则是定义枚举和操作枚举的基本纲领，主要包括：

- 枚举定义时可以声明其基础类型，例如本例 Week 枚举的基础类型指明为 int 型，默认情况时即为 int。通过指定类型限定了枚举成员的取值范围，而被指定为枚举声明类型的只能是除 char 外的 8 种整数类型：byte、sbyte、short、ushort、int、uint、long 和 ulong，声明其他的类型将导致编译错误，例如 Int16、Int64。
- 枚举成员是枚举类型的命名常量，任意两个枚举常量不能具有同样的名称符号，但是可以具有相同的关联值。
- 枚举成员会显式或者隐式与整数值相关联，默认情况下，第一个元素对应的隐式值为 0，然后各个成员依次递增 1。还可以通过显式强制指定，例如 Sun 为 7，Mon 为 1，而 Tue 则为 2，并且成员 Weekend 和 Sun 则关联了相同的枚举值。
- 枚举成员可以自由引用其他成员的设定值，但是一定注意避免循环定义，否则将引发编译错误，例如：

```
enum MusicType
```

```
{
   Blue,
   Jazz = Pop,
   Pop
}
```

编译器将无法确知成员 Jazz 和 Pop 的设定值到底为多少。

- 枚举是一种特殊的值类型，不能定义任何的属性、方法和事件，枚举类型的属性、方法和事件都继承自 System.Enum 类型。
- 枚举类型是值类型，可以直接通过赋值进行实例化，例如：

```
Week myweek = Week.Mon;
```

也可以以 new 关键字来实例化，例如：

```
Week myweek = new Week();
```

值得注意的是，此时 myweek 并不等于 Week 枚举类型中定义的第一个成员的 Sun 的关联值 7，而是等效于字面值为 0 的成员项。如果枚举成员不存在 0 值常数，则 myweek 将默认设定为 0，可以从下面代码来验证这一规则：

```
enum WithZero
{
   First = 1,
   Zero = 0
}
enum WithNonZero
{
   First = 1,
   Second
}
class EnumMethod
{
   public static void Main()
   {
      WithZero wz = new WithZero();
      Console.WriteLine(wz.ToString("G"));
      WithNonZero wnz = new WithNonZero();
      Console.WriteLine(wnz.ToString("G"));
   }
}
//执行结果
```

```
//Zero
//0
```

因此，以 new 关键字来实例化枚举类型，并非好的选择，通常情况下我们应该避免这种操作方式。

l 枚举可以进行自增自减操作，例如：

```
Week day = (Week)3;
day++;
Console.WriteLine(day.ToString());
```

通过自增运算，上述代码输出结果将为：Fri。

8.4.3 枚举种种

1. 类型转换

（1）与整型转换

因为枚举类型本质上是整数类型的集合，因此可以与整数类型进行相互的类型转换，但是这种转换必须是显式的。

```
//枚举转换为整数
int i = (int)Week.Sun;
//将整数转换为枚举
Week day = (Week)3;
```

另外，Enum 还实现了 Parse 方法来间接完成整数类型向枚举类型的转换，例如：

```
//或使用 Parse 方法进行转换
Week day = (Week)Enum.Parse(typeof(Week), "2");
```

（2）与字符串的映射

枚举与 String 类型的转换，其实是枚举成员与字符串表达式的相互映射，这种映射主要通过 Enum 类型的两个方法来完成：

- ToString 实例方法，将枚举类型映射为字符串表达形式。可以通过指定格式化标志来输出枚举成员的特定格式，例如“G”表示返回普通格式、“X”表示返回 16 进制格式，而本例中的“D”则表示返回十进制格式。
- Parse 静态方法，将整数或者符号名称字符串转换为等效的枚举类型，转换不成功则抛出 ArgumentException 异常，例如：

```
Week myday = (Week)Enum.Parse(typeof(Week), "Mon", true);
Console.WriteLine(myday);
```

因此，Parse 之前最好应用 IsDefined 方法进行有效性判断。对于关联相同整数值的枚举成员，Parse 方法将返回第一个关联的枚举类型，例如：

```
Week theDay = (Week)Enum.Parse(typeof(Week), "7");
Console.WriteLine(theDay.ToString());
//执行结果
//Sun
```

（3）不同枚举的相互转换

不同的枚举类型之间可以进行相互转换，这种转换的基础是枚举成员本质为整数类型的集合，因此其过程相当于将一种枚举转换为值，然后再将该值映射到另一枚举的成员。

```
MusicType mtToday = MusicType.Jazz;
Week today = (Week)mtToday;
```

（4）与其它引用类型转换

除了可以显式的与 8 种整数类型进行转换之外，枚举类型是典型的值类型，可以向上转换为父级类和实现的接口类型，而这种转换实质发生了装箱操作。小结枚举可装箱的类型主要包括：System.Object、System.ValueType、System.Enum、System.IComparable、System.IFormattable 和 System.IConvertible。例如：

```
IConvertible iConvert = (IConvertible)MusicType.Jazz;
Int32 x = iConvert.ToInt32(CultureInfo.CurrentCulture);
Console.WriteLine(x);
```

1. 常用方法

System.Enum 类型为枚举类型提供了几个值得研究的方法，这些方法是操作和使用枚举的利器，由于 System.Enum 是抽象类，Enum 方法大都是静态方法，在此仅举几个简单的例子点到为止。

以 GetNames 和 GetValues 方法分别获取枚举中符号名称数组和所有符号的数组，例如：

```
//由 GetName 获取枚举常数名称的数组
foreach (string item in Enum.GetNames(typeof(Week)))
{
    Console.WriteLine(item.ToString());
}
//由 GetValues 获取枚举常数值的数组
foreach (Week item in Enum.GetValues(typeof(Week)))
{
    Console.WriteLine("{0} : {1}", item.ToString("D"), item.ToString());
}
```

应用 GetValues 方法或 GetNames 方法，可以很容易将枚举类型与数据显式控件绑定来显式枚举成员，例如：

```
ListBox lb = new ListBox();
lb.DataSource = Enum.GetValues(typeof(Week));
this.Controls.Add(lb);
```

以 IsDefined 方法来判断符号或者整数存在于枚举中，以防止在类型转换时的越界情况出现。

```
if(Enum.IsDefined(typeof(Week), "Fri"))
{
    Console.WriteLine("Today is {0}.", Week.Fri.ToString("G"));
}
```

以 GetUnderlyingType 静态方法，返回枚举实例的声明类型，例如：

```
Console.WriteLine(Enum.GetUnderlyingType(typeof(Week)));
```

8.4.4 位枚举

位标记集合是一种由组合出现的元素形成的列表，通常设计为以“位或”运算组合新值；枚举类型则通常表达一种语义相对独立的数值集合。而以枚举类型来实现位标记集合是最为完美的组合，简称为位枚举。在.NET 中，需要对枚举常量进行位运算时，通常以 System.FlagsAttribute 特性来标记枚举类型，例如：

```
[Flags]
enum ColorStyle
{
   None = 0x00,
   Red = 0x01,
   Orange = 0x02,
   Yellow = 0x04,
   Greeen = 0x08,
   Blue = 0x10,
   Indigotic = 0x20,
   Purple = 0x40,
   All = Red | Orange | Yellow | Greeen | Blue | Indigotic | Purple
}
```

FlagsAttribute 特性的作用是将枚举成员处理为位标记，而不是孤立的常数，例如：

```
public static void Main()
{
   ColorStyle mycs = ColorStyle.Red | ColorStyle.Yellow | ColorStyle.Blue;
   Console.WriteLine(mycs.ToString());
}
```

在上例中，mycs 实例的对应数值为 21（十六进制 0x15），而覆写的 ToString 方法在 ColorStyle 枚举中找不到对应的符号。而 FlagsAttribute 特性的作用是将枚举常数看成一组位标记来操作，从而影响 ToString、Parse 和 Format 方法的执行行为。在 ColorStyle 定义中 0x15 显然由 0x01、0x04 和 0x10 组合而成，示例的结果将返回：Red, Yellow, Blue，而非 21，原因正在于此。

位枚举首先是一个枚举类型，因此具有一般枚举类型应有的所有特性和方法，例如继承于 Enum 类型，实现了 ToString、Parse、GetValues 等方法。但是由于位枚举的特殊性质，因此应用于某些方法时，应该留意其处理方式的不同之处。这些区别主要包括：

- Enum.IsDefined 方法不能应对位枚举成员，正如前文所言位枚举区别与普通枚举的重要表现是：位枚举不具备排他性，成员之间可以通过位运算进行组合。而 IsDefined 方法只能应对已定义的成员判断，而无法处理组合而成的位枚举，因此结果将总是返回 false。例如：

```
Enum.IsDefined(typeof(ColorStyle), 0x15)
Enum.IsDefined(typeof(ColorStyle), "Red, Yellow, Blue")
```

MSDN 中给出了解决位枚举成员是否定义的判断方法：就是将该数值与枚举成员进行“位与”运算，结果不为 0 则表示该变量中包含该枚举成员，例如：

```
if ((mycs & ColorStyle.Red) != 0)
   Console.WriteLine(ColorStyle.Red + " is in ColorStyle");
```

- Flags 特性影响 ToString、Parse 和 Format 方法的执行过程和结果。
- 如果不使用 FlagsAttribute 特性来标记位枚举，也可以在 ToString 方法中传入“F”格式来获得同样的结果，以“D”、“G”等标记来格式化处理，也能获得相应的输出格式。
- 在位枚举中，应该显式的为每个枚举成员赋予有效的数值，并且以 2 的幂次方为单位定义枚举常量，这样能保证实现枚举常量的各个标志不会重叠。当然你也可以指定其它的整数值，但是应该注意指定 0 值作为成员常数值时，“位与”运算将总是返回 false。

8.4.5 规则与意义

- 枚举类型使代码更具可读性，理解清晰，易于维护。在 Visual Stuido 2008 等编译工具中，良好的智能感知为我们进行程序设计提供了更方便的代码机制。同时，如果枚举符号和对应的整数值发生变化，只需修改枚举定义即可，而不必在漫长的代码中进行修改。
- 枚举类型是强类型的，从而保证了系统安全性。而以类的静态字段实现的类似替代模型，不具有枚举的简单性和类型安全性。例如：

```
public static void Main()
```

```
{
    LogLevel log = LogLevel.Information;
    GetCurrentLog(log);
}
private static void GetCurrentLog(LogLevel level)
{
    Console.WriteLine(level.ToString());
}
```

试图为 GetCurrentLog 方法传递整数或者其他类型参数将导致编译错误，枚举类型保证了类型的安全性。

- 枚举类型的默认值为 0，因此，通常给枚举成员包含 0 值是有意义的，以避免 0 值游离于预定义集合，导致枚举变量保持非预定义值是没有意义的。另外，位枚举中与 0 值成员进行“位与”运算将永远返回 false，因此不能将 0 值枚举成员作为“位与”运算的测试标志。
- 枚举的声明类型，必须是基于编译器的基元类型，而不能是对应的 FCL 类型，否则将导致编译错误。

8.4.6 结论

枚举类型在 BCL 中占有一席之地，说明了.NET 框架对枚举类型的应用是广泛的。本节力图从枚举的各个方面建立对枚举的全面认知，通过枚举定义、枚举方法和枚举应用几个角度来阐释一个看似简单的概念，对枚举的理解与探索更进了一步。

8.5 一脉相承：委托、匿名方法和 Lambda 表达式

本节将介绍以下内容：

— 委托

— 事件

— 匿名方法

— Lambda 表达式

8.5.1 引言

委托，实现了类型安全的回调方法。在.NET 中回调无处不在，所以委托也无处不在，事件模型建立在委托机制上，Lambda 表达式本质上就是一种匿名委托。本节中将完成一次关于委托的旅行，全面阐述委托及其核心话题，逐一梳理委托、委托链、事件、匿名方法和 Lambda 表达式。

8.5.2 解密委托

1. 委托的定义

了解委托，从其定义开始，通常一个委托被声明为：

```
public delegate void CalculateDelegate(Int32 x, Int32 y);
```

关键字 delegate 用于声明一个委托类型 CalculateDelegate，可以对其添加访问修饰符，默认其返回值类型为 void，接受两个 Int32 型参数 x 和 y，但是委托并不等同与方法，而是一个引用类型，类似于 C++中的函数指针，稍后在委托本质里将对此有所交代。

下面的示例将介绍如何通过委托来实现一个计算器模拟程序，在此基础上来了解关于委托的定义、创建和应用：

```
class DelegateEx
{
   //声明一个委托
   public delegate void CalculateDelegate(Int32 x, Int32 y);
   //创建与委托关联的方法，二者具有相同的返回值类型和参数列表
   public static void Add(Int32 x, Int32 y)
   {
      Console.WriteLine(x + y);
   }
```

```
    //定义委托类型变量
    private static CalculateDelegate myDelegate;
    public static void Main()
    {
        //进行委托绑定
        myDelegate = new CalculateDelegate(Add);
        //回调 Add 方法
        myDelegate(100, 200);
    }
}
```

上述示例，在类 DelegateEx 内部声明了一个 CalculateDelegate 委托类型，它具有和关联方法 Add 完全相同的返回值类型和参数列表，否则将导致编译时错误。将方法 Add 传递给 CalculateDelegate 构造器，也就是将方法 Add 指派给 CalculateDelegate 委托，并将该引用赋给 myDelegate 变量，也就表示 myDeleage 变量保存了指向 Add 方法的引用，以此实现对 Add 的回调。

由此可见，委托表示了对其回调方法的签名，可以将方法当作参数进行传递，并根据传入的方法来动态的改变方法调用。只要为委托提供相同签名的方法，就可以与委托绑定，例如：

```
public static void Subtract(Int32 x, Int32 y)
{
    Console.WriteLine(x - y);
}
```

同样，可以将方法 Subtract 分配给委托，通过参数传递实现方法回调，例如：

```
public static void Main()
{
    //进行委托绑定
    myDelegate = new CalculateDelegate(Subtract);
    myDelegate(100, 200);
}
```

2. 多播委托和委托链

在上述委托实现中，Add 方法和 Subtract 可以绑定于同一个委托类型 myDelegate，由此可以很容易想到将多个方法绑定到一个委托变量，在调用一个方法时，可以依次执行其绑定的所有方法，这种技术称为多播委托。在.NET 中提供了相当简洁的语法来创建委托链，以+=和-=操作符分别进行绑定和解除绑定的操作，多个方法绑定到一个委托变量就形成一个委托链，对其调用时，将会依次调用所有绑定的回调方法。例如：

```
public static void Main()
{
    myDelegate = new CalculateDelegate(Add);
    myDelegate += new CalculateDelegate(Subtract);
    myDelegate += new CalculateDelegate(Multiply);
    myDelegate(100, 200);
}
```

上述执行将在控制台依次输出 300、-100 和 20000 三个结果，可见多播委托按照委托链顺序调用所有绑定的方法，同样以-=操作可以解除委托链上的绑定，例如：

```
myDelegate -= new CalculateDelegate(Add);
myDelegate(100, 200);
```

结果将只有-100 和 20000 被输出，可见通过-=操作解除了 Add 方法。

事实上，+=和-=操作分别调用了 Deleagate.Combine 和 Deleagate.Remove 方法，由对应的 IL 可知：

```
.method public hidebysig static void  Main() cil managed
{
  .entrypoint
  // 代码大小       151 (0x97)
  .maxstack  4
  IL_0000:  nop
  IL_0001:  ldnull
  IL_0002:  ldftn      void
InsideDotNet.NewFeature.CSharp3.DelegateEx::Add(int32, int32)
  //部分省略.....
  IL_0023:  call       class [mscorlib]System.Delegate [mscorlib]System.Delegate:: Combine(class [ms
corlib]System.Delegate, class [mscorlib]System.Delegate)
  //部分省略.....
```

```
  IL_0043:  call       class [mscorlib]System.Delegate [mscorlib]System.Delegate:: Combine(class [ms
corlib]System.Delegate, class [mscorlib]System.Delegate)
  //部分省略……
  IL_0075:  call       class [mscorlib]System.Delegate [mscorlib]System.Delegate:: Remove(class [ms
corlib]System.Delegate, class [mscorlib]System.Delegate)
  //部分省略……
  IL_0095:  nop
  IL_0096:  ret
} // end of method DelegateEx::Main
```

所以，上述操作实际等效于：

```
public static void Main()
{
   myDelegate = (CalculateDelegate)Delegate.Combine(new CalculateDelegate(Add),
      new CalculateDelegate(Subtract), new CalculateDelegate(Multiply));
   myDelegate(100, 200);
   myDelegate = (CalculateDelegate)Delegate.Remove(myDelegate,
      new CalculateDelegate(Add));
   myDelegate(100, 200);
}
```

另外，多播委托返回值一般为 void，委托类型为非 void 类型时，多播委托将返回最后一个调用的方法的执行结果，所以在实际的应用中不被推荐。

3. 委托的本质

委托在本质上仍然是一个类，如此简洁的语法正是因为 CLR 和编译器在后台完成了一系列操作，将上述 CalculateDelegate 委托编译为 IL，你将会看得更加明白如图 8-2 所示。

图 8-2 CalculateDelegate 的 IL 分析

所以，委托本质上仍旧是一个类，该类继承自 System.MulticastDelegate 类，该类维护一个带有链接的委托列表，在调用多播委托时，将按照委托列表的委托顺序而调用的。还包括一个接受两个参数的构造函数和 3 个重要方法：BeginInvoke、EndInvoke 和 Invoke。

首先来了解 CalculateDelegate 的构造函数，它包括了两个参数：第一个参数表示一个对象引用，它指向了当前委托调用回调函数的实例，在本例中即指向一个 DelegateEx 对象；第二个参数标识了回调方法，也就是 Add 方法。因此，在创建一个委托类型实例时，将会为其初始化一个指向对象的引用和一个标识回调方法的整数，这是由编译器完成的。那么一个回调方法是如何被执行的，继续以 IL 代码来分析委托的调用，即可显露端倪（在此仅分析委托关联 Add 方法时的情况）：

```
.method public hidebysig static void  Main() cil managed
{
  .entrypoint
  // 代码大小       37 (0x25)
  .maxstack  8
  IL_0000:  nop
  IL_0001:  ldnull
  IL_0002:  ldftn      void
InsideDotNet.NewFeature.CSharp3.DelegateEx::Add(int32, int32)
  IL_0008:  newobj     instance void InsideDotNet.NewFeature.CSharp3.DelegateEx/ CalculateDelega
te::.ctor(object, native int)
  IL_000d:  stsfld     class InsideDotNet.NewFeature.CSharp3.DelegateEx/ CalculateDelegate InsideD
otNet.NewFeature.CSharp3.DelegateEx::myDelegate
  IL_0012:  ldsfld     class
InsideDotNet.NewFeature.CSharp3.DelegateEx/Calculate Delegate InsideDotNet.NewFeature.CSharp
3.DelegateEx::myDelegate
  IL_0017:  ldc.i4.s   100
  IL_0019:  ldc.i4     0xc8
  IL_001e:  callvirt   instance void InsideDotNet.NewFeature.CSharp3.DelegateEx/ CalculateDelegat
e::Invoke(int32, int32)
  IL_0023:  nop
```

```
 IL_0024:  ret
} // end of method DelegateEx::Main
```

在 IL 代码中可见，首先调用 CalculateDelegate 的构造函数来创建一个 myDelegate 实例，然后通过 CalculateDelegate::Invoke 执行回调方法调用，可见真正执行调用的是 Invoke 方法。因此，你也可以通过 Invoke 在代码中显示调用，例如：

```
myDelegate.Invoke(100, 200);
```

其执行过程和隐式调用是一样的，注意在.NET 1.0 中 C#编译器是不允许显示调用的，以后的版本中修正了这一限制。

另外，Invoke 方法直接对当前线程调用回调方法，在异步编程环境中，除了 Invoke 方法，也会生成 BeginInvoke 和 EndInvoke 方法来完成一定的工作。这也就是委托类中另外两个方法的作用。

8.5.3 委托和事件

.NET 的事件模型建立在委托机制之上，透彻的了解了委托才能明白的分析事件。可以说，事件是对委托的封装，从委托的示例中可知，在客户端可以随意对委托进行操作，一定程度上破坏了面向的对象的封装机制，因此事件实现了对委托的封装。

下面，通过将委托的示例进行改造，来完成一个事件的定义过程：

```
public class Calculator
{
   //定义一个 CalculateEventArgs，
   //用于存放事件引发时向处理程序传递的状态信息
   public class CalculateEventArgs: EventArgs
   {
      public readonly Int32 x, y;
      public CalculateEventArgs(Int32 x, Int32 y)
      {
         this.x = x;
         this.y = y;
```

```
        }
    }
    //声明事件委托
    public delegate void CalculateEventHandler(object sender,CalculateEventArgs e);
    //定义事件成员，提供外部绑定
    public event CalculateEventHandler MyCalculate;
    //提供受保护的虚方法，可以由子类覆写来拒绝监视
    protected virtual void OnCalculate(CalculateEventArgs e)
    {
        if (MyCalculate != null)
        {
            MyCalculate(this, e);
        }
    }
    //进行计算，调用该方法表示有新的计算发生
    public void Calculate(Int32 x, Int32 y)
    {
        CalculateEventArgs e = new CalculateEventArgs(x, y);
        //通知所有的事件的注册者
        OnCalculate(e);
    }
}
```

示例中，对计算器模拟程序做了简要的修改，从二者的对比中可以体会事件的完整定义过程，主要包括：

- 定义一个内部事件参数类型，用于存放事件引发时向事件处理程序传递的状态信息，EventArgs 是事件数据类的基类。
- 声明事件委托，主要包括两个参数：一个表示事件发送者对象，一个表示事件参数类对象。
- 定义事件成员。

- 定义负责通知事件引发的方法，它被实现为 protected virtual 方法，目的是可以在派生类中覆写该方法来拒绝监视事件。
- 定义一个触发事件的方法，例如 Calculate 被调用时，表示有新的计算发生。

一个事件的完整程序就这样定义好了。然后，还需要定义一个事件触发程序，用来监听事件：

```
//定义事件触发者
public class CalculatorManager
{
   //定义消息通知方法
   public void Add(object sender, Calculator.CalculateEventArgs e)
   {
      Console.WriteLine(e.x + e.y);
   }
   public void Substract(object sender, Calculator.CalculateEventArgs e)
   {
      Console.WriteLine(e.x - e.y);
   }
}
```

最后，实现一个事件的处理程序：

```
public class Test_Calculator
{
   public static void Main()
   {
      Calculator calculator = new Calculator();
      //事件触发者
      CalculatorManager cm = new CalculatorManager();
      //事件绑定
      calculator.MyCalculate += cm.Add;
      calculator.Calculate(100, 200);
      calculator.MyCalculate += cm.Substract;
```

```
        calculator.Calculate(100, 200);
        //事件注销
        calculator.MyCalculate -= cm.Add;
        calculator.Calculate(100, 200);
    }
}
```

如果对设计模式有所了解，上述实现过程实质是 Observer 模式在委托中的应用，在.NET 中对 Observer 模式的应用严格的遵守了相关的规范。在 Windows Form 程序开发中，对一个 Button 的 Click 就对应了事件的响应，例如：

```
this.button1.Click += new System.EventHandler(this.button1_Click);
```

用于将 button1_Click 方法绑定到 button1 的 Click 事件上，当有按钮被按下时，将会触发执行 button1_Click 方法：

```
private void button1_Click(object sender, EventArgs e)
{
}
```

8.5.4 匿名方法

匿名方法以内联方式放入委托对象的使用位置，而避免创建一个委托来关联回调方法，也就是由委托调用了匿名的方法，将方法代码和委托实例直接关联，在语法上有简洁和直观的好处。例如以匿名方法来绑定 Click 事件将变得非常简单：

```
button1.Click += delegate
{
    MessageBox.Show("Hello world.");
};
```

因此，有必要以匿名方法来实现本节开始的委托示例，了解其实现过程和底层实质，例如：

```
class AnonymousMethodEx
{
    delegate void CalculateDelegate(Int32 x, Int32 y);
    public static void Main()
    {
        //匿名方法
        CalculateDelegate mySubstractDelegate = delegate(Int32 x, Int32 y)
```

```
        {
            Console.WriteLine(x - y);
        };
        CalculateDelegate myAddDelegate = delegate(Int32 x, Int32 y)
        {
            Console.WriteLine( x + y);
        };
        mySubstractDelegate(100, 200);
    }
}
```

事实上，匿名方法和委托在 IL 层是等效的，编译器为匿名方法增加了两个静态成员和静态方法，如图 8-3 所示。

图 8-3 匿名方法的 IL 分析

由编译器生成的两个静态成员和静态方法，辅助实现了委托调用一样的语法结构，这正是匿名方法在底层的真相。

8.5.5 Lambda 表达式

Lambda 表达式是 Functional Programming 的核心概念，现在 C# 3.0 中也引入了 Lambda 表达式来实现更加简洁的语法，并且为 LINQ 提供了语法基础，这些将在本书第 12 章有所交代。再次应用 Lambda 表达式来实现相同的过程，其代码为：

```
class LambdaExpressionEx
{
    delegate void CalculateDelegate(Int32 x, Int32 y);
    public static void Main()
    {
        CalculateDelegate myDelegate = (x, y) => Console.WriteLine(x - y);
        myDelegate(100, 200);
    }
```

}

分析 Lambda 表达式的 IL 代码，可知编译器同样自动生成了相应的静态成员和静态方法，Lambda 表达式在本质上仍然是一个委托。带来这一切便利的是编译器，在此对 IL 上的细节不再做进一步分析。

8.5.6 规则

- 委托实现了面向对象的，类型安全的方法回调机制。
- 以 Delegate 作为委托类型的后缀，以 EventHandle 作为事件委托的后缀，是规范的命名规则。
- 多播委托返回值一般为 void，不推荐在多播委托中返回非 void 的类型。
- 匿名方法和 Lambda 表达式提供了更为简洁的语法表现，而这些新的特性主要是基于编译器而实现的，在 IL 上并没有本质的变化。
- .NET 的事件是 Observer 模式在委托中的应用，并且基于.NET 规范而实现，体现了更好的耦合性和灵活性。

8.5.7 结论

从委托到 Lambda 表达式的逐层演化，我们可以看到.NET 在语言上的不断进化和发展，也正是这些进步促成了技术的向前发展，使得.NET 在语言上更加地兼容和优化。对于技术开发人员而言，这种进步也正是我们所期望的。

然而，从根本上了解委托、认识委托才是一切的基础，否则语法上的进化只能使得理解更加迷惑。本节的讨论，意在为理解这些内容提供基础，建立一个较为全面的概念。

8.6 直面异常

本节将介绍以下内容：

— .NET 异常机制

— .NET 常见的异常类型

— 自定义异常

8.6.1 引言

内存耗尽、索引越界、访问已关闭资源、堆栈溢出、除零运算等一个个摆在你面前的时候，你想到的是什么呢？当然是，异常。

在系统容错和程序规范方面，异常机制是不可或缺的重要因素和手段。当挑战来临的时候，良好的系统设计必定有良好的异常处理机制来保证程序的健壮性和容错机制。然而对异常的理解往往存在或多或少的误解，例如：

- 异常就是程序错误，以错误代码返回错误信息就足够了。
- 在系统中异常越多越能保证容错性，尽可能多的使用 try/catch 块来处理程序执行。
- 使用.NET 自定义 Exception 就能捕获所有的异常信息，不需要特定异常的处理块。
- 将异常类作为方法参数或者返回值。
- 在自定义异常中通过覆写 ToString 方法报告异常信息，对这种操作不能掉以轻心，因为某些安全敏感信息有泄漏的可能。

希望读者在从本节的脉络上了解异常的基本情况和通用规则，将更多的探索留于实践中的体察和品味。

8.6.2 为何而抛？

关于异常，最常见的误解可能莫过于对其可用性的理解。对于异常的处理，基本有两种方式来完成：一种是异常形式，一种是返回值形式。然而，不管是传统 Win32 API 下习惯的 32 位错误代码，还是 COM 编程中的 HRESULT 返回值，异常机制所具有的优势都不可替代，主要表现为：

- 很多时候，返回值方式具有固有的局限性，例如在构造函数中就无法有效的应用返回值来返回错误信息，只有异常才能提供全面的解决方案来应对。

- 提供更丰富的异常信息，便于交互和调试，而传统的错误代码不能有效提供更多的异常信息和调试指示，在程序理解和维护方面异常机制更具优势。

- 有效实现异常回滚，并且可以根据不同的异常，回滚不同的操作，有效实现了对系统稳定性与可靠性的控制。例如，下例实现了一个典型的事务回滚操作：

```
public void ExcuteSql(string conString, string cmdString)
{
   SqlConnection con = new SqlConnection(conString);
   try
   {
      con.Open();
      SqlTransaction tran = con.BeginTransaction();
      SqlCommand cmd = new SqlCommand(cmdString, con);
      try
      {
         cmd.ExecuteNonQuery();
         tran.Commit();
      }
      catch (SqlException ex)
      {
         Console.WriteLine(ex.Message);
         //实现事务回滚
         tran.Rollback();
         throw new Exception("SQL Error!", ex);
      }
   }
   catch(Exception e)
   {
      throw (e);
   }
   finally
   {
      con.Close();
   }
}
```

- 很好地与面向对象语言集成，在.NET 中异常机制已经很好地与高级语言集成在一起，以异常 System.Exception 类建立起的体系结构已经能够轻松应付各种异常信息，并且可以通过面向对象机制定义自己的特定异常处理类，实现更加特性化的异常信息。

- 错误处理更加局部化，错误代码更集中地放在一起，增强了代码的理解和维护，例如资源清理的工作完全交由 finally 子句来执行，不必花费过多的精力去留意其维护。

- 错误代码返回的信息内容有限而难于理解，一连串数字显然不及丰富的文字信息说明问题，同时也不利于快速地定位和修改需要调试的代码。
- 异常机制能有效应对未处理的异常信息，我们不可能轻易地忽略任何异常；而返回值方式不可能深入到异常可能发生的各个角落，不经意的遗漏就会造成系统的不稳定，况且这种维护方式显然会让系统开发人员精疲力竭。
- 异常机制提供了实现自定义异常的可能，有利于实现异常的扩展和特色定制。

综上所述，异常机制是处理系统异常信息的最好机制与选择，Jeffrey Richter 在《Microsoft .NET 框架程序设计》一书中给出了异常本质的最好定义，那就是：

> 异常是对程序接口隐含假设的一种违反。

然而关于异常的焦虑常常突出在其性能对系统造成的压力上，因为返回值方式的性能毋庸置疑更具“先天”的优势。那么异常的性能问题，我们又该如何理解呢？

本质上，CLR 会为每个可执行文件创建一个异常信息表，在该表中每个方法都有一个关联的异常处理信息数组，数组的每一项描述一个受保护的代码块、相关联的异常筛选器（后文介绍）和异常处理程序等。在没有异常发生时，异常信息表在处理时间和内存上的损失几乎可以忽略，只有异常发生时这种损失才值得考虑。例如：

```
class TestException
{
   //测试异常处理的性能
   public int TestWithException(int a, int b)
   {
      try
      {
         return a / b;
      }
      catch
      {
         return -1;
      }
```

```
    }
    //测试非异常处理的性能
    public int TestNoExceptioin(int a, int b)
    {
        return a / b;
    }
}
```

上述代码对应的 IL 更能说明其性能差别，首先是有异常处理的方法：

```
.method public hidebysig instance int32  TestWithException(int32 a,
                                                           int32 b) cil managed
{
  // 代码大小       17 (0x11)
  .maxstack  2
  .locals init ([0] int32 CS$1$0000)
  IL_0000:  nop
  .try
  {
    IL_0001:  nop
    IL_0002:  ldarg.1
    IL_0003:  ldarg.2
    IL_0004:  div
    IL_0005:  stloc.0
    IL_0006:  leave.s    IL_000e
  }  // end .try
  catch [mscorlib]System.Object
  {
    IL_0008:  pop
    IL_0009:  nop
    IL_000a:  ldc.i4.m1
    IL_000b:  stloc.0
    IL_000c:  leave.s    IL_000e
```

```
  } // end handler
  IL_000e:  nop
  IL_000f:  ldloc.0
  IL_0010:  ret
} // end of method TestException::TestWithException
```

代码大小为 17 个字节，在不发生异常的情况下，数据在 IL_0006 出栈后以 leave.s 指令退出 try 受保护区域，并继续执行 IL_000e 后面的操作：压栈并返回。

然后是不使用异常的情形：

```
.method public hidebysig instance int32  TestNoExceptioin(int32 a,
                                                          int32 b) cil managed
{
  // 代码大小       9 (0x9)
  .maxstack  2
  .locals init ([0] int32 CS$1$0000)
  IL_0000:  nop
  IL_0001:  ldarg.1
  IL_0002:  ldarg.2
  IL_0003:  div
  IL_0004:  stloc.0
  IL_0005:  br.s       IL_0007
  IL_0007:  ldloc.0
  IL_0008:  ret
} // end of method TestException::TestNoExceptioin
```

代码大小为 9 字节，没有特别处理跳出受保护区域的操作。

由此可见，两种方式在内存的消化上差别很小，只有 8 个字节。而实际运行的时间差别也微不足道，所以没有异常引发的情况下，异常处理的性能损失是很小的；然而，有异常发生的情况下，必须承认异常处理将占用大量的系统资源和执行时间，因此建议尽可能的以处理流程来规避异常处理。

8.6.3 从 try/catch/finally 说起：解析异常机制

理解.NET 的异常处理机制，以 try/catch/finally 块的应用为起点，是最好的切入口，例如：

```
class BasicException
{
   public static void Main()
   {
      int a = 1;
      int b = b;
      GetResultToText(a, 0);
   }
   public static void GetResultToText(int a, int b)
   {
      StreamWriter sw = null;
      try
      {
         sw = File.AppendText(@"E:\temp.txt");
         int c = a / b;
         //将运算结果输出到文本
         sw.WriteLine(c.ToString());
         Console.WriteLine(c.ToString());
      }
      catch (DivideByZeroException)
      {
         //实现从 DivideByZeroException 恢复的代码
         //并重新给出异常提示信息
         throw new DivideByZeroException ("除数不能为零！");
      }
      catch (FileNotFoundException ex)
      {
         //实现从 IOException 恢复的代码
```

```
            //并再次引发异常信息
            throw(ex);
        }
        catch (Exception ex)
        {
            //实现从任何与 CLS 兼容的异常恢复的代码
            //并重新抛出
            throw;
        }
        catch
        {
            //实现任何异常恢复的代码，无论是否与 CLS 兼容
            //并重新抛出
            throw;
        }
        finally
        {
            sw.Flush();
            sw.Close();
        }
        //未有异常抛出，或者 catch 捕获而未抛出异常，
        //或 catch 块重新抛出别的异常，此处才被执行
        Console.WriteLine("执行结束。");
    }
}
```

1. try 分析

try 子句中通常包含可能导致异常的执行代码，而 try 块通常执行到引发异常或成功执行完成为止。它不能单独存在，否则将导致编译错误，必须和零到多个 catch 子句或者 finally 子句配合使用。其中，catch 子句包含各种异常的响应代码，而 finally 子句则包含资源清理代码。

2. catch 分析

catch 子句包含了异常出现时的响应代码，其执行规则是：一个 try 子句可以关联零个或多个 catch 子句，CLR 按照自上而下的顺序搜索 catch 块。catch 子句包含的表达式，该表达式称为异常筛选器，用于识别 try 块引发的异常。如果筛选器识别该异常，则会执行该 catch 子句内的响应代码；如果筛选器不接受该异常，则 CLR 将沿着调用堆栈向更高一层搜索，直到找到识别的筛选器为止，如果找不到则将导致一个未处理异常。不管是否执行 catch 子句，CLR 最终都会执行 finally 子句的资源清理代码。因此编译器要求将特定程度较高的异常放在前面（如 DivideByZeroExceptio n 类），而将特定程度不高的异常放在后面（如示例中最下面的 catch 子句可以响应任何异常），依此类推，其他 catch 子句按照 System.Exception 的继承层次依次由底层向高层罗列，否则将导致编译错误。

catch 子句的执行代码通常会执行从异常恢复的代码，在执行末尾可以通过 throw 关键字再次引发由 catch 捕获的异常，并添加相应的信息通知调用端更多的信息内容；或者程序实现为线程从捕获异常的 catch 子句退出，然后执行 finally 子句和 finally 子句后的代码，当然前提是二者存在的情况下。

关于：异常筛选器

异常筛选器，用于表示用户可预料、可恢复的异常类，所有的异常类必须是 System. Exception 类型或其派生类，System.Excetpion 类型是一切异常类型的基类，其他异常类例如 DivideByZeroException、FileNotFoundException 是派生类，从而形成一个有继承层次的异常类体系，越具体的异常类越位于层次的底层。

如果 try 子句未抛出异常，则 CLR 将不会执行任何 catch 子句的响应代码，而直接转向 finally 子句执行直到结束。

值得注意的是，finally 块之后的代码段不总是被执行，因为在引发异常并且没有被捕获的情况下，将不会执行该代码。因此，对于必须执行的处理环节，必须放在 finally 子句中。

3. finally 分析

异常发生时，程序将转交给异常处理程序，意味着那些总是希望被执行的代码可能不被执行，例如文件关闭、数据库连接关闭等资源清理工作，例如本例的 StreamWriter 对象。异常机制提供

了 finally 子句来解决这一问题：无论异常是否发生，finally 子句总是执行。因此，finally 子句不总是存在，只有需要进行资源清理操作时，才有必要提供 finally 子句来保证清理操作总是被执行，否则没有必要提供“多余”的 finally 子句。

finally 在 CLR 按照调用堆栈执行完 catch 子句的所有代码时执行。一个 try 块只能对应一个 finally 块，并且如果存在 catch 块，则 finally 块必须放在所有的 catch 块之后。如果存在 finally 子句，则 finally 子句执行结束后，CLR 会继续执行 finally 子句之后的代码。

根据示例我们对 try、catch 和 finally 子句分别做了分析，然后对其应用规则做以小结，主要包括：

- catch 子句可以带异常筛选器，也可以不带任何参数。如果不存在任何表达式，则表明该 catch 子句可以捕获任何异常类型，包括兼容 CLS 的异常或者不兼容的异常。
- catch 子句按照筛选器的继承层次进行顺序罗列，如果将具体的异常类放在执行顺序的末尾将导致编译器异常。而对于继承层次同级的异常类，则可以随意安排 catch 子句的先后顺序，例如 DivideByZeroException 类和 FileNotFoundException 类处于 System.Exception 继承层次的同一层次，因此其对应的 catch 子句之间可以随意安排先后顺序。
- 异常筛选器，可以指定一个异常变量，该变量将指向抛出的异常类对象，该对象记录了相关的异常信息，可以在 catch 子句内获取该信息。
- finally 子句内，也可以抛出异常，但是应该尽量避免这种操作。
- CLR 如果没有搜索到合适的异常筛选器，则说明程序发生了未预期的异常，CLR 将抛出一个未处理异常，应用程序应该提供对未处理异常的应对策略，例如：在发行版本中将异常信息写入日志，而在开发版本中启用调试器定位。
- try 块内定义的变量对 try 块外是不可见的，因此对于 try 块内进行初始化的变量，应该定义在 try 块之前，否则 try 块外的调用将导致编译错误。例如示例中的 StreamWriter 的对象定义，一定要放在 try 块之外，否则无法在 finally 子句内完成资源清理操作。

8.6.4 .NET 系统异常类

1. 异常体系

.NET 框架提供了不同层次的异常类来应对不同种类的异常，并且形成一定的继承体系，所有的异常类型都继承自 System.Exception 类。例如，图 8-4 是异常继承层次的一个片段，继承自上而下由通用化向特定化延伸。

FCL 定义了一个庞大的异常体系，熟悉和了解这些异常类型是有效应用异常和理解异常体系的有效手段，但是显然这一工作只能交给搜索 MSDN 来完成了。然而，我们还是应该对一些重要的.NET 系统异常有一定的了解，主要包括：

- OverflowException，算术运算、类型转换时的溢出。

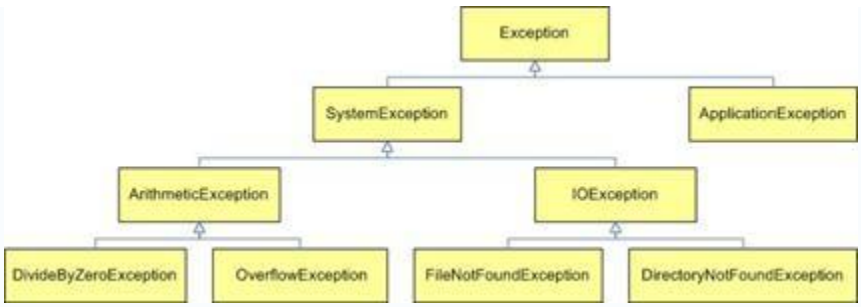


图 8-4 异常类的部分继承体系

- StackOverflowException，密封类，不可继承，表示堆栈溢出，在应用程序中抛出该异常是不适当的做法，因为一般只有 CLR 本身会抛出堆栈溢出的异常。
- OutOfMemoryException，内存不足引发的异常。
- NullReferenceException，引用空引用对象时引发。
- InvalidCastException，无效类型转换引发。
- IndexOutOfRangeException，试图访问越界的索引而引发的异常。
- ArgumentException，无效参数异常。
- ArgumentNullException，给方法传递一个不可接受的空参数的空引用。
- DivideByZeroException，被零除引发。
- ArithmeticException，算术运行、类型转换等引发的异常。

l FileNotFoundException，试图访问不存在的文件时引发。

注意，这里罗列的并非全部的常见异常，更非 FCL 定义的所有系统异常类型。对于异常类而言，更多的精力应该放在关注异常基类 System.Exception 的理解上，以期提纲挈领。

2. System.Exception 类解析

关于 System.Exception 类型，它是一切异常类的最终基类，而它本身又继承自 System.Object 类型，用于捕获任何与 CLS 兼容的异常。Exception 类提供了所有异常类型的基本属性与规则，例如：

l Message 属性，用于描述异常抛出原因的文本信息。

l InnerException 属性，用于获取导致当前异常的异常集。

l StackTrack 属性，提供了一个调用栈，其中记录了异常最初被抛出的位置，因此在程序调试时非常有用，例如：

```
public static void Main()
{
   try
   {
      TestException();
   }
   catch (Exception ex)
   {
      //输出当前调用堆栈上的异常的抛出位置
      Console.WriteLine(ex.StackTrace);
   }
}
private static void TestException()
{
   //直接抛出异常
   throw new FileNotFoundException("Error.");
}
```

- HResult 受保护属性，可读写 HRESULT 值，分配特定异常的编码数值，主要应用于托管代码与非托管代码的交互操作。

还有其他的方法，例如 HelpLink 用于获取帮助文件的链接，TargetSite 方法用于获取引发异常的方法。

还有很多公有方法辅助完成异常信息的获取、异常类序列化等操作。其中，实现 ISerializable 接口方法 GetObjectData 值得关注，异常类新增字段必须通过该方法填充 SerializationInfo，异常类进行序列化和反序列化必须实现该方法，其定义可表示为：

```
[ComVisible(true)]
public interface ISerializable
{
   [SecurityPermission(SecurityAction.LinkDemand, Flags = SecurityPermission Flag.SerializationFor
matter)]
   void GetObjectData(SerializationInfo info, StreamingContext context);
}
```

参数 info 表示要填充的 SerializationInfo 对象，而 context 则表示要序列化的目标流。我们在下文的自定义异常中将会有所了解。

.NET 还提供了两个直接继承于 Exception 的重要子类：ApplicationException 和 SystemException 类。其中，ApplicationException 类型为 FCL 为应用程序预留的基类型，所以自定义异常可以选择 ApplicationException 或者直接从 Exception 继承；SystemException 为系统异常基类，CLR 自身抛出的异常继承自 SystemException 类型。

8.6.5 定义自己的异常类

FCL 定义的系统异常，不能解决所有的问题。异常机制与面向对象有效的集成，意味着我们可以很容易的通过继承 System.Exception 及其派生类，来实现自定义的错误处理，扩展异常处理机制。

上文中，我们简单学习了 System.Exception 类的实现属性和方法，应该说研究 Exception 类型对于实现自定义异常类具有很好的参考价值，微软工程师已经实现了最好的实现体验。我们以实际的示例出发，来说明自定义异常类的实现，总结其实现与应用规则，首先是自定义异常类的实现：

```
//Serializable 指定了自定义异常可以被序列化
[Serializable]
public class MyException : Exception, ISerializable
{
   //自定义本地文本信息
   private string myMsg;
   public string MyMsg
   {
      get { return myMsg; }
   }
   //重写只读本地文本信息属性
   public override string Message
   {
      get
      {
         string msgBase = base.Message;
         return myMsg == null ? msgBase : msgBase + myMsg;
      }
   }
   //实现基类的各公有构造函数
   public MyException()
      : base(){ }
   public MyException(string message)
      : base(message) { }
   public MyException(string message, Exception innerException)
      : base(message, innerException) { }
   //为新增字段实现构造函数
```

```
    public MyException(string message, string myMsg)
        : this(message)
    {
        this.myMsg = myMsg;
    }
    public MyException(string message, string myMsg, Exception innerException)
        : this(message, innerException)
    {
        this.myMsg = myMsg;
    }
    //用于序列化的构造函数，以支持跨应用程序域或远程边界的封送处理
    protected MyException(SerializationInfo info, StreamingContext context)
        : base(info, context)
    {
        myMsg = info.GetString("MyMsg");
    }
    //重写基类 GetObjectData 方法，实现向 SerializationInfo 中添加自定义字段信息
    public override void GetObjectData(SerializationInfo info, StreamingContext context)
    {
        info.AddValue("MyMsg", myMsg);
        base.GetObjectData(info, context);
    }
}
```

然后，我们实现一个自定义异常测试类，来进一步了解.NET 异常机制的执行过程：

```
class Test_CustomException
{
    public static void Main()
    {
        try
        {
            try
```

```
        {
            string str = null;
            Console.WriteLine(str.ToString());
        }
        catch (NullReferenceException ex)
        {
            //向高层调用方抛出自定义异常
            throw new MyException("这是系统异常信息。", "\n 这是自定义异常信息。", ex);
        }
    }
    catch (MyException ex)
    {
        Console.WriteLine(ex.Message);
    }
  }
}
```

结合示例的实践，总结自定义异常类的规则与规范，主要包括：

- 首先，选择合适的基类继承，一般情况下我们都会选择 Exception 类或其派生类作为自定义异常类的基类。但是异常的继承深度不宜过多，一般在 2～3 层是可接受的维护范围。
- System.Exception 类型提供了三个公有构造函数，在自定义类型中也应该实现三个构造函数，并且最好调用基类中相应的构造函数；如果自定义类型中有新的字段要处理，则应该为新的字段实现新的构造函数来实现。
- 所有的异常类型都是可序列化的，因此必须为自定义异常类添加 SerializableAttribute 特性，并实现 ISerializable 接口。
- 以 Exception 作为异常类名的后缀，是良好的编程习惯。
- 在自定义异常包括本地化描述信息，也就是实现异常类的 Message 属性，而不是从基类继承，这显然违反了 Message 本身的语义。

- 虽然异常机制提高了自定义特定异常的方法，但是大部分时候我们应该优先考虑.NET 的系统异常，而不是实现自定义异常。
- 要想使自定义异常能应用于跨应用程序域，应该使异常可序列化，给异常类实现 ISerializable 接口是个好的选择。
- 如果自定义异常没有必要实现子类层次结构，那么异常类应该定义为密封类（sealed），以保证其安全性。

8.6.6 异常法则

异常法则是使用异常的最佳体验规则与设计规范要求，在实际的应用中有指导作用，主要包含以下几个方面：

- 尽可能以逻辑流程控制来代替异常，例如非空字段的处理不要延迟到业务处理阶段，而应在代码校验时完成。对于文件操作的处理，应该首先进行路径是否存在的校验，而不是将责任一股脑推给 FileNotFoundException 异常来处理。
- 将异常理解为程序的错误，显然曲解了对异常本质的认识。正如前文所言，异常是对程序接口隐含假设的一种违反，而这种假设常常和错误没有关系，反倒更多的是规则与约定。例如客户端“无理”的用 Word 来打开媒体文件，对程序开发者来说，这种“错误”是不可见的，这种问题只是违反了媒体文件只能用相关播放器打开的假设，而并非程序开发者的错误。
- 对异常形成文档，详细描述关于异常的原因和相关信息，是减少引发异常的有效措施。
- .NET 2.0 提供了很多新特性来简化异常的处理，同时从性能的角度考虑也是很好的选择，例如：

```
public static void Main()
{
   DateTime now;
   if(DateTime.TryParse("2007/11/7 23:31:00", out now))
   {
      Console.WriteLine("Now it's {0}", now);
   }
}
```

上例中实际实现了一个 Try-Parse 模式，以最大限度地减少异常造成的性能损失。对于很多常用的基础类型成员来说，实现 Try-Parse 模式是避免处理异常性能的一种不错的选择，.NET 类库的很多基础类型都实现了这一模式，例如 Int32、Char、Byte、DateTime 等等。

还有一种 Tester-Doer 模式，同样是用来减少异常的性能问题，在此就不做深入的研究。

- 对于多个 catch 块的情况，应该始终保证由最特定异常到最不特定异常的顺序来排列，以保证特定异常总是首先被执行。

- 异常提示应该准确而有效，提供丰富的信息给异常查看者来进行正确的判断和定位。
- 异常必须有针对性，盲目地抛出 System.Exception 意味着对于异常的原因是盲目的，而且容易造成异常被吞现象的发生。何时抛出异常，抛出什么异常，建立在对上下文环境的理解基础上。
- 尽量避免在 Finally 子句抛出异常。
- 应该避免在循环中抛出异常。
- 可以选择以 using 语句代替 try/finally 块来完成资源清理，详见 6.3 节“using 的多重身份”。

另外，微软还提供了 Enterprise Library 异常处理应用程序块（简称 EHAB）来实现更灵活、可扩展、可定制的异常处理框架，力图体现对异常处理的最新实践方式。

8.6.7 结论

本节旨在提纲挈领的对异常机制及其应用实践做以铺垫，关于异常的性能、未见异常处理及堆栈跟踪等问题只能浅尝于此。在今后的实践中，还应注意应用异常机制处理，要关注上下文的环境做出适当选择。

第 10 章 接触泛型

二十：C#泛型

C#泛型

C#泛型类与结构

C#除可单独声明泛型类型（包括类与结构）外，也可在基类中包含泛型类型的声明。但基类如果是泛型类，它的类型参数要么已实例化，要么来源于子类（同样是泛型类型）声明的类型参数。

```
class C<U, V> {} //合法
class D: C<string,int>{} //合法
class E<U, V>: C<U, V> {} //合法
class F<U, V>: C<string, int> {} //合法
```

```
class G：C<U, V> { } //非法
```

泛型类型的成员

```
class C<V>{
public V f1; //声明字段
public D<V> f2; //作为其他泛型类型的参数
public C(V x) {
this.f1 = x;
}
}
```

泛型类型的成员可以使用泛型类型声明中的类型参数。但类型参数如果没有任何约束，则只能在该类型上使用从 System.Object 继承的公有成员。

泛型接口

```
interface IList<T> {
T[] GetElements();
}
interface IDictionary<K,V> {
void Add(K key, V value);
}
// 泛型接口的类型参数要么已实例化，
// 要么来源于实现类声明的类型参数
class List<T> ：IList<T>, IDictionary<int, T> {
public T[] GetElements() { return null; }
public void Add(int index, T value) { }
}
```

泛型委托

```
delegate bool Predicate<T>(T value);
class X {
static bool F(int i) {...}
static bool G(string s) {...}
static void Main() {
Predicate<string> p2 = G;
Predicate<int> p1 = new Predicate<int>(F);
}
}
```

泛型委托支持在委托返回值和参数上应用参数类型，这些参数类型同样可以附带合法的约束。

泛型方法简介

- C#泛型机制只支持"在方法声明上包含类型参数"——即泛型方法
- C#泛型机制不支持在除方法外的其他成员（包括属性、事件、索引器、构造器、析构器）的声明上包含类　型参数，但这些成员本身可以包含在泛型类型中，并使用泛型类型的类型参数
- 泛型方法既可以包含在泛型类型中，也可以包含在非泛型类型中

泛型方法的声明与调用

```
//不是泛型类,是一个具体的类，这个类不需要泛型类型的实例化
public class Finder {
// 但是是一个泛型方法,请看泛型方法的声明，参数要求泛型化
public static int Find<T> ( T[] items, T item) {
for(int i=0;i<items.Length;i++){
if (items[i].Equals(item)) { return i; }
}
return -1;
}
```

```
}
// 泛型方法的调用<int>不是放到 Finder 后面，而是放在 Find 后面。
int i=Finder.Find<int> ( new int[]{1,3,4,5,6,8,9}, 6);
```

泛型方法的重载

```
class MyClass {
void F1<T>(T[] a, int i); // 不可以构成重载方法
void F1<U>(U[] a, int i);
void F2<T>(int x); //可以构成重载方法
void F2(int x);
//两句申明一样，where 字句,T 继承 A，泛型参数必需要继承 A
void F3<T>(T t) where T : A; //不可以构成重载方法
void F3<T>(T t) where T : B;
}
```

泛型方法的重写

```
abstract class Base
{
public abstract T F<T,U>(T t, U u) where U: T;
public abstract T G<T>(T t) where T: IComparable;
}
class Derived: Base{
//合法的重写，约束被默认继承，只需要写方法的签名
public override X F<X,Y>(X x, Y y) { }
//非法的重写，指定任何约束都是多余的
//重写的时候，不能写约束，也不添加新的约束，只能继承父类的约束。
public override T G<T>(T t) where T: IComparable {}
}
```

泛型约束简介

• C#泛型要求对"所有泛型类型或泛型方法的类型参数"的任何假定，都要基于"显式的约束"，以维护

C#所要求的类型安全。

• "显式约束"由 where 子句表达，可以指定"基类约束"，"接口约束"，"构造器约束""值类型/引用类型约束"共四种约束。

• "显式约束"并非必须，如果没有指定"显式约束"，泛型类型参数将只能访问 System.Object 类型中的公有方法。

基类约束

```
class A { public void F1() {…} }
class B { public void F2() {…} }
class C<S,T>
where S: A // S 继承自 A
where T: B // T 继承自 B
{
// 可以在类型为 S 的变量上调用 F1，
// 可以在类型为 T 的变量上调用 F2
….
}
```

接口约束

```
interface IPrintable { void Print(); }
interface IComparable<T> { int CompareTo(T v);}
interface IKeyProvider<T> { T GetKey(); }
class Dictionary<K,V>
where K: IComparable<K>
where V: IPrintable, IKeyProvider<K>
{
```

```
// 可以在类型为 K 的变量上调用 CompareTo，
// 可以在类型为 V 的变量上调用 Print 和 GetKey
....
}
```

构造器约束

```
class A { public A() { } }
class B { public B(int i) { } }
class C<T>
where T：new()
{
//可以在其中使用 T t=new T();
....
}
C<A> c=new C<A>(); //可以，A 有无参构造器
C<B> c=new C<B>(); //错误，B 没有无参构造器
```

值类型/引用类型约束

```
public struct A { … }
public class B { … }
class C<T>
where T：struct
{
// T 在这里面是一个值类型
…
}
C<A> c=new C<A>(); //可以，A 是一个值类型
C<B> c=new C<B>(); //错误，B 是一个引用类型
```

总结

- C#的泛型能力由 CLR 在运行时支持，它既不同于 C++在编译时所支持的静态模板，也不同于 Java 在编译器层面使用“搽拭法”支持的简单的泛型。
- C#的泛型支持包括类、结构、接口、委托共四种泛型类型，以及方法成员。
- C#的泛型采用“基类, 接口, 构造器, 值类型/引用类型”的约束方式来实现对类型参数的“显式约束”，它不支持 C++模板那样的基于签名的隐式约束。

泛型续：

根据微软的视频教程"跟我一起学 Visual Studio 2005C#语法篇"来学,因为里面有比较多的代码示例,学起来比较容易好理解

1.未使用泛型的 Stack 类

```
1using System;
2
3public class Stack
4{
5    readonly int m_Size;
6    int m_StackPointer = 0;
7    object[] m_Items;
8    public Stack(): this(100)
9    { }
10    public Stack(int size)
11    {
12        m_Size = size;
13        m_Items = new object[m_Size];
14    }
15    public void Push(object item)
16    {
17        if (m_StackPointer >= m_Size)
18            throw new StackOverflowException();
19
20        m_Items[m_StackPointer] = item;
21        m_StackPointer++;
22    }
23    public object Pop()
24    {
25        m_StackPointer--;
26        if (m_StackPointer >= 0)
27        {
28            return m_Items[m_StackPointer];
29        }
30        else
31        {
32            m_StackPointer = 0;
33            throw new InvalidOperationException("Cannot pop an empty stack");
34        }
35    }
36}
37
```

2.使用泛型的类

```
1using System;
2
3public class Stack<T>
4{
5    readonly int m_Size;
6    int m_StackPointer = 0;
7    T[] m_Items;
8    public Stack()
9        : this(100)
10   {
11   }
12   public Stack(int size)
13   {
14       m_Size = size;
15       m_Items = new T[m_Size];
16   }
17   public void Push(T item)
18   {
19       if (m_StackPointer >= m_Size)
20           throw new StackOverflowException();
21
22       m_Items[m_StackPointer] = item;
23       m_StackPointer++;
24   }
25   public T Pop()
26   {
27       m_StackPointer--;
28       if (m_StackPointer >= 0)
29       {
30           return m_Items[m_StackPointer];
31       }
32       else
33       {
34           m_StackPointer = 0;
35           //throw new InvalidOperationException("Cannot pop an empty stack");
36           return default(T);
37       }
38   }
39}
40
41public class Stack1<T> : Stack<T>
42{
43
44}
45
```
下为 PDF 文档,我感觉挺好的,很简单,我听的懂就是好的
/Clingingboy/one.pdf

多个泛型

```
1class Node<K, T>
2{
3    public K Key;
4    public T Item;
5    public Node<K, T> NextNode;
6    public Node()
7    {
8        Key = default(K);
9        Item = default(T);
10        NextNode = null;
11   }
12   public Node(K key, T item, Node<K, T> nextNode)
```

```
13    {
14        Key = key;
15        Item = item;
16        NextNode = nextNode;
17    }
18}
```

泛型别名

```
1using list = LinkedList<int, string>;
```

泛型约束

```
 1public class LinkedList<K, T> where K : IComparable
 2{
 3    Node<K, T> m_Head;
 4    public LinkedList()
 5    {
 6        m_Head = new Node<K, T>();
 7    }
 8    public void AddHead(K key, T item)
 9    {
10        Node<K, T> newNode = new Node<K, T>(key, item, m_Head.NextNode);
11        m_Head.NextNode = newNode;
12    }
13
14    T Find(K key)
15    {
16        Node<K, T> current = m_Head;
17        while (current.NextNode != null)
18        {
19            if (current.Key.CompareTo(key) == 0)
20                break;
21            else
22                current = current.NextNode;
23        }
24        return current.Item;
25    }
26
27}
28
```

```
 1using System;
 2using System.Collections.Generic;
 3using System.Text;
 4
 5namespace VS2005Demo1
 6{
 7    public class MyBaseClassGeneric // sealed,static
 8    {
 9    }
10
11    interface IMyBaseInterface
12    {
13        void A();
14    }
15
16    internal class GenericClass<T> where T : MyBaseClassGeneric,IMyBaseInterface
17    {
18
19    }
20
21    class GClass<K, T> where K : MyBaseClassGeneric,IMyBaseInterface,new() where T : K
```

```
22  {
23
24  }
25
26  class GUClass<K, T> where T : K where K : MyBaseClassGeneric,IMyBaseInterface, new()
27  {
28      GClass<K, T> obj = new GClass<K, T>();
29  }
30
31
32  不能将引用/值类型约束与基类约束一起使用，因为基类约束涉及到类#region 不能将引用/值类型约束与基类约束一起使用，因为基类约束涉及到类
33
34  //class A<T> where T : struct,class
35  //{}
36
37  #endregion
38
39  不能使用结构和默认构造函数约束，因为默认构造函数约束也涉及到类#region 不能使用结构和默认构造函数约束，因为默认构造函数约束也涉及到类
40
41  //class A<T> where T : struct,new()
42  //{}
43
44  #endregion
45
46  虽然您可以使用类和默认构造函数约束，但这样做没有任何价值#region 虽然您可以使用类和默认构造函数约束，但这样做没有任何价值
47
48  class A<T> where T : new()
49  {
50      T obj = new T();
51  }
52
53  class TypeA
54  {
55      public TypeA() { }
56  }
57
58  class TestA
59  {
60      A<TypeA> obj = new A<TypeA>();
61  }
62
63  #endregion
64
65  可以将引用/值类型约束与接口约束组合起来，前提是引用/值类型约束出现在约束列表的开头#region 可以将引用/值类型约束与接口约束组合起来，前提是引用/值类型约束出现在约束列表的开头
66
67  class SClass<K> where K : struct, IMyBaseInterface
68  { }
69
70  class CClass<K> where K : class, IMyBaseInterface
71  { }
72
73  #endregion
74}
75
```

第二十一回：认识全面的 null

说在，开篇之前

说在，开篇之前

null、nullable、??运算符、null object 模式，这些闪亮的概念在你眼前晃动，我们有理由相信“存在即合理”，事实上，null 不光合理，而且重要。本文，从 null 的基本认知开始，逐层了解可空类型、??运算符和 null object 模式，在循序之旅中了解不一样的 null。

你必须知道的.NET，继续全新体验，分享更多色彩。

1 从什么是 null 开始？

null，一个值得尊敬的数据标识。

一般说来，null 表示空类型，也就是表示什么都没有，但是“什么都没有”并不意味“什么都不是”。实际上，null 是如此的重要，以致于在 JavaScript 中，Null 类型就作为 5 种基本的原始类型之一，与 Undefined、Boolean、Number 和 String 并驾齐驱。这种重要性同样表现在.NET 中，但是一定要澄清的是，null 并不等同于 0，""，string.Empty 这些通常意义上的“零”值概念。相反，null 具有实实在在的意义，这个意义就是用于标识变量引用的一种状态，这种状态表示没有引用任何对象实例，也就是表示“什么都没有”，既不是 Object 实例，也不是 User 实例，而是一个空引用而已。

在上述让我都拗口抓狂的表述中，其实中心思想就是澄清一个关于 null 意义的无力诉说，而在.NET 中 null 又有什么实际的意义呢？

在.NET 中，null 表示一个对象引用是无效的。作为引用类型变量的默认值，null 是针对指针（引用）而言的，它是引用类型变量的专属概念，表示一个引用类型变量声明但未初始化的状态，例如：

```
object obj = null;
```

此时 obj 仅仅是一个保存在线程栈上的引用指针，不代表任何意义，obj 未指向任何有效实例，而被默认初始化为 null。

object obj 和 object obj = null 的区别？

那么，object obj 和 object obj = null 有实际的区别吗？答案是：有。主要体现在编译器的检查上。默认情况下，创建一个引用类型变量时，CLR 即将其初始化为 null，表示不指向任何有效实例，所以本质上二者表示了相同的意义，但是有有所区别：

```
// Copyright   : www.anytao.com
// Author      : Anytao, http://www.anytao.com
// Release     : 2008/07/31 1.0
```

```
            //编译器检测错误：使用未赋值变量 obj
            //object obj;

            //编译器理解为执行了初始化操作，所以不引发编译时错误
            object obj = null;

            if (obj == null)
            {
                //运行时抛出 NullReferenceException 异常
                Console.WriteLine(obj.ToString());
            }
```

注：当我把这个问题抛给几个朋友时，对此的想法都未形成统一的共识，几位同志各有各的理解，也各有个的道理。当然，我也慎重的对此进行了一番探讨和分析，但是并未形成完全 100%确定性的答案。不过，在理解上我更倾向于自己的分析和判断，所以在给出上述结论的基础上，也将这个小小的思考留给大家来探讨，好的思考和分析别忘了留给大家。事实上，将

```
        static void Main(string[] args)
        {
            object o;
            object obj = null;
        }
```

反编译为 IL 时，二者在 IL 层还是存在一定的差别：

```
.method private hidebysig static void Main(string[] args) cil managed
{
    .entrypoint
    .maxstack 1
    .locals init (
        [0] object o,
        [1] object obj)
    L_0000: nop
    L_0001: ldnull
    L_0002: stloc.1
    L_0003: ret
}
```

前者没有发生任何附加操作；而后者通过 ldnull 指令推进一个空引用给 evaluation stack，而 stloc 则将空引用保存。

回到规则

在.NET 中，对 null 有如下的基本规则和应用：

- null 为引用类型变量的默认值，为引用类型的概念范畴。
- null 不等同于 0，""，string.Empty。
- 引用 is 或 as 模式对类型进行判断或转换时，需要做进一步的 null 判断。


快捷参考

- 关于 is 和 as 模式，可以参考《你必须知道的.NET》 7.5 节"恩怨情仇：is 和 as
- ”
- 第一回：恩怨情仇：is 和 as

www.anytao.com

- 判断一个变量是否为 null，可以应用==或!=操作符来完成。
- 对任何值为 nul 的 l 变量操作，都会抛出 NullReferenceException 异常。

2 Nullable<T>（可空类型）

一直以来，null 都是引用类型的特有产物，对值类型进行 null 操作将在编译器抛出错误提示，例如：

```
            //抛出编译时错误
            int i = null;
            if (i == null)
            {
                Console.WriteLine("i is null.");
```

```
}
```

正如示例中所示，很多情况下作为开发人员，我们更希望能够以统一的方式来处理，同时也希望能够解决实际业务需求中对于"值"也可以为"空"这一实际情况的映射。因此，自.NET 2.0 以来，这一特权被新的 System.Nullable<T>（即，可空值类型）的诞生而打破，解除上述诟病可以很容易以下面的方式被实现：

```
//Nullable<T>解决了这一问题
int? i = null;
if (i == null)
{
    Console.WriteLine("i is null.");
}
```

你可能很奇怪上述示例中并没有任何 Nullable 的影子，实际上这是 C#的一个语法糖，以下代码在本质上是完全等效的：

```
int? i = null;
Nullable<int> i = null;
```

显然，我们更中意以第一种简洁而优雅的方式来实现我们的代码，但是在本质上 Nullable<T>和 T? 他们是一路货色。

可空类型的伟大意义在于，通过 Nullable<T>类型，.NET 为值类型添加"可空性"，例如 Nullable<Boolean>的值就包括了 true、false 和 null，而 Nullable<Int32>则表示值即可以为整形也可以为 null。同时，可空类型实现了统一的方式来处理值类型和引用类型的"空"值问题，例如值类型也可以享有在运行时以 NullReferenceException 异常来处理。

另外，可空类型是内置于 CLR 的，所以它并非 c#的独门绝技，VB.NET 中同样存在相同的概念。

Nullable 的本质（IL）

那么我们如何来认识 Nullable 的本质呢？当你声明一个：

```
Nullable<Int32> count = new Nullable<Int32>();
```

时，到底发生了什么样的过程呢？我们首先来了解一下 Nullable 在.NET 中的定义：

```
public struct Nullable<T> where T : struct
{
    private bool hasValue;
    internal T value;
```

```
        public Nullable(T value);
        public bool HasValue { get; }
        public T Value { get; }
        public T GetValueOrDefault();
        public T GetValueOrDefault(T defaultValue);
        public override bool Equals(object other);
        public override int GetHashCode();
        public override string ToString();
        public static implicit operator T?(T value);
        public static explicit operator T(T? value);
    }
```

根据上述定义可知，Nullable 本质上仍是一个 struct 为值类型，其实例对象仍然分配在线程栈上。其中的 value 属性封装了具体的值类型，Nullable<T>进行初始化时，将值类型赋给 value，可以从其构造函数获知：

```
        public Nullable(T value)
        {
            this.value = value;
            this.hasValue = true;
        }
```

同时 Nullable<T>实现相应的 Equals、ToString、GetHashCode 方法，以及显式和隐式对原始值类型与可空类型的转换。因此，在本质上 Nullable 可以看着是预定义的 struct 类型，创建一个 Nullable<T>类型的 IL 表示可以非常清晰的提供例证，例如创建一个值为 int 型可空类型过程，其 IL 可以表示为：

```
    .method private hidebysig static void Main() cil managed
    {
        .entrypoint
        .maxstack 2
        .locals init (
            [0] valuetype [mscorlib]System.Nullable`1<int32> a)
        L_0000: nop
        L_0001: ldloca.s a
        L_0003: ldc.i4 0x3e8
        L_0008: call instance void [mscorlib]System.Nullable`1<int3
2>::.ctor(!0)
```

```
        L_000d: nop
        L_000e: ret
    }
```

对于可空类型，同样需要必要的小结：

- 可空类型表示值为 null 的值类型。
- 不允许使用嵌套的可空类型，例如 Nullable<Nullable<T>> 。
- Nullable<T>和 T?是等效的。
- 对可空类型执行 GetType 方法，将返回类型 T，而不是 Nullable<T>。
- c#允许在可空类型上执行转换和转型，例如：

```
            int? a = 100;
            Int32 b = (Int32)a;
            a = null;
```

- 同时为了更好的将可空类型于原有的类型系统进行兼容，CLR 提供了对可空类型装箱和拆箱的支持。

3 ??运算符

在实际的程序开发中，为了有效避免发生异常情况，进行 null 判定是经常发生的事情，例如对于任意对象执行 ToString()操作，都应该进行必要的 null 检查，以免发生不必要的异常提示，我们常常是这样实现的：

```
            object obj = new object();

            string objName = string.Empty;
            if (obj != null)
            {
                objName = obj.ToString();
            }
```

```
            Console.WriteLine(objName);
```

然而这种实现实在是令人作呕，满篇的 if 语句总是让人看着浑身不适，那么还有更好的实现方式吗，我们可以尝试（? :）三元运算符：

```
            object obj = new object();
            string objName = obj == null ? string.Empty : obj.ToString
();
            Console.WriteLine(objName);
```

上述 obj 可以代表任意的自定义类型对象，你可以通过覆写 ToString 方法来输出你想要输出的结果，因为上述实现是如此的频繁，所以.NET 3.0 中提供了新的操作运算符来简化 null 值的判断过程，这就是：??运算符。上述过程可以以更加震撼的代码表现为：

```
            // Copyright   : www.anytao.com
            // Author      : Anytao，http://www.anytao.com
            // Release     : 2008/07/31 1.0

            object obj = null;
            string objName = (obj ?? string.Empty).ToString();
            Console.WriteLine(objName);
```

那么??运算符的具体作用是什么呢？

??运算符，又称为 null-coalescing operator，如果左侧操作数为 null，则返回右侧操作数的值，如果不为 null 则返回左侧操作数的值。它既可以应用于可空类型，有可以应用于引用类型。

4 Nulll Object 模式

模式之于设计，正如秘笈之于功夫。正如我们前文所述，null 在程序设计中具有举足轻重的作用，因此如何更优雅的处理"对象为空"这一普遍问题，大师们提出了 Null Object Pattern 概念，也就是我们常说的 Null Object 模式。例如 Bob 大叔在《敏捷软件开发--原则、模式、实践》一书，Martin Fowler 在《Refactoring: Improving the Design of Existing Code》一书，都曾就 Null Object 模式展开详细的讨论，可见 23 中模式之外还是有很多设计精髓，可能称为模式有碍经典。但是仍然

值得我们挖据、探索和发现。

下面就趁热打铁，在 null 认识的基础上，对 null object 模式进行一点探讨，研究 null object 解决的问题，并提出通用的 null object 应用方式。

解决什么问题？

简单来说，null object 模式就是为对象提供一个指定的类型，来代替对象为空的情况。说白了就是解决对象为空的情况，提供对象“什么也不做”的行为，这种方式看似无聊，但却是很聪明的解决之道。

举例来说，一个 User 类型对象 user 需要在系统中进行操作，那么典型的操作方式是：

```
            if (user != null)
            {
                manager.SendMessage(user);
            }
```

这种类似的操作，会遍布于你的系统代码，无数的 if 判断让优雅远离了你的代码，如果大意忘记 null 判断，那么只有无情的异常伺候了。于是，Null object 模式就应运而生了，对 User 类实现相同功能的 NullUser 类型，就可以有效的避免繁琐的 if 和不必要的失误：

```
    // Copyright   : www.anytao.com
    // Author      : Anytao，http://www.anytao.com
    // Release     : 2008/07/31 1.0

    public class NullUser : IUser
    {
        public void Login()
        {
            //不做任何处理
        }

        public void GetInfo() { }

        public bool IsNull
        {
            get { return true; }
        }
    }
```

IsNull 属性用于提供统一判定 null 方式，如果对象为 NullUser 实例，那么 IsNull 一定是 true 的。

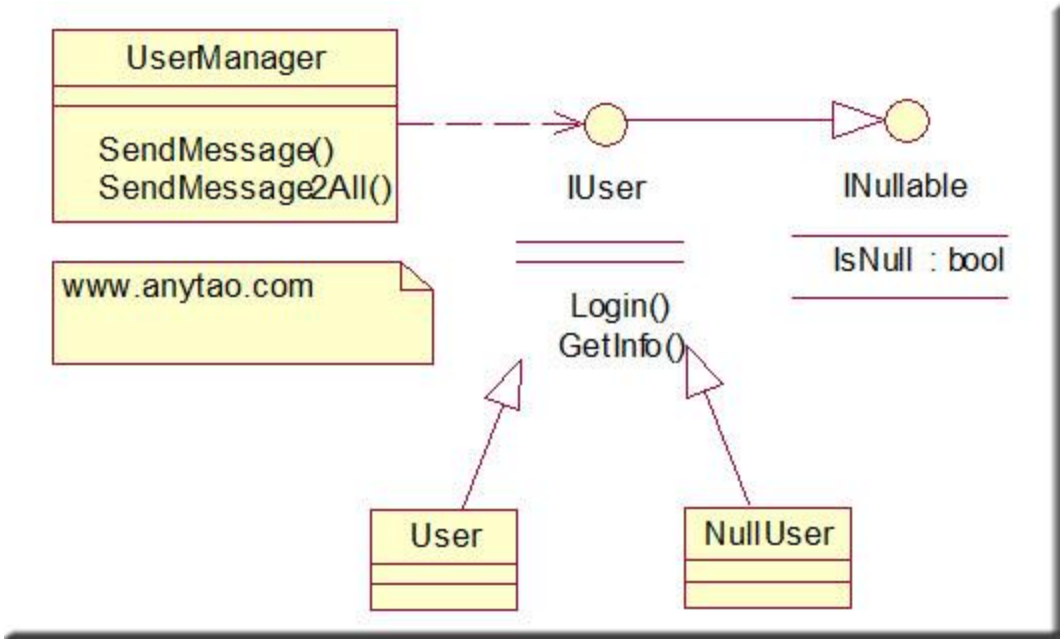
那么，二者的差别体现在哪儿呢？其实主要的思路就是将 null value 转换为 null object，把对 user == null 这样的判断，转换为 user.IsNull 虽然只有一字之差，但是本质上是完全两回事儿。通过 null object 模式，可以确保返回有效的对象，而不是没有任何意义的 null 值。同时，“在执行方法时返回 null object 而不是 null 值，可以避免 NullReferenceExecption 异常的发生。”，这是来自 Scott Dorman 的声音。

通用的 null object 方案

下面，我们实现一种较为通用的 null object 模式方案，并将其实现为具有.NET 特色的 null object，所以我们采取实现.NET 中 INullable 接口的方式来实现，INullable 接口是一个包括了 IsNull 属性的接口，其定义为：

```
    public interface INullable
    {
        // Properties
        bool IsNull { get; }
    }
```

仍然以 User 类为例，实现的方案可以表达为：



图中仅仅列举了简单的几个方法或属性，旨在达到说明思路的目的，其中 User 的定义为：

```
    // Copyright   : www.anytao.com
    // Author      : Anytao, http://www.anytao.com
```

```
    // Release     : 2008/07/31 1.0

    public class User : IUser
    {
        public void Login()
        {
            Console.WriteLine("User Login now.");
        }

        public void GetInfo()
        {
            Console.WriteLine("User Logout now.");
        }

        public bool IsNull
        {
            get { return false; }
        }
    }
```

而对应的 NullUser，其定义为：

```
    // Copyright   : www.anytao.com
    // Author      : Anytao，http://www.anytao.com
    // Release     : 2008/07/31 1.0

    public class NullUser : IUser
    {
        public void Login()
        {
            //不做任何处理
        }

        public void GetInfo() { }

        public bool IsNull
        {
```

```
            get { return true; }
        }
    }
```

同时通过 UserManager 类来完成对 User 的操作和管理，你很容易思考通过关联方式，将 IUser 作为 UserManger 的属性来实现，基于对 null object 的引入，实现的方式可以为：

```
    // Copyright   : www.anytao.com
    // Author      : Anytao，http://www.anytao.com
    // Release     : 2008/07/31 1.0

    class UserManager
    {
        private IUser user = new User();

        public IUser User
        {
            get { return user; }
            set
            {
                user = value ?? new NullUser();
            }
        }
    }
```

当然有效的测试是必要的：

```
        public static void Main()
        {
            UserManager manager = new UserManager();
            //强制为 null
            manager.User = null;
            //执行正常
            manager.User.Login();

            if (manager.User.IsNull)
            {
                Console.WriteLine("用户不存在，请检查。");
```

```
            }
        }
```

通过强制将 User 属性实现为 null，在调用 Login 时仍然能够保证系统的稳定性，有效避免对 null 的判定操作，这至少可以让我们的系统少了很多不必要的判定代码。

详细的代码可以通过本文最后的下载空间进行下载。实际上，可以通过引入 Facotry Method 模式来构建对于 User 和 NullUser 的创建工作，这样就可以完全消除应用 if 进行判断的僵化，不过那是另外一项工作罢了。

当然，这只是 null object 的一种实现方案，在此对《Refactoring》一书的示例进行改良，完成更具有.NET 特色的 null object 实现，你也可以请 NullUser 继承 Use 并添加相应的 IsNull 判定属性来完成。

借力 c# 3.0 的 Null object

在 C# 3.0 中，Extension Method（扩展方法）对于成就 LINQ 居功至伟，但是 Extension Method 的神奇远不是止于 LINQ。在实际的设计中，灵活而巧妙的应用，同样可以给你的设计带来意想不到的震撼，以上述 User 为例我们应用 Extension Method 来取巧实现更简洁 IsNull 判定，代替实现 INullable 接口的方法而采用更简单的实现方式。重新构造一个实现相同功能的扩展方法，例如：

```
    // Copyright   : www.anytao.com
    // Author      : Anytao，http://www.anytao.com
    // Release     : 2008/07/31 1.0

    public static class UserExtension
    {
        public static bool IsNull(this User user)
        {
            return null == user;
        }
    }
```

当然，这只是一个简单的思路，仅仅将对 null value 的判断转换为 null object 的判断角度来看，扩展方法带来了更有效的、更简洁的表现力。

null object 模式的小结

- 有效解决对象为空的情况，为值为 null 提供可靠保证。

- 保证能够返回有效的默认值，例如在一个 IList<User> userList 中，能够保证任何情况下都有有效值返回，可以保证对 userList 操作的有效性，例如：

```
        // Copyright   : www.anytao.com
        // Author      : Anytao，http://www.anytao.com
        // Release     : 2008/07/31 1.0

        public void SendMessageAll(List<User> userList)
        {
            //不需要对 userList 进行 null 判断
            foreach (User user in userList)
            {
                user.SendMessage();
            }
        }
```

- 提供统一判定的 IsNull 属性。可以通过实现 INullable 接口，也可以通过 Extension Method 实现 IsNull 判定方法。
- null object 要保持原 object 的所有成员的不变性，所以我们常常将其实现为 Sigleton 模式。
- Scott Doman 说“在执行方法时返回 null object 而不是 null 值，可以避免 NullReferenceException 异常的发生”，这完全是对的。

5 结论

虽然形色匆匆，但是通过本文你可以基本了解关于 null 这个话题的方方面面，堆积到一起就是对一个概念清晰的把握和探讨。技术的魅力，大概也正是如此而已吧，色彩斑斓的世界里，即便是“什么都没有”的 null，在我看来依然有很多很多。。。值得探索、思考和分享。

还有更多的 null，例如 LINQ 中的 null，SQL 中的 null，仍然可以进行探讨，我们将这种思考继续，所收获的果实就越多。

Anytao | 2008-07-31 | 你必须知道的.NET

参考文献

（Book）Martin Fowler，Refactoring: Improving the Design of Existing Code

（cnblogs）zhuweisky，使用 Null Object 设计模式

（blogs）Scott Dorman，Null Object pattern

二十二回 学习方法论

学习方法论

本文将介绍以下内容：

- .NET 的核心知识汇总
- 学习.NET 的圣经心得

1. 引言

最近常常为学习中的问题而伤神，幸有管伟一起常常就技术问题拿来讨论，我已想将讨论的内容以基本原貌的方式，形成一个系列[和管子对话]，通过记录的方式将曾经的友情和激情记录在园子里，除了勉励自己，也可受用他人。因此[和管子对话]系列，纯属口头之说，一家之言，而且东拉西撤。但是却给我一个很好的启示，就是将学习的东西，尤其是基础性的本质作为系统来常常回味在脑子里，案头间。

所以才有了这个系统[你必须知道的.NET]浮出水面，系列的主要内容就是.NET 技术中的精华要点，以基础内容为主，以设计思想为辅，有自己的体会，有拿来的精品，初步的思路就是以实例来讲述概念，以简单来表达本质。因为是总结，因为是探索，所以 post 中的内容不免有取之于民的东西，我将尽己可能的标注出处。

2. 目录

谈起.NET 基础，首先我将脑子的清单列出，本系列的框架也就自然而然的和盘推出，同时希望园子的朋友尽力补充，希望能把这个系列做好，为初学的人，为迷茫的人，开一条通途

第二十一回：学习方法论

本文，源自我回答刚毕业朋友关于.NET 学习疑惑的回复邮件。

本文，其实早计划在《你必须知道的.NET》写作之初的后记部分，但是因为个中原因未能如愿，算是补上本书的遗憾之一。

本文，作为[《你必须知道的.NET》]系列的第 20 回，预示着这个系列将开始新的征程，算是[你必须知道的.NET]2.0 的开始。

本文，作为一个非技术篇章，加塞儿到《你必须知道的.NET》队伍中，我想至少因为回答了以下几个***必须知道***的非技术问题：.NET 应该学习什么？ .NET 应该如何学习？ .NET 的学习方法？

本文，不适合所有的人。

开始正文：
关于这个问题，也有不少刚刚入行的朋友向我问起。我想可能一千个人就有一千个答案，我不能保证自己的想法适合于所有的人，但是这确实是我自己的体会和经历，希望能给你一些参考的价值。同时，我也严正的声明，我也是个学习者，也在不断的追求，所以这里的体会只是交流，并非说教。

作为同行，首先恭喜你进入了一个艰难困苦和其乐无穷并存的行业，这是软件的现状，也是软件的未来。如果你想迅速成功，或者发家致富，显然是个难以实现的梦想。老 Bill 和李彦宏在这个行业是难以复制的，所以做好长期艰苦卓绝的准备是必须的。至少，我身边的朋友，包括我自己都是经历了这个过程，而且依然在这个过程中，累并快乐着。所以，如此辛苦，又没有立竿见影的"钱"途，想要在这个领域有所发展，只能靠**坚持和兴趣**了。二者缺一不可，对于刚刚毕业的你来说，这个准备是必须有的。这是我的第一个体会，可能比较虚，但是这个在我看来却是最重要的一条。

第一条很关键，但是除了在思想上做好准备，还有应该就是你关心的如何下手这个问题了？从自己的感觉来说，我觉得比较重要的因素主要包括：

1 基础至上。

其实早在两年前，我也存在同样的疑惑，很多的精力和时间花费在了追求技术技巧、技术应用和技术抄袭的自我陶醉状态。历数过去的种种光辉历程，很多宝贵的人生都花在交学费的道路上了。所以，当我把全部的精力投入到基础和本质研究的课题上时，竟然发现了别样的天地。原来再花哨的应用，再绝妙的技巧，其实都架构在技术基础的基础上，没有对技术本质的深刻理解，谈何来更进一步了解其他。这种体会是真实而有效的，所以我将体会、研究和心得，一路分享和记录下来，于是就有了《你必须知道的.NET》这本书的诞生，我切实的觉得从这个起点开始，了解你必须知道的，才能了解那些更广阔的技术领域。

所以，如果能够坚持，不放弃枯燥，从基础开始踏踏实实的学习基础，我想你一定会有所突破。而这个突破，其实也有着由量到质的飞跃，以.NET 为例，我认为了解 CLR 运行机制，深刻的认识内存管理，类型系统，异常机制，熟悉 FCL 基本架构，学习 c#语言基础，认识 MSIL、元数据、Attribute、反射、委托等等，当然还包括面向对象和设计架构，都是必不可少的基础内容。你可以从《你必须知道的.NET》的目录中来大致了解到应该掌握的基础内容，顺便广告了:-)

话音至此，顺便推荐几本基础方面的书，如果有时间可以好好研究研究：

- Don Box, Chris Sells, Essential .NET，一本圣经，深刻而又深邃，为什么不出第二卷？
- Jeffrey Richter, Applied Microsoft .NET Framework Programming，.NET 世界的唯一经典，偶像级的 Jeffrey 是我的导师。
- Patrick Smacchia, Pracical .NET2 and C#2，.NET 领域的百科全书，可以当作新华字典来读技术。
- Richard Jones, Rafael D Lins, Garbage Collection: Algorithms for Automatic Dynamic Memory Management，内存管理方面，就靠它了。
- Christian Nagel, Bill Evjen, Jay Glynn, Professional C# 2005，c#基础大全，大家都在看，所以就看吧。
- Thinking in Java，是的，一本 Java 书，但是带来的不仅仅是 Java，写书写到这份上，不可不谓牛叉。
- Anytao, 你必须知道的.NET，我很自信，没有理由不推荐，这本书有其他作品所没有的特别之处，虽不敢恬列于大师的经典行列，但是推荐还是经得起考验。

我一直主张，书不在多，有仙则灵。上面的几本，在我看来就足以打好基础这一关。当然如果有更多的追求和思索，还远远不够，因为技术的脚步从未止步。但是，至少至少，应该从这里开始。。。

2 你够 OO 吗？

不管对业界对 OO 如何诟病，不管大牛对 OO 如何不懈，那是他们折腾的事业。而我们的事业却无法远离这片看似神秘的王国，因为但凡从项目和产品一路厮杀而来的高手，都理解 OO 的强大和神秘。站在高高的塔尖来看软件，玩来玩去就是这些玩意儿了。所以，在我看来 OO 其实也是软件技术的必要基础，也是技术修炼的基本功之一，因此我也毫不犹豫的将对面向对象的理解纳入了《你必须知道的.NET》一书的第一部分范畴。

然而，实话实说，OO 的修炼却远没有.NET 基础来得那么容易，苦嚼一车好书，狂写万行代码，也未必能够完全领悟 OO 精妙。说得玄乎点儿，这有些像悟道，想起明代前无古人后无来着的心学开创者王阳名先生，年轻时每天格物修炼的痴呆场景，我就觉得这玩意儿实在不靠谱。其实，很少有人能完全在 OO 面前说彻悟，所以我们大家都不例外。但是因为如此重要，以至于我们必须找点儿东西或者思路来摩拳擦掌，了解、深入和不断体会，所以我对面向对象的建议是：始终如一的修炼，打好持久战。

如何打好仗呢，不例外的先推荐几本经典作品吧：

- EricFreeman, Elisabeth Freeman. Head First Design Patterns，标准的言简意赅，形象生动，难得佳作。
- Erich Gamma, Richard Helm, Ralph Johnson, John Vlisside，设计模式-可复用面向对象软件的基础，开山祖师的作品，不独白不读。
- Martin Fowler, Refactoring: Improving the Design of Existing Code，同样的经典，很拉风。
- Robert C. Martin，敏捷软件开发：原则、模式与实践，对于设计原则，无出其右者。
- 张逸，软件设计精要与模式，国内作品的优秀作品，园子里的经典之作。

有了好书，还是远远不够的。所以，还得继续走王阳明的老路，今天格一物，明天格一物，看见什么格什么。用咱们的专业术语说，就是不断的学习和实践他人的智慧结晶，看经典框架，写熟练代码。我的一位偶像曾语重心长的告诉我，做软件的不写上千万行代码，根本就没感觉。按照这个标准衡量一下自己，我发现我还只是小学生一个，所以废话少说，还是去格物吧。

那么 OO 世界的物又是什么，又该如何去格，在我看来大致可以包括下面这些内容，但是分类不按学科标准：

- 面向对象的基本内容：类、对象、属性、方法、字段。
- 面向对象的基本要素：封装、继承、多态，我再外加一个接口。
- 设计原则：接口隔离、单一职责、开放封闭、依赖倒置、Liskov 替换，没什么可说的，这些实在太重要了。
- 设计模式：也没有可说的，实在太重要了。
    - Singleton

- Abstract Factory
- Factory Method
- Composite
- Adapter
- Bridge
- Decorator
- Facade
- Proxy
- Command
- Observer
- Template Method
- Strategy
- Visitor

分层思想：例如经典的三层架构

模块化

AOP

SOA

ORM

……

这些 OO 领域的基本内容，看起来令郎满目，其实互相联系、互为补充，没有独立的分割，也没有独立的概念，了解这个必然牵出那个，所以修炼起来并不孤单，反倒在不断的领悟中能够窃喜原来软件也可以如此精彩。

3 舍得，是门艺术。

有了技术基础，懂得修炼 OO，下面就是舍得的问题了。舍得舍得，不舍怎得？

.NET 技术有着近乎夸张的应用范畴，从 Windows GDI 应用，到 ASP.NET Web 应用，到 WCF 分布式应用，到 Window Mobile 嵌入式应用，到 ADO.NET 数据处理，到 XML Webservice，.NET 无处不在。所以，对于.NET 技术的学习，你应该有个起码的认识，那就是：我不可能了解.NET 的整个面貌， 还有个起码的问题继续，那就是：我还要学吗？

当然不可能了解所有，因此你必须选择和舍得，选择有方向，舍得有兴趣；我还要学吗？当然要学，但是应该首先清楚如何学？在这么多眼花缭乱的技术应用中，有一个基础始终支撑着.NET 技术这艘航母在稳步前行，不管是什么应用，不管是什么技术，不管是什么框架，CLR 总是.NET 技术的心脏。通过表面来倾听心脏的声音，才能更好的了解机器的运转，顺着血管的脉络了解框架，才能明白机制背后的玄机。层出不穷的新技术和新名词总是能吸引你的眼球，但是永远不要只盯着那块蛋糕，而掉了整个礼物，所以对.NET 的学习一定要打好基础，从了解 CLR 底层机制和.NET 框架类库开始，逐渐的追求你的技术选择。

善于分辨，不盲从。每天上 cnblogs、MSDN 和其他的订阅技术文章，是我的习惯，但是如果每篇都读，每篇都看，那就基本没有其他的时间，所以你必须有分辨的能力，和抵抗诱惑的心态。找准自己的方向，并且坚持下来，是难能可贵的。

在这方面，没有参考，也没有推荐，全屏自己的慧眼。眼光，是个关键。

4 读几本经典的作品。

这一点其实并不需要多说，推荐的几本作品值得花点儿功夫来学习，因为这的确是最初的开始，走在路上从起跑线就走错了方向，大致快速追上是比较难得。所以经典的作品就是一个好的起点，我也会不时的在个人博客中推荐更好的专著，希望你继续关注 J

5 遵守规范，养成良好的编程习惯。

其实这是个看似无足轻重的小事儿，我经常看到自以为天下无敌的高手，胡乱的在编辑器中挥洒天赋，一阵高歌猛进，但最后自己都不知道当初的本意是什么。软件是个可持续的资源，于人于己都遵守点儿规则，出来混是要有点儿职业道德。对自己而言，良好的编程习惯正是一个良好学习习惯的开始。看着自己的代码，感觉像艺术一般优雅，大致也就是周杰伦听到东风破时候的感觉吧，怎一个爽字了得。

推荐一本这方面的书：

- Krzysztof Cwalina,Brad Abrams , .NET 设计规范--.NET 约定、惯用法与模式

6 学习，讲究方法。

具体的学习方法，实在是因人而异，我从来不主张学习他人的方法，因为人性是难以复制的东西。自己的只有自己最清楚，所以你可以模仿他人的技艺，但是用于无法刻画其灵魂。关于学习方法这档子事儿，我向来不喜欢参考他人，也更不喜欢推荐。

但是，即便如此，丝毫不减弱学习方法的重要性，懂得了解自己的人是真正的智者，所以挖掘自身潜力永远是摆在自己眼前的课题。寻找一套行之有效的方式方法，非常的重要，但是不要学着模仿，这方面我觉得只有创新才能成功。

如果实在没有自己的方法，我就觉得没有方法就是好方法，苦练多看，永远不过时。

7 找一个好老师。

如果有幸能有一位德高望重而又乐于奉献的师长指导，那的确是人生之幸运，但是这种概率实在是太小了。我没有赶上，所以大部分人也没法赶上。没办法，还是需要好的老师，那么哪儿有这样才高而又德厚的人才呢？

答案是互联网。google，baidu，一个都不能少。

MSDN 是个好工具，博客园是个好地方，《.NET 禅意花园》是个好开始。

8 英文，无可避免。

前面说过，要不断的修炼和格物，要学习好的作品，认识好的框架。很不幸的是，这些好事儿全被老外占了，因为本来就是从他们那里开始的，所以也不需要泄气。中国人自古都是师夷长技以制夷的高手，希望软件产业的大旗别在我们手上倒下。但是，话说回来，英文就成了一个必须而又伤神的拦路虎，但是没办法使劲的嚼吧。多看多写多读，也就能应付了。

关于英文的学习和成长，我并不寄希望于在什么英语速成班里走回头路，学校苦干这么多年也每隔名趟，所以下手还是务实点儿，我推荐几个好的英文网站和大牛博客，算是提高技术的同时提高英语，一箭双雕，一举两得：

- http://www.gotdotnet.com/
- http://codeproject.com/

- http://www.asp.net/
- http://codeguru.com/
- http://www.c-sharpconer.com/
- http://blogs.msdn.com/bclteam/
- http://blogs.msdn.com/ricom/
- http://samgentile.com/blog/
- http://martinfower.com/bliki
- http://blogs.msdn.com/kcwalina/
- http://www.pluralsight.com/blogs/dbox/default.aspx
- http://blogs.msdn.com/cbrumme/

当然这里罗列的并非全部，MSDN、asp.net 自不必说，可以有选择的浏览。

上述 1+7 条，是一些并非经验的经验，谁都知道，但不是谁都能做到。累并快乐着，永远是这个行业，这群人的主旋律。在技术面前，我嫣然一笑，发现自己其实很专注，这就够了。

好了，啰里啰唆，可能是经验，可能是废话。正如一开始所说，作为一个过来人，我只想将自己的心得拿出来交流，绝没有强加于人的想法。除了推荐的几本作品，你可以有选择的参考，其他的甚至可以全盘否定。心怀坦诚的交流，说到底就是希望更多的人少走我曾经曲曲折折的弯路，那条路上实在是幸福与心酸一股脑子毁了一段青春。

祝晚安。