

唱着歌，写着代码，吃火锅。

由此，我就选择了对眼儿的.NET姑娘，在未来的日子比翼双飞，直至今日。

我喜欢托管世界的自由自在，不必为内存管理分心；

我痴迷IL和CLR底层的无限奥秘，把揭开真相作为乐趣；

我欣赏Lambda表达式和LINQ造就的优雅；

我沉醉C#行云流水般的简洁。

Broadview®
www.broadview.com.cn

INSIDE .NET

你必须知道的

.NET

(第2版)



王涛 著



电子工业出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY
<http://www.phei.com.cn>

时间过得太快，
盖茨退休了，
云计算来了，
.NET都论4.0了，
Silverlight可以离线了，
WCF支持Restful了，
MVC借Razor展现，
微软以Windows Phone重整旗鼓，
Facebook连接了全世界，
Google埋头苦干Google+，
而苹果将iPhone扔向全世界，于是
《你必须知道的.NET》
第2版了。



电子工业出版社

Publishing House of Electronics Industry

北京·BEIJING

内 容 简 介

本书来自于微软 MVP 的最新技术心得和感悟，将技术问题以生动易懂的语言展开，层层深入，以例说理。全书主要包括了.NET 基础知识及其深度分析，以.NET Framework 和 CLR 研究为核心展开.NET 本质论述，涵盖了.NET 基本知识几乎所有的重点内容。全书分为 5 个部分，第 1 部分讲述.NET 与面向对象，从底层实现角度分析了.NET 如何实现面向对象机制，进一步分析了面向对象设计原则；第 2 部分论述了.NET 类型系统和 CLR 的内存管理机制，并对 IL 语言进行了相应介绍；第 3 部分论述.NET Framework 框架的方方面面，详细分析了.NET 框架的所有重点、难点和疑点内容，对框架类库的全貌进行了必要的专题性探讨；第 4 部分重点介绍了.NET 泛型和安全性的相关知识和本质解密；第 5 部分对.NET 3.0/3.5/4.0 新特性进行了详细的介绍和引导，对于快速入门.NET 新特性提供了方便之门。

本书适于对.NET 有一定了解的技术学习者、软件工程师和系统架构师阅读，同时也有助于.NET 初学者进行快速提高，可作为大中专院校和.NET 技术培训机构的参考教材。

未经许可，不得以任何方式复制或抄袭本书之部分或全部内容。

版权所有，侵权必究。

图书在版编目（CIP）数据

你必须知道的.NET / 王涛著. —2 版.—北京：电子工业出版社，2011.8

ISBN 978-7-121-14128-7

I. 你… II. 王… III. ①互联网络—程序设计 IV. ①TP393.4

中国版本图书馆 CIP 数据核字（2011）第 144550 号

责任编辑：孙学瑛

文字编辑：江立

印 刷：北京天宇星印刷厂

装 订：三河市皇庄路通装订厂

出版发行：电子工业出版社

北京市海淀区万寿路 173 信箱 邮编 100036

开 本：860×1092 1/16 印张：34.25 字数：700 千字

印 次：2011 年 10 月第 2 次印刷

印 数：4001~7000 册 定价：79.00 元

凡所购买电子工业出版社图书有缺损问题，请向购买书店调换。若书店售缺，请与本社发行部联系，联系及邮购电话：(010) 88254888。

质量投诉请发邮件至 zlts@phei.com.cn，盗版侵权举报请发邮件至 dbqq@phei.com.cn。

服务热线：(010) 88258888。

○○○ 推荐序一

算起来，这是我第三次动笔为这本书写推荐。一开始以为写一个推荐非常容易，但是实际动笔才发现比我想象的要难很多。仿佛我们在准备开发一个系统的时候，实际开发的人都是准备项目可能是困难重重，而旁的人却经常一脸不屑而认为很好完成。牛和鸡的故事（注 1）一次又一次地上演，只不过这一次，我又当牛，又当了鸡。以前也以为写一本书很容易，这主要是源自我经常看到书店里琳琅满目的技术书籍，标榜以“N 天搞定×××”、“×××从入门到精通”以及“玩转×××”，但是每每翻开一看几乎都是官方教材的中文翻译版，或者是某某工作室中十几位同学不断复制粘贴的产物。所以便认为技术类的书籍基本上就是国外资料翻译加国内同行“借鉴”。而鲜有的几本精品往往也淹没在成千上万的图书海洋中，想要找到它们除了自己有孙猴子般火眼金睛的视力和如来佛祖般宽广的人脉推荐，还要有巴菲特一样足够资金支持——在国内图书市场淘到一本好的原创技术书籍，难度不比在潘家园搞到一个宣德炉低。这也是为什么很多人希望国内的技术高人能够肩负起培养下一代的重任，为像我这样的后生多多推荐好的技术书籍的原因。毕竟能够花大笔银子在潘家园买宣德炉的人并不多。

其实我无论如何也没想到王涛会邀请我为他的这部力作写序，而且还是推荐序。一来本人觉得自己能力水平差得太远，自己还需要身边牛人帮我辨识高质量的作品。二来自己在.NET 的圈子里着实算是个新人。虽然近几年也陆续认识了一些高手，但是大都属于对他们高山仰止的状态，所谓身不能至心向往之——这种水平又怎能为别人推荐呢？所以最开始接到王涛的邀请我自然表示力不能及而且层次有限。不过最终还是勉强答应了下来，一方面是整日和王涛胡聊乱侃，不能太折了兄弟的面子；二来，也是主要打动我的原因，我深知这几年他倾注在这部书上的心血。与其让这本好书淹没在一排排“赝品”之中，不如我暂且做个浮标，虽然不及灯塔那么耀眼和挺拔，但是也算增大了它的影响范围，让作为读者的我们更容易看到和知道，而不会被那些粗制滥造的东西蒙蔽了双眼走错了路。

我不清楚翻开这本书的你是否看过了《你必须知道的.NET》第 1 版。如果你认为这本书只是上一本书的添头或者修改那就大错特错了。现在音乐界流行老歌翻唱，几十年前的歌曲，随便换个编曲就可以再卖一次；电影界也是动不动就来个什么什么怀旧版，什么什么经典再映。归根到底就是再从我们这些劳苦大众兜里套点银子出来。但是这本书却不是前一本的所谓“新歌加精选”。虽然我只是看到这部书的两个样章，但是还要惊叹于这本书所涉及的内容之广、见解之深，以至于我看完了样章之后便向王涛提出了个修改意见：一定要加上两个副标题“.NET 程序员面试宝典”和“.NET 应用架构指南”。因为在这本书当中，我看到的不仅仅是和第 1 版一样对于.NET 底层深入的研究和完整的介绍，还能够看到作为一个在.NET 阵营打拼了多年的架构师对于系统架构、设计模式、面向对象等诸多方面的经验、体会与探索。关于某个具体的技术或工具的书籍在国内可能非常普遍，譬如介绍 ASP.NET 的图书可能不下几十种，但是从作者本人经验出发介绍软件设计架构的书籍便是凤毛麟角，偶有几本也是国外图书的翻译版本或者影印版本。而这本书在设计方面的部分我认为是其最大的亮点，没有照本宣科的介绍，没有千篇一律的观点，所有内容都是作者本人的经验分享——有成功的经验，也有失败的经验。这其中可能不免有些内容不尽完美，有些观点尚需推敲，但这正是我们技术

人员所希望看到的：相互交流，集思广益，共同进步。而不是像国内的一些博客站点那样，一遇到观点不同就开始在评论中挖苦鄙视甚至破口大骂。虽然说我们没必要像职业书评家那样，承担着指导读者咒骂作者的使命。所以这样一部呕心沥血的作品，又怎能不让我为之吐血推荐呢？

记得有一次和王涛聊天的时候，我提到了“指月之指”的故事（注 2）。如果说像我这样水平的人写出来的书只能是传递知识的话，那么这本《你必须知道的.NET（第 2 版）》就是在传递智慧。知识只是关于知道和不知道，而智慧是无法传授的，只能自己通过实践的积累慢慢感悟。虽然说和“指月之指”的典故一样，这本书不可能就是软件设计本身，但是正如那指向明月的手指一样，能够让我们可以沿着它的方向去寻找软件设计的精髓。

写到这里，突然心中一凛，这篇推荐序写着写着更多的都是我自己的心情和感受。难道在不经意间我也成了之前所说的“书评家”对这本书开始评头论足起来。还是到此停笔吧，上面的话权当一个疯子在被项目折磨之后的自言自语，书的好坏最终还是要看书的您自己去品评。至少我不想成为《伊索寓言》中所写的那个苍蝇，坐在车轴上嗡嗡大叫：“车的开动，全都是我的功劳”。

徐子岩

2011 年 6 月

推荐人简介

徐子岩，北京工业大学计算机学院毕业。现就职于宇思信德科技（北京）有限公司.NET 开发部架构师、Azure 专家、微软 Windows Azure MVP。精通.NET 平台多项技术，包括 ASP.NET MVC、WCF 等。目前专注于微软 Windows Azure 云计算平台的研究、咨询、设计和开发工作。

注 1：敏捷开发中一个著名的故事，用来说明项目会议是否需要项目组之外的人员参与发言。例如在准备牛排加煎蛋的早餐这个项目中，牛由于是贡献者（贡献自己的肉）所以它的发言是对项目有实际意义的，而鸡只作为参与者（下个蛋完事）所以会提出很多对项目进展不负责任的观点。

注 2：出自《楞伽经》卷四，“如愚见指月，观指不观月；计著名字者，不见我真实。”

○○○ 推荐序二

软件工程师，一个曾经是多么耀眼的职业，如今却沦落为 IT 民工，造成这个局面的原因是多方面的，我不想在这里深入分析。对于软件开发者来说，其心态越来越急躁，不愿意踏踏实实静下心来研究一点技术，做几年的开发工作都争着转向管理方面；而对于 IT 出版业来说，各种粗制烂造的书籍层出不穷，导致软件开发者对国人的书丧失了信心。

让人欣慰的是，国内还有一大批优秀的一线软件工程师，他们仍然保持着对技术的热情，愿意把自己在实践中的经验积累分享给广大软件开发者。本书的作者王涛（Anytao）正是其中一位，屈指算来，我们认识也有五年之久了，从最开始网上认识，到后来成为同事，再后来各自在不同的公司供职，彼此之间的联系并没有中断。我一直比较钦佩 Anytao 对于技术的热情和执着，一个人对技术保持热情不难，难的是把这种热情长期保持下去。

距离本书第 1 版的出版，已经过去三年时间了，在这几年时间里，Anytao 对于.NET 又有了更深的理解，他把自己实践积累的经验和思考整理出书，为广大.NET 爱好者送上了一本精品，这是.NET 爱好者的幸事，也是 IT 出版业的幸事。虽然由于工作原因，我本人现在很少写.NET 方面的程序，但我还是向各位.NET 爱好者强烈推荐本书，如果你真的对软件开发有兴趣，能够在软件开发中体会到编程的使命感和荣誉感，那就静下心来认真读读本书。

最后，在本书出版之际，我和 Anytao 也将踏上人生一段新的征程。在未来的未知世界，继续走在技术的康庄大道上，编织着技术改变世界的梦想。但当我们老去的那一天，再回过头看自己走过的路，一定不会后悔自己现在的选择和对未知的执着，至少我们曾经拼搏过，对酒当歌，夫复何求！

李会军

2011 年 6 月 17 日 于北京

推荐人简介

李会军，网名 TerryLee，互联网公司架构师，编程语言爱好者，关注动态语言及函数式编程，致力于高伸缩高性能网站架构、互联网应用安全、并行计算、分布式存储、分布式计算相关技术的研究。著有《Silverlight 2 完美征程》一书，业余时间喜欢读书，尤其喜好研究历史。

1 版前言 Thinking More

“你站在桥上看风景，看风景的人在楼上看你”。

技术探求，正是如此的富有哲理。在.NET世界里，每个程序设计者都是站在桥头的守望者，渴望品味所有的美景，将技术的各个方面尽收眼底。而现实往往是，你看到的并非全部真实的，技术的理解往往也需要辅助一个望远镜才能看得更加透彻。这本《你必须知道的.NET》既是一本技术的风景画卷，涵盖了.NET基本知识的几乎所有的重点内容；又为你送上手中的望远镜，与作者一起力求对每个技术要点的探讨都更进一步。

走近这幅画卷，除了品味每一处风景，还应学会拨开表象、认识本质、探求细微，更重要的是在这个过程中，你将能收获如何为自己搭建一处技术美景。在楼上看你的人，是否会觉得风景这边独好，就看你的技艺精湛与否了。

面对技术，你别无选择，.NET世界是如此精彩，而我们要做的就是：Thinking More。

本书是什么

对于技术，大部分著作都是从整体角度进行系统性的论述，知识体系一脉相承。拿起这样的书，我们习惯循规蹈矩地从前言看到后记，往往陷入其系统之中，被其思想所固化，而无法找出什么是更值得关注的要点。本书显然不是一本系统性论述技术的专著，因此也无法兼顾.NET技术的所有概念和知识，但是本书力图从重点分析与突出把握的角度来阐释技术，分析问题，将所有.NET开发人员最关心、最困惑的技术内容形成体系进行深度遍历、挖掘和探索。

《你必须知道的.NET》正揭示了这样的一种诉求，将.NET技术中的核心内容以一个个专题的形式来深度刻画，然后形成体系。综观全书内容：一方面，以最少的语言表达最多的技术、体察更深的本质。佛家传道，以例说理，丝丝入扣，环环揭密。本书以“你必须知道”而自诩，唯有意图达到以实例为基点，以归纳为方法的技术论述特点：对于技术的论述和分析，力求做到深入浅出、娓娓道来；对于晦涩艰深的问题以故事性的分析来引导；对于典型的问题以对比的角度来揭密；对于知识性的内容以归纳总结形成纲要。作者对每个技术要点的论述，均结合浅显易懂的实例来展开，将复杂的技术问题化解在循序渐进的思考中。让你的“悟”道，快乐而轻松。

另一方面，.NET技术就是一座美丽的花园，里面开满了各种各样的花朵，就像类型系统、内存机制、垃圾回收、关键字、泛型、安全性、语言特性、框架格局、面向对象等，一支一朵娇艳绽放，要想品味整个花园的芬芳，你就必须了解每朵花的美丽。本书不仅告诉你如何来鉴赏这些花朵，而且告诉你如何通过施肥、除草、浇水来经营这些美丽，一步一步建立对核心技术要点的理解，从而“悟”到整个.NET框架体系和运行机制。

.NET技术正是一个大花园的集合，每个程序开发者也必须经历一次深入的磨练，在基本认识的水平上，

进一步，才能发现更多。就像练武之人，除了研习一招一式，了解常用的控件，了解典型的框架；还得修炼内功，认识运行机制，理解框架类库，品味设计架构。

这些正是本书呈现于读者的内容，也体现了不同于其他.NET专著的风格。

本书有什么

对于.NET来说，应用的范围千头万绪，但至少有一件事必须去做，那就是无限接近和触摸它的内核：CLR，这正是本书所阐述的最核心内容。下面，我们来了解一下《你必须知道的.NET》由哪些绚丽的色彩组成：

- 第一部分：**渊源**，探讨面向对象基本要素和设计原则，建立一个程序设计的基础架构思维，并结合.NET技术来实现相关的面向对象机制，进而探求相关的面向对象原则。从底层角度认识高层本质，是深入理解的不二法门。
- 第二部分：**本质**，在梳理IL基本内容的基础上，了解和掌握探求.NET本质的方法；品味类型系统，了解值类型与引用类型的底层奥秘，揭示参数传递的不惑之解；深入内存管理，认识垃圾回收，以循序渐进的分析，通晓运行时底层机制。
- 第三部分：**格局**，将.NET关键字逐个把玩，深入浅出了解你不知道的关键字秘密；实现巅峰对决，将const和readonly、class和struct、is和as、特性和属性、接口和抽象类、覆写和重载、浅拷贝和深拷贝、静态与非静态以及集合，这些技术重灾区一一澄清，走出理解误区；通过框架诠释，揭开.NET基本技术的本质，深度诠释Object、对象判等、String、枚举、委托和异常等.NET核心话题；最后以命名空间为主线建立对.NET框架的全局纵览，通过梳理命名空间和典型类型，把握.NET框架类库的心脏和骨架。
- 第四部分：**拾遗**，通过对.NET泛型的理解和深入，着重把握建立泛型编程的思维方式；并适度介绍.NET安全性的主要角落，通过对代码访问安全和基于角色的安全论述，来铺陈.NET在安全编程方面的技术体验。
- 第五部分：**未来**，以.NET3.0/3.5新特性为基点，全面阐述.NET新特性的方方面面，在引导性的论述中建立对C#3.0、LINQ、WCF、WPF、WF等新技术和Visual Studio 2008工具的基本认知和学习指导，吹响新技术的号角。

通过5个部分的全面讲述，将基本建立对于面向对象设计与原则、.NET框架体系与运行时机制、.NET框架类库格局与高级特性、.NET安全与新特性的深入理解，对于.NET的认识将在底层把握和设计应用上更进一步。

本书为谁而写

本书起源于作者在国内最专注的.NET技术网站博客园（<http://www.cnblogs.com>）的写作经历，并在博客园的2007年末大盘点Top10的五大排行榜中位列其中3个榜单。作者的系列文章深受大家的关注和讨论，因此本书的内容反映了最直接的技术关注话题，适合于对.NET技术有意进一步提高的所有学习者和开发者。

本书涵盖.NET基本知识的几乎所有的重点内容，如果读者有以下问题、需求或者困惑，那么选择本书非

你莫属：

- 本书并不是从“什么是.NET”这一概念开始的，对于想要了解.NET 基础的读者来说，全书以一个个的专题形式来展开，可以快速建立起对.NET 基本概念的切入。
- 读完了大部头的.NET 巨著，还意犹未尽，抑或是不知所措。本书给你补充未尽的本质，解答未知的困惑，为你迅速进入.NET 底层研究，提供最好的入口。
- 你已经做得够好了，系统地学习了 C# 或者 VB.NET 语言的基础，了解了基本的应用规则，但还是觉得游离于技术之外，并未接触本质。基础研究和高级教程之间往往存在着断层，想在基础之上更进一步，本书可以为你提供更多思考和研究的平台，为你揭开 CLR 的神秘面纱打好基础。
- 对.NET 框架的体系架构和运行机制，有意补充认知的读者，可以通过本书建立起快速的理解。
- 本书没有 ASP.NET，没有 Web Service，也没有.NET Remoting，然而本书的内容对于深刻的理解所有.NET 应用大有裨益。只有从本质上抓住这些基础内容，才能在.NET 应用领域游刃有余，从方法学的角度来看，这才是最有效的技术学习曲线。
- 本书是一部方法论，除了探讨.NET 的基本问题，对.NET 的学习方法和学习工具均有所涉猎。了解一种科学的学习方法，有助于你以更好的质量读完本书，并取得收获。
- 本书是应对技术面试的圣经，综合了来自现实世界的问题和答案，为你快速成长提供了良好的辅助教材。
- 本书并非想创造新的技术和技巧，而是将技术以简单的方式更深一步的讲明白。如果你总是对学习的方法充满了困惑和怀疑，那么以本书作为起点会找到一个更好的方法。
- 对于每个问题的探讨，本书力求深入浅出，让人有胃口读完所关注的话题，并展开思考和讨论。对于厌倦了枯燥论述的读者而言，本书的轻松论述不会让你心感疲惫。

本书如何阅读

关于.NET，本书着眼于基础、本质和方法，对于阅读本书的读者而言，带着思考进行基础和本质的探索，同时也能体验技术学习的有效方法。作者在论述大部分的知识要点时，都会总结和归纳其重要的规律和注意事项，这些归纳为实际的编程提供了良好的遵守法则，读者应该花必要的精力熟练掌握所有的归纳内容。

技术之间是有联系的，平铺直叙的写作和由前到后的阅读都是没有意义的，本书把握从技术的联系点来入手阐述基本知识，从技术的关联中形成有层次的认知角度，能够更加清晰的了解.NET 框架的全局。所以，阅读本书应该在不同的章节间切换，按照作者指引的关联进行跳跃式的阅读，能够收获更多的心得。

关于语言，本书以 C# 语言实现所有的代码示例，这是因为全书虽然以.NET 为核心来论述，但也无可避免的对 C# 语言的某些特性进行了分析。从广义的角度来看，C# 语言本身也是.NET 体系中不可分割的一部分，对于某些语言特性的了解也能从更全面的角度来透视.NET 框架。

关于代码，读者可以通过 <http://www.broadview.com.cn> 或 <http://book.anytao.com> 来下载本书的源代码，解压缩之后按照代码使用说明，通过 Visual Studio 工具进行编译和调试。

支持

虽然作者、审稿和编辑花费了大量的时间对书稿进行了反复的修改和推敲，但是限于时间和水平，仍难免避免失误或错误。为了使本书能更好地服务于读者，请您将关于本书的任何错误信息发至以下任何链接：

- 作者个人邮箱：anytao@live.com
- 作者个人微博：<http://weibo.com/anytao>
- 本书支持网站：<http://book.anytao.net/>
- 博文视点网络：<http://www.broadview.com.cn/>

我们将竭力解决所有的问题，并向您的指正致谢。读者可以在本书的支持网站中查找相应的勘误表来避免错误。您也可以通过邮件或者作者博客（<http://anytao.cnblogs.com/>）进一步取得技术支持联系。

本书支持网站提供了所有代码资源、工具资源及其他导航信息支持，这些资源和信息是对全书内容的有效补充与最佳辅助。

致谢

首先感谢为本书审稿的蒋金楠，他的技术功底和专业素质令我钦佩，他的审阅和建议为本书增色不少，这本书有他的心血和付出。

本书的出版离不开我在博客园的成长和锻炼，感谢杜勇（dudu）站长为.NET技术人员提供了难得的纯学术环境和氛围，感谢所有在博客园中与我笑谈技术、品论人生的朋友；感谢蒋金楠与我一起创建和支持 CLR 研究团队；感谢杜勇、李会军、程杰、刘彦博、张大磊几位朋友在百忙中对本书的审阅及点评；感谢装配脑袋、Jeffrey Zhao、Bruce Zhang 对我的指导和帮助；感谢阿不、宋国安、Volnet、Justin、EagleFish、刘荣华、Jill Zhang、随风流月、丁学、怪怪等对本书的建议和关注；还要感谢我的朋友吴宏杰、管伟、高泽东、党明、达伟对我一直以来的支持。

将最重要的感激送给养育我的父母和伴我成长的妹妹王佳，慈母严父是我人生的灯塔，激励我努力前行。感谢岳父岳母对我的关心和爱护，并将爱送给 Emma，感谢她每天在身边的鼓励与关怀，品尝她愈发炉火纯青的厨艺，让我的思绪在逻辑和理性间飞舞。

最后要感谢电子工业出版社孙学瑛编辑，正是她的不懈努力和不断支持才使我的写书过程充满了自信和快乐。还有对本书投入精力、提出建议的胡辛征编辑和其他博文视点同仁，他们的专业素质和敬业精神令我感动，才使得本书有机会服务于大众。

这本《你必须知道的.NET》送给所有技术之路上的同伴，让我们一起远航。进一步，你便是大内（dotnet）高手。

王涛

2008年1月，于北京

目 录

第1部分 源源——.NET 与面向对象

第1章 OO 大智慧	2	1.4.7 结论	33
1.1 对象的旅行.....	3	1.5 玩转接口	34
1.1.1 引言	3	1.5.1 引言	34
1.1.2 出生	3	1.5.2 什么是接口	34
1.1.3 旅程	3	1.5.3 .NET 中的接口	35
1.1.4 插曲	4	1.5.4 面向接口的编程	38
1.1.5 消亡	6	1.5.5 接口之规则	40
1.1.6 结论	7	1.5.6 结论	40
1.2 什么是继承.....	7	参考文献	40
1.2.1 引言	7		
1.2.2 基础为上	7	第2章 OO 大原则	41
1.2.3 继承本质论	9	2.1 OO 原则综述	42
1.2.4 秘境追踪	13	2.1.1 引言	42
1.2.5 规则制胜	16	2.1.2 讲述之前	42
1.2.6 结论	17	2.1.3 原则综述	43
1.3 封装的秘密.....	17	2.1.4 学习建议	44
1.3.1 引言	17	2.1.5 结论	44
1.3.2 让 ATM 告诉你，什么是封装	17	2.2 单一职责原则	44
1.3.3 秘密何处：字段、属性和方法	19	2.2.1 引言	44
1.3.4 封装的意义	23	2.2.2 引经据典	45
1.3.5 封装规则	23	2.2.3 应用反思	45
1.3.6 结论	24	2.2.4 规则建议	47
1.4 多态的艺术.....	24	2.2.5 结论	48
1.4.1 引言	24	2.3 开放封闭原则	48
1.4.2 问题的抛出	24	2.3.1 引言	48
1.4.3 最初的实现	25	2.3.2 引经据典	48
1.4.4 多态，救命的稻草	27	2.3.3 应用反思	49
1.4.5 随需而变的业务	30	2.3.4 规则建议	52
1.4.6 多态的类型、本质和规则	31	2.3.5 结论	53
		2.4 依赖倒置原则	53

2.4.1 引言	53	3.2.4 解构控制反转 (IoC) 和依赖注入 (DI)	79
2.4.2 引经据典	53	3.2.5 典型的设计模式	82
2.4.3 应用反思	54	3.2.6 基于契约编程：SOA 架构下的依赖	83
2.4.4 规则建议	56	3.2.7 对象创建的依赖	84
2.4.5 结论	56	3.2.8 不规则总结	87
2.5 接口隔离原则	56	3.2.9 结论	87
2.5.1 引言	56	3.3 模式的起点	87
2.5.2 引经据典	56	3.3.1 引言	87
2.5.3 应用反思	57	3.3.2 模式的起点	88
2.5.4 规则建议	59	3.3.3 模式的建议	90
2.5.5 结论	59	3.3.4 结论	91
2.6 Liskov 替换原则	59	3.4 面向对象和基于对象	91
2.6.1 引言	59	3.4.1 引言	91
2.6.2 引经据典	59	3.4.2 基于对象	91
2.6.3 应用反思	60	3.4.3 二者的差别	91
2.6.4 规则建议	61	3.4.4 结论	92
2.6.5 结论	62	3.5 也谈.NET 闭包	92
参考文献	62	3.5.1 引言	92
第3章 OO之美	63	3.5.2 什么是闭包	92
3.1 设计的分寸	64	3.5.3 .NET 也有闭包	93
3.1.1 引言	64	3.5.4 福利与问题	95
3.1.2 设计由何而来	64	3.5.5 结论	96
3.1.3 从此重构	65	3.6 好代码和坏代码	96
3.1.4 结论	67	3.6.1 引言	96
3.2 依赖的哲学	67	3.6.2 好代码、坏代码	97
3.2.1 引言	67	3.6.3 结论	105
3.2.2 什么是依赖，什么是抽象	68	参考文献	105
3.2.3 重新回到依赖倒置	73		

第2部分 本质——.NET 深入浅出

第4章 一切从 IL 开始	108	4.2.1 引言	113
4.1 从 Hello, world 开始认识 IL	109	4.2.2 使用工具	113
4.1.1 引言	109	4.2.3 为何而探索	115
4.1.2 从 Hello, world 开始	109	4.2.4 结论	116
4.1.3 IL 体验中心	109	4.3 教你认识 IL 代码——IL 语言基础	116
4.1.4 结论	113	4.3.1 引言	116
4.2 教你认识 IL 代码——从基础到工具	113	4.3.2 变量的声明	116
		4.3.3 基本类型	117

4.3.4	基本运算	118	5.2.4	对症下药——应用场合与注意事项	158
4.3.5	数据加载与保存	118	5.2.5	再论类型判等	159
4.3.6	流程控制	119	5.2.6	再论类型转换	159
4.3.7	结论	120	5.2.7	以代码剖析	160
4.4	管窥元数据和 IL	120	5.2.8	结论	167
4.4.1	引言	120	5.3	参数之惑——传递的艺术	167
4.4.2	初次接触	120	5.3.1	引言	168
4.4.3	继续深入	123	5.3.2	参数基础论	168
4.4.4	元数据是什么	125	5.3.3	传递的基础	169
4.4.5	IL 是什么	128	5.3.4	深入讨论，传递的艺术	170
4.4.6	元数据和 IL 在 JIT 编译时	130	5.3.5	结论	174
4.4.7	结论	134	5.4	皆有可能——装箱与拆箱	175
4.5	经典指令解析之实例创建	134	5.4.1	引言	175
4.5.1	引言	134	5.4.2	品读概念	176
4.5.2	newobj 和 initobj	134	5.4.3	原理分析	176
4.5.3	ldstr	136	5.4.4	还是性能	179
4.5.4	newarr	137	5.4.5	重在应用	180
4.5.5	结论	139	5.4.6	结论	182
4.6	经典指令解析之方法调度	140	参考文献		182
4.6.1	引言	140	第 6 章 内存天下		184
4.6.2	方法调度简论：call、callvirt 和 calli	140	6.1	内存管理概要	185
4.6.3	直接调度	142	6.1.1	引言	185
4.6.4	间接调度	146	6.1.2	内存管理概观要论	185
4.6.5	动态调度	147	6.1.3	结论	186
4.6.6	结论	147	6.2	对象创建始末	186
参考文献		147	6.2.1	引言	187
第 5 章 品味类型		148	6.2.2	内存分配	187
5.1	品味类型——从通用类型系统开始	149	6.2.3	结论	193
5.1.1	引言	149	6.3	垃圾回收	193
5.1.2	基本概念	149	6.3.1	引言	193
5.1.3	位置与关系	150	6.3.2	垃圾回收	193
5.1.4	通用规则	151	6.3.3	非托管资源清理	197
5.1.5	结论	152	6.3.4	结论	204
5.2	品味类型——值类型与引用类型	152	6.4	性能优化的多方探讨	204
5.2.1	引言	152	6.4.1	引言	204
5.2.2	内存有理	152	6.4.2	性能条款	204
5.2.3	通用规则与比较	156	6.4.3	结论	210
参考文献			参考文献		211

第3部分 格局——.NET 面面俱到

第7章 深入浅出——关键字的秘密	214	7.6.7 结论	245
7.1 把 new 说透	215	7.7 非主流关键字	245
7.1.1 引言	215	7.7.1 引言	245
7.1.2 基本概念	215	7.7.2 checked/unchecked	246
7.1.3 深入浅出	217	7.7.3 yield	247
7.1.4 结论	219	7.7.4 lock	250
7.2 base 和 this	219	7.7.5 unsafe	252
7.2.1 引言	219	7.7.6 sealed	253
7.2.2 基本概念	219	7.7.7 结论	254
7.2.3 深入浅出	220	参考文献	254
7.2.4 通用规则	224		
7.2.5 结论	224	第8章 巅峰对决——走出误区	255
7.3 using 的多重身份	224	8.1 什么才是不变：const 和 readonly	256
7.3.1 引言	224	8.1.1 引言	256
7.3.2 引入命名空间	225	8.1.2 从基础到本质	257
7.3.3 创建别名	225	8.1.3 比较，还是规则	259
7.3.4 强制资源清理	227	8.1.4 进一步的探讨	260
7.3.5 结论	230	8.1.5 结论	263
7.4 认识全面的 null	230	8.2 后来居上：class 和 struct	263
7.4.1 引言	230	8.2.1 引言	263
7.4.2 从什么是 null 开始	230	8.2.2 基本概念	263
7.4.3 Nullable<T>（可空类型）	232	8.2.3 相同点和不同点	264
7.4.4 ??运算符	234	8.2.4 经典示例	265
7.4.5 Null Object 模式	235	8.2.5 结论	268
7.4.6 结论	238	8.3 历史纠葛：特性和属性	268
7.5 转换关键字	238	8.3.1 引言	268
7.5.1 引言	239	8.3.2 概念引入	268
7.5.2 自定义类型转换探讨	239	8.3.3 通用规则	270
7.5.3 本质分析	240	8.3.4 特性的应用	271
7.5.4 结论	242	8.3.5 示例	273
7.6 预处理指令关键字	242	8.3.6 结论	277
7.6.1 引言	242	8.4 面向抽象编程：接口和抽象类	277
7.6.2 预处理指令简述	242	8.4.1 引言	277
7.6.3 #if、#else、#elif、#endif	243	8.4.2 概念引入	277
7.6.4 #define、#undef	244	8.4.3 相同点和不同点	279
7.6.5 #warning、#error	244	8.4.4 经典示例	281
7.6.6 #line	245	8.4.5 他山之石	283
		8.4.6 结论	283

8.5	恩怨情仇: is 和 as	284	9.2.1	引言	326
8.5.1	引言	284	9.2.2	本质分析	326
8.5.2	概念引入	284	9.2.3	覆写 Equals 方法	329
8.5.3	原理与示例说明	284	9.2.4	与 GetHashCode 方法同步	331
8.5.4	结论	285	9.2.5	规则	332
8.6	貌合神离: 覆写和重载	286	9.2.6	结论	332
8.6.1	引言	286	9.3	疑而不惑: interface “继承” 争议	332
8.6.2	认识覆写和重载	286	9.3.1	引言	332
8.6.3	在多态中的应用	288	9.3.2	从面向对象寻找答案	333
8.6.4	比较, 还是规则	289	9.3.3	以 IL 探求究竟	334
8.6.5	进一步的探讨	290	9.3.4	System.Object 真是 “万物之宗” 吗	334
8.6.6	结论	292	9.3.5	接口的继承争议	335
8.7	有深有浅的克隆: 浅拷贝和深拷贝	292	9.3.6	结论	335
8.7.1	引言	292	9.4	给力细节: 深入类型构造器	336
8.7.2	从对象克隆说起	292	9.4.1	引言: 一个故事	336
8.7.3	浅拷贝和深拷贝的实现	294	9.4.2	认识对象构造器和类型构造器	337
8.7.4	结论	296	9.4.3	深入执行过程	339
8.8	动静之间: 静态和非静态	296	9.4.4	回归故事	341
8.8.1	引言	296	9.4.5	结论	342
8.8.2	一言蔽之	297	9.5	如此特殊: 大话 String	342
8.8.3	分而治之	297	9.5.1	引言	342
8.8.4	结论	302	9.5.2	问题迷局	343
8.9	集合通论	302	9.5.3	什么是 string	345
8.9.1	引言	302	9.5.4	字符串创建	345
8.9.2	中心思想——纵论集合	303	9.5.5	字符串恒定性	346
8.9.3	各分秋色——.NET 集合类大观	307	9.5.6	字符串驻留 (String Interning)	346
8.9.4	自我成全——实现自定义集合	314	9.5.7	字符串操作典籍	350
8.9.5	结论	317	9.5.8	补充的礼物: StringBuilder	352
	参考文献	317	9.5.9	结论	354
第 9 章	本来面目——框架诠释	318	9.6	简易不简单: 认识枚举	354
9.1	万物归宗: System.Object	319	9.6.1	引言	355
9.1.1	引言	319	9.6.2	枚举类型解析	355
9.1.2	初识	319	9.6.3	枚举种种	358
9.1.3	分解	320	9.6.4	位枚举	360
9.1.4	插曲: 消失的成员	323	9.6.5	规则与意义	361
9.1.5	意义	325	9.6.6	结论	361
9.1.6	结论	325	9.7	一脉相承: 委托、匿名方法和 Lambda 表达式	362
9.2	规则而定: 对象判等	325			

9.7.1	引言	362	10.1.1	引言	384
9.7.2	解密委托	362	10.1.2	框架概览	384
9.7.3	委托和事件	365	10.1.3	历史变迁	385
9.7.4	匿名方法	367	10.1.4	结论	387
9.7.5	Lambda 表达式	368	10.2	布局——框架类库研究	387
9.7.6	规则	368	10.2.1	引言	387
9.7.7	结论	369	10.2.2	为什么了解	388
9.8	Name 这回事儿	369	10.2.3	框架类库的格局	388
9.8.1	引言	369	10.2.4	一点补充	389
9.8.2	畅聊 Name	369	10.2.5	结论	390
9.8.3	回到问题	371	10.3	根基——System 命名空间	391
9.8.4	结论	371	10.3.1	引言	391
9.9	直面异常	371	10.3.2	从基础类型说起	391
9.9.1	引言	372	10.3.3	基本服务	392
9.9.2	为何而抛	372	10.3.4	结论	394
9.9.3	从 try/catch/finally 说起：解析异常机制	375	10.4	核心——System 次级命名空间	394
9.9.4	.NET 系统异常类	377	10.4.1	引言	394
9.9.5	定义自己的异常类	379	10.4.2	System.IO	395
9.9.6	异常法则	381	10.4.3	System.Diagnostics	396
9.9.7	结论	382	10.4.4	System.Runtime.Serialization 和 System.Xml.Serialization	397
	参考文献	382	10.4.5	结论	399
	第 10 章 格局之选——命名空间剖析	383		参考文献	399
10.1	基础——.NET 框架概览	384			

第 4 部分 捡遗——.NET 也有春天

	第 11 章 接触泛型	402		11.3.1	引言	411
11.1	追溯泛型	403		11.3.2	泛型方法	411
	11.1.1 引言	403		11.3.3	泛型接口	413
	11.1.2 推进思维，为什么泛型	403		11.3.4	泛型委托	415
	11.1.3 解析泛型——运行时本质	405		11.3.5	结论	415
	11.1.4 结论	406	11.4	实践泛型	416	
11.2	了解泛型	406		11.4.1	引言	416
	11.2.1 引言	406		11.4.2	最佳实践	416
	11.2.2 领略泛型——基础概要	406		11.4.3	结论	421
	11.2.3 典型.NET 泛型类	409		参考文献	421	
	11.2.4 基础规则	410				
	11.2.5 结论	411				
11.3	深入泛型	411	第 12 章 如此安全性		422	
			12.1	怎么样才算是安全	423	

12.1.1	引言	423
12.1.2	怎么样才算安全	423
12.1.3	.NET 安全模型	423
12.1.4	结论	424
12.2	代码访问安全	424
12.2.1	引言	424
12.2.2	证据 (Evidence)	425
12.2.3	权限 (Permission) 和权限集	426
12.2.4	代码组 (Code Group)	428
12.2.5	安全策略 (Security Policy)	428
12.2.6	规则总结	429
12.2.7	结论	430
12.3	基于角色的安全	430
12.3.1	引言	430
12.3.2	Principal (主体)	430
12.3.3	Identity (标识)	431
12.3.4	PrincipalPermission	432
12.3.5	应用示例	432
12.3.6	结论	433
	参考文献	433

第 5 部分 未来——.NET 技术展望

第 13 章	走向.NET 3.0/3.5 变革	436
13.1	品读新特性	437
13.1.1	引言	437
13.1.2	.NET 新纪元	437
13.1.3	程序语言新特性	438
13.1.4	WPF、WCF、WF	438
13.1.5	Visual Studio 2008 体验	439
13.1.6	其他	439
13.1.7	结论	439
13.2	赏析 C# 3.0	439
13.2.1	引言	440
13.2.2	对象初始化器 (Object Initializers)	440
13.2.3	集合初始化器 (Collection Initializers)	441
13.2.4	自动属性 (Automatic Properties)	442
13.2.5	隐式类型变量 (Implicitly Typed Local Variables) 和 隐式类型数组 (Implicitly Typed Array)	444
13.2.6	匿名类型 (Anonymous Type)	445
13.2.7	扩展方法 (Extension Methods)	446
13.2.8	查询表达式 (Query Expressions)	448
13.2.9	结论	448
13.3	LINQ 体验	449
13.3.1	引言	449
13.3.2	LINQ 概览	449
13.3.3	查询操作符	451
13.3.4	LINQ to XML 示例	451
13.3.5	规则	453
13.3.6	结论	453
13.4	LINQ 江湖	453
13.4.1	引言	453
13.4.2	演义	453
13.4.3	基于 LINQ 的零代码数据访问 层实现	459
13.4.4	LINQ to Provider	462
13.4.5	结论	463
13.5	抢鲜 Visual Studio 2008	463
13.5.1	引言	463
13.5.2	Visual Studio 2008 概览	464
13.5.3	新特性简介	465
13.5.4	开发示例	465
13.5.5	结论	466
13.6	江湖一统：WPF、WCF、WF	467
13.6.1	引言	467
13.6.2	WPF	467
13.6.3	WCF	468
13.6.4	WF	469
13.6.5	结论	470
	参考文献	470

第 14 章 跟随.NET 4.0 脚步	472		
14.1 .NET 十年	473	14.5.1 引言	497
14.1.1 引言	473	14.5.2 一览究竟	498
14.1.2 历史脚步	473	14.5.3 简单应用	499
14.1.3 未来之变	477	14.5.4 结论	499
14.1.4 结论	479	14.6 协变与逆变	500
14.2 .NET 4.0, 第一眼	480	14.6.1 引言	500
14.2.1 引言	480	14.6.2 概念解析	500
14.2.2 第一眼	481	14.6.3 深入	502
14.2.3 结论	484	14.6.4 结论	504
14.3 动态变革: dynamic	484	14.7 Lazy<T>点滴	504
14.3.1 引言	484	14.7.1 引言	505
14.3.2 初探	485	14.7.2 延迟加载	505
14.3.3 本质: DLR	485	14.7.3 Lazy<T>登场	505
14.3.4 PK 解惑	488	14.7.4 Lazy<T>本质	507
14.3.5 应用: 动态编程	490	14.7.5 结论	509
14.3.6 结论	491	14.8 Tuple 一二	509
14.4 趋势必行, 并行计算	491	14.8.1 引言	509
14.4.1 引言	491	14.8.2 Tuple 为何物	510
14.4.2 拥抱并行	492	14.8.3 Tuple Inside	511
14.4.3 TPL	493	14.8.4 优略之间	513
14.4.4 PLINQ	495	14.8.5 结论	514
14.4.5 并行补遗	496	参考文献	514
14.4.6 结论	497	后记: 我写的不是代码	516
14.5 命名参数和可选参数	497	编后记: 遇见幸福	521

第1部分

渊源——.NET 与面向对象

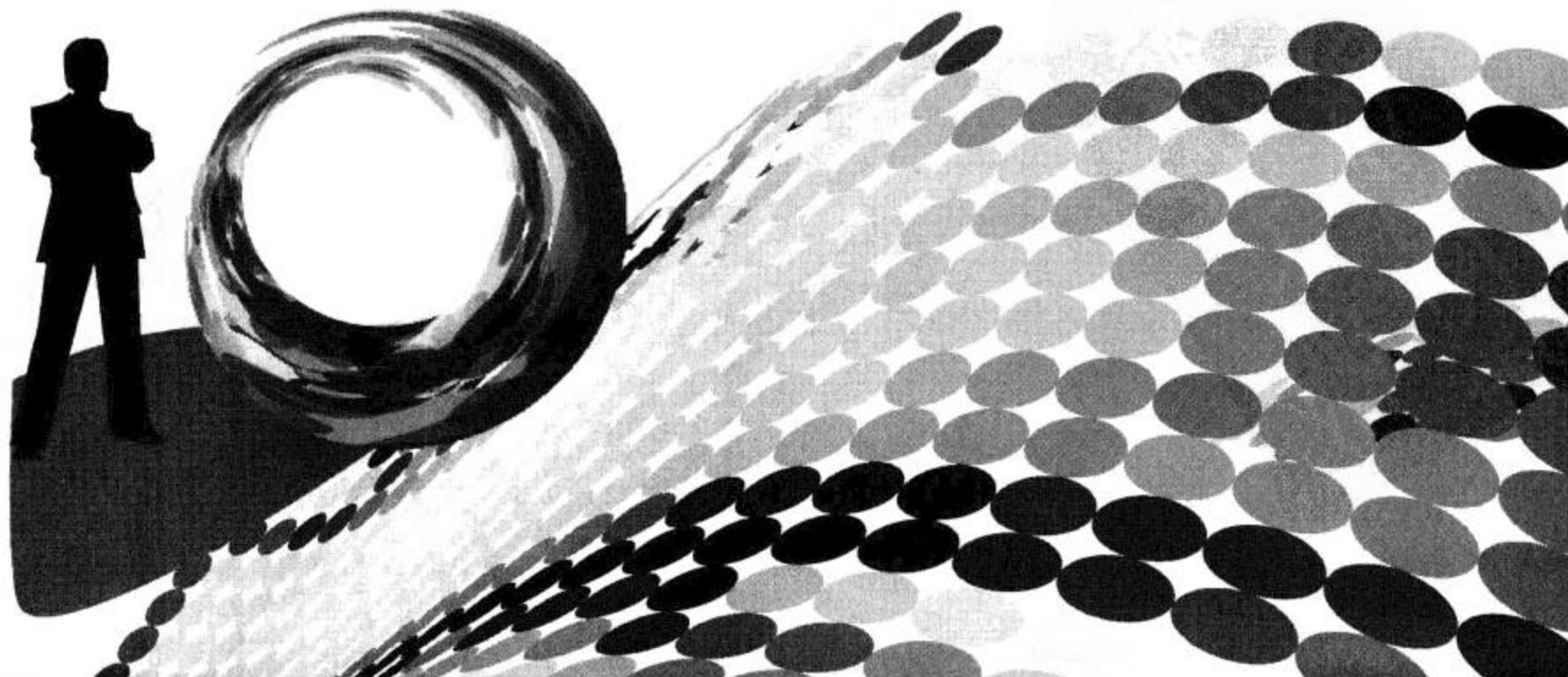
在.NET的世界里，一切都是对象。面向对象思想的掌握是深入理解.NET技术的必修课，在.NET Framework的高楼大厦中，是什么在支撑着各个复杂的系统相互有效地协作，请面向对象这位设计师告诉你答案。在本书的第1部分中，我们以面向对象技术在.NET中的应用为起点，来熟悉和领略面向对象的智慧与原则，修炼深入.NET技术的基本功。磨刀不误砍柴工的道理，谁都清楚，那么在进入.NET技术细节的讨论之前，请先磨好面向对象这把宝刀吧。

本部分主要包括：

- 第1章 OO大智慧
- 第2章 OO大原则
- 第3章 OO之美

第1章 OO 大智慧

1.1 对象的旅行 / 3	1.3.4 封装的意义 / 23
1.1.1 引言 / 3	1.3.5 封装规则 / 23
1.1.2 出生 / 3	1.3.6 结论 / 24
1.1.3 旅程 / 3	1.4 多态的艺术 / 24
1.1.4 插曲 / 4	1.4.1 引言 / 24
1.1.5 消亡 / 6	1.4.2 问题的抛出 / 24
1.1.6 结论 / 7	1.4.3 最初的实现 / 25
1.2 什么是继承 / 7	1.4.4 多态，救命的稻草 / 27
1.2.1 引言 / 7	1.4.5 随需而变的业务 / 30
1.2.2 基础为上 / 7	1.4.6 多态的类型、本质和规则 / 31
1.2.3 继承本质论 / 9	1.4.7 结论 / 33
1.2.4 秘境追踪 / 13	1.5 玩转接口 / 34
1.2.5 规则制胜 / 16	1.5.1 引言 / 34
1.2.6 结论 / 17	1.5.2 什么是接口 / 34
1.3 封装的秘密 / 17	1.5.3 .NET 中的接口 / 35
1.3.1 引言 / 17	1.5.4 面向接口的编程 / 38
1.3.2 让 ATM 告诉你，什么是封装 / 17	1.5.5 接口之规则 / 40
1.3.3 秘密何处：字段、属性和方法 / 19	1.5.6 结论 / 40
	参考文献 / 40



1.1 对象的旅行

本节将介绍以下内容：

- 面向对象的基本概念
- .NET 基本概念评述
- 通用类型系统

1.1.1 引言

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。

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。

对象和人，两个世界，一样情怀。

1.1.2 出生

对象就像个体的人，生而入世，死而离世。

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：

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

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值，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。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 访问修饰符

访问修饰符	访问权限
public	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
{
}
```

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：

```
IPerson aPerson = new PersonAtHome();
aPerson.DoWork();
```

另一个角色又是这个样子：

```
IPerson bPerson = new PersonOnBus();
bPerson.DoWork();
```

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。

这和我们的规则多么一致呀！

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：

- 接口实现多态，就像上例所示。
- 抽象类实现多态，就是以抽象类来实现。

其细节我们将在 1.4 节“多态的艺术”中加以详细讨论。

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。

1.1.5 消亡

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。

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 6.3 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。

1.1.6 结论

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。

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。

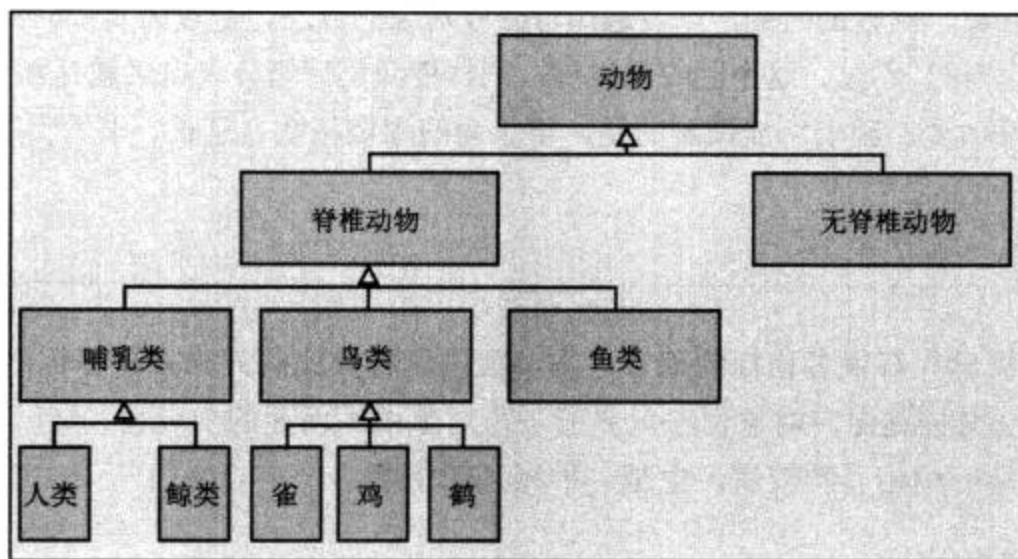


图1-1 继承关系图

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。

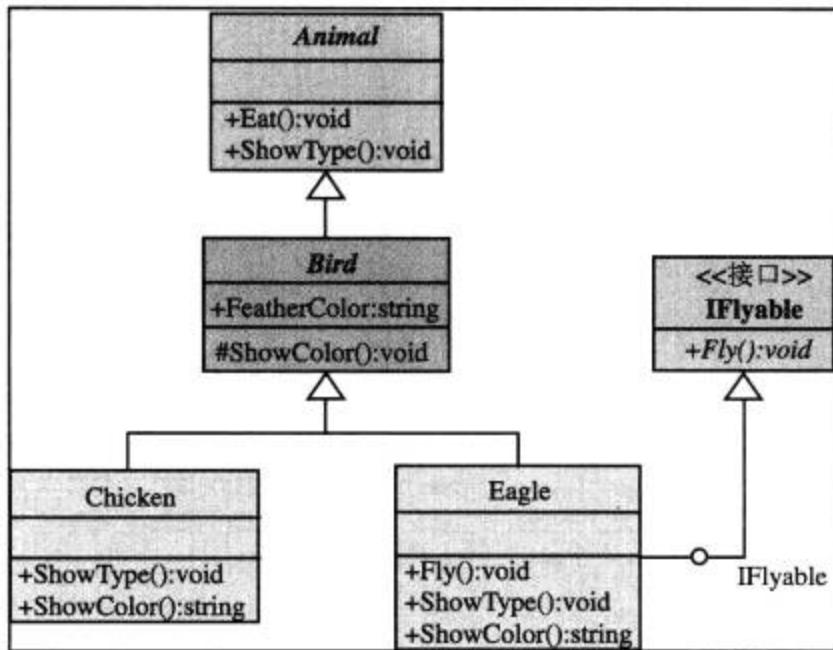


图 1-2 动物系统 UML

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，这样我们无须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 的所有虚方法复制一份，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 9.1 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。

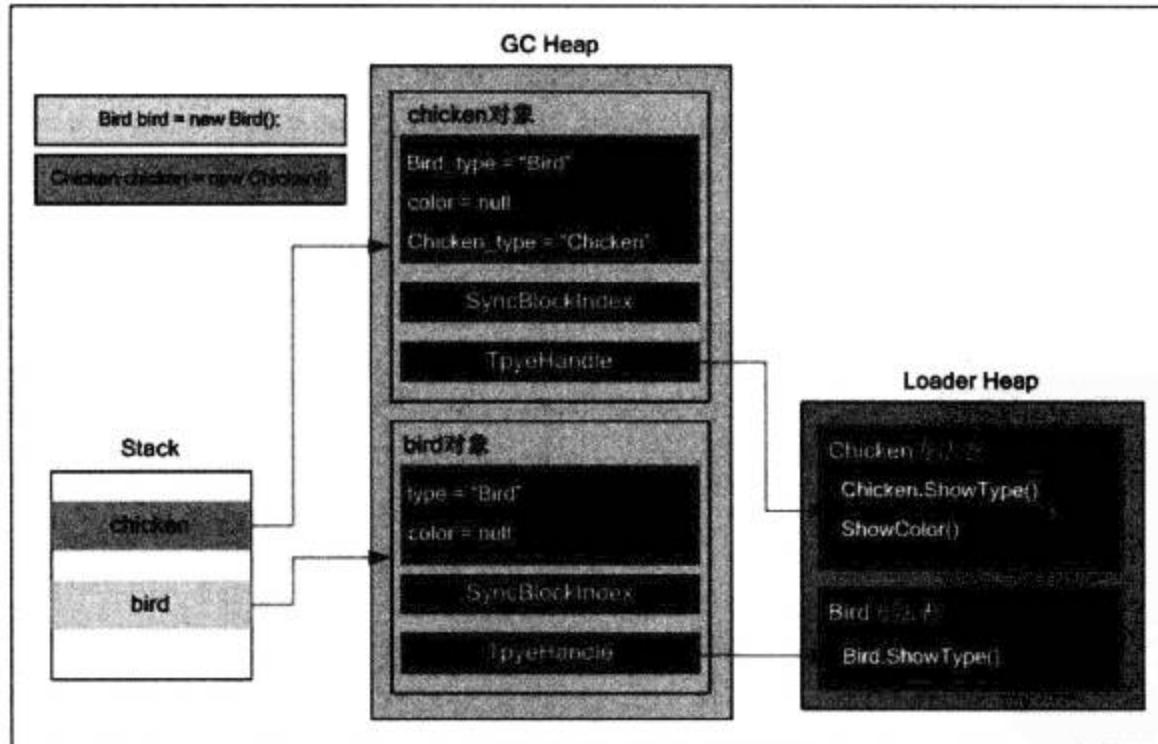


图 1-3 对象创建内存概括

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。引用类的区别决定了不同的对象在方法表中不同的访问权限。

1 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。

2 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 秘境追踪

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，就像追踪继承中的秘境，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 8.4 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。

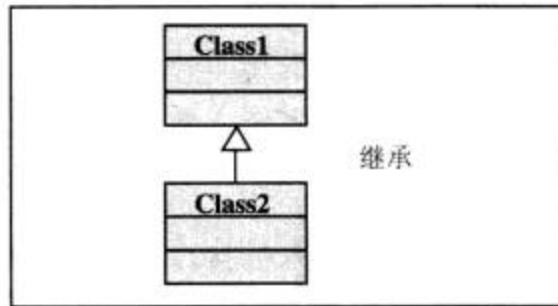


图 1-4 继承关系

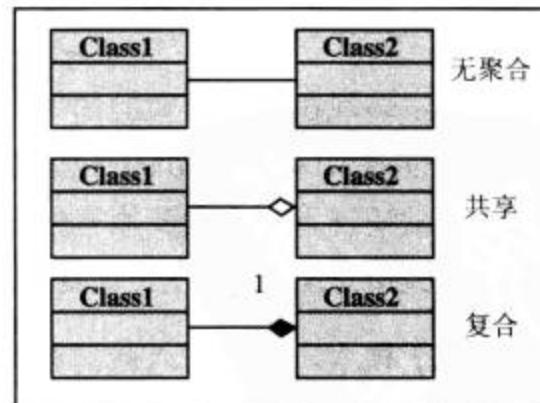


图 1-5 聚合关系

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。

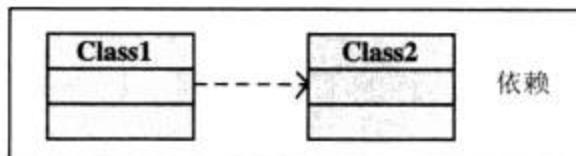


图 1-6 依赖关系

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。

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，通常情况下在不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音符。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 Adapter 模式，其设计 UML 图表示为图 1-7。

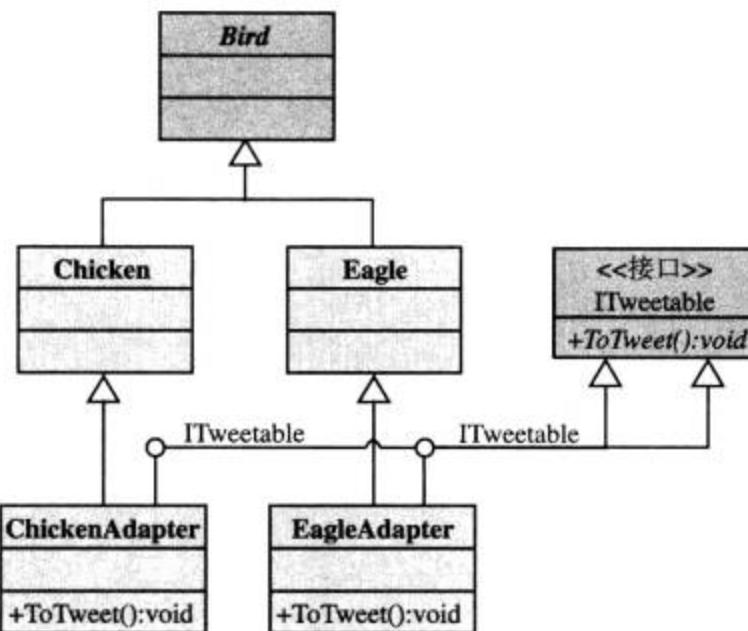


图 1-7 类的 Adapter 模式

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 所示，原有继承体系中的方法和新的方法对对象 ca 都是可见的。

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 1-8 ToTweet 方法的智能感知

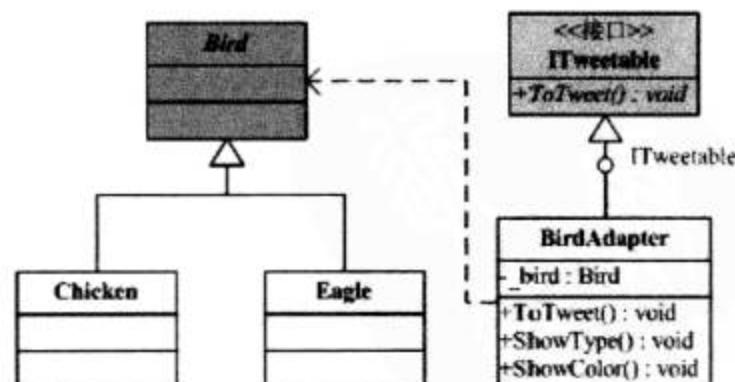


图 1-9 对象的 Adapter 模式

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。

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 Framework 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.WriteLine("你不是合法用户，是不是想被发配南极？");
            }
        }
        #endregion
    }

    /// <summary>
    /// 用户类
    /// </summary>
}
```

```

/// </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一“@&；”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 ArgumentException ("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。

所以我们也就可以定义自己的 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，保证接口稳定性是保证软件兼容、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秘旅程。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，不断地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（Polymorphism）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 所示。

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。

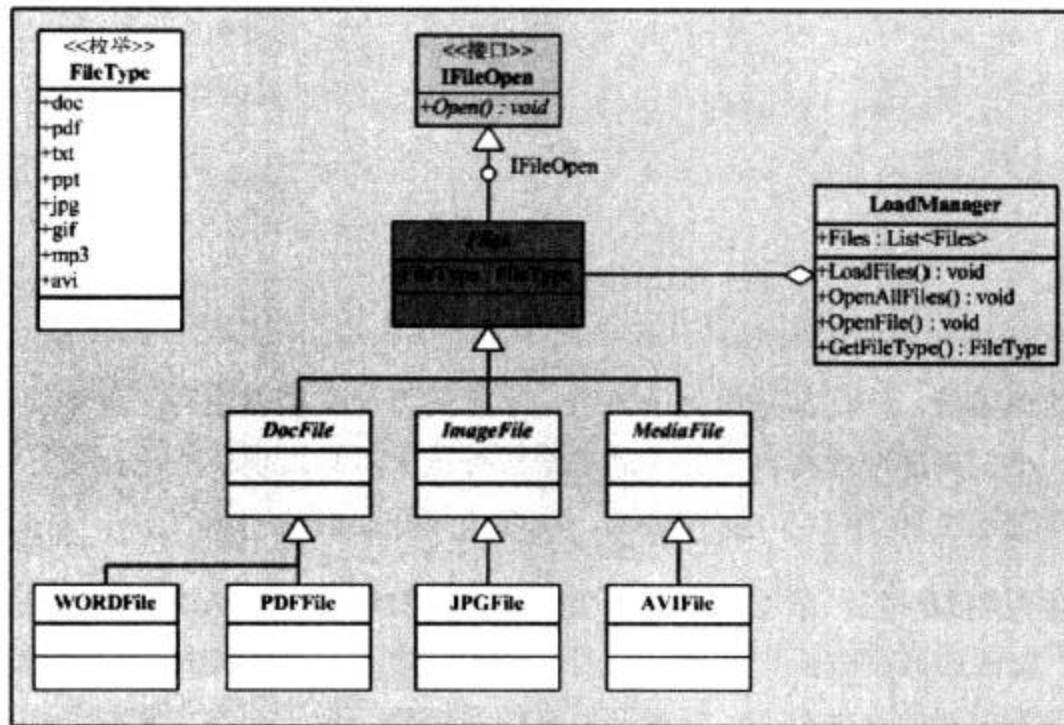


图 1-12 万能加载器系统设计

- 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。
- 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，甚至是无须编译的插件式系统。

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。

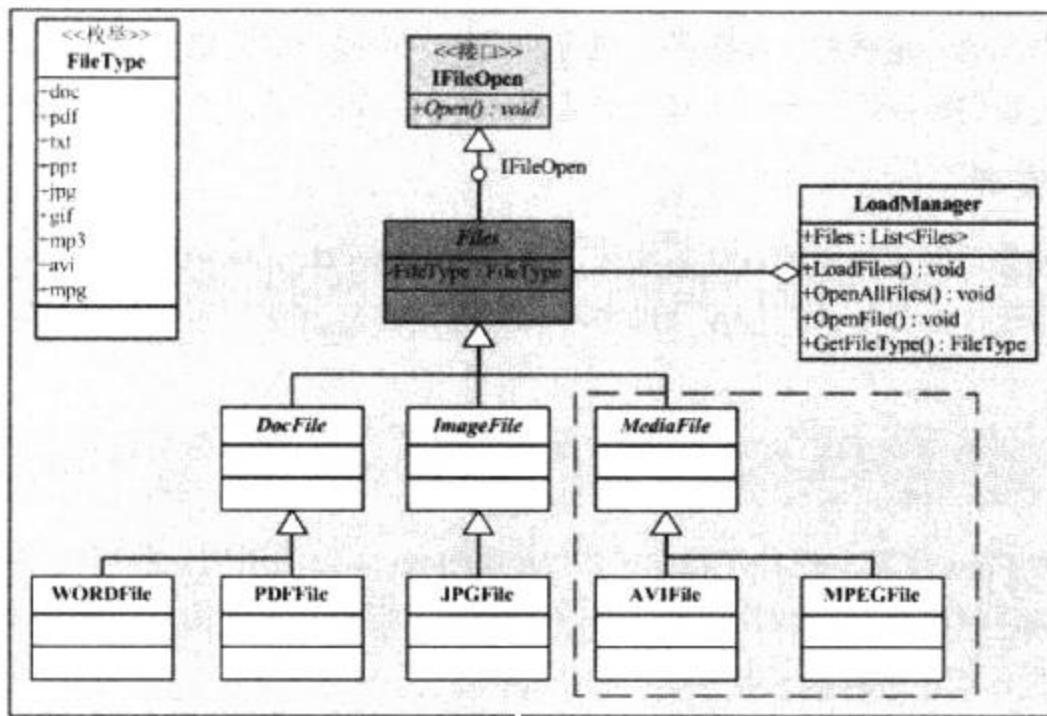


图1-13 万能加载器架构设计调整

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.");
    }
}

```

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基础上，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，并且无须编译即可更新扩展。

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 1.2 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 8.4 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(
        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做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.IMyInterface.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 11.3 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); intToInt32(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，用于显式的释放非托管资源

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 8.9 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进行扩展，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。

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。

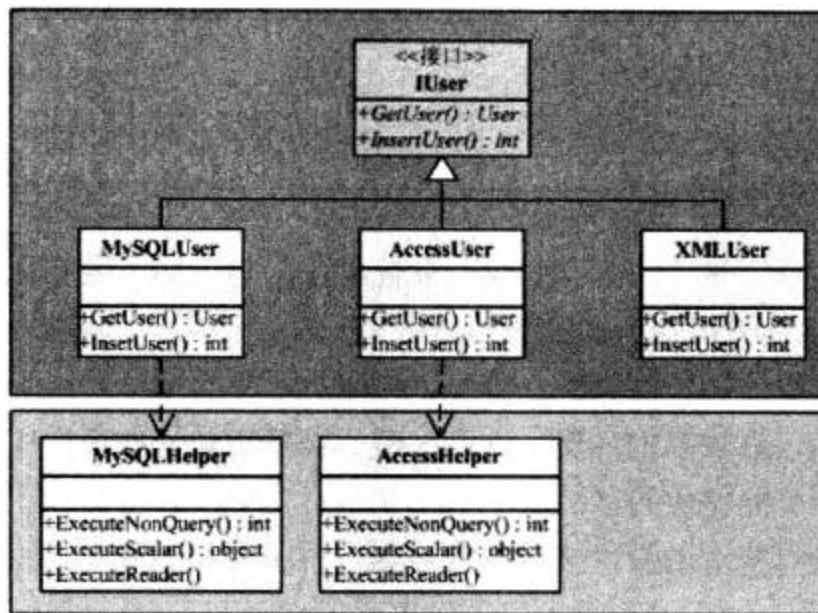


图 1-14 基于 Petshop 的数据访问层设计

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。

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，是否玩得过瘾，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。

参考文献

David Chappell, Understanding .NET

Robert C. Martin, 敏捷软件开发：原则、模式与实践

张逸, 软件设计精要与模式, 电子工业出版社

Jacquie Barker, Grant Palmer, Beginning C# Objects

Don Box, Chris Sells, Essential .NET

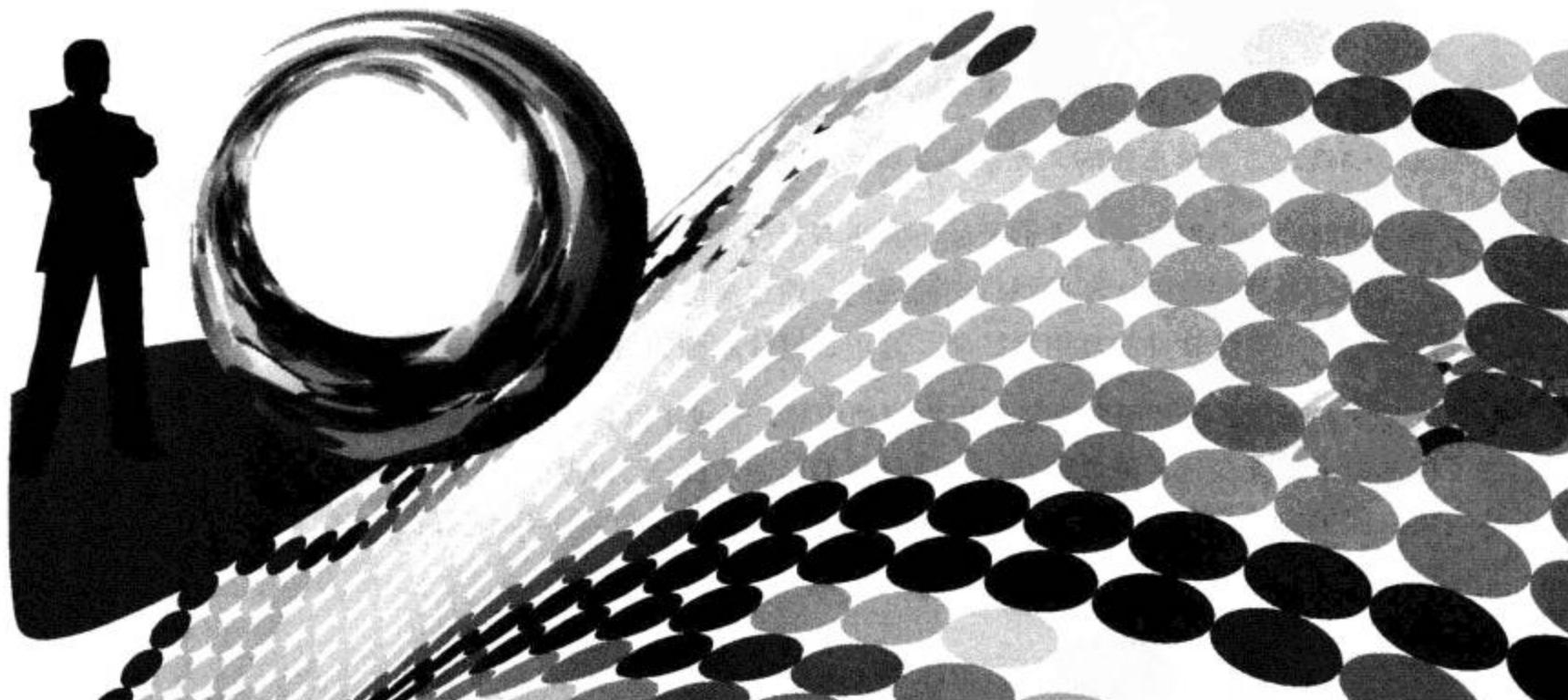
Krzysztof Cwalina, Brad Abrams. Framework Design Guidelines: Conventions, Idioms, and Patterns for Reusable .NET Libraries

虫虫, 从编译的角度看对象

Luca Cardelli & Peter Wegner, On Understanding Types, Data Abstraction, and Polymorphism

第2章 OO 大原则

2.1	OO 原则综述 / 42	2.4	依赖倒置原则 / 53
2.1.1	引言 / 42	2.4.1	引言 / 53
2.1.2	讲述之前 / 42	2.4.2	引经据典 / 53
2.1.3	原则综述 / 43	2.4.3	应用反思 / 54
2.1.4	学习建议 / 44	2.4.4	规则建议 / 56
2.1.5	结论 / 44	2.4.5	结论 / 56
2.2	单一职责原则 / 44	2.5	接口隔离原则 / 56
2.2.1	引言 / 44	2.5.1	引言 / 56
2.2.2	引经据典 / 45	2.5.2	引经据典 / 56
2.2.3	应用反思 / 45	2.5.3	应用反思 / 57
2.2.4	规则建议 / 47	2.5.4	规则建议 / 59
2.2.5	结论 / 48	2.5.5	结论 / 59
2.3	开放封闭原则 / 48	2.6	Liskov 替换原则 / 59
2.3.1	引言 / 48	2.6.1	引言 / 59
2.3.2	引经据典 / 48	2.6.2	引经据典 / 59
2.3.3	应用反思 / 49	2.6.3	应用反思 / 60
2.3.4	规则建议 / 52	2.6.4	规则建议 / 61
2.3.5	结论 / 53	2.6.5	结论 / 62
			参考文献 / 62



2.1 OO 原则综述

本节将介绍以下内容：

- 面向对象基本原则讨论
- 设计原则的历史
- 学习的建议

2.1.1 引言

好的设计，成就好的作品；僵化的设计，则会使你的作品大打折扣。在软件领域更是如此，Bob 大叔在其《敏捷软件开发——原则、模式与实践》一书的序言中就讲到“美的东西比丑的东西创建起来更廉价，也更快捷。”可见追求美好的软件设计不光是代码优雅的问题，更关乎生产成本。对于软件架构的研究经历了很多时间的摸索，从面向过程到面向对象，从设计原则到设计模式，对于如何设计更好软件的探索，从未停止。技术大师总结了很多设计上的经典法则，后来者可以站在巨人的肩膀上尽情享受丰富而珍贵的经验。

在第1章“OO大智慧”中，我们分析和讨论了面向对象技术的基本要素，对继承、封装、多态和接口等进行了基本的讨论，如何将这些基本技术应用的得心应手，是本章要阐释的问题和目标。面向对象的原则也正是如何使用继承、封装、多态和接口技术进行优良设计的原则总结和规律认知，因此这两章的内容相辅相成，互为补充。

2.1.2 讲述之前

在将设计原则和盘推出之前，简单对设计进行一点思考和讨论，这样就能更加明确的了解设计原则的意义和目的。

首先，你的软件系统是否常常有下面的问题或困扰：

僵化。牵一发而动全身，你的系统不可修改或者扩展。

复杂和重复。过分复杂，难于理解，跟踪你的程序注注不知取向，没有设计不可取，过度设计同样不可取；或者系统中充满了重复的结构和实现。

不可复用。过于僵化而不可复用，不能剥离出独立的复用组件。

不够稳定。常常出错而又无法妥善解决问题，系统运行不够可靠。

正是这些问题的出现，促使人们开始不断思考对软件设计、软件架构和软件流程等内容的关注，设计原则也是在这种思考和探索中逐渐归纳总结而成的，通过灵活的应用设计原则，辅助一定的设计模式，封装变化、降低耦合，来实现软件的复用和扩展，这正是设计原则的最终意义。

首先是面向对象。随着面向对象编程思想的成熟，形成了以封装、继承和多态三大要素为主的完整体系，蕴涵了以抽象来封装变化，降低耦合，实现复用的精髓，而这三大因素相互联系、互相制约：封装用以隐藏具体实现，保护内部信息；继承实现复用；而多态改写对象行为，在继承基础上实现更高级别的抽象。在此

基础上逐渐形成的一些基本的OO原则，例如封装变化、对接口编程、少继承多聚合，已经具有了面向对象设计原则的思想，而本节所述的原则可以看成是对这些思想的系统化引申和归纳。

然后是设计模式。模式代表了对经验的总结和提炼，是对重复发生的问题进行的总结和最佳解决策略的探索，自GoF的《设计模式：可复用面向对象软件的基础》这部巨著诞生以来，对于软件设计模式的探索成为每个设计开发者的必修课，而面向对象设计原则可以看做是了解设计模式的基础，为设计模式提供了基本的指导。经典的23个模式背后，都遵循着这些基本原则，而设计原则又由设计模式策略来实现，这就是二者之间的关系，所以了解原则对于认识模式具有绝对的指导意义。

最后，就是设计原则的故事。Robert C. Martin（Bob大叔）的巨著《敏捷软件开发——原则、模式与实践》对敏捷设计原则进行了深刻而生动的论述，不同的模式应对不同的需求，而设计原则则代表永恒的灵魂，需要在实践中时时刻刻地遵守，创造尽可能优雅、灵活的设计。

“你不严格遵守这些原则，违背它们也不会被处以宗教刑罚。但你应当把这些原则看成警铃，若违背了其中的一条，那么警铃就会响起。”。

——ARTHUR J.RIEL，《OOD启示录》

请记住这些技术大师的名字和作品，并研习其中的经验和招式，正是他们让这个领域变得如此光彩夺目，沿着这些智慧的道路走下去，在品读经典的过程中，你会逐渐找到技术神灯之下的奥秘。

2.1.3 原则综述

本章将重点讲述最基本的5个设计原则，分别是：单一职责原则、开放封闭原则、依赖倒置原则、接口隔离原则和Liskov替换原则，在表2-1中列出了主要的一些面向对象设计原则及其简单描述，详细的分析将在本章各节逐步展开。

表2-1 设计原则

设计原则	英文表达	说 明
单一职责原则	SRP，Single Responsibility Principle	一个类，应该仅有一个引起它变化的原因。不要将变化原因不同的职责封装在一起，而应该分离
开放封闭原则	OCP, Open Closed Principle	软件实体应当对修改关闭，对扩展开放
依赖倒置原则	DIP，Dependency Inversion Principle	依赖于抽象，而不要依赖于具体，因为抽象相对稳定
接口隔离原则	ISP，Interface Segregation Principle	尽量应用专门的接口，而不是单一的总接口，接口应该面向用户，将依赖建立在最小的接口上
Liskov替换原则	LSP, Liskov Substitution Principle	子类必须能够替换其基类
合成/聚合复用原则	CARP，Composite/Aggregate Reuse Principle	在新对象中聚合已有对象，使之成为新对象的成员，从而通过操作这些对象达到复用的目的。合成方式较继承方式耦合更松散，所以应该少继承、多聚合
迪米特法则	LoD, Law of Demeter	又叫最少知识原则，指软件实体应该尽可能少的和其他软件实体发生相互作用

在本书中，我们只对前5大原则进行系统论述，关于合成/聚合复用原则，在1.2节“什么是继承”中已经有相关的论述。

2.1.4 学习建议

了解面向对象的设计原则，是个不断实践和研究的过程，对于僵化的代码和设计，应该在可能的情况下有重构的勇气。当然，学习和了解理论基础也是相当重要的方面：

深入了解设计基础。什么是设计基础呢？在我看来，面向对象的基本要素，面向对象的语言基础都是一切设计的基础，没有继承、多态、聚合的深入了解，就无法更好地认识设计原则中的实现理念。

打好设计原则的理论基础。本节简单总结了7种基本的设计原则，对于这些原则的核心思想和应用技巧应该建立基本的认识。

不断实践和反思。对于设计原则的实践，莫过于对僵化设计操刀重构，要有不断完善的勇气和心力，在重构的实践中思考并形成经验。

了解设计模式。设计模式都是对于软件经验的经典总结，模式和原则相辅相成，深入了解常见的设计模式尤为重要，例如Proxy模式就是对单一职责原则的一种体现。

具备必要的修养。除了本章将要讨论的设计原则，还有一些通用的软件规则对于深入理解设计原则大有裨益，例如：高内聚、低耦合、控制器、受保护变化等，对于这些通用规则必须打好基础，在设计原则的很多方面都有这些基础规则的体现。

2.1.5 结论

请从了解设计原则开始，重构你的系统。将僵化而脆弱的设计踢出你的代码，应用面向对象的思想来理清需求，并从中分离抽象和具体，应用前人的经验来实现优雅的设计，构建灵活、可靠和健壮的软件。

本章将以适当的篇幅来关注软件设计原则，以及如何在.NET技术中实现，通过对5个基本设计原则的分析与应用，从中了解和体会如何使设计变得优雅，如何使软件更有扩展性和稳定性，如何避免僵化的设计架构。

2.2 单一职责原则

本节将介绍以下内容：

- 单一职责原则讨论
- 单一职责原则应用
- Proxy模式示例

2.2.1 引言

一个优良的系统设计，强调模块间保持低耦合、高内聚的关系，在面向对象设计中这条规则同样适用，所以

面向对象的第一个设计原则就是：单一职责原则（SRP，Single Responsibility Principle）。

单一职责，强调的是职责的分离，在某种程度上对职责的理解，构成了不同类之间耦合关系的设计关键，因此单一职责原则或多或少成为设计过程中一个必须考虑的基础性原则。

2.2.2 引经据典

关于单一职责原则，其核心的思想是：

一个类，最好只做一件事，只有一个引起它变化的原因。

单一职责原则可以看做是低耦合、高内聚在面向对象原则上的引申，将职责定义为引起变化的原因，以提高内聚性来减少引起变化的原因。职责过多，可能引起它变化的原因就越多，这将导致职责依赖，相互之间就产生影响，从而极大的损伤其内聚性和耦合度。单一职责，通常意味着单一的功能，因此不要为类实现过多的功能点，以保证实体只有一个引起它变化的原因。

因此，SRP 原则的核心就是要求对类的改变只能是一个，对于违反这一原则的类应该进行重构，例如以 Facade 模式或 Proxy 模式分离职责，通过基本的方法 Extract Interface、Extract Class 和 Extract Method 进行梳理。

下面的应用反思中，我们将以具体的示例来进行说明 SRP 的基本内容，以及如何通过代码重构来实现更优良的、职责单一的设计。

2.2.3 应用反思

我们以一个常见的数据库管理系统为例来进行说明，通常情况下根据不同的权限进行数据增删改查的系统比比皆是，然而正是这样一个比较常见的需求场景，却存在着很多充满臭味的设计。一个违背 SRP 原则的设计看起来如图 2-1 所示。

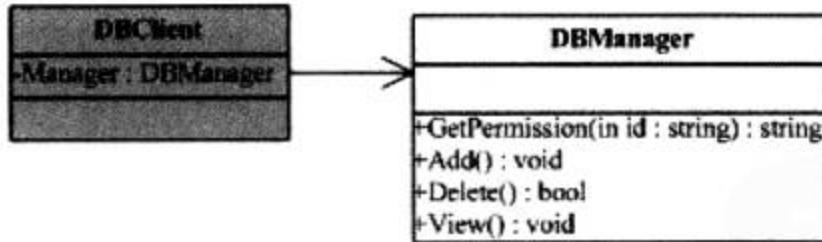


图 2-1 违背 SRP 的设计

在以上设计中，DBMangaer 类将对数据库的操作和用户权限的判别封装在一个类中实现，以添加记录为例，其实现逻辑可以表示为：

```

public void Add()
{
    if (GetPermission(id) == "CanAdd")
    {
        Console.WriteLine("管理员可以增加数据。");
    }
}
  
```

这显然是一个充满了僵化味道的实现，如果权限设置的规则发生改变，那就必须重新修改所有的数据库

操作逻辑，在成千上万行代码中搜索有可能带来隐患的代码，你必须选择将这种代码扫地出门。

在此，权限判断的职责和数据库操作的职责被无理的实现在一个类中，权限的规则变化和数据库操作的规则变化，都有可能引起 DBManager 修改当前代码。按照单一职责原则，一个类应该只有一个引起它改变的原因。所以我们选择以合适的方式来重构有缺陷的设计，在此显然可以通过实现一个 Proxy 模式来解决职责交叉的窘境，修改之后的设计思路如图 2-2 所示。

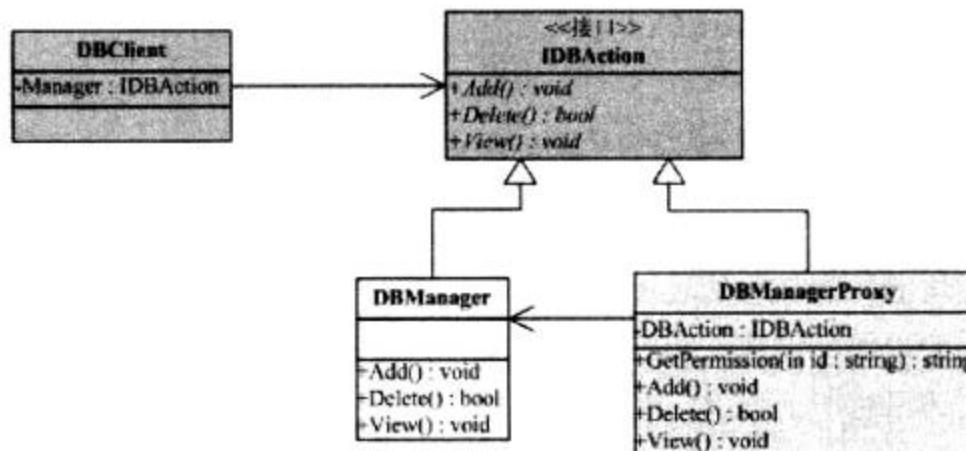


图 2-2 职责分离的设计

以 Proxy 模式调整之后，有效实现了职责的分离，DBManager 类只关注数据操作逻辑，而不用关心权限判断逻辑，在此仅以 Add 操作为例来说明：

```
public class DBManager: IDBAction
{
    public void Add()
    {
        //执行数据增加
    }

    //部分省略....
}
```

而将权限的判断交给 DBManagerProxy 代理类来完成，例如：

```
public class DBManagerProxy : IDBAction
{
    private IDBAction dbManager;

    public DBManagerProxy(IDBAction dbAction)
    {
        dbManager = dbAction;
    }

    //处理权限判断的逻辑
    public string GetPermission(string id)
    {
        //处理权限判断
    }

    public void Add()
    {
        if (GetPermission(id) == "CanAdd")
        {
            dbManager.Add();
        }
    }
}
```

```

    }
    //部分省略...
}

```

通过代理，将数据操作和权限判断两个职责分离，而实际的数据操作由 DBManager 来执行，此时客户端的调用变得非常简单：

```

public class DBClient
{
    public static void Main()
    {
        IDBAction DBManager = new DBManagerProxy(new DBManager("CanAdd"));
        DBManager.Add();
    }
}

```

在该例中，接口 IDBAction 其实体现了设计模式的另一个重要原则：依赖倒置，对此我们在后文有详细论述。在本例中，通过 DBManagerProxy 代理类实现职责分离，DBManager 类将仅有一个变化的原因，那就是数据操作的需求变更；而权限的变更和修改不对 DBManager 造成任何影响，体现了单一职责原则的基本精神。

然而，遵循这条原则的关键，并不是从功能点的多少来划分，而是从引起类变化的原因来把握。当一个变化能导致多个职责同时发生变化的时候，那么这些多个职责应该被封装到一个类，而不是盲目分割，例如 IList 接口定义了集合操作的多个功能点，其定义如下：

```

public interface IList : ICollection, IEnumerable
{
    int Add(object value);
    void Clear();
    bool Contains(object value);
    int IndexOf(object value);
    void Insert(int index, object value);
    void Remove(object value);
    void RemoveAt(int index);
}

```

IList 提供了一组按照索引访问对象的功能，这些职责都是为操作元素而定义的，因此将其不加分割地集中在一起才是较适当的选择，因为避免将相同的职责分散在不同的类中，和避免一个类实现了过多的职责一样重要。衡量的标准不是功能的多少，而是引起变化的因素有什么。

2.2.4 规则建议

关于单一职责原则，我们的建议是：

一个类只有一个引起它变化的原因，否则就应当考虑重构。

SRP 由引起变化的原因决定，而不由功能职责决定。虽然职责常常是引起变化的轴线，但是有时却未必，应该审时度势。

测试驱动开发，有助于实现合理分离功能的设计。

可以通过 Facade 模式或 Proxy 模式进行职责分离。

2.2.5 结论

专注，是一个人的优良品质；单一也是一个类的优良设计。交杂不清的职责将使你的代码看起来别扭，有失美感的设计必然导致丑陋的系统功能，那就不要犹豫向僵化开刀问斩。

2.3 开放封闭原则

本节将介绍以下内容：

- 开放封闭原则讨论
- 开放封闭原则应用

2.3.1 引言

无论如何，开放封闭原则（OCP，Open Closed Principle）都是所有面向对象原则的核心。软件设计本身所追求的目标就是封装变化、降低耦合，而开放封闭原则正是对这一目标的最直接体现。其他的设计原则，很多时候是为实现这一目标服务的，例如以 Liskov 替换原则实现最佳的、正确的继承层次，就能保证不会违反开放封闭原则。

2.3.2 引经据典

关于开放封闭原则，其核心的思想是：

软件实体应该是可扩展，而不可修改的。也就是说，对扩展是开放的，而对修改是封闭的。

因此，开放封闭原则主要体现在两个方面：

对扩展开放，意味着有新的需求或变化时，可以对现有代码进行扩展，以适应新的情况。

对修改封闭，意味着类一旦设计完成，就可以独立完成其工作，而不要对类进行任何修改。

“需求总是变化”、“世界上没有一个软件是不变的”，这些言论是对软件需求最经典的表白。从中透射出一个关键的意思就是，对于软件设计者来说，必须在不需要对原有的系统进行修改的情况下，实现灵活的系统扩展。而如何能做到这一点呢？

只有依赖于抽象。实现开放封闭的核心思想就是对抽象编程，而不对具体编程，因为抽象相对稳定。让类依赖于固定的抽象，所以对修改就是封闭的；而通过面向对象的继承和多态机制，可以实现对抽象体的继承，通过覆写其方法来改变固有行为，实现新的扩展方法，所以对于扩展就是开放的。这是实施开放封闭原则的基本思路，同时这种机制是建立在两个基本的设计原则的基础上，这就是 Liskov 替换原则和合成/聚合复用原则。关于这两个原则，我们在本书的其他部分都有相应的论述，在应用反思部分将有深入的讨论。

对于违反这一原则的类，必须进行重构来改善，常用于实现的设计模式主要有 Template Method 模式和 Strategy 模式。而封装变化，是实现这一原则的重要手段，将经常发生变化的状态封装为一个类。

2.3.3 应用反思

站在银行窗口焦急等待的用户，在长长的队伍面前显得无奈。所以，将这种无奈迁怒到银行的头上是理所当然的，因为银行业务的管理显然有不当之处。银行的业务员面对蜂拥而至的客户需求，在排队等待的人们并非只有一种需求，有人存款、有人转账，也有人申购基金，繁忙的业务员来回在不同的需求中穿梭，手忙脚乱的寻找各种处理单据，电脑系统的功能模块也在不同的需求要求下来回切换，这就是一个发生在银行窗口内外的无奈场景。而我每次面对统一排队的叫号系统时，都为前面长长的等待人群而叫苦，从梳理银行业务员的职责来看，在管理上他们负责的业务过于繁多，将其对应为软件设计来实现，你可以将这种拙劣的设计表示如图 2-3 所示。

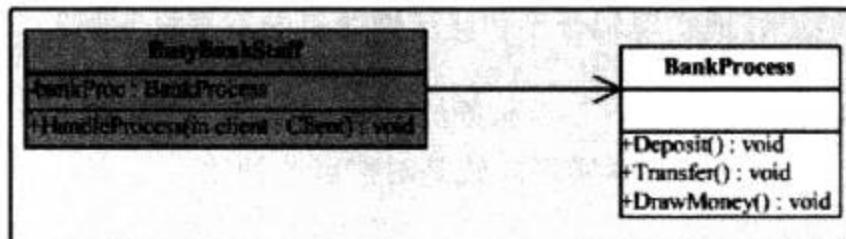


图 2-3 违反 OCP 的设计

按照上述设计的思路，银行业务员要处理的工作，是以这种方式被实现的：

```

class BusyBankStaff
{
    private BankProcess bankProc = new BankProcess();

    // 定义银行员工的业务操作
    public void HandleProcess(Client client)
    {
        switch (client.ClientType)
        {
            case "存款用户":
                bankProc.Deposit();
                break;
            case "转账用户":
                bankProc.Transfer();
                break;
            case "取款用户":
                bankProc.DrawMoney();
                break;
        }
    }
}
  
```

这种设计和实际中的银行业务极其相似，每个 BusyBankStaff（“繁忙的”业务员）接受不同的客户要求，一阵手忙脚乱的选择处理不同的操作流程，就像示例代码中的实现的 Switch 规则，这种被动式的选择造成了大量的时间浪费，而且容易在不同的流程中发生错误。同时，更为严重的是，再有新的业务增加时，你必须修改 BankProcess 中的业务方法，同时修改 Switch 增加新的业务，这种方式显然破坏了原有的格局，以设计原则的术语来说就是：对修改是开放的。

以这种设计来应对不断变化的银行业务，工作人员只能变成 BusyBankStaff 了。分析这种僵化的代码，至少有以下几点值得关注：银行业务封装在一个类中，违反单一职责原则；有新的业务需求发生时，必须通

过修改现有代码来实现，违反了开放封闭原则。

解决上述麻烦的唯一办法是应用开放封闭原则：对扩展开放，对修改封闭。我们回到银行业务上看：为什么这些业务不能做以适当的调整呢？每个业务员不必周旋在各种业务选项中，将存款、取款、转账、外汇等不同的业务分窗口进行，每个业务员快乐地专注于一件或几件相关业务，就会轻松许多。综合应用单一职责原则来梳理银行业务处理流程，将职责进行有效的分离；而这样仍然没有解决业务自动处理的问题，你还是可以闻到僵化的坏味道在系统中弥漫。

应用开发封闭原则，可以给我们更多的收获，首先将银行系统中最可能扩展的部分隔离出来，形成统一的接口处理，在银行系统中最可能扩展的因素就是业务功能的增加或变更。对于业务流程应该将其作为可扩展的部分来实现，当有新的功能增加时，不需重新梳理已经形成的业务接口，然后对整个系统要进行大的处理动作，那么怎么才能更好地实现耦合度和灵活性兼有的双重机制呢？

答案就是抽象。将业务功能抽象为接口，当业务员依赖于固定的抽象时，对于修改就是封闭的；而通过继承和多态机制，从抽象体派生出新的扩展实现，就是对扩展的开放。

依据开放封闭原则，进行重构，新的设计思路如图 2-4 所示。

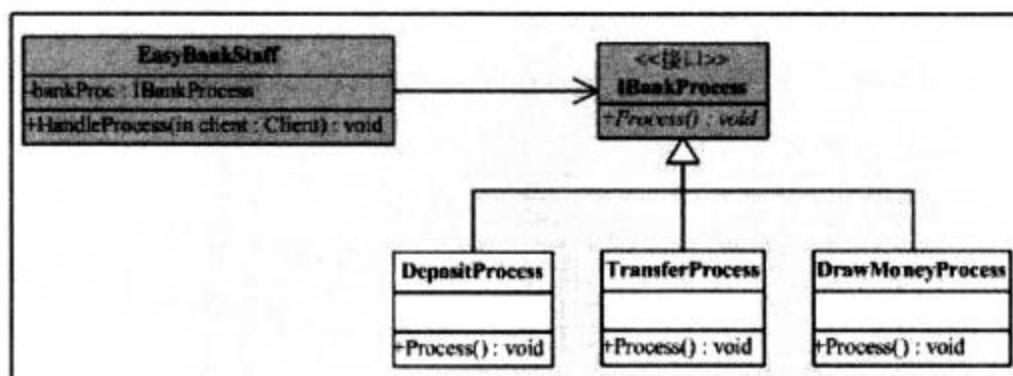


图 2-4 面向抽象的设计

按照上述设计实现，用细节表示为：

```

interface IBankProcess
{
    void Process();
}
  
```

然后在隔离的接口上，对功能进行扩展，例如改造单一职责的示例将有如下的实现：

```

//按银行按业务进行分类
class DepositProcess : IBankProcess
{
    public void Process()
    {
        //办理存款业务
    }
}

class TransferProcess : IBankProcess
{
    public void Process()
    {
        //办理转账业务
    }
}
  
```

```

}

class DrawMoneyProcess : IBankProcess
{
    public void Process()
    {
        //办理取款业务
    }
}

```

这种思路的转换，会让复杂的问题变得简单明了，使系统各负其责，人人实惠。有了上述的重构，银行工作人员彻底变成一个 EasyBankStaff（“轻松”的组织者）：

```

class EasyBankStaff
{
    private IBankProcess bankProc = null;

    public void HandleProcess(Client client)
    {
        //业务处理
        bankProc = client.CreateProcess();
        bankProc.Process();
    }
}

```

银行业务可以像这样被自动地实现了：

```

class BankProcess
{
    public static void Main()
    {
        EasyBankStaff bankStaff = new EasyBankStaff();
        bankStaff.HandleProcess(new Client("转账用户"));
    }
}

```

你看，现在一切都变得轻松自在，匆忙中办理业务的人们不会在长长的队伍面前一筹莫展，而业务员也从烦琐复杂的劳动中解脱出来。当有新的业务增加时，银行经理不必为重新组织业务流程而担忧，你只需为新增的业务实现 IBankProcess 接口，系统的其他部分将丝毫不受影响，办理新业务的客户会很容易找到受理新增业务的窗口，如图 2-5 所示。

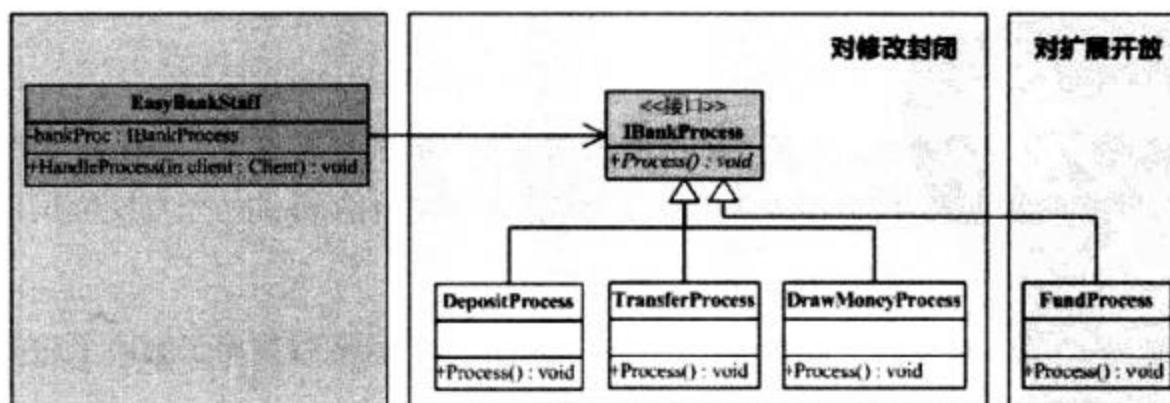


图 2-5 符合 OCP 的设计

对应的实现为：

```

class FundProcess : IBankProcess
{
    public void Process()

```

```
{
    //办理基金业务
}
}
```

可见，新的设计遵守了开放封闭原则，在需求增加时只需要向系统中加入新的功能实现类，而原有的一切保持封闭不变的状态，这就是基于抽象机制而实现的开放封闭式设计。

然而，细心观察上述实现你会发现一个非常致命的问题：人们是如何找到其想要处理的业务窗口，难道银行还得需要一部分人力来进行疏导？然而确实如此，至少当前的设计必须如此，所以上述实现并非真正的业务处理面貌，实际上当前“轻松”的银行业务员，还并非真正的“轻松”，我们忽略了这个业务关系中最重要的一个部分，就是用户。当前，用户的定义被实现为：

```
class Client
{
    private string ClientType;

    public Client(string clientType)
    {
        ClientType = clientType;
    }

    public IBankProcess CreateProcess()
    {
        //实际的处理
        switch (ClientType)
        {
            case "存款用户":
                return new DepositProcess();
                break;
            case "转账用户":
                return new TransferProcess();
                break;
            case "取款用户":
                return new DrawMoneyProcess();
                break;
        }
        return null;
    }
}
```

如果出现新增加的业务，你还必须在长长的分支语句中加入新的处理选项，switch 的坏味道依然让每个人看起来都倒胃口，银行业务还是以牺牲客户的选择为代价，难道不能提供一个自发组织客户寻找业务窗口的机制吗？

我们把答案放在下一节 2.4 节“依赖倒置原则”，其中的设计原则就是用于解决上述问题的。我们对于银行业务的讨论，还会继续进行。

2.3.4 规则建议

开放封闭原则，是最为重要的设计原则，Liskov 替换原则和合成/聚合复用原则为开放封闭原则

的实现提供保证。

可以通过 Template Method 模式和 Strategy 模式进行重构，实现对修改封闭、对扩展开放的设计思路。

封装变化，是实现开放封闭原则的重要手段，对于经常发生变化的状态一般将其封装为一个抽象，例如银行业务中的 IBankProcess 接口。

拒绝滥用抽象，只将经常变化的部分进行抽象，这种经验可以从设计模式的学习与应用中获得。

2.3.5 结论

开放封闭原则，作为名副其实的核心原则，应该备受关注。通过开放封闭原则，可以有效地降低实体与实体之间的耦合性；将容易变化的因素进行抽象处理，可以改善类的内聚性。

2.4 依赖倒置原则

本节将介绍以下内容：

- 依赖倒置原则讨论
- 依赖倒置原则应用

2.4.1 引言

著名的好莱坞法则：不要调用我们，我们会调用你，是对依赖倒置原则（DIP, Dependency Inversion Principle）最形象的诠释，通过抽象机制有效解决类层次之间的关系，降低耦合的粒度，实现对抽象的依赖是依赖倒置原则的核心思想，而如何将这种思想灵活的运用在系统设计的实践中，是一个值得关注和研究的话题。

本节继续对银行系统设计进行讨论，在开放封闭原则实现的设计基础上，重新梳理层次之间的依赖关系，实现一个更加柔性、灵活和可扩展的系统。

2.4.2 引经据典

关于依赖倒置原则，其核心的思想是：

依赖于抽象。

具体而言，依赖倒置体现在：

高层模块不应该依赖于底层模块，二者都应该依赖于抽象。

抽象不应该依赖于具体，具体应该依赖于抽象。

依赖，一定会存在于类与类、模块与模块之间；依赖关系，也一定是系统设计必须关注的要点。面向对象设计在某种层次上，就是一个关于关系处理的哲学，而依赖倒置原则正是这种哲学思想在具体应用中的体现。当两个模块之间存在紧耦合的关系时，最好的办法就是分离接口和实现：在依赖之间定义一个抽象的接口，使得高层模块调用接口的方法，而低层模块实现接口的定义，以此来有效控制耦合关系，达到依赖于抽象的设计目标。

2.4.3 应用反思

在银行业务处理的示例中，我们很好地处理了业务员和业务之间的关系，建立在抽象的基础上，利用多态特性实现了对修改封闭对扩展开放的设计原则，而在示例的最后我们发现银行业务员和银行客户之间，仍然存在依赖问题，并将问题抛向给依赖倒置原则来解决。在此，一个关键的问题仍然是：客户如何自动找到对应的窗口来解决自己的业务需求，本节将继续这一话题的讨论。

现在，认真的梳理银行系统中的几个关系：业务员 EasyBankStaff、业务 IBankProcess 和客户 Client 之间，明显违反了依赖倒置原则，业务员和业务类依赖于具体的客户，而非抽象，如图 2-6 所示。

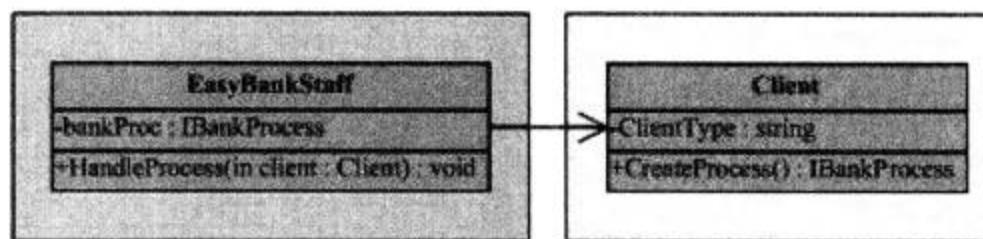


图 2-6 违反 DIP 的设计

CreateProcess 在创建业务类别时，是必须依托于 ClientType 为判断条件的，从而也决定了 HandleProcess 的执行也受制于 ClientType 的条件，我们从 HandleProcess 的处理过程可以了解这一点：

```

public static void Main()
{
    EasyBankStaff bankStaff = new EasyBankStaff();
    bankStaff.HandleProcess(new Client("转账用户"));
}

```

bankStaff 处理 HandleProcess 的过程依赖于具体的 Client 客户，而当有新的业务类型增加时，你必须在系统中依次增加对客户类别的依赖，对于完美的设计来说，这种机制是僵化的，应该实现更好的解决方案。

你必须找出潜在的抽象，使 EasyBankStaff 依赖于此抽象，而抽象的办法就是为 EasyBankStaff 和 Client 之间增加一个抽象接口，同时为了消除客户类别的影响，在此按照依赖倒置的原则，设计新的方案如图 2-7 所示。

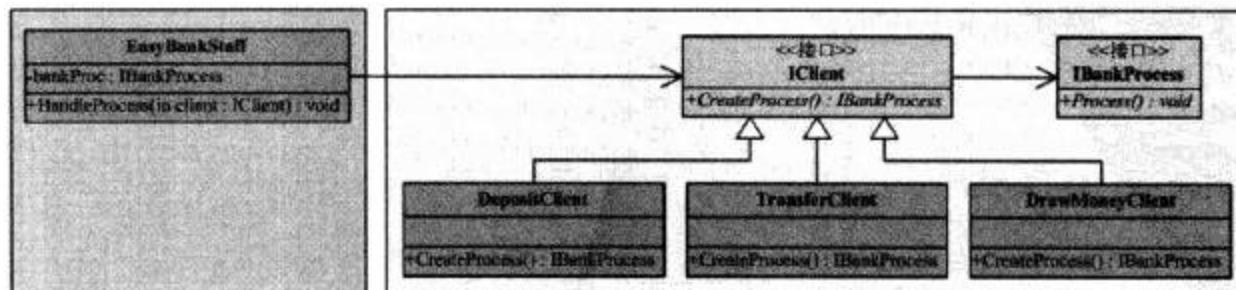


图 2-7 依赖于抽象的设计

新的方案中，通过倒置对 Client 的依赖关系，EasyBankStaff 将依赖于 IClient 接口，而具体的实现由 DepositClient、TransferClient 和 DrawMoneyClient 来实现。具体的实现为：

```
interface IClient
{
    IBankProcess CreateProcess();
}

class DepositClient : IClient
{
    IBankProcess IClient.CreateProcess()
    {
        return new DepositProcess();
    }
}

class TransferClient : IClient
{
    IBankProcess IClient.CreateProcess()
    {
        return new TransferProcess();
    }
}

class DrawMoneyClient : IClient
{
    IBankProcess IClient.CreateProcess()
    {
        return new DrawMoneyProcess();
    }
}
```

在客户端调用时，不需要进行任何类别的判断，终于可以实现用户自动找到窗口的需求了：

```
class BankProcess
{
    public static void Main()
    {
        EasyBankStaff bankStaff = new EasyBankStaff();
        bankStaff.HandleProcess(new TransferClient());
    }
}
```

还记得那个长长的 switch 语句吗？在此已经不复存在了，用户不需要自己做出选择，只需要向银行表明身份，HandleProcess 就可以自行受理其业务。通过依赖于抽象，实现了 Client 对象的依赖倒置，Client 不再被依赖，可以实现更多的灵活性。当有新的业务类别增加时，你只需要实现 IClient 接口即可，例如：

```
class FundClient : IClient
{
    IBankProcess IClient.CreateProcess()
    {
        return new FundProcess();
    }
}
```

大功告成，而这种改变不会影响到原有系统的任何部分，也就能够保证遵守开放封闭原则。由此可以顺便得出一个结论，依赖倒置也是实现开放封闭原则的基础，为有效解决层次之间的依赖关系寻求了最佳方案。

2.4.4 规则建议

抽象的稳定性决定了系统的稳定性，因为抽象是保持不变的，依赖于抽象是面向对象设计的精髓，也是依赖倒置原则的核心思想。

依赖于抽象是一个通用的规则，而某些时候依赖于细节则是在所难免的，必须权衡在抽象和具体之间的取舍，方法不是一成不变的。

依赖于抽象，就是要对接口编程，不要对实现编程。

2.4.5 结论

通过开放封闭原则和依赖倒置原则的综合论述，基本完成了一个银行业务系统的设计雏形，从这个过程中能够逐渐体会到设计原则在软件设计中举足轻重的作用。对于设计而言，这些原则就是纲领性的标准，你必须选择灵活遵守，原则是不变的，而方法是灵活的，在不同的应用中协调和权衡，是面向对象设计的精髓，而这些智慧在逐步的实践和探索中形成。

2.5 接口隔离原则

本节将介绍以下内容：

- 接口隔离原则讨论
- 接口隔离原则应用

2.5.1 引言

现实中有这样一种情况，火车站窗口总是挤满了排队的人，而排队的人并非都是为买票而来，有些人可能为了查询一个列车信息，有些人可能为了退票，但是都必须按照车站的规则排在一条长长的队伍中。你能很容易地想到将不同的人分流到不同的窗口来提高效率，因为这显然是一种时间和人力的浪费。

同样的情况，存在于软件设计中，“胖”接口会强制所有继承的类型实现其所有的方法，而有些方法对客户来说是无用的，这种情况对接口来说就是一个“浪费”，而接口隔离原则（ISP，Interface Segregation Principle）正是应对于这种情况的设计标准。

2.5.2 引经据典

关于接口隔离原则，其核心的思想是：

使用多个小的专门的接口，而不要使用一个大的总接口。

具体而言，接口隔离体现在：

接口应该是内聚的，应该避免出现“胖”接口。

一个类对另一个类的依赖应该建立在最小的接口上，不要强迫依赖不用的方法，这是一种接口污染。

接口有效地将细节和抽象隔离，体现了对抽象编程的一切好处，接口隔离原则强调接口的单一性。而胖接口存在明显的弊端，会导致实现的类型必须完全实现接口所有的方法、属性等；而某些时候，实现类型并非需要所有的接口定义，在设计上这是一种“浪费”，而且在实施上这会带来潜在的问题，对胖接口的修改将导致一连串的客户端程序需要修改，有时候这是一种灾难。在这种情况下，将胖接口分解为多个特定的定制化方法，使得客户端仅仅依赖于它们实际调用的方法，从而解除了客户端不会依赖于它们不用的方法。因此，按照客户需求将客户分组，并依据这种分组来分离接口，是接口隔离原则的重要方法。分离的手段主要有以下两种：

委托分离，通过增加一个新的类型来委托客户的请求，隔离客户和接口的直接依赖，但是会增加系统开销。

多重继承分离，通过接口多继承来实现客户需求，这种方式值得推荐。

另外，对于接口的理解并不完全局限于程序语言上定义的接口概念，还代表了逻辑上的接口概念。程序语言上的接口，就像C#语言中的interface定义的数据类型结构，接口隔离要求interface仅提供客户需要的特征；而逻辑上的接口，代表了方法特征的集合，也可能是一个新的类型。不管怎样，为每个客户实现特定的接口，是接口隔离原则的统一思想。

2.5.3 应用反思

对于接口隔离，一个容易想到的场景是人们对电脑的应用是不同的，孩子用电脑学习，大人用电脑工作，而老人用电脑娱乐，这种场景可以被描述为图2-8。

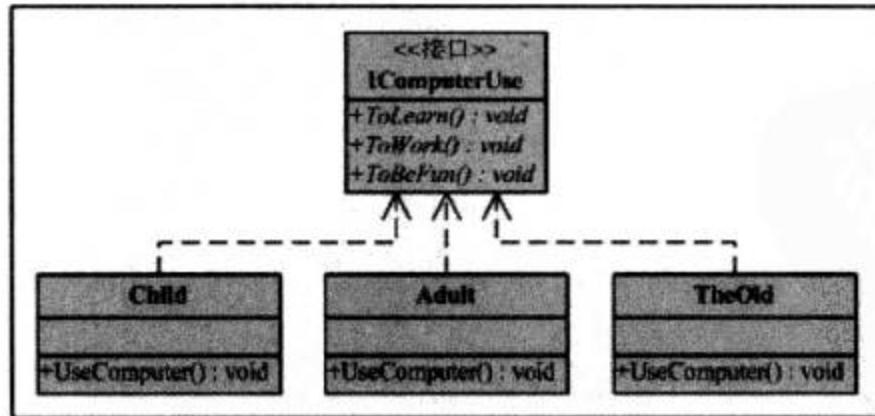


图2-8 “胖”接口设计

这种设计明显违反了接口隔离原则，IComputerUse是一个典型的“胖”接口，对于Adult来说，他可能既需要工作（ToWork），还娱乐（ToBeFun）；而对于Child来说，他只需要学习就行了，工作对他来说就是“浪费”，同样的情况也存在于TheOld。尤其是，当IComputerUse增加了新的方法时，依赖于它的Child、Adult和TheOld都必须被重新编译。按照接口隔离原则，必须按照客户的需求进行分组，将“胖”接口分解

为特定的多个单一接口，客户按照各自的需求实现不同的接口，不再被强迫依赖于它们不用的方法。不必要的耦合被消灭了，这种设计应该如图 2-9 所示。

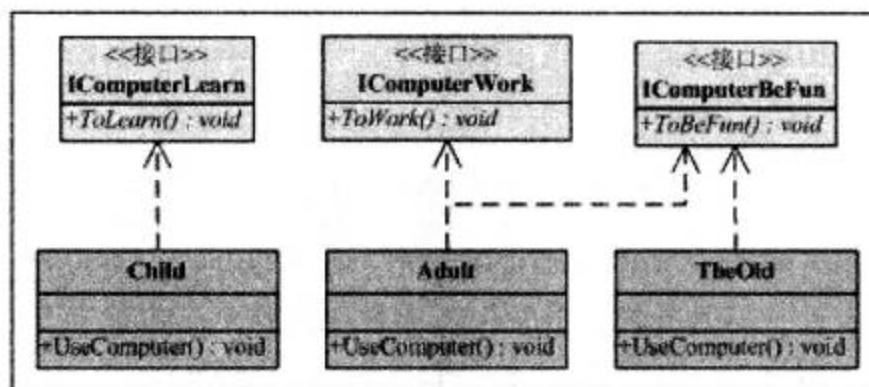


图 2-9 基于接口隔离的设计

具体的实现为：

```
interface IComputerLearn
{
    void ToLearn();
}

interface IComputerWork
{
    void ToWork();
}

interface IComputerBeFun
{
    void ToBeFun();
}

class Adult
{
    private IComputerWork myWork;
    private IComputerBeFun myFun;

    public void UseComputer()
    {
        //主要是工作
        myWork.ToWork();
        //还可以娱乐
        myFun.ToBeFun();
    }
}

class Child
{
    private IComputerLearn myLearn;

    public void UseComputer()
    {
        //只有学习，不会依赖其他的方法
        myLearn.ToLearn();
    }
}
```

当有新的操作需要增加时，仅仅改变发生更改的部分，而不影响到其他的客户需求，不需要对依赖的部分全部重新编译，这正是接口隔离的最根本的目的和魅力体现。

2.5.4 规则建议

将功能相近的接口合并，可能造成接口污染，实现内聚的接口才是接口设计的基本原则。

接口隔离原则，能够保证系统扩展和修改的影响不会扩展到系统的其他部分，一定程度上保证了对开放封闭原则的遵守。

2.5.5 结论

接口隔离原则是一个核心思想并不复杂的设计原则，在实际的系统设计中，接口隔离原则是实现组件化设计的重要基础，在保证类的移植性和可用性方面有着重要的意义。同时，接口隔离原则，也影响着对系统扩展时的隔离和内聚，是保证开放封闭的另一个重要的条件。

2.6 Liskov 替换原则

本节将介绍以下内容：

- Liskov 替换原则讨论
- Liskov 替换原则应用

2.6.1 引言

Liskov 替换原则（LSP，Liskov Substitution Principle）是关于继承机制的应用原则，是实现开放封闭原则的具体性规范，违反了 Liskov 原则必然意味着违反了开放封闭原则。因此，有必要对面向对象的继承机制及其基本原则做以探索，来进一步了解面向对象中实现抽象、多态技术的基础：继承及其规范。

2.6.2 引经据典

关于 Liskov 替换原则，其核心的思想是：

子类必须能够替换其基类。

这一思想体现为对继承机制的约束规范，只有子类能够替换其基类时，才能保证系统在运行期内识别子类，这是保证继承复用的基础。在父类和子类的具体实现中，必须严格把握继承层次中的关系和特征，将基类替换为子类，程序的行为不会发生任何变化。同时，这一约束反过来则是不成立的，子类可以替换基类，但是基类不一定能替换子类。

在面向对象设计中，子类继承于父类，其关系是以 IS-A 确定的，在.NET 语言中以 is 关键字类判断两个对象的类型是否兼容，例如：

```
SonClass son = new SonClass();
FatherClass father = son is FatherClass ? (FatherClass)son : null;
father.Method();
```

上述判断反过来则是不成立的。而 Liskov 替换原则对 IS-A 的要求并不仅仅是类型兼容的问题，它强调从客户角度来看 FatherClass 和 SonClass 在行为上必须是相容的。FatherClass 的 Method 方法和 SonClass 的 Method 方法，必须是按照客户理解需求兼容的，否则就不能说 SonClass IS-A FatherClass，也就意味着是违反 Liskov 替换原则的。

Liskov 替换原则，主要着眼于对抽象和多态建立在继承的基础上，因此只有遵守了 Liskov 替换原则，才能保证继承复用是可靠的。实现的方法是面向接口编程：将公共部分抽象为基类接口或抽象类，通过 Extract Abstract Class，在子类中通过覆写父类方法实现以新的方式支持同样的职责。

2.6.3 应用反思

在继承关系中，子类对父类的继承除了字段、属性，还有方法；而使同一方法在子类中表现为不同的行为是通过多态来实现的，具体在语言操作上表现为父类提供虚函数，而在子类中覆写该虚函数，这是抽象机制的重要基础。子类可以代替父类实现下面的情况：

```
public static void DoSomething(FatherClass f)
{
    f.Method();
}
```

如果 Method 被实现为虚函数，并且在子类被覆写，则传入 DoSomething 的实参既可以是父类，也可以是子类，子类完全可以替代父类在此调用自己的方法，这正是 Livskov 替换原则强调的继承关系。具体的实现为：

```
class FatherClass
{
    //父类的行为
    public virtual void Method()
    {
    }
}

class SonClass : FatherClass
{
    //子类重写父类的行为
    public override void Method()
    {
    }
}
```

对于继承的分析，详见 1.2 节“什么是继承”；对于多态的理解，详见 1.4 节“多态的艺术”。

明显违反 LSP 原则的情况是，Method 方法不为虚函数，而且在子类中存在父类不具有的方法，导致 DoSomething 中就不能以子类代替父类，方法调用必须进行显式的类型转换，这是对 LSP 的明显违反，例如：

```
class FatherClass
{
    public string Type;
    //父类方法
```

```

public void Method()
{
}

public FatherClass()
{
    Type = "FatherClass";
}
}

class SonClass: FatherClass
{
    //子类存在父类不具有的方法
    public void SonMethod()
    {
    }

    public SonClass()
    {
        Type = "SonClass";
    }
}

```

上述实现，带来的直接问题就是父类中不存在子类的 SonMethod 方法，导致子类不能替换父类，对系统的扩展性来说这是一种僵化的设计。当有新的子类被添加时，必须同时修改 DoSomething 的内部判别，例如：

```

public static void DoSomething(FatherClass f)
{
    switch (f.Type)
    {
        case "FatherClass":
            f.Method();
            break;
        case "SonClass":
            ((SonClass)f).SonMethod();
            break;
        case "GrandsonClass":
            ((GrandsonClass)f).GrandsonMethod();
            break;
    }
}

```

对于这种情况，Bob 大叔在《敏捷软件开发：原则、模式与实践》明确指出：对于 LSP 的违反常常会导致使用运行时类型识别，通过一个显式分支语句 if 或者 switch 确定一个对象的类型，以便根据其类型选择合适的方法调用。

这也意味着违反了 Liskov 替换原则就必然违反开放封闭原则，因此这种明显的违规应该被拒绝。开放封闭原则的关键是抽象，体现在基类和子类之间的抽象就是继承，因此 LSP 原则是实现抽象机制的实施规范，而.NET 的虚函数技术能很好解决问题，就像开始的示例一样。

2.6.4 规则建议

Liskov 替换原则是关于继承机制的设计原则，违反了 Liskov 替换原则就必然导致违反开放封闭原则。

Liskov 替换原则能够保证系统具有良好的扩展性，同时实现基于多态的抽象机制，能够减少代

码冗余，避免运行期的类型判别。

子类必须满足基类和客户端对其的行为约定，客户端对行为的期望在基类和子类必须保持一致。

IS-A 是基于行为方式的，它依赖于客户端的调用方式，对象的行为方式才是值得关注的要素。

子类的异常必须控制在父类可以预计的范围，否则将导致替换违规，违反了 Liskov 替换原则。

2.6.5 结论

因此，应该面向接口编程，要求父类尽可能使用接口或抽象类来实现，能够保证不违反 Liskov 替换原则。然而，除了明目张胆的违反 LSP 原则，有些时候对于 LSP 的违反是潜在的，从继承层次上并不能了解其问题。必须从客户的角度理解，才能发现子类的行为和客户对父类的期望是不是一致。因此，按照客户程序的预期来保证子类和父类在行为上的相容，是 Liskov 替换原则的另一关键。

参考文献

Robert C. Martin, 敏捷软件开发：原则、模式与实践

Eric Freeman, Elisabeth Freeman. Head First Design Patterns

Erich Gamma, Richard Helm, Ralph Johnson, John Vlisside, 设计模式：可复用面向对象软件的基础

Martin Fowler, Refactoring: Improving the Design of Existing Code

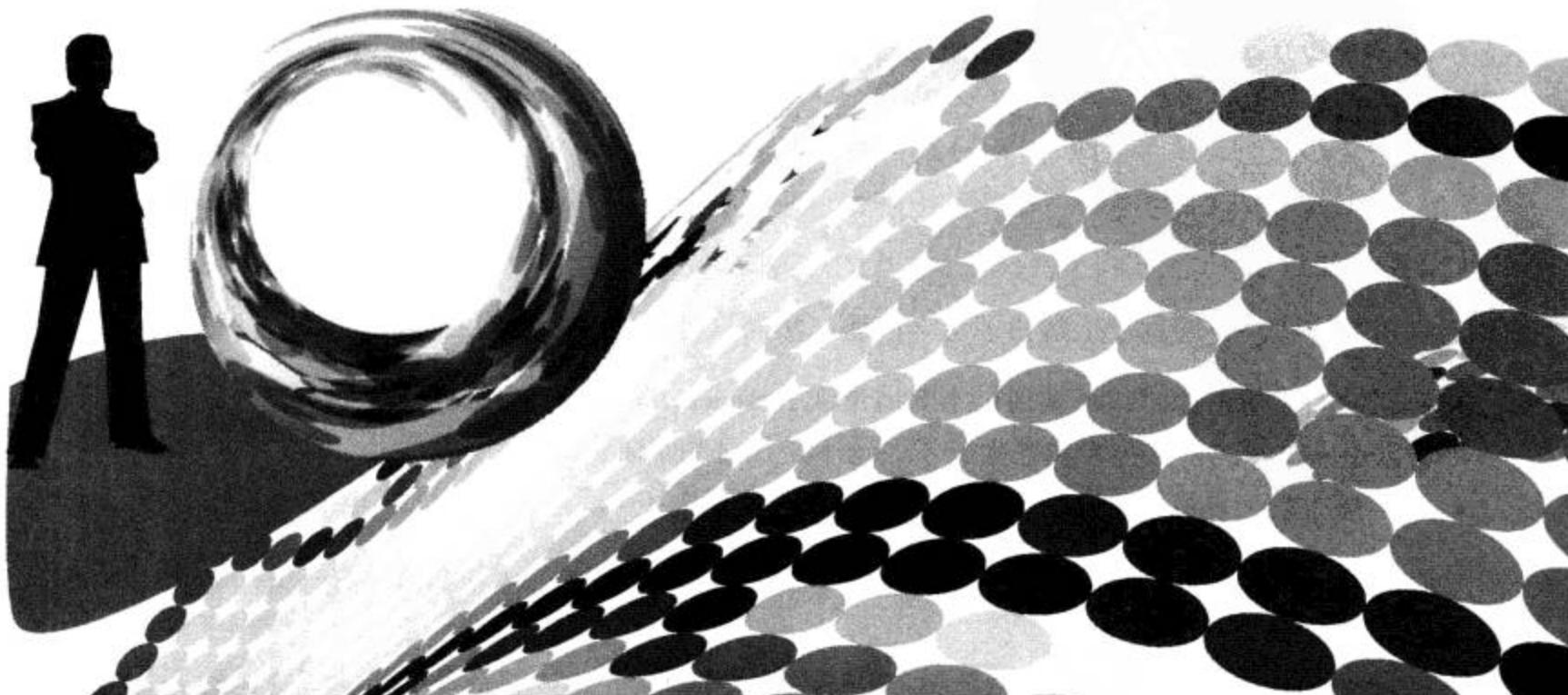
Martin Fowler, Patterns of Enterprise Application Architecture

王咏武, 王咏刚, 道法自然——面向对象实践指南, 电子工业出版社, 2004.10

ARTHUR J.RIEL, OOD 启思录, 人民邮电出版社

第3章 OO之美

3.1 设计的分寸 / 64	3.4 面向对象和基于对象 / 91
3.1.1 引言 / 64	3.4.1 引言 / 91
3.1.2 设计由何而来 / 64	3.4.2 基于对象 / 91
3.1.3 从此重构 / 65	3.4.3 二者的差别 / 91
3.1.4 结论 / 67	3.4.4 结论 / 92
3.2 依赖的哲学 / 67	3.5 也谈.NET 闭包 / 92
3.2.1 引言 / 67	3.5.1 引言 / 92
3.2.2 什么是依赖，什么是抽象 / 68	3.5.2 什么是闭包 / 92
3.2.3 重新回到依赖倒置 / 73	3.5.3 .NET 也有闭包 / 93
3.2.4 解构控制反转 (IoC) 和依赖注入 (DI) / 79	3.5.4 福利与问题 / 95
3.2.5 典型的设计模式 / 82	3.5.5 结论 / 96
3.2.6 基于契约编程：SOA 架构下的依赖 / 83	3.6 好代码和坏代码 / 96
3.2.7 对象创建的依赖 / 84	3.6.1 引言 / 96
3.2.8 不规则总结 / 87	3.6.2 好代码、坏代码 / 97
3.2.9 结论 / 87	3.6.3 结论 / 105
3.3 模式的起点 / 87	参考文献 / 105
3.3.1 引言 / 87	
3.3.2 模式的起点 / 88	
3.3.3 模式的建议 / 90	
3.3.4 结论 / 91	



3.1 设计的分寸

本节将介绍以下内容：

- 设计的由来
- 浅谈重构

3.1.1 引言

有了前面两章“OO大智慧”和“OO大原则”的铺垫，相信读者已经有了对面向对象的基本认知。而本章将继续深入关于面向对象和设计问题的讨论，挑起设计与架构的话题。在高级语言横行的今天，对于静态语言的设计都源于面向对象思想，重构与设计都基于这些简单的标准。

然而，对于设计，还有很多看似“惯常”的法则与经验广泛存在于软件系统中，例如除了经典的23种设计模式，还有很多模式之外的模式，按照粒度的大小、系统的特点、规模的大小，而形成的架构规则。

话说

对设计来说，或许永远没有唯一的答案，你只能无限地接近最好。

设计，没有唯一的答案，但是把握分寸，却是软件设计中需要“用心良苦”的部分。

3.1.2 设计由何而来

设计，从何而来？是需求。是重构。

设计原则是系统设计的灵魂，而设计模式是系统开发的模板，灵活自如的应用才是设计以不变应万变的准则。例如，实现一个用户注册的方法，首先会想到：

```
//初次设计
public void Register(string name, Int32 age)
{
}
```

在一定的需求条件下，这个方法已经能够经受系统的考验，安全而平稳地向数据库中不断插入新的用户信息。然而，当需求发生变化时，你可能不得不对此做出调整，而我们就将这种调整称为**重构**。但是重构远不是扩充，而是设计。例如，现在的注册项发生了变化，还需要同时注册性别、电话，没有设计的调整，就将被实现为：

```
//需求变更
public void Register(string name, Int32 age, bool isMale, Int32 phone)
{
}
```

通过重载方式，一定程度上解决了这一问题，然而这种不能称为重构的调整，至少存在以下的弊端：

- 有新增的注册信息时，还要通过不断地重载Register方法来实现更多信息的扩展。
- 方法Register的参数列表实在太长了，这不是优雅的代码实现。
- 需要修改系统中相关的方法调用来适应新的重载方法。

僵化的调整失去了设计的灵活性，没有思考的程序只能使程序的扩展和维护变得不可收拾，其实对于上述问题，只需要进行简单的重构，就可轻松避免上述 3 个弊端，实现更加柔性的系统。例如，简单重构如下：

```
public class UserInfo
{
    public string Name { get; set; }
    public Int32 Age { get; set; }
    public bool Gender { get; set; }
}
```

通过将用户信息封装为一个类，实现更加简单的参数列表，同时其带来的好处还远不止避免了上述 3 个缺陷，而且能带来对用户信息的封装，实现更可靠的信息隐藏和暴露：

- 可以通过字段和属性封装，实现对于成员的只读、可读可写权限的控制。.NET 3.0 的自动属性为属性封装实现了更为优雅的语法游戏，这些特性让 C# 成为更具有吸引力的高级语言（详见 13.2 节“赏析 C# 3.0”）：

```
//定义可读可写属性  
    public string Mobile { get; set; }  
  
    //定义只读属性  
    public string Password { get; private set; }
```

- 实现一定的逻辑封装，例如对于电子邮件，可以检查其合法性；

```
private string email;

public string Email
{
    get { return email; }
    set
    {
        string strReg = @"^([\w-\.]+@[([0-9]{1,3}\.[0-9]{1,3}\.[0-9]{1,3}\.)|(([\\w-]+\.)+)([a-zA-Z]{2,4}|[0-9]{1,3})\]?)$";
        if (Regex.IsMatch(value, strReg))
        {
            email = value;
        }
        else
        {
            throw new InvalidCastException("Invalid Email Address.");
        }
    }
}
```

那么，设计是如何实现和建立的呢？答案就是面向对象。正如上述演化过程一样，其中应用了面向对象中的封装要素，完成了更加柔性的设计。在 1.3 节“封装的秘密”中，我们就对封装展开了详细的探讨，基于实例的应用和对.NET 实现本质的分析，能够更加强化对于面向对象基本要素的理解。

这些面向对象的思想和应用，来自于实践，完善于重构。

3.1.3 从此重构

设计是如此重要，那么开发者的基本设计能力与素质又从何下手来培养呢？

最好的办法，就是请个老师。从框架中了解，从系统中实现，从书文中汲取。然而，设计能力的提升绝非一朝一夕之功，软件开发中的设计大师，往往必须具备多年的修行方可称之为“架构师”。

一个在简历中轻描淡写的“10年软件设计经验”，并非是所有软件人都能修炼成的真功夫，这里没有任何虚情假意可言。在一个项目的实现过程中，逐渐了解什么是对象、什么是对抽象编程、设计模式是如何应用在实际的系统架构、设计原则到底是什么秘密武器，而重要的是完成一个软件项目，对于更多人来说是认识一种软件开发的科学流程。这种体验，才是难能可贵的经验。在设计的广义概念里，几个必需的概念是应该首先被了解和认知的，以排名不分先后的原则罗列，它们大概包括：

- **面向对象 (Object-Oriented)**，关于面向对象没有必要重复嚼舌了，本书的第1章“OO大智慧”中对.NET的面向对象进行了有别于其他专著的介绍，除了以实例突出面向对象之思想大成，还以浓墨铺陈了.NET是如何在底层技术上来实现继承、多态和接口映射等机制，从而使读者可以更加有效和深刻地把握面向对象之精髓。
- **面向服务 (Service Oriented)**，SO至少是个时髦的话题，WCF伴着.NET 3.5的发布，一个一统江湖的面向服务的基础架构横空出世。可以想象的是，未来的分布式系统架构将变得更加柔性和统一，简单而有效。
- **框架 (Framework)**，所谓框架就是应用系统构建所需的基础设施。应用程序是变化万千的，而基础架构则是相对稳定的。这正如我们基于.NET Framework来实现数以万计的.NET应用：Windows Form Application、Web Form Application、XML Web Service等，都体现了框架和应用的关系。.NET Framework本身就是一个基础性架构，正如经典的MFC一样，提供了.NET应用赖以生存的基础性支持。对于框架的探索和实践，应该是每个有意提高设计能力的开发者的必经之路。选择一个或几个典型的框架进行梳理与实践，才能有效地了解软件世界的庞大体系是如何和谐统一地运作。从经典框架设计中，寻找灵感，锻炼体验，例如分享经典案例Petshop三层架构的实践体验，通过真正的需求、设计、开发和测试，而在旅途中我们就能完成从概念到实践、从表面到思想的体验。
- **设计原则 (Design Principles)**，是面向对象设计程序开发的思想大成，了解面向对象，深入设计模式，则有必要深入设计原则之精髓。可以不了解全部的设计模式，但必须深入每一个设计原则。一个是内功，一个是招式，设计的第一项修炼就从内功开始。本书第2章“OO大原则”对5大设计原则进行了一些以例说理式的实际探讨和分析，几个看似简单的设计原则：单一职责原则、开放封闭原则、依赖倒置原则、接口隔离原则、Liskov替换原则，这5个原则贯彻了一个简单的思想——“面向抽象，松散耦合”。
- **设计模式 (Design Pattern)**，23个设计模式，23个经典智慧，了解和掌握几个重要的设计模式，是修炼面向对象招式的必经之路。无论如何，作为设计者你应该在自己心中知道什么是Abstract Factory、Iterator、Singleton、Adapter、Decorator、Observer、Façade、Template、Command。
- **模式之外**，除了23个经典的设计模式，其实还存在很多模式之外的模式，按照粒度的不同而广泛应用于实际的项目系统中。例如，在SOA系统中的Event driven Architecture、Message Bus以及分布式Broker模式；数据持久层的Repository、Active Record还有Identity Map模式；可伸缩系统下的Map/Reduce、Load Balancer以及Result Cache模式；更高架构层面的MVC、MVP以及Pipeline/Filter模式。在某种意义上而言，模式是一种经验的积累和总结，对于系统设计与架构考量，架构师要完成的不仅仅是对功能性需求的把握，还包括非功能性需求、平台与框架的平衡、性能与安全的考量。

在本书第1部分，以“OO大智慧”和“OO大原则”两章的篇幅，分别讲述了关于面向对象的实现本质

和思想理念，以面向对象技术在.NET中的应用为起点，熟悉和领略面向对象的智慧与原则，修炼深入.NET技术的基本功，为深入理解.NET的程序设计打好必备的基础。而本章将对以上设计问题继续探讨，从点点滴滴入手来关注设计环节的下一个据点。

所以，下面我们将对软件开发中的设计与架构进行更多的探讨，以期收获更多的共识与争论。

3.1.4 结论

周星驰在《食神》中历经磨难之后参加食神大赛，做了一顿令人抓狂的黯然销魂饭，并对对手说了句：其实每个人都是食神，引来对手不屑。同样，现实世界的历练也渗透着同样的感悟，每个人都是食神，或者说每个人都可以是食神。软件设计与架构同样如此，不经一番寒彻骨，哪得梅花扑鼻香。

话说

其实每个人都是食神，其实开发者都是设计师。关键在于掌勺的你，是否能让做饭的家伙油光锃亮。

其实，在设计的领域，你大可不必为看似高深的框架吓倒，也不必为没有经验而怯场。在每个人的代码生涯中，你随时可以是食神，就像上例中通过简单 Extract Class 重构方法，你就可以体验一次化腐朽为神奇的力量。所以，设计无处不在，架构如影随形。而如何将三层架构、Abstract Factory、Extract Method、MVC、OCP 这一竿子打不着的概念有机地、科学地、合理地体现在活生生的软件系统中，是一种功力和经验的体现。

作为学习者，如果还不具备在宏观上把握如何将上述模糊的概念进行统筹和消化，那么作为预备设计师，首先要做的工作就是先逐个认识 SOA、Mapper、Pipeline、DTO、Message Bus 这些概念，有了基本功之后再看着唱本骑驴走远。

3.2 依赖的哲学

本节将介绍以下内容：

- 关于依赖和耦合
- 面向抽象编程
- 依赖倒置原则
- 控制反转
- 依赖注入
- 工厂模式
- Unity 框架应用

3.2.1 引言

“不要调用我们，我们会调用你”是对 DIP 最形象的诠释。作为 5 大设计原则之一的 DIP 原则，有了 2.4 节“依赖倒置原则”由概念而实例的单纯讨论，还不能全面阐释清楚：

- 什么是依赖倒置？

- 为什么依赖倒置？
- 如何依赖倒置？

这几个关键的问题，不单纯地通过 DIP 而 DIP，而是从依赖这个最原始的概念讲起，来了解在面向对象软件设计体系中，关于“关系的处理”，也就是“依赖的哲学”。对，依赖就是关系，处理依赖也就意味着处理关系。人类是最善于搞关系的动物，所以原本简单的理论，在人类的意识哲学中变得复杂而多变，以至于本应简单的道理变得如此复杂，这就是依赖。那么，从依赖讲起来了解依赖倒置原则，首先应该回答以下的问题：

- 控制反转、依赖倒置、依赖注入这些概念，你认识但是否熟悉？
- Unity、ObjectBuilder、Castle 这些容器，你相识但是否相知？
- 面向接口、面向抽象、开放封闭这些思想，你了解但是否了然？

带着对这些问题的思考和思索，本文带领大家就依赖这个话题开始一次循序渐进的面向对象之旅，以解答这些从一开始就有足够吸引力的问题。从原理到实例，从关系到异同，期待接下来的内容能给你带来一些认知的变革。

3.2.2 什么是依赖，什么是抽象

1. 关于依赖和耦合：从小国寡民到和谐社会

在老子的“小国寡民”论中，提出了一种理想的社会状态：邻国相望，鸡犬之声相闻，民至老死，不相往来。这是他老人家的一种社会理想，老死不相往来的人群呈现了一片和谐景象。因为不发生瓜葛，也就无所谓关联，进而无法导致冲突。这是先祖哲学中的至纯哲理，但理想的大同总是和现实的生态有着或多或少的差距，人类社会无法避免联系的发生，所以小国寡民的理想成为一种美丽的梦想，不可实现。同样的道理，映射到软件“社会”中，也就是软件系统结构中，也预示着不同的层次、模块、类型之间也必然存在着或多或少的联系，这种联系不可避免但可管理。正如人类社会虽然无法实现小国寡民，但是理想的状态下我们推崇和谐社会，把人群的联系由复杂变为简单，由曲折变为统一，同样可以使得这种关联很和谐。所以，软件系统的使命也应该朝着和谐社会的目标前进，对于不同的关系处理，使用一套行之有效的哲学，把复杂问题简单化，把僵化问题柔性化，这种哲学或者说方法，在我看来就是：依赖的哲学，也就是本文所要阐释的中心思想。

因为“耦合是不可避免的”，所以首先就从认识依赖和耦合的概念开始，来一步步阐释依赖的哲学思想。

(1) 什么是依赖和耦合

依赖，就是关系，代表了软件实体之间的联系。软件的实体可能是模块，可能是层次，也可能是具体的类型，不同的实体直接发生依赖，也就意味着发生了耦合。所以，依赖和耦合在我看来是对一个问题的两种表达，依赖阐释了耦合本质，而耦合量化了依赖程度。因此，对于关系的描述方式，就可以从两个方面的观点来分析。

从依赖的角度而言，可以分类为：

- 无依赖，代表没有发生任何联系，所以二者相互独立，互不影响，没有耦合关系。
- 单向依赖，关系双方的依赖是单向的，代表了影响的方向也是单向的，其中一个实体发生改变，会对

另外的实体产生影响，反之则不然，耦合度不高。

- 双向依赖，关系双方的依赖是相互的，影响也是相互的，耦合度较高。

从耦合的角度而言，可以分类为（此处回归到具体的代码级耦合概念，以方便概念的阐释）：

- 零耦合，表示两个类没有依赖。
- 具体耦合，如果一个类持有另一个具体类的引用，那么这两个类就发生了具体耦合关系。所以，具体耦合发生在具体类之间的依赖，因此具体类的变更将引起对其关联类的影响。
- 抽象耦合，发生在具体类和抽象类的依赖，其最大的作用就是通过对抽象的依赖，应用面向对象的多态机制，实现了灵活的扩展性和稳定性。

不同的耦合，代表了依赖程度的差别，以“粒度”为概念来分析其耦合的程度。引用中间层来分离耦合，可以使设计更加优雅，架构更加富有柔性，但直接的依赖也存在其市场，过度的设计也并非可取之道。因为，效率与性能同样是设计需要考量的因素，过多的不必要分离会增加调用的次数，造成效率浪费。

后文分析依赖倒置原则的弊端之一正是对此问题的进一步阐述。

(2) 耦合是如何产生的

那么，软件实体之间的耦合是如何产生呢？回归每天挥洒的代码片段，其实就是在重复的创造着耦合，并且得益于对这种耦合带来的数据通信。如果将历史的目光回归到软件设计之初，人类以简单的机器语言来实现最简单的逻辑，给一个输入，实现一个输出，可以表达为如图 3-1 所示的形式。

随着软件世界的革命，业务逻辑的复杂，以上的简单化处理已经不足以实现更复杂的软件产品，当系统内部的复杂度超越人脑可识别的程度时，就需要通过更科学的方法或者方式来梳理，如图 3-2 所示。

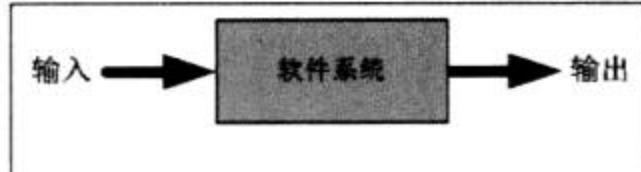


图 3-1 软件的输入和输出

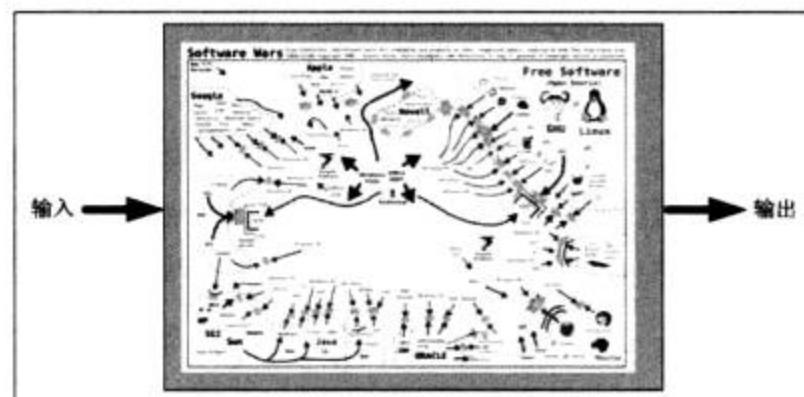


图 3-2 复杂系统的输入和输出

因此，人类开始发挥重组和简单化处理的优势，开发者不得不在软件设计上做出平衡。平衡的结果就是通过对复杂的系统模块化，把复杂问题简单处理，从而达到能够被人脑识别的目的。基于这种指导原则，随着复杂度的增加模块的划分更加朝着精细化发展，尤其是面向对象程序设计理论的出现，使得对复杂的处理实现了更科学的理论基础。然而，复杂的问题可以通过划分实现简单的功能模块或者技术单元，但由此应运而生的子单元会越来越多，而且越来越多的子单元必须发生数据的通信才能完成统一的业务处理，所以产生的数据通信管理也越来越多。对于子单元的管理，也就是本文关注的核心概念——依赖，成为新的软件设计问题，那么总结前人的经验，提炼今人的智慧，对耦合的产生做如下归纳：

- 继承
- 聚合

- 接口
- 方法调用和引用
- 服务调用

了解了耦合发生的一般方式，就可以进入核心思想的讨论，那就是在认识和了解依赖的基础上，最终追求的目标。

话说

设计的目标：高内聚（High cohesion）、低耦合（Low coupling）。

讨论了半天，终于是时候对依赖和耦合进行一点儿总结了，也是该进行一点目标诉求了。在软件设计领域，有那么几个至高原则值得每个开发者铭记于心，它们是：

- 面向抽象编程
- 低耦合，高内聚
- 封装变化
- 实现重用：代码重用、算法重用

对了，就是这些平凡的字眼，汇集了面向对象思想的核心内容，也是本文力求阐释的禅意心经。关于面向抽象编程和封装变化，会在后面详细阐释，在此我们需要将注意力关注于“低耦合，高内聚”这一目标。

低耦合，代表了实现最简单的依赖关系，尽可能地减少类与类、模块与模块、层次与层次、系统与系统之间的联系。低耦合，体现了人类追求简单操作的理想状态，按照软件开发的基本实现技巧来追求软件实体之间的关系简单化，正是大部分设计模式力图追求的目标；低耦合，降低了一个类或一个模块发生修改对其他类或模块造成的影响，将影响范围简单化。在本文阐释的依赖关系方式中，实现单向的依赖，实现抽象的耦合，都是实现低耦合的基础条件。

高内聚，一方面代表了职责的统一管理，一方面体现了关系的有效隔离。例如单一职责原则其实归根结底是对功能性的一种指导性体现，将功能紧密联系的职责封装为一个类（或模块），而判断的准则正是基于引起类变化的原因。所以，封装离不开依赖，而抽象离不开变化，二者的概念和本质都是相对而言的。因此，高内聚的目标体现了以隔离为目标进行统一管理的思想。

那么，为了达到低耦合、高内聚的目标，通常意义上的设计原则和设计模式其实都是朝着这个方向实现的，因此仅仅总结并非普遍意义的规则：

- 尽可能实现单项依赖。
- 不需要进行数据交换的双方，不要实现多此一举的关联，人们将此形象称为“不要向陌生人说话（Don't talk to strangers）”。
- 保持内部的封装性，关联的双方不要深入实现细节进行通信，这是保证高内聚的必需条件。

2. 关于抽象和具体

什么是抽象呢？首先不必澄清什么是抽象，而从什么算抽象说起，稳定的、高层的就代表了抽象。就像一个公司，最好保证了高层的稳定，才能保证全局的发展。在进行系统设计时，稳定的抽象接口和高层逻辑，也代表了整个系统的稳定与柔性。兵熊熊一窝，将良良一窝，软件的构建也正如打仗，良好的设计都是自上而下的。而对具体的编程实践而言，接口和抽象类则代表了语言层次的抽象。

追溯概念的分析，一一过招，首先来看依赖于具体，如图 3-3 所示。

因此，为了分离这种紧耦合，最好的办法就是隔离，引入中间层来分离变化，同时确保中间层本身的稳定性，因此抽象的中间层是最佳的选择（如图 3-4 所示）。

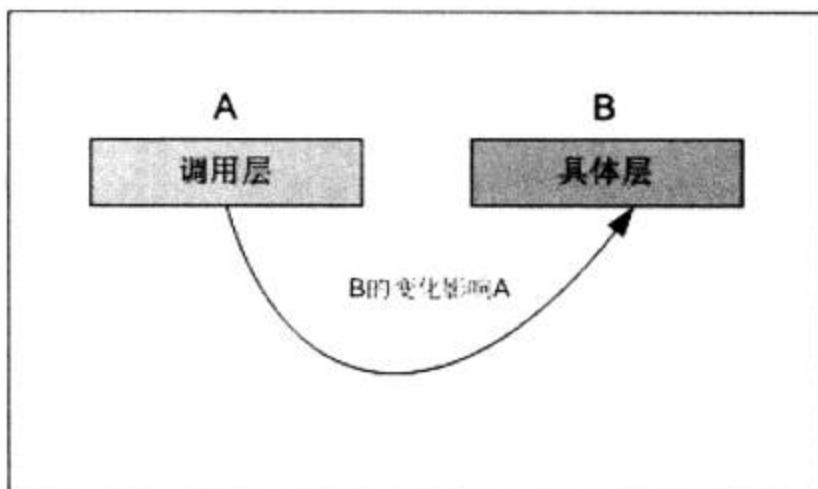


图 3-3 依赖的关系

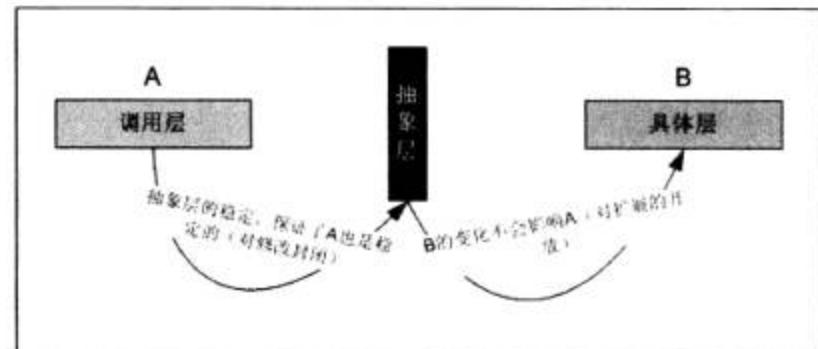


图 3-4 依赖的关系（引入抽象层）

以例而理，从最常见的服务端逻辑举例，如下所示：

```

public interface IUserService
{
}

public class UserService : IUserService
{
}

```

如果依赖于具体：

```

public class UserManager
{
    private UserService service = null;
}

```

或者依赖于抽象：

```

public class UserManager
{
    private IUserService service = null;
}

```

二者的区别仅在于引入了接口 IUserService，从而使得 UserManager 对于 UserService 的依赖由强减弱。然而对于依赖的方式并非仅此一种，设计模式中的智慧正是通过各种编程技巧进行依赖关系的解耦，值得关注和学习，后文将对设计模式进行概要性的讨论。

对 WCF 熟悉的读者一定不难看出这种实现方式如此类似于 WCF 的推荐模式，也是契约编程的基本思想。关于 WCF 及 SOA 的相关内容，我们在后文进行了相关的讨论。

总结一番，什么是抽象，什么是具体？在作者看来，抽象就是系统中对变化封装的战略逻辑，体现了系统的必然性和稳定性，能够被具体层次复用和覆写；而具体则包含了与具体实现相关的逻辑，体现了系统的动态性和变动性。因此，抽象是稳定的，而具体是变动的。

Bob 大叔在《Agile Principles, Patterns, and Practices》一书中直言，程序中所有的依赖关系都应终止于抽

象类或者接口，就是对面向抽象编程一针见血的回应，其原因归根到底源自于对抽象和具体的认知和分解：关联应该终止于抽象，而不是具体，保证了系统依赖关系的稳定。具体类发生的修改，不会影响其他模块或者关系。那么如何做到这种理想的依赖于抽象的设计呢？

- 层次清晰化

将复杂的问题简单化，是人类思维的普世智慧，也自然是实现软件设计的基本思路。将复杂的业务需求通过建模过程的抽象化提炼，去粗取精，去伪存真，凡此种种。而抽象的过程，其目标之一就是形成对于复杂问题简单化的处理过程，只有形成层次简单的逻辑才能将复杂需求中的关系梳理清晰，而依赖的本质正如上文所言，不就是处理关系吗？

所以，清晰的层次划分，进而形成的模块化，是实现系统抽象的必经之路。

- 分散集中化

由需求而设计的过程，就是一个分散集中化的过程，把需求相关的业务通过开发流程的需求分析过程进行整理，逐步形成需求规格说明、概要设计和详细设计等基本流程。分散集中化，是一个梳理需求到形成设计的过程，因此对于把握系统中的抽象和具体而言，是一个重要的分析过程和手段。现代软件工程已经对此形成了科学的标准化流程处理逻辑，例如可以借助 UML 更加清晰地设计流程、分析设计要素，进行标准化沟通和交流。

- 具体抽象化

将具体问题抽象化，是本节关注的要点，而处理的方法是什么呢？答案就在设计模式。设计模式是前辈智慧的总结和实践，所以熟悉和学习设计模式，是学习和实践设计问题的必经之路。然而，没有哪个问题是完全由设计模式全权解决，也没有哪个模式能够适应所有的问题，因此要努力的是尽量积累更多的模式来应对多变的需求。作为软件设计话题中最重量级的话题，关注模式和实践模式是成长的记录。

- 封装变化点

总的来说，抽象和变化就像一对孪生兄弟，将具体的变化点隔离出来以抽象的方式进行封装，在变化的地方寻找抽象是面对抽象最理想的方式。所以，如何去寻找变化是设计要解决的首要问题，例如工厂模式的目标是封装对象创建的变化，桥接模式关注对象间的依赖关系变化等。23个经典的设计模式，从某种角度来看，正是对不同变化点的封装角度提出的不同解决方案。

这一设计原则还被称为 SoC (Separation of Concerns) 原则，定义了对于实现理想的高耦合、低内聚目标的统一规则。

3. 设计的哲学

之所以花如此篇幅来讲述一个看似简单的问题，其实最终理想是回归到软件设计目标这个命题上。如果悉心钻研就可发现，设计的最后就是对关系的处理，正如同生活的意义在于对社会的适应一样。因此，回归到设计的目标上就自然可知，完美的设计过程就是对关系的处理过程，也就是对依赖的梳理过程，并最终形成一种合理的耦合结果。

所以，面向对象并不神秘，以生活的现实眼光来看更是如此。把面向对象深度浓缩起来，可以概括为：

- 目标：重用、扩展、兼容。

- 核心：低耦合、高内聚。
- 手段：封装变化。
- 思想：面向接口编程、面向抽象编程、面向服务编程。

其实，就是这么简单。在这种意义上来说，面向对象思想是现代软件架构设计的基础。下面以三层架构的设计为例，来进一步感受这种依赖哲学在具体软件系统中的应用。关于依赖的抽象和对变化隔离的基本思路，其实也是实现典型三层架构或者多层架构的重要基础。只有使各个层次之间依赖于较稳定的接口，才能使得各个层次之间的变化被隔离在本层之内，不会造成对其他层次的影响，这完全符合开放封闭原则追求的优良设计理念。将这种思路表达为设计，可以表示为如图 3-5 所示的形式。

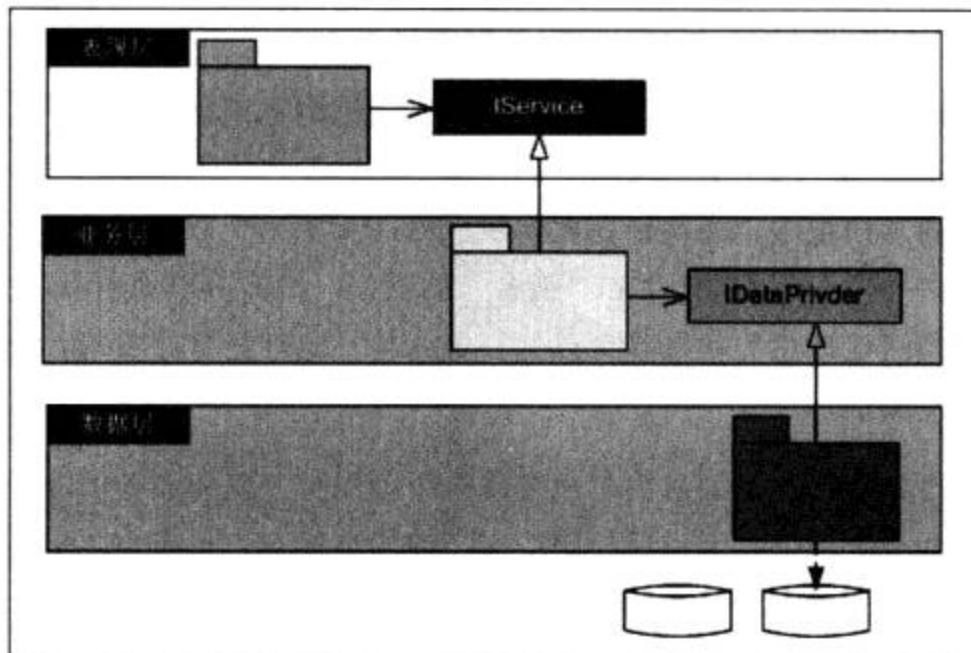


图 3-5 多层架构的依赖

由图 3-5 可知，`IDataProvider` 作为隔离业务层和数据层的抽象，`IService` 作为隔离业务层和表现层的抽象，保证了各个层次的相对稳定和封装。而体现在此的设计逻辑，就正是对于抽象和耦合基本目标概念的体现，例如作为重用的单元，抽象隔离保证了对外发布接口的单一和稳定，所以达到了最高限度的重用；通过引入中间的稳定接口，达到了不同层次的有效隔离，层与层之间体现为轻度耦合，业务层只持有 `IDataProvider` 就可以获取数据层的所有服务，而表现层也同样如此；最后，这种方式显然也直接实践了面向接口编程，面向抽象编程的经典理念。

同样的道理，对于架构设计的很多概念，放大可以扩展为面向服务设计所借鉴，放小这正是反复降调的依赖倒置原则在类设计中的基本思想。因此，牢记一位软件大牛的说法：软件设计的任何问题，都可以通过引入中间逻辑得到解决。而这个中间逻辑，很多时候被封装为抽象，是最为合理和智慧的解决方案。

让我们再次高颂《老子》的小国寡民论，来回味关于依赖哲学中，如何实现更好的和谐统一以及如何遵守科学的软件管理思想：邻国相望，鸡犬之声相闻，民至老死，不相往来。

3.2.3 重新回到依赖倒置

1. 什么是依赖倒置

Bob 大叔在《Agile Principles, Patterns, and Practices》一书中对依赖倒置原则进行了精辟的总结：

- 高层模块不应该依赖于低层模块，二者都应该依赖于抽象。
- 抽象不应该依赖于具体，细节应该依赖于抽象。

其实著名的好莱坞原则更形象地阐述了这一思想：你不要调我，我来调你。不管是通俗的还是高尚的，却都不约而同地揭示了依赖倒置原则的最核心思想就是：

依赖于抽象，对接口编程，对抽象编程！

关于依赖倒置原则的基本概念，可参考2.4节“依赖倒置原则”。回到对思想与设计的层面，相较而言，从实际的生活中来看依赖倒置，就像接下来的实例一样。

2. 从实例开始

综合对依赖倒置的认识，结合到具体的程序实现而言，依赖倒置预示着程序中的依赖关系不应是具体的类型，而应归于抽象类和接口。下面通过一个简单的实例来分析符合依赖倒置和违反依赖倒置及其对于系统设计的影响和区别。示例的客户被假定为某个遥控器生产商，实现一个万能遥控器，该遥控器可以对当前市场上的很多电子设备进行“打开”、“关闭”和“换台”的操作，例如可以使用万能牌遥控器打开海尔电视、创维电视或者长虹电视，当然更理想的状态是可以打开电冰箱、电灯还有门窗等，总之凡是可以通过互联网连接的设备都是未来万能遥控器的潜在需求。

那么该遥控器厂商在设计之初，该如何去考虑实现一个可以打开任何设备的遥控器呢？这一重责首先落在了一位年轻气盛的设计师小王身上，因为遥控器厂家当前的直接客户只有海尔电视一家，所以他轻松地实现了下面的设计，并且兴高采烈地进行了大批量生产（如图3-6所示）。

随后，厂商多了一个重量级客户长虹，所以小王不得不对初试设计进行了改造，勉强适应了新的需求，如图3-7所示。

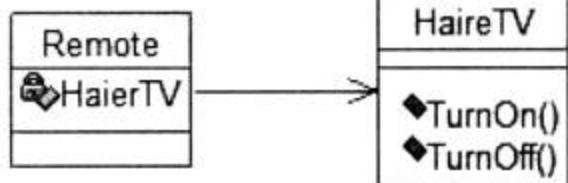


图 3-6 遥控器初次设计

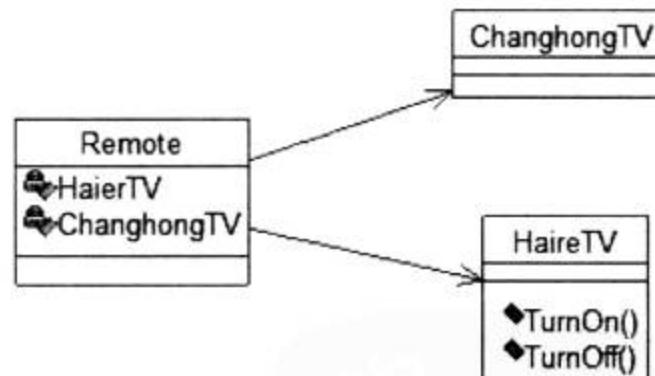


图 3-7 遥控器二次设计

虽然小王应付了这次需求变动，但是原本的设计显然已经捉襟见肘。正当小王绞尽脑汁进行改造的同时，新的需求接踵而来：新飞冰箱、飞利浦照明、盼盼防盗门，一个接一个。小王的最终设计变成了如图3-8所示的模样。

哎，真是太累了。每一次的需求变更都伴随着小王对遥控器`Remote`的再次摧残，`Remote`内部不断增加新的引用和操作处理，显然一个`if/else`式的判断布满了整个`Open`和`Close`的操作中，这种设计显然无法满足OCP对扩展开放及修改封闭的要求。显然，如果想让卖出去的遥控器也适应新的需求，在小王当前的设计实现方案中是根本无法实现的，遥控器厂商总不能召回已经售出的所有控器，再拆开进行重新改造吧。

一筹莫展的小王，终于在崩溃之际想起了退休在家的前设计师老张，并立即请教如何解决当前的问题。老张经验丰富、为人谦和，毫不含糊地给出一个初步的实现（如图3-9所示）：

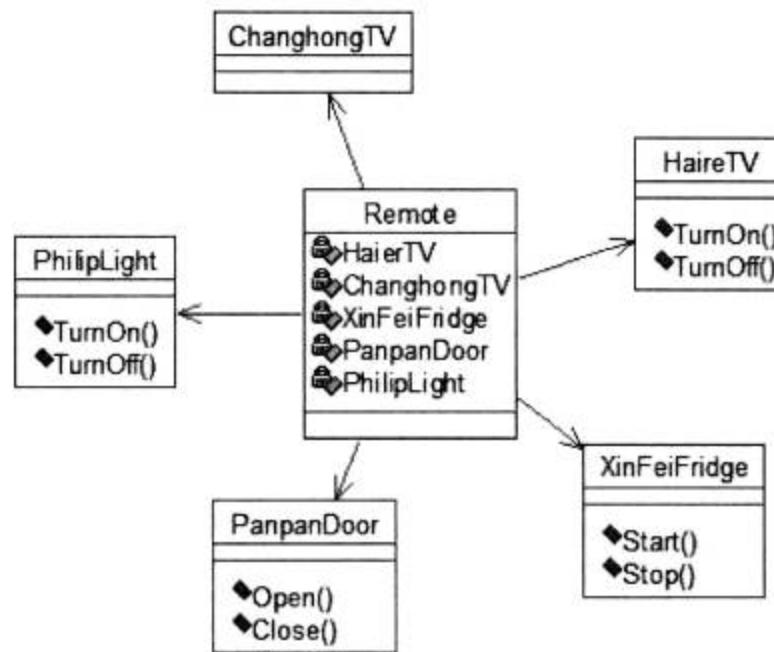


图3-8 遥控器三次设计

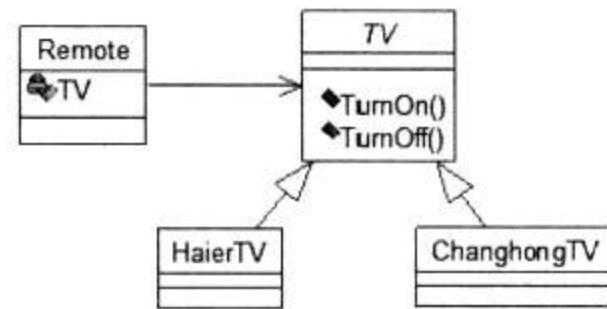


图3-9 重构的遥控器设计

在当前的设计中，老张的思路是让遥控器厂切断和各个厂家的直接联系，而是寻找所有电视厂商的领导（如电视机协会），请电视机协会制定所有电视机厂商必须遵守的打开和关闭等操作的契约，遥控器厂和电视机协会建立直接的联系而不是各个具体的电视厂商，于是便有了上述设计思路。而新的需求来临时，因为各个厂商必须遵守 TurnOn 和 TurnOff 的契约，所以万能遥控器可以应付所有的电视机品牌，实现的具体操作已经由遥控器转移到具体的厂商手上（这也是所有权的倒置体现），小王轻松地大呼一口气，并且青出于蓝地修改了更完善的版本，如图3-10所示。

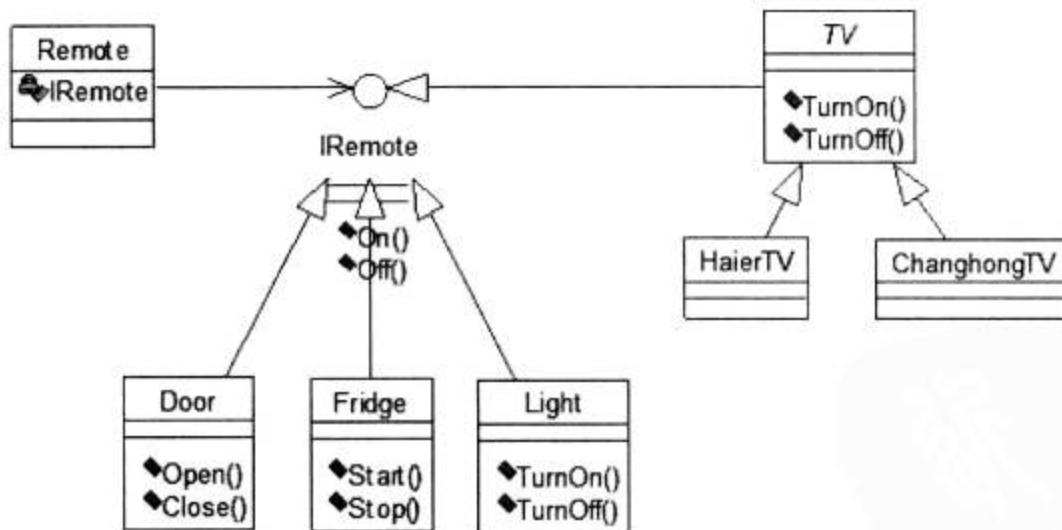


图3-10 重构的遥控器设计

现在，遥控器基本实现了万能的要求，任何新的需求或者修改都可以轻松胜任。小王终于解决了原本设计的所有问题，带着感激盛情邀请老张赴宴致谢。席间就座，小王请教老张重构设计的秘诀，老张神秘一笑，嚼着茴香豆，沾酒在桌子上写了几个大字：**依赖倒置**。于是，小王会意地笑了。

万能遥控器的故事，是系统实现中经常的事儿。而这些设计在实际项目中有广泛的应用，例如对于DataProvider和Service的处理方式，正是一种典型的遵循DIP原则的设计思路。

3. 为什么依赖倒置

依赖倒置原则揭示了面向对象思想中一个最基本而最核心的话题，那就是：**面向抽象编程**。任何对依赖倒置原则的违反都不同程度地偏离了面向对象设计思想的轨道，所以如果你想让自己的程序足够的OO，透彻地了解依赖倒置是必不可少的。

所以，要问答为什么依赖倒置这个话题，可以从以下几个方面来阐释：

- 依赖倒置是保证开放封闭的前提和基础。
- 依赖倒置是对抽象和依赖的基本原则和基本思想的哲学阐释。
- 依赖倒置是框架设计的核心思想。
- 依赖倒置是控制反转和依赖注入的思想基础。

综上而言，依赖倒置是对软件实体关系处理的基本思想原则，也是其他设计原则与设计模式的基础之一，因此遵守依赖倒置是实现OO的基本原则，是必须了解的基础性原则。下面，我们对此进行详细的说明和举例。

4. 为什么是倒置

鲁迅先生有云：其实地上本没有路，走的人多了也便成了路。对依赖倒置原则中的“倒置”二字而言，其实也类似于一条被很多人走过的路，因为习惯性地称呼走过的为“路”，所以只好把违反习惯的东西称为“倒置的路”。这倒置的含义，正在于此。

对于从结构化编程走过的人来说，基于软件复用的考虑，侧重于对具体模块的复用，因而也就习惯了从高层模块出发构建系统流程的思维模式，所以那时的高手一出手就实现了高层依赖于底层的典型套路，如图3-11所示。

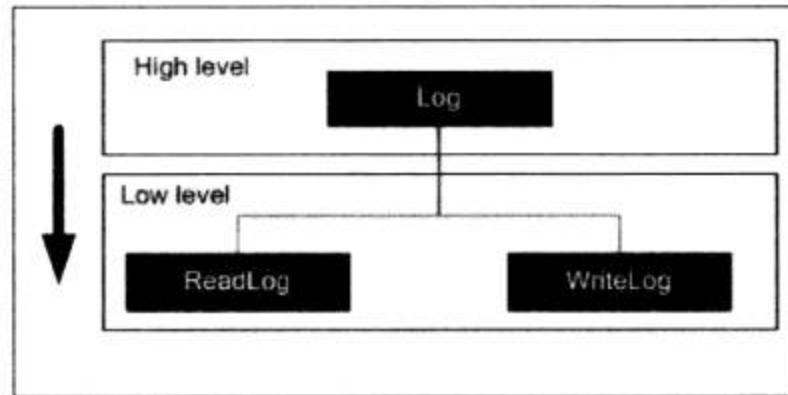


图3-11 高层依赖于底层

高层模块通过自上而下的实现，来完成系统功能的调用，将这种方式表达为代码就是：

```
public static void Main()
{
    try
    {
        //Do something here.
    }
    catch
    {
        Log(true, "XMLLog");
    }
}

public static void Log(bool isRead, string logType)
```

```

    {
        if (isRead)
            ReadLog(logType);
        else
            WriteLog(logType);
    }

```

然而，当软件设计的模式发展到面向对象阶段时，人们发现原来习惯的世界已经变了。基于高层依赖于底层的弊政，也越来越被可扩展性的系统需求折磨得面目全非，例如日志记录的载体发生变化，当前设计中需要同时自上而下地修改实现的逻辑，同时避免出现越来越多的 if/else 结构。所以当新的依赖关系从传统的方式被完全扭转时，“倒置”二字就此诞生了。于是修改 Log 实现的设计思路，将可能变化的逻辑封装为抽象接口，使得高层依赖发生转换，如图 3-12 所示。

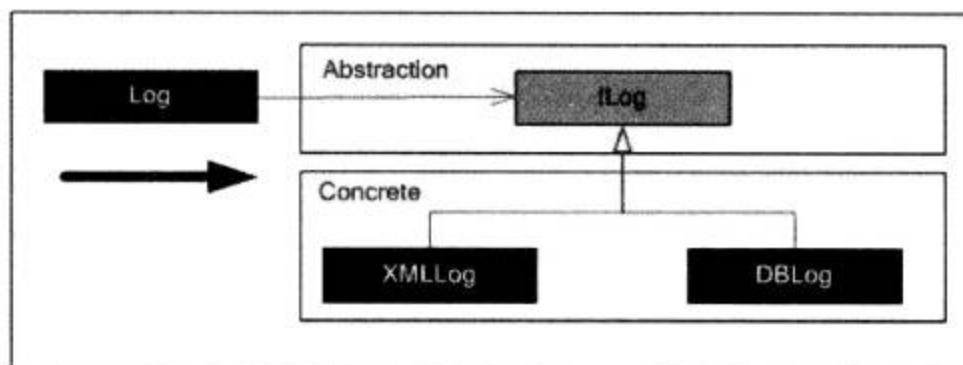


图 3-12 高层依赖于抽象

程序实现的逻辑早已被面向对象的设计思想所取代，新的实现变成了如下形式：

```

public class Client
{
    public static void Main()
    {
        ILog myLogger = new XMLLog();
        try
        {

        }
        catch
        {
            myLogger.Write();
        }
    }
}

public interface ILog
{
    void Read();
    void Write();
}

public class XMLLog : ILog
{
    public void Read()
    {

    }

    public void Write()
    {

    }
}

```

所以，了解了历史才能正视现实，对于软件设计同样如此，只有认清楚依赖倒置产生的历史背景，才能更加熟练地驾驭倒置含义本身带来的误解，而将中心思想牢牢地把握在依赖倒置最核心的设计思想上，那还是万变不离其宗的：依赖于抽象，这简单5字箴言。

对于所属权关系的依赖问题上，可以看到，只有倒置的才是面向对象的，没有倒置的还是面向结构的。如果你的系统中存在着不合理的依赖关系，那么依赖倒置将是检查系统设计最好的标尺，这也是需要深入这一原则的实际意义之一。

5. 如何依赖倒置

如何依赖倒置的关键，还是体现在如何对抽象和具体的封装和分离，实践的基本思路就是**封装变化**。这正如在单一职责原则中反复强调，对一个类只有一个引起它变化的原因。实践依赖倒置，仍然可以从关注变化开始，详细分析和预测系统中的变化点，然后针对每个可能的变化抽象出相对稳定的约束，这是我们实践依赖倒置原则最基本的方法步骤。

就原理而言，依赖倒置要求设计：

- 少继承，多聚合。
- 单向依赖（低耦合，高内聚）。
- 封装抽象。
- 对依赖关系都应该终止于抽象类和接口。

就实践而言，经典的软件设计实践提出了很多值得借鉴的思路，例如每个设计模式就是对一种特定情况的实践总结，在此继续列出一些经典的大师忠言，Bob大叔在《Agile Principles, Patterns, and Practices》一书对此进行了3点总结：

- 任何变量都不应该持有一个指向具体类的指针或者引用。
- 任何类都不应该从具体类派生。
- 任何方法都不应该覆写它的任何基类中已经实现的方法。

实际上，在实际的设计过程中要完全遵守这几点要求是有难度的，所以如何既能很好地遵守设计原则，又能很好地适应代码情况，是值得权衡的问题，需要不断地积累和实践。另外，还有几个经验之谈：

- 系统架构应该有清晰的层次定义，层次之间通过接口向外提供内聚服务，正如在三层架构中的示例一样。
- 典型的以new进行的对象创建操作，是对依赖倒置原则的典型违反，而通过依赖注入进行对象的创建解耦是常用的解决之道。

如何依赖倒置，我们阐释了一点原则还有一点方法，算是对实现依赖倒置的一点小结。然而，在实际的开发过程中，并没有一成不变的规则，当前的面向对象语言本身就提供了对抽象和封装的支持，为实现面向对象设计提供了基础机制。回顾软件开发的历史，不难看出依赖和封装哲学的发展轨迹，在结构化编程中函数是封装的基本单元；随着面向对象的发展，C++/C#高级语言以类为基本单元，第一次将数据和行为有机地组合为一个逻辑单元，于是有了对于不同类之间的关系处理哲学；而SOA中封装的单元上升为一个服务

(service)，是一种更高意义的逻辑封装，实现了更优良的逻辑封装和松散耦合关系。同样的道理，也体现在三层架构的分割和通信中，体现在 ORM 对表现层和领域层的分离中。

因此，依赖倒置是一种高度的智慧和经验总结，如何实现依赖倒置也是一种积累和不断学习的过程。

6. 也有弊端

然而，一味地遵守原则，就等于没有原则。重要的是，需要把握其平衡，在进行开发中适当地把握其程度。Bob 在《敏捷》中也提到这个问题，总结了依赖倒置的两个弊端，同样需要特别的关注：

- 对抽象编程，需要增加必要的类和辅助代码进行支持，某种程度上增加了系统复杂度和维护成本。
- 当具体类不存在变化时，遵守依赖倒置是多此一举。所以，如果具体或细节没有变化可能时，没有必要通过抽象转嫁依赖。

所以，学习模式或者原则必须灵活处理，不能一味强行。

3.2.4 解构控制反转 (IoC) 和依赖注入 (DI)

1. 控制反转

控制反转 (Inversion of Control, IoC)，简言之就是代码的控制器交由系统控制，而不是在代码内部，通过 IoC，消除组件或者模块间的直接依赖，使得软件系统的开发更具柔性和扩展性。控制反转的典型应用体现在框架系统的设计上，是框架系统的基本特征，不管是.NET Framework 抑或是 Java Framework 都是建立在控制反转的思想基础之上。

控制反转很多时候被看做是依赖倒置原则的一个同义词，其概念产生的背景大概来源于框架系统的设计，例如.NET Framework 就是一个庞大的框架 (Framework) 系统。在.NET Framework 大平台上可以很容易地构建 ASP.NET Web 应用、Silverlight 应用、Windows Phone 应用或者 Window Azure Cloud 应用。很多时候，基于.NET Framework 构建自定义系统的方式就是对.NET Framework 本身的扩展，调用框架提供的基础 API，扩展自定义的系统功能和行为。然而，不管如何新建或者扩展自定义功能，代码执行的最终控制权还是回到框架中执行，再交回应用程序。黄忠诚先生曾经在 Object Builder Application Block 一文中给出一个较为贴切的举例，就是在 Window From 应用程序中，当 Application.Run 调用之后，程序的控制权交由 Windows Froms Framework 上。所以，控制反转更强调控制权的反转，体现了控制流程的依赖倒置，所以从这个意义上来说，控制反转是依赖倒置的特例。

2. 依赖注入

依赖注入 (Dependency Injection, DI)，早见于 Martin Flower 的 Inversion of Control Containers and the Dependency Injection pattern 一文，其定义可概括为：

客户类依赖于服务类的抽象接口，并在运行时根据上下文环境，由其他组件（例如 DI 容器）实例化具体的服务类实例，将其注入到客户类的运行时环境，实现客户类与服务类实例之间松散 的耦合关系。

(1) 常见的三种注入方式

简单而言，依赖注入的方式被总结为以下三种。

- 接口注入（Interface Injection），将对象间的关系转移到一个接口，以接口注入控制。

首先定义注入的接口：

```
public interface IRunnerProvider
{
    void Run(Action action);
}
```

为注入的接口实现不同环境下的注入提供器，本例的系统是一个后台处理程序提供了运行环境的多种可能，默认情况下将运行于单独的线程，或者通过独立的 Windows Service 进程运行，那么需要为不同的情况实现不同的提供器，例如：

```
public class DefaultRunnerProvider : IRunnerProvider
{
    #region IRunnerProvider Members

    public void Run(Action action)
    {
        var thread = new Thread(() => action());
        thread.Start();
    }

    #endregion
}
```

对于后台服务的 Host 类，通过配置获取注入的接口实例，而 Run 方法的执行过程则被注入了接口所定义的逻辑，该逻辑由上下文配置所定义：

```
public class RunnerHost : IDisposable
{
    IRunnerProvider provider = null;

    public RunnerHost()
    {
        // Get Provider by configuration
        provider = GetProvider(config.Host.Provider.Name);
    }

    public void Run()
    {
        if (provider != null)
        {
            provider.Run(() =>
            {
                // execute logic in this provider, if provider is DefaultRunnerProvider,
                // then this logic will run in a new thread context.
            });
        }
    }
}
```

接口注入，为无须重新编译即可修改注入逻辑提供了可能，GetProvider 方法完全可以通过读取配置文件的 config.Host.Provider.Name 内容，来动态地创建对应的 Provider，从而动态地改变 BackgroundHost 的 Run() 行为。

- 构造器注入 (Constructor Injection)，客户类在类型构造时，将服务类实例以构造函数参数的形式传递给客户端，因此服务类实例一旦注入将不可修改。

```
public class PicWorker
{
}

public class PicClient
{
    private PicWorker worker;

    public PicClient(PicWorker worker)
    {
        // 通过构造器注入
        this.worker = worker;
    }
}
```

- 属性注入 (Setter Injection)，通过客户类属性设置的方式，将服务器类实例在运行时设定为客户类属性，相较构造器注入方式，属性注入提供了改写服务器类实例的可能。

```
public class PicClient
{
    private PicWorker worker;

    // 通过属性注入
    public PicWorker Worker
    {
        get { return this.worker; }
        set { this.worker = value; }
    }
}
```

另外，在.NET 平台下，除了 Martin Flower 大师提出的三种注入方式之外，还有一种更优雅的选择，那就是依靠.NET 特有的 Attribute 实现，以 ASP.NET MVC 中的 Action Filter 为例：

```
[HttpPost]
public ActionResult Register(RegisterModel model)
{
    // 省略注册过程
    return View(model);
}
```

其中，`HttpPostAttribute` 就是通过 Attribute 方式为 Register Action 注入了自动检查 Post 请求的逻辑，同样的注入方式广泛存在于 ASP.NET MVC 的很多 Filter 逻辑中。

```
[AttributeUsage(AttributeTargets.Method, AllowMultiple = false, Inherited = true)]
public sealed class HttpPostAttribute : ActionMethodSelectorAttribute
{
    // Fields
    private static readonly AcceptVerbsAttribute _innerAttribute = new AcceptVerbsAttribute(HttpVerbs.Post);

    // Methods
    public override bool IsValidForRequest(ControllerContext controllerContext, MethodInfo methodInfo)
    {
        return _innerAttribute.IsValidForRequest(controllerContext, methodInfo);
    }
}
```

关于 Attribute 的详细内容,请参考8.3节“历史纠葛:特性和属性”,其中的 TrimAttribute 特性正是应用 Attribute 注入进行属性 Trim 过滤处理的典型应用。

(2) 依赖注入框架

在.NET世界里,已经有很多久经考验的依赖注入容器可供选择,在实际的应用系统中选择合适的依赖注入容器,会大大减少对于业务对象的管理,建立更加松散的联系。

- Unity
- ObjectBuilder
- Castle
- Sprint.NET

限于篇幅,本书不对具体的容器特性进行探讨,读者可自行研究,其中微软的 Unity 容器正逐渐成为.NET 平台应用的首要选择,在本书后续内容中就有应用 Unity 进行对象创建过程的示例,值得特别关注和研究。

3. 关系: DIP、IoC 还有 DI

总体而言,DIP、IoC 还有 DI 之间有着剪不断理还乱的关系(如图 3-13 所示),其中 DIP 是对于依赖关系的理论总结,而 IoC 和 DI 则体现为具体的实践模式。IoC 和 DI 为消除模块或者类之间的耦合关系提供了有效的解决方案,从而保证了依赖于抽象和稳定模块或者类型,也就意味着坚持了 DIP 原则的大方向。



图 3-13 DIP、IoC 和 DI

而 IoC 和 DI 之间的区别主要体现在关注场合的不同:IoC 强调控制权的反转作用,着眼于流程控制的场合;而 DI 则关注层次与层次、组件与组件、模块与模块或者类型与类型之间的“倒置”,体现为设计模型上的依赖模式解构。

3.2.5 典型的设计模式

细数而来,几乎每个设计模式关注的核心都体现在依赖之上。创建型模式关注实例创建关系的依赖,结构型模式关注构建复杂对象过程的依赖,而行为型模式关注应用运行过程中的通信依赖。纵观 GoF 在《设计模式:可复用面向对象软件的基础》一书中梳理的 23 个设计模式,从依赖观点来看,有如表 3-1 所示的总结。

表 3-1 模式的依赖

类别	依赖	模式
创建型模式	创建型模式的核心关注点就在于对象创建的依赖关系上,将对象的依赖从 new 操作中解脱出来,隔离应用系统和类型实例化间的依赖。例如通过引入工厂,将对象创建职责委托于工厂类,解除了应用系统与类型对象在实例化过程中的直接引用。 详细的讨论,参考 3.2.7 节“对象创建的依赖”的论述	工厂方法、抽象工厂、单例、创建者、原型模式

续表

类别	依 赖	模 式
结构型模式	<p>结构型模式，是将简单类型组合为复杂类型的过程，通过灵活的设计要素，最终保证不同类型间保持尽量间接的引用和尽量松散的耦合，在复杂类型有更多变化与诉求时，以最小的代价兼容变化，扩展诉求。</p> <p>例如，适配器模式的两种不同适配方式，分别代表了通过继承和组合方式实现对象适配的处理；而代理模式，则通过引入代理，来隔离不同层次间的依赖，同时保证真实对象的安全性与封装性</p>	桥接、适配器、组合、外观、装饰、享元、代理
行为型模式	<p>行为型模式，关注对象行为的扩展和对象间数据关系的通信，以面向对象方式描述控制流。</p> <p>例如，职责链避免请求的发送者和接收者直接的直接耦合，而是将多个对象连成一条处理链条；而命令模式的核心则在于将行为的请求者和行为实现者之间通过封装的命令对象解耦</p>	模板方法、迭代器、中介者、职责链、解释器、命令、观察者、备忘录、状态、策略、访问者

3.2.6 基于契约编程：SOA 架构下的依赖

把握软件开发的历史脉搏，依赖关系的落脚点也在其演义过程中逐渐发生着改变，从面向过程以函数为核心，到面向对象以对象为核心，面向组件以组件为核心，再到面向服务中以服务为核心。基于契约编程的思想实现了更松散的耦合模型，当开发者调用 Facebook 服务获取好友列表这样的服务时，并不需要关心具体的服务内部逻辑，也不需要关注服务的物理存储，更不需要关心服务之间的关联关系，而只需要关注服务本身即可，如图 3-14 所示。

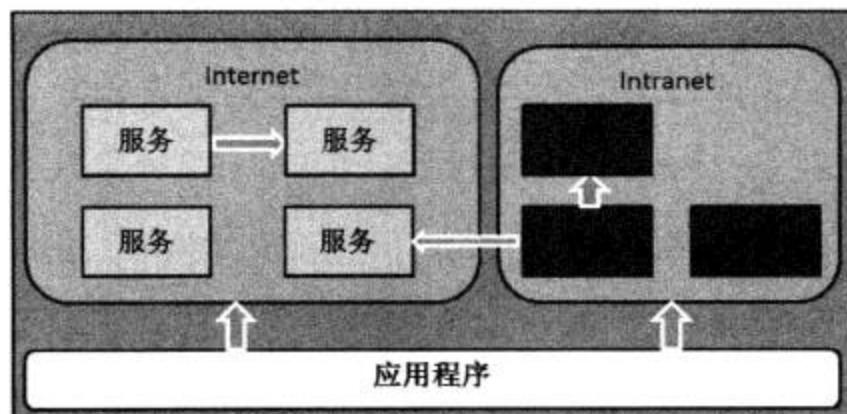


图 3-14 高层依赖于抽象

基于契约编程的依赖，就是对契约本身的依赖，也就是对具体服务的依赖。因此，SOA 架构下的依赖是天生松散耦合、高度抽象、实现技术无关且物理无关的。

- 松散耦合，服务之间的关系、应用与服务的关系都是松散的、单一的，基于消息的低耦合依赖。
- 高度抽象，服务本身不仅抽象了逻辑，还同时抽象了通信的契约与寻址。高度抽象的服务实体，是实现 SOA 架构的基础单元。
- 实现技术无关，服务本身的实现可以是各种各样的技术平台、版本。
- 物理无关，应用访问的服务部署，在物理上和应用本身是可以完全分离的，可以是局域网或者是互联网上任何位置，可以是不同的服务提供商，也可以运行在不同的时区。
- 基于消息，客户端与服务端基于标准的消息协议进行数据通信。

3.2.7 对象创建的依赖

关于依赖的哲学，最典型的违反莫过于对象创建的依赖。自面向对象的大旗树立以来，对于对象创建话题的讨论就从未停止。不管是工厂模式还是依赖注入，其核心的思想就只有一个：如何更好地解耦对象创建的依赖关系。所以，在这一部分，我们就以对象创建为主线，来认识对于依赖关系的设计轨迹，分别论述一般的对象创建、工厂方式创建和依赖注入创建三种方式的实现、特点和区别。

1. 典型的违反

一般而言，以 new 关键字进行对象创建，在.NET 世界里是天经地义的事情。在本书 7.1 节“把 new 说透”中，就比较透彻地分析了 new 在对象创建时的作用和底层机制。对.NET 程序员而言，以 new 进行对象创建已经是习以为常的事情，大部分情况下这种方式并没有任何问题。例如：

```
public abstract class Animal
{
    public abstract void Show();
}

public class Dog : Animal
{
    public override void Show()
    {
        Console.WriteLine("This is dog.");
    }
}

public class Cat : Animal
{
    public override void Show()
    {
        Console.WriteLine("This is cat.");
    }
}

public class NormalCreation
{
    public static void Main2()
    {
        Animal animal = new Dog();
    }
}
```

对 animal 对象而言，大部分情况下具体的 Dog 类是相对稳定的，所以这种依赖很多时候是无害的。这也是我们习以为常的原因之一。

然而，诚如在本文开始对抽象和具体的概念进行分析的结论一样，依赖于具体很多时候并不能有效地保证其稳定性状态。以本例而言，如果有新的 Bird、Horse 加入到动物园中来，管理员基于现有体系的管理势必不能适应形式，因为所有创建而来的实例都是依赖于 Dog 的。所以，普遍的对象创建方式，实际上是对 DIP 原则的典型违反，高层 Animal 的创建依赖于低层的 Dog，和普世的 DIP 基本原则是违背的。

因此，DIP 并不是时时被 OO 所遵守，开发者要做的只是适度的把握。为了解决 new 方式创建对象的依赖违反问题，典型的解决思路是将创建的依赖由具体转移为抽象，通常情况下有两种方式来应对：工厂模式

和依赖注入。

2. 工厂模式

以工厂模式进行对象创建的方法，主要包括两种模式：抽象工厂模式和工厂方法模式，本文不想就二者的区别和意义展开细节讨论，如果有兴趣可以参阅 GoF 的《设计模式：可复用面向对象软件的基础》一书。

本文将视角拉回到 WCF 的 IChannelFactory 和各种 Channel 的创建上，以此借用 WCF 架构中 Channel Layer 的设计思路，应用工厂模式进行对象创建的设计和扩展，来了解应用工厂模式进行对象创建依赖关系解除的实质和实现。

！注意

对于 WCF 中 Channel 的概念可以参考相关的资料，在此你只需将其看成一个简单类型即可。

首先来了解一下 Channel 的创建过程：

```
public class FactoryCreation
{
    public static void Main()
    {
        EndpointAddress ea = new EndpointAddress("http://api.anytao.com /UserService");
        BasicHttpBinding binding = new BasicHttpBinding();
        IChannelFactory< IRequestChannel> facotry = binding.BuildChannelFactory< IRequestC
channel>();
        facotry.Open();
        IRequestChannel channel = facotry.CreateChannel(ea);
        channel.Open();
        //Do something continue...
    }
}
```

在示例中， IRequestChannel 实例通过 IChannelFactory 工厂来创建，因此关注工厂方式的创建焦点就着眼于 IChannelFactory 和 IRequestChannel 上。实质上，在 WCF channel Layer 中， Channel Factory 是创建和管理 Channel 的工厂封装，通过一个个的 Channel Factory 来创建一个个对应的 Channel 实例，所有的 Channel Factory 必须继承自 IChannelFactory，其定义为：

```
public interface IChannelFactory< TChannel > : IChannelFactory, ICommunicationObject
{
    TChannel CreateChannel(EndpointAddress to);
    TChannel CreateChannel(EndpointAddress to, Uri via);
}
```

通过类型参数 TChannel 来注册创建实例的类型信息，进而根据 EndpointAddress 信息来创建相应的对象实例。当然，WCF 中的工厂模式应用，还有很多内容值得斟酌和学习。现有篇幅不可能实现完全类似的设计结构，借鉴于 WCF 的设计思路，对 Animal 实例的创建进行一点改造，实现基于泛型的工厂模式创建设计，首先定义一个对象创建的模板：

```
public interface IAnimalFacotry< TAnimal >
{
    TAnimal Create();
}
```

然后实现该模板的泛型工厂方法：

```
public class AnimalFacotry<TAnimalBase, TAnimal> : IAnimalFacotry<TAnimalBase> where TAnimal : TAnimalBase, new()
{
    public TAnimalBase Create()
    {
        return new TAnimal();
    }
}
```

其中类型参数 `TAnimalBase` 代表了高层类型，而 `TAnimal` 则代表了底层类型，其约定关系在 `where` 约束中有明确的定义，然后是一个基于对工厂方法的封装：

```
public class FacotryBuilder
{
    public static IAnimalFacotry<Animal> Build(string type)
    {
        if (type == "Dog")
        {
            return new AnimalFacotry<Animal, Dog>();
        }
        else if (type == "Cat")
        {
            return new AnimalFacotry<Animal, Cat>();
        }

        return null;
    }
}
```

最后，可以欣赏一下基于工厂方式的对象创建实现：

```
class Program
{
    static void Main(string[] args)
    {
        IAnimalFacotry<Animal> factory = FacotryBuilder.Build("Cat");
        Animal dog = factory.Create();
        dog.Show();
    }
}
```

你看，对象创建的依赖关系已经由 `new` 式的具体依赖转换为对于抽象和高层的依赖。在本例中，完全可以通过反射方式来消除 `if/else` 的运行时类型判定，从而彻底将这种依赖解除为可配置的灵活定制。这正是抽象工厂方式的伟大意义，其实在本例中完全可以将 `IAnimalFacotry` 扩展为 `IAnyFactory` 形式的灵活工厂，可以在类型参数中注册任何类型的 `TXXXBase` 和 `TXXX`，从而实现功能更加强大的对象生成器，只不过需要更多的代码和扩展，读者可以就此进行自己的思考。

3. 依赖注入

领略了工厂模式的强大威力，下面继续介绍更加灵活解耦的依赖注入方式，继续回到对于 `Animal` 实例化的依赖倒置环节，来看看注入方式下如何通过容器来实现实例创建过程。在此选择 Unity 基础容器来实现，引入 `Microsoft.Practices.Unity.dll` 程序集和 `Microsoft.Practices.Unity` 命名空间，然后就可以很容易地通过 Unity 容器来完成对象创建依赖关系的隔离：

```
class UnityCreation
{
    public static void Main()
    {
```

```

IUnityContainer container = new UnityContainer();
container.RegisterType<Animal, Dog>();

Animal dog = container.Resolve<Animal>();
dog.Show();
}
}

```

Unity 提供了强大而灵活的依赖注入支持：方法调用注入、属性注入和构造器注入等多种方式，可以在运行时或通过配置方式来注册和获取类型，是实现处理对象间依赖的有效方式。关于依赖注入，前文已有较多笔墨铺陈，在此就不做过多讨论。

综上而言，以对象创建这样一个常见而又简单的话题为焦点来讨论对这种依赖关系的场景复现。实际上，对象间的关系就像人类社会一样复杂多变，随时准备应对变化的依赖，着眼于对抽象的把握，是把复杂简单化的最佳实践，就类似于以工厂模式或者依赖注入方式将实体对象创建简单化处理的过程一样，是有意义的。

3.2.8 不规则总结

关于依赖的哲学，值得总结的有很多：

- 以 new 创建对象，是对依赖倒置原则的典型违反，可以通过工厂模式或者依赖注入来解决。
- 一个对象持有另外一个具体对象的引用可能破坏了依赖倒置。
- 所有结构良好的面向对象架构都具有清晰的层次定义，每个层次通过一个定义良好的受控接口向外提供一组内聚的服务。
- 依赖倒置预示着程序中的依赖关系不应是具体的类型，而应是抽象类和接口。
- 依赖倒置适用于当一个类向另一个类发送消息的任何情况。

3.2.9 结论

从小国寡民到和谐社会，不是一段寻常的路，走得越远想得越多，才能挥洒自如。正如对依赖的体会，一点一滴积累下来，剥丝抽茧才能层层深入。本文的循序之旅不可能尽述本质，如果能做到一点点亮思想的火柴就已是功得圆满。对于设计的精妙，体会抽象的层次，升华依赖的哲学，就是本文所得。

3.3 模式的起点

本节将介绍以下内容：

- 设计模式概要
- 面向对象

3.3.1 引言

设计模式是面向对象思想的集大成，GOF 在其经典著作《设计模式：可复用面向对象软件的基础》中总

结了23种设计模式，又可分为：创建型、结构型和行为型3个大类。对于软件设计者来说，一般的过程就是在熟练掌握语言背景的基础上，了解类库的大致框架和常用的函数和接口等，然后在百般锤炼中提高对软件设计思想的认识。

软件设计者要清楚自己的定位和方向，一味地沉溺于技术细节的思路是制约个人技术走向成熟的毒药。因此，学习软件设计，了解软件工程，是每个开发人员必备的一课。在本节，我们不欲详细描述各个设计模式的细节，Google和Baidu上的信息已经多如牛毛了，而只提纲挈领式地做以梳理，将23种模式中的关键模式拿出来晒晒。

3.3.2 模式的起点

■ 工厂方法 (Factory Method Pattern)

模式起点：将程序中创建对象的操作单独进行处理，大大提高了系统扩展的柔性，接口的抽象化处理给相互依赖的对象创建提供了最好的抽象模式。

典型应用：工厂方法模式是最简单也最容易理解的模式之一。其关注的核心是对于对象创建这件事儿的分离。

■ 单例 (Singleton Pattern)

模式起点：一个类只有一个实例，且提供一个访问全局点的方式，更加灵活地保证了实例的创建和访问约束，并且唯一约束的实施由类本身实现。

典型应用：一个类只有一个实例，经常被应用于 Façade 模式，称为单例外观。

■ 命令 (Command Pattern)

模式起点：将请求封装为对象，从而将命令的执行和责任分开。通常在队列中等待命令，这和现实多么相似呀。如果你喜欢发号施令，请考虑你的 ICommand 吧。

典型应用：菜单系统。

■ 策略 (Strategy Pattern)

模式起点：策略模式，将易于变化的部分封装为接口，通常 Strategy 封装一些运算法则，使之能互换。

典型应用：数据层常考虑以策略提供算法和数据的分离。

■ 迭代器 (Iterator Pattern)

模式起点：相信任何的系统中，都会用到数组、集合、链表、队列这样的类型吧，那么你就不得不关心迭代模式的来龙去脉。在遍历算法中，迭代模式提供了遍历的顺序访问容器，GOF给出的定义为：提供一种方法访问一个容器（Container）对象中各个元素，而又无须暴露该对象的内部细节。

典型应用：.NET 中就是应用了迭代器来创建用于 foreach 的集合。

■ 模板方法 (Template Method Pattern)

模式起点：顾名思义，模板方法就是在父类中定义模板，然后由子类实现。具体的实现一般由父类定义

算法的骨架，然后将算法的某些步骤委托给子类。

典型应用：ASP.NET 的 Page 类。

■ 观察者（Observer Pattern）

模式起点：定义对象间的一种一对多的依赖关系，当一个对象的状态发生改变时，所有依赖于它的对象都得到通知并被自动更新。观察者和被观察者的分开，为模块划分提供了清晰的界限。

典型应用：在.NET 中使用委托和事件可以更好地实现观察者模式，事件的注册和撤销不就对应着观察者对其对象的观察吗？

■ 职责链（Chain of Responsibility Pattern）

模式起点：将操作组成一个链表，通过遍历操作链表找到合适的处理器。通过统一的接口，被多个处理器实现，每个处理器都有后继处理器，可以将请求沿着处理器链传递。

典型应用：GUI 系统的事件传播。

■ 桥接（Bridge Pattern）

模式起点：把实现和逻辑分开，对于我们深刻理解面向对象聚合复用的思想甚有助益。

典型应用：多版本.NET Framework 通过环境变量与对应版本应用建立桥梁。

■ 代理（Proxy Pattern）

模式起点：将复杂的逻辑封装起来，通过代理对象控制实际对象的创建和访问，由代理对象屏蔽原有逻辑的复杂性，同时控制其可访问性。

典型应用：WCF 服务代理。

■ 装饰器（Decorator Pattern）

模式起点：为原有系统，动态地增加或者删除状态和行为，在继承被装饰类的同时包含被装饰类的实例成员。

典型应用：.NET 中 Stream 的设计。

■ 门面（Façade Pattern）

模式起点：将表现层和逻辑层隔离，封装底层的复杂处理，为用户提供简单的接口，这样的例子随处可见。门面模式很多时候更是一种系统架构的设计，在很多项目中，都实现了门面模式的接口，为复杂系统的解耦提供了最好的解决方案。

典型应用：WSDL 就是一个典型的平台无关的门面应用。

■ 组合（Composite Pattern）

模式起点：不管是个体还是组件，都包含公共的操作接口，通过同样的方式来处理一个组合中的所有对象。组件的典型操作包括：增加、删除、查找、分组和获取子元素等。

典型应用：树形结构的数据组织。

■ 适配器 (Adapter Pattern)

模式起点：在原类型不做任何改变的情况下，扩展了新的接口，灵活且多样地适配一切旧俗。这种打破旧框框、适配新格局的思想，是面向对象的精髓。以继承方式实现类的 Adapter 模式和以聚合方式实现对象的 Adapter 模式，各有千秋，各取所长。看来，把它叫做包装器一点也不为过。

典型应用：RCW（Runtime Callable Wrapper）在 COM Interop 中的应用。

模式本身还有很多的故事和细节，在《设计模式：可复用面向对象软件的基础》中总结了 23 种设计模式，其大致的分类如表 3-2 所示。

表 3-2 经典设计模式

类 别	模 式
创建型模式	工厂方法
	抽象工厂
	单例
	创建者
	原型模式
结构型模式	桥接
	适配器
	组合
	外观
	装饰
行为型模式	享元
	代理
	模板方法
	迭代器
	中介者
	职责链
	解释器
	命令
	观察者
	备忘录
	状态模式
	策略模式
	访问者

3.3.3 模式的建议

模式不是万能妙药，了解和熟悉模式的最终意义在于放下模式，此时无剑胜有剑，模式的建议如下：

- 不要拿着 GOF 的书，从头看到尾，对我们来说那是圣经也是字典，系统地学习其实意义不大，常来翻翻反而收获更丰。
- 在软件设计中体会设计模式，设计就是不断地由需求生成的重构。

- 结合.NET Framework 框架来学习设计模式在.NET 中的应用，对了解设计模式来说是一举两得的事，既体味了设计，又深谙了框架。
- 重构、不断地重构。
- 切勿因模式而模式，任何模式的套用都意味着实现的复杂升级和执行的性能损耗。

3.3.4 结论

仁者见仁。以上只是笔者一家之言，更重要的真知灼见皆来源于实践，设计思想和模式的应用也来源于不断的学习和反复。此文只是开端，未来需要不断的探索。

3.4 面向对象和基于对象

本节将介绍以下内容：

- 基于对象的澄清
- 面向对象的差别

3.4.1 引言

这是一个常常被问起的话题，对于面向对象（Object-Oriented）我们可能有清晰的概念，对于基于对象（Object-Based）我们可能有模糊的认知，而对于二者一词之差的细节，又有多少概念值得深究呢？

关于面向对象的论述，本书第1章“OO的智慧”已经有相对全面的介绍，对于继承、封装和多态这些基本要素，已经有了较为深入浅出的了解，基于如此背景，就先从关注基于对象开始。

3.4.2 基于对象

所谓基于对象，就是一种对数据类型的抽象，封装一个结构包含了数据和函数，然后以对象为目标进行操作。

构建的基础是对象，但是操作对象并不体现出面向对象的继承性，也就是说基于对象局限了通过对象模板产生对象的福利。基于对象，不具有继承特性，也就更无所谓多态，但是对象本身的封装性仍然作为很多技术的基础，例如可以设置属性、调用方法，基于对象的语言特征就是将属性或者方法，包含在以对象为结构的组织中。然而，并不能通过“继承”访问父类对象的属性、方法，这是二者本质的区别所在。

从运行时的角度来看，基于对象的操作可以在编译时确定，不需要虚分派机制的额外消耗，但是必然少了多态带来的类型决断在运行时的灵活性。

3.4.3 二者的差别

面向对象与基于对象，其核心的差别是对于继承的支持。例如C#或者C++是面向对象语言，提供了对继

承的原生态支持；而 VB 和 JavaScript 是基于对象语言，以 JavaScript 为例，其 Prototype-Based 特性实现了以函数为单元的基于对象表现。

举个例子，我们基本认为 C# 是面向对象的语言，而 JavaScript 是基于对象的语言。在 C# 中封装、继承和多态构成了面向对象丰富体验的理论基础；而 JavaScript 中虽然只有简单的几种基本类型，但是对象这一基本概念还是成就了 JavaScript 的无限灵活性。但是，JavaScript 却不是纯粹的面向对象语言，可以以对象进行数据的操作，但是它没有继承和多态带来的面向对象体验。所以面向对象和基于对象的分水岭就定格于此。

关于 OO 的核心和抽象，广义上的核心就是“面向抽象，封装变化”这一基本思想，而狭义上的核心就是多态。

那么，抽象算是什么呢？在 3.2 节“依赖的哲学”中，已经有相当全面的阐释。所谓抽象，代表了软件系统中相对稳定的东西，依赖于稳定的因素可以使得整个系统的耦合度降低，因为稳定就是不变或者不易变。因为不变，所以永恒。架构在永恒之上的东西，正是软件设计理想的交互作用，不过这种理想无法在现实中存在，只能无限地接近，这个被接近的东西就是：抽象。

总结而言，面向对象与基于对象，二者的概念主要体现在：

- 继承是区别面向对象与基于对象的核心所在，对于少了继承性的基于对象来说，自然少了多态性支持。
- 封装是面向对象与基于对象的共同特征。

3.4.4 结论

关于面向对象与基于对象，存在概念上的不清晰界定，在面向对象概念大行其道的今天，本没有特别的意义对二者的差别进行特别的了解。然而，透过对象演义的历史，更能帮助我们理解对象本身被赋予的使命。

简而言之：面向对象基于对象而设计，基于对象面向对象而操作。

3.5 也谈.NET 闭包

本节将介绍以下内容：

- 闭包的简介
- .NET 中的闭包应用

3.5.1 引言

闭包，广泛存在于函数式编程语言的概念中，很多高级语言例如 Smalltalk、JavaScript、Ruby 还有 Python 对闭包都有或多或少的支持。因此，在.NET 平台中，对闭包的支持也不能例外，本文就以此为话题，探讨相关的内容。

3.5.2 什么是闭包

本质上，闭包源自数据概念。在编程语言领域，闭包的概念主要是指由函数以及与函数相关的上下文环

境组合而成的实体。通过闭包，函数与其上下文变量（又被称为自由变量，表示局部变量之外的变量）之间建立起关联关系，上下文变量的状态可以在函数的多次调用过程中持久保持。从作用域的角度而言，私有变量的生存期被延长，函数调用所生成的值在下次调用时仍被保持。从安全性的角度而言，闭包有利于信息的隐蔽，私有变量只在该函数内可见。

在 JavaScript 语言中，闭包无处不在。在官方概念中，一个拥有变量和绑定了该变量的表达式，就形成闭包。简单地说，在 JavaScript 中函数内部的子函数将自然形成一个闭包：

```
<script language="javascript" type="text/javascript">

    function f(){
        x = "Hello, Closure.";

        function fx(){
            alert(x);
        }

        return fx;
    }

    var r = f();
    r();
</script>
```

闭包体现在 JavaScript 中，带来的好处同样是对变量的封装和隐蔽，同时将变量的值保存在内存中。同样的情况，也可以发生在.NET。

3.5.3 .NET 也有闭包

在.NET 中，函数并不是第一级成员，所以并不能像 JavaScript 那样通过在函数中内嵌子函数的方式实现闭包，通常而言，形成闭包有一些值得总结的非必要条件：

- 嵌套定义的函数。
- 匿名函数。
- 将函数作为参数或者返回值。

在.NET 中，可以通过匿名委托形成闭包：

```
delegate void MessageDelegate();

static void Main(string[] args)
{
    string value = "Hello, Closure.";

    MessageDelegate message = delegate()
    {
        Show(value);
    };

    message();
}

private static void Show(string message)
```

```

        Console.WriteLine(message);
    }

```

事实上，大部分支持闭包的高级语言中，函数都是第一级成员，函数可以作为参数传递，也可以作为返回值返回，或者作为函数变量。而在.NET中，这一切都可以通过委托来实现，关于委托的详情请参考9.7节“一脉相承：委托、匿名方法和Lambda表达式”，所以上述逻辑也可以通过Lambda表达式实现更简单的代码。

反编译上述示例为IL代码：

```

.class private auto ansi beforefieldinit Program
    extends [mscorlib]System.Object
{
    .method private hidebysig static void Main(string[] args) cil managed
    {
        // 省略
    }

    .class auto ansi sealed nested private beforefieldinit <>c__DisplayClass1
        extends [mscorlib]System.Object
    {
        .custom instance void [mscorlib]System.Runtime.CompilerServices.CompilerGeneratedAttribute::ctor()
        .method public hidebysig specialname rtspecialname instance void .ctor() cil managed
        {
            // 省略
        }

        .method public hidebysig instance void <Main>b__0() cil managed
        {
            // 省略
        }

        .field public string value
    }
}

```

通过匿名方法将自动形成闭包，自由变量 value 被包装在一个内部类（闭包）中，并升级为实例成员，即使创建该变量的方法执行结束，该变量也不会释放，而是在所有回调函数执行之后才被GC回收。自由变量 value 的生命周期被延长，并不局限为一个局部变量。生命周期的延迟，是闭包带来的福利，但是也往往带来潜在的问题，造成更多的消耗。

1. 闭包与函数

像对象一样的操作函数，是闭包发挥的最大作用，从而实现了模块化的编程方式。不过，闭包与函数并不是一回事儿。

- 闭包是函数与其引用环境组合而成的实体。不同的引用环境和相同的函数可以组合产生不同的闭包实例。
- 函数是一段可执行的代码体，在运行时不会由于上下文环境发生变化。

2. 应用闭包

闭包，是函数式编程的精灵，在.NET平台中，这个精灵同样带来诸多方面的应用，典型的表现主要体现在以下几方面。

- 定义控制结构，实现模块化应用。闭包实现了以最简单的方式开发粒度最小的模块应用，实现一定程度的算法复用，下例的 ForEach 为遍历数组元素提供了复用基础，对于加法运算和减法运算而言，在闭包中改变引用环境变量的值，达到最小粒度的模块控制效果。

```
static void Main()
{
    int[] values = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};

    int result1 = 0;
    int result2 = 100;

    values.ForEach(x => result1 += x);
    values.ForEach(x => result2 -= x);

    Console.WriteLine(result1);
    Console.WriteLine(result2);
}
```

- 多个函数共享相同的上下文环境，进而实现通过上下文变量达到数据交流的作用。

```
static void Main()
{
    int value = 100;

    IList<Func<int>> funcs = new List<Func<int>>();
    funcs.Add(() => value + 1);
    funcs.Add(() => value - 2);

    foreach (var f in funcs)
    {
        value = f();
        Console.WriteLine(value);
    }
}
```

数据共享为不同函数的操作间传递数据带来方便，但是这把双刃剑有时又为不需要共享数据的场合带来问题，以上例而言，value 变量将在不同的操作中() => value + 1 和() => value - 2 间共享数据。如果不希望在两次操作间传递数据，需要特别注意引入中间量协调：

```
static void Main()
{
    int value = 100;

    IList<Func<int>> funcs = new List<Func<int>>();
    funcs.Add(() => value + 1);
    funcs.Add(() => value - 2);

    foreach (var f in funcs)
    {
        int v = f();
        Console.WriteLine(v);
    }
}
```

3.5.4 福利与问题

闭包收获了必然的福利，也不免带来些细问题，简单总结一下主要包括以下两方面。

首先，是福利：

- 代码简化，应用闭包可以实现一定程度的模块化复用，大大简化了代码执行的逻辑。
- 数据共享与延迟。
- 安全性。闭包的场合，有利于上下文信息的封闭性，实现了一定程度的信息隐蔽。

然后，看问题：

- 闭包有一定的优势，同时也带来一定的问题。应用闭包，将不可避免地使程序逻辑变得复杂。
- 很多时候，闭包的延迟特性会带来一定的逻辑问题，例如：

```
static void Main()
{
    IList<Action> actions = new List<Action>();

    for (int i = 1; i < 5; i++)
    {
        actions.Add(() => Console.WriteLine(i));
    }

    actions.ForEach(x => x());
}
```

上述示例如果期望在循环中遍历输出 1、2、3、4、5 这样的值，就会因为闭包的数据共享而产生问题，通常的解决办法是在循环中应用中间量：

```
for (int i = 1; i < 5; i++)
{
    int v = i;
    actions.Add(() => Console.WriteLine(v));
}
```

当然，上述问题并不能将责任一味地归咎于闭包，对于开发者而言，更懂得闭包，才能长袖善舞，否则只会弄巧成拙。

3.5.5 结论

闭包是优雅的，带来代码格局的函数式体验；但闭包也是复杂的，带来潜在的某些问题。它就像一把双刃剑，用好闭包的关键，在于深入地理解闭包，即在于挥剑人自己。

3.6 好代码和坏代码

本节将介绍以下内容：

- 编码的规范
- 面向对象指导

3.6.1 引言

好的代码，是练出来的。坏的代码，是惯出来的。

那么，代码是写给计算机的吗？不是，代码其实是写给人的。Martin Fowler 说：任何一个傻瓜都可以写出计算机可以理解的代码。唯有写出人类容易理解的代码，才是优秀程序员。那么，本文要探讨的其实是写出给人看的好代码，不涉及具体的代码技巧，只关注泛化的代码实践，通过一系列条款来过滤应该关注的好代码和坏代码。

3.6.2 好代码、坏代码

1. 命名很重要，让代码告诉你它自己

命名到底有多重要呢？

重要到这几乎是很多软件项目成功或者失败的“罪魁祸首”，究其原因，代码不光支撑了 0 和 1 在计算机系统中运行的业务逻辑，同时也是开发者进行交流与研究的标准语言。没有意义或者有歧义的命名，就像两个等待交流的人，面对了一堆火星文无从下口，让交流变成灾难，也就导致很多问题。

同时，好的命名是自说明的，让代码告诉开发者“我是谁，我做什么，我怎么做”。当然，除了静态式的必要的注释说明之外，动态式的代码也可以包含传递信息的作用，让代码告诉你它自己，因为代码是“活的代码”。

例如，以某个缓存容器为例，泛型参数明确了容器的 Key 和 Value 的关系，其中的方法也基本明确了作为缓存容器所具有的方法：Add、Set、Clear、Refresh 和 IsExist，而 TryGetValue 是 Try-Parse 模式的表现。其中的变量 container 表示了容器载体；expiration 表示了过期时间；config 表示了容器的配置信息。

```
public class AtCache<TKey, TValue>
{
    public int Count{ }
    public List<TValue> Items{ }
    public int Expiration { }

    public void Add(TKey key, TValue value){ }
    public void Set(TKey key, TValue value, int expiry){}
    public bool TryGetValue(TKey key, out TValue value){}
    public void Clear(){}
    public bool IsExist(TKey key){}
    protected void Refresh(){}

    private ReaderWriterLockSlim rwLocker = new ReaderWriterLockSlim();
    private Dictionary<TKey, CacheItem<TKey, TValue>> container = new Dictionary<TKey, CacheItem<TKey, TValue>>();
    private int expiration;
    private DateTime lastRefresh = DateTime.Now;
    private IAtCacheConfiguration config;
    private List<TValue> items;
}
```

总体来说，让代码告诉它自己，是好代码的体现，而一堆没有意义的代码堆积是让人无法接受和容忍的坏代码。

2. 遵守编码规范

编码规范，就是编码最佳实践，是前辈在编码这件事上的积累和总结，是智慧的延续和工业的实践。在软件产业日益蓬勃的今天，软件工业在于如何更有效率地进行生产这件事儿上，有了巨大的进步和积累，编

码规范正是如此。例如可以随意列出很多的规范：

- 命名规范。
- 避免行数过多的方法。
- 代码缩进。
- 异常规范。
- 设计规范。
- 注释规范。
- 文件的组织规范。
- 配置规范。
- 发布与部署规范。
- 测试规范。
- SQL 规范。

在以上每个领域都有 N 条“法规”，以最佳实践的条款被总结出来，每个条款都渗透着很多前人的智慧。同时，编码规范的应用是有选择和场合的，不同的软件公司和产品，对编码规范都有一定的理解和取舍。

但是，没有规范的编码，一定是有问题、潜伏着坏代码的幽灵。

3. 遵守命名规则

命名已经被反复强调了，遵守编码规范首当其冲就是对于命名规范的遵守，对于命名规则，通常可选择的体系主要有：

- Pascal Casing，混合使用大小写字母，每个单词的首字母必须是大写，例如 FirstName。
- Camel Casing，混合使用大小写字母，第一个单词的首字母是小写，其他单词的首字母是大写，例如 firstName。
- 匈牙利命名法，通过属性、类型和对象描述混合来表示，例如 frmMainWindow，表示一个窗体实例的命名。

不过，对于不同的语言体系而言，一般有着不同的命名规范和体系，很多不同的语言对于命名规范的选择也有差别。以 C# 语言为例，最基本的命名规则包括：

- 以 Pascal Casing 风格定义命名空间、类及其成员、接口、方法、事件、枚举等。
- 以 Camel Casing 规范定义参数、私有成员。
- 避免使用匈牙利命名法。
- 以 Attribute 作为特性的后缀。
- 以 Delegate 作为委托的后缀。
- 以 Exception 作为异常的后缀。

当然，规范还有很多，而这种积累来自于平时对于代码的理解和运用。

4. 多注释，少废话

代码，一定是给人看的，而代码本身的逻辑又决定于方法、类型和依赖的关系之中，所以，必要的注释，是必需且必要的。通过注释的进一步解释，来辅助性地告知代码的逻辑、算法或者流程，不仅是好习惯，更

是好代码。

另一方面，注释不是“无病呻吟”，没有必要表述那些显而易见的逻辑或者说明，同时注意区分单行注释和多行注释的应用。

在.NET平台下，XML格式的注释还肩负了另一项重要的使命，那就是根据注释生成代码文档。例如：

```
/// <summary>
/// 根据用户信息，构建标签信息
/// </summary>
/// <param name="memberId">用户 Id，根据用户 Id，获取<see cref="Member"/>的实例信息</param>
/// <param name="tag">标签信息</param>
/// <returns>标签信息对象</returns>
public Tag BuildTag(int memberId, string tag)
{
    return new Tag();
}
```

在Visual Studio中，可以通过选择Properties→Build来设置“XML documentation files”选项输出生成XML信息，例如上面的注释信息被生成为：

```
<?xml version="1.0"?>
<doc>
    <assembly>
        <name>Anytao.Inside.Ch03.GoodCode</name>
    </assembly>
    <members>
        <member name="M:Anytao.Inside.Ch03.GoodCode.Tag.BuildTag(System.Int32, System.String)">
            <summary>
                根据用户信息，构建标签信息
            </summary>
            <param name="memberId">用户 Id，根据用户 Id，获取<see cref="T:Anytao.Inside.Ch03.GoodCode.Member"/>的实例信息</param>
            <param name="tag">标签信息</param>
            <returns>标签信息对象</returns>
        </member>
    </members>
</doc>
```

通过SandCastle工具就可以基于上述信息生成标准统一的文档信息，基于此方式就可以建立类似于MSDN文档的项目帮助文件，大大简化了这项“复杂”的工作。

5. 用命名空间组织你的代码

命名空间，是逻辑上的组织单元，通过命名空间建立对代码的有机组织，是现代语言的一大“创举”，《Java夜未眠》作者蔡学镛说：一个语言是否适合大型开发，可以从它对模块、命名空间（或类似概念）支持的良窳看出端倪。从这个意义上说，命名空间并不是大型开发或者团队开发最重要的核心概念，但却是加分的必要因素。

关于.NET命名空间的详细内容，请参考7.3节“using的多重身份”。

6. 切勿模式而模式

设计模式是好的，而滥用模式是不好的。

了解和熟悉设计模式，是需要实践和思考的过程，模式并不是一切问题的灵丹妙药，而且大多时候的滥用反而造成更多的问题。滥用模式体现在两个方面：

- 不慎误用，在不合适的场合应用不合适的模式，例如不是所有的场合都需要引入工厂解耦对象创建；对于依赖于执行状态的场合，并非只有状态模式一种选择，工作流或许能带来更好的控制。
- 过度应用，模式的引入都会或多或少地介入了中间层或者中间代码，过度的模式应用将导致代码复杂度的直线上升，除了会带来性能上的问题还有逻辑上的混乱。

举一个简单的例子，策略模式是将算法从宿主类中剥离出来，将易于变化的部分封装为接口，例如：

```
public interface ITax
{
    decimal Calculate(decimal value);
}

public class FoodTax : ITax
{
    public decimal Calculate(decimal value)
    {
        return new decimal(1 + 0.15) * value;
    }
}

public class RetailTax : ITax
{
    public decimal Calculate(decimal value)
    {
        return new decimal(1 + 0.1) * value;
    }
}
```

对于算法分离而言，通过 ITax 策略可以很好地进行不同行业（例如饮食 FoodTax 或者零售 RetailTax）税率的计算，不同的行业提供不同的算法策略，然而对于变化的税率而言，这种实现的方式略显过度，越来越多的算法策略将造成代码的过度膨胀。所以完全可以对策略的方式进行改良，利用委托将税率算法分离看起来更加简洁而优雅：

```
public interface ITax
{
    decimal Calculate(Func<decimal> rateProvider, decimal value);
}

public class Tax : ITax
{
    public decimal Calculate(Func<decimal> rateProvider, decimal value)
    {
        var rate = rateProvider.Invoke();
        return rate * value;
    }
}
```

一下子清爽了很多，避免了“策略”带来过度膨胀，又很好地解决了税率算法的变化与分离，对于客户端的消费并没有太大的差别。

《倚天屠龙记》中有一个重要的片段，张三丰指点张无忌修炼太极，有一段“此时无招胜有招”的精彩论述，武术上真正的无敌不在乎一招一式的死记硬背，也不在于一刀一剑的激情挥洒。同样的道理，似乎更适

合用于软件设计与模式，很多时候，架构与设计的极致不在于对模式的“应用”，而在于对模式的“活用”，在于灵魂附体，在于无招胜有招。

7. 线程安全很重要

线程安全是重要的，在数据共享或同步的场合应将线程安全作为必须考虑的因素，不安全的代码将在多线程运行时造成严重的问题。例如，单例模式就是这样一个需要特别注意的例子：

```
public sealed class Singleton
{
    Singleton() { }

    public static Singleton Instance
    {
        get
        {
            if (instance == null)
            {
                instance = new Singleton();
            }

            return instance;
        }
    }

    private static Singleton instance = null;
}
```

因此，你可以考虑通过“双锁”机制来保证线程的安全，不过在.NET平台还可以有更简单的实现方式：

```
public sealed class Singleton
{
    static Singleton() { }

    Singleton() { }

    public static Singleton Instance
    {
        get
        {
            return instance;
        }
    }

    static readonly Singleton instance = new Singleton();
}
```

利用静态构造函数只能被执行一次且在运行库加载类成员时的特点，保证了 `instance` 的线程安全，避免了不必要的锁检查开销。关于静态构造函数，详见 8.8 节“动静之间：静态和非静态”。

线程安全是个大课题，需要仔细咀嚼。

8. 不断地重构和思考

软件开发就像爬山，而有意思的事情在于，我们爬的并不是一座山，而是一座又一座的山，似乎永无尽头。所以爬山的过程其实是这样，爬上了一座，又从这一座下来，然后接着爬向一座，并且继续如此反复，才能到最高的巅峰。

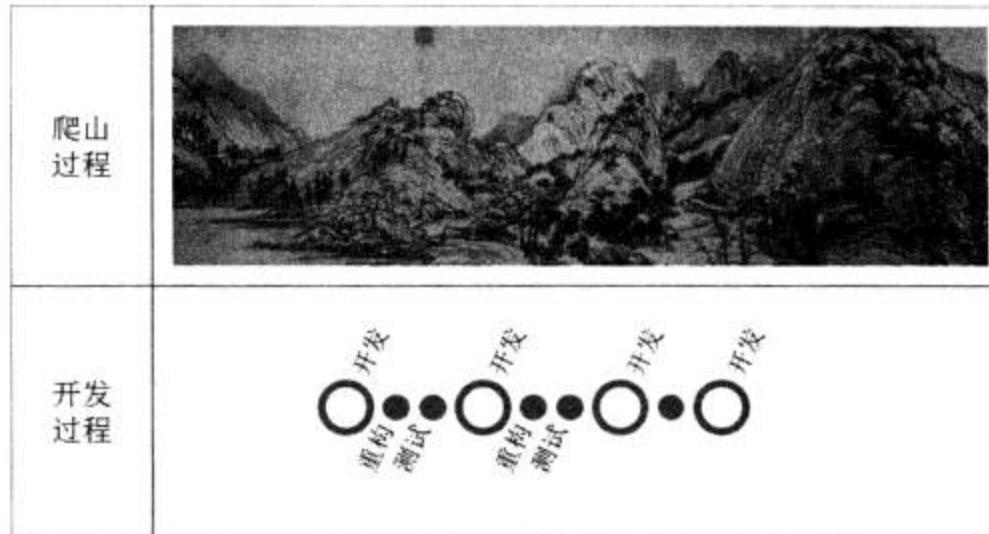


图 3-15 软件开发的爬山模型（图片来源：百度百科，上图为富春山居图部分截图）

所以，可以把软件开发中这种不断重构和完善的过程，叫作爬山模型（图 3-15）。爬山模型的重点在于只有通过不断地重构和演进，才能不断地完善和进步，并最终达到软件产品的高峰。

代码重构是个系统工程，有很多值得借鉴的方法，在《Refactoring: Improving the Design of Existing Code》一书中有详细的讨论：

- 以单元测试驱动。
- 提取类、方法、接口或者子类等。
- 重新组织数据。
- 简化函数调用。
- 借助重构工具。

纵观本书，也从很多方面对于重构提供了思考和实践。

9. 扩展无处不在

扩展性是衡量一个软件产品的重要尺度之一。通过合适的设计为软件系统赋予一定程度的扩展，是架构师着手设计的重要考虑因素，如图 3-16 所示。



图 3-16 架构的考量

扩展是个大课题，涉及软件系统的方方面面，依赖于粒度不同的架构格局。举例来说，数据库设计可以

考虑在横向或纵向的扩展、在多层架构中实现可适配的数据层、为业务层实现注入逻辑设计、在 UI 层提供可配置的界面选择以及为物理架构提供横向扩展的部署设计。实现基于服务的系统，就意味着在服务层支持扩展良好的高层架构；而一个面向接口的设计，将是为扩展提供可能的选择之一；采用 ASP.NET MVC 构建的 Web 系统，将在很多方面被赋予扩展的标签，基于管线模型的设计将扩展深入到几乎所有的方面，例如 ActionFilter、ViewEngine、Route、HtmlHelper、ModelBinder 以及 Controller，开发者可以轻易地替换所有原有支持元素，扩展出不同的“个性”功能；而 MEF（Managed Extensibility Framework）则实现了更灵活的扩展设计，基于 MEF 可以发现并使用扩展，甚至在应用程序之间重用扩展。

在语言层面，考量扩展性的指标遍布于.NET 语言特性的各个细节：

- 基于类的继承、组合和多态。面向对象的基本特征就是扩展良好，而.NET 的面向对象特性，在本书第 1 章“OO 大智慧”已经有了详细的讨论。
- 面向接口和抽象类，接口和抽象类是语言层的抽象载体，而面向抽象编程的设计原则在实际编码中的应用之一就是面向接口和抽象类，详见 8.4 节“面向抽象编程：接口和抽象类”。
- 基于委托和事件的回调。回调是一种扩展良好的实现机制，提供动态扩展性表现，使得框架能够以委托来调用用户代码：

```
private void btnLogin_Click(object sender, RoutedEventArgs e)
{
    MessageBox.Show("Hello, Windows Phone.");
}
```

就像给框架提供了一个“钩子”来动态地将用户代码扩展到框架的逻辑，在单击按钮的时候，执行用户代码的流程逻辑，并将这个流程注入到框架行为中。在.NET 中，可以通过委托实现线程的安全回调，而事件正是这种模式的最佳实践，详情参考 9.7 节“一脉相承：委托、匿名方法和 Lambda 表达式”。

- 应用扩展方法。扩展方法，本身就是为扩展而生的，详见 13.2 节“赏析 C# 3.0”。
- 以部分类延伸类的组织，在很多情况下，为了便于组织和物理上的方便，将一个类分布在多个独立的文件，是一种合适的处理方式；另一方面，对于越来越多的自动生成代码情况，部分类提供了“手动”扩展支持。例如，应用 LINQ to SQL 作为数据访问层框架时，通常可以通过 Visual Studio 自动生成实体类：

```
[global::System.Data.Linq.Mapping.TableAttribute(Name="dbo.Users")]
public partial class User : INotifyPropertyChanging, INotifyPropertyChanged
{
}
```

在这种情况下，就可以考虑通过部分类的方式，为实体类 User 增加新的成员、继承统一的基类：

```
public partial class User : EntityBase
{
    public bool IsAdmin { get; set; }
}
```

- 通过反射注入。反射特性是.NET 平台非常有吸引力的语法游戏，通过反射可以实现动态注入设计，在 3.2 节“依赖的哲学”中有详细的讨论。
- 基于 DLR 实现动态扩展。在.NET 4.0 中，巨大的变革即是动态编程，为静态语言插上动态的翅膀，让动态扩展无处不在，详见 14.3 节“动态变革：dynamic”。
- 让扩展可配置。在 ASP.NET 整体架构中，将 Web 请求的处理设计为管道模型，模型中的重要元素包

括 `HttpApplication`、`HttpModule` 和 `HttpHandler` 等，而对于这些管道中的过滤器（`HttpModule`）和处理器（`HttpHandler`）则通过配置实现可插拔的扩展性设计：

```
<httpModules>
  <add name="TimeLogModule" type="Anytao.Devkit.Core.Web.Modules.TimeLogModule, Anytao.Devkit.Core" />
</httpModules>
```

例如，上述配置可以将 `TimeLogModule` 注入到 HTTP 管道，从而将每个请求的处理时间输出到日志。

```
public class TimeLogModule : IHttpModule
{
    public void Dispose()
    {
    }

    public void Init(HttpApplication context)
    {
        context.BeginRequest += (sender, e) =>
        {
            var sw = new Stopwatch();
            HttpContext.Current.Items["StopWatch"] = sw;
            sw.Start();
        };
        context.EndRequest += (sender, e) =>
        {
            var sw = (Stopwatch)HttpContext.Current.Items["StopWatch"];
            sw.Stop();
            TimeSpan ts = sw.Elapsed;
            string result = string.Format("{0}ms", ts.TotalMilliseconds);
            Logger.Log(result);
        };
    }
}
```

- 基于 `ConfigurationManager` 的配置扩展。一般来说，配置是为扩展而准备的，而扩展可通过配置注入。在.NET 框架中提供非常优秀的配置支持，开发者完全可以通过这套完美的配置框架实现自定义的配置扩展。
- 硬编码总是不好的。任何时候都尽可能将变化的部分从代码中分离，以配置或者其他方式加载，为扩展提供机会。

扩展无处不在。软件设计师的职责，在于将这种无处不在深入到软件系统的各个环节，为各种可能提供基础与准备。

10. 性能是一把尺子

性能，永远是任何软件产品衡量的标准，就像一把标准的千分尺，可以精度准确地为产品打上分数，在.NET 中性能的指标体现在语言的各个方面，在本书 6.4 节“性能优化的多方探讨”中，对于性能的问题有详细讨论。

11. 信赖的是测试，不是自己

质量的保证，一直是复杂的软件开发的软肋，为了保证软件产品的完美，测试是整个开发流程中最重要的部分。现代软件开发也衍生出很科学的测试方法、方式和制度，不管是黑盒的还是白盒的，只要逮住 Bug，就是好测试。

与传统测试方式比较，测试驱动开发（Test Driven Development, TDD）已经被证明是非常靠谱和科学的开发方式。TDD 至少在两个方面为软件开发注入活力：

- 保证质量。足够的覆盖率将能保证软件质量和稳定性，系统的修改和变化将第一时间反馈在测试代码上，结果驱动的方式将能最大限度地评估变化对原有系统造成的影响，从而保证业务代码的正确性。
- 驱动设计。另一方面，为了能够让业务代码具有可测试性，你应重新审视业务代码的设计，就像 Bob 大叔的名言：编写单元测试更像一种设计行为，文档行为而不是验证行为。编写单元测试缩短了反馈周期读数，最小读数基于功能验证。

因此，测试驱动是值得提倡和普及的，将由人的信任测试，转变为由代码的信任测试，信任的是测试，而不是开发者自己。

12. 是进度还是质量，平衡是关键

开发者经常挂在嘴边的一句话是：给我足够的时间，我将实现得更好。然而，实际的情况往往是，开发的周期和开发的进度总是存在着冲突，进而带来进度和质量之间的妥协，而妥协的关键在于平衡。

作为开发者而言，需要评估设计和实现所花费的时间，然后根据评估的结果对进度做以平衡，很多时候，并没有一次就很完美的设计，只有当下适合的设计。平衡进度与质量的关键，是建立起行之有效的开发流程和进度计划，将资源、进度和质量有机地整合在可控制的框架管理中，并准备好三个要素之间的缓冲带，在适合的时候做好调整的准备。

3.6.3 结论

破的窗，将导致更多的窗户被打破，是《程序员修炼之道》一书阐释的“破窗效应”。而亡羊补牢，未为晚也，养成好的代码习惯和意识，学会独立地思考和重构，远远重要于在破的窗补破的局。

参考文献

Erich Gamma, Richard Helm, Ralph Johnson, John Vlisside, 设计模式：可复用面向对象软件的基础

Martin Flower, Closure, <http://www.martinfowler.com/bliki/Closure.html>

刘艺, Delphi 设计模式, 机械工业出版社

Judit Bishop, C# 3.0 Design Patterns, O'Reilly

Brad Abrams, Internal Coding Guidelines,

<http://blogs.msdn.com/b;brada/archive/2005/01/26/361363.aspx>



学习笔记

第2部分

本质——.NET 深入浅出

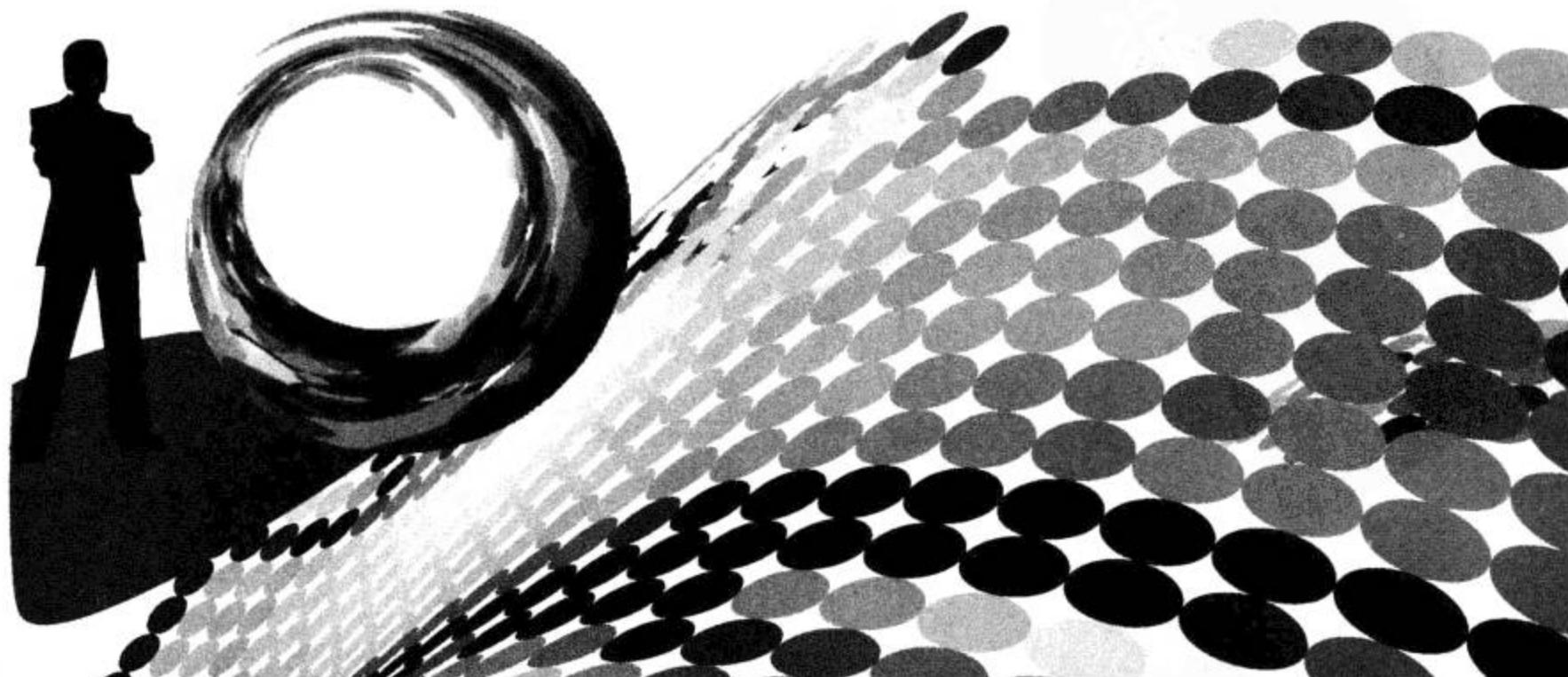
对任何技术的把握，我们都应该抓住核心追溯本质，才能更有效地吸收其精华。这正如高手过招，拳脚之间往往蕴含着内功的修为，不动声色即可手到擒来。对于.NET 技术来说，高手的代码往往简单而有效，不光实现了功能需求，更能体现性能要求。熟练掌握 IL 语言可以有效地了解技术背后的本质，为每个追求技术的人打开达到顶峰的大门；而类型系统与内存管理，将是提高技术修为最有效的内功，第一项修炼就从了解类型、通晓内存开始吧，否则，弯路不知还要走到什么时候。

本部分主要包括：

- 第4章 一切从 IL 开始
- 第5章 品味类型
- 第6章 内存天下

第4章 一切从 IL 开始

4.1 从 Hello, world 开始认识 IL / 109	4.4.3 继续深入 / 123
4.1.1 引言 / 109	4.4.4 元数据是什么 / 125
4.1.2 从 Hello, world 开始 / 109	4.4.5 IL 是什么 / 128
4.1.3 IL 体验中心 / 109	4.4.6 元数据和 IL 在 JIT 编译时 / 130
4.1.4 结论 / 113	4.4.7 结论 / 134
4.2 教你认识 IL 代码——从基础到工具 / 113	4.5 经典指令解析之实例创建 / 134
4.2.1 引言 / 113	4.5.1 引言 / 134
4.2.2 使用工具 / 113	4.5.2 newobj 和 initobj / 134
4.2.3 为何而探索 / 115	4.5.3 ldstr / 136
4.2.4 结论 / 116	4.5.4 newarr / 137
4.3 教你认识 IL 代码——IL 语言基础 / 116	4.5.5 结论 / 139
4.3.1 引言 / 116	4.6 经典指令解析之方法调度 / 140
4.3.2 变量的声明 / 116	4.6.1 引言 / 140
4.3.3 基本类型 / 117	4.6.2 方法调度简论: call、callvirt 和
4.3.4 基本运算 / 118	4.6.2 calli / 140
4.3.5 数据加载与保存 / 118	4.6.3 直接调度 / 142
4.3.6 流程控制 / 119	4.6.4 间接调度 / 146
4.3.7 结论 / 120	4.6.5 动态调度 / 147
4.4 管窥元数据和 IL / 120	4.6.6 结论 / 147
4.4.1 引言 / 120	参考文献 / 147
4.4.2 初次接触 / 120	



4.1 从Hello, world开始认识IL

本节将介绍以下内容：

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

4.1.1 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》，作者推荐所有的程序员有机会都要重温这本历史上的经典之作。从那时起，Hello, world示例就作为几乎所有实践型程序设计书籍的开篇代码，一直延续至今。除了表达对历史与巨人的尊重，本节也以Hello, world示例作为打开IL语言之门的钥匙，开始我们循序渐进的IL认识之旅。

4.1.2 从Hello, world开始

首先，当然是展示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的真相又是什么呢，下面开始我们基于上述示例的IL代码分析。

4.1.3 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为文本MSIL编码，至于其工作原理我们将在本章后续各节中做一个交代，首先查看其结构，如图4-1所示。

由图4-1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。

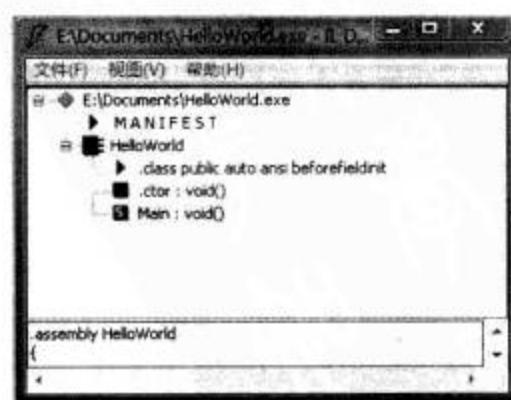


图4-1 HelloWorld的IL分析

1. MANIFEST 清单分析

打开 MANIFEST 清单，可以看到：

```
// Metadata version: v2.0.50727
.assembly extern mscorel
{
    .publickeytoken = (B7 7A 5C 56 19 34 E0 89 ) // .z\V.4..
    .ver 2:0:0:0
}
.assembly HelloWorld
{
    .custom instance void [mscorel]System.Runtime.CompilerServices.CompilationRelaxations-
Attribute::ctor(int32) = ( 01 00 08 00 00 00 00 00 )
    .custom instance void [mscorel]System.Runtime.CompilerServices.RuntimeCompatibility-
Attribute::ctor() = ( 01 00 01 00 54 02 16 57 72 61 70 4E 6F 6E 45 78
                        // ....T..WrapNonEx

63 65 70 74 69 6F 6E 54 68 72 6F 77 73 01 ) // captionThrows.
    .hash algorithm 0x00008004
    .ver 0:0:0:0
}
.module HelloWorld.exe
// MVID: {4E594BE7-8736-4520-8BD0-FC43F60E2FBA}
.imagebase 0x00400000
.file alignment 0x000000200
.stackreserve 0x00100000
.subsystem 0x0003 // WINDOWS_CUI
.corflags 0x00000001 // ILOONLY
// Image base: 0x01080000
```

从这段 IL 代码中，可以得出以下的分析：

- .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 mscorel 表示外部加载了外部核心库 mscorel，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，例如上例中的 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 mscorel.dll 程序集定义了 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 HelloWorld 类提供任何显式的构造函数，因此可以肯定其调用基类 System.Object，而这个 System.Object 就包含在 mscorel 程序集中。
- 在外部指令中还指明了引用版本(.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（如图 4-2 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。

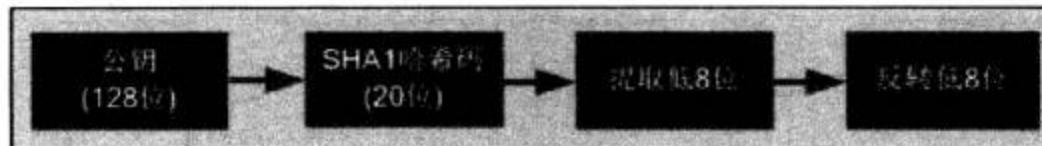


图 4-2 公钥标记的获取

- 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。

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 是一个类，该类继承自外部程序集 mscorelib 的 System.Object 类。
- public 为访问控制权限，表示具有最高的访问权限，对访问成员没有限制。
- 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托管和被托管代码间实现无缝转换。没有被托管的代码，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，详细的情况可以参考 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_ 标记之前的部分为变量的声明和初始化。
- ldarg.0 (ldarg 即 load argument) 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，而在此处 call 并非调用静态方法，而是构造函数.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。关于 call 和 callvirt 的更详细论述详见 4.6 节“经典指令解析之方法调度”。
- ret (ret 即 return) 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 (ldstr 即 load String) 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 7.1 节“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。
- 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. 回归简洁

去粗取精，其中的 IL 代码可以简化，下面的代码是基于上面的分析，并去除不重要的信息，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。

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

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

        //调用父类构造函数
        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
    }
}
```

4.1.4 结论

结束本节，我们从一个点的角度和 IL 进行了一次接触，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。通过一站式地扫描 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不太陌生，而且随着本章的深入我们将逐渐建立起这种认知，从而提高我们掌握和了解这一.NET 底层研究的有效工具。事实上，以 IL 作为研究.NET 本质的有效工具，将伴随在本书的所有角落，我们以 IL 透视.NET 的各个基本知识，就会看得更清，走得更近，想得更深。

4.2 教你认识 IL 代码——从基础到工具

本节将介绍以下内容：

- IL 代码的分析方法
- IL 工具介绍
- IL 分析的意义

4.2.1 引言

回头看，从项目的实践中我发现很多人把目光和焦点集中在如何理解 IL 代码这个问题上，我们的思路慢慢地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。我认为不管是何种途径，了解和认识 IL 代码，对于我们更深刻地理解.NET 和.NET 应用之上的本质绝对大有裨益，这也就是本节研究和分享的理由。

4.2.2 使用工具

俗话说，工欲善其事，必先利其器。了解 IL 就要首先从其使用工具开始。在.NET 世界里有数个不同的 IL 工具，包括编译器和反编译器两个方面。最典型的编译与反编译利器就是.NET Framework 自带的 ILASM.exe 和 ILDASM.exe 工具，同时不得不提的杰出代表还有 Reflector.exe。这几个工具都是了解 IL 的基础，且看我一一道来。

- ILASM.exe

打开记事本输入如下代码：

```
.assembly HelloWorld{}
.assembly extern mscorelbi{}

.class IL_Tools extends [mscorelbi]System.Object
{
    .method public static void HelloWorld()
    {
        .maxstack 1
    }
}
```

```

ldstr "Hello, world"
call void [mscorlib]System.Console::WriteLine(string)
ret
}

.method public static void Main()
{
    .entrypoint
    .maxstack 1

    call void IL_Tools::HelloWorld()
    ret
}
}

```

然后另存为 E:\Documents\IL_Tools.il。我们使用 ILASM.exe 工具来执行 IL 代码，打开 SDK 命令提示行，输入如下命令：

```
ilasm.exe E:\Documents\IL_Tools.il
```

编译成功，即可生成 E:\Documents\IL_Tools.exe 程序，执行情况如图 4-3 所示。

- ILDASM.exe

打开.NET Framework SDK 命令提示行，输入 ildasm 回车即可打开，然后选择欲反编译的可执行文件，如图 4-4 所示。

图 4-4 是我们熟悉的 4.1 节“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，如图 4-5 所示。

- Reflector.exe

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.NET3.5，其功能也相当强大，在使用上也较 ILDASM 更加灵活，如图 4-6 所示。

The screenshot shows the Windows Command Prompt window titled 'SDK 命令提示符'. The command entered is 'ilasm.exe E:\Documents\IL_Tools.il'. The output shows the assembly process:

```

Setting environment to use Microsoft .NET Framework v2.0 SDK tools.
For a list of SDK tools, see the 'StartTools.htm' file in the bin folder.

D:\Program Files\Microsoft Visual Studio 8\SDK\v2.0\bin>ilasm.exe E:\Documents\IL_Tools.il

Microsoft (R) .NET Framework IL Assembler, Version 2.0.50727.1328
Copyright © Microsoft Corporation. All rights reserved.
Assembling 'E:\Documents\IL_Tools.il' to EXE      -> 'E:\Documents\IL_Tools.exe'
Source file is ANSI

Assembled method IL::call::HelloWorld
Assembled method IL::call::Main
Creating PE file

Emitting classes:
Class 1:    IL::call

Emitting fields and methods:
Global
Class 1 Methods: 2
Resolving local member ref: 1->0 defn, 0 refs, 0 unresolved

Emitting events and properties:
Global
Class 1
Resolving local member ref: 1->0 defn, 0 refs, 0 unresolved
Writing PE file
Operation completed successfully

D:\Program Files\Microsoft Visual Studio 8\SDK\v2.0>

```

图 4-3 编译 IL 代码

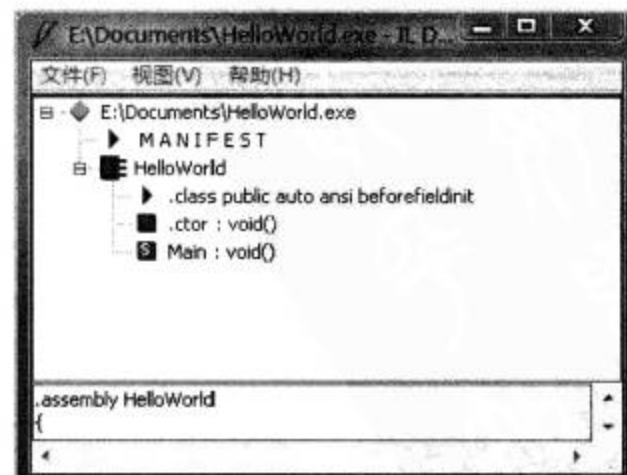


图 4-4 HelloWord 的 IL 示例

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 4-5 IL 树的图标含义（图表来源：MSDN）



图 4-6 Reflector 工具

Reflector 可以方便地将.NET 可执行文件反编译为 IL、C#、VB、Delphi 等多种形式语言，可以说在.NET 开源之前，Reflector 始终是研究.NET 框架最有效的工具之一。同时开源项目中还可以获得很多 Reflector 插件，支持更多的反编译操作，例如典型的有 CodeMetrics、Diff、Graph、CodeSearch 等，从而使其成为深入了解 IL 的最佳利器。

除特别说明外，在本书中所有的 IL 分析，均以最简单的 ILdasm.exe 工具为例来说明。

4.2.3 为何而探索

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成本。对于.NET 设计师或程序员来说，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，需要我们从底层入手来探索，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 来揭秘的，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。

IL 就像是打开.NET 内部机制的大门，我们没有理由放弃一条可以直接通达大门的便捷之路，而盲目地以其他的方式追求深入，无疑会先错过这扇门。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地来关注 IL，有人甚至可以洋洋洒洒地写一堆 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，纯粹陶醉在 IL 中而止步不前，同样可悲。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，找到了平衡的基线后，本章将展开对 IL 代码的认识，期望这些阐述与分析能使大家对 IL 有个概观了解，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。这种方法应该是值得在实践中提倡和发挥的，不是吗？

在本书中，我们会在每个可能的深入探索中提及 IL 语言，给.NET 及其本质更多的曝光机会，也留给我们更多的思考与认知。

4.2.4 结论

认识 IL 代码，是个循序渐进的过程，有了上节的 Hello, World 示例作为铺垫，我们至少可以轻松地认识简单的 IL 代码了。本节从 IL 分析工具的使用角度来介绍如何以有效的手段来帮助我们完成有效的 IL 分析。系统性地对 IL 语言做以全面的探讨，本节是个重要的开端。

4.3 教你认识 IL 代码——IL 语言基础

本节将介绍以下内容：

- IL 指令语言基础
- IL 指令使用方法

4.3.1 引言

在 4.1 节“从 Hello, world 开始认识 IL”中，我们以典型的 Hello, world 程序为例，探讨了 IL 代码的基本结构和大概面貌，了解了 IL 作为一种基于堆栈的面向对象语言的基本特征。第一次亲密接触的激情过后，我们应该静下心思来了解 IL 代码中最为典型的指令，从而逐步了解应该认识的 IL 代码。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，就可以基本上达到对 IL 了解而不过分的标准，因此对 IL 常用命令的分析就是本节的重点和要点。我们通过对常用命令的解释、示例与分析，逐步发现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 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，就像学习高级语言的关键字一样，我们只取其精华的一瓢来饮，由点及面来掌握重点的指令，详细的指令集解释请参见相关的《MSIL 指令手册》。

4.3.2 变量的声明

.locals 指令用于声明局部变量，例如

```
.method public hidebysig static void Main() cil managed
{
    .entrypoint
    // 代码大小      13 (0xd)
    .maxstack 1
    .locals init ([0] int32 i,
                 [1] string str,
                 [2] char c)
    IL_0000:  nop
    IL_0001:  ldc.i4.2
    IL_0002:  stloc.0
```

```

IL_0003: ldstr    "Hello"
IL_0008: stloc.1
IL_0009: ldc.i4.s 65
IL_000b: stloc.2
IL_000c: ret
} // end of method ILsetsTest::Main

```

在例中，.locals 为该方法声明了三个局部变量，分别是 int32 i，string str，char c。init 指令表示这几个变量应以对应的类型默认值进行初始化，通常情况下变量名是可以省略的，在代码中将以零基索引来指明变量。例如：

```

IL_0001: ldc.i4.2
IL_0002: stloc.0

```

IL_0001 语句表示将整数 2 加载到 Evaluation Stack，而 IL_0002 语句则表示将其保存到变量 0，也就是.locals 语句中的第一个变量 i，对应的 C# 代码为：

```
int i = 2;
```

依次类推，stloc.1 表示保存第二个变量 str，stloc.2 表示保存第三个变量 c。当然也可以直接以变量名来表示变量，如下：

```

IL_0001: ldc.i4.2
IL_0002: stloc.i

```

4.3.3 基本类型

类型始终是任何语言基础的起点，IL 语言也不例外，在探索 IL 世界的开始阶段，我们首先来熟悉 IL 的常见基本类型，表 4-1 将最基本的 IL 类型与.NET 基本类型做以比较。

表 4-1 常见的 IL 基本类型与.NET 基本类型比较

IL 类型	.NET 基本类型	CLS 兼容性
void		No
Bool	System.Boolean	No
Char	System.Char	No
int8	System.SByte	No
int16	System.Int16	No
int32	System.Int32	Yes
int64	System.Int64	Yes
native int	System.IntPtr	Yes
Float32	System.Single	No
Float64	System.Double	No
object	System.Object	Yes
Array	System.Array	Yes
string	System.String	Yes
typedef	System.Typed Reference	Yes
&		Yes
*	System.IntPtr	Yes

当然还包括例如 unsigned int8、unsigned int16、unsigned int32、unsigned int64、native unsigned int 等基本类型，详细内容请参阅相关资料。

4.3.4 基本运算

和基本类型一样，基本运算也是任何现代指令集语言必须提供的基本能力，在 IL 中同样提供了丰富的运算指令来完成基本的运算操作，主要包括：

- 算术运算：包括加法指令 add、减法指令 sub、乘法指令 mul、除法指令 div、求余指令 rem。
- 比较运算：包括大于指令 cgt、小于指令 clt 和等于指令 ceq。结果以 1 和 0 表示真假，并压入栈顶。例如，ceq 比较两个数值的大小，如果相等则将 1 装入栈中，否则将 0 装入栈中。
- 位运算：主要包括一元 not 指令，二元与指令 and、或指令 or、异或指令 xor，还有左移指令 shl 和右移指令 shr 等。

其他的操作还有很多，常见的运算指令还包括一些变体，以 add 指令为例：add.ovf 用于执行溢出检查，而 add.ovf.un 表示将两个无符号整数相加，然后执行溢出检查，并将结果压入堆栈。

下面的 IL 程序完成的是一个简单的加、减、乘运算，并将运算结果输出到控制台，我们只需关注其运行指令即可。

```
.method public hidebysig static void Main() cil managed
{
    .entrypoint
    // 代码大小      34 (0x22)
    .maxstack 2
    .locals init ([0] int32 x,
                 [1] int32 y)
    IL_0000:  nop
    IL_0001:  ldc.i4.s  10
    IL_0003:  stloc.0
    IL_0004:  ldc.i4.8
    IL_0005:  stloc.1
    IL_0006:  ldloc.0
    IL_0007:  ldloc.1
    IL_0008:  add
    IL_0009:  call      void [mscorlib]System.Console::WriteLine(int32)
    IL_000e:  nop
    IL_000f:  ldloc.0
    IL_0010:  ldloc.1
    IL_0011:  sub
    IL_0012:  call      void [mscorlib]System.Console::WriteLine(int32)
    IL_0017:  nop
    IL_0018:  ldloc.0
    IL_0019:  ldloc.1
    IL_001a:  mul
    IL_001b:  call      void [mscorlib]System.Console::WriteLine(int32)
    IL_0020:  nop
    IL_0021:  ret
} // end of method IL_Operation::Main
```

4.3.5 数据加载与保存

CLR 运行时，任何有意义的操作都是在堆栈上进行的，因此 IL 中有大量的指令用于完成数据的加载操

作，以便将要操作的数据放在栈上。通常情况下包括将数据保存到栈上和加载栈上数据两个操作。

IL中提供了完成这两种操作的指令，将变量从内存装载到堆栈上的操作指令通常以ld开头，而每一条加载指令都对应了一条存储指令，通常以st开头。例如ldarg对应于starg，而ldloc对应于stloc，不同的指令操作不同的数据类型，常见操作如表4-2所列。

表4-2 IL装载指令

IL装载指令	指令解析
ldarg/ldarga	ldarg 装载成员的一个参数，ldarga用于装载参数的地址
ldfld/ldsfld	用于将实例字段或者静态字段装载入堆栈。ldfld用于实例字段，而ldsfld用于静态字段
ldc	将数字常量装入堆栈。例如ldc.i4.5表示整数5作为一个4字节整数装入栈中
ldloc/ldloca	ldloc将一个局部变量装入堆栈，ldloca将一个局部变量的地址装入堆栈
Ldelem	表示装入数组元素到堆栈，通常要先于表示这个索引的其他装载语句之前使用
ldlen	表示装入数组的长度到堆栈
ldind	表示加载间接寻址，也就是以地址来访问和操作数据内容
ldstr	表示将字符串的引用加载到栈上

！注意

某些指令后会加上.s，例如ldc.i4.s、ldloca.s、ldarga.s，其中的.s表示只取用单个字节。

4.3.6 流程控制

流程控制是语言执行的基础，主要包括分支、条件和循环。IL中所有的流程控制皆利用分支指令来实现，主要包括br、br.s、brtrue和brfalse。

我们以简单的条件跳转语句为例，来简要说明IL分支跳转的执行过程：

```
.method public hidebysig static void Main() cil managed
{
    .entrypoint
    // 代码大小      40 (0x28)
    .maxstack 2
    .locals init ([0] int32 x,
                 [1] int32 y,
                 [2] bool CS$4$0000)
    IL_0000:  nop
    .....部分省略.....
    IL_000d:  stloc.2
    IL_000e:  ldloc.2
    IL_000f:  brtrue.s  IL_001a
    IL_0011:  ldloc.0
    IL_0012:  call       void [mscorlib]System.Console::WriteLine(int32)
    IL_0017:  nop
    IL_0018:  br.s     IL_0021
    IL_001a:  ldloc.1
    IL_001b:  call       void [mscorlib]System.Console::WriteLine(int32)
    IL_0020:  nop
```

```
IL_0021: call    int32 [mscorlib]System.Console::Read()
IL_0026: pop
IL_0027: ret
} // end of method IL_Loop::Main
```

从 IL 代码中可知,如果 brture 执行为真,则程序跳转到 IL_001a 执行,否则顺序执行,执行到 IL_0018 行指令 br.s 指示程序跳转到 IL_0021,跳出条件为假的代码段,从而实现条件分支的执行。

其他分支指令与此类似,每个分支指令都是从栈顶取出数据值按照某种方式比较,然后按照比较结果转移到目标单元执行,文本 IL 中使用标签来表示程序执行的偏移量,如本例中的 IL_001a 和 IL_0021,而实际上编译器中执行的是二进制 IL,因此偏移量同样为二进制值。

4.3.7 结论

本节从几个基本的 IL 指令开始,力求通过概念介绍和实例分析来逐步揭示 IL,正如我们在开始强调的那样,本节只是个开始,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。我们不断地从 IL 世界探求.NET 世界,这个主题在今后的讨论中还将渗透在本书的其他章节。

4.4 管窥元数据和 IL

本节将介绍以下内容:

- 深入了解元数据
- 深入解析 IL 和元数据的关系
- 元数据和 IL 在 JIT 编译时

4.4.1 引言

你可曾想到,我们的 C# (或者 VB.NET) 代码,编译之后究竟为何物?你可曾认知,我们的可执行程序 (*.exe),运行之时的轨迹究竟为哪般?那么,本文通过对 Metadata (元数据) 和 IL (Intermediate Language,中间语言) 的认识开始,来逐步给出答案。在这个探索轨迹上,元数据、IL、程序集、程序域、JIT、虚分派、方法表和托管堆这些形形色色的神秘嘉宾将在某个时刻不期而遇,我们借助于认识元数据和 IL 这两位重量级选手的过程,来顺道探究其他的嘉宾。

4.4.2 初次接触

事实上,编译之后的 cs 代码被组织为两种基本元素:元数据 (Metadata) 和中间语言 (IL)。我们可以以最简单的方式来了解程序集 (*.dll) 或可执行文件 (*.exe) 中包含的 Metadata 和 IL 的秘密,这种方式就是我们常说的反编译,打开 ILDasm 并加载准备的程序集,我们可以看到托管 PE 文件的相关内容,如图 4-7 所示。

详细的结构信息和 IL 代码分析,可以参见 4.3 节“教你认识 IL 代码——IL 语言基础”的介绍,在此就不做太多的分析。另外,我们可以通过执行“View/MetaInfo>Show!”命令或者按 Ctrl+M 组合键来获取该程

序集所使用的 Metadata 信息列表，如图 4-8 所示。

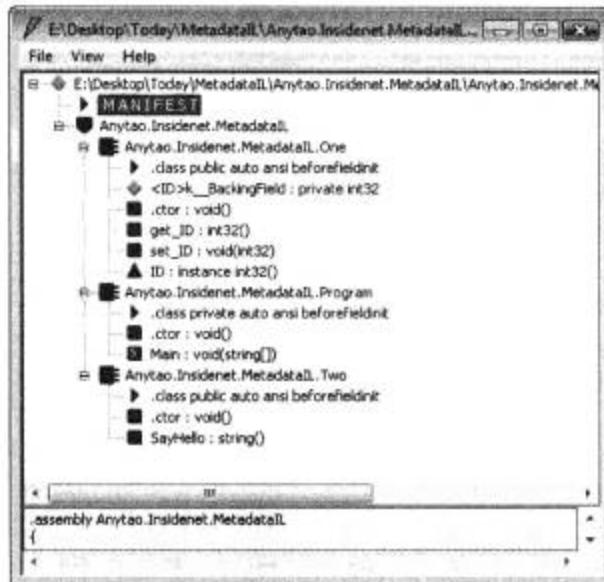


图 4-7 IL 分析

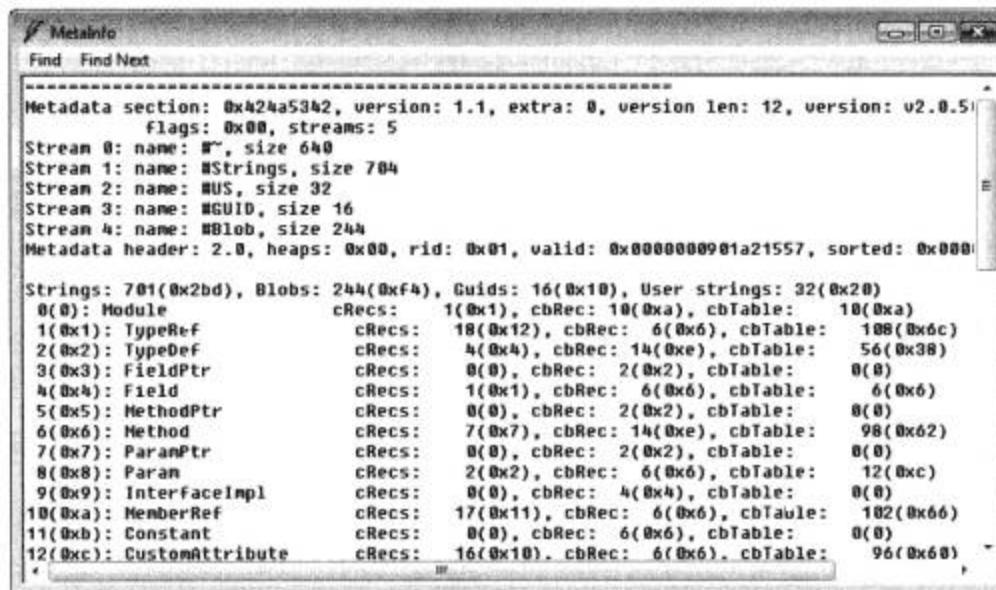


图 4-8 Metadata 信息

其中该程序集使用的元数据主要有：Module、TypeRef、TypeDef、Method、Param、MemberRef、CustomAttribute、Assembly、AssemblyRef 等，同时还包括#Strings、#GUID、#Blob、#US 堆等。

当然，关于 ILDasm 工具，还有很多好玩的使用方式来满足我们探索 IL 代码的好奇心，例如：

```
ildasm Anytao.Insidenet.MetadataIL.exe /output:my.il
```

将反编译结果导出为 il 代码格式，生成一个 my.il 包含了所有的 IL 代码和一个 my.res 包含了所有的资源文件。

```
ildasm Anytao.Insidenet.MetadataIL.exe /text
```

将反编译结果以 Console 形式输出。

当然我们还是推荐以 GUI 形式来查看 IL 细节，组织结构良好的 Class View：

```
ildasm Anytao.Insidenet.MetadataIL.exe
```

下面首先给出参与编译的相关代码文件，然后再展开我们对 Metadata 和 IL 的讨论：

```
public class One
```

```

{
    public int ID { get; set; }
}

public class Two
{
    public string SayHello()
    {
        return "Hello, world.";
    }
}

class Program
{
    static void Main(string[] args)
    {
        int id = 1;
        One one = new One();
        one.ID = id;
        Two two = new Two();
        Console.WriteLine(two.SayHello());
    }
}

```

接着，我们对上述程序的编译执行过程进行一点探索，以命令行编译器来演化其大致的编译过程，以此进一步了解托管模块、程序集和可执行文件之间的关系。

- 打开 Visual Studio 2008 Command Prompt，并定位到 cs 代码所在文件夹，编译 One.cs 为托管模块，执行命令：

```
csc /t:module One.cs
```

执行之后，将生成名为 One.netmodule 的文件。

- 书继续执行，将多个模块打包为程序集。

```
csc /t:library /addmodule:One.netmodule Two.cs
```

执行之后，将生成名为 Two.dll 的文件。

- 最后，编译 Main 函数和 Two.dll 为可执行文件。

```
csc /out:Anytao.Insidenet.MetadatdataIL.exe /t:exe /r:Two.dll /r:mscorlib.dll Program.cs
```

最终将得到本文开始时所加载的用于反编译的程序集文件 Anytao.Insidenet.MetadatdataIL.exe，在该执行命令中对几个指示符开关做点说明：

- /out:Anytao.Insidenet.MetadatdataIL.exe，表示输出的可执行文件，及其名称。
- /t:exe，表示输出的文件类型为 CUI（控制台界面程序）程序；而/t:winexe，表示输出为 GUI（图形界面程序）程序。
- /r:Two.dll，表示引用刚刚生产的 Two.dll 程序集。
- /r:mscorlib.dll，表示引用外部程序集 msclib.dll，因为我们的程序中使用了 Console 静态方法，而该方法被定义在 msclib.dll 中。msclib.dll 是如此的重要，我们将在很多场合与 msclib.dll 不期而遇，其中提供了.NET 框架最基础的运行时支持。

在cmd中的执行过程，是被分解了的编译过程，如图4-9所示，通过分步执行的方式我们对csc编译器的执行过程有个基本的了解，也同时从侧面认识了每次在Visual Studio中执行“Build”或者“ReBuild”的缩影。

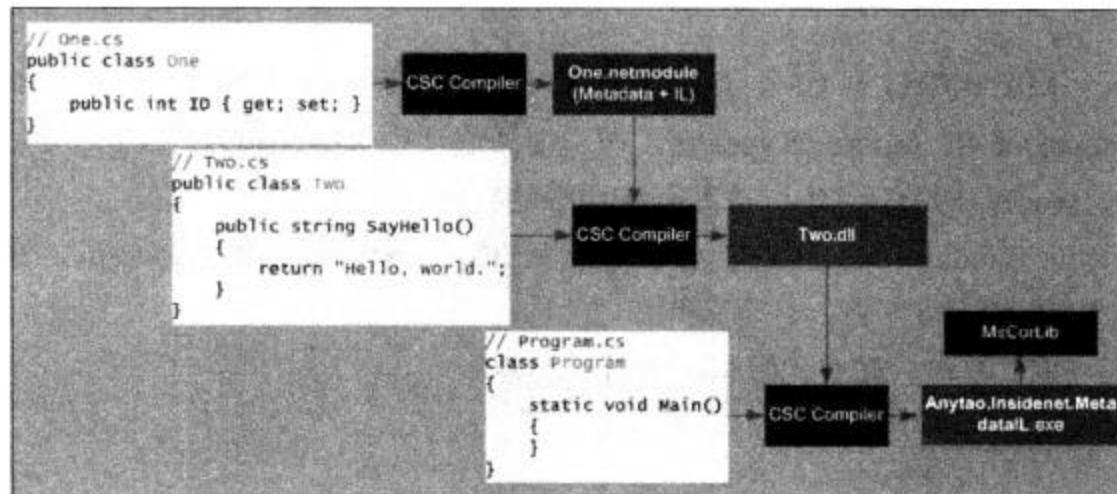


图4-9 编译分解

！注意

在Visual Studio中，编译是分模块进行的，编译结果保存在obj目录中，最后再合并为可执行文件于bin目录。同时在默认情况下，编译过程是增量式的，仅编译发生修改的模块，我将在后文给出较为详细的过程。

同时，我们还可以收获以下几个基本的结论：

- cs代码编译之后将生成元数据和IL，并组成托管模块（Module）的基本单元。
- 多个托管模块组成程序集，其实还包括一定的资源文件，例如图标、图片、文本信息等。
- 程序集或者可执行文件是逻辑组织的基本单元，符合基本的Windows PE文件格式，可以被x86或者x64 Windows直接加载执行。

4.4.3 继续深入

一个或者多个模块，再加上资源文件就形成了程序集（Assembly），作为逻辑组织的基本单元，如图4-10所示。



图4-10 程序集的逻辑组织单元

事实上，图4-10仅仅从粗粒度对程序集的基本组成有个大致的了解，实际上程序集中包含了更复杂的结构和要素，例如PE Signature、Managed Resources、Strong Name Signature Hash，而其中最核心的要素则体现在图4-10中。

- 程序集清单（MANIFEST）包含了程序集的自描述信息，主要包含AssemblyDef、FileDef、ManifestResourceDef和ExportedTypeDef，在反编译选项中MANIFEST包含了详细的内容。本书4.1节“从Hello, world开始认识IL”对其有过详细的描述，在此不再赘述。
- PE文件头，标准Windows PE头文件（PE32或PE32+），PE文件的基本信息，例如文件类型、创建时间、本地CPU信息等。
- CLR头，包含CLR版本、模块元数据、资源等信息。
- 元数据。
- IL。
- 资源文件。

执行View/Statistics菜单，可以打开相关的统计信息：

```

File size          : 5632
PE header size    : 512 (496 used)   ( 9.09%)
PE additional info : 1691           (30.02%)
Num.of PE sections : 3
CLR header size   : 72            ( 1.28%)
CLR meta-data size : 2212          (39.28%)
CLR additional info : 0            ( 0.00%)
CLR method headers : 52            ( 0.92%)
Managed code       : 287           ( 5.10%)
Data               : 2048           (36.36%)
Unaccounted        : -1242          (-22.05%)

Num.of PE sections : 3
.text      - 3072
.rsrc      - 1536
.reloc     - 512

CLR meta-data size : 2212
Module      - 1 (10 bytes)
TypeDef     - 4 (56 bytes)      0 interfaces, 0 explicit layout
TypeRef     - 25 (150 bytes)
MethodDef   - 8 (112 bytes)    0 abstract, 0 native, 8 bodies
FieldDef    - 1 (6 bytes)       0 constant
MemberRef   - 29 (174 bytes)
ParamDef    - 2 (12 bytes)
CustomAttribute- 16 (96 bytes)
StandAloneSig - 4 (8 bytes)
PropertyMap  - 1 (4 bytes)
Property    - 1 (6 bytes)
MethodSemantic- 2 (12 bytes)
Assembly     - 1 (22 bytes)
AssemblyRef  - 1 (20 bytes)
Strings     - 920 bytes
Blobs       - 328 bytes
UserStrings - 68 bytes
Guids       - 16 bytes
Uncategorized - 192 bytes

CLR method headers : 52
Num.of method bodies - 8

```

```

Num.of fat headers - 4
Num.of tiny headers - 4

Managed code : 287
Ave method size - 35

```

关于PE头、CLR头和资源文件的详细内容，限于篇幅我们并不在本文中详细展开，读者可参考MSDN中的详细论述来了解。

IL代码被组织为以下形式：

```

.class public auto ansi beforefieldinit Anytao.Insidenet.MetadataIL.Two
    extends [mscorlib]System.Object
{
    .method public hidebysig instance string
        SayHello() cil managed
    {
        // Code size      11 (0xb)
        .maxstack 1
        .locals init ([0] string CS$1$00Q0)
        IL_0000:  nop
        IL_0001:  ldstr     "Hello, world."
        IL_0006:  stloc.0
        IL_0007:  br.s      IL_0009

        IL_0009:  ldloc.0
        IL_000a:  ret
    } // end of method Two::SayHello

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

} // end of class Anytao.Insidenet.MetadataIL.Two

```

包装在类似于汇编模样的外衣下，我们依稀可见class、System.Object、method、public、string这些面向对象高级语言中的熟悉面孔，不同的只是多了很多beforefieldinit（详情参考9.4节“给力细节：深入类型构造器”）、ret、maxstack、ldstr、stloc这些陌生的指令。然而IL并非一个怪胎，.NET正是基于其本身面向对象的汇编式风格，才造就了IL代码成为名副其实的“中间语言”。通过IL代码，CLR就可在编译时由JIT编译转换为Native Code，我们将在后续章节中继续分析这个过程的来龙去脉。

4.4.4 元数据是什么

元数据，就是描述数据的数据。这一概念并非CLR之独创，Metadata存在于任何对数据和数据关系的描述中，例如程序集清单信息也被称为程序集元数据。另外，一个完备的平台性系统（例如，Microsoft Dynamic CRM系统），基于功能性和扩展性的考量，也往往具有自定义的元数据系统，以描述其类型、界面、关系还有配置等基础信息。

而不同系统的元数据也相应具有本身的特点，.NET元数据也是如此。那么，CLR元数据描述的是哪些

内容呢？正如前文描述的一样，编译之后，类型信息将以元数据的形式保存在PE格式文件中。.NET是基于面向对象的，所以元数据描述的主要目标就是面向对象的基本元素：类、类型、属性、方法、字段、参数、特性等，主要包括：

- 定义表，描述了源代码中定义的类型和成员信息，主要包括：TypeDef、MethodDef、FieldDef、ModuleDef、PropertyDef等。
- 引用表，描述了源代码中引用的类型和成员信息，引用元素可以是同一程序集的其他模块，也可以是不同程序集的模块，主要包括：AssemblyRef、TypeRef、ModuleRef、MethodsRef等。
- 指针表，使用指针表引用未知代码，主要包括：MethodPtr、FieldPtr、ParamPtr等。
- 堆，以stream的形式保存的信息堆，主要包括：#String、#Blob、#US、#GUIDe等。

如前文所述，我们以ILDasm.exe通过反编译的方式，按Ctrl+M组合键来获取该程序集所使用的Metadata信息列表，在.NET中每个模块包含了44个CLR元数据表，如表4-3所示。

表4-3 元数据表

表记录	元数据表	说明
0(0)	ModuleDef	描述当前模块
1(0x1)	TypeRef	描述引用Type，为每个引用到类型保存一条记录
2(0x2)	TypeDef	描述Type定义，每个Type将在TypeDef表中保存一条记录
3(0x3)	FieldPtr	描述字段指针，定义类的字段时的中间查找表
4(0x4)	FieldDef	描述字段定义
5(0x5)	MethodPtr	描述方法指针，定义类的方法时的中间查找表
6(0x6)	MethodDef	描述方法定义
7(0x7)	ParamPtr	描述参数指针，定义类的参数时的中间查找表
8(0x8)	ParamDef	描述方法的参数定义
9(0x9)	InterfaceImpl	描述有哪些类型实现了哪些接口
10(0xa)	MemberRef	描述引用成员的情况，引用成员可以是方法、字段还有属性
11(0xb)	Constant	描述了参数、字段和属性的常数值
12(0xc)	CustomAttribute	描述了特性的定义
13(0xd)	FieldMarshal	描述了与非托管代码交互时，参数和字段的传递方式
14(0xe)	DeclSecurity	描述了对于类、方法和程序集的安全性
15(0xf)	ClassLayout	描述类加载时的布局信息
16(0x10)	FieldLayout	描述单个字段的偏移或序号
17(0x11)	StandAloneSig	描述未被任何其他表引用的签名
18(0x12)	EventMap	描述类的事件列表
19(0x13)	EventPtr	描述了事件指针，定义事件时的中间查找表
20(0x14)	Event	描述事件
21(0x15)	PropertyMap	描述类的属性列表
22(0x16)	PropertyPtr	描述了属性指针，定义类的属性时的中间查找表
23(0x17)	Property	描述属性
24(0x18)	MethodSemantics	描述事件、属性与方法的关联
25(0x19)	MethodImpl	描述方法的实现
26(0x1a)	ModuleRef	描述外部模块的引用

续表

表记录	元数据表	说 明
27(0x1b)	TypeSpec	描述了对TypeDef 或者 TypeRef 的说明
28(0x1c)	ImplMap	描述了程序集使用的所有非托管代码的方法
29(0x1d)	FieldRVA	字段表的扩展, RVA 给出了一个字段的原始值位置
30(0x1e)	ENCLog	描述在 Edit-And-Continue 模式中哪些元数据被修改过
31(0x1f)	ENCMap	描述在 Edit-And-Continue 模式中的映射
32(0x20)	Assembly	描述程序集定义
33(0x21)	AssemblyProcessor	未使用
34(0x22)	AssemblyOS	未使用
35(0x23)	AssemblyRef	描述引用的程序集
36(0x24)	AssemblyRefProcessor	未使用
37(0x25)	AssemblyRefOS	未使用
38(0x26)	File	描述外部文件
39(0x27)	ExportedType	描述在同一程序集但不同模块有哪些类型
40(0x28)	ManifestResource	描述资源信息
41(0x29)	NestedClass	描述嵌套类型定义
42(0x2a)	GenericParam	描述了泛型类型定义或者泛型方法定义所使用的泛型参数
43(0x2b)	MethodSpec	描述泛型方法的实例化
44(0x2c)	GenericParamConstraint	描述了每个泛型参数的约束

然后，是 6 个命名堆，如表 4-4 所示。

表 4-4 命名堆

堆	说 明
#String	一个 ASCII string 数组，被元数据表所引用，来表示方法名、字段名、类名、变量名以及资源相关字符串，但不包含 string literals
#Blob	包含元数据引用的二进制对象，但不包含用户定义对象
#US	一个 unicode string 数组，包含了定义在代码中的字符串（string literals），这些字符串可以直接由 ldstr 指令加载获取
#GUID	保存了 128byte 的 GUID 值，由元数据表引用
#~	一个特殊堆，包含了所有的元数据表，会引用其他的堆
#-	一个未压缩的#~堆。除了#~堆，其他堆都是压缩的

Note：对于#String 和#US，一个简单的区别就是：

```
string hello = "Hello, World";
```

变量 hello 名将保存在#String 中，而代码中字符串信息“Hello, World”则被保存在#US 中。

关于元数据信息的详细描述，例如每个表包含哪些列以及不同表间的关系，请参考[Standard ECMA-335]和[The .NET File Format]。

在 PE 文件格式中，Metadata 有着复杂的结构，我试图从数据库管理数据的角度来理解元数据的结构和关系，所以元数据的逻辑结构被表示为元数据表，类似于数据库表有主键和 Schema，元数据表以 RID（表索引）和元-元数据表示类同的概念，以TypeDef 表为例，通过数据引用关系同时与 Field、Method、TypeRef 等表发生关联，其他表间又有类似的关系，从而形成一个复杂的类数据库结构，如图 4-11 所示。

因此，元数据是保存了类型的编译后数据，是.NET程序运行的基础，我们可以在运行时动态地以反射的方式获取元数据信息，而这些信息在.NET Framework中以System.Type、MethodInfo等封装，例如截取MSDN中一个类间关系的简单示例，如图4-12所示：

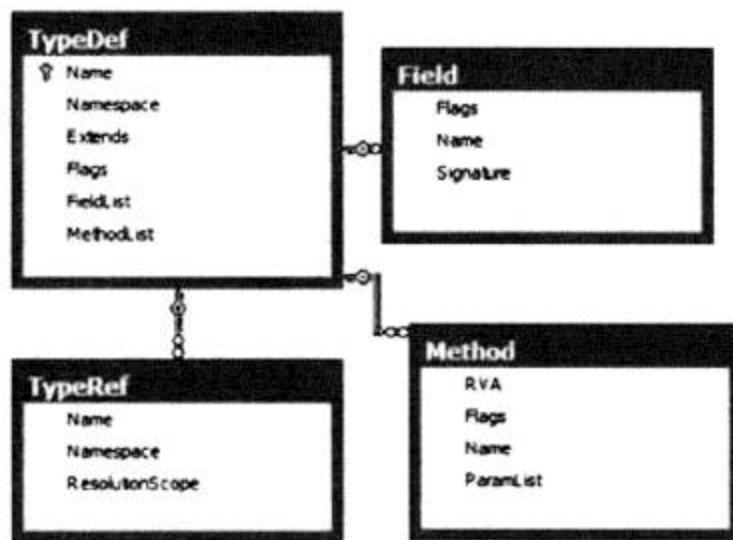


图4-11 元数据表结构

继承层次结构

```

System.Object
System.Reflection.MemberInfo
System.Type
System.Reflection.MethodBase
System.Reflection.ConstructorInfo
System.Reflection.MethodInfo
System.Reflection.FieldInfo
System.Reflection.EventInfo
System.Reflection.PropertyInfo
  
```

图4-12 元数据在BCL的封装

对于每个CLR类型而言都可以通过Object.GetType方法返回其Type，从而任意取到所有运行时的元数据信息：

```

private static void ShowMemberInfo()
{
    var assems = AppDomain.CurrentDomain.GetAssemblies();

    foreach (Assembly ass in assems)
    {
        foreach (Type t in ass.GetTypes())
        {
            foreach (MemberInfo mi in t.GetMembers())
            {
                Console.WriteLine("Name:{0}, Type:{1}", mi.Name, mi.MemberType.ToString());
            }
        }
    }
}
  
```

执行上述方法，将获取一个长长的列表，看到很多熟悉的符号:-)

4.4.5 IL是什么

IL，又称为CIL或者MSIL，翻译为中文就是中间语言，由ECMA组织(Standard ECMA-335)提供完整的定义和规范。顾名思义，任何与CLR兼容的编译器所生成的都是中间语言代码，这是实现CLR跨语言的基础结构之一。IL就像一座桥梁，其指令集独立于CPU指令而存在，可以由JIT编译器在运行时翻译为本地代码执行，连接了任何遵守CLS规范的高级语言，为.NET平台提供了最基本的支持。关于IL的基础内容例如基本类型、IL分析方法、常见指令、基本运算等，详见本章其他内容的相应讨论，在此只对IL基本内容进行一点小结：

- IL 是一种面向对象的机器语言，因此具有面向对象语言的所有特性，类、对象、继承、多态等仍然是 IL 语言的基本概念。
- IL 指令独立于 CPU 指令，CLR 通过 JIT 编译机制将其转换为本地代码。
- IL 和元数据是了解 CLR 运行机制的重要内容，对于我们打开 CLR 神秘面纱有着重要的意义。

如前文论述的一样，可以通过 ILDasm.exe 或者 Reflector 工具对托管代码执行反编译来查看其 IL 代码，在很多情况下 IL 代码分析可以解决高级语言隐藏的语法糖游戏，例如 C#3.0 提出的自动属性、隐式类型、匿名类型、扩展方法等都可以很快从 IL 分析中找到答案，所以适当地了解 IL 是必要的，相应的论述详见 13.2 节“赏析 C# 3.0”。我们通过下面 JIT 编译时的一个片段来了解 IL 代码对于托管程序执行的作用。

另外，Metadata 描述了静态的结构，而 IL 阐释了动态的执行。

IL 代码是通过一个 4 字节大小的地址引用元数据表的。该引用被称为元数据符号（Metadata Token，也就是记录元数据表的位置信息），在 ILdasm.exe 工具中选中“Show token values”，就可以在 IL 代码中看到 IL 代码通过 Metadata Token 引用元数据表的情况：

```
.method /*06000003*/ private hidebysig static
    void Main(string[] args) cil managed
{
    .entrypoint
    // Code size      36 (0x24)
    .maxstack 2
    .locals /*11000002*/ init ([0] int32 id,
        [1] class Anytao.Insidenet.MetadataIL.One/*02000004*/ one,
        [2] class Anytao.Insidenet.MetadataIL.Two/*02000002*/ two)
    IL_0000: nop
    IL_0001: ldc.i4.1
    IL_0002: stloc.0
    IL_0003: newobj     instance void Anytao.Insidenet.MetadataIL.One/*02000004*/::ctor()
/* 06000007 */
    IL_0008: stloc.1
    IL_0009: ldloc.1
    IL_000a: ldloc.0
    IL_000b: callvirt   instance void Anytao.Insidenet.MetadataIL.One/*02000004*/::set_
ID(int32) /* 06000006 */
    IL_0010: nop
    IL_0011: newobj     instance void Anytao.Insidenet.MetadataIL.Two/*02000002*/::ctor()
/* 06000002 */
    IL_0016: stloc.2
    IL_0017: ldloc.2
    IL_0018: callvirt   instance string Anytao.Insidenet.MetadataIL.Two/*02000002*/::
SayHello() /* 06000001 */
    IL_001d: call        void [mscorlib/*23000001*/]System.Console/*01000012*/::WriteLine
(string) /* 0A000011 */
    IL_0022: nop
    IL_0023: ret
} // end of method Program::Main
```

其中，按照 ECMA 定义的规范，元数据第一个字节表示引用的元数据表，而其余三个字节则表示在相应元数据表中的记录，例如 06000003 表示引用了 MethodDef (06) 表的 000003 项 Main 方法。

我们可以通过 Type 的 MetadataToken 属性在运行时反射获取类型的元数据符号，例如：

```
static void Main(string[] args)
{
    Console.WriteLine(typeof(One).MetadataToken);
```

有了上述所有的准备，我们就可以着手分析元数据和 IL 在程序执行时的角色和关联。

下面来说说元数据和 IL 在 JIT 编译时的角色，虽然几个回合的铺垫未免铺张，但是却丝毫不为过，因为只有充分的认知才有足够的体会，技术也是如此。那么，我们就开始沿着方法调用的轨迹，追随元数据和 IL 在那个神秘瞬间所爆发的潜能吧。

4.4.6 元数据和 IL 在 JIT 编译时

CLR 最终执行的只有本地机器码，所以 JIT 编译的作用是在运行时将 IL 代码解析为机器码执行。对于 JIT 编译，可以参阅 MSDN 的详细论述，本文只将目光关注于元数据和 IL 在程序执行时的作用和参与细节。

首先，IL 是基于栈执行的，执行方法调用时，方法参数、局部变量还有返回值等被分配于栈上，并执行其调用过程，既然是关注 JIT 编译时，我们自然而然将关注方法的执行，因为 JIT 编译是以执行方法调用而触发的。

然后，我们对本文开始的代码加点新料：

```
public class Base
{
    public void M()
    {
        Console.WriteLine("M in Base");
    }

    public virtual void N()
    {
        Console.WriteLine("N in Base");
    }
}

public class Three : Base
{
    private static int ID { get; set; }

    public override void N()
    {
        //Something new in Three
        Console.WriteLine("N in Three");
    }

    public void M()
    {
        Console.WriteLine("M in Three");

        M1();
    }

    public void M1()
    {
        Console.WriteLine("M1 in Three");
    }
}
```

还有执行代码：

```
static void Main(string[] args)
{
    Base three = new Three();
    three.M();
    three.N();
}
```

1. 小窥方法表

以该例而言，执行 Main 方法调用时，同时伴随着对于 Three 实例的创建和相应类型信息的加载。然而，类型加载一定是在实例创建之前完成的，也就是我们常常提起的方法表创建。类型加载是由 class loader 负责执行的，其过程简而言之就是从元数据表中获取相应的类型信息，创建方法表（包含 CORINFO_CLASS_STRUCT 结构），其结构主要包括非虚方法表和虚方法表，按照继承的虚方法、新引入的虚方法、实例方法和静态方法的顺序排列，以类 Three 类型为例其 CORINFO_CLASS_STRUCT 结构可以表示如图 4-13 所示的形式。

！注意

在本例中 Three 没有定义任何静态方法，其方法表中父类方法 N 已由子类覆写，同时因为有静态成员存在，CLR 会自动创建类型构造器，详细情况可参考 1.2 节“什么是继承”。

可以通过加载 SOS 调试来了解相应的方法表信息：

- 在 three.N() 调用处打好断点，来查看该时刻的 dump 信息，就像一个内存快照，能发现多少东西就看摄影师的水准了。

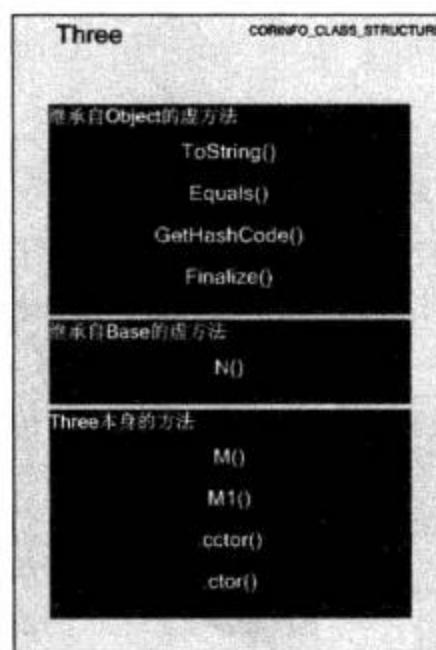


图 4-13 CORINFO_CLASS_STRUCT 结构

- 然后，通过 dumpheap 加载类型信息，获取方法表地址（0x002a354c）。

```
!dumpheap -type Three
Address      MT      Size
01d332c4  002a354c       12
total 1 objects
```

Statistics:

MT	Count	TotalSize	Class Name
002a354c	1	12	Anytao.Insidenet.MetadataIL.Three

- 根据 MT 地址，以 dumpmt 查看相关的 MethodDesc 信息。

```
!dumpmt -md 002a354c
EEClass: 002a15b4
Module: 002a2f2c
Name: Anytao.Insidenet.MetadataIL.Three
mdToken: 02000003 (E:\anytao\Today\OnWriting\MetadataIL\Anytao.Insidenet.MetadataIL\
Anytao.Insidenet.MetadataIL\bin\Debug\Anytao.Insidenet.MetadataIL.exe)
BaseSize: 0xc
ComponentSize: 0x0
Number of IFaces in IFaceMap: 0
Slots in VTable: 10
-----
MethodDesc Table
Entry MethodDesc JIT Name
6f756a70 6f5d1328 PreJIT System.Object.ToString()
6f756a90 6f5d1330 PreJIT System.Object.Equals(System.Object)
6f756b00 6f5d1360 PreJIT System.Object.GetHashCode()
6f7c7460 6f5d1384 PreJIT System.Object.Finalize()
002ac0b8 002a3514 NONE Anytao.Insidenet.MetadataIL.Three.N()
002ac0d0 002a3540 JIT Anytao.Insidenet.MetadataIL.Three..ctor()
002ac0a8 002a34f4 NONE Anytao.Insidenet.MetadataIL.Three.get_ID()
002ac0b0 002a3504 NONE Anytao.Insidenet.MetadataIL.Three.set_ID(Int32)
002ac0c0 002a3520 NONE Anytao.Insidenet.MetadataIL.Three.M()
002ac0c8 002a3530 NONE Anytao.Insidenet.MetadataIL.Three.M1()
```

经过简单的 Dump，方法表的信息和我们图示的信息相差无几，细心的读者可能会发现 Dump 信息中并不包含 Three::cctor()，那么你答对了。图示的 cctor 是我为 Three 实现了类型构造器（静态构造函数）而特别加入的，而代码中 dump 的方法表并没有把类型构造器包含在内，这个小粗心，希望细心的您看得够透。

2. 执行细则

对于具体的执行过程，可表达如下。

- class loader 从 TypeDef 元数据表加载相关元数据信息，包括当前类型、继承层次的所有父类和实现的接口元数据，根据这些信息建立 CORINFO_CLASS_STRUCT 结构，如图 4-14 所示。

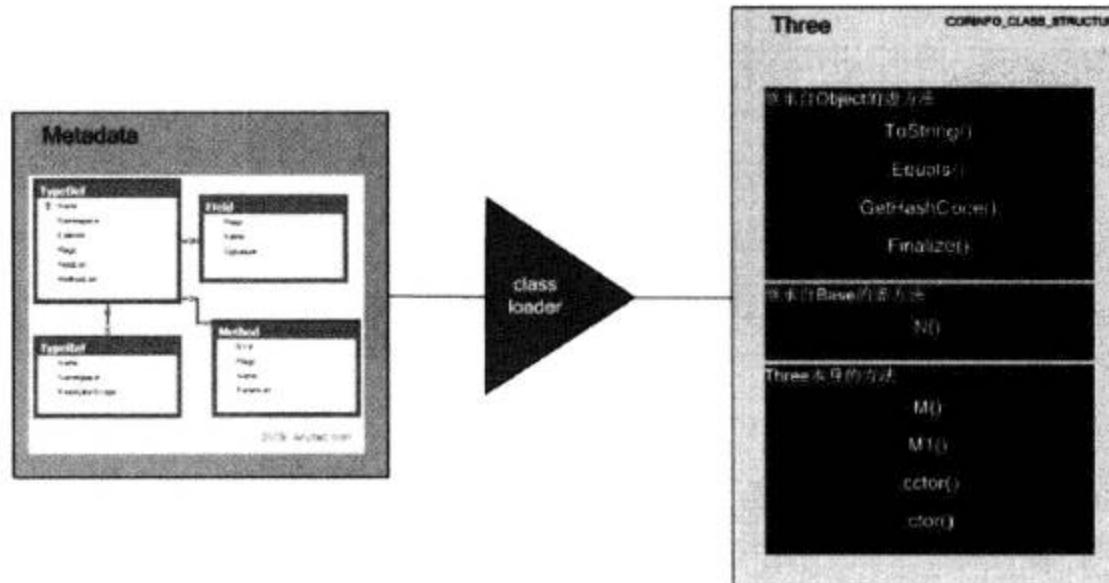


图 4-14 由元数据建立 CORINFO_CLASS_STRUCT 结构

当然，对 class loader，我们可以进行一点知识救急。



上课啦：class loader

Class Loader 是 CLR 提供的基本组件之一，作用正像其名称所宣扬的那样，load 一个 Class 给 CLR，class loader 将 Metadata 和 IL 从 PE 文件中取出，并加载到运行时内存，简单地说就是我们下面要介绍的全过程缩影。

当然，如果你总是对 CLR 的 Classic Loader 耿耿于怀，不能释然，那么我们也可以参考 MSDN 的资料来实现自定义的 Class Loader，希望其中能提供灵光一现的思考。

- 加载之后，方法执行之前的 CORINFO_CLASS_STRUCT 中所有的方法表槽都保存了方法应该执行的行为逻辑，这些信息保存在被称为方法描述（MethodsDesc）的结构中，而 MethodDesc 则被初始化为指向 IL 代码，同时还包含一个指向触发 JIT 编译的 PreJitStub 地址，如图 4-15 所示。

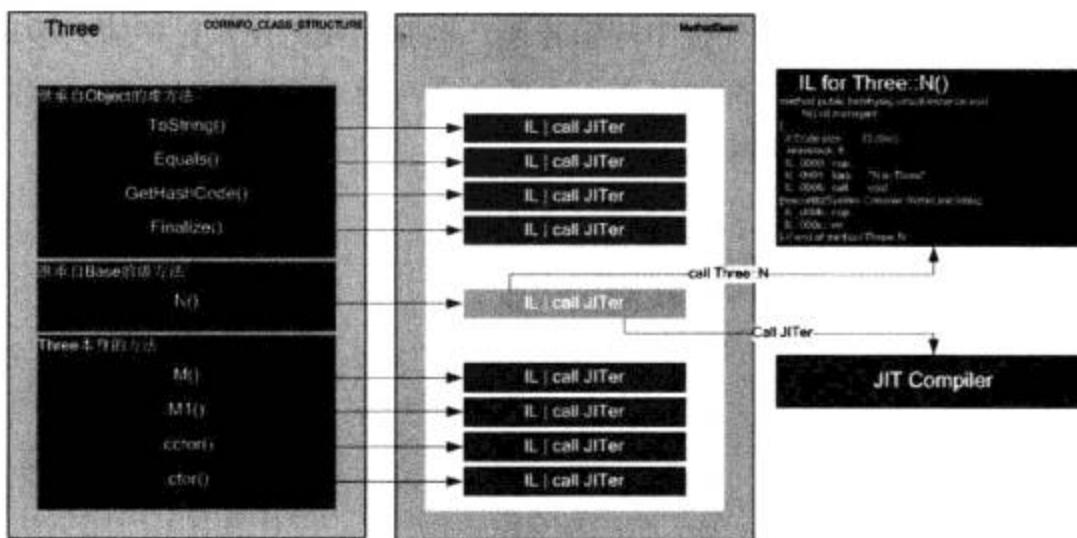


图 4-15 JIT 编译过程：MethodsDesc

上述所有方法描述都指向各自的 IL 代码地址和 JIT 编译器，在此我们仅仅以 N()方法为例来进行说明，详细的情况可以参考 MSDN 相关内容。

- 简单地说，任何方法第一次执行时都会首先触发执行 JIT 编译，JIT 的主要工作就是将 IL 代码翻译为 Native Code，并插入指向 Native Code 的 jmp 指令地址覆盖原来的 Call JIT Compiler 指令，如图 4-16 所示。

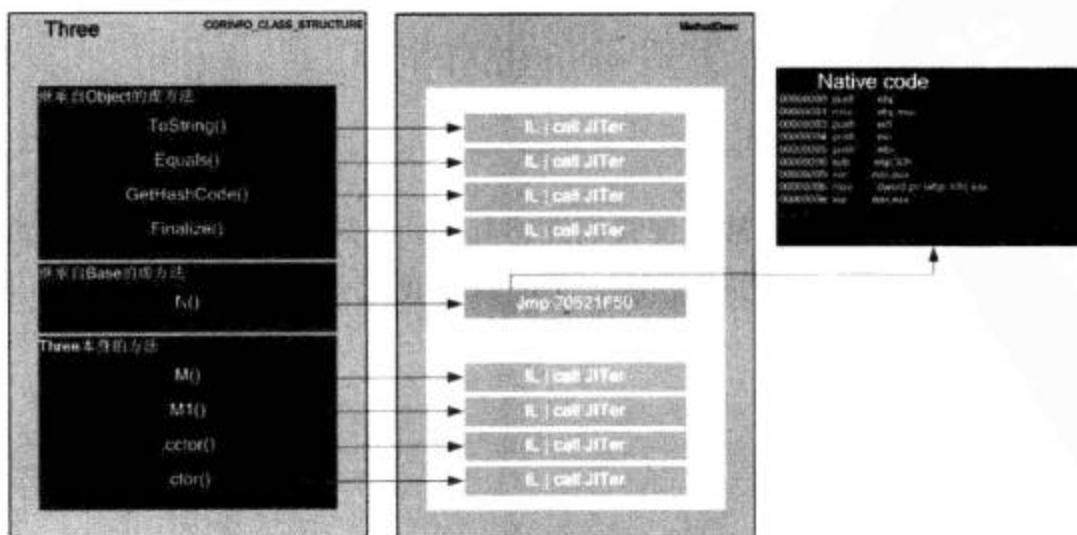


图 4-16 JIT 编译过程：生成 Native Code

- 当该方法再次被执行时，因为 MethodDesc 中保存了机器码地址，以后的执行将不会执行 JIT 编译过程而直接执行 x86(X64)机器码，实现整个执行过程。

纵观整个 JIT 编译的全过程，其细节的实现远比我们这里呈现的复杂，在粗略的步骤中我们大致了解了元数据和 IL 在整个过程中的作用、角色和关系，对了解 CLR 运行机制而言，适当的选择是明智的，如果有更多的心思探索，那么就在以后的岁月中由简及繁吧，但相信这一定是一次美妙的旅程。

4.4.7 结论

Metadata 描述了静态的结构，而 IL 阐释了动态的执行，这一静一动承载了太多的技术奥秘。

当这篇文章行将结束的时候，我发现牵一发而动全身，由此引入的新问题接踵而至，方法调用、程序集、程序域、CLR 加载过程在元数据和 IL 的分析中若隐若现，也驱使我们投入更多的兴趣来举一反三、深入浅出，将这些内容一一过招。由此才能在复杂的概念和本质之余，由点及面地对所有内容综合把握，形成全面的了解，就像串起来的糖葫芦，越吃越有味道。

4.5 经典指令解析之实例创建

本节将介绍以下内容：

- 详细介绍各个实例创建指令
- 对比实例创建指令的异同
- 值类型与引用类型的实例创建

4.5.1 引言

在.NET 类型系统中，CLR 支持两种类型：值类型和引用类型。不同的类型，意味着不同的内存分配和操作方式，值类型和引用类型的创建就是这样。本节从实例创建的角度来分析 IL 中的创建操作指令，请不同的指令逐一登场，比较、鉴别、分析，通过反向过程来体味值类型与引用类型的创建过程。

4.5.2 newobj 和 initobj

newobj 和 initobj 指令就像两个兄弟，常常使我们迷惑其中，只知其然而不知其所以然，这种感觉很郁闷。那么它们是如何完成对引用类型和值类型的创建呢？下面就让我们一探究竟：

1. 代码引入

```
struct MyStruct
{
}

class MyClass
{}
```

```

class ILDemo
{
    public static void Main()
    {
        MyStruct ms = new MyStruct();
        MyClass mc = new MyClass();
        Console.WriteLine("IL Keywords.");
        Console.Read();
    }
}

```

2. 指令说明

```

.method public hidebysig static void Main() cil managed
{
    .entrypoint
    // 代码大小      33 (0x21)
    .maxstack 1
    .locals init (valuetype InsideDotNet.ILBasic.CreateInstance.MyStruct V_0,
                 class InsideDotNet.ILBasic.CreateInstance.MyClass V_1)
    IL_0000: nop
    IL_0001: ldloca.s  V_0
    IL_0003: initobj   InsideDotNet.ILBasic.CreateInstance.MyStruct
    IL_0009: newobj     instance void
    InsideDotNet.ILBasic.CreateInstance.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

```

3. 深入分析

从上面的代码中，我们可以得出哪些值得关注的结论呢？

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，完成对象初始化的；而initobj又如何完成对值类型的初始化呢？

关于newobj指令，我们在后文7.1节“把new说透”将有详细的介绍，在此关于newobj我们有如下分析：

- 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中的引用类型成员初始化为null，而

值类型成员则置为0。

因此，值类型的初始化可以是：

```
//initobj 方式初始化值类型
initobj    InsideDotNet.ILBasic.CreateInstance.MyStruct
```

同时，也可以直接显式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ctor的调用。

```
//调用构造函数方式初始化值类型
call      instance void InsideDotNet.ILBasic.CreateInstance.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用于分配和初始化引用类型对象；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；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。

4.5.3 ldstr

关于对象实例的构造，前面已经有所交代。引用类型一般以newobj指令来完成对象实例的创建，那么string类型又是什么情形呢？

1. 代码引入

```
class Demo_ldstr
{
    public static void Main()
    {
        string fristName = "Wang";
        string secondName = new string('T', 5);
        Console.WriteLine(fristName + secondName);
        Console.Read();
    }
}
```

2. 指令说明

```
.method public hidebysig static void Main() cil managed
{
    .entrypoint
```

```

// 代码大小      36 (0x24)
.maxstack 3
.locals init ([0] string fristName,
[1] string secondName)
IL_0000: nop
IL_0001: ldstr      "Wang"
IL_0006: stloc.0
IL_0007: ldc.i4.s   84
IL_0009: ldc.i4.5
IL_000a: newobj     instance void [mscorlib]System.String::ctor(char,
                                         int32)
IL_000f: stloc.1
IL_0010: ldloc.0
IL_0011: ldloc.1
IL_0023: ret
} // end of method Demo_ldstr::Main

```

3. 深入分析

从 IL 代码中，我们可以清楚地看到

```
string fristName = "Wang";
```

对应的创建指令为

```
IL_0001: ldstr      "Wang"
```

string 类型在这里使用 ldstr 指令来创建，而不是 newobj。那么我们有必要将故事的插曲 ldstr 做以介绍，以便揭开谜底。

ldstr 是 load string 的简写，在 IL 中表示加载字符串，用于构造 string 对象的实例。它是 CLR 一种特殊的构造字符串对象的方式，也是最常用的一种方式，用于从元数据中获得文本常量。MSDN 的解释是：推送对元数据中存储的字符串的新对象引用。这一操作主要包括两个方面：

- 在堆中分配所必需的内存空间。
- 完成从文本使用的格式到运行使用的格式转换操作。

同时，从 IL 分析中，我们依然可以了解到，string 实例的创建也并非 ldstr 此法一种，在特殊情况下，string 的生成不是从元数据的文本常量中构建的，而是使用我们熟悉的 newobj，例如：

```
string secondName = new string('T', 5);
```

对应的 IL 操作为

```
IL_0007: ldc.i4.s   84
IL_0009: ldc.i4.5
IL_000a: newobj     instance void [mscorlib]System.String::ctor(char, int32)
```

84 是字符 “T” 的 ASCII 的值，在此完成的就是将字符值压入栈中，并调用 System.String 构造函数完成初始化。事实上，System.String() 构造函数提供了数个重载方法来以 new 方式创建 string 实例，这种方式并不常见，而更常用的方式还是以 ldstr 方式从元数据获取文本常量，更详细的分析可参考 9.5 节“如此特殊：大话 String”。

4.5.4 newarr

数组的创建又是什么样呢，在 newobj 节最后我们抛出了使用 Newarr 来分配数组元素内存的问题，那么情况究竟是什么样子呢？

1. 代码引入

```
class CreateInstance
{
    public static void Main()
    {
        char[] arrChars = new char[5];
        for (int i = 0; i < 5; i++)
        {
            arrChars[i] = 'a';
        }

        Console.WriteLine(arrChars[0]);
    }
}
```

2. 指令说明

```
.method public hidebysig static void Main() cil managed
{
    .entrypoint
    // 代码大小      41 (0x29)
    .maxstack 3
    .locals init ([0] char[] arrChars,
                 [1] int32 i,
                 [2] bool CS$4$0000)

    IL_0000:  nop
    IL_0001:  ldc.i4.5
    IL_0002:  newarr     [mscorlib]System.Char
    IL_0007:  stloc.0
    IL_0008:  ldc.i4.0
    IL_0009:  stloc.1
    IL_000a:  br.s       IL_0017
    IL_000c:  nop
    IL_000d:  ldloc.0
    IL_000e:  ldloc.1
    IL_000f:  ldc.i4.s   97
    IL_0011:  stelem.i2
    IL_0012:  nop
    IL_0013:  ldloc.1
    IL_0014:  ldc.i4.1
    IL_0015:  add
    IL_0016:  stloc.1
    IL_0017:  ldloc.1
    IL_0018:  ldc.i4.5
    IL_0019:  clt
    IL_001b:  stloc.2
    IL_001c:  ldloc.2
    IL_001d:  brtrue.s   IL_000c
    IL_001f:  ldloc.0
    IL_0020:  ldc.i4.0
    IL_0021:  ldelem.u2
    IL_0022:  call        void [mscorlib]System.Console::WriteLine(char)
    IL_0027:  nop
    IL_0028:  ret
} // end of method CreateInstance::Main
```

3. 深入分析

针对上一段数组创建的IL代码，我们的分析过程为：

- ldc.i4.5，将5作为一个4字节长度整数压入栈顶，对应于arrChars = new char[5]。
- 调用newarr指令为其分配内存空间。具体的细节包括：在调用newarr前将数组的大小压入堆栈，也就是上步完成的工作，然后newarr生成一个指定类型System.Char的数组，并将数组的引用压入堆栈。
- stloc.0表示将数组取出并存入第0个局部变量中。
- stelem.i2，表示为数组赋值，对应于arrChars[0] = 'a'。stelem接受一个数组引用、下标和一个存储到数组中的对象或者值。
- ldelem.u2，接受一个数组引用、下标和类型标记，提取该数组元素地址并装入栈中，对应于arrChars[0]的执行过程。

在上述IL中，基本完成了数组创建、获取数组元素和为数组赋值的相关操作。当然除了以上的指令之外，还有其他的一些操作，主要包括：

- ldlen，用于将数组长度装入栈中。
- ldlema，用于加载数组中特定元素的托管指针。

注意，需要明确的一点是，newarr指令只用于完成一维并且下标从零开始的数组，这种数组被称为一维零基数组（Zero-based Array）。其他类型数组的创建，例如超过一维或者一维但是下标不从零开始的数组，仍然由newobj完成。因此，强烈推荐使用零基数组，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，从而大大提升访问性能。

以简单的情况来说明非零基数组的情况，如下：

```
class CreateInstance
{
    public static void Main()
    {
        int[,] arrNum = new Int32[2, 3];
        arrNum[1, 2] = 5;
        Console.WriteLine(arrNum[1, 2]);
    }
}
```

对应的IL代码如下：

```
.method public hidebysig static void Main() cil managed
{
    .entrypoint
    // 代码大小      33 (0x21)
    .maxstack 4
    .locals init ([0] int32[0...,0...] arrNum)
    IL_0000:  nop
    IL_0001:  ldc.i4.2
    IL_0002:  ldc.i4.3
    IL_0003:  newobj    instance void int32[0...,0...].::ctor(int32,
                                                int32)
    IL_0008:  stloc.0
    ....部分省略....
    IL_001f:  nop
    IL_0020:  ret
} // end of method CreateInstance::Main
```

4.5.5 结论

通过对几个实例创建指令的详细分析，我们基本了解了CLR创建实例的规则与细节，为更进一步地理解

类型系统打下坚实的基础。IL 指令集是个庞大的语法系统集，如果想要逐一掌握所有的 IL 指令以期达到深入了解 IL 语言的目的，显然并非明智之取。

当然，本节只是着眼于 IL 指令角度来阐释实例创建的话题，后文中还将涉及此话题的探讨，从不同的角度来进行另一番梳理。正如对艺术品的鉴赏，需要从不同的方位来品察，对.NET 基础理论的研究也需要从不同的角度来分析，才能更加透彻。

4.6 经典指令解析之方法调度

本节将介绍以下内容：

- 方法调度的实质
- 不同调度指令的区别

4.6.1 引言

方法，用于执行某些操作行为，通常表示为对象或者类型的行为。所有方法，通常包含一个方法名称、参数列表和返回值。那么对方法的调度操作，又该是什么样的情形呢？方法调度是.NET 底层技术中的核心内容之一，也是较为复杂的部分，大师 Don Box 在《Essential .NET》中花了巨幅笔墨来铺陈方法，足以说明方法调度机制的重要性和复杂性。

本节，作为研究 IL 经典指令解析的一篇，没有理由错过解释方法调度的核心指令：call、callvirt、calli。我们不强调.NET 运行机制，而把重心放在如何区别不同的指令执行的不同方式，产生的不同结果，这是直接关系到.NET 执行过程的头等大事，尤其是虚拟方法的多态调用，当然有必要以专题方式来诠释其究竟。通过对静态调度与虚拟调度的区别，我们领会在.NET 中如何通过虚拟方法的调度实现多态技术，从而加深理解.NET 的面向对象机制。

4.6.2 方法调度简论：call、callvirt 和 calli

在 CLR 中，以 static 修饰符来定义静态方法，而以 instance 修饰符来定义实例方法，二者的区别是静态方法只与类型相关，在编译期即可确定执行的操作，因此是静态的；而实例方法只与实例相关，在运行期才能决定执行的操作，因此是动态的。关于静态方法与实例方法，我们将在 8.8 节“动静之间：静态和非静态”中将做全面交代。在.NET 中，默认使用 instance 修饰符。其定义方式为：

```
public class MethodInvoke
{
    public static void StaticMethod()
    {
    }

    public void InstanceMethod()
    {
    }
}
```

对应的 IL 代码为：

```
.method public hidebysig static void StaticMethod() cil managed
.method public hidebysig instance void InstanceMethod() cil managed
```

那么，静态方法与实例方法的调用情况又该如何呢？

```
class IL_Method_Test
{
    public static void Main()
    {
        //调用静态方法
        MethodInvoke.StaticMethod();
        //调用实例方法
        MethodInvoke mi = new MethodInvoke();
        mi.InstanceMethod();
    }
}
```

对应的 IL 代码为：

```
.method public hidebysig static void Main() cil managed
{
    .entrypoint
    // 代码大小      21 (0x15)
    .maxstack 1
    .locals init ([0] class InsideDotNet.ILBasic.MethodInovation.MethodInvoke mi)
    IL_0000:  nop
    IL_0001:  call      void
    InsideDotNet.ILBasic.MethodInovation.MethodInvoke::StaticMethod()
    IL_0006:  nop
    IL_0007:  newobj     instance void
    InsideDotNet.ILBasic.MethodInovation. MethodInvoke::ctor()
    IL_000c:  stloc.0
    IL_000d:  ldloc.0
    IL_000e:  callvirt   instance void
    InsideDotNet.ILBasic.MethodInovation. MethodInvoke::InstanceMethod()
    IL_0013:  nop
    IL_0014:  ret
} // end of method IL_Method_Test::Main
```

分析上述 IL 代码，我们可知以下结论：

- call 指令用于执行静态调度，而 callvirt 用于执行动态调度。
- 默认情况下，IL 执行静态调用，实例调用必须显式指出，例如 callvirt instance void MethodInvoke:: InstanceMethod()。
- 以 ret 标记返回值。

通过上面的实例分析，对方法调度的几个指令我们做以简要的梳理。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。call 指令根据引用变量的类型来调用方法，因此通常用于调用非虚方法。
- 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；callvirt 指令根据引用变量指向的对象类型来调用方法，执行时方法会递归的调用自己直到堆栈溢出，从而实现了在运行时的动态绑定，

因此通常用于调用虚方法。

- 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。
- 另外，还包括使用静态调度的 jmp 指令和类似于 jmp 方式的 tail call 调用模式，这种方式实为少见，因此就不多做介绍。

4.6.3 直接调度

直接调度包括：call 指令的静态调度和 callvirt 指令的虚拟调度。那么，具体又是如何使用它们来进行方法调度呢？二者的区别又在哪里？还是以代码来展开分析，以结果导向本质。

首先是定义部分，我们关注其继承关系、是否以虚函数实现：

```
public class Father
{
    public void DoWork()
    {
        Console.WriteLine("Father.DoWork()");
    }

    public virtual void DoVirtualWork()
    {
        Console.WriteLine("Father.DoVirtualWork()");
    }

    public virtual void DoVirtualAll()
    {
        Console.WriteLine("Father.DoVirtualAll()");
    }
}

public class Son : Father
{
    public static void DoStaticWork()
    {
        Console.WriteLine("Son.DoStaticWork()");
    }

    public new void DoWork()
    {
        Console.WriteLine("Son.DoWork()");
    }

    public new virtual void DoVirtualWork()
    {
        base.DoVirtualWork();
        Console.WriteLine("Son.DoVirtualWork()");
    }

    public override void DoVirtualAll()
    {
        Console.WriteLine("Son.DoVirtualAll()");
    }
}

public class Grandson : Son
{
    public override void DoVirtualWork()
```

```

    {
        base.DoVirtualWork();
        Console.WriteLine("Grandson.DoVirtualWork()");
    }

    public override void DoVirtualAll()
    {
        base.DoVirtualAll();
        Console.WriteLine("Grandson.DoVirtualAll()");
    }
}

```

然后是执行部分，虽然简单，但是结果却值得回味：

```

class IL_Method_Test
{
    public static void Main()
    {
        Father son = new Son();
        son.DoWork();
        son.DoVirtualWork();

        Son.DoStaticWork();

        Father aGrandSon = new Grandson();
        aGrandSon.DoWork();
        aGrandSon.DoVirtualWork();
        aGrandSon.DoVirtualAll();

        Console.Read();
    }
}

```

对执行结果（如图4-17所示），你是否一清二楚呢？



图4-17 方法调度示例的执行结果

下面我们结合相应的IL代码，对执行过程做简要说明，如下：

```

.method public hidebysig static void Main() cil managed
{
    .entrypoint
    // 代码大小      61 (0x3d)
    .maxstack 1
    .locals init ([0] class InsideDotNet.ILBASIC.MethodInovation.Father son,
                 [1] class InsideDotNet.ILBASIC.MethodInovation.Father aGrandSon)
    IL_0000: nop
    IL_0001: newobj   instance void InsideDotNet.ILBASIC.MethodInovation.Son::ctor()
    IL_0006: stloc.0
    IL_0007: ldloc.0
    IL_0008: callvirt  instance void
InsideDotNet.ILBASIC.MethodInovation.Father:: DoWork()
    IL_000d: nop

```

```

IL_000e: ldloc.0
IL_000f: callvirt instance void
InsideDotNet.ILBasic.MethodInovation.Father:: DoVirtualWork()
IL_0014: nop
IL_0015: call void
InsideDotNet.ILBasic.MethodInovation.Son::DoStaticWork()
IL_001a: nop
IL_001b: newobj instance void
InsideDotNet.ILBasic.MethodInovation.Grandson::.. ctor()
IL_0020: stloc.1
IL_0021: ldloc.1
IL_0022: callvirt instance void
InsideDotNet.ILBasic.MethodInovation.Father::DoWork()
IL_0027: nop
IL_0028: ldloc.1
IL_0029: callvirt instance void
InsideDotNet.ILBasic.MethodInovation.Father:: DoVirtualWork()
IL_002e: nop
IL_002f: ldloc.1
IL_0030: callvirt instance void
InsideDotNet.ILBasic.MethodInovation.Father:: DoVirtualAll()
IL_0035: nop
IL_0036: call int32 [mscorlib]System.Console::Read()
IL_003b: pop
IL_003c: ret
} // end of method IL_Method_Test::Main

```

接下来，我们逐一分析其执行过程：

- son.DoWork(), 对应于 callvirt instance void Father::DoWork(), 对于非虚方法，在子类中如果出现相同名称的方法名，那么最好使用 new 关键字来标记对父类的阻断，或者干脆给子类重新命名，本例选择的方式并不值得推荐。
- son.DoVirtualWork(), 对应于 callvirt instance void Father::DoVirtualWork(), 由于 new 关键字的阻断作用，隐藏了基类虚方法的实现，因此调用的仍然是 Father.DoVirtualWork()。
- Son.DoStaticWork(), 对应于 call void Son::DoStaticWork(), 实现以 call 指令调用静态方法。
- aGrandSon.DoWork(), 对应于 callvirt instance void Father::DoWork(), 类似于 son.DoWork(), 在子类 Grandson 中直接继承基类 Father 的公共方法。
- aGrandSon.DoVirtualWork(), 对应于 callvirt instance void Father::DoVirtualWork(), 同样是由于 new 关键字的阻断作用，隐藏了基类成员。
- aGrandSon.DoVirtualAll(), 对应于 callvirt instance void Father::DoVirtualAll(), 由于 DoVirtualAll() 是虚函数，并且在子类中以 override 重写了基类方法，因此最终执行结果为示例所示。

虽然有以上的通用性结论，但是对于 call 和 callvirt 不可一概而论。因为，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。具体的情况包括：

(1) call 调用虚方法的情况

- 密封类型的引用调用虚方法时，采用 call 调用可以减少 callvirt 进行类型检查的时间，提高调用性能。
- 值类型调用虚方法时，因为值类型首先是密封的，其次 call 调用可以阻止值类型被执行装箱。
- 基类调用虚方法时，采用 call 可以避免 callvirt 递归调用本身引起的堆栈溢出。常见的覆写例如，实现 System.Object 的虚方法 Equals()、ToString() 时，就采用 call 调用方式。

以基类调用虚方法为例，我们在自定义类中实现 base.Equals 调用如下：

```
class call_callvirt
{
    public override bool Equals(object obj)
    {
        return base.Equals(obj);
    }
}
```

从 IL 中，可以证实我们的推断：

```
.method public hidebysig virtual instance bool
    Equals(object obj) cil managed
{
    // 代码大小      13 (0xd)
    .maxstack 2
    .locals init ([0] bool CS$1$0000)
    IL_0000:  nop
    IL_0001:  ldarg.0
    IL_0002:  ldarg.1
    IL_0003:  call     instance bool [mscorlib]System.Object::Equals(object)
    IL_0008:  stloc.0
    IL_0009:  br.s     IL_000b
    IL_000b:  ldloc.0
    IL_000c:  ret
} // end of method call_callvirt::Equals
```

(2) callvirt 调用非虚方法的情况

- 常见于在引用类型中调用非虚方法的情况，其原因是 callvirt 调用时，如果引用变量为 null 则会抛出 NullReferenceException，而 call 调用则不会抛出任何异常。为类型安全起见，在 C# 中会调用 callvirt 来完成引用类型的非虚方法调用。例如：

```
class IL_Method_Test
{
    public static void Main()
    {
        MyRefMethod aRefMethod = new MyRefMethod();
        aRefMethod = null;
        aRefMethod.ShowInfo();
    }
}
```

执行上述代码会抛出 NullReferenceException 异常，从 IL 代码分析来看，和我们讨论的规则并无二致，如下：

```
.method public hidebysig static void Main() cil managed
{
    .entrypoint
    // 代码大小      17 (0x11)
    .maxstack 1
    .locals init ([0] class InsideDotNet.ILBasic.MethodInovation.MyRefMethod aRefMethod)
    IL_0000:  nop
    IL_0001:  newobj    instance void
    InsideDotNet.ILBasic.MethodInovation.MyRefMethod::ctor()
    IL_0006:  stloc.0
    IL_0007:  ldnul1
    IL_0008:  stloc.0
    IL_0009:  ldloc.0
```

```

IL_000a: callvirt instance void
InsideDotNet.ILBasic.MethodInovation.MyRefMethod:: ShowInfo()
IL_000f: nop
IL_0010: ret
} // end of method IL_Method_Test::Main

```

由以上的分析看来, call 和 callvirt 指令似乎并无明确的规律, 对虚方法与非虚方法的调用没有找到严格的界定, 其实这就对了。例如, 基于执行性能的考虑, 在密封类中以 call 指令来调用虚方法会更好; 而基于安全机制的考虑, 则在引用类型中以 callvirt 指令来调用非虚方法会更安全。

当然, 我们对这两个指令的把握, 还是有一定的原则可循, 这就是: call 指令调用静态类型、声明类型的方法, 而 callvirt 调用动态类型、实际类型的方法。万变不离其宗, 这条规则就是 call 指令与 callvirt 指令骨子里的区别。

4.6.4 间接调度

在 IL 中 calli 指令通过函数指针来调用一个方法。但是需要明确的是, C#中并没有提供这种机制来发出 calli 指令。如果足够了解 ILGenerator 类和 OpCode 类的话, 可以使用 System. Reflection.Emit.ILGenerator 类的相关方法来实现, 不过前提是你必须熟悉地了解调用指令的堆栈转换操作, 然后以 ILGenerator 的 Emit 方法发出指令来实现, 这对大部分人来说是件有挑战的事情。基于这种方法不光可以实现对 calli 指令的操作, 对其他指令也能胜任。

我们说 calli 指令通过函数指针间接调用一个方法, 那么这个函数指针又如何获取呢? 答案是 ldftn 或 ldvirtftn 指令。我们还是直接以 IL 代码来了解 calli 指令的调用机制吧。

打开记事本输入如下代码:

```

.assembly HelloWorld{}
.assembly extern mscorelib{}

.class IL_calli extends [mscorelib]System.Object
{
    .method public static void HelloWorld()
    {
        .maxstack 1

        ldstr "Hello world by calli."
        call void [mscorelib]System.Console::WriteLine(string)
        ret
    }

    .method public static void Main()
    {
        .entrypoint
        .maxstack 1

        ldftn void IL_calli::HelloWorld()
        calli void()
        ret
    }
}

```

编译成功, 即可生成 IL_calli.exe 程序, 执行结果一切正常。在此, 我们就是以 ldftn 指令获取方法指针,

再以 calli 指令间接调用。

4.6.5 动态调度

方法调度还有一种重要的形式，那就是基于反射技术的动态调度机制。其基本原理是在运行时，查找方法表的信息来实施调用的方式。常见的实现方式有：MethodInfo.Invoke()方式和动态方法委托（Dynamic Method Delegate）方式。

4.6.6 结论

方法调度是.NET机制中重要的环节，以call、callvirt、calli为核心的调用指令为我们揭示了方法调度中的种种情形，尤其是虚方法的调用机制成为实现面向对象多态性的技术基础，绝对值得我们花力气与时间来啃这块硬骨头。本节从方法调度的各个层面与其应用着手，揭示了如何理解与应用方法调度的不同环节。

参考文献

Don Box, Chris Sells, Essential .NET

Jeffrey Richter, Applied Microsoft .NET Framework Programming

David Chappell, Understanding .NET

Joe Duffy, Professional .NET Framework 2.0

Daniel Pistelli, The .NET File Format

Standard ECMA-335, <http://www.ecma-international.org/publications/standards/Ecma-335.htm>

Sameers, Introduction to IL Assembly Language,

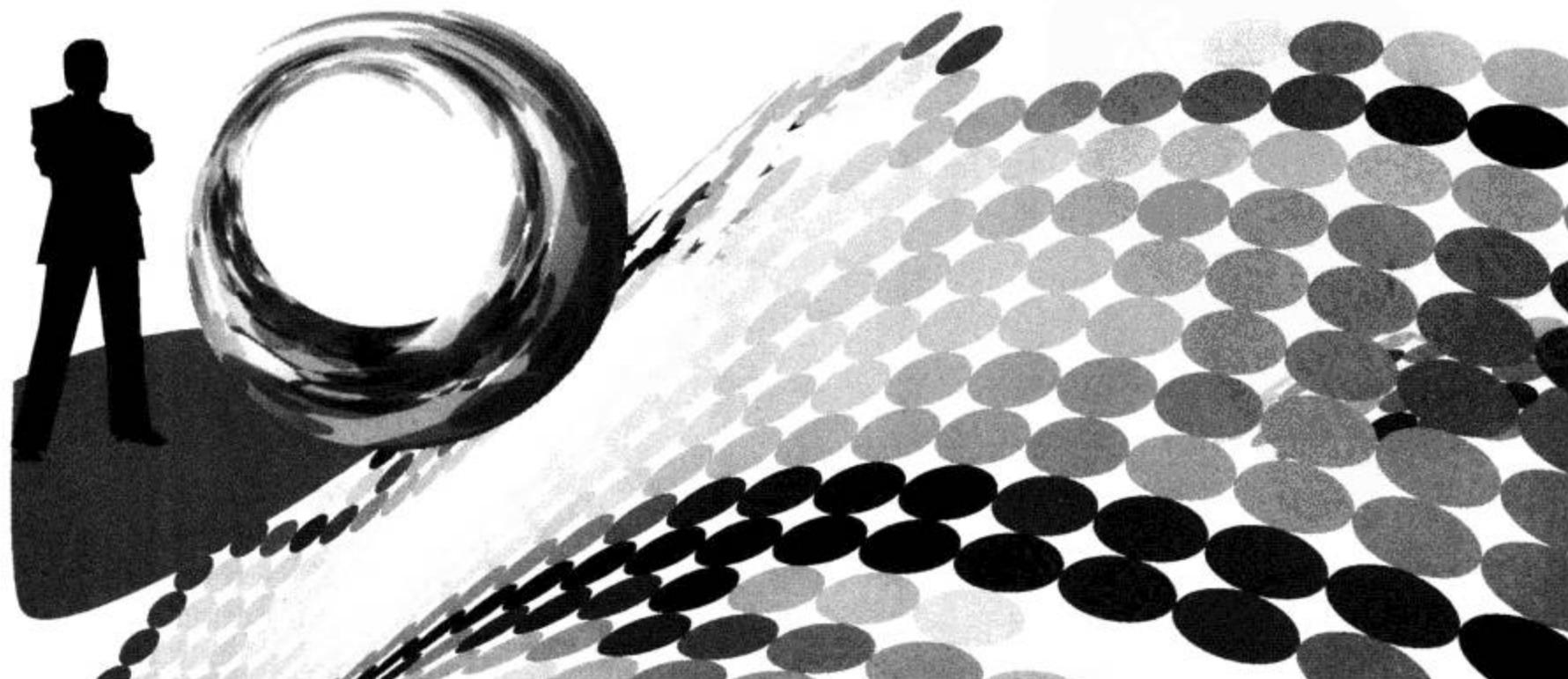
<http://www.codeproject.com/dotnet/ilassembly.asp>

Kenny Kerr, Introduction to MSIL, [http://weblogs.asp.net/kennykerr/archive/tags/ Introduction + to+ MSIL/default.aspx](http://weblogs.asp.net/kennykerr/archive/tags/Introduction+to+MSIL/default.aspx)

Alex F, MSIL Tutorial, http://www.codeguru.com/Csharp/.NET/net_general/il/article.php/c4635

第5章 品味类型

5.1 品味类型——从通用类型系统开始 / 149	5.3 参数之惑——传递的艺术 / 167
5.1.1 引言 / 149	5.3.1 引言 / 168
5.1.2 基本概念 / 149	5.3.2 参数基础论 / 168
5.1.3 位置与关系 / 150	5.3.3 传递的基础 / 169
5.1.4 通用规则 / 151	5.3.4 深入讨论, 传递的艺术 / 170
5.1.5 结论 / 152	5.3.5 结论 / 174
5.2 品味类型——值类型与引用类型 / 152	5.4 皆有可能——装箱与拆箱 / 175
5.2.1 引言 / 152	5.4.1 引言 / 175
5.2.2 内存有理 / 152	5.4.2 品读概念 / 176
5.2.3 通用规则与比较 / 156	5.4.3 原理分拆 / 176
5.2.4 对症下药——应用场合与注意 事项 / 158	5.4.4 还是性能 / 179
5.2.5 再论类型判等 / 159	5.4.5 重在应用 / 180
5.2.6 再论类型转换 / 159	5.4.6 结论 / 182
5.2.7 以代码剖析 / 160	参考文献 / 182
5.2.8 结论 / 167	



5.1 品味类型——从通用类型系统开始

本节将介绍以下内容：

- .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
- .NET 的规范与实现概述

5.1.1 引言

在开篇用图形（如图 5-1 所示）的形式来展示本节主题，是想强调这几个概念的重要性，引起更多的关注，同时希望从剖析其关联的角度来讲述.NET Framework 背后的故事。我们认为想要深入地了解.NET，必须从了解类型开始，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（CTS，Common Type System）。

之所以将最基本的内容加大笔墨以独立的章节来展示，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，我发现有很多朋友对.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，使我们在深入探索知识的过程中，更加游刃有余。

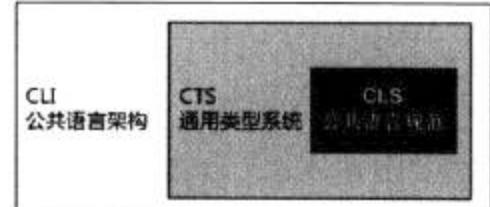


图 5-1 公共语言架构

5.1.2 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？

还是引入一个现实的场景来讨论吧。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，例如 Dim i as Single 变量 i，编译器会自动实现将 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。可以简单地说，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，如图 5-2 所示。



图 5-2 类型体系图（图片来源：MSDN）

.NET 提供了丰富的类型层次结构，从图 5-2 中也可以看出该层次结构的大致概况，本书的基本内容之一就是对这个层次中的各个概念进行深度剖析，在随后的各个章节中将逐渐有所涉猎。关于值类型和引用类型，下一节将有更加详细的探讨和解析，在此不作进一步深入，但是上面的这张图有必要铭记于心，因为没有什么比这个更基础的了。

5.1.3 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，我们期望以这种方式来自然地引出.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，来讲明白各个技术点的细微联系，从大局的角度来把握。我想，这样也可以更好地理解 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 Framework，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（CLI, Common Language Infrastructure），主要包括通用类型系统（CTS）、公共语言规范（CLS, Common Language Specification）和通用中间语言（CIL, Common Intermediate Language）。以图的形式来看看 CTS 在.NET 技术阵营中的位置（如图 5-3 所示），再来简要地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准化。
- CTS，本节主题，此不赘述。
- 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，由本节开头的图，即图 5-1 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，常被称为 MSIL（即代表 IL 的微软实现语言），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，其运行于 Windows 平台之上。
- CLR，.NET 框架核心，也是本书的核心。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，将伴随着本书的展开来慢慢深入，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，本书在第 10 章“格局之选——命名空间剖析”部分，将对 FCL 有详细的讨论。

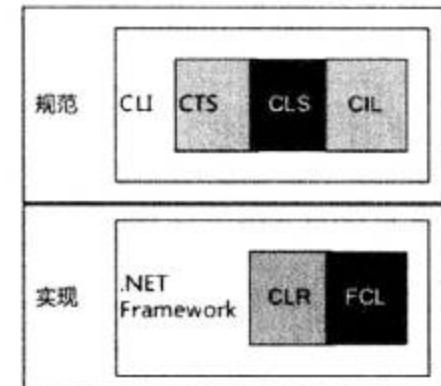


图 5-3 .NET 的规范和实现

这些基本内容相互联系，在此仅仅来澄清其概念、联系和功能，力度显然还不够。我们只是希望能够抛砖引玉，相信以此为入口进行更深入的探索对每个设计人员而言都是一个关键环节，就像对 FCL 的认识，需要实践，需要时间，需要心思，需要积累。

5.1.4 通用规则

总结通用类型的规则，主要包括：

- .NET 中，所有的类型都继承自 System.Object 类。
- 类型转换，通常有 is 和 as 两种模式，具体的探讨可以参考 8.5 节“恩怨情仇：is 和 as”。另外，还有其他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，适用于将字符串类型转换为其他的基本类型。
- 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，可以清楚地做以区别，例如：

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

其实，我们常用的 int、char、string 是 C# 定义的基元类型，对应于 CTS 定义的 System.Int32、System.Char、System.String。编译器将代码编译为 IL 时，会将 int 类型自动翻译为 System.Int32。

- 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，从而可以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。

5.1.5 结论

类型是个老掉牙的话题。然而在实际的程序设计中，我们却经常吃它的亏。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纠缠不清，所以清晰、透彻地了解类型，十分必要。重要的是，我们以什么角度来了解和消化，内功的修炼还是要从心法开始。

品味类型，就从 CTS 开始了。

5.2 品味类型——值类型与引用类型

本节将介绍以下内容：

- 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

5.2.1 引言

终于开始对值类型和引用类型做个全面的讲述了。

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。本节首先从内存的角度了解值类型和引用类型的实质；然后分析不同类型在系统设计、性能优化方面的作用及影响，从内存调试的角度来着眼设计分析，提出关于类型系统的通用性规则；最后结合对内存和规则的讨论，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地对理论做以延伸，并以一定的 IL 分析和内存分析来融会贯通，建立更加透彻和全面的理解。

5.2.2 内存有理

1. 基本概念

从 5.1 节“品味类型——从通用类型系统开始”可知，CLR 支持两种基本类型：值类型和引用类型。

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。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。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 以及其他自定义引用类型时，该变量即为引用类型。

表 5-1 简单地列出类型的进一步细分，以便为我们建立一个清晰的类型概念，这是最基础也是最必需的内容。

表 5-1 .NET 的主要类型

类别	描述
值类型	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	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
	数组类型
	委托类型

2. 内存深入

(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（垃圾回收，Garbage Collection）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：

```
public class Test_ClassAndStruct
{
    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 7.1 节“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（如图 5-4 所示）来展示一下。

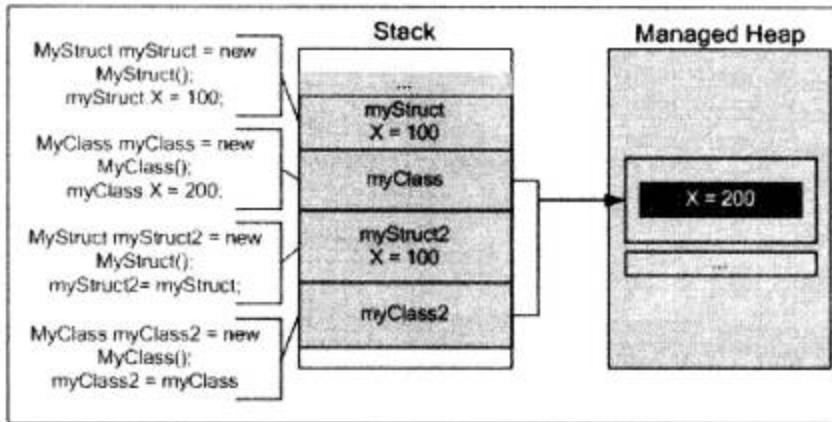


图 5-4 内存概况

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 5-4 中我们还可以显而易见地看出，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，而按址传递会改变实参的值，道理正在于此。

对于内存分配的更详细位置，可以描述如下：

- 值类型变量作为局部变量时，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，例如：如果实例的大小小于 85 000Byte 时，则该实例将创建在 GC 堆上；而当实例大小大于等于 85 000byte 时，则该实例创建在 LOH (Large Object Heap) 堆上。

在 6.2 节“对象创建始末”中将对值类型和引用类型实例的分配过程及其内存布局做以详细的演化与

分析。

3. 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，关于这一话题的论述和关注都不是很多。因此我们很有必要展开来说说，从上文对.NET的内存机制着手来理解会水到渠成。

(1) 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如图 5-5 所示。

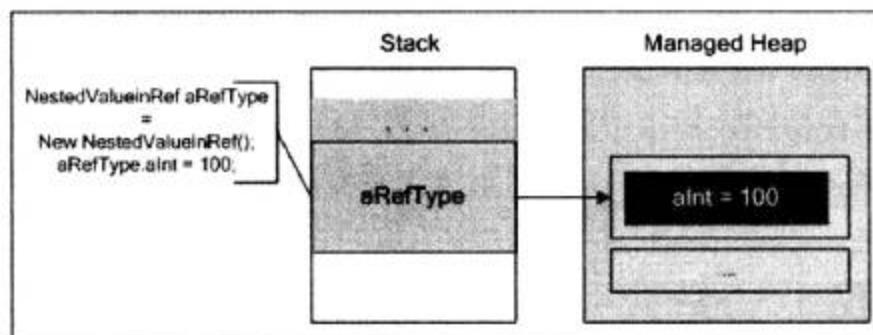


图 5-5 内存概况

(2) 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 mc)
    {
        myClass.X = 1;
        myClass.Y = 2;
        myClass = mc;
    }
}
```

其内存分配图可以表示为如图 5-6 所示。

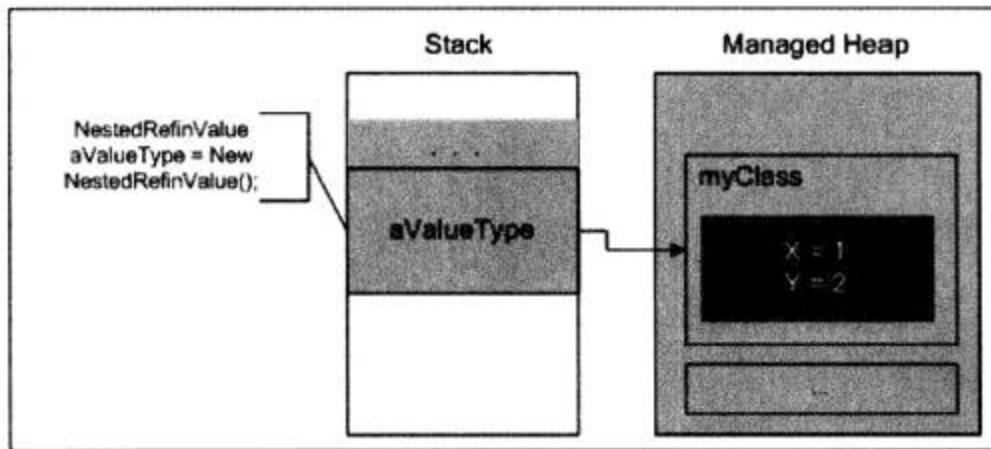


图 5-6 内存概况

由此，可以得出一个简单的结论，那就是：值类型实例总是分配在它声明的地方，声明为局部变量时其被分配在堆栈上，声明为引用类型成员时其被分配到托管堆上；引用类型实例则总是分配在托管堆上。掌握了这一规则，也就掌握了万变不离其宗的法门，对于对象的分配就将一目了然。

4. 一个简单的讨论

通过上面的分析，如果有如下的执行时：

```
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？

分析如下：根据 CLR 的内存机制，数组本身是引用类型，内存分配于托管堆，而 myType 为指向托管堆的引用。不同的是值类型和引用类型数组在托管堆的布局是有区别的：如果 AType 为 Int32 类型，则表示其元素是值类型，myType 将保存指向托管堆中的一块大小为 4×10 byte 的内存地址，并且将所有的元素赋值为 0；而如果 AType 为引用类型，则数组由 10 个引用组成，并且所有的元素被设置为 null 值，表示为空。

5.2.3 通用规则与比较

1. 通用有规则

- string 类型是个特殊的引用类型，具有 Immutability 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。另外，string 类型重载了 == 操作符，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：

```
public static void Main()
{
    string aString = "123";
    string bString = "123";
    //显示为 true, 等价于 aString.Equals(bString)
    Console.WriteLine((aString == bString));
    string cString = bString;
    cString = "456";
    //显示为 false, 等价于 bString.Equals(cString);
    Console.WriteLine((bString == cString));
}
```

至于 string 类型的特殊性解释，详见 9.5 节“如此特殊：大话 String”的论述。

- 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 Test_IsValueType
{
    public static void Main()
    {
        MyStructTester aStruct = new MyStructTester();
        System.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，在 5.4 节“皆有可能——装箱与拆箱”中就将进行详细的论述。
- `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 GC 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下，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))); //输出 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，通常对应于.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`。

2. 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，因此以按值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内存中的同一块地址。

- 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 8.2 节“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 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。通常 struct 比 class 更高效；而引用类型的内存分配与回收，都由 GC 来完成。
- 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。

5.2.4 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，我们就可以很好地总结出值类型和引用类型在系统设计时，如何做出选择。当然这里的重点是告诉你，如何去选择使用值类型，因为引用类型才是.NET 的主体，已经在大部分的场合中赢得青睐。

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系统性能的改善。
- 结构简单，不必在多态的情况下，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，例如在频繁调用 Hashtable 对象的 Add 方法时，将产生大量的装箱操作。关于这点将在 8.9 节“集合通论”中重点讨论。

2. 引用类型的应用场合

- 可以简单地说，引用类型是.NET 世界的全职杀手，.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。
- 参数传递情况也是需要考虑的因素。

5.2.5 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 方法的覆写，例如值类型的“始祖” System.ValueType 就覆写了 Equal 方法，以实现对实例数据的判等。因此，类型的判等也要从覆写或者重载 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。

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。

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 的大部分类都覆写了 Equals 方法，因此判等的返回值要根据具体的覆写情况决定。

- 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 是覆写。

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

5.2.6 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，因此有必要在这个主题里做以总结。常见的类型转换包括：

- 隐式转换：由低级类型向高级类型的转换过程。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式转换，主要指装箱和拆箱转换。

例如，下面示例将不能通过编译检查：

```
public static void Main()
```

```

    short s = 1;
    s = s + 1;
}

```

因为 s 首先会转换为 Int32 型进行+运算，然后将 Int32 型结果赋给 short 型变量，在此必须通过强制类型转换，否则将导致编译错误。

- 显式转换：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 8.5 节“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 以外的其他基元类型都有 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，主要给用户提供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 class Custom
{
    public string Name;
    public Int32 UserType;

    // 强制 Custom 类型转换为 VipCustom 类型
    public static explicit operator VipCustom(Custom user)
    {
        return new VipCustom(user.Name, user.UserType + 1);
    }
}

public class VipCustom
{
    public string Name;
    public Int32 UserType;

    public VipCustom(string name, Int32 userType)
    {
        Name = name;
        UserType = userType;
    }
}

```

注意，所有的转换都必须是 static 的。

5.2.7 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）三个方面来逐个解释知识点，最后做以总结陈述，这样就可以有更深刻的理解。

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，完整的代码可以到 <http://book.anytao.net> 下载。我们的讨论现在开始！

- 代码演示

```
// 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);
    }
}

// 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

```
.method public hidebysig specialname rtspecialname
    instance void .ctor() cil managed
{
    // 代码大小      17 (0x11)
    .maxstack 8
    IL_0000: ldarg.0
    IL_0001: call     instance void [mscorlib]System.Object:::.ctor()
    IL_0006: nop
    IL_0007: nop
    IL_0008: ldarg.0
    IL_0009: ldc.i4.0
    IL_000a: stfld     int32 InsideDotNet.Type.TypeEssence.MyClass::_myNo
    IL_000f: nop
    IL_0010: ret
} // end of method MyClass:::.ctor
```

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. 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，以及简单的赋值操作，然后在内存和 IL 分析中发现其实质。

- 代码演示

```
public static void Main()
{
    // 内存分配于线程的堆栈上
    // 创建了值为等价"0"的实例
    MyStruct myStruct = new MyStruct();

    // 在线程的堆栈上创建了引用，但未指向任何实例
    MyClass myClass;
    // 内存分配于托管堆上
    myClass = new MyClass();

    // 在堆栈上修改成员
    myStruct.MyNo = 1;
    // 将指针指向托管堆
    myClass.MyNo = 2;

    myStruct.ShowNo();
    myClass.ShowNo();

    // 在堆栈上新建内存，并执行成员拷贝
    MyStruct myStruct2 = myStruct;
    // 拷贝引用地址
    MyClass myClass2 = myClass;

    // 在堆栈上修改 myStruct 成员
    myStruct.MyNo = 3;
    // 在托管堆上修改成员
    myClass.MyNo = 4;

    myStruct.ShowNo();
    myClass.ShowNo();
    myStruct2.ShowNo();
    myClass2.ShowNo();
```

}

- 内存实况

首先是值类型和引用类型的定义，这是一切对象的开始，如图 5-7 所示。

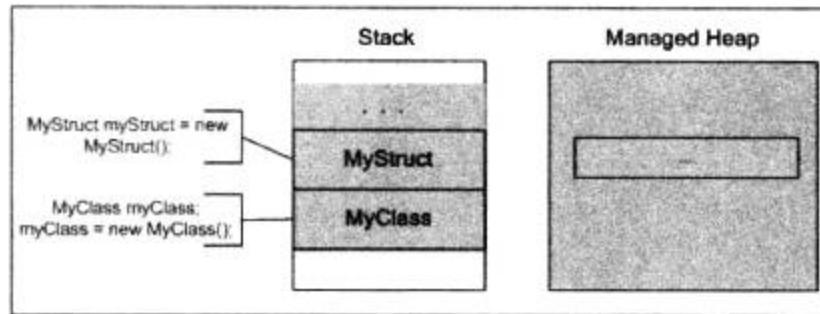


图 5-7 内存概况

然后是初始化过程，如图 5-8 所示。

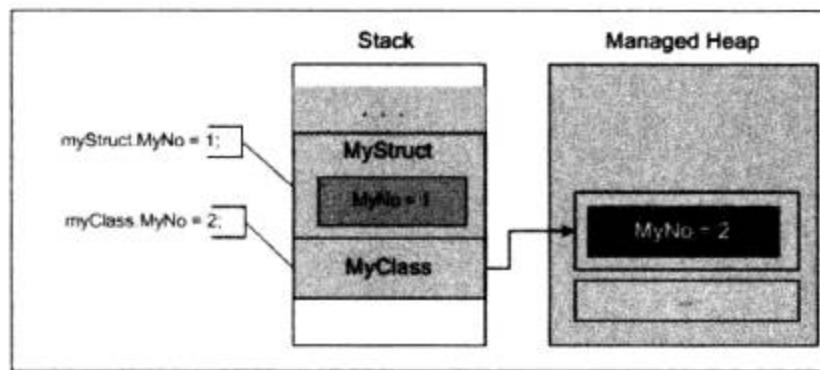


图 5-8 内存概况

简单的赋值和复制，是最基本的内存操作，不妨看看图 5-9。

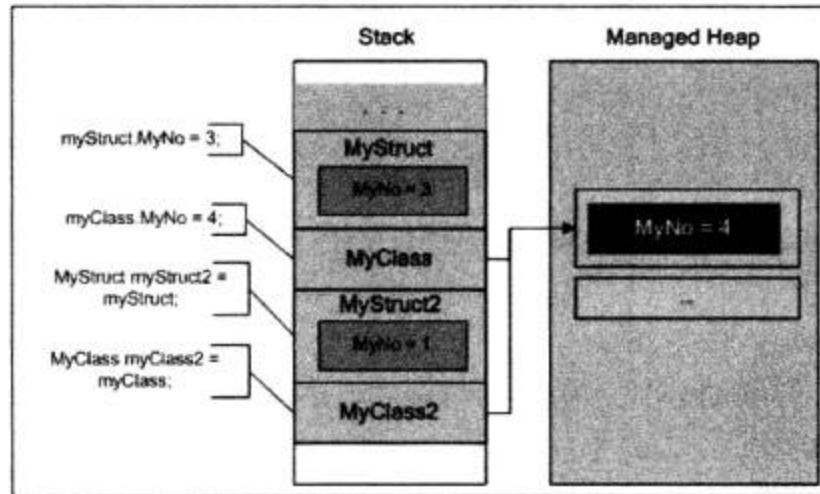


图 5-9 内存概况

3. 参数传递

- 代码演示

```
public class RefAndOut
{
    public static void Main()
    {
        //必须进行初始化，才能使用 ref 方式传递
        int x = 10;
```

```

    ValueWithRef(ref x);
    Console.WriteLine(x);

    // 使用 out 方式传递，不必初始化
    int y;
    ValueWithOut(out y);
    Console.WriteLine(y);

    object oRef = new object();
    RefWithRef(ref oRef);
    Console.WriteLine(oRef.ToString());

    object o;
    RefWithOut(out o);
    Console.WriteLine(o.ToString());
}

static void ValueWithRef(ref int i)
{
    i = 100;
    Console.WriteLine(i.ToString());
}

static void ValueWithOut(out int i)
{
    i = 200;
    Console.WriteLine(i.ToString());
}

static void RefWithRef(ref object o)
{
    o = new MyStruct();
    Console.WriteLine(o.ToString());
}

static void RefWithOut(out object o)
{
    o = new String('a', 10);
    Console.WriteLine(o.ToString());
}
}

```

不必多说，就是一个简要阐释，对于参数的传递将在5.3节“参数之惑——传递的艺术”中详述。

4. 类型转换

前文对类型转换已经做了详细的分析，在此我们只以自定义类型转换为例来做以说明。

- 代码演示

首先是值类型的自定义类型转换：

```

public struct MyStruct
{
    // 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
{
    // 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()
{
    // 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);
}
```

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。

- 代码演示

值类型判等示例:

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

引用类型判等示例:

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

然后是测试示例：

```
public static void Main()
{
    Console.WriteLine("类型判等---");
    // 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));
    // ReferenceEquals 认为 null 等于 null, 因此返回 true
    Console.WriteLine(ReferenceEquals(null, null));

    // 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));
}
```

6. 垃圾回收

首先，垃圾回收机制，很难以三言两语概言。本示例从简单的说明出发，对垃圾回收机制做以简要分析，目的是交代实例由创建到消亡的全过程。

- 代码演示

```
public static void Main()
{
    // 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;
}
```

- 内存实况（如图 5-10 所示）

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，在 6.3 节“垃圾回收”中将对此进行深入浅出的解析。

7. 总结陈述

这些示例主要从基础的方向入手来进行理论的探讨。技术的魅力正是在于千变万化，技术追求者的诉求却是从变化中寻求不变，我想这就是个好方法，本书希望提供这样一个入口，打开一个方法。

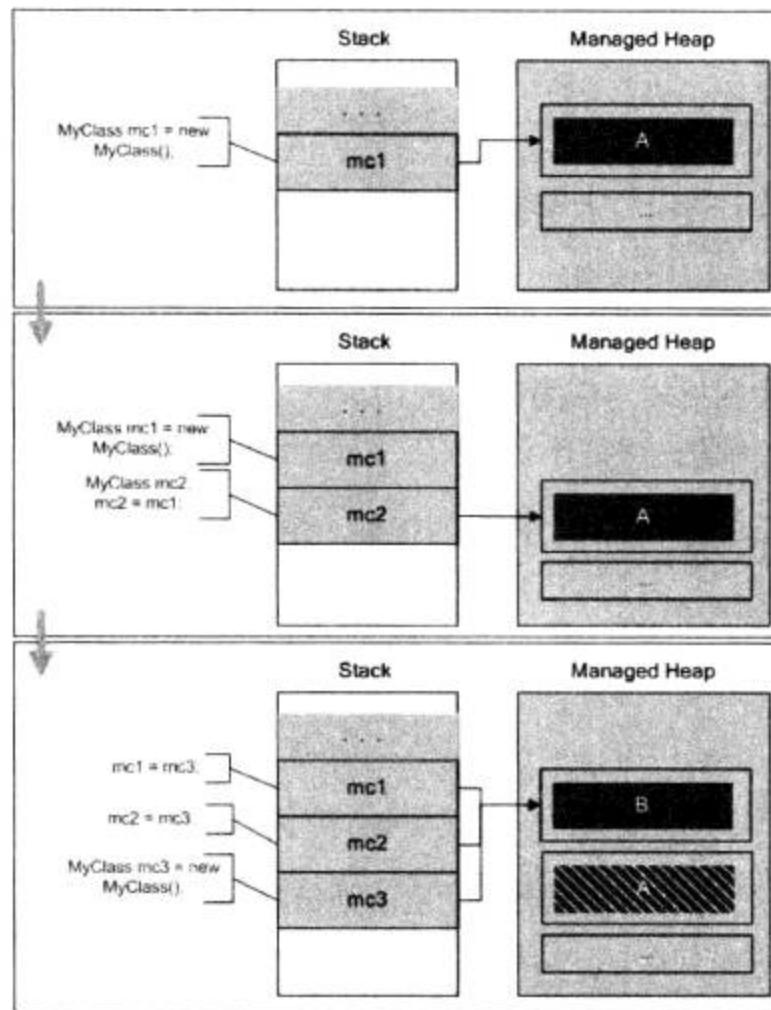


图 5-10 内存概况

5.2.8 结论

本节深入浅出，从内存分析到规则总结，进行了较为全面的论述，最后以实例分析进行深入巩固。本节的多个角度，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。

值类型和引用类型，要说的，要做的，还有很多。品味类型，为应用之路开辟技术基础。品味类型，继续探讨还有更多精彩。品味类型，我们以内存为要点撬开值类型和引用类型的规则和应用。

在此基础上，结合第6章“内存天下”的讨论，读者就能建立起相对清晰而又全面的理解，从而形成一个系统的把握尺度。对.NET的类型系统建立基本的宏观把握和底层认知，这是每个学习者在.NET领域的又一次“质的飞跃”。

5.3 参数之惑——传递的艺术

本节将介绍以下内容：

- 按值传递与按引用传递深论
- `ref` 和 `out` 比较
- 参数应用分析

5.3.1 引言

值传递和引用传递，常常是令我们产生误解和迷惑的地方，你是否对以下问题一清二楚呢？

- 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？

本节通过深入的讨论与分析，梳理出以上问题的正解，帮助我们绕开雷区，避免在实际的程序开发中被这些问题搞得狼狈不堪。

关于参数，如果不够清楚，且看本节一道来。

5.3.2 参数基础论

简单来说，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：

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

    //astr 是形式参数
    private static 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：

- (1) 形参和实参必须类型、个数与顺序对应匹配；
- (2) 参数列表可以为空，这种函数称为无参函数；
- (3) 解析 `Main(string [] args)`，Main 函数的参数可以为空，也可以为 string 数组，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；
- (4) 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，这点多少让习惯了 VB.NET 的开发者感到不爽。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)
        {
            //变相实现参数默认值“People”
            Console.WriteLine(name, "People");
        }
    }
}

```

这种方法可以扩展，可以通过重载更多函数实现多个默认参数方法。应该注意的是 VB.NET 可以通过 Optional 支持默认参数传递。

5.3.3 传递的基础

下面，重点将参数传递的基础做个交代，以便对参数之惑有一个从简入繁的化解过程。我们以条款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，参数传递是本节的核心内容，将在接下来的分析中详述。所以下面几个概念，我们仅做以简要介绍，主要包括：

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。在本书第 11 章将对泛型及泛型类型参数有所讨论。

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 params 修饰符。可变数目参数的好处就是在某些情况下可以方便地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，例如：

```

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 params 关键字实现可变数目参数的规则和使用，做以小结为：

- params 关键字的实质是：params 是定制特性 ParamArrayAttribute 的缩写（关于定制特性的详细论述请参见 8.3 节“历史纠葛：特性和属性”），该特性用于指示编译器的执行过程可以简化为：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，并将该数组作为参数传入调用的方法体。总之 params 用于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，我们可以很方便地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){//...}
```

- params 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，而数组是最简单的选择。
- params 修饰的参数数组，可以是任何类型。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。
- params 必须在参数列表的最后一个，并且只能使用一次。

5.3.4 深入讨论，传递的艺术

默认情况下，CLR 中的方法都是按值传递的，但是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需求下，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象下的实质所在。从而为开篇的几个问题给出清晰的答案。

根据参数类型和传递方式，有以下 4 种不同的情况，必须分别对其进行详细的分析，才能从比较中体味差别和实质：

- 值类型参数的按值传递
- 引用类型参数的按值传递
- 值类型参数的按引用传递
- 引用类型参数的按引用传递

下面将分别对其进行分析，按引用传递部分将合而为一在一个小节中以对比的方式来阐述。

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，被调用方法操作的是属于其本身实例的拷贝，因此不影响原来调用方法中的参数值。以例为证：

```
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);
    }
}
```

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，我们在此必须清楚地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，简单地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，都是传递“值”。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：

```
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，理解了它也就抓住了根源。关于值类型和引用类型的概念可以参考5.2节“品味类型——值类型与引用类型”，通过对值类型与引用类型的理解，能更加深入解开参数传递之惑的谜团。

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？

string本身为引用类型，因此从本节的分析中可知，对于形如

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

的传递形式，可以清楚地知道这是按值传递，也就是本节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？

```
class StringArgs
{
    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5-11所示。

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，先在一个新的地址（假设为0x29）生成一个string对象，该新对象的值为“Changing String”，引用地址为0x29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图5-12。

因此，执行结果就可想而知，我们从分析过程发现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，具体的分析希望读者通过接下来的按引用传递的揭秘之后，自行完成。

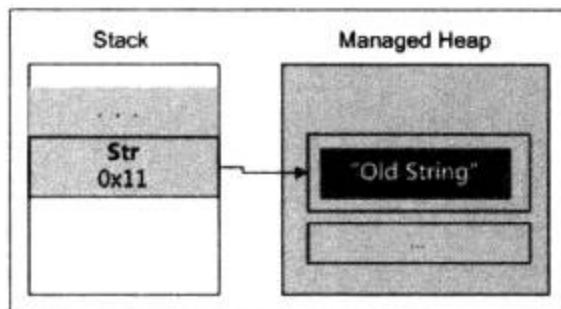


图 5-11 内存概况

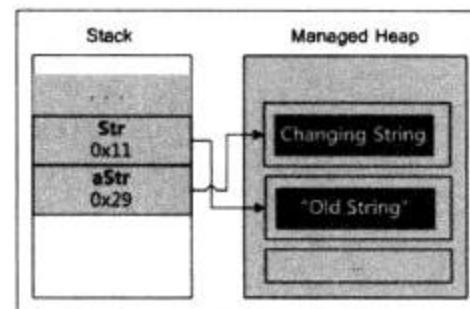


图 5-12 内存概况

3. 按引用传递之 ref 和 out

很多时候我们需要在方法内部创建新对象，并使原有的引用指向这个新对象，来实现对引用类型实例引用的地址的传递。因此，便引入了 `ref` 的概念，它用于标记这一操作并在编译时产生附加信息来记录这个地址。`ref` 解决了由于按值传递时改变引用副本而不影响引用本身的问题，传递的不是引用的副本，而是引用的引用。

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式的使用 `ref` 或者 `out`，否则将导致编译错误；
- CLR 允许通过 `out` 或者 `ref` 参数来重载方法，例如：

```
class Test_RefAndOut
{
    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，因此根据这一本质我们可以得出以下结论：

- 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：

```
class Test_Args
{
    static void Main()
    {
        int i = 100;
```

```

        string str = "One";
        ChangeByValue(ref i);
        ChangeByRef(ref str);
        Console.WriteLine(i);
        Console.WriteLine(str);
    }

    static void ChangeByValue(ref int iValue)
    {
        iValue = 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。

下面我们再进一步总结 ref 和 out 异同，就基本阐述清楚了按引用传递的精要所在：

- 相同点：从 CLR 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，CLR 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：

```

class Test_RefAndOut
{
    static void ShowInfo(string str)
    {
        Console.WriteLine(str);
    }

    static void ShowInfo(ref string str)
    {
        Console.WriteLine(str);
    }

    //ShowInfo 不能定义仅在 ref 和 out 上有差别的重载方法
    static void ShowInfo(out string str)
    {
        str = "Hello, anytao.";
        Console.WriteLine(str);
    }
}

```

上述示例将导致编译错误，编译器会提示：“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式初始化，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。

5.3.5 结论

完成了对值类型与引用类型的论述，相信这些知识的积累，将为我们奠定基础，从而进一步分享参数传

递的艺术，拨开层层迷雾。从探讨问题的角度来说，参数传递的种种误区其实植根于对值类型和引用类型的本质理解上。因此，完成对类型问题的探讨再进入参数传递的迷宫，我们才能轻松地游戏其中。也正是秉承这种思考与探索的方式，才能深入我们心中的.NET 圣殿。

5.4 皆有可能——装箱与拆箱

本节将介绍以下内容：

- 装箱与拆箱原理
- 值类型与引用类型的转换规则
- 类型转换的性能分析

5.4.1 引言

别吵了，别吵了。

是的，关于值类型与引用类型转换的吵闹声，总是不绝于耳。除了因为响亮的名字：装箱与拆箱，更因为这个讨论具有实在的意义。不信？那就请先欣赏几个看起来很简单的示例，从中体会装箱与拆箱的奥妙吧。

```
public static void Main()
{
    char A = 'a';
    object o = A;
    A = 'b';
    Console.WriteLine(A);
    Console.WriteLine(o);
}
```

问题：执行的结果如何？

```
public static void Main()
{
    int i = 100;
    object o = i;
    o = 200;
    Hashtable ht = new Hashtable();
    ht.Add("I", i);
    ht["I"] = o;
    Console.WriteLine(i + " is original, " + (int)ht["I"] + " is changing.");
}
```

问题：发生了几次装箱，几次拆箱？

```
public static void Main()
{
    float f = 1.01F;
    object o = f;
    double d = (double)o;
}
```

问题：运行期将抛出什么异常，为什么？

我们抛出了几个装箱与拆箱的普遍问题，带着思考和疑惑，在本节中从概念到应用来逐个解开答案。

5.4.2 品读概念

装箱与拆箱，正如它们的名字一样，重点在装和拆这两个动作上，而被操作的箱子又该如何呢？下面我们从宏观与微观两个方面来给出其定义：

宏观解释，从类型转换角度来说，装箱与拆箱是值类型与引用类型之间的桥梁，实现二者之间的自由转换，将类型系统统一化处理。具体来讲：

装箱与拆箱，就是值类型与引用类型的转换，装箱就是值类型数据转换为无类型的引用对象，使得我们可以将值类型视为对象来处理，通常这种转换主要指转换为 System.Object 类型或者该值类型实现的任何接口引用类型；而拆箱就是引用类型转换为值类型，通常伴随着从堆中复制对象实例的操作。

微观解释，从内存执行角度来看，值类型的内存分配于线程的堆栈上，而引用类型的内存分配于托管堆。所以从值类型向引用类型的转换，势必牵涉到数据的拷贝与指针引用等操作。具体来讲：

装箱操作，在托管堆中对栈中的值类型进行对象封装化处理，也就是生成一份值类型的对象引用副本，大致的过程为：在托管堆中分配新创建对象的内存，并将值类型的字段拷贝到该内存中，然后返回新对象的地址，这样就完成了将值类型转变为引用类型的过程；而拆箱操作，就是获取已装箱对象中来自值类型部分字段的地址。所以装箱和拆箱并非完全对称的互逆操作，拆箱在执行上并不包含字段的拷贝过程。

有了上面的解释，对装、拆操作以及箱子都有清楚的概念了吧。所谓的装就是在托管堆中创建对象并将值类型实例“装”给该对象，然后返回一把打开这个箱子的钥匙，也就是新对象的地址；拆就是获取箱子中原本属于值类型的指针，并将箱子中装入的数据拷贝给值类型对象的过程；而箱子当然就是托管堆了。所以，引用类型就无所谓装箱拆箱，因为引用类型本身就在“箱子”里。

概念雷区：

(1) 装箱的概念与拆箱的概念不是完全对等的互逆操作，从内存角度上来看，拆箱的性能开销远小于装箱，只是在实际的执行中，拆箱之后常常伴随着字段的拷贝，以 C# 为例而言，编译器总会自动产生拆箱后的字段拷贝。

(2) 只有被装过箱的对象才能被拆箱，而非所有的引用类型。将并非装箱而来的引用类型强制转换为值类型，将抛出 InvalidCastException 异常。

5.4.3 原理分拆

要了解清楚装箱与拆箱的原理，我们很有必要将值类型的特点作为分析的起点。值类型，提供了轻量型的数据结构，具有较少的内存开销，对系统性能有明显的作用。而其缺点也是明显的，就是缺少方法表指针，也就无法在期望 System.Object 或其继承类的方法上调用值类型。这显然是强大的 CLR 所不能容忍的，因此 CLR 提供了装箱与拆箱机制来实现类型系统的统一处理。下面，我们以一个最简单的装箱与拆箱示例来说明装箱与拆箱的执行过程，从内存角度来分析其执行细节，对其原理和性能分析都会有更好的理解。

```
public static void Main()
{
    Int32 x = 222;
```

```

object o = x;
Int32 y = (Int32)o;
}

```

1. 装箱过程解析

- 内存分配：在托管堆中分配内存空间，内存大小为欲装箱值类型的大小加上其他额外的内存空间，主要包括方法表指针和 SyncBlockIndex，这两个成员用于 CLR 管理引用类型对象，其作用在 6.2 节“对象创建始末”中有所讨论。
- 实例拷贝：将值类型的字段拷贝到新分配的内存中。
- 地址返回：将托管堆中的对象地址返回给新的引用类型。

装箱的三步走，可以理解为图 5-13。

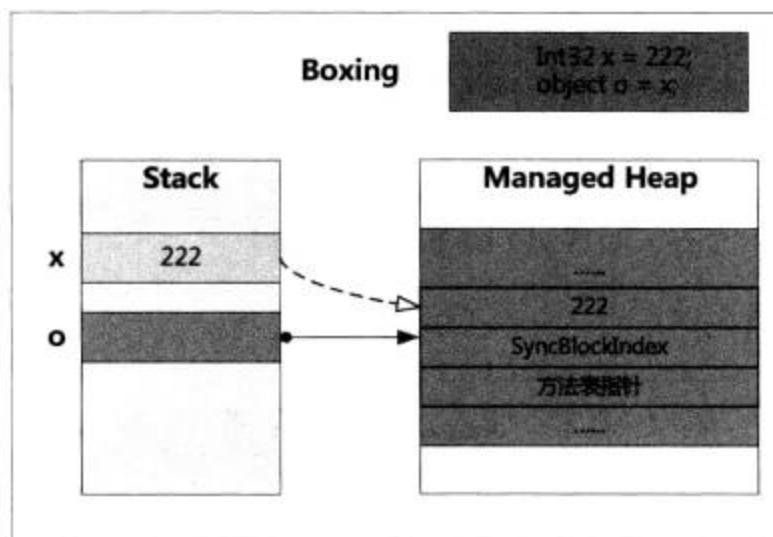


图 5-13 装箱过程内存概况

在 IL 中，装箱过程以 box 指令来实现，我们可以从上述代码的 IL 细节中了解到这一点：

```

.method public hidebysig static void Main() cil managed
{
    ...部分省略...
    IL_0006: stloc.0
    IL_0007: ldloc.0
    IL_0008: box           [mscorlib]System.Int32
    ...部分省略...
    IL_0015: ret
} // end of method BoxinDotNet::Main

```

由此而知，box 指令执行的正是我们分析的装箱三步走的操作过程，完成装箱的引用类型就拥有了方法表指针，一个值类型实例就此变成了一个引用类型对象。

2. 拆箱过程解析

- 实例检查：首先检查是否为 null，如果是则抛出 NullReferenceException 异常；如果不是则检查对象实例，确保它是给定值类型的装箱值，并且保证拆箱后的类型为原来的同一类型，否则会抛出 InvalidCastException 异常。

提示：

在代码中，要显式地判断装箱后的引用类型的原始类型，可以有以下几种方式进行判断：

is 操作符判断

```
if (o is Int32)
    Console.WriteLine("It's Int32 type.");
```

GetType 方法判断

```
Console.WriteLine(o.GetType());
```

- 指针返回：返回已经装箱对象中属于原值类型部分字段的地址。而附加成员：方法表指针与 SyncBlockIndex 对该指针是不可见的。

注意

在.NET 中，实际的拆箱操作过程仅包括上述两个过程，因此装箱与拆箱并非对称的互逆操作。但是，在通常情况下，拆箱操作之后一般都紧跟着字段拷贝操作，例如在 C# 中拆箱之后总会进行字段的拷贝。在本节中我们将字段拷贝作为拆箱的一个过程，但是严格意义上的概念区别我们应该有所了解。因此，CLR 提供了 unbox 和 unbox.any 两个指令来执行拆箱过程，道理就在于此。

- 字段拷贝：将托管堆中实例的字段拷贝到线程的堆栈中。

拆箱三步走，可以理解为图 5-14。

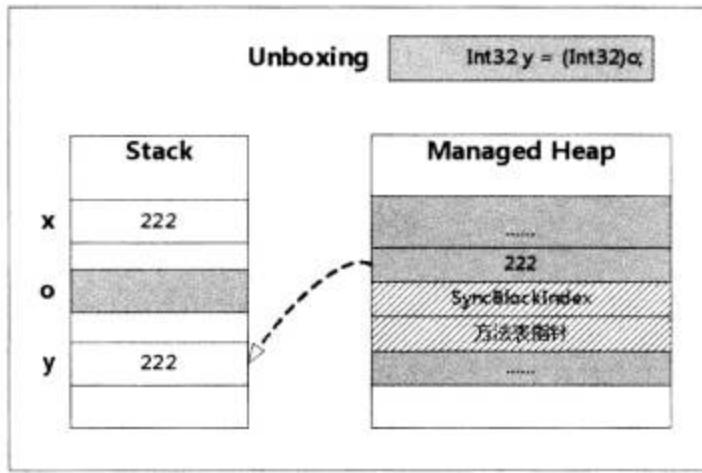


图 5-14 拆箱过程内存概况

不同于装箱过程，CLR 提供了两个指令：unbox 指令和 unbox.any 来执行拆箱操作。我们首先来了解上述代码的 IL 细节：

```
.method public hidebysig static void Main() cil managed
{
    ...部分省略...
    IL_000d:  stloc.1
    IL_000e:  ldloc.1
    IL_000f:  unbox.any  [mscorlib]System.Int32
    IL_0014:  stloc.2
    IL_0015:  ret
} // end of method BoxinDotNet::Main
```

从 IL 中可知，在此以 unbox.any 指令来执行拆箱操作，那么 unbox 和 unbox.any 的关系又如何呢？unbox 指令用于返回已经装箱对象中属于原值类型部分字段的地址，执行后堆栈中保留的是一个到拆箱的数据结构指针；unbox.any 指令则用于托管堆中的实际字段值复制到堆栈中，执行后堆栈中保留的是实际的值。正如前文所述，在 C# 中拆箱之后总会执行字段拷贝操作，因此 C# 中只用到 unbox.any 指令。

3. 规则

- 类型一致，拆箱必须保证执行后的结果是原来未装箱时的类型，否则将抛出 InvalidCastException 异常。
- 装箱与拆箱主要是针对值类型而言的，引用类型总是以装箱形式存在的。
- 装箱和拆箱分为显式转换和隐式转换两种情况，在实际的编码中应该警惕隐式转换带来的性能与异常问题。

5.4.4 还是性能

装箱与拆箱之所以引起过多的关注，一个根本原因恐怕就是性能问题了，基于性能而展开对装箱与拆箱问题讨论是非常必要的。装箱与拆箱对性能有极大的影响，源于其在执行速度和内存拷贝两个方面的浪费，因此软件设计者应尽量减少这种装与拆的发生，那么通常情况下该如何做呢，这是本节要讨论的主题。

(1) 很多.NET 的基本方法都提供了数个重载方法，这些方法大都是基于值类型的，其原因显然是减少装箱带来的性能损失。例如 System.Console.WriteLine()、System.Text.StringBuilder.Append() 等。因此在实际的项目中应该留意发生隐式装箱的可能，并提供相应的多个重载方法来避免装箱的发生。

(2) 装箱与拆箱经常是以隐式发生的，因此为程序带来不可估计的性能损失，在系统中显式的实现装箱操作，是提高性能的较好选择，例如：

```
public static void Main()
{
    Hashtable ht = new Hashtable();
    int x = 100;
    //发生1次装箱
    object o = x;
    for (int i = 0; i < 100; i++)
    {
        //发生100次装箱
        ht.Add(i, o);
        //发生200次装箱
        ht.Add(i, x);
    }
}
```

虽然没有人愿意写出这种性能丑陋的代码，但是像这种不经意的错误确实时常光顾，给系统性能带来极大的伤害。另一方面，装箱与拆箱必然造成多余的对象，给 GC 造成额外负担，除了性能的损失还会引起某些隐藏性的 Bug。

(3) 泛型

正是源于装箱与拆箱的弊端，在.NET 2.0 中引入的泛型技术，在很大程度上解决了值类型转换为引用类型造成的装箱与拆箱之弊。关于 CLR 2.0 中托管泛型的理解，我们将在后文第 11 章“接触泛型”中有详细的论述，在此还是着眼于泛型技术对减少装箱与拆箱的有力改善。行胜于言，还是以实例来比较其优劣吧。

首先是集合转换引起的典型装箱操作：

```
public static void Main()
{
    ArrayList array = new ArrayList();
    array.Add(1);
```

```

    int i = (int)array[0];
}

```

显然，上述示例完成了一次装箱一次拆箱。

然后，以泛型集合来实现相同的功能，又该是什么样的结果呢？

```

public static void Main()
{
    List<int> list = new List<int>();
    list.Add(1);
    int i = list[0];
}

```

当然，通过IL分析来了解真相是最可靠的验证，通过泛型集合实现的操作反编译为IL代码为：

```

.method public hidebysig static void Main() cil managed
{
    .entrypoint
    // 代码大小      24 (0x18)
    .maxstack 2
    .locals init ([0] class [mscorlib]System.Collections.Generic.List`1<int32> list,
                 [1] int32 i)
    IL_0000: nop
    IL_0001: newobj     instance void class [mscorlib]System.Collections.Generic. List`1
<int32>:::ctor()
    IL_0006: stloc.0
    IL_0007: ldloc.0
    IL_0008: ldc.i4.1
    IL_0009: callvirt   instance void class [mscorlib]System.Collections.Generic. List`1
<int32>::Add(!0)
    IL_000e: nop
    IL_000f: ldloc.0
    IL_0010: ldc.i4.0
    IL_0011: callvirt   instance !0 class [mscorlib]System.Collections.Generic. List`1
<int32>::get_Item(int32)
    IL_0016: stloc.1
    IL_0017: ret
} // end of method BoxinDotNet::Main

```

上述示例中如果集合的数据量很大时，对性能的体现就非常明显了。通过简单的比较可知，泛型有效减少了装箱与拆箱的发生，大大提高了系统的性能与稳定。至于，泛型技术如何解决了值类型与引用类型转换的装箱操作本质，并非本节的范畴，在泛型部分我们将详细做以交代。

5.4.5 重在应用

1. .NET中的典型装箱与拆箱

(1) ArrayList

ArrayList是Array的复杂版本，是一种动态数组，其容量可以动态扩充。在ArrayList内部封装了一个System.Object类型的数组，对值类型数组来说ArrayList操作常常伴随着装箱操作，从而带来性能的影响，例如Add、AddRange、Insert方法都是接受引用类型参数。因此，通常情况下我们应该优先考虑使用Array而非ArrayList来操作值类型数组，当然ArrayList也提供了Array不能比拟的其他特性，所以在选择上应酌情而定。

(2) Hashtable

下面是Hashtable最常见的操作。

```

public static void Main()
{
    int i = 100;
    Hashtable ht = new Hashtable();
    ht.Add(1, i);
    ht.Add(2, i + 1);
    ht.Remove(1);
}

```

在上述示例中，先后发生了5次装箱操作，`Hashtable`类型的`Add`方法和`Remove`方法都接受`System.Object`类型的参数，因此在使用`Hashtable`类型时，应该关注其传入参数引起的装箱操作。这种类似的情况，广泛存在于.NET中，有待我们在实际的应用中特别关注。

(3) 枚举

一个典型的.NET值类型就是枚举，所以肯定会涉及装箱与拆箱的问题，枚举类型可以被装箱为`System.Object`、`System.ValueType`和`System.Enum`，以及`System.Enum`实现的三个接口类型`System.IComparable`、`System.IConvertible`、`System.IFormattable`。例如有如下的代码：

```

enum Color
{
    Red,
    Black,
    Blue
}

public static void Main()
{
    Color red = Color.Red;
    IComparable comparer = red;
    IFormattable formater = red;
    IConvertible converter = red;
    Console.WriteLine(red.ToString());
    Console.WriteLine(red);
}

```

从其IL代码中，我们会看得更清楚：

```

.method public hidebysig static void Main() cil managed
{
    .entrypoint
    // 代码大小      54 (0x36)
    .maxstack 1
    .locals init ([0] valuetype InsideDotNet.Type.Box.BoxinDotNet/Color red,
                 [1] class [mscorlib]System.IComparable comparer,
                 [2] class [mscorlib]System.IFormattable formater,
                 [3] class [mscorlib]System.IConvertible converter)
    IL_0000:  nop
    IL_0001:  ldc.i4.0
    IL_0002:  stloc.0
    IL_0003:  ldloc.0
    IL_0004:  box      InsideDotNet.Type.Box.BoxinDotNet/Color
    IL_0009:  stloc.1
    IL_000a:  ldloc.0
    IL_000b:  box      InsideDotNet.Type.Box.BoxinDotNet/Color
    IL_0010:  stloc.2
    IL_0011:  ldloc.0
    IL_0012:  box      InsideDotNet.Type.Box.BoxinDotNet/Color
    IL_0017:  stloc.3
    IL_0018:  ldloc.0

```

```

IL_0019:  box      InsideDotNet.Type.Box.BoxinDotNet/Color
IL_001e:  callvirt  instance string [mscorlib]System.Object::ToString()
IL_0023:  call      void  [mscorlib]System.Console::WriteLine(string)
IL_0028:  nop
IL_0029:  ldloc.0
IL_002a:  box      InsideDotNet.Type.Box.BoxinDotNet/Color
IL_002f:  call      void  [mscorlib]System.Console::WriteLine(object)
IL_0034:  nop
IL_0035:  ret
} // end of method BoxinDotNet::Main

```

可见，共计发生了5次装箱操作，前三次为隐式装箱，转换为对应的接口类型，而后两次完成了两次向引用类型的装箱。

2. 关注不经意的隐式转换

由于装箱与拆箱对性能的影响，在此特别强调值类型使用时应该留意其隐式转换，并尽可能以显式方式来实现，下面的示例揭示了这种经常性的影响，希望引起读者的思考。

```

public static void Main()
{
    int i = 100;
    //执行装箱
    i.GetType();
    //未执行装箱
    i.ToString();
    //进行装箱，避免发生隐式转换
    object o = i;
    Hashtable ht = new Hashtable();
    ht.Add("One", o);
    ht.Add("Two", o);
}

```

分析上述示例可知，`GetType`方法由`System.Object`类型提供，因此值类型调用时必然执行相应的装箱操作；而`ToString`方法则由`int`类型重写，因此不会装箱；接下来`Hashtable`的`Add`方法接受`System.Object`类型的参数，因此通过显式的类型转换来减少隐式的装箱操作。

5.4.6 结论

我们在值类型与引用类型分析的基础上，将类型系统中最为重要的主题：装箱和拆箱，作为重点进行了一番深入浅出的论述，并将核心放在其对性能的影响和应用层面上。这对全面的理解和把握.NET的类型系统是非常重要的，也从另一个角度更加透明地扩展了对值类型与引用类型的认知。

参考文献

Jeffrey Richter, Applied Microsoft .NET Framework Programming

Patrick Smacchia, Practical .NET2 and C#2

David Chappell, Understanding .NET

Jacquie Barker, Grant Palmer, Beginning C# Objects From Concepts to Code

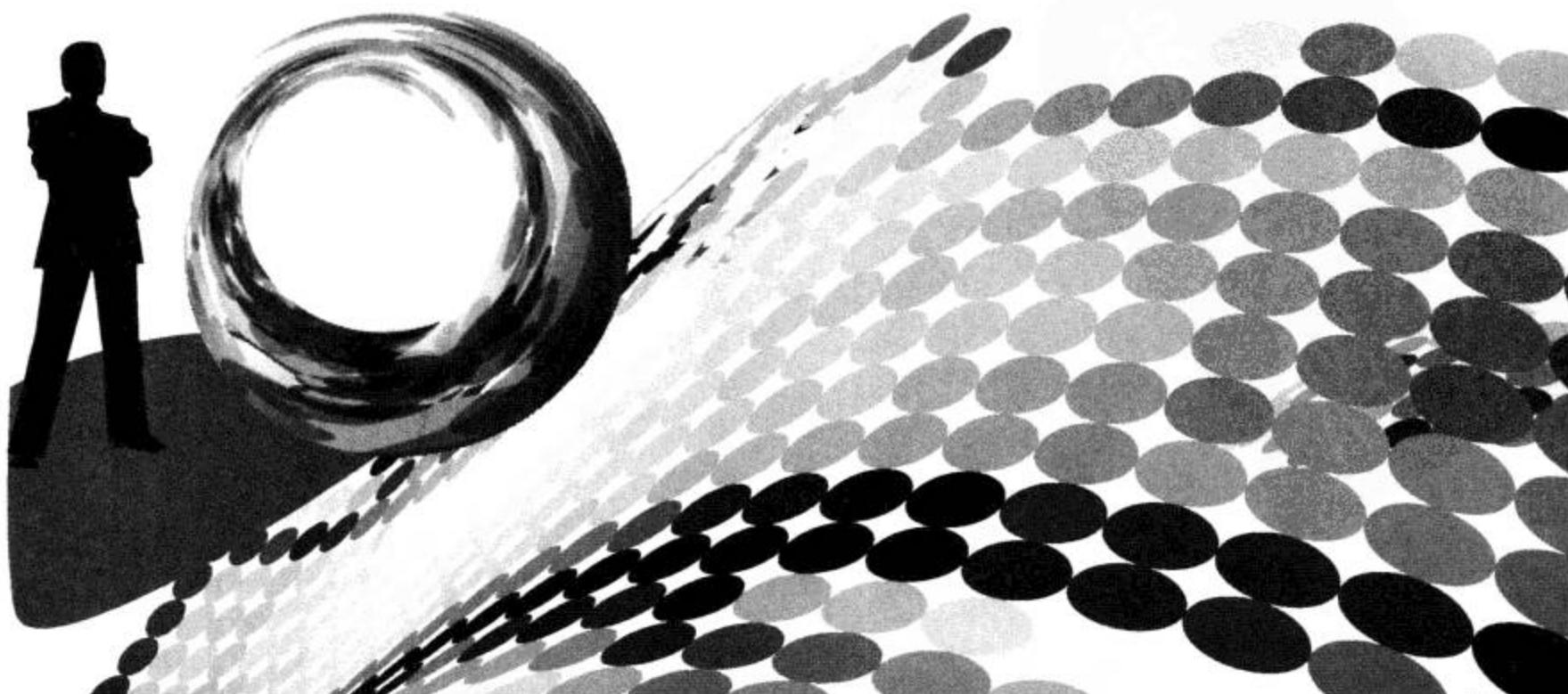
Bill Wagner, Effective C#

NeoRAGE2002's Weblog, 关于 CLR 2.0 中托管泛型的理解,

<http://neoragex2002.cnblogs.com/archive/2005/12/19/300475.html>

第6章 内存天下

- 6.1 内存管理概要 / 185
 - 6.1.1 引言 / 185
 - 6.1.2 内存管理概观要论 / 185
 - 6.1.3 结论 / 186
 - 6.2 对象创建始末 / 186
 - 6.2.1 引言 / 187
 - 6.2.2 内存分配 / 187
 - 6.2.3 结论 / 193
 - 6.3 垃圾回收 / 193
 - 6.3.1 引言 / 193
 - 6.3.2 垃圾回收 / 193
 - 6.3.3 非托管资源清理 / 197
 - 6.3.4 结论 / 204
 - 6.4 性能优化的多方探讨 / 204
 - 6.4.1 引言 / 204
 - 6.4.2 性能条款 / 204
 - 6.4.3 结论 / 210
- 参考文献 / 211



6.1 内存管理概要

本节将介绍以下内容：

- 内存管理的基本内容
- 对象资源访问的过程简述

6.1.1 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。

6.1.2 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 6.2 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 6.2 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 6.3 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 6.3 节“垃圾回收”中，将对垃圾回收机制作以详细的讨论与分析。

6.1.3 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。

6.2 对象创建始末

本节将介绍以下内容：

- 对象的创建过程
- 内存分配分析
- 内存布局研究

6.2.1 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。

6.2.2 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5.2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4.5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 6-1 所示。

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 6-2 所示。

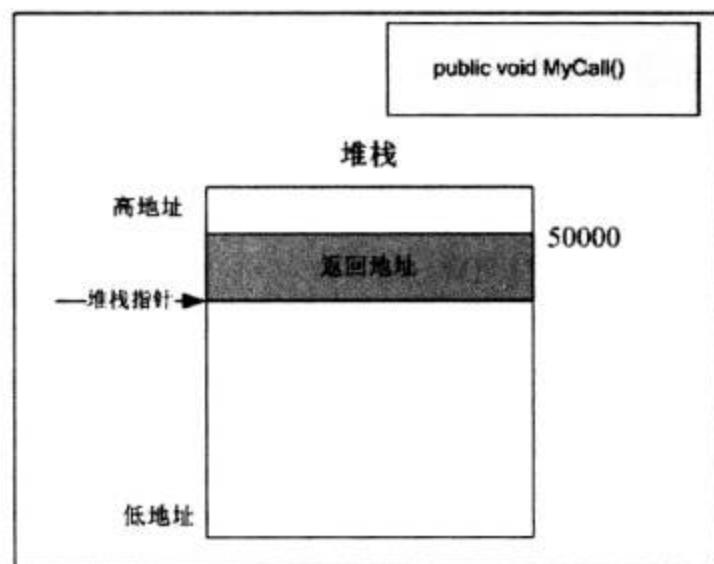


图 6-1 栈上的内存分配

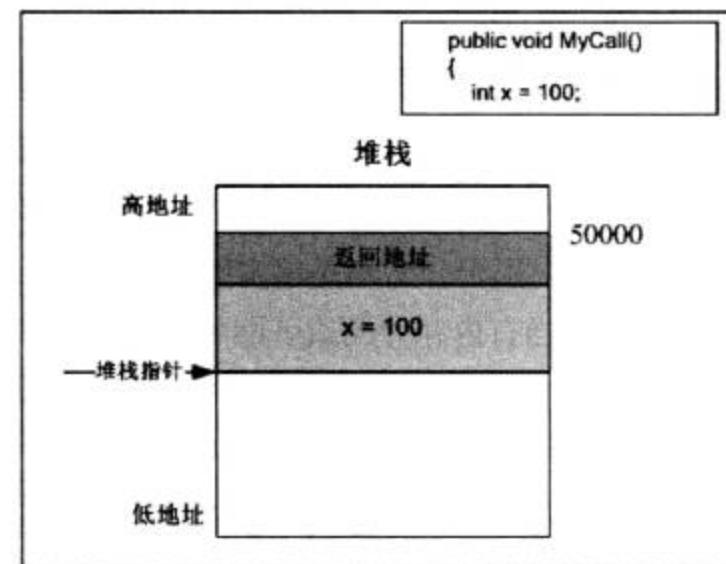


图 6-2 栈上的内存分配

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 6-3 所示。

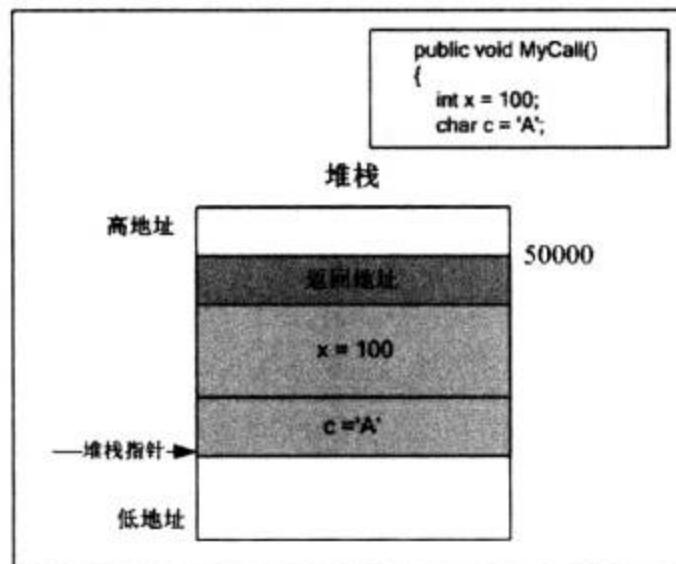


图 6-3 栈上的内存分配

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 6-4 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。

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，栈内是后进先出（LIFO）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。

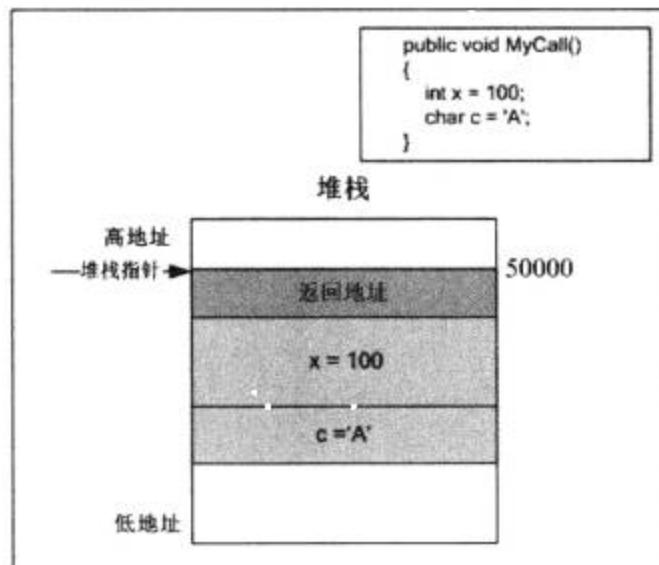


图 6-4 栈上的内存分配

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。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 User()
    {
        id = 4;
        user = new UserInfo();
    }
}

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：

① 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 36Byte，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，占 4Byte。

② 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 8Byte（在 32 位 CPU 平台下）。因此，需要在托管堆上分配的字节总数为 17Byte，而堆上的内存块总是按照 4Byte 进行内存对齐，因此本例中将补齐到 20Byte 的地址空间。

另外，以本例而言，还应注意的是，实际分配的内存空间还包括 UserInfo 类型创建的实例大小，其计算方法类似于 VIPUser 的计算过程。一般而言 UserInfo 的实例化是在 User 类型构造函数中完成的，所以在 GC 堆上分配的空间还包括 16Byte 的成员实例内存，共计 36Byte。

③ CLR 在当前 AppDomain 对应的托管堆上搜索，找到一个未使用的 36 Byte 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 36 个 Byte，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：

① 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。

② 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。

③ 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 8.8 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，然而该过程只能看做是一个简化的描述，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 6-5。

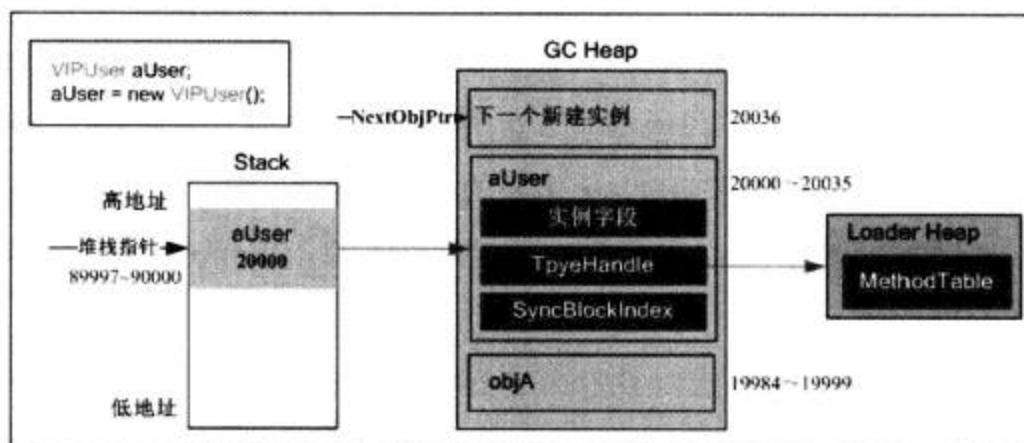


图 6-5 堆上的内存分配

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：

- 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 5.2 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。

- 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，如果子类覆盖了父类方法，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 1.2 节“什么是继承”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 8.8 节“动静之间：静态和非静态”有所讨论。

6.2.3 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。

6.3 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

6.3.1 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。

6.3.2 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 6-6。

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。

- 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。

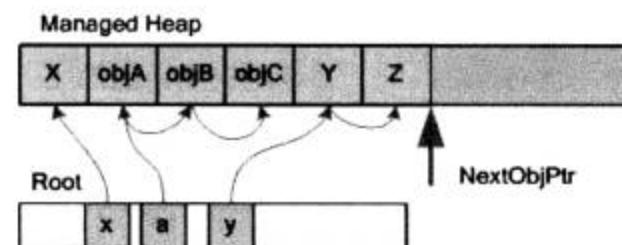


图 6-6 垃圾收集执行前的托管堆

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。

- 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 6-7。

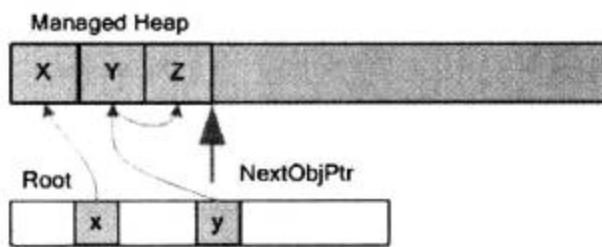


图 6-7 垃圾收集执行后的托管堆

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。

1. 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 6-8 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 6-8 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 6-9 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，从而有效地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。

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6-10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。

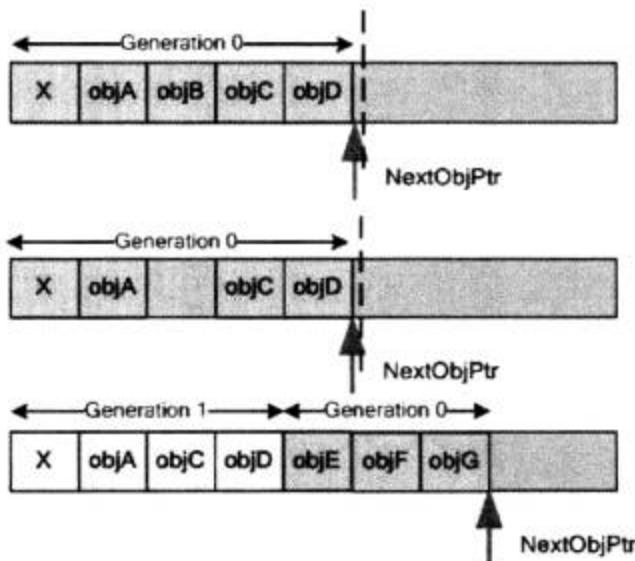


图6-9 初次执行垃圾回收

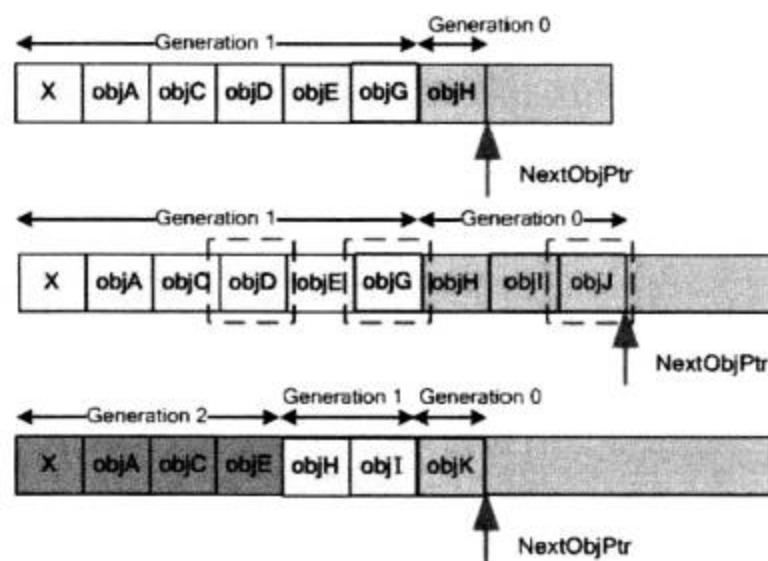


图6-10 执行1代对象垃圾回收

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。

2. 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。
- 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。
- 最好不要通过调用 `GC.Collect` 来强制执行垃圾收集。
- 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。
- 对“胖”对象考虑使用弱引用，以提高性能，详见 6.4 节“性能优化的多方探讨”。

6.3.3 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盖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，事实上显式的重写会引发编译时错误，只需实现析构函数即可。

在具体操作上，终结器的工作原理是这样的：在 System.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 System.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 Finalize 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# 中无法显式的重写 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，也不能被显式调用，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 Dispose 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 7.3 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。

6.3.4 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。

6.4 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

6.4.1 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。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。

6.4.2 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 6.3 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，分别为不同的处理器而设计：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 6.3 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着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 8.9 节“集合通论”和第 11 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 9.3 “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 9.3 “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 9.1 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。在此做一个简单的演示：

```
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 9.9 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 struct 和 class 的比较，详见 8.2 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8.5“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8.1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。

关于装箱和拆箱的详细讨论，参见5.4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4.5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6-11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。

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。

6.4.3 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。

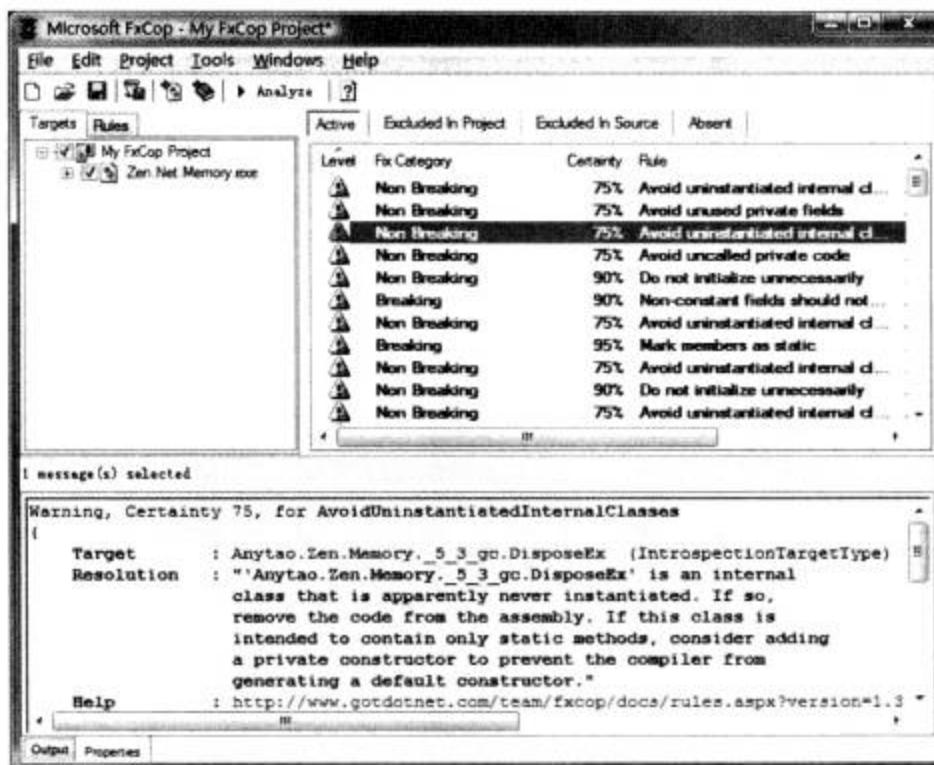


图 6-11 FxCop 代码分析工具

参考文献

Richard Jones, Rafael D Lins, Garbage Collection: Algorithms for Automatic Dynamic Memory Management

Joe Duffy, Professional .NET Framework 2.0

Jeffrey Richter, Applied Microsoft .NET Framework Programming

Christian Nagel, Bill Evjen, Jay Glynn, Professional C# 2005

Emmanuel Schanzer, Performance Considerations for Run-Time Technologies in the .NET Framework,
<http://msdn2.microsoft.com/en-us/library/ms973838.aspx>

Gregor Noriskin, Tips and Tricks for Performance in .NET Applications,

<http://msdn2.microsoft.com/en-us/library/ms973839.aspx>

FxCop Down, <http://www.gotdotnet.com/team/fxcop/>



学习笔记

第3部分

格局——.NET 面面俱到

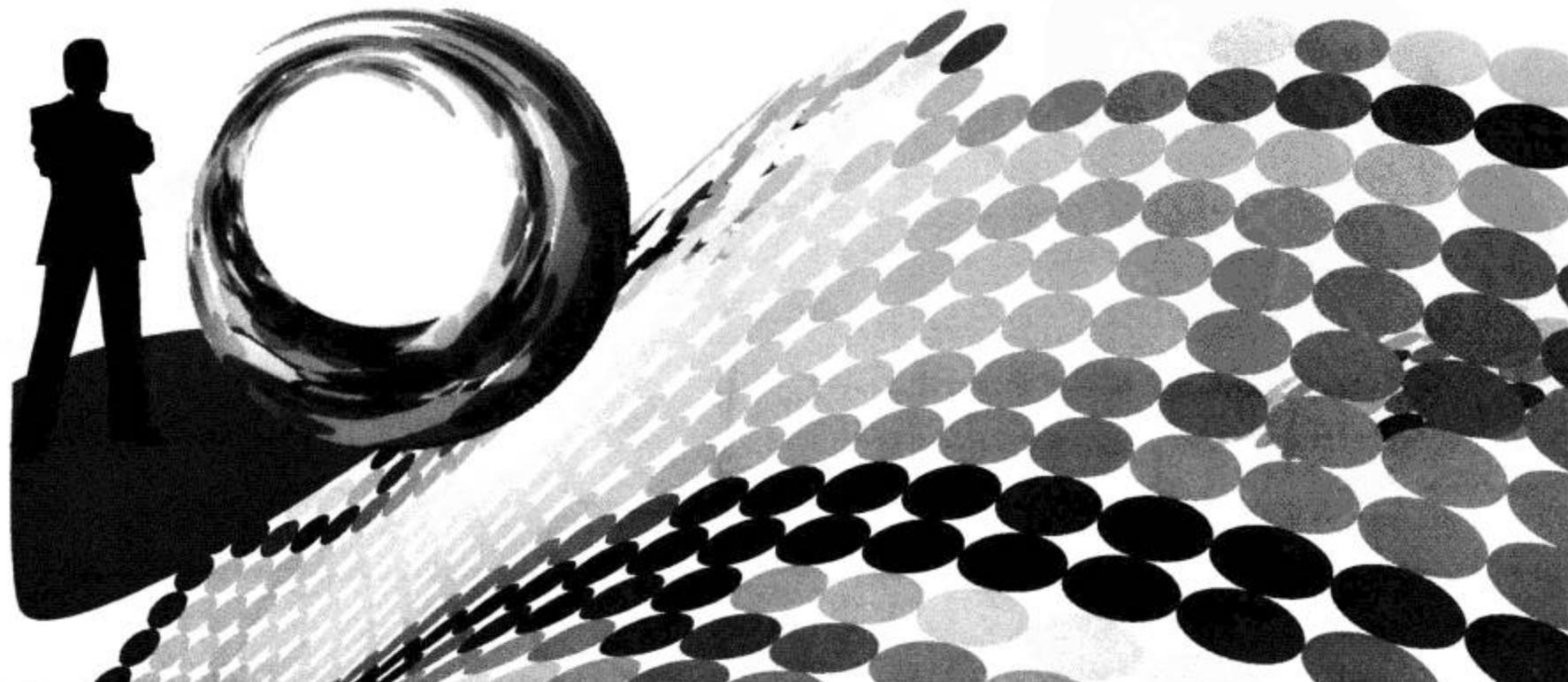
格局代表着宏观，对.NET 的深入浅出要把握切入和深入的方法，技术探讨应该坚持自上而下的展开方式。本部分从宏观出发，在.NET 的大格局下，把触角深入到.NET Framework 的各个方面，从而在复杂的交互中攫取最容易吸收的营养。关键字部分将重点讨论.NET 中使用最频繁，理解最不甚的关键字进行深入挖掘，揭开其秘密；巅峰对决则以独特的视角，来把.NET 技术概念中最容易混淆和误解的概念，单独拿出来 PK，在一正一反的较量中发现其本质，体会其差别，总结其应用；本来面目则从技术基本概念角度出发，进一步阐释.NET 的技术精华，讨论的基线不是简单的概念品评，而是深入到底层来剖析本源；格局之选，从宏观结构出发，请.NET 框架中的经典命名空间粉墨登场，提纲挈领式的介绍各个典型命名空间在.NET 大格局中的作用和功能，为读者打开第一次亲密接触的机会。

本部分主要包括：

- 第7章 深入浅出——关键字的秘密
- 第8章 巅峰对决——走出误区
- 第9章 本来面目——框架诠释
- 第10章 格局之选——命名空间剖析

第7章 深入浅出——关键字的秘密

7.1 把 new 说透 / 215	7.4.6 结论 / 238
7.1.1 引言 / 215	7.5 转换关键字 / 238
7.1.2 基本概念 / 215	7.5.1 引言 / 239
7.1.3 深入浅出 / 217	7.5.2 自定义类型转换探讨 / 239
7.1.4 结论 / 219	7.5.3 本质分析 / 240
7.2 base 和 this / 219	7.5.4 结论 / 242
7.2.1 引言 / 219	7.6 预处理指令关键字 / 242
7.2.2 基本概念 / 219	7.6.1 引言 / 242
7.2.3 深入浅出 / 220	7.6.2 预处理指令简述 / 242
7.2.4 通用规则 / 224	7.6.3 #if、#else、#elif、#endif / 243
7.2.5 结论 / 224	7.6.4 #define、#undef / 244
7.3 using 的多重身份 / 224	7.6.5 #warning、#error / 244
7.3.1 引言 / 224	7.6.6 #line / 245
7.3.2 引入命名空间 / 225	7.6.7 结论 / 245
7.3.3 创建别名 / 225	7.7 非主流关键字 / 245
7.3.4 强制资源清理 / 227	7.7.1 引言 / 245
7.3.5 结论 / 230	7.7.2 checked/unchecked / 246
7.4 认识全面的 null / 230	7.7.3 yield / 247
7.4.1 引言 / 230	7.7.4 lock / 250
7.4.2 从什么是 null 开始 / 230	7.7.5 unsafe / 252
7.4.3 Nullable<T> (可空类型) / 232	7.7.6 sealed / 253
7.4.4 ??运算符 / 234	7.7.7 结论 / 254
7.4.5 Null Object 模式 / 235	参考文献 / 254



7.1 把 new 说透

本节将介绍以下内容：

- 面向对象的基本概念
- 深入浅出 new 关键字
- 对象创建的内存管理

7.1.1 引言

我们有必要将一个关键字拿出来长篇大论的综述吗？回答的关键是：你真的理解了 new 吗？如果是，那请略过此节，如果不是，那请一定认真对待。

下面几个问题，可以大概考察你对 new 的掌握，开篇之前，先做个检验：

- 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？
- 泛型中，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？

本节将给出答案。

7.1.2 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。

这是本节的重点内容，我们将在下文中详细剖析。

- 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，这主要是针对版本控制而言，将基类方法实现为 virtual 能够同时保证向前扩展和向后兼容，在派生类中通过 new 或 override 进行灵活控制；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：

```
class Number
{
    public static int i = 123;
```

```
public virtual void ShowInfo()
{
    Console.WriteLine("base class---");
}

public virtual void ShowNumber()
{
    Console.WriteLine(i.ToString());
}
}

class IntNumber : Number
{
    new public static int i = 456;

    public new virtual void ShowInfo()
    {
        Console.WriteLine("Derived class---");
    }

    public override void ShowNumber()
    {
        Console.WriteLine("Base number is {0}", Number.i.ToString());
        Console.WriteLine("New number is {0}", i.ToString());
    }
}

class Test_Number
{
    public static void Main()
    {
        Number num = new Number();
        num.ShowNumber();
        IntNumber intNum = new IntNumber();
        intNum.ShowNumber();
        intNum.ShowInfo();

        Number number = new IntNumber();
        //究竟调用了谁?
        number.ShowInfo();
        //究竟调用了谁?
        number.ShowNumber();
    }
}
```

- 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。关于多态的了解详见 1.4 节“多态的艺术”的论述。

7.1.3 深入浅出

作为修饰符和约束的情况，不是很难理解的话题，正如我们看到本节开篇提出的问题，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。

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，来了解一下 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 在创建对象时做了什么？

```
class MyClass
{
    private int _id;

    public MyClass(int id)
    {
        _id = id;
    }
}

struct MyStruct
{
    private string _name;

    public MyStruct(string name)
    {
        _name = name;
    }
}
```

```

class NewReflecting
{
    public static void Main()
    {
        int i;
        int j = new int();
        MyClass mClass = new MyClass(123);
        MyStruct mStruct = new MyStruct("My Struct");
    }
}

```

使用 Reflector 工具反编译产生的 IL 代码如下：

```

.method public hidebysig static void Main() cil managed
{
    .entrypoint
    .maxstack 2
    .locals init (
        [0] int32 i,
        [1] int32 j,
        [2] class InsideDotNet.Keyword.New.MyClass mClass,
        [3] valuetype InsideDotNet.Keyword.New.MyStruct mStruct)
L_0000: nop
//初始化 j 为 0
L_0001: ldc.i4.0
L_0002: stloc.1
//使用 newobj 指令创建新的对象，并调用构造函数以 0x76 (123 的 16 进制) 初始化
L_0003: ldc.i4.s 0x7b
L_0005: newobj instance void InsideDotNet.Keyword.New.MyClass::ctor(int32)
L_000a: stloc.2
//加载"My Struct"
L_000b: ldloca.s mStruct
L_000d: ldstr "My Struct"
//调用构造函数执行初始化
L_0012: call instance void InsideDotNet.Keyword.New.MyStruct::ctor(string)
L_0017: nop
L_0018: ret
}

```

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，并且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。类型创建的详细分析请参见 4.5 节“经典指令解析之实例创建”。

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 strA = new string('*', 100);
string strB = new string(new char[] { 'a', 'b', 'c' }));
```

而不是

```
string strC = "*****";
```

7.1.4 结论

能说的就这么多了，作为基本的原理和应用，对大部分的需求是满足了。希望以上介绍，能对你分享 new 关键字和其本质的来龙去脉有所帮助。

言归正传，开篇的几个题目，相信我们已经有了答案，面对技术，我们应该畅所欲言，这种讨论可以使理解更深入，不是吗？

7.2 base 和 this

本节将介绍以下内容：

- 面向对象的基本概念
- base 关键字深入浅出
- this 关键字深入浅出

7.2.1 引言

继续关键字话题的讨论，本节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，但是还是有值得我们深入讨论的地方。

老办法，先将问题罗列开来，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？
- base 可以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 指向哪一层呢？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？

这些问题，请在本节寻找答案。

7.2.2 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。

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 小结其具体功能包括：

- 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. 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 小结其具体功能主要包括：

- 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。

7.2.3 深入浅出

1. 示例为上

下面以一个小示例来进行综合的说明，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，具体如下：

```
public class Action
{
    public static void ToRun(Vehicle vehicle)
    {
        Console.WriteLine("{0} is running.", vehicle.ToString());
    }
}

public class Vehicle
{
    private string name;
    private int speed;
    private string[] array = new string[10];

    public Vehicle()
    {
    }

    //限定被相似的名称隐藏的成员
    public Vehicle(string name, int speed)
    {
        this.name = name;
        this.speed = speed;
    }

    public virtual void ShowResult()
    {
    }
}
```

```

        Console.WriteLine("The top speed of {0} is {1}.", name, speed);
    }

    public void Run()
    {
        //传递当前实例参数
        Action.RunWith(this);
    }

    //声明索引器，必须为 this，这样就可以像数组一样来索引对象
    public string this[int param]
    {
        get{return array[param];}
        set{array[param] = value;}
    }
}

public class Car: Vehicle
{
    //派生类和基类通信，以 base 实现，基类首先被调用
    //指定创建派生类实例时应调用的基类构造函数
    public Car()
        : base("Car", 200)
    {}

    public Car(string name, int speed)
        : this()
    {}

    public override void ShowResult()
    {
        //调用基类上已被其他方法重写的方法
        base.ShowResult();
        Console.WriteLine("It's a car's result.");
    }
}

public class Audi : Car
{
    public Audi()
        : base("Audi", 300)
    {}

    public Audi(string name, int speed)
        : this()
    {}

    public override void ShowResult()
    {
        //直接基类如果有方法实现，则 base 只能继承其直接基类成员；
        //否则 base 会向上层基类访问
        base.ShowResult();
        base.Run();
        Console.WriteLine("It's audi's result.");
    }
}

public class Test_baseAndthis
{
    public static void Main()

```

```
    {
        Audi audi = new Audi();
        audi[1] = "A6";
        audi[2] = "A8";
        Console.WriteLine(audi[1]);
        audi.Run();
        audi.ShowResult();
    }
}
```

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；在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
    IL_0002:  call     instance void
    //base 调用父类成员
    InsideDotNet.Keyword.BaseAndThis.Car::ShowResult()
    IL_0007:  nop
    IL_0008:  ldarg.0
    //base 调用父类成员，因为没有实现 Car.Run()，所以调用更高级父类方法 Vehicle.Run()
    IL_0009:  call     instance void
    InsideDotNet.Keyword.BaseAndThis.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. 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 指向哪一层呢？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。

- (1) 执行 System.Object.Object()。
- (2) 执行 Vehicle.Vehicle(string name, int speed)。
- (3) 执行 Car.Car()。
- (4) 执行 Car.Car(string name, int speed)。
- (5) 执行 Audi.Audi()。

我们在充分了解其实例化顺序的基础上就可以顺利地把握 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：

```
.method public hidebysig static void Main() cil managed
{
    .entrypoint
    // 代码大小      61 (0x3d)
    .maxstack 3
    .locals init ([0] class InsideDotNet.Keyword.BaseAndThis.Audi audi)
    IL_0000: nop
        // 使用 newobj 指令创建新的对象，并调用构造函数初始化
    IL_0001: newobj     instance void InsideDotNet.Keyword.BaseAndThis.Audi::`ctor()
    IL_0006: stloc.0
    IL_0007: ldloc.0
    IL_0008: ldc.i4.1
    IL_0009: ldstr      "A6"
    IL_000e: callvirt   instance void InsideDotNet.Keyword.BaseAndThis.Vehicle:: set_Item
(int32,string)
    IL_0013: nop
    IL_0014: ldloc.0
    IL_0015: ldc.i4.2
    IL_0016: ldstr      "A8"
    IL_001b: callvirt   instance void InsideDotNet.Keyword.BaseAndThis.Vehicle:: set_Item
(int32,string)
    IL_0020: nop
    IL_0021: ldloc.0
    IL_0022: ldc.i4.1
    IL_0023: callvirt   instance string InsideDotNet.Keyword.BaseAndThis.Vehicle:: get_Item
(int32)
    IL_0028: call       void [mscorlib]System.Console::WriteLine(string)
    IL_002d: nop
    IL_002e: ldloc.0
    IL_002f: callvirt   instance void InsideDotNet.Keyword.BaseAndThis.Vehicle:: Run()
```

```
IL_0034: nop
IL_0035: ldloc.0
//base.ShowResult 最终调用到最高级父类 Vehicle 的方法,
IL_0036: callvirt instance void InsideDotNet.Keyword.BaseAndThis.Vehicle::ShowResult()
IL_003b: nop
IL_003c: ret
} // end of method Test_baseAndthis::Main
```

7.2.4 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的公有或受保护成员。
- 除了 base，访问基类成员的另外一种方式是：显式的类型转换来实现。只是该方法不能为静态方法。

7.2.5 结论

base 和 this 关键字，不是特别难于理解的内容，本节之所以将其作为讨论的主题，除了对其应用规则做一个小结之外，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 技术框架中对本质的诉求。对学习者来说，只有从本质上把握概念，才能在变化非凡的应用中，一眼找到答案。

读到这里，相信我们已经对本节开篇抛出的几个问题解惑了。抛砖意在引玉，接下来，就要靠大家在实践中体味了。

7.3 using 的多重身份

本节将介绍以下内容：

- using 指令的多种用法
- using 语句在 Dispose 模式中的应用

7.3.1 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 7.1 节“把 new 说透”中对

此都有详细的论述。本节，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，本节将从这几个角度来阐述 `using` 的多彩应用。

7.3.2 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，文件操作类型位于 `System.IO` 命名空间，XML 操作类型位于 `System.Xml` 命名空间。

误区：

- `using` 在此类似于 Java 语言的 `import` 指令，作用都是引入命名空间（Java 中称作包）这种逻辑结构；而不同于 C 语言中的`#include` 指令，其用于引入实际的物理类库。
- `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。
- `using` 引入命名空间，一般放在 C# 源文件的开头，这一要求不是必须的，但是 `using` 子句的位置必须位于所有其他命名空间的元素之前。

7.3.3 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 InsideDotNet.Keyword.Using
{
    using BoyPlayer = Boyspace.Player;
    using GirlPlayer = Girlspace.Player;

    class UsingEx
    {
        public static void Main()
        {
            BoyPlayer.Play();
            GirlPlayer.Play();
        }
    }
}

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，有效地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符合编码要求。

7.3.4 强制资源清理

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。这些显然不是本节所能完成的宏论，我们只需要首先明确.NET 提供了 Dispose 模式来实现显式释放非托管资源的能力。

关于：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。

本节不再讨论如何让一个类型实现 Dispose 模式来提供显式清理非托管资源的方式，而将集中在如何以 using 语句来简便地应用这种实现了 Dispose 模式的类型的资源清理方式，请参阅第 6 章的相关讨论。

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 IDisposable 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 IDisposable 接口，也就具有了显式回收资源的能力。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，这正是 using 语句带来的简便之处，其实质究竟怎样呢？

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 代码中来揭秘才行：

```
.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，微软建议开发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 语句正是隐式地调用了类型的 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();
}
```

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 语句声明的局部变量，必须是实现了 IDisposable 接口或者可以转换为 IDisposable 接口的类型。
- 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-finally 来实现，因为嵌套的 using 语句可能存在隐藏的 Bug，例如：

```
public class FileHandler : IDisposable
{
```

```

public void Dispose()
{
    //执行清理
}
}

public class DBHandler : IDisposable
{
    public DBHandler()
    {
        throw new Exception();
    }

    public void Dispose()
    {
        //执行清理
    }
}

class MoreUsing
{
    public static void Main()
    {
        FileHandler fh = new FileHandler();
        DBHandler dh = new DBHandler();

        using (fh)
        {
            using (dh)
            {
                //执行操作
            }
        }
    }
}

```

上述示例中 `FileHandler` 实例引用的非托管资源将不能得到及时的释放，解决的最好办法是在此基于 `try-finally` 来实现对多个非托管资源的清理。

- `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.Black))
{
    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，对象的内存永远由垃圾回收器控制。

- 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
}
```

7.3.5 结论

一个简单的关键字，多种不同的应用场合。本节从比较全面的角度，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，本节的情况主要局限在 C# 中。

7.4 认识全面的 null

本节将介绍以下内容：

- 详述 null
- 详述 Nullable<T> 类型
- 简述 ?? 操作符
- 探讨 Null Object 模式

7.4.1 引言

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。

7.4.2 从什么是 null 开始

null，一个值得关注的数据标识。

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，以至于在 JavaScript 语言中，null 类型作为 5 种基本的原始类型之一，与 Undefined、Boolean、Number 和 String 并驾齐驱。同样，这种重要性也表现在.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。

在上述让人拗口抓狂的表述中，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，如图7-1所示。

1. 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。在默认情况下，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，但是有所区别：

```
//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());
}
```

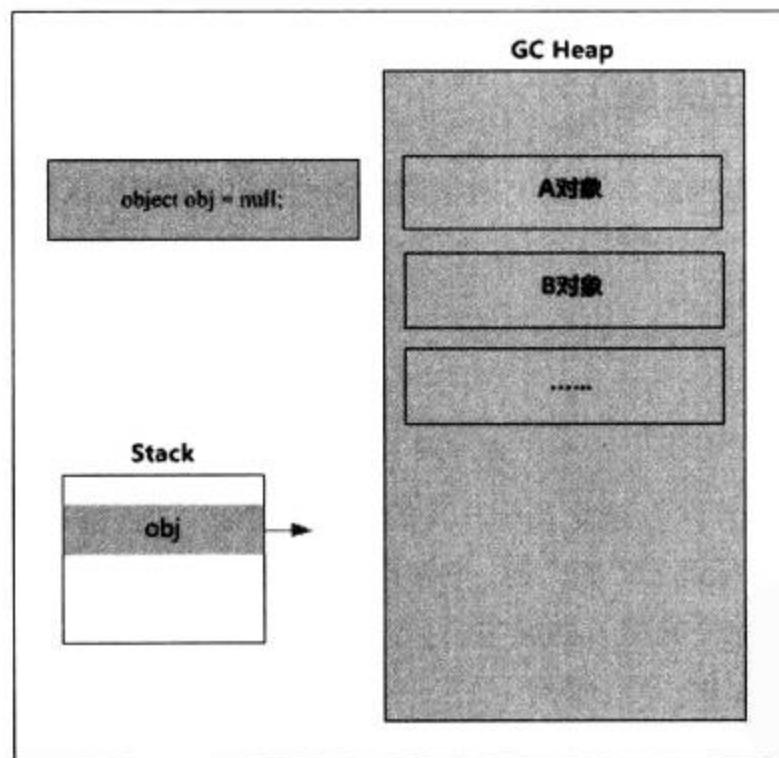


图7-1 null的内存情况

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。

2. 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`，而表示一个空引用。
- 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`检查。
- 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`null`的变量操作，都会抛出`NullReferenceException`异常。

7.4.3 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?`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>` 则表示值既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 中也存在相同的概念。

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情况呢？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` 可以看做是预定义的 `struct` 类型，创建一个 `Nullable<T>` 类型的 IL 表示可以非常清晰地提供例证，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<int32>:::.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;
```

```
int? c = (int?)200;
```

- 同时为了更好地将可空类型与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。

7.4.4 ??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频繁，所以.NET 2.0 中提供了新的操作运算符来简化 null 值的判定过程，这就是??运算符。上述过程能够以更加震撼的代码表现为：

```
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，又可以应用于引用类型。应用??运算符实现了简洁的 null 判定，例如通过??运算符，可以实现一个有意思的代码技巧：

```
string a = null;
object b = null;
```

```
object c = 1;
Console.WriteLine(a ?? null ?? b ?? c ?? null);
```

通过多次的??判定，可以很容易从一堆候选者 a、b、c 中挑出不是 null 的那个。

7.4.5 Null 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 种模式之外还是有很多设计精髓，可以称为模式之外有经典。但是仍然值得我们挖掘、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。

1. 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，就可以有效地避免烦琐的 if 和不必要的失误：

```
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，但是本质上完全是两回事儿。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 NullReferenceException 异常的发生”，这是来自 Scott Dorman 的声音。

2. 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。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 7-2。

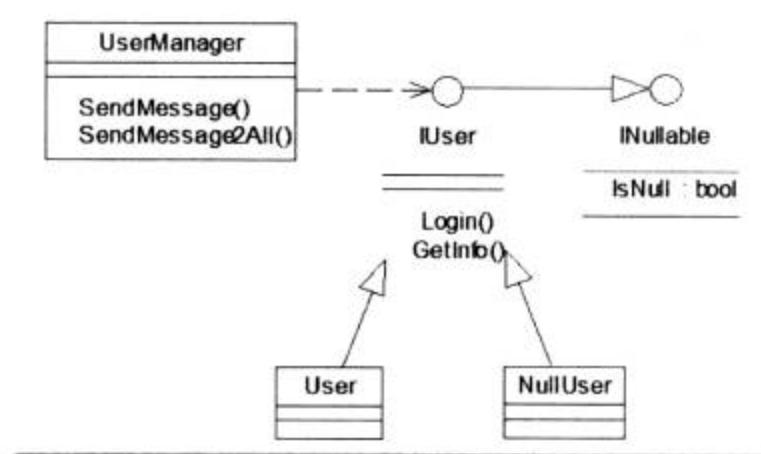


图 7-2 User 的 Null Object 模式

图 7-2 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 被定义为：

```
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 的定义为：

```
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：

```

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。

实际上，可以通过引入 Factory Method 模式来进行 User 和 NullUser 的创建工作，这样就可以完全消除应用 if 进行判断的僵化现象。

当然，这只是 null object 的一种实现方案，除了对《Refactoring: Improving the Design of Existing Code》一书的示例进行改良，更多考虑完成具有.NET 特色的 null object 实现，你也可以请 NullUser 继承 User 并添加相应的 IsNull 判定属性来完成。

3. 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 来取巧实现更简洁的 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：

```
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，仅仅从对 `null value` 的判断转换为对 `null object` 的判断角度来看，扩展方法带来了更有效且更简洁的表现力。

4. 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：

```
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”，他说的很对。

7.4.6 结论

虽然行色匆匆，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，大概也不过如此而已吧，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、思考和分享的内容。

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，所收获的果实就会更多。

7.5 转换关键字

本节将介绍以下内容：

- 简述类型转换
- 详述 `explicit`、`implicit`

7.5.1 引言

类型转换是程序设计的基础话题，本书在 5.2 节“品味类型——值类型与引用类型”中对类型转换的话题已经进行了一些探讨，而本节的重点则以类型转换为基础，来探讨 explicit, implicit 转换运算符的应用。

7.5.2 自定义类型转换探讨

类型转换，就是通过改变变量的类型来改变其表示方式，.NET 高级语言允许在类和结构上声明转换，并且这种转换可以是显式的，也可以是隐式的。显式转换必须由用户强制指定才能执行；隐式转换则由系统自行调用，因此不建议在隐式转换中抛出任何异常。

我们以 C# 语言为例来说明转换运算符的实现规则，一般的定义格式为：

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

其中，转换修饰操作符主要有：

- explicit，用于声明必须强制转换的自定义类型转换操作符。
- implicit，用于声明隐式的自定义类型转换操作符。

下面，我们举例说明这两种转换的具体应用：

```
class MyAge
{
    private Int32 age = 0;

    public Int32 Age
    {
        get { return age; }
        set { age = value; }
    }

    public MyAge()
    {
    }

    private MyAge(Int32 age)
    {
        this.age = age;
    }

    // 整型到 MyAge 的隐式转换
    public static implicit operator MyAge(Int32 year)
    {
        return new MyAge(year > 1980 ? (year - 1980) : -1);
    }

    // MyAge 到整型的显式转换
    public static explicit operator Int32(MyAge age)
    {
        if (age == null)
        {
            throw new ArgumentNullException("参数为空。");
        }
    }
}
```

```

    if ((age.Age < 0) || (age.Age > 150))
    {
        throw new InvalidCastException("不可能的年龄值。");
    }

    return age.Age;
}

//MyAge 到 String 类型的隐式转换
public static implicit operator String(MyAge age)
{
    return "我的年龄是：" + age.Age.ToString();
}
}

```

在该示例中，我们实现了显式和隐式自定义类型转换操作符，接着对上述实现做以简单的测试，如下：

```

class Test_MyAge
{
    public static void Main()
    {
        MyAge age = new MyAge();

        //执行显式类型转换：MyAge 转换为 Int32
        Int32 bornAge = (Int32)age;
        //执行隐式类型转换：Int32 转换为 MyAge
        age = DateTime.Now.Year;
        //执行隐式转换：MyAge 转换为 string
        Console.WriteLine(age);
    }
}

```

根据上述应用，使用转换运算符进行类型转换的一般规则，主要包括：

- 所有的转换都必须是 static 的。
- 类型转换可能存在信息丢失或者精度损失，对于有损转换，最好提供显式的类型转换，并且这种有损转换不被允许时，应在显式转换中抛出 `InvalidCastException` 异常。
- 用户定义运算符不能是封闭类型的对象，也不能转换成封闭类型的对象，否则将引发编译时异常，例如：

```

public static implicit operator MyAge(MyAge age)
{
    return age;
}

```

- 显式转换和隐式转换，是个值得权衡的选择：隐式转换较为灵活，可读性强，但是必须保证不会引发异常或者发生有损转换，否则会引起不必要的转换结果，这种情况下则实现为显式转换较为妥当。
- 运算符只能按值传递，不能按引用传递，因此这里也不能采用 `ref` 或 `out` 参数。

其他的数据类型转换，并非本节研究的内容，详细情况可以参见 5.2 节“品味类型——值类型与引用类型”的论述。

7.5.3 本质分析

了解了自定义类型转换的规则和应用，我们有必要对上述执行过程有个更深入的分析，以便更好的理解

类型转换的本质，在此我们将上述测试代码的 IL 分析一下，便可略知一二：

```
.method public hidebysig static void Main() cil managed
{
    .entrypoint
    // 代码大小      46 (0x2e)
    .maxstack 1
    .locals init ([0] class InsideDotNet.Keyword.Convert.MyAge age,
                 [1] int32 bornAge,
                 [2] valuetype [mscorlib]System.DateTime CS$0$0000)
    IL_0000:  nop
    IL_0001:  newobj     instance void InsideDotNet.Keyword.Convert.MyAge::..ctor()
    IL_0006:  stloc.0
    IL_0007:  ldloc.0
    IL_0008:  call         int32 InsideDotNet.Keyword.Convert.MyAge::op_Explicit (class
InsideDotNet.Keyword.Convert.MyAge)
    IL_000d:  stloc.1
    IL_000e:  call         valuetype [mscorlib]System.DateTime [mscorlib]System. DateTime:::get_Now()
    IL_0013:  stloc.2
    IL_0014:  ldloca.s   CS$0$0000
    IL_0016:  call         instance int32 [mscorlib]System.DateTime:::get_Year()
    IL_001b:  call         class InsideDotNet.Keyword.Convert.MyAge InsideDotNet.Keyword.
Convert.MyAge::op_Implicit(int32)
    IL_0020:  stloc.0
    IL_0021:  ldloc.0
    IL_0022:  call         string InsideDotNet.Keyword.Convert.MyAge::op_Implicit (class
InsideDotNet.Keyword.Convert.MyAge)
    IL_0027:  call         void [mscorlib]System.Console:::WriteLine(string)
    IL_002c:  nop
    IL_002d:  ret
} // end of method Test_MyAge::Main
```

可见在生成 IL 时，显式转换执行了 MyAge::op_Explicit 指令，而隐式转换则执行了 MyAge::op_Implicit 指令，实际上在我们的示例中发生了转换操作符重载操作，我们可以查看 MyAge 类的 3 个转换方法的元数据，可简单表示为：

```
public static int32 op_Explicit(MyAge age)
public static MyAge op_Implicit(int32 year)
public static string op_Implicit(MyAge age)
```

事实上，操作符是高级语言的预定义符号，在编译为 IL 语言时，会翻译成 CLR 识别的操作指令。CLR 规范中，将转换操作符重载定义为 public static 方法，并且必须指明编译器是否能生成隐式的调用代码，还是显式的调用代码，而这一操作在 C# 中正是由 implicit 和 explicit 关键字来完成的。

另外，对于操作符转换，如果将上述 MyAge 到 String 类型的隐式转换修改为显式转换，例如：

```
//MyAge 到 String 类型的显式转换
public static explicit operator String(MyAge age)
{
    return "我的年龄是：" + age.Age.ToString();
}
```

重新反编译执行结果，我们查看修改后的转换操作符方法的元数据，可简单表示为：

```
public static int32 op_Explicit(MyAge age)
public static string op_Explicit(MyAge age)
public static MyAge op_Implicit(int32 year)
```

修改后的两个 `op_Explicit` 方法具有相同的参数列表和方法签名，只有返回值不同，这显然有违一般意义上的重载规则。不过，这正是仅有返回值类型不同的重载方法范例，而 CLR 完全支持这种仅有返回值不同的重载。但这种重载方式并不被 C# 或者 VB.NET 这样的高级语言所支持，但是它却是实现转换操作符重载的重要机制。所以，在 C# 中想要变相地实现按返回值重载的情况时，`implicit operator` 是个可以参考的实现办法，但是这种机制不值得推荐。

7.5.4 结论

类型转换，是程序开发中比较常见的操作之一。深入的理解其本质和应用规则，对于减少不必要的错误操作，实现合理有效的类型转换大有好处。在.NET 框架类库中，很多类型都实现了 `op_Explicit` 和 `op_Implicit` 方法重载来显式或者隐式地转换为其他类型，研究这些类型的规则会带给我们更多的启示。

7.6 预处理指令关键字

本节将介绍以下内容：

- 简述预处理指令
- 逐一解析预处理指令

7.6.1 引言

作为关键字的一个系列，预处理指令不可或缺。而本节之所以将预处理指令关键字作为研究的主题，一个主要的原因正是.NET 高级语言中，预处理指令已经大非从前，很多原 C++/C 预处理指令在 C# 中已经不复存在。某些方面，.NET 提供了定制特性等新的手段代替了预处理指令的功能，并且 C# 编译器也不提供独立的预处理器，因此有必要对这些仅存的“硕果”做梳理，在差别和新规则上进行澄清。

值得注意的是，这部分内容主要针对的是 C# 高级语言的预处理指令说明。

7.6.2 预处理指令简述

预处理指令，主要用于辅助条件编译。预处理命令不会在编译时转化为可执行代码，但是会影响编译过程，例如根据条件跳过某段代码的编译、报告错误和警告条件。在 C# 中，预处理指令，主要如表 7-1 所示。

表 7-1 预处理指令

预处理指令	指令说明
#if、#else、#elif、#endif	用于包括或跳过源代码中由一个或多个符号组成条件的部分代码
#define、#undef	分别用于定义和取消定义条件编译符号
#warning、#error	分别用于发出警告和错误信息

续表

预处理指令	指令说明
#region、#endregion	将一段代码标记为给定名称的块，在Visual Studio中可以通过该指令使用大纲显示功能，二者必须配对使用
#line	用于控制行号，主要用于发布警告或错误信息时使用
#pragma	用于控制如何编译包含指定文件的编译
#pragma warning	用于抑制或者恢复指定的编译警告
#pragma checksum	用于生成源文件的校验和，主要用在ASP.NET调试

关于预处理指令，有如下的规则：

- 预处理指令以#符号开头，且必须独占源代码的单独一行，不用分号结束。
- 预处理指令代码不会编译为可执行文件，只用于控制编译过程。
- C#没有单独的预处理器，预处理指令由编译器来处理。
- C#预处理指令不能创建宏。
- .NET提供了定制特性，可以代替某些指令提供更加灵活的应用，例如使用ConditionalAttribute进行条件编译。

下面，我们就重点分析几个较为常用的预处理指令关键字的应用。

7.6.3 #if、#else、#elif、#endif

这组指令主要用于在调试环境下代码进行条件编译时，用于控制编译器对某个代码段是否进行编译。在使用上，类似于条件判断语句。其一般格式为：

```
#define debug
//...
public static void Main()
{
    #if debug
        Console.WriteLine("It's debug.");
    #elif release
        Console.WriteLine("It's release.");
    #endif
}
```

编译时是这样执行的，编译器遇到#if指令则会首先检查debug符号是否已经定义，如果存在定义则会执行子块中的代码，否则将忽略这段代码而继续执行，#elif的判断过程和#if是一样，直到#endif为止。在条件编译时，调试代码会放在#if子块中，来执行调试过程，结束调试后只需要取消#define定义的debug就可以自动忽略调试代码，这正是条件编译的好处。

#if和#elif支持逻辑运算符，例如可以：

```
#if debug && release
    Console.WriteLine("It's debug.");
#endif
```

同时，还支持嵌套执行，例如可以：

```
#if debug
    Console.WriteLine("It's debug.");
```

```
#if inline
    Console.WriteLine("It's inline.");
#endif
#ifndef release
    Console.WriteLine("It's release.");
#endif
```

在.NET中, ConditionalAttribute特性提供了更加灵活的条件编译方式, 代码的可读性强。在《Effective C#》一书就推荐以 ConditionalAttribute特性来代替#if/#endif程序块执行, 例如:

```
public static void Main()
{
    ShowDebug();
}

[Conditional("debug"), Conditional("release")]
private static void ShowDebug()
{
    Console.WriteLine("It's debug or release.");
}
```

ConditionalAttribute特性可以突破预编译指令的限制, 可以同时为方法指定多个处理标识符, 同时支持 System.Diagnostics.Trace 和 System.Diagnostics.Debug 定义的条件方法。因此, 值得推荐。

7.6.4 #define、#undef

这两个指令分别用于定义和取消定义一个给定名称的符号, 该符号可以应用于#if 指令作为判断条件, 和其他指令搭配才有意义, 例如上文对#if 的分析示例。其定义为:

```
#define debug
#undef debug
```

#define 指令和#undef 指令的定义必须放在 C#源代码的开头, 并且应用范围是整个定义符号所在的文件内。其定义的符号用于指定编译的条件, 除了#if/#elif 的判断条件, 还包括 ConditionalAttribute 特性的执行。

7.6.5 #warning、#error

#warning 指令用于发出警告信息, #error 指令用于发出错误信息。不同的是编译器遇到#warning 指令显示警告指令后, 会继续执行; 而#error 指令显示错误信息后, 会退出编译。这两个指令主要用于进行自定义的编译检测, #warning 提示什么事情值得注意, 而#error 则告诉你什么是错的。例如:

```
#define TRACE_WARNING
#define TRACE_ERROR
//....
public static void Main()
{
    #if TRACE_WARNING
        #warning "别忘了还有事情没有处理。"
    #endif

    #if TRACE_ERROR
        #error "这里是错的, 注意啦。"
    #endif
}
```

执行上述代码，则会在编译器的错误列表中看到一条警告信息和一条错误信息，这些信息对于较大系统的调试，起到了很好的辅助作用。

7.6.6 #line

#line 指令并不常用，主要应用于发布警告或者错误信息时来控制行号使用，用于修改编译器的行号或者修改警告和错误信息的文件名输出，其定义格式为：

```
#line lineNumber|hidden|default
```

其中，lineNumber 表示文件行号，hidden 表示隐藏连续行，default 表示重置文件行号。lineNumber 后可以指定文件名，表示输出到编译器中的文件名，默认情况下为源代码的实际名称。例如：

```
public static void Main()
{
    #line 100 "warning.cs"
    #warning "警告。"

    #line default
    #error "这里是错的，注意啦。"
}
```

编译上述代码，则在错误列表中会看到#line 指令的确修改了输出警告信息的行号、文件名等信息，而#line default 又恢复了行号。

其他的编译指令，例如#region、#endregion、#pragma 在此就不做进一步的讨论，留待读者自行研究。

7.6.7 结论

预编译指令是个不常讨论的话题，总是被我们遗忘在某个角落，但是作为辅助调试的有效手段，基本的了解是必要的。本节对几个常用的编译指令做了简要的分析和介绍，同时对.NET 提供的某些替代特性做了相应的讲述，为更好地实现条件编译等操作，带来便利。

7.7 非主流关键字

本节将介绍以下内容：

- 分析.NET 中的非典型关键字

7.7.1 引言

作为程序设计中的基本内容，对常见关键字的了解是个基本素质，每个高级语言都预设了很多的保留字来为编译器提供有意义的符号。在.NET 的世界也不例外，不同的编译器提供不同的关键字集，当然很多关键字都是形异神同，对于常见的关键字我们基本可以“心领神会”。

弄清每个关键字的详细功能是项难以完成的浩大工程，除了必须了解的基本关键字外，某些特别的关键

字也有充分的理由引起我们的关注。本节的目的正是为大家引荐几个重要但是不常见的关键字，并为其归结一个当下流行的名词：非主流关键字。

本节要研究的几个关键字，主要是：

- checked/unchecked
- yield
- lock
- unsafe
- sealed

下面，我们就逐一梳理，请上面的主角依次登场，让我们看清聚光灯下每个关键字的本来面目和其应用规则。

7.7.2 checked/unchecked

CLR 提供了对于整数运算或者类型转化溢出问题进行控制，我们可以选择进行溢出检查的 IL 指令，也可以选择不进行溢出检查而直接执行的 IL 指令。例如，C#编译器可以使用/checked 命令行开关进行溢出控制，而默认的选项是/checked-，表示关闭溢出检查。例如，我们可以在编译 My.cs 文件时，显式选择打开检测开关：

```
csc My.cs /checked
```

而 Visual Studio 2005/2008 编译器中，可以通过选择项目“属性”，选择“生成”页中的“高级”按钮，可以通过“检查运算上溢/下溢 (K)”选项来设置/checked 命令行开关，如图 7-3 所示。



图 7-3 Visual Studio 2005 中设置检查运算上溢/下溢选项

显然，这种方式在一定程度上缺乏灵活性，我们只能对整个项目的代码选择是否进行溢出检查的控制，而且启用溢出检查会导致系统性能的下降。如果更喜欢由自己掌握这种控制的权力，checked/unchecked 关键字就是用于完成这一需求的。

checked 关键字用于启用溢出检查，unchecked 关键字用于取消溢出检查。当溢出检查启用时，一旦发生溢出，CLR 会抛出 OverflowException 异常。

同时，必须明确的是，这种检查只对整数运算类型转换有效。主要包括：加、减、乘、除、自加、自减以及一元减运算，还有整型间的类型转换。例如：

```

class CheckedEx
{
    public static void Main()
    {
        byte a = 100;
        byte b = 200;
        Console.WriteLine(CheckSum(a, b));

        Int32 i = 256;
        Console.WriteLine(UncheckCast(i));

        //checked语句
        checked
        {
            byte c = (byte)(i + a);
            Console.WriteLine(c);
        }
    }

    private static byte CheckSum(byte a, byte b)
    {
        byte c = 0;
        try
        {
            //进行溢出检查
            c = checked((byte)(a + b));
        }
        catch (OverflowException ex)
        {
            Console.WriteLine(ex.Message);
        }

        return c;
    }

    private static byte UncheckCast(Int32 i)
    {
        //不进行溢出检查
        return unchecked((byte)i);
    }
}

```

对于整数运算或者类型转换，应该根据实际情况进行必要的溢出检查控制，在上述示例中，UncheckCast方法没有进行溢出检查，程序将照常执行，但是执行结果并非预期。另外，从示例中可知，关键字 checked 和 unchecked 既可以是操作符，也可以是语句。

一个值得注意的问题是，checked 和 unchecked 只能直接对整数运算和类型转换有效，而对于调用方法不会产生作用，例如方法 UncheckCast 内部的溢出检查，不受 checked 语句的控制：

```

checked
{
    UncheckCast(1000);
}

```

7.7.3 yield

yield 关键字作用于迭代器块中，用于向枚举器对象提供值或者发出结束信号。其中 yield return 用于依次

返回每个元素，而 yield break 用于终止迭代。在此实现一个简单的 yield 实现，来体会其作用：

```
class UserInfo
{
    string[] users = { "小王", "张三", "李四" };

    public IEnumerator<string> GetEnumerator()
    {
        for (Int32 i = 0; i < users.Length; i++)
        {
            yield return users[i];
        }

        yield break;
        yield return "BREAK"; //代码不可访问
    }

    public static void Main()
    {
        UserInfo userlist = new UserInfo();
        foreach (string str in userlist)
        {
            Console.WriteLine(str);
        }
    }
}
```

那么上述实现的本质如何呢？我们会发现上述示例中，GetEnumerator 方法期望一个 IEnumerator 类型的返回，而 yield return 显然返回的是 string 类型的值。编译器生成一个状态机（state machine）来维护其迭代器状态，foreach 每次循环调用 MoveNext 方法时，都能保证是从前一次 yield return 语句停止的地方开始执行。我们可以利用 Reflector 工具，反编译获取 MoveNext 方法的实现：

```
private bool MoveNext()
{
    switch (this.<>1__state)
    {
        case 0:
            this.<>1__state = -1;
            this.<>2__current = "小王";
            this.<>1__state = 1;
            return true;

        case 1:
            this.<>1__state = -1;
            this.<>2__current = "张三";
            this.<>1__state = 2;
            return true;

        case 2:
            this.<>1__state = -1;
            this.<>2__current = "李四";
            this.<>1__state = 3;
            return true;

        case 3:
            this.<>1__state = -1;
            break;
    }
    return false;
}
```

如果将上述示例代码翻译为 IL 代码，我们会发现 `yield return` 并没有对应专门的 IL 指令，编译器遇到 `yield return` 这种类成员时，就会新产生一个实现了 `IEnumerator` 接口的嵌套类，该类实现了从类成员返回相同 `IEnumerable` 接口，并以该嵌套类的实例代替类成员中的代码，该嵌套类正是用来维持迭代状态的。在此，我们简要地模拟该嵌套类的部分实现细节，以便更好地理解迭代的实现过程，也更好地理解了 `yield return` 背后的故事：

```

class UserInfo : IEnumerable<string>
{
    string[] users = { "小王", "张三", "李四" };

    IEnumerator<string> IEnumerable<string>.GetEnumerator()
    {
        return new NestedEnumerator(this);
    }

    //内部一个实现了 IEnumerator 接口的嵌套类
    class NestedEnumerator : IEnumerator<string>
    {
        private UserInfo userInfo;
        private int index;

        public NestedEnumerator(UserInfo userInfo)
        {
            this.userInfo = userInfo;
            this.index = -1;
        }

        //实现 MoveNext 方法
        public bool MoveNext()
        {
            index++;
            if (index < userInfo.users.Length)
                return true;
            else
                return false;
        }

        //实现 Current 属性
        string IEnumerator<string>.Current
        {
            get { return (string)userInfo.users[index]; }
        }

        //实现 Reset 方法
        public void Reset()
        {
            this.index = -1;
            this.userInfo = null;
        }

        public void Dispose() { }
    }
}

```

上述实现过程，并不完整，但是可以模拟基本的实现过程，由此也就很容易解释 yield 应用中的一些规则，主要包括：

- yield 只能应用于迭代器块中，无论 yield return 返回何种类型，方法必须返回以下的接口类型：
System.Collections.Generic.IEnumerable<T>、 System.Collections.IEnumerable、 System.Collections.Generic.IEnumerator <T>或 System.Collections.IEnumerator。
- yield 语句不允许应用于不安全块中。
- yield return 不支持方法带有 ref 或者 out 参数，因为那会使得状态机难以维护。
- yield 语句不能应用于匿名方法中。
- yield 语句不能出现在 catch 块，或者包含 catch 子句的 try 块中，也不能出现在 finally 块中。
- 迭代程序在一些复杂的应用中，更能显示其强大的威力，例如二叉树结构的数据，或者某些图形结构数据的遍历应用，在此就不做赘述了。

7.7.4 lock

多线程，意味着不同的线程可以异步执行，而带来的问题是必须协调资源的访问，多个线程同时访问同一资源对象必然产生不可预期的结果。因此，保证线程同步是安全执行多线程异步执行的基础，在.NET 中线程同步可以有多种方式：

- lock 语句
- 监视器
- 同步事件和等待句柄
- Mutex 对象

而本节的重点是对 lock 关键字的理解，因此我们只关注 lock 语句在线程同步中的本质和应用。

lock 语句用于给对象获取互斥锁，执行操作语句，然后再释放该锁。在线程同步时，lock 关键字将语句块标记为临界区，能保证代码顺利执行而不被其他线程中断，变量被包装在独占锁中，其他线程的只能等待执行解锁之后才可访问该对象。其一般格式为：

```
public static void Main()
{
    object obj = new object();
    lock (obj)
    {
        //执行线程同步相关操作
    }
}
```

将上述简单的 lock 语句，转换为 IL 代码如下：

```
.method public hidebysig static void Main() cil managed
{
    .entrypoint
    // 代码大小      30 (0x1e)
    .maxstack 2
    .locals init ([0] object obj,
                 [1] object CS$2$0000)
    IL_0000:  nop
```

```

IL_0001: newobj   instance void [mscorlib]System.Object::ctor()
IL_0006: stloc.0
IL_0007: ldloc.0
IL_0008: dup
IL_0009: stloc.1
IL_000a: call      void [mscorlib]System.Threading.Monitor::Enter(object)
IL_000f: nop
.try
{
    IL_0010: nop
    IL_0011: nop
    IL_0012: leave.s  IL_001c
} // end .try
finally
{
    IL_0014: ldloc.1
    IL_0015: call      void [mscorlib]System.Threading.Monitor::Exit(object)
    IL_001a: nop
    IL_001b: endfinally
} // end handler
IL_001c: nop
IL_001d: ret
} // end of method lockEx::Main

```

从 IL 代码中可以看出,lock 语句本质上被扩展为一个使用 System.Threading.Monitor 类的 try-finally 语句,因此以上代码完全等效于:

```

//实现一个等效操作
public static void Main()
{
    object obj = new object();
    System.Threading.Monitor.Enter(obj);
    try
    {
        //执行线程同步相关操作
    }
    finally
    {
        System.Threading.Monitor.Exit(obj);
    }
}

```

上述代码可以生成几乎相同的 IL 代码,可见 lock 的本质正是对 Monitor.Enter 和 Monitor.Exit 的封装。而 lock 看起来更加简洁,并且 finally 块总能保证释放基础监视器(Monitor)资源。

下面的小示例中,应用 lock 保证了多线程环境中对 ArrayList 的同步操作,例如:

```

class LockApp
{
    private static ArrayList al = new ArrayList();

    public static void AddItems()
    {
        lock (al)
    }
}

```

```

    {
        al.Add(DateTime.Now.ToString());
        Thread.Sleep(1000);
    }
}

public static void Main()
{
    Thread t1 = new Thread(new ThreadStart(AddItems));
    Thread t2 = new Thread(new ThreadStart(AddItems));
    t1.Start();
    t2.Start();

    Console.Read();

    foreach (string str in al)
    {
        Console.WriteLine(str);
    }
}
}

```

然后，我们对 lock 的规则做一个小结：

- lock 的对象必须是引用类型参数。
- 避免锁定公共对象或不受应用程序控制的对象实例，最好是定义 private 对象来锁定。
- String 类型对象对多线程操作是安全的，因此不建议锁定字符串类型对象。事实上，由于字符串的驻留机制，在整个应用程序中一个给定的字符串可能只有一个实例，字符串锁定会导致不必要的阻塞或死锁，关于 string 类型的相关信息可以参见 9.5 节“如此特殊：大话 String”。
- 避免死锁。让两个线程以相同的加锁顺序锁定对象，是避免死锁的有效手段。
- Monitor 类还提供了一个 TryEnter 方法，使用上更加灵活，可以有效地解决死锁的发生。二者的区别是，lock 会一直等待锁定对象释放后下一线程才能进入；而 TryEnter 会返回当前线程是否获取该锁，是则返回 true，否则返回 false。
- 线程同步最好只应用在需要的时候，因为锁定对象对系统性能存在影响：一方面是加锁与解锁的系统开销；另一方面可能导致其他线程因为等待释放对象而暂停执行。

7.7.5 unsafe

在 C++ 中，最普遍的指针操作在 C# 中受到限制。因为在托管平台下，一切跳出 CLR 控制进行直接取址或者垃圾回收的操作，被称为“不安全代码”。在托管平台下，内存分配、类型转换和垃圾回收都由 CLR 自行完成。在 C++ 中的指针类型构造，C# 中实现了与之对应的引用类型。然而，某些情况下，我们仍然需要以指针的方式来解决问题，而 C# 提供了这种编写不安全代码的能力，那就是：unsafe。

另外，C# 编译器提供了 /unsafe 开关来控制是否允许对 unsafe 关键字的代码进行编译，在 Visual Studio 2005/2008 中，其设置过程为：选择项目属性，打开“生成”属性页，勾选“允许不安全代码 (W)”即可。

下面，我们举例说明，在C#中应用unsafe来实现不安全代码，像*、&和sizeof这些操作在这里依然有效：

```
public struct MyStruct
{
    public int a;
    public int b;
}

class unsafeEx
{
    public static unsafe void Main()
    {
        MyStruct ms = new MyStruct();
        //获取MyStruct的首地址
        Console.WriteLine((uint)&ms);
        //获取值类型的大小
        Console.WriteLine(sizeof(MyStruct));
        //在堆栈上分配内存空间
        int* p = stackalloc int[10];
        *p = 100;
        Console.WriteLine(*p);
    }
}
```

7.7.6 sealed

sealed，就是“密封”。在.NET中，sealed关键字表示基类不能被继承，或者方法和属性不能被覆写。密封类或者密封方法，用于避免在继承类中非有意的派生和覆写，保证对基类的封装。例如，.NET定义System.String类型为密封类，以避免用户在自定义继承类中添加字段，从而可能破坏对String对象的恒定性假设。

在.NET框架类库中就定义了很多密封类，例如 System.String、System.Drawing.Pen、System.Nullable、System.Math等。下面，我们介绍一个简单的示例：

```
public class MyBase
{
    public virtual void ShowMsg()
    {
    }
}

public sealed class MySealed : MyBase
{
    public sealed override void ShowMsg()
    {
        base.ShowMsg();
    }
}
```

上例中，MySealed类不可再被继承，而方法ShowMsg只有当对基类的虚方法进行覆写时才可定义为密封方法，因此定义密封方法时sealed必须和override同时使用。密封方法可以覆写基类的方法，但其本身不可在子类中覆写。

下面总结其规则，主要包括：

- sealed 不能和 abstract 共用，因为密封类不能被继承，而抽象类又总是希望被继承，二者语义抵触，不可共存。
- 定义密封方法时，sealed 必须和 override 一起使用。
- 优化性能，一方面是密封类不用作基类，另一方面对于密封类成员的虚方法调用将转换为非虚方法的处理机制，因此在执行成本上略高于虚方法的执行机制。
- string 类型是密封的。
- struct 是隐式密封的，因此也不能被继承，例如 System.Decimal 等。
- 密封类一个常见的应用就是：当一个类只有静态成员时，可以考虑将其实现为密封类，例如.NET 框架中的 System.Math、System.Drawing.Pen 等。

7.7.7 结论

本节介绍的几个关键字，在某些特定应用中非常重要，例如 lock 应用于多线程环境下的线程同步，yield return 简化了迭代器实现的原理，unsafe 在不安全代码编写时的应用等。当然，还有更多的关键字，本书并未涉及，还需要读者自己的探索。

总之，对于关键字的理解，是个常被忽略又值得探讨的问题。对于支持.NET 的高级语言而言，有些关键字对应于 IL 的指令，而有些又并非对应于 IL 指令。非主流，仔细玩味也精彩。

参考文献

Stanley B.Lippman, C# Primer

David Chappell, Understanding .NET

Bill Wagner, Effective C#

Christian Nagel, Bill Evjen, Jay Glynn, Professional C# 2005

Scott Dorman, Null Object pattern

Patrick Smacchia, Iterators With C#2,

<http://www.theserverside.net/tt/articles/showarticle.tss?id=IteratorsWithC2>

Mahesh Chand, Using Sealed Class in .NET, <http://www.c-sharpcorner.com/UploadFile/mahesh/SealedClasses11142005063733AM/SealedClasses.aspx>

Bear-Study-Hard, C#学习笔记（二）：构造函数的执行序列

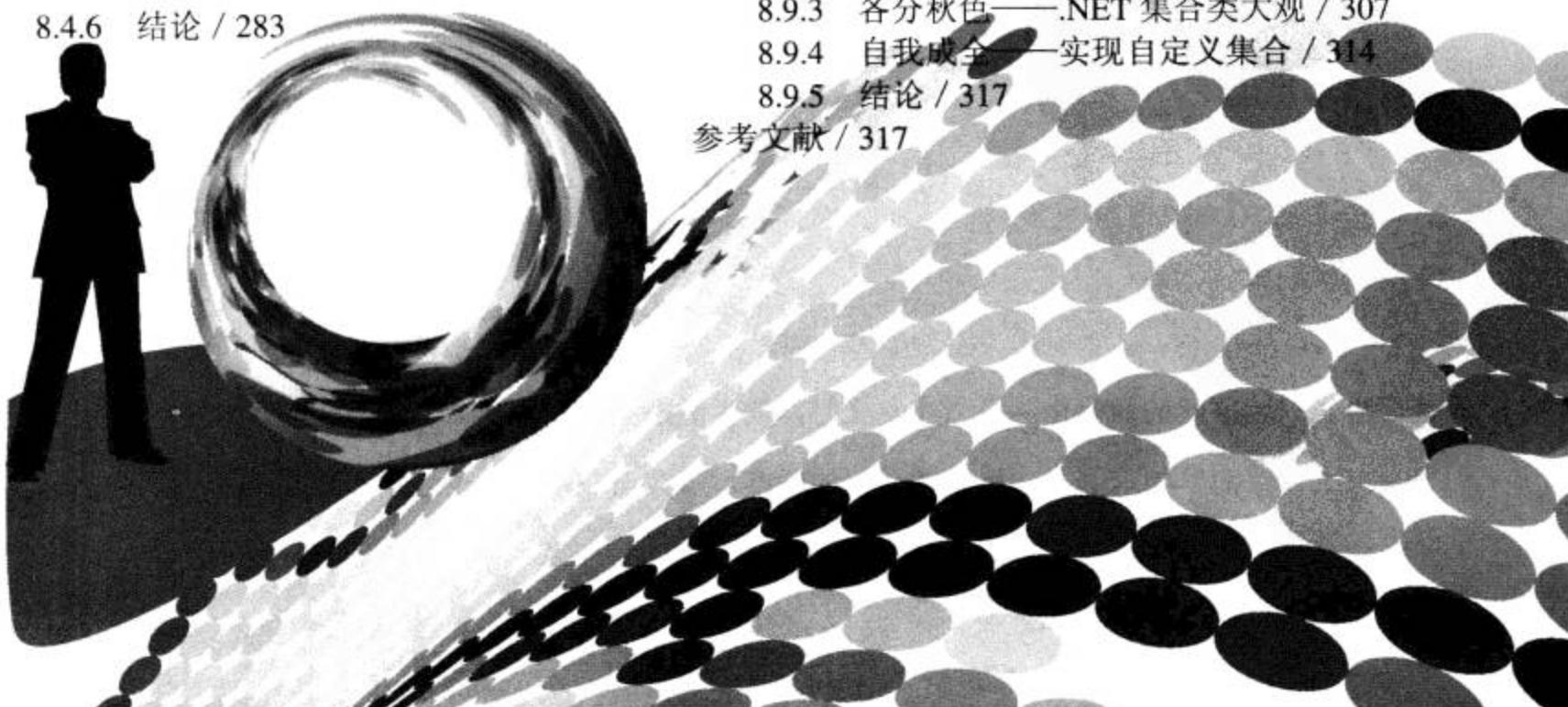
zhuweisky, 使用 Null Object 设计模式

Martin Fowler, Refactoring: Improving the Design of Existing Code

第8章 巅峰对决——走出误区

- 8.1 什么才是不变：const 和 readonly / 256
 8.1.1 引言 / 256
 8.1.2 从基础到本质 / 257
 8.1.3 比较，还是规则 / 259
 8.1.4 进一步的探讨 / 260
 8.1.5 结论 / 263
- 8.2 后来居上：class 和 struct / 263
 8.2.1 引言 / 263
 8.2.2 基本概念 / 263
 8.2.3 相同点和不同点 / 264
 8.2.4 经典示例 / 265
 8.2.5 结论 / 268
- 8.3 历史纠葛：特性和属性 / 268
 8.3.1 引言 / 268
 8.3.2 概念引入 / 268
 8.3.3 通用规则 / 270
 8.3.4 特性的应用 / 271
 8.3.5 示例 / 273
 8.3.6 结论 / 277
- 8.4 面向抽象编程：接口和抽象类 / 277
 8.4.1 引言 / 277
 8.4.2 概念引入 / 277
 8.4.3 相同点和不同点 / 279
 8.4.4 经典示例 / 281
 8.4.5 他山之石 / 283
 8.4.6 结论 / 283
- 8.5 恩怨情仇：is 和 as / 284
 8.5.1 引言 / 284
 8.5.2 概念引入 / 284
 8.5.3 原理与示例说明 / 284
 8.5.4 结论 / 285
- 8.6 貌合神离：覆写和重载 / 286
 8.6.1 引言 / 286
 8.6.2 认识覆写和重载 / 286
 8.6.3 在多态中的应用 / 288
 8.6.4 比较，还是规则 / 289
 8.6.5 进一步的探讨 / 290
 8.6.6 结论 / 292
- 8.7 有深有浅的克隆：浅拷贝和深拷贝 / 292
 8.7.1 引言 / 292
 8.7.2 从对象克隆说起 / 292
 8.7.3 浅拷贝和深拷贝的实现 / 294
 8.7.4 结论 / 296
- 8.8 动静之间：静态和非静态 / 296
 8.8.1 引言 / 296
 8.8.2 一言蔽之 / 297
 8.8.3 分而治之 / 297
 8.8.4 结论 / 302
- 8.9 集合通论 / 302
 8.9.1 引言 / 302
 8.9.2 中心思想——纵论集合 / 303
 8.9.3 各分秋色——.NET 集合类大观 / 307
 8.9.4 自我成全——实现自定义集合 / 314
 8.9.5 结论 / 317

参考文献 / 317



8.1 什么才是不变：const 和 readonly

本节将介绍以下内容：

- 常量类型解析
- const 与 readonly 关键字比较
- const 与 readonly 应用场合

8.1.1 引言

不变的量是程序设计中的平衡剂，是系统中恒定不变的量，在.NET 中提供了两种方式来实现：const 和 readonly，其中 const 是静态常量，而 readonly 是动态常量。严格意义上，const 应该称为常量；而 readonly 则应称为只读变量。在本节中，我们统一以“常量”这个概念来定位这两个不同的概念，但是在本质上希望读者有所区别。

关于 const 和 readonly，我们还是从最需要关注的误区入手来陈述，在引人入胜的示例中，思考答案。

下面是几个典型的应用误区，请问这些应用都错在哪里？

```
private const string str;
```

答案：const 常量在定义时必须指定初始值。

```
private const object o = new object();
```

答案：不能用 new 运算符初始化一个 const 常量，即便是该数据是一个值类型，因为 new 运算总是在运行时才能确定，对于引用类型除了 string 外，必须初始化为 null。

```
private static string astr = "abc";
private const string str = astr + "efg";
```

答案：str 常量的值，必须确定于编译时，解决的办法是给 astr 加上 const 定义，或者将 str 的 const 换成 readonly 即可。

```
public void InlineFunction()
{
    readonly string myName = "Wang";
}
```

答案：不能在局部变量中使用 readonly 定义，而 const 则可以定义字段常量和局部常量。

```
public class MyClass
{
    private static readonly string ASTRING = "First String";
    public MyClass(string str)
    {
        ASTRING = str;
    }
}
```

答案：静态只读字段的初始化，必须在定义时，或者静态无参构造函数中进行。

如果对以上几个简单的示例，你还存在一定的疑惑，那么请在本节的论述中寻找答案。

8.1.2 从基础到本质

首先，我们从一个示例的分析，开始对 `const` 和 `readonly` 从基础到本质的介绍，示例如下：

```
public class ConstAnd_READONLY
{
    //声明字段
    private const string NAME = "Wang Tao";
    private readonly Int32 AGE = 22;
    private readonly string SEX;
    private static readonly string PASSWORD = "000000";

    //构造函数
    public ConstAnd_READONLY()
    {
        AGE = 23;
    }

    public ConstAnd_READONLY(Int32 age, string sex)
    {
        AGE = age;
        SEX = sex;
    }

    //静态无参构造函数
    static ConstAnd_READONLY()
    {
        PASSWORD = "123456";
    }

    //局部常量定义
    public void InlineConst()
    {
        const string myDescription = "Good morning.";
        Console.WriteLine(myDescription);
    }

    public static void Main()
    {
        //访问静态成员
        Console.WriteLine(ConstAnd_READONLY.NAME);
        Console.WriteLine(ConstAnd_READONLY.PASSWORD);

        //访问非静态成员
        ConstAnd_READONLY cr = new ConstAnd_READONLY();
        Console.WriteLine(cr.AGE);
        Console.WriteLine(cr.SEX);
        ConstAnd_READONLY cr2 = new ConstAnd_READONLY(27, "Man");
        Console.WriteLine(cr2.AGE);
        Console.WriteLine(cr2.SEX);

        //局部常量
        cr.InlineConst();
    }
}
```

输出结果为：

Wang Tao
123456
23

27
Man
Good morning.

1. 基础分析

我们从上例的执行过程和执行结果，分析其基础内容，可以有如下的结论：

- const、readonly 和 static readonly 定义的常量，指定初始值后（包括在构造函数内指定初始值）将不可更改，可读不可写。
- const 必须在声明时指定初始值；而 readonly 和 static readonly 在声明时可以指定也可以不指定初始值，同时也可以在构造函数内指定初始值，如果同时在声明时和构造函数内指定了初始值，以构造函数内指定的值为准。例如 ConstAnd_READONLY.PASSWORD 返回的是“123456”而不是声明时的“000000”。关于构造函数的初始化顺序，详细参见 8.8 节“动静之间：静态和非静态”的讨论。
- const 和 static readonly 定义的常量是静态的，只能由类型直接访问；而 readonly 定义的常量是非静态的，只能由实例对象访问。
- static readonly 常量，如果在构造函数内指定初始值，则必须是静态无参构造函数，例如 PASSWORD 参数的初始化过程。
- const 可以定义局部常量和字段常量，例如在 InlineConst 方法中我们定义的 myDescription；而 readonly 和 static readonly 不能定义局部常量，只能定义字段常量。实际上，readonly 应该称之为只读字段，因此局限于定义字段；而 const 才是常量，可以定义字段和局部量。

2. 本质回归

完成基础的简单分析，我们反观其 IL 代码，可以更清楚地认识本质上的 const 和 readonly 是怎么回事，如图 8-1 所示。



图 8-1 const 和 readonly 的 IL 分析

从 IL 中可以看出：

- const 常量 NAME，被定义为 private static literal string，可见 const 常量会自动编译为 static 成员，因此

`const` 常量是静态常量，确定于编译时，属于类型级。

- `readonly` 常量 `AGE` 和 `SEX` 被分别定义为 `private initonly int32 AGE` 和 `private initonly string SEX`，为非静态常量，两个实例构造器`.ctor: void()`和`.ctor: void(int32,string)`用于在运行时在构造函数内初始化 `readonly` 常量，因此 `readonly` 常量确定于运行时，属于对象级。
- `static readonly` 常量 `PASSWORD`，被定义为 `private static initonly string`，并在类型构造器`.cctor: void()`中，进行初始化，详细为：

```
.method private hidebysig specialname rtspecialname static
    void .cctor() cil managed
{
    // 代码大小      23 (0x17)
    .maxstack 8
    IL_0000: ldstr      "000000"
    IL_0005: stsfld     string
    InsideDotNet.Conception.ConstAnd_READONLY.ConstAnd_READONLY:: PASSWORD
    IL_000a: nop
    IL_000b: ldstr      "123456"
    IL_0010: stsfld     string
    InsideDotNet.Conception.ConstAnd_READONLY.ConstAnd_READONLY:: PASSWORD
    IL_0015: nop
    IL_0016: ret
} // end of method ConstAnd_READONLY::cctor
```

可见，不同于 `const` 常量的值确定于编译时，`readonly` 和 `static readonly` 常量都是在运行时通过构造函数进行初始化。

- `const` 常量编译后保存于模块的元数据中，无须在托管堆中分配内存，并且 `const` 常量只能是编译器能够识别的基本类型，例如 `Int32`、`char`、`string` 等；而 `readonly` 常量需要分配独立的存储空间，并且可以是任何类型。

8.1.3 比较，还是规则

我们从基础到本质了解了 `const` 和 `readonly` 的来龙去脉，因此有必要将这些认知进行概括总结，为我们的应用提供条理性的指导规则：

- `const` 默认是静态的，只能由类型来访问，不能和 `static` 同时使用，否则出现编译错误；`readonly` 默认是非静态，由实例对象来访问，可以显式使用 `static` 定义为静态成员。
- `const` 只能应用在值类型和 `string` 类型上，其他引用类型常量只能定义为 `null`，否则以 `new` 为 `const` 引用类型常量赋值，编译器会引发“只能用 `null` 对引用类型（字符串除外）的常量进行初始化”错误提示，原因是构造函数初始化是在运行时，而非编译时；`readonly` 只读字段，可以是任意类型，但是对于引用类型字段来说，`readonly` 不能限制对该对象实例成员的读写控制。
- `const` 必须在字段声明时初始化；而 `readonly` 可以在声明时，或者构造函数中进行初始化，不同的构造函数可以为 `readonly` 常量实现不同的初始值。

```
public class My_READONLY
{
    public readonly string NAME;
```

```

public MyReadonly()
{
    NAME = "Wang";
}

public MyReadonly(int i)
{
    NAME = "Hao";
}
}

```

static readonly 字段只能在声明时，或者静态构造函数中进行初始化。

```

public class MyReadonly
{
    //在声明时初始化
    public static readonly int ID = 99;

    //在静态无参构造函数中初始化
    public static readonly string NAME;

    static MyReadonly()
    {
        NAME = "Wang Tao";
    }
}

```

- const 可以定义字段和局部变量；而 readonly 则只能定义字段。
- const 定义时必须初始化；而 readonly 定义时可以不进行初始化，但是微软强烈建议在定义时进行初始化操作，否则 CLR 将根据其类型赋予默认值。
- 数组和结构体不能被声明为 const 常量，string 类型可以被声明为常量，其源于 string 类型的字符串恒定特性，使得 string 的值具有只读特性。
- 从应用角度来看，对于恒定不变且单独使用的量来说，应该考虑声明为 const 常量，例如圆周率、性能比、折扣率、百分比等；而对于可能随实际运行发生变化的量，应该考虑声明为 readonly 常量，例如日期或时间，数据库中的主键 ID 等。

8.1.4 进一步的探讨

1. const 的可能问题

在引用程序集时，const 常量将直接被编译到引用程序集中；而 readonly 则在动态调用时才获取。因此，在跨程序集的应用系统中，要特别关注 const 常量可能引发的不一致问题，执行的具体步骤为：

首先，创建一个程序集 DLL_ConstAnd_READONLY.dll，并执行以下代码：

```

using System;

namespace DLL_ConstAnd_READONLY
{
    public class MyClass
    {
        public const string CONST_STRING = "First Const String.";
    }
}

```

```

    public static readonly string READONLY_STRING = "First Readonly String.";
}
}

```

然后，创建引用程序集 DLL_Reference.exe 来访问上述定义的两个常量，如下：

```

using System;

namespace DLL_Reference
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine(DLL_ConstAndReadonly.MyClass.CONST_STRING);
            Console.WriteLine(DLL_ConstAndReadonly.MyClass.READONLY_STRING);
        }
    }
}

```

执行 DLL_Reference.exe，输出结果为：

```

First Const String.
First Readonly String.

```

最后，修改 DLL_ConstAndReadonly 中的常量字符串，如下：

```

public const string CONST_STRING = "Second Const String.";
public static readonly string READONLY_STRING = "Second Readonly String.";

```

重新单独编译 DLL_ConstAndReadonly.dll 程序集，并将编译后的新程序集复制到 DLL_Reference.exe 的运行目录中，再次运行 DLL_Reference.exe 应用程序，输出结果为：

```

First Const String.
Second Readonly String.

```

可见，第二次执行的结果并未按照预期的目标执行，这就是 `const` 常量带来的一个常见隐患，我们必须重新编译 DLL_Reference.exe 才能克服这一问题，简单地分析 DLL_Reference.exe 的 IL 代码，就能很快确定其症结所在：

```

.method private hidebysig static void Main(string[] args) cil managed
{
    .entrypoint
    // 代码大小      30 (0x1e)
    .maxstack 8
    IL_0000:  nop
    IL_0001:  ldstr     "First Const String."
    IL_0006:  call       void [mscorlib]System.Console::WriteLine(string)
    IL_000b:  nop
    IL_000c:  ldsfld     string  [DLL_ConstAndReadonly]DLL_ConstAndReadonly. MyClass::
READONLY_STRING
    IL_0011:  call       void [mscorlib]System.Console::WriteLine(string)
    IL_0016:  nop
    IL_0017:  call       int32  [mscorlib]System.Console::Read()
    IL_001c:  pop
    IL_001d:  ret
} // end of method Program::Main

```

从 IL 中可知, const 值在编译时就已经内联到引用程序集, 并没有调用任何外部程序集的操作, 因此并未获取到新程序集中的更改值; 而 static readonly 则是在运行期调用外部程序集动态获取的值: DLL_ConstAnd_READONLY_STRING, 有效地克服了程序集更新引起的一致性隐患。

2. 只读属性还是只读字段

在.NET 中, 关于运行时的只读特性, 还有另外一种更常用的手段来实现, 这就是只读属性 (getter property), 与 readonly 只读字段比较起来, 只读属性提供了对只读变量更好的封装性和更灵活的控制机制, 同时也能够满足对类型字段只读控制的要求, 因此在使用上我们更推荐以只读属性来提供对类型特性的封装。

```
class Readonly_PropertyAndField
{
    public static void Main()
    {
        MyReadonlyEx myReadonly = new MyReadonlyEx("王", "老五");
        Console.WriteLine(myReadonly.Name);
        Console.WriteLine(myReadonly.NickName);
    }
}

//对比只读字段和只读属性
class MyReadonlyEx
{
    //定义只读属性
    private string _firstName;
    private string _secondName;

    public string Name
    {
        get { return _firstName + _secondName; }
    }

    //定义只读字段
    public readonly string NickName;

    public MyReadonlyEx(string firstName, string secondName)
    {
        _firstName = firstName;
        _secondName = secondName;
        NickName = "小王";
    }
}
```

分析上述的示例, 并参考本书前文 1.3 节“封装的秘密”, 结合对.NET 封装特性的理解, 我们不难看出对于类型状态只读特性的实现上, 只读属性具有更好的灵活性和扩展性, 例如 Name 属性除了具有只读这一基本要求外, 还是两个字段(firstName 和 secondName 的组合, 当 Name 属性的需求发生变化时, 我们只需要更改 getter 方法的实现即可, 而不需要做更多的修改; 另外, 只读特性更好地隐藏了内部字段的信息, 只向外暴露安全的属性即可, 有效实现了类的安全性, 而只读字段要想在外部使用, 必须定义为 public。

因此, 尽可能地推荐以只读属性实现对类型读写特性的控制, 而不是只读字段。但是在某些情况下, 使用只读字段可以更加简化, 例如, 微软经典的 Petshop 示例在实现 SQLHelper 代码时, 就是以 static readonly 来定义数据库连接字符串常量:

```
internal static readonly string ConnectionString = ConfigurationManager.ConnectionStrings["AccessDB"].ConnectionString;
```

ConnectionString 的值，并非确定于编译期，而是在运行期由读取配置文件的内容来获取，因此 readonly 也是一种比较灵活的定义只读字段的机制。

8.1.5 结论

const 是编译时常量，readonly 是运行时常量；const 较高效，readonly 更灵活。在应用上，推荐以 static readonly 来代替 const，以平衡 const 在灵活性上的不足，同时克服编译器优化 const 性能时，所带来的程序集引用不一致问题。

8.2 后来居上：class 和 struct

本节将介绍以下内容：

- 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

8.2.1 引言

提起 class 和 struct，我们首先的感觉是语法相似，所受的待遇却截然不同。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，本节将进行全面讨论。

8.2.2 基本概念

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、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. 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 这样的基本类型对待，如图 8-2 所示。

关于值类型和引用类型的分析，请参见本书第 5 章“品味类型”的相关讨论。

8.2.3 相同点和不同点

相同点：语法类似，其相似点还不足以进行必要的分析，重点应该把握其不同点。

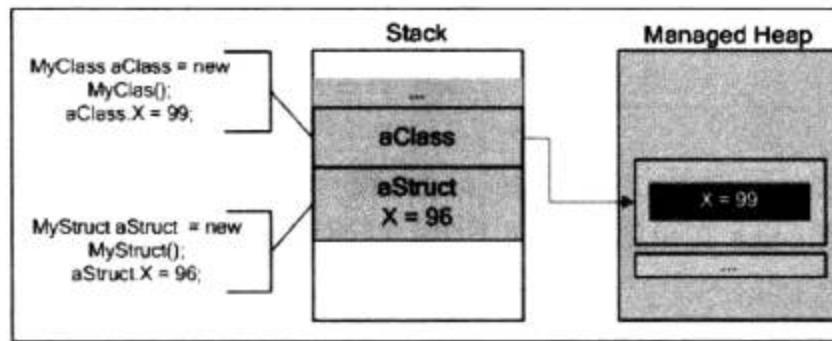


图 8-2 内存概况

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 本身又是个引用类型。
- 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，8.4 节“面向抽象编程：接口和抽象类”对此有详细讨论。
- class 可以声明无参构造函数，可以声明析构函数（注意：.NET 的析构函数不同于 C++ 中的析构函数概念，在本质上是对象的 Finalize 方法，相关讨论请参见 6.3 节“垃圾回收”的分析）；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。例如：

```

struct MyStruct
{
    public int i;
    public string name;

    public void ShowMsg()
    {
        Console.WriteLine("Hello");
    }
}

public class Test_Struct
{
    public static void Main()
    {
        MyStruct ms;
        ms.i = 10;
        ms.name = "小王";
        Console.WriteLine(ms.i);
        // 必须完成所有字段的初始化，才能调用 ShowMsg()
    }
}
  
```

```

        ms.ShowMsg();
    }
}

```

- class 可以为抽象类 (abstract)，可以声明抽象函数；而 struct 不能为抽象，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 GetHashCode()。
- class 的对象复制分为浅拷贝和深拷贝，必须经过特别的方法来完成复制，在 8.7 节“有深有浅的克隆：浅拷贝和深拷贝”有详细的讨论和分析；而 struct 创建的对象复制简单，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 变量和 struct 变量有所不同，详细参见 5.3 节“参数之惑——传递的艺术”。

我们可以简单地理解，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，所以只有深刻地理解内存分配的相关内容，才能更好地驾驭。当然正如本节标题描述的一样，使用 class 基本可以替代 struct 应用的任何场合，class 后来居上。虽然在某些方面 struct 有性能上的优势，但是在面向对象编程里，基本是 class 横行天下。

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，我们应该优先使用 struct 而不是 class：

- 实现一个主要用于存储数据的结构时，应该考虑 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。

以上是 struct 占有一席之地的理由，在某些场合以 struct 处理数据能带来更好的性能。

8.2.4 经典示例

1. 小菜一碟

下面以示例为说明，来阐述本节的基本规则，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
        Family newFamily;
        newFamily.name = "Aero 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("Aero Family", 26, true, "China", new Person("Aero", 26));

        Person person = new Person();
        person.Name = "Aero";

        //按值传递参数
        ShowFamily(myFamily);
        //按引用传递参数
        ShowPerson(person);

        Console.WriteLine("*****");
        Console.WriteLine("I'm {0}", myFamily.name);
        Console.WriteLine("I'm {0}", person.Name);
        myFamily.GetSex();
        myFamily.ShowPerson();
        Console.WriteLine("*****");
        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 = "Aero";
        Console.WriteLine("This is {0}", family.name);
    }
}

```

猜猜运行结果如何，可以顺便检查对这个概念的认识。

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，将有更多的体会。另外，还有以 struct 实现的 System.Guid。

8.2.5 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，本书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。本节关于 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这一概念层次。

8.3 历史纠葛：特性和属性

本节将介绍以下内容：

- 定制特性的基本概念和用法
- 属性与特性的区别
- 反射的简单介绍

8.3.1 引言

attribute 是.NET 框架引入的又一技术亮点，因此我们有必要花点时间来了解本节的内容，走进一个发现 attribute 的快捷入口。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]这些标记，相信我们都见过吧，那么你是否了解其背后的技术呢？

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，将在本节做一总结和比较，希望给你带来收获。另外本节以特性的介绍为主题，属性的论述重点突出在二者的比较上。

8.3.2 概念引入

1. 什么是特性

MSDN 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 attribute，它对程序中的元素进行标注，如类型、字段、方法和属性等。attribute 和 Microsoft .NET Framework 文件的元数据保存在一起，可以用来在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。

我们简单地总结为：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. 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，不是本节的重点，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 = "小王";
        //触发 get 访问器
        Console.WriteLine(myProperty.Name);
        myProperty.Age = 66;
        Console.WriteLine(myProperty.Age.ToString());
        Console.ReadLine();
    }
}

```

3. 区别与比较

通过对概念的澄清和历史的回溯，我们发现特性和属性只是在名称上混淆而已，MSDN 上关于 attribute 的中文解释甚至还是属性，但是作者同意更通常的称呼：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。本节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。

8.3.3 通用规则

下面，将特性的规则做一总结，这些规则是理解 attribute 最重要的条款，必须熟练掌握：

- 定制特性可以应用的目标元素包括：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)，不外乎这些。
- 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，以下表达规则都是有效：[AttributeUsage][Flags]、[AttributeUsage, Flags]、[Flags, AttributeUsageAttribute]、[AttributeUsage(), FlagesAttribute()]。
- attribute 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：

```
namespace InsideDotNet.Conception.AttributeAndProperty
{
    [assembly: MyselfAttribute()] //应用于程序集
    [moduel: MyselfAttribute("小王", 27)] //应用于模块
    class AttributeArea
    {
        //
    }
}
```

- 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(AttributeTargets.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，根据附加信息来决定执行的逻辑分支，降低了系统代码的耦合度。
- 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。

8.3.4 特性的应用

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一简单讨论。我想了解特性，还是以这里为起点，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。

(1) 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，更多的体会还应来自于大家的实践。

(2) 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：

```
[Flags]
enum Animal
{
    Dog = 0x0001,
    Cat = 0x0002,
    Duck = 0x0004,
    Chicken = 0x0008
}
```

因此，以下实现就相当轻松：

```
public static void Main()
{
    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 特性，这里的結果将是“3”。关于位标记，在 9.6 节“简易不简单：认识枚举”中有所论述，详情参见其中的描述。

(3) 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 InsideDotNet.Conception.AttributeAndProperty
{
    class DllImportTest
    {
        [DllImport("User32.dll")]
        public static extern int MessageBox(int hParent, string msg, string caption, int type);

        static int Main()
        {
```

```
        return MessageBox(0, "How to use attribute in .NET", "InsideDotNet", 0);
    }
}
```

(4) Serializable

Serializable 特性表明了应用的元素可以被序列化，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。

(5) 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。在 7.6 节“预处理指令关键字”中对此有所讨论。

还有其他的重要特性，包括：Description、DefaultValue、Category、ReadOnly、Browsable 等，有时间可以深入研究。

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，只要遵守上述的多条规则，实现一个自定义特性其实很容易的，典型的实现方法为：

(1) 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.");
    }
}
```

(2) 应用目标元素

```
[Test("Error Here.")]
public void CannotRun()
{
    //
}
```

(3) 获取元素附加信息

如果没有什么机制来在运行期获取 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, typeof
(TestAttribute));
myAtt.RunTest();
}

```

8.3.5 示例

1. MyselfAttribute

啥也不说了，代码注释中有详细的分析：

```

namespace Anytao.Inside.Ch08.AttributeAndProperty
{
    //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;

        //必须显式的定义其构造函数
        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.")]
[Myself()]
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。

2. TrimAttribute

很多时候，对于 string 属性成员做 Trim 操作，是常常发生的事情。例如，对于回传到数据库服务器的实体数据，需要保证其数据的可靠性，很多时候需要对各个 string 属性成员执行 Trim 检查：

```

user.Name.Trim();
user.Desc.Trim();

DB.ExecuteNonQuery();

```

当实体成员变得很多时，这种代码会导致 Trim 检查漫山遍野地存在于数据层的各个角落，所以需要重新考虑这种过滤操作的设计。通常情况下，以“注入”的方式实现横切式的实现方式，是这种过滤模式的最优选择，而本例的 TrimAttribute 正是这种 Attribute 注入的小技巧：

```
[AttributeUsage(AttributeTargets.Property, Inherited = false, AllowMultiple = false)]
public sealed class TrimAttribute : Attribute
{
    private readonly Type type;

    public TrimAttribute(Type type)
    {
        this.type = type;
    }

    public Type Type
    {
        get { return this.type; }
    }
}
```

然后，通过扩展方法实现对于 TrimAttribute 标记属性进行过滤操作：

```
public static class TrimAttributeExtension
{
    public static void Trim(this object obj)
    {
        Type t = obj.GetType();
        foreach (var prop in t.GetProperties())
        {
            foreach (var attr in prop.GetCustomAttributes(typeof(TrimAttribute), true))
            {
                TrimAttribute tsa = (TrimAttribute)attr;
                if (prop.GetValue(obj, null) != null && (tsa.Type == typeof(string)))
                {
                    prop.SetValue(obj, GetPropertyValue(obj, prop.Name).ToString().Trim(), null);
                }
            }
        }
    }

    private static object GetPropertyValue(object instance, string propertyName)
    {
        return instance.GetType().InvokeMember(propertyName, BindingFlags.GetProperty,
            null, instance, new object[] { });
    }
}
```

这样就可以很好地利用 TrimAttribute 对需要执行 Trim 过滤的数据实体提供注入式的处理逻辑了：

```
public class User
{
    public int Id { get; set; }

    [Trim(typeof(string))]
    public string Name { get; set; }

    public string Desc { get; set; }
}
```

现在，数据层进行 Trim 操作变得简单而优雅：

```
DB.Execute(user.Trim());
```

当然，上述 TrimAttribute 在实现方式上还有很多值得优化和完善的地方。但是，就应用 Attribute 进行过滤、验证操作的设计来说，基本为我们提供了一种解决思路。事实上，在 .NET 提供的 System.ComponentModel.DataAnnotations 验证框架中，正是基于类似的实现逻辑：

```
public class User
{
    public int Id { get; set; }

    [Trim(typeof(string))]
    public string Name { get; set; }

    [Required(ErrorMessage = "Age is required.")]
    [Range(1, 200)]
    public int Age { get; set; }

    [Required]
    [RegularExpression("^[a-zA-Z_\\+-]+(\\.\\.[a-zA-Z_\\+-]+)*@[a-zA-Z-]+(\\.\\.[a-zA-Z-]+)*\\$([a-zA-Z]{2,4})$", ErrorMessage = "A invalid email.")]
    public string Email { get; set; }

    public string Desc { get; set; }
}
```

其中的 Required、Range 还有 RegularExpression 都是一个个 Attribute，用于进行不同方面的验证逻辑注入。通过类似的方式对于数据实体的验证逻辑，完全从数据层的业务逻辑中解放出来，不需要也没有必要再将验证或者过滤这样的非业务逻辑包含在数据层的代码洪流中：

```
static void Main(string[] args)
{
    User user = new User();
    user.Id = 1;
    user.Name = " Wang Tao ";
    user.Email = "aBC";
    user.Desc = " , is a .NET geek. ";

    var context = new ValidationContext(user, null, null);
    var result = new List<ValidationResult>();

    var isValid = Validator.TryValidateObject(user, context, result, true);

    if (!isValid)
    {
        result.ForEach(x => Console.WriteLine(x.ErrorMessage));
    }
    else
    {
        DB.Execute(user);
    }
}
```

你看，干净多了。由此可见，Attribute 作为.NET 特有的语言特性，在实际的应用中发挥着巨大的作用和贡献，为实现优雅而简单的语言特性提供了超酷的体验。当然，其巨大的潜力还不止于此，留待开发者在实际的应用中探索、挖掘和分享。

8.3.6 结论

Attribute 是.NET 引入的一大特色技术，更深层次的应用，例如序列化、程序安全性、对 JIT 的优化调试、设计模式、记录文件名或其他附件信息等多方面的挖掘都是值得期待的，这就是.NET 在技术领域带来的百变魅力。我们在享受这种便利之前，需要首先了解如何使用这种便利的工具，这正是本节为您呈现的魅力体验。

8.4 面向抽象编程：接口和抽象类

本节将介绍以下内容：

- 面向对象思想：多态
- 接口讨论
- 抽象类讨论

8.4.1 引言

接口和抽象类，就是一对欢喜冤家。接口和抽象类，成就了面向抽象编程的设计思想，广泛应用于设计模式的很多精妙的设计中，同时也是实现多态的基础技术之一。而对二者应用，常常交叉而生，进行透彻的分析势在必行，本节的工作即是对接口和抽象类在各方面的应用做一归纳总结，从中挖掘各自的应用场合与实质。

8.4.2 概念引入

1. 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，当一个类实现某个接口时，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。对接口的其他讨论，请参见 1.5 节“玩转接口”的描述。

定义方法为：

```
public interface IComparable
{
    int CompareTo(object o);
}

public class TestInterface: IComparable
{
    public TestInterface()
    {

    }

    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
    TestInterface tester = o as TestInterface;

    if (tester != null)
    {
        //实现抽象方法，调用int的CompareTo实现
        return _value.CompareTo(tester._value);
    }
    return 0;
}
}

```

2. 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override方法来完成。

```

//定义抽象类
abstract public class Animal
{
    //定义静态字段
    static 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 index]
    {
        get {return null;}
        set { ;}
    }
}

//实现抽象类
public class Dog: Animal
{
    public override int Id
    {
        get {return _id;}
        set {_id = value;}
    }

    public override void Eat()
}

```

```

    {
        Console.WriteLine("Dog Eats.");
    }
}

```

8.4.3 相同点和不同点

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。

2. 不同点

- 接口支持多继承；抽象类不能实现多继承。注意，严格意义上来说，接口继承类应该称为类实现接口。
- 接口只能定义抽象规则；抽象类既可以定义规则，还可以提供已实现的成员。
- 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；因为继承不支持，所以抽象类在实现回调时有局限性。在此，实现一个简单的接口支持回调的示例，例如：

```

public interface IMyInterface
{
    void DoWork();
}

public class MyClassWithCallback
{
    private IMyInterface _myInterface;

    public void AddCallback(IMyInterface myInterface)
    {
        _myInterface = myInterface;
    }

    public void RemoveCallback()
    {
        _myInterface = null;
    }

    public void DoWork()
    {
        if (_myInterface != null)
            _myInterface.DoWork();
    }
}

public class MyBaseClass : IMyInterface
{
    public void DoWork()
    {
        Console.WriteLine("Call interface method.");
    }

    public static void Main()
    {
        MyClassWithCallback mc = new MyClassWithCallback();
        IMyInterface mi = new MyBaseClass();
    }
}

```

```

        mc.AddCallback(mi);
        mc.DoWork();
        mc.RemoveCallback();
    }
}

```

示例实现的接口回调中，细心的读者肯定很容易发现将 `IMyInterface` 接口实现为抽象类，同样能够“胜任”回调的功能。因此，对于“继承不支持”的说法，有必要进行一点补充：在示例中，类 `MyBaseClass` 通过实现 `IMyInterface` 接口，具有了回调 `DoWork` 方法的能力。对于 `MyBaseClass` 类来说，继承抽象类和实现接口，都可以实现这一功能。然而，如果 `MyBaseClass` 本身已经存在继承关系时，则不能同时有其他的继承，但是接口却可以有效的解决这一局限性，`MyBaseClass` 在实现 `IMyInterface` 接口的同时还可以继承于其他的基类。

- 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。

通过比较异同，我们只能说接口和抽象类各有所长，而无优劣。在实际的编程实践中，需要酌情量材，相信以下的经验积累，能给大家一些启示，这些经验大都来源于经典，经得起时间考验。在规则与场合中，我们学习这些经典，最重要的是学以致用。

3. 规则与场合

- (1) 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。
- (2) 借助接口和抽象类，23个设计模式中的很多思想被巧妙地实现了，我们认为其精髓简单说来就是：面向抽象编程，通过封装变化来实现实体之间的关系。
- (3) 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。
- (4) 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。
- (5) 接口多定义对象的行为；抽象类多定义对象的属性。
- (6) 接口定义可以使用 `public`、`protected`、`internal` 和 `private` 修饰符，大部分接口都被定义为 `public`，另外方法的访问级别不能低于接口的访问级别，否则将导致编译错误。
- (7) “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。
- (8) 尽量将接口设计成功能单一的功能块，以.NET Framework 为例，`IDisposable`、`IComparable`、`IEquatable`、`IEnumerable` 等都只包含一个公共方法。
- (9) 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，请坚持这些良好的编码规范。
- (10) 在接口中，所有的方法都默认为 `public`。
- (11) 如果预计会出现版本问题，可以创建抽象类。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。

(12) 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现。

(13) 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。

(14) 在抽象方法声明中不能使用 static 或 virtual 修饰符。

以上的规则，是实践面向抽象编程的重要归纳，在实际的应用中应该灵活取舍，以面向对象的原则和设计模式的思想来做出适当的选择。

8.4.4 经典示例

1. 绝对经典

.NET Framework 是学习的最好资源，有意识地研究 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，笔者有以下的建议：

- 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。

2. 别样小菜

下面的实例，着眼于概念和原则的把握，真正的应用一般来自于具体的需求规范。其设计结构如图 8-3 所示。

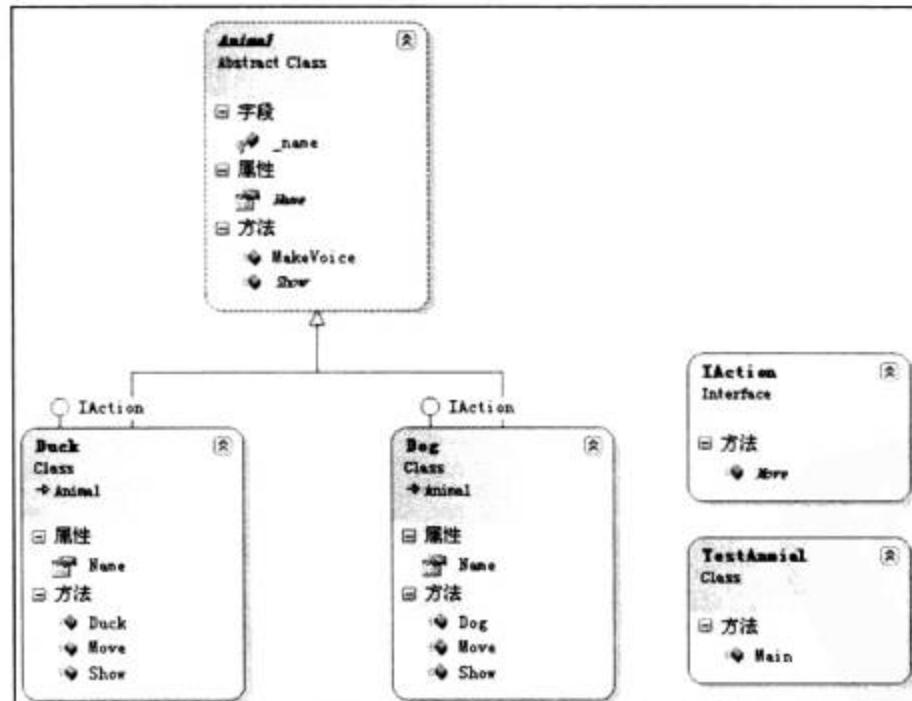


图 8-3 接口和抽象类示例

(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) 客户端调用

```

public class TestAnimal
{
    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();
    }
}

```

8.4.5 他山之石

正所谓真理越辨越明，最后将 MSDN 中对于接口和抽象类的建议做一总结，算是对本节所述规则的有效补充，主要包括：

- 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。

8.4.6 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，是因为深刻的理解这两个面向对象的基本概念，对于盘活面向对象的抽象化编程思想至关重要。本节基本概括了接口和抽象类的概念、异同和使用规则等核心内容。

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 Guthrie 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。对于接口和抽象类正是如此，多用而知其然，多想而知其奥。

8.5 恩怨情仇：is 和 as

本节将介绍以下内容：

- 类型转换
- is 和 as 模式小议

8.5.1 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，想要完全把握系统数据的类型安全，往往力不从心。现在，这一切已经在微软大牛们的设计框架中为我们解决了。在.NET中，一切类型都必须继承自 System.Object 类型，因此我们可以很容易根据 GetType 方法来获得对象的准确类型。那么.NET中的类型转换，还有哪些地方值得思考呢？

8.5.2 概念引入

类型转换包括显式转换和隐式转换，在.NET中类型转换的基本规则如下：

- 任何类型都可以安全地转换为其基类类型，可以由隐式转换来完成。
- 任何类型转换为其派生类型时，必须进行显式转换，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 Convert 类实现类型转换。
- 除了 string 以外的其他基本类型都有 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）。

8.5.3 原理与示例说明

浅谈了类型转换的几个普遍关注的方面，是时候将主要精力放在 is、as 操作符的应用上了。类型转换将是个较大的话题，在 5.2 节“品味类型——值类型与引用类型”中有所描述。

is 和 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。

1. is 模式

- 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：

```
class ISSample
{
    public static void Main()
    {
        object o = new object();

        //执行第一次类型兼容性检查
        if (o is ISSample)
        {
            //执行第二次类型兼容性检查
            ISSample a = (ISSample)o;
        }
    }
}
```

2. as 模式

- 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。
- as 必须和引用类型一起使用。

其典型用法为：

```
class ASSample
{
    public static void Main()
    {
        object o = new object();

        //执行一次类型兼容检查
        ASSample b = o as ASSample;
        if (b != null)
        {
            //执行关于 b 的操作
        }
    }
}
```

对于上述 as 操作，在语义上等效于：

```
ASSample b = o is ASSample ? (ASSample)o : null;
```

但是，实现的语法更加简洁明了，且不会引发任何异常，在类型转换时值得推荐。

8.5.4 结论

综上比较，is 模式和 as 模式，提供了更加灵活的类型转型方式，不过 as 模式较 is 模式在执行效率上更胜一筹，我们在实际的编程中应该量材而用。通常来说，is 用于进行类型判断，而 as 用于类型转型。

8.6 貌合神离：覆写和重载

本节将介绍以下内容：

- 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

8.6.1 引言

覆写（override）与重载（overload），是成就.NET 面向对象多态性的基本技术之一，两个概念貌似而实质决然不同，有必要讨论清楚其区别，消除误解，而更重要的是关注其在多态中的应用。

在本书中，我们先后在 1.2 节“什么是继承？”、1.4 节“多态的艺术”和 7.1 节“把 new 说透”都有关于这一话题的点滴论述，本节以专题的形式对此再做深度讨论，相关的内容请参照前文。

8.6.2 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()
    {
    }

    //覆盖基类成员
    public override void MyFunc(string str)
    {
        //在子类中访问父类成员
        base.MyFunc(str);
    }

    //覆盖基类抽象方法
    public override void Func()
    {
        //base.Func();
        //throw new Exception("The method or operation is not implemented.");
    }
}

```

1. 覆写基础

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中。当子类重写父类的虚函数后，父类对象就可以根据赋予它的不同子类指针动态调用子类的方法。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。

关于：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、abstract 和 override 修饰。

关于：抽象方法

抽象方法就是以 abstract 关键字修饰的方法，抽象方法可以看做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. 重载基础

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按返回值重载，但是这种重载方式不被推荐，本书在 7.5 节“转换关键字”部分对其实现方法和弊端有所讨论，可供参考。
- .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。

8.6.3 在多态中的应用

多态性，简单地说就是“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
{
    //覆盖基类成员
    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，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 1.2 节“继承的本质”和 1.4 节“多态的艺术”。

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地为方法重载提供了很好的交互手段，如图 8-4 所示。

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。

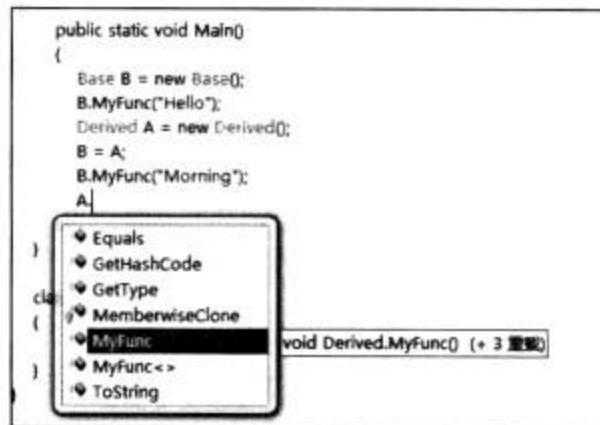


图 8-4 Visual Studio 的智能感知

8.6.4 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
{
    //覆盖基类抽象方法
    public override void Func()
    {
        base.Func();
    }
}
```

- 虚方法不能是静态的、密封的。
- 覆盖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，不可相互替代。

在表 8-1 中，将覆盖与重载做一总结性的对比。

表 8-1 覆写和重载的异同

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，

8.6.5 进一步的探讨

1. override 与 new 的版本控制

在.NET中，override和new关键字还有一个显著的功能就是版本控制。随着类库的不断发展，基类及其派生类的版本控制也在不断前进，同时也有必要保证向后兼容。这是我们进行系统设计需要考虑的一个问题。主要涉及两个方面的兼容：一，如果在基类后引入一个与派生类同名的方法，不会出现意外的问题，这正是new隐藏基类成员所能解决的问题；二，在派生类中，显式的重写基类方法的实现，这正是覆盖所解决的问题。

```
class Base
{
    public virtual void MyFunc(string str)
    {
        Console.WriteLine("{0} in Base", str);
    }
}

class Derived: Base
{
    //覆盖基类成员
    public override void MyFunc(string str)
    {
        Console.WriteLine("{0} in Derived", str);
    }

    public static void Main()
    {
        Base bd = new Derived();
        bd.MyFunc("Hello");
    }
}
```

在上例中，如果派生类Derived的方法MyFunc定义为override，则执行结果为Hello in Derived；如果MyFunc定义为new，则执行结果为Hello in Base。可见，针对不同的需求进行合理设计，是控制版本兼容的关键。因此，我们应该遵守以下的规则：

- .NET默认为非虚方法，因此必须将基类方法定义为虚方法，以virtual修饰。
- 在派生类中定义方法为override或者new。使用override表示覆盖基类成员，实现派生类自己的版本；使用new表示隐藏基类的虚方法，也就是该方法独立于基类的方法。
- 派生类方法如果未被定义为override或new，则编译器将发出警告，并默认定义为带new的方法。
- 在派生类中使用base关键字调用基类方法。

2. 泛型方法的覆盖和重载

在.NET 2.0中，开始引入泛型技术，对于覆盖和重载来说在某些方面又有了新的规则，例如相同的参数列表也可以构成重载，在此我们还是以简单的示例来初步认识一下泛型方法的覆盖和重载。

首先是泛型方法的覆盖：

```
class BaseClass
{
    public virtual void MyFunc<T>(T t) where T: new()
    {
```

```

        Console.WriteLine("{0} in Base", t.ToString());
    }

class DerivedClass: BaseClass
{
    public override void MyFunc<T>(T t)
    {
        Console.WriteLine("{0} in Derived", t.ToString());
    }

    public static void Main()
    {
        BaseClass b = new DerivedClass();
        b.MyFunc<int>(100);
        b.MyFunc<char>('A');
        //b.MyFunc<string>("Hello"); //出现编译错误
    }
}

```

关于泛型方法的覆写，值得注意的是，如果在基类泛型方法中有约束，则在子类重写的时候，不能加入约束，不能添加新的约束，只能默认继承父类约束。否则，将出现编译时错误，提示约束不能被指定。例如，在示例中，`b.MyFunc<string>("Hello")`，出现编译错误的原因是 `string` 类型没有公共的无参构造函数，可见在子类中已经默认继承了父类的约束。

其次，是泛型方法的重载：

```

class BaseClass
{
    public virtual void MyFunc<T>(T t) where T: new()
    {
        Console.WriteLine("{0} in Base", t.ToString());
    }

    public void MyFunc(string t)
    {
        Console.WriteLine(t);
    }
}

class DerivedClass: BaseClass
{
    public static void Main()
    {
        BaseClass c = new BaseClass();
        c.MyFunc<int>(100);
        c.MyFunc("100");
    }
}

```

上述示例中，`public virtual void MyFunc<T>(T t)`和`public void MyFunc(string t)`构成方法重载，而下面的几种情况应该引起读者的特别关注，因为泛型的一些特性，在方法重载时有些隐藏的规则需要挖掘，包括：

```

public void MyFunc<T>(T t);
public void MyFunc<U>(U u);

```

不能构成泛型方法重载，原因是编译器无法确定泛型类型 `T` 和 `U` 是否不同，从而无法在编译时确定这两个方法是否不同，因为实例化时 `T` 和 `U` 可能会传入相同的类型。

```
public void MyFunc<T>(T t) where T : A;
public void MyFunc<T>(T t) where T : B;
```

不能构成泛型方法重载，原因同样是在编译时编译器无法确定约束条件中的 A 和 B 是否不同，从而也无法在编译时确定这两个方法是否不同。

8.6.6 结论

深入地理解覆写和重载，是对多态特性和面向对象机制的有力补充，本节从基本概念到应用领域将两个概念一一梳理，通过对比整理区别，给覆写和重载以更全面的认知角度，同时也更能从侧面深入地了解运行时多态与编译时多态的不同情况。

8.7 有深有浅的克隆：浅拷贝和深拷贝

本节将介绍以下内容：

- 对象的克隆
- 浅拷贝和深拷贝
- 应用与规则

8.7.1 引言

在.NET类库中，对象克隆广泛存在于各种类型的实现中，凡是实现了`ICloneable`接口的类型都具备克隆其对象实例的能力，显然`ICloneable`接口就像是类型的后缀一样，在.NET类库中四处招摇。从`System.String`、`System.Array`、`System.Data.Dataset`、`System.Delegate`等典型类型中，都可见`ICloneable`接口被实现的身影。因此，作为拓展.NET类库的窗口之一，我们有必要从`ICloneable`接口开始，来深入了解对象克隆这一典型话题，同时进一步理解CLR的类型系统特性。

8.7.2 从对象克隆说起

对象克隆，也就是类型实例的拷贝，在实际的系统应用中，对象克隆并不少见，例如值类型之间的相互赋值，引用类型之间的相互赋值，值类型的装箱操作等，其实都进行着对象的克隆操作。然而，必须明确的是，不是所有的.NET对象实例都可以被克隆，事实上只有实现了`ICloneable`接口的类型，才允许其实例被克隆。

首先我们先看看`ICloneable`接口的定义吧。

```
public interface ICloneable
{
    object Clone();
}
```

可见，`ICloneable`只定义了一个无参`Clone`方法，并且返回`object`对象。所以在自定义类中需要开发人员自行完成`Clone`方法，来创建新的对象实例。从创建实例与原来对象实现的初始化状态执行程度上看，对象克隆可分为浅拷贝和深拷贝。

浅拷贝（Shallow Copy），指对象的字段被拷贝，而字段引用的对象不会被拷贝，拷贝对象和源对象仅仅是引用名称有所不同，但是它们共用一份实体。对任何一个对象的更改，都会影响到另一个对象。大部分的引用类型，实现的都是浅拷贝，引用类型对象之间的赋值，就是复制一个对象引用地址的副本，而指向的对象实例仍然是同一个。

```
class Student
{
    public string Name;
    public Int32 Age;

    public Student(string name, Int32 age)
    {
        Name = name;
        Age = age;
    }

    public void ShowInfo()
    {
        Console.WriteLine("{0}'s age is {1}", Name, Age);
    }
}

class Test_ObjectClone
{
    public static void Main()
    {
        Student s1 = new Student("Wang", 22);
        //执行浅拷贝
        Student s2 = s1;
        s2.Age = 27;
        s1.ShowInfo();
    }
}

//执行结果
//Wang's age is 27
```

从上例中可知，`s2 = s1` 执行的就是浅拷贝，`s1`、`s2` 对象执行了托管堆中同一块内存地址，因此对 `s2` 的更改也同样作用于 `s1`，我们可以从执行结果中得到答案。

深拷贝（Deep Copy），指对象的字段被拷贝，同时字段引用的对象也进行了拷贝。深拷贝创建的是整个源对象的结构，拷贝对象和源对象相互独立，不共享任何实例数据，修改一个对象不会影响到另一个对象。显然，值类型之间的赋值操作，执行的就是深拷贝。

```
public static void Main()
{
    Int32 i = 100;
    Int32 j = i;
    j = 200;
    Console.WriteLine(i);
}

//执行结果
//100
```

在示例中，`j = i` 赋值操作执行了深拷贝动作，对 `j` 的修改不会影响到 `i`，同样可以从执行结果得到印证。因此，从上面的分析可知，值类型之间的赋值一般执行的是深拷贝，例如 `Int32`、结构体（struct）和枚

举(enum)等,根据值类型的特点我们不难理解这一点;而引用类型的赋值一般执行的是浅拷贝,其原因主要为以下几个方面:

- 性能因素:深拷贝操作涉及对源对象整个结构的拷贝,创建一个大对象的副本对性能的影响较大。
- 可行性因素:字段的实例对象并非可克隆对象,使得深拷贝复制源对象每个字段的引用对象变得不可行。同时,很多情况下,深拷贝是不可能实现的,因为它要求对象树上的所有对象都实现为深拷贝。
- 稳定性因素:某些类型的字段,涉及循环引用的情况,致使深拷贝陷入死循环。
- 简单性因素:浅拷贝在实现上较深拷贝简单。

当然,也不是所有可克隆的引用类型都实现为浅拷贝,例如 System.String 在表现上类似于值类型可以进行直接的复制,并且返回一个新的字符串对象; System.Xml.XmlNode 的 Clone 方法也实现了深拷贝,具体的实现规则取决于具体的需求语义。

8.7.3 浅拷贝和深拷贝的实现

在了解对象克隆概念的基础上,我们可知对象克隆的关键就是根据其数据成员的类型来完成一定的拷贝操作。如果数据成员包含值类型和 System.String 类型,则会在 Clone 方法中创建一个新的对象,并将源对象中的各个数据成员复制给新对象的数据成员;如果数据成员为引用类型,则要考虑实现为浅拷贝还是深拷贝。

那么,我们通过什么样的方式来实现 Clone 方法的浅拷贝和深拷贝语义呢?最明智的选择,莫过于从.NET类库中取经,用 Reflector 工具来查看.NET类库的具体实现,例如以 System.Array 类的 Clone 方法作为启蒙,来借鉴一下.NET中实现对象克隆的基本方式。

System.Array.Clone 方法的实现细节为:

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

从代码中可知,在 Clone 方法中通过调用基类 System.Object 的受保护方法 MemberwiseClone 就可实现 Array 类型的浅拷贝。因为在.NET中,一切方法都最终继承于 System.Object 类,所以开发人员可以通过调用 MemberwiseClone 方法为自定义类型实现浅拷贝。事实上, MemberwiseClone 方法完成的工作就是初始化一个新的对象,并遍历源对象的所有实例字段,然后将新对象数据成员的值拷贝为源对象各个对应数据成员的值,如果实例字段为引用类型则只是复制该引用。

下面,结合上文中的 Student 类来构建一个 Enrollment 学生登记类,并将其作为引用类型的数据成员,来考察浅拷贝和深拷贝的实现方法。

(1) 浅拷贝实现

学生登记类 Enrollment 的浅拷贝实现细节为:

```
class Enrollment : ICloneable
{
    // 定义引用类型数据成员
    public List<Student> students = new List<Student>();

    public void ShowEnrollmentInfo()
```

```

    {
        Console.WriteLine("Students enrollment information:");
        foreach (Student s in students)
        {
            Console.WriteLine("{0}'s age is {1}", s.Name, s.Age);
        }
    }

    public object Clone()
    {
        //执行浅拷贝
        return MemberwiseClone();
    }
}

```

下面是测试代码，用以考查在浅拷贝和深拷贝两种情况下，展示其执行结果的差别。

```

class Test_ObjectClone
{
    public static void Main()
    {
        Enrollment sourceStudentsList = new Enrollment();
        sourceStudentsList.students.Add(new Student("王小二", 27));
        sourceStudentsList.students.Add(new Student("张三", 22));

        //实现对象的克隆，执行浅拷贝还是深拷贝，
        //取决于Clone方法的具体实现
        Enrollment cloneStudentsList = sourceStudentsList.Clone() as Enrollment;

        sourceStudentsList.ShowEnrollmentInfo();
        cloneStudentsList.ShowEnrollmentInfo();

        //修改克隆对象的数据成员
        cloneStudentsList.students[1].Name = "李四";
        cloneStudentsList.students[1].Age = 36;

        sourceStudentsList.ShowEnrollmentInfo();
        cloneStudentsList.ShowEnrollmentInfo();
    }
}

```

从执行结果可见，修改了克隆对象 `cloneStudentsList` 的数据成员，也同时影响了源对象 `sourceStudentsList` 的结果，因此在此实现的是浅拷贝。

(2) 深拷贝实现

下面通过在 `Clone` 方法中构造新实例对象的办法，来实现 `Enrollment` 类的深拷贝，在此只列出区别于浅拷贝的代码片段，具体为：

```

class Enrollment : ICloneable
{
    //定义引用类型数据成员
    public List<Student> students = new List<Student>();

    //定义默认构造函数
    public Enrollment()
    {

    }

    //提供实现深拷贝的私有实例创建构造函数

```

```

private Enrollment(List<Student> studentList)
{
    foreach (Student s in studentList)
    {
        students.Add((Student)s.Clone());
    }
}

public object Clone()
{
    //执行深拷贝
    return new Enrollment(students);
}
}

```

从执行结果来看，修改了克隆对象 `cloneStudentsList` 的数据成员，并未影响源对象 `sourceStudentsList`，因此在此实现的是深拷贝。

浅拷贝的实现方法相对简单，而深拷贝的实现方法，远不限于本节所提供的一种方式。另外，对于未实现 `ICloneable` 接口但是可序列化的对象来说，可行的做法是通过序列化和反序列化方式来达到对象克隆的效果。

在此我们只关注了引用类型的对象克隆实现方法，关于值类型的实现方式其实类似于本节介绍的方法，需要开发人员在其自定义值类型中实现 `ICloneable` 接口，至于实现浅拷贝或者深拷贝语义，就要根据具体需求来对症下药了。

8.7.4 结论

本节从对象克隆的话题入手，分析了关于对象克隆的浅拷贝和深拷贝问题，并深入讨论了实现对象克隆的具体方法。

从另一方面来看，我们又一次涉及对值类型与引用类型、装箱与拆箱等话题的回顾，加深了对这些问题的进一步了解。从技术认知的角度来说，技术话题都是相辅相成的，我们总是在不断探索新目标的同时，又能有效的实现对原有知识的补充，这是技术学习的科学方法论。

8.8 动静之间：静态和非静态

本节将介绍以下内容：

- 静态与非静态特征对比解析
- 静态成员与静态方法的内存分配解析
- 诠释静态构造函数、静态类、静态成员与静态方法

8.8.1 引言

在.NET 中，静态特征是一个不可或缺的基本要素，.NET Framework 类库本身就实现了很多有静态特征的类型。本节从静态特征与非静态特征的对比角度出发，逐一品读静态特征的各个概念和区别，将静态类、静态构造函数、静态字段、静态属性和静态方法逐一剖析，深入了解。

8.8.2 一言蔽之

在面向对象的世界里，大部分的情况都是实例特征主宰天下，类相当于一个特征模板，而对象则是类特征的拷贝，并且独立于其他对象来操作这些特征。但是，在某些情况下，我们需要某些特征被所有的对象实体所公有，因此有必要实现一种基于类的特征，而不是基于实例对象的特征机制，这就是静态特征。

在 Petshop 4.0 数据库辅助类中，连接字符串对各个实例来说是相对公用的变量，在各个实例中来实现显然不如提取出来以公共变量的形式实现，因此就需要建立静态数据库连接成员，为各个操作实例所共享。

单例模式作为一种典型的设计模式，用于保证在系统中只有一个对象实例，被广泛使用在程序设计中，一般通过一个静态成员、一个静态属性（或方法）和一个私有构造函数来实现，其典型的实现方法为：

```
class Singleton
{
    //定义一个静态成员
    private static Singleton _instance = null;

    //定义一个静态属性（或方法）
    public static Singleton Instance
    {
        get
        {
            if (_instance == null)
            {
                _instance = new Singleton();
            }

            return _instance;
        }
    }

    //定义一个私有构造函数
    private Singleton()
    {
    }
}
```

当然，单例模式远不只此一法。而通过将对象的所有字段设为静态成员的 MonoState 模式，同样可以利用静态特性来实现这一需求，在此就不多做讨论。

这些应用都体现了静态特征的优势所在。值得注意的是，本节对静态这一概念的分析是以 C#语言来展开的，而对应于 VB.NET 的类似概念被称为“共享”，以 shared 关键字标识。

在.NET 中，关于静态特征的概念，主要包括静态字段、静态属性和静态方法等，下面我们通过静态特征与实例特征的对比来逐步剖析对静态特征的理解。

8.8.3 分而治之

1. 静态类与非静态类

一个类如果只包含静态成员和静态方法，则该类可以定义为静态类，定义方法是给类加上 static 修饰符，例如：

```

static class MyStatic
{
    //声明静态字段
    private static string staticString = "static string";

    //声明静态属性
    private static string StaticString
    {
        get { return staticString; }
        set { staticString = value; }
    }

    //实现静态方法
    public static void ShowMsg()
    {
        Console.WriteLine(StaticString);
    }

    public static void Main()
    {
        //访问静态方法
        MyStatic.ShowMsg();
    }
}

```

静态类与非静态类，在以下方面值得总结：

- 静态类只能包含静态成员和静态方法，否则会抛出编译错误；而非静态类既可以包含非静态成员和非静态方法，也可以包含静态成员和静态方法。
- 静态类不可实例化；非静态类可以实例化。不管是静态类还是非静态类，对静态成员和静态方法的调用都必须通过类来实现，例如示例中以 MyStatic.ShowMsg() 来访问静态方法。
- 相对于非静态类来说，静态类有一些特点值得应用，在.NET 类库中就有很多的类被实现为静态，例如 System.Console 就是一个典型的静态类；另外.NET 2.0 新增的 MemberShip 中，修建、修改用户都以静态方法来实现。
- 如果一个类只包含静态成员和静态方法，应该将该类标记为 static，并提供私有的构造函数来避免实例创建，这其实正是一种 MonoState 模式的体现。

2. 静态构造函数与实例构造函数

静态构造函数，用于初始化类中的静态成员，包括静态字段和静态属性。静态构造函数不能带参数、不能有访问修饰符也不能被调用，通常由.NET 运行库在第一次调用类成员之前执行。

```

class ClassHelper
{
    //定义静态字段
    public static string StaticString = "Initial static string.";
    //定义非静态字段
    public string NonStaticString = "Initial non static string.";

    //静态无参构造函数
    static ClassHelper()
    {
        //只能初始化静态成员
        StaticString = "Change static string in static constructor.";
    }
}

```

```

}

//实例构造函数
public ClassHelper()
{
    //初始化非静态成员
    NonStaticString = "Change non static string in instance constructor. ";
    //初始化静态成员
    StaticString = "Change static string in instance constructor.";
}

public static void Main()
{
    Console.WriteLine(ClassHelper.StaticString);

    ClassHelper ch = new ClassHelper();
    Console.WriteLine(ClassHelper.StaticString);
    Console.WriteLine(ch.NonStaticString);
}
}

```

与实例构造函数相比，有以下几个规则值得注意：

- 静态构造函数，可以和无参的实例构造函数同存。虽然参数列表相同，但是二者执行的时间不同，静态构造函数在运行库加载类时执行；而实例构造函数则在实例创建时执行。如上例中，两个无参的构造函数，区别仅在于修饰符的不同。
- 静态构造函数，只能对静态成员进行初始化操作，不能作用于非静态成员；而实例构造函数，可以初始化实例成员，也可以初始化静态成员，但是静态只读字段除外。例如 StaticString 在静态构造函数和实例构造函数中均可进行初始化，根据执行结果可知两种初始化均是有效的。
- 静态构造函数只被执行一次，而且.NET 运行库也无法确定静态构造函数什么时候被执行；而实例构造函数可以在多次实例创建时被执行多次。
- 一个类只能有一个静态构造函数；而一个类可以有多个实例构造函数。
- 静态字段的初始值，在静态构造函数调用之前被指定，例如 StaticString 的初始值在调用其静态构造函数之前就被指定，其执行顺序如图 8-5 所示，所以在测试代码中第一次执行 ClassHelper.StaticString 时获得的是在静态构造函数中新指定的值。
- 静态成员可以在声明时初始化，也可以通过静态构造函数进行初始化，这两种初始化都只能被执行一次。一般来说，简单的静态成员在声明时进行初始化即可，而复杂的静态成员则选择在静态构造函数中进行初始化较佳。
- 关于构造函数的执行顺序，大体上来说是以如图 8-5 所示的顺序来进行的。

构造函数的执行过程是个比较复杂的技术内容，在此仅仅以过程化来简单的理解其执行。另外，需要特别指出的是，执行构造函数时，编译器会首先执行基类的构造函数，然后才执行子类的构造函数，并依次类推执行到 System.Object 类为止；同时，静态成员的内存分配、成员初始化和构造函数都只被执行一次，下一次创建对象实例时，只会执行实例成员的初始化过程。

3. 静态成员与实例成员（静态字段和静态属性）

静态成员主要包括静态字段和静态属性，正如前文所说，静态成员可以实现

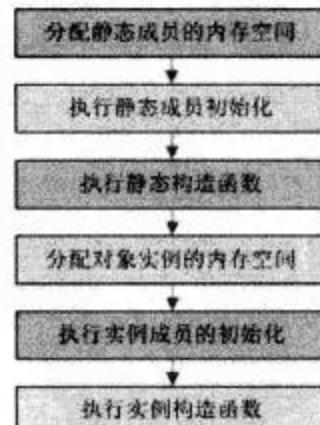


图 8-5 对象的初始化顺序

类中能够被所有实例对象共享的数据，例如在一个缴费登记系统中，消费总额是所有消费的总和，以静态成员来实现这一数据特征有以下好处：

- 不需要在各个对象中重复同样的数据，从性能角度来说，静态成员可以避免不必要的数据冗余。
- 静态成员提供了实现共享数据的有效机制，避免了以附加操作来处理每个对象都共享的数据信息，例如下面示例中的消费总额处理。

```
class Consumer
{
    //定义实例成员：每个人的消费
    private double cost;

    public double Cost
    {
        get { return cost; }
        set { cost = value; }
    }

    //定义静态成员：总计消费
    private static double costAll;

    public static double CostAll
    {
        get { return Consumer.costAll; }
        set { Consumer.costAll = value; }
    }

    public void AddCost()
    {
        costAll += cost;
    }

    public void ShowCost()
    {
        Console.WriteLine(CostAll);
    }
}

class Test_Consumer
{
    public static void Main()
    {
        Consumer c1 = new Consumer();
        c1.Cost = 5.25;
        c1.AddCost();
        Consumer c2 = new Consumer();
        c2.Cost = 3.23;
        c2.AddCost();
        c1.ShowCost();
        c2.ShowCost();
    }
}
```

在示例中，消费总额实现为静态成员，`c1.ShowCost()`和`c2.ShowCost()`的执行结果是相同的，可见静态成员为所有对象所共享；而每个人的消费金额则实现为实例成员，为各个对象特有的数据特征。可见，通过静态成员有效地避免在客户端频繁累加消费总额的操作，而且也避免了处理不当造成的遗漏问题。

关于静态成员，有以下几点总结：

- 静态成员包括静态字段和静态属性，静态字段一般实现为 private，而静态属性一般实现为 public，以体现类的封装原则，可以参考 1.3 节“封装的秘密”来了解字段和属性的区别。
- 静态成员和类相关联，不依赖于对象而存在，只能由类访问，而不能由对象访问；实例成员和具体对象相关联，只能由对象实体访问，而不能由类访问。
- 静态成员属于类所有，无论创建多少实例对象，静态成员在内存中只有一份；实例成员属于类的实例所有，每创建一个实例对象，实例成员都会在内存中分配一块内存区域。

4. 静态方法与实例方法

类似于静态成员共享数据段，静态方法共享代码段。静态方法以 static 关键字标识，否则为实例方法。

```
class MyMethodClass
{
    private static string staticString = "A static string";
    private string nonStaticString = "Not a static string";

    //静态成员
    //不能修饰为 virtual, abstract 或 override
    public static void StaticMethod()
    {
        //只能访问静态成员
        //不能以 this 访问
        Console.WriteLine(staticString);
    }

    //实例成员
    public void NonStaticMethod()
    {
        //可以访问静态成员
        Console.WriteLine(staticString);
        //也可以访问实例成员
        //可以以 this 关键字访问
        Console.WriteLine(this.nonStaticString);
    }
}

class MyMethodClassDerived : MyMethodClass
{
    //派生类定义及实现
}

class Test_MyMethodClass
{
    public static void Main()
    {
        MyMethodClass mmc = new MyMethodClass();
        //实例方法只能由对象访问
        mmc.NonStaticMethod();
        //静态方法只能由类访问
        MyMethodClass.StaticMethod();
        //派生类可以访问静态方法，但是不能覆写静态方法
        MyMethodClassDerived.StaticMethod();
    }
}
```

对应静态方法和非静态方法，我们有以下的规则值得总结：

- 性能上，静态方法和实例方法差别不大。所有方法，不管是静态方法还是实例方法，都是在 JIT 加载类时分配内存，不同的是静态方法以类名引用，而实例方法以对象实例引用。创建实例时，不会再为类的方法分配内存，所有的实例对象共用一个类的方法代码。因此，静态方法和实例方法的调用，区别仅在于实例方法需要当前对象指针指向该方法，而静态方法可以直接调用，在性能上的差别微乎其微。
- 静态方法只能访问静态成员和静态方法，可以间接通过创建实例对象来访问实例成员和实例方法；而实例方法可以直接访问实例成员和静态成员，也可以直接访问实例方法和静态方法。
- 静态方法只能由类访问；实例方法只能由对象访问。
- 静态方法中不能引用 this 关键字访问，否则会引起编译时错误；而实例方法可以引用 this 关键字。
- 静态方法不能被标记为 virtual、abstract 或者 override，静态方法可以被派生类访问，但是不能被派生类覆写。
- Main 方法为静态的，因此 Main 方法中不能直接访问 Main 所在类的实例方法和实例成员。
- 鉴于线程处理的安全性，应该避免提供改变静态状态的静态方法。原因是，如果多线程同时执行该代码时，有可能造成线程处理错误。因此，静态状态必须是线程安全的。
- 静态方法一定程度上是一种结构化的语言机制，从面向对象的角度考虑，静态方法适合实现系统中边缘性的非业务需求，例如提供通用的工具类实现。

8.8.4 结论

通过一连串的对比，关于静态特征与实例特征的各个概念，我们就有了清晰的了解和认知。同时，通过内存分配、性能差别与应用状况等方面，对静态特征进行了全面而深入的剖析。动静之间，承载的是.NET 对不同数据特征的处理机制的灵活权衡，也是对自身程序设计能力的重新审视与提高。

8.9 集合通论

本节将介绍以下内容：

- 关于集合的讨论
- .NET 集合类研究
- 集合的应用

8.9.1 引言

集合（Collection），是用于管理对象的容器类。.NET 框架类库中提供了丰富的集合类型来完成对不同对象集合的管理，历数集合类型，例如：Array、ArrayList、BitArray、Stack、Hashtable、LinkedList、SortedDictionary、Queue、SortedList、List<T>，面对眼花缭乱的.NET 集合类，站在十字路口上，我们该何去何从？

理解集合，了解集合，是程序设计的必经之路，本节从集合的通用特性开始，展开对.NET集合类型的解析，重点讨论其应用角度和使用规则，并以实现自定义集合作为了解集合的又一体验。

8.9.2 中心思想——纵论集合

1. 何为集合，为何而集合

集合的本质是管理对象的容器，它提供了这样一种机制来管理对象：一方面集合将独立的对象汇集成群集，作为一个群集来管理，以便进行整体性操作；而另一方面，集合可以方便地获取群集中的个体，进行个性化操作。

在.NET中，集合被封装为对象，这就意味着我们可以用面向对象的方式来管理与应用集合。一个集合类型在本质上都是一个类，例如ArrayList、Hashtable等，通过封装来实现信息的隐藏，通过继承来实现功能的扩展等，同时泛型集合也提供了更加多样化的应用体验。

总结起来，集合的作用可以概括为：

- 是一个对象容器，并且提供了存取、检索、遍历等基本操作。
- 以统一的方式处理一组对象，简化对象管理操作。
- 实现多个返回值，当方法需要返回多个结果值时，集合提供了很好的实现方法。

2. 集合的通用方法和属性

集合类型，提供了操作集合元素的通用的方法和属性，这些方法是每个集合类型必须实现的基本功能，可以从集合类型的实现接口了解这一点。例如，每个集合类都必然实现了IEnumerable 接口，也就意味着任何集合类具有了GetEnumerator 方法，也使得以foreach 语句遍历集合对象成为可能。

下面，本节将小结集合类的通用方法和属性，为进一步阐述和了解集合特性打好坚实的基础。这些方法和属性是集合基本都具有的特性，了解这些方法有助于从通用角度理解集合特性：

- Add方法，向集合中添加新的元素。
- Insert方法，将元素插入集合的指定位置。
- Remove方法，从集合中移除指定元素。
- Clear方法，清除集合中的所有元素。
- IndexOf/LastIndexOf方法，在集合中进行线性搜索，如果找到第一个匹配的对象，则返回其索引。
- Sort方法，对集合元素进行排序。
- GetEnumerator方法，返回循环访问集合的枚举数。
- Count属性，获取集合的元素个数。
- Item属性，获取集合的指定元素。

当然，这些方法并非每个集合类型都完全具有，也并非完全包括诸多集合类型的所有方法，然而理解这些方法以达到举一反三的目的，省力又省心，对于集合尤为如此。

3. 集合类型的接口

从.NET的角度看集合，可以理解集合为实现了若干接口的类，因此关注这些接口对于理解集合，大有裨

益。以 `ArrayList` 为例，其定义可以表示为：

```
public class ArrayList : IList, ICollection, IEnumerable, ICloneable
```

`ArrayList` 实现了 `IList`、`ICollection`、`IEnumerable` 和 `ICloneable` 接口，我们接着对其一一梳理。

`IEnumerable` 接口的定义可表示为：

```
public interface IEnumerable
{
    IEnumerator GetEnumerator();
}
```

返回 `IEnumerator` 接口对象，以公开枚举数，支持非泛型集合的简单迭代，为 `foreach` 操作提供支持。在.NET 中，集合类必须严格实现 `IEnumerable` 接口才能与 `foreach` 兼容，`IEnumerator` 接口和 `foreach` 方法在后文有所交代。

`ICollection` 接口的定义表示为：

```
public interface ICollection : IEnumerable
{
    void CopyTo(Array array, int index);
    int Count { get; }
    bool IsSynchronized { get; }
    object SyncRoot { get; }
}
```

其中，`CopyTo` 表示将集合拷贝到数组；`Count` 属性则表示条目数量；`IsSynchronized` 和 `SyncRoot` 则用于提供线程同步支持。

`IList` 接口，提供了更广泛的接口约定，其定义可以表示为：

```
public interface IList : ICollection, IEnumerable
{
    int Add(object value);
    void Clear();
    bool Contains(object value);
    int IndexOf(object value);
    void Insert(int index, object value);
    void Remove(object value);
    void RemoveAt(int index);

    bool IsFixedSize { get; }
    bool IsReadOnly { get; }
    object this[int index] { get; set; }
}
```

其中，提供了添加、清空、插入、删除、查找多种操作支持，同时具有固定长度和只读属性，能够按照索引方式访问，为集合类提供了最基本的操作特性。同时 `IList` 接口还继承了 `ICollection` 和 `IEnumerable` 接口，在接口范围上更宽泛。

另外，以 Key-Value 对（键值对）访问方式的集合，还实现了 `IDictionary` 接口，同样提供了添加、清空、删除操作支持，具有固定长度和只读属性，能够以索引方式访问。不同的是其面向的都是操作 Key-Value 对类型的集合。

对于接口的应用，在下文的集合类部分将有更具体的体验和分析。

4. 集合的应用

在认识集合通用特性和实现接口的基础上，我们以 ArrayList 类型为例，来简要说明上述通用属性和方法的应用情况，可表示为：

```
class ArrayListEx
{
    public static void Main()
    {
        //定义集合类型
        ArrayList arrs = new ArrayList();
        //调用 Add 方法，增加集合元素
        arrs.Add(10);
        arrs.Add(new object());
        arrs.Add("I'm OK.");
        arrs.Add(new MyClass());
        arrs.Add(100);

        //调用 Remove 方法，移除指定对象
        arrs.Remove(100);

        //调用 Insert 方法，插入对象到指定索引位置
        arrs.Insert(2, "A insert value");

        //IndexOf 方法，用于从起始位置搜索，并返回第一个匹配项的索引位置
        Int32 index = arrs.IndexOf("I'm OK.");

        //获取集合的指定元素
        object obj = arrs[index];
        Console.WriteLine("{0} -- {1}", obj, index);

        //获取集合的元素个数
        Console.WriteLine(arrs.Count);

        //清除集合的所有元素
        arrs.Clear();
    }
}

class MyClass
{
    public override string ToString()
    {
        return "这是自定义类型。";
    }
}
```

(1) 枚举器

除了常用方法和属性的应用，需要特别注意的是，在.NET 中可以通过枚举器实现简单的循环访问集合：枚举器实现了 IEnumator 接口，而 IEnumator 接口的定义为：

```
public interface IEnumator
{
    bool MoveNext();
    object Current { get; }
    void Reset();
}
```

其中, MoveNext 将枚举数移到集合的下一成员; Current 表示集合的当前成员; Reset 方法将枚举数复位到集合开始, 并设置 Current 到第一个元素之前。因此, 通过枚举数遍历集合元素, 可以表示为:

```
//通过枚举器遍历集合元素
IEnumerator myEnumerator = arrs.GetEnumerator();
while (myEnumerator.MoveNext())
{
    Console.WriteLine(myEnumerator.Current);
}
```

同时, 某些高级语言提供了隐藏枚举复杂性的专门语句, 例如 C# 中的 foreach 语句、VB.NET 中的 For Each 语句都是用于简便的操作枚举器, 例如:

```
//以 foreach 语句来遍历集合
foreach(object o in arrs)
{
    Console.WriteLine(o.ToString());
}
```

然而, 以 ArrayList 集合来管理不同类型的对象, 会存在装箱与拆箱问题, 从而影响性能, 我们可以以泛型集合 List<T> 来代替 ArrayList 完成更灵活的集合操作体验, 并且避免性能的损失。

(2) 线程同步

在.NET 中, 大部分集合类型在默认情况下都不是线程安全的, 大部分集合类通过实现 Synchronized 静态方法来返回同步的集合包装, ArrayList 也不例外。其使用可举例为:

```
public static void Main()
{
    ArrayList al = new ArrayList();
    al.Add("Hello");
    al.Add("World");

    //生成一个线程同步的包装
    ArrayList syncAl = ArrayList.Synchronized(al);
}
```

本质上, Synchronized 方法将返回一个继承自 ArrayList 的私有 SyncArrayList 类, 该类通过在内部包装 ArrayList, 重写所有对集合进行更改的操作, 并且保证所有的操作是在锁定的前提下进行。因此 Synchronized 方法需要传入一个要包装的 ArrayList 对象作为参数, 而该参数正是其内部操作的 ArrayList 实例, 以其 Add 方法为例, 具体实现可以表示为:

```
public override int Add(object value)
{
    lock (this._root)
    {
        return this._list.Add(value);
    }
}
```

了解集合的通用规则和通用方法, 举一反三, 就可以轻便地深入到其他集合类规则, 尽管不同的集合类型设计为不同的用途, 然而掌握了基本内涵就很容易切入到不同的视角来理解其应用规则, 这也是有效学习的关键所在。

关于集合的规则, 这里主要着眼于集合的通用规则, 而不是某个集合类的规则, 主要包括:

- 大部分的集合类，都是接受 System.Object 类型元素的集合，在内部维护一个 System.Object 类型的操作列表，这样的好处是能够实现可接受任何类型的容器类，而由此带来的问题是在装箱时发生的性能损失。同时，也意味着集合的返回类型是 System.Object 类型，因此必须完成转换后才能发生有意义的操作。灵活而低效的特点，使得.NET 提供了一些强类型集合来解决性能问题，例如 CollectionBase、DictionaryBase、ReadOnlyCollectionBase 都是为响应的强类型提供抽象基类而用的；System.Collections.Specialized 命名空间中也定义了一些专门的强类型集合。
- 自定义集合应该选择合适的集合基类继承，而不应创建完全独立的自定义集合类。
- 不同的集合类都有不同的应用场合，因此有必要了解常见集合类的差异性，对其使用条件和优缺点建立一定的认知度。

8.9.3 各分秋色——.NET 集合类大观

在本节，我们近距离的接触.NET 类库中形形色色的集合类，通过实际的应用来领略.NET 集合类的方方面面。

1. 集合的分类

“横看成岭侧成峰，远近高低各不同”诠释的就是对同一事物，站在不同的角度就会有不同的理解。对于集合，同样如此，从不同的角度分类集合也会有不同的结果，例如从是否支持泛型的角度理解，集合可以分为泛型集合和非泛型集合。为了全面的诠释对集合类型的认识，在此我们分两个不同的角度来了解集合的分类。

按照集合类型实现的接口来分，集合主要分为：有序集合、索引集合和键式集合。有序集合，主要是指仅实现了 ICollection 接口的集合类，如 Stack 和 Queue；索引集合，主要是指实现了 IList 接口的集合类，如 Array、ArrayList 等；键式集合主要是指实现了 IDictionary 接口的集合类，如 Hashtable。

.NET 的集合类，集中在 System.Collection 和 System.Collections.Generic 命名空间中，从集合的访问方式来分，大体上包括两类：列表和字典。列表提供了顺序访问集合的方式，例如 Array、ArrayList、IList、Stack；字典则提供了 Key-Value 对（键值对）访问方式，主要有 Hashtable、ListDictionary 等。

在本节中，我们挑选几个集合中的活跃分子来进行剖析，以期达到熟悉集合类型的目的。

2. Array 和 ArrayList

数组（Array）是陪伴每个开发者进入程序设计领域的基础数据结构之一，作为最典型的集合类，数组具有集合类型应有的所有基本特性。ArrayList 可以看作是 Array 的复杂版本，我们在上文中已经对 ArrayList 有所了解。但是二者在规则上，具有明显的区别，因此有必要总结其异同，体验其应用。

相同点：

- Array 和 ArrayList 均实现了相同的接口，因此具有很多相同的操作方法，例如对自身进行枚举，能够以 foreach 语句遍历集合成员等。
- Array 和 ArrayList 创建的对象均存储在托管堆中。

不同点：

- Array 只能存储同构对象，不过声明为 Object 类型的数组除外，因为任何类型都可以隐式转换为 Object

类型; ArrayList 可以存储异构对象, 这是因为在本质上 ArrayList 内部维护着一个 object[] items 类型字段。在应用 ArrayList 时, 应该考虑装箱与拆箱带来的性能损失。因此, 一般情况下, Array 的性能要优于 ArrayList。

- Array 可以是一维的, 也可以是多维的; ArrayList 只能是一维的。
- Array 的容量是固定的, 一旦声明, 不可更改; ArrayList 的容量则是动态增加的。添加元素超过初始容量时, ArrayList 会根据需要重新分配, 还可以通过 TrimToSize 方法将其空项删除来压缩体积, 例如:

```
//说明 ArrayList 的容量是动态的
public static void Main()
{
    //初始化 arrs 容量为 2
    ArrayList arrs = new ArrayList(2);

    arrs.Add(1);
    arrs.Add(2);
    arrs.Add(3);

    Console.WriteLine("当前容量: {0}", arrs.Capacity);      //结果为 4

    arrs.TrimToSize();
    Console.WriteLine("压缩后的容量: {0}", arrs.Capacity); //结果为 3
}
```

注意: 事实上在.NET 中, 除了数组类型, 其他的集合类型的容量都具有动态增加的特性。

- Array 的下限可以设置; 而 ArrayList 的下限只能为 0。
- Array 只有简单的方法来获取或设置元素值, 不能随意增加或者删除数组元素; ArrayList 提供了更丰富的方法来完成对元素的操作, 可以方便的插入或者删除任意指定位置的元素。因此一般情况下, 可以用 ArrayList 来代替 Array。
- 对空数组元素进行操作, 可能会引起系统异常, 这是数组操作常见的问题之一, 例如:

```
public static void Main()
{
    User[] users = new User[10];
    users[0] = new User("小王");
    users[1] = new User("张三");

    //调用了空对象, 引发异常
    Console.WriteLine(users[5].Name);
}
```

因此, 对数组元素操作之前进行是否为 null 的判断, 是十分必要的。

3. Queue 和 Stack

Queue 和 Stack 都是有序集合类, 实现了 ICollection 接口, 对象按照其加入顺序从集合中检索, Queue 是先进先出, Stack 是后进先出。其应用可以小结为:

- Queue 和 Stack 的默认容量不同, Queue 为 32, 而 Stack 为 10。
- 关于 Queue 方法: Enqueue 用于向队尾添加元素, Dequeue 用于从队列返回并移除最开始的元素, Peek 用于从队列返回最开始的元素但不移除。
- 关于 Stack 方法: Push 用于在 Stack 顶部插入元素, Pop 用于返回并移除 Stack 顶部的元素, Peek 用于

返回 Stack 顶部的元素但不移除。

- 二者一般应用在临时存储数据的场合，或者对接收和处理顺序有要求的场合。随机访问集合元素不能使用这两个集合。

关于 Queue 和 Stack 的应用可以参考：

```
class Queue_Stack
{
    public static void Main()
    {
        Queue myQueue = new Queue();
        //向 Queue 尾部添加元素
        myQueue.Enqueue("我");
        myQueue.Enqueue("是");
        myQueue.Enqueue("谁");

        //返回 Queue 开始的元素
        Console.WriteLine(myQueue.Peek());
        //返回并移除 Queue 开始的元素
        myQueue.Dequeue();

        //遍历 Queue
        foreach (object o in myQueue)
            Console.WriteLine(o.ToString());

        Stack myStack = new Stack();
        //向 Stack 顶部插入元素
        myStack.Push("我");
        myStack.Push("是");
        myStack.Push("谁");

        //返回 Stack 顶部的元素
        Console.WriteLine(myStack.Peek());
        //返回并移除 Stack 顶部的元素
        myStack.Pop();

        //遍历 Stack
        foreach (object o in myStack)
            Console.WriteLine(o.ToString());
    }
}
```

4. Hashtable

Hashtable 是典型的字典类型，实现了 IDictionary 和 ICollection 接口，一个 Hashtable 集合元素是一个存储在 DictionaryEntry 对象中的键/值对，由两个对象构成：一个是键（Key），用于快速检索集合元素；一个是值（Value），用于存储用户数据。各个元素按照键值的哈希代码顺序存储，键值必须是唯一的，其不能为 null。其基本操作主要包括：

```
class HashtableEx
{
    public static void Main()
    {
        Hashtable ht = new Hashtable();

        //添加集合元素
        ht.Add("Name", "小王");
    }
}
```

```

ht.Add("Age", 27);
ht.Add("Degree", "硕士");

//以键值查找集合
Console.WriteLine("{0}的年龄是{1}", ht["Name"], ht["Age"]);

//集合遍历
foreach (DictionaryEntry de in ht)
{
    Console.WriteLine("{0}--{1}", de.Key.ToString(), de.Value.ToString());
}

//删除集合元素
ht.Remove("Age");

//集合排序
ArrayList als = new ArrayList(ht.Keys);
als.Sort();
foreach (string key in als)
{
    Console.WriteLine("{0}--{1}", key, ht[key].ToString());
}

//集合清空
ht.Clear();
}
}

```

5. 插曲：Hashtable 的顺序输出

对初学者而言，应用 Hashtable 常常期望能够实现“顺序地”按照 Key-Value 规则输出元素，然而实际的情况却事与愿违：

```

static void Main(string[] args)
{
    Hashtable ht = new Hashtable();
    ht.Add("Key1", "Abc");
    ht.Add("Key2", "Def");
    ht.Add("Key3", "Ghi");
    ht.Add("Key4", "Jkl");
    ht.Add("Key5", "Mno");
    foreach (string str in ht.Keys)
    {
        Console.WriteLine(str + "----" + ht[str]);
    }
}

```

按 F5 键运行上述代码逻辑，我们并没有发现按照“预期”的 Key 顺序输出元素。为了“刨根问底”，需要认识一下.NET 中 Hashtable 的 Add 机制，Hashtable 内部是根据 Key 的哈希码将元素加载到 buckets（存储桶）中，Key 的查找也是根据哈希码在特定的 buckets 中查找。所以改变 Hashtable 的初始容量，也会影响遍历输出的顺序正是这个道理。

从数据结构的角度来看，Hashtable 是不进行排序的，所以也就无所谓输出的顺序问题。如果必须进行必要的排序，.NET 中的其他集合例如 SortedList、SortedDictionary、ArrayList 等都可以作为参考目标。如果欲实现 Hashtable 按 Key 排序或者保持原始输入输出，是需要变通一下设计思路的。

正如在很多最佳实践的场合强调的一样，.NET Framework 框架类库是了解设计和本源的最佳参考资源，

有取之不尽、用之不竭的智慧闪耀其中。在.NET 集合类中，我们熟知的 SortedList 正是 Hashtable 和 Array 的混合体，研究 SortedList 可以带给我们很多的启示，例如 SortedList 内部就通过两个数组分别来维护 Key 和 Value，我们对 Hashtable 的改造正可以参考这种思路来完成。

所以可以借用一下 ArrayList 来辅助实现 Hashtable 的原样输出：

```
static void Main(string[] args)
{
    Hashtable ht = new Hashtable();
    ArrayList al = new ArrayList();
    ht.Add("Key1", "Abc");
    al.Add("Key1");
    ht.Add("Key2", "Def");
    al.Add("Key2");
    ht.Add("Key3", "Ghi");
    al.Add("Key3");
    ht.Add("Key4", "Jkl");
    al.Add("Key4");
    ht.Add("Key5", "Mno");
    al.Add("Key5");

    foreach (string str in al)
    {
        Console.WriteLine(str + "----" + ht[str]);
    }
}
```

有了合理的思路，就可以对上述略显生硬的实现进行重构，实现更灵活而优雅的实现，因此我们着手对 Hashtable 进行扩展，例如：

```
public class MyHashtable : Hashtable
{
    ArrayList al = new ArrayList();

    public override void Add(object key, object value)
    {
        Insert(key, value);
    }

    public override ICollection Keys
    {
        get
        {
            return al;
        }
    }

    public void Sort()
    {
        al.Sort();
    }

    private void Insert(object key, object value)
    {
        base.Add(key, value);
        al.Add(key);
    }
}
```

在上述实现中，不仅可以实现按原样输出（其实就是对上述思路的整合），同时还可以借助 ArrayList.Sort 实现 Hashtable 的按 Key 输出，测试如下：

```

static void Main()
{
    MyHashtable ht = new MyHashtable();
    ht.Add("Key1", "Abc");
    ht.Add("Key2", "Def");
    ht.Add("Key3", "Ghi");
    ht.Add("Key4", "Jkl");
    ht.Add("Key5", "Mno");

    foreach (string str in ht.Keys)
    {
        Console.WriteLine(str + "----" + ht[str]);
    }

    Console.WriteLine("Sort the hashtable by Keys...");

    //Hashtable 按 Key 排序
    MyHashtable ht2 = new MyHashtable();
    ht2.Add("Key5", "Abc");
    ht2.Add("Key2", "Def");
    ht2.Add("Key4", "Ghi");
    ht2.Add("Key1", "Jkl");
    ht2.Add("Key3", "Mno");

    //对 Hashtable 按 Key 排序
    ht2.Sort();
    foreach (string str in ht2.Keys)
    {
        Console.WriteLine(str + "----" + ht2[str]);
    }
}
}

```

当然，解决思路还有很多，我们也可以类似于 SortedList 通过维护一个 Array 来实现，不过从简单性考虑，维护 ArrayList 显然更具优势。

上述解决思路只是变相地解决了 Hashtable 的输出结果，而并未改变 Hashtable 的内部维护机制，如果以下述方式输出，其结果仍然并非理想输出：

```

static void Main()
{
    //Hashtable 按 Key 排序
    MyHashtable ht2 = new MyHashtable();
    ht2.Add("Key5", "Abc");
    ht2.Add("Key2", "Def");
    ht2.Add("Key4", "Ghi");
    ht2.Add("Key1", "Jkl");
    ht2.Add("Key3", "Mno");
    ht2.Sort();

    foreach (DictionaryEntry de in ht2)
    {
        Console.WriteLine(de.Key + "----" + de.Value);
    }
}

```

不过，我们已经“更进一步”地完善 Sorted Hashtable 了。

6. 其他集合类

集合类实在太多了，本节只能提供最典型集合的分析与比较，更多的集合类型应用，还是在实践中体会吧。在此，我们将其他一些重要集合类做一简要介绍：

- System.Collections.SortedList 集合实现了 IDictionary 和 ICollection 接口，是最基本的排序集合，兼有 ArrayList 和 Hashtable 的优点，能够按照关联键值对存储对象进行排序。另外，SortedList 集合还是唯一一个同时支持按照索引数字访问和按照键值访问的内建.NET 集合类。
- BitArray 是用于管理位值的压缩数组，集合的元素为位标志，一个元素一个位，而不是通常的对象元素。
- 在.NET 命名空间 System.Collections.Specialized 中，包含了一些特殊的集合类，例如：字符串集合类 StringCollection 和 StringDictionary 用于对字符串集合进行优化；位向量 BitVector32 专门处理内部使用的布尔值和小整数，性能较 BitArray 更好，因为 BitVector32 是值类型，而 BitArray 是引用类型；NameValueCollection 类则能包含关联同一键值的多个项，一个键值可以对应多个字符串值。

7. 泛型集合

泛型集合主要位于 System.Collections.Generic 和 System.Collections.ObjectModel 命名空间。其中，很多泛型集合都是对非泛型集合的模拟，因此在应用上具有类似的操作特性与规则，主要有：

- Dictionary< TKey, TValue > 泛型类对应于 Hashtable 类。
- List< T > 泛型类对应于 ArrayList 类。
- SortedList< T > 泛型类对应于 SortedList 类。
- Queue< T > 泛型类对应于 Queue 类。
- Stack< T > 泛型类对应于 Stack 类。
- Collection< T > 泛型类对应于 CollectionBase 类。
- ReadOnlyCollection< T > 泛型类对应于 ReadOnlyCollectionBase 类。

典型的泛型类型，当然是 List< T >，其定义可表示为：

```
public class List<T> : IList<T>, ICollection<T>, IEnumerable<T>, IList, ICollection, Ienumerable
```

List 类是 ArrayList 类的泛型等效类，因此也具有 ArrayList 的基本特性，例如集合元素可以动态容量、添加元素、插入元素和删除元素等基本方法。

而泛型集合类也实现了一些新特性来更好地操作集合对象，解决非泛型集合的某些不足。例如下例中 List< T > 泛型类可以通过 FindAll 方法找到指定条件的匹配元素，并以 ForEach 方法来实现对元素的指定操作，例如：

```
class ListT
{
    //List<T>的特性方法说明
    public static void Main()
    {
        List<Int32> nums = new List<int>();
        nums.Add(0);
        nums.Add(20);
        nums.Add(33);
```

```

    nums.Add(59);
    nums.Add(100);

    //按照指定条件查找匹配元素
    List<Int32> numTens = nums.FindAll(IsDivisionByTen);

    //通过ForEach方法对集合成员遍历操作
    numTens.ForEach(Console.WriteLine);
}

//指定的操作条件
private static bool IsDivisionByTen(int num)
{
    if (num % 10 == 0)
        return true;
    else
        return false;
}

```

泛型集合，提供了更多的新特性来完成操作，而且克服了装箱与拆箱的性能损失等不利因素，使得创建强类型集合变得容易，对于集合应用来说的确是不错的选择。关于泛型的相关内容，请参见第10章“接触泛型”的分析和讨论。

8.9.4 自我成全——实现自定义集合

正如本节看到的一样，.NET框架类库已经实现了丰富多彩的集合类，实现了各种各样灵活的集合操作，应用于不同的情况。然而有时候，我们还必须实现更加定制化的集合类来实现更有针对性、更灵活的对象管理，.NET框架也提供了相应的扩展机制来完成这一需求。关于自定义集合我们至少应该遵守以下的规则：

- 选择合适的基类继承，或者实现合适的接口。常用的ArrayList、Hashtable、Stack类型，都可以作为自定义集合的基类。
- 最好将自定义集合实现为强类型。如上文所言，大部分的集合类都接受System.Object作为其内部操作类型，由此引起不必要的性能损失。因此，.NET提供了CollectionBase抽象类作为强类型集合的基类，我们可以继承CollectionBase实现自定义的强类型集合，解决装箱拆箱等性能问题，也使自定义类型变得容易。
- 考虑在自定义集合中提供线程同步和序列化支持。

一般来说，实现自定义集合类有两种方法：一种是通过扩展现有预定义集合实现自定义集合，可以通过继承扩展基类方法，或者通过聚合封装预定义集合来实现集合操作方法；一种是实现全新的集合方法，可以实现相应ICollection、 IList等接口。显然，第一种方式更容易实现自定义集合，且能够轻便地实现相应功能，因此，我们以第一种方式，来实现基于用户自定义类型的集合类，并且继承.NET系统集合类CollectionBase为基类实现，具体的代码如下：

- (1) 首先定义集合操作对象的类型：

```

public class Student
{
    //学生姓名

```

```

private string name;

public string Name
{
    get { return name; }
    set { name = value; }
}

//学生年龄
private Int32 age;

public Int32 Age
{
    get { return age; }
    set { age = value; }
}

public Student(string name, Int32 age)
{
    this.name = name;
    this.age = age;
}
}

```

(2) 实现基于 CollectionBase 的自定义强类型集合:

```

[Serializable]
public class StudentCollection : CollectionBase
{
    //构造函数定义
    public StudentCollection()
    {
    }

    public StudentCollection(Student[] students)
    {
        this.AddRange(students);
    }

    //集合方法实现
    public int Add(Student student)
    {
        return base.List.Add(student);
    }

    public void AddRange(Student[] students)
    {
        if (students == null)
            throw new ArgumentNullException("Student is null.");

        for (Int32 i = 0; i < students.Length; i++)
        {
            this.Add(students[i]);
        }
    }

    public bool Contains(Student student)
}

```

```

{
    return base.List.Contains(student);
}

public void CopyTo(Student[] students, Int32 index)
{
    base.List.CopyTo(students, index);
}

public int IndexOf(Student student)
{
    return base.List.IndexOf(student);
}

public void Insert(Int32 index, Student student)
{
    base.List.Insert(index, student);
}

public void Remove(Student student)
{
    base.List.Remove(student);
}

//省略其他方法实现.....
}

//实现索引器
public Student this[int index]
{
    get { return (Student)base.List[index]; }
    set { base.List[index] = value; }
}
}

```

(3) 测试自定义集合:

```

class Test_StudentCollection
{
    public static void Main()
    {
        StudentCollection sc = new StudentCollection();
        sc.Add(new Student("小王", 27));
        sc.Add(new Student("小佳", 22));

        Student s = new Student("张三", 36);
        sc.Insert(1, s);
        sc.Remove(s);

        foreach(Student student in sc)
        {
            Console.WriteLine("Name:{0}\tAge:{1}", student.Name, student.Age);
        }
    }
}

```

应用扩展预定义集合类的方法，可以很方便地实现自定义集合类，这是我们推荐的办法，当然大部分时

候.NET 预定义集合类型已经能够满足我们的需求，但是自定义集合类的实现也为集合的操作带来更多的灵活性与可靠性。

8.9.5 结论

本节以集合概念为起点，研究其应用和意义；在此基础上深入了解.NET 的各个集合类，由分类而比较、经对比而应用；最后实现自定义集合类，将集合进行全方位的透视。

当然，集合通论并未将集合的方方面面一网打尽。关于线程同步、泛型集合还有其他集合类等，本节篇幅无力深入展开，仅提供了探索思考的起点，由读者自行挖掘。

参考文献

Patrick Smacchia. Pratical .NET2 and C#2

Ben Albahari , C# in a Nutshell

Jacquie Barker, Grant Palmer, Beginning C# Object Conceptions

Jeffrey Richter. Applied Microsoft .NET Framework Programming

Allen Lee, readonly vs. const [C#],

<http://www.cnblogs.com/allenlooplee/archive/2004/10/23/55183.html>

TerryLee. 再谈重载与覆写. <http://www.cnblogs.com/Terrylee/archive/2006/03/10/ 347104.html>

Marc Clifton, A KeyedList implementation.

<http://www.codeproject.com/KB/recipes/keyedlist.aspx>

失落的 BLOGS. C#泛型

<http://www.cnblogs.com/lianyonglove/archive/2007/07/27/720682.html>

天马行空. 深拷贝和浅拷贝. <http://www.cnblogs.com/Phoenix-Rock/archive/2006/11/07/ 552572.aspx>

第9章 本来面目——框架诠释

9.1 万物归宗: System.Object / 319	9.5.3 什么是 string / 345
9.1.1 引言 / 319	9.5.4 字符串创建 / 345
9.1.2 初识 / 319	9.5.5 字符串恒定性 / 346
9.1.3 分解 / 320	9.5.6 字符串驻留 (String Interning) / 346
9.1.4 插曲: 消失的成员 / 323	9.5.7 字符串操作典籍 / 350
9.1.5 意义 / 325	9.5.8 补充的礼物: StringBuilder / 352
9.1.6 结论 / 325	9.5.9 结论 / 354
9.2 规则而定: 对象判等 / 325	9.6 简易不简单: 认识枚举 / 354
9.2.1 引言 / 326	9.6.1 引言 / 355
9.2.2 本质分析 / 326	9.6.2 枚举类型解析 / 355
9.2.3 覆写 Equals 方法 / 329	9.6.3 枚举种种 / 358
9.2.4 与 GetHashCode 方法同步 / 331	9.6.4 位枚举 / 360
9.2.5 规则 / 332	9.6.5 规则与意义 / 361
9.2.6 结论 / 332	9.6.6 结论 / 361
9.3 疑而不惑: interface “继承” 争议 / 332	9.7 一脉相承: 委托、匿名方法和 Lambda 表达式 / 362
9.3.1 引言 / 332	9.7.1 引言 / 362
9.3.2 从面向对象寻找答案 / 333	9.7.2 解密委托 / 362
9.3.3 以 IL 探求究竟 / 334	9.7.3 委托和事件 / 365
9.3.4 System.Object 真是	9.7.4 匿名方法 / 367
9.3.4 “万物之宗” 吗 / 334	9.7.5 Lambda 表达式 / 368
9.3.5 接口的继承争议 / 335	9.7.6 规则 / 368
9.3.6 结论 / 335	9.7.7 结论 / 369
9.4 给力细节: 深入类型构造器 / 336	9.8 Name 这回事儿 / 369
9.4.1 引言: 一个故事 / 336	9.8.1 引言 / 369
9.4.2 认识对象构造器和类型构造器 / 337	9.8.2 畅聊 Name / 369
9.4.3 深入执行过程 / 339	9.8.3 回到问题 / 371
9.4.4 回归故事 / 341	9.8.4 结论 / 371
9.4.5 结论 / 342	9.9 直面异常 / 371
9.5 如此特殊: 大话 String / 342	9.9.1 引言 / 372
9.5.1 引言 / 342	9.9.2 为何而抛 / 372
9.5.2 问题迷局 / 343	9.9.3 从 try/catch/finally 说起: 解析异常机制 / 373
	9.9.4 .NET 系统异常类 / 377
	9.9.5 定义自己的异常类 / 379
	9.9.6 异常法则 / 381
	9.9.7 结论 / 382
	参考文献 / 382



9.1 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
- 对 Object 成员应用 EditorBrowsable 特性

9.1.1 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，我们就和这些公共特性一一过招，了解其作用和意义，深入其功能和应用。

9.1.2 初识

有了对 Object 类型的初步认识，下面我们使用 Reflector 工具加载 mscorelib 程序集来反编译 System.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将在后文表述。

9.1.3 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，关于全名称详见 9.8 节“Name 这回事儿”的论述。在通常的情况下，ToString 方法提供了在子类中重新覆盖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 方法的覆盖，例如 Boolean 类型通过覆盖 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，从而可以提供所属类型的很多信息：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

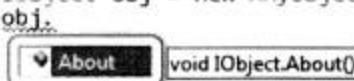
- Equals 静态方法、虚方法和 ReferenceEquals 方法用于对象判等，详细的应用请参考 9.2 节“规则而定：对象判等”。
- GetHashCode 方法，用于在类型中提供哈希值，以应用于哈希算法或哈希表，不过值得注意的是对 Equals 方法和 GetHashCode 方法的覆写要保持统一，因为两个对象的值相等，其哈希码也应该相等，否则仅覆写 Equals 而不改变 GetHashCode，会导致编译器抛出警告信息。
- Memberwise 方法，用于在对象克隆时实现对象的浅拷贝，详细应用请参考 8.7 节“有深有浅的克隆：浅拷贝和深拷贝”。
- Finalize 方法，用于在垃圾回收时实现资源清理，详细应用请参考 6.3 节“垃圾回收”。

9.1.4 插曲：消失的成员

关于 object 的故事，有很多很多。在.NET 世界里，object 是公认的造物主，其麾下的 7 大成员，个顶个的横行在任何系统的任何代码角落。而接下来的故事则着眼于“为熟悉的朋友做点儿不熟悉的事儿”。

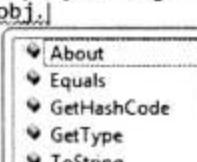
相信吗？你的 object 成员不见了，不信你可以欣赏一下消失了的 object 成员。

```
class Program
{
    static void Main(string[] args)
    {
        IObject obj = new AnyObject();
        obj.
    }
}
```



哈哈！清晰多了吧，比起下面常见的编码方式：

```
class Program
{
    static void Main(string[] args)
    {
        AnyObject obj = new AnyObject();
        obj.
    }
}
```



是不是让人不知所措。大概说来，任何时候，在长长的成员方法列表中，你总能看到它们的身影：Equals、GetHashCode、GetType 和 ToString，谁让 object 是万物的基类呢？不过，有些时候，你可能希望眼根清净，屏蔽掉不会使用的父类成员，使得方法调用变得更加简洁，就像上面的 IObject 成员一样。

那么这一切是如何做到的呢？其实，这是一次赤裸裸的欺骗，而行骗的家伙就是将要闪亮登场的：

```
namespace System.ComponentModel
{
    [AttributeUsage(AttributeTargets.Class | AttributeTargets.Struct | AttributeTargets.Enum | AttributeTargets.Constructor | AttributeTargets.Method | AttributeTargets.Property | AttributeTargets.Field | AttributeTargets.Event | AttributeTargets.Interface | AttributeTargets.Delegate)]
    public sealed class EditorBrowsableAttribute : Attribute
    {
    }
}
```

是的，正是 System.ComponentModel.EditorBrowsableAttribute。以上例而言，其实为 AnyObject 类实现了下面的代码：

```
public interface IObject : IAnyObject
{
    void About();
}

public class AnyObject : IObject
{
    public void About()
    {
    }
}
```

其中的核心在于 IAnyObject 的定义：

```
namespace Anytao.Core.Common
{
    /// <summary>
    /// A common interface for any object
    /// </summary>
    [EditorBrowsable(EditorBrowsableState.Never)]
    public interface IAnyObject
    {
        [EditorBrowsable(EditorBrowsableState.Never)]
        bool Equals(object obj);

        [EditorBrowsable(EditorBrowsableState.Never)]
        int GetHashCode();

        [EditorBrowsable(EditorBrowsableState.Never)]
        Type GetType();

        [EditorBrowsable(EditorBrowsableState.Never)]
        string ToString();
    }
}
```

通常情况下，在基础组件中可以提供一个通用的 IAnyObject 接口，该接口的作用就是将 object 成员魔术般地隐藏掉，就如同本文开始的 IObject 一样。

回到 System.ComponentModel.EditorBrowsableAttribute 特性，就可以了解到其作用就是：标识一个类或者属性在编辑器 Visual Studio 中的可见性。那么，难道它们真的消失了吗？继续应用一开始的代码：

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

```

    {
        IObject obj = new AnyObject();
        Console.WriteLine(obj.ToString());
    }
}

```

我们发现虽然 `ToString` 对 `obj` 是不可见的，但是运行时调用仍然没有问题，所以，总体说来 `System.ComponentModel.EditorBrowsableAttribute` 只是一个障眼法，在此实现了对 Visual Studio 智能感知的控制。其中 `EditorBrowsableState` 选项主要包括了：

- Advanced，针对高级成员的选项设置，同样的方式可以应用 Visual Studio 的 Options->Text Editor->C#->General 的 Hide advanced members 设置。
- Always，总是可见。
- Never，总是不可见。

注意：在 Visual C# 中，`EditorBrowsableAttribute` 并不对同一程序集的成员有效。还等什么？也去试试吧！关于编辑器，还有很多好玩的特性值得挖掘，贵在发现的力量。而这种小把戏，也为 Object 成员在实际开发中的控制增加了一些色彩。

9.1.5 意义

- 实现自上而下的单根继承。
- `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。
- 合理应用编译器的某些特性，可以实现更好的编程体验。

9.1.6 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。

9.2 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

9.2.1 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：

- 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。
- 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。

9.2.2 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 false, 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 5.4 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 9.5 节“为什么特殊: 大话 String”, 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”。

9.2.3 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，`Equals` 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(objB.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。
- 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。

9.2.4 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。

9.2.5 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`IComparable`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。

9.2.6 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。

9.3 疑而不惑：interface“继承”争议

本节将介绍以下内容：

- 继承的细节
- interface 继承争议
- 编译选项`noautoinherit`

9.3.1 引言

在.NET世界里，常常听到的一句话就是“`System.Object`是一切类型的根，是所有类型的父类”，以至于在9.1节以“万物归宗：`System.Object`”这样拉风的题目为`System.Object`授予至高无上的荣誉。

所以，基于这样的观点就有了下面这句自然而然的推理：接口到底继承于`System.Object`吗？

是，或者否。就此展开讨论。

是：持“interface也继承于object”意见，是基于以下的两个观点推断的。

观点一：

接口本质上也是一个`class`，因为接口类型编译之后在IL中被标识为`.class`，既然是类那么不可避免地最终继承于`System.Object`。

观点二：

假如有如下的接口和类型：

```
public interface IObjectable
{
}

public class MyObject : IObjectable
{
}
```

那么，对于 IObjectable 对象而言，下面的调用是可行的：

```
class Program
{
    static void Main(string[] args)
    {
        IObjectable obj = new MyObject();

        //Call Object instance methods
        obj.ToString();
        //Call Object static methods
        IObjectable.Equals(null, null);
    }
}
```

显然，IObjectable 类型变量 obj 可以访问存在于 System.Object 中的实例方法 ToString() 和虚方法 Equals，当然其他的几个公共服务也不例外：GetType()、Equals()、GetHashCode()、ReferenceEquals()，也可以由此找到 interface 可访问 object 方法的蛛丝马迹。

不可否认，以上观点的部分推理是完全正确的，但是却并非最可靠的结论，而对于此话题的明确答案是：interface 不“继承”于 object。接下来，我们就对其原因和原理展开深入讨论，而关于接口本质话题的深度讨论，请参考 1.5 节“玩转接口”和 8.4 节“面向抽象编程：接口和抽象类”的详细分析。

！ 注意

事实上，在.NET 中对于接口，更强调忽略“继承”这一概念，接口是“被实现”，而非“被继承”。

9.3.2 从面向对象寻找答案

接口，就像面向对象设计中的精灵，为面向对象思想注入了灵魂和活力，接口突破了继承只在纵向的扩展方向，在横向给予对象更灵活的支持，在 1.5 节“玩转接口”中已经有了较完整的讨论。

接口，封装了对于行为的抽象，定义了实现者必须遵守的契约。例如，实现了 System.ICloneable 接口的类型被赋予了“可以被复制”这样的契约，实现了 System.Collections.IEnumerable 接口的类型被赋予了“可以被枚举”这样的契约，不同的接口定义了不同的契约，就像不同的法律约束了不同的行为。那么，接口应该赋予的契约至少在层次上保持相对的单纯和统一，如果为所有接口都无一例外地赋予 GetType()、Equals()、GetHashCode()、ReferenceEquals() 还有 ToString() 这样的契约，未免使得接口的纯洁和统一变得无从谈起，例

如强迫任何实现了 System.ICloneable 接口的类型同时遵守其他的约定也是对 ICloneable 本身的侮辱。

从接口单一原则延伸思考，一个包含杂七杂八的接口定义显然不是 interface 应该具有的纯正血统，对于深谙面向对象为何物的.NET 设计者而言，这是不言而喻的问题。所以，从接口本身的职业和意义出发，决断 interface 不从 System.Object 继承是合理的推论。

9.3.3 以 IL 探求究竟

再次应用强大的 IL 武器来探求事实的真相，以 Reflector 打开任何的.NET 既有接口，例如 IList、IEnumerable、ICollection，都会有个共同的发现，那就是你找不到 extends System.Object 这样的标识：

```
.class public interface abstract auto ansi ICloneable
{
    .custom instance void System.Runtime.InteropServices.ComVisibleAttribute::ctor(bool)
= { bool(true) }
    .method public hidebysig newslot abstract virtual instance object Clone() cil managed
    {
    }
}
```

自定义类型也是如此，我们看看 IObjectable 的 IL 反编译定义：

```
.class public interface abstract auto ansi IObjectable
{}
```

而以 extends 标识继承关系是 IL 代码告诉真相的最佳证明。

9.3.4 System.Object 真是“万物之宗”吗

在 9.1 节“万物归宗：System.Object”强调了 System.Object 是一切类型的根。而本节的论述，让这一法则不得不存在例外。其实，这一例外不仅存在于 interface 的情况，如果再次回眸一笑，把 Object 继续把玩，就会发现更多的例外：难道一切类型都得继承自 Object 吗？其实不然。ILASM.exe 进行 IL 代码编译时，有一个参数选项 NOAUTOINHERIT，正如其解释所描述的那样：

```
/NOAUTOINHERIT Disable inheriting from System.Object by default
```

显然 NoAutoInherit 选项提供了为.NET 类型“去掉帽子”的作用，简单言之就是，在未指定基类时，禁止类型自动从 Object 继承。

继续玩一个翻来覆去的 IL 游戏，将本文开始的控制台程序以 ILDASM.exe 工具 Dump 为 IL 代码 My.il，例如 MyObject 被反编译为：

```
.class public auto ansi beforefieldinit Anytao.Insidenet.InterfaceInside.MyObject
    extends [mscorlib]System.Object
    implements Anytao.Insidenet.InterfaceInside.IObjectable
{
    .method public hidebysig specialname rtspecialname
        instance void .ctor() cil managed
    {
        // Code size      7 (0x7)
```

```

.maxstack 8
IL_0000: ldarg.0
IL_0001: call     instance void [mscorlib]System.Object::..ctor()
IL_0006: ret
} // end of method MyObject::..ctor

} // end of class Anytao.Insidenet.InterfaceInside.MyObject

```

接下来，选择删除其中所有 `extends` 继承的代码，再以 `ILASM.exe` 对其进行 `noautoinherit` 编译，并生成：

```
ilasm /exe /output:noobject.exe /noautoinherit my.il
```

新生成的 `noobject.exe` 程序将没有从 `Object` 继承，某种程度上打破了“万物归宗”的传奇，`MyObject` 就像一个无根之木，飘摇在某个进程的深处。

9.3.5 接口的继承争议

那么，该以什么样的称呼来描述接口和类这档子关系呢？很多时候，对接口而言，“继承”二字代表了一种混淆的描述，官方更倾向通过“实现”来定义二者的关系。



插曲：Java 中的 `extends` 和 `implements`

事实上，在 Java 语言中，对于 `extends` 和 `implements` 有着非常明确而清晰的界定：以 `extends` 表示类对类的继承；而以 `implements` 表示类对接口的实现。而 C# 统一通过“`:`”来定义这两种关系，给理解接口“实现”带来了一定的困扰和误解，不过也换来语法表达上的简洁。

类实现了接口：

```
// MyObject 实现了 IObjectable
public class MyObject : IObjectable
{
}
```

类继承了类：

```
// MyObjectA 继承自 MyObject
public class MyObjectA : MyObject, IObjectable
{}
```

在广义上，接口是规范和契约，不可直接被实例化，所以“万物归宗”之物更代表实现了接口的实际类型。从继承的角度而言，继承体现在可实例化类上，在接口出现的场合，继承表现在为实现了接口的类赋予了接口规范定义的契约，而实现了接口的类，才是实实在在的继承。从这个意义而来，对于接口继承的命题，本身就是一个“伪”命题。

9.3.6 结论

如果 interface 不从 object 继承，那么，该如何回答本文开始对此质疑的两种观点呢？

回答观点一：

接口本质上还是一个类，但是一个特殊的类，它的特殊性表现在诸多方面，例如所有的方法和属性都是

抽象的、支持多继承等，而另一个特殊就是，不继承于任何的父类。

虽然这种解释略显牵强，但是如前文回到接口本源的角度而言，却是最好的解释。

回答观点二：

.NET一切类型都隐式继承于 System.Object，那么对于实现了任何接口的类型而言，例如：

```
public class MyObject : IObjectable
{
}
```

其本质上相当于：

```
public class MyObject : Object, IObjectable
{
}
```

所以对于 MyObject 实例 obj 而言，obj.ToString()实质是 MyObject 类继承于 object，并不代表接口 IObjectable 也继承于 object。那么 IObjectable.Equals()则是编译器做了手脚，将 IObjectable.Equals()翻译为 Object.Equals()所致。事实上，对于接口声明类型的方法调用，在实现机制上完全不同于一般的直接方法调用和虚方法分派机制，而这是另外一个重要的话题。

interface，想说爱你不容易，可能还会再次相遇。

9.4 给力细节：深入类型构造器

本节将介绍以下内容：

- 认识类型构造器和对象构造器
- precise 方式和 beforefieldinit 方式对比

9.4.1 引言：一个故事

类型初始化引发的问题，引起很多的关注与疑问，围绕 Type Initializer 和 BeforeFieldInit 而展开很多唇枪舌战的火拼。曾经在论坛的战火，点燃了本节关于类型构造器的探讨，尤其是 precise 方式和 beforefieldinit 方式在类型初始化时的差异性。

首先定义一个演示的类 Foo：

```
class Foo
{
    public static string Field = GetString("Initialize the static field!");
    public static string GetString(string s)
    {
        Console.WriteLine(s);
        return s;
    }
}
```

然后执行如下的逻辑：

```
static void Main()
{
```

```

Console.WriteLine("Start ...");
Foo.GetString("Manually invoke the static GetString() method!");
}

```

结果将如期输出：

```

Start ...
Initialize the static field!
Manually invoke the static GetString() method!

```

如果将执行逻辑修改为：

```

static void Main()
{
    Console.WriteLine("Start ...");
    Foo.GetString("Manually invoke the static GetString() method!");

    string field = Foo.Field;
}

```

而结果发生改变：

```

Initialize the static field!
Start ...
Manually invoke the static GetString() method!

```

继续讲述曲折的故事，如果为 Foo 类型添加静态构造函数：

```

class Foo
{
    // ...
    static Foo() { }
}

```

则输出结果又被改变为：

```

Start ...
Initialize the static field!
Manually invoke the static GetString() method!

```

曲折的故事，必然有曲折的内容，本文从基础开始层层深入，更全面地认识类型构造器和 BeforeFieldInit，并在此基础上，探讨解开故事内在的来龙去脉。

！注意

本故事源于和朋友蒋金楠的讨论，感谢他的慷慨共享。

9.4.2 认识对象构造器和类型构造器

在.NET 中，一个类的初始化过程是在构造器中进行的。根据构造成员的类型，分为类型构造器 (.cctor) 和对象构造器 (.ctor)，其中.cctor 和.ctor 为二者在 IL 代码中的指令表示。.cctor 不能被直接调用，其调用规则正是本文欲加阐述的重点，详见后文的分析；而.ctor 会在类型实例化时被自动调用。

基于对类型构造器的探讨，有必要首先实现一个简单的类定义，其中包括普通的构造器和静态构造器，例如：

```

public class User
{

```

```

static User()
{
    message = "Initialize in static constructor.";
}

public User()
{
    message = "Initialize in normal construcotr.";
}

public User(string name, int age)
{
    Name = name;
    Age = age;
}

public string Name { get; set; }
public int Age { get; set; }
public static string message = "Initialize when defined.";
}

```

我们将上述代码使用 ILDasm.exe 工具反编译为 IL 代码，可以很方便地找到相应的类型构造器和对象构造器的影子，如图 9-1 所示。

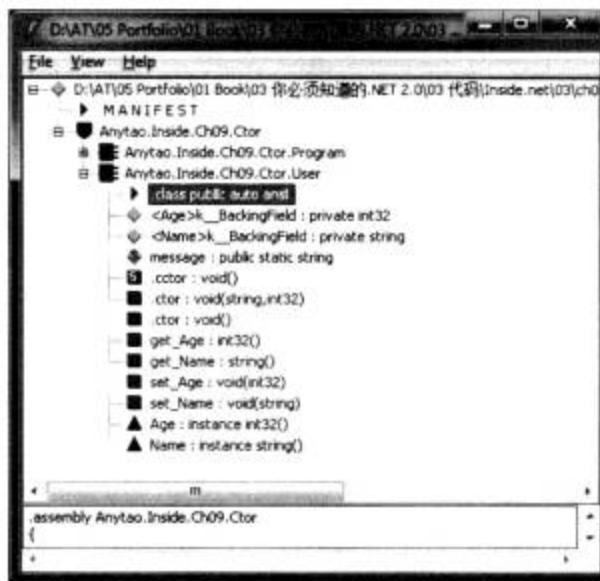


图 9-1 类型构造器和对象构造器

下面简单来了解一下对象构造器和类型构造器的概念。

1. 对象构造器 (.ctor)

在生成的 IL 代码中将可以看到对应的.ctor，类型实例化时会执行对应的构造器进行类型初始化的操作。关于实例化的过程，涉及比较复杂的执行顺序，按照类型基础层次进行初始化的过程可以参阅 8.8 节“动静之间：静态和非静态”一文中有详细的介绍和分析，因此不做过多探讨。

本文的重点以考查类型构造器为主，在此对.ctor 不进行过多的探讨。

2. 类型构造器 (.cctor)

用于执行对静态成员的初始化，在.NET 中，类型在两种情况下会发生对.cctor 的调用：

- 为静态成员指定初始值，例如上例中只有静态成员初始化，而没有静态构造函数时，.cctor 的 IL 代码实现为：

```
.method private hidebysig specialname rtspecialname static
    void .cctor() cil managed
{
    // Code size      11 (0xb)
    .maxstack 8
    IL_0000: ldstr     "Initialize when defined."
    IL_0005: stsfld    string Anytao.Write.TypeInit.User::message
    IL_000a: ret
} // end of method User::cctor
```

- 实现显式的静态构造函数，例如上例中有静态构造函数存在时，将首先执行静态成员的初始化过程，再执行静态构造函数的初始化过程，.cctor 的 IL 代码实现为：

```
.method private hidebysig specialname rtspecialname static
    void .cctor() cil managed
{
    // Code size      23 (0x17)
    .maxstack 8
    IL_0000: ldstr     "Initialize when defined."
    IL_0005: stsfld    string Anytao.Write.TypeInit.User::message
    IL_000a: nop
    IL_000b: ldstr     "Initialize in static constructor."
    IL_0010: stsfld    string Anytao.Write.TypeInit.User::message
    IL_0015: nop
    IL_0016: ret
} // end of method User::cctor
```

同时，必须明确一些静态构造函数的基本规则，主要包括：

- 静态构造函数必须是静态且无参的，并且一个类只能有一个。
- 只能对静态成员进行初始化。
- 静态无参构造函数可以和非静态无参构造函数共存，区别在于二者的执行时间，详见 8.8 节“动静之间：静态和非静态”的论述，其他更多的区别和差异也详见该节的描述。

9.4.3 深入执行过程

因为类型构造器本身的特点，在一定程度上决定了.cctor 的调用时机并非是一个确定的概念。因为类型构造器都是 private 的，用户不能显式调用类型构造器。所以关于类型构造器的执行时机问题在.NET 中主要包括两种方式：

- precise 方式
- beforefieldinit 方式

二者的执行差别主要体现在是否为类型实现了显式的静态构造函数，如果实现了显式的静态构造函数，则按照 precise 方式执行；如果没有实现显式的静态构造函数，则按照 beforefieldinit 方式执行。

为了说清楚类型构造器的执行情况，首先在概念上必须明确一个前提，那就是 precise 的语义明确了.cctor 的调用和调用存取静态成员的时机存在精确的关系，换句话说，类型构造器的执行时机在语义上决定于是否显式地声明了静态构造函数，以及存取静态成员的时机，这两个因素。

以例而理，还是从 User 类的实现说起，一一过招分析这两种方式的执行过程。

1. precise 方式

首先实现显式的静态构造函数方案：

```
// Author : Anytao, http://www.anytao.com
public class User
{
    //Explicit Constructor
    static User()
    {
        message = "Initialize in static constructor.";
    }

    public static string message = "Initialize when defined.";
}
```

对应的 IL 代码为：

```
.class public auto ansi User
    extends [mscorlib]System.Object
{
    .method private hidebysig specialname rtspecialname static void .cctor() cil managed
    {
        .maxstack 8
        L_0000: ldstr "Initialize when defined."
        L_0005: stsfld string Anytao.Write.TypeInit.User::message
        L_000a: nop
        L_000b: ldstr "Initialize in static constructor."
        L_0010: stsfld string Anytao.Write.TypeInit.User::message
        L_0015: nop
        L_0016: ret
    }

    .method public hidebysig specialname rtspecialname instance void .ctor() cil managed
    {
        .maxstack 8
        L_0000: ldarg.0
        L_0001: call instance void [mscorlib]System.Object:::.ctor()
        L_0006: ret
    }

    .field public static string message
}
```

为了进行对比分析，需要同时分析 beforefieldinit 方式的执行情况。

2. beforefieldinit 方式

为 User 类型不实现显式的静态构造函数方案：

```
public class User
{
    //Implicit Constructor
    public static string message = "Initialize when defined.";
}
```

对应的 IL 代码为：

```
.class public auto ansi beforefieldinit User
    extends [mscorlib]System.Object
{
    .method private hidebysig specialname rtspecialname static void .cctor() cil managed
```

```

    {
        .maxstack 8
        L_0000: ldstr "Initialize when defined."
        L_0005: stsfld string Anytao.Write.TypeInit.User::message
        L_000a: ret
    }

    .method public hidebysig specialname rtspecialname instance void .ctor() cil managed
    {
        .maxstack 8
        L_0000: ldarg.0
        L_0001: call instance void [mscorlib]System.Object::..ctor()
        L_0006: ret
    }

    .field public static string message
}

```

3. 分析差别

从 IL 代码的执行过程而言，可以了解的是在显式和隐式实现类型构造函数的内部，除了添加新的初始化操作之外，二者的实现是基本相同的。所以要找出两种方式的差别，最终将着眼点锁定在二者元数据的声明上，隐式方式多了一个称为 `beforefieldinit` 标记的指令。

那么，`beforefieldinit` 究竟表示什么样的语义呢？Scott Allen 对此进行过详细的解释：`beforefieldinit` 为 CLR 提供了在任何时候执行 `cctor` 的授权，只要该方法在第一次访问类型的静态字段之前执行即可。

所以，如果对 `precise` 方式和 `beforefieldinit` 方式进行比较，二者的差别就在于是否在元数据声明时标记了 `beforefieldinit` 指令。在 `precise` 方式下，CLR 必须在第一次访问该类型的静态成员或者实例成员之前执行类型构造器，也就是说必须刚好在存取静态成员或者创建实例成员之前完成类型构造器的调用；在 `beforefieldinit` 方式下，CLR 可以在任何时候执行类型构造器，一定程度上实现了对执行性能的优化，因此较 `precise` 方式更加高效。

值得注意的是，当有多个 `beforefieldinit` 构造器存在时，CLR 无法保证这多个构造器之间的执行顺序，因此在实际的编码时应该尽量避免这种情况的发生。

9.4.4 回归故事

回归开头的故事，结合上文的分析便可给出一些值得参考的答案：

- 可以很容易地确定对于显式实现了静态构造函数的情况，类型构造器的调用在刚好引用静态成员之前发生，所以不管是否在 `Main` 中声明：

```
string field = Foo.Field;
```

执行的结果不受影响。

- 在没有显式实现静态构造函数的情况下，`beforefieldinit` 优化了类型构造器的执行不在确定的时间进行，只要是在静态成员引用或者类型实例发生之前即可，所以在 Debug 环境下调用的时机变得不按常理。然而在 Release 优化模式下，`beforefieldinit` 的执行顺序并不受

```
string field = Foo.Field;
```

的影响，完全符合 `beforefieldinit` 优化执行的语义定义。

- 关于最后一个静态成员继承情况的结果，正像本文描述的逻辑一样，类型构造器是在静态成员被调用或者创建实例时发生，所以示例的结果是完全遵守规范的。不过，并不建议子类不要调用父类静态成员，原因是对于继承机制而言，子承父业是继承的基本规范，除了强制为 `private` 之外，所有的成员或者方法都应在子类中可见。而对于存在的潜在问题，以规范来约束可能会更好。其中，静态方法一定程度上是一种结构化的实现机制，在面向对象的继承关系中，本质上就存在一定的不足。
- 在 C# 规范中，关于 `beforefieldinit` 的控制已经引起很多的关注和非议，一方面 `beforefieldinit` 方式可以有效地优化调用性能，但是以显式或者隐式实现静态构造函数的方式不能更加直观地让程序开发者来控制，因此在以后版本的 C# 中，能实现基于特性的声明方式来控制，是值得期待的。
- 另一方面，在有两个类型的类型构造器相互引用的情况下，CLR 无法保证类型构造器的调用顺序，对程序开发者而言，同样对于类型构造器而言，应该尽量避免要求顺序相关的业务逻辑，因为很多时候执行的顺序并非声明的顺序，这是值得关注的。

9.4.5 结论

除了解决了一个问题，本文算是继续了关于类型构造器在 8.8 节“动静之间：静态和非静态”中的探讨，以更全面的视角来进一步阐释这个问题。在最后，关于 `beforefieldinit` 标记引起的类型构造器调用优化的问题，虽然没有完全了解在 Debug 模式下的 CLR 调用行为，但是深入细节可以掌控对于语言之内更多的理解，从这点而言，它也算是开了一个好头。

9.5 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

9.5.1 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。

9.5.2 问题迷局

有了如此特殊的身份，想要深入其特殊，就并非易事。所以，带着问题思考，是深入探索的不二法门，本文也无一例外地从几个测试开始，希望读者能沿着这几个简单的示例来思考。如果对此饱含热情，不妨可以试试给出答案。

```
static void Main()
{
    string s1 = "abc";
    Console.WriteLine(string.IsInterned(s1) ?? "null");
}
```

这是个简单的例题，可以很快给出答案。

```
static void Main()
{
    string s1 = "ab";
    s1 += "c";
    Console.WriteLine(string.IsInterned(s1) ?? "null");
}
```

稍加修改，这回的答案又该如何分析呢？

```
static void Main()
{
    string s1 = "abc";
    string s2 = "ab";
    s2 += "c";

    string s3 = "ab";

    Console.WriteLine(string.IsInterned(s1) ?? "null");
    Console.WriteLine(string.IsInterned(s2) ?? "null");

    Console.WriteLine(string.IsInterned(s3) ?? "null");
}
```

如果上述执行过程你能很快给出答案，那么恭喜了，第一关看来不是那么费劲。接着思考，继续第二关：

```
static void Main()
{
    string s1 = "abc";
    string s2 = "ab";
    string s3 = s2 + "c";

    Console.WriteLine(string.IsInterned(s3) ?? "null");
}
```

还有一个：

```
static void Main()
{
    string s2 = "ab";
```

```
s2 += "c";  
  
Console.WriteLine(string.IsInterned(s2) ?? "null");  
  
string s1 = "abc";  
}
```

你的答案将会是什么呢？还是接着迎接挑战吧：

```
static void Main()  
{  
    string s2 = "ab";  
    s2 += "c";  
  
    Console.WriteLine(string.IsInterned(s2) ?? "null");  
  
    string s1 = GetStr();  
}  
  
private static string GetStr()  
{  
    return "abc";  
}
```

这是第二关了，你的思考肯定还在继续，第三关也呼之欲出：

```
public const string s1 = "abc";  
  
static void Main()  
{  
    string s2 = "ab";  
    s2 += "c";  
  
    Console.WriteLine(string.IsInterned(s2) ?? "null");  
}
```

最后一个，冲出藩篱：

```
public static string s1 = "abc";  
  
static void Main()  
{  
    string s2 = "ab";  
    s2 += "c";  
  
    Console.WriteLine(string.IsInterned(s2) ?? "null");  
}
```

过关斩将，三轮PK。不管怎样，对于答案和思考，肯定会对string刮目相看，是否和你一直以来的认识统一呢？有了问题，就可以沿着问题，趁机理清几个相关的概念：

- 什么是string？
- 什么是字符串驻留？
- 字符串驻留的运行机制及执行过程？
- 方法调用的执行过程？

或者给问题以答案，或者给答案以问题，你可能永远无法看清全部，但是总能从一点突破很多。事实的关键就在于面对问题，该如何思考？

9.5.3 什么是 string

什么是 string 呢？本质上，string 就是一连串的有顺序的字符集合。

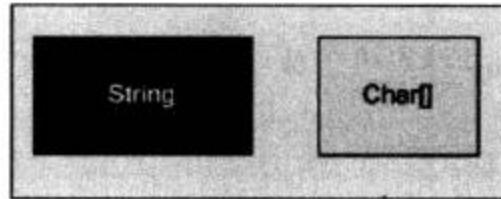


图 9-2 string 和 char 数组

简单地说，string 就是 char[]，如图 9-2 所示。而在.NET 中 string “被” 赋予了类的概念，暗合了.NET 一切皆为对象的大一统格局。回归本质，重新审视如此另类而多彩的 string，你会不禁明白，string 本质上就是一个 16 位 Unicode 字符数组。打开 string 的 Disassemble 代码，我们可直击其本质：

```
[Serializable, ComVisible(true)]
public sealed class String : IComparable, ICloneable, IConvertible, IComparable<string>,
IEnumerable<char>, IEnumerable, IEquatable<string>
{
}
```

结合 string 的定义，可以看出其基本的特性主要包括：

- 引用类型，string 本质上是引用类型，相关内容参考 5.2 节“品味类型——值类型与引用类型”中对值类型和引用类型的讨论。
- 字符串恒等性。
- 字符串驻留。
- 密封性，由 sealed 关键字可见，sealed 特性为实现字符串恒等性和字符串驻留机制，提供了基础保证。

还有大量的命题值得关注，总结起来主要包括：

- 字符串比较：以等价规则而非恒等规则进行比较。
- 常用方法：Trim()、ToLower()、Replace()、Split()、PadRight()、SubString() 和 Join()。
- 格式化。
- 转移字符。
- StringBuilder。
- Encoding，编码。
- Culture & Internationalization，语言文化。
- overloads ==，== 重载。

由此可见，string 真是一个丰富多彩的技术仓库，包含了.NET 技术中很多精髓与内在，虽不能尽述其然，但不错的开局是深入下去的入口。

9.5.4 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，在操作方式上类似于其他基元类型 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 4.5 节“经典指令解析之实例创建”中已有详细的本质分析。

9.5.5 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 immutability 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：

- 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。

9.5.6 字符串驻留（String Interning）

首先给出 MSDN 对于字符串驻留的定义：公共语言运行库通过维护一个表来存放字符串，该表称为拘留

池（或叫驻留池），它包含程序中以编程方式声明或创建的每个唯一的字符串的一个引用。因此，具有特定值的字符串的实例在系统中只有一个。如果将同一字符串分配给几个变量，运行库就会从拘留池中检索对该字符串的相同引用，并将它分配给各个变量。

为了更详细地论述字符串驻留，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。

1. 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 9-3）来说明这一问题。

2. 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";
```

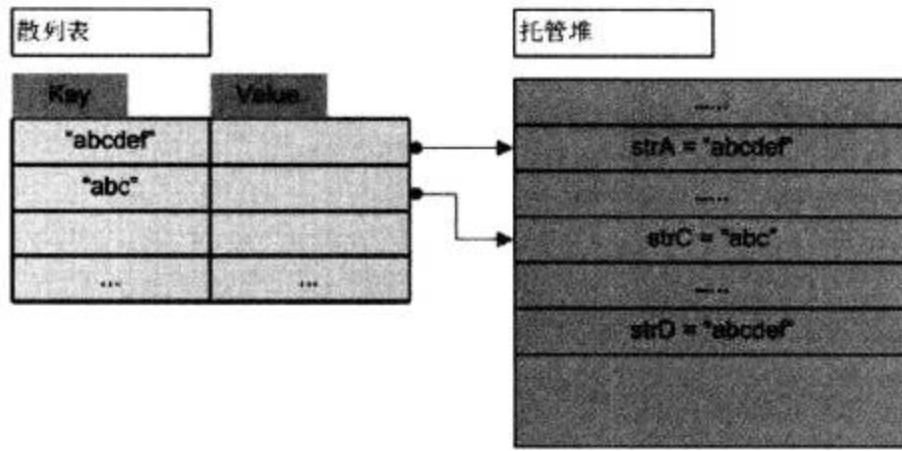


图 9-3 string 的内存概况

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”的 Hash Code 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。

通过这种方式，字符串驻留机制有效实现了对 string 的池管理，节省了大量的内存空间。

3. IsInterned 和 Intern

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，当需要高效地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);
```

对于 IsInterned，MSDN 的官方解释是：str 位于 CLR 的驻留池时，则 IsInterned(str) 将返回对 str 的引用；否则将返回 null 引用。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));
```

所以，很容易解释上述代码的执行结果了。

另外，一定注意的是 `IsInterned` 返回非 `null` 不代表两个字符串引用了相同的内存地址，例如开篇的测试示例：

```
static void Main()
{
    string s1 = "abc";
    string s2 = "ab";
    string s3 = s2 + "c";

    //返回 acb
    Console.WriteLine(string.IsInterned(s3) ?? "null");
    //返回 false
    Console.WriteLine(ReferenceEquals(s1, s3));
}
```

4. 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，驻留池在 CLR 加载时创建，分配在 `System Domain` 中，被进程中的所有 `AppDomain` 所共享，其生命周期不受 GC 控制。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。

9.5.7 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

- 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。

- 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 5.3 节“参数之惑——传递的艺术”。

5. 其他常用方法

表 9-1 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 9-1 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

9.5.8 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` 数组的字段，`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`，因为不必复制字符数组，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。

9.5.9 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,以至于 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。

9.6 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

9.6.1 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.Globalization.CultureInfo 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。

9.6.2 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 = int32(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。

- 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。

9.6.3 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);
```

2. 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)));
```

9.6.4 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他的整数值，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。

9.6.5 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 Studio 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。

9.6.6 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。

9.7 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

9.7.1 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。

9.7.2 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 myDelegate 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 Deleagte.Combine 和 Deleagte.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 [mscorlib]System.Delegate, class [mscorlib]System.Delegate)
//部分省略.....
IL_0043: call class [mscorlib]System.Delegate [mscorlib]System.Delegate::Combine
(class [mscorlib]System.Delegate, class [mscorlib]System.Delegate)
//部分省略.....
IL_0075: call class [mscorlib]System.Delegate [mscorlib]System.Delegate::Remove
(class [mscorlib]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 9-4 所示。

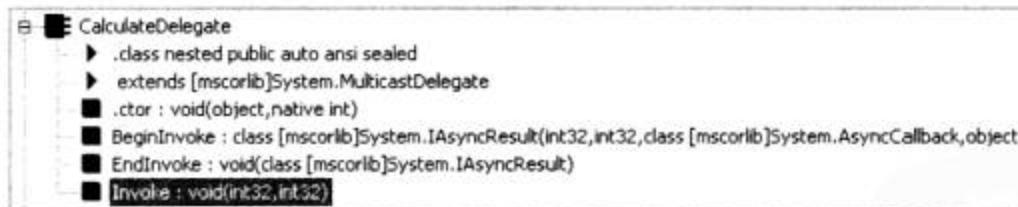


图 9-4 CalculateDelegate 的 IL 分析

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/
CalculateDelegate::ctor(object, native int)
    IL_000d:  stsfld     class InsideDotNet.NewFeature.CSharp3.DelegateEx/ CalculateDelegate
InsideDotNet.NewFeature.CSharp3.DelegateEx::myDelegate
    IL_0012:  ldsfld     class
    InsideDotNet.NewFeature.CSharp3.DelegateEx/Calculate Delegate InsideDotNet.NewFeature.
CSharp3.DelegateEx::myDelegate
    IL_0017:  ldc.i4.s  100
    IL_0019:  ldc.i4     0xc8
    IL_001e:  callvirt     instance void  InsideDotNet.NewFeature.CSharp3.DelegateEx/
CalculateDelegate::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 在代码中显式调用，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#编译器是不允许显式调用的，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。

9.7.3 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 Subtract(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.Subtract;
    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)
{
}

```

9.7.4 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 mySubtractDelegate = 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 9-5 所示。

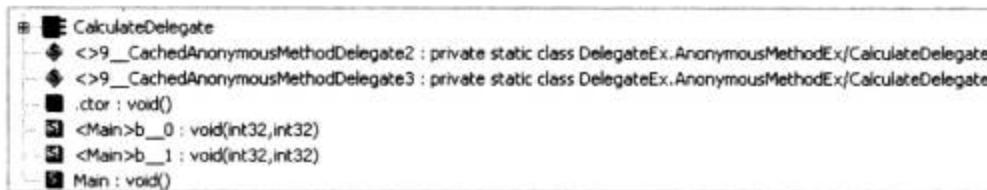


图 9-5 匿名方法的 IL 分析

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。

9.7.5 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。

9.7.6 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 `EventArgs` 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。

9.7.7 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。

9.8 Name 这回事儿

本节将介绍以下内容：

- Name、FullName 还有 AssemblyQualifiedName 探讨
- 不同 Name 的应用场合

9.8.1 引言

在.NET 中一个 Type 名称信息由三个属性来描述，分别是：

- Name，获取当前成员的名称。
- FullName，获取 Type 的完全限定名，包括 Type 的命名空间，但不包括程序集。
- AssemblyQualifiedName，获取 Type 的程序集限定名，其中包括从中加载 Type 的程序集的名称。事实上，AssemblyQualifiedName 被定义为只读 abstract 属性，具体的实现由其派生类来实现，例如 TypeBuilder，可以根据其具体实现类型对此有个大致的了解。

此处的定义毋庸置疑是官方的（MSDN），俗话说事实是检验真理的唯一标准，那么这三个相近的概念，究竟代表了怎样的不同呢？如果需要唯一的限定类型名又应该如何做出选择呢？回到事实近看分晓。

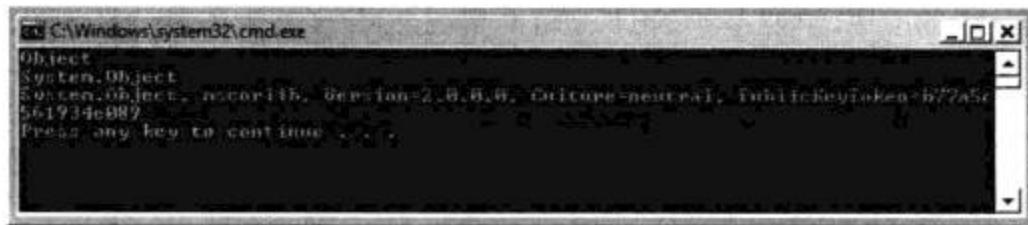
9.8.2 畅聊 Name

1. 由简单开始

由简单开始，不妨看看 object 的三个不同 Name 返回的事实真相：

```
static void Main(string[] args)
{
    Type t1 = typeof(object);
    Console.WriteLine(t1.Name);
    Console.WriteLine(t1.FullName);
    Console.WriteLine(t1.AssemblyQualifiedName);
}
```

执行结果呢？



诚如 MSDN 所说，Name 返回了简单的类型名称，FullName 包含命名空间，而 AssemblyQualifiedName 则包含程序集全名称。对于非强名称程序集，其 AssemblyQualifiedName 依然返回，相关的程序集信息：

```
Console.WriteLine(t3.AssemblyQualifiedName);
Anytao.Learning.ExpressionTree.One, Anytao.Learning.ExpressionTree, Version=1.0.
0.0, Culture=neutral, PublicKeyToken=null
```

2. 向复杂过度

如果我们只把目光停留在简单类型，那么这三个家伙也不值得花点小时间来注意了，除了简单，还得复杂。所以，只好把 Expression 拿来抓丁了：

```
static void Main(string[] args)
{
    Type t2 = typeof(Expression<Func<int, int>>);
    Console.WriteLine(t2.Name);
    Console.WriteLine(t2.FullName);
    Console.WriteLine(t2.AssemblyQualifiedName);
}
```

执行结果呢？



对于答案，引起关注的是在 Expression<Func<int, int>> 中，其 FullName 的 Func<int, int> 类型，以及 int 类型均获取到其 AssemblyQualifiedName，而不是 FullName。这留给读者一个大大的疑问，对其原因进行一点点深究，就可以有这样的思考， Func<T> 以及 int 分别存在于 System.Core 和 mscorelib 程序集中，对于本身程序集而言，完全有可能在其他引用程序集中引入一个 FullName 相同的 Assembly，所以为唯一限定起见，以 AssemblyQualifiedName 标示 Func<T> 和 int 是完全正确的。

同样的问题，还存在于 List<T> 等其他类型。任何可替换类型参数的实际类型，都可能由不同程序集的加载而变得不够“唯一”，所以 AssemblyQualifiedName 来限定 List<T> 的 FullName 是明智的。

3. 顺便看看 Type.ToString()

Type 类型还有一个 ToString()，用于返回 Type 的 Name，那么这个 Name 究竟是这三个中的哪一个呢？如果看了答案，肯定又一次崩溃：

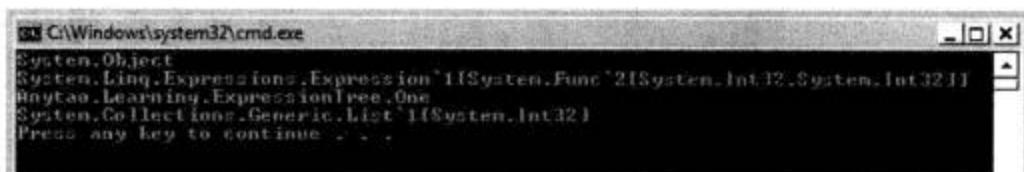
```
static void Main(string[] args)
{
```

```

Type t1 = typeof(object);
Type t2 = typeof(Expression<Func<int, int>>);
Type t3 = typeof(One);
Type t4 = typeof(List<int>);
Console.WriteLine(t1.ToString());
Console.WriteLine(t2.ToString());
Console.WriteLine(t3.ToString());
Console.WriteLine(t4.ToString());
}

```

此时的结果：



很无语，`Type.ToString()`事实上并未返回 `Name`、`FullName` 或者 `AssemblyQualifiedName`，而是不完全的 `FullName`，具体就不做过多陈述了，可以由结果看得很明白。

9.8.3 回到问题

显然，`FullName`在一个程序集中是唯一限定的，包含了所在的命名空间，而 `AssemblyQualifiedName` 则更包含其程序集名称，对于 `List<T>`这样的复杂类型，即便是 `FullName`也将以 `AssemblyQualifiedName` 显示其类型参数，所以对于很多通过 `FullName` 限定缓存 `Key` 的场合，`FullName`是完全胜任这一角色的。

```

static void MapperRegister()
{
    container = new Dictionary<string, IDataMapper>();
    container.Add(typeof(UserMapper).FullName, new UserMapper());
    container.Add(typeof(ProductMapper).FullName, new ProductMapper());
}

```

当然，`AssemblyQualifiedName`也同样可以胜任。

9.8.4 结论

`Name`这回事儿，看起来简单，需要的就是那么一点点挖掘。本文通过对比性的论述，为“更多”地了解这一最普遍的话题，解决了日常开发中可能需要特别关注的部分。

9.9 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

9.9.1 引言

内存耗尽、索引起越界、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，尽可能多地使用 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。

9.9.2 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 ExecuteSql(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 TestNoException(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 受保护区城，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 TestNoException(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::TestNoException

```

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，因此建议尽可能地以处理流程来规避异常处理。

9.9.3 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 = 0;
        GetResultToText(a, b);
    }

    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 DivideByZeroException 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.Exception 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。

9.9.4 .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 9-6 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。

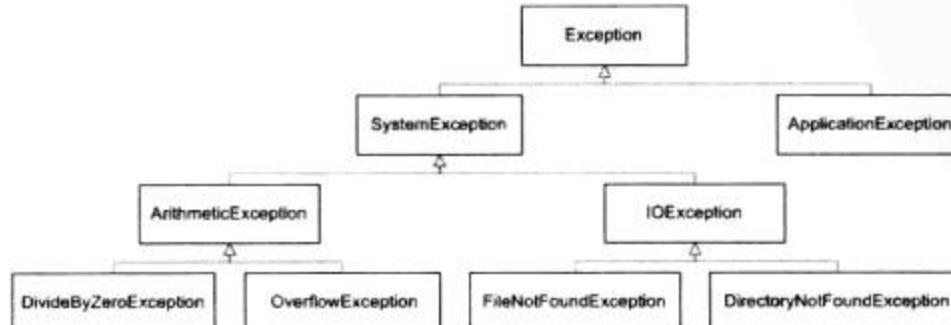


图 9-6 异常类的部分继承体系

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。
- 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。
- 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：

- Message 属性，用于描述异常抛出原因的文本信息。
- InnerException 属性，用于获取导致当前异常的异常集。
- StackTrace 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 = SecurityPermissionFlag.SerializationFormatter)]
    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。

9.9.5 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。

9.9.6 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 7.3 节“using 的多重身份”。
- 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，详见 7.4 节“认识全面的 null”。

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。

9.9.7 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。

参考文献

Christian Nagel, Bill Evjen, Jay Glynn, Professional C# 2005

Jeffrey Richter, Applied Microsoft .NET Framework Programming

Krzysztof Cwalina, Brad Abrams, Framework Design Guidelines: Conventions, Idioms, and Patterns for Reusable .NET Libraries

Stanley B. Lippman, C# Primer

Anand Kumar, Best Practices: Exception Management,

<http://www.dotnetjunkies.com/Article/197E493F-BA73-45A2-B39A-4EA282A2E562.dcit>

Artech, 关于 Type Initializer 和 BeforeFieldInit 的问题。

<http://www.cnblogs.com/aritech/archive/2008/11/01/1324280.html>

李永伦, 通过七个关键编程技巧得益于静态内容。

<http://www.cnblogs.com/allenlooplee/archive/2007/01/22/627386.html>

第10章 格局之选——命名空间剖析

10.1 基础——.NET 框架概览 / 384

10.1.1 引言 / 384

10.1.2 框架概览 / 384

10.1.3 历史变迁 / 385

10.1.4 结论 / 387

10.2 布局——框架类库研究 / 387

10.2.1 引言 / 387

10.2.2 为什么了解 / 388

10.2.3 框架类库的格局 / 388

10.2.4 一点补充 / 389

10.2.5 结论 / 390

10.3 根基——System 命名空间 / 391

10.3.1 引言 / 391

10.3.2 从基础类型说起 / 391

10.3.3 基本服务 / 392

10.3.4 结论 / 394

10.4 核心——System 次级命名空间 / 394

10.4.1 引言 / 394

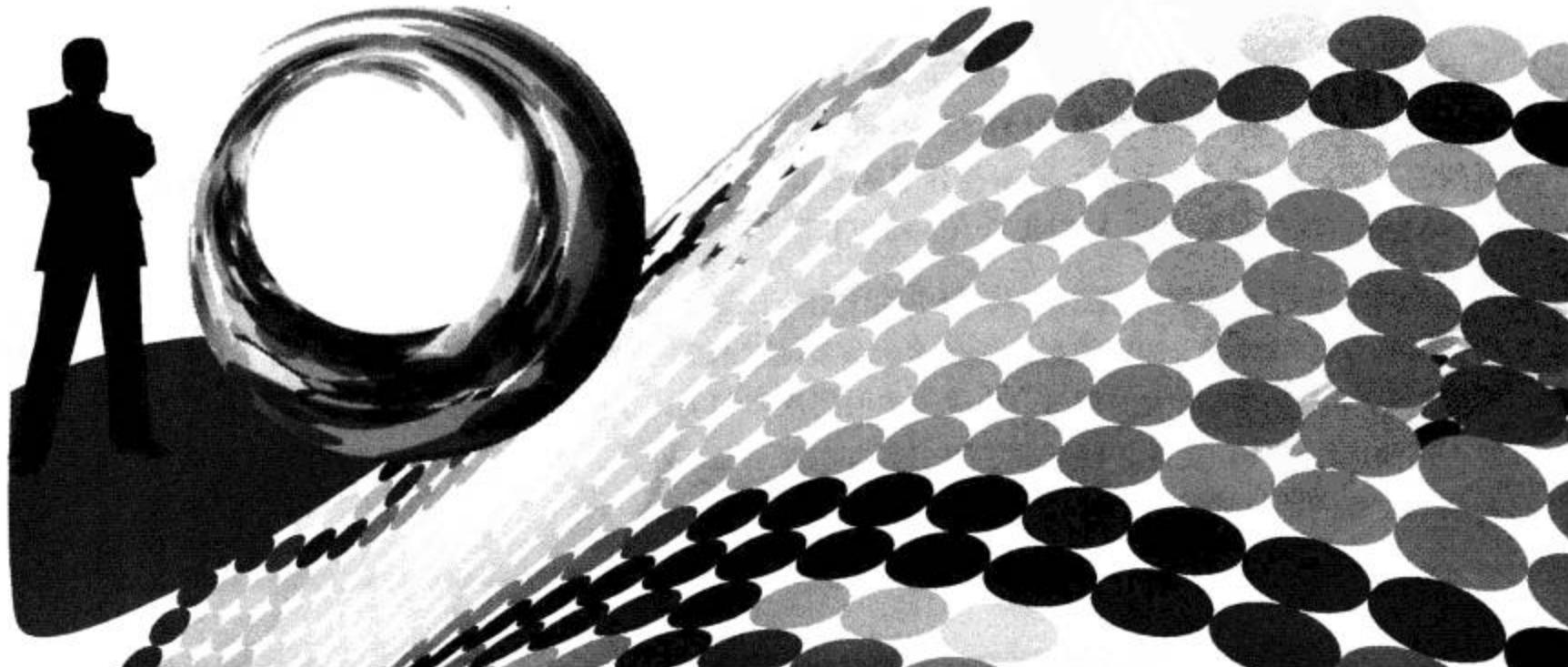
10.4.2 System.IO / 395

10.4.3 System.Diagnostics / 396

10.4.4 System.Runtime.Serialization 和
System.Xml.Serialization / 397

10.4.5 结论 / 399

参考文献 / 399



10.1 基础——.NET 框架概览

本节将介绍以下内容：

- .NET 框架结构
- .NET 框架的历史变迁

10.1.1 引言

直到现在，本书才在此处着笔来了解.NET 框架的基本面貌，这主要是为本章的基本内容——框架类库（Framework Class Library，FCL）格局，做以铺垫。因此，作为.NET Framework 的一部分，我们有理由先了解一下框架类库在.NET 体系中的位置，以及在.NET 历史发展中的变迁，这些对于从宏观角度来认识框架类库大有帮助。

10.1.2 框架概览

.NET Framework (.NET 框架) 主要包括两个部分：CLR（通用语言运行时）和 FCL（框架类库）。其中，CLR 是.NET Framework 的基础，提供了包括内存管理、线程管理和远程处理等核心服务；FCL 是一个基于面向对象的可重用类型集合，用于支持多种应用的快速开发，例如：ASP.NET Web 应用、Windows Form 应用、Web Services 应用和企业服务等多种应用与服务。

.NET Framework 的构成，就以一张经典的图示（图 10-1）来了解：

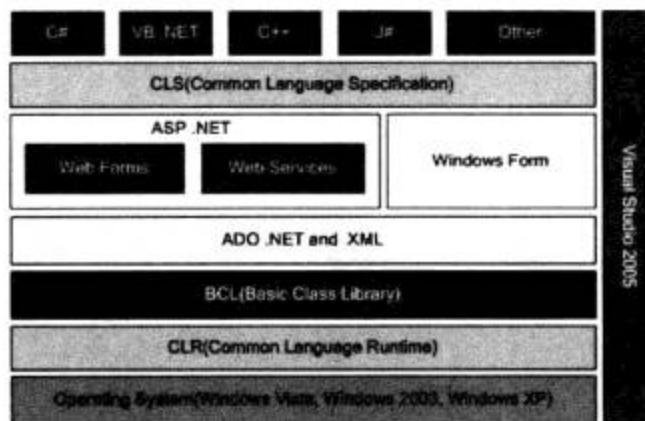


图 10-1 .NET Framework 的构成

由框架的结构组成可知，CLR 是.NET 体系的基础，所有的.NET 代码都运行在 CLR 之上，并且与 FCL 紧密结合，并由此创建基于.NET 的 Windows Forms、Web Forms 和 XML Web Services 等应用程序。而在这个框架的下层，CLR 运行于 Windows 操作系统之上；框架的上层，则是基于.NET 的高级开发语言，目前支持的.NET 语言主要包括 C#、VB.NET、C++.NET 等。

本节的重点是了解 FCL 在.NET 框架中的作用与位置，因此我们将注意力落在框架类库上，继续我们的认识。FCL 包含了数以千计的类型，而且这一数量随着.NET 类库的发展还在继续增加，.NET 通过命名空间在逻辑上组织数量如此巨大的类库，形成一定的层次和归类，了解.NET 就应对 FCL 的核心类型有所了解。开发不同的应用程序，就应了解需要的类型包含在哪个命名空间，本章将逐渐对类库的结构做以梳理，建立起对基本类库的逐步理解。

10.1.3 历史变迁

.NET 概念自 2000 年诞生以来，经历了 7 年的发展，.NET Framework 也由 2002 年发布的.NET 1.0 升级到 2007 年年底的.NET 3.5，一路走来，每一次的更新都给业界带来革命性的颠覆，新的技术层出不穷。关注其历史，也正是关注其脉络，了解技术层级中的变迁，也更有利于通过理解由升级而带来的差异。例如，.NET 2.0 引入泛型技术，一定程度上化解了.NET 1.0 中由装箱与拆箱引起的性能问题；由.NET 1.0 的委托，到.NET 2.0 的匿名方法，再到.NET 3.0 的 Lambda 表达式，自下而上都是相辅相成的，我们可以很容易由其发展看到联系。因此，了解变迁，旨在关注其联系。

1. .NET 2.0 新特性

.NET 2.0 发布于 2005 年，.NET 框架做了很多 API 的更新，引入很多新的特性，包括对 CLR 和类库都有较大的改进，这些特性主要包括：

- 泛型。泛型是.NET 2.0 带来的巨大改进之一，虽然泛型概念并非崭新的技术，然而在.NET 底层架构中引入泛型，也就意味着.NET 的高级语言中都可以方便的使用泛型技术进行开发。泛型使得集合的功能更加强大，效率更高更灵活，关于泛型的应用详见本书第十章“接触泛型”的内容。
- 部分类。部分类可以将单个类分解到多个文件中，而在编译时会将这些文件合并，当作一个类处理。例如，可以在 a.cs 文件中写入如下代码：

```
public class HelloWorld
{
    public void SayHello()
    {
        Console.WriteLine("Hello!");
    }
}
```

而在 b.cs 文件中保存如下代码：

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

可以由 Visual Studio 的智能感知，看到保存在两个文件的 HelloWorld 类，被自动合并为一个类，HelloWorld 实例既可以调用 SayHello 方法，也可以调用 SayHelloWorld 方法，如图 10-2 所示，例如：



图 10-2 部分类的智能感知

- 迭代器。迭代器使得在自定义类中实现 foreach 遍历成为可能，该类必须实现 IEnumerable 接口，本书 8.9 节“集合通论”有所涉猎。
- 匿名方法。匿名方法可以更方便地处理委托，使得方法操作可以直接放在委托中，而不需要另外创建全新的方法。例如：

```
class AnonymousMethodEx
{
    delegate void MyDelegate();

    public void InvokeHello()
    {
        MyDelegate myDelegate = delegate()
        {
            Console.WriteLine("Hello, world.");
        };

        myDelegate();
    }
}
```

- 可空类型。由于泛型机制的引入，使得在.NET 2.0 中可以以 Nullable<T> 来表示可空的值类型。
- 支持 64 位。CPU 已经向着 64 位体系阔步转移，.NET Framework 也必须适应这种变更，在.NET 2.0 中提供了专门用于 64 位的指令集。在 Visual Studio 2005 中建立的应用程序，可以在项目属性中简单的修改编译属性以建立基于 64 位计算机的应用程序。

其他的更新，例如 ASP.NET、ADO.NET 方面都有较大的调整和改进，在此就不做详述。

2. .NET 3.0 新特性

.NET 3.0 依然使用.NET 2.0 的 CLR，这意味着任何基于 2.0 的程序都可以稳定的运行在 3.0 平台之上。在此基础上，.NET 3.0 加入 4 个新的框架：WPF（Windows Presentation Foundation）、WF（Windows Workflow Foundation）、WCF（Windows Communication Foundation）、WCS（Windows CardSpace）。如图 10-3 所示。

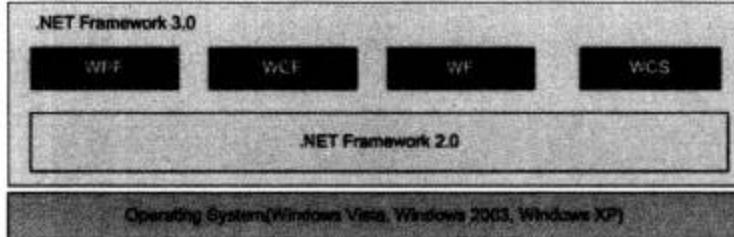


图 10-3 .NET 3.0 的构成

其中，WPF 作为新的图形引擎，为未来的软件带来革命性的界面体验，并且实现将图形元素和业务元素分离的开发模式；WF 是一个企业级工作流开发框架和引擎，提供了面向流程的软件开发模式选择；WCF 通过 SOAP 提供强大的交互通信支持，使得我们可以通过一致的编程模式，使用不同的技术来构建分布式应用系统；WCS 用于创建一个身份标识元系统，并为不同的身份标识管理技术提供一个统一的框架，以支持其工作。

除了引入 4 个新的框架，.NET 3.0 在语言特性方面也有较大的改进，以 C# 为例，扩展方法、匿名类型、Lambda 表达式、对象和集合初始化器等，这些特性为更好地实现软件开发带来便利。.NET 3.5 的更新尘埃未定，.NET 3.5 已经浮出水面，例如在编程语言中内置轻量级的 LINQ 数据查询模型，在类库加入新的类型等。本书在第 12 章“.NET 3.0/3.5 新革命”部分对这些内容做了相应的介绍与分析。由图 10-4 我们可以直观地了解到.NET

历史变迁的轨迹，也更加清楚不同版本之间的变化和更新，例如由 2.0 到 3.0，再到 3.5 版本，CLR 内核保持基本不变，而是在语言特性和新增框架方面有了较大的调整。

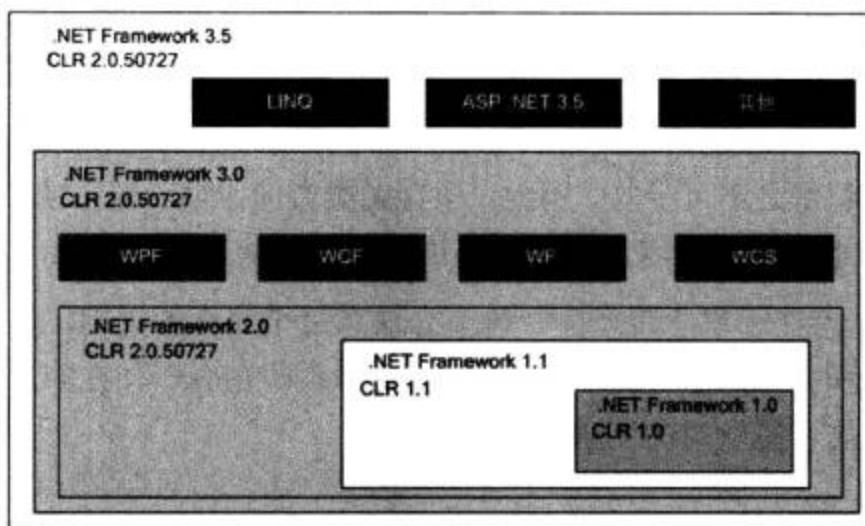


图 10-4 .NET 3.5 的构成

由此可见，在.NET Framework 的发展历史上，2.0 可以看作 1.0 的超集，而 3.0 可以看作 2.0 的超集，每次的版本升级都伴随着新技术的出现和大量 API 的修改，但是微软工程师尽量保证对原有系统做尽可能少的更改。因此，框架类库的变更也是朝着这一方向做出的更新，这为我们更好的理解类库提供了相对稳定的积累过程，对于.NET 1.0 的投资永远不会因为版本的升级而浪费。

10.1.4 结论

对.NET Framework 有了基本的了解，对于我们更好地认识.NET 基础类库大有裨益。理其构成，观其发展，就能更好把握对.NET 基础类库的理解，也对.NET 各个命名空间的功能有了追溯的依据。

在本书即将面世之际，期待已久的.NET Framework 在 2008 年的暖风中终于开源了。对于.NET 开发者而言，这是前所未有的好消息，也标志着对于.NET Framework 类库，我们可以从更接近的角度来看清其面目。如饥似渴的研究这些基础类库，对于热情饱满的.NET 程序者来说，实在是一顿幸福的大餐。

10.2 布局——框架类库研究

本节将介绍以下内容：

- 框架类库的认识
- .NET 命名空间层次结构
- 学习的意义

10.2.1 引言

本节从全局来了解.NET 框架类库（FCL）的基本构成和层次关系，并简单地了解各个主要命名空间的基本功能，从而建立起一个把握全局的认知网络，为深入了解 FCL 打好基础。

10.2.2 为什么了解

有人说,.NET技术的认知和深入,建立在两个基本的方面:一是.NET基本知识和底层机制的研究,这是本书大部分章节的研究工作;二是.NET类库的了解和熟练,这是本章所要力求解决的问题。

显然,数以千万计的.NET类库是任何狂人都无法全面掌控的,也没有必要全面了解。技术的学习在于应用而不在于全面,然而积累却是必不可少的。因此,学会触类旁通,举一反三就显得至关重要。Visual Studio工具的驾轻就熟,MSDN帮助的熟练使用,成为了解技术的必要辅助条件。

因此,熟悉.NET类库,不在于熟悉每个类,每个方法和每个属性的规则,那首先是不可能完成的任务。而更可行的办法就是以骨架和主线入手,将.NET类库形成体系印在心中。在应用的时候,知道在什么地方,找什么类型,用什么方法,这才是学习的制胜法宝。

10.2.3 框架类库的格局

《Framework Design Guidelines》一书强调,一个成功的通用框架必须遵循分层架构原则,不同的开发人员对框架的期望层次不同,以便能够进行不同程度的控制。.NET类库不仅要提供较为底层的通用API,还应建立面向特定应用高层的API,而且不同的抽象应该建立在不同的层次上,形成一定的树状体系划分。

这种层次划分,在.NET中由命名空间来实现。作为逻辑上的组织结构,命名空间由类、结构、枚举、委托和接口组成,命名空间名作为类型的完全限定名,起到有效的层次划分与组织管理作用。例如,我们截取FCL的一个部分结构(如图10-5所示)做一了解:

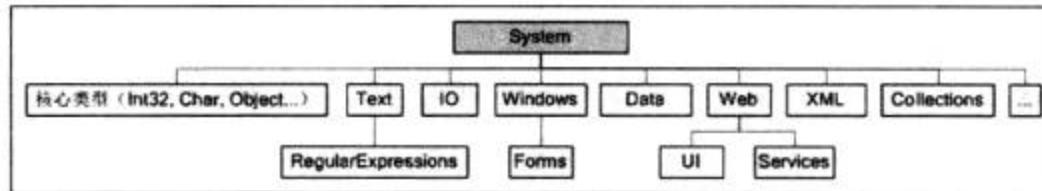


图10-5 FCL的部分结构

其中, System 命名空间位于这个树状体系的最高层,包含了CLR的核心数据类型、接口和特性。而其他的命名空间被称为 System 次级命名空间,分别提供了特定的应用。在此,我们将 FCL 最常用的命名空间用表 10-1 做以梳理和简介,这有助于从全局建立一个基本的理解。

表 10-1 最常用的命名空间

序号	命名空间	简要描述
1	System	包含CLR的基本类型和基类,定义了常用的值类型和引用类型,事件、接口、属性和异常处理等
2	System.Text	包含用于文本处理的类,实现以不同编码方式操作文本。其中的StringBuilder类能够实现对可变字符序列的字符串进行高效处理
3	System.IO	操作I/O流,提供了处理文件、目录和内存流的读写与遍历操作等
4	System.Windows.Forms	包含了用于创建Window GUI应用程序的类,以利用Windows操作系统提供的丰富界面元素
5	System.Data	提供的各种类实现了ADO.NET,实现了.NET访问数据的标志途径。其中包含了数个次级命名空间,来实现对不同数据源的访问与操作,例如System.Data.SqlClient用于访问SQL Server数据, System.Data.OleDb 用于使用OLE DB 方式访问数据库

续表

序号	命名空间	简要描述
6	System.Web	是.NET最重要的命名空间之一。用于实现ASP.NET应用和ASP.NET Web Services的基础类库
7	System.XML	包含了处理XML文档的基础类
8	System.Collections	包含了常见的集合类，例如：Hashtable、ArrayList、Queue等
9	System.Reflection	提供了能够查看程序集元数据的类型，以实现操作程序集、模块、方法等。次级命名空间System.Reflection.Emit则包含了可以动态创建其他类型的类
10	System.Threading	提供了基于.NET开发多线程应用系统的标准方式，实现包括线程、线程池管理；线程同步机制。核心的类型是Thread，它可以创建并控制线程；而Timer类则提供了以指定时间间隔执行方法的机制
11	System.Diagnostics	包含能够与系统进程、事件日志和性能计数器进行交互的类，一般用于帮助诊断和调试应用程序。例如Debug类用于帮助调试代码；Process类能够控制进程访问；Trace类则能够跟踪代码的执行情况
12	System.Globalization	提供了多国语言支持的类
13	System.Drawing	提供支持GDI+服务的接口类型，用于操作二维图形、字体、图元文件等
14	System.ComponentModel	提供了实现基于.NET的控件和组件。其中的Component类是CLR按引用封送的所有组件的基类
15	System.Net	包含用于网络通信的类型，为各种网络协议提供编程接口。其中的WebRequest和WebResponse提供了统一资源标识符的请求和响应，而不必考虑各种协议的应用细节
16	System.Runtime	包含了几个重要的次级命名空间，System.Runtime.InteropServices命名空间用于在托管代码中访问非托管代码的功能；System.Runtime.Remoting则提供了用于远程访问的类；System.Runtime.Serialization用于提供序列化和反序列化功能
17	System.Configuration	包含访问应用程序配置信息的类，常用的类型是ConfigurationSettings用于在运行时读取配置信息
18	System.Security	提供了CLR安全系统基础结构，用以支持加密、安全策略、安全原则、权限设置和证书等服务
19	System.EnterpriseServices	为企业应用程序提供基础结构，可以访问COM+服务，用于管理分布式事务、对象池、队列组件、安全等特性
20	System.Transactions	包含创建事务处理和资源管理的类，使得事务处理变得简单、高效

在表 10-1 中，我们仅仅列出了最常用、最通用和最重要的 20 个命名空间及其简介，更多的命名空间只能在平时的应用和实践中逐渐积累，最重要的是由常见命名空间的简介，对于基本方向有了初步的了解。

对常见命名空间进行必要的梳理是很有必要的，图 10-6 则从功能上对上述命名空间进行了大的分类，主要包括了.NET 基础服务、创建 Web 应用、创建 Windows 窗体应用、数据处理和图形处理几个大类，如图 10-6 所示。

10.2.4 一点补充

- System 是.NET中所有基本类型的根命名空间，而global命名空间则可以看成是所有命名空间的“根”，例如global::System 将总是引用FCL命名空间System，而不管System是否被包含类隐藏，例如：

```
using System;
class GlobalNamespace
```

```

{
    public class System
    {

    }

    public static void Main()
    {
        System.Console.WriteLine("Hello");
    }
}

```

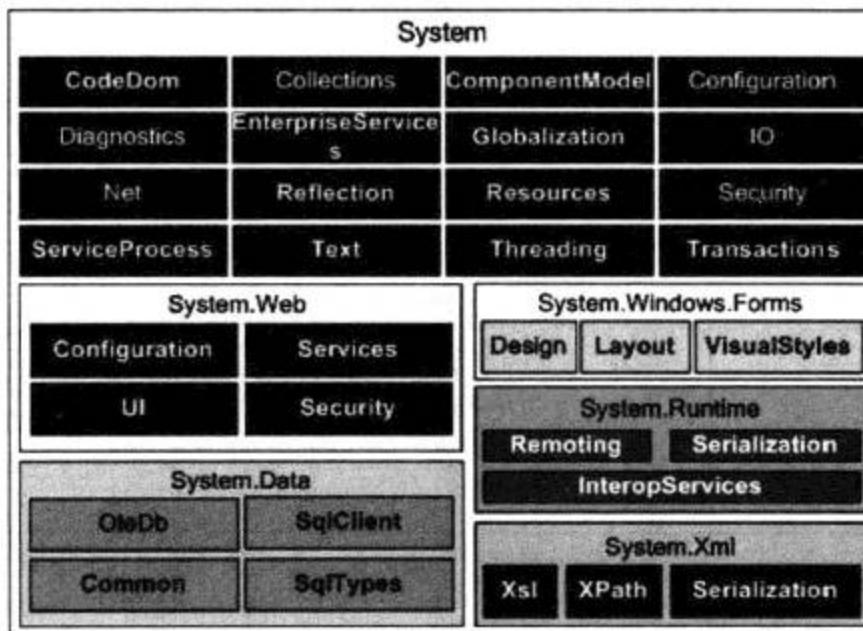


图 10-6 .NET 的常见命名空间

上述代码将出现编译时错误，因为内部类 System 屏蔽了 FCL 定义的 System 命名空间，此时可以使用 global 命名空间来直接引用.NET 的 System，因为 global 标识将使得对命名空间的搜索由全局开始。例如：

```
global::System.Console.WriteLine("Hello");
```

- 命名空间只是一种逻辑结构，而并非物理结构，不同的程序集可以定义同一命名空间的类型，不同的命名空间也可以包含在同一程序集中。命名空间之间不一定就是父子关系，例如：System.Windows 和 System.Windows.Forms 之间并不是父子关系，事实上.NET 中并不存在 System.Windows 这一命名空间。
- 命名空间的一般格式为“公司名.技术名”这样的形式，例如微软的命名空间就主要以 System 或 Microsoft 开头。
- 微软在线 MSDN 为详细的了解 FCL 提供了丰富的文档资料，是深入学习和理解.NET 类库的最佳资源库。

10.2.5 结论

本节从梳理.NET 类库的框架入手，以简洁的视角来透视复杂的类库结构，将最重要、最核心的部分，形成一个比较清晰的体系，每个人在这一过程中，都将建立起自己的类库偏向体系。例如，数据库应用系统应该对 System.Data 命名空间有较深的理解，而 Web 窗体应用则应关注 System.Web.UI 命名空间。这些挑战将会伴随每个程序设计者的职业生涯，在 Baidu、Google 之余将更多的经验积累起来很有必要。

10.3 根基——System 命名空间

本节将介绍以下内容：

- System 命名空间解析
- 基础类型和基础服务

10.3.1 引言

System 命名空间，就像一个最熟悉的“陌生人”。虽然在.NET 世界里到处都可以见到它的身影，但你不见得就了解它的方方面面。System 命名空间是 FCL 基本类型的根命名空间，所有的基础类型都定义在 System 命名空间中。因此，几乎在所有系统代码中，我们都可以看到 using System 这样的操作，用于引入该命名空间。

除了定义基本类型，在 System 命名空间中，我们还应该了解哪些基本内容和基础信息呢？

10.3.2 从基础类型说起

由上文 10.2 节“布局——框架类库研究”，我们简单地了解到 System 命名空间是.NET 基础类型的根命名空间，主要包含了通用类型系统所定义的核心类型，例如 Int16、Int32、Int64、Char、Boolean、Double 等值类型，还包含 String、Delegate、Array 等引用类型，以及一切类型的最终基类 Object 类型。

下面，我们简单分析 System 命名空间中提供的几个重要的基础类型，这些类型可以认为是了解.NET 的重要基础之一，也是本书各个章节的重点阐述对象，必须被每个程序开发者所熟悉，主要包括：

- Object，一切类型的最终基类，在.NET 中任何类型都直接或者间接的继承自 Object 类，它提供了所有类型的基本方法特性，详细参见 9.1 节“万物归宗：System.Object”。
- String，使用最频繁的类型之一，提供了操作字符串的基本方法，详细参见 9.5 节“如此特殊：大话 String”。
- Delegate，委托类型的基类。委托类型实例建立了一个指向一个或者多个方法的签名，实现了类型安全的回调机制，同时可以支持静态方法和实例方法，详细参见 9.7 节“一脉相承：委托、匿名方法与 Lambda 表达式”。
- Attribute，定制特性类型的基类。.NET 通过定制特性可以实现将预定义信息与程序集、类、构造函数、方法、属性相关联，并在运行期通过元数据来获取其信息，从而达到改变代码指向方式的目的，详细参见 8.3 节“历史纠葛：特性和属性”。
- Type，为 System.Reflection 命名空间的类型提供了有效的元数据信息，通过 Type 对象可以获取到类型的声明信息。而 Object.GetType 方法会返回表示类型实例的 Type 对象，这样我们就可以很容易通过 Type 对象和反射机制在运行时动态创建和调用类型的功能。在 9.1 节“万物归宗：System.Object”中对此有所论述。
- ValueType，值类型的基类。详细参见 5.2 节“品味类型——值类型与引用类型”。
- Array，所有数组类型的基类，提供了创建、操作、排序数组的各种方法，详细参见 8.9 节“集合通论”。
- Nullable，提供了处理值类型空值的数据结构，借助于泛型机制，Nullable 实例可以表示任何值类型和

其对应的空值，例如我们可以像引用类型一样用 Nullable 实例和 null 进行比较：

```
public static void Main()
{
    Nullable<int>[] arrs = new Nullable<int>[5];
    arrs[0] = 100;
    arrs[1] = 200;
    arrs[3] = 300;

    foreach (int? j in arrs)
    {
        if (j == null)
            Console.WriteLine("No value");
        else
            Console.WriteLine(j);
    }
}
```

在 C# 中，可以通过简洁的写法来代替 Nullable 对象的声明，例如：

```
int?[] arrs = new int?[5];
```

System 命名空间定义了.NET 框架最基础的类型，这些类型成为构建.NET 框架类库的根本基础，深入地了解 System 命名空间的基础类型，是每个.NET 程序设计者的必修课。

10.3.3 基本服务

除了对基础类型的定义，System 命名空间中还包含异常处理、控制台输入输出、垃圾回收、数据类型转换、数学运算、产生随机数、应用程序环境管理等服务，为.NET 框架提供了最强大的底层功能和服务支持。下面，我们对这些通用服务做以简要的总结：

- Exception，所有异常类的基类，异常发生时应用程序通过引发包括错误信息的异常来报告异常信息，并由异常处理程序进行处理，关于 Exception 类和异常机制的详细内容可参见 9.9 节“直面异常”。
- Console，表示控制台应用程序的标准输入/输出流和错误流，其中 Write 方法用于将文本写入到标准输出流，而 Read 方法则用于从标准输入流读取下一个字符。本书的大部分程序都以 Console 类型方法将执行结果输出到控制台。
- GC，用于控制垃圾回收器。其中 Collect 方法提供了强制执行垃圾回收的机制，但是这一做法不被推荐，因为在托管代码中垃圾回收的工作应该交由 CLR 负责，关于垃圾回收机制可参见 6.3 节“垃圾回收”。
- AppDomain，表示应用程序域，为执行代码提供了有效的隔离和虚拟的安全边界，使一个域的应用程序不能直接访问另一个域的代码，从而保证了系统的安全性。AppDomain 提供了大量的方法用于完成创建域、卸载域、在域中加载程序集和类型、遍历域中的程序集和类型等操作。
- Convert，实现基本数据类型之间的转换，大部分的基本类型都支持这一操作，不过应该注意某些无效转换引发的异常，或者数据转换的精度损失。例如：

```
public static void Main()
{
    try
    {
        double num = 123.456789;
        //正常转换，返回为"123.456789"
        string str = Convert.ToString(num);
```

```

    //转换发生精度损失，返回为
    Int32 iNum = Convert.ToInt32(num);
    //无效转换，引发异常
    DateTime dt = Convert.ToDateTime(num);
}
catch (InvalidCastException ex)
{
    Console.WriteLine("发生无效转换。");
}
}

```

关于类型转换可参见 8.5 节“恩怨情仇：is 和 as”和 5.2 节“品味类型——值类型与引用类型”。

- Math，定义了几十个公共方法用于完成常见的数学运行，例如 Abs 返回数字绝对值、Sin 用于指定角度的正弦值、Sqrt 用于计算平方根、Log 返回指定数字的对数，等等。
- Random，用于计算伪随机数。例如：

```

public static void Main()
{
    Random rd = new Random();
    //产生 5 个随机数
    for(int i = 0; i < 5; i++)
    {
        Console.WriteLine(rd.Next());
    }
}

```

- Environment，用于获取当前环境和平台的信息，比如当前目录信息、当前操作系统信息、环境变量设置等，例如：

```

public static void Main()
{
    //获取当前目录
    Console.WriteLine("当前目录: {0}", Environment.CurrentDirectory);
    //获取机器名
    Console.WriteLine("机器名: {0}", Environment.MachineName);
    //获取操作系统信息
    Console.WriteLine("操作系统: {0}", Environment.OSVersion);
    //获取当前用户名
    Console.WriteLine("当前用户: {0}", Environment.UserName);
}

```

- MarshalByRefObject，支持远程处理的应用程序跨 AppDomain 边界访问对象。而 MarshalByRefObject 是所有可以在 AppDomain 边界外部访问的对象的基类，按引用机制进行封送，也就是 MarshalByRefObject 对象可以跨应用程序域边界被引用，也支持远程引用。
- TimeZone，表示时区。例如 CurrentTimeZone 属性表示当前计算机系统的时区，StandardName 属性表示标准时区名称。
- Version，表示 CLR 的版本号。可以通过例如 Major、Minor 和 Revision 来分别获取当前 CLR 程序集的主版本号、次版本号和修订版本号。
- Uri，提供了对统一资源标识符（Uri）对象封装，以便轻松操作 Uri 信息。例如：

```

public static void Main()
{
    Uri uri = new Uri("http://www.anytao.com/");
    //应用 Uri 实例创建 WebRequest
}

```

```

    WebRequest request = WebRequest.Create(uri);
    WebResponse response = request.GetResponse();

    Console.WriteLine(response.Headers);
    response.Close();
}

```

当然，System 命名空间包含的基本类型和支持服务远远不止本节呈现的内容，100 多个类型提供了各种较为基础的通用服务与支持，对于绝大部分的应用系统来说了解这些类型非常有用。

10.3.4 结论

了解了 System 命名空间，我们会发现自己的“眼界”确实开阔了不少，虽然只是泛泛的接触，但是我们的触角已经从最基本 Object 类型，触及委托、数组、数据转换、跨应用程序域引用、垃圾回收等较为高级的话题，这真的是不错的开端。

这是撑起.NET 之舟遥远航程的起点。在某种意义上，本书的大部分内容就是沿着这个起点开始，将.NET 基本技术做了一次深度遍历，而这个遍历的过程正是我们的学习和深入的方向。

10.4 核心——System 次级命名空间

本节将介绍以下内容：

- System 次级命名空间解析
- 常见输入输出操作
- 系统调试和诊断探索
- 序列化和反序列化简析

10.4.1 引言

经过对 System 命名空间的全盘解码，使得我们更有理由相信，从全局角度将 FCL 的基本面貌梳理一番是很必要的。其作用正如前文所言，主要是建立起对基本框架的全局把握，为具体的应用系统提供思路和方向。

本节我们继续这一梳理过程，而将关注的焦点放在 System 次级命名空间的分析上。然而，从 10.2 节“布局——框架类库研究”一节对于 FCL 体系分析的基础上可知，System 的次级命名空间实在太庞大了，并非本节甚至本书的篇幅所能完全描述的。因此，对于次级命名空间的分析不同于对 System 命名空间的分析，我们只关注最基本最常见的命名空间及其类型。并且，对于有针对性应用的命名空间，例如 System.Web、System.Data、System.Windows.Forms、System.EnterpriseServices、System.Transactions 只能忍痛割爱不做分析，因为具体的应用并非本书的方向，对类似于 Web 应用开发、ADO.NET 开发、Windows 应用程序开发这些主题有大量的其他专著来讲述。

本节不对所有基础命名空间进行分析，同时某些命名空间的研究在本书的其他部分已经做了深入的分析，例如 8.9 节“集合通论”就包括对 System.Collections 的论述，因此本节就选取几个比较常见的命名空间做以分析，主要包括：

- System.IO
- System.Diagnostics
- System.Runtime
- System.XML

其他的命名空间，如 System.Threading、System.Reflection、System.Globalization 等，只能在各自的专题中做以讨论，本节不做赘述。

10.4.2 System.IO

输入输出是最常见的程序操作内容之一，进行文件操作、目录操作和内存流操作是 System.IO 命名空间中各种类型的职责所在。

- File 和 FileInfo，提供了文件操作的各种方法：创建、复制、删除、移动和打开等。其中 File 类主要提供静态方法，而 FileInfo 类提供实例方法。
- Directory 和 DirectoryInfo，提供了目录操作的各种方法：创建、移动和枚举等。其中 Directory 类主要提供静态方法，而 DirectoryInfo 类主要提供实例方法。
- Path，提供了处理目录字符串的各种操作。例如 Path.HasExtension 用于判断是否包含文件扩展名，Path.GetTempPath 用于返回系统临时文件夹路径，Path.GetFileName 则用于返回路径字符串的文件名和扩展名。
- Stream，所有流的抽象基类，提供了对字节序列的抽象和基本的流操作方法，Read 用于读取流，Write 用于写入流，Seek 用于查找流，Flush 用于清除内部缓冲区并将数据写入基础数据源（例如磁盘驱动器），Close 用于刷新数据，并释放系统资源。
- FileStream，File 和 FileInfo 方法仅提供了打开、删除一个文件的方法，但是如何操作文件的内容，可以由 FileStream 来完成。FileStream 支持同步或者异步访问文件内容，Read 用于读取字节块，Write 用于将字节块写入，Seek 用于移动读写位置到文件的任意指定值，Close 方法则用于关闭 FileStream。另外，FileStream 只支持二进制格式的数据。例如：

```
public static void Main()
{
    using(FileStream fs = File.Create(@"c:\temp.txt"))
    {
        byte[] txts = new UTF8Encoding(true).GetBytes("我是小王。");
        //将字节块写入 FileStream
        fs.Write(txts, 0, txts.Length);
    }
}
```

- StreamReader 和 StreamWriter，StreamReader 以特定编码从字节流中读取字符，StreamWriter 以特定的编码向流中写入数据。它们的默认编码均为 UTF-8 格式，例如：

```
public static void Main()
{
    string filePath = @"c:\temp.txt";
    //创建一个文本文件
    if (!File.Exists(filePath))
```

```

    {
        FileStream fs = new FileStream(filePath, FileMode.Append);
        fs.Close();
    }

    //使用 StreamWriter 向文件中写入文本
    using(StreamWriter sw = new StreamWriter(filePath, true, Encoding.UTF8))
    {
        sw.WriteLine("我是小王。");
        sw.WriteLine("我的年龄是。");
    }

    //使用 StreamReader 从文件中读取文本
    using (StreamReader sr = new StreamReader(filePath))
    {
        string txt;
        while ((txt = sr.ReadLine()) != null)
        {
            Console.WriteLine(txt);
        }
    }
}

```

- StringReader 和 StringWriter，功能上类似于 StreamReader 和 StreamWtriter，不同的是 StringReader 和 StringWriter 用于处理内存中的字符串，而并非文件。

System.IO 命名空间中，还有很多的类用于完成特定 I/O 流的操作，例如 BinaryReader、BinaryWriter、MemoryStream、NetworkStream 等，.NET 提供了一整套全面的选项来完成对输入输出流的操作，并力求达到以统一的方式来施行。

10.4.3 System. Diagnostics

System.Diagnostics 命名空间，包含了能够与系统进程、事件日志和性能计数器进行交互的类，一般用于帮助诊断和调试应用程序。例如 Debug 类用于帮助调试代码；Process 类能够控制进程访问；Trace 类则能够跟踪代码的执行情况。

- Process，用于操作本地或者远程进程的访问，通过 Process 可以在托管环境下很容易的操作对外部进程的启动或者停止，例如下例中通过 Process 方法启动 IE 进程打开指定 Url：

```

public static void Main()
{
    Process myProcess = new Process();
    string myUrl = "http://www.anytao.com";

    //必须设置相应的 FileName 和 Arguments 属性
    myProcess.StartInfo.FileName = "iexplore.exe";
    myProcess.StartInfo.Arguments = myUrl;
    //启动进程
    myProcess.Start();
}

```

- Debug 和 Trace，这两个类能够帮助系统开发者有效地进行系统调试和追踪，提供了很多基本相同的方法用于系统调试，例如 Assert 方法用于根据判断条件输出调用堆栈，Write、WriteLine、WriteIf、

WriteLineIf 用于将调试信息输出, TraceSwitch 提供了动态控制跟踪调试的办法。二者的主要区别是发布版本时, 默认情况下是不包含 Debug 语句的, 但是包括 Trace 语句。

```
public static void Main()
{
    Int32 num = 100;

    Debug.Listeners.Add(new TextWriterTraceListener(Console.Out));
    Debug.Assert(num > 100, "num 不大于");
    Debug.WriteLine(num);
}
```

- EventLog, 提供了写入、读取、创建和删除事件日志的方法, 例如:

```
public static void Main()
{
    // 创建 EventLog 实例
    EventLog myLog = new EventLog("MyLog", ".", "AnyLog");
    // 写入日志事件
    myLog.WriteEntry("It's my log.", EventLogEntryType.Information, 1);
    // 读取日志事件
    foreach (EventLogEntry ele in myLog.Entries)
    {
        Console.WriteLine(ele.Message);
    }
}
```

- Stopwatch, 用于高精度检测运行时间。其中, Start 方法表示开始测量, Stop 表示停止测量, Reset 表示停止测量并重置为 0, 最后以 Elapsed 返回测量总时间。例如:

```
public static void Main()
{
    ArrayList al = new ArrayList(100000);

    // 创建并开始测量
    Stopwatch sw = Stopwatch.StartNew();
    for (Int32 i = 0; i < 100000; i++)
    {
        al.Add(i);
    }
    // 停止测量
    sw.Stop();
    // 返回测试总时间
    Console.WriteLine(sw.ElapsedMilliseconds);
}
```

10.4.4 System.Runtime.Serialization 和 System.Xml.Serialization

System.Runtime.Serialization 和 System.Xml.Serialization 主要用于完成序列化和反序列化功能。其中 System.Runtime.Serialization 属于 System.Runtime 的次级命名空间, 而 System.Xml.Serialization 属于 System.Xml 的次级命名空间。

所谓序列化就是指将对象状态转换为可存储或可传输格式的过程, 而反序列化则是从物理介质或流上获取数据, 并还原为对象的过程。其目的就是将对象持久化, 而持久化的对象就可以轻松实现永久存储和值封

送。.NET提供了两种强大的序列化技术：一种是二进制序列化，一种深序列化方式(Deep Serialization)，用于将对象的公有字段、私有字段等转换为字节流，进而写入数据流。这些功能主要由System.Runtime.Serialization命名空间及其次级命名空间的类来提供；另一种是XML序列化，一种浅序列化方式(Shallow Serialization)，仅序列化对象的公共属性和字段，在通过Web传输数据的时候，这种方式符合XML标准化的开放规则，这些功能主要由System.Xml.Serialization命名空间的类来提供。另外，使一个类可序列化的最简单办法就是给类加上SerializableAttribute特性。

- System.Runtime.Serialization，用于实现二进制序列化，其中IFormatter接口提供了对象序列化的功能，而BinaryFormatter和SoapFormatter分别用于将对象序列化为二进制格式和Soap格式。例如：

首先定义一个要序列化对象的类：

```
[Serializable]
public class UserInfo
{
    public string Name;
    public Int32 Age;
    public bool IsVIP;

    public UserInfo(string name, Int32 age, bool isVip)
    {
        Name = name;
        Age = age;
        IsVIP = isVip;
    }
}
```

然后，实现一个序列化的过程方法：

```
public static void BinarySerialize(UserInfo user)
{
    FileStream fs = new FileStream("MySerialze.bin", FileMode.Create);
    BinaryFormatter formatter = new BinaryFormatter();
    //执行二进制序列化
    formatter.Serialize(fs, user);
    fs.Close();
}
```

再是一个反序列化的过程方法：

```
public static UserInfo BinaryDeserialize()
{
    FileStream fs = new FileStream("MySerialze.bin", FileMode.Open, FileAccess.Read,
FileShare.Read);
    BinaryFormatter formatter = new BinaryFormatter();
    //执行二进制反序列化
    UserInfo user = formatter.Deserialize(fs) as UserInfo;
    fs.Close();

    return user;
}
```

最后，实现一个简单的测试过程：

```
public static void Main()
```

```

    UserInfo user = new UserInfo("小雨", 26, true);
    //执行序列化
    BinarySerialize(user);
    //执行反序列化
    UserInfo user2 = BinaryDeserialize();
    Console.WriteLine("Name:{0}\nAge:{1}\nVip:{2}", user2.Name, user2.Age, user2.IsVIP);
}

```

我们可以从结果了解到在执行目录里会生成一个 MySerialze.bin 文件，同时应用 SoapFormatter 也可以完成类似的执行过程，此不赘述。

- System.Xml.Serialization，用于实现 XML 序列化操作，其中最重要的类型为 XmlSerializer 类，它提供了相应的 Serialize 方法和 Deserialize 方法来完成 XML 格式的序列化和反序列化过程。值得注意的是，XML 序列化不能完成方法、索引器、私有字段或只读属性的序列化，而必须由二进制序列化来完成。

在此，我们仍借助于上例中类 UserInfo 的对象序列化，来举例说明 XML 序列化的简单过程：

```

public static void Main()
{
    UserInfo user = new UserInfo("小王", 27, false);
    XmlSerializer serializer = new XmlSerializer(typeof(UserInfo));

    //执行序列化过程
    StreamWriter sw = new StreamWriter("MySerialze.xml");
    serializer.Serialize(sw, user);
    sw.Close();

    //执行反序列化过程
    FileStream fs = new FileStream("MySerialze.xml", FileMode.Open);
    UserInfo user2 = serializer.Deserialize(fs) as UserInfo;

    Console.WriteLine("Name:{0}\nAge:{1}\nVip:{2}", user2.Name, user2.Age, user2.IsVIP);
}

```

如果我们给 UserInfo 类添加一个私有字段，那么在生成的 MySerialze.xml 文件中，则找不到该字段的痕迹，由此证明 XML 序列化是一种浅序列化方式。

在此，我们仅提供了最简单的序列化和反序列化过程，对于 XML 序列化来说，还有很多值得研究的主题需要探索。XML 基于开放的标准，使得 XML 序列化方式尤为重要，例如 Web Services 的数据传输正是通过 XML 序列化来实现的。

10.4.5 结论

对于.NET 框架类库的研究，是个系统性、长期性和实践性的过程。但是，从全局和结构上来理解 FCL，不失为一种可取的研究方式。鉴于 FCL 的学习曲线是如此陡峭，我们研究的步伐和方法，就不应该停止，事实上对类库的学习与理解将伴随每个开发者的认知过程。

参考文献

David Chappell, Understanding .NET

Christian Nagel, Bill Evjen, Jay Glynn, Professional C# 2005

Krzysztof Cwalina, Brad Abrams, Framework Design Guidelines: Conventions, Idioms, and Patterns for Reusable .NET Libraries

袁剑, .NET 框架与多线程,

<http://www.microsoft.com/china/community/Column/94.mspx>

第4部分

拾遗——.NET 也有春天

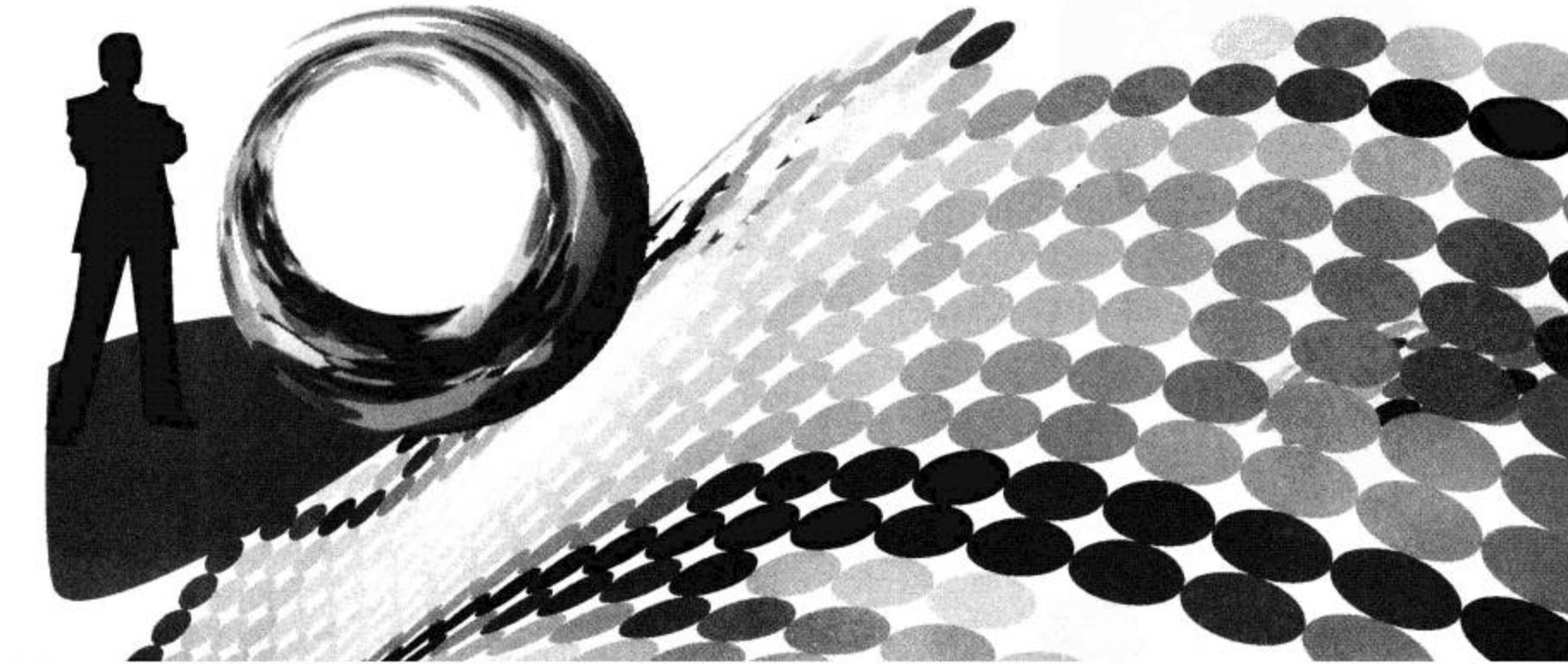
.NET 技术还在不断地发展与完善，技术探讨将不断迎来一个又一个春天，在这一部分中我们将重点讨论.NET 春天中的两个参天大树：泛型和安全性。泛型部分以简单的手法带领读者轻松进入这一领域；安全性的讨论则将对.NET 的安全策略和安全方法进行一次洗礼，综合阐述如何构建更安全的软件系统方法。我们以拾遗的轻松心情进行探讨，在把握基本概念的基础上，探讨其应用策略与注意事项。

本部分主要包括：

- 第 11 章 接触泛型
- 第 12 章 如此安全性

第 11 章 接触泛型

11.1	追溯泛型 / 403
11.1.1	引言 / 403
11.1.2	推进思维，为什么泛型 / 403
11.1.3	解析泛型——运行时本质 / 405
11.1.4	结论 / 406
11.2	了解泛型 / 406
11.2.1	引言 / 406
11.2.2	领略泛型——基础概要 / 406
11.2.3	典型.NET 泛型类 / 409
11.2.4	基础规则 / 410
11.2.5	结论 / 411



11.3	深入泛型 / 411
11.3.1	引言 / 411
11.3.2	泛型方法 / 411
11.3.3	泛型接口 / 413
11.3.4	泛型委托 / 415
11.3.5	结论 / 415
11.4	实践泛型 / 416
11.4.1	引言 / 416
11.4.2	最佳实践 / 416
11.4.3	结论 / 421
	参考文献 / 421

11.1 追溯泛型

本节将介绍以下内容：

- 泛型历史介绍
- 为何而泛型
- 泛型类型实例化

11.1.1 引言

在.NET 2.0中，可以称为革命性壮举的，就是引入激动人心的特性——泛型。作为.NET 2.0的新特性，其实泛型技术并非新鲜出炉，早在多年以前C++中就实现了类似于泛型的模板（Template），不过.NET泛型和C++模板还是有所区别，此为后话。

总之，.NET泛型是CLR和高级语言共同支持的一种全新的结构，实现了一种将类型抽象化的通用处理方式。在泛型机制中，我们不再为特定的类型而编码，取而代之的是一种通用的编码方式，因此泛型本质上就是一种代码重用。这种代码重用并非面向对象编程中通过继承、聚合、多态等方式实现；而是实现为一般化、可重用的算法抽象，但在执行效率上与执行特定类型相同。

从这里开始记录泛型，本章带你进入强大而精彩的抽象世界，领略泛型编程的魅力。

11.1.2 推进思维，为什么泛型

认识泛型，还是从一个基础示例开始，在这个基础示例的演化过程中，对于泛型的认识将会一步步深入，对于算法重用的理解也会一次次清晰。最后形成的泛型编程概念，也就更有说服力。

在泛型中，最重要的应用便是集合类，因此我们的示例也模拟一个简化的集合类：

```
//初始版本的自定义集合--仅能管理 string 类型
class MyArray
{
    //自定义集合的目标元素
    private string[] _items;
    private Int32 _size = 0;

    //实现元素增加功能
    public void Add(string item) { //从略.....}
    //实现索引器
    public string this[Int32 index] { //从略.....}
    //实现构造器---部分省略
}
```

对于上述示例，具体实现代码从略（详细的分析可见该示例演化的最后一个泛型类的实现），可以有如下应用：

```
public static void Main()
{
    MyArray myArr = new MyArray();
    myArr.Add("Hello, world");
```

```
//myArr.Add(123); //类型不兼容，导致编译错误
Console.WriteLine(myArr[0]);
}
```

显然，由代码分析可知，自定义的 MyArray 类型简直太简陋了，只能用于 string 类型对象的管理，对于其他类型的不速之客，编译器毫不客气的给出错误信息，这样的集合类实在太扫兴了。因此，我们有了下面的修改版本，用于实现能够应付灵活需求的集合类：

```
//修改版本的自定义集合--能够管理任何类型
class MyArray
{
    private object[] _items;
    private Int32 _size = 0;

    public void Add(object item) { //从略.....}
    public object this[Int32 index] { //从略.....}
    //实现构造器---部分省略
}
```

改进之后的 MyArray 类型，以 object 作为集合元素的基础类型，这样做好处是 object 是一切类型的基类，因为一切类型都可以隐式的转换为 object 类型，从而使得修改后的 MyArray 可以接受任何类型的元素，我们可以有如下的应用：

```
public static void Main()
{
    MyArray myArr = new MyArray();

    //可以接受任何类型的元素
    myArr.Add("Hello, world");
    myArr.Add(123);
    myArr.Add(new MyArray());

    Console.WriteLine(myArr[2]);
}
```

一切看起来已经“完美无缺”，修改后的 MyArray 可以灵活的应对任何类型的管理，然而深入分析则会发现有些问题仍然存在，继续深究终有隐藏的 Bug 大白于天：

```
//潜在的类型安全问题
Int32 myItem = (Int32)myArr[0];
Console.WriteLine(myItem);
```

上述代码可以安全的通过编译时，但是会导致严重的运行时问题而抛出类型无效转换的异常。由此可见，以 Object 类型作为“通用”类型，虽然某种程度上可以实现灵活的类型控制，然而这种基于类型转换的实现方式，存在潜在的类型安全问题，因此并不值得推荐，在 6.4 节“性能优化的多方探讨”中强调以泛型集合代替非泛型集合，正是这个道理。

同时，这种方式还有讨厌的性能问题：值类型元素 Add 到集合时，必然存在装箱操作；而将元素赋值给一个值类型变量时，又会发生相应的拆箱操作。一来一去的性能损失，在操作大量元素时，显得格外醒目。

正是由于这些问题的出现，才使得泛型机制粉墨登场，我们重新实现自定义的 MyArray 类型，体会泛型带来的全新体验：

```
//最终版本的自定义集合--能够安全管理任何类型
class MyArray<T>
{
```

```

//自定义集合的目标元素
private T[] _items;
//集合大小计数
private Int32 _size = 0;

//实现元素增加功能
public void Add(T item)
{
    _items[_size] = item;
    _size++;
}

//实现索引器
public T this[Int32 index]
{
    get { return _items[index]; }
    set { _items[index] = value; }
}

//实现构造器---在此简单处理，取消动态增加
public MyArray()
{
    _items = new T[100];
}

```

完成一个完整的自定义泛型类，继续进行必要的测试：

```

public static void Main()
{
    MyArray<int> myIntArr = new MyArray<int>();
    myIntArr.Add(123);
    Console.WriteLine(myIntArr[0]);
    //保证类型安全，无法通过编译检测
    //string myItem = (string)myIntArr[0];
    MyArray<string> myStringArr = new MyArray<string>();
    myStringArr.Add("222");
    Console.WriteLine(myStringArr[0]);
}

```

OK。这次才算是完美实现，经过简单的测试，我们有理由相信：自定义的泛型集合类，能够轻松实现对任意类型的管理和操作，并且不会造成类型转换、装箱与拆箱引起的性能问题，其执行效率和特定的 Int32 型、String 型相同；而且泛型保证了类型的绝对安全，不会引起潜在的类型转换问题。

11.1.3 解析泛型——运行时本质

那么，泛型在算法上的重用是如何实现的？追本溯源，从本质入手来探求究竟，CLR 2.0 中实现了专门的 IL 指令支持泛型操作，具体的编译过程为：初次编译时，首先生成 IL 代码和元数据，T 只是作为类型占位符，不进行泛型类型的实例化；在进行 JIT 编译时，将以实际类型替换 IL 代码和元数据中的 T 占位符，并将其转换为本地代码，下一次对该泛型类型的引用将使用相同的本地代码。

对于值类型和引用类型参数，泛型类型实例化有所不同：类型参数为值类型时，JIT 编译器为不同的值类型创建不同的本地代码；类型参数为引用类型时，则共享本地代码的单个副本，这源于引用类型变量都是指向托管堆的引用指针，而对于指针完全可以使用相同的方式来操作。这是 CLR 基于代码优化的考虑，以缓解代码爆炸的压力。

因此，对于我们示例中的 myIntArr 和 myStringArr 来说，在实例化时将生成不同的本地代码，从而产生不同的执行过程，这正是泛型实现代码重用的秘密所在。由此可见，.NET 泛型由 CLR 运行时支持，真正的泛型实例化发生在 JIT 编译时，生成不同的本地代码，由此可以轻松实现支持 CLR 各个语言间的泛型互操作。

11.1.4 结论

由此可见，泛型编程将类型抽象化处理，并在此基础上实现算法规则，将相同的算法运用于不同的类型，从而实现了一种算法上的复用。在 MyArray<T> 中定义好一个 Add 算法，而不必关心要操作的实际类型，事实上这种算法可以应用于任何类型。指定不同的具体类型，就会执行 Add 算法的实现逻辑，并产生相应的结果，而且能够保证绝对的类型安全。这就是泛型的抽象机制和强大威力，深入泛型世界正是逐步建立起这种泛型编程方式的思维演化。

11.2 了解泛型

本节将介绍以下内容：

- 泛型基础讨论
- .NET 泛型类
- 泛型规则

11.2.1 引言

本节将全面讲述泛型的基础内容和基础知识，类型参数、泛型约束、泛型重载等一系列的概念将给予对泛型更全面的认知和理解，同时简洁明了的示例会增加更多对于泛型的思考和体会。

在此基础上，来认识.NET 框架类库中的典型泛型类，将更有助于理解泛型编程的抽象机制，最后再对泛型的特点和规则做以总结，这正是了解泛型最好的开始。

11.2.2 领略泛型——基础概要

1. 泛型基础

再次回到上文最后实现泛型集合类示例，虽然已经领略了泛型的强大能力，但是还有很多基本的概念，需要一一揭开，我们还是以熟悉的示例来解释：

```
class MyArray<T>
{
    // 定义字段
    private T[] _items;

    // 定义方法
    public void Add(T item)
    {
```

```

}

// 定义索引器
public T this[Int32 index]
{
}

// 定义构造器
public MyArray()
{
}
}

```

我们会发现，定义一个泛型类和定义非泛型类没有太大的区别，而主要的不同在于：类型参数化。类型定义时，将指定类型形参（Type Parameter，通常以 T 表示），紧随类名，并包含在<>符号内。对于这种具有类型参数的类型，我们称其为：开放式类型，CLR 禁止为开放式类型构造任何实例；而对于为类型参数传入实际参数的类型，被称为：封闭式类型，CLR 允许为封闭式类型构造实例。例如在创建泛型实例时，将指定实际的数据类型：

```
MyArray<Int32> myArr = new MyArray<Int32>();
```

上述代码将创建一个泛型值类型对象 myArr，并调用构造器完成对象初始化，Int32 作为 T 的类型实参（Type Argument），在 JIT 编译时所有 T 将被替换为 Int32，因此_items 内部数组中将保存 Int32 型的数组元素，对于

```
myArr.Add("ABC");
```

这样的操作，显然会导致类型检查错误。因此，类型形参 T 只是一个占位符，在实际使用时将被替换为实际的数据类型。

根据实参的不同，泛型可以分为：泛型引用类型和泛型值类型。其区别已经在 11.1 节“追溯泛型”中有所交代。

泛型类型的声明也可以包括多个类型形参，例如：

```
System.Collections.Generic.Dictionary< TKey, TValue >
```

通常情况下，将类型参数命名为 T 或者以 T 开头的有意义的参数名是值得推荐的命名方式。

对于自定义泛型类，进行必要的修改，在此增加一个参数化类型字段，如下：

```

class MyArray<T>
{
    // 增加新的字段
    public T myData;

    public MyArray()
    {
        // 初始化默认值
        myData = default(T);
    }
}

```

关于参数化类型 T 的默认值，应该如何做出选择：T 有可能是值类型，因此默认值应为 0；T 也有可能

是引用类型，因此默认值为 null。权衡这一问题的办法，正是修改后的泛型类实现中采用的解决办法：default 关键字。它对于值类型会返回 0，对于引用类型会返回 null，而对于结构类型则初始化其成员为相应的默认值（0 或 null，视具体类型而定）。

2. 约束

约束，指在定义泛型类时，对于能够用于实例化类型参数的类型所做的限制，这种限制能够保证类型参数局限在一定的目标范围，以实现在泛型类中的方法或者运算符能够得到类型参数的支持而不会引起其他问题。这种限制正是通过一个或者多个约束来获得，而约束则是通过 where 子句来实现的，多个约束之间以逗号隔开。通常情况下，在设计泛型类或泛型方法时，如果要实现一个 Object 类不支持的任何方法，则需要对类型参数实现约束。具体为：

```
class MyArray<T> where T: Student, new()
{
}
```

为 MyArray 添加了约束 Student 和 new()，意指 MyArray 只能管理 Student 类型及其派生类型对象，同时必须实现一个公共的无参的构造函数。

约束主要包括：构造器约束、值类型约束、引用类型约束、基类约束、接口约束和裸类型约束，具体情况为：

- T: new()，表示类型参数必须具有公共无参构造函数，有多个约束存在时，必须将 new() 约束置于最后。
- T: struct，表示类型参数必须是值类型。
- T: class，表示类型参数必须是引用类型，显然 class 和 struct 不能同时指定。
- T: 基类名，表示类型参数必须是基类及其派生类型，例如 T: Student。但是不能既指定约束基类，又指定 class。
- T: 接口名，表示类型参数必须是指定的接口或者实现了该接口的接口，可以定义多个接口约束，同时约束接口也可以是泛型的，例如 List 泛型的定义为：

```
public class List<T> : IList<T>, ICollection<T>, IEnumerable<T>, IList, ICollection, Ienumerable
```

- T: U，表示为 T 提供的类型参数必须是 U 提供的类型参数或派生于 U 提供的类型参数，称为裸类型约束，而这种约束并不常见。

泛型约束，提供了更强的类型检查，并且更加有效的参数类型有时能够减少不必要的类型转换而改进性能。但是另一方面，强制性的约束可能使得算法重用的功能受到限制。

3. 泛型类继承

泛型类本质上仍然是一个类，所以仍然具有类所具有的基本特性，因此一个泛型类仍然继承于某个父类，例如所有的泛型类最终都会派生于 System.Object 类。在.NET 框架类库中，这种继承关系也广泛存在于泛型类的定义中，例如：

```
internal class GenericComparer<T> : Comparer<T> where T: IComparable<T>
```

因为本节实现的 MyArray 是个泛型集合类，为了省力起见，为 MyArray 找一个合适的基类来继承将是个不错的主意，所以再次修改 MyArray 的定义为：

```
class MyArray<T>: ArrayList
{}
```

然后，应用新修改的 MyArray 泛型类，构造一个实例对象并完成一定的测试：

```
public static void Main()
{
    MyArray<Int32> myIntArr = new MyArray<Int32>();
    myIntArr.Add(123);
}
```

从 Visual Studio 2005 的智能感知可知，myIntArr 将具有两个重载方法，一个是在 MyArray 泛型类定义的方法 MyArray<T>.add(T item)，如图 11-1 所示。

```
public static void Main()
{
    MyArray<Int32> myIntArr = new MyArray<Int32>();
    myIntArr.Add(123);
} ▲1 (总数 2) ▾ void MyArray<int>.Add (int item)
```

图 11-1 Visual Studio 2005 的智能感知

另一个是继承于 ArrayList 类的方法 ArrayList.Add(object value)，如图 11-2 所示。

```
public static void Main()
{
    MyArray<Int32> myIntArr = new MyArray<Int32>();
    myIntArr.Add(123);
} ▲2 (总数 2) ▾ int ArrayList.Add (object value)  
value: The System.Object to be added to the end of the System.Collections.ArrayList. The value can be null.
```

图 11-2 Visual Studio 2005 的智能感知

实际上，根据本章第一节对泛型实例化运行机制的分析，实例创建时将在 CLR 创建一个新的类型。因此泛型类的继承，实际是新类型对该基类的继承，例如示例中 MyArray<T> 派生于 ArrayList 类，那么新的类型 MyArray<Int32>也就派生于 ArrayList，因此 myIntArr 对象也就具有了 ArrayList 类的相应方法、属性和字段。

11.2.3 典型.NET 泛型类

.NET 框架类库中，System.Collections.Generic 和 System.Collections.ObjectModel 命名空间中，分别定义了大量的泛型类和泛型接口，这些泛型类多为集合类，因为泛型最大的应用正体现于在集合中对不同类型对象的管理。在 8.9 节“集合通论”中，我们对此有过详细的讨论和分析，其中最为典型的就是 List<T> 泛型类，此不赘述。

在表 11-1 中，将列出.NET 框架类库中提供的最常用和最基本的泛型类或泛型接口，并对其功能做以简单介绍。

表 11-1 典型泛型类

泛型类	说明
List<T>	对应于 ArrayList 集合类，可以动态调整集合容量，通过索引方式访问对象，支持排序、搜索和其他常见操作，详细的论述请参见 8.9 节“集合通论”。
SortedList< TKey, TValue >	对应于 SortedList 集合类，表示 Key/Value 对集合，类似于 SortedDictionary< TKey, TValue > 集合类，而 SortedList 在内存上更有优势

续表

泛型类	说 明
Queue<T>	对应于 Queue 集合类，是一种先进先出的集合类，常应用于顺序存储处理
Stack<T>	对应于 Stack 集合类，是一种后进先出集合类
Collection<T>	对应于 CollectionBase 集合类，是用于自定义泛型集合的基类，提供了受保护的方法来实现定制泛型集合的行为。Collection<T>的实例是可修改的，.NET 还为其提供了一个只读版本 ReadOnlyCollection<T>
Dictionary< TKey, TValue >	对应于 Hashtable 集合类，表示 Key/Value 对的集合类，Key 必须是唯一的，其元素类型既不是 Key 的类型，也不是 Value 的类型，而是 KeyValuePair 类型。

11.2.4 基础规则

关于泛型的基本内容，有以下规则值得总结：

- 泛型方法可以存在于泛型类，也可以存在于非泛型类中。
- 允许创建泛型类、泛型接口、泛型委托，但不允许创建泛型枚举类型。
- 在代码中，能使用类型的任何位置都可以使用类型参数 T，T 只是作为一个类型占位符，而非实际的类型。
- 类型参数，应为 T 或以 T 开头的有意义标识符，这是良好的编程规范。
- 泛型被广泛应用于集合类中。在 System.Collections.Generic 和 System.Collections.ObjectModel 命名空间中定义了很多的泛型集合，我们推荐以泛型集合来代替对应的集合类，详见 8.9 节“集合通论”的描述。
- .NET 泛型编译是一种两段式编译：首先编译为 IL 和元数据，然后在 JIT 编译时，获取方法的 IL，并以实际类型替换类型参数，并根据指定类型来创建本地代码。如果实例化的类型参数相同，则不同的泛型实例将共享本地代码，这种优化策略能有效缓解泛型的代码爆炸问题。
- 由于类型参数在编译期并不确定，所以泛型方法重载需要特别留意可能发生的调用混淆。
- .NET 泛型受 CLR 运行时支持；而 C++ 的泛型是一种编译时的模板机制。二者之间有较大差别：.NET 泛型机制在复杂性和灵活性方面较 C++ 模板弱，而运行时支持，为实例化对象保留了泛型类型信息，在运行时可以通过反射来查询。
- .NET 泛型可以应用反射技术，应用大量元数据信息可以保证泛型反射的实现。.NET 2.0 的 Type 类增加了几个新成员用以启用泛型类型的运行时信息，例如 MakeGenericType 方法就用于返回构造泛型类型的 Type 对象。

除了必要的规则总结，认识.NET 泛型的特点同样重要，主要包括：

- 类型安全性。类型安全是泛型最为显著的一个特性，指定的实参类型将替换所有的类型参数，实际会在 CLR 定义一个类型安全的新类型，只有与指定类型兼容的对象才能应用相应的算法，否则将导致编译错误。
- 更好的性能。对于泛型值类型，不会产生以 Object 作为通用类型进行转换的装箱或拆箱问题，值类型实例将以传值方式进行传递，而不会发生任何类型转换。尤其在泛型集合类中，对于性能的节约是显而易见的。
- 优雅的代码实现。泛型就是一个算法模板，应用于多少种类型就有多少种重用，类型安全减少类型转换的发生次数，对于代码的维护更加简洁、优雅。

11.2.5 结论

泛型基础的了解，是深入泛型的根本。从规则总结中，我们认识到关于泛型还有很多问题值得探索，例如泛型中静态字段将如何共享，如何以反射机制获取泛型类型信息等。带着这些问题来寻找答案，对于技术和本质的理解就会层层加深，泛型应用更是如此，一个激动人心的技术需要逐步建立起对其编程思维的认知。

11.3 深入泛型

本节将介绍以下内容：

- 泛型方法
- 泛型接口
- 泛型委托

11.3.1 引言

本节的重点是论述泛型概念中的泛型方法、泛型接口和泛型委托，作为对基础内容的有效补充，这些内容不可或缺。其中，泛型方法提供了更灵活的控制方式，对于类型参数引起的方法重载应该给予关注；泛型接口在类型安全性和性能上，同样具有非同寻常的意义；泛型委托则能有效避免值类型实例传入回调方法时的装箱处理。

11.3.2 泛型方法

泛型方法，提供了更加多变的灵活性。泛型方法可以存在于泛型类，也可以存在于非泛型类中。你可以将类型参数作为某个方法的参数、返回值或者局部变量，该类型参数可能并不被整个类所需要，而更明确的用于某个方法，例如：

```
//实现泛型方法
public void ShowInfo<TData>(TData data)
{
    Console.WriteLine(data.ToString());
}

//实现非泛型方法
public void ShowInfo()
{
    Console.WriteLine(myData.ToString());
}
public void ShowInfo(string str)
{
    Console.WriteLine(str);
}
```

1. 方法重载

由此可见，泛型方法可以以类型参数形成重载。对于泛型方法来说在客户端调用，将会有更灵活的控制方式，例如：

```
public static void Main()
{
    MyArray<Student> myArr = new MyArray<Student>();
    myArr.myData = new Student();
    myArr.ShowInfo();
    //泛型方法具有更灵活的控制
    myArr.ShowInfo<CollegeStudent>(new CollegeStudent());
    myArr.ShowInfo<string>("Hello, world.");
}
```

然而泛型方法重载，存在值得注意的问题：由于类型参数在编译期并不确定，所以重载检查是在实例方法被调用时才发生，而不是在泛型类本身编译时检查，例如下面的泛型类可以安全通过编译：

```
class MyArray<T>
{
    public void ShowInfo<TA, TB>(TA a, TB b)
    {
    }

    public void ShowInfo<TB, TA>(TA a, TB b)
    {
    }
}
```

然而在方法调用时，将会给出调用不明确的编译错误：

```
public static void Main()
{
    MyArray<Student> myArr = new MyArray<Student>();
    myArr.ShowInfo<Student, Student>(new Student(), new Student());
}
```

因此，泛型方法重载应该引起足够的重视，同时一般方法与泛型方法具有相同签名时，将优先调用一般方法，其原因是：类型推断。

2. 类型推断

泛型方法的调用，可以以更简洁的方式来实现，例如：

```
//实现更简洁的方式来调用
myArr.ShowInfo(new CollegeStudent());
myArr.ShowInfo("Hello, world.");
```

这种机制由编译器支持，称为：类型推断。表示编译器在调用时根据方法参数来推断类型参数，从而决定要使用的泛型方法。因此，这种方式不适用于没有参数的方法，同时编译器会优先选择显式的匹配调用，例如：

```
myArr.ShowInfo("Hello, world.");
```

将会调用非泛型方法 ShowInfo(string str)，而非泛型方法 ShowInfo<string>(TData data)，除非以显式的调用方式来实现：

```
myArr.ShowInfo<string>("Hello, world.");
```

3. 泛型方法约束

泛型方法也可以启用约束，其作用仍然是对类型参数加以限制，例如：

```
//实现泛型方法
public void ShowInfo<TData>(TData data) where TData: Student
{
    Console.WriteLine(data.ToString());
}
```

泛型方法启用约束之后，以下执行将无法通过编译：

```
myArr.ShowInfo<string>("Hello, world.");
```

如果泛型方法的类型参数和包含类的类型参数相同，将在方法内隐藏外部类型参数，同时外部参数启用的约束将对内部参数无效，例如上例中可以实现 `ShowInfo<string>` 方法的调用。因此，编译器将给出警告信息，所以建议为泛型方法提供与包含类不同的类型参数标识符，例如：

```
class MyArray<T> where T: Student, new()
{
    //类型参数标识符相同，将给出警告信息
    public void ShowInfo<T>(T t)
    {
    }
}
```

最后，必须明确的是，泛型机制只支持泛型方法，而不支持在其他成员的声明上包含类型参数，例如属性、事件、构造器、索引器等。而这些成员本身可以包含在泛型类中，并操作泛型类的类型参数，示例 `MyArray<T>` 中就包含了构造器和索引器的定义。

11.3.3 泛型接口

CLR 同样提供了对泛型接口的支持，在.NET 集合类中就实现了多个泛型接口，例如 `IList<T>` 泛型接口的实现可以表示为：

```
public interface IList<T> : ICollection<T>, IEnumerable<T>, IComparable<T>
{
    int IndexOf(T item);
    void Insert(int index, T item);
    void RemoveAt(int index);

    T this[int index] { get; set; }
}
```

在.NET 框架类库中的泛型接口，还包括如表 11-2 所示的泛型接口。

表 11-2 典型的泛型接口

泛型接口	说 明
IList<T>	所有泛型列表的基接口，允许以索引单独访问一组对象，定义了 <code>IndexOf</code> 、 <code>Insert</code> 、 <code>RemoveAt()</code> 方法
ICollection<T>	定义了操作泛型集合的常用方法，定义了 <code>Add</code> 、 <code>Clear</code> 、 <code>Contains</code> 、 <code>CopyTo</code> 、 <code>Remove</code> 等常见方法和 <code>Count</code> 、 <code>IsReadOnly</code> 属性
IComparable<T>	定义了通用的比较方法 <code>CompareTo</code> ，为排序实例创建特定的比较方法
IComparer<T>	定义了为集合中元素排序的方法 <code>Compare</code> ，支持排序比较
IEnumerable<T>	公开枚举数，支持 <code>foreach</code> 语义。方法 <code>GetEnumerator</code> 用于返回一个 <code>IEnumerator<T></code> 枚举
IEnumerator<T>	所有泛型枚举数的基接口，用于支持在泛型集合上的迭代
IDictionary< TKey, TValue >	Key/Value 对型泛型集合的基接口，定义用于操作 Key 和 Value 的 <code>Add</code> 、 <code>Remove</code> 和 <code>TryGetValue</code> 等方法

由 IList 的定义可知，类间的继承规则同样适用于泛型接口。同时一个泛型接口，可接受多个类型参数，例如：

```
public interface IDictionary<TKey, TValue>
```

在类型安全和性能优化方面，泛型接口同样带来惊喜，因此泛型接口是值得深入研究的。

下面，我们再次修改 MyArray<T> 泛型类，使其实现一个 IComparable<T> 接口，具体细节为：

```
class MyArray<T> : IComparable<T>
{
    //省略其他实现.....
    public T _value = default(T);

    //实现 IComparable<T>方法
    public Int32 CompareTo(T other)
    {
        IComparable<T> tmp = (IComparable<T>) _value;
        return tmp.CompareTo(other);
    }
}
```

在上述实现中，MyArray<T> 类实现了 IComparable<T> 接口，从而在客户端调用，可以有如下方式：

```
public static void Main()
{
    MyArray<Int32> myArr = new MyArray<int>();
    myArr._value = 100;
    Console.WriteLine(myArr.CompareTo(100));

    MyArray<string> myStrArr = new MyArray<string>();
    myStrArr._value = "ABC";
    Console.WriteLine(myStrArr.CompareTo("abc"));
}
```

从输出结果来看，实现了 IComparable<T> 的 MyArray<T> 类能够实现接口方法的重用，然而仔细探究你会发现有时这种操作并不奏效，因为前提是在实现的方法中有如下的操作：

```
IComparable<T> tmp = (IComparable<T>) _value;
```

也就是实例化的指定类型，必须与 IComparable<T> 兼容，通过在此顺利完成类型转换，而 System.Int32 和 System.String 类型正好都实现了该接口，因此我们的执行安然无恙。而如果有如下的执行操作，将导致 InvalidCastException 异常：

```
MyArray<Student> myStudentArray = new MyArray<Student>();
myStudentArray._value = new Student();
Console.WriteLine(myStudentArray.CompareTo(new Student()));
```

这种问题是泛型接口必须关注和了解的重点，通常解决的办法是为 MyArray<T> 实现指定了具体类型参数的接口，从而保证类型的安全性，例如：

```
class MyArray<T> : IComparable<Int32>, IComparable<String>
{
    public Int32 _intValue = 0;
    public String _strValue = null;

    //实现 IComparable<Int32> 的接口方法
    public Int32 CompareTo(Int32 other)
```

```

    {
        return _intValue.CompareTo(other);
    }

    //实现 Comparable<String>的接口方法
    public Int32 CompareTo(String other)
    {
        return _strValue.CompareTo(other);
    }
}

```

因此，可以为你的类型实现一个指定了具体类型实参的泛型接口，也可以实现一个未指定类型参数的泛型接口，而权衡的关键应视具体情况而定，关键是保证类型的安全性。当然，也可以通过约束或者为 Student 类实现 IComparable<T>来解决上述问题，而泛型接口规则这些要素值得我们注意。

11.3.4 泛型委托

泛型委托支持在返回值和参数上应用类型参数，泛型委托能够有效避免值类型实例传给一个回调方法时的装箱处理。在.NET 集合类中就定义了很多的泛型委托，例如：Converter<TInput, TOutput>用于实现两种类型的转换，Comparison<T>用于比较同一类型的两个对象，常用于集合项的比较；EventHandler<TEEventArgs>则是一个用于处理事件的委托。

下面我们实现一个泛型委托的简单示例，以对此做进一步的了解：

```

class GenericDelegate
{
    //首先声明一个泛型委托
    public delegate string MyGenericDelegate<T>(T t);

    //实现两个测试方法
    public static string GetPoint(Point p)
    {
        return String.Format("The location is ({0}, {1}).", p.X, p.Y);
    }
    public static string GetMsg(string txt)
    {
        return txt;
    }

    //实现测试代码
    public static void Main()
    {
        MyGenericDelegate<string> myStrDel = new MyGenericDelegate<string> (GetMsg);
        Console.WriteLine(myStrDel("Hello, world."));

        MyGenericDelegate<Point> myPointDel = new MyGenericDelegate<Point> (GetPoint);
        Console.WriteLine(myPointDel(new Point(100, 200)));
    }
}

```

11.3.5 结论

继续深入对泛型基本内容的探讨，将伴随着不断的深入而越发全面，最终建立起对整个泛型技术的了解。

CLR 2.0 为支持泛型技术进行了大刀阔斧的修改，创建了新的 IL 指令来识别类型参数，修改元数据表格式，修改 JIT 编译器，创建新的反射成员等。通过对泛型技术的洗礼，我们了解了.NET 泛型的基本面貌。

.NET 提供了 CLR 运行时支持的泛型技术，而对于泛型编程思维的修炼则需要不断的应用技术武器来充实，结合面向对象编程的思维，来设计具有较高层次的复用和抽象，这才是应用泛型的根本。

11.4 实践泛型

本节将介绍以下内容：

- 泛型最佳实践
- 泛型应用误区

11.4.1 引言

泛型是算法重用的基础，泛型是类型安全的解药，泛型是既爱又难以爱的设计精灵。泛型充满了故事与不解，而本文就是书写泛型实践、讲述不解之惑的白皮书。

下面，再次以条款的形式，来讲述泛型实践的故事。

11.4.2 最佳实践

关于泛型的最佳实践，主要包括以下条款。

1. 尽可能应用泛型集合

泛型集合实现了对于数据集的类型安全操作，同时避免了装箱与拆箱的性能损失，因此在默认情况下，优先考虑泛型集合是不变的规则。

详见 8.9 节“集合通论”的讨论。

2. 通过约束保证类型参数的合法性

泛型约束，是减少不必要类型转换的有效制约手段，同时增强了类型检查的范围，有利于改善性能和可靠性。不过，约束带来算法重用在一定程度上受限，相较而言，可预见的约束带来的可靠性显得更加重要。

详见 11.2 节“了解泛型”的讨论。

3. 正确考虑泛型的协变和逆变

在.NET 4.0 中引入了支持泛型协变与逆变的新特性，为泛型通往更广阔层次的应用提供了基础支持，这不是简单的引入两个关键字：in 和 out，而是解决了一直以来泛型隐式类型转换的类型兼容问题，支撑了泛型在继承和多态上的延伸。

在 14.6 节“协变与逆变”中有详细的讨论。

4. 切勿使用过多的类型参数，一般不要超过 2 个类型参数

类型参数过多，会造成语义理解上的困难，尽可能将类型参数的个数控制在 2 个之内，从实践经验而言，是值得推荐的。更多的类型参数，将带来理解上和维护上的巨大成本，甚至为使用上带来很大障碍。

5. 为类型参数赋予有意义的命名

在 3.6 节“好代码和坏代码”中已经讨论了命名规则的重要意义，体现在泛型类型参数上，同样至关重要，有意义的类型参数将能更好地辅助对于泛型类型或者泛型方法的理解，以.NET 框架本身的类型定义来看：

```
public class ChannelFactory<TChannel> : ChannelFactory, IChannelFactory<TChannel>, IChannelFactory, ICommunicationObject
{
}
```

还有：

```
public interface IDictionary< TKey, TValue > : ICollection< KeyValuePair< TKey, TValue >>, IEnumerable< KeyValuePair< TKey, TValue >>, IEnumerable
{
}
```

当然，在自定义的泛型类型中，也最好遵循这一原则，尽量避免以单字母 T、U 或者 V 来命名。

同时类型参数命名，习惯上以 T 作为首字母命名。

6. 注意泛型方法造成的方法重载问题

基于类型推断原理，编译器在调用时才根据方法参数来推断类型参数的实际类型，因此会造成某种情况下的重载匹配优先选择问题：

```
public class PipelineBase<TMessage>
{
    public void Execute(TMessage message)
    {
    }

    public void Execute(IMessage message)
    {
    }
}
```

因此，必须非常明确类型推断的基本原则，才能保证客户端的正确调用：

```
class Program
{
    static void Main(string[] args)
    {
        PipelineBase<object> pipeline = new PipelineBase<object>();

        //调用泛型方法
        pipeline.Execute(1);
        //调用非泛型方法
        IMessage message = new Message();
        pipeline.Execute(message);
    }
}
```

关于类型推断，详见 11.3 节“深入泛型”的讨论。

7. 尽量使用 EventHandler<T>定义事件

EventHandler<T>本质上是一个封装了 EventArgs 的泛型委托，体现了观察者模式的应用实践，通过框架提供的统一模型，为定义事件带来巨大的便利，以一个邮件派发类为例：

```
public class MailDispatcher
{
    public event EventHandler<MailEventArgs> Sending;

    public void Send(EMail mail)
    {
        OnSending(mail, new MailEventArgs(mail));

        try
        {
            // Send mail...
        }
        catch (Exception ex)
        {
            // Error log...
        }
    }

    protected virtual void OnSending(EMail sender, MailEventArgs e)
    {
        EventHandler<MailEventArgs> handler = Sending;

        if (handler != null)
        {
            handler(sender, e);
        }
    }
}
```

Eventhandler<T>简化了事件定义的手续，提供了统一的编程模型，详见 9.7 节“一脉相承：委托、匿名方法和 Lambda 表达式”的讨论。

8. 通过 Nullable<T>处理值类型的可空情况

通过 Nullable<T>，为值类型赋予了“可空”特性，实现了统一方式处理值类型和引用类型在“空”值问题的编程模型。尤其是在数据库操作场合，当字段被设置为可空时，在数据层就可以非常自然地进行直接的数据赋值，否则需要进行可空判断：

```
static void Main()
{
    User model = new User();

    UserEntity entity = DB.Get(1);

    if (entity != null)
    {
        model.Id = entity.UserId;
        model.FriendId = entity.FriendId == null ? -1 : entity.FriendId.Value;
        model.Name = entity.Name;
    }

    //...
}
```

通过 Nullable<T>将能很好地解决可空判断问题，使得值类型可以合法地兼容来自实际数据系统的“可空”情况：

```
public class User
{
    public int Id { get; set; }
    //定义为可空类型
    public int? FriendId { get; set; }
    public string Name { get; set; }
}
```

从而，数据层的赋值变得更加自然：

```
model.FriendId = entity.FriendId;
```

详见 7.4 节“认识全面的 null”的讨论。

9. 通过继承 `IEnumerable<out T>` 公开枚举器

众所周知，`IEnumerable<T>` 是 LINQ to Object 的核心，实现了 `IEnumerable<T>` 的类型意味着公开了枚举器，也意味着可以通过 `foreach` 循环遍历，于是如果有自定义集合：

```
public class MailList : IEnumerable<EMail>
{
}
```

那么，就可以顺理成章地有如下应用：

```
static void Main()
{
    MailList list = new MailList();

    foreach (EMail mail in list)
    {
        //...
    }
}
```

否则，将引发编译时错误。详见 7.7 节“非主流关键字”的讨论。

10. 避免不必要的类型转换，为类型参数使用类型约束

为类型参数约束基类，是避免不必要的类型转换和类型安全的有效手段：

```
public class PipelineBase<TMessage> where TMessage : IMessage
{
    public void Execute(TMessage message)
    {
        IMessage temp = message;

        //...
    }
}
```

如果没有基类约束，需要进行类型的转换，通常考虑以 `as` 运算符进行转换判定。

详见 8.5 节“恩怨情仇：`is` 和 `as`”的讨论。

11. 避免在类型参数上加锁

`Monitor` 只能作用于引用类型，而泛型类型参数如果没有引用类型约束，则在编译期不能预判是否为值类型或者引用类型，如果类型参数为值类型，则会导致 `lock` 失效。因此，在泛型情况下，线程同步需要旗帜鲜明。

明地为类型参数添加引用类型约束:

```
public class ATFactory<TElement> where TElement : class
{
}
```

详见7.7节“非主流关键字”的讨论。

12. 泛型类型序列化需要同时保证类型参数也是可序列化的

一个泛型类型可以被定义为是可序列化的:

```
[Serializable]
public class ATFactory<TElement> where TElement : class
{
}
```

不过,需要特别注意的是,泛型实例是否可完全序列化取决于类型参数是否可序列化,否则不可序列化的类型参数将引起序列化过程的数据丢失:

```
public class Element
{
}
class Program
{
    static void Main(string[] args)
    {
        ATFactory<Element> factory = new ATFactory<Element>();
    }
}
```

因此,factory实例是否可完全序列化取决于Element类是否也同时支持序列化。

13. 选择合适的泛型集合作为自定义集合的基类

泛型最广泛的应用就是集合,很多时候预定义的集合无法满足更精细化的需求,就可以考虑实现自定义的集合类型。从最容易和最安全的角度考虑,继承合适的泛型集合是实现自定义集合类最有效的方式,例如从常见的泛型接口 ICollection<T>、 IList<T>、 IEnumerable<T>以及 IDictionary< TKey, TValue >等继承。

详见8.9节“集合通论”的讨论。

14. 在应用 System.Type 的场合,考虑使用泛型代替

考虑使用泛型代替应用 System.Type 的场合,尤其是应用 System.Type 作为方法参数的场合,例如下面实例中通过组件名字和组件类型在运行时匹配组件实例的方法被实现为:

```
public class ComponentUtils
{
    public bool FindComponent(Type component, string name)
    {
        // ...
    }
}
```

通过泛型类型参数将简化 System.Type 在运行时查询的复杂度,并且类型更安全:

```
public class ComponentUtils
{
```

```

public bool FindComponent<TComponent>(TComponent component, string name) where TComponent : IComponent
{
    // ...
}

```

当然，不是所有的场合都可以通过泛型参数代替 System.Type，需要酌情应用。

15. 在应用 System.Object 的场合，考虑使用泛型代替

考虑使用泛型代替应用 System.Object 的场合，尤其是应用 System.Object 作为方法参数的场合，以最典型的 Swap 方法为例：

```

public static void Swap(ref object obj1, ref object obj2)
{
    object temp = obj1;
    obj1 = obj2;
    obj2 = temp;
}

```

上述方法存在的问题是，对于值类型将引发不必要的装箱和拆箱，因此完全可以通过类型参数代替 System.Object 实现没有类型转换的泛型 Swap 方法：

```

public static void Swap<T>(ref T t1, ref T t2)
{
    T temp = t1;
    t1 = t2;
    t2 = temp;
}

```

当然，不是所有的场合都可以通过泛型参数代替 System.Object，需要酌情应用。

16. 请勿在 Web 服务中考虑泛型

在现行标准中，还没有支持泛型的 Web 服务协议，因此无须考虑实现泛型服务支持，例如目前还不能实现一个 WCF 的泛型服务，不过，目前支持将一个泛型类型定义为数据契约，但是必须在服务契约中指定类型参数，并且类型参数本身也必须是可序列化的数据契约。

11.4.3 结论

实践永远是最佳实践唯一的方式，泛型实践也是如此，通过追溯泛型、了解泛型然后深入泛型，在基本认知的基础上实践泛型，才有了“更懂”泛型的机会。

泛型，为.NET 实现了更高层次的优雅与生产力。

参考文献

Christian Nagel, Bill Evjen, Jay Glynn, Professional C# 2005

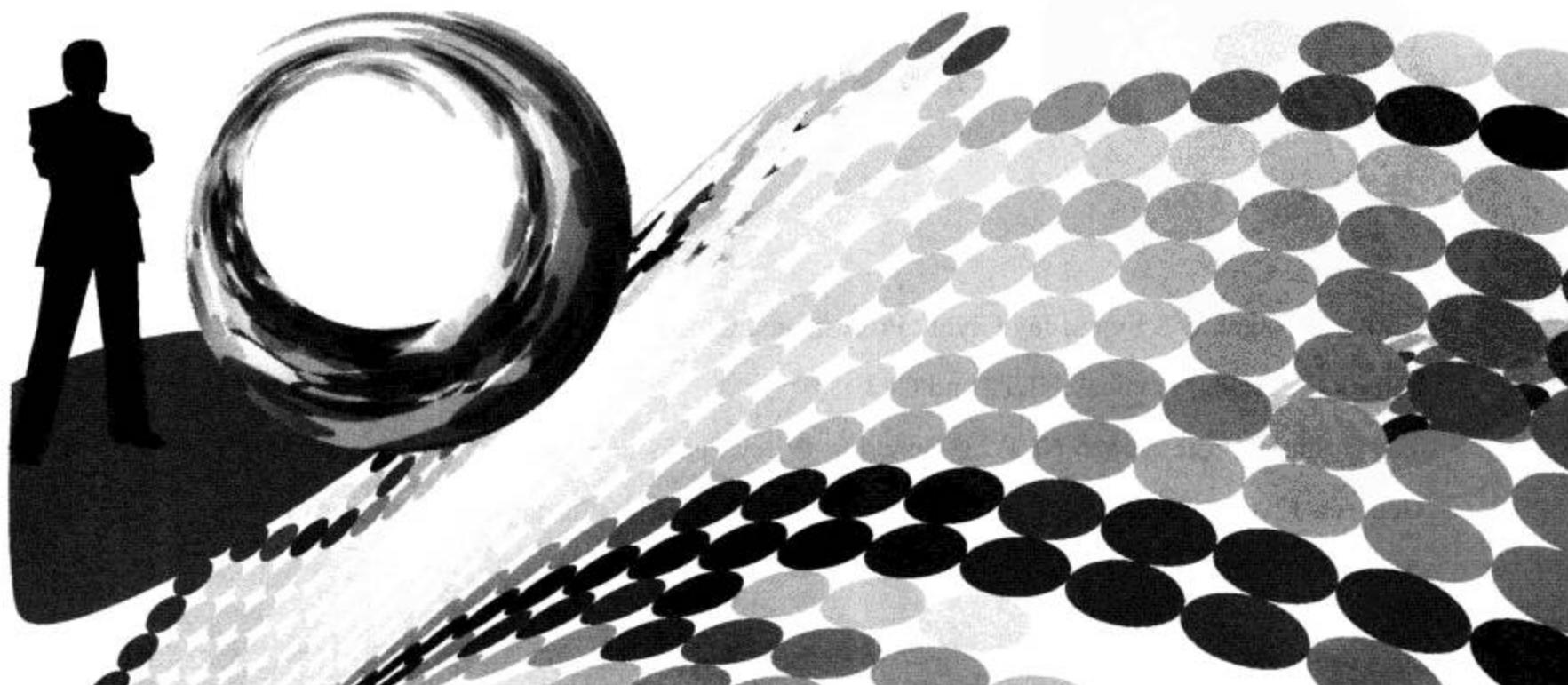
Patrick Smacchia, Practical .NET2 and C#2

Tod Golding, Professional .NET 2.0 Generics, Wiley Publishing, Inc.

Generics FAQ: Best Practices, <http://msdn.microsoft.com/en-us/library/aa479858.aspx>

第12章 如此安全性

12.1 怎么样才算是安全 / 423	12.2.5 安全策略 (Security Policy) / 428
12.1.1 引言 / 423	12.2.6 规则总结 / 429
12.1.2 怎么样才算安全 / 423	12.2.7 结论 / 430
12.1.3 .NET 安全模型 / 423	12.3 基于角色的安全 / 430
12.1.4 结论 / 424	12.3.1 引言 / 430
12.2 代码访问安全 / 424	12.3.2 Principal (主体) / 430
12.2.1 引言 / 424	12.3.3 Identity (标识) / 431
12.2.2 证据 (Evidence) / 425	12.3.4 PrincipalPermission / 432
12.2.3 权限 (Permission) 和权限集 / 426	12.3.5 应用示例 / 432
12.2.4 代码组 (Code Group) / 428	12.3.6 结论 / 433
	参考文献 / 433



12.1 怎么样才算是安全

本节将介绍以下内容：

- 安全性讨论
- .NET 安全策略

12.1.1 引言

网络时代的到来，使得一切信息和系统交互在 Internet 上，信息的互通变得无所不在，而独立运行的软件系统越来越少，更多的软件系统交互的情形越来越多，而恶意代码泛滥的互联网上，系统安全问题成为关注的焦点。

那么，如何构建软件系统的安全，怎么样才算是安全？一般用户下载的系统插件，如何能保证其足够安全，不至于破坏系统数据，甚至本地文件。.NET 的安全策略值得思考，当这么多双眼睛都关注系统安全的时候，你必须有足够的准备来面对可能发生的系统危害。

因此，你必须对 CLR 提供的安全机制有所了解。

12.1.2 怎么样才算安全

传统的安全模型通常以用户信任为出发点，建立在用户和用户组的机制上来提供隔离和访问控制。这种方式至今仍是大部分操作系统应用的安全策略，例如 Windows 系统和 UNIX 系统就采取这种策略。然而，这种模型具有潜在的不足：其基于一个所有代码都具有相同信任度的前提，用户要么可以运行全部代码，要么不能运行，对资源的访问缺乏隔离。在网络互联的世界里，每台机子都可能成为 Internet 的一个终端，基于用户的安全模型显然不能以用户的概念来应付世界另一端的潜在访问者。对于系统的保护，不再建立要么可以运行，要么不能运行的前提条件，一旦获取有效的用户标识和授权，则将对系统造成有可能是毁灭性的安全隐患。从网络下载或者更新的插件、更新包，如何能保证它来自安全的软件提供者，一旦运行这些来路不明的代码，基于角色的安全就无法实现对其的监控，资源得不到有效保护，你只能放弃一切不能够确定的移动代码，或者干脆把网线拔掉。

显然，必须重新考虑对于安全的模型，对于资源的保护策略。在.NET 框架中，基于代码的安全正是针对这一问题而提出的新的安全模型，它实现基于代码的信任，而不是用户。这种原理和大部分的安全保护习惯不同，程序在运行库的严密监视下运行，只有信任的软件才能被授予访问特定资源的权力。

12.1.3 .NET 安全模型

总体来说，.NET 提供了两种安全模型：代码访问安全（Code-Access Security, CAS）和基于角色的安全（Role-Based Security）。其中，代码访问安全用于控制程序的行为，而基于角色的安全用于控制使用者的行为。同时，.NET 的安全系统运行在操作系统的安全系统之上，因此这三种安全性相互补充，相互影响：基于角色的安全构建在代码访问安全性之上，而代码访问安全又构建在操作系统的安全系统之上，如图 12-1 所示。



图 12-1 .NET 的安全模型

代码访问安全根据不同的受信权限来获得可访问的资源，从而免受有害代码的侵害，其基本思想就是由信任软件来保证系统安全，而不是通过信任用户来实现，在规则上具有更高级别的安全保证。代码访问安全基于几个重要的基本概念：证据、安全策略、权限、代码组，实现了对代码的安全监控，可以由任意用户下载或者执行的程序，而不必考虑对“谁来使用”这一问题的设计，只要保证下载的代码是信任就行，这种安全能够有效的保护网络环境下计算机系统免受移动代码的侵害。

而用户安全机制，则可以通过基于角色的安全或操作系统用户安全管理来保护。.NET 同样提供了基于用户的安全，基于用户的身份验证，授权其访问特定资源的权限，.NET 将这一过程抽象为主体（Principals）和标识（Identity）两个基本概念。

另外，.NET 安全系统运行于操作系统安全系统之上，对操作系统的安全性来说，增强和扩展了其安全性。对于资源的保护，仍然受操作系统维护，这也体现了.NET 安全和操作系统安全机制的层次性，某些特定的资源.NET 安全权限也是无法访问的。

12.1.4 结论

安全，不只是人的管理。除了基于角色的安全和操作系统用户管理外，.NET Framework 提供了代码访问安全策略，通过软件的信任级别来管理软件，从而达到对资源的保护目的。这些内容将在本章一一展开讨论，从概念和应用入手来诠释.NET 的安全策略。

12.2 代码访问安全

本节将介绍以下内容：

- 代码访问安全论述
- 代码访问安全应用
- 相关工具介绍

12.2.1 引言

作为.NET 提供的最重要的安全模型，代码访问安全（Code-Access Security，CAS），实现以软件的身份

来管理权限，而不是用户的身份来管理。代码访问安全根据代码的信任级别来管理代码，根据证据来为程序集分配权限，从而实现对资源的访问保护。

在本节，我们通过代码访问安全的基础概念来逐渐建立起对整个 CAS 安全机制的理解，通过概念的内容和概念间的关系，以应用示例为辅助，层层揭开.NET 安全机制的神秘面纱和应用规则，下面就开始我们的探索之旅吧。

12.2.2 证据（Evidence）

证据用于向 CLR 提供代码的特征信息，.NET Framework 必须知道代码的身份和来源，而这些信息的组成就是证据。程序集和应用程序域都接受基于证据的授权，可以由 CLR 提供，也可以由程序集自行提供。其执行过程一般是：CLR 加载程序集时，由 CLR Host 为.NET 安全系统提供证据，然后安全系统根据其证据进行授权，再根据授权来执行对资源的访问控制。目前，.NET 支持的证据主要包括：

- 所有代码（AllMembershipCondition），表示匹配所有的条件。
- 区域（Zone），表示代码来自区域。
- 应用程序目录（Application Directory），表示程序集的安装位置。
- 站点（Site），表示来自 Web 站点。
- URL，表示代码来自的具体位置。
- 发行者（Publisher），表示代码的发布者，可通过 Authenticode Signature 工具检查。
- 强名（StrongName），表示加密强签名，由公钥、程序集名称和版本组成。
- 哈希（Hash），表示 MD5、SHA1 等加密哈希。

其中，StrongName 的程序集被认为是强证据，通过公钥签名的程序集不易伪造，可信度高。在.NET 框架类库中，System.Security.Policy 命名空间中定义了这些证据类，并将其实现为密封类，你不必自己定义安全对象即可将这些证据类继承到安全策略中，你可以通过以下代码来获取当前程序集的所有证据：

```
public static void Main()
{
    // 获取当前程序集的证据
    System.Security.Policy.Evidence e=Assembly.GetExecutingAssembly().Evidence;

    // 枚举程序集具有的所有证据
    IEnumerator enumerator = e.GetAssemblyEnumerator();
    while (enumerator.MoveNext())
    {
        Console.WriteLine(enumerator.Current);
    }
}
```

也可以通过 GetHostEnumerator 枚举主机的所有证据，通过 GetEnumerator 枚举主机和程序集的所有证据。

以其中较为常见的 Zone 类型为例，打开 Internet Explorer 浏览器的 Internet 选项，在其中的安全页你会看到定义好的几个安全区域，在 IE 上可以对这些安全选项加以管理，对其的更改将应用于整个机器。如图 12-2 所示。

在.NET Framework 中，区域由 System.Security.SecurityZone 枚举定义，主要包括以下选项：

- MyComputer，表示本地计算机区域，是隐式的，用于本地计算机上的内容。
- Internet，在Internet上的应用。
- Intranet，本地Intranet上的应用程序。
- Trusted，受信站点上的Internet应用程序。
- Untrusted，表示受限站点上的Internet应用程序。
- NoZone，表示未指定区域。



图 12-2 Internet 安全区域选项

当应用于程序运行时，应于程序将CLR分配到这几个区域，不同的区域被设置了不同的访问权限，安全级别按照：完全信任、中级信任、低级信任和不信任4个级别来设置，不同的级别也就相应的定义了可访问资源的类型。在代码组的讨论中，会看到证据和权限集是如何被组合起来，进行安全判断。

12.2.3 权限（Permission）和权限集

1. 权限

权限，就是对可执行受保护操作的授权（Authorization），以此来实现对特定资源的访问，例如访问注册表的权限；读取事件日志的权限；连接到某个网站的权限等。.NET将权限应用于代码组进行管理，而不是直接对应于程序集，CLR将程序集和代码组进行匹配，并由此来决定为程序集授予的权限，将大大简化权限的管理。目前，.NET用于支持CAS的权限主要包括在由System.Security.CodeAccessPermission类及其派生类构成的继承层次中，CodeAccessPermission类定义了访问权限的基础结构，在此仅列举几个较为常见的权限：

- FileIOPermission，定义了处理文件和文件夹的能力。
- EnvironmentPermission，定义了处理环境变量的能力，包括系统环境变量和用户环境变量。
- EventLogPermission，定义了控制事件日志的能力。
- PrintingPermission，控制对打印机的访问。
- RegistryPermission，控制访问注册表变量的访问能力。
- UIPermission，表示访问用户界面的能力。

在.NET 框架类库中，这些权限类被定义在 System.Security.Permissions 命名空间，主要分为代码访问权限和代码身份权限两种。以 FileIOPermission 权限为例，它提供了四种 IO 访问权限类型：Read、Write、Append、PathDiscovery，这几种类别被定义在 FileIOPermissionAccess 枚举。如果一个程序不具有 FileIOPermission 权限，就无法进行文件或目录的读、写和删除等操作，否则将抛出 SecurityException 异常。

有些时候，系统内置的权限不足以保护某些特殊的系统资源，此时可以考虑实现自定义权限。关于实现自定义权限，有一些基本的规则需要遵守，例如实现 IPermission 和 IUnrestrictedPermission 接口；添加声明式安全支持；以 Permission 为命名后缀等。

.NET 的安全操作主要包括 Demand、Assert、Deny、PermitOnly 等，并且允许以两种方式来执行这些安全操作：一种声明式，通过定制特性将安全约束保留在元数据，在编译时就可以融入到程序集。例如：

```
[FileIOPermission(SecurityAction.Demand, Write = @"E:\tmp.txt")]
class MyClass
{
}
```

另一种是强制式，例如以下代码将检测调用链上各个组件是否具有读写该指定文件的权限：

```
//获取读取和写入 tmp.txt 文件的权限
FileIOPermission fileIOPermission = new FileIOPermission(FileIOPermissionAccess.Read
    | FileIOPermissionAccess.Write, @"E:\tmp.txt");

try
{
    //执行权限检测
    fileIOPermission.Demand();
}
catch (SecurityException se)
{
    Console.WriteLine("没有操作该文件的权限！");
}
```

声明式在编译时表达所有的安全约束，并且只能在程序集、类和完全方法范围内声明；而强制式则在运行时表达安全约束。

2. 权限集

代码访问的权限被聚合成权限集，.NET 包含了已命名的权限集，如果程序集满足代码组的条件，则会授予其被引用的命名权限集，主要包括：

- FullTrust，没有权限限制，允许无限制的访问所有受保护系统资源。
- SkipVerification，跳过验证的程序集。
- Execution，允许代码执行，不能访问受限资源。
- Nothing，不授权限，拒绝所有的资源。
- LocalIntranet，赋予本地 Intranet 上的程序的默认权限。
- Internet，赋予 Internet 应用程序的默认权限。
- Everthing，允许对内置权限的所有资源进行访问。

只有 Everthing 权限集允许被修改，其他权限集都是固定的，在.NET Framework Configuration 工具中，就只提供了对 Everthing 的修改按钮，其他权限集只能被查看。

12.2.4 代码组 (Code Group)

代码组就是证据和权限集的对应组合，也就是说如果一个程序集满足成员条件，则该程序集就被授予对应的权限集，并由此形成一定的逻辑分组。代码组是有层次的，并形成树状结构，根结点是一个命名为“`All_Code`”的代码组，其对应的权限集为“`Nothing`”，表示该代码组未授予任何权限。你可以很容易从.NET Framework Configuration 管理工具中看到其层级结构，如图 12-3 所示。

CLR 将以程序集和代码组进行匹配，如果符合要求就取得该代码组相应的权限集，并继续向下层级对比；如果不符合代码组条件，则不能获得该代码组的权限集，并停止向下对比。那么，各个代码组都包括哪些内容呢？以 `Trusted_Zone` 代码组为例，右键选择查看其属性，如图 12-4 所示。

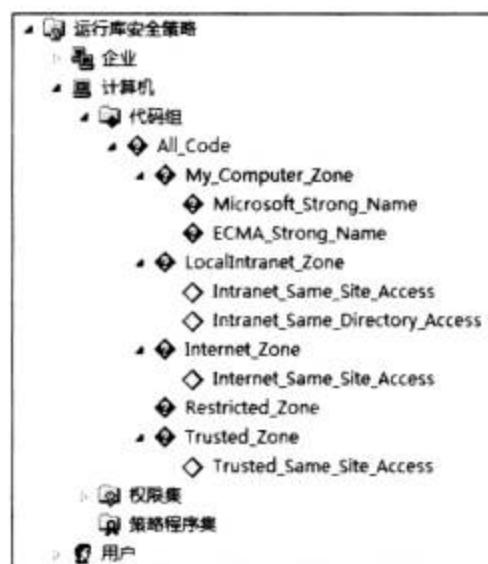


图 12-3 代码组



图 12-4 代码组属性



由图可知，`Trusted_Zone` 代码组将检查程序集的条件类型为“区域”，如果符合“受信任站点”条件，则将授予该程序集 `Internet` 权限，在权限集明细中定义了具有的各个权限。简单地说就是：该代码组将向来自受信任站点的代码授予 `Internet` 权限。

你也可以自定义编辑代码组的属性，或者为代码组添加子代码组，这些操作可以很容易在.NET Framework Configuration 可视化工具中实现。

12.2.5 安全策略 (Security Policy)

安全策略表示了 CLR 在确定代码权限时所遵循的规则，安全系统根据证据的信息来决定授予的使用权限，安全策略定义了一组规则来划分权限等级。通过安全策略将代码组、权限和权限集连接起来，.NET 支持了 4 种级别的安全策略，其中企业级策略级别最高，应用程序域策略级别最低。这些安全策略被定义在 XML 文件中，这些配置文件受 Windows 安全策略保护，只有具有相关权限的 Windows 用户才能有权修改，主要包括：

- 企业级策略 (Enterprise Policy)，由企业管理员拟定，XML 配置文件位置在 `<windir>\Microsoft.NET\Framework\v2.0.50727\CONFIG\enterprisesec.config`。
- 计算机策略 (Machine Policy)，由本地计算机管理拟定，`<windir>\Microsoft.NET\Framework\v2.0.50727\CONFIG\security.config`。

- 用户策略 (User Policy)，由登录用户拟定，%USERPROFILE%\AppData\Roaming\Microsoft\ CLR Security Config\v2.0.50727.1378\security.config。
- 应用程序域策略 (AppDomain Policy)，由 CLR Host 拟定，AppDomain Policy 没有对应的 XML 配置文件，直接写在程序中，一旦拟定不许修改。

注意：上述 XML 配置文件的具体位置可能根据操作系统和.NET Framework 的版本不同而有差别，你可以从.NET Framework Configuration 工具中获得当前.NET 版本的 XML 存储位置。对于前三种安全策略的修改，可以有多种方式选择：

- 直接修改 XML 配置文件。
- 使用.NET Framework 提供的命令行工具 caspol.exe，但是必须了解其命令行信息。
- 使用.NET Framework Configuration 可视化工具，对于 caspol.exe 命令不是很熟悉时，这是最好的管理方式，如图 12-5 所示。



图 12-5 .NET Framework Configuration 工具

安全策略和代码组一起形成一个树状结构，不同的层级对应于不同的管理和宿主方案，程序加载时代码访问安全系统检查所有的安全策略级别，得到的权限授予是各个级别中所有允许的权限的交集。高层级原则可以忽略低层级的策略，企业级策略可以对计算机策略置之不理，而默认的安全策略为计算机策略。

12.2.6 规则总结

- CLR 执行程序时会检查其证据，并根据安全策略的设置，授予其相应的执行权限。
- 代码访问安全源于对软件的信任；基于角色的安全源于对用户的信任。
- 代码访问安全能够处理移动代码对系统的安全保护；基于角色的安全无法应对这种情况。
- 代码访问安全以证据、安全策略、权限和代码组为核心；基于角色的安全以授权和标识为核心。
- 代码访问安全使用堆栈；基于角色的安全不使用堆栈。
- 代码访问安全提供了较为精细的安全检查控制方式，可以通过声明式或强制式两种方式进行。其中 Demand 安全检查是最强的一种，它将执行调用链上每个方法的检查。其他还有包含 LinkDemand、Assert

等方法，但是要留意其有可能引起的安全问题。

- 代码访问安全是可扩展和修改的，通过扩展相关的类，实现自定义的证据、权限和代码组，扩展和修改安全策略，在.NET 安全模型中是很容易的。

12.2.7 结论

代码访问安全是.NET 框架提供的又一特色内核，其改变了传统安全模型关注于用户和角色的思维，而将目标转向对程序的信任。通过代码访问安全可以对来源不同的程序集根据其证据，获得不同的受信度，并通过不同级别的信任，来授予不同的权限，从而大大减少恶意代码对系统的攻击，提供了系统安全保证。代码访问安全是.NET 框架提供的最重要安全模型，能有效地为系统安全提供更广泛的运行机制。

12.3 基于角色的安全

本节将介绍以下内容：

- 基于角色的安全讨论
- 基于角色的安全验证示例

12.3.1 引言

代码访问安全，根据代码的证据来实现对代码的授权，而基于角色的安全提供了另一种安全选择，它根据用户或者角色的证据进行授权，并且不使用堆栈进行跟踪。在很多的应用系统中，通过使用角色对权限进行强制管理，不同角色的成员具有不同的权限，处理事务和访问资源的限制根据其权限有所不同，这种安全模型就是基于角色的安全。

本节将对.NET 基于角色的安全做个讨论，它通过 Principal 信息来支持授权，而 Principal 则由关联的 Identity 来构造。因此，Principal（主体）和 Identity（标识）是基于角色的安全中两个核心的概念。Principal 代表了用户和其角色的抽象，其中包含了对 Identity 对象的引用；而 Identity 则封装了正在验证的用户或实体信息。因为 Principal 压缩了用户和角色，所以.NET 运行库将基于角色的安全都以 Principal 作为基础。下面我们将通过对这几个概念的分析来理解.NET 基于角色的安全机制。

12.3.2 Principal（主体）

Principal 对象封装了当前用户的信息，例如用户身份、角色等，在.NET 中，Principal 类都必须实现 IPrincipal 接口，其定义为：

```
public interface IPrincipal
{
    bool IsInRole(string role);
    IIdentity Identity { get; }
}
```

该接口包含一个 IsInRole 方法，该方法参数为一个表示角色名称的字符串，返回该 Principal 对象是否属

于指定的角色；还包含一个 `Identity` 属性，指向与 `Principal` 对象关联的用户标识。在.NET 中主要有 2 个类实现了该接口，它们位于 `System.Security.Principal` 命名空间，分别是：

- `GenericPrincipal`，表示一般用户，通常和 `GenericIdentity` 类一起使用。
- `WindowsPrincipal`，表示 Windows 用户，实现了 Windows 组成员条件映射到角色的附加功能，通常和 `WindowsIdentity` 类一起使用。

其中，`WindowsPrincipal` 实现了 3 种 `IsInRole` 重载方法：

```
public virtual bool IsInRole(SecurityIdentifier sid);
public virtual bool IsInRole(WindowsBuiltInRole role);
public virtual bool IsInRole(string role);
```

第一种 `sid` 为 Windows 角色标识（RID），它是 Windows 组安全标识符的一个组件；第二种是一个 `WindowBuiltInRole` 枚举；第三种是一个表示角色名称的字符串。通常使用的 Windows 角色及其 RID、`WindowsBuiltInRole` 关系为表 12-1。

表 12-1 Windows 角色

角色名称	RID	<code>WindowsBuiltInRole</code>
BUILTIN\Account Operators	0x224	AccountOperator
BUILTIN\Administrators	0x220	Administrator
BUILTIN\Backup Operators	0x227	BackupOperator
BUILTIN\Guests	0x222	Guest
BUILTIN\Power Users	0x223	PowerUser
BUILTIN\Print Operators	0x226	PrintOperator
BUILTIN\Replicator	0x228	Replicator
BUILTIN\Server Operators	0x225	SystemOperator
BUILTIN\Users	0x221	User

还有 `System.Web.Hosting.IIS7UserPrincipal`、`System.Web.Security.RolePrincipal` 类也实现了 `IPrincipal` 接口，用于表示其他类型的用户身份。你也可以通过实现 `IPrincipal` 接口来实现自定义的 `Principal` 类。

12.3.3 Identity（标识）

`Identity` 对象包含了用户的身份信息和验证身份的方法。在.NET 中，代表用户标识的类必须实现 `IIdentity` 接口，其定义为：

```
public interface IIdentity
{
    string AuthenticationType { get; }
    bool IsAuthenticated { get; }
    string Name { get; }
}
```

该接口包含三个只读属性：`AuthenticationType` 表示使用身份验证的类型；`IsAuthenticated` 表示用户是否通过验证；`Name` 表示当前用户的名称。在.NET 中有 4 个类实现了该接口，它们位于 `System.Security.Principal` 命名空间，分别是：

- `GenericIdentity`，表示一般用户，用于大多数的自定义登录方案。

- WindowsIdentity，表示Windows用户，常用于依赖Windows身份验证的应用程序。
- FormsIdentity，表示一个使用Forms身份验证的用户，常用于ASP.NET应用程序。
- PassportIdentity，表示一个使用Passport的应用程序的用户。必须安装Passport SDK才能使用该类。

12.3.4 PrincipalPermission

为了使基于角色的安全和代码访问安全在使用上趋于一致，.NET框架提供了PrincipalPermission类（位于System.Security.Permission）来实现类似的授权方式，即通过声明式和强制式两种方式来检查活动用户。

声明式，可以使用PrincipalPermissionAttribute特性来声明一个权限，例如：

```
[PrincipalPermissionAttribute(SecurityAction.Demand, Name = "小王", Role = "士官")]
public static void OfficerMethod()
{
    //
}
```

强制式，通过创建PrincipalPermission实例来实现，例如：

```
public static void OfficerMethod()
{
    string name = "小王";
    string role = "士官";
    PrincipalPermission principalPermission = new PrincipalPermission(name, role);
    principalPermission.Demand();
}
```

12.3.5 应用示例

每个线程都与一个Principal对象相关联，而每个Principal对象都有一个指向Identity对象的引用，因此，我们可以根据权限要求来访问该对象，执行一定的安全验证演示：

```
public static void Main()
{
    string[] roles = { "将军", "士官", "士兵" };
    // 创建 GenericIdentity 对象
    GenericIdentity identity = new GenericIdentity("小王");
    // 创建 GenericPrincipal 对象
    GenericPrincipal principal = new GenericPrincipal(identity, roles);
    // 将 principal 对象附加到当前线程
    Thread.CurrentPrincipal = principal;

    // 执行验证
    if (Thread.CurrentPrincipal.Identity.Name == "小王")
    {
        Console.WriteLine("小王可以指挥士兵。");
    }

    // 执行角色检查
    if (Thread.CurrentPrincipal.IsInRole("士官"))
    {
        Console.WriteLine(identity.Name + ", 士官你好。");
    }
}
```

可见，示例中的角色和用户都和 Windows 用户与角色没有任何关系，通过 GenericPrincipal 和 GenericIdentity 可以很容易实现自定义登录的安全管理。例如，根据用户和密码，在数据库中检索来查询用户标识信息以获取权限，程序将通过创建一个基于数据库记录的 Principal 和 Identity 对象。

同理，应用 WindowsPrincipal 和 WindowsIdentity 类来实现依赖于 Windows 用户验证的应用程序，例如：

```
class WindowsRoleBaseSecurity
{
    public static void Main()
    {
        // 创建一个 Windows 标识
        WindowsIdentity wi = WindowsIdentity.GetCurrent();
        // 创建一个 WindowsPrincipal 实例
        WindowsPrincipal wp = new WindowsPrincipal(wi);
        // 将 wp 对象附加到当前线程
        Thread.CurrentPrincipal = wp;

        // 基于角色的验证
        try
        {
            WindowsUserMethod();
        }
        catch (SecurityException ex)
        {
            Console.WriteLine("未通过验证！");
        }
    }

    [PrincipalPermissionAttribute(SecurityAction.Demand, Role="BUILTIN\\Users")]
    public static void WindowsUserMethod()
    {
        Console.WriteLine("通过验证。");
    }
}
```

PrincipalPermissionAttribute 特性保证了每次调用 WindowsUserMethod 时进行运行库检查，例如角色设置为 BUILTIN\Server Operators 时，则可能导致检查失败。同样也可以以强制方式来实现基于角色的安全检查。

12.3.6 结论

.NET 提供了更加灵活和可扩展的基于角色的安全机制，尤其是广泛应用于 ASP.NET Web 应用程序中。.NET 安全检查更加灵活、使用简单，并且很好地同现有身份验证结构相互操作，集成 Windows 用户账户系统，实现了更好的角色安全检查模型。

参考文献

Adam Freeman, Allen Jones, Programming .NET Security

Patrick Smacchia, Practical .NET2 and C#2

Christian Nagel, Bill Evjen, Jay Glynn, Professional C# 2005

蔡学镛, .NET 安全性, <http://www.microsoft.com/taiwan/msdn/columns/DoNet/DotNetSecurity.htm>



学习笔记

第 5 部分

未来——.NET 技术展望

转眼间.NET 十年了，.NET 4.0 如疾风劲雨般登场，一场技术的变革接踵而来，动态编程、并行计算、协变与逆变如期而至，为.NET 在接下来的岁月带来无与伦比的色彩；而.NET 3.0/3.5 也风起云涌，迎来新一轮的技术磨合。

希望与挑战并存的年代，如何以无畏的态度来迎接每一次的技术革命，是每个从事技术研究与应用的人必须思考的话题。.NET 技术从 2.0 起就显现出无与伦比的未来价值，把握技术脉搏就是选择未来的机遇。本部分从.NET 3.0/3.5 的新特性谈起，力求以最简洁的论述，带你进入.NET 技术的最新研究，领略 WPF 带来的神奇表现，体验 WCF 在分布式应用的一统天下，品位 LINQ 带给程序编码的革命走向，熟悉 Visual Studio 2008 的开发魅力，走查 C# 3.0 的最新特性。然后，以浓重的笔墨，将.NET 4.0 的新特性一一展示，重点突出动态编程和并行计算为语言本身带来的变革，同时兼顾一系列新的语言特性，包括泛型的协变与逆变、命名参数和可选参数等。

本部分主要包括：

- 第 13 章 走向.NET 3.0/3.5 变革
- 第 14 章 跟随.NET 4.0 脚步

第13章 走向.NET 3.0/3.5 变革

13.1 品读新特性 / 437

 13.1.1 引言 / 437

 13.1.2 .NET 新纪元 / 437

 13.1.3 程序语言新特性 / 438

 13.1.4 WPF、WCF、WF / 438

 13.1.5 Visual Studio 2008 体验 / 439

 13.1.6 其他 / 439

 13.1.7 结论 / 439

13.2 赏析 C# 3.0 / 439

 13.2.1 引言 / 440

 13.2.2 对象初始化器

 (Object Initializers) / 440

 13.2.3 集合初始化器

 (Collection Initializers) / 441

 13.2.4 自动属性

 (Automatic Properties) / 442

 13.2.5 隐式类型变量 (Implicitly Typed

 13.2.5 Local Variables) 和 隐式类型数组

 (Implicitly Typed Array) / 444

 13.2.6 匿名类型 (Anonymous Type) / 445

 13.2.7 扩展方法

 (Extension Methods) / 446

 13.2.8 查询表达式

 (Query Expressions) / 448

 13.2.9 结论 / 448

13.3 LINQ 体验 / 449

 13.3.1 引言 / 449

 13.3.2 LINQ 概览 / 449

 13.3.3 查询操作符 / 451

 13.3.4 LINQ to XML 示例 / 451

 13.3.5 规则 / 453

 13.3.6 结论 / 453

13.4 LINQ 江湖 / 453

 13.4.1 引言 / 453

 13.4.2 演义 / 453

 13.4.3 基于 LINQ 的零代码数据访问
 层实现 / 459

 13.4.4 LINQ to Provider / 462

 13.4.5 结论 / 463

13.5 抢鲜 Visual Studio 2008 / 463

 13.5.1 引言 / 463

 13.5.2 Visual Studio 2008 概览 / 464

 13.5.3 新特性简介 / 465

 13.5.4 开发示例 / 465

 13.5.5 结论 / 466

13.6 江湖一统: WPF、WCF、WF / 467

 13.6.1 引言 / 467

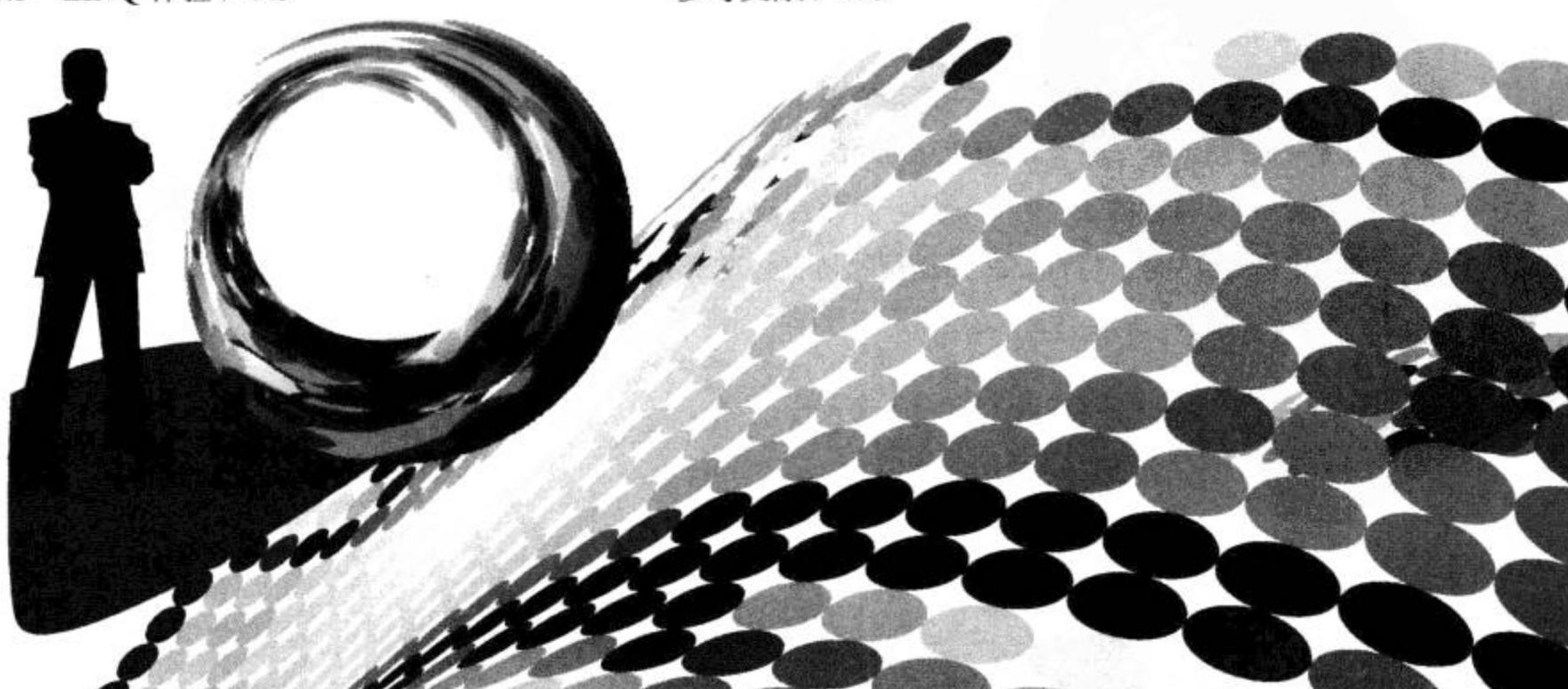
 13.6.2 WPF / 467

 13.6.3 WCF / 468

 13.6.4 WF / 469

 13.6.5 结论 / 470

参考文献 / 470



13.1 品读新特性

本节将介绍以下内容：

- .NET 新纪元
- .NET 新特性解析

13.1.1 引言

.NET 3.0 硝烟未散，.NET 3.5 烽火又燃，回顾.NET 几年来的脚步，每次版本的升级都带来诸多惊喜，新技术和新体验蜂拥而来。对技术开发人员又一次的学习风潮也将昂然而至，如何享受每一次的技术大餐，而不在茫然中迷失，是我们应该思考的问题。

本章将从介绍.NET 3.0 和.NET 3.5 带来的新技术入手，适度遍历未来几年我们应关注的技术方向。当然，这里不可能将所有的新鲜技术一网打尽，然而了解最重要的几个方面，还是很有必要的。

13.1.2 .NET 新纪元

了解.NET 3.0/3.5 必须首先从大局上来把握其基本的脉络，从框架中了解哪些是原来的，哪些是新增的，从而确定基本的认识思路和学习成本，对于.NET 新特性来说，我们在此以.NET 3.5 的基本框架图（图 13-1）做以说明。

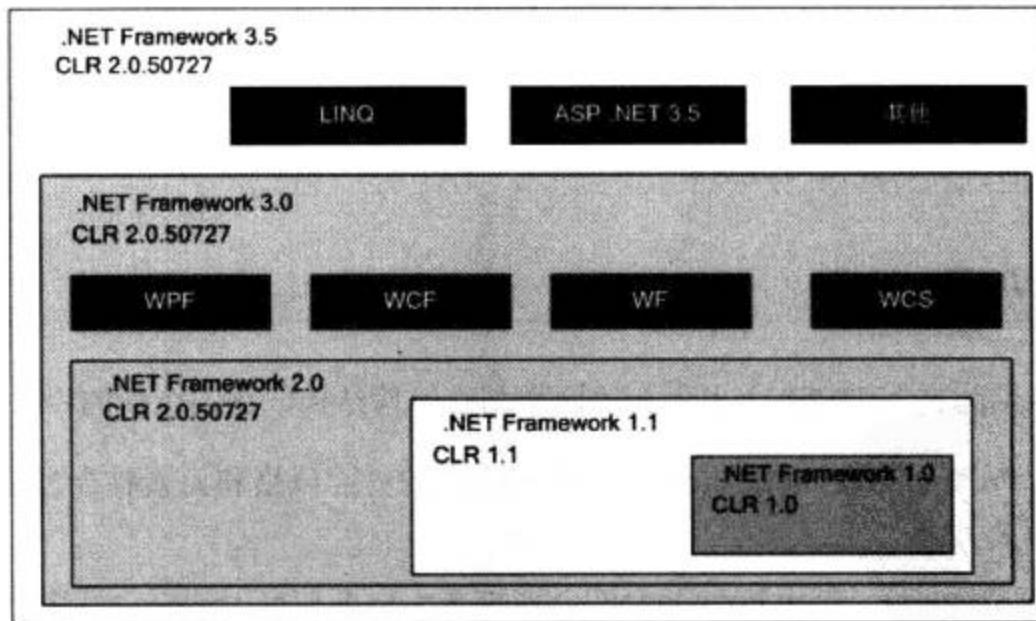


图 13-1 .NET 3.5 框架图

由图 13-1 可知，.NET 3.0/3.5 构建在.NET 2.0 内核之上，仍然使用 2.0 的 CLR 版本，.NET 3.0 加入 4 个新的组件：WPF、WF、WCF、WCS，并引入很多新的语言特性；.NET 3.5 在编程语言中内置 LINQ 数据查询模型等。本节对新特性的论述不做具体版本的区分，一律以统一新特性术语进行论述，而具体的特性格局以图 13-1 所示的框架图为准。而对新技术的阐释，本节更像是一个流水账，详细的描述在本章的后续内容中将做具体分析。

13.1.3 程序语言新特性

1. C# 3.0 新特性

在新的框架中,.NET 并没有对运行库做过多的修改,而是着力提升高级语言的吸引力,在 C# 3.0 中为语言特性注入了很多激动人心的新特性,主要包括:

- 匿名类型
- 对象初始化器
- 集合初始化器
- 自动实现属性
- 扩展方法
- Lambda 表达式
- 查询表达式
- 表达式树
- 不完全方法

这些语言特性为开发人员的编程带来新的体验和方便,同时很多特性为实现 LINQ 提供了基础,在 13.2 节“赏析 C# 3.0”一节将做详细介绍。

2. LINQ

LINQ 是语言集成查询 (Language Integrated Query) 的缩写,它的出现为.NET 注入了吸引我们眼球的活力,在高级语言中植入 LINQ 技术,将使得各种各样的数据以面向对象的方式来处理,例如 LINQ to Objects、LINQ to DataSets、LINQ to SQL、LINQ to Entities 和 LINQ to XML,分别用于处理不同的数据形式。在 13.3 节“LINQ 体验”中我们将做以介绍与分析。

13.1.4 WPF、WCF、WF

有人将.NET 3.0 归结为在.NET 2.0 上引入新的四个基础组件结构,这四个新的组件主要包括:

- WPF (Windows Presentation Foundation), 实现将不同的界面风格和元素整合为统一的处理架构, 提供一致的表现层技术基础。
- WCF (Windows Communication Foundation), 通过基于标准的框架和组合架构进行分布式应用程序的开发,WCF 整合了微软在分布式应用中的诸多技术,例如.NET Remoting、XML Web Services、Enterprise Service 还有 MSMQ 等,从而形成了以一致方式进行分布式处理的框架。
- WF (Windows Workflow Foundation), 一个企业级的工作流开发框架和引擎,提供了运行时支持和灵活的控制机制,支持有人参与的、系统的、连续的状态机工作流。
- WCS (Windows CardSpace), 通过创建一个身份元系统,为不同的身份标识管理提供了统一的框架并使其协同工作。

13.1.5 Visual Studio 2008 体验

随着新框架的发布，新的 Visual Studio 2008 开发工具也有节奏地发布了，新的工具当然也不例外地引入很多新特性，为实现更酷的编程体验，Visual Studio 2008 主要包括以下的新特性：

- 多定向支持（Multi-Targeting Support）。
- JavaScript 智能感知和调试（JavaScript Intellisense and Debugging）。
- Web 设计器和 CSS 支持（Web Designer and CSS Support）。
- 嵌套母版页（Nested Master Page）。

13.1.6 其他

除了上述这些引人关注的新特性之外，还有一些特性同样备受瞩目，这些改进主要包括：

- 垃圾回收
- 基础类库
- 安全性
- 动态语言支持
- Office 更多支持

13.1.7 结论

新特性，就是技术领域的一股旋风，总会刮得轰轰烈烈。在.NET 领域，由于微软强大的技术推动力，其版本升级的力量和进程都非比寻常，经过 5 年多的发展，我们甚至还没有看清.NET 2.0 的时候，就已经是.NET 3.5 了。

技术论坛上经常将追赶新技术的话题作为讨论的热点，普遍的观点是新技术会让开发人员手忙脚乱，而轻松地应付技术升级的脚步似乎是不可能完成的任务。面对层出不穷的新技术，我们该何去何从？其实答案自在人心，对于.NET 3.5 的新的特性，如果你看得够准，认得够清，迅速地消化.NET 3.5 也并非难事。因为技术升级是一个更迭有序的过程，3.5 的基础是 3.0，而 3.0 的基础是 2.0，这条主脉搏是不会变也不可能变的。因此如果能够深入地把握好 2.0，那么 3.0/3.5 的了解其实不在话下。

事实上，.NET 3.5 的 CLR 仍然是 2.0 版本，所以其本身带来的很多新特性，在本质上和 2.0 没有太大的区别，所以本章将为你带来新特性体验，很多从本质上为您分析，其实新特性是如此简单，你大可不必为追赶而伤神。扪心自问，关键是对于.NET 2.0，你已经掌握得够好了吗？

13.2 赏析 C# 3.0

本节将介绍以下内容：

- C#新特性赏析
- 新特性本质分析

13.2.1 引言

本节就以.NET在语言方面的新特性作为入口，来品尝这美味的第一餐，由于本书是以C#语言作为研究基础的，所以在此仅关注C#3.0的最新特性。

可以说，C#3.0带来了超强的编程体验，一系列的语言特性让C#充满了新的活力，代码更加简洁、干净，富有表达意义。同时这些新特性的引入为另一激动人心的技术LINQ做好了铺垫，综观C#3.0的新特性，我们不难发现这些新特性主要都是编译器对C#语言进行的修改，以实现基于函数式编程（Functional Programming）的概念支持，在本节中我们就将对这些新特性从本质上做以分析，从而认识其本来的面目。

下面就开始逐一对这些特性进行品读，从中感受至酷无比的编程体验，通过和.NET2.0方式的对比，一方面体会新特性带来的优质体验；另一方面通过反编译代码来分析，也可了解二者在本质上并没有太大差别。

13.2.2 对象初始化器（Object Initializers）

对象初始化本来是构造器的专职工作，但是有时候对于没有在构造器中完成初始化的成员，还可以通过赋值方式来完成，而对象初始化器正是用于完成这一工作而实现的便捷操作。

1. 2.0方式

对于对象的成员的初始化，在2.0主要通过以下方式来实现：

```
User user = new User();
user.Name = "小王";
user.Age = 27;
```

2. 3.0方式

在3.0中，实现对象初始化有了更好的解决方案，由对象初始化器来完成：

```
User user = new User { Age = 27, Name = "小王" };
```

你看，多简洁而优美的代码，这就是C#3.0中激动人心的特性。

3. 本质分析

以Reflector工具对上述代码进行反编译，生成的结果为：

```
public static void Main()
{
    User <>g__initLocal0 = new User();
    <>g__initLocal0.Age = 0x1b;
    <>g__initLocal0.Name = "小王";
    User user = <>g__initLocal0;
}
```

由反编译结果可知，编译器为对象生成器进行了以下处理：构造一个User类的临时对象<>g_initLocal0，并对该临时变量的成员进行赋值操作，然后将初始化的临时变量赋给我们创建的user对象。因此，CLR并没有为对象生成器做任何附加的操作，完成这一切的是C#编译器。

另外，Visual Studio 2008的智能感知能够自动识别其成员，在初始化器中进行初始化的没有顺序要求，

而且不要求初始化所有的类成员，你可以根据具体的需要进行初始化选择，例如图 13-2。

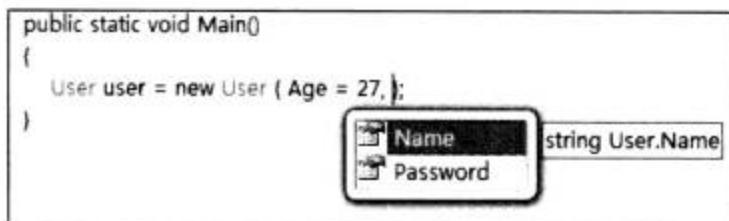


图 13-2 Visual Studio 2008 的智能感知识别对象初始化成员

对象初始化器能够灵活完成对象成员初始化的操作，而在以前这一切需要初始化类提供多个过程烦琐的构造器来完成，有多少种情况就得需要多少个构造器，另外对象初始化器还可以完成有嵌套定义的结构，例如：

```
User user = new User { Name = "小王", Age = 27, UserInfo = new UserInfo{ PhoneNo = 123456789, Email = "mail@anytao.com", IsVIP = true } };
```

13.2.3 集合初始化器 (Collection Initializers)

对象初始化器可以完成对象成员的初始化，而同样的情况存在于集合对象元素的初始化过程，不过这两个过程其实是有区别的。C# 3.0 引入集合初始化器来完成，我们首先了解 2.0 中完成一个集合元素的初始化过程：

1. 2.0 方式

在 2.0 中，集合元素通过 Add 方法来加入：

```
public static void Main()
{
    List<User> users = new List<User>();
    users.Add(new User { Name = "小王", Age = 27 });
    users.Add(new User { Name = "小佳", Age = 22 });
}
```

2. 3.0 方式

而在 3.0 中，集合初始化器能轻松完成这一操作：

```
public static void Main ()
{
    List<User> userss = new List<User>{
        new User{Name = "小王", Age = 27},
        new User{Name = "小佳", Age = 22}
    };
}
```

3. 本质分析

以 Reflector 工具对上述代码进行反编译，生成的结果为：

```
public static void Main()
{
    List<User> <>g__initLocal0 = new List<User>();
    User <>g__initLocal1 = new User();
    <>g__initLocal1.Name = "小王";
    <>g__initLocal1.Age = 0x1b;
    <>g__initLocal0.Add(<>g__initLocal1);
```

```
User <>g__initLocal2 = new User();
<>g__initLocal2.Name = "小佳";
<>g__initLocal2.Age = 0x16;
<>g__initLocal0.Add(<>g__initLocal2);
List<User> usersss = <>g__initLocal0;
}
```

由反编译结果可知，编译为 IL 后仍然是通过生成临时集合对象，并调用其 Add 方法来加入集合元素，最后再将临时对象赋给我们创建的对象。在本质上它和.NET 2.0 的方式是相同的。另外，Visual Studio 2008 同样提供了相应的智能感知支持。

13.2.4 自动属性 (Automatic Properties)

自动属性用于完成对属性的语法简洁化操作，虽然 Visual Studio 2005 提供了代码重构来封装字段，然而还是会让 C# 代码看起来多少有些多余，而自动属性的引入彻底改变了这种现状。

1. 2.0 方式

首先来了解原来的属性定义是如何实现的：

```
class User
{
    private string name;

    // 定义只读属性
    public string Name
    {
        get { return name; }
    }

    private Int32 age;

    public Int32 Age
    {
        get { return age; }
        set { age = value; }
    }
}
```

2. 3.0 方式

```
class User
{
    public string Name { get; private set; }
    public Int32 Age { get; set; }
}
```

自动属性带来的简洁体验，真是无与伦比，另外请注意对只读属性的定义要为 set 访问器添加 private 修饰符。

3. 本质分析

以 Reflector 工具对上述代码进行反编译，生成的结果为：

```

internal class User
{
    //自动生成的私有字段
    [CompilerGenerated]
    private int <Age>k__BackingField;
    [CompilerGenerated]
    private string <Name>k__BackingField;

    //调用 get 和 set 完成属性定义
    public int Age
    {
        [CompilerGenerated]
        get
        {
            return this.<Age>k__BackingField;
        }
        [CompilerGenerated]
        set
        {
            this.<Age>k__BackingField = value;
        }
    }

    public string Name
    {
        [CompilerGenerated]
        get
        {
            return this.<Name>k__BackingField;
        }
        private [CompilerGenerated]
        set
        {
            this.<Name>k__BackingField = value;
        }
    }
}

```

从反编译代码可知，C#编译器为自动属性代码完成了一整套的附加操作：首先根据属性的类型构建两个私有字段<Age>k__BackingField 和<Name>k__BackingField，然后通过 get 和 set 范围器来为这两个成员构建相应的属性定义。唯一要注意的是，对于只读属性的处理，编译器仍然定义了 set 访问器的实现，但是为其加入 private 修饰符。

4. 规则小结

对于字段的某些附加的逻辑，自动属性无法完成，必须应用传统的属性定义方式来实现，例如：

```

public Int32 Age
{
    get { return age; }
    set
    {
        if ((value > 0) && (value < 150))
        {
            age = value;
        }
        else
        {
            throw new ArgumentOutOfRangeException("年龄不合法。");
        }
    }
}

```

上述代码在属性定义中封装了对年龄 Age 的附加验证，而自动属性显然不能胜任这样的工作，此时原来的属性定义方式还会再次归来。

- 必须同时实现 get 和 set，对于只读或者只写属性，为其访问器添加 private 访问修饰符即可。
- 注意和 abstract property 的区别，例如：

```
abstract class User
{
    //定义自动属性
    public string Name {get; set; }
    //定义抽象属性
    abstract public Int32 Age { get; }
```

13.2.5 隐式类型变量(Implicitly Typed Local Variables)和 隐式类型数组(Implicitly Typed Array)

对于熟悉动态语言的人来说，var 关键字并不陌生，在 JavaScript 脚本中随处可见通过 var 声明的变量，现在 C# 3.0 也引入了 var 关键字，用于定义隐式类型变量或隐式类型数组，例如：

```
public static void Main()
{
    //定义基元类型
    var i = 100;
    var str = "Hello, world.";
    //定义数组类型
    var arr = new[] { "小王", "张三", "李四" };
    //定义自定义类型
    var user = new User { Name = "小王", Age = 27 };
}
```

由此可见，隐式类型变量可以为任何.NET 类型，通过本质分析可以进一步了解编译器为隐式类型变量实现的附加操作。

1. 本质分析

以 Reflector 工具对上述代码进行反编译，生成的结果为：

```
public static void Main()
{
    int i = 100;
    string str = "Hello, world.";
    string[] arr = new string[] { "小王", "张三", "李四" };
    User <>g__initLocal0 = new User();
    <>g__initLocal0.Name = "小王";
    <>g__initLocal0.Age = 0x1b;
    User user = <>g__initLocal0;
}
```

由反编译结果可知，编译器在使用隐式变量时，将根据其值的类型来反推出变量本身的类型，然后将类型的声明由编译器自动完成。

2. 规则小结

- 在 2.0 中可以以 Object 根类型来声明任何类型，但是这种方式在使用变量时一般存在类型转换操作，而且如果为值类型，还可能引起装箱，从而影响性能。而以 var 来声明的变量，在编译时就会根据值的类型来确定变量的类型，不存在类型转换和装箱问题。
- 隐式类型变量必须进行初始化，而且值不能为 null，或者类型不定的表达式，否则编译器无法判断其类型。
- 由数组类型反编译的结果可知，隐式数组类型在初始化时，必须保证各个元素为同一类型。
- 隐式类型仍为强类型，只是类型的声明由编译器完成。

13.2.6 匿名类型 (Anonymous Type)

匿名类型，通过隐式类型、对象初始化器来构建一个类型未知的对象，并以内联方式来定义对象的数据结构，所以匿名类型实现了在不定义类型的前提下实现对象的创建。

1. 非匿名类型方式

首先定义一个具体的类：

```
class User
{
    public string Name { get; set; }
    public Int32 Age { get; set; }
}
```

然后通过 new 关键字来创建新的对象：

```
User user = new User { Name = "小王", Age = 27 };
```

2. 匿名类型方式

对于匿名类型对象的创建，则其实现代码表现得更简洁：

```
var user = new { Name = "小王", Age = 27 };
```

var 定义了一个隐式类型变量 user，user 的类型由它的值来决定，因此右边的数据结构将能够返回一个具体的类型，而这一具体类型的创建是由编译器来完成的。

3. 本质分析

以 Reflector 工具对上述代码进行反编译，IL 中将新增一个类，分析其实现主要为：

```
internal sealed class <>f__AnonymousType0<<Name>j__TPar, <Age>j__TPar>
{
    // 定义字段
    [DebuggerBrowsable(DebuggerBrowsableState.Never)]
    private readonly <Age>j__TPar <Age>i__Field;
    [DebuggerBrowsable(DebuggerBrowsableState.Never)]
    private readonly <Name>j__TPar <Name>i__Field;

    // 定义构造器
    [DebuggerHidden]
    public <>f__AnonymousType0(<Name>j__TPar Name, <Age>j__TPar Age);

    // 实现 Object 基类方法，省略.....
}
```

```
// 定义属性
public <Age>j__TPar Age { get; }
public <Name>j__TPar Name { get; }
}
```

由该附加类的构造可知，编译器根据对象初始化器中的数据成员及其类型，构建一个密封类。本例中编译器创建两个数据成员<Name>i__Field 和<Age>i__Field，同时实现相应的属性 Name 和 Age。

对于匿名类型，C# 3.0 的规则是：对象初始化器中的参数名称、类型和顺序都一致时，这些匿名类型的对象将视为同一种类型。例如：

```
var v1 = new { Name = "Aero", Age = 27 };
var v2 = new { Name = "Emma", Age = 22 };
var v3 = new { Age = 27, Name = "Aero" };
```

变量 v1 和 v2 的类型将被视为相同，而 v1 和 v3 的类型则被认为是不同的类型，可以通过对象的 GetType 方法获取 Type 来验证。

13.2.7 扩展方法 (Extension Methods)

扩展方法实现了在保持原有类型不做任何改变的情况下，对其进行扩展。这种扩展可以是同一程序集内的方法扩展，也可以是不同程序集的方法扩展。扩展方法大大提高了系统设计的灵活度，按照开放封闭的原则考虑，这种扩展机制实现了更加柔性的处理机制，你可以很容易实现对第三方程序集的扩展，很容易为已经设计完成的系统创建新的功能点，很容易解决已发布程序集的扩展需求。因此，扩展方法不仅是 LINQ 实现的基础之一，同时也提供了颇具创意的扩展机制，因此值得推荐，以一个简单的实例做进一步说明：

```
static class MyExtensions
{
    // 扩展自定义类型
    public static void ShowInfo(this User user)
    {
        Console.WriteLine("Name: {0}\nAge: {1}", user.Name, user.Age);
    }

    // 扩展 String 的方法，对已有程序集的扩展
    public static void TellType(this string str)
    {
        Console.WriteLine("我是字符串: {0}", str);
    }
}
```

在客户端，可以像调用实例方法一样来调用扩展方法，例如：

```
public static void Main()
{
    User user = new User { Name = "Emma", Age = 26 };
    // 调用扩展方法
    user.ShowInfo();
    string str = "123abc";
    // 调用扩展方法
    str.TellType();
}
```

1. 本质分析

由扩展方法的定义过程可知，扩展方法必须实现为静态方法，只能声明于非泛型或非嵌套的静态类中，

并且必须以 this 关键字作为其第一个参数修饰符。将扩展方法反编译为 IL 语言，以 System.String 的扩展方法 TellType 方法为例：

```
.method public hidebysig static void TellType(string str) cil managed
{
    .custom instance void [System.Core]System.Runtime.CompilerServices.ExtensionAttribute::ctor()
    .maxstack 8
    L_0000: nop
    L_0001: ldstr "\u6211\u662f\u5b57\u7b26\u4e32\uff1a{0}"
    L_0006: ldarg.0
    L_0007: call void [mscorlib]System.Console::WriteLine(string, object)
    L_000c: nop
    L_000d: ret
}
```

你会发现，在 TellType 方法定义中附加了 [System.Core]System.Runtime.CompilerServices.ExtensionAttribute::ctor() 定制特性，C# 编译器由第一个被 this 修饰的关键字来决定对该类型的扩展，而 VB.NET 则直接通过加入 ExtensionAttribute 来定义扩展方法，这种语法在 C# 中不被允许，直接为方法添加 ExtensionAttribtue 将导致编译错误。

因此，扩展方法在本质上是一个静态方法，具有静态方法的一切功能，不同的是 this 关键字标识编译器为其隐式添加了 ExtensionAttribute 特性。以 string 的方法 TellType 为例，在编译器编译时，将根据调用类型 string 来查找 TellType 是实例方法还是扩展方法，如果同时找到实例方法和扩展方法，则根据实例方法优先的原则调用 string 的实例方法，否则再调用该扩展方法。

2. 规则小结

- 扩展方法的优先级规则为：实例方法优先于扩展方法，命名空间内声明的扩展方法优先于命名空间外声明的扩展方法。
- 扩展可以继承，对 System.Object 的扩展将能被所有类型继承，例如：

```
static class MyExtensions
{
    // 扩展 System.Object
    public static void ShowType(this object o)
    {
        Console.WriteLine(o.GetType().ToString());
    }
}
```

在子类中可以继承该扩展方法，例如：

```
public static void Main()
{
    string str = "123abc";
    // 调用继承的扩展方法
    str.ShowType();
}
```

- 这种扩展，只支持对方法的扩展，而不支持属性、事件等。
- 扩展方法是 LINQ 实现的最基本条件，LINQ 中所有的查询操作符都是由扩展方法来定义的。
- 必须以 this 关键字标记扩展方法的第一个参数，且该参数不能为指针类型。
- 扩展方法本身必须被实现为静态方法，同时扩展方法必须被定义在非泛型静态类中，MyExtensions 类必须是静态且非泛型的。

13.2.8 查询表达式 (Query Expressions)

查询表达式就像高级语言中的 SQL 语句，实现了类似于关系化查询的语法集成，一般以 from 子句开头，以 select 或 group 子句结束，我们以一个实例来引入对查询表达式的理解：

```
class QueryExpression
{
    public static void Main()
    {
        List<User> users = new List<User>
        {
            new User{Name = "小王", Age = 27},
            new User{Name = "张三", Age = 32},
            new User{Name = "李四", Age = 15}
        };

        //查询表达式
        IEnumerable<User> selectUsers = from user in users
                                         where user.Age < 30
                                         orderby user.Age descending
                                         select user;

        foreach (User user in selectUsers)
        {
            Console.WriteLine(user.Name);
        }
    }
}
```

在上述实例，实现了从集合中找出年龄小于 30 的用户并将其降序排列，然后依次输出姓名。在 C# 3.0 中，查询表达式被翻译为方法调用，例如 where、select、orderby、group 等子句都对应了相应的 Where、Select、OrderBy、GroupBy 方法，而实际的处理工作由这些方法来完成。例如上述查询表达式实际被翻译为以下形式：

```
IEnumerable<User> selectUsers = users.Where(user => user.Age < 30).
    OrderByDescending(user => user.Age);
```

而这个翻译过程就是一个语法映射过程，这些方法既可以是查询对象的实例方法，也可以是扩展方法，方法的参数可以为委托或表达式树。将上述查询表达式翻译为 IL 语言，可以了解其语法映射的情况，限于篇幅，本节对此不做深入分析。

另外，对于 Lambda 表达式，本书已经在 9.7 节“一脉相承：委托、匿名方法和 Lambda 表达式”中做了相关的分析，本节不再做赘述。

13.2.9 结论

C# 3.0 的新技术琳琅满目，多为实现简洁、干净的代码创造了条件，而这些新特性很多是为了实现 LINQ 而服务的，我们在下一节关于 LINQ 的讨论中可以看到这些新特性的应用。因此，对于这些新的特性，有两点结论可以作为总结：

- 这些特性都是基于编译器的新特性，在 CLR 层并未提供新的实质内容。

- 这些特性基本都是为实现 LINQ 服务的，在平常的编程中也可以有选择的合理应用，也会有效提高编码效率，实现可读性较强的简洁代码。
- 新的一天已经来了，新的技术也来了，你还在等什么？

13.3 LINQ 体验

本节将介绍以下内容：

- LINQ 基本介绍
- LINQ 应用示例

13.3.1 引言

在.NET 3.0/3.5 一系列的新特性中，LINQ 无疑是最重量级的，使用类似于 SQL 的语法来查询任何类型的数据，无疑让人激动不已。打开 Visual Studio 2008 编译器，新生成的代码文件中会将 System.Linq 命名空间作为默认引用命名空间加入，足见 LINQ 的重量级地位，相信在未来程序设计中 LINQ 将会分担更重要的角色。

而.NET 在语言方面的新特性，则主要是为实现 LINQ 而做的准备，在认识这些特性的基础上，理解 LINQ 将变得容易。

13.3.2 LINQ 概览

LINQ 表示语言集成查询 (Language Integrated Query)，是.NET 集成于高级语言中的数据对象语言，为 CLR 提供了信息查询能力。LINQ 以类型安全的通用方式完成数据的基本操作，例如：查询、排序、汇总、对比等，而查询操作符允许作用于所有基于 IEnumerable<T> 接口的源，这些数据源可以是多种多样的，LINQ 目前支持对关系型数据、XML 以及内存中的数据集合等多种形式的支持。同样，LINQ 以这些新特性为基础来构建，而 CLR 并未对 LINQ 增加任何关联模块，编译器会自动完成对 LINQ 语句转换为 IL 兼容的语法映射，.NET 框架整合开发工具和高级语言来提供相关的功能和语法，这些特性正是我们上节介绍的语言新特性部分。

下面，我们从 LINQ 架构入手来了解其基本面貌（如图 13-3 所示）。

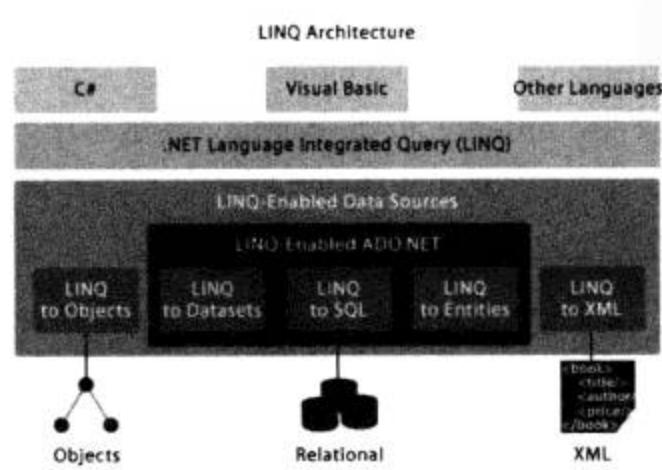


图 13-3 LINQ 架构（图片来源：MSDN）

由架构组成可知，.NET框架提供了5种形式的LINQ，主要是：

- LINQ to Objects，提供对内存中集合数据的实体对象映射。
- LINQ to Datasets，应用于DataSet。
- LINQ to SQL，应用于SQL Server数据库。
- LINQ to Entities，应用于SQL Server之外的关系型数据，并提供了其他数据库的扩展接口。
- LINQ to XML，应用于层次型XML数据。

其中 Datasets、SQL 或者是 Entities 数据被归结为关系型数据。你也可以通过使用 LINQ 的扩展框架，实现更多支持 LINQ 的数据源。

下面，以一个基本的 LINQ to Object 示例，来了解 LINQ 的强大功能，例如：

```
class Linq2Obj
{
    public static void Main()
    {
        List<User> users = new List<User>
        {
            new User{Name = "小王", Age = 27, UserInfo = new UserInfo{IsVIP = true,
Email = "xw@anytao.com"}},
            new User{Name = "张三", Age = 53, UserInfo = new UserInfo{IsVIP = false,
Email = "zs@msn.com"}},
            new User{Name = "李四", Age = 15, UserInfo = new UserInfo{IsVIP = true,
Email = "ls@live.com"}}
        };

        //实现数据对象的查询
        IEnumberable<User> selectUsers = from user in users
                                         where user.Age < 30
                                         orderby user.Age descending
                                         select user;

        foreach (var user in selectUsers)
        {
            Console.WriteLine(user.Name);
        }
    }
}
```

在本例中，通过查询表达式将年龄小于 30 的用户按照降序排列，并返回一个数据集存入 selectUsers 变量中，该数据集是一个实现了 IEnumberable<Anonymous Type>接口的对象。其中标准查询操作符用于完成相应的操作，where 表示查询条件，orderby 和 descending 表示排序方式，而 select 表示选择哪些字段组成变量。

另外，以基于方法的查询可以实现完全相同的功能，例如：

```
var selectUsers = users.Where(user => user.Age < 30)
                        .OrderByDescending(user => user.Age).Select(user => user);
```

限于篇幅，本节不对其他形式的 LINQ 做深入分析，读者在学习之余需要通过不断的实践来建立对 LINQ 的理解和对语法的掌握。

13.3.3 查询操作符

在表 13-1 中我们对常见的 LINQ 查询操作符，进行了简要的介绍，这些操作符是应用 LINQ 的基本要素，需要在不断的实践中掌握。

表 13-1 常见查询操作符说明

查询操作符	说 明
Where	表示查询
Select	表示选择哪些字段
OrderBy、OrderByDescending	按指定的表达式对集合排序，OrderBy 为升序排序，而 OrderByDescending 为降序排序
GroupBy	表示分组集合元素
Distinct	表示查询不重复的结果集，将重复元素过滤
Take	表示获取集合中的前 n 个元素
Skip	表示跳过集合的第 n 个元素
Join	表示对两个集合中键匹配的元素进行 inner join 操作
GroupJoin	表示对两个集合中键匹配的元素进行 grouped join 操作
Union	表示合并集合，并过滤相同的集合项
Concat	表示连接集合，而不过滤相同的集合项
Reverse	表示对集合反向排序
Count	表示集合中的元素个数
Max	获取集合中的最大值
Min	获取集合中的最小值
Sum	表示集合中的数值类型元素之和
Average	表示集合中的数值元素的平均值
First	获取集合的第一个元素
Last	获取集合的最后一个元素
Single	表示获取集合中与指定条件匹配的元素
Range	表示获取一个整数类型的集合
Repeat	表示获取一个指定次数的给定值的集合
Contains	判断集合中是否包含指定的元素
OfType	表示根据参数类型来获取数据集中对应的类型元素
Cast	表示将集合中的元素转换为指定的强类型
ToArray	表示将集合转换为数组
ToList	表示将集合转换为 List<T>泛型集合
ToDictionary	表示将集合转换为 Dictionary< TKey, TValue > 泛型集合

查询操作符是构成 LINQ 的基本元素，熟练地掌握这些基本查询操作符就能更好地完成相应的信息查询语句，对于提高 LINQ 的应用大有裨益。

13.3.4 LINQ to XML 示例

基于 XML 的数据处理越来越成为程序开发的必备内容，在此以 XML 文件的访问为例实现一个简单的

LINQ to XML 操作过程，从中体会 LINQ 在 XML 数据类型的应用规则。

首先，创建一个管理用户信息的 XML 文件，例如：

```
<?xml version="1.0" encoding="utf-8" ?>
<Users>
    <User>
        <Name>小王</Name>
        <Age>27</Age>
    </User>
    <User>
        <Name>张三</Name>
        <Age>53</Age>
    </User>
    <User>
        <Name>李四</Name>
        <Age>15</Age>
    </User>
</Users>
```

将其保存在可执行文件的同一目录下，然后对其执行类似于本节开始提供的 LINQ to Object 示例，例如：

```
class Linq2Xml
{
    public static void Main()
    {
        string filePath = AppDomain.CurrentDomain.BaseDirectory + "User.xml";

        //将 XML 加载到内存对象
        XElement xmlData = XElement.Parse(File.ReadAllText(filePath));

        //执行 LINQ 查询
        var users = from user in xmlData.Descendants("User")
                    where int.Parse(user.Element("Age").Value) < 30
                    orderby int.Parse(user.Element("Age").Value) descending
                    select new
                    {
                        Name = user.Element("Name").Value,
                        Age = user.Element("Age").Value
                    };

        //遍历查询结果
        foreach (var user in users)
        {
            Console.WriteLine(user.Name);
        }
    }
}
```

对于 XML 的加载，本节使用了框架类库提供的新 System.XML.Linq.XElement 类的 Parse 方法来实现，也可以应用 System.XML.Linq.XDocument 类的 Load 方法轻松地完成这一切。对于 LINQ to XML 的查询，如同对于内存中集合的操作处理一样，同样可以应用查询操作符来统一完成查询操作，应用极其灵活而简便。

13.3.5 规则

- LINQ 降低了数据处理的复杂性，以统一的方式来处理不同类型的数据的操作，简化了处理细节，有效提高了编码生产力。
- 以对象的角度来理解数据，促进对象关系映射技术的发展。
- 开发工具 Visual Studio2008 提供了良好的智能感知支持，同时与高级语言良好的集成，实现了强类型检查。
- 将面向对象的概念和动态语言的思想相融合，提高了编码质量。
- 通过使用 LINQ 的扩展框架，LINQ 可以支持更多的数据源。

13.3.6 结论

限于本书的选题角度和篇幅，对于 LINQ 的探索只能“浅尝”至此。LINQ 作为.NET 新特性中的重量级选手，将对未来的数据查询操作产生深远的影响。了解 LINQ 在不同数据类型方面的应用，是技术开发人员面临的又一个必选课题，本节作为对 LINQ 的启蒙性简介，仅为这一探索开了个头，我们还需要将这个旅程更进一步。

13.4 LINQ 江湖

本节将介绍以下内容：

- LINQ 的历史演义
- LINQ to Provider
- 基于 LINQ 的数据访问层实现

13.4.1 引言

LINQ 的江湖，水很深。

毋庸置疑，是 LINQ 掀起了.NET 江湖的惊涛骇浪。那么，什么是 LINQ？简单说，LINQ 就是 Language Integrated Query，这样的回答显得十足官方。对 C# 或者 VB.NET 而言，LINQ 是一系列的语言扩展，是语言层面支持类型安全的数据查询技术。

LINQ 是美丽的，LINQ 是有力的。它的美丽吸引开发者的眼球，写出更美的代码；它的有力解放开发者的双手，成就更好的实现。LINQ 从一开始诞生，就为.NET 世界打开了一扇窗口，从这个窗口看到了语言和数据间实现了前所未有的融合，一种面向对象语言和不同数据源在语言级别的融合，本文将逐层解析这种融合以及建立在融合之上的应用，如图 13-4 所示。

13.4.2 演义

LINQ 的历史，大概要追溯到 2005 年，随.NET 3.5 一起来到开发世界。更早之前，当时的一群微软人，

在茶余饭后思考当下的代码，总感觉不是那么优雅，所以他们想到了如何去改进现存的语法来适应更贴近的编程思想。回溯历史，来看一下 LINQ 的传奇是如何被天才演义的。从这个演义的历史，将逐渐看到技术的融合与创新，听到匿名类型、Lambda 表达式、类型初始化器（详见 13.2 节“赏析 C# 3.0”）这些早已熟知的名字，如何在演义的历史中接踵而来。好了，你现在早已对于下面的执行过程心领神会了：

```
public IList<Graduate> GetGraduates()
{
    var result = (from student in students
                  where student.AvgScore > 60
                  select new Graduate
                  {
                      Id = student.Id,
                      Name = student.FirstName + " " + student.LastName
                  }).ToList();

    return result;
}
```

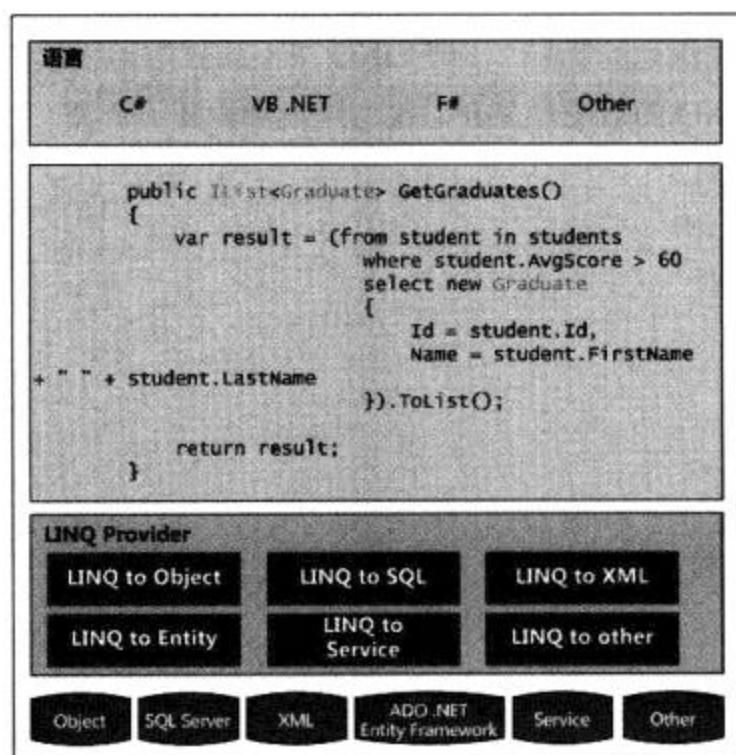


图 13-4 LINQ 与不同数据源的融合

你看，上面的代码和熟悉的 SQL 多么的近似，同时又“天然”地靠近了人类表达思维的方式，这种在代码中处理数据的方式，一定被画上里程碑式的礼赞。然而，要达到上述语法的目标，实际的语言推演过程历经曲折，首先应用查询操作符实现一个等价物：

```
public IList<Graduate> GetGraduates()
{
    var result =
        students.Where(student => student.AvgScore > 60)
        .Select(student => new Graduate
        {
            Id = student.Id,
            Name = student.FirstName + " " + student.LastName
        })
        .ToList();

    return result;
}
```

从IL层面而言，LINQ只是编译器玩儿的语法游戏，不管是查询表达式还是查询操作符，都将编译为相同的IL代码。

然而实际上，上述代码在C# 2.0时代，只能有非常“艰难”的实现，从理论上同样可以应用最基本的.NET语法实现IL基本相同的代码，以上的示例可以被写成：

```
public IList<Graduate> GetGraduates()
{
    IList<Graduate> result =
        Enumerable.ToList<Graduate>(
            Enumerable.Select<Student, Graduate>(
                Enumerable.Where<Student>(students,
                    delegate(Student s)
                    {
                        return s.AvgScore > 60;
                    }),
                    delegate(Student s)
                    {
                        Graduate s1 = new Graduate();
                        s1.Name = s.FirstName + " " + s.LastName;

                        return s1;
                    }
                )
            );
    return result;
}
```

显而易见，这种基本的实现方式晦涩难懂，需要“绞尽脑汁”去理解和实现。在13.2节“赏析C# 3.0”中已经领略了众多表现出色的新特性，现在就再以LINQ的角度回望这些新特性在LINQ语法中集体亮相的光芒。

1. Lambda表达式

在.NET 1.0时代，表达委托的方式，只能依托于最原始的Delegate来封装安全的函数指针，因此要实现类似于：

```
var result =
    students.Where(student => student.AvgScore > 60)
```

中对于条件的解析，代码实现会将大部分精力放在准备工作上：

```
// 01 定义委托
public delegate bool StudentDelegate(Student student);

// 02 实现执行逻辑
private bool IsQualified(Student s)
{
    return s.AvgScore > 60;
}

// 03 模拟实现静态的Where方法
public static class MyExtensions
{
    public static IList<Student> Where(IList<Student> source, StudentDelegate d)
    {
        IList<Student> result = new List<Student>();
```

```

        foreach (var item in source)
        {
            if (d(item))
            {
                result.Add(item);
            }
        }

        return result;
    }
}

// 04 应用委托实现的查询
public IList<Student> GetStudent(IList<Student> students)
{
    StudentDelegate d = new StudentDelegate(this.IsQualified);
    var result = MyExtensions.Where(students, d);
    return result;
}

```

上述的实现，已经完全淹没在“*How to*”的逻辑里，而丧失了“*Do What*”的本源。在.NET 2.0时代，匿名委托让这种情况有所好转，因此便有了稍微简单的表现：

```

public IList<Student> GetStudent(IList<Student> students)
{
    var result = Enumerable.Where(students,
        delegate(Student s)
    {
        return s.AvgScore > 60;
    }).ToList();

    return result;
}

```

除了更优雅地实现，匿名委托带来的另一个好处还有自动形成的闭包。不过，虽然这种进步已经让.NET在表现力方面有了超越的起点，然而正如在本文开始看到的以匿名委托实现的GetGraduates方法一样，还是“掺杂”了太多的“不必要”。

所以，到了.NET 3.0/3.5时代，闪耀环宇的Lambda诞生了，是它赋予了数据查询无与伦比的优雅：

```

var result = Enumerable.Where(students,
    (Student s) => { return s.AvgScore > 60; }).ToList();

```

更进一步借助于“类型推演”，上述实现可以彻底到：

```

var result = Enumerable.Where(students, s => s.AvgScore > 60);

```

从语义而言，GetGraduates要做的只是获取平均成绩大于60分的学生，而Lambda实现的就是这种“*Do What*”的简单逻辑，而不必考虑具体的“*How To*”执行。

关于Lambda更多的论述，详见9.7节“一脉相承：委托、匿名方法和Lambda表达式”。

2. 扩展方法

接下来，该试图抹去`IEnumerable<T>`频繁出现的身影，于是便有了扩展方法的闪亮登场。从一开始，对于数据查询的扩展都必须考虑对`IEnumerable<T>`的兼容性，语言的设计者必须在保证新的语言特性除了具有`Where`、`Select`等扩展操作之外，还要兼容`IEnumerable<T>`原有逻辑不受影响。

简单起见，可以通过静态方法来实现：

```
public static IEnumerable<T> Where<T>(IEnumerable<T> source, Func<T, bool> d)
{
    foreach (var item in source)
    {
        if (d(item))
        {
            yield return item;
        }
    }
}
```

所以，便有了

```
public IList<Student> GetStudent(IList<Student> students)
{
    var result = MyExtensions.Where(students, s => s.AvgScore > 60).ToList();

    return result;
}
```

这种实现已经非常切近优化的语法，但是对于参数 `IEnumerable<T>` 而言，仍“嫌”累赘。所以微软工程师引入了关键字 `this` 来告诉编译器，连这个也免了吧：

```
public static IEnumerable<TSource> Where<TSource>(this IEnumerable<TSource> source,
Func<TSource, bool> predicate);
```

最终，基于扩展方法的扩展，实现了无缝的衔接：

```
var result = students.Where(student => student.AvgScore > 60)
```

正如在 13.2 节“赏析 C# 3.0”中阐述的扩展方法不是专为 LINQ 而准备的，但它确实是实现 LINQ 必不可少的因素。而在实质上，扩展方法实现了通过实例类型变量访问静态方法的可能，这是其深层次上体现“扩展”二字的根本。

3. 对象初始化器

显而易见，如果没有对象初始化器的引入，在 `Select` 块中，只能以下面的方式来实现了：

```
var result = students.Select(s =>
{
    var g = new Graduate();
    g.Id = s.Id;
    g.Name = s.FirstName + " " + s.LastName;

    return g;
});
```

因为，很多时候不能完全“指望”通过类型构造器来实现对象的初始化任务：

```
var g = new Graduate(s.Id, s.FirstName + " " + s.LastName);
```

因为不知道要为属性准备多少相应的类型构造器，因此对象初始化器和集合初始化器便应运而生了：

```
var g = new Graduate { Id = s.Id, Name = s.FirstName + " " + s.LastName };
```

由此带来的好处便是在 LINQ 语句块中，可以非常方便地实现对象初始化的工作：

```
var result2 = students.Select(student => new Graduate
{
```

```

    Id = student.Id,
    Name = student.FirstName + " " + student.LastName
});

```

现在，一切变得异常人性化了，不过还有一些需要进一步完善的地方，所以继续演义吧。

4. 匿名类型

假如没有匿名类型，那么当需要从学生列表获取 Id 和 Name 信息时，需要显式地为 Id 和 Name 定义类型：

```

public class StudentIdName
{
    public int Id { get; set; }
    public string Name { get; set; }
}

public IEnumerable<StudentIdName> GetStudents(IList<Student> students)
{
    var result = students.Select(s => new StudentIdName
    {
        Id = s.Id,
        Name = s.Name
    });

    return result;
}

```

如果情况有变，又需要从学生列表返回学生及其班级信息时，上述定义的 StudentIdName 就无法满足要求，勤劳的开发者只能开始定义新的类型 StudentClass 来承载学生及其班级信息这一实体：

```

public class StudentClass
{
    public int Id { get; set; }
    public string Name { get; set; }
    public Class Class { get; set; }
}

public IEnumerable<StudentClass> GetStudentClass(IList<Student> students)
{
    var result = students.Select(s => new StudentClass
    {
        Id = s.Id,
        Name = s.Name,
        Class = GetClass(s.Id)
    });

    return result;
}

```

对于软件需求而言，常常感受“不变的只有变化”这一铁律的约束，所以疲劳不堪的程序开发者不得不为此实现各种需求的显式类型定义，就如同上述示例中的 StudentIdName 和 StudentClass 一样。为了解决临时类型定义的问题，语言设计者提出了新的特性，这就是匿名类型。以 Select 扩展方法而言：

```
public static IEnumerable<TResult> Select<TSource, TResult>(this IEnumerable<TSource> source, Func<TSource, TResult> selector);
```

因此，定义在 Select 选择器的类型参数 TResult 便可以通过匿名类型构造出“无限”多的可能性了：

```

var result1 = students.Select(s => new { Id = s.Id, Name = s.Name });
var result2 = students.Select(s => new { Id = s.Id, Age = s.Age });
var result3 = students.Select(s => new { Id = s.Id, Class = GetClass(s.Id) });
// .....其他的无限多可能性.....

```

一个很小的改善，换来无限优雅的语言优势。

5. 隐式类型变量和隐式类型数组

有了匿名类型带来的方便性，就必须在类型定义上为“类型决断”提供可能。举例而言，因为 LINQ 的灵活性，没办法为 Select 出来的新的匿名类型准备合适的 Type：

```
??? result = students.Select(student => new
{
    Id = student.Id,
    Desc = student.Name + ":" + student.Age
});
```

所以由编译器来承载这种决断是顺理成章的事情，不管是 `IEnumerable<T>` 还是 `IQueryable<T>`，都可以通过一个小小的 `var` 来兼容所有情况的发生：

```
var result = students.Select(student => new
{
    Id = student.Id,
    Desc = student.Name + ":" + student.Age
});
```

`var` 替换了`???`，而类型决断由编译器进行，现在隐式类型变量承载了任何可以由右表达式推断的类型了。

6. 查询表达式

通过匿名类型、隐式类型变量、对象初始化器还有 Lambda 表达式，LINQ 构造出了无比优雅的数据查询操作方式，通过对 `IEnumerable<T>` 的扩展实现 `Where`、`Select`、`OrderBy`、`Group`、`Skip` 和 `Take` 等查询操作符。

为了有更加符合人类口味的表现方式，语言设计者在查询操作符的基础上更进一步实现了查询表达式，这就是本文开始对于两种方式的对比。两种方式没有本质的区别，体现在 IL 层面上其实是完全相同的代码，所以选择的权利就取决于开发者自己的品味。

总体而言，这种类似于 SQL 表现方式的语言，毫无疑问地将 C# 和 VB.NET 带入了新的里程碑。

LINQ 的诞生，基本经历了一系列的演化历史，而.NET 语言也在这种发展中诞生了一个又一个激动人心的新技术和新概念，所以历史的演义，也是历史的探索。

13.4.3 基于 LINQ 的零代码数据访问层实现

LINQ 带来的变革，并非只为哗众取宠，不光是语法糖上的优雅，更多的是实现了生产力上的巨大提高。因此，就以下面的示例说明如何基于 LINQ 实现几乎零代码的数据访问层。关于数据访问层，有很多的模式适应不同的场合，而本文选择打造基于 Repository 模式和 ADO .NET Entity Framework（或者 LINQ to SQL）的数据访问层。

1. Repository 模式

早在《企业应用架构模式》一书中，Martin Fowler 大叔就对 Repository 模式进行了总结，在此借花献佛，对 Repository 做以简要的介绍，如图 13-5 所示，Repository 层为业务层提供了数据操作的基础架构，屏蔽了数据层实际的数据操作逻辑，封装了对于不同数据源、不同数据结构的包装。对于数据层而言，Repository

是数据的传递者和包装器，对于业务层而言，Repository是“虚拟”的数据和实体。

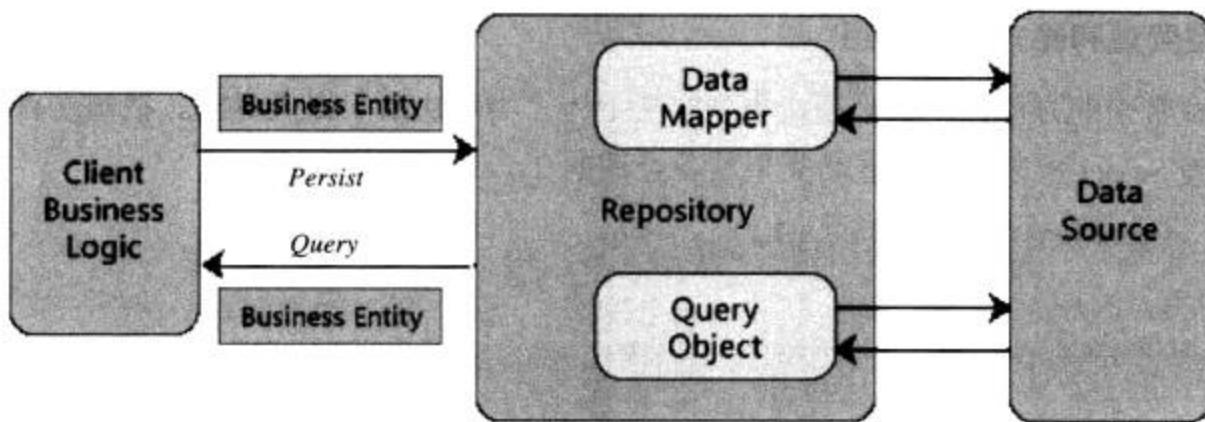


图 13-5 Repository 模式 (图片来源: MSDN)

在基于 LINQ 实现的 Repository 模式中，以 IRepositoryBase< TEntity > 建立对于数据的抽象的逻辑，主要包括：查询单条记录 Load、查询多条记录 Loads、添加新记录 Add、更新记录 Update 和删除记录 Delete。

```

public interface IRepositoryBase< TEntity > : IRepositoryBase where TEntity : EntityBase< TEntity >
{
    IQueryable< TEntity > Loads();
    IQueryable< TEntity > Loads(Expression< Func< TEntity, bool > > predicate);

    TEntity Load(Expression< Func< TEntity, bool > > predicate);

    void Add(TEntity entity);
    void Update(TEntity entity);
    void Delete(TEntity entity);
}
  
```

以此为基础，继续打造轻量型的数据访问层逻辑。具体的增删改查操作被封装在 RepositoryBase< TEntity > 类型中。以 Loads(Expression< Func< TEntity, bool > > predicate) 方法为例，通过获取对 ObjectContext 的包装调用 Entity Framework 查询表达式，进行对数据的增删改查操作：

```

public abstract class RepositoryBase< TEntity, TDataConnector > : RepositoryBase, IRepositoryBase< TEntity >
{
    where TEntity : EntityBase< TEntity >
    where TDataConnector : IDataConnector< TEntity >

    #region IRepositoryBase< TEntity > Members

    public IQueryable< TEntity > Loads(Expression< Func< TEntity, bool > > predicate)
    {
        using (var ctx = GetDataConnector< TDataConnector >())
        {
            return ctx.GetObjectQuery< TEntity >().Where< TEntity >(< predicate >);
        }
    }

    // 省略其他
    #endregion
}
  
```

借助于 Expression< Func< TEntity, bool > > 对于表达式的抽象和 Entity Framework 的 LINQ 支持，就可以非常方便地实现“按需”加载，减少数据的处理压力并提高业务层的灵活性，具体的体验在后文示例中继续。

当然，可以通过继承 IDataConnector< TEntity > 实现不同的数据库连接器，例如基于 LINQ to SQL 的数据连接器和基于 Entity Framework 的数据连接器：

```
public interface IDataConnector<T> : IDisposable
{
    IQueryable< TEntity > GetObjectQuery< TEntity >();
}
```

而具体的 DataConnector 由配置决定，封装在 GetDataConnector() 方法中。有了基础的 Repository 支持，就可以以此为基础，实现具体的 Repository 数据层了。

2. 零代码数据访问层

经过一系列基于 LINQ（此处为 LINQ to Entity）和 Expression Tree 的优化，最后的数据访问层实现了无限近似的零代码，以访问用户数据为例：

```
public class Member : EntityBase< Member >
{
}

public interface IMemberRepository : IRepositoryBase< Member >
{
}

public class MemberRepository : RepositoryBase< Member, IDataConnector< Member > >, IMemberRepository
{
}
```

由 MemberRepository 的实现来看，代码中只是简单地继承了 RepositoryBase< TEntity, TDataConnector > 基类，然后不做任何进一步的处理，业务层就可以通过 MemberRepository 实现对于 Member 实体的增删改查处理，完全解放了开发者在数据层的投入，从此望而生畏的 SQL 拼写任务就此画上句号，好日子就这样来了。

对于业务层而言，应用 Repository 数据访问层的方式显得无比轻松与高效：

```
public class MemberService
{
    IMemberRepository repository = new MemberRepository();

    /// <summary>
    /// Get member info by id
    /// </summary>
    /// <param name="id">Member's id</param>
    /// <returns></returns>
    public Member Get(int id)
    {
        return repository.Load(x => x.Id == id);
    }

    /// <summary>
    /// Verify member's info by email and password
    /// </summary>
    /// <param name="email">Member's email</param>
    /// <param name="password">Member's password</param>
    /// <returns></returns>
    public bool Verify(string email, string password)
    {
        var entity = repository.Load(x => x.Email == email && x.Password == password);
    }
}
```

```
        return entity != null ? true : false;
    }

    /// <summary>
    /// Register a new member
    /// </summary>
    /// <param name="model"></param>
    public void Register(Member model)
    {
        var entity = repository.Loads(x => x.Id == model.Id).FirstOrDefault();

        if (entity == null)
        {
            repository.Add(entity);
        }
    }

    // 省略其他
}
```

因此，在上述示例中，实现了基于 Repository 模式的数据访问层，并通过 LINQ 让这种数据访问层简单到“零”，从而更加印证了 LINQ 变革的力量。

！注意

关于示例 Repository 模式的基础支持代码，请访问本书在线支持站点 <http://book.anytao.net/inside>。

13.4.4 LINQ to Provider

扩展性，是 LINQ 最引人入胜的地方。通过扩展 LINQ 提供器，就可以实现自定义的 LINQ to ABC，为不同的数据源 XML、SQL、Cloud、Web Service 实现无限可能的 LINQ 查询支持。借助于扩展性，LINQ 就像永动机一样将触角扩展到 LINQ to World：LINQ to SQL、LINQ to XML、LINQ to Entity、LINQ to CRM、LINQ to Active Directory、LINQ to Google、LINQ to Flickr、LINQ to JSON、LINQ to Javascript、LINQ to MySQL、LINQ to Oracle、LINQ to Streams、LINQ to WMI、LINQ to NHibernate。

LINQ 扩展的核心抽象于两个看似简单的接口 `IQueryable<T>` 和 `IQueryProvider`：

```
public interface IQueryable<out T> : IEnumerable<T>, IQueryable, IEnumerable
{
}

public interface IQueryable : IEnumerable
{
    Type ElementType { get; }
    Expression Expression { get; }
    IQueryProvider Provider { get; }
}
```

其中，`Expression` 表示查询的表达式目录树，`IQueryable` 即是通过该表达式树表示查询运算符和方法调用，然后由 `Provider` 将其翻译为对应的查询语言。所以，`Expression` 和 `Provider` 是互为依托的：一个表示查询（`Expression`），一个翻译和执行查询（`Provider`）。所以，具体的查询操作由 `Provider` 来执行，那么继续了解一下 `IQueryProvider`：

```

public interface IQueryProvider
{
    IQueryable CreateQuery(Expression expression);
    IQueryable<TElement> CreateQuery<TElement>(Expression expression);
    object Execute(Expression expression);
    TResult Execute<TResult>(Expression expression);
}

```

简单来说，Provider 提供了两个重要的方法：CreateQuery 和 Execute。顾名思义，CreateQuery 是根据表达式树创建 IQueryable 查询实例，而 Execute 则用于实际的执行逻辑。对于自定义的 LINQ to Provider 来说，需要从实现 IQueryable<T>和 IQueryProvider 开始，而实现具体的自定义 Query 并非简单的过程，所以在此不详细论述相关内容，此处仅提供进入探索的起点。

需要特别留意的是，LINQ to Object 有别于其他的 LINQ 实现方式，因其查询数据源实现了 IEnumerable<T>接口的数据，所以查询操作符或方法调用本身是不需要翻译过程的，所以也不需要提供 LINQ to Object 的查询提供器 Provider。而其他 LINQ to Provider，例如 LINQ to SQL 需要以表达式树抽象 T-SQL 表达式，并以具体的 LINQ to SQL Provider 将表达式树翻译为具体的 T-SQL 并交由数据库服务器执行。

从 LINQ 扩展的实践来看，主要的应用场合主要体现在：

- 方便二次开发。为平台性产品提供支持 LINQ 查询的 SDK，例如 LINQ to CRM。
- 包装 Web API。例如 LINQ to Amazon、LINQ to Facebook、LINQ to Flickr 和 LINQ to Google。
- 改善对于不同数据源的 LINQ 查询方式，例如 LINQ to MySQL、LINQ to Oracle、LINQ to NHibernate。

13.4.5 结论

当 LINQ 走进.NET 世界，这里从此不同，语言的特性有了新层次的提升，.NET Coder 可以操着老式的武器（C#、VB.NET）挥舞在新的舞台。

对的。是一系列的语法糖成就了 LINQ 的集大成，在 13.2 节“赏析 C# 3.0”中分别对上述新的语言特性进行了详细的论述，而本文通过 LINQ 的历史演义将其融合。两相比较，不难发现 LINQ 就是集合了一系列语法糖而打造的新特性，但正是这些由小到大的完善，让.NET 语言逐步由命令式向声明式转变，而这种转变带来了巨变。

13.5 抢鲜 Visual Studio 2008

本节将介绍以下内容：

- Visual Studio 2008 概览
- Visual Studio 2008 新特性介绍
- Visual Studio 2008 开发示例

13.5.1 引言

工欲善其事，必先利其器。

伴随着.NET 3.0/3.5 高调入场，Visual Studio 2008 也按照惯例闪耀上市，基于.NET 蜂拥而至的新特性，

Visual Studio 2008 也相应提供了对这些新技术、新特性的支持，WCF、WPF、WF 和 Office 应用有效地融合在 Visual Studio 开发工具中，同时提供了相应的智能感知、工具箱控件和模板支持。

Visual Studio 2008 是面向.NET 应用程序最强大的开发利器，也是和每个.NET 开发者距离最近的编码工具，了解 Visual Studio 2008 正是应了那句老话：磨刀不误砍柴工。

13.5.2 Visual Studio 2008 概览

Visual Studio 2008 是功能强大的开发工具，支持开发.NET 下所有的应用开发，主要包括 Windows 应用、Web 应用、Office 应用、智能设备和其他服务，例如图 13-6 中罗列了 Visual Studio 2008 支持的项目类型的主要部分。

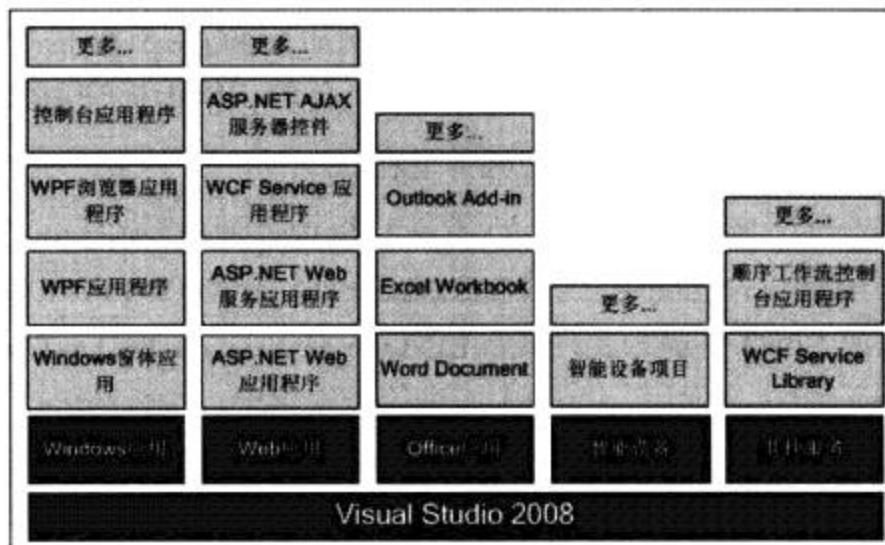


图 13-6 Visual Studio 2008 应用

图 13-7 所示的是一个熟悉的 WPF 应用程序开发 Visual Studio 2008 全景图，其中包含了最常用的工具箱、WPF 窗体设计器和 XAML 编辑器、解决方案资源管理器和属性管理器，以及菜单和工具栏等。



图 13-7 Visual Studio 2008 界面布局

13.5.3 新特性简介

Visual Studio 2008 除了支持 WPF、WCF 和 WF 等应用程序的开发，提供了对 Windows 应用、Web 应用和 Office 应用的诸多扩展，添加了相应的项目模板、智能感知、开发工具集等。除此之外，还包含以下几个主要的新特性：

- 多定向支持(Multi-Targeting Support)，实现了构建多个.NET 框架版本的引用，意味着你可以通过 Visual Studio 2008 实现基于不同.NET 版本的应用程序，而开发工具将根据不同的版本过滤智能感知、工具箱控件和模板。目前支持的.NET 版本为 2.0、3.0 和 3.5。
- JavaScript 智能感知和调试 (JavaScript Intellisense and Debugging)，支持 JavaScript 智能感知和调试，同时集成了 ASP.NET AJAX，也提供了同样完善的智能感知。
- Web 设计器和 CSS 支持 (Web Designer and CSS Support)，支持分割视图的编辑，同时提供了丰富的 CSS 集成，以及其他辅助的特性支持。
- 嵌套母版页 (Nested Master Page)，支持多层嵌套的母版页。
- Visual Studio2008 专业版提供单元测试支持。

13.5.4 开发示例

在此，以 WPF 应用程序的开发为例，说明完成一个.NET 应用程序的一般步骤，由此建立对 Visual Studio 2008 工具工作方式的一般理解：

首先，建立项目，在新建项目模板页，选择.NET Framework 3.5 版本，并选定项目为 WPF 应用程序，输入相应的项目名称和解决方案名称，选择合适的保存路径，如图 13-8 所示。



图 13-8 Visual Studio 2008——建立项目

单击“确定”按钮，进入开发界面，Visual Studio 2008 已经完成了一系列的初始化动作，新建的项目如图 13-9 所示。

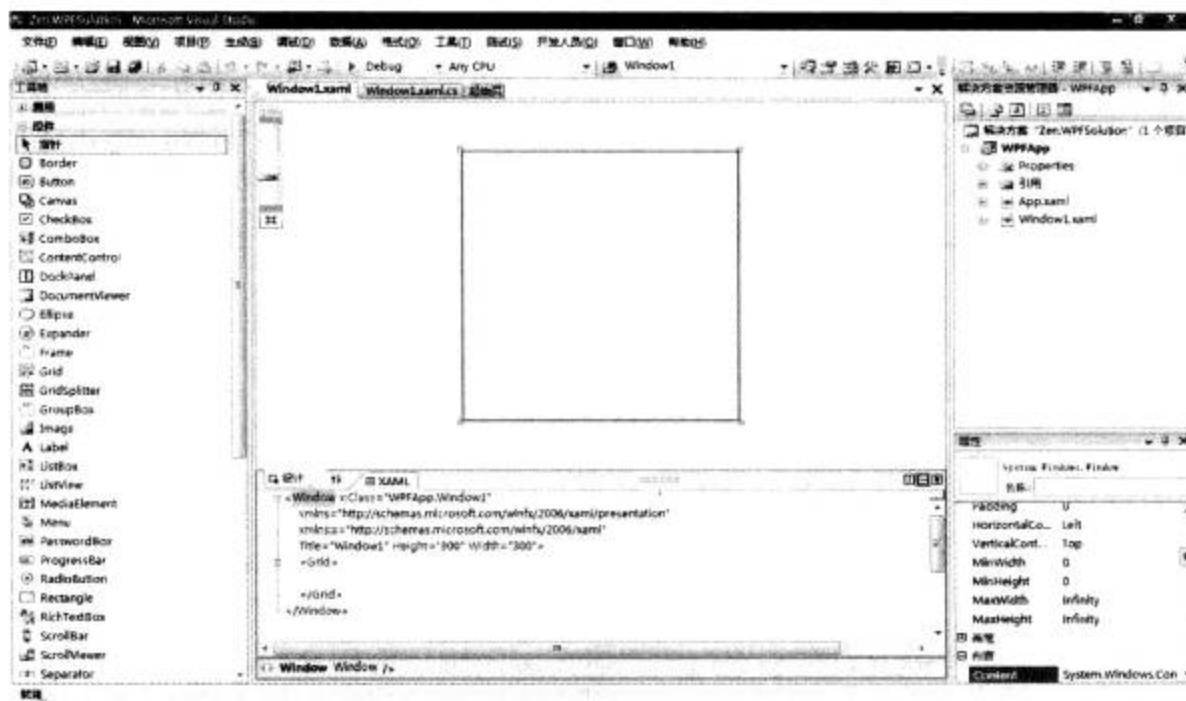


图 13-9 Visual Studio 2008——新建的 WPF 应用程序

然后，从工具栏拖入一个 Button 控件，调整其位置和大小，修改其显示名称，你可以通过在 WPF 窗体设计器中修改，也可以通过 XAML 进行编码修改，或者从属性页中选择相应的属性修改，如图 13-10 所示。



图 13-10 Visual Studio 2008——WPF 窗体设计器

然后双击 Button 按钮，打开代码编辑器，为其输入以下代码：

```
private void btnSayHello_Click(object sender, RoutedEventArgs e)
{
    MessageBox.Show("Hello, Visual Studio 2008.");
}
```

编译程序，将能生成一个 WPF 应用程序，通过简单的几步我们就可以创建一个完整的 WPF 应用示例，更复杂的程序都是从这个起点开始来构建的，所以了解了 Visual Studio 2008 的基本应用，也就能更好的开发优质的软件系统。

13.5.5 结论

Visual Studio 2008 是个重量级的.NET 开发工具，其中的特性和内容并非只言片语所能尽言，本节作为抢

鲜的初次接触，可以从基本面貌对 Visual Studio 2008 有个大致了解。实际的应用和熟悉，是建立在对.NET 基本知识的理解上，并不断在项目中实践。有了深厚的内功，才能用好锋利的刀剑，剑鞘在手，你必须运用自如，才能游刃有余。

Visual Studio 2008 就是你手中那柄所向披靡的剑，关键看用剑的人如何挥舞了。

13.6 江湖一统：WPF、WCF、WF

本节将介绍以下内容：

- 新基础框架简介
- WPF、WCF 和 WF 分析

13.6.1 引言

.NET 3.0 的最大改变就是在.NET 2.0 基础上，加入了几个重量级的基础框架，主要包括 WPF、WCF、WF 和 WCS，如图 13-11 所示。这些新的框架为不同领域的应用提供了基于.NET 的统一解决方案：WPF 提供了处理用户界面的和可视化元素的统一技术基础；WCF 实现了面向服务的应用，有效构建分布式系统的无缝通信；WF 支持广泛的业务执行过程，提供了工作流引擎支持；WCS 为个人信息管理实现了统一而简单的解决方案，个人信息可以以统一的标识畅游于互联网，而不必担忧安全问题。

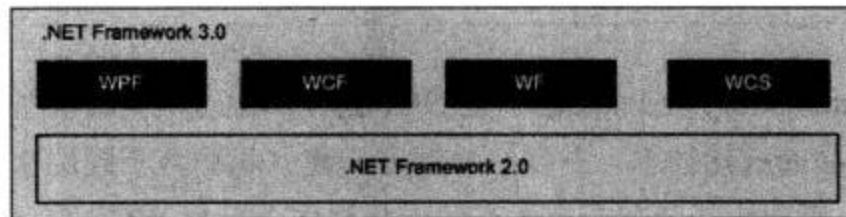


图 13-11 .NET 3.0 框架

这几个新的技术，微软以统一江湖的气势，力求在相关领域实现一致的处理方式，提供通用的处理架构，提高软件开发的效率。本节就以尝鲜的思路来介绍关于 WPF、WCF 和 WF 的基本内容。

13.6.2 WPF

WPF（Windows Presentation Foundation），是用于 Windows 的标准表现层框架，它由一个 Display Engine 和一系列的托管类组成，用于创建表现丰富的界面元素，这些类主要被定义在 System.Windows 及其次级命名空间中，主要包括：System.Windows.Controls、System.Windows.Data、System.Windows.Documents、System.Windows.Media、System.Windows.Shapes 等，分别用于支持界面数据绑定、文档、图形、图像、动画和媒体元素等。由此可见，WPF 实现的是将不同的界面风格，不同的技术元素，整个为一个统一的处理框架，提供一致的技术基础，这正是 WPF 为用户体验带来的震撼整合。

同时，WPF 基于 XAML（可扩展应用程序标记语言）来定义界面元素，例如下面是典型 WPF 窗体（图 13-12）及其 XAML 描述：



```
<Window x:Class="WPFApp.Window1" xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="Window1" Height="300" Width="300">
    <Grid>
        <Button Margin="53,121,25,91" Name="button1" Height="50" Width="200" Click="button1_Click">Hello world</Button>
    </Grid>
</Window>
```

图 13-12 WPF 窗体设计器

XAML 中的元素可以直接映射为 WPF 提供的类和属性，例如上述 Button 可以映射为：

```
Button button1 = new Button();
button1.Content = "Hello world";
button1.Width = 200;
button1.Height = 50;
```

这样的好处是融合了浏览器界面和 Windows Form 界面的优点，分离了界面元素和业务逻辑，设计师可以专注于界面美化，而开发者则只关注业务逻辑和功能需求，这一切在 WPF 架构下变得容易，设计师应用 Expression 工具设计的产品生成相应的 XAML 文件，开发人员可以将这些文件导入到 Visual Studio 直接进行开发，并且可以使用任何支持 CLR 的语言进行编码。

13.6.3 WCF

WCF (Windows Communication Foundation)，是.NET 建立和运行面向服务的应用程序的统一框架。WCF 整合微软所有开发分布式产品的成熟技术，并以统一的方式建立起适合不同层次需要的分布式架构，可以说 WCF 是这些技术的合成品，梳理这些分布式技术有助于从整体上把握 WCF 究竟是什么这个问题。

- .NET Remoting，用于.NET 应用程序之间的通信，可以使用不同的传输协议进行通信，可以寄宿于任何一种托管环境中，因此.NET Remoting 技术是托管环境中最佳的分布式通信选择。
- XML Web Services，提供了基于 SOAP 的交互通信，支持跨平台系统的集成，是目前使用最广泛的分布式技术。
- Enterprise Services，在.NET 组件中支持 COM+ 服务，支持可扩展的事务性应用程序。
- Message Queuing（消息队列），通过 MSMQ（Microsoft Message Queuing）支持队列消息，可以在连接断开的环境中使用。
- WSE（Web Services Enhancements），支持 WS-* Specification 新规范，解决了 XML Web Services 在安全性等方面的问题。

由不同的分布技术应用导致了不同的解决方案，因此，需要整合不同的技术以提供统一的分布式架构，建立单一的应用程序通信基础。WCF 正是在这种环境和基础下诞生的，它集上述各种分布式技术的优点于一身，支持 TCP、HTTP 协议，可以寄宿于 ASP.NET、EXE、COM+、WPF、NT Service、Console Application 等多种平台，提供了 Kerberos、X509、SAML 等多种安全模式。可以从 WCF 的体系结构来了解其基本的面貌（图 13-13）：

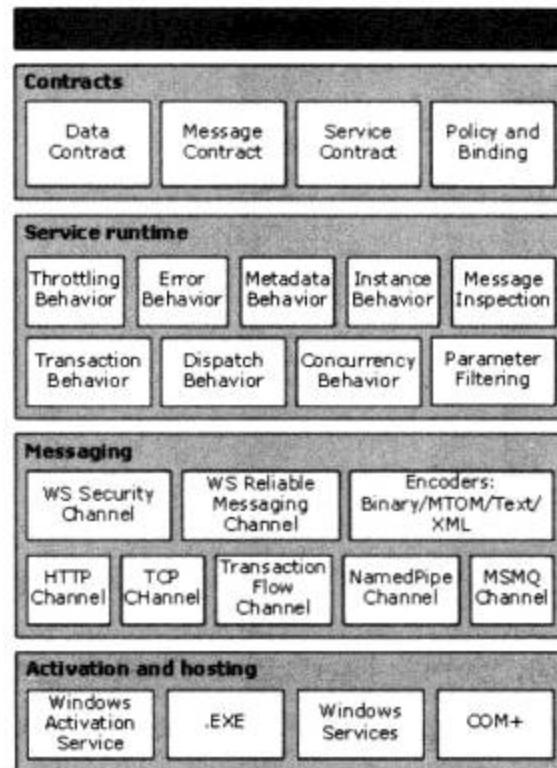


图 13-13 WCF 体系结构（图片来源：MSDN）

有了 WCF，开发人员无须为不同的通信应用不同的编程接口，而以通用的 API 来支持复杂的分布式需求，使得基于 SOA（Service-Oriented Architecture）的分布式系统开发变得简单而可靠。

在 WCF 中，各个应用的通信是由 Endpoints 来实现，客户端通过交换 Endpoints 进行通信，一个 Endpoints 主要由三部分组成：Address、Binding 和 Contract，其中 Address 一般为 Uri 或 Identity 来标识 Endpoint 的位置；Binding 实现了服务器端和客户端之间的通信；Contract 就是 Endpoint 与外部世界通信的一组操作。因此 WCF 的编程基础就是为 WCF Services 定义 Endpoint，而 Client 端的请求正是以 Endpoint 来进行通信的。

13.6.4 WF

WF（Windows Workflow Foundation），是一个企业级的工作流开发框架，适用于 Windows 的通用工作流技术，提供了基于工作流应用程序的统一构建平台，在 Microsoft Office SharePoint Server、Biztalk 和 Windows SharePoint Services 中就应用了 WF 作为工作流引擎。

简单地说，工作流就是一组活动，而这组活动最终由 WF 运行时引擎来执行，保持工作流状态，跟踪工作流执行等等。目前，WF 支持两种形式的工作流：

- 顺序工作流，包含分支、循环和其他控制结构的工作流。
- 状态机工作流，活动由当前的状态和收到的事件来决定执行。

WF 提供了可视化的工作流设计器和基本活动程序库（BAL，Basic Activity Library），其中流程设计器是 Visual Studio 托管的用于构建工作流的图形工具，而 BAL 是 WF 定义好的基本活动集，主要包括：While、IfElse、Code、State、Policy 等，下面我们以一个简单的用户登录的控制台程序为例，来演示一个简单的工作流程序：首先创建一个“顺序工作流控制台应用程序”，你可以从工具箱中将 BAL 直接拖入流程设计器来设计一个工作流程，如图 13-14 所示。

在此，设计一个简单用户验证流程活动，如图 13-15 所示。

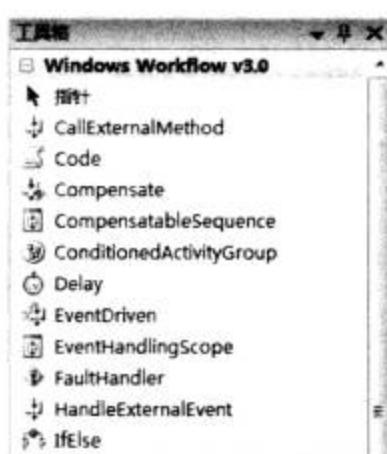


图 13-14 WF---BAL 工具箱

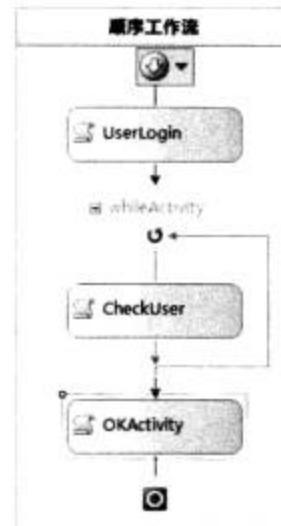


图 13-15 WF——顺序工作流

在该流程中，定义了 3 个 Code 活动，分别为 UserLogin、CheckUser 和 OKActivity；还有一个 While 活动，为 whileActivitiy。其中 whileActivitiy 的 Condition 被实现为代码条件 CheckValidation，其定义为：

```

private void CheckValidation(object sender, ConditionalEventArgs e)
{
    e.Result = UserName == "小王";
}
  
```

而 UserLogin 活动中，是不同的用户登陆模拟场景，具体实现为：

```

private void UserLogin_ExecuteCode(object sender, EventArgs e)
{
    Console.WriteLine("请输入用户名：");
    UserName = Console.ReadLine();
}
  
```

如果验证成功，则会执行 CheckUser 活动中的流程，而如果不成功则继续执行 OKActivity 活动，直到结束。

13.6.5 结论

三大基础框架的引入，使得.NET 构建了对用户界面技术、分布式技术和工作流的统一支持，这些整合体现了.NET 一统江湖的趋势和优势。例如 WPF 实现了对不同用户体验元素的技术融合；WCF 框架提供了跨平台、互操作和安全可信的 SOA 分布式应用；而 WF 实现了基于运行时的工作流引擎。

当这种技术整合越来越统一的时候，对于开发人员来说，就极大地降低了开发的难度和复杂度，在一统江湖的平台下，畅游在各种不同技术环境下的应用不再仅仅是梦想。

参考文献

David Chappell. Introducing Windows Communication Foundation

Microsoft. C# Version 3.0 Specification

Justin Smith. Inside Microsoft Windows Communication Foundation

Scott Guthrie. Using LINQ to XML,

<http://weblogs.asp.net/scottgu/archive/2007/08/07/using-linq-to-xml-and-how-to-build-a-custom-rss-feed-reader-with-it.aspx>

蒋金楠. 深入理解 C# 3.x 的新特性 (2). Extension Method - Part II, <http://www.cnblogs.com/artechn/archive/2007/07/19/823847.html>

Scott Guthrie. Visual Studio 2008 and .NET 3.5 Released, <http://weblogs.asp.net/scottgu/archive/2007/11/19/visual-studio-2008-and-net-3-5-released.aspx>

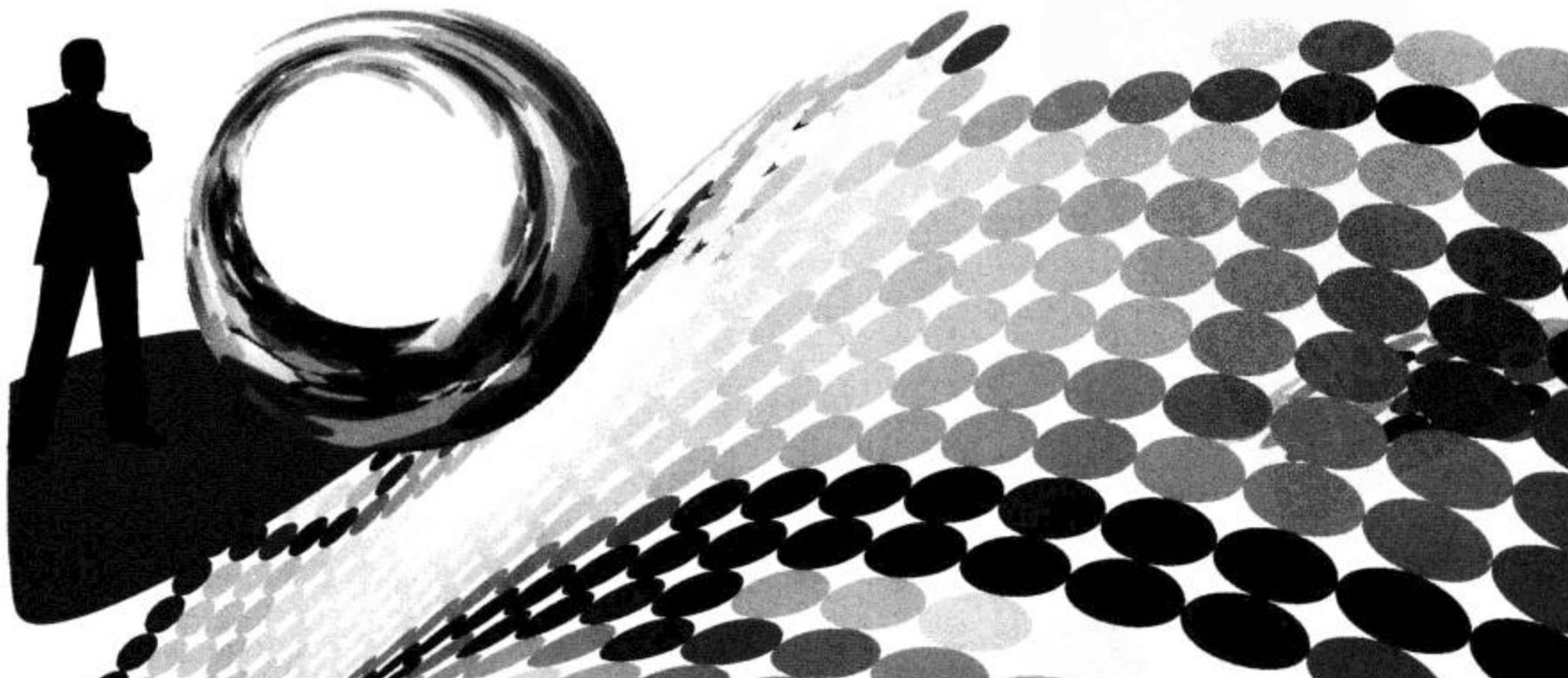
Anson Horton. The Evolution Of LINQ And Its Impact On The Design Of C#, <http://msdn.microsoft.com/en-us/magazine/cc163400.aspx>

Martin Fowler, 企业应用架构模式, 人民邮电出版社

Fabrice Marguerie、Steve Eichert、Jim Wooley, LINQ in Action, Manning Publications

第14章 跟随.NET 4.0脚步

14.1 .NET十年 / 473	14.5 命名参数和可选参数 / 497
14.1.1 引言 / 473	14.5.1 引言 / 497
14.1.2 历史脚步 / 473	14.5.2 一览究竟 / 498
14.1.3 未来之变 / 477	14.5.3 简单应用 / 499
14.1.4 结论 / 479	14.5.4 结论 / 499
14.2 .NET 4.0, 第一眼 / 480	14.6 协变与逆变 / 500
14.2.1 引言 / 480	14.6.1 引言 / 500
14.2.2 第一眼 / 481	14.6.2 概念解析 / 500
14.2.3 结论 / 484	14.6.3 深入 / 502
14.3 动态变革: dynamic / 484	14.6.4 结论 / 504
14.3.1 引言 / 484	14.7 Lazy<T>点滴 / 504
14.3.2 初探 / 485	14.7.1 引言 / 505
14.3.3 本质: DLR / 485	14.7.2 延迟加载 / 505
14.3.4 PK 解惑 / 488	14.7.3 Lazy<T>登场 / 505
14.3.5 应用: 动态编程 / 490	14.7.4 Lazy<T>本质 / 507
14.3.6 结论 / 491	14.7.5 结论 / 509
14.4 趋势必行, 并行计算 / 491	14.8 Tuple —— / 509
14.4.1 引言 / 491	14.8.1 引言 / 509
14.4.2 拥抱并行 / 492	14.8.2 Tuple 为何物 / 510
14.4.3 TPL / 493	14.8.3 Tuple Inside / 511
14.4.4 PLINQ / 495	14.8.4 优略之间 / 513
14.4.5 并行补遗 / 496	14.8.5 结论 / 514
14.4.6 结论 / 497	参考文献 / 514



14.1 .NET 十年

本节将介绍以下内容：

- .NET 历史之旅
- .NET 4.0 接触

14.1.1 引言

语言是程序开发者行走江湖的手上利器，各大门派的高手在论坛、博客、微博为了自家门派争吵不已早已是技术世界中的靓丽风景，虽多少为刚刚踏入江湖的新手提供了思考的素材，但也同时迷惑了初出茅庐者的前行方向。

本文不欲计较门派的高下，旨在明辨技术的真谛，以.NET 平台下的开发利器 C#为主线来梳理.NET 十年的发展、进化和发展，并从其变迁的进程中对于技术发展把玩一番。

14.1.2 历史脚步

.NET 已是豆蔻之年了。

这个日期是从 Anders Hejlsberg 在 1998 年组建 C# 团队开始算起的，掐指算来已是十年有余。作为.NET 平台下的静态强类型语言，在过去十多年的发展历程中披荆斩棘，已经逐渐成为应用开发语言中的佼佼者。从 TIOBE 开发语言排行榜的最新统计来看，C#、VB.NET 位居第四位和第七位，成为开发语言市场的顶级产品。

表 14-1 TIOBE 开发语言排行榜（2011 年 4 月）

2011-4 位置	2008-12 位置	2007-12 位置	开发语言	占有率
1	1	1	Java	19.043%
2	2	2	C	16.162%
3	3	5	C++	9.225 %
4	6	8	C#	7.185%
5	4	4	PHP	6.584%
6	7	6	Python	4.931%
7	5	3	(Visual) Basic	4.682%
8			Objective-C	4.386%
9	8	7	Perl	1.991%
10	9	10	JavaScript	1.513%

因此，在我们讲解历史时，将 C# 发展历史中的里程碑作以标记，来感受一下.NET 这十多年的发展历程，为每位开发者在心底搭起一座通往未来的桥梁，如图 14-1 所示。

下面我们分别从不同的历史阶段来了解 C# 语言的变迁，并讨论在每个变迁的里程碑上值得品味的闪光点，由此为未来的发展打一个基础。

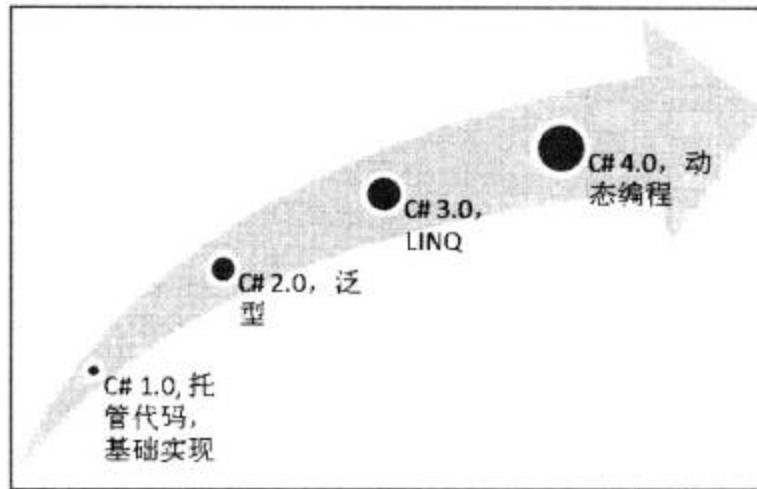


图 14-1 C# 的发展历史表

1. C#1.0, 从诞生到立足

2000 年之夏，微软大当家——盖茨先生着手战略调整，微软大刀阔斧地进行了技术改革与创新，并由此为世界带来一个新的名词，这就是.NET。那么什么是.NET 呢？在.NET 这一概念诞生之初，连微软本身都充满了定位的错乱和迷茫，以至于一时之间处处皆为.NET，大有一统江湖之势。随着.NET 平台的逐渐完善，概念和定位上的混乱已经日渐清晰，作为这场变革中的一项重要内容，一种全新的、能够适应.NET 平台特性的高级语言 C# 也随之诞生。这一任务理所当然地落在了 Anders Hejlsberg 的身上，作为 Delphi 之父，大师级的 Anders 从 1998 年 11 月开始领导了他的小组为这个世界带来全新的语言宠儿，这就是 C#。

初出茅庐的 C#，就像羞羞答答的小姑娘，步步留心、时时在意，学着他人的模样，生怕在前辈面前丢脸。当她以全新的姿态出现在万千程序开发者眼前时，其兼取百家之长、优雅简洁之态、摒弃复杂之弊的特性，立时令所有的观望折服。在迎来所有目光关注的同时，迅速成为高级语言战场的骄子。

那么，C# 诞生之初兼取百家之长，具体都有哪些优秀品质呢？总结起来主要体现在以下三个方面。

- 面向对象编程。C# 实现对属性、事件、委托、方法、索引器、构造器的全面支持，为面向对象的封装、继承、多态和接口提供了语言级别的支持。以继承为例，C# 支持单实现继承和多接口继承，摒弃了 C++ 中多继承带来的复杂性。
- 跨平台运行时支持。CLR 是.NET 平台下应用程序的通用语言运行时，是 C# 程序赖以生存的跨平台环境，因此 C# 具有了.NET 平台语言的所有优势，通用类型系统、自动内存管理、统一异常处理、完全的 FCL 访问权，都成为 C# 无与伦比的优势所在。
- 不断发展的语言特性。在十年的发展史上，.NET 不断兼容并包地由学习到被学习的定位下发展，取得不断完善与进步。基于 LINQ 的函数式编程、基于 DLR 的动态运行时支持以及基于 TPL 的并行计算，都是.NET 在语言特性上不断学习和超越的表现，而且这种超越还在继续。

2. C# 2.0，变革之作

C# 2.0 是一次完善和补充，也是一次变革与重生。在 2.0 中引入了诸多的语言特性，完善了 1.0 基础上的某些不足，例如匿名方法将代码放在委托而无须创建新的方法，详见 9.7 节“一脉相承：委托、匿名方法和 Lambda 表达式”；可空类型实现了对值类型的 null 操作，详见 7.4 节“认识全面的 null”；而部分类将一个类分解到多个类文件中。这些补充和完善，总结起来主要包括：

- 匿名方法
- 可空类型
- 部分类
- 迭代器
- 泛型

其中，泛型支持是 C# 2.0 的重中之重，.NET 框架从 CLR 级别实现了对泛型的支持，提供专门的 IL 指令支持泛型操作，同时配合 C# 语言机制构造一种全新的编程结构，实现了对类型抽象化的通用处理方式，这就是算法重用。

以最简单的交换数据为例，我们来了解一下泛型带来的好处：

```
public static void Swap(ref int x, ref int y)
{
    int temp = x;
    x = y;
    y = temp;
}

public static void Swap(ref string str1, ref string str2)
{
    string temp = str1;
    str1 = str2;
    str2 = temp;
}
```

以上两种算法分别实现了对整型、字符串型数据的交换，在泛型引入之前实现相同的交换算法需要分别为所有的类型构建相似的处理逻辑，这显然是一种代码上的浪费，而泛型特性彻底解决了这一问题：

```
public static void Swap<T>(ref T t1, ref T t2)
{
    T temp = t1;
    t1 = t2;
    t2 = temp;
}
```

对于不同的类型，在运行时以实际类型对 T 占位符进行替换，并转换为本地代码，彻底实现了灵活的类型抽象和算法重用。同时，除了代码级别的重用好处，泛型解决了类型转换、装箱与拆箱、类型安全等诸多问题，为程序设计带来巨大的变革。

更详细的解析，请参考第 11 章“接触泛型”的论述。

3. C# 3.0，涅槃与重生

C# 3.0 是语言发展历史上的里程碑变革，就像凤凰涅槃一般，为 C# 语言注入强大编程体验和活力，简洁、干净、富有意义，这些特性中主要包括：

- 匿名类型
- 自动属性
- 对象初始化器
- 集合初始化器
- 隐式类型变量和隐式类型数组
- 扩展方法
- 查询表达式

除此之外，C# 3.0 的最大亮点就是 LINQ (Language Integrated Query，语言集成查询)，在 CLR 中集成类似于 SQL 式的数据查询能力，一种前所未有的函数式编程体验在面向对象语言中得以大展拳脚，这不得不说是 C# 带来的超酷体验。

本节无法在有限的篇幅来展现优雅的代码，只能取一瓢饮之，并通过简要的对比来领略 C# 3.0 的强大功能，关于 3.0 新特性请参考第 13 章“走向.NET 3.0/3.5 变革”对这些新特性的论述。以对象初始化器为例，在 2.0 时初始化对象成员，我们以这种方式实现：

```
User user = new User();
user.Name = "小王";
user.Age = 28;
```

在 3.0 中，实现对象初始化有了更好的解决方案，由对象初始化器来完成：

```
User user = new User { Name = "小王" , Age = 28};
```

同样的方式可以用来实现集合的初始化和隐式类型的初始化等。

接着，再来了解一下自动属性带来的语法魅力，在 3.0 之前进行属性封装的确是一件麻烦的事情，例如：

```
class User
{
    private string name;
    public string Name
    {
        get { return name; }
    }

    private int age;
    public int Age
    {
        get { return age; }
        set { age = value; }
    }
}
```

而自动属性将这一切化简为无形，语法简单而功能依旧：

```
public class User
{
    public string Name { get; private set; }
    public int Age { get; set; }
}
```

所有的这些新特性最终都为了一个共同的目标而铺垫，这就是 C# 3.0 中最重量级的新特性：LINQ。LINQ 在代码级别实现类似于 SQL 式的查询语法，以类型安全的通用方式完成增、删、改、查等数据操作的基本方式。只要数据源基于 IEnumarable<T> 接口而实现，那么不管是关系型数据、内存中集合还是 XML 都可以作为 LINQ 查询对象进行数据处理，这为面向对象语言实现函数式的编程体验创造了条件，一种全新的编程风格为编程体验刮来一股旋风：

```
public static void Main()
{
    List<User> users = new List<User>
    {
        new User{Name = "小王", Age = 27},
```

```

    new User{Name = "小张", Age = 29},
    new User{Name = "小李", Age = 23}
};

IEnumerable<User> result = from user in users
                           where user.Age < 30
                           orderby user.Age descending
                           select user;

foreach (var user in result)
{
    //执行操作
}
}

```

你看，这种体验果然非同凡响，优雅而简单，没有辜负 Anders 对于代码美学的追求。而作为编程用户，我们同样体味了这种理念在功能和结构上的双重精彩。

14.1.3 未来之变

无论如何，.NET 4.0 已经在叩开 2010 新年的大门之时，以高调的姿态迎来一片掌声，广大的技术爱好者再次感受到.NET 发展中的又一变革。随着.NET 4.0 在 2009 年的发布，我们对于 C# 4.0 的关注也将与日俱增。总体而言，C# 4.0 的重头戏主要着眼在以下几个方面：

- 动态编程
- 并行计算
- 后期绑定
- 协变与逆变

废话少说，接下来我们一一领略 C# 4.0 中的语言特性。

1. 动态编程

众所周知，C# 是静态强类型语言。而在很多情况下，提供“动态”行为，是常常发生的事情，例如通过反射在运行时访问.NET 类型、调用动态语言对象、访问 COM 对象等，都无法以静态类型来获取。因此，C# 4.0 引入的又一个全新的关键字 dynamic，也同时引入了改善静态类型与动态对象的交互能力，这就是动态查找（Dynamic Lookup），例如：

```

dynamic d = GetDynamicObject();
d.MyMethod(22);           // 方法调用
d.A = d.B;                // 属性赋值
d["one"] = d["two"];       // 索引器赋值
int i = d + 100;           // 运算符调用
string s = d(1,2);         // 委托调用

```

详细的讨论，见 14.3 节“动态变革：dynamic”。

2. 并行计算

并行计算的出现，是计算机科学发展的必然结果，随着计算机硬件的迅猛发展，在多核处理器上工作已经是既存事实，而传统的编程模式必须兼容新的硬件环境才能使计算机性能达到合理的效果。用 Anders

大师的话说：未来5到10年，并行计算将成为主流编程语言不可忽视的方向，而4.0为C#打响了实现并发的第一枪。

那么，体验一下应用C#武器来开发并发环境下的超酷感受，System.Threading.Parallel静态类提供了三个重要的方法For、Foreach、Invoke可以为我们小试牛刀：

```
//应用TPL，执行并行循环任务
Parallel.For(0, 10, i =>
{
    DoSomething(i);
});
```

在线程争用执行情况下，相同的操作在双核平台下运行，以StopWatch进行精确时间测试，并行环境下的执行时间为2001ms，而非并行环境下的执行时间为4500ms，平行运算的魅力果然名不虚传。

再接再厉应用PLINQ执行对于并行运算的查询、排序等操作，当前PLINQ支持两种方式ParallelEnumerable类和ParallelQuery类，例如：

```
int[] data = new int[] { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };
int[] selected = (from x in data.AsParallel()
                  select x + 1).ToArray();
```

更详细的对比示例将在本章后续内容中详细讨论。并行计算为托管代码在多核环境下的性能优化提供了统一的解决方案，而未来我们会做得更好。

详细的讨论，见14.4节“趋势必行，并行计算”。

3. 协变和逆变

协变和逆变，是为解决问题而生的。而要理清解决什么样的问题，需要从理清几个简单的概念开始。首先我们进行一点操作：

```
Derived d = new Derived();
Base b = d;
```

Derived类型继承自Based类型，由Derived引用可以安全地转换为Based引用，而这种转换能力可以无缝地实现在Derived数组和Base数组，例如：

```
Derived[] ds = new Derived[5];
Base[] bs = ds;
```

这种原始转换（由子类转换为父类）方向相同的可变性，被称为协变（covariant）；其反向操作则被称为逆变（contravariant）。当同样的情形应用于泛型时，例如：

```
List<Derived> ds = new List<Derived>();
List<Base> bs = ds;
```

类似的操作却是行不通的。所以，这在C#4.0中得到了完善——泛型的协变与逆变：

```
List<Base> bs = new List<Base>();
List<Derived> ds = new List<Derived>();

bs = ds; //List<T>支持对T协变
ds = bs; //List<T>支持对T逆变
```

而在C#4.0中，伴随着协变与逆变特性的加入，C#引入两个关键字in和out来解决问题。

```

public interface ICovariant<out T>
{
    T MyAction();
}

public interface IContravariant<in T>
{
    void MyAction(T arg);
}

```

其中，`out` 表示仅能作为返回值的类型参数，而 `in` 表示仅能作为参数的类型参数，不过一个接口可以既有 `out` 又有 `in`，因此既可以支持协变、支持逆变，也可以同时支持，例如：

```

public interface IBoth<out U, in V>
{
}

```

详细的讨论，见 14.6 节“协变与逆变”。

4. 命名参数和可选参数

命名参数和可选参数是两个比较简单的特性，对于熟悉其他编程语言的开发者来说，可选参数并不陌生，为参数提供默认值就是可选参数：

```

public void MyMethod(int x, int y = 10, int z = 100)
{
}

```

因此，我们可以通过调用 `MyMethod(1)`、`MyMethod(1, 2)` 方式来调用 `MyMethod` 方法。而命名参数解决的是传递实参时避免因为省去默认参数造成的重载问题，例如省去第二个参数 `y` 调用时，即可通过声明参数名称的方式来传递：

```
MyMethod(20, z: 200);
```

相当于调用 `MyMethod(20, 10, 200)`，非常类似于 Attribute 的调用方式。虽然只是小技巧，但也同时改善了方法重载的灵活性和适配性，体现了 C# 语言日趋完美的发展轨迹。

详细的讨论，见 14.5 节“命名参数和可选参数”。

当然，除此之外.NET 4.0 还增加了很多值得期待的平台特性，这也将为 C# 编码带来前所未有的新体验，我们将在本章后续内容中继续阐述。

14.1.4 结论

预测未来，在技术世界是常有的事儿。从高级语言的发展历史来看，编程世界从来就没有停止过脚步，变革时时发生、创新处处存在。以技术人员的角度来观摩未来，带着 C# 4.0 的脚步来看展望，除了在函数式编程、并行计算和动态特性上大展拳脚，Meta Programming 的概念已然浮出水面，将编译器变成一个 Service，你可以自由控制在编译器和运行期的逻辑，那是多么美好而令人向往的未来呀！所以，我们坚信 4.0 之后的天地会随着语言的变迁更加开阔。

概括 Anders 大师在 C# 设计过程中的思想，C# 是语言美学的集大成者。例如，当使用 `foreach` 进行循环遍历之后，当应用 `using` 语句代替 `try/finally` 实现强制资源管理，当应用 `attribute` 进行运行时反射，当以 LINQ 进行语言级别的信息查询，这些语言级别的支持为 C# 实现面向对象编程带来强大的功能动力和美学感受。

.NET 十年，一切还在当下与未来。

14.2 .NET 4.0, 第一眼

本节将介绍以下内容：

- .NET 4.0 新特性简介
- .NET 4.0 语言特性概览

14.2.1 引言

.NET 4.0 来了，作为拉开序幕的第一页，本文以提纲挈领的方式展开对.NET 4.0 的初次全景式体验。从 What's new 的角度，开始对.NET 4.0 新特性的探索之旅。

总结起来，.NET 4.0 的新特性主要体现在.NET 平台组成部分（如图 14-2 所示）的各个方面，例如在运行库中引入 DLR、优化垃圾回收算法；类库中的 Tuple、Lazy<T>、BigInteger 支持；语言上的协变与逆变、并行计算支持，以及 Web 开发、云计算等诸多方面：

- CLR (Common Language Runtime)
- DLR (Dynamic Language Runtime)
- C# and VB.NET
- Web and Networking
- Basic Class Libraries (BCL)
- Data
- Client
- Web
- Communications

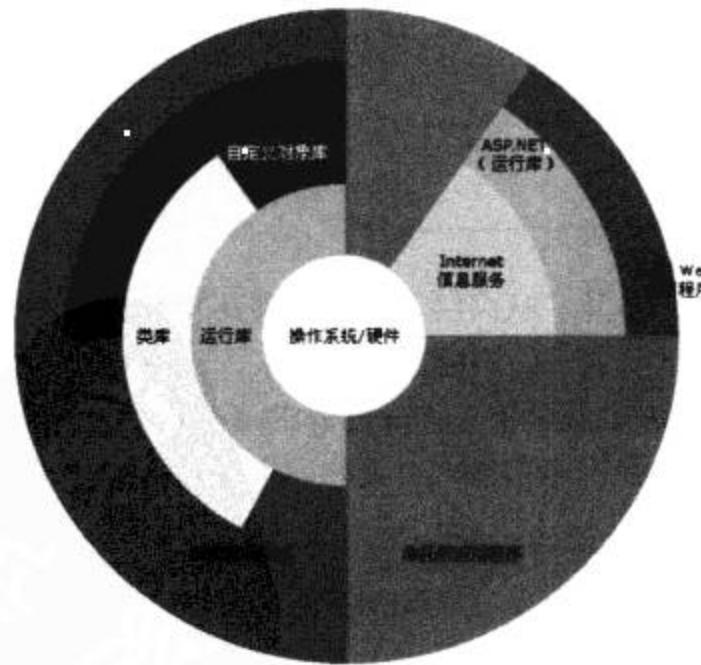


图 14-2 .NET 平台组成 (图片来源: MSDN)

.NET 平台如此庞大，以至于无法在有限的篇幅中表达无限的内容，即便是构建于原有平台上的新特性也是如此。因此，我们只能有选择性地对其中重要的变革加以引导，算是展开对.NET 4.0 的第一眼感受，领略其新感觉、新体验。

14.2.2 第一眼

在第一眼感受中，我们首先瞄上了以下几个方面：

- CLR & DLR
- C#
- Web
- Cloud
- Data
- F#

下面进行一一分解。

1. CLR 4.0

是的，不用怀疑，CLR 4.0 来了，从 CLR 2.0 一跃升级到 CLR 4.0，我们对新建项目进行反编译，即可从 Manifest 中获取当前 CLR 的版本信息 v4.0.30319（如图 14-3 所示）。

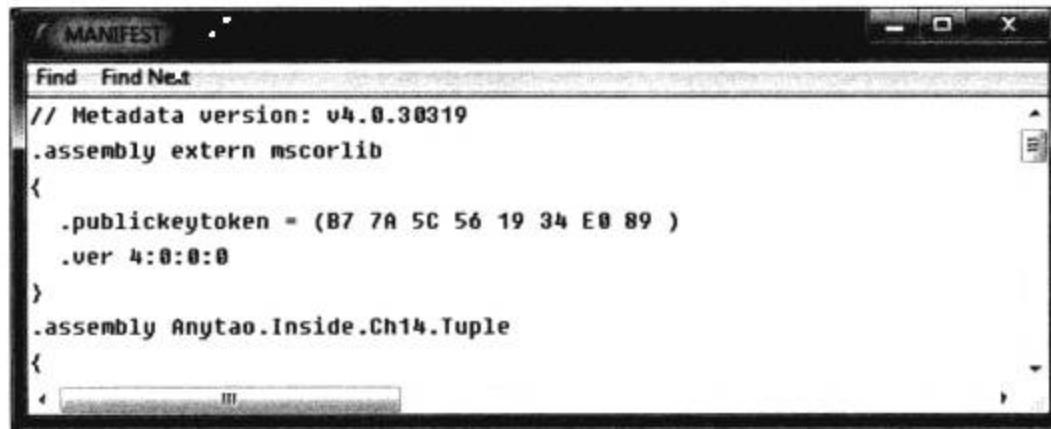


图 14-3 .NET 程序集清单

通常，没有重量级的新成员，.NET 产品组是不敢贸然为新生儿挂上如此响亮的封号：4.0。那么这些分量足够的新家伙到底是谁呢？

- DLR (Dynamic Language Runtime)，动态语言运行时，提供了对动态语言在 CLR 级别的支持。另外，对于 Expression Tree，提供了几个新的类型支持，例如 LoopExpression 和 TryExpression。
- Parallel Computing，并行计算，在多核时代，并行计算已经不可避免。新的平台下，对于并行的支持也随之而来，不需要再直接对线程进行管理。Parallel 和 Task 类，当然还有 PLINQ (Parallel LINQ) 都将粉墨登场。
- GC，GC 也有新变化，主要体现在对于垃圾回收性能上的改善与优化。
- Covariance and Contravariance，逆变与协变，解决了泛型继承的一些问题，算是一场迟来的完善。
- Interoperability，互操作将不依赖于原有的 RIAs，新的 CLR 确保了类型安全操作。

- Lazy Initialization，在未来的日子，通过 System.Lazy<T>为你的实例提供延迟初始化成为可能，这将意味着你的类型可以在实际需要的时候才进行实例化操作、分配内存空间，对性能的控制达到了新的高度。
- In-Process Side-by-Side Execution，In-Process Side-by-Side hosting 解决了不同版本应用在 CLR 4.0 平台上运行的问题。

除此之外，还包括：Security、ETW Events、Code Contracts、Profiling。

这些概念，将在本章逐步一一品味。

2. C# 4.0

C# 4.0 主要引入了以下程序元素：

- Office Programmability
- Dynamic
- Covariance and Contravariance
- Type Equivalence

这些语言特性，也是本章后续阐述的重点。

3. ASP.NET 4.0

对于 Web 开发而言，ASP.NET 从 3.5 开始就已经提供了很多未集成的新东西，其中包括了类似于 MVC 在内的很多了不起的改变。所以，对于.NET 4.0 而言，在 Web 开发方面的贡献，在于对过去的整理，值得关注的内容主要包括：

- ASP.NET Core Service，在输出缓存、SessionState 存储等多个方面进一步完善。
- MVC，ASP.NET MVC 为.NET 平台下的 Web 开发注入前所未有的活力，目前的版本已经更新到 MVC 3.0，值得.NET 平台下致力于 Web 应用的所有人关注。
- Dynamic Data，早已领略了如何一分钟之内开发一个增删改查式的动态站点，在 ASP.NET 4.0 中实现数据驱动的 Web 开发更加 Powerful 了。
- Web Form，在 ASP.NET 4.0 传统的 Web Form 开发被注入了更多的新特性，我们可以像 MVC Application 那样无缝地使用 ASP.NET Routing，对于数据源提供了 Filtering 支持以及更多的 View state 控制。

还有很多，但并非本书关注的核心，留待读者自行探索。

4. Data

在数据方面主要还是 ADO.NET Entity Framework 的增强、改了名的 WCF Data Service 和 Dynamic Data 的更多支持。EF 主要体现在以下方面：

- Persistence-Ignorant Objects，这个特性是令人欢欣鼓舞的，在新的 EF 框架下，EF 实体类和非 EF 实体类都应用 EF 提供的数据支持，这将意味着原有的数据模型（通常称为 POCO Entity）也可以畅享 EF 了。
- Lazy Loading of Related Objects，如果你没有体会过原来 EF 处理延迟加载使用的方式：

```

if (!user.RoleReference.IsLoaded)
{
    user.RoleReference.Load();
}

```

是很难理解的，这一新特性为代码优化提供了机会。

- Functions in LINQ to Entities Queries，一切皆 LINQ 的时代，这点也不奇怪。
- Customized Object Layer Code Generation，为 EF 数据设计器提供了自动生成代码的可配置向导，更加人性化的选择。

除此之外，还包括：Complex Type Support、Naming Service、Improved Model Brower Funcationality。

5. BCL

基础类库的更新，是每次.NET 版本更新的重要内容，.NET 4.0 中基础类库的新增内容，同样惊为天人。在此，仅列举几个重要的方面，更多的内容请参考 MSDN。

- Collections
- BigInteger
- SortedSet<T>
- Tuples
- I/O
- File System Enumeration
- Memory-Mapped Files
- Isolated Storage
- Compression
- Exception
- Reflection
- 64-bit
- Application Domain Resource Monitoring
- Threading
- Unified Model for Cancellation
- Thread-Safe Collection
- Synchronization Primitives

6. Cloud

云计算已经不可避免地到来。在.NET 4.0 平台下，Azure Platform 应用将眼花缭乱，但是可以肯定的是开发模式将更加简化，与.NET 平台的融合更加统一，云应用的价值也更加突显。

7. F#

F#语言是.NET 平台上又一个激动人心的语言，以函数式语言的标签出现在.NET 平台阵营中，将脚本、安全、性能、类型统一在.NET 运行库中。F#具有天生的并行优势，在可预见的未来，F#将更多地参与 C#或者 VB.NET 的融合之中，发挥强有力的魅力。

14.2.3 结论

.NET 4.0来了，在那个夏天，在蚊子与啤酒交错的岁月里。我们一如既往地将角度把握在语言和平台本质的探索上，在可能的角度、最佳实践式的应用和深入本质的探求之路上前行，拍了蚊子、喝着啤酒，好戏才刚刚开始，继续在路上。

14.3 动态变革：dynamic

本节将介绍以下内容：

- dynamic 应用
- DLR 简析
- 面向动态编程

14.3.1 引言

动态类型一直是.NET这种静态类型语言的硬伤，而在.NET 4.0中通过引入 dynamic 关键字为这种硬伤绑上了绷带。例如，在.NET 4.0中可以非常轻松容易地与 COM 对象进行交互，以操作 Office 组件为例，下面的代码看起来简洁而又可读：

```
public static class DynamicWord
{
    public static void Create(string file, string author)
    {
        // Create word app and bind to a dynamic variable
        dynamic app = new Application { Visible = true };

        // Create document
        var doc = app.Documents.Add();
        string text = "Hello, this is anytao. /" + DateTime.Now.ToString();

        dynamic range = doc.Range(0, 0);
        range.Text = text;

        // Save document
        doc.SaveAs(file);
        app.Quit();
    }
}
```

注

上述代码需要引用 Microsoft Word 14.0 Object Library。

通过 dynamic 实现了与 COM 更自然的交互，让.NET 4.0之前操作 Office 组件的开发人员大松一口气。实际上，通过 dynamic 可以将传统的解决方案与.NET 建立起更简单的联系，从而大大减少这种转化的成本。当然，dynamic 的能力还体现在很多方面，我们将在本文进行深入的讨论与应用。

14.3.2 初探

dynamic 是什么？dynamic 首先是一个类型，就像 int、string 或者 object 一样，可以作用于字段、属性、索引器、参数、返回值、局部变量或者类型约束。因此，我们有必要首先定义一个全副武装的 dynamic 类：

```
public class DynamicWork
{
    // A dynamic property
    dynamic Name { get; set; }

    // A dynamic field
    private dynamic value = 100;

    // Dynamic return value, parameter
    public dynamic Work(dynamic x, dynamic y)
    {
        // A dynamic local variable
        dynamic result = x + y;

        return result;
    }
}
```

就像一个 object 可以代表任何类型，dynamic 使得类型决断在运行时进行，方法调用、属性访问、委托调用都可动态分派。例如在运行时调用 Work 方法如下：

```
DynamicWork dw = new DynamicWork();

Console.WriteLine(dw.Work(100, 1)); // Result:101
Console.WriteLine(dw.Work("100", "1")); // Result:1001
```

运行时根据类型信息，来决断不同类型对于“+”操纵符的不同执行逻辑，也就自然而然输出了不同的结果：101 或者 1001。因此，dynamic 为静态类型语言赋予了后期绑定的动态类型语言能力。

同时，动态特性还体现在构建一个动态对象，在 C# 4.0 实现 IDynamicObject 接口的类型，可以完全定义动态操作的意义，通过将 C# 编译器作为运行时组件来完成由静态编译器延迟的操作，例如：

```
dynamic d = new Foo();
string s;

d.MyMethod(s, 3, null);
```

在具体执行过程中，C# 的运行时绑定器基于运行时信息，通过反射获取 d 的实际类型 Foo，然后在 Foo 类型上就利用 MyMethod 方法进行方法查找和重载解析，并执行调用，这正是动态调用的背后秘密：DLR。

14.3.3 本质：DLR

本质上而言，dynamic 是将静态类型动态化，而实现动态化的核心就是在 CLR 基础上扩展而来的 DLR (Dynamic Language Runtime，动态语言运行时)，通过 DLR 为 C# 等静态语言赋予了动态分派的能力。在.NET

4.0 中将引入重要的底层组件 DLR，除了实现动态查找的基础支持，DLR 也同时作为基础设施为类似于 IronRuby、IronPython 这样的动态语言提供统一的互操作机制。用 Anders 大师的话说，CLR 为静态语言提供了统一的框架和编程模型；而 DLR 则为动态语言提供了统一的框架和编程模型。

将本文开始的 DynamicWork 示例在 Reflector 工具中反编译之后，将看到更真实的 dynamic 如下：

```
public class DynamicWork
{
    // Fields
    [Dynamic, CompilerGenerated]
    private object <Name>k__BackingField;
    [Dynamic]
    private object value = 100;

    // Methods
    [return: Dynamic]
    public object Work([Dynamic] object x, [Dynamic] object y)
    { // .....省略.....}

    // Properties
    [Dynamic]
    private object Name
    {
        [return: Dynamic]
        [CompilerGenerated]
        get
        {
            return this.<Name>k__BackingField;
        }
        [param: Dynamic]
        [CompilerGenerated]
        set
        {
            this.<Name>k__BackingField = value;
        }
    }
}
```

对上述代码进行分析，我们发现编译器自动创建了形如<Work>o_SiteContainer0 这样的静态类：

```
[CompilerGenerated]
private static class <Work>o__SiteContainer0
{
    // Fields
    public static CallSite<Func<CallSite, object, object, object>> <>p__Site1;
}
```

其中的 CallSite<T> 表示动态调用点，是 DLR 用于封装动态调用过程的核心组件，CallSite<T> 由 CallSite<T>.Create() 方法创建，并通过指定 CallSiteBinder 来提供规则，维护 3 层缓存策略。

执行 dw.Work(100, 1) 这样的逻辑，继续剥茧可见：

```
[return: Dynamic]
public object Work([Dynamic] object x, [Dynamic] object y)
{
```

```

if (<Work>o__SiteContainer0.<>p__Site1 == null)
{
    <Work>o__SiteContainer0.<>p__Site1 = CallSite<Func<CallSite, object, object,
object>>.Create(Binder.BinaryOperation(CSharpBinderFlags.None, ExpressionType.Add, typeof(D
ynamicWork), new CSharpArgumentInfo[] { CSharpArgumentInfo.Create(CSharpArgumentInfoFlags.No
ne, null), CSharpArgumentInfo.Create(CSharpArgumentInfoFlags.None, null) }));
}
return <Work>o__SiteContainer0.<>p__Site1.Target(<Work>o__SiteContainer0.<>p__Si
tel, x, y);
}

```

其中 CallSiteBinder 是语言相关的，也就是说不同的语言例如 C# 或者 IronRuby 对应着不同的解析规则，由不同的 CallSiteBinder 封装，在运行时为调用点绑定动态操作，其主要作用是处理缓存和语言交互。通过 CallSite<T> 的 Create 方法创建了一个 CallSite<Func<CallSite, object, object, object>> 委托，动态代码的最终执行就体现了对该委托的调用。只有在第一次调用前创建 CallSite，之后的调用才能统一通过 CallSite<T>.Target 方法动态执行。

以在 C# 中调用 IronRuby 为例，动态代码的执行过程大致可以被简化为：IronRuby 语言被解析生成 DLR Tree，DLR 内部再通过调用 DLR Tree 的 Compile 方法将其编译为 IL 代码，本质上是一些可被执行的委托（例如上例中的 Func<CallSite, object, object, object>、Func<CallSite, object, bool>），已经有了 IL 代码，剩下的过程就是我们熟悉的执行过程了，详情参考 4.4 节“管窥元数据和 IL”。

在微软提供的 DLR 架构中，以 CLR 为基础，同时包括了三个重要的组件，如图 14-4 所示。

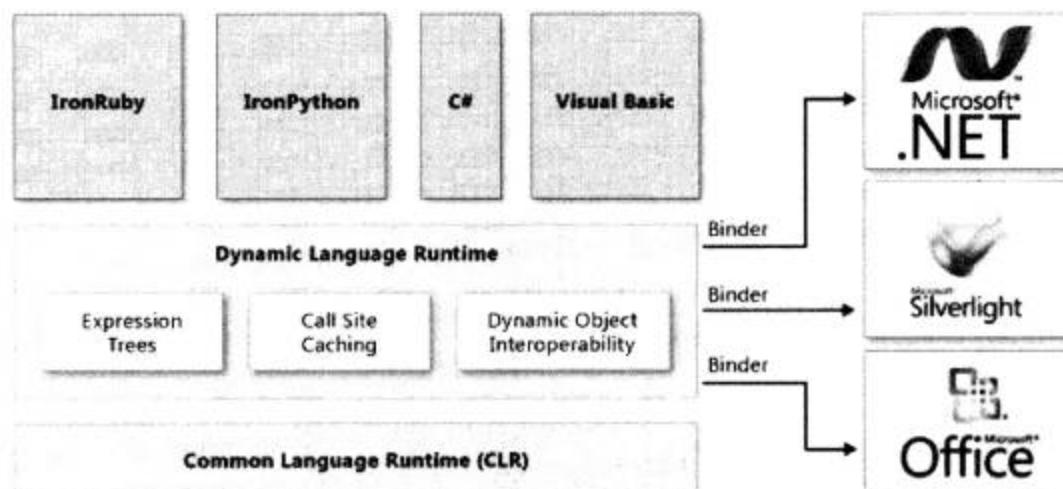


图 14-4 DLR 架构（图片来源：MSDN）

- Expression Trees, DLR Tree 通过扩展 LINQ Expression Tree 而来，DLR 通过 DLR Tree 描述动态表达式。作为高度抽象的语法树，DLR Tree 几乎抽象了所有语言的基础结构，从而为上层开发者提供了可以完全面向 DLR Tree 进行编程的可能，而 DLR 借助表达式树完成动态代码的生成。
- Call Site Caching，Call Site Caching 提供了动态绑定的性能优化，由动态绑定的执行过程可知，将动态表达式解析为 DLR Tree，然后再通过 DLR 生成 IL 代码的过程是一次昂贵的旅程，在性能上需要提供可供改善的方式，这就是 Call Site Caching。通过 Call Site Caching 避免每次动态代码在调用时需要重新编译，而直接从 Caching 中获取。
- Dynamic Object Interoperability，DLR Provider 提供了一系列的接口帮助动态语言开发者实现不同语言的 DLR Provider，其中最重要的包括 IDynamicMetaObjectProvider、DynamicMetaObject、DynamicObject 以及 ExpandoObject。

可以借助继承于 DynamicObject，非常容易地实现自定义动态类型：

```
public class IronObject : DynamicObject
{
    private IDictionary<string, object> members = new Dictionary<string, object>();

    public override bool TryGetMember(GetMemberBinder binder, out object result)
    {
        result = members[binder.Name];
        return true;
    }

    public override bool TrySetMember(SetMemberBinder binder, object value)
    {
        members[binder.Name] = value;
        return true;
    }

    public override IEnumerable<string> GetDynamicMemberNames()
    {
        return members.Keys;
    }
}
```

可以利用 IronObject 实现如下的逻辑：

```
public static void Main()
{
    dynamic iron = new IronObject();
    iron.X = 123;
    iron.Say = new Action<string>(source => Console.WriteLine(string.Format
("IronObject is saying {0}", source)));

    iron.Say("Hello");
}
```

除了可以通过 IDynamicMetaObjectProvider 实现自定义的动态扩展，还可以直接应用 ExpandoObject 来扩展动态成员，例如：

```
public static void Main()
{
    dynamic expo = new ExpandoObject();
    expo.Say = new Func<string, string>(source => string.Format ("Hello, this is {0}", source));

    Console.WriteLine(expo.Say(".NET"));
}
```

其实，DynamicObject 和 ExpandoObject 都实现了 IDynamicMetaObjectProvider 的包装，只是侧重的方向有所不同而已，DynamicObject 实现更多的封装，可以很容易地实现自定义的动态行为；ExpandoObject 则适合为实例成员动态地添加动态行为、属性。从上面的示例来看，我们的 IronObject 其实就是一个简化版的 ExpandoObject。

14.3.4 PK 解惑

1. dynamic VS var

熟悉.NET 3.0 的读者，肯定对 var 关键字带来的匿名类型依然记忆犹新，本书在 13.2 节“赏析 C# 3.0”

中专门解释了 var 关键字带来的语法糖游戏，那么对于：

```
var abc = "Hello, World.";
dynamic abc = "Hello, World.;"
```

之间又有着什么样的关系和秘密？对于开发者而言，在什么情况下考虑应用 var，而又在什么情况下考虑应用 dynamic 呢？

var 和 dynamic，其本质的区别在于类型决断的时机和背后的机理是完全不同的：

- var 属于语法糖游戏，仍然是静态类型在编译时根据表达式返回值进行类型决断，因此需要编译时检查，同时智能感知可用，详细的情况请参考 13.2 节“赏析 C# 3.0”。以上述示例而言，abc 不管在编译时还是运行时都是 string 类型；而 dynamic 源于动态查找，在编译时不作任何类型决断，完全在运行时进行，abc 在编译时是 dynamic 类型，而在运行时才被决断为 string 类型。
- var 只能应用于局部变量；而 dynamic 可以作用于字段、属性、局部变量和返回值。
- var 本质上并不作为类型而存在，因此不能应用 is 或者 as 表达式；而 dynamic 就像 int 或者 string 一样，可以应用 is 或者 as 表达式。

2. dynamic VS object

很多情况下，初见 dynamic 会觉得和 system.object 有很多表现上的相似性。然而，system.object 毕竟是静态类型；而 dynamic 要解决的是对动态对象的绑定。例如一个反射对象、一个 DOM 实例、一个 COM 对象或者一个来自 IronRuby 语言或其他语言的对象。

然而本质上，无论如何，一个 dynamic 对象同时也是一个 object 实例。例如：

```
public static void Main()
{
    dynamic x = 1;

    Console.WriteLine(x);
}
```

翻译为 IL 则被解析为：

```
public static void Main()
{
    object x = 1;
    if (site1 == null)
    {
        site1 = CallSite<Action<CallSite, Type, object>>.Create(Binder.InvokeMember(CSharpBinderFlags.ResultDiscarded, "WriteLine", null, typeof(Program), new CSharpArgumentInfo[] { CSharpArgumentInfo.Create(CSharpArgumentInfoFlags.IsStaticType | CSharpArgumentInfoFlags.UseCompileTimeType, null), CSharpArgumentInfo.Create(CSharpArgumentInfoFlags.None, null) }));
    }
    site1.Target(site1, typeof(Console), x);
}
```

在.NET 的类型系统中，System.Object 就是万物之父，因此在很多场合下 object 类就成为了类型的“适配器”，为那些依据条件返回不同类型的场合“硬着头皮”充当介质。然而这种勉强的传递，毕竟透着不合时宜，而且很多情况还过不了静态编译下错误检查这一关。所以，应用 dynamic 在很多情况下是不错的选择，我们把决定权延迟到运行时决断，让很多静态类型系统下的不合时宜，变得自然而简单。

14.3.5 应用：动态编程

构建于 dynamic 之上的动态编程，为.NET 平台语言消费动态类型对象提供了支持，为 C#这样的静态语言在以下领域产生巨大的变革：

- Office 编程与其他 COM 交互。
- 动态语言支持，在 C# 中消费 IronRuby 动态语言类型将并非难事，体验动态语言特性指日可待。
- 增强反射支持。
- 通过 DLR 可以实现 REPL (READ-EVAL-PRINT-LOOP) 开发模式，关于详细的 REPL 模式，请参考其他动态语言中的详细介绍。

以调用 IronRuby 为例，我们只需引入 IronPython.dll、IronPython.Modules.dll 以及 Microsoft.Scripting.dll，即可通过创建 ScriptRuntime 在 C# 中寄宿 IronPython 环境，进而来操作动态语言的类型信息。

```
ScriptRuntime py = Python.CreateRuntime();
dynamic mypy = py.UseFile("myfile.py");

Console.WriteLine(mypy.MyMethod("Hello"));
```

除了拉近与动态语言之间的距离，无缝地消费动态类型实例，dynamic 同样为语言本身带来精彩而优雅的实现，以 ASP.NET MVC3 中的 ViewBag 为例：

```
public ActionResult Index()
{
    ViewBag.Profile = new Profile
    {
        Name = "Anytao",
        Book = "你必须知道的.NET",
        PublishOn = DateTime.Now,
        Site = new Uri("http://book.anytao.com")
    };

    return View();
}
```

因为 ViewBag 本身被定义为 dynamic 类型，因此可以很容易地为其扩展例如 Name、Book、Date 和 Site 这样的属性，同时在 View 端可以很容易地访问：

```
<p>
<span><b>ViewBag</b></span>

<span>@ViewBag.Profile.Book</span>
<span>@ViewBag.Profile.Name</span><br />
<span>@ViewBag.Profile.PublishOn</span><br />
<span>@ViewBag.Profile.Site</span><br />
</p>
```

ViewBag 的出现是用来改善 MVC2 中的 ViewData。ViewData 本质上是一个强类型 IDictionary，所以在 View 层面进行数据绑定需要将其进行显式的类型转换才能实现编译时的属性访问，某种程度上增加了代码实现的复杂度和冗余。

在“动态”大行其道的今天，无法回避也必须拥抱这个未来的主角。按照惯例，对 dynamic 有如下的小结：

- dynamic 是一个类型，可以作用于字段、属性、索引器、参数、返回值、局部变量或者类型约束。

- dynamic 定义的实例是在运行时决断的，所以不能通过 `typeof` 来作用于 `dynamic`，也不能通过 IntelliSense 感知元数据信息。
- 作为类型，`dynamic` 不能作为任何类型的基类。
- 动态编程实现了与 COM 对象更自然的交互。

14.3.6 结论

动态已经势不可挡，而.NET 4.0 掀开了盖子，给像 C# 这样的静态语言注入了强有力的动力特性，也让构建于.NET 平台上的各色语言 C#、VB.NET、F#、IronPython 还有 IronRuby 实现了更广泛的融合，把我们带入了另一个全新的动态编程时代。

14.4 趋势必行，并行计算

本节将介绍以下内容：

- .NET 的并行计算
- TPL 实践
- PLINQ 实践

14.4.1 引言

大师有云：数未来方向，还看并行。对于微处理器的性能理论，容易让人想起史上最著名的摩尔定律：微处理器的性能每 18 个月翻一番。而这一理论是建立在对微处理器晶体管数量每 18 个月翻一番这一理论的基础之上，当这种增加遇到瓶颈，摩尔定律的预言也就到了该终结的时候。然而，对于性能的提升仍然是信息处理面临的课题，在串行化处理模式走到瓶颈的今天，平行处理成了当下突破性能瓶颈的唯一选择。

当多核成为必然，面向并行编程也将成为程序开发者的必然。因为未来，你可能在任务管理器中看到如图 14-5 所示的超多核系统。

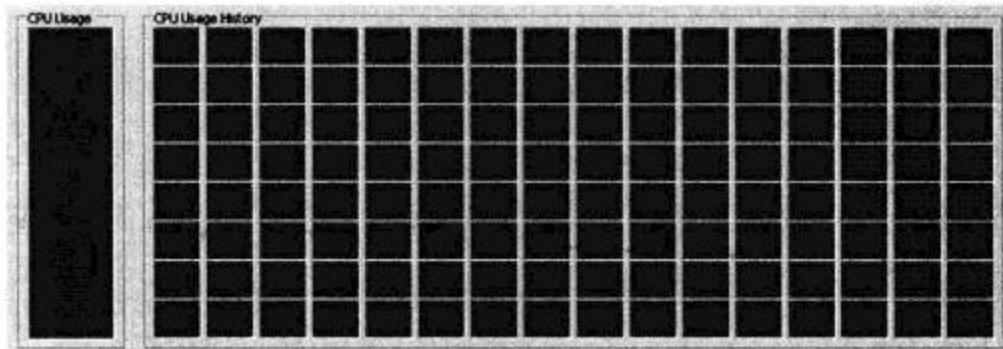


图 14-5 多核系统的任务管理器

从单核、双核、4 核或者 8 核，到未来的 N 核，可能并不会太久。因此，.NET 4.0 将 TPL（Task Parallel Library）和 PLINQ（Parallel LINQ）集成于.NET Framework 中，为 C# 这样的语言赋予了并行化基础，在统一的工作调度程序下进行硬件的并行协调，大大提高应用程序的性能，同时降低现存并发模型的复杂性。

14.4.2 拥抱并行

随着多核时代的到来，每个程序开发者都将面临面向并行编程的选择，并且这种选择是历史的必然，就如同面向对象编程的历史必然一样。并行，其实是人类面向任务最自然的处理方式，只是在 0/1 世界才刚刚开始而已。举例来说，哄孩子的妈妈都会利用闲暇时间来织毛衣，挤公车的白领都带着 iPad 娱乐，而弹钢琴的周杰伦会悠然地哼着“烟花易冷”，这是人类自然处理问题的方式。因此，面对并行，我们并不陌生，而应热情拥抱。

对开发者而言，面对并行，我们将面临选择，你是选，是选还是选呢？

在笔者看来，最重要的转变是并行思维的建立。在传统开发中，我们习惯于在单 CPU 上串行化地执行任务序列，或者通过分时方式在单 CPU 上执行多个任务；而并行编程则强调将一个任务执行在多个 CPU 上，所以在习惯了串行化多任务操作背景下，需要将一个任务分解为可独立运行的多个小任务，并选择在多个 CPU 上执行。

那么，首先来领略一下.NET 4.0 所带来的并行编程吧：

```
static void Main(string[] args)
{
    List<int> list = new List<int>();
    for (int i = 0; i < 100; i++)
    {
        list.Add(i);
    }

    DateTime dt = DateTime.Now;
    Parallel.ForEach(list, x =>
    {
        System.Threading.Thread.Sleep(100);
        Console.WriteLine(x);
    });

    double time = (DateTime.Now - dt).TotalMilliseconds;

    Console.WriteLine(time);
    Console.Read();
}
```

上述代码在 4 核机器上执行时间为 1253 毫秒，而同样的程序采用非并行方式执行，大约需要 10093 毫秒，基本是并行执行的 10 倍左右，并且随着 List 元素的增加，这种性能的差距将变得非常可观。

然而，并行并不意味着线性的性能提升。在最理想的情况下，如果运行于单核机器的时间花费是 2 秒，那么相同的程序在双核上奔跑的时间应该是 1 秒。不过，理想是理想，现实是现实。当硬件实现多核，你的应用并不能自动地以并行方式运行，对程序开发者的挑战，就在于学会并行思维与并行实践。

在.NET 平台之上，F#语言是天生并行的。这源于函数式编程语言在运行时并不修改状态，所以无论运行于多少个线程之上，其状态在不同的线程都是唯一的，从而非常容易并行处理，而且不用担心命令式编程的状态同步问题。而在.NET 平台的其他语言上，例如 C#或者 VB .NET，该如何选择和实践并行呢？

.NET 4.0 的并行扩展构建于.NET 并发运行时支持，仍然基于我们熟悉的线程池而实现，并行扩展解脱了对于线程、CPU 的关注，开发者不必关心死锁和静态条件的影响，无所谓锁、信号量或者其他同步机制，而将关注着眼于业务。

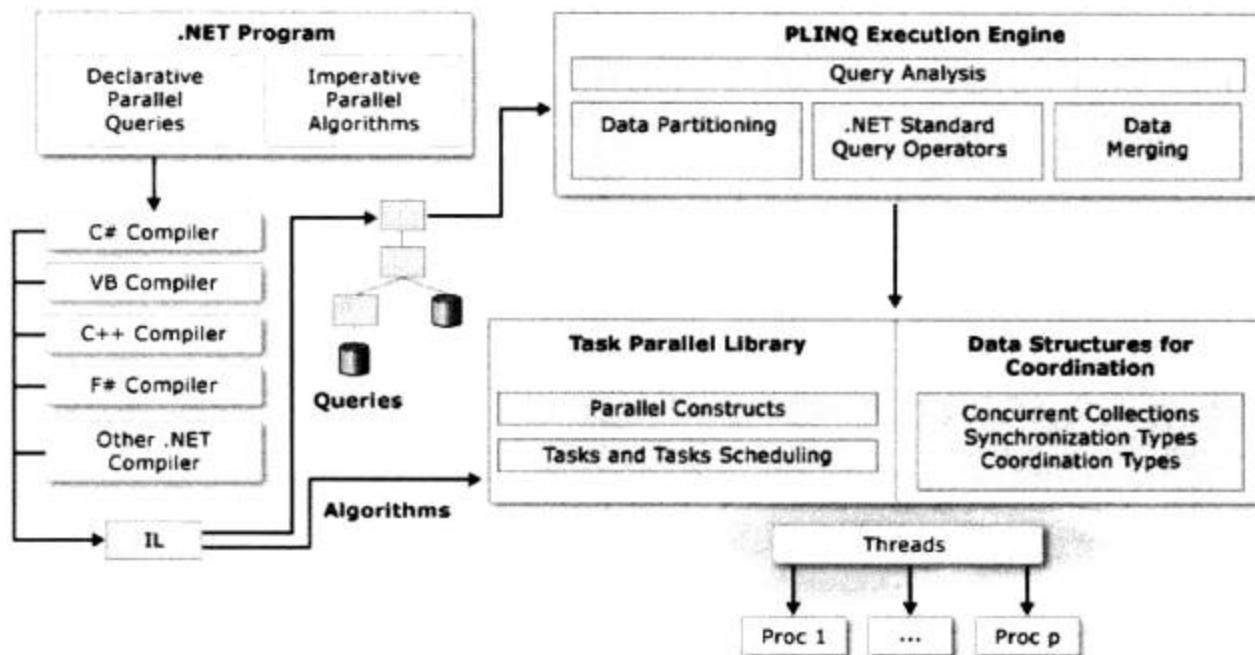


图 14-6 .NET 并行扩展（图片来源：MSDN）

从.NET 4.0 的架构（图 14-6 所示）可见，并行编程模型中主要包括以下几个重要的部分：

- TPL，提供了两个重要的类 Parallel 和 Task，其中 Parallel 主要包含了 For、Foreach 和 Invoke 重载方法；而 Task 则实现了对任务执行的异步支持和控制。
- PLINQ，LINQ to Object 的并行实现。
- Data Structures，.NET 4.0 包含了一系列支持并行编程的新类型，主要包含并发集合类、同步类型和延迟加载。

14.4.3 TPL

TPL（Task Parallel Library）是.NET 4.0 提供的并行扩展的重要部分，为并行模型提供了基础支持。TPL 面向 Task 编程，一个 Task 就是一个可执行的任务单元，TPL 负责线程的创建和 Task 执行，从而实现了开发者对于并行处理的自动化。

1. System.Threading.Tasks.Task

以遍历一个二叉树为例，我们演示通过 Task 实现并行遍历，TaskFactory 的 StartNew 方法用于创建一个 Task，而具体的 Task 逻辑通过 Action 传入。当具体 task 开始执行时，task scheduler 负责将 task 分配给具体的线程执行。

另外，可以通过 Wait/WaitAny/WaitAll 等方式等待 task 执行完成，而 task 执行中出现的异常，CLR 将异常信息包装在 AggregateException 中：

```

public static void ProcessTree<T>(Tree<T> tree, Action<T> action)
{
    if (tree == null)
    {

```

```

        return;
    }

    var left = Task.Factory.StartNew(() => ProcessTree(tree.Left, action));
    var right = Task.Factory.StartNew(() => ProcessTree(tree.Right, action));

    action(tree.Data);

    try
    {
        Task.WaitAll(left, right);
    }
    catch (AggregateException ae)
    {
        ae.Handle((x) =>
        {
            if (x is ATException)
            {
                Console.WriteLine("App exception, please contact administrator.");
                return true;
            }
        });

        return false;
    });
}
}

```

关于 Task 相关的话题,有很多的细节可以深入,例如 Task Scheduler、Child Task 还有 Task Cancellation 等。有兴趣的读者,可以继续挖掘。

2. System.Threading.Tasks.Parallel

Parallel 静态类提供了 3 个重载的静态方法: For、Foreach 还有 Invoke, 其中 For 和 Foreach 通过遍历数据源实现数据并行化, 而 Invoke 是实现隐式任务并行化的基础。

```
public static void Invoke(params Action[] actions);
```

Invoke 以 actions 委托数组为参数, 那么我们以 Invoke 为例来了解任务并行化的执行:

```

public static void DoInvoke()
{
    Parallel.Invoke(
        () => Console.WriteLine("Run task A"),
        () => Console.WriteLine("Run task B"),
        () => Console.WriteLine("Run task C")
    );
}

```

通过 Visual Studio 2010 工具的 Parallel Stacks 分析其运行时的线程情况, 如图 14-7 所示。

同样, 也可以应用 Task 实现显式的任务并行化, 上述逻辑可以以下面的方式实现:

```

public static void DoTasks()
{
    Task[] tasks = new Task[]
    {
        Task.Factory.StartNew(() => Console.WriteLine("Run task A")),
        Task.Factory.StartNew(() => Console.WriteLine("Run task B")),
        Task.Factory.StartNew(() => Console.WriteLine("Run task C"))
    };
}

```

```

    Task.WaitAll(tasks);
    Console.WriteLine("Task Finished.");
}

```

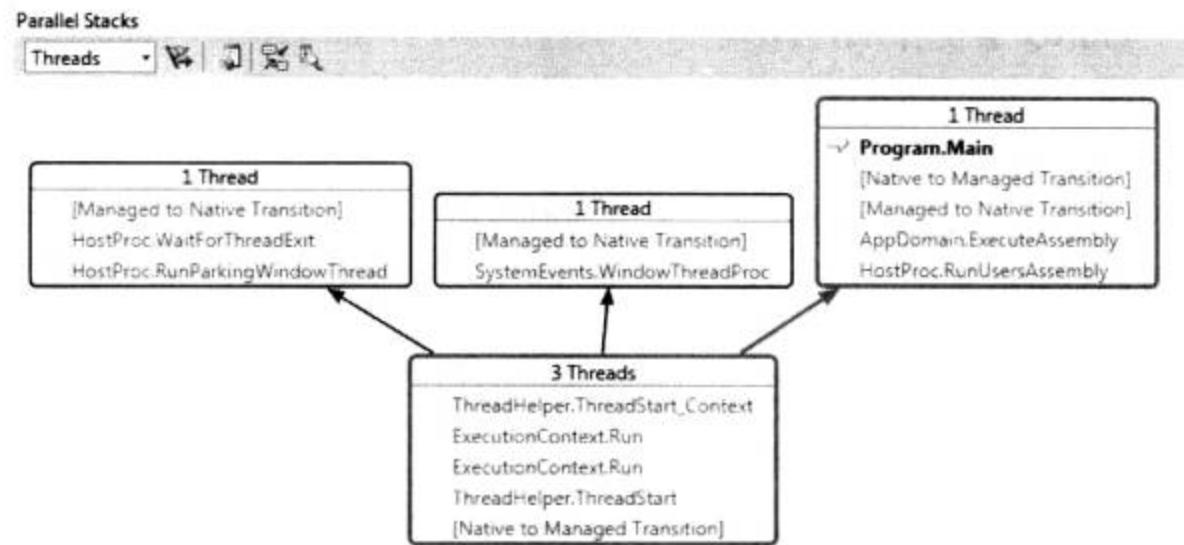


图 14-7 Parallel Stacks 的运行时线程分析

14.4.4 PLINQ

PLINQ (Parallel LINQ) 是声明式的数据并行性框架，提供了以最简单的方式实现并行的 LINQ 查询操作的框架支持，可接受任何的 LINQ to object 和 LINQ to xml 查询，而开发者则完全像进行 LINQ to object 查询一样，将数据查询自动执行在多核。

具体来说，以下两行代码的区别仅仅在于通过 `AsParallel()` 就自动实现了将查询操作运行于多个线程中，对于习惯 LINQ 操作的开发者而言，PLINQ 让一切看起来没有任何改变，而调动起多核运行的复杂逻辑已经被实现了。

```

var ls = list.Select(x => x + 10);
var ls = list.AsParallel().Select(x => x + 10);

```

类似于 `AsParallel()` 的扩展方法被统一封装在 `System.Linq.ParallelEnumerable` 类中，主要公开了 `AsParallel`、`AsSequential`、`AsOrdered`、`ForAll`、`WithCancellation` 以及 `Aggregate` 等扩展方法。剖析这些公开的方法，基本都是对 `IEnumerable<T>` 或 `ParallelQuery<T>` 的扩展，因此 PLINQ 支持所有的 LINQ 运算符，对于任何内存中的集合例如 `IList<T>`、`IEnumerable<T>` 都可以通过 PLINQ 来赋予并行查询。

```

public static ParallelQuery<TSource> AsParallel<TSource>(this IEnumerable<TSource> source);

```

其中 `ParallelQuery<T>` 是 `IEnumerable` 的镜像，因此 `AsParallel` 实例对 `foreach` 和 `IEnumerable<T>` 仍然有效。

```

public class ParallelQuery<TSource> : ParallelQuery, IEnumerable<TSource>, Ienumerable
{
    public virtual IEnumerator<TSource> GetEnumerator();
}

```

以上例而言，`list.AsParallel()` 执行之后，将返回 `ParallelQuery<T>` 实例，而 `Select` 方法实际操作的是 `ParallelQuery<T>` 实例。另外，任何 `ParallelQuery<T>` 实例在继承层次上仍然是 `IEnumerable<T>` 实例，所以完全可以无缝地与 LINQ 查询兼容。

关于 PLINQ，有如下的小结：

- 默认情况下，PLINQ 最多可开启 64 个处理器，可以通过 WithDegreeOfParallelism 指定。
- 默认情况下，PLINQ 会使用所有的处理器，但是可以通过 WithDegreeOfParallelism()方法来指定可使用的处理器数量，例如下面的代码将限制 PLINQ 最多使用 4 个处理器。

```
var list = Enumerable.Range(1, 10000).AsParallel().WithDegreeOfParallelism(4);
```

- PLINQ 并不支持 LINQ to SQL，LINQ to SQL 的具体查询并不发生在内存，而是发生在数据库。
- 在运行时，PLINQ 基础架构会分析查询结构，并分析在并行或顺序执行的查询成本，然后选择是否并行执行。PLINQ 在高成本并发算法和低成本顺序算法中，以顺序算法为默认选择。当然，可以通过 WithExecutionMode<T>手动选择 ParallelExecutionModel.ForceParallelism，要求 PLINQ 强制执行并行算法。

```
public enum ParallelExecutionMode
{
    Default = 0,
    ForceParallelism = 1,
}
```

- 因为并行执行并不能保证执行的顺序性，因此 list.AsParallel().Select(x => x + 10) 并不按 list 原本的元素顺序输出，如果必须保持查询之前的顺序性，可以应用 AsOrder()方法，但是保持顺序将引入额外的开销，对执行性能造成影响。

14.4.5 并行补遗

关于.NET 的并行编程，有如下需要注意的事项：

- 并行并不意味着绝对的性能提升，并且并行本身也会增加额外的开销。在数据源数据量较小时，应用顺序 LINQ（LINQ to Object）在性能上要优于 PLINQ，因为并行执行本身的开销是可观的，并行并不意味着性能的绝对提升，必须量力而行。
- 确保并行执行的循环逻辑是线程安全的，该逻辑将可能运行于多线程中。
- 尽量避免在 lock 中执行任务等待，因为任务的执行逻辑可能锁定了同一个变量，并导致死锁的发生。

```
public static void Main()
{
    lock (sync)
    {
        Task t = Task.Factory.StartNew(() =>
        {
            lock (sync)
            {
                Console.WriteLine("Dead lock here.");
            }
        });
        t.Wait();
    }

    Console.Read();
}
```

- 并行计算，让摩尔定理重新焕发光彩，信息技术仍然以大刀阔斧的步子在高速发展。
- 对并行投入更多的考量，是未来开发的重要方面。

14.4.6 结论

发展总是计算机产业的永恒主题，在以硬件为核心的发展史上，从386到i7，从单CPU到多核。然而，过去的变革历史，并未引起软件的变革要求，开发者习惯了面对一以贯之的编程模型。而多核时代的发展，打破了这一切习惯，需要面对挑战和改变，从软件架构到开发方法，使得软件产品面向多核和并发。

面对并行，.NET已经做好了准备，那么开发者你呢？

14.5 命名参数和可选参数

本节将介绍以下内容：

- 命名参数
- 可选参数

14.5.1 引言

命名参数和可选参数，算是C#语言一次迟到的完善。事实上，类似的语言特性早在很多年前就存在于其他语言，例如VB.NET、Delphi等。动作虽小，意义尤大，因为这些特性的完善意味着.NET在各个层面上走向完美。

那么，我们先一睹命名参数和可选参数的芳容吧：

```
class Program
{
    static void Main(string[] args)
    {
        Write();
        Write("subs");
        Write(y: 200, x: 100);
        Write(x: 200, y: 123, method: "subs");
    }

    static void Write(string method = "add", int x = 1, int y = 2)
    {
        if (method == "add")
        {
            Console.WriteLine("{0} + {1} = {2}", x, y, x + y);
        }
        else if (method == "subs")
        {
            Console.WriteLine("{0} - {1} = {2}", x, y, x - y);
        }
    }
}
```

如示例所示，命名参数和可选参数简单到可以用只言片语来介绍：

- 命名参数，方法调用时通过例如 Write(y: 200, x: 100)的方式提供参数名称，从而可以忽略参数顺序。
- 可选参数，在方法声明时设定默认值，方法调用时，如果有可选参数，则可忽略该参数，直接应用默认值。不过，可选参数必须放在方法声明的后面，否则将引发编译时错误。

14.5.2 一览究竟

将表象还原于本质，是本书习惯的方式，也是揭开神秘面纱的方式，将开始的示例反编译为 IL，发现如下：

```
internal class Program
{
    // Methods
    private static void Main(string[] args)
    {
        Write("add", 1, 2);
        Write("subs", 1, 2);
        int CS$0$0000 = 200;
        int CS$0$0001 = 100;
        Write("add", CS$0$0001, CS$0$0000);
        CS$0$0000 = 200;
        CS$0$0001 = 0x7b;
        string CS$0$0002 = "subs";
        Write(CS$0$0002, CS$0$0000, CS$0$0001);
    }

    private static void Write([Optional, DefaultValue("add")] string method, [Optional, DefaultValue(1)] int x, [Optional, DefaultValue(2)] int y)
    {
        if (method == "add")
        {
            Console.WriteLine("{0} + {1} = {2}", x, y, x + y);
        }
        else if (method == "subs")
        {
            Console.WriteLine("{0} - {1} = {2}", x, y, x - y);
        }
    }
}
```

从 IL 代码中可见，命名参数的作用完全是告诉编译器使用正确的方法签名顺序，保证调用的参数按照方法签名的正确顺序进行调用。IL 为各种高级语言在语法糖的体验上，提供了无限的可能，命名参数只是小菜一碟。

可选参数方面，编译器为其增加了 `OptionalAttribute` 和 `DefaultValueAttribute` 特性，可选参数的调用秘密亦源于此：

```
[ComVisible(true), AttributeUsage(AttributeTargets.Parameter, Inherited = false)]
public sealed class OptionalAttribute : Attribute
{
    // Methods
    public OptionalAttribute();
    internal static Attribute GetCustomAttribute(RuntimeParameterInfo parameter);
    internal static bool IsDefined(RuntimeParameterInfo parameter);
}

[AttributeUsage(AttributeTargets.Parameter)]
public sealed class DefaultValueAttribute : Attribute
```

```

{
    // Fields
    private object value;

    // Methods
    public DefaultParameterValueAttribute(object value);

    // Properties
    public object Value { get; }
}

```

举一反三，完全可以借助于 `OptionalAttribute` 和 `DefaultParameterValueAttribute` 来实现可选参数：

```

static void Register(string email, [Optional, DefaultValue(27)] int age)
{
    Console.WriteLine(string.Format("{0} is {1} now.", email, age));
}

```

同样，在调用方也可以使用同样的方式调用 `Register` 方法：

```

static void Main(string[] args)
{
    Register("anytao@live.com");
    Register("anytao@live.com", 30);
}

```

14.5.3 简单应用

可选参数和命名参数虽然只是简单的语法糖游戏，不过在某些时候可以给代码带来很完美的解决方案，以 ASP.NET MVC 为例，很多时候需要考虑服务端分页的情况，因此 `page` 和 `count` 这两个参数就会广泛存在于很多 `Action`（也就是方法）中，例如：

```

public ActionResult Posts(string author, int page = 0, int count = 20)
{
    // ...省略...
    return View();
}

```

因此，可以很灵活地通过默认的分页信息 `page` 和 `count` 实现对服务端的数据分页请求：

```

http://anytao.com/blog/posts
http://anytao.com/blog/posts?page=2
http://anytao.com/blog/posts?count=15
http://anytao.com/blog/posts?page=2&count=10

```

以上请求在可选参数的帮助下都是合法的。在不指定 `page` 和 `count` 分页信息时，服务端将自动以可选参数的默认值作为分页信息提交，完善了对于参数的简洁度和兼容性。

可选参数也广泛应用在与 COM 对象的交互中，例如在 14.3 节“动态变革：`dynamic`”中对 Office 对象的操作，很多 Office 函数包括了十多个参数，可选参数极大地简化了这种调用。

14.5.4 结论

不断进步的语言特性，不断完善的语言表现。从.NET 3.0 始，我们看到了这种进步和完善在不断的演化，同时带来不断完美的表现。

14.6 协变与逆变

本节将介绍以下内容：

- 协变与逆变的介绍
- 深入理解泛型可变性

14.6.1 引言

从.NET 3.5 到.NET 4.0，最熟悉的 `IEnumerable<T>` 有了简单的变化，检查其定义发现：

```
public interface IEnumerable<out T> : IEnumerable
{
    Ienumerator<T> GetEnumerator();
}

public interface IEnumerable<T> : IEnumerable
{
    Ienumerator<T> GetEnumerator();
}
```

同样的变化还存在于 `Func<T, TResult>` 这样的泛型委托，.NET 4.0 中对泛型类型参数 `T` 增加了 `out` 和 `in` 关键字，由这种变化引入两个熟悉而又陌生的概念：协变（`Convariant`）与逆变（`Contravariant`）。协变与逆变并非概念的创新，而是为解决泛型类型转换中广泛存在的问题：

```
static void Main(string[] args)
{
    Derived d = new Derived();
    Base b = d;

    IEnumerable<Derived> list = new List<Derived> { d };
    IEnumerable<Base> list2 = list;
}
```

子类 `Derived` 实例 `d` 到父类 `Base` 的转换是 1.2 节“什么是继承”中详细讨论的话题，由继承关系带来的子类引用到父类引用的转换是.NET 天经地义的合法规则。而这种规则对于泛型 `IEnumerable<Base>` 和 `IEnumerable<Derived>` 之间，却有一本历史旧账。事实上，上段代码在.NET 4.0 之前是不被允许的。到了.NET 4.0 时代，通过引入 `in` 和 `out` 关键字，.NET 实现了对于泛型类型的协变与逆变支持。

14.6.2 概念解析

那么，什么是协变，什么是逆变呢？

从本质上来看，协变与逆变属于类型可变性的范畴，是.NET 为了支持更广泛的隐式类型转换而存在的。例如对于泛型接口 `IAnyObject<T>`，如果 `IAnyObject<Derived>` 可以转换为 `IAnyObject<Base>`，那么这个过程被称为协变；相反，如果 `IAnyObject<Base>` 可以转换为 `IAnyObject<Derived>`，那么这个过程被称为逆变。除了泛型接口，泛型委托也支持协变与逆变，例如对泛型委托 `Func<T, TResult>` 而言：

```
public delegate TResult Func<in T, out TResult>(T arg);
```

Func<T, TResult>同时支持对 T 的逆变和对 TResult 的协变。因此，下面来进一步认识一下关于协变与逆变的规则与定义。

1. 协变

首先，自定义支持协变的泛型接口：

```
public interface IFactory<out T>
{
    T Create();
}

public class Factory<T> : IFactory<T>
{
    #region IFactory<T> Members

    public T Create()
    {
        return (T)Activator.CreateInstance<T>();
    }

    #endregion
}
```

支持协变的 IFactory<T>接口，以关键字 out 标识类型参数 T，那么对于应用端，下面的调用过程就实现了对 T 的协变：

```
static void Main(string[] args)
{
    IFactory<Derived> f = new Factory<Derived>();
    IFactory<Base> f2 = f; //协变过程
    Base b = f2.Create();
    b.Write();
}
```

对于协变，必须遵循以下规则：

- 泛型参数以 out 关键字标识，并且只能应用于只读属性、方法或者委托的返回值。
- 目标泛型类型的类型参数（例如 Base）必须是被转换泛型类型的类型参数（例如 Derived）的基类。
- 在.NET 中有代表性的实现了协变的泛型接口有 IEnumerable<T>、 IQueryable<T>、 IEnumerator<T>、 IGrouping<Tkey, TElement>等。

2. 逆变

首先，自定义支持逆变的泛型接口：

```
public interface INotifier<in TNotification> where TNotification : INotification
{
    void Notify(TNotification notification);
}

public class Notifier<TNotification> : INotifier<TNotification> where TNotification : INotification
{
    #region INotifier<TNotification> Members

    public void Notify(TNotification notification)
    {
        Console.WriteLine(notification.Message);
    }
}
```

```

    }
    #endregion
}

```

其中的 INotification 实体及其子类定义如下：

```

public interface INotification
{
    string Message { get; }
}

public abstract class Notification : INotification
{
    public abstract string Message { get; }
}

public class MailNotification : Notification
{
    public override string Message
    {
        get { return "You got a email."; }
    }
}

```

支持逆变的 INotifier<T> 接口，以关键字 in 标识类型参数 TNotification，那么对于应用端，下面的调用过程就实现了对 TNotification 的逆变：

```

static void Main(string[] args)
{
    INotifier<INotification> notifier = new Notifier<INotification>();
    INotifier<MailNotification> mailNotifier = notifier; // 逆变过程
    mailNotifier.Notify(new MailNotification());
}

```

对于逆变，必须遵循以下规则：

- 泛型参数以 in 关键字标识，并且只能应用于只写属性、方法或委托的参数。
- 目标泛型类型的类型参数（例如 MailNotification）必须是被转换泛型类型的类型参数（例如 INotification）的子类。
- 在.NET 中有代表性的实现了逆变的泛型接口有 IComparer<T>、IComparable<T> 等。

14.6.3 深入

如前所述，协变与逆变解决的是隐式类型转换对于泛型接口（或委托）的类型安全，而类型转换的本源体现在继承和多态。从本质上看，通过协变与逆变实现了不具有继承关系的两个对象间的隐式转换，因此从这个意义上协变与逆变延伸了继承和多态。

例如，以协变来延伸继承在泛型的扩展，下面就以一个消息分发组件的实现为例来深入对于协变的理解。对于消息而言，复杂的系统可能会包括多种类型的消息，例如 E-mail、SMS 或者其他消息：

```

public interface IMessage
{
    string Message { get; set; }
}

```

```

public class SMS : IMessage
{
    public string To { get; set; }
    public string Message { get; set; }
}

public class Email : IMessage
{
    public string From { get; set; }
    public string To { get; set; }
    public string Subject { get; set; }
    public string Message { get; set; }
}

```

对于消息分发器而言，按照面向抽象编程的原则，在此以 `IMessage` 作为对于不同消息的抽象，因此对于简单分发场合：

```

public class MessageDispatcher
{
    public void Dispatch(IMessage mail)
    {
        Send(mail);
    }

    private void Send(IMessage mail)
    {
        Console.WriteLine("Send message {0}.", mail.Message);
    }
}

```

对于 `MessageDispatcher` 的消费方：

```

static void Main(string[] args)
{
    // MailDispatcher

    IMessage mail = new Email { From = "anytao@live.com", To = "admin@anytao.net", Subject = "Hello", Message = "" };
    MessageDispatcher dispatcher = new MessageDispatcher();
    dispatcher.Dispatch(mail);
}

```

很显然，`Dispatch` 方法体现了面向对象编程中的继承特性。如果考虑实现群组分发的扩展，那么就可以将不同的消息加入到消息群组中，为通过异步进程来处理消息的分发做好准备，因此可以通过简单的 `IEnumerable<T>` 来模拟消息的群组：

```

public class MessageDispatcher
{
    public void Dispatch(IEnumerable<IMessage> mails)
    {
        foreach (var item in mails)
        {
            Send(item);
        }
    }
}

```

同样，对于 `MessageDispatcher` 的消费方：

```

static void Main(string[] args)
{
    IList<SMS> sms = new List<SMS>

```

```

    {
        new SMS { To = "1111111111", Message = "Hello, Yixia." },
        new SMS { To = "123456789", Message = "Happy birthday." }
    };

    dispatcher.Dispatch(sms);
}

```

很显然，`IEnumerable<T>`通过引入 `out` 延伸了 `IEnumerable<T>` 对继承的支持，同样的情况也体现在泛型委托。关于继承与多态的更多讨论，参考本书第1章“OO大智慧”的相关内容。

最后，关于协变与逆变，有如下的小结：

- 在.NET新特性中，协变与逆变只对委托和接口有效，对于普通的泛型类或者泛型方法无效。
- 协变与逆变的可变类型参数必须是引用类型，不能是值类型。从继承的角度而言，值类型都对应于封闭的结构体，是密封的且不具有继承性，所以类型转换存在不兼容问题。
- 泛型接口或者泛型委托可以同时具有协变和逆变类型参数，例如典型的 `Func<T, TResult>` 类型。

```
public delegate TResult Func<in T, out TResult>(T arg);
```

- 在.NET 2.0/3.0时代，委托已经支持一定程度的协变和逆变，例如下面的定义：

```
public delegate void Write(Derived d);

public static void WriteBase(Base b)
{
    Console.WriteLine(b.Id);
}
```

然后，可以有如下的隐式转换：

```
static void Main(string[] args)
{
    Write w = WriteBase; //逆变过程
    w(new Derived() { Id = 100 });
}
```

- 在.NET 1.0时代，协变性体现在数组类型：

```
object[] os = new string[100];
```

- 一般的类或者结构体并不具备可变性，必须明确可变性的应用规则与场合。

14.6.4 结论

协变与逆变为.NET带来概念与语法上的延伸，也同时带来了应用上的灵活性。从历史的脉络来看，在.NET 1.0时代对数组的支持，在.NET 2.0/3.0时代对委托的协变与逆变，再到.NET 4.0时代的泛型接口，我们惊喜于这种不断完善的进步。

14.7 Lazy<T>点滴

本节将介绍以下内容：

- 延迟加载
- `Lazy<T>`及其应用

14.7.1 引言

对象的创建方式，始终代表了软件工业的生产力方向，代表了先进软件技术发展的方向，也代表了广大程序开发者的集体智慧。以 new 的方式创建，通过工厂方法，利用 IoC 容器，都以不同的方式实现了活生生实例成员的创生。而本文所关注的 Lazy<T>也是干这事儿的。不过，简单说来，Lazy<T>要实现的就是按“需”创建，而不是按时创建。

14.7.2 延迟加载

延迟加载广泛存在于软件开发的各个方面，其核心思想体现在按需创建要使用的资源。一般来说，资源的使用意味着成本的投入和时间的消耗，因此将资源的创建实现“按需分配”（Load-On-Demand）本质上是对成本和时间的节省。例如，Web 开发中对于图片加载的延迟处理，以及 ADO.NET Entity Framework 框架对于关联数据加载的 Load 机制，都体现了按需加载的思想。

同样在.NET 中，对于对象的创建也存在这种“按需”的目标。在.NET 4.0 之前，我们通过某些模式实现类似的延迟创建机制：

```
public class LazySingleton : ISingleton
{
    private LazySingleton()
    {
    }

    public static LazySingleton Instance
    {
        get
        {
            return Lazy.data;
        }
    }

    private class Lazy
    {
        static Lazy()
        {
        }

        internal static readonly LazySingleton data = new LazySingleton();
    }
}
```

在以上模式中，LazySingleton 的实例化由第一次引用嵌入类 Lazy 的静态字段 data 时而触发，也就是说只有在使用 Instance 属性时才进行实例化操作。实际上，.NET 4.0 中新的 Lazy<T>实现了类似于上述 LazySingleton 的模式，不过提供更多控制选项（例如支持委托）和线程安全，下面就来一见分晓吧。

14.7.3 Lazy<T>登场

往往有这样的情景，一个关联对象的创建需要较大的开销，为了避免在每次运行时创建这种家伙，有一

种聪明的办法叫做实现“懒对象”，或者延迟加载。.NET 4.0 之前，实现懒对象的机制，需要开发者自己来实现与管理，例如 LazySingleton 的实现方式。可喜的是，在.NET 4.0 中引入另一个好玩儿的家伙 System.Lazy<T>，其定义如下：

```
[Serializable]
public class Lazy<T>
{
    public Lazy();
    public Lazy(bool isThreadSafe);
    public Lazy(Func<T> valueFactory);
    public Lazy(LazyThreadSafetyMode mode);
    public Lazy(Func<T> valueFactory, bool isThreadSafe);
    public Lazy(Func<T> valueFactory, LazyThreadSafetyMode mode);

    public bool IsValueCreated { get; }
    public T Value { get; }

    public override string ToString();
}
```

假设，我们有一个大块头：

```
public class Big
{
    public int ID { get; set; }

    // Other resources
}
```

那么，可以使用如下的方式来实现 Big 的延迟创建：

```
static void Main(string[] args)
{
    Lazy<Big> lazyBig = new Lazy<Big>();
}
```

从 Lazy<T> 的定义可知，其 Value 属性就是包装在 Lazy Wrapper 中的真实 Big 对象，那么当我们第一次访问 lazyBig.Value 时，就会自动创建 Big 实例。

```
static void Main(string[] args)
{
    Lazy<Big> lazyBig = new Lazy<Big>();

    Console.WriteLine(lazyBig.Value.ID);
}
```

当然，由其定义可知，Lazy 远没有这么小儿科，它同时可以为我们提供以下的服务：

- 通过 IsValueCreated，获取是否“已经”创建了实例对象。
- 解决非默认构造函数问题。

显而易见。Big 类并没有提供带参数的构造函数，那么：

```
public class Big
{
    public Big(int id)
    {
        this.ID = id;
    }
}
```

```

public int ID { get; set; }

// Other resources
}

```

上述创建方式将引发运行时异常 MissingMemberException，提示包装对象没有无参的构造函数。那么，这种情形下的延迟加载，该如何应对呢？其实 Lazy<T>的构造中还包括：

```
public Lazy(Func<T> valueFactory);
```

它正是用来应对这样的挑战：

```

static void Main(string[] args)
{
    // Lazy<Big> lazyBig = new Lazy<Big>();
    Lazy<Big> lazyBig = new Lazy<Big>(() => new Big(100));

    Console.WriteLine(lazyBig.Value.ID);
}

```

其实，从 public Lazy(Func<T> valueFactory)的定义可知，valueFactory 可以返回任意的 T 实例，那么任何复杂的构造函数、对象工厂或者 IoC 容器方式都可以在此以轻松的方式兼容，例如：

```

public class BigFactory
{
    public static Big Build()
    {
        return new Big(100);
    }
}

```

可以应用 Lazy<T>和 BigFactory 实现 Big 的延迟加载：

```

static void Main(string[] args)
{
    Lazy<Big> lazyBig = new Lazy<Big>(() => BigFactory.Build());

    Console.WriteLine(lazyBig.Value.ID);
}

```

- 提供多线程环境支持。

另外的构造器：

```

public Lazy(bool isThreadSafe);
public Lazy(Func<T> valueFactory, bool isThreadSafe);

```

中，isThreadSafe 则应用于多线程环境下，如果 isThreadSafe 为 false，那么延迟加载对象则一次只能创建于一个线程。

14.7.4 Lazy<T>本质

反编译 Lazy<T>，其本质实现了与 LazySingleton 基本类似的思想，去掉不必要的逻辑，实现一个精简的 Lazy<T>：

```

[Serializable]
public class MyLazy<T>
{
    static MyLazy()

```

```
{  
}  
  
public MyLazy()  
{  
}  
  
public MyLazy(Func<T> valueFactory)  
{  
    this.valueFactory = valueFactory;  
}  
  
public T Value  
{  
    get  
    {  
        Boxed boxed = null;  
  
        if (this.boxed != null)  
        {  
            boxed = this.boxed as Boxed;  
            if (boxed != null)  
            {  
                return boxed.value;  
            }  
        }  
  
        return this.Init();  
    }  
}  
  
private T Init()  
{  
    Boxed boxed = null;  
  
    if (this.boxed == null)  
    {  
        boxed = this.CreateValue();  
        this.boxed = boxed;  
    }  
  
    return boxed.value;  
}  
  
private Boxed CreateValue()  
{  
    if (this.valueFactory != null)  
    {  
        return new Boxed(this.valueFactory());  
    }  
    else  
    {  
        return new Boxed((T)Activator.CreateInstance(typeof(T)));  
    }  
}
```

```

    }

    private volatile object boxed;
    private Func<T> valueFactory;

    // A nested class for box the data of T
    private class Boxed
    {
        internal Boxed(T value)
        {
            this.value = value;
        }

        internal T value;
    }
}

```

其实代码已经说明了一切，Lazy<T>本身既简单又赋予小小的设计技巧，MyLazy 的使用完全和 Lazy<T>一样：

```

static void Main()
{
    MyLazy<Big> myLazy = new MyLazy<Big>(() => BigFactory.Build());

    Console.WriteLine(myLazy.Value.ID);

    Console.WriteLine(myLazy.Value.ID);
}

```

延迟加载并不适合所有的场合，事实上，在大部分实践中实例的创建都在声明时随即产生，选择延迟加载，需要考虑必要的情况和场合，同时在性能与消耗间平衡。

14.7.5 结论

关于 Lazy<T>的应用，其实已经不是一个纯粹的语言问题，还涉及了对设计的考量，例如实现整个对象的延迟加载，或者实现延迟属性、考量线程安全等。既然是点滴，就不说教太多。因为，.NET 4.0 提供的关注度实在不少，已经让人眼花缭乱了。

14.8 Tuple 一二

本节将介绍以下内容：

- 有序数集合
- Tuple 及其应用

14.8.1 引言

Tuple，是函数式编程的概念之一，早见于 Erlang、F#等动态语言。不过，Tuple 很早被很多技术大牛实

现于.NET平台的实践中。由此可见，Tuple对于我们并不陌生，它并不是.NET 4.0的创造发明，但却是.NET平台语言趋于函数式编程概念的必要补充。

那么，首先来看看什么是 Tuple。

14.8.2 Tuple 为何物

什么是 Tuple？在汉语上我们将其翻译为元组。Tuple 的概念源于数学概念，表示有序的数据集合。在.NET 中 Tuple 被实现为泛型类型， n -Tuple 表示有 n 个元素的 Tuple，集合的元素可以是任何类型，例如定义一个 3-Tuple 表示 Date(Year, Month, Day)时可以定义为：

```
var date = Tuple.Create<int, int, int>(2011, 5, 5);
```

通过 Tuple.Create<int, int, int>将定义一个 Tuple<int, int, int>实例，该实例实现三个数据成员：

```
Console.WriteLine(date.D);
<Ctrl+Alt+Space>
    • CompareTo
    • Equals
    • GetHashCode
    • GetType
    ┌ Item1
    └ Item2
    └ Item3
    • ToString
```

可以有两个方面的理解，在.NET 中关于 Tuple 有如下的定义：

- 广义上，Tuple 就是一种数据结构，通常情况下，其成员的类型及数据是确定的。
- 狭义上，凡是实现了 ITuple 接口的类型，都是 Tuple 的实例。在.NET 4.0 BCL 中，预定义了 8 个 Tuple 类型。例如，最简单的 Tuple 定义为：

```
[Serializable]
public class Tuple<T1> : IStructuralEquatable, IStructuralComparable, IComparable, ITuple
{ }
```

其他所有的 Tuple 类型都实现了 ITuple 接口，该接口被定义为：

```
interface ITuple
{
    int Size { get; }

    int GetHashCode(IEqualityComparer comparer);
    string ToString(StringBuilder sb);
}
```

在该接口中，定义了一个只读属性 Size、两个覆写方法 GetHashCode 和 ToString，实现该接口的 Tuple 八大金刚如下：

```
public class Tuple<T1>
public class Tuple<T1, T2>
public class Tuple<T1, T2, T3>
public class Tuple<T1, T2, T3, T4>
public class Tuple<T1, T2, T3, T4, T5>
public class Tuple<T1, T2, T3, T4, T5, T6>
public class Tuple<T1, T2, T3, T4, T5, T6, T7>
public class Tuple<T1, T2, T3, T4, T5, T6, T7, TRest>
```

! 注

`Size` 属性、`ToString(StringBuilder sb)`方法，均被实现为显式接口方法，所以只能以接口实例访问，不过 `ITuple` 本身被定义为 `internal`，意味着无法在程序中访问 `ITuple`。

在下面的定义中，将 Custom Request 封装为 Tuple：

```
public class MyRequest
{
    public Tuple<string, Uri, DateTime> GetMyRequest()
    {
        return Tuple.Create<string, Uri, DateTime>("anytao.com", new Uri("http://anytao.net/"), DateTime.Now);
    }
}
```

为什么要用 `Tuple` 呢？这是个值得权衡的问题，上述 `MyRequest` 类型中通过 3-Tuple 对需要的 Request 信息进行封装，当然也可创建一个新的 `struct` 来封装，两种方式均可胜任。然而在实际的编程实践中，很多时候需要一种灵活创建一定数据结构的类型，很多时候新的数据结构充当着“临时”角色，大动干戈创建一系列的新类型完全没有必要，而 `Tuple` 就是为此种体验而设计的。例如：

- `Point {X, Y}`，可以表示坐标位置的数据结构。
- `Date {Year, Month, Day}`，可以表示日期结构；`Time {Hour, Minute, Second}`，可以表示时间结构；而 `DateTime {Date, Time}`，则可以实现灵活的日期时间结构。
- `Request {Name, URL, Result}`，可以表示 Request 的若干信息。

随需而取正是 `Tuple` 之精髓。

14.8.3 Tuple Inside

为了对 `Tuple` 一探究竟，我们应用 Reflector 工具打开神秘之门。就实现而言，`Tuple` 类型略显单薄，并没有什么“神奇”的设计，以 `Tuple<T1, T2>` 而言，其部分实现为：

```
[Serializable]
public class Tuple<T1, T2> : IStructuralEquatable, IStructuralComparable, IComparable,
ITuple
{
    // Fields
    private T1 m_Item1;
    private T2 m_Item2;

    // Methods
    public Tuple(T1 item1, T2 item2)
    {
        this.m_Item1 = item1;
        this.m_Item2 = item2;
    }

    string ITuple.ToString(StringBuilder sb)
    {
        sb.Append(this.m_Item1);
        sb.Append(", ");
        sb.Append(this.m_Item2);
        sb.Append(")");
    }
}
```

```

        return sb.ToString();
    }

    int ITuple.Size
    {
        get
        {
            return 2;
        }
    }

    // Properties
    public T1 Item1
    {
        get
        {
            return this.m_Item1;
        }
    }

    public T2 Item2
    {
        get
        {
            return this.m_Item2;
        }
    }

    //More and more...
}

```

其他的 Tuple 类型也大致如此，所以易于知晓 Item1、Item2、...、ItemN 是如何被定义的，同时纳闷 Size 属性将何去何从，打消了期望通过 foreach 来遍历 Tuple 元素的可能。

不过，对于 Tuple 而言，因为其元素数量的有限性，虽然能够满足大部分的需求，但是动态体验是越来越被期望的编程体验。同时，尤其注意 public class Tuple<T1, T2, T3, T4, T5, T6, T7, TRest> 初始化可能引发的 ArgumentException，例如：

```
var t8 = new Tuple<int, int, int, int, int, int, int, int>(1, 2, 3, 4, 5, 6, 7, 8);
Console.WriteLine(t8.Rest);
```

将引发异常：The last element of an eight element tuple must be a Tuple，提示最后的 TRest 应该为 Tuple，所以修改程序为：

```
var t8 = Tuple.Create<int, int, int, int, int, int, int, int>(1, 2, 3, 4, 5, 6, 7, 8);
Console.WriteLine(t8.Rest);
```

则没有任何问题，究其原因很容易从 Tuple<T1, T2, T3, T4, T5, T6, T7, TRest> 构造方法中找到答案：

```

public Tuple(T1 item1, T2 item2, T3 item3, T4 item4, T5 item5, T6 item6, T7 item7, TRest rest)
{
    if (!(rest is ITuple))
    {
        throw new ArgumentException(Environment.GetResourceString("ArgumentException_TupleLastArgumentNotATuple"));
    }
    this.m_Item1 = item1;
    this.m_Item2 = item2;
    this.m_Item3 = item3;
    this.m_Item4 = item4;
}

```

```

    this.m_Item5 = item5;
    this.m_Item6 = item6;
    this.m_Item7 = item7;
    this.m_Rest = rest;
}

```

因此，通过 `Tuple<T1, T2, T3, T4, T5, T6, T7, TRest>` 构造函数进行初始化操作，必须保证最后一个构造函数为 `Tuple` 类型，例如：

```

var trest = Tuple.Create<int>(8);
var t8 = new Tuple<int, int, int, int, int, int, int, Tuple<int>>(1, 2, 3, 4, 5, 6, 7, t
rest);
Console.WriteLine(t8.Rest);

```

而 `Tuple.Create` 静态方法隐式地实现了对最后一个类型参数的构造过程：

```

public static Tuple<T1, T2, T3, T4, T5, T6, T7, Tuple<T8>> Create<T1, T2, T3, T4, T5, T6,
, T7, T8>(T1 item1, T2 item2, T3 item3, T4 item4, T5 item5, T6 item6, T7 item7, T8 item8);

```

`TRest` 类型参数必须被实现为 `ITuple`，否则将引发异常。`TRest` 在某种程度上为元素的扩展带来方便，但是仔细想来，总觉得此处 `TRest` 的设计有点多此一举。既然是类型参数，那么 `T1、T2、...、TN` 其实均可为 `ITuple` 实例，何必非拘泥于最后一个。

另外，从 `Tuple` 的定义可以看到实现了两个陌生的新接口：`IStructuralEquatable` 和 `IStructuralComparable`，其中 `IStructuralEquatable` 与熟悉的 `IEquatable` 同宗，而 `IStructuralComparable` 与 `IComparable` 同源：

```

public interface IStructuralEquatable
{
    bool Equals(object other, IEqualityComparer comparer);
    int GetHashCode(IEqualityComparer comparer);
}

public interface IStructuralComparable
{
    int CompareTo(object other, IComparer comparer);
}

```

所以，非常容易理解其为 `Tuple` 的恒等性和排序性提供了支持，以恒等性为例：

```

// IStructuralEquatable
var t1 = Tuple.Create(1, "Hello");
var t2 = Tuple.Create(1, "Hello");
var t3 = Tuple.Create(2, "AT");

Console.WriteLine(t1.Equals(t2));
Console.WriteLine(t1.Equals(t3));

```

而 `Tuple` 对于 `IStructuralEquatable` 和 `IStructuralComparable` 的具体实现，留待读者继续探索，以进一步了解 `Tuple` 的恒等性和排序性本质。

14.8.4 优略之间

当前，.NET 4.0 预定义的 `Tuple` 类型仅有 8 个，所以应考虑对于 `Tuple` 提供适度扩展的可能，然而遗憾的是 `ITuple` 类型被实现为 `internal`，所以无法直接继承 `ITuple`，只好自定义类似的实现。

优势所在：

- 为方法实现多个返回值体验，这是显然的，`Tuple` 元素都可以作为返回值。

- 灵活构建数据结构，符合随要随到的公仆精神。
- 强类型。

不足总结：

- 当前 Tuple 类型的成员被实现为确定值，目前而言，还没有动态决议成员数量的机制。
- public class Tuple<T1, T2, T3, T4, T5, T6, T7, TRest>，可能引发 ArgumentException。

14.8.5 结论

Tuple 是一剂调味剂，为.NET 的语言特性增加了更多动态色彩。合理应用 Tuple 为语言赋予了更多优异的特性，例如从方法获取多个返回值，赋予简单参数更多的可能值，自然成全了语言的简洁、灵活与可读。

参考文献

Daan Leijien & Judd Hall, Optimize Managed Code For Multi-Core Machines,

<http://msdn.microsoft.com/en-us/magazine/cc163340.aspx>

Chaur Wu, Pro DLR in .NET 4, Apress

TIOBE Programming Community Index for June 2011,

<http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html>

Jon Skeet, Implementing the Singleton Pattern in C#,

<http://www.yoda.arachsys.com/csharp/singleton.html>

Yuhen, Dynamic, <http://www.rainsts.net/article.asp?id=876>

Dino Esposito, C# 4.0 then Dynamic Keyword and COM,

<http://msdn.microsoft.com/zh-cn/magazine/ff714583.aspx>

RednaxelaFx, LINQ 与 DLR 的 Expression Tree (1): 简洁 LINQ 与 Expression Tree,

<http://rednaxelafx.iteye.com/blog/237822>

Stephen Toub, Patterns of Parallel Programming

Joe Duffy & Ed Essey, Running Queries On Multi-Core Processors,

<http://msdn.microsoft.com/en-us/magazine/cc163329.aspx>

施凡, .NET 4.0 中的泛型协变和反变,

http://www.cnblogs.com/Ninputer/archive/2008/11/22/generic_covariant.html

Covariance and Contravariance in Generics, <http://msdn.microsoft.com/en-us/library/dd799517.aspx>

Anatoliy Kolesnick, C# 4.0: Covariance and Contravariance,

<http://blog.t-l-k.com/dot-net/2009/c-sharp-4-covariance-and-contravariance>

Idior, Covariance and Contravariance ,

<http://www.cnblogs.com/idior/archive/2010/06/20/1761383.html>

后记：我写的不是代码

为什么我的眼里常含泪水，那不是眼屎没有擦干净，只是因为我的眼神里充满了：寂寞。

我是个正宗的 80 后，虽然在微博的账号常年显示着 90 后，但那仅仅是个符号，为了艳遇准备的而已。为什么要写我的故事，那是因为我是一个天才（其实不是，假装而已）。

我是如此的谦虚，以至于不想将自己是天才的秘密告诉任何人，因为我始终信奉：谁终将声震寰宇，必先深自缄默。首先，这个秘密也不是自封的，是金子总会发光。关于我天才的故事，你可以顺道向 dudu、TerryLee、dingxue、老赵、脑袋这样的先驱打听，不过如果他们不承认知道这个秘密，我也无可奈何，我还不至于冒着潜规则的风险用委托的方式递归，循环太多，是会溢出的。我为什么是天才，这是有事实基础的事实，因为我有求必应。如果你有任何的内存溢出、Unknown exception、死循环、Silverlight 翘辫子和 MVC 不好用等问题不能解决，可以试着朝厕所喊我三声，问题将迎刃而解，效果立竿见影。但是，如果收效甚微，那肯定是你心诚不灵，可以不断重复以上练习，直到解决为止。我已经靠这种办法，成功解决了很多问题，所以我相信下一次你也能成功。

话说回来，天才的秘诀其实很简单，这也是很多粉丝都在打听的小道消息，答案并不复杂：

外事不决问 Google，内事不决问 Baidu，八卦不决问 Mop。

作为天才，我实在不称职，因为过往的岁月打酱油、叉腰鸡、俯卧撑这些露脸的好事儿我都没赶上。最后，还被贾君鹏同学抢了风头，其实我妈以前经常在女同学家喊我回家吃饭的。不过，作为偶像，我也并非一无是处。一个风和日丽、电闪雷鸣的夜晚，我诞生在秦王一统河山的骊山之旁，渭水之畔，那一年应该是比尔·盖茨在玩儿 DOS 的日子，虽然他已过了穿开裆裤的日子，但是我还在穿。我与生俱来的气质并没有影响到远在美国的盖先生，却引来计划生育队查户口，所幸我们是老大，这让俺爹娘长长地舒了一口气，以至于后来我妹出生的时候，他们都庆幸天才先一步落地了。

后来，我一直深信那天的吉兆将是我非凡气质的开始，我所选择的创造必将不同凡响。直到今天，我的理想（仅限于理想）一直被模仿，却从未被超越。

为了让您有机会了解我天才的往事，我决定不从 1 岁尿炕的日子说起，而直接飞跃到青春期懵懂的岁月。那一年，我 5 岁。班里一个胖脸蛋儿小朋友引起我寂寞眼神的青睐，几年之后我们交换了照片，然后什么也没再发生。虽然这段马拉松式的爱情长跑仅仅维持了几个月，但是在我初恋的记忆永远埋下了阴影，以至于后来的我饱受暗恋的打击，也仍然坚挺地活着。就这样结束了，没有开始，但却有结束，这段爱情故事至今仍在村子里被传颂，不信你可以问村长。

我知道看客们就喜欢八卦，所以先说说我的青春往事，但我并不打算继续沿着这个主题走下去，因为正如标题而言：哥，是写代码的。在我传奇的上个世纪，是另一个偶像把我的目光由爱情转到了代码，这个偶像就是前面提到的穿开裆裤的那个人。当然，能作为我的偶像，也是他上辈子修来的福。那个时候，除了金

庸和琼瑶，最拉风的还有比尔·盖茨先生的光辉事迹，在撬动我激情燃烧的心。

这就是我寂寞的开始。

我告别了寂寞，却选择了更大的寂寞，后来我听窦文涛说天才本来就是寂寞的，就一阵窃喜，这也反过来印证了我确实是个天才。而这个更大的寂寞就是代码，所以，如果你也是有幸写代码的，那么首先恭喜你像我一样生来异秉，然后再恭喜你继续寂寞着。小时候的光辉很快在生活的无聊下消失殆尽，现在的我被人定义为“大器晚成”，因为写代码的人很多都是从5岁开始的（记得吗，5岁的时候，我的世界只有女人——胖脸蛋儿小朋友），而我是从20岁才开始的。晚成的具体时间得追溯我的第一台计算机，那是上大学一年级的时候，为了更好地练习看片和游戏的工作需要，借学习之名骗老爸购置了人生的第一台586，所以，我的故事才刚刚开始。

如果把那台老朋友的配置列个单子，相信90、00后的新生代肯定知道原来电脑也有上古时代，20GB硬盘、128MB内存和600M CPU已经迎来宿舍同仁持续一个学期的尖叫。作为史前人类，那个时候我并没有把太多的注意力关注在南桥和北桥的区别，也不去想0和1的世界规则，更没有时间理会算法和链表。因为时间宝贵，我首先选择献身艺术，从《东京爱情故事》到《流星花园》，从《古惑仔》到《那场风花雪月的事》，从港台大佬到日韩情人，我追寻电视艺术的脚步从来舍不得停滞，我一度怀疑这和十几年前的那场柏拉图式的早恋有关。

而转折就像放屁，总是发生在不经意间，永远没有前奏。

大学三年级，是开始做项目课题的日子，由于深受众多“男叫兽”的压迫，我选择在美丽贤惠的S老师手下做事，S老师并没有多少黑社会老大的脾气，总是和蔼可亲，混吃混喝很方便。做S老师的搭档，正应了那句名言：出来混，迟早是要还的。由于压迫计算机很多年，在换了无数个光驱（看片）和键盘（打游戏）之后，老祖先冯·诺依曼先生终究没有放过我，现在开始计算机压迫我的时代了。学机电出身的我，开始跟着S老师搞起了软件项目，那是游戏和图片以外的东西，我从此被逼无奈地开始了写代码的生涯，在0和1的世界里义无反顾。而我，在很多后来的大牛眼里，属于出身不好的那一类，好像一直以来的代码江湖都是以C++为正统，其他门派都是提鞋的，世袭罔替直到今日。所以，提起出身我都不好意思再提过去的天才故事，因为我不是从C++开始的，而且一直不是，我是从拖VB控件开始的，可能你们一阵窃喜，但是我的心里也并未哇凉哇凉的。这是为什么呢？虽然拖控件看起来级别不高，但是我至今仍然被这句名言所震撼：我脱了，你随意。后来发生的事情也证明，VB、Delphi还有C#，其实并不是拖来拖去那么简单，你也可以很随意，在关注自家GC的同时，去挖COM和C++的墙角。

跟着S老师混，是喜剧的开始，也是寂寞的开始。

我犯了所有男人的通病，为了在女老师面前逞强，我放弃了所有的爱好和虚荣，我放弃了数十年来被评为我最喜爱的赵薇、酒井法子和金泰熙海报，我放弃了学校食堂每天等候欣赏天才的美女和大妈，我头悬梁，我锥刺股，我自大，我心虚，我盲目，我冲动，我迷失，我炼狱，一门心思做起了技术。为了苦练拖控件神功，我试图不拖会怎样，结果发现不脱更刺激。手动控制由代码组织的命名空间、类型对象和事件机制，了解静态成员为什么理直气壮，分析委托对象何以能打回马枪，轻叹多态继承总是多快好省。不经意间，我把老师的不经意项目做成了优秀作品，老师拿着我写的长长代码，眼泪哗啦哗啦的，没有流出来。拖控件也能拖成这样，真是没想到呀，老师语重心长地说道。我说，只要老师满足了，我拖再多也愿意。

优秀的结果，就是我被保送给老师的老公C教授，C教授赫赫有名，他是计算机与信息学院院长。听这

头衔，你就知道我不能逃离代码江湖的深渊了。我还没有从逞能的兴奋中过渡，就结实地掉进了另一个泥潭，说起来吓死你，那天我喝了一整瓶汉斯果啤，在醉梦中仿佛看见了幼时初恋小朋友在得意地笑。那种状态是刚刚好的状态，以后烦闷当头，我都选择以半斤二锅头或几瓶燕京来结束良知。但当时，作为男人，在自信不能被打倒的同时我选择了沉默，正如鲁迅先生所说，不在沉默中爆发，就在沉默中咯屁，而我选择了在沉默中寂寞。

什么是寂寞，寂寞还是写代码，这个江湖还在继续。

在和 C 老师的斗争中，我逐渐学会了批评和批评他人，吹牛和自我吹牛，本文就是例证。同时，也完成了从“逼良为娼”到“丫从良了”的蜕变，技术对我来说不再是泄欲工具，而变成了立体山水。原因很简单，常在河边走，怎能不湿鞋。每天在 Visual Studio 上狂奔，怎能不汗流浃背，汗流的多了，就不是汗了，是寂寞。不过，说起来，这些蜕变并非偶然，而是我逐渐发现的秘密：代码和人生，其实是一样一样的。总结起来，其共同点主要包括：

- 善变。人的善变就像阴晴圆缺，没有启奏，只有通知；人生的善变就像悲欢离合，没有预期，只有结果。代码，也是同样的，任何 Bug 的发生，都没有一定可循的轨迹。
- 0 和 1。如果女人是 0，那么男人一定就是 1 了。人生正是如此，0 和 1 组成的计算机世界之精彩，正如八卦阴阳支配的自然世界一样丰富多彩。
- 自私。人是自私的，技术也一样。很多朋友在来函中都困惑未来的前途，是搞.NET 还是搞 Java，我说技术不是靠搞的，博学仅限于外星人，专注才是正道。对于我这样的人，只信奉这样的教条：信.NET，坚持到底。

从此，我就把代码当成了人生之必需，尤其是深夜时分，(关灯) 此处省略 500 万字……关灯之后的故事，被我写在了《你必须知道的.NET(2.0 版)》一书中，由此可见这本书凝聚了多少心血和高潮。如果你想了解.NET 的万种风情，可以翻开看看，因为她是天堂；如果你想了解代码的递归循环，可以翻开看看，因为她是地狱。如果看了 8 遍，还是没有任何起色，这只能归结于慧根：仁者见仁，淫者见淫。

所以，你可以把标题中的代码二字换成人生，全文其实可以基本通顺。然而，要当天才，是需要成功证明的，但是成功的基本要素却不是人人都有，我总结的要素不多，但都很难办：

- 虚伪。关于虚伪，很多人都信奉：哥不是随便的人，哥随便起来不是人。而天才的信念没有这么多拐弯，我们只承认，哥即使不随便，也绝不是人。写代码的生活，就是一个活在虚伪状态的疯狂之人，才能一直坚守的事情。
- 痒好。关于痒好，据说成功的人都有痒好，例如 X 成功绅士喜欢抠脚，而 Y 成功大婶热衷挖鼻子。我的痒好，就是认准不撒手，一直一直地追着走，显然美丽的.NET 姑娘是经受不了这种死缠烂打式的轰炸，最后只好从了。
- 理想。现在的我，变得豁达。如果你意图告别喧嚣，那么就不要怠慢理想，你可以选择放弃狗屁，但是不要轻言放弃理想。老赵说他的理想是写代码到 60 岁，而我也希望能够寂寞这一生。
- 选择。选择总是难免，而我选择了.NET。说起原因，听来可笑，最初的打算并没有和多少前途和钱途挂上钩。就像姑娘，你总是先挑对眼儿的，不管是豆蔻年华型，还是风韵犹存型，对眼儿总是第一重要的。两情相悦至少，一切前途的担忧与一切钱途的计较，全是扯淡。由此，我为人生选择了对眼儿的姑娘——.NET，与自己在未来的若干个日子里比翼双飞，一直时至今日。俗话说，情人眼里出西施，

我眼里的.NET也是很西施的。我喜欢托管世界的自由自在，不必为内存管理而分心；我痴迷 IL 和 SOS 底层的无线奥秘，把揭开真相作为一种乐趣；我欣赏 Lambda 表达式和 LINQ 造就的优雅；我也沉醉 C#行云流水般的简洁代码。

虽然并非凡人，但是被我喜欢的.NET 姑娘，更是个天才。我看着它从 1.0、1.1、2.0、3.0 到 4.0，一把屎一把尿地成长起来。虽然年轻，可这孩子却一点儿不让人省心，我始终没有完全搞懂它肚子里的花花肠子到底是怎样的格局，或许这终将是个谜。所以，在它面前，我没有高贵的头颅，也没有骄傲的资本，虚心请教、任人宰割的事情常有发生，以至于我对自己怀才不遇的事实感到怀疑。

钟爱的原因有很多，如果你还是想问个问什么，我只能说：我很傻，很天真。没办法，天才也有软肋。

至于是否有第三者（F#、Erlang、Ruby、Python）插足，或者被横刀夺爱（C++、Java），我都无所谓，至少当下我爱得很幸福。老牛人常说，武林是没有门派之分的，搞定一个就可举一反三，并且将最上乘的技术总结为“手中无剑而心中有剑，此时无剑胜有剑”，而我很早就对这句屁话有所怀疑，每次和人过招鼻青脸肿的都是因为没家伙。不信你拿牛刀和牛人练练空手道，看看是有剑的流氓，还是无剑的冤枉。

因为爱上寂寞，随后的事情就变得顺理成章，可以说一发不可收拾，我最终和代码干上了！研究生毕业，我一把鼻涕一把泪离开了每天喊我回家吃饭的慈母严父，只身协调准媳妇来到北京浪迹江湖。刚下火车，我的第一反应是去买张回程票，老子不干了。首都的楼儿高又高，小可的心里漂又漂，最终留下的原因是发现这里的美女也是又高又飘。于是几年前的我，在当时写下了这样的壮语：

数年闻道，一朝醒悟，身于技术，心系天下。内专注于.NET，外切磋于Java，只身江湖于京城，志满筹。
心若凭栏，以立业传道为己任；神思四海，为技术中华而躬身。俱五内，为品茶、读书、立业、写代码而悠然；放五洲，以修身、齐家、治国、平天下为己任。

现在想想，还不是一般的傻。5 年都过去了，我还是这副德性，发起闷骚，仍然是当仁不让。

既来之，则安之，我进入一家国有软件企业，从此接触了更广阔的软件空间。大舞台，当然要有大动作，我的动作很大，基本格局是从早到晚加班，周六周日无休，提成奖金没有。庆幸的是，我在灰天黑地的折磨中，看到了小项目之外的大项目是如何组装、包装到安装，看到 VB 之外、.NET 之外的更多技术是各般模样，看到了项目管理与市场陷阱，看到了办公室真哲学和 KTV 潜哲学，看到了知己和过客在社会阶段的角色，我看到的太多了，所以没有记下太多。在还没有看清太多的时候，抱着体验洋企业的心态，我来到一家外企公司，继续自己关于代码还有寂寞的人生，这一切都是神奇的过程，哥相信人生本来就应该这样神奇。继续的路还有很多，未来的梦还在浮想联翩，即将开始的创造又该是怎番模样，且等下回分解。

如果你相信，任何故事，远没有结束的时候，现在的人还不是永恒不变地重复历史上曾经发生的某个缩影。为了体验多活几次的别样生活，我喜欢在历史和现实中穿梭，你可以负责任地品评历史上所有美女和帅哥，也可以不负责任地将自己对号入座，同时发现原来也和程序世界有惊人的复制可能。已经发生的、正在发生的、将要发生的，不管是什么，都印证了时下最流行的哲理：

哥玩的不是代码，是寂寞。

对我而言，经历总在继续而且难免，但是每个阶段怀揣着感恩的心，为每个阶段的寂寞说声：谢谢，辛苦了。否定别人，首先就否定了自己，不以好坏论长短。

时间过得太快，盖茨退休了，云计算来了，.NET 都论 4.0 了，Silverlight 可以离线了，WCF 支持 Restful

了, MVC 以 Razor 展现, 微软以 Windows Phone 重整旗鼓, Facebook 连接了全世界, Google 埋头苦干 Google+, 而苹果将 iPhone 扔向全世界, 于是《你必须知道的.NET》该第2版了。新鲜每天发生, 故事重复传奇, 这让人时常感怀“过去的日子”, 每日游走于技术、八卦和快乐, 我不想再对寂寞说抱歉。

总体说来, 作为天才(其实不是), .NET 传奇还在继续, 因为绝少有人能够续写, 所以我只好耐着头皮装傻充愣。而写本文, 其实只是娱乐大家, 如有雷同, 就当被雷。故事人物, 除了我并非天才这一基本原则, 其他人物和故事全部虚构, 切勿对号入座。

“涛, 回家吃饭了”, 你看, 我没有骗你吧, 是时候, 我妈又在喊我了。于是, 我带着这本全新面貌的《你必须知道的.NET (第2版)》, 回家了。

对了, 忘了提醒: 不要崇拜哥, 这只是个传说。

王涛

2011年6月, 于北京

(编程之余, 以调侃的方式记录一点技术人生, 仅供娱乐。)

编后记：遇见幸福

记得曾看过一篇文章，描述过编辑应要有一种激情，“这种激情表现在发现好书稿时的拍案叫绝，在找到好作者时的相见恨晚感动莫名，在与作者讨论书稿时的当仁不让，在策划选题方面的绞尽脑汁，在办事效率方面的雷厉风行。”这本书让我经历了这其中的各种滋味。

当我第一次在博客园看见作者的《你必须知道的.NET》文章当中的一篇，着实被作者的文采所深深吸引，在拍案叫绝的同时也满心疑虑：是谁会把这么个枯燥艰涩的东西写得如此耐人琢磨；是谁胆敢用这么个强制性极强看似威严傲慢无礼的名字，“你必须知道的”；是谁在大多数图书教的是如何应用.NET的大环境中居然能够深入.NET 内核 CLR；是谁在大家忙碌浮躁的年代里能够做到文字优雅而精到，一个大系一天一篇地出；……不管他是谁，先联系再说吧！

由于这些满心疑虑，我们找了一些专家进行初步审定，这些专家当初的话仍然记忆犹新：“有意思！”“哎，这本书出来告诉我下”“嗯，可以休息的时候翻翻”“不太适合初学者，得要一些基础学才行”“把全部书稿寄过来吧”……不管了，决定做吧！

交稿速度完全出乎意外的快！还以为，决定了之后怎么着都得大半年，但在我都没感觉到应该去催稿的时候稿件已经完好地躺在我的邮箱了。于是乎，我迫不及待地打印了前 10 页看，哇，不错呀！全部打完看吧！在不知不觉中，稿件基本上翻阅完毕，整个过程下来，几乎忘记了自己作为编辑的职责……最后，我语气平缓地反馈了一句话：“是我这么多年做编辑以来看的最好看的技术书”。

不知道大家看完有何感想，如果有，千万不要吝惜你的文字，请发送到 sxy@phei.com.cn。大家看完之后，可能会体会到上面第一段话当中的“谁”是个什么样的人了吧！告诉大家，这个“谁”不是神人，不是牛人，只不过是中国广大程序员的普普通通的一员，在出版过程当中，感觉到他比普通人多的可能只是平常的勤奋、认真、谦逊和沉淀，关键是善于沉淀！

当然，如果您有信心让我们作为出版者再次经历一次“在发现好书稿时的拍案叫绝，在找到好作者时的相见恨晚感动莫名，在与作者讨论书稿时的当仁不让，在策划选题方面的绞尽脑汁，在办事效率方面的雷厉风行”这种莫大的幸福，请尽管联系我们 sxy@phei.com.cn。



你必须知道的.NET (第2版)

我不清楚翻开这本书的你是否看过了《你必须知道的.NET》第1版。如果你认为这本书只是上一本书的添头或者修改那就大错特错了。现在音乐界流行老歌翻唱，几十年前的歌曲，随便换个编曲就可以再卖一次；电影界也是动不动就来个什么什么怀旧版，什么什么经典再映。归根结底就是再从我们这些劳苦大众兜里套点银子出来。但是这本书却不是前一本的所谓“新歌加精选”。虽然我只是看到这部书的两个样章，但是还要惊叹于这本书所涉及的内容之广。见闻之深。

在这本书当中，我看到的不仅仅是和第1版一样对于.NET底层深入的研究和完整的介绍，还能够看到作为一个在.NET阵营打拼了多年的架构师对于系统架构、设计模式、面向对象等诸多方面的经验、体会以及探索。

徐子君

《实战Windows Azure：微软云计算平台技术详解》作者

博客是一块地，写博客是一种耕耘，**这本是作者辛勤耕耘的一份收成**。基于作者发表在博客园的精品系列文章精心写成的书，相信一定会给读者带来很多收获。

杜勇

网名duodu，http://duodu.cnblogs.com/，微软MVP，
国内最具影响力的.NET技术社区博客园创始人

看过《射雕英雄传》的人都知道，郭靖如果不是受过马钰两年内功的训练，单是江南七怪十几年的招式练习，是不可能学会降龙十八掌，并最终成为绝世高手的。只练招式，那是徒

有其表，遇到稍有内功修炼的武者，就将败下阵来，而内功越深，水平也就越高。要成为高手，必须修炼内功。

本书就是一本修炼.NET内功的书。你可以通过一本.NET入门书几天就学会开发一些小程序，并根据自己的爱好学些编程技术和技巧，但如果你真的想成为.NET的专业高手，想靠它吃饭，靠它发展自己的事业，那么请阅读本书吧。**本书没有以往国内书籍抄袭或拼凑文字的浮躁，也没有国外资料因翻译或文化差异所造成阅读的困惑，而更多的是对.NET底层实现的剖析。**或许阅读之前，你会觉得自己知道的.NET已经很多，但当你读完本书，你会感受到，原来.NET还有很多必须知道的内容我并不知道。

桂杰

网名伤心，http://qj723.cnblogs.com/，博客园专家，畅销书《大话设计模式》、《大话数据结构》的作者

有很多.NET开发人员对于应用层面的东西能够很快掌握，但在脑海里并没有对.NET本质的东西形成一个系统的认识，垃圾回收有着什么样的奥秘，委托、匿名方法、Lambda表达式之间有着怎样的进化关系，**本书正是围绕这些看似平常不过的概念而展开的，一步一步带您进入.NET底层世界，这是一本值得推荐的好书。**

李金宇

网名TerryLee，
http://terrylee.cnblogs.com/，博客园专家，
微软ASP.NET方向最有价值专家，IT168专栏作者



责任编辑：孙学瑛
封面设计：李玲

本书贴有激光防伪标志，凡没有防伪标志者，属盗版图书。

上架建议：计算机>程序设计>.NET



ISBN 978-7-121-14128-7



9 787121 141287 >

定价：79.00元