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

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。

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY
http://www.phei.com.cn

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。

前言　Expect the unexpected

时间就是一条生生不息的河，总在流淌，永不回头。时光激荡在沉淀、叛逆和重生的人生里，总在告诉你，答案就在下一站。《你必须知道的.NET（第 2 版）》距离第 1 版的问世，已经过去整整三年了，三年的时光里，技术与人生都在前行的路上，彼此影响。而留在下一站的答案，迟迟不见你来，直到三年之后。

变化每天发生，创新无所不在，.NET 正昂首阔步迈向成熟与融合，面对充满变化的技术世界，我们要做的就是：Expect the unexpected。

新版有什么

辗转三年，第 2 版延期而至，增加的不多，修改的不少。然而，二版的写作却是一个艰难的过程，我发现想和读者分享的还有很多很多，长长的写作 List 让人望而却步。如果照着趋势继续，可能还需要另外的 700 页和另外一年多的时间，选择和舍弃变得两难，痛定思痛最终呈现在你面前的就是手中的这本《你必须知道的.NET（第 2 版）》。

新增章节

- 第 3 章 “OO 之美”，面向对象的魅力体现在设计的多个方面，借助于.NET 平台语言的特性，领略面向对象更多的奥秘。从设计的分寸体验 OO，以依赖的哲学铺开讨论，认识闭包的.NET 体现，然后知道什么是好代码和坏代码。
- 第 14 章 “跟随.NET 4.0 脚步”，通过.NET 十年的历史演义，为.NET 4.0 的新特性大幕拉开边角，从此踏上并行计算、动态编程的康庄大道，同时还有很多激动人心的 4.0 新体验。
- 4.4 节 “管窥元数据和 IL”，离开了元数据支持，CLR 的很多奥秘将变得苍白和空洞，而本节为很多疑惑给出了答案。
- 7.4 节 “认识全面的 null”，null 关键字是神奇而普通的一等公民，认识全面的 null 将能收获更多语言细节。
- 9.3 节 “疑而不惑：interface‘继承’争议”，通过一段争议，展开一段讨论，其中有质疑也有深入，究竟谁是谁非，且看文中乾坤。
- 9.4 节 “给力细节：深入类型构造器”，类型构造器的讨论，深入到了这个话题的底端，体现了细节深处的语言美丽。另一方面，细节也折射了平常编码中，需要特别关注的部分，为写出高质量代码打好基础。
- 9.8 节 “Name 这回事儿”，你认识 Name 吗，或许认识，或许还没有全面的认识，一个简单的话题，丰富 Name 这回事儿。

- 11.4 节 “实践泛型”，作为最佳实践系列的代表篇章，泛型的应用有很多值得推敲和总结的建设性思想，以条款性的律条，总结出泛型应用中的最佳答案。
- 13.4 节 “LINQ 江湖”，从历史角度演义 LINQ 发展的脉络，从演义的点滴故事中，认识语言的特性，是如何而来，又因何而来。我们调侃的不光是 LINQ，还有杰出工程师们的划时代工作。

除了新增内容，还有某些章节的修订和完善。

修改章节

- 8.3 节 “历史纠葛：特性和属性”，增加了较常见的 Attribute 属性应用。
- 8.9 节 “集合通论”，为 Hashtable 实现顺序输出逻辑。
- 9.1 节 “万物归宗：System.Object”，认识好玩的 EditorBrowsableAttriute，为 Object 成员装扮隐身符。
- 9.5 节 “如此特殊：大话 String”，更多关于字符串本质和字符串驻留的讨论，认识更全面的 string 类型。

篇章的修订，为更多的内容打好补丁，还有将来更多的新内容，将不断的发布在本书支持站点。

支持

虽然作者、审稿和编辑花费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，仍难避免失误或错误。为了使本书能更好地服务于读者，请您通过以下方式与作者或者出版社联系：

- 本书支持网站：http://book.anytao.net/
- 博文视点网络：http://www.broadview.com.cn/
- 作者个人信息：anytao@live.com（邮箱）、http://weibo.com/anytao（微博）
- 策划编辑：http://weibo.com/sunnypub（微博）

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、作者博客（http://anytao.cnblogs.com/）或者作者微博（http://weibo.com/anytao）进一步取得技术支持联系。

本书支持网站（http://book.anytao.net/）提供了所有代码资源、工具资源、勘误、更新内容及与作者的互动，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

感谢子岩和会军的审稿和推荐，他们的神来之笔，为本书的品质注入坚实的保证，也让我信心满满地将自己对技术的理解再次展现在世界面前。

感谢我的父亲、母亲还有家人，他们永远是我可以归宿的港湾，是我人生的支点；感谢 Emma 的体贴入微，让我呼吸在自由的空气里，越发精神抖擞；感谢妹妹王佳，她长大了。

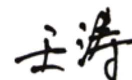
感谢我的朋友徐子岩、罗炳桥、徐彦华、张玉斌、高泽东、易湘、管伟、申毅、达伟、杜勇、吴宏杰、李嘉对我一如既往的关爱和支持；感谢汤文海老师、陈桦老师、Philana、Olav、方浩，他们指导我人生的历程；感谢大磊、春雨、会军、德宇、天卓、吴飞，我们即将揭开新的人生篇章，并肩向前，为技术改变世界的梦想挥洒激情。

本书的出版历经了岁月的考验和折磨，一直坚持不懈的是本书的编辑孙学瑛老师，她的专业精神和专注品质，支持我向前努力。

当然，最重要的感谢要说给一直以来关注作者和这本书的技术同行者，是你们的热情、肯定、反驳与真诚，鼓励一个以软件为人生目标的靠谱年轻人，坚持走在技术的美丽云彩下。

生活是妥协的艺术，技术又何尝不是，然而妥协不是无谓的选择，而是有选择的 *Expect the unexpected*，因此生活才有了一副美景：唱着歌，写着代码，吃火锅。

还等什么，我们出发。



2011 年 6 月，于北京

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，往往会陷入其系统之中，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.NET 3.0/3.5 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。

2008 年 1 月，于北京

目　　录

第 1 部分　渊源——.NET与面向对象

第 1 章　OO 大智慧 2

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

第 2 章　OO 大原则 41

2.1 OO 原则综述 42
2.1.1 引言 42
2.1.2 讲述之前 42
2.1.3 原则综述 43
2.1.4 学习建议 44
2.1.5 结论 44
2.2 单一职责原则 44
2.2.1 引言 44
2.2.2 引经据典 45
2.2.3 应用反思 45
2.2.4 规则建议 47
2.2.5 结论 48
2.3 开放封闭原则 48
2.3.1 引言 48
2.3.2 引经据典 48

2.3.3 应用反思 49
2.3.4 规则建议 52
2.3.5 结论 53
2.4 依赖倒置原则 53
2.4.1 引言 53
2.4.2 引经据典 53
2.4.3 应用反思 54
2.4.4 规则建议 56
2.4.5 结论 56
2.5 接口隔离原则 56
2.5.1 引言 56
2.5.2 引经据典 56
2.5.3 应用反思 57
2.5.4 规则建议 59
2.5.5 结论 59
2.6 Liskov 替换原则 59
2.6.1 引言 59
2.6.2 引经据典 59
2.6.3 应用反思 60
2.6.4 规则建议 61
2.6.5 结论 62
参考文献 62
第 3 章 OO 之美 63
3.1 设计的分寸 64
3.1.1 引言 64
3.1.2 设计由何而来 64
3.1.3 从此重构 65
3.1.4 结论 67
3.2 依赖的哲学 67
3.2.1 引言 67
3.2.2 什么是依赖，什么是抽象 68
3.2.3 重新回到依赖倒置 73
3.2.4 解构控制反转（IoC）和依赖注入（DI） 79
3.2.5 典型的设计模式 82
3.2.6 基于契约编程：SOA 架构下的依赖 83
3.2.7 对象创建的依赖 84
3.2.8 不规则总结 87
3.2.9 结论 87
3.3 模式的起点 87
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
3.4 面向对象和基于对象 91
3.4.1 引言 91
3.4.2 基于对象 91
3.4.3 二者的差别 91
3.4.4 结论 92
3.5 也谈.NET 闭包 92
3.5.1 引言 92
3.5.2 什么是闭包 92
3.5.3 .NET 也有闭包 93
3.5.4 福利与问题 95
3.5.5 结论 96
3.6 好代码和坏代码 96
3.6.1 引言 96
3.6.2 好代码、坏代码 97
3.6.3 结论 105
参考文献 105

第 2 部分 本质——.NET深入浅出

第 4 章 一切从 IL 开始 108
4.1 从 Hello，world 开始认识 IL 109
4.1.1 引言 109
4.1.2 从 Hello，world 开始 109

4.1.3 IL 体验中心 ······ 109
4.1.4 结论 ······ 113
4.2 教你认识 IL 代码——从基础到工具 ······ 113
4.2.1 引言 ······ 113
4.2.2 使用工具 ······ 113
4.2.3 为何而探索 ······ 115
4.2.4 结论 ······ 116
4.3 教你认识 IL 代码——IL 语言基础 ······ 116
4.3.1 引言 ······ 116
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
4.3.5 数据加载与保存 ······ 118
4.3.6 流程控制 ······ 119
4.3.7 结论 ······ 120
4.4 管窥元数据和 IL ······ 120
4.4.1 引言 ······ 120
4.4.2 初次接触 ······ 120
4.4.3 继续深入 ······ 123
4.4.4 元数据是什么 ······ 125
4.4.5 IL 是什么 ······ 128
4.4.6 元数据和 IL 在 JIT 编译时 ······ 130
4.4.7 结论 ······ 134
4.5 经典指令解析之实例创建 ······ 134
4.5.1 引言 ······ 134
4.5.2 newobj 和 initobj ······ 134
4.5.3 ldstr ······ 136
4.5.4 newarr ······ 137
4.5.5 结论 ······ 139
4.6 经典指令解析之方法调度 ······ 140
4.6.1 引言 ······ 140
4.6.2 方法调度简论：call、callvirt 和 calli ······ 140
4.6.3 直接调度 ······ 142
4.6.4 间接调度 ······ 146
4.6.5 动态调度 ······ 147
4.6.6 结论 ······ 147
参考文献 ······ 147
第 5 章 品味类型 ······ 148
5.1 品味类型——从通用类型系统开始 ······ 149
5.1.1 引言 ······ 149
5.1.2 基本概念 ······ 149
5.1.3 位置与关系 ······ 150
5.1.4 通用规则 ······ 151
5.1.5 结论 ······ 152
5.2 品味类型——值类型与引用类型 ······ 152
5.2.1 引言 ······ 152
5.2.2 内存有理 ······ 152
5.2.3 通用规则与比较 ······ 156
5.2.4 对症下药——应用场合与注意事项 ······ 158
5.2.5 再论类型判等 ······ 159
5.2.6 再论类型转换 ······ 159
5.2.7 以代码剖析 ······ 160
5.2.8 结论 ······ 167
5.3 参数之惑——传递的艺术 ······ 167
5.3.1 引言 ······ 168
5.3.2 参数基础论 ······ 168
5.3.3 传递的基础 ······ 169
5.3.4 深入讨论，传递的艺术 ······ 170
5.3.5 结论 ······ 174
5.4 皆有可能——装箱与拆箱 ······ 175
5.4.1 引言 ······ 175
5.4.2 品读概念 ······ 176
5.4.3 原理分拆 ······ 176
5.4.4 还是性能 ······ 179
5.4.5 重在应用 ······ 180
5.4.6 结论 ······ 182
参考文献 ······ 182
第 6 章 内存天下 ······ 184
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

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
7.1 把 new 说透 215
7.1.1 引言 215
7.1.2 基本概念 215
7.1.3 深入浅出 217
7.1.4 结论 219
7.2 base 和 this 219
7.2.1 引言 219
7.2.2 基本概念 219
7.2.3 深入浅出 220
7.2.4 通用规则 224
7.2.5 结论 224
7.3 using 的多重身份 224
7.3.1 引言 224
7.3.2 引入命名空间 225
7.3.3 创建别名 225
7.3.4 强制资源清理 227
7.3.5 结论 230
7.4 认识全面的 null 230
7.4.1 引言 230
7.4.2 从什么是 null 开始 230
7.4.3 Nullable<T>（可空类型） 232
7.4.4 ??运算符 234
7.4.5 Nulll Object 模式 235
7.4.6 结论 238
7.5 转换关键字 238
7.5.1 引言 239
7.5.2 自定义类型转换探讨 239
7.5.3 本质分析 240
7.5.4 结论 242
7.6 预处理指令关键字 242
7.6.1 引言 242
7.6.2 预处理指令简述 242
7.6.3 #if、#else、#elif、#endif 243
7.6.4 #define、#undef 244
7.6.5 #warning、#error 244
7.6.6 #line 245
7.6.7 结论 245
7.7 非主流关键字 245
7.7.1 引言 245
7.7.2 checked/unchecked 246
7.7.3 yield 247
7.7.4 lock 250
7.7.5 unsafe 252
7.7.6 sealed 253
7.7.7 结论 254
参考文献 254
第 8 章 巅峰对决——走出误区 255
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
第 9 章 本来面目——框架诠释 ············318
9.1 万物归宗：System.Object ············319
9.1.1 引言 ············319
9.1.2 初识 ············319
9.1.3 分解 ············320
9.1.4 插曲：消失的成员 ············323
9.1.5 意义 ············325
9.1.6 结论 ············325
9.2 规则而定：对象判等 ············325
9.2.1 引言 ············326
9.2.2 本质分析 ············326
9.2.3 覆写 Equals 方法 ············329
9.2.4 与 GetHashCode 方法同步 ············331
9.2.5 规则 ············332
9.2.6 结论 ············332
9.3 疑而不惑：interface“继承”争议 ············332
9.3.1 引言 ············332
9.3.2 从面向对象寻找答案 ············333
9.3.3 以 IL 探求究竟 ············334
9.3.4 System.Object 真是“万物之宗”吗 ············334
9.3.5 接口的继承争议 ············335

9.3.6 结论 335
9.4 给力细节：深入类型构造器 336
9.4.1 引言：一个故事 336
9.4.2 认识对象构造器和类型构造器 337
9.4.3 深入执行过程 339
9.4.4 回归故事 341
9.4.5 结论 342
9.5 如此特殊：大话 String 342
9.5.1 引言 342
9.5.2 问题迷局 343
9.5.3 什么是 string 345
9.5.4 字符串创建 345
9.5.5 字符串恒定性 346
9.5.6 字符串驻留（String Interning） 346
9.5.7 字符串操作典籍 350
9.5.8 补充的礼物：StringBuilder 352
9.5.9 结论 354
9.6 简易不简单：认识枚举 354
9.6.1 引言 355
9.6.2 枚举类型解析 355
9.6.3 枚举种种 358
9.6.4 位枚举 360
9.6.5 规则与意义 361
9.6.6 结论 361
9.7 一脉相承：委托、匿名方法和 Lambda 表达式 362
9.7.1 引言 362
9.7.2 解密委托 362
9.7.3 委托和事件 365
9.7.4 匿名方法 367
9.7.5 Lambda 表达式 368
9.7.6 规则 368
9.7.7 结论 369
9.8 Name 这回事儿 369
9.8.1 引言 369
9.8.2 畅聊 Name 369
9.8.3 回到问题 371
9.8.4 结论 371
9.9 直面异常 371
9.9.1 引言 372
9.9.2 为何而抛 372
9.9.3 从 try/catch/finally 说起：解析异常机制 375
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
第 10 章 格局之选——命名空间剖析 383
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

第 4 部分　拾遗——.NET也有春天

第 11 章　接触泛型……402

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

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
14.1.1 引言……………473
14.1.2 历史脚步……………473
14.1.3 未来之变……………477
14.1.4 结论……………479
14.2 .NET 4.0，第一眼……………480
14.2.1 引言……………480
14.2.2 第一眼……………481
14.2.3 结论……………484
14.3 动态变革：dynamic……………484
14.3.1 引言……………484
14.3.2 初探……………485
14.3.3 本质：DLR……………485
14.3.4 PK 解惑……………488
14.3.5 应用：动态编程……………490
14.3.6 结论……………491
14.4 趋势必行，并行计算……………491
14.4.1 引言……………491
14.4.2 拥抱并行……………492
14.4.3 TPL……………493
14.4.4 PLINQ……………495
14.4.5 并行补遗……………496
14.4.6 结论……………497
14.5 命名参数和可选参数……………497
14.5.1 引言……………497
14.5.2 一览究竟……………498
14.5.3 简单应用……………499
14.5.4 结论……………499
14.6 协变与逆变……………500
14.6.1 引言……………500
14.6.2 概念解析……………500
14.6.3 深入……………502
14.6.4 结论……………504
14.7 Lazy<T>点滴……………504
14.7.1 引言……………505

14.7.2 延迟加载 ······ 505
14.7.3 Lazy<T>登场 ······ 505
14.7.4 Lazy<T>本质 ······ 507
14.7.5 结论 ······ 509
14.8 Tuple 一二 ······ 509
14.8.1 引言 ······ 509
14.8.2 Tuple 为何物 ······ 510
14.8.3 Tuple Inside ······ 511
14.8.4 优略之间 ······ 513
14.8.5 结论 ······ 514
参考文献 ······ 514
后记：我写的不是代码 ······ 516
编后记：遇见幸福 ······ 521

第 3 章　OO 之美

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

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，而在这个旅程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在实际的项目系统中。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，首先要做好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。

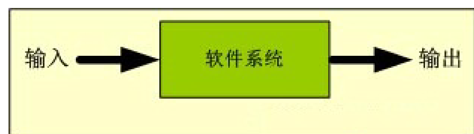


图 3-1　软件的输入和输出

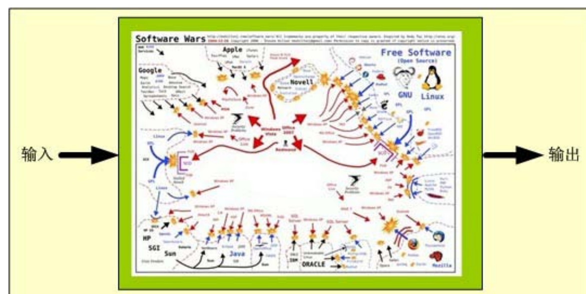


图 3-2　复杂系统的输入和输出

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）。

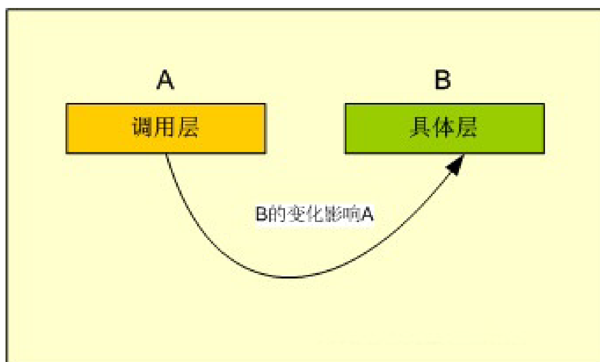


图 3-3　依赖的关系

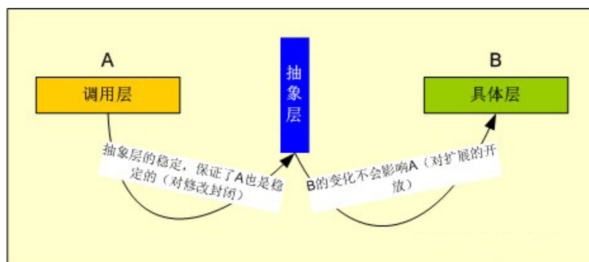


图 3-4　依赖的关系（引入抽象层）

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，也自然而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。

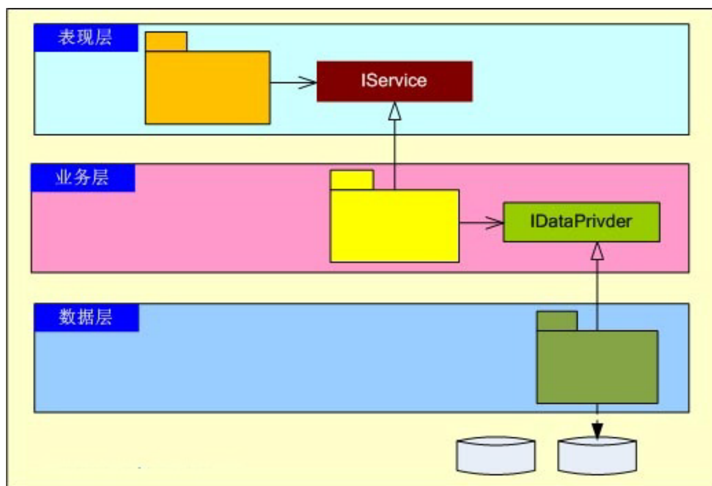


图 3-5　多层架构的依赖

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，总之凡是可以互联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。

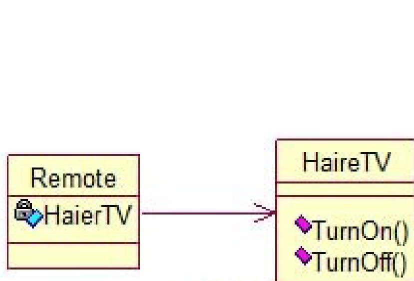


图 3-6　遥控器初次设计

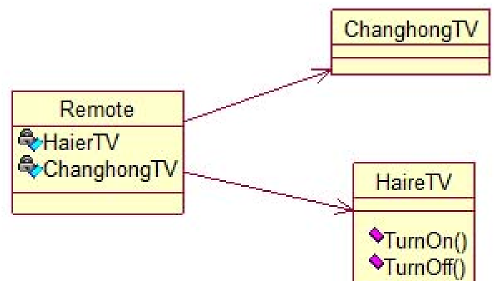


图 3-7　遥控器二次设计

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）：

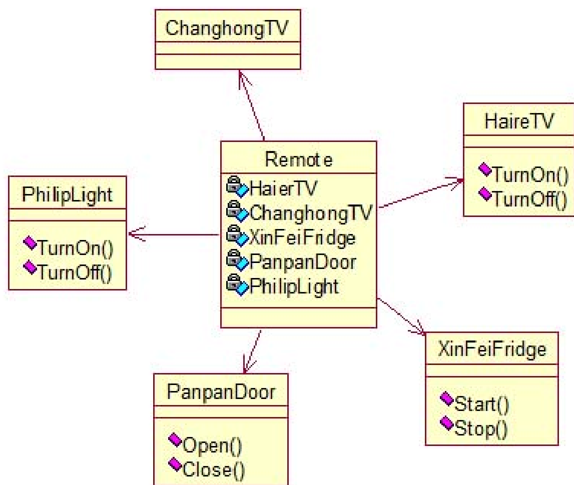


图 3-8　遥控器三次设计

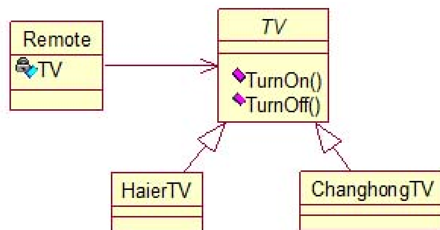


图 3-9　重构的遥控器设计

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。

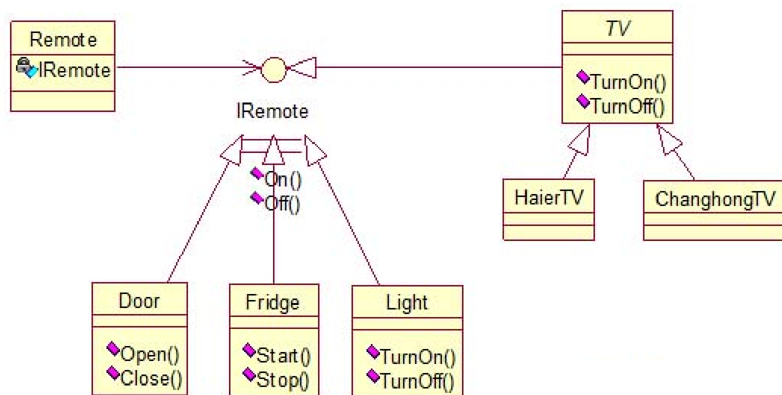


图 3-10　重构的遥控器设计

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。

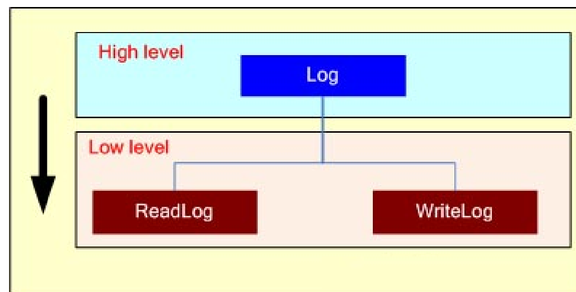


图 3-11　高层依赖于底层

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。

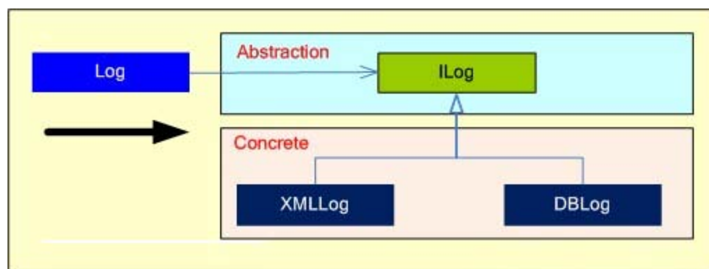


图 3-12　高层依赖于抽象

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(() =>
            {
                 // exceute logic in this provider, if provider is DefualtRunnerProvider,
                 // 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 Woker
        {
            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 AcceptVerbsAttrib
ute(HttpVerbs.Post);

        // Methods
        public override bool IsValidForRequest(ControllerContext controllerContext, MethodIn
fo 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	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。 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	桥接、适配器、组合、外观、装饰、享元、代理
行为型模式	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。 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	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。

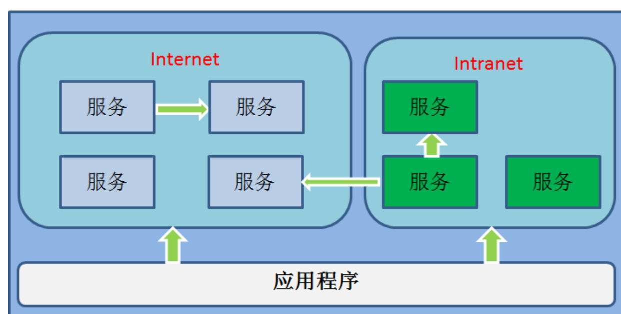


图 3-14　高层依赖于抽象

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的状态。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
hannel>();
            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 TAn
imal : 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
	享元
	代理
行为型模式	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.CompilerGenerated
Attribute::.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 模式的应用体现。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, C
acheItem<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，然后接着爬向下一座，并且继续如此反复，才能到最高的巅峰。

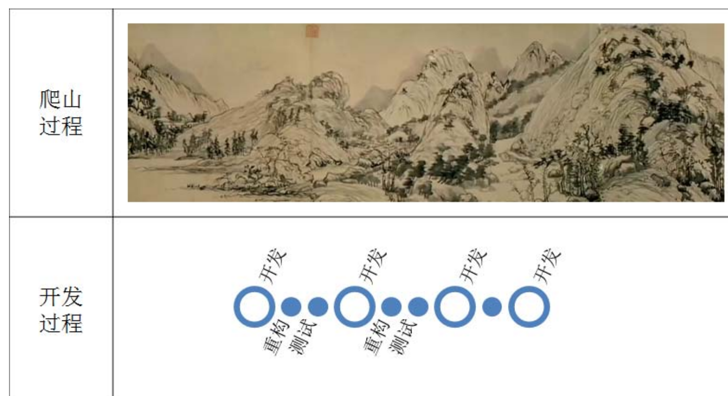


图 3-15　软件开发的爬山模型（图片来源：百度百科，上图为富春山居图部分截图）

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。

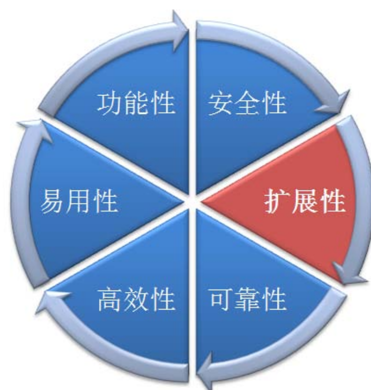


图 3-16　架构的考量

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.D
evkit.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

Juditb 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