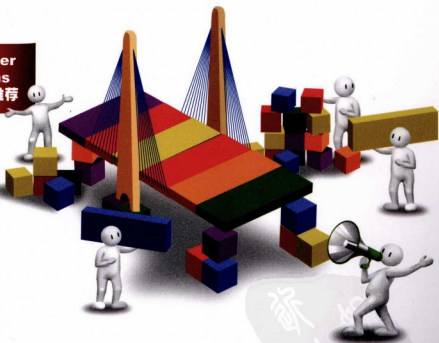


Applying Domain-Driven Design and Patterns

# 领域驱动设计与模式实战

[瑞典] Jimmy Nilsson 著  
赵俐 马燕新 等译

Martin Fowler  
和Eric Evans  
两位大师联袂推荐



- .NET开发人员必读之作
- 《企业应用架构模式》与《领域驱动设计》两大名著精髓的实战演练
- 教你穿越业务层、数据层和UI层之间重重障碍，打通任督二脉

TURING 图灵程序设计丛书

Applying Domain-Driven Design and Patterns  
**领域驱动设计与模式实战**

[瑞典] Jimmy Nilsson 著  
赵俐 马燕新 等译

人民邮电出版社  
北 京





## 图书在版编目(CIP)数据

领域驱动设计与模式实战 / (瑞典) 尼尔森 (Nilsson, J.) 著; 赵俐等译. —北京: 人民邮电出版社, 2009.11  
(图灵程序设计丛书)  
ISBN 978-7-115-21277-1

I. 领… II. ①尼…②赵… III. 软件设计 IV. TP311.5

中国版本图书馆CIP数据核字(2009)第149514号

## 内 容 提 要

本书全面详细地解释了领域驱动设计、测试驱动开发、依赖注入、持久化、重构、模式等很多基本概念, 并以 C# 和 .NET 实例为依托, 展示了这些概念的实际应用和重要价值。更重要的是, 本书还将这些概念整合到一起, 为开发人员从头至尾地揭示了完整的开发路线。阅读本书后, 读者将能真正掌握这些重要概念, 并有效地将它们结合起来, 应用到实际开发过程中。

本书适合软件架构师和开发人员阅读。

图灵程序设计丛书

## 领域驱动设计与模式实战

- ◆ 著 [瑞典] Jimmy Nilsson
- 译 赵 俐 马燕新 等
- 责任编辑 傅志红
- 执行编辑 陈兴璐
- ◆ 人民邮电出版社出版发行 北京市崇文区夕照寺街14号
- 邮编 100061 电子函件 315@ptpress.com.cn
- 网址 <http://www.ptpress.com.cn>
- 北京顺义振华印刷厂印刷
- ◆ 开本: 800×1000 1/16
- 印张: 23.75
- 字数: 561 千字
- 印数: 1—3 000 册
- 2009年11月第1版
- 2009年11月北京第1次印刷
- 著作权合同登记号 图字: 01-2007-1481号
- ISBN 978-7-115-21277-1

定价: 69.00元

读者服务热线: (010)51095186 印装质量热线: (010)67129223

反盗版热线: (010)67171154

# 目 录

## 第一部分 背景知识

### 第1章 应重视的价值,也是对过去几年的沉重反思

1.1 总体价值	2
1.2 应重视的架构风格	3
1.2.1 焦点之一:模型	3
1.2.2 焦点之二:用例	3
1.2.3 如果重视模型,就可以使用领域模型模式	6
1.2.4 慎重处理数据库	9
1.2.5 领域模型与关系数据库之间的阻抗失配	13
1.2.6 谨慎处理分布式	16
1.2.7 消息传递很重要	18
1.3 对过程的各个组成部分的评价	19
1.3.1 预先架构设计	20
1.3.2 领域驱动设计	21
1.3.3 测试驱动开发	22
1.3.4 重构	25
1.3.5 选择一种还是选择组合	26
1.4 持续集成	27
1.4.1 解决方案(或至少是正确方向上的一大步)	27
1.4.2 从我的组织汲取的教训	28
1.4.3 更多信息	28
1.5 不要忘记运行机制	28
1.5.1 有关何时需要运行机制的一个例子	29
1.5.2 运行机制的一些例子	29
1.5.3 它不仅仅是我们的过错	30

1.6 小结	30
第2章 模式起步	32
2.1 模式概述	32
2.1.1 为什么要学习模式	33
2.1.2 在模式方面要注意哪些事情	34
2.2 设计模式	35
2.3 架构模式	42
2.3.1 示例:层	42
2.3.2 另一个示例:领域模型模式	43
2.4 针对具体应用程序类型的设计模式	43
2.5 领域模式	48
2.6 小结	52
第3章 TDD与重构	53
3.1 TDD	53
3.1.1 TDD流程	53
3.1.2 演示	54
3.1.3 设计效果	59
3.1.4 问题	61
3.1.5 下一个阶段	62
3.2 模拟和桩	62
3.2.1 典型单元测试	62
3.2.2 声明独立性	63
3.2.3 处理困难因素	64
3.2.4 用测试桩替换协作对象	64
3.2.5 用模拟对象替换协作对象	66
3.2.6 设计含义	68
3.2.7 结论	68
3.2.8 更多信息	68
3.3 重构	68
3.4 小结	78

## 第二部分 应用DDD

## 第4章 新的默认架构 ..... 80

## 4.1 新的默认架构的基础知识 ..... 80

## 4.1.1 从以数据库为中心过渡到以领域模型为中心 ..... 81

## 4.1.2 进一步关注DDD ..... 81

## 4.1.3 根据DDD进行分层 ..... 82

## 4.2 轮廓 ..... 83

## 4.2.1 领域模型示例的问题/特性 ..... 83

## 4.2.2 逐个处理特性 ..... 84

## 4.2.3 到目前为止的领域模型 ..... 94

## 4.3 初次尝试将UI与领域模型挂接 ..... 95

## 4.3.1 基本目标 ..... 95

## 4.3.2 简单UI的当前焦点 ..... 95

## 4.3.3 为客户列出订单 ..... 95

## 4.3.4 添加订单 ..... 97

## 4.3.5 刚才我们看到了什么 ..... 98

## 4.4 另一个维度 ..... 98

## 4.4.1 领域模型的位置 ..... 99

## 4.4.2 孤立或共享的实例 ..... 99

## 4.4.3 有状态或无状态领域模型实例化 ..... 100

## 4.4.4 领域模型的完整实例化或子集实例化 ..... 100

## 4.5 小结 ..... 101

## 第5章 领域驱动设计进阶 ..... 102

## 5.1 通过简单的TDD实验来精化领域模型 ..... 102

## 5.1.1 从Order和OrderFactory的创建开始 ..... 103

## 5.1.2 一些领域逻辑 ..... 106

## 5.1.3 第二个任务: OrderRepository + OrderNumber ..... 107

## 5.1.4 重建持久化的实体: 如何从外部设置值 ..... 111

## 5.1.5 获取订单列表 ..... 114

## 5.1.6 该到讨论实体的时候了 ..... 115

## 5.1.7 再次回到流程上来 ..... 116

## 5.1.8 总览图 ..... 117

## 5.1.9 建立OrderRepository的伪实现 ..... 118

## 5.1.10 简单讨论一下保存 ..... 120

## 5.1.11 每个订单的总量 ..... 120

## 5.1.12 历史客户信息 ..... 124

## 5.1.13 实例的生命周期 ..... 126

## 5.1.14 订单类型 ..... 128

## 5.1.15 订单的介绍人 ..... 128

## 5.2 连贯接口 ..... 130

## 5.3 小结 ..... 131

## 第6章 准备基础架构 ..... 132

## 6.1 将POCO作为工作方式 ..... 133

## 6.1.1 实体和值对象的PI ..... 133

## 6.1.2 是否使用PI ..... 137

## 6.1.3 运行时与编译时PI ..... 137

## 6.1.4 PI实体/值对象的代价 ..... 137

## 6.1.5 将PI用于存储库 ..... 139

## 6.1.6 单组存储库的代价 ..... 143

## 6.2 对保存场景的处理 ..... 143

## 6.3 建立伪版本机制 ..... 147

## 6.3.1 伪版本机制的更多特性 ..... 148

## 6.3.2 伪版本的实现 ..... 149

## 6.3.3 影响单元测试 ..... 150

## 6.4 数据库测试 ..... 153

## 6.4.1 在每次测试之前重置数据库 ..... 154

## 6.4.2 在测试运行期间保持数据库的状态 ..... 155

## 6.4.3 测试之前重置测试所使用的数据库 ..... 156

## 6.4.4 不要忘记不断演变的模式 ..... 156

## 6.4.5 分离单元测试和数据库调用测试 ..... 156

## 6.5 查询 ..... 159

## 6.5.1 单组查询对象 ..... 160

## 6.5.2 单组查询对象的代价 ..... 161

## 6.5.3 将查询定位到哪里 ..... 162

## 6.5.4 再次将聚合作为工具 ..... 163

## 6.5.5 将规格用于查询 ..... 164

## 6.5.6 其他查询选择 ..... 165

6.6 小结	165	7.7 对实现进行精化	184
第7章 应用规则	166	7.7.1 一个初步实现	184
7.1 规则的分类	166	7.7.2 创建规则类, 离开最不成熟的阶段	189
7.2 规则的原则及用法	167	7.7.3 设置规则列表	191
7.2.1 双向规则检查: 可选的(可能的)主动检查, 必需的(和自动的)被动检查	167	7.7.4 使用规则列表	192
7.2.2 所有状态(即使是错误状态)都应该是可保存的	167	7.7.5 处理子列表	192
7.2.3 规则应该高效使用	167	7.7.6 一个API改进	193
7.2.4 规则应该是可配置的, 以便添加自定义规则	167	7.7.7 自定义	194
7.2.5 规则应与状态放在一起	167	7.7.8 为使用者提供元数据	195
7.2.6 规则应该具有很高的可测试性	168	7.7.9 是否适合用模式来解决此问题	195
7.2.7 系统应阻止我们进入错的状态	168	7.7.10 复杂规则又是什么情况	195
7.3 开始创建API	168	7.8 绑定到持久化抽象	196
7.3.1 上下文, 上下文, 还是上下文	169	7.8.1 使验证接口成为可插入的	196
7.3.2 数据库约束	169	7.8.2 在保存方面实现被动验证的替代解决方案	197
7.3.3 将规则绑定到与领域有关的转换, 还是绑定到与基础架构有关的转换	170	7.8.3 重用映射元数据	198
7.3.4 精化原则: 所有状态, 即使是错误状态, 都应该是可保存的	171	7.9 使用泛型和匿名方法	198
7.4 与持久化有关的基本的规则API的需求	172	7.10 其他人都做了什么	199
7.4.1 回到已发现的API问题上	173	7.11 小结	200
7.4.2 问题是什么	173		
7.4.3 我们允许了不正确的转换	174		
7.4.4 如果忘记检查怎么办	174		
7.5 关注与领域有关的规则	174		
7.5.1 需要合作的规则	176		
7.5.2 使用基于集合的处理方法	177		
7.5.3 基于服务的验证	178		
7.5.4 在不应该转换时尝试转换	179		
7.5.5 业务ID	180		
7.5.6 避免问题	182		
7.5.7 再次将聚合作为工具	183		
7.6 扩展API	183		
7.6.1 查询用于设置UI的规则	184		
7.6.2 使注入规则成为可能	184		
		第三部分 应用 PoEAA	
		第8章 用于持久化的基础架构	202
		8.1 持久化基础架构的需求	203
		8.2 将数据存储到哪里	204
		8.2.1 RAM	204
		8.2.2 文件系统	205
		8.2.3 对象数据库	206
		8.2.4 关系数据库	207
		8.2.5 使用一个还是多个资源管理器	207
		8.2.6 其他因素	207
		8.2.7 选择和前进	207
		8.3 方法	208
		8.3.1 自定义手工编码	208
		8.3.2 自定义代码的代码生成	209
		8.3.3 元数据映射(对象关系(O/R)映射工具)	210
		8.3.4 再次选择	211
		8.4 分类	211

8.4.1 领域模型风格	211	9.5.1 元数据映射: 元数据类型	240
8.4.2 映射工具风格	212	9.5.2 标识字段	240
8.4.3 起点	212	9.5.3 外键映射	241
8.4.4 API焦点	213	9.5.4 嵌入值	241
8.4.5 查询风格	213	9.5.5 继承解决方案	242
8.4.6 高级数据库支持	214	9.5.6 标识映射	243
8.4.7 其他功能	215	9.5.7 操作单元	244
8.5 另一个分类: 基础架构模式	216	9.5.8 延迟加载/立即加载	244
8.5.1 元数据映射: 元数据的类型	216	9.5.9 并发性控制	244
8.5.2 标识字段	217	9.5.10 额外功能: 验证挂钩	245
8.5.3 外键映射	219	9.6 NHibernate和DDD	245
8.5.4 嵌入值	219	9.6.1 程序集概览	245
8.5.5 继承解决方案	219	9.6.2 ISession和存储库	246
8.5.6 标识映射	220	9.6.3 ISession、存储库和事务	246
8.5.7 操作单元	220	9.6.4 得到了什么结果	247
8.5.8 延迟加载/立即加载	220	9.7 小结	247
8.5.9 并发控制	221		
8.6 小结	222		
第9章 应用NHibernate	223		
9.1 为什么使用NHibernate	223		
9.2 NHibernate简介	224		
9.2.1 准备	224		
9.2.2 一些映射元数据	225		
9.2.3 一个小的API示例	229		
9.2.4 事务	231		
9.3 持久化基础架构的需求	232		
9.3.1 高级持久化透明	232		
9.3.2 持久化实体的生命周期所需的特定特性	233		
9.3.3 谨慎处理关系数据库	234		
9.4 分类	235		
9.4.1 领域模型风格	235		
9.4.2 映射工具风格	235		
9.4.3 起点	236		
9.4.4 API焦点	236		
9.4.5 查询语言风格	236		
9.4.6 高级数据库支持	237		
9.4.7 其他功能	239		
9.5 另一种分类: 基础架构模式	240		
		第四部分 下一步骤	
		第10章 博采其他设计技术	250
		10.1 上下文为王	250
		10.1.1 层和分区	250
		10.1.2 分区的原因	251
		10.1.3 限界上下文	252
		10.1.4 限界上下文与分区有何关联	252
		10.1.5 向上扩展DDD项目	252
		10.1.6 为什么对领域模型——SO分区	253
		10.2 SOA简介	253
		10.2.1 什么是SOA	253
		10.2.2 为什么需要SOA	253
		10.2.3 SOA有什么不同	254
		10.2.4 什么是服务	254
		10.2.5 服务中包括什么	254
		10.2.6 深入分析4条原则	255
		10.2.7 再来看一下什么是服务	256
		10.2.8 OO在SOA中的定位	256
		10.2.9 客户-服务器和SOA	257
		10.2.10 单向异步消息传递	257
		10.2.11 SOA如何提高可伸缩性	258

10.2.12	SOA服务的设计	258	11.2	模型-视图-控制器模式	293
10.2.13	服务之间如何交互	259	11.2.1	示例: Joe的Shoe Shop程序	294
10.2.14	SOA和不可用的服务	261	11.2.2	通过适配器简化视图界面	299
10.2.15	复杂的消息传递处理	262	11.2.3	将控制器从视图解耦	299
10.2.16	服务的可伸缩性	262	11.2.4	将视图和控制器结合起来	300
10.2.17	小结	263	11.2.5	是否值得使用MVC	300
10.3	控制反转和依赖注入	263	11.3	测试驱动的Web窗体	300
10.3.1	任何对象都不是孤岛	263	11.3.1	背景	301
10.3.2	工厂、注册类和服务定位器	265	11.3.2	一个示例	301
10.3.3	构造方法依赖注入	267	11.3.3	领域模型	302
10.3.4	setter依赖注入	269	11.3.4	GUI的TDD	302
10.3.5	控制反转	270	11.3.5	Web窗体实现	307
10.3.6	使用了Spring.NET框架的依赖注入	271	11.3.6	小结	309
10.3.7	利用PicoContainer.NET进行自动装配	272	11.3.7	用NMock创建模拟	309
10.3.8	嵌套容器	273	11.4	映射和包装	311
10.3.9	服务定位器与依赖注入的比较	275	11.4.1	映射和包装	311
10.3.10	小结	275	11.4.2	用表示模型来包装领域模型	312
10.4	面向方面编程	276	11.4.3	将表示模型映射到领域模型	313
10.4.1	热门话题有哪些	277	11.4.4	管理关系	316
10.4.2	AOP术语定义	279	11.4.5	状态问题	318
10.4.3	.NET中的AOP	280	11.4.6	最后的想法	319
10.4.4	小结	291	11.5	小结	320
10.5	小结	291	11.6	结束语	320
第 11 章	关注 UI	292	第五部分	附 录	
11.1	撮前结语	292	附录 A	其他领域模型风格	324
			附录 B	已讨论的模式的目录	349

## 第五部分 附 录

附录 A 其他领域模型风格 ..... 324

附录 B 已讨论的模式的目录 ..... 349

# 序

构建企业应用程序并非易事。尽管我们拥有大量工具和框架来简化这项任务，但仍然需要弄清楚如何更好地使用这些工具。大量方法可供我们选用，但关键是要知道在特定情况下应使用哪种方法，因为一种方法很难适用于所有情况。在过去几年中，有一个社区逐渐成长起来，人们不断寻找用于设计企业应用的方法，并以模式的形式将它们记录下来（我整理了一份概述，在<http://martinfowler.com/articles/enterprisePatterns.html>站点上）。参与这项工作的人们（例如我）试图找到公共方法，并描述如何更好地使用它们，以及它们何时适用。最后的结果过于宽泛，会导致为读者提供了过多的选择。

当我开始创作 *Patterns of Enterprise Application Architecture*（《企业应用架构模式》，Addison-Wesley公司2002出版）时，我曾在微软技术方面寻找过这种类型的设计建议。几经努力之后几乎一无所获，只找到了一本讨论此领域的书，就是Jimmy的前一本书。我喜欢他平易近人的写作风格，也喜欢他深入挖掘易被他人忽略的概念的热情。因此，Jimmy决定从我和企业模式社区中的其他人借鉴一些思想，并向读者展示在编写.NET应用程序时如何应用这些思想，我觉得再合适不过了。

这个企业模式社区的主旨是记录好的设计，但另一个思路也贯穿整个社区。我们也是敏捷方法的忠实爱好者，信奉诸如测试驱动开发（TDD）和重构这样的技术。因此，Jimmy也把这些思想带到这本书中。很多人认为模式与TDD是不一致的，因为模式的重点是设计，而TDD的重点是演进。但有很多人正在将这两种技术结合起来使用，这证明这种观点是错误的，Jimmy将这些思路也整合到本书中。

结果就诞生了这本有关在.NET环境中进行设计的书。它受到敏捷方式的推动，又倾注了企业模式社区的成果。它是一本向读者展示了如何将TDD、对象-关系映射和领域驱动设计等方法应用于.NET项目的书。如果读者以前未遇到过这些概念，那么会发现它是一本入门书，书中介绍的技术在很多开发人员看来是未来软件开发的关键。如果你熟悉这些思想，那么本书将帮助你将这些思想传递给你的同事。

很多人感觉微软技术社区在传播好的企业应用设计建议方面不如其他社区。随着技术越来越强大，复杂度越来越高，理解如何更好地使用技术就变得越来越重要。本书在推进这种理解方面迈出了可贵的一步。

Martin Fowler

<http://martinfowler.com>

## 序 二

学习领域驱动设计(DDD)的最好方法是坐在一位友好、耐心且经验丰富的从业者身边,一步一步地共同研究问题。阅读本书正是这种体验。

本书并不是推出一个新的宏大方案,而是自然地报告了一位专家从业者是怎样使用及组合那些吸引他的当前实践的。

Jimmy Nilsson重申了我们很多人都曾经说过的话:几个时下流行的主题,特别是领域驱动设计(DDD)、企业应用架构模式(PoEAA)和测试驱动开发(TDD),它们并不是彼此互斥的替代品,而是成功开发中互相促进的元素。

此外,这三种技术实际上比它们乍看起来要难得多。它们需要各个领域的渊博知识。本书在倡导这些方法上花费了一定的时间,但主要还是关注如何让它们发挥作用的细节。

有效的设计不仅仅是通过死记硬背来学会的一组技术,它更是一个思考的过程。当Jimmy深入剖析一个示例时,他为我们打开了了解他思维的一扇窗子。他不仅展示和解释了他的解决方案,而且让我们看到他是如何得到解决方案的。

当我设计某个系统时,有数十种想法会在脑海中闪现。如果它们是我经常处理的因素,那么会很快地闪过,我几乎意识不到。如果它们属于我不太确信的领域,那么我会更多地思考它们。我认为这是设计者遇到的典型现象,但很难将它传达给另一个人。当Jimmy仔细讨论他的示例时,就好像他把这个过程放了慢动作一样。在每个小的转折点,都会呈现3~4种选择,Jimmy对它们进行权衡和扬弃,最终选择最有利的一个。

例如,我们希望在不涉及持久化技术的情况下实现模型对象。那么,持久化框架会强制性“污染”领域对象实现的8种(是8种!)方式都是什么?是什么因素导致我们在其中的一些要点上做出折中?当前流行的框架(包括.NET)都强加了什么约束?

Jimmy的思考注重实效。他利用他的经验做出一种可能实现最终目标,并且遵守更深层次设计原则的设计选择,而不是看起来最符合教科书示例的选择。而且他的所有决定都是临时的。

Jimmy奉行的第一条设计原则就是DDD的基本目标:一个反映了对业务问题深刻理解的设计,而且在设计形式上,要使设计能够根据新方法做出调整。那么为什么还要讨论这么多技术框架和架构呢?

有一种常见(也很自然)的误解是:这种强调领域问题的优先考虑不需要太多技术才能和技巧。如果这是真的那就好了,要想成为一名胜任的领域设计师并不十分困难。可是,要想在软件中呈现清晰且有用的领域概念,以便它们不被大量的技术细节所淹没,就要求能娴熟地运用技术。据我观察,那些熟练掌握技术和架构原则的人通常知道如何把技术放在其适合的位置上,而且这



些人也是最高效的领域建模人员。

我并不是说应该面面俱到地掌握复杂工具的所有知识，而是强调应该掌握Martin Fowler的PoEAA当中所展示的那类知识，因为盲目应用技术反而会导致对应用产生干扰。

对很多人来说，本书将填补一个空白，教会大家怎样在实践中实现富有表现力的对象模型。我从本书学到不少思考技术框架应用的有用方法，尤其是加强了我对在.NET平台上使用DDD方法的一些理解。

除了技术架构以外，Jimmy还在如何编写测试上花费了大量时间。TDD以一种不同方式补充了DDD。当对于精化更有用模型的重视程度不够时，TDD容易导致零散的应用，这时对某一特性的一味追求会使系统最终无法扩展。综合的测试套件实际上使开发团队能够继续取得进展（与没有TDD时相比），但这只是TDD的最基本价值。

当测试套件充分发挥作用时，它是域模型的实验室，也是通用语言的技术表示。特殊风格的测试推动建模过程向前发展，并保持整个建模过程始终围绕重点展开。开发这些测试的示例贯穿了本书。

Jimmy Nilsson集自信和谦逊于一身，我注意到这是最优秀设计人员的特征。当他说到以前习惯于相信什么，后来为什么观点又改变的时候，我们可以管窥到他是如何获得当前理解的，这帮助读者穿越技术细节，从而理解基本原则。这种谦虚的品质帮助他集思广益，并因此为我们奉上了这本融合各种不同思想的书。他做了各种各样的尝试，并用结果和经验作为自己的指导。他的结论不是以被揭示的真理的面目出现，而是作为他到目前为止的最佳理解，其中的美妙令我们回味无穷。所有这些都使得他的建议对读者更有用。而且这种态度本身也是领导成功软件开发所需的一个重要元素。

Eric Evans



# 前言：跨越鸿沟

本书的封面图片<sup>①</sup>是连接瑞典和丹麦的厄勒海峡大桥。似乎所有软件架构图书的封面都要有一座大桥，但本书选用大桥作为封面却有一些其他原因。

就是这座大桥取代了我小时候乘坐过很多次的渡船。驱车穿越大桥是一种绝佳的体验，虽然我已经不下数十次穿越它，但还是乐此不疲。

题外话，我父亲曾经就在负责架设这座大桥最高部分的施工队中工作。

但这些并不是最主要的原因，主要原因是本书内容正是跨越鸿沟，即跨越用户与开发人员之间的鸿沟，跨越业务与软件之间的鸿沟，跨越逻辑与存储之间的鸿沟，跨越DB人员与OO人员之间的鸿沟……

我并不是拿桥接模式[GoF Design Patterns]来信口开河，这毕竟是前言啊！

## 本书的主旨

本书的主旨是如何将领域模型构建得整洁，且仍具有便于持久化的特性。书中展示了在这样一个领域模型中，持久化解决方案将以什么面貌出现，特别是如何在领域模型与数据库这道鸿沟上架起一座桥梁。

换言之，我的愿望是在本书中将Eric Evans的《领域驱动设计》[Evans DDD]和Martin Fowler的《企业应用架构模式》[Fowler PoEAA]融为一体。

人们可能认为DDD有些抽象。因此，具体示例有助于对本书的理解，例如对持久化概念的理解。这些示例可能比较基础，但它们可作为良好的开端。本书不仅解释了如何使用模式，而且阐述了如何在O/R映射工具中使用模式。

我非常清楚在架构方面“一种规格并非处处适用”。尽管如此，事实证明模式还是具有足够的通用性的，我们可以在各种各样的情况下使用和重用它们。

本书的重点并不是讲模式本身，而是在各个章节中穿插使用它们，将其作为工具和语言，用于讨论不同的设计方面。在这个过程中，不熟悉模式的读者可以获得模式的一些知识并对模式产生兴趣。

对于TDD也是如此。但是，并非所有开发人员都会对此感兴趣，在.NET社区中尤其如此，因为TDD（仅作为模式）被认为是一种应用范围很狭窄的技术，甚至完全是一种默默无闻的技术。本书将教会读者应用TDD。

<sup>①</sup> 指本书的英文原版封面图片。——编者注

## 本书的目的

在不耽搁其他日常项目和工作的前提下撰写我的第一本书[Nilsson NED]，确实让我体会到了什么是艰苦。我当时已下定决心不再写书了。但时间还是改变了这个决定，因为我有话要说，而且不能不说了。

想法的改变源于最近阅读的两本书给我的灵感和触动。第一本书是Martin Fowler的《企业应用架构模式》[Fowler PoEAA]。这本书激发了我再次尝试领域模型模式的兴趣，尽管此前曾有过几次失败经历。

然后我读到了Eric Evans的《领域驱动设计》[Evans DDD]。这本书使我深入理解了如何思考和进行以领域为核心的开发，以及如何在这样的开发中使用特定的领域模型模式风格。

另一个重要影响就是在我多年的模式课程教学过程中所积累的知识。通过与学生们的交流以及教学内容的发展，我自己也有所感悟。

我参与一个宏大的（遗憾的是尚未完成）开源项目Valhalla上工作时，对DDD的认识也发生了转变，这个项目是我和Christoffer Skjoldborg协作开发的。（到目前为止，大部分工作都是由Christoffer完成的。）

总之，我觉得需要写一本应用多于理论的书，但这本书又要有坚实的基础，就像DDD和PoEAA那样。“应用”更贴近我的心声，因为我始终认为自己归根到底是一名开发人员。

## 本书的读者

本书面向广泛的读者。具备以下知识有助于阅读本书。

- 面向对象
- .NET或类似平台
- C#或类似语言
- 关系数据库，例如SQL Server

但是，兴趣和热情可以弥补任何经验的不足。

这里再详细解释一下为什么目标读者是广泛的。首先，我们从平台框架来考虑一下适用的人员。本书面向那些需要一种更关注核心的方法而不是简单的拖放操作（可能这个概括并不是很恰当）的.NET开发人员。Java开发人员可以从如何结合DDD和O/R映射的讨论及示例中获益。

我认为选定的语言和平台越来越不重要了，因此谈论.NET开发人员和Java开发人员看起来有些不合时宜。那么就让我们试着用另一种角度来描述目标读者，即，本书是面向开发人员、团队领导者和架构师的。

选择这种角度后，我认为本书中有些内容既适合中级读者，也适合高级读者。可能也会有些内容适合初级读者。

## 本书的组织

本书共分四部分：背景知识、应用DDD、应用PoEAA和下一步骤。

## 第一部分：背景知识

背景知识这一部分概括地讨论架构和过程，重点讲解领域模型和DDD（领域驱动设计）[Evans DDD]，还将介绍模式和TDD（测试驱动开发）。第一部分包括以下几章。

- 第1章，应重视的价值

该章讨论要想得到高质量的系统开发结果，应重视的架构属性和过程。这一章的讨论受到极限编程（XP）的影响。

- 第2章，模式起步

该章主要提供示例，并讨论不同类型的模式，例如设计模式、架构模式和领域模式。

- 第3章，TDD与重构

第1章讨论了很多TDD和重构内容，而这一章则通过较长的示例并着眼于不同的TDD风格来深入阐述这些内容。

## 第二部分：应用 DDD

这一部分介绍DDD的应用。此外还会为基础架构准备领域模型，重点关注规则方面。

- 第4章，新的默认架构

该章列出了示例应用程序的各方面需求，并初步创建了一个模型作为后续章节的起点。这里使用了基于领域模型的架构。

- 第5章，领域驱动设计进阶

这一章使用第4章中提出的需求作为开始构建DDD风格的领域模型的基础。在这一章中，由于是刚开始使用TDD方法，因此我们放慢构建速度。

- 第6章，准备基础架构

尽管我们尽量推迟基础架构方面，但还是有必要稍微提前思考一下，并针对基础架构需求为领域模型做一些准备。这一章讨论持久化透明（PI）领域模型的利弊。

- 第7章，遵守规则

这一章讨论验证形式的业务规则，以及基于领域模型的解决方案如何处理这样的规则需求，从而与第4章中的需求联系起来。

## 第三部分：应用 PoEAA

在这一部分中，我们将Fowler的《企业应用架构模式》[Fowler PoEAA]中的几种模式作为上下文，以此来讨论基础架构需要为领域模型提供哪些持久化支持。我们将看一下如何通过示例工具来满足这些需求。

- 第8章，用于持久化的基础架构

当我们拥有了很好的领域模型时，就该好好考虑基础架构了，本书中的主要基础架构类型是用于持久化的基础架构。第8章讨论持久化解决方案的不同属性，以及如何对特定解决方案进行分类。

- 第9章，NHibernate实战

这一章通过一个持久化解决方案的示例来使用前一章中的分类，这个示例就是

NHibernate[NHibernate]。

## 第四部分：下一步骤

这一部分主要关注并开始使用其他一些设计技术。另外一个重点是如何在领域模型中处理表示层，以跨越领域模型与表示层之间的鸿沟，此外还将介绍开发人员如何测试UI。这一部分几乎都是邀请其他作者编写的。

- 第10章，博采其他设计技术

这一章首先简单讨论限界上下文(bounded context)，然后讨论现在和将来都应关注的一些设计技术，例如面向服务、依赖注入/控制反转，以及面向方面。

- 第11章，关注UI

这一章主要讨论如何将UI连接到领域模型，以及在使用领域模型时如何提高富客户端应用程序和Web应用程序用户界面的可测试性。

## 附录

本书有两个附录，附录A包含更多领域模型风格的示例，附录B提供了简明的模式目录。

## 关于 C#示例

本书绝不是一本讲述C#的书。但我仍需要一种用于编写示例的语言(语言有多种，但我必须从中选择一种)，这样就选择了C#。

选择C#的主要原因是我当前经常使用它，而且大多数VB.NET和Java开发人员都很容易看懂C#代码。

至于版本方面，本书大部分示例代码可以同时C# 1.1和2.0中运行，个别章节的代码只能在C# 2.0中运行。

## 未涵盖的主题

本书中未涵盖的主题有很多，最主要的是分布式和高级建模。

### 分布式

本书最早的写作计划中是包含分布式方面的完整讨论的，但后来我发现这样本书的重点就过于分散了。但一些章节中仍然穿插了一些这方面的内容。

### 高级建模

本书的书名可能暗示书中会包含特定问题领域的一些高级和有趣的建模示例，但事实并不是这样。我们示例的重点更多地放在了应用TDD以及为DDD添加基础架构上。

## 最后的感言

相信读者能够理解，像本书这样的项目几乎从来不是一个人就能完成的。它是很多人共同努

力的结果。请参见致谢部分的人员名单，但还有更多参与人员未在这个名单中列出，特别是那些负责本书印制的人。虽然如此，但任何在印刷之前未发现的错误都应归咎于我一个人。

我将把读者感兴趣的一些信息发布到网上：[www.jnsk.se/adddp](http://www.jnsk.se/adddp)。

再回头说说跨越鸿沟，厄勒海峡大桥的照片是我的朋友Magnus von Schenck在他的一次海上旅行中拍摄的。

尽管本书的创作不像第一本书那样艰辛，但其中也凝聚了无数的心血、汗水和泪水。我希望通过我的这些付出，能够减少读者的付出。最后愿读者从阅读本书中获得乐趣，祝你们好运！

Jimmy Nilsson

[www.jnsk.se/weblog/](http://www.jnsk.se/weblog/)

2005年9月于瑞典Listerby



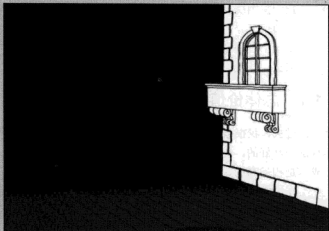
# Part 1

## 第一部分

## 背景知识

背景知识这一部分概括地讨论架构和过程，重点讲解领域模型和DDD（领域驱动设计）[Evans DDD]，还将介绍模式和TDD（测试驱动开发）。

第一部分是有关布置场景的。  
《罗密欧与朱丽叶》需要有一个场景。



### 本部分内容

- 第1章 应重视的价值，也是对过去几年的沉重反思
- 第2章 模式起步
- 第3章 TDD 与重构

# 应重视的价值，也是对过去几年的沉重反思

**本**章的目的是布置场景。本章将回顾过去几年中我如何思考不同概念，以及我的思想是如何随时间改变的。

我们将围绕很多话题讨论并涵盖大量基础知识，但总体思想是讨论在架构和开发过程方面应该重视的价值。

在这个过程中，我们将介绍并讨论很多概念，本书后面将对这些概念进行深入讨论。

那么，让我们开始吧！

## 1.1 总体价值

过去，我很擅长提前计划。我经常前瞻性地为项目添加功能、结构和机制。所添加的工件本身是非常好的，但我总是忘记它们从未发挥任何好的作用。当然，我也为项目增加了不少负担。成本是相当高的，包括开发时间和增加的复杂性。

在过去几年中，我们被鼓励使用另一种方法：“做可能管用的最简单的事。”在很大程度上，这个思想来源于极限编程（Extreme Programming, XP）运动[Beck XP]。另一种非常类似的说法是“你将不需要它”（You Aren't Going to Need It, YAGNI），这是帮助人们保持循规蹈矩的一种好方法。我猜想“保持简单，傻瓜”（Keep It Simple Stupid, KISS）也应属于此类。

这两种方法（预先添加所有能想到的与做最简单的事）是两个极端，但我认为它们都忽略了某些事情，即没有考虑到折中。正如所有关于“这个最好，还是那个最好”的问题一样，答案是“它取决于什么条件”。这是一个有关折中的问题。我倾向于选择这二者中间某个位置的方法，并根据情况选择向哪个方向偏移。“lagom”这个词是一个瑞典语单词，它表示“刚刚好”或“不多也不少”这样的意思。Lagom或“保持平衡”连同上下文敏感性就是我认为应该重视的总体价值，还有就是持续学习。

让我们更仔细地观察一些更具体的价值领域（架构和过程组成），我们从架构开始，以便将我们引入正确的语境。



## 1.2 应重视的架构风格

架构师和开发人员必须基于项目需求作出设计决策。这里并未罗列出有关架构风格的详尽价值，但对于其中的几个主要概念进行了一定的阐述，例如以模型为中心、领域模型、数据库、分发和消息传递。首先，我认为保持模型焦点是一种明智的做法。

### 1.2.1 焦点之一：模型

很长一段时间以来，我一直非常喜欢面向对象范型，但直到前不久，我自己才完全过渡到对该范型的全面采用。以前，使用该范型时存在平台问题，但现在平台已经成熟了。

**说明** 正如某位审稿人所指出的，成熟度取决于我们正在讨论的平台。如果你是VB背景，那么上面所说的话就是合理的，但如果你一直使用Java、SmallTalk、C#等，那么这些平台很长时间以前就已经成熟了。

大约13年前，我正打算构建一个系统，我曾尝试使用可视模型来表达对该系统需求的理解，并使用了一些OMT[Rumbaugh OMT]草图。[OMT是Object Modeling Technique（对象建模技术）的缩写，它是一种带有过程和表示法的方法。其表示法类似于统一建模语言（Unified Modeling Language, UML）中的表示法，这并不单纯是巧合，因为UML表示法的主要灵感就来自OMT。]当时，我们讨论了类之间的多样性，以及行为应该属于哪里等问题。在几轮讨论过后，我意识到了一个问题：在与专家用户进行讨论时，使用我的技术（而没有使用他们的技术）是完全错误的。他们随意地回答我的问题，因此并没有看到或为讨论贡献多大价值。（从总的结果看，系统并未失败，现在仍在使用中，并且是那里的核心系统，但开发过程并不顺利，如果我能改变方法，过程就会顺利得多了。）

### 1.2.2 焦点之二：用例

我时不时地会想起那段经历，并且对于当时的天真感到很好笑。在随后的几年中，我总是试图按照目标群体的方法学来与他们交流。我的意思是与用户打交道时就用用户的方法学，与开发人员打交道时就用软件开发方法学。有一种技术同时在这两大阵营中使用得非常好，那就是用例技术 [Jacobson OOSE]。（至少它在非正式方式的XP中（[Beck XP]）表现非常出色，这种方式就是我曾使用的短文本描述。短文本描述是指对系统中某一功能片段的简短描述，一个例子就是“Register Order for Company Customer”。）

它对于用户来说是很自然的，对于开发人员来说也可以变得自然，因此成为多年以来我一直使用的沟通工具。我所采用的沟通方式是软件中的每个用例都有一个类。

有一件事引起了我的警惕，即由于我对用例的应用方式的缘故，我的思维变得相当程序化。我的设计有点像是事务脚本[Fowler PoEAA]，但我曾尝试通过泛化尽可能多的行为（或至少是适当的行为）来平衡这一点。

几年前，我听了Ivar Jacobson讲解用例，我发现他没有将用例封装到它们自己的类中时，我

感到非常惊讶，因为我以为是要封装的，而且很长时间以来我一直采用这样的做法。

另一件引起我对此方法担忧的事情就是，当我与我的朋友Christoffer Skjoldborg（他是Valhalla开发人员）一起使用领域模型模式时，总是会产生争议[Fowler PoEAA]。他认为用例类几乎没有价值，并坚持认为它们甚至会在与用例部分混合使用时可能成为障碍。

### 处理主逻辑的不同模式

在继续讨论之前，很有必要介绍一些关于事务脚本和领域模型的知识，这两个概念我们已经接触过了。

Martin Fowler在他的*Patterns of Enterprise Application Architecture*[Fowler PoEAA]一书中讨论了在应用程序架构中建立主逻辑结构的三种方式，分别是事务脚本（Transaction Script）、表模块（Table Module）和领域模型（Domain Model）。

事务脚本类似于批处理程序，其中所有功能都是在方法中从头至尾描述的。事务脚本很简单，适合处理简单问题，但用于处理高度复杂问题时就显得无能为力了，否则重复将很难避免。但是，很多非常大的系统仍然是用这种方式构建的。我们也不例外，尽管已经尽力去避免它，但重复还是会发生。

表模块对Recordset进行封装 [Fowler PoEAA]，然后通过调用表模块上的方法来获取客户（id为42）的有关信息。为了获取名字，我们调用一个方法，并将42（即id）作为参数。表模块在内部使用Recordset来应答请求。这当然具有其自身的优点，特别是当具有很好的Recordset模式实现的时候。但是，一个问题是它也可能会在不同表模块之间引入重复。另一个缺点是通常无法使用多态解决方案（polymorphic solution）来解决问题，因为客户根本不会看到对象标识，而只会看到基于值的标识。这有点像使用关系模型来替代面向对象模型。

相反，领域模型使用面向对象来描述模型，而且尽可能接近所选领域的抽象描述。领域模型特别适合处理复杂性，这有两个原因，一是它充分利用了面向对象的强大功能，二是它容易恰当地描述领域的细节。当然，使用领域模型模式也不是毫无问题的——典型的问题就是，要想有效地使用它，就需要经历一个非常艰难的学习过程。

### 1. 重视领域驱动设计

在用例的使用方面，真正令我大开眼界的是领域驱动设计（Domain-Driven Design, DDD）[Evans DDD]。（我们将很快讨论有关DDD的大量知识，但现在，我们先简单概括一下：它将重点集中在领域上，并且对软件产生极大影响。）我仍然认为用例技术是与用户进行交流的极佳方式，但DDD让我觉得，如果能让用户积极参与到有关核心模型的讨论中，效果将会更好。这样就能尽早发现错误，并帮助开发人员更好地理解领域。

说明 “模型”是软件开发中使用极为频繁的一个术语。对于模型是什么，每个人都有个直观的理解，但我还是想描述一下我是如何理解它的。这里先不给出定义，而是简单地描述一下模型的几个特点。（遇上合适的作者，有关模型的内容就可以写成一本书），你可以

将自己的理解与这里的描述作一下对比。模型是：

- 由不同部分组成的
- 用于特定目的的
- 抽象的系统
- 认知的工具

此外：

- 模型有几种表示方法（例如，语言、代码和图解）。
- 一个系统中包含若干个模型。

如果你想知道更多，那么有关模型是什么这个问题有大量的资料。可以从Eric Evans的书[Evans DDD]的前几页入手（在此特别感谢Anders Hesselund和Eric Evans给予我的灵感）。

模型是开发人员与用户之间极好的沟通工具，而且二者之间的沟通越好，所开发出来的软件就会越好，无论从短期还是长期来讲，都是如此。

当然，我们（包括我）一直在使用模型，但在我的旧模型与那些用DDD思想创建的新模型之间有一个区别，即旧模型主要关注基础架构和技术概念。新模型没有将注意力分散到这些地方，而是完全集中在核心领域、领域概念以及当前领域问题上。这是一次大的思想转变。

换句话说，技术性的东西（例如用户界面风格）变化不定，唯有核心业务才是持久的。当核心业务改变时，模型和软件必须随之改变。

模型并不是高科技，也许我只是小题大做了。然而，我认为这恰恰是如何实现高效软件开发的核心，而且在我的经验中，也很少使用模型。这里要讲的意思是，将注意力集中在模型上，拥有一个同时供用户和开发人员使用的模型（具有不同表示），而不要被那些不重要的细节分散了注意力。

## 2. 为什么模型很重要

在这里，我稍微夸张一下。让我们做个游戏：要在一个专门领域（比方说金融领域）构建一个垂直系统，哪类开发人员是最佳人选呢？有一些人选是自不待言的，例如那些精通技术、技巧和经验丰富、拥有巨大社会网络的开发人员，等等。如果开发人员非常精通业务领域，那么也是上佳人选，如果他拥有10年的股市交易经验，那么就更好了，如果要构建的正是那类系统的话。

这类开发人员确实能够找到，但在我的经验中，这些开发人员只是凤毛麟角，而不是普遍现象。在金融系统开发完成并运行后，能够转移到另一个领域开发新系统的开发人员更是少之又少。只是因为开发人员无法在多个领域拥有10年的经验，例如物流领域、医疗领域、保险领域，等等。

另一种解决方案是可以让领域专家用户自己来开发软件。这一度是数十年来的梦想，在某种程度上这慢慢变得越来越可行。同时，这也消耗了他们的时间，而这些时间本可以更好地用在他们自己的核心工作上。而且，这也涉及了大量的技术问题。

那么，下一个最佳解决方案是什么呢？答案是显然的（至少站在今天的立场上是这样）：让开发人员尽可能多地学习他们正在工作的领域，并让用户加入到整个环境中，以便将领域知识带

给项目团队，并积极、有建设性地参与到项目工作中。用户将不仅仅建立需求，虽然这也极为重要，实际上还帮助设计系统的核心。如果我们能够设法建立这样的氛围，那么没有人能够比领域专家用户更好地决定什么是核心，什么是关键抽象，等等。

当然，开发人员仍然是必需的。整个工作是协作完成的。例如，有些事情在业务人员看来是很小的细节，但在我们看来可能就是领域模型中的大事。他们可能会说：“哦，有时它反而像这样。”而且每件事情都完全不同了，但由于它不是最常见的变化，因此他们认为这并不重要。

### 3. 到了再次尝试与客户讨论模型的时候了

为什么现在我能成功地与客户讨论模型，而以前却不能呢？很多事情都改变了。首先，对语境和项目小组有一个认识是很重要的。其次，用例在开始时能够提供大量帮助，然后我们可以将重点转移到核心模型本身上。第三，更多的人现在有了构建系统的经验，他们也进行了相关学习。第四，严格地讲，虽然模型现在被表示为图形（例如UML样式的草图），但这并不是让用户积极参与项目的最重要事情，模型作为一种表示，通常开发人员比用户更喜欢它们。我们要寻找的是一种通用语言（ubiquitous language）[Evans DDD]。

通用语言并不是像UML、XML Schema或C#这样的语言，它是一种自然的但经过浓缩的领域语言。它是一种我们与用户共享的语言，用来描述手头的问题和领域模型。通用语言并不是把从用户那里听到的内容翻译为我们自己的语言，而是减少误解，让用户更容易理解我们的草图，并且真正帮助纠正错误，帮助我们获取有关领域的新知识。如果它是适当的，那么我们可以在图形/代码模型的上下文中与用户讨论通用语言。如果不适当，那么要想领会通用语言，还有最重要的事情要去做。

---

**说明** Eric Evans对前面观点的评论是：“有一件要澄清的事情是，通用语言并不仅仅是领域专家的当前行话。专家的行话有着太多的歧义和假设，而且或许范围也太大了。通用语言是作为领域专家与软件专家之间的协作而演进的。（当然，它就像是领域行话的一个子集。）”

---

我们应该保证通用语言具有完善的定义，并且与软件保持同步。例如，通用语言中的更改要求对软件做出相应更改，反之亦然。这两种工件之间应该互相影响。

### 1.2.3 如果重视模型，就可以使用领域模型模式

如果大家一致认为应该高度重视模型并使用面向对象方法，那么我相信使用领域模型模式来建立应用程序和服务的核心逻辑就是一个自然的结果了[Fowler PoEAA]。图1-1显示了小的领域模型示例。

---

**说明** 如果你对于UML不是非常熟悉，那么我推荐一本好书UML Distilled [Fowler UML Distilled]。虽然了解UML并不是十分重要，但掌握一些框架性知识还是有帮助的，因为本书很多地方将使用UML作为草图绘制工具（当然，掌握UML知识也将使我们在其他一些场合获益，而不仅限于本书）。

---

虽然图1-1只显示了小的领域模型的早期草图，但我们可以看到该模型表示了领域概念，而不是对技术的抽象描述。我们也可以看到它包含几个互相协调的小片段，它们共同形成一个整体。

说明 图1-1之所以采用手写形式，是为了强调它是一张草图。我们将借助于测试驱动开发（Test-Driven Development, TDD）来探索和开发代码中的细节。（本章后面和第3章将讨论TDD。）

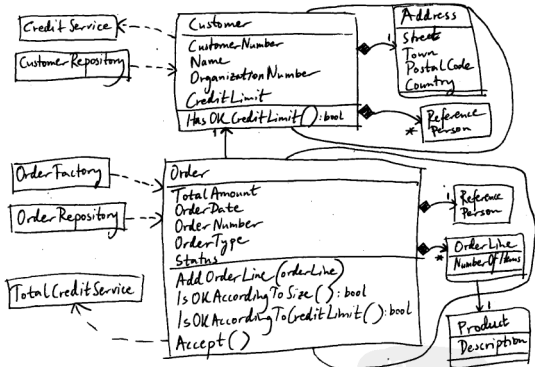


图1-1 领域模型示例，它是示例应用程序的早期草图，我们将在第4章中深入讨论它

在进一步讨论有关领域模型所使用的具体风格之前，我们再次做一下回顾。

### 1. 从我的后视镜看到的领域模型历史

我仍然记得我第一次尝试使用领域模型时的情形，虽然那时我并未将其称为领域模型。当时大概是1991年，我无法理解应该如何让它在用户界面中协调。我不记得应用程序的名字了，但我记得我曾尝试决定如何让菜单外观符合面向对象的特征。那次我没有克服第一道障碍。我认为我当时正在UI与领域模型之间寻找一个很近的映射，但不知道如何实现它。

在那之前的几年时间，我开发了一个据说使用了面向对象思想（比如在UI方面）的应用程序。那个项目没有突破性成就，但确实小有成就。例如，它没有首先选择功能，然后再决定什么对象

(例如portfolio)应用该功能,而是由用户首先选择对象,然后决定对于选定的对象哪些不同功能是可行的。当时,这与大多数业务应用程序在UI中的导航方式是非常不同的。

### 小议裸体对象

也许我并没有完全脱离我的本意。也许我当时正在尝试的是某种类似于裸体对象(Naked Object) [Pawson/Matthews Naked Objects]的东西,但我实际上离它并不远了。

我们不必深入讨论裸体对象的细节,一个简短的解释就足够了。其基本思想是,不仅开发人员喜欢面向对象,而且用户也喜欢。他们的喜欢程度甚至超过我们的想象,而且他们通常很希望看到非常接近于领域模型的UI,最好是接近到这样一个程度:能够使用直接基于领域模型的框架来自动创建UI。

基于裸体对象的系统自动向用户呈现一个表单,其中包含一些小部件,它们公开了领域模型类中的属性。

对于更复杂的任务,需要进行一些自定义(像通常那样),但这里的思想是减少自定义,这再次归功于框架理解了要从领域模型片段创建什么样的UI。因此,当领域模型“完成”时,UI也或多或少被完成了。

目前,我在概念和技术方面没有第一手经验,但我对这种思想的一种有趣的理解是这样的:用户将有机会真正看到和感受模型,这应该对消除开发人员与用户之间的鸿沟有极大的帮助。

在那之后的许多年中,我曾多次尝试领域模型方法。我发现了它的使用问题,特别是与性能开销有关的问题(尤其是在分布式服务中)。尽管如此,实际上我的一些应用程序还是使用该风格构建的,特别是那些较简单的应用程序。还有一件重要的事情是,无论我多么希望构建非常强大的、设计完善的领域模型,但结果总是达不到我的预期。

这个问题并不是我遇到的唯一难题……

### 2. 原来的事实可能是错误的

我过去一直相信领域模型,并多次尝试使用它们,但在面向对象风格中的大多数尝试都得出了同一个结论——它没什么用。例如,我在Visual Basic 6 (VB6)中开发COM+应用程序时就是如此。主要问题是性能开销太高了。

当我写前一本书时[Nilsson NED] (该书描述了.NET应用程序的默认架构),我重用了原来的VB6/COM+知识,而且根本没有开发基于领域模型的架构。

后来我再次开始了新的实验,把在.NET中使用领域模型与使用Transaction Scripts和Recordset做一下对比,看一下是否能在一般情况下减少10%的响应时间(例如取回一个订单、取回某客户的订单清单或保存订单)。令我感到惊讶的是,我可以实现更低的开销。原来的事实是错误的。现在,与在VB6中使用领域模型相比,在.NET中使用基于领域模型的架构变得更具有可行性。一个原因就是实例化时间减少了好几个数量级。因此,当有很多实例时,开销会低得多。另一个原因是在.NET中很容易编写按值编组(Marshal By Value)组件(这样,不仅到实例的引用被移动了,而且至少在概念上整个实例也被移动了)。在COM世界中这并不容易做到。(甚至从一般意义上讲,在VB6中是不可能实现的。)

**说明** 按值编组的重要性较为次要，因为我们现在常见的做法是不通过数据线来按原样发送领域模型，但我的早期测试却使用了这种方法。

然而，另一个原因是.NET更好地支持面向对象，它只是更好的工具箱。

**说明** 值得一提的还有一件事情：过去在微软社区中并不十分重视领域模型。另一方面，Java社区几乎恰恰相反。我记得当我访问一个主要由Java开发人员组成的工作小组时，我问他们最喜欢什么逻辑结构，以及是否大量使用了Recordset（在JDBC版本3中增加的）。他们有趣地看着我，就好像没听懂我的问题一样。我立刻就认识到领域模型或多或少已经成为他们的事实标准了。

直截了当地讲，我想澄清两件事情。首先，我并不是说由于性能原因而应该选择领域模型模式，而是通常可以在不引起性能问题的情况下选择领域模型模式。

其次，我并不是说为了能够使用模型，需要一门特定技术。不同技术或多或少地适用于表示那些与领域具有共性的软件中的模型。当然，没有哪一种技术总是最好的。到底哪种技术适用，取决于与不同领域和不同问题相关的多种因素。因此我把技术看作是助推器。不同的技术可能是比其他技术更好的助推器。

总之，领域模型模式是否可以被足够有效地使用，在很大程度上这是一个设计问题。

有关这方面的一些好消息是我们有大量可以获取的好的信息。领域驱动设计及其构建领域模型结构的风格提供了大量有价值的帮助，这也是本书中要花大量时间去尝试的。

### 3. 一个根结构

那么，如果我们选择以领域模型为中心，这意味着我们将让项目中的所有人员都参与到构建模型的工作中来，当然包括开发人员。

甚至DBA也认同将领域模型视作根结构，尽管数据库设计略微不同。如果这样，我们或许就已经完成了让DBA讲通用语言的工作！（事实上，越是能让DBA喜欢领域模型并遵守它，就越容易实现它。这是因为如果数据库设计与领域模型没有根本性区别的话，而更像是同一个模型的不同视图，那么我们只需创建更少的映射代码。）甚至用户也可以接受领域模型。当然，不同受众对于领域模型有着不同需求，例如，他们在细节级别方面具有不同视图，但它仍然是一个根结构——与我们共存、共同发展并且不断变化的结构……

随后你将会发现，本书从现在开始将会在讨论基于领域模型的解决方案中投入大量精力，但在此之前，我们先讨论一些有关其他架构价值的内容。让我们暂时离开领域问题，讨论一些技术性更强的内容。首先是如何处理数据库。

## 1.2.4 慎重处理数据库

过去，我在构建系统时对性能的考虑很多。例如，我习惯将所有数据库存取编写为手写的存储过程。这通常具有很高的运行时性能效率，特别是对于以下情况：每次对一个实例（例如一位客户）不仅仅具有CRUD（创建、读取、更新、删除）行为，而且还有“更智能”的存储过程，

例如在每个调用中执行多项操作并影响多个行。

### 1. 正确的效率

尽管硬件能力可能仍在按照类似摩尔定律这样的规则增长，但性能问题仍然是每天都会遇到的问题。在硬件能力增长的同时，我们试图用软件来解决的问题的规模也变大了。

以我的经验，性能问题常常是由于糟糕的数据库存取代码、数据库结构或其他类似原因造成的。一个常见的原因是根本没有在数据库调优上花费精力，而只是一味地追求面向对象的“纯度”。这进而又导致大量迂回低效甚至是错误的事务代码、糟糕的索引模式，等等。

稍微具体地讲，我们来看一下示例，其中面向对象被认为是重点，而数据库处理则被忽视了。我们假设有一个Customer类，它具有一个Orders列表。每个Order实例都有一个OrderLine实例列表。如图1-2所示。

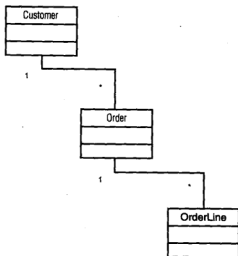


图1-2 Customer/Order示例的UML图

此外，我们假设模型对我们来说非常重要，而且我们不太关心数据是如何在关系数据库中存取的。这里，一个可能（然而幼稚的）方案是让每个类（Customer、Order和OrderLine）负责它本身的持久化/非持久化。这可以通过添加Layer Supertype [Fowler PoEAA]类（在图1-3的示例中称为PersistentObject）来实现，Layer Supertype类实现了GetById()，而且领域模型类从它进行继承（参见图1-3）。

现在，假设我们需要取回一个Customer及其所有Orders，并且对于每个订单，取回所有OrderLines。那么，可以使用以下代码：

```
//A consumer
Customer c = new Customer();
c.GetById(id);
```



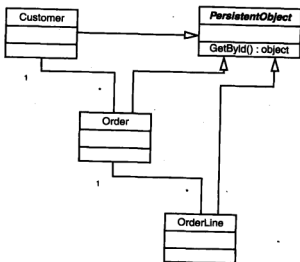


图1-3 Customer/Order示例的UML图，带有Layer Supertype

**说明** 也许你想知道以上代码片段中的 `//A consumer` 是什么意思，那么这里的想法是为像上述的代码片段显示类的名称（有时是方法），以提高清晰度。有时，具体的类名称（像本例中）并不重要，那么我就用一个更一般的名称来替代。

C++中已经建立了此概念，方法名称被写为像 `ClassName::MethodName` 这样，但我认为一些小的注释应该可以起到相同效果。

现在将执行以下SQL语句：

```
SELECT CustomerName, PhoneNumber, ...
FROM Customers
WHERE Id = 42
```

接下来，客户的 `GetById()` 方法将取回订单列表，但通过调用 `GetIdsOfChildren()`，将只取回关键字，它执行的操作类似于：

```
SELECT Id FROM Orders
WHERE CustomerId = 42
```

然后，`Customer` 将通过在 `Order` 标识符的 `DataReader` 上执行迭代过程来逐个实例化 `Order`，它按照以下方式将实际工作指派给 `GetById()` 方法：

```
//Customer.GetById()
...
Order o;

while theReader.Read()
```

```
{
    o = new Order();
    o.GetById(theReader.GetInt32(0));
    c.AddOrder(o);
}
```

然后，对于每个Order，就到了取回所有OrderLines的标识符的时候了……这就是整个过程。这里发生的事情就是在一定程度上从对象的角度来考虑问题（这至少是设计者的意图），而不是从数据集角度来考虑，而关系数据库是基于数据库集的。由于这个原因，到数据库的迂回访问是非常多的（要取回的每一行都另外加了一些操作）。这样，效率就低下到了极点。

**说明** 值得一提的是，以上描述的行为可能恰恰是在特定场景中加载大量数据时需要避免的。事情并不总是一成不变的。要考虑背景。

另一方面，如果我们考虑另一个极端——手写和手工优化的存储过程，那么可能会像这样：

```
CREATE PROCEDURE GetCustomerAndOrders(@customerId INT) AS
    SELECT Name, PhoneNumber, ...
    FROM Customers
    WHERE Id = @customerId

    SELECT Id, ...
    FROM Orders
    WHERE CustomerId = @customerId

    SELECT Id, ...
    FROM OrderLines
    WHERE OrderId IN
        (SELECT Id
         FROM Orders
         WHERE CustomerId = @customerId)
```

它们通常在运行时是有效的（像我所说的那样）。但在维护期间，它们的效率很低。它们将产生大量需要手工维护的代码。此外，Transact SQL（T-SQL，用于Sybase SQL Server和微软SQL Server的SQL方言）中的存储过程并不能帮助一般的技术避免代码重复，因此程序中会有大量重复代码。

**说明** 我知道，有些读者现在会说，在不使用存储过程的情况下可以非常有效地解决前面的示例，或者说该存储过程不是适用于所有情况的最有效的一个。我只是试图从效率观点指出两个极端示例。我的意思是至少从效率的观点将二者做一下对比：一是设计糟糕的代码，它使用了动态SQL，二是设计得更好、编写得更完善的存储过程。

因此，问题是运行时效率对于选择如何设计是否是最重要的因素？

## 2. 可维护性是重点

假设我只能选择一种“能力”作为最重要的一个，那么近来我会选择可维护性。可维护性并

不是我们唯一需要的，它也绝对不是，但它通常比其他方面更重要，例如可伸缩性。借助于好的可维护性，可以轻松实现其他能力和经济高效性。当然，这种说法非常简单，但它确实指出了一个重点。

从另外的角度来看，在整个生命周期期间，将开发新系统的总成本与系统维护的总成本进行比较。以我的经验，大部分成功系统的维护成本要高得多。

总而言之，我目前认为给予数据库一定的关注是值得的。我们应该把数据库看作朋友，而不是敌人。但同时，手写处理数据库的所有代码通常并不是“正确的”方法。这与我要强调的模型毫无关系。正如通常被引用的Donald Knuth的那句名言：“过早的优化是一切麻烦的根源。”因此，除非确实需要优化，否则一定不要提前优化。

简单地讲，我认为适当的设计再加上对象关系映射（O/R映射）就已经足够好了，如果这还不够好，那么我们就应该进行手工调优。O/R映射有点类似于数据库优化器，大部分时间它是足够智能的，但在某些情况下我们需要助它一臂之力。

---

**说明** 本书通篇将讨论大量有关适当设计和O/R映射的内容。现在先将O/R映射定义为用于衔接面向对象和关系数据库的技术。我们描述面向对象模型与关系数据库之间的关系，其他工作则由O/R映射工具来完成。

---

因此，一种方法就是，对于大部分数据库存取代码，使用像O/R映射器这样的工具，然后在需要通过手工编写代码来提高执行效率时再手工编写。

让我们更仔细地看一下O/R映射器必须要处理的问题，即两个不同世界之间的映射：面向对象世界和关系世界。

### 1.2.5 领域模型与关系数据库之间的阻抗失配

前面曾提到，当使用领域模型时，一个问题就是到UI的映射。而另一个众所周知的问题就是到关系数据库的映射。这个问题通常是指关系世界与面向对象世界之间的阻抗失配（impedance mismatch）。

这里将给出我对阻抗失配的一些认识，尽管这些认识很通俗。有关更深入、更正式的信息，参见Cattell [Cattell ODM]。

首先，如果同时使用关系数据库和面向对象模型的话，那么有两个类型系统。问题的一部分原因在于类型系统位于不同地址空间中（即使在同一台机器上也是如此），因此我们必须在它们之间移动数据。

其次，甚至没有基本类型是完全相同的。例如，.NET中的字符串具有可变长度，但在Microsoft SQL Server中，字符串通常是varchar，或者是char或text。如果使用varchar/char，就不得不决定最大宽度。如果使用text，那么程序模型则与SQL Server中的其他字符串类型完全不同。

另一个例子是DateTime。.NET和SQL Server中的DateTime非常类似，但也存在区别。例如，在.NET中，精度可以达到100纳秒，但在SQL Server中“只能”达到3/1000秒。另一个“有趣的”

区别是，如果在.NET 中将DateTime设置为DateTime.MinValue，那么当尝试将它存储到SQL Server的DateTime中时，将导致异常。

还有一个区别是对null的处理能力。在.NET中无法将null存储到普通的int中，但在SQL Server中这却是有效的。

**说明** 上述问题取决于是否使用领域模型。

关键的区别在于如何处理关系。在关系数据库中，关系是通过重复的值形成的。父表的主键（例如Customers.Id）作为子表的外键（例如Orders.CustomerId）重复，这有效地使子表的行“指向”它们的父表。因此，关系模型中的一切事物都是数据，甚至关系也是数据。在面向对象模型中，有很多种不同方式可用来建立关系（例如，通过类似于关系模型中的值，但这并不是典型方式）。最典型的解决方案是使用内置对象标识符，这些标识符使父拥有对其子中的对象标识符的引用。正如我们所见，这是一种完全不同的模型。

关系模型中有两种导航方式。首先，可以使用一个父表的主键，然后使用查询来查找外键值等于父表主键值的所有子表。然后，对于每个子表，其主键可用于进行新的查询，来找到它的所有子表，依此类推。关系模型中另一种（也可能是更典型的）导航方式是使用父集与子集之间的关系连接。在面向对象模型中，典型的导航方式是简单地遍历实例之间的关系。

下面是两个代码片段，这里有一个OrderNumber为42的订单。现在我想知道该订单的客户名称。

C#:

```
anOrder.Customer.Name
```

SQL:

```
SELECT Name
FROM Customers
WHERE Id IN
    (SELECT CustomerId
     FROM Orders
     WHERE OrderNumber = 42)
```

**说明** 对于后面的SQL代码段，使用JOIN也是可以的，但我想这里使用子查询会更清楚。

另一个与导航有关的区别是，对于对象来说，导航是单向的。如果需要双向导航，实际上可以通过两个单独机制来实现。在关系模型中，导航总是双向的。

**说明** 有关方向性的问题也可以认为与我刚才所解释的相反，因为在关系数据库中，在一个方向上只有一个“指针”。我仍然认为“指针”是双向的，因为可以使用它在两个方向上遍历，这也是我认为非常重要的一点。

正如我们已经接触过的，关系模型是基于集合的。每个操作处理的都是集合。（集合可能只有一行，但它仍是集合。）但是，在面向对象模型中，我们每次处理的是一个对象。

此外，关系模型中的数据是“全局的”，而我们在面向对象模型中却要努力保持数据的私有性。

就设计而言，粒度就非常不同了。让我们通过一个示例来解释这一点。假设我们想跟踪某个人的一个家庭电话号码和一个工作电话号码。在关系模型中，通常有一个名为People的表就可以了（复数是事实上的命名标准，至少在数据库人员中是这样的，以便清楚地表示出我们正在处理集合）。

**说明** 如果我是一个挑剔的人，那么当谈论关系模型时，应该使用“关系”（relation）这个词，而不是“表”（table）。

这张表有3行（也许更多，但这里我们只讨论电话号码），分别表示一个主键和两个电话号码。也许会有5列，因为我们可能要将电话号码分解为两列，分别用于地区代码和本地号码，如果又添加了国家代码的话，那么就分为7列。参见图1-4做一下比较。

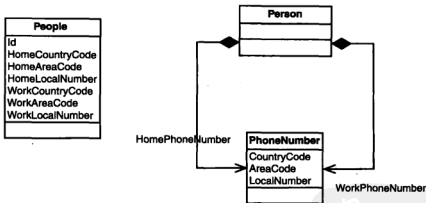


图1-4 分别用关系方式和面向对象方式来表示同一个模型

这里重要一点是，即使是1:1，通常所有列也都是都在一张表中定义的。在面向对象模型中，通常要创建两个类，一个是Person，另一个是PhoneNumber。这样，一个Person实例由两个PhoneNumber实例组成。我们可以在关系模型中做类似的事情，但它通常没有任何意义。在关系模型中，我们尽力避免大量重用定义，特别是因为我们没有与表定义相联系的行为。在面向对象模型中却恰恰相反。换种方式来说就是，在关系模型中，如果将PhoneNumbers移出来，并放到单独的表中，不会使形式变得更正规，而可能只是增加了开销。这归结为一句话：关系模型是用来处理表格类型的基本数据的，这既有好的一面，也有坏的一面。面向对象模型也很善于处理复杂数据。

**说明** 就定义重用而言，关系模型有一个很强的概念，被称为领域。不幸的是，在当今产品中对这个概念的支持并不是很强。

还有一种情况就是，很多产品已经支持复杂数据类型，但仍处在不成熟的阶段。

刚才我们讨论了粒度级的一个例子，其中关系模型比面向对象模型具有更粗的粒度。我们也可以从其他方面来看一下。例如，在关系模型中，一个order可以有多个orderLines，但可以说orderLines是独立的。每个orderLine只是一行，每个order是另外一些这样的行。它们之间存在关系，但行是单元。在面向对象模型中，将order看作单元，并且让它由orderLines组成，这可能是好的解决方案。在这种情况下，关系模型就比面向对象模型具有更细的粒度。

**说明** 这里并不是暗示领域模型中不应该有OrderLine类。这个类是应该有的。这里要说的是，我所要求和使用的单元是order，而orderLines是order的一部分。

最后，但并非不重要，关系模型不支持继承（至少在大多数流行产品中这不是主流）。继承是面向对象模型的核心。当然，我们可以在关系模型中模拟继承，但仅仅是模拟而已。不管选择哪种模拟解决方案，都会带来一定的损害，而且会导致存储及速度开销，和/或关系扭曲。

**说明** 在数据库中深度集成本机XML看起来是缓解阻抗失配问题的最新尝试。但XML实际上也是第三方模型，作为分层结构模型，它对于面向对象世界和关系世界均存在阻抗失配。

因为我的应用程序通常是使用关系数据库，因此阻抗失配导致了很大的问题。

Data Mapper模式[Fowler PoEAA]可用于处理这个问题。Data Mapper模式用于描述领域模型与数据库之间的关系，然后移动工作就可以自动完成。不幸的是，Data Mapper模式本身并不灵活，特别是在.NET平台中，对象关系映射工具（O/R-mapper）产品落后了好几年。在Java-land中，有几个成熟产品，甚至有一个标准化的规范，称为JDO [Jordan/Russell JDO]，这使得两个平台在这方面完全不同了。

领域模型和数据库的讨论先告一段落了，下面我们讨论分布式，以此作为架构小节的结尾。

## 1.2.6 谨慎处理分布式

你是否意识到了“我有一个新工具，让我们用起来”这种综合症？我已经意识到了。我时时被它折磨着。这种综合症会带来不少问题，但我觉得它并不只有负面作用，实际上也有积极的一面。毕竟，作为一个信号，它可能表明了积极的好奇心、健康的进取思想以及对进步的永无止境的追求。

好，我先做了这些铺垫，下面我要讲述我过去的一些恐怖经历了。大约10年前，MTS (Microsoft Transaction Server, 微软事务服务器) 横空出世了，突然间所有COM开发人员都非常容易构建分布式系统。发生了什么事情？因为人们已经构建好了分布式系统的负载。我正是构建这些负载的那些人当中的一员。

说明 这里我将使用MTS这个名字，但你也仍可使用COM+或COM+Component Services或Enterprise Services。

某些系统很适合采用分布式方法。将某些处理任务从客户机和数据库转移到应用服务器上可以带来利益，就像资源共享一样（例如，在服务器端提供连接池，以便可以用少量数据库连接为数百个客户端服务）。某些系统急需这些利益，而且现在也更容易实现这些利益。

### 1. 过度使用从来就不是好事

不幸的是，这些利益的诱惑力太大了，以至于常常导致适得其反的结果。例如，服务器端的工作可以被分解到几台不同的应用服务器上。但应用服务器并没有被克隆以便让所有客户端可以使用任意服务器并获取所需服务，而是客户端调用应用服务器中的一台。该服务器调用另一台应用服务器，后者又调用另一台应用服务器。这只会得到一些小的利益，但它的巨大缺陷是提高了延迟并降低了可用性。

另一个错误是，只有两个并发用户的应用程序也专门针对MTS编写，并在MTS中运行。这是典型的小题大做，因为开发者相信应用程序在不久的将来可能会非常流行，并拥有数以千计的并发用户。问题并不仅仅是更复杂的操作环境，而是应用程序的设计具有很多含义（如果你想成为MTS参与者的话）。这方面的一个典型例子就是避免特定层之间（或更糟的是，在所有层之间）的过度会话。这些做法确实在很大程度上增加了设计的复杂性。因此我们说过度使用从来就不是好事。

你是否注意到，我没有讨论分布式系统初学者在最初尝试分布式系统时会遇到的一些常见陷阱，例如过度会话。即使由于某种原因这些错误被避免了（虽然这样的机会很渺茫），但仍然会有其他的更难以琢磨的问题。

正是由于这些问题，Martin Fowler提出了他的“分布式对象设计第一定律”，即“避免分布式”[Fowler PoEAA]。如果不是绝对有必要设计分布式，那么就不要这样做。成本和复杂性只会非常高。这是一条自然法则。

### 2. 分布式可能是有用的

无论现在还是将来，分布式都有好的一面，例如：

#### ● 容错性

如果所有程序都运行在一台机器上，那么如果这台机器发生故障，就有麻烦了。对于一些应用程序来说，这个风险无关紧要，但对于其他一些程序，其坏处将超乎我们的想象。

#### ● 安全性

对于将应用程序分布到多台服务器上是否会提高安全性这个问题，一直就有激烈的争论。不管怎样，安全性问题是采用分布式系统的一个常见原因。

#### ● 可伸缩性

一些应用程序的负载太高了，因此无法在一台机器上处理。还有一个金钱上的原因是，我们可能不想一开始就购买最昂贵的机器。当问题升级时，我们可能希望通过添加更多机器来解决问题，同时不淘汰原有的机器。

如果不需要分布式系统，那么对设计加以慎重考虑是非常重要的，因为我们会遇到一个大的设计挑战，整个过程很容易出现大量问题。当讨论分布式时，消息传递是个重要概念。

### 1.2.7 消息传递很重要

本书的焦点是一个应用程序或一个服务的设计，而不是如何编排多个服务。尽管如此，我仍认为首先思考消息传递还是有必要的（如果你尚未开始思考的话）。未来，这个问题对于我们来说只会越来越重要。

#### 1. 消息是核心编程模型中的一件重要事情

我曾经认为在分布式系统中将网络进行抽象或隐藏是一种好的方式。这样，客户端程序员就不知道方法调用到底是转换为网络跳转（network jump），还是只是本地函数调用。我喜欢这种方式的原因在于，它可以简化客户端程序员的工作，以便让他们可以将注意力集中在那些对创建用户界面至关重要的因素上，而不会因网络通信这样的事情而分散注意力。

让我们来看一个简单的示例。假设有个Customer类，如图1-5所示。



图1-5 Customer类

Customer类的实例通常（也有希望）一直与客户代码驻留在相同的地址空间中，因此当客户要求Customer的名称时，它只是一个本地调用。

但是，没有任何实际理由阻止我们将Customer仅作为代理[GoF Design Patterns]，这进而将延迟对驻留在应用服务器中的Customer实例的调用。（这里说得有些简单了，但如果在应用服务器的MTS中配置Customer类，那么这或多或少将是后台要发生的事情）。

Szyperski指出[Szyperski Component Software]，位置透明性既是优点，也是负担。优点是所有类型的通信（进程中、进程间和机器间）都被映射到一个抽象，即过程调用。负担则是它隐藏了不同调用类型之间的巨大成本差异，通常是执行时间上的数量级差异。

我认为当前趋势正好与位置透明性相反，而且是通过使消息成为编程模型中的核心事物来使得成本高昂的消息变得明显。作为趋势的感知者，我确实喜欢这种演变。我喜欢它的原因只是因为客户端程序员显然知道它将是网络调用，这并不意味着它一定很难实现。例如，可以提供消息工厂，但是程序员仍然能够更清楚地知道什么花费时间，什么不花费时间。

你可能对消息传递对领域模型有什么影响感到疑惑。领域模型的需求并未消失，但可能会有一些小的变化。第一个出现在我们头脑中的例子就是，插入操作（像在日志中一样）比更新操作（甚至是为了修改）更受人们喜欢。这使得异步通信变得更可用。

#### 2. 推迟它，直到能够更好地完成

消息传递的一个重要优点是它极大地提高了执行灵活性。过去，我认为自己已经很擅长使用批处理作业来处理一些不必实时执行的较长任务。这是一种古老的方法，但我认为它未被充分用



来极大地提高实时的那部分工作的响应时间，例如，长的执行任务可以在低负载的窗口中进行，而且通常是在夜间。

目前，我已经编写了大量同步应用程序。当某一功能片段需要很长的执行时间，而且不必实时执行时，我就把该功能更改到批处理进程中。实时执行的任务则从完全的功能片段更改为请求本身。

此问题的一个更有效的解决方案是从开始就尽可能多地考虑使用异步消息。这样，在适当情况下，就可以实时运行功能，或者功能被放到消息队列中等待尽快执行，或者以给定的时间间隔执行。（批处理解决方案从一开始就是一种内置功能。）

当我们构建自己的用户界面时，基于异步消息的解决方案可能需要一种完全不同的思维，但要想真正解决不同的设计问题，异步解决方案通常比你过去想象的更可行更适用。

最后对以上讨论总结几句话：在我的观点中，无论执行环境如何，聚焦于核心模型都是一种好的思想。然后，根据需要，可以在几种不同情形中使用它。

以上是对架构价值的简单评价。接下来我们讨论过程。

### 1.3 对过程的各个组成部分的评价

我并不是一位过程专家，但在这里我仍要给出我的一些想法。首先，我讲一下这里说的“过程”（process）是指什么。它是推动项目前进的方法学：应该完成什么，应该在何时完成，以及如何完成？

经典的过程就是所谓的瀑布过程。每个阶段都依照严格的线性模式跟在另一个阶段后面，任何时候都不能向后返回。一个简短的描述可能是这样的：首先编写规格说明，然后从该规格说明构建系统，然后是测试，再后是部署。

在其后的很多年中，大量不同的过程被引入，它们都有着各个的价值和缺点。最近有一个过程是极限编程（XP）[Beck XP]，它可以被归为敏捷[Cockburn Agile]过程的旗下，XP可以被描述为瀑布过程的相反过程。XP的基本思想之一是，我们不可能在第一天就知道足够的事情来编写真正完善、详尽的规格说明。知识是在项目期间演进的，这不仅是因为时间的流逝，而且也因为系统的一些部分已经被构建了，这是获得洞察力的非常有效的方式。

---

说明 XP的内容比我刚才说的要多得多。有关更多信息，参见[Beck XP]。我将讨论TDD和重构，它们在XP中都有根源。

另外要注意，XP并不总是从零起步的。在XP论坛中，也有大量有关XP和遗留系统的有趣内容。参见 *Object-Oriented Reengineering Patterns* [Demeyer/Ducasse/Nierstrasz OORP]。

---

我尝试针对现状寻找一个好的组合（由较小组成部分组成）。这里介绍几个我所喜欢的不同组成部分。下面将讨论领域驱动设计、测试驱动开发和重构，但先让我们从预先架构设计开始。

## XP和对用户的关注

XP的另一个新颖之处在于，它对以用户为中心的开发的关注。用户应该参与到项目的整个开发过程中。

对于瑞典人来说，这并不是特别的新鲜事物。我不知道为什么，但在瑞典，我们已经有相当长一段以用户为中心开发的历史，在XP之前很久就是这样了。

这里并不是说瑞典比其他国家做得更好，我们也有相当长一段使用瀑布过程的历史。

### 1.3.1 预先架构设计

虽然我喜欢敏捷思想中的很多主张（如对于相关阶段不可能知道的事情，不做预先和过早作出决定），但我也并不认为新项目应该只从一张白纸开始。通常，在产品环境、预期负载和预期复杂性等方面，我们从一开始就拥有非常多的信息（越多越好）。至少要将这些信息储备到我们的大脑中，并用来进行一些预先架构的初始/早期概念验证。

#### 1. 重用来自成功架构的思想

如果每个新应用程序都从零开始，那么代价是我们无法承担的，对于架构来说尤其如此。我总是思考我目前最喜欢的默认架构，并看一下它是否符合正在构建的新应用程序的情形和需求。我也总是评估最近的几个应用程序，考虑如何能够从架构角度改进它们，在这个过程中，仍然将新应用程序的上下文纳入到我的思考中。经常评估和尝试改进肯定是有价值的。

如果现在假设你喜欢领域模型模式[Fowler PoEAA]的思想，那么做出初始的架构决策实际上并不难，因为大量的重点都将汇集到领域模型上。决定是否使用领域模型实际上是预先架构决定，也是非常重要的决定，它将影响很多后续工作。

还要指出的重要一点是，即使做出预先决定，它也并不是固定的，甚至不一定是有关是否使用领域模型模式的决定。

---

**说明** 我最近听一些权威人士建议，只要可能，就实现具有Recordset模式[Fowler PoEAA]的事务脚本模式[Fowler PoEAA]，并且当有必要时迁移到另一个解决方案。我不同意这种观点。是的，可能有一些不好的事情会发生，但在项目后期我们不喜欢这种迁移，因为它影响的东西太多了。

---

#### 2. 一致性

作出预先架构决定的一个重要原因是可以让开发团队保持同一个方向。为了保持一致性，需要一些方针。

对于为公司构建很多应用程序的IS部门来说，亦是如此。如果应用程序的架构在某种程度上具有类似性，那么是非常有益的，因为这使得在项目之间调动人员变得更加容易且更高效。

#### 3. 软件工厂

这恰好将我们带到了对软件工厂[Greenfield/Short SF]的一些讨论中（灵感来自产品线架构

[Bosch Product Line])。软件工厂的思想是在软件公司中有两条生产线。一条生产线负责创建架构、框架等一些用于应用程序家族的对象。另一条生产线则负责使用第一条生产线生产的产品来创建应用程序，从而摊低框架在多个项目上的成本。

说明 一个问题在于发明框架是很麻烦的。更好的想法是直接获取框架[Fowler Harvested-Framework]。

软件工厂的另一个问题在于，它们可能需要庞大的组织才能被有效地使用。据一位曾经使用过产品线架构的朋友说，这样的组织需要有数千名员工，仅有50人或100人是不够的，原因在于大量投资和开销成本。他还说（而且我也从其他人那里听到过相同说法），即使在那些已经开始使用产品线架构的组织中，也不一定会一直使用和自动使用。它所带来的开销和官僚主义也不容低估。可以考虑一下曾在几个应用程序中使用过的小框架，然后设想一下大得多的框架，你就会理解这种说法了。

也就是说，我坚信软件工厂是一个有意义的计划。

软件工厂的核心是领域特定语言（Domain Specific Language, DSL）[Fowler LW]。一个通俗的描述是这样的：DSL是使用专门针对当前任务的语言来处理子问题。语言本身可以是图形化或文本化的，也可以是通用语言，例如XML、UML和C#，或特定语言，例如VS.NET中的WinForms编辑器，甚至是为自己的特定任务定义的一个小语言。

另一种方法是模型驱动架构，但它与DSL有很多类似之处。

#### 4. 模型驱动架构

模型驱动架构（Model-Driven Architecture, MDA）[OMG MDA]类似于“通过用UML绘图来编程”。MDA领域的一个常见思想是创建平台独立模型（Platform Independent Model, PIM），然后可以将此模型转换为平台特定模型（Platform Specific Model, PSM），并从这里转化为可执行的形式。

仔细想想，它就像是用第三代编程语言（例如C#）来编写一个100%的全新程序一样，这是一种过度的行为。现在应该到了提高抽象水平的时候了。

很多人都曾遇到过MDA的一个问题是，它与UML的紧密耦合。这会衍生两个问题：一是当在代码与UML之间切换时会产生精度损失，二是UML作为一种通用语言，有利也有弊。

现在，我们总结一下DSL和MDA这两种方法与模型驱动开发的关系。我坚信很多人正在朝着模型驱动开发的方向前进，因此它早晚有一天会名声大震。即使在今天，使用现有的模型驱动开发工具就对我们大有帮助。

此外，我还认为DSL与MDA这两种方法非常符合领域驱动设计，特别是在聚焦于模型的思考方面。

### 1.3.2 领域驱动设计

我们已经在架构一节讨论了模型焦点和领域驱动设计（Domain-Driven Design, DDD），但这里还要从过程的角度对DDD做几点补充。当使用DDD [Evans DDD]作为过程时，大部分精力集中

在构建好的模型上，以及在软件中尽可能按照这个模型来实现。

所有重点是创建一个尽可能简单的模型，但这个模型仍然要抓住应用程序的领域重点。在开发期间，过程实际上就是开发人员和领域专家对知识进行共同消化的过程。所获取的知识被放入模型中。

### 1. 发现旧知识是捷径

当然，并非所有知识都只能从零获取。根据领域的不同，有大量知识可以从书本中获得，并且不仅限于目标领域的专门图书。实际上有几本软件图书中也包含这样的信息。直到最近，我只选择了 *Data Model Patterns* [Hay Data Model Patterns] 和 *Analysis Patterns* [Fowler Analysis Patterns] 这两本书，但我强烈建议你手头上要有一本有关架构类型模式的书，这就是 *Enterprise Patterns and MDA* [Arlow/Neustadt Archetype Patterns]。

### 2. 通过重构获取更深入的知识

我认为一个必须要重视的价值是持续评估，对于DDD也是如此。当前模型是否为最佳模型？我们应该对当前模型提出有建设性的问题，如果发现了重要的简化，那么不要害怕重构。

当发现新特性时，同样的原则也适用。经过几轮简单的重构后，新特性不仅成为可能，而且更易于实现，从而更好地融入模型中。

重构本身也可以带来更深入的知识，例如重构可以使一切豁然开朗，并带来罕见的“尤里卡时刻”<sup>①</sup>。

我们将很快再次回到重构的讨论中来，但首先讨论一些与重构密切相关的知识，即测试驱动开发。

## 1.3.3 测试驱动开发

我经常听开发人员说他们无法使用自动单元测试，原因是他们没有任何好的工具。思维比工具重要得多，尽管工具当然也提供帮助。

几年前我编写了自己的工具（参见图1-6），并用它来注册和执行存储过程、COM组件和.NET类的测试。借助于这个工具，我可以跳过带有97个按钮的表单（必须以特定顺序将这些按钮挤压到一张表单上），以执行测试。

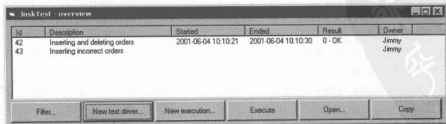


图1-6 我原来的测试工具JnskTest的屏幕截图

① “尤里卡”，Eureka，出自古希腊语，意思是：好啊！有办法啦！——译者注

后来，当NUnit [NUnit]（从其他xUnit 版本派生出来的一个版本）发布时（参见图1-7），我开始使用该工具作为替代。使用NUnit是一种更有效的方式。例如，我的工具无法反映现有测试是什么，必须显式地注册有关它们的信息。

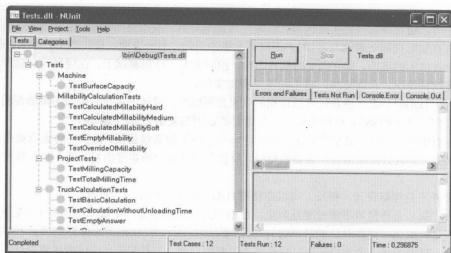


图1-7 NUnit, GUI版本

**说明** 现在，我使用另一个名为Testdriven.Net的工具，但正如我所说，使用什么工具并不重要。

无论使用什么过程，都可以使用自动单元测试。这里强烈建议你尝试一下测试驱动开发（TDD）是否适用，因为它可以实现更好的效果。

### 1. 下一个层次

TDD主张在编写实际代码之前编写测试。在这个过程中，测试将驱动设计和编程。

TDD听起来枯燥无味，而且开发人员通常认为它是一种折磨。这些观点都是大错特错的！以我的经验，事实恰好相反，它实际上会带来极大的乐趣，而且为我带来了惊喜。之所以说它带来乐趣，是因为我们在做出修改后，可以得到即时反馈，而且由我们是专业人员，因此会从创建高质量应用程序中得到乐趣。

另一种解释是，TDD不是有关测试的。它是有关编程和设计的，是有关编写更简单、更整洁、更健壮的代码的！（当然，所创建的单元测试的“副作用”也是极为重要的！）

### 2. 为什么使用TDD

我最初使用TDD的原因是想提高项目质量。质量的提高可能是最明显，也是最重要的效果，我们不想创建那些在客户初次使用时就崩溃的应用程序，或者那些当我们需要对其进行增强时却崩溃掉的应用程序。这样的应用程序是无法接受的。

TDD并不会自动帮助我们发布毫无缺陷的产品，但确实会提高质量。

**说明** 自动测试本身并不是使用TDD的主要原因，它们只是一个好的副产品。如果质量就是一切，那么还有其他的正式方法，但对于许多场景来说，它们被认为过于“昂贵”了。再一次强调，上下文是重要的。

我们可以通过在编写完实际代码之后编写测试来看到质量改进的效果。这里的意思是不必应用TDD（在实际代码之前编写测试），而只需大量的原则。另一方面，使用TDD要求写好测试。否则就会有很大的风险，即在面对时间压力时，没时间编写任何测试了，在项目的后期阶段，经常会出现这种情况。再次提醒，TDD使测试得以实行。

当应用TDD时，第二种我们可以预期的效果是可以看到设计的简化。用两句俗语说就是“简单就是美”以及“KISS”。这些简化是非常重要的，因为复杂化会产生bug。

使用TDD时，我们不必事先创建大量涵盖每个小细节的高级蓝图，而只需将重点集中在核心客户需求上，并且只需添加客户需要的材料。要更多地从客户的角度出发，而不是从技术角度出发。

TDD并不是忽略设计。相反，可以在使用TDD的整个过程中进行设计。

过去，我总是使简单事情过度复杂化。TDD帮助我集中精力，不做那些目前不需要的事情。要想达到这种效果（提高设计的简单性），就需要使用TDD。仅仅在后来编写测试是不够的。

然而，TDD的另一个效果是可以提高整个过程的效率。乍看起来，这可能违背直觉。当我们开始一个新项目时，似乎只有项目取得进展并开始编写实际代码才给人十分高效的感觉。这么做，往往开始时效率是很高的，但很常见的情况是当项目接近尾声时，效率完全降下来了。bug开始大量出现，客户要求作出一些非常重要的修改，这些修改会扰乱所有事情，而且我们发现还有一些误会……情况就是这样。

测试将迫使我们对需求提出疑问，而且是尽早提出疑问。因此，我们会尽早发现是否已理解需求，也可以尽早发现需求中的欠缺和矛盾之处。

另外，我们不应询问客户我们是否应该使用TDD，至少不应在要求更多费用/时间等一些事情的同时提出这样的问题。客户将告诉我们照做即可。当对项目做一个从头至尾的全盘考虑时，如果使用TDD不会产生额外成本，那么就on应该使用它。之后，客户将会在实现预期质量时感到高兴。

**说明** 我们先放开TDD不谈，来关注一下自动测试。

我的一位同事曾经非常怀疑自动单元测试（在实际代码之前或之后创建）的需要。他告诉我在他20年的职业生涯中，自动单元测试从未对他产生过帮助。然而，我认为最近他的想法有所改变了。我们一起做一个项目，他用C++编写COM组件，我则负责编写测试，这些测试既用作规格说明，也用作测试。当我们完成时，客户更改了一项需求。我的同事做了一个小的修改，但测试发现了一个发生概率只有4/1000的bug。在测试运行几秒钟后，就发现了这个bug，相比之下，如果手工来完成，则需要几小时。而且如果采用手工测试的话，可能根本不会发现这个bug，但在使用它就会出现。

### 3. TDD流程

现在，我已经尝试促动你开始使用TDD，那么让我们更仔细地看一下过程流程是什么样的。（我们将在第3章中再次讨论这个问题，届时将通过一个真实世界的演示进行更深入的研究。）假设我们对于需求有了适当的了解，那么流程应该是这样的。

首先，我们开始编写一个测试。然后有意地不通过测试，以便看一下测试是否符合我们的思路。这是简单且重要的一步，即使这样，我有几次还是忽略了它，结果只是招致了麻烦。

第二步是编写能够通过测试的最简单代码。

第三步是在必要情况下进行重构，因为我们可能会发现有问题的代码（例如代码重复），然后重新开始所有事情，添加另一个测试。

如果我们使用JUnit语言，那么第一步将得到红灯/红杠，第二步将得到绿灯/绿杠。

前面曾提到重构，作为一般TDD过程的第三步，这里有必要更多地解释一下这个术语。

### 1.3.4 重构

持续学习是我们在学校里时常听到的。持续学习非常适合重构的情况[Fowler R]，例如，重构可以得到更好的模型。

重构是一步一步地执行小的、众所周知的变更，以便改进现有代码的设计。即在不改变显性行为的情况下提高可维护性。换句话说，改变方式，不改变内容。

简单地讲，重构的作用是将糟糕的代码转化为良好的代码，就是这么简单。

因此，我们不必在前期就拿出一个完美的设计。这是一个好消息，因为无论如何前期设计也无法做到完美。

#### 1. 为什么使用重构

我们在识别糟糕代码方面没有任何问题。麻烦之处在于知道何时去修复它。在我看来，应该在问题出现后立即处理。我们应该使用重构，因为如果不对代码进行持续维护，那么它就会退化和崩溃。

---

说明 Mark Burhop曾说过：“好的开发人员都坚持一套软件开发法则。这套法则中包含这样一条——尘封不动的代码将滋生bug。”

---

让我们用房子作个类比。如果我们总是不去修理窗户、修缮屋顶、粉刷墙壁，那么问题将随时间而变得严重。这是一条不变的法则。因此房屋早晚会坍塌，到那个时候，它就毫无价值了。没有人想看到这种情况，不是吗？

软件是不同的，因为它不是用有形材料构建的，而且不会受到天气和大风的影响。但是在bug修复和软件扩展期间，如果不应用重构，那么我们都直观地想象出随着时间推移将会发生什么情况。

#### 2. 如何使用重构

在应用程序生命周期的所有阶段中，都可以使用重构，例如，在应用程序第一个版本的开发

期间。但我们假设不使用重构，而只使用前期的、传统的以设计为重点的过程（现在通常被称为Big Design Up-Front, BDUF）。我们将在初始的详细设计上花费大量时间，创建大量详细的UML图，但作为结果，我们希望开发过程非常顺利和快速。即使假设情况是这样，仍然存在代码非常糟糕这种风险。

相反，假设我们接受这样一个事实：不可能第一次就做得完全正确。在这种情况下，一种略微不同的方法是，将一些工作从初始详细设计转移到开发中（当然，所有开发也都是设计，特别是在这种情况下），并且准备好当了解更多内容后进行持续重构。我们在开发期间会学到更多东西，在我看来，这种方法会得到更高质量的代码。

因此，与其进行过多的猜测，不如进行更多的学习和验证。

---

**说明** 我对于BDUF的态度可能过激了，我的目的是引起读者对后面有关糟糕代码讨论的注意。

在前期做大量的猜测通常导致时间的浪费，因为那些毕竟只是猜测。

我的朋友Jimmy Ekbäck给出了如下评论：“BDUF可能比浪费时间更糟糕，原因就在于错误的猜测。BDUF也可能导致一系列连锁恶劣反应。”

---

### 3. 重构 + TDD = 事实

为了能够以一种安全的方式使用重构，我们必须执行全面的测试。如果不这样做，就将引入bug，或者是仅由于可维护性的缘故而不进行任何修改（原因是引入bug的风险太大了）。而当我们由于可维护性而停止修改时，代码就已经开始慢慢退化了。

---

**说明** 第3章将讨论更多有关TDD和重构的内容，重点提供一些实际例子。

---

用TDD和重构来修复bug也是一种好的思想。首先通过红色测试（red test）来暴露bug，然后解决bug，并得到绿色，然后进行重构。

#### 1.3.5 选择一种还是选择组合

我再一次肯定你会对选择哪种方式感到迷惑。例如，侧重前期设计还是TDD？

在我看来，可以将前期设计和TDD成功结合起来。例如，可以使用DDD建立一些前期架构，并在每个步骤中（包括重构）都使用TDD。然后返回到架构上，并根据已经学到的知识来修改它。然后再使用DDD，以此类推。

---

**说明** 不得不承认我经常后退到原来进行详细前期设计的习惯中。然而，以不同方式来思考问题通常是最有效的办法。使用一些自顶向下方法，再使用一些自底向上方法。使用一些由内向外方法，再使用一些由外向内方法。

---

预先做大量设计（Big Design Up-Front, BDUF）存在一些重大问题，我认为这一点是众所周知的。同时，我们从一开始就知道一些问题。这就是平衡问题。



最后，有关DDD和TDD的最后一个要点是：领域模型非常适合TDD。当然，我们也可以在面对数据库的设计中应用TDD，但当使用领域模型时，我一直未能适当而有效地应用它。

**说明** 当我与Eric Evans讨论TDD和/或DDD时，他说了以下一段我认为抓住了重点的话：“就我自己而言，我实际上是在编写测试的同时来处理模型。编写测试使我能够看到不同的责任分配将产生哪类客户端代码，以及方法名称等内容的细微调整对于传达意图和得到好的流程来说有什么效果。”

无论侧重TDD还是BDUF，都有大量有用的技术。在本章即将结束之际，我们将关注几种这样的技术，例如操作方面，但第一个示例是持续集成。

## 1.4 持续集成

我们都知道，每月做一个完美手工集成的工作量有多大，即使团队只由两名开发人员组成。当增加更多开发人员时，问题将呈指数级增长。它会带来各种问题，例如浪费时间，易犯错误，而且我们会发现它不会按照我们的计划进行。项目中最常说的一句话就是“再多给一天时间”。我的朋友Claudio Perrone描述了此问题的流行解决方案，称为持续集成。

### ◆ 集成问题

作者：Claudio Perrone

不同开发人员创建和修改的所有组件迟早都要构建和装配到一起，以形成一个独立系统。过去，我偶尔目睹过超长的项目延迟，它们是由于最后的集成工作引起的，在开发周期的最后阶段发现了未预料到的缺陷。

生成（build）工作可能由于多种原因而失败，通常各种原因交织在一起，例如

- 未经充分测试的组件。
- 在异常处理、null值、参数、全局变量等方面错误的（但通常似是而非的）假设。
- 脆弱的设计。
- 在不同平台上无法预料的行为。
- 发布与调试生成（debug build）之间的无法预料的差异。
- 混淆问题。
- 安装和许可问题。
- 底层架构中的bug。

注意，即使尽了最大努力，也只能通过开发人员的原则和沟通能力来控制其中的一部分问题。现实情况是，如果团队不经常进行集成，而且系统中新的类非常多的话，那么层出不穷的bug将令我们疲于奔命，这种不愉快的场景通常被称为“集成地狱”。

### 1.4.1 解决方案（或至少是正确方向上的一大步）

显著减少集成问题的关键在于自动执行生成工作，并使用增量集成，至少每天（甚至连续不

断地)对所有代码进行重新生成和测试。这里的思想是,如果最近添加的代码打断了生成工作,那么要么立即修复它,要么回滚变更,使系统恢复到上一个已知的最佳状态。

构成持续集成系统的基本部分如下所示。

- 专用于生成过程的机器。
- 充当所有源代码中央存储库的源代码控制系统。
- 监视服务,负责检查源代码控制系统对源代码的更改。
- 脚本引擎,当由前一个服务触发时,它能够创建生成。
- 报告系统,可以提供有关生成结果的即时反馈。

有几种可用的集成产品可能值得我们关注一下。目前,我最喜欢的是CruiseControl.NET [CC.NET],我将其与NAnt一起使用,NAnt是很流行的基于XML的生成引擎。这两种工具都是开源的。配置CruiseControl.NET需要花费大量工作,但一旦配置好并运行起来,它就可以在每次代码库发生变更时调用NAnt脚本。它使用多种客户端模块来通知当前所有生成的进度和状态,这些模块包括一个小的应用程序,它使用Windows托盘图标来显示汇总信息一览,通过彩色编码和气球通知的方式。

### 1.4.2 从我的组织汲取的教训

当我们第一次在InnerWorkings集中讨论实现持续集成系统的可能性时,我们当中的大多数人认为它确实是一个伟大的创意。尽管我们已经处在紧迫的进度压力之下,但我们知道确实值得投入几天时间来实现这个系统。如果让我现在思考这件事,那么我们当时肯定低估了这样的系统可能对产品质量和团队信心产生的深远影响。

目前,我们大概有500种持续集成解决方案,而且这个数字仍在增加。三分之一的解决方案共享一个自定义构建的公共框架,此框架也是集成的。所有这些解决方案的集成步骤包括编译、模糊处理(obfuscation)、打包、测试以及使用不同配置和平台进行部署。

在开始这个项目之前,有人说不可能用持续集成方式将所有这些解决方案集成到一起。然而,我却相信这是可能的,而且持续集成是成功的关键因素。

### 1.4.3 更多信息

想了解更多有关持续集成知识,可以去看Martin Fowler 和Matthew Foemmel所撰写的文章“Continuous Integration” [Fowler/Foemmel CI]。

感谢Claudio! 现在让我们来讨论一些操作方面的内容。

## 1.5 不要忘记运行机制

不久以前,我在瑞典的一家大公司中与一支团队进行了交谈。当时我谈到了Valhalla框架,以及这个框架如何对待特殊的时间点。

他们问我建立了哪些运行机制,例如日志、配置、安全性等方面的机制。当我告诉他们我们尚未考虑这些机制时,他们先是愣了一下,继而大笑起来。他们说已经花费若干年在他们自己的

框架中建立这些机制了，而我们甚至却未开始考虑这些事情。

幸运的是，我可以在某种程度上进行自辩。我们已经在运行问题上思考很多，但我们打算在建立运行机制之前先设定框架的核心部分。毕竟，核心部分影响最后的机制。我还为他们推荐了我的上一本书[Nilsson NED]，这本书的前面章节讨论了大量有关这样的机制的问题（例如跟踪、日志和配置）。

### 1.5.1 有关何时需要运行机制的一个例子

为什么运行机制很重要呢？让我们举个例子。假设一个正在使用中的应用程序缺少跟踪机制。（这并非只是虚构。我知道这个运行问题经常被忘记。尽管最近这几年来我自己谈到这个问题时很生气，但有些我自己的一些原有应用程序也是在没有跟踪机制的情况下使用的。）当一个奇怪的问题发生时，如果在错误日志中并未明确暴露其原因，那么问题的原因就很难找到，问题也很难解决。

#### 1. 未实施跟踪

任何时候都可以在特殊的时间点上添加跟踪机制，但整个过程可能至少要花费几天时间。如果问题很严重，那么客户将会希望我们在更短时间内找到并解决问题，而不希望花费几天时间。一个常见（也是最低效的）方法是做出临时性变更，并在每次变更后寄希望于问题会消失。

我们可能会采取的替代做法是到处添加一些临时性跟踪。这将使得代码变更糟糕，而且要跟踪到问题，也需要花一定的时间。下一次又出现另一个问题时，情况又是一样。我们将不得不从头开始。

另一个可能的做法是在生产环境中运行调试器。然而，这种方法也存在问题，例如可能对其他系统造成过多干涉，或者是所使用的工具使代码变得复杂而难以调试。

对生产环境中的代码进行更改也是有风险的，即使对于只是添加跟踪这样的小更改。虽然这并不是巨大的风险，但的确是存在风险的。

---

**说明** 如果有机会使用面向方面编程（Aspect-Oriented Programming, AOP），那么可能只需几分钟时间即可添加跟踪，我们将在第10章讨论很多有关AOP的内容。

---

#### 2. 已实施跟踪

如果已经有了一个正在工作的跟踪解决方案，那么就会知道它在查找和解决问题方面的效率如何。跟踪问题可能不再需要花费几天时间，而只需几分钟即可。

因此，保持谨慎是很重要的，而且当涉及操作机制时，不要总是认为“你将不需要它”（YAGNI）。如果需要添加机制，那么采用YAGNI思想通常会产生很大代价。记住，YAGNI思想适用于添加某物的现在和将来代价很高的情况，在这种情况下，我们可以一直等到真正需要时再添加。当代价现在很低而将来很高时，而且很可能需要它时，那么就要做出不同的决定了。

### 1.5.2 运行机制的一些例子

以下是一个简短的运行机制列表，它适用于大多数企业级应用程序。

- 跟踪

正如我们刚刚讨论过的，能够在用户运行系统的同时侦听所发生的事情是很有帮助的。这不仅是非常有效的bug跟踪解决方案，而且还可用于调查哪里存在瓶颈。

- 日志

错误、警告和信息消息必须被记录下来。这对于调查已发生的问题来说尤其重要。我们不能寄希望于用户为我们记下这些东西。还有一个原因就是，我们可能希望收集一些不显示给用户的信息。

- 配置

你是否遇到过这样的情况：仅仅因为数据库服务器以新的名称迁移到新机器上，而不得不重新编译旧的应用程序？我就曾遇到过。当然，这类信息应该是可配置的且取决于应用程序，大量信息都属于此类情况。

- 性能监视

对于问题跟踪来说，基于领域模型信息和应用程序其他部分信息对性能进行监视会有极大帮助，同时还可以对系统保持密切关注。这样，我们就很容易跟踪它了，与两个月之前执行某一场景的时间相比，现在可能只需30%的时间就可以了。

- 安全性

近来，安全性可能不再需要任何进一步的解释。我们显然需要慎重思考身份验证和授权等方面。我们还必须针对各种类型的攻击对应用程序进行保护。

- 审计

作为安全方面的一个部分，审计是很重要的，以便可以事后检查谁在何时都做了什么。

### 1.5.3 它不仅仅是我们的过错

在开发人员的讨论中，我曾经有几次问过运行人员他们对于运行机制的需求，但他们说得并不多。我猜想他们并未希望从应用程序那里获得大量支持。

也就是说，如果可能的话，处理此问题的巧妙方式是，在早期就调配一些负责运行的人员，并让他们参与到系统中，进而可以创建真正有用的运行机制。在这里，系统的普通客户不是好的需求创建者。运行机制是非功能性需求的典型示例，普通客户一般对此不会有太大贡献。

机制的灵活性可能很重要，因为不同客户使用不同的运行平台。有一些诸如Windows Management Instrumentation (WMI) 这样的标准，但如果为其构建框架的话，最好能保持一定的灵活性，以便容易切换到不同的日志输出格式。一位客户可能使用CA Unicenter，另一位客户可能使用Microsoft Operations Manager (MOM)，而又一位客户可能使用某些不理解WMI的产品，等等。

## 1.6 小结

我们以运行方面的简短讨论结束了本章。本书后面不再讨论有关运行方面的内容。重点将集中在应用程序的核心上，即核心业务价值。

对于“现代软件开发中应该重视什么”这个问题，本章的讨论当然并不详尽。我的希望是简要讨论每位开发人员都值得思考的几个观点。在这个过程中，我们引入了领域驱动设计、领域模型、测试驱动开发、模式等概念，这些概念既是关于架构的，也是关于过程的。

我想再次强调的三个观点是平衡、随需应变和持续学习。对于开发人员、架构师以及我们每个人来说，这些都具有宝贵的价值。

那么，在布景搭建好之后，现在到了讨论更多不同类型模式知识的时候了。



**我**们经常会遇到新的设计问题。一般情况下，我们都能够解决问题，但有时却身陷困境。有时发现解决方案有严重缺陷，有时创建了拙劣的解决方案。通过重用好的、经过充分检验且灵活的解决方案，可以最小化这些风险，并更快地得到所需结果。实现此目的的一种工具是模式。借助于模式的较高抽象级别，还可以发现并成功解决更大的设计问题。考虑模式还有助于开发人员之间的更好理解，因为系统开发与沟通有着很大的关系。

我曾多次听过有关模式的论调，说模式没有技术意义，是无用的并且夸大其辞。如果你也抱着这种观点，那么本章的目的就是扭转这一点，因为错误之大，莫过于此。模式具有很大的实际意义，它在日常工作中极为有用，而且令所有（或至少是大部分）开发人员感兴趣。也许你并未注意到，但本书已经讨论过几次有关模式的内容。一个例子就是第1章引入的领域模型模式[Fowler PoEAA]。在本章中，我们将讨论三种不同类别的模式：设计模式（通用设计模式和应用类型的具体设计模式）、架构模式和领域模式。

---

**说明** 注意，这里的类别比较模糊，而且与模式本身相比，类别根本不重要，也不会引起我们的兴趣。因此，如果类别没有为你提供帮助，就不要让它妨碍你对模式的理解。

---

即使你已经对模式有所了解，也会发现本章是很有趣的。这里不会重复一些已有的解释，而是提出我自己对模式的理解。例如，本章的讨论将以领域模型为中心，而设计模式通常不会这样考虑。此外，我还希望本章多处提到的反射会引起你的兴趣。

在开始之前，我们先简要讨论一下模式的概念，以及为什么应该学习模式。

## 2.1 模式概述

模式提供了用于解决反复发生的设计问题的简单、雅致的解决方案。模式提供的主要优点是灵活性、模块性以及创建可理解的、清晰的设计。注意我忽略了可重用性，尽管这有些不公平。模式将重点从代码重用转移到知识重用上。因此模式在很大程度上也是有关可重用性的，只是不是我们通常所考虑的重用。

---

**说明** Gregory Young曾指出很多模式是有关重用的，方式是通过解耦合。依赖注入（Dependency Injection）模式就是一个好的例子（将在第10章中讨论）。

---

当我们研究模式时，你可能会想：“这不就是我们一直做的那样吗？”模式的一个重点在于，它们不是发明的，而是归纳或提取的。它是有关经过检验的解决方案的。但解决方案并不是模式的全部。模式可以用三个部分来描述：上下文、问题和解决方案。

从我们自己的错误中学习是一种极为有效的实践，但有时从别人积累的知识中学习也是一种很好的捷径，这是学习模式的一个很好的原因。让我们来看一下是否还有其他原因。

### 2.1.1 为什么要学习模式

最明显的原因可能在于模式是很好的抽象，它们提供了系统设计的构建块。

如果开发团队很关注模式，那么模式将成为语言中的非常重要的部分。通常不必描述每种设计思想的具体细节，而只需说出模式名称就足够了，然后团队中的每位成员就可以评估它对于该特定问题是否是好的思想。在团队语言中添加模式可能是使用模式的最重要原因，因为语言的公共理解、丰富性和表达性都得到了提高。再次强调，开发与沟通有着很大的关系。

我喜欢模式的另一个原因是模式的使用能力是一个长期技巧。作为一个参照，我大约是在1988年开始学习SQL的，而且我现在仍然可以把使用SQL作为一种很好的谋生手段。我所使用的产品和平台几经变迁，虽然基本概念并未改变。模式也是类似的。《设计模式》一书[GoF Design Patterns]是在1995年出版的，但它在今天仍然具有极大的参考价值。另外值得一提的就是模式与语言、产品和平台都是无关的。（不同平台可能具有对某些特定实现变体的特定支持，[GoF Design Patterns]就属于这种情况，它是将面向对象作为假设而编写的。）

如果你学习过《设计模式》[GoF Design Patterns]，那么将会发现模式与好的面向对象设计原则是一致的。你可能会问，什么是好的面向对象设计？Robert C. Martin在《敏捷软件开发：原则、模式与实践》[Martin PPP]一书中讨论了一些这样的原则。例子包括单一职责原则（SRP）、开放-封闭原则（OCP）和里氏替换原则（LSP）。

#### 关于Martin原则的更多知识……

下面我们对Martin原则做一下更多解释。

##### □ 单一职责原则（SRP）

项（例如类）应该只有一个职责，并很好地履行该职责。如果一个类既负责呈现又负责数据存取，那么它就是违背SRP原则的很好示例。

##### □ 开放-封闭原则（OCP）

类应该对修改关闭，但对扩展开放。当更改类时，通常要冒着破坏某些东西的风险。但如果不改类，而是用于类对它进行扩展，那么就是风险较小的更改方案。

##### □ 里氏替换原则（LSP）

假设有一个继承层次结构，它由Person和Student构成。无论何时我们可以使用Person，也应该可以使用Student，因为Student是Person的子类。乍看起来，这可能是顺理成章的，但当我们考虑反射时（反射是一种通过编程方式来检查实例类型的技术，它可以读取并设置属性和字段，并在不必预先知道类型的情况下调用其方法），它就不是这样明显了。使用反射来

处理Person的方法可能不需要Student。

反射问题是一个句法问题。Martin使用了一个更具有语义含义的例子：Square是一种Rectangle（正方形是一种矩形）。但是，当我们对Square使用SetWidth()时是没有意义的，至少必须要在内部调用SetHeight()。而Rectangle则完全不需要这样做。

当然，所有这些原则在特定上下文中是存在争议的。它们应该仅用作指南，而不是“唯一真理”。例如，OCP可能很容易被过度使用。我们可能会遇到这样的情况：对如何实现方法有了更好的理解，并希望修改它，而不是扩展它。而且为类添加方法也可以看作是一种扩展。

模式不仅非常有利于前期设计，而且在重构期间也非常有用（或许更为有用）……“我的代码变得越来越混乱。哦，我需要使用那种模式！”这是一个类似于鸡生蛋还是蛋生鸡的问题，但我决定在本书中先从模式开始，然后再讨论重构（见第3章）。

### 2.1.2 在模式方面要注意哪些事情

诚然，我找不出太多不学习模式的理由，但至少有一个常见的负面效果需要注意。我所想到的是，初学模式的开发人员经常乐于将17种模式挤压到每一个解决方案中。通常这种效果并不会持续太久。

可能持续更久的是过度设计的风险。如果不是17种模式的话，至少可以考虑很多种解决问题的方式。最初的解决方案给人的感觉是不正确的，因为它没有使用特定模式。

**说明** 我的一位朋友告诉我最近与某公司开发人员讨论的一个设计问题。他只用3分钟时间就想出了一个非常简单的解决方案。（问题本身也非常简单。）然而，另一位开发人员对此解决方案感到不满意，因此他们又花了三天的思考，才就此达成一致。

这种事情是很难避免的。在我的观点中，TDD是一种好的技术，它可以避免过度设计问题。设计重点大部分集中在解决当前问题上，当必要时通过重构来引入模式。

你可能会感觉模式概念就像“银弹”。它当然不是，而只是工具箱中的另一个工具而已。

模式本身可能给人感觉很“简单”，但在上下文中以及与其他模式组合使用时，它们变得非常复杂。我不记得有多少次听新闻组上的人们这样说：“我理解模式X，但当我尝试在应用程序中与模式Y和Z一起使用时，它却变得非常复杂。请帮我！”这实际上并不能成为放弃学习模式的原因。本书将在某种程度上解决这一问题，但不是现在。我们首先讨论一些单独的模式。

我相信Joshua Kerievsky的《重构与模式》[Kerievsky R2P]一书直指要点，书中反复强调，最常使用的模式是在前期设计中不使用模式，而是通过重构来形成模式。

#### 采用模式

Gregg Irwin对采用模式的建议是：

“对我来说，很多概念（例如模式）的学习阶段是这样的。

(1) 使用它，但未意识到正在使用它。



- (2) 听说它，阅读了一些有关它的知识，并开始尝试它。
- (3) 了解更多，开始明确地使用它（如果是不成熟的话）。
- (4) 开始热衷并传播它（可选）。
- (5) 突然有所领悟。
- (6) 学到更多，并“更成熟地”、更含蓄地应用它。
- (7) 随着时间的流逝，看到了缺点。
- (8) 对概念提出疑问（通常是因为错误地应用了它）。
- (9) 要么忘记了它，要么是增加了知识或经验 [必要时重复步骤(5)~(9)]。
- (10) 使用它，但未意识到正在使用它。

现在该到讨论第一种模式类别的时候了。让我们来看一下作为模式的陌生者，你是否能够理解它（而不仅仅是感到疑问）。

## 2.2 设计模式

当我在这里提到设计模式时，第一个想到的就是《设计模式》[GoF Design Patterns]一书，到现在为止，我们已经多次提到过这本书。它并不是有关设计模式的唯一书籍，但一般被奉为是这一主题的圭臬。

设计模式是抽象的和相当低级别的，它们是关于技术的，而且通常与领域有关。它通常与正在构建哪个层或正在构建什么类型的系统无关。设计模式仍然是有用的。

设计模式的一种描述方式是：它们是关于子系统或组件精化的。当我们讨论到其他两种模式类别时（下面即将会讨论），你将发现在那两个类别的模式中，使用一种或多种设计模式是很常见的，或者是可以以更具体的方式来应用一些具体设计模式。

---

**说明** 当谈到低级别时，这里有一个有趣的小故事。我曾被问及我与一位专业级朋友之间有什么区别。我回答说我的朋友使用低级别的编程，而我使用高级别的编程。问我这个问题的人丝毫不懂编程，但当他听到我的“自吹自擂”后，和你一样心神领会了。

---

《设计模式》[GoF Design Patterns]是一本深奥的书。每次我读这本书，都会理解和学到更多东西。例如，我经常想“那不正确”或“那不是最佳解决方案”或甚至“那是愚蠢的”。但经过一番深思熟虑后，我每次都确定它们是“正确的”。

到目前为止，我们进行了很多讨论，但还没有实践。现在我们通过解释设计模式来使它变得具体。我选择我所喜欢的一种设计模式——状态，下面就来讨论它。

### 示例：状态模式

基于问题的教学是一种好的教学方法，因此这里将采用它。下面是一个问题。

#### 1. 问题

销售订单具有不同的状态，例如“NewOrder”、“Registered”、“Granted”、“Shipped”、

“Invoiced”和“Cancelled”。对于订单可能从什么状态“进入”到什么状态，有着严格的规则。例如，直接从Registered状态进入Shipped状态是不允许的。

取决于状态的不同，行为上也有区别。例如，当状态为Cancelled时，就无法调用AddOrderLine()来为该订单添加更多项。（这条规则也适用于Shipped和Invoiced状态。）

要记住的另一件事是，特定行为将导致状态的转换。例如，当AddOrderLine()被调用时，状态将从Granted转换回New Order。

## 2. 解决方案提议1

为了解决这个问题，需要在代码中描述状态图。在图2-1中，你将发现一个非常简单和经典的状态图，它描述了每次用户按下按钮时，按钮是如何在Up和Down状态之间变换的。

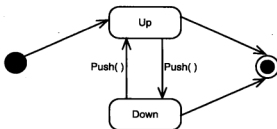


图2-1 按钮的状态图

如果将此技术应用于订单的状态图，那么就可以得到图2-2所示的状态。

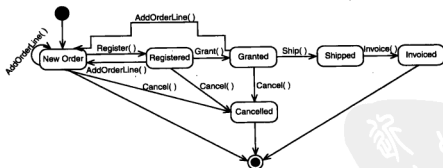


图2-2 订单的状态图

一个明显的解决方案是使用类似于以下的enum:

```

public enum OrderState
{
    NewOrder,
    Registered,
    Granted,

```

```

        Shipped,
        Invoiced,
        Cancelled
    }
}

```

然后可以像下面这样将一个私有字段用于Order类中的当前状态:

```
private OrderState _currentState = OrderState.NewOrder;
```

然后,除了方法应该做什么以外,还需要在方法中处理两件事情。一是必须检查在该状态下方法是否可以被调用,二是考虑是否需要进行状态转换,如果需要,应转换到什么新状态。在AddOrderLine()方法中,可能会像下面这样:

```

private void AddOrderLine(OrderLine orderLine)
{
    if (_currentState == OrderState.Registered || _currentState == OrderState.Granted)
        _currentState = OrderState.NewOrder;
    else if (_currentState == OrderState.NewOrder)
        //Don't do any transition.
    else
        throw new ApplicationException(...

    //Do the interesting stuff...
}

```

正如我们在代码段中看到的,方法中添加了很多枯燥的代码,仅仅是因为要顾及状态图。糟糕的if状态在将来修改时极易出错。Order类中将遍布这类代码。我们所做的只是将状态图的知识传播到几个不同的方法中。这是隐晦且不良的代码重复的一个典型例子。

即使对于这样的简单示例来说,我们也应该减少代码重复和分隔。让我们来尝试修改一下。

### 3. 解决方案提议2

提议2只做了略微变化。我们可以有一个名为\_ChangeState()的私有方法,它可以从AddOrderLine()调用,\_ChangeState()可以有一个长的分支语句,像下面这样:

```

private void _ChangeState(OrderState newState)
{
    if (newState == _currentState)
        return; //Assume a transition to itself is not an error.

    switch (_currentState)
    {
        case OrderState.NewOrder:
            switch (newState)
            {
                case OrderState.Registered:
                case OrderState.Cancelled:
                    _currentState = newState;
            }
        }
    }
}

```

```

        Break;

    default:
        throw new ApplicationException(...
        break;
    }
    case OrderState.Registered:
        switch (newState)
        {
            case OrderState.NewOrder:
            case OrderState.Granted:
            case OrderState.Cancelled:
                _currentState = newState;
                break;

            default:
                throw new ApplicationException(...
                break;
        }
        ...
        //And so on...
    }
}

```

AddOrderLine() 现在就像下面这样:

```

public void AddOrderLine(OrderLine orderLine)
{
    _changeState(OrderState.NewOrder);

    //Do the interesting stuff...
}

```

在上面这段代码中我有些过于懒惰了, 因为只显示了这个巨大分支语句结构的开始部分, 但我认为仍然可以明显看出这是不良代码的一个典型示例, 特别是因为这段代码是简化后的, 而且并未讨论必需的所有方面或所有状态, 甚至代码尚未结束。

**说明** 通常, 对于某些情况来说, 以上示例足够好并且是“正确的”解决方案。我这样说的目的是为了强调一个事实: 对于问题来说, 并不存在唯一的、总是正确的解决方案。还要注意的, 下一章将讨论更多有关不良代码的问题。

我在几个项目中使用过这样的解决方案, 但我并不是非常喜欢它。它开始看起来非常好, 但当问题增加时, 这种解决方案就会带来麻烦。让我们来看另一种解决方案。

#### 4. 解决方案提议3

第三种解决方案基于一个表 (某种配置信息), 它描述了在特定激励下, 应该发生什么情况。因此这次我们描述表中的转换, 而不是提议1和提议2的代码中的状态转换, 如表2-1所示。

表2-1 状态转换

当前状态	允许的新状态
NewOrder	Registered
NewOrder	Cancelled
Registered	NewOrder
Registered	Granted
Registered	Cancelled
...	...

这样,当当前状态为NewOrder时, `_ChangeState()` 方法就只能够检查作为参数出现的新状态是否是可接受的。如果当前状态为NewOrder,那么只有Registered和Cancelled是允许的新状态。我们还应该添加另一个列,如表2-2所示。

表2-2 修订后的状态转换

当前状态	方 法	新 状 态
NewOrder	Register()	Registered
NewOrder	Cancel()	Cancelled
Registered	AddOrderLine()	NewOrder
Registered	Grant()	Granted
Registered	Cancel()	Cancelled
...	...	...

现在, `_ChangeState()` 方法不应该再以参数的形式(而是以方法名称的形式)来获取新状态。这样 `_ChangeState()` 通过在表中查找来决定新状态应该是什么。

这种方法既整洁又简单。这里的一大优点是很容易获取可能的不同状态转换的概览。主要问题在于:很难处理那些取决于当前状态的行为,然后当某个方法执行时进入几个可能状态中的一个。当然,它并不比提议2困难,但仍然不够好。我们可以在表中注册有关在特定转换时应执行什么委托的信息(委托就像强类型的函数指针),也可以对这个思路进行扩展,以解决其他问题,但我认为这具有一定的风险,因为在调试期间它会变得有些混乱。

我们是否还有更多思路?让我们对一些知识进行重用,并对设计模式中的状态模式进行实验。

#### 5. 解决方案提议4

状态模式的一般结构如图2-3所示。

思路是将不同状态作为单独的类封装起来(参见ConcreteStateA和ConcreteStateB)。这些具体的状态类从抽象的State类继承。Context有状态实例(作为字段),并且当Context收到Request()调用时,就调用状态实例的Handle()。Handle()对于不同状态类具有不同的实现。

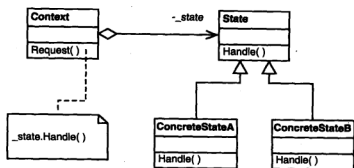


图2-3 状态模式的一般结构

这是一般的结构。让我们看一下将它应用于目前问题时，将会是什么情况。图2-4所示是一个具体示例的UML图。

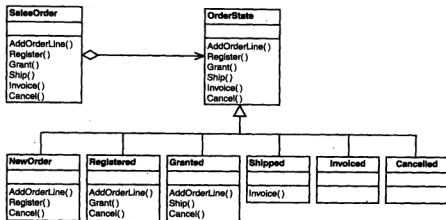


图2-4 状态模式的具体示例

**说明** 对于这个示例来说，添加另一个抽象类（将其作为NewOrder、Registered和Granted的基类）可能是有意义的。新的类将实现AddOrderLine()和Cancel()。

在这个具体示例中，Order类是来自一般结构的Context。再一次，Order具有OrderState的一个字段，尽管这次OrderState不是枚举，而是类。由于重构的缘故，原有测试可能需要枚举，这样我们就可以保持该枚举（或许作为属性，其实现检查状态继承层次结构中的当前实例是什么），从而不用对外部接口进行修改。

新创建的Order在实例化时获得NewOrder的新的状态实例，并将它自己发送给构造方法。像下面这样：

```
internal OrderState _currentState = new NewOrder(this);
```

注意，字段被声明为内部的，这样状态类就可以通过它自身来更改当前状态，Order就可以完全将状态转换委托给不同的状态类。（也可以令OrderState成为Order的内部类，这样就不需要internal语句了。）

这次，Order上的Register()方法就非常简单了。它可能像下面这样：

```
public void Register()
{
    _currentState.Register();
}
```

NewOrder上的Register()方法也非常简单。至少它可以关注其自己的状态，这使得代码整洁且清晰。可能像下面这样：

```
public void Register()
{
    _parent._Register();
    _parent._currentState = new Registered(_parent);
}
```

在更改状态之前，有一个对父（\_parent.\_Register()）的回调，通知它在状态改变之前要做的事情。（注意，“回调”的是内部方法\_Register()，而不是公共的Register()。）当然，这只是选项的一个示例。其他示例将把代码放在OrderState基类中，或放在NewOrder类本身中，具体放在哪里取决于哪个类是最佳位置。

正如我们所看到的，如果需要在状态转换之前或之后做一些事情，那么这是很简单的，而且是被良好封装的。如果要禁用NewOrder类中的特定转换，只需跳过该方法的实现，并对该类使用基类OrderState的实现即可。基类的实现抛出异常，指示它是非法的状态转换（如果那就是所需的行为的话）。另一种典型的默认实现是不做任何事情。

---

**说明** 如果需要更多上下文感知的异常，当然也可以实现子类中的方法，即使它们所做的所有事情只是引发异常。

这也暗示着我们可以不必使用OrderState的基类，而使用接口来替代。我猜想如果GoF的书是今天才编写的，那么很多模式将使用接口，而不使用抽象基类（或至少是一起使用这两者）。状态模式不是这方面的最典型示例，但它仍然是可能的示例。

---

## 6. 应用提示

当使用状态模式时，我们实际上是将一个单一字段交换到一堆单独的类中。这乍听上去似乎不是一个好思想，但我们会得到这样一个很好的结果：将行为移动到它应该属于的地方，并且也非常符合单一职责原则（SRP）。

当然也有缺点，一个典型的缺点就是当我们使用诸如状态这样的解决方案时，程序可能大量充斥较小的类。

到底哪种解决方案是首选确实易引起争议，但我们在这里应该认真考虑状态模式。你可能会发现它能够以最小量的代码重复解决问题，而且它将责任划分为封装的、紧密结合的单元，即具体的状态类。但要注意的是，模式类也很容易被过度使用，像每种工具那样。要明智地使用它！

这就是设计模式的一个示例，它是很普通的示例。后面我们将再次讨论另一个家族的更多设计模式，但首先讨论另一个模式类别。

## 2.3 架构模式

一谈到“架构模式”这个术语，我们经常会把它与Buschmann等人的书《面向模式的软件架构》[POSA 1]中所讨论的一些模式联系起来。在该书中，有几个模式被归入架构模式的类别下。管道与过滤器模式（Pipes and Filters）和反射模式（Reflection）就是两个例子。

管道与过滤器模式是有关通过管道来传送数据并在过滤器中处理数据流。此模式被SOA社区选为有用的模式，用于基于消息的系统。

反射模式同时被构建到Java和.NET中，它允许仅使用对象的元数据来编写那些从对象读取并向对象写入数据的程序，而不必预先知道有关对象类型的任何信息。

如果说设计模式是关于精化子系统和组件的，那么架构模式就是关于建立子系统结构的。我们通过一个常见示例来具体说明这一点：层 [POSA 1]。

### 2.3.1 示例：层

层或分层是基本的架构原则，它是指将职责划分为独立的、紧密结合的单元（类的簇），并定义这些单元之间的依赖性。大多数开发人员对此都应该很熟悉。

由于它是一个非常常用且已经被很好理解的模式。因此这里我们只给出一个简单示例，对此模式类别有所了解即可。

#### 1. 问题

假设我们为SalesOrder应用程序构建了一组类，例如Customer、Order、Product，等等。这些类封装了它们对于领域的意义，以及它们是如何实现持久化/非持久化和如何呈现的。当前实现是对于文件系统实现持久化，并将对象呈现为HTML片段。

不巧的是，我们现在发现必须将对象作为XML使用，并使用关系数据库来实现持久化。

#### 2. 解决方案提议1：应用层

此问题最常见的解决方案是将类的一些职责分离出来。新的需求导致了一个明显的事实，即类明显破坏了SRP原则。

我们尝试将类分解，以便可以从技术角度来集中处理职责。

因此，原有的类现在将只关注领域意义（我们将其称为Domain层）。呈现（或使用）职责通过另一组类（名为Consumer的另一个层）来处理，持久化职责则由再另外一组类（Persistence层）来处理。

这里，这三个层很自然地给人感觉是一种典型的解决方案，每个层负责一类责任。两个新的层具有两个实现，分别用于处理原有需求和新需求。



各层之间的依赖性也被定义了,在此例中我们决定使用层依赖于领域层,领域层依赖于持久层。这样,Consumer层完全不知道Persistence层,我们认为在这个特定项目中,这一点是有利的。我们将在第4章中再次讨论分层这一主题,该章的讨论角度将略有不同。

### 2.3.2 另一个示例:领域模型模式

我们已经在第1章中讨论过领域模型模式[Fowler PoEAA](本书通篇还会做更多讨论)。我将领域模型模式看作是架构模式的一个示例。

这里我们准备讨论有关领域模型模式的示例,因为整本书都是有关它的。相反,我们将继续关注另一个模式维度,即领域是否具有依赖性。接下来,我们看一下针对具体应用程序类型的设计模式。

## 2.4 针对具体应用程序类型的设计模式

与到目前为止我们讨论过的设计模式相比,另一组设计模式不像它们那样通用,例如那些用于构建企业应用的模式。

定义企业应用需要一定技巧,但我们可以将它当作一个拥有很多用户和/或数据的大型信息系统。

讨论了此类模式的一本主要图书是Martin Fowler的《企业应用架构模式》[Fowler PoEAA]。

与一些设计模式相比,这里的模式最初看上去可能不如它们那样流行或有吸引力,但它们却非常实用,它们涵盖了大量基础,并包含很多经验和知识。正如前面所说,它们不如其他设计模式通用,而且仅关注大型信息系统。

它们主要用于选定的逻辑结构,例如领域模型。这里的模式并不是过多地关注领域模型本身(或用于结构化主逻辑结构的任何其他模型)应该如何被结构化,而是更多地关注用于支持领域模型的基础架构。

我们通过一个示例使它更具体,这里选择查询对象[Fowler PoEAA]。

### 示例:查询对象

现在,我们假设有一个用于SalesOrder应用程序的领域模型。有Customer类和Order类,而且Order类由很多其他类组成。这是一种简单且清晰的结构。

要在领域模型中导航,有几种解决方案可供选择。一种解决方案是有全局根对象,它拥有对根状集合的引用。在本例中,客户集合就是这些集合中的一个示例。那么,开发人员要做的事情就是从全局根对象开始,并从这里导航至客户集合,然后对该集合进行迭代,直到找到所需的内容,或者也会导航至客户的销售订单(如果对其感兴趣的话)。

一个类似的范型是所有集合都是全局的,这样就可以直接访问客户集合,并对其进行迭代。

这些范型都很容易于理解和使用,但一个缺点是,从分布式角度来看,它们有某种程度上的欠缺。假设我们有一个运行在客户端上的领域模型(每个客户端都有一个领域模型,或者是领域模型实例的小的子集,而没有共享的领域模型实例),而数据库运行在数据库服务器上(这是一种

很常见的部署模型)。那么,当要求根对象获取100万个客户的客户集合时,将会发生什么情况?为了让客户端在集合上进行本地迭代,可能需要将所有客户都返回到客户端上。等待如此巨大的集合的转移过程也不会令人愉快。

另一个选项是在整体结构中添加应用服务器,并要求它只向客户端发送集合引用,这样要传送的数据当然就少得多了。另一方面,这会产生难以置信的网络调用量,因为客户端对列表进行迭代,并通过网络请求下一个客户,这个过程要重复100万次(如果客户实例本身不是按值编组,而是按引用编组,那么情况将更糟)。还有一个选项是对客户集合进行分页,这样客户端一次可以从服务器获取100个客户。

我知道所有这些解决方案都有一个共同的问题。我们并不希望查看所有客户,而是需要一个子集,这就引出了对下一个问题的讨论。

### 1. 问题

用户需要一种能够灵活搜索客户的表单。他们希望能够搜索满足以下条件的客户。

- ☐ 句子中包含“aa”。(针对一家瑞典汽车公司的隐性营销。)
- ☐ 上个月订购了某物。
- ☐ 具有总值大于100万的订单。
- ☐ 具有名为“Stig.”的推荐人。

但在同一张表单上,他们还应该能够搜索那些位于瑞典特定地区的客户。再次强调,搜索表单需要非常灵活。

下面将要讨论三种不同的解决方案提议,即“在领域模型内部进行过滤”、“使用巨大的参数列表在数据库中进行过滤”和“查询对象”。

### 2. 解决方案提议1: 在领域模型内部进行过滤

让我们退一步,承认可以使用已经讨论的任何一种解决方案,这样就可以在某个地方对集合进行物化,然后用过滤器来检查每个实例。满足过滤器标准的所有实例均被添加到新的集合中,此集合就是最终结果。

这是一种很简单的解决方案,但在很多真实世界解决方案中并没有实际的可用性。我们将浪费空间和时间。我们不仅有100万个客户,而且还必须对客户订单进行物化。该解决方案根本行不通,而且当问题升级时,情况会更糟……

当然,这里的结论在很大程度上取决于执行平台。记住前面曾讲过的每个客户端中领域模型实例的部署模型子集,数据库位于数据库服务器上,没有共享的领域模型实例。

如果相反,应用服务器上有一个共享的领域模型实例的子集(这也有其自己的问题,本章后面将讨论到),那么这可能是适当的解决方案,但仅对服务器端逻辑有效。请求共享领域模型实际子集的客户必须以一种方式表示出它们的标准。

### 3. 解决方案提议2: 使用巨大的参数列表在数据库中进行过滤

数据库通常很擅长存储和查询,因此我们充分利用它们的优势。我们只需用SQL语句来表示出我们的需要,然后将结果转换为领域模型中的实例即可。

以下SQL语句可以解决第一个问题:

```

SELECT Id, CustomerName, ...
FROM Customers
WHERE CustomerName LIKE '%aa%'
AND Id IN
    (SELECT CustomerId
     FROM ReferencePersons
     WHERE FirstName = 'Stig')
AND Id IN
    (SELECT CustomerId
     FROM Orders
     WHERE TotalAmount > 1000000)
AND Id IN
    (SELECT CustomerId
     FROM Orders
     WHERE OrderDate BETWEEN '20040601' AND '20040630')

```

**说明** 一个有争议的地方是：我是否能够将针对Orders的两个子选择合并为一个。正如需求所表示的，我认为不能合并（因为如果合并它们，意义将稍微改变）。不管怎样，这实际上对于这里的讨论并不重要。

这里我们只对感兴趣的实例进行了物化。然而，我们可能不希望包含领域模型的层必须包含这段SQL代码的全部。在该例中，领域模型的重点是什么？使用层只需要处理两个模型。

现在我们有了一个新问题。使用层应该如何表示它想要表示的东西？负责数据库与领域模型之间映射的领域层可以为使用层提供搜索方法。提议2的代码如下：

```

public IList SearchForCustomers
(string customerNameWithWildCards
, bool mustHaveOrderedSomethingLastMonth
, int minimumOrderAmount
, string firstNameOfAtLeastOneReferencePerson)

```

这可能解决了第一个查询的需求，但没有解决第二个。我们需要添加更多参数，像下面这样：

```

public IList SearchForCustomers
(string customerNameWithWildCards
, bool mustHaveOrderedSomethingLastMonth
, int minimumOrderAmount
, string firstNameOfAtLeastOneReferencePerson
, string country, string town)

```

你是否能看到结果？参数列表很快就变得没有实际意义，因为可能有大量其他参数也是必需的。当然，在调用方法时，编辑器可能有所帮助（编辑器显示了每个参数的占位符），但使用方法很容易出错，而且也没有实际意义。当需要另一个参数时，我们不得不返回并修改所有原来的调用，或者至少需要提供新的重载。

另一个问题是如何用参数列表中的基本数据类型来表示特定事情，因为这是一种相当无力的方式。一个典型示例是mustHaveOrderedSomethingLastMonth参数。在那之前的一个月是什么情况？或者上一年是什么情况？当然，我们可以使用两个日期作为参数，并将定义时间间隔的责

任移交给方法的使用者，但当我们只关心位于某一特定城镇的客户时，又将会是什么情况呢？日期参数应该是什么？我想可以使用最小和最大日期来创建可能的最大时间间隔，但我们并不能非常直观地看出这就是表示“所有日期”的方式。

**说明** Gregory Young对如何表示优先次序（presedence）这个问题有过评论。用参数列表来表示它是有问题的：(criterion1 and criterion2) or (criterion1 and criterion2)

我想我们很快也会放弃这个解决方案。最后的结论与我以前使用VB6时的结论是一样的，因此我使用基于数组的解决方案。数组的第一列是字段名（例如CustomerName），第二列是操作员（例如人口普查员），第三列是判断标准，例如“\*aa\*”。每个判断标准在数组中占有一行。

该解决方案可以通过参数列表来解决一些问题，但它也存在自身的问题。在添加新判断标准时，不必更改任何原有的使用者代码。这是一个优点，但对于高级判断标准来说，它就无能为力了，因此这里退后一步，并公开数据库模式，以便处理这条判断标准：“是否有任何总量大于100万的订单？”然后使用完整的IN子句作为判断标准。

基于数组的解决方案向着正确方向迈出了一步，但如果用对象来替代的话，可以变得更灵活一些。不幸的是，我们实际上无法在VB6中编写按值编组的组件。这个问题有一些解决方案，例如使用一种更灵活的数组结构，但整个事情在.NET中更为自然。接下来看一下查询对象模式。

#### 4. 解决方案提议3：查询对象

查询对象的思想是封装Query实例中的判断标准，然后将该Query实例发送到另一个层，在那里，它被翻译为所需的SQL。一般解决方案的UML图可能如图2-5所示。

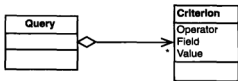


图2-5 一般查询对象解决方案的类图

判断标准（criterion）可以使用另一个查询（虽然在图2-5中对此的典型描述并不明显）。这样就更容易在SQL中创建等效的子查询。

让我们尝试用查询对象语言来解决问题。但首先假设领域模型如图2-6所示。

看一下用C#新创建的初步查询语言是什么形式的：

```

Query q = new Query("Customer");
q.AddCriterion("CustomerName", Op.Like, "**aa*");
Query sub1 = new Query("Order");
sub1.AddCriterion("TotalAmount", Op.GreaterThan, 1000000);
q.AddCriterion(sub1);
  
```

```

Query sub2 = new Query("Order");
sub2.AddCriterion("OrderDate", Op.Between,
  
```

```
DateTime.Parse("2004-06-01"), DateTime.Parse("2004-06-30"));
q.AddCriterion(sub2);

q.AddCriterion("ReferencePersons.FirstName", Op.Equal, "Stig");
```

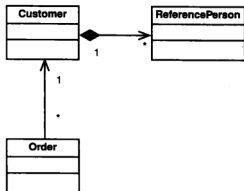


图2-6 用作示例的领域模型

**说明** Query构造方法的参数不是表名称，而是领域模型的类名称。AddCriterion()的参数也同样如此，即它不是表列，而是类字段/属性。在本例中，属性名称或字段名称被用在领域模型中。

还要注意，在这个具体示例中，有关ReferencePersons的判断标准并不需要子查询，因为领域模型可以从Customer导航至ReferencePerson。另一方面，Orders却需要子查询，原因正好相反。

## 5. 应用提示

如果你擅长使用SQL，那么第一印象可能是SQL版本更富于表现力，更易读，因此更好。SQL无疑是一种强大的查询语言，但记住我们要完成的是什么任务。我们希望能够尽可能多地用领域模型来工作（适当地），从而实现具有更高可维护性的解决方案。还要注意的，上面显示的C#代码有些不必要的“啰嗦”了。本书后面将讨论在通用查询对象实现之上编写薄层的语法将会是什么样的。

这样，我们得到的结果就是有关数据库模式代码的进一步透明性。通常，我认为这是一件好事。当实际需要时，总可以走出这个小的沙箱，并充分利用数据库的强大功能来表示SQL查询，而不必局限于领域模型。

另一件要指出的事情是，创建适当的查询对象实现很快就会变成一项非常复杂的任务，因此要注意不要承担过量的工作。

一个小的、良好的副作用是查询对象很容易用于本地过滤，例如保持所有产品的一个缓存列表。使用领域模型的开发人员只需像平常那样创建查询对象即可，但查询对象的使用方式略微不

同，它与数据库不发生接触。

---

**警告** 我知道，缓存的可用性正如它的危险性一样。注意，它可能会产生意外，这是一个警告。

---

一些擅长DDD的读者可能更喜欢将规格模式（Specification pattern）[Evans DDD]作为此问题的解决方案。这恰好联系到我们将要讨论的第三个，也是最后一个模式类别：领域模式。

## 2.5 领域模式

与设计模式和架构模式相比，领域模式有着非常不同的关注点。其关注点完全放在以下几个方面：如何进行领域模型本身的结构化，如何在模型中封装领域知识，以及如何应用通用语言，并且不让基础架构分散我们对重要部分的注意力。

领域模式与设计模式有一些重叠，例如设计模式中的策略模式（Strategy）[GoF Design Patterns]也被认为是一种领域模式。重叠的原因在于，诸如策略这样的模式也是用于领域模型结构化的很好工具。

作为设计模式，它们具有技术性和通用性。作为领域模式，它们关注领域模型的核心。它们使得领域模型更清晰、更富有表现力，并且更有目的性，而且让所获得的领域知识在领域模型中变得明显。

在前一小节结束时，曾提到规格模式作为领域模式的一种，是查询对象模式的一种替代。这是解释我对领域模式看法的一个好的方式。查询对象是一种技术模式，其中使用者可以用基于对象的语法来定义查询，可用于查找领域模型中的任何对象。规格模式也可用于查询，但它不使用通用的查询对象及逐条设置判断标准，而是将规格用作一个概念，这个概念本身就封装了领域知识，并传递其目的。

例如，为了查找黄金客户，我们可以同时使用查询对象和规格，但解决方案将不同。如果使用查询对象，那么将表示出有关如何定义黄金客户的标准。如果使用规格，那么将有一个可能名为GoldCustomerSpecification的类。在该例中，标准本身并未显露或被复制，而是封装在该类中（这个类具有妥善描述的名称）。

领域模式的来源是[Arlo/Neustadt Archetype Patterns]，但我从另一个好的来源选择了领域模式示例，它就是Eric Evans的书Domain Driven Design[Evans DDD]。我选中的示例模式是工厂模式（Factory）。

---

**说明** 对模式敏感的读者可能会感到混淆，因为这里讨论了工厂这种领域模式，而Design Patterns一书[GoF Design Patterns]也有一些工厂模式。再次强调，设计模式将更多重点放在技术层面上，而领域模式的重点放在语义领域模型层面上。

它们在具体目的和典型实现方面也有所不同。GoF的工厂模式被称为工厂方法（Factory Method）和抽象工厂（Abstract Factory）。工厂方法是将“正确”类的实例化推迟到子类

中进行。抽象工厂是有关创建依赖对象家族的。

DDD的工厂模式是直接的实现级模式，而且只与捕获和封装特定类的创建概念有关。

## 示例：工厂

有人说过，软件行业受到工业主义的影响。对于影响是好是坏尚有争议，但影响是肯定存在的。我们谈论软件工程是软件开发的好的原则，我们谈论架构、产品线等。这里就是另外一种这样的影响，即工厂模式。但首先，还是让我们来描述本例中的问题。

### 1. 问题

这次的问题是订单结构非常复杂，它需要处理两种特殊情况的订单：第一是当客户创建对数据库未知的新订单的时候。第二是当客户索取旧订单的时候（此旧订单需要从数据库被物化到领域模型中）。在这两种情况下，都需要创建订单实例，但就结构而言，它们的相似性仅此而已。

问题的另一部分是，订单应该总是有客户，否则，创建订单就没有意义了。然而问题的另一部分在于，我们必须能够创建新的贷记订单（credit order）和重复的订单。

### 2. 解决方案提议1

最简单的解决方案是只使用公共构造方法，像下面这样：

```
public Order()
```

然后，在调用这个构造方法后，使用者必须用正确方式来设置要插入的实例的属性，或者通过要求数据库提供值来完成这一步。

不幸的是，这将招致大量的麻烦。例如，我们可能在属性上有脏跟踪，而且我们可能不希望脏跟踪向刚刚从持久化重建的实例发出脏信号。另一个问题是如何设置标识，它可能根本就是不可设置的。反射可以解决这两个问题（至少如果标识符没有被声明为只读的话），但这是领域模型使用方开发者（consumer developer）应该关心的事情吗？我肯定地认为不是。我们可以研究一些更深奥的解决方案，但我确信大多数人将同意一个典型且明显的解决方案是使用参数化的构造方法作为替代。

由于过去我在VB6上花费了大量编程时间，因此我并未被参数化的构造方法所迷惑。读者是否相信没有参数化的构造方法？实际上我自己费了好大努力才相信这一点。

不管怎样，在C#和Java等一些语言中，我们确实拥有参数化构造方法的可能性，而且它可能是我们考虑用来解决此问题的第一种解决方案。因此，我们使用Order类的三个公共构造方法。

```
public Order(Customer customer);  
public Order(Order order,  
    bool trueForCreditFalseForRepeat);  
public Order(int orderId);
```

当创建数据库中不存在的Order新实例时，使用前两个构造方法。第一个构造方法用于创建新的、普通的Order。到目前为止，一切都很好，但这里并未考虑目的需求，这样就可以创建作为备用的Orders。当添加了目的需求时，就要修改第一个构造方法了，以便让它可用于两个不同

目的。

第二个构造方法用于创建贷记Order或旧Order的副本。这当然没有我希望的那样清晰。

最后一个构造方法在取回旧Order时使用，但揭示了使用哪个构造方法的唯一事情就是参数。这并不清晰。另一个问题（特别是对于第三个构造方法）在于，构造方法中有过多的处理被认为是不好的。直接跳到数据库给人的感觉非常糟糕。

### 3. 解决方案提议2

根据*Effective Java* [Bloch Effective Java]一书所说，57项最佳实践中的第一项就是考虑提供静态工厂方法来替代构造方法。静态工厂方法可能像下面这样：

```
public static Order Create(Customer customer);
public static Order CreateReservation(Customer customer);
public static Order CreateCredit(Order orderToCredit);
public static Order CreateRepeat(Order orderToRepeat);
public static Order Get(int orderId);
```

这种静态方法的一个优点是它有一个揭示了它的用途的名称。例如，第五个方法与解决方案1中对应的构造方法（第三个构造方法）相比就清晰得多，对吗？实际上，我认为与解决方案1相比，上面所有的工厂方法都是如此，而且现在我添加了备用需求和重复订单需求，而没有产生构造问题。

Bloch还提出了静态方法不必在每次被及时创建新实例，这可能是一大优点。另一个优点（也是更典型的）就是，它们可以返回它们的返回类型的任何子类型的实例。

有缺点吗？我认为主要缺点是，我在类中编写的那些创建型代码可能违反了SRP原则 [Martin PPP]。Evans的书[Evans DDD]使用汽车引擎这个比喻对这一点给出了解释。汽车引擎本身并不知道它是如何被制造的，这并不是它的责任。想象一下，如果汽车引擎不仅负责运行，还要负责它自身的制造，那么情况将会变得多么复杂。在创建型代码很复杂的情况下，这个缺点将显而易见。

在这个比喻上再发挥一下：引擎也可以从另一个位置获取，例如从本地或中央仓库的库架上，也就是说，通过从数据库获取旧实例并物化它，从而进行重构。这与创建实例是完全不同的，无论对于真实的物理引擎，还是对于软件中的Order实例，都是如此。

我们现在距离模式解决方案已经很近了。让我们使用一个类似于提议2的解决方案，但将创建型行为归到它自己的一个类中，现在先忘掉“从数据库获取”这件事 [获取操作由另一个领域模式来处理，即存储库模式 (Repository)，在后面章节中我们将讨论更多有关它的内容]。

### 4. 解决方案提议3

现在我们开始使用工厂模式作为领域模式。我们从一个图开始，如图2-7所示。

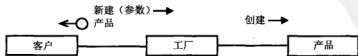


图2-7 工厂模式的实例图



从使用者角度看，获取已就绪的新Order实例的代码可能像下面这样：

```
anOrder = OrderFactory.Create(aCustomer);  
aReservation = OrderFactory.CreateReservation(aCustomer);  
  
aCredit = OrderFactory.CreateCredit(anOldOrder);  
aRepeat = OrderFactory.CreateRepeat(anOldOrder);
```

对于使用者来说，这并不比使用普通构造方法更困难。它略带入侵性，但并不严重。

使用者为了获取旧的Order，必须与其他部分进行对话（不是工厂，而是存储库）。它更清晰，而且更富有表现力，但它是我们稍后要讨论的话题。

当工厂代码位于外部类中时，只能通过领域模型中其他类的构造方法对订单进行实例化，这是无法避免的，但可以通过把构造方法设置为内部方法来缓解这个问题，而且由于工厂就在那里，因此领域模型开发人员本身能够理解那就是对类进行实例化的方式，而不是直接使用目标类的构造方法。

---

说明 Eric Evans对上段文字的评论是：“我希望有一天语言可以支持这样的概念。（正如它们现在具有构造方法一样，也许它们将允许我们声明工厂，等等。）”

---

## 5. 应用提示

首先，请注意有时构造方法正是我们需要的。例如，可能没有任何有意义的层次结构，客户端希望选择实现，构造非常简单，而且客户端拥有对所有属性的访问权。重要的是要理解这样一个事实：不是只为了创建每个实例而去创建工厂，而是在工厂有助于增加清晰度并揭示领域模型目的的情况下使用工厂。

工厂的典型特点是设置具有有效状态的实例。我现在喜欢做的一件事就是设置具有Null对象的子实例[Woolf Null Object]（如果这是适当的话）。例如，获取Order。假设Order的传输由transporter负责。

首先，Order尚未被传输（在这种情况下，我们根本没考虑很多有关传输的事情），所以无法为它提供任何Transporter对象，但我并没有将Transporter属性保留为null，而是将属性设置为空的Transporter（也许是“未选择的”Transporter）。这样，就可以总是发现一条类似于下面这样对Transporter属性的描述：

```
anOrder.Transporter.Description
```

如果Transporter为null，那么我首先需要检查它。在这方面，工厂是非常方便的解决方案。（也可以用工厂来解决构造方法的问题，但有很多地方需要使用Null对象，而且如果有很多选项的话，那么不太容易确定将哪些选项应用于Null对象。我的意思是说构造方法的复杂性增加了。）

当然，我们可以有几种不同的工厂方法，并让它们带有很多参数，这样就可以对工厂外部的创建有很好的控制。

使用工厂还可以隐藏无法避免的基础架构需求。

工厂是我们耳熟能详的一个词，而且我们也经常会看到它们的一些不太合乎语义的（或至少

是不同的)使用方式,其中所有实例(新或“旧”)都是用工厂创建的。例如,在COM中,每个类都必须有类工厂,像很多框架一样。这并不是领域模式中的工厂模式:它们的名称和技术相同,但目的不同。

另一个示例是我们可以(间接)让工厂到数据库中去获取默认值。再次强调,好的做法是在构造方法中不进行过多的处理,因为它们并不适合包含那种逻辑。

---

**说明** 现在我们有了一个问题。从外部设置一些属性应该是不可能的,但是,我们却需要从那里获取一些值。这发生在工厂设置默认值的时候(以及存储库从数据库获取旧实例的时候)。你是否对这一点感觉很不好?

后面我们将讨论这个问题,但现在要记住的是,不管怎样,这通常是一个问题,因为反射可用于更改内部状态,至少在安全设置允许的情况下是这样。

我在这一点上的态度是:当程序员使用领域模型时,不可能完全阻止他们做一些糟糕的事情,但我们应该让程序员很难做出愚蠢举动,而且必须检查是否发生这样的事情。

总之,我通常并不认为它是一个真正的问题。

---

总的来说,我认为工厂模式的使用清楚表明一些复杂实例化从Order转移到它自己的概念中。这在某种程度上也有助于提高领域模型的清晰度。实例化逻辑的有趣和复杂是一个好的迹象。

## 2.6 小结

本章是对模式走马观花式的介绍,目的是引起你对于模式的兴趣。希望这个目的已经达到了,因为后面的章节将使用模式作为一种重要工具。

现在到了深入探讨如何使用TDD的时候了,我们将通过一些示例进行讨论。



我们都认同，好的示例就是最有力的说明。TDD可以这样描述：使用测试或示例来详细说明行为。这些示例应可反复用于多种用途。例如，在开发过程中，可利用它们来查找缺失或不正确的需求。在重构时，测试将起到安全网的作用。开发完成之后则可作为质量文档。

在本章中，我们不仅仅会举例说明测试，也会通过一些示例讨论TDD本身。我认为这是一种好的讨论方法。

对TDD和基于状态的风格进行一些初步讨论之后，我的朋友Claudio Perrone将介绍以交互方式创建示例的一些技巧，并且使用桩（stub）与模拟（mock）编写测试。

TDD中所使用的关键准则就是重构。重构也是以示例为中心的。你可以这样认为：编写解决方案的第一个示例，然后使用重构优化示例，直至完成。因此，我们不必一开始就拿出完美的设计。这是一个好消息，因为那几乎是不可能完成的任务。

同样，我们也会通过一些示例来讨论重构。

让我们从TDD的一些基础知识开始。

### 3.1 TDD

使用TDD几年之后，我对这种语言的喜爱程度有增无减。我越来越确信，要成为一名更出色的程序员，TDD是最重要的一种技术。耳听为虚，眼见为实，我们必须亲自尝试，亲身体验，但我们应该先了解一下大致的情況。下面我们简短地复习一下第1章中关于流程的内容。

#### 3.1.1 TDD 流程

首先，我们开始编写一个测试。故意令测试失败，以此确保测试确实如预期一样发挥了作用。

第二步是编写一段最简单的代码，使测试通过。

第三步就是重构。

随后重复所有步骤，添加另外一个测试。

如果使用xUnit语言，第一步就应该得到一个红灯/红条，第二步应得到绿灯/绿条（遗憾的是，编译错误和重构尚无颜色指示）。

所以，我们的“口诀”就是：红、绿、重构、红、绿、重构……

### 3.1.2 演示

这里假设你熟悉Testdriven.net [Testdriven.net]、NUnit [NUnit]或其他某些类似的工具（或者从现在开始选定某种此类工具，自学基础知识）。因而，本文不会详细讨论工具，而是集中关注如何应用TDD的概念。

我知道，有很多关于TDD用法的好的演示材料，例如[Beck TDD]、[Astels TDD]、[Martin PPP]等。无论如何，我希望给出自己的演示。我不会沿用那种常见做法，自己编写一个“玩具”示例，而是使用来自实际应用程序的示例。

几年前，Dynapac邀请我编写一个应用程序，用于与他们生产的新铣刨机系列产品绑定。铣刨机的用途是在铺设新沥青路面之前移除旧沥青铺层。铣刨机能将沥青铺层的表面恢复成特定的轮廓。包块、车辙和路面上的其他不平整之处均可得到修整，得到统一、规整的表面。

这个应用程序用于计算多种不同的事物，帮助优化铣刨机的使用。应用程序需要能够计算旧沥青铺层的硬度，切削特定突起部分所需的时间，保证以最低成本进行切削所需的卡车，还有其他许多方面。

我将使用这个实际应用程序来演示TDD的用法，使你能了解流程。我将重点讨论卡车数量的计算。

确定运送旧沥青的正确卡车数量是非常重要的。如果卡车数量过少，铣刨机就不得不频繁停工或以较慢的速度移动。如果卡车数量过多，就是浪费卡车和人员。无论是哪种情况，项目的成本都会增加。

让我们以TDD的方式来处理这个问题。首先在文本文件中写下已确定的必需功能，例如：

- 计算切削能力
- 计算切削容量
- 计算所需卡车的数量

前面说过，目前我侧重于卡车数量的计算，因此在文件中为这行作了一个标记，表示我决定从这里开始。只要我（更有可能是客户提出）确定了另外一个目前不打算实现的功能，就将其添加到这个文件中。因而这个文本文件将帮助我牢记一切，同时又保证我每次关注一个问题。

---

**说明** 测试与重构可以写在同一个文件中，但我并不喜欢这样做。而是在编写测试时，为目前非重点的测试标上Ignore属性的标签。我已经发现但并不希望立即处理的重构需求很可能并不值得处理，或者也可再被发现后才处理。

---

下一项任务就是考虑卡车计算的测试。最初我认为这非常简单，但随着思考的深入，我发现这实际上相当复杂。

我们首先添加一个简单的测试。该测试对输入进行检查，当未提供任何输入时，需要的卡车数量应该是0。首先，创建一个新项目，将其命名为Tests。添加一个TRuckCalculationTests类，然后完成必要的准备工作[使之成为根据NUnit语言进行测试的一个测试固件（test fixture）]，例如设置nunit.framework的引用，为NUnit.Framework添加using子句，使用[TestFixture]

对类进行修饰。随后编写第一个测试，如下所示：

```
[Test]
public void WillGetZeroAsResultWhenNoInputIsGiven()
{
    TruckCalculation tc = new TruckCalculation();

    Assert.AreEqual(0, tc.NeededNumberOfTrucks);
}
```

实际上，在编写这个极为简单的测试时，我确定了多个小的设计决策。首先，我确定需要名为truckCalculation的类。其次，我确定这个类应有名为NeededNumberOfTrucks的属性。

当然，测试项目仍然无法编译，因为我们还没有编写它所测试的代码，下面继续添加“实际”项目。我添加了另外一个名为Dynapac.PlanPave.DomainModel的项目。在该项目中，添加如下类：

```
public class TruckCalculation
{
    public int NeededNumberOfTrucks
    {
        get {return -1;}
    }
}
```

这里令人困惑的部分就是我将返回值伪造为-1。这是为了迫使测试出错。切记，我们应始终从不成功的测试开始，而对于这项测试，返回零显然不会失败。它应以有意义的方式失败，但我并不能直接给出非常有意义的失败方式，所以必须按照这样的方法，从简单入手（我并未将此写入实际项目）。

随后，返回Tests项目，设置对Dynapac.Plan-Pave.DomainModel项目的引用。我们还添加了using子句，构建了整个解决方案。此后，我们执行了Tests项目中的测试（实际上或者只是目前为止构建的测试）。希望得到红条。之后再返回专用代码并将其更改为返回0；重新执行测试并得到绿条。

这里是否有要重构的内容？至此，我对这段代码非常满意。这是我能想到的最简单的代码，而且满足了测试（或者说是测试项目）。

我们已经向正确的方向迈出了一小步。我们已经开始实现了计算卡车数量的新功能，也得到了一个简单但好的测试。

让我们再前进一小步，由测试来驱动我们的进度。因此需要给出另一个测试。在测试领域中，我需要显示经过舍入的结果和更加精确的结果——至少要精确到一位小数。舍入结果必须始终采用只入不舍的方式，因为不可能出现半台卡车的现象。（我知道你现在正想什么，但混合使用各种不同规格的卡车绝对不是客户所需的）。这将会是一个更加正式、更加必要的测试，但我并不想立刻着手开始这个测试。而是希望增加一个有趣的步骤，添加带有Ignore属性的舍入测试，如下所示：

```
[Test, Ignore("Deal with a little later")]
```

```
public void CannotRoundDecimalTruckDown()
{
}
```

我希望为计算做些准备。这项计算真是太困难了，必须考虑到运输距离、运输速度、卡车容量、切削容量、卸车时间、装车时间等。继续分析，将问题略加简化，假设目前并不关注运输方面，也不关心装车的时间，甚至不关心其他一些尚未考虑到的因素，而是仅仅关注切削容量、卡车容量和卸车时间。这样问题就足够简单了，我可以编写出下面这样的测试：

```
[Test]
public void CanCalculateWhenOnlyCapacityIsDealtWith()
{
    TruckCalculation tc = new TruckCalculation();

    tc.MillingCapacity = 20;
    tc.TruckCapacity = 5;
    tc.UnloadingTime = 30;

    Assert.AreEqual(2, tc.NeededNumberOfTrucks);
}
```

我在测试中假设切削容量是20吨/小时，每辆卡车每次可以处理5吨沥青，也就是需要卸车4次。我还假设卡车的卸车需要花费30分钟，因此每辆卡车在一小时内只能使用两次。这也就表示我们需要2辆卡车，我对此进行了测试。

**说明** 目前，计算看上去可能有些奇怪，因为所考虑的因素过少，我仅仅是尝试从简单入手。这里最重要的是不是计算本身，而是使用TDD开展工作的流程，因此请不要让计算本身分散了注意力。

尝试编译Tests项目时，它失败了，因为新测试需要三项新属性。下面我们为TruckCalculation类添加这些属性。如下所示：

```
//TruckCalculation
public int TruckCapacity = 0;
public int MillingCapacity = 0;
public int UnloadingTime = 0;
```

这甚至算不上属性，而只是公共字段。稍后将对此加以讨论。现在生成解决方案，希望能得到红条。

一切恰如预期。我们需要对NeededNumberOfTrucks略加修改，以实现真正的计算（或至少更加贴近真实的计算）。

```
//TruckCalculation
public int NeededNumberOfTrucks
{
    get
    {
        return MillingCapacity /
```

```

        (TruckCapacity * 60 / UnloadingTime);
    }
}

```

我意识到，这一步或许跨得过大，但我现在对此充满信心。

再次生成并重新执行测试。绿条……红条。怎么会是这样？上一个成功运行的测试并不是问题。问题在于原有的测试现在失败了。它抛出了异常。当然，第一个测试并未给truckCapacity和UnloadingTime属性提供任何值，我将它除以零。这是一个有些愚蠢的错误，但看到红条让我很高兴。即便这样简单的初始测试也能帮我在创建了bug后的短短几分钟乃至几秒内发现bug，这样极短的反馈循环是非常强大的。

我需要进一步修改计算：

```

//TruckCalculation
public int NeededNumberOfTrucks
{
    get
    {
        if (TruckCapacity != 0 && UnloadingTime != 0)
            return MillingCapacity /
                (TruckCapacity * 60 / UnloadingTime);
        else
            return 0;
    }
}

```

再次生成，运行测试，得到绿条。我们又向正确的方向迈出了一步——有了测试的保障（至少是一定程度上的保障）。

是否需要重构？我确信，有些读者并不欣赏我使用公共字段。我自己也曾经非常痛恨公共字段，但现在已经改变了想法。只要字段可读、可写，在读取或写入值时无需拦截，公共字段就与属性一样好。它们的优点在于简单，但若需要在设置一个值时进行拦截，公共字段就存在缺点。如有必要，我将在后面进行重构。

---

**说明** 就反射而言，公共字段和公共属性之间存在差别，如果选择在两者之间切换，这可能会产生问题。

一位评论者曾指出了我并未考虑到的另一项差异：公共属性无法用作ref参数，但公共字段可以。

整个讨论均依赖于语言。例如，对于C#或VB.NET来说，用户使用公共字段和公共属性的方式完全相同，但在C++和Java中则不是这样。

---

我要重构的另外一项内容就是为测试类添加[SetUp]方法，在实例级别实例化计算成员。在考虑重构时不要忘记测试。无论如何，我并不认为有必要做出这样的更改，至少在目前不是。这能稍微减少一些重复的内容，但也会使测试变得不那么清晰。对于测试而言，清晰明确总是优先

考虑事项。

我不满意的另外一个方面就是计算本身的代码有些混乱。我认为良好的解决方案应该消除卫语句（guard）中出现零值的不常见情况。通过这种方式，即使普通、实际的计算更加明确。这项更改并非十分重要，而是一种个人偏好，但我认为这能使代码的可读性提高。无论如何，让我们使用卫语句来实现称为Replace Nested Conditional的重构[Fowler R]：

```
//TruckCalculation
public int NeededNumberOfTrucks
{
    get
    {
        if (TruckCapacity == 0 || UnloadingTime == 0)
            return 0;

        return MillingCapacity /
            (TruckCapacity * 60 / UnloadingTime);
    }
}
```

**说明** 与模式名相同，重构名称也可以作为开发人员之间沟通的手段。

生成并运行测试，我们仍然会看到绿条，代码更加清晰，但我们还可以做得更好。我认为这里最好像下面这样使用Consolidate Conditional Expression [Fowler R]：

```
//TruckCalculation
public int NeededNumberOfTrucks
{
    get
    {
        if (!_IsNotCalculatable())
            return 0;

        return MillingCapacity /
            (TruckCapacity * 60 / UnloadingTime);
    }
}
```

采用这种做法时，公式变得不那么清晰了。如果我能使用Extract Method重构[Fowler R]将部分代码更改为属性调用，其目的会更加清晰，如下所示：

```
public int NeededNumberOfTrucks
{
    get
    {
        if (!_IsNotCalculatable())
            return 0;

        return MillingCapacity /
            _SingleTruckCapacityDuringOneHour;
    }
}
```



目前看来这非常好，但现在应该添加另外一项测试。我认为简单介绍如何编写新类来计算必要的卡车数量已足以展示TDD的流程。

在初次编写测试时，部分读者可能更希望得到智能感知（intellisense，可通过“猜测”要编写的内容来节省键入时间）的帮助。我也非常喜欢智能感知，但在这里我并不为没有它的帮助而遗憾。切记，我们在这里讨论的是接口，最好编写两次，细心检查。

另外还要注意，应该根据对所编写代码的自信程度来改变工作速度。如果因过于自信、过于急切而导致难题，就应该放慢速度，编写一些较小的测试和较小的代码块返回去继续重来。

### 3.1.3 设计效果

在演示中，我提到过编写测试就是一项设计工作。下面列举一些我在TDD设计中发现而在具体的预先设计中没有的效果。

#### □ 更多的客户端控制

我曾经认为，应该尽可能向外界隐藏与配置有关的内容，使类自行完成配置。要使用TDD，就需要使测试能够按照测试的需求设置实例。这实际上是一件好事，因为类重用起来更容易。（当然，应警惕潜在错误。）

#### □ 更多接口/工厂

为了支持桩或模拟对象的使用（或至少使这些对象的使用更为简单），需要通过接口聚合功能，以便传入测试对象而非真实对象，因为在测试环境中设置真实对象可能较为困难。如果你具有COM背景知识，就应该了解，这种处理接口的方法还有其他许多优点。

---

**说明** 本章稍后的部分还会进一步讨论桩和模拟。目前，可以说桩就是一种“替身”，等同于提供封装值的空接口，这些值是专为能够开发和测试桩的使用者而创建的。模拟是用于测试的另一类“替身”，在设置时可指定其预期调用方式，并可在此后验证这些预期。

---

#### □ 公开更多子结果

为了每次取得一小步进展，需要公开子结果，使之可测试。（例如，如果继续上文中的演示，你将看到我公开更多关于运输和装车时间的子结果。）只要子结果为只读状态，这就不会成为大问题。我们常常需要在用户界面中公开子结果。应注意不要公开过多的算法。应在这种公开与信息隐藏的惯例之间取得平衡。

#### □ 抓住重点

如果我从详细的预先设计入手处理演示的例子，就必然要创建卡车类，在其中包含大量关于卡车的属性。由于我关注的仅仅是使测试能够运行的必要内容，因而完全跳过了卡车类。我需要的与卡车相关的内容只有卡车容量属性，我将此属性直接设定在卡车计算类中。

#### □ 更少耦合

TDD带来的耦合更少。例如，通过自动测试来测试UI非常困难，UI因此无法与域模型混合。接口的更广泛应用也有助于减少耦合。

如上文所述，设计效果不仅仅包括优点。同样也存在许多其他效果，例如：

#### □ 第二版API

使用TDD时，在你的使用者使用API之前，API已经用过了一次。使用者很有可能会发现，API比应用其他方法开发的更好、更稳定。

#### □ 未来维护

如果使用广泛的测试套件，维护将轻而易举。需要更改时，测试将立即告诉我们更改是否会给使用者造成负面影响。

如果没有广泛的测试套件，在需要更改时，就可以先行创建测试。这并不是一项有趣的工作，在编写出代码的很久之后很可能无法创建高质量的测试，但与莽撞地进行修改然后祈祷一切顺利相比，它是一种更好的方法。而且在后续进行更改时，也就有了可以利用的测试。尽管使用了TDD，生产代码中仍可能会出现bug。发生这种情况时，需要修复bug，这个过程与普通的TDD流程极为相似。首先编写公开bug的测试，然后编写代码来使测试（新测试和所有旧测试）成功执行。最后，如果必需则进行重构。

#### □ 文档

我们所编写的测试就是高质量的文档。这是优先编写测试的另一个原因。在理解事物的时候，示例总是很有帮助，测试正是表现哪些能及哪些不能正常工作的示例。

看待单元测试的另外一种方式就是展示假设，并了解其他开发人员的想法。也就是说，其他开发人员也会展示他们对于你的类的假设。如果他们的测试失败，很可能是你的类行为未如预期，而这对你来说是非常宝贵的信息。

#### □ 更加智能化的编译器

过去，我曾竭力使编译器在我出错时提供信息，以此为我提供帮助。与此相比，测试则更进了一个层次。测试在查找编译器遗漏的问题时非常有帮助。也可以说，编译器擅长发现语法问题，而测试能发现语义问题。

使用TDD时，类型安全的重要性甚至可能略有下降，因为测试能轻松捕捉到类型错误。这又意味着你可以利用更动态的测试。

如果沿用TDD的理念，我们还会发现调试器的使用也比其他方法更加省时省力。

---

**说明** 尽管如此，有时我们可能仍然需要检查变量值等内容。可以使用`Console.WriteLine()`调用，可以在调试器中运行测试。我个人认为这是一种信号，表明了急于使用此类技术时步伐过大，或者测试过少。

---

#### □ 其他测试中的可重用机会

在开发过程中编写的测试在集成测试中也是有用的。根据我的经验，许多生产时出现的

问题都是由于生产环境与预期存在差异而导致的。如果进行了广泛的测试，就可以利用它们来检查最终应用程序的部署。一次付出，长期受益。

总而言之，测试友好的设计（或可测试性）是现今一项非常重要的设计目标。在做设计选择时，这或许也是一大主要因素。

### 3.1.4 问题

上面的介绍或许令我看上去有几分像推销员。但实际上问题也同样存在。下面我将说明一些问题。

#### □ UI

很难将TDD用于UI。TDD更适合领域模型和核心逻辑，但这不一定是一件坏事。它能帮助我们更严格地将UI与领域模型区分开来，这也是一项基本的最佳设计实践（应始终保持这种做法，TDD也促进了这种实践）。确保编写极为简练的UI，将逻辑从表单中提取出去。

#### □ 持久数据

使用TDD时，数据库确实会导致棘手的问题，这是因为在一次测试中写入数据库之后，数据库在下一次测试中的状态就发生了变化，这会导致隔离方面的麻烦。我们可以编写代码，在各次测试之间将数据库设置为某种明确的状态，但这又增加了测试人员的工作负担，测试的运行也会更加缓慢。

在编写测试时，还要使其不受状态变化的影响，这可能会稍微降低测试的能力。另外一种解决方法是，在集成测试过程中使用桩或模拟对象来模拟数据库，在测试之前恢复数据库备份。

还有另外一种方法，就是专为测试使用事务。毫无疑问，如果测试集中关注事务语义，这就不是一种有用的解决方案。

---

**说明** 我们将在第6章中讨论数据库边界测试。

---

#### □ 错误的安全感

得到绿条并不意味着零bug，它仅仅意味着测试未检测到任何bug。应确保TDD不会带来错误的安全感。切记，结果为绿色的测试不能证明不存在bug，而只表示所执行的特定测试并未检测到问题。

另一方面，测试不完美并不意味着不应使用它们。这样的测试依然能帮助我们更轻松地提高质量！

就像我的朋友Mats Helander曾经说过的那样，测试的价值并不在于绿色，而在于红色！

#### □ 缺乏全局观

TDD可视为自下而上的方法，我们必须注意可能会时常忽视全局。时常将代码可视化，UML无疑是一种很好的方法，这将使我们获得全局观。你很可能发现，在为未来发展

准备代码时，需要应用某些重构。

□ 要维护的代码更多

维护方面就像一把双刃剑。测试对维护工作大有裨益，但同时也使我们需要维护更多代码。

只能说，TDD难以应用到UI和数据库，这也是领域模型模式的优点之一，因为我们通常可以在不涉及数据库和UI的前提下运行有意义的测试，这优于其他逻辑结构！我们将在后续章节中讨论数据库测试和UI测试。

总之，我认为TDD是一种极为出色的工具！

### 3.1.5 下一个阶段

上面介绍了基本概念，但我们涉及的仍然仅仅是表面。开始应用TDD后，可能要经历过一些问题之后才能开始顺利工作。

经过一段时间，我们可能会发现设定某些测试时过于复杂，大部分测试都是枯燥的内容——至少测试本身不能令人感兴趣。这是一个良好的迹象，因为你已经到达了TDD的下一个阶段。在下一节中，我的朋友Claudio Perrone将进一步探讨模拟和桩。

## 3.2 模拟和桩

我曾提到过将在本书后续内容中讨论数据库测试。现在应该具体讨论其相关内容了。在这里，Claudio Perrone将介绍如何在测试中应用桩和模拟技术。在上一节中，我所探讨的是基于状态的测试。而在这一节中，Claudio将重点关注基于交互的测试。

为了保持同步，Claudio将首先对与数据库相关的示例使用经典方法，下一章再深入探讨领域模型模式。

下面请看Claudio的介绍。

### 3.2.1 典型单元测试

作者：Claudio Perrone

测试对象行为的一种常见方法就是，为其设置相关的上下文信息，调用其方法之一，编写一些断言，检查返回值以验证方法是否按照预期对环境状态做出了更改。

下面的示例测试了业务组件UserBC的SaveUser()方法，演示了这种方法：

```
[Test, Rollback] //Automatic rollback
                //(using Services w/o components)
public void TestUserBCCanSaveUser ()
{
    // Setting up context information (preconditions)
    IUserInfo user = new UserInfo(
        "Claudio", "Perrone", "MyUniqueLogin", "MyPassword");
    // I'm not testing yet, just clarifying initial state
    Assert.IsTrue(user.IsNew);
    Assert.AreEqual(NEW_OBJECT_ID, user.ID);
```

```

Assert.IsFalse(
    IsUserInsertedInDataBase(
        "Claudio", "Perrone", "MyUniqueLogin", "MyPassword"));

// Preconditions are ok - now exercising method to test
UserBC bcUser = new UserBC();
bcUser.SaveUser(user);

// Verifying post-conditions (state changed as expected)
Assert.IsFalse(user.IsNew);
Assert.IsTrue(user.ID != NEW_OBJECT_ID);
Assert.IsTrue(
    IsUserInsertedInDataBase(
        "Claudio", "Perrone", "MyUniqueLogin", "MyPassword"));
}

```

**说明** NUnit目前并未提供Rollback属性（尽管其路线图中计划提供）。我使用了Roy Osherove编写的一个修订版本，参见<http://weblogs.asp.net/rosheroove/archive/2004/07/12/180189.aspx>。

由于UserBC业务组件要获取业务实体，并将其保存到数据库中，因而测试验证其行为的方法就是这样的：创建实体，将其作为参数传递给SaveUser()方法，然后验证该实体是否已保留在数据存储中。

### 3.2.2 声明独立性

对于这种测试风格，可能出现的一个潜在问题是对象很少是独立操作的。正在测试的对象所依赖的其他对象往往要承担部分工作。

为了通过示例演示这个问题，我们按照如下方式实现UserBC类（方便起见，忽略所有异常处理代码）：

```

public class UserBC
{
    public void SaveUser(IUserInfo user)
    {
        user.Validate();
        UserDao daoUser = new UserDao();
        if (user.IsNew)
            daoUser.Insert(user);
        else
            daoUser.Update(user);
    }
}

```

在本例中，UserBC将用户验证委托给UserInfo业务实体。如果违反了一条或多条业务规则，UserInfo将抛出自定义异常，其中包含一组验证错误。UserBC还会将业务实体的保持委托给UserDao数据访问对象，该对象负责所有数据库实现细节的抽象。

由于IUserInfo是SaveUser()方法的显式参数，因而可以说UserBC显式地声明了对于

IUserInfo接口的依赖性（由UserInfo类实现）。但对于UserDao的依赖性隐藏在UserBC类实现的内部。

使事情更加复杂的是，依赖性是可传递的。举例来说，如果UserBC依赖于UserInfo，而UserInfo需要一组BusinessRule对象来验证其内容，那么UserBC就依赖于BusinessRule。BusinessRule对象中的bug会导致测试中断，同时也会导致其他一些测试出错。

实际上，领域模型处理的复杂逻辑往往是通过一连串对象实现的，这些对象会将部分行为转发给其他协作对象，直至得到所需的结果。因而，如果某个单元测试的目的是验证具有众多依赖项的对象的行为，而如果这一连串对象中的某个对象存在bug，则测试可能会失败。这就造成了难以辨别错误的起因，测试实际上也就是小规模集成测试，而非“纯粹的”单元测试。

### 3.2.3 处理困难因素

在这个分布式应用程序的世界中，还有一个与协作对象有关的问题极为常见。当我们尝试使用的方法依赖于难以重建的组件或条件时，如何才能测试对象的行为？例如下面这样的组件。

- 尚未实现。
- 难以设置或测试（例如，用户界面组件或消息传递通道）。
- 过于缓慢（例如数据访问层组件、服务代理或分布式组件）。
- 包含难以再现的行为（例如间断性的网络连接或并发问题）。

### 3.2.4 用测试桩替换协作对象

对于前述问题，一种可行的解决方案是使用测试桩替换部分或全部协作对象。测试桩就是对真实对象的模拟，专门用于测试。它提供了一种为所测试的代码提供预期值的机制。

回头来看我们的示例。在本例中，SaveUser()方法内有两个密切协作的对象：UserInfo和UserDao。UserInfo易于替换，因为我们的测试只需提供实现了IUserInfo接口的对象，调用其Validate()方法时不会抛出异常即可（当然，除非我们希望测试当发生验证异常时SaveUser()方法的行为）。

UserDao的替换较为困难，因为对象的构建是内嵌在UserBC类之中的。我们需要替换这个协作对象，因为它对于数据库的访问会减慢测试的执行速度。此外，检查数据库中的值是一项耗时的操作，可能只有在独立测试UserDao或在集成测试中才是有价值的。可行的解决方案之一是，从UserDao中提取出接口，为UserBC类添加构造方法，对IUserDao设置显式依赖性。

所需代码非常简单，引入bug的可能性极低。在某些情况下，甚至可以考虑除现有构造方法以外编写此代码，仅用于测试。

```
public class UserBC
{
    private IUserDao _daoUser;

    // Default constructor
    public UserBC()
    {
```

```

    _daoUser = new UserDao();
}

// Constructor used by testing code
public UserBC(IUserDao daoUser)
{
    _daoUser = daoUser;
}

public void SaveUser(IUserInfo user)
{
    user.Validate();

    if (user.IsNew)
        _daoUser.Insert(user);
    else
        _daoUser.Update(user);
}
}

```

现在可以轻松创建一组桩，实现IUserInfo和IUserDao。它们的实现比较简单，因此这里只介绍UserDaoStub。

```

public class UserDaoStub : IUserDao
{
    private IUserInfo _userResult = null;

    // Note: Test will need to set the expected value
    // to be returned when Insert is called
    public IUserInfo UserResult
    {
        get { return _userResult; }
        set { _userResult = value; }
    }

    public void Insert(IUserInfo user)
    {
        // Note: Before calling this method our test will need to
        // set up the UserResult property with the expected values
        user.ID = UserResult.ID;
        user.Name = UserResult.Name;
        // etc
    }
    ...
}

```

此时有几个重要的方面需要考虑。

首先，我们的初始测试设定了调用SaveUser()方法后IUserInfor对象的几种预期状态。如果修改测试，使之使用UserDaoStub对象，我们就需要在执行Insert()方法之前设定UserResult属性来设置预期结果。

要考虑的第二个方面是对于数据库的依赖性现已完全去除，初始测试速度大大提高。

然而，由于SaveUser()委托了大多数行为，因此这样的测试仍然不会提供任何价值。实际上，经过进一步的考虑，我还应该补充一点：我们犯下了致命的错误（遗憾的是，这也是一种常见的错误），即在桩返回的值中加入了断言。因而我们测试的实际上是桩，而不是UserBC！

另一方面，如果希望了解数据访问层抛出异常时（如数据库中已有用户登录时发生的DalUnique-ConstraintException），UserBC类会做何反应，情况就截然不同了。

桩中的Insert()方法现更改如下：

```
// UserDaoStub
public void Insert(IUserInfo user)
{
    if (ThrowDalUniqueConstraintExceptionOnInsert)
        throw new DalUniqueConstraintException();

    user.ID = UserResult.ID;
    user.Name = UserResult.Name;
    // etc
}
```

假设我们希望UserBC捕捉DalUniqueConstraintException，并抛出恰当的业务异常，以保持层的抽象。那么以下测试演示了这种场景：

```
[Test] // Note no database is needed anymore!
[ExpectedException(
    typeof(BusinessException),
    "The provided login already exists.")]
public void TestUserBCSaveUserThrowsBusinessException()
{
    // Setting stubs to remove all dependencies
    IUserInfo user = new UserInfoStub (
        "Claudio", "Perrone", "Login", "MyPassword");

    UserDaoStub daoUser = new UserDaoStub();
    daoUser.ThrowDalUniqueConstraintExceptionOnInsert = true;

    // Executing test - it should throw a business exception
    UserBC bcUser = new UserBC(daoUser);
    bcUser.SaveUser(user);
}
```

正如预期，测试速度极快，无需访问数据库或其他协作对象，而且我们能够在难以重建的条件下快速验证所测试类的功能性。因而模拟其他条件也就简单多了。

这给我们提供了关于桩的重要经验。使用测试桩，我们能够隔离被测试的代码，观察它对伪协作对象模拟的外部条件会做何反应。

### 3.2.5 用模拟对象替换协作对象

桩的一种知名变体就是模拟对象的概念。模拟对象是对真实对象的模拟。它可替换协作对象，为被测试的代码提供预期值。此外，它还提供了一种机制，可设置预期使用方式，并能根据这些预期提供一些自我验证。



为了演示其含义，我们创建一个测试，关注UserBC与协作对象的交互。这一次，我们将使用名为NMock [NMock]的常用的.NET模拟对象框架。这种开源框架有一项很好的特性，就是在运行时使用反射从现有类或接口动态生成模拟。

```
[Test]
public void TestUserBCSaveUserInteractsWell()
{
    // (1 - Setup) Create mocks dynamically based on interface
    DynamicMock mockUser =
        new NMock.DynamicMock(typeof(IUserInfo));
    DynamicMock mockUserDao =
        new DynamicMock(typeof(IUserDao));

    // Set up canned values (same as stubs)
    mockUser.SetupResult("FirstName", "Claudio",
        typeof(string));
    mockUser.SetupResult("ID", NEW_OBJECT_ID, typeof(int));
    mockUser.SetupResult("IsNew", true, typeof(bool));

    // Generate mock instances (need to cast)
    IUserInfo user = (IUserInfo) mockUser.MockInstance;
    IUserDao daoUser = (IUserDao) mockUserDao.MockInstance;

    // (2 - Expectations) How we expect UserBC to deal with
    //      mocks
    mockUser.Expect("Validate");
    mockUserDao.Expect("Insert", user);
    mockUserDao.ExpectNoCall("Update", typeof(IUserInfo));

    // (3 - Execute) Executing method under test
    UserBC bcUser = new UserBC(daoUser);
    // Unexpected calls on the mocks will fail here
    // (e.g. calling Validate twice)
    bcUser.SaveUser(user);

    // (4 - Verify) Checks that all expectations have been met
    mockUser.Verify();
    mockUserDao.Verify(); // Note: No need for assertions
}
```

**说明** NMock可生成实现接口的类，或生成实际类的子类。无论是哪种情况，我们都要使用多态形式，使用所生成类的一个实例来替代实际类。尽管这种方法非常强大，但也存在一些明显限制。例如，不可能创建密封类的模拟，模拟方法必须被标记为virtual方法。NMock的一种有趣的替代方案就是Pretty Objects提供的POCMock[POCMock]商业框架。在这种情况下，模拟的创建是通过静态地替代协作对象来实现的。

可以看到，我们的测试中并不需要断言，因为它们位于模拟对象的内部，设计用于确保模拟按照预期被所测试的代码调用。例如，调用Validate()两次就会立即抛出异常，甚至在调用

Verify()之前就会出现异常。

这个例子使我们得出了以下的观察结果：模拟对象在验证其用法正确与否时，使我们能够更深入地了解所测试的对象与协作对象的交互方式。

### 3.2.6 设计含义

初次接触模拟对象时，我认为找到一种简单的机制来替换协作对象是一项十分复杂的工作。根本问题在于，我的代码与这些协作对象的具体实现紧密耦合，而不仅仅是其接口。后来我发现了两种重要的机制——依赖注入（Dependency Injection）和服务定位器（Service Locator），这才令我的想法发生了彻底改变。

第一个原理——依赖性注入，假设类显式地声明其协作对象的接口（例如，在构造方法中声明或作为方法的参数），但将创建具体实现的责任留给容器。由于类不再控制其协作对象的创建，因而这条原理也称为控制反转（Inversion of Control）。

第二个原理——服务器定位器，表示一个类通过与另外一个对象（定位器）的依赖关系来定其具体协作对象。定位器的最简单示例就是使用配置文件动态加载所需协作对象的工厂对象。

---

说明 第10章中将进一步介绍依赖性注入、控制反转和服务定位器。

---

### 3.2.7 结论

使用模拟对象和桩，我们能够进一步隔离需要测试的代码。这是一个优点，但同时也是一种限制，因为测试有时会隐藏集成问题。在测试的速度加快时，它们与被测试系统的耦合程度往往过于紧密。因而，只要被测试系统出现更好的实现，它们立刻就会被淘汰。

重构活动旨在引入模拟，以便使对象与其依赖项的特定实现解耦。尽管通常可以通过包含虚拟方法的类来创建模拟，但建议尽可能使用接口。因而，我们常常会看到系统被设计成使用模拟对象测试，而此类模拟对象包含大量专为测试而引入的接口。

### 3.2.8 更多信息

关于模拟对象和桩（以及基于状态的测试和基于交互的测试）之间差别的深入探讨，参见Martin Fowler的“Mocks Aren't Stubs”[Fowler Mocks Aren't Stubs]。

感谢Claudio！我们又掌握了一种强大的工具。我们能否再接受一种强大工具？接下来我们将介绍重构。

## 3.3 重构

前面曾提到过，重构是TDD一般流程中的第三个步骤，我认为现在应该简单解释一下这个术语。重构的含义是逐步实现较小、众所周知的更改，以改进现有代码的设计。也就是说，实现更改以提高代码的可维护性，而不会改变代码的观测行为。换言之，改变的是方式而非内容。

“重构”这个术语已经成为一个热门词汇，有着很多的过度使用和错误使用。例如，我们不

会对代码库进行重大、根本性的更改，而是说重构它。而实际上这并不是这个术语的核心含义。

第1章已经很详细地说明了重构的含义，以及为什么要使用重构。在本节中，我们将探讨如何使用重构。我们将从用于清理例程的基本重构代码入手。随后简要讨论如何通过新工具支持来提高工作效率。最后，我们将进行模式重构，以避免出现可维护性问题。

## 清理不良代码

之前的TDD示例确实讨论了一些重构的内容，但我认为现在应该给出另外一个重构的示例。首先从以下代码片段开始，这是一个Person类：

```
public class Person
{
    public string Name = string.Empty;
    public DateTime BirthDate = DateTime.MinValue;
    public IList Children = new ArrayList();

    public int HowOld()
    {
        if (BirthDate != DateTime.MinValue &&
            BirthDate < DateTime.Now)
        {
            int years = DateTime.Now.Year -
                BirthDate.Year;
            if (DateTime.Now.DayOfYear <
                BirthDate.DayOfYear)
                years--;

            return years;
        }
        else
            return 0;
    }
}
```

### 不良代码

至此为止，我已经多次提到不良代码（smelly code），在后面几页的内容中，不良代码将作为我们的重构工作的关注点。因而，应该指出Martin Fowler和Kent Beck在介绍重构的图书中列举了一组不良代码[Fowler R]。

典型的示例——往往也是第一个被提到的示例，就是重复代码（Duplicated Code）。

#### 1. 例程清理

我们刚才介绍的Person类是一个正确的类，可解决手头的问题。从该类的本质可以推导出，需求是跟踪特定人员的姓名和生日、他们的孩子以及年龄。同样，我们假设这些需求均已满足。一切都很好，但我们的技术债务（technical debt）已经增加了，因为这段代码较为杂乱，此时就到了重构发挥作用的时候。我们要通过一些示例来清理代码，我称之为“例程清理”，所有示例均基于Fowler的著作[Fowler R]。

● 例程清理示例一：封装字段

第一个示例的目标是Name字段。这目前是一个公共字符串，对某些用户来说可能过于复杂。我们将它从原有形式：

```
public string Name = string.Empty;

转变为下面这样的形式：

//Person

private string _name = string.Empty;
public string Name
{
    get {return _name;}
}

public Person(string name)
{
    _name = name;
}
```

之前我曾经提到过，需求已得到了满足。确实是这样，可以设定并读取Name。但完全可以做得更成功。Name字段的只读方面之前尚未被处理，但现在我们将加以处理。

我们是否已经完成了任务？能否做得更好？我认为使用公共只读字段的效果应该更好，如下所示：

```
//Person
public readonly string Name;
public Person(string name)
{
    Name = name;
}
```

这更加清晰明确，但遗憾的是，这种做法同样存在缺陷。最令人担忧的缺陷就是无法通过反射设置Name，而这对许多工具来说都是非常重要的，例如许多对象关系（O/R）映射工具。此时，我需要通过反射设置Name的选项，因此我决定转而采用基于get的版本。

这提醒了我……是否所有使用者代码仍然能够运行？毕竟代码的可见行为发生了细微的变化，因为我要求Name通过构造方法获得值。我得到了编译器错误，这需要解决，但是我们的工作是否就到此为止了呢？

事实并非如此。假设有如下代码片段：

```
//Some consumer
public void SetNameByReflection()
{
    Person p = new Person();

    FieldInfo f = typeof(Person).GetField("Name",
        BindingFlags.Instance | BindingFlags.Public);
```

```
f.SetValue(p, "Jimmy");
```

```
...
```

该代码片段通过反射设置了Name字段（你可能会对为什么这样做感到好奇，但这并不是我们的侧重点）。这段代码无法工作，但编译器也不会检测出问题。我们最好能编写出自动测试，检测到这样的问题。

因此，我们得到的教训就是重构需要自动测试！实际上，重构和TDD是相互依存的。就像本章开头提到的那样，TDD的“口诀”就是：红、绿、重构；红、绿、重构……

（重构确实不需要TDD，它需要的是自动测试。另一方面，使用TDD是一种保证编写此类自动测试的好方法。）

#### ● 例程清理示例二：封装集合

我希望处理的第二个问题是子类的公共列表，如下所示：

```
public IList Children = new ArrayList();
```

解决方案应令使用者能够在列表中进行添加和删除，而父类不会知道这一切。（为了保证公平，此处不会违背任何需求，但我仍然不喜欢这种做法。）除了Person实例之外，还可以添加字符串和整数。使用者甚至可以交换整个列表。

典型的解决方案是创建类型安全的Children列表类，使用仅接受Person实例的Add()方法。这就解决了类型安全不足的问题。也可以使父类能够知道添加和删除操作，但并未解决交换整个列表的问题。这可以通过对列表本身进行封装字段重构来解决。

这种解决方案的问题在于，我们必须编写实际上并不需要的哑元代码（dummy code）来获得类型安全的列表。这确实是可行的，但并不是我喜欢的风格。如果我们针对的是通用平台（例如.NET 2.0），就可以更好地避免缺乏类型安全性的问题。

也就是说，无论是否有通用平台，我都倾向于使用封装集合重构。首先，使用封装字段重构得到以下代码：

```
//Person
private IList _children = new ArrayList();
public IList Children
{
    get {return _children;}
}
```

至少使用者不再能够交换整个列表了，但使用者仍可在列表中添加任意内容并任意删除项目。因而，下面需要为使用者提供一个交换后的列表，如下所示：

```
get {return ArrayList.ReadOnly(_children);}
```

最后，需要为客户提供添加元素的方法，因此为Person类添加了AddChild()方法，如下所示：

```
//Person
```

```
public void AddChild(Person child)
{
    _children.Add(child);
}
```

这样我们就有了类型安全性，添加新实例时，父类就能了解相关情况，以便拦截操作（如果需要的话，而实际上在本例中并不需要这样做）。

遗憾的是，这改变了Person类的可见行为，必须据此衡量是否继续。在本例中，大多数必要的使用者更改都不会被编译器发现。幸运的是，我们有了自动测试，以第二编译器层的形式检测问题。如下所示的测试将有所帮助：

```
[Test]
public void CanCalculateNumberOfKids()
{
    Person p = new Person("Stig");
    p.Children.Add(new Person("Ulla"));
    p.Children.Add(new Person("Inga"));
    Assert.AreEqual(2, p.Children.Count);
}
```

**说明** 为了保持清晰，就像我们在这里看到的一样，测试并未指出需要更改的实际使用者代码。而只是提供了Person类的测试代码。要指出使用者代码本身的必要更改，就必须要有使用者测试。

我们很容易更改p.Children.Add()行，因此将得到以下代码：

```
[Test]
public void CanCalculateNumberOfKids()
{
    Person p = new Person("Stig");

    p.AddChild(new Person("Ulla"));
    p.AddChild(new Person("Inga"));
    Assert.AreEqual(2, p.Children.Count);
}
```

IList具有Add()方法，这有些令人遗憾，尽管它被包装为只读列表，但测试仍很容易捕捉到问题。要解决这个问题，可将Children更改为ICollection。

是否还有其他问题需要处理？还要处理HowOld()方法。它还不太清晰，是吗？

#### ● 例程清理示例三：提取方法

针对这个问题，或许你目前已经有了另外一种更出色的解决方案？在考虑更好的解决方案之前，我希望简化代码，使之更易于阅读和理解。重构本身实际上就是尝试理解代码的一种方法。它有着很好的边际效应，我们对代码的理解越好，就能越轻松地发现简单且更好的解决方案。

我们得到以下代码：

```
//Person
```

```

public int HowOld()
{
    if (BirthDate != DateTime.MinValue &&
        BirthDate < DateTime.Now);
    {
        int years = DateTime.Now.Year - BirthDate.Year;

        if (DateTime.Now.DayOfYear < BirthDate.DayOfYear)
            years--;

        return years;
    }
    else
        return 0
}

```

首先，有两种不同的情况都会返回0。这并不理想。我倾向于在不正确的情况下抛出异常。其次，我希望使用提取方法重构，消除不直观的逻辑表达式，使用解释性的方法名称取而代之。因此将

```

if (BirthDate != DateTime.MinValue &&
    BirthDate < DateTime.Now)

```

修改为

```

if (_CorrectBirthDate())

```

无论逻辑表达式是什么，它在HowOld()上下文中的用途现在都变得更加明确了。

另外一项需要考虑的风格问题就是Guard子句，这意味着应首先消除不常见乃至从未出现过的情况，通过这种方式，就可以忽略其他子句，更改后的代码如下：

```

if (!_CorrectBirthDate())
    throw new ArgumentException("Incorrect birthdate!");

```

或许更改为下面这样的形式更好：

```

if (_IncorrectBirthDate())
    throw new ArgumentException("Incorrect birthdate!");

```

我认为这更清晰、更顺畅。

需要关注不常见场景时，接下来的工作就是计算本身。如下所示：

```

int years = DateTime.Now.Year - BirthDate.Year;
if (DateTime.Now.DayOfYear < BirthDate.DayOfYear)
    years--;

return years;

```

此处再次使用了提取方法重构，将计算转移到名为\_NumberOfYears()的另外一个私有方法之中。完整的HowOld()方法如下所示：

```

//Person

```

```
public int HowOld()
{
    if (_IncorrectBirthDate())
        throw new ArgumentException("Incorrect birthdate!");

    return _NumberOfYears();
}
```

毫无疑问，我们能够发现更多任务（例如在新创建的私有方法 `_NumberOfYears()` 中，或从类本身中提取出某些逻辑），但至此我们已经理解了主旨。

有必要强调的是，应该在每一步完成后重新执行测试，确认未引入任何错误，至少以测试为依据无任何错误。

但我们的做法耗时而易于出错。使用大量规程有可能解决易出错的问题，但仍然耗时。我的意见是，如果确定了概念，但发现应用概念要耗费大量时间，则此时正是利用工具支持的最佳时机。

## 2. 清理工具

有多种重构工具可用。举例来说，如果关注.NET，就可以选择ReSharper [ReSharper]和Refactor! [Refactor!]，这两个都是Visual Studio .NET的插件。在Visual Studio .NET 2005中，IDE还提供了内置的重构支持。

这些重构工具实现了与上文类似的更改，以帮助我们提高工作效率。其价值不容低估。实际上，熟悉了重构工具之后，我们绝不会再放弃它！

工具不仅能帮助实现所需更改，还能告诉我们在何时需要做出哪些更改。因而，这些工具就能减轻工作负担，通过提供可视化的提醒使我们不必为某些必须考虑重构的规程而担忧。

---

**说明** 你是否记得，我曾经提到过失败的测试是红色的，而成功执行的测试是绿色的，但重构的需求没有指示颜色。实际上，重构工具ReSharper使用橙色来指示需要重构（根据ReSharper）。因此在开始一次新的迭代周期之前，应注意观察，通过清理代码使状态由橙色转为绿色，之后再继续工作。当然，这并不意味着完全不必考虑存在哪些代码错误，它仅仅是一种辅助。

---

如果参与流程的开发人员具备好的重构工具，那么工作速度将令人惊叹，也绝不会创建出不良的代码。这对代码质量大有裨益！

至此，我们探讨了一些简单的主题。当然，这里并没有任何问题，但我们会逐渐不满足于进行小规模清理。有时可能需要清理持续增加的一系列新问题。

## 3. 避免增长问题

或许你认为，使用模式自然是此类问题的解决方案。模式的问题在于，它们往往用于预先设计，这又意味着过多的猜测（而正如我们所讨论的那样，重构是在开发过程的后阶段应用的）。应该选择模式还是选择重构呢？

深入讨论这个问题之前，我认为有必要给出另外一个关于模式的简单示例。我希望讨论模板



方法模式[GoF Design Patterns]。主要思想是在如下所示的基类中定义整体算法或模板方法：

```
public abstract class TotalBase
{
    public void TotalAlgorithm()
    {
        _DoSomeStuff();

        VariationPoint();

        _DoSomeMoreStuff();
    }

    protected abstract void VariationPoint();
}
```

现在，子类可继承自TotalBase，并提供VariationPoint()方法的自定义实现。如下所示：

```
public class TotalSub : TotalBase
{
    protected override void VariationPoint()
    {
        Console.WriteLine("Hello world!");
    }
}
```

使用者代码不关心算法是怎样创建的。它甚至并不了解基类，而是仅仅知道子类。使用者代码应如下所示：

```
//Some consumer
TotalSub t = new TotalSub();
t.TotalAlgorithm();
```

注意，我们在基类中确定挂钩函数(hook)是可选的还是强制的。上一个示例选择了强制，因为我对VariationPoint()方法使用了abstract。如果使用的是virtual，就应该是可选的，例如希望使子类可以有选择地定义整体模板方法的一个片段时。

假设我们已经确定使用模板方法，这又带来了确定在何处允许或强制要求变体的问题。这是一个很难做出的决定。

然而，模式非常有用，我们不必做出选择，而是可以同时使用模式和重构。事实上，我认为这正是最合理的方法。通过重构得到模式，或者从模式进行重构！这就是Kerievsky在《重构与模式》[Kerievsky R2P]一书中探讨的主题。

下面通过第四个示例来看一下实际情况，它将与我们的刚刚讨论过的模式联系在一起。

#### ● 避免增长问题示例一：表单模板方法

假设我们编写了批处理监控程序，可在其中挂接不同的程序。要使一个程序成为批处理程序，必须根据某些规则加以调整，基本上就是（通过某些自定义的日志例程）告知批处理监控程序，程序是在何时启动和停止的。

经过一段时间之后，我们可能已经编写了多个批处理程序，还会决定为每个批处理程序重用基类，从而不必反复编写日志代码。子类中Execute()方法的示例如下：

```
//A subclass to the batchprogram baseclass
public void Execute()
{
    base.LogStart();

    _DoThis();
    _DoThat();

    base.LogEnd();
}
```

这里的不当之处在哪里？如你所见，子类负责在Execute()方法执行中的特定点调用基类。我认为这与子类的联系过于松散，子类的开发人员必须牢记添加这些调用，并在正确的位置进行调用。另一个不当之处（如果使用具有多个日志记录点的实际示例，这个不当之处会更加明显，例如介绍之后、第一个阶段之后、第二个阶段之后，等等）就是在核心方法中存在过多的基础架构代码，也就是日志调用。正因如此，对于感兴趣的方法的调用（例如\_DoThis()和\_DoThat()）未能像预期那样明显。

我认为这段代码急需模板方法。因此我们将使用称为表单模板方法[Kerievsky R2P]的重构。

基本思想是不在子类中使用公共Execute()，而是将这项责任转移给超类。首先将Execute()方法移动到超类。如下所示：

```
//The batchprogram baseclass
public void Execute()
{
    _LogStart();

    DoTheWork();

    _LogEnd();
}
```

\_LogStart()和\_LogEnd()方法现均为私有方法，而非受保护方法。DoTheWork()方法是按照以下方式定义的，目的在于强制子类实现该方法：

```
protected abstract void DoTheWork();
```

最后，子类中的DoTheWork()实现如下所示：

```
//A subclass to a batchprogram baseclass
protected override void DoTheWork()
{
    _DoThis();
    _DoThat();
}
```

这几乎是一种截然不同的方法。不是由子类向上调用超类，而是由超类向下调用子类。如果

变异点 (variation point) 合理, 这将成为一种极为强大的解决方案, 明确而简单! 子类可完全忽略基础架构 (如日志记录), 而仅仅关注最感兴趣的具体内容, 也就是实现 `DoTheWork()`。(同样请牢记这是一个简化版本。在作为这个简化示例的依据的实际情况中, 子类要实现多种方法, 而不是一种。)

另一个问题是对于状态模式[GoF Design Patterns]来说, 很难确定是否预先应用解决方案。如果在一开始就应用了解决方案, 往往需要重复设计。通常情况下, 最好等到能够确定确实存在问题的时候。第2章中已经详细介绍了状态模式, 我相信你已经对如何使用它有了充分的认识。

我们的讨论从重构过渡到了模式。在介绍模式时, 我也提到了重构。下面来看一个例子。

#### ● 避免增长问题示例二: 内联单体

单体模式[GoF Design Patterns]被越来越多的人视为最糟糕的实践。原因在于创建了全局实例, 全局实例本身就存在问题。另外一个问题是, 单体往往会导致难以编写测试 (切记, 可测试性十分重要), 代码中与单体的交互也不够明确。

假设有如下代码片段:

```
public void DoSomething()
{
    DoThis();
    DoThat();
    MySingleton.Instance().Execute();
}
```

称为内联单体的重构[Kerievsky R2P]不会采用与此完全相同的形式, 而是以略有差异的变体形式出现。

`DoSomething()` 方法不会调用全局变量 (单体本身), 与此不同, `DoSomething()` 方法将请求将实例作为参数。

另外一种典型的解决方案是实现 `DoSomething()`, 查找已在构建时作为私有成员注入的实例, 或在调用 `DoSomething()` 之前通过 `setter` 设定。 `DoSomething()` 应如下所示:

```
public void DoSomething()
{
    DoThis();
    DoThat();
    nowMyPrivateMember.Execute();
}
```

这样, 单体消失, 可测试性就高得多了!

---

**说明** 公平地说, 如果单体是由使用者注入的, 则可在不影响先前代码可测试性的前提下使用单体。随后可在测试过程中使用测试对象, 在运行时使用实际单体。

---

是否存在问题? 最明显的问题就是: 如果需要在调用栈中向下传递值, 参数就要经过漫长的传递, 之后才能使用。另一方面, 使用值的位置非常明确, 这本身可能也是不良代码的一个迹象, 表示了或许需要一定的重构。因而, 单体确实可能隐藏了不良的代码。

此外，依赖性注入（第10章将进一步讨论相关内容）可很好地解决单体过多的问题。

### 3.4 小结

总之，我想强调TDD加上重构等于“事实”。两者有着密切的依存关系。

为了以安全的方式使用重构，必须进行广泛的测试。如果未进行广泛测试，可能会引入bug，也可能因考虑到可维护性的问题而选择不做出任何改变，毕竟引入bug的风险太大。因可维护性的考虑而停止更改时，代码实际上就开始慢慢降级了。

同时，为了使用TDD，必须谨慎监视代码，通过应用重构保证它始终清晰明确。借助桩和模拟来隔离单元测试。如果未监视测试代码，代码也会开始降级。

与大多数人在学校里学到的方法相比，TDD加重构是一种截然不同的代码编写方法。亲手尝试吧，它将彻底颠覆你的世界。

掌握这些工具之后，我们也就为讨论架构做好了准备。



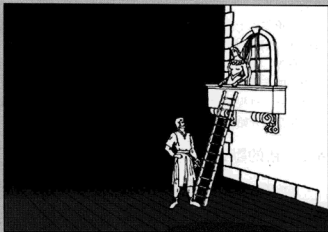
# Part 2

## 第二部分

## 应用 DDD

这一部分介绍 DDD 的应用。此外还会为基础架构准备领域模型，重点关注规则方面。

本部分是最重要的核心内容。



### 本部分内容

- 第 4 章 新的默认架构
- 第 5 章 领域驱动设计进阶
- 第 6 章 准备基础架构
- 第 7 章 应用规则

过去，我的应用程序架构一直以数据为中心。近来这种状况逐渐发生了改变，我目前的默认架构基于领域模型。这就是将本章命名为“新的默认架构”的原因。

你可能想问，对谁来说是新的？这取决于具体情况。向他人描述领域模型时，人们可能会说：“哦？事情不是一向如此吗？”与此同时，还会有一大批开发人员会说：“嗯……这在现实中确实有效吗？”

无论你属于上述哪种类型，我们都将踏上领域模型的探索之旅，特别将关注在领域驱动设计（DDD）的精神指导下如何构建领域模型。

首先，我们将简单探讨一下领域模型和DDD。为了使讨论更加具体，我们将讨论一个示例应用程序的需求列表，简要勾勒出针对不同特性的可行解决方案。为了更好地理解需求，我们还将使用另外一种视图并初步确定初始UI。本章最后将讨论领域模型的执行风格。

让我们开始吧。

## 4.1 新的默认架构的基础知识

如前所述，新的默认架构基于领域模型。第1章中讨论了大量与领域模型有关的内容，这里再简单说明一下。

---

**说明** Paul Gielen认为，对于本章章名中的“默认”一词，应该有所警示。风险在于，有些读者可能会认为这种方法适用于所有系统。我想强调的是事实并非如此，本书的目的并不是提供一种模板或者类似的东西。绝对不是那样！DDD的主旨就是由业务问题来控制解决方案的形式。

---

当在架构上下文中谈及“领域模型”时，其含义通常是使用领域模型模式[Fowler PoEAA]。这与过去的面向对象相似，在面向对象方法中，我们尝试将问题领域中的简化视图（或者更准确地说，是选定的抽象）尽可能接近地映射到面向对象模型。最终的代码非常接近于所选定的抽象。这一点同时适用于状态和行为，以及可能的导航路径和对象之间的关系。

说明 有人(很遗憾,我不记得究竟是谁了)曾经将C#中采用领域模型模式的应用程序称为“在C#中以Small-Talk的形式编程”。对于有些人来说,这种说法最初听上去似乎并不太准确,但事实的确如此。

领域模型并非银弹,但确实提供了一些积极的属性,特别对于长期使用的大规模和/或复杂应用程序来说更是如此。正如Martin Fowler所说的那样[Fowler PoEAA],如果在一个项目中开始使用领域模型,那么此后即便在小型、简单的应用程序中,也会希望使用它。

#### 如何识别一个应用程序是否将长期使用

提到长期使用的应用程序,我总会想起我曾被要求构建的一个应用程序。一家全球数据通信提供商要求我构建一个基本帮助台应用程序,用于他们在瑞典的新办公室。时间十分紧迫,他们要求在第二天完成此应用程序,这样才能保证于媒体前来参加开幕活动时,不必在高科技办公室内使用纸笔记录。由于此应用程序在短短几周之后将会被企业策略标准应用程序所取代,因而他们告诉我没有必要将它构建得十分完美。

你或许已经猜到了。我匆忙构建的这个并不完善的应用程序使用了5年之久。当然,我们改进了很多,但我无法说服他们将最初使用的平台改为更适合任务关键型应用程序的技术。

#### 4.1.1 从以数据库为中心过渡到以领域模型为中心

我从以数据库为中心过渡到以领域模型为中心的主要原因是我的侧重点从效率改变为维护。并不是新的设计风格必须牺牲效率,但我尽力避免出现过细而又不成熟的优化,因为此类优化往往要付出过于高昂的代价。我尝试尽可能明确地建立领域模型,这使得在真正需要的情况下更容易实现优化。我并未遗忘数据库,只不过不再将它作为第一关注点。我竭力找到更合适的折中方法,因此我确实认为有必要巧妙处理数据库。

从长远的角度来看,以领域模型为中心的设计更加清晰,也是一种更忠实于领域抽象的实现,因而可维护性更高。另外一个非常重要的原因是,强大的领域模型是减少逻辑重复的优秀工具。(面向对象也具有这些特征。我将领域模型视为一种风格,其中以一种更加纯粹的方式来使用面向对象技术,竭力编写可维护的代码,竭力提高可测试性)。

听到“领域模型模式”[Fowler PoEAA]这个词时,你或许有非常具体的认识,但毫无疑问,这种模式的应用方法多种多样。让我们将这些不同的方法称为风格。

#### 4.1.2 进一步关注 DDD

这将我引向了我目前偏爱的领域模型风格的主旨,也就是Eric Evans提出的DDD[Evans DDD]。

第1章已经详细介绍过,DDD有一些不同寻常之处,例如,它是一组模式,用于帮助建立领域模型。很快我们就会开始实际应用这些工具。

但在深入细节之前,先简单介绍一下根据DDD进行分层的知识。

### 4.1.3 根据 DDD 进行分层

在我的上一本书中[Nilsson NED], 详细介绍了一种严格的分层模式, 我认为有必要说明一下我如今对于分层的看法。

一方面, 我仍然非常重视分层。对DDD的着迷使我竭力尝试将基础架构(例如持久化)从核心领域层中移出, 移到它自己的层中。

另一方面, 我对于分层的观点比过去宽松得多, 主要关注的是一个层: 领域层。我不会将领域层分割成更细的层, 而是使之保持粗粒度。(实际上, 某些DDD模式能够方便地处理过去使用分层处理的一些问题。)

至此, 我们已经提到了两个层: 基础架构层和领域层。在这两个层之上就是提供一些协调功能的应用程序层, 但此层非常简单, 仅是将工作委托给领域层。(参见服务层模式[Fowler PoEAA], 它与场景类相似, 将所有工作委托给领域模型。)

最后, 最顶层就是用户界面层。

我绘制了一个简单的分层模式示例图, 参见图4-1。

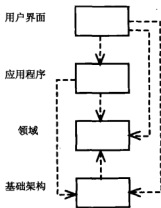


图4-1 DDD项目的典型分层

**说明** 再次强调, 不要仅仅因为在本书中读到了一些东西, 就机械地根据分层进行一些操作!

我们应该置疑每一项决策!

关于我现在使用分层的方式与过去的使用风格之间的差别, 我认为有几个方面需要说明。首先, 应用程序层并非必不可少, 只有在确实能够增加价值的时候才使用该层。由于我的领域层比过去丰富得多, 因而对于使用应用程序层往往并不是很感兴趣。

另外一个差别就是基础架构层不再向下传递所有调用, 而是“了解”领域层, 并在从持久化状态进行重建时创建实例。要点在于, 领域模型应忽视基础架构。



说明 另外还有必要说明的是,除了分层之外,我们并未考虑分区或在其他维度将切片比作分层。这对于大型应用程序非常重要。我们将在第10章中再次讨论相关内容。

现在该到开始旅程的时候了,我们将尝试一些在本章前面和前几章中讨论过的概念,重点关注领域层。我认为最好的方法就是设计一个示例。

## 4.2 轮廓

为了使这个示例更加具体,我们将使用一个问题/特性列表,讨论如何解决订单应用程序的某些常见设计问题。

我知道,如何通过我目前偏爱的领域模型风格处理特性列表,取决于多种因素。上文已经提到,“默认架构”这种说法有些奇怪,因为架构首先取决于手头的问题。

但这里仍然要讨论一种可能的解决方案。我要给出的是一个最基本的轮廓,在后续章节中逐渐补充细节。此处讨论的内容类似于初次尝试,我将一切保留原样,随着后续章节的介绍逐步推进设计。

### 4.2.1 领域模型示例的问题/特性

后续章节将重用这组需求列表来讨论某些概念。下面让我们来深入了解一下每个问题/特性的具体内容(随机排序)。

(1) 应用灵活但复杂的过滤器来列出客户。

客户支持员工需要能够以一种非常灵活的方式搜索客户。他们需要在各种字段中使用通配符,例如姓名、位置、街路地址、介绍人等。他们还需要能够根据特定类型的订单、特定规模的订单、特定产品的订单等条件搜索客户。这里探讨的是一种功能全面的搜索工具。最终得到的结果是带有客户编号、客户姓名和位置的客户列表。

(2) 在查看特定客户时列出订单。

在列表中应该可以查看每个订单的总价,以及订单的状态、类型、订单日期和介绍人的姓名。

(3) 一份订单可具有多个不同的行。

一份订单应有多个订单行,每一行描述一种产品和该产品的订购数量。

(4) 并发冲突检测非常重要。

使用乐观并发控制确实可行。也就是说,在用户完成了一些工作并尝试保存时,通知用户此操作与之前的保存操作冲突,这种方法是可接受的。只有真正导致出现不一致的冲突才应被视为冲突。因此解决方案需要为客户和订单确定版本控制单元。(这会对其他特性造成轻微的影响。)

(5) 客户对我们的应付款项不能超过一定的额度。

限制是特定于每位客户的。我们在最初添加客户时定义限制,此后可以更改限制。如果存在总价超过限制的未付款订单,就应视为出现了不一致,但在一种情况下允许存在这样的不一致,即在用户降低限制时。降低限制的用户将得到通知,但允许执行保存操作。然而,如果添加或修

改订单操作会导致超出限制，则此类操作就不能实现。

(6) 订单的总价不能超过100万SEK（SEK是瑞典货币，本例中将使用这种货币）。

此限制（不同于上一条限制）是系统级的规则。

(7) 每份订单和每位客户都应有一个唯一、用户友好的编号。

序列号中允许有间距。

(8) 在一位新客户被视为可接受之前，必须联系信用卡机构检查客户的信用。

也就是说，此时应检查第(5)步中提到的为客户定义的限制，确定是否合理。

(9) 一份订单必须有一位客户，一个订单行必须对应于一份订单。

不得有任何未定义客户的订单。订单行也是如此：订单行必须属于一份订单。

(10) 保存订单后，其订单行应该是原子的。

实际上，我并不确信此特性是否是必要的。如果首先创建了一份订单，随后又添加了订单行，那么可能无需此规则，但我希望在本示例中应用这条规则，以使示例中拥有与交易保护相关的特性。

(11) 订单具有可由用户更改的接受状态。

用户可将此状态更改为不同的值（例如批准/否决）。要更改为其他状态值，可通过领域模型中的其他方法隐式更改。

这样，我们就得到了一份好的、简单的特性列表，可以使用这份列表从整体角度简要讨论解决方案。

---

**说明** 你或许感到特性列表过于关注数据。在某些情况下，在实践中可以通过具有某些行为的属性来解决过分关注数据的问题。

但在这里有必要说明一下，我们希望应用的是领域模型。主要问题在于，如果使用关系数据库，应如何处理数据。因而，必须比在面向对象方法中更加关注数据。

---

确定了问题/特性之后，就可以讨论处理这些问题的可能解决方案了。

#### 4.2.2 逐个处理特性

所有这些特性能归结为什么结果？让我们通过探讨解决本章前面列举的问题/特性的常见方法来了解这一点。这里只关注领域模型，从而将精力集中在所讨论的“正确”事情上，尽力捕获和形成通用语言，而不会被基础架构或其他层分散精力。

同样，最好对技术环境有一定的了解。这里将技术复杂度降低，确定上下文是在客户端面执行的富GUI应用程序，外加领域模型以及使用关系数据库的物理数据库服务器，所有这一切都在LAN上运行。

---

**说明** 下文将提到大量模式，但请注意，我们会在后续章节中回头讨论这些内容并提供大量细节（参见附录B）。

---

就这样简单……

### 1. 应用灵活但复杂的过滤器来列出客户

第一项需求关注的主要是数据，以及一部分搜索行为。首先我们简单说明一下Customer类及其环境（目前没有Order）的形式（如图4-2所示）。

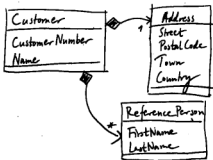


图4-2 Customer类及其环境

**说明** 无论你是否相信，使用手绘图是有理由的。我展示的是早期的草图，这只是我认为可以从这里入手的基本想法。代码是可执行的、简洁的工件。代码是真实模型的最重要的表示。

如你所见，我使用了一个简单的（自然的）Customer类，它由Address和ReferencePersons组成。我不喜欢将此类搜索行为直接置于Customer类本身当中，因而此时类中只有如图4-2所示的数据。

**说明** 我将使用Party Archetype模式[Arlow/Neustadt Archetype Patterns]或其他类型的角色实现，但目前先使用简单的结构。

我们利用查询对象[Fowler PoEAA]来解决第一项需求，第2章已经介绍了相关内容，但在实践中，我发现最好尽可能封装查询对象的使用。关于查询的结果，有许多与标准无关的方面需要定义。我考虑了排序、其他隐藏标准、优化、将查询对象发送到何处来执行，等等。因而选择使用存储库[Evans DDD]来封装查询的执行。这还能减少使用者所用的代码长度，提高编程API中的明确性。

在第2章中，我们讨论了工厂模式[Evans DDD]，提到过它与实例生命周期的开始有关。存储库则处理实例生命周期的其余部分，从创建之后一直到实例的“消亡”。例如，当想要获取过去已经保存的实例时，存储库将会把数据库连接到领域模型。存储库将使用基础架构来完成其任务。

我可以令存储库上的方法接受一个庞大的参数列表，每个参数对应于一个可能的过滤器字段。这实际上预示着不良代码：不够清晰、不易维护的代码。此外，存储库难以理解应如何使用

字符串类型的“Name”参数，至少很难理解在字符串为空时应如何使用。这是否意味着正在查找姓名为空的客户？还是表示我们并不关心姓名？确实，我们可以发明一个“有魔力”的字符串，表示查找姓名为空的客户。或者更好一些，强制要求用户给他们不感兴趣的姓名添加星号通配符。但无论哪种方法都不是理想的解决方案。另一方面，从查询对象[Fowler PoEAA]入手，特别是特定于领域的查询对象或规格模式[Evans DDD]，是一种有些过分、速度过快的方法，不是这样吗？让我们尽可能保持简单，仅考虑两种可能的标准。图4-3提供了一个这样的例子。

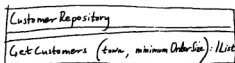


图4-3 灵活处理标准的CustomerRepository

**说明** 当然，可以在存储库中使用查询对象。

DDD和TDD是一种极优秀的组合。通过测试代码试验模型并获得即时反馈是获得深入见解的一种好方法。因此，我不会立刻采用TDD路线。我将通过一些简单的测试来解释模型概要，但只是为了提供解决方案思想的另一种视角。下一章将返回来更彻底地尝试真正的TDD方法，并开发模型。

因此，为了获取某个城镇（Ronneby）中至少具有一份价值大于1 000 SEK订单的所有客户，可以使用以下测试代码：

```
[Test]
public void
    CanGetCustomersInSpecificTownWithOrdersOfCertainSize()
{
    int numberOfInstancesBefore = _repository.GetCustomers
        ("Ronneby", 1000).Count;

    _CreateACustomerAndAnOrder("Ronneby", 20000);

    Assert.AreEqual(numberOfInstancesBefore + 1
        , _repository.GetCustomers ("Ronneby", 1000).Count);
}
```

正如示例所显示的那样，我们使用了存储库方法GetCustomers()来获取满足城镇和最低订单规模的标准的所有客户。

## 2. 在查看特定客户时列出订单

我希望在Customer和Order之间建立双向关系。也就是说，每个Customer都具有一个Order的列表，每个Order都具有一个Customer。但双向关系的代价较高，包括高的复杂性、紧密耦合和开销。因而，我认为能够通过OrderRepository获得Customer的Order就足够好了。这就得到了如图4-4所示的模型。

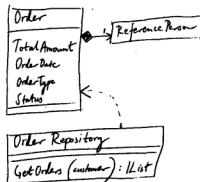


图4-4 OrderRepository、Order及其环境

因此在使用者希望查看特定Customer的Order时，必须查询OrderRepository，代码如下：

```

[Test]
public void CanGetOrdersForCustomer()
{
    Customer newCustomer = _CreateACustomerAndAnOrder
        ("Ronneby", 20000);

    IList ordersForTheNewCustomer =
        _repository.GetOrders(newCustomer);

    Assert.AreEqual(1, ordersForTheNewCustomer.Count);
}
  
```

我希望令Customer具有一个OrderList属性，并隐式地与存储库对话，但我认为在这里使用显式方法更好。我还避免了Customer与OrderRepository之间的耦合。

值得单独提一下的是每个订单的总价值。这或许比最初的预计更加复杂，因为计算需要使用order的所有orderLine。确实，这并不复杂，但的确警示了我。我知道这还不成熟，但无法避免（举例来说，在需要为某位客户显示庞大的一组订单时，载入orderLine的成本高昂）。我认为一种有用的策略是在orderLine未载入时，为TotalAmount属性使用一个简单的总计字段，该字段可能存储在Orders表中。如果orderLine已载入，则为TotalAmount属性使用完整的计算。

这并不是我们目前需要考虑的问题。无论如何，对于使用者来说，这段代码非常简单，如下所示：

```

[Test]
public void CanGetTotalAmountForOrder()
{
    Customer newCustomer = _CreateACustomerAndAnOrder
        ("Ronneby", 420);

    Order newOrder = (Order)_repository.GetOrders
        (newCustomer)[0];
}
  
```

```

Assert.AreEqual(420, newOrder.TotalAmount);
}

```

在我看来，考虑这个问题的“正确”方式是：订单行已经存在，因为我们使用了一个聚合，而且订单行是订单的完整组成部分。这或许意味着，性能在此类情况下会有所降低。如果是这样，可能必须通过延迟加载模式[Fowler PoEAA]（例如，用于实时从数据库中载入清单）或只读的优化视图来解决这一问题。

如前所述，当发现简单而直接的解决方案不够好时，就要采用这样的方法来处理。

### 3. 一份订单可具有多个不同的行

第三项特性非常直观。图4-5描述了增强后的模型。

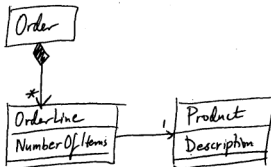


图4-5 Order和OrderLine

注意，这里也考虑到了非双向的关系。如果确实需要特定逻辑，那么我或许也可以容忍同时发送Orderline及其Order。将一份订单与其他订单的订单行一起发送的风险是很小的。

```

[Test]
public void CanIterateOverOrderLines()
{
    Customer newCustomer = _CreateACustomerAndAnOrder
        ("Ronneby", 420);

    Order newOrder = (Order)_repository.GetOrders
        (newCustomer)[0];

    foreach (OrderLine orderLine in newOrder.OrderLines)
        return;

    Assert.Fail("I shouldn't get this far");
}

```

另外还要注意，图4-5所显示的模型的形式与描述相同事物的关系模型的形式之间存在的明显区别。在关系模型中，Product和OrderLine之间应该存在一对多的关系和多对一的关系，但目前为止，我们在这个特定的领域模型中并未考虑一对多的关系。

#### 4. 并发冲突检测非常重要

特性4有些复杂。在这里，合理的解决方案应该是，Customer是一个独立单元，Order与OrderLine是其本身的并发单元。这样的组合使之能够为后面的特性所需的逻辑使用领域模型。我为此使用了称为聚合[Evans DDD]的模式，这也就意味着要决定哪些对象属于特定对象集群，该集群通常要作为一个单元来处理，一起载入（在默认情况下，至少在写入场景中需要一起载入），使用共同评估规则等。

但有必要说明，首先，也是最重要的，聚合模式并不是一种技术模式，而应该在它对模型有意义的时候才使用这种模式。

并发单元是聚合的功用之一（参见图4-6）。

#### 5. 客户对我们的应付款项不能超过一定的额度

特性5非常有趣。我们脑中浮现出的第一种解决方案可能是为Customer添加TotalCredit属性。但这种方法存在问题，因为过于透明了，使用者在调用属性时看不到代价。它也存在一致性的问题。我甚至不认为Customer聚合具有Order，因此也不会认为在领域模型中直接处理一致性时，该单元是正确的选择，这暗示了应该寻求另一种解决方案。

我认为领域模型中应该使用服务[Evans DDD]来表明某位客户的当前总贷款。图4-7展示了这种方法。

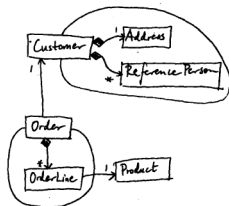


图4-6 聚合

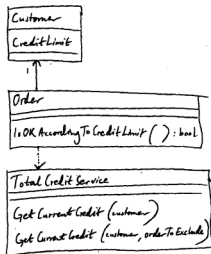


图4-7 TotalCreditService

TotalCreditService的GetCurrentCredit()重载的原因是,能够很好地检查当前的贷款额,而无需考虑当前正在创建或修改的订单。至少这是一种合理的想法。该服务的实际应用如下所示(此处忽略不同片段之间的交互)。

```
[Test]
public void CanGetTotalCreditForCustomerExcludingCurrentOrder()
{
    Customer newCustomer = _CreateACustomerAndAnOrder
        ("Ronneby", 22);

    Order secondOrder = _CreateOrder(newCustomer, 110);

    TotalCreditService service = new TotalCreditService();
    Assert.AreEqual(22+110
        , service.GetCurrentCredit(newCustomer));
    Assert.AreEqual(22,
        service.GetCurrentCredit(newCustomer, secondOrder));
}
```

现在,你应该了解了Order、Customer和图4-7及上述代码片段所示服务之间的协作思想。

## 6. 订单的总价不能超过100万SEK

由于我已经确定了并发冲突检测特性,即Order(包括其OrderLine在内)是其自身或更加概念性的聚合的并发冲突检测单元,因而可以在领域模型中处理此特性。聚合不变量是订单的价值必须小于或等于100万SEK。其他任何用户都不得干涉,根据这条规则为某个定单创建出了问题,否则我或其他用户会遇到并发冲突问题。因此我在Order上创建了IsOKAccordingToSize()方法,使用者可随时调用此方法。参见图4-8了解更多信息。

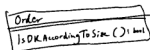


图4-8 Order与IsOKAccordingToSize()方法

下面这段简单的测试代码展示了API的形式。

```
[Test]
public void CanCheckThatAnOrdersTotalSizeIsOK()
{
    Customer newCustomer = _CreateACustomerAndAnOrder
        ("Ronneby", 2000000);

    Order newOrder = (Order)_repository.GetOrders
        (newCustomer)[0];

    Assert.IsFalse(newOrder.IsOKAccordingToSize());
}
```

说明 随后需要考虑是否应为此类规则使用规格模式[Evans DDD]。



### 7. 每份订单和每位客户都应有一个唯一、用户友好的编号

作为一名开发人员，我非常不喜欢这项需求，但从用户的角度来说，这项需求十分重要和常见。在第一次迭代中，数据库就能很好地处理这项需求。举例来说，在SQL Server中，我会使用IDENTITY。除了在实体被插入数据库表中之后刷新实体之外，领域模型不用执行任何其他操作。在此之前，领域模型可能会为OrderNumber和CustomerNumber显示0（参见图4-9）。

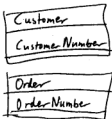


图4-9 增强的Order和Customer

哦……我已经对这样的草案感到有些后悔了。这意味着在Order初次被保存时，就需要立即分配OrderNumber。我并不确定这是个好的想法。

实际上，我融合了两个独立的概念，也就是用于将实例耦合到数据库行的标识字段模式[Fowler PoEAA]（通过将数据库行的主键用作对象中的值）以及具有业务含义的ID。有时这两者是相同的，但在大多数情况下应该是两个不同的标识符。

在客户被保存时，将获得用户友好的编号作为标记，我认为这也有些奇怪。不应该是客户在系统中操作时才分配编号吗（在检查过信用，或许是在手工批准之后）？我确信有理由在后面的章节中重新讨论这个问题。

### 8. 在一位新客户被视为可接受之前，必须联系信用卡机构检查客户的信用

领域模型中还需要另外一个服务[Evans DDD]，它封装我们与信用卡机构的通信方式。图4-10展示了该服务的形式。

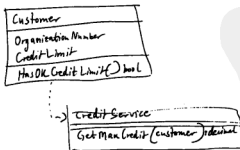


图4-10 CreditService

同样，这里的思想是实体（本例中为Customer）和服务之间进行协作，以便让实体回答问

题。让我们来看一下如何在测试中处理这样的协作。(另外还要注意,实际的服务可能使用Customer的OrganizationNumber,它与CustomerNumber不同,而是权威机构为注册企业提供的正式标识。)

```
[Test]
public void CantSetTooHighCreditLimitForCustomer()
{
    Customer newCustomer = _CreateACustomer("Ronneby");

    //Inject a stubbed version of CreditService
    //that won't allow a credit of more than 300.
    newCustomer.CreditService = new StubCreditService(300);

    newCustomer.CreditLimit = 1000;

    Assert.IsFalse(newCustomer.HasOKCreditLimit);
}
```

**说明** Gregory Young指出,这是一种不良的编码风格。问题在于,操作很有可能不是原子的。首先设置值(从而覆盖旧值),然后再进行检查(如果能记住的话)。或许Customer.RequestCreditLimit(1000)这样的代码是更好的解决方案。在第7章中,我们将回头讨论这个问题。

### 9. 一份订单必须有一位客户,一个订单行必须对应于一份订单

特性9是一个常见而合理的需求,我认为这非常简单,最好通过数据库中的引用完整性约束来处理。我们能够,也应该在领域模型中进行测试。如果存在这样明显的错误,那么将领域模型更改发送到持久存储区也毫无意义。但我没有选择进行检查,因为通过初次尝试,它已经变成一种强制性操作,这得益于将OrderFactory类用作创建Order的方式,而且我采用相同方法处理了OrderLine,因而OrderLine一定具有Order和Product(参见图4-11)。

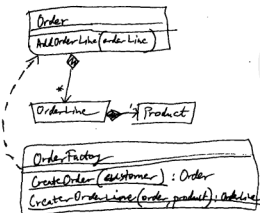


图4-11 OrderFactory和增强的Order

接下来看一下测试中不同部分之间的交互。

```
[Test]
public void CanCreateOrderWithOrderLine()
{
    Customer newCustomer = _CreateACustomer("Karlaskrona");

    Order newOrder = OrderFactory.CreateOrder(newCustomer);

    //The OrderFactory will use AddOrderLine() of the order.
    OrderFactory.CreateOrderLine(newOrder, new Product());

    Assert.AreEqual(1, newOrder.OrderLines);
}
```

嗯……这似乎是一种过度的设计，不够流畅。下一章将介绍实践中的想法。

尽管现在讨论的是OrderFactory，但顺便提一下，我很欣赏为OrderType、Status和ReferencePerson使用Null对象[Woolf Null Object]的思想。（这既适用于领域模型，实际上也适用于基本的关系数据库。）Null对象模式意味着使用空实例而非null（空实例表示成员的默认值，例如对应于字符串的string.Empty）。通过这种方法，总是可以确保能够“follow the dots”，如下所示：

```
this.NoNulls.At.All.Here.Description
```

涉及数据库时，可以减少外部连接，而更频繁地使用内部连接，因为外键至少指向null符号，而外键列不能为null。总而言之，null对象会以无法预料的方式大幅度提高复杂程度。

如图4-11所示，我还为Order类添加了AddOrderLine()方法，用于为Order添加OrderLine。这是封装集合重构[Fowler R]实现的一部分，主要表示父类应保护对集合的所有更改。

另一方面，数据库处于最远端，我认为这项规则也适用于本例。

#### 10. 保存订单后，其订单行应该是原子的

我也将Order及其OrderLine视为一个聚合，我计划为此特性使用的解决方案就是以此为依据的。

我将使用操作单元模式[Fowler PoEAA]的实现来跟踪已经被更改的实例，哪些实例是新建的，哪些实例被删除。随后，操作单元协调这些更改，在保存过程中使用物理数据库事务。

#### 11. 订单具有接受状态

如前所述，订单具有接受状态（参见图4-12）。因而，只需添加Accept()方法。这里把是否在内部使用状态模式[GoF Design Patterns]的决策留到后文讨论。最好在重构期间制订这样的实现决策。



图4-12 Order和Status

讨论Order的状态时，还要添加更多方法。目前，并无明显的需求，因而只讨论Accept()。以下代码展示了当前思想：

```
[Test]
public void CanAcceptOrder()
{
    Customer newCustomer = _CreateACustomer("Karlskrona");
    Order newOrder = OrderFactory.CreateOrder(newCustomer);

    Assert.IsFalse(newOrder.Status == OrderStatus.Accepted);

    newOrder.Accept();

    Assert.IsTrue(newOrder.Status == OrderStatus.Accepted);
}
```

### 4.2.3 到目前为止的领域模型

如果总结上面讨论的领域模型，它应如图4-13所示。

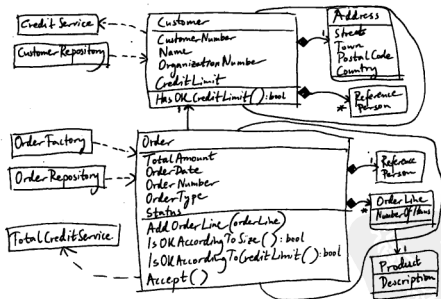


图4-13 到目前为止，能实现特性列表需求的领域模型草图

**说明** 在图4-13中，ReferencePerson类位于两个不同的聚合中，但实例并非如此。这个例子既表明了静态类图的表现能力不足，也表明了通过简短注释来简单解释问题。

那是什么？一片混乱？我同意，确实如此。在深入讨论细节时，我们将更好地分解此模型。

**说明** 在图4-13和所有代码片段中展示的模型是预先创建的，继续开发应用程序时，它们将绑定在一起，进一步改进。

另外需要注意的一点是，我仅展示了领域模型的核心部分，而未包含与基础架构有关的部分，例如上面提到的操作单元[Fowler PoEAA]。当然，这是有意而为。在本书的后文中，我们将回头讨论此问题。现在关注领域模型。

前面讲过，领域模型模式有很多种使用方式。为了展示一些不同的风格，我邀请几位朋友描述了他们在应用领域模型时常用的方法。附录A提供了相关内容。

为了更好地理解需求，我将从另外一个角度观察这些需求，并为后面的UI拟定一些窗体。

## 4.3 初次尝试将 UI 与领域模型挂接

我曾经自下而上地介绍过一种新架构。我的意思是，从数据库开始介绍架构。一位读者明确地告诉我，应该从UI程序员的角度开始讨论该架构的形式。如果从这个角度看不是很好，那么就没有必要再读下去了。我确信，这种思路是有价值的，因此现在要从UI程序员的视角观察上面简单介绍过的领域模型。

### 4.3.1 基本目标

我希望你考虑一下，我们是否满足了我的一个基本目标，也就是“为左层提供简单性”（或者“为顶层提供简单性”，具体取决于你如何看待分层架构中的各层）。也就是说，为UI程序员提供简单的API，使他们能够轻松查看、感受和理解模型，并且能够关注UI问题，而不必去考虑领域模型的复杂协议。

忽略数据绑定绝非基本目标，但在初次讨论UI时，我希望暂时忽略数据绑定。数据绑定并非本书的关注点，但在后续章节中将简单介绍。

### 4.3.2 简单 UI 的当前焦点

我认为，目前的领域模型有些抽象，但从UI的角度进行讨论能使这样的情况有所改观。我们应该尝试以下操作。

- 为客户列出订单。
- 添加一份订单。

同样，这里不会使用常规的数据绑定，而是使用简单而直接的代码将UI挂接到领域模型。

### 4.3.3 为客户列出订单

我需要为客户列出订单。随后在构建领域模型时，只需在领域模型之上添加“视图”，再为一些Order创建伪类即可。我设想的窗体如图4-14所示。

Order #	Date	Total Amount
42	2005-05-17	57 000 SEU
314	2005-05-22	12 000 SEU

图4-14 为客户列出订单

**说明** 你可能认为我将开始讨论工具（VS.NET或其他类似的窗体编辑器），但由于我讨论的是初具想法的窗体，因而决定使用纸笔来表达我的想法。（但在本书中，我会提供在图形处理程序中呈现的图像。）

这非常简单直接。为使之能够测试和显示，需要编写一个函数，该函数可从Main()中调用，为客户和一些订单创建伪类。

用户可能会从客户清单窗体中选择客户，我们要为窗体的构造方法提供customer实例，以显示customer的细节（下面将此窗体称为CustomerForm）。该窗体在私有字段(\_customer)中存储customer实例。

该窗体的\_PaintCustomer()方法中的代码如下：

```
//CustomerForm
private void _PaintCustomer()
{
    //Get the data:
    IList theOrders = _orderRepository.GetOrders(_customer);

    //Paint the customer:
    txtCustomerNumber.Text =
        _customer.CustomerNumber.ToString();
    txtName.Text = _customer.Name;

    //Paint the orders:
    foreach (Order o in theOrders)
    {
        //TODO...
    }
}
```

新华书店  
PDF

应该指出的是，在构建订单时，我希望在前述代码中使用Null对象[Woolf Null Object]。因此，所有订单都具有ReferencePerson（空字段），即便尚未决定也是如此，不必进行任何null检查。我认为，这样一个细节示例正展示了如何使UI程序员的工作更加简单。

必须手动编写代码来填充UI，这项任务有些繁琐。但回顾上文会发现，它阅读起来十分简单。确实如此，我们尝试完成的是一个简单但静态的窗体。

#### 4.3.4 添加订单

另外一个窗体示例就是使之能够添加订单。假设首先选择客户，随后得到如图4-15所示的窗体。

Order #	<input type="text"/>	
Date	<input type="text"/>	
Customer #	<input type="text"/>	
Name	<input type="text"/>	
Chosen	<input type="text" value="v"/>	
Ref Person	<input type="text"/>	
Order type	<input type="text" value="v"/>	
Status	<input type="text"/>	
Total Amount	<input type="text"/>	
<input type="button" value="Add line"/>		
Product	Number of Items	Price for each
<input type="text"/>	<input type="text"/>	<input type="text"/>
<input type="text"/>	<input type="text"/>	<input type="text"/>
<input type="text"/>	<input type="text"/>	<input type="text"/>
<input type="button" value="Save"/> <input type="button" value="Close"/>		

图4-15 用于添加订单的窗体

要添加订单，首先实例化Order，然后将该order发送给窗体的构造方法。

\_PaintOrder()的代码如下：

```
//OrderForm._PaintOrder()
txtOrderNumber.Text = _ShowOrderNumber(_order.OrderNumber);
txtOrderDate.Text = _order.OrderDate.ToString();
//And so on, the code here is extremely similar in
//principle to the one shown above for _PaintCustomer().
```

为了避免txtOrderNumber出现零值，使用帮助方法(\_ShowOrderNumber())，显示“New”而非0。

### 4.3.5 刚才我们看到了什么

描述此类UI的一种方式是使用Fowler的分离表示模式[Fowler PoEAA2]。思路是将操纵表示的逻辑与代码分离开来。这是长期以来一直备受推崇的一条基本原则，但往往会被滥用。我们能够在一定程度上自动而无代价地实现此目标，因为我们使用的是DDD。

这一节旨在更好地理解需求，从另一个角度——UI来考虑这些需求。从现在开始，直到第11章为止，我们不会讨论与UI有关的更多内容。

下面从另外一个角度来考虑需求。

## 4.4 另一个维度

到目前为止，本章主要讨论了领域模型的逻辑结构，当然，这只是整个问题的一个维度。还有其他维度。下面讨论一些不同的执行风格。首先讨论为特性清单而设定的部署环境。上文提到，我们要创建的是无应用服务器的富客户（WinForms），它可能类似于图4-16所示的。

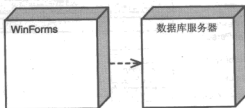


图4-16 WinForms，无应用服务器

为了更好地讨论，这里暂时增加一点复杂程度，添加应用服务器需求，如图4-17所示。

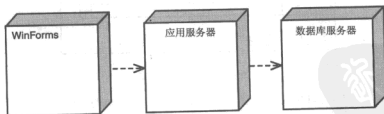


图4-17 WinForms和应用服务器

第一个问题是：领域模型应在哪个层（或者哪些层）上执行？应该仅在教育服务器上执行，随后为客户层提供哑数据结构，数据传输对象（Data Transfer Objects, DTO）[Fowler PoEAA]？应仅在使用者一端执行领域模型，通过从应用服务器查询DTO来填充？应同时在两个层上执行领域模型？还是使用两个不同的领域模型？



**说明** 考虑应用服务器的目的是很重要的：是否真正需要应用服务器，是否真正需要分布领域模型。应牢记Fowler的分布式对象设计第一法则：不要分布对象！[Fowler PoEAA]

暂时假设领域模型应该（至少）在应用服务器上执行。是否应使用共享的领域模型实例化，以保证有一个实例来表示被所有用户使用的一个客户？还是应为每位用户或每次会话时使用一个领域模型实例化，从而有多个实例在特定的时间点表示一位特定客户？

第三个问题是，领域模型实例化在各次调用间是应该有状态的，还是应该在每次调用后释放？

第四个问题是，我们是否应该在每次请求时获取尽可能多的数据，并且一直不释放实例，从而尝试建立完整的领域模型实例化？

问题有很多，答案总是取决于具体情况。要讨论的问题之一就是尚未决定关于基础架构的过多细节（以DDD的方式）。

无论如何，我想简单讨论一下我通常采用的处理方式。

**说明** “领域模型实例化”这个词表示的是领域模型的一组实例，而不是它的类。当然，我们往往是在用户（而不是实例）之间共享类。但我认为这个术语能使讨论更加清晰。

#### 4.4.1 领域模型的位置

首先，如果能够控制使用者和应用服务器，那么最好在使用者和应用服务器端同时公开和使用领域模型。如果未在这两个地方同时使用领域模型，就存在一定的风险，可能要完成不必要的工作，并且能力可能被削弱，因为我们要创建两个类似但略有差异的领域模型，一个用于使用者，一个用于应用服务器。我们还需要使用适配器，以便能够在两个领域模型之间进行转换。（在这里，最重要的一点或许就是应该谨慎决定在哪种情况下使用一个模型，在哪种情况下使用两个模型。如果认为应该使用一个模型，但在实际中是两个模型，那么可能就会遇到难解的问题。）

值得一提的是，对于复杂的场景，我更倾向于使用表示模型[Fowler PoEAA2]作为领域模型的UI优化视图或版本。

#### 4.4.2 孤立或共享的实例

在应用服务器上，是否应为每位用户或每次会话使用共享的领域模型实例化？在理论上，我更喜欢共享领域模型实例化的思路，但在实践中，我常常避免这种做法。这种做法比预想的要复杂得多，因此我会采用为每位用户使用领域模型实例化的方式，或者通常为每个会话使用领域模型实例化。这样做更简单。

如果被实例化的领域对象的共享集必须以分布方式执行，例如在“不共享任何内容的”集群中的两台应用服务器上执行，那么就会出现一个严重的问题。我们要应对分布式缓存的问题，这十分复杂（这只是保守的说法），至少在需要实时一致性的时候是这样。由于问题过于复杂，因

而目前暂不考虑此问题，只要找到一种能够很好地向上扩展的通用解决方案即可。但在特定场景中，可以找到足够好的解决方案。当然，如果令所有领域模型的实例都驻留在内存中，那么需要向外扩展的情况就更少，至少从效率方面考虑是这样。但如果遇到问题，就必然是一个难题，需要通过巧妙方法来解决。

---

**说明** “不共享任何内容的”集群这个词需要略加解释。这里指的是应用服务器不共享CPU、磁盘或内存。两台应用服务器之间彼此完全独立，这对可伸缩性十分有益。

---

要了解更多内容，推荐阅读Pfister的*In Search of Clusters*[Pfister Wolfpack]。

### 4.4.3 有状态或无状态领域模型实例化

在应用服务器上，我不希望领域模型实例化是有状态的，并且在各次请求之间释放实例。另一方面，在使用者一端，我尝试在各次请求之间保持领域模型实例化，但在用例完成后通常会释放实例。

### 4.4.4 领域模型的完整实例化或子集实例化

最后，我不会尝试实例化整个领域模型。在使用者一方，如果数据库较大，那么这种做法十分愚蠢。我会使用过去的思考方法，即仅获取必需的内容。在完成任务时释放数据，为其他数据留出空间，不会保留旧数据过长的时间。另一方面，静态数据的情况截然不同。应尽可能缓存只读数据。

---

**说明** 有一个名为Prevayler [Prevayler]的开源项目（以及其他许多类似的项目）支持有状态、共享和完全的领域模型实例化。这意味着领域模型和数据库在某种程度上是等同的。要做到这一点，需要大量RAM，如今的内存价格不断下降，64位机器也逐渐普及，因而这已经不成问题。

我的想法是，每次数据库发生更改时，就写入顺序日志文件。如果电源不中断，就可以取回领域模型实例化，方法是读取在特定时间点创建的镜像，并读取整个日志文件。这是一种非常简单有效的执行方法，因为所有数据都驻留在内存中。

这样我们就得到了对象数据库，但这是个简单的数据库，因为不存在故障处理或其他类似的需求。所有实例都始终驻留在内存中，无需与关系数据库之间进行O/R映射，甚至不必与对象数据库之间进行映射，而只需领域模型实例化。

---

在本书后续的内容中，将避免因应用服务器而带来的额外复杂性，并且假设处理的是富客户端，包括领域模型或Web应用程序，该富客户端直接与领域模型交互。

之所以这样决定，原因之一就是，我认为使用应用服务器的传统方法在某种程度上正在消亡，至少正在发生变化。是什么推动了这样的变化？或许是SOA，但那又是另外一个故事了（第10章将简要介绍）。

## 4.5 小结

我知道，读者都希望这一章永不结束，但遗憾的是，我们要告一段落了。我们已经介绍了很多基础知识，深入讨论了基于领域模型的“全新”默认架构，还通过一个示例简单介绍了它的应用方法。现在该到深入研究领域模型，并对本章简要介绍的初步草案进行大量更改的时候了。

新学网  
PDG

现在，一些读者可能会着急了，他们会说：“来吧，让我们看一下那些展示了如何实现领域模型及其环境的工具和技巧。”

抱歉，这并不是我们的前进方向。相反，我想在不思考工具和基础架构的情况下创建领域模型。这意味着将过去我一度嘲笑过的OO应用于实际应用程序。正如你所知，我从那时起就转变了思想。这在很大程度上是一个设计问题。

到目前为止，本书只是概括和抽象地介绍了模型的草图。现在该到改变的时候了。从现在开始，我们将讨论如何解决现实中的问题。

本章将讨论如何发展和精化前一章得到的初步领域模型。我们将应用TDD来缓慢地推进设计，在这个过程中推进模型。讨论将有所反复，而且进度会稍微慢一些（下一章将改变这种讨论方式），所以请读者做好准备。

首先讨论精化领域模型。

## 5.1 通过简单的 TDD 实验来精化领域模型

你是否记得第4章结尾的早期草图？它将在图5-1中再次出现。

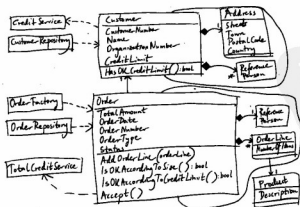


图5-1 来自第4章的早期草图

这张草图有点极端了，提供了过多的细节，通常不需要这样。不管怎样，现在最好是实际了解需求的一些细节。现在，让我们用TDD的方式来实现它，并看一下需要解决什么问题，在这个过程中，模型也将随之推进。

我通常反对逐个讨论特性，但既然已经在上一章中讨论了一些UI草图，因此我们将从尝试支持这些UI草图开始，然后再总结一下都发现了哪些特性。

### 5.1.1 从 Order 和 OrderFactory 的创建开始

我们从创建Order类和OrderFactory开始讨论。除了图5-2中所显示的部分之外，开始时暂不考虑其他方面。



图5-2 要实现的第一部分，Order和OrderFactory的一些部分

**说明** 前几个步骤是正确创建领域对象所必需的。我们必须首先完成这些步骤，然后才能进入有趣和关键的部分，也就是测试和实现模型所推动的逻辑。

我们将为它编写第一个测试。测试的目的是确保当要求工厂创建新订单时，不会返回null。代码如下：

```
[Test]
public void CanCreateOrder()
{
    Order o = OrderFactory.CreateOrder(new Customer());

    Assert.IsNotNull(o);
}
```

**说明** 你可能对这里的测试命名感到奇怪。命名xUnit测试的早期惯例是为它们加上前缀“Test”，但JUnit测试不必这样命名，因为它可以理解属性Test。这意味着我们可以对测试自由命名，也意味着从技术术语中脱离出来，并集中关注领域概念和设计的讨论。我喜欢将测试命名为它们要验证的目的的声明。这样，测试名称的集合就成为一个“什么应该成为可能”和“什么不应该成为可能”的列表。

首先，测试看起来非常简单。为了编译测试，需要创建空的Order类。现在一切情况良好。接下来，需要创建具有单一方法的OrderFactory类，但在编写该方法时，问题突然出现了。第4章中定义的规则之一是，Order必须有一个特定的Customer，这是通过将该责任分配给OrderFactory实现的，在创建时为Order提供一个Customer。因此，OrderFactory的Create-

`Order()`方法需要接收到作为参数的`Customer`实例（我们可以在代码中看到它）。这并不是高深的科学，我们只是刚开始在`Order`聚合上工作，并且将暂时忘记其他一切事情。

很容易想到的第一种解决方案（也可以在前面编写的测试中看到它）是也只创建空的`Customer`类。毕竟，我们将很快需要它，而且在这个测试中不打算与它进行交互，因此在构建块之间并没有增加太多的耦合（至少现在没有）。另一方面，感觉上我们在第一个小测试中涉及了太多的事情。这是一个典型的缺点，因此应尽量避免它。

第二种解决方案是创建`Customer`的模拟类，从而在聚合之间创建更松散的耦合，至少在本测试中是这样。我当然喜欢模拟的思想，但感觉在这里使用有些小题大做了。毕竟，我根本就未打算与`Customer`进行交互。我只是需要`Customer`作为占位符。

第三种解决方案是创建名为`ICustomer`的接口（或类似接口，或者令`Customer`成为接口），然后创建此接口的桩实现。感觉上这仍然有些小题大做，而且下一步的目的是什么？是为了能够交换`ICustomer`实现吗？这是否真正是我期望的？我想不是，因此为了避免错误，我决定不从这个解决方案开始。

第四种解决方案是跳过`Customer`参数。可以暂时这样做，目的是为了继续前进，然后在后面添加`Customer`参数时再更改测试。我还确定的一件事是，我不希望将`Order`和`Customer`组合到一起成为工厂责任。那么，使用TDD是否立即就能得到一个过早的设计决定呢？

在工厂中不将`Order`和`Customer`绑定到一起实际上将提高UI设计的灵活性，这样就不必在刚开始时就选择客户。相反，我将首先让用户填充有关新订单的其他信息，然后询问用户客户是谁。另一方面，在某个地方获得了灵活性通常也意味着在其他地方增加了复杂性。例如，假设价格是依赖于客户的，如果在未定义`Customer`的情况下将`OrderLine`添加到`Order`，那么应该使用哪个价格？然后又会发生什么事情？这当然并不是无法解决，但我们不应该引入复杂性，如果可以避免的话。

第四种解决方案的另一个缺点在于，从工厂中取回的`Order`实际上不是处于有效状态的。规则是每个`Order`都应该有`Customer`。当然，订单不是保存在工厂中的，但它仍然未处于有效状态。在执行工厂之后，使用者必须通过某些操作将订单转换为有效状态。

第五种解决方案是使用`Object`作为`CreateOrder()`的参数类型，但这种方法的目性不强，而且需要进行转换。具有同样问题的还有以下这种方法：创建所有领域模型类都必须实现的通用接口，例如`IEntity`，然后使用此接口作为参数。这相当于用一把大锤子来钉小螺丝钉。

你可能会想，我在这里违反了TDD原则，因为我思考了很多种解决方案，而没有通过测试、编码和重构来进行尝试。另一方面，在写下测试名称的时候，很多创意和思想（以及决定）会快速地在脑海中闪现，而且这是半意识的。（当然，这取决于对当前问题的经验。）

不管怎样，我们不要考虑得过多，相反，应该决定一种解决方案，并进行尝试。我将采取第一个解决方案。添加空的`Customer`类，它具有公共的默认构造方法，这种方法实际上很简单且快速，因此就让我们这样做。现在编译测试，得到红条，因为只编写了签名，而且只从`OrderFactory`中的`CreateOrder()`方法返回`null`。

现在该到实现`CreateOrder()`的时候了。第一次尝试可能像下面这样：

```
//OrderFactory
public static Order CreateOrder(Customer customer)
{
    return new Order();
}
```

为了编译上段代码，Order必须有一个可访问的默认构造方法。它可以是公共方法，但我决定让它成为内部方法，以便使Order的使用者只能通过OrderFactory对它进行实例化。当然，这并不是完美的保护，但至少错误的直接实例化将只在领域模型内部发生，这是正确方向上的一步。

是否到了重构的时候？工厂实际上添加了什么值？它未添加任何值，而只是略微增加了一些复杂性。实际上，至少应该在开始时不使用工厂，因为在未添加值的情况下，不应该使用它。现在，我仍然认为它需要处理很多操作，例如为customer创建快照，添加null对象，创建不同种类的订单，等等。

但是，到目前为止，使用了工厂的代码看起来很蹩脚，你是否也这样认为？它可以稍后添加值，但现在却比它原本应该的样子复杂得多。让我们直接改变它，使用可能是最简单的构造方式：直接使用构造方法。

**说明** 从工厂进行重构给人的感觉是奇怪的。但它只是我开始工作并且从一开始就采用了过于详细的设计的一个信号。

还有一点值得指出，第4章的草图中有一个CreateOrderLine()方法，但如果没有它，代码将更简单。

```
[Test]
public void CanCreateOrder()
{
    Order o = new Order(new Customer());

    Assert.IsNotNull(o);
}
```

是否还需要编写其他测试？当然，我们一直在讨论的是如何处理customer参数。因此，一个有趣的测试是检查新创建的order是否有customer。目前尚未验证这一点。让我们添加一个测试来验证它。

```
[Test]
public void CanCreateOrderWithCustomer()
{
    Order o = new Order(new Customer());

    Assert.IsNotNull(o.Customer);
}
```

这个测试告诉我们需要在Order上添加Customer字段或属性。（我认为不仅我的测试需要它，

而且一些需求也需要它。)

为了降低复杂性，我决定在Order类中添加只读属性，像下面这样。

```
//Order
public readonly Customer Customer;
```

如果没有办法更改Order的Customer，那么可以预期它总是存在的（并且永远不会改变），这将简化其他规则。如果不需要灵活性……

现在，每个测试都编译了，它们工作良好，而且确实得到了红条。通过修改构造方法很容易修复红条，因此它使用customer参数。构造方法现在像下面这样，我们得到了绿条。

```
//Order
public Order (Customer customer)
{
    Customer = customer;
}
```

### 5.1.2 一些领域逻辑

在图5-2中，我曾提到还应该为Order添加OrderDate。它的语义是什么？当第一次创建order时，它应该获得初始的OrderDate，而当order进入Ordered状态时，它应该获得最终的OrderDate（或类似的东西）。我们在测试中把这一点表达出来：

```
[Test]
public void OrderDateIsCurrentAfterCreation()
{
    DateTime theTimeBefore = DateTime.Now.AddMilliseconds(-1);

    Order o = new Order(new Customer());

    Assert.IsTrue(o.OrderDate > theTimeBefore);
    Assert.IsTrue(o.OrderDate
        < DateTime.Now.AddMilliseconds(1));
}
```

测试的思想是设置一个时间点的间隔，然后检查OrderDate是否在这个间隔之内。

像通常那样，这将无法编译。需要添加公共的OrderDate属性。这次，我准备使用私有字段加上公共的getter方法，因为我打算在生命周期的后面更改OrderDate值。这次，我让Order的构造方法来设置OrderDate字段，而不为构造方法添加另外一个参数。为了清楚起见，以下是代码片段：

```
//Order
private DateTime _orderDate;

public Order (Customer customer)
{
    Customer = customer;
    _orderDate = DateTime.Now;
}
```



```
public DateTime OrderDate
{
    get {return _orderDate;}
}
```

我们得到绿条。

本小节开始时提供了一个图。它描述了图5-2中模型的一个子集。模型略有推进，如图5-3所示，图中显示了代码。

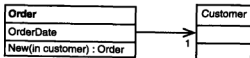


图5-3 要实现的第一部分，即Order（修改后）

**说明** 你可能奇怪为什么我有时采用手绘图形，而有时却又用UML工具来绘图。原因是，最初我的目的是说明一件事情，只快速手绘草图就可作为思考和交流过程的帮助。当已经实现了这件事情时，那么就用UML工具对实现进行可视化。

比较图5-3和图5-2，区别并不是很大。实际上，区别就是必须实现Customer类的第一个版本以及从Order到Customer的关系。这实际上没有真正更改模型，而只是子集图。

**说明** 图5-3中还显示了构造方法，目的是提供类如何工作的更详细的视图。

在进行下一步之前，现在该到重构的时候了。首先，我希望把对Order的构造方法的调用移出来，移至[SetUp]，但这样就必须对最后一个测试的时间间隔稍加修改，这就使得测试有些偏离重点了。此外，到目前为止所显示的三个测试只是有关创建的，因此我希望在测试方法本身中进行调用。我们暂时不进行重构，继续精化领域模型。

### 5.1.3 第二个任务：OrderRepository + OrderNumber

在前一小节中，我们讨论过为了让OrderFactory满足需要，来从头创建新的Order实例，它将是什么形式的，但在该节中我们决定暂时跳过工厂。我认为，下一个问题很自然就是，OrderRepository应该如何工作。

第4章中讨论了OrderRepository的GetOrders()。现在，我认为最重要，也是最典型的方法就是GetOrder()，因此我们将从此方法以及Order实体的标识符开始。

我们所讨论的OrderNumber到底是什么？通常，订单具有某种形式的唯一标识，人们可利用此标识来识别特殊的订单。一个常见的解决方案是，从递增的数字序列中为每个订单分配一个新的数字。图5-4中再次显示了修改之后的Order类，以及OrderRepository类中的新增方法。

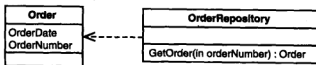


图5-4 Order和OrderRepository。

我知道，我应该讨论OrderRepository，但首先执行一些与OrderNumber有关的操作是非常重要的。当创建Order时，是否需要为它提供标识？我认为以下说法是正确的：OrderNumber为0，直到订单第一次被保存（即状态为Ordered）。在这之前，我们不想在那个仅仅可能是真实订单上浪费OrderNumber。

**说明** 这在很大程度上取决于需求（无论它是好是坏），但我越来越觉得我们正在把两个不同的事情，即业务标识和持久化标识混合到一起。下一章中将回头讨论这个问题。

还有一点是我非常不喜欢的，即持久化与业务规则之间的耦合。也就是说，当Order被保存时，它也处于“Ordered”状态。我们暂时放下此问题，留到第7章讨论业务规则时再作讨论。

因此，像通常那样，可以编写一个非常简单的测试，像下面这样：

```
[Test]
public void OrderNumberIsZeroAfterCreation()
{
    Order o = new Order(new Customer());

    Assert.AreEqual(0, o.OrderNumber);
}
```

猜一下将发生什么情况？此测试不能编译，因此为Order类添加新的只读属性，并让它使用私有的\_orderNumber字段。

这样，就可以从构造方法的角度来处理OrderNumber了。但是，如果OrderNumber属性是只读的，那么，当使用OrderRepository来查找和重建旧的Order时，如何为它赋值呢？为了使整个问题更清晰，让我们后退一步，并编写一个测试和[Setup]方法（在用名称\_repository声明OrderRepository之后）：

```
[SetUp]
public void Setup()
{
    _repository = new OrderRepository();
}

[Test]
public void OrderNumberCantBeZeroAfterReconstitution()
{
    int theOrderNumber = 42;
```

```
_FakeAnOrder(theOrderNumber);

Order o =
_repository.GetOrder(theOrderNumber));

Assert.AreEqual(theOrderNumber, o.OrderNumber);
}
```

像通常那样，整个测试将无法编译。没有OrderRepository类，因此编写此类，它只带有GetOrder()方法的签名，而没有代码。测试代码仍然无法编译，因此必须在测试类中建立\_FakeAnOrder()的桩实现。我得到一个红条。

现在离想要解决的问题越来越远了。为了能够编写\_FakeAnOrder()方法，需要让存储库知道Order。可以继续前进，并在OrderRepository中实现新的方法，OrderRepository的用途只为了支持其他测试，但这是一个很好的信号，说明还需要编写其他测试。需要对保存Order进行测试，或者至少需要让存储库知道Order。

---

**说明** 我必须承认在如何保存方面，我有很多想法，但这里将缓慢地前进，并编写简单的代码（我将对这些代码进行大量重构）。请不要对此感到失望，它并不是最终的解决方案。

---

因此，为OrderNumberCantBeZeroAfterReconstitution()测试添加Ignore属性，这样它现在不会给出红条。然后编写如下测试：

```
[Test]
public void CanAddOrder()
{
    _repository.AddOrder(new Order(new Customer()));
}
```

像通常那样，它无法编译。需要为OrderRepository添加AddOrder()方法。而且照常只添加签名，但这并不足以得到红条。事实上，CanAddOrder()方法中根本没有测试代码。我并不喜欢这种思想：仅出于外部测试类能够检查存储库内部状态这一个目的就让存储库发布任何方法。当然，可以使用GetOrder()方法，但这样就陷入了“鸡生蛋还是蛋生鸡”的问题中。此方法的实现远远无法作为开始。

相反，我们后退一步，并为存储库添加私有的IList，用于保存订单。我们并不向外部发布任何有关IList的信息，它在很大程度上是实现细节，而且我认为很快就会除去它。相反，使用另一个断言，它不是xUnit断言，而是来自Diagnostics名称空间的断言，用于检查那些我认为应该检查的内容。Assert()所做的事情是检查声明是否为真。如果不为真，开发人员将收到通知。AddOrder()方法如下：

```
//OrderRepository
public void AddOrder(Order order)
{
    int theNumberOfOrdersBefore = _theOrders.Count;
```

```
//TODO Add here...
```

```
Trace.Assert(theNumberOfOrdersBefore
    == _theOrders.Count - 1);
```

### 是否需要另一个断言库

对于普通的Diagnostics断言的使用，应该三思而后行。一个问题是它可能无法与NUnit进行很好的集成。另一个问题是无法根据不同的情况（例如开发、持续集成、beta测试和生产期间）自定义它的行为。我在前一本书中曾讨论过这方面的更多内容[Nilsson NED]。

你可能还奇怪，为什么我没有确保可以用NUnit测试中的某种方式来表达测试。我可以公开测试所必需的状态，但我选择不必要的时候避免这样做。如果采用这种做法，那么将更难重构存储库类，而且我确信我是需要重构的。

前面所使用的技术在某种程度上受到了Bertrand Meyer所著的*Design By Contract* [Meyer OOSC]一书的启发。尽管它完全不是形式化的，但断言表示了AddOrder()保证了什么，即它对于后置条件的保证。

那么AddOrder()又需要什么前置条件呢？是否要求之前order对于存储库是未知的？不，我不喜欢将这看作错误。是否要求\_theOrders不为null？但如果\_theOrders为null，方法将抛出异常。此外，从整体看，\_theOrders是非常短暂的解决方案，这一点我们很快就会看到。现在暂时离开这个话题。

因此，我们创建MyTrace.Assert()，如果它接收到false参数，那么只会抛出异常。这样，它至少就能够与NUnit/Testdriven.net进行很好的集成。

编译，测试，然后得到红条。情况正常，因此，我们用TODO注释来交换add调用：

```
//OrderRepository
public void AddOrder(Order order)
{
    int theNumberOfOrdersBefore +1 = _theOrders.Count;

    _theOrders.Add(order);

    MyTrace.Assert(theNumberOfOrdersBefore
        == _theOrders.Count);
}
```

现在得到了绿条。这里编写的代码有些奇怪，但我们前进了，而且创建的测试并不奇怪，因此让我们继续。

**说明** 我还使用了模拟(mock)来验证被测系统(SUT)按预期工作，但你根据第3章的内容就可以想象出来它是如何工作的，因此我们继续前进。

现在该到返回OrderNumberCantBeZeroAfterReconstitution()的时候了，上次遇到问题

的地方是用于测试的帮助方法，名为 `_FakeAnOrder()`。此方法如下：

```
//A test class
public void _FakeAnOrder(int orderNumber)
{
    Order o = new Order(new Customer());

    _repository.AddOrder(o);
}
```

现在又有了另一个问题。你是否看到了？是的，问题就是如何将伪 `OrderNumber` 写入订单实例。`OrderNumber` 属性是只读的（这是有意义的），因此使用它是无法写入的。

事实上，这是一个通用问题。它可以这样表示：如何从外部（例如在存储库中）设置实例中的值（这些实例正在从持久化状态进行重建）？

### 5.1.4 重建持久化的实体：如何从外部设置值

前面曾提到设置实例中的值的通用问题（这些实例正在进行重建，重建的方式是从数据库读回它）。对于 `OrderNumber` 来说，这显然是个问题，因为 `OrderNumber` 将永远无法通过让使用者与属性进行直接交互的方式来更改，但它或多或少与其他属性是相同的。我们暂时假设 `OrderDate` 是读/写。如果使用者设置了新的 `OrderDate`，那么可能需要进行一些测试。当正从数据库读取 `Order`，并作为实例来重建它，而此时 `OrderDate` 正在获取一个值，那么这时执行这些检查就是没有意义的。

有几种可能的方式可解决此问题。让我们看一下将遇到什么情况。

#### 1. 使用特定的构造方法

我们可能有一个只能用于此重建的特定构造方法。它可能像下面这样：

```
//Order
public Order(int orderNumber, DateTime orderDate,
    Customer customer)
```

它可以工作，但 `Order` 的使用者并不十分清楚他们不应该使用该构造方法。当然，可以让它成为内部方法，从而极大地缓解这个问题。

如果使用静态命名的方法作为工厂，那么构造方法的目的可能就变得稍微清楚了，像下面这样：

```
//Order
public static Order ReconstituteFromPersistence(int orderNumber
    DateTime orderDate, Customer customer)
```

这不仅使得方法的目的更清楚，而且指出了这样一个事实：使用者不应该混用此方法。然而，使用者可能仍然会错误地使用了此方法，但仍可以通过使用内部方法来解决。当类更接近于真实情况时，例如当有50个属性需要在重构时设置，那么这也是有问题的。如果遇到这种情况，参数列表在很早以前就变得不易处理了。

#### 2. 使用特定方法，此方法通常在特定接口中

另一个选项是确定 `Order` 类必须实现的方法，通常此方法应该只能通过特定接口使用。这样

就大大降低了使用者错误地混用方法的风险。如果使用者想有意混用方法，那么也能够做到，但在这某种程度上总是个问题。如果允许从外部设置实例的值，那么使用者也可以这样做。这实际上不一定是件坏事。相反，我们应采取一种被动方法，并决定可能的用法不会导致问题。

那么这个特定方法可能是什么形式的？一种情况可能是在Order上具有如下方法：

```
//Order
public void SetFieldWhenReconstitutingFromPersistence
    (int fieldKey, object value)
```

这肯定有些混乱。现在必须设置字段键的映射，而且必须从这时开始维护此映射，既作为映射来维护，也作为SetFieldWhenReconstitutingFromPersistence()代码本身来维护。

一个类似的解决方案（仍不具有很好的可维护性）是用fieldKey参数来交换fieldName参数。

### 3. 对私有字段使用反射

可以使用反射，但像通常那样，这也是利弊共存的。缺点是反射的速度慢（至少与普通访问相比，但它是否过于缓慢？），我们不能剥夺用户的权限，而且必须从外部了解内部结构（例如字段名称）。（最后一件事情可以通过添加仅用于反射的私有属性得到部分解决，或者通过设置命名规则来部分解决，这样，如果有名为Name的属性，那么就应该有名为\_name的私有字段。这种方法可能稍微健壮一些，但问题仍然存在。）

最重要的优点是它不是插入式的（intrusive）。我们可以按自己所需的方式来创建自己的领域模型，而不必添加特定于基础架构的、模糊的结构。

---

**说明** 还有一些小的细节，例如Customer不能是只读的。后面的几章中将深入讨论这一点。

---

### 4. 完全不同的解决方案

我们可以采用一种完全不同的解决方案：例如，始终保持值在实例的外部，位于“安全的地方”，这样实例只是像代理一样[GoF Design Patterns]。它实际上是一种完全不同的解决方案，当然也有自己的优缺点。然而，我现在真地并不想做出这样一个大的、专门的决定。它将使得领域模型类中混合那些与领域毫无关系的内容。我们尽可能保持简单前进。

所有这些都具有很大的基础架构特色。我们将把如何从外部设置值的决定推迟到后面，也就是当开始思考采用什么基础架构来支持领域模型时，再做出这个决定。另一方面，现在需要做出某种决定，以便继续进行测试。就目前来讲，最简单的机制可能是什么？遗憾的是，实际上现在没有简单的解决方案。这是一个好的信号，说明应该回头重新思考整个事情。

---

**说明** 这么长的一段讨论只因为开始处理OrderNumber的语义就结束了，这看起来有些不合理。但既然进行到这里了，那么就让我们结束讨论吧。

---

现在，我决定对私有字段使用反射。在OrderRepository中编写帮助方法（或许后面应该将它提取出来），代码如下：

```
//OrderRepository
public static void SetFieldWhenReconstitutingFromPersistence
(object instance, string fieldName, object newValue)
{
    Type t = instance.GetType();
    System.Reflection.FieldInfo f = t.GetField(fieldName
        , BindingFlags.Instance | BindingFlags.Public
        | BindingFlags.NonPublic);
    f.SetValue(instance, newValue);
}
```

至少，我们可以继续前进了。前面这一大段讨论仅仅是因为OrderNumber是只读的。但要记住，这是一段通用讨论，中心内容就是：当重建持久化的实例时，如何让存储库设置值才能不遇到只读属性问题和setter方法中代码问题。

你是否记得OrderNumberCantBeZeroAfterReconstitution()测试？下面重复这个测试：

```
[Test]
public void OrderNumberCantBeZeroAfterReconstitution()
{
    int theOrderNumber = 42;
    _FakeAnOrder(theOrderNumber);

    Order o =
        _repository.GetOrder(theOrderNumber);

    Assert.IsTrue(o.OrderNumber != 0);
}
```

前面曾添加了Ignore属性，这样测试就无法执行了。现在，移除Ignore属性，并得到红条。现在该到更改\_FakeAnOrder()的时候了。要确保新订单的OrderNumber为42，像下面这样：

```
//A test class
public void _FakeAnOrder(int orderNumber)
{
    Order o = new Order(new Customer());

    OrderRepository.SetFieldWhenReconstitutingFromPersistence
        (o, "_orderNumber", orderNumber);

    _repository.AddOrder(o);
}
```

仍然得到红条，但这次是由于某种其他原因。我担心造成这种情况：有多种原因导致红条，这不是推荐的做法。我在TDD方法中跳跃了太大的一步。

不管怎样，问题在于尚未实现OrderRepository.GetOrder()。它还只是空方法（返回null）。现在，在IList中有订单，因此只能在列表中对各项进行迭代，逐个检查它们。代码如下：

```
//OrderRepository
public Order GetOrder(int orderNumber)
{

```

```

foreach (Order o in _theOrders)
{
    if (o.OrderNumber == orderNumber)
        return o;
}
return null;
}

```

我知道，这是不成熟的实现，但它正是现在需要的。猜一下会得到什么结果？绿条。

这样又回到正确轨道上，但完成了一件大事，即重构了SetFieldWhenReconstitutingFromPersistence()方法。SetFieldWhenReconstitutingFromPersistence()方法不属于OrderRepository。现在，创建RepositoryHelper类，并把SetFieldWhenReconstitutingFromPersistence()方法移到该类中。在更改对SetFieldWhenReconstitutingFromPersistence()的调用后，仍然得到绿条。

看一下领域模型图现在的情况。修订后的图如图5-5所示。

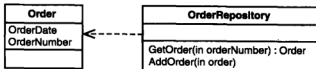


图5-5 修订之后的Order和OrderRepository

我越来越感觉到整个事情并不像我想象的那么好，但我不知道原因。我确信当继续前进时，问题会更清楚，因此现在就继续前进。列表中的下一项是什么？或许是获取特定客户的订单列表。

### 5.1.5 获取订单列表

我们正在讨论的是OrderRepository中的另外一个方法，它的名称是GetOrders()，它在开始时带有客户作为参数。当找到一些Order时，问题的第二部分就是显示像OrderNumber和OrderDate这样的一些值。图5-6显示了将在本节中创建的一些内容。

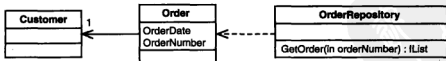


图5-6 用于列出订单的Order、Customer和OrderRepository

**说明** 你是否注意到图5-6中的图在两种不同表达方式中的多重性区别（一种表达方式是UML图，另一种表达方式是数据库图）？如果你来自数据库背景，那么诸如从Customer到Order的导航关系这样的区别可能会令你惊讶，多重性就是一对一关系，因为它是从Order到Customer。



现在该到进行另一个测试的时候了。仍然需要在OrderRepository中创建Order的伪类，以便运行测试。代码如下：

```
[Test]
public void CanFindOrdersViaCustomer()
{
    Customer c = new Customer();

    _FakeAnOrder(42, c, _repository);
    _FakeAnOrder(12, new Customer(), _repository);
    _FakeAnOrder(3, c, _repository);
    _FakeAnOrder(21, c, _repository);
    _FakeAnOrder(1, new Customer(), _repository);

    Assert.AreEqual(3, _repository.GetOrders(c).Count);
}
```

如你所见，我更改了\_FakeAnOrder()，因此它现在还带有Customer和OrderRepository作为参数。当然，这是简单的更改。另一件相当简单的事情是，需要使解决方案成为可编译的，并添加GetOrders()方法。代码如下（当然，第一步还是必须得到红条）：

```
//OrderRepository
public IList GetOrders(Customer customer)
{
    IList theResult = new ArrayList();

    foreach (Order o in _theOrders)
    {
        if (o.Customer.Equals(customer))
            theResult.Add(o);
    }

    return theResult;
}
```

我并不确定是否喜欢以上代码。我正在考虑检查等同性。假设有两个Customer实例（不是指向同一实例的两个变量，而是两个独立的实例），它们都具有相同的CustomerNumber。那么，调用Equals()将返回true还是false？这取决于Equals()是否已被重写，以及被重写的实现是否使用CustomerNumber来决定等同性。

**说明** 熟悉标识映射模式[Fowler PoEAA]概念并把它看作一个必要手段的读者可能会奇怪，为什么我们还要讨论这些。标识映射的思想是这样的：对于单一客户，将不会有单独的实例。标识映射帮助我们确保这一点。

语言本身（例如C#）却不会确保这一点。当我们在第8章和第9章中讨论持久化解决方案时将涉及这一点。在这之前，先保留这个问题。

### 5.1.6 该到讨论实体的时候了

由于Order和Customer是实体的例子，因此本章中有几个地方已经涉及了实体这个主题，但

让我们后退一步，集中关注一下这个概念。

对于我们来说有一件很重要的事情是，随着时间的推进对领域中的一些内容进行跟踪。无论客户更改了名称还是地址，客户仍然是同一位客户，而且跟踪这件事情也是有意义的。另一方面，如果某位客户的介绍人改变了，那么它可能就不是要跟踪的事情了。客户通常就是实体[Evans DDD]的典型示例。再次强调，实体是一种通过标识（而不是其值）跟踪的对象。

举一个极端的例子来说明什么不是实体，如整数42。我根本并不关心42的标识，而只是关心值。当值更改时，我并不认为更改的是42的值，而更改为了一个全新的值，与原值没有任何关系。42是个值对象[Evans DDD]，而不是实体。

如果将这个极端示例用于我们的领域，那么可以这样说：我们对通过标识来跟踪Customer的ReferencePerson并不感兴趣。我们感兴趣的是通过值对它进行跟踪。本章后面还会讨论与值对象有关的内容。

---

**说明** 当然，什么应该按照标识进行跟踪，什么应该仅被看作值，这在很大程度上依赖于领域。举一个例子。如前所述，这个应用程序中的ReferencePerson可以作为值对象来处理，但如果应用程序被用作推销员的销售支持应用程序，那么这些推销员可能就会把ReferencePerson看作一个实体。

---

### 5.1.7 再次回到流程上来

遗憾的是，在重写Equals后，CanFindOrdersViaCustomer()测试将无法成功执行。因为，测试中使用的所有客户都具有相同的CustomerNumber，它为0，因此我发现了5个（而不是3个）Order具有正确的Customer。稍微更改一下测试。创建另一个小的帮助方法，名为\_FakeACustomer()，像下面这样：

```
//A test class
private Customer _FakeACustomer(int customerNumber)
{
    Customer c = new Customer();

    RepositoryHelper.SetFieldWhenReconstitutingFromPersistence
        (c, "_customerNumber", customerNumber);

    return c;
}
```

---

**说明** \_FakeACustomer()应该将客户与CustomerRepository关联起来（或者是当操作单元执行的时候，与操作单元关联起来）。我们暂时跳过这个问题，因为现在它不影响流程。

---

然后修改CanFindOrdersViaCustomer()测试，因此使用\_FakeACustomer()，而不仅仅是直接调用Customer的构造方法。修改之后的代码如下：

```
[Test]
public void CanFindOrdersViaCustomer()
{
    Customer c = _FakeACustomer(7);

    _FakeAnOrder(42, c, _repository);
    _FakeAnOrder(12, _FakeACustomer(1), _repository);
    _FakeAnOrder(3, c, _repository);
    _FakeAnOrder(21, c, _repository);
    _FakeAnOrder(1, _FakeACustomer(2), _repository);

    Assert.AreEqual(3, _repository.GetOrders(c).Count);
}
```

再次得到绿条。

现在该总结我们的图进行到什么地方了。在图5-6之后，我们还为Customer类添加了CustomerNumber，如图5-7所示。

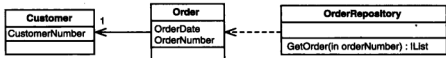


图5-7 用于列出订单的Order、Customer和OrderRepository（修订后）

### 5.1.8 总览图

到目前为止，我们正在尝试完成的事情是什么？对于像我一样在数据库和以数据库为中心的应用程序上工作十余年的人来说，思考一下就会发现代码很奇怪。例如，根本没有到数据库的跳转。当然，当关注领域模型和编写测试时，这一点是好的方面，但我希望稍后也能在不涉及数据库的情况下编写测试。

到目前为止，测试为我们带来了哪些结果？对于刚起步的人来说，我们已经将测试用作一种发现和跟踪方式，用于那些我们希望领域模型实现的行为和规范。通过以伪实现/初步实现作为开始，我们能够将主要精力集中在创建API上。我们还尽力尝试提出和使用通用语言。最后，测试提供了一个坚实的基础，借助于这个基础，可以将代码从初步实现转变为“实物”。

我们还考虑了设计选择的最终目标。还以存储库形式创建了抽象层，从而可以推迟对数据库的处理（这样做是值得的）。

因此，我们现在已经完成的工作是编写了OrderRepository的第一个伪版本！我们还为OrderRepository赋予了几种不同职责。这可能是导致设计不够简单的原因。我并不要现在就关注基础架构，而是想将它推迟到后面进行。然而，由于我注意到我目前正在领域模型中的基础架构上工作，因此暂时使用特性列表，并考虑通过一些重构来得到OrderRepository伪实现的更好解决方案。

说明 我正在尝试使用具有特定明确意义的“桩”、“伪”和“模拟”。以下是来自Astels[Astels TDD]

的一段引用。

桩：具有方法的类，但方法不执行任何操作。它们的作用只是为了使系统能够编译和运行。

伪：具有方法的类，方法返回一个固定值，或返回一些可以被硬编码或用编程方式设置的值。

模拟：一个类，在此类中可以设置一些期望值，例如调用什么方法，带有哪些参数，调用频率如何，等等。也可以为不同的调用情况设置返回值。模拟还提供了一种用于验证期望值是否被满足的方式。

但这并不是唯一的定义。Meszaros[Meszaros XUnit]的讨论就略有不同。

测试桩是供测试使用的对象，测试用它来替代SUT所依赖的真实组件，以便测试可以控制SUT的间接输入。这允许测试强制SUT执行那些在实际组件上无法执行的操作。

伪对象是用于在测试中替代真实依赖性组件功能的对象，但目的并不是验证间接输入和输出。通常，它将实现真实依赖性组件的相同功能或功能子集，但实现方式简单得多。

模拟对象是供测试使用的对象，测试用它来替代SUT所依赖的真实组件，以便测试能够观察其间接输出。通常，模拟对象通过两种方式来模拟实现，一是返回硬编码的结果，二是返回测试预加载的结果。

### 5.1.9 建立 OrderRepository 的伪实现

我的第一个思路是应该对 OrderRepository 的一些内部部分建立伪实现，以便可以在不同情况中使用相同的 OrderRepository，并在构造方法中得到一个像 DataFetcher 这样的实例，此实例将执行适当的操作。当我更多地思考它时，又感觉过于小题大做了。如果决定让 OrderRepository 与关系数据库对话（而不是作为伪实现），那么在某种程度上整个 OrderRepository 必须被交换。整个类也将必须被交换。

或许图5-8中所示的接口正是现在所需的。

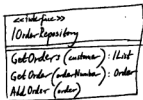


图5-8 IOrderRepository，第一个提议

从直觉上讲，我很喜欢此接口。可能是由于我过去的COM经历所致，或者是因为很多作者（例如 [Szyperski Component Software]、[Johnson J2EE Development without EJB] 和 [Löwy Programming .NET Components]）都明确表示，对接口编程比对具体类编程更好。

正如我所提到的，我在接口的签名中确实有具体类（Customer和Order）。这次仍然暂时放下这一点不管，因为目前在为这些类创建接口的过程中未看到任何值。

另一件事情是创建一个更通用的存储库接口可能会更好，如图5-9所示。

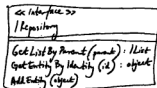


图5-9 IRepository（或者更确切地说是IRepository），第二个提议

这可能引起我的兴趣，但现在我更倾向于选择图5-8中的更特定的接口，因为它清楚地表明了什么是需要的。在使用通用的IRepository时，需要进行更多转换，并且可能错误地尝试以同一个相同形式收集所有即将出现的存储库。

另一个解决方案是使用IRepository接口中的泛型再加上基类，这样至少可以实现类型安全性（并避免代码重复），而且避免额外的类型转换。一个问题是可以为存储库添加Delete()方法，尽管并不是所有存储库都需要它。而且即使对于泛型来说，GetListByParent (parent)也是有麻烦的，因为当某个实体有多个父时该如何处理？当然，这只是早期的草图，而且此问题可以得到解决。但我强烈认为更好的方法是现在让代码尽可能清楚，并跳过这样的泛化。这让人感觉不重要。通常，没有一成不变的决定。后面当能够更容易看到哪些部分可以被泛化时，可能需要改变这些决定，但现在，将使用IOrderRepository。

由于此原因，将OrderRepository类重命名为OrderRepositoryFake，并且让类实现IOrderRepository。还需要更改测试类中的[SetUp]代码。目前的代码如下：

```
//A test class, the [SetUp]
```

```
_repository = new OrderRepository();
```

现在它可能是这样：

```
//A test class, the [SetUp]
```

```
_repository = new OrderRepositoryFake();
```

\_repository的声明必须使用IOrderRepository作为类型。

测试仍然可以工作。

我猜想纯粹的XPers并不是特别需要刚才的重构。实际上现在并不是必须进行该更改，它是为未来（当需要用实际存储库代替伪版本时）所做的准备。我同意这一点，但不能不为未来连接基础架构做一些考虑（参见后面的章节）。

我希望这里的准备工作被证明是好的。我又陷入了过早猜想的旧习惯中……另一方面，我们当然可以自由选择自己的平衡，而且不必遵守XP教条！

### 5.1.10 简单讨论一下保存

前面曾提到过保存数据。AddOrder() 实际上并不执行保存，它只是将Order添加到存储库。那么它应该意味着保存吗？不，我认为并不需要这种行为。我希望从AddOrder()得到的结果是存储库（以及底层基础架构）从该时刻起应该知道实例，并且在调用PersistAll()时处理它。

我们需要让基础架构来处理保存功能。现在的问题在于是对存储库实例调用PersistAll()，还是对更高层次上的某个对象调用它（该对象监视所有存储库的实例）。我认为我再次离开了领域模型，而正在进入基础架构部分。我很高兴这个问题被提出来了，但有些内容的讨论不得不等到下一章再进行。在这方面，有很多问题需要讨论。现在，我很满意将AddOrder()作为让存储库知道实例的方式。就目前的测试来讲，以下事实实际上并不重要：实例是暂时的（但准备好了被保存），而且在该时刻不保存。

我像读者一样急于编写更多的测试。列表上有什么？我已在特性2中解决了此问题的大部分。尚未解决的问题是显示每个订单的总量、订单类型和介绍人。

让我们逐个地来解决它们，从每个订单的总量开始。

### 5.1.11 每个订单的总量

乍看起来，这像是个琐碎的任务。需要创建OrderLine类，并将它作为列表嵌入到Order类中。然后，就可以只在OrderLine集合上进行迭代，并计算总量。当然，实际上可以从使用者代码进行计算，但我并不想这样做。这将会把算法暴露给使用者，而且也不够清晰，目的性也不明确。在实际中，当我们进一步前进添加很多不同种类的折扣时，算法将变得更复杂。相反，我们需要在Order类中有一个名为TotalAmount的只读属性，它可以在OrderLine集合上进行内部迭代，并计算出值。

但这对于我来说过于超前了，我成了“过早的优化者”（尽管我尽力避免）。仅仅为了能够显示订单的TotalAmount而实例化customer的每个order的所有OrderLine，是一种不明智的做法，这为我们敲响了警钟。根据执行环境和所选择的基础架构的不同，这可能并不是问题。但另一方面，在分布式系统中，它就是个问题，而且可能成为一个损害数据库服务器、网络和垃圾收集器的大问题。

我必须承认，我不得不强迫自己现在不处理此优化。直接修复它很简单，但现在，正在确定领域模型应该是什么形式的时候，它实际上并不重要。现在，更重要的事情是设计的简单性和清晰性，过后可以通过性能分析程序（profiler）来查看它是否成为真正的性能问题。因此，我从最简单的解决方案开始，然后稍后重构TotalAmount属性（如果性能不是足够好的话）。

完成了！现在我跳过了一个优化。这种感觉真是太好了。

现在，既然我们想到了TotalAmount，那么就后退一步。现在该到编写一个测试的时候了（迟做比不做好）。我从能想到的最简单的测试开始。

```
[Test]
public void EmptyOrderHasZeroForTotalAmount()
{
```

```
Order o = new Order(new Customer());

Assert.AreEqual(0, o.TotalAmount);
}
```

很容易使此测试成为可编译的。只需为Order类添加一个公共的只读属性，名为TotalAmount，并让它返回-1。编译测试并得到红条。更改它，让属性返回0，这样就得到绿条。

现在可以离开该测试了，但我现在拥有了上下文，因此需要进行一点工作。这里是另一个测试：

```
[Test]
public void OrderWithLinesHasTotalAmount()
{
    Order o = new Order(new Customer());

    OrderLine ol = new OrderLine(new Product("Chair", 52.00));
    ol.NoOfUnits = 2;

    o.AddOrderLine(ol);

    Assert.AreEqual(104.00, o.TotalAmount);
}
```

在这个简单测试中，完成了很多设计，现在需要创建两个类，并在原有类上创建新方法，目的只是能够编译测试。我认为现在最好注释掉该测试，并从一个稍微简单的测试开始，主要关注OrderLine类。

首先，我希望OrderLine默认地从所选定的产品获取价格。我们来编写一个测试。

```
[Test]
public void OrderLineGetsDefaultPrice()
{
    Product p = new Product("Chair", 52.00);

    OrderLine ol = new OrderLine(p);

    Assert.AreEqual(52.00, ol.Price);
}
```

虽然这是一个比较大的跳跃，需要编写一定量的代码，但我现在很有信心，因此让我们来编写Product类。构造方法中的第二个参数应该是单位价格。Product类还应该有两个只读属性：Description和UnitPrice。

我还编写了OrderLine类，它有两个成员：Product和Price。代码如下：

```
public class OrderLine
{
    public decimal Price = 0;
    private Product _product;

    public OrderLine(Product product)
    {
```

```

    _product = product;
}

public Product Product
{
    get {return _product;}
}
}

```

现在测试可以编译了，但得到红条。我需要更改构造方法，以便从Product提取出价格，并放入OrderLine本身中。现在，构造方法如下：

```

//OrderLine
public OrderLine(Product product)
{
    _product = product;
    Price = product.UnitPrice;
}

```

再次得到绿条。我编写了一个测试，用来证明可以重写OrderLine中的默认价格，但我们对它的兴趣并不大，因此先将它放在这里。

要编写的下一个测试是orderLine的TotalAmount的计算。测试如下：

```

[Test]
public void OrderLineHasTotalAmount()
{
    OrderLine ol = new OrderLine(new Product("Chair", 52.00));
    ol.NumberOfUnits = 2;

    Assert.AreEqual(104.00, ol.TotalAmount);
}

```

这里要做出两个额外的设计决策。我们需要在OrderLine上有NumberOfUnits字段和TotalAmount。令NumberOfUnits为公共字段，并将TotalAmount设为只读属性（返回0）。编译测试，但得到红条。然后将TotalAmount更改为下面这样：

```

//OrderLine
public decimal TotalAmount
{
    get {return Price * NumberOfUnits;}
}

```

再次得到绿条。

看一下刚刚所做的修改。在图5-10中，可以看到一个包含OrderLine和Product的图。

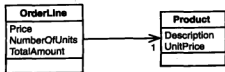


图5-10 OrderLine和Product



现在该回到前面被注释掉的那个测试了。下面列出了这个测试，但比前面的测试稍微“大了一些”，它处理两个OrderLine。

```
[Test]
public void OrderWithLinesHasTotalAmount()
{
    Order o = new Order(new Customer());

    OrderLine ol = new OrderLine(new Product("Chair", 52.00));
    ol.NoOfUnits = 2;
    o.AddOrderLine(ol);

    OrderLine ol2 = new OrderLine(new Product("Desk", 115.00));
    ol2.NoOfUnits = 3;
    o.AddOrderLine(ol2);

    Assert.AreEqual(104.00 + 345.00, o.TotalAmount);
}
```

为了使此测试能够编译，需要为Order添加AddOrderLine()和TotalAmount。暂时令只读属性TotalAmount只返回0，并且在OrderWithLinesHasTotalAmount()测试上设置[Ignore]。反过来，我编写另外一个测试为集中关注AddOrderLine()。此测试如下：

```
[Test]
public void CanAddOrderLine()
{
    Order o = new Order(new Customer());

    OrderLine ol = new OrderLine(new Product("Chair", 52.00));
    o.AddOrderLine(ol);

    Assert.AreEqual(1, o.OrderLines.Count);
}
```

为了编译此测试，需要为Order添加AddOrderLine()和OrderLine。如果这二者当中有一个不工作，那么测试将失败，因此按我需要的方式来编写OrderLine，但不结束AddOrderLine()。代码如下：

```
//Order
private IList _orderLines = new ArrayList();

public IList OrderLines
{
    get {return ArrayList.ReadOnly(_orderLines);}
}

public void AddOrderLine(OrderLine orderLine)
{
}
```

编译测试，得到红条。像下面这样为AddOrderLine()方法添加一行代码：

```
//Order
public void AddOrderLine(OrderLine orderLine)
{
    _orderLines.Add(orderLine);
}
```

编译测试，得到绿条。

现在该从OrderWithLinesHasTotalAmount()测试上删除Ignore属性了。测试得到红条，因为Order的TotalAmount只返回0。

像下面这样重写TotalAmount属性：

```
//Order
public decimal TotalAmount
{
    get
    {
        decimal theSum = 0;
        foreach (OrderLine ol in _orderLines)
            theSum += ol.TotalAmount;

        return theSum;
    }
}
```

简单且直接，我们得到绿条。你是否注意到我克制住了对优化的强烈愿望？我只是用最简单的方式编写了它，而且并未发现这样编写有什么问题。毕竟，我现在将Order及其OrderLine看作聚合，也因此将它们看作是默认的加载单元。

### 5.1.12 历史客户信息

我觉得现在应该进行一些重构了。有没有不良的代码？我不喜欢的一件事情就是，Order与Customer之间有一个关系。当然，Order有一个Customer，这是没问题的，但如果我们在一年之后再来看Order，那么可能希望看到的是Order被创建时的Customer信息，而不是现在的Customer信息。

另一个问题是构造方法带有Customer参数。这意味着Order可能直接就进入了持久对象图，而不调用IOrderRepository.AddOrder()，这可能不是好的想法。（当然，这也取决于基础架构，但并不是显然有边界。）AddOrder()所表明的是：当一个订单到了保存的时候就应该保存它。

这些问题都不难解决，但如何才能清晰地表达出来？我编写了一个测试，以尝试一种提议。

```
[Test]
public void OrderHasSnapshotOfRealCustomer()
{
    Customer c = new Customer();
    c.Name = "Volvo";

    Customer aHistoricCustomer = c.TakeSnapshot();

    Order o = new Order(aHistoricCustomer);
```

```
c.Name = "Saab";
Assert.AreEqual("Saab", c.Name);
Assert.AreEqual("Volvo", o.Customer.Name);
}
```

但我对此解决方案并不是很确定……目前，Customer是个很小的类型，但它将会增大很多。即使它现在很小，然而我们是否有兴趣跟踪Customer在该时刻所拥有的介绍人呢？

我认为这里更好的解决方案可能是创建另一个类型（它只带有我们感兴趣的那些属性），因为它在聚合之间也创建了显式的边界，但更重要的原因是，这正是底层模型所需要的。让我们创建具有最少属性的CustomerSnapshot（在某种程度上受到了[Fowler Snapshot]的目的启发，但在实现上有所不同），并将它作为值对象。只需对测试进行很少的转换（在纠正了两个编译错误之后：Customer属性的类型和构造方法的参数）：

```
[Test]
public void OrderHasSnapshotOfRealCustomer()
{
    Customer c = new Customer();
    c.Name = "Volvo";

    CustomerSnapshot aHistoricCustomer = c.TakeSnapshot();

    Order o = new Order(aHistoricCustomer);

    c.Name = "Saab";

    Assert.AreEqual("Saab", c.Name);
    Assert.AreEqual("Volvo", o.Customer.Name);
}
```

另一件要考虑的事情是，让使用者还是让Order构造方法负责创建快照。以前，我让使用者负责创建。这里改变这一点，以便构造方法再次带有Customer实例作为参数，代码如下：

```
[Test]
public void OrderHasSnapshotOfRealCustomer()
{
    Customer c = new Customer();
    c.Name = "Volvo";

    Order o = new Order(c);

    c.Name = "Saab";
    Assert.AreEqual("Saab", c.Name);
    Assert.AreEqual("Volvo", o.Customer.Name);
}
```

Order的构造方法如下：

```
//Order
public Order(Customer customer)
{
```

```
Customer = customer.TakeSnapshot();
}
```

另一件要考虑的事情是，当客户实例被创建时，是否就是创建快照的正确时刻。这时创建快照是否太早了？如果客户发生改变，会是什么情况？是否应该在转换时（即客户接受订单时）再创建快照？有很多有趣且重要的问题，但现在我还是像上面那样开始。

让我们以更新Customer作为本小节讨论的结束（参见图5-11）。

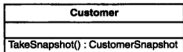


图5-11 用于创建Customer快照的新增方法

现在又到重构的时候了。让我想一下……找到Order时将Customer实例发送到GetOrders()，对于这件事情我并不完全满意。在功能方面，实际上只有客户的ID才是重要的，而且使用当前客户来获取具有历史客户信息的订单感觉上有些奇怪。不管怎样，我都想更多地考虑一下这个问题。

另一个问题是应该如何命名用于查看订单历史客户信息的Order属性。现在，它的名称是Order.Customer，但这并未指明Customer信息在时间方面的任何信息。或许Order.Customer-Snapshot更好。我认为是这样的，因此就这样更改，新的Order类如图5-12所示。

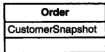


图5-12 Customer属性更改后的名称

然而，还有一件事是，Customers是直接实例化的，而不是通过CustomerFactory。我想，现在肯定是没有问题的。我们在需要的时候才添加工厂，在此之前则不添加。

或许应该再次添加该OrderFactory，并为CreateOrderLine()给它添加方法，并让它带有Product和Order两个参数（或至少带有Order）。另一方面，在不保存的情况下即可使用OrderLine是一件很好的事情（当为持久订单添加订单行时就是如此）。让我们来看一下后面是否需要改变这一点，但这又引起了另一个问题。我认为现在需要讨论一下实例的生命周期。

### 5.1.13 实例的生命周期

在前一节中，我们讲过，当将一个订单行添加到持久订单，该订单行就变成持久的。这里的意思是，当执行以下操作时，实例即开始其生命周期（实例是短暂的）：

```
Product p = new Product();
```

Product也是短暂的。因为它从未变成持久的，因此不会从数据库获取它。如果想让它变成持久的（而且在下次调用PersistAll()时保存到数据库），那么需要通过存储库来调用

AddProduct()。根据所使用的基础架构，存储库使得基础架构可以感知产品的存在。

然后，当请求存储库帮助从数据库重建实例时，被取回的实例在获取的时候是持久的。在下次调用PersistAll()时，将保存对它的所有更改。

但OrderLine是什么情况呢？我并没有要求存储库执行AddOrderLine()，但确实要求聚合根Order执行AddOrderLine()，而且由于这一点，订单行也变成持久的。在聚合之内，持久化方面应该是层叠的。

我们来总结一下我的领域模型中所需的生命周期语义，如表5-1所示。

表5-1 领域模型实例的生命周期语义总结

操 作	结果（短暂的/持久的）
新调用	短暂的
Repository.Add(实例) 或 persistentInstance.Add(实例)	在领域模型中是持久的
x.PersistAll()	在数据库中是持久的
Repository.Get()	在领域模型（和数据库）中是持久的
Repository.Delete(实例)	短暂的（而且在调用x.PersistAll时将从数据库删除实例）

图5-13中所示是用状态图形式描述的生命周期。

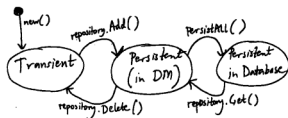


图5-13 一个实例的生命周期

**说明** Evans关于这一点也做了很多讨论……[Evans DDD]。他的解释很精辟，一定要读一读。还应该阅读的是Jordan/Russell [Jordan/Russell JDO]。JDO具有更复杂（且灵活）的语义，后面我们会发现前面创建的简单语义是不够的。

不要将这些内容与基础架构（例如NHibernate）如何对待短暂的内容混淆了。这里讨论的是领域模型实例所需的生命周期语义。然后需要让基础架构来帮助得到所需的语义。后面的章节中将讨论很多这方面的内容。

讨论到有关基础架构的内容时，目光敏锐的读者可能想知道表5-1中x.PersistAll()的意义。这里是用x来表示该未知基础架构。现在仍然可以暂时忘记这一点。

回到特性2上来，列出一位客户的订单。下一个子特性是订单必须有订单类型。

### 5.1.14 订单类型

这里，一个简单的解决方案是为每个Order添加OrderType实例，如图5-14所示。



图5-14 Order和OrderType

OrderType是否真是实体[Evans DDD]呢？我认为它不是，它是值对象[Evans DDD]。可能的值集合对于整个应用程序来说是“全局的”，它是个非常小的集合，而且是静态的。

这里，我可能走上了一条错误的道路。当然，可以将OrderType作为值对象来实现，但很多事情表明枚举就已经足够好了，特别是现在，因此我定义了OrderType枚举，并为Order添加了这样的字段。

### 5.1.15 订单的介绍人

现在该到再次讨论Order的ReferencePerson的时候了。我们已经有值对象[Evans DDD]的示例，尽管它与OrderType略有不同（OrderType是值对象候选），因为介绍人对于整个系统来说不是全局的，而是特定于每个Customer或Order的。（这是一个很好的信号，说明ReferencePerson不应该是枚举。）

图5-15中显示了正在使用的ReferencePerson。

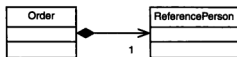


图5-15 Order和ReferencePerson

当使用值对象模式[Evans DDD]时，很显然的是应该使用.NET中的结构体，而不是类，但有一些建议说明了什么时候应该使用结构体，什么时候不该使用，而且当使用结构体时，将遇到更多的基础架构问题。此外，当使用结构体时，我们将看到更多封箱（boxing），因此这个决定并不是很明确。我实际上正在学习使用类，但通常令它们为不变的。如果你也采取这种路线，那么当重写Equals()时，需要使用类的所有值。

用于值对象ReferencePerson的Equals()如下所示。（注意，这里比较了值对象的所有字段，这一点与前面的实体不同。还要注意的，无论是否使用标识映射，都需要重写值对象的Equals()。）

```

//ReferencePerson
public override bool Equals(object o)
{
    if (o == null)
        return false;

```

```

if (this.GetType() != o.GetType())
    return false;
ReferencePerson other = (ReferencePerson) o;

if (! FirstName.Equals(other.FirstName))
    return false;
if (! LastName.Equals(other.LastName))
    return false;

return true;
}

```

**说明** Ingemar Lundberg 为以上代码给出了一个更简练的版本:

```

//ReferencePerson.Equals()
ReferencePerson other = o as ReferencePerson;
return other != null
    && this.GetType() == other.GetType()
    && FirstName.Equals(other.FirstName)
    && LastName.Equals(other.LastName);

```

这样，就已经编写完了领域模型的一些核心功能。图5-16显示了当前已开发的领域模型。

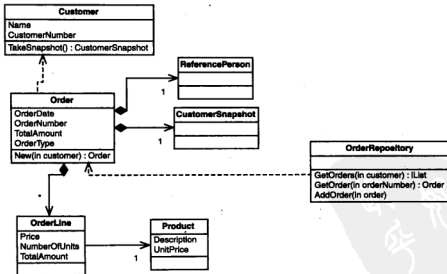


图5-16 到目前为止的领域模型实现

我认为可以先做到这里了，我们需要休息一下。休息也意味着修改。当在后面的章节中返回来精化领域模型时，将不再像现在这样使用TDD方式来描述所有小的设计决策。我们将使用一种

更精练的描述。

下面讨论一下另一种API风格，以此作为本章的结束。

## 5.2 连贯接口

至此，我们已经简要叙述了一个用于我们的领域模型的API，它是非常基本且初步的API。创建带有两个products的order的代码如下（假设已经有了一个特定的customer）：

```
Order newOrder = new Order(customer);
OrderLine orderLine;

orderLine = new OrderLine(productRepository.GetProduct("ABC"));
orderLine.NumberOfUnits = 42;
newOrder.AddOrderLine(orderLine);

orderLine = new OrderLine(productRepository.GetProduct("XYZ"));
orderLine.NumberOfUnits = 3;
newOrder.AddOrderLine(orderLine);
```

以上代码很容易理解。但它有些冗长了，而且没有做到尽可能连贯。简化上段代码的方法是为OrderLine的构造方法添加numberOfUnits，从而将它缩减为三个语句，像下面这样：

```
Order newOrder = new Order(customer);

newOrder.AddOrderLine(new OrderLine
    (42, productRepository.GetProduct("ABC")));

newOrder.AddOrderLine(new OrderLine
    (3, productRepository.GetProduct("XYZ")));
```

但它仍不是很连贯。Martin Fowler的“FluentInterface”[Fowler FluentInterface]是个很好的启发。让我们来看一下是否能够让API更连贯一些，代码如下：

```
Order newOrder = new Order(customer)
    .With(42, "ABC")
    .With(3, "XYZ");
```

这样既简短又整洁。在这段代码中，保留第一行的原因是这样非常清楚。然后，With()方法带有numberOfUnits作为参数，并且带有product标识，以便With()可以在内部使用productRepository来查找product。（暂时假设With()可以在一个众所周知的地方找到productRepository。）

With()还返回order，因此可以先后使用一系列语句。

让我们尝试使用一种变体。假设无法一次创建整个order，而是需要添加orderLine，作为从用户操作返回的结果。这样我们可能会选择使用以下语句来添加orderLine：

```
newOrder.AddOrderLine("ABC").NumberOfUnits = 42;
```

这次，AddOrderLine()返回orderLine，而不是order。可以继续采用这方法，这样就可以



通过将NumberOfUnits更改为方法（而不是再次返回orderLine），从而设置更多的orderLine值。

本节中要注意的一件重要的事情是，如果采用此方法，API会有多大程度的改善。

## 5.3 小结

我们用整章的篇幅非常详细地讨论了模型的一些最初部分。我认为详细的模型讨论最好采用代码和TDD的形式，而不是UML草图，本章正是遵循了这种思想。

在结束本章之前，总结一下现在我们在特性列表中所处的位置是一个好的思路。至此，我们已经处理了特性2、3、7和9。虽然不多，但进展是明显的。而且进展是通过大量测试保障的，这非常好。

我们将在第7章中继续精化领域模型。在下一章中，重点将有所转变。正如你可能已经注意到的，本章中已经几次涉及与基础架构有关的讨论，因此就让我们带着这些问题来学习下一章内容——准备添加基础架构。



**现**在该到探索将DDD应用于我们的问题领域的时候了。现在还没有具体的下一步骤，但你可能想知道我将何时开始处理基础架构。例如，现在还无法将订单数据保存到数据库中，也不能从数据库进行取消保存的操作（物化或重建）。事实上，我们甚至还没有思考过使用数据库，更谈不上决定了。

这是有意而为之的，因为需要将这个决定推迟一下，至少要推迟到对现有基础架构选项感到安全和熟悉的时候。假设这是我们的第一个DDD项目（当然情况并非如此），但我们假设这是第一个。

将绑定关系数据库的时间推迟一下的原因在于，这样可以将注意力集中在领域模型上，而尽可能减少分散注意力。注意，我是熟知数据库的，因此才说数据库是容易让人分心的事物。如果我来自OO一方，那么就不敢下这个结论了。

注意，这里并不是说数据库交互不重要。相反，我知道它非常重要。但暂时不接触数据库的好处在于可以更高效地尝试和探索模型的修改（无论是大的修改还是小的修改）。

此外，编写测试也容易得多，因为在两次执行之间，测试中的更改是稳定的。测试本身的执行速度也更快，这使得开发人员在每次代码更改后更容易管理测试的运行。

当然，这种方法也存在一些问题。我们已经假设能够很好地控制未来数据库交互问题，因此这并不是问题所在。相反，我们很可能需要编写一些早期的UI原型，这有两个目的：一是作为检验领域模型的一种方式，二是作为与客户交流的一种方式。如果这些原型拥有一些“活的”数据（即用户可以与其交互，或为其增加数据，等等），那么它们将比早期原型的普通数据更有用。

回忆一下，第5章结尾时初步讨论了有关增加早期UI示例的内容。如果重启之后这些数据还存在，那么用户将会发现，尝试这些示例将更有趣。（当然，如果直接开始使用计划的持久解决方案，这可能很容易做到。但是，这有可能会提高早期重构期间的开销，因此需要创建一个廉价的“假想”。）

---

**说明** 注意不要让客户（或你的老板）认为，在看到一两个原型后，整个应用程序就完成了。不要犯此类错误。

---

在这之后，能够做的第二件事情是为保存功能编写测试，仍然不涉及真实的数据库。你可能

想问这样做的目的是什么。再次强调，我希望将注意力集中在模型上，集中在“围绕”模型的语义上，等等。

当然，可以在存储库中单独处理这个问题，正如到目前为止已经做的那样，但我注意到了保持一致性以及对所有存储库只使用一种解决方案的价值。我还希望有一个能够扩展到早期集成测试的解决方案，而且最重要的是，我现在希望编写真实的存储库，而不是浪费时间编写那些应被丢弃的临时性代码！

因此，尽管前面提到现在谈基础架构为时过早，但本章将处理基础架构的准备工作，而且我们采用的方式是，确保在添加基础架构时不需要对本章的准备工作进行返工。我们需要的是编写适用于任何基础架构的代码。

## 6.1 将 POCO 作为工作方式

刚刚也说过，我们尽量将注意力集中在应用程序的主体上，而不因那些与基础架构有关的事情而分心。Plain Old Java Object (POJO) 和 Plain Old CLR Object (POCO) 是在 Java 世界中发起的运动，其目的是反对 J2EE 及其对应用程序的巨大影响，例如它如何增加了每件事的复杂性，并使 TDD 几乎不可能实现。Martin Fowler、Rebecca Parsons 和 Josh MacKenzie 发明了 POJO 这个术语，用于描述不包含“哑”代码的类（只有执行环境才需要这样的代码）。类应该将注意力集中在当前业务问题上。领域模型中的类不应包含任何其他事情。

**说明** 这项运动是轻量级 Java 容器的主要灵感之一，例如 Spring [Johnson J2EE Development without EJB]。

在 .NET 世界中，经过一段时间，Plain Old... 才引起人们的注意，但它现在被称为 POCO。

POCO 在某种程度上是一个已有的术语，但它并没有特指与持久化有关的基础架构。当我与 Martin Fowler 讨论这一点时，他说或许持久化透明 (Persistence Ignorance, PI) 是一种更好、更清晰的描述。我也赞成他这种观点，因此本章从现在开始将改用 PI 来描述。

### 6.1.1 实体和值对象的 PI

那么假设我们想要使用 PI。这将会是什么情况呢？PI 意味着干净、普通的类，在这些类中，我们关注当前业务问题，而不涉及与基础架构有关的方面。但这还远远不够。如果我们看一下 PI 不是什么，就会简单得多了。首先，通过一个简单的“石蕊试验”，可以看到在领域模型中是否具有对任何与外部基础架构有关的 DLL 的引用。例如，如果使用 NHibernate 作为 O/R 映射工具，并具有对 `nhibernate.dll` 的引用，那么这就是一个很好的信号，说明已经为领域模型增加了一些代码，它们并不是真正的核心代码，而是会令我们分心的代码。

那么分心的地方有哪些？例如，如果对于持久对象使用基于 PI 的方法，那么需求中不会包含以下任何事情。

- 从特定的基类（对象除外）进行继承。

- 只通过一个提供的工厂进行实例化。
- 使用专门提供的数据类型，例如为集合提供的数据类型。
- 实现一个特定接口。
- 提供特定构造方法。
- 提供必需的特定字段。
- 避免某些结构。

以上列表中至少还应包括一点，但这一点过于明显了，所以没有列出。我们应该不必编写数据库代码，例如对领域模型类中存储过程的调用，但这点太明显了，因此上面并没有专门写出它。

让我们更仔细地看一下其他每个要点。

### 1. 从特定基类进行继承

框架在支持持久化方面有一个很常见的需求是从框架所提供的一个特定基类进行继承。

以下代码可能看起来并不是很差：

```
public class Customer : PersistentObjectBase
{
    public string Name = string.Empty;
    ...

    public decimal CalculateDepth()
    ...
}
```

虽然它并不是很差，但确实带有一些对我们来说不是最好的语义和机制。例如，我们对于 `Customer` 类使用了唯一的继承可能性，因为 .NET 只有单继承。当然，这是否是一个大的问题还存在争议，因为我们通常可以“绕过”它进行设计。

如果已经开发了领域模型，并且现在想让它成为持久的，那么这个问题更糟糕。继承需求可能要求对领域模型做出很多修改。当使用 TDD 方法开发领域时，情况更是如此。我们从一开始就被约束不能使用继承，但由于持久化需求，又不得不保留继承。

要警惕的是，继承是否为子类引入大量公共功能，这可能使得子类的使用者不得不吃力地应付那些他不感兴趣的方法。

还有一点是，它通常不像前一个示例那样整洁，但在大部分时间里，`PersistentObjectBase` 都强制要求我们为 `PersistentObjectBase` 中的方法提供一些方法实现，就像在模板方法模式中的一样 [GoF Design Patterns]。这仍然不是一个严重的问题，但使问题进一步累积。

**说明** 这并不一定必须成为一个需求，而可以看作是提供了一种方便，允许从框架所需的接口实现中获得最大价值（如果框架是那种风格的话）。我们将在稍后的部分讨论这个常见需求。

这就是在 Christoffer Skjoldborg 和我开发的 Valhalla 框架中所采用的做法。但坦白地讲，在这种情况下，基类 `EntityBase` 要兼顾的工作太多了，以至于用自定义代码实现接口（而不是从 `EntityBase` 继承）实际上只是一个理论上的选项。

## 2. 只通过提供的工厂进行实例化

不要误会我的意思，我并不反对使用工厂。不管怎样，当不是我自己的合理决定时，我并不会强迫使用它们。这意味着我不会编写这样的代码：

```
Customer c = new Customer();
```

而需要编写以下代码：

```
Customer c = (Customer)PersistentObjectFactory.CreateInstance
    (typeof(Customer));
```

---

**说明** 我知道，你认为我使用了非常长的名字而不尊重事实，但它确实不是很好，不是吗？

```
Customer c = (Customer)POF.CI(typeof(Customer));
```

---

再次强调，这并不是严重错误，但在大多数情况下不是最佳选择。与第一个实例化代码相比，此代码只是略显奇怪，而且这样的代码经常会增加测试的复杂性。

通常，必须使用提供的工厂的原因之一在于，我们将因此从脏检查（dirty checking）中获得帮助。因此，领域模型类可以动态地生成子类，而且在子类中，可以在属性中维护一个（或几个）脏标志（dirty-flag）。工厂使得这对于使用者是透明的，这样它就对子类进行实例化，而不是对工厂使用者所要求的类进行实例化。遗憾的是，为了做到这一点，我们还必须使属性保持为虚拟的，而且不能使用公共字段（这是略微减轻了PI级别的两个小的细节）。（可以使用公共字段，但在生成的子类中，它们不能被“重写”，如果子类的目的是处理脏跟踪，那么这就是个问题。）

---

**说明** 当在.NET中使用面向方面编程（AOP）时，有几种不同技术，其中我们刚刚讨论过的运行时生成子类可能是最常用的。我经常看到一些人不得不将成员声明为虚拟的，目的是可以作为缺点拦截（或通知），但Roger Johansson为我指明了一些事情。假设想要禁止重写成成员，从而避免额外工作，并避免支持生成子类的责任。那么这个决定既应该影响普通的子类生成，也应该影响那些为AOP原因而使用的子类生成。如果将成员声明为虚拟的，那么就已经准备了要重定义它，同理，重定义既是通过普通子类生成完成的，也是通过AOP子类生成完成的。

这是有意义的，不是吗？

---

用这种方式解决的另一个常见问题是对延迟加载（Lazy Load）的需求，但我将把它用作下一节的示例。

## 3. 使用“专门”提供的数据类型，例如为集合提供的数据类型

对于领域模型类中的集合使用特殊的数据类型并非罕见，特殊的地方就在于“如果可以自由选择，那么就不会使用这些类型”。

这种需求的最常见原因可能是为了支持延迟加载[Fowler PoEAA]，或者说是隐式的延迟加载，以便不必主动编写代码让它发生。（延迟加载意味着从数据库获取数据是及时的。）

但特定数据类型也可能引入其他功能，例如特殊的删除处理，这样，一旦从集合中删除了一

个实例，该实例将向操作单元（Unit of Work）[Fowler PoEAA]注册这次删除操作。（操作单元用于跟踪那些在当前逻辑操作单元末尾应该对数据库执行的操作。）

**说明** 你是否注意到我刚刚说过特定数据类型会引入功能？是的，我并不想过多地否定NPI（Not-PI）。

我们可以从这种双向性中获得帮助，从而不必亲自为其编写代码。这是AOP解决方案为我们提供帮助的另一个示例。

#### 4. 实现特定接口

然而，领域模型在实现持久化方面的另一个常规需求是，它们需要实现一个或多个基础架构提供的接口（infrastructure-provided interface）。

如果实现接口所需编写的代码很少，那么这自然就是一个较小问题，反之，它将是一个较大的问题。

基于接口的功能的一个示例可能是：在无需使用setter方法的情况下，用来自数据库的值来填充实例（setter方法可能具有一些我们在重建期间不想执行的特定代码）。

另一个常见示例是提供用于对实例中状态进行优化访问的接口。

#### 5. 提供特定构造方法

另一种提供值（这些值用于从数据库重构实例）的方式是提供特定构造方法，这些构造方法与当前业务问题根本无关。

还有一种情况是需要一个默认构造方法，以便框架可以很容易实例化领域模型类（作为对数据库执行Get操作的结果）。再次强调，它并不是一个非常严重的问题，但不管怎样都是一件分散注意力的事情。

#### 6. 提供必需的特定字段

一些基础架构解决方案要求领域模型类提供特定字段，例如基于Guid的Id字段，或基于int的Version字段。（基于Guid的Id字段是指Id字段使用Guid作为数据类型。）这简化了基础架构，但可能使得领域模型的开发工作更难一些。至少它以我们不需要的方式对类产生了影响。

#### 7. 避免某些结构/强制使用某些结构

前面曾经提到过，我们有可能被强制使用虚拟属性，即使实际上并不想这样做。还有一种情况是不得不避免特定构造方法。一个典型示例就是只读字段。只读（像使用关键字readonly时）字段无法从外部设置（使用构造方法除外），当创建纯粹的PI领域模型类时，就需要使用只读字段。

将私有字段与get-only属性一起使用很接近只读字段，但二者并不完全相同。一个有争议的问题是，只读字段是否是最清楚地表达了目的性的解决方案。

**说明** 一个已经有过很多讨论的问题是，.NET属性在修饰领域方面是否起到好的作用（即，使用如何保持领域模型的信息来修饰领域模型）。

我的观点是，这样的属性可能是一件好事，而且如果它们被看作可以重写的默认信息，那么实际上不会使PI的级别下降。我认为主要问题在于它们是否过于冗长，以至于分散了读者对代码的注意力。

### 6.1.2 是否使用 PI

使用PI还是不使用PI——当然这并不是一个完全黑白分明的问题。这当中存在一个折中区域，但现在，如果我们有了使用PI的意图，应该感到高兴，但不是绝对地使用它。任何极端的事情都会招致高昂代价。我们将在第9章讨论基础架构解决方案时再回头讨论这一点。

---

**说明** 在现实生活中，一个完全黑白分明的例子是什么？我经常提醒我妻子的一件事是，当地人说“那位女士怀孕了”的时候。

---

到目前为止我们尚未讨论的一件事是，是否使用PI还取决于我们正处在哪个“时间”点上。

### 6.1.3 运行时与编译时 PI

到目前为止，我们在不涉及时间的上下文中讨论了PI，但可能最重要的是在编译时使用PI，运行时则次之。我知道你会问：“这意味着什么？”假设我们永远不必处理（甚至看不到）那些已经创建的与基础架构有关的代码。与手工维护类似代码相比，这种解决方案可能更好。

对这个主题的认识与个人认知有关，因为执行别人编写的东西是有争议的。调试过程可能变成一场噩梦！

---

**说明** Mark Burhop做出了如下评论：

这是20世纪90年代早期来自C程序员对C++的最初争论。“C++内置了我没有编写的新代码。”“C++隐藏了实际正在发生的事情。”我并不认为这样的论点站得住脚。

---

与Java相比，要想为.NET类插入字节级的代码也更难。它不受框架的支持，因此我们只能靠自己了，而这在大多数情况下无法完成。

通常采用的做法是使用一些替代技术，例如将运行时生成子类与提供的工厂结合起来使用，但与插入代码相比区别不大。我们将其统称为发送代码（emitting code）。

### 6.1.4 PI 实体/值对象的代价

我猜想对所有这些事情的一个可能的反应是：“PI看起来好极了，为什么不一直使用它呢？”当每件事情看起来都非常整洁、非常好而且没有缺点时，缺点就会出现了，这是一条自然法则（至少是软件法则）。在这里，我认为缺点就在于开销。

本章前面并没有提到这一点：要达到高级别的PI（至少是运行时PI），将以牺牲速度为代价，因为我们不得不使用反射，这会产生极高的代价。（如果你认为编译时PI已经足够好了，那么不必使用反射，但可能需要使用一个AOP解决方案，这样可以得到更好的性能。）

通过紧凑循环（tight loop）中的某种操作，我们很容易证明用反射方式读写字段比普通调用方式慢好几个数量级。然而，代价是否过高？这显然取决于具体情况。我们必须运行测试来了解这是否适用于自己的情况。不要忘记，与领域模型中的很多操作相比，跳转到数据库的代价是很

高的，但同时，我们不能同类型进行比较。例如，普通读取操作与基于反射的读取操作是没有可比性的。

### 1. 有关速度的典型示例

我们通过一个示例来使读者对整件事情有更好的理解。持久化框架中的一个常见操作是，在一个场景结束时决定是否应将实例存储到数据库中。一个常见的解决方案是，如果要存储实例，就让它负责发出IsDirty信号。或者更好的做法是，当实例变脏后，可以自己发信号给操作单元，这样操作单元在存储更改的时候将记住该信号。

但这需要对PI进行一定程度的“滥用”，除非已经使用了AOP。

---

**说明** 此解决方案还有其他一些缺点，例如，如果是通过反射完成的，那么它将不会注意到更改，因此不会存储实例更改。这个缺点有些麻烦。

---

一种替代解决方案是不发出任何信号，而是让基础架构从数据库获取实例时记住它的状态。然后，在存储时，将当前状态与从数据库读取时的状态进行比较。

你是否注意到，这不仅仅是普通读取操作与基于反射的读取操作的比较，而是完全不同的两种方法的比较，具有完全不同的性能特征。为了获得真实的体验，你可以自己进行比较。从数据库中获取100万个实例，然后修改一个实例，并测量一下两种情况下存储操作的时间差异。我知道，这是另一个麻烦的情形，但仍是我们需要考虑的。

### 2. 其他示例

以上是有关速度代价的一些事情，但这并不是全部的代价。前面曾经指出的另一个代价是，如果尝试使用高级别的PI，那么可能会自动得到更少的功能。我们已经检查了在放弃一些PI时可能免费获得的很多特性，例如自动双向支持和自动隐式延迟加载。

脏跟踪也并不是仅与性能有关。当绘制表单时，使用者也可能对使用该信息很感兴趣，例如，想知道启用哪些按钮。

因此，像往常一样，有一个折中问题。对于是否使用PI这个问题，折中就是两个因素的比较：一方面是开销和较少的功能性，另一方面是应用程序核心中那些分散注意力的代码，这些代码将我们连接到某种特定的基础架构，而且也使得TDD的使用更困难。利弊共存，这也是合理的，不是吗？

### 3. 代价结论

因此，总的结论是注意折中并认真选择。例如，如果得到了所需的东西，同时伴随着可以忍受的缺点，那么就不要过分苛求。

也就是说，我现在位于赞成PI的阵营中，主要是因为它对于TDD来说是很好的，而且也可以让实体和值对象更整洁更清晰。

如果首选方法不同，那么也将产生巨大的差别。如果你喜欢从代码开始，那么可能会很喜欢PI。如果你使用集成工具，从详细的UML设计开始，并从这个设计中生成领域模型，那么PI对你来说可能根本就不重要。



但是，有些方面对于领域模型来说比对实体和值对象更重要。我想到的就是存储库。但我们几乎没有讨论过将PI用于存储库。

### 6.1.5 将 PI 用于存储库

我承认，将PI用于存储库意味着对存储库执行推（push）操作。这是因为存储库的目的给了使用者一个假象，即完整的领域模型实例集合就在眼前，只要遵守协议，到存储库获取实例即可。产生这种假象的原因是：存储库在特定情况下与基础架构对话，而与基础架构对话并不是使用PI时要做的事情。

例如，存储库需要拉（pull）一些东西，以便让基础架构工作。这意味着带有存储库的程序集（assembly）需要有一个到基础架构DLL的引用。这进而又意味着我们必须做出选择：是希望存储库位于单独的DLL中，还是希望它与领域模型分离，或是希望领域模型引用一个基础架构DLL（但我们很快会讨论一种灵活处理这个问题的解决方案）。

#### 1. 测试存储库

还有一种情况是，当我们想要测试存储库时，它们被连接到O/R映射工具和数据库。

---

**说明** 暂时假设我们将使用O/R映射工具。我们将在随后的几章中更彻底地讨论不同的选项。

---

与分别测试实例和值对象相比，这突然使测试变得更困难了。

当然，你可以为O/R映射工具建立一个模拟实现。我自己没有这样做过，但在是否“划算”方面，还不尽人意。如果不这样做，可能会节省大量工作。

#### 2. 进行小规模集成测试

在前面的章节中，没有真正显示任何集中关注存储库的测试代码。大多数有意义的测试都应该使用领域模型。如果未做到这一点，那么这可能就是一个信号，说明你的领域模型还没有像它应该的那样丰富（如果想从它那里获得最大价值的话）。

也就是说，我的确在一些测试中使用了存储库，但实际上多是作为小的集成测试，目的是看一下使用者、实体和存储库之间是否按计划进行了合作。事实上，这是存储库在为领域模型提供持久能力方面的优势之一（与其他方法相比），因为很容易编写存储库的伪版本。问题在于我编写了太多的哑代码，过后不得不丢弃这些代码，或者至少需要在另一个程序集中重写它，而在这个程序集中，存储库并不仅仅是伪版本。

还有一件事情是，我从伪版本获取的语义实际上并不“正确”。例如，以下代码难道不奇怪吗？

```
[Test]
public void FakeRepositoryHaveIncorrectSemantics()
{
    OrderRepository r1 = new OrderRepository();
    OrderRepository r2 = new OrderRepository();

    Order o = new Order();

    r1.Add(o);
    x.PersistAll();
}
```

```
//This is fine:
Assert.IsNotNull(r1.GetOrder(o.Id));
//This is unexpected I think:
Assert.IsNull(r2.GetOrder(o.Id));
}
```

**说明** 目光敏锐的读者会发现，从前一章开始，我就决定将AddOrder()更改为Add()。

前段代码有些超前了，因为我们很快将要讨论保存功能。不管怎样，我想证明的是，到目前为止，所使用的伪版本存储库并没有像预期那样工作。尽管我认为我已经用PersistAll()对持久化做了所有更改，但只有第一个存储库实例能够找到订单，而第二个存储库实例就找不到了。你可能感到奇怪为什么我要将代码写成那样，这是一个好问题，但我觉得这这也是一个不礼貌的问题。

我们能做的是为每个存储库建立模拟实现，以测试实体、存储库与使用者之间的合作情况。这是一项简易的工作，而且是测试使用者和实体的很好方式。然而，存储库本身的测试价值当然并不大。我们相当于回到了问题的原态，因为我们想做的是进一步为O/R映射工具（如果它就是用于处理持久化的工具的话）建立模拟实现，我们已经讨论过这一点了。

### 3. 较早的方法

因此，首先有存储库是好的，特别是当涉及可测试性的时候。因此，我过去常常接受这个麻烦，并通过为每个存储库创建一个接口来处理此问题，然后创建两个实现类，一个用于伪版本的基础架构，另一个用于真实基础架构。它可能像以下代码这样。首先是领域模型程序集中的一个接口：

```
public interface ICustomerRepository
{
    Customer GetById(int id);
    IList GetByNamePattern(string namePattern);
    void Add(Customer c);
}
```

然后是位于两个不同程序集中的两个类（例如FakeCustomerRepository和MyInfrastructureCustomerRepository），但它们都在一个名称空间——领域模型中，除非有多个领域模型的分区（参见图6-1）。

这意味着对于存储库来说，领域模型本身不受所选择的基础架构的影响，如果这不耗费任何代价，那么就是非常好的。

但它确实是有代价的。它也意味着我必须为每个聚合根编写两个存储库，而且每种情况下都具有完全不同的存储库代码。

进一步来说，它意味着存储库的生产版本位于另一个程序集中（伪版本的存储库也是如此），尽管存储库是领域模型本身的一部分。你会说：“两个额外的程序集，这并不是大问题。”但对于大的应用程序来说（其中领域模型被分为几个不同程序集），我们就会知道这通常并不仅仅是两个额外存储库程序集的问题，而是领域模型程序集的总数乘以3。这是因为每个领域模型程序集

将拥有其自己的存储库程序集。

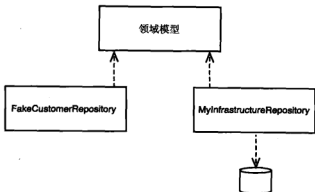


图6-1 两个存储库程序集

尽管我认为它是一个消极方面，但也不会像伪版本存储库中包含不良代码那样糟糕。只是感觉不好而已。

#### 4. 是否有更好的解决方案

我决定要尝试的解决方案是创建一个抽象层，我将其称为NWorkspace [Nilsson NWorkspace]。它是一组适配器接口，我已经以伪版本形式为它们编写了实现。伪版本只是两个级别的散列表，一组散列表用于持久实体（模拟数据库），另一组散列表用于操作单元和标识映射（标识映射对当前加载的标识，通常是主键进行跟踪）。

我所编写的另一个实现则用于特定的O/R映射工具。

**说明** 从现在开始，当我使用NWorkspace这个名称时，你应该将它看作“持久抽象层”。NWorkspace只是个示例，它本身并不重要。

借助于这个抽象层，可以将存储库移回领域模型中，而且每个聚合根只需要一个存储库实现。一个存储库可同时用于O/R映射工具和伪版本。当用于伪版本时，对数据库不是持久的，而只是在内存中保持实例的散列表，但与用于O/R映射工具时具有类似的语义（参见图6-2）。

也可以将伪版本序列化到文件，或者从文件进行反序列化，这对于创建符合要求的、现实的且易于重构的应用程序早期版本是非常有用的。

另一个感觉很容易实现的可能性（至少是对于小的抽象层API而言）是为基础架构（而不是每个存储库）建立模拟实现。事实上，这并不是为一个基础架构产品建立模拟实现的问题，而是所有基础架构产品都将同时拥有对抽象层的适配器实现（如果发生这种情况，那么除了我编写的那两个实现以外，还会有其他的实现——可能情况并不会这样）。更为重要的是，然后就是为抽象层建立模拟实现了。

现在谈论PI存储库还为时过早，但利用此解决方案可以避免在领域模型中对基础架构的引

用。也就是说，在实际应用程序中，不管怎样我都已经将存储库保存在一个单独的程序集中。我认为它澄清了耦合，也使得一些困难工作更易于实现，而且允许一些存储库方法在必要时使用原始SQL（使用连接字符串作为是否应该使用优化代码的标记）。

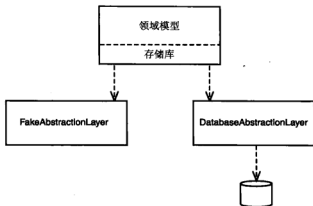


图6-2 使用了抽象层的一组存储库

然而，与其引用持久框架，不如通过适配器接口来引用NWorkspace DLL。但这看起来步伐迈得太大了，虽然方向正确。还有一种情况是存储库中只有很少或没有抽象，它们相当“直接”（即，是否以任何适当的方式找到NWorkspace API）。

因此，我们不必依照基础架构供应商的API的代码来编写一组存储库，同时再用哑代码编写另一组存储库为，而只需依照一个（单纯的）标准API来编写一组存储库即可。

---

**说明** 我为我的罗嗦表示歉意，但我必须重申：这是我追求的概念！我自己的实现根本不重要。

---

让我们寻找另一个术语来描述这些存储库，而不将其称为PI存储库。单组存储库（single-set Repository）如何？好，现在当构建一个单组存储库时，我们有了一个描述它的术语了，这组存储库同时用于伪版本情形和带有数据库的情形。比命名这些存储库更有趣的事情可能是观察它们的行为。

### 5. 单组存储库中的一些代码

为了提醒读者伪版本存储库中的代码是什么形式的，以下是第5章中的一个方法：

```
//OrderRepository, a Fake version
public Order GetOrder(int orderNumber)
{
    foreach (Order o in _theOrders)
    {
        if (o.OrderNumber == orderNumber)
            return o;
    }
    return null;
}
```

}

这段代码并不是特别复杂，但它是不好的代码。

如果我们假设`OrderNumber`是`Order`的一个标识字段[Fowler PoEAA]（标识字段是指将数据库中的行绑定到领域模型中实例的字段），那么当我们使用抽象层时，代码可能会像下面这样（以下代码中的`_ws`是`IWorkspace`的一个实例，`IWorkspace`是抽象层的主接口）：

```
//OrderRepository, a single-set version
public Order GetOrder(int orderNumber)
{
    return (Order)_ws.GetById(typeof(Order), orderNumber);
}
```

我认为这段代码相当简单且直接。而且方法既可用于伪版本，也可用于使用真实基础架构的时候！

### 6.1.6 单组存储库的代价

这样，我就有了另一个抽象。抽象开始越来越多了，你是否这样认为？但另一方面，我相信每个抽象都增加了价值。

当然，这也是有代价的。增加的抽象层的最明显代价是，必须对正在使用的O/R映射工具完成运行时转换。理论上，O/R映射工具可以有抽象层的本地实现，但要想实现这一点，必须创建这样的（真正常用的）抽象层。

此外，为特定的O/R映射工具构建抽象层和适配器也会有代价。这是典型的与框架有关的问题。构建框架的代价很大，但如果框架有用的话，它可以被使用很多次。

如果运气好的话，正在使用的基础架构将会有适配器实现，这样就不再有这方面的代价了，至少消除了构建框架的代价。尽管如此，但还是有更多的代价。我们不仅必须学习所选择的基础架构，而且还要学习抽象层，这些代价是不容忽视的。

---

**说明** 过去，我们的工作容易得多，因为只需要对Cobol和文件略知一二即可。现在，我们必须成为C#或Java、关系数据库、SQL、O/R映射工具等方面的专家。如果某人尝试通过添加另一个层来使整个事情变得简单，那么这将使事情变得更复杂，特别是对于初学者。

---

当然，另一个代价是抽象层将是一种最小公因子。我们将无法发挥基础架构的最大能力。当然，我们总是可以绕过抽象层，但这就会导致复杂性和外部存储库代码等一些代价。因此，重要的是确定使用抽象层会满足我们的多少需要，是30%、60%，还是90%。如果比例不高，那么是否应该使用抽象层就值得怀疑了。

下面让我们暂时返回使用者的身份，关注一下更改的保存功能。

## 6.2 对保存场景的处理

正如前一章结尾时所说的，从现在开始，我将加快速度，不讨论我思考过程中的所有步骤。

相反，我们将直接移动到已决定的解决方案。不是“已完成的”解决方案，而是“暂时”决定的解决方案。

这样，我们将讨论测试，特别是将其作为一种澄清方式。

现在，我们将从使用者的角度来讨论保存场景。以下是一个测试，它显示了如何保存两个新的Customer。

```
[Test]
Public void CanSaveTwoCustomers()
{
    int noOfCustomersBefore =
        _GetNumberOfStoredCustomers();

    Customer c = new Customer();
    c.Name = "Volvo";
    _customerRepository.Add(c);

    Customer c2 = new Customer();
    c2.Name = "Saab";
    _customerRepository.Add(c2);

    Assert.AreEqual(noOfCustomersBefore,
        _GetNumberOfStoredCustomers());

    _ws.PersistAll();

    Assert.AreEqual(noOfCustomersBefore + 2,
        _GetNumberOfStoredCustomers());
}
```

乍看起来，以上代码相当简单，但它“隐藏着”很多我们先前未讨论过的事情。首先，它揭示了我希望能够编写哪类使用者代码。我希望能够对几个不同实例做很多事情，然后通过一个单独调用（例如PersistAll()）来保存所有工作（对\_GetNumberOfStoredCustomers()的调用将访问持久引擎，以检查持久客户的数量。直到调用PersistAll()之后，持久客户的数量才会增加）。

这个问题中还有一点：存储库是在实例化时通过\_ws输入的。这样，我就可以控制哪些存储库应该参与到相同的操作单元中，哪些存储库应该被隔离到另一个操作单元中。

然而，另一件有趣的事情是，我让存储库来帮助通知（Add()调用）操作单元有个新实例需要保留到下一个PersistAll()调用。这里的意思是，对于持久实例，我希望它保持生命周期（这个问题在第5章已经讨论过，这里不再重复）。

值得注意的是，如果希望将聚合根与操作单元关联起来就已经足够，那么当调用PersistAll()时，作为聚合一部分的那些实例以及聚合根所达到的那些实例都应该被保存到数据库。

再次强调，聚合为我们提供了很多帮助，它们提供了一个工具，可帮助我们知道将被保存的整个图的大小，因为聚合根被标记为已保存。此外，聚合还有助于简化。

**说明** 我们通常可以通过配置O/R映射工具来确定可达性 (reachability) 的范围。但即便如此, 聚合仍不失为一个非常好的指南, 即, 在进行配置时, 可以使用聚合来确定可达性的范围。

让我们更仔细地看一下迄今为止所讨论的决策背后的推理。

#### 决策的原因

为什么要选择以上做法呢? 首先, 我希望使用操作单元模式。我希望利用它的特点来创建逻辑操作单元。

这样就可以对领域模型进行很多更改, 收集操作单元中的信息, 然后要求操作单元将收集到的更改保存至数据库。

操作单元有几种风格。我喜欢的一种风格是尽量让它对使用者透明, 这样唯一要传递的消息就是对存储库调用Add() (然后存储库再与操作单元进行交互)。

如果通过存储库进行重建, 那么操作单元实现可以注入某个对象, 此对象能够收集有关更改的信息。此外, 还可以在读取时拍快照, 然后用它来控制操作单元在保存的时候所做的更改。

我还选择在存储库外部控制是否进行保存 (PersistAll())。在这个特殊示例中, 只需在CustomerRepository上直接调用PersistAll()即可, 但我没有选择这样做。为什么呢? 为什么不让存储库完全隐藏操作单元? 其实, 也可以这样做, 但我经常发现需要对单一逻辑单元中的几个聚合进行同步更改 (因此也需要对几个不同存储库进行同步更改), 这就是原因所在。因此, 这样的代码不仅是可编写的, 而且还非常典型。

```
Customer c = new Customer();
_customerRepository.Add(c);
```

```
Order o = new Order();
_orderRepository.Add(o);
```

```
_ws.PersistAll();
```

一种替代方法可能像下面这样:

```
Customer c = new Customer();
_customerRepository.Add(c);
_customerRepository.PersistAll();
```

```
Order o = new Order();
_orderRepository.Add(o);
_orderRepository.PersistAll();
```

但这样我就有了两个不同的操作单元实例和两个标识映射 (这并不是必需的, 但出于讨论的目的, 我们假设是这样), 如果我们不仔细的话, 这可能产生非常奇怪的效果。毕竟, 第一个示例中的前5行是一种情况, 而且由于此原因, 我发现更直观且适当的方式是将它看作有关如何处理操作单元和标识映射的一种场景, 即此场景只有一个操作单元和一个标识映射。

另一个可能发生的问题是, 当存储库隐藏操作单元时, 这可能意味着有两个数据库事务。这

进而又意味着，当操作结果不符合预期时，我们可能必须准备添加补偿操作。在前面的示例中，如果被添加到数据库中的是Customer，而不是Order，那么结果还不算太坏。然而，取决于聚合的设计，这可能成为一个问题。

也就是说，聚合“应该”设计成这样：在调用PersistAll()时保持一致的状态。但聚合之间的松散关系通常无法满足这样的严格要求。这可能使我们更倾向于第二种解决方案。另一方面，第二种解决方案将在同一个PersistAll()中存储两个完全无关的customers（如果这两个customers关联到存储库的话）。与把customer及其order分组到一起相比，这实际上不太重要。聚合是有关对象（而不是类）的。

第二个示例中解决方案的关键之处在于，是否一个聚合来自一个数据库，而另一个聚合存储在数据库服务器的另一个数据库中。这样，最简单的方法可能就是有两个操作单元实例，每个实例用于一个数据库。因此，解决方案2的耦合度就略低一些。

---

**说明** 我甚至可以用Add()来执行事务，但这样就有了不同于前面已经讨论过的语义。一定要在正确的时刻调用Add()，这是至关重要的。对于我选择的解决方案来说，只要在PersistAll()之前调用即可。

用Add()来执行事务时，还意味着它肯定不是“收集所有更改并在调用PersistAll()时保存它们”，从目前的解决方案来看，这仍然是很难实现的。

既然我们已经着手做了，为什么不把整个事情封装到实体中，以便可以编写以下代码呢？

```
Customer c = new Customer();
c.Name = "Saab";
c.Save();
```

我认为这不是我所喜欢的风格，它破坏了单一职责原则（SRP）[Martin PPP]，而且PI级别较低。这也进入了“尝试”领域。

---

另外还存在不一致的问题，即一个保存操作进行得很好，而其他保存操作则不然。（你可能会说，物理事务和操作单元不必保持相同，但这增加了复杂性。我认为：如果不必做那些不应该做的事，那就不要做。）

然而，使用我所喜欢的技术，如果确实需要的话，完全可以实现一次存储一个聚合，方法就是让存储库隐藏操作单元和标识映射。代码如下：

```
Customer c = new Customer();
_customerRepository.Add(c);
_ws1.PersistAll();
```

```
Order o = new Order();
_orderRepository.Add(o);
_ws2.PersistAll();
```

---

**说明** 对于前一个场景来说，最终结果与使用单个\_ws是相同的，但这取决于特定的示例。

---



这里的意思是：根据需要，可以有一个操作单元标识映射，也可以有多个。这是我非常喜欢的一个灵活性，也是我选择的原因之一，也就是说，操作单元属于领域模型使用者（即应用程序层或表示层），而不属于领域模型本身。

如果假设每个存储库都有其自己的标识映射，那么当从两个不同存储库引用同一个订单时，就容易产生混淆，至少当在两个存储库中对同一个逻辑订单（但有两个不同实例）进行修改时易产生混淆。

在风险方面，问题可归结为我们愿意承担哪种风险。是否愿意冒提交未完成实例的风险（因为PersistAll()将处理比我们预期的更多的实例）？还是愿意冒忘记提交更改的风险（因为我们必须记住对哪些存储库调用了PersistAll()）。

这并不是说它是一个毫无问题的解决方案，再次强调，标识映射和操作单元适用于此场景。

---

**说明** 通常，使用哪种编程模型是个人决定。每种编程模型都是利弊共存的。

---

前面曾反复地一起提及操作单元和标识映射。这是很常见的一对组合，不仅在我的书中是这样，而且在产品中也是如此，例如，JDO中的持久管理器[Jordan/Russell JDO]和Hibernate中的会话[Bauer/King HiA]。

我认为这种组合值得作为一种模式，而且我曾考虑将它写下来，但当我与Martin Fowler讨论时，他告诉我，当他在[Fowler PoEAA]中讨论标识映射和操作单元时曾讨论过这一点了。这已经足够了，因此我决定不再重复讨论。

现在我们已经讨论了很多，但还没有实际操作。让我们开始建立伪版本机制，并看一下结果如何。

## 6.3 建立伪版本机制

下面一段时间，我们的讨论将基于接口，至少开始时是这样。我们已经接触了几个方法，但让我们从头开始。抽象层接口的一个简化版本如下（本章前面已将此接口称为IWorkspace）。

```
public interface IWorkspace
{
    object GetById(Type typeToGet, object idValue);
    void MakePersistent(object o);
    void PersistAll();
}
```

现在整个接口是相当直观的。第一个方法是GetById()，它用于重构来自数据库的对象。我们要做的事情就是输入对象类型和标识值。

第二个方法是MakePersistent()，它用于将新实例与IWorkspace实例关联起来，以便可以在下次调用PersistAll()时保存它们。最后，PersistAll()用于将操作单元中发现的数据保存到数据库中。

如果使用GetById()从数据库读取实例，那么就不需要MakePersistent()了。因为实例已

经与操作单元关联到一起。

到目前为止，API非常简单，而且我认为在这个抽象层中保持较低的复杂性是非常重要的。现在的功能并不完备，所以我们需要添加更多功能。

### 6.3.1 伪版本机制的更多特性

第一个能想到的事情是，需要将事务作为一个非常重要的概念来处理，至少要站在一个正确立场上来处理。另一方面，这对于大多数UI程序员来说并不重要（对于UI代码也不重要）。也就是说，我不想把事务管理的职责施加到UI程序员的身上，因为这会让他们过多地分心，而且这也是他们无法承担的重要任务。我仍然希望有适当的事务范围，以便不仅只有一个预定义的可能的

事务集。

因此，我的目标非常类似于COM+中那些声明性事务，但我选择了一个完全不同的API。我没有选择通过设置属性来描述它们是否需要事务，PersistAll()将在一个显式的事务中完成其所有内部工作，尽管并没有显式地要求它这样做。

我知道，从表面上看来，这对于很多资深人员来说过于简单了，对我来说也是如此，因为我相信事务处理如此重要，以至于我喜欢手工处理它。然而，如果目标是能够处理90%的情形，那么PersistAll()就可以很好地使用显式事务，而且也很简单。

再次强调，它听起来过于简单了，当然也存在问题。一个典型的问题就是日志。假设登录到数据库服务器，如果普通工作失败，我们并不希望日志操作失败。这是一个很容易处理的问题。我们只需将一个单独的工作区用于日志即可。如果想让日志使用抽象层，可以仅仅将它作为服务来实现，此服务可能与抽象层毫无关系。事实上，我们很有可能会使用第三方产品，或者某种类似于log4net[Log4Net]的工具。它不会干扰抽象层的事务API，也不会受其干扰。

另一个问题是，与即将调用的PersistAll()一样，可能需要在同一个事务中调用GetById()方法。这并不是默认发生的事情，但如果要强制将其设为默认行为，可以在GetById()之前调用以下方法：

```
void BeginReadTransaction(TransactionIsolationLevel til)
```

为进一步强调这一点，对GetById()还有一个重载，以请求一个独占锁，但这会得到一个警告标志。使用这种方法时要清楚知道自己在做什么！例如，在调用BeginReadTransaction()之后或以独占锁读取之后，并且在调用PersistAll()之前，这段时间内不应该有任何用户交互。

但我离题了——什么对于伪版本来说是重要的？由于伪版本只针对单一用户场景，因此事务语义并不十分重要，而且出于简单性考虑，这些可能根本不必处理。当使用了伪版本时，在编写使用者代码时仍然可以考虑使用事务处理，当然，这样，在用实际基础架构来替换伪版本时，就不必再更改代码。

现在，我听到了资深开发者的担心：“Rollback()发生了什么事情？”

我的观点是，API中的回滚并不重要。如果PersistAll()负责提交或内部回滚，那么当使用者从PersistAll()接管控制权后，所有更改（或者没有更改）已经被保存。（回滚通过异常被通知给使用者。）

有一个例外情况是：在调用`BeginReadTransaction()`之后想要取消操作。可以调用以下方法。

```
void Clean()
```

它将回滚正在执行的事务，并清理操作单元和标识映射。在一次失败的事务之后，使用`Clean()`是一个好主意，因为在`NWorkspace`中将根本不会尝试回滚领域模型实例中的更改。当然，关键在于失败事务的问题是什么，但简单的答案是重新开始。

一些问题可能导致`PersistAll()`内部的重试。例如，在死锁的情况中，`PersistAll()`可能在确定失败之前进行多次重试。这是简化使用方程程序员（consumer programmer）工作的另一个方面，这样程序员可以将注意力集中在对其重要的事情上，即创建好的用户体验，而不是遵循许多许多的规程。

现在，我们已经讨论了很多对于`NWorkspace`来说很重要的功能，但它们对于`NWorkspace`的伪版本并不重要。让我们回到讨论中，关注一下伪版本及其实现。

### 6.3.2 伪版本的实现

这里并不打算详细介绍伪版本实现的细节，而只是从概念上讨论它是如何构建的，主要目的是掌握它的基本思想和使用方式。

伪版本实现使用两个标识映射层。第一个层非常类似于持久框架中的普通标识映射，而且它当前场景已读取的所有实体保持跟踪。第二个标识映射层用于模拟持久引擎，因此在这里，实例不是在场景级别保存的，而是在全局级别保存（即，用于所有场景的同一个标识映射集）。6

因此，当发出对`GetById()`的调用时，如果在标识映射中找到符合请求类型的ID，那么不会有到数据库的往返访问（或者在伪版本的情况下，不会跳转到第二个标识映射层）。另一方面，如果ID不是在第一个标识映射层发现的，那么它就是从第二层获取的，复制到第一层，然后返回给使用者。

`MakePersistent()`非常简单，实例只与第一个标识映射层相关。当调用`PersistAll()`时，第一层中的所有实例均被复制到第二层，简单而清楚。

这描述了基本功能。此外，有些问题也值得讨论一下。例如，我不希望伪版本以任何方式影响领域模型。如果影响了，那么就回到原来的状态，然后为领域模型添加与基础架构有关的抽象，或者更糟，添加与伪版本有关的抽象。

又例如，我不知道哪个（哪些）是类的标识字段。在实际基础架构的情形中，可能通过一些元数据知道这一点。我们可以读取伪版本中的相同元数据来查明哪些是类的标识字段，但这样伪版本就必须知道如何处理（从理论上讲）几个不同的元数据格式，而我肯定不喜欢这样。

我所采取的简单解决方案是假设一个名为`Id`的属性（或字段）。如果领域模型的开发人员使用另一种惯例，那么可以在实例化`FakeWorkspace`时将该惯例描述给伪版本。

再次强调，这些可能是我们现在不想知道的信息，但它让我们了解了一个重要的事实：与`NWorkspace`的基础架构实现相比，在伪版本的实例化阶段我们可能或需要做一些另外的事情。

另外一个例子是，我们可以按以下方式从文件读取或保存到文件：

```
//Some early consumer
IWorkspace ws = new
    WorkspaceFake.FakeWorkspace("c:/temp/x.nworkspace");

//Do stuff...

((WorkspaceFake.FakeWorkspace)ws).
    PersistToFile("c:/temp/x.nworkspace");
```

我们已经讨论了很多关于PI和伪版本机制的内容，这可能使你认为如果现在选择了伪版本机制的话，那么过后必须要使用PI支持的基础架构。事实完全不是这样。实际上，没有PI支持的基础架构并不会增加TDD的使用难度。传统上是这样，但并不是必须如此。

在TDD方法中，伪版本是否对单元测试有很大影响？

### 6.3.3 影响单元测试

不是，肯定并非所有测试都受影响。大多数测试的编写应该尽可能与领域模型中的类隔离开来，不应该存在任何对存储库的调用。例如，测试应该在领域模型类中的所有逻辑的开发期间编写。这些测试根本不受影响。

处理存储库的那些单元测试是受影响的，而且是受到积极影响。可能有人说这更多的是有关集成测试的，但事实并不一定如此。存储库也是单元，因此对它们的测试也是单元测试。

即使对涉及的存储库进行集成测试时，尽早编写测试也是一种好的方法，而且在编写测试（以及存储库）时，要确保在基础架构就绪时能够使用这些测试。

我认为一个合理的目标是让所有测试保持良好状态，从而可同时用于伪版本机制和基础架构的测试。这样，就可以在日常工作中对伪版本机制执行测试（出于执行时的原因），并且每天执行若干次基础架构测试，同时在检入时对自动构建进行测试。

也可以在基础架构就绪前的很长一段时间内工作。当然，必须考虑一些有关持久化的事情，特别是对于第一个DDD项目来说，从开始就以迭代方式工作是很重要的。但是，当积累了更多经验时，推迟持久化的添加将最大限度地缩短开发时间，并得到最清晰的代码。

这也为我们提供了另一种重构节奏的可能性，可以得到更即时的反馈——是否喜欢结果。首先，我们在伪版本中让每件事情都开始工作（这比让整个事情，包括数据库在内，达到正确水平更容易，而且更快），如果结果令我们满意，则继续让每件事情在基础架构中开始工作。我相信最大的胜利是，这将鼓励我们更乐于进行重构（而一般情况下重构是一个难点），特别是当对结果不确定的时候。现在我们就可以更轻松地尝试它们了。

当然，尽力避免代码重复也是非常重要的，它对于单元测试与对于“实际”代码一样重要。（至少重要性相差无几。还有一个重要方面是测试的全面性。）因此，我只想实现这种效果：编写一次测试，然后能够同时对伪版本和实际基础架构执行它们。（注意，这当然只适用于某些测试。大多数测试根本不是针对存储库的，因此在这方面对测试进行划分是很重要的。）

#### 与存储库相关的测试的结构

实现此目标的一种方法是影响存储库的每组测试编写一个基类。然后我使用模板方法模

式，按照自己需要的方式来设置IWorkspace。代码如下，首先看一下基类。

```
[TestFixture]
public abstract class CustomerRepositoryTestsBase
{
    private IWorkspace _ws;
    private CustomerRepository _repository;

    protected abstract IWorkspace _CreateWorkspace();

    [SetUp]
    public void SetUp()
    {
        _ws = _CreateWorkspace();
        _repository = new CustomerRepository(_ws);
    }

    [TearDown]
    public void TearDown()
    {
        _ws.Clean();
    }
}
```

然后是子类，像下面这样。

```
public class CustomerRepositoryTestsFake :
    CustomerRepositoryTestsBase
{
    protected override IWorkspace _CreateWorkspace()
    {
        return new FakeWorkspace("");
    }
}
```

这就是与存储库有关的测试的细节，但测试本身是什么情况呢？我们可以从几种不同的风格中进行选择，但我喜欢的一种风格是尽可能多地在基类中进行定义，同时可以在子类中决定在该时刻是否应该实现某个特定测试。我还希望确保不会忘记在子类中实现测试。为了满足这些需求，我最喜欢的风格就是下面这样（首先是基类中的一个简化测试）。

```
[Test]
public virtual void CanAddCustomer()
{
    Customer c = new Customer();
    c.Name = "Volvo";
    c.Id = 42;
    _repository.Add(c);
    _ws.PersistAll();
    _ws.Clean();

    //Check
    Customer c2 = _repository.GetById(c.Id);
    Assert.AreEqual(c.Name, c2.Name);
}
```

```
//Clean up
_repository.Delete(c2);
_ws.PersistAll();
}
```

注意，当new FakeWork-space("")完成时，标识映射的第二层并未被清理，因为伪版本的第二层标识映射是静态的，因此当伪版本实例被重建时，它不受影响。当然，仅对于数据库来说是如此。仅仅打开新连接并不意味着Customers表被清理了。

因此，伪版本以这种方式工作是非常好的，因为这样，在对伪版本执行测试之后，将需要执行清理操作，正如对实际数据库进行测试之后要做的那样（如果为数据库测试选择了该方法的话）。

当然，IWorkspace必须具有Delete()功能（这一点目前尚未讨论），否则它将不可能执行清理操作。事实上，从简单性来看，Delete()非常有趣，因为它需要自己的一个标识映射（用于操作单元中的伪版本）。已经注册为删除的那些实例将一直保持在那里，直到PersistAll()被调用时才执行永久删除操作。

为了支持这一点，IWorkspace将使用以下方法：

```
void Delete(object o);
```

遗憾的是，它也引入了一个新的问题。被删除对象的关系将发生什么变化？这并不简单。同样，为了确定删除操作应该级联（cascade）到多远——需要更多元数据，这些元数据是有关实际基础架构的，但在这里并没有用处。（目前针对伪版本的使用惯例是不使用级联的。）

下面回到测试的讨论中。在子类中，当进行CanAddCustomer()测试时，通常有三个选择。第一个选择不执行任何操作，在这种情况下，按照基类中定义的那样来运行测试。这可能正是我所希望的。

如果对于特定子类，暂时不支持特定测试的话，那么就应该使用第二个选项。子类中的测试像下面这样。

```
[Test, Ignore("Not supported yet...")]
public override void CanAddCustomer() {}
```

这样，在测试执行期间，将明确地指出只是暂时忽略它。

最后，如果“从未”计划在子类中支持测试，那么可以在子类中这样编写。

```
[Test]
public override void CanAddCustomer()
{
    Console.WriteLine
        ("CanAddCustomer() isn't supported by Fake.");
}
```

即便尽最大努力，仍然可能会忘记测试。必须编写以下代码，跳过Test属性。

```
public override void CanAddCustomer() {}
```

此问题有一种解决方案，但我发现我在这里显示的风格在代码量与“忘记”测试的风险之间有了很好的平衡。

我知道，这不是YAGNI，因为现在除了伪版本实现以外，我们没有任何实现，但可以将这看作快速指示器，它指示了后面将要发生的事情。

---

**说明** 这种风格也可以仅用于每个存储库有多个实现的时候。

---

很多应用程序几乎不可能用这种方式来处理整个系统，特别是在生命周期的后期。对于很多应用程序，它可能只对早期开发测试和早期演示有用，但尽管如此，它的帮助作用也是很大的。如果能够自始至终提供帮助，作用就更大了。

在实际工作中，我基本上（当然，总会有例外情况）将注意力集中在编写针对伪版本实现的测试的核心上。我还编写针对伪版本实现的CRUD测试，但在后者的情况中，也通过继承得到用于数据库的测试。这样，就可以测试映射细节。

也就是说，无论是否使用抽象层这样的思想，迟早都会遇到数据库测试的问题。我请我的朋友Philip Nelson编写了有关数据库测试的一个小节内容。

## 6.4 数据库测试

作者：Philip Nelson

当我们使用DDD和所有自动测试特性时，可能需要运行一个包含数据库访问的测试。当第一次遇到这个问题时，看起来就像根本不存在问题。创建一个测试数据库，指向测试数据库上的应用程序，然后开始测试。

我们来更多地关注一下自动测试，假设用JUnit或NUnit编写了第一个测试。此测试只从数据库加载一个User领域对象，并验证所有属性已正确设置。这个测试可以反复运行，每次都能正常工作。现在，编写测试来验证是否能够更新字段，我们知道数据库中有一个特定对象，因此就更新它。

测试有效，但引入了一个问题。读取测试不再有效，因为我们刚刚更改了底层数据，因此它与第一个测试的预期不再匹配。这没有问题，这些测试应该共享相同的数据，因此我们为更新测试创建一个不同的User。现在，这两个测试是互相独立的，而且下面这一点很重要：在任何时候，我们都必须确保测试之间没有残余效应。数据库支持的测试总是带有一些必须满足的预先条件。在编写测试的过程中，必须考虑这样一个事实：为了满足测试所要求的预先条件，必须对数据库进行重置。下面我们就来讨论此问题的更多细节。

在某个时刻，我们已经为所有基本的User对象的生命周期状态[即创建、读取、更新及删除(CRUD)]编写完测试。在应用程序中，很可能会有数据库支持的一些其他操作。例如，User对象可能帮助实施一种“最大失败登录尝试”策略。当对失败登录尝试和最后一次登录时间字段的更新进行测试时，我们意识到更新实际上未生效。更新测试没有捕获问题，因为数据库已经将字段设置为预期的值。聪明人可能很快能发现问题。但年轻的Joe却苦苦思索了4小时。

他的编码技巧并不存在太大的问题，问题出在他对两件事的理解上：一是代码如何连接到数据库，二是如何将此测试中的数据与所有其他测试中的数据分离开来。他只是没有注意到Last-UpdateDate字段在两次测试之间被重置了。毕竟，代码的设计目的是隐藏数据库的实现细节，不是吗？

此时，我们开始认识到，每个测试都有独立的测试数据，这比我们希望的要复杂得多。我们可能已经清楚地理解（或尚未理解）在测试之间重置数据的重要性，但现在应该理解了。幸运的是，我们的xUnit测试框架正好有我们所需的東西。有一个安装和卸载代码块是用于此目的的。但像编程中的大多数事情一样，有多种方式可以实现此目的，它们各有利弊。我们必须理解折中，并确定最适合自己情况的平衡。

我将可用技术分为4类。

- 在每次测试之前重置数据库。
- 在运行期间保持数据库的状态。
- 在运行之前，仅对正在运行的测试或一组测试重置数据。
- 将单元的测试与数据库调用的测试分离开来。

#### 6.4.1 在每次测试之前重置数据库

乍看起来，这可能是最符合需要的，但可能也是最耗时的选项。此方法的有利方面是，在测试开始时，整个系统就处于已知状态。我们不必担心在测试运行期间会有奇怪的交互，因为它们都是在同一个地方开始的。不利方面就是时间。在项目的最初部分过去之后，我们会发现自己在等待，当测试套件运行时。如果正在执行单元测试，并且在每次更改之后运行测试，那么这很快就会占用很大一部分时间。有一些选项可以提供帮助。这些选项是否适用，取决于很多事情，包括从所使用的架构类型到数据库系统的类型。

一个简单但速度较慢的方法是在每次测试之前从备份恢复数据库。在有很多数据库系统的情况下，这就不是一种实用的方法了，因为它将耗费大量时间。然而，仍然有些系统可以使用这种方法。例如，如果对一个基于文件的数据库进行编程，那么只需关闭连接并复制一个文件来返回初始状态。另一种可能是内存内数据库（in-memory database），例如HSQL for Java，这些数据库可以被非常快地恢复。即使正在使用的是更标准的系统，例如Oracle或SQL Server，如果数据访问的设计足够灵活，那么也能够切换为内存内数据库或基于文件的数据库，以便进行测试。这特别适用于使用O/R映射工具的情况，这里，O/R映射工具负责构建实际的SQL调用，并且兼容多种语言。

项目DbUnit提供了另一种在运行测试之前重置数据库的方式。本质上，它是用于JUnit的框架，允许我们定义“数据集”，这些数据库在每次运行测试之前被加载到干净的表上。Ruby on Rails具有一个类似的系统，允许用YAML（YAML Ain't a Markup Language的标记语言）格式来描述测试类数据，并在测试期间应用它。这些工具可能工作良好，但仍可能遇到问题，因为这些数据库插入可能是一些非常慢的日志操作。另一种可能有效的方法是使用批量加载实用工具，这些批量加载可以不再是大量的日志操作。下面介绍一下如何将这种方法应用于SQL Server。首先，使用Simple Recovery的



测试数据库上的数据库恢复选项。这可能消除大部分日志操作，并提高性能。然后，在测试设计期间，执行以下操作。

- 将数据插入到数据库中，此数据库将仅用作一个干净的测试数据源。
- 调用`transact sql`命令，将测试数据导出到文件。
- 编写用于将数据批量插入到这些文件中的`transact sql`命令。

然后，在设计测试安装方法期间，执行以下操作。

- 截取所有表。这不是日志操作，速度将非常快。
- 发出前面调用的批量复制命令，将测试数据加载到测试数据库中。

此技术的一种变体是安装测试数据库，然后使用常规数据管理技术加载初始测试数据。然后，假设数据库支持这种操作，那么将底层数据文件从服务器分离出来。为这些文件制作一个副本，因为它们将作为干净的测试数据源。然后，编写支持以下功能的代码。

- 将服务器从其数据文件分离出来。
- 复制干净的测试数据，覆盖实际的服务器文件。
- 将数据库连接到副本。

在很多情况下，此操作的速度非常快，而且可以在测试固件中运行，如果数据量不是过大，则可以在测试本身中运行。

一些O/R映射工具支持的另一种变体是从映射信息构建数据库模式。然后就可以使用测试安装中的领域模型来填充每个测试套件或每个测试固件所需的数据。

### 6.4.2 在测试运行期间保持数据库的状态

你可能会想：“这是一种数据管理操作！”事实的确如此。还有另外一种更容易执行的技术。从根本上讲，我们所做的事情是在事务中运行每次测试，然后在测试结束时回滚事务。这略带有一些挑战性，因为事务通常不在公共接口上公开，但这仍取决于我们所使用的架构。对于.NET环境，Roy Osherove提出了一个非常简单的解决方案，即利用ADO.NET对COM+事务征募（transaction enlistment）的支持。利用此工具，我们能够在特定测试方法上添加[Rollback]属性，从而使这些方法可以在它们自己的COM+事务中运行。当测试方法结束时，事务就自动回滚，无需我们编码，也与类的拆卸（tear-down）功能无关。

这种技术的最大优点在于，测试可以完全不必知道底层发生了什么事情。此功能已经被打包到XtUnit[XtUnit]项目中，并作为NUnit测试框架的一个扩展。这是到目前为止保持数据库简捷风格的最简单方法。但这也是有代价的。事务本身需要被记录，这增加了执行时间。COM+事务使用Distributed Transaction Coordinator，而且分布式事务比本地事务慢。这些因素加在一起可能对测试套件的执行速度产生极大影响。

自然，如果正在测试事务语义本身，那么此技术可能会遇到一些另外的问题。因此，取决于项目特点，这可能成为一个真正伟大的解决方案，或者是一个随着测试数量增加需要替换的解决方案。幸好，在测试速度成为问题时，我们不必重写大部分代码，因为解决方案对测试是透明的。我们要么能够容忍速度问题，要么就随着项目的增长不得不采用其他技术。

### 6.4.3 测试之前重置测试所使用的数据

这种方法减轻了在测试调用之前不得不重置整个数据库的负担。相反，我们可以在测试前后向数据库发出一组命令，这在`setup`和/或`teardown`方法中是很典型的。好消息是对单个测试类的处理减少了很多。前面的技术仍然可以使用，但现在我们有了另外的选项，即发出简单的插入、更新或删除命令。即便需要日志，这也减轻了问题，因为影响更小了。再次强调，如果系统支持的话，减少数据安装选项的日志工作仍是一个好想法。

但也有不利方面。一旦假定自己准确地知道哪些数据将受到测试的影响，那么这意味着还要理解另外两件事情，一是正在测试的代码对数据执行了什么操作，二是正在测试的代码调用了哪些额外的代码。这可能在我們不注意的情况下发生改变。测试可能会中断，但这并不是代码被破坏的信号。某个被记录的bug可能根本不是bug，而只是日常的维护工作。

我们还是有可能成功使用这种技术的，但经过多年的测试编写之后，我发现这是最有可能在原因不明情况下发生中断的方法。但是，我可以最大限度地避免这个问题。如果为那些被测试代码调用的类建立模拟实现，该代码将不会影响数据库。这有些类似于单元测试。如果正在进行自动测试，其中要测试的是实际类之间的交互，那么建立模拟实现就没有用处了。无论如何，建立模拟实现操作需要做出一个架构决策，以便容易用测试框架来代替外部类。如果打算这样做，那么很自然就会发现我们正在向着最后一个选项靠近，下面我们就介绍这个选项。

### 6.4.4 不要忘记不断演变的模式

毫无疑问，随着时间的推进我们不断对数据库做出更改。虽然在开始阶段，可能只想直接修改数据库，并按需要来调整代码，但在有了一个已发布的系统后，就应该思考如何让这些更改成为过程的一部分。使用测试系统实际上使这项工作变得更容易。

我的首选方法是创建并运行针对数据库的更改脚本（`alter script`）。在数据库重置后，可以立即运行这些脚本，但由于所有数据库状态均包含在受源代码控制的开发环境中，因此开发脚本并使用它们来修改测试环境可能是有意义的。当测试通过时，检入所有修改，然后令脚本自动设置，以便根据需要对QA环境来运行。

如果这发生在构建服务器上，那么会更好，因为更改脚本的运行通常在被应用于实际系统之前得到测试。当然，这假定QA环境被常规地重置，以匹配实际环境。我相信这个过程肯定有很多变体，但最重要的事情是提前计划，以确保代码和数据库更改的可靠发布。

### 6.4.5 分离单元测试和数据库调用测试

敏捷社区的邮件列表和论坛中讨论最多的事情就是与数据库进行交互的那些代码的测试。在使用（并且仍在定义）TDD技术的人中，如果所有代码都可以像它们被编写时那样作为单元进行独立测试，那么这将具有极高的价值。在TDD中，为类编写测试并且通过直接调用数据库开始的实现是没有意义的。相反，调用被委托给不同的对象，此对象对数据库进行抽象，当编写测试时，用桩或模拟来替代数据库。然后编写实际实现数据库抽象的代码，并测试它是否正确调用底层数据存取对象。最后，我们将编写少量测试，用于测试基础架构是否正确工作（基础架构将实际数

数据库调用连接到代码)。俗话说，编码中的大多数问题都可以用另一个间接层来解决。通常，知易行难。

首先，让我们来澄清几个定义。桩是代码的一个替代片段，这个替代刚好不引起调用者的问题，而且没有更多内容。数据库调用（例如对JDBC Statement的调用）的桩仅包含一个基本的类，它实现Statement的接口，并且在被调用时简单地返回ResultSet，可能会忽略传递的参数，当然不执行数据库调用。

模拟Statement将做所有这些事情，并允许设置我们的期望。稍后将对这一点进行更多讨论。像桩一样，测试将使用模拟命令来替代实际命令，但是当测试完成时，它将“请求”模拟命令“验证”它是否被正确调用。Statement模拟的期望包括：在调用时所需的任何参数的值，executeQuery被调用的次数，要传递给语句的正确SQL，等等。换句话说，我们告诉模拟命令都期望些什么。然后要求它去验证在测试完成后，确实实现了这些期望。

当你考虑它时，一定会认同下面这一点：User类的单元测试应该不必关心数据都执行了什么操作。只要类与数据库抽象层正确交互，就可以假设数据库将正确完成其工作，或者至少对数据存取代码的测试将为我们验证这一点。我们只需验证代码将所有字段正确传递给数据库存取代码即可，这就是需要做的全部工作。如果总是这样简单该有多好！

从逻辑上可以推出这样一个结论：最后代码中将包含为很多类编写的模拟实现。我们不得不用足够的数据来填充这些模拟实现的成员和属性，以满足测试的需要。这会产生大量代码，多出来的大量代码将使代码更难以维护。考虑下面这种替代方案。很多语言中都提供了一些新框架，它们允许从实际类或接口的定义来创建模拟对象。这些动态创建的模拟对象允许我们设置期望、设置期望的返回值，并且可以在无需编写大量代码（这些代码用于模拟类本身）的情况下执行验证操作。有一种著名的技术是Dynamic Mocks，利用它可以简单地将要模拟的类传递给框架，并返回将会实现其接口的对象。

在有关如何模拟代码方面，有很多其他信息源，但有关如何有效模拟数据库存取代码的信息却并不多。数据存取代码需要处理以下不断运动的部分：连接、命令对象、事务、结果以及参数。数据库存取库本身驱动了各种数据存取帮助工具的创建，这些工具的目标是简化此代码。看起来所有这些工具，无论是帮助工具，还是像JDBC或ADO.NET这样的底层数据存取库，在编写时都未考虑过测试。虽然这些工具中很多提供了数据存取上的抽象，但事实证明，要模拟所有这些运动的部分是一项非常棘手的工作。还有另外一个问题就是，对数据存取代码与类的其余部分之间的所有那些字段的映射进行测试。下面就提供一些帮助完成这些测试的建议。

应该在不涉及数据库存取代码的情况下测试每件能够测试的事情。数据存取代码应该最大限度地减少操作，在大多情况下，应该只做CRUD。如果我们能够在不涉及数据库的情况下测试类的所有功能，那么就可以编写只执行这些简单CRUD调用的数据库测试。借助于帮助方法，我们能够利用反射来比较对象的方法验证前后的字段集是否符合预期，而不用对所有属性测试进行手工编码。如果使用O/R映射工具（例如Hibernate），这将特别有帮助，因为这时数据存取是完全隐藏的，但映射文件本身需要进行测试。如果可以在不涉及数据库的情况下对类的所有其他功能进行验证，那么我们只需验证类的CRUD方法和映射正确工作即可。

数据存取代码调用的测试应该与数据库代码本身的测试分开。例如，如果有一个User类，它是通过UserRepository保存的，或者是通过nTier技术中的一个数据存取层保存的，那么要测试的所有事情就是UserRepository被上层的类正确调用。UserRepository的测试将对数据库的CRUD功能进行测试。

为了测试特定的数据存取类型，这些简单的CRUD测试可能不够充分。例如，对那些与类的持久化不直接相关的存储过程的调用就属于这种情况。在某个时刻，可能需要测试这些过程被正确调用，或者在测试中需要从这些调用之一返回的数据，以进行后面的测试。这里是在这些情况下常用的一些技术：实际上是直接模拟JDBC、ADO.NET或其他一些数据存储库。

我们必须使用工厂来创建针对接口（而不是具体类型）的数据存取对象和程序。框架可能会为我们提供工厂，像Microsoft Data Access Application Block在其较新版本中所提供的那样。然而，我们还需要能够使用这些工厂来创建数据存取类的模拟实现，很多工厂框架并不提供这种模拟实现的开箱即用支持。如果我们有个工厂，它可以通过配置测试代码来提供模拟实现，那么就可以代替这些数据存取类的模拟版本。然后，可以验证任何类型的数据库调用。尽管如此，我们可能仍然需要这些类的模拟实现。对于ADO.NET来说，这些功能可以通过一个称为.NET Mock Objects [MockObjects]的项目获取。可能还有一些可用于其他环境的版本。就我所知，SnapDAL框架 [SnapDAL]是唯一一种可以直接使用模拟的基于工厂的框架组合，它是基于.NET Mock Objects构建的，提供了ADO.NET通用类的模拟实现，目的是完全支持ADO.NET通用接口的模拟对象。

是否能够使用这些框架取决于很多因素，但无论如何，应用程序都需要支持工厂，此工厂能够返回实际的数据存取类或数据存取类的模拟。当我们到了可以创建数据存取类的模拟实例的时候，就可以使用以下技术中的一些或全部技术进行测试。

- 模拟返回结果集，结果集可用于代码的桩。结果集以标准的模拟对象样式从测试代码获取其数据。
- 模拟可以从文件（例如XML文件或由业务单元提供的验收测试数据的电子表格）返回测试数据。
- 模拟可以从备用数据库或从正被测试的实际查询精化后的查询返回测试数据。
- 如果可以取代一个数据存取命令，就可以“记录”数据存取，然后从测试中的模拟对象“回放”它，因为我们将拥有对所有参数和返回结果的访问权。
- 模拟可以具有对一些重要事情的验证期望，包括被打开和关闭的连接、提交或回滚的事务、生成和被捕获的异常，以及读取器所读取的数据，等等。

我希望这些思想在测试数据库连接的代码方面可以起到抛砖引玉的作用。这段内容基本上是按照复杂性的顺序编写的，其中数据库重置或回滚技术是最易于实现的。将数据存取代码完全从其余代码中分离出来始终是一项好的实践，并且会提高测试成功率和测试速度。在最细的粒度级别，应允许为实际数据库调用建立模拟版本，这在测试数据的收集、提供和组织方面提供了极大的灵活性。如果使用了遗留代码，而且这些代码使用了你的语言中较低级别的数据存取库，那么就可以推进这种方法。如果开始新应用程序，则可以简单地采用这种思想：在测试之间重置数据库，并在测试速度开始影响效率时改进测试结构。

感谢Phil! 现在我们为处理数据库测试问题做好准备, 无论选择什么方法。

说明 Neeraj Gupta对此的评论是: “你可以使用Oracle数据库的Flashback特性将数据库退回到测试之前的状态。”

我们已经讨论了很多, 但在准备基础架构方面有一个很大的难题尚未讨论。如何解决\_GetNumberOfStoredCustomers?

尚未讨论的问题显然就是查询。

## 6.5 查询

在不同的基础架构解决方案中, 查询是完全不同的。查询对使用者也有很大的影响。在开始时, 查询需求可能并不明显, 但经过一段时间后, 我们通常会发现很多查询需求, 在满足这些需求的同时, 通常也把自己绑到所选择的基础架构上了。

让我们后退一步, 采取另一种不是基于查询的解决方案。前面曾介绍过如何通过存储库的标识来获取结果, 代码如下:

```
//OrderRepository
public Order GetOrder(int orderNumber)
{
    return (Order)_ws.GetById(typeof(Order), orderNumber);
}
```

然而, 如果orderNumber不是标识, 那么存储库方法的接口就必须做出明确的更改, 以返回一个列表, 因为在到达特定状态之前, 多个订单可以有orderNumber 0。但然后呢? GetById()现在是无用的, 因为orderNumber不是标识(而且假设它不是独一无二的, 因为我说过答案可以是列表)。我需要一种方式来到达Order的伪版本的标识映射的第二层。假设可以通过GetAll(Type typeOfResult)来到达第二层, 代码如下:

```
//OrderRepository
public IList GetOrders(int orderNumber)
{
    IList result = new ArrayList();
    IList allOrders = _ws.GetAll(typeof(Order));

    foreach (Order o in allOrders)
    {
        if (o.OrderNumber == orderNumber)
            result.Add(o);
    }

    return result;
}
```

这仍然是不良的代码, 而且当有了基础架构时, 这显然不是我们想要编写的代码, 至少我们不想将它用于实际的应用程序。

### 6.5.1 单组查询对象

就像存储库一样,理想情况是从第一天开始,就在单一实现中“正确地”编写查询,这个单一实现可同时用于伪版本和实际基础架构。那么如何做到这一点呢?

假设要使用查询对象[Fowler PoEAA](将查询作为对象封装起来,提供面向对象的话语来使用查询),而且将签名从GetAll()更改为GetByQuery(),并让它带有IQuery参数(像NWorkspace中定义的那样)。代码如下:

```
//OrderRepository
public IList GetOrders(int orderNumber)
{
    IQuery q = new Query(typeof(Order));
    q.AddCriterion("OrderNumber", orderNumber);
    return _ws.GetByQuery(q);
}
```

这段代码相当直接。我们只需通过指定希望在结果中使用哪种类型来创建一个IQuery实例。然后设定一个标准,以便尽最大可能压缩结果集的大小,通常所采用的方式是在数据库中处理查询(或者当使用伪版本实现时,在伪版本代码中处理查询)。

---

**说明** 查询界面很容易变得流畅,但现在我们暂时停留在可能遇到的最基本情况。

---

这就是需要实例化领域模型一部分时要做的事情。我们回到前面讨论过的\_GetNumberOfStoredCustomers()。当使用新添加的查询工具时,该代码会是什么形式?假设它调用存储库和像下面这样的方法:

```
//CustomerRepository
public int GetNumberOfStoredCustomers()
{
    return _ws.GetByQuery(new Query(typeof(Customer))).Count;
}
```

它可以工作,但对于生产场景来说,任何DBA都无法接受该解决方案。至少当结果是一个SELECT时会是这样,因为SELECT将获取所有的行(只为了能够计算有多少行)。大多数情况下这是无法接受的。

我们需要在查询API中添加更多功能。这里有一个示例,其中我们有可能返回简单的类型(例如int)再加上一个聚合查询(这次,聚合并不是指DDD的聚合模式,而是指SQL中的SUM或AVG查询)。

```
//CustomerRepository
public int GetNumberOfStoredCustomers()
{
    IQuery q = new Query(typeof(Customer),
        new ResultField("CustomerNumber", Aggregate.Count));
    return (int)_ws.GetByQuery(q)[0];
}
```

上段代码略显生硬和不成熟，但我认为从中可以看出一种思想，即如何设计NWorkspace中的基本查询API。

拥有标准查询语言是很一件很好的事情，不是吗？或许正是缺乏这样一种语言才导致对象数据库技术未得到充分利用。当然，现在对象查询语言（OQL），但我认为它出现得有点晚了，而且是一种很难实现的复杂标准。它是胜任的，但太复杂了。（这可能是妨碍对象数据库成为主流技术的原因。）第8章中将讨论更多有关对象数据库的内容。

现在，我需要的是用于持久化框架的查询标准，它需要像SQL一样普通，但面向的对象是领域模型。直到有了这样一个标准后，NWorkspace版本才可以起到桥梁作用，至少对我来说是这样的。但这是否有任何代价？是否会有像免费午餐一样的事情？

### 6.5.2 单组查询对象的代价

转换当然是有代价的，这里也是有代价的。首先，从IWorkspace实现转换到基础架构解决方案将产生性能代价。然而，在端到端场景中，性能代价可能很低。

还有能力损失的代价，因为NWorkspace-API是简化后的，而且适当的基础架构解决方案需要提供更多功能，而这可能是更糟的。然而，另一个代价是，NWorkspace的API本身是很不成熟的，而且不如基础架构解决方案的查询API。所有的代价看起来都是合理的，如果可以得到很多回报，那么这些代价是可以容忍的。

前面省去了有关查询和标识映射的一个很重要的细节：当查询时是否忽略缓存。

#### 查询和缓存

前面曾提到当执行GetById()时，如果该操作必须通过完成持久化来执行，那么所获取的实例将在返回给使用者之前被添加到标识映射。对于GetByQuery()来说也是这种情况，实例将被添加到标识映射。

然而，这当中有一个很大的区别：GetByQuery()在进行持久化之前不会调查标识映射。一部分原因是我们需要使用后端的力量，但最重要的是我们不知道标识映射中（或者缓存中，如果使用缓存的话）是否具有用于查询的所有必要信息。为了查明是否有这些信息，需要访问数据库。这又带来了另外一个问题。GetById()以标识映射开始，而GetByQuery()却不是。这是完全不同的行为，它实际上意味着GetByQuery()绕过了缓存，这会产生问题。如果我们对已经添加到标识映射或操作单元中的新的Customer Volvo（但尚未进行持久化）进行查询，那么通过ID查询时可以找到它，但通过名称查询时则无法找到，这真是一个奇怪的现象。

实际上，这是一个痛苦的决定，但我决定让GetByQuery()在访问数据库之前默认执行一个隐式的PersistAll()调用。（重写GetByQuery()可避免隐式调用，但默认情况下是隐式调用PersistAll()。）我的结论是，这种默认风格可能最符合NWorkspace的其余部分，也符合其简单性和降低出错风险的目标。这就是为什么对事务语义做出一些牺牲的原因。一些人可能会争辩说这违反了“最小惊讶”原则。但我认为在这种情况下，什么会引起惊讶取决于个人的背景。

最大的缺点无疑是当执行GetByQuery()时，可能会产生与保存有关的异常。这是一个很难做出的决定。但现在需要做出一些决定，以便继续前进。

你是否记得简单且未经过优化的 `_GetNumberOfStoredCustomers()` 版本？它不仅速度慢，而且当代码类似于以下形式时，可能不会按照预期工作（优化后的版本也无法工作）。

```
public int GetNumberOfStoredCustomers()
{
    return _ws.GetByQuery(new Query(typeof(Customer))).Count;
}
```

它不工作的原因在于 `GetByQuery()` 将执行那种隐式的 `PersistAll()`。相反，必须像下面这样使用重载，其中 `false` 是由于 `implicitPersistAll` 参数造成的。

```
public int GetNumberOfStoredCustomers()
{
    return _ws.GetByQuery(new Query(typeof(Customer)
        , false)).Count;
}
```

---

**说明** 当然我们可以（也应该）另外使用聚合版本。这里的重点是如何处理隐式的 `PersistAll()`。

所有这些会在一定程度上影响编程模型。首先，在对同一个工作区实例进行查询和更改时，无疑应该尝试采用一种模块化的工作方式。这样，当做出更改时，应在查询之前确保更改的正确性，因为查询将使更改变成持久的。

---

**说明** 你可能是正确的，我促成了一种拙劣的使用方程序员风格，我应该为此负责。他们只管编码，代码能工作即可，即使他们明显忘记了保存等一些功能。

---

一个预想不到的副作用是可能从 `GetByQuery()` 得到完全不同的异常，因为异常实际上可能来自 `PersistAll()`。因此，在使用者代码中调用 `PersistAll()` 无疑是一种好的思想。

如果你不喜欢这种行为，仍然可以使用重载。（实际上，`GetById()` 为我们提供了另外一种重载方式，即重载直接到达数据库，而无需检查标识映射。）“我不关心我自己的暂时工作，我需要知道数据库中有什么。”

以上是查询如何与标识映射关联。下面介绍将查询存放到哪里。

### 6.5.3 将查询定位到哪里

在我看来，至少有以下三个位置可以存放查询实例。

- 在存储库中。
- 在存储库的使用者中。
- 在领域模型中。

下面分别讨论这三个位置。

#### 1. 在存储库中

或许最常见的查询设置位置就是存储库。这样查询就变成了用于处理方法请求（例如 `GetCustomersByName()` 和 `GetUndeliveredOrders()`）的工具。也就是说，使用者可以将参数发送



给方法，但这些参数只是普通类型，而不是查询对象。然后依照方法和可能的参数值来设置查询对象。

## 2. 在存储库的使用者中

在这种情况下，查询是在存储库的使用者中设置的，并作为参数发送到存储库。这通常用于具有很高灵活性的查询中，例如当用户可以在一个大的过滤表中选择填充任意字段的时候。在存储库上的这样一个典型方法就是GetCustomersByFilter()，它带有一个IQuery作为参数。

## 3. 在领域模型中

最后，在领域模型中设置带类型的查询对象（仍然是这些查询实现了NWorkspace的IQuery）是一件有趣的事情。例如，使用者仍将获得强大的查询能力（这些查询将被发送到存储库），而且具有直观且类型安全的API。当然，API的样式完全由领域模型开发人员决定。

使用者不必编写以下无类型的简单查询：

```
//Consumer code
IQuery q = new Query(typeof(Customer));
q.AddCriterion("Name",
"Volvo");
```

而可以用以下代码设置相同的查询：

```
//Consumer code
CustomerQuery q = new CustomerQuery();
q.Name.Eq("Volvo");
```

除了设置更简单的使用者代码以外，这还进一步封装了领域模型。

这也降低了灵活性，而且这是一个优点。不要让每件事情都成为可能。

假设你喜欢将查询存放在领域模型中，那么我们需要哪些查询？所需查询的数量是多少？

## 6.5.4 再次将聚合作为工具

上面最后一个问题可能看起来很宽泛，但实际上我有一个观点。请猜一下？是的，我认为应该使用聚合。

每个聚合都是在领域模型中使用查询对象的典型候选。当然，我们也可以走XP路线，即在第一次需要的时候，再创建它们，这可能更好。

当谈到聚合时，我想再次指出的是我将聚合看作一种默认机制，用于确定loadgraph应该有多大。

---

**说明** loadgraph是指我们应该从目标实例的多远开始加载实例。例如，当请求某个特定的order时，是否也应该加载其使用者？是否应加载其orderLine？

---

当默认的性能级别不够好时，有很大的性能优化空间。一个典型的示例是当需要列出实例（例如Order）时，不加载完整的列表。相反，可以创建一个简化的类型，例如OrderSnapshot，它只包含该列表所需的字段。这些列表也不会与操作单元和标识映射集成，这种情况可能正是我们需要的，仍然是由于性能原因（或者将导致问题，像通常那样）。

抽象层可以支持创建这样的列表，这样代码就不必知道抽象层的实现了。代码可能像下面这样：

```
//For example OrderRepository.GetSnapshots()
IQuery q = new Query(typeof(Order), typeof(OrderSnapshot)
    , new string[] { "Id", "OrderNumber", "Customer.Id"
    , "Customer.Name" });
q.AddCriterion("Year",
    year);
result_ws.GetByQuery(q);
```

为了使这段代码能工作，OrderSnapshot必须具有适当的构造方法，像下面这样（这里假设Id被实现为Guid）：

```
//OrderSnapshot
public OrderSnapshot(Guid id, int orderNumber
    , Guid customerId, string customerName)
```

**说明** 类型过于繁多是O/R映射工具的致命弱点，前面曾提供了一个示例。以我的经验，问题摆在那里，但并不是毫无解决办法。首先，我们只在必要时才执行此操作，因此并不是模型中的所有聚合根都有一个快照类。其次，我们通常可以不将单一快照类用于几种不同的列表场景。这种优化效果一般可达到足够好，因此无需进一步的操作。

另一个常见方法是使用延迟加载来调整，即把一些数据的加载延迟到只在需要时才进行。（下一章将讨论更多有关这方面的内容。）

如果这还不足以满足要求，那么可以手工编写SQL代码，并实例化自己的快照类型。我们需要清楚认识到的是，在这个时候也正在开始使用数据库模型了。

#### 使用一个存储库集合，还是两个

综合所有这些因素，我们现在理解了这样一个事实：在如何构造存储库方面有很多种可能性，但我们可能想知道在真实世界的项目中是如何构造存储库的。

在我的最新的大型项目中（正处在生产环境中，它不是“玩具”项目），我使用一个单组的存储库，同时用于伪版本执行和对数据库的执行。有几个优化使用了本地SQL，但我在那里采用了一点手段，以便当优化后的方法找到一个被注入的连接串时，它调用另外一个完全是我自己编写的方法。

否则，未优化的代码将被使用。此时，伪版本将使用未优化的版本，而与数据库有关的代码通常将使用优化后的版本。

再次强调，这只用在几种情况中。它并不是非常整洁，但目前工作良好。

### 6.5.5 将规格用于查询

另一种查询方法是使用规格模式[Evans DDD]（把用于描述某件事情，例如不允许购买更多

东西的某位客户“是什么样的”的概念规格封装起来)。概念获得了一个描述性的名称,而且可以反复使用。

使用这种方法类似于创建类型安全的、特定于领域模型的查询,但它更进一步,因为它不是我们所使用的通用查询,而是特定查询,它基于特定于领域的概念。这是比领域模型中查询定义(例如CustomerQuery)更高的一个层次。规格并不是公开通用属性(以便根据这些属性进行过滤),而是一个具有非常特定的领域意义的概念。一个常见示例是黄金客户的概念。规格描述了一个客户要成为黄金客户,需要什么条件。

代码非常清晰,而且目的明确,因为抓住了概念,而且概念被描述为规格类。

这些规格类可以很好地发出IQuery命令,这样,在想要使用抽象层的时候,就可以通过IWorkspace(或等效的工作区)的GetByQuery()来使用它们。

### 6.5.6 其他查询选择

到目前为止,我们已经讨论了将查询对象用作查询语言(尽管有时是被嵌入到其他结构中)。事实上,当查询变得复杂时,更有力的做法是能够在一种类似于SQL的基于字符串的语言中编写查询。

我并没有考虑过对NWorkspace应用任何更多查询语言。但或许某天我可以说服一些人为NWorkspace查询添加支持,以便支持他们的查询输出。这样,他们的查询(例如类似于SQL的一些查询,但查询对象是领域模型)就可以用于那些具有NWorkspace的适配器实现的基础架构解决方案。

我一直认为,使用哪种查询语言既是一种编程语言选择,也是一种生命周期选择。当然,选择哪种查询语言(如果能选择的话)可能更多地取决于具体情况。这个主题的讨论可以告一段落了(第8章和第9章将讨论更多内容)。现在该到继续下一主题的时候了。

## 6.6 小结

我们已经讨论了持久化透明(PI)是什么,以及如何保持基础架构位于领域模型类外部的一些思想,同时还讨论了如何准备基础架构。我们还讨论了数据库测试,并还对引入抽象层的思想给予了很多关注。

在下一章中,我们将把焦点再次切换到领域模型的核心上,这次我们将关注规则。你会发现第4章中设置的列表中的很多需求都是有关规则的,因此当讨论领域模型时,这是关系密切的主题。

在上一章中，我们做了必要的停顿，讨论了一些基础架构准备工作。现在我们将返回到核心模型的讨论上。本章将讨论规则。

规则是一个庞大的话题。这里只讨论它的一部分，主要关注验证规则。在这个过程中，我们将讨论第4章中定义的需求，并在适当的地方详加讨论。

回头看第4章中列出的需求(稍后将重复此列表)，可以清楚地看到大部分需求都与规则有关，这是领域模型真正的优势所在。在无需规则引擎的情况下，我们也可以走得很远。Evans认为[Evans DDD]，同时使用几种范型是有问题的，因此如果可以避免领域模型与规则引擎的混用，那么将大有帮助。(举例来说明混合范型有多复杂，考虑一下为什么OO与关系模型的混合如此难处理。)

在我的前一本书中[Nilsson NED]，有一章名为“业务规则”，其中提供了大量有关将规则评估定位到哪儿的示例问题。由于某些原因，几乎所有示例最后都是在数据库中检查的。然而，这是由于我所处的数据库时代造成的，现在，我是一位领域模型人员了，因此你会发现本章是完全不同的。这样是否更好？我认为肯定是的，而且希望你也赞同这一观点。

那么，我们应该从哪里开始呢？从规则的分类开始如何？

## 7.1 规则的分类

如果尝试将规则分类，很快会发现有很多种尺度可以考虑。例如，有如下几种属性。

- 在哪里执行（在哪个层中）。
- 何时执行。
- 复杂度（简单约束、跨越几个字段、跨越几个实例，等等）。
- 是否开始生效。
- 等等。

在这方面不打算做过多讨论，而只是推荐几本书（例如von Halle的书[von Halle BRA]、Ross的书[Ross BRB]以及Halpin的书[Halpin IMRD]），对此我感到抱歉。你将发现这些书在很大程度上是以数据为中心的，但不管怎样它们都很有趣。

让我们从另一个角度开始。先定义一些原则，然后应用它们。

## 7.2 规则的原则及用法

我认为原则这个词非常好，因为它并不严格。原则比分类要简单得多，这可能是由于其固有的模糊性。但原则仍然描述了目的。这正是我现在所需要的。

### 7.2.1 双向规则检查：可选的（可能的）主动检查，必需的（和自动的）被动检查

我们应该能够预先（主动地）检查是否存在问题（例如，尝试保存当前更改），以便采取纠正措施。这样，编程模型将更容易。

如果没有预先检查，就有可能发生错误，但可以通过异常（被动地）知道有了错误。

### 7.2.2 所有状态（即使是错误状态）都应该是可保存的

用户讨厌那些由于错误而无法保存当前更改的应用程序。我的一位朋友将它比作由于拼写错误而不允许保存字处理文档。

“所有”是一个很大的词，但我们至少将尝试接近它。应尽力避免出现那些无法保存的状态。

这里，还有一件重要的事情是考虑错误意味着什么。如果一个订单过大的话，那么它有可能被认为是错误的。但直到用户尝试提交订单之前，它并不是一个真正的错误，而只是需要在提交之前进行处理的一种情况。而且，用户应该能够在提交订单之前保存它，以便过后可以继续在其上工作，以解决问题，然后提交。

### 7.2.3 规则应该高效使用

规则的使用应该具有很高的效率，使用规则和定义规则时都应如此。

如何才能提高规则的使用效率呢？方法是令元数据（用于设置UI）的获取成为可能。当进行检查时，这将减少发现问题时间，因为UI起到帮助作用，使得一些问题不会发生。（注意，在规则检查方面，UI并不是所需的全部。）

实现高效定义规则的方法是，通过一个小的框架将注意力集中在感兴趣的规则部分上，并且它将为我们的处理样板代码。

### 7.2.4 规则应该是可配置的，以便添加自定义规则

一些规则与场景有关，而有些规则与场景无关，因此有两点很重要：不仅需要能够在类定义中直接声明所有规则，而且还可能需要为特定用例动态添加自定义规则，这些自定义规则在部署期间作为配置信息。

我认为当构建供很多不同客户使用的产品时，配置方面特别有用。如果正在构建的系统仅供一家客户使用，那么它的价值就很小了。

### 7.2.5 规则应与状态放在一起

规则的定义至少应该尽可能靠近相关的领域模型类。这并不是指规则本身的实现，而是指声明。我们需要对使用进行封装，以便通过检查其类就可以很容易获得领域模型的“完整”视图。

我认为这条原则对另一条原则起到帮助作用，即“规则应该是一致的”。当代码库达到一定规模时，就有了一种风险：规则可能互相影响，甚至矛盾。将规则连同状态放在一起可以在一定程度上帮助管理规则库。

### 7.2.6 规则应该具有很高的可测试性

我们将花费很多时间来使用和测试规则。出于这个原因，对于规则有一个测试友好的设计是很重要的。

在实际中，添加规则也会产生测试问题。我们需要将实例设置为正确的状态。可以编写测试帮助程序（例如测试固件中的私有方法）来处理此问题，但它仍然是一个障碍。

### 7.2.7 系统应阻止我们进入错的状态

简化规则使用的一个好方法是尽可能避免对它们的需要。例如，实现此目的的一种方法是使用那些将领域对象设置为有效状态（此状态永远不会变成无效的）的创建方法。

## 7.3 开始创建 API

本章稍后将回来讨论这些原则并评估如何实施它们，现在回到现实问题中，讨论一下第4章中描述的一些问题。我们从外部API开始，并看一下它可以归结到何处。我相信这将为我们提供有关整个系统的一个很好的、实用的认识。

回顾一下，第4章中介绍的问题或请求如下几点。

- (1) 应用灵活但复杂的过滤器来列出客户。
- (2) 在查看特定客户时列出订单。
- (3) 一份订单可具有多个不同的行。
- (4) 并发冲突检测非常重要。
- (5) 客户对我们的应付款项不能超过一定的额度。
- (6) 订单的总价不能超过100万SEK。
- (7) 每份订单和每位客户都应有一个唯一的、用户友好的编号。
- (8) 在一位新客户被视为可接受之前，必须联系信用卡机构检查客户的信用。
- (9) 一份订单必须有一位客户，一个订单行必须对应于一份订单。
- (10) 保存订单后，其订单行必须是原子的。
- (11) 订单具有可由用户更改的接受状态。

取一个问题，比方说问题6“订单的总价不能超过100万SEK”。

应该如何处理这个问题？可以有几个选择。

可以创建类似于规则检查引擎这样的工具，可以将要验证的实例发送到这里，或者可以让实例负责自己的验证。我喜欢让实例尽可能多地（或者至少在适当的时候）负责自己的验证，因此我倾向于后者。或许这有些不成熟，但也是一种简单的积极方式。

如果让我编写一个测试来开始验证，那么它可能像下面这样：

```
[Test]
public void CantExceedMaxAmountForOrder()
{
    Order o = new Order(new Customer());

    OrderLine ol = new OrderLine(new Product());
    ol.NumberOfUnits = 2000000;
    ol.Price = 1;
    o.AddOrderLine(ol);

    Assert.IsFalse(o.IsValid);
}
```

我们所做的事就是为名为IsValid的Order添加了一个属性。这是一种相当常见的方法。它也存在一些明显的问题，例如：

- 问题是什么？我们只知道存在一个问题，而不知道它是什么。
- 我们允许了一个不正确的转换。
- 如果我们忘记检查了怎么办？

我们所做的所有事情就是检查是否每件事都正确。稍后会回到所提及的问题上，但首先需要讨论一些更基本的事情。问题6中的一个更重要的问题是：

在什么条件下，订单是有的（或者是无效的）？

### 7.3.1 上下文，上下文，还是上下文

一如既往，上下文是非常重要的，但在开始的方法中很容易被忘记。我们需要在继续前进之前解决此问题，因为它对于余下的讨论具有至关重要的意义。

我认为使用常见方法IsValid的原因是过于关注持久化了，这就像某种通用的、捕获一切错误（catch-all）的测试一样。如果我们略微放松规则与持久化的耦合，并尝试遵守这个原则：让所有状态都成为可保存的，那么可能会得到另一种方法。

因而，我认为转换将是一件与规则有关的有趣事情。例如，当一个order处于NewOrder状态时，或多或少每件事情都是允许的。但当order转换为Ordered状态时，只有当满足特定规则时，那些事情才是允许的。

因此，保存处于NewOrder状态的order应该是允许的，即使该订单进入Ordered状态是无效的。这种处理规则的方式将关注模型意义上的规则，而不让技术性（例如是否持久化）影响模型。

也就是说，现实情况是充满了实际问题，我们需要很好地处理它们。因此，即使我们有意将注意力放在模型上，也必须处理好基础架构。让我们看一下考虑将是否持久化作为一种特殊状态是否有帮助，但首先重要的是讨论这样一个问题：如果我们没有足够地考虑基础架构，那么什么将导致有关规则的问题。

### 7.3.2 数据库约束

尽管我们可能选择将所有规则都放在领域模型中，但最后仍然可能有一些规则在数据库中。一个典型的原因是效率。一些规则在数据库中进行检查时，具有最高的效率。这就是将它

们放到数据库中的原因。我们在设计时或许可以避免大部分这样的情况，但总有一些无法避免的情况。

另一个原因是我们可能无法设计一个新数据库，或者无法完全按自己的意愿设计一个数据库，而是不得不调整为当前的数据库设计。然而，另一个原因是，能够管理自己的数据库通常被认为是一件好事，特别是当几个不同系统使用同一个数据库的时候。

第1章中讨论过面向对象与关系数据库之间的阻抗失配问题，这里将再次遇到这个问题。例如，数据库中的大部分字符串都将有静态最大长度，例如`IVARCHAR(50)`。字符串不允许超过这个长度是一个通用规则，而此规则主要与持久化有关。如果字符串过长，将不允许转换为Persisted状态。

因此，我们可能必须处理这样一些规则：直到状态被保存到数据库之前，不会发现这些规则的有效性。这进而又意味着一系列问题。

首先，解析错误信息很麻烦，特别是当我们尝试让一个典型的O/R映射解决方案在不同数据库之间具有可移植性的时候。换言之，问题的核心在于，O/R映射工具在遇到错误时可能无法进行映射，并通知我们哪些字段在领域模型中引起问题，例如，当有一个重复键错误的时候。

此外，发生错误的时候可能不方便进行处理。领域模型已经发生更改，要想从问题恢复可能非常难，除非重新开始。事实上，一般的规则是应该重新开始，而不是尝试去做一些补救。

我们仍然希望在任意时刻都能够保存实例。（在任何情况下，我们都已经设置了一些首选项，即使无法完全遵从它们。追求完美也意味着代价将极为高昂。）

在我看来，应尽最大努力进行主动设计，或适当进行主动设计，以便最大限度地减少来自数据库的异常。对于那些无法避免的异常，就准备好在代码中逐个地处理它们，或者准备回滚事务，丢弃对领域模型的更改，并重新开始。这可能有些过激了，但我的观点就是这样。所有这些的关键在于设计，因此要尽最大可能减少问题。

这就是纯粹从技术方面来看待这个问题（数据库可能会为我们带来麻烦）。但事实上规则并非只与基础架构有关，也不是只与领域有关。它们之间有联系。

### 7.3.3 将规则绑定到与领域有关的转换，还是绑定到与基础架构有关的转换

前面一直强调的是应尽力避免进入一个无效的状态。如果我们遵从此规则，就总可以实现持久化（如果暂时不考虑与基础架构有关的约束）。

该原则属于理想情况。在实际中，在多步骤的操作中，很难不破坏某些原则。例如，是否应该禁止将某个客户名称更改为规则认为过短的名称？或许这是应该禁止的，但它可能会阻止用户执行他想要的操作：删除原有名称，并插入一个新名称。

我们可以通过一个名为`Rename(string newName)`的方法来处理这个问题。在该方法执行期间，对象的状态是不正确的，但我们只考虑方法执行之前和执行之后的问题。

---

**说明** 根据*Design by Contract*[Meyer OOSC]，这与不变量是一致的。这样的不变量类似于断言，实例在方法执行之前和之后应该始终遵守这些断言。唯一的例外是在方法执行期间。

---



另一种方法是在完成更改之前不设置属性。让我们来看一下是否可以遇到另外一个示例。假设有两个字段，这两个字段要么都设置，要么都不设置。同样，有一个方法可以解决此问题，但如果此问题升级，或者UI将用setter方法来公开属性，那么显然有时很难保持它们的一致状态。

另一个问题是，我们是否应该确信不会达到一个无效的状态？如果我们确实达到了一个无效的状态（真正的无效状态，例如当order的状态为Ordered时，进行了一次错误的更改），那么是否应该允许保存该状态？

数据库可能没有针对某些错误情况的保护机制。但即使尽力让数据库在规则方面进行自我管理，仍然可能会漏掉一些方面，而且如果我们过渡到以领域模型为焦点的方式来构建应用程序，那么数据库在自我保护方面可能就不太严格了。毕竟，以领域为焦点的规则在某种程度上是高级规则，在领域模型中表达这些规则要比在关系数据库中表达它们简单得多，并且我们也希望将它们放在领域模型中。

问题是：我们是否可以信任“环境”？也就是说，是否可以确信不会对一个不可能到达的状态进行持久化？例如，是否可以期望没有任何bug？是否可以期望使用者没有创造力，因而不会使用反射将一些字段设置为其需要的值？

### 7.3.4 精化原则：所有状态，即使是错误状态，都应该是可保存的

让我们看一下是否可以构造一个有关如何思考下面这条原则的一个示例：所有状态，即使是错误状态，都应该是可保存的。

再次假设订单可以有两种状态：NewOrder和Ordered。我们只有一条与领域有关的转换规则：订单不能过大，以便进入Ordered状态。这为我们提供了有效的组合，如表7-1所示。

表7-1 订单状态与规则结果的有效组合

订单状态	规则结果
NewOrder	没有过大
NewOrder	过大
Ordered	没有过大

到目前为止，这看起来很简单，而且正是我们所需要的。让我们添加持久化维度，如表7-2所示。

表7-2 订单状态与规则结果的有效组合

订单状态	规则结果	持久化与否
NewOrder	没有过大	未持久化
NewOrder	没有过大	持久化
NewOrder	过大	未持久化
NewOrder	过大	持久化
Ordered	没有过大	未持久化
Ordered	没有过大	持久化

首先要指出的是，我们希望对Ordered状态的转换与“过大”这种组合是不可能的。

更重要的是，如果由于某种原因这种组合确实发生了，那么它就构成了这样一个示例：这是一种发生真实异常错误的状态，而且它应该是不可保存的。

**说明** 在是否于持久化时也检查与领域有关的转换规则方面，你的做法可能不同。在很多情况下，这可能被认为是一种略显极端的做法。

注意，遵守精化后的“所有状态都应该是可保存的”原则并不意味着后退到本章开始提到的IsValid思想。它现在是IsValidRegardingPersistence，而且很多“错误”现在也可以持久化。

## 7.4 与持久化有关的基本的规则 API 的需求

既然我们已经决定了一个特定的上下文，即向持久化的状态转换，那么就让我们开始讨论与之相关的API吧。然后，我们将回到其他（并且从更有趣的模型角度）状态转换问题上来，既有一般问题，也有特定问题。

**说明** 像通常那样，与API本身的讨论相比，这里的讨论实际上更多地关注一些通用思想。我并不是试图告诉读者应该如何创建接口或API，而只是试图给出我对这个主题的一些思想，并希望启发读者找到自己的解决方案。

我要举一个新的代码示例，在这个示例中应用这样一种思想：通过主动设计来避免对那些异常状态错误和/或将从持久化引擎抛出异常的问题进行持久化。

我们将讨论后者：显示一种持久化引擎将抛出异常的情况。（前者的问题更难遇到。如果每件事情都按预期工作，该状态就不应该出现。）

假设Order有Note，而Note在数据库中被定义为VARCHAR(30)。取决于要持久化的解决方案，当尝试保存一个长字符串时，可能会得到一个异常，或者是不知不觉丢失了信息。这里的要点是主动捕获问题。代码如下：

```
[Test]
public void CantExceedStringLengthWhenPersisting()
{
    Order o = new Order(new Customer());
    o.Note = "0123456789012345678901234567890";

    Assert.IsFalse(o.IsValidRegardingPersistence);
}
```

因此，我不会询问订单是否为IsValid，而是将属性名更改为IsValidRegardingPersistence，以便明确下面这一点：订单在被持久化的上下文中是无效的。

我们设置一个简单的接口，目的是让使用者可以询问一个聚合根它是否处于一个可以被持久化的有效状态。代码可能像下面这样：

```
public interface IValidatableRegardingPersistence
{
    bool IsValidRegardingPersistence {get;}
}
```

因此，我们让那些具有持久化相关规则的类型实现该接口。通常，使用者只需询问聚合[Evans DDD]根，它是否处于一个可以被持久化的有效状态，并且聚合根将检查其子。

#### 是否在外部分处理与持久化有关的规则

正如我们在代码片段中看到的那样，我选择让领域模型类本身负责检查规则。考虑我们喜欢的持久化处理方式（从“外部”），这可能感觉有些奇怪。

这可能是一个值得思考的问题，但现在我坚持我的第一种想法，部分原因是我认为规则是领域模型的核心。规则并不像持久化那样“仅仅”是一个基础架构方面。

下面我们回到原来的轨道上来。

### 7.4.1 回到已发现的 API 问题上

本章前面发现了一些与API有关的问题。我推迟了对这些问题的讨论，因为我想首先解决缺乏上下文的问题。在解决这个问题之后，让我们回到这些问题上来，看一下它们是否仍适用于前面的代码片段（`CantExceedStringLengthWhenPersisting()`）。问题是：

- 问题是什么？我们只知道存在一个问题，而不知道它是什么。
- 我们允许了一个不正确的转换。
- 如果我们忘记检查了怎么办？

让我们在与持久化有关的规则的上下文中讨论这三个问题。

### 7.4.2 问题是什么

第一个问题仍然适用。如果实例是无效的，那么很有必要找到究竟破坏了哪些规则。聚合根将从其本身和所有子中收集被破坏的规则，并以`IList`的形式返回被破坏的规则。

让我们为接口添加代码以处理这个问题。代码如下：

```
public interface IValidatableRegardingPersistence
{
    bool IsValidRegardingPersistence {get;}
    IList BrokenRulesRegardingPersistence {get;}
}
```

**说明** 在我看来，被破坏的规则列表是通知模式[Fowler PoEAA2]的一个实现。

我们曾经提到的有关该模型的一个重要方面是，我们需要被通知的消息不仅是错误，还有警告和信息。另一个立即能想到的事情是被破坏规则的列表主要关注聚合，但我们当然可以将这样的几个列表聚合为（这并不是故意制造出来的双关语）应用层[Evans DDD]中的单一列表，例如这几个列表来自几个聚合根实例。

### 7.4.3 我们允许了不正确的转换

当然, Note变得过长了, 而且这可能被认为是一种向无效状态的转换。但我并不认为这会引起非常大的兴趣。我可能会让order与一个过长的Note共存一段时间, 以便让用户可以删除Note的几个字母。

遗憾的是, 在这种情况下, 过长的Note将不可能被保存。另一方面, 这里在非持久化和持久化之间的转换很有趣。但它是不被允许的。(正如我所说, 这将得到一个异常, 或者字符串被默默地截短了。)

向持久化的转换有一些“特殊”。有关允许不正确转换这个问题, 我们将在接下来与领域有关的规则的上下文中进行讨论。

### 7.4.4 如果忘记检查怎么办

如果使用者没有遵守这条规程: 在尝试转换之前首先检查转换可能性, 那么我们就有了一个异常情况, 而且这个异常情况将抛出异常。同样的情况也适用于这里。因此, 如果我们尝试向持久化转换, 就可能会抛出一个异常, 此异常是ApplicationException的子类, 且保持一个被破坏的规则列表IList。

使用者再次未遵守规程, 从而产生了一种异常情况。

我们遇到了一个特殊的转换问题: 向持久化转换。现在应该转移注意力了, 离开与基础架构有关的转换, 暂时转向与领域有关的转换, 但我们将保留一些思想, 例如如果客户未遵守规程, 那么这这将是一个异常。

## 7.5 关注与领域有关的规则

让我们逐个来处理第4章中剩余的特性需求。我们从第6个特性需求开始, 即“订单的总价不能超过100万SEK”, 但是当我们意识到草拟的API缺少上下文时, 我们被打断了。我们来看一下是否可以测试进行一些转换。转换前的代码像下面这样:

```
[Test]
public void CantExceedMaxAmountForOrder()
{
    Order o = new Order(new Customer());

    OrderLine ol;

    ol = new OrderLine(new Product());
    ol.NumberOfUnits = 2000000;
    ol.Price = 1;
    o.AddOrderLine(ol);

    Assert.IsFalse(o.IsValid);
}
```

我们现在知道, 这段代码存在几个问题。为了遵照到目前为止所讨论的思想, 可以为Ordered

状态添加一个转换方法，例如，此方法的名称可以是`OrderNow()`。但我对此仍然不满意，因为`OrderNow()`很有可能在有机会检查之前就抛出了一个异常。

我仍然暂时不考虑什么被持久化，什么不被持久化。这里的有趣部分是，当我们尝试向`Ordered`状态转换时，将无法完成转换。我们对测试进行一些修改来反映出这一点。

```
[Test]
public void CantGoToOrderedStateWithExceededMaxAmount()
{
    Order o = new Order(new Customer());

    OrderLine ol = new OrderLine(new Product());
    ol.NumberOfUnits = 2000000;
    ol.Price = 1;
    o.AddOrderLine(ol);

    try
    {
        o.OrderNow();
        Assert.Fail();
    }
    catch (ApplicationException ex) {}
}
```

要考虑的一件重要事情是，当处于`Ordered`状态时，是否应该检查`AddOrderLine()`，以便不破坏规则。如果允许使用`AddOrderLine()`，那么可以让`AddOrderLine()`将订单状态转换回`NewOrder`。这样就避免了问题。

另一个问题是当处于`Ordered`状态时，如果`OrderLine`中的一个被更改了，那么这将破坏规则。这种情况下的一个典型解决方案是使用不可变的`OrderLine`，这样它们就不会被更改。我们修改测试来反映这一点。

```
[Test]
public void CantGoToOrderedStateWithExceededMaxAmount()
{
    Order o = new Order(new Customer());

    o.AddOrderLine(new OrderLine(new Product(), 2000000, 1));

    try
    {
        o.OrderNow();
        Assert.Fail();
    }
    catch (ApplicationException ex) {}
}
```

**说明** 我可以使用NUnit的`ExpectedException`属性来替代`try catch`和`Assert.Fail()`结构。这会使测试更短。这种结构的唯一优点是，可以检查在调用`OrderNow()`时是否发生预期的异常，这实际上通常是一个小的优点。

当添加一个OrderLine时,无法再对它进行更改,而且对于这个简单示例来说,现在只需在OrderNow()和AddOrderLine()方法中检查此规则。

我们接着看下一条规则。

### 7.5.1 需要合作的规则

特性5是:客户对我们的应付款项不能超过一定的额度。这条规则乍听上去非常简单,但问题在细节之中。让我们开始,并看一下得到什么。最初的测试可能像下面这样:

```
[Test]
public void CanHaveCustomerDependentMaxDebt()
{
    Customer c = new Customer();
    c.MaxAmountOfDebt = 10;

    Order o = new Order(c);
    o.AddOrderLine(new OrderLine(new Product(), 11, 1));

    try
    {
        o.OrderNow();
        Assert.Fail();
    }
    catch (ApplicationException ex) {}
}
```

然而,这可能根本无法工作。为什么呢?订单需要能够找到当前客户的处于特定状态下的其他订单,通常解决此问题的是OrderRepository:

---

**说明** 回忆一下,我选择寻找Order与Customer类之间的单向关系。这就是为什么我通过OrderRepository来帮助(作为模拟双向性的可能策略之一)从Customer定位到其Order。

---

我们可以将\_orderRepository插入到构造方法中的订单来处理这一点,插入方法如下:

```
Order o = new Order(c, _orderRepository);
```

但这是否是一个好的解决方案?通常,这取决于特定的应用程序。当到了调用OrderNow()的时候,如果让Order来检查Customer的短期债务(current debt),那么可能对用户的可用性来说有点太晚了。可能更好的解决方案是在创建订单之前,先查询一下新订单的最大数量。然后,用户创建新订单,而且当达到限制时,用户可以从UI中得到反馈。这并不改变我们刚刚看到的API,但却方便多了。

另外一种处理方法是让Customer负责调用IsOrderOK(orderToBeChecked)。在某种程度上,让Customer负责决定这一点是很自然的,因为Customer拥有做出判断所需的数据。如果我们考虑领域,并尝试在领域中编写代码,那么这是否像是我们将检查订单的责任交给(或应该交给)客户了?可能确实是这样的,如果我们略微不同地描述此问题的话,比方说应该要求客户接

受订单。事实上，这非常类似于前面编写的代码（除了将OrderNow()定位到Order上以外）。这可能是某些事情的迹象。

在以上代码片段所显示的解决方案中，另一个问题是\_orderRepository被注入到order实例中。如果从持久化状态重建一个订单，并且通过使用O/R映射工具来进行重建，那么我们就不再负责order实例的创建了。尽管可以在重建期间将\_orderRepository注入到存储库中，但它感觉有点生硬（例如，当读取一个订单列表，然后将存储库注入列表的每个实例中的时候）。考虑了这一点之后，我们尝试让订单负责检查它自己，但通过一个setter方法来提供存储库，或者在检查时，将存储库作为一个参数传递给OrderNow()方法（但该策略可能无法很好地扩展）。

---

**说明** 我认为这里的讨论指出了至少存在于某些OR映射工具中的弱点：我们没有在重建期间尽可能多地控制实例创建，特别是考虑到流行度日益提高的依赖注入框架（我们将在第10章中讨论它）的时候。

---

## 7.5.2 使用基于集合的处理方法

无论选择前面讨论的哪种风格，都应花几分钟时间来思考一下如何检查短期债务。当然，也可以编写像下面这样的代码（假设使用订单，而不是发票来确定短期债务，这可能有点繁琐）：

```
//Order, code for checking current debt for OrderNow()
decimal orderSum = 0;
IList orders = _orderRepository.GetOrders(c);

foreach (Order o in orders)
{
    if (_HasInterestingStateRegardingDebt(o))
        orderSum += o.TotalValue;
}
```

我认为此代码是一个禁忌！更好的解决方案是通过一个名为CurrentDebtForCustomer()的方法在OrderRepository上公开一个方法，该方法带有一个Customer作为参数。

---

**说明** 在设计存储库实例时，提供重载通常是一个好的思想，例如X(Customer)和X(Guid)。

仅为了能够调用方法（如果保持Customer的ID）而必须实例化Customer是一种很讨厌的限制。

---

我知道数据库人员现在会这样想：“实时一致性是什么情况？存储库方法在执行时，是否具有显式事务中的可序列化的事务隔离级别（至少在保存期间）？”这是可能的，但存在问题。

一个问题是，我们正在使用的O/R映射工具（假设目前正在使用一个O/R映射工具，后面章节将讨论更多有关内容）可能对那些与验证、锁定等有关的保存过程没有详细控制。另一个问题是保存过程的性质意味着这样一个事实：如果有很多脏实例要保存，可能需要一些时间，因此客

户的所有订单的锁定时间可能会相当长。

在这种情况下，通常不必要求实时一致性，只需非常接近实时一致性即可。也就是说，在保存此订单之前，检查其他订单的和，但不为读取操作设置较高的隔离级别，甚至在同一个事务中不进行检查。

这与Evans对聚合[Evans DDD]的看法非常吻合。他的看法是争取在聚合内部保持实时一致性，而不是在聚合之间保持实时一致性。聚合之间的一致性问题可以稍后再处理。

因此，如果我们发现此方法是可接受的（这里我们谈论的仍然是较低的风险，至少对于典型系统是这样），但希望能检测到问题，那么可以创建某种批处理程序，它不时地执行，并检查这样的问题，再发回报告。或者可以在保存时在队列上添加一个请求，让它来检查有关特殊客户的此类问题。它是一个不太可能出现的问题，为什么要让任何用户都必须同步地等待它被检查呢？

我认为此方法是原子信号，指示我们应该使用一个服务[Evans DDD]来替代，以便让整个事情更明显。此服务可以检查短期债务，因此Order可以在适当的时候主动使用它。在保存时，也可以很好地被动使用它，而且可以在队列上添加某种请求，并且可以在保存后被动地使用它。此服务可以像我们刚刚讨论的那样使用相同的存储库方法，但调用者当然无需知道这一切。

在保存之后，应该被动地做什么？或者需要将订单移动到两种状态之一，以便当用户存储订单时，它可以进入NewOrder状态。然后，该服务开始执行，并将订单更改为Accepted或Denied状态，如图7-1所示。

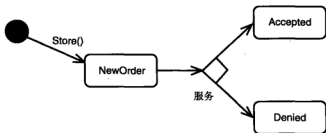


图7-1 状态图

在讨论完所有这些事情之后，我认为我们有三种适当的问题解决方案：一种具有紧密的耦合（每件事情都是实时完成的，可能具有很高的事务隔离级别），另一种具有较小的耦合度（但可能略微复杂，因为有更多的“运动部分”），以及最后一种解决方案，其中除了测试中所显示的那些事情以外，我们不做任何多余的事情（但它具有一个小的不一致性风险，将会导致延迟）。是否有种解决方案适合你？

刚才讨论了利用一个服务来帮助摆脱代价高昂的处理问题。领域模型中的服务也通常来源于另一项需要。

### 7.5.3 基于服务的验证

完全出于巧合，我们的确需要这样的一条规则，它从本质上来说是基于服务的。我们来考虑



特性8：在一位新客户被视为可接受之前，必须联系信用卡机构检查客户的信用。

我们可以在`Grant()`方法中很好地调用这种信贷机构服务，或者可以使用一种类似于刚刚讨论过的那种解决方案，让我们的服务在保存之后开始运行，既管理对外部服务的调用，也负责将客户移动到两种状态之一。这种方法的一个问题是，我过多地将领域规则绑定到持久化操作了。

这种将部分功能移到当前操作单元或标识映射的沙箱之外的方法还存在另外一个较大的问题，即本地缓存中的内容将无法与数据库保持同步，因为处理工作是在自主服务中进行的。当然，这是我们在缓存方面一直存在的风险（如果缓存可以被绕过的话）。

如果我们知道由于实时调用显式服务将使缓存失效，那么就很容易处理了。只需终止缓存即可。但是，要想重建它可能是非常昂贵的，或者我们准确地知道需要刷新什么。另一方面，如果不知道令缓存失效的操作何时发生，那么就不得不加以额外的注意，并尽可能经常刷新缓存。我认为，所有这些都暗示着对缓存的管理要严格，而且不要让缓存过大或存活得太久，至少对于那些更改相当频繁的数据是这样的。

---

**说明** 当这里谈论缓存时，主要考虑的是标识映射[Fowler PoEAA]。

---

让我们来关注更多有关转换的话题。

### 7.5.4 在不应该转换时尝试转换

以前，我利用不同的状态来改善对“总是可保存”这条原则的履行。但一个“严峻的”时间段是当用户想要在状态图中进行一次转换的时候。当然，我们可以用类似于前面讨论的方式来处理这个问题，这样进入到一个临时状态中，然后由服务来负责转换到实际状态或失败状态。我猜想对于有些情况来说，这种方法是有意义的，但对于特性11来说，它可能就是荒谬的，原因就在于上下文。

---

**说明** 读者是否注意到所有这些是如何聚合到一起的？前面曾讨论过`OrderNow()`，它可能看起来是那种能够解决“总是可保存”问题的技术解决方案，但它是需求的核心。根据需求，`Accept()`可能更有意义。

---

“荒谬”是一个很强烈的词，但如果我们假设要检查的规则只是一些简单的约束，而不涉及跨聚合边界的问题，那么我宁愿应用前面讨论过的原则，即让用户做主动检查，如果结果为正，那么状态更改方法将成功执行。代码比语言更有说服力：

```
[Test]
public void CanMakeStateTransitionSafely()
{
    Order o = new Order(new Customer());

    Assert.IsTrue(o.IsOKToAccept);
}
```

```
//We can now safely do this without getting an exception...
o.Accept();
}
```

检查IsOKToAccept属性是可选的,但如果不检查它,而在一个不恰当的时刻调用Accept(),这就是一种异常的做法了,并且会得到一个异常。异常是由异常问题引起的。对于“预期问题”,bool就可以了。

我们通常不仅仅是关心它是否正确,而且还想得到有关它为什么不正确的一个原因列表。在这些情况下,主动方法可以返回一个被破坏的规则列表,可能像下面这样:

```
[Test]
public void CanMakeStateTransitionSafely()
{
    Order o = new Order(new Customer());

    Assert.AreEqual(0, o.BrokenRulesIfAccept.Count);
    //We can now safely do this without getting an exception...
    o.Accept();
}
```

我喜欢逐个情况地进行检查。没有标准接口或其他的东西可去实现,只有一条用于接口设计的原则。因此,这实际上只是一个规程,目的是使得对被破坏的规则进行主动检测成为可能。

### 事务抽象

事务抽象是一种功能强大的抽象,也是一种未被充分利用的抽象。转换问题的一种替代解决方案就是将事务抽象用于领域模型和验证。这样,在事务执行期间,状态是否正确就不重要了,重要的是在事务执行之前或之后状态是否正确。例如,API可能会像下面这样:

```
//Some consumer...
DMTransaction dMTx = Something.StartTransaction();
c.Name = string.Empty;
c.Name = "Volvo";
dMTx.Commit();
```

在这段代码中,直到Commit()执行之前,不会对规则进行检查。我们知道,这会引发大量需要处理的问题,但它可能是一种未来的有趣解决方案。

如果考虑.NET 2.0的新的System.Transaction名称空间,那么此解决方案就更有趣了。在将来,我们可能有事务列表类、散列表等。如果考虑跨越领域模型和数据库的事务,它甚至更有趣。同理,问题有很多,但也非常有趣。

显然,状态转换问题有很多变体。一个这样的问题就是决定何时应该在状态转换的同时执行一个操作。请看下面的讨论。

## 7.5.5 业务ID

状态转换与操作成对出现的一个例子是特性列表中的第7项。

此问题的一个自然的解决方案是让数据库用一个内置机制来处理此问题，例如用于微软SQL Server的IDENTITY。但这种方法存在一些问题。首先，我们将会有一些领域模型外部的处理工作，这是有代价的。我们需要刷新受影响的实例，这并不是很困难，但需要对其进行跟踪，而且它可能有些复杂。

另一个问题是，值可能过早地确定下来了，此时用户还远远没有决定继续执行Order，用户仍然只是考虑它处于“也许”状态。（显然，对此也有解决方案。）

还有一个问题是，如果同时使用了IDENTITY作为主键，那么O/R映射工具通常将导致问题。使用Guid作为主键（它通常只需较小的数据库耦合）的语义通常略有不同。

---

**说明** 我绝对不是说只是因为有了IDENTITY列，就必须使用它（就像在SQL Server中调用自动增加属性一样）作为主键。我们可以只将它当作一个备用的键，但我们时常会禁不住使用它作为主键。

---

那么，替代方法是什么？有几种替代方法。或许最简单也是最直接的一种方法是让存储库中有一个方法，此方法以某种方法捕获下一个自由的Id。它可能看起来像下面这样（假设Order不应该得到OrderNumber，直到它被接受）：

```
//Order
public void Accept()
{
    if (_status != OrderStatus.NewOrder)
        throw new ApplicationException
            ("You can only call Accept() for NewOrder orders.");
    _status = OrderStatus.Accepted;
    _orderNumber = _orderRepository.GetNextOrderNumber();
}
```

---

**说明** 与其使用ApplicationException（或者一个子类），不如使用InvalidOperationException或它的派生类。

---

然而，这会引发大量问题，至少在两种情况下易引发问题：一是在经常处于某种压力下的系统中，二是如果OrderNumber序列中不允许有孔（hole）。这意味着GetNextOrderNumber()不会将更改写到数据库中，或者至少不会提交这些更改。这意味着需要一个昂贵的锁，来包装Accept()调用及其后面的保存方法。

另一种方法是使用可能成为下一个值的那个值，然后做好值发生重复的准备（如果其他人已经使用它的话）。

如果在执行持久化方法（persist method）期间已经准备好了捕获异常（这些异常是由于重复的键引发的），那么接下来做什么呢？可以再次调用Accept()，但然后Accept()将以它当前被编写的方式抛出一个异常。

或许更好的方法是在代码中请求一个新的OrderNumber，以此来尝试从重复键的异常中恢

复。这进而又会产生另一系列的问题，例如在执行保存方法期间与数据库对话，或者检测哪些实例存在问题（而且，如果实例有多个候选键的话，选择哪个列）。

所有这些问题都可以避免，只要满足两个条件：一是接受数字序列中有孔，二是用于捕获下一个值的方法也提交当前最大值，而且其工作方式类似于IDENTITY在SQL Server中的工作方式。这里的意思是，IDENTITY允许有孔，如果执行ROLLBACK，它将浪费一个值。

为了使这一点更清晰：GetNextOrderNumber()实际上是对数据库进行即时修改，并且不等特对保存方法的下一次调用，我认为我们应该使用一个服务来替代。在执行提交等操作时，存储库不会自己做出决定，但服务则可以，因为服务具有自主性和原子性。

因此，在做出这些更改后，Accept()方法现在看起来可能像下面这样（假设在调用之前，IOrderNumberService已经被注入到order中的\_orderNumberService）：

```
//Order
public void Accept()
{
    if (_status != OrderStatus.NewOrder)
        throw new ApplicationException
            ("You can only call Accept() for NewOrder orders");

    _status = OrderStatus.Accepted;
    _orderNumber = _orderNumberService.GetNextOrderNumber();
}
```

现在，在执行保存方法期间产生问题的风险就大大降低了，除非意外情况，否则不会出现

问题。以上是通过接受序列中有孔来避免问题（对于已定义的需求来说，这实际上是行得通的），但有时这却行不通。我非常不喜欢这样的序列，但有些情况下我们确实无法避免它们。那么怎么办呢？我猜想得从头开始。如果这还不够，我可能会将整个转换方法变成一个服务，将订单转换为Accepted状态，并设置一个ordernumber。这可能不像在领域模型中工作那样单纯或清楚，但当实际需要时，我们就应该这样做。

到目前为止，我们已经讨论了很多需要处理的与领域模型规则有关的不同问题。在继续讨论之前，还要再看一个问题。

### 7.5.6 避免问题

特性9可作为一个非常好的示例，它说明了这样一种情况：如果可能的话，我将根本不想有这样一条规则，但是在构造方法中，却一定要执行它。这样就不需要检查它了。

整洁、简单、有效，刚刚好。我知道，这为读者敲响了警钟。当然，也存在问题。

例如，我们可能希望在不必首先决定客户的情况下有一个订单的实例（由于UI设计的缘故）。

说明 当然，这可以在UI中处理，有几种不同的方式。在不必要的情况下创建一个“问题”给人的感觉并不好。

另一个问题是，创建代码（creation code）可能变得如此复杂，以至于它本身就会抛出异常，然后我们又会再次陷入如何公开被破坏的规则等一些问题中。还应指出，出现异常应该是一种反常情况，因此以下说法是不正确的：在创建代码中抛出异常是由于有一条被破坏的规则。

有一件事情在某种程度上可以减轻负担。我们可以让特性需求描述持久实例，而不是暂时实例。例如，可以考虑OrderLine。在我当前的设计中，OrderLine本身并不是一个聚合，而是Order聚合的一部分。因此，使OrderLine持久化的唯一方法是使用AddOrderLine()方法将它关联到一个Order。在那之前，我可能不关心OrderLine没有Order。OrderLine不能被存储，保存方法也不会由于孤立的OrderLine而抛出异常（因为保存方法根本就不知道它的存在）。

当然，OrderLine中的一些处理可能需要知道Order的知识，但如果没有这种需求的话，我认为这就是一个很好的解决方案了。

---

**说明** 现在，我认为在OrderLine是否需要其Order的知识这个问题上，是存在争议的。可能在大多数情况下，我们可以避免这种双向性，有些情况则无法避免。

---

我不知道你是否已经注意到我反复地讨论一种DDD模式——聚合。现在，我又要暂时中断一下，在继续前进之前讨论一下该模式。

### 7.5.7 再次将聚合作为工具

用于检查规则的单元就是聚合！我发现聚合设计极为重要，而且在我的设计中我也在这上面花费了大量时间。

本章前面曾简单地讨论了规则聚合，现在我们进一步讨论它。

#### 1. 聚合根，聚合规则

当聚合根的使用者查询是否有破坏的持久化规则时，所接收到的被破坏的规则将来自完整的聚合，而不是仅来自根实例。

聚合不一定必须为子类使用相同的IValidatableRegardingPersistence接口。我实际上主张不使用它，因为它将扰乱用于验证持久化的主动检查策略。（不是导致严重问题，而是在原因不明的情况下导致重复的执行周期。）

此外，这还导致聚合根必须决定是否应该使用子类的普通规则。上下文是关键。

#### 2. 一条明显的聚合级规则：版本控制

这段有关聚合的简短一节将以特性4的简单讨论结束。并发冲突检测是非常重要的。该特性可以看作是一种业务规则，但我倾向于将其看作技术问题，而不是业务问题。而且大多数O/R映射程序都至少具有一些并发冲突检测支持，但这里暂不讨论，这部分内容推迟到第8章和第9章再讨论。

让我们稍微转移一下注意力。在讨论API的实现之前，先来看一下是否应该增加任何东西。

## 7.6 扩展 API

我们已经讨论过与持久化规则有关的规则API，以及如何获取与其有关的问题信息。对于

API, 我们还需要做些什么?

根据正在构建的应用程序, 我们可能实际上需要很多其他与规则有关的事情。在本章开头的原则的启发下, 一个立即想到的事情就是: 能够主动检查规则到底有多大的好处, 特别是在UI中。另一个重要事情是能够添加自定义规则, 不仅在编译时可以添加, 而且在部署时也可以这样做。我们从在UI中使用规则开始讨论。

### 7.6.1 查询用于设置 UI 的规则

我曾提到[Nilsson NED]我想在用户界面中重用逻辑层的规则。不是重复, 而是重用。现在就是重用的时候了。

这里将要使用的方法是让类公开那些可以被UI使用的方法(如果UI需要使用这些方法的话)。

一个类似的方法是Web服务用于公开XSD模式时所采取的方法, 即提供有关数据的需求。我要采用的方法略有不同: 提供具有类似信息的实例列表。

---

说明 Martin Fowler在一些细节中描述了问题[Fowler Fixed-LengthString]。他提出了一个略微不同的方法, 例如, 一个字符串对象知道它自己的最大长度。

---

### 7.6.2 使注入规则成为可能

能够逐项定义规则是很好的, 因为一些规则显然在很大程度上依赖上下文。

通常情况下, 我们希望使用者能够添加自己的特定规则, 而且, 如果我们创建并公开了一种供使用者用来添加规则的简单语言, 那么我们希望使用者能够很容易使用此语言来添加规则。

## 7.7 对实现进行精化

到目前为止, 我们已经从使用者的角度集中讨论了规则API。有关如何实现规则的思想也是很有趣的。下面就让我们揭开Order类的面纱, 并尝试一些思想。

在实际开始“精化”之前, 需要有一个较为简单的问题解决方案作为开始。

### 7.7.1 一个初步实现

让我们想一下……首先, 需要一个测试。我们可以重用前面的一些东西, 只需稍做修改。可以重用CantExceedStringLengthWhenPersisting()测试。此测试如下:

```
[Test]
public void CantExceedStringLengthWhenPersisting()
{
    Order o = new Order(new Customer());
    o.Note = "0123456789012345678901234567890";

    Assert.IsFalse(o.IsValidRegardingPersistence);
}
```

我们需要在要编译的测试的Order类上编写一个IsValidRegardingPersistence属性。它可能像下面这样：

```
//Order
public bool IsValidRegardingPersistence
{
    get
    {
        if (Note.Length > 30)
            return false;

        return true;
    }
}
```

**说明** 当然，我并未忘记TDD的“口诀”：红，绿，重构；红，绿，重构。我只是不想减慢读者的阅读速度。

好，这就可以了。让我们复制一个测试的副本（剪切板继承），更改它的名称，然后稍微扩展。

```
[Test]
public void TryingIdeasWithTheRulesAPI()
{
    Order o = new Order(new Customer());
    o.Note = "012345678901234567890123456789";
    Assert.IsTrue(o.IsValidRegardingPersistence);

    o.OrderDate = DateTime.Today.AddDays(1);
    Assert.IsFalse(o.IsValidRegardingPersistence);

    o.OrderDate = DateTime.Today;
    Assert.IsTrue(o.IsValidRegardingPersistence);

    o.Note += "012345";
    Assert.IsFalse(o.IsValidRegardingPersistence);
}
```

正如你所见，我添加了一个额外的规则，说明OrderDate不能是未来的一个日期，而且由于某种原因（原因并不是延伸很远），该规则是一条有关持久化的必须执行的规则。

**说明** 为了使OrderDate规则明显（且必须）与基础架构关联起来，必须让它的长度恰好等于数据库中DATETIME的长度（如果这二者的长度存在不同的话）。

为了得到绿色测试，我们需要对IsValidRegardingPersistence稍微修改，像下面这样：

```
//Order
public bool IsValidRegardingPersistence
{
    get
```

```

    {
        if (Note.Length > 30)
            return false;

        if (OrderDate > DateTime.Today)
            return false;

        return true;
    }
}

```

就目前的规则复杂度来讲，以上代码就可以工作了。代码并不多，对吗？

下一步是向用户报告，因此我们需要编写一个BrokenRulesRegardingPersistence属性。再一次，我们做一些基本工作，但首先需要对测试做少量修改。

```

[Test]
public void TryingIdeasWithTheRulesAPI()
{
    Order o = new Order(new Customer());
    o.Note = "012345678901234567890123456789";
    Assert.IsTrue(o.IsValidRegardingPersistence);
    Assert.AreEqual(0, o.BrokenRulesRegardingPersistence.Count);

    o.OrderDate = DateTime.Today.AddDays(1);
    Assert.IsFalse(o.IsValidRegardingPersistence);
    Assert.AreEqual(1, o.BrokenRulesRegardingPersistence.Count);

    o.OrderDate = DateTime.Today;
    Assert.IsTrue(o.IsValidRegardingPersistence);
    Assert.AreEqual(0, o.BrokenRulesRegardingPersistence.Count);

    o.Note += "012345";
    Assert.IsFalse(o.IsValidRegardingPersistence);
    Assert.AreEqual(1, o.BrokenRulesRegardingPersistence.Count);
}

```

然后编写第一个实现：

```

//Order
public IList BrokenRulesRegardingPersistence
{
    get
    {
        IList brokenRules = new ArrayList();

        if (Note.Length > 30)
            brokenRules.Add("Note is too long.");

        if (OrderDate > DateTime.Today)
            brokenRules.Add("OrderDate is in the future.");

        return brokenRules;
    }
}

```





这通常并不是有关被破坏规则的足够详细的信息，但至少我们已经开始了。现在，有一件感觉更糟的事情是，我不希望在`IsValidRegardingPersistence`和`BrokenRulesRegardingPersistence`属性中表达相同的规则，因此，现在我们将`IsValidRegardingPersistence`更改为一个执行效率较低的解决方案，它消除了重复。

**说明** 如果把这种执行上的低效性放到与一些数据库调用的关系中，那么它通常并不是一个很令人担忧的问题。

```
//Order
public bool IsValidRegardingPersistence
{
    get { return BrokenRulesRegardingPersistence.Count == 0; }
}
```

我们已经确定了查询实例是否处于有效状态（可以在任何时候进行保存），以及当前问题是什么（“确定”是一种大话，但这里这样说并不过分）。

还有另一个与转换相关的规程。让我们来处理转换，并获取与问题有关的信息。我们将被动地处理各种情况（即，如果在有已知问题的时候尝试进行转换，那么我们将做出反应）。

假设有这样一条规则：要想对一个`order`执行`Accept()`，那么该`order`的`Customer`必须处于`Accepted`状态。该规则可能像下面这样：

```
[Test]
public void TryingTheAcceptTransitionWithTheRulesAPI()
{
    Order o = new Order(new Customer());

    Assert.IsFalse(o.IsOKToAccept);
    o.Customer.Accept();
    Assert.IsTrue(o.IsOKToAccept);
}
```

应该特别注意的是，新`Order`（`NewOrder`状态）允许有一个未被接受的`Customer`。

然后，如果令`Customer`进入`Accepted`状态，则也允许对`Order`执行`Accept()`。

另一件要说明的事情是，在转换时，`Order`的通用规则也适用，因此，我们将其添加到测试中。

```
[Test]
public void TryingTheAcceptTransitionWithTheRulesAPI()
{
    Order o = new Order(new Customer());

    Assert.IsFalse(o.IsOKToAccept);
    o.Customer.Accept();
    Assert.IsTrue(o.IsOKToAccept);

    o.OrderDate = DateTime.Today.AddDays(1);
    Assert.IsFalse(o.IsOKToAccept);
}
```

```
try
{
    o.Accept();
    Assert.Fail();
}
catch (ApplicationException ex) {}
}
```

说明 同时还会发生一件事：在构造方法中，OrderDate的默认新Order被设置为DateTime.Today。这就是为什么它直接就是IsOKToAccept的原因。

这里也显示了如果在转换无效时尝试转换，将会发生什么情况。

为此，我们需要在Order中加入一些操作。首先，在IsOKToAccept中加入一些操作。

```
//Order
public bool IsOKToAccept
{
    get
    {
        if (Customer.Status != CustomerStatus.Accepted)
            return false;

        return IsValidRegardingPersistence;
    }
}
```

可以看到，我首先检查了特定规则（或者说在此情况中的规则），然后检查与持久化有关的规则，因为我不希望接下来遇到无法保存的情况。因此，我希望尽早地指出问题（也是与持久化有关的问题）。

如果也想用一个方法来通知在调用Accept()时会有哪些被破坏的规则，那么仍然需要对测试做一些修改，但这很简单，现在显示与否也无关紧要。

Accept()方法也同样如此。它相当简单，只需要在执行转换之前检查IsOKToAccept。

可能稍微有些“复杂”（或者说不太明显）的是：现在IsValidRegardingPersistence的规则变得有些复杂，因为在进入Order的Accepted状态之后，我们将不得不在IsValidRegardingPersistence属性中检查相同的规则。我们将实际的处理转移到BrokenRulesRegardingPersistence属性中（正如我们在前面看到的），因此这也是我们将看到修改的地方。

```
//Order
public IList BrokenRulesRegardingPersistence
{
    get
    {
        IList brokenRules = new ArrayList();

        if (Note.Length > 30)
            brokenRules.Add("Note is too long.");
    }
}
```

```

        if (_orderDate > DateTime.Today)
            brokenRules.Add("OrderDate is in the future.");

        if (_OrderIsInThisStateOrBeyond(OrderStatus.Accepted))
            _CollectBrokenRulesRegardingAccepted(brokenRules);

        return brokenRules;
    }
}

```

因此，在 `_CollectBrokenRulesRegardingAccepted()` 中，将有一个针对进入 `Accepted` 状态的特定规则评估（在这种情况下是已经处于 `Accepted` 状态）。

**说明** 你可能已经注意到，我对 `_CollectBrokenRulesRegardingAccepted()` 使用了收集参数（Collecting Parameter）模式[Beck SBPP]。我发送了被破坏规则的列表，并要求方法在适当的时候向这个列表添加更多被破坏的规则。

对所有这些总结一下就是，我们可以识别大量糟糕的和缺乏的功能。例如

- ❑ `BrokenRulesRegardingPersistence` 中的代码爆炸，因为规则评估是手工编码的。
- ❑ 返回的有关被破坏规则的信息只是字符串，这些信息并不够。
- ❑ 尚未解决自定义问题。
- ❑ 尚未解决与规则本身有关的信息提供问题（这些信息供UI使用）。

现在该到精化这个初步实现的时候了，这里不显示精化的具体迭代过程。我们重用测试，并对实现略加泛化。

### 7.7.2 创建规则类，离开最不成熟的阶段

我认为创建用于满足基本需求的规则类（以便可以通过声明来使用它们）将减轻代码爆炸和手工规则评估问题，并且可以处理自定义问题，以及向外界提供有关规则的信息。

复杂规则通常需要为每种情况进行手工编码，而且很难通过在配置文件中添加信息来进行自定义。此外，向一般表单提供有关复杂规则的信息并不重要。这些规则并不是我的目标。我并不是对它们不感兴趣，相反，我需要对它们进行手工编码。如果它们可以很好地适应其他代码，那么情况就很好，但我想说的是，声明它们并不重要。

应该声明的是简单规则：那些没必要反复手工编写的规则。让我们来尝试一下。

**说明** 一如既往，我们要注意不要陷入事先构建框架的陷阱中。通常更好的做法是让它们代码中显示自己，然后从那里开始工作[Fowler Harvested Framework]。

我的规则类应实现一个名为 `IRule` 的接口：

```

public interface IRule
{
    bool IsValid {get;}
}

```

```
int RuleId {get;}
string[] ParticipatingLogicalFields {get;}
}
```

IsValid用于检查规则是否有效。RuleId用于使规则可以被翻译为信息这样的内容。它需要一些额外信息（通常是长的常量列表的形式），但这里不讨论它，这些内容留给读者。

有关ParticipatingLogicalFields的思想是，使用者应该能够选出那些影响某个特定的被破坏规则的字段。

并且，一个规则类可能看起来像下面的示例。此示例用于检查一个日期在特定范围之内。

```
public class DateIsInRangeRule : RuleBase
{
    public readonly DateTime MinDate;
    public readonly DateTime MaxDate;

    //Note: Only the "largest" constructor is shown here.
    public DateIsInRangeRule(DateTime minDate, DateTime maxDate,
        int ruleId, string[] participatingLogicalFields,
        string fieldName, object holder)
        : base(ruleId, participatingLogicalFields, fieldname
            , holder)
    {
        MinDate = minDate;
        MaxDate = maxDate;
    }
    public override bool IsValid
    {
        get
        {
            DateTime value = (DateTime)base.GetValue();

            return (value >= MinDate && value <= MaxDate);
        }
    }
}
```

我还需要一个基类（RuleBase），它提供通用功能，例如通过反射来读取值，它还用于从子类分离出基础架构部分。留给子类的所有东西就是“有趣的”代码（以及一些构造方法——前面的代码片段中只显示了一个）。

**说明** 顺便说一下，fieldName可以是一个字段或属性。

如果读者想知道holder是什么，它就是领域模型实例，例如一个订单实例，它使用规则对象。

我认为这里用一个图可以帮助解释不同的部分（参见图7-2）。

然后我们“只”需根据不同需要编写类即可，例如DateIsInRangeRule。这部分代码具有很高的可重用性，但需要花费一定的工作量。

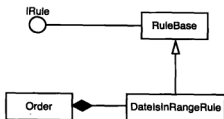


图7-2 IRule、RuleBase、DateIsInRangeRule以及一个领域模型类的概览图

读者可能想知道为什么我想要使用反射。“想要”一词并不完全准确，但我需要反射的积极效果。我的意思是，我希望能够定义一次规则，然后在不提供值的情况下执行它（即规则无需显式地根据这些值进行检查）。相反，我希望提供具有值的字段或属性。让我们看一下这是如何实现的。

**说明** 一个可能的优化是只在实际需要的时候才使用反射。例如，当我们使用引用类型时，不必通过反射来得到相同的效果。另一方面，这确实使复杂性稍有增加，因此要非常注意。Gregory Young给出了如下建议：“这里的另一个答案可能是使用代码生成来分摊反射的开销。”

### 7.7.3 设置规则列表

到目前为止，我们有了一个规则类。假设还有一个用于MaxStringLengthRule的类似规则。现在该到应用它们的时候了。当在Order类中使用它们时，代码可能像下面这样：

```

//Order
private IList _persistenceRelatedRules = new ArrayList();
private DateTime _orderDate;
public string Note = string.Empty;

private void _SetUpPersistenceRelatedRules()
{
    _persistenceRelatedRules.Add(new DateIsInRangeRule
        (MyDateTime.MinValue, DateTime.Today, "OrderDate", this));

    _persistenceRelatedRules.Add(new MaxStringLengthRule
        (30, "Note", this));
}
  
```

然后，在Order的构造方法中，只需调用\_SetUpPersistenceRelatedRules()。

**说明** 我还没有为规则实例提供好的标识符。因此，这里使用了一个构造方法，在这个构造方法中没有提供任何值。

### 7.7.4 使用规则列表

现在已经有了与持久化相关的规则了。让我们看一下这将如何影响BrokenRulesRegarding-Persistence实现。

```
//Order
public IList BrokenRulesRegardingPersistence
{
    get
    {
        IList brokenRules = new ArrayList();

        RuleBase.CollectBrokenRules(brokenRules
            , _persistenceRelatedRules);

        if (!_OrderIsInThisStateOrBeyond(OrderStatus.Accepted)
            _CollectBrokenRulesRegardingAccepted(brokenRules);

        return brokenRules;
    }
}
```

**说明** 你是否记得我曾说过被破坏规则的列表是通知模式[Fowler PoEAA2]的一个实现？我认为它仍然是的，但现在略有不同了。

Gregory Young对不同之处是这样描述的：“关键区别在于，实际规则（IRule）是相对于代理对象（surrogate object）公开的，它只是简单地说明它已经被破坏了，像在通知模式中那样。”

此解决方案的一个很好的额外优点是很容易编写RuleBase.IsValid()（我们为它提供了\_persistenceRelatedRules列表，作为参数）。这样，就可以在无需增加代码重复量的情况下提高IsValid属性的执行效率。也就是说，该实现应该只查看它是否找到任何无效的规则。它不用收集一个被破坏规则的列表，也不必遍历所有规则（即使第一个规则被破坏），等等。

回到基本的流程上来。我们来看一下RuleBase中的CollectBrokenRules()是什么情况。

```
//RuleBase
public static void CollectBrokenRules
(IList brokenRules, IList rulesToCheck)
{
    foreach (IRule r in rulesToCheck)
        if (! r.IsValid)
            brokenRules.Add(r);
}
```

这是与持久化有关的规则，代码直接明了。但每个转换的规则又是什么情况呢？

### 7.7.5 处理子列表

此问题的一个解决方案是设置特定于转换的列表，例如\_acceptedSpecificRules。这一步

完成后，可以像下面这样编写BrokenRulesRegardingPersistence方法。

```
//Order
public IList BrokenRulesRegardingPersistence
{
    get
    {
        IList brokenRules = new ArrayList();

        RuleBase.CollectBrokenRules(brokenRules
            , _persistenceRelatedRules);

        if (_OrderIsInThisStateOrBeyond(OrderStatus.Accepted)
            RuleBase.CollectBrokenRules(brokenRules,
                _acceptedSpecificRules);

        return brokenRules;
    }
}
```

这样，大部分工作就转换为定义规则列表了（对于此方法，我们仍然讨论到这里就为止了，但实际上还有很多工作要做）。

现在，我将所有规则放到一个桶中（如果有基于转换的列表，那么实际上就是若干个桶）：这是一个基于实例的桶。稍做处理后，到目前为止所示的两个规则示例（DateIsInRangeRule和MaxStringLengthRule）都可以被更改为一个静态规则列表。我需要更改日期规则，以便它自己就可以评估DateTime.Today是什么，而不用将它拿到构造方法中。我还可以为构造方法提供相对日期，例如-1和1表示今天的前一天和后一天。还可以创建某种功能，让静态列表只存活到新日期到来之时。当然，如果可能的话，应尽力将规则移动到静态列表中。这将潜在地减少大量开销。这样，它就不会过多地影响原则。

与范围有关的细节讨论暂时到这里，我们继续前进。

## 7.7.6 一个API改进

我的朋友Ingemar Lundberg指出，使用BrokenRulesRegardingPersistence的属性看起来有些奇怪，而且Christian Crowhurst建议更好的做法是让方法使用一个收集参数（collecting parameter），并返回bool。代码如下：

```
public bool CheckX(IList brokenRules)
```

你是否记得IsOKToAccept和BrokenRulesIfAccept？这种思想将减小API的规模，并且不会将检查和详细的错误信息分割为两个单独的片段。这两个片段可以被很好地重写到一个方法中，像下面这样：

```
public bool CheckOKToAccept(IList brokenRules)
```

使用者代码将略受影响，像下面这样：

```
//Some consumer...
```

```
if (! o.IsOKToAccept)
    _Show(o.BrokenRulesIfAccept);
```

我们通常会编写如下代码：

```
//Some consumer...
IList brokenRules = new ArrayList();
if (! o.CheckOKToAccept(brokenRules))
    _Show(brokenRules);
```

brokenRules作为一个收集参数，如果当CheckOKToAccept()开始时，它没有被初始化(null)，那么CheckOKToAccept()可能忽略它，并且只返回bool。这至少是一种方法，但对没有参数的CheckOKToAccept()进行重载感觉上意图更明确。(它实际上不是分裂为两个方法，而只是从一个重载到另一个重载的链锁反应。此外，即使bool为True，返回的列表也可以包含警告。)

另外一个结果是它使使用者避免了两次规则检查代码的执行，一次是IsOKToAccept，另一次是BrokenRulesIfAccept。例如，客户端不必编写以下代码。

```
//Some consumer...
if (! o.CheckOKToAccept())
{
    IList brokenRules = new ArrayList();
    o.CheckOKToAccept(brokenRules);
    _Show(brokenRules);
}
```

我喜欢这一点。

## 7.7.7 自定义

有很多种方法可对领域模型进行自定义，例如，可以让使用应用程序的客户添加一些他们的行为。一个有吸引力的解决方案是使用装饰模式[GoF Design Patterns]，另一种解决方案是提供继承挂钩(inheritance hook)和可配置的创建代码。

另一种(也是更简单的)技术是允许定义规则，或者至少允许添加特定的规则实例，方法是以可配置文件的形式来描述这些规则。

当前这个解决方案草稿已经具备了大部分功能。我们所需的是添加规则的方式，因此开始编写一个新的接口，名为ICustomRules。

```
interface ICustomRules
{
    void AddCustomRule(IRule r);
}
```

在这个草稿中，我只假设我们可以添加与持久化有关的规则，而且它们被添加到基于实例的规则列表中。如果需要，可以对其进行精化(这也是应该做的)。

---

**说明** 谈起向静态列表添加规则实例时，使我想起了我以前的一个愿望。我也希望能够定义静态接口。为什么不这样做呢？

---



现在，我们可以在领域模型类实例化期间添加更多规则了，但正如我所说，实例化有些复杂，或许应该使用某种类型的工厂。

好消息是在进行上述自定义的过程中，可以提取出有关规则的信息，这两项工作可以无缝地一起进行，提取出来的信息可以非常方便地用于创建表单。

### 7.7.8 为使用者提供元数据

与持久化有关的规则应该被作为一种可以从领域模型实例中主动提取出来的内容。这是为了让使用者能够轻松地使用它们。我为 `IValidatableRegardingPersistence` 接口添加了另一个方法。

```
interface IValidatableRegardingPersistence
{
    bool IsValidRegardingPersistence {get;}
    IList BrokenRulesRegardingPersistence {get;}
    IList PersistenceRelatedRules {get;}
}
```

然后，由使用者来决定对其所发现的规则执行哪些操作。当UI显示新表单的时候，可能有一些规则很容易被UI使用，例如所需的字段、特定的值范围、字符串的长度，等等。

从概念上讲，我喜欢这种解决方案。这样的解决方案应该能够用于很多项目。但我仍然感觉它不够完善，特别是在特定于转换的规则方面，至少当问题的规模增大的时候。

### 7.7.9 是否适合用模式来解决此问题

你认为在这里使用状态模式[GoF Design Patterns]会怎样？我们曾在第2章中讨论过很多有关状态模式的内容。如果你问我，那么我会说它是非常有吸引力的，而且它意味着我可以将特定于转换的列表的责任委托给特定的状态类。实际上，状态模式在很大程度上就是用于解决问题的。

也可以使用继承层次结构中有关不同状态的几个层，例如，如果一个基于实例的规则定义对于几种状态都相同的话。

### 7.7.10 复杂规则又是什么情况

正如我所说，我不介意手工编写复杂规则。我希望把时间花在这上面。

对复杂规则应用规格模式[Evans DDD]通常是一个好的思想。

当使用规格模式时，我们的目标是在代码中表达重要的领域概念。DDD的一个公共主题是聚焦于概念。（重点在领域概念上，而不仅仅是技术概念。例如，数字间隔和字符串长度更多的时候是技术概念。）

例如，可能有一个规格类，名为 `ReadyToInvoiceSpecification`，它有一个像下面这样的方法：

```
public bool Test(Order o);
```

让我们举一个快速且简单的示例。假设要为某个客户创建单一的一张发票，即使此客户当前有多个可以开具发票的订单。或许你想询问客户是否有什么物品要开具发票。Customer类上的一个方法可能像下面这样（假设已经注入了一个IOrderRepository）：

```
//Customer
public IList OrdersToInvoice(ReadyToInvoiceSpecification
    specification)
{
    IList orders = _orderRepository.GetOrders(this);
    IList result = new ArrayList();
    foreach (Order o in orders)
    {
        if (specification.Test(o))
            result.Add(o);
    }
    return result;
}
```

某个特定设计是否是推荐或者甚至是可能的，通常依赖于上下文，但这里的重点只是为读者提供一个简单的示例，用以显示如何应用概念，并显示出代码非常清晰且灵活。（记住，规格参数在被发送之前已经配置了。）

再一次，使订单可以开发票的那些规则被封装在规格中了，而不是分散在几个方法中。

**说明** 规格实现的一个很好的特性是，如果可以在数据库中对其进行评估，那么它可用于以概念为中心的查询。

规格方法的问题是如何让UI能够自动进行一些预先的智能处理，而不让规则被破坏。另一方面，我通常可以容忍这个问题，特别是当规格用于处理一些更高级问题的时候（这些问题很难前瞻性地解决）。另一方面，如果一个规格正在使用一个类似于IRule的概念，那么规格中的一些部分可用于前瞻性地自动设置UI。

还必须指出的是封装为规格的复杂规则为单元测试带来了好处。单独测试规格是很容易的。

## 7.8 绑定到持久化抽象

在前一章中，我们讨论了如何通过使用一个抽象层来准备持久化基础架构，我将该抽象层称为NWorkspace（当然，它只是这样的抽象层的一个示例）。正如你可能已经猜到的，它也具有一些验证支持，因为当调用PersistAll()时，需要处理一些被动性质（reactive nature）。

当然，我并不希望持久化抽象必须使用一个特定接口才能在聚合根上实现。

### 7.8.1 使验证接口成为可插入的

因此，我使得将IValidator注入IWorkspace成为可能。IValidator接口如下：

```
public interface IValidator
{
```

```

bool IsValidatable(object entity);
bool IsValid(object entity);
IList BrokenRules(object entity);
}

```

IWorkspace实现与IValidator实现进行对话，然后，IValidator实现决定在实体上寻找什么接口。在我的示例中，IValidator实现寻找IValidatableRegardingPersistence实现，并且知道对IValidatableRegardingPersistence实现执行什么操作才能在执行PersistAll()期间抛出异常（如果必要的话）。为了更具体地说明，参见图7-3。

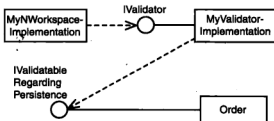


图7-3 协作部分概要图

正如对数据库本身一样，IValidator是被动式的，但它的反应比数据库早了一步，而且通常在略微更复杂和更关注领域的规则上做出反应。

为了尽早测试，我还有一个IValidator的实现，名为ValidatorThatDoesNothing。它可以用于在不想对接口进行验证的情况下进行单元测试。然后，可以将ValidatorThatDoesNothing注入到IWorkspace实现中，而不必注入到那个寻找IValidatableRegardingPersistence实例的实现。编写它需要30秒，我认为与它的用处相比，这30秒是值得的。

## 7.8.2 在保存方面实现被动验证的替代解决方案

前面曾提到，特定的O/R映射工具是有问题的，问题就在于保存过程中的验证点上能够做些什么。因此，思考一下如何处理此问题是有意义的。

在我的实现中，我喜欢实时检查规则。也就是说，当某人想要知道的时候才进行检查。如前所述，我喜欢利用服务来处理在验证点上能够做什么。

一种替代解决方案是在每次更改属性之后，更新被破坏的规则列表，以便在必要的时候进行预先计算。取决于调用模式的不同，效率也有所不同。

**说明** Gregory Young指出，利用前述的规则上的元数据（哪些字段影响哪些规则），可以很有效地重新检查规则，而且总能保持最新的验证。

一个问题是需要脏跟踪提供一些帮助，否则就要在setter方法中自己编写大量代码（除非利用AOP来处理横切关注点（crosscutting concern）。我们还必须使用属性来代替公共字段，即使公共字段也能工作得很好）。

说明。这种风格用于CSLA.NET框架中[Lhotka BO]，此框架还支持多步骤撤销操作（我们尚未讨论它）。

另一个解决方案是保持一个IsValidated标志，每次重新生成被破坏规则列表时，都将其设置为true，而且在每次更改后，将其设置为false。在保存过程的验证点上，我们希望IsValidated为true，否则就将抛出一个异常。这对编程模型有轻微的影响，因为它使得IsValidRegardingPersistence调用或BrokenRulesRegardingPersistence调用成为必需的。另一个缺点（如果必需的调用是一个缺点的话，我认为它是一个缺点）是与前面刚刚提到的脏跟踪具有类似问题。

不管怎样，准备几种不同的方法是有利的，万一首选的方法有问题，就可以使用其他的方法。

### 7.8.3 重用映射元数据

在继续讨论之前，我需要从我的系统中总结出另一个思想。前面曾提到，我喜欢这个原则：定义一次，然后可以在大多数适用的地方执行（例如在UI、领域模型或数据库中）。这种思想非常符合此原则。

如果你将大量精力花费在手工定义简单规则上，例如最大字符串长度，那么就应该考虑动态设置这些规则了，而不是通过读取某位置的元数据来设置。例如，数据库中包含信息，但根据映射解决方案的不同，使用这些信息可能会有一点麻烦。另一个问题是元数据映射数据本身。或许我们将在元数据中找到一些规则的来源。

不管怎样，上述思想的一个重要部分是：当应用程序开始时，从某个地方读取元数据，动态设置规则，这样就可以用了。

当然，像本章中讨论的其他思想一样，这并不是每一个问题的答案。它只是用于解决庞大问题的一个工具。

## 7.9 使用泛型和匿名方法

如果使用的是一种支持泛型的环境（例如.NET 2.0），那么当然可以将泛型应用于前面讨论过的解决方案提议，并且可以稍微提高清晰度、类型安全性和性能。这适用于到目前为止所讨论的大部分章节。

下面举一个小例子：

```
ICollection<BrokenRulesRegardingPersistence> {get;}
```

可以像下面这样使用泛型：

```
ICollection<IRule> BrokenRulesRegardingPersistence {get;}
```

我们并不是必须要实现一个重大的改进（正如本书通篇所展示的那样），而是要有一个能够对代码中的很多地方产生积极影响的改进。

在我工作的环境中，还有另一个没有的技术，即匿名方法（类似于其他语言中的块或闭包）。

思想是能够将一个代码块作为参数发送给方法，但代码块可以使用接收它的方法的范围之内

的变量。

有些人认为它是一种在不创建新类的情况下实现规格的方式，但我认为这意味着我们在规格命名上将失去一些明确性，而这种明确性恰恰是规格的强大特性之一。相反，我现在认为匿名方法是可为重用资产提供变量的更好技术。例如，我们可以根据自己的需要注入方法实现的一部分。这是另一种考虑自定义的方式。

为了给那些新接触此技术的读者提供一些初步的宏观印象，我们用前面讨论的泛型和匿名方法组成一个代码片段，并看一下是否可以改进它。读者是否记得用于判断Customer是否需要开发票的那个循环？代码如下：

```
//Customer
public IList OrdersToInvoice(ReadyToInvoiceSpecification
    specification)
{
    IList orders = _orderRepository.GetOrders(this);
    IList result = new ArrayList();
    foreach (Order o in orders)
    {
        if (specification.Test(o))
            result.Add(o);
    }

    return result;
}
```

这里是再次稍加修改的版本：

```
//Customer
public List<Order>OrdersToInvoice(Predicate<Order>
    readyToInvoicePredicate)
{
    List<Order> orders = _orderRepository.GetOrders(this);
    return orders.FindAll(readyToInvoicePredicate);
}
```

这次，参数是一个断言，用于判断Order是否需要开发票。调用提供了匿名方法作为参数。调用可以像下面这样，假设此时只需使用一个非常简单的算法：

```
c.OrdersToInvoice(delegate(Order o) {return o.Status == OrderStatus.ToBeInvoiced;});
```

该“规格”或断言被发送到泛型List的FindAll()方法，以便过滤出可以开发票的那些订单。在编写本书时，有关泛型和匿名方法的内容对我来说都有些早了，但我敢肯定在不久的将来，它们将极大改变我的设计。我并不是指像上面的示例那样的小的方面，而更可能是一种思想的转变。

但同时，它们也只是工具箱中的一些更多的工具而已，当然，它们不会改变任何事情。

## 7.10 其他人都做了什么

前面曾偶尔提到过其他人在这方面所做的工作和所写的著作，在本章结束之前，我们来看一

下另外两个资源。

第一个资源是*Streamlined Object Modeling* [Nicola et al. SOM], 此书全面论述了在面向对象上下文中的不同设计和实现规则及策略。

第二个资源是*Enterprise Patterns and MDA* [Arlow/Neustadt Archetype Patterns], 此书讨论了用于组织规则的原型模式。读者将会从该书中获得很多有用的思想。

## 7.11 小结

虽然经过了多年研究,但在这个主题上,我们感觉仍然有些不够成熟,而且没有明显的解决方案。我们已经接触了几种用于处理规则的思想,有些是典型的,有些则不是。

我们讨论了很多作为实现技术的规则对象,但也接触了一些其他的变体。当然,没有适用于所有情况的解决方案。例如,我所介绍的规则对象可能最适用于很简单的规则以及有很多规则的情况。

本章遗留下来未做讨论的主要问题是,我们应该尽最大可能解除持久化与领域相关规则之间的耦合。(虽然如此,有些规则仍然会与持久化有联系。)使用不同的状态是一个非常强大的解决方案,可帮助实现“所有状态都应该是可保存的”这个目标。最后,要记住规则与上下文有很大的联系,而且要重点关注规则的模型意义。

在本章中,我们处理了第4章中定义的很多特性需求。下面继续讨论所需的基础架构,重点关注持久化支持。



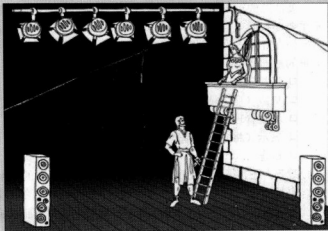
# Part 3

## 第三部分

### 应用 PoEAA

在这一部分中，我们将 Fowler 的《企业应用架构模式》[Fowler PoEAA] 中的几种模式放到上下文中，以此来讨论基础架构需要为领域模型提供哪些持久化支持。我们将看一下如何通过一个示例工具来满足这些需求。

这一部分内容离开前面的主题，主要关注基础架构。基础架构是一个非常有价值的方面。



#### 本部分内容

- 第 8 章 用于持久化的基础架构
- 第 9 章 应用 NHibernate

现在讨论到哪里了？我们已经从头构建了一个领域模型，它是DDD[Evans DDD]风格的。到目前为止，我们已经尽最大可能推迟与基础架构有关的决策。当然，第6章中讨论了很多如何简化基础架构添加的内容，但现在是时候添加一些用于支持领域模型的机制了，即基础架构。此外，本章还将对第4章中用来满足特性列表的领域模型做出一些修改。

要注意的重要一点是，这些讨论是必不可少的，而不是单单为了讨论而讨论。这并不是说它不重要，我们只是想尽可能把大部分精力放在领域问题上。

但在开始讨论基础架构之前，一个显然的问题是需要什么样的基础架构？事实上，我们需要大量的基础架构，例如用于实现以下功能的基础架构。

- 授权
- 集成（服务请求和响应）
- 数据管理和访问（持久化）
- 表示（将它称为基础架构可能有些名不副实了）
- 日志

等等，还有很多。我看问题的方式可能是由于我的数据库背景造成的，但我认为持久化最明显，也是最有趣的基础架构。至少这是下面章节中要集中关注的。

当定义所需的持久化基础架构时，我们也将对当前领域模型设计稍加修改，以便使它更好地适合典型的持久化基础架构。

---

**说明** 不使用迭代的设计方法是很危险的。即使你只负责领域模型和持久化，也应该在领域模型上进行少量的工作，选择和尝试持久化基础架构，使用领域模型，检查持久化基础架构，等等。特别是最开始的几个DDD项目更应如此，但这在某程度上也总是一种好的思想。

还要澄清的一点是：我深刻理解UI的至关重要性，而且仅仅将UI说成“机制”可能是一种不当的说法。当然，我（和用户）非常重视有用的UI。而且项目时间的很大一部分花费在UI上也是一种常见情况。尽管如此，这里的重点是领域模型。讨论良好UI设计不是本书的目的（尽管第11章涵盖了一些有关UI的内容）。

---



我们已经决定了集中关注持久化基础架构。下一个问题就是：要想将注意力集中在领域上，持久化基础架构都有哪些需求？

## 8.1 持久化基础架构的需求

再次强调，我希望尽量让基础架构不妨碍领域模型，以便集中精力创建一个功能强大的领域模型，从而在不过多分心的前提下解决业务问题。例如，我们应该能够尽可能使用高级别的持久化透明（PI），因此这就是一个基础架构需求。

同时，还有一点也很重要，即不要为领域模型的使用者施加过多的责任。我希望为用户提供提供一个非常简单的API，以便程序员能够将注意力集中在那些对他们来说重要的事情上——为他们的用户提供良好的体验。

第5章中已经定义了持久化实体的生命周期所需的特性，表8-1重新列出这些特性。

表8-1 领域模型实例的生命周期语义总结

操 作	结果（短暂的/持久的）
新调用	短暂的
Repository.Add(instance)或persistentInstance.Add(instance)	在领域模型中是持久的
x.PersistAll()	在数据库中是持久的
Repository.Get()	在领域模型（和数据库）中是持久的
Repository.Delete(instance)	短暂的（而且在调用x.PersistAll()时将数据库删除实例）

另一种思考持久化基础架构需求的方式是关注当前的非功能性需求。典型的示例是可伸缩性、安全性和可维护性。

当然，这有很大的差别，不仅是由于需求，而且是因为对于同一组非功能性需求来说，持久化框架也是有时成功，有时失败。导致这些问题的原因在于成功与否取决于应用程序的类型和使用模式等。我们必须负责定义自己的非功能性需求组合，并检查特定持久化框架是否能够通过仔细检查来满足这些需求。记住，如果非功能性需求过高，那么将产生很多额外代价。

**说明** 通常，我们很难让客户表达任何非功能性需求。如果让客户来表达，那么有种风险就是他要求每件事情都达到完美，这就是为什么要记住代价的原因所在。

有关更多非功能性需求的讨论，参见 [POSA 1]，[Fowler PoEAA]或[Nilsson NED]。

你可能已经注意到，我并没有把以下这一点作为持久化框架上的一项需求提出来：持久化框架在使用者与领域模型之间有一个网络。建立一个网络是很容易的，就像我们在Valhalla[Valhalla]中所做的那样。但是，更好的方式可能是在涉及网络的地方沿着显式边界线来思考，并以面向服务的方式来设计交叉边界。因此远程表示往往是必不可少的。

这样，我们为领域模型定义了要满足的需求，并且为持久化框架定义了如何使用API。

现在有了需求，让我们继续前进，并做出一些有关基础架构的选择。首先是用于存储数据的位置。

## 8.2 将数据存储到哪里

假设我们从一张干净的白纸开始，我们将如何存储数据？至少有4种选择。

- RAM
- 文件系统
- 对象数据库
- 关系数据库

---

**说明** 我认识到这可能是一种苹果和桔子的情况，因为它是一个由两部分组成的问题：一部分是存储什么（对象、诸如XML这样的层次结构或表），另一部分是存储到哪里（RAM、文件系统、对象数据库或关系数据库）。但这并不是没有问题的。在多次思考之后，我认为我最初描述它的方式是足够好的，因此我们继续。

---

下面从RAM开始。

### 8.2.1 RAM

将数据存储到RAM中并不一定像刚听上去那样愚蠢。我的思路是将领域模型看作数据库本身，同时将所有更改的持久化存储保持为磁盘上的一个记录，以便在系统崩溃时可以重新创建领域模型。

这种方式的优点是避免了从领域模型到其他地方的映射。领域模型本身就是持久的。

也可以在内存中存储层次结构，XML文档就是一个典型示例。但这确实会导致一些阻抗失配问题，因为使用了两种模型，因此需要在它们之间进行转换。另一方面，这也得到了一些不需要代价的功能，例如查询功能（如果你发现XPath和XQuery是好的查询语言的话）。

无论在RAM中“存储”什么，一个问题是RAM的大小。除去系统和其他应用程序所使用的RAM外，如果机器上还剩余2 GB的RAM，那么数据库就不应该超过2 GB，否则就会影响性能。

另一方面，64位的服务器越来越普及，而且RAM的价格也一直在降低。对于大部分应用程序来说，当前硬件和下一代硬件通常就足够了，至少对于最大规模的数据库来说是足够的。

另一个问题是在系统崩溃后重建领域模型是要花费时间的，因为恢复一个大的记录将耗时。如果基于RAM的解决方案经常在磁盘上创建当前领域模型的快照，那么问题就最小化了。但问题仍然存在；而且创建快照本身也是有问题的，因为它将削弱系统的能力，而且在创建快照时可能导致其他请求处于长时间的等待状态。

---

**说明** Gregory Young指出，为了减少快照的问题，可以在领域中使用上下文边界，而且可以为它们单独创建快照。

---

当对模式做出修改时，这种方法会遇到一个更严重的问题。我们必须将对象序列化到磁盘上，然后使用新模式再将它们反序列化到领域模型中，这是一项繁重的任务。使用XML来代替纯的

类有助于缓解此问题。

另一个大的问题是事务问题。要满足原子性、一致性、不相关性和持久性（ACID），很难避免不对可伸缩性造成大的影响。首先，在这种情况下与其使用“试试看”的方法，不如尽力准备好一个事务，以便查看任务可否可能成功。（“我需要做这件事，它会不会使我陷入麻烦？”）当然，这无法解决全部问题，但至少可以减少问题。在这里，保持主动性是有利的，特别是考虑到我们有很高的效率，因为可能不涉及进程边界或网络。

---

**说明** 这取决于拓扑。如果使用另一个机器上的领域模型（因此“数据库”也在这台另外的机器上），那么就需要跨越进程边界和网络。

---

再次强调，保持主动性并不会提供任何事务语义，而只是减少了回滚的可能。我曾听到的一种方法是，如果需要事务的话，那么要求RAM的大小是领域模型的2倍，以便可以在领域模型的副本上进行更改。如果所有更改都是成功的，再在实际领域模型中重做这些更改。同时，领域模型本质上是单用户的。记住，操作速度通常很快，但看起来对我来说并不是这样。我也将它看作是一个不成熟的信号，因此希望将来能有更好的解决方案。在本书编写时，有一些轻量级的事务管理器正在使用，或者正在被引入，这可能是此问题的更好解决方案。

还有一个问题是，没有明显的查询语言选择。有一个开源语言可作为备选，但现在仍然不够成熟。当考虑到终端用户的报告和临时查询需求（而不只是来自应用程序内部的查询）时，问题更加突出。例如，在这种情况下，典型的终端用户报告编写工具的用处是很小的。

---

**说明** 如果可能的话，应该在一台专用服务器上完成报告，并使用专用的数据库，这样，这个问题就比我们最初预期的小多了。另一方面，在现实中，在支持日常事务工作方面，对于什么是报告，什么是列表，通常存在一个灰色地带。这也是一个问题。

---

然而，另一个问题在于，领域模型中的导航通常是基于对列表进行遍历的。可能没有内置的索引支持。当然，可以在一些地方使用散列表，但它们只能解决部分问题。也可以在必要的时候添加一个内存内（in-memory）索引解决方案。另一方面，我们应该注意，这个问题不会像对于基于磁盘的解决方案那样在早期就出现。

最后，正如我曾几次提到的，这种方法略显不成熟，但它非常有趣，而且对于未来是很有希望的，至少在特定情形中是这样。

### 8.2.2 文件系统

另一种解决方案是使用文件系统来代替RAM。要保存的内容与在RAM中保存的内容是相同的，即领域模型对象或XML。事实上，此解决方案可能非常接近于RAM解决方案。如果数据库很小的话，那么它们可能是相同的，而且可能“只有”当RAM被填充到一定程度时，才溢出到磁盘上。

此方法与RAM解决方案具有类似的问题，有一点除外：在这种情况下RAM的大小不再是一

个很大的限制因素。另一方面，性能特征也不太明显。

我相信一种很有吸引力的做法是编写自己的领域模型（或XML文档）存储解决方案，但通常当我们决定部署自己的基础架构时，必须准备好应付大量工作。我知道，“它太难了”或“太复杂”是最能鼓舞开发人员的一句话，因此，如果没有人这样做过，那么现在可能每个人都正在编写自己的基于文件系统的解决方案，不是吗？我们应该有所准备，最初的简单只是欺骗性的，麻烦在细节之中，而且随着工作的进展，复杂度也会增加。

如果决定构建一个在保存时可能会溢出到磁盘上的领域模型，那么所创建的系统实际上非常像是一个对象数据库。（或许这给出了一个更好的工作量和复杂度的感觉。）

### 8.2.3 对象数据库

历来，对象数据库有很多种不同的风格，但共同特征是尝试避免对象与其他一些存储格式之间的转换。在某种程度上，这样做的原因与推迟向领域模型添加基础架构的原因是相同的，还有就是性能原因。

---

**说明** 如果考虑混合（例如对象-关系数据库），那么将有更多种风格，但我认为这些混合通常是来自关系背景的风格，而不是来自面向对象一端。

---

事实证明，分散注意力的事情并非不存在。事实上，可以说阻抗失配仍然存在，但它在对象与关系数据库之间搭建了一座桥梁，因此使用对象数据库还是非常干净的。

到目前为止，对象数据库的问题如下所示。

- 缺少标准。
- 未达到“关键多数”阶段。
- 成熟度。
- 更多时候不是大众产品。
- 与其他系统的集成。
- 报告。

---

**说明** 我的一个“损友”Martin Rosén-Lidholm曾指出，与.NET世界中的面向数据的解决方案相比，很多相同的参数实际上可用于DDD和O/R映射，考虑微软的应用程序块、纸张、指南，等等。

我暂时不考虑他的这一观点。对于微软的指南来说，领域上的焦点可能变得越来越流行，Martin也完全同意这一点。

---

我当然并不是对象数据库方面的专家。这些年来，我使用过几种对象数据库，仅此而已。有关更多信息，我推荐两本我最喜欢的书：[Cattell ODM]和[Connolly/Begg DB Systems]。

有一次，大概是在1994年，当时我认为对象数据库被作为事实标准了。但这种思想纯粹是基于技术特征的，事实却并不这样简单。对象数据库在十年之前是大有希望的。现在，它们大部分

时候用于特定情况，但最重要的是，它们仍然只是有希望而已。我认为当今的事实标准仍然是关系数据库。

### 8.2.4 关系数据库

如前所述，在应用程序中存储数据的事实解决方案是使用关系数据库，即使使用领域模型，也仍然如此。

在关系数据库中存储数据意味着数据是以表格格式存储的，其中每一项都是数据，包括关系。在很多应用程序中，这被证明是一种简单且（足够）有效的解决方案。但没有任何一种解决方案是毫无问题的，在这种解决方案中，当想要在关系数据库中存储领域模型时，问题就是阻抗失配问题。第1章中已经对这个问题有过详尽的讨论了。

如果采取此路线，那么最常见的解决方案是使用数据映射模式[Fowler PoEAA]的实现。数据映射模式的目的是在领域模型与持久化表示之间搭建一座桥梁，用于双向传送数据。稍后我们将回头讨论这种模式。

使用哪种存储解决方案并不是一个明显的选择。但我们仍然必须做出选择。

在选择和继续讨论之前，先来考虑几个其他问题。

### 8.2.5 使用一个还是多个资源管理器

另一个完全不同的问题是应该使用一个还是多个资源管理器。事实可能证明我们没有选择。

领域模型在有多个资源管理器的情况下表现非常优秀，因为如果必要的话，它完全可以向使用者隐藏这种复杂性。但我们也应该清楚地认识到，多个资源管理器的存在增加了将领域模型映射到其持久存储的复杂性。为了使讨论更简单，这里假设只使用一个资源管理器。

### 8.2.6 其他因素

在现实中，我们很少是从一张干净的白纸开始的。有很多影响决策的因素，例如我们自己知道的那些内容。提高效率的一种方式是我们所熟知的技术。

除了前面讨论的纯技术因素之外（这些因素中并没有哪一个是明显的决定性因素），其他的典型因素是客户已经在哪些系统中进行了投资（购买和对员工做了培训）。

解决方案的成熟度对数据有很大的影响。丢失业务流程的关键数据是无法接受的，因此如果选择了一种客户认为不可靠的解决方案，那么客户通常是非常挑剔的。

### 8.2.7 选择和前进

考虑技术原因，以及上述的其他因素，关系数据库毫无疑问是通常的选择。因此，我们假设使用关系数据库，并继续前进。这看起来是一个适当的选择，也将是未来很长一段时间中的默认选择。

使用这种解决方案后，实际上需要在持久化基础架构的需求列表上增加一些需求。

- 仔细地处理关系数据库。（尽可能接近其手工编程方式。）
- 强大的查询支持。（在很大程度上，这正是前一条需求的重点。）

- 并发冲突检测支持。
- 高级映射支持，例如不同的继承策略（即使我们在数据库中可能非常谨慎地使用“继承”）和细粒度的领域模型类型。

如前所述，这样就需要一个数据映射模式的实现。问题是如何实现此模式。

## 8.3 方法

当然，有很多种使用数据映射模式的不同方式，但最典型的使用方式是下面几种。

- 自定义手工编码。
- 自定义代码的代码生成。
- 元数据映射（O/R映射）。

先来讨论自定义手工编码。

### 8.3.1 自定义手工编码

使用这种方式，我们通常编写自己的持久化代码。代码将驻留在存储库中。当然，应该使用帮助方法类，但这不会解决所有问题。一些典型的技术问题如下所示。

- 如何在一个完整的图中保存更改。
- 如何表示和转换查询。
- 是否需要操作单元。
- 是否需要标识映射。
- 是否需要延迟加载。

**说明** 这里有必要用一句话来描述一下上面提到的每种模式。操作单元模式 [Fowler PoEAA] 捕获在执行逻辑操作单元期间对领域模型所做更改的信息。然后，这些信息可用于影响持久化表示。

标识映射模式 [Fowler PoEAA] 在会话中对每个实体只保持一个实例。它像是一个基于标识的缓存。这对于消除阻抗失配有至关重要的意义。当使用对象时，我们希望能够使用内置对象标识，即对象的地址。

最后，延迟加载模式 [Fowler PoEAA] 是在刚刚来得及的时候加载子图。

根据我们设置的需求，操作单元和标识映射是必不可少的。查询也是需要的。延迟加载可能是不需要的。如果偶尔需要它，可以很容易自己解决。尽管如此，但在需要时能够得到延迟加载支持也是一件非常好的事情。

现在就有一些工作要做了。这些只是几个示例，真正要做的比这多得多。

如果决定遵守上面定义的需求，那么实际上是创建一个特定的而且（很有可能）是简化的 O/R 映射工具（如果采用自定义手工编码的话）。这意味着我们离开了这个类别，而转移到了第三个。现在暂时忽略这一点，只假设我们将尽可能不使用操作单元、标识映射、延迟加载、动态和

灵活查询，等等。

首先，必须决定各个方面的工作细节。完成所有这些工作之后，必须将其应用于整个领域模型。这是一个冗长、耗时且易出错的过程。更糟的是，从现在开始，当每次对领域模型做出修改时，都有可能再次遇到问题。

### 8.3.2 自定义代码的代码生成

另一种方法是使用代码生成(例如自定义生成，它是一个使用自定义编写的模板的通用工具，或者在适当的情况下使用购买的完整解决方案)来处理数据映射模式。也就是说，使用一个类似于自定义手工编码的解决方案，但当进行设计时，在代码生成的帮助下反复使用这种方法。

此方法与自定义手工编码具有相同的问题，当然，此外还存在代码生成本身的复杂性问题。优点是当设计本身成熟时，效率将大大提高。

基于代码生成的解决方案的另一个常见问题是源代码控制。领域模型的小的修改常常强迫我们检出(如果所使用的源代码工具需要在做出修改之前检出)用于处理数据映射的所有类，并全部重新生成它们。这个问题可能更糟，而且不易解决。

此外还有更多问题，例如为代码库增加了大量无关代码，这些代码不是给任何人看的，而是供编译器阅读的。

一个常见的问题是生成的代码通常与特定数据库产品紧密耦合。尽管用一个抽象层可以解决此问题，但如果需要支持不同数据库模式的话，问题将仍然存在。这样，就需要保持每个模式都有一个生成的代码库。这也是一个适用于不同数据库产品的可行选项，尽管它并不是一个非常顺畅的解决方案。

另一个问题是很难在运行时进行调优。这里用关系数据库的优化器作为一个比较。只有“what”(内容)才是静态决定的，而“how”(方式)则是在运行时决定的，因此根据情况的不同，方式可能会发生变化。对于目前的大多数应用程序来说，我并不认为这是一个大的问题。此外，如果对于特定场景的相同方法分别应用生成代码和反射代码，那么通常从生成代码可以得到比反射代码更高的性能。

或许一个更糟的问题是新的生成器版本。我有种直觉是，让一个新的生成器版本进入生产状态是很难的，因为它必须重新生成所有代码。

此外，如果数据库模式发生改变，那么在不重新编译的情况下就不可能做出更改。但这无论如何也不是一种推荐的做法。这些类型的更改应由开发人员来完成。

双向代码生成和单的正向代码生成有着相当大的区别。如果采用双向风格，那么对生成代码的修改将被保留，而在正向代码生成中，将永远不会修改生成的代码，因为修改将在下一次生成时丢失。

最后再讨论它的一个优点。基于代码生成的解决方案可能比基于元数据映射的解决方案更容易调试。所有代码都已经就绪了，但代码可能很难理解，而且代码量很大。

通常，进一步探索的催化剂是动态性的缺乏，例如缺乏查询动态性。前面也曾说过，如果决定实现动态查询、操作单元、标识映射等，那么应该认真看一下下一个类别：O/R映射工具。

### 8.3.3 元数据映射（对象关系（O/R）映射工具）

特定风格的数据映射模式就是所说的元数据映射模式。我们在元数据中定义领域模型与关系数据库之间的关系。其余的工作则自动完成。

最典型的实现可能是O/R映射工具。这里将使用术语O/R映射工具来表示一个负责元数据映射的产品家族。

---

**说明** 我的朋友Mats Helander将O/R映射描述为“太极”。

太极由两个同等重要的部分组成。第一部分是学习缓慢地抬起和放下手臂，同时调均呼吸。第二部分是其他动作。这并不是一个玩笑。无论你多么精通太极，都应该在第一部分上花费与所有其他动作同样多的时间。

O/R映射也是由两个同等重要的部分组成。第一部分是在内存中的对象与数据库行之间来回传送数据。另一个部分是其他操作。

---

正如你现在可能已经猜到的，大多数O/R映射工具都具有内置的操作单元、标识映射、延迟加载和查询支持。

但没有任何一种解决方案是毫无问题的。O/R映射工具的一个常见约束是它们不能创建真正好的SQL代码。让我们看一些常见示例。第一个例子是带有WHERE子句的UPDATE。参见以下示例：

```
UPDATE Inventory
SET Balance = Balance - 1
WHERE Id = 42 AND Balance >= 1
```

这意味着当库存中有产品的话，我只想更改余额（balance）。否则，就不更改余额。O/R映射工具通常并不直接支持这一点。相反，这里O/R映射工具采用的方法是通过一个乐观锁定来读取Inventory行，做出更改，然后再把它写回去（最大的希望是没有并发异常）。

要澄清的是，这里是一个乐观方法的例子（在此例中是一个SQL批处理，但注意我们没有保持任何锁，因为没有显式事务，因此这例证了此场景）。

```
--Remember the old Balance.
SET @oldBalance =
  (SELECT Balance
   FROM Inventory
   WHERE Id = 42)

--Calculate what the new Balance should be...
```

```
UPDATE Inventory
SET Balance = @newBalance
WHERE Id = 42 AND Balance = @oldBalance
--If @@ROWCOUNT now is 0, then the update failed!
```

另外一种方法通过一个悲观锁定来读取Inventory行，做出更改，然后再把它写回去。这两



种方法都将有损于可伸缩性。

O/R映射工具的另一个问题是它们对于大量行的更新通常是低效的。通过O/R映射工具来更新所有产品的操作步骤可能是这样的：将所有产品读取到一个列表中，对列表执行循环，并逐个更新产品。如果这可以用单一的UPDATE语句完成，那么吞吐量将极大提高。

还有一个问题是很难平衡要读取的数据量。无论读取的数据量过大还是过小，在延迟加载时都会导致大量双向操作。类型爆炸是此问题的一个常见结果，即定义了类型定义的多种变体。

但我们也可以用另外一种方式来看待这个问题。一个人编写的所有代码是否都能够非常好？是否都是一致的？团队中的所有开发人员是否都像最优秀的那个人一样？即使这些条件都满足，我们是否应该在自动方法已经实现足够好的性能时还要投入这些人员的工作时间？

说明 公正地讲，上述观点也适用于代码生成。

我们发现利弊是共存的。我喜欢这一点，因为它使我感觉到我（在某种程度上）理解了正在讨论的技术。

### 8.3.4 再次选择

这是否是一个简单的选择？当然不是，但我的经验是能够最好地满足已定义需求的方法就是O/R映射工具。从理性上讲，它似乎是适用于很多情况的适当方法，但也是存在问题的。

如果采用YAGNI模式，那么O/R映射工具是有意义的，因为我们可以用一种简单方式来解决问題（有可能不必为领域模型增加抽象），并且具有很好的发展前景。当性能未达到足够好时，可以通过自定义代码来处理这些情况（可能这些情况会很少）。这可能是处理问题的非常有效的方式，至少在不是所有问题都是性能问题的时候。（所有问题都是性能问题的情况是很少见的。）

因此，我们假定选择O/R映射工具。

## 8.4 分类

让我们更仔细地看一下O/R映射工具到底是什么。我们从两个不同的角度来讨论它。首先，从一个维度来讨论它，这个维度就是不同特征。接下来，从一些已实现的PoEAA模式的角度来讨论它。在下一章中，我们将在一些示例中重用这些分类，这些示例讨论了PoEAA模式在特定的O/R映射工具中是如何工作的。

清楚起见，假设要讨论的每个方面都是可用的，即使是自定义代码（映射风格除外）。模式的描述同样如此。但为了使讨论更具体，我们将从现在开始考虑O/R映射工具。

第一个分类是有关所支持的领域模型风格的。

### 8.4.1 领域模型风格

要使用O/R映射工具，必须对领域模型进行多大程度的修改？用什么方式来修改？三个最典

型也是通用的方面是

- 持久化透明
- 继承
- 接口实现

#### 1. 持久化透明

持久化透明意味着不修改领域模型，以便使它可以持久化。当然，这里有一个规模问题，而且这也不是“全黑”或“全白”的问题。例如，基于反射的方法设置了一些需求。基于AOP的方法则设置了其他一些需求。

#### 2. 继承

过去一个常见的方法是要求领域模型类从持久化框架所提供的超类进行继承。

#### 3. 接口实现

最后，过去还有另一种常见的方法，即要求领域模型类实现一个或多个由持久化框架提供的接口。

这只是显示了如何修改领域模型以适应持久化框架的一个方面。可能还需要处理大量与版本控制和延迟加载有关的事情，本章后面将讨论这些问题。

### 8.4.2 映射工具风格

基于元数据的映射通常有两种不同的实现方式。

- 代码生成
- 框架/反射

目光敏锐的读者可能已经注意到，这里只再次提到了代码生成，这次是在O/R映射工具的上文中。然而，有一个重要区别。现在，代码生成已经完成，通常这是在编译之前完成的，采用的方式是读取元数据，并分离出映射代码。当然，自定义代码的生成也可以用类似方式来完成，但主要区别在于要支持的目标不同。这个定义并不是非常清楚，但它是一个开始。

框架/反射意味着在编译之前，没有源代码生成用于映射工作。相反，映射是通过在运行时读取元数据完成的。

---

**说明** 正如读者可能怀疑的那样，事实可能会产生失真。例如，应该如何对静态AOP进行分类？

这里只是试图保持描述和分类的简单和干净。

还要注意的重要一点是，代码生成和框架不一定是互斥的。

---

### 8.4.3 起点

当使用某个特定O/R映射工具时，通常可以从以下列表选择一个或多个起点。

- 数据库模式
- 领域模型
- 映射信息

起点的意思是当开始构建应用程序时，将注意力放在什么地方。当然，如果O/R映射工具支持多个起点，那么将会非常有益，因为这样就有很大机会将它应用于适合自己的不同情况。

现在我更喜欢从领域模型开始工作。但是，我们可能无法总是从它开始，而是不得不从数据库模式开始，或至少需要密切关注数据库模式。也有可能必须从映射信息开始，这些信息使用类似于UML的绘图工具描述了类与表之间的关系。

如果选择从领域模型开始，但要尝试使用的持久化框架不支持此模型，那么可以通过将UML编辑器或数据库设计看作领域模型的编写方式来绕过此问题。但是，这有些笨拙，而且TDD的应用也不自然。

#### 对遗留数据库进行映射

尽管起点可能是数据库，但这并不一定暗示着持久化框架强制这样做，也不暗示着设计者应该首选这种工作方式。有时可能必须使用一个遗留数据库。

要做好准备，用O/R映射工具对遗留数据库进行映射要困难得多（除非映射工具是专门为此设计的），特别是当不允许更改数据库设计的时候。通常，数据库至少允许添加视图和表，即使不允许更改现有列。这可能是有帮助的！（更改那些不影响现有应用程序的内容可能是允许的，但一定要注意，因为现有应用程序可能不是最健壮的。）

如果遗留数据库设计不是非常完善的话，这特别易引起麻烦，这种情况可能在应用程序进入生产环境几年后才发生。

有关这方面的另外一件事情是当“完成”起点之后（例如领域模型），如何前进？所选择的O/R映射工具是否为创建其他两部分提供了任何帮助？例如，已经有了领域模型，现在需要数据库和元数据。是否能够自动获得这两部分？答案是不能，在不支持的情况下这会中断。我们通常很容易创建这样的基本工具。但如果可以自动完成的话，这也是一种优势。

因此，现在假设已经有了领域模型、数据库和映射信息。让我们来讨论一下API将会是什么形式的。

#### 8.4.4 API 焦点

API焦点有两个来源。

- 关系表
- 领域模型

API焦点属于一个延伸事物，因为从一开始，O/R映射工具的目的就是让开发人员使用对象来代替表。因此，对于典型的O/R映射工具来说，API焦点总是领域模型。（关注关系表的一个API示例是Recordset模式，它是通过.NET中的DataTable实现的。）

API的另一个非常重要的方面是查询，并非对于每个O/R映射工具查询都是相同的。

#### 8.4.5 查询风格

很难对查询风格进行分类，但让我们来尝试一下。从开发人员的角度来看，通过O/R映射工

具进行的查询通常以以下一种或多种方式完成。

- 基于字符串的SQL风格的查询。
- 基于查询对象的查询。

这里有必要稍微详细地介绍一下每种方式。

### 1. 基于字符串的SQL风格的查询

这种风格的查询语言很类似于SQL，但它使用领域模型中的类，而不是数据库中的表。一些查询（特别是高级查询）可以用这样的语言很好地表达出来。另一方面，典型的缺点就是缺乏类型安全性。

但是，要注意的重要一点是，这种表达查询的方式在实现级别上不一定是基于字符串的，而只有在API级别才是基于字符串的。

### 2. 基于查询对象的查询

第二种典型的查询语言是使用对象来表示查询，遵循查询对象模式[Fowler PoEAA]。通常，简单查询都可以用这种方法更好地表示，它也有可能具有类型安全性，并且开发人员不必了解大量有关查询语义的知识即可获得很高的效率。但是，问题在于复杂查询可能需要用大量代码来表示，而且代码很快就变得难以阅读。

### 3. 原始SQL

另一种适用于一些少见情况的较好解决方案是能够用SQL来表示查询，但接收的结果是领域模型实例。当然，这种方法的问题是导致代码与数据库的耦合，但如果找不到另一种解决方案，那么至少这是一个极好的选项。当需要优化时，此方法也非常方便。而且，如果需要将实体作为结果，并且SQL集成可以自动为我们处理这件事，那么这将减少需要编写的代码量。

当然，也应该允许在不接收领域模型实例结果的情况下跳转到SQL。

### 4. 选择哪种方法

理想情况下，工具应该支持多种方法。这是因为有时基于字符串的表示是最好的；而有些情况下基于查询对象的表示则是最适当的。

查询风格本身只是整体的一部分。另一个重要方面是查询解决问题的能力如何。这是下一个主题的一部分。

## 8.4.6 高级数据库支持

换种说法就是：“数据库管理员将有多尊重你？”

为了足够灵活以便不必跳出沙箱，O/R映射工具需要支持一些从领域模型角度来看不太明显的数据库关闭操作。到优化时间时，这些操作将提供极大的方便。

一些示例包括如下这些。

#### □ 聚合

SQL的一个非常基本的特性是能够执行诸如SELECT SUM(x)、MIN(y)、MAX(z)这样的操作。当以领域模型为焦点时，需要这些聚合的机会略微降低，但它们仍然很有用，而且有时是必要的。

#### □ 排序

可能最高效的排序方法是在无需在领域模型中进行排序的情况下对数据库中的结果集进行排序。

#### □ 分组

使用SQL时，另外一件事情是可以使用GROUP BY。这对于临时查询和报告尤为常用，但有时在以领域模型为焦点的应用程序中也很有用。

#### □ 标量查询

并不是在所有情况下都需要获取完整的实例。有时只需字段值，或许有些值来自某个类，而另外一些值则来自另一个类。

如果有其他方式可以将一些功能移到数据库中，那么也将是一件很大的优点，这些功能最适合放在数据库中，它们通常是一些数据密集型的操作。这里考虑是调用存储过程的能力，但也可能包括其他一些用于移动功能的自定义方式，例如用户定义的功能。

这方面的主要问题是可能会损失应用程序和数据库设计的可移植性。但如果将它用作一种优化技术，而且仅在必要时使用，那么问题就被最小化了。

另一个大的问题是，如果我们跳出沙箱，那么就只能靠自己了，但这还不是问题的全部。我们还必须谨慎地处理与沙箱的集成问题。情况可能是必须在调用存储过程后清除标识映射。没有人能为我们做出决定，必须自己做出判断。

**说明** 这是一个很好的示例，印证了Joel Spolsky在他的文章“The Law of Leaky Abstractions”[Spolsky Leaky Abstractions]中所说的话。抽象是一个伟大的功能，但我们必须掌握它背后的大量原理知识。

我认为这就是思考和使用O/R映射工具的方式。我们不应该期望它们隐藏每件事情，而是需要知道抽象背后正在发生什么事情。将O/R映射工具看作一个工具，让它帮助我们完成那些可以手工完成，但实际上不想手工完成的乏味任务。

### 8.4.7 其他功能

以上就是特性的整个范围，但还有一些其他特性，例如

#### □ 支持哪些后端

如果特殊的O/R映射工具支持几种不同的后端，那么这将会是一个很好的特性，这样就不会只绑定到一个后端，例如Oracle或SQL Server。

#### □ 不仅是一个O/R映射工具

很多O/R映射工具产品具有更多功能。例如，它们可以帮助进行数据绑定（在无需自定义编码的情况下将UI中的控件绑定到对象），帮助设置业务规则，或帮助进行双向管理。

#### □ 高级缓存

高级缓存对读者来说不一定非常重要，但如果确实需要它，那么所选择的产品支持高级缓存就是非常好的。高级缓存并不仅仅指标识映射形式的缓存，还包括用于查询的缓存。

例如，当查询特定客户时，该查询可以在二级缓存中执行，而不接触数据库。

#### □ 事务控制的高级支持

此特性的例子包括事务控制的程度如何，这包括多个方面，例如隔离级别，通过避免从读锁定到写锁定的升级来减小死锁风险，以及是否支持分布式事务。这些是完全不同的方面，但可能都非常重要，这一点毫无疑问。

这方面还有一个有趣的问题是使用什么技术进行事务控制。典型的选项是手工控制、基于拦截的控制以及通过持久化管理器来隐藏所有事情。

#### □ 开放源码

开放源码与商业产品相比的利弊并不在本书的讨论范畴之内。但当我们选择O/R映射工具时，它仍然是一个要考虑的因素。

#### □ 版本（第几代）

这是最后一项了，考虑一下O/R映射工具是第几代是很重要的。这也表明了我们所期望的成熟度。

记住这句话：“有时更少意味着更多。”可能产品的焦点越突出，它对细节的关注就越少。当然，事实并不是必须如此。这里只是想指出仅通过计算特性数目是无法判断哪个产品是最好的。下面从一个完全不同的角度来看一下分类。

## 8.5 另一个分类：基础架构模式

在分类的第二部分中，将使用一些PoEAA模式[Fowler PoEAA]，其中的一些模式以基础架构为焦点。

再次强调，即使读者选择自定义路线，这段描述可能也是有用的。或许这段描述将帮助一些读者放弃创建自定义解决方案的想法，而是从现有解决方案中选择一个。这通常是一种好的思想，因为构建自己的完整解决方案需要非常大的工作量。既然有现成的，就不妨试一下。

### 8.5.1 元数据映射：元数据的类型

我们需要用元数据来描述领域模型与数据库模式之间的关系，这就是元数据映射 [Fowler PoEAA]的全部。O/R映射工具是元数据映射模式的实现，但不同的映射工具使用不同类型的元数据。典型的示例包括

#### □ XML文档或其他文档格式

#### □ 属性（注释可能是一个更清晰的术语，例如C#中的[MyAttribute]）

#### □ 源代码

像通常那样，每种类型的元数据都有自身的弊病。例如，XML文档受到“XML地狱”的困扰。对于大多数开发人员来说，XML并不是一种使用效率非常高的格式。人们常说，XML不是供人们使用的，而是供解析器使用的，但现实情况是工具未达到标准，因此我们经常发现自己坐在那里编辑巨大的XML文档。

XML文档的另一个问题在于它们位于领域模型源代码的外部，因此很容易导致互相不同步

的问题。而且，很多IDE缺乏对XML文档语义的理解，因此不能无缝地进行重构工作。

属性用于修饰领域模型，因此与领域模型失去同步的风险在某种程度上来说是较小的。这种情况下的一个更大的问题是领域模型与数据库的耦合稍微大一些。如果打算将领域模型与两个不同的数据库一起使用，那么使用属性与使用某种外部类型的元数据相比，有两个不同版本的领域模型源代码的风险将加大。这可能不是一个很大的问题，但它有可能变成大问题。在这种情况下，也很难获得对映射的总体理解，但有些工具可以提供帮助。

对于在源代码中提供映射信息是否成为一个单独的类别，是有争议的。它实际上只是另一种文档格式。不管怎样，我认为只有两个类别还是能够稍微降低成本的。这里的区别在于，源代码以过程方式（而不是声明方式）描述了映射信息。我也将这个选项看作介于XML文档和属性之间的一种技术。元数据在源代码中，而且是编译后的，但它的编写方式是作为一个概览提供给我们。对于一些人来说，这意味着“用优雅的C#代码来代替拙劣的XML”。我并不能说我完全反对。这仍是一个深奥的选项，而不是一个常用选项。

像通常那样，选项不一定是“一个且只有一个”。例如，也许信息是作为属性提供的，但可以用XML信息重写。

你是否想知道元数据中都描述了些什么？元数据是领域模型与数据库之间的关系，但要想更具体地了解，就需要一个示例。下面以标识字段作为一个示例。

## 8.5.2 标识字段

实体的标识字段[Fowler PoEAA]中保持数据库基本表中的主键的值。这就是实体实例与表行之间的关系的处理方式。

对于新实体，标识字段的值至少可以在4个不同的层中生成。

- ☐ 使用者
- ☐ 领域模型
- ☐ 数据库
- ☐ O/R映射工具

从O/R映射工具的角度来看，前两个层是非常类似的。站在O/R映射工具的角度看，值是被提供的，而且超出了O/R映射工具的控制范围。O/R映射工具只能寄希望于领域模型的使用者遵守规程。如果可以选择的话，在领域模型和使用之间我更喜欢领域模型。这里，O/R映射工具的一个问题是使用标识字段值来判断实例是否为新，以及是应该插入，还是应该更新。

**说明** 这是让一个对象具有两种职责的典型示例。它很简单，而且刚看上去时是很好的，但具有潜在问题。

第三个选项很常见，但它有一些语义问题。它意味着在保存实体时，数据库使用像IDENTITY (SQL Server) 或Sequence (Oracle) 这样的命令来设置主键的值。问题是这处在生命周期的晚期，而且当标识值发生改变时，实体所在的集合可能会发生问题。

最后，O/R映射工具本身可以负责标识字段的值的生成，这是最方便的解决方案，至少对于O/R映射工具是如此。

综上所述，实体保持两个标识是一种好的思想。一个是自然标识，我们将其称为业务标识(Business Identity)，它可以在生命周期的不同时间获得值，而不一定从一开始就获得值(然而，并不是所有实体都具有一个这样的标识)。每个类有多个这样的标识也是常见情况。

第二个是标识字段，大多是持久化ID或数据库ID。如果将这关系数据库术语进行比较，可以得到表8-2。

表8-2 领域模型术语与关系数据库术语的比较

领域模型	关系数据库
业务ID	替代键
标识字段	主键

[Bauer/King HiA]中对此给出了很好的讨论。

这实际上是一项需要作为当前领域模型中实现细节来应用的内容。例如，Order的OrderNumber是一个业务ID，而不是标识字段。这在Equals()/HashCode()实现中是非常明显的，它为数据集带来了问题。因此添加了Id字段，这些字段的值是在新创建的实体实例被关联到其存储库时由O/R映射工具生成的。之后，这些值将永远不发生改变。这些Id字段使用了Guid，如图8-1所示。

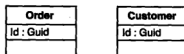


图8-1 为拥有业务ID的实体添加的标识字段

**说明** OrderNumber和CustomerNumber仍在使用。还要注意的，存储库将发生小的变化，原因就在于它们的存在。GetOrder()和GetCustomer()将有新的重载。

这是基础架构非常需要的改变，因此是抽象的典型示例。这并不是仅为某个特定O/R映射工具所做的更改，而是作为一种在某种程度上适用于所有O/R映射工具的简化。

**说明** 有讽刺意味的是，当构建Valhalla时，我们决定应强制性地使用Guid。但我们对这个决定并不是完全满意，因为它为开发人员施加了一个强制要求，而且未来还需要对它做出更改。现在使用其他持久化框架时，可以自由选择。不管怎样，我仍然认为使用Guid作为标识字段通常是一个好的思想。

下面回到元数据的讨论上来，给出另一个有关它包含什么的示例。



### 8.5.3 外键映射

另一个经常出现在元数据中的对象是外键映射模式[Fowler PoEAA]。它是对外键及其在领域模型中关联的一个描述。

与标识字段不同，它不是在领域模型中被复制的值。相反，它只是一种元数据组成。

有很多种关系可以使用，其中O/R映射工具所支持的一种关系可能有很大的不同。就我的经验来看，大多数情况下，我们将只使用相对很少的关系类型，但当需要一些更复杂的关系时，我们将很乐于看到这些关系是受支持的。

### 8.5.4 嵌入值

消除阻抗失配的一个重要方式是在领域模型中使用粗粒度的表和细粒度的类。这就需要使用嵌入值模式[Fowler PoEAA]。

这意味着应该能够将客户存储在数据库的单一Customers表中，但将客户用作领域模型中的一个Customer对象和一个Address对象。（此示例非常简单，在实际中一般会有很大的区别。）

在最简单的表单中（称为第一层），只能够描述嵌入值与数据库表列之间的关系。第二层是编写辅助代码的地方，这些代码帮助进行复杂情况下的转换。

### 8.5.5 继承解决方案

领域模型中的继承层次结构在关系数据库中并没有完美匹配，因为继承不是一个关系数据库概念（至少在SQL:1999之前是这样的，在本书编写时只有很少的数据库产品支持）。此外，领域模型中的继承的使用比很多人的预期要少得多。尽管如此，当需要使用继承时，应该能够通过O/R映射工具来支持它。

此问题有三种不同类型的解决方案。它们是单表继承（Single Table Inheritance）、类表继承（Class Table Inheritance）和具体表继承（Concrete Table Inheritance）[Fowler PoEAA]。之所以将它们归到一起，是因为它们只是相同问题的不同解决方案。

主要区别是用多少个表来存储继承层次结构。假设Person是基类，Student和Teacher是子类。不同模式将导致以下典型的表，如表8-3所示。从这个表可以很容易推导出都有哪些列归属这里。

表8-3 继承层次结构的持久化模式以及所需的表

模 式	数据库中的表
单表继承	People
类表继承	People、Students、Teachers
具体表继承	Student、Teachers

如果O/R映射工具只支持其中的一种，那么数据库设计的灵活性就下降了（与拥有三种选择相比）。我们必须决定这一点是否重要。

### 8.5.6 标识映射

在创建O/R映射的早期尝试中，最初我认为可以忽略标识映射，但结果是它变得太复杂了，而且为使用方程程序员施加了过多的责任。

另一方面，很多事情取决于领域模型设计。如果不同聚合的实体之间一直不存在关系，那么对标识映射的需要就减少了。因此DDD思想（例如简化以及解除领域模型当中的耦合）使得我们更容易在没有标识映射的情况下工作。尽管如此，有一个活动的标识映射还是有用的。

---

**说明** 如果将O/R映射工具看作一匹忙碌于数据库和对象之间的“耕马”，那么标识映射可能就不重要了。但这并不是我们要在这里讨论的，因为这样的话我们将需要做更多工作。这里，我们仅支持把领域模型作为尽可能简单的解决方案（而不是尽可能好）。

---

标识映射也可用于其他方面，而不仅仅用于控制使用者所知道的对象图。例如，当从数据库读取数据时，它可用于在领域模型中的对象之间建立M:N关系。

---

**说明** M:N描述了对象之间的“基数/多重性”（cardinality/multiplicity）关系，这种关系是多对多的。例如，一所房屋里有很多人，同时每个人又可以有他自己的几所房屋。

---

标识映射也通常被认为是用于提高性能的一种缓存，但正如读者所知，我并不是非常喜欢缓存友好的风格，因此我将标识映射看作是编程模型的一个便利之处，而不是作为一种提高性能的手段。

在能够（或必须）将标识映射用于哪个“会话”级别方面，不同的O/R映射工具也有所不同，这个级别可以是机器、进程和/或会话。

另一个通常与标识映射一起使用的模式是操作单元。

### 8.5.7 操作单元

大多数O/R映射工具都使用（或至少是支持）操作单元模式。主要区别实际上是操作单元对于使用的透明度如何。当O/R映射工具是“运行时持久化透明”风格时，使用者可能必须显式地与操作单元对话，以便注册一个新实例（此实例在下次保存操作时被插入）。其他O/R映射工具则更为透明，但这样可能就必须用O/R映射工具提供的一个工厂来进行实例化。这些方法都存在利弊。

“会话”级别可能对于操作单元也是不同的，至少在理论上如此，这一点类似于上面讨论的标识映射。

### 8.5.8 延迟加载/立即加载

前面曾讲过，我的领域模型风格是在聚合[Evans DDD]中不使用太多的延迟加载[Fowler PoEAA]。尽管如此，延迟加载是一个需要在持久化基础架构中使用的难点，并且需要将它用作一种优化手段。

相反，作为聚合的默认策略，我使用了立即加载（eager load）。也可以说成积极加载、贪婪加载、跨越加载或预先加载，无论使用哪种说法都可以。从现在开始，我们将它称为立即加载（eager load）。

立即加载正好与延迟加载相反，立即加载完整的图，而不是将部分图的加载推迟到后面。

立即加载是与聚合[Evans DDD]一起使用的，至少是作为一种默认的解决方案。

**说明** 当谈到读取场景时，有些读者可能认为这里将过多的重点放在聚合上了。我一直将聚合用作默认的加载模式，而且在必要时对其进行优化。

毕竟，聚合对于编写场景是最重要的。当从这个角度来看问题时，在对聚合实例进行更改之前，在规程中添加一项类似于GetForWrite()这样的需求可能是有意义的。GetForWrite()将从聚合加载所有内容，它可能还具有读取的一致性。

如果确实有一个跨越很多实例的概念，而且打算将它作为一个单元，那么可以考虑聚合，而且它是有意义的。

我还认为只读/可写区别通常对于用户来说非常有用，用户必须主动选择进入写模式。另一个好的方面是，当使用乐观并发控制时，用户将不会对一个旧对象做出更改，这意味着冲突的风险减小了。这种方法也可以很好地应用于悲观并发控制模式。

立即加载有很多种不同的实现变体。在SQL中，它通常是用OUTER JOINS处理的，然后结果集被分解为图。另一种最常见的解决方案是将几条SELECT语句编写为批处理操作。

胜任的O/R映射工具应该同时支持延迟加载和不同的立即加载策略，对于列表和单一实例来说都应如此。它还应该可以对单一实例的字段组进行延迟加载，尽管我认为这并不是特别重要的。这样的属性组总是可以被提取到一个值对象中，在大多数情况下这是有很大意义的。这样就再次回到了实例的延迟加载/立即加载。

### 8.5.9 并发控制

正如前面已经多次提到的，聚合模式是用于控制并发性的优秀工具。利用它可以获得想要使用的单元，并将其作为单一整体来使用。然而它并不是完整的解决方案。我们还需要避免冲突，或者需要检测是否已经发生冲突，以避免得到不一致的数据（特别是在不被通知的情况下）。

从[Fowler PoEAA]中，可以找到以下解决方案。

#### □ 粗粒度锁

此模式不采用实例级别的锁定，而是建议对一个更粗粒度的单元进行锁定。例如，可以在聚合根级别上使用它，从而隐式地锁定聚合的所有部分。

#### □ 乐观离线锁

期望没有冲突，但在提交之前检查。

#### □ 悲观离线锁

通过一种独占检出机制来防止冲突。

这提醒了我现在需要为领域模型添加一些版本控制信息，将其作为一种表明需要在哪里处理

并发性控制的方式，以便处理特性4。我认为聚合根应该有一个Version字段，以便支持乐观离线锁。这并非一项自动操作。相反，应该在需要的地方添加它。图8-2显示了更改的地方。

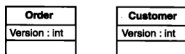


图8-2 为一些聚合根添加的版本字段

O/R映射工具不支持悲观离线锁并不是一个非常严重的问题。首先，使用它时应该非常小心，因为它的开销是很昂贵的。其次，如果需要它，很容易自己构建一个适当的解决方案。（我在前一本书中讨论了一个基于自定义锁定表的解决方案[Nilsson NED]。）

## 8.6 小结

现在，我们已经讨论了在某种程度上对持久化基础架构的需要。在引入了主题并定义了一些持久化基础架构需求之后，我们从不同角度讨论了这个主题，例如分类和一些PoEAA模式。

但整个讨论还是相当抽象的。一些示例有助于使讨论更具体。这就是下一章的目的。在下一章中，我们将使用NHibernate作为示例。

**本**章与大部分其他章节均不同，因为我们将以动手实践为主，而不再讨论某种程度的抽象。本章通过使用NHibernate作为持久化框架的一个示例来应用第8章中的内容。

本章以介绍NHibernate作为开始，包括如何起步、如何处理映射以及API是什么形式的。

然后在第8章所讨论的分类方面给出NHibernate的定位，并看一下NHibernate如何实现前一章所讨论的基础架构模式。

本章以讨论如何将NHibernate放到整个DDD视图中作为结束。

## 9.1 为什么使用 NHibernate

读者可能会奇怪为什么选择NHibernate。原因并不在于NHibernate是可用的最好解决方案。什么车是最好的车？这是个愚蠢的问题，不是吗？标准的答案总是：“看情况。”

以下是本章选择NHibernate的动机。

### □ Java继承

NHibernate是流行的Hibernate[[Hibernate](#)]的一个端口，因此很多人知道它，而且大量的书和其他资源可用。此外，与.NET环境相比，对象关系映射在Java环境中是相当成熟的。

### □ 开放源码

NHibernate是开源的，这使得它非常易于获取和尝试。

本章绝不是NHibernate的详尽指南。目的只是使第8章中的讨论更具体一些。我们也将避免逐个地列举特性，这种风格并不是很有趣，而且在本书完成写作的几星期后，这些特性也就不再是最新的了。

本章内容可能也很容易适合任何其他持久化框架（虽然这并不是主要目的之一），因此比较不同解决方案变得略微容易一些。让我们看一下是否是这样（希望有一个单独的小节是“在比较中应该包括什么其他内容？”）。

本章亦不会在抽象层上下文（例如NWorkspace[[Nilsson Workspace](#)]）中讨论NHibernate。这里的重点将放在NHibernate本身及其API上。

首先对NHibernate做一下介绍。

## 9.2 NHibernate 简介

这里要讨论的是一个用于处理元数据映射[Fowler PoEAA]的产品,它也是一个O/R映射工具,名称是NHibernate。

如前所述, NHibernate是Java环境Hibernate中每个流行O/R映射工具的一个端口,它最初是由Gavin King [Bauer/King HIA]创建的。此端口基于2.1版本的Hibernate,这个版本被认为是一个相当古老的版本,但端口并没有严格地为该特定Hibernate版本迁移代码库。相反,来自后续版本的Hibernate特性以及其他特性被添加到各个地方。

NHibernate (和Hibernate) 是开源的。在本书编写时, NHibernate发布了1.0版本。可以从[NHibernate]下载它。

实践出真知,让我们来看一下如何使用NHibernate。

### 9.2.1 准备

假设有一个领域模型,现在想要根据这个领域模型编写一些使用者代码。

**说明** 一个额外的需求是需要使得领域模型成为持久的。因此,我们还要有一个数据库产品,但现在暂不考虑数据库模式。

首先,需要设置使用者对NHibernate框架的引用(即nhibernate.dll)。这并不是很难。

然后需要配置领域模型使用者对NHibernate的使用。有两种配置方式:在代码中配置和在.config文件中配置。通常使用.config文件,因此这里假设使用这种方式。要配置的一些内容包括使用哪种数据库产品、数据库的连接字符串和日志。在NHibernate站点上可以找到大量.config文件的例子,可供复制和粘贴,因此这里不再讨论细节,假设已经有了一个适当的.config文件。

然后是设置SessionFactory,每个应用程序只能进行一次这样的设置,因为如果创建多个的话,代价会相当高昂。SessionFactory将分析所有元数据,并为此建立基于内存的结构。顾名思义,然后SessionFactory用于对那些实现ISession的新实例进行实例化。

ISession既是一个用于执行数据库操作的字符串,也是一个接口,使用者需要一直与此接口进行交互。

接下来,最好是能用一个帮助方法来管理ISession。要实现此帮助方法,有几种不同的复杂级别,但这里做一个简单的尝试就足以实现我们的目的了。假设领域模型的名称是ADDDP.Ordering.DomainModel,那么一个名为NHConfig的简化帮助方法如下。

```
public class NHConfig
{
    private static ISessionFactory _sessionFactory;

    static NHConfig()
    {
        Configuration config = new Configuration();
        config.AddAssembly("ADDDP.Ordering.DomainModel");
    }
}
```

```
        _sessionFactory = config.BuildSessionFactory();  
    }  
  
    public static ISessionFactory GetSessionFactory()  
    {  
        return _sessionFactory;  
    }  
}
```

再假设正在使用一个富客户端应用程序，因此有一个ISession实例就足够了。至少，我们这样开始。这样，以下代码就适合于应用程序的开始，或者适用于单一窗体。

```
_session = NHConfig.GetSessionFactory().OpenSession();  
_session.Disconnect();
```

这样，当想令ISession开始工作时，可以反复使用以下这个小代码片段。

```
_session.Reconnect();  
  
//Do stuff...  
  
_session.Disconnect();
```

最后，当应用程序终止时（或窗体关闭时，取决于所选择的策略），可以像下面这样关闭ISession。

```
_session.Close();
```

---

**说明** 每个窗体使用一个ISession或每个应用程序使用一个ISession只是几种可能策略中的两种。

---

这可能很有趣，但从完成数据库工作的角度来看，它是完全无用的。打开和关闭ISessions是不够的，我们还需要将更改保存到数据库。为了实现此目的，需要进行一些更多准备。

## 9.2.2 一些映射元数据

在到目前为止的准备工作中，领域模型本身还没有受到影响。然而，我们需要做一些事情，不是为了影响领域模型，而是为了补充它。我们需要添加映射信息，以便显示领域模型与数据库之间的关系。通常，这些映射信息是在每个实体的单独的XML文件中创建的[Evans DDD]，而且这些文件存储在领域模型的项目目录中。

到目前为止，我们尚未创建数据库模式，因此可以按自己喜欢的那样自由工作。现在唯一有的就是领域模型。我们可以使用一些工具来创建映射信息和/或数据库模式，但对于现在讨论的简介来说，唯一重要的事情是在给定特定领域模型实体和数据库表的情况下，显示映射文件是什么形式的。我们从领域模型中选择一个简单的实体。我认为Customer很符合需求。图9-1中给出了实体及其值对象[Evans DDD]。

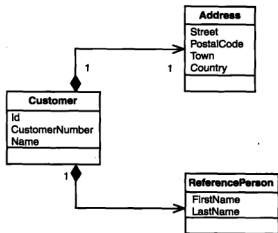


图9-1 Customer实体以及Address和ReferencePerson值对象

在数据库中，Customer和Address可能都存储在一个表中。这两个类的DDL如下。

```

create table Customers (
  Id UNIQUEIDENTIFIER not null,
  CustomerNumber INT not null,
  Name VARCHAR(100) not null,
  Street VARCHAR(50) not null,
  PostalCode VARCHAR(10) not null,
  Town VARCHAR(50) not null,
  Country VARCHAR(50) not null,
  primary key (Id)
)
  
```

**说明** 在这段代码中，Town和Country的编写给人感觉有些奇怪。在实际情况中，可能会将它们提取出来，但对于现在的讨论来说，它并不重要。

上面的DDL中省略了Reference Persons表，但可以从图9-1推导出它。

那么，丢失的部分（在这里也是有趣的部分）就是映射文件。

**说明** 再次强调，请注意我从未说过必须在映射文件之前创建数据库表。工作顺序由自己决定。如果可以的话，我首选从领域模型开始，然后编写映射文件，然后再从映射信息自动生成数据库。为了完成这些工作，可以使用以下代码片段。

```

Configuration config = new Configuration();
config.AddAssembly("ADDDP.Ordering.DomainModel");
SchemaExport se = new SchemaExport(config);
se.Execute(true, true, false, true);
  
```

依我的经验，除了映射文件中描述的内容之外，通常还需要为数据库添加一些更多的约



来,我通过编写一些自定义代码来实现这一点。虽然这种方法有点原始,但它确实能够解决问题。

在做完这一步的工作之后,就可以按自己所需的频率来重新生成开发数据库,甚至在每次测试执行之前生成(如果速度不会变得很慢的话)。

我们一段一段地来编写代码。首先,需要描述文档,像下面这样的Customer.hbm.xml。

```
<?xml version="1.0" encoding="utf-8" ?>
<hibernate-mapping xmlns="urn:hibernate-mapping-2.0"
  namespace="ADDDP.Ordering.DomainModel"
  assembly="ADDDP.Ordering.DomainModel">
```

这里提供了namespace和assembly标签,这样其余的映射信息的表示就轻松多了,因为不必再重复它们。

**说明** 如果希望将元数据放到程序集中,那么不要忘记将XML文件的属性设置为一个嵌入资源(Embedded Resource)。

然后描述实体类的名称和表名称,例如,像下面这样:

```
<class name="Customer" table="Customers">
```

然后描述标识字段[Fowler PoEAA],不仅是类和表中的名称(在下面的示例中,属性名和列名称是相同的,因此不需要列标签),而且还包括用于创建新值的策略。代码如下:

```
<id name="Id" access="field.camelcase-underscore"
  unsaved-value="00000000-0000-0000-0000-000000000000" >
  <generator class="guid" />
</id>
```

在这个特殊示例中,当一个新实例被关联到ISession时,NHibernate将生成一个guid。

然后映射简单的属性,例如Name,代码如下:

```
<property name="Name" type="AnsiString" length="100"
  not-null="true" />
```

通常,通过反射对类进行检查后,NHibernate可以知道使用哪些属性类型,但这里为字符串提供了特定类型,以便在从映射信息自动生成模式时,能够得到VARCHAR,而不是NVARCHAR。length也用于从映射信息生成DDL,但因为我喜欢采取该路线,因此通过额外工作来添加该信息。

**说明** 这类信息对于自定义验证也很有用。

NHibernate不仅允许映射属性,而且还允许对字段进行映射。事实上,Customer.Name不是属性(NHibernate认为它是默认的,因此不必表示出来),而是一个公共字段,因此需要添加一个

access标签以便正确工作，像下面这样：

```
<property name="Name" access="field" type="AnsiString"
length="100" not-null="true" />
```

它适用于所有访问程序类型，甚至是私有类型，因此可以相当自由地选择一种策略。例如，我们经常需要在属性中公开一些与存储内容有关的稍微不同的事情，或者在get/set方法上执行一些拦截操作。在这些情况下，需要对私有字段进行映射。

映射私有字段的代码如下：

```
<property name="_customerNumber"
access="field" not-null="true" />
```

假设对私有字段进行映射，那么使用命名规则是一种好的思想。因为查询代码不必使用私有字段的名称（例如 \_customerNumber），而是可以使用普通的、更好的属性名称（例如 CustomerNumber）。代码如下：

```
<property name="CustomerNumber"
access="field.camelcase-underscore" not-null="true" />
```

**说明** 当然，这里（照常）有一些陷阱，但我发现对私有字段进行映射通常是足够好的，而不仅仅为了持久化方面就引入单独的私有属性。如果映射可以工作，则没有问题。如果无法工作，那么在必要的时候总是可以更改它。

这是一个热议的话题。很多人选择映射到特定于持久化的私有属性。

最后（这一步也可以在简单属性之前完成，这里只是碰巧才出现这种情况），需要描述如何将所使用的值对象映射到单一 Customers 表。Customer 的映射文件可能会像下面这样（Address 类本身没有任何映射信息，因为它是一个值对象，而不是实体）。

```
<component name="Address" access="field">
  <property name="Street" access="field.camelcase-underscore"
type="AnsiString" length="50" not-null="true" />

  <property name="PostalCode"
access="field.camelcase-underscore"
type="AnsiString" length="10" not-null="true" />

  <property name="Town" access="field.camelcase-underscore"
type="AnsiString" length="50" not-null="true" />

  <property name="Country" access="field.camelcase-underscore"
type="AnsiString" length="50" not-null="true" />
</component>
```

值对象（例如 ReferencePerson）的列表如下：

```
<bag name="ReferencePersons" access="field.camelcase-underscore"
cascade="all">
  <key column="OrderId" />
```

```

<composite-element class="ReferencePerson">
  <property name="FirstName"
    access="field.camelcase-underscore" not-null="true"/>

  <property name="LastName"
    access="field.camelcase-underscore" not-null="true"/>
</composite-element>
</bag>

```

下面再举一个例子。前面的章节中曾提到过OrderLine应该是一个值对象。如果是这样，那么Order中有关OrderLine的映射就应该类似于上述代码。

但过后我们不难发现OrderLine可能有它自己的一个列表，例如注释列表。如果是这样，那么就有理由将OrderLine转换为实体。也就是说，由于技术和基础架构原因（而不是概念原因）将其转换为实体。我们必须注重实际。如果是这样，Order中的映射信息就更改为下面这样（而且OrderLine将具有它自己的映射信息）。

```

<bag name="OrderLines" access="field.camelcase_underscore"
  cascade="all">
  <key column="OrderId" />
  <one-to-many class="OrderLine" />
</bag>

```

对于本示例，我们还假设OrderLine有一个回指指向Order的字段（尽管前面曾提到保守地尝试使用双向关系）。因此，在OrderLine的XML文件中（此文件是在OrderLine被转换为实体时添加的），有一个如下的多对一片段。

```

<many-to-one
  name="Order"
  access="field.camelcase-underscore"
  class="Order"
  column="OrderId" />

```

这里的一个重点是需要自己实现双向性。因此在Order的AddOrderLine()方法中，代码如下。

```

//Order
public void AddOrderLine(OrderLine ol)
{
  _orderLines.Add(ol);
  ol.Order = this;
}

```

最后，这也意味着OrderLine将有自己的一个标识字段，无论此字段是否在领域模型中使用。如果填充了文件中缺少的部分，使用者代码将更有趣。

### 9.2.3 一个小的 API 示例

现在，我们来看一些小的代码片段，它们显示了当用NHibernate来执行一些创建、读取、更新、删除（CRUD）操作时，使用者代码的情况。CRUD通常是与持久化有关的应用程序工作的

主要部分。第一个代码片段是CRUD中的创建。

### 1. CRUD-C: 创建

要在数据库中创建一个新的实例（或行），只需实例化一个新的Customer，设置其属性，并调用所需的方法。当准备好保存它时，使用前面讨论的代码片段来重新连接一个ISession。然后，将实例关联到ISession，再调用ISession上的Flush()来存储所有更改。

总结一下，代码如下：

```
//A consumer
Customer c = new Customer();
c.Name = "Volvo";

_session.Reconnect();
_session.Save(c);
_session.Flush();
_session.Disconnect();
```

本例给出了具体说明，并通知ISession应该执行INSERT（因为调用了Save()）。也可以调用SaveOrUpdate()，这样NHibernate就自行决定是应该执行INSERT，还是执行UPDATE。在这种情况下所使用的信息是标识字段的值，并将它与在映射文件中为unsaved-value指定的值进行比较（对于Guid来说是00000000-0000-0000-0000-000000000000）。如果标识字段与unsaved-value匹配，则执行INSERT；否则，就执行UPDATE。

还应指出的一个重点是，INSERT被延迟了，当调用Save()时不会执行它，而当调用Flush()时才会执行。原因在于，这样能够在使用者中做出很多更改，然后尽可能在最后一块保存所有更改。

如何判断是否成功地写入了数据库？最简单的检查方式是读回实例，我们来试一下。

### 2. CRUD-R (One): 读取一个实例

第二个示例是通过标识字段读取一个实例（数据库中的一行）。从现在开始，假设使用一般的重新连接/断开连接代码片段，并只集中关注特定的代码。

像通常那样，我们与ISession实例进行对话。调用Load()/Get()，并给定正在查找的类型及其标识字段值。代码如下：

```
Customer c = (Customer)_session.Load(typeof(Customer), theId);
```

但要注意的是，如果在上次调用Flush()以来未关闭ISession，那么这不一定证明任何事情，因为当调用Load()/Get()时，ISession将提供来自标识映射[Fowler PoEAA]的实例，而不会到数据库中取回实例。强制跳转到数据库（而不是打开一个新的ISession）的方法是调用实例上的Evict()，并说明不再希望标识映射保持对实例的跟踪。代码如下：

```
_session.Evict(customer);
```

这是一种好方法，但在不知道标识值的情况下该怎么办？或许我们只知道客户名称的一部分，而且想获取与该名称模式匹配的所有实例。这就将我们引入了下一个示例：读取多个实例。

### 3. CRUD-R (Many): 读取多个实例

NHibernate中有两种具体的查询语言。但本例将使用Hibernate查询语言(HQL)。它非常类似于SQL,但它们是不同的。为了获取名称以“Vol”开头的所有客户的订单,可以编写以下代码。

```
//A consumer
string hql = "select from Customer where Name like 'Vol%'";
IList result = _session.CreateQuery(hql).List();
```

这样,在结果中,将得到一个名称以“Vol”开头的列表,正符合我们的要求。

通常,我们不会编写这样的静态代码,而会使用参数化方法。但再次强调,这里只是显示非常简单的代码,以便使你能够快速理解它是如何工作的。

### 4. CRUD-U: 更新

我们找到了很多名称以“Vol”开头的实例。需要对其中的一个实例进行一些修改。我们对该实例的属性进行修改,现在已经准备好了保存这些修改。再次强调,可以使用SaveOrUpdate(),但我们知道在这里使用UPDATE是有问题的,因此使用Update(),代码如下(继续前面的代码片段)。

```
Customer c2 = (Customer) result[0];
c2.Name = "Ford";
_session.Update(c2); //Can be skipped
_session.Flush();
```

事实上,由于我们是通过读取找到实例的,因此根本不必调用Update()。实例已经与ISession关联,因此在调用Flush()时,已经更新了它。

### 5. CRUD-D: 删除

在找到的实例时,有一个实例是错误的,因此应该删除它。删除操作可以通过以下代码完成。

```
//A consumer
Customer c3 = (Customer) result[1];
_session.Delete(c3);
_session.Flush();
```

你可能已经猜到,Delete()被延迟到调用Flush()时才执行。

删除操作很直观,是吧?

## 9.2.4 事务

在离开API示例之前,还应了解一件重要的事情,即当使用NHibernate时应如何控制事务,因为事务的使用量非常大。

如果你习惯于其他的手工事务控制方式,那么也将很容易理解如何通过NHibernate来控制事务。要做的事情就是抓取一个ITransaction实例,然后用它来执行Commit()或Rollback()(Commit()将默认地自动执行Flush())。完整的代码片段如下。

```
//A consumer
ITransaction tx = _session.BeginTransaction();
try
```

```

{
    tx.Commit();
}
catch (Exception ex)
{
    tx.Rollback();
    ...
}

```

**说明** 以上代码片段中使用了异常处理，因为在这里使用Rollback()是很有意义的。当然，其他代码片段也应该使用异常处理，但在本书中，异常处理主要是一个分散注意力的事物。

这个简短的介绍暂告一段落。有关如何起步的更多信息，可参见NHibernate站点[NHibernate]。下面继续介绍选定的持久化框架示例（它具有第8章中呈现的结构）。首先，看一下总体需求。

## 9.3 持久化基础架构的需求

如前所述，在进入细节之前，有必要讨论一下第8章中定义的总体需求。我们来考虑一下PI级别、支持的生命周期以及数据库处理方面的需求。

### 9.3.1 高级持久化透明

NHibernate的编程模型并未远离运行时PI，特别是在领域模型本身方面。但这也伴随着一些代价。例如，NHibernate对逆向属性管理（inverse property management）不起任何作用，而我们需要通过逆向属性管理来实现双向关系。相反，我们完全要靠自己来解决。

另外一个代价是，NHibernate无法利用IsDirty或类似的属性来通知我们实例是否是脏的。（在保存时，NHibernate将确定应该保存哪些实例，但这与刚才谈及的脏实例的通知无关。）

这只是高级PI的代价的两个示例。另一个代价是性能，因此较高的PI可能会导致性能问题。

为了使NHibernate能够保存领域模型类，还需要对这些类进行一些处理。一些典型的示例如下。

- 字段不能使用readonly属性。

如果在类中需要一个只读属性，最显而易见的方式是使用readonly关键字，但这样NHibernate就无法通过反射来设置字段。因此，必须将其转换为私有字段加公共get属性。

- 所有类都必须有默认构造方法。

即使领域模型中不需要构造方法，也必须有一个，它可以不带任何参数。构造方法可以是内部的，甚至是私有的，但仍然需要添加它。

- 应避免强类型集合。

不管怎样，我往往不认为强类型集合是一件非常糟糕的事情。举例来说，我喜欢使用IList，而不是CustomerList类。当然，泛型也是很好的方式。

（遗憾的是，在本书编写时，NHibernate尚未提供很好的泛型支持，因此，在很多地方会

遇到一些障碍)。

□ 不要设置标识字段。

通常，我们会为标识字段提供默认值，例如`Guid.NewGuid()`，但当使用NHibernate时却不要这样做，因为NHibernate使用标识字段值来确定应该使用UPDATE还是INSERT。如果提供了默认值（除非将它设置为`unsaved-value`），那么在未明确提供指南的情况下，NHibernate将总执行UPDATE。（也可以用`version`标签来指示INSERT/UPDATE。）

虽然这并不是一个特别严重的问题，但仍然有一些方面会降低PI级别。坦白地讲，我不知道在使用`readonly`声明时应该如何避免问题，因此我猜想在其他解决方案中，情况也不会更好，除非这些解决方案对代码进行了控制。

### 9.3.2 持久化实体的生命周期所需的特定特性

第5章中讨论了领域模型实例需要支持的简单生命周期（第8章中又重复了此讨论）。这里再次重复一下，参见表9-1。

表9-1 领域模型实例的生命周期语义总结

操 作	结 果（短暂的/持久的）
新调用	→ 短暂的
<code>Repository.Add(instance)</code> 或 <code>persistentInstance.Add(instance)</code>	→ 在领域模型中是持久的
<code>x.PersistAll()</code>	→ 在数据库中是持久的
<code>Repository.Get()</code>	→ 在领域模型（和数据库）中是持久的
<code>Repository.Delete(instance)</code>	→ 短暂的（而且在调用 <code>x.PersistAll</code> 时将从数据库删除实例）

这很容易被映射到NHibernate，因此也很容易支持它。我们添加一个列，以便将表9-1映射到NHibernate（见表9-2）。

表9-2 领域模型实例的生命周期语义总结（添加了NHibernate）

操 作	NHibernate	结 果（短暂的/持久的）
新调用	新调用	→ 短暂的
<code>Repository.Add(instance)</code> 或 <code>persistentInstance.Add(instance)</code>	<code>ISession.Save()</code> , <code>Update()</code> , <code>SaveOrUpdate()</code> , 或与持久实例关联	→ 在领域模型中是持久的
<code>x.PersistAll()</code>	<code>ISession.Flush()</code>	→ 在数据库中是持久的
<code>Repository.Get()</code>	<code>ISession.Load()</code> , <code>Get()</code> or <code>List()</code>	→ 在领域模型（和数据库）中是持久的
	<code>ISession.Evict()</code> , <code>Clear()</code> or <code>Close()</code>	→ 短暂的（但不会被删除）；不管标识映射/操作单元
<code>Repository.Delete(instance)</code>	<code>ISession.Delete()</code>	→ 短暂的（而且在调用 <code>x.PersistAll</code> 时将从数据库删除实例）

如表9-2所示，添加了一个额外的行，它讲的是如何释放一个实例（或所有实例）的标识映

射/操作单元。这并不是我预想的，但在实际中确实遇到了需要它的情况，例如当在会话之间“移动”实例的时候。

现在暂不讨论有关它的更多内容。

---

**说明** 当然，有大量的细节，但这里讨论的是总览视图。

---

我知道，你认为在第5章的简单生命周期一节中，有关NHibernate的考虑得太多了。也许是这样的，但你会发现大多数其他的O/R映射工具也支持这样做。

### 9.3.3 谨慎处理关系数据库

最后，O/R映射工具需要谨慎处理关系数据库。也就是说，它应该采用与手工编程相同的方式或至少类似方式来处理数据库，并加以特别的注意。

如果说前一项需求（生命周期）是“容易的”，那么这项需求就难得多了，而且在不同的O/R映射工具之间会有很大的变化。

在NHibernate于特定情况下如何处理数据库方面，我们很容易挑出毛病，并认为手工处理会做得更好。本书反复强调，对每个数据库调用都进行优化是没有意义的，因为大多数情况下这样做并不重要。然而，我们总是可以进入到代码中，并在真正需要的地方进行优化。设定这项需求的原因仍然是只想做必要的优化，而且当进行优化时，除了那些绝对必要的SQL以外，我不想使用其他的普通SQL。

---

**说明** 我不希望你把这理解成“不惜任何代码避免SQL”或“仅使用功能非常丰富的O/R映射工具”。

SQL是一个伟大的工具，我喜欢在最合适的情况下使用它。

---

那么关于NHibernate，有哪些需要注意的问题区域？举例来说，一个明显的问题区域就是当将处理工作从数据库移出时，我们避免将逻辑留在存储过程中（实际上也一起避免存储过程）。这是有意而为之的，因为我们希望通过将所有逻辑都放到领域模型及其周围环境中而获得更好的可维护性。（再次强调，当真正需要时，可以对存储过程中的少数地方进行优化。）

另一个典型的问题区域是基于集合的处理工作，例如增加某个特定类的所有实例的列值（例如，将所有产品的Price设置为 $\text{Price} = \text{Price} * 1.1$ ）。用SQL执行此操作是很常见的，也是推荐的。使用NHibernate时，集合是一个关于如何根据数据库进行处理的单行集合。这也可能是一件好事。因为验证规则通常是基于实例的，而且我们也需要为大规模的更新来处理它们。但是，代价可能是相当高的。

第三个问题是领域模型的使用者可能根本不关心数据库。使用者可能在后端创建了一个巨大的负载，而没有认识到这一点，或实际上几乎没有能力认识到。这是领域模型所提供的良好隔离和抽象的一个缺点。

第四个问题是在可选更新中，如果满足特定规则，例如：



```
UPDATE x SET y = 42 WHERE y >= 1
```

那么，这必须通过预读操作来处理。

第五个问题是当需要读取很多复杂类型时（即使实际读取的信息量很少），存在“读取方式过多”问题。如果在某种程度上未使用仔细的聚合设计，而且未使用延迟加载，那么这个问题尤其常见。

然而，还有另一个典型问题，即n+1选择问题。假设获取数据库中的所有Order，而且OrderLine是延迟加载的。这样就涉及了每个Order及其OrderLines。然后就会为每个Order的OrderLine发出一条新的SELECT。如果不使用延迟加载，那么问题会很小，但这样就面临着在某些场景下获取的数据量过大的风险。

所有这些示例均不是特定于NHibernate的，但它们对于O/R映射工具都很常见。与NHibernate更相关的一个问题的例子是IDENTITY或序列。使用这样一个标识字段可能导致未能如期过早地执行了数据库的INSERT操作。

总之，我的观点是NHibernate很好地满足了这些总体需求，特别是当我们能够接受O/R映射工具的典型工作性质和所涉及的折中的时候。

## 9.4 分类

上面介绍了NHibernate，以及它是如何处理需求的。现在我们做好进一步研究NHibernate的准备了，这里我们借助于第8章中所使用的分类。但这里不会以抽象方式来讨论各种选择，而是将NHibernate作为每个分类的各种选择的一个示例，还将更详细地讨论解决方案的形式。

首先是NHibernate希望我们使用的领域模型风格。我们已经接触过这个话题，但这里再多讲几句。

### 9.4.1 领域模型风格

第8章中曾讲过领域模型风格有三个方面是O/R映射工具最常使用的。

- ☐ PI
- ☐ 继承
- ☐ 接口实现

本例中的选择是很简单的。我认为，NHibernate提供了高级别的PI。我们无需从特定的基类继承，而且不必实现特定的接口，甚至不必在领域模型中引用任何NHibernate组合。

### 9.4.2 映射工具风格

基于元数据的映射通常有两种不同的映射工具风格。

- ☐ 代码生成
- ☐ 框架/反射

同样，定位NHibernate是很容易的，因为它是一个框架/反射解决方案的典型示例。当然，NHibernate中有一些代码生成支持，但这些支持通常用于下面这种情况：通过查看领域模型、

元数据或数据库模式这三个部分当中的一个（或两个），来生成其中的一个部分。例如，假设有元数据，那么从元数据可以创建数据库模式。这并不属于上面提到的映射工具风格的代码生成。映射工具风格是根据在运行时如何进行映射而定义的，鉴于此，NHibernate未使用代码生成。

此规则的一个例外是为支持延迟加载[Fowler PoEAA]而执行的那些操作。举例来说，在运行时，NHibernate将用列表类来交换一些代理。为了实现这一点，必须将列表作为一个接口（而不是作为具体类）公开。

**说明** 注意，本章前面并未提及将接口的使用作为降低PI级别的一种方式。做出这个决定的原因是，延迟加载是个特性，而不是必须使用的功能。

对于此问题有各种各样的观点，但我认为，延迟加载是优化技术，而不是必须使用的功能。

NHibernate使用框架/反射方法是好是坏？实际上对于O/R映射工具来说是好的。

### 9.4.3 起点

当使用一个特定的O/R映射工具时，通常可以从以下列表中选择一个或多个起点。

- 数据库模式
- 领域模型
- 映射信息

NHibernate允许从任何起点开始，而且它并不是随意的选择，而是通过设计来支持。也就是说，如果必须从不允许更改的遗留数据库模式开始，那么关注一下诸如iBATIS[iBATIS]此类的框架是一个好的思想。

### 9.4.4 API 焦点

API焦点有两个来源。

- 表
- 领域模型

但实际上这并不适用于最常见的O/R映射工具，因为它们的主要目的是提供领域模型视图，同时使用关系数据库来提供持久化功能。因此这里的答案很简单。NHibernate使用领域模型作为API焦点。

### 9.4.5 查询语言风格

查询语言是个有趣的话题。不同解决方案在查询能力方面有着非常大的区别。第8章中定义的子类别包括

- 基于字符串的SQL风格
- 基于查询对象

NHibernate同时具有这两种风格的实现，分别称为HQL和Criteria对象。前面在搜索名称以

“Vol”开头的所有Customer时，已经给出了HQL查询的一个例子。如果用Criteria对象表达同样的事情，代码可能像下面这样：

```
//A consumer
IList result = _session.CreateCriteria(typeof(Customer))
    .Add(Expression.Like("Name", "Vol%"))
    .List();
```

上面这段代码也是一个连锁方法调用的例子。它的工作原理是，Add()返回一个仅作为CreateCriteria()的ICriteria。这使得标准查询的使用更具有可读性。

**说明** 本章只是简单介绍了NHibernate是如何完成不同操作的，但当涉及查询这个庞大的话题时，简单介绍就不够明显了。更多信息参见[Bauer/King HiA]。

第8章中还曾提到将SQL作为最后的手段来解决性能问题是一个好的思想，这对于NHibernate同样适用。NHibernate同时支持这两种查询风格，因此既可以取回实体，也可以进行原始连接，并执行任何所需的操作。

NHibernate还提供了介于普通查询机制和原始SQL之间的查询，它们称为报告查询[有时也称为变平查询(flattening query)]。代码如下：

```
//A consumer
string hql = "select new CustomerSnapshot(c.Id, c.Name) " +
    "from Customer c";
IList result = _session.CreateQuery(hql).List();
```

因此，这里的结果不是Customer实例，而是CustomerSnapshot实例。为了让它工作，需要在值对象（前例中的CustomerSnapshot）上提供一个匹配的构造方法，并通过如下导入指令引用映射文件中的CustomerSnapshot。

```
<import class="CustomerSnapshot" />
```

注意，根据NHibernate，CustomerSnapshot只保持平面数据（通常是只读的）。标识映射中不会跟踪标识，而且操作单元中不会对更改进行跟踪。

查询实际上也是下一个分类的核心内容。

#### 9.4.6 高级数据库支持

在第8章中的8.4.6节列举了以下示例（对于数据库人员而言可能谈不上是高级支持，但对于对象人员来说是高级的）。

- 聚合
- 排序
- 分组
- 标量查询

NHibernate中的HQL和Criteria对象均支持这4项操作。我们为每个操作举一个例子，从聚合

查询开始（查找Customer的实例个数）。在HQL中，代码如下：

```
//A consumer
string hql = "select count(*) from Customer";
int numberOfCustomers =
    (int)_session.CreateQuery(hql).UniqueResult();
```

**说明** 同样，注意HQL中使用的不是表名称，而是类名称。

第二个示例是对数据库中的结果集进行排序，这样在领域模型中就不用排序了（通常这是一个优点），HQL中的代码如下所示。在本例中，我们按Name对所有customers的获取操作进行排序。

```
select from Customer order by Name
```

分组主要用于报告的目的，但它也有其他用途。这里，按Name进行分组，并计算每个名称的实例个数（有些复杂，但它显示了语法）。

```
//A consumer
string hql =
    "select c.Name, count(*) from Customer c group by c.Name";
IList result = _session.CreateQuery(hql).List();
```

最后，标量查询的一个例子是只获取名称以“Vol”开头（这是我特别喜欢使用的查询标准）的Customer的Name，但事实上，这只是执行前面讨论的分组查询的另一个示例。为了获取结果中的值，名为result的IList包含object数组的一个列表。下面是继续上面示例的代码片段，它列出每个名称的名称和实例个数：

```
//A consumer, continuing from previous snippet
foreach (object[] o in result)
{
    Console.WriteLine("{0} {1}", (string)o[0], (int)o[1]);
}
```

标量查询对于以下情况特别有用：获取完整Customer实例的代价太高，因而只想获取一个更轻量级的结果，而且不想定义一个类。

当然，在这种情况下结果不是类型安全的，但我们可以使用一个变体，即本章前面讨论的报告查询（或变平查询）。

一种替代方法是提供映射变体，但这样做会有更大的问题。这显然违反了聚合原则，因为这允许在没有强制加载完整聚合的情况下更新实例。我们应注意这个问题。

下面给出一行代码作为本节的结束，对于具有对象背景的人员来说，此代码可能是“显然”的，但对于SQL人员来说，可能就有些神秘了。

```
select o.Customer.Name, count(*) from Order o group by o.Customer.Name
```

查询本身非常简单，但注意数据是从两个表获取的，而且并未明确指定是通过from子句（即，没有连接）获取的。当然，映射信息用于确定HQL查询在执行时都执行哪些操作。

### 9.4.7 其他功能

最后，看一下NHibernate的其他功能的组合。

- 支持哪些后端

如果列出所支持的后端，那么它可能变成一个冗长而乏味的特性列举，因此这里不讨论细节。简单地说，NHibernate支持几种不同的数据库（支持大多数主流的商业和开源数据库）。有关最新的信息，可访问站点[NHibernate]。如果漏掉了需要使用的数据库，那么也可以自己编写插件。

- 不仅是O/R映射工具

NHibernate主要是个O/R映射工具。它的原则是只做一件事，并做好它。

---

**说明** 当然，NHibernate还具有其他功能，但这些功能都不是重点。它们并不多，也不大。

---

- 高级缓存

与Hibernate相比，NHibernate的高级缓存还是一个不太成熟的领域，但这种情况很快就会有所改观。另一方面，它对我来说往往并不重要。二级缓存有很多问题，因此我通常不考虑该选项。

- 事务控制的高级支持

NHibernate提供了很好的手工事务控制支持。在有些情况下，如果不注意，可能会发生意外（例如，当在一个事务中间执行查询时，NHibernate将在执行查询之前，通过执行Flush()把更改保存到数据库）。这是两种选择之间的一个折中问题：一种选择是不通过缓存进行查询，另一种选择是只使用常规的事务控制。好消息是可以通过自己编写代码来控制它。与第8章的讨论结合起来，NHibernate中典型的事务处理方式是“手工控制”。

---

**说明** 除此之外，也可以构建一种使用第8章中提到的任何自动方法的功能。

---

- 开放源码

如前所述，NHibernate是一个开源项目，因此可以对源代码进行分析，甚至可以构建自己的分支代码。

- 版本（第几代）

现在很难确定NHibernate处于哪个版本。在本书编写时，构建它的团队将其命名为版本1，但同时，我认为其质量要比典型的第一个版本高。NHibernate也是Hibernate的一个端口，而Hibernate已经是第二代了，或者实际上是第三代，但这在NHibernate中并未反映出来。不管怎样，我们还是将它称为第一代。

下面我们从一个完全不同的角度（即基础架构模式角度）来看一下NHibernate在另一种分类中的定位。

## 9.5 另一种分类：基础架构模式

简单起见，我们从元数据类型开始。

### 9.5.1 元数据映射：元数据类型

第8章中为元数据映射模式[Fowler PoEAA]定义了三种不同的元数据类型。

- XML文档或其他文档格式
- 属性
- 源代码

NHibernate的这个分类是很简单的，因为其元数据通常是XML文档。在实际工作中，也可以用源代码来描述元数据，但这种方法不是主流。

也有一些基于属性的方法，但它们也是以XML文档的格式来生成元数据。

### 9.5.2 标识字段

前面已经提到在NHibernate中处理标识字段的方式，但这里再深入讨论一下。前面显示了如何映射Guid类型的标识字段。下面看一下Product实体，它有一个用于标识字段的int，它的值是在数据库中用IDENTITY（自动增加的列）命令生成的。代码如下：

```
<id name="Id" access="field.camelcase-underscore"
    unsaved-value="0" >
  <generator class="identity" />
</id>
```

如第8章所述，当有新实体时，标识字段的值甚至可以在4个不同的层上生成。

- 使用者
- 领域模型
- 数据库
- O/R映射工具

如果值是Guid类型的，那么可以在所有层上生成它，但推荐在O/R映射工具层上生成。在上面显示的IDENTITY示例中，值是在数据库层上生成的，这实际上对编程模型有很大影响。首先，有一个比我们预期更早的数据库跳转。其次，可能有一个INSERT仅仅抽取了IDENTITY值。这两个例子都是有问题的，因此在不必要的情况下应避免使用IDENTITY。

#### NHibernate支持COMB

事实上，Guid通常是一个很好的解决方案。但或许有些读者读过我的一篇文章，它讨论了当使用Guid作为主键时，INSERT的一个可能的代价[Nilsson COMB]。问题在于，对于使用Guid作为主键的那些表来说，INSERT的吞吐量可能比使用int时的吞吐量低很多倍。一般情况下，这并不成为问题，但当表非常大而且INSERT的加载开销很高时，这个问题就会凸现出来。我在文章中建议了一个称为“COMB”的解决方案。从根本上说，它是一个Guid，

但它是顺序类型的。我很惊讶地看到NHibernate在一个特定的alpha版本中添加了COMB支持。为了使用COMB生成器，元数据如下（唯一的区别是生成器是guid.comb，而不只是Guid）：

```
<id name="Id" access="field.camelcase-underscore"
    unsaved-value="00000000-0000-0000-0000-000000000000" >
  <generator class="guid.comb" />
</id>
```

最简单的方法是让NHibernate生成Guid，但这当然并不总是可行的解决方案。有时，必须使用指定的标识字段，这使得操作更难以处理。例如，无法通过查看unsaved-value来确定NHibernate是使用UPDATE，还是使用INSERT。相反，我们一般通过显式地调用Save()或Update()（而不是SaveOrUpdate()）来通知NHibernate执行什么操作。

**说明** 正如本章前面所述，可以使用version标签加上unsaved-value标签来帮助设置标识字段值。

这些只是标识字段策略的几个示例。NHibernate支持更多。我们也很容易插入自己的策略。

### 9.5.3 外键映射

映射文件中的外键映射代码可能像下面这样。这里（在Order.hbm.xml文件中）描述的是一个Order有一个Customer。

```
<many-to-one name="Customer" access="field.camelcase-underscore"
    class="Customer" column="CustomerId" not-null="true" />
```

简单地讲，NHibernate为很多关系类型提供了健壮的支持，这可能解决大部分相关需求。但是，要想有效地使用所有变体却并不容易，但几个基本变体的使用还是很简单。

另一方面，在NHibernate中使用嵌入值模式[Fowler PoEAA]还是非常容易的。

### 9.5.4 嵌入值

NHibernate使用component（看似它不具有太多不同意义）来描述嵌入值。我们再来重复一下在映射文件中如何描述具有Address的Customer。

```
<component name="Address" access="field">
  <property name="Street" access="field.camelcase-underscore"
    type="AnsiString" length="50" not-null="true" />

  <property name="PostalCode"
    access="field.camelcase-underscore"
    type="AnsiString" length="10" not-null="true" />

  <property name="Town" access="field.camelcase-underscore"
    type="AnsiString" length="50" not-null="true" />
```

```
<property name="Country" access="field.camelcase-underscore"
  type="AnsiString" length="50" not-null="true" />
</component>
```

注意Customers表包含所有列，但Customer类只有一个Address字段。

这一步完成之后，使用来自使用者的嵌入值就很好理解了。举例来说，代码如下：

```
Console.WriteLine(aCustomer.Address.Street);
```

然而，要注意值对象Address是否应该是不可变的。在这个特殊示例中，我选择它是不可变的，但这并不总是一个简单的选择。

如果决定使用可变类型，可以编写以下代码，而不必实例化新的Address。

```
aCustomer.Address.Street = "Large Street 42";
```

**说明** 如果有一个需要在很多其他类中使用的嵌入值，那么实现IUserType是一种很好的方法，这样就不必反复地描述映射了。如果存储类型与要公开的类型不同，也可以使用这种方法，因为可以通过IUserType实现完成转换。

这样，一个不利因素是领域模型将必须引用nhibernate.dll，或者必须在单独的集合中编写特定代码，并继承基本的领域模型类。

应该尽可能避免此问题。当可以使用常规的嵌入值方法时，那么一切良好。对于复杂情况来说，通常可以在属性get/set与私有字段之间进行手工转换，然后将这些私有字段映射到数据库。

避免特定结构或用自定义get/set代码处理它的一个典型示例是应用类型安全的枚举模式[Bloch Effective Java]（例如，用于为枚举添加行为）。首先，要慎重考虑是否真正不能使用常规的C#enum。第二种选择是自定义get/set代码，这样就不必处理IUserType了。但这感觉有些小题大做了。

在结束这段说明之前，再强调一点：在领域模型中，不要一味地避免对nhibernate.dll的引用。如果收益大于代价，就应该使用它。

## 9.5.5 继承解决方案

NHibernate支持全部三种不同类型的继承解决方案，即单表继承、类表继承和具体表继承[Fowler PoEAA]。如果假设Customer和Vendor从Company继承，那么映射信息如下。

```
<discriminator column="Type" type="AnsiString" length="8"
  not-null="true" />
```

这样，每个子类就需要以下代码，这段代码中描述了细节（即不属于Company类的映射信息）。

```
<subclass discriminator-value="Customer" name="Customer">
  <property name="CustomerNumber"
    access="field.camelcase-underscore"/>
```



```

</subclass>

<subclass discriminator-value="Vendor" name="Vendor">
  <property name="VendorNumber"
    access="field.camelcase-underscore"/>
</subclass>

```

从以上代码可以理解一件事情：在每种情况下唯一的变体是只使用一个单一的特定字段，CustomerNumber和VendorNumber。在本例中，只有一个Companies表，它的形式如下（这意味着这里使用了单表继承）。

```

create table Companies (
  Id UNIQUEIDENTIFIER not null,
  Type VARCHAR(8) not null,
  Version INT not null,
  Name VARCHAR(100) not null,
  Street VARCHAR(50) not null,
  PostalCode VARCHAR(10) not null,
  Town VARCHAR(50) not null,
  Country VARCHAR(50) not null,
  CustomerNumber INT null,
  VendorNumber INT null,
  primary key (Id)
)

```

这意味着对于Vendors来说，CustomerNumber将为NULL，反之亦然。Type列将具有值Customer或Vendor（或通常一些更小的符号）。

**说明** 我知道，有些读者对这个示例感到不解，因为他们首选使用一些类似于Party archetype这样的模式，而不是[Arlow/Neustadt Archetype Patterns]。或者至少不会这样过多地使用继承（如果唯一的变体只是单一字段的话）。

我的意图只是显示一个明显且清晰的示例。

而且可以预料得到，当继承映射就绪时，使用者代码可能会忘记继承层次结构实际上是如何存储的，反而将注意力集中在如何使用继承层次结构上。

### 9.5.6 标识映射

NHibernate使用一个ISession级别的标识映射。如果按照下面的步骤来操作，就很容易看到这一点：使用Load()来按Id读取实例，并直接在数据库中对行做出更改，然后再再次用Load()读取实例（具有相同的ISession实例）。我们看不到更改的原因是NHibernate在执行第一个Load()调用之后不会再返回数据库，相反，它从标识映射来抓取实例。

NHibernate使用标识映射的另外一种情况是，查询的目的不是为了查找实例，而是为了将标识放到映射中，以便支持未来从标识映射调用Load()。

如果想强制实例不使用标识映射，可以在ISession上调用Evict()，或者调用Clear()，

从所有实例清除标识映射，当然也可以调用`Close()`来关闭`ISession`。

### 9.5.7 操作单元

NHibernate使用操作单元模式[Fowler PoEAA]这一点不足为奇，同样，它是在`ISession`中使用它的。尽管如此，与最典型的实现相比，它的实现完全不同。NHibernate并不是向操作单元注册已发生的操作，而是在从数据库读取实例时创建它们的快照。在执行刷新操作时，`ISession`将它已知的实例与快照进行比较，以创建一个该时刻的操作单元。

清楚起见，需要注册删除操作，而且应该使用`SaveOrUpdate()`来注册实例的通知。

像通常一样，此解决方案也是利弊共存的，一个明显的缺点就是当需要检查很多实例时，只应该刷新其中的一个。这也是为什么Hibernate（是的，是Hibernate，而不是NHibernate，至少在本书编写时是这样）版本3更改了这一点的原因之一。

### 9.5.8 延迟加载/立即加载

如果NHibernate确实需要使用（自动的）延迟加载[Fowler PoEAA]，那么很容易实现。只需在元数据中像下面这样声明它即可，例如：

```
<bag name="OrderLines" access="field.camelcase-underscore" cascade="all" lazy="true">
  <key column="OrderId" />
  <one-to-many class="OrderLine" />
</bag>
```

现在，当首次需要`OrderLine`时，就可以延迟加载`Order`的`OrderLine`了。但是，为了做到这一点，必须有一个连接的`ISession`。

如果不在元数据中使用`lazy="true"`声明，那么就会有效地获得立即加载。NHibernate支持几种不同的立即加载方法，例如OUTER JOIN或几条按先后顺序执行的SELECT批处理语句。

重要的一点是，当使用几条SELECT语句进行立即加载时，可能导致读取的不一致性（在延迟加载中这也可能是一个较大的问题）。要避免此问题，可以使用显式的事务控制，并且将事务隔离级别提高到可序列化级别[Nilsson NED]，但它很少成为严重的问题，因此这里不讨论过多的细节。

也可以在代码中控制是否需要使用立即加载/延迟加载。在使用查询时可以进行这样的控制，本章中对查询的讨论已经很全面了，因此这里不再重复。

### 9.5.9 并发性控制

最后，我们需要支持并发性控制。Fowler[Fowler PoEAA]给出了以下解决方案。

- 粗粒度锁
- 隐式锁
- 乐观离线锁
- 悲观离线锁

如果在实体中添加<version>标签,则NHibernate使用乐观离线锁,代码如下。

```
<version name="Version" access="field.camelcase-underscore" />
```

这样,如果在保存时,数据库中的行有另外一个不符合我们预期的版本列的值,就说明其他人已经更新了该行,这就产生了并发性冲突(而且将得到一个StaleObjectStateException异常)。

其他三种机制需要我们自己处理。NHibernate未对它们提供开箱即用的支持。

### 9.5.10 额外功能: 验证挂钩

第8章中未讨论验证挂钩,但本章将它作为一个额外功能添加到这里。NHibernate有一个IValidatable接口,它供NHibernate在执行Flush()时调用。

但是,这样就需要在领域模型中引用nhibernate.dll,因此最好使用一个类似于第7章中使用的那个解决方案。也就是说,可以使用一个IInterceptor(它是另外一个NHibernate接口,但在本例中领域模型不使用它)实现来隐式地检查规则,从而使用所需的领域模型类接口。(当然,并不是必须要使用领域模型类的接口。)

为了使用一个自定义的IInterceptor实现,需要在实例化ISession时将该实现作为参数提供给ISession,代码如下。

```
//A consumer
_session = NHConfig.GetSessionFactory()
    .OpenSession(new MyCustomInterceptorImpl());
```

需要注意的是,当使用这种方法时,有些操作在验证代码中是不支持的。例如,不能使用同一个ISession实例进行调用,因为这会在某处导致问题,但我们已经在第7章中讨论了一些用于处理此问题的策略,因此这里不再重复。

## 9.6 NHibernate 和 DDD

我知道有些读者已经迫不及待地想要看到将整个NHibernate放到DDD上下文中的的一些示例。本章就以这部分内容作为结束。

同样,这里不会使用像NWorkspace这样的抽象层,而且这里描述的内容也适用于其他O/R映射工具(如果使用抽象层,问题会更简单)。

首先,简单讨论一下如何创建程序集。

### 9.6.1 程序集概览

从概念上讲,我倾向于将存储库作为领域模型的一部分,但当在O/R映射工具上未使用抽象层时,应将存储库放在一个单独的程序集中。这样,就得到了如图9-2所示的依赖关系。

注意,领域模型现在对NHibernate没有依赖关系。同样,领域模型可以在需要的时候使用存储库。一种方法是存储库实现领域模型中的接口。然后,这些接口实现(存储库)在适当的点被注入到领域模型实例中。

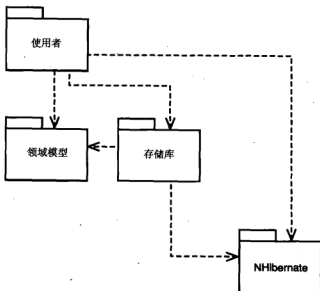


图9-2 典型的程序集依赖关系

### 9.6.2 ISession 和存储库

为了使用此方法，假设使用者将ISession实例注入到存储库中。这意味着如果存储库可以处理一部分工作的话，那么以下这行代码

```
Customer c = (Customer)_session.Load(typeof(Customer), theId);
```

可以被转换为下面的代码（在这个简单示例中，要处理的工作并不多）。

```
Customer c = customerRepository.GetCustomer(theId);
```

以上代码片段假设customerRepository是通过以下方式实例化的。

```
//A consumer
ICustomerRepository customerRepository =
    new CustomerRepository(_session);
```

因此，从此时起，存储库就有了一个ISession，可用它与后端进行对话。

### 9.6.3 ISession、存储库和事务

最后，我倾向于在使用者中（也就是在存储库的外部）控制操作单元的开始和结束。以下代码片段解释了它的形式（这是非常紧凑的结构，并且不是推荐的事务代码）。

```
//A consumer
customerRepository.Add(customer);
orderRepository.Add(order);
```

```
_session.BeginTransaction().Commit();
```

在这段代码中，两个存储库共享同一个ISession。

### 9.6.4 得到了什么结果

让我们后退一步。前面认为使用O/R映射工具是理所当然的，并讨论了它是如何适合存储库的概念的。

现在换一种思考方式。我的意思是，我们有一个存储库，现在想用实际代码填充它。假设我们决定编写手工代码。存储库使用者的API可能与其他实现相同，但在不使用以下代码行的情况下。

```
Customer c = (Customer)_session.Load(typeof(Customer), theId);
```

可能需要编写大量代码。例如，需要定义SELECT语句，以便获得重建customer实例所需的所有列。

然后需要实例化customer实例，并将resultset行的所有值传送给实例。

当然，我们可以完成这些操作，这个问题并不复杂，但它通常并不是我们感兴趣的工作。而且在这方面还有很多问题需要考虑。例如，customer还有一个需要同时重建的Reference-Persons列表，而且还需要执行其他一系列操作。

在代码完成之后，对模式或领域模型的每项修改都将导致繁重的工作。记住，这是单一对象的单个获取操作。

如果使用O/R映射工具，对更改的处理一般通过对映射信息做出少量修改就可以了。当无需对不感兴趣的持久化代码进行手工编码时，处理更改的效率会高得多。

我们可以从一个更大的角度来思考这个问题，因为如果知道修改的代价不会很高，那么我们更愿意进行重构（与代价高昂的情况相比）。

让我们以一个小故事来结束本节，它来自一个最近的项目。在此项目中，两种处理更改的方法的代价可能相差一个数量级。第一种方法是用O/R映射工具来处理更改，它只影响一部分（即领域模型和数据库模式），第二种方法是手工处理，它影响持久化代码，这两种方法相比，前者可能便宜一个数量级！（尽管如此，但不要忘记O/R映射工具当然不是银弹，它也是利弊共存的。）

## 9.7 小结

通过应用O/R映射工具，例如NHibernate，可以严格地遵守并稳妥推进第4章中开始的设计和代码。使用NHibernate后，我们能够PI方式构建自己的领域模型。

我将这个工具家族（O/R映射工具）视为助推器，它们使得集中关注DDD成为可能，但同时也需要提供一些方法，以便在必要时手工处理部分持久化代码。

当然，这并不只是金色森林和绿色森林的问题。例如，我非常希望能够从O/R映射工具（例如NHibernate）中看到DDD的应用，这将简化整个工作，并得到更好的最终结果。同样，将

NHibernate作为基础架构的一部分来应用是在正确方向上迈出的一大步。

那么，下一步是什么？它就是本书的最后一部分：第四部分。

第四部分的第一章从更大的上下文讨论开始，其中有多模型同时运行。然后关注一些对于现在和未来有趣的设计技术，包括面向服务架构（SOA）、控制反转/依赖注入（IoC/DI）和面向方面编程（AOP）。



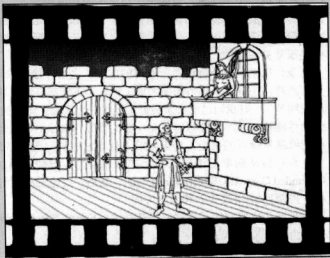
# Part 4

## 第四部分

### 下一步骤

这一部分主要关注并开始使用其他一些设计技术。另外一个重点是如何在领域模型中处理表示层，以跨越领域模型与表示层之间的鸿沟，此外还介绍如何处理开发人员的 UI 测试。这一部分几乎都是邀请其他作者编写的。

这一部分主要讨论其他技术。但这并不是暗示着“影院比戏院好”，本部分只是介绍不同的技术。



#### 本部分内容

- 第 10 章 博采其他设计技术
- 第 11 章 关注 UI

我们现在讨论到哪里了？

到目前为止，我们已经花费了大量时间来讨论和应用DDD的基础知识。我们编写了很多单元测试，认真设计了领域模型，并通过O/R映射工具完成了领域模型的持久化。那么还有哪些内容未讨论到？

这里讲一个小故事，讲的是一位著名的业内领导者，他就任于一家跨国公司。公司在利润、质量等方面又上新台阶后，他召开了一次会议，在会上说：“现在，所有人都用一分钟时间来想一下我们的成就……”一分钟之后他又重新开始，列举了问题，做了批评，如此这般。这在企业中是很常见的做法。

我们也来花一分钟思考所取得的成绩，之后，接下来需要关注什么？

当然，还有很多事情需要考虑。首先，我们甚至连领域模型的基础知识尚未讨论完整，更不用说细节了。但我们已经有了良好的开端，而且做了充分的准备工作，可以着手解决很多问题了，包括性能特征、并发性问题、安全性，等等。实际上，这些问题将留给读者作为练习，因为我希望能够激发读者的兴趣，如果以前未处理过这些问题，那么一定要尝试它。

本章将从简单讨论如何向上扩展目前为止已使用的DDD原则开始，主要关注限界上下文(Bounded Context)模式[Evans DDD]。然后，讨论三种值得一提的不同设计技术，即面向服务架构、控制反转/依赖注入和面向方面。

我请几位朋友编写了这些设计技术，下面就是他们的贡稿，但我自己撰写了第一节，即“上下文为王”。

## 10.1 上下文为王

忽视上下文的重要性是一种常见现象。例如，当开发人员在事先没有经验的情况下讨论和使用GoF模式[GoF Design Patterns]时，通常会忘记模式的适用性是取决于上下文的。

无论是好的思想、解决方案还是其他任何事物，如果放到错误的上下文中，那么它只能成为一个错误的部分。事实就是如此。

在讨论更多与特定上下文有关的概念之前，我们先后退一步。首先，简单讨论一下层和分区。

### 10.1.1 层和分区

早在1994年，Grady Booch就出版了*Object-Oriented Analysis and Design*（《面向对象分析与设



计》[Booch OOAD]一书。它使我对几年前从Jim Rumbaugh的书*Object Modeling Technique*（《对象建模技术》）[Rumbaugh OMT]中学到的知识有了更好理解。一个例子就是层和分区。Booch描述的层概念是我很熟悉的，但他描述的分区概念就不同了。他将分区看作是层相对的其他维度的切片。

举个例子来具体说明一下，UI和领域模型是两个典型的层。两个可能的分区则是用于注册订单的子系统和管理用户投诉的子系统。这两个分区都可能有一个UI和一个领域模型。

无论是过去还是现在，分层都得到了比分区多得多的关注。分层是提供“关注点分离”的一种非常重要的方式。单一职责原则（SRP）[Martin PPP]可能是人们根深蒂固的一种思想。

但在某种程度上，这种完全将注意力放在分层上的做法正在改变。DDD对分层的要求宽松多了，至少与过去所使用的严格分层相比是这样的[Nilsson NED]。（是的，我们仍然非常重视将基础架构从领域模型中分离出来。）

---

说明 第4章中已经讨论过根据DDD进行分层。

---

同时，人们也越来越认识到不能创建一个单独的、巨大的领域模型，至少在大型项目中这样做往往过于复杂，而且代价高昂，或者效率低下。此外，面向服务（SO）也成为人们对分区感兴趣的一种推动力量。整个问题变得更明显了。

由于人们对分区的兴趣不断增加，对分层的兴趣可能就减少了。对于一个小的分区来说，严格的分层就不再重要了，也不会提供很多益处。

让我们更仔细地看一下分区的原因。

### 10.1.2 分区的原因

虽然分层通常是基于技术原因，但分区往往是基于组织原因。当然，分区也有可能是因为技术原因，例如，某个特定分区可能很快就变得过时，而且可能在后面需要替换为另一个实现。这样，思路就是使得替换工作明显且易于完成。

同样，组织也是分区的重要原因，例如某个团队负责这个分区，而另一支团队负责另一个分区。这与我同事之间关于分层动机的讨论是完全一致的。他们都不认为分层是用于组织团队的好方法。分区也是我经常用来成功组织团队的方法。所有这些都意味着这样一个事实：围绕分区来组织团队是一种更好的思想。这样，在每个分区中，对分层的重视就小多了。当然，我们知道不同的人具有不同的兴趣和专业特长，而且我们不应该反对这些。不应该只是简单地建立三个大的部门，即UI开发人员部门（负责为所有应用程序构建UI）、领域模型开发人员部门和数据开发人员部门。除了分区和分层这两个维度之外（如果必须选择的话），最重要的是应该解决的领域问题，而不是要使用什么技术。在组织团队时也应该记住这一点。

是否分区的另一个考虑因素是开发工作的规模。对于一个小的系统来说，一个分区可能就满足要求了，但对大型系统来说，有一个好的分区设计可能对成功具有至关重要的意义。

总之，在这方面，首先是分区问题，然后是分区中的分层问题，而不应该反向思考。

让我们看一下是否能够将这与DDD结合起来，这将我们引入了对限界上下文模式[Evans DDD]的讨论。

### 10.1.3 限界上下文

一个常见问题是“应该如何处理复杂性？”答案同样也很简单：分而治之。使用层次结构是分而治之的一种方式。

DDD在很大程度上就是有关层次结构的。本书一直强调的一个特别有用的工具是聚合模式，它是一个粗粒度的单元。实体是另一个单元，而值对象则又是另一个单元（这三个概念在[Evans DDD]都有详细讨论）。

如果沿相反的方向移动，即从聚合向“上”移动，那么将发现限界上下文的位置与聚合并不相同，而与子系统或子模型相同。

限界上下文的基本思想是明确表示模型适用的上下文。限界上下文内部的模型及其概念应该有严格的限定，外部的对象实际上是无关紧要的。

从一个更窄的角度来看，划分几种限界上下文的原因是为了能够在一个模型中思考所有细节，而且能够保持它的纯净。同时，从更宽的角度来看，可以考虑“大的”抽象，这样更容易掌握并从总体上思考抽象。

聚合的一大优势就在于其边界。它简化了工作，降低了耦合度，并使得连接更清晰。同样，在本章中我们也会发现边界的优点，这里是在一个更高的级次上，即限界上下文。

---

**说明** 尽管我们看到了边界的极大价值，但它也是有代价的。在通信和性能方面，连接可能会引入大量的开销。通常，应保持它的简单，而且当获益大于代价时，再增加复杂性。

---

如果将这前面章节中使用的示例（即销售订单）结合起来，那么就有两个限界上下文，一个用于订单，另一个用于发货。在本书的这个示例中并未这样进行讨论，但在实际的大型项目中，从技术观点来看这是有意义的，而且不同的组织部分可以很好地处理每个业务流程，这就是分割的好理由。

### 10.1.4 限界上下文与分区有何关联

前面谈到了分区和限界上下文，但这并不意味着它们是同一种技术。可以将限界上下文看作与典型的分区是相同的。也可以说限界上下文由几个分区构成，但反之则不成立。

### 10.1.5 向上扩展 DDD 项目

使用限界上下文的一个典型原因是它们提供了将DDD向上扩展到更大的上下文的一种方式。不应由一个庞大的团队来掌控一个大的领域模型，更容易理解模型的方法是将模型分割为几个较小的模型，从而让开发小组可以关注他们自己的模型。

限界上下文还可以防止通用语言变得不灵活，并避免松散的定義。相反，每个限界上下文可以有一种通用语言，这样就有了多种通用语言。从某种程度上讲，这是一个缺点，但另一方面，

每种通用语言都具有更强的功能，而且更准确。与其有一个坏的，不如有两个好的。

当我们发现所预想的单一模型开始出现一些具有不同意义的概念，而且通用语言不如以前清晰和简捷的时候，那么这可能就是一个很好信号，说明它实际上已经变为两个模型了。我们不必抵制这种变化，而可以促进它，并定义两个不同的限界上下文。

现在该到切换重点的时候了，下面讨论一下领域模型分区的另一个原因。

### 10.1.6 为什么对领域模型——SO 分区

假设已经有一个大的领域模型，为什么要对它进行分区呢？这并不是一个新问题，但与其他分布式系统解决方案（例如EJB和COM+）相比，SO概念使这个问题变得更明显。在前者的这些系统中，分区是有帮助的，但通常并不是严格强制执行的，而分区却是SO系统的一项标准。

**说明** 我知道很多SO人员并不是从领域模型开始考虑问题，而是从接口开始。我经常是两个方面都考虑一些。

本书不是关于集成和消息传递的，而只关注一个服务或一个应用程序的架构和设计。但下面一节将暂时离开重点，看一下Udi Dahan写的面向服务架构（SOA）简介。

## 10.2 SOA 简介

作者：Udi Dahan

DDD和SOA都是相对较新的概念，DDD已经有了很好的定义，而SOA则稍次之。尽管有些人声称SOA早在CORBA的早期就出现了，但却从来没有像现在这样有如此广泛的供应商和行业购买这样“一种只是架构”的技术。

### 10.2.1 什么是 SOA

尽管人们已经提出了很多SOA定义，但还没有哪一个定义成为最后的胜利者。大多数定义都基于以下思想：系统由独立的服务组成。然而，服务是由什么组成的却并不清楚。一个无可反驳的SOA定义是SOA是一种软件开发方式。特别地，SOA处理的是分布式系统或应用程序。分布是一个主要的架构问题，对这一点的认可是SOA的基本趋势之一。

### 10.2.2 为什么需要 SOA

要在分布式环境中运行的软件需要不同的设计。这是由分布式对象思维引发的根本转变，其中应用程序将“透明地”使用非本地资源，而不必知道这些资源的位置。分布式对象架构的问题主要是性能。使用本地资源和远程资源的语义是不同的，在分布式系统开发中有一个众所周知的深刻教训是：Chunky over chatty。

胖接口（Chunky interface）又称为粗粒度接口，与细粒度接口相比，它在单个调用中需要执行更多操作，而细粒度接口则要求多个调用执行同样多的工作。需要在细粒度接口上实现的显式协调和资源锁定操作被完全封装在一个大块调用（chunky call）中。这减小了客户代码与服务器

代码实现之间的耦合。

实际上，分布式系统架构的演变被新思想的引入和发人深省的实际性能效果打断了。尽管摩尔定律被奉行多年，而且显然还将继续下去，但大规模系统开发世界却仍然以性能作为主导。然而，性能并不是作为一个静态度量指标而受到如此多的关注，而是架构师们致力于使之实现最大化的可伸缩性，即不同负载下对性能的动态度量。

SOA是架构演进阶梯的下一个台阶，它使得人们能够在可伸缩的解决方案中构建新系统并利用旧系统。

### 10.2.3 SOA有什么不同

当前的分布式系统开发实践是设计一些层，然后将这些层部署到物理层中，相比之下，SOA将这两个概念统一到“服务”当中，服务同时作为设计单元和部署单元。SOA与当前实践的另一个区别是服务之间的通信编码，虽然在层的通信方式上没有约束，但服务必须根据某一模式定义的消息交换模式进行通信。无论在理论上还是在实际中，都支持在服务之间通信时仅使用单向异步消息传递，但是，这尚未得到整个行业的支持。

---

**说明** 注意，“模式”（schema）在这里不是指XML，而是指在服务之间流动的消息集合。虽然XML（以及进而所使用的SOAP）不是SOA的先决条件，但XML的扩展性使得模式的版本控制更容易。

---

### 10.2.4 什么是服务

除了同时作为设计单元和部署单元之外，在围绕SOA的各种讨论中，对服务就没有具体的定义了，这一点不免令人惊讶。大多数文献没有定义服务是什么，而只是描述了服务的属性。这样的例子就是面向服务有4条原则。

- 服务是自治的。
- 服务具有显式的边界。
- 服务公开模式和契约，而不公开类或类型。
- 服务兼容性是基于策略确定的。

虽然这些属性确实推动了可伸缩服务的设计（后面将详细讨论），但它们在分布式系统设计方面显然只提供了很少的指南。刚刚开始采用面向服务技术的架构师们在很多方面缺乏信息，例如哪些元素应该被分组到服务中，如何识别和定义服务边界，以及如何使用经典的OO模式来实现这些服务。此外，SOA的“纯粹的”架构方面的技术仍然处于不断变化的状态中，而且各种供应商产品除了对现有产品进行重新包装之外，没有什么新的变化。SOA是否正被作为一种天花乱坠的广告向人们大肆轰炸？

### 10.2.5 服务中包括什么

大多数开发人员和架构师对于前面提出的那个问题（即哪些设计元素应该被分组到一个服务

中)并不陌生。这个问题在引入组件和对象时已经被提出来了。虽然将几个对象或类组织到一个组件中是一种好的思想,但同样的方法却不适用于组件和服务。服务并不只是组件的集合,它更是软件构建方式的根本改变。

考虑到服务的通信是基于消息交换的,因此我们发现服务由不同类型的元素组成,而不仅仅是由“组件”或“对象”组成的。为了公开功能,服务使用一种消息模式。此模式由以下几个元素组成,最基本的元素包括:用于定义实体的“数据类型”(例如Customer)、定义了可以执行哪些操作的消息(不一定在每个实体上都定义),以及定义了与已定义消息之间的交互的服务契约(例如,CustomerChangeRequest将导致CustomerChangeReply)。显然消息模式既不是对象,也不是组件,然而它却是服务的一个必不可少的部分。

另一个总被忽视的部分是服务主机。如果没有某种环境的话,服务将无法提供任何功能。使用HTTP的Web服务将使用IIS、Apache、WebSphere或某种其他Web服务器作为主机。其他服务可能以COM+、Windows服务或甚至是简单的控制台应用程序作为主机。尽管行业正向着与传输无关的服务的方向发展,但特定的主机实现细节(像协议)确实对服务有很大的影响。HTTP和其他基于TCP的协议都是面向连接的协议。在分布式系统中,元素可能会发生故障、重启或只是响应速度缓慢,因此面向连接的协议可能导致性能下降,直到发生超时。连接协议(例如UDP)会导致其他问题,因为没有内置机制来识别消息是否到达目的地。如果说有服务拥有什么必不可少的部分的话,那么就是其主机。

如果不考虑这些非典型的元素,那么服务确实是由组件和/或对象组成的。然而,没有任何组合可以满足前面给出的4条SOA原则。下面就来详细讨论它们的意义。

### 10.2.6 深入分析 4 条原则

前两条原则(自治性和显式边界)是互相关联的,每个服务都独立于其他服务,而且显然一个服务结束的地方就是另一个服务开始的地方。在这方面的一些衍生规则包括:当某个服务崩溃时,不应导致其他服务崩溃。因此,在最基本层面上,不同服务应该在不同的进程中运行,因此,在调用方法时,某个线程不能进入一个不同的服务。在更深的层面上,某个服务不应该锁定另一个服务中的资源,ACID类型的事务应该被限定在单个服务的范围内。

**说明** 术语“事务”最初用于描述关系数据库的一种使用方式,而且事务的4个属性被写成首字母缩略词ACID:原子性、一致性、隔离性、持久性。数据库的一个常见实现是满足这些需求,相关的行、页和表被锁定到一个事务范围内。

第三条原则(模式,而不是类)看起来是关于互操作性的讨论,即在服务之间进行通信时,我们使用XML,将它作为所谓的“最小公分母”。然而,从根本上讲,这条原则讨论的是关注点的分离,也就是说,将实现所需行为的类从“调用它”的外部客户分离出来。这与Java RMI、COM或.NET远程调用是完全不同的,在后面这些技术中,客户代码必须引用服务器类型。当使用模式时,需要在外部消息格式与内部类调用之间执行一个显式的映射。这样就可以更改服务的内部实现,具体说就是在不影响外部客户的情况下对调用哪个类或接口进行更改。

第三条原则还有一层更深的含义，即版本控制。虽然类型可以组成层次结构，从而一个类型是另一个类型的一种，但历史已经证明对类型进行版本控制是非常难的——这使我们不由想到了“DLL地狱”。对一个服务进行版本控制的目的是避免更新已部署的客户代码。具体说就是已部署的客户将继续与先前的端点进行通信。模式继承的优点（也是类型中没有的）是模式可以是兼容的，尽管它们是不同的。这就是XML的可扩展性的重要之处。

例如，在部署了服务和客户的第一个版本之后，我们可能需要部署服务的第二个版本，以便可以部署几个新客户（版本2），它们需要从服务得到更高级的功能。显然，我们不希望这样的更改导致现有客户的中断，也不希望必须更新它们。为了实现这一点，版本2的服务的模式需要与版本1的模式兼容，当不使用模式进行通信时，这是一项更困难的任务，而且不是开发人员友好的。我们可以想象一下这种情况：必须在每个方法签名的结尾添加类似于“params object[] extraData”这样的代码。

SOA中第四个服务属性（兼容性是基于策略的）进一步强化了前面两点——由服务决定是否对给定请求做出响应，而且它添加了另一个维度，即元数据是作为请求的一部分传递的。这种元数据称为策略。在大多数给定的表示中，策略是用加密或身份验证这样的示例来描述的。这种思想基于关注点分离，定义了逻辑通信的模式与加密策略是分开的。如果有一些要在服务之间传递的信息，而又不想在模式中表示出来，那么一种方法就是使用策略。

### 10.2.7 再来看一下什么是服务

既然已经讨论了这些服务属性，那么就让我们总结一下服务是什么。以前，我们都编写过与用户交互的应用程序。现在，我们将应用程序称为“服务”，而且它们不仅能够与用户进行交互，而且还能与其他“服务”进行交互。注意，分布式应用程序不能被称为“分布式服务”，而是作为“协同工作的一组服务”这样一种模型。一个不易理解的经验规则是“每个服务有一个可执行文件”，或者对于Web应用程序来说是“每个服务有一个Web根目录”。另一方面就是数据库，虽然不一定每个服务都必须有一个数据库，但两个服务是不应该共享一个数据库的。共享数据库实际上为服务打开了后门，实质上是绕过了验证检查和逻辑，这样就违反了自治性原则，并增加了服务之间的耦合。

### 10.2.8 OO在SOA中的定位

我们已经明白了应用程序实际上就是一个服务，是更大整体当中的一部分，那么面向对象技术作为一种已经公认的应用程序开发实践，在SOA中又处于什么位置呢？尽管一些人提出SOA将成为下一代应用程序开发方法，并有效地取代OO，但这件事尚未发生。主流开发方法从面向过程的语言过渡到OO编程花费了20多年时间。从OO向SOA的转变不会这么久，可能在更短时间内就会完成。

事实上，SOA和OO是在不同的层次上操作的。OO处理的是单个部署单元中的设计和代码，而SOA则处理由多个部署单元构成的解决方案。SOA并不意味着用新的或替代的方式来设计或编写给定应用程序或服务的内部逻辑。没有理由阻止SOA和OO成为两种协作技术。当开发OO应用

程序时，SOA提出的一个问题是，这些应用程序将如何接受和响应那些来自非用户源的消息，而且甚至是在与用户活动并行的情况下做出响应。某个给定应用程序在接收请求的同时，能够向其他服务发出它自己的请求，这个事实使我们得出这样一个结论：交互已经不再遵从经典的“客户-服务器”模式。

## 10.2.9 客户-服务器和 SOA

就在不久之前，大多数应用程序开发还是客户-服务器式的。例如，客户软件向数据库服务器“发送请求”（SQL），并从其接收响应。后来，客户-服务器逐渐失宠，并被3层（或N层）架构所取代，应用服务器开始出现。同样，请求沿一个方向发送，响应则沿另一个方向返回。

然而，随着系统的增长而且互连程度变得更高，N层概念也变得模糊了，因为应用服务器开始处理来自其他应用服务器的请求，这些应用服务器就充当了其客户。在实践中看不到分层架构的固有秩序了。最后，客户软件开始从其服务器异步地接收事件，并实际响应请求，而这些操作并不是显式设计的。

除了“客户软件也应该被设计为服务”这种SOA理论之外，实际上客户代码也需要能够接收消息。考虑下面的情况：服务器上的某项操作需要很长时间才能完成，比HTTP的超时期限还要长，甚至可能是几天时间。对于那些需要人员干预的操作来说，这并不稀奇。虽然客户可以通过对服务器进行轮询来完成操作，但这为服务器增加了过量的负载，并需要在客户上执行后台任务。如果可以使用同一个后台任务来侦听工作是否完成的消息，那么就可以从服务器上移除此负载。对于当今技术来说，这种解决方案不仅是可行的，而且很容易实现。

采用这种方式，客户可以接收任意数量的消息类型，从而有效地成为一个服务。事实上，对于这些消息交换的更恰当描述是“发起者与目标之间的会话”，而不是“客户与服务器之间的请求/响应对”。在哪些服务可以发起会话，哪些服务可以响应会话方面，没有固有的约束。

## 10.2.10 单向异步消息传递

单向异步消息传递出现在响应时间很长的情况中，在这些情况下，必须从请求消息流断开响应消息。然而，发出请求的服务是否能够知道消息到达目标并得到响应需要多长时间？即使前一个结果很快到达了，也不能保证未来的响应性。本质上，如果与一个服务的所有交互都基于这种思想，那么该服务的客户将不受服务负载的影响。这种范型还具有其他优点。

由于发起者不再需要同步响应，因此目标服务现在可以在最适当的时候处理请求。如果目标服务正在处理大量负载，则可以将新消息的处理安排到后面再进行。换言之，我们可以随时间来执行预先规划好的负载均衡。例如，可以将代码编写成根据请求的性质来安排它的未来处理。

在前面的同步请求-响应范型中，发起者只能接收来自目标的响应。现在，由于发起者正在侦听响应，因此就没有“只有目标服务能返回响应”这个约束了。目标服务现在可以将返回响应的任务卸载给其他服务。

使用单向异步消息交换模式有很多优势。解决方案的灵活性和可伸缩性都增加了。这些获益也显然是有代价的，但后面我们会看到这些代价并不是很大。

### 10.2.11 SOA 如何提高可伸缩性

通过使用单向异步消息传递，服务可以通过对请求进行排队来调整负载。在负载不断增加的情况下，这能够避免服务性能的下降，因为影响性能的资源（例如线程）不会被耗尽。通过保持在任意负载下资源利用率均接近为常数，可以提供最大化的性能，这样使用单向异步消息传递的服务可以实现更好的性能特征，如图10-1所示。

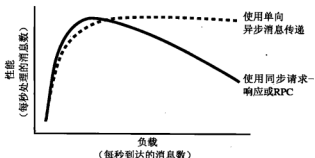


图10-1 单向异步和同步请求-响应/RPC消息传递的负载性能

注意，虽然在很少的负载下，但处理一条消息的时间仍然比使用经典的RPC更长。与繁重负载下所实现的更大吞吐量相比，时间较长的问题就显然比较次要了。

### 10.2.12 SOA 服务的设计

采用单向异步消息传递的服务的高级设计如图10-2所示。

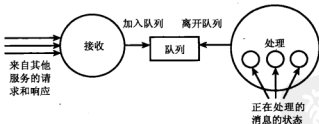


图10-2 采用单向异步消息传递的服务的假定高级设计

在这种设计中，当收到消息时，除了将它放在队列中之外，不进行其他处理（这最大限度地减少了应用服务器打开的线程数，不管负载或处理时间如何）。接下来，在未来的某个时刻，一个处理线程将从队列中取回该消息并处理它。如果此处理需要向其他服务发送消息，那么接收响应的方式将与其他任何请求均相同。此外，处理线程不需要等待这些响应，而是可以继续处理新的消息，先前消息的处理状态可以保存在某个存储中，当响应到达时，再从这里检索出来。注意，当使用进程外队列时（例如MSMQ），则可以用进程来代替线程，以便通过在单独的机器上运行这些进程来更好地支持向外扩展的增加。



### 10.2.13 服务之间如何交互

我们通过一个示例来看一下，当某个请求需要从其他服务检索信息时，服务如何处理该请求，并关注一下实际的处理是如何工作的。（注意，在下面的图中，当一个水平箭头出现在另一个箭头下面时，这意味着下面的箭头表示一个事件，此事件发生在上一个箭头的事件之后的某个时间点。）

首先，一个请求到达我们的服务，它被放到一个队列中，然后被处理。为了处理该请求，需要从其他三个服务检索信息。在请求被指派给其他服务之后，保存原始请求的处理状态（挂起响应1、2、3）（参见图10-3）。

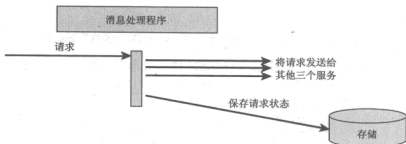


图10-3 请求被指派给其他服务，状态被保存

当来自服务2的一个响应到达时，它在队列中等待，直到它到达消息处理程序。消息处理程序从存储中检索原始消息的状态，并更新它（挂起响应1和3）。由于原始请求的处理尚未完成，因此响应2的处理现在已经完成了（参见图10-4）。

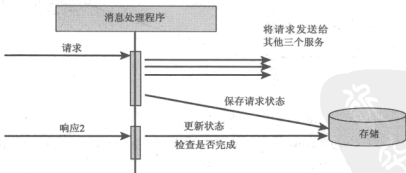


图10-4 一个响应被处理

如果在前一个请求完成处理之前，另一个请求到达了，那么它对于原始请求的处理没有影响。它可以开始自己的处理，并可以用相同方式向任意多的其他服务发送请求。我们可以用相同的资源来交叉处理多个请求（参见图10-5）。

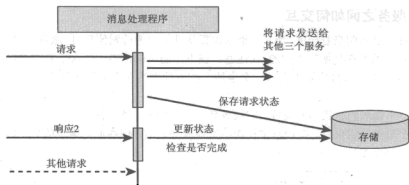


图10-5 另一个进入的请求不影响仍在处理过程中的第一个请求

当来自服务3的响应到达时，处理方式与服务2的响应处理方式相同。由于尚未收到所有必需的响应，因此仍然不向原始请求的发起者发送响应（参见图10-6）。

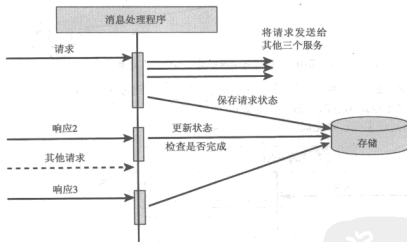


图10-6 另一个响应到达，但仍有一个响应未到达

最后，只有当来自服务1的响应到达时，原始请求的处理才能完成。在更新原始请求存储中的状态之后，现在服务知道所有请求都已到达，这样就可以将响应发送给发起者了（参见图10-7）。

注意，消息处理逻辑不必直接将响应发送给发起者，而是可以对响应进行排队。将返回响应与消息处理分离开的优点是可以减少消息处理逻辑的负担。同样，从关注点分离的角度来看，如何以及何时返回响应不在消息处理的职责范围内。

尽管我们已经考虑到服务可能要花费一定时间才能响应请求，但尚未解决的问题是如何处理不返回响应的情况。当前的设计未解决这个问题，因为它有意避开了面向连接协议中的超时问题。

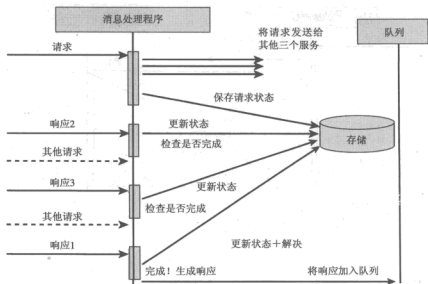


图10-7 第一个请求的响应到达，可以将结果加入队列了

### 10.2.14 SOA 和不可用的服务

虽然在经典的RPC风格的分布式系统中，在尝试连接已关闭的远程服务器时，将收到异常，但在刚才的这种设计中，根本不会收到任何异常或响应。这样，就需要通过某种方式知道特定请求的处理是否已经超时。一种简单方法是为每个发送的请求分配一个定时器。但是，这并不是问题的有趣部分。

当等待一个响应的时间期满时，应该执行什么操作？这是另一个设计领域，在这里需要使用领域逻辑。领域逻辑需要回答各种问题，包括

- 是否应该返回错误？
- 是否应返回部分结果？
- 是否应该重新尝试发送请求？
- 是否应该结合使用上述这些方法？
- 是否不管消息如何都使用相同的行为？

用前述服务设计方法来替代经典的RPC风格设计还有另一个优点，即当远程服务不可用时，所执行的领域逻辑是与主消息处理路径明确分离的。在RPC风格的设计中，在消息处理代码的中间会抛出一个异常，这样就必须编写代码，以便通过上述问题所描述的各种方式来处理异常，而这将使得领域逻辑的目的性变得模糊。

在图10-8所示的交互图中，可以看到用于处理超时的代码不仅是与消息处理代码分离的，而且也在时间上是分离的。请求和失败处理逻辑都被封装。

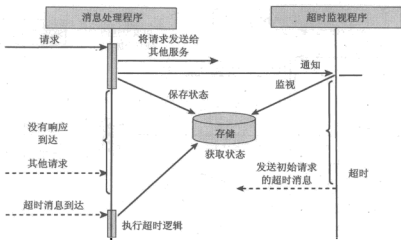


图10-8 应用超时监视程序

### 10.2.15 复杂的消息传递处理

当使用前面的设计来开发服务时，也能够构建复杂且健壮的服务间 workflow。例如，当收到 Request A 时，向 Alpha 和 Beta 发送请求。当从 Alpha 或 Beta 收到一个响应时，如果响应值大于某个阈值，则向 Gamma 服务发送一个带有该阈值的请求，否则就向 Delta 服务发送一个带有响应值的请求。如果 Delta 服务不可用，则返回部分结果。如果 Alpha 服务不可用或返回一个错误，则应将错误返回给请求的发起者，以此类推。在整个过程中，也可以处理其他消息，而且可以同时展开很多其他复杂处理。并发执行不仅是可能的，而且具有资源高效性，即使在负载下也是如此。

### 10.2.16 服务的可伸缩性

我们已经看到了单向异步消息传递为 SOA 提供的更大可伸缩性。但是，采用这种方式设计的服务是否能够有效地利用同一台机器上的多个处理器？对于不断增加的服务器的数量又如何？

首先看一下消息处理。每个请求的消息处理线程/进程是无状态的，它在存储中查找适当的状态，并相应地处理结果。因此，我们可以增加并发消息处理程序的数量，而不会发生错误。由于每个消息处理程序都是独立于其他处理程序的，因此不需要内存锁定，某个给定的消息处理程序也不需要等待（除非没有消息可处理）。

对于具有多个处理器的一台机器来说，这意味着增加消息处理程序的数量，并增大吞吐量。对于多个服务器来说，可以将收到的消息放在一台服务器上，进行排队，并存储在该服务器上，同时可以将其他消息处理程序放在其他服务器上。

向外扩展场景的最后一个优势是它可以在运行时添加更多服务器。通过监视队列中项的数目，可以知道系统何时到达峰值容量，然后添加更多资源，而无需系统停机，也不会对性能有任何损害。

### 10.2.17 小结

有关SOA的一个要记住的要点是，它“只是”分布式系统开发最佳实践的一次演进。关注点分离和接口与实现独立这两条核心OO原则在面向服务实践中仍然有所体现。尽管我们已经触及了很多话题，但在如此短的一个简介中无法深入地介绍每个话题。希望读者能够理解SOA不是什么，以及在服务之间哪些工具是不应该使用的。至于服务逻辑的内部构造的问题，我想它是永远无法明确回答的，但我确信Jimmy会将读者引入正确的轨道。

感谢Udi!

如果读者想要更深入地了解SOA或更专业的消息传递知识，[Hohpe/Woolf EIP]一书可以作为一个好的起点。

SOA背后的推动力量之一是降低耦合度。另一种用于降低耦合的技术就是下一节的主题。

本书已经多次提到依赖注入，但我只是手工地应用它。在很多人看来，在构造方法中注入实例所需要的内容是最自然的操作。对于小规模的系统，它很容易实现，而且是需要测试期间执行的操作，但从大规模系统的运行时角度来看，它就变得复杂和笨重了，而且更难重用。

在下一节中，Erik Dörnenburg将讨论控制反转（作为原则）以及依赖注入（作为模式）。Erik，现在读者交给你了。

## 10.3 控制反转和依赖注入

作者：Erik Dörnenburg

前面的章节讨论了如何设计和开发领域对象和相应的基础架构。本节介绍一种特殊的设计技术，或者叫做模式，它可用于在运行时组装对象。总体原则被称为控制反转（Inversion of Control），而特定模式则被称为依赖注入（Dependency Injection）。

### 10.3.1 任何对象都不是孤岛

在提供业务功能的应用程序中，几乎任何对象都是依赖于其他对象的。这些可能是提供数据或某种形式的服务的对象。当使用TDD来设计和开发对象时，我们通常显式地提供所需对象，并且用桩或模拟来代替它们，以便在测试中隔离对象。但是在运行时，对象则需要一个真实的实现，而且应用程序中的所有对象必须以某种方式被实例化并组装到一起。

解决对象之间依赖性的最常见方法是将创建或获取依赖对象的责任分配给具有依赖性的对象。最简单的方式是让具有依赖性的对象实例化它所需要的对象。

举例来说，假设我们正在创建一个计价器（pricer），它是一个用于计算某种金融工具（例如债券）价格的对象。当然，计价器需要被计算的债券，但我们决定不把它作为计价器对象状态的一部分，而是在每个计价请求时传入它。因此，类的代码如下：

```
public class BondPricer {
    public double GetPrice(Bond bond) {
        /* do something really complicated here */
    }
}
```

债券的价格还依赖于其他因素，一个例子就是贴现曲线（discount curve）。（读者是否熟悉这个领域并不重要。对于本例来说，计价器可能还依赖于“袋熊”，实际上具体依赖什么并不重要。）一条曲线可用来对很多不同的债券计价，它相对来说是一个常量。因此，我们决定将曲线作为计价器状态的一部分，并将它保存在成员变量中，这些成员变量在构造方法中被初始化。

```
//BondPricer
private readonly DiscountCurve curve;

public BondPricer() {
    curve = new DiscountCurve();
}
```

注意，我们可以根据需要来创建曲线，但何时运用延迟实例化却不同。在任何情况下，计价器对曲线都有一个直接的依赖。也可能贴现曲线对象本身依赖于其他对象（它在适当的时候对这些对象进行实例化）。这看上去是一种可行的方法，但更仔细地检查一下，问题就变得明显了：我们有了一组具有函数功能的对象用来对债券计价，但解决方案的灵活性和可维护性如何？

假设贴现曲线对象从一个数据库读取曲线上的点。这是用户想要使用的方式。在后面的迭代中，我们需要构建一个供用户分析假设场景的特性。为了使用此特性，用户需要在UI中指定曲线上的点。最简单的方法是从DiscountCurve类创建一个接口，并提供两个独立的实现，即DatabaseDiscountCurve和InMemoryDiscountCurve。我们修改BondPricer类，令它只使用接口上的方法，从而不再与基于数据库的实现耦合。

遗憾的是，尽管债券计价器中的计算代码只依赖贴现曲线接口，但我们必须在某个时刻实例化IDiscountCurve接口的一个具体实现，在本例中就是BondPricer的构造方法。在这个时间点上，必须决定在两个实现中需要实例化哪个实现，但无论选择哪个，类都会与接口以及该实现发生耦合。图10-9显示了类的设计，并清楚地演示了问题：BondPricer与DatabaseDiscountCurve之间的creates链接。

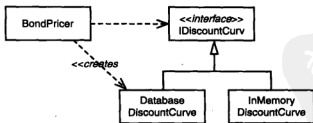


图10-9 使用直接实例化时的依赖性

直接实例化的另一个问题是在应用程序中没有单一位置来控制贴现曲线的创建，这样就无法将相同的实例重用于不同目的。这可能是一个严重的问题，如果原始版本的曲线从数据库加载信息，那么显然重新创建新实例可能对应用程序性能产生极大影响。

在讨论控制反转和依赖注入之前，先来看用于解决这些问题的另一种常用方法。

### 10.3.2 工厂、注册类和服务定位器

一种解决耦合问题的方法是使用工厂模式。创建一个工厂类，由它提供所需的对象，在我们的示例中，工厂就是一个实现贴现曲线接口的对象。

```
public class DiscountCurveFactory {
    public static IDiscountCurve GetCurve() {
        return new DatabaseDiscountCurve();
    }
}

public class BondPricer {
    private readonly IDiscountCurve curve;

    public BondPricer() {
        curve = DiscountCurveFactory.GetCurve();
    }
}
```

使用这个工厂，计价器就不再与曲线实现紧密耦合了，这正是我们要实现的目标。遗憾的是，使用工厂后，工厂类本身与特定的IDiscountCurve产生了紧密耦合，因此仍然很难在不同的计价器应用程序中使用不同的曲线实现。为了克服此问题，可以创建一个工厂接口，并传递不同的工厂实现，但这意味着必须获取正确工厂，这类似于获取正确的对象。换言之，我们可以通过添加一个间接层来提供额外的灵活性，但这只是将已有的关键问题转移到了别处。

工厂模式也解决了实例管理的问题，但这在代码中并未体现出来。由于我们在代码中有一个中心位置（即GetCurve()方法）负责提供曲线实例，因此可以实现应用于整个应用程序的替代策略。例如，我们不必总是创建新曲线，而可以维护单一实例，或者如果应用程序是多线程的，则可以管理一个实例池。

在进一步完全解除耦合的下一步骤中，可以用注册模式[Fowler PoEAA]的实现来代替工厂。与工厂类似，注册类提供了给定类型的对象，但它们不是硬编码类型的对象，而是在外部做出这个决定，并使用反射来实例化对象。这种方法是工厂模式的一种改进，因为现在有了两个分离的关注点：有一个负责提供对象的类（即注册类），并且可以把用于决定在每种情况下使用哪个具体类的代码移动到代码库中更适当的位置。

这样一个注册类的实现是很简单的，但它很有效，并且最终实现了我们的目标，即从IDiscountCurve的具体实现解除BondPricer的耦合。

```
public class DiscountCurveRegistry {
    private Type curveType;

    public static void RegisterCurveType(Type aType) {
        curveType = aType;
    }

    public static IDiscountCurve GetCurve() {
        return (IDiscountCurve)Activator.CreateInstance(curveType);
    }
}
```

图10-10显示了得到的类设计。它突出显示了计价器只依赖于接口，但该图也使我们注意到了注册类的两个重要含义：计价器依赖于注册类，而且我们需要其他一些代码（即一个汇编程序），它为注册类提供具体实现。

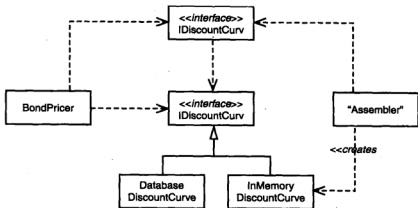


图10-10 使用注册类时的依赖性

本例中使用的曲线对象是数据源，但很容易看出，我们可以使用相同的模式来定位用于提供对象的服务（例如审计服务，此服务将计算后的价格写入日志）。采用这种使用方式时，模式通常被称为**服务定位器**[Alur/Crupi/Malks Core J2EE Patterns]。

在实现中，大多数定位器又前进了一步，不提供单独的注册集合和为每个服务获取方法，而是提供一个通用方法集合和一种动态命名模式。

```

public class Locator {
    public static void RegisterType(String name, Type aType) { ... }
    public static Object GetInstance(String name) { ... }
}

```

使用服务定位器，我们向特定名称（最常见的是接口名称）的定位器注册一个具体曲线实现。这一步完成后，债券计价器就可以通过查询定位器来获得贴现曲线了。

```

public class BondPricer {
    private readonly IDiscountCurve curve;

    public BondPricer() {
        curve = (IDiscountCurve)Locator.GetInstance("MyBank.IDiscountCurve");
    }
}

```

我们选择在命名模式中使用字符串，这是一种常见的做法；但也可以使用对象类型。使用对象类型的优点是可以保持对IDE所提供的自动重构的完全支持，也有助于更好地进行错误检查，尽管这些优点以牺牲一些灵活性为代价。不管选用哪种方法，模式都是相同的，并提供了一定程度的解耦。



### 10.3.3 构造方法依赖注入

利用注册类和服务定位器，可以解除某个对象与它所依赖（间接使用）的那些对象的耦合，在本例中是一个在某种程度上优化的查找表。设置代码（setup code）负责决定使用哪个具体实现，还负责查找具有依赖性的对象所依赖的那些对象。它使用被动定位器对象，并在必要时主动从定位器“拉取”（pull）它所需的对象。

相比之下，依赖注入将这个过程反转过来，并使具有依赖性的对象成为被动的。对象声明其依赖性，但却完全忽略依赖性的来源。解决依赖性并查找依赖对象的过程留给外部机制来完成，外部机制可以采用多种方式来实现。通常，应用程序包含通用基础架构代码，这些代码加载一个配置文件，或使用一个内部注册表来保持所有对象和它们依赖性的列表。但是，依赖注入模式并不关注这个方面。严格地说，基础架构代码是补充性的，甚至通过从应用程序代码传入对象也可以满足依赖性要求。

依赖注入有几种不同的类型。首先看一下构造方法依赖注入。使用这种模式时，对象在其构造方法中声明依赖性，而且创建对象的代码将确保在运行时提供有效的实例。

```
public class BondPricer {
    private readonly IDiscountCurve curve;

    public BondPricer(IDiscountCurve aCurve) {
        curve = aCurve;
    }
}
```

上段代码的一个要点是，除与问题领域有关的概念之外，它不依赖于任何其他方面。它不需要引用工厂或注册类，而且与查找或创建实例操作无关。同样，依赖注入有利于设计出优雅的软件，这样的软件通常显示了领域和基础架构关注点之间的良好分离。

图10-11显示了使用依赖注入的设计。这里还包括一个汇编程序对象，它负责创建计价器和计价器所依赖的曲线。该图凸现了这样一个事实：利用此设计，任何特定于领域的对象均不依赖于通用基础架构代码（例如汇编程序）。

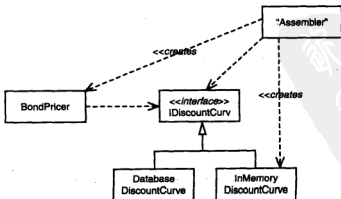


图10-11 使用依赖注入时的依赖性

债券计价器与任何基础架构代码之间均没有耦合，这也提高了可测试性。考虑计价器的第一个实现，它使用了贴现曲线的初始实现（从数据库加载曲线上的点）。这个类的测试总是需要访问数据库，从前面的章节我们知道，这是一种不好的思想。使用构造方法注入以后，很容易提供每个依赖的桩实现或模拟实现，因此很容易在不涉及数据库的情况下进行计价器的测试。

```
[Test]
public void testPriceIncludesValuesForOutstandingCoupons () {
    IMock curveMock = new DynamicMock(typeof(IDiscountCurve));
    IDiscountCurve curve = curveMock.MockInstance as IDiscountCurve;
    BondPricer pricer = new BondPricer(curve);
    /* do actual test */
}
```

这种方法比用其他模式实现的任何解决方案都简单。例如，如果选择一种采用了注册类的设计，那么测试就必须创建一个注册类，并向其注册测试的模拟实现。利用依赖注入，只需简单地传入所需的对象。这种简单性的基本原因是依赖注入模式提供了很高程度的解耦，这使得更容易将一个对象用于多种环境，至少可用于常规的运行时环境和测试环境。

我们尚未通过引入依赖注入模式来代替所有的注册模式功能。在前面的设计中，注册类还提供了另一个重要的服务，即抽象类型（IDiscountCurve接口）与要在给定应用程序或上下文中使用的具体实现之间的映射。

奇怪的是，在负责创建对象的代码中对映射进行硬编码就已经足够了。在我们的示例中，可能有两个从用户界面中不同的点调用的不同方法，一个方法基于从数据库获取的信息对选定债券进行计价，另一个方法首先要求用户提供几个点，然后在内存中根据这些点构造一个曲线，最后基于该曲线创建一个计价器。在这两种情况下，都可以简单地在相应方法中配置计价器。事实上，在这个例子中使用注册类反倒显得不恰当了。

```
public class BondInfoForm : Form {
    private TextBox priceTextBox;

    private Bond SelectedBond { get {...} }

    public void CalculateActualPrice() {
        DatabaseDiscountCurve curve = new DatabaseDiscountCurve();
        BondPricer pricer = new BondPricer(curve);
        priceTextBox.Text = pricer.GetPrice(this.SelectedBond);
    }

    public void CalculateScenarioPrice() {
        InMemoryDiscountCurve curve = new InMemoryDiscountCurve();
        this.GetCurvePointsFromUser(curve);
        BondPricer pricer = new BondPricer(curve);
        priceTextBox.Text = pricer.GetPrice(this.SelectedBond);
    }
}
```

在很多情况下，简单性是一种折中，而且简单设计往往无法解决更复杂的需求。幸运的是，基于依赖注入模式设计的组件在保持简单性的同时，并没有在创建复杂映射和配置模式方面增加

难度，而且有几种轻量级的IoC容器可以履行与注册类同样的职责。（后面将讨论两种这样的容器。）这意味着我们可以在两种解决方案之间做出选择，一种是前面所示的简单硬编码的解决方案，另一种是一些基础架构代码提供的更灵活的映射。

另一个有趣的方面是具有可选依赖性的对象。在债券计价器的示例中，将地区节假日考虑在内可以提高特定计算的准确性，这意味着节假日日程表是一个可选的依赖项。利用构造方法依赖注入，可以通过提供两个构造方法来表示它，分别用于每个有效的依赖项集合。

```
public class BondPricer {
    private readonly IDiscountCurve curve;
    private readonly IHolidayCalendar calendar;

    public BondPricer(IDiscountCurve aCurve) {
        curve = aCurve;
    }
    public BondPricer(IDiscountCurve aCurve
        , IHolidayCalendar aCalendar) : this(aCurve) {
        calendar = aCalendar;
    }
}
```

也可以使用一个构造方法，它带有所有参数，并且为可选依赖项传入null，但这降低了契约的明确性，特别是当依赖项的数目增加的时候。因此，推荐的方法是每个有效的依赖项集合提供一个构造方法，而且如果可能的话，将构造方法链接起来。如果无法做到这一点，那么可能是因为类具有过多的依赖项，因此应该被分解。

### 10.3.4 setter 依赖注入

顾名思义，setter依赖注入使用setter方法（而不是构造方法）来注入依赖性。因此，计价器有一个默认的构造方法，但它需要一个带有setter方法的额外属性。

```
public class BondPricer {
    private IDiscountCurve curve;

    public BondPricer() {
    }
    public IDiscountCurve DiscountCurve {
        set { curve = value; }
    }
}
```

setter注入与构造方法注入提供了相同程度的解耦，而且很容易看到，在单元测试中创建计价器对象像构造方法注入时一样直接。一个重要的区别在于，直到setter方法被调用之前，计价器一直处于无效状态，在构造方法注入中则不是这样。此外，我们不能将曲线变量声明为只读变量，因为它不是在构造方法中初始化的，最后，在实现最终的初始化功能方面（在所有依赖均已被注入后，需要运行这些功能），使用setter注入略显复杂，并且易出错。

尽管存在这些缺点，但setter注入仍然是有用的，因为它可以更好地扩展到那些具有很多依赖

项的组件。在具有三个可选依赖项的组件中，如果必须使用构造方法注入，那么将需要8个构造方法。人们可能会说这个组件需要重构，但重构并不总是可行，而且为这样的情况准备一种可以方便使用的模式是很有利的。类似地，很多现有对象都有一个默认的构造方法和一些属性，因此只能够通过支持setter依赖注入的IoC容器来使用它们。

setter依赖注入还有更多风格，例如接口驱动的setter依赖注入，它使用标记接口来区分描述依赖项的setter方法和其他公共setter方法。但这些风格或是未得到广泛应用，或是未对setter注入的关键特征有重大改变，因此这里不再讨论它们。

### 10.3.5 控制反转

在讨论了依赖注入模式及其具体实现之后，我们来看一下控制反转，也称为好莱坞原则（“不要打电话来，我们会打给你！”）。严格地讲，IoC并不是一种模式，而是一条通用原则，它只是表明了对象应该依靠它们的环境来提供其他对象，而不是主动获取对象。

从历史上看，IoC这个术语通常用来描述包含容器的方法，容器在运行时创建组件。看一下这些容器，Martin Fowler提出了一个重要问题，即“它们反转的是哪个控制方面？”他指出，人们把通过这些容器实现的基于setter和基于构造方法的依赖注入方法看作是一种具体的模式，并为其使用了一个更具体的名称：依赖注入[Fowler Inversion-OfControl]。这也使得人们能够将依赖注入模式与其他类型的IoC区分开来。

有一种广泛使用的模式，它既遵守IoC原则，又不是依赖注入的变体，它就是上下文配置依赖查找（Contextualized Dependency Lookup）。在我们的债券计价器中使用这种模式意味着继续使用类似于注册类的类，在此模式中它通常被称为上下文，但由容器为对象提供这个上下文，这就是控制反转方面。在代码中可能会像下面这样：

```
public interface IServiceContext {
    public Object GetInstance(String name);
}

public interface IComponent {
    public void Initialize(IServiceContext context);
}

public class BondPricer : IComponent {
    private IDiscountCurve curve;

    public void Initialize(IServiceContext context) {
        curve = (IDiscountCurve)context
            .GetInstance("MyBank.IDiscountCurve");
    }
}
```

更仔细地检查一下可以发现，此模式就像是服务定位器与setter依赖注入模式之间的一种混合。这并非偶然，因为历史上此模式就是第一批IoC模式当中的一个，并且被证明是更完善的依赖注入模式的一个重要的跳板。

### 10.3.6 使用了 Spring.NET 框架的依赖注入

Spring.NET应用程序框架[Spring.NET]是Java Spring框架[Spring]的一个端口，它提供了一个非常灵活的IoC容器，此容器支持各种风格的依赖注入。简单地说，此容器就是一个组件，实际上可以说是一个具有多个实现的接口，它负责加载配置，并相应地设置对象。历史上，“容器”这个术语解释了这样一件事：在Java平台上，IoC容器可以是标准J2EE容器的一个备选。

回到前面描述的构造方法注入的示例中，看一下如何利用Spring.NET所提供的容器来获得一个已配置好的计价器实例。

用Spring.NET来描述组件及其依赖性的首选方式是使用XML配置文件。我们通过类型来声明对象，包括对象所在的程序集（从这个程序集中加载对象），然后通过计价器声明中的一个嵌套的constructor-arg元素来解决计价器对贴现曲线的依赖性。

```
<objects>
  <object name="DbCurve" type="MyBank.DatabaseDiscountCurve, MyBank"/>
  <object name="DbPricer" type="MyBank.BondPricer, MyBank">
    <constructor-arg><ref local="DbCurve"/></constructor-arg>
  </object>
</objects>
```

要在代码中访问对象，可以创建IApplicationContext的一个实例，并通知它从配置文件加载配置，然后通过名称向它请求一个对象。上下文将确保所有依赖对象均被创建，并被注入到需要的地方。

```
IApplicationContext context = new XmlApplicationContext("spring.xml");
BondPricer pricer = (BondPricer)context.GetObject("DbPricer");
```

在实际应用程序中，如此直接地使用应用程序上下文的情况并不常见，因为这使得负责创建计价器的代码依赖于上下文。这非常类似于我们在注册模式中想要避免的类与注册类之间的依赖。如果假设需要计价器的类是一个Windows窗体，那么可以对窗体本身使用依赖注入，并将窗体添加到XML配置中。现在，当创建窗体时，计价器是通过容器注入的，而且窗体不必处理应用程序上下文。

然而，在这种情况下，必须有其他的某个代码片段负责创建窗体，而且此代码将从上下文中查找窗体，因此与上下文是耦合的。显然，这个问题是无法通过一直传递下去来解决的。为了解决这个问题，大多数容器提供了与应用程序框架（例如Spring.Web）的集成，以便特定于初始应用程序的组件（即Windows窗体和Web页面）可以简单地使用依赖注入，同时应用程序框架负责使用上下文来创建它们。如果对于特殊的容器/框架组合没有集成，那么最好尽可能多地使用依赖注入，并将对上下文的依赖限制到一个类中。

如果首选使用setter注入，则可以使用前面讨论的基于setter的计价器，并针对计价器来更改Spring.NET配置元素。获取计价器的代码保持不变，因为容器负责对象的创建。

```
<objects>
  <object name="DbCurve" type="MyBank.DatabaseDiscountCurve, MyBank" />
  <object name="DbPricer" type="MyBank.BondPricer, MyBank" >
```

```

    <property name="DiscountCurve">
      <ref local="DbCurve"/>
    </property>
  </object>
</objects>

```

Spring.NET的创作者们通常倡议使用setter注入，但容器为两种风格提供了同样好的支持。事实上，我们完全可以将构造方法注入和setter注入结合起来使用。如果在设计计时，在构造方法中将贴现曲线的依赖性声明为必需的（或非可选的），并且为节假日日程表依赖项提供一个可设置的属性，那么就可以像下面这样来配置带有节假日日程表依赖项的计价器。

```

<objects>
  <object name="DbCurve" type="MyBank.DatabaseDiscountCurve, MyBank" />
  <object name="Calendar" type="MyBank.DefaultHolidayCalendar, MyBank" />
  <object name="DbPricer" type="MyBank.BondPricer, MyBank" >
    <constructor-arg><ref local="DbCurve"/></constructor-arg>
    <property name="HolidayCalendar">
      <ref local="Calendar"/>
    </property>
  </object>
</objects>

```

这些只是Spring.NET的初步应用，但即使这些简单的特性也能够为那些根据依赖注入模式设计的组件的配置提供极大帮助。人们甚至会说，将过多和过于复杂的逻辑放到配置文件中是得不偿失的，因为这样提供的灵活性是很少需要的，而又以牺牲可维护性为代价。

### 10.3.7 利用 PicoContainer.NET 进行自动装配

前一节中使用的配置文件很容易理解，而且它们的目的很明确，即声明少数几个类以及它们之间的依赖关系。然而，它们相对来说有些冗长了，而且重复了也可以通过.NET类型系统获得的信息。例如，计价器类上的构造方法声明中表示了计价器依赖于曲线这个事实。实际上，我们的示例中所需的所有信息都可以从类型系统获得。

大多数IoC容器都支持基于运行时信息来解决依赖关系。这个特性被称为自动装配（auto-wiring），因为依赖组件的设置（即装配）是自动发生的，而无需任何额外配置。Spring.NET支持将自动装配作为XML文件的备选，但本节中将使用PicoContainer.NET[PicoContainer]增加变化，因为自动装配是此容器的首选选项。

使用PicoContainer.NET时，只需一个步骤即可使容器解决依赖关系并创建对象，即向容器注册具体的实现类。

```

IMutablePicoContainer pc = new DefaultPicoContainer();
pc.RegisterComponentImplementation (typeof(DatabaseDiscountCurve));
pc.RegisterComponentImplementation ("pricer", typeof(BondPricer));

```

通过将声明放在代码中，可以消除冗余，而且可以从IDE获得更好的支持，因为它现在能够检测到拼写错误的类型名称，也能够自动重构中安全地重命名类型。

在Spring.NET中，从容器获取对象的方式几乎与从应用程序上下文获取对象的方式相同，由

于这个原因，有关容器依赖性的相同原则也是适用的：几乎所有类都应该使用依赖注入，只留下一个中央类与容器进行交互，以便获取初始组件。

```
BondPricer pricer = (BondPricer) container.GetComponentInstance("pricer");
```

前述示例还显示了这样一个事实：除了使用类型系统以外，PicoContainer.NET还支持已命名的组件。在本例中就是计价器。这样就可以在不指定类型的情况下查找对象，因此又提供了另一种程度的解耦。

可选依赖性会自动处理的，因为PicoContainer.NET使用最复杂的构造方法来满足依赖性需求。在上面的示例中，PicoContainer.NET将一个参数构造方法用于BondPricer，它只带有一个曲线，但如果增加了节假日日程表的注册，那么在获取计价器对象时，PicoContainer.NET使用两个参数构造方法。

```
ImmutablePicoContainer pc = new DefaultPicoContainer();
pc.RegisterComponentImplementation (typeof(DatabaseDiscountCurve));
pc.RegisterComponentImplementation (typeof(HolidayCalendar));
pc.RegisterComponentImplementation ("pricer", typeof(BondPricer));
```

在对象依赖性方面，使用基于类型的自动装配方法来代替基于配置文件的方法并未损失任何灵活性，同时获得了更好的开发体验。如果类在其构造方法中需要一些参数，例如主机名、端口号和超时设置，那么自动装配方法的效果就要差一些，因为基本类型没有传递有关其目的的太多信息。如果向容器注册两个整数，一个用于指定端口号，另一个用于指定超时时间，那么容器将把它们看作同样的整数，而且在带有一个整数的构造方法中无法确定到底使用哪一个。

最基本的问题是，容器是否应该支持用处理配置参数设置的相同方式来注入所需组件，或者是否应将这两个需求看作不同的关注点，并对于每个关注点都用最适合它的方法来处理。当管理依赖性时，使用基于类型的自动装配的优点肯定大于配置文件方法的优点，因此可以将自动装配方法用于此目的。在相同的场景中，可以使用setter方法来设置已命名的属性的配置参数，这种方法更直观。

适用于.NET平台的第三种IoC容器是Windsor，它是Castle项目[Castle]的一部分。Castle项目的核心容器实现（开发人员将其称为微内核）使用基于类型的自动装配方法，但它也允许用插件来添加额外的依赖性解决策略。基于微内核的Windsor容器[Windsor]还增加了配置文件支持，因此开发人员能够选择最适当的机制。这是由NanoContainer[NanoContainer]首先开创的一种成功方法，NanoContainer是在名为PicoContainer的Java版本上开发的一个容器。

### 10.3.8 嵌套容器

Spring.NET中的容器被命名为ApplicationContext，这暗示着容器为那些从容器检索组件的对象定义了一个上下文。这些对象与返回的对象进行交互，而且它们的行为依赖于容器所提供的具体实现。因此，容器可以被看作上下文的提供者（对象在这个上下文中操作）。名称进一步暗示着，从配置文件创建的上下文是应用程序级的上下文，而且应用程序中的所有查找操作都共享同一个上下文。

Web应用程序使用多个上下文，而且大多数Web框架使用显式的上下文，形式包括应用程序、会话和请求对象。请求处理代码（例如Web网页后面的代码）通常依赖多个组件。一些组件是应用程序上下文的一部分，因为它们可以被所有请求处理程序共享，而有些组件则是个别的请求上下文的组成部分，因为它们仅供单个请求使用。

一些IoC容器实现支持这种模型，并允许容器的嵌套，这意味着一个容器可以有父容器，而且作为一个直接结果，可以有子容器。当某个容器无法满足用于创建组件所需的所有依赖性时，它就尝试从其父容器获取遗漏的组件，父容器进而又可以向它自己的父容器请求组件。这种设置非常类似于责任链设计模式[GoF Design Patterns]。

作为一个示例，考虑请求处理程序或页面的创建（这里，请求处理程序或页面对数据库连接和日志程序有一个依赖关系）。数据库连接在某个时刻只能供一个请求使用，而日志程序则可以被所有请求共享。然而，这对于请求处理程序是无关紧要的。

```
public ResultPage : Page {
    private Logger logger;
    private IDbConnection connection;

    public ResultPage(Logger aLogger, IDbConnection aConnection) : base() {
        logger = aLogger;
        connection = aConnection;
    }
}
```

在我们的应用程序中，有一个与应用程序关联的容器。此外，每个会话都有一个子容器，相应的会话容器将为会话中的每个请求创建一个子容器。应用程序将请求URL映射到一个网页类型，并包含以下通用代码，这段代码为一个给定的网页类型创建新的请求处理程序。

```
public RequestHandlerFactory {
    IPicoContainer sessionContainer;

    public Page CreatePage(Type pageType) {
        IMutablePicoContainer reqContainer = CreateRequestContainer();
        reqContainer.RegisterComponentImplementation(pageType);
        return (Page)reqContainer.GetComponentInstance(pageType);
    }

    private IMutablePicoContainer CreateRequestContainer() {
        IMutablePicoContainer pc = new DefaultPicoContainer(sessionContainer);
        pc.RegisterComponentImplementation (typeof(SqlDbConnection));
        /* register all other components here that should be available
           to requests on a per request basis. */
        return pc;
    }
}
```

在CreatePage()方法中创建了请求容器（request container）之后，代码向请求容器注册页面类型，以便页面自己可以使用依赖注入。当它在第三行代码中检索实际的页面实例时，请求容器尝试创建页面，但假设需要一个ResultPage，那么它就无法解决Logger类上的依赖性了，因为这尚未向请求容器注册。在这种情况下，它将尝试从其父容器（即会话容器）获取遗漏的组件，



而会话容器也不包含日志程序，因此再从其父容器（即应用程序容器）请求日志程序。假设一个日志程序被注册到该容器，那么就满足了页面的所有依赖项，并且由请求容器来创建该页面。

这种方法有两个优点：一方面是可以注册组件，并在适当的级别上共享组件，另一方面是从容器检索组件的对象只需使用一个容器即可，并且可以通过它在容器链中查找所需的组件。

### 10.3.9 服务定位器与依赖注入的比较

至此，我们已经讨论了本章开头所描述的用于解决耦合和实例管理问题的两种不同模式。剩下的问题就是何时使用哪种模式。

现在，服务定位器的使用更常见一些，但这并不绝对是因为它更好地匹配大多数应用程序。相反，最可能的原因是服务定位器模式已经存在很长时间，而依赖注入是相对较新的模式，它尚未广为人知。事实上，开发人员在尝试改进服务定位器模式时，往往就得到了依赖注入模式。

通过依赖注入模式所实现的组件与依赖解决机制的完全解耦提高了可测试性，因为测试不必再设置专门的定位器版本来提供对象的桩或模拟。它还有助于大型系统中的重用，因为避免了系统中所使用的不同组件被绑定到不同定位器实现的问题。

依赖注入模式（特别是构造方法依赖注入）的另一个优点是在代码中很容易看到对象的所有依赖项。现代的IDE能够在一个给定的类中集中显示所有服务定位器的使用（按Ctrl-Shift-F7键），但即使对于高级IDE来说，这也不如查看构造方法来简单。

人们经常提到的一个问题是，依赖注入模式将对象之间的生命周期联系在一起了，例如对象A依赖于对象B和C，那么A的生命周期就与B和C的生命周期联系在一起，因为B和C必须在A之前被创建，以便可以被注入。在服务定位器模式中就不存在这个问题，因为可以将B和C的创建推迟到A通过定位器请求它们时再进行。幸运的是，依赖注入模式也可以做到这一点，方法就是利用PicoContainer中的组件适配器的一种用法，尽管这种用法不太自然。容器不注入实际类的一个实例，而是注入一个代理，它仅在第一次被使用时创建实际实例，而且从那时起，将所有方法调用转发给实例。这对于具有依赖性的对象是完全透明的，但有效地解除了它们的生命周期之间的耦合。

### 10.3.10 小结

本节讨论了任何软件系统构建中都会出现的两个问题：依赖于其他对象的对象耦合和组件实例的管理。我们首先描述了如何通过最常用的工厂、注册和服务定位器模式来解决这些问题。在认识到这些模式都有特定缺点之后，又讨论了依赖注入模式，它采用一种全新的方式来解决这些问题。此模式基于控制反转原则，它提供了更好的解耦和扩展性，可用来解决从简单到非常复杂的依赖性的各种问题。它还提高了可测试性，这对于TDD是非常重要的，并且使领域模型完全与基础架构代码分离开来，这使其成为DDD的有用工具。

像任何其他模式一样，依赖注入并不是银弹，而且也无法总是能够用依赖注入模式来替代服务定位器模式的使用。理解每种模式的区别和益处并恰当地使用它们是很重要的。我们已经看到依赖注入模式有几种各有利弊的风格。此外，选择某一特定风格并不影响可测试性，它对于开发

来说是很重要的，而且在运行时使用的容器通常支持所有主要风格。因此，我们可以根据需要进行选择（甚至是混合）不同的风格。

进行系统设计时，最重要的考虑因素是分离关注点（在本例中关注点就是组件的配置和使用），然后从丰富的候选目录中选择最适当的设计模式。

感谢Erik！

我只想强调一下Erik最后说的话。依赖注入印证了Martin Fowler在他的DSL文章[Fowler LW]中的观点，这篇文章指出，开发可以被看作是创建和配置良好的抽象。依赖注入就是一种完成部分配置工作的方法。

注意，上面没有说“配置方面”，而只说成“部分配置工作”，这是为了不泄露“方面”这个词——它是下一主题的线索。事实上，依赖注入和面向方面编程（AOP）通常是成对出现的，我们很快就会发现这一点。

读者是否记得有关存储库设计的讨论？我们可以从以下几条原则中选择。

- (1) 特定的存储库。
- (2) 基类。
- (3) 带有泛型的基类。

方法(2)的问题（除了缺乏类型安全性之外）和方法(3)的问题是它们将创建一个自动的、非自然的抽象，所有存储库都将具有该抽象。这是有问题的，例如，并不是所有存储库都应该有删除的可能性。

此问题的解决方法可能是创建一个更小的基类，并为各种变体编写专门代码。如果采用此路线，可能会发现基类将只有2个或3个方法（`GetById()`、`MakePersistent()`等），这就与方法(1)非常接近了。

我们可以尝试将通用代码提取出来，放到特定存储库所委派的帮助方法类中，从而减少代码重复。这样就必须编写冗长乏味的委托代码，这不会令任何人高兴。

另一种更好的方法是不从基类开始（这个基类从来没有正确的粒度），而是从几个小的方面来构建适当的存储库，从而将几个适当的方面混合为适当的存储库。这是问题的另外一种思考方式。它不一定总是正确的方式（不会有任何方式总是正确的），但它肯定值得更多的思考。我请我的朋友Aleksandar Seovi编写了面向方面（AO）的一个简介。

## 10.4 面向方面编程

作者：Aleksandar Seovi

在过去几年中，AOP成为软件开发社区的热门话题。AOP是在20世纪90年代由Gregor Kiczales[Gregor Kiczales]和他在Xerox PARC实验室的团队提出的，试图通过将系统的横切关注点封装到可重用的方面中来解决面向对象系统中的代码重复问题。

2001年发布的AspectJ[AspectJ]被认为是一个参考AOP实现，它引发了Java社区的大量活动，包括大量的备选AOP实现。

虽然与AOP相关的活动主要仍发生在Java社区中,但现在也有几个.NET AOP实现,我们可以利用它们来获得AOP的益处,这些实现包括Spring.NET AOP[Spring.NET]、Loom.NET[Loom.NET]和AspectSharp[AspectSharp]。

### 10.4.1 热门话题有哪些

AOP的基本前提是:尽管OOP与过程语言相比能够减少代码重复,但它仍然留下了大量重复代码。核心OOP特性(例如继承和多态性)以及文档化的设计模式(例如模板方法[GoF Design Patterns])有助于最小化核心应用程序逻辑中的代码重复。然而,实现横切关注点(例如日志或安全性)的代码仍然很难通过OOP本身实现模块化(如果不是说不可能话)。

尽管读者遇到的每篇AOP文章都会有一些老生常谈的示例,但我们仍将使用方法调用日志来显示AOP要尝试解决的问题。

假设我们已经实现了一个简单的计算器服务,它支持4种基本的算术运算:加、减、乘、除,如以下代码所示。注意,方法具有很高的简单性和可读性,因为它们只包含核心逻辑。

```
public class Calculator : ICalculator
{
    public int Add(int n1, int n2)
    {
        return n1 + n2;
    }

    public int Subtract(int n1, int n2)
    {
        return n1 - n2;
    }

    public int Divide(int n1, int n2)
    {
        return n1 / n2;
    }

    public int Multiply(int n1, int n2)
    {
        return n1 * n2;
    }
}
```

出于讨论的原因,假设新用户请求要求我们记录对服务方法的每个调用。应该记录的内容包括:传递给它的类名称、方法名称和参数值,以及它返回的值。

如果只使用OOP,那么这个看起来很简单地调用要求必须修改所有4个方法,并为它们添加日志代码。最终结果可能会像下面这样:

```
using System;

namespace LoggingAspectDemo
```

```
{
    public class LoggingCalculator : ICalculator
    {
        public int Add(int n1, int n2)
        {
            Console.Out.WriteLine("-> LoggingCalculator.Add
                (" + n1 + ", " + n2 + ")");
            int result = n1 + n2;
            Console.Out.WriteLine("<- LoggingCalculator.Add
                (" + n1 + ", " + n2 + ") returned " + result);

            return result;
        }

        public int Subtract(int n1, int n2)
        {
            Console.Out.WriteLine("-> LoggingCalculator.Subtract
                (" + n1 + ", " + n2 + ")");
            int result = n1 - n2;
            Console.Out.WriteLine("<- LoggingCalculator.Subtract
                (" + n1 + ", " + n2 + ") returned " + result);
            return result;
        }

        public int Divide(int n1, int n2)
        {
            Console.Out.WriteLine("-> LoggingCalculator.Divide
                (" + n1 + ", " + n2 + ")");
            int result = n1 / n2;
            Console.Out.WriteLine("<- LoggingCalculator.Divide
                (" + n1 + ", " + n2 + ") returned " + result);

            return result;
        }

        public int Multiply(int n1, int n2)
        {
            Console.Out.WriteLine("-> LoggingCalculator.Multiply
                (" + n1 + ", " + n2 + ")");
            int result = n1 * n2;
            Console.Out.WriteLine("<- LoggingCalculator.Multiply
                (" + n1 + ", " + n2 + ") returned " + result);

            return result;
        }
    }
}
```

可以看到，每个方法中都有类似的日志代码。

**说明** 我们知道,在不使用AOP的情况下,也有更好的方式来实现上面的示例,例如使用装饰模式[GoF Design Patterns],但这里主要是想介绍AOP,因此请保持耐心。

另外,装饰模式会影响使用者,因为它必须实例化装饰方法,而不是实例化被装饰的类,而且,如果需要添加多个服务的话,就必须实现多个装饰方法,并将它们连接成链。

这种方法有很多问题。

- 有大量的代码重复,因此永远不会成为好的设计。在这种情况下,可以通过创建用于日志的静态实用工具方法来简化重复代码,这些方法带有类和方法名及参数,或者返回参数形式的值,但重复本身是无法通过使用OOP消除的。
- 日志代码分散了人们对方法的核心逻辑的注意力,因此影响了方法的可读性,也不利于方法的长期维护。
- 实现核心逻辑的代码过于冗长了。本来可以采用简单单行代码形式的方法现在必须被分解为结果赋值及其返回值,这样做仅仅是为了将结果输入到日志代码中。
- 如果在某个时刻需要移除日志代码,那么我们将面临着一项令人望而生畏的任务:手工删除日志代码,或从类中注释掉所有日志代码。
- 如果决定记录完整的类名称,包括名称空间,那么必须查找所有包含日志代码的方法,并手工添加漏掉的名称空间信息。
- 如果决定记录异常(例如溢出或被零除),也面临同样的情况,必须添加一个try/catch代码块和一个声明,它将记录发生在整个代码库中很多地方的异常。

当然,有人会说,可以不使用简单的Console.Out.WriteLine声明,而使用高级日志解决方案,例如Log4Net [Log4Net]或Enterprise Library Logging Application Block[Enterprise Library Logging],这样就可以通过简单地更改配置设置来关闭日志功能,而不必修改任何类。

虽然这是完全正确的,但实际上忽略了本例中的重点。核心逻辑仍然与辅助的日志代码混合在一起,而且如果想对日志逻辑做出修改的话,仍然必须手工编写很多方法。此外,对于安全策略实施或事务来说,就无法像日志这样容易地打开或关闭它了。

使用AOP之后,我们能够对横切代码(例如日志或安全性)分离到几个方面中,而且很容易将这些方面应用于需要它们的所有类和方法。当然,除此之外,AOP还有大量的功能,它可以将业务规则从核心逻辑中分离出来,而且通过为现有对象添加状态和行为来更改它们。后面将举一些这方面的示例。

要理解的重要一点是:AOP并非OOP的替代,它只是OOP的一个扩展,弥补一些单独使用OOP方法时很难实现的功能。好的面向对象原则和实践仍然是适用的。实际上,现在有一个健壮OO设计比以往更加重要,因为它将使得方面的应用更容易。

## 10.4.2 AOP术语定义

在深入讨论并展示AOP示例之前,理解基本的AOP术语是很重要的,这并不像乍看上去那么困难。因此,这里提供以下术语的一些定义。

- 通知 (advice) 是要封装和重用的一个代码片段。例如, 日志代码可以实现为一个通知, 并且在需要时重用。有几种类型的通知, 例如前置通知、后置通知和环绕通知。
- 引入 (introduction) 在某种程度上是一种特殊类型的通知, 它只应用于类。它可以将新成员 (既可以是状态, 也可以是行为) 引入到现有类中, 并且可以在那些不支持多重继承的语言中实现多重继承的获益, 而且没有折中问题。(在一些语言中, 引入被称为 “mixin”。
- 连接点 (joinpoint) 是代码中可以应用通知的任意位置。理论上, 连接点几乎可以是任何对象, 例如实例变量、for 循环、if 语句等。但在实践中, 最常用的连接点是类、方法和属性。
- 切入点 (pointcut) 标识了一组连接点, 在这些连接点需要应用通知。例如, 如果需要将事务通知应用于所有标记了 Transaction 属性的方法, 那么必须声明一个标识了这些方法的切入点。
- 方面 (aspect) 将通知以及它们对应的切入点组织到一起, 这类似于类将数据和相关行为组织到一起的方式。

### 10.4.3 .NET 中的 AOP

理论先介绍到这里, 下面看一下 AOP 如何帮助我们解决开发过程中每天都会遇到的实际问题。

本节中的示例是使用 Spring.NET AOP 实现构建的, 因此读者需要下载并安装最新版本的 Spring.NET, 以便尝试这些示例。

**说明** 注意, 在本书编写时, 本节中的示例基于 Spring.NET 中已经规划的 AOP 实现支持。有关变更的信息, 可访问本书的网站 ([www.jnsk.se/adddp](http://www.jnsk.se/adddp)) 或查询官方的 Spring.NET 文档, 网址为: [www.springframework.net](http://www.springframework.net)。

#### 1. 使用方面来模块化日志代码

下面就来看一下如何利用 AOP 解决与前面的日志代码相关的问题。

创建一个日志通知。第一步是创建一个用于封装日志代码的通知。

```
using System;
using System.Text;
using AopAlliance.Interceptor;

namespace LoggingAspectDemo
{
    public class MethodInvocationLoggingAdvice
        : IMethodInterceptor
    {
        public object Invoke(IMethodInvocation invocation)
        {
            Console.Out.WriteLine("-> * +
```

```

        GetMethodSignature(invocation));
        object result = invocation.Proceed();
        Console.Out.WriteLine("<- " +
            GetMethodSignature(invocation)
            + " returned " + result);
        return result;
    }

    private string GetMethodSignature
        (IMethodInvocation invocation)
    {
        StringBuilder sb = new StringBuilder();
        sb.Append(invocation.Method.DeclaringType.Name)
            .Append('.')
            .Append(invocation.Method.Name)
            .Append('(');

        string separator = ", ";
        for (int i = 0; i < invocation.Arguments.Length;
            i++)
        {
            if (i == invocation.Arguments.Length - 1)
            {
                separator = "";
            }
            sb.Append(invocation.Arguments[i])
                .Append(separator);
        }
        sb.Append(')');

        return sb.ToString();
    }
}

```

此通知将记录类名称、方法名称和参数值，以及它所应用的方法的每个调用的返回值。

它使用ToString()方法来写入参数和返回值。这可能无法应用于所有情形，但对于大部分方法都是适用的，特别是当专门为应用程序类实现了ToString()的时候。如果发生异常，此通知将只是传播它，因此已有的任何错误处理代码都可以很好地工作。

最关键的地方是在Invoke()方法中。这里的基本思想是记录方法调用的开始，捕获将invocation.Proceed()调用的值赋给结果变量的赋值结果，并在返回它之前记录结果。对invocation.Proceed()的调用是至关重要的，因为这是导致计算器方法执行的原因。

GetMethodSignature()帮助方法通过检查调用对象来构建通用形式的方法签名。注意，GetMethodSignature()方法被调用两次，尽管看上去可以对通知进行一些优化，优化方法是将其结果赋给局部变量并将此变量作为替代。调用两次的原因在于，如果应用了通知的方法对参数

进行了修改,那么也可以看到这些修改。当记录对那些具有“外部”参数的方法的调用时,这特别有帮助。

现在已经定义了通知,那么如何记录那些应用了通知的方法所抛出的任何异常呢?读者可能已经猜到,这只需做出很少修改,即在通知的Invoke()方法中添加必要的逻辑。

```
public Object Invoke(IMethodInvocation invocation)
{
    Console.Out.WriteLine("-> "
        + GetMethodSignature(invocation));
    try
    {
        object result = invocation.Proceed();
        Console.Out.WriteLine("<- "
            + GetMethodSignature(invocation)
            + " returned " + result);
        return result;
    }
    catch (Exception e)
    {
        Console.Out.WriteLine("!! "
            + GetMethodSignature(invocation)
            + " failed: " + e.Message);
        Console.Out.WriteLine(e.StackTrace);
        throw e;
    }
}
```

将GetMethodSignature()更改为包含类的名称空间更简单,因此留给读者作为练习。

**应用日志通知。**创建通知只完成了一半工作。毕竟,要想发挥其作用就得使用它。我们需要从第一个代码清单开始就将日志通知应用于Calculator类,在Spring.NET中,这可以用两种方式来完成。

第一种方法是使用代码来应用它。虽然在这个特殊示例中这种方法很简单,但它并不理想,因为这里是手工将通知应用于特定对象,而不是基于方面配置并通过容器自动应用它。不管怎样,我们来看一下如何在代码中应用日志通知。

```
using Spring.Aop.Framework;
```

```
...
```

```
ProxyFactory pf = new ProxyFactory(new Calculator());
pf.AddAdvice(new MethodInvocationLoggingAdvice());
ICalculator calculator = (ICalculator) pf.GetProxy();
```

只需3行代码即可将日志通知应用于Calculator实例的所有方法,并获取一个对它的引用。现在可以调用其上的任何方法了,可以看到调用被正确记录下来。

Spring.NET用于AOP的声明式方法提供了更强的功能,而且不需要编写任何代码。然而,它要求使用Spring.NET来管理对象,这超出了本节的范围。出于讨论完整性的目的,这里显示一下本例中声明式方面定义的形式,但更多细节需要参考Spring.NET参考手册。



```
<aop:aspect name="MethodInvocationLoggingAspect">
  <aop:advice
    type="LoggingAspectDemo.MethodInvocationLoggingAdvice"
    pointcut=
      "class(LoggingAspectDemo.Calculator) and method(*)"/>
</aop:aspect>
```

这将使得Spring.NET容器将我们创建的通知应用于Calculator类的所有实例的所有方法。

只需对以上代码做出下面的修改即可将日志通知应用于LoggingAspect-Demo名称空间中的所有类的所有方法，这体现了声明式方法和AOP的强大功能。

```
<aop:aspect name="MethodInvocationLoggingAspect">
  <aop:advice
    type="LoggingAspectDemo.MethodInvocationLoggingAdvice"
    pointcut="class(LoggingAspectDemo.*) and method(*)"/>
</aop:aspect>
```

## 2. 为现有类添加状态和行为

我们要介绍的第二个示例使用引入和前置通知的组合来实现和执行对象锁定。

假设应用程序中有一些领域对象需要在会话中锁定。当对象被锁定时，如果其他会话试图修改其状态，则应抛出LockViolationException。

如果只使用OOP方法，需要直接为类添加它所需要的状态和方法，或者从实现核心锁定和解锁功能的基类继承它们。

直接添加的方法是不好的，因为这导致大量重复代码。可以使用第二个解决方案，条件是需要同步行为的类属于同一个类层次结构，并且所有类都必须支持锁定。如果它们属于不同的层次结构，或者在所有类中都引入锁定行为，那么就又退回到开始了。

更好的方法是利用AOP特性并通过声明方法只向需要锁定的类添加锁定支持。使用引入和前置通知的组合可以很容易实现此目的。

**实现锁定引入。**在本例中，第一步是创建一个用于向类添加状态和行为的引入。要创建引入，需要定义一个要引入的接口。

```
namespace LockableAspectDemo
{
    public interface ILockable
    {
        void Lock();
        void Unlock();
        bool IsLocked { get; }
    }
}
```

下一步是创建一个用于实现此接口的引入类。在本例中，假设正在编写一个ASP.NET Web应用程序，并且将使用HttpContext.Current.Session.SessionID作为用于锁定目的的会话ID。

```
using AopAlliance.Aop;
```

```
namespace LockableAspectDemo
{
    public class LockableIntroduction : ILockable, IAdvice
    {
        private bool isLocked;
        private string sessionId;

        public void Lock()
        {
            isLocked = true;
            sessionId = HttpContext.Current.Session.SessionID;
        }

        public void Unlock()
        {
            isLocked = false;
            sessionId = null;
        }

        public bool IsLocked
        {
            get
            {
                return isLocked && sessionId
                    != HttpContext.Current.Session.SessionID;
            }
        }
    }
}
```

至此，我们已经创建了一个引入。现在，需要创建一个通知，用于执行此引入所实现的锁定。实现一个锁定执行通知。在这个特殊示例中，我们不必像在日志例子中那样创建功能非常完备的环绕通知。通知所需的全部功能就是检查目标对象是否被锁定，并且在锁定时抛出一个异常。为实现此目的，可以使用一个更简单的前置通知，它不需要调用IMethodInvocation.Proceed()。

```
using System;
using System.Reflection;
using Spring.Aop;

namespace LockableAspectDemo
{
    public class LockEnforcerAdvice : IMethodBeforeAdvice
    {
        public void Before(MethodBase method, object[] args,
            object target)
        {
            ILockable lockable = target as ILockable;
            if (lockable != null && lockable.IsLocked)
            {
                throw new LockViolationException
                    ("Attempted to modify locked object.", target);
            }
        }
    }
}
```

新华书店  
PDG

```
    }
}
}
```

将锁定方面应用于领域对象。最后一步是将前面定义的引入和通知应用于要锁定的领域对象。对于本示例，假设想使Account类及其所有后代成为可锁定的。

首先，创建一个样例Account类。

```
namespace LockableAspectDemo.Domain
{
    public class Account
    {
        private string name;
        private double balance;

        public Account() {}

        public Account(string name, double balance)
        {
            Name = name;
            Balance = balance;
        }

        public virtual string Name
        {
            get { return name; }
            set { name = value; }
        }

        public virtual double Balance
        {
            get { return balance; }
            set { balance = value; }
        }

        [StateModifier]
        public virtual void Withdraw(double amount)
        {
            balance -= amount;
        }

        [StateModifier]
        public virtual void Deposit(double amount)
        {
            balance += amount;
        }
    }
}
```

简单起见，我们将省略后代，但可以假设还需要对CheckingAccount、SavingsAccount和MoneyMarketAccount类应用锁定，这三个类是从基类Account继承的。

有几个原因使得此类适合使用基于代理的AOP方法。首先,属性是使用私有字段和属性的getter及setter方法的组合实现的。如果使用公共字段,那么AOP代理就不可能拦截属性访问。

其次,所有属性和方法都被声明为虚拟的。这样,就可以使用Spring.NET来创建一个动态的代理,方法是继承Account类并注入拦截代码。如果方法是最终的,那么Account类必须实现一个接口,以便使用复合代理。如果接口和虚拟方法都不存在,就不可能为Account类创建代理并对其应用AOP通知。

最后一个要注意的问题是StateModifier属性的使用。定义此属性的主要目的是为了显示如何基于属性来应用通知,但属性还有许多其他用途。它提供了额外的元数据,使得我们可以很明显地看出哪些方法修改了对象的状态。

也可以在属性的setter方法前面使用StateModifier属性,但我认为这有些多余。除此之外,在属性中省略StateModifier可以显示复合切入点是如何定义的。

虽然可以仅使用代码来配置锁定方面,但这种方法有些繁琐,因此在本例中将只显示声明式方法。

```
<aop:aspect name="LockEnforcementAspect">
  <aop:pointcut name="StateModifiers"
    expression="class(Account+)
      and (setter(*) or attribute(StateModifier))"/>
  <aop:advice type="LockableAspectDemo.LockableIntroduction"
    pointcut="class(Account+)"/>
  <aop:advice type="LockableAspectDemo.LockEnforcerAdvice"
    pointcut-ref="StateModifiers"/>
</aop:aspect>
```

大多数声明应该都很简单。值得注意的方面有如下这些。

- 在切入点的定义中,类名称后面的加号指定了切入点应该匹配指定的类及其派生类。
- 表达式setter(\*)用于匹配所有属性setter方法,不管它们的类型和名称如何。
- 表达式attribute(StateModifier)用于匹配所有用StateModifier属性标记的成员。

最后,需要创建一个客户,它将使用被引入的ILockable接口(就好像此接口是由Account类直接实现的一样)。

```
using LockableAspectDemo.Domain;

namespace LockableAspectDemo.Services
{
    public class AccountManager
    {
        public void TransferFunds(Account from, Account to
            , double amount)
        {
            ILockable acctFrom = from as ILockable;
            ILockable acctTo = to as ILockable;

            try
            {
```



```

        acctFrom.Lock();
        acctTo.Lock();

        from.Withdraw(amount);
        to.Deposit(amount);
    }
    finally
    {
        acctFrom.Unlock();
        acctTo.Unlock();
    }
}
}
}

```

可以看到，使用AOP可以真正模块化对象锁定特性，并在需要的地方应用它，而没有纯OOP方法的缺点。

**说明** 虽然从技术上讲，前面的示例与使用锁定声明从多个线程同步访问acctFrom和acctTo没有什么区别，但在这两种方法的语义之间却有很大的区别。使用ILockable时，可以在多个HTTP请求（而不仅仅为单个请求）期间保持锁定。

### 3. 将业务规则移到方面中

最后一个示例是某种程度上的高级AOP使用。我们并不推荐在一开始就尝试将所有业务规则都移到方面中。最好的方法是逐渐将AOP引入到项目中，利用它来处理明显的横切关注点，例如日志、安全性和事务。然而，理解其全部的潜力是很重要的，以便在进一步熟悉之后有效地应用它。

每个业务应用程序都有各种业务规则，这些规则通常被嵌入到应用程序逻辑中。虽然这种嵌入对于大多数规则 and 应用程序来说是好的，但也降低了灵活性，而且在业务规则发生变化时，通常需要修改代码。通过将一些业务规则移入方面中，可以更容易定制应用程序。与前面的例子不同，将业务规则移入方面中的主要原因不是为了删除代码重复，而是为了提高应用程序的灵活性。

最适合移入方面中的是那些导致在实现核心逻辑的同时也需要实现二级逻辑的规则。例如，前面示例中的Account.Withdraw()方法的核心逻辑是减少账户余额。二级逻辑是保证最低余额需求，或者当余额低于一定数额时用电子邮件通知账户持有者。

下面看一下为Account类添加了最低余额保证和余额警告时的情况。

```

using System;

namespace BusinessRulesDemo.Domain
{
    public class Account
    {
        private AccountHolder accountHolder;
        private string name;
        private double balance;
    }
}

```

```
private double minimumBalance;
private double alertBalance;

// Spring.NET IoC injected collaborators
private INotificationSender notificationSender;

public Account() {}

public Account(string name, double balance)
{
    Name = name;
    Balance = balance;
}

public virtual Person AccountHolder
{
    get { return accountHolder; }
    set { accountHolder = value; }
}

public virtual string Name
{
    get { return name; }
    set { name = value; }
}

public virtual double Balance
{
    get { return balance; }
    set { balance = value; }
}

public virtual double MinimumBalance
{
    get { return minimumBalance; }
    set { minimumBalance = value; }
}

public virtual double AlertBalance
{
    get { return alertBalance; }
    set { alertBalance = value; }
}

// injected by Spring.NET IoC container
public virtual INotificationSender NotificationSender
{
    set { notificationSender = value; }
}

[StateModifier]
public virtual void Withdraw(double amount)
{
    if (balance - amount < minimumBalance)
    {
```



```

        throw new MinimumBalanceException();
    }

    balance -= amount;

    if (balance < alertBalance)
    {
        notificationSender.SendNotification
            (accountHolder.Email,
             "Low balance",
             "LowBalance.vm", this);
    }
}

[StateModifier]
public virtual void Deposit(double amount)
{
    balance += amount;
}
}
}

```

虽然读者可能认为最低余额检查应该保留在Account类中，但显然通知发送代码不是Withdraw()方法的主要功能，因此非常适合将它移到方面中。下面对它进行重构。

创建一个用于发送通知的通知。首先，创建一个通用的通知，它用于发送通知。

```

using System.Reflection;
using BusinessRulesDemo.Domain;
using Spring.Aop;

namespace BusinessRulesDemo
{
    public class AccountBalanceNotificationAdvice :
        IAfterReturningAdvice
    {
        private INotificationSender sender;
        private string subject;
        private string templateName;

        public INotificationSender Sender
        {
            get { return sender; }
            set { sender = value; }
        }

        public string Subject
        {
            get { return subject; }
            set { subject = value; }
        }

        public string TemplateName
    }
}

```



```

    {
        get { return templateName; }
        set { templateName = value; }
    }

    public void AfterReturning(object returnValue,
        MethodBase method, object[] args, object target)
    {
        Account account = (Account) target;
        if (account.Balance < account.AlertBalance)
        {
            sender.SendNotification(
                account.AccountHolder.Email,
                subject, templateName,
                target);
        }
    }
}

```

这个示例中使用了一个后置通知，它检查新余额是否低于警告余额水平，如果是，则发送一个通知。我们还在通知类上定义了几个属性，目的是为了显示如何使用Spring.NET IoC容器来配置通知。

在这个示例中，通知与Account类之间的紧密耦合不会产生问题，因为我们知道通知将只应用于Account类的实例，不管怎样，我们只是使用通知将通知发送规则移到Account类的核心逻辑之外。

这实际上为我们带来了灵活性。如果将这个应用程序出售给另一位客户，而他要求删除通知(notification)，那么可以很容易满足这个要求，方法就是从配置文件中删除通知应用(advice application)。类似地，如果客户要求实现以下逻辑：当余额低于允许的最小数额时，从信贷额度账户中自动转账，那么就可以简单地实现另一个通知(advice)，并将它应用于Account类，甚至可以同时保留通知(notification)通知。

**应用通知。**就像在前面的示例中一样，我们将只显示声明方面配置，以完成这个示例。

```

<aop:aspect name="BalanceNotificationAspect">
  <aop:advice
    type="BusinessRulesDemo.AccountBalanceNotificationAdvice"
    pointcut="class(Account*) and method(Withdraw)">
    <aop:property name="Sender" ref="NotificationSender"/>
    <aop:property name="Subject" value="Low Balance"/>
    <aop:property name="Template"
      value="~/Templates/LowBalance.vm"/>
  </aop:advice>
</aop:aspect>

```

这里，唯一值得指出的是可以使用Spring IoC特性来配置通知，就像用它来配置其他任何对象一样。本示例用它来指定NVelocity消息模板的通知主题和文件名，以及要使用的通知发送者的实例。



#### 10.4.4 小结

AOP能够帮助我们解决一些日常遇到的问题。虽然.NET中尚未提供语言级和CLR级的AOP支持,但现在.NET开源社区中的一些工具提供了一组很好的AOP特性。

或许开始学习更多AOP知识的最佳方式是下载一些这样的工具,并试用它们,甚至可以将它们引入到实际项目中,首先可以从一些基本用法开始,例如日志或剖析操作。当掌握了更多AOP知识并更熟悉所选择的工具后,我们将发现面向方面技术可以提供更多帮助的领域。

感谢Aleksandar!

这样,当将AOP应用于前面讨论的存储库设计问题时,一些需要为CustomerRepository引入的接口可能就是GetByIdable、Deletable等。

这篇讨论相当清晰。当然,AOP还有一些较明显的应用方面,例如常规的日志例子。正如Aleks指出的,这样的例子有些老生常谈,但毕竟是解决常见问题的一种非常有用的技术。

#### 10.5 小结

本章介绍了三种不同且非常重要的设计技术,这三种技术是读者应该了解或开始研究的。最后一章将讨论表示服务,通常这也是一个非常重要的方面,在过去、现在和将来都很重要。



**数**据库和用户界面（UI）的测试是一项困难的任务。第6章中讨论了数据库测试，但并未介绍UI测试，本章会介绍大量这方面的内容。

我们还将讨论一些比常规方法更符合DDD原则的UI设计，至少这些设计比.NET中的快速应用开发（RAD）更符合DDD原则。在这个过程中，我们将考查模型-视图-控制器模式（MVC）和模型-视图-表示器模式（MVP）。

我们还将看一下其他类型的映射，本章不再关注对象关系映射，而是UI映射中的对象-对象映射。

首先，为了将读者引入正确的上下文，这里先给出我的老朋友Byström写的一段“提前结语”。读者可能会问：“什么是提前结语？”就是过早的或预先的结语。我请Dan为本书写了一个简短结语。他所写的内容很好，也很恰当，但相比之下，更像是最后一章的引言，而不像是全书的结尾。因此，下面就奉上Dan的提前结语。

## 11.1 提前结语

作者：Byström

现在你坐在这里，欣赏着你的工作成果：你所见过的最完美的领域模型。你终于完成了当初看上去不可能完成的任务，将客户的大部分业务映射到领域模型。你不仅剔除了大量特殊情况，而且还向客户展示了超乎他们想象的更常用的业务视图。

如果说这还不够好，那么界面是整洁的，代码甚至更加整洁。无论谁使用这个领域模型，都将对你的成就赞叹不已，并暗地里嫉妒你。难道还有什么不满足吗？现在完全可以享受快乐了！

“等一下，上面说的是无论谁？那么无论谁到底是谁？”

你说：“嗯，是指使用.NET语言的任何程序员。”（或者你会说是指所有Java开发人员，或其他的一些开发人员，具体取决于你的实现。）

“那么成千上万的程序员会使用你的领域模型？这真是非同寻常啊！”

“是的，这不是很好吗？”你回答说，“我已经把大部分业务模型和整个数据存储问题封装在了这个模型中，这样就可以在不涉及这些繁杂细节的情况下简单地使用它了。”

“你确定除了程序员之外没有其他人对使用你的领域模型感兴趣吗？”

“您的意思是？”

“嗯，我只是好奇……你不觉得有很多用户会从你的工作中受益吗？事实上，难道不是用户订购你的领域模型吗？”

“对，是这样，但您到底想知道什么呢？”

“你不认为需要一个用户界面吗？以便使这些用户能够与你的全新领域模型进行交互？”

“哦……”

我现在有很好的机会与Jimmy进行项目合作，他负责领域模型和O/R映射，我则负责UI。作为必然的幸运组合，我们都感觉自己是成功者，而认为另一个人只是配角。

当我开发应用程序时，我认为UI是核心部分，因为如果没有UI，应用程序将无法使用。在大部分应用程序中，数据表示和数据存储也是如此。实际上这只是角度、个人品味和兴趣的问题。

一些程序员很容易将直接的数据库访问混合到UI中。一个非常典型的示例就是经典的数据绑定，也就是将用户控件直接绑定到数据库字段。每次我尝试这样的绑定时，都下决心不再这样做了！（另一方面，我非常希望发现一种直接绑定到领域模型的简单方式。）

随着应用程序的增长和维护的需要，逻辑层作为一个福音出现了。

将UI数据从逻辑上分离到一个单独视图（即表示模型[Fowler PoEAA2]）中是另一个伟大的进步。

实际上，这种方式对于简单UI并没有太大价值，但我肯定会将它用于UI中的复杂部分，尽管它不会一直保持完全纯净。我很无奈项目的预算都是有限的。（另一方面，如果问题很简单，那么应该使用简单的解决方案。）

事实上，当我第一次遇到SQL数据库时（在此之前，我总是编写自己的存储机制，在可能的时候，我现在仍会这样做），我立即认识到在数据库和UI之间需要某种类型的层，尽管我用了10年时间才遇到领域模型。

因此，领域模型是UI程序员的一个伟大工具。但它实际上还刚刚开始，乐趣也从这里开始！太好了！感谢Dan！

下面，就让我们在领域模型的丰富上下文中考虑与表示服务有关的更多内容，首先从富客户的角度开始，重点关注测试。

Martin Fowler认为[Fowler PoEAA]，MVC模式是最容易被误解的模式。同时，它的功能非常强大，而且被广泛应用。该模式的目的是和基本思想并不是很难掌握，像平常一样，关键在于细节。Christoffer Skjoldborg将为我们澄清如何应用MVC，而且他将通过详细示例来讨论它。

## 11.2 模型-视图-控制器模式

作者：Christoffer Skjoldborg

模型-视图-控制器模式（MVC）的目的是将UI行为分解为不同的片段，以便提高可重用性和可测试性。三个片段分别是视图（View）、模型（Model）和控制器（Controller）。

模型是实体状态的某种表示。它的形式多种多样，从简单的数据结构（XML文档、数据集

和数据传输对象)到功能完备的领域模型。

视图应该仅包含与“在屏幕上绘制像素”有关的逻辑。当应用MVC时,主要设计目标是尽可能保持视图“是哑的(dumb)”。基本原理是视图或屏幕实际上需要用人眼来测试。与自动的单元测试相比,这是个易出错且代价高昂的过程。如果能够将视图中的逻辑减到最少,那么引入那些只能靠手工测试才能检测到的bug的风险也就降到最低了,换句话说,我们可以最小化必须执行的手工测试工作。

最后,控制器是一种“粘合剂”,作为MVC的核心,它将模型和视图联系在一起。当用户与视图进行交互时,控制器接收消息,并将这些消息转换为要在底层模型上执行的操作,然后再相应地更新视图。

MVC模式发源于Smalltalk社区,它的诞生可以追溯至1970~1980年代[MVC History]。

如前所述,引入MVC的主要原因之一是提高可测试性。其他获益还包括

- 显式地将代码从视图中提取出来的过程使得系统结构更多地依赖于UI代码。很多现代工具允许仅通过一些点击操作即可快速添加各种事件代码,但这样就缺少了周详的思考。通过部署MVC模式,设计人员必须付出更多努力,并且要明确地指定要将哪些对象放到UI中。
- 很容易更改外观,或支持全新的UI甚至是平台。有时,MVC的重用价值被夸大了,特别是对于跨平台(富客户/Web/手持设备)可移植性来说。通常,为了充分利用目标平台的功能,需要对控制器进行大量修改。
- 不同团队成员可以同时视图和控制器上工作(即使使用了文件锁定源代码控制系统),因为它们现在很容易分离。

### 11.2.1 示例: Joe 的 Shoe Shop 程序

MVC模式是一种宽松的设计模型,这意味着使用它时要做出大量的实现细节决定。这实际上是一个很麻烦的过程,因此在深入讨论MVC的各种变体之前,这里先给出一个示例实现,将它作为我们学习的开始。像其他书中的示例一样,这个示例非常简单,便于读者从总体上理解它。但是,它与相同复杂度的真实应用程序中的实际做法差不多,而且当这个示例完成后,我们将细化一些需要在示例基础上进行改进的关键领域,并提供一些改进建议。

这里使用C#和Windows Forms,但代码足够简单,即使读者不熟悉C#和Windows Forms,也很好理解。

考虑Joe的鞋店(Shoe Shop程序)的产品目录(见图11-1)。Joe只卖两种类型的鞋:舞鞋和安全鞋。舞鞋有一个硬度属性,安全鞋有一个最大压力属性。所有鞋都有ID、名称和价格。Joe用来维护目录的屏幕可能像下面这样。

可以对鞋执行浏览、添加、删除和更新操作。当保存一个新鞋后,ID不可再更改。当Joe选择一款舞鞋时,他希望看到鞋的硬度,当选中一款安全鞋时,希望看到鞋可以承受的最大压力。虽然这个视图可能不是读者所见过的最复杂的视图,但它确实有几个重要的常见概念,例如网格以及需要基于屏幕其他部分来动态修改的部分。

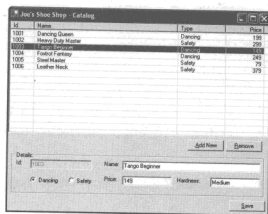


图11-1 Joe的鞋的ShopCatalog屏幕

为了将逻辑从视图中分离出来，首先粗略地确定需要处理哪些事件和用户交互。在这个示例中，需要处理的事件如下。

- LoadView，用于在第一次启动时加载数据，并实例化整个视图。
- SelectedShoeChanged，当用户更改选中的鞋时，执行此操作。
- AddNewShoe，当用户单击Add New按钮时，执行此操作。
- RemoveShoe，当用户单击Remove按钮时，执行此操作。
- ChangeType，用于更改变量标签（Hardness或Max Pressure）的描述，并清除值。
- Save，保存用户做出的更改。

所有这些事件都是控制器类的主要候选方法。尽管MVC模式正式规定控制器应接收事件，并在视图上执行操作，但更实用的方法是让视图订阅事件，然后把事件的处理委托给控制器。（这是基本MVC模式的一种稍微变化的形式，Martin Fowler将其称为模型-视图-表示器[Fowler PoEAA2]。）

现在，我们已经清楚地知道应将哪类逻辑放到控制器中，但仍需要令控制器能够与视图进行对话。记住，应尽可能使视图是“哑”的，因此令控制器执行所有困难工作，而只是为视图分配一些不需要进一步处理的简单指令。我们通过定义一个ICatalogView接口来实现此目的，视图必须实现此接口。

```
public interface ICatalogView
{
    void SetController(CatalogController controller);
    void ClearGrid();
    void AddShoeToGrid(Shoe shoe);
    void UpdateGridWithChangedShoe(Shoe shoe);
    void RemoveShoeFromGrid(Shoe shoe);
    string GetIdOfSelectedShoeInGrid();
    void SetSelectedShoeInGrid(Shoe shoe);
    string Id{get;set;}
```

```

    bool CanModifyId{set;}
    string Name{get;set;}
    ShoeType Type{get;set;}
    string Price{get;set;}
    string VariantName{get;set;}
    string VariantValue{get;set;}
}

```

乍看上去，ICatalogView接口的一些方法名称似乎公开了一些很复杂的功能，但实际上功能只包括以下操作：在屏幕上绘制像素，与各种UI部件进行交互，启用/禁用按钮，等等。以下代码清单显示了一个实现子集。

**说明** 本节将集中讨论加载一个带有鞋清单的视图，所有与视图、控制器和测试有关的代码也均围绕这一主题。

如果想查看完整的数据库，可以从本书网站下载它，网址为：[www.jnsk.se/addp](http://www.jnsk.se/addp)。

```

public void SetController(CatalogController controller)
{
    _controller=controller;
}

public void ClearGrid()
{
    // Define columns in grid
    this.grdShoes.Columns.Clear();
    this.grdShoes.Columns.Add("Id", 50
        , HorizontalAlignment.Left);
    this.grdShoes.Columns.Add("Name", 300
        , HorizontalAlignment.Left);
    this.grdShoes.Columns.Add("Type", 100
        , HorizontalAlignment.Left);
    this.grdShoes.Columns.Add("Price", 80
        , HorizontalAlignment.Right);

    this.grdShoes.Items.Clear();
}

public void AddShoeToGrid(Shoe shoe)
{
    ListViewItem parent;
    parent=this.grdShoes.Items.Add(shoe.Id);
    parent.SubItems.Add(shoe.Name);

    parent.SubItems.Add(Enum.GetName(typeof(Shoe.ShoeType),
        shoe.Type));

    parent.SubItems.Add(shoe.Price.ToString());
}

public void SetSelectedShoeInGrid(Shoe shoe)

```



```

{
    foreach(ListViewItem row in this.grdShoes.Items)
    {
        if(row.Text==shoe.Id)
        {
            row.Selected=true;
            break;
        }
    }
}

```

SetController()可以通知视图必须将哪些事件转发给控制器实例，并且所有事件处理程序只是简单地调用控制器上的相应“事件”方法。

注意，ICatalogView接口上的几个方法使用了Shoe对象。这个Shoe类是一个模型类，因此让视图了解模型是一种很好的思想。在本例中，Shoe是一个非常简单的领域类，它不带有行为，而实际的领域模型在领域类中可能有更多功能。

```

public class Shoe
{
    public enum ShoeType
    {
        Dancing=1,
        Safety=2
    }
    public string Id;
    public string Name;
    public ShoeType Type;
    public decimal Price;
    public string VariantValue;

    public Shoe(string id, string name, ShoeType type,
        decimal price, string variantValue)
    {
        Id=id;
        Name=name;
        Type=type;
        Price=price;
        VariantValue=variantValue;
    }
}

```

**说明** 读者可能奇怪为什么这里使用了单数的“Shoe”，而不是“Pair of Shoes”或类似于类名称的形式。这里只是采用了鞋类专业人员所使用的语言。我注意到他们总是说“这只鞋”和“那只鞋”，而非专业人员会说“这双鞋”或“这些鞋”。还有一个原因就是Shoe比PairOfShoes更适合作为类名称。

现在有了两个要处理的部分：视图和模型。它们主要都与状态有关。模型以结构化格式保持内存中状态，视图则保持可视表示及状态的修改方式。让我们通过控制器将它们连接到一起。

我们已经有了需要处理的事件类型的粗略想法，但为了了解细节，这里将从编写每个事件的测试开始。测试中将使用NMock [NMock]。

代码清单在下面给出，但这里首先对一些部分给出解释。getTestData()是一个私有帮助方法，它在内存中建立鞋的列表。在测试套件的其余部分中，此列表将用作模型。

如前所述，控制器上的每个方法都有一个测试，这里重点关注加载带有数据的视图，因此，所显示的测试是LoadView\_test()。

首先创建用于在测试中表示视图的viewMock，然后创建鞋的列表 (Model)，最后创建要测试的控制器。注意，在其构造方法中视图的模拟实现是如何输入到控制器中的。它将在这个模拟实现上操作 (就像在真实视图上操作一样)，因为ICatalogView接口就是控制器对视图所知道的全部。

接下来通知viewMock要进行哪些调用，以及带有哪些参数。这些参数表示需要哑视图从控制器接收的“订单”。然后调用要在控制器中测试的事件处理程序，最后请求viewMock验证是否满足期望。在后台，viewMock将确定每个期望的调用是否都已收到以及是否带有正确的参数。代码清单如下：

```
private IList getTestData()
{
    IList shoes=new ArrayList();
    shoes.Add(new Shoe("1001","Dancing
    Queen",Shoe.ShoeType.Dancing,199m,"Soft"));

    shoes.Add(new Shoe("1002",
    "Heavy Duty Master",Shoe.ShoeType.Safety,299m,"500 lbs"));

    shoes.Add(new Shoe("1003",
    "Tango Beginner",Shoe.ShoeType.Dancing,149m,"Medium"));

    return shoes;
}

[Test]
public void LoadView_test ()
{
    DynamicMock viewMock=new DynamicMock(typeof(ICatalogView));
    IList shoes=getTestData();
    CatalogController ctrl = new CatalogController
        ((ICatalogView)viewMock.MockInstance, shoes);

    // Set expectations on view mock
    viewMock.Expect("ClearGrid");
    viewMock.Expect("AddShoeToGrid",new object[]{shoes[0]});
    viewMock.Expect("AddShoeToGrid",new object[]{shoes[1]});
    viewMock.Expect("AddShoeToGrid",new object[]{shoes[2]});
    viewMock.Expect("SetSelectedShoeInGrid"
        , new object[]{shoes[0]});

    ctrl.LoadView();
}
```



```
viewMock.Verify();
}
```

所有测试都完成之后（尽管上面的代码清单只显示了一个），剩下的工作就是显示Catalog-Controller的实际实现（参见以下代码清单）。

```
public class CatalogController
{
    ICatalogView _view;
    IList _shoes;
    Shoe _selectedShoe;

    public CatalogController(ICatalogView view, IList shoes)
    {
        _view=view;
        _shoes=shoes;
        view.SetController(this);
    }

    public void LoadView()
    {
        _view.ClearGrid();
        foreach(Shoe s in _shoes)
            _view.AddShoeToGrid(s);

        _view.SetSelectedShoeInGrid((Shoe)_shoes[0]);
    }
}
```

### 11.2.2 通过适配器简化视图界面

在Joe的Shoe Shop程序中，ICatalogView为控制器需要访问的每个可能的细节公开了一个属性。这样做是可以的，但如果有10个输入框的话，会得到非常大的界面，而且需要控制每个输入框的可见性、是否启用、字体、背景颜色、前景颜色、边框样式，等等。为了避免这个问题，可以为每个UI控件类型创建一个适配器/包装器[GoF Design Patterns]。TextBox的适配器可以公开前面提到的属性，这样视图界面就只需为每个文本框公开TextBoxAdapter数据类型的属性。在后台，视图将在实例化期间为适配器提供对实际UI控件的引用，而且适配器将只是简单地将它接收到的调用委托给实际的实现。

### 11.2.3 将控制器从视图解耦

有时将控制器从视图中解耦出来是有优势的。这有几种方式。一种简单的解决方案是为控制器创建一个接口，并使视图只知道该接口，而不知道实际的实现。一种更复杂的方式是令视图向匿名订阅者发出事件（观察者模式[GoF Design Patterns]）。我不是特别喜欢这种方法，因为我想象不出为什么视图需要将用户交互传递给多个订阅者，更不用说未知类型的订阅者了！如果不同的控制器或其他视图需要知道特定视图的更改，那么该视图的控制器应该通知它们。在必要的时候，这可以通过事件来实现，重点是控制器应百分百控制这对哪些对象是可见的。

不管实现细节如何,将控制器从视图解耦的优点是可以改变视图背后的控制器。当视图可以在很多不同上下文中出现时,这可能非常有用,而且如果总是强制将某一特定控制器用于视图,那么就没有太大意义了。这方面的一个例子是针对特殊客户将特殊的逻辑应用于视图。在这种情况下,可以根据系统客户的不同,通过一个工厂方法来提供适当的控制器。

### 11.2.4 将视图和控制器结合起来

更复杂的视图通常由多个较小的视图构成。例如向导,用户通过前进或后退按钮在一组视图中进行导航,甚至可以一次跳跃多个步骤。在这个过程中可以保存信息,但直到用户在最后一个屏幕上单击保存之前,保存命令不会真正提交。

另一个示例就是主/从(Master/Detail)视图,其中对从视图的更改可能导致主视图中的变化。

从技术上讲,这两个示例都可以被编码为一个巨大的控制器,但这样做极有可能导致混乱,并且可重用性也降至最低了。例如,我们可能需要在应用程序的其他地方使用向导中的一些步骤,或者在显示主视图时不想显示从视图。这里的解决方案显然是将视图和控制器分割为更小的部分。(“如果要吃掉大象,必须将它分割为小块。”)在向导中,可以有一个FlowController,它负责控制向导的向前和向后的基本流,而且每个步骤都可以有其自己的视图和控制器。

在主/从模式中,可以通过从视图的控制器来引发更改事件。这样,任何其他控制器(包括主视图的控制器)都可以订阅此事件。

### 11.2.5 是否值得使用 MVC

应用MVC模式的关键获益是提高了UI代码的可测试性。其次,它强制对UI的编码和设计过程采用一种结构化的方法,这本身就可以得到更整洁的代码。可以说,我们被强制进行更多的思考。

另一方面,应该注意能够对每行代码进行测试并不意味着应该这样做。利用MVC模式,可以为每个视图编写扩展的测试套件,而且前几次得到UI逻辑测试的绿条当然会让人感觉很舒服。然而,对于一些非常简单的系统来说,如果由于系统中各个部分之间的互相依赖性很低而使得未来更改不会遭到破坏,那么测试的价值并不大。特别是当这些UI测试套件为代码库施加了巨大的维护开销的时候。每次对UI进行很小的修改时,都不得不对相应的UI测试做出大量修改。

感谢Chris!

无论是否想通过严格的单元测试来测试UI,我认为Christoffer刚刚向我们展示的技术都是非常宝贵的,它使得Windows Forms应用程序的视图部分更简捷,从而只编制窗体,并且将一些表示逻辑提取到其自己的层中。

这也是下一节的主题,但上下文换做了Web。Ingemar Lundberg讨论如何将MVP模式[Fowler PoEAA2]用作测试Web应用程序的工具。

## 11.3 测试驱动的 Web 窗体

作者: Ingemar Lundberg

如何对GUI进行单元测试和应用TDD?

本节将简单介绍对GUI进行单元测试的总体思想。在这个过程中将展示一些诸如自分流[Feathers Self-Shunt]和模拟这样的技术。但是,本节不会介绍TDD本身。自分流(简单来说)是一种将测试类也用作桩或模拟的技术。

### 11.3.1 背景

典型的GUI代码由一系列回调方法组成,它们作为用户执行一些GUI操作(例如按下按钮)的结果被调用。通常,回调方法对领域对象执行某种操作(甚至可能是执行SQL)。问题在于回调方法是从WinForms或ASP.NET这样的环境中被调用的,而这些环境无法与JUnit测试连接到一起。

目标是尽可能减少测试不到的代码。第一步是尽可能将代码移到回调方法之外。如果代码不在回调方法中,就可以对其进行单元测试。

但这还不够。GUI的状态如何?我们知道,如果用户选中某一行,那么应该禁用一些按钮。这是无法通过简单地将代码移到回调方法之外解决的。不要误会我的意思——它是很好的第一步。

为了进一步推进测试,需要使用最有效的工具:抽象。我们需要一个交互模型,而且需要窗体的抽象。现在,我们暂停对GUI的讨论,开始讨论窗体,或更具体地说是ASP.NET Web窗体。但是,这种技术也适用于Windows Forms,它甚至是与平台无关的。

### 11.3.2 一个示例

我认为,到目前为止这是编程中最困难的部分——找到一个用于说明重点的问题场景,而又不会将读者引入歧途。我需要用个示例来说明GUI状态的变化,换句话说,当用户执行某些操作时,需要禁用一些对象。

我们瑞典人的传统是只在周六才让孩子们吃糖果。假设我有一盒巧克力棒、一盒棒棒糖和半盒袋装花生。孩子们可以选择三个糖果,但不能有两个以上的糖果是同一种类的。简化的模型就是我们永远不会用完糖果。

这无疑是一个不太聪明的示例,但希望它能够说明问题。图11-2显示了最终结果。我本来想显示最初用铅笔画的GUI草图,但这个图也是草图。左侧的窗格中列出了所有类型的糖果。用户(所有年龄的孩子)可以从这个窗格中挑选星期六糖果,选中的糖果显示在右侧的窗格中。可以看到,这里已经不允许再挑选另一块巧克力了,而只能选择一块棒棒糖或一袋花生。

Chocolate	Chocolate
Lollipop >>	Chocolate
Peanuts >>	Chocolate

图11-2 选项、限制和选择

这里只给出很少的样式,读者不必在意这一点(而且我们不发表对这个主题的任何观点)。

### 11.3.3 领域模型

为了继续讨论，需要一个领域模型，如图11-3所示。

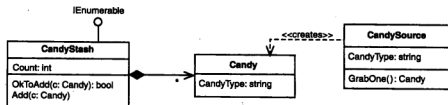


图11-3 领域模型

我已经对这个模型进行了单独的测试。OkToAdd() 检查前面给出的业务规则。可以添加三项，但只能有两项是相同类型的 (CandyType)。这个示例很简单。

### 11.3.4 GUI 的 TDD

重点是最后一步。在开始之前，要提醒读者的是，即使引入窗体的抽象，也会有一个非常具体的工作方式视图，如图11-4所示。

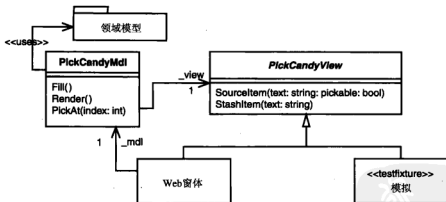


图11-4 UI 的模型

#### 1. 显示糖果类型

当第一次显示窗体时，需要用所有糖果类型的名称填充左侧的窗格。可以从所有糖果源中挑选。测试代码如下。（基于前面的经验，这里将迈出一大步，可以将它称为模式。）

```
[Test] public void FillSourcePanel()
{
    mdl.Fill();
    mdl.Render();
    // using A=NUnit.Framework.Assert
```

```

    A.AreEqual(SrcPanel, "a> b> c> ");
}

```

这个测试无法编译。需要为PickCandyMdl添加Fill()和Render()。这很容易，只需添加两个空的公共方法，一个名为Fill()，另一个名为Render()。但如何表示测试呢？要断言什么？断言是以非常抽象的方式来表示“窗体”的形式。字符串SrcPanel用来替代最后要得到的HTML表（窗体中左侧的窗格）。

那么SrcPanel是如何得到的？SrcPanel是测试固件(类)中的一个字符串，PickCandyView的自分流[Feathers Self-Shunt]实现负责操纵该字符串。为视图创建的模拟实现采用如下模式（此模式是可重用的）。

```

[TestFixture] public class PickCandyTests: PickCandyView
{
    void PickCandyView.SourceItem(string text, bool pickable)
    {
        SrcPanel += text + (pickable ? ">"
            : string.Empty) + " ";
    }

    PickCandyMdl mdl;
    string SrcPanel;

    [SetUp]
    protected void Setup()
    {
        mdl = new PickCandyMdl();
        SrcPanel = string.Empty;
    }
    ...snip...
}

```

上段代码是可编译的，但它得到红条。那么丢失了什么东西？窗体模型（即类型PickCandyMdl的mdl）需要知道其视图。这很简单，为模型添加SetView()方法。（顺便提一下，这是使用PickCandyView.SourceItem()形式的“接口实现”，例如{interface name}.{member name}时出现的几种情况之一。）

#### 接口方法使用私有实现还是公共实现

方法实现只能在类上存在。如果类Foo实现了一个接口Bar，那么此接口的方法就是该类的一部分。在大多数情况下，我会将Bar的成员实现为公共的。记住，类的接口本身就是其全部公共成员。Foo是一个Bar，因此Bar的成员是Foo的一部分。在另一些情况下，将接口的成员声明为私有成员的目的是解决命名冲突，例如当对一个模拟实现执行自分流的时候。这里实际上没有把接口（PickCandyView和CandySourceRetriever）看作是测试固件的接口的一部分。

窗体模型需要知道如何获取所有Candy Source。一种简单的方法是为模型提供一个Candy Source的列表。但这里将使用一种略微复杂的方式，它可以在一般情形中提高可测试性。这里，

让窗体模型（或者是包/程序集）拥有一个表示所需的检索操作的接口。

```
public interface CandySourceRetriever
{
    IList Retrieve();
}
```

### 为什么没有为接口使用I-name惯例

考虑一下，此惯例有什么优点？它出自这样一种环境：接口概念本身是一种惯例和规则，而不是一个优秀的语言概念（midl或多或少地改变了这一点）。当尝试实例化一个（新的）接口时，C#编译器将抽象类和接口作为同一种事物来对待（至少我的版本如此）。

```
public interface Bar {}
public class Foo { public static void Main() {new Bar();} }
ingo.cs(2,47): error CS0144: Cannot create an instance of the
abstract class or interface
'Bar'
```

为什么不对抽象类使用A-name惯例呢？因为当引用一个具有方法和属性的类型时，我们并不关心它是接口、抽象类，还是具体类。看上去框架类库（FCL）在未多加考虑的情况下就采用了I-name惯例。

### 题外话：另一种解释

I-name惯例在COM中是有意义的。COM实际上不是面向对象的，不是吗？它过去和现在都是面向组件的。是否有理由将组件的接口与组件本身作为两种不同概念来区别对待？也许有理由，但问题是FCL中的接口是否就是组件接口，或者它们是否是多重继承中的难题（编译器的解决方案或折中处理。如果是这样，那么CLR中的接口就非常接近于抽象类了。抽象类没有A-name惯例。我们可以不使用I-name惯例，因为它毫无目的。当引用或使用一个具有方法和属性的类型时，我们不关心它是接口、抽象类，还是具体类。

这里也把这个接口作为自分流来创建模拟实现。当然，窗体模型需要知道其检索程序。综合所有这些因素，测试固件可以采用如下形式作为开始。

```
[TestFixture] public class PickCandyTests: PickCandyView, CandySourceRetriever
{
    void PickCandyView.SourceItem(string text, bool pickable)
    {
        SrcPanel += text + (pickable ? ">"
            : string.Empty) + " ";
    }

    IList CandySourceRetriever.Retrieve()
    {
        return sources;
    }
}
```

```

PickCandyMdl mdl;
ArrayList sources;
string SrcPanel;

[SetUp] protected void Setup()
{
    mdl = new PickCandyMdl();
    mdl.SetView(this);
    mdl.SetRetriever(this);
    sources = new ArrayList();
    sources.Add(new CandySource("a")); // think chocolate
    sources.Add(new CandySource("b")); // ... lollipop
    sources.Add(new CandySource("c"));
    SrcPanel = string.Empty;
}
...snip...

```

当然，测试仍会得到红条，因为我们尚未在PickCandyMdl上实现任何操作。现在就来完成这一步。

```

[Serializable] public class PickCandyMdl
{
    public void Fill()
    {
        _retrieved = _retriever.Retrieve();
    }

    public void Render()
    {
        foreach(CandySource s in _retrieved)
            _view.SourceItem(s.CandyType, true);
    }

    public void SetView(PickCandyView v)
    {
        _view = v;
    }

    public void SetRetriever(CandySourceRetriever r)
    {
        _retriever = r;
    }

    [NonSerialized] PickCandyView _view;
    CandySourceRetriever _retriever;
    IList _retrieved;
}

```

可以看到，大部分代码都很简单。在Fill()中，我们挑选可用的糖果源，在Render()中，对它们进行迭代，从而将要显示的信息传递给视图。注意，我们已经创建了模拟实现，因此在所有情况下糖果源将不再是“可选的”（真正的常量）。SetView()和SetRetriever()都很简单。

如果在ASP.NET应用程序中对会话状态进行进程外处理,那么需要\_view上的[NonSerialized]属性。

现在运行测试,得到绿条。太好了!

## 2. 挑选糖果

我们需要能够从糖果源列表中挑选糖果,这可以用以下测试来表示。

```
[Test] public void PickingCandy ()
{
    mdl.Fill();
    mdl.PickAt(0);
    mdl.Render();
    A.AreEqual(SrcPanel, "a> b> c> ");
    A.AreEqual(StashPanel, "a ");
}
```

StashPanel是右侧窗格的测试固件表示,右侧窗格显示了(孩子们)已经挑选完的糖果(在实际窗体中是一个HTML表)。我们需要一个新方法PickAt(),在视图上也需要有一个新的方法。

```
public interface PickCandyView
{
    void SourceItem(string text, bool pickable);
    void StashItem(string text);
}
```

PickAt()是在窗体模型中引入CandyStash的很好理由,以下代码中的\_stash暗示了这一点。

```
public void PickAt(int index)
{
    Candy c = ((CandySource)_retrieved[index]).GrabOne();
    _stash.Add(c);
}
```

还需要补充Render()。

```
public void Render()
{
    foreach(CandySource s in _retrieved)
        _view.SourceItem(s.CandyType, true);
    foreach(Candy c in _stash)
        _view.StashItem(c.CandyType);
}
```

读者可以按照我实现SourceItem()的相同方式来实现StashItem()方法。

现在运行测试将再次得到绿条。在继续讨论之前,对第一个测试做一下补充,即断言StashPanel为空(string.Empty)。

## 3. 处理动态显示

当挑选了两个相同类型的糖果时,该类型就不再是“可挑选的”(参见应用程序的第一个图)。这可以用以下测试来表示。



```
[Test] public void PickingTwoOfSameKind()
{
    mdl.Fill();
    mdl.PickAt(0); mdl.PickAt(0);
    mdl.Render();
    A.AreEqual(SrcPanel, "a b> c> ");
    A.AreEqual(StashPanel, "a a ");
}
```

已选糖果中现在包含两个“a”。SrcPanel中的“a”后面没有大于号(>)，这一点是关键。这个测试未引入任何新方法，但它得到红条，原因是前面简化了Render()的实现。还记得真正的常量吗？CandyStash上的OkToAdd()方法不正是我们正在寻找的方法吗？

```
public void Render()
{
    foreach(CandySource s in _retrieved)
        _view.SourceItem(s.CandyType, _stash.OkToAdd(s.GrabOne()));
    foreach(Candy c in _stash)
        _view.StashItem(c.CandyType);
}
```

我从糖果源中选择一个(s.GrabOne())，并将它传递给OkToAdd()，以便看一下它是否是“可挑选的”。CandySource是一个无限源（它实际上是一个对象工厂）。在现实生活中我们不会这样浪费巧克力棒，但这不是现实生活，而是软件。

如果已经挑选了三个允许的混合，那么就不允许再挑选了。这可以用以下测试来表示。

```
[Test] public void PickingThreeMix()
{
    mdl.Fill();
    mdl.PickAt(0); mdl.PickAt(0); mdl.PickAt(1);
    mdl.Render();
    A.AreEqual(SrcPanel, "a b c ");
    A.AreEqual(StashPanel, "a a b ");
}
```

这个测试得到绿条。OkToAdd()的“可挑选性”检查是很完美的，读者是否也这样认为？这点不足为奇，因为OkToAdd()抓住了业务规则的本质。

### 11.3.5 Web 窗体实现

有几种方式可用来处理Web应用程序中状态。在这个场景中，已经在文档窗格上禁用视图状态，并只依靠服务器上的Session对象。在开始访问Web窗体时，构建一个页面，并且在结束处理之前重新构建它（在PreRender事件中），以便反映模型的当前状态。记住，当挑选了糖果之后，模型的状态将发生改变。

为此，在Web窗体的标准OnInit()方法中添加一个MyInit()调用（Web窗体是用Visual Studio .NET创建的）。

```
void MyInit()
```

```

{
    if (!IsPostBack)
    {
        _mdl = new PickCandyMdl();
        Session["PickCandyMdl"] = _mdl;
        _mdl.SetRetriever(CreateRetriever());
        _mdl.Fill();
    }
    else
    {
        _mdl = (PickCandyMdl)Session["PickCandyMdl"];
    }
    _mdl.SetView(this);
    MdlRender();
}

PickCandyMdl _mdl;
HtmlTable _srcTbl, _stashTbl;

void MdlRender()
{
    _srcTbl = new HtmlTable();
    srcpnl.Controls.Add(_srcTbl);
    _srcTbl.Width = "100%";

    _stashTbl = new HtmlTable();
    stashpnl.Controls.Add(_stashTbl);
    _stashTbl.Width = "100%";

    _mdl.Render();
}

private void WebForm1_PreRender(object sender, System.EventArgs e)
{
    srcpnl.Controls.Clear();
    stashpnl.Controls.Clear();
    MdlRender();
}

```

Srcpnl (糖果源窗格) 和stashpnl (已选糖果窗格) 都是通过Visual Studio中的窗体设计器添加的Panel (WebControls名称空间中的控件)。

当然, 窗体实现了PickCandyView。代码非常简单。所做出的任何决定 (if语句) 都只关注交互的外观, 而不关注交互的状态。这是很关键的。交互的状态必须由模型来负责。

```

void PickCandyView.SourceItem(string text, bool pickable)
{
    int index = _srcTbl.Rows.Count;
    HtmlTableRow tr = AddTR(_srcTbl);
    HtmlTableCell td = AddTD(tr);
    td.InnerText = text;

    td = AddTD(tr);
}

```

```

    if (pickable)
    {
        LinkButton lb = new LinkButton();
        td.Controls.Add(lb);
        lb.ID = "pick@" + index.ToString();
        lb.Text = ">>";
        lb.Click += new EventHandler(Pick_Click);
    }
}

void PickCandyView.StashItem(string text)
{
    AddTD(AddTR(_stashTbl)).InnerText = text;
}

```

最后，这里就是挑选回调方法。

```

private void Pick_Click(object sender, EventArgs e)
{
    _mdl.PickAt(IndexFromID(sender));
}

```

我确信读者可以理解以上代码。唯一不好理解的就是CreateRetriever()方法。我们已经在测试固件中看到了类似的方法。

图11-5用连环画形式显示了应用程序。孩子挑选了两个巧克力棒和一袋花生。

Chocolate >>	Chocolate	Chocolate	Chocolate	Chocolate	Chocolate
Lollipop >>		Lollipop >>	Chocolate	Lollipop	Chocolate
Peanuts >>		Peanuts >>		Peanuts	Peanuts

图11-5 糖果挑选应用程序

### 11.3.6 小结

处理GUI的大部分代码都位于交互模型中。只有一小部分不重要的代码未经过测试。然而，这种方法测试的并不是图形外观，而是GUI的交互。

在这个远未结束的示例中，只在窗体模型中做了很少的处理。如果想要添加用于取消选择的特性（返回糖果源），那么可以开发一个包装功能（例如CandyItem对象），用于包装所选中的糖果。

### 11.3.7 用 NMock 创建模拟

作为一个小的附录和趣味话题，下面显示另一种在使用NMock[NMock]进行测试期间创建模拟的方式。这个介绍将会很简短，主要列出测试代码，以便和前面的测试进行比较。首先看一下测试的设置。测试固件类不实现PickCandyView和CandySourceRetriever接口，因为在前面的自分流测试固件中已经实现它们。

```

PickCandyMdl mdl;
IMock data, view;

```

```

const bool Pickable=true, NotPickable=false;
[SetUp] protected void Setup()
{
    data = new DynamicMock(typeof(CandySourceRetriever));
    view = new DynamicMock(typeof(PickCandyView));

    mdl = new PickCandyMdl();
    mdl.SetView((PickCandyView)view.MockInstance);
    mdl.SetRetriever((CandySourceRetriever)data.MockInstance);
    ArrayList sources = new ArrayList();
    sources.Add(new CandySource("a")); // think chocolate
    sources.Add(new CandySource("b")); // ... lollipop
    sources.Add(new CandySource("c"));

    data.SetupResult("Retrieve", sources);
}

```

DynamicMock的实例化是关键，它创建了一些使用所指定的接口的对象。IMock.MockInstance是动态“编码和编译”的对象的模拟实现。当使用Retrieve()来调用数据模拟时，可以通过SetupResult()来通知它返回源。

现在看一下与上一个测试等价的测试。

```

[Test] public void FillSourcePanel()
{
    // Expectations
    view.Expect("SourceItem", "a", Pickable);
    view.Expect("SourceItem", "b", Pickable);
    view.Expect("SourceItem", "c", Pickable);
    view.ExpectNoCall("StashItem", typeof(string));
    // Execution
    mdl.Fill();
    mdl.Render();
    // Verification
    view.Verify();
}

```

代码中的注释只是为了显示在使用NMock进行测试时的几个常见阶段（除设置之外）。我们表明希望PickCandyView的SourceItem()被调用三次，并给出了期望值。还表明了StashItem()不被调用。

在执行阶段，通过调用要测试的对象上的一些方法来执行它。最后，请求视图模拟来验证在执行期间是否满足期望。

我认为最好遵循TDD的红-绿（-重构）公式。至少在前面的测试中，如果忘记调用Verify()，那么无论是否满足期望值，都将得到绿条。只有在首先得到红条的情况下，才能确信验证的有效性（断言）。

感谢Ingo!

再次强调，测试是一个非常重要的因素，而且值得花力气去准备和调整。现在，我们已经讨论了两种不同的UI变体，但它们的实现方式是类似的。

最后（但并非不重要）将讨论UI数据的一些方面（与领域模型中的数据相对）。

一些读者可能会认为在表示服务的映射方面我的方法有些小题大做了，因为映射通常是用于持久化服务的解决方案。但我认为DDD的主旨是尽可能将注意力集中在核心（即领域模型）上，然后创建简单灵活的解决方案，用来处理领域模型“周围的”基础架构问题，例如将领域模型转换为用于另一个层的另一种模型。这正是对象-对象映射和UI映射具有巨大潜力的原因所在，特别是对于那些只是浪费时间和手工代码的例行任务。

在下一节中，Mats Helander将讨论UI映射的一些方面，重点关注具有友好表示的模型（例如表示模型[Fowler PoEAA2]）与领域模型之间的自动映射。

## 11.4 映射和包装

作者：Mats Helander

借助于O/R映射，我们能够将领域模型连接到关系数据库。再加上领域模型类和服务层[Fowler PoEAA]添加的业务方法，就有了一个相当稳固的架构，这为应用程序奠定了结构和功能基础。

这为我们引出了最后一个分支——表示层（PL）。虽然开发人员可能都认为领域模型是应用程序的心脏，但用户可能只是将他们所看到的表象作为最终的判断依据。

这意味着无论领域模型（以及为其提供支持的服务层）有多么好，如果没有通过一些精彩的UI来提供对模型的访问，那么不会有人为我们的工作喝彩（除了开发伙伴以外）。

好消息是如果幸运的话，将PL连接到领域模型只是小事一桩。坏消息是如果不走运的话，这可能就是一项很艰难的任务。

在最简单的情形中，领域模型已经是“可表示的”，只需直接将数据绑定到领域模型对象即可。然而，应用程序的接口通常要求以一种比领域模型中的形式更加“用户友好的”方式来呈现这些信息。

### 11.4.1 映射和包装

当领域模型对象无法立即显示时，必须做出一个选择：是应该只包装DM对象，还是应该映射一个新的对象集合？

例如，假设有一个名为Person的DM类，它带有FirstName和LastName属性。然而，在UI中需要有一个文本框用于编辑一个人的姓名。

当然，一种处理方式是在UI中编写一些代码，这些代码简单地读取领域模型Employee对象的名字和姓并将它们写到文本框中，并且在用户点击保存时，读取文本框中的内容，并将其写到Employee对象的属性中。

大多数时候，在所谓的“真实世界”中，这是解决UI需求的方式，这本质上是一种堆积操作的方式。

最后，用这种方式得到的应用程序不具有可维护性，这就激发我们寻找一种能够随着应用程序复杂性和规模的增加而扩展的设计。

一种常规的解决方案是用一个新的对象集合来补充领域模型对象，这个集合通常被称为表示模型（PM）。

表示模型对象是PL的一部分，而且应该具有符合PL需求的结构和行为（实际是符合UI的需要）。在我们的示例中，PM Person类有一个FullName属性，而没有领域模型Person类中的FirstName和LastName属性。

为什么不将GetFullName()方法放到领域模型Person类中呢？如果在某种业务逻辑操作中使用它的话，那么可以这样做，但如果GetFullName()方法只在PL中使用，就应该将此方法放到PM中，而不是放到领域模型中。

保持领域模型远离非业务方面（例如表示和持久化）与保持业务逻辑层远离表示逻辑和数据存取逻辑同等重要，原因是相同的：保持领域模型远离任何非业务方面是保证应用程序“核心”的可理解性和可维护性的关键。

因此，对于本示例来说，假设GetFullName()只在表示中使用，因此应该将它放在PM中，而不是放在领域模型中。

问题变成了如何将PM类连接到领域模型类，因此我们面临以下选择：映射还是包装？

#### 11.4.2 用表示模型来包装领域模型

包装通常是一种更简单的解决方案，它不需要在PM中有额外的状态管理框架。这里的思想是将DM Employee对象传递给PM Employee对象的构造方法。然后，PM Employee保持对DM对象的一个内部引用，并委托所有对DM属性的调用。参见以下代码清单。

```
//Presentation Model wrapper object
namespace MyCompany.MyApplication.Presentation
{
    public class EmployeeWrapper
    {
        private Employee employee;

        public EmployeeWrapper (Employee employee)
        {
            this.employee = employee;
        }

        public int Id
        {
            get { return this.employee.Id; }
            set { this.employee.Id == value; }
        }

        public string FullName
        {
            get
            {
                return this.employee.FirstName + " " +
                    this.employee.LastName;
            }
        }
    }
}
```



```

set
{
    //This should of course be complemented with
    //some more cunning logic, as well as some
    //verification, but this is just an example...
    string[] names = value.Split(" ".ToCharArray())
        , 2);
    this.employee.FirstName = names[0];
    this.employee.LastName = names[1];
}
}
}
}

```

使用EmployeeWrapper类的目的是从领域模型提取Employee对象，然后将其传递给EmployeeWrapper构造方法（参见以下代码清单）。

```

//Using the Presentation Model wrapper object
Employee employee = employeeRepository.GetEmployeeById(42);
EmployeeWrapper employeeWrapper = new EmployeeWrapper(employee);

SomeControl.DataSource = employeeWrapper;

```

包装方法的主要优点是简单且灵活。我们可以在包装器对象中编写任何转换，如果需要的话，这些转换可以比相对简单的FullName转换复杂得多。

一个主要缺点是包装器类中将包含大量重复代码，它们用于将调用委托给那些不需要任何转换的属性，例如前面示例中的Id属性。

这可能导致包装器类跳过转换操作，并使得表示模型对象从领域模型对象进行继承，从而重写所有需要转换的属性，而保持其他属性不变。这使得我们不得不接受领域模型的公共API，因为子类将继承超类的所有公共成员。

在我们的示例中，可以令PM Employee从领域模型Employee继承，并且可以为PM Employee添加FullName属性，但PM Employee也将公开领域模型Employee的FirstName和LastName属性，因为这些公共属性是继承的。

包装领域模型对象（而不是从它们继承）提供了更高级别的封装，这通常值得在PM和领域模型属性之间的委托上花费一些额外工作（无可否认这些是冗长乏味的工作）。

此外，对于不需要高级转换的属性，它们的委托代码可以通过代码生成解决方案来处理，甚至可以仅通过代码片段模板来生成。如果这两种方法都无效，那么可以让实习学生来编写这样板式的代码！

### 11.4.3 将表示模型映射到领域模型

包装方法的替代方案是在DM对象与PM对象之间来回复制数据。在这种情况下，需要编写类似于下面这样的代码。

```

//Moving data manually between Presentation Model
//and Domain Model

```

```
Employee employee = employeeRepository.GetEmployeeById(42);
EmployeeView employeeView = new EmployeeView();

employeeView.Id = employee.Id;
employeeView.Salary = employee.Salary;
employeeView.FullName = employee.FirstName + " " +
employee.LastName;
```

像包装示例一样，用于复制数据的代码可以放在PM对象的构造方法中，在这种情况下，构造方法将以参数形式接收DM对象。清晰起见，前面的示例只显示了操作代码行。

如果你受到和我一样的困扰（即抽象问题），那么立即会意识到将这些工作抽象到某种框架中的潜力。简言之，我们应该能够指定（例如在XML文件中指定）将哪些PM属性映射到哪些DM属性，然后令框架使用反射来移动数据。

这样的框架必须将一个对象集映射到另一个对象集，因此这种类型的框架的逻辑术语就应该是“对象/对象映射工具”或简称O/O映射工具！

如果决定编写这样一个框架，那么可能会意识到解决非转换问题将是一个非常直接的任务，例如前面示例中的Id到Id的映射以及Salary到Salary的映射。另一方面，将FullName映射到First-Name和Last-Name将较为复杂。

这个问题很容易解决，但问题就变成了将高级转换服务构建到O/O框架中是否是正确的方法。

另一种方法是不让O/O映射工具负责实际结构转换，而是令其只负责在具有相同或类似结构的对象之间来回移动数据。

在我们的示例中，解决方案变成这样：除了其自己的FullName属性之外，还为PM Employee类提供一个FirstName和LastName属性，但也令FirstName和LastName属性是受保护的。

O/O映射工具应该可以使用反射来访问受保护的PM属性，因此它能够从领域模型的FirstName和LastName属性映射到受保护的PM属性，但PM对象只会把FullName属性公开给客户。

事实上，O/O映射工具甚至可以直接将数据写到PM对象的私有字段中，因此完全不必实现受保护的属性，映射工具可能应该直接访问私有字段，以避免触发属性的getter和setter方法的副作用（仅当客户代码访问属性时，才应调用getter和setter方法，而框架移动数据时不应调用它们）。

然而，一些O/O映射工具可能提供了高级特性（例如延迟加载），这些特性依赖于某些属性，以便可以拦截对这些高级的特性的调用，由于此原因，需要保留这些属性。以下代码清单显示了一个依赖O/O映射的PM对象。

```
//Presentation Model object that relies on O/O Mapping
namespace MyCompany.MyProject.Presentation
{
    public class EmployeeView
    {
        private int id;
        private decimal salary;
        private string firstName;
```



```

private string lastName;

public EmployeeView() {}

public int Id
{
    get { return this.id; }
    set { this.id = value; }
}

public decimal Salary
{
    get { return this.salary; }
    set { this.salary = value; }
}

protected string FirstName
{
    get { return this.firstName; }
    set { this.firstName = value; }
}

protected string LastName
{
    get { return this.lastName; }
    set { this.lastName = value; }
}

public string FullName
{
    get
    {
        return this.firstName + " " +
            this.lastName;
    }
    set
    {
        string[] names = value.Split(" ".ToCharArray(), 2);

        this.firstName = names[0];
        this.lastName = names[1];
    }
}
}
}

```

当然，无论在框架中执行什么操作，都可以通过手工方式来完成。如果要在不使用O/O映射框架的情况下将数据从领域模型手工复制到PM，以便将数据写入受保护的属性，那么既可以使用反射，也可以创建快捷方法，通过名称来访问属性。使用第二种方法的代码如下。

```

//Move data between Presentation and Domain Model without
//transformation

```

```

Employee employee = employeeRepository.GetEmployeeById(42);
EmployeeView employeeView = new EmployeeView();

employeeView.Id = employee.Id;
employeeView.Salary = employee.Salary;

//We have to use a method that lets us access protected
//properties by name to write to the protected methods.
//Alternatively, we could use reflection.
employeeView.SetPropertyValue("FirstName", employee.FirstName);
employeeView.SetPropertyValue("LastName", employee.LastName);

```

使用这种方法，在PM和领域模型之间移动数据的任务就变得非常直接了，如果不考虑那些麻烦的引用属性的话，这些任务甚至变得很容易。

#### 11.4.4 管理关系

当考虑到对象之间的关系时，任务再次变得困难了。对象图（通过关系互连到一起的一组对象）可能很大，如果令O/O映射工具从DM Employee对象填充PM Employee对象会导致填充几百个相关对象，那么应用程序的性能将受到极大影响。

另一个问题是必须记住从PM引入属性返回PM对象。例如，当读取PM Employee对象的AssignedToProject属性时，我们希望返回一个PM Project对象，而不是DM Project对象。下面显示了具有一个引用属性的包装器对象的初步实现。

```

//Naive implementation of reference property in wrapper object
namespace MyCompany.MyApplication.Presentation
{
    public class EmployeeWrapper
    {
        private Employee employee;

        public EmployeeWrapper (Employee employee)
        {
            this.employee = employee;
        }

        public Project AssignedToProject
        {
            get { return this.employee.Project; }
            set { this.employee.Project = value; }
        }
    }
}

```

这里的问题在于，PM EmployeeWrapper对象的Project属性将返回一个DM Project对象，而不是PM ProjectWrapper对象。这通常并不是我们想要的结果，因此必须仔细地编写PM引用属性，可以采用以下方式：

```

//Slightly less naive implementation of reference property
//in wrapper object

```

```
namespace MyCompany.MyApplication.Presentation
{
    public class EmployeeWrapper
    {
        private Employee employee;

        public EmployeeWrapper (Employee employee)
        {
            this.employee = employee;
        }

        public ProjectWrapper AssignedToProject
        {
            get
            {
                return new
                    ProjectWrapper(this.employee.Project);
            }
            set
            {
                this.employee.Project =
                    value.GetDomainObject();
            }
        }

        public Employee GetDomainObject()
        {
            return employee;
        }
    }
}
```

注意在AssignedToProject() setter方法中使用的ProjectWrapper类上的GetDomainObject()方法。当向PM EmployeeWrapper对象的AssignedToProject属性写入数据时，将一个ProjectWrapper对象传递给setter方法，但必须将一个领域模型Project对象传递给被包装的Employee对象的Project属性。因此，我们需要从被传入的ProjectWrapper对象访问所需的Project对象，GetDomainObject()方法可以完成此任务。

因此，当实现引用属性时，必须提供一个用于访问领域模型对象的方法（PM对象在内部引入此领域模型对象），否则的话就不需要此方法。

为了讨论的完整性，也为了能够在ProjectWrapper类中实现相应的AssignedEmployees属性，我们还在EmployeeWrapper类上提供了一个GetDomainObject()方法。

但实际上这还不够。如果用一种批判的眼光来看代码，我们会注意到每次从AssignedToProject属性读取时，都会创建一个新的ProjectWrapper对象。假设正在使用领域模型层中的标识映射，那么当反复读取相同属性时所创建的新ProjectWrapper对象都将包装相同的DM Project对象，因此不会带来数据不一致和数据破坏的风险，但不管怎样这也不是理想的解决方案。

最后，我们可能需要一种可以同时处理PM中的标识映射问题的成熟解决方案，以确保不会

出现在一个会话中有两个不同的PM实例表示同一个领域模型实例的情况。

这样,就必须重写AssignedToProject属性中的代码,以便不再创建ProjectWrapper对象本身的新实例,而是通过ProjectWrapper对象的标识映射向PM存储库请求一个实例。

这意味着EmployeeWrapper对象将需要一个对ProjectRepository对象的引用。这就要向迄今为止表示模型所带给我们的简单性说再见了。最终结果是,在领域层中管理引用属性的所有“精彩纷呈的”恐惧和困难再次向我们袭来。

通过引用属性“解决”这些问题的一种方式是不在PM对象中使用引用属性,使用“变平”对象只公开基本属性,这些对象很可能也来自引用对象。参见以下代码清单。

```
//"Flattened" Presentation Model object, exposing only primitive
//properties
namespace MyCompany.MyApplication.Presentation
{
    public class EmployeeWrapper
    {
        private Employee employee;

        public EmployeeWrapper(Employee employee)
        {
            this.employee = employee;
        }

        public int Id
        {
            get { return this.employee.Id; }
            set { this.employee.Id = value; }
        }

        public string AssignedToProjectName
        {
            get { return this.employee.Project.Name; }
            set { this.employee.Project.Name = value; }
        }
    }
}
```

通常这种方法很有用,特别是因为很多UI控件都设计成显示表的行,而变平的对象很适合这种范例。但当我们想要表示可导航的深层次结构(而不是表格形式的内容)时,变平对象的方法就显得不足了。然而,我们经常会用一些专门为应用程序中的某些表格编写的平面对象来补充完全分区的PM。

作为一个很好的示例,“变平的”PM对象很好地体现了PM与领域模型之间的区别,而且区别越大,我们拥有两种独立模型(而不是将PL所需的特性添加到领域模型中)的动机就越大。

### 11.4.5 状态问题

当在映射和包装之间进行选择时,应特别注意这两种方法中对状态的不同处理方式。

当包装领域模型对象时,应用程序只使用数据的一个实例。如果在同一时间有两个视图显示

了同一位雇员，而且在一个视图中更新了该雇员的名字，那么在被包装的对象中只会更新数据的一个实例。如果刷新第二个视图，它应该显示更新后的数据，因为它也是直接映射到同一个被包装的、更新后的对象。

然而，将PM对象映射到领域模型对象时，可能就会有几个不同的PM对象集合表示同一个领域模型对象，但它们具有不同的状态。

即使PM有一个标识映射，也可能有两个不同的PM Employee类，它们的外观略有不同（用于不同的表示目的），在这种情况下，标识映射就起不到任何帮助作用了。

这可能导致冲突。另一方面，它也可能会导致复杂的、脱节的数据管理。

例如，在某个向导中，一个映射的PM对象集可能会被单独使用很长一段时间。在向导的最后一步，用户可以决定向领域模型提交更改（然后向数据库提交更新后的领域模型），或者放弃更改。如果用户一直在使用一个PM对象集（它直接包装了领域模型对象），那么取消更改的选项是不可用的，因为当用户在向导中工作时，领域模型对象已经被不断更新！

当然，假设领域模型支持在断开连接的情况下进行操作（例如，使用操作单元[Fowler PoEAA]），那么即使没有PM，取消选项也是存在的，以便不将更改转发给数据库。但是，如果在直接使用领域模型的向导的最后按下取消按钮，即使数据库不发生更改，领域模型仍然已经被更改了，这是为了在应用程序的下一个屏幕显示它时使更改是可见的。

如果要在用户取消时准备丢弃领域模型，那么就没有问题。如果操作单元支持领域模型状态的完全回滚，那么也不会产生问题。但如果正在编写一个富客户，想在应用程序中全程使用领域模型（例如，当用户取消时不想丢弃DM），而且操作单元又不支持完全回滚，那么就会遇到这个问题。

在某些情况下，我们将发现包装是最符合需要的，而在其他情况下用映射更好一些。有时在同一个应用程序中会同时使用这二者，或许使用包装作为默认方法，而为向导或其他“批处理作业”使用映射，以便在不提交时丢弃它们。

### 11.4.6 最后的想法

有时我们会很幸运。当可以直接向用户呈现领域模型时（或者只需要很少的支持工作），应该感谢幸运之星，因为这并未改变这样一个事实：将PM连接到领域模型是一个令人非常头痛的问题。

如果决定使用“变平”PM对象，问题就小多了，因为大部分更严重的问题都将作为管理引用属性的结果直接出现。

包装和映射之间的选择受到很多因素的影响，包括状态是如何管理的，以及有多大的机会可以通过代码生成（包装）或使用O/O映射框架（映射）来减少应用程序的样板式编码工作。

通过尝试将结构化转换保持在任何框架或代码生成的范围之外（而不是将所有这样的高级转换逻辑都置于PM类的代码中，并把它们设置成受保护的，从而隐藏PM中的非转换成员），可以极大地降低这样的框架和代码生成工具所需的复杂度，从而更有机会自己编写一些实用代码，或从网上查找一些有用的东西。

如果你置疑所有这些额外的架构开销（包括PM、O/O映射工具和转换逻辑方法等形式）是否真正需要，那么或许你已经是足够幸运的，因为已经有了一个可表示的领域模型。

然而，底线是应该真地要尽最大努力避免为了符合PL的需要而去修改领域模型，不要让领域模型同时充当PM的角色，除非可以在不修改领域模型的情况做到这一点。

一旦认识到领域模型在应用程序核心中的表示方式与呈现给用户的方式不同，那么就需要使用PM了，至少这样做符合以下原则：永远不要让领域模型负担PL方面。

我自己就非常极端地保持领域模型完全脱离PL方面。这意味着如果我的领域模型直接与数据绑定，并且只添加了某些属性的话（例如用于指定目录和绑定到属性网格的默认值的属性），那么我不会将这些PL属性添加到领域模型对象中。相反，我将用一个PM来补充领域模型，即使除了属性之外，这个PM在每个细节上都与领域模型相同。当然，那些不像我这样极端的人可能不采取这种做法。

但我这样做的一个原因在于，有了PM以及用于包装或映射到领域模型的支持基础架构之后，我总是发现有大量的机会可以精化PM的表示方面，而且很快两个模型会变得越来越不同，这就使我有更大的动机用PM来补充领域模型。

感谢Mats！

## 11.5 小结

Mats通过对表示服务的讨论介绍了如何再次从领域模型获益。

虽然我们将重点集中在领域模型的核心上，但这并不是说表示服务就不重要。事实恰好相反！表示服务的重要性是我们努力开发领域模型的原因之一。可能将上下文中那些分心的事物移到PL的外部，更容易也更能创建出一流的用户界面。

## 11.6 结束语

现在我们已经完整介绍了用DDD方法开发应用程序的思考过程。本书采用了高度结构化的论述，体现在以下方面。

- (1) 首先尝试收集需求并理解问题。
- (2) 接下来考虑使用哪种方法来建立主逻辑。根据Fowler的观点，我们可以从事务脚本、表模块和领域模型中进行选择[Fowler PoEAA]。如果通过需求看出这是一个复杂（在行为和/或关系方面）且长期的项目，那么可能会选择领域模型。
- (3) 然后需要一种特定的领域模型风格，不仅为了避免常见的陷阱，而且为了创建一些真正有用的功能。这就需要用到DDD了。我们尝试设置通用语言[Evans DDD]。我们在领域模型上努力工作，尝试使它包含丰富的知识。
- (4) 然后需要思考一些分散注意力的方面：所需的基础架构。最典型的问题是如何处理持久化。如果可能的话，对象关系映射是DDD项目的常用选择。（这里不使用“基础架构”这个过于泛滥的术语，而是把它描述为从模型到其他层的映射，例如表示层、持久化层和集成服务层。）
- (5) 最后，还有其他一些事情需要考虑，例如如何处理表示。

我并不认为这是一个瀑布过程，而是一本书的自然顺序，因此有时这也是叙述的顺序。要记住和强调的重要一点是迭代和增量风格的开发，这是缓解风险和项目成功所必需的。

好，本书到此就结束了。但正如Dan所说，它只是个开始。

还有一件事：本书以讨论如何评价“lagom”（不多也不少，恰好是均衡的）作为开始。那么本书是否做到了“lagom”？我认为对一些人来说，本书在某些方向上有些极端了，在其他方向上则可能是“lagom”。

“什么是均衡的”完全在于观察者的眼睛和上下文。







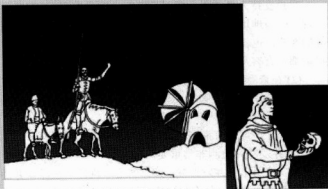
# Part 5

## 第五部分

## 附 录

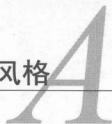
以下两个附录提供了领域模型风格的更多示例，以及一个模式汇总目录。

附录部分介绍了其他的风格和模式汇总。



### 本部分内容

- 附录 A 其他领域模型风格
- 附录 B 已讨论的模式目录



**领域**模型模式有很多种不同的使用方式。我请几位朋友介绍他们最喜欢的领域模型应用方式。第4章中的需求列表作为他们的输入，读者将在附录A中发现它们的答案。<sup>①</sup>下面再次给出这些需求。

(1) 应用灵活但复杂的过滤器来列出客户。

客户支持人员需要非常灵活的客户搜索方式。他们需要在很多字段上使用通配符，例如名称、位置、街道地址、介绍人，等等。还需要查询具有特定种类订单、特定规模订单、特定产品订单等的客户。也就是说，他们需要一种功能完备的搜索实用工具。搜索结果是一个客户列表，每条记录都带有客户编号、客户名称和位置。

(2) 在查看特定客户时列出订单。

每个订单的总价值、状态、类型、日期和介绍人姓名在列表中应该是可见的。

(3) 一份订单可具有多个不同的行。

一份订单可以有多个订单行，其中每个行描述了产品及其订单数量。

(4) 并发冲突检测非常重要。

可以使用乐观并发控制。即，当用户执行一些操作并尝试保存之后，可以通知他们将会与上一次保存产生冲突。只有导致实际不一致的冲突才应该被认为是冲突。因此，解决方案需要决定为客户和订单使用版本控制单元。（这将对其他特性产生小的影响。）

(5) 客户对我们的应付款项不能超过一定的额度。

此限制是特定于每个客户的。当最初添加客户时，就要定义这项限制，而且过后可以修改额度。如果具有超过总限额的订单存在，可以认为发生了不一致，但有一种情况允许存在这种不一致性，即用户降低了限额。降低了限额的用户会收到通知，但保存操作仍然是允许的。但是，添加订单或修改订单则被禁止，以便不超过限额。

(6) 订单的总价不能超过100万SEK。

与前一个限制不同，此限制是一条系统级规则。（SEK是瑞典货币，但这无关紧要。）

(7) 每份订单和每位客户都应有一个唯一、用户友好的编号。

序号之间允许有间隔。

① 注意，当他们接受输入时，想法是在其场景中要求一个应用服务器，这是后来针对第4章做出的修改。

(8) 在一位新客户被视为可接受之前，必须联系信用卡机构检查客户的信用。

也就是说，对前面为客户定义的限制进行检查，以确定其是否合理。

(9) 一份订单必须有一位客户，一个订单行必须对应于一份订单。

未定义客户的订单是不能存在的。同样的规则也适用于订单行，它们必须属于某个订单。

(10) 保存订单后，其订单行应该是原子的。

坦白地讲，我实际上并不确定是否需要此特性。首先创建订单然后再添加订单行是可以的，但我们需要这样一条规则，以便有一个与事务保护相关的特性需求。

(11) 订单具有用户可修改的接受状态。

用户可以将此状态修改为不同的值（例如批准/拒绝）。也可以有其他状态值，但这些值是在领域模型中由其他方法隐式修改的。

首先请Mats Helander来介绍一下“面向对象数据模型、智能服务层和文档”。

## A.1 面向对象数据模型、智能服务层和文档

作者：Mats Helander

有时，在黑暗的冬夜，我曾模糊地回忆起在领域模型出现之前的生活。记得在那些日子里，我总是往热巧克力上倒点白兰地，然后再往壁炉里扔根柴。

我的应用程序架构演变最后导致了领域模型的使用，它遵循一种读者熟悉的模式。下面快速介绍一下这个演变过程，因为它有助于为我当前使用领域模型的方法提供一个上下文。

### A.1.1 起初

在我开始编写客户-服务器应用程序时，总是将所有应用程序代码放到客户中，客户直接与数据库进行交互。当开始编写Web应用程序时，我使用ASP页面，所有应用程序代码都放在这些页面中，包括用于调用数据库的代码。

当应用程序的复杂性稍微增加时，这种类型的应用程序架构的结果无一例外地陷入一片混乱，因此我很快学会将大部分代码从表示层中提取出来，并将它们放到业务逻辑层（BLL）中。

很快，应用程序就变得更易于开发和维护，而且一个额外的优点是BLL可以在不同的表示层之间重用。还应注意到的是，在这段时期内，在组件中编译大量代码使得Web应用程序的性能和可伸缩性有很大提高。

下一步是从BLL中提取出所有负责与数据库通信的代码，并将这些代码放到它们自己的层中，即数据访问层（DAL）。我记得当看到DAL通常比BLL更大时，我感到有些惊讶，这意味着处理数据库通信的应用程序逻辑比实际的业务逻辑更多。

这个时候就是领域模型即将到来的时候。PL将与BLL对话，BLL又与DAL对话，而BLL则与数据库对话。然而，数据库与应用程序之间的数据传递采用记录集的形式，即数据库行的内存中表示（参见图A-1）。

有时，使用关系形式的数据正好是我们需要的，在这些情况下，前面描述的架构仍然很适用。但我越来越认识到业务逻辑并不总是适合使用关系数据结构。

表示层 (可以有多个)

业务逻辑层数据访问层

关系数据库 (可以有多个)

图A-1 领域模型之前的经典的分层应用程序架构

### A.1.2 面向对象和关系数据结构

通过使用Computer Associates的一个名为Jasmine的面向对象数据库, 我开始理解了这样一个事实: 当开发业务逻辑时, 使用面向对象数据结构比使用相应的关系结构要简单得多。

举例来说, 实体之间的关系更简单。多对多关系尤其如此, 因为关系数据结构需要一个额外的表, 而面向对象数据结构中却不需要额外的类。

集合属性是另一个需要在关系模型中使用额外表的情况, 而在面向对象数据结构中也不需要额外的类。此外, 面向对象数据结构是类型安全的, 并且支持继承。

### A.1.3 领域模型和对象-关系映射

OO数据库并不常见。我所遇到的大多数项目仍然以关系数据库为中心。这就是为什么要使用对象-关系映射的原因。

对象-关系映射是指使用面向对象的数据结构, 并将其映射到关系数据库中的表。优点是能够像使用OO数据库(例如Jasmine)那样来使用面向对象数据结构, 同时仍使用关系数据库来存储所有数据。

此外, 也可以以业务方法的形式将业务逻辑添加到面向对象数据结构的类中。你可能已经猜到, 带有业务方法的面向对象数据结构就是领域模型。

通过学习对象-关系映射, 我最后采取的步骤是将领域模型包含在应用程序架构中。领域模型层被插入到BLL和DAL之间, 利用领域模型层, 业务逻辑现在能够使用面向对象数据结构了(参见图A-2)。

表示层 (可以有多个)

业务逻辑层领域模型层数据访问层

关系数据库 (可以有多个)

图A-2 更新后的分层应用程序架构, 包括领域模型在内

这意味着原来使用Recordset(记录集包含Employees表的一行)的BLL方法现在可以使用Employee领域模型类了。我们获得了类型安全性, 而且代码也变得更短, 更易于阅读, 此外在开发期间还可以使用微软的智能感知特性在数据结构中导航。

从那时起, 我就成为了领域模型的忠实用户, 而且不再回头——除了那些寒冷黑暗的冬夜。

如前所述, 领域模型的另一个优点是可以将业务逻辑分布到各个领域模型类中。在业务逻辑的放置位置上我曾经做过很多实验。我曾将大部分业务逻辑放到业务逻辑层中, 然后又将它们移到领域模型层中。

### A.1.4 服务层

我也曾将术语服务层[Fowler PoEAA]改为业务逻辑层,因为它更好地解释了当将大部分业务逻辑放到领域模型层时的操作。当大部分业务逻辑处于服务层中时,就没有理由再将术语改回业务逻辑层了,尽管我有时将其称为“厚的”服务层,以区别于更传统的薄服务层。

近来,我几乎总是将大部分业务逻辑放到“厚的”服务层中。虽然一些坚持纯OO的人指出应该将结构和行为结合起来,但我经常发现更实用的方法是将与业务逻辑有关的行为保持在服务层中。

这样做的主要原因是,负责控制领域模型行为的业务规则与那些控制领域模型结构的规则往往在变化频率和变化时间上不同。另一个原因则是使用不同业务规则集能够提高领域模型的可重用性。

通常,唯一可以放在领域模型类上的业务方法是满足以下两个条件的方法:一是这些方法是非常基本的方法,在任何业务规则集下都是有效的;二是它们的变化时间和变化频率与领域模型结构的变化时间和变化频率相同。

把大部分业务逻辑放到服务层之后,应用程序架构就对进入SOA世界做好了充分准备。但在进入SOA并讨论如何处理Jimmy的示例应用程序之前,首先强调一下此方法的另一个特殊的获益。

### A.1.5 组合

如前所述,领域模型所提供的面向对象数据结构并不一定都适合于我们要执行的每种操作。有很多原因使得关系数据库仍在广泛使用,并且很多人在实际的OO数据库到来之前都首选使用对象-关系映射。SQL所提供的关系数据结构和基于集合的操作完美适用于很多任务。

在这些情况下,对于那些实际上直接在关系数据结构上执行基于集合的操作的领域模型对象来说,强制将方法放到这些模型上看起来并不是自然的方式。

相比之下,令那些直接(或通过DAL)访问数据库的服务层方法与那些在领域模型上执行操作的方法共存并没有什么坏处。这样也更容易将服务器层方法从使用一个方法转换为使用另一个方法。

简言之,可以按自己的需要自由地实现每个服务层方法,在不需要的地方可以忽略领域模型,而在需要领域模型的地方则可以充分利用它(以我的经验来看通常是需要领域模型的)。

### A.1.6 Jimmy的应用程序和SOA

下面回到Jimmy的应用程序和全新的SOA世界上来。到目前为止,实际上我只介绍了我的应用程序架构在服务器上的形式,这种架构对于Web应用程序来说可能是足够好的,但我们现在要处理的是富客户,而且接下来的内容将会把应用服务器作为一个变体。

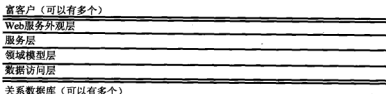
当很多(如果不是说全部的话)业务逻辑被分布到整个领域模型时,一种尝试性的方法就是用这种方式将数据和行为一起打包,并传递到富客户中,再为其提供对领域模型的访问。在这种架构中,大部分行为都位于服务层中,因此一种更自然的方法是分别公开行为和数据。

具体来说就是用一个Web服务外观层(它是远程外观层模式[Fowler PoEAA]的一种类型)来补充服务层,这个外观层只是一个薄的层,它用于在服务层中将业务逻辑方法公开为Web服务。富客户通过调用Web服务与服务器进行通信,并交换包含序列化的领域模型对象的XML文档。

对于我们的示例应用程序来说，这意味着每当富客户需要新信息时，它将调用服务器上的Web服务，然后Web服务返回XML文档形式的信息。下面举一个例子作为开始：列出匹配一个过滤器的客户。

首先创建一个GetCustomersByFilter()服务层方法，它负责实现实际的过滤功能。服务层方法带有过滤器参数，并返回匹配的领域模型对象。然后，添加一个GetCustomersByFilter() Web服务，它向富客户公开服务层方法。

Web服务接受相同的过滤器参数，并在一个调用中将它们传递给服务层方法。然后，服务层方法返回的领域模型对象将被序列化为XML文档，Web服务最终返回的就是此文档。然后，富客户就可以使用XML文档中的信息来为用户显示客户列表了（参见图A-3）。



图A-3 富客户应用服务器分层应用程序架构（包括领域模型在内）

富客户应用程序开发人员完全可以自行决定是否创建一个客户端领域模型，以便将来自XML文档的数据填充到其中。开发人员也可以决定存储信息，供离线使用。一种方式是只将XML文档保存到磁盘。另一种方式是将客户端领域模型存储到客户机器上的一个离线数据库中。

注意，最后的客户端领域模型不一定要与服务器上的领域模型匹配。相应地，如果将客户端领域模型存储到离线数据库中，那么客户端数据库可能就要遵循客户端领域模型的设计，因此看起来就不一定像服务器端数据库了。

**说明** 服务器端数据模式与客户端数据库模式之间的一个重要区别在于，如果服务器端的数据库表使用了自动增加的标识字段，那么对应的客户端数据库表通常不应使用这样的标识字段。假设在服务器上创建一位新雇员。当在Employees表中插入一个新行时，在其自动增加的ID列中自动分配一个值。然后对此雇员进行序列化，并发送给客户，客户端用这些数据填充一个领域模型，并将其存储到客户端数据库中，以供离线使用。客户Employees表中的新行不应被分配一个新的ID，而应该使用服务器端数据库为这个雇员分配的ID。这意味着客户和服务端上的ID列的属性可能是不同的。

这里的重点是，富客户应用程序的开发人员可以自由决定是否需要使用领域模型，如果需要使用的话，可以根据自己的需要自由设计它。客户与服务器之间的通信是完全面向文档的，任何一端均不能对另一端做出任何有关领域模型的假设。

例如，富客户可能会调用很多不同公司的Web服务，所有服务都能够返回客户列表，但它们的XML文档中都包含稍有不同的字段。处理此问题的一种方式创建一个“超集”（superset）客

户Customer领域模型对象，它具有从所有被调用公司返回的所有不同字段的属性。

接下来，我们希望富客户应用程序的用户能够查看属于某位特定客户的订单。当然，第一个Web服务返回的XML文档中包含每位客户的唯一ID，因此我们知道该客户的ID。

必须执行的操作是实现一个Web服务，它获取客户的ID，并返回该客户的订单。同样，首先创建一个GetOrdersByCustomerID()服务层方法，它接受客户的ID，并返回属于该客户的领域模型Order对象。

然后添加一个GetOrdersByCustomerID() Web服务，它获取客户ID，并将其传递给服务层方法，最后将返回的订单对象序列化到一个XML文档中，然后Web服务再返回这个文档。

然而，在这种情况下，需要对XML序列化加以额外的注意。我们希望XML文档包含每个订单的总价值，但Order领域模型对象不包含任何TotalValue属性，因为订单的总价值并未存储在数据库的任何字段中。

相反，总价值是在需要的时候计算出来的，计算方法是将各个订单行的值加到一起。每个订单行的值的计算方法是产品的购买价格乘以购买数量。

由于发送给客户端的XML文档中不包含订单行，因此客户应用程序无法计算订单的总价值，这意味着必须在服务器上计算，而且结果必须包含在XML文档中。

很多人都喜欢把计算订单行的值和订单总价值这种方法放到OrderLine和Order领域模型对象上，但我喜欢将它们放到服务层中。

### A.1.7 服务层设计

我通常围绕领域模型中使用的相同实体来组织服务层，而不是按任务来组织，但后者也是一种很流行的方法。这意味着如果在领域模型层中有一个Employee对象，那么在服务层中就会有一个EmployeeServices类。

对于那些不需要任何转换控制的服务，我通常将服务层方法实现为静态方法，这样就可以在不实例化任何服务层类的对象的情况下调用它们。

因此，在目前的示例中，我们将一个静态GetValue()方法放到OrderLineServices服务层类上，并将一个静态GetTotalValue()方法放到OrderServices类上。这些方法将接受作为参数来使用的对象。这意味着不会用下面的方式来调用Order对象上的方法。

```
myOrder.GetTotalValue();
```

相反，我们将用以下方式来调用服务层方法。

```
OrderServices.GetTotalValue(myOrder);
```

这种过程语法可能会遭到一些OO纯化论者的强烈反对，但我认为将领域模型的行为与结构分离开来的获益足以抵偿这种略显拙劣的语法了。

当然，服务方法不必是静态方法，有时它们也不能是静态的，例如，如果当它们应该初始化一个声明式转换时，或者当需要使用依赖注入来配置SL类时，就不应使用静态服务方法（参见第10章）。

但是当可以使用静态方法时，我通常会使用它们，原因很简单：这可以减少代码（每次需要调用一个服务时，不必浪费一行代码来实例化SL对象）。由于可以减少代码，因此我们的示例中仍将继续使用静态服务方法。

这里的重点是，我们实际上并未失去很好地组织方法的机会，也就是说，可以把方法组织为就好像把它们放到领域模型类上一样。因为每个领域模型类都有一个对应的服务层类，因此可以把业务逻辑划分为就好像分布到实际领域模型上一样。

这里的获益体现在以下方面：当基本数据结构未发生改变时，`GetTotalValue()`的规则却发生了改变，例如当突然要实现一条系统级规则的时候（订单总价值不能超过100万SEK）。这样，就可以只修改相关的服务层方法，而不必重新编译（甚至无需访问）领域模型层，这样领域模型层可以保持已通过测试和受信任的状态。

现在回到`GetOrdersByCustomerID()` Web服务，我们只添加了一个对服务层`OrderServices`。`GetTotalValue()`方法的调用，我们在序列化例程期间将每个订单传递给此方法，并在XML文档中添加一个字段，用于保存结果。

这为我们提供了一个很好的示例，说明了如果客户应用程序开发人员选择实现一个客户端领域模型的话，此模型与服务器端领域模型会有什么不同。客户领域模型`Order`对象可能包含一个总价值属性，它用于描述XML文档中所显示的总价值，但服务器端领域模型`Order`对象却没有这个属性。

采用这种方法，可以在XML文档中包含更多的字段，用于保存服务层方法计算出来的值。

例如，考虑“客户的应付款项不能超过一定额度”这条系统级规则。不同客户的额度是不同的，而且通过特殊的服务层方法计算。当设计从`GetCustomersByFilter()` Web服务返回的XML文档时，可能需要包含一个用来存放每位客户最大欠款额的字段，或者至少需要有一个标志来表明客户是否超过了限额。

这说明服务器提供的文档不一定与领域模型匹配，因为客户与服务器之间的通信完全是面向文档的。

看一下目前的结果，富客户应用程序的用户现在应该能够查看过滤后的客户列表，并查看属于每位客户的订单列表。接下来，用户需要检出属于特定订单的订单行。

实现此目的的方法并没有什么新的内容：客户通过订单的ID调用Web服务，Web服务将调用转发给服务层方法，并将结果序列化为XML文档。现在，我们知道了客户获取数据的方法，那么客户提交数据又是什么情况呢？

### A.1.8 提交数据

假设用户需要将订单的状态从“Not inspected”更新为“Approved”或“Disapproved”。乍看上去，这像是一个直观的请求，但实际上我们正在打开一个并发性问题的潘多拉盒子，因为我们从只读操作转变为了读写操作。

那么就让我们暂时把它当成一个直观的请求，而不做更深入的考虑。这样，接下来应该做什么？

第一步是创建一个`UpdateOrderStatus()`服务层方法以及一个相应的Web服务。方法应该接



受订单的ID和新订单状态（作为参数），并返回一个表示操作是否成功完成的布尔值。（这里不讨论任何高级的跨平台异常处理，尽管这是一个非常有趣的话题！）

到目前为止一切都很好。服务层方法很容易实现，而且Web服务只是一个包装器，它将调用转发给服务层方法。现在来看一下将会遇到的问题。

如果两个用户试图同时更新同一个订单的状态，那么就会出现問題。假设用户A获取了一个订单，将它标记为“Not inspected”，然后检查它。不久之后，用户B获取了同一个订单，并做了同样的标记，然后也决定检查它。然后，用户A发现了订单的一个小问题，因此在随后的UpdateOrderStatus() Web服务调用中将该订单标记为“Disapproved”。

在用户A刚刚完成对订单状态的更新后，用户B继续调用同一个Web服务来更新同一个订单的状态。然而，用户B未注意到订单的问题，因此想将它标记为“Approved”。那么用户B是否应该能够重写用户A已提交并保存在数据库中的订单状态呢？

标准答案是：“不，因为用户B不知道用户A已经做出修改。”如果用户B已经知道用户A做出了修改，并仍然决定重写它，那么答案就是“是的”，因为这样就可以确信B不会错误地重写A的数据。因此，问题就变成这样：如何跟踪B或者任何其他客户端“知道”什么？

一种方法是跟踪要更新的字段在数据库中的初始值，然后当保存已修改的数据时，检查原始值是否与当前值匹配。如果数据库中的值已更改，那么客户端就不知道这些更改，因此应丢弃客户端的数据。

这种方法称为乐观并发。下面看一下用一种非常简单的方式来为我们的示例实现乐观并发。

首先，客户端应用程序必须这样编写：当用户更改订单状态时，通过XML文档从数据库返回的初始状态仍然保持不变。

然后，必须修改UpdateOrderStatus() Web服务和同名的服务层方法，以接受一个额外的参数，即初始订单状态值。因此，当客户端调用Web服务时，传递三个参数：订单的ID、新订单状态和初始订单状态。

最后，应该更新UpdateOrderStatus() 服务层方法，以便将一个检查包括进来，即检查数据库中的当前订单状态是否与传递给初始订单状态参数的状态匹配，只有当值是匹配的时候，才执行更新操作。

这种版本的乐观并发（即用数据库中的状态值来匹配更新后的字段的初始值）将提供更好的性能和可伸缩性，但实现起来略显麻烦，至少在富客户中是这样的。另一种使用乐观并发的方式是在数据库表中添加一个版本控制字段。

有时像乐观并发这样的策略也是无效的，因此必须应用特定于领域的解决方案。一种常见的方案是使用一种检入/检出系统，当用户获取某个要进行写访问的对象时，将此对象标记为“checked out”或“locked”，以便阻止其他用户更新它，直到用户完成操作。其他更高级的方案还包括对同时更新的数据进行合并。

决定在特殊情况下适用哪种策略的唯一方法就是咨询领域专家，也就是说，理解这些决策的业务含义（需要哪些行为）的人。并发策略的选择没有通用方法。

尽管如此，我认为在本例中，使用乐观并发是适当的，有无版本控制字段均可。关键仍然是

应该由某个知道业务需要什么预期行为的人来做出决定。

或许这里所用方法的最大问题是我们已经要求客户跟踪初始值。这可能要求客户应用程序开发人员付出过多的努力,因此更好的替代方案将初始值存储在服务器的Session变量或类似的结构中。

### A.1.9 粒度

另一个要考虑的问题是服务粒度是否过细了:是否不必为每个可更新的订单字段提供一个单独的Web服务,而是实现一个单独的UpdateOrder() Web服务呢?

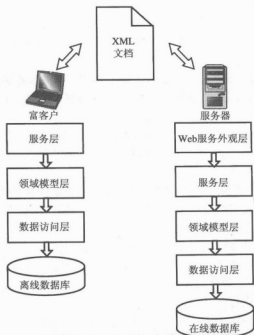
在这种情况下,Web服务不必为每个可更新的字段提供一个参数,而是可以接受一个表示更新后订单的XML文档。也就是说,可以不使用以下代码。

```
[WebMethod]public bool UpdateOrder(int orderID, int orderStatus
, DateTime orderDate, int customerID)
```

而是这样:

```
[WebMethod]
public bool UpdateOrder(string xmlOrder)
```

采用这种方式,客户以XML文档的形式同时接收和发送数据。这种完全面向文档的方法有助于保持Web服务API的粗粒度,并避免客户与服务器之间过于频繁的会话(参见图A-4)。



图A-4 完整应用程序架构概览

### A.1.10 简单讨论一下事务

最后,说明一下事务。如前所述,在不需要事务的时候,我通常使用静态的服务层方法,但在业务应用程序管理订单的情况下,我相信事务应该作为解决方案的一部分。

举例来说,这暗示着UpdateOrderStatus()方法应该是事务的。这意味着它不应该再是静态方法,而应该将它改变为一个实例级方法,并用[AutoComplete].NET属性来标记它。由于我还打算用[transaction]属性来标记此方法所属的整个类,并令其从ServicedComponent继承,因此决定将此方法从OrderServices类中提取出来,并放到一个名为OrderServicesTx的新类中,其中“Tx”表示“事务的”。

这样就足够了。UpdateOrderStatus() Web服务现在要做的只是创建OrderServicesTx类的一个实例,并调用该对象上的UpdateOrderStatus()方法,而不是调用OrderServices类上的原有静态方法。OrderServicesTx.UpdateOrderStatus()的执行将使用分布式事务,如果方法完成,并且未产生异常和回滚,那么这些分布式事务将自动被提交。

服务层是在应用程序中实施事务控制的最佳位置,也是进行安全检查、日志和其他类似操作的理想位置。

注意,服务层中的所有方法不一定围绕来自领域模型层的实体进行组织。有了EmployeeServices和CustomerServices等一些类,并不意味着在服务层中必须有LoggingServices和SecurityServices类。

### A.1.11 小结

我的首选风格是把大部分业务逻辑放到服务层中,并且令业务逻辑在由领域模型表示的面向对象数据结构上进行操作。客户与服务器之间的通信完全是面向文档的,不尝试将领域模型导出到客户,甚至不让客户访问(例如,通过远程访问)服务器端领域模型。

我希望已经讲清楚了为什么要使用领域层和如何使用它,以及我将如何设计Jimmy的应用程序。

## A.2 数据库模型就是领域模型

作者: Frans Bouma

为了在语义的基础上使用数据,一种有用的方式是指定一部分逻辑将会使用的元素的通用定义。例如,一个订单系统使用Order元素,当然还有其他元素。为了能够定义此逻辑的工作方式,一种实用的方式是给出Order概念的一个定义:我们能够通过指定Order元素上的行为来描述系统功能,并提供Order元素的定义。

这个Order元素包含其他元素(像OrderID和ShippingDate这样的值),并与另一个元素OrderRow具有紧密的连接,而OrderRow元素也包含其他元素。我们甚至可以说Order包含一个OrderRow元素集,就像它包含值元素一样。那么,包含值OrderID与包含OrderRow元素集有什么区别吗?当订单系统进入程序编码阶段后,这个问题的答案对于Order概念的进一步实现方式非常重要。

## A.2.1 实体的特性

看一下E.F. Codd博士开发的关系模型[Codd Relational Model], 它们应被看作两个独立的元素: Order和OrderRow, 它们之间具有关系, 而且元素本身形成了一种基于它们包含的值(例如OrderID)的关系。

然而, 看一下Martin Fowler[Fowler PoEAA]和其他人开创的领域模型, 它就不必采取这种方式。我们可以将集合中的OrderRow元素看作一个Order值, 并把Order作为一个单元(包括OrderRow元素)来使用。

### 1. 什么是实体

1970年, Peter Chen博士为他的实体关系模型[Chen ER]定义了实体概念, 他的模型是在Codd的关系模型基础上构建的。当在关系模型中定义Order和OrderRow时, 实体的概念是非常有用的。Chen将实体定义为:

“**实体和实体集**。用e表示我们所思考的实体。实体可分为不同的实体集, 例如EMPLOYEE、PROJECT和DEPARTMENT。每个实体集都有一个相关的断言, 用于测试某个实体是否属于它。例如, 如果已知某个实体位于实体集EMPLOYEE中, 那么它应具有该实体集中其他实体所具有的公共属性。其中的一个属性就是上面提到的测试断言。”

使用实体关系模型, 能够定义像Order和OrderRow这样的实体, 并把它们放到关系模型中, 这里, 关系模型定义了数据库。但是, 使用Eric Evans的实体定义, 我们距离关系模型就很远了, 这种实体定义可以总结如下: “通过不同状态或甚至跨不同实现跟踪的对象。” Evans的定义与Chen的实体定义之间的重要区别在于, Chen的定义是一个抽象元素, 它的存在不需要有状态, 甚至无需物理表示。而在Evans的定义中, 实体是物理存在的, 它是一个对象, 具有状态和行为。使用抽象实体定义时, 我们不受实体数据所在上下文的影响, 因为实体数据不是由实体本身解释的(因为实体中没有行为), 而是由外部逻辑进行解释。

为了避免误解, 我们采用Peter Chen博士的定义。原因是它为我们每天都会遇到的事物(无论是物理项, 还是虚拟项)定义了一个抽象术语, 而无需考虑上下文或其中包含的数据, 因为只有定义才是重要的。因此, 它是描述像Customer或Order这样的关系模型元素的理想候选。这样, 一个物理的Customer就被称为一个实体实例。

### 2. 实体驻留在什么地方

每个应用程序都必须处理状态。状态实际上是一个过于泛化的术语。大多数应用程序都具有几种不同状态, 用户状态和应用程序状态是其中最重要的状态。用户状态可以被看作是所有对象/数据存储的状态, 这里, 在给定时刻T, 数据存储在每个用户的基础上(即, 具有“用户范围”)保存数据。用户状态的一个例子是在ASP.NET应用程序中, 在给定时刻某个给定用户的会话对象的内容。应用程序状态则与用户状态不同, 我们也可以将它看作所有对象/数据存储的状态, 但在这里, 数据存储是在应用程序范围的基础上保存数据。一个例子是在给定时刻, 某个给定Web应用程序中所有用户共享的数据库的内容。将用户状态看作应用程序状态的一个子集是不明智的, 因为当用户处在一个5步骤的向导中间时, 用户状态保存了步骤1和步骤2的数据, 但是, 应用程序状态并未发生任何改变, 而当向导完成后, 应用程序状态才会发生改变。

定义实体实例的驻留位置是很重要的：在应用程序状态中还是在用户状态中。如果实体实例驻留在用户状态中，那么它对于拥有用户状态的用户来说就是本地的，并且其他用户看不到该实体，因此也不能使用它。当创建实体实例时，例如在上述订单系统中实际创建一个订单，它驻留在实际的应用程序中，并且是应用程序状态的一部分。然而，在订单创建过程中，举例来说，当用户填写订单表单时，实际上并没有创建订单，这时用户状态中是一个临时的数据集，它结束之后才变成订单。当在应用程序状态中实际创建订单时，实体实例才会被保存。

应用程序状态有几种不同类型：共享的、保存实体实例的内存中系统（in-memory system）或者将实体实例保存在数据库中。大多数软件应用程序在处理实体实例时使用某种类型的持久化存储来存储实体实例数据，以便在电源中断或发生其他宕机事件而导致内存内容丢失时不丢失数据。如果应用程序使用一种持久化存储，那么就可以将持久存储中的数据称为实际的应用程序状态：当应用程序宕机时（例如，为了维护目的），应用程序并不丢失任何状态，即不丢失订单实体实例，当应用程序再次启动时，状态仍然可用。因此，内存中的实体实例是持久化存储中的实际实体实例的一个镜像，而且应用程序逻辑使用该镜像来更改驻留在持久化存储中的实际实体实例。

### 3. 将类映射到表上与将表映射到类上

O/R映射负责处理关系模型与对象模型之间的转换，也就是说，在对象与持久化存储中的实体实例之间进行转换。可以将它概括地定义为：

将对象模型中的类的字段关联到关系模型中的实体的属性，反之亦然。

这就出现了一个鸡生蛋还是蛋生鸡的问题：哪个在前，哪个在后？是首先定义实体类（表示实体定义的类，例如表示Order实体的Order类），并使用这些类来创建带有属性的关系模型实体，还是首先定义关系模型，并在定义实体类时再使用此关系模型？

像其他事物一样，这个问题没有明确的答案。可能最好的答案是“它取决于什么”。如果采用领域模型方法，则可以从领域开始，用领域模型来定义类，一些类可能还位于继承层次结构中。使用这种类模型，只需用一个关系模型来存储数据，这个关系模型甚至可以是一个带有主键的表，包括对象ID、一个二进制的blob字段（用于对象）以及一些用于描述对象的元数据元素。这样，当我们确定关系模型最符合对象模型的需要时，就可以很自然地将类映射到关系模型中的元素上。

如果从关系模型开始，并构建一个E/R模型，则可能需要将关系模型中的实体映射到类上。这与领域模型方法不同，原因是关系模型不支持继承层次结构：无法将Person ← Employee ← Manager这样的层次结构建模为一个具有层次结构的关系模型。当然，可以创建一个在语义上被解释为继承层次结构的关系模型，然而，根据定义它不代表继承层次结构。

这是两种方法之间的根本区别。从类到数据库的方法使用的是关系模型，数据库只是存储数据的地方，而从关系模型到类的方法是将类用作以OO风格来使用关系模型的一种方式。

由于我们选择了使用Chen的实体定义方式，因此将使用首先定义关系模型的方法，然后再得到类。后面在“将NIAM/ORM用于类和数据库的理想世界”一节，我们将看一下如何衔接这两种方法。

### 4. 以OO风格来使用数据

表示实体的类是开发人员在代码中定义实体的方式，正如一个物理实现的带有表的E/R模型

定义了持久化存储中的实体一样。使用前面讨论的O/R映射技术，开发人员可以使用实体类实例中的内存镜像来操纵持久化存储中的实体实例。这通常是一个批处理风格的过程，因为开发人员在工作时并不连接到持久化存储。控制环境是O/R映射工具，它控制持久化存储中的实体实例与驻留在实体类实例中的内存镜像之间的连接。

开发人员可能会要求O/R映射工具将一个给定的Order实例集合加载到内存中。这将导致在实体类实例中为持久化存储中的每个Order实例创建一个镜像。开发人员现在就能够通过实体类实例来操纵每个实体实例镜像，或者在一个窗体中显示实体实例镜像，或作为一个服务输出来提供它。必须保存被操纵的实体实例镜像，以便保存更改。从开发人员的观点来看，这是把对象中被操纵的实体实例数据保存到持久化存储中，就像用户把一段在字处理器中编写的文本保存到文件中一样。O/R映射工具为开发人员执行这种保存操作。但由于我们正在使用镜像，因此O/R映射工具实际执行的操作是用镜像中存储的修改（是从开发人员的代码接收的）来更新持久化存储中的实体实例。

关系模型中实体之间的关系在代码中是通过O/R映射工具提供的功能来表示的。这样，开发人员能够遍历从一个实体到另一个实体的关系。例如，在订单系统中，通过将Customer实例加载到内存中，开发人员能够利用O/R映射工具所提供的功能到达Customer的Order实体实例，这里，O/R映射工具提供的功能可以是Customer对象中的一个集合对象，也可以是向O/R映射工具发出的一个新的请求（请求与给定Customer实例相关的Order实例）。

这种实体使用方式在很大程度上是静态的：在运行时通过来自几个相关实体的属性组合来构造实体并不会得到表示实体的类，因为类必须在编译时存在。这并不意味着通过属性组合（例如，通过一个带有外部连接的选择）在运行时构造的实体实例无法被加载到内存中。然而，它们不表示一个可持久化的实体，而是表示一个虚拟实体。这个额外的抽象层主要用于只读场景中，例如在报告应用程序和只读列表中，在这些场景中，通常需要来自相关实体的属性组合。这样的列表的定义的一个例子就是Order实体的所有属性与Customer实体的“公司名称”属性的组合。

要成功地以OO风格使用数据，关键在于控制实体实例的内存镜像与物理实体实例之间的连接的功能应该足够灵活，以便可以定义报告功能和组合属性集的列表，并把它们加载到内存中，而不必仅仅为了以更动态的方式使用实体实例中的数据而使用另外一个应用程序。

## A.2.2 将功能研究作为应用程序的基础

为了有效地设置关系模型，也就是设置关系模型中实体定义与表示实体的类之间的映射以及这些类本身，关键是要重用软件开发项目前期（即功能研究阶段）的工作成果。这个阶段通常使用更经典的软件开发方法。在这个阶段中，我们用一种抽象方式来确定功能性需求和系统功能，并记录它们。多年以来，人们已经定义了几种技术来帮助完成这个阶段，其中的一种技术就是NIAM [Halpin/Nijssen Conceptual Schema]，后来T.A. Halpin [Halpin IMRD]将它进一步发展为对象角色建模（ORM）。使用NIAM和ORM，可以用更易于理解的句子与客户交流功能研究结果，例如“Customer has Order”和“Order belongs to Customer”。然后，这些句子可用于在抽象的NIAM/ORM模型中定义实体和关系。通常，用一种可视工具来完成此任务，例如Microsoft Visio。

## 1. 功能研究结果的重要性

用NIAM或ORM这样的技术对研究结果进行建模的优点是，抽象模型既记录了功能研究阶段的研究结果，同时它也是应用程序要使用的关系模型的来源。使用Microsoft Visio这样的工具，可以通过从NIAM/ORM模型来生成E/R模型来生成关系模型，此模型可用于在数据库系统中构造物理关系模型。然后，就可以使用在数据库系统中构成关系模型的元数据来生成类和结构映射。

这样做的优点在于，开发人员在项目早期研究中所使用的类层次结构有一个理论基础。这意味着当应用程序的设计（例如一个功能片段）发生更改时，可以沿着相同路径来更改：更改NIAM模型，再用新的E/R模型（此模型是用更新后的NIAM模型创建的）来调整关系模型，然后调整类以符合新的E/R模型。相反方向也是正确的：找到一个开发人员必须使用的代码结构的原因。例如，为了使代码结构能够遍历实体实例对象之间的关系，只需沿着相反路径从类返回功能研究结果，并揭示出代码结构的理论基础。这种理论基础与实际代码之间的健壮连接是开发成功的、可维护的软件系统的关键。

## 2. 功能流程是数据使用者，也是业务逻辑所在的地方

由于实际的实体定义位于关系模型中（即在数据库的内部），而且内存中的实体实例只是数据库中实际实体实例的镜像，因此在这些实体中没有放置行为（或者说业务逻辑规则）的位置。当然，向实体类添加行为是很容易的。问题是当实体类表示关系模型中的实体定义时，这是否符合逻辑。答案取决于要作为行为添加到实体的业务逻辑的类别。业务逻辑可粗略地分为三类。

- 面向属性的业务逻辑。
- 面向单一实体的业务逻辑。
- 面向多个实体的业务逻辑。

面向属性的业务逻辑包含像`OrderId > 0`这样的规则。这些都是非常简单的规则，它们是施加在单个实体字段上的约束。当将实体字段设置为一个值时，可以实施这个类别中的规则。

面向单一实体的业务逻辑包含像`ShippingDate >= OrderDate`这样的规则，这些规则也作为约束。当将实体加载到内存中的实体对象时，或将其保存到持久化存储中时，或测试实体在给定上下文中是否有效时，可以实施这个类别中的规则。

面向多个实体的业务逻辑包含跨一个以上的实体的规则：例如，用于检查一个Customer是否为Gold Customer的规则。为了使规则为真，它必须查询与该Customer相关的Order实体以及这些Order实体相关的Order Detail实体。

所有这三个类别都对实体所在的上下文有依赖性，尽管并不是给定类别中的所有规则都是依赖于上下文的规则。面向属性的业务逻辑中的大多数规则均不绑定到实体所在的上下文，而且很适合作为行为添加到实体类中。面向单一实体的业务逻辑通常不适合作为行为添加到实体类中，因为当在另一个上下文中使用实体时，这个类别中的大部分规则（这些规则用来使实体在给定上下文中有效）将会发生变化。面向多个实体的业务逻辑中的规则跨越多个实体，因此无法放在单一实体中，此外它们与所在上下文的绑定程度很高。

可插入的规则。为了令实体可用作一个不被绑定到给定上下文的概念，可以利用可插入的规则来解决面向属性和面向单一实体业务逻辑中与上下文绑定有关的业务逻辑规则问题。可插入的规则是指包含业务逻辑规则的对象，它们在运行时被插入到实体对象中。这种方法的优点是实体类与上下文之间没有绑定，因此可在系统设计所需的任何上下文中使用；只需为每个上下文创建一组可插入的规则对象，甚至可以为每个实体创建更多可插入规则对象，而且根据上下文状态的不同，可以通过简单地设置对象引用将来将规则应用于实体。决定将哪些规则对象插入到实体中的过程就是维护实体所在上下文的过程，也就是表示实际业务流程的过程，实际业务流程被称为功能流程。

**功能流程。**我们在“功能研究结果的重要性”一节中描述了功能研究阶段，并强调了保持功能与实际实现之间的健壮连接的重要性。通常，系统必须实现特定业务流程的自动化，而且功能研究将以抽象形式来描述这些流程。为了尽可能保持研究与实现之间的紧密连接，一个常见的步骤是在对抽象业务流程建模之后，对实际实现进行建模，这样就得到了称为功能流程的类，因为它们或多或少是一些不包含数据的类，而只具有单一功能。

功能流程是实现面向多个实体的业务逻辑规则的理想候选。在我们的Gold Customer示例中，可以把“将Customer升级为Gold Customer”这个流程实现为一个功能流程，它使用一个Customer实体对象及其Order实体对象，更新Customer实体的一些字段，并在升级过程完成后保存该Customer实体。此外，由于功能流程实际上执行业务流程的步骤，因此它们也是上下文应该出现的地方（实体在此上下文中使用），也是在给定时刻T，在给定上下文状态下，决定将哪些规则插入到实体类中的唯一正确的地方。

### 3. 将NIAM/ORM用于类和数据库的理想世界

对于大多数人来说，现实情况并不总是符合我们所期望的理想世界，日常的软件开发也不例外。在功能研究小节中，物理关系模型元数据用于生成映射和类，从而得到开发人员需要的实体类定义。一种更理想的方法是直接使用NIAM/ORM模型来生成实体类，这样就不用从元数据到类定义的转换了。这使得我们更接近最终目标，即应用程序的设计像应用程序本身一样便于使用。

我们很难确定这个理想世界何时将会变为现实，甚至无法确定它是否会成为现实，因为在计算机科学世界的外部存在大量因素，它们影响着软件项目的成功。不管怎样，一件有趣的事情是明白用当今开发的技术（例如模型驱动的软件开发）能够完成什么任务。

## A.3 实用主义和非传统方法

作者：Ingemar Lundberg

当Jimmy最初询问我是否愿意撰写这篇“客座章节”时，我立即回答说：“是的，当然。”我必须承认我有点过于自信了。不出所料，我很快就对自己产生了怀疑。我要讲些什么内容？是否能够在有限的篇幅内足够精确地介绍这些内容？最重要的是，如何能让我的.NET架构适合领域模型概念？

我将尝试概括地描述一下我为前任雇主构建的架构，从现在开始，我将它称为我的架构。但是，它并不是我近来所使用的架构。我还将讨论Jimmy的实践中的一些问题，这些问题对于我的



架构中的强项和缺点很有意义。在这个有限的篇幅中可能无法给出全面的讨论,但由于本节的目的是展示解决一个问题可以有多种方式,因此我希望这篇简短描述能够有效地实现这个目的。注意,当我们讨论如何完成操作时,上下文是这个特殊架构,而不是通用架构。

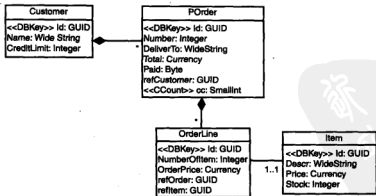
### A.3.1 背景和历史

这里需要介绍一些背景和历史,目的是解释一下在我的架构中采取一些操作的原因。这里要指出有二个或三个事件对我的工作状态产生了主要影响。首先,在.NET之前,我曾经设计了一个MTS/COM+架构。当.NET问世时,旧架构(自己开发的)概念刚刚被开发出来几年时间,因此才刚刚“融合”到人们(开发人员)的思想中,而且丢弃它们的代价过于昂贵了<sup>①</sup>。第二件事就是我的架构不一定是“最好的”。我希望它在设计艺术/开发技术水平、个人使用它的技巧和所需的性价比之间取得良好的平衡。

### A.3.2 架构概述

那么,我的架构的概念是什么?熟悉MTS/COM+的读者应该深入学习过无状态OO编程的概念(这是否是术语上的矛盾?)。过程对象[Ewald TxCOM+]是一个无状态对象,它表示一个实例中的单一、完整的业务对象(BO)。它包含某个特定业务对象类型的一个或一组业务对象的全部操作集。无状态实现了这种可能性。

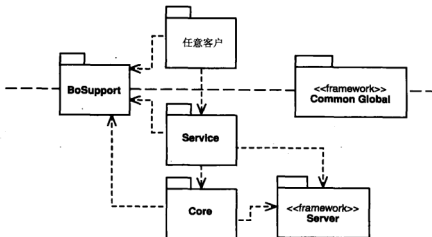
架构的基础是一个协作类的框架。它本身就是可用的,但为了真正有效地使用它,需要有一个辅助工具BODef(图A-11提供了它的一部分快照),我们在这个工具中定义业务对象,并生成样板代码(boilerplate code)。代码生成器不仅仅是一个向导,因为它允许反复生成代码,而又不失手工添加代码的功能。它使用Generation Gap模式[Vlissides Pattern Hatching]来完成此任务。图A-5显示了当前问题的BO的一个(部分)定义。



图A-5 示例的业务对象

① 如果读者是IT部门中的技术领先者,那么我确信当你已经准备好做下一件事时,你会注意到周围的人实际上尚未领会你最近正在谈论的事情。

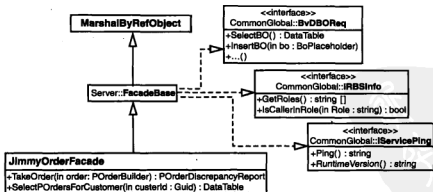
图A-6显示了划分应用程序类的“标准”方式。所有包/程序集（标记为<<framework>>的那些除外）都是特定于应用程序的。BoSupport程序集：不仅在中间层中是可用的，而且在远程客户中也是可用的。在Jimmy要解决的问题中，所需的富客户将通过远程连接与中间层进行通信<sup>①</sup>。



图A-6 总体架构

Service程序集至少应包含一个外观层对象（façade object），它负责实现一些用于支持基本CRUD操作的基本接口，IBvDBOReq是最重要的外观层对象，参见图A-7。

这个外观层不是生成的，相反，我们从FacadeBase继承所有基本功能，包括上述接口的实现。



图A-7 外观层的架构

在持久化存储方法方面，关系数据库系统（RDBMS）就像文件系统一样是不可或缺的。事实

① Jimmy的说明：正如本附录的引言中提到的，开始时是需要应用服务器的，但后来就不需要了。

上，RDBMS是一项不言而喻的需求，如果想对应用程序数据进行正确的备份处理，那么最好使用一种关系数据库系统。这是一条重要的声明。这就是为什么我们无需假定不使用RDBMS的原因。

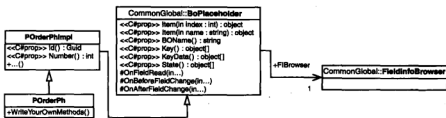
### 1. 数据检索

从中间层返回给客户的数据可以使用任何格式：DataTable、DataSet、强类型的DataSet、XML或序列化的对象图，以上列出的是几种主要的格式。最简单也是最常用的返回格式（在我的示例中）是DataTable。要返回一批“业务对象数据”，只需将每个业务对象的数据作为一行返回。

### 2. 业务对象占位符

DataTable中的一行很容易转换为一个业务对象占位符BoPlaceholder。BoSupport程序集中的BoPlaceholder类可以是一个独立使用的类，但它也是特定业务对象占位符的基类。

特定占位符是由BODef生成的，例如图A-8中的POrderPh（采购订单）。每次选择生成时，都会生成一个Impl类。叶类（leaf class）只生成一次，这在很大程度上是Generation Gap模式的本质。



图A-8 BoPlaceholder的使用

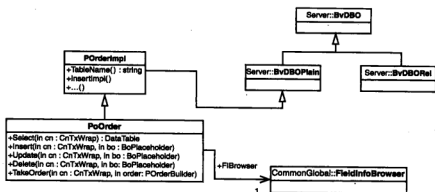
可以将这些占位符对象看作是信息载体或数据传输对象，但它们也可以在更复杂的行为场景中使用。而且，派生的BoPlaceholder位于BoSupport程序集中，它既存在于中间层中，也存在于客户层中（富客户/胖客户）。

可以将DataTable从中间层返回给客户，并从DataRow实例转换为BoPlaceholder实例。也可以返回一个直接在客户上使用的BoPlaceholder的集合（例如ArrayList）。BoPlaceholder是可序列化的。但是要记住，我们可以找到很多如何对DataTable进行数据绑定的样例，但将数据绑定到BoPlaceholder的样例却是非常少见的（尽管有可能）。

### 3. 过程对象

业务规则的最后执行和实施是核心业务对象的工作，核心业务对象也称为过程对象。这些对象是无状态的，而且生命周期很短。图A-9中显示了POrder过程对象，它们遵循Generation Gap模式。这些对象是在Core程序集中实现的。由于我认为授权是业务规则的一个核心部分，因此在这些对象的（公共）方法上指定角色就不足为奇了。

尽管业务规则的实施是核心对象的责任（在图A-9中，POrder就是一个核心对象），但业务规则仍然很有可能是在BoSupport程序集中实现的，以便可以在富客户中使用它们。然而，当我这样做的时候，总是在保存更改之前对过程对象中的规则检查两次。我们知道，客户在把规则提交给中间层中的外层对象之前，很有可能已经重写了某条规则（或许是通过反射重写的），例如提高了订单限额。

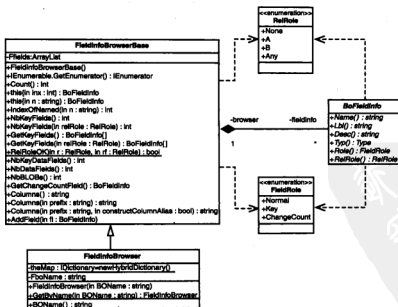


图A-9 在POrder中定位自定义规则，在本例中是TakeOrder()方法

过程对象通常具有BoPlaceholder，并“隐藏”在外层对象后面。从概念上讲，过程对象和占位符（例如POrder和POrderPh）都是相同（逻辑）业务对象的一部分。

#### 4. 元数据

读者可能已经在前面的一些图中注意到，在框架类中有很多可用的元数据。图A-10给出了完整的元数据视图，但并未对其进行深入解释。元数据可以在数据存储处理和UI处理中提供一些帮助。



图A-10 完整的元数据视图

中间层中的SQL语句是从元数据动态创建的，而且我们可以得到一个非常好的、经过检验的实现。如果发现瓶颈问题，可以选择用自己的方式来实现SQL处理，例如使用存储过程。

### A.3.3 Jimmy 的订单应用程序演习

让我们将注意力转向Jimmy的问题，即订单处理应用程序。假设“下订单”功能（即申请）的用户通过电话与客户交谈（即为客户服务）。我们将这个人称为接单员。如果读者注意到我的观点的话，在某种程度上胖客户暗示出了这一点。很容易看出，我们需要一个电子商务应用程序，即使现在不需要，将来也是需要的。我认为这对设计并不会产生很大的影响，但我对场景的解释仍然会影响系统中需要的角色。现在有用户（User）和管理员（Admin）角色。用户可以执行大部分操作，管理员则可以执行任何操作。（电子商务应用程序可能需要一个PublicUser角色。）

为了保持简洁，这里删除了大部分真实世界的属性。大部分属性并不难处理，例如客户的收票人的地址和订单发货地址。在实际应用程序开发的早期阶段，我通常集中精力关注对象之间的关系（当然，也关注它们的行为和协作），而不是具体的属性。

#### 1. 简单的注册处理

这个问题要求对客户数据进行处理。我将这称为最基本的注册处理功能。下面将展示我的解决方案，它不仅揭示了我的一些思想，而且也体现了将业务对象的元数据包括进来的潜力。

图A-11是我的业务对象定义工具（即BODef）的一个快照，它显示了Customer对象。可以看到，除了字段名称及其类型之外，字段还附加了更多信息。

Name	Type	Role	Label	Desc
id	GUID	DBKey	id	id
Name	WideString	Customer name	Name of the customer	Name of the customer
CreditLimit	Integer	Credit limit	The amount the customer may owe us	The amount the customer may owe us

图A-11 BODef中的Customer类

图A-12显示了订单应用程序中的一个窗体，标签的数据来自元数据。利用这个网格，可以添加、编辑和删除客户。实现此窗体的代码非常简单，（参见以下代码清单）。关键在于通用的TableHandler类，它充分利用了Customer BO的元数据，以及通用的外观层接口IBvDBOReq，并且也充分利用了架构的BoPlaceholder和DataRow关系。

Customer name	Credit limit
JNSK	50000
Lundberg's Software	50000

图A-12 Customer的一个默认实现

```
public CustomerFormList()
{
    InitializeComponent();
}
```

```

_tabHndlr = new TableHandlerGui(SvcFactory.Facade,
    CustomerPh.FieldInfoBrowser.BCName);
_tabHndlr.TableChanged += new
    TableHandler.TableHandlerEventHandler
    (TableHandlerEventHandler);
_tabHndlr.HookUpGrid(grd);
_tabHndlr.FillGrid();
}
TableHandler _tabHndlr;

string[] confirmString = new string[] {
    "Do you want to add Customer?",
    "Do you want to save changes to Customer?",
    "Do you want to delete Customer?" };

void TableHandlerEventHandler(object sender,
    TableHandler.TableHandlerChangedEventArgs args)
{
    args.AcceptChanges = MessageBox.Show(
        confirmString[(int)args.TableHandlerChangeType],
        "Confirm", MessageBoxButtons.YesNo) == DialogResult.Yes;
}

```

讨论每个小的细节没有什么意义。读者可能想要获得一些如何处理元数据的思路。除了这里显示的之外，元数据还可以描述业务对象之间的关系。（在我的架构中，关联对象是“一等公民”。）

## 2. 下订单的业务规则

从Jimmy的问题描述的字里行间中可以看出，这正是他希望我们这些被邀请的作者深入探讨的地方。当然，我很愿意探讨一下。

首先讲一下问题描述中的哪些内容是我们不想讨论的，但这并不是说不实现它们。我们将忽略搜索问题。在我的优先级列表中，由于篇幅不足，它被省略了。同样省略的还有为每个客户列出的订单，以及新客户的信用检查。在我的伪实现中，我将订单的唯一编号归类为一个较次要的问题。我只是在订单被提交时为其分配一个编号，接单员在该时刻返回它，而且在此之前不需要它。

这里为订单接受/填写过程提供了良好的支持。我描述了一个“构建器<sup>①</sup>”，它用于操纵订单表（可以把它想象成一张纸），订单表作为一个隐喻，我将围绕它进行推理。我也将这个构建器想象为一个可序列化的对象，在完成时，将此对象从客户提交给中间层。

多用户方面增加了一些需要解决的难题。例如，当接单员与假定的客户对话时，如果某个货物的价格发生改变，那么接单员是否应该立即知道这一点？用货物存储库（item repository）来检查价格是毫无帮助的，因为如果在上午10:55检索价格并将其显示到屏幕上，很有可能在上午10:56价格就改变了。库存也存在同样的问题，是否有货？还是需要从供应商订货？

解决此问题的一种方式，一旦价格或库存发生改变，中间层立即通知所有客户（接单员），

<sup>①</sup> 举例来说，读者可以设想一下用于构建字符串的StringBuilder类。

以便客户可以对此采取行动。然而，这将使事情变得相当复杂。客户不再只是与中间层对话，中间层将主动与所有（活动的）客户对话。除此之外，时限问题并未消除，只是最小化了。

我想在保持简单的同时仍然支持这些需求。我需要关注一下业务流程。接单员通过电话与客户交谈。他们讨论客户的需求、价格和货物的可用性。最终客户决定购买，接单员将订单提交给中间层。作为响应，中间层返回一个报告。此报告表明最后的价格和可用性。可用性将对交付日期产生影响。报告中标明与先前（口头）给出的事实的所有不符之处（如果有的话）。接单员与客户讨论这些不符，如果客户对此不满，可以取消订单。取消订单是其自己的一个事务，即撤销/删除事务。

我知道，如果为客户提供了一个价格，那么应该保持这个价格，至少在某种程度上应该如此<sup>①</sup>。这很容易纠正。一种简单的方式是将价格设定工作只留给少数几个人，而且他们只在下班之后才更改价格。另一种方式是令系统本身将价格更改推迟到下一个非交易时刻再生效。如果客户缓存了价格信息，那么必须知道如何在适当的情况下刷新（或删除）缓存。

然而，可用性的问题（即库存）与价格正好相反，它是一个动态问题。因此，需要提供一个报告，作为一个已提交的订单的响应。这个报告还可以报告所发现的任何价格上的不一致（即使已经实现了一个避免价格不一致的策略），只是为了确保这一点。报告还将最终确定订单是否在全局最大限定之内，以及客户的欠款是否超过限额。

当保存一个订单时，我们遵照给定的价格。我们不想由于实际价格高于承诺价格而丢失客户。这要求我们必须与订单一起保存承诺的价格（图A-5中的OrderPrice），而不是仅仅引用货物（它具有一个价格）。另一种方式是保存货物的历史价格，而这可能会引起很大的混乱。而且，我们不会因为要补充货物而“重新打开”已提交的订单，比方说，在客户与我们交谈后不久发现需要三个物品，而不是二个。在这种情况下，可以做一个没有发货和/或开发票成本的订单<sup>②</sup>。减少订单行的货物数量是另外一种很容易处理的情况（在真实世界中，它可能影响大量订货的折扣价）。

**POrderBuilder**。POrderBuilder类为采购订单的获取和处理提供了支持。它是一个同时在客户和中间层中使用的类，而且它是通过序列化传输的。POrderBuilder在几种情况下与仓库进行交互，例如当调用AddItemToOrder()创建一个订单行的时候。名为ItemStore的抽象类捕获这种交互，ItemStore仅由一个方法构成，即GetItem()。

当调用CheckDiscrepancies()方法时，POrderBuilder创建差异报告。典型的场景是客户使用一个本地版本的ItemStore（更多细节参见下面的“ItemStore在客户、服务器和测试中的不同实现”小节），它在与客户会话期间给出价格和库存。当订单完成时，接单员将它提交给中间层。在某个时刻，中间层在被传递的POrderBuilder上调用CheckDiscrepancies()，以生成报告。此时，ItemStore是一个驻留在靠近数据库的中间层中的对象。

单元测试需要在POrderBuilder上具有CheckDiscrepancies()方法。参见下一个代码清单。在[SetUp]时，创建一个POrderBuilder，即bldr，并用一些代码行来填充它。ItemStore

① 如果由于接单员口误，在价格中多谈了一个零，那么是否有责任？

② 在这个简单示例中，这是很多未明确表述的问题之一。

还创建了一个桩 `itmStore`，并在 `[SetUp]` 期间将其关联到 `bldr`。在第一行代码中，测试模拟了 `ItemStore` 在库存方面的情形是如何变化的（这时就应该生成差异报告了）。参见以下代码。

```
[Test]
public void Stock()
{
    itmStore.Items[0].Stock = 1;

    POrderDiscrepancyReport r = bldr.CheckDiscrepancies();
    Assert.AreEqual(0, r.PriceCount);
    Assert.AreEqual(1, r.StockCount);
    Assert.AreEqual(1, r.GetStockAt(0).NumberOfMissingItems);
}
```

请想象一下在提交订单之前，在订单提交处理的早期是如何创建差异报告的。然而，这里有一个事务处理问题，这是一个经常被忽视或过于简化的问题。如果不仔细，可能检查不出问题，换言之，没有任何差异，但当提交订单时，库存中的最后一件货物已经被“分配”给另一个订单和客户。也就是说，差异检查应该在订单提交的同一个事务中进行，并且应该通过锁定来防止其他订单从仓库中“偷走”正打算要出售的货物。

`ItemStore` 在客户、服务器和测试中的不同实现。更仔细地看一下 `ItemStore`，可能会发现它很有趣。上面讨论了一个本地客户版本和一个中间层版本（即事务版本）。这究竟是怎么回事呢？那么，下面就来看一下。

图 A-13 显示了 `POrderBuilder` 如何保持了一个对 `ItemStore` 的引用。这个引用被实现为一个由应用程序代码分配的公共属性。在客户中，我使用一个与订单提交期间在中间层中所使用的事务实现完全不同的 `ItemStore` 实现（如果在这一点上的叙述过于夸大了，那么我在此致歉）。在方法的数量方面，`ItemStore` 并不是一个必须要实现的接口，但是，它却是 `POrderBuilder` 的核心。例如，图 A-13 中显示的所有方法都使用 `ItemStore`。



图 A-13 `POrderBuilder` 和 `ItemStore` 之间的关系

我已经将 `ItemStore` 的客户版本实现为简单的“从存储和缓存中读取所有货物”。如果货物的数量并不大的话，这种策略很有效。即使出现货物量过大的情况，也没有更好的方法可用，对中间层进行“更智能的”查询。在客户中，当接单员具有“处理订单”的特权时，则实例化一个 `POrderBuilder`，并为其分配一个客户 `ItemStore` 实现的实例。在任何时候，都可以调用 `CheckDiscrepancies()` 来检查客户的 `ItemStore`。

如果客户接受，下一步就是将订单提交给中间层。在传输过程中，`POrderBuilder` 实例被序列化，而不是 `ItemStore` 实例。对 `ItemStore` 的引用被标记为 `[NonSerialized]`。在外层对象



中, 创建一个连接/事务, 并且接受订单的工作被传递给过程对象 (即业务对象), 参见以下代码。

```
public POrderDiscrepancyReport TakeOrder(POrderBuilder order)
{
    POrder bo = new POrder();
    CnTxWrap cn =
        CnTxWrap.BeginTransaction(bo.CreateConnectionForMe());
    try
    {
        POrderDiscrepancyReport res = bo.TakeOrder(cn, order);
        cn.Commit();
        return res;
    }
    catch
    {
        cn.Rollback();
        throw;
    }
}
```

既然已经讨论到这里了, 就让我们来看一下过程对象的TakeOrder()方法, 参见以下代码清单。

```
[PrincipalPermission(SecurityAction.Demand, Role = Roles.User)]
[PrincipalPermission(SecurityAction.Demand, Role =
    @"InternalUnitTesting")]
public POrderDiscrepancyReport TakeOrder(CnTxWrap cn,
    POrderBuilder order)
{
    ItemStore itmStore = new TxItemStore(cn);
    order.ItemStore = itmStore;
    POrderDiscrepancyReport res = order.CheckDiscrepancies();

    POrderPh ord =
        ((POrderBuilder.IOrderInternal)order.POrder).POrder;
    new POrder().InsertNoRBS(cn, ord);

    OrderLine olproc = new OrderLine();
    for(int i=0; i<order.OrderLineCount; i++)
    {
        OrderLinePh ol =
            ((POrderBuilder.IOrderLineInternal)order[i]).OrderLine;
        olproc.InsertNoRBS(cn, ol);
    }

    return res;
}
```

可以看到, ItemStore实际上是TxItemStore的一个实例, cn参数被传递给ItemStore, 该参数包含 (或者包装) 一个数据库连接和一个即将要发生的事务。这个ItemStore实现读取带有UPDLOCK提示 (或者任何你所喜欢的方法) 的存储, 导致存储被锁定, 以防止其他用户在此时更新它。必须承认, 这是某种瓶颈问题, 但它确实提高了正确性。

如果违反了一些重要业务规则，例如当超过全局最大限制时，`CheckDiscrepancies()`将抛出一个异常。如果唯一的差异是价格不符或库存缺货，那么这些将记录到报告中，但订单完全可以完成。

你是否记得前面讲过的同时在客户和服务端上“运行的”执行两次检查的业务规则？本质就是永远不要信任客户。检查是在`CheckDiscrepancies()`中进行的。例如，在客户上对价格做出的任何更改都将作为不一致出现。在这个特殊情形中，客户在我们控制的环境中运行，但不管怎样，都应该保持谨慎。在服务端上执行的代码比在客户上执行的代码安全得多。记住，服务端通常无法知道正在发送信息的调用者是我们的程序，还是另一个程序。

### A.3.4 最后的一些要点

坦白地讲，很多人认为这并不是纯粹的OO。例如，我不并害怕使用在某种程度上比语言本身提供的对象引用“更弱的”对象引用。参见图A-13中的Guid形式的项目引用。但是，如果稍微大胆一点的话，可以思考一下有哪些明显的理由支持使用“指针”引用。将其与代理引用对比，后者提供了延迟实例化的选项。

这里要表达的意思是，对与错都不是绝对的。通常，没有完全对或错的东西（Jimmy肯定也同意这一点）。我们只需完成工作，为客户提供所需的软件，并且知道我们的设计目标是什么（例如，将业务规则集中到一个地方），并实现这些目标。

## A.4 小结

现在总结一下Mats、Frans和Ingo介绍的几种风格的变体。哪种风格是最好的？这还取决于具体情况！生活中充满妥协。读者是否想听一些更好和更有用的建议？不同风格有哪些优点和缺点？这里的建议是：为自己的特殊情况做出自己的选择。

我确信读者已经领会了这里的观点。可以看到，我并未邀请我的朋友写一些他们认为哪种是最佳方法的想法，而是请他们为读者展示对于我们所讨论问题的其他一些观点，以及来自其他风格的一些灵感。

如果让我对这些风格做一下简短总结，我会像下面这样来表达。在Mats的风格中，我认为题目就是一个很好的总结，即“面向对象数据模型、智能服务层和文档”。Mats喜欢把行为放到服务层中，并把领域模型看作是数据的面向对象表示。“文档”是不同层之间的数据交换格式。

Frans强调了数据库功能的重要性，数据库中的表示就是它的重要功能之一。

Ingo讨论了寻找正确抽象的重要性，而且强调了我不应该期望对所有情形都使用一种标准风格。此外还讨论了大量的实用话题。

## 已讨论的模式目录

**本**附录列出了本书讨论过的模式目录。这里没有新的信息，而只是为读者提供一个服务。读者可以在这里找到各种模式的一个非常简练的描述，而且大部分模式还提供了一个URL链接，以供读者查阅更多信息。

当描述用引号引起来时，表示此描述直接摘自所提供的URL（对于一些[GoF Design Patterns]中的模式，引号表示其中的内容来自所提及的图书的内封）。

### 抽象工厂[GoF Design Patterns]

提供接口，用于在无需指定这些接口的具体类的情况下创建依赖对象家族。

### 聚合[Evans DDD]

具有边界的集群实体和值对象。令聚合中的某个实体作为访问点（聚合根）。

### 限界上下文[Evans DDD]

定义不同模型的边界。

### 责任链[GoF Design Patterns]

通过为多个对象提供处理请求的机会来避免把请求的发送者耦合到其接收者。将接收对象连成链，并沿着此链传递请求，直到有一个对象处理它。

### 类表继承[Fowler PoEAA]

“通过为每个类使用一个表的形式来表示类的继承层次结构。”

<http://www.martinfowler.com/eaCatalog/classTableInheritance.html>

### 粗粒度锁[Fowler PoEAA]

“用一个锁来锁定一组相关对象。”

<http://www.martinfowler.com/eaCatalog/coarseGrainedLock.html>

### 收集参数模式[Beck SBPP]

一个要传递给方法的对象，用于从方法收集信息。

### 具体表继承[Fowler PoEAA]

“在层次结构中通过为每个具体类使用一个表的形式来表示类的继承层次结构。”

<http://www.martinfowler.com/eaCatalog/concreteTableInheritance.html>

### 数据映射工具[Fowler PoEAA]

“一个用于在对象和数据库之间移动数据的映射工具层，同时保持它们互相独立，而且与映射工具本身独立。”

<http://www.martinfowler.com/eaCatalog/dataMapper.html>

### 数据传输对象[Fowler PoEAA]

“在进程之间携带数据的对象，目的是减少方法的调用次数。”

<http://www.martinfowler.com/eaCatalog/dataTransferObject.html>

### 装饰方法[GoF Design Patterns]

动态地将额外责任施加给某个对象。装饰方法提供了一种灵活的方法，可用于替代为了扩展功能而建立子类的方法。

### 依赖注入

不使用令实例查找其自己的依赖性的方法，而是将依赖性注入到实例中。

### 领域模型[Fowler PoEAA]

“领域的对象模型，它将行为和代码整合到一起。”

<http://www.martinfowler.com/eaCatalog/domainModel.html>

### 嵌入值[Fowler PoEAA]

“将一个对象映射到另一个对象的表的几个字段中。”

<http://www.martinfowler.com/eaCatalog/embeddedValue.html>

### 实体[Evans DDD]

从根本上讲，很多对象不是通过它们的属性定义的，而是通过连续性和标识的线索定义的。

### 工厂[Evans DDD]

当对象或完整聚合的创建变得很复杂或揭示了过多的内部结构时，工厂提供了封装。

### 工厂方法[GoF Design Patterns]

定义一个用于创建对象的接口，但让子类来决定实例化哪个类。工厂方法使得类的实例化听从子类。

## 外键映射[Fowler PoEAA]

“将对象之间的关联映射为表之间的外键引用。”

<http://www.martinfowler.com/eaCatalog/foreignKeyMapping.html>

## Generation Gap [Vlissides Pattern Hatching]

“只对生成的代码进行一次修改或扩展，而不管它被生成多少次。”

<http://www.research.ibm.com/designpatterns/pubs/gg.html>

## 标识字段[Fowler PoEAA]

“在对象中保存一个数据库ID，以便维护一个内存对象与数据库行之间的标识。”

<http://www.martinfowler.com/eaCatalog/identityField.html>

## 标识映射[Fowler PoEAA]

“通过在一个映射中保持每个被加载的对象来确保每个对象只加载一次。当引用对象时，使用映射来查找它们。”

<http://www.martinfowler.com/eaCatalog/identityMap.html>

## 隐式锁[Fowler PoEAA]

“允许框架或层超类型代码获取离线锁。”

<http://www.martinfowler.com/eaCatalog/implicitLock.html>

## 层超类型 [Fowler PoEAA]

“充当其层中所有类型的超类型。”

<http://www.martinfowler.com/eaCatalog/layerSupertype.html>

## 层[POSA]

（也在[Fowler PoEAA]中。）

“一些结构的应用，可以被分解为几组子任务，其中每组子任务都处于某一特殊的抽象级别。”

## 延迟加载[Fowler PoEAA]

“一个对象不会包含所有你需要的数据，但是知道如何获取数据”。

<http://www.martinfowler.com/eaCatalog/lazyLoad.html>

## 元数据映射[Fowler PoEAA]

“在元数据中保持对象-关系映射的细节。”

<http://www.martinfowler.com/eaCatalog/metadataMapping.html>

### 模型视图控制器[Fowler PoEAA]

“将用户界面交互分解为三个不同的角色。”

<http://www.martinfowler.com/eaCatalog/modelViewController.html>

### 模型视图表示器[Fowler PoEAA2]

“将表示行为与视图分离开，同时允许视图接收用户事件。”

<http://www.martinfowler.com/eaDev/ModelViewPresenter.html>

### 通知[Fowler PoEAA2]

“一个对象，它收集领域层中的错误信息及其他信息，并将这些信息传递给表示层。”

<http://www.martinfowler.com/eaDev/Notification.html>

### Null 对象[Woolf Null Object]

提供默认数据和行为的对象（如果没有此对象，就会产生无效引用）。

### 乐观离线锁[Fowler PoEAA]

“通过检测冲突并回滚事务来防止并发业务事务之间的冲突。”

<http://www.martinfowler.com/eaCatalog/optimisticOfflineLock.html>

### Party Archetype [Arlow/Neustadt Archetype Patterns]

一种用于表示人员和组织信息的方式。

### 悲观离线锁[Fowler PoEAA]

“通过一次只允许一个业务事务访问数据的方法来防止并发业务事务之间的冲突。”

<http://www.martinfowler.com/eaCatalog/pessimisticOfflineLock.html>

### 管道和过滤器[POSA]

通过管道来传递数据，并在过滤器中处理数据流。

### 表示模型[Fowler PoEAA2]

“对表示层的状态和行为进行表示，独立于界面中所使用的GUI控件。”

<http://www.martinfowler.com/eaDev/PresentationModel.html>

### 代理[GoF Design Patterns]

为另一个对象提供一个代理或占位符，以控制对它的访问。

### 查询对象[Fowler PoEAA]

“表示数据库查询的对象。”



<http://www.martinfowler.com/eaCatalog/queryObject.html>

## Recordset [Fowler PoEAA]

“表格数据的内存中表示。”

<http://www.martinfowler.com/eaCatalog/recordSet.html>

## 反射[POSA]

在无需事先知道类型的情况下，用编程方式通过元数据来检查实例的类型并与实例进行交互。

## 注册[Fowler PoEAA]

“一种众所周知的对象，其他对象可以用它来查找公共的对象和服务。”

<http://www.martinfowler.com/eaCatalog/registry.html>

## 远程外观[Fowler PoEAA]

“在细粒度的对象上提供一个粗粒度的外观层，以提高网络传输的效率。”

<http://www.martinfowler.com/eaCatalog/remoteFacade.html>

## 存储库[Evans DDD]

（也在[Fowler PoEAA]中。）

用于在生命周期中间定位某个特定实体（或一组实体）的对象。

## 独立表示[Fowler PoEAA2]

“确保任何负责操纵表示的代码都只操纵表示，将所有领域和数据源逻辑放到明确独立的程序领域中。”

<http://www.martinfowler.com/eaDev/SeparatedPresentation.html>

## 服务层[Fowler PoEAA]

“用一个服务层来定义应用程序的边界，该服务层建立了一组可用的操作，并协调应用程序在每个操作中的响应。”

<http://www.martinfowler.com/eaCatalog/serviceLayer.html>

## 服务定位器[Alur/Crupi/Malks Core J2EE Patterns]

“使用服务定位器来实现和封装服务及组件查找。服务器定位器隐藏查找机制的实现细节，并封装相关的依赖项。”

<http://www.corej2eepatterns.com/Patterns2ndEd/ServiceLocator.htm>

## 服务[Evans DDD]

可以将那些不自然属于实体或值对象的责任提取到服务中。

## 单表继承[Fowler PoEAA]

“将类的继承层次结构表示为单一表，表中包含对应于各个类的所有字段的列。”

<http://www.martinfowler.com/eaCatalog/singleTableInheritance.html>

## 单体[GoF Design Patterns]

单体确保一个类只有一个实例，并为它提供了一个全局访问点。

## 规格[Evans DDD]

（也在[Fowler Analysis Patterns]中。）

通过描述概念的名称来捕获和公开断言。

## 状态[GoF Design Patterns]

当对象的内部状态发生更改时，允许对象更改其行为。对象将表现为更改它的类。

## 表模块[Fowler PoEAA]

“处理数据库表的所有行或视图的业务逻辑的单一实例。”

<http://www.martinfowler.com/eaCatalog/tableModule.html>

## 模板方法[GoF Design Patterns]

定义某个操作中的算法框架，将一些步骤推迟到子类中。模板方法允许子类在不更改算法结构的情况下重新定义算法的特定步骤。

## 事务脚本[Fowler PoEAA]

“按过程来组织业务逻辑，其中每个过程处理表示层的一个请求。”

<http://www.martinfowler.com/eaCatalog/transactionScript.html>

## 操作单元[Fowler PoEAA]

“维护一个对象列表（这些对象受到某个业务事务的影响），并协调更改的写出以及并发问题的解决。”

<http://www.martinfowler.com/eaCatalog/unitOfWork.html>

## 值对象[Evans DDD]

（也在[Fowler PoEAA]中。）

很多对象没有概念上的标识。这些对象描述了某个事务的一些特征。





“本书向读者展示了如何将测试驱动设计、对象-关系映射和领域驱动设计等方法应用于.NET项目……书中介绍的技术在很多开发人员看来是未来软件开发的关键……随着技术越来越强大，复杂度越来越高，理解如何更好地使用技术也变得越来越重要。本书在推进这种理解方面迈出了可贵的一步。”

——Martin Fowler, ThoughtWorks公司首席科学家,《重构》与《企业应用架构模式》作者

“学习领域驱动设计的最好方法是坐在一位友好、耐心且经验丰富的从业者身边，一步一步地共同研究问题。阅读本书正是这种体验。”

——Eric Evans, 领域驱动设计创始人

## Applying Domain-Driven Design and Patterns

# 领域驱动设计与模式实战



模式、领域驱动设计和测试驱动开发赋予架构师和开发人员前所未有的能力，使他们能够创建功能强大、健壮且可维护的系统。但是，如何在实际项目中充分发挥这些利器的潜力呢？

本书中，作者将Martin Fowler《企业应用架构模式》和Eric Evans《领域驱动设计》两部经典名著中的思想精髓以及重构、测试驱动开发等技术融会贯通，并通过大量C#实例加以阐释，跨越了领域模型、数据库与UI层之间的障碍，真实展示了创建高质量的企业级应用架构的全过程。

本书就像是精彩纷呈的旅行见闻，每一处的所思所想都闪耀着智慧的光芒，生动诠释了作者对面向对象开发中各种设计选择的深刻理解。



**Jimmy Nilsson** 资深软件架构师，有超过20年从业经验，2008年在瑞典主要IT媒体评选的全国软件架构师和开发人员排行榜上名列第二。目前担任factor10咨询公司CEO，客户包括爱立信、微软、沃尔沃等。本书是他的代表作，已被翻译为日、俄等多种文字，他的另一部著作.NET Enterprise Design with Visual Basic .NET and SQL Server 2000也获得Amazon 4星半评价。他的博客是<http://JimmyNilsson.com/blog/>。



[www.informit.com](http://www.informit.com)

本书相关信息请访问：图灵网站 <http://www.turingbook.com>

读者/作者热线：(010)51095186

反馈/投稿/推荐信箱：[contact@turingbook.com](mailto:contact@turingbook.com)



ISBN 978-7-115-21277-1



**分类建议** 计算机/程序设计

ISBN 978-7-115-21277-1

定价：69.00元

人民邮电出版社网址：[www.ptpress.com.cn](http://www.ptpress.com.cn)