

**java学习群：524621833**  
**欢迎小白or大神入群交流**



# 对《J2EE核心模式》的好评

“《J2EE核心模式》的作者们提取了一组真正实用的模式。他们介绍了应该如何应用这些模式、如何重构系统以便从模式中获益。拥有这本书就像有一个专家组坐在你旁边一样。”

——Grady Booch, Rational软件公司首席科学家, 序言节选

“本书是对J2EE模式的一次出色的汇集。它固化了J2EE开发的重要经验。没有这本书, 就别开发EJB……而且, 越来越多的人选用“重构”的方法, 来对现有系统做出更动。本书的作者把我在重构方面的研究应用在一个新方向——也就是J2EE设计的世界之中, 这样做还尚属首次。我不仅因为有人在我的研究基础上进行工作而心怀感激, 而且, 读到他们依据自己的经验, 实际描述了如何进行系统重构的转换, 我也感到非常喜悦。”

——Martin Fowler, ThoughtWorks 首席科学家, 序言节选

“《J2EE核心模式》是一本福音书, 应该与每个J2EE应用服务器一起配套发售。这本书提供了一套明确无误的、经过实战检验的模式语言以及多种重构策略, 对于在真实环境下设计、实现、维护健壮的J2EE应用系统很有帮助。本书的作者都是J2EE架构设计战壕中的老兵, 本书总结了他们的多年经验, 囊括了J2EE平台下可供架构师使用的多种技术和API, 令人信服地解答了为何、何时、如何使用J2EE平台。”

——Sean Neville, Macromedia公司JRun企业版架构师

“作者们介绍了大量对于应用程序架构极有帮助的模式, 这是一项了不起的工作。单单是书中的‘重构’部分就值整本书的价钱! ”

——Craig McClanahan, Struts首席架构师

“随着J2EE平台渐趋成熟, 在应用系统设计的过程中采用成熟的工程原则, 对目前来说比以前更为重要。有很多架构师、分析师都要决定使用何种策略在J2EE平台上实现企业应用, 对于他们, 《J2EE核心模式》是一部价值连城的技术指南。书中详细分析了用于各种最佳实践的通用模式, 同时也讲解了避免不佳实践的方法。”

——Craig Russell, JDO架构师

# 译 者 序

为一部由Grady Booch和Martin Fowler作序的作品写序言，这个念头本身就足够荒谬和僭越，不啻于在莎翁之后再写一个关于丹麦王子复仇的剧本。大师们的判断是中肯而毫不含糊的：“没有这本书，就别开发EJB。”他们的担保足以确认本书在其论域中舍我其谁的地位。是的，这就是“那本”J2EE书。

当然，对于广大中国开发者而言，我们早就已经在“没有这本书”的条件下开发了大量J2EE乃至EJB应用系统。那些波折的、不乏磨难的开发历程似乎使不少人具备了一种不无理由的自信，在掌握了若干API细节、若干应用服务器配置诀窍、若干框架类库用法之后，他们或是公开、或是暗自地把自己当成了当之无愧的Java企业开发专家。——不，这些话没有任何揶揄的意思：我们想说的其实是，本书恰恰是为以上这一类开发者写的。对于他们想成为“Java企业开发专家”的隐秘欲望，本书就是最大限度的补救和成全。如果说，此前的各种教程都是在介绍J2EE开发中的“内容”要素——也就是，教给我们“做什么”的话，本书关注的则是这里的“形式”要素，即“怎样做”才能开发出高效的、优雅的J2EE系统。读者从中学到的，将不仅仅是“J2EE技术”，而是“如何使用J2EE技术进行设计”。

换句话说，如果你以前没有进行过J2EE实践，但明早将应聘一个需要“1年J2EE开发经验”的职位，本书中不包含你今晚要彻夜吞咽的那一类知识；相反，如果你，这位未来的“Java企业开发专家”，追求的职位是“资深Java应用系统架构师”，如果你预料到未来的上司明天将问起“怎样实现访问控制”、“何时采用细粒度的接口设计”等“高阶”问题，那么恭喜你，今晚——乃至今后——阅读本书，你选对了补课的读物。

作为本书第1版的忠实读者，我们（半是欣喜、半是惊讶地）发现，眼前的这部第2版构成了全新的阅读体验。作者们按照最新版J2EE技术规范（尤其是EJB 2.1）全面修订了技术细节；根据模式社区的研究交流，作者们补入了若干模式；即使是一些不涉及技术更新的部分，论述方式、示例也完全不同于第1版；原有的PSA项目（第1版“尾声”一章）融入了其余各章的“示例代码”部分；而新增的讨论“微架构”的尾声、对Web Service等技术的关注、对各种的持久化方案（定制持久化、EJB、JDO等）的深入讨论，都体现出作者们对本书新版的大量投入。

受益于本书有年，在此，我们想冒昧地为本书的中国读者们建议一条高效的阅读路径：与第1章相比，第5章“J2EE模式概览”是读者更合理的起步点。请特别关注其中对“分层”、“术语”和模式/策略区别的讨论，这些都是贯穿全书的重要概念！其次，应该通读第2章“表现层设计考虑和不佳实践”和第3章“业务层设计考虑和不佳实践”：即使你不打算使用任何模式，甚至，即使你根本不关心J2EE开发，只要你的工作与分布式企业应用系统有关，这两章涉及的问题都是你迟早会遇到的。至于每个具体模式本身，我们则推荐读者留意其中详尽的“策略”部分和那些散布其中的“设计手记”。前者讨论了对同一个模式的多种实现方案，后者则突出介绍了特定开发领域的一些核心概念和考虑。

一部英文技术论著在汉语中的旅行，永远是一段难以捉摸的行程。对于本书的汉语译者，“技术难度”并非挑战：全书讨论的正是译者们最为熟知的一个领域，所以我们能够负责任地说，在这个中译本里，没有任何技术细节会因为译者的无知或生疏而发生变形或曲解。这次翻译的原则和前提是对我原文的彻底领会。事实上，译者在翻译工作中遇到的困难主要发生在“语汇”层面。简单地说，J2EE专著的译者总要面对“翻，还是不翻”的两难处境：对象、函数的名称，UML图中的各种元素，这些内容由英语表示早就是约定俗成，即使是英语程度略低的开发者大概也都能读懂，所以，在读者能够理解的部分尽可能保留原文似乎是一种合理的选择——毕竟开发工作最终是与代码有关，而代码则肯定是要用“英文”的。但在另一方面，翻译的责任就在于让不谙英文的读者也能通达作品，如果译文中大量段落（不包括示例代码）都仍保留为英文或“类英文”，那么读者也就无法直观地获得原文包含的信息。反复权衡之后，在这个译本中译者的解决方式还是折衷的。工作中我们采取了以下原则：

1) 术语尽可能采用通用文献定译，不自创译法。对于各个模式的名称、模式文档模板各部分名称、重构手法名称，我们参考了李英军等译《设计模式》（机械工业出版社，2000年）、熊节等译《重构》（中国电力出版社，2003年）等译作，以及IBM DeveloperWorks中文网站的部分资源。

2) 本领域的一些常见术语，如果没有定译，本书也不自创新语，强译为中文，而是保留英文原字。这一类的术语包括：applet、servlet、bean、JavaBean、entity bean、session bean、EJB、finder、Context、cookie、RowSet、null、scriptlet、Web Service。根据我们的观察，国内的开发者在日常工作中已经习惯按原文使用以上术语。在一些情况下，我们也以注释形式澄清了这些术语的用法。另外，一些非常直观的英文表达方式，比如“versus/vs”（“A versus B”即“A对B”、“A与B相比较/对照”），我们也径用原文——改为汉语既罗嗦，也不直观。

3) 模式中的对象名称，往往按照代码风格命名，比如“BusinessObject”、“CustomerTO”等。如果对此完全不加翻译，那么很多充斥这类表达的段落就很难理解。我们的原则是，在每个自然段第一次出现某个这类表达方式时，用括号注明，比如“BusinessObject（业务对象）”、“CustomerTO（客户传输对象）”等。希望这个做法能够维持易懂和简洁之间的平衡。

4) 书中示例代码占有相当大的比重，而代码注释则是理解这些代码的关键。我们把所有代码注释译为中文。而对在视图中显示特定结果的代码（比如调试信息等），我们没有改为中文，只是在必要时对输出信息的含义加以注解。如果读者更信赖代码原貌，还可以从本书官方网站<http://www.corej2eepatterns.com/> 下载原始代码。

5) 原书不包含注解，目前的所有注解都是译注。

6) 原书申义未畅处，译文中以方括号[]加以解释、补足，略去生涩。这与上面三条原则一样，都类似于在原作讲话时的插嘴——但翻译任务本身，似乎本就已经是一种“插嘴”了。在博学的读者看来，有时候译者或许还不如保持体面的沉默——但我们只能力图做到插嘴而不多嘴。

7) 原书引用了Apache项目的若干代码，所以附录中包含Apache软件授权协议一页。中译本照录了这份法律文件，未加翻译。

8) 几个关键术语的译名考虑：

- application：一般译为“应用程序”或“应用”。本书中这个词单独出现时，往往指的是

“企业应用”，亦即企业信息应用系统。考虑到“应用程序”容易被理解为“桌面程序（desktop application）”，在该词含有“企业应用”意味时，我们译为“应用系统”，其他情况下则译为“应用”，以示区别。

- client：译为“客户端”。但本书中所说的“客户端”常常是指特定组件的调用者，不一定是“桌面程序客户端”，反倒很可能本身也是另一种组件、甚至一个子系统。希望读者注意该词在书中的用法。
- POJO：软件方法论大师Martin Fowler在《Patterns of Enterprise Application Architecture》（PEAA）中创造的说法，是plain old Java object的缩写，指普通Java对象（而不是EJB等组件）。中译本仍采用“POJO”名称。
- enterprise bean：直译为“企业bean”，在本书中就是“enterprise JavaBean/EJB”的另一说法。为了直观，我们统一译为“EJB”。
- tier/layer：字面上都是“层” / “层次”。本书中“tier”指的往往是“架构”意义上的分层，比如“表现层”、“业务层”、“集成层”等，而“layer”既分享了前者的含义，有时也指tier内部的中间层次，比如“会话门面”就构成了客户端和业务服务之间的一个“layer”。这两种意思实在很难区分，中译本只能都译为“层”、“层次”。希望读者在阅读中体察这种细微差别。
- delegate：是设计模式中的重要概念。一般译为“委派”。但在我们看来，这个译法还不完整，因为“委派”在汉语中只是动词，而delegate往往还充当名词。这次中译本的做法是，动词delegate仍译为“委派”，比如“A把功能F委派给业务层的B”，而名词delegate则译为“代表”，比如“B是A在业务层的代表”。希望读者体察，并推荐更好的译法。

原书中所有模式、重构手法、策略的名称以斜体标出，要点以黑体标出。中译本一仍其旧。

原书经多人、多版修订完成，难免有错漏、乱排之处。译者根据本书官方网站的最新勘误表订正，并结合参照本书第1版《Core J2EE Patterns: Best Practices and Design Strategies》（Addison Wesley，2001），另外修正了数十处错误。

刘天北 熊节

# Grady Booch序

在软件世界中，每个开发机构就像是一个部落，而一个模式就是对部落的某种共同记忆的一种有形表现。模式是对共通问题的共通解决方案，因此，对于某个特定开发机构的文化氛围、某种特定的问题领域来说，命名、确定一个模式，也就是把已经在先前的经验中获得证明的共通解决方案进一步整理成文，设为圭臬。如果你有一套不错的模式语言能随时派上用场，那就像是在开发过程中身旁一直坐着一个专家组：在采用专家们提出的一个模式时，你也就实际获益于他们那些来之不易的经验知识。事实上，那些最好的模式大都不是凭空发明的，而往往是专家们从现有的成功系统中发现、提取而来的。所以，一个成熟的模式中充满了实际有效的内容，不存在空泛不实的内容，同时，它也体现了设计者的智慧和设计思路。

那些深刻的、真正有用的模式大多都是很古老的东西，见到这么一个模式的时候，你往往会说：“嘿，我从前就这么做过。”但是，只有当专家为这个模式命名之后，你才获得了一整套讨论这个问题的语汇；此前，由于缺乏这种语汇，你往往想不到怎样使用这个模式，因此命名有助于我们更好地应用模式。最终，这样一个模式的功效在于，它能让你的系统变得更为简单。

而模式还不仅有助于构建更简单而又实际有效的系统，它们还有助于构建优美的系统。在一种极度缺乏时间的文化氛围中，编写优美的软件常常是不可能的，这是很可悲的事情。因为，我们作为专业人士，本来都应该致力于构建高质量的产品。而通过应用一组恰当的模式，你就有可能为自己的系统注入某种程度的优雅——缺乏模式的帮助，这种优雅往往就无从谈起。

《J2EE核心模式》的作者们提取了一组真正实用的模式。别误解我的意思：J2EE当然是一种重要的软件平台，它能够让开发团队构建出特别强大的软件系统。但是，目前的现实却是，在J2EE提供的抽象层次、服务与开发团队必须构建的最终应用之间，还存在着非常巨大的语义断裂。本书描绘的这些模式，事实上正是人们一次又一次用来填补这种断裂的共同解决方案。应用这些模式，也就是采用了一条最主要的避免软件风险的措施：只要编写更少代码就能获得相同的效果。你也不必再重新自己动手寻找解决方案，这些模式已经在很多现存系统的应用中得到了验证，所以只需应用它们就是了。

作者们不仅完成了一组模式的命名，还利用UML确定了模式的语义，使它们更容易为人理解。并且，他们也介绍了应该如何应用这些模式、如何重构你的系统以便从模式中获益。再说一遍，拥有本书就像一个专家组坐在你旁边一样。

Grady Booch  
Rational软件公司首席科学家

# Martin Fowler序

在1998年末，我所在的ThoughtWorks公司就开始使用J2EE了。在那个时候，我们发现了很多很酷（虽然有点儿不成熟）的技术，但是很少有人能说明怎样才能恰当地应用这些技术。也许是因为我们具备在其他OO服务器环境下编程的大量经验，所以我们自己能够应付这些问题。但是我们也见到很多客户费尽了周折——并不是因为技术本身的问题，而是因为不知道怎样才能恰当地应用这些技术。

使用模式，能够固化设计经验——也就是说，模式有助于将经常重现的问题的实际解决方案进行归类、编目。多年以来，对于模式的这种用途，我一直都是个由衷的爱好者。最近的几年里，业界的很多先驱者都在使用J2EE，都在寻找构成一个有效的J2EE解决方案的核心模式。本书对这些模式进行了出色的汇集，其中揭示的很多技术都是我们自己通过多次尝试、多次错误才获得的。

这也就是本书的重要之处。对API倒背如流是一回事；知道如何设计优秀的软件则是另一回事。本书确实致力于固化这一类设计知识，这也是我见到的第一本这么做的书，看到作者们做得这么漂亮，我由衷感到欣慰。如果你在J2EE平台下工作，你也就需要了解这些模式。

另外，本书也提出了这样一种观点：当编码开始后，设计并非就结束了。人们在设计时做出的一些决定常常不符合实情。在这种情况下，在构建过程中还需要修正原本的设计，而这种修正必须以一种规范的形式进行。越来越多的人选用“重构”的方法来对现有系统做出更动。本书的作者把我在重构方面的研究应用在一个新领域——也就是J2EE设计的世界之中，这样做还尚属首次。我不仅因为有人在我的研究基础上进行工作而心怀感激，而且，读到他们依据自己的经验，实际描述了如何进行系统重构的转换，我也感到非常喜悦。

说到底，这种经验正是最宝贵的东西。把设计经验固化在书本中，这是最难做的一件事，但是要想让我们的行业进一步发展，这件事又非做不可。本书固化了J2EE开发中的重要经验。没有这本书，就别开发EJB。

Martin Fowler  
ThoughtWorks 首席科学家

# 前　　言

自从本书第1版出版以来，关于最初那15个模式，我们收到了大量反馈意见。最近几年来，J2EE模式社区目录服务器（JPCLS）上的活动一直都非常活跃、非常成功，每天都有很多精彩的意见交流。在这段时间里，我们也和客户一起进行了不少重要的大型J2EE架构设计、开发项目。把这段时期的经验和反馈植入到原有模式的更新工作和新模式的归档工作中，也确实是一个费力而艰苦的过程。我们特别关注了反馈中提及最多的内容：对J2EE技术规范和Web Service最新版本的支持。

我们完全修订、更新了最初的15个模式，使得本书覆盖了J2EE技术1.4版的规范。我们在这些最初的模式中加入了很多新的策略。另外，我们还记录了6种新模式，以便改进模式语言，为构建、理解、使用J2EE框架提供更好的概念抽象。虽然这些模式中的每一个本身都极为实用，但我们还进一步相信，当开发者将其组合起来解决大型问题时，它们更能显出威力。因此在本书的新版中，引入了一个我们正在探究的、与此相关的全新领域，我们称此为“微架构”。

所谓“微架构”，就是搭建应用程序和系统的积木块。与列入目录的那些单独模式相比，这个概念是一种更高层面的抽象，它常常表现为一组相互关联的模式组合，用于解决在应用架构中经常重现的一些共通问题。

我们乐于把“微架构”当作一种由相互关联的模式组成的网络，由此形成一种现成的解决方案，用于解决一个粒度更大的问题，比如子系统的设计。

本版中包括了一个叫Web Worker的微架构。它所解决的问题是：一个J2EE应用怎样与一个工作流系统集成。它特别讨论了使用系统集成模式让工作流系统中的用户与J2EE应用进行交互的问题。

本书讲述的是Java 2企业版平台（J2EE）的模式。本书新版中记录的J2EE模式，能够用于解决在J2EE平台上进行软件应用开发的设计者常常遇到的那些问题。在这个模式目录中记录的模式都是在设计实战中发现的，正是因为使用了它们，我们才能为自己的客户创建出了成功的J2EE应用。

本书描述了很多在J2EE平台下证明可行的解决方案，重点强调了以下核心J2EE技术：JavaServer Pages (JSP)、servlet、Enterprise JavaBeans (EJB) 组件、Java Message Service (JMS, Java消息服务)、JDBC以及 Java Naming and Directory Interface (JNDI, Java命名与目录接口)。对于那些在J2EE平台下经常重现的问题，我们通过J2EE模式目录和J2EE重构给出了解决方案。在开发新系统或是改进现有系统的设计时，你可以应用这些想法。本书记录的这些模式能够有助于你迅速熟练地掌握J2EE技术，从而构建出健壮、高效的企业应用。

今天，正如以往一样，我们中间有很多人天真地以为，学会了某种技术，也就等于是学会了用这种技术进行设计。诚然，对于利用某一技术进行设计来说，懂得这种技术是成功的重要元素之一。但现在有很多Java图书，对技术细节（比如API的一些专门用法等等）做出了出色的

讲解，但对如何应用这种技术却未作深入考察。要想学会设计，就需要实际设计经验，需要和其他开发者一起分享关于最佳实践和不佳实践的知识。

本书中传达的经验来自我们的工作实战。我们属于Sun公司的Sun Java中心（SJC）咨询机构。在工作当中，我们经常遇到一些情况，因为技术发展过于迅速，设计者和开发者都仍然在奋力理解技术本身，而无暇理解如何使用该项技术进行设计。

因此，简单地告诉设计者和开发者怎样写出优秀代码，或是建议他们使用servlet和JSP开发表现层，用EJB组件开发业务层，这都是不够的。

那么，在这样的情况下，一个热心的J2EE架构师又怎样才能不单单是学到“做什么”、还能学到“不做什么”呢？哪些实践构成了最佳实践？哪些是不佳实践？怎样完成从问题到设计，再到实现的整个过程？

## Sun Java中心与J2EE模式目录

从初创时期以来，Sun Java中心的架构师们就在与来自全球的客户一起合作，致力于成功地设计、规划、构建、部署各种不同类型的基于Java和J2EE的系统。Sun Java中心是一个快速成长的咨询机构，一直在招募新员工，加入它经验丰富的架构师队伍。

目前已经有大量已验证有效的设计和构架，将这些设计经验固化下来并和其他人一起分享，是我们行业的一项重要需要。我们很早就认识到了这种需要，从1999年就开始以模式的形式记录我们在J2EE平台下的工作经验。虽然我们翻阅了各种现有文献，却没能发现有哪个模式目录是专门记载J2EE平台下的模式的。有很多书论及J2EE技术中的一种或多种，出色地介绍了技术，剖析了技术规范中的微妙细节。我们发现其中有些书还提供了一些设计上的考虑思路，因此也特别有益。

在2000年6月的JavaOne大会上，我们第一次公开发表了我们关于J2EE模式的想法。从那以来，我们收到了来自架构师和开发者的大量热忱反馈。其中一些人表示特别乐意进一步学习模式，还有一些人则说，他们使用过这些模式，只不过没有加以命名、也没有记录下来罢了。人们体现出来的对J2EE模式的兴趣鼓励我们进行进一步的工作。

因此，我们整理出了J2EE模式目录，在2001年3月，这个目录的beta版通过Java开发者联盟（JDC）首次公布给了J2EE社区。基于整个社区的大量反馈，那一份beta版的文稿最终发展成了你现在见到的这本书。

我们希望这些在J2EE平台下的模式、最佳实践、策略、不佳实践和重构能让大家从中受益。

## 本书的讨论范围

本书讨论的内容包括：

- 在J2EE平台下使用模式。

基于我们在J2EE平台的经验，我们编纂了本书中的模式目录。这一份J2EE模式目录描述了在J2EE平台下架构和设计应用的最佳实践。本书着重考察了以下J2EE技术：servlet、JSP、EJB组件和JMS。

- 通过最佳实践来设计应用了servlet、JSP、EJB组件和JMS技术的应用系统。

仅仅学会了技术本身和API还不够，同样重要的是要学会怎样使用技术进行设计。我们记录了在我们的经验中应用这些技术的最佳实践。

- 防止在J2EE平台的设计和架构中“重新发明轮子”。

模式鼓励设计的重用。重用现成的解决方案，能够缩短设计开发应用程序的周期——这也当然包括J2EE应用。

- 鉴别出现存系统中的不佳实践，并利用J2EE模式重构这些设计，以形成更好的解决方案。

知道哪些做法有效，这是一件好事。但知道哪些做法无效也同样重要。我们在本书中记录了自己在设计J2EE应用时遇到的一些不佳实践。

## 本书不讨论的内容

本书没有讨论以下内容：

- 如何使用Java或J2EE技术编程

本书讨论的不是编程。虽然很多内容都基于J2EE技术，但我们没有描述API细节。如果你希望学习Java编程，或是学习使用J2EE中的任何一种技术，现有很多种出色的著作，还有不少在线资源，都可以作为教程。如果你想要学习某一门特定的技术，我们强烈推荐Java官方主页<http://java.sun.com>上的各种在线教程。J2EE技术的官方技术规范也可通过Java主页获得。

- 采用哪种开发过程和方法论

我们并不特别推荐任何一种开发过程或方法论，因为本书讨论的内容与这两方面都关系不大。所以，本书不会教授任何可以用于开发项目的过程或方法论。如果你想要学习过程和方法论的话，现已有很多论著讨论各种面向对象的方法论，对于那些轻量级的过程，比如极限编程，也有不少新书论及。

- 怎样使用统一建模语言（UML）

本书不会教你如何使用UML。我们大量地使用了UML（特别是类图和序列图）来记录模式，描述静态和动态交互关系。如果要学习UML，请参考Grady Booch、Ivar Jacobson 和James Rumbaugh的著作《UML用户指南》[Booch]以及《UML参考手册》[Rumbaugh]。

## 谁应该读这本书

本书写给所有热心关注J2EE的人，程序员，架构师，开发者以及技术经理。简单地说，就是任何对在J2EE平台下设计、架构、开发应用程序有点儿兴趣的人。

我们力图让这本书成为一部写给J2EE架构师和设计者的培训指南。我们认为良好的设计、架构得当的项目具有很高的重要性，所以我们需要优秀的架构师达到这个水准。

对于那些开发者水准参差不齐的开发团队，如果我们把模式、最佳实践和不佳实践都做出详尽的归档，以此在团队中实现知识与经验的共享和传播，这可能会起到难以估价的帮助作用；我们也希望本书能部分地满足类似需求。

## 本书的组织

本书的组织分为两部分。

### 第一部分

第一部分“模式和J2EE”是一个关于J2EE和模式的导论。它考察了开发JSP、servlet和EJB时的设计考虑。这一部分也包括了J2EE平台下的不佳实践和重构。

第1章“导论”简要地讨论了多个问题，包括模式、J2EE平台、模式的定义以及模式的归类。最后引入了J2EE模式目录。

第2章“表现层设计考虑和不佳实践”、第3章“业务层设计考虑和不佳实践”分别讨论了表现层以及业务/集成层的设计考虑和不佳实践。这里所说的设计考虑，是指在J2EE平台下工作时，一个J2EE开发者/设计者/架构师需要考虑的问题。在阅读这两章中的论题时，可以参照其他的多种资源（比如官方技术规范以及一些出色的相关论著）来获得相关问题的一些细节信息。

第4章“J2EE重构”考察了一些重构，我们在自己的实际工作中遇到了这些重构，它们也确实帮助我们把原本不够理想的设计提升为更好的方案。这些重构也提供了看待本书其他内容的另一种思路，我们认为这对于模式目录是一种有价值的补充材料。本章体现出Martin Fowler和他的著作《重构》[Fowler]对我们的影响。对于熟悉《重构》一书的读者，本章的形式也应该相当眼熟。但是，这一章的内容完全基于J2EE技术，而Martin Fowler在他的论著中则是在另一个层面考察重构的。

### 第二部分

第二部分“J2EE模式目录”列出了J2EE模式目录。目录中包含的模式构成了本书的核心内容。

第5章“J2EE模式概览”，是J2EE模式目录的一个综述。这一章一开始对模式的理念进行了高层次的讨论，并且解释了我们按照系统的分层对模式进行归类的原因。该章也介绍了我们用来记录本书所有模式的“J2EE模式模板”。该章考察了所有的J2EE模式，并且用一张图描述了模式之间的相互关系。另外该章还包括了一种我们称为“模式目录路线图”的东西。这张路线图列举了一些与J2EE设计和架构相关的常见问题，并且把这些问题与特定的模式或重构关联起来，通过这些模式、重构给出了问题的解决方案。理解模式之间的关系以及这张路线图，对于实际应用这些模式至关重要。

第6章“表现层模式”描述了8种模式，它们处理的是在J2EE平台的Web应用设计中，怎样使用servlet、JSP、JavaBeans和定制标记的问题。在这些模式中描述了多种实现策略，并且也提出了一些常见问题，比如请求处理、应用分隔、生成复合视图等。

第7章“业务层模式”，描述了9种模式，它们处理的是怎样应用EJB在J2EE平台下设计业务组件的问题。该章介绍的模式提供了应用EJB和JMS技术的最佳实践。另外，这些模式的相关部分还涉及了其他技术——比如JNDI、JDBC等——的讨论。

第8章“集成层模式”描述了4种模式，它们处理的是怎样把J2EE应用与资源层和各种外部系统集成起来的问题。这些模式使用了JDBC和JMS技术在业务层和资源层之间实现集成。

“尾声”讨论的是一个高层次的主题：怎样利用多个模式一起解决一个大型问题。该章详尽

地讨论了“Web Worker微架构”这个示例，展示了如何通过多个模式来集成一个J2EE应用和一个工作流系统。

## 本书的官方网站和联络信息

在本书的官方网站上，我们会提供内容的更新信息以及其他一些资料。网址是：

<http://www.corej2eepatterns.com>

这个网站也附属于Sun Java蓝图网站：

<http://java.sun.com/blueprints/corej2eepatterns>

你的评论、建议、反馈都可以通过以下邮箱寄给作者：

j2eepatterns-feedback@sun.com

另外还有J2EE模式社区邮件列表服务，邮箱为j2eepatterns-interest@java.sun.com，可以免费订阅和参与。通过以下网址，你可以订阅兴趣小组的邮件，也可以浏览以往的讨论存档：

<http://archives.java.sun.com/archives/j2eepatterns-interest.html>

## 致谢

我们想感谢Sun全球软件服务副总裁Cheryln Chin、Sun杰出工程师和首席服务架构师James Baty，如果没有他们的支持、远见以及他们对我们工作的信赖，本书的工作就不可能完成。

我们愿将最大的感激和谢意致予Rajmohan “Raj” Krishnamurthy。如果没有他的帮助，本书就不会有这么多示例代码，而且我们也从他出色的评论意见中受益匪浅。他对本书新版的规划、开发、评审工作做出了不可或缺的帮助。

本书内容经过多位专家的审读，他们的深刻见解、评论意见、反馈建议，为本书的最终成型做出了重要贡献，通过他们的帮助，各个模式的表述比初稿更加清晰、实用；因此我们也愿对以下专家表示谢意：ThoughtWorks公司首席科学家Martin Fowler；Sun J2EE 蓝图团队的Sean Brydon和Inderjeet Singh；Sun公司的Craig Russel，他是Java数据对象（JDO）技术规范的负责人/产品架构师；ObjectIdentity公司的JDO专家David Jordan；Sun公司的JSP技术规范负责人Mark Roth；Domain Language的Eric Evans；BEA系统公司的解决方案架构师Mario Kosmiskas；LogicLibrary负责技术的副总裁Brent Carlson；Macromedia 的Sean Neville；Sun Java中心的Java架构师Sameer Tyagi；Chris Steel；Bill Dudney；Gary Bollinger；以及ThoughtWorks公司的Gregor Hohpe。

像这样一本书，肯定需要来自各方面的难以计数的帮助才能得以完成，所以我们很难面面俱到地感谢每一个人为此做出的贡献。

我们想感谢James Gosling和Michael Van de Vanter 领导的Sun Jackpot团队，他们的工作将本书推进到了全新的舞台上。

我们还想感谢Chuck Geiger领导的eBay.com V3团队、Terry Bone领导的福特金融中心的ATD框架团队，他们在企业中实际应用了J2EE模式来构建下一代的系统架构和平台。

感谢Sun Java中心的同事Murali Kaundinya、Ashok Mollin、Ramesh Nagappan和Heidi Schuster。

我们想感谢JetBrains公司提供的IntelliJ IDEA开发工具，为本书编写示例代码时我们使用了这种工具，相当满意。

我们还想感谢J2EE模式社区邮件列表（j2eepatterns-interest@sun.com）上的很多成员，多年以来他们的讨论和反馈一直很有帮助。

特别要对本书的技术编辑Solveig Haugland说一声“谢谢”。她是我们团队的重要一员。她在技术上的编辑工作大大提高了本书终稿的质量。

我们想感谢Prentice Hall出版社的Greg Doench和Debby Van Dijk给予我们的信任和鼓励。

特别感谢无糖红牛饮料提供的动力，让我们能每天写作16小时。

## 第1版致谢

我们想感谢Sun全球Java中心的主管Stu Stern和负责.COM咨询的副总裁Mark Bauhaus，如果没有他们的支持、远见以及对我们工作的信赖，本书的工作就不可能完成。

我们想感谢Ann Betser，要不是她的支持、鼓励和循循善诱的建议，我们的工作也不会成功。

我们想对Sun Java中心（SJC）PSA/iWorkflow参考实现开发团队的架构师们表达诚挚的感谢，他们是：Fred Bloom、Narayan Chintalapati、Anders Eliasson、Kartik Ganeshan、Murali Kalyanakrishnan、Kamran Khan、Rita El Khoury、Rajmohan Krishnamurty、Ragu Sivaraman、Robert Skoczylas、Minnie Tanglao和Basant Verma。

我们想感谢Sun Java中心J2EE模式工作组的成员们：Mohammed Akif、Thorbiörn Fritzon、Beniot Garbinato、Paul Jatkowski、Karim Mazouni、Nick Wilde和Andrew X. Yang。

我们想感谢Sun Java中心的首席方法专家Brendan McCarthy，他令我们的工作诸事协调，并提出了大量建议。

我们想感谢把这些模式介绍给客户的Jennifer Helms和John Kapson。

我们想对以下来自世界各地的Sun Java中心架构师表达谢意，他们的支持、反馈、建议都令我们受益匪浅，他们是：Mark Cade、Mark Cao、Torbjörn Dahlén、Peter Gratzer、Bernard Van Haecke、Patricia de las Heras、Scott Herndon、Grant Holland、Girish Ippadi、Murali Kaundinya、Denys Kim、Stephen Kirkham、Todd Lasseigne、Sunil Mathew、Fred Muhlenberg、Vivek Pande、John Prentice、Alexis Roos、Gero Vermaas、Miguel Vidal。

我们想对支持、鼓励我们的管理者Hank Harris、Dan Hushon、Jeff Johnson、Nimish Radia、Chris Steel和Alex Wong表达谢意。

我们还想感谢在Sun公司中与我们合作的以下同事：

Sun软件系统组的Bruce Delagi；Sun软件工程部门的Mark Hapner、Vlada Matena；Forte产品组的Paul Butterworth和Jim Dibble；iPlanet产品组的Deepak Balakrishna；J2EE蓝图团队的Larry Freeman、Cori Kaylor、Rick Saletta和Inderjeet Singh；Heidi Dailey；Java开发者联盟的Dana Nourie、Laureen Hudson、Edward Ort、Margaret Ong和Jenny Pratt。

我们想感谢以下各位对本书的反馈、建议和支持：

ThoughtWorks公司的Martin Fowler和Josh Mackenzie；Richard Monson Haefel；Goldman

Sachs公司的 Phil Nosonowitz 和Carl Reed; Rational软件公司的Jack Greenfield、Wojtek Kozaczynski和Jon Lawrence; TogetherSoft的Alexander Aptus; Zaplets.com 的Kent Mitchell ; Bill Dudney; David Geary; Hans Bergsten; J2EE模式兴趣小组 ([j2eepatterns-interest@java.sun.com](mailto:j2eepatterns-interest@java.sun.com)) 的成员。

我们想对本书的首席技术编辑Beth Stearns表示特别的谢意和感激，她负责整理我们的手稿，让全书明了可读，与此同时还要随时掌控我们的工作进度，同我们一道完成一个高强度的工作计划。

我们想感谢技术编辑Daniel S. Barclay、Steven J. Halter、Spencer Roberts和Chris Taylor，他们出众的专业能力、细致的评审反馈对本书的完成非常重要。

我们想感谢Prentice Hall出版社的Greg Doench、Lisa Iarkowski、Mary Sudul和Debby Van Dijk; Sun公司出版社的Michael Alread和Rachel Borden，他们使本书的诞生成为可能。

# 目 录

对《J2EE核心模式》的好评

译者序

Grady Booch序

Martin Fowler序

前言

## 第一部分 模式和J2EE

第1章 导论	3
什么是J2EE	4
什么是模式	5
历史回顾	5
模式的定义	5
模式的分类	6
J2EE模式目录	7
演化过程	7
怎样使用J2EE模式目录	8
使用模式的益处	9
模式、框架和重用	10
小结	11
第2章 表现层设计考虑和不佳实践	13
表现层设计考虑	14
会话管理	14
控制客户端访问	16
验证	20
助手类属性——完整性和一致性	21
表现层不佳实践	23
多个视图中都包括控制代码	23
把表现层的数据结构暴露给业务层	24
把表现层数据结构暴露给业务领域对象	24
允许重复提交表单	25
把敏感资源暴露给客户端的直接访问	25
假定<jsp:setProperty>会重置Bean属性	26

创建出“胖控制器”	26
把视图助手当成scriptlet使用	26
第3章 业务层设计考虑和不佳实践	31
业务层设计考虑	32
使用session bean	32
使用entity bean	34
缓存EJB的远程引用和句柄	36
业务层和集成层不佳实践	36
把对象模型直接映射为entity bean模型	36
把关系型模型直接映射为entity bean模型	37
把每个用例映射为一个session bean	37
通过Getter/Setter方法暴露EJB的所有属性	38
在客户端中包括服务寻址代码	38
把entity bean当成只读对象使用	39
把entity bean当成细粒度对象使用	39
存储entity-bean的整个从属对象拓扑结构	40
把EJB相关的异常暴露给非EJB客户端	40
使用entity bean finder方法返回大型结果集	41
客户端负责聚合来自业务组件的数据	41
把EJB用于长时间持续的事务	42
每次调用无状态session bean都要重建	42
对话状态	42
第4章 J2EE重构	45
表现层的重构	46
引入控制器	46
引入同步器令牌	48
隔离不同逻辑	51
对业务层隐藏表现细节	57
去除视图中的转换	60
对客户端隐藏资源	63
业务层和集成层的重构	66
用session bean包装entity bean	66

引入业务代表	67
合并session bean	69
减少entity bean之间的通信	70
将业务逻辑移至session bean	71
一般的重构	72
分离数据访问代码	72
按层重构系统架构	73
使用连接池	75

## 第二部分 J2EE模式目录

第5章 J2EE模式概览	81
什么是模式	82
发现模式	83
模式 vs. 策略	83
分层思路	83
J2EE模式	85
表现层模式	85
业务层模式	85
集成层模式	86
J2EE模式目录指南	86
术语	86
UML的使用	88
模式模板	89
J2EE模式关系	90
与现有其他模式的关系	93
模式路线图	93
小结	96
第6章 表现层模式	97
拦截过滤器	98
问题	98
约束	98
解决方案	98
效果	113
相关模式	113
前端控制器	114
问题	114
约束	114

解决方案	114
效果	125
相关模式	125
Context对象	125
问题	125
约束	126
解决方案	126
效果	144
相关模式	144
应用控制器	145
问题	145
约束	145
解决方案	145
效果	171
相关模式	171
视图助手	172
问题	172
约束	172
解决方案	173
效果	186
相关模式	187
复合视图	187
问题	187
约束	188
解决方案	188
效果	195
示例代码	195
相关模式	198
服务到工作者	198
问题	198
约束	198
解决方案	198
效果	201
示例代码	202
相关模式	207
分配器视图	207
问题	207

约束	207	解决方案	274
解决方案	208	效果	281
效果	211	示例代码	282
示例代码	212	相关模式	284
相关模式	216	复合实体	285
<b>第7章 业务层模式</b>	<b>217</b>	问题	285
业务代表	218	约束	286
问题	218	解决方案	286
约束	218	效果	292
解决方案	218	示例代码	293
效果	223	相关模式	303
示例代码	223	传输对象	304
相关模式	227	问题	304
服务定位器	228	约束	304
问题	228	解决方案	304
约束	228	效果	311
解决方案	228	示例代码	311
效果	237	相关模式	318
示例代码	238	传输对象组装器	318
相关模式	247	问题	318
会话界面	247	约束	319
问题	247	解决方案	319
约束	248	效果	321
解决方案	248	示例代码	322
效果	251	相关模式	326
示例代码	252	值列表处理器	326
相关模式	259	问题	326
应用服务	260	约束	327
问题	260	解决方案	327
约束	260	效果	331
解决方案	260	示例代码	332
效果	267	相关模式	338
示例代码	267	<b>第8章 集成层模式</b>	<b>339</b>
相关模式	272	数据访问对象	340
业务对象	273	问题	340
问题	273	约束	340
约束	274	解决方案	340

效果	365	约束	414
相关模式	366	解决方案	414
服务激活器	367	效果	431
问题	367	相关模式	431
约束	367	尾声	433
解决方案	367	Web Worker微架构纵览	434
效果	380	工作流简介	434
相关模式	381	Web Worker微架构	436
业务领域存储	381	问题	436
问题	381	约束	438
约束	382	解决方案	438
解决方案	382	效果	463
效果	412	参考书目	465
相关模式	413	Apache软件授权协议, 1.1版	471
Web Service中转	413	索引	473
问题	413		

# 第一部分 模式和J2EE

第一部分包括以下章节：

- 第1章——导论
- 第2章——表现层设计考虑和不佳实践
- 第3章——业务层设计考虑和不佳实践
- 第4章——J2EE重构

在应用J2EE模式目录中的各个模式时，开发者总需要考虑很多从属的设计问题，这几章讨论了不少这类问题。其中包括的不少问题都会对系统的许多方面（安全、数据完整性、可维护性以及可扩展性等）产生影响。

这些设计问题中，有很多都可以用模式的形式表达、记录，但是与已经列入J2EE模式目录的那些模式相比，它们所主要针对的问题，抽象层次要更低一些。所以，我们没有以模式形式来记录它们，而是采用了一种更随意的格式，仅仅把它们视为在基于模式目录、实现整个系统时，我们所要考虑的设计问题。由于本书的篇幅、论域所限，不可能面面俱到地考察其中的每个问题，但我们还是愿意指出这些关注点，并且鼓励读者自己对这些问题作出进一步的研究。

第1章对模式和J2EE作了一次高层次的考察。该章给出了模式的多种不同定义、讨论了如何给模式分类，并介绍了使用模式的益处。该章为全书的“J2EE模式”主题设定了基调，并告诉读者我们为什么要推出J2EE模式目录。

1  
2  
3

第2章和第3章集中考察了对一些特定问题的不太理想的解决思路——我们把这些解决方案成为“不佳实践”。我们在描述每个不佳实践时，都会给出一个简略的“问题概述”，然后再有一个“参照解决方案”列表。所谓“参照解决方案”，也就是一系列指向本书其他段落的“指针”，并附上相关资料，以此作为对那些问题的推荐解决方法。一般说来，这些“参照解决方案”要么是模式目录中的一种模式，要么是一种重构，要么就是两者的一个混合体。

第4章讨论了J2EE平台下的重构。该章中用来描述重构的格式是基于Martin Fowler的《重构》一书[Fowler]的——对于那些有意深研系统设计的人来说，该书是一部出色的指南。每个重构都首先简要地描述了问题、介绍了解决方案的概况；然后再讨论为什么要做出这样的优化，并详细介绍该种重构的具体作法。

4

---

Θ 边栏所示为原书页码。——编辑注



# 第1章 导论

本章将涉及下列主题：

- 什么是 J2EE
- 什么是模式
- J2EE 模式目录
- 模式、框架和重用

## 学习目标

通过本章的学习，读者将能够：

- 理解 J2EE 和模式的基本概念。
- 了解 J2EE 模式目录的组织结构。
- 识别 J2EE 模式与传统设计模式、框架的区别。
- 了解 J2EE 模式在企业应用开发中的应用价值。

最近5年来，企业软件开发领域的版图发生了极大的变化。处在变化中心的正是Java2企业版平台（J2EE）；它为开发分布式的、针对服务器的应用系统提供了一种统一的技术平台。J2EE技术具有高度的战略意义和强大的功能支持，因此获得了业界的广泛应用；由此，整个软件开发社区都受益于这种开放的标准——我们可以依据此标准来为企业开发基于服务的软件架构。

但与此同时，很多人都把“学习J2EE技术”和“学习应用J2EE技术进行设计”混为一谈。目前已经有很多Java专著对特定的技术领域做出了出色的介绍，但却没有多少书能够讲清楚如何应用特定的技术。

J2EE架构师们要懂得的，不仅仅是相关的API。他们还应该理解以下内容：

- 有哪些最佳实践？
- 有哪些不佳实践？
- 有哪些反复出现的常见问题？对这些问题，有哪些解决方案是已经得到了验证的？
- 7 • 怎样对一个不够理想的设计案例、或者一种不佳实践进行代码重构，最终形成专家们描述为“模式”的那种优秀解决方案？

本书讨论的正是以上问题。而且，通过一种标准的模式模板来记录、交流模式，这也就构成了一种非常强大的沟通、重用机制，由此，我们设计、构建软件的方法也得到了极大的改进。

## 什么是J2EE

J2EE是一种用来开发分布式企业软件应用系统的平台。Java语言从创生之日起，就获得了广泛接纳，经历了巨大的发展。越来越多的技术都成了Java平台的一部分，为了适应不同的需要也开发出了很多全新的API和标准。最终，Sun公司联合了多家业界巨头，在开放的Java社区组织名义下，把所有与企业开发相关的标准、API整合起来，构成了J2EE平台。

对于企业，J2EE平台有很多优势：

- J2EE为企业级运算的许多领域（比如数据库连接、企业业务组件、面向消息的中间件（MOM）、Web相关组件、通信协议以及互操作性）设立了标准。
- J2EE促进人们基于开放的标准开发软件；如此构建的系统实现，出自名门、安全稳固，因此J2EE构成了一种可靠的技术投资。
- J2EE是一种标准的开发平台，基于此开发的软件组件能够在不同厂商的产品中相互移植，从而避免了被一家厂商锁定。
- 在软件开发过程中采用J2EE能够缩短开发周期，使产品尽快投放市场——这是因为，系统的很多底层架构和基础部分都已经由产品厂商按照J2EE规范标准实现出来了。因此大多数IT企业可以不再开发中间件，集中精力构建符合自己商业需要的应用。
- J2EE提高了程序员的生产力，因为对于Java程序员们，相对来说很容易就能学会基于Java语言的J2EE技术。所有企业软件开发都能够在J2EE平台上、利用Java语言完成。
- 8 • J2EE增进了现存各种异构系统之间的互操作性。

## 什么是模式

### 历史回顾

20世纪70年代，Christopher Alexander [Alex, Alex2] 完成了多部专著，其中记录了土木工程学和建筑学中的一些模式。随后，软件开发社区从他的作品中汲取了“模式”的观念，当然，软件社区早就萌生了对类似想法的兴趣。

软件开发中的模式观念通过《设计模式：可重用面向对象软件的基础》一书得到了普及，该书由Erich Gamma、Richard Helm、Ralph Johnson和John Vlissides四位作者（也被成为“四人帮”或GoF）完成。毫无疑问，虽然这“四人帮”的工作使模式成为被全世界的软件开发团队广泛讨论的话题，但记住这一点也很重要：他们书中描述的模式并不是他们自己发明的。相反，只是在发现类似的设计方案在很多项目中反复出现之后，他们才总结出这些模式，把它们记录在书中。

自从GoF的大作出版以来，又有很多软件模式论著相继问世，其中涉及了多种不同领域和用途。在本书最后的参考书目中，我们选入了若干模式论著，也鼓励大家能够深研这些论著中介绍的（不同于本书的）种种模式。

### 模式的定义

模式是人们用来讨论问题和解决方案的。简言之，有了模式，我们就能记录那种已知反复出现的问题，以及在特定上下文中对它的解决方案，并且，借助模式我们还能和他人讨论这些内容。“反复出现”这个词是前文的一个关键要素，因为模式的目的，就是要鼓励反复进行概念重用。

关于这一点，我们在第5章《什么是一个模式》的一节中作了更详细的讨论。

Christopher Alexander在《模式语言》[Alex2]中给模式下了一个非常有名的定义：

“每个模式都是一个法则，由三部分组成。它表现的是一种特定的上下文、一个问题和一个解决方案之间的关系。”

——Christopher Alexander

其后Alexander进一步扩展了他的定义，另外模式领域的著名专家Richard Gabriel [Gabriel]也更详尽地讨论过这个定义[Hillside]。Gabriel把Alexander的定义改写成了一个适用于软件业的版本：

“每个模式都是一个法则，由三部分组成。它表现的是一种特定的上下文，一个特定的、在该上下文中反复出现的约束系统，以及一种能够引导这些约束自身解决的软件构造这三者之间的关系。”（见《Hacking的永恒之道》）

——Richard Gabriel

这个定义看起来相当严格，不过也还另有一些比较宽松的定义。比如说，Martin Fowler在《分析模式》[Fowler2]中就是这么定义模式的：

一个模式，就是在某个实际上下文中有用、并且或许在其他上下文中也有用的想法。

——Martin Fowler

如你所见，模式有很多种定义，但是所有这些定义似乎都有一个共同的主题，那就是在一种特定的上下文中反复出现的一个“问题/解决方案”对子。

模式有以下共通特征：

- 模式是通过经验观察得出的。
- 一般来说模式都要按一种专门的结构格式来记录（见后文“模式模板”部分）。
- 采用模式能够避免重新发明轮子。
- 不同的模式处于不同的抽象层次上。
- 模式要经历不断的改进、完善。
- 模式是可以重用的工作。
- 模式可以用来让大家交流系统设计和最佳实践。
- 多个模式可以拼合起来，从而解决一个大型问题。

有很多头脑卓越的专家都曾经花过很多功夫，专门深研、精练对于软件模式的定义。毫无疑问，我们自己的头脑算不上卓越，也不想花时间进一步讨论这些东西。相反，我们还是要博采众长，更专注于那些最简明的、所有定义中会反复出现的主题。

10

## 模式的分类

既然模式表现的是在一种特定上下文中专家对反复出现的问题的解决方案，那么在很多不同的抽象层次、各种各样的业务领域中就都会发现模式。对于软件模式，人们想出过很多种分类，最常见的如下：

- 设计模式
- 架构模式
- 分析模式
- 创建型模式
- 结构型模式
- 行为型模式

我们仅仅简要地列出了这几种类别，但其中已经包含了多种不同的抽象层次，也有多种正交的分类模型了。事实上，人们已经提出过很多不同的分类方法，但是为了利用模式记录你的思考，并没有哪种分类方法是唯一正确的。

对于我们提供的那个目录里的模式，我们就简单地称作J2EE模式。其中每个模式都既是设计模式也可算作架构模式，而每个模式的“策略”部分，则会考察一些抽象层次比较低的技术细节。所以我们采用的分类方法，仅仅是把模式按照逻辑架构层次分为三类：

- 表现层
- 业务层
- 集成层

也许，当这个模式目录演化到一定地步的时候，模式的数量会比现在多很多，以至于必须

放弃这么简单的三分法，而采用某种更复杂的分类方式。不过在目前，我们还是想保持简明，没有必要就不引入新的术语概念。

## J2EE模式目录

### 演化过程

我们在Sun服务机构Java中心（Sun Services Java Center）与世界各地的客户一道进行J2EE平台上的系统开发，本书中描述的J2EE模式，就是基于源自这些开发的集体经验。Sun服务机构Java中心，作为Sun专业服务机构的一部分，是一家咨询企业，专门为客户提供基于Java技术的解决方案。从J2EE平台诞生之日起，我们就开始基于这一平台创建解决方案了，我们致力于提高系统的服务品质，因此工作目标包括增进系统的可扩展性、可用性和性能。11

在早先的日子里，当我们基于J2EE平台设计、开发、实现各种各样的系统时，我们就已经开始记录自己的经验了，但是当时记录的方式还不太正规，多数都是以设计考虑、设计想法以及札记的形式完成的。随着这个知识库的增长，我们也认识到了，需要以稍微正规一些的文档格式来固化、交流这一类知识。因此，我们改用模式的格式来记录这些想法，因为要是想把那些关于重复出现的问题和解决方案的知识记录下来并用于交流，模式是最理想、最合适的一种形式了。

这项任务的第一步，就是要选定合适的抽象层次，用来划分这些模式。有一些问题和解决方案之间会发生重叠，因为有一些问题的核心是相同的，但解决方案的实现方式则有些差异。为了体现这种重叠，我们必须要考虑抽象层次的问题以及在定义每个模式采取何种粒度的问题。正如你现在看到的这个J2EE模式目录这样，我们最终选取的抽象层次介乎设计模式和架构模式之间。与具体解决方案相关，有很多抽象层次较低的实现细节，在我们的模式模板中（见后文“模式模板”一节），我们把这些细节放到“策略”部分来讨论。这样一来，我们就可以首先从一个比较高的抽象层次讨论模式，同时也没有牺牲对具体实现细节的考察。

每个模式都要经过很多次的命名和重命名。而且，根据整个社区的反馈，每个模式也都经过了很多次的重写。不用说，本书收入的这些模式——就像所有模式一样——都出于持续的改进过程之中，随着技术和规范的变化也一定还将继续演进。

目前，J2EE模式目录中收入了21个模式，分别见于本书第6章“表现层模式”、第7章“业务层模式”以及第8章“集成层模式”。每个模式都是按照我们的模式模板来记录的。12

表1-1列出了目录中收入的模式。

表1-1 J2EE模式目录中的模式

层	模式名称
表现层	拦截过滤器
	前端控制器
	Context对象
	应用控制器

(续)

层	模式名称
业务层	视图助手
	复合视图
	服务到工作者
	分配器视图
集成层	业务代表
	服务定位器
	会话门面
	应用服务
	业务对象
	复合实体
	传输对象
	传输对象组装器
	值列表处理器
Web层	数据访问对象
	服务激活器
	业务领域存储
	Web Service中转

## 怎样使用J2EE模式目录

无论使用哪一类模式，我们都会遇到这样的挑战：怎样才能组合使用模式，让它们发挥最佳作用？正如Christopher Alexander在他的专著《模式语言》[Alex2]所说的那样：

简而言之，没有哪个模式是一个孤立的个体。每个模式都只有靠与其他模式相互支持才得以存在于世界之中：每个模式都嵌入到更大的模式里，被同样大小的模式环绕，并且还有更小的模式嵌入在它的内部——这也就是所谓“相互支持”的意思。

—Christopher Alexander

对于这样一条金科玉律，J2EE模式目录中的模式也并不例外。第5章“J2EE模式概览”介绍了一个模式关系图，描述了在这个模式目录中模式之间是怎样相互支持的。第5章还给出了一张J2EE模式目录路线图，这其实是一张表，列举了一些与J2EE设计和架构相关的常见问题，并且把这些问题与特定的模式或重构关联起来，通过这些模式、重构给出了问题的解决方案。为了从本书的模式中获得最大收益，我们建议你充分理解上述模式关系图和模式路线图。

在你仔细每个模式的时候，你就会发现模式内部还嵌入有多种子模式和策略，同时模式也嵌入在更大的模式中，并且还支持着其他的模式。有时候，某个模式是基于其他一些模式构成的——这可能是J2EE模式目录中的其他模式，也可能是其他著名文献中描述的模式，比如《设计模式：可重用面向对象软件的基础》[GoF]、或者《软件架构模式》系列著作[POSA1, POSA2]中描述过的模式。

为了帮助你进一步理解模式、它们的相互关系以及怎样选取/应用模式，我们在本书的第一部分加入了一些辅助性的章节。我们给出了J2EE平台下的一些不佳实践和重构。对于这些章节中列出的每种不佳实践，我们也给出了相关的重构和模式，采用这些解决方案，就能够缓解不

佳实践造成的问题。在“J2EE重构”一章中，我们介绍了若干种重构，它们能够逐步将一个不太理想的解决方案提升为一个比较理想的解决方案。在每种重构的“作法”部分，我们也列出了相关的模式以及会对该重构产生影响的设计考虑。

最后，在“尾声”一章中，我们讨论了自己在微架构领域研究的新思路。所谓微架构，就是用来构造系统和应用的“积木块”，与模式目录中描述的那些单个模式相比，微架构处于一个更高的抽象层次上。按照我们的考虑，微架构就是相互关联的模式构成的一张网络，作为一个现成的解决方案，它能够解决一个规模较大的问题，比如说，一个子系统的设计。

我们选择了“Web Worker微架构”做一个示范，展示一下怎样把J2EE模式结合起来，用以集成一个J2EE应用和一个工作流系统。

## 使用模式的益处

你可以使用本书中的J2EE设计模式来改善你的系统设计，并且可以在项目生命周期的任何一点上应用这些模式。目录中记录的模式处于一个比较高的抽象层次上，所以在项目前期应用模式就会大有裨益。但也可以采用另一种方式：如果你在具体实现阶段应用一个模式，也许你就需要改写现有的代码。在这种情况下，第4章“J2EE重构”介绍的种种重构可能会对你有所帮助。

14

使用模式有什么益处？下面的段落中，我们介绍了在项目中使用、采取模式的一些益处。简单地说，模式能够：

- 让你利用一个经过验证可行的解决方案。
- 为你提供一套共通语汇。
- 约束解决方案的空间。

### 让你利用一个经过验证可行的解决方案

模式提供的解决方案已经在不同的时间、不同的项目中被反反复复地用于解决类似的问题。所以，模式构成了一种强大的重用机制，能够让开发者、架构师避免重新发明轮子。

### 为你提供一套共通语汇

模式为软件设计者提供了一套共通的语汇。作为设计者，我们使用模式不仅有助于利用、复制成功的设计，而且还有助于在开发者之间通过共通的语汇和格式交流想法。

一个设计师如果不依靠模式，那就需要花上更多的力气才能把自己的设计介绍给其他设计师和开发者。利用模式的语汇，软件设计师能够高效地进行交流。这和真实世界中的情形相似：日常生活中，我们相互沟通、交换意见也要用上一套共通的语汇。就像真实世界中一样，开发者通过学习、理解模式而积累起自己的语汇，而当记录下新的模式之后，大家的设计语汇也就相应地增长了。

一旦开始应用这些模式，你就会注意到自己很快就能把模式的名称融汇到自己的语汇中——而且你也就不再使用那些罗嗦冗长的说法，只简单地提及模式名称了。比如说，假设需要解决

问题的方案必须要使用传输对象模式。一开始，你可能会直接描述这个问题，没有给它加上模式标签。你会说，你的应用需要在EJB之间交换数据，需要在远程调用造成的网络负载之下尽可能提升系统性能，等等。可是，一旦你学会了对这个问题采用传输对象模式，再遇到类似情况你就会用这个简单的术语描述了，而且也能够直接从这个模式入手进行开发。

**15** 为了理解模式语汇对我们的影响，在你和你的团队成员熟悉了模式目录之后，请考虑以下练习。试试不用模式的名称，解释下面这些简单的句子传达的意思（其中来自J2EE模式目录的模式名称用斜体标出）：

- 我们应该在servlet和session bean中使用数据访问对象。
- 用传输对象为EJB传送、获取数据，并且用业务代表封装对所有业务服务的访问，这个想法如何？
- 咱们用一下前端控制器和服务到工作者吧。也许对一些复杂页面，还要用上复合视图。

## 约束解决方案的空间

模式的应用引入了一种重要的设计元素——约束。应用了模式，也就给最终的解决方案的空间带来了约束，或者说创造出了一种边界，设计和实现都必须在这个边界内部完成。因此，模式强烈地要求开发者要让系统实现遵从边界。如果实现越出边界，就会破坏对模式和设计的遵从，这就可能导致出现未经意料的“反模式”。

但模式并不会扼杀创造性。恰恰相反，模式描述了某种抽象层次上的结构或构造。为了在这种边界之内实现模式，设计师和开发者还可以做出很多种具体选择。

## 模式、框架和重用

所谓“软件重用”，真是一个了不起的目标，我们多年以来一直追求实现重用，但至今只获得了相对的、不太显著的成功。事实上，多数商业软件的成功重用都出现在用户界面领域，而不是在业务组件领域——但后者恰恰是我们关注的焦点。作为业务系统架构师，我们力求推进重用，而我们关心的重用都处于设计和架构的层次上。对于推进这个层次上的重用，J2EE模式目录是一种非常有效的途径。

目录中的各模式之间存在多种关系，这些关系常常被视为是模式语言的一部分。我们在图5-2中给出了这些关系的图示。另外，还有一种描述关系的方法，那就是利用“模式框架”的概念——所谓模式框架，也就是在一个整合的应用场景之下的一组模式集合。在很多时候，我们都要在模式的层面上把解决方案组合起来，或者把组件装配到一起，此时“模式框架”的概念就非常重要。上文所说的“微架构”利用了这一概念，并为组合应用模式形成解决方案提供了一个更广阔的基础。

**16** 所以，开发者们不仅要孤立地理解单个模式，还必须注重它们之间的关系和组合；事实上开发者们也一直在询问怎样才能最好地把多个模式连接在一起，形成大型解决方案。我们所说的“利用J2EE模式框架”，恰恰指的就是按这种方式把目录中的模式组合到一起。在这个语境中，所谓“框架”也就是将模式连接起来，形成一个解决方案以实现一组需求。我们认为，这样的应

用方式将推动下一代J2EE开发工具的发展。而对“模式驱动的开发过程”实现自动化需要：

- 确定应用场景，为系统的每个层次提出合适的模式。
- 确定模式组合（或者说解决方案的主旨），从而给出模式框架。
- 为系统方案中的每个角色选定具体实现策略。

在“尾声”一章的微架构部分，我们稍微详尽地考察了这个尚在演进之中的开发领域。

## 小结

到此为止，你应该对模式的构成、本书的论域有了比较充分的理解。下一章将讨论表现层的设计考虑和不佳实践。



## 第2章 表现层设计考虑和不佳实践

本章将涉及下列主题：

- 表现层设计考虑
- 表现层不佳实践

## 表现层设计考虑

当开发者应用本书J2EE目录中列出的表现层模式时，也应该考虑一些相关的设计问题。这些问题在不同的层面上与应用模式进行设计有关，而且，它们也会影响系统的很多方面，比如安全、数据完整性、可维护性与可扩展性。我们将在本章讨论这些问题。

虽然很多这类问题都可以用模式的形式记录，我们却决定不这样做，因为，与模式目录中的表现层模式相比，它们处理的是抽象层次更低的内容。所以，我们不按模式的形式记录它们，而是采用了一种更随意的方法：在基于我们的模式目录实现系统的时候，你肯定会考虑到这些问题，我们正是从这种考虑的角度描述每个问题的。

### 会话管理

“用户会话”这个概念，描述的是在客户端和服务器之间的多次请求构成的一种“对话”。以下段落的讨论都是基于这个“用户会话”概念的。

#### 在客户端保存会话状态

把会话状态保存在客户端，就需要把会话串行化，并且把它放进HTML视图页面，传回客户端。

在客户端保存会话状态有以下优点：

- 相对来说容易实现。
- 当要保存的状态比较少时，效果很好。

而且，当需要在多台物理服务器上实现负载均衡时，使用在客户端保存会话状态的策略，也不需要在服务器之间复制会话状态。

为了在客户端保存会话状态，有两种常见策略——HTML隐藏字段，和HTTP cookie——我们在后文中将讨论这两种策略。第三种策略是直接把会话状态放进页面的URL里。比如像下面这样：

`<form action=someServlet?var1=x&var2=y method=GET>`

虽然这种策略不太常用，但是另外两种策略中包含的很多局限对于它也同样存在。

#### HTML隐藏字段

虽然这种策略相对容易实现，但是采用HTML隐藏字段把会话状态保存在客户端会有很多缺点。当需要保存的状态比较多的时候，缺点就尤其明显。如此保存大量的状态会对系统性能产生负面影响。因为每一张视图页面的标记里都包含会话状态，所以这些状态在每次请求和响应中都要通过网络往复传输。

而且，如果采用隐藏字段来保存会话状态，那么需要保存的状态就只能是字符串形式的值。所以任何对象引用也必须“字符串化”。并且，如果不做特别加密的话，会话状态就会以文本形式直接暴露在HTML编码中。

### HTTP cookie

与隐藏字段策略一样，HTTP cookie策略也是相对容易实现的。不幸的是，它也与前者一样具有很多相同的缺点。尤其是，当保存的状态比较多的时候就会降低系统性能，因为每次请求和响应中，所有的会话状态都必须通过网络往复传输。

同样，为了在客户端保存会话状态，也会出现数据量大小和数据类型的限制。对于cookie header的大小有限制，这样也就限制了能够保存的数据量。而且，和隐藏字段相同的是，当使用cookie保存会话状态的时候，状态也只限于“字符串格式”的值。

## 在客户端保存会话状态时的安全问题

当你把会话状态保存在客户端时，也就引入了必须考虑的安全问题。如果你不想把数据暴露给客户端，那么也就需要用某种加密手法来保护数据。

虽然在一开始，在客户端保存会话状态相对容易实现，但是这种办法有很多缺点，要克服它们可需要不少时间和脑力。所以，对于那些处理大量数据的项目来说——企业系统大多如此——这些缺点远远足以抵消它的优点了。

## 在表现层保存会话状态

当会话状态由服务器管理时，就通过一个会话ID（session ID）来获取状态，状态通常都在服务器端持久保存，除非发生以下情况中的一种：

- 超过了预先定义的会话超时期（timeout）。
- 人工指定会话无效。
- 一个状态从会话中被删除了。

21

**注意：**如果服务器关机（shutdown）了，一些在内存中进行会话管理的机制可能就没法恢复原有的会话数据。

显然，当会话状态量很大时，就应该在服务器端保存会话状态。状态保存在服务器上，你就不会受到数据量大小或是数据类型方面的限制——这些限制都存在于客户端会话管理中。而且，也避免了把会话状态暴露给客户端的安全问题。另外，既然会话状态不会在每个请求中都通过网络传输一次，系统性能也就不会受到影响。

这个策略的灵活性也是一个优点。在服务器上保存会话状态，就可以按照需要和代价，在繁、简之间灵活选择，同时也能兼顾可扩展性和性能。

如果把会话状态保存在服务器上，就必须选择怎样让运行着这个系统的每台服务器都能获取会话状态。当采用多台硬件服务器实现负载均衡时，软件系统就要以集群方式运行，所以就要在这个集群的多个服务器之间复制会话状态，这就有了上面提到的问题，而且这个问题本身还包括很多方面。不过，现在很多应用服务器产品都提供了各种各样的现成解决方案。另外，还可以在比应用服务器更高的层次解决这个问题。比如，当使用负载管理软件（如Resonate公司[Resonate]提供的那一种）时，就可以“粘住”用户（a sticky user experience），也就是说，在同一台服务器上处理同一个用户会话的请求。这个术语也被称为“服务器亲合性（server affinity）”。

另外，还有把会话状态保存在业务层或资源层的办法。可以使用EJB组件在业务层保存会话

状态，在资源层，关系型数据库也可以用于保存会话状态。关于在业务层保存会话的详情，请参阅第3章的“使用session bean”一节。

## 控制客户端访问

出于很多原因，我们需要控制客户端对特定应用资源的访问。在本节中我们就考察两种场景。

**22** 限制或控制客户端访问的原因之一，是要保护视图或者视图的一部分，不令其被客户端直接访问到。比如，当只有注册/登录后的用户才能访问某个特定视图时，或者视图的某个特定部分只能被某种角色的用户访问时，就会出现这样的需要。

描述了这种情况之后，我们将讨论另外一种场景：控制用户访问应用系统的流程。这后一种讨论会考察“重复表单提交”的问题，因为多次提交可能会导致意外的重复事务操作。

## 保护视图

在一些情况下，会限制某些用户完整地访问一项资源。为了达到这个目的有几种策略。第一，可以加入一种应用逻辑，每当控制器或者视图被处理时，就运行这一逻辑，这样就限制了用户访问。第二，可以配置运行时系统（runtime system），在某些资源被访问之前，必须要通过另一种应用资源的内部调用才能完成。这样，对特定资源的访问就被转向另外一种表现层资源（比如servlet控制器）上。因此受限资源也就不能通过一个浏览器调用直接访问了。

处理这种问题的一个常见办法，是采用一个控制器，作为这种访问控制的一个委派点（delegation point）。另一个常见的变种，是在视图中直接加入保护。在后文的“表现层重构”一节和模式目录中，我们讨论了利用控制器进行资源保护的做法，所以这里我们集中考察基于视图的控制策略。我们首先讨论基于视图的策略，然后再考虑通过配置来控制访问的策略。

### 在视图内部中实现保护

在视图的处理逻辑中实现保护，又有两种常见的变体。一种阻塞对整个资源的访问，另一种只阻塞对局部资源的访问。

### 每页加入“要么全部-要么没有”的保护

**23** 在有些情况下，视图处理代码中采用的逻辑会以“要么全部-要么没有（all-or-nothing）”的方式准许或者拒绝用户访问。换句话说，这种逻辑会整个地阻止特定用户访问特定的视图。通常，保护最好还是封装在集中的控制器里，这样处理逻辑就不会充斥在各种代码中了。所以，每页加入保护的办法应该用于只有很少一部分页面需要保护的情况下。往往是这么一种情况：一个不太懂技术的人员要把不多的一些静态页面放到网站上。如果客户端必须登录之后才能查看这些页面，那么就可以在每页的顶端加入一个定制标记助手，用以完成访问检查，如例2.1中所作。

#### 例2.1 每页加入“要么全部-要么没有”的保护

```

1 <%@ taglib uri="/WEB-INF/corej2eetaglibrary.tld" prefix="corePatterns" %>
2
3 <corePatterns:guard>
4 <HTML>
```

```

5 .
6 .
7 .
8 </HTML>

```

### 加入对页面局部的保护

在另一些情况下，视图中的处理逻辑代码只是拒绝对局部视图的访问。这种策略可以和上面提到的“要么全部-要么没有”策略结合使用。为了让讨论更为清晰，我们用一个类比：把这比作对一座大楼中的房间的访问。“要么全部-要么没有”的警卫会告诉用户他们能不能走入房间，而这里第二种警卫的逻辑，则要告诉用户进了房间之后允许他们看什么东西。以下列出了一些例子，说明为什么可能会用到这种策略。

#### 根据用户角色不显示视图的局部内容

根据用户的角色，视图的局部内容可能会不显示出来。比如，当查看机构信息的时候，一个经理就可以访问某个子视图，其中包括员工考评的资料。而下属员工则只能看到机构信息，不能访问任何与员工考评相关的局部用户界面，如例2.2中所示。

#### 例2.2 根据用户角色不显示视图的局部内容

```

1 <%@ taglib uri="/WEB-INF/corej2eetaglibrary.tld"
2   prefix="corePatterns" %>
3
4 <HTML>
5 .
6 .
7 .
8 <corePatterns:guard role="manager">
9 <b>This should be seen only by managers!</b>
10 </corePatterns:guard>
11 .
12 .
13 .
14 </HTML>

```

24

#### 根据系统状态或错误条件不显示视图的局部内容

根据系统环境，显示的布局可能会变化。比如，如果一个管理CPU硬件的用户界面在一个单CPU的硬件设备上使用，那么专门用于多CPU设备管理的视图内容就不应该显示出来。

### 通过配置实现保护

为了限制客户端直接访问特定的视图，可以配置表现层引擎<sup>Θ</sup>，只有先通过某种其他的内部资源（比如一个使用了请求分配器的servlet控制器）才能访问这些资源。而且servlet技术规范2.2以上版本还规定了，Web容器要内置实现一些安全机制，可以利用这些机制来完成配置保护。

<sup>Θ</sup> 表现层引擎：the presentation engine，指Web容器，比如servlet引擎或/jsp引擎（tomcat, jetty等）。通常可以利用配置（比如修改web.xml）来限制对特定资源的访问。

安全控制是通过部署描述符（称为web.xml）来定义的。

Servlet技术规范中规定，基本认证身份方法以及基于表单的身份认证方法都依赖于部署描述符中的配置信息。在这里就不重复技术规范了，可以参照规范的当前版本，来获得这些认证方法的细节信息。（请参照<http://java.sun.com/products/servlet/index.html>）

如果你理解了通过在配置中声明的方法实现安全的做法和效果，我们就简要地就这个问题再做一些讨论，考察一下如何通过配置来实现“要么全部-要么没有”的保护。最后，我们还会描述另一种简单而通用的“要么全部-要么没有”地保护资源的办法。

### 通过标准安全限制实现资源保护

你可以给应用系统配置安全限制，并且可以首先在配置中声明安全限制，再通过编程方式实现基于用户角色的访问控制。指定某些角色的用户可以访问某些资源，而其他人则不能访问。而且，正如上面的“在视图内部中实现保护”一样，页面的局部内容也可以通过用户角色进行限制。如果有一些资源应该整个地拒绝所有浏览器直接访问（像上一节中“要么全部-要么没有”的情况一样），那么可以把这些资源指定到一个未经分配给任何用户的安全角色。只要指定的安全角色没有分配给用户，那么所有的浏览器请求就都不能访问这样配置过的资源。例2.3是web.xml配置文件中的一个片断，定义了一个安全角色来限制浏览器的直接访问。

我们定义的角色名称叫“sensitive（敏感）”，受限访问的资源是sensitive1.jsp, sensitive2.jsp和sensitive3.jsp。如果客户端不是具有“sensitive”这个角色的用户或组，那么就不能直接访问这些JSP页面。另一方面，由于在服务器内部分配请求可以不受这些安全限制的控制，servlet控制器就能够首先处理一个请求，再把它转发给这3项资源：servlet控制器具有访问以上JSP页面的权限。

最后，还要注意，不同的产品厂商对servlet技术规范2.2版的这个特性的实现并不一致。但是支持servlet技术规范2.3版的服务器在这方面应该是完全一致的。

### 例2.3 利用未经分配的安全角色实现“要么全部-要么没有”的访问控制

```

1  <security-constraint>
2      <web-resource-collection>
3          <web-resource-name>SensitiveResources </web-resource-name>
4          <description>A Collection of Sensitive Resources
5          </description>
6          <url-pattern>/trade/jsp/internalaccess/sensitive1.jsp
7          </url-pattern>
8          <url-pattern>
9              /trade/jsp/internalaccess/sensitive2.jsp
10         </url-pattern>
11         <url-pattern>
12             /trade/jsp/internalaccess/sensitive3.jsp
13         </url-pattern>
14         <http-method>GET</http-method>
15         <http-method>POST</http-method>
16     </web-resource-collection>
17     <auth-constraint>
```

```

18      <role-name>sensitive</role-name>
19      </auth-constraint>
20  </security-constraint>

```

### 通过一个简单、通用的配置实现资源保护

还有一个简单、通用的办法，比如一个JSP页面，可以限制客户端直接访问某个特定的资源。这个方法无需像例2.3那样改动配置文件。只需要把那些限制访问的资源放到Web应用的WEB-INF/目录下即可。比如说，在一个名为securityissues的Web应用中，要是想阻挡浏览器直接访问一个叫info.jsp的视图，我们就可以把该JSP源文件放在以下子目录中：

```
/securityissues/WEB-INF/internalaccessonly/info.jsp.
```

对于/WEB-INF/及其子目录，不允许直接的公共访问，所以info.jsp就受到了保护。另一方面，在必要的时候，一个servlet控制器仍然能够把请求转发给该资源。这是一种“要么全部-要么没有”的控制方法，因为这样配置之后，相关资源就整个地不能直接通过浏览器访问了。

这个方法的例子，请参见第4章的4.1.6节“对客户端隐藏资源”。

## 重复的表单提交

使用浏览器客户端的用户一不留意，就可能按了“后退”按钮，把已经提交过的表单重新提交一次，这就可能引起一个重复的事务操作。与此类似，用户还可能在接到确认页面之前就点了“停止”按钮然后又重新提交相同的表单。对于大多数这类情况，我们都希望能够捕捉、禁止这些重复提交，应用servlet控制器就能提供一个处理这种问题的控制点。

### 同步器令牌（又名“似曾相识”令牌<sup>①</sup>）

这种策略专门处理重复表单提交的问题。在用户会话中设置一个“同步器令牌”，每一个返回给客户端的表单中都包含这个令牌。当表单提交时，就对表单中的同步器令牌和会话中的同步器令牌做出比较。当同一表单第一次提交时，两个令牌的值是匹配的。如果令牌不匹配，那么提交的表单就被禁用了，并会返回一个错误信息给用户。发生令牌不匹配的情况可能是：用户提交了表单之后，又点了浏览器的“后退”按钮，然后重新提交同一表单。

另一方面，如果两个令牌的值匹配，那么我们就能够确认控制流程是运行无误的。这时，会话中的令牌值更改为一个新数值，同时表单提交也获得了接受。

这个策略也可以用于控制浏览器直接访问特定页面（作用与上文提到的资源保护是一样的）。比如说，假定一个用户对应用系统中的页面A做了一个书签，但A本来只能通过页面B和C访问。那么，当用户从书签里选择了A时，就违反了规定的访问次序，因此同步器令牌就会处于不同步的状态了——也有可能此时就根本没有令牌。无论是哪种情况，此时如果必要均可以禁止访问。

请参见第4章“表现层重构”的“引入同步器令牌”一节，其中提供了一个使用该策略的例子。

26

27

<sup>①</sup> “似曾相识”原文是“Déjà vu”，在英语中，这是一个来自法语的外来语，直译就是“曾经见过”。为什么同步器令牌/重复表单处理会跟“似曾相识”有关呢？还请从下文寻找答案。

## 验证

经常需要在客户端和服务器两边都进行验证。虽然客户端的验证处理一般比服务器验证简单，但是它提供的往往是层次较高的检查，比如一个表单字段是否为空等等。服务器端验证经常更为复杂。这两种验证都适用于应用系统，但是我们不鼓励系统中只实现客户端验证。不能仅仅依靠客户端验证一个重要的原因是：客户端运行的脚本语言是可以在浏览器中配置的，所以用户可以在任何时候禁用脚本语言。

对验证策略的细节进行考察超出了本书的范围。但我们仍然想提及这些问题，因为在设计系统时不可避免地要考虑它们；我们也同时希望你参阅现有的其他文献，进一步深研验证问题。

### 在客户端验证

输入验证在客户端进行。通常，这需要在客户端视图中加入脚本代码，比如JavaScript。正如上文所说，客户端验证是对于服务器端验证的一个有益补充，但不能仅仅依靠客户端验证。

### 在服务器端验证

输入验证在服务器端进行。服务器端验证有几种常见策略。其中包括基于表单的验证和基于抽象类型的验证。

#### 基于表单的验证

基于表单的验证策略，会强制应用系统加入大量方法，用以验证每个表单提交的各个状态。通常，这些方法在处理逻辑部分有很多重合之处，所以降低了系统的重用度和模块化程度。对于提交的每个Web表单，都要有一个专门的验证方法，所以并没有代码对必填字段或数值字段之类的共通内容进行集中处理。在这种情况下，虽然多个不同的表单中都会有一个必填字段，但是每一次这种字段都要单独处理，因此也就在应用程序的很多地方造成了冗余。这种策略相对来说容易实现，也比较高效，但是应用系统越大，它造成的重复代码就越多。

要想做出一个更灵活、更可重用、更可维护的解决方案，就应该在另一个抽象层次上考虑模型数据。这也就是下面“基于抽象类型的验证”的处理思路，后面的例2.5是一个处理实例。例2.4的代码则是采用基于表单的验证的例子。

#### 例2.4 基于表单的验证

```
/**如果名或姓的字段为空，则会给客户端返回错误。
应用这种策略时，验证所有必填字段的代码
都是重复的。如果这种验证逻辑被抽象为一
个单独的组件，那么就能在多个表单之间实
现重用（见基于抽象类型的验证策略）*/
1 public Vector validate()
2 {
3     Vector errorCollection = new Vector();
4     if ((firstname == null) || (firstname.trim().length() < 1))
5         errorCollection.addElement("firstname required");
6     if ((lastname == null) || (lastname.trim().length() < 1))
7         errorCollection.addElement("lastname required");
```

```

8     return errorCollection;
9 }

```

### 基于抽象类型的验证

无论是客户端或服务器端都可以使用这个策略，不过最好还是利用基于浏览器或者“瘦客户端”的环境，在服务器端实现这个策略。

从状态中抽象出类型和限制信息，放入一个通用的框架中。这也就区分了模型的验证与应用这些模型的业务逻辑，从而降低了二者的耦合。

在验证模型时，进行的工作是在模型状态与元数据、限制信息之间做比较。关于特定模型的元数据和限制信息通常保存在某种简单的存储介质（比如一个属性文件）中。这种做法的一个优势是系统就能够更为通用，因为类型和限制信息被从应用逻辑中提取出来了。

举例来说，可以使用一个组件或者一个子系统来封装验证逻辑，例如验证字符串是否为空、数值是否在有效的范围内、字符串是否符合特定格式，等等。如果多个不同的业务应用组件需要验证模型的不同方面，那么每个业务组件用不着自己包含验证代码。相反，会采用集中化的验证机制。通常，这种集中化的验证既可以编程实现（利用某种工厂模式），也可以利用配置声明实现（通过某种配置文件）。

29

因此，这样的验证机制就更为通用，更专注于模型的状态以及对它的要求，而独立于应用的其他部分。应用这种策略的一个缺陷是，在效率和性能上它可能具有潜在的损失。另外，一种比较通用的解决方案虽然往往相当强大，但是也经常是难于理解、不易维护的。

下面就是一个实际的样例场景。用了一个XML配置文件来描述各种验证要求，比如“必填字段”、“数字字段”等等。然后再用一些处理器类（handler classes）实现其中每个验证。最后，在HTML表单的值和具体的验证类型之间建立映射。例2.5中给出了验证一个特定表单字段的代码片断。

### 例2.5 基于抽象类型的验证

```

1 //firstNameString = "Dan"
2 //formFieldName = "form1.firstname"
3 Validator.getInstance().validate(firstNameString, formFieldName);

```

## 助手类属性——完整性和一致性

JavaBean助手类通常用于存放由客户端请求传来的中间状态。JSP运行时引擎提供了一种机制，自动地把servlet请求对象中的参数值复制到JavaBean助手类中。JSP的相关句法是这样的：

```
<jsp:setProperty name="helper" property="*"/>
```

这句话就能让JSP引擎把所有匹配的参数值都复制到一个叫做“helper”的JavaBean的属性中，该JavaBean的代码如例2.6所示。

### 例2.6 助手类属性——一个简单的JavaBean助手

```

1 public class Helper
2 {
3     private String first;

```

30

```

4     private String last;
5
6     public String getFirst() {
7         return first;
8     }
9
10    public void setFirst(String aString) {
11        first=aString;
12    }
13
14    public String getLast() {
15        return last;
16    }
17
18    public void setLast(String aString) {
19        last=aString;
20    }
21
22 }

```

但是“匹配”是怎么确定的呢？如果一个请求的参数中，与助手类中的属性对比，存在名称和类型都一样的参数，那么这就被视为“匹配”了。所以，每个参数都要与助手类的每个属性比较名称，如果相同，则再比较它的类型和该属性setter方法的类型。

虽然这个机制相当简单，但是它也可能产生一些让人混淆、出人意料的副作用。首先我们要做一个重要的说明：当请求中的参数值为空的时候会发生什么。很多开发者假定，当一个请求参数的值是一个空字符串的时候，如果它和助手类的一个属性匹配，那么该属性也就应该赋值为空字符串或者null。但是，技术规范则规定，在这种情况下不对该属性的值做任何变化。而且，因为JavaBean助手类的实例通常要在多个请求之间重用，因此这种混淆就可能会导致数据的不一致和不正确。

图2-1描绘了这种情况会引起的问题。

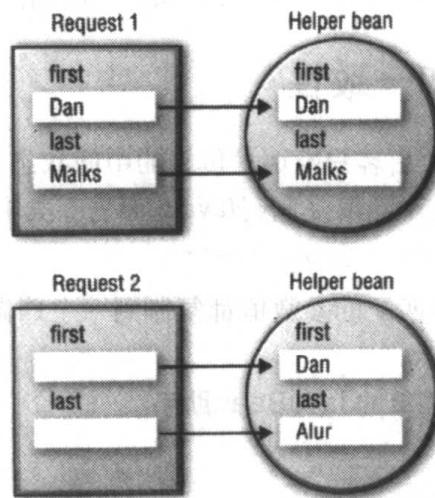


图2-1 助手类属性

请求1 (Request1) 中包含了参数“first”和“last”的值，因此助手类 (Helper bean) 的两个相应属性都赋了值。请求2 (Request2) 却只包含了“last”参数的值，因此助手类只有一个属性被赋值。而“first”属性的值没有变化。它并没有被重置为空字符串或者null，因为请求参数中并不包括这个值。正如图2-1所示，如果在两次请求之间不手工重置助手类的值的话，就会产生数据的不一致。

与此相关，还有另一个问题需要在设计应用的时候加以考虑：当HTML表单中的控件没有被选择时，这个HTML表单会怎样提交。比如说，如果一个表单中有一组多选框，那么，看上去一个挺合理的推论是：如果每个选择框都没有选中，在服务器端则会清空所有值。但是，其实在这种情况下，在这个表单构成的请求对象中根本不会有这一组多选框的参数出现。因此，服务器也就不会收到任何与这一组多选框相关的参数（关于HTML技术规范全文，请参考<http://www.w3.org>）。

既然没有数值被传送到服务器上，那么即使加入了<jsp:setProperty>操作，匹配的属性还是不会改变。所以，在这种情况下，除非开发者手工更改属性值，应用程序中就会包含潜在的不一致、不正确的数据。如上所述，对这个问题有一个简单的解决方案，那就是在多次请求之间要重置JavaBean的所有状态。

31

32

## 表现层不佳实践

所谓不佳实践，也就是与模式推荐的思路冲突一些不太理想的解决方案。当我们记录下模式和最佳实践的时候，自然而然地也就放弃了这些不太理想的方法。

在本书的这一部分，我们着重讨论了（我们所认为的）表现层不佳实践。

在以下每节中都简要地描述了一种不佳实践，然后再提供了大量的参照内容，包括设计问题、重构、模式等等，这样也就为解决特定的问题提供了进一步的信息和更好的方案。但是，对于每个不佳实践我们并不做深入讨论，而是给出了提纲挈领的描述，作为此后深入考察的出发点。

“问题概述”部分提供了对一个不太理想的解决方案的简述，参考解决方案部分则包括：

- **模式**提供了这类应用场景的模式信息以及实现代价；
- **设计考虑**提供了相关的设计细节；
- **重构**描述了从一个不太理想的解决方案（所谓不佳实践）到一个理想的解决方案、一种最佳实践或一个模式的过程。

请把本书的这个部分视为一张路线图，利用这里的参考资料来查找见于本书其他部分的细节和描述。

### 多个视图中都包括控制代码

#### 问题概述

每个JSP视图的开头都可以包括一些定制标记助手类，用于实现访问控制和其他各种检查。如果大量视图都包括相似的助手类引用，那么这些代码就会变得难于维护，因为一点儿修改就

要引起多处的变动。

## 参照解决方案

合并控制代码，引入一个控制器和相关的命令助手。

- **重构** 见第4章的“引入控制器”。
- **重构** 见第4章的“隔离不同逻辑”。
- **模式** 见第6章的前端控制器中的“命令与控制器策略”部分。

33 当必须要在多个文件中包含类似的控制代码时——比如对于特定用户只显示JSP视图的局部内容——应把这个操作委派给一个可重用的助手类。

- **模式** 见第6章“视图助手”部分。
- **设计考虑** 见本章的“保护视图”部分。

## 把表现层的数据结构暴露给业务层

### 问题概述

表现层的数据结构，比如说HttpServletRequest，应该只限于表现层。把这一类细节暴露给业务层（或者其他任何层），都会增加这些层次之间的耦合，从而急剧降低服务的可重用度。如果业务层方法的参数表中有一个HttpServletRequest类型的输入参数，那么这个服务的任何客户端（甚至不是在Web环境中的客户端）也都要把它们的请求状态包装成HttpServletRequest对象。而且，在这种情况下，业务层的服务需要懂得如何跟表现层专用的数据结构通信，这也就增加了业务层代码的复杂度，增加了各个层次之间的耦合。

## 参照解决方案

不要让业务层使用那些原本专门用于表现层的数据结构，而应该把相关的状态复制到通用的数据结构中，让两个层次共享这些通用数据结构。或者，还可以把相关状态从表现层专用的数据结构中抽取出来，作为独立的参数在各层次之间传递。

34

- **重构** 见第4章“对业务层隐藏表现细节”一节。

## 把表现层数据结构暴露给业务领域对象

### 问题概述

把原本用于处理请求的数据结构（比如HttpServletRequest）暴露给业务领域对象，会不必要地增加应用系统中两个独立方面之间的耦合。业务领域对象应该是可重用的组件，如果它们的实现要依赖于与通信协议或系统分层相关的细节，那么它们的重用潜力也就会大大降低。而且，维护和调试这种紧耦合的应用系统也要困难得多。

## 参照解决方案

不要把HttpServletRequest对象作为参数传给业务领域对象，而应该从请求对象中复制出状态，然后放进一个共通的数据结构，再把这个新对象传给业务领域对象。或者，还可以从HttpServletRequest对象中抽取出相关状态，然后把状态的各个部分作为独立的参数传递给业务领域对象。

- **重构** 见第4章“对业务层隐藏表现细节”一节。

## 允许重复提交表单

### 问题概述

一个桌面应用程序可以控制客户端的用户导航，而基于浏览器的客户端环境就缺乏这种控制，这是这种环境的一个局限。比如，用户可能提交了一个订货表单，这就产生了一个事务操作：从信用卡账号划款，并启动送货流程。但是，在收到确认页面之后，如果用户点击了“后退”按钮，同一份表单还会被重新提交。

## 参照解决方案

为了解决这个问题，需要监管、控制请求流程。

- **重构** 见第4章的“引入同步器令牌”一节。
- **重构** 见本章前面的“控制客户端访问”部分。
- **设计考虑** 见本章前面的“同步器令牌（又名“似曾相识”令牌）”部分。

35

## 把敏感资源暴露给客户端的直接访问

### 问题概述

安全性是企业环境开发的最重要的问题之一。如果并不需要让客户端直接访问某些信息，则这种信息必须要受到保护。如果某些特定的配置文件、属性文件、JSP页面和类文件没有得到妥帖的保护，那么客户端就可能不经意地或恶意地获得敏感信息。

## 参照解决方案

保护敏感资源，禁止客户端直接访问。

- **重构** 见第4章的“对客户端隐藏资源”一节。
- **重构** 见本章前面的“控制客户端访问”部分。

## 假定 `<jsp:setProperty>` 会重置Bean属性

### 问题概述

虽然标准标记`<jsp:setProperty>`的通常作用是把请求中的参数值赋值到JavaBean助手类的同名属性中，当参数值为空的时候这一标记的作用往往会引起混淆。比如，空值的参数会被系统忽略，但很多开发者却错误地假设相应的JavaBean属性也会被赋给null或空字符串值。

### 参照解决方案

记住`<jsp:setProperty>`的这种不太直观的赋值机制，在使用bean属性之前先做初始赋值。

- **设计考虑** 见本章前面的“助手类属性——完整性和一致性”一节。

## 创建出“胖控制器”

### 问题概述

在多个JSP页面中重复的控制代码往往应该被重构，放入一个控制器。但是，如果太多的代码都放进了一个控制器里，控制器本身就会变得太重，难于维护、测试和调试。比如说，要是给一个servlet控制器（尤其是一个“胖控制器”）做单体调试，就要比给一个独立于HTTP协议的单独的助手类做单体测试复杂得多。  
36

### 参照解决方案

控制器通常是处理请求的第一个接触点，但是它应该也是一个委派点，与其他控制类协作完成操作。可以把控制代码封装到命令对象（command objects）中，然后再让控制器把操作委派给这些命令对象。对这种独立于servlet引擎的JavaBean命令对象进行单体测试很容易，而测试模态化不强的控制器代码则相对困难。

- **重构** 见第4章的“引入控制器”一节。
- **模式** 见第6章的“命令与控制器策略”一节的前端控制器部分一节。
- **重构** 见第4章的“隔离不同逻辑”一节。
- **模式** 见第6章的“视图助手”一节。

## 把视图助手当成scriptlet使用

### 问题概述

应用了视图助手，就能够把视图中包含的Java scriptlet代码数量降低到最少。但是，如果这些助手仅仅体现了与原来视图中的Java scriptlet代码相同层次的抽象，那么即使采用了助手，还是和scriptlet中一样暴露了代码的具体实现——这正与引入助手的初衷相反。

## 参照解决方案

本节中给出了同一段代码的4种思路，首先是把HTML标记和scriptlet代码混用，后面的每个例子都递增地做出一点儿改进。在决定自身应用系统的结构的时候，也要权衡一下这些考虑。

如例2.7所示，在视图中包含的Java scriptlet代码包含了格式处理逻辑的太多实现细节。使用视图助手取代这种scriptlet代码，就能够隐藏格式处理逻辑的实现细节。虽然也可以让助手暴露一些实现细节，但是应该尽量避免这种暴露，让助手体现出更高层次的抽象。

### 例2.7 视图中的Java Scriptlet

```

1   <html>
2   <head><title>Member Ratings</title></head>
3   <body>
4
5   <jsp:useBean id="memberlist" type="java.util.List" scope="request"/>
6
7   <div align="center">
8   <h3> 10 Most Credit-Risk Members</h3>
9   <table border="1" >
10  <tr>
11      <th> Name </th>
12      <th> Phone </th>
13      <th> Email </th>
14      <th> City </th>
15      <th> Balance </th>
16  </tr>
17  <%
18 Iterator members = memberlist.listIterator();
19 int maxrows = 0;
20 while ( members.hasNext() ) {
21     MemberTO member = (MemberTO)members.next();
22     // 业务规则：这张报表里不包括白金会员
23     if ( ! member.getPrivilege().equalsIgnoreCase("Gold") ||
24         member.getPrivilege().equalsIgnoreCase("Silver") ) &&
25         (member.getCreditBalance() > 5000) && (maxrows < 10) ) {
26     %>
27     <tr>
28         <td><%= member.getName()%></td>
29         <td><%= member.getPhone()%></td>
30         <td><%= member.getEmail()%></td>
31         <td><%= member.getCity()%></td>
32         <td><%= member.getCreditBalance()%></td>
33     </tr>
34  <%
35     maxrows++;
36 }
37 }
```

```

38  %>
39  </table>
40
41  </div>
42
43  </body>
44
45  </html>
```

38

如例2.8所示，按这种方法使用JSTL[JSTL]助手，和例2.7中的Java scriptlet代码一样暴露出了实现细节。这样使用助手确实是一种改进，但并没有最大程度地体现助手在可读性、模块化和重用度上的优势。

### 例2.8 使用JSTL助手

```

1  <%@ taglib prefix="c" uri="http://java.sun.com/jstl/core" %>
2
3  <html>
4  <head><title>Member Ratings</title></head>
5  <body>
6
7  <jsp:useBean id="memberlist" type="java.util.List" scope="request" />
8
9  <div align="center">
10 <h3> 10 Most Credit-Risk Members</h3>
11 <table border="1" >
12   <tr>
13     <th> Name </th>
14     <th> Phone </th>
15     <th> Email </th>
16     <th> City </th>
17     <th> Balance </th>
18   </tr>
19   <c:set value="0" var="RowCount"/>
20   <c:forEach var="member" items="${memberlist}">
21   <c:if test="${(member.privilege == 'Gold' or member.privilege=='Silver') & (member.creditBalance > 5000) and (RowCount < 10 )}">
22     <tr>
23       <td><B><c:out value="${member.name}" /></B></td>
24       <td><c:out value="${member.phone}" /></td>
25       <td><c:out value="${member.email}" /></td>
26       <td><c:out value="${member.city}" /></td>
27       <td><c:out value="${member.creditBalance}" /></td>
28     </tr>
29   <c:set value="${RowCount+1}" var="RowCount"/>
30   </c:if>
31 </c:forEach>
32 </table>
33 </div>
```

39

```

35  </body>
36  </html>
```

例2.9中，视图助手提供了一个更高层次上的抽象，这样在读代码的时候，就能够马上理解它的意图——构造一张会员表。例2.9在“buildMemberTable”标记中混合了一些HTML标记，但是提供了一种模块化的、可重用的代码。

### 例2.9 使用标记库

```

1  <%@ taglib uri='/WEB-INF/corej2eetaglibrary.tld' prefix='cjp' %>
2
3  <html>
4  <head><title>Member Ratings</title></head>
5  <body>
6
7  <jsp:useBean id="memberlist" type="java.util.List" scope="request"/>
8
9  <div align="center">
10 <h3> 10 Credit-Risk Members</h3>
11 <cjp:buildMemberTable privilege="Gold,Silver" credit="5000" rows="10"/>
12 </div>
13 </body>
14 </html>
```

最后，例2.10是标记文件（需JSP2.0以上版本支持），这种做法不仅具备定制标记的可读性和抽象层次，还具有上述JSTL例子的易用性（另见例2.11）。

### 例2.10 标记文件助手

```

1  <%@ taglib prefix="tags" tagdir="/WEB-INF/tags" %>
2
3  <html>
4  <head><title>Member Ratings</title></head>
5  <body>
6
7  <div align="center">
8  <h3> 10 Most Credit-Risk Members</h3>
9
10 <tags:memberList privilege="Gold" credit="5000" rows="10">
11   <jsp:attribute name="nameStyle">
12     <B>${name}</B>
13   </jsp:attribute>
14 </tags:memberList>
15
16   <br>
17
18 <tags:memberList privilege="Silver" credit="4000" rows="7">
19   <jsp:attribute name="nameStyle">
20     <font color="red"><B>${name}</B></font>
21   </jsp:attribute>
22 </tags:memberList>
```

```

23
24  </body>
25  </html>
```

### 例2.11 标记文件助手

```

1   <%@tag import='java.util.*'%>
2   <%@tag import='util.*'%>
3   <%@ taglib prefix="c" uri="http://java.sun.com/jstl/core_rt" %>
4   <%@ attribute name="nameStyle" fragment="true" %>
5   <%@ attribute name="privilege"%>
6   <%@ attribute name="credit"%>
7   <%@ attribute name="rows"%>
8
9   <%@ variable name=given="name" %>
10
11 <jsp:useBean id="memberlist" type="java.util.List" scope="request"/>
12
13 <table border="1" >
14   <tr>
15     <th> Name </th>
16     <th> Phone </th>
17     <th> Email </th>
18     <th> City </th>
19     <th> Balance </th>
20   </tr>
21 <c:set value="0" var="RowCount"/>
22 <c:forEach var="member" items="${memberlist}">
23 <c:if test="${(member.privilege == privilege)
24           and (member.creditBalance > credit) and (RowCount < rows )}">
25   <tr>
26     <td>
27       <c:set var="name" value="${member.name}"/>
28       <jsp:invoke fragment="nameStyle"/>
29     </td>
30     <td><c:out value="${member.phone}" /></td>
31     <td><c:out value="${member.email}" /></td>
32     <td><c:out value="${member.city}" /></td>
33     <td><c:out value="${member.creditBalance}" /></td>
34   </tr>
35   <c:set value="${RowCount+1}" var="RowCount" />
36 </c:if>
37 </c:forEach>
38 </table>
```

41

42

## 第3章 业务层设计考虑和不佳实践

本章将涉及下列主题：

- 业务层设计考虑
- 业务层和集成层不佳实践

43  
45

在上一章中，我们讨论了如何通过集成层来实现企业级应用的松耦合。在这一章中，我们将深入探讨业务层设计，以及如何通过业务层设计来提高企业的竞争力。业务层是企业级应用的核心，它负责处理企业的核心业务逻辑。因此，业务层的设计是否合理，将直接影响到企业的整体竞争力。在这一章中，我们将讨论业务层设计的一些常见问题，以及如何避免这些问题。

### 3.1 业务层设计考虑

在设计业务层时，需要考虑以下几个方面：一是业务逻辑的抽象程度。业务逻辑的抽象程度越高，就越容易被复用。二是业务逻辑的可维护性。业务逻辑的可维护性越高，就越容易进行修改和优化。三是业务逻辑的可扩展性。业务逻辑的可扩展性越高，就越容易进行新的功能添加。四是业务逻辑的可测试性。业务逻辑的可测试性越高，就越容易进行单元测试和集成测试。五是业务逻辑的可移植性。业务逻辑的可移植性越高，就越容易移植到不同的平台上。

在设计业务层时，还需要注意以下几点：一是避免过度设计。过度设计会导致业务逻辑过于复杂，难以维护。二是避免重复设计。重复设计会导致业务逻辑过于冗余，降低系统的性能。三是避免死锁。死锁会导致系统无法正常运行。四是避免线程安全问题。线程安全问题会导致线程之间的数据冲突，影响系统的稳定性。

在设计业务层时，还需要注意以下几点：一是避免过度设计。过度设计会导致业务逻辑过于复杂，难以维护。二是避免重复设计。重复设计会导致业务逻辑过于冗余，降低系统的性能。三是避免死锁。死锁会导致系统无法正常运行。四是避免线程安全问题。线程安全问题会导致线程之间的数据冲突，影响系统的稳定性。

## 业务层设计考虑

在使用本书介绍的业务层和集成层模式时也需要了解一些相关的设计问题——本章就将讨论这些问题。这里包括多种主题，也会影响到系统的很多方面。

当基于J2EE模式目录实现应用系统时，肯定要考虑这些问题，本章的讨论正是从这种考虑的角度，简要描述每个问题的。

### 使用session bean

按照EJB技术规范，session bean是一种具备以下特征的分布式业务组件：

- 每个session bean专门服务于一个客户端或用户。
- 每个session bean的生命时间等于客户端的会话时间。
- session bean在服务器崩溃后不能存活。
- session bean不是一个持久化对象。
- session bean会超时（time out）。
- session bean可以涉及事务。
- session bean既可以用来构造客户端和业务层组件之间的有状态对话模型，也可以用来构造二者间的无状态对话模型。

**注意：**在本节中，我们在EJB的语境中使用了“工作流”的概念，来表现与EJB通信相关的逻辑。比如说，session bean A怎样调用session bean B，然后再调用entity bean C，这就是一种“工作流”。

### session bean——无状态vs.有状态

有两种不同风格的session bean：无状态的和有状态的。一个无状态session bean不保存任何对话状态。所以，只要一个客户端对一个无状态session bean的方法调用完成了，容器就可以再把该session bean用于另一个客户端。这就允许容器维护一个session bean的池，并在多个客户端之间重用session bean。容器把无状态session bean保存在池中，这样就能更高效地在多个客户端之间实现共享。一个客户端完成了对无状态session bean的方法调用后，容器就把该session bean放回池内。而同一客户端的下一次调用可能被容器分配给池里的另一个session bean实例。  
46

有状态session bean保存对话状态。有状态session bean也可能被放入池中，但是既然这种session bean负责保存一个特定客户端的状态，它就不可能同时被多个客户端共享，也不可能既处理这个客户端、又处理那个客户端的请求。

容器把有状态session bean放入池中的机制，和无状态session bean是不同的，因为有状态session bean要保存客户端的会话状态。有状态session bean被分配给一个客户端，而且只要该客户端会话依然活跃，那么它就被一直分配给那个客户端。所以为了维持会话状态，有状态session bean比无状态session bean需要更多的资源负载。

很多设计者认为，对于设计可扩展的系统而言，使用无状态session bean是一种更可行的session bean设计策略。这种想法来自使用一些旧式技术构建分布式对象系统的经验，因为这些旧式技术缺乏内置的管理组件生命周期的基础架构，所以当系统对资源的需求增长的时候，这些系统很快地就会失去可扩展的特性。之所以会丧失可扩展性，是因为旧式技术缺乏对组件生命周期的管理，因此随着客户端和对象的数量的上升，服务消耗的资源也会持续增加。

EJB容器管理EJB的生命周期，并且负责监控系统资源，从而能够实现对EJB实例的最优管理。容器管理EJB池，并且负责把EJB放入、放出内存（分别称为“激活”和“钝化”），这样就能够优化调用和资源的消耗。

如果在可扩展性上存在问题，通常是由对有状态session bean和无状态session bean的误用造成的。必须按照实现的业务流程选择使用有状态或无状态session bean。如果一个业务流程只需要一次方法调用就能完成服务，这就是一个无对话的业务流程，这样的流程适合由无状态session bean实现。如果一个业务流程需要多次方法调用才能完成服务，这就是一个有会话的业务流程，适合用有状态session bean实现。

但是，有些设计者喜欢用无状态session bean，认为这样能够提高可扩展性，而且他们可能会错误地把所有业务流程都用无状态session bean建模。如果使用无状态session bean来实现现有会话的业务流程，每次方法调用都需要客户端把状态传递给EJB，或者在业务层重建状态，或者需要从持久化存储中获取状态。这些技术手段都可能导致可扩展性的降低，因为它们要么会导致网络负载、要么需要一定的重建时间、要么会花费时间访问持久化层。

## 在业务层保存状态

在第2章的“在表现层保存会话状态”部分，我们讨论了把状态保存在Web服务器上的设计考虑。这里继续讨论，并着重考察一下什么时候才适合用有状态session bean而不是HttpSession保存状态。

一个考虑是，要确定访问你的业务系统的客户端是哪种类型的。如果系统架构只是一个基于Web的应用，所有的客户端要么通过servlet、要么通过JSP，经由Web服务器来访问系统，那么对话状态就可以在Web层由HttpSession负责维护。这个场景如图3-1所示。

另一方面，如果你的应用系统支持多种类型的客户端，包括Web客户端、Java应用程序、其他应用程序、甚至还有其他的EJB，那么对话状态就可以在EJB层由有状态session bean来维护，如图3-2所示。

在前面（见第2章的“在客户端保存状态”部分）和本章中，我们对状态管理的主题进行了一些基本讨论。完整、深入的讨论超出了本书的论域，因为这个问题有很多层面，而且非常依赖于部署环境，包括：

- 硬件
- 负载管理
- Web容器的集群
- EJB容器的集群
- 服务器亲合性

47

48

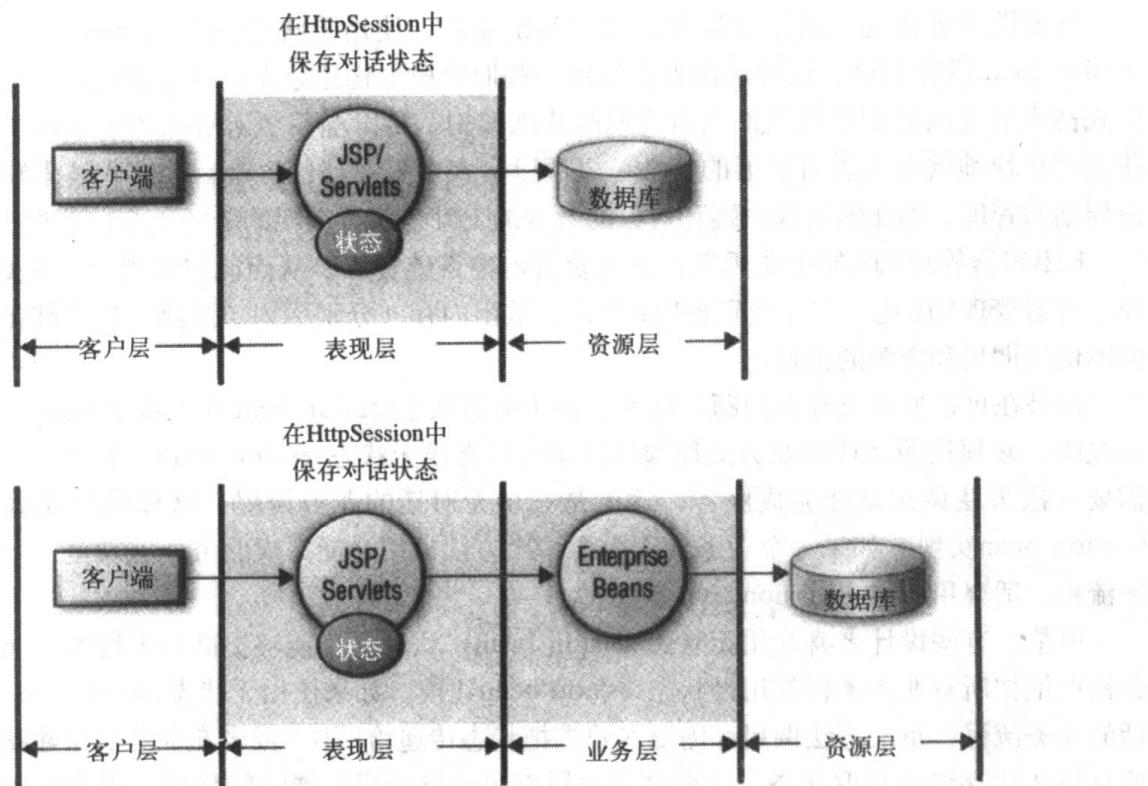


图3-1 在 HttpSession 中保存状态

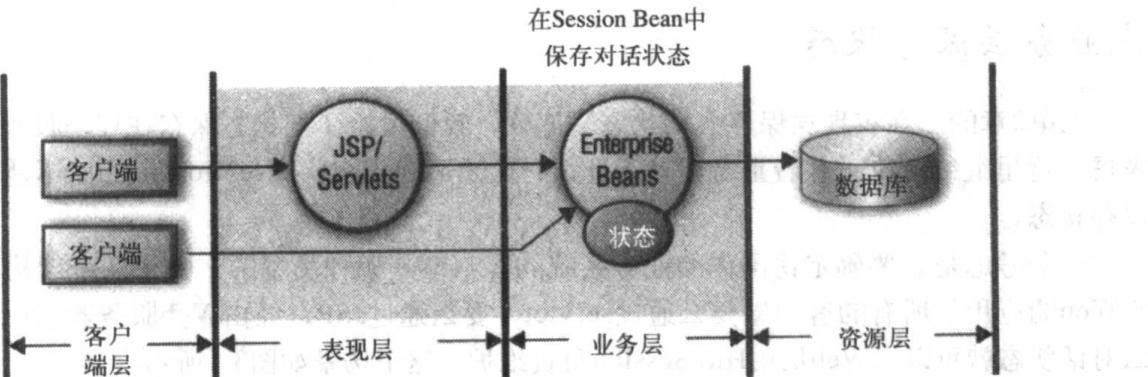


图3-2 在 Session Bean 中保存状态

- 会话复制

- 会话持久化

我们略微涉及了这个问题，因为在开发和部署的整个过程中都应该考虑到它。

## 使用 entity bean

能否恰当地使用 entity bean，与洞察力、经验、需要、技术各方面都有关系。entity bean 适合作为粗粒度的业务组件使用。按照 EJB 技术规范，entity bean 是具备以下特征的分布式对象：

- entity bean 为持久化数据提供了一种对象视图。
- entity bean 具有事务。

- entity bean服务于多个用户。
- entity bean是持续存活的。
- 在容器崩溃之后entity bean还能够存活。这样的崩溃对于它的客户端来说通常是透明的。

总结了以上定义之后，entity bean的恰当用法也就是作为一种分布式的、共享的、有事务的、持久化的对象使用。另外，EJB容器还具备其他的基础架构，用于支持一些系统特质，包括可扩展性、安全性、性能、集群等。总而言之，对于利用分布式业务组件实现和部署应用系统，这一切构成了一种非常可靠、健壮的平台。

## entity bean的主键

每个entity bean都通过主键唯一地识别。主键可以是一个简单键，由单独一个属性构成，也可以是复合键，由来自entity bean的一组属性构成。如果entity bean本来就有一个单字段的主键，而且该主键也属于java的基本数据类型，那么在实现entity bean时就可以不再另外定义一个外在的主键类。部署者可以在部署描述符中指定entity bean的主键字段。但是，当主键是一个复合键的时候，就必须定义一个单独的主键类。这个类必须是一个简单java类，实现了Serializable（可串行化）接口，并且具备该entity bean的复合键的那些属性。主键类的属性名称、类型必须与entity bean匹配，而且必须在entity bean的实现类和主键类中都声明为public（公共）。

主键类应该实现java.lang.Object的可选方法，比如equals和hashCode方法——这是一个推荐实行的最佳实践。

- 之所以覆盖equals()方法，是为了通过比较复合键的每个部分，来正确地确定两个主键是否同一。
- 之所以覆盖Object.hashCode()，是为了返回一个能代表每个主键实例hash code（哈希码）的唯一数值。要使用主键的各个属性计算出这个hash code，以保证对于每个主键这个值确实唯一。

## entity bean中的业务逻辑

在entity bean设计中常常遇到的一个问题是：它应该包含什么样的业务逻辑。有些设计者认为，entity bean应该只包含持久化逻辑，以及一些获取/设置数据值的简单方法，即entity bean不应该包含业务逻辑。这个想法又经常被误解为“entity bean中只应该包含与获取/设置数据值相关的代码”。

50

总体来说，所谓“业务逻辑”包括任何提供某种服务的处理逻辑。对于我们的讨论，不妨把“业务逻辑”视为与处理操作、工作流、业务规则、数据等等有关的所有逻辑。以下列举了一系列样本问题来考察把业务逻辑加入entity bean的可能后果：

- 这样的业务逻辑是否会引入实体之间（entity-entity）的关系？
- entity bean是否要负责管理用户交互的工作流？
- entity bean是否会担负起本该属于其他业务组件的责任？

如果你对以上问题中的任何一个回答“是”，就说明在这个entity bean中引入业务逻辑会产

生负面影响；尤其是当使用远程entity bean的时候。最好应该改进设计，尽可能避免entity bean之间的依赖关系，因为这种依赖关系会增加系统负担，从而降低应用系统的整体性能。

总体来说，entity bean包含的业务逻辑应该是自足的，只处理它本身的数据和它的从属对象的数据。因此，可能很需要完成以下重构：找出那些引入entity bean之间交互的业务逻辑，把它们从entity bean中抽取出来，利用会话门面模式将它们转移到一个session bean中。第7章中的复合实体模式和一些重构都讨论了与entity bean设计相关的这个问题。

如果找出了任何与多个entity bean相关的工作流，那么就可以通过一个session bean而不是entity bean来实现这个工作流。可以利用第7章中的会话门面或者应用服务模式。

- 见第4章中的“合并session bean”重构。
- 见第4章中的“减少entity bean之间的通信”重构。
- 见第4章中的“将业务逻辑移至session bean”重构。
- 见第7章中的会话门面模式。
- 见第7章中的业务对象模式。
- 见第7章中的复合实体模式。
- 见第7章中的应用服务模式。

**51** 对于entity bean中的bean-管理持久化，数据访问代码最好是在entity bean的外部实现。

- 见第4章中的“分离数据访问代码”重构。
- 见第8章中的数据访问对象模式。

## 缓存EJB的远程引用和句柄

当客户端使用一个EJB的时候，它们也许需要缓存对一个EJB的引用，以备将来之用。在使用业务代表（见第7章中的业务代表模式）的时候就会遇到这种情况，因为一个业务代表会连接到一个session bean，并且代替客户端调用该session bean上的业务方法。

当客户端第一次使用业务代表的时候，业务代表需要利用EJB Home对象进行寻址，从而获取session bean的远程引用。但是在其后的请求中，业务代表就可以缓存一个引用或者（如果必要的话）该引用的句柄，这样就避免了多次寻址。

**52** EJB Home的句柄同样可以缓存，这样就避免了为了获取EJB Home对象多次进行Java命名与目录接口（JNDI）寻址。使用EJB句柄或者EJB Home句柄的技术细节可以参照当前的EJB技术规范。

## 业务层和集成层不佳实践

### 把对象模型直接映射为entity bean模型

#### 问题概述

在设计EJB应用系统的时候，常见的一种做法就是直接把对象模型映射为entity bean模型；这也就是说，对象模型中的每个类映射为一个entity bean。结果就会出现许许多多的细粒度的entity bean。

EJB的数量越多，容器和网络的负担就越重。同样，这样一种映射还会把对象之间的关系转化为entity bean之间的关系。最好还是避免这种做法，因为对于远程entity bean，entity bean之间的关系会引入严重的性能问题。

## 参照解决方案

找出对象模型中的父对象-从属对象关系，把它们设计成粗粒度的entity bean。这样entity bean就会少得多，而每个entity bean都由对象模型中的一组相关对象组成。

- **重构** 见第4章中的“减少entity bean之间的通信”。

- **模式** 见第7章中的复合实体模式。

把相关的工作流操作合并到session bean中，从而提供一个统一的、粗粒度的服务访问层。

- **重构** 见第4章中的“合并session bean”。

- **模式** 见第7章中的会话界面模式。

## 把关系型模型直接映射为entity bean模型

### 问题概述

在设计EJB模型的时候，如果把每个表中的每一行设计成一个entity bean，这就会导致一种不佳实践。因为entity bean最好设计为一种粗粒度对象，而这种映射则会产生大量细粒度的entity bean，因此也会影响系统的可扩展性。53

同样，这样的映射还会把表之间的关系（也就是说主键/外键关系）实现为entity bean之间的关系。

## 参照解决方案

按照面向对象的思路设计EJB应用，而不应该仅仅依靠现存的关系型数据库的设计来生成EJB模型。

- **不佳实践** 见本章前面“把对象模型直接映射为entity bean模型”的参照解决方案。

找出对象模型中的父对象-从属对象关系，然后设计出粗粒度的业务对象，从而避免entity bean之间的关系。

- **重构** 见第4章中的“减少entity bean之间的通信”。

- **重构** 见第4章中的“将业务逻辑移至session bean”。

- **模式** 见第7章中的复合实体模式。

## 把每个用例映射为一个session bean

### 问题概述

有些设计者把每个用例实现成一个单独的session bean。这样就产生了很多细粒度的控制器，

每一个只负责一种交互。这种设计方法的缺点是，它会产生大量session bean，从而显著地增加了应用的复杂度。

## 参照解决方案

应用会话门面模式，把一组相关的交互聚合在同一个session bean中。这样应用系统中的session bean就能少得多，同时也充分利用了会话门面模式的优点。

- 重构 见第4章中的“合并session bean”。
- 模式 见第7章中的会话门面模式。

54

## 通过Getter/Setter方法暴露EJB的所有属性

### 问题概述

通过Getter/Setter方法暴露EJB的所有属性是一种不佳实践。这会迫使客户端进行许许多多细粒度的远程调用，从而可能在各系统层次之间产生大量的网络传输。每个方法调用都有转化为远程调用的潜在性，所以每次这样的调用都可能产生网络负担，影响系统的性能和可扩展性。

## 参照解决方案

使用值对象来在客户端和服务器之间传递聚合数据，而不是把每个属性都用getter/setter方法暴露出来。

- 模式 见第7章中的传输对象模式。

## 在客户端中包括服务寻址代码

### 问题概述

客户端和表现层对象常常需要通过寻址访问EJB。在EJB环境中，容器利用JNDI提供这种寻址服务。

把定位服务的负担留给应用程序客户端，会让应用程序客户端的代码中包括大量重复的寻址代码。如果寻址方法发生任何改变，所有需要执行寻址服务的客户端也都要改动。另外，让客户端包括寻址代码，也会把底层实现的复杂度暴露给客户端，并且让客户端依赖于寻址代码。

## 参照解决方案

利用第7章中的服务定位器来封装寻址机制的实现细节。

- 模式 见第7章中的服务定位器。

利用第7章中的业务代表，封装业务层组件（比如session bean和entity bean）的实现细节。这也就能简化客户端代码，因为这样它们就不再需要再跟EJB和服务打交道了。业务代表同样也会

使用到服务定位器。

- 重构 见第4章中的“引入业务代表”。
- 模式 见第7章中的业务代表。

## 把entity bean当成只读对象使用

### 问题概述

entity bean的任何方法都根据部署描述符中确定的事务隔离级别，服从于事务语义。所以，如果把entity bean当成只读对象使用，那就是干脆浪费了这么昂贵的资源，并且还会导致在持久化存储中产生不必要的更新事务。这是因为在entity bean的生命周期中，容器会调用ejbStore()方法。既然在方法调用的过程中，容器没法知道数据是否被更改，所以它只能假定确实发生了更改，并且调用ejbStore()操作。所以，对于只读entity bean和读-写entity bean，容器不加区分。但是，有些容器可能提供了只读entity bean，不过这只是厂商专有的实现。

### 参照解决方案

用第8章中的数据访问对象模式封装对数据源的所有访问。这为数据访问代码提供了一种集中层，并且也简化了entity bean的代码。

- 模式 见第8章中的数据访问对象。

使用session bean实现只读访问功能。通常这被实现为一个利用了DAO的会话门面。

- 模式 见第7章中的会话门面。

为了获取一组传输对象，可以实现第7章中的值列表处理器模式。

- 模式 见第7章中的值列表处理器。

为了从业务层获取复杂数据模型，可以实现第7章中的传输对象模式。

- 模式 见第7章中的传输对象。

## 把entity bean当成细粒度对象使用

### 问题概述

entity bean原本是设计为粗粒度的、有事务的、持久化的业务组件。如果把一个远程entity bean当作细粒度对象使用，就会增加整体的网络通信量和容器的负担。这也会影响应用系统的性能和可扩展性。

应该把细粒度对象看成不与其他对象（通常是一个粗粒度的父对象）关联就没有实际意义的对象。比如说，一个“货品”对象就可以被视为细粒度对象，因为它本身价值不大，除非是与“订单”对象关联起来。在这个例子中，“订单”对象就是粗粒度对象，而“货品”对象就是细粒度的（从属）对象。

## 参照解决方案

在基于一个现存的关系型数据库数据结构（RDBMS schema）设计EJB时，

- **不佳实践** 见本章前面的“把关系型模型直接映射为entity bean模型”。

当利用对象模型设计EJB时，

- **不佳实践** 见本章前面的“把对象模型直接映射为entity bean模型”。

应该设计粗粒度的entity bean和session bean。为了促进粗粒度的EJB设计，可以采用以下模式和重构。

- **模式** 见第7章复合实体。
- **模式** 见第7章会话门面。
- **重构** 见第4章“减少entity bean之间的通信”。
- **重构** 见第4章“将业务逻辑移至session bean”。
- **重构** 见本章前面的“entity bean中的业务逻辑”。
- **重构** 见第4章“合并session bean”。

## 存储entity-bean的整个从属对象拓扑结构

### 问题概述

当一个entity bean的从属对象拓扑结构非常复杂时，如果整个地装载并存储从属对象树，那么系统性能会发生急剧的下降。无论是首次装载数据，还是重新装载数据以实现与持久化存储的同步，当容器调用entity bean的ejbLoad()方法时，装载整个从属对象树都是浪费的。同样，无论何时，当容器调用entity bean的ejbStore()方法时，存储整个对象树也是相当昂贵的和不必要的。57

## 参照解决方案

确定上次存储操作执行以来，哪些从属对象发生了改动，然后仅仅把这些对象保存到持久化存储中。

- **模式** 见第7章中的复合实体模式及存储优化（脏数据标示器）策略。

实现一种策略，只装载最常访问、请求的数据。按照需求随时装载其他的从属对象。

- **模式** 见第7章中的复合实体模式及懒装载策略。

应用了这些策略，就有可能避免整个地装载、保存从属对象树。

## 把EJB相关的异常暴露给非EJB客户端

### 问题概述

EJB可以向客户端抛出业务应用异常。当应用系统抛出一个应用异常的时候，容器就简单地把异常抛给了客户端。这样客户端就能优雅地处理异常，并且可能会执行另外一个操作。大多

数应用开发者想必都能理解、处理这样的应用级别的异常。

设计并使用应用异常是一种良好的编程实践。但是，即使遵从了这一实践，客户端还是会接到与EJB相关的异常，比如`java.rmi.RemoteException`。例如，当EJB或容器遇到了与EJB相关的系统故障时，就会抛出这样的异常。

有些应用开发者甚至可能都不太了解EJB异常和相关的语义，但是既然业务层组件抛出了这样的非应用异常，应用开发者还是得挑起重担，搞明白这些异常的细节内容。而且，这些非应用异常还可能根本不提供帮助用户处理问题的相关信息。

## 参照解决方案

采用业务代表，消除客户端和业务层之间的耦合，对客户端隐藏业务层的实现细节。业务代表拦截了所有服务异常，并可能会抛出应用异常。所谓业务代表，是一种普通Java对象，和客户端处于同一物理位置。通常，业务代表由EJB开发者编写，提供给客户端开发者。

- **重构** 见第4章中的“引入业务代表”。
- **模式** 见第7章中的“业务代表”。

## 使用entity bean finder方法返回大型结果集

### 问题概述

应用系统常常需要具备搜索和获取一系列数值的能力。使用EJB finder方法查找大量entity bean，就会返回一组远程引用。此后，客户端还必须对每个远程引用调用方法，才能获得数据。这属于远程调用，因此当调用者对那一组远程entity bean引用中的每一个都进行远程调用时，代价就特别昂贵，尤其会影响系统性能。

## 参照解决方案

利用session bean和DAO来实现查询，获取一组传输对象，而不是远程引用。利用DAO而不是EJB finder方法来执行查询。

- **模式** 见第7章中的“值列表处理器”。
- **模式** 见第8章中的“数据访问对象”。

## 客户端负责聚合来自业务组件的数据

### 问题概述

应用客户端（无论是在实际客户端中还是在表现层）通常为了业务应用，需要来自业务层的数据模型。但是，由于数据模型是由业务组件（包括entity bean、session bean以及业务层的各种对象）实现的，客户端就要定位这种种组件、与它们交互、并从中获得必要的数据，这样才

能构造出数据模型。

由于客户端需要调用业务层的多个方法，所以客户端的操作会导致网络传输的负担。另外，客户端也与应用模型产生了紧耦合。当应用系统中存在多种类型的客户端时，这种耦合问题还会加倍：对模型做一处修改，与业务组件构成的这些模型交互的所有客户端就都要做相应的修改。  
59

## 参照解决方案

解除客户端和模型构造之间的耦合。实现一个业务层组件，让它来负责构造应用系统需要的数据模型。

- **模式** 见第7章中的传输对象组装器。

## 把EJB用于长时间持续的事务

### 问题概述

EJB（在EJB2.0版之前）主要适用于同步处理。而且，当EJB的每个方法实现都能在一段可以预估并能够接受的时间范围内给出结果，那么EJB就最能发挥其特长。

如果为了处理客户端请求，EJB的某个方法需要相当长的一段时间，或者，如果在处理过程中它会发生阻塞，那么它也就阻塞了该EJB使用着的容器资源，比如内存和线程。这将严重地影响系统性能，耗尽系统资源。

所以，如果一个EJB事务需要很长时间才能完成，它就会锁住其他EJB实例需要的资源，从而造成性能瓶颈。

## 参照解决方案

使用支持Java消息服务（JMS）API的消息中间件（message-oriented middleware，MOM）实现异步处理服务，用以实现长时间持续的事务。

- **模式** 见第8章中的服务激活器。

## 每次调用无状态session bean都要重建对话状态

### 问题概述

有些设计者为了增进系统的可扩展性选用无状态session bean。但是，他们也许不经意地决定把所有的业务流程都用无状态session bean实现，即使session bean需要对话状态时也如此。由于采用了无状态session bean，每个方法调用时都需要重建对话状态。而状态可能还要靠从数据库获取。这完全违背了使用无状态session bean来提高性能和可扩展性的初衷，而且可能严重地降低系统性能。  
60

## 参照解决方案

在选择无状态session bean模式之前先分析一下系统的交互模型。有时候，既需要在多个方法调用之间（利用有状态session bean）维护对话状态，而每次调用时（为无状态session bean）重建状态又要花不小代价；选择有状态session bean还是无状态session bean，就决定于这些考虑。

• 模式 见第7章中的传输对象组装器、无状态会话门面策略和有状态会话门面策略。

• 设计考虑 见本章前面的“session bean——无状态vs.有状态”和“在业务层保存状态”。



## 第4章 J2EE重构

本章将涉及下列主题：

- 表现层的重构
- 业务层和集成层的重构
- 一般的重构

62

63

## 表现层的重构

这里所介绍的重构是用于表现层的。

### 引入控制器

控制逻辑散布于整个应用程序，并经常在多个JavaServer Page (JSP) 视图中重复出现（见图4-1）。将控制逻辑抽取至一个或多个控制器类，使其作为处理客户端请求时的第一个接触点。

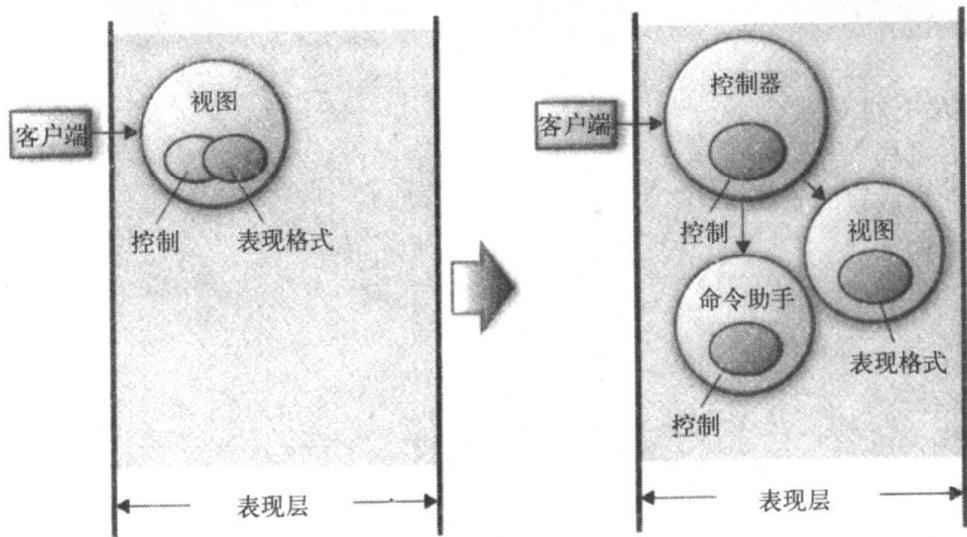


图4-1 引入控制器

### 动机

如果“负责逻辑控制”的代码在多个JSP中重复出现，一旦控制逻辑发生改变，就必须到每个JSP中去维护这些代码。将它们抽取到一个或多个集中的控制器类，能够提高应用程序的模块化程度、可复用性和可维护性。

### 作法

- 以前端控制器模式作为指引，用提炼类（Extract Class）[Fowler]<sup>⊖</sup> 的重构手法创建一个控制器类，将各个JSP中重复的控制逻辑搬入这个控制器。
  - ⇒ 参见前端控制器。
  - ⇒ 请记住，控制器是一个中转站：它判断“应该如何处理客户端请求”，并将请求委派给合适的业务组件。在引入控制器时，你应该对代码进行合理的划分，目的是模块化和复用。不必将所有控制代码硬塞进一个控制器，可以考虑创建一些助手组件，让控制器将部分任务转交给它们。参见第2章的不佳实践——创建出“胖控制器”。
- 也可以用命令（Command）模式[GoF]将控制代码封装成一个命令对象（command object），

<sup>⊖</sup> 所有注明[Fowler]的重构，都出自Martin Fowler的名著《重构》(Refactorings)。见参考书目。

使其与控制器协同工作。

⇒ 参见前端控制器，“命令加控制器策略”。

⇒ 参见应用控制器，“命令处理器策略”。

## 示例

假想如例4.1的这样一段代码结构——它出现在我们的很多个JSP中。

### 例4.1 引入控制器——重构之前的JSP结构

```

1  <HTML>
2  <BODY>
3  <control:grant_access/>
4  .
5  .
6  .
7  </ BODY>
</ HTML>
```

代码中的三个点代表各个JSP的主体部分，它们与本例的主题无关，所以在此略过。虽然每个JSP的主体部分各不相同，但在页面顶端，以定制标记的方式实现的助手却是相同的：它负责控制页面的访问权限。这是一个“全有或全无”类型的控制，也就是说，客户端要么被授权访问整个页面，要么被彻底拒绝。

如果我们对设计进行修改，引入一个控制器（就像“作法”中所介绍的那样），那么我们的所有JSP将不再含有例4.1中的<control:grant\_access/>标记，取而代之的是一个集中的控制器，后者将负责处理访问控制检查的工作。例4.2就是从控制器类抽取的一小段代码，在这里控制器被实现为一个servlet。65

### 例4.2 引入控制器——重构之后的控制器结构

```

1
2  if (grantAccess())
3  {
4      dispatchToNextView();
5  }
6  else
7  {
8      dispatchToAccessDeniedView();
9 }
```

当然，在某些场合，“嵌入JSP的助手标记”也能够胜任“承载控制代码”的职责。譬如说，如果只有少数几个JSP需要这样的访问控制，那么完全可以在这几个页面中加入定制标记助手，以此达到目的。另外，如果需要为一个复合视图（参见复合视图）中特定的子视图进行独立的访问控制，这也可以说成为“在JSP中使用定制标记”的另一个理由。

如果此前已经在使用控制器，我们仍然可能希望将访问控制的行为放置到一个集中的地方统一管理，因为需要控制的页面数目可能会与日俱增。在“已经拥有控制器”的情况下，只需将控制代码从视图中抽取出来，并添加到现有的控制器中就可以了。这时，我们其实是在迁移方法（使用迁移方法（Move Method）[Fowler]的重构手法），而非提取出一个新的类。66

## 引入同步器令牌

客户端对需要监控的资源发出重复请求，或者直接回到以前做过书签的页面，而不是按照你所希望的顺序访问特定视图。

用一个共享令牌来监控请求流程及客户端对特定资源的访问（见图4-2）。

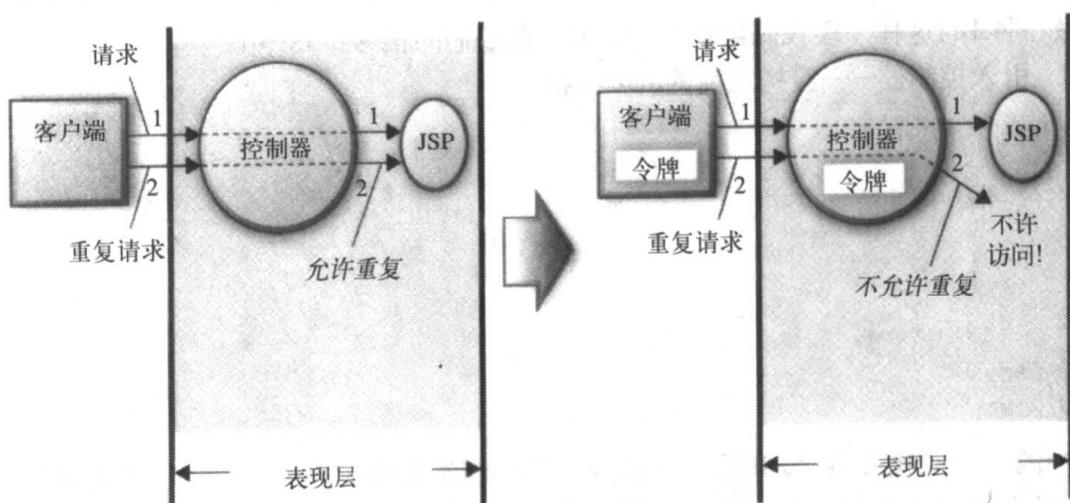


图4-2 引入同步器令牌

### 动机

存在几种情况需要控制进入服务器的请求。最常见一个理由就是：希望控制客户端的请求提交，不允许重复提交同一个请求。这种重复提交通常是因为使用者用浏览器的“后退”或“停止”按钮阻止了页面切换，然后再次提交表单。

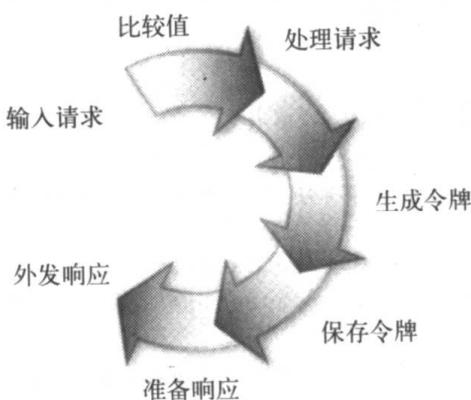
### 作法

- 创建一个或多个助手类，用于生成一次性的、唯一的令牌，并负责对令牌进行比对。
  - ⇒ 另外，也可以将上述逻辑加入到现有的控制组件中。
  - ⇒ 负责处理提交操作的组件（通常是控制器，但也可能是一个JSP）将请求委派给助手，后者将针对每次客户端提交新建一个令牌，并将其暂存起来。

令牌的副本会分别保存在服务器端和客户端浏览器上。在客户端，令牌通常被保存在表单的隐藏字段中；而在服务器端，针对每个用户保存令牌的理想位置则是用户会话(session)。

#### 设计手记：何时生成并保存一个令牌？

何时对令牌进行检查？在处理一个刚到达的请求之前，需要先对同步器令牌进行比对。在处理请求之后、将响应准备好并发送至客户端之前，需要生成并保存一个新的令牌值。更多的信息请参阅本章的“引入同步器令牌”重构。



图示：同步器令牌的生命周期

- 加入检查逻辑，判断随客户端请求一起到达的令牌是否与用户会话中保存的令牌匹配。
    - ⇒ “客户端当前请求中携带的令牌”与“服务器在前一次响应时发送给客户端的令牌”应该是同一个。如果这两个令牌相匹配，就说明当前的请求不是重复提交；而如果两者不匹配，就有可能是客户端进行了重复提交。
    - ⇒ 正如前文所暗示的，“两个令牌不匹配”也可能是由其他原因所导致的，例如使用者通过书签直接访问了某一页面。不过，“重复提交请求”是最常见的原因。（更多信息请参阅第2章，“表现层设计考虑”中的“控制客户端访问”）。
  - 通常由控制器来负责令牌的生成和比对。如果没有现成的控制器，应该考虑引入一个。
    - ⇒ 参见本章的“引入控制器”重构。
    - ⇒ 如果没有控制器来集中管理令牌的生成和比对，就必须在每个JSP中引用这一行为。
    - ⇒ JSP通常将这一行为委派给一个助手组件，后者将对“令牌管理”的职责进行封装。这个助手组件的实现机制可以是JavaBean或者定制标记（参见“应用控制器”模式）。
- 本节所列出的代码片段遵循Apache软件授权协议1.1版，该协议的条文参见本书附录。

68

## 示例

作为一个表现层框架，Struts应用了一些J2EE模式和重构手法。Struts使用的请求流程控制技术恰好与我们在这里所介绍的相同，因此我们从这个开源框架中选取一些代码片段作为示例。Struts并没有创建一个单独的工具类用于封装“生成、比对令牌”的逻辑，而是将这些功能作为整个控制机制的一部分添加到已有的类中——这个类就是Action（操作），所有操作的共同超类。在Struts的概念中，“操作”实现控制器的功能；但还不仅限于此，它同时也是一个命令对象：这正是对前端控制器模式中“命令加控制器”策略的应用。正如例4.3所示，Action类中的saveToken()方法负责生成并保存令牌的值。

### 例4.3 生成并保存令牌

```

1
2  /**
3  * 在用户的当前会话中

```

```

4   * 保存一个新的事务令牌，如果必要
5   * 则需首先创建一个新会话
6   *
7   * @param request 处理中的servlet请求
8   */
9  protected void saveToken(HttpServletRequest request) {
10
11 HttpSession session = request.getSession();
12 String token = generateToken(request);
13 if (token != null)
14     session.setAttribute(TRANSACTION_TOKEN_KEY, token);
15 }

```

Copyright (c) 1999 The Apache Software Foundation. All rights reserved.

69

这个方法根据会话ID和当前时间生成一个唯一的令牌，并将其保存在用户会话中。

如果我们希望向用户显示一个HTML页面（这个页面通常会包含一个表单，让用户填写之后传回服务器），并且不希望用户在这个页面上重复提交请求，那么在生成要传到客户端的HTML代码之前需要首先调用saveToken方法，将一次性的令牌值保存到会话中：

```
saveToken(request);
```

另外，负责最终生成HTML展示的JSP还需要委派一个助手类来生成一个隐藏字段，在其中容纳令牌的值。于是，客户端最终收到的页面（其中通常包含一份表单，用户可以将其提交回服务器）中将带有这样一个隐藏字段：

```
<input type="hidden" name="org.apache.struts.taglib.html.TOKEN" value =
"8d2c392e93a39d299ec45a22">
```

在这个隐藏字段中，value属性记录的就是由saveToken()方法所生成的令牌值。

当客户端提交含有上述隐藏字段的页面时，控制器将委派一个命令对象（如前所述，它是Action类的子类）来比较“用户会话中的令牌值”和“请求对象的参数所携带的、来自页面隐藏字段的令牌值”。命令对象将使用其超类（Action类）提供的isTokenValid方法对两个值进行比较，例4.4列出了这个方法的代码片段。

#### 例4.4 检查令牌合法性

```

1
2 /**
3  * 如果用户当前会话中
4  * 保存着一个事务令牌，而且操作请求的参数中，
5  * 提交的值与令牌值匹配。
6  * 则返回 <code>true</code>。
7  *
8  * 在以下情况下，
9  * 返回<code>false</code>：
10 * <ul>
11 * <li>没有会话与此请求关联</li>
12 * <li>会话中没有保存事务令牌</li>
13 * <li>请求中没有包括事务令牌

```

70

```

14  * 参数</li>
15  * <li>请求中包括的事务令牌值
16  *   与用户会话中的事务令牌
17  *   不匹配</li>
18  * </ul>
19  *
20  * @param request 处理中的servlet请求
21  */
22
23 protected boolean isTokenValid(HttpServletRequest request) {
24
25     // 获取会话中保存的
26     // 事务令牌
27     HttpSession session = request.getSession(false);
28     if (session == null)
29         return (false);
30     String saved = (String)
31         session.getAttribute(TRANSACTION_TOKEN_KEY);
32     if (saved == null)
33         return (false);
34     // 获取请求中包括的
35     // 事务令牌
36
37     String token = (String)
38         request.getParameter(Constants.TOKEN_KEY);
39     if (token == null)
40         return (false);
41
42     // 值是否匹配?
43     return (saved.equals(token));
44 }

```

Copyright © 1999 The Apache Software Foundation. All rights reserved.

如果两个令牌值相匹配，我们就可以断定：本次请求不是重复的提交；如果两者不相匹配，我们则可以选择合适操作来处理可能的重复提交。

71

## 隔离不同逻辑

业务逻辑与表现格式混在同一个JSP视图中。

将业务逻辑抽取至一个或多个助手类，并由JSP或控制器来调用它们。

## 动机

我们希望在应用程序中建立更为清晰的抽象，提高其内聚性，降低耦合度，以获得更高的模块化程度和可复用性。对于一个划分合理、模块化的应用程序，可以更好地区分开发者的角

色：Web开发者负责表现格式，而软件开发者则只需关心业务逻辑。

## 作法

- 以视图助手模式作为指引，使用提炼类（Extract Class）[Fowler]的重构手法新建助手类，并将业务代码从JSP移至助手类中。
- 在JSP中委派这些助手类完成业务操作。
  - ⇒ 参见视图助手模式。
  - ⇒ 正如图4-3的“后置分解”图所示，视图可以作为处理客户端请求的第一个接触点。参见分配器视图模式。
- 如果没有现成的控制器，可以考虑引入一个。
  - ⇒ 参见本章的“引入控制器”重构手法。
  - ⇒ 正如图4-4所示，控制器可以使用一个命令助手。
  - ⇒ 如上面的“前置分解”图所示，控制器也可以作为处理客户端请求的第一个接触点。参见服务到工作者模式。

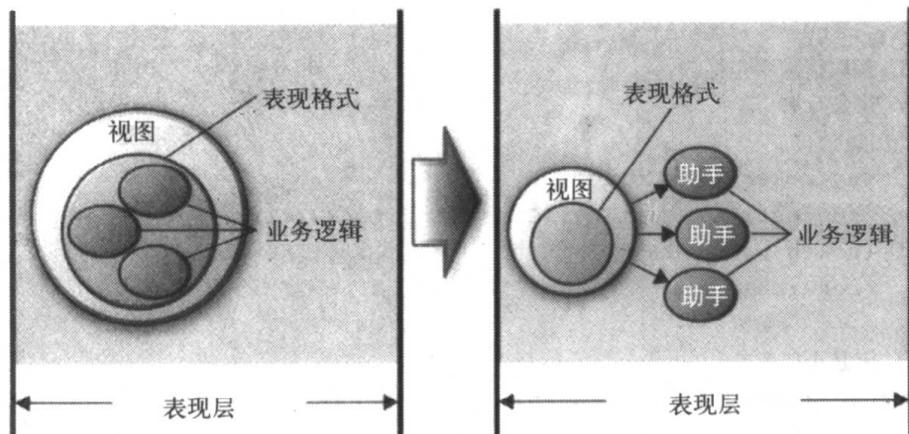


图4-3 隔离不同逻辑：后置分解

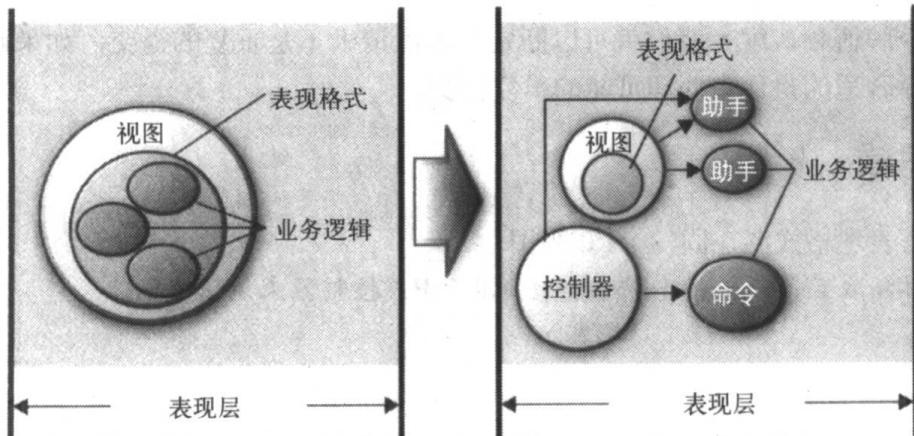


图4-4 隔离不同逻辑：前置分解

## 示例

首先，请看例4.5列出的代码。这是一个JSP，其中有大量的scriptlet代码，业务逻辑和视图逻辑混淆在一起。

### 例4.5 含有scriptlet代码的JSP

```

1   <html>
2   <head><title>Employee List</title></head>
3   <body>
4   <%-- Display All employees belonging to a department and earning
5   at most the given salary --%>
6
7   <%
8
9       // 获取员工将在其中列出的
10      // 部门
11      String deptidStr = request.getParameter(
12          Constants.REQ_DEPTID);
13
14      // 获取最高薪酬限制
15      String salaryStr = request.getParameter(
16          Constants.REQ_SALARY);
17
18      // 验证参数
19
20      // 如果薪酬或部门没有给出，则转向
21      // 出错页面
22      if ( (deptidStr == null) || (salaryStr == null) )
23      {
24          request.setAttribute(Constants.ATTR_MESSAGE,
25              "Insufficient query parameters specified" +
26              "(Department and Salary)");
27          request.getRequestDispatcher("/error.jsp").
28              forward(request, response);
29      }
30
31      // 把字符转换成数字
32      int deptid = 0;
33      float salary = 0;
34      try
35      {
36          deptid = Integer.parseInt(deptidStr);
37          salary = Float.parseFloat(salaryStr);
38      }
39      catch(NumberFormatException e)
40      {

```

```

41         request.setAttribute(Constants.ATTR_MESSAGE,
42             "Invalid Search Values" +
43             "(department id and salary )");
44         request.getRequestDispatcher("/error.jsp").
45             forward(request, response);
46
47     }
48
49     // 检查数字是否在合法限制内
50     if ( salary < 0 )
51     {
52         request.setAttribute(Constants.ATTR_MESSAGE,
53             "Invalid Search Values" +
54             "(department id and salary )");
55         request.getRequestDispatcher("/error.jsp").
56             forward(request, response);
57     }
58
59 %>
60
61 <h3><center> List of employees in department # <%=deptid%>
62   earning at most <%= salary %>. </h3>
63
64 <%
65   Iterator employees = new EmployeeDelegate().
66       getEmployees(deptid);
67 %>
68
69 <table border="1" >
70   <tr>
71     <th> First Name </th>
72     <th> Last Name </th>
73     <th> Designation </th>
74     <th> Employee Id </th>
75     <th> Tax Deductibles </th>
76     <th> Performance Remarks </th>
77     <th> Yearly Salary</th>
78   </tr>
79 <%
80   while ( employees.hasNext() )
81   {
82     EmployeeVO employee = (EmployeeVO)
83     employees.next();
84
85     // 如果满足搜索条件, 则显示
86     if ( employee.getYearlySalary() <= salary )
87     {

```

```

88  %>
89      <tr>
90          <td> <%=employee.getFirstName()%></td>
91
92          <td> <%=employee.getLastName()%></td>
93          <td> <%=employee.getDesignation()%></td>
94          <td> <%=employee.getId()%></td>
95          <td> <%=employee.getNoOfDeductibles()%></td>
96          <td> <%=employee.getPerformanceRemarks()%>
97              </td>
98          <td> <%=employee.getYearlySalary()%></td>
99      </tr>
100 <%
101     }
102   }
103 %>
104 </table>
105
106 <%@ include file="/jsp/trace.jsp" %>
107 <P> <B>Business logic and presentation formatting are
108     intermingled within this JSP view. </B>
109
110 </body>
111 </html>

```

这个JSP将生成一个HTML表格，在其中列举“属于某一薪酬水平”的员工信息。这个JSP同时封装了表现格式和业务逻辑，如图4-5所示。

于是，我们首先按照视图助手模式的要求修改设计，将JSP视图中的scriptlet代码抽取出来，如例4.6所示。

#### 例4.6 抽出了scriptlet代码之后的JSP

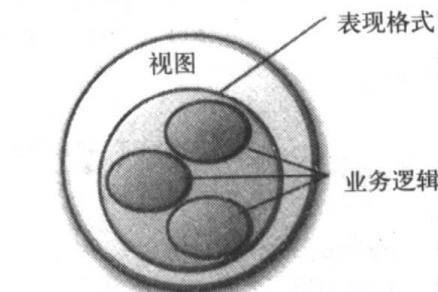


图4-5 混合了业务逻辑和表现格式的视图

```

1  <%@ taglib uri="/WEB-INF/corepatternstaglibrary.tld"
2      prefix="corepatterns" %>
3
4  <html>
5  <head><title>Employee List</title></head>
6  <body>
7
8  <corepatterns:employeeAdapter />
9
10 <h3><center>List of employees in
11   <corepatterns:department attribute="id"/>
12   department - Using Custom Tag Helper Strategy </h3>
13
14 <table border="1" >
15     <tr>

```

```

16      <th> First Name </th>
17      <th> Last Name </th>
18      <th> Designation </th>
19      <th> Employee Id </th>
20      <th> Tax Deductibles </th>
21      <th> Performance Remarks </th>
22      <th> Yearly Salary</th>
23  </tr>
24  <corepatterns:employeelist id="employeelist_key">
25  <tr>
26      <td><corepatterns:employee
27          attribute="FirstName"/></td>
28      <td><corepatterns:employee
29          attribute="LastName"/></td>
30      <td><corepatterns:employee
31          attribute="Designation"/> </td>
32      <td><corepatterns:employee
33          attribute="Id"/></td>
34      <td><corepatterns:employee
35          attribute="NoOfDeductibles"/></td>
36      <td><corepatterns:employee
37          attribute="PerformanceRemarks"/></td>
38      <td><corepatterns:employee
39          attribute="YearlySalary"/></td>
40      <td>
41  </tr>
42  </corepatterns:employeelist>
43 </table>
44
45 </body>
46 </html>

```

此外，我们还编写了两个定制标签助手，分别用于封装业务逻辑和“将数据模型填入HTML表格”

77 表现格式处理逻辑。这两个助手是<corepatterns:employeelist>标签和<corepatterns: employee>标签。

在图4-6中可以看到，我们已经把箭头左边的设计变成了右边这样。

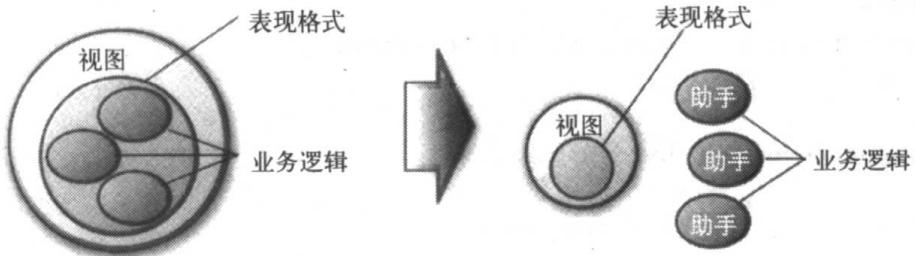


图4-6 将业务逻辑抽取至助手类

现在，业务逻辑不再直接嵌入JSP，它们已经被抽取至助手类。这些助手类各自完成不同的任务，包括内容获取、访问控制以及根据显示需要对模型进行转化等。在完成第三项任务（根

据显示需要对模型进行转化)时,助手类实际上封装了一部分表现逻辑,例如“将结果集格式化为HTML表格”(参见本章“去除视图中的转换”重构手法)。应该让JSP直接向助手类要求完整的HTML表格,而不是在JSP中嵌入scriptlet代码以生成表格,这有助于达到我们的目标:尽可能地将程序逻辑抽离视图。

助手组件的实现机制可以是JavaBean或者定制标记(参见应用控制器模式)。以JavaBean形式实现的助手适合用于封装“获取内容”、“保存结果”之类的逻辑,而定制标记助手则更擅长前面提到的“根据显示需要对模型进行转化”——例如用结果集中的数据创建HTML表格。不过,两者的效用多少有些重叠,因此最终决定“如何实现助手”的可能会是别的因素,例如开发者的经验、程序可控性的考量等等。

现在,该射出我们的第二枚子弹了:将业务操作委派给助手,如图4-7所示。

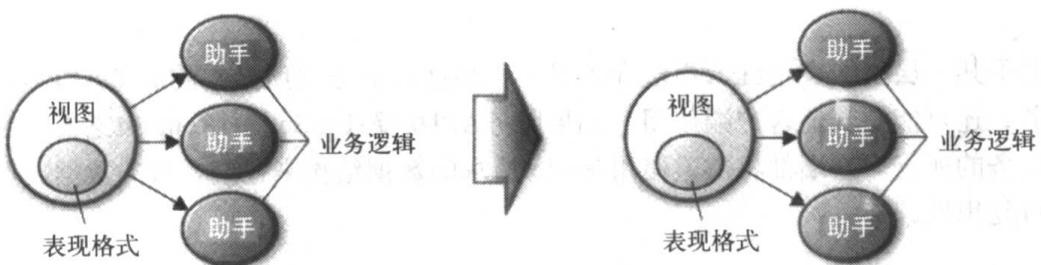


图4-7 将工作委派给助手

78

JSP视图委派助手类来处理并生成视图。一般来说,我们会在JSP之前放置一个控制器,作为客户端请求的第一接触点(参见前端控制器模式和本章的“引入控制器”重构手法)。控制器会将请求分配给视图,但在此之前,它也可以首先委派一些工作给助手组件(参见服务到工作者模式)。

引入控制器之后,我们的设计就变成了图4-8这样。

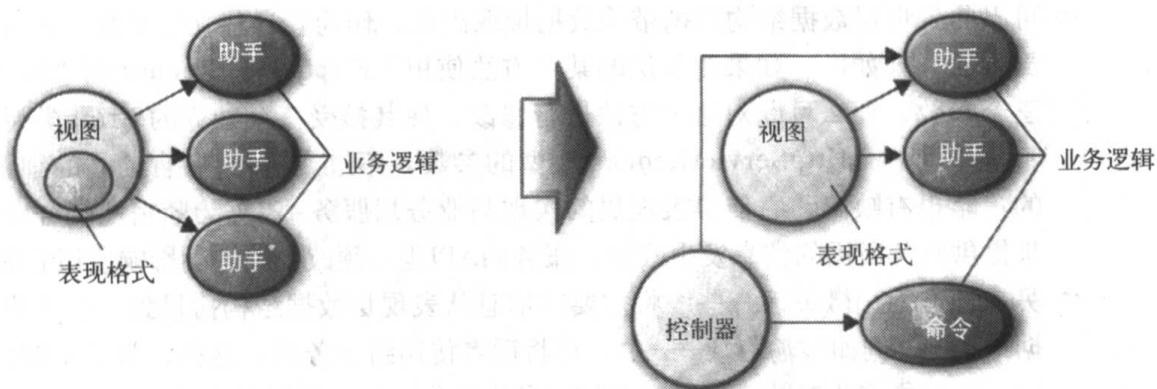


图4-8 引入控制器

79

## 对业务层隐藏表现细节

对用户请求的处理和/或与通信协议相关的数据结构被表现层暴露给业务层。

将所有涉及“处理用户请求”和“与协议相关的表现层数据结构”的代码从业务层中去掉。使用更为通用、与表现层无关的数据结构在这两层之间传递值(见图4-9)。

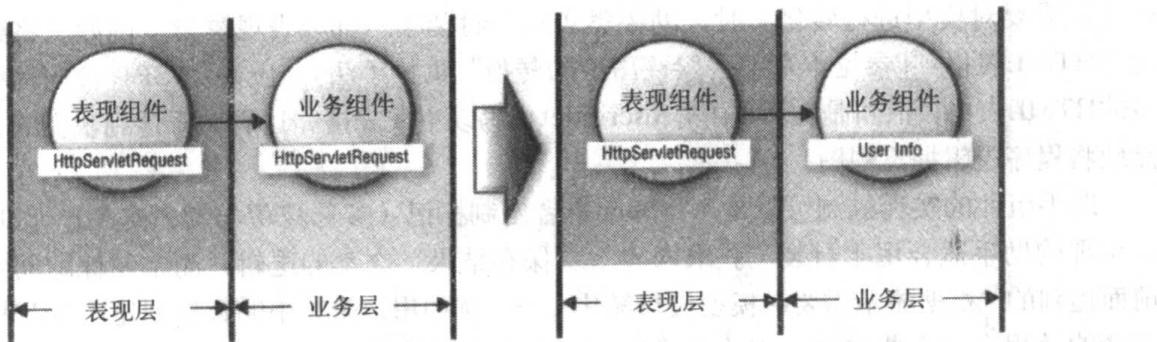


图4-9 对业务层隐藏表现细节

## 动机

特定于某一层次的实现细节不应介入另一个层次。业务层所暴露的服务API不仅要向表现层提供服务，还可能被其他客户端使用。如果服务API接受HttpServletRequest之类的参数，那么使用这些服务的所有客户端都将被迫使用servlet请求的数据结构来包装各自的数据，这将严重降低服务的可复用性。

## 作法

- 将业务层中所有涉及表现层数据结构的地方替换成更通用的数据结构类型。
  - ⇒ 如果业务层的方法接受的参数类型是HttpServletRequest之类特定于表现层的类型，将其替换成更通用的参数类型，例如String、int或者UserInfo。
- 对于表现层中调用上述业务方法的地方，作相应的调整。
  - ⇒ 可以将表现层数据结构所携带的数据抽取出来，作为各自独立的参数一起传递给业务层方法。譬如说，如果业务层的某个方法使用了HttpServletRequest所携带的x、y和z三个参数，那么可以对这个方法进行修改，使其接受三个独立的String类型参数，而不是接受一个HttpServletRequest类型的参数。不过，“传递各自独立的细粒度参数”的策略也有缺陷，它使得表现层的实现与业务层服务API更为紧密地耦合在一起：如果提供服务所需的信息发生改变，服务的API也必须改变，进而影响使用它的客户端。
  - ⇒ 另一种方案稍微灵活一些：将需要的信息从表现层数据结构拷贝到一个更为通用的数据结构——例如传输对象——中，再将后者传递给业务层。这样，服务API将始终接受同一个对象作为参数。即便实现细节发生变化，接口也无须变动。
- 另外，如果使用了表现层框架（例如当下流行的Struts框架[Struts]），也可以考虑覆盖该框架所提供的接口类型。
  - ⇒ 在处理用户请求时，表现层框架通常会创建一套自己的数据结构。譬如说，框架通常会把HttpServletRequest数据结构所携带的相关信息拷贝到一个更为通用的、专属于该框架的数据类型中，这个过程对于程序员是完全透明的。这个数据类型扮演的角色就是传输对象，但它毕竟是特定于框架的。如果直接将其传递给业务层，会使业务服务与处理用户请求的框架发生耦合。

- ⇒ 自然，也还可以采用前面所介绍的办法：将信息从这个特定于框架的数据结构再拷贝到一个更通用的数据结构，然后将后者传递给业务层。不过，还有一种更经济的办法：创建一个通用的接口类型，使其具有和“特定于框架的数据结构”相同的接口方法。然后，只要把新建的接口类型覆盖到特定于框架的对象上，就可以将这个对象以新建接口的类型传递给业务层。这样，无须手工复制信息，业务层也不会与框架发生耦合。
- ⇒ 举例来说，假设框架提供了a.framework.StateBean接口，应用程序中的my.stuff.MyStateBean实现了该接口。如果框架实例化MyStateBean，得到的对象将是StateBean类型的。

81

**注意：**实例的创建通常是通过工厂完成的，实例化代码可能如下所示。简单起见，这里没有列举出参数。

```
a.framework.StateBean bean = new my.stuff.MyStateBean(...);
```

- ⇒ 如果业务层直接接受这个bean作为参数，那么参数类型将是StateBean。

```
public void aRemoteBizTierMethod(a.framework.StateBean bean)
```

- ⇒ 为了避免业务层与特定框架发生耦合，我们不直接传递类型为StateBean的参数，而是新建一个名为my.stuff.MyStateVO的接口，并在my.stuff.MyStateBean中实现该接口。

```
public class MyStateBean extends a.framework.StateBean implements MyStateVO
```

- ⇒ 现在，业务层的方法签名变成了这样：

```
public void aRemoteBizTierMethod(my.stuff.MyStateVO bean)
```

- ⇒ 于是就不需要将参数拷贝到更为通用的传输对象中，而表现层框架特有的类型也不会被暴露给其他层次。

□ 最后，应该提醒读者：也可以对表现层的业务领域对象使用同样的重构手法，以降低逻辑上不相关联的各个部分之间的耦合度（如图4-10所示）。

- ⇒ 前面的“动机”和“作法”完全适用于这里的情形，因为我们不希望降低基本业务领域对象（例如Customer（客户）对象）的可复用性。

- ⇒ 这一重构将“与协议相关”的数据结构局限在“处理请求的组件”（例如控制器）范围内。在“示例”一节的例4.7和例4.8中，你可以看到一个“消除业务领域对象与HttpServletRequest之间耦合”的例子。

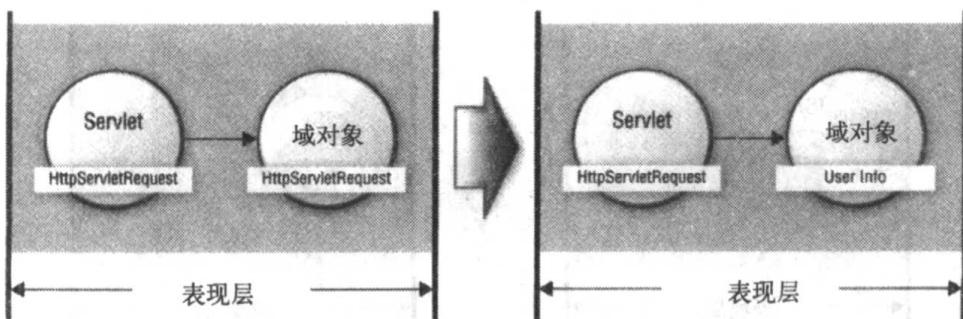


图4-10 对业务领域对象隐藏表现细节

82

## 示例

例4.7中的Customer类接受HttpServletRequest实例作为其构造函数的参数，这严重降低了它作为业务领域对象应有的通用性。如果一个非web的客户端想要使用Customer（客户）类，它必须首先通过某种途径生成一个HttpServletRequest对象，这是极不得体的。

### 例4.7 业务领域对象与HttpServletRequest对象之间存在紧密的耦合

```

1  /**
2  * 下面的代码片段给出了一个与HttpServletRequest紧耦合的业务领域对象 */
3  public class Customer
4  {
5      public Customer ( HttpServletRequest request )
6      {
7          firstName = request.getParameter("firstname");
8          lastName = request.getParameter("lastname");
9      }
10 }

```

我们不应向一个通用的Customer对象暴露HttpServletRequest这样一个特定于web环境的对象，应该将它们解耦，如例4.8所示。

### 例4.8 消除业务领域对象与HttpServletRequest对象之间的耦合

```

1 // 与HttpServletRequest没有耦合的业务对象
2 public class Customer
3 {
4     public Customer ( String first, String last )
5     {
6         firstName = first;
7         lastName = last;
8     }
9 }
10 }

```

83

## 去除视图中的转换

为满足显示的需要，在视图组件中对模型的部分数据进行转换。

将所有转换代码抽离视图组件，将其封装至一个或多个助手类（见图4-11）。

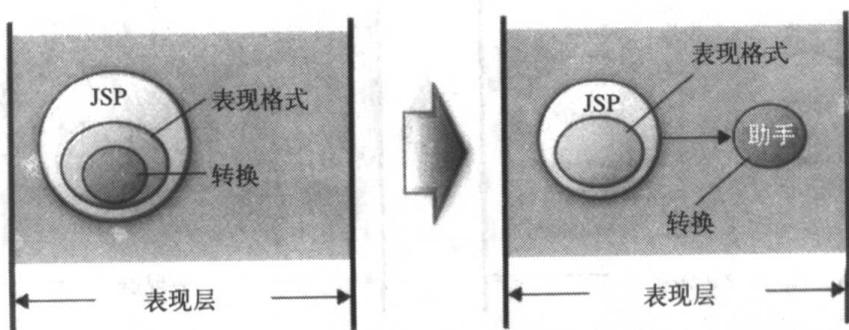


图4-11 去除视图中的转换

## 动机

如果直接在JSP视图中嵌入“对模型进行转换以便显示”的逻辑，会降低应用程序的模块化程度和可复用性。这样的转换很可能在多个JSP中出现，于是不得不以“拷贝-粘贴”的方式“复用”它们——重复代码是最令维护者头疼的。

## 作法

- 使用提炼类（Extract Class）[Fowler]的重构手法，将“对模型进行转换和调整”的逻辑从各个JSP中抽取至助手类。
  - ⇒ 一个例子：用一些代码将数据库结果集填入HTML表格。
- 在JSP中调用助手类的方法，完成数据的转换和调整。
  - ⇒ JSP把请求委派给助手组件，实际的转换操作由后者完成。

## 示例

在这个例子中，我们面对的是这样一个逻辑：将一组元素（例如一个结果集）转换成一张HTML表格。84

在某种意义上，这也可以看作一个显示格式逻辑，不过它终归还是执行转换的代码——从一个中间模型生成最终的表格。如果将动态转换的实现封装在定制标记中，而不是直接嵌在JSP里，我们就可以复用它。

例4.9是一个JSP的例子，上述转换逻辑被直接嵌在它的源码中。

### 例4.9 嵌入了转换逻辑的视图

```

1 <html>
2   <head><title>Employee List</title></head>
3   <body>
4
5   <h3><head><center> List of employees</h3>
6
7
8   <%
9     String firstName =
10    (String)request.getParameter("firstName");
11    String lastName =
12    (String)request.getParameter("lastName");
13    if ( firstName == null )
14      // 如果没有给出，则设置为空
15      firstName = "";
16    if ( lastName == null )
17      lastName = "";
18
19    EmployeeDelegate empDelegate = new
20      EmployeeDelegate();

```

```

21     Iterator employees =
22         empDelegate.getEmployees(
23             EmployeeDelegate.ALL_DEPARTMENTS);
24     %>
25
26 <table border="1" >
27     <tr>
28         <th> First Name </th>
29         <th> Last Name </th>
30         <th> Designation </th>
31     </tr>
32 <%
33     while ( employees.hasNext() )
34     {
35
36         EmployeeVO employee = (EmployeeVO)
37                         employees.next();
38
39         if ( employee.getFirstName().
40             .startsWith(firstName) &&
41             employee.getLastName().
42             .startsWith(lastName) ) {
43     %>
44     <tr>
45         <td><%=employee.getFirstName().toUpperCase() %></td>
46         <td> <%=employee.getLastName().toUpperCase() %></td>
47         <td> <%=employee.getDesignation()%></td>
48     </tr>
49 <%
50         }
51     }
52 %>
53 </table>

```

首先，我们需要把转换逻辑抽取到助手类中。在这里，定制标记助手是最合适的，因为我们希望尽量去掉JSP中的scriptlet代码。然后，我们将修改JSP，使其委派助手标记完成转换处理。例4.10展示了经过这两步重构之后的JSP。

#### 例4.10 转换逻辑被抽取到助手类中

```

1
2 <html>
3 <head><title>Employee List - Refactored </title>
4 </head>
5 <body>
6
7 <h3> <center>List of employees</h3>
8
9 <corepatterns:employeeAdapter />

```

```

10
11 <table border="1" >
12   <tr>
13     <th> First Name </th>
14     <th> Last Name </th>
15     <th> Designation </th>
16   </tr>
17
18 <corepatterns:employeelist id="employeelist"
19   match="FirstName, LastName">
20 <tr>
21
22   <td><corepatterns:employee attribute= "FirstName"
23     case="Upper" /> </td>
24   <td><corepatterns:employee attribute= "LastName"
25     case="Upper" /></td>
26   <td><corepatterns:employee attribute= "Designation" />
27     </td>
28   <td>
29 </tr>
30 </corepatterns:employeelist>
31 </table>

```

现在，我们来考察另一种转换。有时，模型的转换会通过XSL转化来进行，这也可以用定制标记助手实现。同样，这一重构将转化逻辑抽离了JSP，使我们拥有模块划分更清晰、更容易复用的组件。在下面的例子中，JSP使用定制标记助手来执行转换，而不是直接在其内部执行：

```

1  <%@taglib uri="http://jakarta.apache.org/taglibs/xsl-1.0" prefix="xsl" %>
2  <xsl:apply nameXml="model" propertyXml="xml" xsl="/stylesheet/transform.xsl"/>

```

在这里，我们用Jakarta taglibs项目[JakartaTaglibs]所提供的xsl:apply标记来生成整个页面的输出，也可以用同样的方式生成页面的一部分。对该标记的上述调用要求页面范围内存在一个名为“model”的bean，并且这个bean必须有一个名为“xml”的属性。换句话说，在页面范围内必须有这样一个bean存在：它的实例名是“model”，而且它拥有具备下列签名的方法：

```
public String getXml()
```

值得一提的是，这两种转换都可以在完全不依赖于JSP的环境下进行。你可以综合考量各方面的因素（例如内容的存储格式、是否存在各种遗留技术等），再决定是否选择这种方式。

## 对客户端隐藏资源

客户端可以直接访问某些资源（例如JSP视图），但这种访问理应是受限的。

利用容器配置或者控制组件将这些资源隐藏起来（见图4-12和图4-13）。

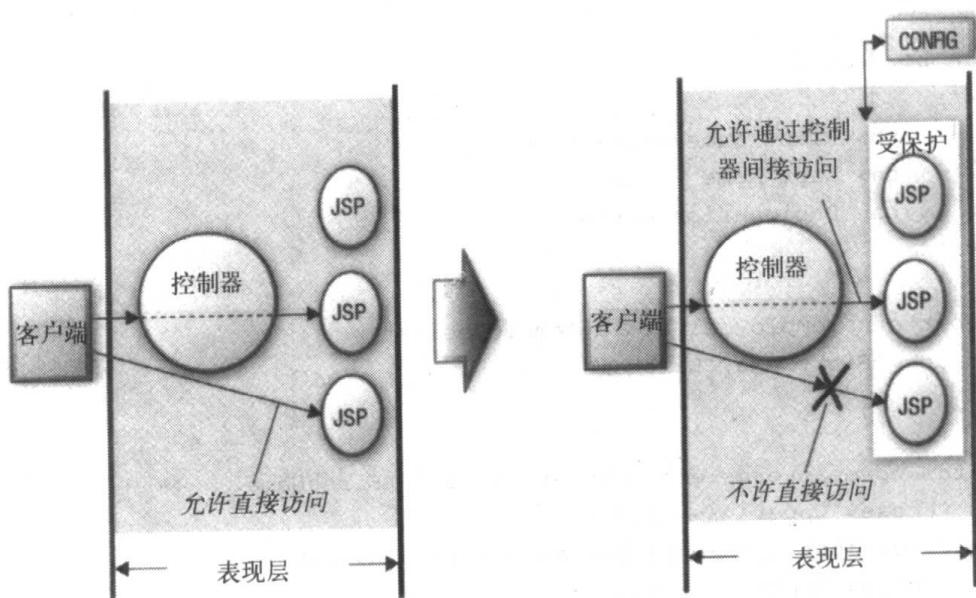


图4-12 利用容器配置限制访问

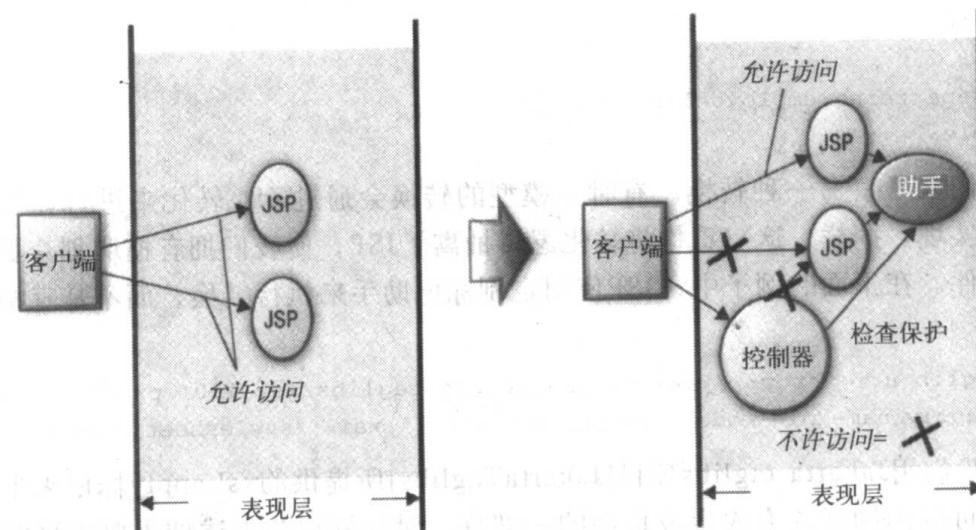


图4-13 使用控制组件进行访问控制

88

## 动机

很多时候，你会希望对进入应用程序的请求加以控制。本重构所描述的就是基于许可的控制和保护措施。

如果必须对客户端请求的顺序或流程加以控制，可以使用本章所介绍的“引入同步令牌”重构手法。

## 作法

- 将资源（例如Web资源、servlet，以及其他任何资源）移到Web应用的/WEB-INF/目录（或其中的子目录）之下，就可以利用配置控制对它们的访问。

⇒ 譬如说，对于一个名为**securityissues**的Web应用，如果要阻止用户通过浏览器直接访问其中名为info.jsp的视图，可以将JSP源文件放入下列子目录：

```
/securityissues/WEB-INF/internalaccessonly/ info.jsp.
```

□ 也可以用控制组件来限制对资源的访问。

⇒ 可以考虑引入一个控制器（参见本章的“引入控制器”），用于管理被保护资源的访问。

⇒ 另外，也可以让被保护资源管理自己的访问控制。在这种情况下，应该将实际的访问控制操作委派给一个助手类。

□ 创建一个或多个助手类。

⇒ 你可能选择用一个控制器统一管理资源访问，也可能让各个JSP分别管理各自的访问控制。不管选择哪种实现策略，控制器和JSP都可以委派这些助手类检查是否允许客户端访问该资源。

## 示例

### 借助容器配置的访问限制

只要把一个JSP移到/WEB-INF/目录下，客户端就不能直接访问它，必须通过控制器才能访问。

假设我们的web应用名为**corepatterns**，在web应用的根目录下有一个JSP源文件：

```
/corepatterns/secure_page.jsp
```

89

缺省情况下，通过下列URL，客户端就可以直接访问这个JSP资源：

```
http://localhost:8080/corepatterns/secure_page.jsp
```

要限制客户端的直接访问，我们只需将JSP源文件移至/WEB-INF/目录下：

```
/corepatterns/WEB-INF/privateaccess/secure_page.jsp
```

目录/WEB-INF/以及其中的所有子目录都只允许内部请求（例如由控制器或Request Dispatcher发出的请求）访问，客户端请求必须经由转发之后才能间接访问它们。于是，用户现在要使用类似下面这样的URL来访问secure\_page.jsp文件：

```
http://localhost:8080/corepatterns/controller?view=/corepatterns/WEB-INF/privateaccess/secure_page.jsp
```

---

**注意：**上述URL仅仅起示范的作用，“由客户端传递路径信息给服务器”并不是一种值得推荐的作法。视图所接受的查询参数不应该暴露服务器的目录结构。这个例子之所以这样做，仅仅是为了方便读者理解而已。

---

如果处理请求的是一个servlet控制器，那么它可以借助RequestDispatcher将请求转发至secure\_page.jsp文件。

另一方面，如果用户试图通过下列URL地址直接访问secure\_page.jsp文件，服务器就会告诉他：无法找到所请求的资源（如图4-14所示）。

```
http://localhost:8080/corepatterns/WEB-INF/privateaccess/secure_page.jsp
```

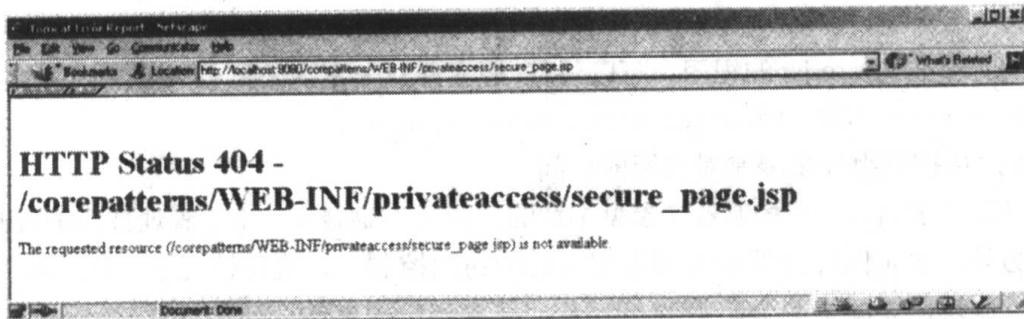


图4-14 通过简单的文件配置限制对资源的直接访问

90

### 使用控制组件进行访问控制

限制客户端访问的另一种选择是委派一个控制组件对资源加以控制，如图4-13和例4.11所示。

#### 例4.11 使用控制组件进行访问控制<sup>①</sup>

```

1  <%@ taglib uri="/WEB-INF/corepatternstaglibrary.tld" prefix="corepatterns" %>
2  <corepatterns:guard>
3  <html>
4  <head><title>Hide Resource from Client</title></head>
5  <body>
6
7
8  <h2>This view is shown to the client only if the
9  control component allows access. The view delegates
10 the control check to the guard tag at the top of the
11 page.</h2>
12 </body>
13 </html>
```

91

## 业务层和集成层的重构

### 用session bean包装entity bean

来自业务层的entity bean被暴露给位于其他层次的客户端。

使用会话面模式对entity bean加以封装（见图4-15）。

### 动机

entity bean是粗粒度的分布式持久对象。如果将entity bean暴露给位于另一个层次的客户端，会导致不必要的网络开销，进而影响系统性能。客户端在entity bean之上执行的每次方法调用都将是一次远程网络方法调用，这是一笔昂贵的开销。

另外，entity bean采用了容器管理的事务机制。如果将entity bean暴露给客户端，可能给客

<sup>①</sup> 本例中HTML页面里的文字意为：“只有当控制组件允许访问的时候，本视图才会显示。视图把控制检查的工作委派给页面顶端的保护标记（guard tag）。”

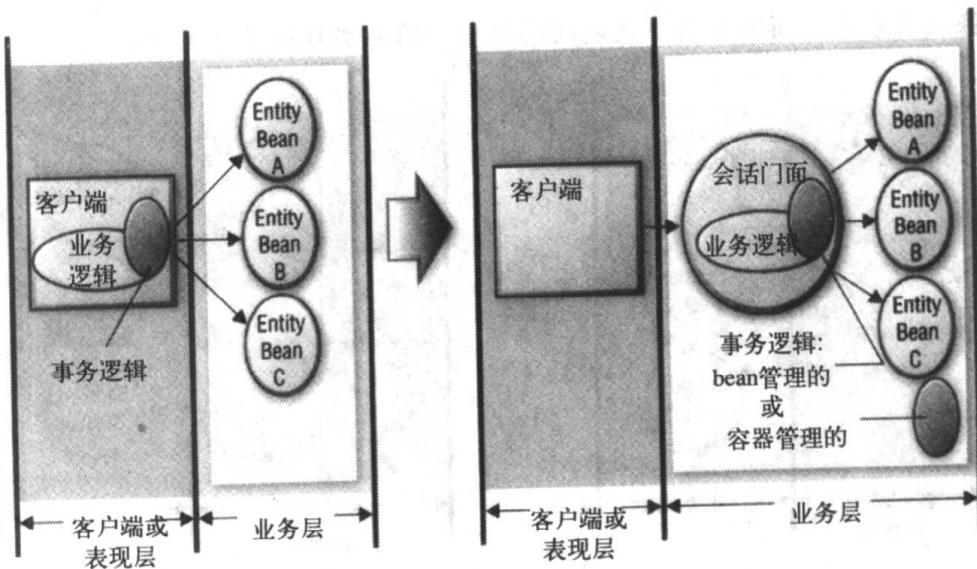


图4-15 用session bean包装entity bean

客户端开发者带来一个沉重的负担：在同时操作多个entity bean的时候，需要理解并设计entity bean的事务处理，同时还必须在多个事务之间划清界线。客户端开发者不得不从事务管理器获得一个用户事务，然后在这个事务的上下文中编写“与entity bean交互”的代码。由于客户端自己实现事务管理机制，也就不可能享受容器管理事务时“自动划分事务边界”的好处了。

## 作法

- 将“与entity bean交互”的业务逻辑移出客户端。
  - ⇒ 使用提炼类（Extract Class）[Fowler]的重构手法，将这部分逻辑从客户端抽取出来。
- 用一个session bean作为entity bean的门面。
  - ⇒ 这个session bean可以容纳“与entity bean交互”的逻辑，以及相关的工作流逻辑。
  - ⇒ 参见会话门面。
- 遵循会话门面模式，让多个session bean彼此配合，组成一个完整而统一的访问层，将entity bean彻底隐藏起来。
  - ⇒ 现在，客户端与entity bean之间的交互已经被移至位于业务层的会话门面中。
  - ⇒ 因此，客户端执行远程方法调用的次数大大减少了。
- 如果使用bean管理的事务机制，则需要在session bean中实现事务逻辑；如果使用容器管理的事务机制，只需在部署描述符中设定session bean的事务属性就可以了。
  - ⇒ 由于session bean担起了“与entity bean交互”的重任，客户端就无须再操心事务的划分了。
  - ⇒ 现在，所有事务划分的工作将由session bean或者容器来完成，这取决于设计者选择用户管理事务还是容器管理事务。

## 引入业务代表

位于业务层的session bean被暴露给位于其他层次的客户端。

92

93

使用业务代表模式，消除不同层次之间的耦合，隐藏实现细节（见图4-16）。

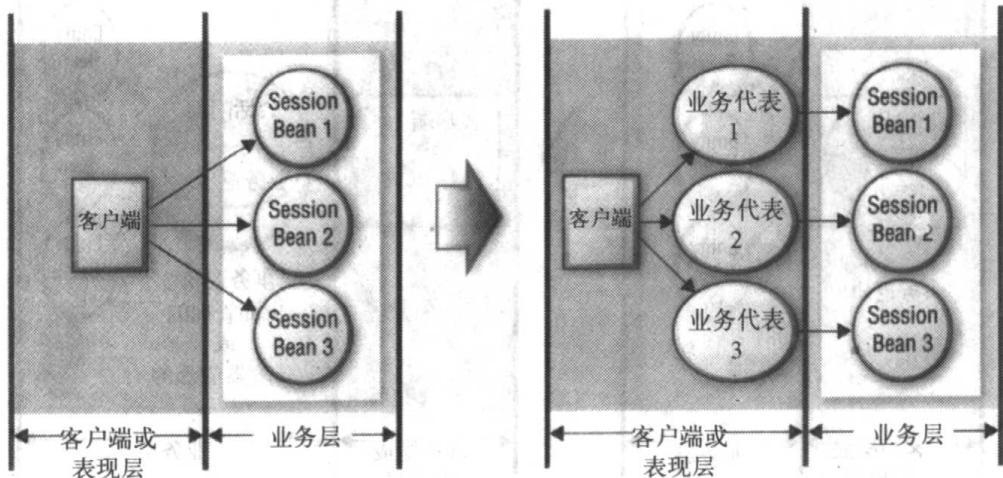


图4-16 引入业务代表

## 动机

正如在本章“用session bean包装entity bean”中所讨论过的，session bean常被用于实现entity bean的门面，它向业务服务提供了粗粒度的接口。但是，如果直接将session bean暴露给客户端应用程序，会造成客户端代码和session bean之间的紧密耦合。

另外，如果将session bean暴露给客户端，还会导致客户端代码中到处出现对session bean的调用。这样一来，session bean的接口一旦发生任何变化，就会对所有调用该session bean的客户端代码产生影响，极大地降低了代码的坚固性。而且，在使用EJB的同时，客户端也不得不了解服务层面上的异常。如果考虑到应用程序可能拥有多个不同类型的客户端，它们都通过session bean接口获得自己需要的服务，这时“session bean接口变化”造成的影响将成倍放大。

## 作法

- 对于每个被跨层暴露给客户端的session bean，引入一个业务代表。
  - ⇒ 业务代表以POJO的形式实现，它封装业务层的细节，并为客户端拦截所有服务层面的异常。
  - ⇒ 参见业务代表。
- 让每个业务代表分别应付一个session bean，通常让前者作为后者的门面。业务代表与会话门面之间设计为一对一的关系。
  - ⇒ 业务代表隐藏了业务服务的实现细节，从而降低了客户端层与业务服务(session bean)之间的耦合度。
  - ⇒ 在使用业务代表时，客户端只需对其进行本地方法调用，无须付出远程方法调用的昂贵开销。
- 将“与寻址服务和缓存有关”的代码封装至业务代表。
  - ⇒ 业务代表可以通过服务定位器查找业务服务。
  - ⇒ 参见服务定位器。

## 合并session bean

在session bean和entity bean之间存在一对一的对应关系。

在业务服务和session bean之间建立一对一的对应。如果某些session bean纯粹只作为entity bean的代理存在，将它们去掉或者合并，让session bean代表粗粒度的业务服务。

### 动机

在session bean和entity bean之间建立一对一的对应关系不会带来任何好处，只会凭空增加一层为entity bean充当代理的session bean。出现这种情况通常是因为开发者完全针对entity bean来创建session bean，而不是通过后者来表现粗粒度的服务。

有些设计者把“用session bean包装entity bean”解读为“每个entity bean都必须由属于它自己的session bean来保护”，这是一种误读，因为这会导致session bean沦落为entity bean的代理，而不是它本应成为的门面。关于“将entity bean暴露给客户端”这种作法的缺陷，在前面的“用session bean包装entity bean”部分中有深入的讨论。

在图4-17中，不同的客户端需要与不同的服务相交互，每次交互都涉及一个或多个entity bean。如果session bean与entity bean之间只有一对一的对应关系，那么客户端就必须分别与每个entity bean前面的session bean交互。既然session bean仅仅是entity bean的代理，这样的情况比“直接将entity bean暴露给客户端”也好不到哪儿去。

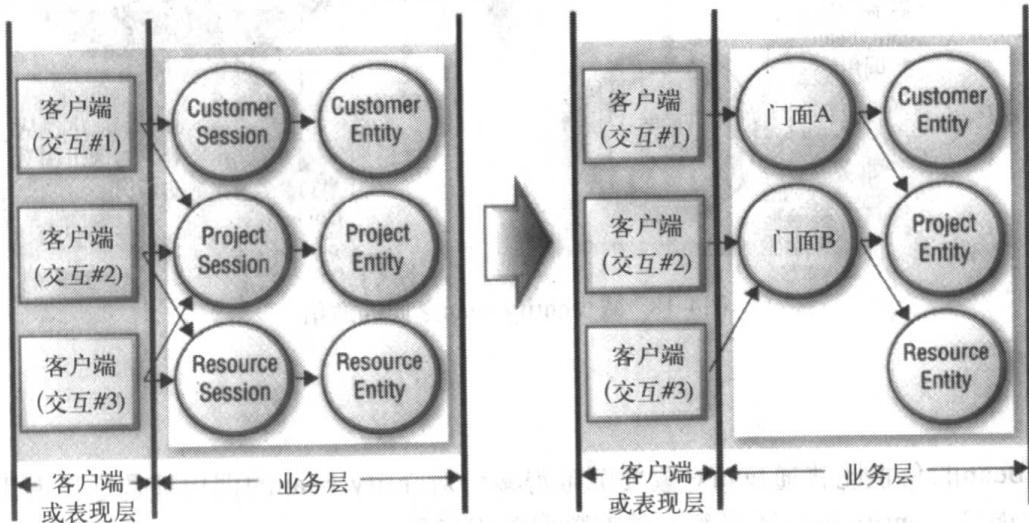


图4-17 合并session bean

### 作法

- 将session bean作为entity bean的门面。也就是说，session bean应该向客户端提供粗粒度的业务服务接口。
- 将一组业务上相关的、细粒度的session bean或者“纯粹是entity bean的代理”的session bean合并为一个。

- ⇒ session bean代表的是粗粒度的业务服务。
- ⇒ entity bean代表的是粗粒度、事务性的持久化数据。
- ⇒ 参见会话门面。
- 将一组业务上相关的、涉及同一个或几个entity bean的交互合并至一个会话门面，不要用一个单独的session bean来实现每次交互。
  - ⇒ 这样一来，session bean的数量将大大减少。每个session bean对内访问多个entity bean，对外提供风格统一的、粗粒度的业务服务。
  - ⇒ 会话门面的数量取决于对客户端交互的分组情况，而不取决于entity bean的数量。

97

## 减少entity bean之间的通信

各个entity bean之间的关联给模型带来不必要的负担。

用粗粒度的entity bean（复合实体）将彼此依赖的对象包容在一起，从而减少entity bean之间的关联（见图4-18）。

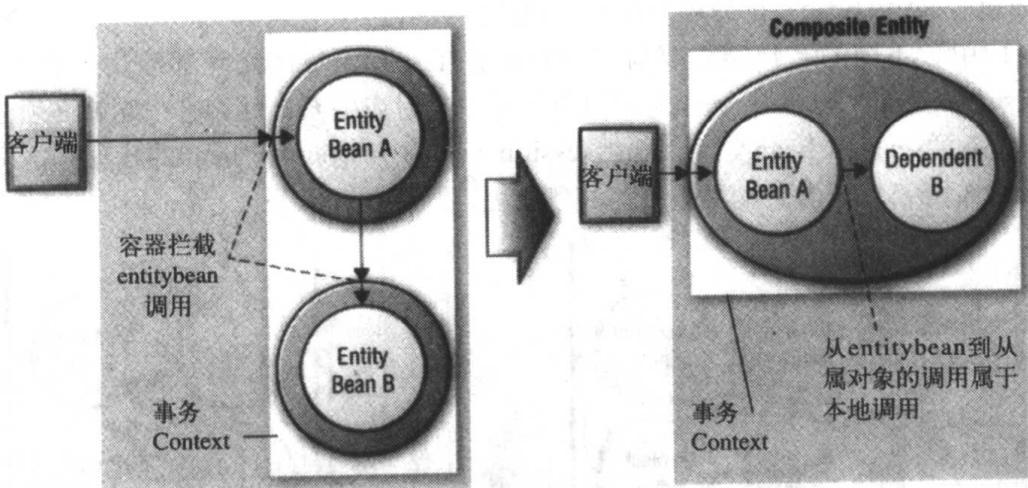


图4-18 减少entity bean之间的通信

## 动机

entity bean的负担比普通Java对象要沉重得多。对entity bean的调用都是远程调用，会造成网络开销。而且，entity bean还必须与外部数据源相交互。

哪怕两个entity bean位于同一个容器中，当其中一个调用另一个时，同样会使用远程方法调用的语义（容器也会涉足这次通信之中）。某些容器实现或许会对此类调用进行优化，因为不难发现这次调用来自同一个容器中的另一个对象。但这种优化终归是特定于厂商的，你不能依赖它。

另一个问题是：entity bean无法划分事务。在entity bean中，只能使用容器管理的事务。也就是说，在调用一个entity bean的方法时，容器可能开启一次新的事务，也可能加入当前的事务，还可能什么都不做，这取决于被调用方法的事务属性声明。当一个entity bean方法被调用时，事务会将所有与这个entity bean存在依赖关系的entity bean统统包含进来，并将它们绑定在事务上。

98

下文中。这会降低entity bean的整体性能和吞吐量，因为事务有可能锁定多个entity bean。在最糟糕的情况下，甚至有可能导致死锁。

## 作法

- 将entity bean设计并实现为粗粒度的对象。每个entity bean由两部分组成：根对象和从属对象。
  - ⇒ 将“entity bean之间的关联”转化为“entity bean与从属对象之间的关联”。
  - ⇒ 从属对象并不是entity bean，而是被包含在entity bean内部的普通Java对象。entity bean与其从属对象之间的关联是一种本地关联，不会造成网络开销。
  - ⇒ 使用懒装载策略和存储优化（脏数据标示器）策略，分别对复合实体的读取操作和存储操作进行优化。
  - ⇒ 参见复合实体。
- 将涉及“操作其他entity bean”的业务逻辑从entity bean中抽取出来，移至session bean中。
  - ⇒ 遵循会话门面模式，使用提炼方法（Extract Method）[Fowler]和/或搬移方法Move Method [Fowler]的重构手法，将上述业务逻辑移至session bean中。
  - ⇒ 参见会话门面。

99

## 将业务逻辑移至session bean

各个entity bean之间的关联给模型带来不必要的负担。

将涉及entity bean之间关联的工作流逻辑封装到一个session bean（会话门面）内部（见图4-19）。

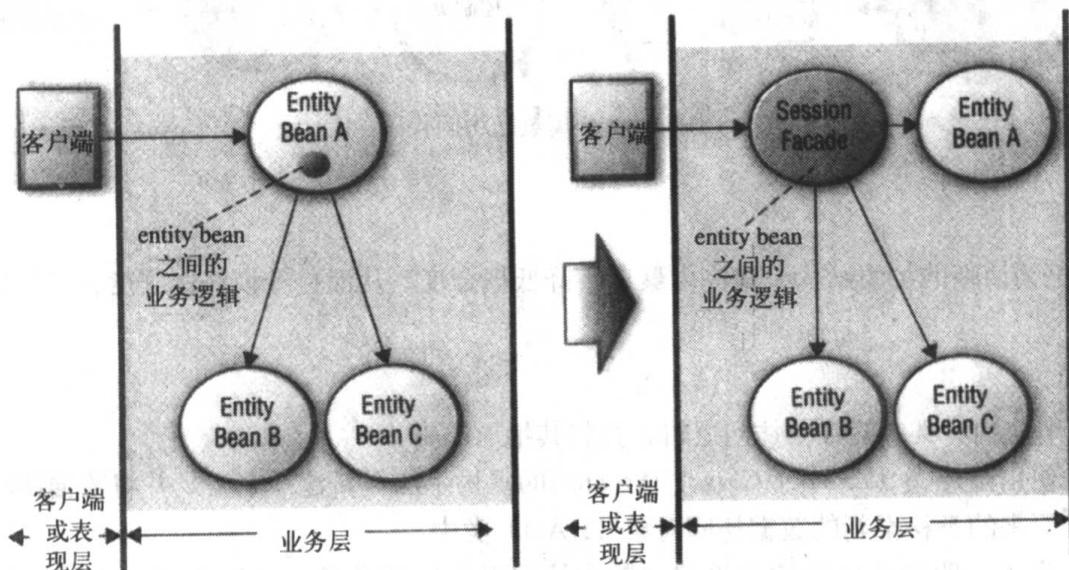


图4-19 将业务逻辑移至session bean

## 动机

在本章的“减少entity bean之间的通信”中，我们讨论了“entity bean之间的直接依赖”带

来的问题。现在的问题是：entity bean可能包含某些业务逻辑，需要使用别的entity bean，这就造成了entity bean之间直接或间接的依赖关系。“减少entity bean之间的通信”一节所讨论的问题同样适用于这里的场景。

## 作法

- 将涉及其他entity bean的业务逻辑从entity bean中抽取出来，移至session bean中。
  - [100] ⇨ 遵循会话界面模式，使用提炼方法（Extract Method）[Fowler]和/或搬移方法（Move Method）[Fowler]的重构手法，将上述业务逻辑移至session bean中。
  - ⇨ 参见会话界面。
  - [101] ⇨ 参见本章的“用Session bean包装entity bean”。

## 一般的重构

### 分离数据访问代码

负责访问数据的代码被直接嵌在一个具有其他——与数据访问无关的——职责的类中。

将负责数据访问的代码抽取至一个新的类中，并将这个类从逻辑上和/或物理上移至更接近数据源的地方（见图4-20）。

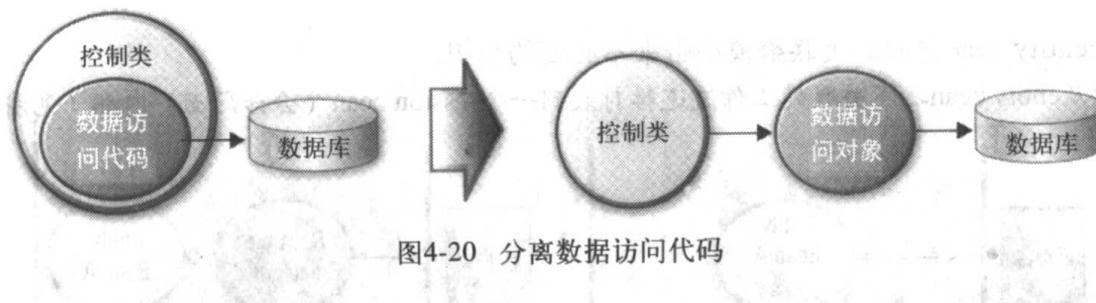


图4-20 分离数据访问代码

## 动机

建立更为清晰的抽象结构，提高内聚度，降低耦合度，从而提升模块化程度和可复用性。

## 作法

- 找出控制器类中的数据访问逻辑，并将其抽取出来。
  - ⇨ 使用提炼类（Extract Class）[Fowler]的重构手法，新建一个类，并将数据访问代码从原来的类移至新的数据访问对象（DAO）类中。
  - ⇨ 为了表明这个新建的类扮演“数据访问对象”的角色，可以考虑在它的名称中加上“DAO”字样。
  - ⇨ 参见数据访问对象模式。
- 在控制器中使用新建的DAO来访问数据。
- [102] 关于“应用程序划分”的相关信息，参见本章的“按层重构系统架构”。

## 示例

在下面的例子中，为了访问一些用户信息，servlet里嵌入了数据访问代码。首先，我们按照前面介绍的作法对设计加以修改，如图4-21所示。

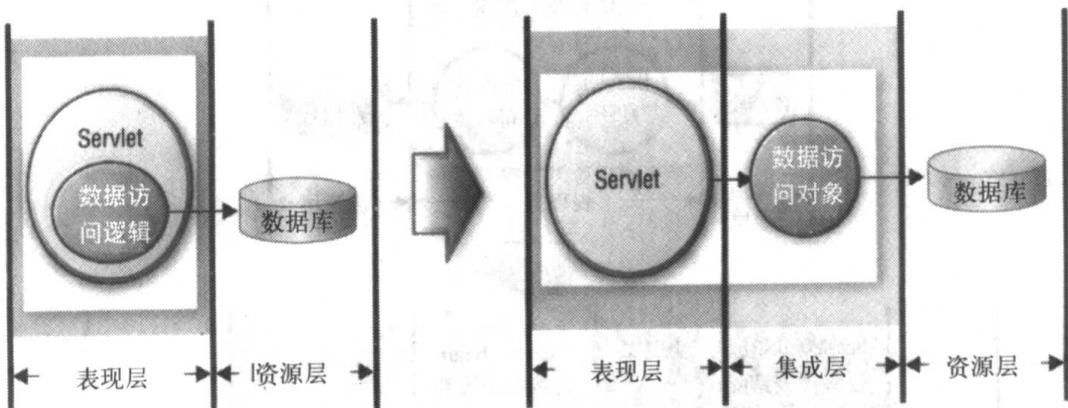


图4-21 分离数据访问代码——servlet的例子

现在，我们拥有两个类：一个是servlet，扮演“控制器”的角色；另一个是新建的UserDAO类，扮演“数据访问对象”的角色，用于访问用户信息。UserDAO封装了所有与Java数据库连接（JDBC）有关的代码，消除了servlet与实现细节之间的耦合。因此，servlet的代码就大大简化了。

在第二个例子中，一个使用bean管理的持久化策略的EJB里嵌入了持久化逻辑。持久化代码与EJB代码的混淆导致EJB耦合度极高，缺乏坚固性。如果将持久化代码作为EJB的一部分，持久化策略的任何改变都将导致这部分代码的修改，从而必须修改、重新发布整个EJB。这样的耦合对于EJB的维护非常不利。在这里，“分离数据访问代码”的重构手法同样有用。

103

借助本重构手法，我们对设计进行了修改，如图4-22所示。

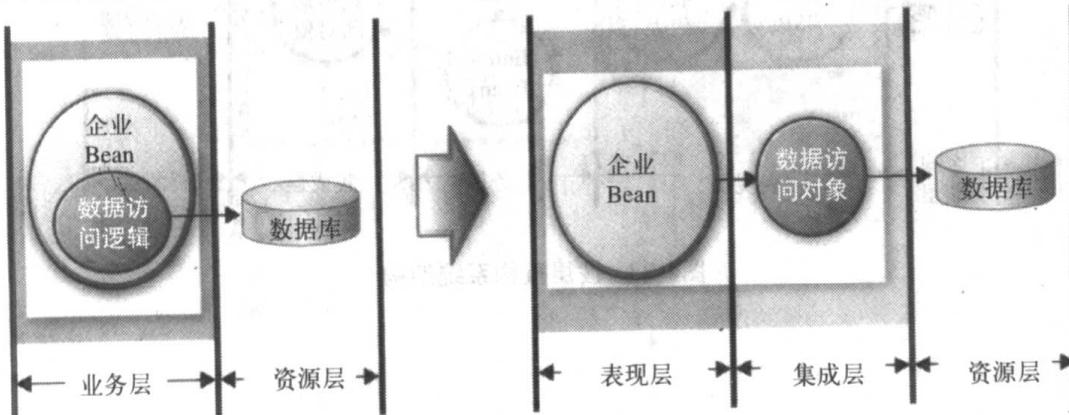


图4-22 分离数据访问代码——EJB的例子

104

## 按层重构系统架构

日益复杂的系统架构要求改变数据访问逻辑和处理逻辑所处的位置。

让数据访问代码从逻辑上和/或物理上靠近实际的数据源。将处理逻辑从客户端和表现层移至业务层（见图4-23）。

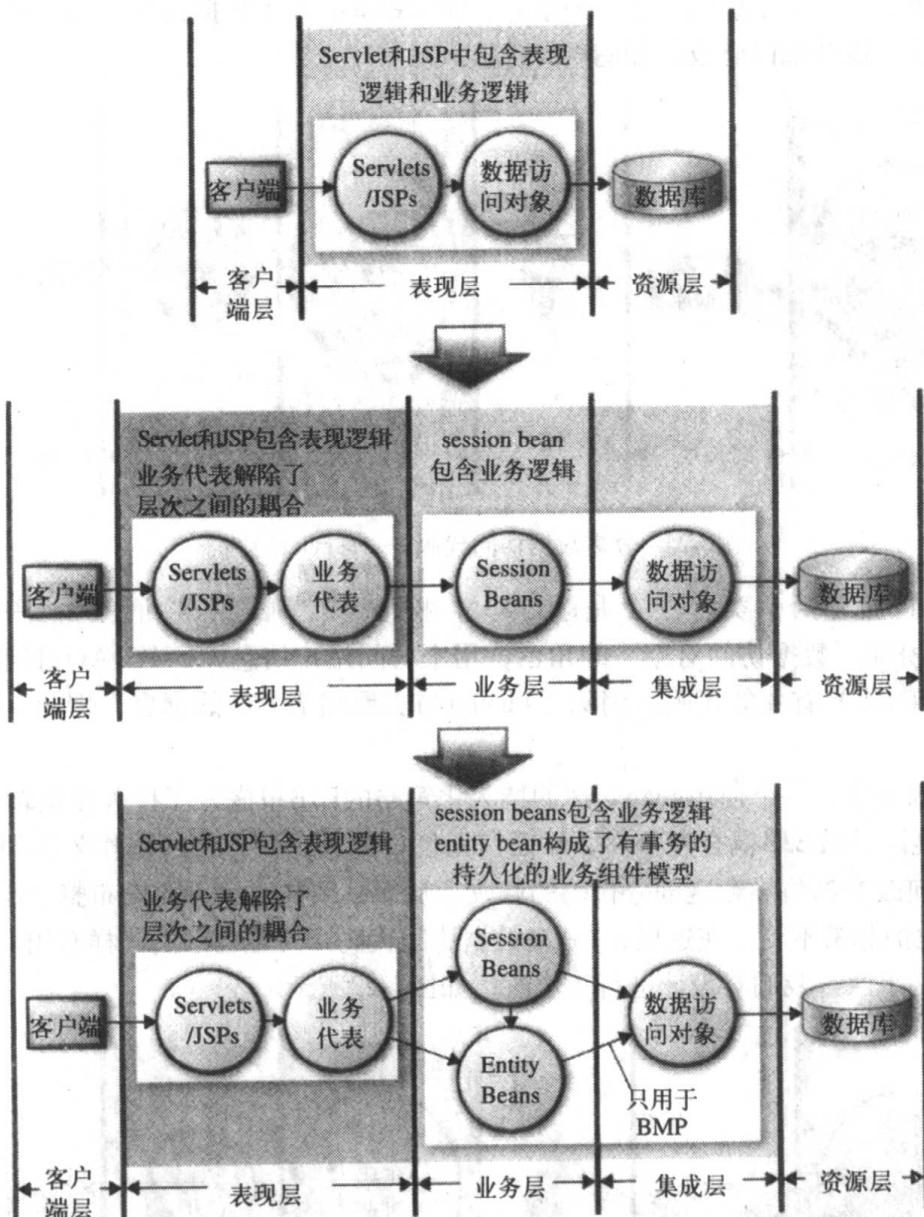


图4-23 按层重构系统架构

## 动机

本章的“分离数据访问代码”重构手法已经介绍了如何对数据访问逻辑进行重构，本重构手法则将讨论应用程序中其他类型的业务逻辑。

J2EE平台对servlet、JSP和EJB组件各自扮演的角色作出了清晰的划分，使它们分别关注应用程序中某一部分的问题。这样的划分对于可扩展性、灵活性、事务管理、安全性…都大有

裨益。

随着业务需求日益复杂，设计方案也必须不断改进，才能更好地应对业务服务所面临的持久化、事务管理、安全性、可扩展性等诸多方面的问题。当业务的复杂度达到一定程度时，我们需要引入session bean和entity bean，以便为所有客户端提供集中的业务处理能力，并利用EJB容器所提供的种种便利。

有些设计者盲目地使用了重量级组件（例如EJB），却没有首先确定应用程序是否确实需要这样做。在作出“是否使用EJB”的决策之前，需要综合考虑应用程序需求的复杂度，会影响这一决策的需求因素包括事务管理、安全性、可扩展性和分布式处理等。

## 作法

- 将数据访问代码从控制对象和实体对象中分离出来，移至数据访问对象中。
  - ⇒ 参见本章“分离数据访问代码”。
- 将表现逻辑和业务处理分离开。引入session bean，用于负责业务处理。将表现处理留在servlet和JSP中。
  - ⇒ 如果应用程序的需求变得日益复杂，或者需要对业务逻辑加以整合、以便为所有客户端（不仅仅是表现层客户端）提供统一的业务服务，就应该进行这一步重构。
  - ⇒ 引入session bean，使其作为业务服务处理组件。session bean通过数据访问对象访问持久化存储介质。
  - ⇒ session bean可以根据需要使用容器管理的事务划分或者bean管理的事务划分。
  - ⇒ 参见会话界面。
- 引入entity bean，使其作为模型共享的、事务性的、粗粒度的持久化业务对象。如果你的需求尚且不需要使用entity bean，请跳过这一步。
  - ⇒ 如果持久化业务组件变得日益复杂，并且你希望利用entity bean所提供的便利（例如容器管理的事务和容器管理的持久化（CMP）），就应该进行这一步重构。
  - ⇒ entity bean提供了容器管理的事务作为事务划分机制，这使得开发者可以通过部署描述中的声明来划分事务，而不必在EJB中对事务逻辑进行硬编码。
  - ⇒ 参见传输对象和复合实体。
- 使用业务代表，消除表现层和业务层组件之间的耦合。
  - ⇒ 业务代表能够消除表现层组件和业务层组件之间的耦合，并将诸如寻址之类的实现细节隐藏起来。
  - ⇒ 参见业务代表。

106

107

## 使用连接池

数据库连接没有被共享。每个客户端管理各自的数据库连接，并通过各自的连接发起数据库调用。

使用连接池，在池中保存几个预先初始化好的连接，以此提升可扩展性和性能（见图4-24）。

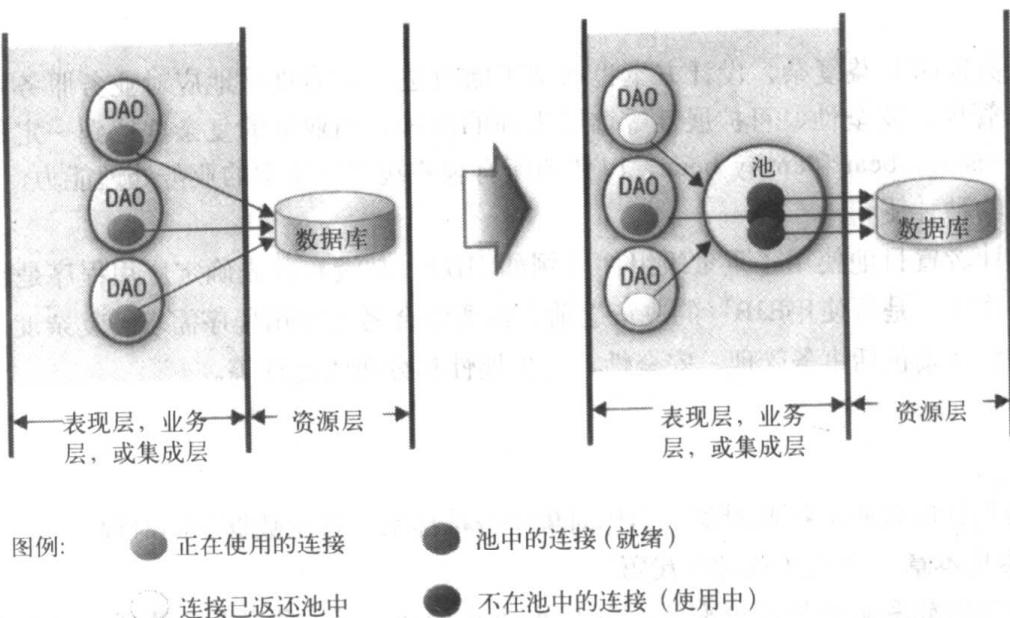


图4-24 使用连接池

108

## 动机

“开启数据库连接”是一个开销相当大的操作，会耗费大量的时间和资源。如果对数据库连接管理不当，系统的性能和可扩展性都会大受影响。由于数据库允许的连接数有限，如果每个客户端都管理自己的连接，连接总数可能会很快被耗尽。

如果项目以“分阶段进行”的方式引入EJB技术，就可能在表现层遇到这个问题。在这样的项目中，表现层组件最初是直接与数据库交互的，随后又将数据访问代码移到业务层、并封装在EJB层中。参见本章的“分离数据访问代码”和“按层重构系统架构”两种重构手法。

## 作法

- 创建一个接口，用于管理数据库连接，在其中提供用于获取和归还连接的方法。
  - ⇒ 使用提炼类 (*Extract Class*) [Fowler]和/或搬移方法 (*Move Method*) [Fowler]的重构手法，将现有的“获取数据库连接”的代码移至一个单独的类中，并使该类继承上述连接管理接口。
  - ⇒ 将“管理数据库连接”的代码抽取出之后，将原来使用这些代码的地方改为调用新建类的实例，例如 `connectionMgr.getConnection()` 和 `connectionMgr.returnConnection(conn)`。
  - ⇒ 请注意，在JDBC规范第2版中已经包含了“引入连接池”的一种标准机制。如果可能的话，应该尽量采用这种机制。在JDBC规范第2版中，这个管理接口是 `javax.sql.DataSource`，它提供了一个工厂，用于管理池中的 `Connection` (连接) 对象。
  - ⇒ 到目前为止，被标准化的只有连接池的结构和接口，但连接池的功能都是大同小异的。

- ⇒ 现在，我们还没有真正实现连接池。可以利用JDBC 2.0 DataSource工厂，这也是我们推荐的作法。109
- 修改连接管理器中“获取连接”方法的实现，使其预先初始化一些Connection实例，并让所有用户共享这些实例。这样，我们就引入了池缓存机制。
  - ⇒ 有很多公开的池缓存实现，你可以从中选择一个。
  - ⇒ 连接管理器实例的客户端通常是DAO，参见本章的“分离数据访问代码”。
  - ⇒ 随着项目的进展，数据访问代码通常会迁至（逻辑上）更接近数据库的地方。参见本章的“按层重构系统架构”。110



## 第二部分 J2EE模式目录

第二部分包括以下章节：

- 第5章——J2EE模式概览
- 第6章——表现层模式
- 第7章——业务层模式
- 第8章——集成层模式
- 尾声——Web Worker微架构

第5章“J2EE模式概览”提供了对J2EE模式目录的一个概览，并且讨论了我们把模式分层的思路。该章包括模式目录的一份指南，介绍了该目录常用的术语以及用来描述模式的UML构造型（UML Stereotype）。另外也定义、讨论了用来记录模式的模板。该章中的一个重要内容是对目录中各个模式之间关系的讨论——不仅讨论了这些模式相互的关系，而且还讨论了它们与其他文献（比如《设计模式》[GoF]、《软件架构模式》卷一[POSA1]、卷二[POSA2]等）所记载的模式的关系。本章另外一项重要内容是J2EE模式路线图，这其实是一张表，列举了一些与J2EE设计和架构相关的常见问题，并且把这些问题与特定的模式或重构联系起来。

111  
113

下面的几章描述了J2EE模式目录中的模式。

第6章“表现层模式”，提供了表现层的模式。这些模式描述了使用servlet和JSP技术的最佳实践。

第7章“业务层模式”，提供了业务层的模式。这些模式描述了利用EJB和POJO<sup>Θ</sup>设计业务层的最佳实践。

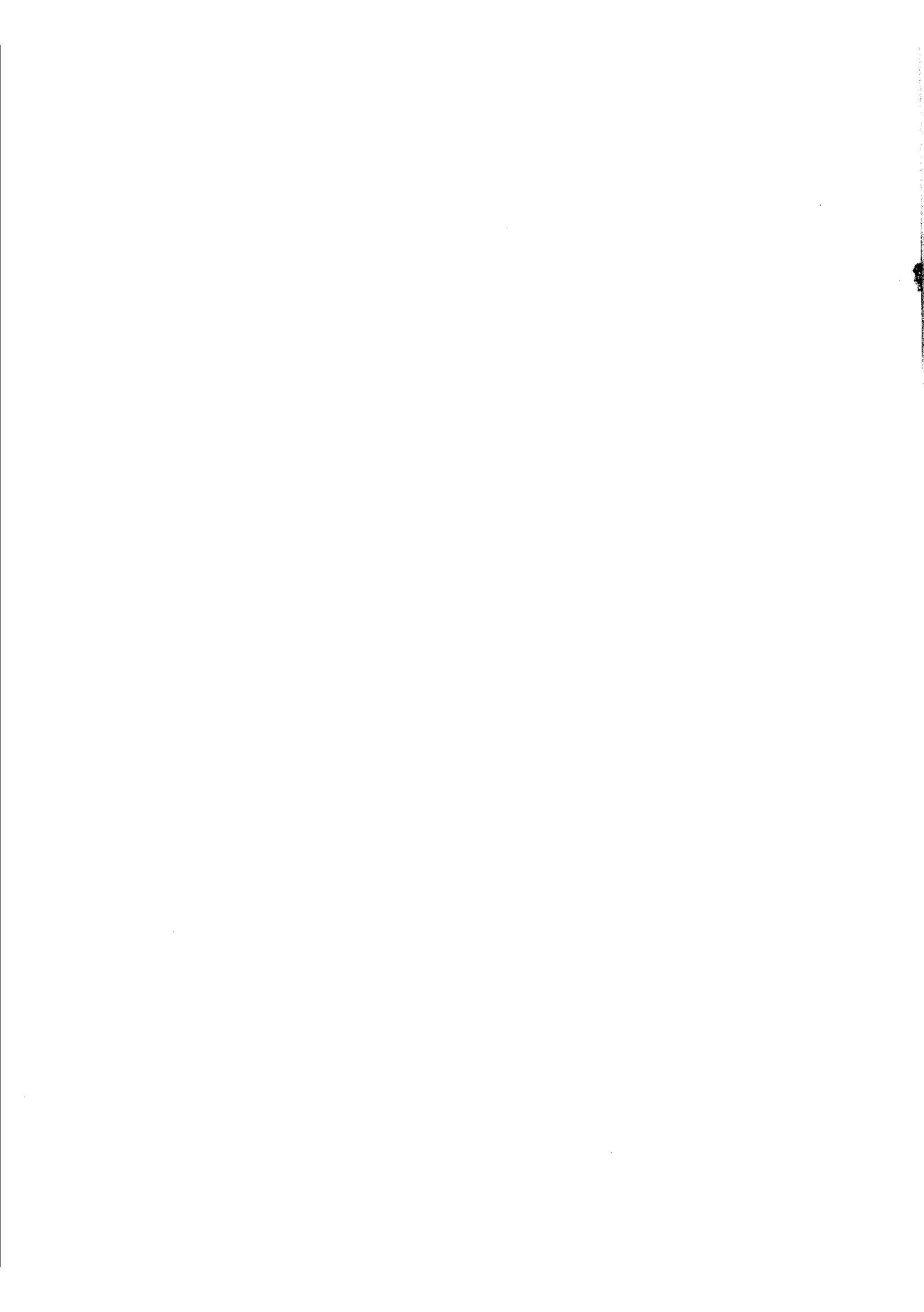
第8章“集成层模式”，提供了集成层的模式。这些模式描述了使用Web Service、JDBC、JMS技术的最佳实践。

尾声描述了一种用于J2EE工作流集成的微架构Web Worker。

114

---

<sup>Θ</sup> POJO：软件方法论大师Martin Fowler在《Patterns of Enterprise Application Architecture》中创造的说法，是plain old Java object的缩写，指普通的Java对象（而不是EJB等组件）。



## 第5章 J2EE模式概览

本章将涉及下列主题：

- 什么是模式
- 分层思路
- J2EE模式
- 目录指南
- J2EE模式关系
- 与现有其他模式的关系
- 模式路线图
- 小结

115

116

### 本章提要

本章首先对“模式”进行定义，然后讨论了分层设计在J2EE中的应用。接着，通过分析J2EE的目录指南，对J2EE模式进行了深入的探讨。最后，通过对比，展示了J2EE模式与其他模式的关系，并提出了J2EE模式的路线图。

J2EE模式提供了一组基于J2EE平台的对常见问题的解决方案。它们体现的是Sun Java中心的Java架构师们从大量成功的J2EE技术实践中获得的集体经验和技术能力。Sun Java中心是Sun公司下属的咨询机构，专门为客户提供基于Java技术的解决方案架构设计。Sun Java中心从J2EE平台创生之日起就开始基于这一平台创建解决方案了，主要关注于提高系统的服务质量（QoS，Quality of Service），包括提高系统的可扩展性、可用性、性能、安全性、可靠性和灵活性。

这些J2EE模式描述了企业应用开发者通常遇到的问题，并且提供了对这些问题的解决方案。这些解决方案来自我们一直以来同无数J2EE客户所做的工作，以及我们与其他遇到了类似问题的Java架构师的交流。这些模式固化了以上解决方案的精华，它们体现了长久以来通过我们的集体经验对解决方案做出的提升、精炼。换句话说，它们抽取出了每个问题中的核心要素，因此它们给出的解决方案，也体现了对理论和实践的一种切实可行的萃取。

我们的工作集中于J2EE领域，尤其集中于J2EE技术的几种关键成分，如EJB、JSP和servlet。在同J2EE客户一起使用以上技术的过程中，我们渐渐发现，有一些常见问题和一些难点往往会让优良的技术实现难于进行。因此，我们也研究出了一套有效的最佳实践和思路，用于组合使用多种J2EE技术。

这里描述的模式提取出了这些“最佳实践”的思路，并且以切实明了的方式呈现出来，使你能够把这些模式用到自己的应用系统中，满足实际需要。这些模式清晰、简要地介绍了多种已经验证可行的技巧。它们让你能够轻而易举地重用一些成功的设计和架构。简而言之，使用这些模式，就能够成功、快速地设计J2EE系统。

## 什么是模式

在第1章中，我们讨论了专家们对模式各种各样的定义。我们也讨论了与模式相关的一些问题，比如说使用模式的益处。这里，我们要在J2EE模式目录的语境中重新考察这些讨论。

正如第1章所述，一些专家把模式定义为：在一种上下文中，一类问题的一种可重复使用的解决方案。

这里的术语——上下文、问题、解决方案——需要进行一些解释。首先，什么叫“上下文”？上下文就是一种环境、一些周边事物、一种情况，或者说是某物处于其中的一些相关条件。第二，什么叫“问题”？问题就是一种未经解决的疑问，也就是某些需要研究、解决的东西。通常，问题受到它所出现的上下文的约束。最后，所谓“解决方案”也就是，在特定上下文中，有助于解决问题的那种答案。

那么，如果我们有了在某种上下文中对某个问题的解决方案，这就是一个模式了吗？不一定。因为模式的定义里还有一个特征，那就是“重复使用”。也就是说，只有能够反复应用，一个模式才能称得上有用。这就齐了吗？也许还不是。正如你所见，虽然模式的概念相当简单，但是实际上给这个术语下定义还是很复杂的事情。

我们给出了一些参考文献，你可以自己深研模式的历史，了解其他领域的模式。但是在我们的模式目录中，模式是按照以下主要特征描述的：问题、解决方案，以及其他重要方面（比

如约束和效果)。后面讨论模式模板的一节(见本章中的“模式模板”)更具体地解释了模式的这些“特征”。

## 发现模式

在Sun Java中心,我们实施了很多J2EE项目。好多次我们都发现,有一些类似的问题会在多个项目之中反复出现。我们还发现,对这些问题的解决方案也很相近。虽然具体的实现策略有所不同,但是整体解决方案非常类似。让我们简单谈谈我们发现模式的过程吧。118

当我们发现一种问题和解决方案反复出现,我们就尝试用模式模板来捕捉、记录它的特征。起初,我们把这些初步文档当成候选的模式。但是,除非我们能够多次在不同项目中都观察、记录到它们的应用,否则我们不会把这些候选模式加入到模式目录中。我们也实行一种称为“模式挖掘”的过程,也就是在已经实现的解决方案中寻找模式。

验证模式的过程中,我们会使用模式社区常说的大三律(Rule of Three)。这一规则适用于把候选模式加入模式目录的过程。根据这一规则,只有经过三个以上不同系统的检验,一个解决方案才能从候选模式升格为模式。当然,这条规则的解释空间还很大,但是对于发现一个模式而言,它确实有助于强调模式的上下文因素。

相近的解决方案往往体现的是同一个模式。但是在确定模式的构成时,很重要的一点是要考虑怎样才能最好地交流这个模式。有时候,如果能够给另一种解决方案单独命名,就能够增进开发者之间的交流。在这种情况下应该考虑将两种相近的解决方案记录为两个不同的模式。另一方面,有时候把相近的想法记录为同一个模式的不同策略,也许会对交流解决方案更有帮助。

## 模式 vs. 策略

当我们开始记录J2EE模式的时候,我们决定在一个比较高的抽象层次上进行记录。但另一方面,每个模式都包括多种策略,这也就提供了底层的实现细节。通过记录“策略”,每个模式就从多个不同的抽象层次包含了解决方案的内容。我们本来也可以把这些策略记录为模式的;但是我们认为,现在的模板结构最能够体现出策略与处于它们之上的模式之间的关系。

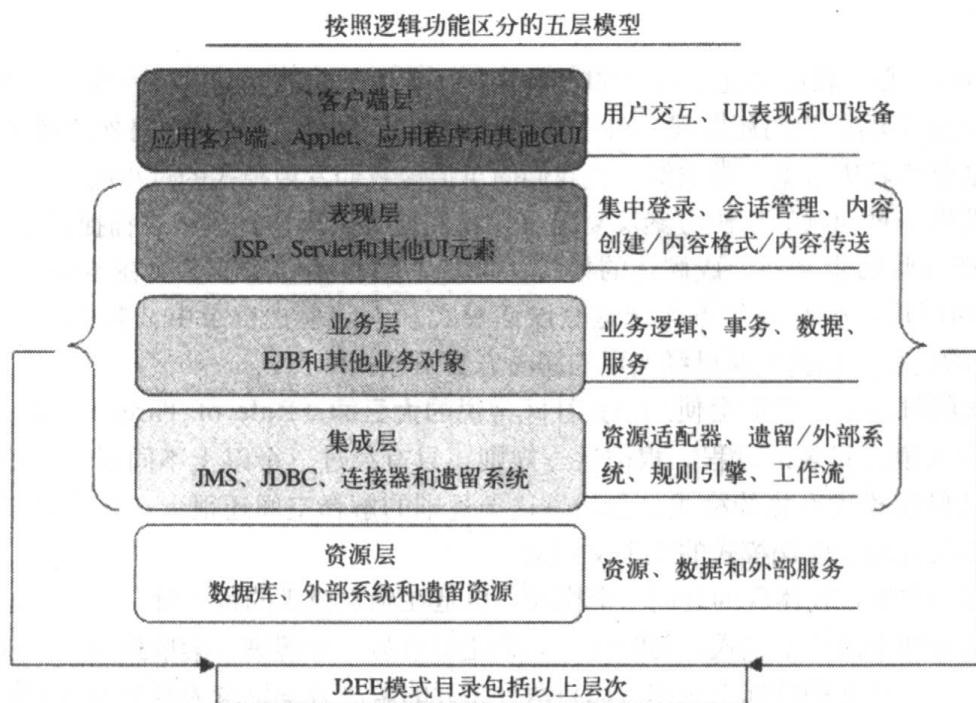
虽然对于是否把策略转化为模式一直就有激烈争论,但是目前我们还是暂时不做这样的决定,因为相信现在的记录方式相当清晰。我们也注意到,关于策略和模式之间的关系,有以下结论:

- 相对于策略,模式处于较高的抽象层次。
- 模式中包括了一些我们最为推荐、或者最为常见的实现方法,这就是策略。
- 策略为模式提供了扩展点。开发者会发现、发明出实现模式的新方法,这也就为那些众所周知的模式创造出新策略。119
- 策略命名了特定模式中的一些底层方面,因此也就增进了开发者对这些方面的交流。

## 分层思路

既然本目录描述了一些有助于构建J2EE应用程序的模式,既然J2EE平台(以及J2EE应用)

是多层系统，所以我们也要从多个层面考虑整个系统。所谓“层”，也就是系统中按照不同考虑方面划分的逻辑区间。系统中每个层都负有独立的职责。我们把层之间的区别看成是一种逻辑区分。每个层与相邻的层之间都存在松耦合。整个系统可以表现为多个层堆叠而成，如图5-1所示。



120

图5-1 分层思路

### 客户端层

该层包括了访问系统或应用的所有设备或系统客户端。客户端可以是Web浏览器、Java应用程序或者其他应用程序、Java applet、WAP手机、网络应用程序，或者是今后才会发明的某种设备。甚至可以是一个批处理过程。

### 表现层

表现层封装了服务于访问系统的客户端的所有表现逻辑。表现层拦截了客户端的请求，提供了单一的登录入口，构造了会话的管理，控制了对业务服务的访问，构造了响应，并且把这些响应传送给客户端。Servlet和JSP处于该层。注意，servlet和JSP本身不是UI（用户界面）元素，但它们产生UI元素。

### 业务层

该层提供了应用客户端需要的各种业务服务。该层中包括业务数据和业务逻辑。通常，应用系统中的大多数业务处理都集中在该层。但是，也可能出现这样的情况：由于原本遗留的系统，有些业务处理出现在资源层。对于实现业务层的业务对象，EJB组件是通常的、首选的解决方案。

### 集成层

这一层负责与外部资源、外部系统（比如数据存储和遗留应用系统）通信。只要业务对象需要处于资源层的数据和服务，那么业务层和集成层之间就存在耦合。这一层中的组件可能使用JDBC、J2EE连接器技术、或者某种厂商专有的中间件，从而与资源层协作。

### 资源层

资源层中包括业务数据和外部资源（比如大型主机、遗留系统<sup>②</sup>、B2B集成系统、以及类似于信用卡认证之类的服务）。

## J2EE模式

我们使用以上分层思路，按照功能来划分J2EE模式，而我们的模式目录正是遵循了这一思路。其中表现层模式包括了与servlet和JSP技术相关的模式。业务层模式包括了与EJB技术相关的模式。集成层模式包括了与JMS和JDBC相关的模式。关于模式之间的关系，请参见本章的图5-2。

[121]

### 表现层模式

表5-1列出了表现层模式，另外包括对每个模式的简要说明。

表5-1 表现层模式

模 式 名 称	简 要 说 明
拦截过滤器 ( <i>Intercepting Filter</i> )	用于对请求的预处理和后处理
前端控制器 ( <i>Front Controller</i> )	提供了用于管理请求处理的一个集中控制器
<i>Context</i> 对象 ( <i>Context Object</i> )	以独立于具体通信协议的形式封装了状态，使状态能够在整个应用系统中共享
应用控制器 ( <i>Application Controller</i> )	实现了操作 (action) 和视图管理的集中化、模块化
视图助手 ( <i>View Helper</i> )	把与表现格式无关的逻辑封装在助手组件中
复合视图 ( <i>Composite View</i> )	从多个子组件创建一个聚合视图
服务到工作者 ( <i>Service to Worker</i> )	把前端控制器模式、视图助手模式和一个分配器组件结合起来
分配器视图 ( <i>Dispatcher View</i> )	把前端控制器模式、视图助手模式和一个分配器组件结合起来，延迟了很多视图处理操作

[122]

### 业务层模式

表5-2列出了业务层模式，另外包括对每个模式的简要说明。

② 遗留系统 (legacy system)，指的是原有业务系统中要集成进入新系统的部分。

表5-2 业务层模式

模 式 名 称	简 要 说 明
业务代表 ( <i>Business Delegate</i> )	封装了对业务服务的访问
服务定位器 ( <i>Service Locator</i> )	封装了服务和组件的寻址
会话界面 ( <i>Session Façade</i> )	封装了业务层组件，把粗粒度服务暴露给远程客户端
应用服务 ( <i>Application Service</i> )	集中、聚合了系统行为，提供了一个统一的服务层
业务对象 ( <i>Business Object</i> )	使用业务模型区分业务数据和业务逻辑
复合实体 ( <i>Composite Entity</i> )	使用本地entity bean和POJO实现业务对象
传输对象 ( <i>Transfer Object</i> )	在各层之间传输数据
传输对象组装器 ( <i>Transfer Object Assembler</i> )	把来自多个数据源的数据组装成一个复合传输对象
值列表处理器 ( <i>Value List Handler</i> )	处理查询、缓存结果，提供逐个访问结果和选择特定结果的能力

## 集成层模式

表5-3列出了集成层模式，另外包括对每个模式的简要说明。

表5-3 集成层模式

模 式 名 称	简 要 说 明
数据访问对象 ( <i>Data Access Object</i> )	抽象并封装了对持久化存储的访问
服务激活器 ( <i>Service Activator</i> )	接收消息，并异步调用处理过程
业务领域存储 ( <i>Domain Store</i> )	为业务对象提供了一套透明的持久化机制
Web Service中转 ( <i>Web Service Broker</i> )	通过XML和web协议暴露出一个或多个服务

## J2EE模式目录指南

为了帮助你高效地理解、应用本目录中的J2EE模式，我们建议在阅读单个模式之前首先熟悉本节的内容。这里，我们介绍了模式的术语，解释了我们对统一建模语言 (UML)、构造型 (stereotype) 和模式模板的使用。简单地说，就是解释了如何使用模式，另外还从较高层面提供了一张目录中的模式的路线图。

## 术语

企业计算领域的工作者，特别是使用基于Java的系统的那些公司，把大量术语和缩写混进了他们的语言中。虽然很多读者对这些术语还算熟悉，但是有时候术语的用法还会根据场景变化。为了避免误解，保持术语含义的一致性，我们在表5-4中定义了我们对这些术语和缩写的使用方法。

表5-4 术语

术语	描述/定义	使用环境
BMP	Bean管理持久化: entity bean的一种策略, bean开发者负责实现entity bean的持久化逻辑	业务层模式
CMP	容器管理持久话: entity bean的一种策略, 容器服务负责透明地管理entity bean的持久化	业务层模式
复合 (Composite)	一个包含了其他对象的复杂对象。也与GoF的《设计模式》中介绍的复合模式有关(见本表后面介绍的GoF)	复合视图、复合实体
控制器 (Controller)	与客户端交互, 控制、管理每个请求的处理	表现层和业务层模式
数据访问对象 (Data Access Object)	一个对象, 封装并抽象了对持久化存储或外部系统的数据访问	业务层和集成层模式
代表 (Delegate)	一个对象, 作为另一个组件的居中代理; 一个中间层。代表具有代理和门面的特性	业务代表以及其他很多模式
从属对象(Dependent Object)	一种对象, 不能独立存在, 其生命周期由另一个对象管理	业务对象和复合实体
分配器 (Dispatcher)	控制器的职责包括选定和分派合适的视图。这种操作可以单独放入一个独立的组件, 该组件称为分配器	分配器视图、服务到工作者
EJB (Enterprise Bean)	指EJB组件, 可以是session bean或者entity bean实例。当我们使用EJB这个概念的时候, 也就意味着这个bean实例既可以是entity bean, 也可以是session bean	本书多处
门面 (Facade)	一种隐藏底层复杂性的模式; 在GoF《设计模式》一书中有描述	会话门面模式
工厂 (抽象工厂或工厂方法) (Factory (Abstract Factory or Factory Method))	GoF《设计模式》一书中描述的模式, 用于创建对象或整个一个对象家族	业务层模式、数据访问对象
迭代器 (Iterator)	一种模式, 提供了对集合的访问; 在GoF《设计模式》一书中有描述	值列表处理器
GoF	“四人帮”——指设计模式名著《设计模式: 可重用面向对象软件的基础》的作者: Erich Gamma、Richard Helm、Ralph Johnson 和John Vlissides[GoF]	本书多处
助手 (Helper)	负责帮助控制器和/或视图。比如, 控制器和视图可能会把以下操作委派给助手: 内容获取、验证、保存模型或者按照显示用途调整模型	表现层模式、业务代表
独立对象(Independent Object)	一种对象, 可以独立存在, 并可能会管理其从属对象的生命周期	复合实体模式
模型 (Model)	对系统或子系统的一种物理或逻辑表示	表现层和业务层模式
持久性存储 (Persistent Store)	指各种持久化存储系统, 比如RDBMS、ODBMS、文件系统等等	业务层和集成层模式
代理 (Proxy)	一种模式, 作为另外一个对象的替代, 从而控制对该对象的访问; 在GoF《设计模式》一书中有描述	本书多处

124

125

126

(续)

术 语	描述/定义	使 用 环 境
Scriptlet	JSP中直接包含的应用逻辑	表现层模式
Session Bean	指无状态或有状态session bean。可能也是session bean的home对象、远程对象以及bean的实现的一个总称	业务层模式
单件 (Singleton)	一种模式，提供了一个对象的单一实例，在GoF《设计模式》一书中有描述	本书多处
模板 (Template)	模板文本指的是JSP视图封装的文字文本。另外，模板还可以指在显示中组件的一种特定布局	表现层模式
传输对象 (Transfer Object)	一种可串行化的POJO，用来在不同的对象、层之间传输数据。其中不包含任何业务方法	业务层模式
视图 (View)	视图负责管理所有构成显示的图形和文本。它与助手交互，从而获得填充显示的数据。另外，它还可能把一些操作——比如内容获取操作——委派给助手	表现层模式

## UML的使用

在模式目录中，我们大量使用了UML，尤其是以下内容：

- **类图**——我们用类图表现模式解决方案的结构以及实现策略的结构。这种图示体现了解决方案的静态视图。
- **序列图（或交互图）**——我们用序列图/交互图表现解决方案或策略中多个参与者之间的交互。这体现了解决方案的动态视图。
- **构造型 (stereotypes)**——我们使用构造型来表现在类图和交互图中不同类型的对象。表5-5中列出了一些构造型和它们的含义。

模式目录中的每个模式都包括一张类图（体现了解决方案的结构），以及一张序列图（体现了该模式的交互）。另外，如果一个模式包含多种策略，那么还会使用类图和序列图解释每个策略。

要想了解有关UML的更多内容，请参照参考书目。

## UML构造型

在阅读模式和相关图示的时候，你会遇到一些构造型。所谓构造型 (stereotypes) 也就是设计师、架构师们创造、使用的一些术语。在本书的UML中，我们创造、使用了以下构造型，目的是更简洁、更易懂的方式给出图示。请注意：一些构造型和上一节中介绍的一些术语有关。除了这些构造型，我们还把一些模式名称和模式中的主要角色作为构造型使用，以便描述模式及其策略。

表5-5 UML构造型

构造型	含义
EJB	表示一个EJB组件；与一个业务对象相连。这个角色通常由session bean或entity bean完成
SessionEJB	表示作为整体的session bean，对session bean远程接口、home接口和bean实现不做区分
EntityEJB	表示作为整体的entity bean，对entity bean远程接口、home接口、bean实现和主键不做区分
View	一个视图：向客户端表现、显示信息
JSP	一张JSP页面；通常视图由JSP实现
Servlet	一个Java servlet；通常控制器由servlet实现
Singleton	一个类，只有单一实例，符合单件模式
Custom Tag	正如JavaBean一样，JSP定制标记也被用来实现助手对象。助手负责的操作包括：获取视图所需的数据、按视图的需要转换数据模型。助手可以简单地获取原始数据，也可以按照Web内容的格式组织数据，从而满足视图对数据的需求

## 模式模板

所有的J2EE模式都是按照一个事前定义好的模式模板来组织的。这个模式模板包括若干部分，每个部分都体现了特定模式的一个单独的属性。你也会注意到，我们给每个J2EE模式起了一个描述性的名称。虽然无法把一个模式的方方面面都包含在名称中，但模式的名称还是力图让读者能够了解模式的功能。就像现实生活中的名字一样，我们对模式的命名也会影响读者理解模式、并最终应用模式的方式。

129

我们采用的模式模板包括以下部分：

- **问题：**描述了开发者面对的设计问题。
- **约束：**列出了影响问题和解决方案的各种原因和动机。在约束部分将列出选择这个模式的原因，讲明使用该模式的道理。
- **解决方案：**简要地描述了解决思路，然后描述了解决方案中的细节因素。这一部分又包括两个小部分：
  - **结构：**使用UML类图表现解决方案的结构。这个部分中UML序列图体现了该解决方案的动态机制。对于图中的参与者和协作关系将有详细解释。
  - **策略：**描述了实现一个模式的不同方式。请参照本章前面的“模式 vs. 策略”部分，以便更好地理解“策略”的含义。当能够用代码说明一个策略时，我们就会在这一部分加入一些代码片段。如果需要的代码比“片段”还更详尽、更长，我们则会把这些代码放到模式模板的“示例代码”部分。
- **效果：**这里我们会讨论应用模式的权衡考虑。总的来说，这个部分集中考虑使用特定模式或策略的结果，并且会说明应用模式后产生的益处和弊端。
- **示例代码：**这一部分包括对模式和策略的样例实现和代码清单。如果示例代码能够充分包括在“策略”部分的讨论中，那么这个部分可能就会省去。

- 相关模式：这个部分列出在J2EE模式目录或其他外部资源中（比如GoF的设计模式中）的相关模式。对于每一个相关模式，都会有一段简要说明，介绍它与讨论中的模式的关系。

## J2EE模式关系

最近，一组架构师和设计师表达了这样一种关注：对于怎样组合使用模式、构成大型解决方案，开发人员们似乎缺乏理解。我们在这里用一种高抽象层次的图示来体现模式以及它们之间的关系，借此尝试解决上述问题。这种图示称为J2EE模式关系图，如图5-2。在尾声的“Web Worker微架构”中，我们另外考察了一种样板用例，示范使用多个模式组合起来，构成一个模式框架，从而实现一个用例。  
[130]

单个的模式在实现一种需求的时候，体现出它们的上下文、问题和解决方案。但是，模式目录就像一幅大画，需要后退几步、站远一些，才能看到全景，理解怎样最佳应用多种模式。如果具备了对这种全景（大画）的理解，也就能更好地在J2EE应用系统中使用模式。

还是重复一下第1章引用过的Christopher Alexander的话：模式不能孤立存在，它需要和其他模式相互支持，这样才能具有意义、具有用处。理论上说，这个模式目录中的每个模式与其他模式之间都存在关系。在设计、构造一个解决方案时，如果理解了模式之间的这种关系，能够具有以下帮助：

- 提醒你在考虑采用一个新模式解决问题的时候，应该考虑这会引入什么新问题。这是一种多米诺效应：当把一个特定的模式引入架构时，会引入什么新问题呢？在编码开始之前找出这种冲突，是非常重要的。
- 提醒你回顾模式关系图，从而考虑替代方案。当确定了可能的问题之后，回顾一下模式关系图，看看有什么替代方案。也许新问题可以通过另外的模式解决，也可能通过结合使用你已经选中的模式和另一个模式来解决。

图5-2表现了模式之间的关系。

拦截过滤器拦截输入的请求和输出的响应，并加入了一个过滤器。过滤器可以通过配置声明加入和取消，这样多个过滤器还可以不冲突地组合使用。当预处理以及/或者后处理完成后，这组过滤器中的最后一个把控制交给原来的目标对象。对于输入的请求来说，这个目标对象通常是前端控制器，但也可能是一个视图。

前端控制器是一个容器，用来装载表现层的共通处理逻辑（如果不采用这个控制器，逻辑代码就会被错误地放在视图里）。控制器负责处理请求，进行内容的获取、安全性、视图管理、导航等操作，并委派一个分配器组件分派视图。  
[131]

应用控制器集中了控制、数据获取、视图和命令处理的调用。前端控制器为输入的请求提供了一个集中的访问点和控制器，而应用控制器则负责找出并调用命令，并且还要找出并分派视图。

*Context*对象以独立于具体通信协议的形式封装了状态，使状态能够在整个应用系统中共享。使用*Context*对象能够简化测试过程，提供一个更为通用的测试环境，减少了对特定容器的依赖。

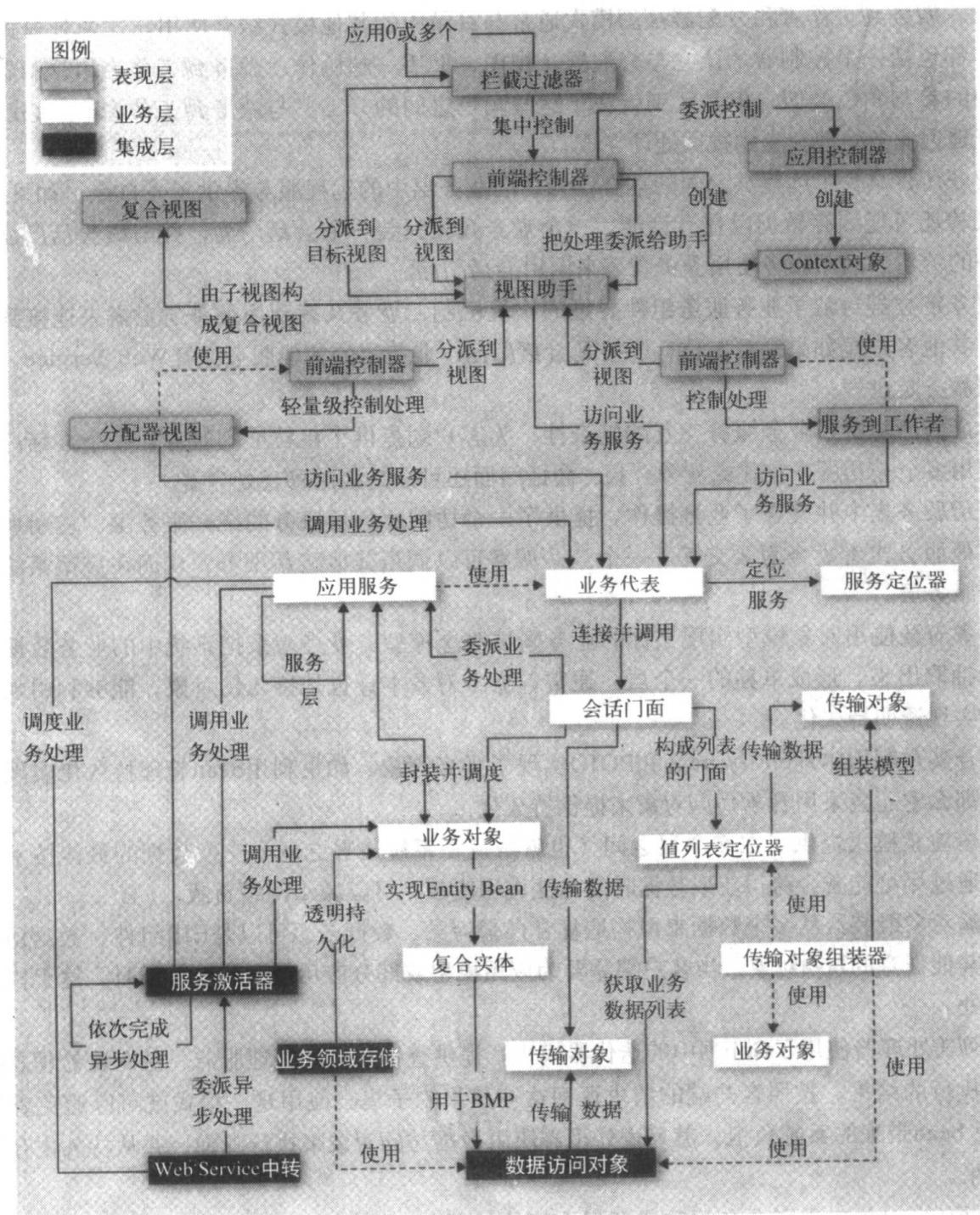


图5-2 J2EE模式关系图

132

视图助手鼓励把用于表现格式的代码和其他业务逻辑分离开。它要求用助手组件来封装与初始化内容获取、验证、模型的转换和格式化相关的逻辑。这样，就可以用视图组件来封装表现格式的代码。助手组件通常要通过业务代表或应用服务把请求委派给业务层，而视图为了创建模板则可能由多个子组件组成。

复合视图把大量小块数据拼合成一个视图。多个静态或动态的子视图一起构成了一个独立

的模板。服务到工作者和分配器视图模式通常与目录中的其他模式结合使用。这两种模式结构相同，都包括一个控制器，让它与分配器、视图、助手一起协作。服务到工作者和分配器视图模式中的参与角色类似，但是区别就在于这些角色之间的分工。与服务到工作者不同分配器视图一直延迟业务处理，直到视图处理完成。

业务代表减少了远程层之间的耦合，为访问业务层中的远程服务提供了入口点。如果必要，业务代表还可能缓存数据以提升性能。一个业务代表封装一个会话门面，并与该会话门面维持一对一的关系。应用服务使用业务代表来调用会话门面。

服务定位器封装了业务服务组件寻址的实现机制。业务代表利用服务定位器来连接到会话门面。其他客户端如果需要定位、连接到会话门面、业务层的其他服务以及Web Service，都可以使用服务定位器。

会话门面隐藏了业务服务交互的复杂性，为客户端提供了粗粒度的服务。一个会话门面可能会调用多个应用服务实现或业务对象。会话门面还可以封装值列表处理器。

133

应用服务集中并聚合了业务操作，提供了一个访问业务层服务的统一服务层。应用服务可以与其他服务或者业务对象交互。一个应用服务可以调用其他应用服务，从而在应用系统中形成一个服务层。

业务对象使用对象模型实现了你的业务领域概念模型。业务对象把系统中的业务数据和业务逻辑抽取出，形成单独的一个层。通常，业务对象本身也是持久化对象，能够利用业务领域存储实现透明持久化。

复合实体利用本地entity bean和POJO实现了业务对象。如果利用bean管理持久化实现复合实体，那么它还会采用数据访问对象来提供持久化。

传输对象模式提供了在多个层之间（也就是说在系统边界之间）交换数据的最佳技术和策略，它通过尽可能减少跨层获取数据时的方法调用次数，可以减小网络负载。

传输对象组装器从多个数据来源构造复合传输对象。数据来源可以是EJB组件、数据访问对象或者其他任意的Java对象。当客户端需要为应用模型或部分应用模型获取数据时，这个模式就最有用处。

值列表处理器使用了GoF书中的迭代器模式，提供查询执行和处理服务。值列表处理器缓存了查询执行的结果，按照客户端的请求返回这一结果的子集。使用这一模式就可以避免查询大量entity bean带来的系统负担。值列表处理器使用数据访问对象来执行查询，并从持久化存储中获取结果。

数据访问对象实现了业务层和资源层之间的松耦合。数据访问对象封装了所有的数据访问逻辑，包括从一个持久化存储中创建、获取、删除、以及修改数据。数据访问对象使用传输对象传送、接收数据。

服务激活器利用JMS，使得企业应用系统具备异步处理能力。服务激活器可以调用应用服务、会话门面或业务对象。你还可以使用多个服务激活器，为需要长时间运行的任务提供并行异步处理。

业务领域存储提供了一种为你的对象模型实现透明持久化的强大机制。它组合、连接了多个其他的模式，包括数据访问对象等。*Web Service*中转利用了XML和标准Internet协议，把应用系统中的一个或多个服务暴露、中转给外部客户端。*Web Service*中转可以与应用服务和会话界面交互。*Web Service*中转使用一个或多个服务激活器来执行对请求的异步处理。

134

## 与现有其他模式的关系

软件模式文档是一种财富，今天这种财富已经相当充裕易得了。很多种专著都记录了模式，而这些模式处于不同的抽象层次上。有架构模式、设计模式、分析模式以及编程模式。其中最受欢迎、影响最大的还是《设计模式：可重用面向对象软件的基础》一书，通常称为“四人帮”或GoF书。GoF书中的模式描述了面向对象设计中的专家解决方案。我们还参考了Martin Fowler所著《企业应用架构模式》[PEAA]一书中的模式。

我们的模式目录包括的模式，描述了应用程序的结构和设计元素。这个模式目录的基调主题是对J2EE平台的支持。目录中有一些模式是基于其他文献中的模式或者与其相关的。在这些情况下，为了体现这种关系，我们会为特定的J2EE模式按照现有其他模式的名称命名，并且/或者在模式介绍结尾的“相关模式”部分列出参考文献和引文。举例来说，有些模式是基于GoF书中的模式的，但是放在J2EE语境中考虑。在这种情况下，该J2EE模式的名称中就包括GoF书中模式的名字，并且在“相关模式”部分会出现对GoF书的参照。

## 模式路线图

这里我们列出了架构师在创建J2EE解决方案时会遇到的一些常见需求。我们先用简短地描述出需求或动机，而后给出一个或多个能够满足该需求的模式。虽然这个清单并非穷尽了所有需求，我们仍希望它有助于你根据自己的需要快速选取合适的模式。

表5-6描述了表现层模式通常实现的功能，并且指明了哪个模式提供哪种解决方案。

表5-6 表现层模式

如果你在找这个	请在这里找
对请求作预处理或后处理	模式 拦截过滤器
在请求处理中加入日志、调试以及其他操作	模式 前端控制器 模式 拦截过滤器
集中对请求处理的控制	模式 前端控制器 模式 拦截过滤器 模式 应用控制器
为了降低控制器组件和助手组件之间的耦合度，创建一种通用的命令接口或者一种context对象	模式 前端控制器 模式 应用控制器 模式 Context对象
控制器应该用servlet还是JSP实现	模式 前端控制器

135

(续)

如果你在找这个	请在这里找
从多个子视图创建一个视图	<b>模式 复合视图</b>
视图应该用servlet还是JSP实现	<b>模式 视图助手</b>
如何划分视图和模型	<b>模式 视图助手</b>
在哪里封装表现层相关的格式化数据逻辑	<b>模式 视图助手</b>
助手组件应该用JavaBean还是定制标记实现	<b>模式 视图助手</b>
合并多个表现层模式	<b>模式 拦截过滤器</b> <b>模式 分配器视图</b>
在哪里封装用于选择、分派视图的视图管理 /导航逻辑	<b>模式 服务到工作者</b> <b>模式 分配器视图</b>
在哪里保存会话状态	<b>设计考虑 “在客户端保存会话状态”</b> <b>设计考虑 “在表现层保存会话状态”</b> <b>设计考虑 “在业务层保存会话状态”</b>
136 控制客户端对特定视图或子视图的访问	<b>设计考虑 “控制客户端访问”</b> <b>重构 “对客户端隐藏资源”</b>
控制对应用系统请求的流程	<b>设计考虑 “重复的表单提交”</b> <b>重构 “引入同步器令牌”</b>
控制重复的表单提交	<b>设计考虑 “重复的表单提交”</b> <b>重构 “引入同步器令牌”</b>
利用<jsp:setProperty> 的JSP标准属性复制机制时、包含的设计问题	<b>设计考虑 “助手类属性——完整性和一致性”</b>
降低表现层和业务层之间的耦合	<b>重构 “对业务层隐藏表现细节”</b> <b>重构 “引入业务代表”</b>
分离数据访问代码	<b>重构 “分离数据访问代码”</b>

表5-7描述了表现层模式实现的功能，并且指明了可以从哪个模式或哪组模式中获得解决方案。

表5-7 业务层模式

如果你在找这个	请在这里找
尽量减小表现层和业务层之间的耦合	<b>模式 业务代表</b>
137 为客户端缓存业务服务	<b>模式 业务代表</b>
隐藏业务服务的寻址/创建/访问的实现细节	<b>模式 业务代表</b> <b>模式 服务定位器</b>
隔离服务寻址中对特定厂商/技术的依赖	<b>模式 服务定位器</b>
为业务服务的寻址和创建提供统一的方法	<b>模式 服务定位器</b>
隐藏EJB和JMS组件寻址的复杂度和依赖性	<b>模式 服务定位器</b>
在业务对象和客户端之间、不同层之间传输 数据	<b>模式 传输对象</b>
为远程客户端提供简单、统一的接口	<b>模式 业务代表</b> <b>模式 会话门面</b> <b>模式 应用服务</b>
为访问业务层组件提供粗粒度方法、从而 减少远程方法调用	<b>模式 会话门面</b>
管理EJB组件之间的关系，隐藏交互复杂度	<b>模式 会话门面</b>

(续)

如果你在找这个	请在这里找	
避免使业务层组件直接暴露给客户端	模式 会话界面 模式 应用服务	138
为访问业务层组件提供统一的边界	模式 会话界面 模式 应用服务	
用对象实现复杂的业务领域概念模型	模式 业务对象	
为业务对象和entity bean的设计找出粗粒度对象和从属对象	模式 业务对象 模式 复合实体	
设计粗粒度entity bean	模式 复合实体	
减小或消除entity bean客户端对数据库结构的依赖	模式 复合实体	
减小或消除entity bean之间的远程关系	模式 复合实体	
减少entity bean数量，提高可维护性	模式 复合实体	
从多个业务层组件中获取应用数据模型	模式 传输对象组装器	
轻松构造应用数据模型	模式 传输对象组装器	
对客户端隐藏数据模型构造的复杂性	模式 传输对象组装器	
提供业务层查询和结果列表处理	模式 值列表处理器	
尽可能减小使用EJB finder方法带来的负载	模式 值列表处理器	
在服务器端为客户端提供具有向前/后导航功能的查询结果缓存	模式 值列表处理器	
选用有状态session bean还是无状态session bean的权衡	设计考虑 “Session Bean——无状态vs.有状态”	
保护entity bean、禁止客户端直接访问	重构 “用session bean包装entity bean”	139
封装业务服务，隐藏业务层实现细节	重构 “引入业务代表”	
在entity bean中包含业务逻辑	设计考虑 “Entity Bean中的业务逻辑” 重构 “将业务逻辑移至session bean”	
把session bean作为粗粒度业务服务提供	重构 “合并session bean” 重构 “用session bean包装entity bean”	
减少和/或消除由entity bean之间的通信引起的网络和容器负担	重构 “减少entity bean之间的通信”	
分离数据访问代码	重构 “分离数据访问代码”	

表5-8描述了集成层模式通常实现的功能，并且指明了哪个模式提供哪种解决方案。

表5-8 集成层模式

如果你在找这个	请在这里找
尽可能减小业务层和资源层之间的耦合	模式 数据访问对象
集中对资源层的访问	模式 数据访问对象
尽可能减小业务层组件中访问资源的复杂度	模式 数据访问对象
为企业应用提供异步处理能力	模式 服务激活器
向业务层发送异步请求	模式 服务激活器

(续)

	如果你在找这个	请在这里找
140	把请求作为一组并行任务异步处理	模式 服务激活器
	透明地实现对象模型持久化	模式 业务领域存储
	实现定制持久化框架	模式 业务领域存储
	用XML和标准Internet协议暴露WebService	模式 Web Service中转
	把现存服务聚合、中转为WebService	模式 Web Service中转

## 小结

到此为止，我们已经看到了J2EE模式背后的基本概念，了解了按系统分层为模式分类的方式，考察了不同模式之间的关系，并且也见到了那一张有助于指导你使用特定模式的路线图。在后面的几章中，我们将逐个考察模式。

后面三章分别讨论表现层、业务层和集成层模式，读者可以就感兴趣的模式专门研读相关

141

章节。

# 第6章 表现层模式

本章将涉及以下主题:

- 拦截过滤器
- 前端控制器
- Context对象
- 应用控制器
- 视图助手
- 复合视图
- 服务到工作者
- 分配器视图

142  
143

## 拦截过滤器

### 问题

要在请求被处理之前、之后拦截并操作一个请求和它的响应。

所谓请求的预处理和后处理，是指在请求的核心处理操作之前、之后进行的操作。这之中有些操作决定了处理过程是否还要继续，另一些则把输入、输出数据流转化成适合进一步处理的形式。比如：

- 客户端是否有一个有效的会话？
- 请求的目录路径是否违反了限制条件？
- 系统是否支持客户端的浏览器类型？
- 客户端是用哪一种编码方式发送数据的？
- 请求数据流是否加密？是否压缩？

对于这些（以及其他很多）请求处理过程，一种常见的处理思路是用一长串的条件检查实现，通常会有嵌套的if/then/else语句来控制执行的流程。但是，既然在每个处理语句中执行的都是相似的操作，就会生成许多重复代码。这样进行预处理和后处理导致代码质量脆弱、满是复制-粘贴风格的程序，因为控制流程以及特定的处理操作都和应用逻辑混在了一起。

而且，这样的做法还会加大预处理/后处理组件与核心应用处理代码之间的耦合。

### 约束

- 要在多个请求之间实现集中、通用的处理，比如检查每个请求的数据编码方式、为每个请求留下日志信息或压缩输出响应。
- 要在预处理/后处理组件与请求处理核心服务之间实现松耦合，这样再加入/删除预处理/后处理操作就很容易，不会干扰核心服务。
- 要使预处理/后处理组件之间相互独立，每个组件都能自足存在，从而增进重用。

144

### 解决方案

使用拦截过滤器，作为一个可插拔式的过滤器，实现请求、响应的预处理和后处理。另有一个过滤器管理器，负责把各个处于松耦合关系的过滤器结合成一个链，并把控制依次委派给合适的过滤器。这样一来，不必改动现有代码就能够以各种方式加入、删除、合并这些过滤器。

就像其他各种系统中一样，如果系统中存在大量重复任务，就应该找到重复的操作，提取出来，并且使它的粒度变得足够粗，以便能够在多处应用。这就是可插拔式过滤器的作用。

这样就可以给主要处理过程加入各种常见服务，比如日志、调试等等。过滤器独立于主要应用逻辑的代码，所以可以通过部署声明加入或删除过滤器。

通过部署描述符中的声明来控制过滤器，正如“Servlet技术规范”2.3版中描述的一样。“Servlet技术规范2.3版”中包括一种标准机制，可以用来构造过滤器链，并且可以从链中加入/

删除过滤器而不会干扰核心服务。部署配置文件构造了一个过滤器链，并且可以把特定的URL映射到这个过滤器链上。当客户端请求的资源符合配置中被映射的URL时，链中的过滤器，就在系统处理该请求的目标资源之前（预处理）和/或之后（后处理）被调用了。

## 结构

图6-1是拦截过滤器的类图。

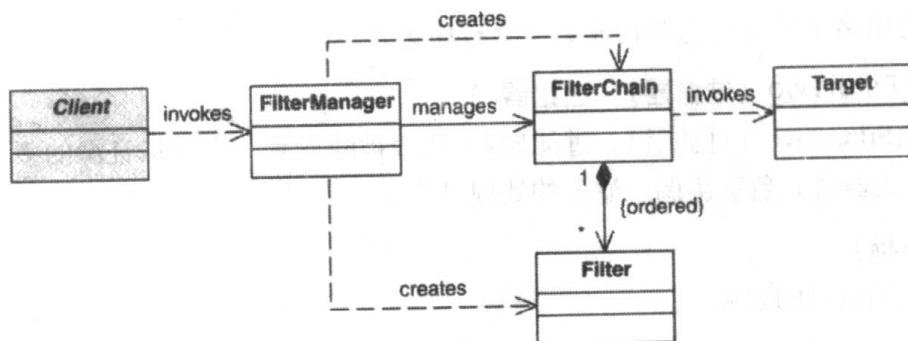


图6-1 拦截过滤器的类图

145

## 参与者和责任

图6-2是拦截过滤器的序列图。

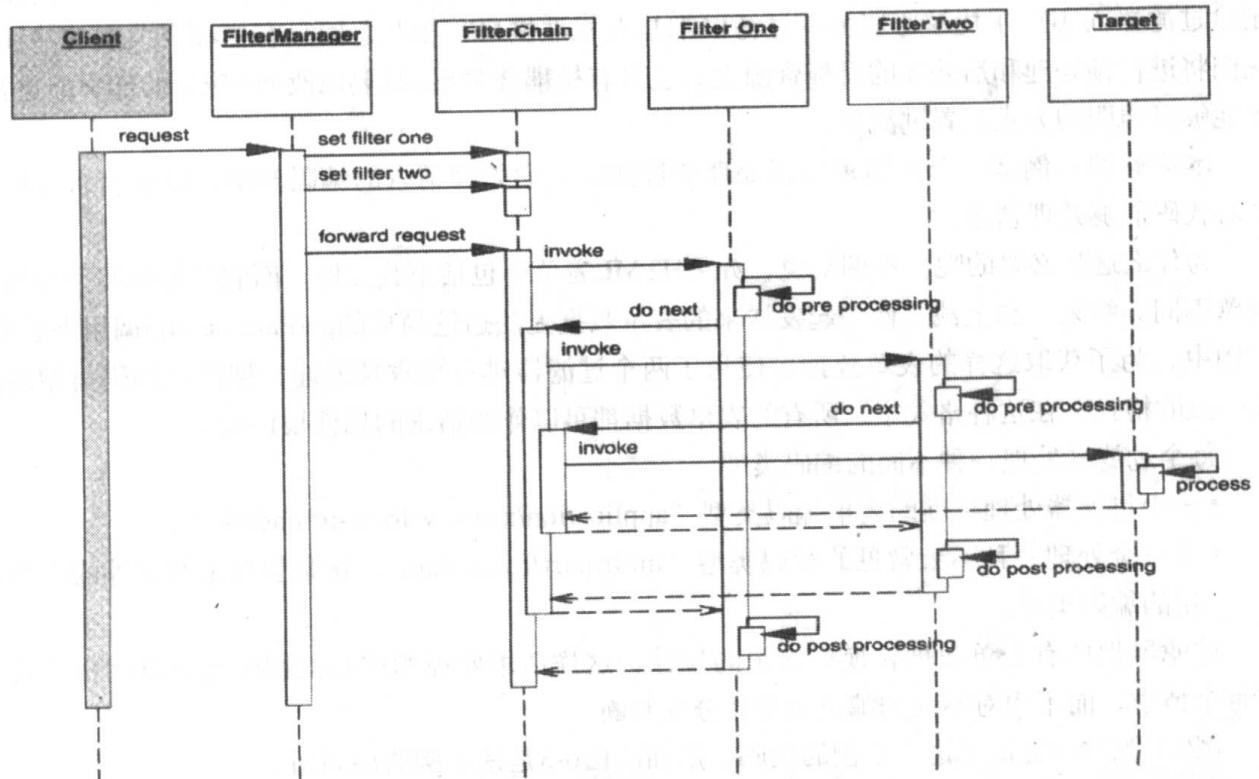


图6-2 拦截过滤器序列图

**Client (客户端)**

客户端把请求发送到FilterManager (过滤器管理器)。

**FilterManager (过滤器管理器)**

过滤器管理器负责管理过滤器的处理过程。它用合适的过滤器、按正确的顺序创建FilterChain (过滤器链)，并初始化处理过程。

**FilterChain (过滤器链)**

过滤器链是由多个独立过滤器构成的一个有序集合。

**FilterOne, FilterTwo (过滤器1, 过滤器2)**

FilterOne和FilterTwo (过滤器1、过滤器2) 代表被映射到一个目标资源的多个独立过滤器。

**146** FilterChain (过滤器链) 负责协调过滤器的处理过程。

**Target (目标)**

目标是客户端请求的资源。

**策略****标准过滤器策略**

通过部署描述符中的声明来控制过滤器，正如“Servlet技术规范”2.3版中描述的一样。“Servlet技术规范2.3版”中包括一种标准机制，可以用来构造过滤器链，并且可以从链中加入/删除过滤器而不会干扰核心服务。过滤器都具有标准接口，相互之间只有松耦合关系，并且与它们将进行预处理和后处理的目标资源之间也只有松耦合关系。只要修改web应用的部署描述符，就能够以声明的方式配置过滤器。

本策略的示例是一个对请求作预处理的过滤器，它处理请求的编码类型，以便请求处理的核心代码能够处理请求。

为什么这是必要的呢？举例来说，如果HTML表单中包括上载文件，而它的编码类型与基本表单不同。结果，和上载文件一起发送来的表单数据无法通过简单的getParameter()调用来获取。本例中，为了获取这样的表单数据，使用了两个过滤器进行请求预处理，把所有编码类型转换成一致的格式。在这种格式中，所有的表单数据都可以作为请求的属性被访问。

每个过滤器处理一种不同的编码类型：

- 一个过滤器处理标准的表单编码类型 “application/x-www-form-urlencoded”。
- 另一个处理一种不太常见的编码类型 “multipart/form-data”，这是包含上载文件的表单所用的编码类型。

过滤器把所有表单数据转换成请求的属性，这样请求处理的核心机制就能够以同样方式处理每个请求，而不用对不同的编码类型作分支判断了。

例6.1到例6.4提供了这个示例的代码，后面的图6-3是该示例的序列图。

例6.1中给出了基本过滤器，体现了标准过滤器的默认回调方法操作。

### 例6.1 基本过滤器——标准过滤器策略

```

1  public class BaseEncodeFilter implements javax.servlet.Filter {
2
3      private javax.servlet.FilterConfig filterConfig;
4
5      public void doFilter(
6          javax.servlet.ServletRequest servletRequest,
7          javax.servlet.ServletResponse servletResponse,
8          javax.servlet.FilterChain filterChain)
9          throws java.io.IOException,
10         javax.servlet.ServletException {
11
12     filterChain.doFilter(servletRequest, servletResponse);
13 }
14
15     protected javax.servlet.FilterConfig getFilterConfig() {
16         return filterConfig;
17     }
18
19     public void destroy() { }
20
21     public void init(javax.servlet.FilterConfig filterConfig)
22         throws javax.servlet.ServletException {
23         this.filterConfig = filterConfig;
24     }
25
26 }
```

147

例6.2是一个处理常见应用表单编码格式，转换请求数据的过滤器。

### 例 6.2 StandardEncodeFilter——标准过滤器策略

```

1  public class StandardEncodeFilter extends BaseEncodeFilter {
2      // Creates new StandardEncodeFilter
3      public StandardEncodeFilter() { }
4
5      public void doFilter(javax.servlet.ServletRequest
6          servletRequest, javax.servlet.ServletResponse
7          servletResponse, javax.servlet.FilterChain filterChain)
8          throws java.io.IOException,
9         javax.servlet.ServletException {
10
11     String contentType = servletRequest.getContentType();
12     if ((contentType == null) || contentType.equalsIgnoreCase(
13         "application/x-www-form-urlencoded")) {
14         translateParamsToAttributes(servletRequest, servletResponse);
15     }
16
17     filterChain.doFilter(servletRequest, servletResponse);
```

148

```

18 }
19
20 private void translateParamsToAttributes(
21     ServletRequest request, ServletResponse response) {
22     Enumeration paramNames = request.getParameterNames();
23
24     while (paramNames.hasMoreElements()) {
25         String paramName = (String) paramNames.nextElement();
26         String [] values;
27         values = request.getParameterValues(paramName);
28         if (values.length == 1)
29             request.setAttribute(paramName, values[0]);
30         else
31             request.setAttribute(paramName, values);
32     }
33 }
34 }

```

例6.3是一个处理多部分（multipart）编码格式，转换请求数据的过滤器。这些过滤器的代码基于servlet技术规范2.3版的最终稿。以上两个过滤器类都继承了基本过滤器（见后面的“[基本过滤器策略](#)”）。

### 例6.3 MultipartEncodeFilter —— 标准过滤器策略

```

1 public class MultipartEncodeFilter extends BaseEncodeFilter {
2     public MultipartEncodeFilter() { }
3     public void doFilter(
4         javax.servlet.ServletRequest servletRequest,
5         javax.servlet.ServletResponse servletResponse,
6         javax.servlet.FilterChain filterChain)
7         throws java.io.IOException,
8         javax.servlet.ServletException {
9
10    String contentType = servletRequest.getContentType();
11    // 只有当请求是multipart编码的时候
12    // 才做过滤
13    if (contentType.startsWith("multipart/form-data")) {
14        try {
15            String uploadFolder = getFilterConfig().
16                getInitParameter("UploadFolder");
17            if (uploadFolder == null)
18                uploadFolder = ".";
19            /** The MultipartRequest class is:
20             * Copyright (C) 2001 by Jason Hunter
21             * <jhunter@servlets.com>. All rights reserved.
22             */
23            MultipartRequest multi = new MultipartRequest(
24                servletRequest, uploadFolder, 1 * 1024 * 1024 );
25            Enumeration params = multi.getParameterNames();

```

```

26     while (params.hasMoreElements()) {
27         String name = (String)params.nextElement();
28         String value = multi.getParameter(name);
29         servletRequest.setAttribute(name, value);
30     }
31     Enumeration files = multi.getFileNames();
32     while (files.hasMoreElements()) {
33         String name = (String)files.nextElement();
34         String filename = multi.getFileSystemName(name);
35         String type = multi.getContentType(name);
36         File f = multi.getFile(name);
37         // 这里如果必要，可以进行
38         // 文件操作
39     }
40     } catch (IOException e) {
41         LogManager.logMessage("error reading or saving file"+ e);
42     }
43 } // end if
44 filterChain.doFilter(servletRequest, servletResponse);
45 } // end method doFilter()
46 }

```

例6.4中给出的是包括了这个示例的那个web应用的部署描述符片段。这个片段中表现了上述两个过滤器如何被注册，然后又映射到相关资源上（这个例子里只是映射到一个简单的测试servlet上）。

#### 例6.4 部署描述符——标准过滤器策略

```

1 .
2 .
3 .
4 <filter>
5   <filter-name>StandardEncodeFilter</filter-name>
6   <display-name>StandardEncodeFilter</display-name>
7   <description></description>
8   <filter-class> corepatterns.filters.encodefilter.
9       StandardEncodeFilter</filter-class>
10 </filter>
11 <filter>
12   <filter-name>MultipartEncodeFilter</filter-name>
13   <display-name>MultipartEncodeFilter</display-name>
14   <description></description>
15   <filter-class>corepatterns.filters.encodefilter.
16       MultipartEncodeFilter</filter-class>
17   <init-param>
18     <param-name>UploadFolder</param-name>
19     <param-value>/home/files</param-value>
20   </init-param>
21 </filter>

```

```

22 .
23 .
24 .
25 <filter-mapping>
26   <filter-name>StandardEncodeFilter</filter-name>
27   <url-pattern>/EncodeTestServlet</url-pattern>
28 </filter-mapping>
29 <filter-mapping>
30   <filter-name>MultipartEncodeFilter</filter-name>
31   <url-pattern>/EncodeTestServlet</url-pattern>
32 </filter-mapping>
33 .
34 .
35 .

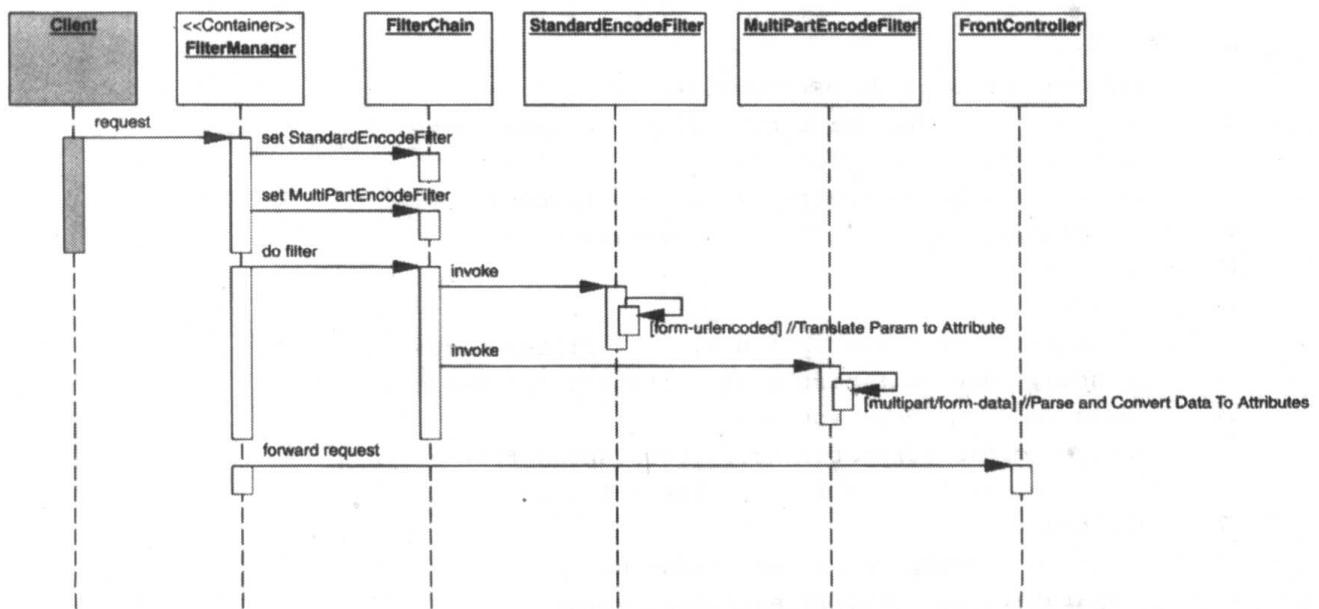
```

当客户端对控制器servlet发送请求时，上面的StandardEncodeFilter（标准编码过滤器）和MultiPartEncodeFilter（多部分编码过滤器）拦截了请求。Web容器充当了过滤器管理器的角色，通过调用过滤器的doFilter方法，把控制权转发给过滤器。每个过滤器完成了处理之后，就把控制权转交给它所属的FilterChain（控制器链）。过滤器指示FilterChain（控制器链）执行下一个过滤器操作。当两个过滤器都接收并交还了控制权之后，下一个接到控制权的组件才是实际的目标资源，在本示例中也就是那个控制器servlet。

Servlet技术规范2.3版中支持的过滤器还能够包装请求和响应对象。使用这个功能可以构造出一个非常强大的机制，甚至比后面的“定制过滤器策略”利用定制实现构造的机制还要强大。当然，还可以定制实现一个这两种策略的混合体，但是即使如此，做出的实现较之servlet技术规范支持的那种标准过滤器策略，也许还是不够强大。

151

本示例的序列图如图6-3所示。



152

图6-3 拦截过滤器，标准拦截策略——编码转换示例

## 定制过滤器策略

使用你自己的定制策略实现过滤器。与servlet 2.3以上版本规范中支持的、由容器提供的首选标准过滤器策略相比，这个策略没有前者灵活、强大。它之所以不够强大，是因为它不能以一种标准和可移植的方式支持对请求和响应对象的包装。而且不能够修改请求对象，当过滤器要控制输出流的时候，还必须引入某种形式的缓存机制。

要实现定制过滤器策略，可以用装饰器模式（Decorator pattern）[GoF]，在请求处理的核心逻辑之外再包装上过滤器。比如，可以在一个身份认证过滤器之外再包装一个调试过滤器。

例6.5和例6.6给出了编程实现这个机制的例子。

### 例6.5 实现过滤器——调试过滤器

```

1  public class DebuggingFilter implements Processor {
2      private Processor nextProcessor;
3
4      public DebuggingFilter(Processor nextProcessor) {
5          this.nextProcessor = nextProcessor;
6      }
7
8      public void execute(ServletRequest request,
9              ServletResponse response)
10             throws IOException, ServletException {
11         // 在这里可以作一些过滤处理
12         // 比如显示请求的参数等等
13         nextProcessor.execute(request, response);
14     }
15 }
16 }
```

### 例6.6 请求处理

```

1  public void processRequest(ServletRequest req, ServletResponse res)
2      throws IOException, ServletException {
3      Processor processors = new DebuggingFilter(
4          new AuthenticationFilter(new CoreProcessor()));
5      processors.execute(req, res);
6
7      //然后把控制分派到下一个资源
8      //可能就是要显示的视图
9      dispatcher.dispatch(req, res);
10 }
11 }
```

153

在servlet控制器中，把控制权委派给ProcessRequest方法，用以处理输入请求，如例6.7所示。

### 例6.7 实现过滤器——核心处理器

```

1  public class CoreProcessor implements Processor {
2
3      public CoreProcessor() { }
4 }
```

```

5     public void execute(ServletRequest req, ServletResponse res) {
6         throws IOException, ServletException {
7             //这里实现核心处理机制
8         }
9     }

```

仅仅是为了示例需要，假设每个处理组件在执行中都向标准输出写一句消息。一种可能的执行输出如例6.8所示。

### 例6.8 写到标准输出里的消息

```

1 Debugging filter preprocessing completed...
2 Authentication filter processing completed...
3 Core processing completed...
4 Debugging filter post-processing completed...

```

例6.8显示的是当一个处理器链被依次执行的时候的标准输出。链中的每个处理器（除了最后一个）都被视为一个过滤器。最后一个处理器组件也就是封装核心处理过程的地方（这个核心处理过程实际完成每个请求）。如果采用这种设计，当需要修改请求处理方式的时候，就不仅要修改所有的过滤器类，还要修改CoreProcessor（核心处理器）类的代码。

图6-4是例6.5、例6.6、例6.8中的过滤器控制流程的序列图

请注意，使用装饰器模式的时候，每个过滤器都有一个通用的接口，需要直接调用下一个过滤器。可以使用FilterManager（过滤器管理器）和FilterChain（过滤器链），用另一种方法实现这个策略。使用这两个组件之后，由它们负责协调、管理过滤处理，每个单独的过滤器用不着和其他过滤器直接通信。

154

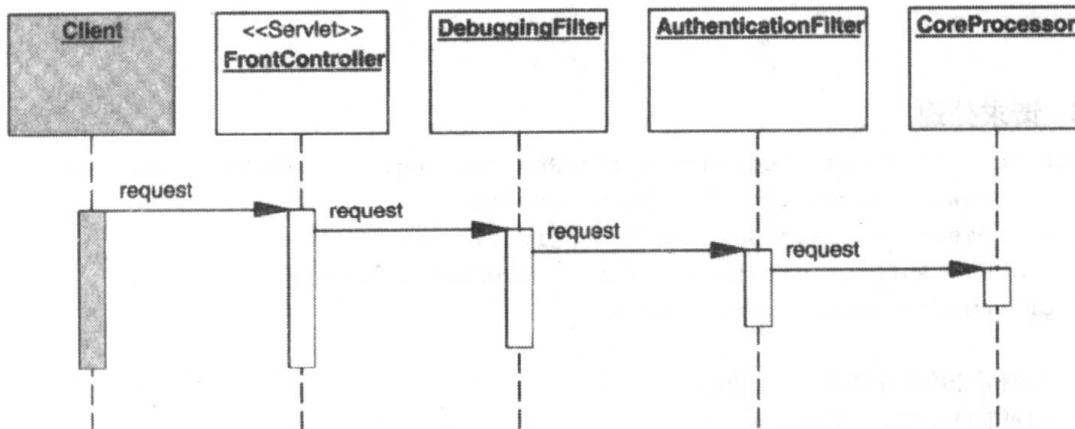


图6-4 定制过滤器策略（装饰器实现）的序列图

采用了FilterManager（过滤器管理器）和FilterChain（过滤器链）的实现与servlet 2.3技术规范的做法非常接近，虽然它还属于一种定制策略。例6.9就是这样一个FilterManager类，它创建了一个FilterChain；相应的FilterChain如例6.10所示。FilterChain按次序把过滤器加入链中，进行过滤处理，并且最终处理目标资源。（为了简洁，把过滤器加入链的工作放在了FilterChain的构造函数中，但通常应该在代码注释的地方进行。）

后面的图6-5是这段代码的序列图。

### 例6.9 FilterManager——定制过滤器策略

```

1  public class FilterManager {
2      public void processFilter(Filter target,
3          javax.servlet.http.HttpServletRequest request,
4          javax.servlet.http.HttpServletResponse response)
5              throws javax.servlet.ServletException,
6                  java.io.IOException {
7
8      FilterChain filterChain = new FilterChain();
9      // 必要时过滤器管理器在这里构造过滤器链
10     // 把请求交给过滤器链处理
11     filterChain.processFilter(request, response);
12
13     // 处理目标资源
14     target.execute(request, response);
15 }
16 }
```

155

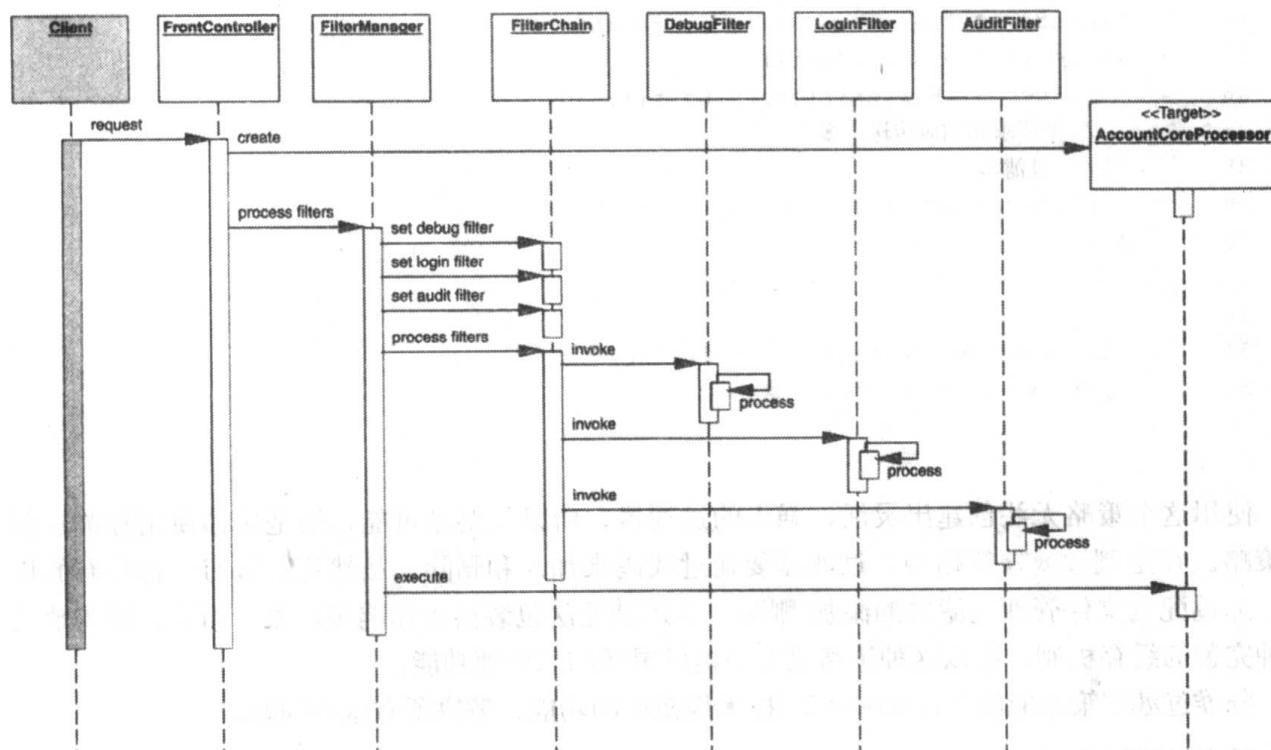


图6-5 定制过滤器策略（不用装饰器的实现）的序列图

156

### 例6.10 FilterChain——定制过滤器实现

```

1  public class FilterChain {
2      // 过滤器链
3      private List myFilters = new ArrayList();
4
5      // 创建新的 FilterChain
```

```

6     public FilterChain() {
7         // 加入默认过滤器服务（只是为了示例简单起见）。
8         // 这一处理本来应该放在FilterManager 中,
9         // 之所以放在这里
10        // 只是因为这是一个示例。
11        addFilter(new DebugFilter());
12        addFilter(new LoginFilter());
13        addFilter(new AuditFilter());
14    }
15
16    public void processFilter(
17        javax.servlet.http.HttpServletRequest request,
18        javax.servlet.http.HttpServletResponse response)
19        throws javax.servlet.ServletException,
20        java.io.IOException {
21
22        Filter filter;
23        // 应用过滤器
24        Iterator filters = myFilters.iterator();
25        while (filters.hasNext()) {
26            filter = (Filter)filters.next();
27            // 把请求和响应传送给多个
28            // 过滤器
29            filter.execute(request, response);
30        }
31    }
32
33    public void addFilter(Filter filter) {
34        myFilters.add(filter);
35    }
36 }

```

使用这个策略无法创建出灵活、强大的过滤器，所以只要是可能，还是应该使用标准过滤器策略。在定制过滤器策略中，过滤器要通过代码来加入和删除。虽然可以编写一种独有的机制，通过配置文件管理过滤器的添加/删除，但还是无法包装请求和响应对象。而且，因为缺乏一种完善的缓存机制，所以这种策略也无法提供灵活的后处理功能。

**157 标准过滤器策略能够利用servlet 2.3技术规范中的功能，解决所有这些问题。**

### 基本过滤器策略

一个基本过滤器是所有过滤器的超类。过滤器的共通功能可以封装在基本过滤器里，在所有过滤器中得到共享。比如，对于标准过滤器策略中的容器回调函数的默认操作，就大可以在基本过滤器中实现。例6.11给出了实现这一操作的一种方法。

#### 例6.11 基本过滤器策略

```

1  public class BaseEncodeFilter implements javax.servlet.Filter {
2      private javax.servlet.FilterConfig filterConfig;
3

```

```

4     public BaseEncodeFilter() { }
5
6     public void init(javax.servlet.FilterConfig filterConfig) {
7         this.filterConfig = filterConfig;
8     }
9
10    public void doFilter(javax.servlet.ServletRequest servletRequest,
11                          javax.servlet.ServletResponse servletResponse,
12                          javax.servlet.FilterChain filterChain)
13                          throws java.io.IOException,
14                          javax.servlet.ServletException {
15
16        filterChain.doFilter(servletRequest, servletResponse);
17    }
18
19    protected javax.servlet.FilterConfig getFilterConfig() {
20        return filterConfig;
21    }
22 }
```

### 模板过滤器策略

这个策略的一个主要优点在于，它能够抽象出servlet API的所有底层细节，专注于实现预处理和后处理的所有逻辑。这就是说，你不再需要关心方法调用的语义，比如chain.doFilter(req, res)等，只要使用简单的方法调用，比如doPreProcessing(req, res)，后者当然直观得多。

这种策略是基于基本过滤器策略的，基本过滤器作为一个基类，封装了过滤器API的所有细节（见本章前面的基本过滤器策略）。模板过滤器策略使用这个积累提供了模板方法[GOF]的功能。在这一策略中，基本过滤器规定了每个过滤器都要完成的大致步骤，每个过滤器子类确定具体怎样完成这些步骤。这种实现思路体现了开发框架中常见的控制反转——也就是说，由超类（或基类）规定子类的控制流程。通常，超类中的方法都是粗粒度的、基本的方法，给模板规定了一个专门的结构。

这种策略可以和任何其他策略混合使用。例6.12和例6.13示范了如何将这个策略与基本过滤器策略结合使用。

例6.12是一个称为TemplateFilter的基类，如下所示。

#### 例6.12 使用模板过滤器策略

```

1  public abstract class TemplateFilter implements javax.servlet.Filter {
2      private FilterConfig filterConfig;
3      public void init(FilterConfig filterConfig) throws ServletException {
4          this.filterConfig = filterConfig;
5      }
6
7      protected FilterConfig getFilterConfig() {
8          return filterConfig;
9      }
10 }
```

```

11     public void doFilter(ServletRequest request,
12         ServletResponse response, FilterChain chain)
13         throws IOException, ServletException {
14
15     // 每个过滤器的预处理操作
16     doPreProcessing(request, response);
17
18     // 把控制权交给链中的下一个过滤器或
19     // 交给目标资源。这个方法调用也是
20     // 预处理过程和后处理之间的逻辑分界
21     chain.doFilter(request, response);
22
23     // 每个过滤器的后处理操作
24     doPostProcessing(request, response);
25 }
26     public abstract void doPreProcessing(ServletRequest request,
27         ServletResponse response) { }
28
29     public abstract void doPostProcessing(ServletRequest request,
30         ServletResponse response) { }
31
32     public void destroy() { }
33 }
```

这样定义了TemplateFilter类之后，每个作为子类的过滤器只需要doPreProcessing和doPostProcessing方法。但是子类也可以实现全部3个方法。例6.13中就是一个过滤器子类，实现了两个强制实现的方法（所谓强制，是因为超类把这两个方法声明为抽象方法）。

### 例6.13 负责记录日志的过滤器

```

1  public class LoggingFilter extends TemplateFilter {
2      public void doPreProcessing(ServletRequest req, ServletResponse res) {
3          // 在这里做预处理，比如在请求处理之前
4          // 记录一些与请求相关的信息。
5      }
6
7      public void doPostProcessing(ServletRequest req, ServletResponse res) {
8          // 在这里做后处理，比如在请求通过处理、
9          // 响应已经生成之后，记录一些与请求
10         // 相关的信息。
11     }
12 }
```

在图6-6的序列图中，过滤器子类（比如LoggingFilter）通过覆盖抽象方法doPreProcessing和doPostProcessing定义了专门的处理操作。作为结果，模板过滤器规定了每个过滤器的控制流程，这样过滤器开发者就可以专注于预处理/后处理逻辑的语义了。最后，因为模板过滤器是一个基本过滤器，它也封装了所有过滤器都要用到的代码。

图6-6是这个策略的序列图。

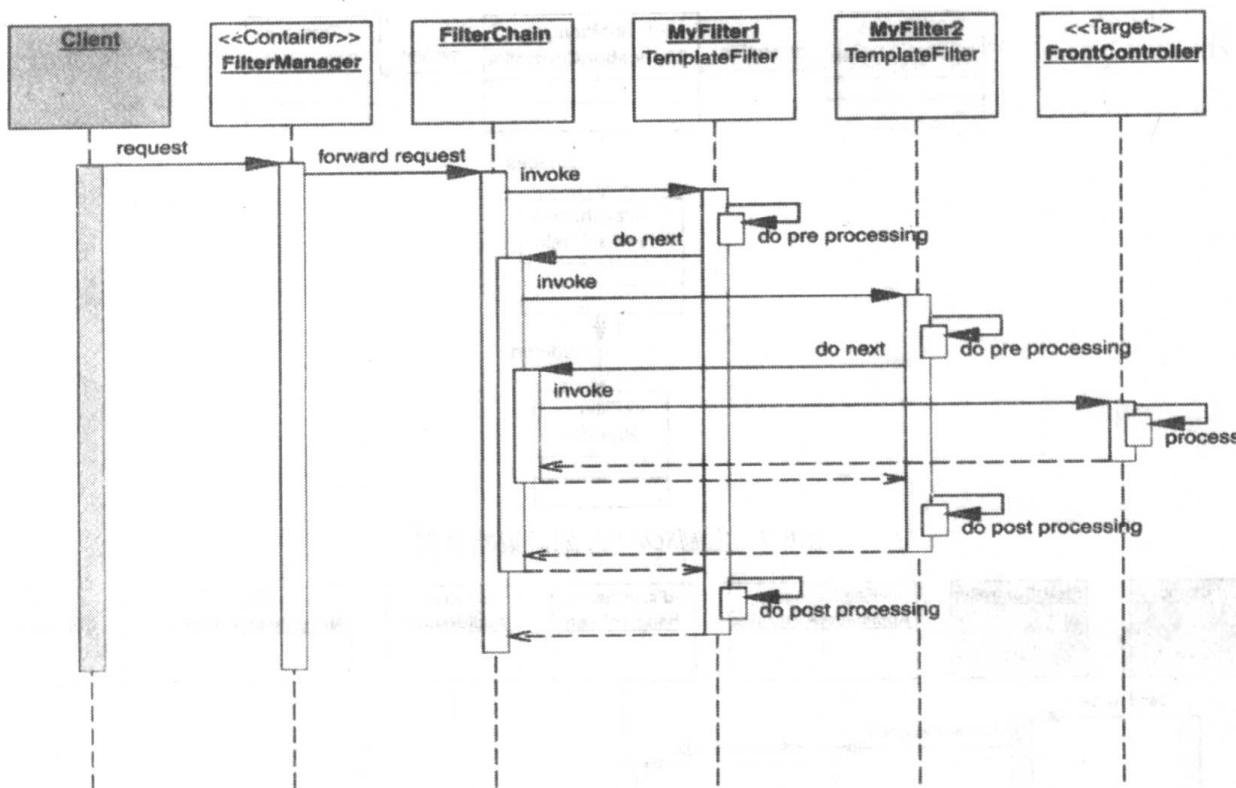


图6-6 拦截过滤器，模板过滤器策略序列图

### Web Service消息处理策略

虽然拦截过滤器的使用在表现层很常见，但在其他层面也会用到某种拦截过滤器策略。定制SOAP过滤器和JAX-RPC过滤器消息处理策略，就是在客户端、表现层和集成层使用过滤器的例子，它们主要用于Web Service请求的预处理和后处理。

过滤器链中的各个过滤器相互保持松耦合，用于处理和操作Web Service请求，所以也经常被称为“Web Service处理器”。这些过滤器拦截输入的消息，对消息中的信息进行预处理和后处理。

与此类似，过滤器还可以对输出的响应进行操作。比如，一个消息处理器可以验证数字签名，给消息中加上签名，或者记录消息日志。在每个消息进入和离开处理器机制的时候，消息都要在XML和Java对象之间相互转化。

用于Java的带附件的SOAP API (SOAP With Attachments API for Java, 又称SAAJ) [SAAJ] 提供了处理SOAP消息的功能，开发者可以使用它构造预处理/后处理过滤器。必要时应该使用 SOAP消息处理器；使用了它，就能获得非常强大的处理机制。对Web Service的进一步讨论，请参见本章的应用控制器和第8章的“Web Service中转”。

#### 定制SOAP过滤器策略

这个策略通常作为应用控制器的“定制SOAP消息处理策略”的一部分使用。使用了一个 Context对象来在不同的处理器之间共享SOAP消息的细节，因为处理器不能够直接引用其他处理器。

图6-7和6-8表现的是这个策略，图6-9表现的则是JAX-RPC过滤器策略，二者之间最大的不同在于：在这个策略中可以编写一个定制处理器，从而利用某种SOAP库（比如SAAJ）完成参与者的责任。

#### Web Service

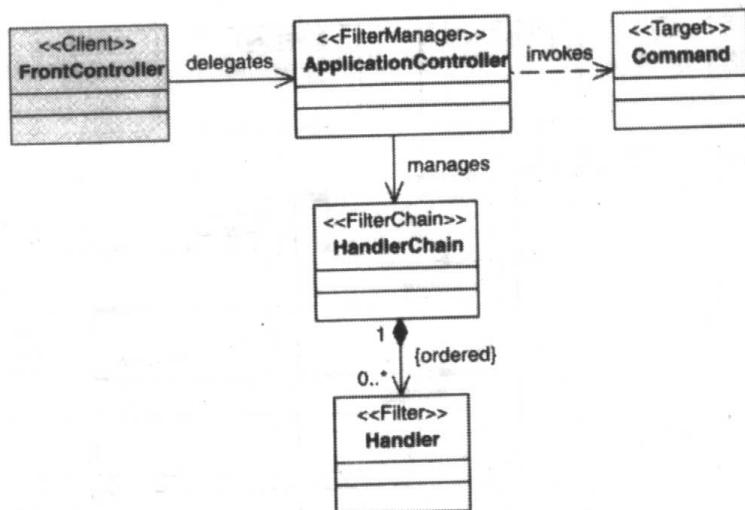


图6-7 定制SOAP过滤器策略类图

162

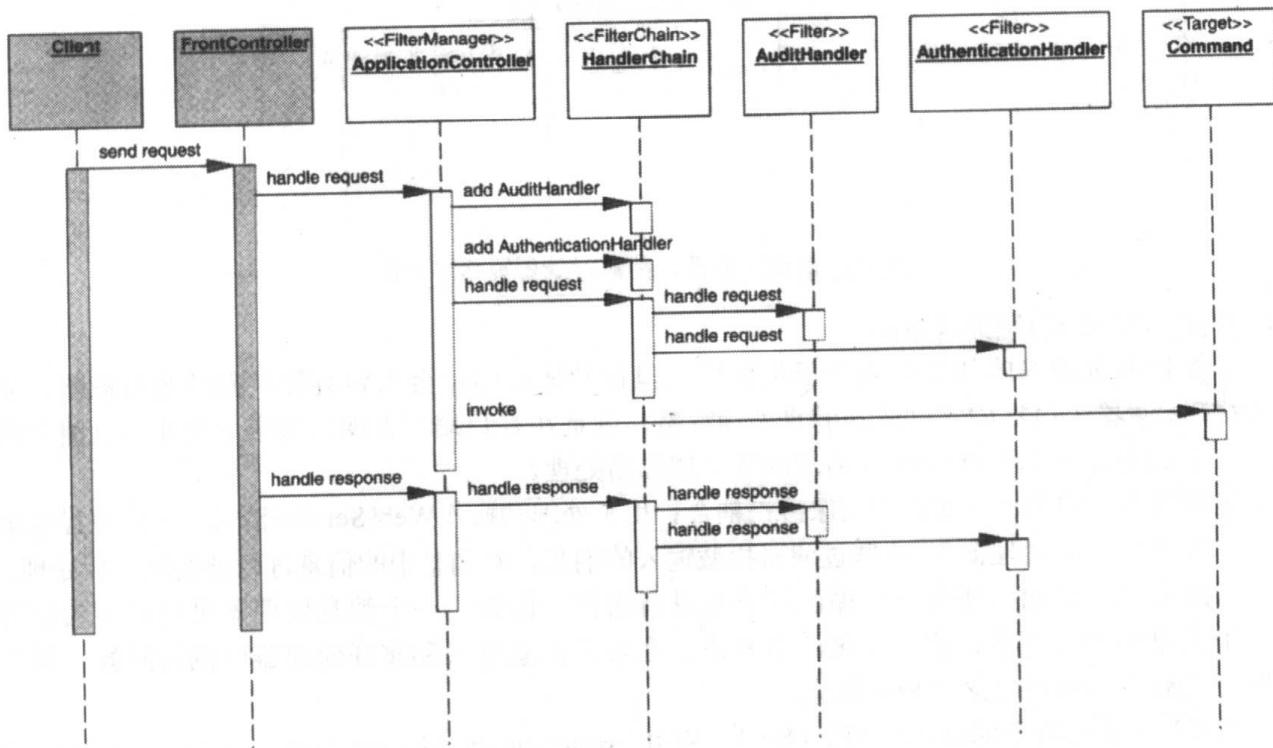


图6-8 定制SOAP过滤器策略序列图

### JAX-RPC过滤器策略

这个策略通常作为应用控制器中的“JAX-RPC消息处理策略”的一部分使用。使用了一个Context对象来在不同的处理器之间共享SOAP消息的细节，因为处理器不能够直接引用其他处理器。

这个策略和定制SOAP过滤器策略的主要区别在于，这个策略中，大多数参与者的责任都由JAX-RPC运行时引擎完成。处理器既可以在RPC调用的客户端实现，也可以在RPC调用的服务端实现。这样，在网络连接的两侧都可以加入处理器。使用SAAJ和JAX-RPC构造消息处理器的做法，如果应用得当就能提供强大的处理能力。

Web Service

163

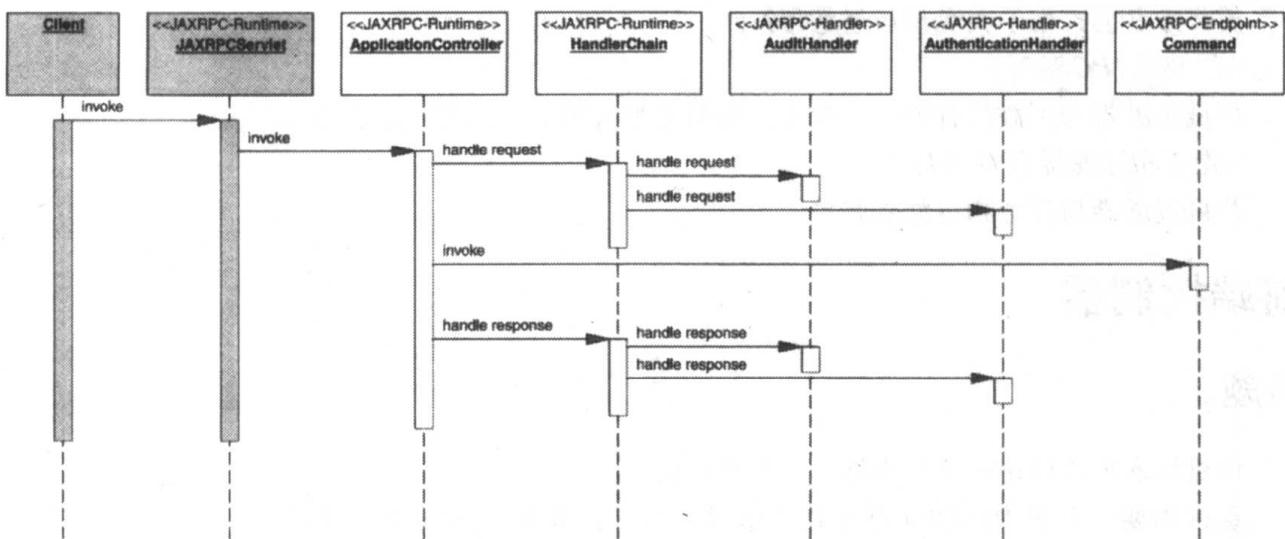


图6-9 JAX-RPC 过滤器策略序列图

164

## 效果

- 利用松耦合的处理器实现控制集中化

和控制器一样，过滤器也能够实现多个请求的集中处理。过滤器更适合发送请求和响应消息供目标资源如控制器进行最终处理。另外，控制器往往能够把不相干的多种常见服务（比如身份认证、日志、加密等等）的管理联系起来。而过滤器则允许处理器之间实现更松的耦合，处理器可以按照多种方式相互组合。

- 增进了重用

过滤器对应用系统作出了更清晰的划分，因此能够鼓励重用。你可以透明地加入、删除来自现成代码的可插拔式拦截器，而且所有拦截器都有标准接口，所以它们能够按照不同方式相互组合、实现重用。

- 能够灵活地通过声明配置

各种各样的服务能够按照不同方式组合起来，不需要对核心代码重新编译。

- 不便于信息共享

在过滤器之间共享信息不太方便，因为按照定义，过滤器之间只存在松耦合。如果必须在过滤器之间共享大量信息，那么这种做法可能代价甚高。

## 相关模式

- 前端控制器

前端控制器能解决类似的问题，但是它最好用来实现核心处理。

- 装饰器 [GoF]

拦截过滤器与装饰器有关，因为后者提供了动态可插拔的包装器。

- 模板方法 [GoF]

模板方法模式用于实现模板过滤器策略。

- **拦截器 [POSA2]**

拦截过滤器与拦截器有关，后者允许透明地加入服务，并能让服务自动触发。

- **管道和过滤器 [POSA1]**

拦截过滤器与管道和过滤器有关。

165

## 前端控制器

### 问题

你想给表现层的请求处理安排一个集中访问点。

系统需要一个集中的访问点来处理请求。如果没有集中访问点，那么多个请求之间共用控制代码就会在许多文件（比如视图文件）中重复出现。如果控制代码和视图创建代码混在一起，整个应用系统的模块化程度和内聚性都要下降。另外，如果控制代码在多处散放，也不便于代码维护，只要代码有一处改动，就需要改动多个文件。

### 约束

- 你想要避免重复控制逻辑。
- 你想要对多个请求采取共通的处理逻辑。
- 你想把系统处理代码和视图分割开。
- 你想要把系统访问点集中在一处。

## 解决方案

使用一个前端控制器，作为最初的接触点，用来处理所有相关请求。前端控制器集中了控制逻辑，避免了逻辑的重复，完成了主要的请求处理操作。

前端控制器为处理请求提供了一个集中的入口点。它能够集中控制逻辑，因此就减少了直接置入视图的代码量。比如，对于JSP视图来说，这就能避免把大量的Java代码（称为scriptlet代码）置入JSP页面中。

这个模式和拦截过滤器相似，因为二者都提取、合并了表现层的一些共通控制逻辑。一个主要的区别在于，拦截过滤器提供了一组松耦合的、结成一条链的处理器，以及一些特别适合进行预处理/后处理的强大策略。而使用前端控制器集中控制、减少视图中的业务、处理逻辑，就能够提高多个请求之间代码的重用度，减少代码重复。除非控制代码微乎其微，否则使用前端控制器就要优于在多个视图中置入代码，因为后者会导致一种“复制-粘贴”式的重用，而这更容易产生错误。

另外，如果在多个视图置入代码，还会使团队中的每个开发者用不同方法完成同样的任务——而这种处理本来应该是完全一致的，应该以统一、规范的方式集中完成。比如，如果由每个开发者在页面中加入<tag:checkLogin/>标记，实现对特定页面的访问控制，那么就很难保证一

166

致的效果，因为可能就有某个开发者忘了作这件事。

前端控制器通常会使用应用控制器，后者负责操作和视图管理（action and view management）。

- 操作管理，也就是定位特定的操作（action）、并把控制权路由到该操作上，由它处理请求。
- 视图管理，则是指找到合适的视图，并分派到该视图上。虽然这个任务也可以放进前端控制器完成，但是把它独立出来，成为应用控制器的部分功能，能够增进模块化、可维护性和可重用性。

虽然处理相关请求的工作集中了，但这并不意味着对系统中处理器数量做了任何限制。一种最常见的用法是由一个前端控制器处理所有请求，但是应用系统当然也可以有多个控制器，每一个控制器对应于一类服务。

对于前面只使用一个控制器的通常情况，服务器管理任务比较少，因为web服务器只需注册一个组件。这也意味着增加一种服务并不需要在web服务器上注册该服务。这减少了管理负担，更便于热部署。

而对于使用多控制器的情形，每个控制器都必须在web服务器上注册。增加一种新服务也需要在web服务器上注册。

### 设计手记：处理请求

处理请求需要两种操作：请求处理和视图处理。在请求处理过程中，表现层中又必须进行几种操作：

- 协议处理和上下文转换
- 导航和路由
- 核心处理
- 分派

协议处理是指处理与特定协议相关的请求，上下文转换是指把与特性协议相关的状态转换成更为通用的形式。一个例子是，从HttpServletRequest实例中获取参数，再复制到一个MyRequest对象中，从而能够在servlet环境之外独立应用。167

导航和路由是指确定一个特定请求的路由，比如用哪些对象进行请求的核心处理，用哪个视图显示结果。

核心处理也就是对请求的实际处理。

分派是指把控制权从应用系统的一个部分转交给另一个部分，比如从请求处理机制转交给视图处理组件。

### 结构

图6-10是前端控制器的类图。

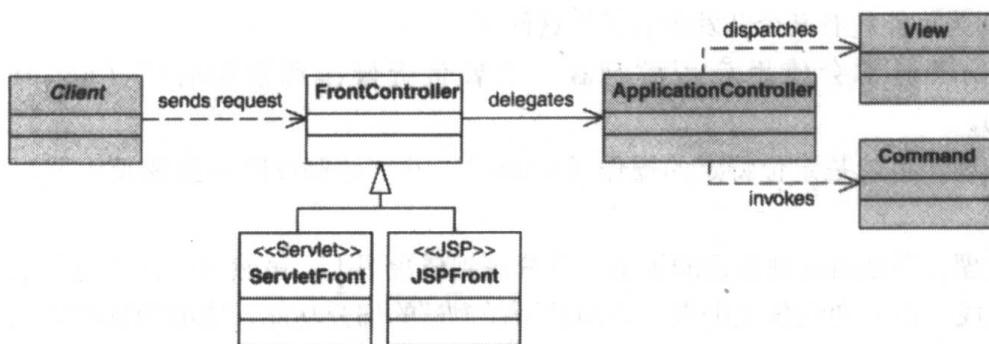


图6-10 前端控制器类图

前端控制器的通常用法，是采取“命令加控制器策略”，把它和应用控制器、命令（commands）结合使用。但是，当然也可以绕过应用控制器，直接从控制器中调用助手对象中类型确定<sup>①</sup>的方法。比如说，如果有一个本地的业务层，那么控制器就可以直接调用业务对象、应用服务或一个POJO门面，这时上述三者都充当了助手的角色。如果业务层是远程的，那么控制器可以调用业务代表，这时业务代表同样充当了助手。直接从前端控制器调用助手对象的场景，我们用另一张类图表示，见图6-11。

但是，无论是本地业务层或远程业务层，这样使用控制器，都要通过条件控制<sup>②</sup>来把输入请求解析到正确的助手对象上，而这种做法是应该限制使用的（只是在特定场景下才适合）。为了减少单个控制器中的条件控制，可以采取的一种做法是：提供多个控制器，每一个用于一组用例，比如一个AccountController、一个OrderController等。这样就能把不同条件下的控制集中起来，提高代码的可维护性，虽然这也意味着要加入多个servlet控制器。这样下来增加了管理、部署、维护的复杂度。

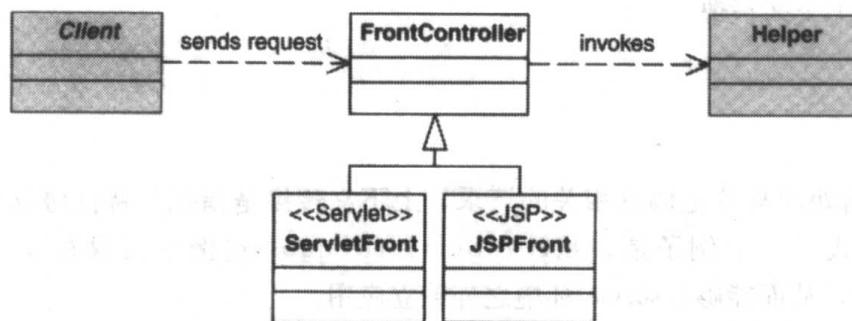


图6-11 直接调用助手类的前端控制器类图

## 参与者和责任

图6-12是前端控制器的序列图。

- ① 类型确定的方法：原文为**strongly typed methods**，之所以这里强调绕过应用控制器就可以调用“类型确定的方法”，是因为采用命令模式（command pattern）的时候，方法调用是动态的，类型并不确定。
- ② 条件控制：也就是if-then或者case之类的控制语句。

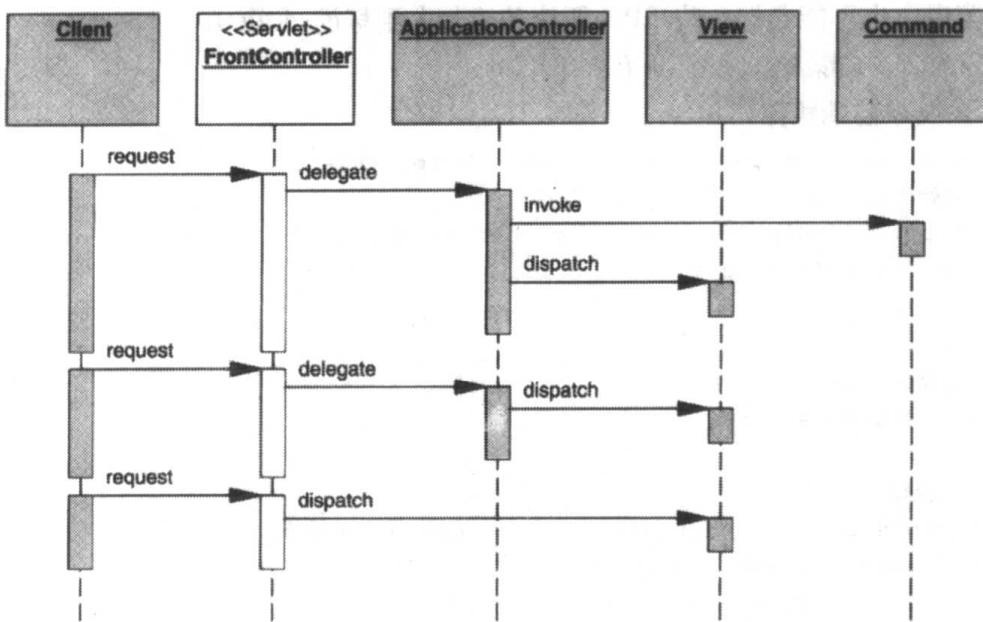


图6-12 前端控制器类图

### FrontController（前端控制器）

FrontController（前端控制器）是系统中负责处理请求的最初接触点。它委派一个 169 ApplicationController（应用控制器）来完成操作和视图管理。

### ApplicationController（应用控制器）

ApplicationController（应用控制器）负责操作和视图管理，包括定位负责请求的特定操作（action）并把控制权路由到该操作上；另外还包括找到合适的视图，并分派到该视图上。

### Command（命令）

命令执行用于处理请求的操作。

### View（视图）

视图是指返回给客户端的显示界面。

## 策略

实现控制器有几种策略。

### Servlet前端策略

这个策略推荐把控制器实现为servlet。虽然它和JSP前端策略在语义上是等效的，但是该策略还是首选策略。控制器负责处理与业务操作和控制流程有关的请求。这种责任与显示格式的操作有关，但二者在逻辑上又相互独立，所以这部分内容更适合用servlet而不是JSP来封装。

但是Servlet前端策略也确实有一些潜在的缺陷。尤其是，它不能利用JSP运行时环境提供的一些功能，比如说自动把请求参数复制到助手类的属性中等等。幸运的是，这个缺点无足轻重，因为很容易就可以自己编写出（或者找到）具有类似功能的通用工具。同样，在今后版本的

servlet技术规范中，也可能会把一些JSP的便捷功能加入到标准servlet中。

例6.14是Servlet前端策略的一个例子。

### 例6.14 Servlet前端策略

```

1  public class FrontController extends HttpServlet {
2      // 初始化servlet。
3      public void init(ServletConfig config) throws ServletException {
4          super.init(config);
5      }
6
7      // 销毁servlet。
8      public void destroy() { }
9
10     /** 处理HTTP
11      * <code>GET</code>和<code>POST</code> 方法的请求。
12      * @param request servlet请求
13      * @param response servlet响应
14      */
15     protected void processRequest(HttpServletRequest request,
16         HttpServletResponse response) throws ServletException,
17         java.io.IOException {
18
19     String page;
20     /**ApplicationResources 提供了一个简单的API
21      * 用以处理常数和其他
22      * 预定义值*/
23     ApplicationResources resource = ApplicationResources.getInstance();
24     try {
25         // 创建 Context对象
26         RequestContext requestContext =
27             new RequestContext(request, response);
28
29         // 调用请求处理组件，处理
30         // 输入请求
31         ApplicationController applicationController = new
32             ApplicationControllerImpl();
33         ResponseContext responseContext =
34             applicationController.handleRequest(requestContext);
35
36         // 调用响应处理组件，处理
37         // 响应逻辑
38         applicationController.handleResponse(
39             requestContext, responseContext);
40     } catch (Exception e) {
41         LogManager.logMessage("FrontController:exception : " +
42             e.getMessage());
43         request.setAttribute(resource.getMessageAttr(),

```

```

44         "Exception occurred : " + e.getMessage());
45     page = resource.getErrorPage(e);
46     // 把控制权分派给“当前网站不可用”视图
47     dispatch(request, response, page);
48   }
49 }
50
51 /** 处理HTTP <code>GET</code>方法
52  * @param request servlet请求
53  * @param response servlet响应
54  */
55 protected void doGet(HttpServletRequest request,
56   HttpServletResponse response) throws ServletException,
57   java.io.IOException {
58
59   processRequest(request, response);
60 }
61
62 /** 处理HTTP <code>POST</code> 方法
63  * @param request servlet 请求
64  * @param response servlet 响应
65  */
66 protected void doPost(HttpServletRequest request,
67   HttpServletResponse response)
68   throws ServletException, java.io.IOException {
69
70   processRequest(request, response);
71 }
72
73 protected void dispatch(HttpServletRequest request,
74   HttpServletResponse response, String page)
75   throws javax.servlet.ServletException, java.io.IOException {
76
77   RequestDispatcher dispatcher = this.getServletContext() .
78     getRequestDispatcher(page);
79   dispatcher.forward(request, response);
80 }
81
82 /** 返回servlet的简要描述 */
83 public String getServletInfo() {
84   return "Front Controller Pattern" +
85   " Servlet Front Strategy Example";
86 }
87 }

```

171

## JSP前端策略

控制器完成的那些处理操作，并不特别与显示格式相关，所以用JSP来实现控制器组件也不

172 太合适。因此，相对于JSP前端策略来说，Servlet前端策略是实现控制器的首选。

还有另一个不推荐使用JSP实现控制器的原因：如果改变请求处理逻辑，就需要软件开发者来改动标记页面<sup>Θ</sup>。在完成编码、编译、测试、调试这整个的工作周期的时候，软件开发者通常会觉得JSP前端策略比较难于处理。

例6.15是JSP前端策略的一个例子。

### 例6.15 JSP前端策略

```

1  <%@page contentType="text/html"%>
2  <%@ page import="corepatterns.util.*" %>
3  <%@ taglib prefix="c" uri="http://java.sun.com/jstl/core" %>
4
5  <html>
6  <head><title>JSP Front Controller</title></head>
7  <body>
8
9  <h3><center> Employee Profile </h3>
10 <%
11 <%
12     /** 这里放置控制逻辑...
13      * 在代码中的某个位置我们获取员工信息
14      * 把它封装到一个值对象里，然后把这个bean用键“employee”放在
15      * 请求的作用域（request scope）中。我们省略了这段代码。
16      * 在此处我们可以把控制权分派给另一个JSP，也可以直接
17      * 在这段代码的剩下部分执行操作。
18      */
19 <%
20 <jsp:useBean id="employee" scope="request"
21     class="corepatterns.util.EmployeeTO"/>
22 <FORM method=POST >
23 <table width="60%">
24 <tr>
25   <td> First Name : </td>
26   <td> <input type="text" name="<%="Constants.FLD_FIRSTNAME%>" 
27           value="

```

173

<sup>Θ</sup> 所谓“标记页面”，原文是a page of markup，是指JSP页面。页面中有大量HTML、JSP标记，所以这样说。在软件业发达国家，JSP页面往往由软件工程师之外的专人负责设计。所以也就有了下面“软件开发者觉得JSP前端策略难于处理”的说法。

```

36   </tr>
37   <tr>
38     <td> Employee ID : </td>
39     <td> <input type="text" name="<%>=Constants.FLD_EMPID%" value="

```

### 命令加控制器策略

本章后面的应用控制器模式的“命令处理器策略”，描述了使用命令完成操作管理的通用做法。而把命令处理器和前端控制器一起使用，这就叫做“命令加控制器”策略。

使用命令模式[GoF]，就能够给处理请求的助手对象提供一个通用的接口。这样就能尽可能减小组件之间的耦合。当开发者需要添加请求处理操作的时候，命令服务也提供了一种灵活、便于扩展的机制。

因为命令处理和命令调用之间没有耦合，所以命令处理机制可以被各种客户端重用，而不仅仅限于浏览器。这个策略也便于创建复合命令（见[GoF]复合模式）。例6.16给出了这种策略的示例，图6-13是该策略的序列图。

#### 例6.16 命令加控制器策略

```

1  /** 这个processRequest 方法会被servlet的doGet和doPost
2  * 方法调用*/
3  protected void processRequest(HttpServletRequest request,
4      HttpServletResponse response) throws ServletException,
5      java.io.IOException {
6
7  String resultPage;
8  try {
9      RequestHelper helper = new RequestHelper(request);
10
11     /** 这个getCommand() 方法内部使用了一个工厂，
12      * 按照以下形式获取命令对象：
13      * Command command = CommandFactory.create(
14      *     request.getParameter(op));
15      */
16     Command command = helper.getCommand();
17     // 把请求委派到一个作为命令对象的助手上
18     resultPage = command.execute(request, response);

```

```

19     } catch (Exception e) {
20         LogManager.logMessage("FrontController", e.getMessage());
21         resultPage = ApplicationResources.getInstance().getErrorPage(e);
22     }
23     dispatch(request, response, resultPage);
24 }

```

图6-13是这个策略的序列图。

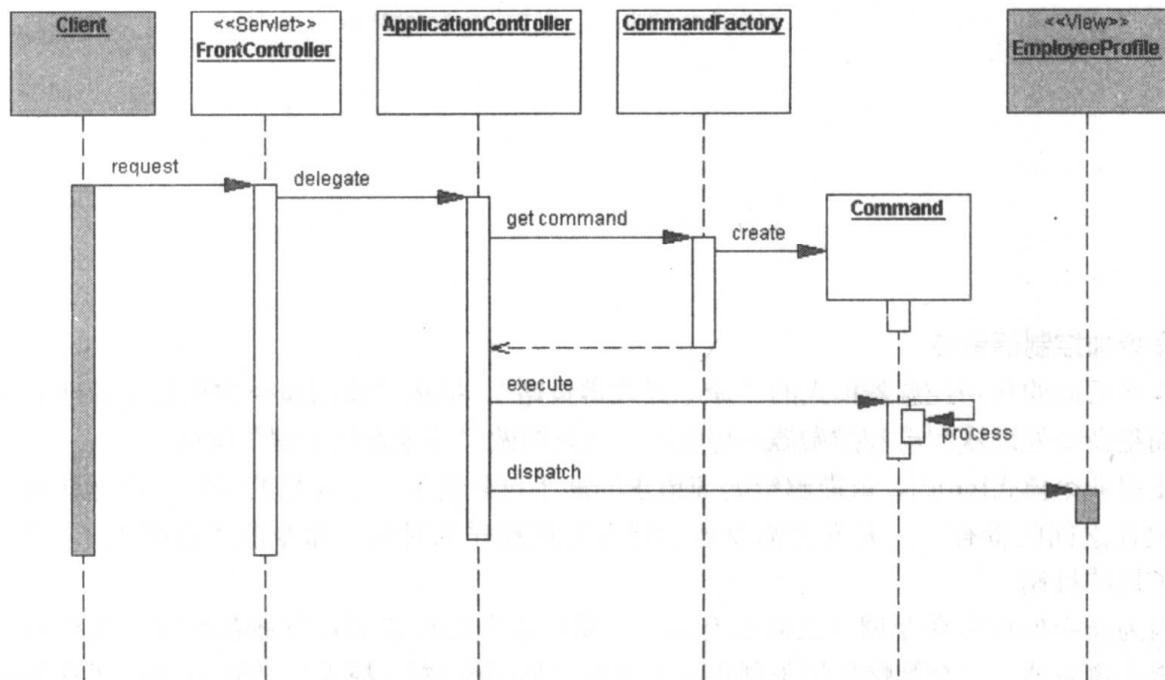


图6-13 命令加控制器策略序列图

### 物理资源映射策略

这个策略中，所有的请求都映射于资源的物理名称，而不是逻辑名称。一个例子就是“<http://some.server.com/resource1.jsp>”这样一个URL。如果采用控制器，这个URL可能就会写成“<http://some.server.com/servlet/Controller>”。

175

与物理资源映射相比，我们通常推荐逻辑资源映射策略，因为后者能够提供更高的灵活性。

### 逻辑资源映射策略

在这个策略中，请求映射于资源的逻辑名称，而不是物理名称。可以修改部署文件中的声明，这样就能让逻辑名称对应于不同的物理名称。

比如，URL “<http://some.server.com/process>” 可以这样映射：

`process=resource1.jsp`

或者

`process=resource2.jsp`

或者

`process=servletController`

逻辑资源映射策略比物理资源映射策略要灵活的多，因为后者与要锁定资源的实际名称。

### 多路资源映射策略

实际上这是逻辑资源映射策略的一个子策略。它不仅仅把一个逻辑名称，而是把整个一组逻辑名称，都映射到同一个物理资源上。比如说，就可以采用通配符，把所有结尾是.ctrl的请求映射到一个特定的处理器上。176

表6-1给出了一种可能的请求映射。

表6-1 请求和映射

请    求	映    射
http://some.server.com/action.ctrl	*.ctrl = servletController

这也就是JSP引擎使用的策略，它可以保证对JSP资源（名称以.jsp结尾的资源）的请求可以被特定的处理器处理。

可以增添请求的相关信息，以供逻辑映射应用。见表6-2。

表6-2 请求和映射（附带额外信息）

请    求	映    射
http://some.server.com/profile.ctrl?usecase=create	*.ctrl = servletController

使用这一策略的一个主要益处在于，它为设计请求处理组件提供了极大的灵活性。如果结合其他策略（比如命令加控制器策略）使用本策略，就能创建一个强大的请求处理机制。

设想一个控制器，处理所有以.ctrl结尾的请求（像表6-2的例子中那样）。而且，再假设资源名称中句点之前的部分（比如表6-2例子中的“Profile”）是用例名称的一部分。这也就是告诉请求处理器，要处理一个名为“create profile（创建用户信息）”的用例了。这种多路资源映射把请求发送到servletController（这也包含在表6-2的映射信息中）。控制器创建一个合适的命令对象，正如命令加控制器策略中所述。

控制器怎么知道应该委派给哪一个命令对象呢？因为它可以利用请求的URI中包含的附加信息，所以它就可以把请求委派给负责处理“用户信息创建”的命令对象。这可以是一个专门负责用户信息创建和修改的ProfileCommand（用户信息命令）对象，也可以是一个专门程度更高的ProfileCreationCommand（用户信息创建命令）对象。177

### “控制器中的分配器”策略

如果视图管理功能以及相关的分配器功能在系统中比重很小，那么它们可以放进控制器中，如图6-14和图6-15所示。

这样，通常由应用控制器完成的视图管理功能，被包括在前端控制器中实现了。如果系统只需要最基本的视图管理，那么最好用这种方式实现。视图（View）使用一个业务助手（Business Helper）来进行有限的操作管理，再使用一个视图助手（ViewHelper）来生成视图。这种处理请求的思路被称为分配器视图。

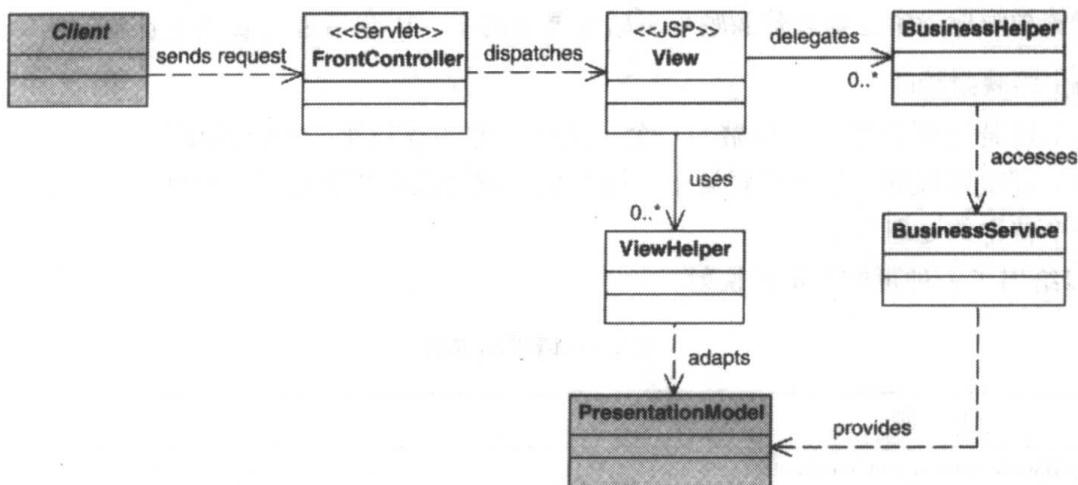


图6-14 “控制器中的分配器”类图

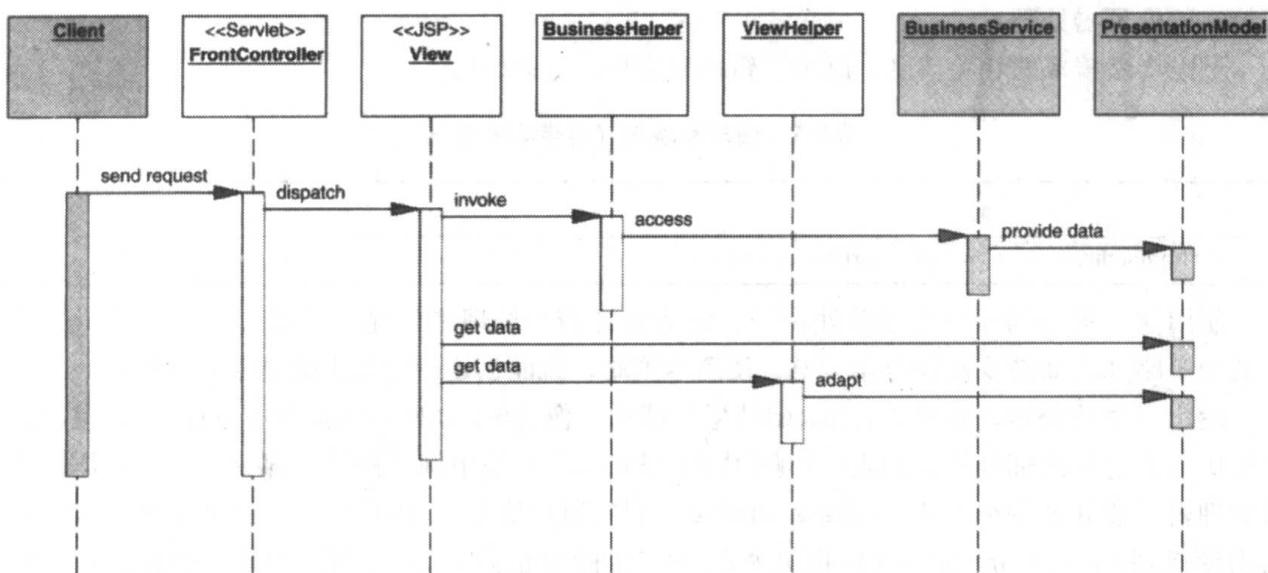


图6-15 “控制器中的分配器”序列图

### 基类前端策略

这个策略与 *Servlet* 前端策略结合使用；它实现了一个控制器基类，其他的控制器可以扩展其实现。基类前端中包括共通的、默认的实现，每个子类则可以覆盖这些实现。这个策略的缺点在于，虽然超类代码可以促进功能的共享和重用，但是这也导致了整个类型结构都比较脆弱——对于一个子类作一些必要修改也会影响到所有的子类。

### 过滤器控制器策略

拦截过滤器同样也支持集中控制的请求处理逻辑。这也就意味着，控制器的一些功能可以由过滤器实现。但是，过滤器主要着重在拦截请求，并通过过滤器链来进行“装饰”，而前端控制器则为所有相关请求充当了一个中心访问点，并且还充当了应用控制器的委派点。虽然一些控制逻辑（比如日志和调试信息）既可以由过滤器也可以由前端控制器实现，但这两种组件事实上是相辅相成的，可以结合、协作地使用。

## 效果

- 集中控制

控制器集中了在各个请求之间共通的处理控制逻辑。对于请求控制机制来说，控制器是最初的访问点，它负责把请求委派给应用控制器，进行底层的业务处理、视图生成操作。

- 提高系统可维护性

程序中的控制逻辑需要阻挡对应用系统的非法访问。集中了控制逻辑，也就更加便于监管这一部分的控制流程。而且，因为程序只有一个人口点，因此与在所有页面中都加入检查代码相比，监控这样的系统需要更少的资源。

- 增进了重用

共通的代码放进了控制器，或者通过控制器被管理/委派，这样能够更清晰地划分应用系统，鼓励了重用。

- 增进了开发团队中职责之间的区分

采用控制器能够更清晰地划分开发团队中各人的职责，因为一些人（软件开发者）能够更容易地维护程序逻辑，另一些人（Web设计人员）则负责维护用于视图生成的标记页面。

## 相关模式

- 拦截过滤器

拦截过滤器和前端控制器都集中了请求处理的部分功能。

- 应用控制器

应用控制器封装了操作管理和视图管理代码，前端控制器会把请求委派给这两种代码。

- 视图助手

视图助手把业务逻辑和处理逻辑从视图中分解出来，放到了助手对象中，因此为控制和分派提供了一个集中处理点。流程控制逻辑放进了控制器中，与视图格式相关的代码则放进了助手对象。

- 分配器视图和服务到工作者

分配器视图模式和服务到工作者模式有不同的用途。服务到工作者采用了一个基于控制器的架构，因此其中前端控制器就至关重要；而分配器视图采用的则是基于视图的架构。

180

## Context 对象

### 问题

不想在与协议无关的环境上下文中使用针对特定协议的系统信息。

在请求和响应的整个生命周期中，一个应用系统通常要使用系统信息，比如请求、配置、安全数据等等。系统信息的获取方式与环境上下文（context）有关。当负责业务应用的组件和服务必须使用一些处于它们的环境上下文之外的系统信息时，这些组件的灵活性和可重用性都会下降。所以，在相关的环境上下文之外使用一个特定协议的API，这就会把特定的接口和处理

细节暴露给使用这个API的所有组件。这样，所有作为API的客户的组件也就和那个特定协议产生了紧耦合。

比如说，Web组件接收HTTP协议的请求。但是，如果让表现层和其他层次的组件共享这些HTTP请求，就会把协议细节暴露给所有这些组件。如果协议本身或者其中的某个细节改变了；比如说，如果表单的某个元素或者字段名称变了；那么所有的客户端代码<sup>Θ</sup>也必须变化。再举两个更通用的系统数据的例子：系统各处都要使用配置和安全数据，但是这两种数据与业务应用的服务之间只存在松耦合。

## 约束

- 组件和服务要访问系统信息。
- 要解除系统信息的协议细节与业务应用组件/服务之间的耦合。
- 只想在特定环境上下文中暴露与协议相关的API。

## 解决方案

使用*Context*对象，按照协议无关的方式封装状态，然后在整个应用系统中使用这种封装后的对象。

181 使用*Context*对象封装系统数据，就可以让应用系统的其他部分也能使用这种数据，同时也不避免了在应用系统和特定协议之间造成耦合。

比如，HTML表单中的每个字段对应于HTTP请求中的一个参数；*Context*对象可以按照协议无关的方式存储这种数据，同时也便于数据的转换和验证。应用系统的其他部分可以直接通过这个*Context*对象来访问这些数据，并不需要对HTTP协议有任何知识。如果协议发生了任何改变，都可以交给这个*Context*对象处理，系统的其他部分无需变动。

既然减少了对特定协议的依赖，那么也就更加方便测试，很容易在一种更加通用的环境下进行测试，减少了对特定容器的依赖。比如，可以用*Context*对象包装*HttpServletRequest*，这样就可以在Web容器之外轻松地使用这个*Context*对象进行测试。为了满足测试需要，可以让这个*Context*对象装载假数据，而无需包含与容器相关的*HttpServletRequest*类或者HTTP请求的属性等等。使用了*Context*对象，还能把状态抽象出来，形成可以在不同层次间共享的数据结构，从而有助于对业务层隐藏表现层的实现细节。*Context*对象对于协议是中立的，它所提供的松耦合特性能够增进应用系统的可重用性和可维护性。

使用*Context*对象，也就引发了它与传输对象之间关系的问题。传输对象在远程的层次之间传输状态。虽然在传输数据的目的上，*Context*对象和传输对象的作用一致，但是*Context*对象的主要目标还是通过协议无关的方式共享系统数据，从而增进应用系统的可重用性和可维护性。

通过这种方式，*Context*对象减少了原本不相关的系统组件和业务应用组件之间的耦合。另一方面，传输对象的主要目标是减少远程网络通信从而提高性能。而且，传输对象往往还与其他传输对象具有关系，对应于它们所代表的业务数据，而*Context*对象体现的则是底层状态。

<sup>Θ</sup> 这里所说的“客户端代码”指的是特定组件的使用者。不一定是实际的图形界面客户端。

## 结构

图6-16是Context对象的类图。

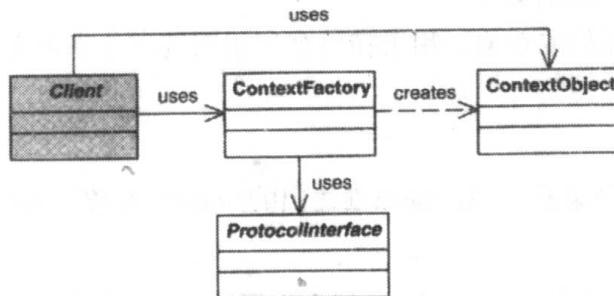


图6-16 Context对象的类图

182

## 参与者和责任

图6-17是Context对象的序列图。

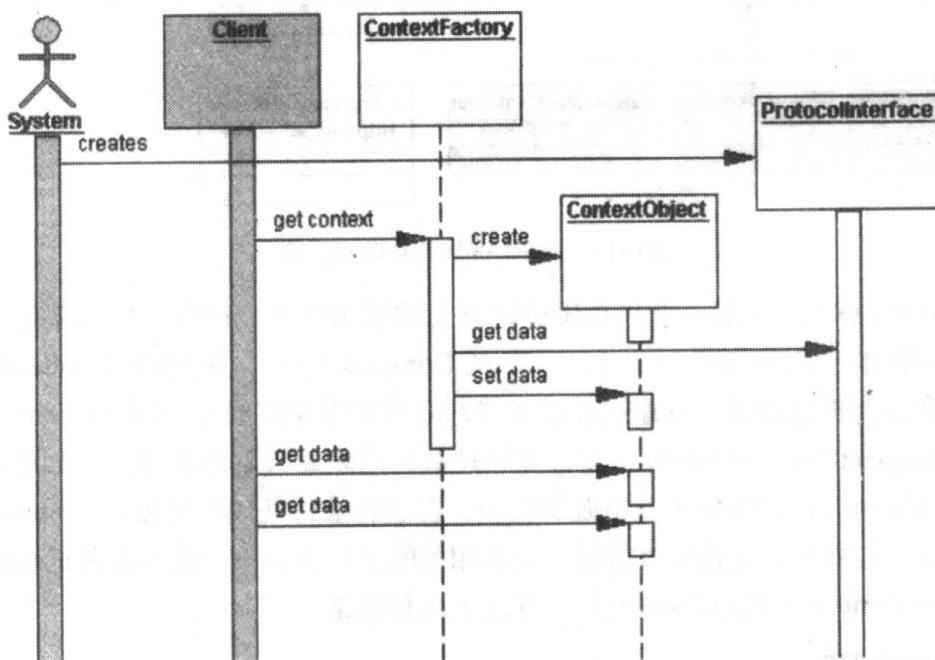


图6-17 Context对象的序列图

客户端使用ContextFactory（Context工厂）创建一个使用ProtocolInterface（协议接口）的ContextObject（Context对象）。ContextObject对其他的业务应用组件和服务隐藏了ProtocolInterface的底层细节。

### 客户端 (Client)

创建了使用ProtocolInterface 的Context对象。

### ProtocolInterface（协议接口）

一个暴露了协议或者系统某层次细节的对象。

### ContextFactory (Context工厂)

ContextFactory创建了与具体协议和层次无关的ContextObject。

### ContextObject (Context对象)

ContextObject是一个通用的对象，用于在整个应用系统中分享中立于业务领域的状态数据。

183

## 策略

实现Context对象有多种策略，我们按照创建出的Context对象的类型来归纳这些策略。

### 请求Context策略

如果一个ContextObject封装了请求状态，那么也可以把它称作RequestContext(请求Context)。

图6-18是RequestContext的结构：

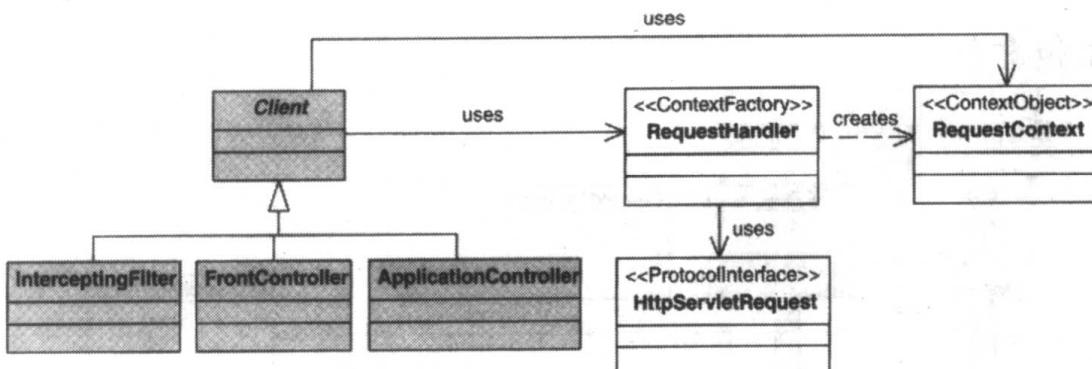


图6-18 请求Context策略的类图

一个HttpServletRequest是一个与具体协议有关的请求对象，所以在应用系统中对这个对象的暴露应该是有限的。ContextFactory创建了一个ContextObject，并把HttpServletRequest的状态传递给它。通常由拦截过滤器、前端控制器或者应用控制器来充当客户端（Client）的角色。

在这里，RequestContext中的数据通常要经过最初的表单级别的验证；比如说，对空字段的检查，或者对信用卡号码的位数是否正确的检查。在进行业务处理的时候，ContextObject的状态通常还要经过第二轮检查，这轮检查就是与业务相关的了，比如一个值是否在正确的范围内等。

实现RequestContext（请求Context）还有以下几种思路。

### 请求Context表策略

这是最基本的适配器策略。把Context对象实现为一个标准的表（Map），并把相关的请求状态传输到其中，然后再系统内部共享这个表。这个思路的优点在于它的简明。

但是简明也包含代价。使用标准表实现的是一个弱类型<sup>①</sup>的ContextObject。这样做也就缺乏编写自己的定制类能带来的特性（包括定制助手方法以及强类型<sup>②</sup>）。

<sup>①</sup> 弱类型，原文是“loosely typed”，也就是说，在表（Map）对象中的每条数据，其类型是不确定的。

<sup>②</sup> 定制助手方法以及强类型，这是说，如果不用表（Map）对象装载数据，而使用自己编写的类，就能在该类中包括一些辅助的方法，而且还可以保证数据的类型是确定的。

184

例6.17是请求Context表策略的例子。

### 例6.17 请求Context表策略

```

1  public class FrontController extends HttpServlet {
2  . . .
3      private void processRequest(HttpServletRequest request,
4          HttpServletResponse response) throws ServletException,
5              java.io.IOException {
6
7          // 用表策略创建RequestContext对象
8          Map requestContextMap = new HashMap(request.getParameterMap());
9          Dispatcher dispatcher = new Dispatcher(request, response);
10         requestContextMap.put("dispatcher", dispatcher);
11
12         // 创建 ApplicationController (应用控制器) 实例
13         ApplicationController applicationController =
14             new ApplicationControllerImpl();
15
16         // 请求处理
17         ResponseContext responseContext =
18             applicationController.handleRequest(requestContextMap);
19
20         // 响应处理
21         applicationController.handleResponse(requestContextMap,
22             responseContext);
23     }
24     . . .
25 }
```

### 请求Context POJO策略

另一种思路是使用一个POJO来封装请求。同样，根据特定的应用场景，这种做法也可能带来特定的好处和缺点。使用POJO，就意味着可以加入强类型控制，而且可以加入助手方法，从而根据现有属性创建更复杂的属性。换句话说，这个bean暴露的属性并不一定都是这个bean本身的字段。

185

例6.18到例6.21是这个策略的代码。

### 例6.18 前端控制器

```

1  public class FrontController extends HttpServlet {
2      private ApplicationController applicationController;
3
4      public void init(ServletConfig servletConfig) throws
5          ServletException{
6
7          super.init(servletConfig);
8
9          // 初始化请求处理组件 (无状态)
10         applicationController = new ApplicationControllerImpl();
```

```

11     applicationController.initialize();
12 }
13
14 // 被调用doGet和doPost方法调用
15 protected void process(HttpServletRequest request,
16     HttpServletResponse response) throws java.io.IOException {
17
18     // 按照请求类型创建RequestContext
19     RequestContextFactory requestContextFactory =
20         RequestContextFactory.getInstance();
21     RequestContext requestContext =
22         requestContextFactory.createRequestContext(request);
23
24     // 处理请求
25     ResponseContext responseContext =
26         applicationController.handleRequest(requestContext);
27
28     // 视图管理——导航并分派到合适的视图
29     Dispatcher dispatcher = new Dispatcher(request, response);
30     responseContext.setDispatcher(dispatcher);
31     applicationController.handleResponse(requestContext,
32         responseContext);
33 }
34 . . .
35 }

```

#### 例6.19 ApplicationControllerImpl（应用控制器实现）

186

```

1 public class ApplicationControllerImpl implements
2     ApplicationController {
3
4     public void initialize() {
5         commandMapper = CommandMapper.getInstance();
6     }
7
8     public ResponseContext handleRequest(RequestContext requestContext) {
9         ResponseContext responseContext = null;
10        try {
11            // 验证请求参数
12            requestContext.validate();
13
14            // 把命令名称翻译成Command类的对象
15            String commandName = requestContext.getCommandName();
16            Command command = commandMapper.getCommand(commandName);
17
18            // 调用命令
19            responseContext = command.execute(requestContext);
20
21            // 确定视图名称

```

```

22     CommandMap mapEntry = commandMapper.getCommandMap(commandName);
23     String viewName = mapEntry.getViewName();
24     responseContext.setLogicalViewName(viewName);
25 } catch (ValidatorException e1) {
26     // Handle Exception
27 }
28 return responseContext;
29 }
30 . .

```

### 例6.20 RequestContextFactory (请求Context工厂)

```

1 // POJO ContextObject 工厂
2 public class RequestContextFactory {
3     public RequestContext createRequestContext(ServletRequest request) {
4
5         RequestContext requestContext = null;
6         try {
7             // 从请求对象中确定命令字符串
8             String commandId = getCommandId(request);
9             // 用CommandMap (命令映射表) 来为给定的命令确定
10            // POJO RequestContext 类
11            CommandMapper commandMapper = CommandMapper.getInstance();
12            CommandMap mapEntry = commandMapper.getCommandMap(commandId);
13            Class requestContextClass = mapEntry.getContextObjectClass();
14
15            // 创建POJO实例
16            requestContext = (RequestContext)
17                requestContextClass.newInstance();
18
19            // 设置于协议相关的请求对象
20            requestContext.initialize(request);
21        } catch(java.lang.InstantiationException e) {
22            // 处理异常
23        } catch(java.lang.IllegalAccessException e) { }
24        return requestContext;
25    }
26
27    private String getCommandId(ServletRequest request) {
28        String commandId = null;
29        if ( request instanceof HttpServletRequest) {
30            String pathInfo = ((HttpServletRequest)request).getPathInfo();
31            commandId = pathInfo.substring(1); // 删除路径前面的符号 '/'
32        }
33        return commandId;
34    }
35 }

```

**例6.21 ProjectRegistrationRequestContext (项目注册请求Context)**

```
1 // 一个专门的POJO请求Context ——它的实例由
2 // RequestContextFactory类负责创建
3 public class ProjectRegistrationRequestContext extends
4     HttpRequestContext {
5
6     public ProjectRegistrationRequestContext(HttpServletRequest request) {
7         super(request);
8         initialize(request);
9     }
10
11    public ProjectRegistrationRequestContext() { }
12
13    public void initialize(ServletRequest request) {
14        setRequest(request);
15        setProjectName(request.getParameter("projectName"));
16        setProjectDescription(request.getParameter("projectdescription"));
17        setProjectManager(request.getParameter("projectmanager"));
18    }
19
20    public String getProjectDescription() {
21        return projectDescription;
22    }
23
24    public void setProjectDescription(String projectDescription) {
25        this.projectDescription = projectDescription;
26    }
27
28    public String getProjectManager() {
29        return projectManager;
30    }
31
32    public void setProjectManager(String projectManager) {
33        this.projectManager = projectManager;
34    }
35
36    public String getProjectName() {
37        return projectName;
38    }
39
40    public void setProjectName(String projectName) {
41        this.projectName = projectName;
42    }
43
44    public boolean validate() {
45        // 验证规则
46        return true;
```

```

47     }
48     .
49 }
```

这种做法的一个主要缺点是必须给每个属性都定义用于访问的方法，所以如果某种应用碰巧有很多属性，也必须给每个属性定义一个或多个用于访问的方法。从开发和维护的角度看，这个任务都很可能是让人畏惧的。

可以采取一个混合思路实现这个策略，既提供了POJO的优点，又具有基于表的策略在开发和维护上的便利性。这个混合思路就是：采用代码生成工具来生成ContextObject。业界常见的POJO策略例子就是Struts [Struts] 框架的ActionForm类。189

这种混合思路的一个做法是Struts框架的DynaActionForm。混合思路的另外一个做法是使用某种代码生成工具，比如xDoclet [xDoclet]。

例6.22到例6.24示范了这两种办法。

### 例6.22 使用DynaActionForm 创建POJO请求Context

```

1 <struts-config>
2   <form-beans>
3     <form-bean name="projectRegistrationForm"
4       type="org.apache.struts.action.DynaActionForm">
5       <form-property name=" projectName " type="java.lang.String"/>
6       <form-property name=" projectDescription "
7         type="java.lang.String"/>
8       <form-property name=" idCustomer " type="java.lang.String"/>
9       <form-property name=" projectManager " type="java.lang.String"/>
10      <form-property name=" startDate " type="java.util.Date"/>
11      <form-property name=" endDate " type="java.util.Date"/>
12      <form-property name=" emailAlias " type="java.lang.String"/>
13   </form-bean>
14   .
15 </struts-config>
```

### 例6.23 使用Xdoclet生成Context对象POJO

```

1 /**
2  * 这是一个例子，示范如何使用XDoclet 标记
3  * 通过Entity Bean的类定义
4  * 来生成POJO Context对象
5  *
6  * @ejb.finder
7  *   role-name="Manager"
8  *   signature="Collection findAll()"
9  *   transaction-type="NotSupported"
10 *
11 * @ejb.transaction
12 *   type="Required"
13 *
14 * @ejb.pk
```

[190]

```

15     *      generate="true"
16     *
17     * @ejb.data-object
18     *      name="{0}TO"
19     *      equals="true"
20     *
21     * @ejb.persistence
22     *      table-name="employee"
23     *
24     * @struts.form
25     *      name="Context"
26     *      include-all="true"
27     *
28     */
29 // Entity Bean 类
30 public abstract class EmployeeBean implements EntityBean {
31     private EntityContext ctx;
32     ...
33 }

```

在上面这个EmployeeBeanEntity类上运行xdoclet，就能生成以下代码。

#### 例6.24 Xdoclet输出

[191]

```

1  /**
2   * Generated by XDoclet/ejbdoclet/strutsform.
3   * This class can be further processed with
4   * XDoclet/webdoclet/strutsconfigxml.
5   *
6   * @struts.form name="corej2eepatterns.ejb.Employee.Context"
7   */
8  public class EmployeeContextForm extends
9      org.apache.struts.action.ActionForm implements java.io.Serializable{
10
11     protected java.lang.String id;
12     protected java.lang.String lastName;
13     protected java.lang.String firstName;
14     protected java.lang.String phone;
15     protected java.lang.String fax;
16     protected java.util.Date creationDate;
17
18     /** Default empty constructor. */
19     public EmployeeContextForm() { }
20
21     ...
22 }

```

#### 请求Context验证策略

因为RequestContext（请求Context）可以封装输入请求的状态，所以在表现层，请求数据的

验证通常会与一个RequestContext协作完成。一旦服务器接收了一个请求，请求的状态数据往往首先被验证，然后再被系统的其他部分使用。RequestContext存储了请求状态，这样，如果验证时发现数据有错，需要原封不动地把这些数据连同错误信息返回给客户端，那么就可以使用RequestContext的数据。

对于一个Web应用系统，通常有两种层次的验证。

- 第一类是表单层次的验证，用于验证输入数据的格式是否合格，比如说，信用卡号码数字长度是否正确。
- 第二类是业务验证，比如信用卡的认证。

以上这两种验证都包含一些业务考虑，并且事实上，有时候很难确定某种验证到底是属于表单层次还是业务层次。比如，可以设想一下信用卡过期日期的例子。虽然处理这个日期值肯定是一种业务层次的验证，但是在把日期数据传给业务层之前，很可能也需要首先对日期值作一次表单层次的初步检查，看看这个日期是否还没有过去。另外一个例子是对输入的用户名和密码作初步的字符串长度检查，另外还要检查字符串是否按照规定包含字母和数字，接着才是对用户名/密码的实际身份认证。

如果能够把这些验证规则从其他程序逻辑中分离出来，就能够增进验证机制整体的可重用性，因为这样一来，就可以单独修改这些验证规则的代码。而且，如果这些验证规则可能会跟系统的多个层次都有关系，那么把这些规则从与特定层相关的代码中分离出来，就能够降低各层之间的耦合。

上面提到的这些验证设计考虑与使用Context对象的意图和设计相当一致，这也就是为什么这两种技术的实现往往会相互联系（见例6.25至例6.27）。对这个策略的以下实现使用Jakarta Commons Validator<sup>⊖</sup> [Jakarta-Valid]。

### 例6.25 用请求Context对象支持数据验证

```

1 // RequestContext基类，提供了基本的验证架构
2 public class RequestContext {
3     public RequestContext (ServletRequest request) {
4         setRequest(request);
5     }
6
7     public ValidatorResults validate() throws ValidatorException {
8         try {
9             InputStream inputStream =
10                 Thread.currentThread().getContextClassLoader().
11                 getResourceAsStream(getValidationResourceName());
12             // 创建ValidatorResources的一个实例，以便从一个xml文件
13             // 初始化验证器
14             ValidatorResources resources = new ValidatorResources();
15             ValidatorResourcesInitializer.initialize(resources, inputStream);
16

```

192

<sup>⊖</sup> Jakarta Commons Validator，这是Apache基金的Jakarta项目中的公用库（Commons）的一部分，专门提供验证功能。

```

17     // 利用上面装载的资源和表单的名称构造验证器
18     Validator validator = new Validator(resources,
19         getValidationFormName());
20
21     // 把名称bean作为一种资源加入验证器
22     // 以便据此进行验证
23     validator.addResource(Validator.BEAN_KEY, this);
24
25     // 验证
26     ValidatorResults validatorResultsAggregate = null;
27     validatorResultsAggregate = validator.validate();
28
29     // 进行通用的验证错误处理操作（如果需要的话）
30     return validatorResultsAggregate;
31 } catch (IOException e) {
32     // 处理异常
33 }
34 return null;
35 }
36
37 . . .
38
39 public void setRequest(ServletRequest request) {
40     this.request = request;
41 }
42
43 private ServletRequest request;
44 }
```

### 例6.26 一个专门<sup>Θ</sup>的请求Context，实现专门的（项目注册）Context验证

```

1  public class ProjectRegistrationRequestContext extends RequestContext {
2
3     public ProjectRegistrationRequestContext(HttpServletRequest request) {
4         super(request);
5         initialize(request);
6     }
7     public ProjectRegistrationRequestContext() { . . . }
8
9     public void initialize(ServletRequest request) {
10        setRequest(request);
11        setProjectName(request.getParameter("projectName"));
12        setProjectDescription(request.getParameter("projectdescription"));
13        setIdCustomer(request.getParameter("customerid"));
14        setProjectManager(request.getParameter("projectmanager"));
15        setStartDate(request.getParameter("startdate"));
16        setEndDate(request.getParameter("enddate"));
```

<sup>Θ</sup> 所谓专门，是相对于上面作为整个机制架构的基类而言。本例实现的是对一种“项目注册”数据的验证。

```

17     setEmailAlias(request.getParameter("emailalias"));
18 }
19 .
20 .
21 .
22 // 专门的Context验证
23 public ValidatorResults validate() {
24     ValidatorResults validatorResultsAggregate = null;
25     try {
26         validatorResultsAggregate = super.validate();
27         ValidatorResult validatorResult = validatorResultsAggregate.
28             getValidatorResult("projectName");
29         if (validatorResult.isValid("required") == false) {
30             // 处理“没有项目名称”的情况
31         }
32
33         validatorResult = validatorResultsAggregate.
34             getValidatorResult("projectDescription");
35         if (validatorResult.isValid("required") == false) {
36             // 处理“没有项目说明”的情况
37         }
38
39         validatorResult = validatorResultsAggregate.
40             getValidatorResult("emailAlias");
41         if (validatorResult.isValid("email") == false) {
42             // 处理“email无效”的情况
43         }
44     } catch (ValidatorException e) {
45         // 处理异常
46     }
47     return validatorResults;
48 }
49
50     public String getValidationFormName() {
51         return "ProjectRegistrationForm";
52     }
53 }
```

194

### 例6.27 FormValidators.xml

```

1  FormValidators.xml
2
3  <form-validation>
4      <global>
5          <validator name="required" classname=
6              "corepatterns.ContextObject.adapterpojo.RequestValidator"
7              method="validateRequired" methodParams=
8              "java.lang.Object,org.apache.commons.validator.Field"/>
9
```

```

10      <validator name="email" classname=
11          "corepatterns.ContextObject.adapterpojo.RequestValidator"
12          method="validateEmail" methodParams=
13          "java.lang.Object,org.apache.commons.validator.Field" />
14
15  </global>
16  <formset>
17      <form name="ProjectRegistrationForm">
18          <field property="projectName" depends="required">
19              <arg0 key="ProjectRegistrationForm.projectname.displayname"/>
20          </field>
21          <field property="projectDescription" depends="required">
22              <arg0 key=
23                  "ProjectRegistrationForm.projectdescription.displayname"/>
24          </field>
25          <field property="emailAlias" depends="required">
26              <arg0 key="ProjectRegistrationForm.emailalias.displayname"/>
27          </field>
28      </form>
29  </formset>
30  </form-validation>

```

195

## 配置Context策略

如果一ContextObject封装了配置状态，那么也可以称其为ConfigurationContext（配置Context）。

### JSTL 配置策略

为了让不同环境上下文中的组件都能使用配置数据，应用系统常常会把这些数据分类封装。比如，JSP标准标记库（JSTL）技术规范[JSTL]1.0版（其中规定了JSP的标准标记）；就包括了一个配置类，专门提供这样的功能。这个类能够让处于各种不同的环境上下文（包括不同的JSP作用域）中的组件使用配置信息。这个类就是javax.servlet.jsp.jstl.core.Config，其提供了用于获取和设置配置数据（比如一个特定用户的区域<sup>Θ</sup>设置）的方法。以下例子（例6.28和例6.29）示范了如何在整个web层通过Config类共享配置数据。

### 例6.28 JSTL 配置策略

```

1  <!--
1   JSTL Config 类封装了配置数据，可供在多种不同的环境上下文中使用
1  -->
2  <corej2eepatterns:Announcements>
3
4      <b>Announcements:</b><br>
5
6      <!-- Display Request-specific Announcements, if any -->
7      <c_rt:out value='<%= Config.get(request,

```

<sup>Θ</sup> 区域，原文locale，指特定客户端的地区性配置，包括语言、日期/货币格式等等。

```

8     Constants.ANNOUNCEMENTS) %>'/> <br>
9
10    <!-- Display User-specific Announcements, if any -->
11    <c_rt:out value='<%= Config.get(session,
12          Constants.ANNOUNCEMENTS)%>' /><br>
13
14    <!-- Display Global Announcements, if any -->
15    <c_rt:out value='<%=
16          Config.get(application,Constants.ANNOUNCEMENTS)%>' /> <br>
17
18 </corej2eepatterns:Announcements>

```

196

### 例6.29 利用JSTL Config类进行标记处理

```

1  public class AnnouncementsTag extends TagSupport implements Tag {
2      public int doStartTag() throws JspException {
3
4          String announcements = (String)Config.find(pageContext,
5              Constants.DISPLAY_ANNOUNCEMENT);
6          if ( "true".equalsIgnoreCase(announcements)) {
7              .
8
9              return EVAL_BODY_INCLUDE;
10         }
11         return SKIP_BODY;
12     }
13 }

```

### 安全性Context策略

如果ContextObject封装了安全性状态，那么也可以称它为SecurityContext（安全性Context）。

图6-19给出了使用SecurityContext实现身份认证的序列图。

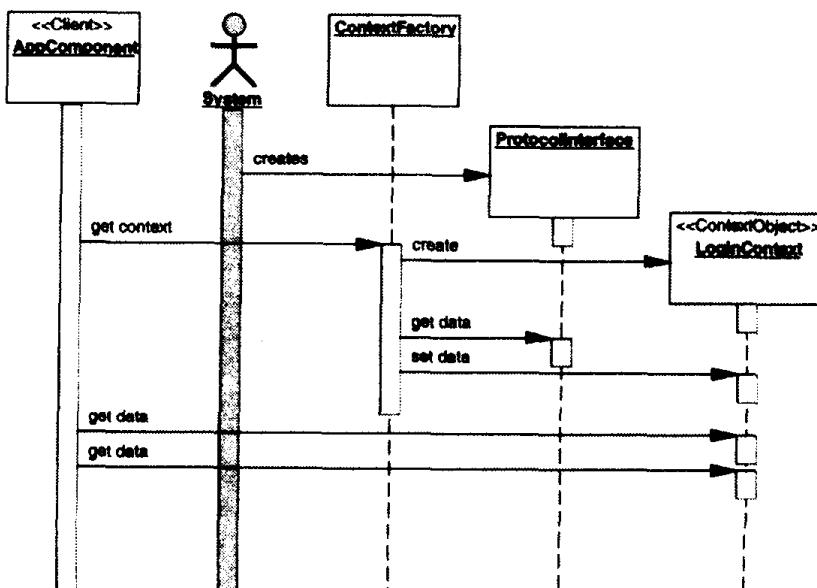


图6-19 使用SecurityContext实现身份认证

197

Java认证和授权服务（Java Authentication and Authorization Service，JAAS）[JAAS]是Java 2标准平台1.4版[J2SE1.4]的一部分。JAAS使用了一个称为LoginContext的ContextObject提供认证服务。JAAS LoginContext对客户端隐藏了认证机制的底层实现细节，为系统与用户之间的交互提供了一种模块化的、可插拔的回调机制。

LoginContext要用到一个配置文件，其中包括了整个机制用于用户认证的所有相关细节。这样，ContextObject就对客户端隐藏了具体的实现细节（参见例6.30）。

### 例6.30 使用LoginContext 传播安全性Context

```

1  public class FrontController extends HttpServlet {
2
3      . . .
4      protected void process(HttpServletRequest request,
5          HttpServletResponse response) throws java.io.IOException {
6
7          // 为请求创建Context对象
8          RequestContext requestContext =
9              RequestContextFactory.getInstance().
10             createContext(request);
11
12         // 使用JAAS框架进行认证
13         // 获取认证凭证
14         String username = requestContext.getStringParameter("UserName");
15         String password = requestContext.getStringParameter("Password");
16         try {
17             // 对于底层的各种可插拔的认证模块，LoginContext
18             // 是一个工厂类
19             LoginContext loginContext = new LoginContext("AuthLevel1",
20                 new AuthCallbackHandler(username, password));
21
22             // 认证SubjectΘ
23             loginContext.login();
24
25             // 获取通过认证的Subject
26             Subject subject = loginContext.getSubject();
27
28             // 在会话作用域传播安全性Context
29             HttpSession session = request.getSession();
30             session.setAttribute("SecurityContext", subject);
31         } catch (LoginException le) {
32             // handle exception
33         }
34     }
35 }
```

<sup>Θ</sup> 在JAAS术语中，待认证的用户或服务被称为一个subject（主体、主题）。

## 通用Context对象策略

### Context对象工厂策略

*Context*对象工厂用于获取不同类型的*Context*对象，这些*Context*对象的作用大同小异。使用一个工厂来获取*Context*对象，就能够让工厂来控制对象的创建、复制，并且在必要的时候还可以用池来存储这些对象（参见例6.31和例6.32）。

#### 例6.31 Context对象工厂

```

1  public class FrontController extends HttpServlet {
2
3      protected void processRequest(HttpServletRequest request,
4          HttpServletResponse response)
5          throws ServletException, java.io.IOException {
6
7      . . .
8      // 确定输入请求的类型
9      String requestType = getRequestType(request);
10
11     // 根据输入请求类型创建专门的
12     // RequestContext 对象实例
13     RequestContextFactory requestContextFactory =
14         RequestContextFactory.getInstance();
15
16     RequestContext requestContext =
17         requestContextFactory.getRequestContext(requestType, request);
18     . . .
19 }
```

#### 例6.32 RequestContextFactory（请求Context工厂）

```

1  public class RequestContextFactory {
2      static RequestContextFactory factory = new RequestContextFactory();
3
4      static public RequestContextFactory getInstance() {
5          return factory;
6      }
7
8      public RequestContext getRequestContext(String requestType,
9          ServletRequest request) {
10
11         RequestContext requestContext;
12         if (Constants.SOAP_PROTOCOL.equalsIgnoreCase(requestType) == true) {
13             requestContext = new SOAPRequestContextImpl(
14                 (HttpServletRequest) request);
15         } else {
16             requestContext = new HttpRequestContext(
17                 (HttpServletRequest) request);
```

```

18     }
19     return requestContext;
20 }
21 }
```

### Context对象自动复制策略

实现本模式的底层机制，与在对象之间复制状态有关。在映射关系相对简单的时候（比如，当需要在对象之间复制名称类似的属性时），可以采用反射（reflection）自动完成这一操作。这样，就不用手动复制对象的每一个状态，而是使用一个工具类自动复制bean的属性。比如，可以使用一个工具，在HttpRequest对象的参数名称和一个JavaBean的属性名称之间建立匹配，从而把HttpRequest对象的状态自动复制到JavaBean中。事实上，JSP容器内置的“通配符复制”<sup>Θ</sup>机制就提供了这种功能。但是这种自动复制功能只能由JSP页面使用，所以它没法作为一个独立的工具类起作用，也就无法让前端控制器或其他控制代码直接调用。

开发源码社区提供了好几种使用这个策略的工具，最著名的是Jakarta Commons子项目的BeanUtils包，它提供了多种bean内省<sup>Θ</sup>功能（见<http://jakarta.apache.org/commons/beanutils/api/index.html>）。例6.33是一个例子（另外请参见例6.34和例6.35）。

#### 例6.33 ContextObject的自动复制： RequestContextFactory（请求Context工厂）

```

1 // Context对象工厂
2 public class RequestContextFactory {
3
4     . .
5
6     public RequestContext createRequestContext(Class contextClass,
7         ServletRequest request) {
8         RequestContext requestContext = null;
9         try {
10             // 创建 ContextObject 类的实例
11             requestContext = (RequestContext) contextClass.newInstance();
12
13             // 使用内省，从请求实例中把值复制到
14             // ContextObject
15             AutoPopulateRequestContext.populateBean(requestContext, request);
16         } catch(java.lang.InstantiationException e) {
17             // 处理异常
18         } catch(java.lang	IllegalAccessException e) {
19             // 处理异常
20         }
21         return requestContext;
22     }
23 }
```

200

<sup>Θ</sup> 通配符复制，原文为wildcarding，指JSP能够默认地把表单字段的值复制到JavaBean中。

<sup>Θ</sup> 内省，原文为introspection，是JavaBean的一个特性，支持在对象之间自动复制属性。

**例6.34 ContextObject的自动复制: AutoPopulateRequestContext (自动复制请求Context)**

```

1 // Context对象助手
2 public class AutoPopulateRequestContext {
3
4     public static void populateBean(Object bean, ServletRequest request) {
5
6         Enumeration enum = request.getParameterNames();
7
8         while ( enum.hasMoreElements() ) {
9             String parameterName = (String)enum.nextElement();
10            if ( PropertyUtils.isWriteable(bean, parameterName) ) {
11
12                String values[] = request.getParameterValues(parameterName);
13                try {
14                    if ( values.length == 1 ) {
15                        PropertyUtils.setSimpleProperty(
16                            bean, parameterName, values[0]);
17                    } else {
18                        for ( int iValue=0; iValue < values.length; iValue++ )
19                            PropertyUtils.setIndexedProperty(
20                                bean, parameterName, iValue, values[iValue]);
21                    }
22                } catch (IllegalAccessException e) {
23                    // 处理异常
24                } catch (InvocationTargetException e) {
25                    // 处理异常
26                } catch (NoSuchMethodException e) {
27                    // 处理异常
28                }
29            }
30        }
31    }
32 }
```

201

**例6.35 ContextObject的自动复制: ProjectRegistrationRequestContext (项目注册请求Context)**

```

1 // 一个专用的Context对象
2 public class ProjectRegistrationRequestContext extends RequestContext {
3     public String getProjectDescription() {
4         return projectDescription;
5     }
6
7     public void setProjectDescription(String projectDescription) {
8         this.projectDescription = projectDescription;
9     }
10
11    public String getProjectManager() {
12        return projectManager;
```

```

13 }
14
15 public void setProjectManager(String projectManager) {
16     this.projectManager = projectManager;
17 }
18
19 public String getProjectName() {
20     return projectName;
21 }
22
23 public void setProjectName(String projectName) {
24     this.projectName = projectName;
25 }
26
27 public boolean validate() {
28     return true;
29 }
30
31 private String projectName;
32 private String projectDescription;
33 private String projectManager;
34 }

```

202

## 效果

- 提高了可重用性和可维护性

应用这一模式后，应用组件和子系统变得更加通用，可以由多种不同类型的客户端重用，因为应用接口并没有被与特定协议相关的数据类型污染。

- 提高了可测试性

使用*Context*对象模式，有助于消除对特定协议的依赖，如果代码依赖于这些协议，那么就会把运行时环境和一种容器（比如web服务器或者应用服务器）绑定。限制或消除了这种依赖之后，测试就变得更容易了，因为可以用自动测试工具（比如JUnit [Junit]）直接测试*Context*对象。

- 减少了对接口变化的限制

如果接口中包括与特定协议相关的对象，那么这些协议细节可能会限制今后的接口变化。而用*Context*对象封装了这些对象之后，接口只需接受*Context*对象，这样接口就不再绑定于特定协议，也就不再受到过多限制。这对于框架开发特别重要，但是对通常的开发也很有价值。

- 降低了性能

使用这个模式会带来轻微的性能影响，因为需要在不同对象之间传输状态。但是该模式提高了应用系统中各个子组件的可重用性和可维护性，这足以补偿性能上的影响了。

## 相关模式

- 拦截过滤器

在处理Web请求时，拦截过滤器可以使用ContextFactory（Context工厂）创建Context对象。

- 前端控制器

在处理Web请求时，前端控制器可以使用ContextFactory（Context工厂）创建Context对象。

- 应用控制器

在处理Web请求时，应用控制器可以使用ContextFactory（Context工厂）创建Context对象。

- 传输对象

传输对象专门负责在不同的远程层次之间传递状态，以减少网络通信，而Context对象则用于隐藏实现细节，提高重用和可维护性。

203

204

## 应用控制器

### 问题

要集中地、模块化地进行操作管理和视图管理。

在表现层，当处理每个请求的时候通常要作两种决定：

- 首先，把输入请求解析到一个操作（action），让它处理该请求。这叫做操作管理。
- 其次，还要选定返回给客户端的视图，并且把请求分派到这个视图。这叫做视图管理。

可以把操作管理和视图管理封装在应用系统的不同部分。把这部分内容封装在前端控制器里，确实能够集中这些功能，但是随着应用系统的增长，代码变得更加复杂，所以最好还是把这些内容分离出来，放在一组专门的类中，这样可以增进代码的模块化、可重用性和可扩展性。

### 约束

- 要重用操作控制和视图控制代码。
- 要提高请求处理逻辑的可扩展性，比如要能够递增地在系统中加入新的用例功能。
- 要增进代码的模块化和可重用性，便于扩展系统，并且便于在Web容器之外测试请求处理代码的各个单独部分。

### 解决方案

用一个应用控制器把请求处理组件（比如命令和视图）的获取和调用集中起来。

在表现层，把输入的请求参数映射到特定的请求处理类和负责处理请求的视图组件。这里，操作管理一词指为特定的请求找到合适的操作（action），并调用该操作；视图管理一词指把请求导航、分派到合适的视图或者视图生成机制。前端控制器是请求的一个集中访问点和控制器，而对于命令、视图的选择和分派则可以由一组独立的组件来完成。

205

把这些代码从前端控制器中分离出来有几个好处。首先，它把对请求的基本管理逻辑从前端控制器那些与特定协议相关的代码中分离出来。这增进了系统的模块化程度，并且提高了可重用性。应用控制器的一些组件可以用来处理不同通信渠道的请求，比如来自web应用的请求和来自Web Service的请求等。

**注意：**如果采用这种方式集中地、模块化地实现请求处理机制，那么请求处理的一些重要内容，包括验证、错误处理、认证和访问控制就能够轻松地以可插拔方式加入。

另外，把请求管理的这些内容从一个基于servlet的前端控制器中分离出来，就更便于在Web容器之外测试这些代码，从而简化了测试。当考虑采用后面介绍的操作处理和视图处理实现策略的时候，请记住视图处理工作往往是嵌入在命令处理工作内完成的（见后面“嵌套命令管理和视图管理”中的设计手记）。但是我们单独记录了每一种策略，因为它们在逻辑上构成了应用控制器的不同方面（参见图6-20）。

这个模式往往与模式目录中的其他模式结合使用。*Context*对象起到了在前端控制器和应用控制器之间通信的作用，隐藏了一些与协议相关的底层细节（比如HttpServletRequest）。*Context*对象通常由*Context*对象工厂创建，而往往可以让同一个类充当应用控制器和*Context*对象工厂这两种角色。在执行验证操作时，应用控制器常常需要和*Context*对象协作。

如果应用系统要求用户按一定顺序访问各个屏幕界面，就可以使用这个模式。对这个需求的实现思路在后面的“导航和流程控制策略”部分还有详细介绍。

最后，这个模式已经被实现为Struts框架[Struts]的一部分了，这个框架通过在配置文件中声明来实现映射，从而进行操作管理和视图管理。后面的“嵌套命令管理和视图管理”部分介绍了Struts是怎样使用这个模式的。

206

## 结构

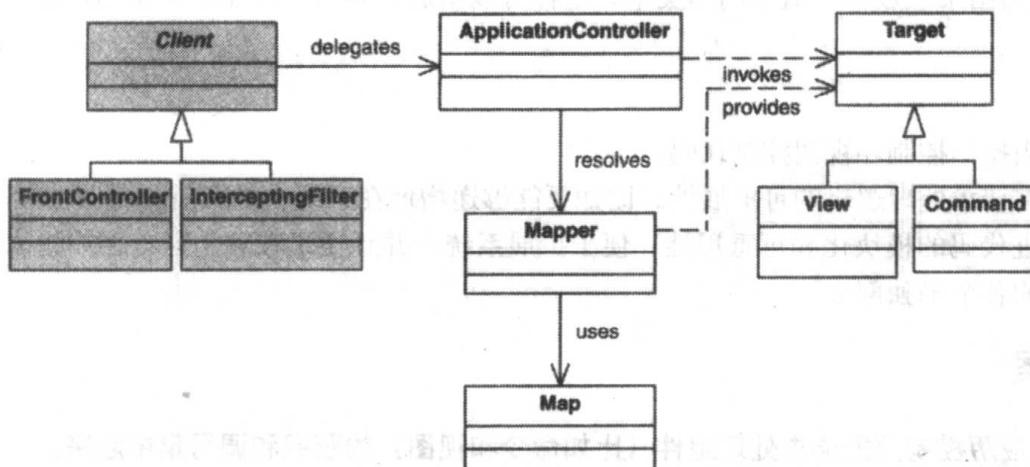


图6-20 应用控制器的类图

## 参与者和责任

### 客户端 (Client)

调用应用控制器（参见图6-21）。在表现层，通常是FrontController（前端控制器）或InterceptingFilter（拦截过滤器）充当这个角色。

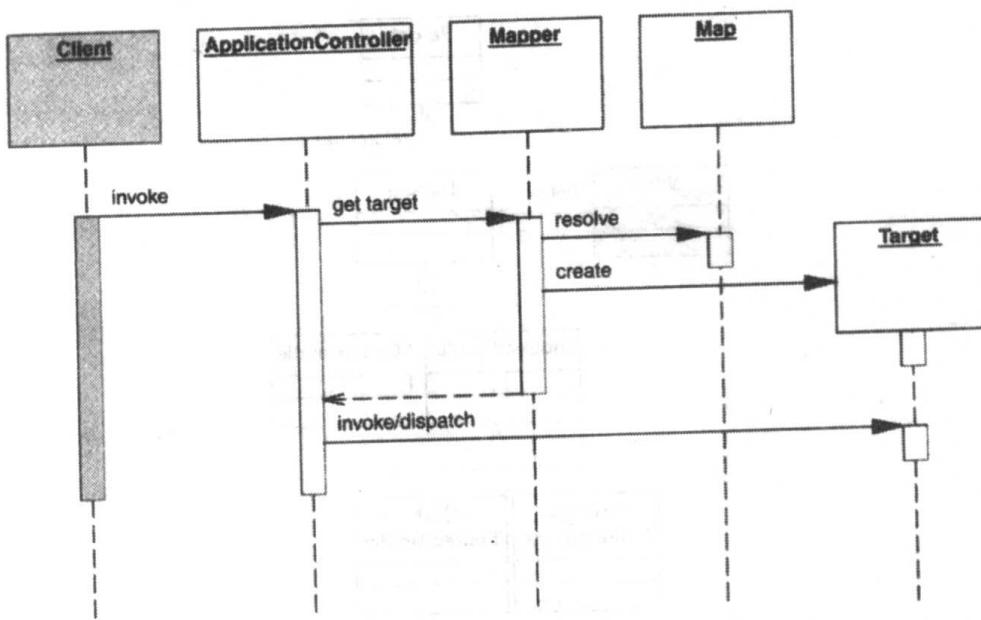


图6-21 应用控制器的序列图

207

### ApplicationController (应用控制器)

使用Mapper (映射器) 为输入请求解析出合适的操作和视图，然后再把请求委派或分派到这些操作和视图上。

### Mapper (映射器)

用一个Map (表)，把输入请求翻译成合适的操作和视图。映射器起到了工厂的作用。

### Map (表)

盛放对目标资源句柄的引用。表既可以用一个类实现，也可以用一个registry (注册器) 来实现（参见设计手记“句柄”）。

### Target (目标)

用来完成特定请求的资源，包括命令 (commands)、视图和样式表 (style sheets)。

## 设计手记：句柄

句柄由Mapper (映射器) 暴露并保存在Map (表) 中，既可以是一个直接句柄，也可以是一个间接句柄。

- 所谓直接句柄，指的是这样一种情况：实际对象就存放在Map中，可以直接被获取。
- 所谓间接句柄，指的则是：Map中存放的是一个对象，这个对象提供对实际资源的间接引用。

在表现层中，间接句柄的一个常见例子是一个字符串，它的值描述了相对于某个环境上下文的目标位置。另一个例子是其中包含对目标对象引用的一个对象。

图6-22体现了这些关系。

208

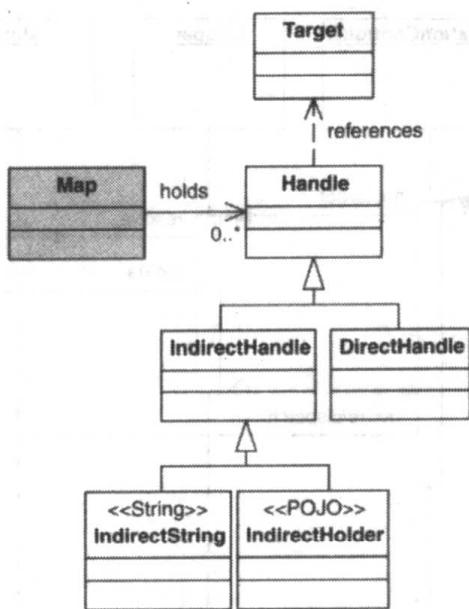


图6-22 句柄类图

## 策略

### 命令处理器策略

如果应用控制器获取并调用命令对象[GoF]，它也就被称为命令处理器。命令处理器负责管理命令对象的生命周期，而命令对象在一个请求内（而不是多个请求中）实现应用系统的用例和功能。命令对象可以通过命令工厂方法[GoF]获取，它封装了特定用例的处理逻辑，而且与调用它的对象之间只有松耦合。命令对象封装了用例相关的操作，并且具有通用接口，这样就可以使系统的可扩展性、模块化更高，而且也简化了加入新操作的工作。

既然命令通过一个通用的接口（比如说execute()）调用，就能够构建一个简单的、可重用的请求处理框架。可以聚合多个命令，从而创建出复合[GoF]命令，而且应用控制器还可以提供一种“补救”机制，比如就可以“撤销（undo）”一个特定的请求。  
209

而且还可以把应用控制器和它的底层命令用于多种不同的环境上下文中，因为使用这一模式能够让命令的寻址、获取和调用与任何特定的协议和环境无关。

图6-23中的类图显示了本模式中的通用角色是如何映射到与命令相关的组件上的，其中ApplicationController（应用控制器）起命令处理器的作用。

另一种实现方法是，让一个命令处理机<sup>①</sup>（Command Processor）[POSA]来执行命令，如图6-24所示。命令处理机管理命令对象，提供执行调度、日志，并且会缓存命令，以备以后的“撤销”操作。不过，既然ApplicationController（应用控制器）本身就能完成CommandProcessor（命令处理机）的很多责任，所以不大需要在物理上单独再编写这样一个类。

<sup>①</sup> 命令处理机（Command Processor）与命令处理器（Command Handler）其实在字面上差别不大。但对于实现而言，处理机往往由单独一个类实现，而处理器的功能则由应用控制器充任。详见下文。

在表现层，从前端控制器把请求委派给命令处理器是一种常见做法，在“前端控制器”一节中，这种经典用法被称为“命令加控制器策略”。以下的图6-25是这个做法的序列图，其中客户端（Client）的角色由前端控制器充任。

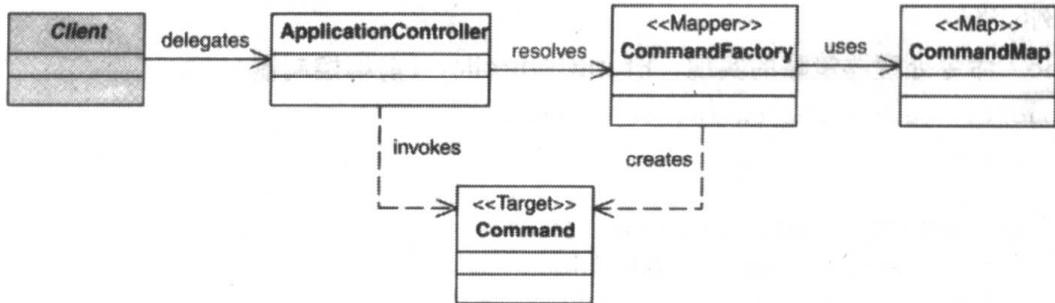


图6-23 命令处理器策略的类图

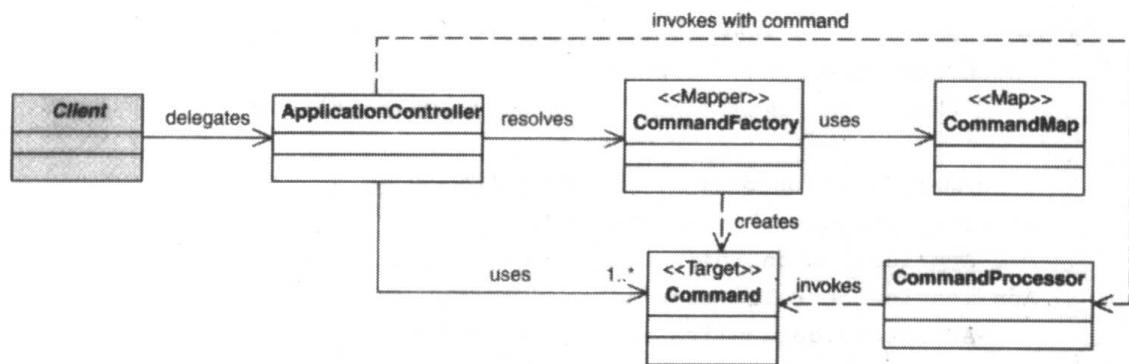


图6-24 使用CommandProcessor（命令处理器机）的命令处理器策略的类图

210

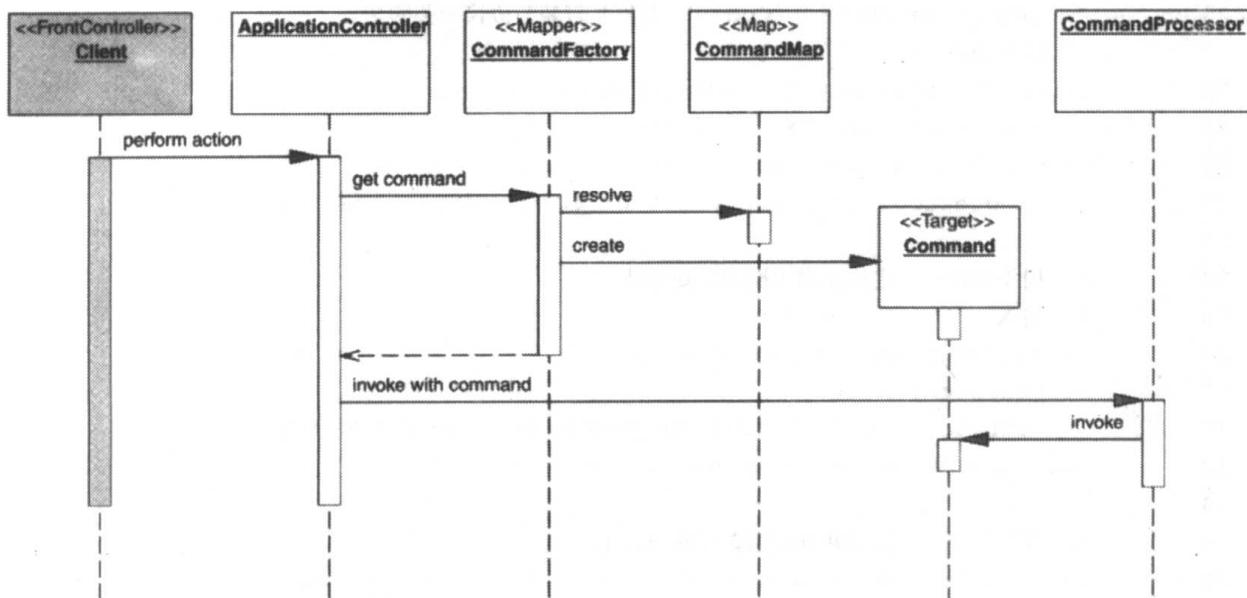


图6-25 命令处理器策略的序列图

在使用命令的过程中，ApplicationController（应用控制器）可以直接调用目标对象上的强类型方法。在这种情况下，ApplicationController（应用控制器）就会直接调用业务代表；如果

业务层处于远程，或者需要调用的是一个业务对象，那么ApplicationController就可以调用一个应用服务；而如果业务层就在本地，那么ApplicationController可以调用POJO服务门面。

例6.36示范了结合FrontController（前端控制器）实现命令处理器策略的方法（另外请参见例6.37和例6.38）。

### 例6.36 命令处理器策略的实现：FrontController（前端控制器）

```

1  public class FrontController extends HttpServlet {
2
3
4      /** 处理HTTP <code>GET</code> 和
5       * <code>POST</code> 方法的请求。
6       * @param request servlet 请求
7       * @param response servlet 响应
8       */
9
10     protected void processRequest(HttpServletRequest request,
11         HttpServletResponse response)
12         throws ServletException, java.io.IOException {
13
14     // 创建 ApplicationController (应用控制器) 处理请求
15     ApplicationControllerFactory ACFactory =
16         ApplicationControllerFactory.getInstance();
17     ApplicationController applicationController =
18         ACFactory.getApplicationController(request);
19     applicationController.init();
20
21     // 创建 ContextObject (Context对象) 封装与协议相关的
22     // 请求状态
23     RequestContextFactory requestContextFactory =
24         RequestContextFactory.getInstance();
25     RequestContext requestContext =
26         requestContextFactory.getRequestContext(request);
27
28     // 操作管理——定位并调用操作，处理特定
29     // 请求
30     ResponseContext responseContext;
31     responseContext =
32         applicationController.handleRequest(requestContext);
33     responseContext.setResponse(response);
34
35     // 视图管理——导航并分派到合适的视图上
36     applicationController.handleResponse(requestContext,
37         responseContext);
38     applicationController.destroy();
39
40 }

```

211

### 例6.37 ApplicationController (应用控制器) 接口

```

1 interface ApplicationController {
2     void init();
3     ResponseContext handleRequest(RequestContext requestContext);
4     void handleResponse(RequestContext requestContext, ResponseContext
5             responseContext);
6     void destroy();
7 }

```

212

### 例6.38 WebApplicationController (Web应用控制器) 实现

```

1 // 这个应用控制器的实现专门负责处理
2 // Web应用请求
3 class WebApplicationController implements ApplicationController {
4
5     public void init() { }
6
7     public ResponseContext handleRequest(RequestContext requestContext) {
8         ResponseContext responseContext = null;
9         try {
10             // 确定命令名称
11             String commandName = requestContext.getCommandName();
12
13             // 把命令名称解析到一个命令对象
14             CommandFactory commandFactory = CommandFactory.getInstance();
15             Command command = commandFactory.getCommand(commandName);
16
17             // 用CommandProcessor (命令处理器) 执行命令
18             CommandProcessor commandProcessor = new CommandProcessor();
19             responseContext = commandProcessor.invoke(command,
20                     requestContext);
21         } catch (java.lang.InstantiationException e) {
22             // 处理异常
23         } catch (java.lang.IllegalAccessException e) {
24             // 处理异常
25         }
26         return responseContext;
27     }
28     . .
29 }

```

### 视图处理器策略

如果应用控制器要负责获取、调用与视图相关的对象，那么也称它为“视图处理器”。应用控制器解析出合适的视图，并把请求分派到该视图，这个处理过程通常称作“视图管理”。前面的“命令处理器策略”已经介绍了“操作管理”，现在介绍视图管理，请注意这是两种不同的处理过程。但是在实际开发中，这两种处理过程通常是结合在一起的。关于这一点，请参见后面

213 的设计手记“嵌套命令管理和视图管理”。集中视图管理逻辑，能够增进系统的模块化程度、可扩展性、可维护性和可重用性。

本节后面的段落还表述了另外一种使用XSLT生成视图的策略，称为“转化处理器”。下面的图6-26显示了应用控制器模式的通用角色是如何映射到视图组件上的。ApplicationController（应用控制器）委派到一个Mapper（映射器），该Mapper是一个ViewFactory（视图工厂），而最终的目标（Target）是一个视图（View）。

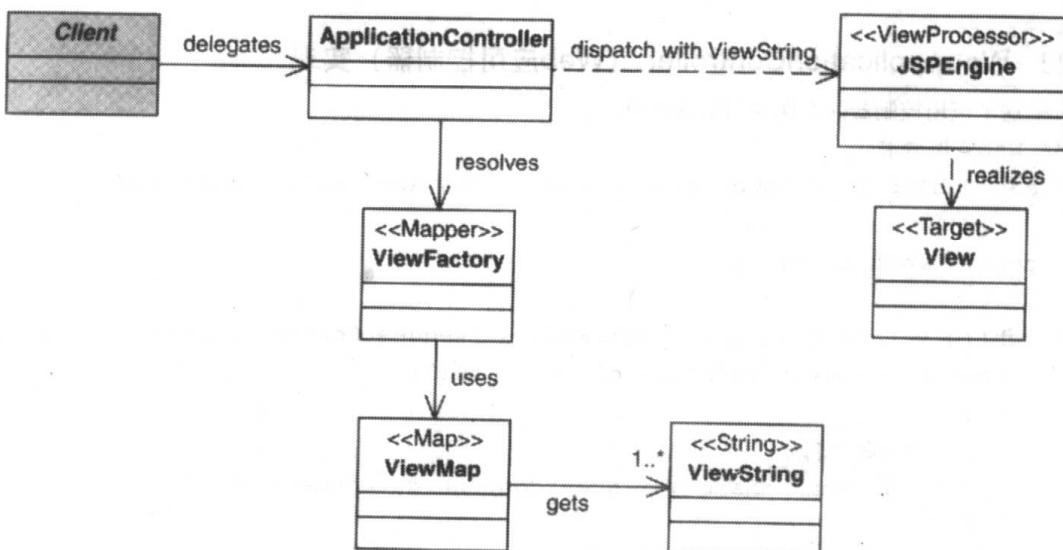


图6-26 视图处理器策略的类图

图6-27体现了这些角色之间的交互。

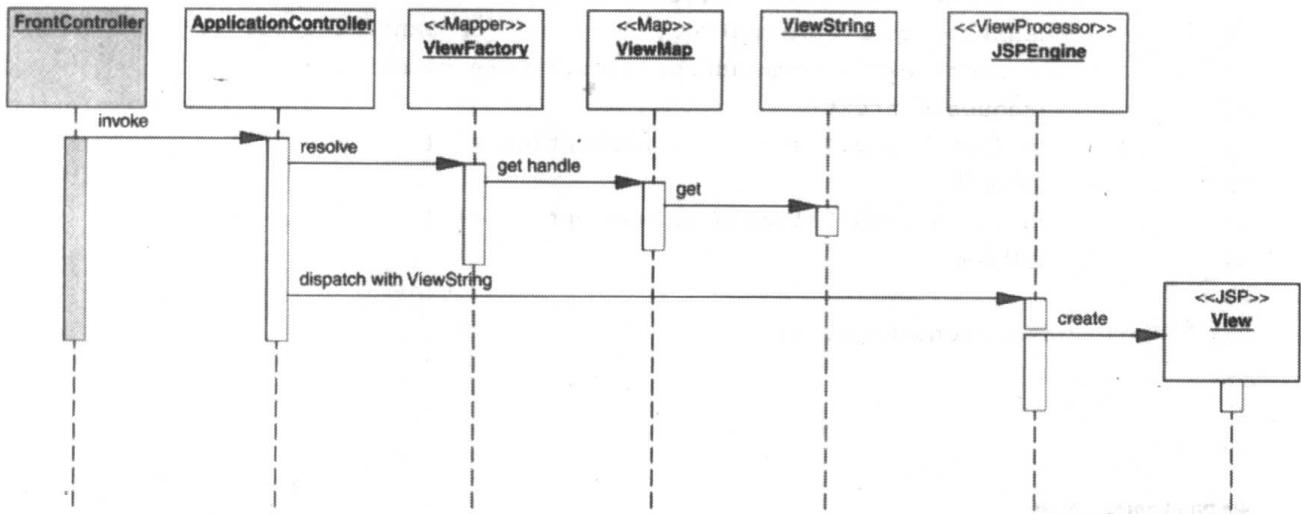


图6-27 视图处理器策略的序列图

图中的参与者“ViewString（视图字符串）”，事实上是视图（View）的一个间接句柄。它的作用与下面“嵌套命令管理和视图管理”中的参与者ViewHandle（视图句柄）类似。

### 设计手记：嵌套命令管理和视图管理<sup>⊖</sup>

虽然为了记录应用控制器模式的多种实现策略，在逻辑上要把操作管理和视图管理区分为两个处理过程，但其实这两种处理过程往往是相互混合、甚至是嵌套的。

下面的图6-28中表现了一种常见的情况，前端控制器作为客户端，把一个输入请求委派到应用控制器，应用控制器负责为请求解析出合适的命令，调用这个命令，再分派视图。这种应用场景通过Struts [Struts]框架实现。在命令调用完成、得到结果之后，基于这个结果，命令会准备并且返回一个间接视图句柄（见前面的“设计手记：句柄”）。

这样一来，命令也就与ApplicationController（应用控制器）一起承担了视图管理的某些责任（参见后面的例6.39至例6.44）。

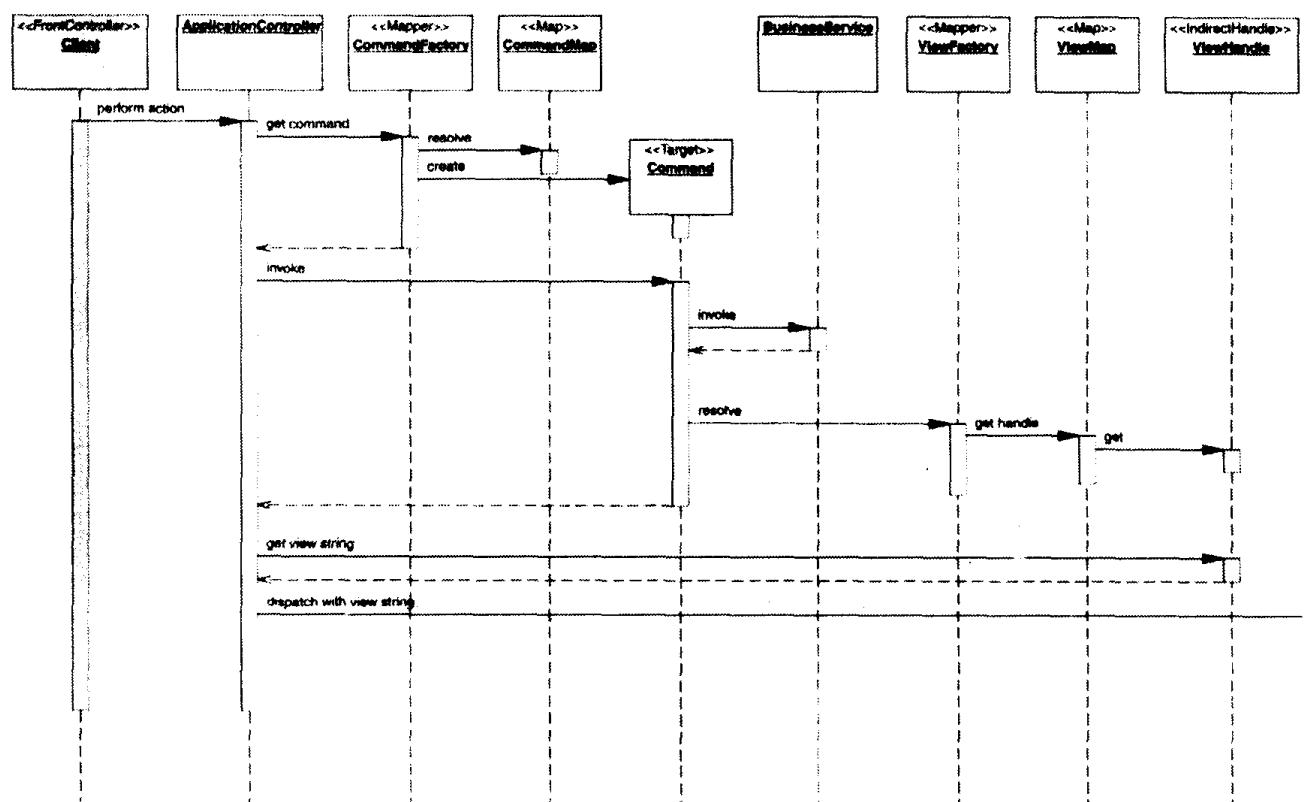


图6-28 嵌套命令管理和视图管理

### 应用控制器和Struts

Struts 框架 [Struts]中使用了应用控制器模式。Struts中的ActionServlet类是一个前端控制器，它把请求处理的任务委派给RequestProcessor（请求处理器），而RequestProcessor正是一个应用控制器。

RequestProcessor类起了命令控制器和视图处理器的作用，既完成操作管理，也完成视图管理。RequestProcessor使用了一个CommandMapper（命令映射器，上面例6.38的代码中称为CommandFactory）。这个映射器在Struts中是通过ModuleConfig（模块配置）组件和struts-

<sup>⊖</sup> 所谓嵌套（nested），是指命令管理代码和视图管理代码混杂在一起（通常是命令对象也负责部分视图管理）。详见本手记的讨论。

**[216]** config.xml实现的，它利用Struts ActionConfig（起到了CommandMap（命令映射）作用），为每个输入请求定位匹配的间接命令句柄。

然后，RequestProcessor使用这个命令句柄来获得命令的实例，并执行该命令；而这个命令则继而调用业务服务。ActionMapping（操作映射）则完成了ViewMapper（视图映射，见图6-28中的ViewFactory）功能，它被实现为一个Struts ActionForward（操作转发）类，能够获取匹配的POJO间接视图句柄，并用这个句柄来分派合适的视图。

#### 例6.39 ActionServlet示例代码

```

1 // 前端控制器
2 public class ActionServlet extends HttpServlet {
3
4     protected void process(HttpServletRequest request,
5         HttpServletResponse response) {
6
7         // 调用ApplicationController（应用控制器）进行命令管理和视图管理
8         getRequestProcessor(getModuleConfig(request)).
9             process(request, response);
10    }
11    .
12 }
```

#### 例6.40 ApplicationController（应用控制器）实现

```

1 public class RequestProcessor {
2     public void process(HttpServletRequest request,
3         HttpServletResponse response)
4         throws IOException, ServletException {
5
6         // 从请求对象确定命令Id（路径）
7         String path = processPath(request, response);
8         .
9         // 通过命令Id确定命令映射
10        ActionMapping mapping = processMapping(request, response, path);
11        if (mapping == null) {
12            return;
13        }
14
15        // 通过命令映射的安全设置检查用户的授权
16        if (!processRoles(request, response, mapping)) {
17            return;
18        }
19
20        // 通过CommandMap（命令映射）确定命令Context对象
21        ActionForm form = processActionForm(request, response, mapping);
22
23        // 从请求自动复制Context对象
24        processPopulate(request, response, form, mapping);
25 }
```

```

26     // 验证Context对象中的请求状态
27     if (!processValidate(request, response, form, mapping)) {
28         return;
29     }
30     // 创建或者获取命令实例，处理请求
31     Action action = processActionCreate(request, response, mapping);
32     if (action == null) {
33         return;
34     }
35     . . .
36
37     // 执行命令，获得ViewHandle（视图句柄）
38     ActionForward forward = action.execute(mapping, form, request,
39         response);
40
41     // 使用ViewHandle分派到合适的视图
42     processActionForward(request, response, forward);
43
44 }
45
46
47     // 处理响应
48     protected void processActionForward(HttpServletRequest request,
49             HttpServletResponse response, ActionForward forward)
50             throws IOException, ServletException {
51     . . .
52     // 分派到视图
53     RequestDispatcher rd =
54         getServletContext().getRequestDispatcher(uri);
55     rd.forward(request, response);
56     . . .
57
58 }
59 }
```

#### 例6.41 配置文件中的命令映射和视图映射部分

```

1   <action-mappings>
2   <!-- Edit mail subscription -->
3   <action path="/editSubscription" type=
4       "corepatterns>EditSubscriptionAction" attribute=
5       "subscriptionForm" scope="request" validate="false">
6       <forward name="failure" path="/jsp/mainMenu.jsp"/>
7       <forward name="success" path="/jsp/subscription.jsp"/>
8   </action>
9
10  </action-mappings>
```

### 例6.42 样本命令（Action）实现

```

1  public final class EditRegistrationAction extends Action {
2      public ActionForward execute(ActionMapping mapping, ActionForm form,
3          HttpServletRequest request, HttpServletResponse response)
4          throws Exception {
5          // 执行业务逻辑
6          .
7          // 使用viewMapper（视图映射器）来确定和获取匹配的viewHandle（视图句柄）
8          return (mapping.findForward("success"));
9      }
10 }

```

### 例6.43 视图处理器策略的实现：Web ApplicationController（Web应用控制器）

```

1  // 这是一个应用控制器实现，专门处理Web应用请求
2  class WebApplicationController implements ApplicationController {
3
4  .
5  // 处理视图导航，分派到匹配的视图
6  public void handleResponse(RequestContext requestContext,
7      ResponseContext responseContext) {
8      ViewFactory viewFactory = ViewFactory.getInstance();
9
10     // 根据逻辑视图名称、用户客户端类型、用户的区域设置等
11     // 确定视图模板
12     String viewTemplate = viewFactory.getViewTemplate(
13         requestContext, responseContext.getLogicalViewName());
14
15     // 分派到视图处理器
16     dispatch(requestContext.getRequest(), responseContext.getResponse(),
17             viewTemplate);
18 }
19
20     public void destroy() { }
21
22     // 分派器方法
23     private void dispatch(HttpServletRequest request,
24             HttpServletResponse response, String page) {
25
26         try {
27             RequestDispatcher dispatcher = request.getRequestDispatcher(page);
28             dispatcher.forward(request, response);
29         } catch(Exception e) {
30             // 处理异常
31         }
32     }
33 }

```

### 例6.44 视图工厂

```

1  public class ViewFactory {
2      private static ViewFactory ourInstance;
3
4      public synchronized static ViewFactory getInstance() {
5          if (ourInstance == null) {
6              ourInstance = new ViewFactory();
7          }
8          return ourInstance;
9      }
10
11     public String getViewTemplate(RequestContext requestContext,
12         String logicalViewName) {
13
14         String viewHandle;
15         Locale locale = requestContext.getUserLocale();
16         String userAgent = requestContext.getUserAgent();
17
18         // 通过视图映射确定ViewHandle (视图句柄)
19         viewHandle = getViewHandle(logicalViewName, userAgent, locale);
20         return viewHandle;
21     }
22
23     private String getViewHandle(String logicalViewName,
24         String userAgent, Locale locale) {
25
26         // 根据逻辑视图名称、用户客户端类型、用户的区域设置等
27         // 确定视图模板
28         ViewMapKey viewKey = new ViewMapKey(
29             logicalViewName, userAgent, locale);
30         return (String) viewMap.get(viewKey);
31     }
32
33     private ViewFactory() {
34         initializeViewMap();
35     }
36
37     private void initializeViewMap() {
38         // 初始化存根——通常要从一个ViewRegistry (视图注册器) 中
39         // 装载信息
40         // viewMap.put ( logicalViewName, viewTemplateName);
41     }
42
43     private Map viewMap = new HashMap();
44 }
```

220

### 转化处理器策略

如果 ApplicationController (应用控制器) 通过一个转化引擎 (transformation engine) 来获取并调用与视图相关的对象，那么我们就称其为 TransformHandler (转化处理器)。图6-29体现了本模式中的通用角色是如何映射到这些与视图相关的组件上的。

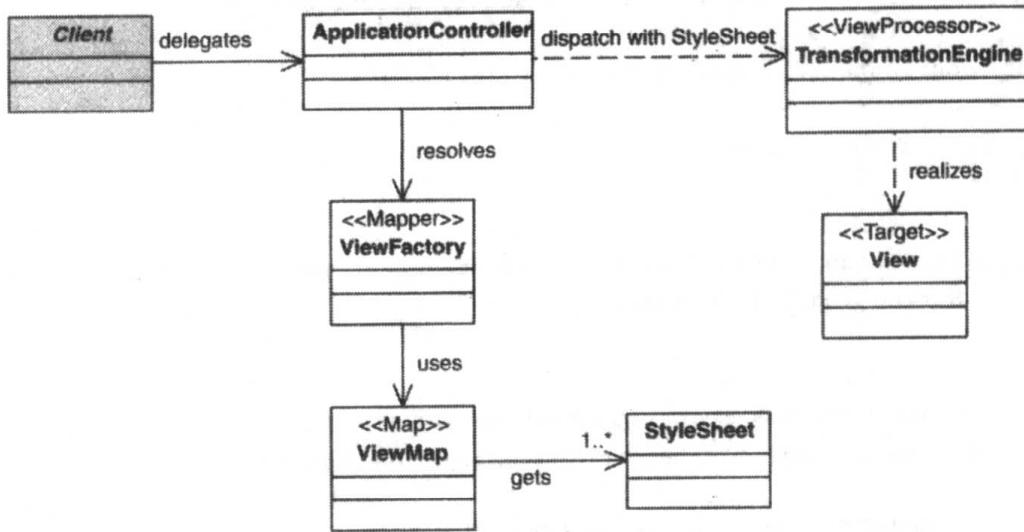


图6-29 转化处理器策略的类图<sup>①</sup>

221

ApplicationController (应用控制器) 和一个Mapper (映射器，这里就是ViewFactory (视图工厂)) 协作，最终目标 (Target) 则是一个视图 (View)，由样式表 (StyleSheet) 生成。

图6-30体现了这些组件之间的交互。

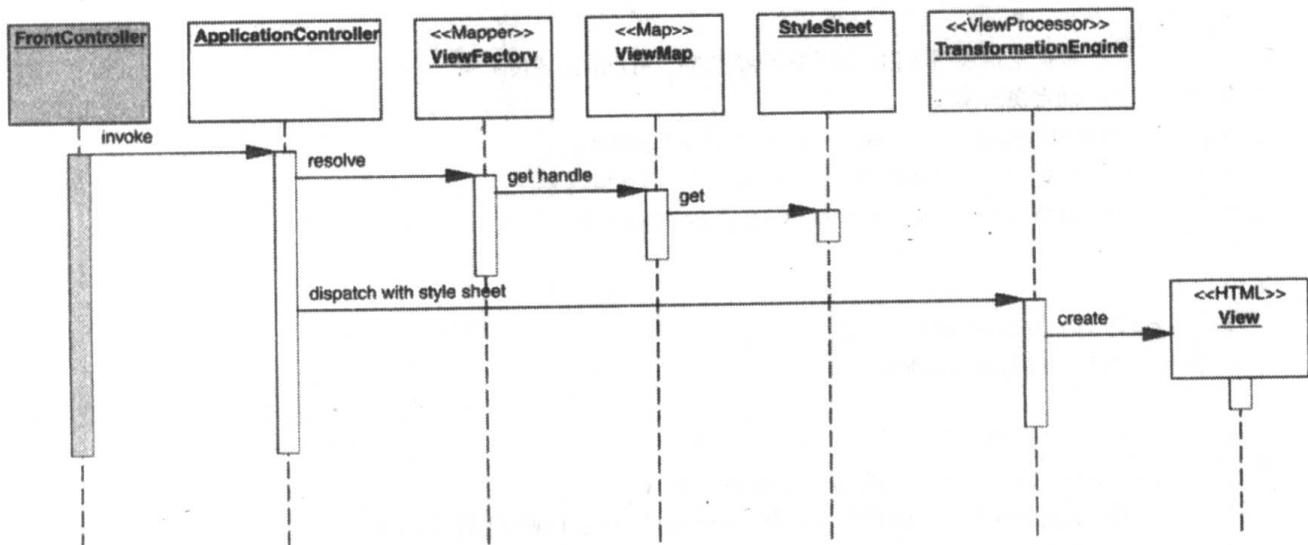


图6-30 转化处理器策略的序列图

<sup>①</sup> 原书此处和下一张图示（图6-30）、下一个例子（例6.45）的说明中，“转化处理器”作“Transformer Handler”，疑为“Transform Handler”之误。

转化处理器完成视图转化操作（见后文视图助手中“基于模板的视图转化策略”部分），因此是一种视图处理器，见例6.45及例6.46。

#### 例6.45 转化处理器策略：ApplicationControllerImpl（应用控制器实现）

```

1 // 应用控制器的一种实现，使用
2 // 视图转化处理器策略，处理
3 // Web应用的请求
4 public class ApplicationControllerImpl
5     implements ApplicationController {
6
7     . . .
8
9     public void handleResponse(RequestContext requestContext,
10         ResponseContext responseContext, ServletContext servletContext) {
11
12     ViewMapper viewMapper = ViewMapper.getInstance();
13     String stylesheet = viewMapper.getViewTemplate(requestContext,
14         responseContext.getLogicalViewName());
15
16     TransformHelper helper;
17     helper = new TransformHelper();
18
19     Reader xmlReader = new StringReader(
20         (String)responseContext.getData());
21     InputStream xslStream =
22         servletContext.getResourceAsStream(stylesheet);
23     helper.transform(requestContext.getRequest(),
24         responseContext.getResponse(), xmlReader, xslStream);
25 }
26 }
```

222

#### 例6.46 TransformHelper（转化助手）

```

1 class TransformHelper {
2
3     public void transform(ServletRequest request,
4         ServletResponse response, Reader xmlReader,
5         InputStream xslStream) {
6
7     try {
8         // SAXParserFactory (SAX解析器工厂)
9         SAXParserFactory parserFactory = SAXParserFactory.newInstance();
10        SAXParser parser = parserFactory.newSAXParser();
11        Source xmlSource = new StreamSource(xmlReader);
12        Source xslSource = new StreamSource(xslStream);
13        Result outputTarget = new StreamResult(response.getWriter());
14
15        Transformer transformer = TransformerFactory.newInstance().
16            newTransformer(xslSource);
```

```

17
18     transformer.transform(xmlSource, outputTarget);
19 } catch(Exception e) {
20     // 处理异常
21 }
22 }
23 }
```

### 导航和流程控制策略

通常，用户都应该按照一定的顺序在应用系统的各个屏幕界面之间转移。可以用好几种办法来控制这个流程（见例6.47及例6.48）。

- 最基本的办法就是，在允许用户访问某个页面之前，先检查一下是否满足某个前置条件（precondition）。比如，在允许用户察看账户信息之前，先检查一下用户是否已经登录。这个办法算得上是一种基本的基于规则（policy-based）的策略，但是，对于那些复杂的、多屏幕界面的需求（比如引导用户按照特定顺序通过一组界面）就不太适宜了。
- 如果需要按照当前系统的状态控制屏幕界面的呈现顺序，那么就可以用一个简单状态机来实现流程控制。很容易就能够实现通过文件声明来配置这些规则，所以这个策略也有助于减少导航/流程控制限制与代码之间的耦合。
- 最后，还有另外一个常见的流程控制需求，即“限制重复的请求”；对于这一点请参见第4章的“引入同步器令牌”重构。这种重构可以与应用控制器和前端控制器一起应用，从而控制重复发送的请求。

#### 例6.47 导航和流程控制策略

```

1 // 使用应用控制器实现流程控制
2
3 public class ApplicationControllerImpl
4     implements ApplicationController {
5
6     public ResponseContext handleRequest(Map requestContextMap) {
7         ResponseContext responseContext = null;
8         try {
9
10            // 根据ContextObject里的状态确定
11            // 请求的是哪种订单处理操作
12            String orderEvent = getOrderByEvent(requestContextMap);
13            String orderState = getOrderStatus(requestContextMap);
14
15            // 根据业务事件和当前状态创建命令
16            CommandAndViewFactory factory =
17                CommandAndViewFactory.getInstance();
18            CommandViewHandle result = factory.getCommand(
19                orderEvent, orderState);
20
21            // 从间接句柄获得命令实例
22            Command command =
```

```

23         (Command)result.getCommandHandle().newInstance();
24
25     // 执行业务层服务
26     responseContext = command.execute(requestContextMap);
27
28     // 设置视图名称
29     responseContext.setLogicalViewName(result.getViewName());
30 } catch(Exception e) {
31     // 处理异常
32 }
33 return responseContext;
34 }
35 . .
36 private String getOrderStatus(Map requestContext) {
37     String id = ((String[])requestContext.
38             get(OrderStatus.STATUS_PARAM))[0];
39     return id;
40 }
41
42 private String getOrderEvent(Map requestContext) {
43     String orderEvent = ((String[])requestContext.
44             get(Constants.REQ_OPCODE))[0];
45     return orderEvent;
46 }
47 }

```

#### 例6.48 CommandAndViewFactory (命令/视图工厂)

```

1 // 这既是命令工厂，也是视图工厂，它根据当前应用系统的状态
2 // 和请求的操作创建命令和视图实例
3 public class CommandAndViewFactory {
4     private static CommandAndViewFactory ourInstance;
5
6     public synchronized static CommandAndViewFactory getInstance() {
7         if (ourInstance == null) {
8             ourInstance = new CommandAndViewFactory();
9         }
10        return ourInstance;
11    }
12
13    private CommandAndViewFactory() {
14        commandviewMap = CommandViewMap.getInstance();
15    }
16
17    public CommandViewHandle getCommand(String event, String state) {
18        CommandViewHandle handle =
19            commandviewMap.getCommandViewHandle(event, state);
20        return handle;
21    }

```

224

225

```

22
23     public String getView(String event, String state) {
24         return commandviewMap.
25             getCommandViewHandle(event, state).getViewName();
26     }
27
28     CommandViewMap commandviewMap;
29 }
30
31 class CommandViewMap {
32
33     private static CommandViewMap ourInstance;
34
35     public synchronized static CommandViewMap getInstance() {
36         if (ourInstance == null) {
37             ourInstance = new CommandViewMap();
38         }
39         return ourInstance;
40     }
41
42     private CommandViewMap() {
43         initialize();
44     }
45
46     // 初始化命令视图映射，放入所有合法的状态/事件组合
47     private void initialize() {
48         addEventStateEntry(
49             OrderEvents.EnterOrder, OrderStatus.ORDER_NEW,
50             PlaceOrderCommand.class,
51             "/jsp/ApplicationController/OrderPlaced");
52
53         addEventStateEntry(OrderEvents.ApproveOrder,
54             OrderStatus.ORDER_PLACED, ApproveOrderCommand.class,
55             "/jsp/ApplicationController/OrderApproved");
56
57         addEventStateEntry(OrderEvents.DeclineOrder,
58             OrderStatus.ORDER_PLACED,
59             DeclineOrderCommand.class,
60             "/jsp/ApplicationController/OrderDeclined");
61
62         addEventStateEntry(OrderEvents.TerminateOrder,
63             OrderStatus.ORDER_PLACED, TerminateOrderCommand.class,
64             "/jsp/ApplicationController/OrderTerminated");
65
66         addEventStateEntry(OrderEvents.ShipOrder,
67             OrderStatus.ORDER_APPROVED, TerminateOrderCommand.class,
68             "/jsp/ApplicationController/OrderShipped");

```

```
69
70     addEventStateEntry(OrderEvents.TerminateOrder,
71         OrderStatus.ORDER_APPROVED, TerminateOrderCommand.class,
72         "/jsp/ApplicationController/OrderTerminated");
73
74     addEventStateEntry(OrderEvents.TerminateOrder,
75         OrderStatus.ORDER_DECLINED, TerminateOrderCommand.class,
76         "/jsp/ApplicationController/OrderTerminated");
77     . . .
78 }
79
80 void addEventStateEntry(String event, Object state,
81     Class domainCommand, String viewName) {
82     CommandViewHandle newResponse =
83         new CommandViewHandle(domainCommand, viewName);
84     if ( !events.containsKey(event))
85         events.put(event, new HashMap());
86     getEventMap(event).put(state, newResponse);
87 }
88
89 private Map getEventMap(String key ) {
90     return (Map)events.get(key);
91 }
92
93 public CommandViewHandle getCommandViewHandle(
94     String operation, String state) {
95     return (CommandViewHandle)getEventMap(operation).get(state);
96 }
97 private Map events = new HashMap();
98 }
99
100 class CommandViewHandle {
101     public CommandViewHandle(Class domainCommand, String viewName) {
102         this.domainCommand = domainCommand;
103         this.viewName = viewName;
104     }
105
106     public Class getCommandHandle() {
107         return domainCommand;
108     }
109
110     public String getViewName() {
111         return viewName;
112     }
113
114     private Class domainCommand;
115     private String viewName;
116 }
```

### 设计手记：事件监听器——ServletContext、HttpSession和定制事件监听器

表6-3总结了一些与应用的生命周期有关的事件和一些基于属性的事件，这些事件与ServletContext或 HttpSession有关，而请求处理的一些方面也要依赖于这些事件。另外，还可以使用定制事件监听器和定制的事件处理机制来触发请求处理中的一些特定内容。

- 生命周期：**当servlet context准备就绪可以处理第一个请求的时候或当servlet context濒临被销毁的时候，容器都会发出信号。
- 属性改变：**当servlet context中的属性发生添加、替换或删除的时候，容器会发出信号。
- 生命周期：**当创建了一个HttpSession的时候，当HttpSession超时过期的时候或者当它失效的时候，容器会发出信号。
- 属性改变：**当HttpSession对象中的属性发生添加、替换或删除的时候，容器会发出信号。

表6-3 servlet容器支持的事件类型

事件类型	说明	监听器接口
<b>servlet context事件</b>		
生命周期	servlet context刚刚创建完毕，已经就绪可用，可以开始处理第一个请求；或者是servlet context马上就要销毁	javax.Servlet.ServletContextListener
属性改变	servlet context中的属性发生添加、替换或删除	javax.Servlet.ServletContextAttributesListener
<b>HttpSession事件</b>		
生命周期	HttpSession刚刚创建完毕，或是它失效或超时过期	javax.Servlet.http.HttpSessionListener
属性改变	HttpSession对象中的属性发生添加、替换或删除	javax.Servlet.http.HttpSessionAttributesListener

### 消息处理策略（见例6.49及例6.50）

### Web Service

应用控制器常常在表现层中与前端控制器一起使用，负责操作处理和视图处理。不过，应用控制器也还可以用于其他的层次或环境中。比如说，当Web Service的请求需要路由和操作管理的时候，就可以使用应用控制器来处理这种请求。所以，虽然在一般情况下应用控制器用于表现层，但也能以各种不同的方式把它用于其他层次。

带用于Java的附件API的SOAP（SOAP With Attachments API for Java，又称SAAJ）[SAAJ]提供了处理SOAP消息的功能，开发者可以使用它以过滤器形式构造消息处理器。必要时应该使用SOAP消息处理器；使用了它，就能获得非常强大的处理机制。

### 定制SOAP消息处理策略

### Web Service

对于一个Web Service请求来说，应用控制器负责消息处理，其中包括要调用消息处理器链完成预处理和/或后处理。这个处理器链中的处理器之间相互处于松耦合关系，每一个处理器都是一个拦截过滤器，正如拦截过滤器一节的“定制SOAP过滤器策略”所描述。

图6-31和图6-32是本策略中的参与者和它们的交互方式。

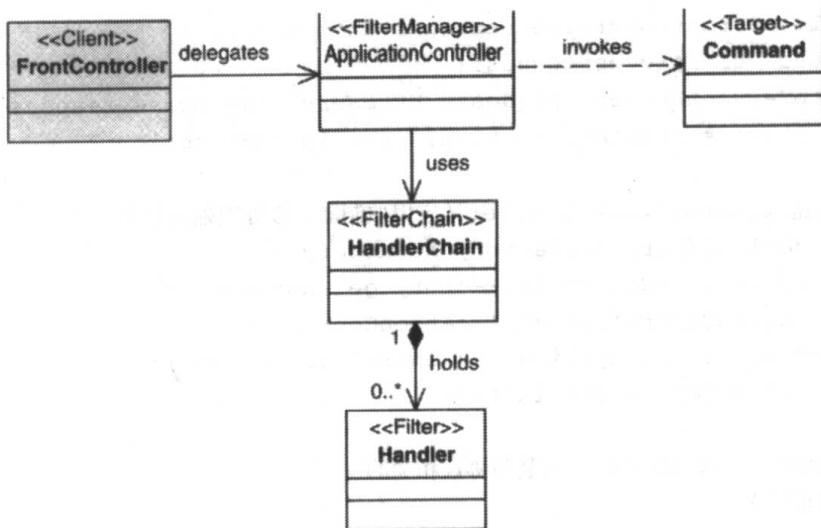


图6-31 定制SOAP处理器策略

230

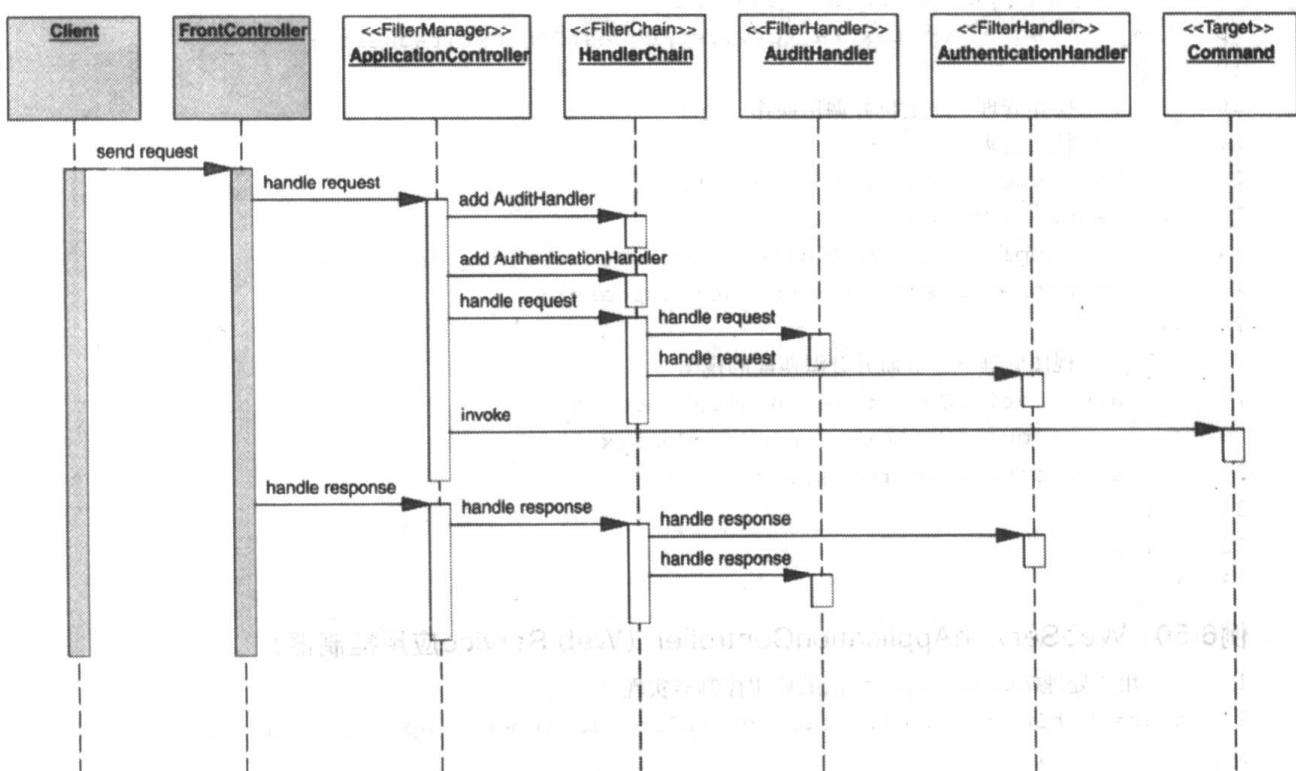


图6-32 定制SOAP处理器策略序列图

231

通常，前端控制器充当了SOAP请求的集中访问点。在本策略中，前端控制器把请求委派给应用控制器，应用控制器则使用定制SOAP过滤器来完成消息的预处理和后处理。然后，应用控制器进行操作管理和视图管理，其中也包括相关的验证和错误处理。其实，在这个策略里，所谓“视图”就是准备返回给客户端的响应。

**例6.49 SOAP 消息处理策略**

```

1  public class FrontController extends HttpServlet {
2      protected void processRequest(
3          HttpServletRequest request, HttpServletResponse response)
4          throws ServletException, java.io.IOException {
5
6      // 创建 ApplicationController (应用控制器) 以处理输入请求
7      ApplicationControllerFactory ACFactory =
8          ApplicationControllerFactory.getInstance();
9      ApplicationController applicationController =
10         ACFactory.getApplicationController(request);
11     applicationController.init();
12
13    // 创建 ContextObject, 封装与协议相关的
14    // 请求状态
15    RequestContextFactory requestContextFactory =
16        RequestContextFactory.getInstance();
17    RequestContext requestContext =
18        requestContextFactory.getRequestContext(request);
19
20    // 操作管理——定位并调用操作, 处理
21    // 特定请求
22    ResponseContext responseContext;
23    responseContext =
24        applicationController.handleRequest(requestContext);
25    responseContext.setResponse(response);
26
27    // 视图管理——导航并分派匹配的视图
28    applicationController.handleResponse(
29        requestContext, responseContext);
30    applicationController.destroy();
31 }
32 . .
33 }
```

**例6.50 WebService ApplicationController (Web Service应用控制器)**

```

1 // 用于处理 Web Service 请求的应用控制器实现
2 class WebService ApplicationController implements ApplicationController {
3
4     HandlerChain handlerChain;
5
6     public WebService ApplicationController() { }
7
8     public void init() {
9         handlerChain = new HandlerChain();
10        handlerChain.addHandler(new AuditHandler());
11        handlerChain.addHandler(new AuthenticationHandler());
```

```

12     }
13
14     public ResponseContext handleRequest(RequestContext requestContext) {
15         ResponseContext responseContext = null;
16         try {
17             SOAPRequestContext soapRequestContext =
18                 (SOAPRequestContext)requestContext;
19
20             // 从输入的HTTP流创建SOAPMessage (SOAP消息)
21             SOAPMessage soapMessage = soapRequestContext.getSOAPMessage();
22
23             // 通过消息处理器执行SOAPMessage的预处理
24             handlerChain.handleRequest(soapMessage);
25
26             // 根据与协议相关的请求确定命令名称
27             String commandName = soapRequestContext.getCommandName();
28
29             // 把命令名称解析为命令对象
30             CommandFactory commandFactory = CommandFactory.getInstance();
31             Command command = commandFactory.getCommand(commandName);
32
33             // 使用CommandProcessor (命令处理机) 调用命令
34             CommandProcessor commandProcessor = new CommandProcessor();
35             Object result = commandProcessor.invoke(command, requestContext);
36
37             // 根据命令的结果创建响应context
38             responseContext = ResponseContextFactory.getInstance().
39                 createResponseContext(result, null);
40         } catch(java.lang.InstantiationException e) {
41             // 处理错误
42         } catch(java.lang.IllegalAccessException e) {
43             // 处理错误
44         }
45         return responseContext;
46     }
47
48     public void handleResponse(
49         RequestContext requestContext, ResponseContext responseContext) {
50         try {
51
52             // 根据前面获得的命令结果创建
53             // 响应的SOAPResponse (SOAP响应) 消息
54             SOAPMessage soapMessage = null;
55             Object payload = responseContext.getData();
56
57             soapMessage = SOAPHttpFacade.createSOAPMessage(payload);
58             if (soapMessage != null) {
59                 // 使用SOAP消息处理器进行响应消息的后处理

```

```

60         handlerChain.handleResponse(soapMessage);
61
62         // 需要调用saveChanges（保存变更）方法，因为我们要使用
63         // MimeHeaders（MIME头）来设置HTTP响应信息。这些
64         // MimeHeaders将作为保存的一部分被生成出来。
65
66         if (soapMessage.saveRequired()) {
67             soapMessage.saveChanges();
68         }
69
70         HttpServletResponse response = (HttpServletResponse)
71         responseContext.getResponse();
72         response.setStatus(HttpServletResponse.SC_OK);
73
74         SOAPHttpFacade.putHeaders(
75             soapMessage.getMimeHeaders(), response);
76         writeResponse(
77             (HttpServletResponse)responseContext.
78                 getResponse(), soapMessage);
79     }
80     } catch (SOAPException e) {
81         // 处理异常
82     }
83 }
84
85 public void destroy() { }
86
87 private void writeResponse(
88     HttpServletResponse response, SOAPMessage soapResponse) {
89     try {
90         // 把消息输出到响应流中
91         response.setContentType("text/xml"); // SOAP 1.1消息
92         OutputStream outputStream = response.getOutputStream();
93         soapResponse.writeTo(outputStream);
94         outputStream.flush();
95     } catch(SOAPException e) {
96         // 处理异常
97         response.setStatus(HttpServletResponse.SC_NO_CONTENT);
98     } catch(java.io.IOException e) {
99         // 处理异常
100        response.setStatus(HttpServletResponse.SC_NO_CONTENT);
101    }
102 }
103 . . .
104 }
```

234

后面的图6-33表现了JAX-RPC 消息处理策略。

### JAX-RPC 消息处理策略

对于一个Web Service请求，应用控制器负责管理请求处理的各个方面，包括管理一个消息处理器链，以完成预处理和后处理。如果这些消息处理器执行的是与客户端相关的逻辑，那么它们就会在Web Service的客户端层实现。如果这些处理器中包含了对于接收端的各个请求和响应共通的代码，那么它们就会在表现层或集成层实现。这些松耦合的处理器其实是一些拦截过滤器，已经在拦截过滤器的“JAX-RPC过滤器策略”部分介绍过了。

通常，前端控制器充当了SOAP请求的集中访问点。在本策略中，前端控制器把请求委派到应用控制器，应用控制器再使用JAX-RPC过滤器策略执行消息的预处理和后处理（见图6-33）。然后应用控制器完成操作管理和视图管理，其中包括适当的验证和错误处理。其实，在这个策略里，所谓“视图”就是准备返回给客户端的响应。消息处理器则是使用SAAJ构建的。

本策略和前面介绍的定制SOAP消息处理策略之间的最大不同在于，本策略中JAX-RPC运行时引擎完成了本模式中的大多数参与者的责任。

235

见后面的例6.51。

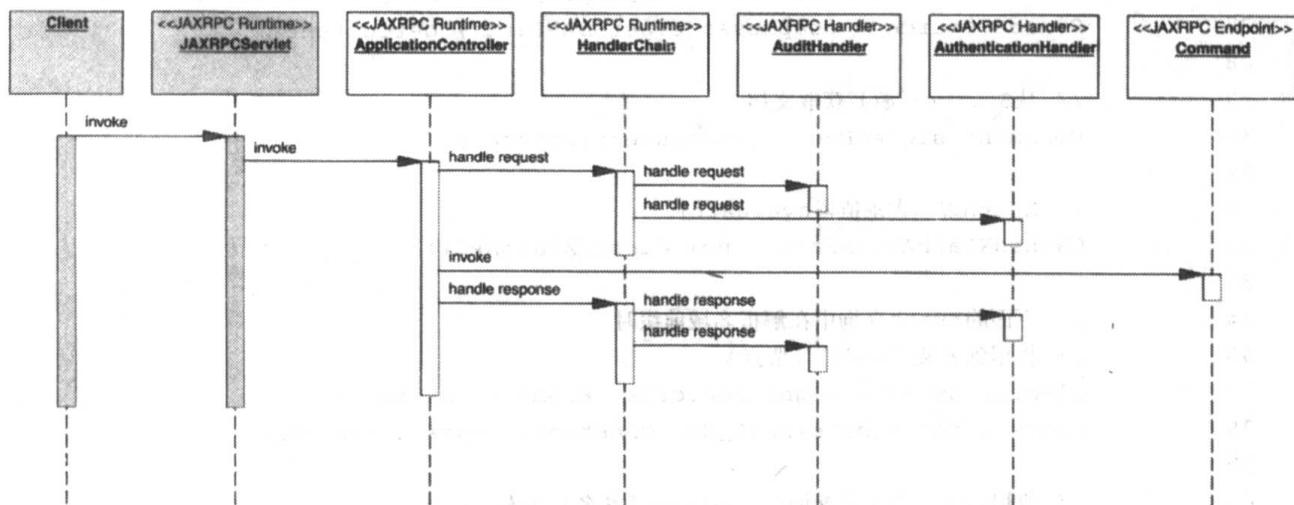


图6-33 JAX-RPC消息处理策略的序列图<sup>①</sup>

236

### 例6.51 JAX-RPC消息处理策略

```

1 import javax.xml.rpc.handler.Handler;
2 import javax.xml.rpc.handler.soap.SOAPMessageContext;
3 import javax.xml.soap.SOAPMessage;
4 ...
5 import org.apache.xml.security.signature.XMLSignature;
6 import org.apache.xpath.CachedXPathAPI;
7 import org.apache.xml.security.utils.Constants;
8
9 public class AuthenticationHandler extends GenericHandler
  
```

<sup>①</sup> 本图和下面的例6.51的说明文字中，策略名称都写作“JAX-RPC Message-Handler”（JAX-RPC消息处理器）。译者据前面的段落标题更正。

```

10     implements Handler {
11
12     private String DSIG_NS = "xmlns:ds";
13     private String DSIG_SIGNATURE_SEARCH_EXPR = "//ds:Signature";
14     String BaseURI = "http://xml-security";
15
16     public AuthenticationHandler() { }
17
18     public void init(HandlerInfo config) {
19         org.apache.xml.security.Init.init();
20     }
21
22     public boolean handleRequest (MessageContext context) {
23         try {
24             SOAPMessageContext soapMessageContext =
25                 (SOAPMessageContext) context;
26             SOAPMessage soapMessage = soapMessageContext.getMessage();
27             Source source = soapMessage.getSOAPPart().getContent();
28
29             // 从Source (源) 获取文档
30             Document signedDoc = getDocument(source);
31
32             // XPath表达式求值器evaluator
33             CachedXPathAPI xPath = new CachedXPathAPI();
34
35             // 下面的XPath查询中在解析名域前缀时
36             // 将用到名域 “Node” (节点)
37             Element nsctx = signedDoc.createElement("nsctx");
38             nsctx.setAttribute(DSIG_NS, Constants.SignatureSpecNS);
39
40             // 使用XPath表达式查询Signature (签名) 元素
41             Element signatureElem =
42                 (Element)xPath.selectSingleNode(
43                     signedDoc, DSIG_SIGNATURE_SEARCH_EXPR, nsctx);
44
45             // 检查并确认文档有签名
46             if (signatureElem == null) {
47                 // 处理无签名的情况
48                 System.out.println("The document is not signed");
49                 return false;
50             }
51
52             // 验证签名
53             XMLSignature signature = new XMLSignature(signatureElem, BaseURI);
54             boolean verify = signature.checkSignatureValue(
55                 signature.getKeyInfo().getPublicKey());
56

```

237

```

57     // 给出验证消息
58     System.out.println("The signature is" + (verify ? "": " not ") +
59         "valid");
60
61     return verify;
62 } catch(Exception e) {
63     // 处理异常
64 }
65     return false;
66 }
67
68 public boolean handleResponse(MessageContext context) {
69     return true;
70 }
71 .
72 }

```

## 效果

- 提高了模块化程度

把常见的操作管理和视图管理代码分离出来，形成单独的类，提高了整个应用系统的模块化程度。也更便于测试了，因为应用控制器中的各方面功能并不依赖于Web容器。

- 提高了可重用性

可以重用通用的、模块化的组件。

238

- 提高了可扩展性

请求处理的机制预留了添加功能的可能，而且功能添加也独立于协议和网络访问部分的代码。因为流程控制可以通过在配置文件中的声明设置，所以也就降低了导航/流程控制规则与代码之间的耦合，即使不重新编译或修改代码，也可以修改这些规则。

## 相关模式

- 前端控制器

前端控制器使用应用控制器来进行操作管理和视图管理。

- 服务定位器

服务定位器负责服务的定位和获取。服务定位器是一个粗粒度对象，常常在底层使用某种复杂的定位机制，但不做路由管理。它也不进行视图管理。

- 命令处理机 [POSA]

命令处理机管理命令的调用，提供了执行调度、日志、撤销/重复等功能。

- 命令模式 [GoF]

命令用一个对象封装了请求，把请求和对请求的调用区别开来。

- 复合模式 [GoF]

复合将对象组合成树形结构以表示“部分-整体”的层次结构，这样客户端对单个对象和复

合对象的访问就能具有一致性。.

- **应用控制器[PEAA]**

Martin Fowler[PEAA]对应用控制器的描述似乎主要是关于使用状态机控制用户的导航，类似于我们在“导航和流程控制策略”部分介绍的内容。但是[PEAA]中的应用控制器和我们介绍的**239**应用控制器，具有同样的核心意图。

## 视图助手

### 问题

需要把视图和相关的处理逻辑分离开。

如果在视图组件里把控制逻辑、数据访问逻辑和格式逻辑都混在一起，就会在模块化、重用、维护和开发团队的角色分工上造成麻烦。把模型从视图和控制代码中区分出来是一个高层次的设计目标，而如果在视图中把各种逻辑混在一起，恰恰就违背了这一设计目标。

在表现层把模型、视图和控制组件相互分离开，这是一个很重要的设计目标。前端控制器能够封装控制逻辑，但是模型组件和显示组件的分离也很重要，是一种必须加以重视的设计考虑。

所谓请求处理，总与两种类型的工作有关：对请求的控制处理和对视图的处理。而在“视图处理”里面，又分为两个阶段：**视图准备阶段**和**视图创建阶段**，这两个阶段都发生在web层，但彼此又有很大区别，每个阶段涉及的处理逻辑与另外一个阶段全不相干。

- 视图准备阶段涉及的是请求处理、操作管理和视图管理。请求被解析到一个具体的操作上，然后调用该操作，确定匹配的视图，再把请求分派给该视图。
- 随后，在视图创建阶段，视图从模型中获取内容，并使用助手类来获取和调整模型的状态。模型的内容经常存放在传输对象里，把内容抽取出来并调整妥当之后，这些内容就会被纳入静态的文本视图模板，按要求格式化、转换，最后生成了一个动态的响应。

如果在视图创建中采用了JSP或者其他基于模板的视图形式，就能在视图中加入Java scriptlet。但是，如果这样使用scriptlet不便于重用处理逻辑，而且会降低整个应用系统的灵活性。

而且，视图中的程序逻辑代码通常都起着控制作用，或者是执行某些视图准备操作（比如获取内容）。随着应用系统的复杂度上升，在视图组件里把控制逻辑、数据访问逻辑和格式逻辑都混在一起，就会在模块化、重用、维护和开发团队的角色分工上造成麻烦。把模型从视图和控制代码中区分出来是一个高层次的设计目标，而如果在视图中把各种逻辑混在一起，恰恰就违背了这一设计目标。<sup>①</sup>

### 约束

- 要使用基于模板的视图，比如JSP。

---

<sup>①</sup> 我们看到本段的最后两句和本节开头部分重复。事实上，这类重复在书中还有很多。译者对类似的文风不敢恭维。正如编写代码时一样，论著中的重复也降低了整个作品的可维护性。我们注意到，书中另有多处错漏都是因为copy and paste引起的。

- 要避免把程序逻辑放到视图里。
- 要把程序逻辑从视图中分离出来，这样就可以更加明确软件开发者和网页设计者之间的分工。

## 解决方案

使用视图封装显示格式的代码，使用助手封装视图处理逻辑。助手通过POJO、定制标记或标记文件的形式实现，视图把处理逻辑交给这些助手完成。助手在视图和模型之间充当了适配器的作用，同时也会执行一些与格式逻辑相关的处理，比如生成一个HTML表（HTML table）等。

本模式有助于区分视图和助手之间的责任。在本模式中，主要抽象出了视图组件和助手组件，二者各自负有专门的责任，所以本模式也称作“**视图和助手**”。

本模式主要用于基于模板的视图，比如JSP。其中，用静态的模板文本和标记来定义整个的模板视图，程序逻辑则封装在助手中，在模板中只包括对这些助手的引用。

把处理逻辑封装在助手（而不是视图）中，能够使应用系统更加模块化，使组件更易于重用。如果把处理逻辑放在基于模板的视图中，那么常见的重用办法就只能是把这个逻辑复制、粘贴到其他地方。这样的复制，会给系统维护带来困难，因为如果要修改一个bug，就可能要改动多个文件。所以，如果JSP视图中有大量的scriptlet代码，就应该采用本模式，以实现“隔离不同逻辑”的重构。

采用本模式的总体目标，在于把不同的处理逻辑（比如控制逻辑、格式逻辑、业务逻辑和数据访问逻辑等）隔离在视图之外。

- 业务逻辑通常由模型对象（比如传输对象或业务对象）完成。
- 数据访问逻辑封装在数据访问对象中。
- 控制逻辑由前端控制器，再加上命令和助手共同完成；而格式逻辑则由助手完成。

可以看到，在助手对象和模型对象之间存在某种重叠，因为模型的一项主要责任就是管理那些实现请求的业务逻辑，而这种操作常常要由web层的助手对象来初始化和调用。事实上，模型往往封装在业务对象中，而助手会把请求委派给这些业务对象。视图助手对象的主要责任在于：完成模型的格式化和调整，并将之用于特定的视图。

JSP标准标记库（JavaServer Pages Standard Tag Library）[JSTL]提供了一组标准标记，支持一些常见的需求，比如迭代、条件选择逻辑等。使用了这些标记，就不用再以scriptlet代码的形式把处理逻辑放在JSP中了。

JSTL中的另外一项主要内容是“数据访问标记”，这种标记能够从视图中访问数据库。在一些简单的应用中可以不必多做考虑，直接使用JSTL的数据访问机制。但是，对于各种规模的企业应用，使用这种数据访问标记都会违背“把数据访问逻辑和视图分离开”的设计原则。对于这些应用，一种更好的选择是使用数据访问对象，这种对象封装了数据访问代码，提供了一种更加便于维护、便于重用的解决方案。

最后，JSTL和JSP2.0中还集成了一种强大的表达式语言（EL），这也是对在JSP中使用Java scriptlet代码的一种替代方案。以下例子体现了EL的强大功能。对于在JSP中访问JavaBean助手

的代码：

```
<% bean.getVar1(); %> 或 <% var1 = bean.getVar1(); var1.getVar2(); %>
```

可以用以下形式，通过EL简化：

```
 ${bean.var1} 或 ${bean.var1.var2}
```

除了采用JSTL之外，也可以开发自己的助手（使用定制标记或标记文件），这也会带来一些益处，因为这种方案可以提供一种更高层次的抽象，从而更为清晰地传达代码的设计意图。

但是，无论采用JSTL还是定制标记，都应该避免简单地把助手当成scriptlet使用。

关于JSTL和EL，请参见后文的设计手记“实现 vs 意图”、标记文件助手策略、以及<http://java.sun.com/jsp>网站的资料。

如果并没有使用前端控制器和/或拦截过滤器作为集中控制机制，那么视图本身就是请求处理的最初接触点（后文“服务到工作者”模式中对此有详细描述）。对于一些情况，这是一种不错的选择。242 但是对于一些情况，我们则推荐另外的做法。

对于比较简单的情况（也就是说，只有很少、甚至根本没有操作管理或业务服务调用的情况），这种做法的效果不错。举例来说，请求可能只要求静态内容、或者要求一些已经存在于内存中的信息（所以只需要把这些信息按照显示要求加以格式化即可）。这种类型的用例会进行“视图创建”操作，但是无须作“视图准备”。

而对于更加动态的请求（这些请求既需要视图准备、也需要视图创建），就应该使用以控制器为中心的解决方案，比如“服务到工作者”模式。

但也应该记住：在同一个应用系统中，通常会结合使用这两种方式，用于处理合适的用例。

在本模式中，可以采用多种策略实现视图组件。首选策略是基于模板的视图策略，也就是使用某种基于模板的实现机制（比如JSP）作为视图组件。另一种主要的策略是基于控制器的视图策略，它使用某种控制组件（比如servlet）作为视图或视图管理组件。在后面的“视图转化”设计手记中，讨论了控制器结合XSLT实现视图创建的做法。

## 设计手记：视图处理

正如上文所述，请求处理可以划分为两个主要方面：请求操作处理和视图处理。在视图处理过程中，表现层要执行多项操作。这些操作又可以再分为两大类：**视图准备**和**视图创建**。

- **视图准备**——包括模型对象的复制、以及确定和/或生成一个合适的视图组件。视图准备是请求处理的一项核心操作，其中，模型对象完成内容的获取，并且保存从数据源中获取的内容状态。通常使用传输对象获取数据，数据会封装在业务对象中。
- **视图创建**——包括把模型的状态转化成视图组件中的动态内容。

## 结构

243 图6-34是视图助手的类图。

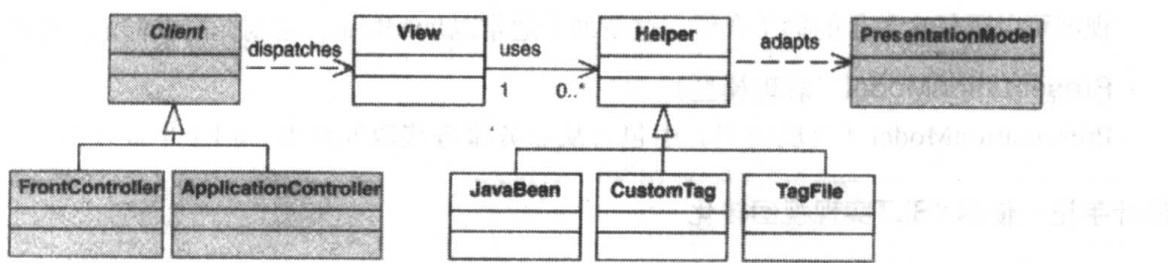


图6-34 视图助手的类图

## 参与者和责任

图6-35是视图助手的序列图。通常由一个集中的前端控制器在客户端和视图之间进行协调，这样在本序列图中，控制器也就起到了Client（客户端）的作用。但是，也有一些情况（尤其是在响应主要或完全是静态的情况下），系统中不会使用集中的前端控制器，客户端直接调用视图。这种使用模式就称为“分配器视图”。

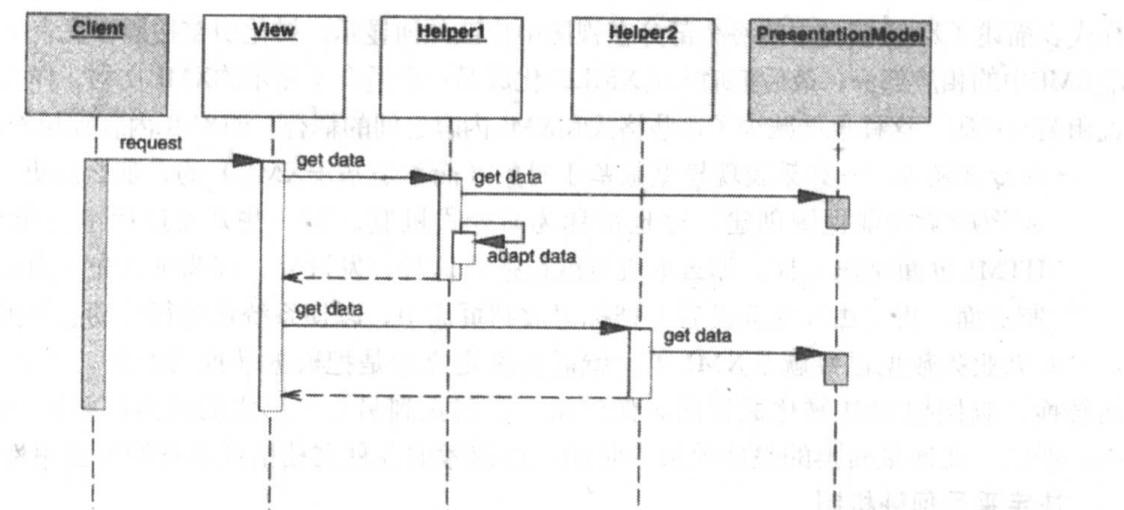


图6-35 视图助手序列图

正如前面的类图中所示，也可能根本没有助手与视图关联。这就是一种最简单的情况了，此时要么页面是完全静态的，要么页面中包括了非常少量的、内联的scriptlet代码。

### Client（客户端）

客户端分派到视图。

### View（视图）

视图把信息表现、显示给客户端。在动态页面中显示的信息是助手类从表现模型中获取并转化而得的。

### Helper1, Helper2（助手1、助手2）

助手封装了视图生成和视图格式化的处理逻辑。助手通常要为视图转换PresentationModel（表现模型），或者提供了对PresentationModel中的原始数据的访问。

视图可以跟任意数量的助手合作，这些助手通常以JavaBean、定制标记或标记文件的形式实现。

### PresentationModel（表现模型）

PresentationModel（表现模型）中包含从业务服务获取的数据，用于生成视图。

## 设计手记：使用XSLT实现视图转化

为了从表现模型中生成视图，有两种主要做法。如果表现模型是基于XML的，那么通常使用函数式机制（functional approach）；如果表现模型是基于对象的，那么往往使用命令式机制（imperative approach）。

- 函数式机制——如果表现模型是XML，那么视图生成的常用机制就是：利用扩展样式表语言（XSL）执行视图转化操作。XSL语言由3大组件构成，其中两种都用于访问和转化XML。这两种组件就是XSL 转化（XSLT） 和XML路径语言（XPath），后者是XSLT [XSL]使用的一种强大的表达式语言。

这种机制和常见的JSP机制有什么不同呢？这个机制中，XML和样式表一起送进转化引擎，其中样式表描述了XML模型中的每个部分在视图中应该如何显示。转化引擎按照样式表中的格式规则匹配XML中的相应部分，最后把输入的XML转化成另一个适合于显示的XML文档。样式表封装了与格式相关的代码，这样也就减少了这些格式和XML内容之间的耦合，而XML内容则起着模型的作用。

- 命令式机制——如果表现模型是基于对象（而不是基于XML）的，那么就更适合使用JSP和视图助手实现视图创建，这也被称为命令式机制。有一些开发过程中会用到很多流行的HTML页面制作工具，那么本机制就很适合这种开发过程，因为可以先（由web设计者）创建页面，再（由软件开发者）把标记放到页面中，以便系统在运行时动态替换这些标记。

如果业务数据已经就是XML了，就需要决定究竟是把数据转换成对象，还是不采用任何中间转换，直接把XML转化成视图。任何从一种形式到另一种形式的转换都要耗费时间、降低效率，所以，在衡量转换的整体效果的时候，应该考虑各种文档格式本身的可重用性。

### 决定采用何种机制

#### XML/XSLT函数式机制的优点

如果业务层已经在XML格式上花了很大功夫了，那么就大可以把它当作基础，构建此后的各种机制。在这种环境下，业务层可以对web层的客户端返回XML，这样，在视图创建中使用函数式机制（并配合转化引擎和样式表）也就更为适宜。

XML/XSLT机制的优点是它的语言中立性和平台中立性。XML模型可以在一种平台上由代码生成，而接下来的视图转化则可以在另一种平台上进行。第8章中的“Web Service中转”模式正是在WebService请求处理的场景中处理这个问题的。

#### XML/XSLT函数式机制的相关问题

经常能够听到这样一种说法：使用XML/XSLT机制能够避免把与格式无关的逻辑同格式标记混在一起。其实这个说法并不完全准确。实际上，使用XSL的表达式语言Xpath，把处理逻辑（比如条件选择和流程控制）和格式逻辑混在一起倒是很容易、很常见的做法。这跟JSP中，HTML格式标记和Java scriptlet代码混在一起的情况差不多。

应该谨记：在大多数采用OO方法的软件开发机构中，能精通XSLT（目前这是最典型的

XML转化机制)的人少之又少。选用XSLT时应该考虑到的另外一点是,在这种环境下测试和调试都非常困难,对那些习惯了OO开发周期的人就尤其如此。如果准备大量使用XSLT,首先就要保证有足够的对此有经验的人手。

### 执行视图转化

视图转化既可以在客户端执行,也可以在表现层执行。通常还是在表现层完成,这样也能减少对客户端暴露安全信息和数据访问信息的风险。视图转化往往作为应用控制器中的转化处理器策略的一部分执行。

视图转化既可以解释执行,也可以编译执行,既可以在客户端执行,也可以在表现层执行。[246]

- 转化可以在运行时结合样式表解释执行,如图6-36中所示。

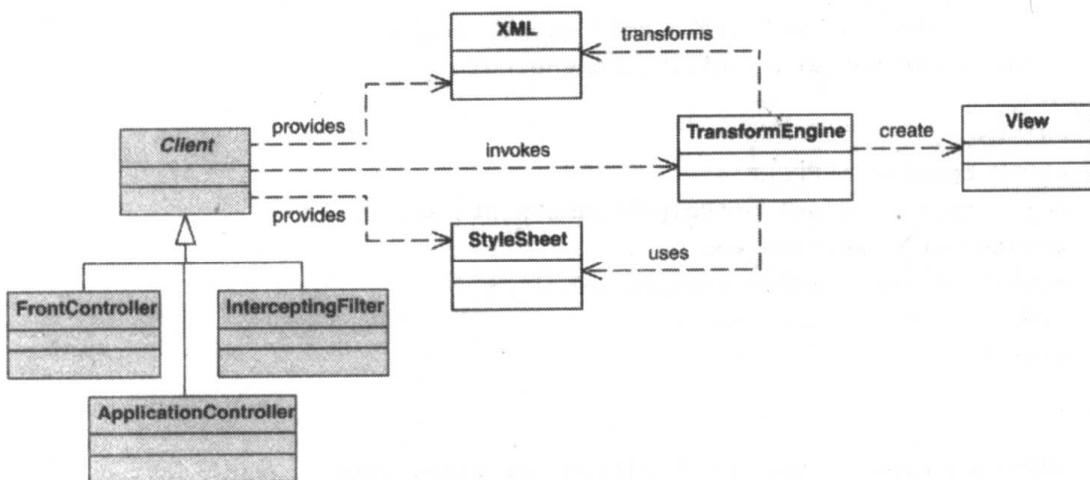


图6-36 使用样式表

- 另一种办法是把样式表编译成转化引擎,这种引擎称为translet,专门接收、处理XML,如图6-37所示。Java XML处理API (Java API for XML Processing, JAXP) 1.2版专门提供了编译执行转化的标准机制,其中就使用了一种XSLT编译器 (XSLTC) [JAXP]。如果在一种应用场景中包含大量重复的转化操作,编译执行样式表就能够极大地提升性能。另外,编译后的转化引擎也要比解释执行的转化引擎更为紧凑、轻便,因为编译器在编译过程中只需要生成特定样式表中显式引用到的XSLT功能代码。

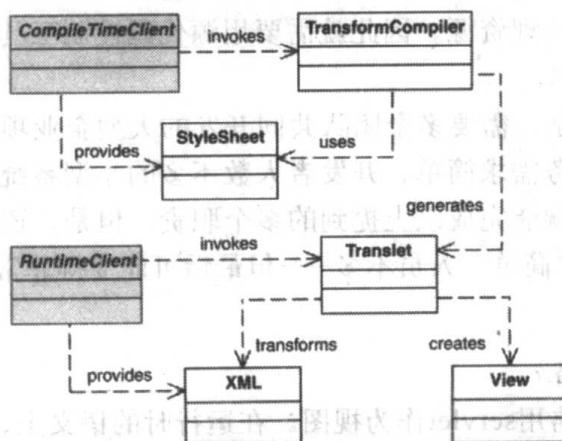


图6-37 使用Translet

## 策略

### 基于模板的视图策略

使用基于模板的视图（比如JSP）充当视图组件，这就称为“**基于模板的视图策略**”。与基于控制器的视图策略相比，虽然在运行时的语义上这两种策略是等效的，但我们推荐使用本策略。基于模板的视图策略有助于明确团队分工，更好地划分程序员与网页设计者之间的角色。

例6.52是本策略的代码示例。它是一个源文件（welcome.jsp）中的片断，servlet首先把JavaBean WelcomeHelper放到请求的作用域（request scope）中，然后再把请求分派到该JSP页面。

#### 例6.52 基于模板的视图策略

```

1  <jsp:useBean id="welcomeHelper" scope="request"
2   class="corepatterns.util>WelcomeHelper" />
3
4  <HTML>
5  <BODY bgcolor="#FFFFFF">
6  <c:if test = "${welcomeHelper.nameExists == true}">
7  <center><H3> Welcome <b>
8  <c:out value='${welcomeHelper.name}' />
9  </b><br><br> </H3></center>
10 </c:if>
11
12
13 <H4><center>Glad you are visiting our site!</center></H4>
14
15 </BODY>
16 </HTML>
```

另一种实现办法是后面介绍的**基于控制器的视图策略**，它通常会把HTML标记直接放在Java servlet代码中。

像这样把Java代码和HTML标记混在一起，就会让整个项目中的各个成员职责分工也不够明确，而且也增加了不同开发组的不同成员依赖于相同资源的可能性。如果一个工作者要处理的视图模板中包含其不熟悉的代码或标记，那么就更可能会因为某个随手操作把问题引入整个系统。多人需要共享同样的物理资源，因此就需要用源代码控制工具进行管理，这也增加了管理的复杂度，降低了工作效率。

对于那些系统需求复杂、需要多个团队共同开发的大型企业项目来说，就更容易产生这些问题。相比之下，对于业务需求简单、开发者人数不多的小型系统，这些问题发生的概率要小一些，因为同一个开发者就能完成以上提到的多个职责。但是，还请记住这一点：项目在开始的时候往往都很小——需求简单、人员不多——但最后可能变得非常复杂，到那时以上建议就不无裨益了。

### 基于控制器的视图策略

**基于控制器的视图策略**用servlet作为视图。在运行时的语义上，这个策略和使用JSP的基于模板的视图策略（通常推荐使用该策略）是等效的，因为JSP在运行时会被转译成servlet。但在

开发时，基于控制器的视图策略往往会给软件开发团队和网页设计团队带来更大的麻烦，因为它要把网页标记直接写在Java代码里。把HTML标记包括在Java代码中，会让视图模板变得难于更新和修改。

还有一种类似的做法，但在本策略中没有描述，那就是用控制器管理视图创建。该做法让控制器调用转化引擎（transformation engine）、样式表（style sheet）和XML模型，完成视图转化。

**例6.53** 示范了基于控制器的视图管理器策略。

### 例6.53 基于控制器的视图管理器策略

```

1  public class EmployeeListServlet extends HttpServlet {
2      public void init(ServletConfig config) throws ServletException {
3          super.init(config);
4      }
5
6      public void destroy() { }
7
8      /**处理HTTP <code>GET</code> 和
9       * <code>POST</code> 方法的请求。
10      * @param request servlet请求
11      * @param response servlet响应
12      */
13     protected void processRequest(
14         HttpServletRequest request, HttpServletResponse response)
15         throws ServletException, java.io.IOException {
16         String title = "Controller-based View Strategy";
17         try {
18             response.setContentType("text/html");
19             java.io.PrintWriter out = response.getWriter();
20             out.println("<html><title>" + title + "</title>");
21             out.println("<body>");
22             out.println("<h2><center>Employees List</h2>");
23             EmployeeDelegate delegate = new EmployeeDelegate();
24             /** ApplicationResources (应用资源) 提供了一套简单的API,
25              * 用来获取常数和其他预定义的值**/
26             Iterator employees = delegate.getEmployees(
27                 ApplicationResources.getInstance().getAllDepartments());
28             out.println("<table border=2>");
29             out.println("<tr><th>First Name</th> " +
30                     "<th>Last Name</th> " +
31                     "<th>Designation</th><th>Id</th></tr>.");
32             while (employees.hasNext()) {
33                 out.println("<tr>"); .
34                 EmployeeTO emp = (EmployeeTO)employees.next();
35                 out.println("<td> " + emp.getFirstName() + "</td> ");
36                 out.println("<td> " + emp.getLastName() + "</td> ");
37                 out.println("<td> " + emp.getDesignation() + "</td> ");
38                 out.println("<td> " + emp.getId() + "</td> ");

```

249

```

39         out.println("</tr>");
40     }
41     out.println("</table>");
42     out.println("<br><br>");
43     out.println("</body>");
44     out.println("</html>");
45     out.close();
46   }
47   catch (Exception e) {
48     LogManager.logMessage("Handle this exception",
49     e.getMessage() );
50   }
51 }
52
53 /** 处理HTTP <code>GET</code> 方法。
54 * @param request servlet 请求
55 * @param response servlet 响应
56 */
57 protected void doGet(HttpServletRequest request,
58   HttpServletResponse response)
59   throws ServletException, java.io.IOException {
60   processRequest(request, response);
61 }
62
63 /** 处理HTTP <code>POST</code> 方法。
64 * @param request servlet 请求
65 * @param response servlet 响应
66 */
67 protected void doPost(HttpServletRequest request,
68   HttpServletResponse response)
69   throws ServletException, java.io.IOException {
70   processRequest(request, response);
71 }
72
73 /** 返回servlet的简要说明。*/
74 public String getServletInfo() {
75   return "Example of Servlet View. " + "JSP View is preferable.";
76 }
77
78 /** 分配器方法*/
79 protected void dispatch(HttpServletRequest request,
80   HttpServletResponse response, String page)
81   throws javax.servlet.ServletException, java.io.IOException {
82   RequestDispatcher dispatcher =
83     getServletContext().getRequestDispatcher(page);
84   dispatcher.forward(request, response);
85 }
86 }

```

## JavaBean助手策略

本策略用JavaBean实现助手。使用助手，就能够把应用系统中的业务逻辑、处理逻辑和格式逻辑从视图中提取出来，放到助手组件中，从而把视图与这些逻辑明确地区分开来。

与使用定制标记助手策略相比，使用JavaBean助手策略的整体工作量要少一些，因为JavaBean很容易编写，也很容易与JSP环境集成。而且，就算是开发新手也能理解JavaBean。另外，采用这个策略，需要生成的文件全在JavaBean里了，所以这种策略也更便于维护和管理。例6.54是本策略的一个例子。

### 例6.54 JavaBean助手策略

```

1  <jsp:useBean id="welcomeHelper" scope="request"
2    class="corepatterns.util.WelcomeHelper" />
3
4  <HTML>
5  <BODY bgcolor="#FFFFFF">
6  <c:if test = "${welcomeHelper.nameExists == true}">
7
8  <center><H3> Welcome <b>
9  <c:out value='${welcomeHelper.name}' />
10 </b><br><br> </H3></center>
11 </c:if>
12
13 <H4><center>Glad you are visiting our site!</center></H4>
14
15 </BODY>
16 </HTML>
```

251

本策略的一个问题是，要高效地从视图中抽取出实现细节，也是一件困难的事。

## 定制标记助手策略

用定制标记实现助手，让定制标记来负责调整模型，以便用于视图。

与JavaBean助手策略相比，使用定制标记助手策略的整体工作量要更大，因为开发定制标记是相当复杂的。编写定制标记而不是JavaBean，不仅会增加开发过程的复杂度，而且还会让整个系统的集成工作、全部代码的管理工作更加复杂。而定制标记的优势在于它的灵活性和可扩展性。

使用这个策略需要修改多个文件才能配置整个环境，这些文件包括定制标记本身、标记库描述符(tag library descriptor)，另外还有配置文件。例6.55是使用这个策略的JSP视图页面的片段。

### 例6.55 定制标记助手策略

```

1  <%@ taglib uri="/web-INF/corepatternstaglibrary.tld"
2  prefix="corepatterns" %>
3  <html>
4  <head><title>Employee List</title></head>
5  <body>
6
7  <div align="center">
```

```

8   <h3> List of employees in <corepatterns:department attribute="id"/>
9     department - Using Custom Tag Helper Strategy. </h3>
[252]
10  <table border="1" >
11    <tr>
12      <th> First Name </th>
13      <th> Last Name </th>
14      <th> Designation </th>
15      <th> Employee Id </th>
16      <th> Tax Deductibles </th>
17      <th> Performance Remarks </th>
18      <th> Yearly Salary</th>
19    </tr>
20    <corepatterns:employeelist id="employeelist_key">
21    <tr>
22      <td><corepatterns:employee attribute="FirstName" /> </td>
23      <td><corepatterns:employee attribute="LastName" /></td>
24      <td><corepatterns:employee attribute="Designation" /> </td>
25      <td><corepatterns:employee attribute="Id" /></td>
26      <td><corepatterns:employee attribute="NoOfDeductibles" /></td>
27      <td><corepatterns:employee attribute="PerformanceRemarks" /></td>
28      <td><corepatterns:employee attribute="YearlySalary" /></td>
29      <td>
30    </tr>
31    </corepatterns:employeelist>
32  </table>
33  </div>
34  </body>
35  </html>

```

### 设计手记：实现 vs 意图

把视图与分隔控制代码、业务逻辑和格式处理逻辑分隔开来，这是一种重要的设计目标；从2.0版本开始，JSP提供了一些很强大的功能，能够直接实现以上目标。为此，JSP技术规范中包括了两种重要内容：标记文件和表达式语言（EL）。（见<http://java.sun.com/jsp>。）

这些功能支持模式的多种实现策略。使用助手把程序代码和处理逻辑从模板标记中抽取出来，就能够更好地把控制代码、模型代码与视图分隔开，这是该做法的一种主要益处。而为了评判分隔的效果，有一种很好的办法：看看最后留在模板中的调用助手的代码，究竟是暴露了设计意图，还是暴露了实现细节。

比如说，假设有一些scriptlet代码用来执行条件检查，然后再遍历一个集合中的结果，生成HTML表。可以把这些代码抽取到一个定制标记助手中，让它执行相同的处理，但是把实现细节隐藏在标记内部了。

但也有另外一种作法，而且还不算罕见，那就是使用语义上等效的标记来替换scriptlet代码，比如用<IF>标记、<ForEach>标记替换条件判断和迭代处理。这样的用法中，虽然最后的页面比较干净（因为scriptlet代码和标记不会混在一起了），但是替换后的标记其实在语义上是和代码

等效的，所以暴露了同样的实现细节。

实际上，很容易用一种编程语言替换另外一种，同时却暴露出同样的实现细节。把助手当成scriptlet使用是一种不高明的做法，不过如果只在限制场合使用，也不为大过。为了避免这个问题，还有一种更强大的处理策略，那就是使用标记文件助手。

### 标记文件助手策略

用标记取代Java scriptlet是一种优化，但如果在JSP视图中，大部分的、甚至所有的标记都在语义上是与它们所取代的Java scriptlet等效的，那么实际上只是用一种编程语言取代另一种罢了，仍然会暴露代码的实现细节。标记文件（tag files）就能够避免在页面标记中暴露过多的实现细节，从而解决这个问题；它的作用是把可重用的代码抽取到一个处理器中，而不必编写定制标记。另外，与定制标记相比，标记文件还更容易配置和部署，因为需要维护的文件数量要比前者少。

但是，还是应该让开发团队中的非技术人员（网页设计人员）明白，虽然创建标记文件很容易，但是要理解其中关键的抽象机制、理解实现这些抽象机制的最佳办法，这可并不容易。所以，即使采用了标记文件，仍然应该在开发团队中保持清晰的角色/职责划分，不要把非技术人员的工作和技术人员的工作混为一谈。

举个例子，例6.56是JSP页面中的一段，其中使用了JSTL标记。与scriptlet代码相比，这已经是一种优化了，但是它还是暴露出HTML表的创建细节。

#### 例6.56 使用JSTL，example.jsp

```

1  <%-- example.jsp --%>
2  <%-- 注意：这个例子使用了JSTL (URL ref) --%>
3
4  <table>
5  <c:forEach var="product" items="${products}" varStatus="status">
6  <tr>
7  <td><c:out value="${status.count}" /></td>
8  <td><c:out value="${product.name}" /></td>
9  </tr>
10 </c:forEach>
11 </table>
12

```

254

JSP2.0以上版本支持标记文件和表达式语言（Expression Language，EL），二者能够优化JSP中的代码。使用它们，能够更好地体现代码的设计意图、隐藏实现细节。如例6.57所示。

#### 例6.57 使用标记文件和EL，example.jsp

```

1  <%-- example.jsp --%>
2  <%-- Using a tag file with the EL--%>
3
4  <%@ taglib prefix="tag" tagdir="/web-INF/tags/" %>
5
6  <tag:buildtable products="${products}" />
7

```

其中引用的标记文件buildtable.tag放在/web-INF/tags目录下，web容器负责自动生成一个标记处理器，用于给出预期的输出内容。buildtable.tag文件的代码如例6.58所示。

#### 例6.58 标记文件，buildtable.tag

```

1 <%-- /web-INF/tags/buildtable.tag --%>
2 <%@ attribute name="products" type="java.util.Collection" %>
3 <%@ taglib prefix="c" uri="http://java.sun.com/jstl/core" %>
4 <table>
5   <c:forEach var="product" items="${products}" varStatus="status">
6     <tr>
7       <td><c:out value="${status.count}" /></td>
8       <td><c:out value="${product.name}" /></td>
9     </tr>
10   </c:forEach>
11 </table>
```

#### “业务代表用作助手”策略

助手组件经常要调用远程的业务层服务。使用业务代表，能够隐藏这种调用的实现细节。助手调用了业务服务，无须了解服务的物理实现细节和分布细节。

255

可以把助手组件和业务代表这两个概念结合起来，因为它们都是通过POJO实现的；而且这样结合之后，就不用给系统新添一个层次了。但是，助手和业务代表之间也有一个主要区别：助手组件是由负责表现层的开发者编写的，而业务代表通常则是由负责业务层服务的开发者编写的。业务代表可能也形成了一个框架中的一部分。这就意味着，如果采用这个策略，就需要（编写助手/业务代表的）表现层开发者和（实现服务的）业务层开发者双方都参与其中。

在团队中，如果以上两种开发角色存在重叠<sup>①</sup>，那么就可以考虑采用“业务代表用作助手”的策略。

#### 设计手记：助手

JavaBean助手可以用作命令对象（见前文的前端控制器中的命令加控制器策略），也可以用于保存中间模型<sup>②</sup>（见后文的传输对象），还可以用来调整该模型，形成视图显示（见前文的应用控制器）。

和JavaBean助手一样，定制标记助手也能充任以上所有角色（除了命令对象之外）。但与JavaBean不同的是，定制标记助手更适合用于视图中的流程控制和迭代。这样使用定制标记助手，能够封装逻辑，以防止在JSP中直接用scriptlet代码编写逻辑。

标记文件（需要JSP 2.0以上版本支持）也可以像定制标记一样达到以上目标，但是与编写定制标记相比，编程工作量要小一些（参见上文的标记文件助手策略）。这里我们把定制标记助手和标记文件助手统称为标记助手。

<sup>①</sup> 也就是说，同样一些人既完成业务服务的实现，也编写表现层的视图助手，那么他们就能够负责改动业务代表的代码。

<sup>②</sup> 中间模型：intermediate model，所谓“中间”，是指这种模型既非原始数据（raw data），也非最后经过格式化的显示内容。

另一个推荐使用标记助手的领域，就是可以用标记实现原始数据的调整和格式化，以供显示之用。比如，可以用标记来遍历一个集合中的结果，把这些结果格式化为HTML表格，然后再把该表格置入JSP视图中，而这些操作都无须使用Java scriptlet代码。

考虑这样一个例子：web客户端请求系统中的账户信息，如图6-38所示。

256

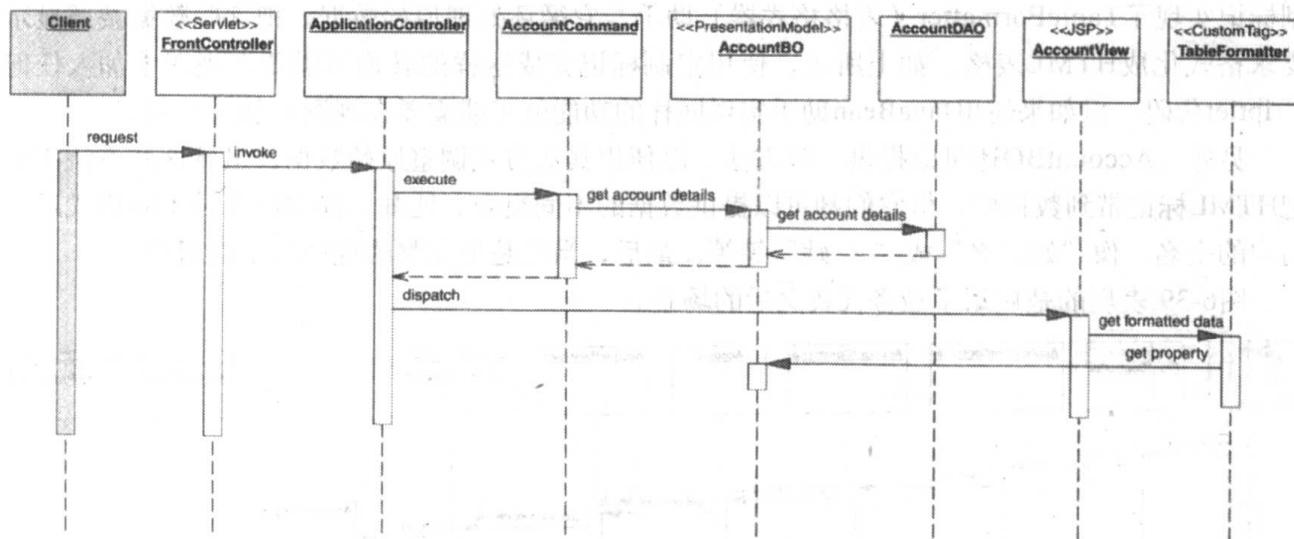


图6-38 在非远程的业务层访问中使用助手

257

图中有两个JavaBean助手：AccountCommand对象和AccountBO对象。TableFormatter 对象则是定制标记助手。

前端控制器处理请求，把请求委派给应用控制器。应用控制器使用命令处理器策略，获取匹配的命令对象。在本例中，命令对象处理的是对账户信息的请求。应用控制器调用命令对象，后者则调用业务对象，获取账户信息。业务对象调用持久化机制，从数据库中获取这些信息。获取数据之后，应用控制器就使用视图处理器策略，解析出匹配的视图，并且把请求分派给这个视图。

那么，助手和命令对象是怎样与业务对象交互的呢？业务对象又是怎样与数据库交互的呢？让我们考察两种情况，一种比较简单，另一种则复杂一些。

### 简单情况

在这种比较简单的情况下，假设这样一种部署场景：表现层和业务层都处于同一个web服务器或应用服务器中，共享同一个进程空间。而且，假设未来的项目会考虑使用EJB，但是本应用系统则没有使用它。假设目前业务对象使用数据访问对象来访问数据库。数据访问对象封装了对数据库的JDBC/SQL查询，这样就对业务对象隐藏了数据访问的实现细节。这个场景如上面图6-38的序列图所示。

### 更复杂的情况

到了某个阶段，整个部署场景会发生变化，现在，业务层对表现层来说处于远程。而且还引入了EJB，部分原因是利用会话门面模式连接各个分布部署的层次。另外，也引入了服务定位器和业务代表，用以对表现层的客户端隐藏EJB寻址、调用和异常处理的实现细节。业务代表还能够通过缓存来提高系统性能。这种业务代表通常由业务服务的开发者编写，它能够降低各

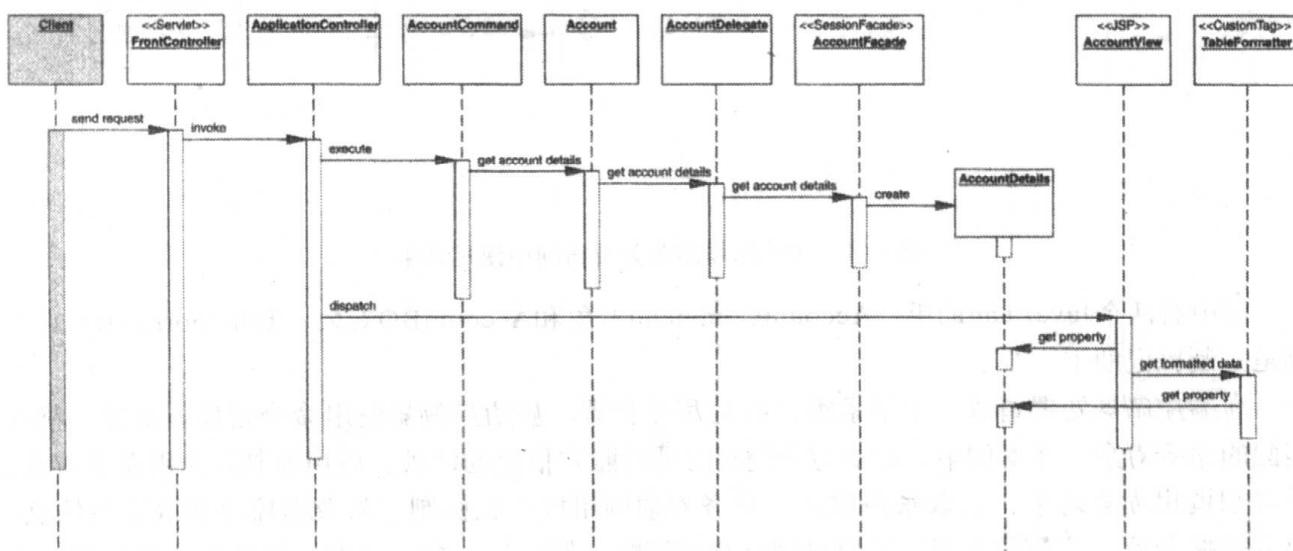
层次之间的耦合。

FrontController（前端控制器）把请求分派给匹配的视图（本例中是AccountView.jsp）。视图从AccountBO（账户业务对象）中获取了数据（这是原始数据和格式化后的一种混合），这里AccountBO既起到了业务对象的作用，也还是一种助手，起着表现模型的作用。我们用定制标记实现了TableFormatter（表格格式器）助手，它循环处理原始数据，把这些数据按照显示要求格式化成HTML表格。如上所述，使用定制标记完成这样的转换不需要在视图中加入任何scriptlet代码，但如果使用JavaBean助手实现同样的功能就可能需要在视图中加入代码了。

另外，AccountBO还可以提供一些方法，以便用其他方式调整原始数据。虽然这些方法不会把HTML标记带到数据中，但它们却可以提供数据的不同组合。比如，能够以多种不同格式返回用户的全名，像“姓，名”或“名姓”等等。最后，当然是把完整的视图显示给用户。

258

图6-39 表现的是应用了业务代表之后的场景。



259

图6-39 在远程的业务层访问中使用助手

## 效果

- 明确了应用系统各部分的分隔，增进了可重用性和可维护性

把HTML从处理逻辑（比如控制逻辑、业务逻辑、数据访问逻辑和页面格式逻辑）中分离出来，能够使系统各部分的分隔更加明确。系统被分为逻辑上互不相关的若干部分，其中每个部分都能够被封装在高内聚的、可重用的组件里。控制逻辑可以转移到前端控制器和命令助手里，业务逻辑则放在业务对象里。数据访问对象负责数据访问逻辑，页面格式逻辑则转移到标记助手里。这样一来，系统中可能出现多种形式的助手，包括JavaBean、定制标记或标记文件（JSP 2.0以上版本），这些助手封装了各种处理逻辑，避免它们以scriptlet的形式分散在视图中。

比如，应该尽量减少JSP页面中的Java程序逻辑，而且也要尽量减少程序代码中的HTML标记内容。如果没有尽量减少上面两种现象的出现，往往就会遇到困难和窘境，在大型项目中尤其如此。

如果把程序逻辑从JSP中抽取出来，封装到助手中，就能够增进逻辑的可重用性，减少视图中的代码重复，简化维护工作。

- 明确了开发团队中的角色分工

开发团队中的多个角色如果处理同一种资源，那么他们各自的任务就可能产生依赖和冲突；使用助手类把处理逻辑从视图中分离出来，能够降低这种情况的发生。例如，如果把处理逻辑放进了视图里，那么就要有一个软件开发者来负责HTML标记中的代码。而一个负责web开发的设计者可能要修改页面的布局，甚至还会修改到一些与Java代码混在一起的组件。那么，上面两种人对彼此的任务细节都并不熟悉，这也增大了后期修改给系统引入bug的可能性。

- 简化了测试

既然处理逻辑都被分离出来、放到了助手组件里，那么测试独立的代码段落就变得更容易了。与测试这样封装成单独的类的代码相比，测试JSP中的代码段落要困难得多。

260

- 用助手简单地替换scriptlet

把处理逻辑从页面中抽取出来说，这么做的一个重要原因就是要减少在页面中直接暴露的实现细节。但应该谨记的是，单单在JSP中使用JavaBean或者定制标记，这算不上是万灵药。如果只是使用一些通用的助手，把JSP中内嵌的Java代码替换成对相应助手的引用，那其实也一样会产生暴露实现细节的问题——代码不应该过多暴露实现细节，而应该尽可能体现设计意图。

举个例子：可以使用一种“条件助手”，比如用一个定制标记实现“if”语句的条件选择逻辑。如果大量使用这种助手标记，那可能只起到了替换相应的scriptlet代码的作用。结果，替换之后的文件看起来仍然是页面中包含程序逻辑的样子。简单地用助手替换scriptlet是一种不佳实践，虽然很多人为了好歹用上“视图助手”模式，经常会这么做。

## 相关模式

- 前端控制器

前端控制器通常把请求委派给应用控制器，由后者执行操作管理和视图管理。

- 应用控制器

应用控制器负责视图准备和视图创建，并把控制权委派给视图和助手类。

- 视图转化

实现视图创建的另一种办法是采用视图转化。

- 业务代表

业务代表能够减少助手对象和它所调用的远程业务服务之间的耦合。

261

## 复合视图

### 问题

需要结合使用多个模块化的、原子化的<sup>Θ</sup>视图组件，创建一个复合的整体视图，同时还要独

<sup>Θ</sup> 原子化的：atomic，是指一个成分不能再向下细分。

独立地管理页面的内容和布局。<sup>①</sup>

开发、维护动态视图，是一种很困难的工作，因为在多个视图之间，总有一些内容和布局都是共通的。如果这些内容、布局与其他部分都混杂在一起，视图就很难维护和扩展了。如果共通代码在多个视图之中多次复制，那么系统的重用性和模块化程度都要受到影响。

## 约束

- 需要在多个视图中重用一些共通的子视图，比如页眉（header）、页脚（footer）和表格（table）；在不同的页面中，这些子视图可能出现在页面布局的不同位置上。
- 子视图中的内容可能常常需要改动，或者这些子视图需要实现访问控制，比如说按照用户角色限制对子视图的访问。
- 需要避免在多个视图中直接嵌入和复制子视图，否则，对布局的修改就很难管理、维护。

## 解决方案

**使用由多个原子化的子视图构成的复合视图。整个模板中的每个子视图都可以动态地纳入整体，页面布局的管理可以独立于页面的内容。**

复合视图的基本机制，在于对模板中动态或静态的模块化元素的包含和替换。复合视图能够促进模块化设计，从而增进对视图中各种原子化部件的重用。各个显示组件可以按照多种方式组合使用，复合视图正是组合这些显示组件生成页面的一种恰当方法。

一个示例场景如下：在门户网站中，可能在一个页面里包括很多独立的子视图，比如新闻喂送<sup>②</sup>、天气信息以及股票价格等。而页面布局的管理和修改则独立于子视图的内容。

本模式的另一个优点在于，Web设计师可以做出网站布局的原型，对于模板中的每个独立区域，则可以暂时先填入静态内容。随着网站开发过程的推进，再用实际的内容替换原先用于占位的静态内容。这种做法能够提高系统的模块化程度和可重用性，同时也增进了可维护性。

图6-40是Sun公司的Java主页，java.sun.com。可以看出4个主要区域：导航（Navigation）、查询（Search）、专题报道（Feature Story）和头条新闻（Headline）。每个子视图组件的内容都可能来自不同的数据源，但是它们能够无缝地合成一个复合页面。

当然，这个模式也不无缺陷。此模式会产生一定的运行时负荷，这也是提高灵活性带来的代价。另外，如果使用特别复杂的页面布局机制，也会带来管理和开发方面的问题，因为需要维护更多的文件，技术实现的方式也更为间接、难于理解。

① 独立地管理页面的布局和内容：指对页面布局的修改不会影响到内容，同样，对子视图内容的修改也不会影响到整个布局。

② 新闻喂送：news feed，网站内容管理的一种常用技术。由发布新闻的服务器提供新闻文档，而门户服务器自动获取这种文档，并包含到门户页面中。比如，<http://www.theserverside.com/rss/theserverside-rss2.xml>就是著名Java社区TSS的news feed。也可以编写服务器端程序，自动把以上文档的内容包含到自己的网页中。

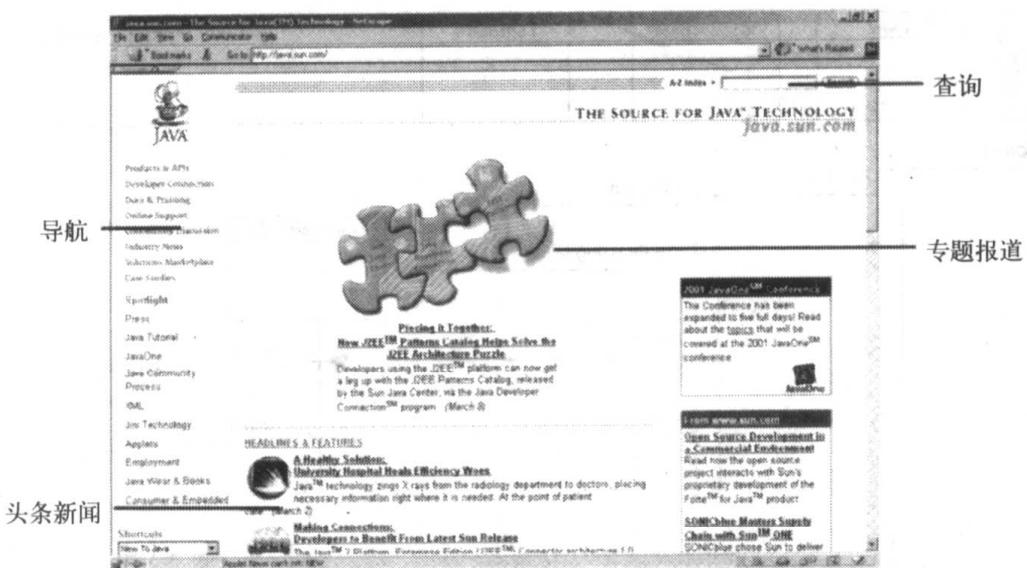


图6-40 模块化页面，包括导航、查询、专题报道和头条新闻4个区域

## 结构

图6-41 是复合视图模式的类图。

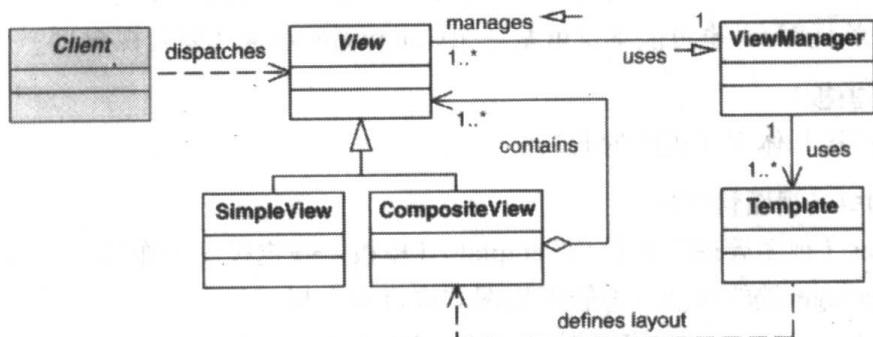


图6-41 复合视图模式的类图

## 参与者和责任

图6-42是复合视图模式的序列图。

**Client (客户端)**

客户端的请求分派到一个视图。

**View (视图)**

视图也就是最终的显示。

**SimpleView (简单视图)**

SimpleView (简单视图) 是整个复合视图的一个原子化的局部。也称为视图片段或子视图。

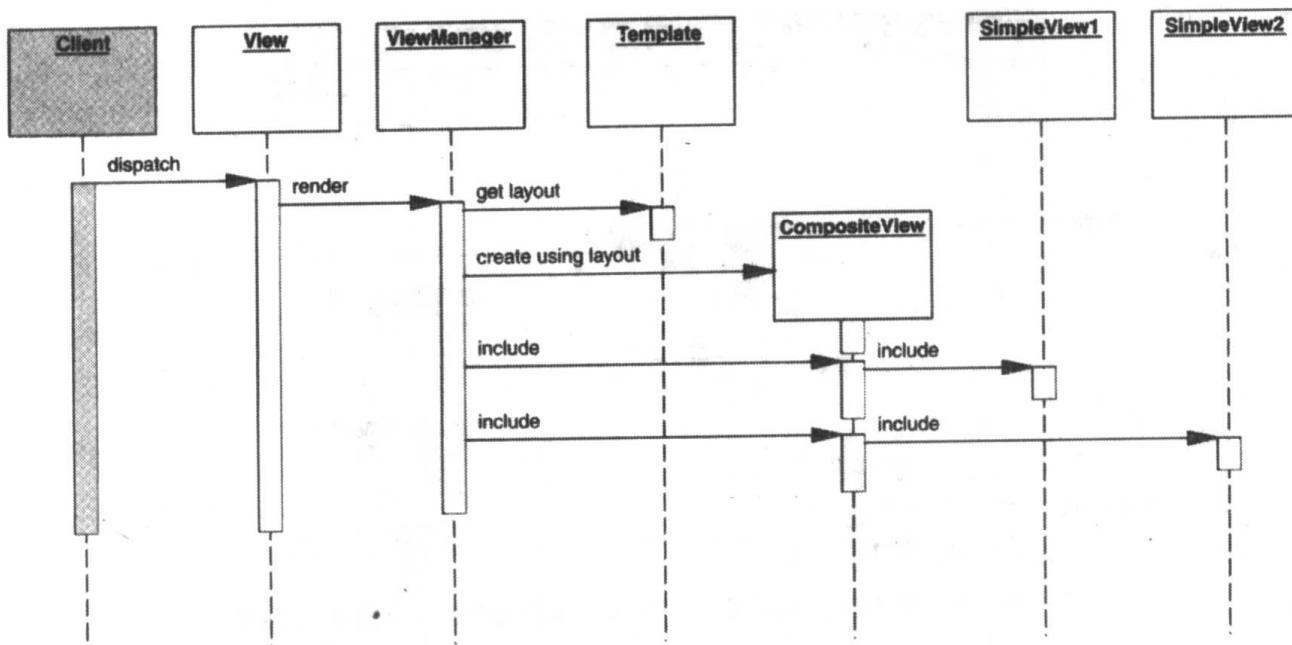


图6-42 复合视图模式的序列图

### CompositeView（复合视图）

CompositeView（复合视图）由多个View（视图）构成。其中每一个View既可以是SimpleView（简单视图），也可能本身就是一个CompositeView（复合视图）。

### Template（模板）

264

Template（模板）体现了视图的布局。

### ViewManager（视图管理器）

ViewManager（视图管理器）使用Template（模板）规定视图的布局，并负责填充匹配的内容，这样ViewManager就能够独立地管理视图的内容和布局。

一个简单的ViewManager可以使用标准JSP“包含”标记（<jsp:include>），把多个SimpleView（简单视图）包含到模板中。

另一种办法是，可以采用一个更复杂的ViewManager，使用POJO或定制标记助手，以一种更全面、更健壮的方式实现内容管理和布局管理。使用POJO或者定制标记，能便于实现“有条件包含”。比如，只有当用户符合特定角色或者系统满足特定条件的时候，才包含某些子视图。而且，使用助手组件完成视图管理，也能够对页面的整体结构进行更复杂的控制，这有助于创建可重用的页面布局。

## 策略

### JavaBean视图管理策略

使用JavaBean和JSP标准标记来实现内容管理和布局管理，如例6.59所示。视图委派到一个JavaBean，后者用定制的逻辑控制视图布局和构成。与JSP的标准包含功能相比，这个作法的功能更为强大，因为它能够基于用户角色或安全策略来确定页面布局。虽然这个策略在语义上与

定制标记视图管理策略是等效的，但是它不如后者来得漂亮，因为本策略还是要在页面里引入一定的scriptlet代码。

虽然定制标记视图管理策略是我们推荐的首选策略，但是与其相比，JavaBean视图管理策略要求的总体工作量倒要少一些，因为JavaBean很容易编写，也便于集成到JSP环境中。而且，即使是开发新手也能理解JavaBean。从可维护性的角度考虑，这个策略也更为简单，因为需要管理、配置的文件就是所有的JavaBean而已。

例6.59示范了JavaBean视图管理策略。

### 例6.59 JavaBean视图管理策略

```

1  <%@ taglib prefix="c" uri="http://java.sun.com/jstl/core" %>
2  <%@ taglib uri="/web-INF/corej2eetaglibrary.tld" prefix="cjp" %>
3
4  <jsp:useBean id="contentFeeder"
5    class="corepatterns.compositeview.javabean.ContentFeeder"
6    scope="request" />
7
8  <table valign="top" cellpadding="30%" width="100%">
9
10   <cjp:personalizer interest='global'>
11     <tr>
12       <td><B><c:out value="${contentFeeder.worldNews}" /></B></td>
13     </tr>
14   </cjp:personalizer>
15
16   <cjp:personalizer interest='technology'>
17     <tr>
18       <td>
19         <U><c:out value="${contentFeeder.technologyNews}" /></U>
20       </td>
21     </tr>
22   </cjp:personalizer>
23
24   <cjp:personalizer interest='weird'>
25     <tr>
26       <td><I><c:out value="${contentFeeder.weirdNews}" /></I></td>
27     </tr>
28   </cjp:personalizer>
29
30   <cjp:personalizer interest='astronomy'>
31     <tr>
32       <td><c:out value="${contentFeeder.astronomyNews}" /></td>
33     </tr>
34   </cjp:personalizer>
35 </table>
```

266

在引入了JSTL [JSTL] 和表达式语言 (EL) [JSTL, JSP]之后，视图中的流程控制可以由

JSTL标记和EL完成，而JavaBean则主要用于存放状态。在使用这种策略的时候，要点在于不能过度地把JavaBean当成Scriptlet使用。

### 标准标记视图管理策略

使用标准JSP标记（比如<jsp:include>标记）实现视图管理。很容易实现这种使用标准标记管理视图布局和构成的策略。但是，本策略缺乏定制标记视图管理策略的强大和灵活，因为每个页面的布局仍然是写在该页面里的。这也就意味着，虽然这个策略允许页面内容动态改变，但是如果要改变整个网站的布局风格，则必须修改大量JSP文件。

本策略如例6.60所示。

#### 例6.60 标准标记视图管理策略

```

1 <html>
2 <body>
3 <jsp:include
4   page="/jsp/CompositeView/javabean/banner.seg" flush="true"/>
5 <table width="100%">
6   <tr align="left" valign="middle">
7     <td width="20%">
8       <jsp:include page="/jsp/CompositeView/javabean/ProfilePane.jsp"
9         flush="true"/>
10    </td>
11    <td width="70%" align="center">
12      <jsp:include page="/jsp/CompositeView/javabean/mainpanel.jsp"
13        flush="true"/>
14    </td>
15  </tr>
16 </table>
17 <jsp:include page="/jsp/CompositeView/javabean/footer.seg"
18   flush="true"/>
19 </body>
20 </html>
```

267

如果用标准标记创建了一个复合视图，那么视图中既可以包含静态内容，比如HTML文件，也可以包含动态内容，比如JSP。另外，内容既可以在翻译时包含，也可以在运行时包含。

如果内容在翻译时包含，那么在JSP被重新编译成servlet时，才会检查各部分内容的更动，这时子视图的内容更动才会反映到页面中。也就是说，每次JSP重新编译的时候，整个页面才会被重新组合、生成。

例6.61就是这样一个JSP的片段，它使用标准JSP包含指令<%@ include %>，在编译时包含内容，生成复合页面。

而运行时的内容包含则意味着，如果下属的子视图发生了改变，到下一次客户端访问复合页面时就会体现到该页面中。这种机制更为动态，可以通过标准JSP包含标记<jsp:include>实现，如例6.62所示。当然，这种视图生成会带来一定的运行时负载，但是以此为代价，换来的则是立竿见影地体现内容改动的灵活性。

### 例6.61 用翻译时内容包含实现复合视图

```

1  <table border=1 valign="top" cellpadding="2%" width="100%">
2      <tr>
3          <td><%@ include file="news/worldnews.html" %> </td>
4      </tr>
5      <tr>
6          <td><%@ include file="news/countrynews.html" %> </td>
7      </tr>
8      <tr>
9          <td><%@ include file="news/customnews.html" %> </td>
10     </tr>
11     <tr>
12         <td><%@ include file="news/astronomy.html" %> </td>
13     </tr>
14 </table>
```

268

### 例6.62 用运行时内容包含实现复合视图

```

1  <table border=1 valign="top" cellpadding="2%" width="100%">
2      <tr>
3          <td><jsp:include page="news/worldnews.jsp" flush="true"/> </td>
4      </tr>
5      <tr>
6          <td><jsp:include page="news/countrynews.jsp" flush="true"/></td>
7      </tr>
8      <tr>
9          <td><jsp:include page="news/customnews.jsp" flush="true"/> </td>
10     </tr>
11     <tr>
12         <td><jsp:include page="news/astronomy.jsp" flush="true"/> </td>
13     </tr>
14 </table>
```

### 定制标记视图管理策略

使用定制标记实现视图管理器，让该管理器执行内容管理和布局管理。这是我们推荐使用的实现复合视图的首选策略。与JSP include标记相比，用定制标记处理页面内容和布局更为强大、更为灵活，但是要付出的工作量也更大。使用这种策略能很容易地基于用户角色或安全策略等来管理页面布局。

与其他视图管理策略相比，本策略需要的工作量更大，因为开发定制标记比简单应用JavaBean或标准标记要复杂得多。还仅仅是开发过程复杂，要想集成、管理完成后的标记也有很高的难度。使用这种策略需要创建大量文件，包括标记本身、标记库描述符、配置文件等，还要利用这些文件配置环境。但是，如果使用某个现成的第三方标记库，那么应用这个策略的复杂度就会低得多。

例6.63的JSP片段是一种应用了第三方库的可能实现方法。对于实现的细节可以参照后面的示例代码。区域（region）标记起到了视图管理器的作用，它通过模板规定了页面布局，并且把模板中的各个段落的逻辑名称（比如banner）和实际内容（比如banner.jsp）匹配起来。

269

### 例6.63 使用第三方类库的JSP片段

```

1   <region:render template='/jsp/CompositeView/templates/portal.jsp'>
2
3       <region:put section='banner'
4           content='/jsp/CompositeView/templates/banner.jsp' />
5
6       <region:put section='controlpanel'
7           content='/jsp/CompositeView/templates/ProfilePane.jsp' />
8
9       <region:put section='mainpanel'
10      content='/jsp/CompositeView/templates/mainpanel.jsp' />
11
12      <region:put section='footer'
13          content='/jsp/CompositeView/templates/footer.jsp' />
14
15  </region:render>
```

### 转化器视图管理策略

用XSL转化器实现视图管理。这种策略通常与定制标记视图管理策略结合使用，其中定制标记负责实现并委派相应的组件。事实上，页面中就可以直接加入一个或多个视图转化，从而创建整体的复合视图。以下的JSP片段体现了如何在JSP中使用定制标记，结合样式表（stylesheet）和转化器（xsl transformer）把模型（model）转换为视图。

```
<xsl:transform model="portfolioHelper"
    stylesheet="/transform/styles/generalPortfolio.xsl"/>
```

当然，也可以把这个转化抽象到一个助手中，这样就能够更清晰地表达转化的设计意图，而不是在视图中直接暴露实现细节。如下所示：

```
<finance:renderPortfolio type="General" model="portfolioHelper" />
```

### 早绑定资源策略

这其实是翻译时内容包含的一个别名，这种做法已经在上面的标准标记视图管理策略中介绍了，代码如例6.62所示。这个策略适用于相对静态的模板的维护和更新，如果视图的页眉和页脚不会频繁改动，则推荐使用本策略。

### 晚绑定资源策略

这其实是运行时内容包含的一个别名，这种做法已经在上面的标准标记视图管理策略中介绍了，代码如例6.63所示。如果复合页面的子视图内容会频繁改动，那就适合采用本策略。

---

### 设计手记：动态资源子视图

如果在运行时包含的子视图是一种动态资源（比如一个JSP），那么这个子视图本身可能也是一个复合视图，其中还包括更多的运行时内容。采用这种设计之前，应该在这种嵌套复合结构带来的灵活性和它的运行时负载之间做出权衡，并且还应该结合项目本身的实际需求考虑这个决定。

## 效果

- 增进了模块化和重用

本模式能促进模块化设计。使用本模式，就能在多个视图中重用模板中的一个原子化的部分（比如一个股票价格表），并且把这些可重用的部分结合其他多种信息一起显示出来。采用本模式，就可以把这个股票价格表抽取到独立的模块里，在必要时再包含该模块即可。这种动态的页面布局和构成，能够减少代码复制，鼓励重用，并能够提高可维护性。

- 添加基于角色或安全策略的访问控制

复合视图能够基于一些运行时的判断（比如用户角色或系统的安全策略），按条件包含模板中的不同视图部分。

- 提高了可维护性

当模板不是直接包括在视图标记里的时候，管理模板各部分的改动就容易得多。如果把模板内容与视图本身分隔开，那么就能够独立于模板的布局，对模板的各个模块化的部分加以修改。而且，只要复合视图的实现策略得当，客户端能够马上看到这些修改。当然，对页面布局的修改也容易得多，因为页面布局控制被完全集中化了。

- 降低了可维护性

把大量原子化的显示内容聚合到一起，形成一个单一的视图，这可能会引入显示错误，因为各个子视图都是整个页面的片段。这种局限性可能导致维护上的问题。比如，如果一个JSP生成了一个HTML页面，该主页面还包括3个子视图，每个子视图都有HTML的开始、结束标记（也就是<HTML> 和 </HTML>标记），那么构成的复合页面就是不合法的。所以，使用这个模式时应该注意子视图一定不能是完整的视图。为了创建合法的复合视图，对标记的使用就要特别严格、留意，这也就可能成为一个维护上的问题。

- 降低了性能

生成包含大量子视图的显示页面可能会降低性能。运行时包含子视图会导致客户端每次请求该视图时都发生延时。如果应用环境对响应时间有特定的要求，那么这种性能下降（虽然通常都微乎其微）就可能是没法接受的。一种替代方案是采用在翻译时包含子视图，不过这种做法稍微限制了系统的灵活性：只有页面被重新编译时，子视图的改动才会反映出来。

## 示例代码

271

实现复合视图的策略中，最强大的就是定制标记视图管理策略了。事实上，目前有几种现成的定制标记库，支持用这种方法实现复合视图。这些库提供了视图管理功能，能够把布局控制和内容分隔开来，而且支持模块化的、可插拔式的子视图。

本例采用了David Geary编写的一种模板库，该库在Geary的《Advanced JavaServer Pages（高级JSP）》[Geary]一书中有详细介绍。

这个模板库包括3种基本组件：段落（section）、区域（region）和模板（template）。

- “段落”是一种可重用的组件，用于生成HTML或JSP。
- “区域”定义多个段落，从而确定页面内容。

- “模板”控制生成后的页面中各个区域和段落的布局。

例6.64是定义并生成区域的一种做法。

#### 例6.64 一个区域和其中的各个段落

```

1 <region:render template='portal.jsp'>
2   <region:put section='banner'      content='banner.jsp' />
3   <region:put section='controlpanel' content='ProfilePane.jsp' />
4   <region:put section='mainpanel'    content='mainpanel.jsp' />
5   <region:put section='footer'      content='footer.jsp' />
6 </region:render>
```

在区域中，把段落的逻辑名称（比如banner）和该部分实际内容（比如banner.jsp）匹配

[272]

起来。

区域以及下属段落的布局由模板定义，每个区域都有相关的模板。在本例中，这个模板称为portal.jsp，其定义如例6.65。

#### 例6.65 模板定义

```

1 <region:render section='banner' />
2 <table width="100%">
3   <tr align="left" valign="middle">
4     <td width="20%">
5       <!-- menu region -->
6       <region:render section='controlpanel' />
7     </td>
8     <td width="70%" align="center">
9       <!-- contents -->
10      <region:render section='mainpanel' />
11    </td>
12  </tr>
13 </table>
14 </region:render>
```

如果一个网站中有大量页面，但都使用一种一致的布局，那么就可以用一个JSP实现模板定义，代码类似于例6.65所示；另外还会用很多JSP来定义下属的区域和段落，代码类似于例6.66。所谓“段落”，就是一种JSP页面片段，起着子视图的作用；多个片段按照模板中定义的布局共同构成整个的复合视图。例6.66是一个称为banner.jsp的段落。

#### 例6.66 段落子视图——banner.jsp

```

1 <table width="100%" bgcolor="#C0C0C0">
2 <tr align="left" valign="middle">
3   <td width="100%">
4
5   <TABLE ALIGN="left" BORDER=1 WIDTH="100%">
6     <TR ALIGN="left" VALIGN="middle">
7       <TD>Logo</TD>
8       <TD><center>Sun Java Center</TD>
9     </TR>
```

```

10    </TABLE>
11
12    </td>
13  </tr>
14 </table>
```

在复合视图模式的实现策略中，定制标记视图管理策略是最强大的一种（当然这还有待争论），因为，正如转化器视图管理策略一样，它封装了页面布局。把页面布局封装起来，就更易于修改和重用。但还不仅仅是封装页面布局而已。既然定制标记视图管理策略使用了JSP定制标记，就可以很容易地加入一些诱人的功能，比如：

- 基于用户的角色选择包含某些内容。
- 区域中嵌套另一个区域。
- 把一些区域定义为继承另一些区域。

例6.67至例6.69示范了Regions标记库<sup>Θ</sup>对以上功能的支持。

#### 例6.67 基于用户角色显示内容

```

1  <region:render template='portal.jsp'>
2      <region:put section='banner' content='banner.jsp' role='customer' />
3  </region:render>
```

在例6.68中，门户页面会向客户显示广告。

#### 例6.68 嵌套区域

```

1  <region:define id='BANNER' template='twoColumns.jsp'>
2      <region:put section='logo' content='logo.jsp' />
3
4      <region:put section='advertisement' content='advertisement.jsp' />
5  </region:define>
6  ...
7  <region:define id="PORTAL" template='portal.jsp'>
8      <region:put section='banner' content='BANNER' />
9  ...
10     <region:put section='footer' content='footer.jsp' />
11 </region:render>
```

在例6.68中定义了两个区域：门户和广告栏。PORTAL（门户）区域中包括一个banner（广告）段落，用于放置BANNER（广告）区域。（在Regions标记库中，可以用<region:define>标记来定义区域。）

#### 例6.69 扩展区域

```

1  <region:define id='PORTAL' template='portal.jsp'>
2      <region:put section='banner' content='BANNER' />
3  ...
4      <region:put section='footer' content='footer.jsp' />
```

<sup>Θ</sup> Regions标记库：Regions就是“区域”的意思。这也就是上面提到的文献[Geary]中介绍的那种标记库。当前该库可以从以下地址下载：<http://www.javaworld.com/javaworld/jw-12-2001/jw-1228-jsptemplate.html>。

```

5   </region:render>
6
7   <region:define id='PORTAL_WITH_SPECIAL_FOOTER' region='PORTAL'>
8       <region:put section='footer' content='specialfooter.jsp'/>
9   </region:define>

```

例6.69同样定义了两个区域：一个是普通门户（portal），另一个则是有特殊页脚的门户。普通门户（PORTAL）区域中定义的所有区域，都可以由PORTAL\_WITH\_SPECIAL\_FOOTER（有特殊页脚的区域）来继承，但是后者没有继承前者的页脚，而是用一种特殊的页脚覆盖了它。

复合视图是一种模块化的、灵活的可扩展的办法，可以用来为J2EE应用构建JSP视图。

## 相关模式

- 视图助手

复合视图能够起到视图助手模式中“视图（View）”的作用。

- 复合模式[GoF]

复合模式[GoF]描述了一个多部分组成的复合对象内部的部分-整体的层次关系；复合视图就是基于这个模式的。

275

## 服务到工作者

### 问题

需要在把控制权交给视图之前就完成核心的请求处理过程和业务逻辑调用。

在有些用例中，需要在生成视图之前调用业务服务和数据服务。对于任何特定的请求来说，在请求处理的“视图准备”阶段究竟应该完成多少工作，这是一个重要的设计考虑；以下问题有助于就此作出决定。

- 控制逻辑有多复杂？
- 响应内容有多动态？
- 业务逻辑和业务模型有多复杂？

### 约束

- 在请求处理过程中，想要执行某种业务逻辑，以便获取数据，生成动态响应。
- 在系统中，视图的选择可能要依赖于业务服务调用的结果。
- 可能必须要在应用系统中使用一个框架或类库。

### 解决方案

使用服务到工作者模式集中控制权管理和请求处理，在把控制权交给视图之前获取表现模型。视图则根据获得的表现模型生成一个动态响应。

服务到工作者模式是由其他几个模式组合构成的。它分别使用了前端控制器、应用控制器和视图助手，以实现集中控制、请求处理和视图创建的功能。

服务到工作者和分配器视图是两种最常见的表现层应用场景。服务到工作者模式是一种以控制器为中心的架构，而分配器视图模式则是一种以视图为中心的架构。与服务到工作者模式相反的是，分配器视图模式要把业务处理推迟到视图处理时才进行。

以下是服务到工作者模式的一个典型用例。客户端向控制器发送一个请求，其中包括一个参数，描述了需要完成的操作：

```
http://some.server.com/Controller?action=login
```

前端控制器处理请求中那些与网络协议相关的元素。然后，前端控制器把请求委派给应用控制器，同时也传给应用控制器一个新创建的Context对象。应用控制器则起着命令处理器的作用，它负责把操作的逻辑名称“login”映射到匹配的命令（比如LoginCmd）上，并实际调用该命令。

### 设计手记：前端控制器

通常，前端控制器在把控制权交给应用控制器之前，要创建一个Context对象，从而用一种与网络协议无关的形式封装请求的状态。

在命令的执行中会调用业务服务，根据这次调用返回的结果，应用控制器选择匹配的视图，并把控制权分派到该视图——这也就起到了视图处理器的作用。

最后，一个应用系统究竟是使用服务到工作者模式，还是使用分配器视图模式，这也与对特定框架或类库的使用有关，因为那些框架或类库可能会隐含地支持某些特定的模式组合。比如，如果你的开发团队必须使用Struts框架[Struts]，那么你的应用系统中就已经包括了对服务到工作者模式的强大支持，但在一些情况下也可以使用分配器视图。

## 结构

图6-43是使用命令的服务到工作者模式的类图。

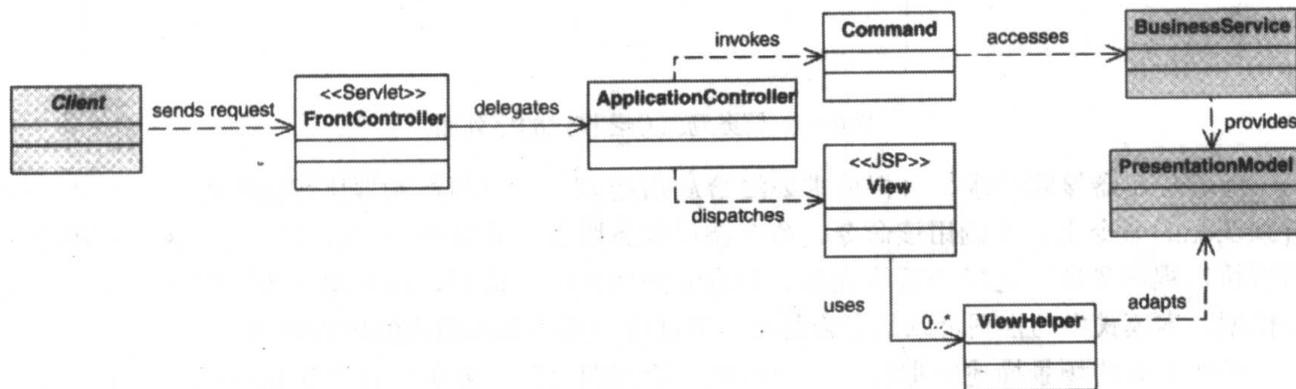


图6-43 使用命令的服务到工作者模式的类图

图6-44是使用助手的服务到工作者模式的类图。

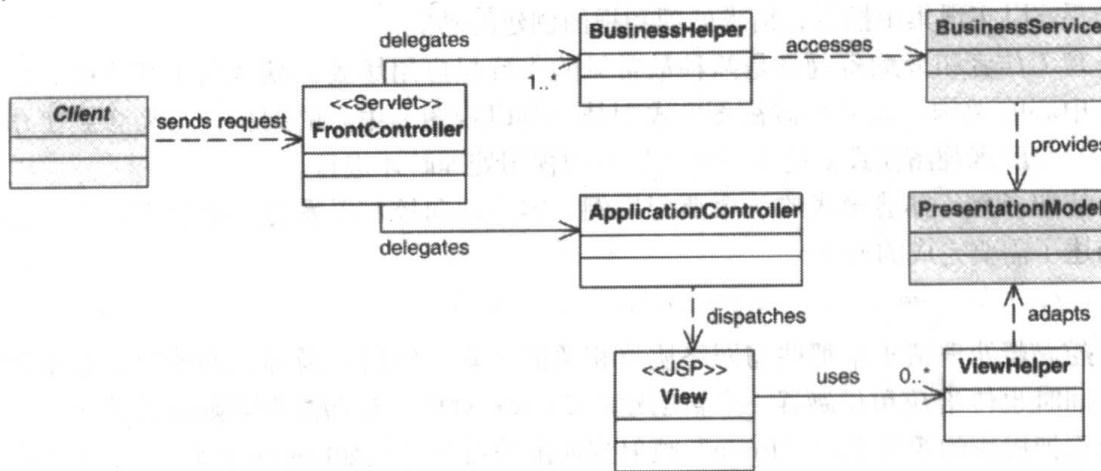


图6-44 使用助手的服务到工作者模式的类图

## 参与者和责任

图6-45是服务到工作者模式的序列图。

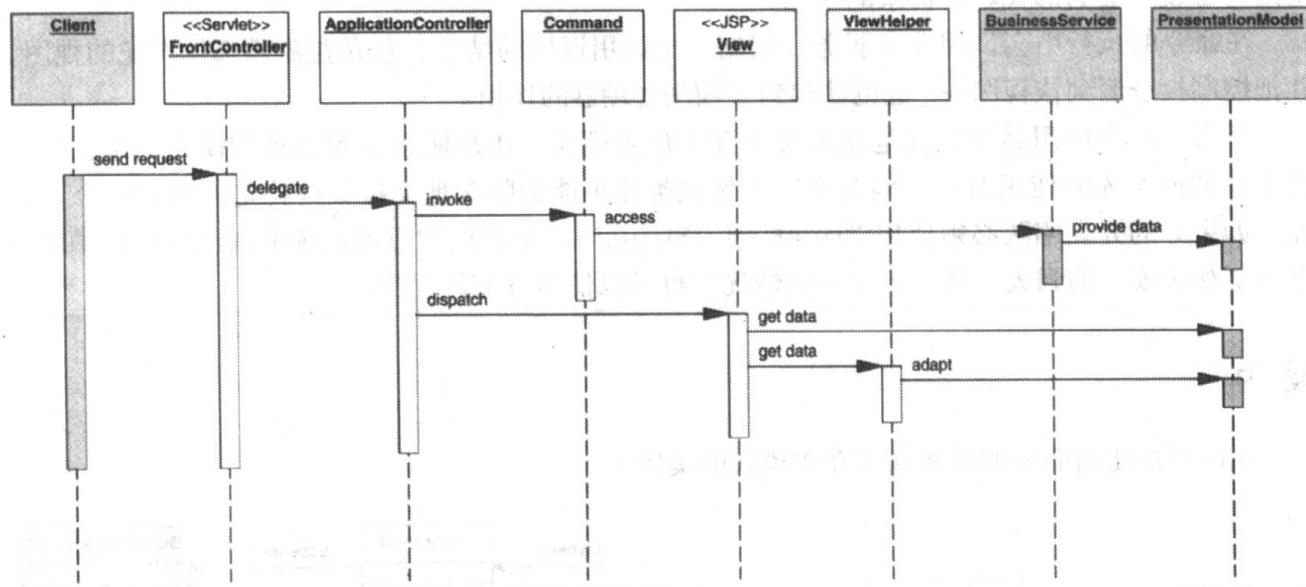


图6-45 服务到工作者模式的序列图

前端控制器接收请求，再把请求委派给应用控制器，应用控制器执行操作管理，把请求解析到匹配的命令上，并调用该命令。命令调用业务服务，并返回一个表现模型。这时应用控制器再执行视图管理，选择合适的视图，并分派到该视图。最后，视图助手为视图调整并转化表现模型。本模式中的视图可以是复合视图，不过这一点并非本模式的核心内容。

正如上面的场景描述的那样，在控制权交给视图之前，业务处理逻辑就已经完成了。与此相反，分配器视图则要在控制权交给视图之后再执行业务逻辑。所以，在这一点上，这两种模式恰好形成了两个极端，而对于同一个应用系统，通常会分别使用这两种模式，以满足不同用

例的需要。

### FrontController（前端控制器）

[279]

FrontController（前端控制器）首先处理请求，然后再把请求委派给应用控制器，完成操作管理和视图管理。

### ApplicationController（应用控制器）

本模式中， ApplicationController（应用控制器）负责操作管理和视图管理。它负责根据请求内容，选择合适的操作和视图来完成请求处理。在一些比较简单的用例中，以上功能可以放在前端控制器里，采用“控制器中的分配器”策略。

### View（视图）

视图向客户端表现和显示数据。由视图助手来调整和转化PresentationModel（表现模型），从而形成最后的显示内容。视图可以是复合视图。

### BusinessHelper（业务助手）、ViewHelper（视图助手）

助手类负责帮助视图或控制器来进行特定的处理。在处理请求时， BusinessHelper（业务助手）帮助控制器初始化业务处理过程；而 ViewHelper（视图助手）则负责获取并调整 PresentationModel（表现模型），从而生成视图。

### PresentationModel（表现模型）

PresentationModel（表现模型）中包含从业务服务获取的数据，用于生成视图。

### BusinessService（业务服务）

BusinessService（业务服务）封装了业务逻辑和业务状态。如果是远程业务服务，可以由通过业务代表访问。

## 策略

以下是与服务到工作者模式相关的策略：

- “前端控制器”中的“*Servlet前端策略*”。
- “前端控制器”中的“*JSP前端策略*”。
- “视图助手”中的“*基于模板的视图策略*”。
- “视图助手”中的“*基于控制器的视图策略*”。
- “视图助手”中的“*JavaBean助手策略*”。
- “视图助手”中的“*定制标记助手策略*”。
- “前端控制器”中“*控制器中的分配器*”。

[280]

## 效果

- 集中了控制，增进了系统的模块化程度、可重用性和可维护性

把控制逻辑和请求处理逻辑集中起来，这有助于提高系统的模块化程度和可重用性。一些共通的请求处理代码可以重用，这就减少了把逻辑放在视图中会导致的代码重复。减少了重复，

也就提高了可维护性，因为如果需要修改，那么只要改一处就可以了。

#### • 明确了各角色之间的分工

把控制逻辑和请求处理逻辑集中起来，就能把这些逻辑从用于生成视图的代码中分离出来，这样也就能使开发团队中各人的职责分工更为明确。软件开发者可以集中精力维护程序逻辑，而页面制作者可以集中关注视图。

## 示例代码

例6.70到例6.74的示例代码是服务到工作者模式的一个实现，其中使用了一个控制器servlet、一个应用控制器（该应用控制器又使用了一个命令处理器）和一个用于分派视图的视图处理器。这个实现中包括了Servlet前端策略、“命令加控制器”策略、基于模板的视图策略和JavaBean助手策略。

本例中的视图是一个非常简单的复合视图。后面的图6-50给出了显示结果的屏幕截图。

例6.70给出了一个控制器servlet，它把请求委派给一个应用控制器。应用控制器找到匹配于请求操作的命令对象，用这个命令对象处理请求。这种结合使用前端控制器和应用控制器（作为命令处理器）的做法，也称为“命令加控制器”策略。通过调用工厂来获得命令对象——该工厂返回的命令对象都具有通用的接口，这个接口的代码如例6.71所示。本例使用了LogManager（日志管理器）来记录日志信息。

在后面的屏幕截取图6-50和图6-51中，我们能看到页面下面打印的日志消息，这也就达到了示例目的。

### 例6.70 使用“命令加控制器”策略的控制器Servlet

```

281 1  public class Controller extends HttpServlet {
2      /** 处理HTTP <code>GET</code> 和
3          * <code>POST</code> 方法的请求。
4          * @param request servlet请求
5          * @param response servlet响应
6          */
7      protected void processRequest(HttpServletRequest request,
8          HttpServletResponse response)
9          throws ServletException, java.io.IOException {
10         String next;
11
12         try {
13             // 创建Context对象
14             RequestContext requestContext =
15                 new RequestContext(request, response);
16
17             // 调用请求处理组件，处理
18             // 输入的请求
19             ApplicationController applicationController =
20                 new ApplicationControllerImpl();
21             ResponseContext responseContext =

```

```

22         applicationController.handleRequest(requestContext);
23
24     // 调用响应处理组件，完成
25     // 响应逻辑的处理
26     applicationController.handleResponse(requestContext,
27         responseContext);
28 }
29 catch (Exception e) {
30     LogManager.logMessage(
31         "FrontController(CommandStrategy)",
32         e.getMessage() );
33
34     /** ApplicationResources（应用资源）提供了一套简单的API，
35      * 用以获取常数和其他
36      * 预定义的值*/
37     next = ApplicationResources.getInstance().getErrorHandler(e);
38 }
39
40     dispatch(request, response, next);
41
42 }
43
44 /** 处理HTTP <code>GET</code> 方法。
45  * @param request servlet请求
46  * @param response servlet响应
47  */
48 protected void doGet(HttpServletRequest request,
49     HttpServletResponse response)
50     throws ServletException, java.io.IOException {
51     processRequest(request, response);
52 }
53
54 /** 处理HTTP <code>POST</code> 方法。
55  * @param request servlet请求
56  * @param response servlet响应
57  */
58 protected void doPost(HttpServletRequest request,
59     HttpServletResponse response)
60     throws ServletException, java.io.IOException {
61     processRequest(request, response);
62 }
63
64 /**返回servlet的简要说明。*/
65 public String getServletInfo() {
66     return getSignature();
67 }
68
69 /** 分派器方法*/

```

```

70     protected void dispatch(HttpServletRequest request,
71         HttpServletResponse response, String page)
72         throws javax.servlet.ServletException, java.io.IOException {
73     RequestDispatcher dispatcher =
74         getServletContext().getRequestDispatcher(page);
75     dispatcher.forward(request, response);
76 }
77
78     public void init(ServletConfig config) throws ServletException {
79         super.init(config);
80     }
81
82     public void destroy() { }
83
84     private String getSignature() {
85         return "ServiceToWorker-Controller";
86     }
87 }
88

```

283

### 例6.71 命令接口

```

1   public interface Command {
2       ResponseContext execute(RequestContext requestContext);
3   }

```

每个命令对象[Gof]都实现了通用的命令接口，本例中的命令对象是ViewAccountDetails类的一个实例，该类的代码如例6.72所示。这个命令类的实例使用业务代表调用远程服务。然后再使用视图助手来确定下一步将把控制权分派给哪个视图，并且实际分派到该视图（另请参见例6.73）。

### 例6.72 ViewAccountDetailsCommand（账户明细视图命令）

```

1   public class AccountCommand implements Command {
2       public AccountCommand() { }
3
4       // 账户明细视图操作
5       public ResponseContext execute(RequestContext requestContext) {
6
7           String accountId =
8               requestContext.getStringParameter("AccountId");
9
10      /** 使用业务代表，从应用服务中
11          * 获取数据，并把结果保存在
12          * 一个响应对象里。
13          * 注意：本来可以通过工厂来避免直接
14          * 创建对象，但是为了示例清楚，这里
15          * 还是给出了对象创建的过程**/
16
17      AccountDelegate delegate = new AccountDelegate();
18      AccountTO accountTO;

```

```

19     accountTO= delegate.getAccountProfile(accountId);
20
21     // 可以从业务层组件获得这一信息
22     String logicalViewName = "AccountProfile";
23
24     ResponseContextFactory factory =
25         ResponseContextFactory.getInstance();
26
27     ResponseContext responseContext =
28         factory.createResponseContext(accountTO, logicalViewName);
29
30     return responseContext;
31
32 }
33 }
```

284

### 例6.73 AccountDelegate (账户代表)

```

1  public class AccountDelegate {
2      public AccountProfileTO getAccountProfile(String accountId) {
3          AccountProfileTO accountProfile = null;
4          try {
5              AccountSessionHome home =
6                  (AccountSessionHome) ServiceLocator.getInstance().
7                  getEJBHome("Account", AccountSessionHome.class);
8              AccountSession session = home.create();
9
10             // 调用账户会话门面, 获取账户信息
11             accountProfile = session.getAccountDetails(accountId);
12         }
13         catch(CreateException ex) {
14             // 把session bean创建异常转译成
15             // 应用程序异常
16         }
17         catch(RemoteException ex) {
18             // 把远程异常转译成
19             // 应用程序异常
20         }
21         return accountProfile;
22     }
23 }
```

通过业务代表调用业务服务，获得账户传输对象，这个传输对象对于视图起到了临时数据模型的作用。在本例中，请求被分派给accountdetails.jsp 页面，该页面的代码如例6.74所示。传输对象是通过标准标记<jsp:useBean>引入的，而传输对象的属性则通过标准标记 <jsp:getProperty> 来访问。另外，视图采用了一种非常简单的复合视图策略，在翻译时（translation-time）嵌入了一个子视图trace.jsp，这个子视图负责显示日志信息（在本例中完全出于示范目的）。之所以采用以上这些基本的视图管理策略，只是为了达到示例的效果。

285

**例6.74 视图——accountdetails.jsp**

```
1 <html>
2 <head><title>AccountDetails</title></head>
3 <body>
4
5 <jsp:useBean id="account" scope="request"
6   class="corepatterns.util.AccountTO" />
7
8 <h2>
9   <center> Account Detail for <c:out value='${account.accountHolder}' />
10 </h2> <br><br>
11 <table border=3>
12 <tr>
13 <td>
14 Account Number :
15 </td>
16 <td>
17 <c:out value='${account.number}' />
18 </td>
19 </tr>
20
21 <tr>
22 <td>
23 Account Type:
24 </td>
25 <td>
26 <c:out value='${account.type}' />
27 </td>
28 </tr>
29
30 <tr>
31 <td>
32 Account Balance:
33 </td>
34
35 <td>
36 <c:out value='${account.balance}' />
37 </td>
38 </tr>
39
40 <tr>
41 <td>
42 OverDraft Limit:
43 </td>
44 <td>
45 <c:out value='${account.overdraftLimit}' />
```

```

46  </td>
47  </tr>
48
49  </table>
50
51  <br>
52  <br>
53
54  </center>
55  <%@ include file="/jsp/trace.seg" %>
56  </body>
57  </html>

```

## 相关模式

- 前端控制器、应用控制器和视图助手

服务到工作者模式是一种以控制器为核心的架构，它的重点是前端控制器。前端控制器把请求委派到应用控制器，让后者负责导航和分派，然后再把请求委派给视图和各种助手类。

- 复合视图

本模式中的视图可以是复合视图。

- 业务代表

使用业务代表来隐藏业务服务的远程语义。

- 分配器视图

分配器视图模式是一种以视图为核心的架构。与服务到工作者模式相反，在分配器视图模式中，在控制权交给视图后才进行业务处理。

287

## 分配器视图

### 问题

需要用视图来处理请求、生成响应，同时又要让它来完成数量有限的业务处理。

在一些特定的用例中，在生成视图之前很少需要、甚至完全不需要进行业务处理。比如，当视图是静态的或者是从一个现有的表现模型（presentation model）中生成时就是如此。这时，视图只要求有限的业务服务或数据访问服务。

### 约束

- 有一些静态视图。
- 有一些从现有的表现模型生成的视图。
- 视图独立于任何业务服务响应。

- 只需要完成有限的业务处理。

## 解决方案

**使用分配器视图，把视图本身作为请求的最初访问点。在必须进行业务处理的情况下，如果业务处理的数量有限，那么就由这些视图完成。**

如果生成响应只需要很少的动态内容（通过调用业务服务或数据访问服务）、甚至根本不需要动态内容，那么就可以采用这种办法。以下是分配器视图的两种最常见的用法。

- 1) 响应完全是静态的。

例如：响应是一个静态的HTML页面。

- 2) 响应是动态的，但由一个现有的表现模型生成。

例如：前面的一个请求负责获取用于视图的表现模型。这个中间模型放在某种临时的存储（比如HttpSession）中。后面的一个请求由视图直接处理，该视图简单地使用这个现有的表现模型，生成一个动态响应。

还有另一种思路：如果响应需要使用大型的动态组件，而表现模型需要通过调用业务服务才能获得时，则可以使用服务到工作者模式。

使用分配器视图模式的时候应该特别留意。在很多只需要有限的业务处理的应用系统中，人们会想到使用本模式。例如，当开发者们编写一些“一次性的”测试工具或者报表工具时，如果这些工具的应用场合有限，那么开发者们经常就会使用分配器视图，把数据访问查询直接放在JSP页面里。虽然对于那些用途有限的简单应用，这么做常常能够节省时间，但是一定要注意，这种节省并不是没有代价的，采用这种模式可能明显地降低系统的可维护性和可重用性。

前端控制器可以用作分配器视图的一部分，但是控制器通常不做其他操作，只是把请求转发给视图。这也就是“前端控制器”一节中“控制器中的分配器”策略介绍的内容。客户端发出请求中可以包括一个参数，用以说明要进行的操作。例如：

```
http://some.server.com/Controller?action=showaccount.jsp
```

在这种情况下，控制器的惟一责任就是把请求分派给视图showaccount.jsp。

在某种特别有限的形式之下，应用控制器也可以用作分配器视图的一部分，完成基本的视图管理。如果客户端的请求中包括对操作的一个逻辑参照，那么应用控制器可以负责把该逻辑名称解析到具体的视图上。比如，考虑以下请求：

```
http://some.server.com/Controller?action=showaccount
```

应用控制器起到了视图处理器的作用，把逻辑名称“showaccount”解析到一个实际的视图（可能就叫showaccount.jsp）上。然后应用控制器就把请求分派给该视图。

## 结构

图6-46是分配器视图的类图。

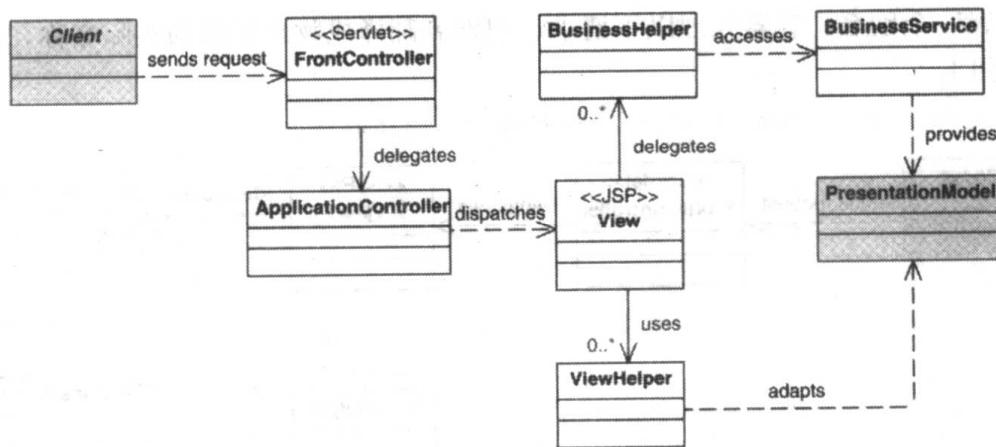


图6-46 分配器视图的类图

289

## 参与者和责任

图6-47 是分配器视图的序列图。

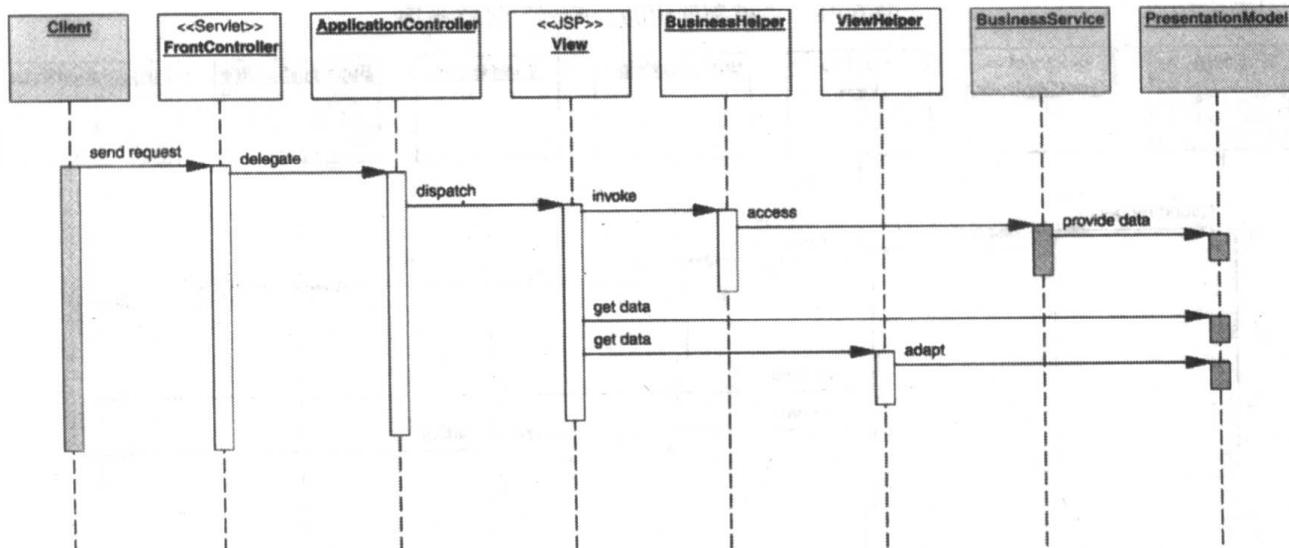


图6-47 分配器视图的序列图

290

注意：与服务到工作者模式中不同，如果调用业务服务，那么这种调用会一直推迟到控制权交给视图后才被执行。

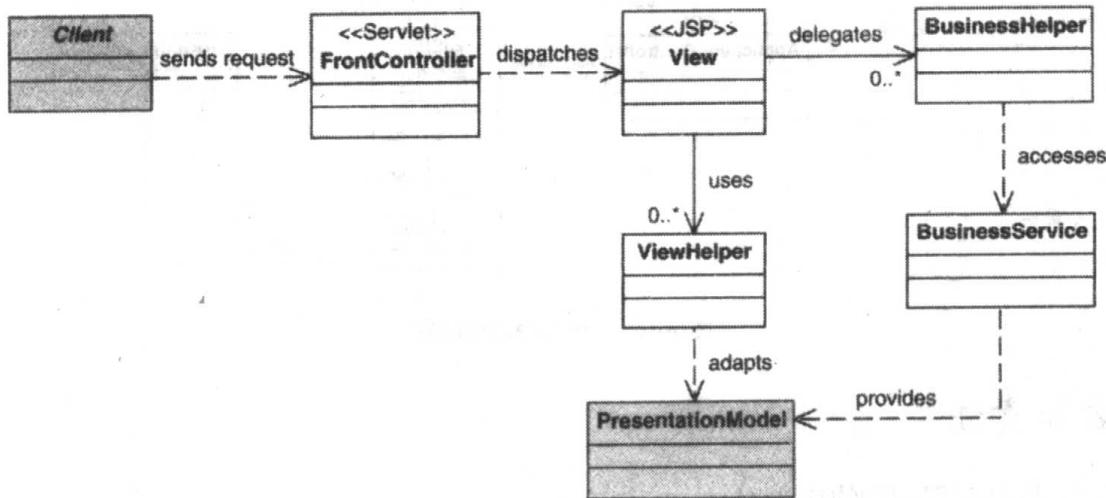
虽然通常会用应用控制器来进行视图管理，但由于分配器视图模式中对视图管理功能需求有限，所以视图管理可以直接封装在控制器里（见前端控制器一节的“控制器中的分配器”策略），或者封装在视图中。如果系统中只有很少的操作管理逻辑或是根本没有操作管理，那么使用“控制器中的分配器”策略就很常见。

见后面的图6-48和图6-49。

事实上，如果采取分配器视图的做法，那么最常见的做法是视图管理由容器完成，因为往往根本不存在任何整个应用系统级别的逻辑了。举例来说，可能有一个视图叫“main.jsp”，并

通过部署配置一个别名。容器处理以下请求，把别名翻译成物理资源的实际名称，并把请求分派到那个视图上：

`http://some.server.com/first --> /mywebapp/main.jsp`



291

图6-48 “控制器中的分配器”策略类图

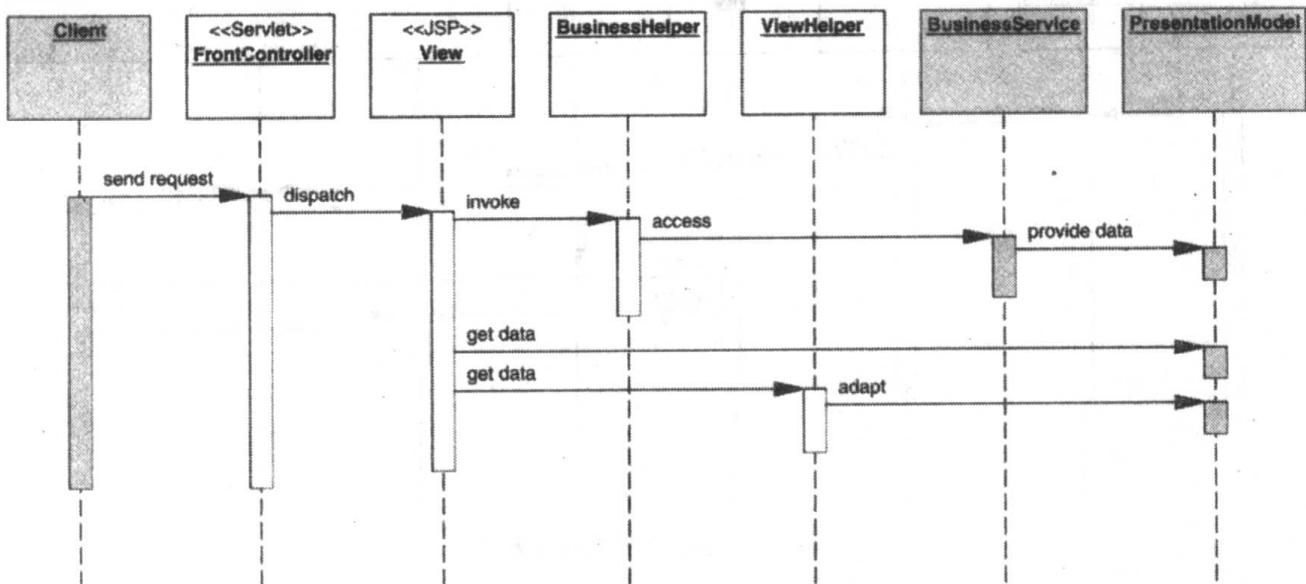


图6-49 “控制器中的分配器”策略序列图

请注意在这个场景中，应用控制器模式里的各个角色是怎样由容器机制负责实现的。

正如序列图所示，如果需要进行任何业务处理的话，在本模式中这种业务处理会一直推迟到控制权交给视图后才被执行。另一方面，服务到工作者模式则会在把控制权交给视图之前就完成业务处理。所以，在这一点上，这两种模式恰好形成了两个极端，而对于同一个应用系统，通常会分别使用这两种模式，以满足不同用例的需要。

#### FrontController（前端控制器）

FrontController（前端控制器）可以用于请求的最初处理，虽然在这种用法中它的责任是有限的。

### ApplicationController (应用控制器)

ApplicationController (应用控制器) 可以用于进行有限的视图管理, 但不负责操作管理。也可以由容器直接完成应用控制器的作用和责任。

### View (视图)

视图向客户端表现和显示数据。由视图助手来调整和转化PresentationModel (表现模型), 从而形成最后的显示内容。视图可以是复合视图。

### BusinessHelper (业务助手)、ViewHelper (视图助手)

助手类负责帮助视图或控制器来进行特定的处理。在处理请求时, BusinessHelper (业务助手) 帮助控制器初始化业务处理过程; 而ViewHelper (视图助手) 则负责获取并调整 PresentationModel (表现模型), 从而生成视图。292

### PresentationModel (表现模型)

PresentationModel (表现模型) 中包含从业务服务获取的数据, 用于生成视图。

### BusinessService (业务服务)

BusinessService (业务服务) 封装了业务逻辑和业务状态。远程业务服务通过业务代理访问。

## 策略

以下是与分配器视图相关的一些策略。

- “前端控制器”中的“*Servlet前端策略*”。
- “前端控制器”中的“*JSP前端策略*”。
- “视图助手”中的“*基于模板的视图策略*”。
- “视图助手”中的“*基于控制器的视图策略*”。
- “视图助手”中的“*JavaBean助手策略*”。
- “视图助手”中的“*定制标记助手策略*”。
- “前端控制器”中“*控制器中的分配器*”。

## 效果

### 充分利用了框架和类库

很多框架和类库都专门实现了或支持着一些模式。各种标准库和定制库提供了许多视图适配器和转换器, 另外还有一些用途有限的数据访问标记; 这样分配器视图模式就能够获得这些类库的充分支持。标准库中最常见的一个例子是JSTL (标准标记库) [JSTL]。

### 可能导致视图与模型/控制逻辑之间区分不清

在分配器视图模式中, 由于业务处理是视图完成的, 所以本模式不适合那些业务处理或数据访问较多的请求操作。正如前面“前端控制器”和“视图助手”等部分讨论过的一样, 应该尽量避免把任何形式的处理逻辑放进视图。设计的首要目标, 就是要把控制逻辑和业务逻辑从视图中分离出来, 并且隔离那些各自独立的处理逻辑。

### 把处理逻辑从视图中分离出来, 增进了可重用性

本模式可以使用视图助手来调整、转换表现模型，用于视图。这样，就避免了把处理逻辑混入视图中，而是把这些逻辑抽取出来，形成了可重用的组件，这样就能避免暴露实现细节，而只是显示出代码的设计意图。

## 示例代码

以下代码给出了分配器视图模式的一种实现，其中包括的策略有：*Servlet前端策略*、“*控制器中的分配器*”策略、*基于模板的视图策略*以及*定制标记助手策略*和*JavaBean助手策略*。同时还使用了一个非常基本的复合视图。在图6-51中给出了最后的显示结果。

例6.75给出了控制器servlet的代码，它只是完成身份认证，然后就把控制权交给合适的视图。请注意，控制器本身并不直接委派任何助手组件来调用业务层服务。如果请求处理中需要调用业务服务，那么这种调用也会被推迟到控制权交给视图时再进行。本例中的视图名称是accountdetails.jsp，代码如例6.76所示。

示例代码中使用了LogManager（日志管理器）来记录日志信息。为了示例清楚，我们在输出页面的底部显示了这些日志信息，如图6-50、图6-51所示。

### 例6.75 分配器视图控制器Servlet

```

1  public class Controller extends HttpServlet {
2
3      /** 处理HTTP <code>GET</code> 和
4       * <code>POST</code> 方法的请求。
5       * @param request servlet请求
6       * @param response servlet响应
7       */
8
9      protected void processRequest(HttpServletRequest request,
10          HttpServletResponse response)
11         throws ServletException, java.io.IOException {
12
13         String nextview;
14
15         try {
16             LogManager.recordStrategy(request, "Dispatcher View",
17                 " Servlet Front Strategy; " +
18                 " Template-Based View Strategy; " +
19                 " Custom tag helper Strategy");
20             LogManager.logMessage(request, getSignature(),
21                 "Process incoming request. ");
22
23             // 使用助手对象，收集
24             // 参数信息
25             RequestHelper helper = new
26                 RequestHelper(request, response);
27             LogManager.logMessage(request, getSignature(),
28                 " Authenticate user");
29
30             Authenticator auth = new BasicAuthenticator();
31             auth.authenticate(helper);

```

```

29
30      // 出于篇幅考虑，这里的示例作了极大的简化。
31      // 通常，在这里需要完成
32      // 从逻辑名称到资源名称的
33      // 映射。
34      LogManager.logMessage(request, getSignature(),
35          "Getting nextview");
36      nextview = request.getParameter("nextview");
37
38      LogManager.logMessage(request, getSignature(),
39          "Dispatching to view: " + nextview);
40  }
41  catch (Exception e) {
42      LogManager.logMessage("Handle exception appropriately",
43          e.getMessage() );
44
45
46      /** ApplicationResources（应用资源）提供了一组简单的API,
47       * 用于获取参数和其他的
48       * 预定义值**/
49      nextview = ApplicationResources.getInstance().
50          getErrorPage(e);
51  }
52  dispatch(request, response, nextview);
53 }
54
55 /** 处理HTTP <code>GET</code> 方法。
56  * @param request servlet请求
57  * @param response servlet响应
58  */
59 protected void doGet(HttpServletRequest request,
60     HttpServletResponse response)
61     throws ServletException, java.io.IOException {
62     processRequest(request, response);
63 }
64
65 /** 处理HTTP <code>POST</code> 方法。
66  * @param request servlet请求
67  * @param response servlet响应
68  */
69 protected void doPost(HttpServletRequest request,
70     HttpServletResponse response)
71     throws ServletException, java.io.IOException {
72     processRequest(request, response);
73 }
74
75 /** 返回关于servlet的简要说明*/
76 public String getServletInfo() {

```

```

77         return getSignature();
78     }
79
80     public void init(ServletConfig config) throws ServletException {
81         super.init(config);
82     }
83
84     public void destroy() { }
85
86     /**
87      * 分配器方法
88      */
89     protected void dispatch(HttpServletRequest request,
90                           HttpServletResponse response, String page)
91     throws javax.servlet.ServletException, java.io.IOException {
92         RequestDispatcher dispatcher =
93             getServletContext().getRequestDispatcher(page);
94         dispatcher.forward(request, response);
95     }
96
97     private String getSignature() {
98         return "DispatcherView-Controller";
99     }
100 }
```

296

注意：为了获取表现模型，视图使用了定制标记助手来实现业务服务的调用。这样使用定制标记的时候，应该让标记委派到具体执行操作的单个组件。这样，通用的处理逻辑与标记的实现之间只存在松耦合。如果不这样结合使用标记助手和分配器视图，那么就有可能把处理逻辑混合在视图之中，而这正是我们需要避免的。

#### 例6.76 视图—accountdetails.jsp

```

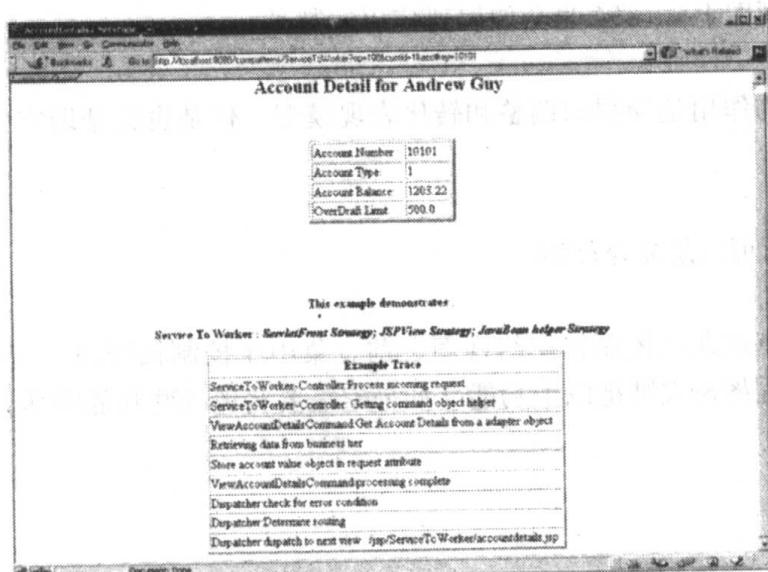
1  <%@ taglib uri="/web-INF/corepatternstaglibrary.tld"
1  prefix="corepatterns" %>
2
3  <html>
4  <head><title>AccountDetails</title></head>
5  <body>
6
7  <corepatterns:AccountQuery queryParams="custid,acctkey" scope="request" />
8
9  <h2>
10 <center> Account Detail for <corepatterns:Account attribute="owner" />
11 </h2>
12 <br><br>
13
14 <tr>
15   <td>Account Number :</td>
16   <td><corepatterns:Account attribute="number" /></td>
```

```

17  </tr>
18
19
20  <tr>
21      <td>Account Type:</td>
22      <td><corepatterns:Account attribute="type" /></td>
23  </tr>
24
25  <tr>
26      <td>Account Balance:</td>
27      <td><corepatterns:Account attribute="balance" /></td>
28  </tr>
29
30  <tr>
31      <td>OverDraft Limit:</td>
32      <td><corepatterns:Account attribute="overdraftLimit" /></td>
33  </tr>
34  <table border=3>
35  </table>
36 </corepatterns:AccountQuery>
37
38  <br>
39  <br>
40
41  </center>
42  <%@ include file="/jsp/trace(seg" %>
43  </body>
44 </html>

```

297



298

图6-50 服务到工作者模式的示例输出<sup>①</sup>

<sup>①</sup> 从本图的日志信息可以看出，服务到工作者模式中，在把控制权交给视图之前就已经完成了业务处理。

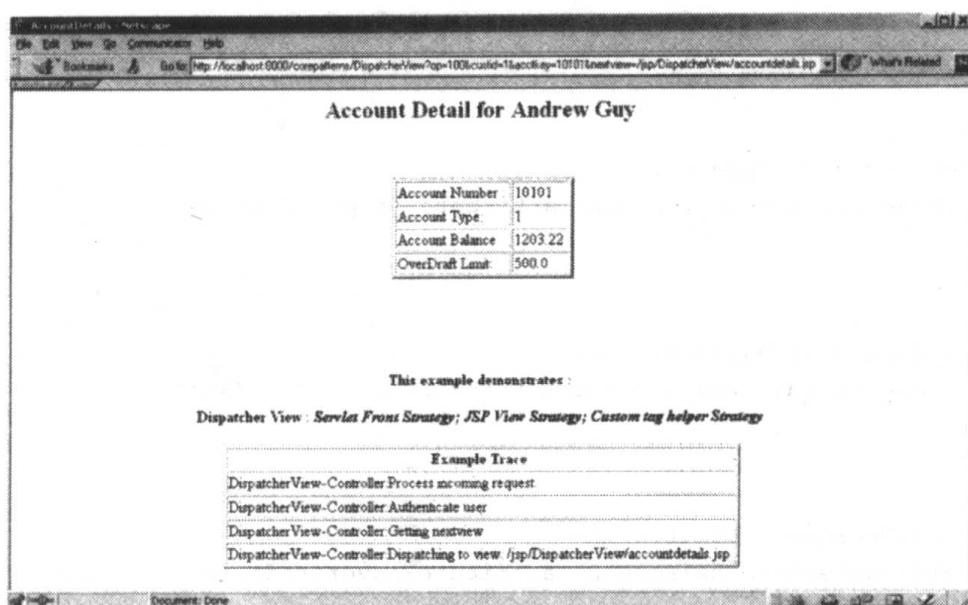


图6-51 分配器视图模式的示例输出

## 相关模式

- **前端控制器**

在分配器视图模式中，可以使用前端控制器处理请求，也可以从一开始就使用视图来处理请求。

- **应用控制器**

分配器视图模式往往并不使用应用控制器。如果系统中需要一定的视图管理功能（用于把请求解析到实际的视图上），那么也会使用到应用控制器。

- **视图助手**

视图助手主要的作用是为视图调整和转化表现模型，但是也会帮助完成视图中任何有限的业务处理操作。

- **复合视图**

本模式中的视图可以是复合视图。

- **服务到工作者**

服务到工作者模式在把控制权交给视图之前，集中了控制权管理、请求处理和业务处理。与此相反，分配器视图模式则把以上功能（如果它需要实现这些功能的话）推迟到视图处理时才进行。

## 第7章 业务层模式

模型

本章将涉及以下主题：

- 业务代表
- 服务定位器
- 会话门面
- 应用服务
- 业务对象
- 复合实体
- 传输对象
- 传输对象组装器
- 值列表处理器

300  
301

本章从企业级应用集成与设计的视角出发，对业务层模式进行深入分析。通过本章的学习，读者将能够掌握如何在企业级应用集成与设计中有效地利用业务层模式，从而提高系统的可维护性和可扩展性。

在企业级应用集成与设计中，业务层模式是一种重要的设计模式。它通过将业务逻辑分离出来，使得系统更加模块化、易于维护和扩展。业务层模式通常包括以下几个部分：业务代表（Business Representative）、服务定位器（Service Locator）、会话门面（Session Facade）、应用服务（Application Service）、业务对象（Business Object）、复合实体（Composite Entity）、传输对象（Transport Object）和传输对象组装器（Transport Object Assembler）。值列表处理器（Value List Processor）也是业务层模式的一个重要组成部分，它负责处理和转换值列表数据。

## 业务代表<sup>①</sup>

### 问题

与业务服务组件（business service component）远程通信的任务复杂度很高，需要对客户端隐藏这种复杂度。

当客户端直接访问远程业务服务组件时，可能发生以下问题。

客户端直接与业务服务接口交互。这意味着，当这些业务服务接口代码改变时，客户端代码可能也要改变。这就增加了系统维护的工作量，降低了系统的灵活性。

另一个问题与网络传输性能有关。当客户端直接与业务服务API交互时，客户端的一个操作，就可能需要与业务服务进行多次复杂的、细粒度的交互。这将导致把业务逻辑完全置入客户端，不采用任何客户端缓存，也不使用任何服务器端聚合，由此就降低了系统可维护性。而既然多次远程交互都要经过网络完成，这也就降低了系统的网络传输性能。

第三个问题是，让客户端如此紧密地和业务服务API交互，可能意味着，客户端代码为了与一个远程分布的中间层交互，就必须包含一些底层架构<sup>②</sup>代码。这些底层架构代码包括：命名服务（比如JNDI），网络连接故障处理，以及重试逻辑等等。

### 约束

- 需要使表现层组件和客户端（后者包括各种设备、web service、大型客户端程序<sup>③</sup>等）访问业务服务层。
- 要尽可能地降低客户端和业务服务层之间的耦合，从而隐藏服务层的具体实现细节（比如寻址、访问等等）。
- 要避免不必要的远程服务调用。
- 要把网络异常信息转译成应用程序异常，或者是用户使用异常。
- 要把服务创建、重配置、调用重试等细节对客户端隐藏起来。

302

### 解决方案

使用业务代表模式，封装对业务服务的访问。业务代表模式隐藏了服务层的具体实现细节（比如寻址、访问机制等等）。

<sup>①</sup> 业务代表：原文business delegate，又译“业务委派”。Delegate（委托、代表）是设计模式中的基本思路（参见《设计模式》英文版第1章p20的讨论）。而所谓业务代表，是指在客户端和业务服务层之间，增设一个“代表层”，所有客户端到服务器的调用，都“委托”该层完成（另可参见本书第5章中“术语”部分的介绍）。“委派”是动词，而“delegate”指的是该层所起的作用，故译为“代表”，望读者注意体察，勿与“代理（proxy或broker）”混淆。

<sup>②</sup> 底层架构：infrastructure，这里指系统架构中，不参与业务逻辑，专门调用特定底层服务的部分。

<sup>③</sup> 大型客户端程序：rich client，指界面表现丰富的客户端程序，通常用传统语言编写，与单纯的浏览器网页有别。

业务代表能起到客户端业务抽象层的作用：它抽象并隐藏了业务服务层的实现细节。因此，采用业务代表能够降低客户端和业务服务层之间的耦合。根据业务代表模式的具体实现策略不同，它能不同程度地起到隔离作用，使客户端免受业务服务层改动的影响。所以，采用本模式具有一定的潜在效果，当业务服务层API或者它的底层实现改动时，客户端所需要的改动可能大大减少。

但是，当业务服务层API发生改动时，业务代表的接口方法可能仍需修改。不过显然，业务服务层内部发生改动的可能性，远比业务代表改动的可能性大得多。

如果一个设计目标会导致一系列从头到尾（up front）的附加工作，而回报却要未来才能看到（比如把业务层抽象出来之类的设计），开发人员可能对此抱有怀疑。不过，使用业务代表确实具有以下好处：

- 主要好处是隐藏了底层服务的细节。比如说，采用了业务代表后，命名、寻址之类的服务对客户端来说就完全透明了[亦即，更改这些服务的实现细节不会影响到客户端代码]。
- 业务代表能够处理各种服务端异常，比如java.rmi.Remote异常、JMS异常等等。业务代表可以把这些系统级的异常转化成应用程序级异常。对于客户端来说，应用程序级异常比较容易处理，而对实际用户来说，往往也更友好。
- 当服务层工作发生故障时，业务代表能够透明地做出重试或者恢复操作，无需让客户端知道这类异常——只有业务代表能够断定，这个问题没法自行解决时，才会通知客户端。有了这样明显的收益，采用业务代表模式自然顺理成章。<sup>②</sup>
- 业务代表能缓存结果数据，以及对远程业务服务的引用。采用缓存，能够显著改善系统性能。因为系统中可能存在一些多余的往返传输（round trip），它们会造成一定潜在开销，缓存能够大大限制这些往返传输，从而提高网络传输性能。

业务代表会使用服务定位器来定位特定业务服务。服务定位器负责隐藏业务服务的寻址细节。

当业务代表和会话门面模式一起使用时，它们之间往往具有一对一的关系。原因如下：封装在一个业务代表中的业务逻辑，可能会与多个业务服务产生交互，由此，本来会在业务代表和业务服务之间产生一对多的关系，但是这样的多个业务服务又往往会被组装到同一个会话门面中。所以业务代表和会话门面就具有了一对一关系。

我们把业务代表归为业务层模式，而不是表现层模式，是因为业务代表是一种逻辑抽象，而不是物理抽象。在同表现层协作时，业务代表组件事实上居于表现层之中。但是，这些组件又是业务层的一种扩展。出于以上原因，又因为这一模式与会话门面的紧密联系，我们建议由负责业务服务层的开发人员来实现业务代表。

## 结构

图7-1为业务代表模式的类图。客户端要求BusinessDelegate（业务代表）组件提供到底层业务服务的访问。BusinessDelegate（业务代表）利用一个ServiceLocator（服务定位器）来定位那个特定的BusinessService（业务服务）组件。

303

<sup>②</sup> 本书第一版中，以上三点“好处”没有分段描述，所以“这样明显的收益”统称三者，本句也是一个总结。本版中，这三点分段编辑，但想必忘了对此加以修饰。

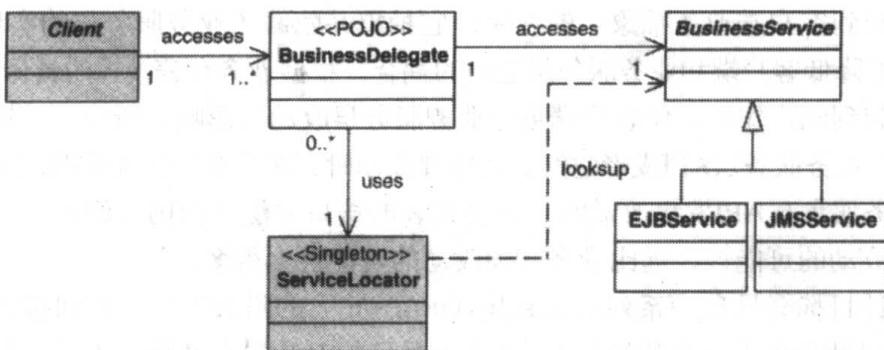


图7-1 业务代表模式的类图

## 参与者和责任

图7-2和图7-3中的序列图，表现了业务代表模式的典型交互场景。

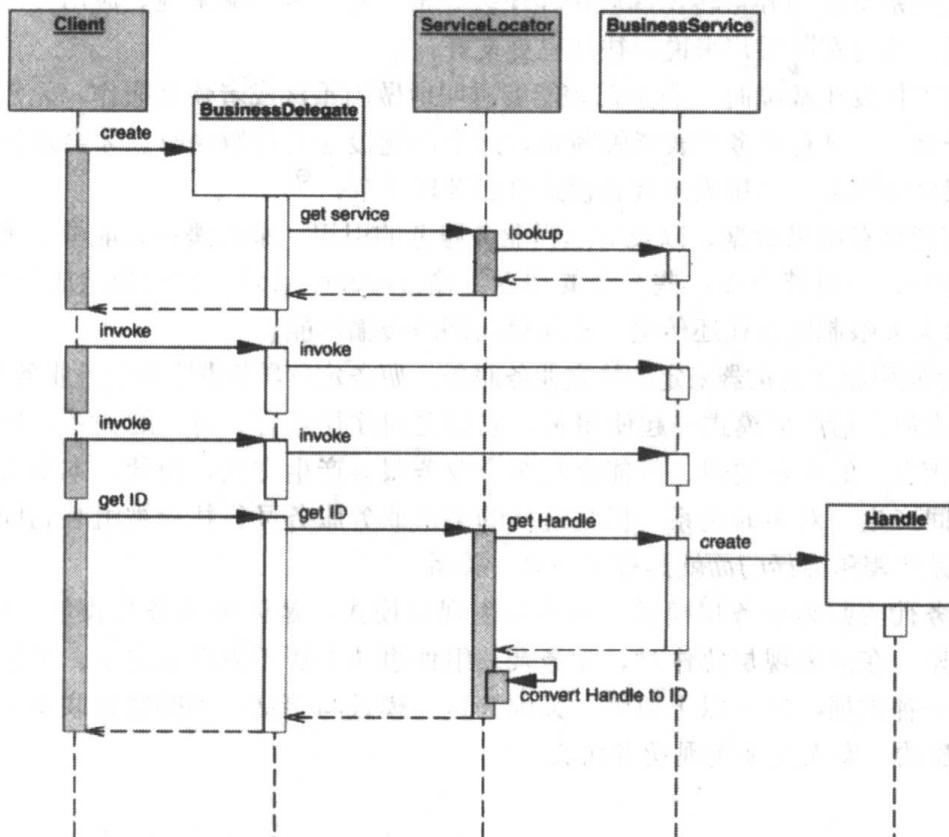


图7-2 业务代表模式序列图

304

序列图中，消息getID表明：BusinessDelegate（业务代表）能够获得一个以String类型表示的BusinessService（业务服务）句柄，比如说是EJBHandle（EJB 句柄）对象。客户端可以利用这个ID字符串，重新连接前一次它使用的同一个BusinessService（业务服务）。

这一技术避免了重新寻址，因为句柄能够重新连接到它对应的BusinessService（业务服务）实例上。值得注意的是，句柄对象是由EJB容器提供商实现的，所以也许不能跨越不同厂商的不同容器产品使用。

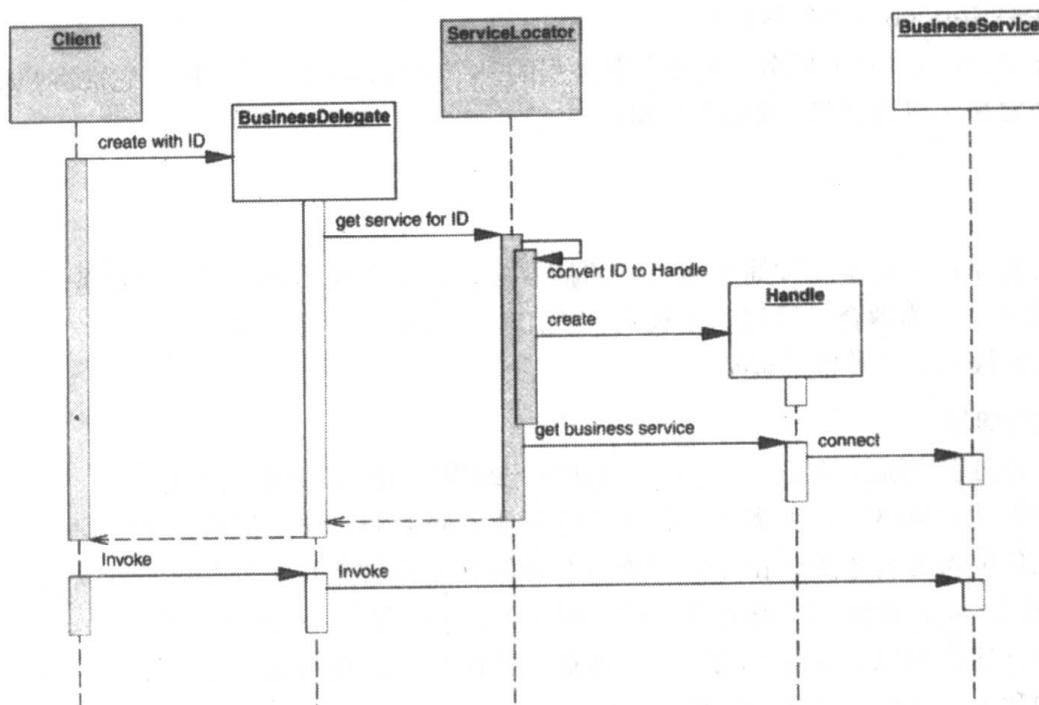


图7-3 带ID的业务代表模式序列图

图7-3中的序列图表现的是，通过使用句柄，客户端获得了前一次使用的BusinessService（业务服务）的引用（这个业务服务可能是session bean、entity bean，等等）。

### BusinessDelegate（业务代表）

BusinessDelegate（业务代表）的作用为业务服务层提供控制和保护。BusinessDelegate向客户端提供两种类型的构造函数：

- 一个默认构造函数，用于实例化BusinessDelegate（业务代表）。
- 一个构造函数，用于带一个ID参数实例化BusinessDelegate（业务代表）。这里的ID参数是一个远程对象的以字符串形式表示的句柄（这个远程对象可以是EJBHome、EJBObject等等）。

当BusinessDelegate（业务代表）被不带ID参数地创建后，它向ServiceLocator（服务定位器）索取业务对象。通常，ServiceLocator由服务定位器模式实现，它返回一个服务工厂对象，比如一个EJBHome。BusinessDelegate（业务代表）使用该服务工厂来定位、创建或删除特定的BusinessService（业务服务），比如一个EJB。

当BusinessDelegate（业务代表）被带ID参数地初始化后，它就利用这个ID字符串，重新连接到原来的BusinessService（业务服务）上，从而对客户端隐藏了BusinessService（业务服务）命名/寻址的实现细节。客户端从不直接远程调用一个BusinessService，而是通过利用BusinessDelegate（业务代表）完成任务。

### ServiceLocator（服务定位器）

ServiceLocator（服务定位器）是通过服务定位器模式实现的一个角色。它封装了定位一个BusinessService（业务服务）组件的实现细节。

305

306

### BusinessService (业务服务)

BusinessService (业务服务) 是业务服务层的一个组件，比如一个EJB，它被客户端所访问。这个组件经常被实现为会话界面或JMS组件。

## 策略

业务代表模式提供了一种简单而强大的抽象层，从而降低了业务层中的业务服务组件与应用的其他部分之间的耦合。通过这个模式，应用开发人员很容易就能够使用业务服务。以下是实现业务代表模式的几种常见策略。

### 委派代理策略<sup>①</sup>

业务代表向客户端提供了一个接口，由此客户端得以访问业务服务API。

使用委派代理策略时，业务代表起到了远程业务服务的代理的作用，把客户端的方法调用传递到所代理的服务对象上。由于实现业务代表的是业务层开发人员，我们可以在这里方便地加入诸如验证、业务数据/引用缓存等功能。缓存中，可以包括session bean的home的引用或远程对象的引用，这样可以减少寻址次数，从而提高系统性能。利用服务定位器，业务代理也能对字符串型ID和上述引用之间进行转换。

### 委派适配器策略

在运行一个B2B环境时应该想到，与系统交互的那些外接系统不一定都是J2EE应用。必须给这些外部系统提供与系统交互的整合方案，在这里使用XML是一种常见做法。对于完成这类任务，委派适配器（delegate adapter）是一个好的策略。图7-4是一个例子。

307

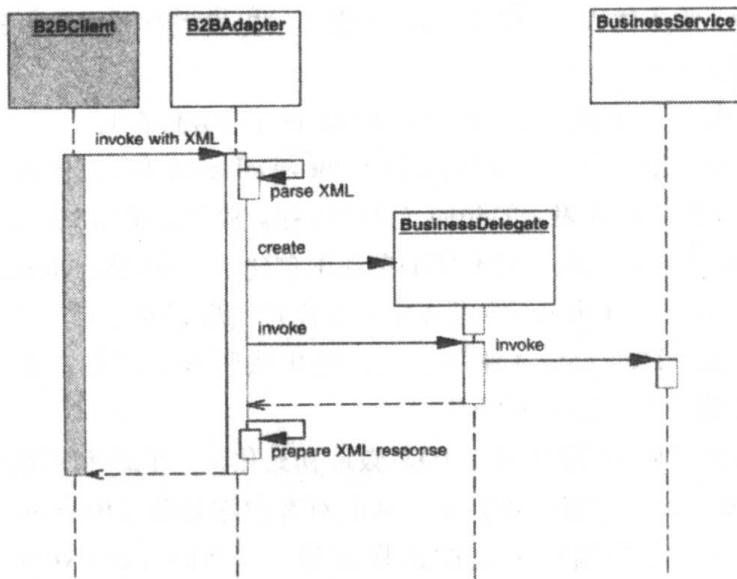


图7-4 在业务代表中使用适配器策略

<sup>①</sup> 委派代理策略：本版此处作delegate's strategy。而本书第一版此处为delegate proxy strategy（委派代理策略），本版下文也作delegate proxy strategy。这个标题改动令人费解，故本处仍译为“委派代理策略”。“代理(proxy)”是一种重要的结构型设计模式。注意它与“代表/委派/委托(delegate)”的区别。

## 效果

- **减小耦合，增进了可维护性**

通过隐藏所有业务层的实现细节，业务代表模式减小了表现层和业务层的耦合。由于所有改动都被集中于业务代表上，修改维护变得相当容易。

- **转译业务服务异常**

业务代表模式把与网络和底层架构相关的异常转译成业务异常，这样一来，客户端可以不去过问底层实现的具体细节知识。

- **提高可用性<sup>①</sup>**

当业务代表遇到业务服务故障时，无需把问题提交给客户端，业务代表首先就能尝试自动恢复。如果恢复成功，客户端甚至可以对此毫无察觉。如果自动恢复也失败了，业务代表就会向客户端通知故障。另外，如果必要，业务代表的方法还可以做到同步（synchronized）。

- **为业务层提供一个简单、统一的接口**

业务代表以一个简单Java对象形式实现，这样，应用开发人员就能更容易地使用业务层组件，无需跟业务服务的复杂实现细节打交道。

- **改善系统性能**

业务代表能够为表现层组件缓存信息，这样很多公用服务请求的效率就能够大大提高。

- **引入一个附加层**

业务代表模式引入了一个附加层，这可能增加了系统复杂性，降低了可维护性。但该模式的收益远抵消了这些弊病。

- **隐藏远程访问**

位置透明性[指该模式使客户端不直接处理远程访问]是这个模式的一个好处。但如果记不住业务代表位于哪个[物理]层面，该模式的这个特点也会引发问题。业务代表是一个客户端代理，用于委派执行远程服务调用。即便业务代表是用POJO实现的，在调用业务代表上的方法时，通常它也必须跨越网络，访问远程业务服务，从而完成这个请求。所以，也应该尽量减少调用业务代表，以免引起网络拥堵。

## 示例代码

### 实现业务代表模式

考虑一个专业服务应用系统<sup>②</sup>（Professional Service Application, PSA），该系统把资源（也就是咨询顾问）指派给项目（也就是外包的工程）。表现层组件必须访问一个会话界面。我们利

① 可用性：原文availability。high availability（高可用性）是目前服务器端运算的热门话题。

② 专业服务应用系统，这是全书的一个常用例子，本节中首次出现。请注意这个例子中各个业务实体的含义：事实上这是一个人力资源外包公司，所管理的“人力资源”也就是具有专长的咨询顾问人员；公司与其他企业签订合同，把这些“资源”分包给那些企业的各个“项目”；每个“资源”——也就是每位咨询顾问都有自己的技能集（skillset）、停用时间（block out time）。

用业务代表实现一个ResourceDelegate（资源代表）对象，该对象封装了与会话门面ResourceSession（资源会话）交互的细节。

本例中，ResourceDelegate（资源代表）的实现代码如例7.1所示，相应的会话门面ResourceSession（资源会话）远程接口如例7.2所示。

### 例7.1 实现业务代表模式—ResourceDelegate（资源代表）

```

1
2 // imports
3 ...
4
[309] 5 public class ResourceDelegate {
6
7     // 会话门面的远程引用
8     private ResourceSession session;
9
10    // 会话门面Home对象的类
11    private static final Class homeClazz =
12        corepatterns.apps.psa.ejb.ResourceSessionHome.class;
13
14    // 默认构造器。负责home对象的寻址，并创建一个新的session bean,
15    // 然后连接到该session bean。
16    public ResourceDelegate() throws ResourceException {
17        try {
18            ResourceSessionHome home =
19                (ResourceSessionHome) ServiceLocator.getInstance().
20                    getRemoteHome("Resource", homeClazz);
21            session = home.create();
22        } catch (ServiceLocatorException ex) {
23            // 把服务定位器异常翻译为
24            // 应用异常
25            throw new ResourceException(...);
26        } catch (CreateException ex) {
27            // 把session bean创建异常翻译为
28            // 应用异常
29            throw new ResourceException(...);
30        } catch (RemoteException ex) {
31            // 把远程异常翻译为
32            // 应用异常
33            throw new ResourceException(...);
34        }
35    }
36
37    // 另一种构造器。接受一个ID（句柄ID），连接到一个此前存在的
38    // session bean而不是创建一个
39    // 新的session bean。
40    public ResourceDelegate(String id)
41        throws ResourceException {

```

```
42     // 按照给定的ID重新连接到响应的session bean上。
43     reconnect(id);
44 }
45
46 // 返回一个字符串ID，以便客户端此后使用该ID
47 // 重新连接到session bean
48 public String getID() {
49     try {
50         return ServiceLocator.getId(session);
51     } catch (Exception e) {
52         // 抛出应用异常
53     }
54 }
55
56 // 本方法使用字符串ID实现重新连接
57 public void reconnect(String id) throws ResourceException {
58     try {
59         session =
60             (ResourceSession) ServiceLocator.getService(id);
61     } catch (RemoteException ex) {
62         // 把远程异常翻译成
63         // 应用异常
64         throw new ResourceException(...);
65     }
66 }
67
68 // 以下是代理给会话门面的业务方法。
69 // 如果遇到任何服务异常，
70 // 这些方法就将其转换成业务异常，
71 // 比如ResourceException、SkillSetException等等
72 public ResourceTO setCurrentResource(String resourceId)
73 throws ResourceException {
74     try {
75         return session.setCurrentResource(resourceId);
76     } catch (RemoteException ex) {
77         // 把服务异常翻译成
78         // 应用异常
79         throw new ResourceException(...);
80     }
81 }
82
83 public ResourceTO getResourceDetails()
84 throws ResourceException {
85
86     try {
87         return session.getResourceDetails();
88     } catch (RemoteException ex) {
```

```

311
89     // 把服务异常翻译成
90     // 应用异常
91     throw new ResourceException(...);
92 }
93 }
94
95 public void setResourceDetails(ResourceTO to)
96 throws ResourceException {
97 try {
98     session.setResourceDetails(to);
99 } catch (RemoteException ex) {
100     throw new ResourceException(...);
101 }
102 }
103
104 public void addNewResource(ResourceTO to)
105 throws ResourceException {
106 try {
107     session.addResource(to);
108 } catch (RemoteException ex) {
109     throw new ResourceException(...);
110 }
111 }
112
113 // 所有其他引向session bean的代理方法
114 ...
115 }

```

### 例7.2 ResourceSession (资源会话) 的远程接口

```

312
1 // imports
2 ...
3 public interface ResourceSession extends EJBObject {
4
5     public ResourceTO setCurrentResource(String resourceId)
6     throws RemoteException, ResourceException;
7
8     public ResourceTO getResourceDetails()
9     throws RemoteException, ResourceException;
10
11    public void setResourceDetails(ResourceTO resource)
12    throws RemoteException, ResourceException;
13
14    public void addResource(ResourceTO resource)
15    throws RemoteException, ResourceException;
16
17    public void removeResource()
18    throws RemoteException, ResourceException;
19

```

```

20 // 以下方法管理资源的停用时间
21 public void addBlockoutTime(Collection blockoutTime)
22 throws RemoteException, BlockoutTimeException;
23
24 public void updateBlockoutTime(Collection blockoutTime)
25 throws RemoteException, BlockoutTimeException;
26
27 public void removeBlockoutTime(Collection blockoutTime)
28 throws RemoteException, BlockoutTimeException;
29
30 public void removeAllBlockoutTime()
31 throws RemoteException, BlockoutTimeException;
32
33 // 以下方法用于管理特定资源的技能集
34 public void addSkillSets(Collection skillSet)
35 throws RemoteException, SkillSetException;
36
37 public void updateSkillSets(Collection skillSet)
38 throws RemoteException, SkillSetException;
39
40 public void removeSkillSet(Collection skillSet)
41 throws RemoteException, SkillSetException;
42
43 ...
44 }
45

```

## 相关模式

- 服务定位器

业务代表通常使用服务定位器封装业务服务寻址的实现细节。当业务代表需要进行业务服务的寻址时，它就把相应的寻址操作委派给服务定位器。

- 会话门面

在大多数EJB应用系统中，业务代表都会与会话门面交互，并且和该门面保持一对一关系。通常，负责实现会话门面的开发人员也要提供相应的业务代表实现。

313

- 代理 (*Proxy*) [GoF]

业务代表为业务层对象提供了一种“替身”，所以也就能起代理的作用。委派代理策略就提供了这种功能。

- 适配器 (*Adapter*) [GoF]

业务代表可以使用适配器设计模式来实现与其他系统的集成（如果不使用适配器模式，这种集成往往无法实现，系统之间很难做到兼容）。

- 中转 (*Broker*) [POSA1]

业务代表解除了业务层组件与其他层的客户端之间的耦合，所以也就起到了中转的作用。

314

## 服务定位器

### 问题

需要以一种统一的、透明的方式定位业务组件和业务服务。

J2EE应用系统的客户端需要定位、访问业务层的组件和服务。比如，如果表现层的一个业务代表需要访问业务层的一个会话界面，这个业务代表就首先要执行对这个会话界面的EJB Home的寻址，然后调用home的create（创建）方法，获得一个会话界面实例。与此类似，当一个JMS客户端需要获得一个JMS Connection（连接）或JMS Session（会话）的时候，它首先也要执行JMS ConnectionFactory（连接工厂）对象的寻址，然后再通过这个工厂获得Connection或Session对象。

在J2EE应用系统中，业务层组件（比如EJB组件）和集成层组件（比如JDBC数据源和JMS组件）通常都要在一个中央注册表（registry）中注册。客户端使用JNDI（Java命名和目录接口）API来与这个注册表交互，获得一个InitialContext（初始上下文）对象，该对象中装载着组件的名称/对象绑定信息。而为客户端实现寻址机制的时候，也要对付好几个问题，分别跟复杂性、代码重复、性能恶化和厂商依赖性有关。

使用JNDI API和InitialContext（初始上下文）可能会相当复杂，因为这可能要包括重复使用InitialContext对象、寻址操作、强制类型转化操作以及对底层异常和超时的处理操作。应用系统客户端需要从这种复杂性中隔离出来。不然，在多种不同的客户端中就会重复JNDI代码，因为所有访问一个由JNDI管理的服务/组件的客户端都要执行同样的寻址。创建一个JNDI InitialContext对象，执行EJB home对象的寻址可能是一种代价很高的操作，如果反复执行这样的操作，可能会引起系统性能恶化。

另一个问题是：InitialContext（初始上下文）和JNDI注册表中注册的其他context工厂都是厂商提供的实现。如果应用系统客户端直接访问以上对象的这些特殊实现，那么就会给应用系统引入厂商/产品依赖性，使代码难以移植。

315

### 约束

- 要利用JNDI API完成业务组件、业务服务的寻址和调用，其中业务组件中包括EJB和JMS组件等，业务服务则包括数据源等。
- 要把J2EE应用系统客户端的寻址机制集中起来并实现重用。
- 要封装各厂商对注册表的不同实现，并对客户端隐藏这种厂商依赖性和复杂性。
- 要避免初始上下文（initial context）的创建和服务的寻址引起的性能负载。
- 要通过EJB实例的句柄对象重建到一个事前曾经访问过的EJB实例的连接。

### 解决方案

使用服务定位器，实现、封装对服务/组件的寻址。服务定位器能够隐藏寻址机制的实现细节，封装这一机制对不同实现的依赖。

应用系统客户端可以通过服务定位器实现重用，降低代码的复杂性，提供唯一的控制点，并且提供缓存机制，改善系统性能。通常整个应用系统只需要一个服务定位器。不过，在一个应用系统里有两个服务定位器也不罕见，其中一个用于表现层，另一个用于业务层。服务定位器降低了客户端对底层寻址机制的依赖性，并且优化了寻址和对象创建这两个集中使用资源的操作。

服务定位器通常使用单件（*Singleton*）模式实现。但是，J2EE应用系统运行在J2EE容器中，而容器通常会使用多个类装载器（class loader）和Java虚拟机，这样就不可能真正实现只有一个实例的单件。所以，在分布式的环境中，可能需要多个服务定位器实例。不过，服务定位器并不缓存那些需要并发访问的数据，所以也就用不着在多个实例之间实现修改的复制和同步。

## 结构

图7-5中的类图体现了服务定位器模式中的关系。

316

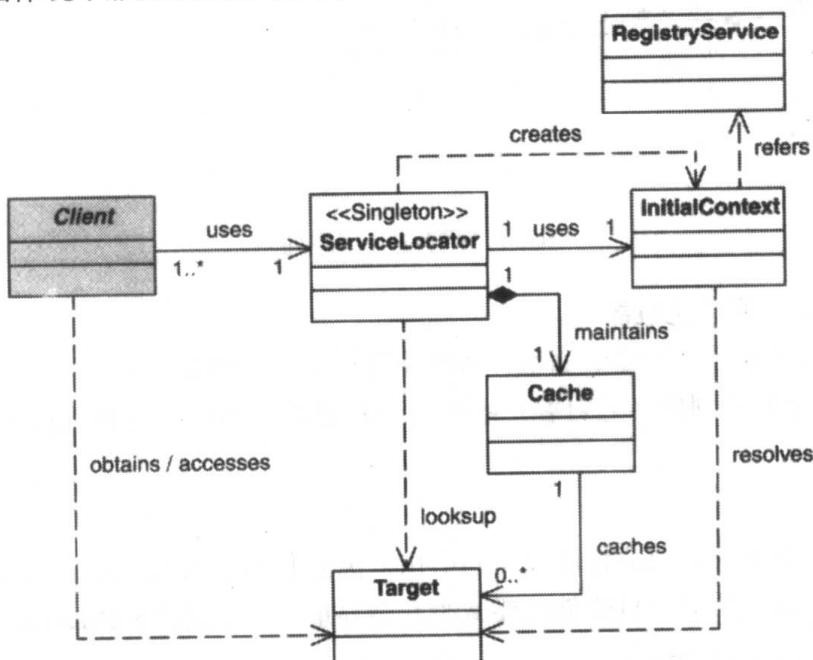


图7-5 服务定位器类图

## 参与者和责任

图7-6中的序列图，体现了服务定位器模式中多个参与者之间的交互。

### Client（客户端）

Client（客户端）也就是服务定位器的一个客户端，它需要定位/访问业务层或集成层的组件/服务。比如，一个业务代表在定位、访问相关的会话门面的时候，就充当了服务定位器的客户端这个角色。同样，当一个数据访问对象使用服务定位器获取一个JDBC DataSource（数据源）实例的时候，它充当的也是Client的角色。

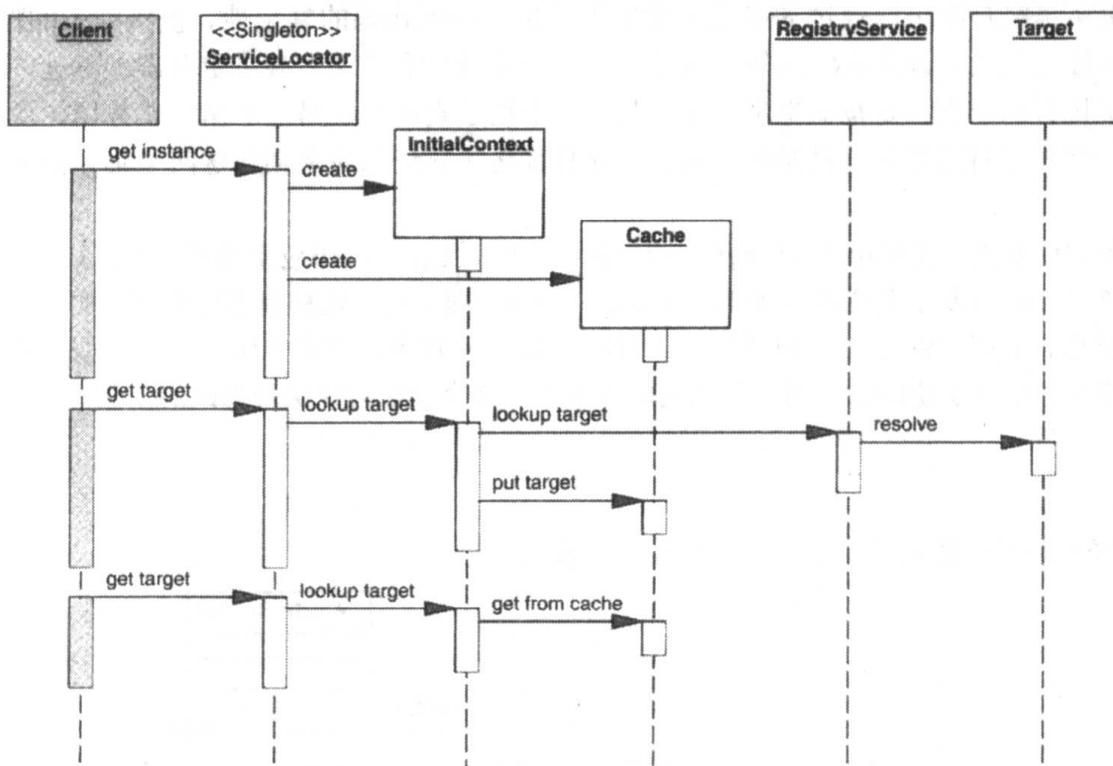


图7-6 服务定位器序列图

### ServiceLocator（服务定位器）

317 ServiceLocator（服务定位器）封装了API寻址（命名）服务、对厂商产品的特定依赖、寻址复杂性以及业务对象的创建；它对客户端提供一种简单的借口。这就降低了客户端的复杂性，促进了重用。

### Cache（缓存）

Cache（缓存）中保留着以前寻址操作的引用，因此起到了备用的ServiceLocator（服务定位器）的作用。使用Cache的唯一目的就是要减少冗余寻址，从而优化ServiceLocator。

### InitialContext（初始上下文）

InitialContext（初始上下文）对象是整个寻址、创建过程的起点。服务提供者负责提供这个上下文对象，而此对象随着ServiceLocator（服务定位器）寻址Target（目标）类型的不同也会有所不同。如果一个ServiceLocator提供不同类型的Target组件（比如EJB、JMS组件等等），它也就会相应地使用不同类型的上下文对象，其中每一种上下文对象都来自不同的服务提供者。比如，一个EJB应用服务器的上下文提供者，就可能不同于一个JMS服务的上下文提供者。

### Target（目标）

318 Target（目标）也就是Client（客户端）通过ServiceLocator（服务定位器）寻址的业务层/集成层的服务或组件。比如，如果Client要对一个EJB组件执行寻址，那么Target也就是EJB Home对象。其他可以充任这个Target角色的组件（取决于寻址的目标）还包括：JDBC DataSource（数据源）实例，JMS ConnectionFactory（连接工厂）对象——比如，在发布/订阅

消息模型中，就是TopicConnectionFactory（主题连接工厂），而在点对点消息模型中，则是QueueConnectionFactory（队列连接工厂）。

### RegistryService（注册表服务）

RegistryService（注册表服务）是一种注册表实现，其中包含了对服务或组件的引用，这些服务/组件注册为服务提供者，可供Client（客户端）使用。RegistryService是一种发布、寻址服务，其中包括JNDI注册表、UDDI注册表以及eBXML中的RegRep注册表。

## 策略

以下策略介绍了用于EJB和JMS组件的服务定位器。还可以在一个服务定位器中结合使用EJB服务定位器策略和JMS服务定位器策略，这样就能够满足所有寻址需求了。

### EJB服务定位器策略

图7-7是EJB服务定位器策略的类图。在J2EE应用系统中，EJB客户端可以使用服务定位器来完成EJB组件的寻址。首先建立JNDI环境（定位、身份认证凭证等等），以便连接到本应用系统所使用的命名和目录服务<sup>Θ</sup>上。Client（客户端）使用一个InitialContext（初始上下文）来定位JNDI注册表服务，并且使用一个已注册的JNDI名称来定位所需的EJB Home对象——这也就是本策略中的Target（目标）。

319

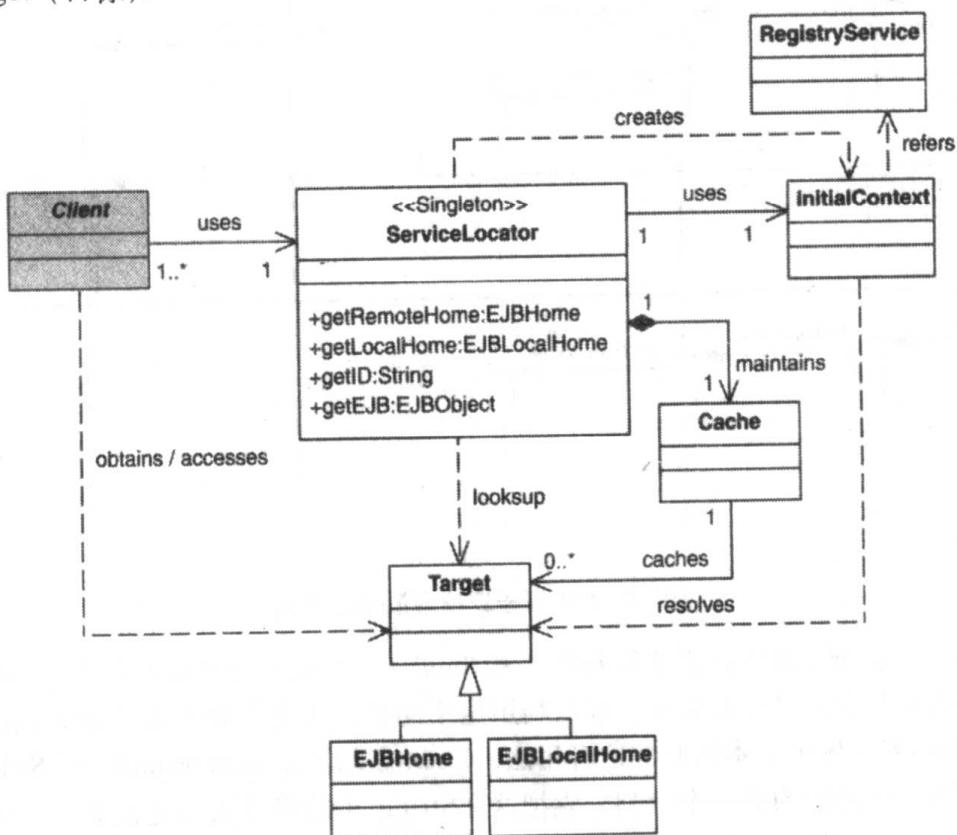


图7-7 EJB服务定位器策略类图

<sup>Θ</sup> “命名和目录服务”，也就是JNDI服务。

获得了EJB Home对象之后，就可以缓存在ServiceLocator（服务定位器）中，以供未来使用，这也就避免了在客户端下次需要同一个home对象的时候再次执行JNDI寻址。客户端获得了EJB Home对象后，它就能够创建、删除或者（对entity bean而言）寻找EJB实例。EJB服务定位器既能够完成本地home对象的寻址，也能完成远程home对象的寻址。

当执行远程home对象寻址的时候，EJB服务定位器需要使用PortableRemoteObject.narrow()方法，把寻址获得的对象转变为正确的EJB Home类。而当执行本地EJB home对象寻址的时候，只需要一次类型转换即可，不必调用PortableRemoteObject.narrow()。

EJB服务定位器策略中各个参与者的交互如图7-8所示。

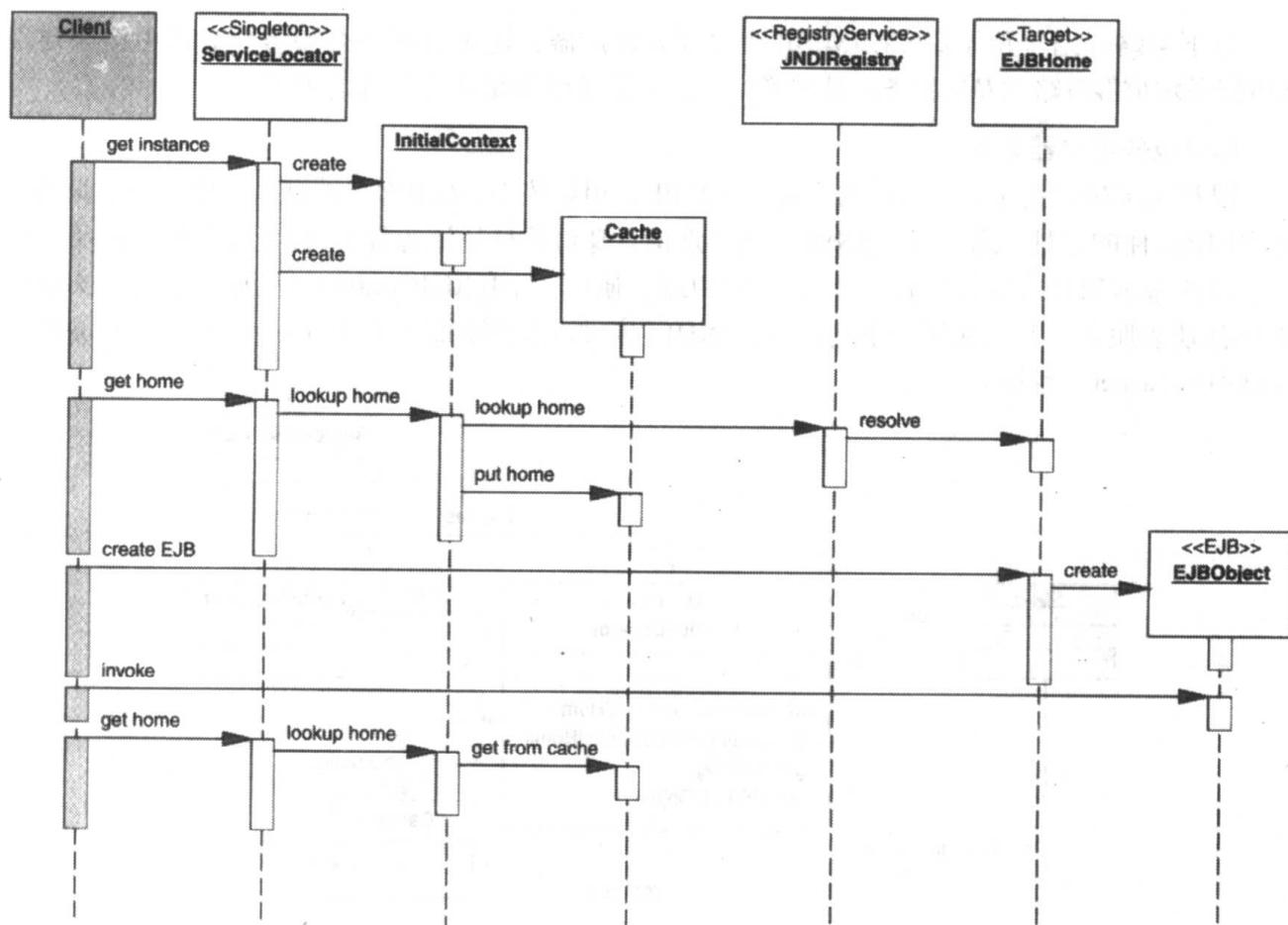


图7-8 EJB服务定位器策略序列图

还可以进一步扩展EJB服务定位器策略，为Client（客户端）提供串行化形式的EJB句柄。只要相应的EJB组件依然存在，Client就可以利用这个句柄在事后重新连接到原来连接过的EJB组件。图7-9中的序列图体现了本策略的这种用法：Client获取了session bean的一个String（字符串）形式的EJB句柄。在其后的另一个时刻，它把这个String ID传送给服务定位器，以便重新连接到同一个session bean。服务定位器把这个ID转化为EJB句柄，然后重新连接到该session bean——只要这个session bean还没有被删除，也没有超时。

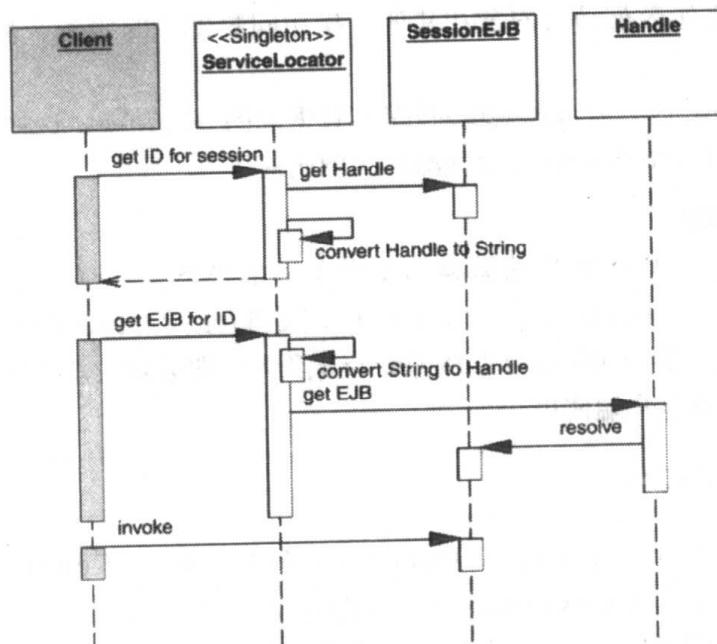


图7-9 使用了EJB句柄的EJB服务定位器策略序列图

### JDBC数据源服务定位器策略

本策略的序列图如图7-10所示。

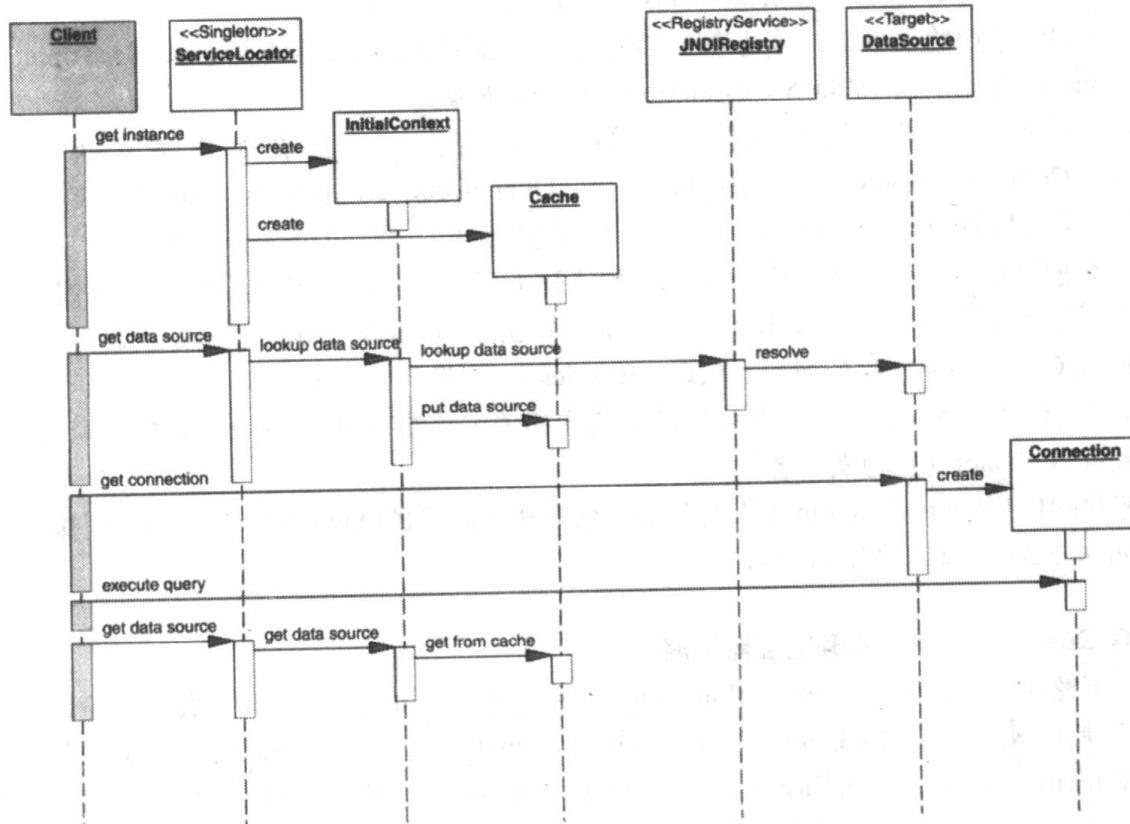


图7-10 JDBC服务定位器策略序列图

很多应用组件，比如数据访问对象和其他直接访问数据库的服务，都需要定位一个JDBC DataSource（数据源）实例。

**322** 服务定位器可以满足这一需求，实现对JDBC数据源的寻址功能。这样，图7-5中的“Target（目标）”角色就由JDBC DataSource（数据源）实例充任了。

### JMS 服务定位器策略

在一个应用系统中，JMS客户端需要为多种不同的JMS组件寻址，比如Topic（主题）、TopicConnectionFactory（主题连接工厂）、Queue（队列）以及QueueConnectionFactory（队列连接工厂）等对象。以下的两种策略，*JMS Queue*（队列）服务定位器策略和*JMS Topic*（主题）服务定位器策略可供JMS客户端使用。

#### 设计手记：客户端和JMS组件

JMS组件——包括Topic（主题）、Queue（队列）、QueueConnection（队列连接）、QueueSession（队列会话）、TopicConnection（主题连接）和TopicSession（主题会话）——的寻址和创建，会涉及以下步骤。注意，在这些步骤中，Topic适用于发布/订阅的消息模型，而Queue则适用于点对点的消息模型。

- 323**
- 首先建立JNDI环境（定位、身份认证凭证等等），以便连接到本应用系统所使用的命名和目录服务上。
  - 客户端通过JNDI命名服务获得JMS服务提供者的初始上下文。
  - 客户端使用初始上下文，通过给出主题或队列的JNDI名称获得该主题/队列。Topic（主题）和Queue（队列）都是JMS Destination（目标）对象。
  - 客户端给出主题/队列连接工厂的JNDI名称，使用初始上下文获得相应的TopicConnectionFactory（主题连接工厂）/QueueConnectionFactory（队列连接工厂）。
  - 客户端使用TopicConnectionFactory（主题连接工厂）获取TopicConnection（主题连接），或者使用QueueConnectionFactory（队列连接工厂）获取QueueConnection（队列连接）。
  - 客户端使用TopicConnection（主题连接）获取TopicSession（主题会话），或者使用QueueConnection（队列连接）获取QueueSession（队列会话）。
  - 客户端使用TopicSession（主题会话）获取相关主题的TopicSubscriber（主题订阅者）或TopicPublisher（主题发布者）。
  - 客户端使用QueueSession（队列会话）获取相关队列的QueueReceiver（队列接受者）或QueueSender（队列发送者）。

### JMS Queue（队列）服务定位器策略

在本策略中，QueueConnectionFactory（队列连接工厂）是JMS组件服务定位器的Target（目标）。使用JNDI名称完成QueueConnectionFactory的寻址。ServiceLocator（服务定位器）可以缓存QueueConnectionFactory，以供将来使用。这也就避免了当客户端再次需要QueueConnectionFactory的时候重复执行JNDI寻址。

ServiceLocator（服务定位器）可以把缓存的QueueConnectionFactory（队列连接工厂）直接

返回给客户端。然后，Client（客户端）就可以使用QueueConnectionFactory（队列连接工厂）创建一个QueueConnection（队列连接）。有了QueueConnection，才能够获得QueueSession（队列会话）、创建Message（消息）、QueueSender（队列发送者，用于把消息发送到队列中）和QueueReceiver（队列接收者，用于从队列中接收消息）。

图7-11是JMS Queue（队列）服务定位器策略的类图。在图中，Queue（队列）是一个JMS Destination（目标）对象，它注册在JNDI注册器服务中，代表着一个JMS队列。可以通过执行JNDI名称的寻址，从初始化上下文中获得这个Queue对象。

324

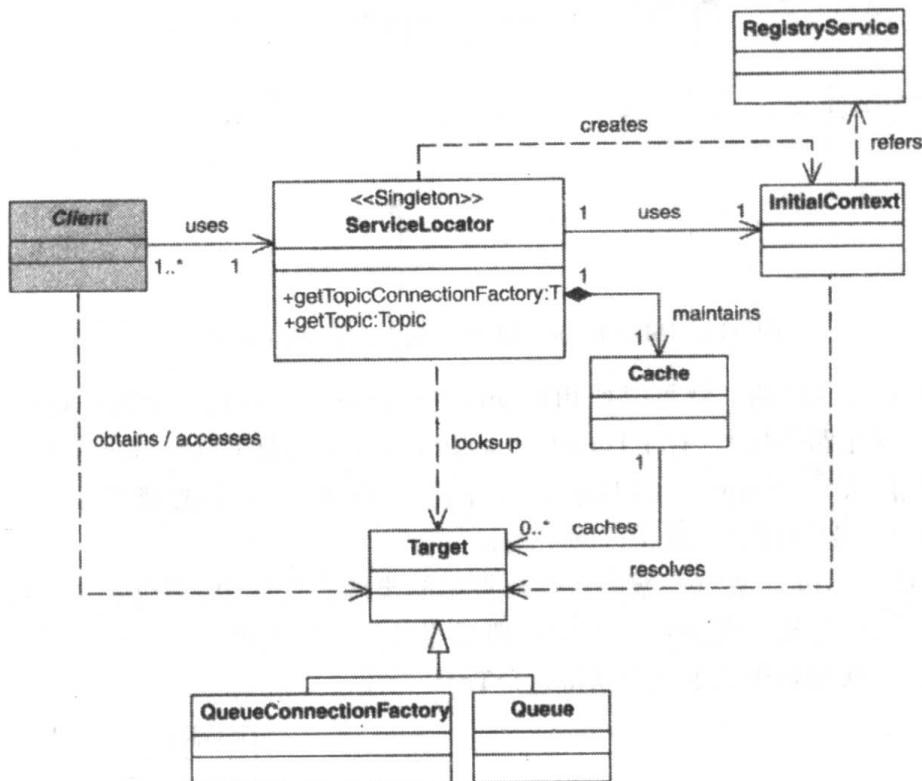


图7-11 JMS Queue（队列）服务定位器策略类图

图7-12体现了使用JMS Queue（队列）的点对点消息服务的服务定位器策略中，各个参与者之间的交互。

325

#### JMS Topic（主题）服务定位器策略<sup>⊖</sup>

在本策略中，TopicConnectionFactory（主题连接工厂）是JMS组件服务定位器的Target（目标）。TopicConnectionFactory的寻址是使用其JNDI名称完成的。ServiceLocator（服务定位器）可以缓存TopicConnectionFactory，以供将来使用。

这也就避免了当客户端再次需要TopicConnectionFactory（主题连接工厂）的时候重复执行JNDI寻址。ServiceLocator（服务定位器）可以把缓存的TopicConnectionFactory直接返回给客户端。

⊖ 本策略与上一策略“JMS Queue（队列）服务定位器策略”中的叙述结构几乎完全一致，只不过段落划分略有变化。读者也可以对比这两个部分，看出JMS中的发送/接收（Queue）和发布/订阅（Topic）两种机制的异同。

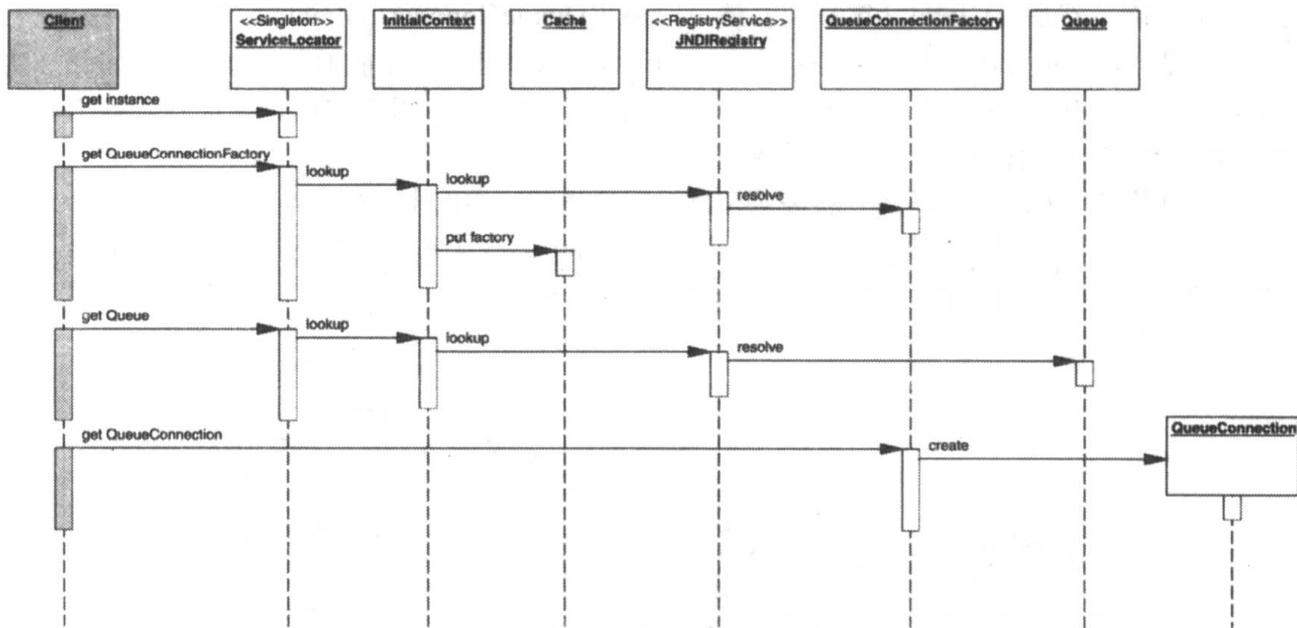


图7-12 JMS Queue（队列）服务定位器策略序列图

然后，Client（客户端）就可以使用TopicConnectionFactory（主题连接工厂）创建一个TopicConnection（主题连接）。有了TopicConnection，才能够获得TopicSession（主题会话）、创建Message（消息）、TopicPublisher（主题发布者，用于把消息发布到主题中）和TopicSubscriber（主题订阅者，用于订阅主题）。

图7-13是JMS Topic（主题）服务定位器策略的类图。在图中，Topic（主题）是一个JMS Destination（目标）对象，它注册在JNDI注册器服务中，代表着一个JMS主题。可以通过执行JNDI名称的寻址，从初始化上下文中获得这个Topic对象。

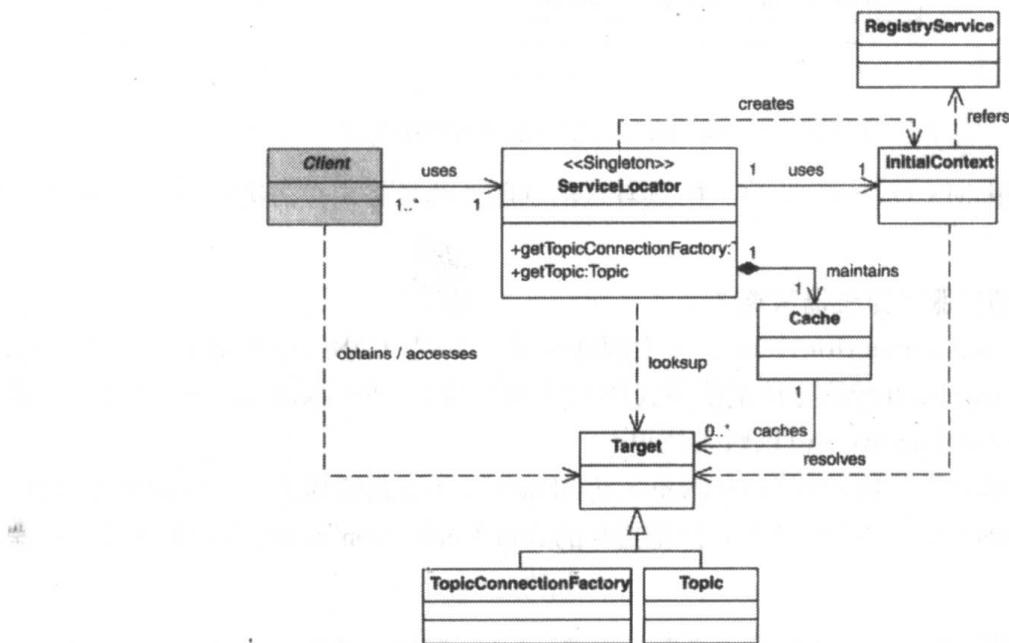


图7-13 JMS Topic（主题）服务定位器策略类图

图7-14体现了使用发布/订阅消息机制的JMS Topic（主题）定位服务器策略中的参与者交互。

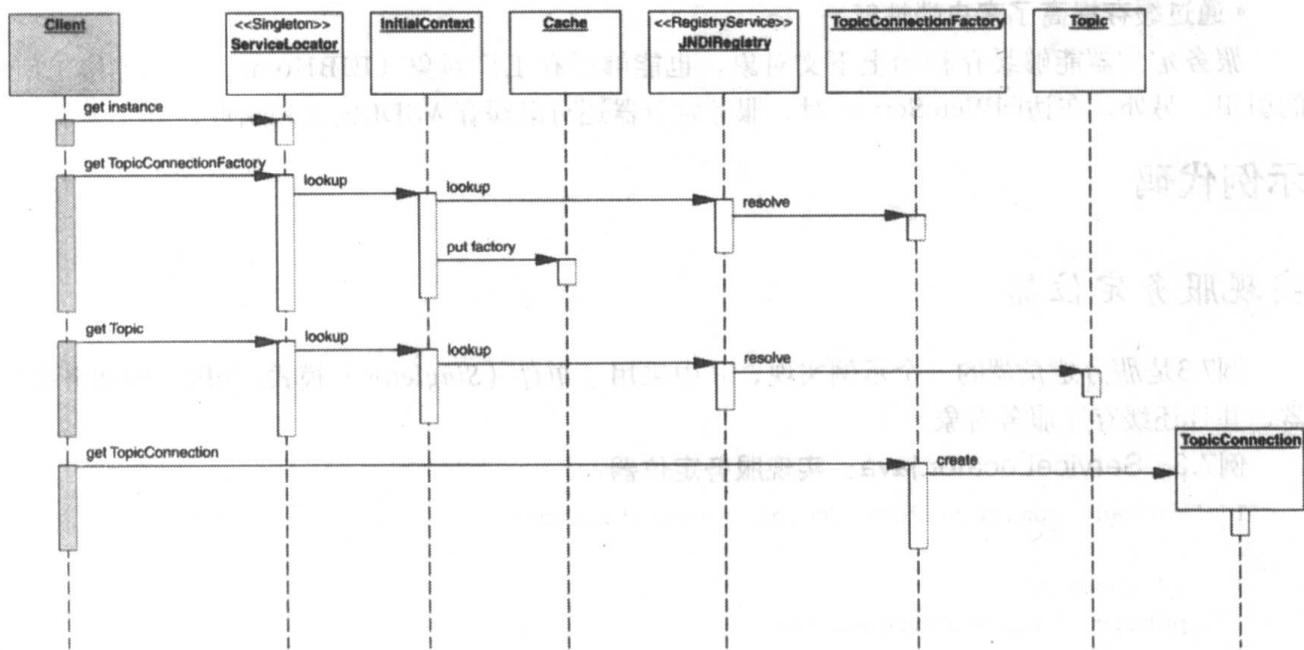


图7-14 JMS Topic（主题）服务定位器策略序列图

### Web Service定位器策略

如果应用系统需要定位一个通过UDDI注册表或ebXML注册表/注册库发布的Web Service，那么就可以使用Web Service定位器策略。Web Service定位器能够定位并获取一个到注册表/注册库的连接，还能提供其他方法，用于搜寻注册表/注册库中的各种不同服务。后面例7.9中的示例代码是本策略的一个示例实现。

## 效果

- 抽象了系统的复杂性

服务定位器封装了服务寻址、创建过程的复杂性（这在“问题部分”有介绍），并对客户端隐藏了这种复杂性。

- 为客户端提供了统一的服务访问方式

服务定位器提供了一种有用的、精确的接口，可供所有客户端使用。这个接口确保了所有类型的客户端都能以统一的方式访问业务对象（指导址和创建过程）。这种统一性减少了开发和维护的工作量。

- 更便于添加EJB业务组件

EJB客户端不用知道EJB Home对象，所以在后期还可以添加新开发、部署的EJB的EJB Home对象，不会影响到客户端。同样，JMS客户端也不用直接知道JMS连接工厂，所以也可以加入新的连接工厂，不会影响到客户端。

- 提高了系统的网络性能

客户端不涉及寻址和对象创建。既然由服务定位器执行这些工作，那么业务对象的寻址和

创建过程中的多次网络调用就可以合成在一起。

- **通过缓存提高了客户端性能**

329 服务定位器能够缓存初始上下文对象，也能够缓存工厂对象（EJBHome，JMS连接工厂）的引用。另外，在访问Web Service时，服务定位器还可以缓存WSDL定义和端点。

## 示例代码

### 实现服务定位器

例7.3是服务定位器的一个示例实现，其中采用了单件（*Singleton*）模式[GoF]实现服务定位器，并且还缓存了服务对象。

#### 例7.3 ServiceLocator.java：实现服务定位器

```

1  package com.corej2eepatterns.servicelocator;
2
3  // imports
4  public class ServiceLocator {
5
6      private InitialContext initialContext;
7      private Map cache;
8
9      private static ServiceLocator _instance;
10
11     static {
12         try {
13             _instance = new ServiceLocator();
14         } catch (ServiceLocatorException se) {
15             System.err.println(se);
16             se.printStackTrace(System.err);
17         }
18     }
19
20     private ServiceLocator() throws ServiceLocatorException {
21         try {
22             initialContext = new InitialContext();
23             cache = Collections.
24                 synchronizedMap(new HashMap());
25         } catch (NamingException ne) {
26             throw new ServiceLocatorException(ne);
27         } catch (Exception e) {
28             throw new ServiceLocatorException(e);
29         }
30     }
31
32     static public ServiceLocator getInstance() {

```

```

33         return _instance;
34     }
35
36     // 此处实现各个寻址方法
37
38 }

```

330

## 实现EJB服务定位器

例7.4是一个用EJB服务定位器策略，实现本地、远程EJBHome对象寻址的实例。另外，例7.5中的示例代码实现了一些辅助方法，用于获取字符串形式的EJB句柄，并且通过字符串句柄来重建对于以前使用过的EJB组件的连接。

### 例7.4 ServiceLocator.java：实现EJB服务定位器策略

```

1 package com.corej2eepatterns.servicelocator;
2
3 // imports
4 public class ServiceLocator {
5
6     // 按照给定的本地home对象的JNDI名称
7     // 为本地home寻址
8     public EJBLocalHome getLocalHome(String jndiHomeName)
9     throws ServiceLocatorException {
10        EJBLocalHome localHome = null;
11        try {
12            if (cache.containsKey(jndiHomeName)) {
13                localHome = (EJBLocalHome)
14                    cache.get(jndiHomeName);
15            } else {
16                localHome = (EJBLocalHome)
17                    initialContext.lookup(jndiHomeName);
18                cache.put(jndiHomeName, localHome);
19            }
20        } catch (NamingException nex) {
21            throw new ServiceLocatorException(nex);
22        } catch (Exception ex) {
23            throw new ServiceLocatorException(ex);
24        }
25        return localHome;
26    }
27
28    // 按照给定的远程home对象名称
29    // 为远程home对象寻址
30    public EJBHome getRemoteHome(
31        String jndiHomeName, Class homeClassName)
32    throws ServiceLocatorException {
33        EJBHome remoteHome = null;

```

331

```

34         try {
35             if (cache.containsKey(jndiHomeName)) {
36                 remoteHome =
37                     (EJBHome) cache.get(jndiHomeName);
38             } else {
39                 Object objref =
40                     initialContext.lookup(jndiHomeName);
41                 Object obj = PortableRemoteObject.
42                     narrow(objref, homeClassName);
43                 remoteHome = (EJBHome) obj;
44                 cache.put(jndiHomeName, remoteHome);
45             }
46         } catch (NamingException nex) {
47             throw new ServiceLocatorException(nex);
48         } catch (Exception ex) {
49             throw new ServiceLocatorException(ex);
50         }
51         return remoteHome;
52     }
53
54     . . .
55 }

```

### 例7.5 ServiceLocator.java: 实现EJB服务定位器策略——使用句柄

```

1 package com.corej2eepatterns.servicelocator;
2
3 // imports
4 public class ServiceLocator {
5     . . .
6
7     public EJBObject getService(String id)
8     throws ServiceLocatorException {
9         if (id == null) {
10             throw new ServiceLocatorException(
11                 "Invalid ID: Cannot create Handle");
12         }
13         try {
14             byte[] bytes = new String(id).getBytes();
15             InputStream io = new ByteArrayInputStream(bytes);
16             ObjectInputStream os = new ObjectInputStream(io);
17             javax.ejb.Handle handle =
18                 (javax.ejb.Handle) os.readObject();
19             return handle.getEJBObject();
20         } catch (Exception ex) {
21             throw new ServiceLocatorException(ex);
22         }
23     }
24 }

```

```

25     // 以串行化的格式, 返回给定的EJBObject
26     // 的字符串句柄。
27     public String getId(EJBObject session)
28     throws ServiceLocatorException {
29         String id=null;
30         try {
31             javax.ejb.Handle handle = session.getHandle();
32             ByteArrayOutputStream fo =
33                 new ByteArrayOutputStream();
34             ObjectOutputStream so =
35                 new ObjectOutputStream(fo);
36             so.writeObject(handle);
37             so.flush();
38             so.close();
39             id = new String(fo.toByteArray());
40         } catch (RemoteException rex) {
41             throw new ServiceLocatorException(rex);
42         } catch (IOException ioex) {
43             throw new ServiceLocatorException(ioex);
44         }
45
46         return id;
47     }
48     . .
49 }

```

333

## 实现JMS服务定位器

例7.6和例7.7是JMS服务定位器策略的实现，示范了如何获取Topic、TopicConnectionFactory（主题连接工厂）、Queue（队列）和QueueConnectionFactory（队列连接工厂）。

### 例7.6 ServiceLocator.java：实现JMS Topic（主题）服务定位器策略

```

1  package com.corej2eepatterns.servicelocator;
2
3  // imports
4  public class ServiceLocator {
5      . .
6
7      // 寻址并返回一个TopicConnectionFactory（主题连接工厂）
8      public TopicConnectionFactory getTopicConnectionFactory(
9          String topicConnectionFactoryName)
10     throws ServiceLocatorException {
11         TopicConnectionFactory topicFactory = null;
12         try {
13             if (cache.containsKey(topicConnectionFactoryName)) {
14                 topicFactory = (TopicConnectionFactory)
15                     cache.get(topicConnectionFactoryName);

```

```

16         } else {
17             topicFactory = (TopicConnectionFactory)
18                 initialContext.
19                     lookup(topicConnectionFactoryName);
20             cache.put(topicConnectionFactoryName,
21                     topicFactory);
22         }
23     } catch (NamingException nex) {
24         throw new ServiceLocatorException(nex);
25     } catch (Exception ex) {
26         throw new ServiceLocatorException(ex);
27     }
28     return topicFactory;
29 }
30
31 // 寻址并返回一个Topic (主题)
32 public Topic getTopic(String topicName)
33 throws ServiceLocatorException {
34     Topic topic = null;
35     try {
36         if (cache.containsKey(topicName)) {
37             topic = (Topic) cache.get(topicName);
38         } else {
39             topic =
40                 (Topic)initialContext.lookup(topicName);
41             cache.put(topicName, topic);
42         }
43     } catch (NamingException nex) {
44         throw new ServiceLocatorException(nex);
45     } catch (Exception ex) {
46         throw new ServiceLocatorException(ex);
47     }
48     return topic;
49 }
50
51     . . .
52 }

```

### 例7.7 ServiceLocator.java：实现JMS Queue（队列）服务定位策略

```

1 package com.corej2eepatterns.servicelocator;
2
3 // imports
4 public class ServiceLocator {
5     . . .
6
7     public QueueConnectionFactory getQueueConnectionFactory(
8         String queueConnectionFactoryName)
9     throws ServiceLocatorException {

```

334

```

10     QueueConnectionFactory queueFactory = null;
11     try {
12         if (cache.containsKey(queueConnectionFactoryName)) {
13             queueFactory = (QueueConnectionFactory)
14                 cache.get(queueConnectionFactoryName);
15         } else {
16             queueFactory = (QueueConnectionFactory)
17                 initialContext.lookup(
18                     queueConnectionFactoryName);
19             cache.put(queueConnectionFactoryName,
20                     queueFactory);
21         }
22     } catch (NamingException nex) {
23         throw new ServiceLocatorException(nex);
24     } catch (Exception ex) {
25         throw new ServiceLocatorException(ex);
26     }
27
28     return queueFactory;
29 }
30
31     public Queue getQueue(String queueName)
32     throws ServiceLocatorException {
33         Queue queue = null;
34         try {
35             if (cache.containsKey(queueName)) {
36                 queue = (Queue) cache.get(queueName);
37             } else {
38                 queue =
39                     (Queue) initialContext.lookup(queueName);
40                 cache.put(queueName, queue);
41             }
42         } catch (NamingException nex) {
43             throw new ServiceLocatorException(nex);
44         } catch (Exception ex) {
45             throw new ServiceLocatorException(ex);
46         }
47
48         return queue;
49     }
50     . . .
51 }

```

335

## 实现JDBC数据源服务定位器

例7.8是JDBC数据源服务定位器策略的一个示例实现。

**例7.8 ServiceLocator.java：实现JDBC数据源服务定位器策略**

```

1 package com.corej2eepatterns.servicelocator;
2
3 // imports
4 public class ServiceLocator {
5     . . .
6
7     public DataSource getDataSource(String dataSourceName)
8     throws ServiceLocatorException {
9         DataSource dataSource = null;
10        try {
11            if (cache.containsKey(dataSourceName)) {
12                dataSource =
13                    (DataSource) cache.get(dataSourceName);
14            } else {
15                dataSource = (DataSource)
16                    initialContext.lookup(dataSourceName);
17                cache.put(dataSourceName, dataSource );
18            }
19        } catch (NamingException nex) {
20            throw new ServiceLocatorException(nex);
21        } catch (Exception ex) {
22            throw new ServiceLocatorException(ex);
23        }
24        return dataSource;
25    }
26    . . .
27 }

```

336

**实现Web Service定位器**

例7.9是一个Web Service定位器策略的示例实现。

**例7.9 ServiceLocator.java：实现Web Service定位器策略**

```

1 package com.corej2eepatterns.servicelocator;
2 // imports
3 public class ServiceLocator {
4     . . .
5
6     // 返回一个到UDDI注册表的连接
7     public Connection getRegistryConnection(String registryURL)
8     {
9         javax.xml.registry.Connection
10        registryConnection = null;
11        try {
12            if (cache.containsKey(registryURL)) {
13                registryConnection =

```

337

```

14         (Connection) cache.get(registryURL);
15     } else {
16         // 首先设置标准的JAXR属性
17         // 比如 (javax.xml.registry.queryManagerURL)
18         // 再创建一个到注册表的连接
19
20         javax.xml.registry.ConnectionFactory factory =
21             ConnectionFactory.newInstance();
22
23         // 定义连接配置属性
24         Properties props = new Properties();
25         props.setProperty(
26             "javax.xml.registry.queryManagerURL",
27             registryURL);
28         props.setProperty(
29             "javax.xml.registry.factoryClass",
30             "com.sun.xml.registry.uddi.ConnectionFactoryImpl");
31         factory.setProperties(props);
32
33         registryConnection = factory.createConnection();
34         cache.put(registryURL, registryConnection);
35     }
36     } catch (Exception e) {
37         throw new ServiceLocatorException(e);
38     } catch (JAXRException je) {
39         throw new ServiceLocatorException(je);
40     }
41     return registryConnection;
42 }
43
44 // 按照一个已发布服务的tModelKey , 为一个服务URI端点
45 // 搜索注册表。
46 // registryURL - 是UDDI 提供者 (provider) 的URL
47 // tModelKey - 是特定的已发布服务的
48 //      搜索字符串 (search string)
49 public String getServiceAccessURI(
50     String registryURL, String tModelKey) {
51
52     javax.xml.registry.Connection regCon = null;
53     javax.xml.registry.RegistryService regSvc = null;
54     javax.xml.registry.BusinessQueryManager
55         queryManager = null;
56     javax.xml.registry.BusinessLifeCycleManager
57         lifeCycleManager = null;
58     String serviceAccessURI = null;
59     try {
60         if (cache.containsKey(tModelKey)) {

```

338

```

61         serviceAccessURI = (String) cache.get(tModelKey);
62     } else {
63         regCon = getRegistryConnection(registryURL);
64         regSvc = regCon.getRegistryService();
65         queryManager = regSvc.getBusinessQueryManager();
66         javax.xml.registry.infomodel.RegistryObject
67             regobj = businessQueryManager.
68                 getRegistryObject(tModelKey,
69                     BusinessLifeCycleManager.CONCEPT);
70         Collection coll = new ArrayList();
71         coll.add(regobj);
72
73         //寻找机构
74         BulkResponse results =
75             businessQueryManager.findOrganizations(
76                 null, null, null, coll, null, null);
77
78         Collection co = results.getCollection();
79         Iterator it = co.iterator();
80
81         while (it.hasNext()) {
82             Organization org = (Organization)it.next();
83             String orgName = org.getName().getValue();
84             Collection cc = org.getServices();
85             for (Iterator iterator=cc.iterator();
86                  iterator.hasNext();) {
87                 Service service = (Service) iterator.next();
88                 Collection cb = service.getServiceBindings();
89                 for (Iterator ito = cb.iterator();
90                      ito.hasNext();) {
91                     ServiceBinding serviceBinding =
92                         (ServiceBinding) ito.next();
93                     if (serviceBinding != null ||
94                         serviceBinding.getAccessURI() != null) {
95                         serviceBinding.getAccessURI();
96                         serviceAccessURI =
97                             serviceBinding.getAccessURI();
98                         cache.put(tModelKey,
99                             serviceBinding.getAccessURI());
100                    }
101                }
102            }
103        }
104    }
105    } catch (JAXRException e) {
106        throw new ServiceLocatorException(e);
107    }

```

```

108
109     return serviceAccessURI;
110 }
111 .
112 .
113 }
114

```

## 相关模式

- **业务代表**

业务代表使用服务定位器，定位并获取业务服务对象的引用，其中包括EJB对象、JMS主题以及JMS队列等。这样就能从业务代表中分离出服务定位逻辑，促进松耦合，提高可维护性。

- **会话门面**

会话门面使用服务定位器，定位并获取session bean和entity bean的home和远程引用，另外也使用服务定位器定位数据源。

- **传输对象组装器**

传输对象组装器使用服务定位器来定位对session bean和entity bean的引用，它需要通过这些EJB访问数据，构造复合传输对象。

- **数据访问对象**

数据访问对象使用服务定位器为数据源寻址，并获得一个到该数据源的引用。

340

## 会话门面

### 问题

需要把业务组件和业务服务暴露给远程客户端。

之所以使用会话门面，是出于两个考虑：一是要控制客户端对业务对象的访问，二是要降低远程客户端和细粒度的业务组件、业务服务交互造成的网络负载。

**客户端对业务组件的访问** 通常，多层的J2EE应用系统有一些服务器端组件，这些组件可能以业务对象、POJO或entity bean的形式实现。但是，如果暴露出这些组件，让客户端直接访问它们，就可能导致以下问题：

- 在客户端和业务组件之间存在了紧耦合，这也就导致了两个层次之间存在直接的依赖关系，如果修改了业务组件的接口，这也会直接影响客户端。
- 直接让客户端访问业务组件，也会要求客户端包含复杂的逻辑，因为这样客户端才能协调多个业务组件、与它们交互，并且这也需要客户端能够“感知”这些业务组件之间可能存在的复杂关系。这样一来客户端就要包含一些复杂逻辑，完成寻址、事务划界、安全管理等功能，而且还要执行业务处理。从而加大了客户端的复杂度和责任。
- 如果有多种不同类型的客户端，那么让客户端直接访问业务组件，就会导致对通用业务组

件的用法不一致；因为每一种客户端都包含独立于其他种类客户端的交互逻辑。这可能也会导致不同客户端之间的代码重复，所以降低了系统实现的可维护性和灵活性。

**341 远程客户端访问细粒度的组件** 应用系统可能包含细粒度的业务层组件和服务，而这些组件/服务却要由远程客户端访问。如果客户端直接访问业务组件，那么它们可能会对多个细粒度组件多次执行远程访问。这会让应用系统变得非常“罗嗦”<sup>Θ</sup>，大量的远程网络调用会降低系统的网络性能。

## 约束

- 需要避免客户端直接访问业务层组件，从而防止客户端和业务层之间出现紧耦合。
- 需要给业务对象和其他业务层组件提供一个远程访问层。
- 需要聚合应用服务和其他服务，并把它们暴露给远程客户端。
- 需要把所有需要暴露给远程客户端的业务逻辑集中、聚合起来。
- 需要隐藏业务组件和业务服务之间复杂的交互和依赖关系，从而提升系统的可维护性，集中逻辑处理，提高灵活性，增强应对变化的能力。

## 解决方案

**使用会话界面，封装业务层组件，对远程客户端暴露粗粒度服务。**客户端不用直接访问业务组件，而是访问会话界面。

会话界面由session bean实现，负责与业务组件（比如业务对象、应用服务等）的交互。会话界面提供了一个远程服务层，它只暴露出客户端需要使用的接口。

当会话界面中包含的业务逻辑很少、甚至一点儿也不包含的时候，它也最为有效。如果存在业务逻辑的话，就应该放在应用服务里，而会话界面负责调用应用服务。

### 设计手记：用例和会话界面

那么如何通过考察用例来确定会话界面呢？如果把每个用例都映射为一个会话界面，最后就会产生太多的门面。这也违背了“使用少量粗粒度的session bean”的设计意图，所以我们不推荐这种做法。

**342** 应该在给过程建模、设计服务的过程中，一边确定用例一边就确定相关的服务。然后，就要组合、分隔这些服务，以便把它们实现为粗粒度的会话界面。或者也可以把这些服务设计、实现为一个“应用服务”层，然后再通过一个“会话界面”层把这个“应用服务”层暴露给远程客户端。通过这种方法，把相关服务组合在一起，就能让应用系统中只存在较少的会话界面。

举例来说，考虑一个银行应用系统，可以把所有管理账户的交互操作组合在同一个会话界面里。这也就包括所有管理账户的用例，比如创建新账户、修改账户信息、查看账户信息等等，所有这些用例都要与“账户（Account）”业务对象交互。

<sup>Θ</sup> “罗嗦”，chatty，指包含大量的网络交互。

## 结构

图7-15是会话门面模式的类图。

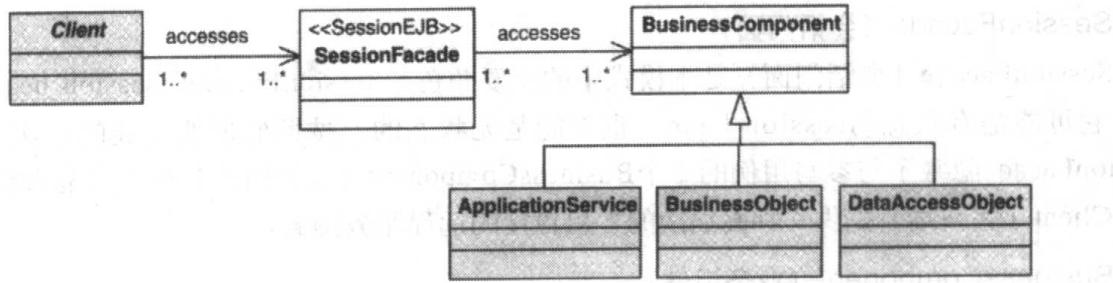


图7-15 会话门面模式类图

## 参与者和合作<sup>Θ</sup>

图7-16是会话门面模式的序列图，其中，会话门面要和entity bean业务对象以及一个POJO应用服务交互，这些参与者共同完成了来自客户端的请求。

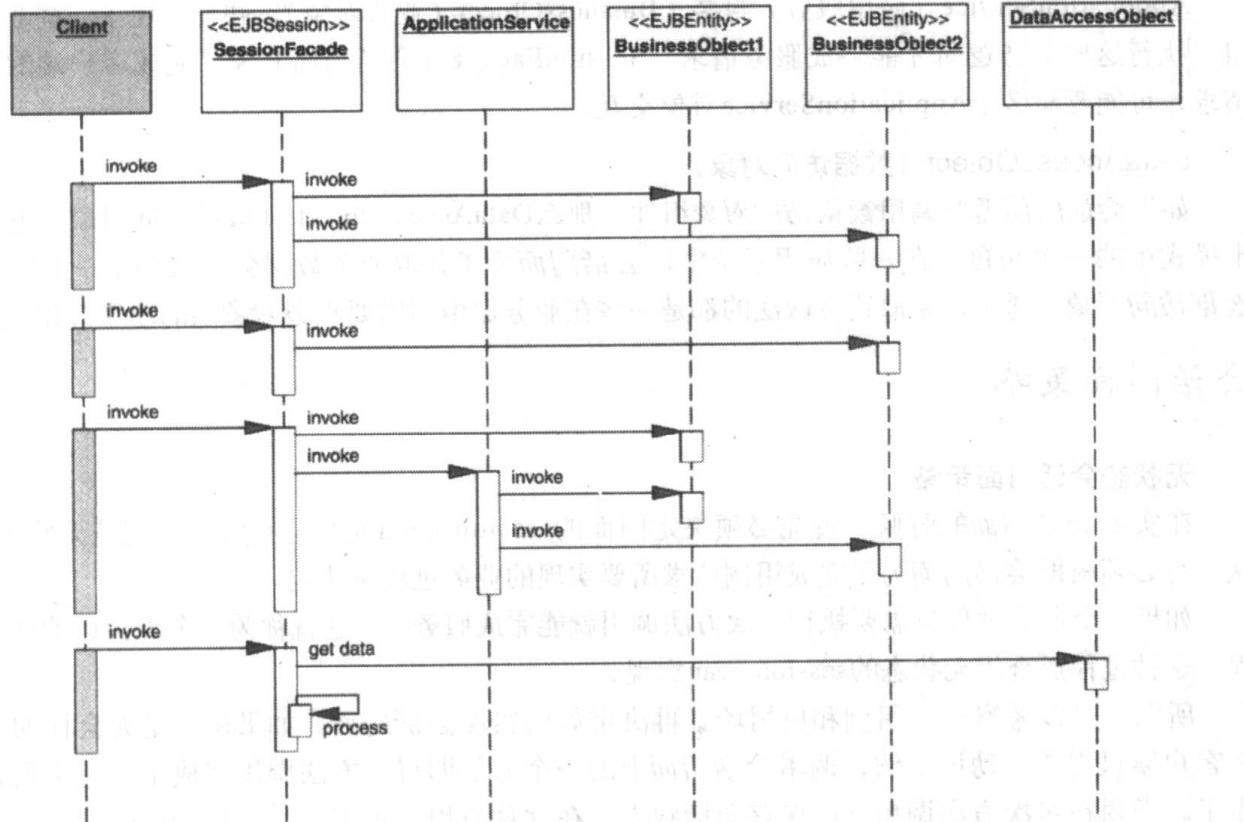


图7-16 会话门面模式序列图

<sup>Θ</sup> 与其他章节比较，本章可能出自另一个作者的手笔，所以整个模式的模板略有不同。其他模式中此处都作“参与者和责任”。另外“策略”一节的标题中加入了模式名称“会话门面”，也是一个特别的地方。

### Client (客户端)

Client (客户端) 也就是会话门面的客户端，它需要访问业务服务。Client通常是另一个层次中的业务代表。

### SessionFacade (会话门面)

SessionFacade (会话门面) 是本模式中的主要角色。SessionFacade以session bean的形式实现，它可能是有状态的session bean，也可能是无状态的，按照它所要完成的用例需求而定。SessionFacade隐藏了与参与用例的多个BusinessComponent (业务组件) 交互的复杂性，因此就能给Client (客户端) 提供一种高层次的、粗粒度的远程服务抽象。

### 343 BusinessComponent (业务组件)

BusinessComponent (业务组件) 参与完成客户端的请求。BusinessComponent可能实现为业务对象（其中包含了业务数据和业务功能），或者也可能实现为应用服务。比如，在序列图中，业务对象参与者就被实现entity beans，图中称为BusinessObject1 和BusinessObject2。图中的ApplicationService (应用服务) 参与者则是用POJO实现的。

### ApplicationService (应用服务)

ApplicationService (应用服务) 封装了BusinessObject (业务对象)，并且实现了一些业务逻辑，执行这些业务逻辑才能完成服务请求。SessionFacade (会话门面) 为了完成客户端的一次请求，可能要与多个ApplicationService对象交互。

### 344 DataAccessObject (数据访问对象)

如果会话门面需要调用数据访问对象组件，那么DataAccessObject (数据访问对象) 就也是本模式中的一个角色。在一些应用系统中，会话门面为了获取业务数据会直接调用一个以上的数据访问对象。通常，采取这种做法的都是一些在业务层中不需要业务对象层的小型应用系统。

## 会话门面策略

### 无状态会话门面策略

在实现会话门面的时候，首先必须决定门面的session bean究竟是有状态的还是无状态的。这一点必须根据会话门面为了完成用例请求所要实现的业务过程来决定。

如果一个业务过程只需要执行一次方法调用就能完成服务<sup>Θ</sup>，这就称为一个无会话的业务过程。这种过程适合用无状态的session bean实现。

所以，可以考察一下用例和应用场景再决定如何实现会话门面。如果用例是无会话的，那么客户端只需要启动该用例，调用会话门面上的一个方法即可。方法调用完成了，用例也就完成了。无须在多次方法调用之间保存会话状态。在这种应用场景下，会话门面可以用无状态的session bean实现。

---

<sup>Θ</sup> 这里所谓“一次方法调用”是指客户端只调用一次服务器端业务方法，而不是指业务层内部的方法调用也只有一次。

### 有状态会话门面策略

如果一个业务过程需要多次方法调用才能完成服务，这就称为一个有会话的业务过程。在客户端的多次方法调用之间，必须保存会话状态。在这种应用场景下，使用有状态的session bean实现会话门面可能更为适宜。

## 效果

- 引入了一个层次，专门为远程客户端提供服务

会话门面在客户端和业务层之间又引入了一个层次，提供粗粒度的远程服务。对于一些应用系统来说，这可能是一种多余的负载，尤其是在业务层并没有使用EJB组件实现的情况下。但是，对于J2EE应用系统来说，会话门面几乎是必不可少的，因为J2EE应用系统大多要提供远程服务，而且要利用EJB容器的一些优势，比如事务、安全性、生命周期管理等。

345

- 暴露统一的粗粒度接口

会话门面封装了底层的业务组件交互的复杂性，给客户端提供了一个简单的粗粒度的服务层接口，用于访问系统，这个接口易于理解、易于使用。而且可以给每个会话门面配备一个业务代表，这样客户端开发者就能够更容易地利用上会话门面的优势了。

- 减少了各层次之间的耦合

使用会话门面，能够消除业务组件和客户端之间的耦合，减少表现层和业务层之间的紧耦合和依赖。还可以另外实现应用服务，封装那些要调用多个业务对象的复杂业务逻辑，而会话门面则可以把业务逻辑委派给应用服务，由后者实现。

- 明确的层次划分，增进了灵活性和可维护性

结合使用会话门面和应用服务，就能给各种交互划分层次，并把它们集中起来，从而提高了系统的灵活性。这也减少了系统中的耦合，从而提高了系统应对修改的能力。虽然对业务逻辑的修改可能仍然会导致应用服务、甚至会话门面的修改，但是由于分层清晰了，所以更容易维护这些修改。

- 降低了复杂度

使用应用服务能够降低会话门面的复杂度。使用业务代表来访问会话门面能降低客户端代码的复杂度。这也有助于使系统变得更容易维护、更为灵活。

- 提高了性能，减少了细粒度的远程方法

会话门面还能够提高系统的性能，因为它把多种细粒度的交互聚合成一个粗粒度的方法，从而减少了远程网络调用。而且，会话门面通常也和参与操作的业务组件处于同一个进程空间中，这也是二者之间的通信更为快捷。

- 集中了安全管理

可以在会话门面的层面上管理应用系统的安全策略，因为该层面正是系统呈现给客户端的部分。因为会话门面提供粗粒度的访问，所以在这个层面定义安全策略就更为容易，也更易于管理，比起针对每个参与处理的细粒度业务组件实现安全策略要好得多。

346

- 集中了事务控制

会话门面是业务层服务的一个粗粒度的远程访问点，因此就更易于在会话门面层集中应用

事务控制。在会话门面上，可以按照一种粗粒度的形式集中管理、定义事务控制。与在细粒度业务对象上管理事务或在客户端管理事务相比，这种做法要容易的多。

#### • 减少了暴露给客户端的远程接口

会话门面给业务组件提供了一种粗粒度的访问机制，极大地减少了暴露给客户端的业务组件数量。这也就减少了系统的负载，因为与客户端直接访问单个业务组件的情况相比，让客户端通过会话门面访问服务，交互次数比较少。

## 示例代码

### 实现会话门面

考虑一个专业服务应用系统（PSA）<sup>⑨</sup>。与entity bean（比如Project/项目、Resource/资源）相关的工作流<sup>⑩</sup>被封装在ProjectResourceManagerSession（项目资源管理器会话）中，而后者则是通过会话门面模式实现的。例7.10体现了Resource和Project entity bean的交互，另外也包括与其他业务组件（比如值列表处理器、传输对象组装器）的交互。

#### 例7.10 实现会话门面——Session Bean

```

1  package corepatterns.apps.psa.ejb;
2
3
4  import java.util.*;
5  import java.rmi.RemoteException;
6  import javax.ejb.*;
7  import javax.naming.*;
8  import corepatterns.apps.psa.core.*;
9  import corepatterns.util.ServiceLocator;
10 import corepatterns.util.ServiceLocatorException;
11
12 // 注意：为了简洁起见，省略了异常处理的try/catch细节。
13
14 public class ProjectResourceManagerSession
15     implements SessionBean {
16
17     private SessionContext context;
18
19     // 门面封装了entity beans的
20     // 远程引用
21     private Resource resourceEntity = null;
22     private Project projectEntity = null;
23     ...
24

```

347

<sup>⑨</sup> 专业服务应用系统是全书的一个常见例子。首次出现是在本章的“业务代表”一节的示例代码中。

<sup>⑩</sup> 这里所谓工作流，更多地是指一般的处理逻辑和交互。

```
25 // 默认的ejbCreate方法
26 public void ejbCreate() throws CreateException { }
27
28 // 本方法创建门面，并且使用
29 // 主键值建立与相关entity beans
30 // 的连接
31 public void ejbCreate(String resourceId, String projectId,
32     ...)
33 throws CreateException, ResourceException {
34
35     try {
36         // 定位并连接到entity bean
37         connectToEntities(resourceId, projectId, ...);
38     } catch (...) {
39         // 处理异常
40     }
41 }
42
43 // 本方法用主键值把
44 // 会话门面连接到相关的entity bean
45 private void connectToEntities (String resourceId,
46     String projectId)
47 throws ResourceException {
48     resourceEntity = getResourceEntity(resourceId);
49     projectEntity = getProjectEntity(projectId);
50     ...
51 }
52
53 // 本方法用主键值把会话门面
54 // 重新连接到另一组entity bean上
55 public resetEntities(String resourceId, String projectId,
56     ...)
57 throws PSAException {
58
59     connectToEntities(resourceId, projectId, ...);
60 }
61
62 // 私有方法，用来获取Resource entity bean的Home
63 private ResourceHome getResourceHome()
64 throws ServiceLocatorException {
65     return ServiceLocator.getInstance().getLocalHome(
66         "ResourceEntity", ResourceHome.class);
67 }
68
69 // 私有方法，用来获取Project entity bean的Home
70 private ProjectHome getProjectHome()
71 throws ServiceLocatorException {
```

```

72     return ServiceLocator.getInstance().getLocalHome(
73         "ProjectEntity", ProjectHome.class);
74     }
75
76     // 私有方法, 用来获取Resource entity bean
77     private Resource getResourceEntity(String resourceId)
78     throws ResourceException {
79         try {
80             ResourceHome home = getResourceHome();
81             return (Resource) home.findByPrimaryKey(resourceId);
82         } catch (...) {
83             // 处理异常
84         }
85     }
86
87     // 私有方法, 用来获取Project entity bean
88     private Project getProjectEntity(String projectId)
89     throws ProjectException {
90         // 与getResourceEntity类似
91         ...
92     }
93
94     // 本方法封装了把一个资源分配给
95     // 一个项目的工作流。它要与Project和
96     // Resource Entity bean交互
97     public void assignResourceToProject(int numHours)
98     throws PSAException {
99
100        try {
101            if ((projectEntity == null) ||
102                (resourceEntity == null)) {
103                // SessionFacade (会话门面) 无法连接到entity bean
104                throw new PSAException(...);
105            }
106
107            // 获取Resource (资源) 数据
108            ResourceTO resourceTO =
109                resourceEntity.getResourceData();
110
111            // 获取Project (项目) 数据
112            ProjectTO projectTO = projectEntity.getProjectData();
113
114            // 首先, 把Resource加入Project
115            projectEntity.addResource(resourceTO);
116
117            // 给Project添加新Commitment (事项)
118            CommitmentTO commitment = new CommitmentTO(...);

```

```
119
120      // 把事项加入Project (项目)
121      projectEntity.addCommitment(commitment);
122
123  } catch (...) {
124      // 处理异常
125  }
126 }
127
128 // 类似地实现其他业务方法
129 // 完成各种用例/交互
130 public void unassignResourceFromProject()
131 throws PSAException {
132     ...
133 }
134
135 // 以下是与ResourceEntity (资源entity bean) 相关的方法
136 public ResourceTO getResourceData()
137 throws ResourceException {
138     ...
139 }
140
141 // 更新Resource Entity Bean
142 public void setResourceData(ResourceTO resource)
143 throws ResourceException {
144     ...
145 }
146
147 // 创建新的Resource Entity bean
148 public ResourceTO createNewResource(ResourceTO resource)
149 throws ResourceException {
150     ...
151 }
152
153 // 以下方法管理资源的停用时间
154 public void addBlockoutTime(Collection blockoutTime)
155 throws RemoteException, BlockoutTimeException {
156     ...
157 }
158
159 public void updateBlockoutTime(Collection blockoutTime)
160 throws RemoteException, BlockoutTimeException {
161     ...
162 }
163
164 public Collection getResourceCommitments()
165 throws RemoteException, ResourceException {
```

350

```
166     ...
167 }
168
169 // 以下方法与ProjectEntity (项目Entity bean) 相关
170 public ProjectTO getProjectData()
171 throws ProjectException {
172     ...
173 }
174
175 // 更新Project Entity Bean
176 public void setProjectData(ProjectTO project)
177 throws ProjectException {
178     ...
179 }
180
181 // 创建新的Project Entity bean
182 public ProjectTO createNewProject(ProjectTO project)
183 throws ProjectException {
184     ...
185 }
186
187 ...
188
189 // 以下是会话门面的其他方法的示例
190
191 // 这个方法代理对传输对象组装器的调用,
192 // 返回一个复合传输对象。
193 // 参见传输对象组装器模式。
194 public ProjectCTO getProjectDetailsData()
195 throws PSAException {
196     try {
197         ProjectTOAHome projectTOAHome = (ProjectTOAHome)
198             ServiceLocator.getInstance().getRemoteHome(
199                 "ProjectTOA", ProjectTOAHome.class);
200
201         // 传输对象组装器session bean
202         ProjectTOA projectTOA = projectTOAHome.create(...);
203         return projectTOA.getData(...);
204     } catch (...) {
205
206         // 处理/抛出异常
207     }
208 }
209
210 // 这个方法代理对ValueListHandler (值列表处理器) 的调用,
211 // 返回项目的列表。参见值列表处理器模式。
212 public Collection getProjectsList(Date start, Date end)
```

351

```

213     throws PSAException {
214         try {
215             ProjectListHandlerHome projectVLHHome =
216                 (ProjectVLHHome)
217                 ServiceLocator.getInstance().getRemoteHome(
218                     "ProjectListHandler", ProjectVLHHome.class);
219
220             // 值列表处理器session bean
221             ProjectListHandler projectListHandler =
222                 projectVLHHome.create();
223             return projectListHandler.getProjects(start, end);
224         } catch (...) {
225             // 处理/抛出异常
226         }
227     }
228
229     ...
230
231     public void ejbActivate() {
232         ...
233     }
234
235     public void ejbPassivate() {
236         context = null;
237     }
238
239     public void setSessionContext(SessionContext ctx) {
240         this.context = ctx;
241     }
242
243     public void ejbRemove() {
244         ...
245     }
246 }

```

[352]

会话门面的远程接口代码如例7.11所示。

### 例7.11 实现会话门面——远程接口

```

1
2 package corepatterns.apps.psa.ejb;
3
4 import java.rmi.RemoteException;
5 import javax.ejb.*;
6 import corepatterns.apps.psa.core.*;
7
8 // 注意：为了简洁起见，省略了所有try/catch的处理细节。
9
10 public interface ProjectResourceManager extends EJBObject {

```

```
11 .
12 public resetEntities(String resourceId, String projectId,
13     ...)
14 throws RemoteException, ResourceException ;
15
16 public void assignResourceToProject(int numHours)
17 throws RemoteException, ResourceException ;
18
19 public void unassignResourceFromProject()
20 throws RemoteException, ResourceException ;
21
22 ...
23
24 public ResourceTO getResourceData()
25 throws RemoteException, ResourceException ;
26
27 public void setResourceData(ResourceTO resource)
28 throws RemoteException, ResourceException ;
29
30 public ResourceTO createNewResource(ResourceTO resource)
31 throws ResourceException ;
32
33 public void addBlockoutTime(Collection blockoutTime)
34 throws RemoteException, BlockoutTimeException ;
35
36 public void updateBlockoutTime(Collection blockoutTime)
37 throws RemoteException, BlockoutTimeException ;
38
39 public Collection getResourceCommitments()
40 throws RemoteException, ResourceException;
41
42 public ProjectTO getProjectData()
43 throws RemoteException, ProjectException ;
44
45 public void setProjectData(ProjectTO project)
46 throws RemoteException, ProjectException ;
47
48 public ProjectTO createNewProject(ProjectTO project)
49 throws RemoteException, ProjectException ;
50
51 ...
52
53 public ProjectCTO getProjectDetailsData()
54 throws RemoteException, PSAException ;
55
56 public Collection getProjectsList(Date start, Date end)
57 throws RemoteException, PSAException ;
```

```

58
59 ...
60 }

```

354

会话门面的Home接口如例7.12所示。

### 例7.12 实现会话门面——Home接口

```

1 package corepatterns.apps.psa.ejb;
2
3
4 import javax.ejb.EJBHome;
5 import java.rmi.RemoteException;
6 import corepatterns.apps.psa.core.ResourceException;
7 import javax.ejb.*;
8
9 public interface ProjectResourceManagerHome
10 extends EJBHome {
11
12     public ProjectResourceManager create()
13     throws RemoteException, CreateException;
14
15     public ProjectResourceManager create(String resourceId,
16             String projectId, ...)
17     throws RemoteException, CreateException;
18 }

```

## 相关模式

- **业务代表**

业务代表在客户端提供了对会话门面的抽象。业务代表把客户端的请求代理给专门提供特定服务的会话门面（其间可能还会对请求加以调整）。

- **业务对象**

业务对象会参与用例的请求处理，会话门面封装了这些复杂的交互过程。

- **应用服务**

在一些应用系统中，使用应用服务封装复杂的业务逻辑和业务规则。在这种应用系统中，会话门面的实现就要简单一些，因为它主要是把请求委派给应用服务和业务对象。

355

- **数据访问对象**

会话门面有时候可能会直接访问数据访问对象以获取并存储数据。一些不使用业务对象的简单应用系统往往采取这种做法。这时，会话门面就封装了少量的业务逻辑，并使用数据访问对象来实现数据持久化。

- **服务定位器**

会话门面可能使用服务定位器来实现对其他业务组件（比如entity bean和session bean）的寻址。这能够降低门面中的代码复杂度，发挥服务定位器模式的优势。

- **中转 [POSA1]**

会话门面能够消除业务对象以及细粒度的服务与客户端之间的耦合，起到了中转 (*Broker*) 的作用。

356

- **门面 [GoF]**

会话门面模式是基于GoF的门面 (*Façade*) 模式设计的。

## 应用服务

### 问题

需要把多个业务层组件和服务之间的业务逻辑集中起来。

服务门面（例如会话门面或POJO门面）中包括很少、甚至根本不包括业务逻辑，只是提供一个简单的、粗粒度的接口而已。而对应于一组相互关联的业务操作，存在一些功能，相互之间具有内聚力，业务对象模式就封装了这样一些功能，并且也为它们提供了一个接口。

为了实现用例，应用系统会调度使用多个业务对象和业务服务。但是不能在业务对象内部完成这种调度多个对象和服务、实现用例的操作，因为这样会增加业务对象之间的耦合，降低它们之间的内聚性。同样，也不能把这种负责“调度”的业务逻辑放在服务门面里，因为这种业务逻辑可能会在多个门面中造成代码重复，这也就降低了通用代码的可重用性和可维护性。

对于那些不使用EJB组件的J2EE应用系统，业务层组件（比如业务对象）和其他服务是以POJO形式实现的。虽然这些对象都是本地对象，但也要避免直接向客户端暴露这些对象，因为这样的暴露会导致在客户端和业务层组件之间产生耦合和依赖。即使应用系统中根本不使用业务对象，还是希望在业务层中封装业务逻辑，而不是把这种逻辑放在门面或者客户端中。

### 约束

- 需要尽量减少服务门面中的业务逻辑。
- 需要让某种业务逻辑调度多个业务对象或业务服务。
- 需要在现存的业务层组件和服务之上提供一种粗粒度的服务API。
- 需要在业务对象之外封装专门针对特定用例的业务逻辑。

357

## 解决方案

**使用应用服务，集中、聚合特定的功能，提供一个统一的服务层。**

需要有一些业务逻辑封装业务对象和业务服务，而应用服务就集中实现了这样一种业务逻辑。采用这种做法，在业务对象之外实现业务逻辑，能够减少业务对象之间的耦合。使用应用服务能够把抽象层次更高的业务逻辑封装在一个独立的组件里，由该组件调用底层的业务对象和业务服务。

即使在应用系统中没有使用业务对象，也可以用应用服务来提供一个集中的业务逻辑实现层。这种情况下，应用服务可以包括系统中所有需要实现服务的过程式业务逻辑，在需要处理持久化数据的时候，还可以调用数据访问对象。

在非EJB应用系统中，如果需要减少表现层组件与业务层组件（比如业务对象和其他服务）之间的耦合，应用服务就能在这两个层次之间提供中间协调功能。与服务门面相比，应用服务提供的接口是细粒度的，而与底层的业务对象和其他服务相比，它的接口则是粗粒度的。

应用服务为服务门面（见本部分后面的“设计手记：服务门面”）提供了后台的底层架构。使用了该模式，门面实现起来就更简单了，门面包含的代码也更少了，因为它可以把业务处理委派给应用服务。应用服务包含业务逻辑，而服务门面通常则不包含业务逻辑。服务门面调用包含业务逻辑的应用服务和业务对象。

比如说，考虑这样一种情形：用会话门面实现了服务门面。但有些应用系统中的业务逻辑会越来越复杂，所以会话门面和它的各个方法也变得冗长、臃肿。采用应用服务，让会话门面把处理请求的工作委派给它，就可以避免这种情况。

如果把业务逻辑放在一个远程会话门面里，那么也会降低逻辑的可重用性。因为，既然逻辑放在了远程会话门面中，在不同的用例之间就不便于重用这些逻辑；而且很可能这些代码的重用就是一种剪刀浆糊的做法，所以也就难于维护门面代码。因此，如果需要在多个门面之间重用业务逻辑，应用服务就是放置这些逻辑的好选择，它能够使会话门面的实现变得更简单、更漂亮，而且也更容易维护。

应用服务也有助于处理多种不同的交互形式（用例不同、客户端类型不同、通信渠道不同等）。在有些应用系统中，不同的用例与同一个业务对象之间的交互会有很大不同。另外，应用系统还可能会有多种类型的客户端（或通信渠道）。如果对同一个业务对象，每一种客户端或用例都要加入特殊的处理方法，那么这个业务对象就会变得特别复杂，因为它要处理所有可能的情况。所以，对所有请求类型都通用的处理逻辑最好就封装在业务对象里，从而实现重用；相反，每种请求类型中特殊的处理逻辑（比如根据用例和客户端类型不同而变化的交互方式）则最好封装在应用服务中。

358

为了处理多种类型的交互形式，可以使用后文介绍的应用服务分层策略。该策略把不同的应用服务分成自下而上的多个层次，最底层的服务封装了通用的业务逻辑，然后的一个层次专门处理与特定用例相关的业务逻辑，最后，顶层的应用服务则负责与特定客户端相关的业务处理。

应用服务可以用来调用外部服务。如果应用系统需要访问外部服务，比如E-mail系统、遗留系统或Web Service，那么就可以在应用服务中实现这些服务的访问逻辑，这样就能提供一个可重用的服务组件。

### 设计手记：业务逻辑

在应用服务和业务对象中实现业务逻辑的时候，可能会考虑业务逻辑究竟放在哪里合适。业务对象中也能包含业务逻辑，但是有些开发者就是忽视了这一点，不在业务对象中实现任何业务逻辑。

他们可能会把所有业务逻辑作为一个单独的层次，由应用服务实现。另一方面，也有些人会把所有业务逻辑都实现在业务对象中，根本不加入那个服务层，这样业务对象就变得臃肿不堪、难于重用。

359

## 设计手记：服务门面

对于使用EJB组件的企业应用，会话门面利用session bean在业务层实现了门面模式[GoF]。这些门面的作用，是以粗粒度的形式封装业务功能，并把这些功能暴露给应用客户端，从而隐藏业务组件的复杂性以及业务组件之间的交互细节。但是，如果正在设计的企业应用并不包含EJB组件，仍需要实现类似于会话门面的功能。这时的实现方法就不是session bean了，相反，我们使用POJO实现门面，这样就有了“*POJO门面*”这个概念。

为了用一套通用的词汇、说法讨论*POJO门面*和会话门面，我们引入了“服务门面”的概念。所谓服务门面，就是一种业务层的门面，由POJO或session bean实现。所以，如果在书中看到了“服务门面”这个概念，它指的既可以是*POJO门面*，也可以是会话门面。图7-17体现了这种关系。

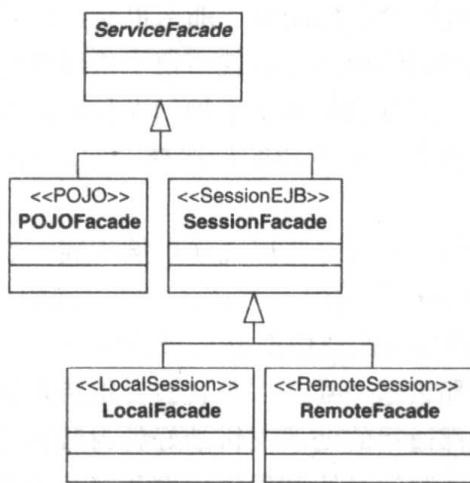


图7-17 服务门面类图

## 结构

图7-18表现了应用服务的结构。客户端访问应用服务，调用业务方法。然后，应用服务实现中的每个业务方法可能会调用多个业务对象、数据访问对象或其他应用服务。  
360

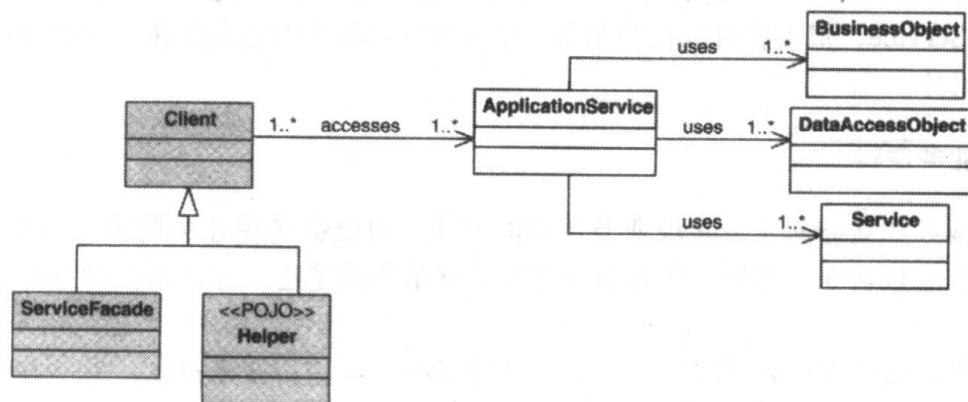


图7-18 应用服务类图

## 参与者和责任

图7-19体现了应用服务模式中多个参与者的交互。客户端调用ApplicationService（应用服务）对象上的业务方法。业务方法执行一些必须的业务处理，包括初始化、调用业务对象、调用其他应用服务、访问数据访问对象、以及执行业务逻辑，从而最终完成服务请求。

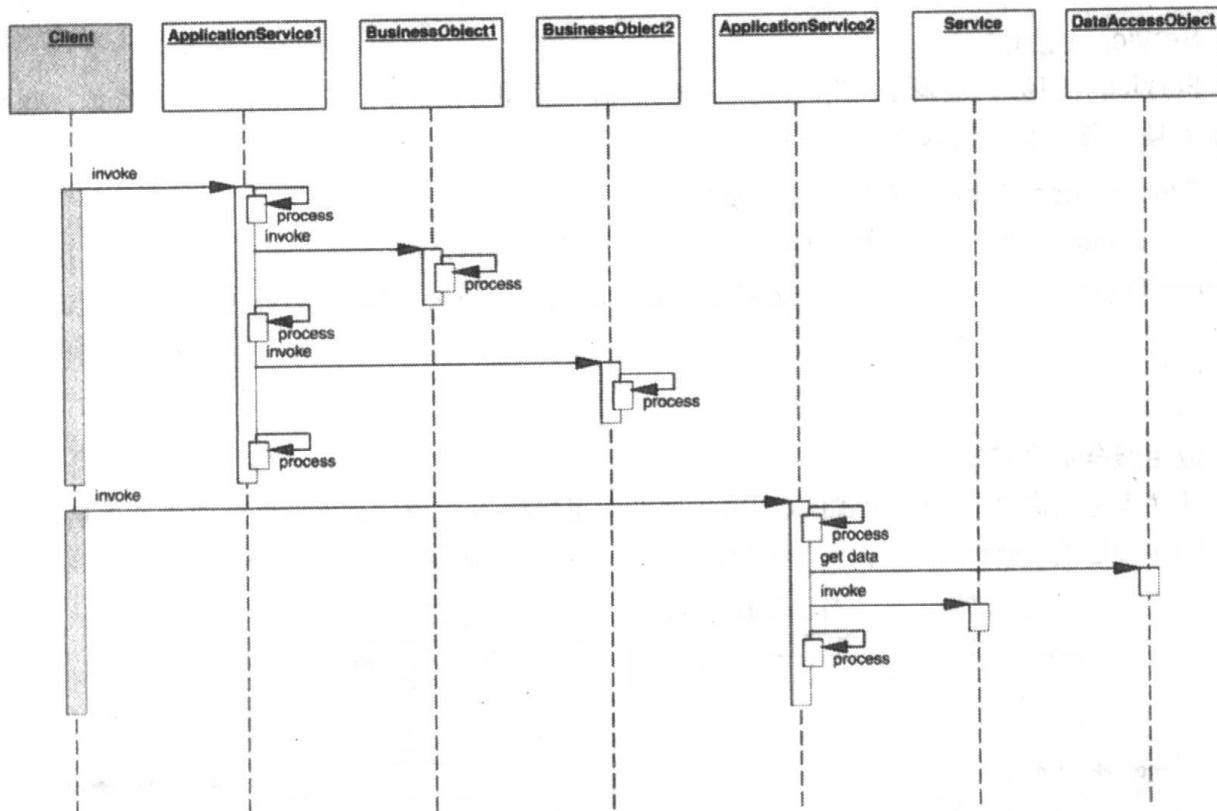


图7-19 应用服务序列图

序列图中ApplicationService（应用服务）和BusinessObject（业务对象）上的“process（处理）”消息，表现的是业务逻辑的执行。ApplicationService封装了那些外在于业务对象的业务逻辑。而BusinessObject封装了内在的业务逻辑。换句话说，ApplicationService封装的业务逻辑要涉及多个BusinessObject，而BusinessObject封装的业务逻辑，则只处理被该BusinessObject包含、并由它管理的业务数据。

361

### Client（客户端）

在这里，充当客户端角色的往往是服务门面，该门面可以使用POJO实现，也可以实现为会话门面。客户端也可能是另一个应用服务，或者是一个POJO助手对象。

### ApplicationService（应用服务）

ApplicationService（应用服务）是本模式中的主要角色，它封装了提供某种特定服务的业务逻辑。它可以封装多种类型的业务逻辑，比如：某种需要调用多个BusinessObjects（业务对象）的通用逻辑，某种与特定用例相关的业务逻辑，或是与特定类型的客户端/通信渠道相关的逻辑。ApplicationService可能会调用BusinessObject的业务方法，也可能调用另一个

ApplicationService。比较常见的做法是用POJO实现应用服务，因为这样做能够在不同的客户端之间增进控制逻辑的可重用性，也便于划分各种应用服务的层次。

362

### BusinessObject（业务对象）

BusinessObject（业务对象）是ApplicationService（应用服务）为了执行服务需要访问的业务对象实例。通常，一个ApplicationService为了完成一个服务请求要访问多个BusinessObjects。

### Service（服务）

Service（服务）也就是应用系统中提供任何一种服务的任何一种组件。这可能是一些经常用于处理业务层服务请求的通用功能或工具。

### DataAccessObject（数据访问对象）

DataAccessObject是数据访问对象的实例，对于ApplicationService（应用服务）不调用BusinessObject（业务对象），直接访问业务数据的情况，就要有DataAccessObject的参与。

## 策略

### 应用服务命令策略

应用服务可以通过命令模式[GoF]来实现。你可以使用应用控制器来提供请求处理机制。

图7-20是把Command（命令）和ApplicationService（应用服务）分开的一种做法。

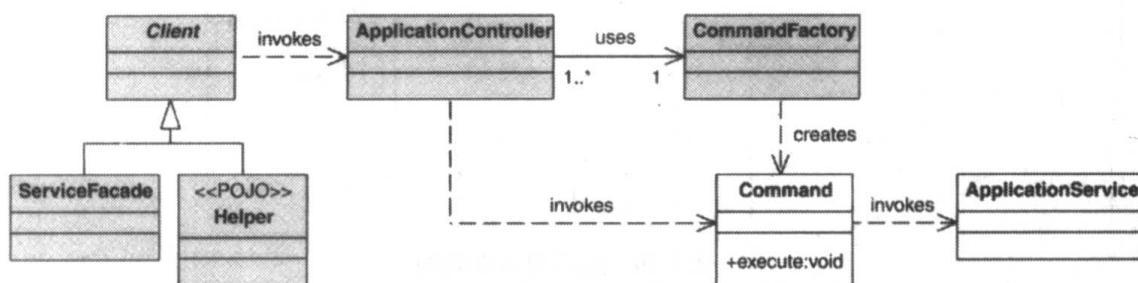


图7-20 应用服务命令策略类图

但是，所谓“命令”和“应用服务”是逻辑上的角色，所以可以把这两种角色结合起来，实现一个ApplicationService（应用服务）。这样就不必用Command（命令）调用ApplicationService（应用服务）实例了，而是把这两个合成在一起，从而用ApplicationService来实现命令模式，如图7-21所示。

363

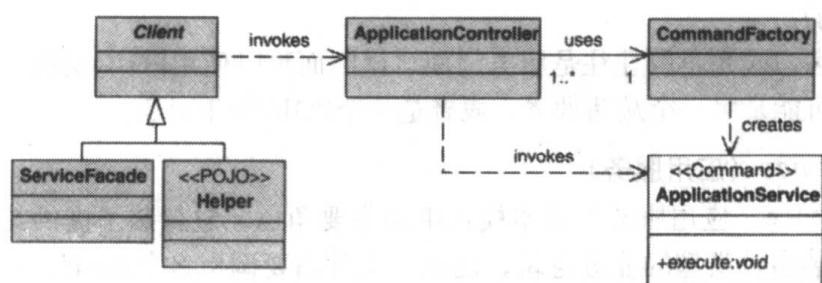


图7-21 应用服务命令策略的另一种作法的类图

图7-22的序列图表现了应用服务命令策略中的交互。

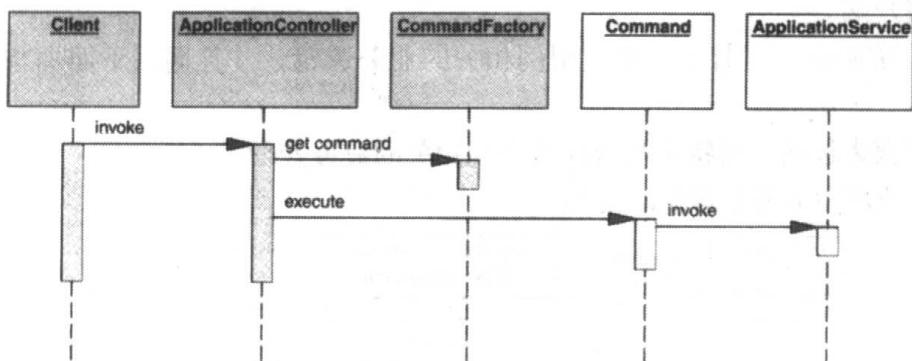


图7-22 应用服务命令策略序列图

采用后一种作法的应用服务命令策略的序列图，如图7-23所示。

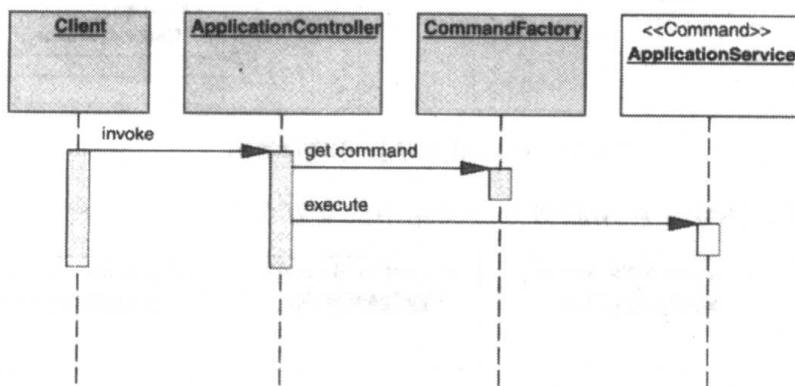


图7-23 应用服务命令策略的另一种作法的序列图

364

## 采用GoF策略模式的应用服务策略

可以使用GoF书中的策略模式[GoF]实现应用服务（见图7-24）。因为应用服务主要实现的是处理逻辑，所以当同一服务存在多种变体的时候，使用策略模式实现就非常理想。

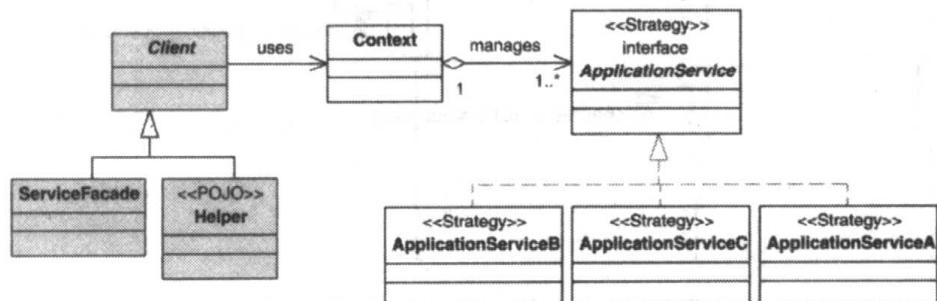


图7-24 采用GoF策略模式的应用服务策略的类图

## 应用服务分层策略

可以根据应用服务的功能和可重用性为它们分类。

在最下面是那些封装了通用的业务逻辑的应用服务，这些业务逻辑既不针对特定的用例，也不针对特定的客户端。

中间的一类应用服务，封装了与特定用例相关的业务逻辑，并且调用下面那些通用的应用服务。

最顶层的应用服务包括了与特定类型的客户端相关的业务处理。

图7-25体现了应用服务分层策略的结构。

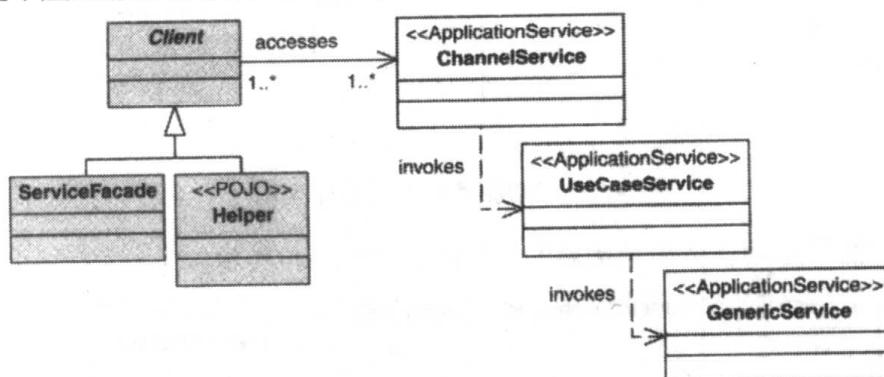


图7-25 应用服务分层策略的类图

图7-26中的序列图，体现了应用服务分层策略中的交互关系。

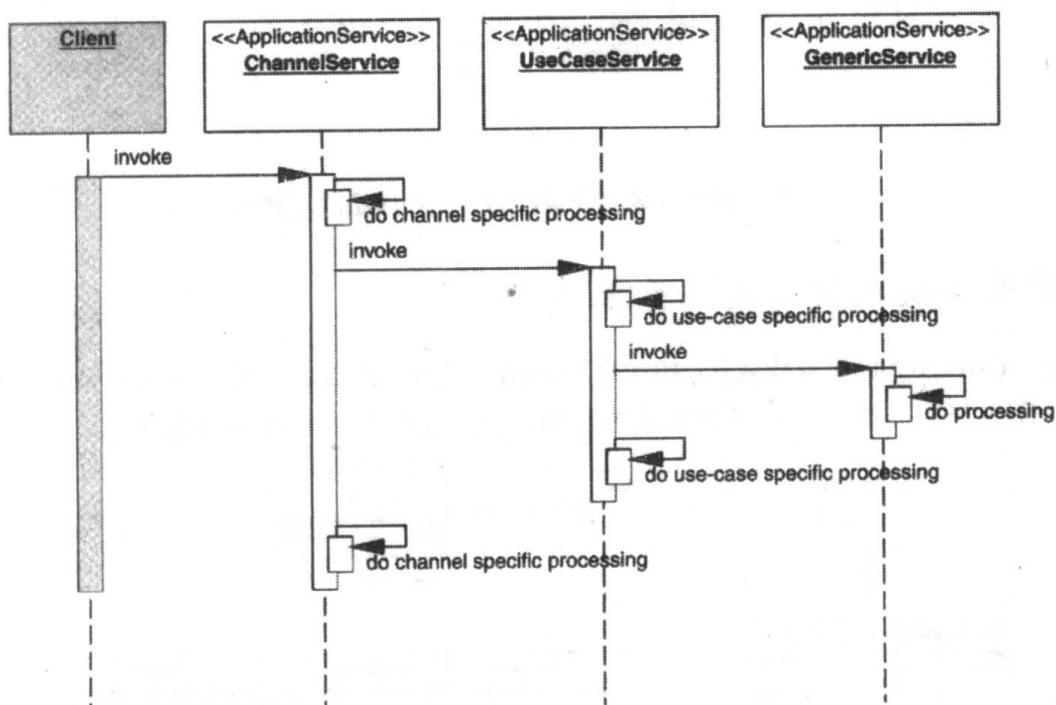


图7-26 应用服务分层策略的序列图

客户端调用ChannelService（渠道服务），这是ApplicationService（应用服务）的一个实现，封装了与特定通信渠道相关的预处理/后处理逻辑。执行了预处理之后调用UseCaseService（用例服务），这是ApplicationService的另一个实现，封装了与特定用例相关的预处理/后处理。

UseCaseService执行预处理，调用GenericService（通用服务），这也是ApplicationService的一个实现，封装了对于所有通信渠道和用例都通用的业务处理逻辑。

以上每一个层次的应用服务都是可选的，只是在必要时才应该实现。比如说，可能在系统中就没有GenericService（通用服务）。

## 效果

- 集中了可重用的业务逻辑和工作流逻辑

应用服务创建了一个服务的层次，封装了业务对象层。这也就创建了一个集中的层次，封装了一些通用的、针对多个业务对象的调用操作。

- 提高了业务逻辑的可重用性

应用服务创建了一组可重用的组件，能够在不同的用例实现之间重用。应用服务封装了涉及多个业务对象的操作。

- 避免了代码重复

应用服务创建了一个集中化的可重用的业务服务层，这样就避免了在它的客户端（比如门面和助手）产生代码重复，也避免了其他应用服务中的代码重复。

- 简化了门面的实现

业务逻辑从服务门面（无论是用SessionBean实现的会话门面，还是POJO门面）中转移出来。这样，门面就变得更简单了，只负责与应用服务交互，把请求委派给一个或多个应用服务完成处理。

- 在业务层中引入了附加的层次

应用服务在业务层中创建了一个附加的层次，看起来这对于一些应用环境来说好像是一种不必要的负担，但是这个附加层次也封装了一些可重用的通用业务逻辑，从而给应用系统中引入了一种有效的抽象。

## 示例代码

图7-27的类图，表现了一个示例，该例是在一个订单处理系统中使用应用服务模式的。其中的组件和各自相应的责任如下（见例7.13至例7.20）：

- OrderSystemFacade（订单系统门面）是订单系统的一个会话门面。它负责与应用服务组件（CompanyAppService和OrderAppService）交互。
- CompanyAppService（公司应用服务）是一个应用服务组件，它负责一些业务逻辑，并与几个业务对象—Company（公司）、Account（账户）和AccountManager（账户管理者）——交互。
- OrderAppService（订单应用服务）是一个应用服务组件，它负责一些业务逻辑，并与几个业务对象—Order（订单）和Account（账户）——交互。
- EmailAppService（Email应用服务）是一个应用服务组件，它负责实现email服务。EmailAppService被CompanyAppService和OrderAppService使用。

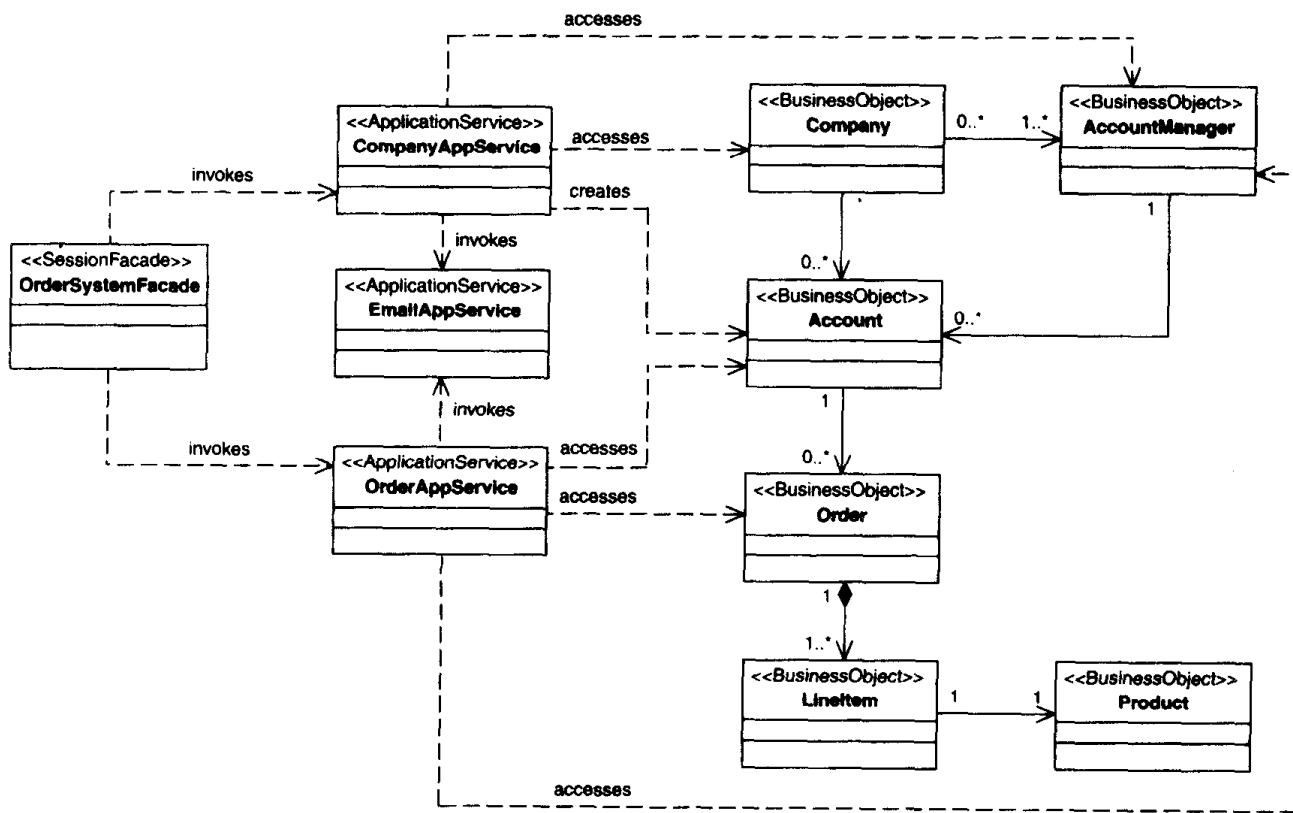


图7-27 应用服务：订单系统示例类图

### 例7.13 OrderServiceFacadeSession (订单服务门面SessionBean)

```

1 import javax.ejb.SessionContext;
2 import javax.ejb.EJBException;
3 import java.rmi.RemoteException;
4 import java.util.Date;
5
6
7 public class OrderSystemFacade
8 implements javax.ejb.SessionBean, OrderSystemClient {
9
10    public void setSessionContext(
11        SessionContext sessionContext)
12        throws EJBException, RemoteException {
13    }
14
15    public void ejbRemove()
16        throws EJBException, RemoteException {
17    }
18
19    public void ejbActivate()
20        throws EJBException, RemoteException {
21    }
22
  
```

```

23     public void ejbPassivate()
24         throws EJBException, RemoteException {
25     }
26
27     public void placeOrder( OrderTO order ) {
28         OrderAppService pas = new OrderAppService();
29         pas.placeOrder( order );
30     }
31
32     public void addAccount( AccountTO account ) {
33         CompanyAppService cas = new CompanyAppService();
34         cas.addAccount( account );
35     }
36
37     public void createCompany( CompanyTO company ) {
38         CompanyAppService cas = new CompanyAppService();
39         cas.createCompany( company );
40     }
41 }
```

#### 例7.14 CustomerAppService (客户应用服务)

```

1
2     public class CompanyAppService {
3
4         public void addCompany( CompanyTO companyTO ) {
5             Company customer = new Company( companyTO );
6             // 客户的持久化存储
7         }
8
9         public void addAccount( String companyId,
10             AccountTO accountTO ) {
11             throws AccountException {
12
13             Company company = null;
14             // 获取公司
15             // company = findCompany( companyId );
16
17             if ( company == null )
18                 throw new AccountException(
19                     "Company Id: " + companyId + " doesn't exist");
20
21             Account account = new Account( accountTO );
22             company.addAccount( account );
23
24             AccountManager accountManager = null;
25             accountManager = getAccountManager( company.getZip() );
26             account.addAccountManager( accountManager );
27 }
```

```

28         // 发一封email给账户管理者, 通知
29         // 有新账户了
30         EmailAppService emailAppService =
31             new EmailAppService();
32         emailAppService.notifyAccountManager(accountManager,
33             account);
34     }
35 }
```

### 例7.15 OrderAppService (订单应用服务)

```

1
2     public class OrderAppService {
3
4         public OrderAppService() {
5             }
6
7         public void placeOrder(OrderTO orderTO)
8             throws AccountException {
9             Company company = null;
10            Account account = null;
11
12            String companyId = orderTO.companyId;
13            String accountId = orderTO.accountId;
14
15            // Company (公司) 的寻址
16            company = getCompany(companyId);
17
18            // Account (账户) 的寻址
19            account = company.getAccount(accountId);
20
21            // 把订单添加到账户中
22            Order order = new Order(orderTO);
23
24            // 计算折扣
25            calculateDiscount(company, account, order);
26
27            account.addOrder(order);
28            AccountManager accountManager =
29                account.getAccountManager();
30
31            // 发一封email给账户管理者, 通知
32            // 有新订单了
33            EmailAppService emailAppService =
34                new EmailAppService();
35            emailAppService.notifyAccountManager(accountManager,
36                order);
37        }
38 }
```

```

39     public void calculateDiscount(Company company,
40         Account account, Order order) {
41     }
42
43 }
```

### 例7.16 EmailAppService (Email应用服务)

```

1 // import
2
3 public EmailAppService {
4
5     public void notifyAccountManager(
6         AccountManager accountManager, Order order) {
7         // 集成Email API, 发送通知
8         // 邮件
9     }
10
11    public void notifyAccountManager(
12        AccountManager accountManager, Account account) {
13        // 集成Email API, 发送通知
14        // 邮件
15    }
16 }
```

371

### 例7.17 Order (订单)

```

1
2 import java.util.Collection;
3
4 public class Order {
5     String orderId;
6     Collection lineItems;
7
8     . . .
9
10    public void addLineItems( LineItem[] lines ) {
11        for( int i = 0; i < lines.length; i++ ) {
12            lineItems.add( lines[ i ] );
13        }
14    }
15 }
```

### 例7.18 OrderTO (订单传输对象)

```

1
2 import java.util.Date;
3
4 public class OrderTO implements java.io.Serializable {
5     public String companyId;
6     public String accountId;
```

```

7     public String orderId;
8     public LineItemTO[] lineItems;
9     public Date    orderDate;
10 }

```

### 例7.19 LineItem (订单中的货品)

```

1
2 import java.util.Date;
3
4 public class LineItem {
5     String productId;
6     float   price;
7     int      quantity;
8     Date    deliveryDate;
9
10    public LineItem(LineItemTO lineItem) {
11        this.productId = lineItem.getProductId();
12        this.price = lineItem.getPrice();
13        this.quantity = lineItem.getQuantity();
14        this.deliveryDate = lineItem.getDeliveryDate();
15    }
16 }

```

372

### 例7.20 LineItemTO (订单货品传输对象)

```

1
2 import java.util.Date;
3
4 public class LineItemTO implements java.io.Serializable {
5     public String productId;
6     public float   price;
7     public int      quantity;
8     public Date    deliveryDate;
9 }

```

## 相关模式

- **会话门面**

应用服务为会话门面提供了后台的底层架构。使用了应用服务，会话门面实现起来就更简单了，门面包含的代码也更少了，因为它可以把业务处理委派给应用服务。

- **业务对象**

对于使用业务对象的应用系统，应用服务封装了那些要操作多个业务对象的业务逻辑，并且负责和多个业务对象交互。

- **数据访问对象**

在一些应用系统中，应用服务可以直接调用数据访问对象，访问数据存储中的数据。

- **服务层 [PEAA]**

应用服务类似于PEAA一书中提到的“服务层”模式，因为二者的目标都是在应用系统中添加一个服务层。服务层模式中介绍了怎样利用一组服务来为应用系统创建一个边界层次。

- **事务脚本 [PEAA]**

如果在不使用业务对象的系统中采用应用服务，那么应用服务本身就成了一个服务对象（service object），可以在其中实现过程式<sup>①</sup>的逻辑。PEAA一书中的事务脚本模式介绍的就是如何在应用系统中以过程式的方法实现业务逻辑。

373

## 业务对象

### 问题

有一个业务领域概念模型，其中包括业务逻辑和关系。

如果业务操作中只有很少、甚至没有业务逻辑，那么应用系统通常会让客户端直接访问数据存储中的业务数据。比如说，一个表现层组件（比如一个命令助手或者一个JSP视图）或者一个业务层组件，都可能直接访问一个数据访问对象。在这种情况下，业务层中就没有“对象模型”的概念了。应用需求由一个面向过程的实现完成。如果在应用系统中，数据模型和业务领域概念模型特别接近，那么采取这种做法也是可以接受的。

但是，如果概念模型包含了多种业务功能和关系，那么使用面向过程的方式实现这种应用系统就会导致以下问题：

- 降低了可重用性，业务逻辑代码发生了重复。
- 面向过程的系统实现变得很臃肿，既冗长又复杂。
- 系统可维护性很低，因为代码重复很多，而且业务逻辑分散在多个不同的模块中。

**设计手记：术语解释——业务模型、业务对象模型、业务领域模型、对象模型、数据模型 (Business Model, Business Object Model, Domain Model, Object Model, Data Model)**

为了理解这些概念，我们在此援引Jacobson等著的《统一软件开发过程》一书中的定义 [Jacobson, et al.]。

- **业务模型**由两种模型构成：**业务用例模型**描述的是业务角色和业务过程，而**业务对象模型**描述的则是业务用例所使用的业务实体。
- **业务领域模型**则定义为一种抽象模型，它固化了在系统的上下文环境中最重要的若干类对象。业务领域对象代表着在系统运转的环境中存在的一些“东西”和发生的一些事件。

另外，Jacobson等人还说，业务领域模型是业务模型的另一种说法，这两个概念可以互换使用。我们在实践中看到分析师、设计师和开发者们确实是混用这几个概念的：业务模型、业务对象模型、业务领域模型、业务领域对象模型。这样的混用冲淡了这些概念的含义，把它们变得有点儿含混不清了。

374

① 所谓过程式的逻辑，是相对于“面向对象”而言的。如果不采用业务对象，而让一整段“过程”代码（可能就是一个函数/方法）执行对一个请求的处理，这就是一种“过程式的”做法。

在本书的讨论中，“**概念模型**”这个说法指的是抽象模型，它描述的主要是一些业务领域实体、这些实体之间的关系以及业务规则。而“**对象模型**”这个说法指的则是概念模型的一种具体的面向对象的实现模型，它描述的是为了实现这个概念模型所需要的类和关系。另外一个说法“**数据模型**”，指的则是数据实现模型，比如用于RDBMS（关系型数据库管理系统）数据库的实体-关系模型（ER）。

## 约束

- 概念模型包含结构化的、相互关联的复合对象。
- 概念模型包含复杂的业务逻辑、验证和业务规则。
- 把业务状态和业务功能同系统的其他部分区分开，增进系统的内聚和可重用性。
- 集中应用系统中的业务逻辑和业务状态。
- 提高业务逻辑的可重用性，避免代码重复。

## 解决方案

**使用业务对象，利用对象模型把业务数据和业务逻辑分离出来。**

业务对象负责封装并管理业务数据、业务功能和持久化机制。业务对象有助于把持久化机制从业务逻辑中区分出来。业务对象负责维护核心的业务数据，并且实现在整个应用系统或业务领域中通用的一些功能。

在一个使用了业务对象的应用系统中，客户端与业务对象交互，而后者则可以使用多种持久化策略管理自身的持久化机制。业务对象实现了一个可重用的业务实体层，该层描述了特定的业务领域。一个业务对象实现了一种定义完备的业务领域概念，并且还包含了一些适用于该概念的业务逻辑和业务规则。至于一些高层次的业务逻辑或者作用于多个业务对象的外在业务逻辑，则由一个服务层通过应用服务和会话层面实现，这样也就能把对象模型和客户端隔离开来，避免了直接访问。

实现业务对象，主要有两种策略。每种策略中要创建的对象类型、实现的持久化机制都不同。

- 第一种策略是使用POJO，并且选择一种适合需要的持久化机制。可以从以下机制中选择：定制数据访问对象、用业务领域存储实现JDO形式的定制持久化框架、标准JDO实现。
- 第二种策略是按照复合实体模式中的做法使用entity bean。使用这个策略还要决定是使用BMP还是CMP来完成持久化。

另外，对于这两种策略，在实现业务对象时都要考虑安全性、事务管理、池、缓存、并行处理等需求。

有些较简单的应用系统可能会把业务对象暴露给所有的客户端。这样访问业务对象会直接导致细粒度的交互。如果这是一个POJO应用系统，客户端和业务对象都处于同一虚拟机的进程空间里，那么这种做法可能还行得通；但是对于远程应用系统来说，这种实现方式可能就是低效的和缺乏一致性的。对于比较复杂的应用系统，就不应该直接把业务对象暴露给所有客户端，而应使用服务门面和POJO应用服务封装业务对象。

对于远程应用系统，就应该使用会话界面，构成一个远程服务界面，业务对象通过这个界面暴露给客户端。这样，就不会把业务对象返回给业务层之外的客户端；而是使用传输对象在客户端和业务对象之间交换数据。

确定、设计业务对象，并为之提供持久化机制，这是一种复杂的工作，常常要花费相当的时间和资源。以下部分讨论了怎样确定业务对象，以及怎样选择合适的持久化机制。

### 设计手记：在非商业空间<sup>①</sup>中的业务对象

对于我们处理的大多数问题领域来说，“业务对象”这个概念都是有效的，所以我们才选择它作为模式名称。但是，对于一些“非商业”的问题领域，“业务对象”这个概念可能就不太容易被接受了。这样的问题领域包括卫生保健、政府机构、军事机构和慈善机构，这些机构并不把自己看成是“商业的”。还有一些高科技应用系统、统计应用系统、数学建模、分析建模等，其中也没有“商业”成份。

在这些情况下，就不要使用“业务对象”这个概念了。可以使用“领域对象（Domain Object）”的说法，这个名称下体现的概念其实和本书中讨论的“业务对象”是一致的。376

## 挑选/确定业务对象

业务领域概念模型中的实体，也就是用户在用例中创建、访问并操纵的那些对象。实体体现了业务系统中的那些通常是有状态的、持久化的、持续存活的业务数据。实体可以在多种用例中、按不同的目的重用，它们也就是这些用例中的那些“名词”。比如，在订单处理系统中的实体就会是“订单”、“订单货品”、“发票”和“账户”。这些实体完全是面向用户、或者说面向业务的，系统的使用者就能够确定、并且描述这些实体。

确定了以上实体之后，还要寻找“关联实体”，也就是体现了两种实体之间的某种有意义的业务关联的实体。比如说，在订单处理系统中，可以找出两种独立的实体“订单”和“订单货品”，然后就能确定“订单-订单货品”这种关联实体。所以，当一个用户希望创建一个新订单以订购一种货品时，这个“创建订单”用例就必须还要创建一个相应的“订单-订单货品”关联实体。这种关联实体通常是面向设计师/架构师的，用户往往体察不到，因为往往是开发者来创建这种实体，用以支持前面那些面向用户实体的实现。

确定了所有实体之后，必须把它们实现为有状态的、持久化的、持续存活的业务对象。通常，业务对象具有唯一性，并且对应于数据存储（比如关系型数据库）中的一条物理数据。

请切记：确定业务对象是一个复杂的过程。有大量各种文献都介绍了业务对象实体的建模和确定方法。可能会用到的资料有：

- Object-Oriented Software Engineering: A Use Case Driven Approach (《面向对象软件工程：用例驱动方法》) Ivar Jacobson著 [OOSE]

<sup>①</sup> 非商业：这里的“商业”和“业务对象”名称中的“业务”都是英语business的翻译。在模式名称中，称为“业务”更为贴切；而这里作者谈的“非商业领域”，指的是一些并非完全以营利为目的的应用领域。所以只能一词多译，希望读者体察。

- Software Reuse: Architecture Process and Organization for Business Success (《软件重用：架构、过程和组织》) Jacobson等著[SR]
- Domain-Driven Design (《业务领域驱动设计》) Eric Evans 著[DDD]

## 业务对象和持久化

好了，找到了所有的业务对象，那么就该确定如何存储数据了。业务对象有助于把持久化存储逻辑从业务逻辑里分离出来。既然几乎所有业务对象都需要持久化存储，那么就需要某种持久化机制，把对象状态映射到一种数据存储中。一种做法是使用业务领域存储提供一种框架为业务对象实现透明的持久化。

377 虽然可以直接在业务对象里编码实现持久化，我们还是强烈建议不要这样做，因为这样会把持久化逻辑和业务对象混在一起，最后会导致业务层和持久化存储层的组件之间产生紧耦合。把系统中两种不相关联的内容这样合并到一起，会降低对象的内聚性、模块化程度和可维护性。而且，业务对象模式的主要目的在于为业务状态和业务功能建模。随着业务模型向复杂方向发展，把持久化存储逻辑放在业务对象中就会使对代码难以维护，使对象模型和数据模型之间产生紧耦合。

### 设计手记：实现业务对象持久化

业务对象的一个核心特征，就是它代表着持久化的业务信息。可以通过不同方式实现持久化。

- 使用entity bean持久化
- 是用定制数据访问对象
- 使用Java数据对象 (JDO)
- 使用业务领域存储

#### 使用Entity Bean持久化

可以按照复合实体模式中介绍的方式，使用entity bean实现业务对象。对于entity bean来说，还可以选择是BMP还是CMP实现持久化。如果选择了BMP，那么通常就要用到数据访问对象。

#### 使用定制数据访问对象

378 可以使用数据访问对象为业务对象提供持久化机制。这样做的时候，首先要创建业务对象，然后由它把业务数据的存储/获取操作，委派给定制的数据访问对象。业务对象的创建和其他对象的创建并无不同。但是，如果要为业务对象实现持久化存储，就应该把持久化逻辑委派给数据访问对象完成，而不应该把这种逻辑放在业务对象内部。还可以使用数据访问对象获取现有的业务对象的状态。虽然这种做法能把数据访问逻辑从业务逻辑中分离出来，但是它还不是一种透明地实现业务对象持久化的方法，因为业务对象还要知道如何调用数据访问对象。如果想透明地使用数据访问对象，更好的方法是通过业务领域存储使用它们。

#### 使用Java数据对象 (JDO)

Java数据对象是另一种实现业务对象持久化的方法。这个方法很有吸引力，因为持久化机制对于业务对象是完全透明的。JDO技术使用了一种叫“增强器（enhancer）”的机制，它能够对业务对象的类文件或者字节码执行“增强”，加入持久化功能。很多现成的持久化解决方案都是基于JDO的，所以，采用了这个策略就能够选择最适合自身需求的JDO方案。

## 使用业务领域存储

业务领域存储模式能够为对象模型提供透明的持久化机制。其中还包括用定制持久化机制实现的策略，以及用JDO形式的持久化机制实现的策略。关于使用这一机制实现业务对象持久化的细节，请参见第8章的业务领域存储一节。

## 结构

业务对象模式的类图如图7-28所示。

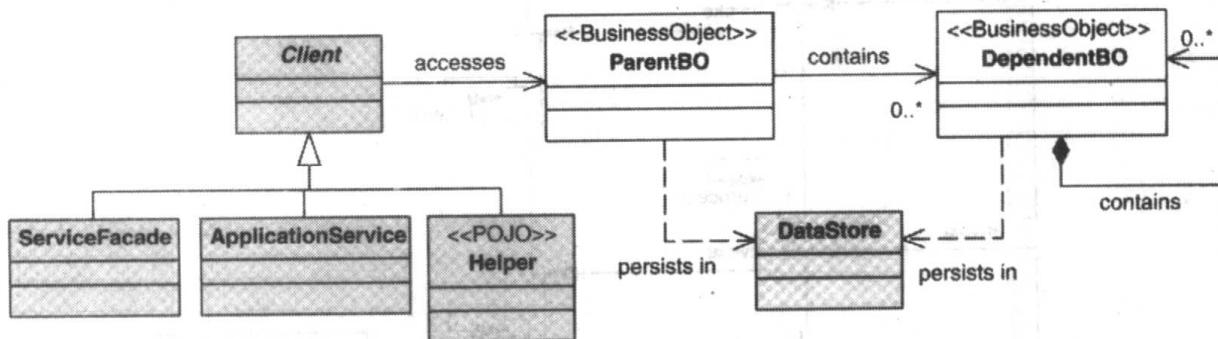


图7-28 业务对象类图

## 参与者和责任

图7-29表现了业务对象模式中多个参与者之间的交互。

客户端调用ParentObject（父对象）的一个业务方法，后者从数据存储中装载数据。然后ParentObject基于数据执行内部业务逻辑。如果有需要保存到数据存储中的修改，那么ParentObject就保存该数据。ParentObject还可以调用从属BusinessObject（业务对象）——DependentBO1和DependentBO2——的方法，实现客户端的请求。每个从属对象也执行自身的（内部）业务逻辑和业务规则，并且在需要时也会与数据存储交互。

379

### Client（客户端）

Client代表了BusinessObject（业务对象）的客户端。它通常可能是会话界面、助手对象（见第6章的视图助手一节），或者是一个需要访问该BusinessObject的应用服务。

380

### ParentBO（父业务对象）

ParentBO是顶级BusinessObject（业务对象），它代表了BusinessObject的复合模型中的父对象。父对象封装了从属对象，实现了自身内部业务逻辑和业务规则。

### DependentBO（从属业务对象）

这就是由ParentBO（父业务对象）负责管理的从属BusinessObject（业务对象）。从属BusinessObject和父对象之间存在着紧耦合，依靠父对象实现生命周期管理。从属对象不能脱离父对象独立存在。每个从属对象都要实现它自身的内部业务逻辑和业务逻辑，以完成自己的责任。

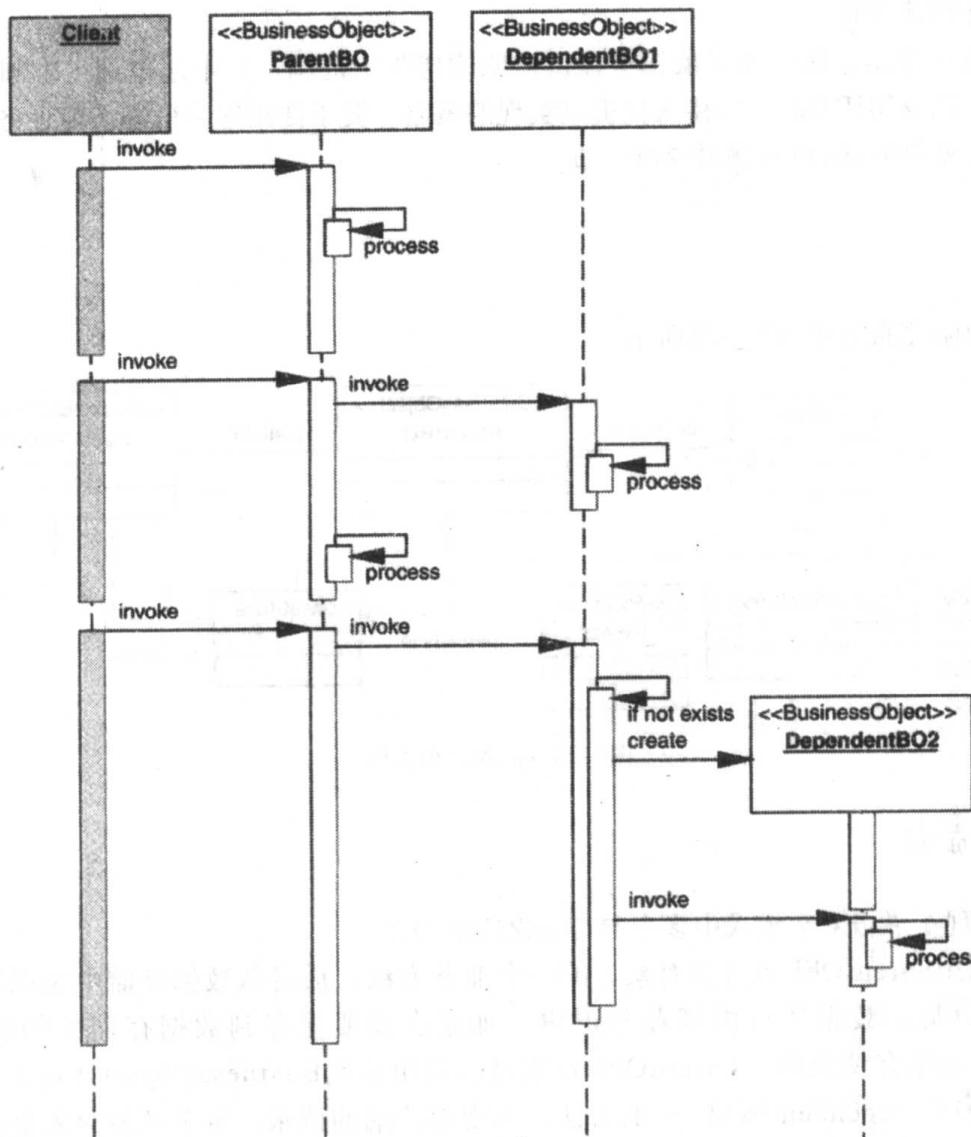


图7-29 业务对象交互图

### 设计手记：业务对象和验证

业务对象起着“数据的保护者”的作用，负责维护数据的完整性。为了达到这个目的，业务对象就必须对客户端传来的数据执行验证。

在多层应用系统中，验证可以在很多不同的地方执行。比如，在对一个应用系统提交新数据的时候，表现层就可以对表单的输入执行验证，保证所有必填字段都不为空，并且保证对表单中的所有字段都执行了数据完整性和数据类型的检查。另外还可以有其他的验证，帮助应用系统避免对业务层的多余调用。这些验证可以在表现层的前端控制器、应用控制器和命令对象中执行。

当表现层认为数据有效之后，它就允许请求调用业务层，而业务层也要执行自己的一套验证。其中有些验证可能是与表现层执行的验证重复的。

但这种重复可能也是必需的，因为业务层不能完全依靠其他的层次确保数据完整性。有些调用业务对象的客户端可能都不属于表现层。这些客户端可能属于集成层，所以，在业务层加入验证无论如何是必要的。业务层中有一些重要的接触点，适合包含验证逻辑，这里包括会话界面、应用服务和业务对象。

## 策略

### POJO业务对象策略

即使使用了EJB容器，还是能够用POJO来实现业务对象。举例来说，可以在一个EJB应用系统里只使用会话界面，但不实现entity bean。当然也可以在一个J2EE应用系统中使用JSP和servlet技术，但根本不用EJB。在这种情况下，可用POJO实现业务对象。采用这种方式实现业务对象的时候，应该考虑这一做法对系统架构的各方面（包括安全性、事务、缓存、池、并行处理和持久化等方面）的影响。381

缓存是一种优化技术，一些应用系统实现了某种缓存机制，用于优化业务对象的装载和存放过程。虽然对单进程空间的POJO应用系统来说，实现缓存相对还算容易；但是对于多层分布式的应用系统来说，同一业务对象可能在多个虚拟机或多个容器中都有实例，这样缓存就很麻烦了。在这多个实例之间保持业务数据的一致性和完整性是一个很难解决的问题。

这需要在多个实例之间、实例和数据存储之间实现状态的同步，这样才能保证业务数据的完整性，避免数据失效。

在这个策略里需要选择究竟采用何种持久化机制。选择不同的持久化机制，持久化的透明程度也不同。以下就是对各种选择的一个总结，这个总结是基于前面的设计手记“实现业务对象持久化”做出的。

- 如果使用数据访问对象，实现自己的定制JDBC持久化机制，那么持久化机制与业务对象之间就存在着非常紧密的关系。业务对象本身就负责实现持久化。
- 想要更为透明的持久化机制，可以采用业务领域存储模式，它仍然是一种定制的解决方案，但是解除了持久化机制和业务对象实现之间的耦合。
- 与此类似也可以用JDO实现业务领域存储的一些优势，但是JDO是一种业界标准做法，也具有一套标准的对象持久化API。
- 最后也可以使用一些现成的对象-关系型数据库映射方案，不同的方案功能各不相同，透明化程度也不相同。

### 复合实体业务对象策略

在EJB应用系统中可以使用entity bean实现业务对象，这个策略在复合实体模式一节中有详细介绍。在EJB 2.x之前的技术规范版本中，entity bean一直都是远程对象，这也导致了系统性能低下。382

EJB 2.x技术规范引入了本地entity bean的概念。如果用entity bean实现业务对象，我们推荐采用本地entity bean。这样可以避免远程对象的网络负荷，同时能够充分利用EJB技术带来的强大功能，包括安全性、事务、容器管理持久化（CMP）等。另外，必须使用会话界面来封装

entity bean业务对象。

无论怎样管理持久化<sup>①</sup>，entity bean都能够提供事务管理、安全性管理等优势。虽然不是每个应用系统都有对事务和安全性的需求，但对于许多采用了基于角色的安全策略、容器-管理的事务等设计的应用系统，entity bean还是相当适用的。关于利用entity bean实现业务对象的进一步细节，请参见本章复合实体模式一节。

### 设计手记：使用POJO还是Entity Bean实现业务对象？

虽然使用POJO业务对象从表面上看似乎更为简单和直白，但之所以产生这种印象，往往是因为我们忽略了底层架构和容器支持对维持系统正常运转所起的作用。很多人都低估了一种功能完备的业务对象底层架构的实现难度。业务对象的生命周期管理极为复杂，而且可能还需要很多支持服务，实现持久化、安全、事务、缓存、对象池以及并行控制等需求（我们先只举这些）。有一些应用系统把以上需求中的大部分工作都委派给了持久化存储，这样，在业务层就采用一种面向过程的做法或者是一种轻量级的业务对象包装器做法。

当抉择究竟是使用POJO还是Entity Bean实现业务对象的时候，也需要评估以上需求。如果不慎审慎的话，最后所做的工作很可能就是只自行开发了一种类似于EJB容器的系统，而不是采用现成可用、性能早就得到验证的容器实现。表7-1对比了POJO和Entity Bean的两种做法，通过对比有助于理解如何权衡选择使用POJO还是Entity Bean实现业务对象。

表7-1 POJO业务对象和Entity Bean业务对象之间的功能比较

功 能	POJO	entity bean
安全性	如果由会话门面管理对POJO业务对象的所有访问，那么安全模型就能够由EJB容器提供，并在会话门面中得到充分利用 如果根本不使用EJB技术，就只能利用自己的定制框架，或者使用Java认证和授权服务[JAAS]的Java API，实现应用系统的安全性	如果由会话门面管理对业务对象的所有访问，那么安全模型就能够由EJB容器提供。无论是会话门面本身，还是entity bean实现的业务对象，都能够充分利用这个安全模型
事务	与上面的“安全性”情况类似。如果采用了会话门面，那么session bean就可以利用EJB容器提供的事务服务。如果根本没有采用EJB技术，那么很可能就要依靠持久化策略来实现事务服务了	与上面的“安全性”情况类似。如果采用了会话门面，那么session bean就可以利用EJB容器提供的事务服务。而且，entity bean默认就具有容器管理的事务
缓存	除非实现了专门的缓存机制，不然POJO业务对象是不会被缓存的。对于POJO来说，缓存能够显著地提高系统性能，但是也会增加应用系统的复杂度	EJB容器管理entity bean的生命周期。entity bean通常不被缓存。当事务开始的时候，容器从可用实例的池中激活一个实例，如果实例并不存在，则创建一个新的实例

383

① 也就是说，无论你采用CMP还是BMP。

(续)

功 能	POJO	entity bean
池	POJO业务对象不会被显式的放入池中。如果需要池就必须自己实现这个机制，不过这个做法也未必有明显的好处	因为容器管理了entity bean的生命周期，所以entity bean的池操作也委派给容器管理，无需为entity bean业务对象实现任何池机制
并行	你自己需要负责以下内容：正确处理对同一业务对象实例的并行访问，并且一直维持数据完整性。可能需要使用某种锁定机制，比如乐观锁定或悲观锁定。Martin Fowler在他的[PEAA]一书中对并行问题进行了很好的讨论，相关的章节是“并行”和“离线并行”，其中的模式包括：乐观锁定、悲观锁定、粗粒度锁定和隐性锁定	容器会确保对entity bean的并行请求会被依序处理，并且确保一直维持entity bean的数据完整性
数据同步	在多层应用系统中，可能会发生这样的情况：在不同的Java虚拟机或容器中存在着同一业务对象的多个实例。那么，就确保在你的系统实现中，业务对象的一个实例上发生的修改要传播给所有现存的示例——如此保证正确的数据同步，这可是你的责任	如果多个不同的虚拟机或者容器需要同一个entity bean的多个实例，那么EJB容器需要负责确保：同一个业务对象的数据在不同实例之间应该同步；对业务对象的修改应该传播到不同容器中的实例上，这样才能保证数据完整性
持久化	既然业务对象是用POJO实现的，就不能使用entity bean的CMP策略了。但是为了实现POJO业务对象持久化，也还有一些其他强大的机制可资利用，比如DAO、业务领域存储、JDO、或者商业O-R映射工具等（参见前文的设计手记：“实现业务对象持久化”）	用entity bean实现业务对象，就能够充分利用CMP的强大功能，而且还可以用CMR实现业务对象entity bean之间的关系。还可以选择用BMP实现定制的持久化机制，但这样就需要自己编码实现业务对象之间的关系，因为CMR只对CMP entity bean适用

## 效果

- 在业务模型的实现中促进了面向对象实践

业务对象构成了一个逻辑责任层，该层反映的是对业务模型的对象化模型实现。对于多层的OO应用系统来说，这种用对象实现业务层的做法再自然不过了。

- 集中了业务功能和业务状态，促进了重用

业务对象使用一组独立的组件，抽象并实现了业务逻辑、业务状态和业务功能，从而提供了一种集中化、模块化地实现多层架构的做法。这种集中抽象出了业务层中的多个用例之间、多种客户端之间的核心逻辑，并且促进了这种逻辑的重用。

- 避免了代码重复，提高了代码的可维护性

因为业务状态和业务功能都被集中起来了，就不用把业务逻辑放在客户端，这样也就避免

384

385

了代码的重复。使用业务对象能够提高系统的整体可维护性，因为业务对象促进了代码的重用和集中化。

- 把持久化逻辑从业务逻辑中分离出来

可以把持久化机制从业务对象中分离出来，并且对后者隐藏这部分机制。有多种不同的持久化策略，比如JDO、定制JDBC、对象-关系型数据库映射<sup>⊖</sup>工具或entity bean实现业务对象的持久化。

- 促进了面向服务的架构

对于应用系统中的所有客户端，业务对象起到了一个集中化的对象模型的作用。可在业务对象之上构建多种服务，而业务对象本身也可以使用其他多种服务，比如持久化存储、业务规则、业务集成等等。这就隔离的多层应用系统的多个不同方面，促进了面向服务的系统架构。

- POJO 实现可能引入失效数据，并且容易受到失效数据的影响

如果在分布式多层应用系统中把业务对象实现为POJO，那么在多个虚拟机或容器中就可能有同一个业务对象的实例。应用系统要自己负责这些实例中的业务数据的一致性和完整性。这也许就会需要在多个实例之间、实例和数据存储之间实现状态同步，这样才能保证业务数据的完整性，避免数据失效。另一方面，如果采用entity bean实现业务对象，就是由容器负责所有业务对象实例的创建、同步以及其他生命周期管理操作，也就无需过问数据完整性的问题了。

- 添加了一个额外的间接层

在有些应用系统中，采用这种方式，严格分离系统的各种层次，与其说是必要，不如说是画地为牢。有些应用系统中的业务模型、业务逻辑很简单，还有一些应用系统中的数据模型就足以反映业务模型，使用表现层组件直接通过数据访问对象访问资源层反而更为简洁，在这些情况下，添加额外的间接层尤其像是画地为牢。但是，很多设计师一开始就会假定数据模型完全能满足需要，但是后来却意识到自己当时的分析不够充分，过早做出了结论。而在开发的后期阶段再弥补这个错误，往往是代价昂贵的。

386

- 可能产生臃肿的对象

有些用例可能只需要调用封装在业务对象内部的一些功能。随着越来越多的与用例相关功能被实现在业务对象里，业务对象就会变得臃肿。有一些功能与业务对象关系不太紧密、而只是与一个特定的用例相关；还有一些功能涉及多个业务对象——对于这两种功能，最好还是不包含到业务对象中，而是以应用服务的形式实现，以避免产生臃肿的业务对象。

## 示例代码

以下代码是业务对象模式的示例。

### 实现POJO业务对象

本例示范了POJO业务对象策略。CustomerBO（客户业务对象）和ContactInfoBO（联系人

---

<sup>⊖</sup> 对象-关系型数据库映射：object-relational mapping，简称O-R Mapping，是采用中间层，把对象透明地映射到关系型数据库的存储的一种做法。

业务对象) 的示例代码如例7.21和例7.22所示。

### 例7.21 业务对象: CustomerBO.java

```

1
2  public class CustomerBO {
3      // CustomerData (客户数据) 是数据对象
4      private CustomerData customerData;
5
6      // ContactInfoBO (联系人业务对象) 是从属业务对象
7      private ContactInfoBO contactInfoBO;
8
9      public CustomerBO(CustomerData customerData) {
10         // 验证CustomerData的值
11         . . .
12         // 把客户数据复制到这个对象中
13         this.customerData = customerData;
14     }
15
16     public ContactInfoBO getContactInfoBO () {
17         // 如果ContactInfoBO没有创建, 那么就从
18         // 客户数据中获取联系人数据, 并用ContactInfoBO 把它
19         // 包装起来
20         if (contactInfoBO == null)
21             contactInfoBO = new ContactInfoBO(
22                 customerData.getContactInfoData());
23         return contactInfoBO;
24     }
25
26     // CustomerBO的业务方法
27     . . .
28 }
```

387

### 例7.22 业务对象: ContactInfoBO.java

```

1
2  public class ContactInfoBO {
3      // ContactInfoData (联系人信息数据) 是数据对象
4      private ContactInfoData contactInfoData;
5
6      public ContactInfoBO(ContactInfoData contactInfoData) {
7          // 验证ContactInfoData的值
8          . . .
9          // 把联系人信息复制到这个对象中
10         this.contactInfoData = contactInfoData;
11     }
12
13     // 这个方法返回地址信息
14     public AddressData getAddressData () {
15         return contactInfoData.getAddressData();
```

```

16      }
17
18      // ContactInfoBO的业务方法
19      ...
20  }

```

## 实现Entity Bean业务对象

后面的复合实体模式介绍了用entity bean实现业务对象的方法。请参考复合实体模式的示例代码部分。

## 相关模式

- **复合实体**

**388** 可以使用EJB 2.x里的本地entity bean来实现业务对象。我们推荐使用本地entity bean而不是远程entity bean实现业务对象。复合实体模式讨论了使用entity bean实现业务对象的细节。

- **业务服务**

与单个业务对象相关的独立业务逻辑和业务规则通常在业务对象内部实现。但是，在大多数应用系统中，可能还会有一些业务逻辑要涉及多个业务对象。另外，在业务对象固有实现的一些功能之上，可能还需要提供一些针对特定的用例、客户端或通信渠道的业务功能。使用应用服务，能够实现涉及多个业务对象的业务逻辑，这样也就为业务对象提供了一个服务封装层。

- **传输对象**

可以使用传输对象为业务对象传出/传入数据。有些开发者倾向于用传输对象来表现业务对象的内部状态。但是，我们不提倡使用业务对象包装传输对象，因为这违背了传输对象模式的设计意图，而且使这两种对象之间产生了紧耦合（见稍后的设计手记：“业务对象包装传输对象”部分）。

- **数据访问对象**

可以使用数据访问对象实现业务对象的持久化存储，而底层则使用定制JDBC机制实现。

- **业务领域存储**

可以使用业务领域存储来实现业务对象的持久化，这样就能利用定制的透明持久化机制的强大功能（这种机制与标准JDO实现有类似之处）。

- **事务脚本和业务领域模型 [PEAA]**

事务脚本（Transaction Script）模式考察了使用面向过程的代码实现业务逻辑的做法，而业务领域模型（Domain Model）模式则是用面向对象的方式实现业务逻辑。业务领域模型和业务对象非常相似，但是对于实践中的开发者和架构师来说，“业务对象”的说法更为常用、更为准确。我们见过很多项目都广泛使用“业务对象”这个说法，而且这些用法在概念上确实是与本模式一致的。

### 设计手记：业务对象包装传输对象<sup>①</sup>

有些开发者倾向于使用业务对象包装传输对象，因为业务对象需要维护的状态信息与传输对象体现的状态信息十分接近。开发者愿意这样做的另一个原因是，他们觉得传输对象会使系统膨胀，而如果能够以某种方式重用它们，也许就能够限制这种膨胀，从而达到多方面利用传输对象的目的。但是，出于以下理由，这个道理是说不通的：

- 这符合传输对象的设计意图吗？

传输对象模式原本的设计意图是在各层次之间传送数据，而不是给业务对象的状态建模。所以，如果采用业务对象包装传输对象的策略，就违背了这个设计意图，因为它给传输对象添加了额外的含义。如果传输对象的重用要比忠实于设计意图还重要，可能就会倾向于使用这个策略。但是无论如何，传输对象的本意是用来满足与用例相关的数据传输需要的，所以也极为忠实地反映了用例本身的需求。

- 要是传输对象添加或者删除一个字段会怎么样？

传输对象的修改可能也会导致业务对象的修改——如果传输对象的修改也要反映给业务对象的客户端的话。举例来说，如果删除了一个字段，那么业务对象可能也需要作相应的修改，才能反映这个改动。因此，这种修改就会在整个业务对象层、以及该层以外的部分引起广泛的连锁反应。

- 要是业务对象添加或者删除一个字段会怎么样？

如果给业务对象添加一个字段、或者删除一个现存字段，那么传输对象也必须做出相应改动，才能保持一致。这种改动会产生瀑布效应，造成业务对象和传输对象的所有客户端都要做出改动。

- 要是业务对象需要把另一个传输对象（不同于它所包装的那个传输对象）发送给客户端，会怎么样？

虽然业务对象已经包装了一个传输对象，但此时它只能按照客户端的请求创建一个新的传输对象。这样一来不仅没有限制传输对象的膨胀，反而加剧了这种膨胀。

- 要是业务对象需要给客户端传送一些衍生字段<sup>②</sup>，又该怎么办？

这意味着业务对象为了传递这些衍生数据，还要创建另一个传输对象。或者，也可以修改现有的传输对象，加入新的字段，用以传输衍生数据。这也就产生了前面所说的修改传输对象会造成的问题。

390

## 复合实体

### 问题

需要用entity bean实现业务领域概念模型。

所谓业务对象，就是包含业务逻辑和业务状态的对象。在J2EE应用系统中，可以用entity

<sup>①</sup> 所谓包装，就是用传输对象表现业务对象的内部状态。

<sup>②</sup> 所谓衍生字段（derived field），并不是说这些字段是从父类继承来的，而是指这些字段的值是通过计算其他字段的值而得到的。

bean实现业务对象；但是在用entity bean实现业务对象，会产生一些问题。

第一个问题是，需要决定是使用远程entity bean还是本地entity bean。远程entity bean会增加网络负载，所以如果使用不当就会降低系统性能。EJB2.0技术规范引入了本地entity bean，但对于只支持EJB 1.1版本的系统实现来说也许就不可用。有些EJB1.1容器会优化entity bean之间的调用；但是，这种优化是厂商自己提供的，并不标准。在EJB 2.x的容器中，本地entity bean和它的客户端处于同一虚拟机中，这就能进一步控制系统性能和网络负载。但是，还是需要注意，与POJO业务对象相比，entity bean即使处于本地也还不够高效，因为容器还要透明地提供很多优化，另外，对于entity bean的每个调用，容器都要提供很多其他服务，比如安全管理、事务管理等等。

另一个问题是，如何实现entity bean业务对象的持久化存储。当然，可以使用“容器-管理持久化（CMP）”，实现透明的持久化，无须自己编写持久化代码。但是，如果有一个以前遗留的持久化实现，或者需要使用自己的持久化机制实现某些独特的需求，那么CMP可能就不太合适了。这样一来，就要实现“bean-管理持久化（BMP）”，把业务对象保存到数据存储中。

最后，还要考虑一个问题：业务对象的关系使用“容器-管理关系（CMP）”实现，还是用“bean-管理关系（BMP）”实现。只有使用了CMP，才能用上CMR，另外CMR还要求所有相互关联的entity bean都得是本地entity bean。

[391]

## 约束

- 需要避免远程entity bean的缺点（比如网络负载和远程的entity bean关系）。
- 需要用定制的或者是以前遗留的持久化实现，完成bean-管理持久化（BMP）。
- 在使用entity bean实现业务对象时，需要高效地实现对象之间的父-子关系。
- 需要用entity bean封装并聚合现存的POJO业务对象。
- 需要充分利用EJB容器的事务管理和安全功能。
- 需要把数据库的物理设计对客户端封装起来。

## 解决方案

使用复合实体，结合本地entity bean和POJO，实现业务对象的持久化。复合实体能够把一组相互关联的业务对象聚合为粗粒度的entity bean实现。

在J2EE应用系统中，业务对象被实现为父对象或从属对象。

- 父对象可以重用、可以单独部署，它管理自己的生命周期，并且管理它与其他对象的关系。每个父对象可能会包含一个以上的对象，并管理这些对象的生命周期，这些对象称为从属对象。
- 从属对象可能包含其他从属对象，也可能不包含。在从属对象的生命周期和它的父对象的生命周期之间存在紧耦合。从属对象并不直接暴露给客户端，客户端只能通过父对象访问它们。从属对象并不独立存在，它们的识别和管理都要依赖于父对象。

使用复合entity bean，就能够实现父对象和相关的从属对象。业务对象的聚合，能够结合互

相关联的持久化对象（比如订单和订单中的货品），形成一个粗粒度的系统实现，这样也就可能减少应用系统中entity bean的数量。

在EJB 1.1的技术规范中，entity bean只能实现为远程对象。这就意味着要面对各种分布式组件的常见问题，其中主要包括网络负载和对象粒度。在这种情况下，应该把父对象实现为一个远程entity bean，把从属对象实现为POJO。如果把从属对象也实现为entity bean，那可并不高明，因为这会降低应用系统的性能。之所以如此，是由于如果entity bean之间存在父-子关系，那么这些entity bean之间的通信就会增加网络负载。[392]

在EJB 2.x版本中，即可以把entity bean实现为远程对象，也可以实现为本地对象。使用EJB 2.x实现entity bean的时候，应该以本地entity bean的形式实现父业务对象。至于从属对象，则既可以实现为POJO（和EJB 1.1的情况一样），也可以实现为本地entity bean。而父业务对象和从属对象之间的关系，同样既可以使用bean-管理关系实现（如果使用了BMP），也可以使用CMR实现（如果使用了CMP）。如果用CMR实现，那么从属对象就不能是POJO，只能是本地entity bean。

为了实现业务对象的持久化存储，必须选择是采用容器-管理持久化（CMP）还是bean-管理持久化（BMP）。一些新的应用系统会使用与EJB 2.x兼容的容器，CMP越来越受到这些系统的欢迎。虽然EJB 2.x的CMP比EJB 1.x中的要复杂的多，但是它也并不是总能符合性能上的需求。而且，可能还有一些独特的需求，或者已经有遗留的持久化方案，这样就没法使用CMP满足需求了。在这些情况下，必须采用BMP来实现持久化。

使用复合实体，能够在应用系统中充分利用entity bean的优势和EJB架构的一些强大功能，比如容器-管理事务、安全和持久化等。

无论使用远程对象还是本地对象实现复合实体，都不应该直接把entity bean暴露给客户端。应该把所有的entity bean封装在会话门面后面，让客户端访问会话门面，会话门面再与幕后的相关复合实体交互。

有些应用系统的业务逻辑交互很少，业务逻辑很简单，这样可能就会直接把复合实体暴露给客户端。但是，这样做时候应该注意，因为如果应用系统向复杂方向发展，就会遇到诸如系统的可维护性之类的问题。

## 结构

图7-30中的类图体现了复合实体模式的结构。

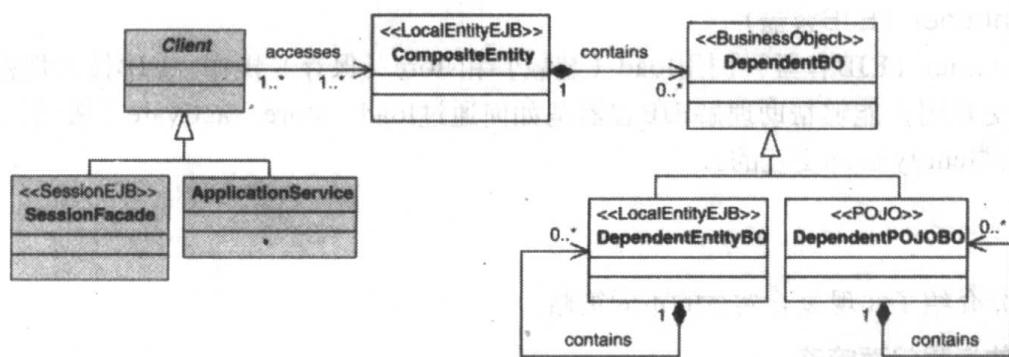


图7-30 复合实体模式类图

图7-31中的序列图体现了复合实体模式中的交互。

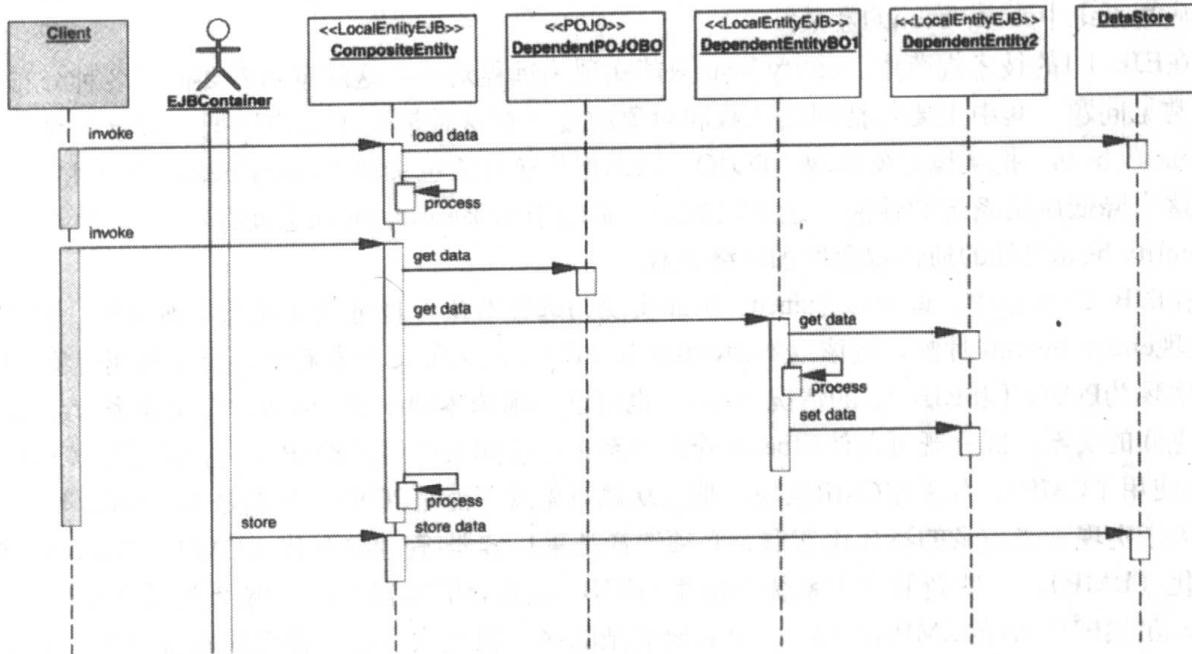


图7-31 复合实体序列图

## 参与者和责任

### CompositeEntity (复合实体)

394 CompositeEntity (复合实体) 是一种粗粒度的entity bean，其中还包含从属对象。

DependentBO (从属业务对象)、DependentEntityBO (从属实体业务对象) 和 DependentPOJOBO (从属POJO业务对象)

从属对象依靠父对象管理生命周期。从属对象还可以包含其他从属对象；这样一来，整个复合实体中就可能包含一棵对象树。可以把从属对象实现为本地 entity bean (比如 DependentEntity1 和 DependentEntity2)，也可以实现为POJO (如DependentPOJO)。

### DataStore (数据存储)

DataStore (数据存储) 代表了为业务数据提供持久化的持久化存储。

### EJBContainer (EJB容器)

EJBContainer (EJB容器) 调用load (装载) 和store (保存) 操作。EJB技术规范中有一些详细的对象交互图，能够帮助理解EJB容器是如何通过load、store、activate (激活)、passivate (钝化) 方法与entity bean交互的。

## 策略

这一部分介绍了实现复合实体的不同策略。

### 复合实体远程界面策略

如果应用系统中的业务逻辑非常简单、甚至根本不存在，那么业务对象就可以实现系统中

所有的业务逻辑和业务规则。对于这样的应用系统，使用会话门面封装业务对象复合entity bean，似乎会引入一个多余的session bean层。会话门面只是复合实体的一个代理，没有带来太大价值。

在这种情况下，远程客户端可以直接访问复合实体。复合实体中的父对象用一个远程entity bean实现，从属对象则实现为本地entity bean或POJO。复合实体（也就是远程entity bean父对象）起到了门面的作用，它封装了父对象和从属对象，把一个比较简单的接口暴露给客户端。但是随着应用系统越来越复杂，这个做法也就会导致很多问题。开发者们会（错误地）把新的业务对象都实现为远程entity bean，而不按照我们推荐的做法，采用会话门面封装entity bean。

必须注意，这个策略很少被人使用。即使是最简单的应用系统，往往也会有一个由应用服务和会话门面组成的服务层。

395

### 设计手记：实体门面（见图7-32）

业务对象实现了复杂的关系，封装了从属业务对象。实际上，对于它所封装的所有对象来说，它起着门面 [GoF] 的作用。在不使用entity bean的J2EE 应用系统中，采用POJO实现业务对象。

在使用entity bean的J2EE应用系统中，业务对象是用复合实体实现的，通常的做法是采用本地entity bean和POJO从属对象。但是，如果复合实体远程门面策略适合需要，也可以使用远程entity bean实现复合实体的父对象，由本地entity bean或POJO实现从属对象。

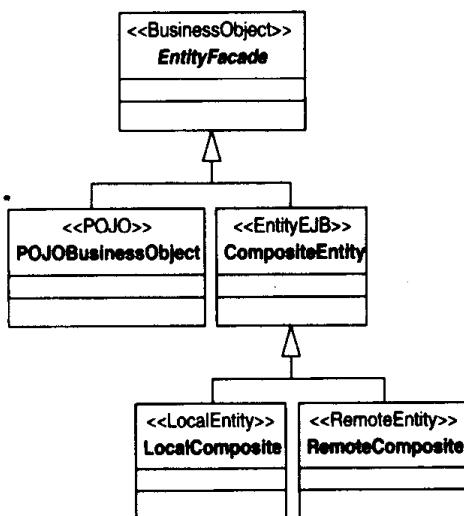


图7-32 实体门面类图

### 复合实体BMP策略

如果使用BMP实现复合实体，可能就要考虑采取一些策略优化实现。以下的两个策略：懒装载策略和存储优化（脏数据标示器）策略，就与BMP复合实体实现有关。

#### 懒装载策略

在复合实体下属的对象树中，可以包含许多层次的从属对象。但是，当容器调用复合实体的ejbLoad()方法时，如果直接装载所有这些从属对象，就会花费不少时间和资源。

396

一种优化的方法是，使用一种“懒”装载策略装载这些从属对象。ejbLoad()被调用的时候，

首先只装载那些对于复合实体客户端最为重要的从属对象。此后，如果客户端要访问一个从属对象，而它还没有被从数据库装载过，那么复合实体可以再按需执行装载操作。

因此，在初始化的过程中，如果一些从属对象没有被用到，那也就不会装载它们。但是，如果此后客户端需要这些从属对象，就可以到那时再装载它们。从属对象一旦被装载，此后容器对ejbLoad()方法的调用也必须重新装载这些从属对象，这样才能与持久化存储中的变化同步。

### 存储优化（脏数据标示器）策略

使用bean-管理持久化的时候，在调用ejbStore方法保存整个对象拓扑结构的过程中，会发生一个常见问题。因为EJB容器没法知道entity bean及其从属对象中哪些数据被修改了，所以就要由开发者负责判断保存哪些数据以及怎样保存这些数据。

有些EJB容器具备这样一种功能：它能够判断复合实体的对象拓扑结构中的哪些对象已经被更新过，需要执行保存。开发者要在从属对象上实现一个特别的方法，比如就叫isDirty()，容器调用这个方法来判断自上次ejbStore操作之后对象是否又更新过。

另外还有一个通用的解决方案，那就是使用一个叫DirtyMarker（脏数据标示器）的接口，类图如图7-33所示。

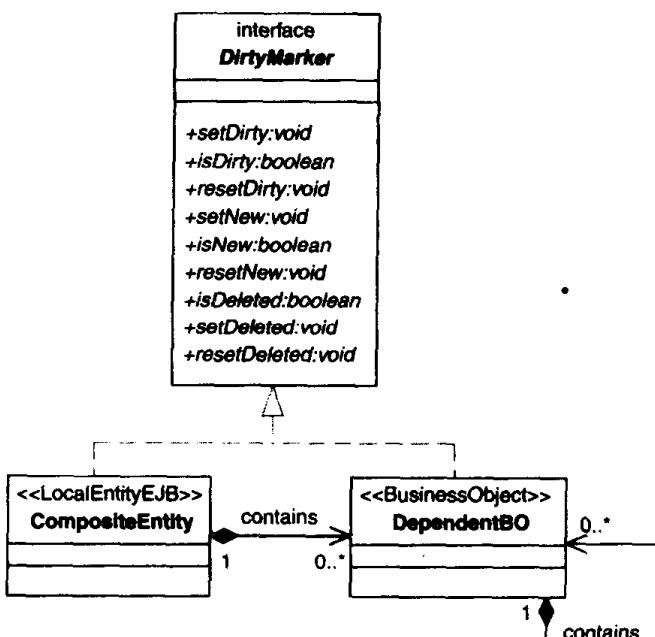


图7-33 存储优化策略类图

这里的实际做法是，让从属对象实现DirtyMarker（脏数据标示器）接口，然后调用者（通常就是ejbStore()方法）就能知道从属对象的状态是否已被修改。这样，调用者就能够决定是否要保存从属对象的数据。

图7-34是本策略中交互的一个例子。

客户端对CompositeEntity（复合实体）执行更新操作，导致了从属对象DependentObject3发生改变。DependentObject3是通过父对象DependentObject2被访问的。CompositeEntity是DependentObject2的父对象。更新操作执行后，就会调用DependentObject3上的setDirty()方法。

此后，当容器调用该CompositeEntity实例的ejbStore()方法时，ejbStore()就能够判断，哪些从属对象变“脏”<sup>⊖</sup>了，然后就可以选择性地把这些修改保存到数据库中。保存成功后，“脏数据标志”也就被重置了。

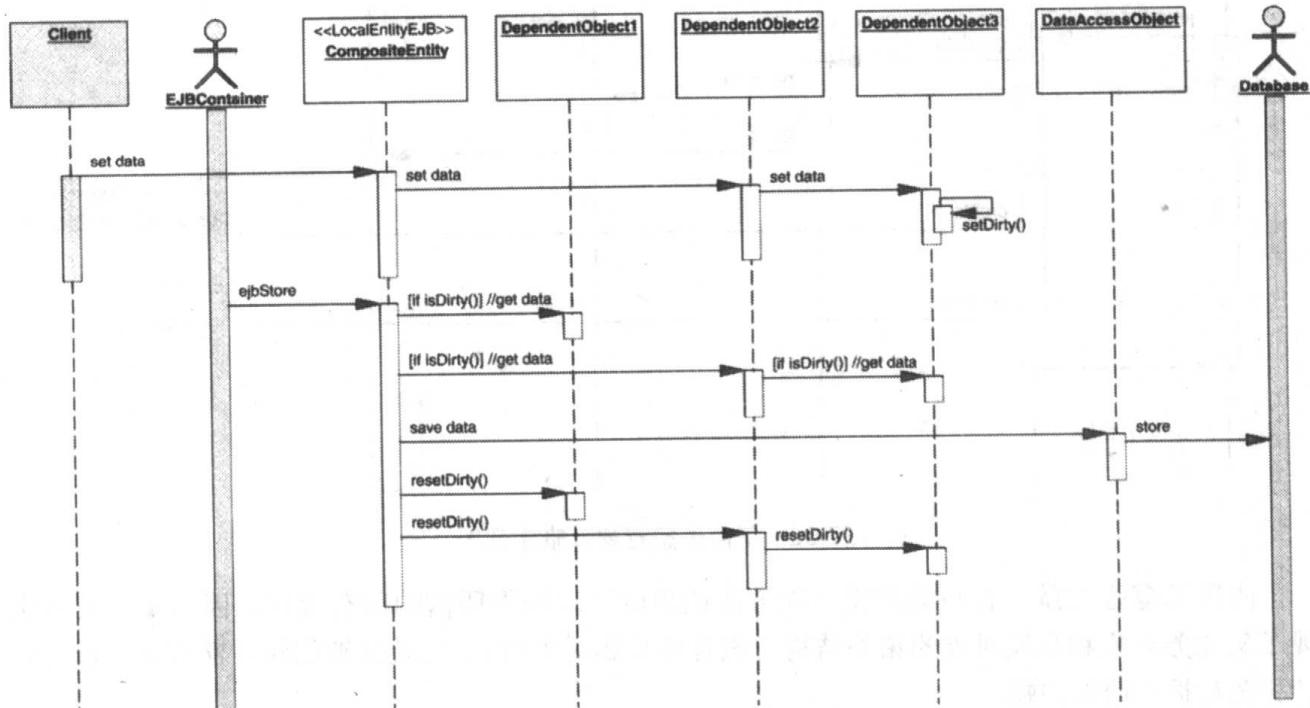


图7-34 存储优化策略序列图

DirtyMarker（脏数据标示器）还可以包含其他的一些方法，用于识别从属对象的其他持久化状态。比如，如果复合实体中加入了一个新的从属对象，ejbStore()方法就应该能够识别出来，并且进行相应操作——在这个情况下，从属对象并不是“脏”的，只不过是一个新对象而已。所以，可以扩展DirtyMarker接口，加入一个叫isNew()的方法，这样ejbStore()方法发现了新数据，就可以调用数据库的“插入”操作，而不是“更新”操作。与此类似，该接口还可以加入一个叫isDeleted()的方法，这样ejbStore()在判断出现删除后，也就能调用“删除”操作。

如果在调用ejbStore()方法的时候复合实体并没有受到更新，那么从属对象也就都没有被更新。

容器调用ejbStore()方法的时候，可能会导致把整个从属对象树都保存到数据库中，造成巨大的负载压力；本策略就能够避免这种情况发生。

如果有大量的从属对象，还可以进一步做出优化，可以让父对象维护一个集合，其中包含所有“脏”了的从属对象的引用。当保存复合实体的时候，遍历这个集合比检查所有从属对象判断谁变“脏”了要快。

### 复合传输对象策略

本策略的序列图如图7-35所示。

<sup>⊖</sup> 所谓“脏”，当然是指数据被修改过了。

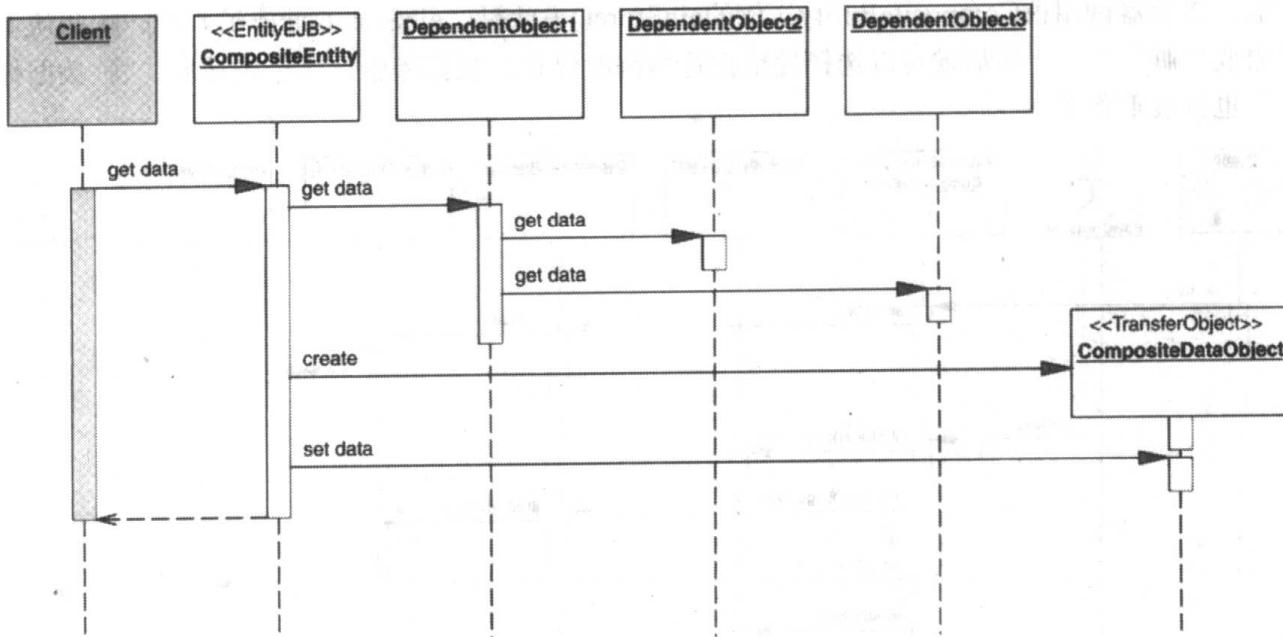


图7-35 复合传输对象策略序列图

使用了复合实体，客户端只需一次方法调用就可以获得所需的所有数据。因为复合实体实现了父业务对象和从属对象的拓扑结构（或者说从属对象树），它就能够创建传输对象，通过它把相关数据传给客户端。

传输对象可以是一个简单对象，也可以是一个具有子对象的（也就是包含了拓扑结构的）复合对象，究竟是哪种情况，还要由客户端请求的数据来决定。传输对象被串行化后，按值传递给客户端。  
400

我们推荐把复合实体实现为本地entity bean，本地EJB的返回值是按值传递的。这样，把传输对象返回给客户端（通常是会话界面）就避免了数据被客户端改动。

## 效果

- 增加了可维护性

如果使用复合实体模式，把业务对象中的父对象实现为entity bean，从属对象则实现为POJO，那么就能够减少细粒度entity bean的数量。如果使用的是EJB 2.x，可以把从属对象也实现为本地entity bean，以便利用EJB的其他特性，比如CMR和CMP。这样就能够提高应用系统的可维护性。

- 提高了网络性能

把业务对象中的父对象和从属对象聚合成数量较少的粗粒度entity bean，在EJB 1.1的环境下能够提高系统的整体性能。这种做法能降低网络负载，因为它消除了entity bean之间的通信。对于EJB 2.x，用本地entity bean把业务对象实现为复合实体，也具有这样的优点，因为这样一来所有与entity bean的交互都与客户端同处于本地了。但是应该记住，即使是这种“本地”部署，也还是不如POJO业务对象的做法高效，因为EJB容器还要负责entity bean的生命周期、安全性、事

务管理等服务。

- **减少了对数据库方案的依赖**

复合实体为数据库中的数据提供了一种对象化的视图。这样，数据库的方案（schema）<sup>①</sup>对客户端是隐藏的，因为entity bean与数据库方案之间的映射被封装在复合实体中了。对数据库的修改可能会要求修改复合实体bean。但是客户端并不会受到影响，因为复合实体没有把数据库方案暴露给外界。

- **增大了对象粒度**

使用了复合实体，客户端通常就只需要对父对象寻址，而不必定位大量细粒度的从属entity bean。父对象对于从属对象起到了门面[GoF]的作用，暴露出一个较为简单的接口，隐藏了从属对象的复杂性。复合实体避免了对从属对象的细粒度方法调用，降低了网络负载。

401

- **简化了复合传输对象的创建**

复合实体能够创建一个包含了该entity bean及其从属对象的所有数据的复合传输对象，而且只需一个方法调用，就能把这个传输对象返回给客户端。这减少了客户端和复合实体之间的远程调用次数。

## 示例代码

考虑一个专业服务自动化应用系统（PSA）<sup>②</sup>，其中业务对象Resource（资源）是用复合实体实现的。Resource对象体现的事实是专业服务公司的一个雇员（一个咨询顾问），作为“资源”被指派给一个项目。每个Resource对象可以有以下从属对象：

- BlockOutTime（缺勤时间）——这是Resource（资源）出于种种原因（比如参加培训、假期、请假等等）而不可用的时间。由于每个资源都可能在多个时间缺勤，所以Resource-对-BlockOutTime关系是一种“一对多”的关系。
- SkillSet（技能集）——体现的是一个Resource（资源，也就是一个咨询专家）所具有的Skill（技能）。由于每个资源都可能有多种技能，所以Resource与SkillSet之间的关系是一种“一对多”的关系。

## 实现复合实体模式

Resource（资源）业务对象是通过复合实体模式（ResourceEntity——资源实体）实现的，如例7.23所示。该entity bean与从属对象（BlockOutTime 和 SkillSet对象）的“一对多”关系是用集合实现的。

### 例7.23 ResourceEntityBean.java: 复合实体<sup>③</sup>

```

1
2 package corepatterns.apps.psa.ejb;
3

```

<sup>①</sup> 数据库方案，指数据库的设计、数据表的结构等。

<sup>②</sup> 这是全书的一个常见例子。首次出现是在本章的“业务代表”一节的示例代码中。

<sup>③</sup> 以下示例代码中的缩写“TO”，都是传输对象（Transfer Object）的简称。

4 import corepatterns.apps.psa.core.\*;  
5 import corepatterns.apps.psa.dao.\*;  
6 import java.sql.\*;  
7  
8 import javax.sql.\*;  
9 import java.util.\*;  
10 import javax.ejb.\*;  
11 import javax.naming.\*;  
12  
13 public class ResourceEntityBean implements EntityBean {  
14 public String employeeId;  
15 public String lastName;  
16 public String firstName;  
17 public String departmentId;  
18 ...  
19  
20 // BlockOutTime (缺勤时间集) 从属对象的集合  
21 public Collection blockOutTimes;  
22  
23 // SkillSet (技能集) 从属对象的集合  
24 public Collection skillSets;  
25  
26 ...  
27  
28 private EntityContext context;  
29  
30 // Entity Bean方法实现  
31 public String ejbCreate(ResourceTO resource)  
32 throws CreateException {  
33 try {  
34 this.employeeId = resource.employeeId;  
35 setResourceData(resource);  
36 getResourceDAO().create(resource);  
37 } catch (Exception ex) {  
38 throw new EJBException("Reason:" + ...);  
39 }  
40 return this.employeeId;  
41 }  
42  
43 public String ejbFindByPrimaryKey(String primaryKey)  
44 throws FinderException {  
45 boolean result;  
46 try {  
47 ResourceDAO resourceDAO = getResourceDAO();  
48 result = resourceDAO.findResource(primaryKey);  
49 } catch (Exception ex) {  
50 throw new EJBException("Reason:" + ...);  
51 }  
52 }  
53 }

402

```
51      }
52      if (result) {
53          return primaryKey;
54      }
55      else {
56          throw new ObjectNotFoundException(...);
57      }
58  }
59
60  public void ejbRemove() {
61      try {
62          // 删除从属对象
63          if (this.skillSets != null) {
64              SkillSetDAO skillSetDAO = getSkillSetDAO();
65              skillSetDAO.deleteAll(employeeId);
66              skillSets = null;
67          }
68          if (this.blockoutTime != null) {
69              BlockOutTimeDAO blockouttimeDAO =
70                  getBlockOutTimeDAO();
71              blockouttimeDAO.deleteAll(employeeId);
72              blockOutTimes = null;
73          }
74
75          // 从持久化存储中删除资源
76          ResourceDAO resourceDAO = new ResourceDAO();
77          resourceDAO.delete(employeeId);
78      } catch (ResourceException ex) {
79          throw new EJBException("Reason:" + ...);
80      } catch (BlockOutTimeException ex) {
81          throw new EJBException("Reason:" + ...);
82      } catch (Exception exception) {
83          ...
84      }
85  }
86
87
88  public void setEntityContext(EntityContext context) {
89      this.context = context;
90  }
91
92  public void unsetEntityContext() {
93      context = null;
94  }
95
96  public void ejbActivate() {
97      employeeId = (String)context.getPrimaryKey();
```

403

404

```

98      }
99
100     public void ejbPassivate() {
101         employeeId = null;
102     }
103
104     public void ejbLoad() {
105         try {
106             // 装载资源
107             ResourceDAO resourceDAO = getResourceDAO();
108             setResourceData((ResourceTO)
109                 resourceDAO.findResource(employeeId));
110
111             //如果必要，装载其他从属对象
112             ...
113         } catch (Exception ex) {
114             throw new EJBException("Reason:" + ...);
115         }
116     }
117
118     public void ejbStore() {
119         try {
120             // 存储资源信息
121             getResourceDAO().update(getResourceData());
122
123             // 如果必要，保存从属对象
124             ...
125         } catch (SkillSetException ex) {
126             throw new EJBException("Reason:" + ...);
127         } catch (BlockOutTimeException ex) {
128             throw new EJBException("Reason:" + ...);
129         }
130         ...
131     }
132
133     public void ejbPostCreate(ResourceTO resource) {
134     }
135
136     // 用于获取Resource（资源）传输对象的方法
137     public ResourceTO getResourceTO() {
138         // 创建一个新的Resource传输对象
139         ResourceTO resourceTO = new ResourceTO(employeeId);
140
141         // 复制所有值copy all values
142         resourceTO.lastName = lastName;
143         resourceTO.firstName = firstName;
144         resourceTO.departmentId = departmentId;

```

```

145     ...
146     return resourceTO;
147 }
148
149 public void setResourceData(ResourceTO resourceTO) {
150     // 把传输对象中的值复制给entity bean
151     employeeId = resourceTO.employeeId;
152     lastName = resourceTO.lastName;
153     ...
154 }
155
156 // 本方法获取从属传输对象
157 public Collection getSkillSetsData() {
158     // 如果skillSets尚未装载，那么首先执行装载。
159     // 参见对懒装载策略的实现
160     return skillSets;
161 }
162 ...
163
164 // 其他get/set方法（如果需要的话）
165 ...
166
167 // Entity bean业务方法
168 public void addBlockOutTimes(Collection moreBOTS)
169 throws BlockOutTimeException {
170     // 注意：moreBOTS是BlockOutTimeTO对象的
171     // 集合
172     try {
173         Iterator moreIter = moreBOTS.iterator();
174         while (moreIter.hasNext()) {
175             BlockOutTimeTO botTO =
176                 (BlockOutTimeTO) moreIter.next();
177             if (! (blockOutTimeExists(botTO))) {
178                 // 把BlockOutTimeTO添加到集合中
179                 botTO.setNew();
180                 blockOutTime.add(botTO);
181             } else {
182                 // BlockOutTimeTO已经存在了，不能添加
183                 throw new BlockOutTimeException(...);
184             }
185         }
186     } catch (Exception exception) {
187         throw new EJBException(...);
188     }
189 }
190
191 public void addSkillSet(Collection moreSkills)

```

```

192     throws SkillSetException {
193         // 与addBlockOutTime()方法的实现相似
194         ...
195     }
196
197     ...
198
199     public void updateBlockOutTime(Collection updBOTs)
200     throws BlockOutTimeException {
201         try {
202             Iterator botIter = blockOutTimes.iterator();
203             Iterator updIter = updBOTs.iterator();
204             while (updIter.hasNext()) {
205                 BlockOutTimeTO botTO =
206                     (BlockOutTimeTO) updIter.next();
207                 while (botIter.hasNext()) {
208                     BlockOutTimeTO existingBOT =
209                         (BlockOutTimeTO) botIter.next();
210                     // 比较键值, 定位该BlockOutTime
211                     if (existingBOT.equals(botTO)) {
212                         // 在集合中找到了该BlockOutTime
213                         // 把旧的BlockOutTimeTO替换成新的
214                         botTO.setDirty(); //修改过的旧从属对象
215                         botTO.resetNew(); //不是一个新的从属对象了
216                         existingBOT = botTO;
217                     }
218                 }
219             }
220
221         } catch (Exception exc) {
222             throw new EJBException(...);
223         }
224     }
225
226     public void updateSkillSet(Collection updSkills)
227     throws CommitmentException {
228         // 与updateBlockOutTime方法类似...
229         ...
230     }
231
232     ...
233
234 }

```

407

## 实现懒装载策略

考虑这样一种情况，当容器第一次调用ejbLoad()方法装载复合实体的数据时，只装载“资

源”本身的数据。也就是说，只装载ResourceEntity的各个属性，而不包括从属对象的集合。只是在客户端调用某个业务方法，需要用到从属对象的时候，才装载从属对象的数据。而ejbLoad()方法需要跟踪这样装载的从属对象，在重新装载的时候还要包括它们。

ResourceEntity类中的相关方法如例7.24所示。

#### 例7.24 实现懒装载策略

```

1
2     ...
3     public Collection getSkillSetsData() {
4         throws SkillSetException {
5             checkSkillSetLoad();
6             return skillSets;
7         }
8
9     private void checkSkillSetLoad()
10    throws SkillSetException {
11        try {
12            // 懒装载策略...按需装载
13            if (skillSets == null)
14                skillSets =
15                    getSkillSetDAO().findAllSkills(resourceId);
16        } catch (Exception exception) {
17            // 没有技能，抛出异常
18            throw new SkillSetException(...);
19        }
20    }
21
22    ...
23
24    public void ejbLoad() {
25        try {
26            // 装载资源
27            ResourceDAO resourceDAO = new ResourceDAO();
28            setResourceData(
29                resourceDAO.findResource(employeeId));
30
31            // 如果懒装载对象已经被装载了,
32            // 那么就要重新装载它们。
33            // 如果没有装载它们，那也就不必在这里装载...
34            // 以后会用懒装载方式装载它们。
35            if (skillSets != null) {
36                reloadSkillSets();
37            }
38            if (blockOutTimes != null) {
39                reloadBlockOutTimes();
40            }
41            ...

```

```

42         throw new EJBException("Reason:" + ...);
43     }
44 }
45 ...
46 ...
47

```

## 实现存储优化(脏数据标示器)策略

为了使用存储优化策略，从属对象需要实现DirtyMarker（脏数据标示器）接口，如例7.25所示。在这个策略中，ejbStore()方法只会保存标示为“已修改”的数据，代码如例7.26所示。

**例7.25 SkillSet（技能集）从属对象实现了DirtyMarker（脏数据标示器）接口**

```

1  public class SkillSetTO
2  implements DirtyMarker, java.io.Serializable {
3      private String skillName;
4      private String expertiseLevel;
5      private String info;
6
7      ...
8
9      // 脏数据标志位
10     private boolean dirty = false;
11
12     // 新数据标志位
13     private boolean isNew = true;
14
15     // 数据删除标志位
16     private boolean deleted = false;
17
18     public SkillSetTO(...) {
19         // 初始化
20         ...
21         // 新传输对象
22         setNew();
23     }
24
25     // SkillSet的get/set方法和其他方法。
26     // 所有set方法，以及所有其他涉及修改的方法
27     // 都必须调用setDirty()
28     public setSkillName(String newSkillName) {
29         skillName = newSkillName;
30         setDirty();
31     }
32     ...
33
34     // DirtyMarker（脏数据标示器）方法

```

```

35 // 以下几个方法只用于已修改的传输对象
36
37 public void setDirty() {
38     dirty = true;
39 }
40 public void resetDirty() {
41     dirty = false;
42 }
43 public boolean isDirty() {
44     return dirty;
45 }
46
47 // 以下几个方法只用于新的传输对象
48 public void setNew() {
49     isnew = true;
50 }
51 public void resetNew() {
52     isnew = false;
53 }
54 public boolean isNew() {
55     return isnew;
56 }
57
58 // 以下几个方法只用于已删除的传输对象
59 public void setDeleted() {
60     deleted = true;
61 }
62 public boolean isDeleted() {
63     return deleted;
64 }
65 public void resetDeleted() {
66     deleted = false;
67 }
68
69 }

```

410

### 例7.26 实现存储优化

```

1
2 ...
3
4 public void ejbStore() {
5     try {
6         // 保存资源对象的数据
7         getResourceDAO().update(getResourceData());
8
9         // 从属对象的存储优化
10        // 检查数据是否为“脏”并且

```

```

11      // 存储已修改的数据
12      if (skillSets != null) {
13          // 获取用于存储的DAO（数据访问对象）
14          SkillSetDAO skillSetDAO = getSkillSetDAO();
15          Iterator skillIter = skillSet.iterator();
16          while (skillIter.hasNext()) {
17              SkillSetTO skill =
18                  (SkillSetTO) skillIter.next();
19              if (skill.isNew()) {
20                  // 这是一个新的从属对象，把它加入存储
21                  skillSetDAO.create(skill);
22                  skill.resetNew();
23                  skill.resetDirty();
24              }
25              else if (skill.isDeleted()) {
26                  // 删除该技能
27                  skillSetDAO.delete(skill);
28                  // 从属对象集合中也要删除该技能
29                  skillSets.remove(skill);
30              }
31              else if (skill.isDirty()) {
32                  // 该技能已修改，需要保存
33                  skillSetDAO.update(skill);
34                  // 保存完毕，重置脏数据标志。
35                  skill.resetDirty();
36                  skill.resetNew();
37              }
38          }
39      }
40
41      // 其他从属对象的
42      // 存储优化的实现也与此类似，比如
43
44      // BlockOutTime, ...
45      ...
46  } catch (SkillSetException ex) {
47
48      throw new EJBException("Reason:" + ...);
49  } catch (BlockOutTimeException ex) {
50      throw new EJBException("Reason:" + ...);
51  } catch (CommitmentException ex) {
52      throw new EJBException("Reason:" + ...);
53  }
54 }
55 ...
56 ...

```

411

## 实现复合传输对象策略

考虑这样一个需求：客户端需要获得ResourceEntity（资源实体）中的所有数据，而不仅仅是一部分数据。这可以通过复合传输对象策略实现，如例7.27所示。

412

### 例7.27 实现复合传输对象

```

1
2  public class ResourceCompositeTO {
3      private ResourceTO resourceData;
4      private Collection skillSets;
5      private Collection blockOutTimes;
6
7      // 传输对象构造函数
8      ...
9
10     // get/set方法
11     ...
12 }
```

ResourceEntity提供了一个getResourceDetailsData()方法，返回ResourceCompositeTO 复合传输对象Composite Transfer Object，如例7.28所示。

### 例7.28 创建复合传输对象

```

1
2  ...
3  public ResourceCompositeTO getResourceDetailsData() {
4      ResourceCompositeTO compositeTO =
5          new ResourceCompositeTO (getResourceData(),
6              getSkillsData(), getBlockOutTimesData());
7      return compositeTO;
8  }
9  ...
```

## 相关模式

- 业务对象

业务对象模式从总体上描述了J2EE应用系统中业务领域模型实体的实现方法。而复合实体则是采用entity bean实现业务对象的一种策略。

- 传输对象

复合实体创建一个复合传输对象，并把它返回给客户端。传输对象用于盛放复合实体及其从属对象的数据。

413

- 会话界面

一般来说，复合实体不直接暴露给应用客户端。通常使用会话界面，封装entity bean，接收对应用集成和Web Service的请求，并且给客户端提供一个简单的、粗粒度的服务接口。

- 传输对象组装器

414

复合实体返回复合传输对象，在这一点上它与传输对象组装器类似。但是，在复合实体模式中，复合传输对象中的所有数据都来自复合实体本身，而传输对象组装器的数据源则可能是不同的entity bean、session bean、数据访问对象和应用服务等。

## 传输对象

### 问题

需要跨层次传输多种数据元素<sup>①</sup>。

J2EE应用系统把服务器端的业务组件实现为会话界面和业务对象，这些组件的一些方法需要把数据返回给客户端。这些组件通常实现为远程对象，比如session bean和entity bean。如果这些业务组件暴露的是细粒度的get/set方法，客户端为了获得它需要的所有数值，就必须调用多个getter方法。

但是，这样一来会造成性能上的问题，因为对EJB的每次方法调用都可能是远程的。远程调用产生了网络负载，即使客户端和EJB容器运行在同一个JVM、操作系统或物理主机上。所以，如果只需要每次获得一组数据，还要多次调用远程对象的getter方法，那就是极为低效的。执行的远程调用越多，应用系统也就越是“啰嗦”，整个应用的性能也就越是恶化。

即使不是访问远程组件，也仍会需要访问封装在另一个层次中的组件，比如业务层中的业务对象以及集成层中的数据访问对象。虽然这些组件不是远程对象，在发送、获取数据的时候，仍然应该通过粗粒度的接口访问它们。

### 约束

- 要让客户端访问其他层中的组件，从而获取并更新数据。
- 要减少网络中的远程请求。
- 要避免“啰嗦的”、高网络负载的应用系统造成的网络性能恶化。

### 解决方案

415

#### 使用传输对象跨层次传输多种数据元素。

设计传输对象，就是要优化跨层次的数据传输。这样就可以不再逐个传输单独的数据元素，而是用一个传输对象，以单一的结构盛放请求或响应需要的所有数据元素。

传输对象按值传送<sup>②</sup>给客户端。所以，对传输对象的所有调用都作用于原始传输对象的拷贝上。

<sup>①</sup> 所谓“多种数据元素”，举例来说，就像是一个“客户”对象的“姓名”、“年龄”、“地址”等属性。客户端的一个请求需要获取该客户对象的多个属性；而在请求处理过程中，如果这些属性中的每一个都要执行一次跨层次的调用（getName()、getAge()、getAddress() 等等）才能获得，这就造成了系统的性能下降。所以要把“多种数据元素”一次传输——所以就有了传输对象。

<sup>②</sup> 按值传送，表明传递的是“数值”，而不是对象本身的“引用”。

## 结构

图7-36的类图是一种形式最简单的传输对象模式。

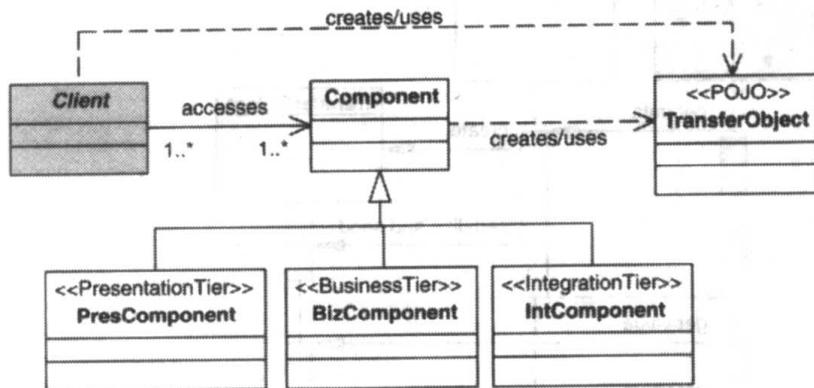


图7-36 传输对象类图

如该类图所示，TransferObject（传输对象）的一个实例由Component（组件）随需要构造，并返回给客户端。Component在接受请求后才会创建TransferObject并返回给客户端。TransferObject被串行化，然后通过网络送到客户端，Client（客户端）接收该对象，并且作为一个本地拷贝使用。同样，Client也会创建自己的TransferObject实例，并发送给Component，用以执行更新。

TransferObject（传输对象）类可以有一个接受所有必填属性<sup>①</sup>的构造函数，用来创建TransferObject的一个实例。TransferObject可以把所有的成员都声明为public，这样就能暴露所有属性，而无须使用getter/setter方法。如果需要控制TransferObject的传入/传出数值，那么就应该按需要把各个成员定义为protected或private，并且提供getter/setter方法用以访问这些数值。如果希望TransferObject不可变——也就是说，在创建之后就不能修改，那么就可以不提供setter方法。

究竟是否要把TransferObject的属性声明为private，并且由getter/setter方法访问，这个设计选择应该根据应用需求决定。

416

## 参与者和责任

图7-37的序列图表现了传输对象模式中的交互情况。

### Client（客户端）

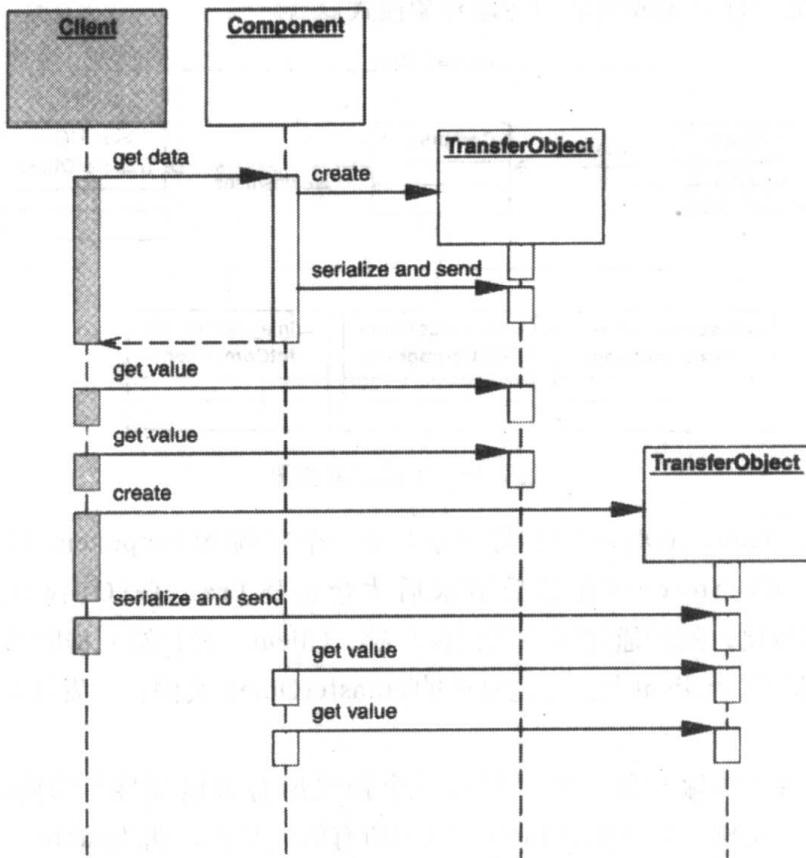
Client（客户端）需要访问一个Component（组件），从而发送/获取数据。通常，Client是处于另外一个层次的组件。比如，表现层的一个组件就可以充当业务层组件的客户端。

### Component（组件）

Component（组件）是与客户端处于不同层次的组件，客户端需要访问它，从而发送/获取

<sup>①</sup> 必填属性：也就是该对象中不能为空的字段。

数据。Component可以处于表现层（那样就叫PresComponent——表现层组件），业务层（BizComponent——业务层组件）或集成层（IntComponent——集成层组件）。



417

图7-37 传输对象序列图

### PresComponent（表现层组件）

PresComponent（表现层组件）是表现层的一个组件，比如一个助手对象、BusinessDelegate（业务代表）的一个示例、一个命令对象，等等。

### BizComponent（业务层组件）

BizComponent（业务层组件）是业务层的一个组件，比如业务对象、应用服务、服务门面等等。

### IntComponent（集成层组件）

IntComponent（集成层组件）是集成层的一个组件，比如数据访问对象等。

### TransferObject（传输对象）

TransferObject（传输对象）是一个可串行化的POJO，其中包含若干成员，能够通过一次方法调用就聚合、传递所有数据。

## 策略

前两个策略——可更新的传输对象和多重传输对象——在大多数情况下都可以使用。但是，

对于实体继承传输对象策略，则只有在Component（组件）是entity bean的时候才能使用。

### 可更新的传输对象策略

在本策略中，TransferObject（传输对象）不仅把数值从Component（组件）传输给客户端，而且还能从客户端把新数据或者对数据的改动传回Component。

图7-38表现了Component（组件）和TransferObject（传输对象）之间的关系。

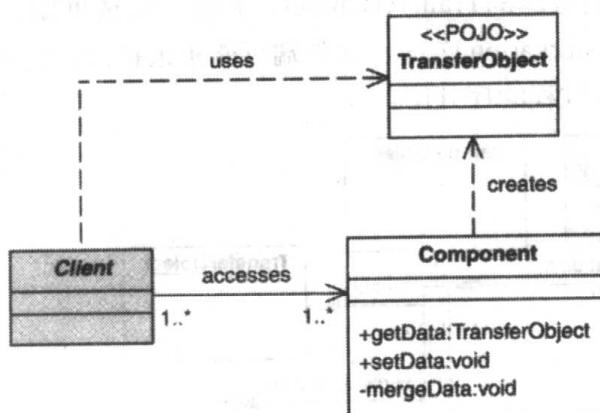


图7-38 可更新的传输对象策略类图

418

Component（组件）创建TransferObject（传输对象）。当然应该记住，客户端对Component的值的访问，不仅是要读取这些值，而且可能还要修改，所以该Component必须提供setter方法。

但是，Component（组件）并不为每个属性提供细粒度的set方法（这会造成更高的网络负载），而是暴露一个粗粒度的setData()方法，该方法接收一个TransferObject（传输对象）作为参数。传给该方法的TransferObject盛放了客户端发来的、更新后的数值。因为该TransferObject必须是可改动的，所以它必须为客户端能够修改的每个属性提供setter方法。TransferObject的setter方法中可以包括字段级别的数据验证以及数据完整性检查。客户端从Component获得TransferObject之后，就可以（在必要时）在本地调用该TransferObject的setter方法，修改属性的值。但是，这些本地的改动并不会影响到Component本身，直到客户端调用了它的setData()方法。

这个setData()把客户端的TransferObject（传输对象）拷贝串行化，并发给Component（组件）。Component从客户端获取了修改后的TransferObject，然后把改动合并到自己的属性中。

这个“合并”操作可能会使Component（组件）和TransferObject（传输对象）的设计变得很复杂。一种策略是，只更新有改动的属性，而不是更新所有属性。可以在TransferObject使用一个改动标志位（change flag），这样客户端要判断哪些属性有改动，就不用直接对比各个属性值了。

可更新的传输对象策略会使对更新的传播、同步、版本控制变得更加复杂。

采用可更新的传输对象策略，则会允许客户端修改本地的TransferObject（传输对象）拷贝。但是在客户端获取了TransferObject的这个本地拷贝之后，Component（组件）上还可能发生更新，而这些更新可能没有传播到本地拷贝中。之所以发生这种情况，是因为Component本身没法“感知”到客户端的TransferObject拷贝。这样一来，本地拷贝中的数据就可能是不正确的。

另外，Component（组件）还要应付一种情况：两个以上的客户端同时要求更新远程数据。

允许这种同步的更新可能会导致数据的冲突和不一致。避免这种冲突的一个办法是版本控制。Component可以包含一个版本号或是一个“上次修改”时戳（time stamp）。Component把这个版本号/时戳复制到TransferObject（传输对象）中。这样更新事务通过使用版本号/时戳属性就能够避免冲突。如果一个客户端的TransferObject已经失效，但仍然试图更新Component，那么Component就能够检测到TransferObject上失效的版本号/时戳，并且把错误情况报告给客户端。然后，客户端必须获取最新版本的TransferObject，并重新尝试更新。在极端的情况下，这可能导致“客户端饥饿（client starvation）”——客户端可能永远也没法完成更新。

419

图7-39是整个更新交互过程的序列图。

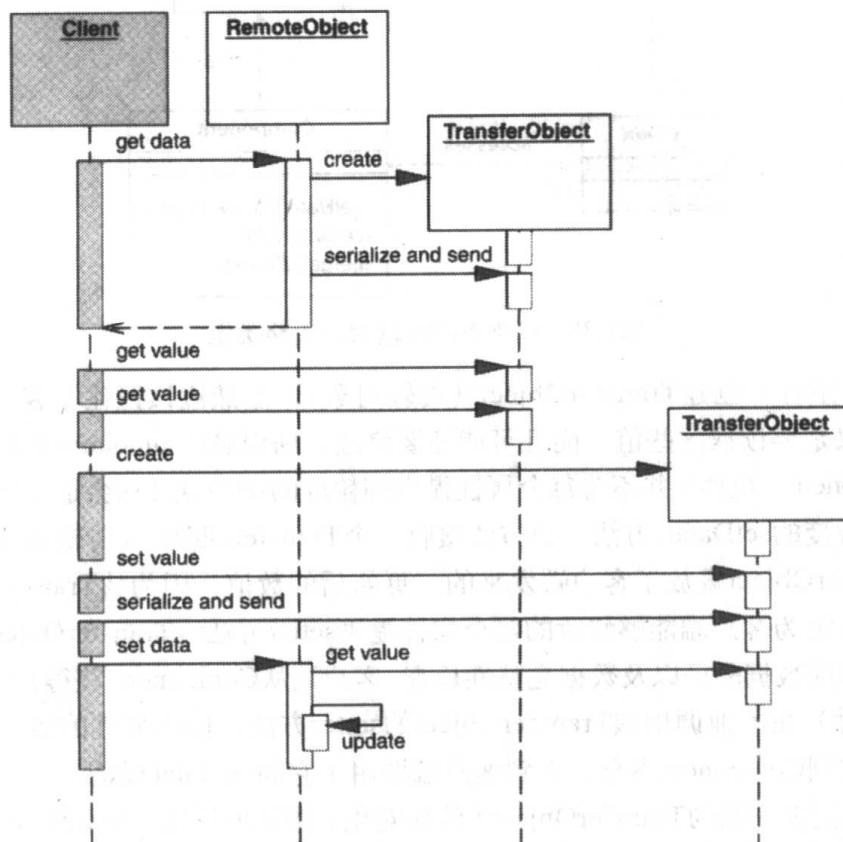


图7-39 可更新的传输对象策略序列图

### 多重传输对象策略

有些Component（组件）可能很复杂，封装了多种数据元素。当这些Component生成TransferObject（传输对象）的时候，就没有必要只生成一种TransferObject，然后把Component包含的所有数据元素都放在这个TransferObject。相反，应该让这个Component生成多种TransferObject，每一个TransferObject装载的数据正好足够满足一个请求或用例的需求。

在下列环境下可以采用这个策略：

- Component（组件）是用session bean实现的，通常是一个会话界面，而这个session bean为了满足服务请求，可能要与多个其他的业务组件交互。所以这个session bean会从不同的数据源生成TransferObject（传输对象）。

420

• 与此类似，如果Component（组件）是用复合实体实现的，那么该entity bean也可能封装了由从属对象构成的一套拓扑结构。

在这两种情况下，生成多个传输对象、体现被封装的数据的不同部分，都是一种很好的实践。

比如，在一个交易应用系统中，可能用一个复合实体实现一个“客户投资”业务对象，这可能就是一个粗粒度的复杂组件，包括多个从属对象。所以，这个复合实体就可以生成多个不同的传输对象，体现投资信息的不同部分内容，比如CustomerInformation（客户信息）、StockHoldingsList（所持股票列表）等等。

再用同一个应用系统举一个例子：一个CustomerManager（客户管理器）服务对象，实现了会话门面模式。这个门面为了提供服务，会与多个其他组件和业务组件交互。因此，这个CustomerManager也可以生成多种不同的小型传输对象，比如CustomerAddress（客户地址）、ContactList（联系人列表）等等，其中每个传输对象都体现了整个模型的一个部分。

在这两种应用场景中都可以在实现Component（组件）的时候加入多个getter方法，这样客户端就可以通过这些方法获取不同的传输对象。

多重传输对象策略的类图如图7-40所示。

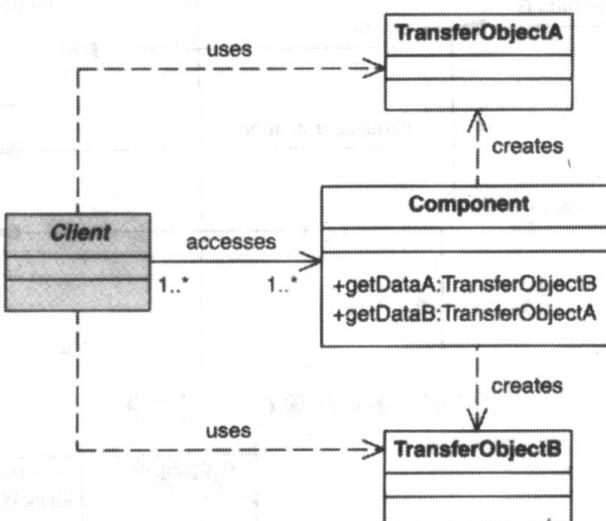


图7-40 多重传输对象策略的类图

如果客户端需要类型为TransferObjectA的传输对象，它就调用Component的getDataA()方法，请求TransferObjectA。如果它需要类型为TransferObjectB的传输对象，它就调用Component的getDataB()方法，请求TransferObjectB，以此类推。

421

图7-41的序列图体现了这种交互。

### 实体继承传输对象策略

如果Component（组件）是由entity bean实现的，那么组件中的数据通常直接对应于客户端需要的数据。在这种情况下，既然entity bean和传输对象之间存在一种一对一的映射关系，那么entity bean就可以通过继承来避免代码重复。

在本策略中，entity bean继承了TransferObject（传输对象）类，这样二者就有同样的属性和getter/setter方法。

本策略的类图如图7-42所示。

TransferObject（传输对象）实现了一个以上的getData()方法，如多重传输对象策略中所介绍。Entity bean继承了这个TransferObject对象，所以客户调用entity bean的时候，也就调用了继承的getdata()方法，获得TransferObject的一个实例。  
422

图7-43是本策略的序列图。

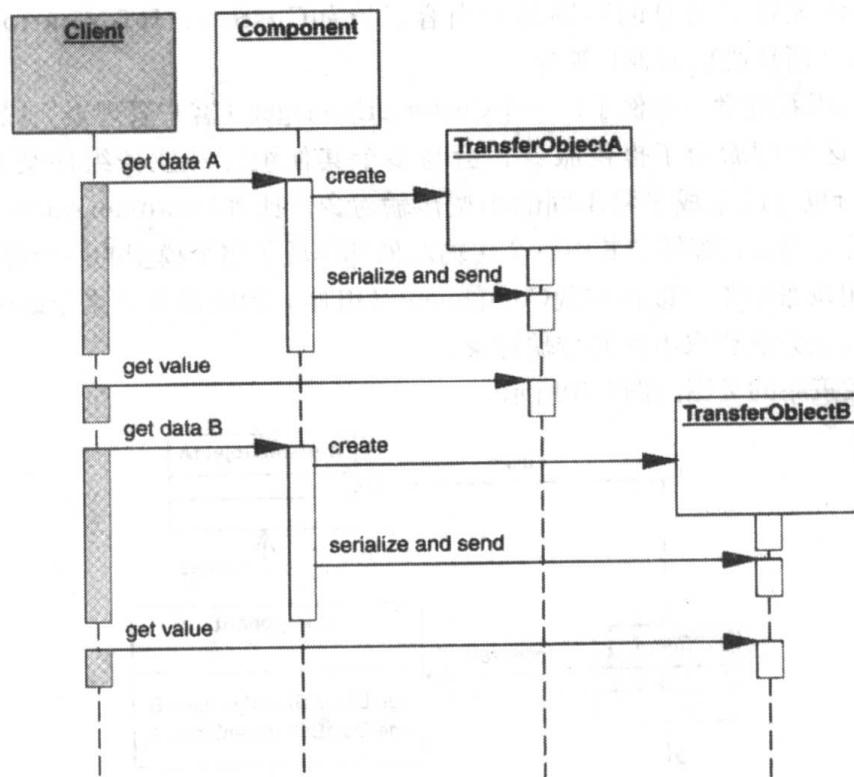


图7-41 多重传输对象的序列图

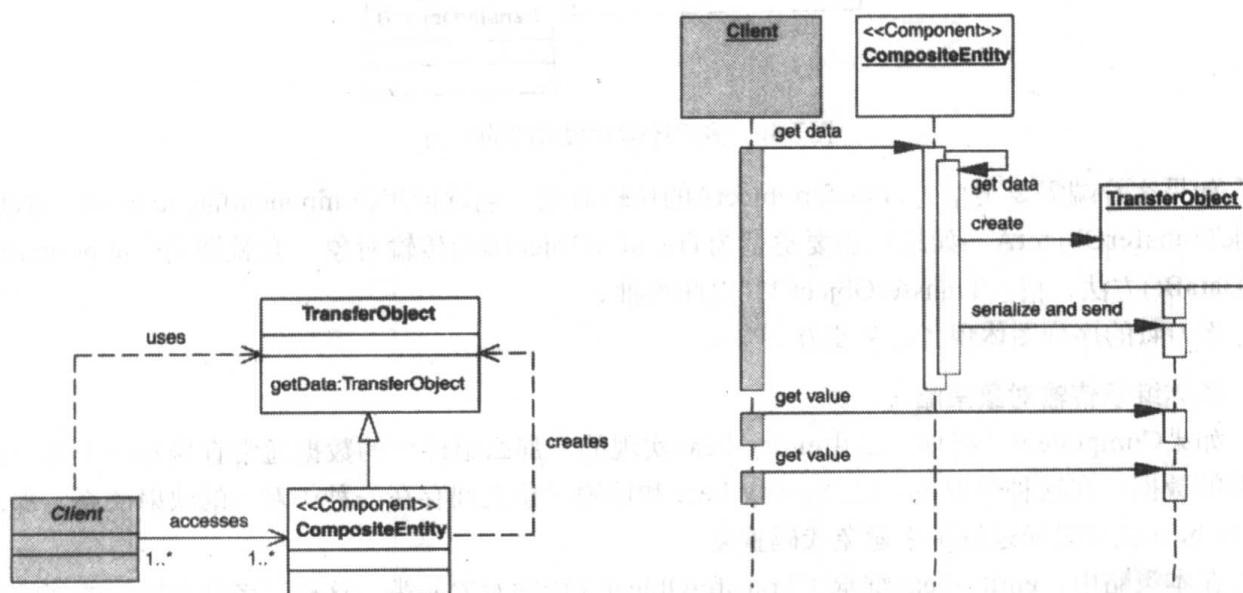


图7-42 实体继承传输对象策略类图

图7-43 实体继承传输对象策略序列图

这样一来，本策略就能够消除entity bean和TransferObject（传输对象）之间的代码重复。该策略也有助于管理TransferObject的设计需求，因为对需求的各种改动都被隔离在TransferObject类中了，这样就避免了需求的改动对entity bean造成影响。

但是由于采用了继承，本策略也会有一定代价。修改TransferObject（传输对象）类，也就会影响它的所有子类，可能会导致对象结构中的其他文件也要发生改动。

后文示例代码中的例7.36“实体继承传输对象策略——传输对象类”、例7.37“实体继承传输对象策略——Entity Bean类”是本策略的一个示例实现。

## 效果

- **降低了网络负载**

只需要一次远程调用，传输对象就能够把一组数值从远程对象传给客户端，这样也就减少了远程调用的次数。整个应用系统的“罗嗦”程度降低了，网络性能提高了。

- **简化了远程对象和远程接口**

远程对象提供了粗粒度的getData() 和 setData()方法，通过传输对象获取和设置数据值。这就消除了远程对象的细粒度get/set方法。

- **用较少的远程调用传输了较多的数据**

在本模式中，客户端不用多次跨过网络调用远程对象以获取属性值，而是执行单独一次方法调用，返回已经聚合的数据。如果考虑采用本模式必须权衡以下考虑：一方面网络调用少了，另一方面，每次调用传送的数据则多了。

- **减少了代码重复**

可以使用实体继承传输对象策略，消除entity bean和相关传输对象之间的代码重复。

- **引入失效的传输对象**

使用传输对象可能会在系统的各部分引入失效的数据。但是这是一种常见的副作用，只要数据和远程数据源之间中断了连接，它就会产生，因为远程对象通常不会跟踪所有获取了数据的客户端，所以也很难通过这种方法在整个系统中传播对数据的更新。

- **由于同步和版本控制，增加了系统的复杂度**

如果使用可更新的传输对象策略，就必须对并行访问进行设计。也就是说，由于加入了同步和版本控制机制，设计可能会更加复杂。

## 示例代码

### 实现传输对象模式

设想这样一个例子：一个业务对象，称为“Project（项目）”，用entity bean形式建模并实现。当客户端调用Project entity bean的getProjectData()方法时，这个entity bean就要通过传输对象把数据传送给客户端。本例中的传输对象，ProjectTO，如例7.29所示。

423

424

**例7.29 实现传输对象模式——传输对象(Transfer Object)类**

```

1
2 // 用来存放项目细节信息的传输对象
3 public class ProjectTO implements java.io.Serializable {
4     public String projectId;
5     public String projectName;
6     public String managerId;
7     public String customerId;
8     public Date startDate;
9     public Date endDate;
10    public boolean started;
11    public boolean completed;
12    public boolean accepted;
13    public Date acceptedDate;
14    public String projectDescription;
15    public String projectStatus;
16
17    // 传输对象的构造函数...
18 }

```

使用该传输对象的entity bean代码如例7.30所示。

**例7.30 实现传输对象模式——Entity Bean类**

```

1
2 ...
3 public class ProjectEntity implements EntityBean {
4     private EntityContext context;
5     public String projectId;
6     public String projectName;
7     public String managerId;
8     public String customerId;
9     public Date startDate;
10    public Date endDate;
11    public boolean started;
12    public boolean completed;
13    public boolean accepted;
14    public Date acceptedDate;
15    public String projectDescription;
16    public String projectStatus;
17    private boolean closed;
18
19    // 其他属性...
20
21    private ArrayList commitments;
22    ...
23
24    // 本方法用Project数据给传输对象赋值
25    public ProjectTO getProjectData() {

```

```

26     ProjectTO proj = new ProjectTO();
27     proj.projectId = projectId;
28     proj.projectName = projectName;
29     proj.managerId = managerId;
30     proj.startDate = startDate;
31     proj.endDate = endDate;
32     proj.customerId = customerId;
33     proj.projectDescription = projectDescription;
34     proj.projectStatus = projectStatus;
35     proj.started = started;
36     proj.completed = completed;
37     proj.accepted = accepted;
38     proj.closed = closed;
39     return proj;
40 }
41 ...
42 }
```

## 实现可更新的传输对象策略

你可以通过扩展例7.30来实现该策略。这时，entity bean会提供一个setProjectData()方法，传入一个传输对象，并用该对象中盛放的数据更新entity bean。该策略的示例代码如例7.31所示。

426

### 例7.31 实现可更新的传输对象策略

```

1 ...
2 ...
3 public class ProjectEntity implements EntityBean {
4     private EntityContext context;
5     ...
6
7     // 属性和其他方法如
8     // 上面的例7.30中所示
9     ...
10
11    // 用一个传输对象更新entity bean的值
12    public void setProjectData(ProjectTO updatedProj) {
13        mergeProjectData(updatedProj);
14    }
15
16    // 本方法把传输对象的值
17    // 合并到entity bean的属性中
18    private void mergeProjectData(ProjectTO updatedProj) {
19        // 在合并之前，可能在这里加入
20        // 加入版本控制，以免丢失
21        // 其他客户端执行的更新
22        projectId = updatedProj.projectId;
23        projectName = updatedProj.projectName;
```

```

24     managerId = updatedProj.managerId;
25     startDate = updatedProj.startDate;
26     endDate = updatedProj.endDate;
27     customerId = updatedProj.customerId;
28     projectDescription = updatedProj.projectDescription;
29     projectStatus = updatedProj.projectStatus;
30     started = updatedProj.started;
31     completed = updatedProj.completed;
32     accepted = updatedProj.accepted;
33     closed = updatedProj.closed;
34 }
35 ...
36 }
```

427

## 实现多重传输对象策略

也许从数据源需要的信息，并不是每次都是一样的类型、一样的数量。为了满足这种对不同信息类型的需求，可以使用多重传输对象，其中每一个传输对象用来存放一种类型的信息，这样就可以在请求时根据需要的信息选择合适的传输对象了。

考虑这样一个专业服务应用系统<sup>①</sup>：其中客户端需要获取繁简程度不同的“资源（resource）”信息。可以提供两种传输对象：ResourceTO（资源传输对象）和ResourceDetailsTO（资源详细信息传输对象），客户端通过访问这两种对象获取数据。ResourceTO传输的一组数据中，包括的属性比较少，而ResourceDetailsTO传输的数据中包括的属性则比较多。

这个例子采用的是一种子集/父集的办法，一个传输对象带有的数据比较多，另一个包含的数据则只是全部数据的子集。另外，还可以创建传输对象，用来盛放不同的、相互没有重叠的数据，比如，可以有一个ContactInfo（联系人信息）传输对象盛放一个经理的邮件地址和电话号码，再有一个HRInfo 传输对象，存放该经理的姓名和薪水。

例7.32和例7.33的示例代码，给出了两个这样的传输对象。例7.34的示例代码是一个entity bean，它能够产生以上传输对象；而例7.35则给出了entity bean的客户端代码。

### 例7.32 多重传输对象策略——ResourceTO（资源传输对象）

```

1
2 // ResourceTO: 这个类中存放关于资源的
3 // 基本信息
4 public class ResourceTO implements java.io.Serializable {
5     public String resourceId;
6     public String lastName;
7     public String firstName;
8     public String department;
9     public String grade;
10    ...
11 }
```

<sup>①</sup> 专业服务应用系统是全书的一个常见例子。首次出现是在本章的“业务代表”一节的示例代码中。

### 例7.33 多重传输对象策略——ResourceDetailsTO（资源详细信息传输对象）

```

1 // ResourceDetailsTO: 这个类存放关于资源的
2 // 详细信息
3 public class ResourceDetailsTO
4 implements java.io.Serializable {
5     public String resourceId;
6     public String lastName;
7     public String firstName;
8     public String department;
9     public String grade;
10    public String ...
11    // 其他数据...
12    public Collection commitments;
13    public Collection blockoutTimes;
14    public Collection skillSets;
15 }

```

428

### 例7.34 多重传输对象策略——资源Entity Bean

```

1
2 // imports
3
4 public class ResourceEntity implements EntityBean {
5     // entity bean 的属性
6     ...
7
8     // entity bean 的业务方法
9     ...
10
11    // 多重传输对象方法: 获取ResourceTO (资源传输对象)
12    public ResourceTO getResourceData() {
13
14        // 创建新的ResourceTO实例, 并从entity bean中
15        // 把属性值复制给这个传输对象
16        ...
17    }
18
19    // 多重传输对象方法: 获取
20    // ResourceDetailsTO (资源详细信息传输对象)
21    public ResourceDetailsTO getResourceDetailsData() {
22
23        // 创建新的ResourceDetailsTO实例, 并从entity bean中
24        // 把属性值复制给这个传输对象
25        ...
26    }
27
28    // entity bean的其他方法

```

29 ...  
 30 }

429

### 例7.35 多重传输对象策略——Entity Bean的客户端

```

31 ...
32 private ResourceEntity resourceEntity;
33 private static final Class homeClazz =
34     corepatterns.apps.psa.ejb.ResourceEntityHome.class;
35 ...
36
37 try {
38     ResourceEntityHome home = (ResourceEntityHome)
39         ServiceLocator.getInstance().getLocalHome(
40             "Resource", homeClazz);
41     resourceEntity = home.findByPrimaryKey(resourceId);
42 } catch (ServiceLocatorException ex) {
43     // 把服务定位器的异常翻译成
44     // 应用异常
45     throw new ResourceException(...);
46 } catch (FinderException ex) {
47     // 把entity bean finder异常翻译成
48     // 应用异常
49     throw new ResourceException(...);
50 } catch (RemoteException ex) {
51     // 把Remote (远程) 异常翻译成
52     // 应用异常
53     throw new ResourceException(...);
54 }
55 ...
56
57 // 获取资源基本数据
58 ResourceTO to = resourceEntity.getResourceData();
59 ...
60
61 // 获取资源详细数据
62 ResourceDetailsTO dto =
63     resourceEntity.getResourceDetailsData();
64 ...
65

```

## 实现实体继承传输对象策略

考虑这样一个例子：一个叫ContactEntity（联系人实体）的 entity bean，从一个名叫 ContactTO（联系人传输对象）的传输对象中继承了所有属性。例7.36中的ContactTO传输对象就是这种策略的示例代码。

430

**例7.36 实体继承传输对象策略——传输对象类**

```

1   // 这个传输对象类将被
2   // entity bean继承
3   public class ContactTO implements java.io.Serializable {
4
5       // 公共成员
6       public String firstName;
7       public String lastName;
8       public String address;
9
10      // 默认构造函数
11      public ContactTO() {}
12
13
14      // 一个接受所有值的构造函数
15      public ContactTO(String firstName, String lastName,
16                      String address) {
17          init(firstName, lastName, address);
18      }
19
20      // 本构造函数按照一个现存的
21      // 传输对象实例创建一个新的传输对象
22      public ContactTO(ContactTO contact) {
23          init(contact.firstName, contact.lastName,
24               contact.address);
25      }
26
27      // 本方法设置所有属性的值
28      public void init(String firstName, String lastName,
29                      String address) {
30          this.firstName = firstName;
31          this.lastName = lastName;
32          this.address = address;
33      }
34
35      // 创建一个新的传输对象
36      public ContactTO getData() {
37          return new ContactTO(this);
38      }
39  }

```

与本策略相关的entity bean示例代码如例7.37所示。

431

**例7.37 实体继承传输对象策略——Entity Bean类**

```

1
2   public class ContactEntity extends ContactTO
3   implements javax.ejb.EntityBean {
4       ...

```

```

5   // 客户端调用ContactEntity bean实例的
6   // getData 方法。
7   // getData()方法是从传输对象继承而来,
8   // 返回ContactTO传输对象。
9   ...
10 }

```

## 相关模式

- **会话门面**

会话门面模式经常要与参与该模式的业务对象交换数据，这一交换机制往往是由传输对象实现的。门面充当了底层服务的代理，从业务对象获取传输对象，并传递给客户端。

- **传输对象组装器**

传输对象组装器把来自多种不同的数据源的数据构建成复合传输对象。数据源通常是session bean或者entity bean，客户端请求这些数据源中的数据，而数据源则把数据以传输对象的形式提供给传输对象组装器。这些传输对象构成了传输对象组装器要组装的复合对象的各个部分。

- **值列表处理器**

值列表处理器也是一种提供动态构造的传输对象列表的模式，在请求处理过程中，它负责访问持久化存储，返回传输对象的列表。

- **复合实体**

传输对象模式满足了在多个层次之间传输对象的需求。在设计entity bean的时候，这是一个重要的考虑：可以使用传输对象把数据传输给entity bean并从entity bean中获取数据。

432

## 传输对象组装器

### 问题

需要获取一个应用模型，但这个模型中包括了来自多个不同业务组件的传输对象。

应用系统的客户端经常需要从业务层获取业务数据（也叫应用模型），然后要么把这些数据显示出来，要么执行一些中间处理。应用模型代表着封装在业务层业务组件中的业务数据。如果客户端需要应用模型数据的话，就必须要从多种不同的数据源（比如业务对象、数据访问对象、应用服务以及业务层的其他对象）中定位、访问、获得模型的各个部分。这种做法会产生几个问题：

- 如果让客户端直接访问另一个层次中的组件，就会在不同层次之间造成耦合。由于这种紧耦合的存在，如果修改了业务层组件，也就会在各个客户端中间产生连锁反应。客户端代码的复杂性增加了，因为客户端必须要和多种业务组件交互，并且那些用于构造应用模型的业务逻辑也被放进了客户端代码中。
- 另外，不同的客户端为了获取、构造模型数据，都要实现一些相似的逻辑，这也就加大了系统维护的工作量。如果客户端访问的业务组件是分布式的远程组件，那么性能也会随之

下降，因为客户端必须越过网络访问多个业务组件才能获得全部业务模型数据。

## 约束

- 要集中封装业务逻辑，避免在客户端实现这些逻辑。
- 在构造业务层对象模型的数据时，尽量减少对远程对象的网络调用。
- 创建一个复杂模型，并交给客户端，用于界面显示。
- 要对客户端隐藏模型实现的复杂性，而且还想降低客户端与业务组件之间的耦合。

433

## 解决方案

使用传输对象组装器，以复合传输对象的形式构建应用模型。传输对象组装器从各种不同的业务组件和业务服务中聚合多个传输对象，并且最后把复合传输对象返回给客户端。

传输对象组装器从业务组件中获取传输对象。然后，传输对象组装器对这些传输对象进行处理，创建并组装一个体现了应用模型数据的复合传输对象。客户端使用传输对象组装器来获取只读的应用模型，以供显示或其他中间处理之用。客户端并不修改复合传输对象中的数据。

前面讲过，业务对象也能够使用封装在它本身中的数据生成复合传输对象。另一方面，传输对象组装器则从多种数据源（比如业务对象、会话门面、应用服务、数据访问对象，以及其他各种服务）获取数据。传输对象组装器可以使用服务定位器来定位业务服务组件（比如会话门面等）。

## 结构

图7-44表现了传输对象组装器模式中的关系。

434

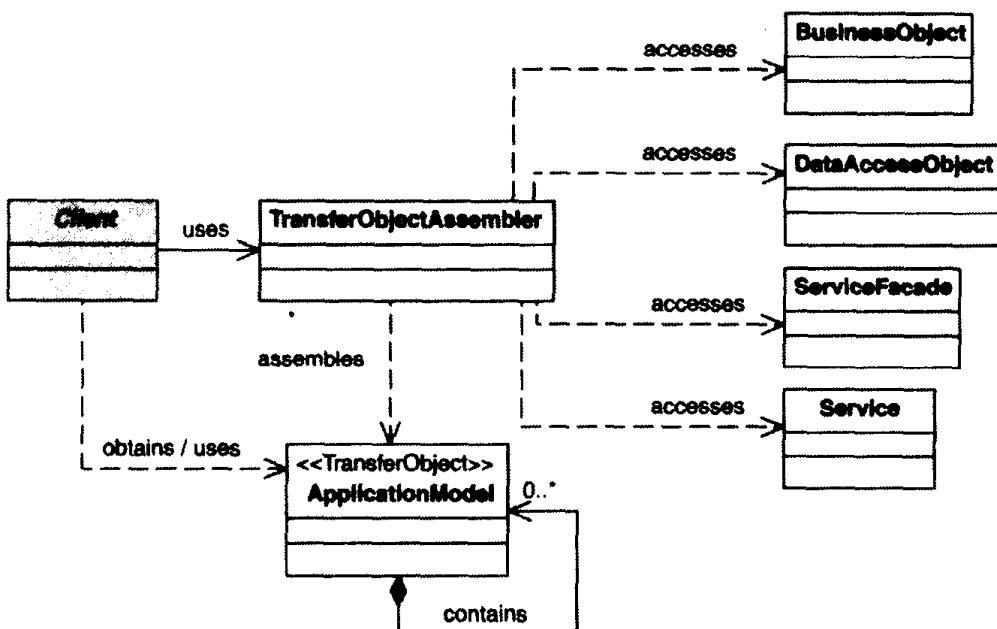


图7-44 传输对象组装器类图

## 参与者和责任

图7-45中的序列图体现了传输对象组装器模式中各个参与者之间的交互。

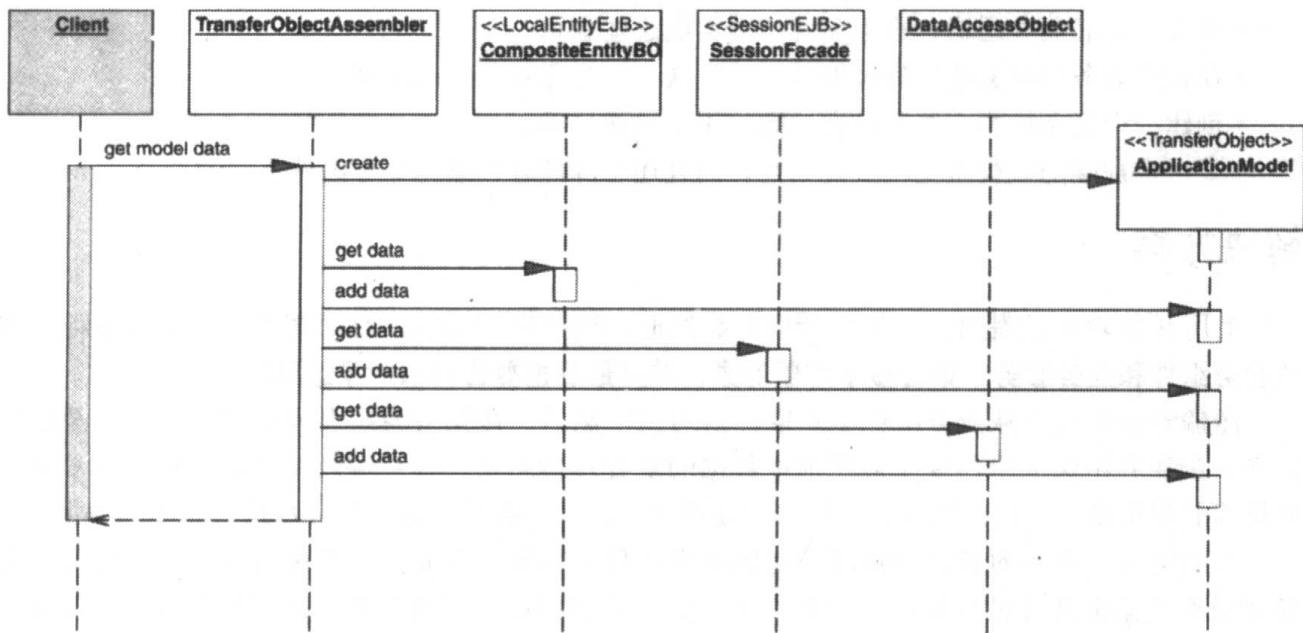


图7-45 传输对象组装器序列图

### Client (客户端)

Client (客户端) 调用TransferObjectAssembler (传输对象组装器), 获取应用模型数据。Client可以是表现层的一个组件, 也可以是会话门面, 其他远程客户端可以通过这个门面访问TransferObjectAssembler。如果TransferObjectAssembler是用session bean实现的, 那么Client可以是会话门面或者业务代表。

### TransferObjectAssembler (传输对象组装器)

TransferObjectAssembler (传输对象组装器) 是本模式中的主要类。当客户端请求应用模型数据的时候, TransferObjectAssembler 根据应用需求构造一个新的复合传输对象。

### ApplicationModel (应用模型)

ApplicationModel (应用模型) 对象是一个复合传输对象, 它是由TransferObjectAssembler (传输对象组装器) 构造的, 并由后者返回给Client (客户端)。

### BusinessObject (业务对象)

BusinessObject (业务对象) 是业务对象模式的一个实现, 它为TransferObjectAssembler (传输对象组装器) 提供传输对象, 以便让后者组装ApplicationModel (应用模型)。

### SessionFacade (会话门面)

SessionFacade (会话门面) 是会话门面模式的一个实现, 它能提供用于构造ApplicationModel (业务模型) 传输对象的部分数据。

### DataAccessObject (数据访问对象)

`DataAccessObject` (数据访问对象) 是数据访问对象模式的一个实现，当 `TransferObjectAssembler` (传输对象组装器) 需要直接从持久化存储中获取数据的时候，就要用上它。

### Service (服务)

这里 `Service` (服务) 可以是任意的服务对象，只要其中包含了一个业务层的应用服务，在构造 `ApplicationModel` (应用模型) 对象的过程中，需要该应用服务提供数据。

436

## 策略

### POJO传输对象组装器策略

传输对象组装器可以用一个普通Java对象实现，不一定是EJB。在这种实现中，通常用 `session bean` 充当传输对象组装器的前端。这个 `session bean` 一般就是会话门面，除了给传输对象组装器充当前端之外，它还负责提供其他业务服务。无论采用哪种实现策略，传输对象组装器都是在业务层运行的。这样做的原因，就是在传输对象组装器访问各种作为数据源的业务组件的时候，不至于需要跨层次的远程调用。

### Session Bean传输对象组装器策略

本策略用 `session bean` 实现传输对象组装器 (如上面的类图所示)。如果使用 `session bean` 实现传输对象组装器，从而形成一个业务服务，那么通常会采用无状态 `session bean`。要注意的一点是，构成应用模型的那些业务组件会一直涉及多个客户端的事务过程。所以，当传输对象组装器根据多个业务组件构造一个复合传输对象时，它只不过是生成了那个特定的构造时刻的一个业务层对象模型快照。因此，如果其后另一个客户端修改了一个以上的业务对象，从而改变了业务应用模型，那么业务层对象模型也就改变了。

所以，用有状态 `session bean` 实现传输对象组装器，并不比无状态 `session bean` 更好。因为既然底层业务对象模型都改变了，也就没有必要再保持应用模型的状态了。也就是说，如果底层对象模型改变了，那么组装器中存放的传输对象也就失效了。

当客户端下一次向传输对象组装器索取传输对象的时候，传输对象组装器要么返回一个失效的状态，要么根据最新的快照重建传输对象。所以，传输对象组装器通常用无状态 `session bean` 实现。

但是，如果底层的业务对象模型很少改变，那么就可以用有状态 `session bean` 实现传输对象组装器，一直保留构建好的传输对象。但在这种情况下，传输对象组装器就必须包括一种机制，能够发觉底层业务模型的变化，在下一次客户端请求时，根据变化后的数据重建应用模型。

437

## 效果

- 分离了业务逻辑，简化了客户端逻辑

如果客户端包括一些与分布式组件交互的处理逻辑，那么就很难清晰地把业务逻辑同客户端层截然分开。传输对象组装器中包含了用于维护对象关系的业务逻辑，以及用于构造符合模

型的复合传输对象的业务逻辑。这样，客户端就不需要知道如何构造模型，也不需要了解为这个复合模型提供数据的多个组件。

- **减少了客户端和应用模型之间的耦合**

传输对象组装器对客户端隐藏了构造模型数据的复杂性，降低了客户端和模型之间的耦合。在这种松耦合的情况下，如果模型发生了变化，只需要传输对象组装器相应地改变就可以了，这样一来，就能对客户端隐藏模型的变化。

- **提高了系统的网络性能**

传输对象组装器减少了从业务层获取应用模型时所需的远程调用次数，因为通常只需一次方法调用就能获取应用模型。但是，这个复合传输对象可能包含大量数据。这也就意味着，虽然使用传输对象组装器能够减少网络调用的次数，但是一次调用中传输的数据量增加了。在使用这个模式的时候，应该权衡考虑这个代价。

- **提高了客户端性能**

服务器端的传输对象组装器无需使用任何客户端资源，就能以复合传输对象的形式构造模型数据。在组装模型的过程中，客户端没有消耗任何资源。

- **可能会引入失效的数据**

传输对象组装器按照需要，以复合传输对象的形式构造应用模型，而数据来源则是业务模型当前状态的一个快照。客户端获得了复合传输对象之后，这个对象就在客户端本地了，与原本的数据之间不存在网络联系。所以，如果在业务组件上发生了后续改动，这种改动也不会传播到客户端的应用模型数据上。因此，这些应用模型数据在客户端获取之后可能会失去时效性。

438

## 示例代码

### 实现传输对象组装器

考虑一个项目管理应用系统，其中用多个业务层组件定义了一个复杂的模型。假设由一个客户端想获取一种由不同的业务对象中的数据构成的模型数据，包括：

- Project（项目）组件中的项目信息
- ProjectManager（项目管理者）组件中的项目管理者信息
- Project组件中的项目任务列表
- Resource（资源）组件中的资源信息

例7.38给出了一个复合传输对象，其中就包括以上信息。可以使用传输对象组装器模式来实现对这个复合传输对象的组装。传输对象组装器的示例代码如例7.42所示。

#### 例7.38 复合传输对象类

```

1
2  public class ProjectDetailsData
3      implements java.io.Serializable {
4          public ProjectTO projectData;
5          public ProjectManagerTO projectManagerData;

```

```

6     public Collection listOfTasks;
7     ...
8 }
```

ProjectDetailsData（项目详细数据）中的任务列表是多个TaskResourceTO（任务资源传输对象）构成的一个集合。TaskResourceTO（任务资源传输对象）是TaskTO（任务传输对象）和ResourceTO（资源传输对象）的一个结合物。例7.39、例7.40和例7.41给出了这些类。

### 例7.39 TaskResourceTO（任务资源传输对象）类

```

1
2  public class TaskResourceTO implements java.io.Serializable {
3      public String projectId;
4      public String taskId;
5      public String name;
6      public String description;
7      public Date startDate;
8      public Date endDate;
9      public ResourceTO assignedResource;
10     ...
11
12     public TaskResourceTO(String projectId, String taskId,
13             String name, String description, Date startDate,
14             Date endDate, ResourceTO assignedResource) {
15         this.projectId = projectId;
16         this.taskId = taskId;
17         ...
18         this.assignedResource = assignedResource;
19     }
20     ...
21 }
```

439

### 例7.40 TaskTO（任务传输对象）类

```

1
2  public class TaskTO implements java.io.Serializable {
3      public String projectId;
4      public String taskId;
5      public String name;
6      public String description;
7      public Date startDate;
8      public Date endDate;
9      public String assignedResourceId;
10
11     public TaskTO(String projectId, String taskId, String name,
12             String description, Date startDate, Date endDate,
13             String assignedResourceId) {
14         this.projectId = projectId;
15         this.taskId = taskId;
16         ...
```

```

17     this.assignedResource = assignedResource;
18 }
19 ...
20 }

```

#### 例7.41 ResourceTO (资源传输对象) 类

```

1
2 public class ResourceTO implements java.io.Serializable {
3     public String resourceId;
4     public String resourceName;
5     public String resourceEmail;
6     ...
7
8     public ResourceTO (String resourceId, String resourceName,
9         String resourceEmail, ...) {
10        this.resourceId = resourceId;
11        this.resourceName = resourceName;
12        this.resourceEmail = resourceEmail;
13        ...
14    }
15 }

```

ProjectDetailsAssembler (项目详细数据组装器) 类负责组装ProjectDetailsData (项目详细数据) 对象, 其代码如例7.42所示。

#### 例7.42 实现传输对象组装器

```

1
2 public class ProjectDetailsAssembler
3     implements javax.ejb.SessionBean {
4
5     ...
6
7     public ProjectDetailsData getData(String projectId) {
8
9         // 构造复合传输对象
10        ProjectDetailsData pData = new
11            ProjectDetailsData();
12
13        // 获得项目详细信息
14        ProjectHome projectHome =
15            ServiceLocator.getInstance().getLocalHome(
16                "Project", ProjectEntityHome.class);
17        ProjectEntity project =
18            projectHome.findByPrimaryKey(projectId);
19        ProjectTO projTO = project.getData();
20
21        // 把项目信息加入到ProjectDetailsData (项目详细数据) 中
22        pData.projectData = projTO;

```

```
23
24     // 获得项目管理者详细信息;
25     ProjectManagerHome projectManagerHome =
26         ServiceLocator.getInstance().getLocalHome(
27             "ProjectManager", ProjectEntityHome.class);
28
29     ProjectManagerEntity projectManager =
30         projectManagerHome.findByPrimaryKey(
31             projTO.managerId);
32
33     ProjectManagerTO projMgrTO = projectManager.getData();
34
35     // 把项目管理者信息加入到ProjectDetailsData(项目详细数据)中
36     pData.projectManagerData = projMgrTO;
37
38     // 获得该项目的TaskTO(任务传输对象)列表
39     Collection projTaskList = project.getTasksList();
40
41     // 构造TaskResourceTO(任务资源传输对象)列表
42     ArrayList listOfTasks = new ArrayList();
43
44     Iterator taskIter = projTaskList.iterator();
45     while (taskIter.hasNext()) {
46         TaskTO task = (TaskTO) taskIter.next();
47
48         // 获得资源详细信息;
49         ResourceHome resourceHome =
50             ServiceLocator.getInstance().getLocalHome(
51                 "Resource", ResourceEntityHome.class);
52
53         ResourceEntity resource =
54             resourceHome.findByPrimaryKey(
55                 task.assignedResourceId);
56
57         ResourceTO resTO = resource.getResourceData();
58
59         // 使用任务、资源数据, 构造一个新的using Task
60         // TaskResourceTO(任务资源传输对象)
61         TaskResourceTO trTO =
62             new TaskResourceTO( task.projectId, task.taskId,
63                 task.name, task.description, task.startDate,
64                 task.endDate, resTO);
65
66         // 把TaskResourceTO(任务资源传输对象)加入到列表中
67         listOfTasks.add(trTO);
68     }
69 }
```

```

442    70      // 把任务列表加入到ProjectDetailsData（项目详细数据）中
        71      pData.listOfTasks = listOfTasks;
        72
        73      // 把其他数据加入到传输对象中
        74      ...
        75
        76      // 返回复合传输对象
        77      return pData;
        78
        79  }
        80  ...
        81 }

```

## 相关模式

- **传输对象**

传输对象组装器使用传输对象模式创建数据并把数据传送到客户端。它创建出的传输对象装载着体现了应用模型的数据，并把这些数据从业务层传送给请求数据的客户端。

- **业务对象**

传输对象组装器使用相关的业务对象获取、构建相关的应用模型。

- **复合实体**

复合实体根据自身的数据生成复合传输对象。另一方面，传输对象组装器则要从多种数据源（比如会话界面、业务对象、应用服务、数据访问对象以及其他服务）中获取数据，构造应用模型。

- **会话界面**

如果用session bean实现传输对象组装器，就可以把它当成会话界面模式的一种有限的、特殊的用法。如果客户端需要更新一些提供了应用模型数据的业务组件，它就要访问这种会话界面（session bean），由后者提供更新服务。

- **数据访问对象**

传输对象组装器可以直接通过数据访问对象从数据存储中获取数据。

- **服务定位器**

传输对象组装器使用服务定位器，定位、使用各种业务组件。

443

## 值列表处理器

### 问题

远程客户端要遍历一个很大的结果列表。

很多J2EE应用系统都会让客户端执行各种查询。这些查询经常从表现层开始，由业务层执行，最后显示在浏览器里。

可以用多种方法完成查询。如果是用entity bean实现的业务对象，可以用entity bean的finder

方法。如果没有用entity bean，那么通常就会使用数据访问对象执行查询。当查询只返回一个很小的结果集的时候，不会出现什么问题。但是，如果查询返回了大量匹配的entity bean，那么entity bean的finder方法的效率就很低了。

另一个问题在于，也许客户端没有处理大型结果集的能力，所以就要由服务器来处理结果。客户端往往都不会使用查询的全部结果，所以在查看、使用了一部分结果之后，剩下的也就抛下不用了。比如，用户可能用浏览器执行一次查询，查看了开头的几条结果，抛下了剩下的结果，又执行另一次查询了。

所以，通常并不需要把整个的查询结果都返回给客户端。如果客户端只显示了前几条结果，然后就抛下了这次查询，那么网络带宽也没有浪费，因为数据可以在服务器端缓存，可能永远也不会送到客户端。

## 约束

- 需要避免使用EJB finder方法处理大型查询造成的负载。
- 需要实现一种只读用例，这种用例不需要事务。
- 需要为客户端提供一种机制，能够高效地进行查询、并且遍历一个大型结果集。
- 需要在服务器端维护查询结果。

444

## 解决方案

**使用值列表处理器来执行查询、缓存结果，并且让客户端遍历、选择查询结果。**

值列表处理器提供了查询和迭代的功能。为了完成一次查询，值列表处理器使用数据访问对象执行查询，从数据库获取匹配的结果。即使应用系统使用了entity bean实现业务对象，本模式也避免使用EJB finder方法。

### 设计手记：采用Entity Bean的finder方法和EJB Select方法的缺点

对于给定查询条件、查找匹配实体的任务，entity bean的finder方法是很有用的。但是，如果用finder方法执行大型查询，就会产生巨大的性能消耗。如果entity bean是远程的，finder方法就会返回一个由远程引用组成的集合；如果entity bean是本地的，则会返回一个由本地引用构成的集合。另外，entity bean还可以有一些EJB Select方法，也与finder方法类似，不过EJB Select方法主要是在entity bean内部使用，不暴露给客户端。Entity bean使用Select方法，借助一种EJB查询语言（EJB-QL）编写的预定义查询，获取其他相关的entity bean。

在使用远程entity bean的情况下，对entity bean的每次调用都是一个远程网络调用，所以可能代价很高。而在使用本地entity bean的情况下，对entity bean的调用就是本地的，比起远程调用要高效得多。但是如果用finder或Select方法调用多个entity bean以获得一组数值，这还是可能成为代价很高的一种操作。

还有一点需要注意：根据EJB技术规范，容器会对finder方法找到的entity bean执行ejbActivate（EJB激活）方法，这可能会消耗可观的资源。有些厂商提供的容器实现，甚至还会

在finder方法中引入额外的负载，因为它们可能还会在这里就把entity bean实例和EJBObject实例关联起来，以便客户端访问entity bean。可是，如果客户端并不希望访问这些entity bean、不想调用它们的方法，那么这个做法就会造成不必要的资源消耗。

如果应用系统的查询会产生大量的匹配结果，那么以上负载就可能降低系统的性能。

## 结构

445

图7-46是值列表处理器模式的类图。

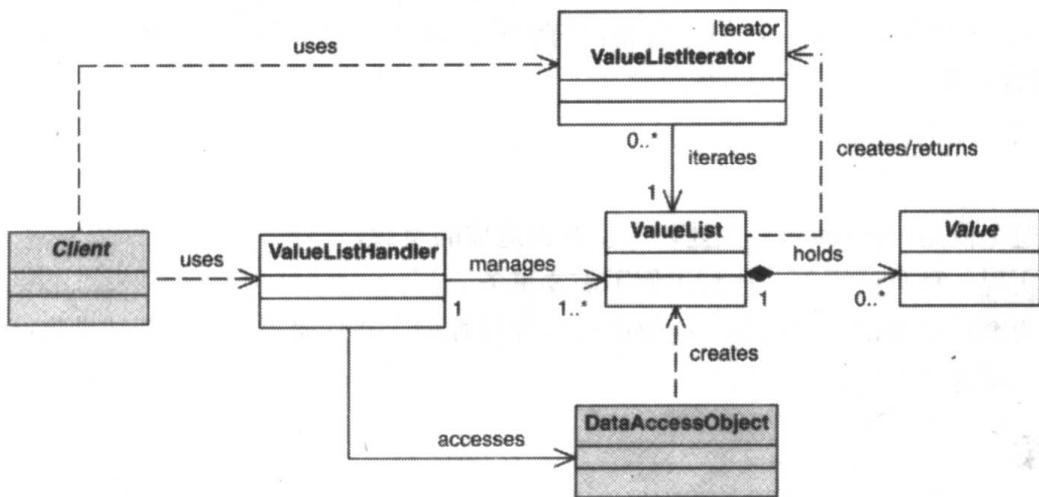


图7-46 值列表处理器模式类图

## 参与者和责任

图7-47体现了值列表处理器模式中的交互。

### Client（客户端）

Client（客户端）也就是任何需要执行查询、获取大型结果集的客户端。它可以是一个表现层组件，需要把查询结果显示给用户；也可以是一个session bean，其中封装了ValueListHandler（值列表处理器）。

### ValueListIterator（值列表迭代器）

ValueListIterator（值列表迭代器）提供了一种迭代机制，可以让客户端遍历ValueList（值列表）的内容。

### ValueListHandler（值列表处理器）

ValueListHandler（值列表处理器）执行查询，获取查询结果数据，并采用一个私有的集合（也就是ValueList对象）管理这些结果。ValueListHandler通常使用数据访问对象创建、操纵ValueList（值列表）集合。当客户端请求结果数据的时候，ValueListHandler根据原始的ValueList创建一个子列表，把这个子列表传送给Client（客户端）。通常，ValueListHandler只管理一个ValueList。不过，当ValueListHandler必需组合、处理多次查询的结果的时候，它也能管理一个以上的ValueList实例。

446

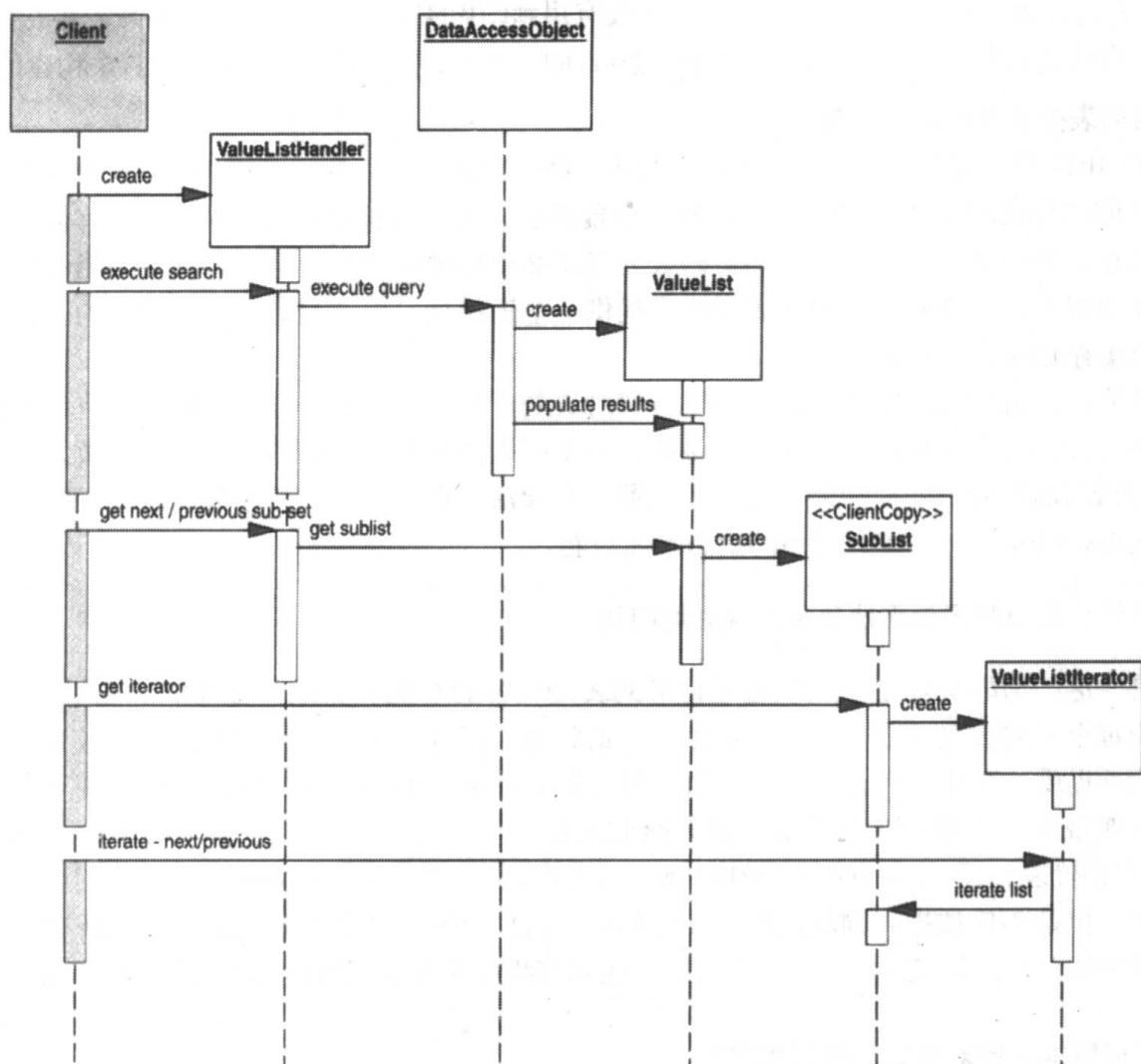


图7-47 值列表处理器模式交互图

### DataAccessObject (数据访问对象)

ValueListHandler (值列表处理器) 使用DataAccessObject (数据访问对象) 访问数据源，执行查询，获取结果。

### ValueList (值列表)

ValueList (值列表) 是一个集合，其中盛放着查询的结果数据。通常可以使用Java Collection (集合) API中的List (列表) 来实现它，或者还可以按照特殊需求实现自己定制的List。

447

### Value (值)

Value (值) 代表查询获得的结果数据对象。

## 策略

### POJO处理器策略

可以用POJO实现值列表处理器。这时，处于任何层次上的、需要迭代和数据缓存功能的客

户端、都可以使用这个值列表处理器。对于没有用到EJB的应用系统来说，这个策略是很有用的。比如，有些比较简单的应用系统，就是通过servlet、JSP、应用服务和数据访问对象构建的。

### 值列表处理器会话门面策略

在EJB应用系统中，通常要使用业务代表，经过会话门面，然后才能访问粗粒度的服务。虽然这里值列表处理器也可以用POJO实现，可能仍会考虑使用会话门面封装它，这样就能够在业务层缓存、管理查询的结果。因为值列表处理器必须要维护结果数据，也就必须用有状态的会话门面实现它。这样做能给值列表处理器提供一个远程接口，从而能使用有状态的会话门面在业务层缓存和管理结果数据。

但是，会话门面还提供了其他一些与值列表处理器无关的业务方法，所以，应该把两个会话门面分开：一个是提供其他业务方法的，一个则提供值列表处理器方法。这样做，就能够把其他会话门面和列表处理功能分割开，只留一个会话门面单独负责管理值列表处理器，充分利用有状态session bean在业务层中的状态管理功能。

### 设计手记：为值列表处理器单独实现会话门面

在会话门面的实现中，应该避免把值列表处理器的功能和其他业务功能混在一起。相反，应该为每个值列表处理器实现一个单独的会话门面。而且，还必须为每个这样的值列表处理器会话门面提供一个业务代表。比如，如果用一个session bean AccountSearchFacade（账户查询门面）实现了一个值列表处理器，就要给它配备一个单独的业务代表AccountSearchDelegate（账户查询代表）。可能另有AccountDelegate（账户代表）和 AccountFacade（账户门面）实现其他用例，但是不应该把查询功能混杂在它们中。这样分隔了各个会话门面之后，所有的查询过程都被封装在不同的组件中了，从而有助于区别有状态的会话门面和无状态的会话门面。

448

### 使用数据访问对象获取值列表策略

值列表处理器模式中的一个主要参与者就是这个处理器所管理的值列表。值列表通常是由数据访问对象获取的，而数据访问对象采用的策略既可以是传输对象集合策略，也可以是RowSet包装器列表策略。

可以根据应用需求决定何种策略最好。如果查询是预先确定了的，并且已知这些查询只返回比较小的结果集，可以使用传输对象集合策略。如果不能预知查询的大小，或者所执行的查询肯定会返回大量结果，那么最好还是使用RowSet包装器列表策略。

### 设计手记：值列表处理器：从数据访问对象获取何种类型的对象

正如数据访问对象模式的多个策略中介绍的一样，值列表处理器从数据访问对象获得的查询结果可以有很多形式： JDBC ResultSet（结果集）， JDBC RowSet（行集），或者Java Collection（集合）——比如List（列表）。虽然可以让数据访问对象直接返回ResultSet，但是，你可能还是会考虑一下：要从集成层把结果数据返回给业务层和其他层次，究竟哪一种对象形式更好呢？重要的是应该记住： ResultSet和关系型数据库的数据表结构是紧密相关的。所以，如果要与ResultSet交互，那么业务层（客户端）也就要知道表的结构、字段的位置以及各字段

的数据类型。

而且，`ResultSet`（结果集）还负责维护、管理数据库连接，所以把它从集成层传递出来，就可能破坏了它的封装和保护，甚至会在数据库连接上引发性能问题或误用。`RowSet`（行集）在`ResultSet`之上提供了一种改良的接口。正如数据访问对象模式中介绍的，可以实现自己的`RowSet`，或者使用其他的离线`RowSet`<sup>Θ</sup>，这样，在数据访问对象返回`RowSet`之前，就已经释放了数据库连接。

但是，如果要在业务层使用`RowSet`，那也还是跟`ResultSet`有一样的毛病：业务层也必须知道表的结构、字段的位置、字段的数据类型。

所以一个更好的方案是，让数据访问对象封装`RowSet`和`ResultSet`接口，不把任何与`java.sql`或`javax.sql`包相关的东西暴露给数据访问层以外的部分。而数据访问对象可以返回一个集合或者列表，用它来遍历查询结果，并且也为数据提供了一种面向对象的（而不是关系型的）视图。所以，在数据访问对象的策略中，我们推荐用传输对象集合策略或`RowSet`包装器列表策略实现值列表处理器策略。

449

## 效果

- 提供了EJB finder方法的一种高效的替代方案

值列表处理器提供了一种避免使用EJB finder执行查找的替代方案，在大量查询的情况下，finder方法效率低下。

- 缓存查询结果

如果客户端需要显示一个大型的结果集的子集，那么就需要把结果集缓存起来。结果集可以采取两种形式：既可以使用DAO传输对象集合策略，把结果集实现为一种可以迭代访问的传输对象集合；也可以使用DAO RowSet包装器列表策略，这时结果集就是一种定制的List（列表）实现，其中封装了一个JDBC `RowSet`。

- 提供了灵活的查询功能

可以通过加入一些特殊的查询机制或通过使用模板方法在运行时构造查询参数等方法，实现非常灵活的值列表处理器。换句话说，使用了值列表处理器，开发者就可以不受EJB finder方法的限制，实现智能查询、智能缓存的算法。

- 提高了网络性能

网络性能提高了，因为不用给客户端传送整个的结果集，而是按照请求的需要，只传送结果的一个子集。如果客户端只显示了最开始的几条数据，然后就把查询抛开了，那么网络带宽也没有浪费，因为剩下的数据只是在服务器端缓存而已，不会送给客户端。

但是，如果客户端/用户确实要处理整个结果集，那么为了获取结果集，就要对服务器执行多次远程调用。如果客户端/用户从一开始就知道自己需要整个结果集，那么可以给值列表处理器添加一个方法，让它用一次方法调用就把整个结果集发送给客户端。

<sup>Θ</sup> 离线`RowSet`，也就是说该`RowSet`已经失去数据库连接。

- 能够推迟对entity bean事务的使用

在服务器端缓存结果、尽可能降低finder产生的负载，也有助于优化事务管理。举个例子，有一个查询，使用值列表处理器显示图书列表，在获取列表的过程中却没有调用Book（图书）entity bean的finder方法。只是在其后，当用户要修改一本书的详细信息的时候，客户端调用了会话门面，而门面则按照该用例需要的事务语义，定位并调用相关的Book entity bean实例。<sup>①</sup>

450

- 明确了系统的层次划分，分离了不同关注点

值列表处理器封装业务层的列表处理功能，而且相应地使用了集成层的数据访问对象。这种做法明确了应用系统中的层次划分，把业务逻辑留给业务层组件，数据访问逻辑则留给数据访问对象。

- 创建一个很大的传输对象列表可能是代价昂贵的

当数据访问对象执行一次查询、生成传输对象的集合时，如果查询结果返回的匹配记录数量很多，可能会消耗大量资源。这时就不应该把所有的传输对象实例都创建出来，而要限制在查询中DAO能获取的记录数量，具体做法是：设置DAO可以从数据库取回的最大结果数量。也可能会需要使用数据访问对象模式中的缓存RowSet策略和RowSet包装器列表策略。

### 设计手记：EJB Home业务方法

按照EJB 技术规范2.1版，EJB Home上也可以加入业务方法，这些方法由bean开发者实现。这些Home方法实现的是那些作用于一个以上的entity bean的“集合性”<sup>②</sup>的业务逻辑。虽然可以使用这些方法完成业务对象的集合性操作，但是它却不能像值列表处理器一样，在业务层维护缓存的列表，所以也就缺乏后者的灵活性。如果要使用EJB Home业务方法返回集合型的结果数据<sup>③</sup>，也可以利用EJB Home业务方法（而不是数据访问对象），为值列表处理器获取ValueList（值列表）。

但是，如果EJB Home业务方法本身也要使用数据访问对象，那么再在值列表处理器中使用EJB Home业务方法，就未免多此一举，可能会给系统增添一个新的间接层。

## 示例代码

### 实现值列表处理器

考虑这样一个例子：要获取并显示Project（项目）业务对象的列表。在这个例子中，就可以使用值对象列表模式。这个实现的示例代码如例7.43所示，该类被称为ProjectsListHandler（项目列表处理器），它负责提供项目的列表。这个类继承了抽象基类ValueListHandler（值列表

<sup>①</sup> 也就是说，在“列表”操作中，因为没有使用EJB finder方法，可以不涉及事务；而只在修改详细信息的时候再启用事务。这就“推迟”了对事务的使用，也就优化了资源管理。

<sup>②</sup> 所谓“集合性”，是指这种业务方法不只是针对单个entity bean（否则就可以在entity bean本身，而不是Home上实现了）；它可以操作多个entity bean的数据。

<sup>③</sup> “集合型的结果数据”，当然是指返回的往往是多条数据——比如一个列表。

处理器), 该基类则提供了一种对值对象列表模式的通用实现。

例7.44列出了ValueListHandler(值列表处理器)的示例代码(另外请参见例7.45)。例7.47中的ProjectsListIterator(项目列表迭代器)实现了迭代器接口ValueListIterator(值列表迭代器), 该接口的代码则如例7.46所示。例7.48中则是数据访问对象ProjectDAO(项目DAO)的相关示例代码, ValueListHandler使用该对象来执行查询, 获取匹配的结果。[451]

**注意:** ProjectListHandler示例中使用了一个POJO来实现值列表处理器。如果要使用值列表处理器会话门面策略, 就可以用一个有状态session bean来实现这个ProjectListHandler。

#### 例7.43 实现值列表处理器: ProjectListHandler

```

1
2 package com.corej2eepatterns.vlh;
3
4 // imports
5
6 public class ProjectListHandler
7 extends ValueListHandler {
8
9     private ProjectDAO dao = null;
10    ...
11
12    // 客户端创建一个ProjectTO(项目传输对象)实例, sets the
13    // 把该对象的各字段按照查询所需的条件设置,
14    // 并且把这个ProjectTO实例以projectCriteria(项目查询条件)
15    // 参数的形式传给构造函数和setCriteria()方法
16    public ProjectListHandler()
17        throws ProjectException, ListHandlerException {
18        try {
19            this.dao = PSADAOFactory.getProjectDAO();
20        } catch (Exception e) {
21            // 处理异常, 抛出ListHandlerException(列表处理器异常)
22        }
23    }
24
25    // 执行查询。只要正确地设置了查询条件, 客户端
26    // 就可以调用这个方法。
27    // 本方法执行查询, 并使用最新的数据
28    // 更新列表
29    public void executeSearch(ProjectTO projectCriteria)
30        throws ListHandlerException {
31        try {
32            if (projectCriteria == null) {
33                throw new ListHandlerException(
34                    "Project Criteria required...");
```

[452]

```

37         dao.findProjects(projectCriteria);
38         setList(resultsList);
39     } catch (Exception e) {
40         // 处理异常，抛出ListHandlerException (列表处理器异常)
41     }
42 }
43 }
```

**ValueListHandler (值列表处理器)** 基类是一个通用的迭代器，提供了迭代功能。

#### 例7.44 ValueListHandler (值列表处理器) 抽象基类

```

1
2 package com.corej2eepatterns.vlh;
3
4 // imports
5
6 public abstract class ValueListHandler {
7
8     List valueList;
9     ...
10
11    // 只需要实现这个方法
12    public abstract void executeSearch(Object criteria);
13
14    ...
15
16    protected void setList(List valueList) {
17        this.valueList = valueList;
18    }
19
20    public List getNextElements(
21        int startPosition, int endPosition) {
22        return valueList.subList(startPosition, endPosition);
23    }
24
25    public List getPreviousElements(
26        int startPosition, int endPosition) {
27        return valueList.subList(startPosition, endPosition);
28    }
29
30    public Object getValue(int index) {
31        return valueList.get(index);
32    }
33
34    public int size() {
35        return valueList.size();
36    }
37
38    // 如果需要多个值列表，
```

```

39     // 那么还要提供多个方法，接收参数list id (列表ID)
40     // 比如public List getPreviousElements(int listId,
41     // int startPosition, int endPosition)，等等
42
43     ...
44 }
45

```

#### 例7.45 ValueList（值列表）的实现：ProjectsValueList（项目值列表）

```

1
2 package com.corej2eepatterns.vlh;
3
4 // imports
5
6 public class ProjectsList extends ArrayList {
7     ...
8
9     // 如果必要，实现iterator()方法，返回你的定制
10    // 迭代器
11    public Iterator iterator() {
12        return new ProjectsListIterator(this);
13    }
14
15    // 如果必要，实现listIterator()方法，返回你的定制
16    // 迭代器
17    public ListIterator listIterator() {
18        return new ProjectsListIterator(this);
19    }
20
21    public List subList(int fromIndex, int toIndex) {
22        ProjectsList subList = new ProjectsList();
23        for (int index=fromIndex; index<toIndex; index++) {
24            subList.add(this.get(index));
25        }
26        return subList;
27    }
28
29    ...
30 }
31

```

454

#### 例7.46 ValueListIterator（值列表迭代器）接口

```

1
2 package com.corej2eepatterns.vlh;
3
4 // imports
5
6 // 本示例继承了ListIterator

```

```

7  public interface ValueListIterator extends ListIterator {
8      // 在此处定义一些便用方法
9  }

```

#### 例7.47 ValueListIterator（值列表迭代器）实现：ProjectsListIterator（项目列表迭代器）

```

1
2  package com.corej2eepatterns.vlh;
3
4  // imports
5
6  // 通常这个类会编写成你要实现的定制值列表类的
7  // 内部类。在本例中，ProjectListIterator（项目列表迭代器）本可以
8  // 实现为ProjectsList（项目列表）的内部类的。
9  // 这里我们完全是出于示例目的，所以还是把它们分成了
10 // 两个类。.
11
12 public class ProjectsListIterator implements ValueListIterator
13 {
14     private List projectsList;
15     private int currentIndex = -1;
16     private int size = 0;
17
18     public ProjectsListIterator(List projectsList) {
19         this.projectsList = projectsList;
20         size = projectsList.size();
21         currentIndex=0;
22     }
23
24     // 实现其他方法
25     public boolean hasNext() { ... }
26     public Object next() { ... }
27     public boolean hasPrevious() { ... }
28     public Object previous() { ... }
29     public int nextIndex() { ... }
30     public int previousIndex() { ... }
31     public void remove() { ... }
32     public void set(Object o) {...}
33     public void add(Object o) { ... }
34 }

```

455

#### 例7.48 ProjectDAO（项目数据访问对象）类

```

1
2  package com.corej2eepatterns.dao;
3
4  // imports
5
6  public class ProjectDAO {
7      final private String tableName = "PROJECT";

```

456

```

8
9      // select语句要使用的字段
10     final private String fields = "project_id, name, " +
11         "project_manager_id, start_date, end_date, " +
12         "started, completed, accepted, acceptedDate, " +
13         "customer_id, description, status";
14
15     // 以下列出的只是与ValueListHandler (值列表处理器)
16     // 相关的方法。
17     // 其他细节请参见数据访问对象模式。
18     ...
19     private List findProjects(ProjectTO projCriteria)
20     throws SQLException {
21
22         Statement stmt= null;
23         List list = null;
24         Connection con = getConnection();
25         StringBuffer selectStatement = new StringBuffer();
26         selectStatement.append("SELECT " + fields + " FROM " +
27             tableName + "where 1=1");
28
29         // 根据projCriteria 参数中设置的值,
30         // 把其他的一些条件添加到
31         // where子句中
32
33         if (projCriteria.projectId != null) {
34             selectStatement.append (" AND PROJECT_ID = ' " +
35             projCriteria.projectId + "' ");
36         }
37         // 检查其他字段, 把查询条件添加到where子句里
38         ...
39
40         try {
41             stmt = con.prepareStatement(selectStatement);
42             stmt.setString(1, resourceId);
43             ResultSet rs = stmt.executeQuery();
44             list = createResultsList(rs);
45             stmt.close();
46         }
47         finally {
48             con.close();
49         }
50         return list;
51     }
52
53     private List createResultsList(ResultSet rs)
54     throws SQLException {

```

```

55     ArrayList list = new ArrayList();
56     while (rs.next()) {
57         int i = 1;
58         ProjectTO proj = new ProjectTO(rs.getString(i++));
59         proj.projectName = rs.getString(i++);
60         proj.managerId = rs.getString(i++);
61         proj.startDate = rs.getDate(i++);
62         proj.endDate = rs.getDate(i++);
63         proj.started = rs.getBoolean(i++);
64         proj.completed = rs.getBoolean(i++);
65         proj.accepted = rs.getBoolean(i++);
66         proj.acceptedDate = rs.getDate(i++);
67         proj.customerId = rs.getString(i++);
68         proj.projectDescription = rs.getString(i++);
69         proj.projectStatus = rs.getString(i++);
70         list.add(proj);
71     }
72     return list;
73 }
74 ...
75 }
```

457

## 相关模式

- **迭代器 [GoF]**

值列表处理器模式使用了GoF书（《设计模式：可重用面向对象软件的基础》）中介绍的迭代器模式。

- **数据访问对象**

值列表处理器使用了数据访问对象执行查询，至于返回的查询结果，既可以采用DAO传输对象集合策略，返回传输对象的一个集合；也可以采用DAO RowSet包装器列表策略，返回一个定制的List（列表）实现。

- **会话门面**

值列表处理器常常被实现为一种特别形式的会话门面，负责管理查询结果，并提供一个远程接口。有些应用系统可能会让会话门面既暴露其他业务方法，也包含值列表处理器的功能。不过，最好还是把值列表处理器的列表处理功能和会话门面的业务方法区别开。所以，如果值列表处理器需要一个远程的接口，就应该单独使用一个session bean，作为门面封装这个值列表处理器。

458

## 第8章 集成层模式

本章将涉及下列主题:

- **数据访问对象**
- **服务激活器**
- **业务领域存储**
- **Web Service中转**

459  
461

在设计企业级应用时，如果希望在不同的系统之间共享数据，或者希望在不同的系统之间共享服务，那么集成层模式就是一种非常好的选择。

集成层模式是通过一个或多个中间层来实现不同系统之间的集成的。

集成层模式可以分为以下几种类型：

• **数据访问对象（DAO）模式**：通过一个或多个中间层来实现不同系统之间的数据共享。

• **服务激活器（SA）模式**：通过一个或多个中间层来实现不同系统之间的服务共享。

• **业务领域存储（BFS）模式**：通过一个或多个中间层来实现不同系统之间的业务逻辑共享。

• **Web Service中转（WS）模式**：通过一个或多个中间层来实现不同系统之间的Web Service共享。

集成层模式的优点在于能够提高系统的可维护性、可扩展性和可重用性。

缺点在于集成层模式可能会增加系统的复杂性，同时也会增加系统的性能开销。

因此，在设计企业级应用时，需要根据实际情况选择合适的集成层模式。

在设计企业级应用时，如果希望在不同的系统之间共享数据，或者希望在不同的系统之间共享服务，那么集成层模式就是一种非常好的选择。

集成层模式是通过一个或多个中间层来实现不同系统之间的集成的。

集成层模式可以分为以下几种类型：

• **数据访问对象（DAO）模式**：通过一个或多个中间层来实现不同系统之间的数据共享。

• **服务激活器（SA）模式**：通过一个或多个中间层来实现不同系统之间的服务共享。

• **业务领域存储（BFS）模式**：通过一个或多个中间层来实现不同系统之间的业务逻辑共享。

• **Web Service中转（WS）模式**：通过一个或多个中间层来实现不同系统之间的Web Service共享。

集成层模式的优点在于能够提高系统的可维护性、可扩展性和可重用性。

缺点在于集成层模式可能会增加系统的复杂性，同时也会增加系统的性能开销。

## 数据访问对象

### 问题

需要将数据访问及操作的逻辑封装在一个单独的层次中。

很多真实的J2EE应用系统将持久对象实现为业务对象，具体的实现技术则可能是POJO或者entity bean。如果应用系统需求更简单一些，也可以放弃业务对象，代之以会话门面、应用服务或者别的助手对象——这些助手对象将直接访问并操作持久化存储介质中的数据。不论是业务对象还是这些助手组件，它们都需要访问持久化存储介质中的业务数据。

通常情况下，大多数企业级应用会使用关系型数据库管理系统（RDBMS）作为持久化存储介质。但是，企业数据也可能存在于别的地方，例如大型主机或者遗留系统、轻量级目录访问协议（LDAP）仓库、面向对象数据库（OODB）、普通文件等。另外，可以把外部系统的服务所提供的数据也视为持久化数据，这样的例子包括B2B集成系统、信用卡机构的服务等等。

对于这些形形色色的持久化存储介质，它们的访问机制、支持的API和功能特性也是各不相同的。即便是遵从同一套API，底层的实现者也可能在标准的特性之外再提供一些专有的扩展。

如果将持久化逻辑与应用逻辑混淆，就会导致应用程序直接依赖于持久化存储机制的实现。一旦在组件中出现这样的代码级依赖，再想把应用程序从一种数据源移植到另一种数据源就会困难重重。当数据源发生变化时，组件也必须加以修改才能使用新的数据源。

### 约束

- 需要实现一个数据访问机制，用于访问、操作持久化存储介质中的数据。
- 需要消除应用程序其余部分对持久化存储实现机制之间的耦合。
- 需要为不同类型的数据源（例如RDBMS、LDAP、OODB、XML仓库、普通文件等等）提供一个统一的持久化机制和统一的数据访问API。
- 需要对数据访问逻辑加以组织，将非标准的专有特性封装起来，使系统便于维护和移植。

462

### 解决方案

**使用数据访问对象提炼、封装对持久化存储介质的访问。数据访问对象负责管理与数据源的连接，并通过此连接获取、存储数据。**

数据访问对象（常被简称为DAO）实现了使用数据源所需的访问机制。不论使用哪种数据源，DAO总是向使用者提供统一的API。需要数据访问操作的业务组件只需使用DAO暴露给使用者的简单接口，DAO将数据源的实现细节完全隐藏起来。当底层数据源实现发生变化时，DAO暴露给使用者的接口不需要任何改变，因此可以放心地修改DAO的实现细节，而不会对DAO使用者的实现造成任何影响。从根本上来说，可以把DAO看作业务组件与数据源之间的一个适配器。

DAO应该被实现为无状态的对象。它不对任何查询操作的结果（或使用者以后可能需要使用的其他任何数据）进行缓存。因此，DAO是轻量级的对象，不存在出现线程或同步问题的可能性。DAO封装了底层持久化API的细节。举例来说，当应用程序使用JDBC作为持久化手段时，

DAO将所有对JDBC的使用都封装在数据访问层内部，也不会向数据访问层之外的客户端暴露任何属于java.sql.\*或javax.sql.\*包的异常、数据结构、对象或者接口。

## 结构

图8-1展示了数据访问对象模式的结构。

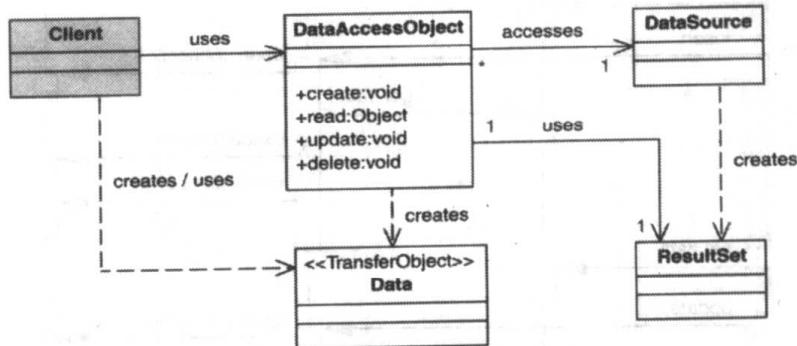


图8-1 数据访问对象模式类图

463

## 参与者和责任

图8-2展示了在一次基本的操作——从数据源获取数据——中，数据访问对象模式各参与者的交互情况。图8-3是同一幅图的第二部分。

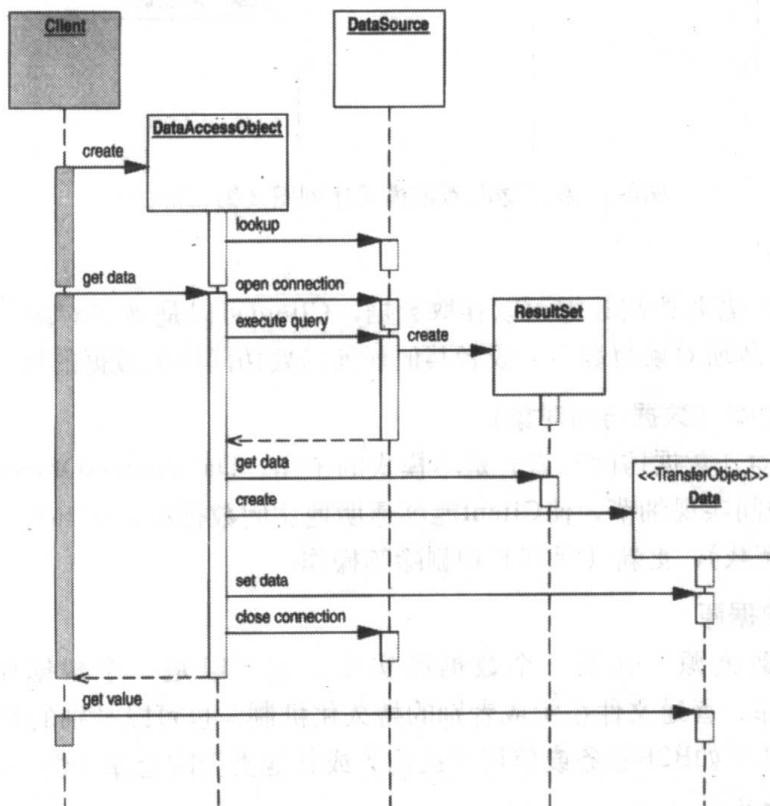


图8-2 数据访问对象模式序列图（第一部分）

464

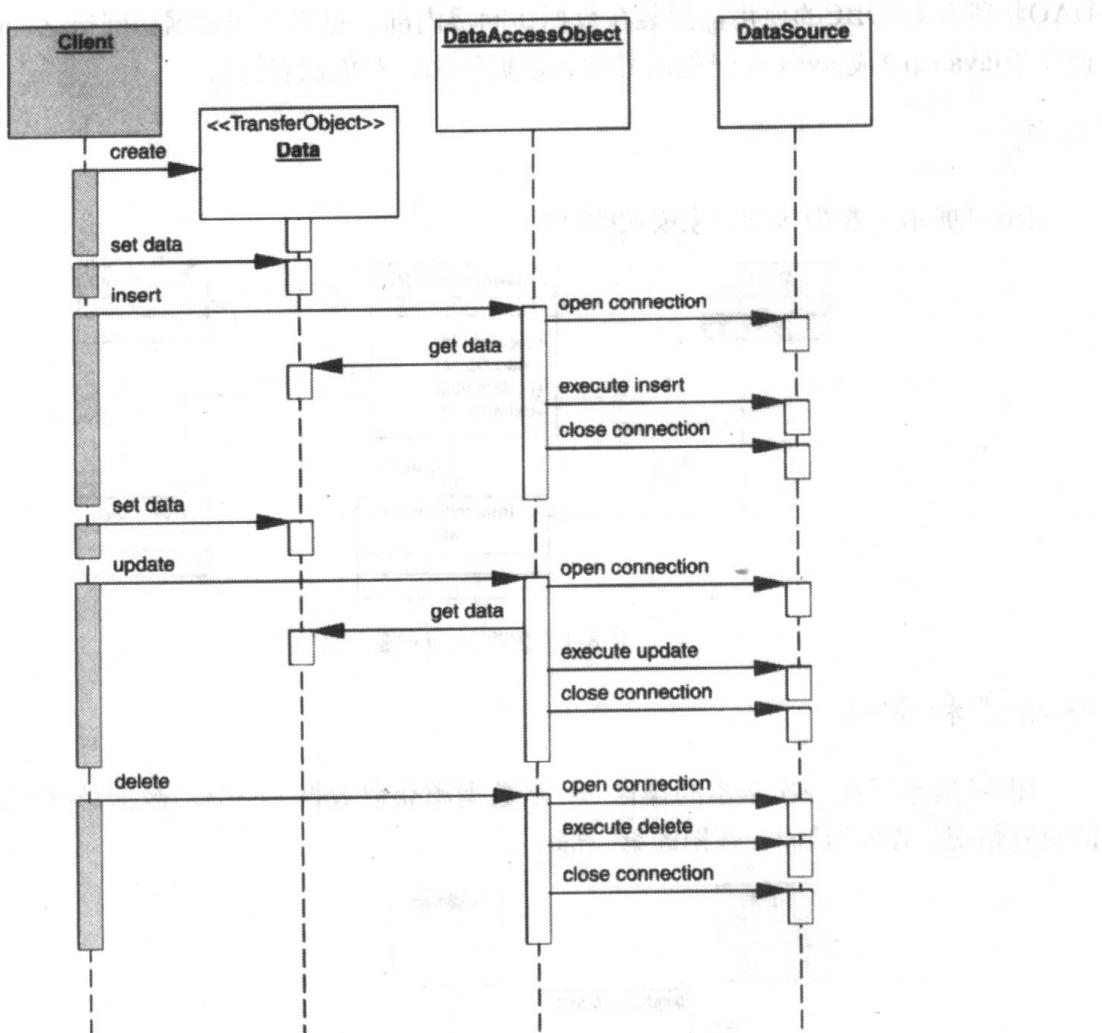


图8-3 数据访问对象模式序列图（第二部分）

### Client（客户端）

Client（客户端）需要访问数据源以存取数据。Client可以是业务对象、会话界面、应用服务、值列表处理器、传输对象组装器，或者其他任何需要访问持久数据的助手对象。

### DataAccessObject（数据访问对象）

DataAccessObject（数据访问对象）是本模式的主角。DataAccessObject对Client（客户端）隐藏了底层的数据访问实现细节，使Client能够透明地访问数据源。DataAccessObject实现了创建（插入）、查找（装载）、更新（保存）和删除等操作。

### DataSource（数据源）

DataSource（数据源）代表一个数据源实现。它可以是一个数据库，例如RDBMS、OODBMS、XML仓库、普通文件系统或者别的持久化机制；也可以是别的系统（例如遗留系统或大型主机）、服务（例如B2B服务或信用卡机构）或其他类型的仓库（例如LDAP）。

### ResultSet（结果集）

ResultSet（结果集）代表一次查询操作的结果。如果DataSource（数据源）是RDBMS（此

时应用程序使用JDBC API)，这个角色就由java.sql.ResultSet的实例来扮演。

### Data (数据)

Data (数据) 代表一个传输对象，它被用作数据的载体。DataAccessObject (数据访问对象) 可以使用传输对象将数据返回给客户端；也可以从客户端那里接收传输对象形式的数据，并用这些数据对数据源进行更新。

## 策略

### 定制数据访问对象策略

如果编写自己的数据访问层，可以使用数据访问对象模式和本节所介绍的各种策略。请注意，这些策略大多是互补的，可以将多种策略结合使用，使数据访问对象更强大、更灵活。但是，有些策略可能会给予实现增加过度的复杂性，因此应该审慎地使用它们。

在最基本的情况下，数据访问对象应该提供在数据库中创建、删除、更新、查找数据的操作，例8.2就是这样一个DAO实现的范例。一般来说，数据访问对象将通过传输对象与客户交换数据，如例8.1所示。此外，数据访问对象还应该提供用于查找多个结果的方法 (finder方法)，这类方法将返回一个列表 (List) 或别的集合 (Collection) 对象。

466

例8.1中的示例代码展示了一个用于表示“消费者”数据的传输对象，CustomerDAO用它来传递数据。CustomerDAO的示例实现如例8.2所示，它提供了在数据库中创建、更新、查找、删除消费者记录的方法。

### 例8.1 CustomerTO.java: CustomerDAO及其客户使用的传输对象

```

1 package com.corej2eepatterns.to;
2
3 public class CustomerTO implements java.io.Serializable {
4     private String id;
5     private String name;
6     private String address;
7     ...
8
9     public String getId(){ return id; }
10    public void setId(String id){ this.id = id; }
11    public String getName(){ return name; }
12    public void setName(String name){ this.name = name; }
13    public String getAddress(){ return address; }
14    public void setAddress(String address){
15        this.address = address;
16    }
17    // 其他getter、setter方法
18    ...
19 }
```

### 例8.2 CustomerDAO.java: 数据访问对象

```

1 package com.corej2eepatterns.dao;
2
```

```

3 // imports
4
5 public class CustomerDAO {
6     protected static final String FIELDS_INSERT =
7         "customer_name, customer_address, " +
8         "customer_contact, customer_phone, customer_email";
9
10    protected static final String FIELDS_RETURN =
11        "customer_id, " + FIELDS_INSERT;
12
13    protected static String INSERT_SQL =
14        "insert into customer ( " + FIELDS_INSERT +
15        " ) " + "values ( ?, ?, ?, ?, ? )";
16
17    protected static String SELECT_SQL = "select " +
18        FIELDS_RETURN +
19        " from customer where customer_id = ? ";
20
21    protected static String UPDATE_SQL =
22        "update customer set customer_name = ?, " +
23        "customer_address = ?, customer_contact = ?, " +
24        "customer_phone = ?, customer_email = ? " +
25        "where customer_id = ? ";
26
27    protected static String DELETE_SQL =
28        "delete from Customer where customer_id = ? ";
29
30 // 用于连接到后端数据库的数据源
31 private DataSource datasource;
32
33 public CustomerDAO() throws DAOException {
34     try {
35         // 以下代码只是为了清楚起见才给出的。通常
36         // 数据源寻址由服务定位器完成,
37         // DAO只是用服务定位器来获取
38         // 数据源。
39         InitialContext initialContext =
40             new InitialContext();
41         datasource = (DataSource) initialContext.lookup(
42             OracleDAOFactory.DATASOURCE_DB_NAME);
43     } catch (NamingException e) {
44         throw new DAOException (
45             "Cannot locate data source at " +
46             DAOFactory.DATASOURCE_DB_NAME, e);
47     }
48 }
49
50 public String create(CustomerTO cust) throws DAOException {

```

468

```
51     // 初始化变量
52     Connection con = getConnection();
53     String customerId = null;
54
55     PreparedStatement prepStmt = null;
56     try {
57         // 创建并构造statement
58         prepStmt = con.prepareStatement(INSERT_SQL);
59         int i = 1;
60         prepStmt.setString(i++, cust.getName());
61         prepStmt.setString(i++, cust.getAddress());
62         ...
63
64         // 执行statement
65         prepStmt.executeUpdate();
66
67         // 获取新创建的客户ID值
68         ...
69
70     } catch (Exception e) {
71         // 处理异常
72     } finally {
73         // 关闭连接
74     }
75
76     // 返回新创建的客户ID值
77     return customerId;
78 }
79
80 public CustomerTO find(String customerId)
81     throws DAOException {
82     // 初始化变量
83     CustomerTO cust = null;
84     Connection con = getConnection();
85     PreparedStatement prepStmt = null;
86     ResultSet rs = null;
87
88     try {
89         // 构造statement, 获取结果
90         prepStmt = con.prepareStatement(SELECT_SQL);
91         prepStmt.setString(1, customerId);
92         rs = prepStmt.executeQuery();
93         if (rs.next()) {
94             // 使用rs中的数据创建传输对象
95             cust = new CustomerTO();
96             cust.setId(rs.getString(1));
97             cust.setName(rs.getString(2));

```

```

98      . . .
99      }
469 100 } catch (Exception e) {
101     // 处理异常
102 } finally {
103     // 关闭连接
104 }
105 return cust;
106 }

107
108 public void update(CustomerTO cust) throws DAOException {
109     Connection con = null;
110     PreparedStatement prepStmt = null;
111     try {
112         // 准备statement
113         con = getConnection();
114
115         prepStmt = con.prepareStatement(UPDATE_SQL);
116         int i = 1;
117
118         // 首先添加字段
119         prepStmt.setString(i++, cust.getName());
120         prepStmt.setString(i++, cust.getAddress());
121         . .
122
123         // 现在加入where参数
124         prepStmt.setString(i++, cust.getId());
125         int rowCount = prepStmt.executeUpdate();
126         prepStmt.close();
127         if (rowCount == 0) {
128             throw new DAOException(
129                 "Update Error:Customer Id:" + cust.getId());
130         }
131     } catch (Exception e) {
132         // 处理异常
133     } finally {
134         // 关闭连接
135     }
136 }
137
138 public void delete(String customerId) throws Exception {
139     // 设置变量
140     Connection con = getConnection();
141     PreparedStatement prepStmt = null;
142
143     try {
144         // 执行数据库更新

```

```

145     prepStmt = con.prepareStatement(DELETE_SQL);
146     prepStmt.setString(1, customerId);
147     prepStmt.executeUpdate();
148 } catch (Exception e) {
149     // 处理异常
150 } finally {
151     // 关闭连接
152 }
153 }
154
155 // 其他方法如finder等等
156 ...
157 }

```

### 数据访问对象工厂策略（2种）

借助抽象工厂（*Abstract Factory*）[GoF]和工厂方法（*Factory Method*）[GoF]模式，可以使数据访问对象的创建极具灵活性。

如果应用程序只使用一种持久化存储介质（例如Oracle RDBMS），也不需要切换底层的存储实现，就可以实现DAO工厂方法策略，用于创建应用程序所需的各种DAO（见例8.3）。

#### 例8.3 OracleDAOFactory.java：数据访问对象，工厂方法 [GoF]策略

```

1 package com.corej2eepatterns.dao;
2
3 // imports
4 public class OracleDAOFactory extends DAOFactory {
5
6     // 整个包级别的常数，用于按照数据源名称
7     // 实现JNDI寻址
8     static String DATASOURCE_DB_NAME =
9         "java:comp/env/jdbc/CJPOradb";
10
11    public CustomerDAO getCustomerDAO()
12        throws DAOException {
13        return (CustomerDAO) createDAO(CustomerDAO.class);
14    }
15
16    public EmployeeDAO getEmployeeDAO()
17        throws DAOException {
18        return (EmployeeDAO) createDAO(EmployeeDAO.class);
19    }
20
21    // 创建其他DAO实例
22    ...
23
24    // 用来创建DAO实例的方法。可以进一步优化，
25    // 缓存DAO类而不是每一次都创建。
26    private Object createDAO(Class classObj)

```

```

27         throws DAOException {
28     // 创建一个新的DAO——采用classObj.newInstance()或者
29     // 从缓存中获取; 然后返回DAO实例
30 }
31 }
```

*DAO工厂*方法策略是最常用的策略。还可以借助抽象工厂模式进一步提高工厂实现的灵活性。如果需要经常切换持久化存储介质，或者同时使用多种存储介质，就会需要这种灵活性。为了满足“灵活切换持久化存储介质”的要求，必须对多种数据源进行封装，每个DAO工厂提供针对一种持久化介质的DAO实现。大多数应用程序通常只使用一种数据源（例如Oracle RDBMS），因此*DAO工厂*方法策略一般就足够了；不过，对于复杂的商业框架来说，*DAO抽象工厂*策略带来的灵活性会非常有用。

*DAO抽象工厂*策略的示例代码如例8.4所示。

#### 例8.4 DAOFactory.java：数据访问对象工厂，抽象工厂[GoF]策略

```

1 package com.corej2eepatterns.dao;
2
3 // imports
4
5 // 抽象类: DAO工厂
6 public abstract class DAOFactory {
7
8     // 本工厂支持的DAO类型列表
9     public static final int CLOUDSCAPE = 1;
10    public static final int ORACLE = 2;
11    public static final int SYBASE = 3;
12
13    . .
14    // 需要创建的每一种DAO都有一个专门的
15    // 方法。各种具体工厂必须
16    // 实现这些方法。
17    public abstract CustomerDAO getCustomerDAO()
18        throws DAOException;
19    public abstract EmployeeDAO getEmployeeDAO()
20        throws DAOException;
21
22    . .
23    public static DAOFactory getDAOFactory(int whichFactory)
24        switch (whichFactory) {
25            case CLOUDSCAPE:
26                return new CloudscapeDAOFactory();
27            case ORACLE:
28                return new OracleDAOFactory();
29            case SYBASE:
30                return new SybaseDAOFactory();
31            . .
32        default:
```

```

33         return null;
34     }
35 }
36 }
```

### 传输对象集合策略

除了基本的创建、查找、更新、删除操作之外，大多数应用程序客户还需要一些这样的方法：根据特定的查询条件查找多个结果。数据访问对象可以实现一些finder方法，它们接受查询条件作为参数，并一次性返回指定个数的结果。数据访问对象根据传输对象携带的查询条件创建一条SQL语句，然后在数据源上执行该SQL查询，以获得一个ResultSet对象；随后，数据访问对象对ResultSet对象进行处理，从中取出客户要求的那么多行结果记录；针对每行记录，数据访问对象创建一个传输对象，并将其加入到一个集合（collection）中；最后，数据访问对象把这个集合返回给客户端。

如果客户端需要获取的结果总数不多，并且希望以传输对象的形式返回，这种策略就很合适。但是，如果查询操作返回的结果集非常庞大，这种策略就会耗费大量资源，因为它必须为每一行记录的数据创建一个传输对象实例。如果需要庞大的结果集，可以考虑使用缓存RowSet策略、只读RowSet策略或者RowSet包装器列表策略。473

传输对象集合策略的示例实现如例8.5所示。

#### 例8.5 CustomerDAO.java：传输对象集合策略

```

1 package com.corej2eepatterns.dao;
2
3 // imports
4
5 public class CustomerDAO {
6 . . .
7
8     // 创建一个传输对象列表，并返回该列表
9     public List findCustomers(CustomerTO criteria)
10        throws DAOException {
11
12     Connection con = getConnection();
13     ResultSet rs = null;
14     ArrayList custList = new ArrayList();
15     String searchSQLString = getSearchSQLString(criteria);
16
17     try {
18         con = getConnection();
19         java.sql.Statement stmt =
20             con.createStatement(. . . );
21         rs = stmt.executeQuery(searchSQLString);
22         while(rs.next()) {
23             // 使用rs中的数据创建传输对象
24             cust = new CustomerTO();
25             cust.setId(rs.getString(1));
26             cust.setName(rs.getString(2));
27             cust.setAddress(rs.getString(3));
28             cust.setCity(rs.getString(4));
29             cust.setZip(rs.getString(5));
30             cust.setPhone(rs.getString(6));
31             custList.add(cust);
32         }
33     } catch (SQLException e) {
34         throw new DAOException("Error in CustomerDAO.findCustomers()", e);
35     }
36     return custList;
37 }
```

```

26         cust.setName(rs.getString(2));
27         ...
28
29         // 把TO (传输对象) 添加到列表中
30         custList.add(cust);
31     }
32 } catch (Exception e) {
33     // 处理异常
34 } finally {
35     // 关闭连接
36 }
37 return custList;
38 }
39
40 . . .
41
42 }

```

474

### 缓存RowSet策略

对于需要返回多个结果元素的操作（例如搜索），数据访问对象固然可以创建一个传输对象的集合，并将其返回给客户。但是，如果搜索操作返回了一个庞大的列表，创建所有的传输对象将造成极大的开销。为了避免这种性能上的开销，可以使用离线的缓存RowSet（实现JDBC中的RowSet接口）。RowSet可以用于读取和更新数据，有很多公开的实现可供选择。Sun提供的RowSet实现是*CachedRowSet*、*JdbcRowSet*和*WebRowSet* [SUNRS]，其他厂商（例如Oracle）也提供了自己的RowSet实现[ORARS]，可以在这些公开的实现中挑选一个满足需要的。

在使用支持更新功能的RowSet实现（不管是在线的还是离线的）时要特别注意。如果只需要对结果列表进行只读操作（譬如在web页面上显示结果），只读RowSet策略或者RowSet包装器列表策略会更合适。

---

### 设计手记：关于RowSet实现

---

JDBC规范组计划在今后的版本中提供5个标准的RowSet实现，取代现在sun.jdbc.rowset包中提供的实现[SUNRS]：

- *JdbcRowSet*——一个在线RowSet，提供JavaBean语意。
- *CachedRowSet*——离线RowSet，提供JavaBean语意和可靠的同步机制。
- *WebRowSet*——离线RowSet，提供与XML数据源交互的同步机制。
- *FilteredRowSet*——离线RowSet，可以为RowSet中的数据提供过滤之后的内部/外部视图。
- *JoinRowSet*——离线RowSet，可以在多个CachedRowSet之间建立SQL JOIN关联。

475

例8.6是一个使用了缓存RowSet实现（sun.jdbc.rowset.CachedRowSet）的示例DAO。

#### 例8.6 CustomerDAO.java：缓存RowSet策略

```

1 package com.corej2eepatterns.dao;
2

```

```

3 // imports
4
5 public class CustomerDAO {
6     . .
7
8     // 使用查询执行的结果集创建
9     // CachedRowSet (缓存RowSet)
10    public RowSet findCustomersRS(CustomerTO criteria)
11        throws DAOException {
12
13        Connection con = getConnection();
14        javax.sql.RowSet rowSet = null;
15        String searchSQLString = getSearchSQLString(criteria);
16        try {
17            con = getConnection();
18            java.sql.Statement stmt =
19                con.createStatement(. . .);
20            java.sql.ResultSet rs =
21                stmt.executeQuery(searchSQLString);
22            rowSet = new CachedRowSet();
23            rowSet.populate(rs);
24        } catch (SQLException anException) {
25            // 处理异常...
26        } finally {
27            con.close();
28        }
29        return rowSet;
30    }
31
32    . .
33
34 }

```

### 只读RowSet策略

尽管缓存 RowSet 策略可以用于创建任何形式的 RowSet，但很多 J2EE 应用仅仅需要只读的 RowSet 就足够了。在这种时候，不一定要使用支持更新操作的 RowSet 实现，也可以实现一个自己的 RowSet，只提供读取数据的功能。这种策略被称为“只读 RowSet 策略”，它的结构如图 8-4 所示。

Client（客户端）请求 DataAccessObject（数据访问对象）执行搜索。在这里，Client 的角色通常也由 DataAccessObject 来扮演，参见“Rowset 包装器列表策略”一节的解释。

图 8-5 的序列图说明了只读 RowSet 策略的工作原理。

Client（客户端）使用 DataAccessObject（数据访问对象）执行搜索查询。DataAccessObject 执行查询，获得一个 ResultSet（结果集），并使用 ResultSet 创建一个 ReadOnlyRowSet（只读 RowSet）。ReadOnlyRowSet 从 ResultSet 中抽取出结果值，并将其保存在自己的一个数组中，这就是 DataRows（数据行）。

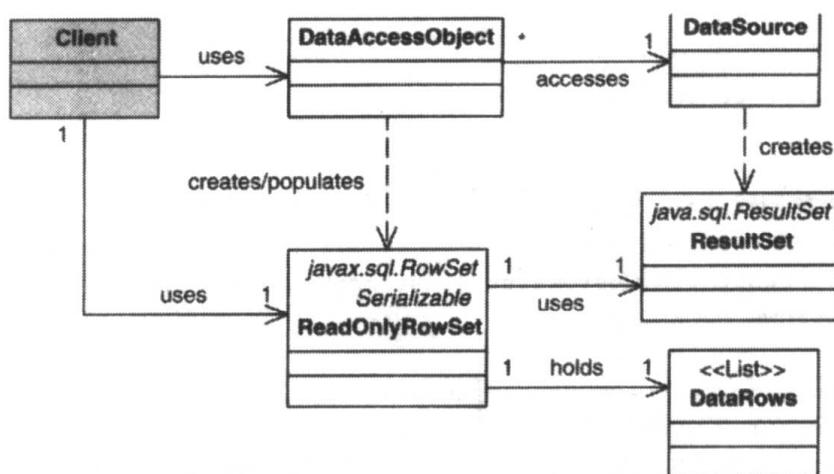


图8-4 只读Rowset策略的类图

477

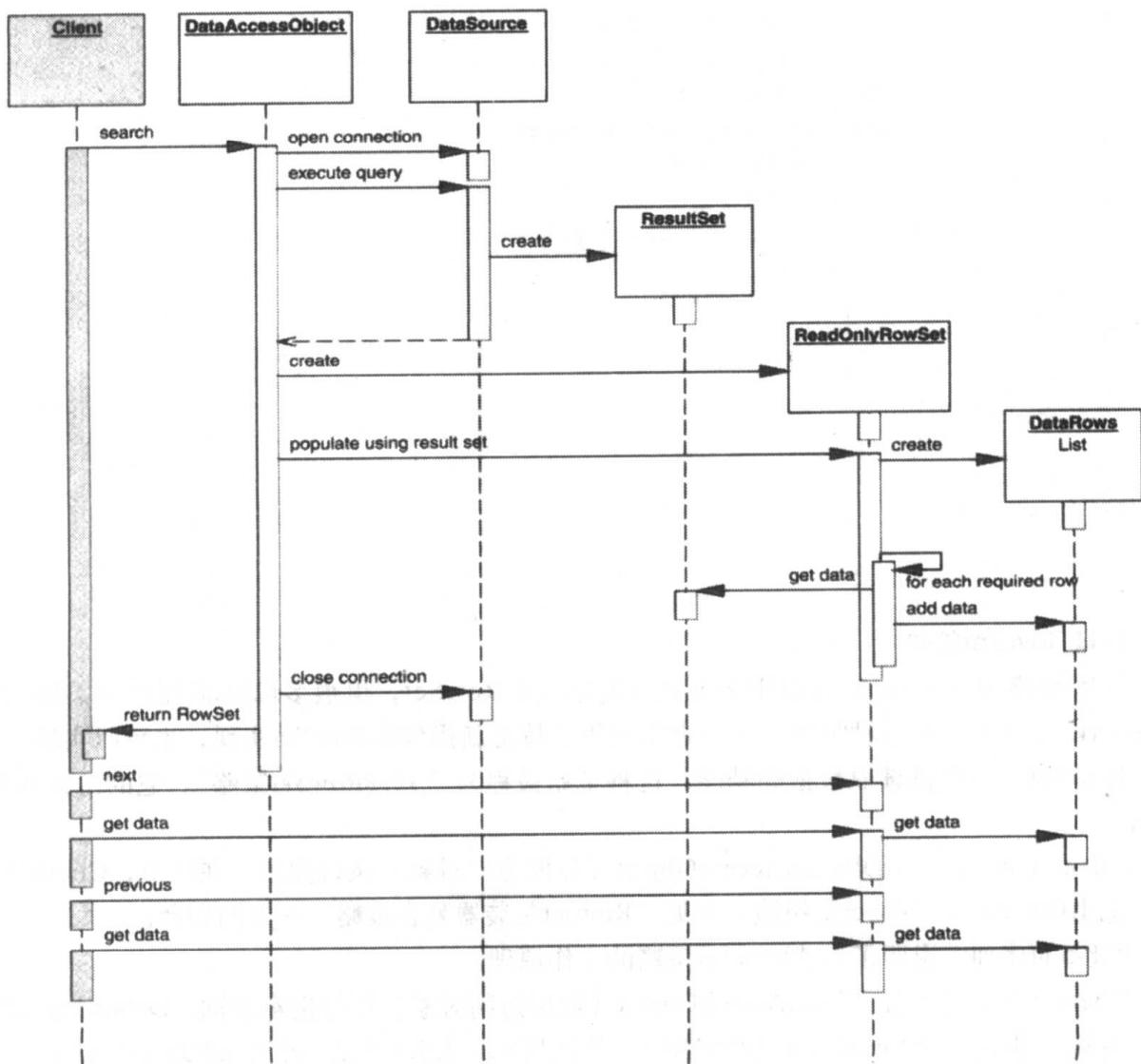


图8-5 只读RowSet策略序列图

`DataRow`s（数据行）列表创建完毕之后，`ReadOnlyRowSet`只从`ResultSet`中抽取出Client所请求的那么多行数据填入其中。这个步骤完成之后，与数据库的连接就被关闭，不再使用`ResultSet`。因此，本策略中所介绍的`ReadOnlyRowSet`是一个离线缓存`RowSet`实现。

下列代码示例展示了只读`RowSet`策略的实现（参见例8.7及例8.8）。例8.7是一个数据访问对象的代码，它首先执行查询操作，获得一个`ResultSet`实例，然后创建了一个`ReadOnlyRowSet`实例，并把`ResultSet`携带的数据填入其中（见例8.8）。CustomerDAO的使用者调用`findCustomersRORS()`方法，并将CustomerTO传输对象作为搜索条件传入其中，同时在`howManyRows`参数中执行返回的最大行数。

### 例8.7 CustomerDAO.java：创建`ReadOnlyRowSet`（只读`RowSet`）

```

1  package com.corej2eepatterns.dao;
2
3  // imports
4
5  public class CustomerDAO {
6
7
8      // 使用查询执行的结果集创建
9      // 创建ReadOnlyRowSet（只读RowSet）
10     public RowSet findCustomersRORS(CustomerTO criteria,
11         int startAtRow, int howManyRows)
12         throws DAOException {
13
14     Connection con = getConnection();
15     javax.sql.RowSet rowSet = null;
16     String searchSQLString = getSearchSQLString(criteria);
17
18     try {
19         con = getConnection();
20         java.sql.Statement stmt = con.createStatement(. . .);
21         java.sql.ResultSet rs =
22             stmt.executeQuery(searchSQLString);
23         rowSet = new ReadOnlyRowSet();
24         rowSet.populate(rs, startAtRow, howManyRows);
25     } catch (SQLException anException) {
26         // 处理异常...
27     } finally {
28         con.close();
29     }
30     return rowSet;
31 }
32
33
34 }
```

478

### 例8.8 `ReadOnlyRowSet.java`：填充`RowSet`

```

1  package com.corej2eepatterns.dao.rowset;
2
```

479

```

3 // imports
4
5 public class ReadOnlyRowSet implements RowSet, Serializable {
6
7     . . .
8
9     private Object[] dataRows;
10
11    . . .
12
13    /** 这是一个只读RowSet */
14    public boolean isReadOnly() {
15        return true;
16    }
17
18    public void setReadOnly(boolean flag) throws SQLException {
19        throw new SQLException(
20            "ReadOnlyRowSet: Method not supported");
21    }
22
23    // 复制结果集的数值 (不包括开头的startRow行),
24    // 由howManyRows 参数规定复制的
25    // 最大行数
26    public void populate(ResultSet resultSet,
27        int startRow, int howManyRows)
28        throws SQLException {
29
30        // 为了简洁,一些不太相关的代码没有显示...
31
32        // Create a list to hold the row values
33        List dataRows = . . . ;
34
35        // 根据元数据 (metadata) 判断列数
36        int numberofColumns =
37            resultSet.getMetaData().getColumnCount();
38
39        // 如果beginAtRowΘ (从某行开始) 的值设置了, 则越过开头的若干行
40        setStartPosition(startAtRow, resultSet);
41
42        // 如果总行数没有限定,
43        // 则取回结果集中的所有行
44        if (howManyRows <= 0) {
45            howManyRows = Integer.MAX_VALUE;
46        }
47        int processedRows = 0;

```

<sup>Θ</sup> 此处beginAtRow就是参数中的startRow。

```

48     while ((resultSet.next()) &&
49             (processedRows++ < howManyRows)) {
50         Object[] values = new Object[numberOfColumns];
51
52         // 从当前行读取数值，并保存在
53         // 值数组中
54         for (int i=0; i<numberOfColumns; i++) {
55             Object columnValue =
56                 this.getColumnValue(resultSet, i);
57             values[i] = columnValue;
58         }
59
60         // 把值数组添加到列表中
61         dataRows.add(values);
62     }
63
64 } // RowSet构造器到此为止
65
66 . . .
67
68 // 设置结果集，从给定行号处开始
69 private void setStartPosition(
70     int startAtRow, ResultSet resultSet)
71     throws SQLException {
72     if (startAtRow > 0) {
73         if (resultSet.getType() !=
74             ResultSet.TYPE_FORWARD_ONLY) {
75             // 利用JDBC 2.0 API移动游标
76             if (!resultSet.absolute(startAtRow)) {
77                 resultSet.last();
78             }
79         } else {
80             // 如果结果集不支持JDBC 2.0
81             // 则手工越过开头的beginAtRow行
82             for (int i=0; i< startAtRow; i++) {
83                 if (!resultSet.next()) {
84                     resultSet.last();
85                     break;
86                 }
87             }
88         }
89     }
90 }
91
92 // 从当前行读取一个列的数值，创建一个匹配的Java对象。
93 // 用于装载该数值。如果读取时发生错误，或者
94 // 是SQL null（空值），则返回null。

```

```

95     private Object getColumnValue(
96         ResultSet resultSet, int columnIndex) {
97
98
99
100    }
101
102    // 实现RowSet 和ResultInterface方法
103
104
105  }

```

### RowSet包装器列表策略

虽然传输对象集合策略可以用于实现finder方法，但如果查询返回的结果集非常庞大就必须创建大量的传输对象，这可能会造成巨大的开销，而客户端真正需要的只是其中的一小部分——客户端执行搜索之后，通常只使用前面的几条结果，剩下的就直接扔掉了。如果应用程序执行的查询会返回庞大的结果集，RowSet包装器列表策略的效率会更高——虽然实现这个策略需要一些额外的工作量，并且增加了复杂度。

不管选择哪种RowSet实现，都不希望将RowSet策略暴露给数据访问对象的外部用户。对于那些来自java.sql和javax.sql包的对象，最好是将用到它们的程序全部封装在数据访问对象内部。这样的封装确保了客户端不会依赖于低层次的JDBC接口（例如ResultSet和RowSet接口）。RowSet包装器列表策略使用只读RowSet策略来获取查询结果，然后对RowSet实例进行了封装，以一种面向对象的方式将结果交给客户端。这层包装是通过一个定制的List实现来完成的，这个List提供了对RowSet的遍历、缓存等功能，客户端可以通过它来操作被封装的RowSet——更为重要的是，这种操作是以面向对象的形式进行的，因为我们的List实现将RowSet中的数据当成一个个对象暴露给客户端。

482

RowSet包装器列表策略的优点在于：只有在真正必要的时候——也就是说，只有在客户端需要的时候，才创建并返回传输对象。而使用传输对象集合策略时，数据访问对象必须创建所有的传输对象，将它们装进一个Collection对象，然后才能把这个Collection返回给客户端。

图8-6的类图展示了RowSet Wrapper List策略的结构。

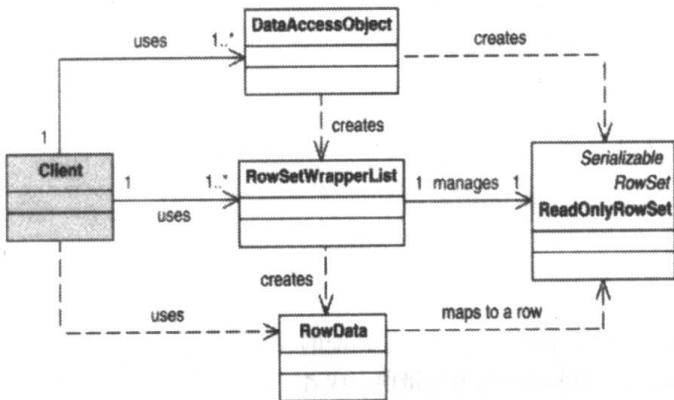


图8-6 RowSet包装器列表策略的类图

图8-7的序列图展示了RowSet包装器列表策略中各对象的交互情况。

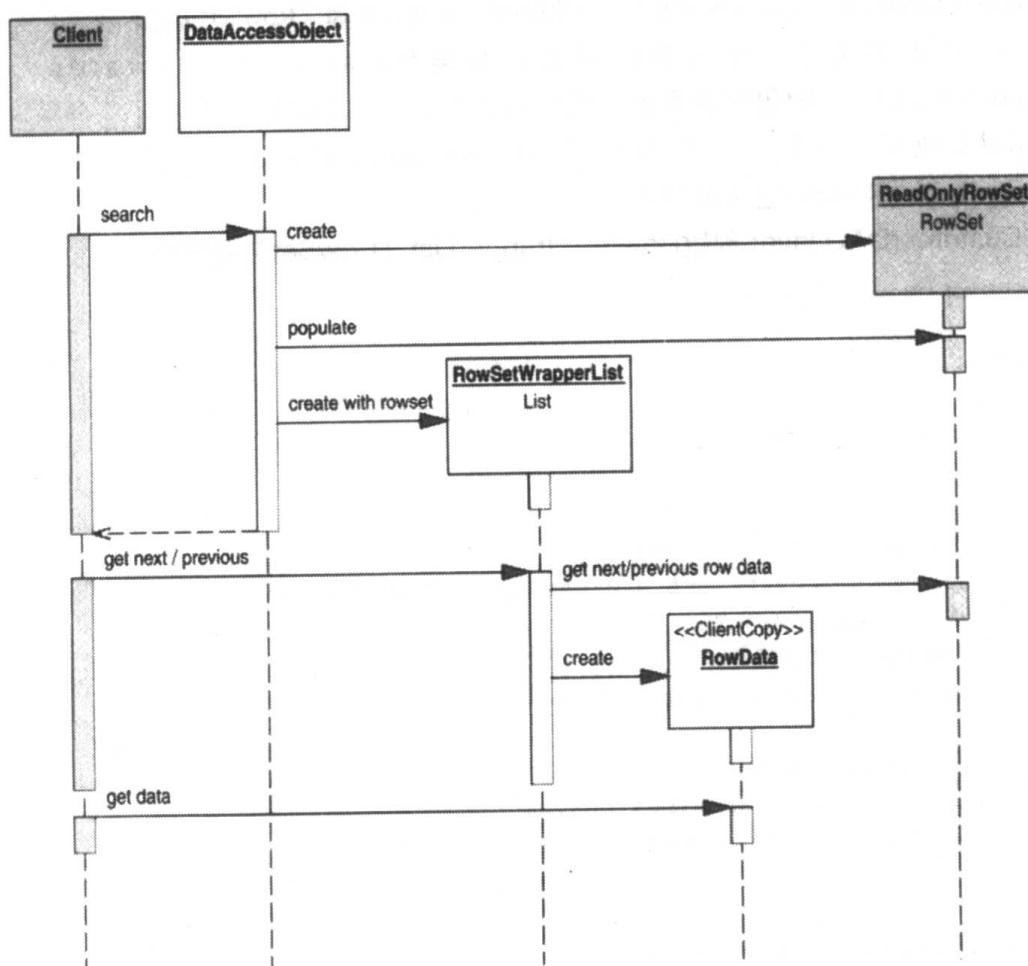


图8-7 RowSet包装器列表策略的序列图

DataAccessObject（数据访问对象）首先使用只读RowSet策略获得一个ReadOnlyRowSet（只读RowSet），然后用这个对象创建一个RowSetWrapperList（RowSet包装器列表）对象。随后，DataAccessObject把RowSetWrapperList返回给Client（客户端），后者用它遍历查询的结果。

RowSetWrapperList（RowSet包装器列表）实现了List接口，提供了遍历ReadOnlyRowSet（只读RowSet）中数据的功能。RowSetWrapperList对Client（客户端）封装了所有数据访问层的功能，Client无须对RowSet、ResultSet或其他来自java.sql和javax.sql包的类有任何了解。当客户端想要访问结果列表中的前一行或后一行时，RowSetWrapperList将返回一个传输对象。

483

## 设计手记

如果想要获得更高的效率，还可以再进行优化：不必在客户端每次请求前一行或后一行时都创建并返回一个传输对象，可以实现一个游标，它只需要创建一次，就可以反复使用。不过，在将游标返回给客户端之前，RowSetWrapperList（RowSet包装器列表）必须保证它指向了正确的行。为简单起见，我们没有选择游标，而是使用传输对象。

484 下列示例代码展示了RowSet包装器列表策略的实现（参见例8.9至例8.13）。在例8.9中，CustomerDAO 使用前面示例中获得的ReadOnlyRowSet创建了一个RowSetWrapperList（RowSet包装器列表）实例。

此外，我们还实现了一个定制的列表迭代器DataRowListIterator，将它作为 RowSetWrapperList（RowSet包装器列表）的内部类来实现，如例8.11所示。

最后，例8.13展示了一个简单的DAO客户端CustomerDAOClient，它要使用CustomerDAO和 RowSetWrapperList（RowSet包装器列表）。

#### 例8.9 CustomerDAO.java: 创建RowSetWrapperList（RowSet包装器列表）

```

1 package com.corej2eepatterns.dao;
2
3 // imports
4
5 public class CustomerDAO{
6     ...
7
8     public List findCustomersRL(
9         CustomerTO cust, int startAtRow, int howManyRows)
10    throws Exception{
11        // 创建查找SQL字符串
12        String searchSQLString = getSearchSQLString(cust);
13
14        execute the search
15        return executeSearch(searchSQLString,
16            StartAtRow, howManyRows);
17    }
18
19    private List executeSearch(
20        String searchSQLString, int startAtRow,
21        int howManyRows)
22        throws Exception {
23
24        RowSetWrapperList results = null;
25        try {
26            RowSet rowSet = getRORowSet(searchSQLString,
27                int startAtRow,
28                int howManyRows);
29
30            results = new RowSetWrapperList (rowSet);
31        } catch (Exception ex) {
32            throw new Exception(ex);
33        }
34
35        return results;
36    }
37
38    private String getSearchSQLString(CustomerTO cust) {

```

```

39     // 这是创建并返回“SELECT” SQL语句
40     // (作为字符串)的使用的方法，在WHERE语句中，
41     // 为已提供的CustomerTO传输对象
42     // 加入非空值
43 }
44
45 . .
46
47 }

```

### 例8.10 RowSetWrapperList.java

```

1 package com.corej2eepatterns.to.list;
2
3 // imports
4
5 // List接口的只读实现。
6 // 支持迭代器和绝对定位。
7
8 public class RowSetWrapperList
9 implements List, Serializable{
10
11     // 存放RowSet实例的变量
12     private RowSet rowSet;
13
14
15     public RowSetWrapperList(RowSet rowSet){
16         this.rowSet = rowSet;
17         ...
18     }
19
20     // 返回当前行作为传输对象
21     public Object get(int index){
22         try {
23             rowSet.absolute(index);
24         } catch (SQLException anException) {
25             // 处理异常
26         }
27         // 创建新传输对象并返回
28         return
29             TORowMapper.createCustomerTO(this);
30     }
31     ...
32
33     // 返回当前列表的子列表
34     public List subList(int fromIndex, int toIndex) {
35         // 用所需行创建新RowSet

```

```
36     ReadOnlyRowSet roRowSet = new ReadOnlyRowSet();
37     roRowSet.populate(this.rowSet, fromIndex, toIndex);
38
39     // 创建新RowSetWrapperList实例
40     // 并返回
41     return
42         new RowSetWrapperList(roRowSet);
43     }
44
45     // 在相应列中的清单,
46     // 返回其元素上的迭代器。
47     // 对相同的RowSetWrapperList对象,
48     // 可以定义多个独立的迭代器。
49
50     public Iterator iterator() {
51         try {
52             rowSet.beforeFirst();
53         } catch (SQLException anException){
54             System.out.println(
55                 "Error moving RowSet before first row." +
56                 anException);
57         }
58
59         return this.listIterator();
60     }
61
62     // 创建List迭代器, 使其能够在RowSet上进行
63     // 迭代
64     public ListIterator listIterator() {
65         // ListResultIterator被实现为内部类
66         return new DataRowListIterator();
67     }
68
69     // 实现List接口的方法
70
71     . . .
72
73 }
```

### 例8.11 RowSetWrapperList.java: 内部类DataRowListIterator

```

1  package com.corej2eepatterns.to.lists;
2
3  // imports
4
5  public class RowSetWrapperList
6  implements List, Serializable{
7
8  . . .
9
10 private class DataRowListIterator implements ListIterator {
11     int currentRow=0;
12     // 设置RowSet游标到下一行,
13     // 并返回传输对象
14     public Object next() {
15         // 从RowSetWrapperList的下一行中获得传输对象
16         currentRow++;
17         return this.get(currentRow);
18     }
19
20     public Object previous() {
21         // 从RowSetWrapperList的前一行中
22         // 获取传输对象
23         currentRow--;
24         return this.get(currentRow);
25     }
26 }
27
28 // 实现列表迭代器 (List Iterator) 接口
29 public boolean hasNext() {
30     // 用isLast、isAfterLast和isEmpty方法检查游标在
31     // RowSet中的位置, 相应地返回
32     // true或false
33 }
34
35 public boolean hasPrevious() {
36     // 用isFirst、isBeforeFirst和isEmpty方法检查游标在
37     // RowSet中的位置, 相应地返回
38     // true或false
39 }
40
41 // 实现其他ListIterator方法
42
43 public int nextIndex() {
44     . . .
45 }
46

```

```

47     public int previousIndex() {
48         . . .
49     }
50
51     // 不实现以下可选方法，而是抛出
52     // UnsupportedOperationException异常
53     public void set(Object o) {
54         throw new UnsupportedOperationException();
55     }
56
57     public void add(Object o) {
58         throw new UnsupportedOperationException();
59     }
60     . . .
61 }

```

### 例8.12 TORowMapper.java：示范实现

```

1  package com.corej2eepatterns.util;
2
3  // imports
4  public class TORowMapper {
5
6      // 创建Customer TO
7      public CustomerTO createCustomerTO(RowSet rowSet) {
8          CustomerTO to = new CustomerTO();
9          to.setId(getString(rowSet), 0);
10         to.setName(getString(rowSet), 1);
11         . . .
12     }
13
14     // 创建其他TO（传输对象）
15     . . .
16
17     // 实现create方法要用到的一些方法
18     protected boolean wasNull(RowSet rowSet) {
19         try {
20             return rowSet.wasNull();
21         } catch (SQLException e) {
22             throw new RuntimeException(e.getMessage());
23         }
24     }
25
26     protected String getString(RowSet rowSet, int columnIndex) {
27         try {
28             return rowSet.getString(columnIndex);
29         } catch (SQLException e) {
30             throw new RuntimeException(e.getMessage());
31         }

```

```

32     }
33
34     protected boolean getBoolean(
35         RowSet rowSet, int columnIndex) {
36         try {
37             return rowSet.getBoolean(columnIndex);
38         } catch (SQLException e) {
39             throw new RuntimeException(e.getMessage());
40         }
41     }
42
43     protected java.util.Date getDate(
44         RowSet rowSet, int columnIndex) {
45         try {
46             return rowSet.getDate(columnIndex);
47         } catch (SQLException e) {
48             throw new RuntimeException(e.getMessage());
49         }
50     }
51
52     // 其他基本类型的getXXX方法，用于给所有必要的数据类型
53     // 获取数据
54     . . .
55
56 }

```

490

### 例8.13 DAO客户端：使用RowSetWrapperList (RowSet包装器列表)

```

1  package com.corej2eepatterns.rowset;
2
3  // imports
4
5  public class CustomerDAOClient {
6
7  . . .
8
9  public void search() {
10    try {
11        CustomerDAO dao = new CustomerDAO();
12        CustomerTO criteria = new CustomerTO();
13
14        criteria.setZip("94539");
15
16        // 搜索所有邮政编码是94539的客户,
17        // 最多返回1000行匹配的结果
18        List searchResults =
19            dao.findCustomersRL(criteria, 0, 999);
20
21        int resultSize = searchResults.size();

```

```

22         for (int rowNum=0; rowNum < resultSize; rowNum++) {
23             CustomerTO customerTO =
24                 (CustomerTO)searchResults.get(rowNum);
25             System.out.println("Customer Row #" +
26                 rowNum + " has ");
27             System.out.println("Customer Id = " +
28                 customerTO.getId());
29             System.out.println("Name = " +
30                 customerTO.getName());
31             . . .
32         }
33         . . .
34         . . .
35
36     // 获得一个ListIterator (列表迭代器)
37     ListIterator iterator = searchResults.listIterator();
38
39     // 使用迭代器访问下一行
40     // 或上一行
41     . . .
42
43 } catch (Exception e) {
44     // 处理异常
45 }
46 }
47 . . .
48 }

```

491

## 设计手记：如何设计DAO

在数据访问对象模式中，我们已经多次提到：应该使用传输对象在数据访问对象和客户端之间传递数据。于是，问题也就来了：执行一次查询之后，应该返回给客户端什么对象？

尽管数据访问对象在执行查询之后会创建一个ResultSet实例，但最好是始终将ResultSet封装在数据访问对象内部。ResultSet实例是带有数据库连接的，将它返回给数据访问层之外的客户端会破坏封装，还可能引发连接管理的问题。另一个问题是：ResultSet是一个低层次的接口，与RDBMS中的关系型数据结构绑定在一起；而数据访问对象不仅要迎合RDBMS，也应该适用于其他持久化存储策略。因此，避免将“与实现细节相关的数据结构”（例如ResultSet）暴露到数据访问层之外是非常必要的。如果向客户端暴露了这些数据结构，会引发下列问题：

- 客户端不得不导入并处理来自java.sql包的接口和异常。
- 为了取出ResultSet中的数据，客户端必须指定数据库字段或别名。于是，客户端就不得不了解持久化实现的细节，例如关系表和字段。
- ResultSet携带了活跃的数据库连接。
- RowSet提供了比ResultSet略高的抽象级别，还提供了一些附加的方法，这使得它略微易用一点。看上去，用RowSet实例来返回结果似乎是不错的办法，但它也存在类似的问题：客

客户端将不得不导入并处理来自java.sql和javax.sql包的接口和异常。

- 为了从RowSet中取出数据，客户端必须指定数据库字段或别名。于是，客户端就不得不了解持久化实现的细节，例如关系表和字段。
- 如果使用在线的RowSet，当数据访问对象把RowSet返回给客户端之后，数据库连接将仍然保持活跃。
- RowSet与ResultSet的关系非常紧密。实际上，RowSet接口继承了ResultSet接口。

尽管存在这些问题，但“从数据访问对象返回RowSet对象”仍然不失为一个可行的做法——取决于如何使用返回的RowSet对象。另一方面，如果用传输对象在数据访问对象和客户端之间传递数据，则可以彻底解决客户端与JDBC API之间的耦合问题。[492]

此外，对于finder方法来说，比较简单的传输对象集合策略和略显复杂的RowSet包装器列表策略都是数据访问对象向外部客户端返回结果时不错的选择：两者都对客户端隐藏了RowSet实现。更重要的是，这两种策略使客户端可以通过面向对象的方式访问结果数据，而不必处理ResultSet和RowSet这样来自JDBC API的数据结构。

## 效果

### • 用松耦合处理程序集中控制

过滤器提供了一个处理多个请求的集中地方，其作用就像控制器。过滤器更适合处理一个目标源（如控制器）的多个处理程序的请求与答复。另外，控制器经常进行彼此不相关的日常服务，如验证、日志、加密等等。过滤器与处理程序的关系不太紧密，因此可以用于各种场合。

### • 对持久化数据的透明访问

借助数据访问对象对数据源的封装，客户端可以透明地访问各种持久化存储介质，不必了解存储介质的位置和实现。

### • 为数据库数据结构提供面向对象的视图和封装

客户使用传输对象或者数据游标对象（RowSet包装器列表策略）与数据访问对象交换数据，而不必依赖于低层次的、与数据库数据结构相关的实现细节（例如ResultSet和RowSet）。如果依赖这些实现细节，客户端就不得不了解数据库的表结构、字段名等信息。使用传输对象和数据游标，客户端可以用一种面向对象的方式处理数据。

### • 简化数据库移植

以DAO构成的数据访问层简化了应用程序在不同数据库实现之间的移植。客户无须了解底层的数据存储实现，因此移植动作只对DAO层产生影响。[493]

### • 降低客户代码的复杂度

DAO封装了所有与持久化介质交互的代码，客户端只需使用数据访问层提供的简单API即可。这降低了客户代码执行数据访问的复杂度，提高了可维护性和开发效率。

### • 将所有数据访问代码组织到一个独立的层次中

数据访问对象把实现数据访问功能的代码都组织到一个独立的层次中，将应用程序的其他

部分与持久化存储介质和外部数据源隔离开。由于所有数据访问操作都委派给DAO，这个层次就隔离了应用程序的其余部分和数据访问的实现。将数据访问操作集中在一个层次中，使得应用程序更容易维护和管理。

- 增加了一个层次

DAO在数据的使用者和数据源之间构成了一个新的对象层次，要获得本模式的好处，就需要设计并实现这个层次。这个层次看上去似乎增加了不必要的开发和运行开销，但为了消除数据访问实现与应用程序其余部分之间的耦合，它通常是必不可少的。

- 需要设计整个类体系 (class hierarchy)

如果使用工厂策略，需要设计并实现工厂和工厂所制造的产品（即DAO）的类体系。这会增加开发的工作量和设计的复杂程度，因此需要考虑清楚：是否真的需要它所提供的灵活性。在能够满足需要的前提下，尽量使用DAO工厂方法策略，只有在绝对必要的时候才使用DAO抽象工厂策略。

- 面向对象设计增加了复杂度

*RowSet*包装器列表策略封装了对数据访问层的依赖和JDBC API，提供给客户一个面向对象的结果数据视图。尽管这种策略避免了缓存*RowSet*策略直接使用JDBC RowSet API带来的种种弊端，相对于传输对象集合策略又有效率上的提升，但它也给实现增加了相当大的复杂度。需要权衡利弊，作出决定。

494

## 相关模式

- 传输对象

在最基本的情况下，数据访问对象使用传输对象与客户交换数据。在数据访问对象模式的其他策略中，传输对象模式也有应用，例如*RowSet*包装器列表策略就把一个列表作为传输对象返回给客户。

- 工厂方法 (*Factory Method*) [GoF] 和抽象工厂 (*Abstract Factory*) [GoF]

数据访问对象工厂策略使用工厂方法模式来实现具体工厂和它们的产品（即DAO）。为了增加灵活性，抽象工厂模式也得到了应用（参见数据访问对象工厂策略的介绍）。

- 中转 (*Broker*) [POSA1]

DAO模式与中转模式有着千丝万缕的联系，后者描述了分布式系统中消除客户端与服务器之间耦合的办法。DAO模式可以看作中转模式一个更为专门的应用，只用于消除资源层与位于其他层（例如业务层或者表现层）的客户之间的耦合。

- 传输对象组装器

传输对象组装器使用数据访问对象来获取数据，以便组装它所需要的传输对象，并将其作为模型数据发送给客户。

- 值列表处理器

值列表处理器需要对一个结果列表进行操作，这样的列表通常来自数据访问对象。虽然值列表处理器也可以处理*RowSet*对象，但数据访问对象最好是向它提供*RowSet*包装器列表。

495

## 服务激活器

### 问题

需要异步调用某些服务。

在企业级应用中，大多数处理是以同步形式进行的：客户端调用一个业务服务，然后等待业务服务完成处理并返回结果。但是，有时业务处理需要耗费相当可观的时间和资源，还可能跨越多个应用程序，甚至需要结合企业内外的多个应用程序才能完成。对于这些耗时较长的处理，要求应用程序客户端一直等待它完成是不切实际的。

### 约束

- 需要以异步形式调用业务服务、POJO或EJB组件。
- 为了实现异步处理服务，需要集成发布/订阅和点到点的消息机制。
- 需要执行的业务任务在逻辑上由多个业务任务组合而成。

### 解决方案

**用服务激活器接收异步请求，并调用一个或多个业务服务。**

服务激活器用一个JMS监听器来实现，它代表了可以监听、接收JMS消息的服务。如果应用程序使用了EJB组件，并且EJB容器实现了EJB 2.0或以后的版本，就可以用消息驱动bean (message-driven bean, MDB) 来接收异步请求；如果应用程序没有使用EJB技术，或者EJB容器尚不兼容EJB 2.0标准，就必须用Java消息服务 (JMS) 实现自己的定制解决方案。

如果客户端需要异步调用一个业务服务（例如EJB或者POJO服务），它就要创建一条消息，并将其发送给服务激活器，后者将对收到的消息进行分析，以解释客户端的请求。解析出客户端的请求之后，服务激活器找到正确的业务服务组件，调用它来异步处理客户端的请求。496

处理完成之后，客户端可能需要接收处理的结果。为了将处理的结果告知客户端，服务激活器可以向客户端发送一个响应，在其中告诉客户端处理是否成功，并将结果或者指向结果的句柄提供给客户端。如果处理失败，响应中可以包含失败的详细信息，告诉客户端如何从失败中恢复——重新提交请求，或者改正造成失败的问题。服务激活器可能会使用服务定位器来定位服务组件。

客户端提交请求时，可以在请求中包含一个独一无二的请求标识符。处理请求的服务激活器或者业务服务可以将这个标识符附着在结果中，并随响应返回。

图8-8展示了服务激活器的结构。

### 参与者和责任

图8-9展示了服务激活器模式各参与者的交互情况。497

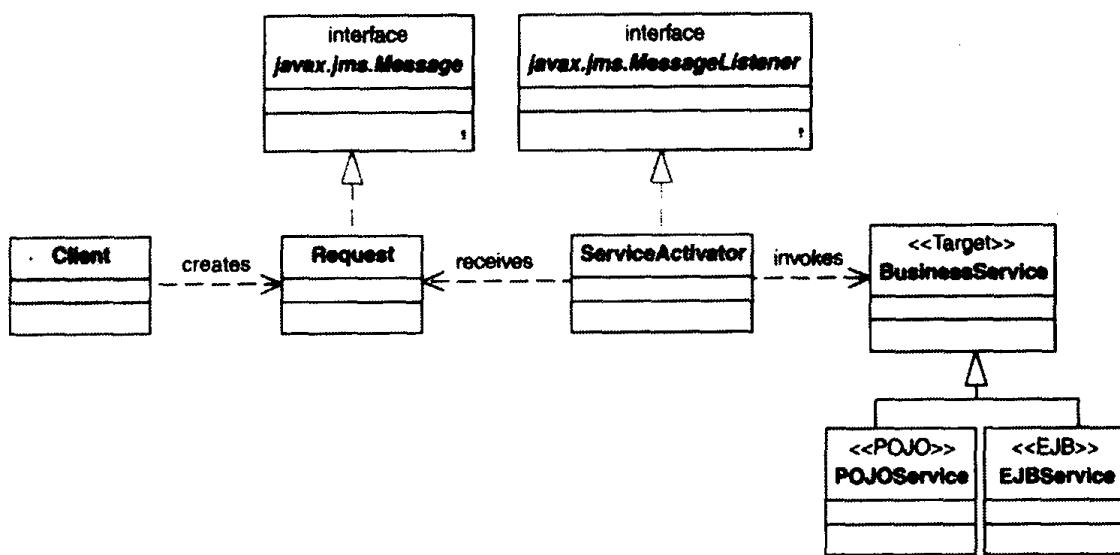


图8-8 服务激活器类图

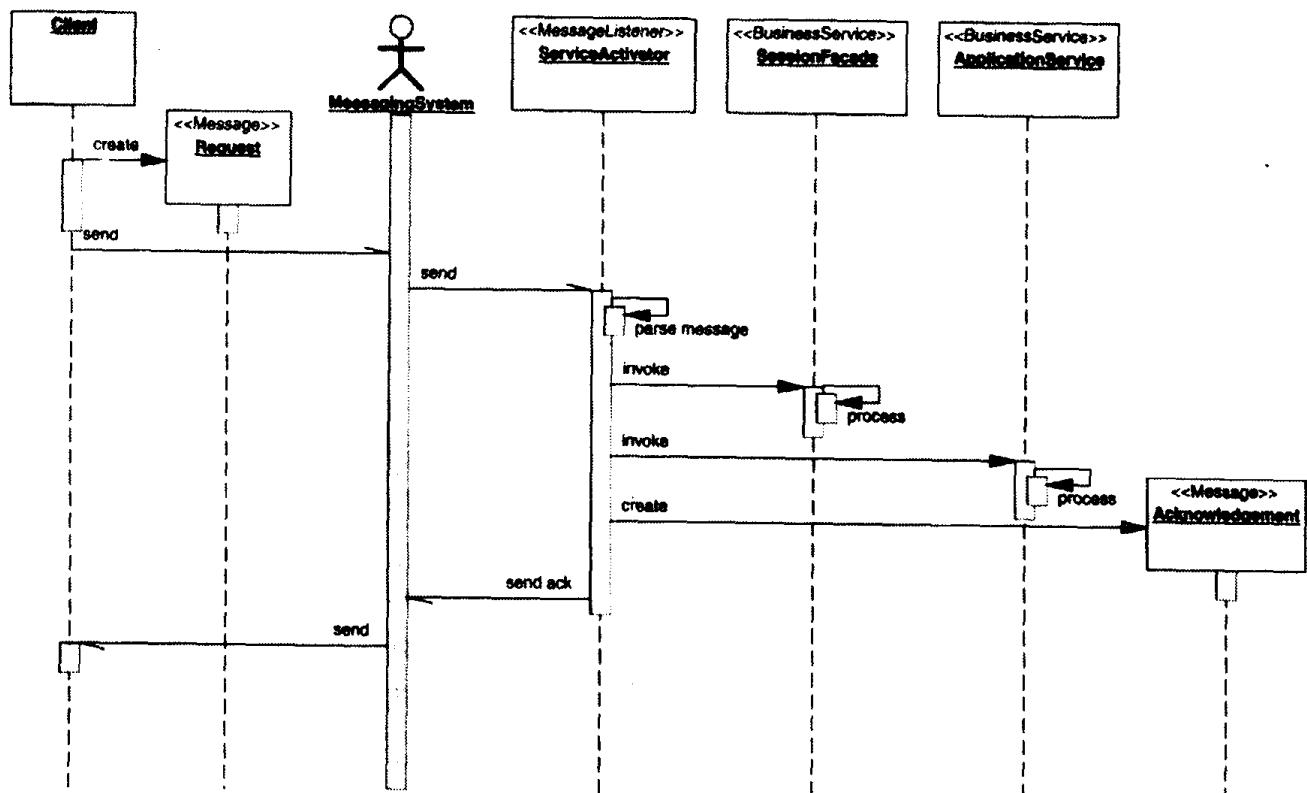


图8-9 服务激活器序列图

### Client (客户端)

498

Client (客户端) 代表了需要以异步方式调用业务方法的客户端，它可以是任何类型的应用程序组件（例如POJO或者EJB组件），惟一的要求是它必须能够创建、发送JMS消息。

如果一个EJB组件需要以异步方式调用业务层的另一个组件，它也可以扮演本模式中的Client角色。

## 设计手记：作为Client（客户端）的遗留系统

在将遗留应用程序与J2EE平台集成时，可以用附加的Java应用程序在遗留系统一端作为消息生成器，两者共同扮演Client（客户端）的角色。遗留系统可能不是用Java编写的，而是与消息中间件集成以发送、接收消息。在Java应用程序中，服务激活器可以接收来自遗留系统的请求消息并异步处理它们。

### Request（请求）

Request（请求）是由Client（客户端）创建的消息对象，Client将使用消息中间件（message-oriented middleware, MOM）将Request发送给ServiceActivator（服务激活器）。根据JMS规范，Request对象必须实现javax.jms.Message接口。JMS API提供了几种消息类型（例如TextMessage、ObjectMessage等），可以根据想要发送的消息类型选择其中一种作为请求对象。

### ServiceActivator（服务激活器）

ServiceActivator（服务激活器）是本模式的主要类，它实现JMS规范中定义的javax.jms.MessageListener接口。ServiceActivator需要实现onMessage()方法，当新的消息到达时，该方法将被调用。ServiceActivator对消息（请求）进行解析（unmarshal），以判断需要做什么事情。ServiceActivator可以使用服务定位器以查找或创建BusinessService（业务服务）组件。

### BusinessService（业务服务）

BusinessService（业务服务）是客户端请求进行异步处理的目标对象，它通常是一个会话界面或者应用服务。如果应用程序的业务逻辑非常简单，也可以用一个业务对象作为BusinessService。BusinessService可以是业务层的任何服务组件，只要它能够处理Client（客户端）的请求即可。

499

## 设计手记：业务服务的实现策略

BusinessService（业务服务）角色可以实现为POJO，也可以实现为EJB组件。

- **业务对象**——如果应用程序非常简单、业务逻辑非常少，业务对象可以直接作为目标组件处理请求。
- **会话界面**——如果应用程序中的业务逻辑涉及了多个业务对象，服务激活器通常不会直接与业务对象交互，而是与封装了业务逻辑的会话界面或者应用服务交互。
- **POJO**——在不使用EJB组件的应用程序中，可以用POJO来实现业务逻辑组件，例如应用服务。在这种情况下，服务激活器可以调用一个或多个POJO的方法，以完成异步处理。

### Response（响应）

Response（响应）是由ServiceActivator（服务激活器）或者BusinessService（业务服务）负责创建并发送的消息对象。Response可能仅仅是对请求的确认，让Client（客户端）知道Request（请求）已经到达目的地；也可能是异步处理的结果。

## 策略

### POJO服务激活器策略

服务激活器最直观的实现策略就是用一个独立的JMS监听器来监听并处理JMS消息，这通常出现在不使用EJB的POJO应用程序中。这种策略可以直接将JMS集成到POJO应用程序中，无须借助J2EE容器的支持。

如果使用了应用服务器，那么可以将服务激活器作为一个服务安装在应用服务器上运行。这可以简化服务激活器的管理，因为可以利用应用服务器提供的功能来监视、（手动或自动地）控制服务激活器的运行情况。

**500** 例8.14展示了[一个POJO 服务激活器的实现](#)。不妨假想一个订单处理应用程序：消费者在线购买商品，订单的履行过程是在后台进行的。有时，订单会被外包给第三方的仓储商店去履行，这时在线商店就需要异步调用后者提供的“履行订单”服务。这个例子可以用于说明如何用点到点（PTP）消息机制来实现异步处理，但使用发布/订阅机制的方式也类似于此，只不过要使用的JMS目标类型是主题而不是队列。

可以在应用服务器中实例化OrderServiceActivator（订单服务激活器）类。可以在一个独立的服务器上运行它，也可以将其注册为Web容器或者EJB容器的启动服务来运行。

### 设计手记：使用哪种JMS目标（Target），主题（Topic）还是队列（Queue）？

如果应用程序需要保证消息（请求）只被发送给一个接收者，就应该使用点到点的消息信道——用JMS队列（Queue）来实现。譬如说，如果订单处理系统必须确保订单请求只被交给一个订单处理程序，并且保证后者只对同一份订单做一次处理，就应该在消息发送者和订单处理程序之间使用一个队列来传递消息。

如果应用程序需要将相同的消息（请求）发送给多个接收者，每个接收者都收到同一份请求的副本，就应该使用发布/订阅的消息信道——用JMS 主题（Topic）来实现。譬如说，在订单处理系统中，如果想要将订单请求抄送给另外几个组件（例如审计服务、日志服务等等），就应该在消息发送者和这些接收者之间使用一个主题来传递消息。每个接收者将收到请求的一份副本，并分别对各自的副本进行处理：订单处理程序负责处理订单；审计服务将请求存入审计系统；日志服务在“已接收订单”的日志中记录该条消息，等等。

*POJO服务激活器策略的类图如图8-10所示。*

OrderDispatcherFacade（订单分配器门面）是一个会话门面，它扮演服务激活器的客户端，将Order（订单）请求发送给OrderServiceActivator（订单服务激活器），如图所示（见例8.15）。当OrderDispatcherFacade的createOrder()方法被Web请求调用时，它首先对请求信息进行检验，并新建一个Order对象，然后调用OrderSenderAppService（订单发送器应用服务）的sendOrder()方法，把Order发送给OrderServiceActivator。收到Order请求之后，OrderServiceActivator会找到并调用后端的“履行订单”服务——OrderProcessorBD和OrderProcessorFacade——对订单进行处理（见例8.16）。

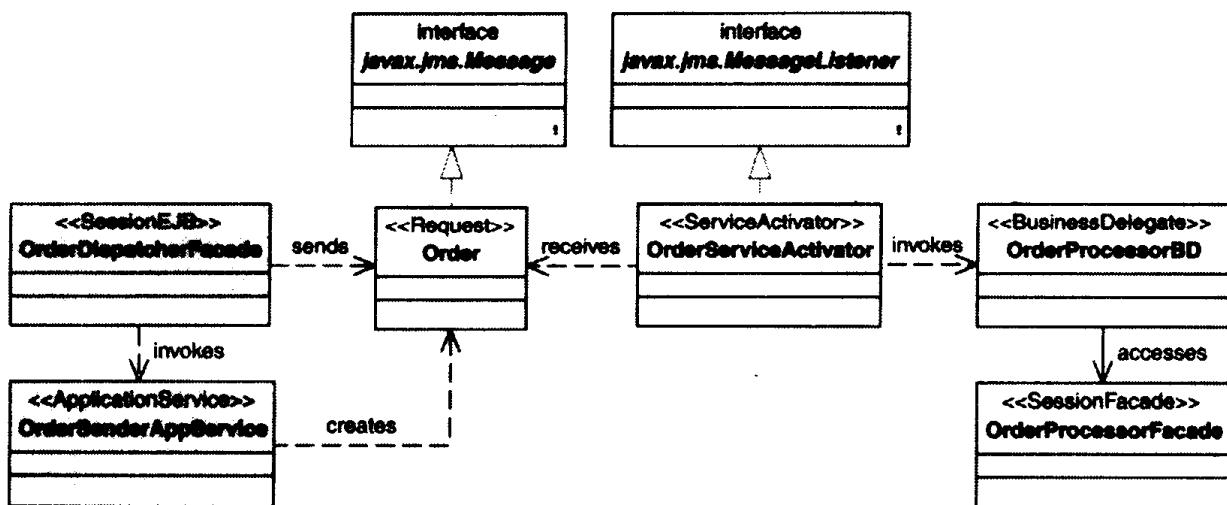


图8-10 POJO服务激活器策略的类图

#### 例8.14 POJO服务激活器：OrderServiceActivator.java

```

1 // imports. . .
2
3 public class OrderServiceActivator implements javax.jms.MessageListener{
4
5     // 队列会话和接受器: 细节参见JMS API。
6     private QueueSession orderQueueSession;
7     private QueueReceiver orderQueueReceiver;
8
9     // 注意: 这个值本该从属性文件或者环境变量里读出。
10    // 只是为了使示例清晰才直接写在这里。
11    private String connectionFactoryName =
12        "PendingOrdersQueueFactory";
13    private String queueName = "PendingOrders";
14
15    // 使用服务定位器来定位由JNDI管理的JMS组件
16    // 比如队列 (Queue) 或队列连接工厂 (Queue Connection factory)
17    private ServiceLocator serviceLocator;
18    private QueueConnectionFactory queueConnectionFactory;
19    private QueueConnection queueConnection;
20
21    public OrderServiceActivator(
22        String connectionFactoryName, String queueName) {
23        super();
24        this.connectionFactoryName = connectionFactoryName;
25        this.queueName = queueName;
26        startListener();
27    }
28
29    private void startListener() {
30        try {

```

```

31     serviceLocator = ServiceLocator.getInstance();
32     queueConnectionFactory =
33     serviceLocator.getQueueConnectionFactory(
34         connectionFactoryName);
35     queueConnection =
36     queueConnectionFactory.createQueueConnection();
37
38     // 以下方法的用法和参数见JMS API
39     orderQueueSession =
40     queueConnection.createQueueSession (...);
41     Queue ordersQueue =
42     serviceLocator.getQueue(queueName);
43     orderQueueReceiver =
44     orderQueueSession.createReceiver(ordersQueue);
45     orderQueueReceiver.setMessageListener(this);
46 }
47 catch (JMSEException excp) {
48     // 处理错误
49 }
50 }
51
52 // JMS API 规定了javax.jms.MessageListener 接口中
53 // 的onMessage 方法。
54 // 当有消息抵达ServiceActivator (服务激活器)
55 // 监听的队列
56 // 的时候，就会调用onMessage() 方法。
57 // 详情参见JMS技术规范和API。
58 public void onMessage(Message message) {
59     try {
60         // 解析消息。关于Message对象，参见JMS API。
61         // 从消息中获取订单对象
62         ObjectMessage objectMessage =
63             (javax.jms.ObjectMessage) message;
64         Order order = (Order) objectMessage.getObject();
65
66         // 使用订单处理器业务门面的业务代表来
67         // 调用订单处理机制
68         OrderProcessorDelegate orderProcessorBD =
69             new OrderProcessorDelegate();
70         orderProcessorBD.processOrder(order);
71
72         // 此处发送响应...
73     }
74     catch (JMSEException jmsexcp) {
75         // 如果有JMSEException异常，在此处处理
76         // 发送错误响应，抛出运行时异常
77     }
78     catch (Exception excp) {

```

```

79         // 处理其他异常
80         // 发送错误响应，抛出运行时异常
81     }
82 }
83 public void close() {
84     try {
85         // 关闭前清理资源
86         orderQueueReceiver.setMessageListener (null);
87         orderQueueSession.close();
88     }
89     catch(Exception excp) {
90         // 异常处理——无法关闭
91     }
92 }
93 }
```

#### 例8.15 服务激活器客户端：OrderDispatcherFacade.java

```

1  // imports. . .
2  public class OrderDispatcherFacade
3      implements javax.ejb.SessionBean {
4
5      // 创建新订单的业务方法
6      public int createOrder(...) throws OrderException {
7
8          // 创建新的业务订单entity bean
9
10
11         // 订单创建成功。发送订单至
12         // 异步后端处理
13         OrderSenderAppService orderSender =
14             new OrderSenderAppService();
15         orderSenderAppService.sendOrder(order);
16
17         // 如果完成，关闭发送器...
18         orderSenderAppService.close();
19
20         // 其他处理
21
22     }
23 }
```

504

可以将与JMS相关的代码放在一个单独的应用服务中以便复用，这个JMS应用服务—OrderSenderAppService—如例8.16所示。

#### 例8.16 Order Sender应用服务：OrderSenderAppService.java

```

1  // imports...
2
3  public class OrderSenderAppService {
```

```

4   // 队列会话和发送器: 详情参见JMS API
5   private QueueSession orderQueueSession;
6   private QueueSender orderQueueSender;
7
8   // 这些值可以从属性文件中读出
9   private String connectionFactoryName =
10   	"PendingOrdersQueueFactory";
11   private String queueName = "PendingOrders";
12
13  // 使用服务定位器来定位由JNDI管理的JMS组件
14  // 比如队列 (Queue) 或
15  // 队列连接工厂 (Queue Connection factory)
16   private JMSServiceLocator serviceLocator;
17
18  // 以下方法初始化并创建队列发送器
19   private void createSender() {
20     try {
21       // 使用ServiceLocator (服务定位器), 获取
22       // 队列连接工厂, 与上面的服务激活器
23       // 代码类似。
24       serviceLocator = ServiceLocator.getInstance();
25       queueConnectionFactory =
26         serviceLocator.getQueueConnectionFactory(
27           connectionFactoryName);
28       queueConnection =
29         queueConnectionFactory.createQueueConnection();
30
31       // 以下方法的用法和参数见JMS API
32       orderQueueSession =
33         queueConnection.createQueueSession(. . .);
34       Queue ordersQueue =
35         serviceLocator.getQueue(queueName);
36       orderQueueSender =
37         orderQueueSession.createSender(ordersQueue);
38     }
39     catch(Exception excp) {
40       // 处理异常——无法创建发送器
41     }
42   }
43
44
45  // 以下方法用于分配订单, 以异步处理形式
46  // 完成服务
47  public void sendOrder(Order order) {
48    try {
49      // 创建一个新的Message (消息), 用于发送Order (订单) 对象
50      ObjectMessage orderObjectMessage =

```

```

51         queueSession.createObjectMessage(order);
52
53     // 按照要求、设置对象型消息的属性
54     // 和投递模式
55     // 参见JMS API中ObjectMessage的部分
56
57     // 把订单置入一个对象型消息中
58     orderObjectMessage.setObject(order);
59
60     // 把消息发送给队列
61     orderQueueSender.send(orderObjectMessage);
62
63     .
64 } catch (Exception e) {
65     // 处理异常
66 }
67     .
68 }
69
70     .
71
72     public void close() {
73         try {
74             // 在关闭前清理资源
75             orderQueueReceiver.setMessageListener(null);
76             orderQueueSession.close();
77         }
78         catch (Exception excp) {
79             // 处理异常——无法关闭
80         }
81     }
82 }
```

506

### MDB服务激活器（消息驱动bean 服务激活器）策略

如果使用EJB 2.0以上的版本，就可以用消息驱动bean（MDB）来实现服务激活器。MDB是一种特殊的无状态session bean，和前面介绍的POJO服务激活器一样，MDB也实现了JMS的MessageListener接口。

在使用MDB时，只需实现监听器的onMessage()方法。把MDB部署到EJB容器中以后，EJB容器会根据MDB部署属性中的声明将其注册到合适的消息队列或主题。由于容器会管理MDB，因此与POJO方案相比，用MDB实现服务激活器需要的工作量更少。所以，如果容器支持MDB，就应该使用MDB服务激活器策略而非POJO服务激活器策略。

例8.17展示了用MDB实现的OrderServiceActivator类。

#### 例8.17 MDB服务激活器, OrderServiceActivatorMDB.java

```

1  public class OrderServiceActivator
2      implements javax.ejb.MessageDrivenBean,
```

```

3     javax.jms.MessageListener {
4     . . .
5
6     // 当与该服务激活器相应的消息到达时,
7     // EJB容器调用onMessage方法
8     public void onMessage(Message message) {
9         try {
10             // 解析消息。关于Message对象, 参见JMS消息API。
11             // 从消息中获取订单对象
12             ObjectMessage objectMessage =
13                 (javax.jms.ObjectMessage) message;
14             Order order = (Order) objectMessage.getObject();
15
16             // 使用订单处理器业务门面的业务代表来
17             // 调用订单处理机制
18             OrderProcessorDelegate orderProcessorBD =
19                 new OrderProcessorDelegate();
20             orderProcessorBD.processOrder(order);
21
22             // 在此处发送响应...
23         }
24         catch (JMSEException jmsexcp) {
25             // 如果有JMSEException异常, 则处理
26             // 发送错误响应, 抛出运行时异常
27         }
28         catch (Exception excp) {
29             // 处理其他异常
30             // 发送错误响应, 抛出运行时异常
31         }
32     }
33
34     // implement other methods required by the
35     // Message Driven Bean
36     public void setMessageDrivenContext(
37         javax.ejb.MessageDrivenContext ctx)
38         throws javax.ejb.EJBException { . . . }
39
40     public void ejbCreate() { . . . }
41
42     public void ejbRemove() { . . . }
43 }

```

### 命令请求策略

在Client(客户端)发送的请求中, 可以包含一个命令(*Command*) [GoF]对象, 或是对位于业务层的命令对象的逻辑引用。服务激活器收到请求之后将解析出其中包含的命令, 创建与之对应的命令实例, 并执行这个命令。

命令请求策略的类图如图8-11所示。

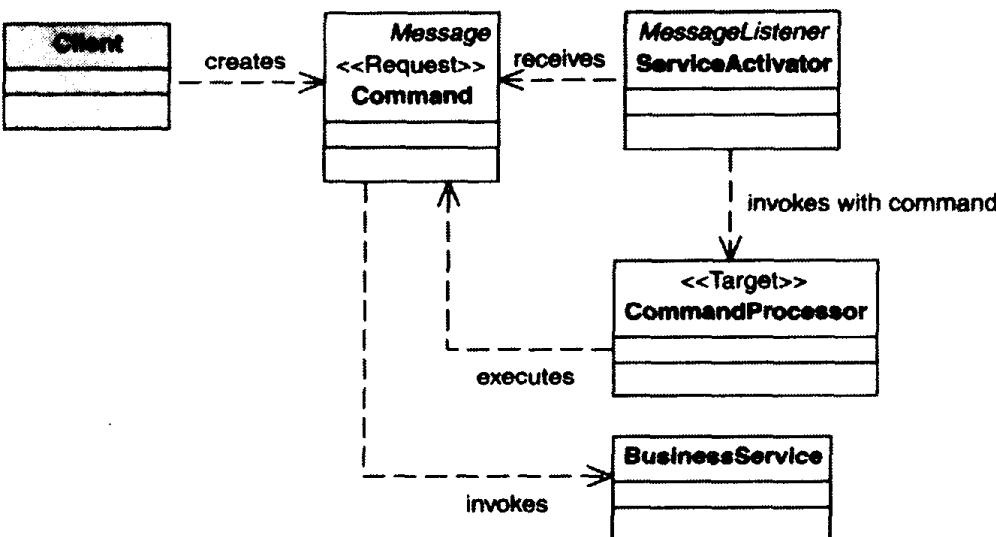


图8-11 命令请求策略的类图

### 服务激活器聚合器策略

有时，一项业务服务在逻辑上是由几个子任务组合而成的，每项子任务都可以异步处理。在这种情况下，客户端可以创建多个任务，然后一次性地将它们提交给服务激活器进行异步处理。不过，只有当这些子任务彼此之间互不依赖并且需要耗费大量资源和时间时，这样的并行处理才有意义。

譬如说，在一个旅游网站上预定旅行计划可能需要做三件事：订机票、订酒店、订出租车。这三件事是可以并行开展的。收到每个请求之后，服务激活器会将该请求交给与之对应的工作对象去处理——后者通常是另一个服务激活器，监听着它自己的消息队列。这种办法的问题在于：客户端必须自己动手将服务激活器中多个异步工作对象返回的结果聚合起来。

另一种选择也是可以让服务激活器来负责处理结果的聚合。如果在客户端代码中无法实现监听和聚合的逻辑，在服务激活器中实现会比较简单，因为后者本来就是一个监听器。使用这种策略时，需要多个服务激活器协同工作，以完成并行处理。参与其中的各个服务激活器扮演不同的角色，使用不同的消息队列。

服务激活器聚合器策略的结构如图8-12所示。

509

- **任务管理器 (task manager)** 服务激活器负责接收来自客户端的最初请求，将整个服务处理分解成几项任务 (task)，并将每项任务发送给能够处理该任务的服务激活器。
- **工作者 (worker)** 服务激活器收到任务请求后，对任务进行处理，并将结果发送给聚合器 (aggregator) 服务激活器，后者将把所有的响应聚合起来，将完整的结果发回给最初发出请求的客户端。由于聚合工作的性质要求，聚合器被实现为一个有状态的服务激活器。

Client（客户端）发送一个Request（请求）对象以及唯一的请求标识符给TaskManager（任务管理器），后者是一个服务激活器，负责将Request分解成多个Task（任务），并将每个Task分发给一个Worker（工作者）。每个被分发出去的Task对象都带有原始的唯一的请求标识符，以及

在同一个Request分解出来的所有Task对象中唯一的任务号。Worker也被实现为服务激活器，它们各自监听不同的任务信道（Queue或者Topic）。

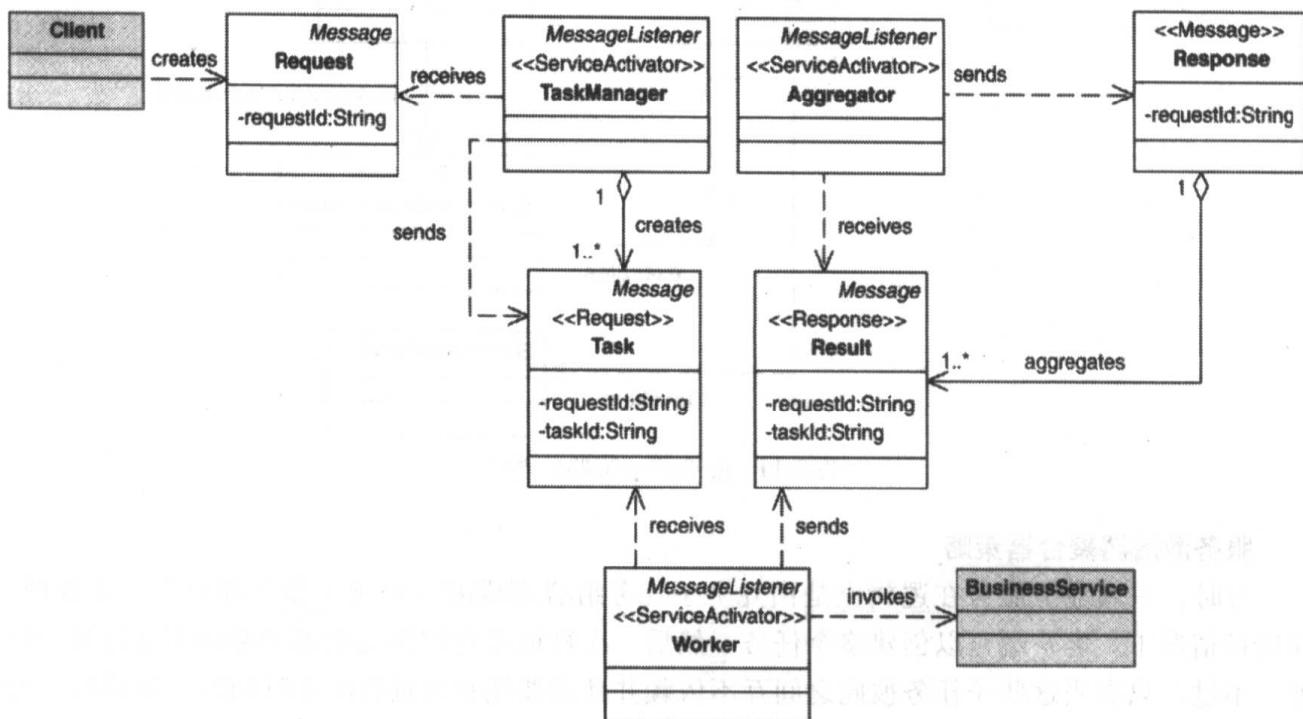


图8-12 服务激活器聚合器策略的类图

接收到Task（任务）对象之后，Worker（工作者）对Task进行异步处理，然后将Result（结果）发送给Aggregator（聚合器），后者根据原始的唯一的请求标识符和一个请求内唯一的任务号将所有的Result响应聚合到同一个Response（响应）对象中，并将其发回给最初发送请求的Client（客户端）。在实现Aggregator时，需要确保它能够处理以任意顺序到达的Result响应消息。也就是说，不论任务2的响应在任务1的响应之前还是之后抵达，Aggregator都应该能够作出正确的行为。

如果Aggregator（聚合器）被实现为一个无状态的对象（例如MDB，属于无状态session bean），它将无法保存聚合过程的中间结果。这样一来，就不得不将中间结果保存在持久化存储介质（例如消息队列或者数据库）中。在这种情况下，Aggregator收到Result（结果）响应消息之后可能做下列事情：

- 检查持久化存储介质，以获取该任务的当前状态。
- 检查新到达的Result是否足够完成聚合过程（如果所有需要的Result都已到达，则聚合过程可以完成）：
  - 如果是，从持久化存储介质取出所有先前存储的Result对象，加上新到达的Result对象，用于创建一个聚合之后的Response对象，并将其发回给客户端。
  - 反之，将新到达的Result对象也存入持久化存储介质，继续等待其他Result到达。

图8-13至图8-15展示了角色之间的交互情况。

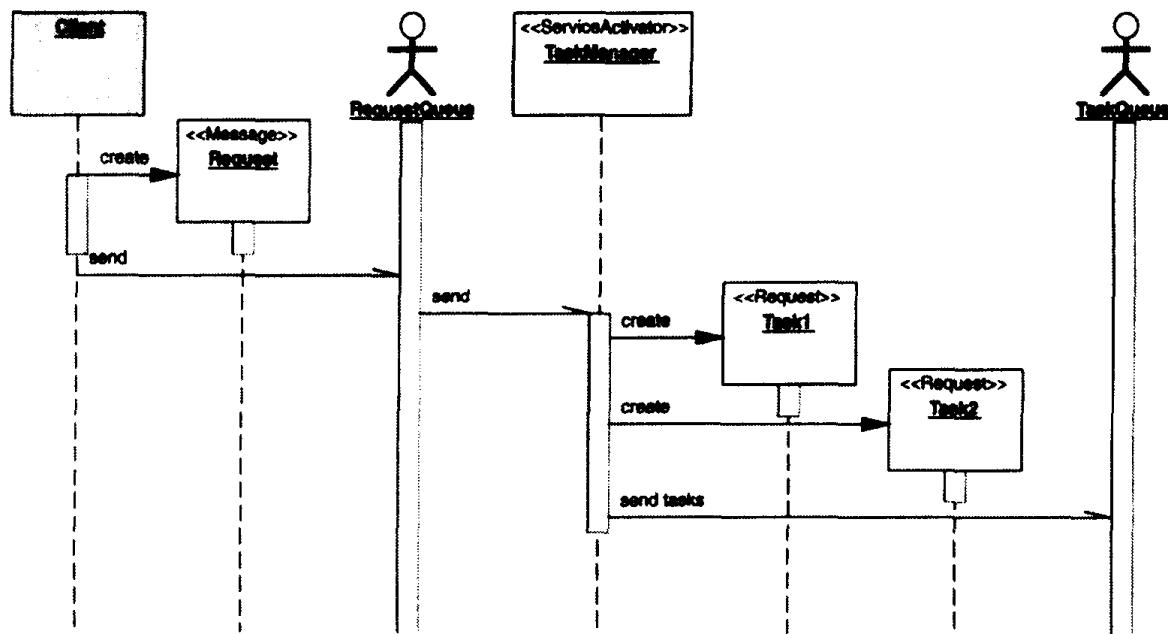


图8-13 服务激活器聚合器策略的序列图（第1部分）

511

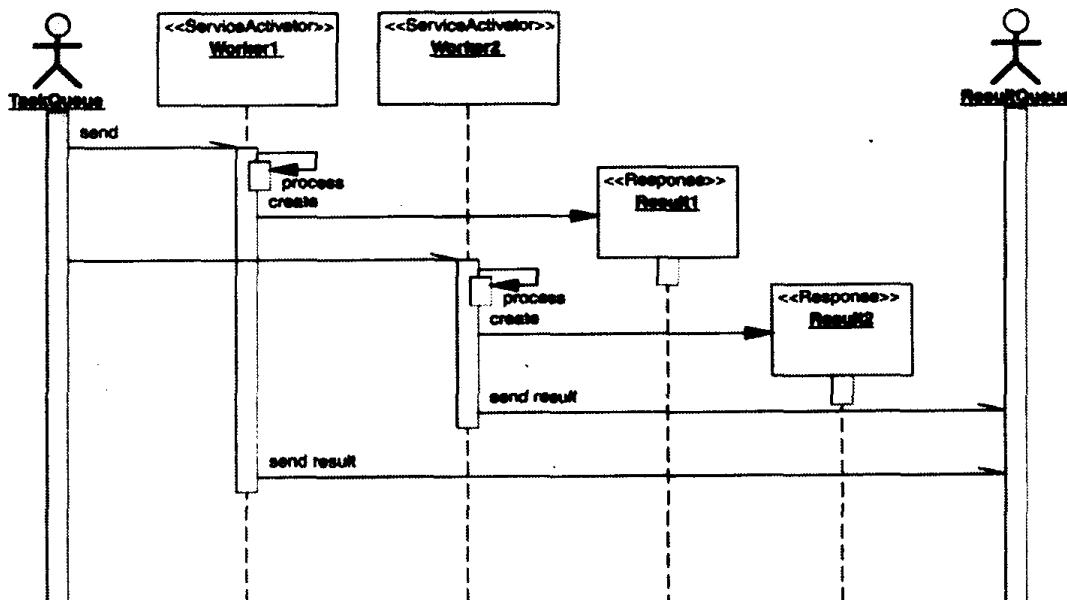


图8-14 服务激活器聚合器策略的序列图（第2部分）

512

## 响应策略

在服务激活器发回响应时，有几种策略可以用于确保响应被交给了最初发出请求的客户端。

### 数据库响应策略

轮询数据库——可以把响应（其中携带了异步处理的结果）保存在数据库中，客户端定时轮询数据库，根据唯一的请求标识符检查结果是否已到达。在大多数情况下，只有当预计“提交请求”与“返回结果”之间的间隔不长时，才适合用轮询的方式。此外，还需要决定客户端在

何时放弃轮询、结果在数据库中保存多长时间、数据库中过期结果的清理策略等等。如果有很  
多客户端频繁地轮询数据库，也会对应用程序的性能造成巨大的影响。

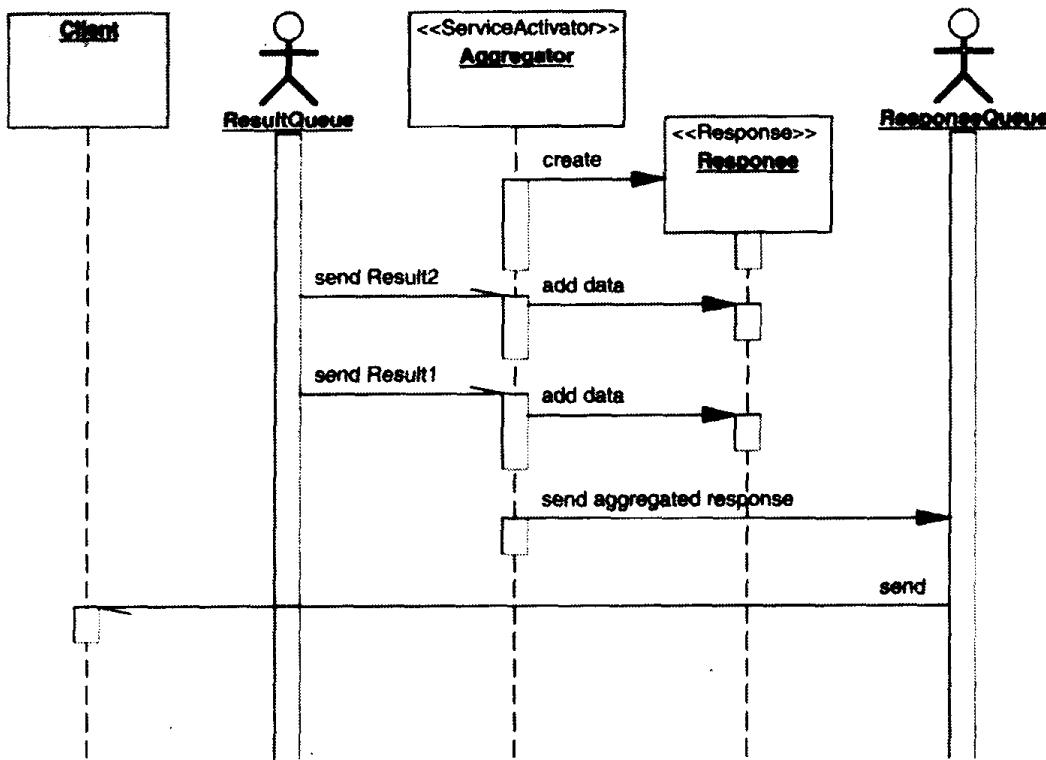


图8-15 服务激活器聚合器策略的序列图（第3部分）

### Email响应策略

发送一封email——在某些应用程序中，可以用一种简单的方式异步地返回结果：将结果保  
存在数据库中（数据库响应策略就是如此）。但是，客户端无须轮询数据库以获得结果，而是在  
处理完成之后，由服务激活器或者业务服务通过email将处理的情况告知客户端。在需要与最终  
用户发生交互的应用程序中，这种方式可能最合适。譬如说，一个基于Web的网上商店就可以  
通过email告知客户端订单的处理情况：订单已接受、订单已发货等等。客户端只在收到确认信  
息之后才去查询数据库，因此这种策略比无条件的定时轮询具有更强的可扩展性。

### JMS消息响应策略

发送一条JMS消息——这种策略和前一种策略很相似。如果客户端是一个Java应用程序或者  
组件，可以考虑将其实现为另一个JMS监听器。服务激活器或者业务服务可以发送一条JMS消息  
作为响应，其中可能仅仅包含一条确认信息，也可能包含结果数据。

513

## 效果

- 将JMS集成到企业应用中

服务激活器使得可以在POJO企业应用系统（使用POJO服务激活器策略）和EJB企业应用系  
统（使用MDB服务激活器策略）中利用JMS。不论应用系统运行在什么平台上，只要拥有JMS

运行时环境，就可以在应用系统中实现服务激活器模式，以提供异步处理的能力。

- 为任何业务层组件提供异步处理能力

服务激活器模式使你能够为任何类型的EJB（包括无状态session bean、有状态session bean和entity bean）提供异步调用功能。服务激活器在客户端和业务服务之间扮演“中间人”的角色，为任何实现业务服务的组件提供异步调用的能力。

- 可以作为独立的JMS监听器运行

*POJO*服务激活器可以作为独立监听器运行，无须任何容器支持。但是，在关键性的应用程序中，需要对服务激活器进行监控，以确保其可用性。可以给应用程序加上额外的管理和维护功能，但这会带来一定的开销。*MDB*服务激活器是由应用服务器来监管的，因此可能是更好的选择。

## 相关模式

- 会话门面

会话门面封装了系统的复杂性，提供了对业务对象的粗粒度访问。服务激活器可以将会话门面看作业务服务来访问，调用后者所提供的业务处理方法。

- 应用服务

服务激活器同样可以把应用服务作为业务服务来访问，调用后者的方法来处理请求。

- 业务代表

服务激活器通常会借助业务代表来访问会话门面。这样做可以简化服务激活器的代码，并使其被多个不同的层次复用。

- 服务定位器

客户端可以使用服务定位器模式来查找、创建与JMS相关的服务对象。服务激活器可以使用服务定位器模式来查找、创建EJB组件。

- 半同步/半异步 (*Half-Sync/Half-Async*) [POSA2]

服务激活器模式与半同步/半异步模式紧密相关，后者描述了这样一个体系结构：划分出分别用于同步处理和异步处理的不同层次，并在两层之间引入作为中介的消息队列层，以降低系统的耦合程度。

- 聚合器 (*Aggregator*) [EIP]

聚合器模式讨论了这样一个问题：将一个请求转化为多个异步任务，然后将各个任务的结果聚合起来。服务激活器聚合器策略的基本概念与之相似。

514

515

## 业务领域存储

### 问题

需要将持久化逻辑从对象模型中分离出去。

很多系统都拥有复杂的对象模型，它们需要复杂的持久化策略。由于EJB 2.x容器管理的持

久化 (CMP) 增加了容器管理关联 (CMR) 的特性, 用CMP作为复杂对象模型的持久化策略具有更高的可行性 (参见第7章的复合实体模式)。但是, 有些开发者不使用entity bean, 甚至直接在web容器中运行应用程序, 他们更希望将持久化逻辑从对象模型中分离出去。

用业务对象模式实现的对象模型可以使用继承, 并且可以拥有多层的依赖关系。“透明持久化”的概念 (业务对象不负责实现特定于持久化的代码) 极富吸引力, 随着对象模型在J2EE应用程序中的地位日益显要, 这个概念也变得愈加流行。如果决定不用entity bean来实现持久对象模型, 就必须选择一种持久化策略。

## 约束

- 需要避免在业务对象中放入持久化细节。
- 不想使用entity bean。
- 应用系统可能需要在web容器上运行。
- 对象模型使用了继承, 或者具有复杂的关联。

## 解决方案

使用业务领域存储模式实现透明于对象模型的持久化。与J2EE提供的容器管理持久化和bean管理持久化 (将持久化代码放在对象模型中) 不同, 业务领域存储的持久化机制是与对象模型分离的。

516

### 设计手记: EJB持久化 vs. 透明持久化

使用entity bean作为持久化机制和使用透明持久化机制, 两者之间有一个关键的区别: entity bean给对象模型增加了限制, 譬如不能继承、需要对EJB编程以支持持久化。透明持久化则保证持久化代码独立于对象模型之外, 因此减少了对后者的局限。

可以用两种方式实现业务领域存储: 自己编写一个持久化框架, 或者使用现成的持久化产品。持久化产品通常会基于Java数据对象 (JDO) 规范或者某种专有的O/R映射解决方案。

在实现业务领域存储之前, 首先必须拥有一个对象模型。很多应用程序并没有一个明确的对象模型, 而是像事务脚本 (*Transaction Script*) 模式[PEAA]所描述的那样直接调用数据源。

---

**注意:** 本模式中的很多角色与Martin Fowler在[PEAA]中介绍的O/R映射模式有关。为了清楚起见, 我们将在角色旁边加注<PatternName>[PEAA]字样, 以指出对应的PEAA模式。

---

在使用J2EE时, 你可以选择的持久化方案包括下列几种:

- 1) 无对象模型, 每次CRUD业务操作直接读/写数据源——事务脚本[PEAA]。
- 2) 使用entity bean CMP。
- 3) 使用entity bean BMP——复合实体。
- 4) 在POJO对象模型中编码实现持久化——活跃记录 (*Active Record*) [PEAA]。
- 5) 将持久化逻辑从POJO对象模型中分离出去。

在方案3 (entity bean BMP) 和方案4 (在对象模型中实现持久化) 中, 持久化代码是混杂在对象模型中的。在方案5 (分离持久化逻辑和对象模型) 中, 持久化机制是与对象模型分离的。业务领域存储模式就是方案5的一种实现。

## 结构

图8-16展示了业务领域存储模式中核心类的类图。

517

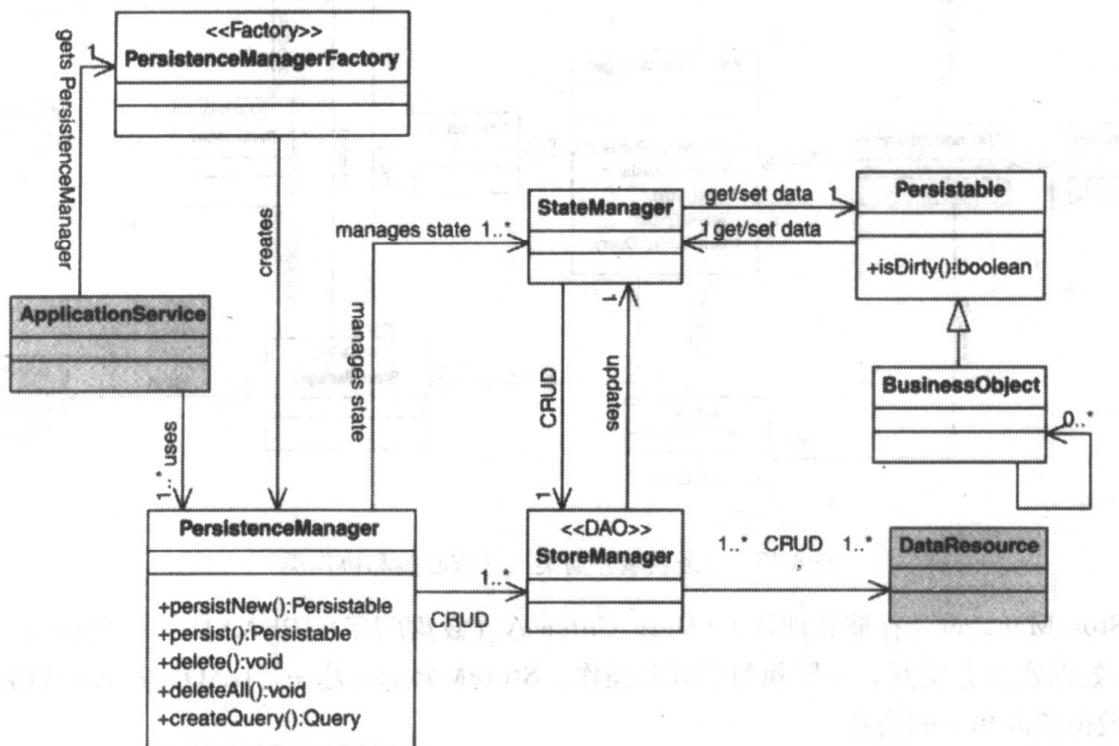


图8-16 业务领域存储类图 (核心类)

图8-17也是业务领域存储模式的类图, 不过将协作类也加入其中。

518

## 参与者和责任

图8-16展示了业务领域存储模式核心类的类图, 下面列出这些核心类。

- **ApplicationService** (应用服务) —— 应用服务对象, 需要与可持久化的业务对象交互。
- **Persistable** (可持久化接口) —— 所有需要被持久化的业务对象必须实现的接口或基类。
- **PersistenceManagerFactory** (持久化管理器工厂) —— 负责创建、管理**PersistenceManager** (持久化管理器) 对象。
- **PersistenceManager** (持久化管理器) (工作单元 (*Unit of Work*) [PEAA]) —— 管理对象模型的持久化和查询。**PersistenceManager**不和业务对象直接交互, 而是和**StateManager** (状态管理器) 打交道, 在需要时命令后者更新对象的状态。
- **StateManager** (状态管理器) (数据映射器 (*Data Mapper*) [PEAA]) —— 与**Persistable** (可持久化) 对象一道管理后者的状态。**StateManager**负责执行事务性存取操作, 从**Data**

resource（数据资源）那里存取Persistable对象的状态。

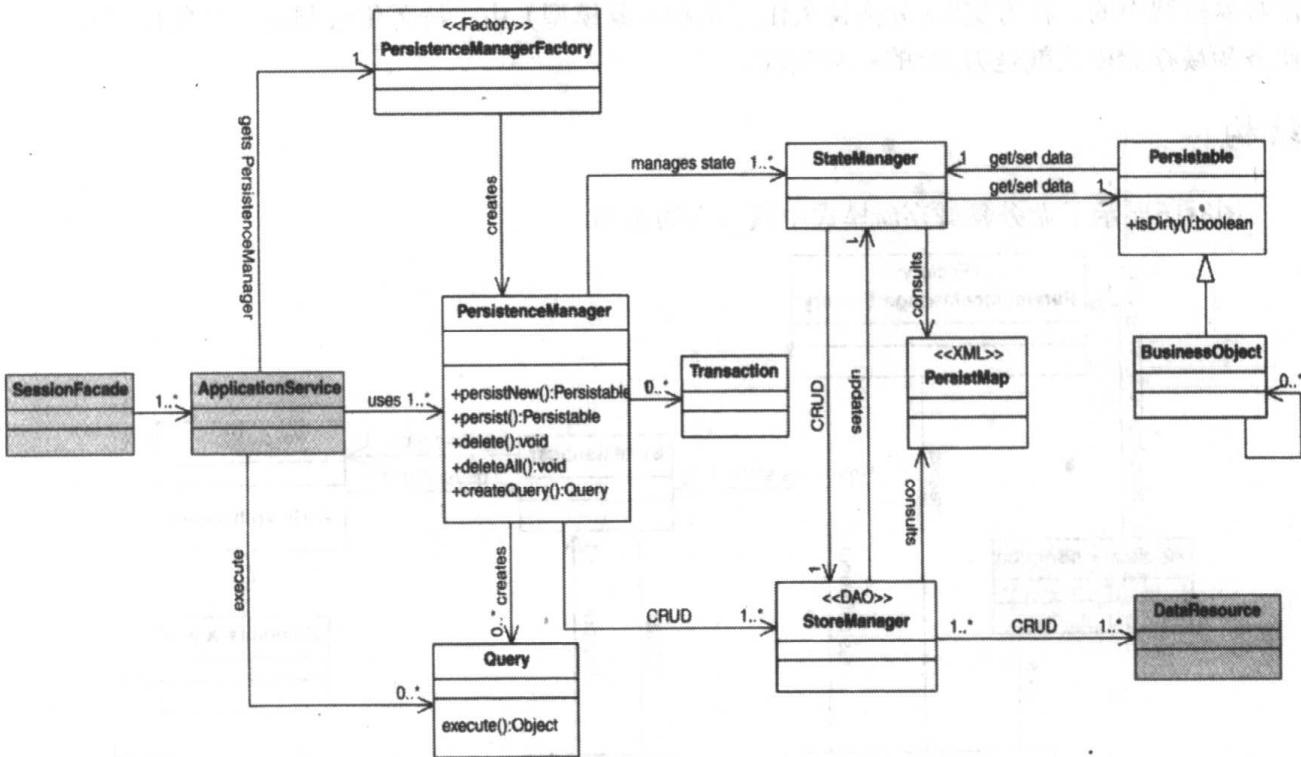


图8-17 业务领域存储类图（核心类和协作类）

- **StoreManager**（存储管理器）（*Data Gateway*（数据门径）[PEAA]）——与Data resource（数据资源）交互，实际执行CRUD操作。StoreManager是一个DAO，在其中封装所有与数据资源相关的信息。
- **Data resource**（数据资源）——可以是任何用于管理数据的资源服务，例如关系型数据库、面向对象数据库、EIS等等。

图8-17展示了加入协作类之后的业务领域存储模式类图。在实现业务领域存储模式时，这些协作类通常会同时出现，但它们不是本模式的核心。

- **SessionFacade**（会话门面）——服务层的入口点，与一个或多个ApplicationService（应用服务）交互。
- **PersistMap**（持久化表）——保存对象之间的关联信息，负责处理可持久对象与数据资源之间的映射关系。
- **Transaction**（事务）——PersistenceManager（持久化管理器）产生的工作，不是一个独立的组件。Transaction对象用于设置面向事务的策略，以及在非受控环境下划分事务。
- **Query**（查询）——封装一个查询，通常包含要查询的实例范围、过滤条件、排序条件、参数声明等信息。

图8-18展示了创建并持久化业务对象时的交互图。

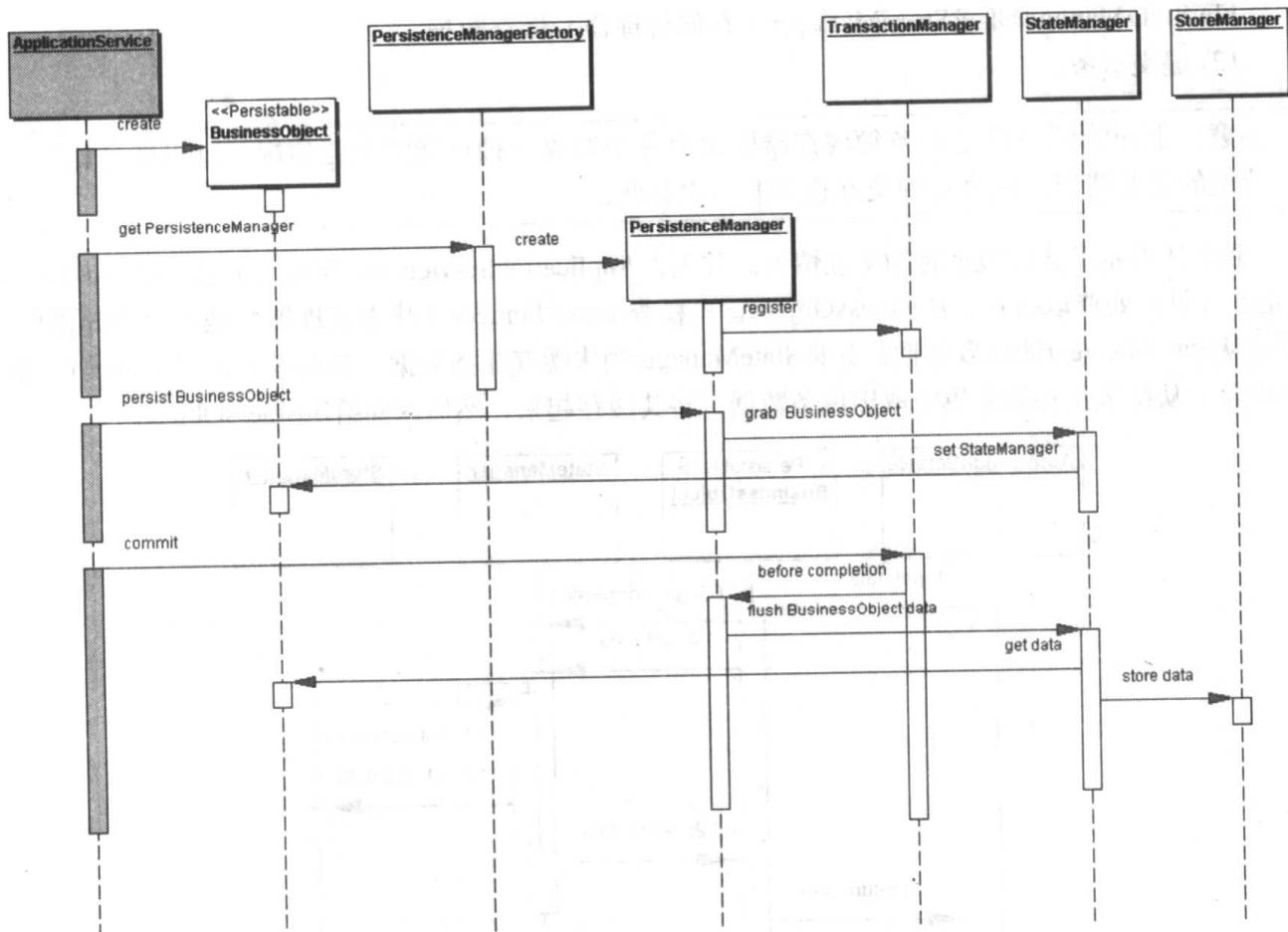


图8-18 创建并持久化业务对象时的交互图

521

要进行的操作依次是：

### 对象创建和持久化

- 1) ApplicationService（业务服务）创建BusinessObject（业务对象）。
- 2) ApplicationService从PersistenceManagerFactory（持久化管理器工厂）处获得PersistenceManager（持久化管理器）。
- 3) PersistenceManager向TransactionManager（事务管理器）注册自己。
- 4) ApplicationService要求PersistenceManager持久化BusinessObject。
- 5) PersistenceManager创建StateManager（状态管理器），并要求后者控制BusinessObject。
- 6) StateManager告知BusinessObject，它是后者的State Manager。
- 7) ApplicationService要求PersistenceManager（隐式或显式地）提交事务。
- 8) TransactionManager（如果是EJB容器，该对象由EJB容器负责初始化）要求PersistenceManager刷新所有处在事务中的StateManager。
- 9) PersistenceManager要求StateManager刷新数据。
- 10) StateManager从BusinessObject获得数据。

11) StateManager要求StoreManager（存储管理器）保存数据。

12) 提交事务。

**注意：**上面的例子只是业务领域存储模式中各个对象一种可能的交互情况。这可能是比较典型的交互情况，但真实的交互也并非一定如此。

图8-19展示了获取数据时的交互情况。首先，ApplicationService（应用服务）从BusinessObject（业务对象）处获取数据。BusinessObject首先检查StateManager（状态管理器）是否允许此操作，然后从StateManager处获取数据。如果StateManager尚未缓存此条数据，就要求StoreManager（存储管理器）从持久化存储介质中取出该条数据，将其缓存起来，然后返回给BusinessObject。

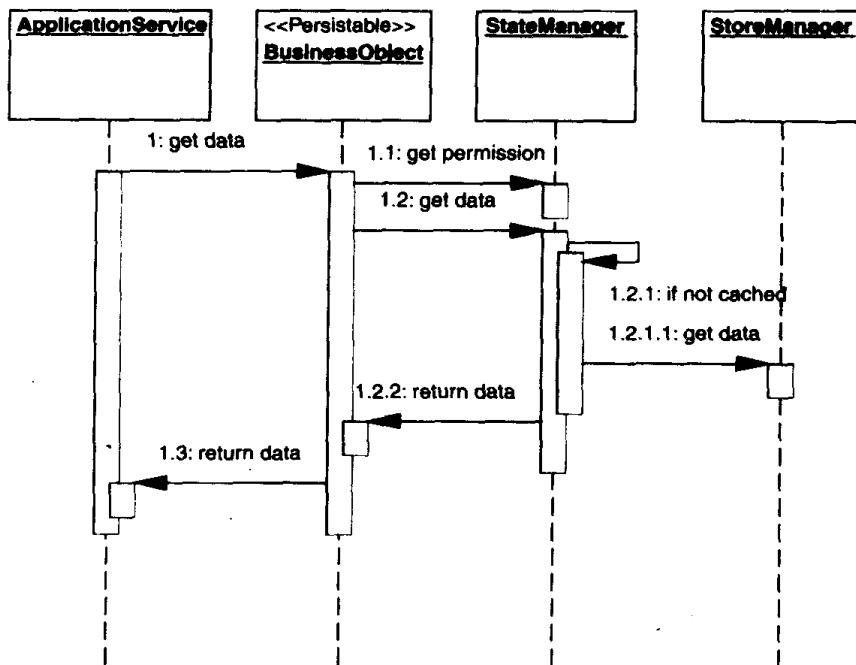


图8-19 获取数据时的交互图

### 设计手记：StateManager（状态管理器）

StateManager可以使用标示表标识符表（Identity Map）模式[PEAA]，将所有已经加载的对象保存在一个Map中，以确保每个对象都只被加载一次。另一种选择是由PersistenceManager（持久化管理器）而不是StateManager来维护一份列表，在其中记录哪些对象已经被缓存了。

图8-20展示了创建、执行一个查询时的序列图。首先，ApplicationService（应用服务）要求PersistenceManager（持久化管理器）创建一个Query（查询）对象并运行它。随后，Query会要求StoreManager（存储管理器）实际执行查询。StoreManager执行查询后，把结果数据返回给Query。这时，PersistenceManager将创建一个StateManager对象和一个BusinessObject（业务对象）对象，再遍历结果集中的每一行，用结果集携带的数据填充刚创建的两个对象。最后，把BusinessObject返回给ApplicationService。

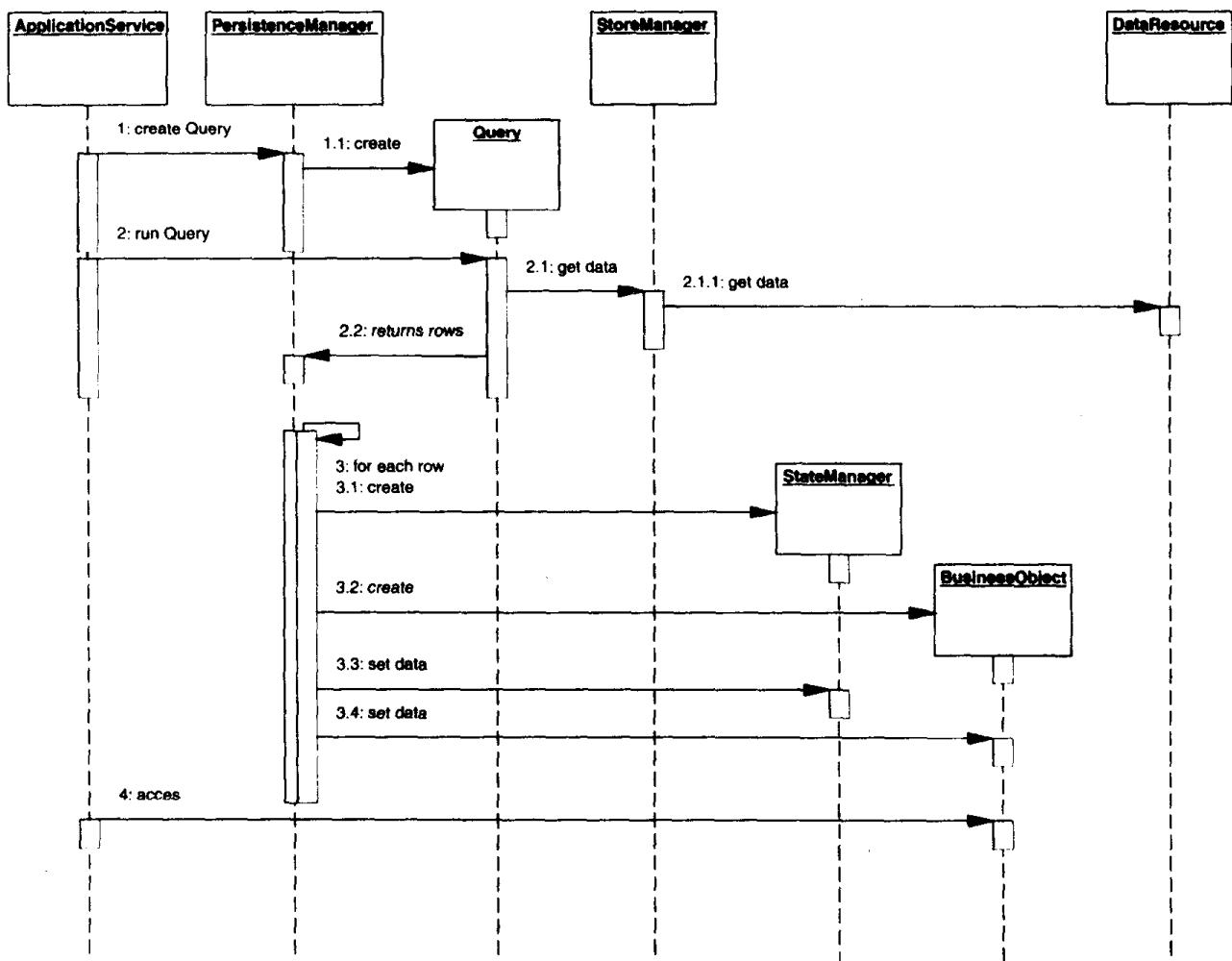


图8-20 创建并执行查询时的序列图

### 设计手记：与Query（查询）对象的交互

前面的例子只是业务领域存储各组件在创建、执行Query（查询）对象时一种可能的交互方式。下面还有两种可能的变化：

- BusinessObject（业务对象）的填充不仅可以由PersistenceManager（持久化管理器）负责，也可以由StoreManager（状态管理器）负责。
- PersistenceManager可以维护Query对象的缓存机制。在重复执行同一个查询时，PersistenceManager就可以直接使用以前缓存下来的Query对象，而不必每次都新建一个。

### 设计手记：持久化增强器

有些工具（例如基于JDO的工具）会对类的源代码或字节码进行增强。不管增强的是源代码还是字节码，我们都把它称为“类增强”。可以用一个增强器对需要持久化的类进行增强，使它们满足StateManager（状态管理器）所要求的契约。正如图8-22所示，如果可持久化的业务对象

需要更新自己的状态，它应该主动调用StateManager。

为了实现这一效果，每当ApplicationService（应用服务）或其他任何客户端访问BusinessObject（业务对象）的数据时，后者必须告知StateManager。成熟的增强器（例如JDO增强器）会对字节码中的getfield/putfield指令进行替换。这不会妨碍使用getXxx/setXxx方法访问字段的模式，这些方法最终将生成getfield/putfield字节码，生成之后的字节码才是增强器处理的目标。

如果使用JDO产品，字节码增强器可以使得在不修改代码的同时提供对持久化的支持。但是，如果要自己搭建一个持久化框架，就必须自己编写增强器来对源代码或者字节码进行增强。

## 策略

### 定制持久化策略

图8-21展示了定制持久化策略的示例类图，图中包含了下列一些类：

- EmployeeApplicationService（员工应用服务）——管理业务操作，协调Employee（员工）业务对象的持久化逻辑。
- PersistenceManagerFactory（持久化管理器工厂）——创建PersistenceManager（持久化管理器），供EmployeeApplicationService使用。
- PersistenceManager——管理Employee的持久化，并负责StateManager（状态管理器）的创建和交流。
- TransactionManager（事务管理器）——在本例中，TransactionManager完全只是一个摆设。如果使用JTS（Java事务服务）提供的功能，就必须将TransactionManager挂接到JTS上。JTS负责通知PersistenceManager提交它所管理的所有数据。
- Transaction（事务）——将系统事务机制（譬如JTS）包装起来。
- StateManager——EmployeeStateManager（员工事务管理器）的基接口。
- EmployeeStateManager——管理Employee数据的进出，与EmployeeStoreManager（员工存储管理器）一道协调CRUD操作。
- EmployeeTO（员工传输对象）——EmployeeStateManager和EmployeeStoreManager之间交换数据所使用的传输对象。
- Persistable（可持久化接口）——需要被持久化的业务对象必须继承的基接口。
- Employee——本例中主要的业务对象。
- EmployeeStoreManager——这是一个数据访问对象，针对特定数据资源实现了Employee的CRUD操作。

526

图8-22展示了创建一个Employee（员工）业务对象并将其持久化的过程，具体的交互情况如下：

- EmployeeApplicationService（员工应用服务）创建Employee（员工）。
- EmployeeApplicationService向PersistenceManagerFactory（持久化管理器工厂）请求PersistenceManager（持久化管理器）。

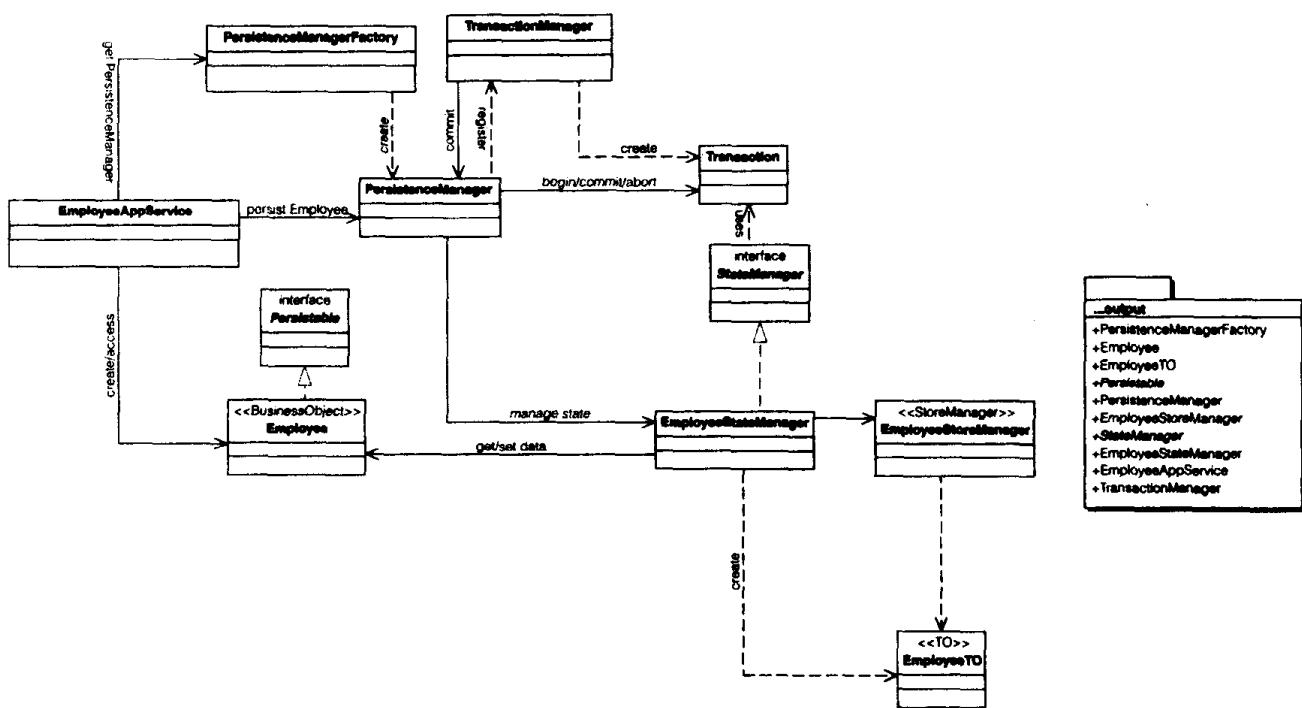


图 8-21 定制持久化策略示例类图

527

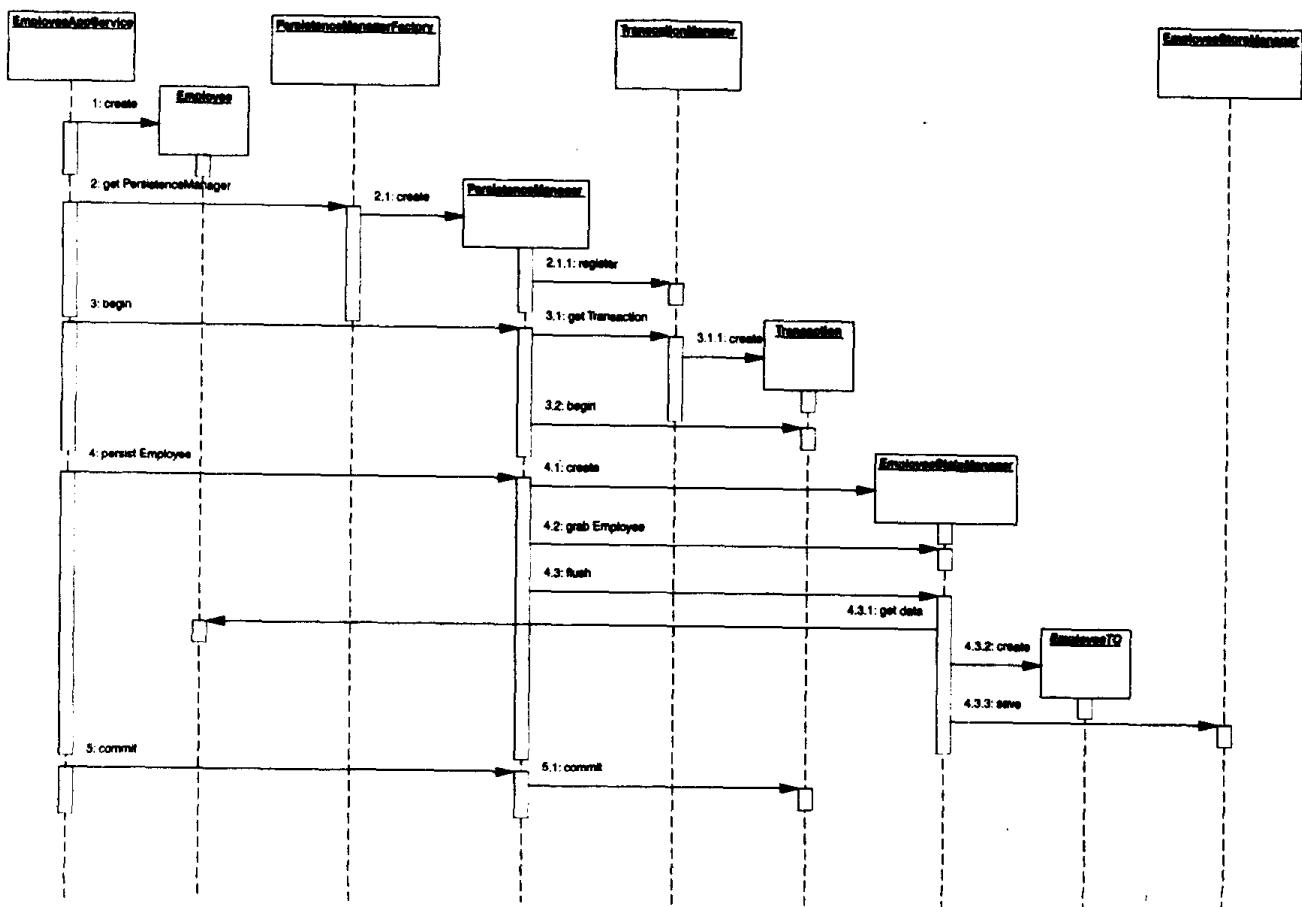


图 8-22 定制持久化策略示例交互图

528

- PersistenceManagerFactory创建PersistenceManager，将其返回给EmployeeApplicationService。
- PersistenceManager向TransactionManager（事务管理器）注册自己。
- EmployeeApplicationService要求PersistenceManager开启事务。
- PersistenceManager创建一个Transaction（事务），并调用后者的“开启”方法。
- EmployeeApplicationService要求PersistenceManager持久化Employee。
- PersistenceManager创建EmployeeStateManager（员工状态管理器），要求后者控制Employee对象，并更新其中的数据。
- EmployeeStateManager从Employee中获得数据，用这些数据创建EmployeeTO（员工传输对象）传输对象（方便起见，我们在这里使用了传输对象模式）。
- EmployeeStateManager将EmployeeTO发送给EmployeeStoreManager（员工存储管理器），以保存数据。
- EmployeeApplicationService要求PersistenceManager提交事务。
- PersistenceManager调用Transaction的“提交”方法，将数据提交到数据源。

### 设计手记：StateManager（状态管理器）代码生成

定制持久化策略中最重要的组成元素就是状态管理器（参见例8.21）。在本例中，EmployeeStateManager（员工状态管理器）和Employee（员工）业务对象之间存在紧密的耦合：由于要负责管理Employee的状态，因此EmployeeStateManager必须对Employee对象所拥有的数据有详尽的了解。真正实现定制持久化策略时，可以根据业务对象类自动生成State Manager。

529

下列代码展示了定制持久化策略的范例实现（见例8.18至例8.25）。

#### 例8.18 EmployeeApplicationService（员工应用服务）代码

```

1 import javax.transaction.*;
2
3 public class EmployeeApplicationService {
4     public String createEmployee(String lastName,
5         String firstName, String ss, float salary,
6         String jobClassification, String geography) {
7     String id = null;
8     String divisionId = null;
9
10    // 创建一个新ID
11    divisionId =
12        getDivisionId(jobClassification, geography);
13
14    // 创建新Employee（员工）
15    Employee e = new Employee(
16        id, lastName, firstName, ss, salary, divisionId);
17
18    PersistenceManagerFactory factory =

```

```

19         PersistenceManagerFactory.getInstance();
20         PersistenceManager manager =
21             factory.getPersistenceManager();
22         try {
23             manager.begin();
24             e = (Employee) manager.persistNew(e);
25             manager.commit();
26         } catch (SystemException e1) {
27         } catch (NotSupportedException e1) {
28         } catch (HeuristicRollbackException e1) {
29         } catch (RollbackException e1) {
30         } catch (HeuristicMixedException e1) {
31     }
32     return id;
33 }
34
35 public void setEmployeeSalary(String id, float salary) {
36     PersistenceManagerFactory factory =
37         PersistenceManagerFactory.getInstance();
38     PersistenceManager manager =
39         factory.getPersistenceManager();
40     Employee e = manager.getEmployee(id);
41     if (e != null) {
42         e.setSalary(salary);
43     }
44     try {
45         manager.begin();
46         e = (Employee) manager.persist(e);
47         manager.commit();
48     } catch (SystemException e1) {
49     } catch (NotSupportedException e1) {
50     } catch (HeuristicRollbackException e1) {
51     } catch (RollbackException e1) {
52     } catch (HeuristicMixedException e1) {
53 }
54 }
55 }

```

530

### 例8.19 Persistable（可持久化）接口和Employee（员工）类

```

1  public interface Persistable {
2  }
3
4  public class Employee implements Persistable {
5      protected String id;
6      protected String firstName;
7      protected String lastName;
8      protected String ss;
9      protected float salary;

```

```

10     protected String divisionId;
11
12     public Employee(String id) {
13         this.id = id;
14     }
15
16     public Employee(String id, String lastName,
17                     String firstName, String ss, float salary,
18                     String divisionId) {
19         this.firstName = firstName;
20         this.lastName = lastName;
21         this.firstName = firstName;
22         this.ss = ss;
23         this.salary = salary;
24         this.divisionId = divisionId;
25     }
26
27     public void setId(String id) {
28         this.id = id;
29     }
30
31     public void setFirstName(String firstName) {
32         this.firstName = firstName;
33     }
34
35     public void setLastName(String lastName) {
36         this.lastName = lastName;
37     }
38
39     public void setSalary(float salary) {
40         this.salary = salary;
41     }
42
43     public void setDivisionId(String divisionId) {
44         this.divisionId = divisionId;
45     }
46
47     public void setSS(String ss) {
48         this.ss = ss;
49     }
50
51     . . .
52 }
```

#### 例8.20 EmployeeStateManager (员工状态管理器) 类

```

1     public class EmployeeStateManager implements StateManager {
2         private final int ROW_LEVEL_CACHING = 1;
3         private final int FIELD_LEVEL_CACHING = 2;
```

```
4
5     int cachingType = ROW_LEVEL_CACHING;
6     boolean isNew;
7     private Employee employee;
8     private PersistenceManager pm;
9
10    public EmployeeStateManager(PersistenceManager pm,
11        Employee employee, boolean isNew) {
12        this.pm = pm;
13        this.employee = employee;
14        this.isNew = isNew;
15    }
16
17    public void flush() {
18        if (pm.isDirty(employee)) {
19            EmployeeTO to =
20                new EmployeeTO(employee.id, employee.lastName,
21                    employee.firstName, employee.ss,
22                    employee.salary, employee.divisionId);
23
24            EmployeeStoreManager storeManager =
25                new EmployeeStoreManager();
26            if (isNew) {
27                storeManager.storeNew(to);
28                isNew = false;
29            } else {
30                storeManager.update(to);
31            }
32            pm.resetDirty(employee);
33        }
34    }
35
36    public void load() {
37        EmployeeStoreManager storeManager =
38            new EmployeeStoreManager();
39        EmployeeTO to = storeManager.load(employee.id);
40        updateEmployee(to);
41    }
42
43    public void load(int field) {
44        if (fieldNeedsReloading(field)) {
45            EmployeeStoreManager storeManager =
46                new EmployeeStoreManager();
47            if (cachingType == FIELD_LEVEL_CACHING) {
48                //Object o =
49                //    storeManager.loadField(employee.id, field);
50                //    updateEmployee( field, o );
51            }
52        }
53    }
54
55    private void updateEmployee(EmployeeTO to) {
56        employee.id = to.id;
57        employee.lastName = to.lastName;
58        employee.firstName = to.firstName;
59        employee.ss = to.ss;
60        employee.salary = to.salary;
61        employee.divisionId = to.divisionId;
62    }
63
64    private boolean fieldNeedsReloading(int field) {
65        return (field & (1 << 1)) != 0;
66    }
67
68    private void resetDirty(Employee employee) {
69        this.employee = employee;
70    }
71
72    private void resetDirty() {
73        this.employee = null;
74    }
75
76    private void resetDirty(boolean isNew) {
77        this.isNew = isNew;
78    }
79
80    private void resetDirty(boolean isNew, Employee employee) {
81        this.isNew = isNew;
82        this.employee = employee;
83    }
84
85    private void resetDirty(boolean isNew, EmployeeTO to) {
86        this.isNew = isNew;
87        this.employee = to.toEmployee();
88    }
89
90    private void resetDirty(boolean isNew, EmployeeTO to, Employee employee) {
91        this.isNew = isNew;
92        this.employee = employee;
93    }
94
95    private void resetDirty(boolean isNew, EmployeeTO to, Employee employee, PersistenceManager pm) {
96        this.isNew = isNew;
97        this.employee = employee;
98        this.pm = pm;
99    }
100 }
```

532

```

51         } else {
52             EmployeeTO to = storeManager.load(employee.id);
53             updateEmployee(to);
54         }
55     }
56 }
57
58 private boolean fieldNeedsReloading(int field) {
59     // 此处使用缓存和数据的有效性规则
60     // 数据可以按字段级别缓存，也可以按行级别缓存
61
62     switch (field) {
63         case EmployeeStateDelegate.LAST_NAME:
64             if (employee.lastName == null)
65                 return true;
66             break;
67         case EmployeeStateDelegate.FIRST_NAME:
68             if (employee.firstName == null)
69                 return true;
70             break;
71         case EmployeeStateDelegate.DIVISION_ID:
72             String did = employee.divisionId;
73             if (did == null || did.indexOf("99-") == -1)
74                 return true;
75             break;
76         case EmployeeStateDelegate.SS:
77             if (employee.ss == null)
78                 return true;
79             break;
80         case EmployeeStateDelegate.SALARY:
81             if (employee.salary == 0.0)
82                 return true;
83             break;
84     }
85     return false;
86 }
87
88 private void updateEmployee(EmployeeTO to) {
89     employee.id = to.id;
90     employee.lastName = to.lastName;
91     employee.firstName = to.firstName;
92     employee.ss = to.ss;
93     employee.salary = to.salary;
94     employee.divisionId = to.divisionId;
95     isNew = false;
96 }
97
98 public boolean needsLoading() {

```

```

99     if (pm.needLoading(employee))
100        return true;
101    else
102      return false;
103  }
104 }
```

534

### 例8.21 EmployeeTO（员工传输对象）类和EmployeeStoreManager（员工存储管理器）类

```

1  public class EmployeeTO {
2    public String id;
3    public String lastName;
4    public String firstName;
5    public String ss;
6    public float salary;
7    public String divisionId;
8
9    public EmployeeTO(String id, String lastName,
10                      String firstName, String ss, float salary,
11                      String divisionId) {
12      this.id = id;
13      this.lastName = lastName;
14      this.firstName = firstName;
15      this.ss = ss;
16      this.salary = salary;
17      this.divisionId = divisionId;
18    }
19  }
20
21 public class EmployeeStoreManager {
22   public void storeNew(EmployeeTO to) {
23     String sql = "Insert into Employee( id, last_name, " +
24           " first_name, ss, salary, division_id ) " +
25           " values( ?, ?, ?, ?, ?, ? )";
26   . .
27  }
28
29  public void update(EmployeeTO to) {
30    String sql = "Update Employee set last_name = ?, " +
31          " first_name = ?, salary = ?, " +
32          " division_id = ? where id = ?";
33   . .
34  }
35
36  public void delete(String empId) {
37    String sql = "Delete from Employee where id = ?";
38   . .
39  }
40 }
```

535

```

41     public EmployeeTO load(String empId) {
42         . . .
43     }
44 }
```

### 例8.22 PersistenceManagerFactory (持久化管理器工厂) 类

```

1  public class PersistenceManagerFactory {
2      static private PersistenceManagerFactory me = null;
3      public synchronized static
4          PersistenceManagerFactory getInstance() {
5          if (me == null) {
6              me = new PersistenceManagerFactory();
7          }
8          return me;
9      }
10     private PersistenceManagerFactory() {
11     }
12     public PersistenceManager getPersistenceManager() {
13         return new PersistenceManager();
14     }
15 }
```

### 例8.23 PersistenceManager (持久化管理器) 类

```

1  import javax.transaction.*;
2  import java.util.HashSet;
3  import java.util.Iterator;
4
5  public class PersistenceManager {
6      HashSet stateManagers = new HashSet();
7      TransactionManager tm;
8      Transaction txn;
9
10     public PersistenceManager() {
11         tm = TransactionManager.getInstance();
12         tm.register(this);
13     }
14
15     public Persistable persistNew(Persistable o) {
16         if (o instanceof Employee) {
17             return setupEmployee(
18                 new EmployeeStateDelegate((Employee)o), true);
19         }
20         return o;
21     }
22
23     public Persistable persist(Persistable o) {
24         // 必须是一个EmployeeStateDelegate (员工状态代表)
25         if (o instanceof Employee) {
```

```
26     EmployeeStateDelegate esd =
27         (EmployeeStateDelegate) o;
28     return esd;
29 }
30 return o;
31 }
32
33 public void commit()
34     throws SystemException, NotSupportedException,
35     HeuristicRollbackException, RollbackException,
36     HeuristicMixedException {
37     if (txn == null) {
38         throw new SystemException(
39             "Must call Transaction.begin() before" +
40             " Transaction.commit()");
41     }
42     Iterator i = stateManagers.iterator();
43     while (i.hasNext()) {
44         Object o = i.next();
45         StateManager stateManager = (StateManager) o;
46         stateManager.flush();
47     }
48     txn.commit();
49     txn = null;
50 }
51
52 public void begin()
53     throws SystemException, NotSupportedException {
54     txn = tm.getTransaction();
55     txn.begin();
56 }
57
58 public Employee getEmployee(String employeeId) {
59     EmployeeStateDelegate esd =
60         new EmployeeStateDelegate(employeeId);
61     setupEmployee(esd, false);
62     return esd;
63 }
64
65 private EmployeeStateDelegate setupEmployee(
66     EmployeeStateDelegate esd, boolean isNew) {
67     EmployeeStateManager stateManager =
68         new EmployeeStateManager(this, esd, isNew);
69     stateManagers.add(stateManager);
70     esd.setStateManager(stateManager);
71     return esd;
72 }
```

537

```

73
74     public void setDirty(Persistable o) {
75         // 设置脏数据标示器为true
76     }
77
78     public void resetDirty(Persistable o) {
79         // 重置脏数据标示器为false
80     }
81
82     public boolean isDirty(Persistable o) {
83         // 检查对象是否“脏”了(即被修改过了)
84         return true;
85     }
86
87     public boolean needLoading(Persistable o) {
88         // 察看是否需要装载
89         return true;
90     }
91 }
```

#### 例8.24 TransactionManager (事务管理器) 类

538

```

1  import javax.transaction.*;
2  import java.util.Iterator;
3  import java.util.LinkedList;
4
5  public class TransactionManager {
6      static TransactionManager me = null;
7
8      private LinkedList persistenceManagers = new LinkedList();
9
10     class PManager {
11         Thread thread;
12         PersistenceManager manager;
13
14         PManager(Thread thread, PersistenceManager manager) {
15             this.thread = thread;
16             this.manager = manager;
17         }
18
19         boolean equals(
20             Thread thread, PersistenceManager manager) {
21             if (this.thread == thread &&
22                 this.manager == manager) {
23                 return true;
24             }
25             return false;
26         }
27     }
```

```

28
29     public synchronized static
30         TransactionManager getInstance() {
31         if (me == null) {
32             me = new TransactionManager();
33         }
34         return me;
35     }
36
37     private TransactionManager() {
38     }
39
40     public Transaction getTransaction() {
41         return new Transaction();
42     }
43
44     public void register(PersistenceManager manager) {
45         . . .
46     }
47
48     public void notifyCommit(Thread t)
49         throws SystemException, HeuristicRollbackException,
50             NotSupportedException, RollbackException,
51                 HeuristicMixedException {
52         Iterator i = persistenceManagers.iterator();
53         while (i.hasNext()) {
54             . . .
55             pm.manager.commit();
56         }
57     }
58 }
```

### 例8.25 Transaction (事务) 类

```

1  import javax.ejb.SessionContext;
2  import javax.naming.InitialContext;
3  import javax.naming.NamingException;
4  import javax.transaction.*;
5
6  public class Transaction {
7      UserTransaction txn;
8
9      public void setSessionContext( SessionContext ctx ) {
10         ctx.getUserTransaction();
11     }
12     public Transaction() {
13         InitialContext ic = null;
14         try {
15             ic = new InitialContext();
```

```

16         txn = (UserTransaction)ic.lookup(
17             "java:comp/UserTransaction");
18     } catch (NamingException e) {
19     }
20 }
21 public void begin()
22     throws SystemException, NotSupportedException {
23     txn.begin();
24 }
25
26 public void commit()
27     throws SystemException, HeuristicRollbackException,
28     RollbackException, HeuristicMixedException {
29     txn.commit();
30 }
31
32 public void rollback() throws SystemException {
33     txn.rollback();
34 }
35 }
```

540

### 状态的随需获取

前面的例子重点关注如何创建并持久化Employee（员工）业务对象。但是，很多时候需要从持久化介质中取出业务对象，然后访问其中的数据。此时，EmployeeStateManager（员工状态管理器）可以选择在创建Employee的同时将数据填入其中，也可以等到确实需要时再从EmployeeStoreManager那里获取数据。

例8.26展示了一种可行的状态缓存方案，用一个状态代表（EmployeeStateDelegate）缓存Employee（员工）的状态。这里最大的区别在于：EmployeeStateDelegate类继承了Employee类，并覆盖了所有的getter方法。当应用服务向PersistenceManager（持久化管理器）请求一个Employee业务对象时，它获得的实际上是一个EmployeeStateDelegate对象——同样是一个Employee对象。在调用getter方法时，EmployeeStateDelegate首先检查Employee对象中是否已经有需要获取的数据，如果没有，再请求EmployeeStateManager（员工状态管理器）将数据加载进来。

### 设计手记：定制持久化策略

可以看到，定制持久化策略非常强大，并且可以很方便地自动生成StateManager（状态管理器）和StateDelegate（状态代表）的代码。但是，随着对象模型的扩充，自动生成的类也会飞快增加，系统的复杂度会快速上升。所以，如果对象模型比较复杂，最好是让JDO风格的产品提供帮助。

#### 例8.26 EmployeeStateDelegate（员工状态代表）类

```

1  public class EmployeeStateDelegate extends Employee {
2      static final int LAST_NAME = 1;
3      static final int FIRST_NAME = 2;
```

```
4     static final int SS = 3;
5     static final int SALARY = 4;
6     static final int DIVISION_ID = 5;
7
8     private EmployeeStateManager stateManager;
9
10    public EmployeeStateDelegate(String id, String lastName,
11        String firstName, String ss, float salary,
12        String divisionId) {
13        super(id, lastName, firstName, ss, salary, divisionId);
14    }
15
16    public EmployeeStateDelegate(Employee e) {
17        super(e.id, e.lastName, e.firstName,
18            e.ss, e.salary, e.divisionId);
19    }
20
21    public EmployeeStateDelegate(String employeeId) {
22        super(employeeId);
23    }
24
25    public EmployeeStateDelegate(String employeeId,
26        EmployeeStateManager stateManager) {
27        super(employeeId);
28        this.stateManager = stateManager;
29    }
30
31    public void setStateManager(
32        EmployeeStateManager stateManager) {
33        this.stateManager = stateManager;
34    }
35
36    public String getFirstName() {
37        stateManager.load(FIRST_NAME);
38        return firstName;
39    }
40
41    public String getDivisionId() {
42        stateManager.load(DIVISION_ID);
43        return divisionId;
44    }
45
46    public String getLastName() {
47        stateManager.load(LAST_NAME);
48        return lastName;
49    }
50
51    public String getSS() {
```

```

52     stateManager.load(ss);
53     return ss;
54 }
55
56     public float getSalary() {
57         stateManager.load(SALARY);
58         return salary;
59     }
60
61     public EmployeeStateManager getStateManager() {
62         return stateManager;
63     }
64 }
```

### JDO策略

例8.27至例8.39展示了一个人力资源（HR）服务的实现，其中业务领域存储模式用JDO（Java数据对象）来实现。

例8.27展示了HRApplcationService（人力资源应用服务）类的源代码。这个类是顶级的应用服务，其中包含了业务对象的创建、持久化等逻辑。这里涉及的业务对象包括Employee（员工）、Department（部门）和Project（项目），其中Employee对象又分为两类：FullTimeEmployee（全职员工）和PartTimeEmployee（兼职员工）。

HRApplcationService（人力资源应用服务）用到了两个助手类：Factory（工厂）和Queries（查询）。HRApplcationService将所有业务对象和查询对象的创建工作委派给Factory（见例8.27和例8.28），FactoryImpl（Factory接口的实现类）负责与PersistenceManager交互，后者就是用JDO实现的。QueriesImpl（Queries接口的实现类）则负责执行查询，它借助Factory来访问JDO查询机制。

可以看到，这个例子明确地体现了业务领域存储模式中的一些角色，包括：

- PersistenceManagerFactory（持久化管理器工厂）——在本例中实现为FactoryImpl类。
  - PersistenceManager（持久化管理器）——基于JDO的PersistenceManager由FactoryImpl类负责创建。
  - Query（查询）——在本例中实现为QueriesImpl类，它把实际的查询工作委派给JDO的Query类。
  - BusinessObject（业务对象）——Employee、Department、Project、PartTimeEmployee和FullTimeEmployee等接口都是业务对象，分别由EmployeeImpl、DepartmentImpl、ProjectImpl、PartTimeEmployeeImpl和FullTimeEmployeeImpl类实现。
  - PersistMap（持久化表）——源码中没有出现，但JDO运行时环境会用到它。
- 另外，JDO运行时环境隐含地实现了业务领域存储模式的其余角色：
- Persistable（可持久化接口）——JDO的代码生成器不需要此接口。
  - Transaction（事务）——本例中没有体现。
  - StateManager（状态管理器）——由JDO代码生成器自动生成，JDO运行时环境会用到它。

- StoreManager（存储管理器）——由JDO代码生成器自动生成，JDO运行时环境会用到它。

### 例8.27 HRApplicationService（人力资源应用服务）类

```

1 // HRApplicationService.java
2
3 package com.corej2eepatterns.service;
4
5 import com.corej2eepatterns.business.hr.*;
6 import java.math.BigDecimal;
7
8 /**
9  * @author Craig Russell
10 */
11 public class HRApplicationService {
12     private Factory factory;
13     private Queries queries;
14
15     /** 创建HRApplicationService 的一个新实例*/
16     public HRApplicationService(Factory factory) {
17         this.factory = factory;
18         this.queries = factory.getQueries();
19     }
20
21     public Employee getEmployee(long id) {
22         return queries.getEmployee(id);
23     }
24
25     public Department getDepartment(String name) {
26         return queries.getDepartment(name);
27     }
28
29     public Project getProject(String name) {
30         return queries.getProject(name);
31     }
32
33     public PartTimeEmployee createPartTimeEmployee(
34             long id, String firstName, String lastName,
35             BigDecimal wage, String departmentName) {
36         PartTimeEmployee employee =
37             factory.createPartTimeEmployee(
38                 id, firstName, lastName);
39         employee.setWage(wage);
40         Department department =
41             queries.getDepartment(departmentName);
42         employee.setDepartment(department);
43         factory.persistObject(employee);
44         return employee;
45     }

```

```

46
47     public FullTimeEmployee createFullTimeEmployee(
48         long id, String firstName, String lastName,
49         BigDecimal salary, String departmentName) {
50         FullTimeEmployee employee =
51             factory.createFullTimeEmployee(
52                 id, firstName, lastName);
53         employee.setSalary(salary);
54         Department department =
55             queries.getDepartment(departmentName);
56         employee.setDepartment(department);
57         factory.persistObject(employee);
58         return employee;
59     }
60 }
```

**例8.28 Factory (工厂) 接口**

```

1 // Factory.java
2
3 package com.corej2eepatterns.business.hr;
4
5 /**
6  * @author Craig Russell
7 */
8 public interface Factory {
9
10    PartTimeEmployee createPartTimeEmployee(
11        long id, String firstName, String lastName);
12
13    FullTimeEmployee createFullTimeEmployee(
14        long id, String firstName, String lastName);
15
16    Department createDepartment(String name);
17    Project createProject(String name);
18    Queries getQueries();
19    void persistObject(Object object);
20    void persistObjects(Object[] objects);
21 }
```

545

**例8.29 FactoryImpl (工厂实现) 类**

```

1 // FactoryImpl.java
2
3 package com.corej2eepatterns.business.impl;
4
5 import com.corej2eepatterns.business.hr.*;
6 import javax.jdo.PersistenceManager;
7
8 /**
```

```
9     * @author Craig Russell
10    */
11    public class FactoryImpl implements Factory {
12        PersistenceManager pm;
13
14        /** 创建FactoryImpl 的一个新实例*/
15        public FactoryImpl(PersistenceManager pm) {
16            this.pm = pm;
17        }
18
19        PersistenceManager getPersistenceManager() {
20            return pm;
21        }
22
23        public Department createDepartment(String name) {
24            return new DepartmentImpl(name);
25        }
26
27        public FullTimeEmployee createFullTimeEmployee(
28            long id, String firstName, String lastName) {
29            return new FullTimeEmployeeImpl(
30                id, firstName, lastName);
31        }
32
33        public PartTimeEmployee createPartTimeEmployee(
34            long id, String firstName, String lastName) {
35            return new PartTimeEmployeeImpl(
36                id, firstName, lastName);
37        }
38
39        public Project createProject(String name) {
40            return new ProjectImpl(name);
41        }
42
43        public Queries getQueries() {
44            return new QueriesImpl(this);
45        }
46
47        public void persistObjects(Object[] objects) {
48            pm.makePersistentAll(objects);
49        }
50
51        public void persistObject(Object object) {
52            pm.makePersistent(object);
53        }
54    }
```

### 例8.30 Queries (查询) 接口

```

1 // 查询类
2 package com.corej2eepatterns.business.hr;
3
4 /**
5  * @author Craig Russell
6  */
7 public interface Queries {
8     Department getDepartment(String name);
9     Employee getEmployee(long id);
10    Project getProject(String name);
11 }

```

### 例8.31 QueriesImpl (查询实现) 类

```

1 // QueryImpl.java
2
3 package com.corej2eepatterns.business.impl;
4
5 import com.corej2eepatterns.business.hr.Department;
6 import com.corej2eepatterns.business.hr.Employee;
7 import com.corej2eepatterns.business.hr.Project;
8 import com.corej2eepatterns.business.hr.Queries;
9
10 import javax.jdo.Query;
11 import javax.jdo.PersistenceManager;
12 import java.util.Collection;
13 import java.util.Iterator;
14
15 /**
16  * @author Craig Russell
17  */
18 public class QueriesImpl implements Queries {
19     private FactoryImpl factory;
20
21     /** 创建QueryImpl 的一个新实例*/
22     public QueriesImpl(FactoryImpl factory) {
23         this.factory = factory;
24     }
25
26     public Department getDepartment(String name) {
27         PersistenceManager pm =
28             factory.getPersistenceManager();
29         Query q = pm.newQuery(
30             DepartmentImpl.class, "this.name == name");
31         q.declareParameters("String name");
32         Collection departments = (Collection) q.execute(name);
33         Iterator iterator = departments.iterator();

```

548

```

34     return (iterator.hasNext()) ?
35         (Department) iterator.next() : null;
36     }
37
38     public Employee getEmployee(long id) {
39         PersistenceManager pm =
40             factory.getPersistenceManager();
41         Query q = pm.newQuery(
42             EmployeeImpl.class, "this.id == id");
43         q.declareParameters("long id");
44         Collection employees =
45             (Collection) q.execute(new Long(id));
46         Iterator iterator = employees.iterator();
47         return (iterator.hasNext()) ?
48             (Employee) iterator.next() : null;
49     }
50
51     public Project getProject(String name) {
52         PersistenceManager pm =
53             factory.getPersistenceManager();
54         Query q = pm.newQuery(
55             ProjectImpl.class, "this.name == name");
56         q.declareParameters("String name");
57         Collection projects = (Collection) q.execute(name);
58         Iterator iterator = projects.iterator();
59         return (iterator.hasNext()) ?
60             (Project) iterator.next() : null;
61     }
62 }
```

### 例8.32 Project (项目) 接口

```

1 // Project.java
2
3 package com.corej2eepatterns.business.hr;
4
5 import java.util.Set;
6
7 /**
8 * @author Craig Russell
9 */
10 public interface Project {
11     String getName();
12     Set getEmployees();
13 }
14
```

### 例8.33 ProjectImpl (项目实现) 类

```

15 // ProjectImpl.java
16
```

```

17 package com.corej2eepatterns.business.impl;
18
19 import com.corej2eepatterns.business.hr.Project;
20
21 import java.util.Collections;
22 import java.util.HashSet;
23 import java.util.Set;
24
25 public class ProjectImpl implements Project {
26     String name;
27     Set employees = new HashSet();
28
29     /** 创建ProjectImpl 的一个新实例*/
30     ProjectImpl(String name) {
31         this.name = name;
32     }
33
34     public Set getEmployees() {
35         return Collections.unmodifiableSet(employees);
36     }
37
38     public String getName() {
39         return name;
40     }
41
42     boolean addEmployee(EmployeeImpl employee) {
43         return employees.add(employee);
44     }
45
46     boolean removeEmployee(EmployeeImpl employee) {
47         return employees.remove(employee);
48     }
49 }
```

#### 例8.34 FullTimeEmployee (全职员工) 接口

```

1 // FullTimeEmployee.java
2
3 package com.corej2eepatterns.business.hr;
4
5 import java.math.BigDecimal;
6
7 /**
8 * @author Craig Russell
9 */
10 public interface FullTimeEmployee extends Employee {
11     BigDecimal getSalary();
12     void setSalary(BigDecimal salary);
13 }
```

**例8.35 FullTimeEmployeeImpl (全职员工实现) 类**

```

1 //FullTimeEmployeeImpl.java
2
3 package com.corej2eepatterns.business.impl;
4
5 import com.corej2eepatterns.business.hr.FullTimeEmployee;
6
7 import java.math.BigDecimal;
8
9 /**
10  * @author Craig Russell
11  */
12 public class FullTimeEmployeeImpl
13     extends EmployeeImpl implements FullTimeEmployee {
14     private BigDecimal salary;
15
16     /** 创建FullTimeEmployeeImpl 的一个新实例*/
17     public FullTimeEmployeeImpl(
18         long id, String firstName, String lastName) {
19         super(id, firstName, lastName);
20     }
21
22     public BigDecimal getSalary() {
23         return salary;
24     }
25
26     public void setSalary(BigDecimal salary) {
27         this.salary = salary;
28     }
29 }
```

550

**例8.36 Department (部门) 接口**

```

1 // Department.java
2 package com.corej2eepatterns.business.hr;
3
4 import java.util.Set;
5
6 public interface Department {
7     String getName();
8     Set getEmployees();
9 }
```

**例8.37 DepartmentImpl (部门实现) 类**

```

1 // DepartmentImpl.java
2 package com.corej2eepatterns.business.impl;
3
4 import com.corej2eepatterns.business.hr.Department;
5
```

551

```

6   import java.util.Collections;
7   import java.util.HashSet;
8   import java.util.Set;
9
10  /**
11   * @author Craig Russell
12   */
13  public class DepartmentImpl implements Department {
14
15      String name;
16      Set employees = new HashSet();
17
18      /** 创建DepartmentImpl 的一个新实例*/
19      DepartmentImpl(String name) {
20          this.name = name;
21      }
22
23      public Set getEmployees() {
24          return Collections.unmodifiableSet(employees);
25      }
26
27      boolean addEmployee(EmployeeImpl employee) {
28          return employees.add(employee);
29      }
30
31      boolean removeEmployee(EmployeeImpl employee) {
32          return employees.remove(employee);
33      }
34
35      public String getName() {
36          return name;
37      }
38  }

```

### 例8.38 Employee (员工) 接口

```

1  // Employee.java
2  package com.corej2eepatterns.business.hr;
3
4  import java.util.Set;
5
6  /**
7   * @author Craig Russell
8   */
9  public interface Employee {
10      long getId();
11      String getFirstName();
12      String getLastName();
13      Department getDepartment();

```

```

14     void setDepartment(Department department);
15     Set getProjects();
16     boolean addProject(Project project);
17     boolean removeProject(Project project);
18 }

```

### 例8.39 EmployeeImpl (员工实现) 类

```

1 // EmployeeImpl.java
2
3 package com.corej2eepatterns.business.impl;
4
5 import com.corej2eepatterns.business.hr.Department;
6 import com.corej2eepatterns.business.hr.Project;
7 import com.corej2eepatterns.business.hr.Employee;
8
9 import java.util.Collections;
10 import java.util.HashSet;
11 import java.util.Set;
12
13 /**
14  * @author Craig Russell
15 */
16 public class EmployeeImpl implements Employee {
17     private long id;
18     private String firstName;
19     private String lastName;
20     private DepartmentImpl department;
21     private Set projects = new HashSet();
22
23     /** 创建EmployeeImpl 的一个新实例*/
24     EmployeeImpl(long id, String firstName, String lastName) {
25         this.id = id;
26         this.firstName = firstName;
27         this.lastName = lastName;
28     }
29
30     public boolean addProject(Project project) {
31         boolean result = projects.add(project);
32         if (result) {
33             ((ProjectImpl) project).addEmployee(this);
34         }
35         return result;
36     }
37
38     public boolean removeProject(Project project) {
39         boolean result = projects.remove(project);
40         if (result) {
41             ((ProjectImpl) project).removeEmployee(this);

```

```

42     }
43     return result;
44 }
45
46     public Department getDepartment() {
47         return department;
48     }
49
50     public void setDepartment(Department department) {
51         if (this.department != null) {
52             this.department.removeEmployee(this);
53         }
54         this.department = (DepartmentImpl) department;
55         this.department.addEmployee(this);
56     }
57
58     . . .
59
60     public Set getProjects() {
61         return Collections.unmodifiableSet(projects);
62     }
63 }

```

554

## 效果

- **创建自己的持久化框架是一件复杂的工作**

实现业务领域存储模式和透明持久化所需的所有特性并非易事，这不仅是因为持久化问题本身的复杂性，更因为本模式框架中诸多参与者之间存在复杂的交互。因此，除非其余所有选择尽皆不可行，否则不应考虑自己实现一套透明持久化框架。

- **多层对象树的加载和存储需要优化技术**

业务对象可能存在极其复杂的多层次体系和关联。在对彼此相关的业务对象进行持久化操作时，可能只想持久化多层次体系中被修改了的那些对象。同样，在加载业务对象多层次体系时，也会希望提供各种级别的懒加载方案：一开始只加载其中最常用的部分，在确实需要时再加载其余的部分。

- **增进对持久化框架的理解**

如果使用第三方持久化框架，对业务领域存储模式的理解将极大地帮助理解该框架。对比业务领域存储模式的介绍，可以很快理解该框架是如何实现透明持久化的。

- **小规模的对象模型可能不需要完整的持久化框架**

如果对象模型非常简单，并且只需要对其进行简单的持久化操作，使用基于业务领域存储模式的持久化框架就有些大材小用。在这种情况下，一个基于数据访问对象模式的简单框架就足够了。

- **提高可持久对象模型的可测试性**

业务领域存储模式将持久化逻辑与持久性的业务对象分离，从而极大地提升应用系统的可

测试性，因为在测试对象模型时不必使用真实的持久化机制。由于持久化机制是透明的，可以在完成对业务对象模型和业务逻辑的测试之后再切换到真实的持久化机制。

- 分离业务对象模型和持久化逻辑

业务领域存储模式提供了透明的持久化机制，因此业务对象不必包含任何与持久化操作相关的代码。这样，开发者就不必分心实现业务对象中错综复杂的持久化逻辑。

555

## 相关模式

- 工作单元 (*Unit of Work*) [PEAA]

维护一份列表，在其中记录业务事务影响到的所有对象。工作单元模式与 Persistence Manager（持久化管理器）密切相关。

- 查询对象 (*Query Object*) [PEAA]

代表一次数据库查询的对象。业务领域存储模式中的Query（查询）角色与此模式相关。

- 数据映射器 (*Data Mapper*) [PEAA]

由Mapper（映射器）对象构成的一个层次，负责管理数据在对象和数据库之间的转移。StateManager（状态管理器）与此模式相关。

- 表数据门径 (*Table Data Gateway*) [PEAA]

作为数据库门户的对象。StoreManager（状态管理器）与此模式相关。

- 从属映射 (*Dependent Mapping*) [PEAA]

用父类执行子类的数据库映射。具有继承关系的对象和PersistMap（持久化表）角色都与此模式相关。

- 业务领域模型 (*Domain Model*) [PEAA]

拥有业务数据、可以执行业务行为的对象模型。BusinessObject（业务对象）与此模式相关。

- 数据传输对象 (*Data Transfer Object*) [PEAA]

等同于传输对象。

- 标识符表 (*Identity Map*) [PEAA]

确保每个对象只被加载一次。StateManager（状态管理器）与此模式相关。

- 懒装载 (*Lazy Load*) [PEAA]

对象中只包含部分的数据，但知道如何获取完整的数据。StateManager（状态管理器）与 StoreManager（存储管理器）的交互中需要实现懒装载模式。

556

## Web Service中转

### 问题

需要为一个或多个服务提供通过XML和Web协议进行访问的途径。

J2EE应用程序通过服务门面暴露粗粒度的业务服务。但是，如果作为Web Service，这些服务接口的粒度可能还是过细，或者它们的设计思路决定了不适于将它们暴露到应用程序之外。

另一方面，企业服务可能运行在不同的平台上，用不同的语言实现——例如J2EE和.NET应用程序。这种异质性常常造成它们无法彼此兼容，增加系统之间无缝集成的难度。这也就意味着不管面对的是J2EE服务、.NET服务还是别的遗留服务，都不希望直接将它们逐一作为Web Service暴露出来。即便是对于粗粒度的J2EE服务，出于业务需求的考虑，通常也只希望暴露其中的一部分服务方法供Web Service访问。而且，系统常常需要聚合几个现有的服务，使它们彼此协作。

**注意：**W3C给Web Service做了如下定义：

- Web Service是一个软件系统，它的唯一身份由URI[RFC 2396]决定，它的公开接口和绑定(binding)<sup>Θ</sup>都用XML定义、描述。其他软件系统可以查找到Web Service的定义，并以其定义中所指定的方式与它交互。Web Service与其他系统的交互采用基于XML的消息，并使用Internet协议传输该消息。
- 一组终端(end point)的集合体。
- 终端是连接绑定和网络地址的纽带，由一个URL指定，用于与一个服务实例通信。端口(port)则指定了一个地址，通过这个特定的地址，使用特定的协议和数据格式，就可以访问一项服务。

## 约束

- 需要复用现有的服务，将它们暴露给客户端使用。
- 对于暴露给客户端的服务，希望监视它们的使用情况，甚至还有可能加以限制，这取决于业务需求和系统资源占用情况。
- 必须用公开的标准来暴露服务接口，以便集成异质应用程序。
- 业务需求和现有服务功能之间可能存在一些差距，希望弥补这一差距。

557

## 解决方案

使用Web Service中转暴露一个或多个可通过XML和web协议访问的服务，并将对服务的请求转发给真实的服务组件。

Web Service中转是一个粗粒度的、以Web Service形式暴露出来的服务组件。它负责协调一个或多个服务组件之间的交互、对响应信息进行聚合，还可能要负责划分或补偿<sup>Θ</sup>事务。Web Service中转可以用RPC(远程过程调用)风格的接口(例如WSDL)暴露自己，也可以用消息接口暴露自己。如果选择RPC风格的接口，可以用J2EE 1.4提供的session bean web service终端或者JAX-RPC实现对象作为Web Service终端——两者都以WSDL的格式暴露服务接口。图8-23展示了Web Service中转模式的结构。

- Θ 在Web Service中，绑定(binding)是指以下几部分的合成物：一个具体的通信协议；操作所需的数据类型；以及针对一个特定端口类型而定义的消息。
- Θ 对于嵌套的事务，简单的回滚操作往往无法满足ACID的要求——即使外层事务被回滚，内层事务却已经被提交了。在这种情况下，需要这样一种机制：允许内层事务提交，但事务管理者可以在需要的时候进行某些操作以取消内层事务已提交的操作。这种机制被称为“事务补偿”。

## 结构

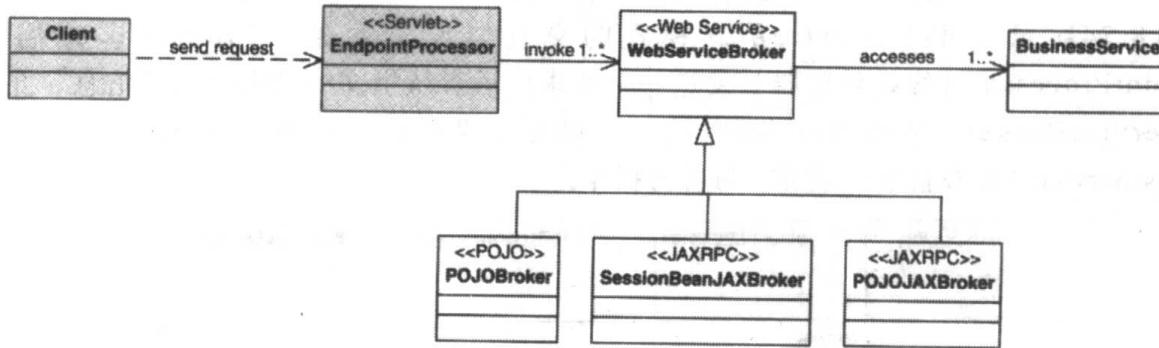


图8-23 Web Service 中转类图

## 参与者和责任

图8-23已经画出了本模式的参与者，分别介绍如下：

### Client（客户端）

Client可以是任何能够发出Web Service请求的系统。

558

### EndpointProcessor（终端处理器）

EndpointProcessor是一个servelt，它是客户端接触到Web Service的第一个入口，负责接收、处理请求。这里的请求通常是基于HTTP请求的，例如SOAP请求。一般来说，EndpointProcessor是内建在运行时系统中的，使用JAX-RPC时就是如此。也可以自己创建一个EndpointProcessor。

### WebServiceBroker（Web Service中转）

WebServiceBroker是一个Web Service，它为其他一个或多个服务担当中转——这些服务可能是J2EE服务，例如会话界面或者应用服务；也可能是遗留EIS系统。WebServiceBroker的实现方式有三种。

#### POJOBroker（POJO中转）

POJOBroker是以POJO形式实现的WebServiceBroker（Web Service中转）。

#### SessionBeanJAXBroker（Session Bean JAX中转）

SessionBeanJAXBroker是一个基于JAX-RPC的session bean，它被声明为一个Web Service终端。它通常以无状态session bean的方式实现，用WSDL来描述。SessionBeanJAXBroker组件需要在兼容EJB 2.1规范的EJB容器中运行。

#### POJOJAXBroker（POJO JAX中转）

和SessionBeanJAXBroker（Session Bean JAX中转）一样，POJOJAXBroker也是基于JAX-RPC的。但POJOJAXBroker不是session bean，而是一个POJO，它无须运行在EJB容器中——只要有web容器就够了。POJOJAXBroker的代码编写和RMI组件很相似。另外，它必须借助JAX-RPC运行时环境才能正常工作。

### BusinessService (业务服务)

BusinessService可以是J2EE的会话界面、应用服务，也可以是在EIS之上的POJO门面。

图8-24展示了Web Service中转模式的交互情况。首先，Client（客户端）向EndpointProcessor（终端处理器）发送一个请求；后者从报文中抽取出请求信息，并调用WebServiceBroker（Web Service中转）；随后，WebServiceBroker调用一个或多个BusinessService（业务服务），完成实际业务操作。

559

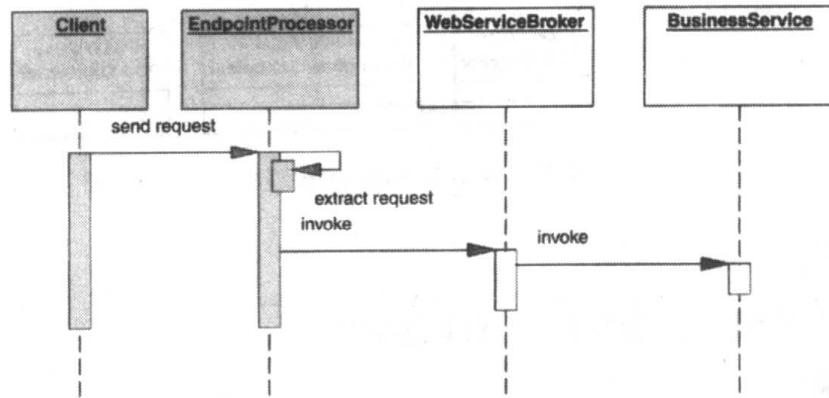


图8-24 Web Service中转交互图

### 策略

#### 定制XML消息策略

定制XML消息策略的基本思想是：直接向Web Service中转发送消息（序列图如图8-25所示）。XMLDoc代表的可能是在HTTP上发送的XML文档，也可能是在SOAP消息中封装的XML文档。

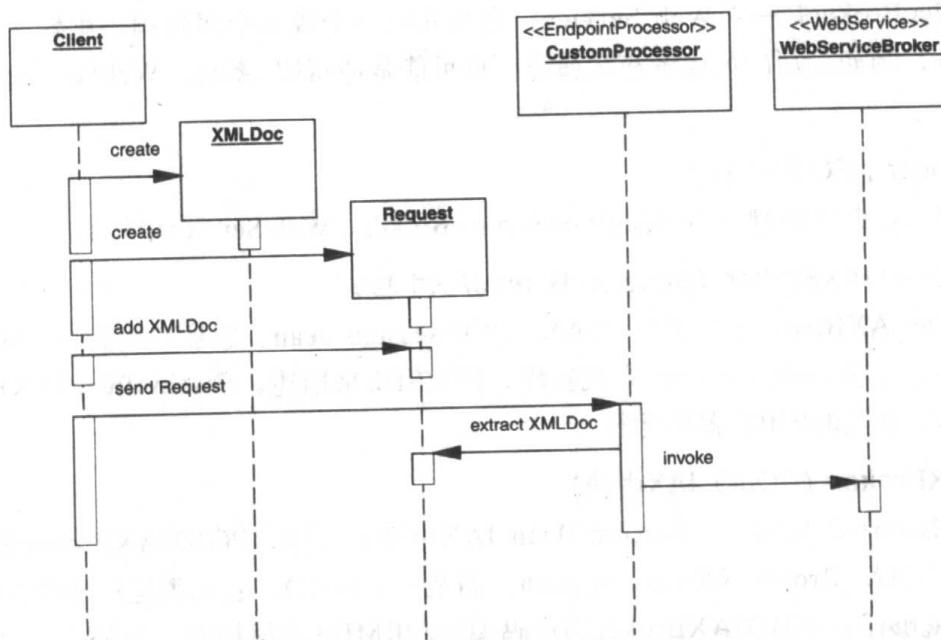


图8-25 定制XML消息策略序列图

图8-25展示了该策略的交互情况：首先，Client（客户端）向CustomProcessor（定制处理器，这是一个servlet）发出请求；收到请求之后，CustomProcessor从中提取出XMLDoc，对其进行处理，然后调用WebServiceBroker（Web Service中转）。

### Java绑定器策略

Java绑定器策略借助Java绑定技术（例如JAXB）来处理XMLDoc，从中生成传输对象。使用JAXB非常简单，但需要预先做一些准备工作：首先必须定义出XML schema或者DTD，然后用Java绑定器工具生成所需的传输对象类。在运行期，Java绑定器运行时环境会处理XMLDoc，组装出传输对象。

实现这一策略的途径有两种。可以将XMLDoc发送给CustomProcessor（定制处理器），由后者调用Java绑定器机制；也可以用基于JAX-RPC的WebServiceBroker（Web Service中转）来接收被封装为javax.xml.transform.Source数据类型的XML文档，从中提取出真正的XML文档，然后发送给Java绑定器进行处理。

图8-26展示了使用CustomProcessor（定制处理器）实现Java绑定器策略的类图。560

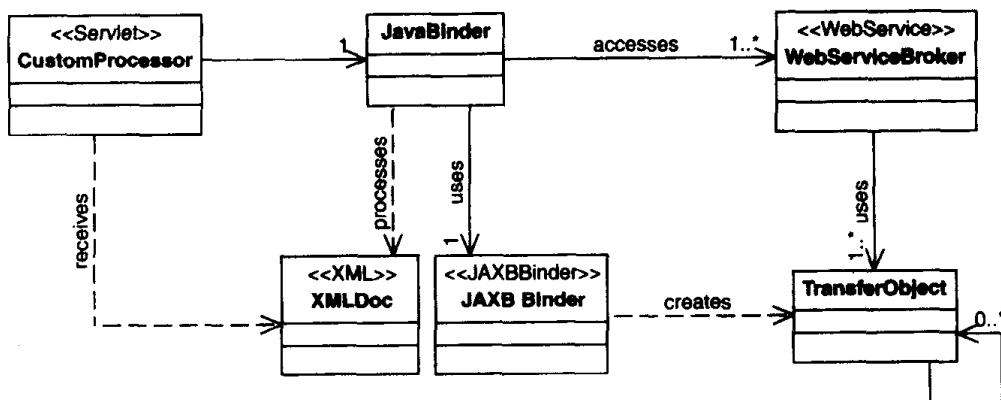


图8-26 Java绑定器策略类图

561

### Java绑定器策略示例

图8-27展示了用JAXB实现Java绑定器策略的一个例子：Purchase Order（订购单）。

首先，我们要用JAX-B编译器——xjc——来处理名为PurchaseOrder.xsd的XMLschema（见图8-28）。具体的命令如下：

```
xjc PurchaseOrder.xsd
```

请注意，默认情况下，xjc会将生成的文件放入“generated”目录。JAXB将根据这份schema生成下列类：

- PurchaseOrder（订购单）
- Address（地址）
- Item（货品）

图8-28展示了PurchaseOrder（订货单）XML Schema的结构，创建一份订购单的步骤如下：

- Client（客户端）创建一个PurchaseOrder XML文档（例8.40），将其放入HTTP请求中，发送给CustomProcessorServlet（定制处理器Servlet）的doPost()方法。

562

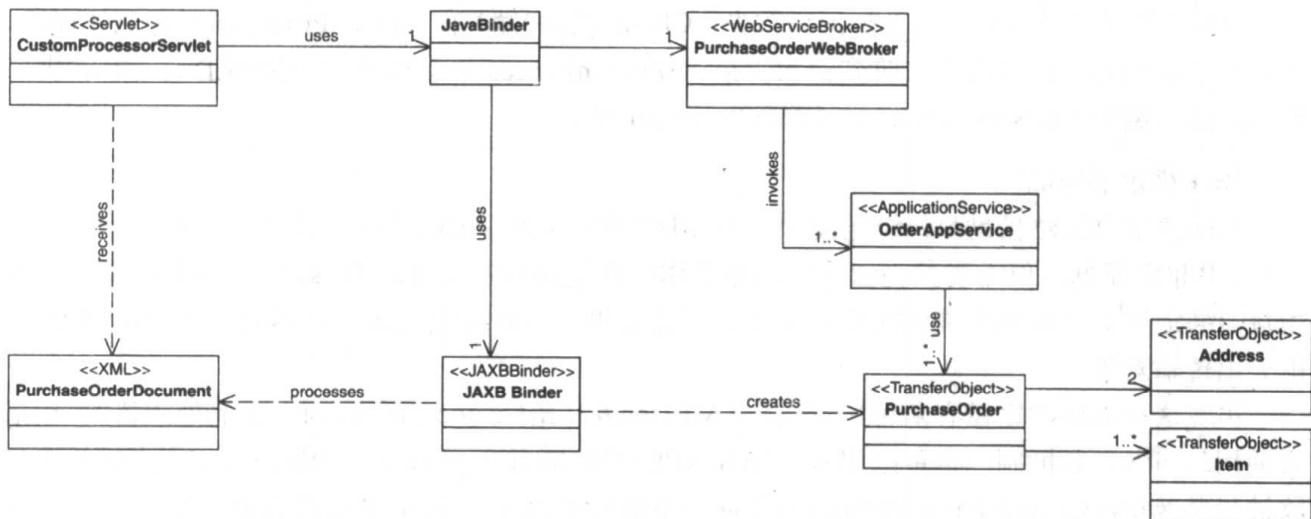


图8-27 Java绑定器策略类图：Purchase Order示例

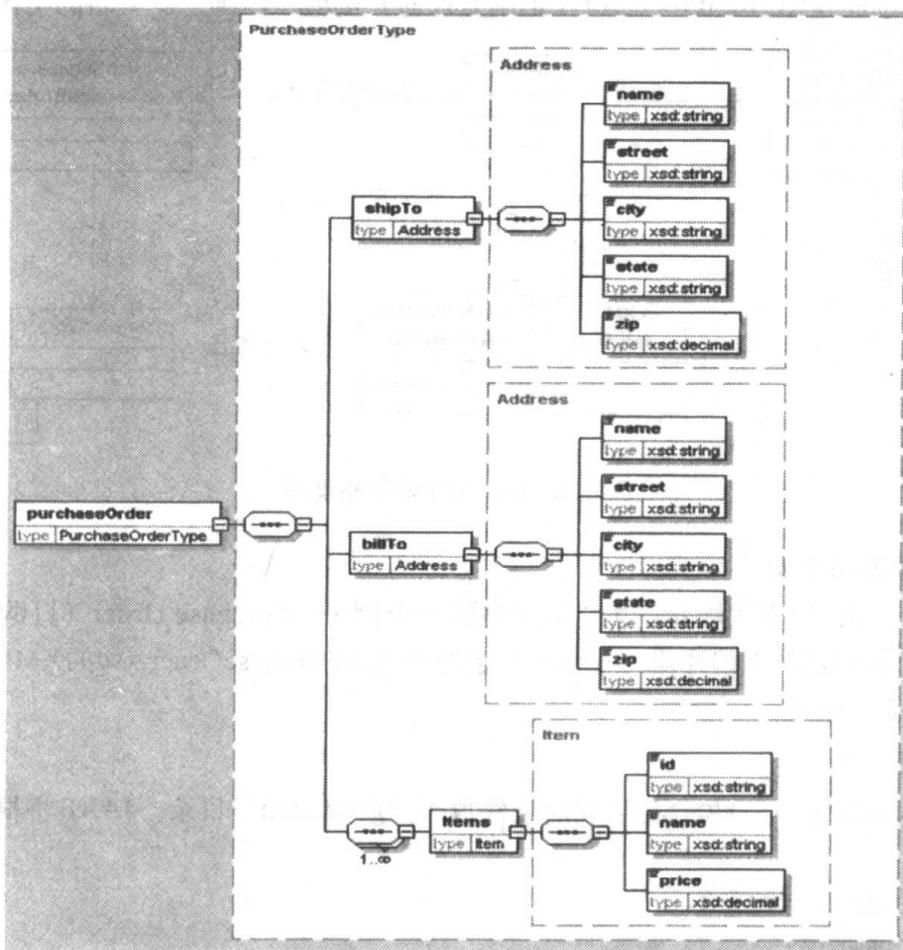


图8-28 PurchaseOrder XML Schema图

- CustomProcessorServlet从HTTPRequest中提取出PurchaseOrder XML字符串，然后调用JavaBinder（Java绑定器）的routeDocument()方法，将XML字符串传递给JavaBinder（参见例8.42）。

- JavaBinder创建一个JAXBContext对象，并告诉它：JAXB所生成的文件位于“generated”包中。然后，JAXB的Unmarshaller（解列器）将PurchaseOrder XML字符串解列（unmarshal）<sup>Θ</sup>为PurchaseOrder对象树，并作为一个Object返回。
- JavaBinder判断出返回的对象是PurchaseOrder，随后创建一个PurchaseOrderWebBroker（订货单Web中转，参见例8.43），并将PurchaseOrder对象传递给后者的placeOrder()方法。
- PurchaseOrderWebBroker创建一个OrderAppService（订单应用服务，参见例8.44），并将PurchaseOrder对象传递给后者的placeOrder()方法。
- OrderAppService从PurchaseObject树中取出所需的数据，并执行适当的业务逻辑。

例8.40至例8.44展示了本例涉及的XML文件和Java类。

#### 例8.40 PurchaseOrder（订货单）XML文件

```

1  <?xml version="1.0" encoding="UTF-8"?>
2
3  <purchaseOrder orderDate="2003-01-15"
4    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
5    xsi:noNamespaceSchemaLocation="po2.xsd">
6
7      <shipTo>
8        <name>Jo Miller</name>
9        <street>Grand View Drive</street>
10       <city>Bethesda</city>
11       <state>MD</state>
12       <zip>20816</zip>
13     </shipTo>
14     <billTo>
15       <name>Jane Miller</name>
16       <street>Grand View Drive</street>
17       <city>Bethesda</city>
18       <state>MD</state>
19       <zip>20816</zip>
20     </billTo>
21     <Items>
22       <id>101</id>
23       <name>Core J2EE Patterns</name>
24       <price>59.99</price>
25     </Items>
26     <Items>
27       <id>102</id>
28       <name>Enterprise Patterns</name>
29       <price>59.99</price>
30     </Items>
31     <Items>
32       <id>103</id>
33       <name>WebService Patterns</name>

```

564

<sup>Θ</sup> unmarshal，JAXB术语，与marshal相对。从Java对象生成XML文档，这称为marshal，所以相应地，从XML文档反向生成Java对象，自然就称为unmarshal。今试译为“解列”。

```

31      <price>59.99</price>
32  </Items>
33 </purchaseOrder>
```

#### 例8.41 CustomProcessorServlet (定制处理器Servlet) 类

```

1  import javax.servlet.RequestDispatcher;
2  import javax.servlet.ServletException;
3  import javax.servlet.http.HttpServlet;
4  import javax.servlet.http.HttpServletRequest;
5  import javax.servlet.http.HttpServletResponse;
6
7  public class CustomProcessorServlet extends HttpServlet {
8      public String getServletInfo() {
9          return "Servlet description";
10     }
11
12     protected void dispatch(HttpServletRequest request,
13         HttpServletResponse response, String page)
14         throws javax.servlet.ServletException,
15         java.io.IOException {
16
17         RequestDispatcher dispatcher =
18             getServletContext().getRequestDispatcher(page);
19         dispatcher.forward(request, response);
20     }
21
22     public void init() throws ServletException {
23     }
24
25     protected void doGet(HttpServletRequest request,
26         HttpServletResponse response)
27         throws javax.servlet.ServletException,
28         java.io.IOException {
29     }
30
31     protected void doPost(HttpServletRequest request,
32         HttpServletResponse response)
33         throws javax.servlet.ServletException,
34         java.io.IOException {
35         String xmlDocument =
36             request.getParameter("PurchaseOrder");
37         JavaBinder binder = new JavaBinder();
38         binder.routeDocument(xmlDocument);
39     }
40 }
```

### 例8.42 JavaBinder (Java绑定器) 类

```
1 import generated.Item;
2 import generated.PurchaseOrder;
3
4 import javax.xml.bind.JAXBContext;
5 import javax.xml.bind.JAXBException;
6 import javax.xml.bind.Unmarshaller;
7 import javax.xml.transform.stream.StreamSource;
8 import java.io.StringReader;
9 import java.util.Iterator;
10 import java.util.List;
11
12 public class JavaBinder {
13     public JavaBinder() {
14     }
15
16     private Object parse(String xmlDocument) {
17         Object o = null;
18         try {
19             JAXBContext jc =
20                 JAXBContext.newInstance("generated");
21             Unmarshaller u = jc.createUnmarshaller();
22             o = u.unmarshal(new StreamSource(
23                 new StringReader(xmlDocument)));
24         } catch (JAXBException e) {
25         }
26         return o;
27     }
28
29     public void routeDocument(String xmlDocument) {
30         Object o = parse(xmlDocument);
31         if (o instanceof PurchaseOrder) {
32             PurchaseOrder po = (PurchaseOrder) o;
33             printIt(po);
34             PurchaseOrderWebBroker broker =
35                 new PurchaseOrderWebBroker();
36             broker.placeOrder(po);
37         }
38     }
39
40     private void printIt(PurchaseOrder po) {
41         System.out.println(po.getBillTo());
42         System.out.println(po.getShipTo());
43         System.out.println(po.getOrderDate());
44         List items = po.getItems();
45         Iterator i = items.iterator();
46         while (i.hasNext()) {
```

```

47     Item item = (Item) i.next();
48     System.out.println("Id : " + item.getId() +
49         " Name : " + item.getName() +
50         " Price : " + item.getPrice());
51 }
52 }
53 }
```

567

#### 例8.43 PurchaseOrderWebBroker (订货单Web中转) 类

```

1 import generated.PurchaseOrder;
2
3 public class PurchaseOrderWebBroker {
4     public void placeOrder(PurchaseOrder po) {
5         // 如果需要，在此处进行安全性检查
6         OrderAppService as = new OrderAppService();
7         as.placeOrder(po);
8     }
9 }
```

#### 例8.44 OrderAppService (订单应用服务) 类

```

1 import generated.Address;
2 import generated.PurchaseOrder;
3
4 import java.util.List;
5
6 public class OrderAppService {
7     public OrderAppService() {
8     }
9
10    public void placeOrder(PurchaseOrder purchaseOrder) {
11        Address shipTo = purchaseOrder.getShipTo();
12        Address billTo = purchaseOrder.getBillTo();
13        List items = purchaseOrder.getItems();
14
15        // 更新数据存储
16    }
17 }
```

### JAX-RPC策略

JAX-RPC策略用JAX-RPC作为通信机制。在这种策略中，WebServiceBroker (Web Service中转) 被实现为一个符合EJB 2.1规范的无状态session bean，并作为一个Web Service终端暴露出来。请注意，作为Web Service终端的session bean必须满足两点限制：

- Web Service session bean必须有一个Remote (远程) 类和一个Implementation (实现) 类。
- Web Service session bean必须是无状态的。

图8-29展示了这种策略的结构。EndpointProcessor (终端处理器) 是一个基于JAX-RPC终端的servlet，它负责处理外来的请求，并将其转发给WebServiceBroker (Web Service中转)。请注

568

意，这是内建在JAX-RPC运行时环境中的，对开发者完全透明。另外，如图所示，WebServiceBroker是由WebServiceBrokerWSDL（Web Service中转WSDL）来描述的。后者不是一个对象，而是一份基于XML的WSDL文件，其中定义了WebServiceBroker的接口。

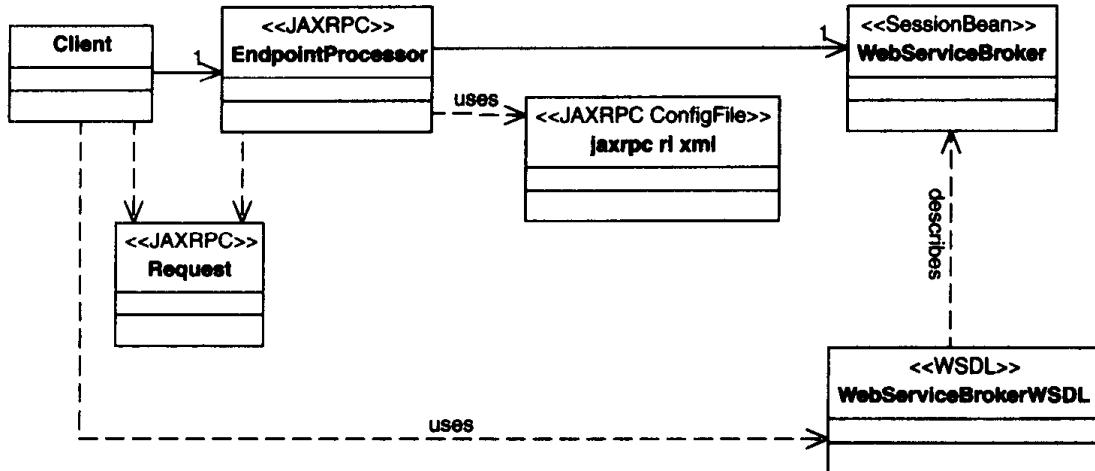


图8-29 JAX-RPC策略类图

### JAX-RPC策略示例

图8-30展示了JAX-RPC策略示例程序中各个类的类图。如图所示，Company WebService Broker（公司Web Service中转）是一个被声明为Web Service终端的无状态session bean，它会调用CompanyFacadeSession（公司门面会话）和EmployeeFacadeSession（员工门面会话）——两者都是会话门面。

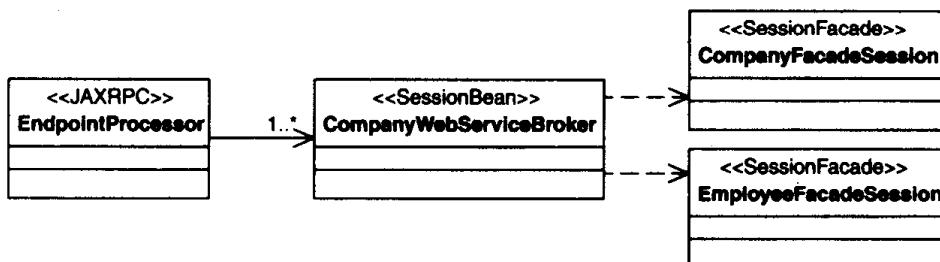


图8-30 JAX-RPC Style策略示例

### 示例代码

下列代码片段出自EmployeeFacade（员工门面）、EmployeeFacadeHome（员工门面Home）和EmployeeFacadeSession（员工门面会话）三个类，它们共同组成了EmployeeFacade组件（见例8.45）。 569

EmployeeFacade（员工门面）和EmployeeFacadeSession提供了两个业务方法：

```

addEmployee( String employeeId, String lastName, String firstName )
assignToDivision( String divisionId )
    
```

请注意，assignToDivision()方法只需要divisionId参数。这也就是说，该方法的调用者必须面对一个有状态的EmployeeFacade，其中包含了employeeId信息。

在EmployeeFacadeHome中，有两个用于创建EmployeeFacade的方法：

```

1 create()
2 create( String companyId, String employeeId )

```

**EmployeeFacadeSession**是有状态的，可以用无参数的create()方法或带有参数的create(companyId, employeeId)方法来创建它。完成创建之后，assignToDivision()和giveRaise()方法会调用业务对象或者DAO来对数据进行持久化。

#### 例8.45 EmployeeFacade（员工门面）、EmployeeFacadeHome（员工门面Home）和EmployeeFacadeSession（员工门面会话）代码示例

```

1 // EmployeeFacade
2 import java.rmi.RemoteException;
3
4 public interface EmployeeFacade
5     extends javax.ejb.EJBLocalObject {
6     public void assignToDivision(String divisionId);
7     public void giveRaise( float amount );
8 }

1 // EmployeeFacadeHome
2 import javax.ejb.CreateException;
3 import javax.ejb.FinderException;
4
5 public interface EmployeeFacadeHome
6     extends javax.ejb.EJBLocalHome{
7     public EmployeeFacade create( )
8         throws CreateException;
9     public EmployeeFacade create(
10         String companyId, String employeeId )
11         throws CreateException;
12 }

1 // EmployeeFacadeSession
2 import javax.ejb.SessionContext;
3 import javax.ejb.EJBException;
4 import javax.ejb.CreateException;
5 import javax.ejb.FinderException;
6
7 public class EmployeeFacadeSession
8     implements javax.ejb.SessionBean {
9     String companyId;
10    String employeeId;
11
12    public void assignToDivision(String divisionId) {
13        // 调用业务对象或DAO, 把一个员工
14        // 添加到Division (部门)
15    }
16    public void giveRaise( float amount ) {

```

```

17     // 调用业务对象或DAO, 添加
18     // 新员工
19 }
20 public void setSessionContext(SessionContext sessionContext)
21     throws EJBException {     }
22
23 public void ejbRemove() throws EJBException {     }
24 public void ejbActivate() throws EJBException {     }
25 public void ejbPassivate() throws EJBException {     }
26 public void ejbCreate( ) throws CreateException {     }
27 public void ejbCreate( String companyId, String employeeId )
28     throws CreateException {
29     this.companyId = companyId;
30     this.employeeId = employeeId;
31 }
32 }
```

下列代码片段来自**CompanyFacade**（公司门面）、**CompanyFacadeHome**（公司门面Home）和**CompanyFacadeSession**（公司门面会话）三个类，它们共同组成了**CompanyFacade**组件（见例8.46）。**CompanyFacade**和**CompanyFacadeSession**有三个业务方法：

571

```

createCompany( String companyId )
addDivision( String divisionId, String divisionName )
addEmployee( String employeeId, String lastName, String firstName)
```

**CompanyFacadeHome**有两个创建方法：

```

create()
create( String companyId )
```

#### 例8.46 CompanyFacade（公司门面）、CompanyFacadeHome（公司门面Home）和CompanyFacadeSession（公司门面会话）代码示例

```

1 //CompanyFacade
2 import java.rmi.RemoteException;
3
4 public interface CompanyFacade
5     extends javax.ejb.EJBLocalObject {
6     public void createCompany(
7         String companyId, String companyName );
8     public void addDivision(
9         String divisionId, String divisionName );
10    public void addEmployee(String employeeId,
11        String lastName, String firstName );
12 }
13
14 // CompanyFacadeHome
15 import javax.ejb.CreateException;
16
17 public interface CompanyFacadeHome
```

```

5      extends javax.ejb.EJBLocalHome {
6      public CompanyFacade create( )
7          throws CreateException;
8      public CompanyFacade create( String companyId )
9          throws CreateException;
10 }

1 // CompanyFacadeSession
2 import javax.ejb.SessionContext;
3 import javax.ejb.EJBException;
4 import javax.ejb.CreateException;
5
6 public class CompanyFacadeSession
7     implements javax.ejb.SessionBean {
8     String companyId;
9
[572]10    public void createCompany( String id, String name ) {
11        // 创建Company (公司) 并令其持久化
12        // 保存companyId, 用于有状态的对话
13        this.companyId = id;
14    }
15
16    public void addDivision(
17        String divisionId, String divisionName) {
18        // 调用业务对象或DAO, 给一个Company (公司)
19        // 添加一个Division (部门)
20    }
21
22    public void addEmployee(String employeeId,
23        String lastName, String firstName) {
24        // 调用业务对象或DAO, 把员工分配给
25        // Division (部门)
26    }
27
28    public void setSessionContext(SessionContext sessionContext)
29        throws EJBException {
30    }
31
32    public void ejbRemove() throws EJBException { }
33    public void ejbActivate() throws EJBException { }
34    public void ejbPassivate() throws EJBException { }
35    public void ejbCreate() throws CreateException { }
36    public void ejbCreate( String companyId )
37        throws CreateException {
38        // Company (公司) 业务对象的寻址
39        this.companyId = companyId;
40    }
41 }

```

下列代码片段出自 CompanyFacadeService (公司门面服务) 和 CompanyFacade (公司门面) 类, 它们共同组成了 WebServiceFacade (Web Service 门面) 组件 (见例8.47)。由于 CompanyFacadeService 是无状态的, 所有与操作相关的信息都必须在方法的参数中传递进来。这两个类提供了四个方法:

573

```
createCompany( String companyId, String companyName )
addCompanyDivision( String companyId, String divisionId, String divisionName )
addEmployee( String companyId, String employeeId,
    String lastName, String firstName )
assignEmployeeToDivision (String companyId, String employeeId, String divisionId )
```

可以看到, 这几个方法都在参数列表中传入了执行事务操作所需的全部数据。表8-1列出了 CompanyWebServiceBroker (公司 Web Service 中转) 的所有方法, 以及各方法相应调用的 SessionFacade (会话门面) 方法。

**表8-1 CompanyWebServiceBroker、CompanyFacadeSession (公司门面会话) 和EmployeeFacadeSession (员工门面会话) 的方法**

CompanyWebServiceBroker方法	被调用的SessionFacade方法
createCompany()	CompanyFacadeSession.createCompany()
addCompanyDivision()	CompanyFacadeSession.addDivison()
addEmployee()	CompanyFacadeSession.addEmployee()
assignEmployeeToDivision()	EmployeeFacadeSession.assignToDivision()

### 例8.47 CompanyWebService (公司 Web Service) 和 CompanyWebServiceBroker (公司 Web Service 中转)

```
1 import java.rmi.RemoteException;
2
3 public interface CompanyWebService extends javax.ejb.EJBObject {
4     public void createCompany(
5         String companyId, String companyName )
6         throws RemoteException;
7
8     public void addCompanyDivision(String companyId,
9         String divisionId, String divisionName )
10    throws RemoteException;
11
12    public void addEmployee(String companyId, String employeeId,
13        String lastName, String firstName )
14        throws RemoteException;
15
16    public void assignEmployeeToDivision(String companyId,
17        String divisionId, String employeeId )
18        throws RemoteException;
19 }
20
21 // CompanyWebServiceBroker
22 import javax.ejb.SessionContext;
23 import javax.ejb.EJBException;
```

574

```
4 import javax.ejb.CreateException;
5
6 public class CompanyWebServiceBroker
7     implements javax.ejb.SessionBean {
8     public void setSessionContext(SessionContext sessionContext)
9         throws EJBException {    }
10
11    public void ejbRemove() throws EJBException {    }
12    public void ejbActivate() throws EJBException {    }
13    public void ejbPassivate() throws EJBException {    }
14    public void ejbCreate() throws CreateException {    }
15
16    private CompanyFacadeHome getCompanyHome()
17        throws ServiceLocatorException {
18        CompanyFacadeHome home = (CompanyFacadeHome)
19            ServiceLocator.getInstance().getLocalHome(
20                "CompanyFacade");
21        return home;
22    }
23
24    public void createCompany(
25        String companyId, String companyName ) {
26        try {
27            CompanyFacadeHome home = getCompanyHome();
28            CompanyFacade cf = home.create();
29            cf.createCompany( companyId, companyName );
30        } catch (CreateException e) {
31        } catch (ServiceLocatorException e) {
32        }
33    }
34
35    public void addCompanyDivision(String companyId,
36        String divisionId, String divisionName ) {
37        try {
38            CompanyFacadeHome home = getCompanyHome();
39            CompanyFacade cf = home.create( companyId );
40            cf.addDivision( divisionId, divisionName );
41        } catch (ServiceLocatorException e) {
42        } catch (CreateException e) {
43        }
44    }
45
46    public void addEmployee(String companyId, String employeeId,
47        String lastName, String firstName ) {
48        try {
49            CompanyFacadeHome home = getCompanyHome();
50            CompanyFacade cf = home.create( companyId );
```

```

51         cf.addEmployee(employeeId, lastName, firstName);
52     } catch (ServiceLocatorException e) {
53     } catch (CreateException e) {
54     }
55 }
56
57 private EmployeeFacadeHome getEmployeeHome()
58     throws ServiceLocatorException {
59     EmployeeFacadeHome home = (EmployeeFacadeHome)
60         ServiceLocator.getInstance().getLocalHome(
61             "EmployeeFacade");
62     return home;
63 }
64
65 public void assignEmployeeToDivision(String companyId,
66     String divisionId, String employeeId ) {
67     try {
68         EmployeeFacadeHome home = getEmployeeHome();
69         EmployeeFacade ef =
70             home.create( companyId, employeeId );
71         ef.assignToDivision( divisionId );
72     } catch (ServiceLocatorException e) {
73     } catch (CreateException e) {
74     }
75 }
76 }
```

例8.48展示了名为CompanyWebService.wsdl的WSDL文件，EJB 2.1容器将使用这份文件来描述CompanyWebServiceBroker session bean。

576

#### 例8.48 CompanyWebService.wsdl

```

1  <?xml version="1.0" encoding="UTF-8"?>
2  . . .
3
4  <wsdl:message name="createCompanyRequest">
5      <wsdl:part name="CompanyId" type="xsd:string"/>
6      <wsdl:part name="CompanyName" type="xsd:string"/>
7  </wsdl:message>
8  <wsdl:message name="addEmployeeRequest">
9      <wsdl:part name="CompanyId" type="xsd:string"/>
10     <wsdl:part name="EmployeeId" type="xsd:string"/>
11     <wsdl:part name="LastName" type="xsd:string"/>
12     <wsdl:part name="FirstName" type="xsd:string"/>
13 </wsdl:message>
14 <wsdl:message name="addCompanyDivisionRequest">
15     <wsdl:part name="CompanyId" type="xsd:string"/>
16     <wsdl:part name="DivisionId" type="xsd:string"/>
17     <wsdl:part name="DivisionName" type="xsd:string"/>
```

```
18      </wsdl:message>
19      <wsdl:message name="assignEmployeeToDivisionRequest">
20          <wsdl:part name="CompanyId" type="xsd:string"/>
21          <wsdl:part name="DivisionId" type="xsd:string"/>
22          <wsdl:part name="EmployeeId" type="xsd:string"/>
23      </wsdl:message>
24
25      <wsdl:message name="addEmployeeResponse">
26      </wsdl:message>
27
28      <wsdl:message name="createCompanyResponse">
29      </wsdl:message>
30
31      <wsdl:message name="addCompanyDivisionResponse">
32      </wsdl:message>
33
34      <wsdl:message name="assignEmployeeToDivisionResponse">
35      </wsdl:message>
36
37      . . .
38
39      </wsdl:portType>
40
577 41      <wsdl:binding name="CompanyWebServiceSoapServiceBinding"
42          type="impl:CompanyWebServiceBroker">
43          <wsdlsoap:binding style="rpc" transport=
44              "http://schemas.xmlsoap.org/soap/http"/>
45          <wsdl:operation name="createCompany">
46              <wsdlsoap:operation soapAction="" />
47              <wsdl:input name="createCompanyRequest">
48                  <wsdlsoap:body use="encoded" encodingStyle=
49                      "http://schemas.xmlsoap.org/soap/encoding/"
50                      namespace="http://DefaultNamespace" />
51              </wsdl:input>
52              <wsdl:output name="createCompanyResponse">
53                  <wsdlsoap:body use="encoded"
54                      encodingStyle=
55                      "http://schemas.xmlsoap.org/soap/encoding/"
56                      namespace="http://DefaultNamespace" />
57              </wsdl:output>
58          </wsdl:operation>
59
60          <wsdl:operation name="addCompanyDivision">
61              <wsdlsoap:operation soapAction="" />
62              <wsdl:input name="addCompanyDivisionRequest">
63                  <wsdlsoap:body use="encoded"
64                      encodingStyle=
65                      "http://schemas.xmlsoap.org/soap/encoding/"
```

```

66             namespace="http://DefaultNamespace" />
67         </wsdl:input>
68     . . .
70
71     <wsdl:service name=" CompanyWebService " >
72         <wsdl:port name=" CompanyWebServicePort " >
73             binding=
74                 "impl: CompanyWebServiceBrokerSoapBinding " >
75             <wsdlsoap:address location=
76                 "http://localhost:8080/CompanyWebService" />
77         </wsdl:port>
78     </wsdl:service>
79
80 </wsdl:definitions>

```

578

例8.49展示了创建一个“公司”对象的客户端代码。

#### 例8.49 JAX-RPC Web Service客户端代码

```

1 Context ctx = new InitialContext();
2 CompanyWebService service = (CompanyWebService)
3     ctx.lookup("java:comp/env/service/CompanyWebService");
4 CompanyWebServiceProvider provider =
5     service.getCompanyWebServiceProviderPort();
6 provider.createCompany("1001", "Joes Company" );

```

## 效果

- 在客户端和服务之间增加了额外的一个层次。
- 需要对现有的远程会话门面进行重构，使其支持本地访问。
- 由于使用web协议进行通信，可能增加网络的负载。

## 相关模式

- 聚合器 (*Aggregator*)

<http://www.enterpriseintegrationpatterns.com/Aggregator.html>

- 应用服务

可以从*Web Service*中转组件调用应用服务组件。

- 会话门面

可以从*Web Service*中转组件调用会话门面组件。

- 中介 (*Mediator*)

中介[Frank Buschmann, et al.]也可以作为一个交互的中心，类似于*Web Service*中转。

<http://www.vico.org/pages/PatronsDisseny/PatternBroker/>

- 消息路由器 (*Message Router*)

<http://www.enterpriseintegrationpatterns.com/MessageRouter.html>

579



## 尾 声

本章将涉及以下问题:

- Web Worker微架构

580

## Web Worker微架构纵览

在本章中，我们将讨论一个模式领域中的高阶话题：微架构。我们给“微架构”下了这样一个定义：一组被同时使用的模式，用于实现系统中的一个特定部分或一个特定的子系统。在我们看来，微架构就是构造系统的“积木块”：每一个微架构是一个内聚的、相对独立的部分，将它们组合在一起就构成了整个系统的架构。微架构常被用于解决粗粒度、高层面的问题——单个模式面对这类问题往往力不从心。因此，比起“J2EE模式目录”中介绍的单个模式，微架构所代表的抽象层次要更高一筹。可以将微架构看作“将一组模式组合而成的惯用解决方案”。

架构之间的差异就在于其中架构子元素的外在特征，以及这些元素之间的交互。作为一个整体，架构具有概念完整性，各个独立的微架构也一样。

微架构有很多种不同的形式，我们在这里将要介绍的是Web Worker工作流集成微架构。Web Worker关注的不是工作流处理范畴以内的模式，而是如何将J2EE系统与工作流系统结合起来。

### 工作流简介

工作流管理联盟（WfMC）将工作流定义为：

“业务处理的（完全或部分）自动化运转，文档、信息或任务按照一系列程序化的任务规定在其中不同的参与者之间传递，以完成各步骤的业务操作。”

工作流系统由工作流过程组成，后者又包含了行为、决策点和子过程等元素。行为（activity）将生成工作项（work item），完成一个工作项的方式有两种：

- 在手工工作流过程中，工作项将被交给具有适当角色的用户。用户隐式或显式地接受（获取）工作项，并对其进行手工处理。
- 在自动工作流过程中，工作项将由系统而非人来完成。工作项完成之后，用户或系统会将其提交给工作流系统。

组成工作流系统的元素包括：

- 工作流定义
- 工作流过程
- 行为
- 决策点
- 工作项
- 工作列表

下面是一个简单的例子：“烤蛋糕”的工作流定义。这个工作流是由下列行为和决策点组成的（请注意：在这个例子中，只有当前一个行为结束之后，后一个行为才能开始）：

- 1) 开始工作流过程
- 2) 将原料倒进碗里——行为
- 3) 将原料混合搅拌成糊状——行为
- 4) 将糊状物倒进烤盘——行为

- 5) 打开烤箱——行为
- 6) 在烤箱里烤蛋糕——行为
- 7) 蛋糕烤好了吗? ——决策点。是: 转到行为8; 否: 转到行为6
- 8) 将蛋糕取出烤箱——行为
- 9) 等待10分钟, 让蛋糕变凉——行为
- 10) 把蛋糕端上桌——行为
- 11) 结束工作流过程

上面的“烤蛋糕”工作流定义描述了烤一个蛋糕所需的工作流, 它与具体要烘烤的蛋糕种类无关。当“烤蛋糕”工作流定义执行起来之后, 运行中的实例被称为“工作流过程”。可以同时运行三个“烤蛋糕”工作流过程, 分别是:

- 烤一个巧克力蛋糕,
- 烤一个奶油蛋糕,
- 烤一个芝士蛋糕。

现在, 我们来看一个更为复杂的例子——招聘一名员工。图9-1所示的工作流定义叫做“招聘员工”。[582]

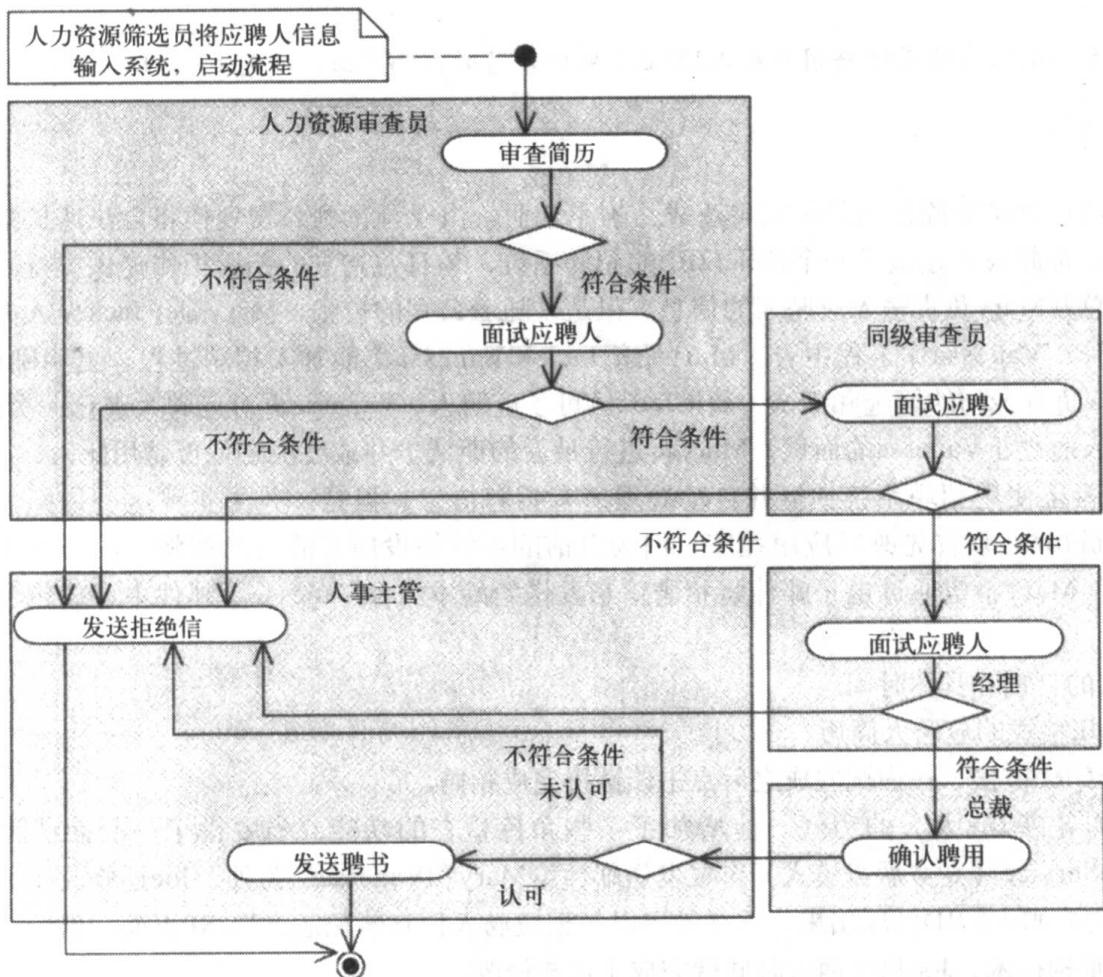


图9-1 “招聘员工”工作流

当人力资源筛选员（HR Screener）将应聘人信息输入系统时，工作流过程就开始运转了。首先被启动的是“审查简历”行为，人力资源审查员（HR Reviewer）会对应聘人的简历进行审查，并提交一个工作项，判断应聘人是否符合招聘条件。

如果应聘人不符合条件，启动“发送拒绝信”行为，全过程结束。反之，则启动“面试应聘人”行为，同级审查员（Peer Reviewer）将对应聘人进行面试。

**583** 面试之后，同级审查员提交工作项，判断应聘人是否符合条件。如果符合，启动第二个“面试应聘人”行为，经理（Manager）将对应聘人进行面试。

第二次面试之后，经理将提交工作项，决定是否希望聘用这名应聘人。如果希望聘用这个人，就将启动“确认聘用”行为，总裁（President）必须对聘用决定加以确认。

如果总裁认可了经理的招聘决定，人事主管（HR Director）将发送一份聘书；如果总裁不认可，人事主管将发送拒绝信。

最后，工作流过程结束。

## Web Worker微架构

### 问题

需要用一个工作流系统将用户引入J2EE应用中合适的Web页面，以完成他们的工作。

### 问题故事

拉斯伯利离岸资源公司是一家飞速成长的小企业。由于预见到公司规模将会快速扩张，拉斯伯利离岸资源公司建造了一个基于J2EE的招聘系统，为日益增加的招聘工作提供支持。一开始，只有总裁Mary负责输入应聘人的信息。但是，随着公司的扩张，Mary雇了Jack作人力资源（HR）主管、Vanessa作工程主管。Mary希望Jack和Vanessa都能加入招聘过程。更明确地说，她希望Jack负责最初的筛选和面试。如果Jack认可了应聘人，Vanessa再对应聘人进行一次面试。如果应聘人通过了Vanessa的面试，Mary再进行最后的面试，并最终决定是否聘用此人。

Mary希望使用招聘系统捕捉并自动处理所有招聘信息。但是，为了实现这一目标，公司的开发人员Joe必须首先改写应用程序，因为以前的系统假设所有的信息都将由同一个人来输入。另外，Mary希望通过电子邮件将招聘信息发送给她本人和Vanessa，邮件中应该包含下列信息：

- 面试的日期和具体时间。
- HTML格式的应聘人简历。
- 一个URL链接，在面试完成之后点击该链接完成招聘。

**584** 这没有花费Joe太大的力气。他增加了一些角色检查的功能；当安排了一次面试时，让HiringAppService（业务服务模式）对象发送邮件给Mary和Vanessa。另外，Joe还修改了一些表现代码，将应聘人的ID抽取出来，这样就可以根据应聘人信息装配适当的JSP页面。借助一些常用的J2EE重构技术，Joe只用两天时间就完成了这些修改。

第二年，拉斯伯利离岸资源公司飞快地成长。现在，又有两名人力资源筛选员和三名工程

主管涉及到招聘过程中，Joe几乎每天都会收到新的需求——什么人在什么时间可以输入什么信息。终于，他意识到：在过去的一年中，他已经在业务逻辑之外加入了极其大量的过程逻辑，这些过程逻辑已经变得极其复杂了。

一开始，基本的招聘过程逻辑非常简单：

- 1) 如果Jack认可应聘人，Vanessa对应聘人进行面试。
- 2) 如果Vanessa认可应聘人，Mary对应聘人进行面试。
- 3) 如果Mary认可应聘人，Jack向应聘人发送聘书。

但是，由于面试过程的参与者越来越多，现在的招聘过程有太多的步骤和分支，代码已经变成了一团乱麻，完全无法管理。而且，以前每个角色只有一个人，而现在每个角色可能对应多个人。

于是，Joe说服了Vanessa购买一个工作流系统来运行业务逻辑。这个系统让Joe的生活轻松多了：他把动态的过程逻辑放进工作流系统中，对过程逻辑的修改不会触及招聘系统的业务逻辑。

现在，Joe只剩下一个问题要解决了：如何将基于J2EE的招聘系统与工作流系统集成，并达到与以前一样的效果。他首先让自己熟悉了用户、Web应用和工作流系统三者之间的交互（如图9-2所示）。其中的步骤包括：

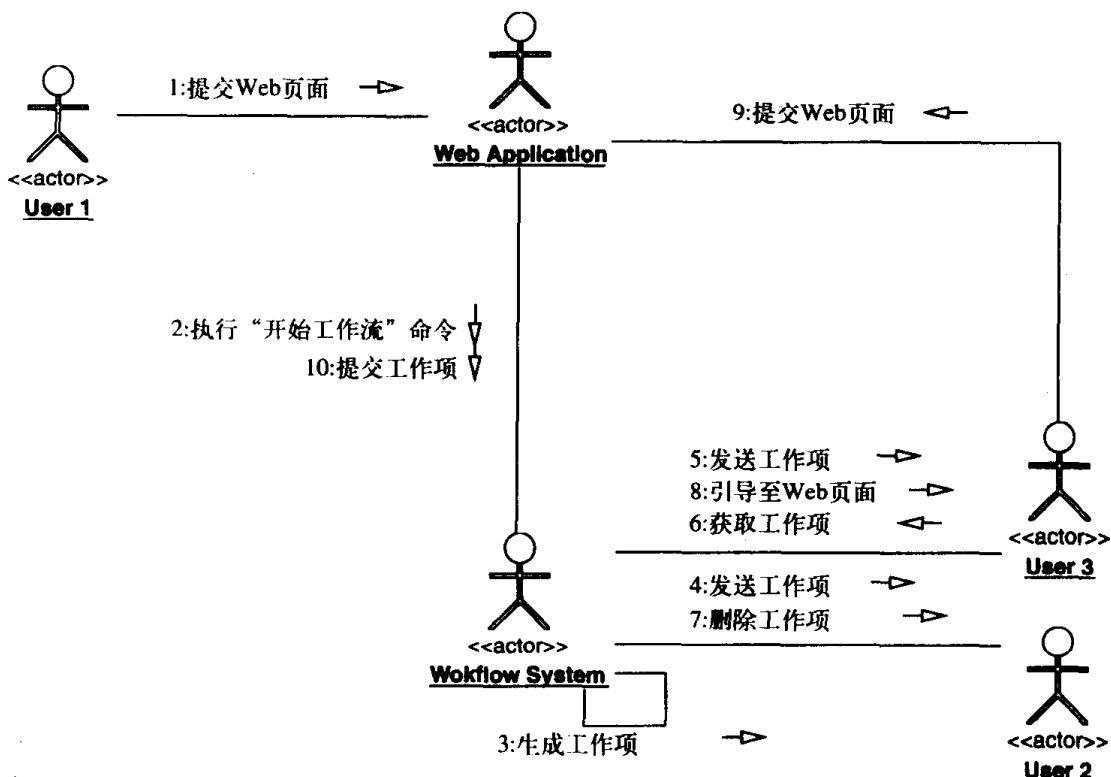


图9-2 工作流交互的例子

步骤1：用户1向Web应用提交一个Web页面。

步骤2：Web应用将业务事件翻译成工作流命令（例如“开始工作流”），并执行该命令。工作流命令可能包括开始、暂停或停止一个工作流过程；或者获取、释放或提交一个工作项。

步骤3：工作流系统生成工作项。

步骤4和5：将工作项发送给具有适当角色的所有用户（工作列表）。

585

- 步骤6：用户3通过工作列表获取工作项。
- 步骤7：从用户2的工作列表中删除该工作项。
- 步骤8：用户3被引导至一个Web页面。
- 步骤9：该项工作完成之后，用户3提交Web页面。
- 步骤10：工作项被提交。

在这个例子中，Web应用和工作流系统之间的集成多少显得有些神奇，特别是：

- Web页面的提交动作如何被转化为工作流命令？
- 工作流系统如何将用户引导至恰当的Web页面？
- 作为对已获取的工作项的回应，Web页面的提交动作是如何被转化为工作项的提交动作的？

586

这些问题正是*Web Worker*要解决的。

## 约束

- 面临明确的、不断变化的业务过程逻辑。
- 业务过程逻辑非常复杂，可以作为一个单独的子系统来运行。
- 有以用户为中心的业务处理工作（工作项）。
- 工作流定义表现为依次出现和/或并列出现的行为。
- 过程逻辑需要与用户相协调。
- 希望用户从工作列表中获取工作，并通过Web应用提交工作。

## 解决方案

### 使用*Web Worker*集成用户、Web应用和工作流系统。

*Web Worker*主要关注两个结合点：Web应用和工作流系统之间的结合点，以及工作流系统和用户之间的结合点。由于工作流系统被作为黑盒对待，因此只要在工作流系统和Web应用之间使用一个对内适配器（称为*Action Adapter*，操作适配器），在工作流系统、用户和Web应用三者之间使用一个对外适配器（称为*Work Adapter*，工作适配器），就可以完成集成。

**备注1：***Action Adapter*位于集成层，并被业务层调用。表现层不调用*Action Adapter*。

**备注2：***Work Adapter*只负责将用户引导至最初的Web页面，而不负责引导用户到完成工作

587

所需的所有Web页面。

图9-3展示了使用了*Action Adapter*和*Work Adapter*之后系统之间的交互情况，依次是：

- 步骤1：用户将Web页面提交给Web应用。
- 步骤2：Web应用发送应用程序事件给*Action Adapter*。
- 步骤3：*Action Adapter*发送工作流命令给工作流系统。
- 步骤4：工作流系统发送工作项至工作列表。
- 步骤5：用户从工作列表获取工作项。
- 步骤6：工作流系统发送工作流事件给*Work Adapter*。
- 步骤7：*Work Adapter*将用户引导至工作页面。
- 步骤8：用户手工完成工作，并提交工作页面。

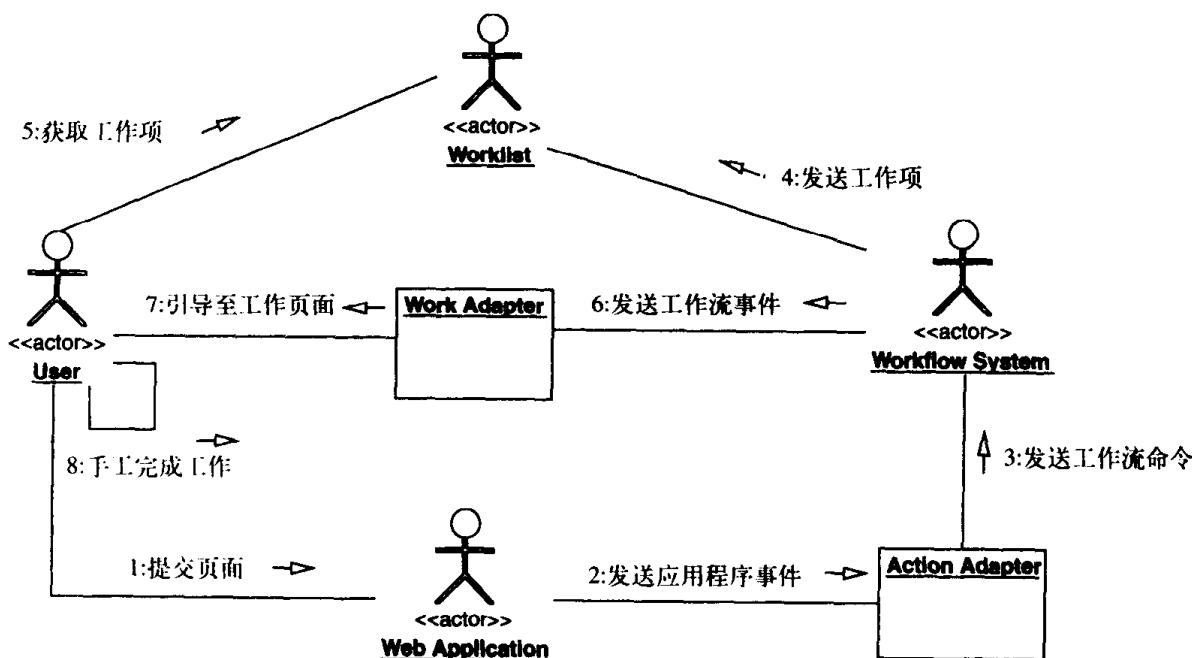


图9-3 Action Adapter和Work Adapter协作图

当用户完成了需要手工进行的工作，并提交工作页面之后，重复上述过程。不过，在本例中，对页面的提交动作将最终被转化为对工作项的提交动作。

588

图9-4展示了“招聘员工”工作流中的交互情况。这个工作流包含下列步骤：

步骤1：人力资源筛选员将新的应聘人信息添加到Web应用。

步骤2：Web应用发送“新增应聘人”应用程序事件到Action Adapter。

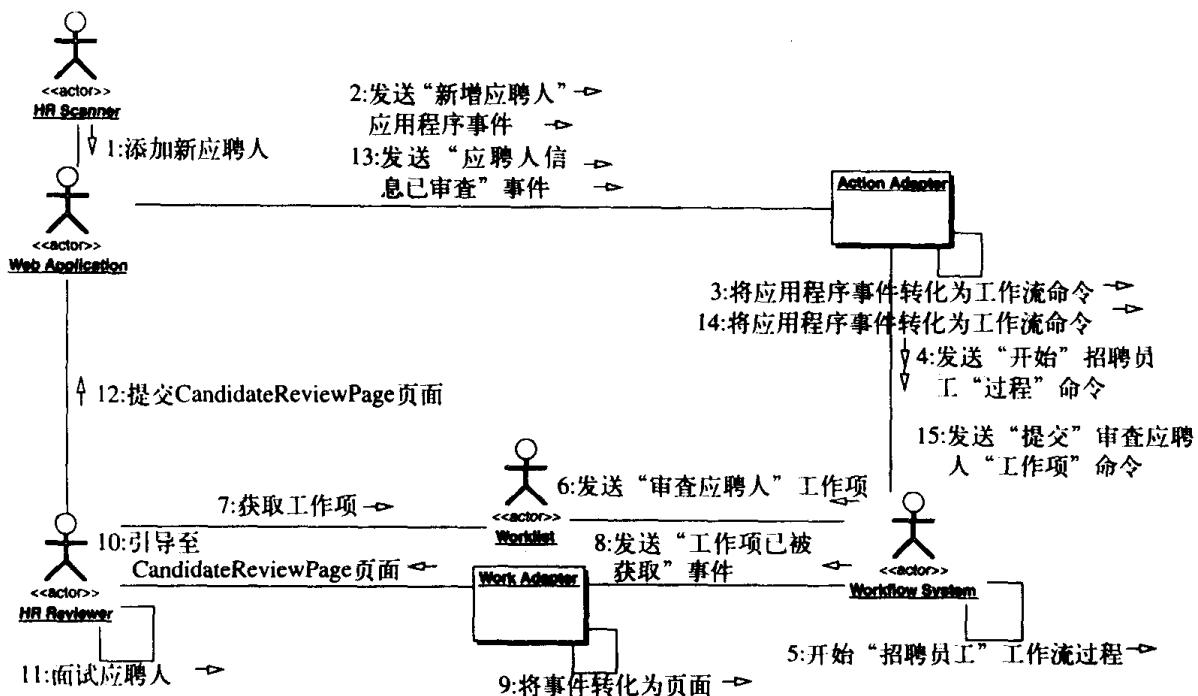


图9-4 “招聘员工”工作流的交互情况（使用了Action Adapter和Work Adapter）

步骤3: Action Adapter将应用程序事件转化为工作流命令 (“开始‘招聘员工’过程”命令)。

步骤4: Action Adapter发送“开始‘招聘员工’过程”命令。

步骤5: 工作流系统开始“招聘员工”工作流过程。

步骤6: 工作流系统发送“审查应聘人”工作项到工作列表。

步骤7: 人力资源审查员获取该工作项。

步骤8: 工作流系统发送“工作项已被获取”事件到Work Adapter。

589 步骤9: Work Adapter将“工作项已被获取”事件转化为CandidateReviewPage页面。

步骤10: Work Adapter将人力资源审查员引导至CandidateReviewPage页面。

步骤11: 人力资源审查员面试应聘人。

步骤12: 人力资源审查员提交CandidateReviewPage页面给web应用。

步骤13: Web应用发送“应聘人信息已审查”事件给Action Adapter。

步骤14: Action Adapter将应用程序事件转化为工作流命令 (“提交‘审查应聘人’工作项”命令)。

590 步骤15: Action Adapter发送“提交‘审查应聘人’工作项”命令给工作流系统。

## 结构

### Action Adapter (操作适配器)

如图9-5所示, Action Adapter组件由下列模式和类组成:

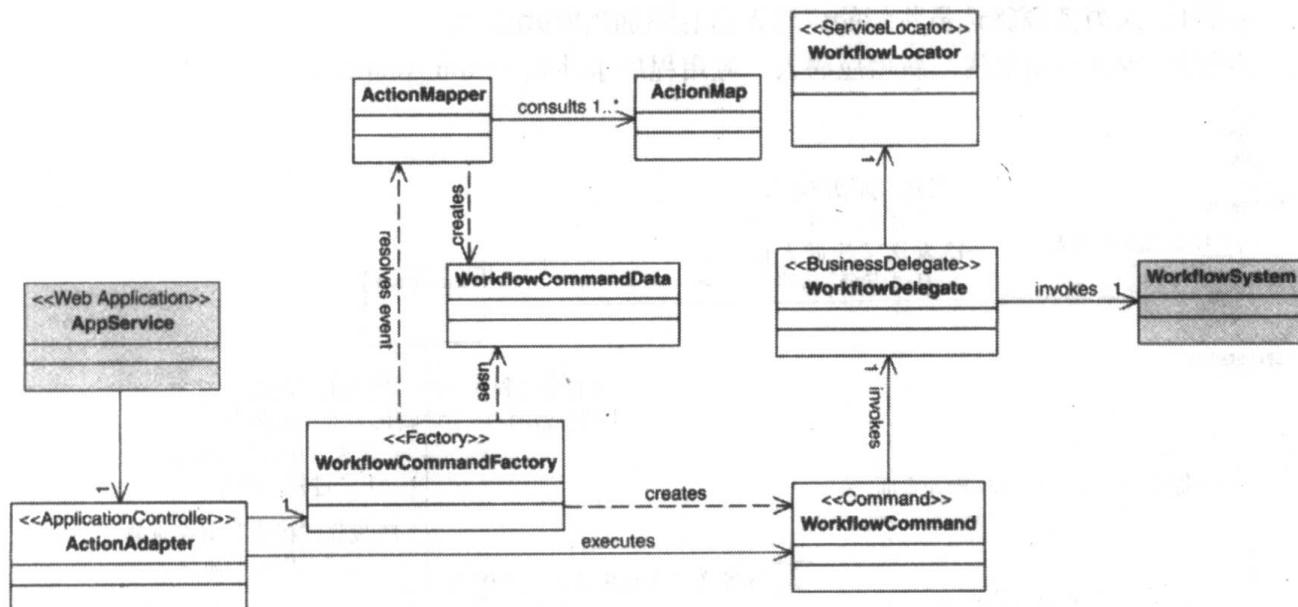


图9-5 Action Adapter类图

### 应用控制器模式, 采用命令处理器策略

- ActionAdapter (操作适配器)
- ActionMapper (操作映射器)
- ActionMap (操作表)

- WorkflowCommandFactory (工作流命令工厂)
- WorkflowCommandData (工作流命令数据)
- WorkflowCommand (工作流命令)

### 业务代表

- WorkflowDelegate (工作流代表)

### 服务定位器

- WorkflowLocator (工作流定位器)

591

ActionAdapter具有与AppService (应用服务) 接口对应的、业务相关的接口。比如说, CandidateAppService (应聘人应用服务, 一种AppService) 可能有一个interviewApproved()方法 (即“面试通过”), 那么CandidateActionAdapter (应聘人操作适配器, 一种ActionAdapter) 就必定同样拥有interviewApproved()方法。

ActionAdapter将应用程序事件发送给WorkflowCommandFactory, 以获取代表工作流命令的WorkflowCommand。WorkflowCommandFactory使用ActionMapper将事件转化为WorkflowCommandData对象, 而ActionMapper在执行转化时则需要查阅ActionMap中保存的“事件-命令”映射关系。然后, WorkflowCommandFactory使用WorkflowCommandData对象创建WorkflowCommand对象。

*Web Worker*有两种工作流命令: 工作流过程命令和工作项命令。

- 工作流过程命令由工作流过程的开始、暂停和结束组成。
- 工作项命令由工作项的获取、中止和提交组成。

就工作流过程命令而言, “开始”命令必须传入一个工作流过程定义的名字, 例如“招聘员工作流”。用这个名字, 工作流系统将根据工作流定义新建一个工作流过程实例。另一方面, “暂停”和“结束工作流”过程命令则必须在特定的工作流过程实例上进行。大多数工作流系统会提供一个唯一的工作流过程ID, 这些命令将被施加于工作流过程ID之上。

### Work Adapter (工作适配器)

如图9-6所示, Work Adapter组件由下列模式和类组成:

#### 应用控制器模式, 采用命令处理器策略

- WorkMapper (工作映射器)
- WorkMap (工作表)
- WorkPageFactory (工作页面工厂)
- WorkPageRef (工作页面引用)

### 服务激活器

- WorkAdapter (工作适配器)
- WorkDispatcher (工作分配器)

当工作流事件发生时, WorkAdapter将收到工作流系统的通知。请注意, 有些工作流系统会异步调用一个监听器, 有些则要求监听器轮询工作流系统以获得事件通知。WorkAdapter感兴趣的事件包括:

592

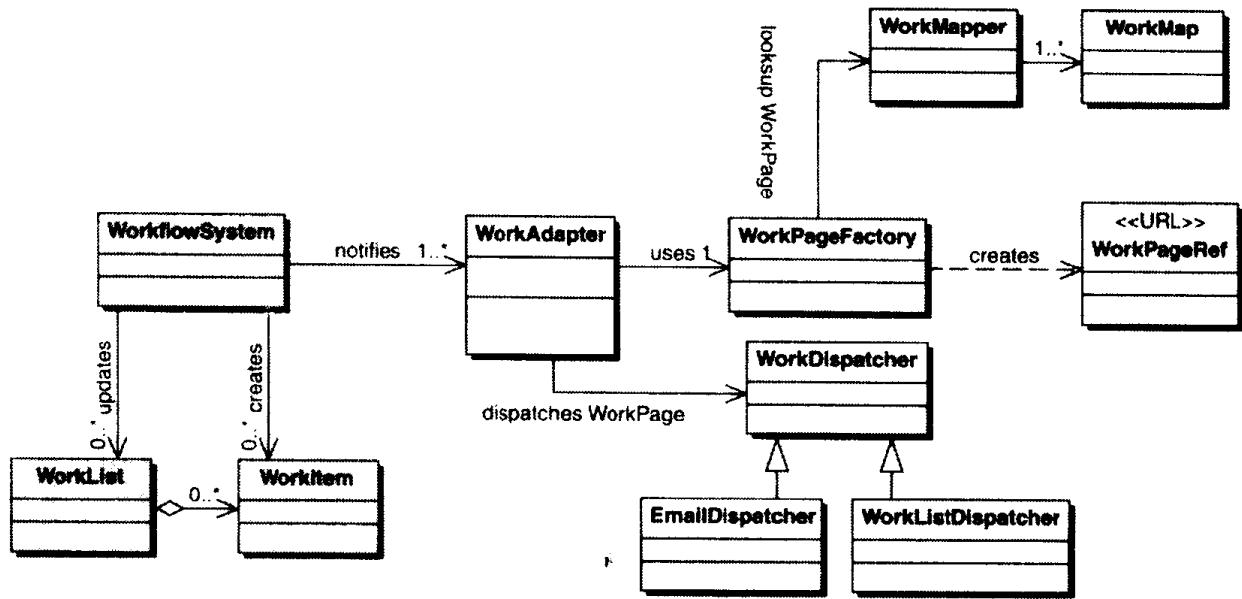


图9-6 Work Adapter类图

- 工作流过程开始
- 工作流过程暂停
- 工作流过程结束
- 工作项被获取
- 工作项被中止
- 工作项被提交
- 工作项过期

如果监听到的工作流事件是“工作项被获取”，WorkAdapter将要求WorkPageFactory创建一个WorkPageRef对象。WorkPageRef对象封装了一个业务Web页面的URL，当工作项处理完成之后，必须提交这个web页面。

WorkAdapter将工作项信息发送给WorkMapper，后者将查阅WorkMap以获得恰当的WorkPage URL。WorkPage是一个Web页面，当工作完成之后，用户将被引导至这个页面。与这个页面相关的信息可以通过电子邮件或其他形式获得。随后，WorkAdapter使用WorkDispatcher将WorkPage的URL告知用户。

图9-6展示了两种典型的WorkDispatcher: EmailDispatcher (Email分配器) 和 WorkListDispatcher (工作列表分配器)。EmailDispatcher将通过电子邮件把WorkPage的URL发送给用户，WorkListDispatcher则会把WorkPage的URL发送到用户的工作列表 (WorkList) 中。

## 参与者及职责 (见例9.1至例9.11)

### Action Adapter

图9-7展示了Action Adapter的交互过程。这幅交互图涵盖了“招聘员工”工作流的前三个事件：

- 创建应聘人对象
- 审查应聘人
- 面试应聘人

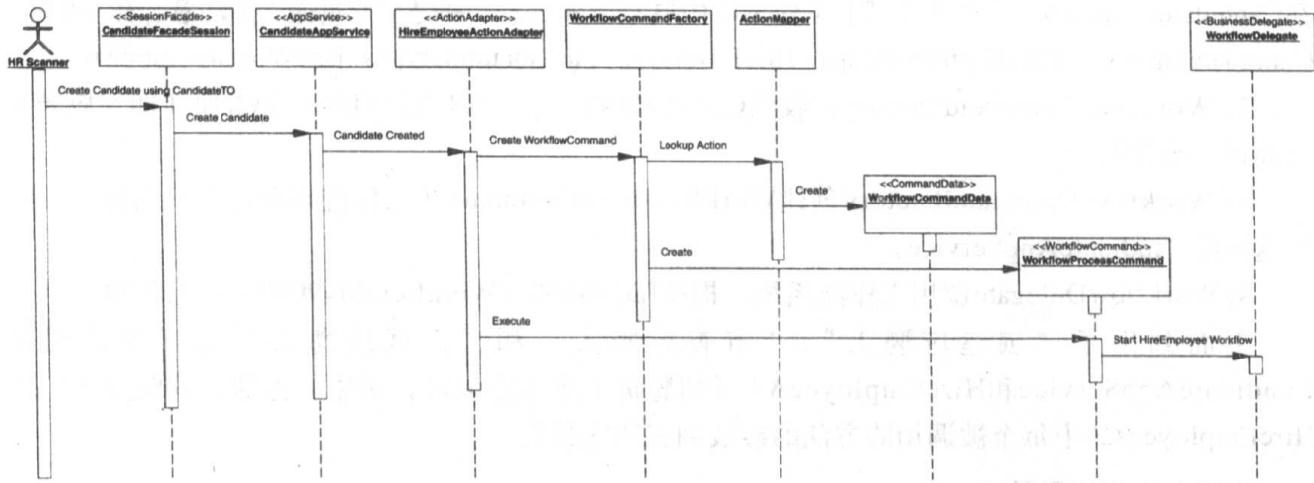


图9-7 Action Adapter序列图 (应聘人的交互)

“创建应聘人”的步骤是：

- 1) CandidateAppService调用HireEmployeeActionAdapter（招聘员工操作适配器）的candidateCreated()方法（即“创建应聘人”方法）。
- 2) HireEmployeeActionAdapter发送CANDIDATE\_CREATED业务事件（即“应聘人已创建”事件）给WorkflowCommandFactory，以获得一个WorkflowCommand对象。
- 3) WorkflowCommandFactory要求ActionMapper提供CommandData对象。
- 4) ActionMapper查阅HireEmployeeActionMap（招聘员工操作表），将业务事件转化为CommandType（命令类型）对象。
- 5) ActionMapper返回一个StartWorkflow（开始工作流）命令给WorkflowCommandFactory。
- 6) WorkflowCommandFactory创建WorkflowProcessCommand（工作流处理命令）对象，并将其返回给CandidateAppservice。
- 7) CandidateAppService要求WorkflowProcessCommand开始执行，这会引发对WorkflowDelegate的调用。
- 8) WorkflowDelegate调用工作流系统，开始“招聘员工”工作流过程。

“审查应聘人”的步骤与“创建应聘人”大致相同，只有以下几处例外：

- 1) 第一个获取工作项的人力资源审查员必须负责审查应聘人的简历。审查完成之后，审查员在CandidateReview（应聘人审查）工作页面中填写审查信息，并提交该页面。请注意，在本例中，CandidateReview工作页面的URL是：HireEmployee/CandidateReview.jsp? WorkItemId=8000。
- 2) WorkflowCommandFactory接收到CANDIDATE\_REVIEWED业务事件（即“应聘人已审查”事件）。
- 3) WorkflowCommandFactory创建WorkflowItemCommand对象，并将其返回给CandidateAppService。

4) WorkflowDelegate调用工作流系统，提交ID为8000（WorkItemId=8000）的工作项。

“面试应聘人”的步骤与“创建应聘人”大致相同，只有以下几处例外：

1) 第一个获取工作项的人力资源审查员必须负责审查应聘人的简历。审查完成之后，审查员在CandidateInterview（应聘人面试）工作页面中填写审查信息，并提交该页面。请注意，在本例中，CandidateInterview工作页面的URL是：HireEmployee/CandidateInterview.jsp?WorkItemId=9000。

2) WorkflowCommandFactory接收到CANDIDATE\_INTERVIEWED业务事件（即“应聘人已面试”事件）。

3) WorkflowCommandFactory创建WorkflowItemCommand（工作流事项命令）对象，并将其返回给CandidateAppService。

4) WorkflowDelegate调用工作流系统，提交ID为9000（WorkItemId=9000）的工作项。

下面列出了“创建应聘人”、“审查应聘人”和“面试应聘人”交互所调用的CandidateAppService和HireEmployeeAA（招聘员工操作适配器）对象的方法，同时还列出了HireEmployeeAA中每个被调用的方法的参数列表和参数值。

### 交互和方法调用信息

#### 创建应聘人

- CandidateAppService: 方法——createCandidate()
- HireEmployeeAA: 方法——candidateCreated()
- HireEmployeeAA: 参数列表——userId, workflowProcessId, workItemId
- HireEmployeeAA: 参数值——User-101, null, null

#### 审查应聘人

- CandidateAppService: 方法——submitReview()（即“提交审查”方法）
- HireEmployeeAA: 方法——candidateReviewed()
- HireEmployeeAA: 参数列表——userId, workflowProcessId, workItemId
- HireEmployeeAA: 参数值——User-102, null, 8000

#### 面试应聘人

- CandidateAppService: 方法——submitInterview()（即“提交面试”方法）
- HireEmployeeAA: 方法——candidateInterviewed()
- HireEmployeeAA: 参数列表——userId, workflowProcessId, workItemId
- HireEmployeeAA: 参数值——User-103, null, 9000

下面将分别针对每次交互，列出传递给ActionMapper.getCommandData()方法的业务事件、HireEmployeeActionMap返回的命令字符串（来自HireEmployeeActionMap.xml文件）、以及ActionMapper返回的CommandData类型。

### 业务事件、命令字符串和返回的类型

#### 创建应聘人

- 传递给ActionMapper的业务事件——CANDIDATE\_CREATED
- HireEmployeeActionMap返回的命令字符串——StartWorkflow

- ActionMapper返回的CommandData类型—— WorkflowProcessData

**审查应聘人**

- 传递给ActionMapper的业务事件——CANDIDATE REVIEWED

- HireEmployeeActionMap返回的命令字符串——CommitWorkItem (即“提交工作项”)

- ActionMapper返回的CommandData类型—— WorkItemData (即“工作项数据”)

**面试应聘人**

- 传递给ActionMapper的业务事件—— CANDIDATE INTERVIEWED

- HireEmployeeActionMap返回的命令字符串——CommitWorkItem

- ActionMapper返回的CommandData类型—— WorkItemData

下面将分别针对每次交互，列出WorkflowCommandFactory返回的WorkflowCommand对象，以及WorkflowCommand.execute()方法被调用时调用的WorkflowDelegate方法。传递给WorkflowDelegate方法的参数列表和参数值也将同时列出。

### WorkflowCommand和WorkflowDelegate操作

**创建应聘人**

- WorkflowCommandFactory返回的WorkflowCommand对象——WorkflowProcessCommand

- 被调用的WorkflowDelegate方法——start()

- 参数列表 ——WorkflowName (即“工作流名称”)

- 参数值——HireEmployee (即“招聘员工”)

**审查应聘人**

- WorkflowCommandFactory返回的WorkflowCommand对象——WorkItemCommand

- 被调用的WorkflowDelegate方法——commitWorkItem()

- 参数列表——WorkItemId

- 参数值——8000

**面试应聘人**

- WorkflowCommandFactory返回的WorkflowCommand对象—— WorkItemCommand

- 被调用的WorkflowDelegate方法——commitWorkItem()

- 参数列表——WorkItemId

- 参数值——9000

### 例9.1 CandidateFacadeSession (应聘人面试会话) 源码

```

1 package ActionAdapter;
2 import javax.ejb.EJBException;
3 import javax.ejb.SessionContext;
4 import java.rmi.RemoteException;
5
6 public class CandidateFacadeSession implements
7     javax.ejb.SessionBean {
8     public void createCandidate(CandidateTO to)
9         throws CandidateException {
10        CandidateAppService as = new CandidateAppService();

```

```

11     as.createCandidate(to.getId(),
12         to.getLname(),
13         to.getFname(),
14         to.getStreet(),
15         to.getCity(),
16         to.getState(),
17         to.getZip(),
18         new WorkflowContext(
19             to.getWorkflowName(), to.getWorkflowProcessId(),
20             to.getWorkItemId() ) );
21     }
22
23     public void submitReview(CandidateTO to,
24         ReviewInfoTO rto ) {
25         CandidateAppService as = new CandidateAppService();
26         as.submitReview(to.getId(), rto,
27             new WorkflowContext(
28                 to.getWorkflowName(), to.getWorkflowProcessId(),
29                 to.getWorkItemId() ) );
30     }
31
32     public void submitInterview(CandidateTO to,
33         InterviewInfoTO ito) throws CandidateException {
34         CandidateAppService as = new CandidateAppService();
35         as.submitInterview(to.getId(), ito,
36             new WorkflowContext(
37                 to.getWorkflowName(), to.getWorkflowProcessId(),
38                 to.getWorkItemId() ) );
39     }
40     public void setSessionContext(SessionContext sessionContext)
41         throws EJBException, RemoteException { }
42
43     public void ejbRemove() throws EJBException,
44         RemoteException { }
45
46     public void ejbActivate() throws EJBException,
47         RemoteException { }
48
49     public void ejbPassivate() throws EJBException,
50         RemoteException { }
51 }

```

599

### 例9.2 HireEmployeeAA (招聘员工操作适配器) 源码

```

1 package ActionAdapter;
2
3 import Core.HireEmployeeConstants;
4
5 public class HireEmployeeAA {

```

```

6     WorkflowCommandFactory factory;
7
8     public HireEmployeeAA() {
9         factory = WorkflowCommandFactory.getInstance();
10    }
11
12    public void candidateCreated(String userId,
13        String workflowProcessId, String workItemId) {
14        doCommand(userId, workflowProcessId, workItemId,
15            HireEmployeeConstants.CANDIDATE_CREATED);
16    }
17
18    public void candidateInterviewed(String userId,
19        String workflowProcessId, String workItemId) {
20        doCommand(userId, workflowProcessId, workItemId,
21            HireEmployeeConstants.CANDIDATE_INTERVIEWED);
22    }
23
24    public void candidateReviewed(String userId,
25        String workflowProcessId, String workItemId) {
26
27        doCommand(userId, workflowProcessId, workItemId,
28            HireEmployeeConstants.CANDIDATE_REVIEWED);
29    }
30
31    private void doCommand(String userId,
32        String workflowProcessId, String workItemId,
33        String action) {
34        WorkflowCommand command =
35        factory.createCommand( userId, workflowProcessId,
36            workItemId, action );
37        command.execute();
38    }
39 }

```

600

### 例9.3 HireEmployeeConstants (招聘员工常数) 源码

```

1 package Core;
2
3 public class HireEmployeeConstants {
4     final public static int START_WORKFLOW_TYPE = 1;
5     final public static int STOP_WORKFLOW_TYPE = 2;
6     final public static int PAUSE_WORKFLOW_TYPE = 3;
7
8     final public static int ACQUIRE_WORKITEM_TYPE = 4;
9     final public static int COMMIT_WORKITEM_TYPE = 5;
10    final public static int ABORT_WORKITEM_TYPE = 6;
11
12    final public static String START_WORKFLOW = "StartWorkflow";

```

```

13     final public static String STOP_WORKFLOW = "StopWorkflow";
14     final public static String PAUSE_WORKFLOW = "PauseWorkflow";
15
16     final public static String ACQUIRE_WORKITEM =
17         "AcquireWorkItem";
18     final public static String COMMIT_WORKITEM =
19         "CommitWorkItem";
20     final public static String ABORT_WORKITEM = "AbortWorkItem";
21     final public static String CANDIDATE_CREATED =
22         "CandidateCreated";
23     final public static String CANDIDATE_REVIEWED =
24         "CandidateReviewed";
25     final public static String CANDIDATE_INTERVIEWED =
26         "CandidateInterviewed";
27 }

```

601

#### 例9.4 ActionMapper (操作映射器) 源码

```

1 package ActionAdapter;
2
3 import Core.HireEmployeeConstants;
4 import java.util.Properties;
5
6 public class ActionMapper {
7     String workflowName = "HireEmployee";
8     String mapFile;
9
10    public ActionMapper(String mapFile) {
11        this.mapFile = mapFile;
12    }
13
14    public CommandData getCommandData(String action,
15        String workflowProcessId, String workItemId) {
16        // 按照操作 (Action) 从mapFile中找到真正的commandString (命令字符串)
17        String commandString = "";
18        // 从mapFile中获得Action (操作) 的属性
19        Properties properties = null;
20        CommandData data = null;
21        if(commandString.equals(
22            HireEmployeeConstants.START_WORKFLOW)) {
23            data = new WorkflowCommandData(workflowName,
24                workflowProcessId, HireEmployeeConstants.
25                START_WORKFLOW_TYPE);
26        } else if (commandString.equals(
27            HireEmployeeConstants.STOP_WORKFLOW)) {
28            data = new WorkflowCommandData(workflowName,
29                workflowProcessId, HireEmployeeConstants.
30                STOP_WORKFLOW_TYPE);
31        } else if(commandString.equals(

```

```

32         HireEmployeeConstants.PAUSE_WORKFLOW)) {
33         data = new WorkflowCommandData(workflowName,
34             workflowProcessId, HireEmployeeConstants.
35             PAUSE_WORKFLOW_TYPE);
36     } else if(commandString.equals(
37         HireEmployeeConstants.ACQUIRE_WORKITEM)) {
38         data = new WorkItemCommandData(workItemId,
39             HireEmployeeConstants.ACQUIRE_WORKITEM_TYPE);
40     } else if(commandString.equals(
41         HireEmployeeConstants.ABORT_WORKITEM)) {
42         data = new WorkItemCommandData(workItemId,
43             HireEmployeeConstants.ABORT_WORKITEM_TYPE);
44     } else {
45         data = new WorkItemCommandData(workItemId,
46             HireEmployeeConstants.COMMIT_WORKITEM_TYPE);
47     }
48     return data;
49 }
50 }
```

#### 例9.5 HireEmployeeActionMap.xml文件（即招聘员工操作表）

```

1 <ActionMap>
2   <Action Name="CandidateCreated">
3     <WorkerRole>HR Screener</WorkerRole>
4     <WorkflowProcessCommand>StartWorkflow
5     </WorkflowProcessCommand>
6     <WorkflowName>HireEmployee</WorkflowName>
7   </Action>
8   <Action Name="ReviewResume">
9     <WorkerRole>HR Reviewer</WorkerRole>
10    <WorkItemCommand>CommitWorkItem</WorkItemCommand>
11  </Action>
12  <Action Name="CandidateInterviewed">
13    <WorkerRole>HR Reviewer</WorkerRole>
14    <WorkItemCommand>CommitWorkItem</WorkItemCommand>
15  </Action>
16 </ActionMap>
```

#### 例9.6 CommandData（命令数据）、WorkflowCommandData（工作流命令数据）和WorkItemCommandData源码（工作项命令数据）

```

1 package ActionAdapter;
2
3 abstract public class CommandData {
4
5     protected int commandType;
6
7     protected CommandData(int commandType) {
8         this.commandType = commandType;
```

```

9     }
10
11    public int getCommandType() {
12        return commandType;
13    }
14 }

1 package ActionAdapter;
2 public class WorkflowCommandData extends CommandData {
3     private String workflowName;
4     private String workflowProcessId;
5
6     public WorkflowCommandData(String workflowName,
7         String workflowProcessId, int commandType) {
8         super(commandType);
9         this.workflowName = workflowName;
10        this.workflowProcessId = workflowProcessId;
11    }
12
13    public void setWorkflowName(String workflowName) {
14        this.workflowName = workflowName;
15    }
16
17    public void setWorkflowProcessid(String workflowProcessId) {
18        this.workflowProcessId = workflowProcessId;
19    }
20
21    public String getWorkflowProcessId() {
22        return workflowProcessId;
23    }
24
25    public String getWorkflowName() {
26        return workflowName;
27    }
28 }
29

1 package ActionAdapter;
2
3 public class WorkItemCommandData extends CommandData {
4     private String workItemId;
5
6     public WorkItemCommandData(String workItemId,
7         int commandType) {
8         super(commandType);
9         this.workItemId = workItemId;
10    }

```