TURING 图灵程序设计丛书 Java系列

MANNING

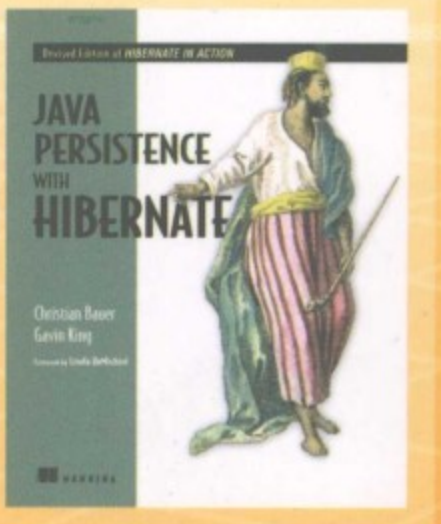
Java Persistence with Hibernate

Hibernate实战（第2版）

[德] Christian Bauer
[澳] Gavin King 著

杨春花 彭永康 俞黎敏 译

- Hibernate之父亲自执笔
- 无可替代的Hibernate和ORM技术圣经
- 职业Java程序员必备秘籍



人民邮电出版社
POSTS & TELECOM PRESS

“Hibernate对Java Persistence乃至EJB 3.0的技术方向产生了巨大影响……本书必将成为广大开发人员的知识宝库。”

——Linda Demichiel，Sun公司EJB 3.0和Java Persistence标准负责人

“本书是终极解决方案。如果你是Hibernate开发人员，不要犹豫了，到书店去，买上一本！”

——JavaLobby

Java Persistence with Hibernate

Hibernate实战（第2版）

持久化是现代企业级应用的核心特性。作为Java世界最流行的持久化框架，Hibernate已经成为开发Java EE应用系统中不可替代的支柱之一。随着EJB 3.0和Java Persistence标准的发布，Hibernate也迎来了更加成熟的新版本——Hibernate 3.2。新版本实现了Java Persistence标准，既可以用于流行的开源技术组合（如Spring、Struts和Seam等），也可以与其他EJB 3.0组件和Java EE服务集成。

本书是毋庸置疑的Hibernate和ORM（对象/关系映射）权威著作，由包括Hibernate之父在内的两位核心开发人员亲自执笔，详细讲述了Hibernate 3.2、Java Persistence和EJB 3.0标准。

本书通过一个应用将数百个例子融合起来，不仅深入剖析了Hibernate 3.2和Java Persistence丰富的编程模型，还深刻阐述了数据库设计、ORM和优化等方面的基本原则、策略和最佳实践。书中处处闪烁着作者超凡的真知灼见，将大大升华读者对ORM乃至企业级应用开发的理解。

本书英文影印版同时可供，ISBN 978-7-115-16061-4，定价89元。



Christian Bauer Hibernate核心开发人员，并负责维护Hibernate的文档与网站。目前他是Red Hat公司JBoss部门负责Hibernate、EJB 3.0和Seam的产品经理。



Gavin King Hibernate项目的创始人和主要开发人员，目前是Red Hat公司Hibernate和Seam项目的负责人。他是EJB 3.0 (JSR 220)专家组成员，对Java Persistence标准以及整个EJB 3.0标准的制定都产生了巨大影响。他还是雄心勃勃的Web Beans JSR 299标准项目的负责人。

图灵Java图书阅读路线图



MANNING

本书相关信息请访问：图灵网站 http://www.turingbook.com
读者/作者热线：(010)88593802
反馈/投稿/推荐信箱：contact@turingbook.com

分类建议 计算机/Java

人民邮电出版社网址：www.ptpress.com.cn

ISBN 978-7-115-17448-2

9 787115 174482 >

ISBN 978-7-115-17448-2/TP

定价：99.00 元

TURING 图灵程序设计丛书 Java系列

Java Persistence with Hibernate

Hibernate实战（第2版）

[德] Christian Bauer
[澳] Gavin King 著
杨春花 彭永康 俞黎敏 译

人民邮电出版社
北 京

图书在版编目（CIP）数据

Hibernate 实战：第 2 版 /（德）鲍尔（Bauer, C.），（澳）金（King, G.）著；杨春花，彭永康，俞黎敏译．
—北京：人民邮电出版社，2008.4
（图灵程序设计丛书）
ISBN 978-7-115-17448-2

Ⅰ. H… Ⅱ. ①鲍… ②金…③杨…④彭…⑤俞…
Ⅲ. JAVA 语言－程序设计 Ⅳ. TP312

中国版本图书馆 CIP 数据核字（2008）第 000463 号

内 容 提 要

本书全面讲述了 Hibernate 和 Java Persistence，并对两者进行了详细的对比分析。书中通过一个应用将数百个例子融合起来，不仅深入剖析了 Hibernate 3.2 和 Java Persistence 丰富的编程模型，还深刻阐述了数据库设计、对象/关系映射（ORM）和优化等方面的基本原则、策略和最佳实践。书中处处闪烁着作者超凡的真知灼见，将大大升华读者对 ORM 乃至企业级应用开发的理解。

本书适合广大的开发人员，从 ORM 的初学者到经验丰富的开发人员。

图灵程序设计丛书

Hibernate 实战（第 2 版）

◆ 著 [德] Christian Bauer [澳]Gavin King
译 杨春花 彭永康 俞黎敏
责任编辑 傅志红 张继发

◆ 人民邮电出版社出版发行 北京市崇文区夕照寺街 14 号
邮编 100061 电子函件 315@ptpress.com.cn
网址 http://www.ptpress.com.cn
三河市海波印务有限公司印刷
新华书店总店北京发行所经销

◆ 开本：800×1000 1/16
印张：40
字数：945 千字 2008 年 4 月第 1 版
印数：1－5 000 册 2008 年 4 月河北第 1 次印刷
著作权合同登记号 图字：01-2007-3048 号
ISBN 978-7-115-17448-2/TP

定价：99.00 元

读者服务热线：(010)88593802 印装质量热线：(010)67129223
反盗版热线：(010)67171154

版 权 声 明

Original English language edition, entitled *Java Persistence with Hibernate* by Christian Bauer and Gavin King, published by Manning Publications Co., 209 Bruce Park Avenue, Greenwich, CT 06830. Copyright © 2007 by Manning Publications Co.

Simplified Chinese-language edition copyright © 2008 by Posts & Telecom Press. All rights reserved.

本书中文简体字版由 Manning Publications Co.授权人民邮电出版社独家出版。未经出版者书面许可，不得以任何方式复制或抄袭本书内容。

版权所有，侵权必究。

对本书的赞誉

“Hibernate对Java Persistence乃至EJB 3.0的技术方向产生了巨大影响……本书必将成为广大开发人员的知识宝库。”

——Linda Demichiel，Sun公司EJB 3.0和Java Persistence标准负责人

“本书从实践出发，将概念和应用结合。……针对你的项目，本书重在讲述方法……相信你一定可以从中看到无数美妙的风景。”

——杨戈（Yanger），Spring中文论坛创始人

“本书是终极解决方案。如果你是Hibernate开发人员，不要犹豫了，到书店去，买上一本！”

——JavaLobby

“本书是结合了最新的JPA标准的Hibernate指南，既详细讲述JPA标准的方方面面，又讲述Hibernate框架的各个重要的细节用法，不但对于使用Hibernate的开发人员来是不可或缺的参考书。”

——范凯（Robbin），JavaEye网站创始人

“本书是使用ORM实现Java持久化最全面和最权威的著作！”

——Will Wagers，C#online.net

“本书让人大呼过瘾。对于我来讲，可能第三章就已经值回书价了。”

——豆瓣评论

“我衷心推荐本书！”

——Stuart Caborn，Thoughtworks公司资深顾问

“关于Hibernate的权威之作，适合任何开发人员。”

——Patrick Peak，BrowserMedia公司CTO，*Hibernate Quickly*一书作者

推 荐 序 一

Hibernate持久化框架已经诞生6年了，在这期间，Hibernate从当初不为人知的开源软件，到现在成为Java编程语言在操作关系数据库方面的持久化框架的统治者，不能不说是一个奇迹。

Java的持久化操作最早要追溯到1996年发布的JDBC接口。当时，JDBC是一个借鉴了ODBC标准，采用纯Java编写的数据库持久化接口。因为JDBC可以用同样的方式存取大多数主流关系数据库，可以做到操作系统无关、数据库无关、纯Java编写，且具备良好的可移植性，因此JDBC发布之后获得了极大的成功，时至今日，它还是Java程序员最常用的API之一。但是对程序员来说，JDBC API显得非常笨拙，如果数据库字段比较多，编写起来非常麻烦，代码量很大，而且不容易维护。因此，为了简化JDBC编程，随后又出现了很多基于JDBC的持久化框架，而Hibernate正是其中的佼佼者。

在Hibernate出现之前，最主流的持久化框架是1999年发布的EJB 1.1的实体bean，特别是在2001年EJB 2.0规范引入本地接口之后，一直到2003年，实体bean变得非常流行，成为应用服务器厂商力推的产品卖点之一。这段时间Java企业应用开发言必称EJB，好像不用EJB都显得不入流，你都不好意思说自己是Java高级程序员。但是，实体bean在持久化方面并没有起到积极的作用，反而极大增加了持久化代码的复杂度，因而逐渐被越来越多的开发人员反对和抛弃。

2001年，澳大利亚墨尔本一位名为Gavin King的27岁Java程序员，上街买了一本SQL编程的书，他厌倦了实体bean，认为自己可以开发出一个符合对象关系映射理论，并且真正好用的Java持久层框架，因此他需要先好好学习一下SQL。这一年的11月，Hibernate的第一个版本发布了。在整个2002年，实体bean还是万众瞩目的焦点，但是已经有很多人开始关注和使用Hibernate了。

2003年9月，Hibernate开发团队进入JBoss公司，开始全职开发Hibernate，从这个时候开始，Hibernate得到了突飞猛进的普及和发展。2004年，整个Java社区开始从实体bean向Hibernate转移，特别是在Rod Johnson的著作*Expert One-on-One J2EE Development without EJB*出版后，由于这本书以扎实的理论、充分的论据和翔实的论述否定了EJB，提出了轻量级敏捷开发理念之后，以Hibernate和Spring为代表的轻量级开源框架开始成为Java世界的主流和事实标准，Hibernate开始大红大紫。在2004年Sun领导的J2EE 5.0标准制定委员会当中，Gavin King已经是实体bean下一代标准的主要制定者，而J2EE 5.0标准当中的持久化框架标准正是以Hibernate为蓝本的。当2006年J2EE 5.0标准正式发布以后，持久化框架标准Java Persistent API（简称JPA）基本上是参考Hibernate实现的，而Hibernate从3.2版本开始，已经完全兼容JPA标准，成为最主流的JPA框架。

我是在2003年3月知道Hibernate框架的，当时我正在寻找一个可以替代实体bean的Java持久层框架，在考察了众多开源项目之后，发现Hibernate正是可以满足我需求的框架，因此在随后的几

个项目当中使用了Hibernate，并且获得了比较满意的效果。在2003年，虽然Hibernate在国外已经是非常有名气的Java开源框架，但是企业应用开发的主流仍然是EJB，因此国内的Java社区非常排斥这种既非J2EE官方标准，又没有大厂商支持的“草根”框架Hibernate。由于找不到一个可以交流Hibernate编程经验的地方，我萌生了自己创建一个Hibernate交流社区的想法，并在2003年9月11日凌晨正式开通了公共交流论坛，这就是JavaEye网站诞生的由来。JavaEye在2003年是国内唯一交流Hibernate的技术网站，不夸张地说，为国内推广和普及Hibernate立下了汗马功劳，影响了一大批Java开发人员。时至今日，JavaEye网站已经成为国内仅次于CSDN的第二大软件开发者交流社区。虽然JavaEye早已经不再是一个单纯的Hibernate技术讨论区，但是当初如果没有Hibernate的缘故，今天就不会有JavaEye网站了。

根据我四年多的Hibernate编程经验，Hibernate并不是一个非常易学易用的框架，它的掌握难度要远远高于同样十分流行的Spring框架。由于对象的持久化操作本身就包含非常多的概念和难点，运用不善还很容易导致性能低下，因此尽管现在Hibernate在国内已经非常普及和流行了，几乎每个软件公司在招聘Java程序员的时候都要求掌握Hibernate，但实事求是地讲，能把Hibernate用好的Java程序员凤毛麟角，在JavaEye网站上对Hibernate的置疑声音也从来就没有停歇过。其实善于使用Hibernate，能把Hibernate用好的案例也有不少，比如Google公司就在一些非常大规模的项目当中使用了Hibernate和WebWork框架（有传言说是Adwords项目，但我没有考证过），并且贡献了基于Hibernate的Hibernate Shards开源项目和从WebWork项目当中抽取出来的Google Guice开源项目。

因此要想把Hibernate用好，对于Hibernate框架的很多重要的运用细节需要非常的了解和熟悉，除了在项目当中摸索和总结Hibernate使用经验之外，由Gavin King共同执笔编写的这本书就是最好和最权威的Hibernate指南。这本书的第1版叫做*Hibernate in Action*，在Amazon上销量很好，但是由于版权问题，很遗憾未能引入国内。而本书的第2版，即现在这版已经是结合了最新的JPA标准的Hibernate指南，既详细讲述JPA标准的方方面面，又讲述Hibernate框架的各个重要的细节用法，不但对于使用Hibernate的开发人员来是不可或缺的参考书，同样对于使用其他JPA实现的开发人员来说，也是重要的参考。

范凯（Robbin），JavaEye网站创始人

http://www.JavaEye.com

推 荐 序 二

Hibernate自2001年诞生以来，在Java的ORM领域几乎已经成为事实的标准。“掌握Hibernate”已经近乎成为使用Java的软件公司招聘程序员的一个基本要求。之所以形成这一流行做法，有Hibernate本身优秀的理念、成熟的设计与广泛的正面使用体验的因素，也有Hibernate本身的开源、免费的因素，尤其重要的是它包含内容丰富、更新及时的文档。

正因为Hibernate本身基于LGPL开源，我翻译了其2.0版本的文档。该文档在网上公开后获得了始料未及的反应，并且由于Hibernate社区的开放性，它被作为Hibernate文档的正式中文翻译版本纳入了其源代码管理，这是对我翻译工作的最好肯定。正因为其广泛的流传，在Hibernate 3.0版本发布的时候，得以组织起20余人的团队，在短短的三周时间内就推出了经过审校的质量相当高的文档翻译版本，并且不断根据新版本的发布进行更新。仅在“满江红”主站，这一版本就有超过15万次的PDF下载量，HTML版本访问量更是数百万次计。这一中文版本的文档形成了学习Hibernate的良好氛围，对其普及做出了极为重要的贡献，也直接促成了“满江红开放技术研究组织”的成立，同时对后续的很多文档的发布产生了非常积极的示范效应。

但是，官方文档仅有200余页，着重描述的是对其使用参数的设置说明，以及对HQL、关联等概念的使用说明，解释“如何做”而非“为何要这样做”，这对广大学习者仍然有较大的学习障碍。Hibernate的原作者Gavin King和Christian Bauer编写了*Hibernate in Action*（即《Hibernate实战》第1版）一书对其进行了详细的解释。该书英文版一推出，我们就开始了翻译工作，但是后来因版权问题，没能第一时间引进，这实在是一大憾事。

你现在手中的这本《Hibernate实战（第2版）》就是对*Hibernate in Action*的深化升级。这本书的优点在于其实用性，加入了大量的图片和示例辅助说明，比Hibernate随附的文档要清晰明了得多，其英文版我也一直放在电脑包中随身携带，作为速查手册使用。这是一本“高级版文档”和权威的学习指南。

本书是由俞黎敏组织并翻译、审校的。俞黎敏长期工作在开发第一线，对Hibernate有深刻的理解，是“满江红”的管理员之一，也是Gavin King新项目Seam的中文文档翻译负责人。他对本书的理解相当到位，翻译准确度很高。本书正弥补了当初没有能成功引进*Hibernate in Action*的遗憾。在此对他表示由衷的祝贺。

Java作为目前的主流开发语言，Hibernate和JPA仍然具有长期的生命力。希望本书能够进一步推进Hibernate和JPA在国内的应用。

曹晓钢

“满江红开放技术研发组织”共同创始人

http://www.redsaga.com

译 者 序

现在回想起来，自己最早接触数据库相关技术应当可以追溯到12年前，也就是我在大学里开始自学dBASE编程的时候。当时，我一次次地溜进企管系正在进行会计电算化试验的学校机房，翻开自己的笔记本，一行行地敲入程序，然后运行，输入生成表单所要求的数据后，再显示。看到屏幕上显示着一行行不停地滚动的数据，我心中一阵狂喜，感觉这一切是那么地神奇、有趣！

时至今日，无论是因特网应用程序还是企业应用系统，大多数都少不了数据库系统。关系数据库是现代企业的核心，真可谓“无数据库不成应用”。在Java的阵营里，JDBC是数据库操作最基本的API，但JDBC有它的不足之处，如工作量大，不易于重构，等等。Java开发人员之所以觉得难以使用JDBC API，是因为它只是数据库系统之上的一个薄层，只实现通信协议，所有与数据库交互的其他方面都必须由开发人员来承担。而且关系型系统中数据的表格表示法与用于Java应用程序的面向对象方式之间也存在本质上的不匹配，大大影响了Java应用程序的开发效率。于是，Hibernate应运而生。它通过ORM方法，在关系型数据和Java对象之间架起了桥梁，以一种非常实际、直接并且现实的方式很好地解决了对象/关系不匹配的问题。

虽然ORM工具为开发人员承担了95%的对象持久化工作，但是它并不是所有持久化任务的一颗银弹。你应当学会如何为充分利用ORM工具、为自己提高效率而进行相应的抉择，也应当能够充分地认识到它所带来的影响，以及何时需要和不需要它。

本书的作者是开发Hibernate项目的核心人员，他们所提供的见解是最前沿、最直接的。毋庸置疑，本书是Hibernate方面的权威之作，它展示了用Java的面向对象世界映射现实世界的范式，提供了对ORM深层问题和Hibernate背后设计选择的深刻见解，可以让读者对ORM作为一种企业技术的有效应用有深刻的理解。它比较适合中高级水平的读者，当然也适合刚准备进入ORM领域的读者，但是他们至少需要具备面向对象软件开发的基础知识和实践经验，并且熟悉Java编程语言和UML以及数据库的基本知识。

本书的前三章对想要利用Hibernate迅速创建应用程序的开发人员来说是不可或缺的，同时它们对于真正想要理解Hibernate概念、框架、方法以及形成框架设计的原因的人来说也显得尤为重要。

如果你知道`Session.get()`与`Session.load()`的区别，了解Hibernate的`save()`方法与JPA的`persist()`的区别，并且知道JPA没有Hibernate Session API所提供的`evict(object)`方法来剔除单个持久化实例的话，那就可以从任一部分开始阅读。

对于老用户来说，如果你想了解Hibernate 3.x与Hibernate 2.x有何不同，可以认真看看Hibernate 3到底在内核持久化引擎上面做了哪些改动。比如，新内核引擎基于一个事件/监听器的

模型，如果Hibernate需要保存一个对象，就会触发一个事件。这些细节的底层实现与思想聆听作者为你娓娓道来，你可以有更加深刻的体会。

通过本书，你可以知道如何高效地使用HQL的最佳实践，也可以从中知道HQL与Criteria的优缺点，以及Criteria API在什么情况下会是更好的选择；你可以清楚地获知JBoss Cache支持什么样的事务，也可以由详细地对比分析JPA与Hiberante之后得到，譬如，如果你必须使用定位参数，要记住Hibernate是从0开始计数，而Java Persistence则从1开始计数，你必须给JPA QL查询字符串中的每一个问号都添加一个数字。而且可以知道如何减少内存消耗并防止很长的脏查询周期，但是JPA却不支持，以及更多其他信息，如distinct在SQL结果中重复吗？它是在SQL级别上操作还是内存中进行二次过滤呢？等等。

本书旨在使你更快速、更专业地进入ORM——这个企业级应用程序开发领域最核心的技术之一，并从事数据库相关的设计工作。当阅读本书时，你会遇到许多需要动手进行验证的实例，可以利用本书附带的示例程序进行练习与实践。下载地址为：http://caveatemptor.hibernate.org/。CaveatEmptor示例应用程序代码以原生的Hibernate、Java Persistence和JBoss Seam三种实现进行提供，方便读者学习与使用。

正如在翻译过程中发现原著的错误一样，虽然我们在翻译过程中竭力以求信、达、雅，但限于自身水平，必定仍会有诸多不足，还望各位读者不吝指正。大家可以通过访问我的博客http://YuLimin.JavaEye.com/或者发送电子邮件到YuLimin@163.com进行互动。

翻译时所发现的原书错误通过与作者的交流得到了确认，详细情况可以参见我的博客或Hibernate官方论坛 http://forum.hibernate.org/search.php?search_author=YuLimin。

关于术语的翻译，可以到http://yulimin.javaeye.com/blog/70124讨论。

感谢ajax.cn的李锟（dlee）对我在翻译中碰到的问题进行的深入讨论，并对本书翻译时所采用的术语进行了认真的磋商；感谢JavaEye的网友们在术语表讨论贴中提出许多中肯的建议；感谢满江红开源组织的曹晓钢提供的一些翻译注意事项和热情的帮助；感谢图灵公司的编辑。

本书由我组织翻译，杨春花负责翻译第1章到第8章，彭永康负责翻译第9章到第16章，我负责翻译第17章并对全书所有章节进行全面审校。参与翻译与审校的还有：蒋凌锋、魏伟、叶荣生、凌家亮、黄健、万国辉、宋可信、李宗波等，在此再次深表感谢。

本书章节安排合理，内容承上启下，阅读起来有一气呵成之感，不会让你感觉到是在啃“砖块”，而是在享受。快乐分享，实践出真知，最后，祝大家能够像我一样在阅读中享受本书带来的乐趣！

Read a bit and take it out, then come back read some more.

俞黎敏

2007年年底于广州

译 者 简 介

杨春花

满江红开放技术研究组“Seam 2.0 Reference”中文翻译工作之核心翻译与审校人员。

彭永康

现任景德镇陶瓷学院信息工程学院信息与计算科学教研室主任，副教授。曾参与国家863CIMS单元目标产品项目“陶瓷产品CAD集成系统”，中小企业创新基金项目“陶瓷产品CAD集成软件”等。主要研究方向：计算机图形学、多媒体技术。

俞黎敏（ID：YuLimin，网名：阿敏总司令）

技术顾问，深入了解电力、电信行业的系统，并负责核心系统研发与管理工作。开源爱好者，曾经参与Spring中文论坛组织“Spring 2.0 Reference”中文翻译的一审与二审工作，满江红开放技术研究组织的“Seam 1.2.1 Reference”中文翻译工作，并组织和完成“Seam 2.0 Reference”中文翻译工作。利用业余时间担任CSDN、CJSDN、Dev2Dev、Matrix、JavaWorldTW、Spring中文等Java论坛版主，在各大技术社区推动开源和敏捷开发做出了积极的贡献。博客地址为：http://YuLimin.JavaEye.com

第2版序

两年前，*Hibernate in Action*一出版就被公认为是关于Hibernate的权威性图书，同时它也成为了ORM（对象/关系映射）方面的权威之作。

之后，持久化（persistence）的前景也随着JPA（Java Persistence API）的发布而改变，这个新标准用以解决Java EE和Java SE的ORM，并在Java Community Process的主导下开发，成为EJB 3.0规范的一部分。

在JPA的开发过程中，EJB 3.0专家组从已经在Java社区应用的ORM框架的经验中受益颇丰。Hibernate作为先驱者之一，已经对Java Persistence技术的导向产生了非常显著的影响力。这不仅是由于有Gavin King和EJB 3.0标准化成果的其他Hibernate团队成员的参与，很大程度上还归功于Hibernate带给ORM的直接且实用的方法，以及其API的简单、清楚和强大——因此它们的结果激励了Java社区。

除了对Java Persistence的贡献之外，Hibernate的开发人员还使得Hibernate有了很大进展，这体现在本书所描述的Hibernate 3中。这些进展包括支持对大型数据集的操作、特别适用处理遗留数据库的额外和更复杂的映射选项、数据过滤器、管理对话的策略，以及与利用JSF和EJB 3.0进行Web应用程序开发的新框架Seam的集成。

因此，本书远远不只是*Hibernate in Action*的第2版。它全面概括了所有JPA的功能和Hibernate 3的功能，并对这两者进行了详细的对比分析。它描述了如何使用Hibernate实现Java Persistence标准，以及如何利用Hibernate扩展来实现Java Persistence。

更重要的是，在介绍Hibernate和Java Persistence的过程中，Christian Bauer和Gavin King举例并解释了在设计和使用ORM框架时需要考虑到的一些基本原则和决策。他们对ORM深层问题的深刻见解，让读者对于ORM作为一种企业技术的有效应用有了深刻的理解。

因此，本书适合广大的开发人员，从ORM的初学者到经验丰富的开发人员，帮助他们学习更多在Java社区中已经出现和正在继续出现的尖端技术。

Linda Demichiel

（Sun公司规范主管，EJB 3.0和Java Persistence首席架构师）

第1版序

毋庸置疑，关系数据库是现代企业的核心。

包括Java在内的现代程序设计语言，提供了应用程序级业务实体的直观、面向对象的视图，但这些业务实体底层的企业数据本质上是密切相关的。此外，相对于早期的导航模型以及后来的OODB模型，关系模型的主要优势在于，它被设计为与其提供数据的编程处理和应用程序级视图在本质上是无关的。

许多人已经做了很多尝试来搭建关系和面向对象技术之间的桥梁，或者用一个取代另一个，但是这两者之间的差异是当今企业计算面临的难点之一。通过其ORM方法，Hibernate架起了关系型数据和Java对象之间的桥梁。Hibernate是以一种非常实际、直接并且现实的方式来迎合这一挑战的。

正如Christian Bauer和Gavin King在书中所述，除了最简单的企业环境之外，最有效地应用ORM技术要求理解关系型数据和对象以及它们之间的中介的工作原理，并对中介进行配置。这要求开发人员通晓应用程序及其数据需求，以及SQL查询语言、关系存储结构和关系技术提供的最优化潜能。

Hibernate不仅提供了一个应对这些需求的功能齐全的解决方案，而且还提供了一个灵活的可配置的架构。Hibernate开发人员专注于设计，并使它具有模块化、可热插拔、可扩展以及可用户定制等特点。结果，在它最初发布的几年里，Hibernate迅速变成了企业开发人员所适用的一种前沿ORM技术——这也是理所当然的。

本书全面概括了Hibernate，涵盖了如何利用它的类型映射能力和工具来为关联和继承建模；如何利用Hibernate查询语言有效地获取对象；如何在托管和非托管环境下配置Hibernate；以及如何使用它的工具。此外，本书还提供了对ORM深层问题和Hibernate背后设计选择的深刻见解。这些见解可以让读者对ORM作为一种企业技术的有效应用有深刻的理解。

*Hibernate in Action*是当今企业计算中利用Hibernate和ORM的权威指南。

Linda Demichiel

（Sun公司，EJB首席架构师）

第2版前言

本书第1版*Hibernate in Action*的前言开始引用了Anthony Berglas的话："不能只因为某人能用鼻子把细枝沿地面向前推，就认为那就是收集柴火的最好办法。"自第1版出版后，软件开发人员赖以管理信息的Hibernate项目和策略以及理念都得到了进一步发展。然而，基本的问题仍然一样——我们每天打交道的每家公司仍然在使用SQL数据库，并且Java在行业中成了企业应用程序开发的首选。

在关系型系统中数据的表格表示法与用于面向对象的Java应用程序的对象网络仍有本质上的区别。我们仍然看到对象/关系不匹配，而且经常看到这种不匹配的重要性和代价被低估了。

另一方面，我们现在有各种各样的工具和解决方案可用于处理这个问题。我们已经捡好了柴火，并且袖珍打火机也换成了火焰投射器。

现在Hibernate已经准备好了第3次重要的发布，Hibernate 3.2就是本书所讲述的版本。相比Hibernate的老版本，这个全新的重要发布具有两倍的特性——因此本书的厚度也几乎是*Hibernate in Action*的两倍。其中大部分特性都是包括你在内的每天使用Hibernate的开发人员所需要的。有时我们说，Hibernate能解决Java应用程序开发人员在创建数据库应用程序时必须处理的所有问题的90%。有了最新的Hibernate版本，这个数字就接近99%了。

随着Hibernate的成熟以及其用户基数和社区的不断成长，许多开发人员发现缺少数据管理和数据库应用程序开发的Java标准。我们甚至在*Hibernate in Action*中告诉过你不要使用的EJB 2.x的实体bean。

下面谈谈EJB 3.0和新的Java Persistence标准。这个新的行业标准对Java开发者社区而言是向前迈出了重要的一步。它定义了一个轻量级、简化的编程模型和强大的对象/关系持久化。这个新标准的许多关键理念都模仿Hibernate和其他成功的对象/关系持久化方案。最新的Hibernate版本实现了Java Persistence标准。

因此，除了这个新的集各种功能于一身的Hibernate之外，现在你可以像使用其他Java Persistence提供程序（provider）一样使用Hibernate，不管是否使用其他EJB 3.0组件和Java EE 5.0服务。Hibernate与这么一个丰富的编程模型的深入整合，使你能够设计和实现以前用手工难以创建的应用程序功能。

本书旨在提供一个对Hibernate和Java Persistence（以及所有相关的EJB 3.0理念）完整而准确的指南。希望你会喜欢学习Hibernate，并且把本书放在案头作为每日工作的参考。

第 1 版前言

不能只因为某人能用鼻子把细枝沿地面向前推，就认为那就是收集柴火的最好办法。

——Anthony Berglas

今天，许多软件开发人员都用企业信息系统（EIS）。这种应用程序创建、管理和存储结构化信息，并在多个物理位置的许多用户之间共享信息。

EIS数据存储涉及大量使用基于SQL的数据库管理系统。我们曾经工作的每家公司至少都用一个SQL数据库；大部分公司的业务核心完全依赖关系数据库技术。

在过去的5年里，Java程序语言的广泛应用已经使软件开发中面向对象的范式占据了统治地位。现在，开发人员享受着面向对象的好处。然而，大量的业务还是依赖于对昂贵的关系数据库系统的长期投资。不仅仅是特殊的供应商产品不容易更改，而且现有遗留数据也必须可用于（且经由）这个全新的面向对象的Web应用程序。

然而，在关系型系统中数据的表格表示法与应用在面向对象的Java应用程序中的对象网络仍有本质上的不同。这种区别导致了所谓的对象/关系范式（object/relational paradigm）的不匹配。

传统上，这种不匹配的重要性和代价被低估了，并且解决这种不匹配的工具也不充足。同时，Java开发人员抱怨针对不匹配的关系技术；数据专家则抱怨对象技术。

ORM是对这种不匹配问题的自动化解决方案的命名。对于厌烦了冗长乏味的数据访问代码的开发人员来说，好消息是ORM已经成熟了。用ORM中间件创建的应用程序会更加便宜、更高性能、更少的供应商依赖性，并且更能够应付内部对象或者底层SQL模式的改变。令人惊异的是，这些好处现在都可以供Java开发人员免费享用。

Gavin King于2001年后期开始开发Hibernate，这时他发现流行的持久化解决方案——CMP Entity Beans——还没有发展成带有复杂数据模型的重要应用程序。Hibernate开始了作为一个独立的、非商业的、开源项目的生涯。

Hibernate团队（包括作者）通过勤奋的实践已经掌握了ORM——也就是说，通过聆听用户需求并实现满足这些需求所需要的东西。结果，Hibernate成了一个实用的解决方案，增进了开发人员的生产力和技术领导能力。Hibernate已经被数以万计的用户使用，并且用到了数以千计的生产应用中。

当那个时代的需求变得势不可挡时，Hibernate团队断定这个项目未来的成功（和Gavin持续的明智）要求专业开发人员将全部精力投入到Hibernate。Hibernate在2003年后期加入了jboss.org，现在具有了商业特征。可以从JBoss公司购买商业支持和培训，但是商业培训不应该是学习

Hibernate的唯一途径。

很明显，许多（可能甚至大多数）Java项目都受益于ORM解决方案（如Hibernate）的使用，虽然这在几年前还不明显。随着ORM技术日益成为主流，产品文档（如Hibernate的免费用户手册）已不再充足。我们认识到Hibernate社区和新的Hibernate用户需要一本全面的图书，不仅仅学习利用Hibernate来开发软件，还要理解和领悟对象/关系的不匹配，以及Hibernate设计背后的动机。

本书是我们利用一年多所有空闲时间努力创作的结晶，也是许多激烈辩论和学习经验的来源。我们希望本书成为关于Hibernate的优秀指南（即“Hibernate圣经”，如一位评论家所言），并且大体上也是对象/关系不匹配和ORM的第一个全面的文档。

希望你会发现它对你有所帮助，并且喜欢用Hibernate工作。

关 于 本 书

我们编写本书时有3个目标，因此你可以把它当作以下3种书来读：

- 一本关于Hibernate、Java Persistence和EJB 3.0的教程，指导你完成利用这些解决方案的第一步。
- 一本关于ORM、对象处理、查询、性能优化和应用程序设计的所有基础和高级Hibernate特性的学习指南。
- 一本在你需要一个关于Hibernate和Java Persistence功能的完整而技术准确的定义时的参考书。

图书通常要么是教程要么是参考指南，这种伸缩性是要付出很多努力的。如果你刚接触Hibernate，建议你从头开始读本书，从第1章和第2章的教程开始。如果你已经用过第1版的Hibernate，那么应该快速阅读前两章以获得整体概念，然后阅读第3章。

适当的时候，我们会告诉你某个特定的节或者主题是否为可选的，或者是否为第一次阅读时可以忽略的参考资料。

导读

本书分成三大部分。

第一部分介绍对象/关系范式的不匹配，并阐述ORM（Object/Relational Mapping，对象/关系映射）背后的基本原理。通过一个实际操作的教程，让你开始第一个Hibernate、Java Persistence或者EJB 3.0项目。我们探讨领域模型的Java应用程序设计和创建ORM元数据的方法。

第二部分重点介绍如何把Java类和属性映射到SQL表和列。我们利用XML映射文件和Java注解，探讨Hibernate和Java Persistence中所有基础和高级的映射方法；介绍如何处理继承、集合和复杂的类关联。最后讨论与遗留数据库模式的整合，以及一些特别灵活的映射策略。

第三部分介绍关于对象的处理，以及如何使用Hibernate和Java Persistence加载和存储数据。我们介绍编程接口、如何编写事务性和支持对话的应用程序，以及如何编写查询。随后，把关注点放在分层Java应用程序的正确设计和实现上。我们讨论最常用于Hibernate的设计模式，例如DAO（Data Access Object，数据访问对象）模式和EJB Command模式。你将学会如何轻松地测试Hibernate应用程序，以及使用ORM软件时有哪些相关的其他最佳实践。

最后介绍JBoss Seam框架，它进一步深化了许多Hibernate概念，使你能够轻松地创建会话的Web应用程序。我们确信你会对这一章感兴趣，即使你不打算使用Seam。

读者对象

本书的读者应该具备面向对象软件开发的基础知识，并且应该已经在实践中使用过这些知识。要理解应用程序示例，应该熟悉Java编程语言和UML（Unified Modeling Language，统一建模语言）。

我们的目标读者主要是使用基于SQL的数据库系统工作的Java开发人员。我们将介绍如何利用ORM充分提升工作效率。

如果你是一位数据库开发人员，本书可以帮助你入门面向对象的软件开发。

如果你是一位数据库管理员，将会关注ORM如何影响性能、如何调优SQL数据库管理系统和持久层的性能来实现性能目标。由于数据访问是大部分Java应用程序的瓶颈，因此本书十分关注性能问题。许多数据库管理员（DBA）对于把性能委托给工具生成的SQL代码很是不安，这是可以理解的；我们力求减少这种担忧，并且强调不应该使用工具托管的数据访问的情况。你可能会欣慰地发现，我们并没有宣称ORM对于每一个问题都是最好的解决方案。

代码约定

本书提供了丰富的示例，包括所有Hibernate应用程序所需要创建的东西：Java代码、Hibernate配置文件和XML映射元数据文件。代码清单和正文中的源代码使用等宽字体。此外，正文中的Java方法名称、组件参数、对象属性、XML元素和属性也都用等宽字体表示。

Java、HTML和XML可能都很长。在许多情况中，源代码（在线提供）已经重新设定了格式，如增加换行，并重新进行缩排以便适应页面空间。在少数情况中，这样可能还不够，代码清单还包括了续行符（➥）。此外，在正文中描述代码时通常删去了源代码中的注释。

有些源代码清单中配有代码注释，强调重要的概念。有时候，编号的项目符号与紧接在代码清单后面的正文解释相对应。

源代码下载

Hibernate是一个以LGPL（Lesser GNU Public License）许可证形式发布的开源项目。在Hibernate网站（www.hibernate.org/）上有下载Hibernate包的说明，Hibernate包以源代码或者二进制的形式提供。

本书Hello World和CaveatEmptor示例的源代码可以根据自由许可证（像BSD一样）从 http://caveatemptor.hibernate.org/下载。CaveatEmptor示例应用程序代码在这个网站上以不同的风格提供，例如关注原生的Hibernate、Java Persistence和JBoss Seam。也可以在出版社的网站www.manning.com/ bauer2下载本书的示例代码[①]。

① 本书示例代码也可以从图灵网站（www.turingbook.com）的本书配套主页上下载。——编者注

作者在线

购买本书也意味着你获得了免费访问Manning出版社所运营的私有Web论坛的权利，在这个论坛上你可以对本书作出评论，可以提出技术问题并从作者和其他读者那里获得帮助。为了要访问论坛并且订阅其内容，可以将Web浏览器指向 http://www.manning.com/crane。该网页上提供了注册以后如何访问论坛的信息，也说明了你可以获得什么类型的帮助，并且说明了论坛的行为准则。

Manning出版社对读者的承诺是提供一个读者之间以及读者和作者之间能够展开深入对话的场所，但是并不承诺作者必须要参与到何种程度，作者对于论坛的贡献是志愿的（并且是无偿的）。我们建议你向作者多提一些有挑战性的问题，以免他们丧失兴趣！

只要本书还在印刷销售，作者在线论坛和以前讨论的存档都可以从出版社的网站上访问到。

封面图片

本书的封面图片来自于一本土耳其奥斯曼帝国服饰集锦的画册，由伦敦老邦德街的William Miller于1802年1月1日出版。画册中的扉页已经丢失了，因此我们无法确定确切的创作时间。书中的目录使用英语和法语来标识图画，每张画有创作它的两位艺术家的名字，他们无疑会为他们的艺术能够美化两百年之后的一本计算机编程书籍的封面而感到惊讶。

Manning出版社的一位编辑在位于曼哈顿西26街“Garage”的古董跳蚤市场购买了这本画册。卖主是住在土耳其安卡拉的一个美国人，交易的时间发生在那天他准备收摊的时候。Manning出版社的编辑随身没有携带购买这本画册所需的足够现金，并且卖主礼貌地拒绝了他使用信用卡和支票。因为卖主当天晚上要飞回安卡拉，所以似乎已经没有希望了。那么是怎么解决的呢？无非就是一个通过握手来约定的老式的君子协议。卖主提议通过银行转账付款，编辑仅仅在纸片上抄下了收款银行的信息，随后画册就到了他的手里。不用说，第二天就将款付给了卖主。我们感谢这位陌生人能够如此信任我们的同事。这使得我们回忆起了那个很久以前的美好时代。

来自奥斯曼帝国画册的图片，就像在丛书封面上出现的其他插图一样，将两个世纪之前服饰习俗的丰富性和多样性带到了我们的身边。这些图片令我们回忆起与那个时期（以及除了通信极其发达的今天之外的每一个其他历史时期）之间的隔绝和距离。

从那以后，服饰习俗所代表的不同区域的差异已经改变了，那个时期丰富多彩的服饰文化也逐渐褪色了。现在甚至连区分一个大陆的居民和另一个大陆的居民常常都是困难的。也许我们应该尽量乐观地看待这些变化，文化上、视觉上的差异已经转变为更加多样化的个人生活上的差异，或者更加多样化和更加有趣的智力上和技术生活上的差异。

我们使用来自这本画册中的图片作为本书的封面，这为我们带来了两个世纪之前丰富的、多样性的区域生活。我们Manning出版社的编辑们使用这种方式来赞美计算机行业中的独创性和主动性，当然，还包括其中的乐趣。

致谢

本书从薄薄的*Hibernate in Action*增长为目前厚厚的一本。如果没有许多人的帮助，我们是不可能做到的。

Emmanuel Bernard作为本书的技术审核，干得很棒；感谢他花了许多时间来编辑我们有问题的代码示例，也要感谢其他审核人员：Patrick Dennis、Jon Skeet、Awais Bajwa、Dan Dobrin、Deiveehan Nallazhagappan、Ryan Daigle、Stuart Caborn、Patrick Peak、TVS Murthy、Bill Fly、David Walend、Dave Dribin、Anjan Bacchu、Gary Udstrand和Srinivas Nallapati。特别感谢Linda DiMichiel为本书作序，就像她为第1版作序一样。

Marjan Bace再次在Manning出版社组建了一支优秀的制作团队：Sydney Jones编辑我们粗糙的原稿并把它变成一本真正的书；Tiffany Taylor、Elizabeth Martin和Andy Carroll找出了我们所有的排印错误，使得本书更易于阅读；Dottie Marsico负责排版，使本书拥有了漂亮的外观；Mary Piergies协调和组织制作进度。感谢与我们共事的所有人。

目　录

第一部分　从 Hibernate 和 EJB 3.0 开始

第 1 章　理解对象/关系持久化 ……2

1.1　什么是持久化 ……3
1.1.1　关系数据库……3
1.1.2　理解SQL……4
1.1.3　在Java中使用SQL……5
1.1.4　面向对象应用程序中的持久化……5
1.2　范式不匹配 ……6
1.2.1　粒度问题……8
1.2.2　子类型问题……9
1.2.3　同一性问题……10
1.2.4　与关联相关的问题……11
1.2.5　数据导航的问题……12
1.2.6　不匹配的代价……13
1.3　持久层和其他层 ……14
1.3.1　分层架构……14
1.3.2　用SQL/JDBC手工编写持久层……15
1.3.3　使用序列化……16
1.3.4　面向对象的数据库系统……16
1.3.5　其他选项……17
1.4　ORM……17
1.4.1　什么是ORM……17
1.4.2　一般的ORM问题……19
1.4.3　为什么选择ORM……19
1.4.4　Hibernate、EJB 3和JPA简介……21
1.5　小结……24

第 2 章　启动项目 ……25

2.1　启动Hibernate项目……25
2.1.1　选择开发过程……26
2.1.2　建立项目……28
2.1.3　Hibernate配置和启动……34
2.1.4　运行和测试应用程序……42
2.2　启动Java Persistence项目……48
2.2.1　使用Hibernate Annotations……48
2.2.2　使用Hibernate EntityManager……51
2.2.3　引入EJB组件……57
2.2.4　切换到Hibernate接口……62
2.3　反向工程遗留数据库……64
2.3.1　创建数据库配置……64
2.3.2　定制反向工程……65
2.3.3　生成Java源代码……67
2.4　与Java EE服务整合……69
2.4.1　与JTA整合……70
2.4.2　JNDI绑定的`SessionFactory`……73
2.4.3　JMX服务部署……75
2.5　小结……75

第 3 章　领域模型和元数据 ……77

3.1　CaveatEmptor应用程序……77
3.1.1　分析业务领域……78
3.1.2　CaveatEmptor领域模型……78
3.2　实现领域模型……80
3.2.1　处理关注点渗漏……81
3.2.2　透明和自动持久化……81
3.2.3　编写POJO和持久化实体类……82
3.2.4　实现POJO关联……84
3.2.5　把逻辑添加到访问方法……88
3.3　ORM元数据……89
3.3.1　XML中的元数据……89
3.3.2　基于注解的元数据……91
3.3.3　使用XDoclet……95
3.3.4　处理全局的元数据……97
3.3.5　运行时操作元数据……101
3.4　其他实体表示法……102
3.4.1　创建动态的应用程序……103
3.4.2　表示XML中的数据……108

3.5 小结……112

第二部分 映射概念和策略

第4章 映射持久化类……114
4.1 理解实体和值类型……114
4.1.1 细粒度的领域模型……114
4.1.2 定义概念……115
4.1.3 识别实体和值类型……115
4.2 映射带有同一性的实体……117
4.2.1 理解Java同一性和等同性……117
4.2.2 处理数据库同一性……117
4.2.3 数据库主键……120
4.3 类映射选项……123
4.3.1 动态的SQL生成……123
4.3.2 使实体不可变……124
4.3.3 给查询命名实体……125
4.3.4 声明包名称……125
4.3.5 用引号把SQL标识符括起来……126
4.3.6 实现命名约定……126
4.4 细粒度的模型和映射……127
4.4.1 映射基础属性……128
4.4.2 映射组件……133
4.5 小结……137

第5章 继承和定制类型……139
5.1 映射类继承……139
5.1.1 每个带有隐式多态的具体类一张表……140
5.1.2 每个带有联合的具体类一张表……142
5.1.3 每个类层次结构一张表……145
5.1.4 每个子类一张表……148
5.1.5 混合继承策略……151
5.1.6 选择策略……154
5.2 Hibernate类型系统……155
5.2.1 概述实体和值类型……155
5.2.2 内建的映射类型……156
5.2.3 使用映射类型……160
5.3 创建定制的映射类型……161
5.3.1 考虑定制的映射类型……161
5.3.2 扩展点……162
5.3.3 定制映射类型的案例……163
5.3.4 创建UserType……164
5.3.5 创建CompositeUserType……166
5.3.6 参数化定制类型……169
5.3.7 映射枚举……171
5.4 小结……175

第6章 映射集合和实体关联……177
6.1 值类型的set、bag、list和map……177
6.1.1 选择集合接口……178
6.1.2 映射set……179
6.1.3 映射标识符bag……180
6.1.4 映射list……181
6.1.5 映射map……182
6.1.6 排序集合和有序集合……183
6.2 组件的集合……185
6.2.1 编写组件类……185
6.2.2 映射集合……186
6.2.3 启用双向导航……186
6.2.4 避免非空列……187
6.3 用注解映射集合……189
6.3.1 基本的集合映射……189
6.3.2 排序集合和有序集合……190
6.3.3 映射嵌入式对象的集合……191
6.4 映射父/子关系……192
6.4.1 多样性……193
6.4.2 最简单的可能关联……193
6.4.3 使关联双向……195
6.4.4 级联对象状态……197
6.5 小结……203

第7章 高级实体关联映射……204
7.1 单值的实体关联……204
7.1.1 共享的主键关联……205
7.1.2 一对一的外键关联……208
7.1.3 用联结表映射……210
7.2 多值的实体关联……213
7.2.1 一对多关联……214
7.2.2 多对多关联……219
7.2.3 把列添加到联结表……223
7.2.4 映射map……229
7.3 多态关联……231
7.3.1 多态的多对一关联……232

7.3.2 多态集合……233
7.3.3 对联合的多态关联……234
7.3.4 每个具体类一张多态表……236
7.4 小结……238
第8章 遗留数据库和定制SQL……239
8.1 整合遗留数据库……239
8.1.1 处理主键……240
8.1.2 带有公式的任意联结条件……250
8.1.3 联结任意的表……254
8.1.4 使用触发器……257
8.2 定制SQL……260
8.2.1 编写定制CRUD语句……261
8.2.2 整合存储过程和函数……265
8.3 改进Schema DDL……271
8.3.1 定制SQL名称和数据类型……271
8.3.2 确保数据一致性……273
8.3.3 添加领域约束和列约束……274
8.3.4 表级约束……276
8.3.5 数据库约束……278
8.3.6 创建索引……279
8.3.7 添加辅助的DDL……280
8.4 小结……282
第三部分 会话对象处理
第9章 使用对象……284
9.1 持久化生命周期……284
9.1.1 对象状态……285
9.1.2 持久化上下文……287
9.2 对象同一性和等同性……289
9.2.1 引入对话……289
9.2.2 对象同一性的范围……290
9.2.3 脱管对象的同一性……292
9.2.4 扩展持久化上下文……296
9.3 Hibernate接口……296
9.3.1 保存和加载对象……297
9.3.2 使用脱管对象……302
9.3.3 管理持久化上下文……306
9.4 JPA……308
9.4.1 保存和加载对象……309
9.4.2 使用脱管的实体实例……312
9.5 在EJB组件中使用Java Persistence……315
9.5.1 注入EntityManager……315
9.5.2 查找EntityManager……317
9.5.3 访问EntityManagerFactory……318
9.6 小结……319
第10章 事务和并发……320
10.1 事务本质……320
10.1.1 数据库和系统事务……321
10.1.2 Hibernate应用程序中的事务……323
10.1.3 使用Java Persistence的事务……331
10.2 控制并发访问……334
10.2.1 理解数据库级并发……334
10.2.2 乐观并发控制……338
10.2.3 获得额外的隔离性保证……343
10.3 非事务数据访问……346
10.3.1 揭开自动提交的神秘面纱……346
10.3.2 使用Hibernate非事务地工作……347
10.3.3 使用JTA的可选事务……349
10.4 小结……349
第11章 实现对话……351
11.1 传播Hibernate Session……351
11.1.1 Session传播的用例……352
11.1.2 通过线程局部传播……353
11.1.3 利用JTA传播……355
11.1.4 利用EJB传播……356
11.2 利用Hibernate的对话……357
11.2.1 提供对话保证……357
11.2.2 利用脱管对象的对话……358
11.2.3 给对话扩展Session……360
11.3 使用JPA的对话……366
11.3.1 Java SE中的持久化上下文传播……367
11.3.2 在对话中合并脱管对象……368
11.3.3 在Java SE中扩展持久化上下文……369
11.4 使用EJB 3.0的对话……373
11.4.1 使用EJB的上下文传播……373
11.4.2 利用EJB扩展持久化上下文……376
11.5 小结……380

第 12 章 有效修改对象……381
12.1 传播性持久化……381
12.1.1 按可到达性持久化……382
12.1.2 把级联应用到关联……383
12.1.3 使用传播性状态……386
12.1.4 利用JPA的传播性关联……391
12.2 大批量和批量操作……392
12.2.1 使用HQL和JPA QL的大批量语句……392
12.2.2 利用批量处理……395
12.2.3 使用无状态的会话……397
12.3 数据过滤和拦截……398
12.3.1 动态数据过滤……399
12.3.2 拦截Hibernate事件……402
12.3.3 内核事件系统……408
12.3.4 实体监听器和回调……410
12.4 小结……411
第 13 章 优化抓取和高速缓存……412
13.1 定义全局抓取计划……412
13.1.1 对象获取选项……412
13.1.2 延迟的默认抓取计划……415
13.1.3 理解代理……415
13.1.4 禁用代理生成……417
13.1.5 关联和集合的即时加载……418
13.1.6 通过拦截延迟加载……420
13.2 选择抓取策略……422
13.2.1 批量预抓取数据……422
13.2.2 通过子查询预抓取集合……424
13.2.3 通过联结即时抓取……425
13.2.4 给二级表优化抓取……428
13.2.5 优化指导方针……430
13.3 高速缓存基本原理……436
13.3.1 高速缓存策略和范围……436
13.3.2 Hibernate高速缓存架构……439
13.4 高速缓存实践……442
13.4.1 选择并发控制策略……442
13.4.2 理解高速缓存区域……444
13.4.3 设置本地的高速缓存提供程序……445
13.4.4 设置重复的高速缓存……446
13.4.5 控制二级高速缓存……449
13.5 小结……451
第 14 章 利用 HQL 和 JPA QL 查询……452
14.1 创建和运行查询……452
14.1.1 准备查询……453
14.1.2 执行查询……460
14.1.3 使用具名查询……463
14.2 基本的HQL和JPA QL查询……465
14.2.1 选择……466
14.2.2 限制……467
14.2.3 投影……472
14.3 联结、报表查询和子查询……473
14.3.1 联结关系和关联……473
14.3.2 报表查询……482
14.3.3 利用子查询……485
14.4 小结……487
第 15 章 高级查询选项……488
15.1 利用条件和示例查询……488
15.1.1 基本的条件查询……489
15.1.2 联结和动态抓取……493
15.1.3 投影和报表查询……497
15.1.4 按示例查询……500
15.2 利用原生的SQL查询……503
15.2.1 自动的结果集处理……503
15.2.2 获取标量值……504
15.2.3 Java Persistence中的原生SQL……505
15.3 过滤集合……507
15.4 高速缓存查询结果……508
15.4.1 启用查询结果高速缓存……509
15.4.2 理解查询高速缓存……509
15.4.3 什么时候使用查询高速缓存……510
15.4.4 自然标识符高速缓存查找……511
15.5 小结……512
第 16 章 创建和测试分层的应用程序……513
16.1 Web应用程序中的Hibernate……513
16.1.1 用例简介……514
16.1.2 编写控制器……514
16.1.3 OSIV模式……516

16.1.4　设计巧妙的领域模型……518
16.2　创建持久层……521
16.2.1　泛型的数据访问对象模式……521
16.2.2　实现泛型CRUD接口……523
16.2.3　实现实体DAO……525
16.2.4　利用数据访问对象……526
16.3　命令模式简介……529
16.3.1　基础接口……530
16.3.2　执行命令对象……531
16.3.3　命令模式的变形……533
16.4　利用EJB 3.0设计应用程序……534
16.4.1　利用有状态的bean实现会话…534
16.4.2　利用EJB编写DAO……536
16.4.3　利用依赖注入……537
16.5　测试……538
16.5.1　理解不同种类的测试……539
16.5.2　TestNG简介……539
16.5.3　测试持久层……542
16.5.4　考虑性能基准……549
16.6　小结……550
第 17 章　JBoss Seam 简介……551
17.1　Java EE 5.0编程模型……551
17.1.1　JSF详解……552
17.1.2　EJB 3.0详解……553
17.1.3　用JSF和EJB 3.0编写Web应用程序……554
17.1.4　分析应用程序……562
17.2　用Seam改善应用程序……564
17.2.1　配置Seam……565
17.2.2　将页面绑定到有状态的Seam组件……566
17.2.3　分析Seam应用程序……570
17.3　理解上下文组件……575
17.3.1　编写登录页面……575
17.3.2　创建组件……577
17.3.3　给上下文变量起别名……579
17.3.4　完成登录/注销特性……580
17.4　验证用户输入……583
17.4.1　Hibernate Validator简介……584
17.4.2　创建注册页面……584
17.4.3　用Seam实现国际化……591
17.5　利用Seam简化持久化……594
17.5.1　实现对话……595
17.5.2　让Seam管理持久化上下文……601
17.6　小结……604
附录 A　SQL 基础知识……606
附录 B　映射快速参考……609

Part 1 第一部分

从 Hibernate 和 EJB 3.0 开始

本部分介绍为什么对象持久化这个主题如此复杂，在实践中可以应用什么解决方案。第 1 章介绍对象/关系范式不匹配（object/relational paradigm mismatch）和几种处理这种不匹配的策略，其中最重要的就是 ORM（Object/Relational Mapping，对象/关系映射）。第 2 章将循序渐进地讲解 Hibernate、Java Persistence 和 EJB 3.0——你将以各种不同的方式来实现和测试“Hello World”示例。有了这些准备之后，在第 3 章你就可以学习如何设计和实现 Java 中复杂的业务领域模型，以及可以使用哪些映射元数据选项了。

读完这部分，你将理解为什么需要 ORM，以及 Hibernate、Java Persistence 和 EJB 3.0 在实际应用程序中的工作原理。你将编写自己的第一个小项目，并且准备好迎接更复杂的问题。你也将理解如何用 Java 领域模型来实现现实世界的业务实体，以及应该以什么样的格式使用 ORM 元数据。

本部分内容

- 第 1 章　理解对象/关系持久化
- 第 2 章　启动项目
- 第 3 章　领域模型和元数据

第1章 理解对象/关系持久化

本章内容

- 利用SQL数据库持久化对象
- 对象/关系范式不匹配
- 面向对象应用程序中的持久层
- ORM的背景

在所开发的所有软件项目中，管理持久化数据的方法一直都是关键的设计决策。考虑到持久化数据对于Java应用程序并不是什么新的或者不寻常的需求，我们当然希望能够在一些类似的、公认的（well-established）持久化解决方案中进行简单的选择，就像Web应用程序框架（Struts还是WebWork）、GUI组件框架（Swing还是SWT）或者模板引擎（JSP还是Velocity）那样。每一种竞争的解决方案都有自身的各种优缺点，但是它们的范围和总体倾向是相同的。糟糕的是，持久化技术的情况不是这样，这里我们会发现，同一个问题却有一些截然不同的解决方案。

几年来，持久化已经成了Java社区中一个热门的争论话题。许多开发人员甚至对这个问题的范围都存在不同意见。持久化是一个已经被关系技术及其扩展（例如存储过程，stored procedure）解决了的问题，还是一个必须用特别的Java组件模型（例如EJB实体bean）解决的更为普遍的问题？应该用SQL和JDBC手工编写哪怕是最原始的创建、读取、更新、删除（Create、Read、Update、Delete，CRUD）操作，还是应该让这项工作自动进行？如果每个数据库管理系统都有自己的SQL方言，那么如何实现可移植性？我们应该完全放弃SQL并采用一种不同的数据库技术（例如对象数据库系统）吗？争论还在继续，但是现在一种称作ORM的解决方案已经得到了广泛的认可。Hibernate就是一种开源的ORM服务实现。

Hibernate是一个目标远大的项目，旨在成为Java中管理持久化数据问题的一种完整的解决方案。它调解应用程序与关系数据库的交互，把开发人员解放出来，使他们把精力集中在手中的业务问题上。Hibernate是一种无干扰的解决方案。编写业务逻辑和持久化类时，不需要遵循许多Hibernate特定的规则和设计模式；因而，Hibernate可以顺利地与大多数新的和现有的应用程序进行整合，而不需要对应用程序的其他部分进行破坏性的改变。

本书是关于Hibernate的。我们将涵盖基础的和高级的特性，并介绍一些利用Hibernate开发新

应用程序的方法。但这些建议经常不特定于Hibernate。有时候，它们可能是我们使用持久化数据时关于最佳做事方式的想法，只不过是在Hibernate的环境中进行了阐述。本书也是关于Java Persistence的，它是持久化的一种新标准，也是最新EJB 3.0规范的一部分。Hibernate实现Java Persistence，并支持所有标准的映射、查询和API。然而，在我们可以开始使用Hibernate之前，你需要理解对象持久化和ORM的核心问题。本章阐述了需要像Hibernate这样的工具，以及像Java Persistence和EJB 3.0这样的规范的原因。

首先，我们定义面向对象的应用程序环境中持久化数据的管理，并讨论SQL、JDBC和Java的关系，Hibernate就是构建在这些基础的技术和标准之上。随后讨论所谓的*对象/关系范式不匹配*和我们在利用关系数据库进行面向对象的软件开发时所遇到的一般性问题。这些问题清楚地表明，我们需要一些工具和模式，把必须花在应用程序的持久化相关代码上的时间减到最少。在查看了可选择的工具和持久化机制之后，你会发现ORM对于许多场景都是最好的解决方案。我们对ORM优缺点的讨论，为你在给自己的项目选择持久化方案时，提供了做出最好决策的完整背景。

我们也探讨各种各样的Hibernate软件模块，以及如何把它们合并到只使用Hibernate，或者使用Java Persistence和EJB 3.0兼容的特性。

学习Hibernate的最好方式不一定是线性的。我们理解你可能想要立即尝试一下Hibernate。如果这是你喜欢的方式，那就跳到本书的第2章，看看“Hello World”示例，并建立一个项目。但是我们建议你在循环通读本书的某一时刻再回到这一章。这样，你就准备妥当，具备了学习其余内容所需的所有背景概念。

1.1 什么是持久化

几乎所有的应用程序都需要持久化数据。持久化在应用程序开发中是基本概念之一。如果一个信息系统在断电时没有保存数据，这个系统就没有什么实用价值了。当我们在Java中谈到持久化时，一般是指利用SQL在关系数据库中存储数据。我们先简单地看看这项技术，以及如何在Java中使用它。有了这个信息基础，再接着讨论持久化，以及如何在面向对象的应用程序中实现它。

1.1.1 关系数据库

就像大部分其他的开发人员一样，你可能已经使用过关系数据库。我们大部分人每天都在使用关系数据库。关系技术是个已知数，仅此一点就成为许多组织选择它的一个充分理由。但是只提这一点有些贬低了它应得的尊重。关系数据库的地位如此根深蒂固，是因为它们是一种出奇灵活和稳健的数据管理方法。由于关系数据模型完整且一致的理论基础，关系数据库可以有效保证和保护数据的完整性，这是它众多的优良特性之一。有些人甚至会说计算领域的最后一项大发明就是用于数据管理的关系概念，它由E.F Codd（Codd，1970）于30多年前首先提出。

关系数据库管理系统既不特定于Java，也不是一种特定于某个特殊应用程序的关系数据库。这个重要的原理就是*数据独立*（data independence）。换句话说，我们无法充分强调这个重要的事实：*数据比任何应用程序都存在得更长久*。关系技术提供了一种在不同应用程序或者构成同一应用程序（例如事务引擎和报告引擎）的不同技术之间共享数据的方式。关系技术是许多异构的系

统和技术平台的一个共同特性。因此，关系型数据模型经常是业务实体常用的企业级表示法。

关系数据库管理系统具有基于SQL的应用编程接口（Application Programming Interface，API）；因此，我们称当今的关系数据库产品为SQL数据库管理系统（database management system），或者当我们谈到特定系统时，称之为SQL数据库（database）。

在更详细地探讨SQL数据库应用程序方面之前，必须提到一个重要的问题：虽然有些产品也作为关系数据库销售，但是只提供SQL数据语言接口的数据库系统并不是真正的关系数据库，并且在很多方面甚至与原始概念相去甚远。自然，这样就导致了混乱。SQL从业者抱怨关系型数据模型在SQL语言方面的不足，而关系型数据管理专家则报怨SQL标准在关系模型和理念方面实现得不够。应用程序开发人员被夹在其中，承受着传送一些有效东西的压力。我们将在本书中始终强调有关这个问题的一些重要而有意义的方面，但是通常关注应用程序方面的。如果你有兴趣了解更多的背景资料，强烈推荐Fabian Pascal（Pascal，2000）所著的*Practical Issues in Database Management: A Reference for the Thinking Practitioner*。

1.1.2　理解 SQL

要有效地使用Hibernate，扎实地理解关系模型和SQL是前提条件。你需要理解关系模型，以及像保证数据完整性的标准化这样的话题，还要利用你的SQL知识调优Hibernate应用程序的性能。Hibernate让许多重复的编码任务自动化，但是如果要利用现代SQL数据库的全部功能，你的持久化技术必须扩充至超越Hibernate本身。记住，根本的目标是稳健、高效的持久化数据管理。

回顾一些本书中用到的SQL术语。你用SQL作为数据定义语言（Data Definition Language，DDL），用CREATE和ALTER语句创建数据库Schema。创建了表（和索引、序列等）之后，又用SQL作为数据操作语言（Data Manipulation Language，DML）来操作和获取数据。操作数据的操作包括插入（insertion）、更新（update）和删除（deletion）。通过限制（restriction）、投影（projection）和联结（join）操作（包括笛卡儿积，Cartesian product）执行查询来获取数据。为了有效地生成报表，可视需要使用SQL对数据进行分组（group）、排序（order）和统计（aggregate）。甚至可以相互嵌套SQL语句；这种技术使用了子查询（subselect）。

你可能已经使用SQL多年，熟悉这门语言的基本操作和语句编写。但我们从自身的经验中知道，有时候SQL仍然难以记住，而且一些术语的用法也很不同。要理解这本书，我们必须使用相同的术语和概念，因此如果我们提到的有些术语对你来说是陌生的或者不够清楚，建议你读一下附录A。

如果需要更多的资料，尤其是有关任何性能方面和SQL如何执行的，去找一本Dan Tow在2003年出版的优秀著作*SQL Tuning*。也看看Chris Date在2003年出版的著作*An Introduction to Database Systems*，了解（关系）数据库系统的理论、概念和思想。对于你在数据库和数据管理方面可能遇到的所有问题，后者是一本极好的参考书（很厚）。

虽然关系数据库是ORM的一部分，但是，另一部分却由Java应用程序中的对象组成，它们需要用SQL持久化到数据库中和从数据库中加载。

1.1.3 在 Java 中使用 SQL

在Java应用程序中使用SQL数据库时，Java代码通过Java数据库连通性（Java DataBase Connectivity，JDBC）API把SQL语句发到数据库。无论是手工编写SQL并嵌入到Java代码里面，还是由Java代码在运行中生成，都要用JDBC API绑定实参，来准备查询参数、执行查询、滚动查询结果表、从结果集中获取值，等等。这些都是底层的数据访问任务；作为应用程序开发人员，我们更关注需要这些数据访问的业务问题。我们真正想编写的是把对象的代码——类的实例——保存和获取到数据库，或者从数据库获取，使我们从这类底层的苦差事中解脱出来。

由于数据访问任务通常很单调乏味，我们不禁要问：关系型数据模型和（特别是）SQL都适合面向对象应用程序中的持久化吗？我们立即回答：是的！SQL数据库支配了计算行业有许多原因——关系数据库管理系统是唯一公认的数据管理技术，并且它们通常是任何Java项目的必要条件（requirement）。

然而，在过去的15年里，开发人员一直在谈论范式不匹配的问题。这种不匹配解释了为什么都要在每一个企业项目中与持久化相关的问题上付出如此巨大的努力。这里所说的范式（paradigm）是指对象模型和关系模型，或者可能是面向对象编程（Object-Oriented Programming，OOP）和SQL。

让我们通过询问在面向对象的应用程序开发环境中，持久化意味着什么，来开始对不匹配问题的探讨。首先，把本节开头所述的过分简化的持久化定义，扩展为在维护和使用持久化数据中对所涉及内容的一个更广泛、更成熟的理解。

1.1.4 面向对象应用程序中的持久化

在面向对象的应用程序中，持久化允许一个对象在创建之后依然存在。对象的这种状态可以被保存到磁盘，且相同状态的对象可以在未来的某个时候被重新创建。

这并非只限于单独的对象——整个关联对象网络也可以被持久化，且以后在一个新的进程中被重新创建。大多数对象并不是持化久的；瞬时（transient）对象的生命周期有限，由实例化它的进程的寿命所决定。几乎所有的Java应用程序都混合包含了持久对象和瞬时对象；因此，我们需要一个子系统来管理持久化数据。

现代的关系数据库为持久化数据提供了一个结构化的表示法，能够对数据进行操作、排序、搜索和统计。数据库管理系统负责管理并发性和数据的完整性；它们负责在多用户和多应用程序之间共享数据。它们通过已经利用约束实现的完整性规则来保证数据的完整性。数据库管理系统提供数据级的安全性。当我们在本书中讨论持久化时，考虑以下这些事情：

- 结构化数据的储存、组织和获取；
- 并发性和数据完整性；
- 数据共享。

特别是，我们正在使用领域模型的面向对象的应用程序环境中考虑这些问题。

使用领域模型的应用程序并不直接使用业务实体的表格式表示法；该应用程序有它自己的业

务实体的面向对象模型。例如，如果一个在线拍卖系统的数据库有ITEM和BID表，Java应用程序就会定义Item和Bid类。

然后，业务逻辑并不直接在SQL结果集的行和列上进行工作，而是与这个面向对象的领域模型及其作为关联对象网络的运行时实现进行交互。Bid的每个实例都引用一个拍卖Item，而且每个Item都可以有一个对Bid实例的引用集合。业务逻辑并不在数据库中执行(作为SQL存储过程)；而是在应用层的Java中实现的。这就允许业务逻辑使用高级的面向对象的概念，例如继承和多态。比如，我们可以使用众所周知的设计模式，如Strategy（策略）、Mediator（中介者）和Composite（组合）（Gamma等，1995），所有这些模式都依赖于多态的方法调用。

现在给你一个警告：并非所有的Java应用程序都以这种方式设计，它们也不应该只以这种方式设计。简单的应用程序不用领域模型可能更好。复杂的应用程序可能必须重用现有的存储过程。SQL和JDBC API对于纯表格式数据的处理堪称完美，并且JDBC的行集合（RowSet）使CRUD操作变得更容易了。使用持久化数据的表格式表示法很直接且易于理解。

然而，对于含有重要业务逻辑的应用程序来说，领域模型方法帮助明显改善代码的可重用性和可维护性。实际上，这两种策略都是常用和必需的。许多应用程序都需要执行修改大组数据、接近数据的过程。同时，在应用层中执行一般在线事务处理逻辑的面向对象的领域模型时，其他的应用程序模块可以从中受益。你需要一种有效地把持久化数据带近应用程序代码的方法。

如果我们再次考虑SQL和关系数据库，最终会发现两种范式之间的不匹配。SQL操作如投影和联结始终会导致结果数据的表格式表示法。[这就是传递闭包（transitive closure），关联操作的结果也始终是一种关联。] 这与Java应用程序中用来执行业务逻辑的关联对象网络大不相同。这些是根本不同的模型，而不只是把同一模型形象化的不同方式。

带着这些认识，就可以开始看一些问题了——有些已经十分了解，有些则还不太了解——必须通过一个结合了这两种数据表示法的应用程序来解决：一个面向对象的领域模型和一个持久化的关系模型。让我们深入探讨一下所谓的范式不匹配。

1.2　范式不匹配

对象/关系范式不匹配可以分为几个部分，我们将依次进行分析。让我们从独立于问题的一个简单示例开始探讨。构建它时，就会开始看到不匹配问题的出现。

假设你必须设计和实现一个在线电子商务应用程序。在这个应用程序中，需要用一个类表示一个系统用户的信息，用另一个类表示用户的账单的明细，如图1-1所示。

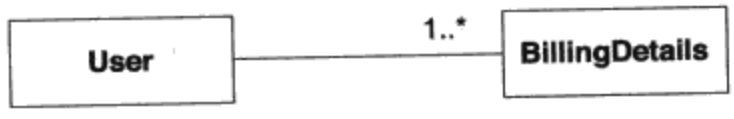


图1-1　User和BillingDetails实体的一个简单的UML类图

在这幅图中，可以看到一个User有许多BillingDetails。可以从两个方向遍历类之间的关系。表示这些实体的类可能极为简单：

```
public class User {
    private String username;
    private String name;
    private String address;
    private Set billingDetails;

    // Accessor methods (getter/setter), business methods, etc.
    ...
}
public class BillingDetails {
    private String accountNumber;
    private String accountName;
    private String accountType;
    private User user;

    // Accessor methods (getter/setter), business methods, etc.
    ...
}
```

注意，我们只关注与持久化有关的实体状态，因此省略了属性访问方法和业务方法的实现（例如getUsername()或者billAuction()）。

可以很容易地给这个例子想出一个好的SQL Schema设计：

```
create table USERS (
    USERNAME varchar(15) not null primary key,
    NAME varchar(50) not null,
    ADDRESS varchar(100)
)

create table BILLING_DETAILS (
    ACCOUNT_NUMBER varchar(10) not null primary key,
    ACCOUNT_NAME varchar(50) not null,
    ACCOUNT_TYPE varchar(2) not null,
    USERNAME varchar(15) foreign key references USER
)
```

这两个实体之间的关系表示为外键（foreign key），如BILLING_DETAILS中的USERNAME。对于这个简单的领域模型，对象/关系的不匹配几乎不明显；编写JDBC代码来插入、更新和删除关于用户和账单明细的信息很简明。

现在，看一下当我们考虑更现实一点的事情时会发生什么。当给应用程序添加更多的实体和实体关系时，范式不匹配的问题就显而易见了。

目前这个实现的最明显问题是把地址设计成了一个简单的String值。在大部分系统中，必需分别保存街道、城市、省/州、国家和邮政编码的信息。当然，可以直接给User类添加这些属性，但是由于系统中的其他类极有可能也含有地址信息，这使得创建一个单独的Address类变得更有意义了。更新过的模型如图1-2所示。

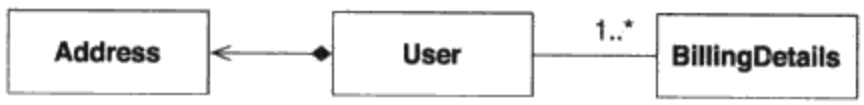


图1-2 User有一个Address

也应该添加一个ADDRESS表吗？没有必要。通常只需把地址信息保存在USERS表中单独的列

里。这种设计可能执行得更好，因为如果要在单个查询中获取用户和地址，就不需要表联结。最好的解决方案可能是创建一个用户定义的SQL数据类型来表示地址，在USERS表中使用新类型的单个列，而不用几个新列。

基本上，可以选择添加（一种新SQL数据类型的）几个列或者单个列。这明显是个粒度（granularity）问题。

1.2.1　粒度问题

粒度是指你正在使用的类型的相对大小。

回到我们的示例上。给数据库目录添加一个新的数据类型，在单个列中保存Address的Java实例，听起来像是最好的办法。Java中的一个新Address类型（类）和一个新的ADDRESS的SQL数据类型应该保证互用性。然而，如果检查当今的SQL数据库管理系统对UDT（User-defined DataType，用户定义的数据类型）的支持，就会发现各种各样的问题。

对传统SQL而言，UDT支持是许多所谓的*对象-关系扩展*（object-relational extension）之一。单独这个术语很费解，因为它意味着数据库管理系统有（或者假定要支持）一个完善的数据类型系统——有点像如果有人告诉你一个系统能够以关系的方式处理数据，你就信以为真。不幸的是，UDT支持是大部分SQL数据库管理系统的一个有点模糊的特性，当然不能在不同的系统之间移植。此外，SQL标准支持用户定义的数据类型，但是少得可怜。

这种局限性不是关系型数据模型的错。你可以将没有把如此重要的一项功能标准化的这一失误，归咎于20世纪90年代中期供应商之间的对象-关系型数据库之战。如今，大部分开发人员都接受类型系统有限的SQL产品——没有要询问的问题。然而，即使SQL数据库管理系统中使用了完善的UDT系统，我们仍然可能重复类型声明，在Java中编写新类型，并在SQL中再次重复。试图给Java空间找到一种解决方案，例如SQLJ，不幸的是还没有成功。

基于这些或者任何其他原因，在SQL数据库内部使用UDT或者Java类型，都还不是目前行业的一项共通的实践，你不可能遇到一个广泛使用UDT的遗留Schema。因此我们不能并且也不会在单个有着与Java层相同的数据类型的新列中保存新Address类的实例。

对于这个问题，务实的解决方案是使用几列内建的供应商定义的SQL类型（例如布尔、数值和字符串数据类型）。USERS表通常定义如下：

```
create table USERS (
    USERNAME varchar(15) not null primary key,
    NAME varchar(50) not null,
    ADDRESS_STREET varchar(50),
    ADDRESS_CITY varchar(15),
    ADDRESS_STATE varchar(15),
    ADDRESS_ZIPCODE varchar(5),
    ADDRESS_COUNTRY varchar(15)
)
```

领域模型中的类按照各种不同的粒度等级排列起来——从粗粒度实体类如User，到细粒度类如Address，直到简单String值的属性如zipcode。相比之下，在SQL数据库等级中只有两种粒

度等级是可见的：表（如USERS）和列（如ADDRESS_ZIPCODE）。

许多简单的持久化机制无法识别这种不匹配，因此不再强制对象模型上更不灵活的SQL表示。我们已经看到无数User类中包含名为zipcode的属性！

这证实了粒度问题并非特别难以解决。如果不是因为它在许多现有系统中很明显的话，我们甚至可能不讨论它。4.4节描述了这个问题的解决方案。

当考虑依赖继承（inheritance）的领域模型时，一种更复杂和更值得关注的问题出现了。继承是面向对象设计的一个特性，可用来以一种更新更有趣的方式给电子商务应用程序中的用户开账单。

1.2.2 子类型问题

在Java中，用超类（superclass）和子类（subclass）实现类型继承。为了举例说明为什么这会出现不匹配的问题，我们来添加到电子商务应用程序中，以便现在不仅可以接受银行账户的账单，还可以接受信用卡和借记卡。在该模型中体现这种变化的最自然方式是给BillingDetails类使用继承。

我们可能有一个抽象的BillingDetails超类，以及几个具体的子类：CreditCard、BankAccount等。每个子类都定义了略微不同的数据（和处理这些数据的完全不同的功能）。图1-3中的UML类图说明了这个模型。

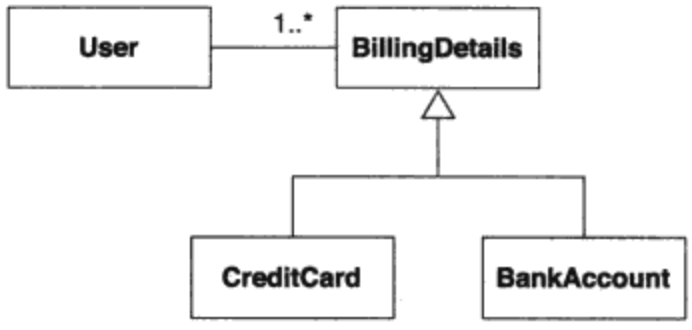


图1-3 为不同的记账策略使用继承

SQL或许应该包括对超表（supertable）和子表（subtable）的标准支持。这实际上支持我们创建从父表中继承某些列的表。然而，这样的一种特性备受置疑，因为它会在基表（base table）中引进一个新的概念：虚拟列（virtual column）。传统上，我们期待虚拟列仅存在于虚拟表（也称作视图，view）中。此外，理论上来说，应用在Java中的继承是类型继承（type inheritance）。表不是一种类型，因此超表和子表的概念令人置疑。无论是哪种情况，此处我们都可以走个捷径，并且发现SQL数据库产品一般不实现类型或者表继承，即使它们真的实现了，也不遵循标准的语法，并且通常会使你遭遇到数据完整性的问题（对可更新视图的有限完整性规则）。

5.1节将讨论ORM解决方案（例如Hibernate）如何解决把一个类层次结构持久化到一个或者多个数据库表的问题。这个问题现在社区中已经被很好地理解了，并且大部分解决方案支持几乎相同的功能。

但是我们还没有结束对继承的讨论。一旦把继承引进到模型中，就有了多态（polymorphism）

的可能。

User类和BillingDetails超类有一个关联。这是个多态关联（polymorphic association）。运行时，User对象可以引用BillingDetails任何子类的一个实例。类似地，我们想要能够编写引用BillingDetails类的多态查询（polymorphic query），并让查询返回它子类的实例。

SQL数据库也缺乏一种明显的表示多态关联的方式（或者至少是一种标准化的方式）。一个外键约束精确地引用一张目标表（target table）；定义一个引用多表的外键不容易。必须编写一个程序化的约束来加强这种完整性规则。

子类型的这种不匹配的结果是，模型中的继承结构必须在一个不提供继承策略的SQL数据库中被持久化。庆幸的是，第5章介绍的继承映射解决方案中，有3种被设计为适应多态关联的表示法和多态查询的有效执行。

对象/关系不匹配问题的另一个方面是对象同一性（object identity）。你可能注意到了我们把USERNAME定义为USERS表的主键。这是个好办法吗？我们如何处理Java中的同一对象？

1.2.3　同一性问题

虽然对象同一性问题一开始可能并不明显，但我们经常在日渐增长和扩展的电子商务系统中遇到，例如当需要检查两个对象是否为同一对象的时候。解决这个问题有3种方法：2种使用Java，1种使用SQL数据库。不出所料，它们只要一点点帮助就可以协同工作了。

Java对象定义两个不同的同一性（sameness）概念：

- 对象同一性（粗略等同于内存位置，用a == b检查）
- 等同性，通过equals()方法（也称作值等同，equality by value）的实现来确定。

另一方面，数据库行的同一性用主键值表达。如9.2节所述，equals()和==都不会必然等于主键值。几个不恒等的对象同时表示数据库的同一行很常见，例如，在并发运行的应用程序线程中。此外，给持久化类正确实现equals()包含了一些微妙的困难。

让我们用一个示例来讨论另一个有关数据库同一性的问题。在USERS表的定义中，我们使用了USERNAME作为主键。不幸的是，这个决定使得用户名变得很难改变；我们不仅要更新USERS中的USERNAME列，还要更新BILLING_DETAILS中的外键列。为解决这个问题，本书后面推荐使用代理键（surrogate key），每当你无法找到一个好的自然键（natural key）时（我们也将讨论使得一个键成为好键的因素）。代理键列是个对用户没有意义的主键列；换句话说，它是个不呈现给用户的键，而只用作软件系统内部的数据识别。例如，可以把表定义变成这样：

```
create table USERS (
    USER_ID bigint not null primary key,
    USERNAME varchar(15) not null unique,
    NAME varchar(50) not null,
    ...
)
create table BILLING_DETAILS (
    BILLING_DETAILS_ID bigint not null primary key,
    ACCOUNT_NUMBER VARCHAR(10) not null unique,
    ACCOUNT_NAME VARCHAR(50) not null,
```

```
    ACCOUNT_TYPE VARCHAR(2) not null,
    USER_ID bigint foreign key references USERS
)
```

USER_ID和BILLING_DETAILS_ID列包含系统生成的值。这些列是纯粹为了方便数据模型而引入的，因此它们在领域模型中该如何（如果要）表示？4.2节将讨论这个问题，并且找到一种使用ORM的解决方案。

在持久化的上下文中，同一性与系统如何处理高速缓存和事务密切相关。不同的持久化方案选择不同的策略，这已经成了混乱的一个方面。我们将在第10章和第13章涵盖所有这些令人关注的话题以及它们是如何相关的。

目前为止，我们设计的构架电子商务应用程序已经识别了映射粒度、子类型和对象同一性的不匹配问题。我们差不多准备转向应用程序的其他部分了，但是首先需要讨论一下关联（association）这个重要的概念：类之间的关系如何被映射和处理。数据库中的外键就是你所需要的一切吗？

1.2.4 与关联相关的问题

在领域模型中，关联表示实体之间的关系。User、Address和BillingDetails类都是相互关联的；但是不像Address、BillingDetails那么独立。BillingDetails实例则保存在它们自己的表中。关联映射和实体关联的管理在任何对象持久化解决方案中都是重要概念。

面向对象的语言利用对象引用（object reference）表示关联；但是在关系领域中，关联则被表示为外键（foreign key）列，带有几个键值的复本（和一个保证完整性的约束）。这两种表示法之间有着本质的区别。

对象引用具有固有的方向性；关联是从一个对象到另一个对象。它们是指针。如果对象之间的关联应该在两个方向导航，就必须定义两次关联，在每个关联的类中定义一次。你已经在领域模型类中见过：

```
public class User {
    private Set billingDetails;
    ...
}
public class BillingDetails {
    private User user;
    ...
}
```

另一方面，外键关联不是生来就有方向性。导航（navigation）对于关系型数据模型没有任何意义，因为可以用表联结（table join）和投影创建任意的数据关联。具有挑战性的是，把一个完全开放的数据模型——独立于使用数据的应用程序——桥接到一个应用程序依赖的导航模型，这个特定的应用程序需要关联的一个约束视图。

不可能只看Java类就确定一个单向关联的多样性。Java关联可以有多对多（many-to-many）的多样性。例如，类可以看起来像下面这样：

```
public class User {
    private Set billingDetails;
    ...
}
public class BillingDetails {
    private Set users;
    ...
}
```

另一方面，表关联始终是一对多（one-to-many）或者一对一（one-to-one）。通过查看外键的定义，你立即就会知道多样性。以下是给一对多关联（或者，如果从另一个方向看，是多对一的）在BILLING_DETAILS表中的一个外键声明：

```
USER_ID bigint foreign key references USERS
```

这些是一对一的关联：

```
USER_ID bigint unique foreign key references USERS
BILLING_DETAILS_ID bigint primary key foreign key references USERS
```

如果你想在一个关系数据库中表示一个多对多的关联，就必须引入一张新表，称作链接表（link table）。这个表不会出现在领域模型中。对于我们的例子来说，如果认为用户和账单信息之间的关系为多对多，链接表就可以定义如下：

```
create table USER_BILLING_DETAILS (
    USER_ID bigint foreign key references USERS,
    BILLING_DETAILS_ID bigint foreign key references BILLING_DETAILS,
    PRIMARY KEY (USER_ID, BILLING_DETAILS_ID)
)
```

第6章和第7章将详细讨论关联和集合映射。

目前为止，我们考虑的主要是结构性的问题。通过考虑系统的一个纯静态视图就会明白。也许对象持久化中最困难的问题就是动态的（dynamic）问题。它涉及关联，在1.1.4节中介绍对象网络导航（object network navigation）和表联结之间的区别时，就已经提示了这一点。我们来更深入地探讨这个重要的不匹配问题。

1.2.5　数据导航的问题

在Java和在关系数据库中访问数据的方式有着根本的区别。在Java中，当你访问用户的账单信息时，调用`aUser.getBillingDetails().getAccountNumber()`或者类似的东西。这是访问面向对象的数据最自然的方式，通常称作遍历对象网络。跟着实例之间的指针，从一个对象导航到另一个对象。不幸的是，这并不是从SQL数据库中获取对象的有效方法。

为了提高数据访问代码性能，你能做的最重要的事情就是将请求数据库的次数减到最少。最明显的做法是将SQL查询的次数减到最少。（当然，第二步还有其他更复杂的方法。）

因此，使用SQL有效地访问关系型数据通常需要在有关的表之间使用联结。获取数据时联结中包括的表数目决定了能够在内存中遍历对象网络的深度。例如，如果需要获取一个`User`，而对用户的账单明细不感兴趣，可以编写这个简单的查询：

```
select * from USERS u where u.USER_ID = 123
```

另一方面，如果需要获取一个User，随后访问每个关联的BillingDetails实例（比如，列出用户的所有信用卡），可以编写一个不同的查询：

```
select *
    from USERS u
    left outer join BILLING_DETAILS bd on bd.USER_ID = u.USER_ID
    where u.USER_ID = 123
```

如你所见，获取初始User时，要有效使用联结就要知道你计划访问对象网络的哪个部分——在你开始遍历对象网络之前！

另一方面，只有当对象被初次访问时，所有对象持久化解决方案才提供抓取关联对象的数据的功能。然而，这种渐进风格（piecemeal）的数据访问在关系数据库的上下文中效率很低，因为它要给每个节点或者每个被访问的对象网络的集合执行一条语句。这就是可怕的n+1查询问题（n+1 select problem）。

在Java和关系数据库中访问对象的这种不匹配，可能是Java应用程序中性能问题的一个最普遍的根源。在太多的选择和太大的选择之间有个自然压力，它获取不需要的信息到内存中。然而，虽然已经有无数的图书和杂志文章建议我们对字符串拼接使用StringBuffer，但是似乎不可能找到任何有关避免n+1查询问题的建议。幸运的是，Hibernate提供了从数据库中有效地、透明地抓取对象网络到访问它们的应用程序中的成熟特性。第13章～第15章将讨论这些特性。

1.2.6 不匹配的代价

现在已经有了相当一些对象/关系不匹配问题，你可能凭经验就知道，找到这些解决方案的代价不菲（时间和精力）。这种代价通常被低估了，这是许多软件项目失败的主要原因。依据我们的经验（也得到受访开发人员的一致确认），有30%的Java应用程序代码编写是用来处理乏味的SQL/JDBC和手工桥接对象/关系范式的不匹配。尽管付出了这么多努力，最终的结果仍然不尽如人意。我们曾经见过由于数据库抽象层的复杂性和低灵活性而几乎垮掉的项目。我们也见到Java开发人员（和DBA）在必须给项目做出有关持久化策略的设计决定时很快失去了信心。

主要的成本之一在于模型化方面。关系和领域模型都必须包含相同的业务实体，但是一位面向对象的纯化论者给这些实体建模的方法，与一位经验丰富的关系型数据建模者给出的不同。这个问题通常的解决方案是扭曲领域模型和被实现的类，直到它们与SQL数据库Schema相匹配。（遵循数据独立的原则，必定是个安全的长久之计。）

这可能会成功，但要以失去面向对象的一些优势为代价。记住，关系模型以关系理论为基础。面向对象则没有这样严格的数学定义或者理论主体，因此我们无法用数学来解释应该如何为这两种范式建立起某种关系——没有优雅的转化等着被发现。（放弃Java和SQL从头开始的做法并不算优雅。）

领域模型不匹配并不是低灵活性和导致更高成本的低生产力的唯一根源。更深层次的原因是JDBC API本身。JDBC和SQL提供了一个*面向语句*（statement-oriented，即面向命令，command-

oriented）的方法，把数据从SQL数据库移进移出。如果要查询或者操作数据，涉及的表和列至少必须被指定3次（插入，更新，选择），这增加了设计和实现所需要的时间。每个SQL数据库管理系统的独特方言并没有改善这种情形。

为了巩固你对对象持久化的理解，在探讨可能的解决方案之前，需要讨论应用程序架构（application architecture）和持久层（persistence layer）在典型应用程序设计中的作用。

1.3　持久层和其他层

在一个中型或者大型应用程序中，按关注点组织类通常比较有意义。持久化是一个关注点；其他还包括表现、工作流和业务逻辑[①]。一个典型的面向对象的架构包括表示这些关注点的代码层。标准的且必定最好的实践是，把所有的类和负责持久化的组件都组合到一个分层系统架构中一个单独的持久层。

在本节中，我们先来看看这类架构的层，以及为什么要使用它们。随后，集中在我们最关注的那个层——持久层——以及可以实现它的几种方法。

1.3.1　分层架构

分层架构定义了实现不同关注点的代码之间的接口，允许改变实现关注点的方法，无需明显破坏其他层中的代码。分层也决定了产生中间层依赖的种类。规则如下：

- 各层交流从上到下，一个层仅仅依赖于就在它下方的那个层。
- 每个层都不知道除了它下方的层之外的其他层。

不同的系统对关注点的组合不同，因此它们定义不同的层。一个典型、公认的高级应用程序架构使用3个层，分别用于表现、业务逻辑和持久化，如图1-4所示。

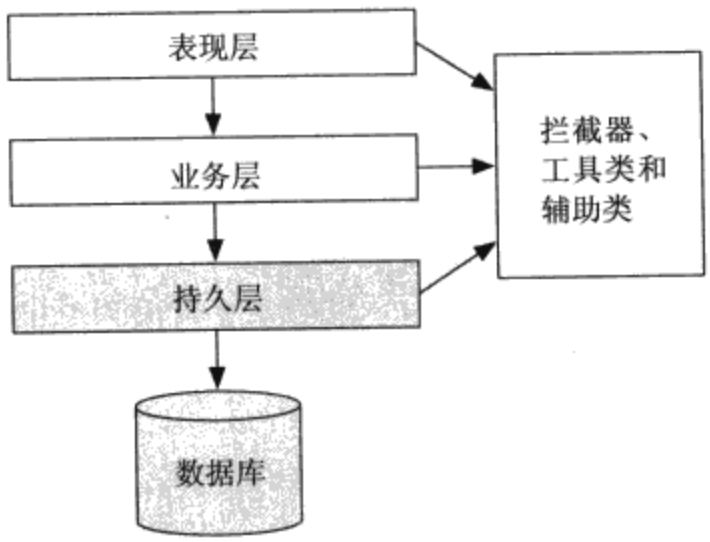


图1-4　持久层是层架构的基础

让我们深入看看图中的层和元素：

① 也有所谓的横切（cross-cutting）关注点，可被一般地实现——例如按框架代码。典型的横切关注点包括日志、授权和事务划分。

- 表现层——用户界面逻辑在最顶层。负责页面以及屏幕导航的表现和控制的代码在表现层。
- 业务层——下一层的具体形式在不同的应用程序之间很不相同。然而，业务层负责实现任何业务规则或者会被用户理解为问题领域一部分的系统需求，这一点已经得到广泛认同。这一层通常包括几种控制组件——知道何时调用哪个业务规则的代码。在有些系统中，这一层对业务领域实体有着自己的内部表示，在其他系统中它重用由持久层定义的模型。第3章会继续讨论这个问题。
- 持久层——持久层是负责向（或者从）一个或者多个数据存储器中存储（或者获取）数据的一组类和组件。这个层必须包括一个业务领域实体的模型（即使只是一个元数据模型）。
- 数据库——数据库存在于Java应用程序本身之外。它是系统状态实际的、持久化的表示。如果使用了一个SQL数据库，数据库就包括关系Schema和可能的存储过程。
- 辅助和工具类——每个应用程序都有一组基础的辅助和工具类，用在应用程序的每一个层中（例如用于错误处理的Exception类）。这些基础元素不会形成一个层，因为它们不遵循分层架构中的中间层依赖的规则。

现在我们来简单地看看可以由Java应用程序实现持久层的各种方法。别着急，我们很快就要讲到ORM和Hibernate了。通过看看其他方法，可以学到很多东西。

1.3.2 用 SQL/JDBC 手工编写持久层

对于应用程序员来说，处理Java持久化最常用的方法是直接使用SQL和JDBC。毕竟，开发人员熟悉关系数据库管理系统，他们理解SQL，并且知道如何使用表和外键。此外，他们始终可以使用众所周知且被广泛使用的DAO模式，来隐藏业务逻辑中复杂的JDBC代码和不可移植的SQL。

DAO是个很好的模式——因此我们甚至经常推荐把它与ORM一起使用。然而，给每个领域类手工编写持久化代码涉及的工作量相当大，特别当支持多个SQL方言的时候。这项工作通常消耗很大一部分开发精力。甚至，当需求改变时，手工编写的解决方案总是需要更多的注意力和维护精力。

为什么不实现一个简单的映射框架来适应项目的特殊需求呢？这种努力的结果甚至可以在未来的项目中重用。许多开发人员已经采用了这种方法；如今许多自制的对象/关系持久层应用于产品系统中。但是我们不推荐这种方法。优秀的解决方案已经存在：不仅有商业供应商销售的工具（非常昂贵），也有免许可的开源项目。我们确信你肯定能够找到一种满足你需求的解决方案，包括业务需要和技术需求。这种解决方案可能比你在有限的时间内能够创建的解决方案完成更多的工作，并且做得更好。

开发一套适度的全特性ORM可能要花费许多开发人员几个月的时间。例如，Hibernate大约有80 000行代码，其中有些比典型的应用程序代码更难，还有25 000行单元测试代码。这可能比你的应用程序中的代码还多。在这样一个大项目中，可能很容易忽略大量的细节——依据我们两位作者的经验！即使一个现有的工具没有完全实现你其中的两三个更奇异的需求，也仍然可能不

值得去创建自己的工具。任何ORM软件都会处理单调的普遍的案例——正是削减生产力的那些案例。如果你需要手工编写某些特别的案例也可以；但是很少有应用程序是主要由特殊的案例组成的。

1.3.3　使用序列化

Java有一个内建的持久化机制：序列化提供了把对象网络（应用程序的状态）快照写到字节流的能力，然后它可能被持久化到一个文件或者数据库中。序列化也被Java的远程方法调用（Remote Method Invocation，RMI）用于给复杂的对象实现传递（pass-by）值语义。序列化的另一种用法是在集群机器中跨节点复制应用程序状态。

为什么不给持久层使用序列化？不幸的是，序列化之后的关联对象网络只能被当作一个整体访问；没有反序列化整个流，就不可能从流中获取任何数据。因而，结果字节流必须被当作不适合进行任意的搜索或者大型数据集的统计，甚至不可能独立地访问或者更新单个对象或者对象的子集。对于设计用来支持高并发性的系统来说，除了在每个事务中加载和覆盖整个对象网络外，别无他法。

鉴于当前的技术，序列化还不足以作为高并发性的Web和企业应用程序的持久化机制。序列化有一项特别的作用，是桌面应用程序最适当的持久化机制。

1.3.4　面向对象的数据库系统

由于我们是在Java中使用对象，所以如果有一种方法能把那些对象存储到数据库中、又根本不必扭曲对象模型就好了。在20世纪90年代中期，面向对象的数据库系统备受关注。它们以一个网络数据模型为基础，在关系型数据模型出现之前的十几年前，这种方法很普遍。基本的思想是存储一个对象网络，包括它所有的指针和节点，并随后重新创建相同的内存图。这可以用各种元数据和配置设置进行优化。

与其说是像外部数据仓库，面向对象的数据库管理系统（OODBMS）则更像是应用程序环境的一个扩展。OODBMS通常以一个多层实现作为主要特征，包含一个后端数据仓库、对象高速缓存，以及一个紧密结合的通过一个私有网络协议进行交互的客户端应用。对象节点保存在内存页面中，该内存页面向（或者从）数据仓库中进行传输。

面向对象的数据库开发从宿主语言绑定的自顶向下的定义开始，这些绑定给编程语言增加了持久化的能力。因此，对象数据库提供了与面向对象应用程序环境的无缝整合。这不同于当今的关系数据库所用的模型，在这里，与数据库的交互通过一种中间语言（SQL）进行，并且独立于特定应用程序的数据是主要的关注点。

有关面向对象数据库的背景信息，我们在*An Introduction to Database Systems*（2003）中推荐有相应的章节。

我们不想深入研究为什么面向对象的数据库技术还没有流行起来；我们将发现对象数据库还没有被广泛采用，可能近期内也不会。考虑到当前的政治现实（预先确定的部署环境）和数据独立的普遍需求，我们相信压倒多数的开发人员会有更多的机会使用关系技术。

1.3.5 其他选项

当然，也有其他类型的持久层。XML持久化是序列化模式的变形；这种方法允许通过一个标准化的工具接口来轻松地访问数据，从而解决了字节流序列化的一些限制。然而，在XML中管理数据会使你遭受一个对象/层次结构不匹配的危险。此外，XML本身也没有其他益处，因为它只是另一种文本文件格式，对数据管理没有继承能力。可以使用存储过程（有时候甚至可以在Java中编写），把问题移进数据库层。所谓的对象-关系数据库已经作为一个解决方案销售，但是它们只是提供一个更完善的数据类型系统，对于我们的问题只提供半个解决方案（和更加费解的术语）。我们确信还有许多其他的示例，但没有一种可能很快变得普及起来。

政治和经济上的约束（在SQL数据库上的长期投资），数据独立和访问有价值遗留数据的需求都要求一种不同的方法。针对这些问题，ORM可能是最实用的解决方案。

1.4 ORM

看过了对象持久化可供选择的方法之后，该介绍我们认为是最好的、Hibernate使用的一种解决方案ORM了。除了ORM的悠久历史之外（最早的研究论文发表于20世纪80年代末期），开发人员对ORM术语的使用也各不相同。有些人称它为“对象关系映射”，其他人则喜欢简单的“对象映射”；我们专门使用术语“对象/关系映射”和它的首字母（Object/Relational Mapping）缩写词ORM。正斜杠强调了当这两个领域整合时产生的不匹配问题。

本节首先看看什么是ORM。然后列举出一个好的ORM解决方案需要解决的问题。最后，讨论ORM提供的大致的好处，以及我们为什么推荐这种解决方案。

1.4.1 什么是 ORM

简而言之，ORM就是利用描述对象和数据库之间映射的元数据，自动（且透明）地把Java应用程序中的对象持久化到关系数据库中的表。

ORM本质上是把数据从一种表示法（可逆）转换为另一种表示法进行工作。这意味着某些性能损失。然而，如果ORM作为中间件实现，就有许多手工编码的持久层所没有的优化机会。控制转换的元数据的规定和管理在开发时增加了企业日常开支，但是其成本却少于维护一个手工编码的解决方案所需的成本。（甚至对象数据库需要大量的元数据。）

常见问题 ORM难道不是一个Visio插件吗？首字母缩写的ORM也可以是对象角色建模（Object Role Modeling）的意思，且这个术语出现在相关的ORM之前。它描述了一种在数据库建模中使用的信息分析方法，主要由Microsoft Visio（一种图形化建模工具）支持。数据库专家们用它作为最普及的实体-关系建模（Entity-Relationship Modeling）的替代或者补充。然而，如果你和Java开发人员谈ORM，通常是在ORM的环境中。

ORM解决方案包含下面的4个部分：

- 在持久化类的对象上执行基本的CRUD操作的一个API；
- 用于指定引用类或者类属性的查询的一种语言或者API；
- 用于指定映射元数据的一种工具；
- 用于实现ORM的一项技术，与事务对象交互，执行脏检查（dirty checking）、延迟关联抓取以及其他优化功能。

我们使用完整的ORM术语，包括从一个基于元数据的描述中自动产生SQL的任何持久层。我们不包括开发人员用JDBC手工编写SQL来手动解决ORM问题的持久层。使用ORM，应用程序与ORM API和领域模型类交互，并从底层的SQL/JDBC中被抽象出来。依赖于这些特性或者特定的实现，ORM引擎也可能承担如乐观锁（optimistic locking）和高速缓存这类问题，完全免去了应用程序对这些问题的关注。

来看一下可以实现ORM的各种方法。Mark Fussel（Fussel，1997），ORM领域的一位开发人员，定义了下列4个ORM质量等级。我们稍微改写了他的描述，并把它们放在当今Java应用程序开发的上下文中。

1．纯关系

整个应用程序（包括用户界面）都围绕着关系模型和基于SQL的关系操作而设计。这种方法，除了它不足以用于大型系统之外，对于那些容许低级代码重用的简单应用程序来说，它不失为一种极好的解决方案。直接的SQL可以在各个方面进行调优，但是缺点（例如缺乏可移植性和可维护性）也是很显著的，尤其对长期运行而言。这类应用程序经常大量地使用存储过程，把一些工作从业务层转移到了数据库中。

2．轻量对象映射

实体被表示为手工映射到关系表的类。使用众所周知的设计模式，把手工编码的SQL/JDBC从业务逻辑中隐藏起来。这种方法非常普遍，对于那些带有少量实体的应用程序，或者那些使用普通的元数据驱动的数据模型的应用程序来说，它是很成功的。存储过程在这种类型的应用程序中可能也有一席之地。

3．中等对象映射

这种应用程序围绕对象模型而设计。SQL使用一个代码生成的工具在创建时产生，或者通过框架代码在运行时产生。对象之间的关联得到持久化机制的支持，并且查询可能使用一种面向对象的表达式语言来指定。对象由持久层高速缓存。ORM产品和自制的持久层都至少支持这一级别的功能。它非常适合一些复杂事务的中等规模的应用程序，特别是在不同的数据库产品之间的可移植性很重要的时候。这些应用程序通常不使用存储过程。

4．完全对象映射

完全的对象映射支持完善的对象模型：组合、继承、多态和可达的持久化。持久层实现了透明的持久化；持久化类不必继承任何特殊的基类，或者实现特殊的接口。高效的抓取策略（延迟、即时和预取）和高速缓存策略被透明地实现到应用程序。这一级别的功能无法通过自制的持久层实现——它相当于几年的开发时间。许多商业的和开源的Java ORM工具已经实现了这个质量等级。

这个等级符合本书中使用的ORM定义。来看一下希望通过实现完全对象映射的一个工具能够得以解决的一些问题。

1.4.2 一般的 ORM 问题

下列问题（称作ORM问题）列表标识了Java环境中被完全的ORM工具解决的一些根本问题。特别的ORM工具可能提供额外的功能（例如积极高速缓存），但这是个相当详尽的概念问题和特定于ORM问题的列表。

(1) 持久化类看起来什么样？持久化工具有多透明？我们必须对业务领域的类采用编程模型和惯例吗？

(2) 映射元数据如何定义？由于对象/关系转换完全由元数据控制，这个元数据的格式和定义很重要。ORM工具应该提供图形化用户界面（GUI）以便图形化地处理元数据吗？或者对于元数据的定义有没有更好的方法？

(3) 对象同一性和等同性如何与数据库（主键）同一性相关？如何映射特定类的实例到特定表的行？

(4) 应该如何映射类继承层次结构？有几种标准的策略。多态关联、抽象类和接口怎么样呢？

(5) 持久化逻辑如何在运行时与业务领域的对象交互？这是个一般的编程问题，并且有许多解决方案，包括源代码生成、运行时反射、运行时字节码生成和创建时字节码增强。这个问题的解决方案可能影响你的构建过程（但宁可如此，因为影响构建过程总好过影响用户）。

(6) 什么是持久化对象的生命周期？有些对象的生命周期取决于其他关联对象的生命周期吗？如何把一个对象的生命周期转变为一个数据库行的生命周期？

(7) 提供什么工具用来排序、搜索和统计？应用程序可以在内存中处理其中一些事情，但是为了有效地使用关系技术，经常需要由数据库来完成这项工作。

(8) 如何利用关联有效地获取数据？对关系型数据的有效访问通常经由表联结来完成。面向对象应用程序通常通过导航对象网络来访问数据。如果可能，这两种数据访问模式应该避免：n+1查询问题和它的补充笛卡儿积问题（在单个查询中抓取太多的数据）。

在一个ORM工具的设计和架构上强加基础约束的另外两个问题，对于任何数据访问技术都是共通的：

- 事务和并发性；
- 高速缓存管理（和并发性）。

如你所见，一个完全的ORM工具需要处理一个相当长的问题列表。现在为止，你应该开始体会到ORM的价值了。下一节介绍使用ORM解决方案所获得的一些其他益处。

1.4.3 为什么选择 ORM

ORM的实现非常复杂——虽然没有应用程序服务器复杂，却比Web应用程序框架如Struts或者Tapestry要复杂得多。为什么要把另一个复杂的基础元素引入到我们的系统中呢？值得这么做吗？

给这些问题提供完整的答案，将要占用本书大部分的篇幅，但是本节只对最具竞争力的益处提供一个快速的概括。可是，首先要迅速处理一个非益处的问题。

ORM一个假定的益处是使开发人员避免杂乱的SQL。持这种观点的人认为不能期待面向对象的开发人员很好地理解SQL或者关系数据库，并且他们认为SQL有点讨厌。正好相反，我们认为Java开发人员必须足够熟悉并欣赏关系模型和SQL，以便用ORM进行工作。ORM是一项高级的技术，将被为其付出艰辛努力的开发人员所用。要有效地使用Hibernate，必须能够观察和解读它生成的SQL语句，并理解对于其性能的含义。

现在，来看看ORM和Hibernate的一些益处。

1．生产力

与持久化相关的代码可能会是Java应用程序中最冗长乏味的代码。Hibernate去除了许多琐碎的工作（比你想象的更多），并让你把精力集中在业务问题上。

无论你喜欢哪种应用程序开发策略——自顶向下，从一个领域模型开始；或者自底向上，从一个现有的数据库Schema开始——Hibernate与适当的工具一起使用，将明显减少开发时间。

2．可维护性

更少的代码行（LOC）使得系统更易于理解，因为它强调业务逻辑甚于那些费力的基础性工作。最重要的是，系统包含的代码越少则越易于重构。自动的对象/关系持久化充分地减少了LOC。当然，统计代码行是衡量应用程序复杂性的一种有争议的方式。

然而，Hibernate应用程序更易维护还有其他原因。在手工编码的持久化系统中，关系表示法和对象模型实现领域之间存在着一种必然的压力。改变一个，通常都要改变另一个，并且一个表示法设计经常需要妥协以便适应另一个的存在。（在实际应用程序中，通常是领域的对象模型发生妥协。）ORM提供了两个模型之间的一个缓冲，允许面向对象在Java方面进行更优雅的利用，并且每个模型的微小变化都不会传递到另一个模型。

3．性能

一种普遍的断言是，手工编码的持久化与自动的持久化相比总是至少可以一样快，并且经常更快。这是真的，就像汇编代码总是至少可以与Java代码一样快，或者手工编写的解析器总是至少可以与由YACC或者ANTLR产生的解析器一样快，这同样是真的一样——换句话说，这有点离题了。这种断言的言下之意是，手工编码的持久化在实际应用程序中将至少完成得一样好。但是，这种含意只有当实现至少一样快的手工编码的持久化所需的努力，类似于使用自动的解决方案所付出的努力时才是对的。真正值得关注的问题是，当我们考虑到时间和预算的约束时会发生什么？

给定一项持久化任务，有多种优化可能。有些（例如查询提示）用手工编码的SQL/JDBC更容易实现。然而，大部分优化用自动的ORM则更容易实现。在有时间限制的项目中，手工编码的持久化通常允许你进行一些优化。Hibernate始终允许使用更多的优化。此外，自动的持久化把开发人员的工作效率提高了那么多，使得开发人员能够花更多的时间对其他的少数瓶颈进行手工优化。

最后，实现你的ORM软件的人，可能比你更有时间研究性能优化问题。例如，你知道高速

缓存PreparedStatement实例给DB2 JDBC驱动带来明显的性能提升，却破坏了InterBase JDBC驱动吗？你认识到只更新表中被改变的列对于有些数据库会明显变快，却潜在地减慢了其他的数据库吗？在你手工编写的解决方案中，试验这些不同策略之间的冲突容易吗？

4．供应商独立性

ORM从底层的SQL数据库和SQL方言中把应用程序抽象出来。如果这个工具支持许多不同的数据库（大部分都支持），那么这会给你的应用程序带来一定程度的可移植性。你不必期待一劳永逸（一次编写/到处运行），因为数据库的能力各异，实现完全的可移植性将需要牺牲这个更强大平台的一些优势。然而，用ORM通常更容易开发跨平台的应用程序。即使你不需要跨平台操作，ORM仍然可以帮助减小一些被供应商锁定的风险。

此外，数据库独立性在这种开发场景中也有帮助——开发人员在开发时使用轻量级的本地数据库，但实际的产品部署在不同的数据库上。

有时候，需要选择一种ORM产品。为了做出有根据的决定，需要一张可用的软件模块和标准的列表。

1.4.4 Hibernate、EJB 3 和 JPA 简介

Hibernate是一个完全的ORM工具，提供前面列举的所有ORM的益处。在Hibernate中使用的API是原生的，并且是由Hibernate的开发人员设计的。对于查询接口和查询语言，以及ORM元数据如何定义，这也是一样的。

在用Hibernate开始第一个项目之前，应该考虑EJB 3.0标准和它的子规范Java Persistence。让我们回顾历史看看这个新标准是如何产生的。

许多Java开发人员认为EJB 2.1实体bean是持久层实现的技术之一。EJB编程和持久化模型在行业中已经被广泛采用，并且已经成了J2EE（或者现在称作Java EE）成功的一个重要因素。

然而，过去几年中，开发人员社区中的EJB批评家的声音却越来越大（特别有关实体bean和持久化），一些公司认识到应该改进EJB标准。Sun，作为J2EE的倡导者，知道修改势在必行，并启动了一个新的Java规范要求（JSR），目标是在2003年初简化EJB。这个新的JSR，EJB 3.0（JSR 220），引起了很大的关注。来自Hibernate团队的开发人员加入了早期的专家组，帮助制订出新规范的雏形。其他的供应商，包括Java行业中所有主要公司和许多小公司，也贡献了他们的努力。新标准的一个重要决定是，指定和标准化实际应用程序中有用的东西，借鉴现有成功的产品和项目的思想和理念。因此，Hibernate作为一个成功的数据持久化解决方案，在新标准的持久化部分扮演了重要的角色。但是Hibernate和EJB3之间的关系具体是什么，以及什么是Java Persistence呢？

1．理解标准

首先，难以（如果不是不可能）把规范和产品进行比较。问题应该是“Hibernate实现EJB 3.0规范吗？它对我的项目有什么影响？我必须使用其中一个吗？”

新的EJB 3.0规范有几个部分：第一部分给会话bean、消息驱动bean以及部署规则等，定义新

的EJB编程模型。规范的第二部分专门处理持久化：实体、ORM元数据、持久化管理器接口和查询语言。第二部分被称作JPA，可能因为它的接口是在javax.persistence包中。本书将始终使用这个首字母的缩写。

这种分离也存在于EJB 3.0产品中，有些实现支持规范所有部分的一个完全的EJB 3.0容器，其他产品可能只实现Java Persistence部分。新标准中设计了两条重要的原则：

- JPA引擎应该是可插拔的，这意味着如果你不满意，应该能够从中取出一种产品并用另一种代替——即使你要保留相同的EJB 3.0容器或者Java EE 5.0应用程序服务器。
- JPA引擎应该能够在EJB 3.0（或者任何其他）运行时环境之外运行，而简单的标准Java中不需要容器。

这种设计的结果是，开发人员和架构师有了更多的选择，这样带动了竞争，因此提高了产品的整体质量。当然，实际的产品也提供超出规范的特性，作为特定于供应商的扩展（例如性能调优，或者因为供应商关注一个特定的垂直的问题领域）。

Hibernate实现Java Persistence，并且由于JPA引擎必须是可插拔的，新的和值得关注的软件结合成为可能。可以从不同的Hibernate软件模块中选择，并根据项目的技术和业务需求把它们结合起来。

2. Hibernate Core

Hibernate Core也称作Hibernate 3.2.x或者Hibernate。它是持久化的基础服务，带有原生的API和它存储在XML文件中的映射元数据。它有一种查询语言称作HQL（与SQL几乎相同），以及用于Criteria和Example查询的可编程查询接口。对于每个东西都有几百种选项和特性可用，因为Hibernate Core真正是所有其他模块创建的基础和平台。

Hibernate Core可以单独使用，独立于任何框架或者任何包含所有JDK的特定运行时环境。它适用于每一个Java EE/J2EE应用程序服务器、Swing应用程序、简单的servlet容器等。只要你能够给Hibernate配置数据源，它就能够实现。应用程序代码（在持久层中）将使用Hibernate API和查询，并且你的映射元数据编写在原生的Hibernate XML文件中。

原生的Hibernate API、查询和XML映射文件是本书的主要关注点，它们首先在所有代码示例中得以阐述。因为Hibernate功能是所有其他可用选项的一个超集（superset）。

3. Hibernate Annotations

JDK 5.0提供了定义应用程序元数据的一种新方法：类型安全的注解直接嵌入到Java源代码中。许多Hibernate用户已经熟悉这个概念，就像XDoclet软件在编译时支持Javadoc元数据属性和预处理程序一样（对于Hibernate，它产生XML映射文件）。

使用Hibernate Core顶部的Hibernate Annotations包，现在可以使用类型安全的JDK 5.0元数据作为原生的Hibernate XML映射文件的替代或者补充。一旦见过与Hibernate XML映射文件并排的映射注解时，你就会发现它的语法和语义很常见。然而，基础注解不是私有的。

JPA规范定义ORM元数据语法和语义，主要机制为JDK 5.0注解。（是的，Java EE 5.0和EJB 3.0需要JDK 5.0。）Hibernate Annotations一般来说是实现JPA标准的一组基础注解，它们也是更高级的和更奇异的Hibernate映射和调优所需的一组扩展注解。

可以使用Hibernate Core和Hibernate Annotations减少映射元数据的代码行，相比于原生的XML文件，你可能更喜欢注解更易重构的能力。如果完整的可移植性不是你最关注的，则可以只用JPA注解，或者增加一个Hibernate扩展注解。（在实际应用程序中，你应该相信已经选择的产品，而不是始终否认它的存在。）

本书将通篇讨论注解在开发过程中的影响，如何在映射中使用注解，以及原生的Hibernate XML示例。

4. Hibernate EntityManager

JPA规范也定义编程接口、持久化对象的生命周期规则和查询特性。JPA这部分的Hibernate实现可被用作Hibernate EntityManager，这是另一个可以堆在Hibernate Core顶部的可选模块。当需要简单的Hibernate接口或者甚至需要JDBC连接时，可以退回。Hibernate原生的特性在各个方面都是JPA持久化特性的一个超集。（简单的事实就是，Hibernate EntityManger是对提供JPA兼容性的Hibernate Core的一个小包装。）

使用标准化的接口和标准化的查询语言有个好处：可以使用任何EJB 3.0兼容应用程序服务器执行JPA兼容的持久层。或者，可以在简单的Java中任何特定的标准运行时环境之外使用JPA（这就是Hibernate Core可以被用在任何地方的真正含义）。

Hibernate Annotations应该与Hibernate EntityManager结合考虑。如果你针对JPA接口使用JPA查询编写应用程序代码，而没有用JPA注解创建大部分映射，这是不正常的。

5. Java EE 5.0应用程序服务器

本书不涵盖所有EJB 3.0，我们的重点自然是在持久化和规范的JPA部分。（当然，当谈到关于应用程序架构和设计时，我们将介绍许多托管EJB组件的技术。）

Hibernate也是JBoss应用程序服务器（JBoss AS）的一部分，是J2EE 1.4和（不久的）Java EE 5.0的一个实现。Hibernate Core、Hibernate Annotations和Hibernate EntityManager结合起来，形成了这个应用程序服务器的持久化引擎。因此，可以独立使用每件东西，也可以在应用程序服务器内部使用并享有所有EJB 3.0的益处，例如会话bean、消息驱动的bean和其他的Java EE服务。

要完成这部分的学习，还必须理解Java EE 5.0应用程序服务器不再是J2EE 1.4时代的庞然大物。事实上，JBoss EJB 3.0容器也是个可嵌入的版本，它在其他应用程序服务器的内部，甚至Tomcat或者单元测试里面，或者Swing应用程序中运行。第2章将准备一个利用EJB 3.0组件的项目，给简单的整合测试安装JBoss服务器。

如你所见，原生的Hibernate特性实现了规范的重要部分，或者是必要时提供额外功能的自然的供应商扩展。

有个简单的技巧，可以立即知道正在看的代码是JPA还是原生的Hibernate。如果只看见有`javax.persistence.*`导入，就是正处在规范内工作；如果也导入`org.hibernate.*`，就是正在使用原生的Hibernate功能。稍后会介绍一些更多的技巧，帮你清楚地把可移植的代码和特定于供应商的代码分开。

常见问题　Hibernate的前景如何？Hibernate Core将被独立开发，且比EJB 3.0或者Java Persistence规范更快。它将成为新思想进行试验的地方，就如它一向所做的那样。对于所有使用Hibernate Annotations和Hibernate EntityManager的Java Persistence用户来说，任何为Hibernate Core开发的新特性，都可以立即、自动地作为扩展使用。随着时间的过去，如果一个特定的概念已经证明是没用的，Hibernate开发人员将与其他专家组成员一起，为最新的EJB或者Java Persistence规范制订未来的标准。因此，如果你对快速演变的标准感兴趣，我们鼓励你使用原生的Hibernate功能，并发送反馈给相应的专家组。对整体可移植性的渴望和供应商扩展的排斥，是我们看到EJB 1.x和2.x停滞的主要原因。

对ORM和Hibernate进行了如此一番赞赏之后，应该来看一些实际的代码了。到了结束理论，并建立第一个项目的时候了。

1.5　小结

本章已经讨论了对象持久化的概念，以及ORM作为一种实现技术的重要性。

对象持久化意思是单独的对象可以比应用程序过程存在得更久，它们可以被保存到数据存储并可以在稍后的某个时间点被重新创建。当数据存储是一个基于SQL的关系数据库管理系统时，对象/关系的不匹配出现了。例如，对象网络不能被保存到一个数据库表，对于可移植的SQL数据类型的列，它必须是可分解的和持久化的。针对这个问题的一种好的解决方案是ORM，如果想要Java领域模型有更丰富的类型时，ORM特别有用。

领域模型表示用在Java应用程序中的业务实体。在分层的系统架构中，领域模型用来在业务层执行业务逻辑（在Java中，而不是在数据库中）。这个业务层与下方的持久层交流，以加载和存储领域模型的持久化对象。ORM是管理持久化的持久层中的中间件。

ORM不是所有持久化任务的一颗银弹[①]，它的任务是为开发人员减轻95%的对象持久化工作，例如用许多表联结编写复杂的SQL语句，从JDBC结果集中把值复制给对象或者对象网络。一个全特性的ORM中间件解决方案可以提供数据库可移植性、某种优化技术比如高速缓存，以及利用SQL和JDBC在有限的时间内手工编码不易实现的其他可行的功能。

比ORM更好的解决方案有一天可能会实现。我们（和许多其他人）可能不得不对SQL、持久化API标准和应用程序集成给予重新思考。当今的系统发展成为真正的关系数据库系统——带有无缝的面向对象集成，还只是一种单纯的推测。但是我们不能等待，因为并没有迹象表明这些问题中有任何一项将很快得到改善（一个数十亿美元的行业非常不灵活）。ORM是当前可用的最好的解决方案，为每天面对对象/关系不匹配的开发人员节省了时间。有了EJB 3.0，Java行业认可的完全ORM的一个规范最终也是可用的。

① silver bullet，常用来比喻新技术，尤其是人们寄予厚望的新科技。日常生活中，“银弹”也指一劳永逸的办法，类似于“灵丹妙药”。——编者注

第2章 启动项目

本章内容

- 用Hibernate和Java Persistence实现“Hello World”
- 正向和反向工程的工具集
- Hibernate配置和整合

你想开始使用Hibernate和Java Persistence，想通过一个循序渐进的示例进行学习，想看到这两个持久化API，以及如何从原生的Hibernate或者标准化的JPA中受益。这就是本章的内容：一次简单的“Hello World”应用之旅。

然而，在Hibernate参考文档中已经有了一个公开可用的很棒的完整教程，因此为了避免重复，这里将给出有关Hibernate整合和配置的更为详细的指导。如果你想从一个更为简短的教程开始，以便能在一个小时之内完成，建议你考虑阅读Hibernate参考文档。它引导你使用Hibernate：从一个简单独立的Java应用程序，到最本质的映射概念，最后示范了在Tomcat上部署的一个Hibernate Web应用程序。

本章将介绍如何给一个整合了Hibernate的简单Java应用程序建立项目基础，你会看到在这样的环境中配置Hibernate的更多细节。我们也会讨论Hibernate在一个托管环境中（即在提供了Java EE服务的环境中）的配置和整合。

将Ant作为“Hello World”项目的一个构建工具，并创建不仅能够编译和运行项目，而且能够利用Hibernate Tools的构建脚本。根据开发过程，可以使用Hibernate工具集自动导出数据库Schema，或者甚至从一个现有（遗留）的数据库Schema中通过反向工程得出一个完整的应用程序。

就像每一位好的工程师，在开始第一个真正的Hibernate项目之前，你应该准备工具，并决定采用什么样的开发过程。并且，根据所选择的过程，你自然会喜欢使用不同的工具。让我们看看这个准备阶段，以及你有哪些选择，然后开始一个Hibernate项目。

2.1 启动 Hibernate 项目

在有些项目中，应用程序的开发是通过开发人员以面向对象方式分析业务领域来驱动的。在其他项目中，则深受现有关系数据模型的影响，比如一个遗留的数据库或者由专业的数据建模者

设计的一个全新模式。有几种选择，并且要在开始之前回答下列问题：

- 可以从头开始简洁地设计一个新的业务需求吗？还是有遗留数据和（或）遗留应用程序代码？
- 某些必要的部分可以从现有的文件中自动生成吗（例如，从现有的数据库Schema中产生Java源代码）？数据库Schema可以从Java代码和Hibernate映射元数据中产生吗？
- 哪种工具可用来支持这项工作？其他工具能支持整个开发周期吗？

当我们在接下来的几节中建立基础的Hibernate项目时，将讨论这些问题。下面是你建立基础的Hibernate项目的过程：

(1) 选择开发过程。
(2) 建立项目基础。
(3) 编写应用代码和映射。
(4) 配置和启动Hibernate。
(5) 运行应用。

读完接下来的几节，你就能为自己的项目准备正确的方法了，并且也具备了了解本章稍后要介绍的更复杂场景的背景信息。

2.1.1　选择开发过程

我们先从总体上看看可用的工具、用作源输入的制品，以及生成的输出。图2-1展现了Ant的各种导入和导出任务，所有功能对Eclipse的Hibernate Tools插件都可用。阅读本章时请参照这张图[①]。

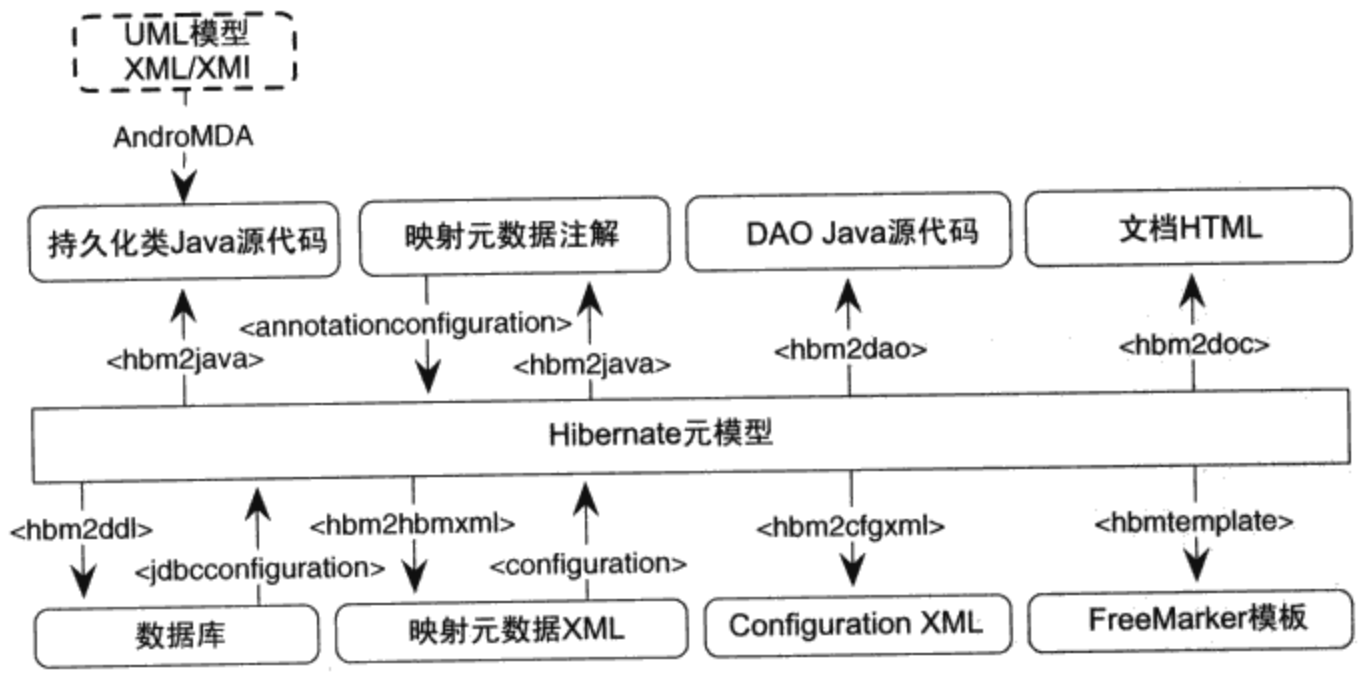


图2-1　用于Hibernate开发的工具的输入和输出

① AndroMDA是一种从UML图文件产生POJO源代码的工具，严格来说不是常用Hibernate工具箱的一部分，因此不在本章讨论。更多有关用于AndroMDA的Hibernate模块信息，请见Hibernate网站上的社区区域。

说明 Eclipse IDE的Hibernate Tools是Eclipse IDE的插件（Eclipse 的JBoss IDE的一部分——一组向导、编辑器和Eclipse中的额外视图，帮助开发EJB3、Hibernate、JBoss Seam和基于JBoss中间件的其他Java应用）。正向和反向工程的特性相当于基于Ant的工具。另外的Hibernate Console视图允许对数据库执行特别的Hibernate查询（HQL和Criteria），并以图表形式浏览结果。Hibernate Tools XML编辑器支持映射文件的自动完成，包括类、属性，甚至表格和列名称。在撰写本书期间，图形工具仍在开发中，并以beta版的形式发行使用。然而，软件的未来发行版将废弃所有的屏幕截图。Hibernate Tools的文档包含许多屏幕截图和详细的项目构建指导，可以轻松地使用Eclipse IDE构建你的第一个“Hello World”程序。

以下为常见的开发场景：

- 自顶向下（top-down）——在自顶向下的开发中，从一个现有的领域模型开始，使用Java完成领域模型的实现，并（理想地）完成数据库Schema方面的自主。你必须构建映射元数据——用XML文件或者通过注解Java源代码，然后选择性地让Hibernate的hbm2ddl工具产生数据库Schema。没有现成的数据库Schema，对于大部分Java开发人员来说，是最舒适的开发风格。你甚至可以使用Hibernate Tools在每个应用程序于开发过程中重启时，自动刷新数据库Schema。
- 自底向上（bottom-up）——反之，自底向上开发始于一个现有的数据库Schema和数据模型。此时，最容易进行的方式就是使用反向工程工具从数据库中抽取元数据。这个元数据可被用来生成XML映射文件，例如用hbm2hbmxml。利用hbm2java，Hibernate映射元数据用来生成Java持久化类，甚至数据访问对象——换句话说，是Java持久层的一个骨架。或者，不编写到XML映射文件，被注解的Java源代码（EJB 3.0实体类）可以直接由工具生成。但是，并非所有的类关联细节和Java专有的元信息都可以用这种策略自动从SQL数据库Schema中生成，因此需要一些手工的工作。
- 起自中间层（middle out）——Hibernate XML映射元数据提供充分的信息来完全推导出数据库Schema，并给应用的持久层生成Java源代码。此外，XML映射文档不会太冗长。因此，有些架构师和开发人员更喜欢起自中间层开发，他们在那里开始手写Hibernate XML映射文件，然后用hbm2ddl生成数据库Schema，用hbm2java生成Java类。Hibernate XML映射文件在开发期间被不断更新，且其他所要创建的东西从这个主定义中生成。另外的业务逻辑或者数据库对象通过子类化和辅助的DDL添加。这种开发风格只建议经验丰富的Hibernate专家使用。
- 在中间会合（meet in the middle）——最难的场景是合并现有的Java类和现有的数据库Schema。此时，Hibernate工具集就无能为力了。当然，不可能把任意的Java领域模型映射给一个给定的Schema，因此这个场景通常至少需要重构一些Java类、数据库Schema或者兼而有之。映射元数据几乎一定要在XML文件中用手工编写（虽然它可能使用注解，如果很匹配的话）。这会是个难以置信的痛苦场景，的确痛苦，幸运的是这种情况非常罕见。

我们现在要更深入地探讨工具和它们的配置选项，并给典型的Hibernate应用程序开发建立一

个工作环境。你可以按照我们循序渐进的指导，构建同样的环境；或者可以只是零碎地选取一些你所需要的，例如Ant构建脚本。

首先假设开发过程是自顶向下的，并且将快速完成一个不涉及任何遗留数据Schema或者Java代码的Hibernate项目。之后，将把代码迁移到JPA和EJB 3.0，然后从一个现有的数据库Schema反向工程，自底向上地开始一个项目。

2.1.2 建立项目

假设你已经从Hibernate网站（http://www.hibernate.org/）下载了Hibernate最新的正式版本，并且已经打开了档案文件。你还需要在开发机器上安装Apache Ant。你还应该从http://hsqldb.org/下载HSQLDB的最新版本，并解压文件包；你将用这个数据库管理系统进行测试。如果你已经安装了另一个数据库管理系统，只要给它获得一个JDBC驱动程序即可。

不是你将在本书后面开发的完善应用，而是从“Hello World”例子开始。这样，你就可以关注开发过程，而不会被Hibernate的细节弄得心烦意乱。首先建立项目目录。

1. 创建工作目录

在系统上创建一个新目录，可以在你喜欢的任何位置，如果用的是Microsoft Windows，C:\helloworld是个好选择。后面的示例将把这个目录当作WORKDIR。创建lib和src子目录，并复制所有需要的库：

```
WORKDIR
    +lib
      antlr.jar
      asm.jar
      asm-attrs.jars
      c3p0.jar
      cglib.jar
      commons-collections.jar
      commons-logging.jar
      dom4j.jar
      hibernate3.jar
      hsqldb.jar
      jta.jar
    +src
```

你在lib目录中见到的这些库来自Hibernate发行包，一个典型的Hibernate项目需要这些库中的大部分。hsqldb.jar文件来自HSQLDB发行包；如果要使用一个不同的数据库管理系统，就用一个不同的驱动程序JAR代替。记住，你在此处见到的这些库，其中有些对于你正在使用的Hibernate的特定版本来说可能不需要，它可能比我们撰写本书时用过的版本更新。为了确定你具备了正确的程序库，要始终检查Hibernate发行包中的lib/README.txt文件。这个文件包含所有必需的和可选的第三方程序库的一个最新列表——你只需要列出运行时所需要的库。

在这个“Hello World”应用程序中，你想要把消息保存在数据库中，并从数据库中加载消息。要给这个业务案例创建领域模型。

2. 创建领域模型

Hibernate应用程序定义了被映射到数据库表的持久化类。你以业务领域的分析为基础定义这些

类；因此，它们是一个领域模型。“Hello World”示例由一个类和它的映射组成。我们来看一下简单的持久化类是什么样子，如何创建映射，以及可以用Hibernate中的持久化类的实例完成哪些事情。

这个例子的目标是在一个数据库中存储消息，并获取消息用来显示。你的应用程序有一个简单的持久化类Message，它表示这些可打印的消息。代码清单2-1显示了Message类。

代码清单2-1 Message.java：一个简单的持久化类

```
package hello;

public class Message {
    private Long id;
    private String text;
    private Message nextMessage;

    Message() {}

    public Message(String text) {
        this.text = text;

    }

    public Long getId() {
        return id;
    }
    private void setId(Long id) {
        this.id = id;
    }

    public String getText() {
        return text;
    }
    public void setText(String text) {
        this.text = text;
    }

     public Message getNextMessage() {
         return nextMessage;
     }
     public void setNextMessage(Message nextMessage) {
         this.nextMessage = nextMessage;
     }
}
```

标识符属性

消息文本

另一个Message对象的引用

Message类有3个属性：标识符属性、消息文本和另一个Message对象的引用。标识符属性允许应用程序访问持久化对象的数据库同一性——主键值。如果Message的两个实例具有相同的标识符值，它们在数据库中就表示相同的行。

这个例子对标识符属性的类型使用Long，但这不是必要条件。事实上Hibernate允许给标识符类型使用任何东西，稍后你会见到。

你可能已经注意到Message类的所有属性都有JavaBeans风格的属性访问方法。类也有一个无参构造函数。本例中介绍的持久化类看起来通常有点像这样。无参构造函数是个必要条件（像Hibernate这样的工具在这个构造函数上使用反射来实例化对象）。

Message类的实例可以由Hibernate管理（变成持久化），但是并非一定。因为Message对象不实现任何特定于Hibernate的类或者接口，可以像使用任何其他的Java类一样使用它：

```
Message message = new Message("Hello World");
    System.out.println( message.getText() );
```

这个代码片段正确地完成了你对“Hello World”应用程序的期待：把Hello World打印到控制台。这看起来可能像是我们在这里自作聪明；事实上，我们正在示范一项重要的特性，它把Hibernate和一些其他的持久化方案区别开来。无论如何，持久化类都可以被用在任何执行环境中——不需要专门的容器。注意这也是新的JPA实体的好处之一，这些实体也是简单的Java对象。

把Message类的代码保存到你的源代码文件夹，该文件夹位于一个以hello命名的目录和包（package）中。

3. 映射类到数据库Schema

为了让对象/关系映射发挥神奇的魔力，Hibernate需要更多关于Message类究竟应该如何被持久化的信息。换句话说，Hibernate需要知道该类的实例要如何存储和加载。这个元数据可被写进一个XML映射文档，除了其他东西之外，这个文档还定义了Message类的属性如何映射到MESSAGES表的列。来看看代码清单2-2中的映射文档。

代码清单2-2　简单的Hibernate XML映射

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC
    "-//Hibernate/Hibernate Mapping DTD//EN"
    "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">

<hibernate-mapping>
    <class
        name="hello.Message"
        table="MESSAGES">

        <id
            name="id"
            column="MESSAGE_ID">
            <generator class="increment"/>
        </id>

        <property
            name="text"
            column="MESSAGE_TEXT"/>

        <many-to-one
            name="nextMessage"
            cascade="all"
            column="NEXT_MESSAGE_ID"
            foreign-key="FK_NEXT_MESSAGE"/>

    </class>

</hibernate-mapping>
```

这个映射文档告诉Hibernate，Message类将被持久化到MESSAGES表，标识符属性映射到具名MESSAGE_ID的列，text属性映射到具名MESSAGE_TEXT的列，并且具名nextMessage的属性是一个关联，带有映射到具名NEXT_MESSAGE_ID的外键列的多对一多样性。Hibernate也为你生成数据库Schema，并添加名为FK_NEXT_MESSAGE的外键约束到数据库目录。（现在不用担心其他细节。）

这个XML文档不难理解，你可以很容易地手工编写和维护它。稍后，我们讨论一种直接在源代码中使用注解来定义映射信息的方法；但是无论你选择哪种方法，Hibernate都有足够的信息生成所有需要插入、更新、删除和获取Message类实例的SQL语句。你不再需要用手工编写这些SQL语句。

创建名为Message.hbm.xml的文件，包含代码清单2-2中所示的内容，并把它与源代码hello包中的Message.java文件放在一起。hbm的后缀是被Hibernate社区接受的一个命名约定，且大部分开发人员更喜欢把映射文件与它们领域类的源代码放在一起。

让我们在“Hello World”应用程序的主代码中加载和存储一些对象。

4. 存储和加载对象

你阅读本书的真正目的是要学习Hibernate，因此让我们把一个新的Message保存到数据库（请见代码清单2-3）。

代码清单2-3 “Hello World”的主应用程序代码

```
package hello;

import java.util.*;
import org.hibernate.*;
import persistence.*;

public class HelloWorld {

    public static void main(String[] args) {

        // First unit of work
        Session session =
            HibernateUtil.getSessionFactory().openSession();
        Transaction tx = session.beginTransaction();

        Message message = new Message("Hello World");
        Long msgId = (Long) session.save(message);

        tx.commit();
        session.close();

        // Second unit of work
        Session newSession =
            HibernateUtil.getSessionFactory().openSession();
        Transaction newTransaction = newSession.beginTransaction();

        List messages =
            newSession.createQuery("from Message m order by
            m.text asc").list();

        System.out.println( messages.size() +
            " message(s) found:" );
```

```
            for ( Iterator iter = messages.iterator();
                  iter.hasNext(); ) {
                Message loadedMsg = (Message) iter.next();
                System.out.println( loadedMsg.getText() );
            }

            newTransaction.commit();
            newSession.close();

            // Shutting down the application
            HibernateUtil.shutdown();
        }

    }
```

把这段代码放在你项目的源代码文件夹中的HelloWorld.java文件中，在hello包中。我们来浏览一下这段代码。

类有一个标准的Java main()方法，可以直接从命令行调用它。在主应用程序代码内部，用Hibernate执行两个独立的工作单元。第一个单元存储一个新的Message对象，第二个单元加载所有对象并把它们的文本打印到控制台。

调用Hibernate Session、Transaction和Query接口访问数据库：

- Session（会话）——Hibernate Session集多种功能于一身。它是个单线程、非共享的对象，表示使用数据库的一个特定工作单元。它有持久化管理器API，调用它来加载和存储对象。（Session的内部由一列SQL语句组成，这些语句要与数据库在某个时点上进行同步，且托管持久化实例的一个映象由Session监控。）
- Transaction（事务）——这个Hibernate API可以用来编程式地设置事务范围，但它是可选的（事务范围不是可选的）。其他的选择还有JDBC事务划分、JTA接口，或者带有EJB的容器托管事务。
- Query（查询）——数据库查询可以写进Hibernate自己的面向对象的查询语言（HQL）或者简单的SQL中。这个接口允许你创建查询、在查询中绑定参数给占位符，并以各种方式执行查询。

先忽略调用HibernateUtil.getSessionFactory()的代码行——我们很快就会谈到它。

第一个工作单元，如果运行起来，会导致一些类似下列SQL的执行：

```
insert into MESSAGES (MESSAGE_ID, MESSAGE_TEXT, NEXT_MESSAGE_ID)
    values (1, 'Hello World', null)
```

等等，MESSAGE_ID列正被初始化为一个奇怪的值。你没有在任何地方设置Message的id属性，因为希望它的值为NULL，对吗？事实上，id属性很特殊。它是一个标识符属性：它保存一个生成的唯一值。当调用save()时，这个值被Hibernate分配到Message实例。（我们稍后将讨论这个值如何生成。）

看看第二个工作单元。这个文字型字符串"from Message m order by m.text asc"是个Hibernate查询，用HQL表达。调用list()时，这个查询在内部被翻译成下列SQL：

```
select m.MESSAGE_ID, m.MESSAGE_TEXT, m.NEXT_MESSAGE_ID
    from MESSAGES m
```

```
    order by m.MESSAGE_TEXT asc
```

如果运行这个main()方法（现在不要尝试——你仍需要配置Hibernate），控制台输出如下：

```
1 message(s) found:
    Hello World
```

如果以前从未使用过像Hibernate这样的ORM工具，你可能希望在代码或者映射元数据中的某个地方看到SQL语句，但是没有。所有SQL都在运行时生成（事实上，所有可重用的SQL语句都是在启动时生成）。

下一步一般是配置Hibernate。然而，如果你有信心，可以另外添加两个Hibernate特性——自动脏检查和级联——在第三个工作单元，通过给主应用程序添加下列代码：

```
// Third unit of work
Session thirdSession =
    HibernateUtil.getSessionFactory().openSession();
Transaction thirdTransaction = thirdSession.beginTransaction();

// msgId holds the identifier value of the first message
message = (Message) thirdSession.get( Message.class, msgId );

message.setText( "Greetings Earthling" );
message.setNextMessage(
    new Message( "Take me to your leader (please)" )
);

thirdTransaction.commit();
thirdSession.close();
```

这段代码在同一个数据库事务内部调用3个SQL语句：

```
select m.MESSAGE_ID, m.MESSAGE_TEXT, m.NEXT_MESSAGE_ID
from MESSAGES m
where m.MESSAGE_ID = 1

insert into MESSAGES (MESSAGE_ID, MESSAGE_TEXT, NEXT_MESSAGE_ID)
values (2, 'Take me to your leader (please)', null)

update MESSAGES
set MESSAGE_TEXT = 'Greetings Earthling', NEXT_MESSAGE_ID = 2
where MESSAGE_ID = 1
```

注意Hibernate如何侦测到text和第一条消息的nextMessage属性的修改，并自动更新数据库——Hibernate的确进行了自动脏检查。当你在一个工作单元内部修改对象状态时，这个特性为你免除了显式要求Hibernate去更新数据库的动作。类似地，当从第一条消息创建引用时，新消息被变成了持久化。这个特性称作级联保存（cascading save）。它为你免除了通过调用save()显式地使新对象变成持久化的动作，只要它可以通过一个已经持久化的实例获得。

还注意到SQL语句的排列与你设置属性值的顺序不同。Hibernate使用一个完善的运算法则来确定一种有效的顺序，避免违背数据库外键约束，但对于用户仍然足以预测。这个特性称作事务迟写（transactional write-behind）。

如果现在运行应用程序，会得到下列输出（你必须把第二个工作单元复制到第三个之后，再次执行查询-显示步骤）：

```
2 message(s) found:
Greetings Earthling
Take me to your leader (please)
```

现在你有了领域类、一个XML映射文件和一个加载和存储对象的“Hello World”应用程序代码。在可以编译和运行这段代码之前，需要创建Hibernate的配置（并揭开HibernateUtil类的神秘面纱）。

2.1.3 Hibernate 配置和启动

初始化Hibernate的常规方法是从一个Configuration对象中创建一个SessionFactory对象。如果你喜欢的话，可以把Configuration当作Hibernate配置文件（或者属性文件）的一个对象表示法。

在将其包装在HibernateUtil类中之前，先来看看这些变量。

1. 构建SessionFactory

这是一个典型的Hibernate启动过程的例子，在一行代码中，使用自动的配置文件侦测：

```
SessionFactory sessionFactory =
    new Configuration().configure().buildSessionFactory();
```

等等，Hibernate怎么知道配置文件放置的位置，以及要加载哪个配置文件呢？

调用new Configuration()时，Hibernate在classpath的根目录下搜索名为hibernate.properties的文件。如果找到了，所有hibernate.*属性都会被加载并添加到Configuration对象。

调用configure()时，Hibernate在classpath的根目录下搜索名为hibernate.cfg.xml的文件，如果无法找到就会抛出一个异常。当然，如果你没有这个配置文件，就不一定要调用这个方法。如果XML配置文件中的设置与较早设置的属性完全相同，XML设置就覆盖前面的设置。

hibernate.properties配置文件的位置始终在classpath的根目录中，处在任何包之外。如果要使用一个不同的文件，或者要Hibernate在classpath的子目录中查找XML配置文件，就必须把路径当作configure()方法的一个实参进行传递：

```
SessionFactory sessionFactory = new Configuration()
            .configure("/persistence/auction.cfg.xml")
            .buildSessionFactory();
```

最后，在创建SessionFactory之前，你始终可以在Configuration对象中编程式地设置额外的配置选项或者映射文件位置：

```
SessionFactory sessionFactory = new Configuration()
          .configure("/persistence/auction.cfg.xml")
          .setProperty(Environment.DEFAULT_SCHEMA, "CAVEATEMPTOR")
          .addResource("auction/CreditCard.hbm.xml")
          .buildSessionFactory();
```

许多配置源在这里得到应用：首先读取classpath中的Hibernate.properties文件（如果有的话）。接下来，添加来自/persistence/auction.cfg.xml的所有设置，并覆盖以前应用的任何设置。最后，通过编程设置另一个配置属性（默认的数据库Schema名称），并且添加另一个Hibernate XML映射

元数据文件到配置中。

当然，可以通过编程设置所有选项，或者在不同的XML配置文件之间切换，得到不同的部署数据库。对于如何配置和部署Hibernate，实际上并没有限制；最后，只需要从一个准备好的配置中创建SessionFactory。

说明 方法链（method chaining）——方法链是一种编程风格，为许多Hibernate接口所支持。这种风格在Smalltalk中比在Java中更普及，有些人认为，比起更受认可的Java风格，它更不易阅读且更难以调试。然而，在许多情况下它很方便，例如用于本节中已经见过的配置片段。它是这样工作的：大部分Java开发人员声明设置方法（setter method）或者添加方法（adder method）为void类型，意味着它们不返回值；但是在Smalltalk中没有void类型，设置方法或者添加方法通常返回接收对象。我们在一些代码实例中使用这种Smalltalk风格，但是如果你不喜欢，就不必用它。如果你真的使用了这种编码风格，最好在一个不同的行中编写每一种方法调用。否则，你的调试器在这些代码中可能举步维艰。

既然了解Hibernate如何启动，以及如何创建SessionFactory，接下来要做什么？必须给Hibernate创建一个配置文件。

2. 创建XML配置文件

假设你要让事情保持简单，并且像大部分用户一样，你决定给包含所有配置细节的Hibernate使用单个XML配置文件。

建议你给新的配置文件使用默认名hibernate.cfg.xml，并把它直接放在项目的源代码目录中，处在任何包之外。这样，编译之后它就会在classpath的根目录中结束，并且Hibernate会自动找到它。看看代码清单2-4中的文件。

代码清单2-4 简单的Hibernate XML配置文件

```
<!DOCTYPE hibernate-configuration SYSTEM
"http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">

<hibernate-configuration>
  <session-factory>
    <property name="hibernate.connection.driver_class">
        org.hsqldb.jdbcDriver
    </property>
    <property name="hibernate.connection.url">
        jdbc:hsqldb:hsql://localhost
    </property>
    <property name="hibernate.connection.username">
        sa
    </property>
      <property name="hibernate.dialect">
          org.hibernate.dialect.HSQLDialect
      </property>

      <!-- Use the C3P0 connection pool provider -->
```

```
        <property name="hibernate.c3p0.min_size">5</property>
        <property name="hibernate.c3p0.max_size">20</property>
        <property name="hibernate.c3p0.timeout">300</property>
        <property name="hibernate.c3p0.max_statements">50</property>
        <property name="hibernate.c3p0.idle_test_period">3000</property>

        <!-- Show and print nice SQL on stdout -->
        <property name="show_sql">true</property>
        <property name="format_sql">true</property>

        <!-- List of XML mapping files -->
        <mapping resource="hello/Message.hbm.xml"/>

    </session-factory>
</hibernate-configuration>
```

文档类型声明（Document Type Declaration，DTD）被XML解析器用来针对Hibernate 配置DTD验证这个文件。注意这与用于Hibernate XML映射文件的那个DTD不一样。还要注意我们在属性值里面增加了一些换行，使它更易于阅读——在真正的配置文件中你不应该这么做（除非你的数据库用户名包含了换行）。

在配置文件中首当其冲的是数据库连接设置。你要告诉Hibernate用了哪种数据库JDBC驱动，以及如何通过URL、用户名和密码连接到数据库（此处省略了密码，因为默认时HSQLDB不需要）。你设置了一个Dialect，因此Hibernate知道它必须生成哪个SQL变种来与数据库进行对话；Hibernate配有许多方言——查看一下Hibernate API文档中的清单。

在XML配置文件中，可以指定Hibernate属性而不用hibernate作为前缀，因此你可以写成hibernate.show_sql或者只是show_sql。除此之外，属性名称和值与编程式的配置属性是一致的——也就是说，与org.hibernate.cfg.Environment中定义的常量一致。例如，hibernate.connection.driver_class属性具有常量Enviroment.DRIVER。

在探讨一些重要的配置选项之前，考虑配置中命名Hibernate XML映射文件的最后一行。在创建SessionFactory之前，Configuration对象需要知道你所有的XML映射文件。SessionFactory是给一组特定的映射元数据表示特定Hibernate配置的一个对象。可以在Hibernate XML映射文件中列出所有的XML映射文件，也可以在Configuration对象中编程式地设置它们的名称和路径。不管怎样，如果把它们当作一个资源（resource）列出，映射文件的路径在classpath中就是个相对位置，在这个实例中，hello是classpath根目录中的一个包。

你还启用了把所有Hibernate执行的SQL打印到控制台，并告诉Hibernate给它设置好格式，以便可以检查幕后到底发生了些什么。本章稍后会回到日志（logging）的话题。

另一个偶尔会派上用场的技巧，是使系统属性的配置选项更动态些：

```
...
<property name="show_sql">${displaysql}</property>
...
```

现在你可以在启动应用程序时，在命令行中指定一个系统属性，例如用java-displaysql=true，并且这将被自动应用到Hibernate配置属性上。

数据库连接池设置值得特别关注。

3. 数据库连接池

一般来说，不需要在每次要与数据库交互时都创建连接。反之，Java应用程序应该使用连接池（pool）。需要在数据库上工作的每个应用程序线程都从这个池中请求连接，然后当执行完所有SQL操作后把它返回到池中。这个池维护着连接，并使打开和关闭连接的成本减到最少。

使用连接池有3个原因：

- 获得新的连接很昂贵。有些数据库管理系统甚至给每个连接启动一个全新的服务器进程。
- 为数据库管理系统维护许多闲置的连接很昂贵，并且连接池可以最优化闲置连接的使用（或者没有请求时就断开连接）。
- 给某些驱动程序创建预编译的语句也很昂贵，且连接池可以对跨请求的连接高速缓存语句。

图2-2显示了连接池在非托管的应用程序运行时环境中的角色（即不要任何应用程序服务器）。

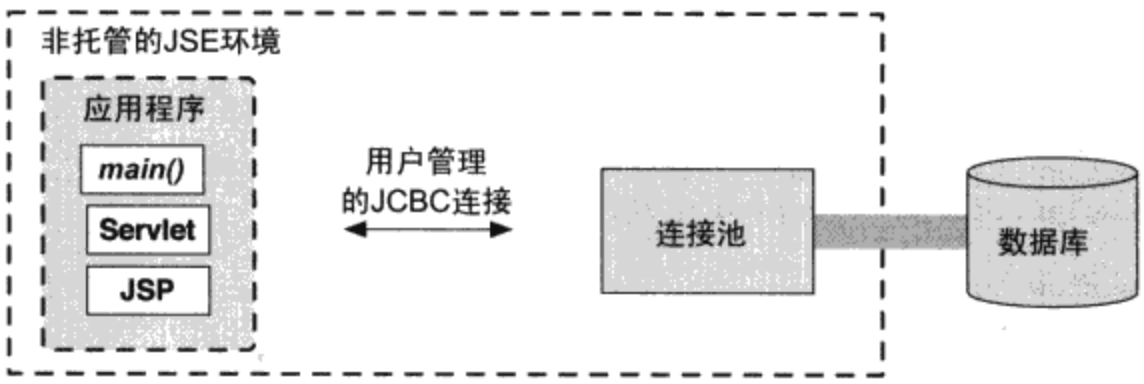


图2-2 非托管环境中的JDBC连接池

没有应用程序服务器提供连接池，应用程序要么实现它自己的池化算法，要么依赖于第三方的库（如开源C3P0连接池软件）。没有Hibernate，应用程序代码就调用连接池来获取JDBC连接，然后用JDBC编程接口执行SQL语句。当应用程序关闭SQL语句，且最后关闭连接时，预编译的语句和连接没有被释放，而是返回到池中。

使用Hibernate，这个图就变了：它充当JDBC连接池的一个客户端程序，如图2-3所示。应用程序代码给持久化操作使用Hibernate的`Session`和`Query` API，它（可能）用Hibernate的`Transaction` API管理数据库事务。

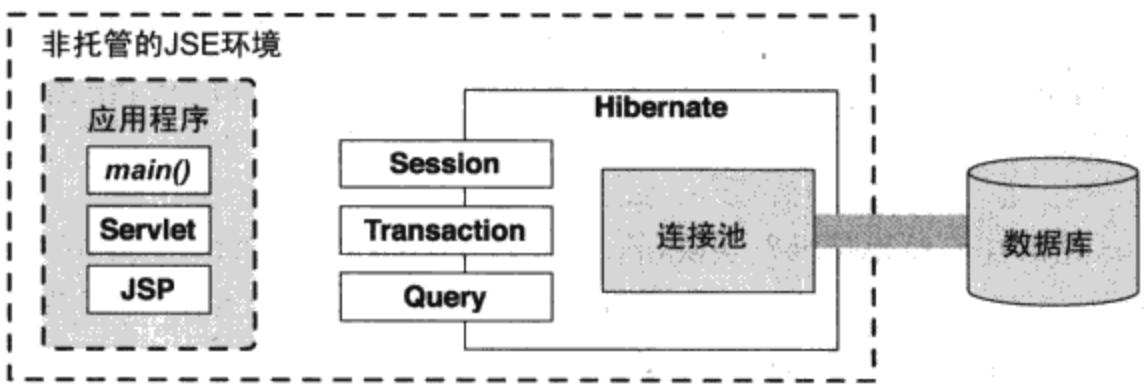


图2-3 非托管环境中带有连接池的Hibernate

Hibernate定义了一个插件架构，允许与任何连接池软件整合。然而，对C3P0的支持是内建的，并且这个软件与Hibernate捆绑在一起，因此你将会用到它（你已经复制了c3p0.jar文件到库目录下，对吗？）。Hibernate替你维护连接池，并且传递配置属性。如何通过Hibernate配置C3P0呢？

配置连接池的一种方法是把设置放进hibernate.cfg.xml配置文件，就像前一节中所做的那样。

另一种方法是，可以在应用的classpath根目录中创建一个hibernate.properties文件。C3P0的hibernate.properties文件的一个实例如代码清单2-5所示。注意这个文件，除了映射资源清单之外，相当于代码清单2-4中所示的配置。

代码清单2-5　给C3P0连接池设置使用hibernate.properties

```
hibernate.connection.driver_class = org.hsqldb.jdbcDriver
hibernate.connection.url = jdbc:hsqldb:hsql://localhost
hibernate.connection.username = sa
hibernate.dialect = org.hibernate.dialect.HSQLDialect

hibernate.c3p0.min_size = 5                    ❶
hibernate.c3p0.max_size = 20                   ❷
hibernate.c3p0.timeout = 300                   ❸
hibernate.c3p0.max_statements = 50             ❹
hibernate.c3p0.idle_test_period = 3000         ❺

hibernate.show_sql = true
hibernate.format_sql = true
```

❶ 这是C3P0随时做好准备的最少的JDBC连接数量。

❷ 这是池中最多的连接数量。如果超出这个数量，就会在运行时抛出异常。

❸ 指定超时周期（在本例中为300秒），在它之后闲置连接将从池中移除。

❹ 最多高速缓存50个预编译的语句。高速缓存预编译的语句是利用Hibernate获得最好性能的要素。

❺ 这是在连接被自动验证之前，一个以秒为单位的闲置时间。

指定表单`hibernate.c3p0.*`的属性选择C3P0作为连接池（`c3p0.max_size`选项是必需的——你无需任何其他的切换来启用C3P0支持）。C3P0有着比前一个例子所展示的更多特性；参考Hibernate发行包的etc/子目录中的属性文件，找到一个可以从中复制的综合示例。

类`org.hibernate.cfg.Environment`的Javadoc也为每一个Hibernate配置属性提供文档。甚至，可以在Hibernate参考文档中找到一个带有全部Hibernate配置选项的最新表格。然而，本书通篇都将阐述最重要的设置。现在你已经知道了在开始之前需要的一切。

常见问题　可以提供自己的连接吗？实现`org.hibernate.connection.ConnectionProvider`接口，用`hibernate.connection.provider_class`配置选项命名你的实现。现在，如果Hibernate需要数据库连接，它将依赖于你定制的提供程序（provider）。

既然已经完成了Hibernate配置文件，你就可以继续前进，在应用程序中创建`SessionFactory`了。

4. 处理SessionFactory

在大部分Hibernate应用程序中，SessionFactory应该在应用程序初始化期间被实例化一次。然后单独的实例应该为特定程序中的所有代码所用，任何Session都应该用这个单独的SessionFactory来创建。SessionFactory是线程安全的，且能够被共享；Session是个单线程的对象。

一个经常被问及的问题是，这个工厂创建后应该被存储在什么地方，以及它如何被轻松地访问。有更高级而舒适的选项如JNDI和JMX，但是它们通常只能在完全的Java EE应用程序服务器中才能使用。反之，我们将引入一个实用、快速的解决方案，解决Hibernate启动的问题（那一行代码）和SessionFactory的存储与访问：你将使用一个静态的全局变量和静态的初始化。

变量和初始化这两者都可以在单个类中实现，这个类将称作HibernateUtil。这个辅助类在Hibernate社区中大名鼎鼎——在没有Java EE 服务的普通Java应用程序中，它是Hibernate启动的一种常用模式。代码清单2-6中显示了一个基本的实现。

代码清单2-6 HibernateUtil类用于启动而SessionFactory用于处理

```
package persistence;

import org.hibernate.*;
import org.hibernate.cfg.*;

public class HibernateUtil {
    private static SessionFactory sessionFactory;

    static {
      try {
         sessionFactory=new Configuration()
                           .configure()
                           .buildSessionFactory();
      } catch (Throwable ex) {
         throw new ExceptionInInitializerError(ex);
      }
    }

    public static SessionFactory getSessionFactory() {
        // Alternatively, you could look up in JNDI here
        return sessionFactory;
    }

    public static void shutdown() {
        // Close caches and connection pools
        getSessionFactory().close();
    }

}
```

创建一个静态初始化器代码块来启动Hibernate；这个块正好被这个类的加载器（loader）在初始化中执行一次，在该类被加载的时候。应用程序中的HibernateUtil的第一次调用加载类，创建SessionFactory，同时设置静态变量。如果出现问题，任何Exception或者Error就都被

包起来并抛到静态的块之外（这就是你为什么要捕捉Throwable的原因）。ExceptionInInitializerError中的包装对于静态的初始化器来说是强制的。

你已经在一个名为persistence的新包中创建了这个新的类。在一个全特性的Hibernate应用程序中，经常需要这样一个包——例如，把你定制的持久层拦截器和数据类型转换器包成基础结构的一部分。

现在，每当你需要在应用程序中访问一个Hibernate Session时，就可以轻松地用HibernateUtil.getSessionFactory().openSession()获得，就像稍早你在HelloWorld主应用代码中所做的那样。

你差不多准备运行和测试应用程序了。但是因为你必定想要知道幕后发生了什么，因此首先启用日志。

5. 启用日志和统计

你已经见过了hibernate.show_sql配置属性。当你用Hibernate开发软件时，将会频繁地需要用到它；它把所有生成的SQL的日志启用到控制台。你将用它解决问题、性能调优，及查看正在进行的工作。如果还启用了hibernate.format_sql，输出就会更易阅读，但会占用更多的屏幕空间。目前为止你还没有设置的第三个选项是hibernate.use_sql_comments——它导致Hibernate把注释放在所有生成的SQL语句内部，来提示它们的出处。例如，你可以轻松地看出某个特定的SQL语句是从显式的查询还是所需的集合初始化中生成的。

启用SQL输出为stdout只是你的第一个日志选项。Hibernate（和许多其他的ORM实现）异步地执行SQL语句。当应用程序调用session.save()时，通常不执行INSERT语句；或者当应用程序调用item.setPrice()时，也不会立即发布UPDATE。反之，SQL语句通常在事务结束时发布。

这意味着追踪和调试ORM代码有时候并不容易。理论上，应用程序可能把Hibernate当作黑盒处理，并忽视这个行为。然而，当正在解决一道难题时，需要能够准确地看到Hibernate内部正在做什么。由于Hibernate是开源的，你可以轻松地进入Hibernate代码，并且这偶尔还能帮上很大的忙！经验丰富的Hibernate专家仅仅通过查看Hibernate日志和映射文件来调试问题；我们鼓励你在由Hibernate生成的日志输出上花些时间，并熟悉一下它的内部情况。

Hibernate通过Apache通用日志工具（commons-logging）这个指示输出到Apache Log4j（如果把log4j.jar放在classpath中）或者JDK 1.4 logging（如果你正在JDK 1.4或者其更高的版本上运行，且没有Log4j的话）很薄的抽象层，对所有值得关注的事件都作了日志记录。推荐Log4j，因为它更成熟、更普及，并且在更积极的开发之中。

为了从Log4j中看到输出，你需要classpath中名为log4j.properties的文件（就在hibernate.properties或者hibernate.cfg.xml旁边）。而且，别忘记复制log4j.jar库到lib目录下。代码清单2-7中的Log4j配置示例指示所有的日志消息到控制台。

代码清单2-7　log4j.properties配置文件的一个示例

```
# Direct log messages to stdout
log4j.appender.stdout=org.apache.log4j.ConsoleAppender
log4j.appender.stdout.Target=System.out
```

```
log4j.appender.stdout.layout=org.apache.log4j.PatternLayout
log4j.appender.stdout.layout.ConversionPattern=%d{ABSOLUTE}
    ➥%5p %c{1}:%L - %m%n

# Root logger option
log4j.rootLogger=INFO, stdout

# Hibernate logging options (INFO only shows startup messages)
log4j.logger.org.hibernate=INFO
# Log JDBC bind parameter runtime arguments
log4j.logger.org.hibernate.type=INFO
```

这个配置文件中的最后一个类特别值得关注：如果把它设置成DEBUG等级，就会启用JDBC绑定参数的日志，提供通常你在特别的SQL控制台日志中看不到的信息。更全面的示例，请查阅捆绑在Hibernate发行包的etc/目录中的log4j.properties文件，也看一下Log4j文档了解更多信息。注意，在产品运行时永远不要给DEBUG级别下的任何东西记录日志，因为这么做会严重影响应用程序的性能。

也可以通过启用实时统计来监控Hibernate。没有应用程序服务器(也就是说，如果你没有JMX部署环境)，运行时从Hibernate引擎之外获取统计的最容易的方法就是SessionFactory：

```
Statistics stats =
    HibernateUtil.getSessionFactory().getStatistics();

stats.setStatisticsEnabled(true);
...
stats.getSessionOpenCount();
stats.logSummary();

EntityStatistics itemStats =
    stats.getEntityStatistics("auction.model.Item");
itemStats.getFetchCount();
```

统计接口中，Statistics用于全局的信息，EntityStatistics用于有关一个特定实体的信息，CollectionStatistics用于一个特定的集合任务，QueryStatistics用于SQL和HQL查询，SecondLevelCacheStatistics用于详细的关于可选的二级数据高速缓存中的一个特定区域的运行时信息。一种方便的方法是logSummary()，它通过单个调用把一个完整的摘要打印到控制台。如果想要通过配置来启用统计集合，并且不用编程，就设置hibernate.generate_statistics配置属性为true。有关各种统计获取方法的更多信息请见API文档。

在运行“Hello World”应用程序之前，检查一个工作目录是否具备所有必需的文件：

```
WORKDIR
build.xml
+lib
  <all required libraries>
+src
  +hello
    HelloWorld.java
    Message.java
    Message.hbm.xml
```

```
+persistence
  HibernateUtil.java
hibernate.cfg.xml (or hibernate.properties)
log4j.properties
```

第一个文件build.xml是一个Ant构建定义。它包含构建和运行应用程序的Ant目标，这个我们接下来会讨论。你还将添加一个可以自动生成数据库Schema的任务。

2.1.4　运行和测试应用程序

要运行应用程序，首先要对它进行编译，并通过正确的数据库Schema启动数据库管理系统。

Ant是Java的一个强大的构建系统。一般要给项目编写build.xml文件，并用Ant命令行工具调用你在这个文件中定义的构建任务。如果支持的话，也可以从Java IDE中调用Ant任务。

1．用Ant编译项目

现在要给“Hello World”项目添加build.xml文件和一些任务。构建文件的初始内容如代码清单2-8所示——直接在WORKDIR中创建这个文件。

代码清单2-8　“Hello World”的一个基础的Ant构建文件

```
<project name="HelloWorld" default="compile" basedir=".">

    <!-- Name of project and version -->
    <property name="proj.name"      value="HelloWorld"/>
    <property name="proj.version"   value="1.0"/>

    <!-- Global properties for this build -->
    <property name="src.java.dir"   value="src"/>
    <property name="lib.dir"        value="lib"/>
    <property name="build.dir"      value="bin"/>

    <!-- Classpath declaration -->
    <path id="project.classpath">
        <fileset dir="${lib.dir}">
            <include name="**/*.jar"/>
            <include name="**/*.zip"/>
        </fileset>
    </path>

    <!-- Useful shortcuts -->
    <patternset id="meta.files">
        <include name="**/*.xml"/>
        <include name="**/*.properties"/>
    </patternset>

    <!-- Clean up -->
    <target name="clean">
        <delete dir="${build.dir}"/>
        <mkdir dir="${build.dir}"/>
    </target>

    <!-- Compile Java source -->
    <target name="compile" depends="clean">
        <mkdir dir="${build.dir}"/>
```

```
        <javac
            srcdir="${src.java.dir}"
            destdir="${build.dir}"
            nowarn="on">
            <classpath refid="project.classpath"/>
        </javac>
    </target>

    <!-- Copy metadata to build classpath -->
    <target name="copymetafiles">
        <copy todir="${build.dir}">
            <fileset dir="${src.java.dir}">
                <patternset refid="meta.files"/>
            </fileset>
        </copy>
    </target>

    <!-- Run HelloWorld -->
    <target name="run" depends="compile, copymetafiles"
        description="Build and run HelloWorld">
        <java fork="true"
              classname="hello.HelloWorld"
              classpathref="project.classpath">
              <classpath path="${build.dir}"/>
        </java>
    </target>

</project>
```

这个Ant构建文件的前半部分包含属性设置，例如项目名称以及文件和目录的全局位置。你可能已经发现这个构建是以现有的目录结构（即你的WORKDIR）为基础的（对于Ant，这与basedir是相同目录）。当用没有具名的任务调用这个构建文件时，default的目标就是compile。

接下来，定义一个以后能够很容易引用的名称project.classpath，作为项目的库目录中所有库的一个快捷操作。另一个很方便的的快捷样式定义为meta.files。你在构建过程中，需要使用这个过滤器分开处理配置和元数据文件。

clean任务移除了所有创建的和编译的文件，并清理了项目。最后3个任务compile、copymetafiles和run，应该不言自明的。根据所有Java源文件的编译运行应用程序，并复制所有映射和属性配置文件到构建目录下。

现在，在WORKDIR中执行ant compile，来编译“Hello World”应用。编译期间你应该看不到错误（也没有任何警告），而在bin目录中可以找到编译好的类文件。也调用ant copymetafiles一次，检查是否所有的配置和映射文件都正确地复制到了bin目录。

在运行应用之前，启动数据库管理系统，并导出一个新的数据库Schema。

2．启动HSQL数据库系统

Hibernate支持25个以上预设好的SQL数据库管理系统，并支持任何可被轻松添加的未知方言。如果你有一个现成的数据库，或者如果你了解基础的数据库管理，也可以用喜欢的系统设置取代以前创建的配置选项（主要是连接和方言设置）。

要向世界说hello，你需要一个易于安装和配置的轻量级、实用的数据库系统。HSQLDB是个

不错的选择，它是用Java编写的一个开源的SQL数据库管理系统。它可以与主应用程序同步运行，但是依我们的经验，用一个TCP端口监听给连接独立运行HSQLDB通常更方便。你已经把hsqldb.jar文件复制到WORKDIR的库目录中——这个库包括了连接到运行示例所需的数据库引擎和JDBC驱动程序。

要启动HSQLDB服务器，得打开一个命令行，切换到你的WORKDIR，并运行图2-4所示的命令。你应该看到启动消息，最后是一个教你如何关闭数据库系统的帮助消息（用Ctrl+C也可以）。你还将在WORKDIR中看到一些新文件，从test开始——这些是HSQLDB用来给你存储数据的文件。如果想要从一个新的数据库开始，就在服务器重启之间删除这些文件。

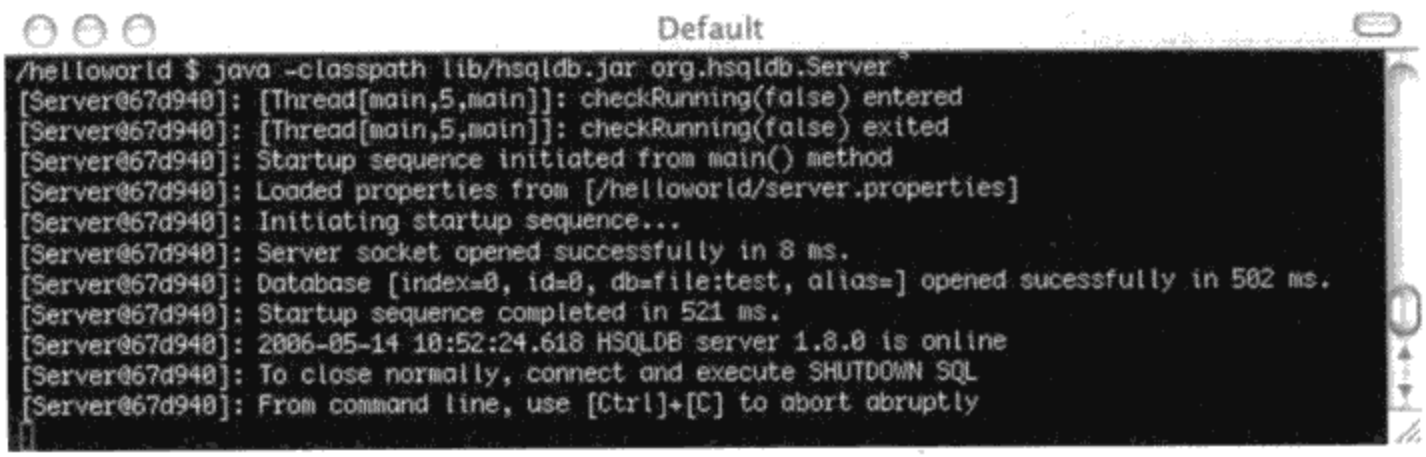


图2-4　从命令行启动HSQLDB服务器

现在有了一个没有内容甚至也没有Schema的空数据库。接下来要创建Schema。

3. 导出数据库Schema

可以用CREATE语句手工编写SQL DDL，并在数据库中执行这个DDL，来创建数据库Schema。或者（且这是更方便的做法）可以让Hibernate完成这项工作，并给你的应用程序创建一个默认的Schema。在Hibernate中自动生成SQL DDL的先决条件始终是Hibernate映射元数据定义，或者在XML映射文件中，或者在Java源代码的注解中。假设你已经随着前几节设计和实现了领域模型类，并用XML编写了映射元数据。

用来生成模式的工具是hbm2ddl；它的类是org.hibernate.tool.hbm2ddl.SchemaExport，因此它有时也称作SchemaExport。

运行这个工具并创建一个Schema有3种方法：

- 可以在常规的构建过程中，在一个Ant任务中运行<hbm2ddl>。
- 可以在应用程序代码中用程序运行SchemaExport，可能在HibernateUtil启动类中。然而，这不常用，因为Schema的生成很少需要编程式的控制。
- 当通过设置Hibernate.hbm2ddl.auto配置属性为create或者create-drop来创建SessionFactory时，可以启用Schema的自动导出。第一个设置在创建SessionFactory时产生DROP语句，之后产生CREATE语句。第二个设置在应用关闭时添加另外的DROP语句，同时关闭SessionFactory——实际上在每一次运行之后留下了一个干净的数据库。

编程式的生成模式很简单：

```
Configuration cfg = new Configuration().configure();
SchemaExport schemaExport = new SchemaExport(cfg);
schemaExport.create(false, true);
```

从一个Configuration中创建一个新的SchemaExport对象；所有设置（例如数据库驱动程序、连接URL等）都传递到SchemaExport构造函数。creat（false，true）调用触发DDL生成程序，没有任何SQL输出到stdout（因为false设置），但DDL立即在数据库中执行（true）。更多信息和其他的设置请见SchemaExport API。

你的开发过程决定了是否应该用hibernate.hbm2ddl.auto配置设置启用自动的Schema导出。许多Hibernate新用户发现SessionFactory中的自动删除和重新创建有点费解。一旦你进一步熟悉Hibernate之后，我们鼓励你去探索这个选项，以便在集成测试中加快周转时间。

这个配置属性的另一个选项是update，开发期间会很有用：它启用内建的SchemaUpdate工具，使得Schema演变更加容易。如果启用，Hibernate就在启动时读取JDBC数据库元数据，并通过把旧的Schema和当前的映射元数据相比较，来创建新的表和约束。注意这项功能取决于JDBC驱动程序提供的元数据的特性，一个缺乏许多驱动程序的领域。因此，这个特性在实际应用中并没有它听起来的那么令人激动和有用。

警告 我们见过Hibernate用户尝试用SchemaUpdate自动更新一个产品数据库的Schema。这可能很快带来灾难，并且你的数据库管理员（DBA）也不会允许。

也可以编程式地运行SchemaUpdate：

```
Configuration cfg = new Configuration().configure();
SchemaUpdate schemaUpdate = new SchemaUpdate(cfg);
schemaUpdate.execute(false);
```

最后再次设置false使SQL DDL不能输出到控制台，并且仅仅在数据库中直接执行语句。如果导出DDL到控制台或者一个文本文件，你的DBA可能会用它作为产生一个优质Schema演变脚本的起点。

开发中有用的另一个hbm2ddl.auto设置是validate。它使SchemaValidator能够在启动时运行。这个工具可以把映射和JDBC数据库进行比较，并告诉你这个Schema和映射是否匹配。也可以编程式地运行SchemaValidator：

```
Configuration cfg = new Configuration().configure();
new SchemaValidator(cfg).validate();
```

如果发现映射和数据库Schema之间不匹配，就会抛出异常。

因为正把系统构建在Ant之上，理想的情况是，每当你需要Schema时，就给Ant构建添加一个schemaexport目标，它从数据库中生成并导出一个新的Schema（请见代码清单2-9）。

代码清单2-9 Schema导出的Ant目标

```
<taskdef name="hibernatetool"
         classname="org.hibernate.tool.ant.HibernateToolTask"
```

```
            classpathref="project.classpath"/>

    <target name="schemaexport" depends="compile, copymetafiles"
        description="Exports a generated schema to DB and file">

        <hibernatetool destdir="${basedir}">
            <classpath path="${build.dir}"/>

            <configuration
                configurationfile="${build.dir}/hibernate.cfg.xml"/>

            <hbm2ddl
                drop="true"
                create="true"
                export="true"
                outputfilename="helloworld-ddl.sql"
                delimiter=";"
                format="true"/>

        </hibernatetool>

    </target>
```

在这个目标中，首先定义了一个你想要使用的新Ant任务HibernateToolTask。这个一般的任务能够完成很多工作——从Hibernate映射元数据中导出一个SQL DDL Schema只是其中的一项工作。你将在本章的所有Ant构建中自始至终使用它。确保在任务定义的classpath中包括了所有Hibernate库、必要的第三方库和JDBC驱动程序。还需要添加hibernate-tools.jar文件，可以在Hibernate Tools下载包中找到它。

Ant目标schemaexport使用这个任务，它还取决于构建目录中编译好的类和复制好的配置文件。<hibernatetool>任务的基本用法始终是一样的：配置是所有代码工件（artifact）生成的起点。此处显示的变种<configuration>，它理解Hibernate XML配置文件，并读取给定配置中列出的所有Hibernate XML映射元数据文件。从这些信息中，生成了一个内部的Hibernate元数据模型（即hbm在各处所代表的含义），导出器随后处理这个模型数据。本章稍后讨论能够读取注解或者反向工程数据库的工具配置。

目标中的另一个元素是所谓的导出器（exporter）。工具配置传递它的元数据信息给你选中的导出器；在前面的例子中，指<hbm2ddl>导出器。就像你可能已经猜到的，这个导出器理解Hibernate元数据模型，并生成SQL DDL。你可以用几种方法控制DDL生成：

- 导出器生成SQL，因此在Hibernate配置文件中设置一种SQL方言是强制的。
- 如果drop设置为true，将首先生成SQL DROP语句，并且所有的表和约束都被移除，如果有的话。如果create设置为true，接下来就生成SQL CREATE语句，来创建所有的表和约束。如果这两个选项你都启用了，实际上就在Ant目标的每一次运行中删除和重新创建数据库Schema。
- 如果export设置为true，所有的DDL语句就直接在数据库中被执行。导出器使用在配置文件中找到的连接设置打开一个和数据库的连接。
- 如果有outputfilename，所有的DDL语句就被写到这个文件，并且文件保存在你配置的

destdir中。附加delimiter字符给写到这个文件的所有SQL语句，并且如果format是启用的，所有SQL语句都会被很好地缩进。

现在通过在WORKDIR中运行ant schemaexport，可以生成、输出并直接把Schema导出到一个文本文件和数据库。所有的表和约束都被删除，然后被再次创建，你就有了一个新的数据库。（忽略任何报告所称的因为表不存在而不能被删除的错误消息。）

检查你的数据库是否在运行，以及它是否有正确的数据库Schema。HSQLDB包括的一个有用工具是一个简单的数据库浏览器。可以用下列Ant目标调用它：

```
<target name="dbmanager" description="Start HSQLDB manager">
    <java
        classname="org.hsqldb.util.DatabaseManagerSwing"
        fork="yes"
        classpathref="project.classpath"
        failonerror="true">
        <arg value="-url"/>
        <arg value="jdbc:hsqldb:hsql://localhost/"/>
        <arg value="-driver"/>
        <arg value="org.hsqldb.jdbcDriver"/>
    </java>
</target>
```

登录以后应该看到图2-5所示的Schema。

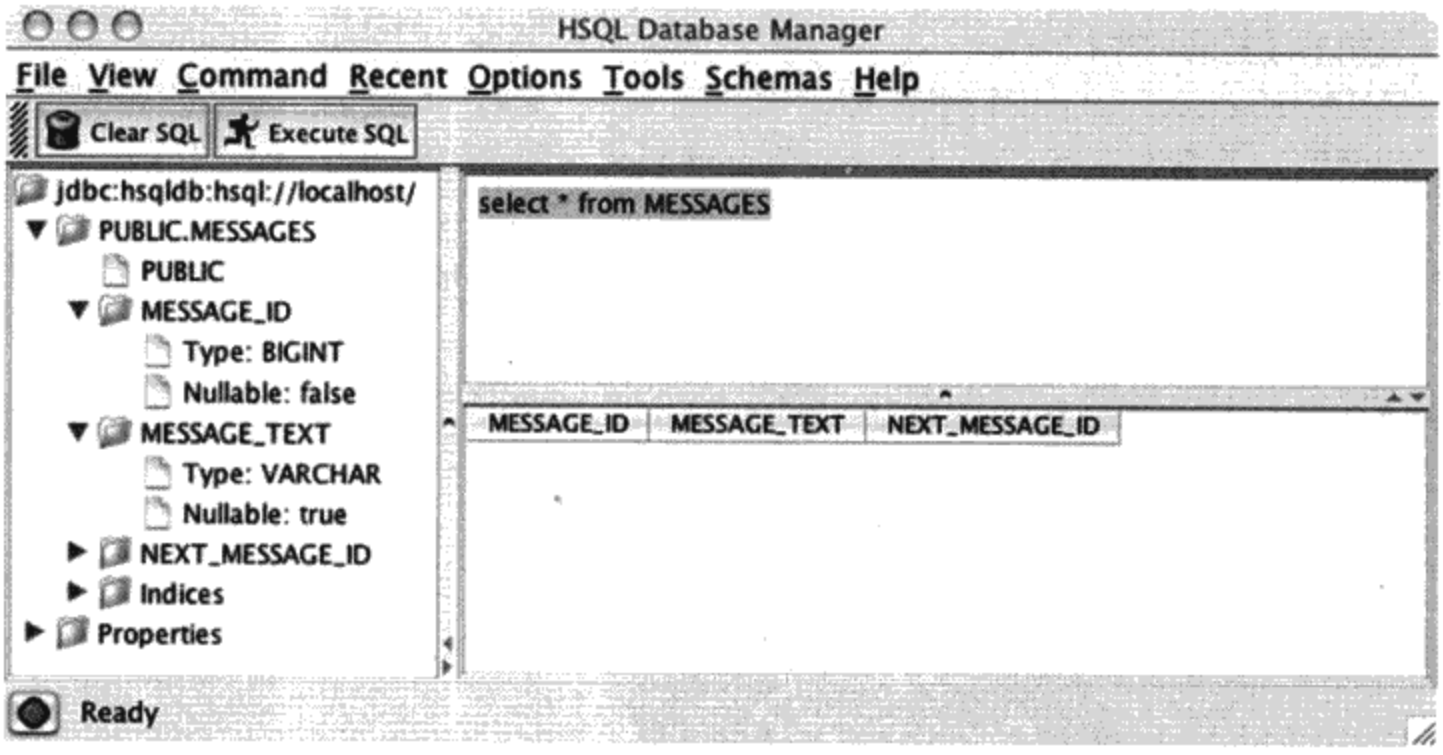


图2-5 HSQLDB浏览器和SQL控制台

用ant run运行应用程序，并观察控制台的Hibernate日志输出。应该看到你的消息被存储、加载和打印了。在HSQLDB浏览器中启动一个SQL查询，直接检查你的数据库内容。

现在有了一个有效的Hibernate基础结构和Ant项目构建。可以跳到下一章，继续编写和映射更复杂的业务类。然而，建议你在“Hello World”应用上花些时间，并且用更多的功能对它进行

扩展。例如，可以尝试不同的HQL查询或者日志选项。别忘了数据库系统仍然在后台运行着，你必须导出一个新的Schema或者停止并删除数据库文件，以便重新得到一个干净的空数据库。

在下一节中，我们用Java Persistence接口和EJB 3.0重新创建一遍这个“Hello World”示例。

2.2　启动 Java Persistence 项目

接下来的几节介绍JPA和新EJB 3.0标准的一些优势，以及注解和标准的编程接口如何简化应用程序开发，甚至与Hibernate比较。显然，如果你不时需要在不同的运行时环境下移植或者部署应用程序，设计和链接到标准的接口就是一个优势。尽管除了可移植性之外，还有许多很好的理由来深入探讨JPA。

现在将引导你进行另一个“Hello World”示例，这次用Hibernate Annotations和Hibernate EntityManager。你将重用在前一节中引入的基础项目架构，以便可以看到JPA与Hibernate的区别之处。介绍完注解和JPA接口之后，将介绍应用如何与其他的托管组件（EJB）整合和交互。本书稍后将讨论更多的应用设计示例；然而，这第一眼就将让你尽快决定一种特定的方法。

2.2.1　使用 Hibernate Annotations

首先利用Hibernate Annotations，用内嵌元数据替代Hibernate XML映射文件。你可能想要在进行下列变化之前复制现有的“Hello World”项目目录——从原生的Hibernate迁移到标准的JPA映射（且稍后迁移到程序代码）。

复制Hibernate Annotations库到WORKDIR/lib目录——查看Hibernate Annotations文档中的必需的库的清单。（在编写本书之时，hibernate-annotations.jar和ejb3-persistence.jar中的API存根是必需的。）

现在删除src/hello/Message.hbm.xml文件。下面用src/hello/Message.java类源代码中的注解来替代这个文件，如代码清单2-10所示。

代码清单2-10　用注解映射Message类

```
package hello;

import javax.persistence.*;

@Entity
@Table(name = "MESSAGES")
public class Message {

    @Id @GeneratedValue
    @Column(name = "MESSAGE_ID")
    private Long id;

    @Column(name = "MESSAGE_TEXT")
    private String text;

    @ManyToOne(cascade = CascadeType.ALL)
    @JoinColumn(name = "NEXT_MESSAGE_ID")
```

```
    private Message nextMessage;

    private Message() {}

    public Message(String text) {
        this.text = text;
    }

    public Long getId() {
        return id;
    }
    private void setId(Long id) {
        this.id = id;
    }

    public String getText() {
        return text;
    }
    public void setText(String text) {
        this.text = text;
    }

     public Message getNextMessage() {
         return nextMessage;
     }
     public void setNextMessage(Message nextMessage) {
         this.nextMessage = nextMessage;
     }
}
```

你在这个更新过的业务类中可能注意到的第一点是javax.persistence接口的引入。这个包内部是映射@Entity类到数据库@Table所需要的所有标准的JPA注解。你把注解放在类的私有字段上，数据库标识符映射从@Id和@GeneratedValue开始。这个JPA持久化的提供程序发现@Id注解在一个字段中，并假设它应该在运行时直接通过字段在对象中访问属性。如果把@Id注解放在getId()方法中，默认时将通过获取方法（getter method）和设置方法启用属性访问。因此，所有其他的注解遵循选中的策略，也被放在字段或者获取方法中。

注意不需要@Table、@Column和@JoinColumn注解。一个实体的所有属性自动被认为是持久化的，使用默认的策略和表/列名称。为了清楚起见，你在此处添加它们，并得到与使用XML映射文件相同的结果。现在比较这两种映射元数据的策略，你会发现注解更加方便，并且明显减少了元数据的行数。注解也是类型安全（type-safe）的，当你输入时它们在IDE中支持自动完成（就像任何其他Java接口一样），并且它们使类和属性的重构更加容易。

如果担心JPA接口的导入将把你的代码绑定到这个包，就应该了解仅当注解被Hibernate在运行时使用的时候，才需要它处在你的classpath中。可以加载和执行这个类，而不用classpath上的JPA接口，只要你不想用Hibernate加载和存储实例。

刚接触注解的开发人员的第二个关注点，有时候与Java源代码中包含配置元数据的内容有关。按定义，配置元数据是可以随应用的每一次部署而改变的元数据，例如表名。JPA有一个简单的解决方案：你可以覆盖或者替代所有用XML元数据文件注解的元数据。本书稍后将介绍如

何实现这个解决方案。

假设这就是你想从JPA中得到的所有东西——用注解取代XML。你不想使用JPA编程接口或者查询语言，你将使用Hibernate Session和HQL。唯一需要对项目所做的其他改变，除了删除目前没用的XML映射文件之外，就是hibernate.cfg.xml中Hibernate配置的一个变化：

```
<!DOCTYPE hibernate-configuration SYSTEM
"http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">

<hibernate-configuration>
<session-factory>
    <!-- ... Many property settings ... -->

    <!-- List of annotated classes-->
    <mapping class="hello.Message"/>
</session-factory>
</hibernate-configuration>
```

Hibernate配置文件之前已经有一个包含所有XML映射文件的清单。这已经被一个包含所有被注解的类清单所替代。如果使用SessionFactory编程式的配置，addAnnotatedClass()方法就会替代addResource()方法：

```
// Load settings from hibernate.properties
AnnotationConfiguration cfg = new AnnotationConfiguration();
// ... set other configuration options programmatically

cfg.addAnnotatedClass(hello.Message.class);
SessionFactory sessionFactory = cfg.buildSessionFactory();
```

注意现在你已经用AnnotationConfiguration代替了基本的Hibernate的Configuration接口——这个扩展识别被注解的类。至少，你还需要改变HibernateUtil中的初始化器以便使用这个接口。如果用一个Ant目标导出数据库Schema，那么就在build.xml文件中用<annotationconfiguration>取代<configuration>。

这就是用注解运行“Hello World”应用程序需要改变的所有内容。试着再次运行它，可能通过一个全新的数据库。

注解元数据也可以是全局的，虽然“Hello World”应用程序中并不需要。全局的注解元数据被放在名为package-info.java的文件中，它处在一个特定的包目录中。除了列出被注解的类之外，还需要添加包含全局元数据的包到你的配置中。例如，在一个Hibernate XML配置文件中，要添加以下代码：

```
<!DOCTYPE hibernate-configuration SYSTEM
"http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">

<hibernate-configuration>
<session-factory>
    <!-- ... Many property settings ... -->

    <!-- List of annotated classes-->
    <mapping class="hello.Message"/>

    <!-- List of packages with package-info.java -->
```

```
    <mapping package="hello"/>
</session-factory>
</hibernate-configuration>
```

或者可以用编程式的配置实现相同的结果：

```
// Load settings from hibernate.properties
AnnotationConfiguration cfg = new AnnotationConfiguration();
// ... set other configuration options programmatically

cfg.addClass(hello.Message.class);

cfg.addPackage("hello");

SessionFactory sessionFactory = cfg.buildSessionFactory();
```

把这一步再深入，用使用JPA的代码代替加载和存储消息的原生的Hibernate代码。利用Hibernate Annotations和Hibernate EntityManager，可以创建可移植的和标准兼容的映射和数据访问代码。

2.2.2 使用 Hibernate EntityManager

Hibernate EntityManager是围绕提供JPA编程接口的Hibernate Core的一个包装，支持JPA实体实例的生命周期，并允许你用标准的Java Persistence查询语言编写查询。由于JPA功能是Hibernate原生能力的一个子集，你可能想知道为什么应当在Hibernate的顶层使用EntityManager包。本节稍后将介绍各种优势，但是一旦你给Hibernate EntityManager配置了项目，马上就会看到一种特殊的简化：你不再非得在配置文件中列出所有被注解的类（或者XML映射文件）了。

我们来修改“Hello World”项目并给它准备好全面的JPA兼容性。

1. 基本的JPA配置

SessionFactory在Hibernate应用程序中表示一个特定的逻辑数据存储配置。EntityManager Factory在JPA应用程序中扮演着同样的角色，你用配置文件或者在应用程序代码中配置EntityManager Factory（EMF），就像配置SessionFactory一样。EMF的配置，与一组映射元数据（通常是被注解的类）一起，被称作持久化单元（persistence unit）。

持久化单元的概念也包括应用程序的包，但是我们想让“Hello World”尽可能地保持简单；假设你要从一个标准化的JPA配置开始，没有特殊的包。不仅内容，就连持久化单元的JPA配置文件的名称和位置也是标准的。

创建名为WORKDIR/etc/META-INF的目录，并把名为persistence.xml的基础配置文件放在那个目录下，如代码清单2-11所示：

代码清单2-11 持久化单元配置文件

```
<persistence xmlns="http://java.sun.com/xml/ns/persistence"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
    http://java.sun.com/xml/ns/persistence/persistence_1_0.xsd"
   version="1.0">

   <persistence-unit name="helloworld">
      <properties>
```

```
            <property name="hibernate.ejb.cfgfile"
                value="/hibernate.cfg.xml"/>
        </properties>
    </persistence-unit>

</persistence>
```

每一个持久化单元都需要一个名称，在这个例子中为helloworld。

说明　前面的持久化单元配置文件中的XML首部，声明了应该使用什么模式，并且它始终相同。以后的例子中我们将把它省略，并假设你会添加它。

使用任意数量的属性对持久化单元进行进一步配置，这些属性全是特定于供应商的。前一个例子中的属性hibernate.ejb.cfgfile，就像是一个杂物箱。它引用包含着这个持久化单元所有设置的hibernate.cfg.xml文件(在类路径的根目录中)——你正在重用现有的Hibernate配置。随后，将把所有的配置细节移进persistence.xml文件，但是现在你更关注使用JPA运行“Hello World”。

JPA标准指出，persistence.xml文件要出现在被部署的持久化单元的META-INF目录中。因为你并没有真的打包和部署持久化单元，这意味着必须把persistence.xml复制到构建输出目录的META-INF目录下。修改build.xml，并把下列代码添加到copymetafiles目标：

```
<property name="src.etc.dir" value="etc"/>
<target name="copymetafiles">

    <!-- Copy metadata to build -->
    <copy todir="${build.dir}">
      <fileset dir="${src.java.dir}">
        <patternset refid="meta.files"/>
      </fileset>
    </copy>
    <!-- Copy configuration files from etc/ -->
    <copy todir="${build.dir}">
      <fileset dir="${src.etc.dir}">
        <patternset refid="meta.files"/>
      </fileset>
    </copy>

</target>
```

WORKDIR/etc目录下与meta.files模式相匹配的所有内容都被复制到构建输出目录，这是运行时classpath的一部分。

让我们用JPA重写主应用程序代码。

2. 使用JPA的“Hello World”

这些是Java Persistence中主要的编程接口：

- javax.persistence.Persistence——给EntityManagerFactory的创建提供一种静态方法的一个启动类。

- javax.persistence.EntityManagerFactory——等同于Hibernate SessionFactory。这个运行时对象表示一个特定的持久化单元。它是线程安全的，通常被当作一个单例（singleton）处理，并给EntityManager实例的创建提供方法。
- javax.persistence.EntityManager——等同于Hibernate Session。这个单线程、非共享的对象表示数据访问的一个特定工作单元。它提供方法去管理实体实例的生命周期并创建Query实例。
- javax.persistence.Query——等同于Hibernate Query。一个对象是一种特定的JPA查询语言或者原生的SQL查询表示法，它允许参数的安全绑定，并给查询的执行提供各种各样的方法。
- javax.persistence.EntityTransaction——等同于Hibernate Transaction，在Java SE环境中用于RESOURCE_LOCAL事务的划分。在Java EE中，依赖JTA标准的javax.transaction.UserTransaction接口进行编程式的事务划分。

要使用JPA接口，需要复制必要的库到WORKDIR/lib目录下；在Hibernate EntityManager绑定的文件中查看最新的列表。然后可以在WORKDIR/src/hello/HelloWorld.java中重写代码，并从Hibernate接口切换到JPA接口（请见代码清单2-12）。

代码清单2-12 使用JPA的“Hello World”主应用程序代码

```
package hello;

import java.util.*;
import javax.persistence.*;

public class HelloWorld {

    public static void main(String[] args) {

        // Start EntityManagerFactory
        EntityManagerFactory emf =
                Persistence.createEntityManagerFactory("helloworld");

        // First unit of work
        EntityManager em = emf.createEntityManager();
        EntityTransaction tx = em.getTransaction();
        tx.begin();

        Message message = new Message("Hello World");
        em.persist(message);

        tx.commit();
        em.close();

        // Second unit of work
        EntityManager newEm = emf.createEntityManager();
        EntityTransaction newTx = newEm.getTransaction();
        newTx.begin();

        List messages = newEm
            .createQuery("select m from Message m
        ➥ order by m.text asc")
```

```
            .getResultList();

        System.out.println( messages.size() + " message(s) found" );

        for (Object m : messages) {
            Message loadedMsg = (Message) m;
            System.out.println(loadedMsg.getText());
        }

        newTx.commit();
        newEm.close();

        // Shutting down the application
        emf.close();
    }
}
```

在这段代码中你可能注意到的第一点是不再有Hibernate导入，只有javax.perstence.*。用一个对Persistence的静态调用和持久化单元的名称来创建EntityManagerFactory。其余代码应该是不言自明的——就像使用Hibernate一样使用JPA，虽然在API中有些小差别，并且方法的名称也略有不同。此外，没有对基础结构的静态初始化使用HibernateUtil类；你可以编写一个JPAUtil类，并且如果愿意，可以把EntityManagerFactory的创建移到那去，或者可以移除目前没用的WORKDIR/src/persistence包。

JPA也支持编程式的配置，通过选项的映射（map）：

```
Map myProperties = new HashMap();
myProperties.put("hibernate.hbm2ddl.auto", "create-drop");
EntityManagerFactory emf =
 Persistence.createEntityManagerFactory("helloworld", myProperties);
```

定制的程序属性覆盖了已经在persistence.xml配置文件中设置的任何属性。

试着用一个全新的数据库运行这个移植的HelloWorld代码。你在屏幕上应该看到与使用原生的Hibernate完全相同的日志输出——JPA持久化提供程序引擎是Hibernate。

3. 元数据的自动侦测

之前我们承诺过你将不一定要在配置中列出所有被注解的类或者XML映射文件，但是它们仍然在hibernate.cfg.xml中。我们来启用JPA的自动侦测特性。

给org.hibernate包切换到DEGUG日志后再次运行“Hello World”应用程序。日志中应该出现一些额外的行：

```
...
Ejb3Configuration:141
    - Trying to find persistence unit: helloworld
Ejb3Configuration:150
    - Analyse of persistence.xml:
        file:/helloworld/build/META-INF/persistence.xml
PersistenceXmlLoader:115
    - Persistent Unit name from persistence.xml: helloworld
Ejb3Configuration:359
    - Detect class: true; detect hbm: true
JarVisitor:178
```

```
    - Searching mapped entities in jar/par: file:/helloworld/build
JarVisitor:217
    - Filtering: hello.HelloWorld
JarVisitor:217
    - Filtering: hello.Message
JarVisitor:255
    - Java element filter matched for hello.Message
Ejb3Configuration:101
    - Creating Factory: helloworld
...
```

启动时，Persistence.createEntityManagerFactory()方法试图查找具名helloworld的持久化单元的位置。它在classpath中搜索所有META-INF/persistence.xml文件，然后如果发现匹配就配置EMF。日志的第二部分显示了一些你可能意想不到的东西。JPA持久化提供程序试图在构建输出目录下找到所有被注解的类和所有Hibernate XML映射文件。不需要hibernate.cfg.xml中被注解类的列表（或者XML映射文件的列表），因为hello.Message这个被注解的实体类已经找到了。

不是只从hibernate.cfg.xml中移除这个单独的不必要选项，而是要移除整个文件，并把所有的配置细节移到persistence.xml里面（请见代码清单2-13）。

代码清单2-13 完整的持久化单元配置文件

```
<persistence-unit name="helloworld">
 <provider>org.hibernate.ejb.HibernatePersistence</provider>

 <!-- Not needed, Hibernate supports auto-detection in JSE
     <class>hello.Message</class>
  -->

<properties>
    <property name="hibernate.archive.autodetection"
        value="class, hbm"/>

    <property name="hibernate.show_sql" value="true"/>
    <property name="hibernate.format_sql" value="true"/>

    <property name="hibernate.connection.driver_class"
              value="org.hsqldb.jdbcDriver"/>
    <property name="hibernate.connection.url"
              value="jdbc:hsqldb:hsql://localhost"/>
    <property name="hibernate.connection.username"
              value="sa"/>

    <property name="hibernate.c3p0.min_size"
              value="5"/>
    <property name="hibernate.c3p0.max_size"
              value="20"/>
    <property name="hibernate.c3p0.timeout"
              value="300"/>
    <property name="hibernate.c3p0.max_statements"
              value="50"/>
```

```
        <property name="hibernate.c3p0.idle_test_period"
                  value="3000"/>

        <property name="hibernate.dialect"
                  value="org.hibernate.dialect.HSQLDialect"/>

        <property name="hibernate.hbm2ddl.auto" value="create"/>

    </properties>
</persistence-unit>
```

在这个配置文件中有3个值得关注的新元素。首先，你设置了应该为这个持久化单元所用的一个显式的<provider>。这通常只在你同时使用几个JPA实现时才需要，但是，我们当然希望Hibernate是唯一的实现。接下来，如果你在非Java EE的环境中部署，规范要求列出包含<class>元素的所有被注解的类——Hibernate处处支持映射元数据的自动侦测，使它变成可选。最后，Hibernate配置设置archive.autodetection告知Hibernate要自动扫描哪些元数据：被注解的类（class）和（或）Hibernate XML映射文件（hbm）。默认情况下，Hibernate EntityManager对两者都进行扫描。配置文件的其他内容包含本章稍早我们在一般的hibernate.cfg.xml文件中解释过和用过的所有选项。

被注解的类和XML映射文件的自动侦测是JPA一个很棒的特性。通常只有在Java EE应用程序服务器中才可用；至少，这是EJB 3.0规范保证的东西。但是，Hibernate作为一个JPA提供程序，也在简单的Java SE中实现它，虽然你也许不能够使用与任何其他JPA提供程序完全相同的配置。

现在，你已经构建了一个完全遵循JPA规范的应用程序。项目目录看起来应该像这样（注意我们也把log4j.properties移到了etc/目录下）：

```
WORKDIR
+etc
  log4j.properties
  +META-INF
   persistence.xml
+lib
  <all required libraries>
+src
  +hello
    HelloWorld.java
    Message.java
```

所有的JPA配置设置都捆绑在persistence.xml中，所有的映射元数据都包括在Message类的Java源代码中，Hibernate在启动时自动扫描并找到元数据。相较于纯粹的Hibernate，现在你拥有这些好处：

- 被部署元数据的自动扫描，这在大项目中是个重要的特性。如果几百个实体由一个大团队开发时，维护一系列被注解的类或者映射文件就变得很困难。
- 标准的和简化的配置，配置文件有标准的位置，以及一个部署概念——持久化单元——在一个应用程序文档（EAR）中包有几个单元（JAR）的更大项目中，它有着更多的优势。
- 标准的数据访问代码、实体实例的生命周期和完全可移植的查询。应用程序中没有私有的导入。

这些仅仅是JPA的其中一些优势。如果你把它与完全的EJB 3.0编程模型和其他的托管组件结合起来，就会看到它真正的威力。

2.2.3 引入 EJB 组件

当你还使用EJB 3.0会话bean和消息驱动的bean（和其他Java EE 5.0标准）时，Java Persistence就开始初露锋芒。EJB 3.0规范已经被设计为允许持久化的整合，因此你可以在Bean方法范围获得自动的事务划分，或者跨越状态会话EJB生命周期的一个持久化上下文（如Session）。

本节将从一个托管的Java EE环境中的EJB 3.0和JPA开始，你将再次通过修改“Hello World”应用程序学习基础的知识。首先需要Java EE环境——提供Java EE服务的运行时容器。有两种方法可以获得它：

- 可以安装一个支持EJB 3.0和JPA的完全Java EE 5.0应用程序服务器。几个开源（Sun GlassFish、JBoss AS、Object Web EasyBeans）和其他私有许可的选择在编写本书时市面上已有出售，当你读到本书时可能有更多的选择了。
- 可以安装一个模块化的服务器，从完全Java EE 5.0包中选择，仅仅提供你需要的服务。至少，你可能想要一个EJB 3.0容器、JTA事务服务和一个JNDI注册。编写本书之时，只有JBoss AS在一个易于定制的包中提供了模块化的Java EE 5.0服务。

为了使事情保持简单，并展示从EJB 3.0开始有多么容易，你将安装和配置模块化的JBoss Application Server，并只启用所需要的Java EE 5.0服务。

1. 安装EJB容器

到http://jboss.com/products/ejb3下载这个模块化的可嵌入服务器，并解压下载到的文档。把服务器带来的所有库复制到项目的WORKDIR/lib目录下，并将所有被包括的配置文件复制到WORKDIR/src目录。现在你应该具备下列目录布局：

```
WORKDIR
+etc
  default.persistence.properties
  ejb3-interceptors-aop.xml
  embedded-jboss-beans.xml
  jndi.properties
  log4j.properties
  +META-INF
    helloworld-beans.xml
    persistence.xml
+lib
  <all required libraries>
+src
  +hello
    HelloWorld.java
    Message.java
```

JBoss可嵌入的服务器依赖于Hibernate提供Java Persistence，因此default.persistence.properties文件包含全部部署所需的Hibernate默认设置（例如JTA整合设置）。ejb3-interceptors-aop.xml和embedded-jboss-beans.xml配置文件包含服务器的服务配置——你可以看一下这些文件，但是现在

不需要修改它们。在编写本书之时，默认启用的服务是JNDI、JCA、JTA和EJB 3.0容器——正是你所需要的。

为了迁移“Hello World”应用程序，需要一个托管的数据源，这是一个由可嵌入的服务器处理的数据库连接。配置一个托管数据源最容易的方法是，添加一个把数据源当作托管服务部署的配置文件。创建代码清单2-14中的文件为WORKDIR/ect/META-INF/helloworld-beans.xml。

代码清单2-14　JBoss服务器的数据源配置文件

```
<?xml version="1.0" encoding="UTF-8"?>
<deployment xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="urn:jboss:bean-deployer bean-deployer_1_0.xsd"
  xmlns="urn:jboss:bean-deployer:2.0">

  <!-- Enable a JCA datasource available through JNDI -->
  <bean name="helloWorldDatasourceFactory"
   class="org.jboss.resource.adapter.jdbc.local.LocalTxDataSource">

      <property name="jndiName">java:/HelloWorldDS</property>

      <!-- HSQLDB -->
      <property name="driverClass">
          org.hsqldb.jdbcDriver
      </property>
      <property name="connectionURL">
          jdbc:hsqldb:hsql://localhost
      </property>
      <property name="userName">sa</property>

      <property name="minSize">0</property>
      <property name="maxSize">10</property>
      <property name="blockingTimeout">1000</property>
      <property name="idleTimeout">100000</property>

      <property name="transactionManager">
          <inject bean="TransactionManager"/>
      </property>
      <property name="cachedConnectionManager">
          <inject bean="CachedConnectionManager"/>
      </property>
      <property name="initialContextProperties">
          <inject bean="InitialContextProperties"/>
      </property>
  </bean>

  <bean name="HelloWorldDS" class="java.lang.Object">
      <constructor factoryMethod="getDatasource">
          <factory bean="helloWorldDatasourceFactory"/>
      </constructor>
  </bean>

</deployment>
```

同样，XML首部和Schema声明对于这个例子也不重要。你创建了两个bean：第一个是可以产生第二个bean类型的工厂。LocalTxDataSource现在实际上是数据库连接池，所有的连接池设

置都可在这个工厂使用。该工厂把一个托管数据源绑定在JNDI名称java:/HelloWorldDS下面。

第二个bean配置声明，如果另一个服务在JNDI注册中查找名为HelloWorldDS的注册对象，它应该如何被实例化。你的“Hello World”应用程序需要这个名称下的数据源，服务器在LocalTxDataSource工厂中调用getDatasource()来获得它。

还要注意我们在属性值中添加了一些换行，使它更易于阅读——在实际配置文件中你不应该这么做（除非你的数据库用户名称包含换行）。

2. 配置持久化单元

接下来，需要改变“Hello World”应用程序的持久化单元配置，来访问一个托管的JTA数据源，而不是本地资源的连接池。把WORKDIR/etc/META-INF/persistence.xml文件变成如下：

```
<persistence ...>

   <persistence-unit name="helloworld">
      <jta-data-source>java:/HelloWorldDS</jta-data-source>
      <properties>
          <property name="hibernate.show_sql" value="true"/>
          <property name="hibernate.format_sql" value="true"/>
          <property name="hibernate.dialect"
                    value="org.hibernate.dialect.HSQLDialect"/>
          <property name="hibernate.hbm2ddl.auto" value="create"/>

      </properties>
   </persistence-unit>

</persistence>
```

你去除了许多不再有关系的Hibernate配置选项，例如连接池和数据库连接设置。反之，你用JNDI中绑定的数据源名称设置了一个<jta-data-source>属性。别忘了你仍然需要配置正确的SQL方言，以及没有出现在default.persistence.properties中的任何其他的Hibernate选项。

现在，环境的安装和配置完成了，（我们很快就要介绍jndi.properties文件的用途）并且能够用EJB重写应用程序代码。

3. 编写EJB

设计和创建含有多个托管组件的应用程序有许多种方法。这个“Hello World”应用程序太简单，不足以介绍详细的例子，因此我们只介绍最基础的EJB类型：一个*无状态会话*bean（stateless session bean）。（你已经见过实体类——被注解的简单Java类，能够有持久化实例。注意术语实体bean仅仅指旧的EJB 2.1实体bean；EJB 3.0和Java Persistence给简单的实体类标准化了一个轻量级的编程模型。）

每个EJB会话bean都需要一个*业务接口*（business interface）。这不是一个需要实现预设方法或者扩展现有方法的特殊接口，它是简单的Java。在WORKDIR/src/hello包中创建下列接口：

```
package hello;

public interface MessageHandler {

    public void saveMessages();

    public void showMessages();
}
```

MessageHandler能够保存和显示消息；它很容易理解。实际的EJB实现这个业务接口，这个接口被默认当作一个本地接口（也就是说，远程的EJB客户端程序无法调用它），请见代码清单2-15。

代码清单2-15　“Hello World”EJB会话bean应用程序代码

```
package hello;

import javax.ejb.Stateless;
import javax.persistence.*;
import java.util.List;

@Stateless
public class MessageHandlerBean implements MessageHandler {

    @PersistenceContext
    EntityManager em;

    public void saveMessages() {
        Message message = new Message("Hello World");
        em.persist(message);
    }

    public void showMessages() {
        List messages =
            em.createQuery("select m from Message m
            ➥ order by m.text asc")
                .getResultList();

        System.out.println(messages.size() + " message(s) found:");

        for (Object m : messages) {
            Message loadedMsg = (Message) m;
            System.out.println(loadedMsg.getText());
        }

    }
}
```

在这个实现中要观察几件值得关注的事。首先，这是个简单的Java类，没有强烈地依赖于任何其他的包。它变成了一个只有单个元数据注解@Stateless的EJB。EJB支持容器托管的服务，因此可以应用@PersistenceContext注解，并且每当调用这个无状态bean中的方法时，服务器就注入一个全新的EntityManager实例。每一种方法也都通过容器自动分配一个事务。事务在调用方法时启动，并在方法返回时提交。（当方法内部抛出异常时它则回滚。）

你现在可以修改HelloWorld主类并把所有存储和显示消息的工作委托给MessageHandler。

4．运行应用程序

“Hello World”应用程序的主类在JNDI注册中找到MessageHandler的无状态会话bean之后调用它。显然，托管环境和整个应用程序服务器（包括JNDI注册）都必须先被启动。你在HelloWorld.java的main()方法中完成所有这些工作（请见代码清单2-16）。

代码清单2-16　“Hello World”主应用程序代码调用EJB

```
package hello;
```

```
import org.jboss.ejb3.embedded.EJB3StandaloneBootstrap;
import javax.naming.InitialContext;

public class HelloWorld {

  public static void main(String[] args) throws Exception {

    // Boot the JBoss Microcontainer with EJB3 settings, automatically
    // loads ejb3-interceptors-aop.xml and embedded-jboss-beans.xml
    EJB3StandaloneBootstrap.boot(null);

    // Deploy custom stateless beans (datasource, mostly)
    EJB3StandaloneBootstrap
        .deployXmlResource("META-INF/helloworld-beans.xml");

    // Deploy all EJBs found on classpath (slow, scans all)
    // EJB3StandaloneBootstrap.scanClasspath();

    // Deploy all EJBs found on classpath (fast, scans build directory)
    // This is a relative location, matching the substring end of one
    // of java.class.path locations. Print out the value of
    // System.getProperty("java.class.path") to see all paths.
    EJB3StandaloneBootstrap.scanClasspath("helloworld-ejb3/bin");

    // Create InitialContext from jndi.properties
        InitialContext initialContext = new InitialContext();

        // Look up the stateless MessageHandler EJB
        MessageHandler msgHandler = (MessageHandler) initialContext
                                    .lookup("MessageHandlerBean/local");

        // Call the stateless EJB
        msgHandler.saveMessages();
        msgHandler.showMessages();

        // Shut down EJB container
        EJB3StandaloneBootstrap.shutdown();
    }
}
```

main()中的第一个命令启动了服务器的内核，并部署了在服务配置文件中找到的基础服务。接下来，你稍早在helloworld-beans.xml中创建的数据源工厂配置得到了部署，且数据源被容器绑定到JNDI。从那时起，容器就准备好部署EJB了。部署所有EJB最容易的（但经常不是最快的）方法是让容器在整个classpath中搜索任何具有EJB注解的类。要学习许多其他可用的部署选项，请查阅下载文件中捆绑的JBoss AS文档。

要查找EJB，需要一个InitialContext，这是JNDI注册的入口点。如果实例化InitialContext，Java就会自动在classpath中查找文件jndi.properties。要在WORKDIR/etc中，用与JBoss服务器的JNDI注册配置相匹配的设置创建这个文件：

```
java.naming.factory.initial
    ➥ org.jnp.interfaces.LocalOnlyContextFactory
java.naming.factory.url.pkgs org.jboss.naming:org.jnp.interfaces
```

你不需要准确了解这个配置的含义，但是它主要把InitialContext指向在本地虚拟机上运行的一个JNDI注册（远程的EJB客户端程序调用需要一个支持远程通信的JNDI服务）。

默认情况下，按一个实现类的名称查找MessageHandler bean，本地的接口带有/local后缀。EJB如何命名，它们如何被绑定到JNDI，以及你如何不同地查找它们，如何定制它们，这些都是JBoss服务器的默认值。

最后，调用MessageHandler EJB，并让它在两个单元内自动完成所有工作——每个方法调用将产生各自的事务。

这样，就用托管的EJB组件和集成的JPA完成了我们的第一个例子。你可能已经知道自动的事务划分和EntityManager注入如何改善代码的可读性。接下来，将介绍有状态会话bean如何用事务的语义帮助你实现用户和应用程序之间的复杂会话。此外，EJB组件不包含任何不必要的胶合代码（glue code）或者基础结构方法，并且它们在任何EJB 3.0容器中都是完全可重用、可移植和可执行的。

说明　持久化单元的包——我们之前没有过多谈论持久化单元的包——你不需要为了任何部署而打包“Hello World”例子。然而，如果要使用特性例如一个完全应用程序服务器上的热重新部署（hot redeployment），需要正确地打包应用。这包括JAR、WAR、EJB-JAR和EAR的一般组合。部署和打包经常也是特定于供应商的，因此应该参考你应用程序服务器的文档了解更多信息。JPA持久化单元的作用范围可以到JAR、WAR和EJB-JAR，这意味着这些文档中的一个或者几个包含了所有被注解的类，以及带有这个特定单元的所有设置的一个META-INF/persistence.xml配置文件。可以把一个或者几个JAR、WAR和EJB-JAR包在单个企业应用文档EAR中。你的应用程序服务器应该正确地发现所有持久化单元，并自动地创建必要的工厂。用@PersistenceContext注解中的一个单元名称属性，指示容器从一个特定的单元注入EntityManager。

应用程序的完全可移植性通常不是使用JPA或者EJB 3.0的主要原因。毕竟，你决定了要用Hibernate作为JPA持久化的提供程序。来看看你可以如何退回并不时使用Hibernate原生的特性。

2.2.4　切换到 Hibernate 接口

决定用Hibernate作为JPA持久化的提供程序有几个原因：首先，Hibernate是一个很好的JPA实现，它提供许多不影响代码的选项。例如，可以在JPA配置中启用Hibernate的二级数据高速缓存，并透明地提升应用的性能和可量测性，而不需要动任何代码。

其次，必要时可以使用原生的Hibernate映射或者API。3.3节讨论映射的混合（特别是注解），但是这里我们要说明，需要的时候，你如何能够可以在JPA应用中使用Hibernate API。显然，把Hibernate API导入到代码中使得把代码移植到一个不同的JPA提供程序变得更困难了。因此，正确地隔离这部分代码变得尤其重要，或者至少用文档说明你为什么和何时使用过一个原生的Hibernate特性。

你可以从它们对应的JPA接口退回到Hibernate API，获得Configuration、SessionFactory，必要时甚至是Session。

例如，不必使用Persistence静态的类创建EntityManagerFactory，相反使用Hibernate的Ejb3Configuration：

```
Ejb3Configuration cfg = new Ejb3Configuration();
EntityManagerFactory emf =
  cfg.configure("/custom/hibernate.cfg.xml")
     .setProperty("hibernate.show_sql", "false")
     .setInterceptor( new MyInterceptor() )
     .addAnnotatedClass( hello.Message.class )
     .addResource( "/Foo.hbm.xml")
     .buildEntityManagerFactory();

AnnotationConfiguration
        hibCfg = cfg.getHibernateConfiguration();
```

Ejb3Configuration是一个新的接口，它与常规的Hibernate Configuration一样，并没有进行扩展（这是一个实现细节）。这意味着，例如，可以从Ejb3Configuration中获得简单的Annotation Configuration对象，并通过编程把它传递到SchemaExport实例。

如果需要编程式地控制二级高速缓存区，SessionFactory接口就很有用。可以先通过转换EntityManagerFactory获得SessionFactory：

```
HibernateEntityManagerFactory hibEMF =
        (HibernateEntityManagerFactory) emf;
SessionFactory sf = hibEMF.getSessionFactory();
```

使用同样的方法可以从EntityManager中获得Session：

```
HibernateEntityManager hibEM =
    (HibernateEntityManager) em;
Session session = hibEM.getSession();
```

这不是从标准的EntityManager中获得原生API的唯一方法。JPA规范支持返回底层实现的getDelegate()方法：

```
Session session = (Session) entityManager.getDelegate();
```

或者可以获得一个注入到EJB组件的Session（虽然这仅适用于JBoss应用程序服务器）：

```
@Stateless
public class MessageHandlerBean implements MessageHandler {

    @PersistenceContext
    Session session;

    ...
}
```

极少数情况下，能够从Hibernate Session退回到简单的JDBC接口：

```
Connection jdbcConnection = session.connection();
```

这最后一个选项有一些警告：不允许你关闭从Hibernate中获得的JDBC Connection——它会自动进行。这个规则的例外情况是，在一个依赖于积极的连接释放的环境中，这意味着在JTA或者CMT环境中，你必须关闭在应用程序代码中返回的连接。

直接访问JDBC连接的一种更好更安全的方法是通过Java EE 5.0中的资源注入。在一个EJB、EJB监听器、servlet、servlet过滤器甚至JavaServer Faces backing bean中，注解一个字段或者设置方法，像下面这样：

```
@Resource(mappedName="java:/HelloWorldDS") DataSource ds;
```

目前为止，假设你是在一个没有涉及遗留应用程序代码（或者现有的数据库Schema）的新Hibernate或者新JPA项目上进行。现在转换一下视角，考虑一个自底而上的开发过程。在这样一个场景中，你或许想要自动地从现有的数据库Schema中反向工程所需要创建的东西。

2.3　反向工程遗留数据库

当映射遗留数据库时，第一步可能涉及一个自动的反向工程过程。毕竟，实体Schema已经存在于你的数据库系统中。为了使这项工作更容易些，Hibernate配有一组工具，可以从这个元数据（包括XML映射文件和Java源代码）中读取Schema，并生成各种需要创建的东西。所有这些都是基于模板的，因此许多定制工作成了可能。

可以用工具和Ant构建中的任务控制反向工程的进行。你之前用来从Hibernate映射元数据中导出SQL DDL的`HibernateToolTask`，具有更多的选项，大部分选项都与反向工程有关，关于XML映射文件、Java代码，甚至整个应用程序骨架如何可以从现有的数据库Schema中自动生成。

我们将首先介绍如何编写一个可以把现有的数据库加载到Hibernate元数据模型中的Ant目标。接下来，应用各种导出组件，并从数据库表和列中生成XML文件、Java代码和其他有用的工件。

2.3.1　创建数据库配置

假设你有一个新的WORKDIR目录，里面除了lib目录（以及它的一般内容）和空的src目录之外什么也没有。要从现有的数据库中生成映射和代码，首先需要创建一个包含数据库连接设置的配置文件：

```
hibernate.dialect = org.hibernate.dialect.HSQLDialect
hibernate.connection.driver_class = org.hsqldb.jdbcDriver
hibernate.connection.url = jdbc:hsqldb:hsql://localhost
hibernate.connection.username = sa
```

直接把这个文件保存在WORKDIR中，并命名为helloworld.db.properties。这里显示的四行是连接到数据库并读取所有表和列所需的最少代码。你可能已经创建了一个Hibernate XML配置文件，而不是helloworld.db.properties，但是没有必要把这弄得比需要的更复杂。

接下来编写Ant目标。在项目的一个build.xml文件中，添加下列代码：

```
<taskdef name="hibernatetool"
     classname="org.hibernate.tool.ant.HibernateToolTask"
     classpathref="project.classpath"/>

<target name="reveng.hbmxml"
        description="Produces XML mapping files in src directory">

    <hibernatetool destdir="${basedir}/src">

        <jdbcconfiguration
            propertyfile="${basedir}/helloworld.db.properties"
            revengfile="${basedir}/helloworld.reveng.xml"/>
```

```
        <hbm2hbmxml/> <!-- Export Hibernate XML files -->
        <hbm2cfgxml/> <!-- Export a hibernate.cfg.xml file -->
    </hibernatetool>
</target>
```

Ant的这个HibernateToolTask定义和以前的一样。假设你将重用前面章节中介绍过的大部分构建文件，并且如project.classpath这样的引用也是一样的。<hibernatetool>任务用WORKDIR/ src设置为所有生成工件的默认目标目录。

<jdbconfiguration>是一个Hibernate工具配置，可以通过JDBC连接到数据库，以及从数据库目录中读取JDBC元数据。通常用两个选项对它进行配置：数据库连接设置（属性文件）和一个可选的反向工程定制文件。

然后，由工具配置生成的元数据被导入到导出器。这个范例的Ant目标命名了这样两个导出器：hbm2hbmxml导出器，就像你能从它的命名中猜到的那样，它从配置中取出Hibernate元数据（hbm），并生成Hibernate XML映射文件；第二个导出器可以准备一个hibernate.cfg.xml文件，该文件列出所有生成的XML映射文件。

在谈论这些以及各种其他的导出器之前，要花点时间讨论反向工程定制文件以及如何使用它。

2.3.2 定制反向工程

JDBC元数据——也就是说，你可以通过JDBC从一个数据库中读取有关它自身的信息——通常不足以创建一个完美的XML映射文件，至于Java应用代码更不必说了。反过来也可能是对的：你的数据库可能包含你想要忽略（如特定的表或者列）的信息，或者你希望用非默认的策略转换的信息。可以用一个使用XML语法的反向工程配置文件定制反向工程的过程。

假设你正在使用单独的MESSAGE表和仅有的几个列，对本章前面创建的“Hello World”数据库反向工程。使用helloworld.reveng.xml文件，如代码清单2-17所示，可以定制这个反向工程。

代码清单2-17 被定制反向工程的配置

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE hibernate-reverse-engineering SYSTEM (1)
  "http://hibernate.sourceforge.net/
    ➥ hibernate-reverse-engineering-3.0.dtd">

<hibernate-reverse-engineering>

    <table-filter match-name=".*" package="hello"/> (2)

    <table name="MESSAGES" schema="PUBLIC" class="Message"> (3)

        <primary-key> (4)

   <generator class="increment"/>
   <key-column name="MESSAGE_ID" property="id" type="long"/>
</primary-key>

<column name="MESSAGE_TEXT" property="text"/> (5)
```

```
        <foreign-key constraint-name="FK_NEXT_MESSAGE">
            <many-to-one property="nextMessage"/>
            <set exclude="true"/>
        </foreign-key>
    </table>
</hibernate-reverse-engineering>
```

❶ 这个XML文件有它自己的DTD用于验证和自动完成。

❷ 表过滤器可以用一个正则表达式按名称把表排除。然而，在这个例子中，给所有匹配这个正则表达式的表生成的类定义了一个默认的包。

❸ 可以按名称定制单独的表。模式名称通常是可选的，但是默认时HSQLDB把PUBLIC模式分配给所有表，因此获取JDBC元数据时，识别表就需要这项设置。这里也可以给生成的实体设置一个定制的类名。

❹ 主键列生成一个具名id的属性，默认时是messageId。也显式地声明应该使用哪个Hibernate标识符生成器。

❺ 可以排除单独的列，或者，在这个例子中，可以指定生成属性的名称——默认时是messageText。

❻ 如果外键约束FK_NEXT_MESSAGE从JDBC元数据中获取，默认时就把一个多对一的关联创建到该类的目标实体。通过按名称匹配外键约束，可以指定是否应该生成一个反向集合（一对多）（本例排除了这一点），以及这个多对一的属性应该如何命名。

如果现在使用这个定制运行Ant目标，它就会在源目录的hello包中生成Message.hbm.xml文件。（首先要把Freemarker和jTidy JAR文件复制到你的库目录里面。）你所做的定制造成了Hibernate映射文件与你稍早手写的相同，如代码清单2-2所示。

除了XML映射文件之外，Ant目标还在源目录中生成一个Hibernate XML配置文件：

```
<hibernate-configuration>
  <session-factory>
    <property name="hibernate.connection.driver_class">
        org.hsqldb.jdbcDriver
    </property>
    <property name="hibernate.connection.url">
        jdbc:hsqldb:hsql://localhost
    </property>
    <property name="hibernate.connection.username">
        sa
    </property>
    <property name="hibernate.dialect">
        org.hibernate.dialect.HSQLDialect
    </property>

    <mapping resource="hello/Message.hbm.xml" />

  </session-factory>
</hibernate-configuration>
```

这个导出器把你用于反向工程的所有数据库连接设置都写进了这个文件，假设这就是当你运

行应用程序时要连接到的数据库。它也把所有生成的XML映射文件添加到了配置中。

下一步做什么呢？可以开始给Message的Java类编写源代码。或者让Hibernate Tools为你生成领域模型的类。

2.3.3 生成 Java 源代码

假设Message类有一个现成的Hibernate XML映射文件，并且想要给这个类生成源代码。如第3章所述，一个简单的Java实体类实现Serializable是很理想的，它有一个无参的构造函数（constructor），带有所有属性的获取方法和设置方法，并且有一个封装的实现。

可以使用在Ant构建中的Hibernate Tools和hbm2java 导出器生成用于实体类的源代码。这个源工件可以是任何能够被读进Hibernate元数据模型的东西——如果你想要定制Java代码生成，Hibernate XML映射文件是最好的。

把下列目标添加到Ant构建：

```
<target name="reveng.pojos"
        description="Produces Java classes from XML mappings">

    <hibernatetool destdir="${basedir}/src">

        <configuration>
            <fileset dir="${basedir}/src">
                <include name="**/*.hbm.xml"/>
            </fileset>
        </configuration>

        <hbm2java/> <!-- Generate entity class source -->

    </hibernatetool>

</target>
```

<configuration>读取所有的Hibernate XML映射文件，<hbm2java> 导出器通过默认的策略生成Java源代码。

1. 定制实体类生成

默认情况下，hbm2java给每个被映射的实体生成一个简单的实体类。这个类实现Serializable标识接口（marker interface），并具备所有属性的访问方法和必要的构造函数。类的所有属性对字段都有私有（private）可见性，尽管你可以在XML映射文件中用<meta>元素和属性改变这个行为。

对默认的反射工程行为所做的第一个改变是限制Message属性的可见性范围。默认时，所有访问方法生成时都包含公有（public）可见性。假设Message对象是不可变的；你不想在公共接口上公开设置方法，而只公开获取方法。不用<meta>元素增强每个属性的映射，而是在类级中声明一个元属性（meta-attribute），如此来把设置应用给那个类中的所有属性：

```
<class name="Message"
   table="MESSAGES">

       <meta attribute="scope-set">private</meta>
```

```
        ...
</class>
```

其中scope-set属性定义了属性设置方法的可见性。

hbm2java 导出器也在下一个更高的级别接受元属性，即在根<hibernate-mapping>元素中，然后它被应用到所有在XML文件中被映射的类。也可以把细粒度的元属性添加到单个属性、集合或者组件映射。

生成实体类的一个（虽然小）改进，是在生成的toString()方法的输出中包含Message的text。文本是应用程序的日志输出中一个很好的可视控件元素。可以改变Message的映射，来把它包括在生成的代码中：

```
<property name="text" type="string">
    <meta attribute="use-in-tostring">true</meta>
    <column name="MESSAGE_TEXT" />
</property>
```

Message.java中toString()方法的生成代码看起来像下面这样：

```
public String toString() {
    StringBuffer buffer = new StringBuffer();
    buffer.append(getClass().getName())
        .append("@")
        .append( Integer.toHexString(hashCode()) )
        .append(" [");
        .append("text").append("='").append(getText()).append("' ");
        .append("]");
    return buffer.toString();
}
```

元属性可以被继承；也就是说，如果你在一个<class>元素的级别中声明一个use-in-tostring，这个类的所有属性都被包括在toString()方法中。这种继承机制适用于所有hbm2java的元属性，但是你可以选择把它关闭：

```
<meta attribute="scope-class" inherit="false">public abstract</meta>
```

在scope-class的元属性中设置inherit（继承）为false，仅仅创建了这个<meta>元素的父类为public abstract，但没有创建任何（可能）嵌套的子类。

在编写本书之时，hbm2java 导出器支持17种元属性用于调优代码的生成。大部分与可见性、接口实现、类扩展和预定义的Javadoc注释有关。完整的列表请参阅Hibernate Tools文档。

如果你用的是JDK 5.0，可以通过在<hbm2java>任务中设置jdk5="true"，切换到自动生成的静态导入（static import）和泛型（generic）。或者可以生成包含注解的EJB 3.0实体类。

2．生成Java Persistence实体类

一般地，在实体类源代码中使用Hibernate XML映射文件或者JPA注解来定义映射元数据，因此从XML映射文件中生成包含注解的Java Persistence实体类不太合理。然而，可以直接从JDBC元数据中创建包含注解的实体类源代码，并跳过XML映射步骤。看看下列Ant目标：

```
<target name="reveng.entities"
        description="Produces Java entity classes in src directory">
```

```
        <hibernatetool destdir="${basedir}/src">

            <jdbcconfiguration
                propertyfile="${basedir}/helloworld.db.properties"
                revengfile="${basedir}/helloworld.reveng.xml"/>

            <hbm2java jdk5="true" ejb3="true"/>
            <hbm2cfgxml ejb3="true"/>

        </hibernatetool>

    </target>
```

这个目标生成包含映射注解的实体类源代码和列有这些被映射类的hibernate.cfg.xml文件。可以直接编辑Java源代码来定制映射，如果在helloworld.reveng.xml中定制太受限制的话。

还要注意所有的导出器都依赖于以FreeMarker模板语言编写的模板。可以用任何喜欢的方式定制模板，甚至可以编写自己的模板。甚至编程定制代码的生成也是可能的。Hibernate Tools参考文档说明了如何使用这些选项。

Hibernate Tools提供的其他导出器和配置如下：

- <annotationconfiguration>取代一般的<configuration>，如果你想要从被注解的Java类中读取映射元数据，而不是从XML映射文件中读取的话。它唯一的实参是包含一列已注解类的hibernate.cfg.xml文件的位置和名称。使用这种方法从被注解的类中导出一个数据库Schema。
- <ejb3configuration>相当于<annotationconfiguration>，除了它可以自动在classpath中扫描被注解的Java类之外；它不需要hibernate.cfg.xml文件。
- <hbm2dao>导出器可以基于数据访问对象（Data Access Object，DAO）模式，给持久层创建额外的Java源代码。在编写本书之时，这个导出器的模板已经很旧了，需要更新。我们希望完成后的新模板将类似于16.2节中所示的DAO代码。
- <hbm2doc>导出器生成给表和Java实体提供文档的HTML文件。
- <hbmtemplate>导出器可以用一组定制的FreeMarker模板参数化，通过这种方法可以生成你想要的任何东西。Hibernate Tools捆绑了利用JBoss Seam框架来生成完整的、可运行的骨架应用程序的模板。

可以利用这些工具的导入和导出功能来实现创造性。例如，可以用<annotationconfiguration>读取被注解的Java类，并用<hbm2hbmxml>将它们导出。这允许你用JDK 5.0和更方便的注解进行开发，而在产品中（在JDK 1.4中）则部署Hibernate XML映射文件。

来用一些更高级的配置选项来结束本章的讨论，并把Hibernate和Java EE服务整合起来。

2.4 与 Java EE 服务整合

假设你已经试过了本章前面介绍过的“Hello World”示例，并且熟悉基础的Hibernate配置，以及Hibernate如何与简单的Java应用程序整合。现在，我们要讨论更高级的原生的Hibernate配置选项，以及一般的Hibernate应用程序如何使用Java EE应用程序服务器提供的Java EE服务。

如果你用Hibernate Annotations和Hibernate EntityManager创建过第一个JPA项目，接下来的配置建议跟你就没有太大关系了——如果你使用JPA，就已经深入到Java EE，不需要另外的整合步骤了。因此，如果使用Hibernate EntityManager的话，可以跳过本节的内容。

Java EE应用程序服务器，例如JBoss AS、BEA WebLogic和IBM WebSphere，为Java实现标准的（特定于Java EE）的托管环境。Hibernate能够与之整合的三大最值得关注的Java EE服务是JTA、JNDI和JMX。

JTA允许Hibernate参与托管资源中的事务。Hibernate可以通过JNDI查找托管资源（数据库连接），并且能够把自身当作一项服务绑定到JNDI上。最后，Hibernate可以通过JMX被部署，然后通过JMX容器被当作一项服务来管理，并且使用标准的JMX客户端程序在运行时被监控。

来看看每一项服务，以及如何把Hibernate和它整合起来。

2.4.1　与 JTA 整合

JTA是Java企业应用中事务控制的标准服务接口。它公开了几个接口，例如用于事务划分的`UserTransaction` API和用于参与列事务生命周期中的`TransactionManager` API。事务管理器能够协调跨资源的单个事务——想象在单个事务的两个数据库中的两个Hibernate `Session`中进行。

JTA事务服务由所有Java EE应用程序服务器提供。然而，许多Java EE服务可以被独立使用，并且可以给应用部署JTA提供程序，例如JBoss Transaction或者ObjecWeb JOTM。我们对于配置的这部分不多说，而是关注Hibernate与JTA服务的整合，在完全的应用程序服务器中或者使用独立的JTA提供程序时也一样。

看看图2-6。使用Hibernate的`Session`接口访问（多个）数据库，Hibernate的责任是与托管环境的Java EE服务整合。

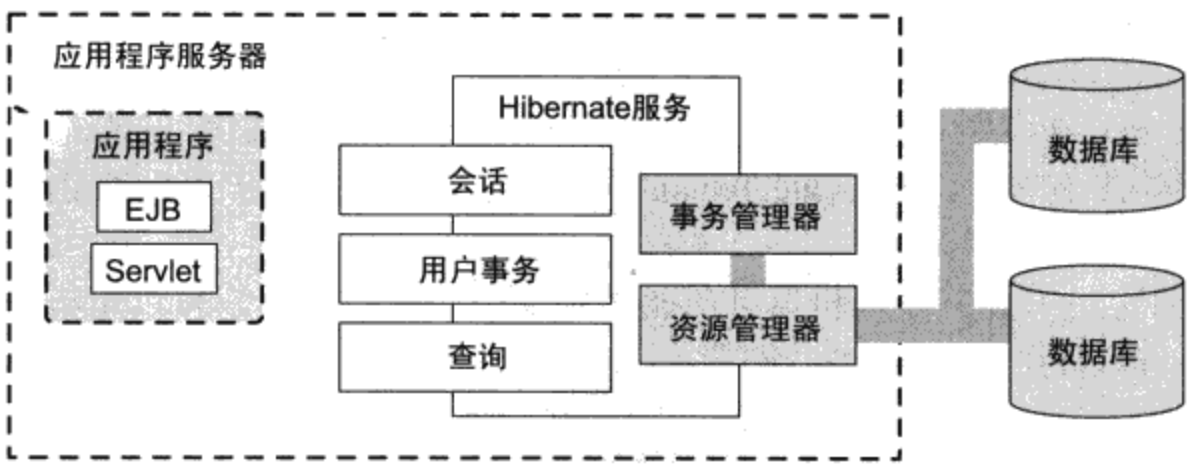


图2-6　包含托管资源的环境中的Hibernate

在这样一个托管环境中，Hibernate不再创建和维护JDBC连接池——Hibernate通过在JNDI注册中查找`Datasource`对象获得数据库连接。因此，你的Hibernate配置需要一个对JNDI名称的引用，在这个引用中能够获得托管的连接。

```
<hibernate-configuration>
<session-factory>

    <property name="hibernate.connection.datasource">
        java:/MyDatasource
    </property>

    <property name="hibernate.dialect">
        org.hibernate.dialect.HSQLDialect
    </property>
    ...

</session-factory>
</hibernate-configuration>
```

利用这个配置文件，Hibernate使用名称java:/MyDatasource在JNDI中查找数据库连接。当你配置应用程序服务器和部署应用程序时，或者当你配置独立的JTA提供程序时，这就是你应该把托管的数据源绑定到上面的名称。注意Hibernate仍然需要一个方言设置来生成正确的SQL。

说明 Hibernate和Tomcat——Tomcat不是一个Java EE应用程序服务器；它只是一个servlet容器，虽然servlet容器具有一些通常只有应用程序服务器才有的特性。这些特性中有一种可以与Hibernate共用：Tomcat连接池。Tomcat内部使用DBCP连接池，但是把它当作一个JNDI数据源公开，就像真实的应用程序服务器一样。要配置Tomcat数据源，需要根据Tomcat JNDI/JDBC文档中的说明编辑server.xml。Hibernate可以被配置为通过设置hibernate.connection.datasource来使用这个数据源。记住，Tomcat不提供事务管理器，因此你仍然有个简单的JDBC事务语义，Hibernate可以用可选的Transaction API把它隐藏起来。另一种方法是，可以与你的Web应用程序一起部署一个可与JTA兼容的、独立的事务管理器，考虑用它获得标准的UserTransaction API。另一方面，一般的应用程序服务器（尤其当它像JBoss As一样模块化时）可能比在Tomcat加DBCP加JTA更易于配置，并且提供更好的服务。

要完全把JTA与Hibernate整合，需要告诉Hibernate更多有关事务管理器的信息。Hibernate必须钩入事务生命周期，例如管理它的高速缓存。首先，要告诉Hibernate你正在使用哪种事务管理器：

```
<hibernate-configuration>
<session-factory>

    <property name="hibernate.connection.datasource">
        java:/MyDatasource
    </property>
    <property name="hibernate.dialect">
        org.hibernate.dialect.HSQLDialect
    </property>

    <property name="hibernate.transaction.manager_lookup_class">
        org.hibernate.transaction.JBossTransactionManagerLookup
    </property>

    <property name="hibernate.transaction.factory_class">
        org.hibernate.transaction.JTATransactionFactory
```

```
        </property>
        ...
    </session-factory>
    </hibernate-configuration>
```

你要给应用程序服务器选择适当的查找类，就如在前面的代码中所做的那样——Hibernate给最普及的JTA提供程序和应用程序服务器绑定了类。最后，告诉Hibernate要在应用程序中使用JTA事务接口设置事务范围。JTATransactionFactory完成几件事：

- 它为JTA启用了正确的Session范围和传播，如果你决定用SessionFactory.getCurrentSession()方法代替手工打开和关闭每个Session的话。我们在11.1节中将更深入地讨论这一特性。
- 它告诉Hibernate你正准备在应用程序中调用JTA UserTransaction接口，来启动、提交或者回滚系统事务。
- 万一你不想使用标准的UserTransaction，它还把Hibernate Transaction API切换到JTA。如果你现在使用Hibernate API启动事务，它会检查是否有JTA事务正在进行中，如有可能，参与这个事务。如果没有JTA事务正在进行，就会启动一个新事务。如果你用Hibernate API提交或者回滚，它要么忽略这个调用（如果Hibernate参与了一个现有的事务），要么设置系统事务为提交或者回滚。如果你在一个支持JTA的环境中部署的话，不建议使用Hibernate Transaction API。然而，这个设置使现有的代码可以在托管和非托管环境之间移植，虽然它可能使用不同的事务行为。

还有其他内建的TransactionFactory选项，并且你可以通过实现这个接口自己编写。JDBCTransactionFactory在非托管环境中是默认的，并且你在本章没有JTA的简单的“Hello World”示例中自始至终都在使用着它。如果使用JTA和EJB，并且计划在托管的EJB组件上声明设置事务范围——换句话说，如果要在一个Java EE应用程序服务器中部署EJB应用，但不在应用程序代码中使用UserTransaction接口编程式地设置事务范围——就应该启用CMTTransactionFactory。

按偏好顺序排列，我们建议的配置选项如下：

- 如果应用程序必须在托管和非托管环境中运行，应该把事务整合和资源管理的任务转给部署程序。在应用程序代码中调用JTA的UserTransaction API，让应用程序的部署程序相应地配置应用程序服务器或者独立的JTA提供程序。在Hibernate配置中启用JTATransactionFactory，与JTA服务整合，并设置正确的查找类。
- 考虑使用EJB组件通过声明设置事务范围。数据访问代码并没有被绑定到任何事务API，CMTTransactionFactory在后台为你整合和处理Hibernate Session。这是最容易的解决方案——当然，现在部署程序有责任提供一个支持JTA和EJB组件和环境。
- 用Hibernate Transaction API编写代码，并通过设置JDBCTransactionFactory或者JTATransactionFactory，让Hibernate在不同的部署环境中切换。注意事务语义可能改变，并且事务的启动或者提交可能导致一个你无法预料的空操作（no-op）。当需要事务划分的可移植性时，这永远是最后一种选择。

常见问题 如何使用带有Hibernate的几个数据库？如果要使用几个数据库，就创建几个配置文件。每个数据库都分配它自己的SessionFactory，并且从不同的Configuration对象中构建几个SessionFactory实例。每个Session都从任何SessionFactory中被打开，在JNDI中查找托管数据源。现在协调这些资源就是事务和资源管理器的责任了——Hibernate只在这些数据库连接上执行SQL语句。事务范围用JTA进行编程式地设置，或者由包含EJB的容器和一个声明的集合来处理。

Hibernate不仅可以在JNDI中查找托管资源，还可以把自身绑定到JNDI。接下来我们看看这个特性。

2.4.2 JNDI 绑定的 SessionFactory

我们已经遇到了每个Hibernate新用户都必须处理的一个问题：SessionFactory应该如何存储，以及它在应用程序代码中应该如何访问？本章稍早提出过这个问题，通过编写一个在静态字段中存放SessionFactory的HibernateUtil类，并提供静态的getSessionFactory()方法。然而，如果在一个支持JNDI的环境中部署应用程序，Hibernate可以把SessionFactory绑定到JNDI，你需要时可以到那里查找。

说明 Java命名和目录接口API（JNDI）允许对象向（从）一个层次结构（目录树）中存储（获取）。JNDI实现注册（registry）模式。基础的对象（事务上下文、数据源等）、配置设置（环境设置、用户注册等），甚至应用对象（EJB引用、对象工厂等）都可以被绑定到JNDI。

如果hibernage.session_factory_name属性设置为JNDI节点的名称，Hibernate的SessionFactory就会自动把自身绑定到JNDI。如果运行时环境不提供默认的JNDI上下文（或者如果默认的JNDI实现不支持Referenceable的实例），就需要使用hibernate.jndi.url和hibernate.jndi.class属性指定一个JNDI的初始上下文。

这里有一个例子，说明利用Sun公司（免费的）基于文件系统（file-system-based）的JNDI实现fscontext.jar，把SessionFactory绑定到java:/hibernate/MySessionFactory的Hibernage配置：

```
hibernate.connection.datasource = java:/MyDatasource
    hibernate.transaction.factory_class = \
        org.hibernate.transaction.JTATransactionFactory
    hibernate.transaction.manager_lookup_class = \
        org.hibernate.transaction.JBossTransactionManagerLookup
    hibernate.dialect = org.hibernate.dialect.PostgreSQLDialect
    hibernate.session_factory_name = java:/hibernate/MySessionFactory
    hibernate.jndi.class = com.sun.jndi.fscontext.RefFSContextFactory
    hibernate.jndi.url = file:/auction/jndi
```

当然，也可以使用基于XML的配置文件。这个例子是不现实的，因为通过JNDI提供连接池

的大部分应用程序服务器，还包含可编写的默认上下文的一个JNDI实现。JBoss AS必然也有，因此可以跳过最后两个属性，只要给SessionFactory指定一个名称即可。

说明　JNDI和Tomcat——Tomcat提供一个只读的JNDI上下文，servlet容器启动后，它不可以从应用程序级代码中编写。Hibernate无法绑定到这个上下文：你必须使用一个完全的上下文实现（如Sun FS上下文），或者通过在配置中省略session_factory_name属性，禁用SessionFactory的JNDI绑定。

SessionFactory在构建时被绑定到JNDI，也就是调用Configuration.buildSessionFactory()的时候。为了保证应用程序代码的可移植，你可能想要实现这个构建以及HibernateUtil中的查找，并在你的数据访问代码中继续使用辅助类，如代码清单2-18所示。

代码清单2-18　用于SessionFactory的JNDI查找的HibernateUtil

```
public class HibernateUtil {  lookup of SessionFactory

    private static Context jndiContext;

    static {
        try {
            // Build it and bind it to JNDI
            new Configuration().buildSessionFactory();
            // Get a handle to the registry (reads jndi.properties)
            jndiContext = new InitialContext();

        } catch (Throwable ex) {
            throw new ExceptionInInitializerError(ex);
        }
    }

    public static SessionFactory getSessionFactory(String sfName) {
        SessionFactory sf;
        try {
            sf = (SessionFactory) jndiContext.lookup(sfName);
        } catch (NamingException ex) {
            throw new RuntimeException(ex);
        }
        return sf;
    }
}
```

另一种方法是，可以用一个JNDI调用直接在应用程序代码中查找SessionFactory。然而，在应用程序的某个地方，仍然至少需要启动代码的new Configuration().buildSessionFactory()行。去除Hibernate启动代码的这最后一行，并完全消除HibernateUtil类的一种方法是，把Hibernate部署为一项JMX服务（或者通过使用JPA和Java EE）。

2.4.3 JMX服务部署

Java的世界充满了这些规范、标准和实现。一个相对较新但是很重要的标准是它的第一版：Java管理扩展（Java Management Extension，JMX）。JMX是关于系统组件管理或者系统服务管理的。

Hibernate哪些东西符合这个新标准呢？当Hibernate被部署在一个应用程序服务器中时，它利用其他的服务，就像使用托管的事务和池化数据源（pooled datasource）一样。而且，利用Hibernate JMX整合，Hibernate可以作为一项托管的JMX服务，被其他的服务依赖和使用。

JMX规范定义了下列组件：

- JMX MBean——可以重用（通常是构成基础）的一个组件，给管理公开接口。
- JMX容器（container）——调解对MBean的一般访问（本地的或者远程的）。
- JMX客户端程序（client）——可能通过JMX容器用来管理任何Mbean。

支持JMX的应用程序服务器（例如JBoss AS）表现得就像JMX容器一样，并允许MBean被配置和初始化成应用程序服务器启动过程的一部分。你的Hibernate服务可能被打包和部署为JMX MBean；给这个绑定的接口是`org.hibernate.jmx.HibernateService`。可以用任何标准的JMX客户端程序通过这个接口启动、停止和监测Hibernate核心。另一个可以选择进行部署的MBean接口是`org.hibernate.jmx.StatisticsService`，它让你用一个JMX客户端程序启用并监测Hibernate的运行时行为。

JMX服务和MBean如何部署是特定于供应商的。例如，在JBoss应用程序服务器中，你只需要添加jboss-service.xml文件到应用的EAR，把Hibernate部署为一个托管的JMX服务。

这里不对每一个选项进行解释，JBoss应用程序服务器的信息请见参考文档。它包含一个逐步介绍Hibernate整合和部署的小节（http://docs.jboss.org/jbossas）。在支持JMX的其他服务器上配置和部署应该与此类似，你可以改写和移植JBoss配置文件。

2.5 小结

在本章中，你已经完成了第一个Hibernate项目。我们讨论了如何编写Hibernate XML映射文件，以及可以在Hibernate中调用什么API来与数据库整合。

然后介绍了Java Persistence和EJB 3.0，并阐述了它如何简化最基础的Hibernate应用程序：通过自动的元数据扫描、标准的配置和打包，以及在托管EJB组件中的依赖注入。

如果必须从一个遗留数据库开始，可以使用Hibernate工具集从一个现有的模式中反向工程XML映射文件。或者，如果使用JDK 5.0和（或）EJB 3.0，可以直接从一个SQL数据库中生成Java应用代码。

最后，我们讨论了Java EE环境中更高级的Hibernate整合和配置选项——如果你用的是JPA或者EJB 3.0，它就已经替你完成了整合。

Hibernate功能和Java Persistence之间的高级概述和对比如表2-1所示。（在每一章的最后都可以找到一张类似的对照表。）

表2-1　Hibernate和JPA对照表

Hibernate Core	Java Persistence和EJB 3.0
与所有东西在任何地方整合。灵活，但有时配置复杂	适用于Java EE和Java SE。简单和标准的配置：在Java EE环境中无需额外的整合或者特殊的配置
配置需要一系列XML映射文件或者被注解的类	JPA提供程序自动扫描XML映射文件和被注解的类
私有但是强大。持续改善原生的编程接口和查询语言	标准和稳定的接口，包含一个充分的Hibernate功能的子集。有可能轻松地退回到Hibernate API

第3章将介绍一个更复杂的示例应用程序，本书后续部分将始终使用这个应用程序。你将学习如何设计和实现一个领域模型，以及哪些映射元数据选项在大项目中是最好的选择。

第3章

领域模型和元数据

本章内容

- CaveatEmptor示例应用程序
- 富领域模型的POJO设计
- ORM元数据选项

前一章的“Hello World”示例介绍了Hibernate；然而，它对理解包含复杂数据模型的现实应用程序的需求并没有帮助。本书后续内容将用更复杂的示例应用程序——CaveatEmptor，一个在线拍卖系统——来演示Hibernate和Java Persistence。

我们对应用程序的讨论从给持久化类引入编程模型开始。设计和实现持久化类是一个多步骤的过程，我们将详细阐述。

首先，将学习如何识别问题领域的业务实体。创建这些实体及其属性的一个概念模型，称为领域模型，通过创建持久化类在Java中实现它。我们花点时间探讨这些Java类看起来到底应该是什么样子，并看一下这些类的持久化能力，以及这个方面如何影响设计和实现。

然后探讨映射元数据选项——可以告诉Hibernate你的持久化类及其属性如何与数据库表和列联系起来的方法。这可能涉及编写XML文档，该文档最终与编译好的Java类一起部署，并在运行时被Hibernate读取。另一种选项基于EJB 3.0标准，直接在持久化类的Java源代码中使用JDK 5.0元数据注解。读完本章之后，你将知道如何在复杂的现实项目中设计领域模型的持久化部分，以及你主要喜欢和使用哪种映射元数据选项。

最后，在本章的最后一节（可能是可选的），看看Hibernate独立表示的能力。Hibernate中一项相对较新的特性允许你在Java中创建完全动态的领域模型，例如一个没有任何具体类而只有`HashMap`的模型。Hibernate也支持使用XML文档的领域模型表示法。

让我们从示例应用程序开始。

3.1 CaveatEmptor 应用程序

CaveatEmptor在线拍卖应用程序演示了ORM技术和Hibernate功能；可以从http://caveatemptor.hibernate.org中下载应用程序的源代码。本书将不太关注用户界面（它可能基于Web或者富客户端

程序）；而是集中在数据访问代码上。然而，当必须做出有关影响用户界面的数据访问代码的设计决策时，我们一般两者都考虑。

为了理解ORM中涉及的设计问题，先假设这个CaveatEmptor应用程序还不存在，你正从头开始构建。第一个任务就是分析。

3.1.1　分析业务领域

软件开发的工作从问题领域的分析开始（假设没有已经存在的遗留代码或者遗留数据库）。

在这个阶段，你在问题领域专家的帮助下，辨别与软件系统有关的主要实体。实体通常是为系统用户所理解的概念：付款、客户、订单、货品、出价等。有些实体可能是用户考虑的更不具体的事物的抽象概念，例如定价算法，但是即使这些通常对于用户也是可以理解的。所有这些实体都在业务的概念视图中，我们有时候称它为业务模型。开发人员和面向对象的软件架构师分析业务模型，并创建面向对象的模型，这些工作仍然处于概念阶段（没有Java代码）。这个模型可能像仅存在于开发人员心中的意象（mental image）一样简单，或者可能像通过计算机辅助软件工程（Computer-Aided Software Engineering，CASE）工具如ArgoUML或者TogetherJ创建的UML类图一样精细。用UML表达的一个简单模型如图3-1所示。

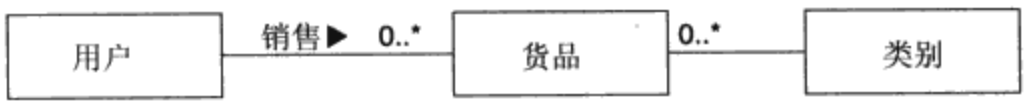


图3-1　典型在线拍卖模型的一个类图

这个模型包含了你一定要在任何典型的拍卖系统中找到的实体：类别、货品和用户。这些实体和它们的关系（可能还有它们的属性）都由问题领域的这个模型表示。我们称这种来自问题领域的面向对象的实体模型（仅包含用户关注的那些实体）为领域模型。它是真实世界的一个抽象视图。

领域模型分析和设计的目标是，为应用程序捕捉业务信息的本质。开发人员和架构师们可能不是从面向对象的模型开始，而是从一个数据模型开始应用程序设计（可能用一个实体关系图表达）。我们通常说，关于持久化，在这两者之间稍有不同；它们不过是不同的起点。最终，我们最关注业务实体的结构和关系，以及必须用来保证数据完整性的规则（例如，关系的多样性），以及用来操作数据的逻辑。

在对象模型中，有个关注点是多态的业务逻辑。对于我们以及自顶向下的开发方法而言，如果能够在多态的Java中实现逻辑模型会很有帮助；因此，初稿是一个面向对象的模型。然后衍生为逻辑关系数据模型（通常没有额外的图），并实现真正的物理数据库Schema。

来看看CaveatEmptor应用程序的问题领域的分析结果。

3.1.2　CaveatEmptor 领域模型

CaveatEmptor网站拍卖许多不同种类的货品，从电子设备到机票，应有尽有。拍卖根据英国的拍卖策略进行：用户连续在一件货品上出价，直到那件货品的出价期终止，最高的出价者胜出。

在任何商店中，货物都是按照类别分类，并把类似的货物集中到一个分区或者架子上。拍卖目录需要某种货品类别的层次，以便买家能够浏览这些类别或者按类别和货品属性任意搜索。货品清单显示在目录浏览器中，并搜索结果屏幕。从清单中选择一件货品，把买家带到一个货品细节的视图中。

一次拍卖由一连串的出价组成，但只有一个胜出。用户细节包括姓名、注册ID、地址、电子邮件地址和账单信息。

信任网（Web of Trust）是一个在线拍卖网站的基本特征。信任网允许用户对可信度（或者不可信度）建立信誉。买家可以对卖家进行评价（卖家也可以对买家进行评价），并且所有其他用户都可以看见评价。

领域模型的高级概览如图3-2所示。我们来简要地讨论这个模型中值得关注的一些特性。

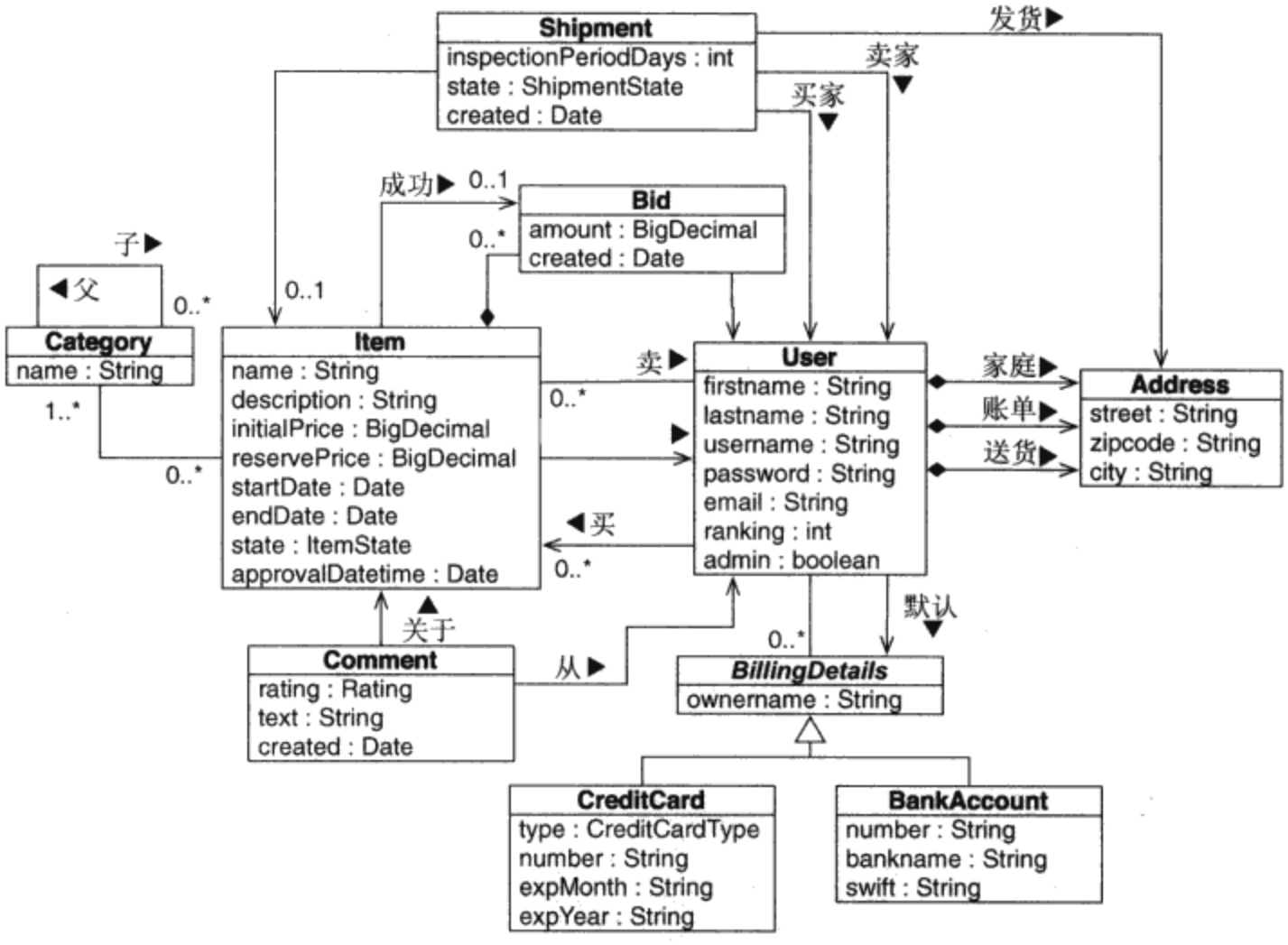


图3-2 CaveatEmptor领域模型及其关系的持久化类

每件货品只能被拍卖一次，因此你不需要把Item与任何拍卖实体区分开来。反之，你有单个拍卖货品实体命名为Item。这样，Bid直接与Item关联。Users只可以在一个拍卖上下文中写有关其他用户的Comments；因此Item和Comment之间有了关联。User的Address信息被当作一个单独的类建模，即使User可能只有一个Address；它们也可能有3个地址，分别为家庭住址、付账地址和收货地址。你一定要允许用户有多个BillingDetails。各种付款策略被表示为一个抽

象类的子类（允许未来进行扩展）。

Category可以被嵌套在另一个Category内部。这通过一个递归关联表达，从Category实体到它自身。注意单个Category可以有多个子类，但是最多只有一个父类。每个Item至少属于一个Category。

领域模型中的实体应该封装状态和行为。例如，User实体应该定义客户的姓名和地址，以及计算货品运输成本（对于这个特定的客户）所需的逻辑。领域模型是一个富对象模型，包含复杂的关联、交互和继承关系。关于使用领域模型的面向对象方法的一个有趣且详细的讨论，请参阅这两本书：*Patterns of Enterprise Application Architecture*（Fowler，2003）[①]或者*Domain-Driven Design*（Evans，2003）[②]。

本书将不过多讨论业务规则或者领域模型的行为。这不是因为我们认为它不重要，而是因为这个关注点与持久化问题最正交（orthogonal）。它是我们持久化实体的状态，因此我们把讨论的重点放在如何在领域模型中最好地表示状态，而不是关于如何表示行为上。例如，本书不关注售出的物品如何计算税，或者系统如何确认新的用户账号。而是更关注用户和他们所卖物品之间的关系如何表示，以及如何使这个关系持久化。后面的几章会在深入讨论分层应用程序设计和逻辑分离以及数据访问时，重新讨论这个问题。

说明　没有领域模型的ORM——我们强调具有完全ORM的对象持久化最适合基于富领域模型的应用程序。如果你的应用没有实现复杂的业务规则或者实体之间的复杂事务（或者如果你的实体不多），可能不需要领域模型。许多简单的和有些不那么简单的问题特别适合面向表的解决方案，其中的应用程序是围绕数据库数据模型，而不是围绕一个面向对象的领域模型而设计的，通常在数据库（存储过程）中执行逻辑。然而，领域模型越复杂、越富有表现力，你就越能从利用Hibernate中受益；在处理对象/关系持久化的全复杂性时，它更大放异彩。

既然有了一个（初步的）包含领域模型的应用程序，下一步就要在Java中实现它。来看一些需要考虑的东西。

3.2　实现领域模型

在Java中实现领域模型时，必须处理几个典型的问题。例如，如何把业务关注点与横切关注点（例如事务和持久化）分开？你需要自动的还是透明的持久化？必须使用一个特定的编程模型来实现这一点吗？本节将探讨这类问题，以及在典型的Hibernate应用程序中如何处理这些问题。

我们从任何实现都必须处理的一个问题开始：关注点的分离。领域模型实现通常是一个中心的组织组件；每当实现新的应用程序功能时，都要大量重用它。基于这个原因，你应该准备想尽办法确保除了业务方面的关注点之外，其余都不要渗透进领域模型实现。

① 中文版《企业应用架构模式》由机械工业出版社出版，影印版已由中国电力出版社出版。——编者注

② 中文版《领域驱动设计》已由清华大学出版社出版，英文影印版已由人民邮电出版社出版。——编者注

3.2.1 处理关注点渗漏

领域模型实现是如此重要的一块代码，它不应该依赖于正交的Java API。例如，领域模型中的代码不应该执行JNDI查找或者通过JDBC API调用数据库。这事实上允许你在任何地方重用领域模型实现。最重要的是，它使得对领域模型进行单元测试变得更加容易，而不需要特定的运行时环境或者容器（或者需要虚拟任何服务依赖）。这种分离强调了逻辑单元测试和集成单元测试之间的区别。

假设领域模型应该只注重给业务领域建模。但是还有其他的关注点，例如持久化、事务管理和授权。你不应该把处理这些横切关注点的代码放在实现领域模型的那些类中。当这些关注点开始出现在领域模型类中时，就是一个关注点渗透的例子。

EJB标准解决了渗透关注点的问题。如果用实体编程模型实现领域类，容器就会替你顾及一些关注点（或者至少让你把那些关注点具体化到元数据中，例如注解或者XML描述符）。EJB容器利用拦截（interception）防止某些横切关注点的渗透。EJB是一个托管的组件，在EJB容器内部执行；容器拦截对bean的调用，并执行它自己的功能。这种方法允许容器以一种普通的方式实现预设的横切关注点——安全性、并发性、持久化、事务和远程。

不幸的是，EJB 2.1规范在有关你必须如何实现领域模型方面强加了许多规则和约束。这本身就是一种关注点的渗透——本例中，容器实现者的关注点就已经渗透了！这一点在EJB 3.0规范中已经得到了解决，它是非侵入性的，并且更接近传统的JavaBean编程模型。

Hibernate不是一个应用程序服务器，它不会试图实现全部EJB规范的所有横切关注点。Hibernate是只针对持久化这个关注点的一种解决方案。如果你需要声明安全性和事务管理，应该通过会话bean访问实体实例，利用这些关注点的EJB容器实现。EJB容器中的Hibernate取代了（EJB 2.1，使用CMP的实体bean）或者实现了（EJB 3.0，Java Persistence实体）持久化方面。

Hibernate持久化类和EJB 3.0实体编程模型提供透明的持久化。Hibernate和Java Persistence还提供自动的持久化。

我们来更详细地探讨这两个术语，并找出一种准确的定义。

3.2.2 透明和自动持久化

我们使用透明（transparent）一词，意味着领域模型的持久化类和持久化逻辑之间一个完整的关注点分离，在这里持久化类不知道且不依赖于持久化机制。我们使用自动（automatic）一词，是指让你不用处理低级机械化细节的一种持久化解决方案，例如编写大部分SQL语句和利用JDBC API工作。

例如，`Item`类就没有任何代码依赖于任何Hibernate API。此外：

- Hibernate不需要任何特殊的超类或者接口被持久化类继承或者实现。也没有任何特殊的类被用来实现属性或者关联。（当然，你始终有选择使用这两种技术的权利。）透明的持久化改善了代码的可读性和维护性，就如你很快要见到的。
- 持久化类可以在持久化上下文之外被重用，例如在单元测试或者在用户界面（UI）层中。

可测试性是使用富领域模型应用程序的一个必要条件。

- 在一个包含透明持久化的系统中，对象不知道底层的数据仓库；它们甚至不需要知道它们正被持久化或者获取。持久化关注点被具体化到一个普通的持久化管理器接口——在Hibernate中，是Session和Query。在JPA中，EntityManager和Query（有相同的名称，但是有着不同的包，以及略微不同的API）起着同样的作用。

透明的持久化产生了一定程度的可移植性：不用特殊的接口，持久化类从任何特定的持久化解决方案中分离出来。我们的业务逻辑在任何其他的应用程序上下文中都是可以重用的。可以轻松地切换到另一种透明的持久化机制。由于JPA遵循同样的基础原则，Hibernate持久化类和JPA实体类之间没有区别。

根据透明持久化的这个定义，某些非自动的持久层也成了透明的（例如DAO模式），因为它们用抽象的编程接口分离与持久化相关的代码。只有没有依赖的简单Java类公开给业务逻辑或者包含业务逻辑。反之，有些自动的持久层（包括EJB 2.1实体实例和一些ORM解决方案）是非透明的，因此它们需要特殊的接口或者侵入的编程模型。

我们认为透明是必需的。透明的持久化应该成为任何ORM解决方案的主要目标。然而，没有任何自动的持久化解决方案是完全透明的：每个自动的持久层（包括Hibernate）都在持久化类上强加了一些要求。例如，Hibernate要求把集合值（collection-valued）属性定义为接口（如java.util.Set或者java.util.List），而不是定义为如java..util.HashSet这样的实际实现（不管怎么说，这是一种好实践）。或者，JPA实体类必须有一个特殊的属性，称作数据库标识符。

现在你知道为什么持久化机制应该尽可能不影响你如何实现领域模型了，并且知道透明和自动的持久化是必需的。应该使用哪种编程模型呢？要遵守哪些具体的要求和约定？真的需要特别的编程模型吗？理论上而言，不必；但是实际应用中，应该采用一种被Java社区广为接受的遵守规则的、一致的编程模型。

3.2.3　编写 POJO 和持久化实体类

作为对EJB 2.1实体实例的反应，许多开发人员开始谈论POJO（Plain Old Java Object，简单Java对象）[①]，这是一种本质上复兴JavaBean的基本（back-to-basics）的方法，是一种用于UI开发的组件模型，并把它重新应用给业务层。（现在大部分开发人员几乎把术语POJO和JavaBean同义地使用。）EJB规范的修订给我们带来了新的轻量级实体，称它们为“能够持久化的JavaBean”是很恰当的。Java开发人员不久将把所有这三个术语作为同义词用给同一种基础的设计方法。

在本书中，我们给任何能持久化实例的类实现使用持久化类；如果有些Java最佳实践是相关的，就用POJO；当Java实现遵循EJB 3.0和JPA规范时则用实体类。再次强调，你不应该太关注这些区别，因为最终的目标是尽可能透明地应用持久化方面。几乎每一个Java类都可以是持久化类，或者POJO，或者如果遵循一些好的实践的话，也可以是实体类。

① POJO有时候也写成Plain Ordinary Java Objects。这个术语于2002年由Martin Fowler、Rebecca Parsons和Josh Mackenzie提出。

Hibernate与用POJO实现的领域模型合作得最好。Hibernate强加给领域模型实现的少数必备条件，对于POJO实现也是最佳实践，因此大部分POJO不用任何改变就可以与Hibernate兼容。Hibernate的必备条件几乎与EJB 3.0实体类的相同，因此POJO实现可以轻松地用注解标识，并创造一个EJB 3.0兼容的实体。

POJO声明了定义行为的业务方法和表示状态的属性。有些属性表示与其他用户自定义的POJO的关联。

一个简单的POJO类如代码清单3-1所示。这是领域模型User实体的一个实现。

代码清单3-1 User类的POJO实现

```
public class User
        implements Serializable {          <-- Serializable的声明

    private String username;
    private Address address;

    public User() {}                       <-- 无参类构造函数

    public String getUsername() {
        return username;
    }

    public void setUsername(String username) {
        this.username = username;
    }

    public Address getAddress() {
        return address;
    }

    public void setAddress(Address address) {
        this.address = address;
    }

    public MonetaryAmount calcShippingCosts(Address fromLocation) {
        ...
    }                                      <-- 业务方法

}
```


Hibernate不要求持久化类实现Serializable（可序列化）。然而，当对象被存储在一个HttpSession中，或者用RMI按值传递时，就需要序列化。（这可能发生在Hibernate应用程序中。）类可以是抽象的，必要时，可以扩展非持久化的类。

不同于JavaBeans规范，它不需要特定的构造函数，而Hibernate（和JPA）则要求每个持久化类都有个无参构造函数。Hibernate在这个构造函数上使用Java Reflection API调用持久化类来实例化对象。构造函数可以是非公共的，但必须至少是包可见（package-visible）的，如果运行时生成的代理要用于性能优化的话。代理生成也要求这个类不作final声明（也没有final方法）！（13.1节将回到代理的主题。）

POJO的属性实现了业务实体的属性——例如，User的username。属性通常被实现为私有的或者受保护的实例变量，以及公共的属性访问方法：一种获取实例变量的值的方法，以及改变它

的值的方法。这些方法相应称作获取方法和设置方法。代码清单3-1中的示例POJO给username和address属性声明了获取方法和设置方法。

JavaBean规范定义了命名这些方法的指导方针，允许普通的工具（如Hibernate）轻松地发现和操作属性值。获取方法的名称以get开头，接着是属性名称（首字母大写）；设置方法的名称以set开头，并且类似地也跟着属性名称。用于Boolean属性的获取方法可以用is而不是get开头。

可以选择持久化类的实例状态应该如何被Hibernate持久化，通过直接访问它的字段，或者访问方法。你的类设计不受这些考虑因素的干扰。可以使一些访问方法为非公共的，或者把它们完全移除。有些获取方法和设置方法完成一些比访问实例变量更复杂的事（例如验证），但是琐碎的访问方法很普遍。它们的主要优势是在类的内部表示法和公共接口之间提供一个额外的缓冲，允许两者的独立重构。

代码清单3-1中的示例也定义了一种把一件货品运输到一个特定用户的成本计算的方法（我们省略了这种方法的实现）。

JPA实体类的必备条件是什么？好消息是，目前为止对POJO所讨论的所有约定，也都是JPA实体的必备条件。你必须应用一些额外的规则，但是它们同样简单；稍后会回到这些话题。

既然已经涵盖了用POJO持久化类作为编程模型的基础知识，现在来看看如何处理那些类之间的关联。

3.2.4　实现 POJO 关联

你用属性表达POJO类之间的关联，并用访问方法在运行时从一个对象到一个对象进行导航。考虑由Category类定义的那些关联，如图3-3所示。

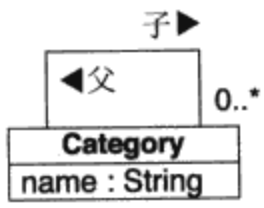


图3-3　包含关联的Category类图

和所有的图一样，上图省略了与关联相关的属性（我们称之为parentCategory和childCategories），因为它们会把图弄乱。操作它们值的这些属性和方法也称作脚手架代码（scaffolding code）。

这就是Category类一对多自关联的脚手架代码：

```
public class Category {
    private String name;
    private Category parentCategory;
    private Set childCategories = new HashSet();

    public Category() { }
    ...
}
```

为了允许关联的双向导航，你需要两个属性。parentCategory字段实现关联的单值端，并且被声明为Category类型。多值端，由childCategories字段实现，必须是集合类型。由于不允许重复，你选择一个Set（集），并把实例变量初始化为HashSet的一个新实例。

Hibernate需要用于集合类型属性的接口，因此必须使用java.util.Set或者java.util.List，而不是HashSet。这与实体中集合的JPA规范的要求相一致。运行时，Hibernate用它自己的一个类的实例把HashSet实例包起来。（这个特殊的类对于应用程序代码是不可见的。）不管怎样，这是集合接口编程的一个好实践，而不是具体的实现，因此这个限制不应该烦扰你。

现在你有了一些私有的实例变量，但是Hibernate没有公共的接口允许从业务代码或者属性管理中访问（如果它不应该直接访问字段的话）。来给这个类添加一些访问方法：

```
public String getName() {
    return name;
}

public void setName(String name) {
    this.name = name;
}

public Set getChildCategories() {
    return childCategories;
}

public void setChildCategories(Set childCategories) {
    this.childCategories = childCategories;
}

public Category getParentCategory() {
    return parentCategory;
}

public void setParentCategory(Category parentCategory) {
    this.parentCategory = parentCategory;
}
```

只有当这些访问方法是持久化类——被应用逻辑用来在两个对象之间创建一种关系——的外部接口的一部分时，它们需要再次被声明为public。然而，管理两个Category实例之间的链接，比在数据库字段中设置外键值更难。依据我们的经验，开发人员经常不知道从一个包含双向引用的网络对象模型中所产生的这种复杂性。我们将一步一步地探讨这个问题。

把一个子Category添加给一个父Category的基本过程看起来像这样：

```
Category aParent = new Category();
Category aChild = new Category();
aChild.setParentCategory(aParent);
aParent.getChildCategories().add(aChild);
```

每当在父Category和子Category之间创建一个链接时，需要两个动作：

- 必须设置子类的parentCategory，有效地破坏子类和它的旧父类之间的关联（任何子类都只能有一个父类）。
- 子类必须被添加到这个新的父Category的childCategories集合中。

说明　Hibernate中的托管关系——Hibernate不管理持久化关联。如果你要操作一个关联，必须编写与没有Hibernate时要编写完全相同的代码。如果关联是双向的，则关系的两侧都必须考虑。编程模型（如EJB 2.1实体bean）通过引入容器托管的关系扰乱了这个行为——如果一侧被应用程序修改，容器就会自动改变关系的另一侧。这就是为什么使用EJB 2.1实体bean的代码不能在容器之外被重用的原因之一。EJB 3.0实体关联是透明的，就像在Hibernate中一样。如果你不理解Hibernate中关联的这种行为，就问问你自己："没有Hibernate时我会怎么做？"Hibernate不会改变一般的Java语义。

给这些操作进行分组的Category类添加一种方便的方法是个好主意，这样允许重用，帮助确保正确性，并且最终保证数据的完整性：

```
public void addChildCategory(Category childCategory) {
     if (childCategory == null)
      throw new IllegalArgumentException("Null child category!");
     if (childCategory.getParentCategory() != null)
      childCategory.getParentCategory().getChildCategories()
                                        .remove(childCategory);
    childCategory.setParentCategory(this);
    childCategories.add(childCategory);
}
```

addChildCategory()方法不仅在处理Category对象时减少了代码行，而且加强了关联的基数性（cardinality）。它避免了漏掉两个必需动作中的其中一个时产生的错误。如果可能的话，应该始终对关联提供这种操作的分组。如果你把它与关系数据库中外键的关系模型相比较，就很容易明白网络和指针模型如何使一个简单的操作变得复杂：不用声明约束，而是需要过程性代码（procedural code）来保证数据的完整性。

由于想要addChildCategory()成为子类中唯一外部可见的存储器方法（mutator method）（可能还要加上removeChildCategory()方法），你可以让setChildCategories()方法为私有，或者删除它，并使用直接的字段访问进行持久化。获取方法仍然返回一个可以修改的集合，因此客户端可以用它进行在另一侧不受影响的变化。应该考虑静态的方法Collection.unmodifiable-Collection(c)和Collections.unmodifiableSet(s)，如果你更喜欢在返回它们之前把内部集合包在获取方法里面的话。然后，如果客户端试图修改集合，它就会获得一个异常；每次修改都被强制通过关系管理的方法。

在Category和Item类之间存在着一种不同的关系：一个双向的多对多关联，如图3-4所示。

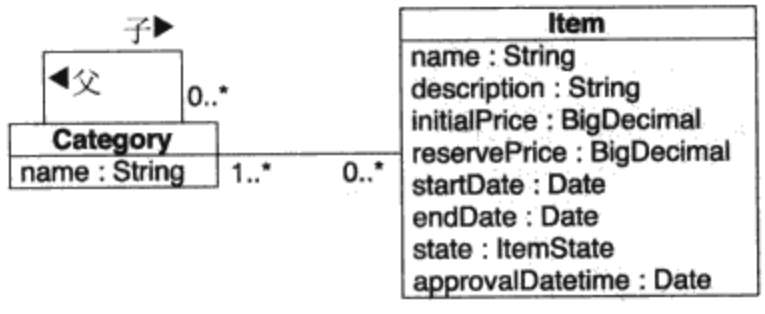


图3-4　Category和关联的Item类

对于多对多的关联，两侧都用集合值属性实现。添加访问Item关系的新属性和方法到Category类，如代码清单3-2所示。

代码清单3-2 Category到Item脚手架代码

```
public class Category {
    ...
    private Set items = new HashSet();
    ...

    public Set getItems() {
        return items;
    }

    public void setItems(Set items) {
        this.items = items;
    }
}
```

Item类的代码（多对多关联的另一侧）类似于Category类的代码。添加集合属性，标准的访问方法，以及简化关系管理的一种方法，如代码清单3-3所示。

代码清单3-3 Item到Category脚手架代码

```
public class Item {

    private String name;
    private String description;
    ...
    private Set categories = new HashSet();
    ...

    public Set getCategories() {
        return categories;
    }

    private void setCategories(Set categories) {
        this.categories = categories;
    }

    public void addCategory(Category category) {
        if (category == null)
            throw new IllegalArgumentException("Null category");
        category.getItems().add(this);
        categories.add(category);
    }
}
```

addCategory()方法类似于Category类的addChildCategory()便利方法。它被客户端用来操作Item和Category之间的链接。为了增加可读性，我们在后面的代码样例中将不显示便利方法了，并假设你会根据自己的偏好来添加它们。

给关联处理使用便利方法并不是改善领域模型实现的唯一途径。也可以把逻辑添加到你的访问方法中。

3.2.5　把逻辑添加到访问方法

我们喜欢使用JavaBean风格的访问方法的原因之一在于它们提供封装：一个属性的内部隐藏实现可以不做任何改变地变换为公共接口。这让你能够从数据库的设计中抽象类的内部数据结构——实例变量，如果Hibernate在运行时通过访问方法访问属性的话。它也促使公共API和类的内部表述的独立重构更加容易。

例如，如果数据库把用户名称保存为单个NAME列，但是User类有firstname和lastname属性，你可以把下列持久化的name属性添加到这个类：

```
public class User {
    private String firstname;
    private String lastname;
    ...

    public String getName() {
        return firstname + ' ' + lastname;
    }

    public void setName(String name) {
        StringTokenizer t = new StringTokenizer(name);
        firstname = t.nextToken();
        lastname = t.nextToken();
    }
    ....

}
```

稍后，你会明白Hibernate定制类型是处理多个这种情况的一种更好的方法。然而，有多种选择的话也自有好处。

访问方法也可以执行验证。例如，在下列例子中，setFirstName()方法验证大写的名称：

```
public class User {
    private String firstname;
    ...

    public String getFirstname() {
        return firstname;
    }

    public void setFirstname(String firstname)
        throws InvalidNameException {

        if ( !StringUtil.isCapitalizedName(firstname) )
            throw new InvalidNameException(firstname);
        this.firstname = firstname;
    }
    ....

}
```

当从数据库加载对象时，Hibernate可以使用访问方法来填充一个实例的状态，有时候你宁愿在Hibernate初始化新加载的对象时不发生这种验证。在这种情况下，告诉Hibernate直接访问实例变量就有意义了。

另一个要考虑的问题是脏检查。Hibernate自动侦测对象状态的改变，以便使更新过的状态与数据库同步。从获取方法返回一个不同的对象，通常比由Hibernate传递到设置方法的对象来得安全。Hibernate按值比较对象——不是按对象同一性——来确定一个属性的持久化状态是否需要被更新。例如，下列获取方法就不会导致不必要的SQL UPDATE（更新）：

```
public String getFirstname() {
    return new String(firstname);
}
```

这有个重要的例外：集合是按同一性比较的！对于一个被映射为持久化集合的属性，你应该从获取方法中返回与Hibernate传递到设置方法中完全相同的集合实例。如果没有，Hibernate将更新数据库，即使不需要更新，保存在内存中的状态每次也都会与数据库同步。在访问方法中通常应该避免这种代码：

```
public void setNames(List namesList) {
    names = (String[]) namesList.toArray();
}

public List getNames() {
    return Arrays.asList(names);
}
```

最后，必须知道访问方法中的异常如何处理，如果你在加载和存储实例时配置Hibernate来使用这些方法的话。如果抛出RuntimeException，当前的事务就被回滚，你要自己处理这个异常。如果抛出已检查应用异常，Hibernate就会把这个异常包在一个RuntimeException里面。

可以看到Hibernate没有用POJO编程模型对你进行不必要的限制。可以在访问方法中实现所需的任何逻辑（只要在获取方法和设置方法中保持相同的集合实例）。Hibernate如何访问属性是完全可以配置的。这种透明度保证了一个独立的和可重用的领域模型实现。并且目前为止我们所阐述的一切，对于Hibernate持久化类和JPA实体都是相当正确的。

现在给持久化类定义ORM。

3.3 ORM 元数据

ORM工具要求元数据指定类和表、属性和列、关联和外键、Java类型和SQL类型等之间的映射。这种信息称作ORM元数据。元数据是关于数据的数据，映射元数据定义和支配在面向对象和SQL系统中不同的类型系统和关系表示法之间的转化。

作为开发人员，编写和维护这个元数据是你的工作。本节将讨论各种方法，包括在XML文件和JDK 5.0源码注解中的元数据。通常在一个特定的项目中决定使用一种策略，读完这几节内容后，你将具有做出明智决定的背景信息。

3.3.1 XML 中的元数据

任何ORM解决方案都应该提供一个人类可读的、易于手工编辑的映射格式，而不只是一个GUI映射工具。目前，最受欢迎的对象/关系元数据格式是XML。以XML编写或者包含XML的映

射文档都是轻量级的、人类可读的、易于通过版本控制系统和文本编辑器操作的，并且它们可以在部署时被定制（或者甚至在运行时用编程式的XML生成）。

但是基于XML的元数据真的是最好的方法吗？在Java社区中可以看到相当一部分人还是反对过度使用XML的。每一个框架和应用程序服务器似乎都需要它自己的XML描述符。

在我们看来，这种反对有三大原因：

- 基于元数据的解决方案经常被不恰当地应用。元数据生来就没有简单的Java代码那么灵活或者易于维护。
- 许多现有的元数据格式不是被设计成易读和易于用手工编辑的。造成痛苦的一个主要原因尤其在于属性和元素值缺乏有意义的默认值，明显要求比应该需要的输入多得多。更糟糕的是，有些元数据模式只使用XML元素和文本值，没有任何属性。另一个问题是模式太过一般，其中的每个声明都被包在meta元素的普通的扩展属性中。
- 好的XML编辑器，特别是在IDE中，不像好的Java编码环境一样普遍。最糟糕且最容易确定的是，经常不提供文档类型声明（DTD），阻止了自动完成和验证。

在ORM中无法避免对元数据的需求。然而，Hibernate在设计时充分认识到了典型的元数据问题。Hibernate的XML元数据格式非常易读，并定义了有用的默认值。如果没有属性值，就在被映射的类上用反射来确定默认值。Hibernate还包含提供文档化的完整DTD。最终，IDE对XML的支持近来已经得到了改善，现代的IDE提供动态的XML验证，甚至提供一项自动完成的特性。

来看一下可以在Hibernate中使用XML元数据的方法。你在前一节中创建了`Category`类；现在要把它映射到数据库中的`CATEGORY`表。要这么做，得编写代码清单3-4中的XML映射文件。

代码清单3-4　Category类的Hibernate XML映射

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC                    ❶
    "-//Hibernate/Hibernate Mapping DTD//EN"
    "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">

<hibernate-mapping>                                   ❷
    <class                                            ❸
        name="auction.model.Category"
        table="CATEGORY">

        <id                                           ❹
            name="id"
            column="CATEGORY_ID"
            type="long">
            <generator class="native"/>
        </id>

        <property                                     ❺
            name="name"
            column="NAME"
            type="string"/>

    </class>
</hibernate-mapping>
```

❶ Hibernate映射DTD应该在每个映射文件中声明——XML的语法验证需要这么做。

❷ 映射在<hibernate-mapping>元素内部声明。你喜欢包括多少个类映射都可以，包括本书稍后会提到的其他特殊的声明。

❸ Category类（在auction.model包中）被映射到CATEGORY表。这张表中的每一行都表示类型Category的一个实例。

❹ 我们还没有讨论对象同一性的概念，因此你可能对这个映射元素感到好奇。这个复杂的话题将在下一章中讨论。要理解这个映射，了解CATEGORY表中的每一行都有一个与内存中实例的对象同一性相匹配的主键值就足够了。<id>映射元素被用来定义对象同一性的细节。

❺ 类型java.lang.String的属性名称被映射到一个数据库的NAME列。注意映射中声明的类型是一个内建的Hibernate类型（string），而不是Java属性的类型，或者SQL列类型。把它当作表示其他两个类型系统之间桥梁的一个转换器。

我们已经在这个例子中有意省略了集合和关联映射。关联尤其是集合映射更加复杂，因此本书的第二部分将再回到这个话题。

虽然可能通过使用多个<class>元素在一个映射文件中给多个类声明映射，但我们建议的实践（也是一些Hibernate工具所期待的实践）是给每个持久化类使用一个映射文件。习惯上给文件与被映射的类相同的名称，并附加一个后缀（例如Category.hbm.xml），并把它放在与Category类相同的包中。

就像已经提到过的，XML映射文件并不是在Hibernate应用程序中定义映射元数据的唯一方法。如果使用JDK 5.0，最好的选择是基于EJB 3.0和Java Persistence标准的Hibernate Annotations。

3.3.2 基于注解的元数据

基本思想是把元数据与它所描述的信息放在一起，而不是把它分离到一个不同的文件中去。Java在JDK 5.0之前并不具备这项功能，因此开发了一种可供选择的方案。XDoclet项目使用支持键/值对的特殊Javadoc标签，引入了包含元信息的Java源代码的注解。通过标签的嵌套，非常复杂的结构也得到了支持，但是只有一些IDE允许为自动完成和验证定制Javadoc模板。

Java规范请求（JSR）175在Java语言中引入了注解概念，给注解的定义使用类型安全和声明的接口。自动完成和编译时检查不再是问题。我们发现，与XDoclet相比，注解元数据并不冗长，并且它有更好的默认值。然而，JDK 5.0注解有时候比XDoclet注解更难于阅读，因为它们不在普通的注释块里面；你应该使用一个支持注解的可配置语法的突出显示的IDE。除此之外，我们在过去几年的日常工作中，并没有发现使用注解的严重缺点，并且我们认为注解-元数据支持将成为JDK 5.0最重要的特性之一。

现在我们要介绍映射注解并使用JDK 5.0。如果你必须使用JDK 1.4，但是又喜欢使用基于注解的元数据，那就考虑我们后面讨论的XDoclet。

1. 定义和使用注解

在注解第一个持久化类之前，我们来看看注解是如何创建的。自然地，你通常使用预设的注解。然而，知道如何扩展现有的元数据格式，或者知道如何编写你自己的注解是一项非常有用的

技能。下列代码示例显示了一个Entity注解的定义：

```
package javax.persistence;

@Target(TYPE)
@Retention(RUNTIME)
public @interface Entity {
    String name() default "";
}
```

第一行定义了包，像往常一样。这个注解在javax.persistence包中，它是EJB 3.0所定义的JPA。它是规范中最重要的注解之一——可以在POJO上应用它，使它成为一个持久化的实体类。接下来一行是一个注解，该注解把元信息添加到@Entity注解（有关元数据的元数据）。它指定@Entity注解只可以放在类型声明中；换句话说，只可以用@Entity注解标识类，而不标识字段或者方法。为这个注解所选择的保持策略是RUNTIME；其他方案（对于其他使用案例）包括编译期间移除注解元数据，或者只有字节码中包含的内容，没有可能的运行时反射。你甚至想要在运行时保存所有实体元信息，以便Hibernate可以在启动时通过Java Reflection读取。这个例子遵循的是注解的实际声明，包括它的接口名称和属性（在这个例子中只有一个属性name和一个空的字符串默认）。

使用这个注解来使POJO持久化类成为Java Persistence实体：

```
package auction.model;

import javax.persistence.*;

@Entity
@Table(name = "ITEM")
public class Item {
    ...
}
```

这个公共的类Item，已经被声明为持久化实体。它的所有属性现在都用默认策略自动持久化。还显示了另一个注解，它声明这个持久化类被映射到的那个数据库Schema中的表名。如果省略这个信息，JPA提供程序默认为未限定类名（unqualified class name），就像如果你在一个XML映射文件中省略表名时Hibernate所做的处理那样。

所有这些都是类型安全的，因此当Hibernate启动时，被声明的注解通过Java Reflection读取。你不需要编写任何XML映射文件，Hibernate无需解析任何XML，并且启动更快。IDE也可以轻松地验证和强调注解——毕竟它们是普通的Java类型。

注解明显的好处之一是它们对于敏捷开发的灵活性，如果你经常重构代码、重新命名、删除或者移动类和属性的话。大部分开发工具和编辑器都无法重构XML元素和属性值，但是注解是Java语言的一部分，包括在所有重构操作中。

应该应用哪些注解呢？有几个标准的且特定于供应商的包可供你选择。

2．考虑标准

基于注解的元数据对于你如何编写Java应用程序有着重大的影响。其他编程环境，比如C#和.NET，早就具备这种支持，开发人员很快就采用了元数据属性。在Java世界里，注解的广泛应

用是在Java EE 5.0中。所有被认为是Java EE一部分的规范，如EJB、JMS、JMX甚至servlet规范，都将被更新，并因为元数据需要而使用JDK 5.0注解。例如，J2EE 1.4中的Web服务通常需要XML文件中的重要元数据，因此我们期待看到通过注解真正提高生产力。或者，你可以通过在一个字段上添加注解，让Web容器把一个EJB句柄注入到servlet中。Sun公司引进了一项规范成果（JSR 250）来处理不同规范的注解，给整个Java平台定义一般的注解。然而，对于你在持久层上的工作，最重要的规范则是EJB 3.0和JPA。

一旦你已经在classpath中包括了JPA接口，来自Java Persistence包的注解在javax.persistence中就可以使用了。可以用这些注解声明持久化的实体类、可嵌入的类（下一章将讨论这些）、属性、字段、键等。JPA规范覆盖了基础的和最相关的高级映射——编写一个可移植的应用程序所需要的一切，包含一个可插拔的、标准的持久化层，在任何运行时容器的内部和外部都适用。

Java Persistence中没有指定哪些注解和映射特性？特定的JPA引擎和产品一般可以提供这些优势——所谓的供应商扩展。

3. 利用供应商扩展

即使你用javax.persistence包中JPA兼容的注解映射了应用程序的大部分模型，有时候还是必须使用供应商扩展。例如，你希望在高质量的持久化软件中可用的几乎所有的性能调优选项，例如抓取和高速缓存设置，都只作为特定于Hibernate的注解。

我们在一个例子中看看这会是什么样。再次注解Item实体源代码：

```
package auction.model;

import javax.persistence.*;

@Entity
@Table(name = "ITEM")
@org.hibernate.annotations.BatchSize(size = 10)
@org.hibernate.annotations.DiscriminatorFormula(
    "case when ITEM_IS_SPECIAL is not null then A else B end"
)
public class Item {
    ...
}
```

这个例子包含了两个Hibernate注解。第一个注解@BatchSize是一个抓取选项，可以在本书稍后要验证的几种情况下提升性能。第二个注解@DiscriminatorFormula是一个Hibernate映射注解，当类继承无法用简单的文字值（literal value）决定的时候（此处它映射了一个遗留的列ITEM_IS_SPECIAL——或许某种标记——到一个文字值），对于遗留的模式特别有用。这两个注解都用org.hibernate.annotations作为包名的前缀。把这当作一个好的实践，因为你现在可以很容易地看到这个实体的什么元数据是来自JPA规范，以及哪些标签是特定于供应商的。你还可以轻松地搜索源代码里的“org.hibernate.annotations”，并在单个搜索结果中得到你应用程序中所有非标准注解的一个全面概览。

如果转换Java Persistence的提供程序，你只需要替换特定于供应商的扩展，并且可以期待最复杂的解决方案中会有一个类似的特性设置。当然，我们希望你永远不要这么做，并且它在实际

应用中很少发生——只是备用着。

类中的注解仅仅涵盖适用于该特定类的元数据。然而，在更高的级别上经常需要元数据，用于整个包，甚至整个应用程序。在讨论这些方法之前，要介绍另一种映射元数据格式。

4．JPA和EJB 3.0中的XML描述符

EJB 3.0和Java Persistence标准都争先包含了注解。然而，专家组已经注意到了XML部署描述符在某些情况下的优势，特别对于随着每次部署而改变的配置元数据。结果，EJB 3.0和JPA中的每一个注解都可以用一个XML描述符元素替换。换句话说，如果你不想用，就不必使用注解（虽然我们强烈鼓励你重新考虑，并试试注解，如果这是你对注解的第一反应的话）。

来看一下特定持久化单元的JPA XML描述符示例：

```
<?xml version="1.0" encoding="UTF-8"?>

<entity-mappings
  xmlns="http://java.sun.com/xml/ns/persistence/orm"
  xmlns:xsi="http://www.w3.org/001/MLSchema-instance"
  xsi:schemaLocation=
      "http://java.sun.com/xml/ns/persistence/orm orm_1_0.xsd"
  version="1.0">

    <persistence-unit-metadata>
        <xml-mapping-metadata-complete/>
        <persistence-unit-defaults>
            <schema>MY_SCHEMA</schema>
            <catalog>MY_CATALOG</catalog>
            <cascade-persist/>
        </persistence-unit-defaults>
    </persistence-unit-metadata>
    <package>auction.model</package>

    <entity class="Item" access="PROPERTY"
            metadata-complete="true">
        <attributes>
            <id name="id">
                <generated-value strategy="AUTO"/>
            </id>
        </attributes>
    </entity>

</entity-mappings>
```

这个XML由JPA提供程序自动挑选，如果把它放在classpath中持久化单元的META-INF目录下的一个名为orm.xml的文件中的话。你会看到只需要给一个类命名一个标识符属性；就像在注解中一样，实体类的所有其他属性也通过一个明显的默认映射自动地被认为是持久化的。

也可以给整个持久化单元设置默认映射，例如Schema名称和默认的级联选项。如果包括<xml-mapping-metadata-complete>元素，JPA提供程序则完全忽略这个持久化单元中实体类的全部注解，并且仅仅依赖如同在orm.xml文件中定义的映射。你可以（在这个例子中是多余的）在一个实体级别上用metadata-complete="ture"启用它。一旦启用，JPA提供程序假设实体的所有属性都以XML格式映射，因此这个实体的所有注解都应该被忽略。

如果不想忽略而是要覆盖注解元数据，就先从orm.xml文件中移除全局的<xml-mapping-metadata-complete>元素。还要从任何应该覆盖（而不是取代）注解的实体映射中移除metadata-complete="true"属性：

```
<entity-mappings ...>

    <package>auction.model</package>

    <entity class="Item">
        <attributes>
            <basic name="initialPrice" optional="false">
                <column name="INIT_PRICE"/>
            </basic>
        </attributes>
    </entity>

</entity-mappings>
```

此处把initialPrice属性映射到了INIT_PRICE列，并指定它不可为空。Item类的initialPrice属性上的任何注解都被忽略，但Item类中的所有其他注解仍然得到应用。还要注意你并没有在这个映射中指定一种访问策略，因此字段或者访问方法访问根据Item中@Id注解的位置来使用。（第4章会讨论这个细节。）

在Java Persistence中使用XML部署描述符的一个明显的问题是，它们与原生的Hibernate XML映射文件的兼容性。这两种格式根本不兼容，你应该决定使用一种或者另一种。JPA XML描述符的语法比原生的Hibernate XML映射文件更接近于实际的JPA注解。

当决定一种XML元数据格式时，还要考虑供应商扩展。Hibernate XML格式支持所有可能的Hibernate映射，因此如果有些东西无法在JPA/Hibernate注解中被映射，则可以用原生的Hibernate XML文件映射。同样的事情对于JPA XML描述符则不是如此——它们只提供覆盖规范的方便且具体化的元数据。Sun公司不允许供应商扩展使用另外的命名空间。

另一方面，你无法用Hibernate XML映射文件覆盖注解；必须用XML格式定义一个完整的实体类映射。

基于这些原因，我们不介绍所有三种格式的所有可能的映射；我们把焦点放在原生的Hibernate XML元数据和JPA/Hibernate注解上。然而，如果你想学，将会学到有关JPA XML描述符的足够知识来使用它。

如果正使用JDK 5.0，就考虑JPA/Hibernate注解为首选。如果想把一个特定的类映射具体化，或者利用一个不能作为注解使用的Hibernate扩展，就回到原生的Hibernate XML映射文件。如果不打算使用任何供应商扩展（实际上，这是不可能的），或者如果只想覆盖几个注解，或者如果需要甚至包括部署描述符的完整可移植性，就考虑用JPA XML描述符。

但是如果你习惯使用JDK 1.4（或者甚至1.3），却又仍然想要享受行内元数据的更好重构能力和减少代码行的好处时，该怎么办？

3.3.3 使用 XDoclet

XDoclet项目给Java带来了面向属性的编程概念。XDoclet利用Javadoc标签格式（@attribute）

来指定类、字段，或者方法级的元数据属性。Manning出版社还有一本关于XDoclet的书*XDoclet in Action*（Walls和Richards，2004）。

XDoclet被实现为生成Hibernate XML元数据（或者其他东西，取决于插件）的一个Ant目标，作为构建过程的一部分。用XDoclet创建Hibernate XML映射文档很简单；不用手工编写，而是用定制的Javadoc标签给持久化类的Java源代码做上标记，如代码清单3-5所示。

代码清单3-5　用XDoclet标签给包含映射元数据的Java类做标记

```
/**
 * The Category class of the CaveatEmptor auction site domain model.
 *
 * @hibernate.class
 *  table="CATEGORY"
 */
public class Category {
    ...

    /**
     * @hibernate.id
     *  generator-class="native"
     *  column="CATEGORY_ID"
     */
    public Long getId() {
        return id;
    }

    ...

    /**
     * @hibernate.property
     */
    public String getName() {
        return name;
    }

    ...
}
```

当被注解的类到位，并且准备好了Ant任务时，你就可以自动生成和前一节（见代码清单3-4）所示一样的XML文档了。

XDoclet的缺点在于需要另一个构建步骤。大部分大Java项目已经使用Ant，因此这通常不成问题。有争议的是，XDoclet映射在部署时更不易配置；但是没有东西阻止你在部署之前用手工编辑生成的XML，因此这可能不是重要的拒绝理由。最终，对XDoclet标签验证的支持在你的开发环境中可能不可用。然而，最新的IDE至少支持标签名称的自动完成。本书不涵盖XDoclet的内容，但你可以在Hibernate网站上找到示例。

无论使用XML文件、JDK 5.0注解，还是XDoclet，都将经常发现你必须在几个地方重复元数据。换句话说，你需要添加不只一个属性、不只一个持久化类、或者甚至整个应用程序都可以应用的全局信息。

3.3.4 处理全局的元数据

考虑以下情形：你所有的领域模型持久化类都在同一个包中。然而，你必须在每个XML映射文件中指定完全限定（fully qualified）的类名，包括包名。声明包名一次，然后仅使用短的持久化类名，这样会容易得多。或者，不要通过access="field"映射属性给每个单独的属性启用直接的字段访问，而使用单个的转换为所有属性启用字段访问。类或者包范围的元数据会更方便。

有些元数据对整个应用程序有效。例如，查询字符串可以被具体化到元数据，并且在应用程序代码中通过一个全局唯一的名称调用。类似地，查询通常与特定的类无关，有时，甚至与特定的包无关。其他应用程序范围的元数据包括用户自定义的映射类型（转换器）和数据过滤器（动态视图）定义。

来快速看一些在Hibernate XML映射和JDK 5.0注解中全局元数据的例子。

1. 全局的XML映射元数据

如果检查XML映射DTD，会看到<hibernate-mapping>根元素有全局的选项，被应用到它内部的（多个）类映射——这些选项中有一些如下列例子所示：

```
<hibernate-mapping
    schema="AUCTION"
    default-lazy="false"
    default-access="field"
    auto-import="false">

<class ...>
    ...
</class>

</hibernate-mapping>
```

schema属性使得一个数据库Schema前缀AUCTION能够被Hibernate用给已映射类所生成的所有SQL声明。通过设置default-lazy为false，给有些类关联启用默认的外联结抓取，在13.1节中有这个主题。（这个default-lazy="true"切换有一个值得关注的副作用：它切换到了Hibernate 2.x默认的抓取行为——如果你移到Hibernate 3.x但不想更新所有抓取设置时可以使用它。）利用default-access，通过Hibernate为这个文件中所有被映射的类的所有持久化属性启用直接的字段访问。最后，auto-import设置对这个文件中的所有类关闭。4.3节将讨论实体的引入和命名。

提示 没有类声明的映射文件——任何复杂的应用都需要和呈现全局的元数据。例如，可以轻松地导入许多接口，或者具体化几百个查询字符串。在大型的应用程序中，通常创建没有实际类映射的映射文件，并且只有导入、外部查询，或者全局过滤器和类型定义。如果看看DTD，就会看到<class>映射在<hibernate-mapping>根元素内部是可选的。把全局的元数据分开，并把它组织到单独的文件中，例如AutionTypes.hbm.xml、AuctionQueries.hbm.xml等，并在Hibernate配置中加载它们，就像一般的映射文件一样。然而，确保在把所有定制的类型和过滤器应用到类映射的任何其他映射元数据之前，就已经加载了这些类型和过滤器。

来看看包含JDK 5.0注解的全局的元数据。

2. 全局的注解元数据

注解天生就被织入一个特定类的Java源代码中。虽然可能把全局的注解放在类的源文件中（在顶部），我们还是喜欢把全局的元数据放在一个单独的文件中。这称作包元数据（package metadata），它被特定包目录中具名package-info.java的文件所启用：

```
@org.hibernate.annotations.TypeDefs({
    @org.hibernate.annotations.TypeDef(
        name="monetary_amount_usd",
        typeClass = MonetaryAmountType.class,
        parameters = { @Parameter(name="convertTo", value="USD") }
    ),
    @org.hibernate.annotations.TypeDef(
        name="monetary_amount_eur",
        typeClass = MonetaryAmountType.class,
        parameters = { @Parameter(name="convertTo", value="EUR") }
    )
})
@org.hibernate.annotations.NamedQueries({
    @org.hibernate.annotations.NamedQuery(
        name = "findItemsOrderByPrice",
        query = "select i from Item i order by i.initialPrice)"
    )
})
package auction.persistence.types;
```

包元数据文件的这个例子，在包auction.persistence.types中，它声明了两个Hibernate类型转换器。5.2节将讨论Hibernate类型系统。现在你可以通过它们的名称在类映射中引用用户自定义的类型。可以用同样的机制来具体化查询和定义全局的标识符生成器（最后这个例子中没有显示）。

这就是前一个代码示例中只包括来自Hibernate包的注解，而没有Java Persistence注解的原因之一。JPA规范（在最后一刻）所做的一个改变是移除了JPA注解的包可见性。结果，没有任何Java Persistence注解可以被放在package-info.java文件中。如果需要可移植的全局的Java Persistence元数据，就把它放在orm.xml文件中。

注意，如果没有使用自动侦测，就必须在Hibernate或者JPA持久化单元配置中指定一个包含元数据文件的包——详情请见2.2.1节。

全局的注解（Hibernate和JPA）也可以被放在特定类的源代码中，就在import部分之后。注解的语法与package-info.java文件中的相同，因此这里不再重复。

现在你知道了如何编写本地的和全局的映射元数据。大型应用程序中的另一个问题是元数据的可移植性。

3. 使用占位符

在任何较大的Hibernate应用程序中，都要在映射元数据中面对本地代码的问题——有效地把映射绑定到一个特定数据库产品的代码。例如，公式、约束或者过滤器映射中的SQL语句，都不

被Hibernate解析，而是直接传递到数据库管理系统。这样的好处是灵活性——可以调用任何原生的SQL函数或者数据库系统支持的关键字。把原生的SQL放在映射元数据中的缺点是损失数据库的可移植性，因为映射及你的应用程序将只适用于特定的DBMS（或者甚至DBMS版本）。

甚至简单的东西，例如主键生成策略，通常不能跨越所有的数据库系统进行移植。第4章将讨论一种特殊的标识符生成器，叫做native，这是个内建的巧妙的主键生成器。在Oracle中，它用一个数据库序列在表中给行生成主键值；在IBM DB2中，它默认使用一个特殊的同一性主键列。以下是如何用XML映射它的方法：

```
<class name="Category" table="CATEGORY">

    <id name="id" column="CATEGORY_ID" type="long">
        <generator class="native"/>
    </id>

    ...
</class>
```

稍后将讨论这个映射的细节。值得关注的部分是声明class="native"作为标识符生成器。假设这个生成器提供的可移植性不是你所要的，可能因为你用了一个定制的标识符生成器，你编写的一个类实现了Hibernate的IdentifierGenerator接口：

```
<id name="id" column="CATEGORY_ID" type="long">
    <generator class="auction.custom.MyOracleGenerator"/>
</id>
```

XML映射文件现在被绑定到了一个特定的数据库产品，你损失了Hibernate应用程序的数据库可移植性。处理这个问题的一种方法是在你的XML文件中使用一个占位符，在映射文件被复制到目标目录下的构建期间，这个XML文件被替换（Ant支持这个）。只有当你有使用Ant的经验，或者应用程序的其他部分已经需要构建时的替换时，才建议使用这种机制。

一种更优雅的变更方法是使用定制的XML实体（与应用的业务实体无关）。假设你要在XML文件中具体化一个元素或者属性值，来使它保持可移植：

```
<id name="id" column="CATEGORY_ID" type="long">
    <generator class="&idgenerator;"/>
</id>
```

&idgenerator;值称作实体占位符。可以在XML文件的顶部定义它的值为实体声明，作为文档类型定义的一部分：

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping SYSTEM
    "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd"
[
<!ENTITY idgenerator    "auction.custom.MyOracleGenerator">
]>
```

当映射文件被读取的时候，XML解析器现在将在Hibernate启动时替换占位符。

可以把这个更推进一步，在单个的文件中把这个新增物具体化到DTD，并在所有其他的映射文件中包括全局的选项：

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping SYSTEM
    "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd"
[
<!ENTITY % globals SYSTEM "classpath://persistence/globals.dtd">
%globals;
]>
```

这个例子显示了作为DTD一部分的一个外部文件所包含的内容。所用的语法，就像经常采用XML时一样，相当粗糙，但是每一行的用途应该很清楚。所有全局的设置都被添加到classpath中persistence包里的globals.dtd文件中：

```
<!ENTITY idgenerator    "auction.custom.MyOracleGenerator">
<!-- Add more options if needed... -->
```

要从Oracle切换到一个不同的数据库系统，只要部署一个不同的globals.dtd文件。

你经常不仅要替换一个XML元素或者属性值，还要在所有的文件中包括映射元数据的整个块，例如当许多类共享一些常用的属性，并且你无法在单个位置使用继承来捕捉它们的时候。利用XML实体替换，可以把一个XML片段具体化到一个单独的文件，并把它包括在其他XML文件中。

假设所有的持久化类都有一个dateModified属性。第一步是把这个映射放在它自己的文件中，比如DateModified.hbm.xml：

```
<property name="dateModified"
          column="DATE_MOD"
          type="timestamp"/>
```

这个文件不需要XML首部或者任何其他标签。现在为一个持久化类把它包括在映射文件中：

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping SYSTEM
    "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd"
[
<!ENTITY datemodified   SYSTEM "classpath://model/DateModified.hbm.xml">
]>

<hibernate-mapping>

<class name="Item" table="ITEM"
    <id ...>

    &datemodified;

    ...
</class>
```

DateModified.hbm.xml的内容将被&datemodified;占位符包括和替换。当然，这也适用于较大的XML片段。

当Hibernate启动和读取映射文件时，XML DTD必须由XML解析器解析。内建的Hibernate实体解析器在classpath中查找Hibernate-mapping-3.0.dtd；它应该在试图从因特网上查找之前先在hibernate3.jar文件中查找DTD，每当一个实体URL被加上 http://hibernate.sourceforge.net/前缀时这都会自动发生。Hibernate实体解析器也可以侦测classpath://前缀，然后在classpath中搜索资

源，部署时可以从中复制资源。必须重复这个常见的问题：如果在你的映射中有正确的DTD引用，并且在classpath中有正确的JAR时，Hibernate就永远不会在因特网上查找DTD。

目前为止我们已经阐述过的方法——XML、JDK 5.0注解和XDoclet属性——假设所有的映射信息在开发（或者部署）时都是已知的。然而，假设有些信息在应用程序启动前未知。能够在运行时通过编程操作映射元数据吗？

3.3.5 运行时操作元数据

在运行时浏览、操作或者构建新映射，对于应用程序来说有时候有用。XML API（如DOM、dom4j和JDOM）都允许XML文档的直接运行时操作，因此你能够在把XML文档传递给Configuration对象之前，在运行时对它进行创建或者操作。

另一方面，Hibernate也公开了一个配置时的元模型，它包含了在静态映射元数据中声明的所有信息。这个元模型直接的编程式操作有时候有用，尤其对于允许通过用户编写的代码进行扩展的应用程序。一种更强大的方法是映射元数据完全纲领性的和动态的定义，没有任何静态的映射。但这很怪异，应该为完全动态的应用程序的一个特定类或者应用程序构建工具包所保留。

下列代码将一个新的属性motto添加到了User类：

```
// Get the existing mapping for User from Configuration
PersistentClass userMapping =
    cfg.getClassMapping(User.class.getName());

// Define a new column for the USER table
Column column = new Column();
column.setName("MOTTO");
column.setNullable(false);
column.setUnique(true);
userMapping.getTable().addColumn(column);

// Wrap the column in a Value
SimpleValue value = new SimpleValue();
value.setTable( userMapping.getTable() );
value.setTypeName("string");
value.addColumn(column);

// Define a new property of the User class
Property prop = new Property();
prop.setValue(value);
prop.setName("motto");
prop.setNodeName(prop.getName());
userMapping.addProperty(prop);

// Build a new session factory, using the new mapping
SessionFactory sf = cfg.buildSessionFactory();
```

PersistentClass对象表示了单个持久化类的元模型，你从Configuration对象中获取它。Column、SimpleValue和Property是Hibernate元模型的所有类，可以在org.hibernate.mapping包中使用它们。

提示　记住，把一种属性添加到一个现有的持久化类映射，如此处所示，相当容易，但是给之前未被映射的类编程式地创建一个新的映射就比较麻烦了。

一旦创建了SessionFactory，它的映射就是不可变的。SessionFactory内部使用一种与配置时所用的不同的元模型。没有办法从SessionFactory或者Session中退回到原始的Configuration。（注意，你可以从Session中获得SessionFactory，如果希望访问一个全局设置的话。）然而，应用程序可以通过调用getClassMetadata()或者getCollectionMetadata()来读取SessionFactory的元模型。这里有个例子：

```
Item item = ...;
ClassMetadata meta = sessionFactory.getClassMetadata(Item.class);
String[] metaPropertyNames =
            meta.getPropertyNames();
Object[] propertyValues =
            meta.getPropertyValues(item, EntityMode.POJO);
```

这个代码片段为一个特定的实例获取Item类的持久化属性名称和那些属性的值。它帮助你编写一般的代码。例如，可以使用这个特性给UI组件贴标签，或者改进日志的输出。

虽然你已经在前几节中见过一些映射的构造，但目前为止我们尚未介绍任何更复杂的类和属性映射。你现在应该决定要在项目中使用哪种映射元数据选项，然后在下一章学习更多有关类和属性映射的内容。

或者，如果你已经是一个经验丰富的Hibernate用户，可以阅读最新的Hibernate版本，看看它允许你如何表示一个没有Java类的领域模型。

3.4　其他实体表示法

本书目前为止讨论的始终是基于Java类的领域模型实现——我们称它们为POJO、持久化类、JavaBean或者实体。基于包含一般属性、集合等的Java类的领域模型实现，是类型安全的。如果访问类的一个属性，IDE就提供基于模型的强类型的自动完成，且编译器检查你的源代码是否正确。然而，你在领域模型实现中对这种安全性花费了更多的时间——而时间就是金钱。

在接下来的几节中，我们介绍Hibernate处理不通过Java类实现的领域模型的能力。我们基本上用类型安全换取了其他的好处，因为天下没有免费的午餐，每当犯下一个错误时，运行时就会有更多的错误。在Hibernate中，可以给应用程序选择一种实体模式（entity mode），或者甚至混合多种实体模式得到一个的模型。甚至可以在单个Session中的实体模式之间进行切换。

Hibernate中有3种内建的实体模式：

- POJO——基于POJO、持久化类的一种领域模型实现。这是目前为止你所见到的，是默认的实体模式。
- MAP——不需要Java类；用HashMap在Java应用程序中表示实体。这个模式允许完全动态应用程序程序的快速原型。
- DOM4J——不需要Java类；实体被表示为XML元素，基于dom4j API。这种模式对于导出

或者导入数据，或者通过XSLT处理来渲染和转换数据时特别有用。

为什么可能要跳过下一节，稍后再回来呢？有两个原因：第一，包含POJO的静态领域模型实现是一个常见的案例，而动态的或者XML表示法可能不是你立即需要的特性。第二，我们正要展现一些映射、查询和其他可能你目前为止还没有见过的操作，即使利用默认的POJO实体模式也没有见过。然而，如果对使用Hibernate有足够的信心，就继续往下读。

我们从MAP模式开始，探讨Hibernate应用程序如何被完全动态地输入。

3.4.1 创建动态的应用程序

动态的领域模型是一个被动态地输入的模型。例如，不是用表示一件拍卖货品的Java类，而是在一个Java Map中使用许多值。拍卖货品的每个属性都由一个键（属性名称）及其值来表示。

1. 映射实体名称

首先，要通过命名业务实体启用这个策略。在一个Hibernate XML映射文件中，使用entity-name属性：

```
<hibernate-mapping>

<class entity-name="ItemEntity" table="ITEM_ENTITY">
    <id name="id" type="long" column="ITEM_ID">
        <generator class="native"/>
    </id>

    <property name="initialPrice"
              type="big_decimal"
              column="INIT_PRICE"/>

    <property name="description"
              type="string"
              column="DESCRIPTION"/>

    <many-to-one name="seller"
                 entity-name="UserEntity"
                 column="USER_ID"/>
</class>

<class entity-name="UserEntity" table="USER_ENTITY">
    <id name="id" type="long" column="USER_ID">
        <generator class="native"/>
    </id>

    <property name="username"
              type="string"
              column="USERNAME"/>

    <bag name="itemsForSale" inverse="true" cascade="all">
        <key column="USER_ID"/>
        <one-to-many entity-name="ItemEntity"/>
    </bag>

</class>

</hibernate-mapping>
```

在这个映射文件中有三个值得关注的东西要看。

第一，把几个类映射混合到一个里面了，这是之前我们不建议的做法。这一次，并没有真正映射Java类，而是映射实体的逻辑名称。你没有把Java源文件和同名的XML映射文件放在一起，因此可以用你喜欢的任何方式组织元数据。

第二，<class name="...">属性已经被<class entity-name="..."取代。为了清楚起见，也附加了...Entity给这些逻辑名称，并把它们与你之前用一般的POJO创建的其他非动态映射区别开来。

第三，所有的实体关联，例如<many-to-one>和<one-to-many>，现在也引用逻辑的实体名称。关联映射中的class属性现在是entity-name。这并非严格需要——Hibernate会知道你正在引用一个逻辑的实体名称，即使你使用class属性。然而，当你稍后混合几种表示法时它可以避免混淆。

来看看使用动态的实体看起来是什么样子。

2. 使用动态的映射

为了创建其中一个实体的实例，要在Java Map中设置所有的属性值：

```
Map user = new HashMap();
user.put("username", "johndoe");

Map item1 = new HashMap();
item1.put("description", "An item for auction");
item1.put("initialPrice", new BigDecimal(99));
item1.put("seller", user);
Map item2 = new HashMap();
item2.put("description", "Another item for auction");
item2.put("initialPrice", new BigDecimal(123));
item2.put("seller", user);

Collection itemsForSale = new ArrayList();
itemsForSale.add(item1);
itemsForSale.add(item2);
user.put("itemsForSale", itemsForSale);

session.save("UserEntity", user);
```

第一个映射是UserEntity，你设置username属性为一个键/值对。接下来的两个映射都是ItemEntity，这里你通过把user映射放进item1和item2映射里面，设置了到每件货品的seller的链接。你正有效地链接映射——这就是为什么这种表示法策略有时也被称作“带有映射的映射的表示法。”

一对多关联反面的集合用一个ArrayList初始化，因为你用包（bag）[①]语义映射了它（Java没有包实现，但是Collection接口有包语义）。最后，Session中的save()方法被给定了一个逻辑的实体名称，其中user映射作为一个输入参数。

Hibernate知道UserEntity引用被动态映射的实体，并知道它应该把这个输入当作一个必须

① 即允许重复元素的无序集合。——译者注

被相应保存的映射来处理。Hibernate也级联到itemsForSale集合中的所有元素；因此，所有货品的映射也都成了持久化的。一个UserEntity和两个ItemEntity被插入到它们各自的表中。

常见问题 可以在动态模式中映射一个Set（集）吗？基于set的集合不适用于动态的实体模式。在前一个代码示例中，想象itemsForSale是一个Set。Set在它的元素中检查重复元素，因此当你调用add(item1)和add(item2)时，这些对象中的equals()方法也被调用。然而，item1和item2是Java Map实例，一个映射的equals()实现基于映射的键组。因此，由于item1和item2是带有相同键的映射，因此当它们被添加给一个Set时，并没有明显的区别。只有当你在动态的实体模式中需要集合时才使用bag或者list。

Hibernate处理映射就像处理POJO实例一样。例如，使一个映射持久化触发了标识符分配；处于持久化状态的每个映射都有一个包含生成值的标识符属性集。此外，持久化的映射被自动检查在一个工作单元内部的任何修改。例如，要在一件货品上设置新的价格，可以加载它，然后让Hibernate完成所有的工作：

```
Long storedItemId = (Long) item1.get("id");

Session session = getSessionFactory().openSession();
session.beginTransaction();

Map loadedItemMap = (Map) session.load("ItemEntity", storedItemId);
loadedItemMap.put("initialPrice", new BigDecimal(100));

session.getTransaction().commit();
session.close();
```

具有类参数如load()的所有Session方法也以一种接受多个实体名称的重载变体出现。在加载了一件货品的映射之后，设置新的价格，并通过提交事务使得所做的修改持久化，默认情况下，它会触发Session的脏检查和清除。

也可以在HQL查询中引用实体名称：

```
List queriedItemMaps =
    session.createQuery("from ItemEntity where initialPrice >= :p")
            .setParameter("p", new BigDecimal(100))
            .list();
```

这个查询返回ItemEntity映射的集合。它们处于持久化状态。

把这更推进一步，把POJO模型与动态的Map混合。为什么要把领域模型的静态实现与动态的映射表示法混合起来，有两个原因：

- 你要默认使用基于POJO类的静态模型，但有时候要把数据轻松地表示为映射的映射。这在生成报表时，或者必须实现一个可以动态地表示不同实体的普通用户界面时，特别有用。
- 要把模型的单个POJO类映射到几张表，然后通过指定一个逻辑的实体名称在运行时选择表。

可以找到混合实体模式的使用案例，但这种案例非常罕见，因此我们要把精力集中在更常见的案例上。

因此，首先我们将混合静态的POJO模型，有时还要对有些实体启用动态的映射表示法。

3. 混合动态的和静态的实体模式

要启用混合的模型表示法，得编辑XML映射元数据，并声明POJO类名和逻辑的实体名称：

```
<hibernate-mapping>

<class name="model.ItemPojo"
       entity-name="ItemEntity"
       table="ITEM_ENTITY">
    ...
    <many-to-one name="seller"
                 entity-name="UserEntity"
                 column="USER_ID"/>

</class>

<class name="model.UserPojo"
       entity-name="UserEntity"
       table="USER_ENTITY">
    ...
    <bag name="itemsForSale" inverse="true" cascade="all">
        <key column="USER_ID"/>
        <one-to-many entity-name="ItemEntity"/>
    </bag>
</class>

</hibernate-mapping>
```

显然，你还需要两个类model.ItemPojo和model.UserPojo，它们实现了这些实体的属性。两个实体之间的多对一和一对多关联仍然基于逻辑名称。

从现在开始Hibernate将主要使用逻辑名称。例如，下列代码就不起作用：

```
UserPojo user = new UserPojo();
...
ItemPojo item1 = new ItemPojo();
...
ItemPojo item2 = new ItemPojo();
...
Collection itemsForSale = new ArrayList();
...

session.save(user);
```

前面的例子创建了几个对象，设置了它们的属性，并链接它们，然后通过把user实例传递给save()，试图通过级联保存对象。Hibernate检查这个对象的类型，试图查出它是什么实体，并且因为Hibernate现在专门依赖逻辑的实体名称，因此它无法给model.UserPojo找到映射。当使用一个混合的表示法映射时，你要告诉Hibernate逻辑名称：

```
...
session.save("UserEntity", user);
```

一旦你改了这一行，前一个代码示例就起作用了。接下来，考虑加载，以及查询返回什么。默认情况下，特定的SessionFactory处于POJO实体模式，因此下列操作返回model.ItemPojo的实例：

```
Long storedItemId = item1.getId();
ItemPojo loadedItemPojo =
    (ItemPojo) session.load("ItemEntity", storedItemId);

List queriedItemPojos =
    session.createQuery("from ItemEntity where initialPrice >= :p")
            .setParameter("p", new BigDecimal(100))
            .list();
```

可以全局地或者临时地切换到一个动态的映射表示法，但是实体模式的全局转换会带来严重的后果。要全局地转换，得把下列代码添加到Hibernate配置中；例如，在hibernate.cfg.xml中：

```
<property name="default_entity_mode">dynamic-map</property>
```

现在所有的Session操作都期待或者返回动态输入的映射！存储、加载和查询POJO实例的前一个代码实例不再有效；你要存储和加载映射。

你更有可能想要临时切换到另一种实体模式，因此假设你把SessionFactory置于默认的POJO模式中。要在一个特定的Session中切换到动态映射，可以在现有会话的上面打开新的临时Session。下列代码使用了这样的一个临时的Session来为一位现有的卖家保存一件新的拍卖货品：

```
Session dynamicSession = session.getSession(EntityMode.MAP);

Map seller = (Map) dynamicSession.load("UserEntity", user.getId() );

Map newItemMap = new HashMap();
newItemMap.put("description", "An item for auction");
newItemMap.put("initialPrice", new BigDecimal(99));
newItemMap.put("seller", seller);

dynamicSession.save("ItemEntity", newItemMap);

Long storedItemId = (Long) newItemMap.get("id");

Map loadedItemMap =
    (Map) dynamicSession.load("ItemEntity", storedItemId);

List queriedItemMaps =
    dynamicSession
        .createQuery("from ItemEntity where initialPrice >= :p")
         .setParameter("p", new BigDecimal(100))
         .list();
```

用getSession()打开的临时dynamicSession不需要被清除或关闭；它继承原始Session的上下文。你只用它在选择的表示法中加载、查询或者保存数据，即前一个例子中的EntityMode.Map。注意，你不能用POJO实例链接一个映射；seller引用的必须是一个HashMap，而不是UserPojo的一个实例。

我们说过，逻辑实体名称的另一个好的用例是把一个POJO映射到几张表，因此我们来看看这个。

4．映射一个类多次

想象你有一些表，它们有一些共同的列。例如，你可能有ITEM_AUCTION和ITEM_SALE表。通常映射每张表给一个实体持久化类，分别为ItemAuction和ItemSale。在实体名称的帮助下，你可以保存工作，并实现单个持久化类。

为了把两张表都映射给单个持久化类，要使用不同的实体名称（通常是不同的属性映射）：

```
<hibernate-mapping>

<class name="model.Item"
       entity-name="ItemAuction"
       table="ITEM_AUCTION">

   <id name="id" column="ITEM_AUCTION_ID">...</id>
   <property name="description" column="DESCRIPTION"/>
   <property name="initialPrice" column="INIT_PRICE"/>

</class>

<class name="model.Item"
       entity-name="ItemSale"
       table="ITEM_SALE">

   <id name="id" column="ITEM_SALE_ID">...</id>
   <property name="description" column="DESCRIPTION"/>
   <property name="salesPrice" column="SALES_PRICE"/>

</class>

</hibernate-mapping>
```

model.Item持久化类有着你映射过的所有属性：id、description、initialPrice和salesPrice。根据在运行时使用的实体名称，有些属性被认为是持久化的，其他则为瞬时的：

```
Item itemForAuction = new Item();
itemForAuction.setDescription("An item for auction");
itemForAuction.setInitialPrice( new BigDecimal(99) );
session.save("ItemAuction", itemForAuction);

Item itemForSale = new Item();
itemForSale.setDescription("An item for sale");
itemForSale.setSalesPrice( new BigDecimal(123) );
session.save("ItemSale", itemForSale);
```

由于逻辑实体名称，Hibernate知道它应该把数据插入到哪张表中。根据用于加载和查询实体的实体名称，Hiberante从适当的表中选择。

需要这项功能的场合很少，你可能和我们一样认为前一个使用案例并不好或者不普遍。

下一节介绍第三种内建的Hibernate实体模式，领域实体用XML文档表示。

3.4.2　表示 XML 中的数据

XML只是一种文本文件格式；它没有成为数据存储或者数据管理的媒介的继承能力。XML

数据模型是脆弱的，它的类型系统复杂且低能，它的数据完整性几乎完全程序化，并且它还引入了几十年前就淘汰了的分层数据结构。然而，XML格式的数据对于在Java中使用却很有吸引力；因为我们有一些好工具。例如，可以用XSLT转换XML数据，这是我们认为最佳的使用案例之一。

Hibernate没有内建的功能来存储XML格式的数据；它依赖于关系表示和SQL，这种策略的好处应该很清楚。另一方面，Hibernate能够以XML格式把数据加载和呈现给应用程序的开发人员。它允许你使用一组复杂的工具，而无需任何额外的转换步骤。

假设你在默认的POJO模式下工作，并且很快想要获得一些以XML表示的数据。用EntityMode.DOM4J打开一个临时的Session：

```
Session dom4jSession = session.getSession(EntityMode.DOM4J);

Element userXML =
    (Element) dom4jSession.load(User.class, storedUserId);
```

此处返回的是一个dom4j Element，可以用这个dom4j API来读取和操作它。例如，可以用下列片段把它很漂亮地打印到控制台：

```
try {
    OutputFormat format = OutputFormat.createPrettyPrint();
    XMLWriter writer = new XMLWriter( System.out, format);
    writer.write( userXML );
} catch (IOException ex) {
    throw new RuntimeException(ex);
}
```

如果假设你重用来自前面那些例子中的POJO类和数据，你会看到一个User实例和两个Item实例（为了清楚起见，我们不再命名它们为UserPojo和ItemPojo）：

```
<User>
  <id>1</id>
  <username>johndoe</username>
  <itemsForSale>
    <Item>
      <id>2</id>
      <initialPrice>99</initialPrice>
      <description>An item for auction</description>
      <seller>1</seller>
    </Item>
    <Item>
      <id>3</id>
      <initialPrice>123</initialPrice>
      <description>Another item for auction</description>
      <seller>1</seller>
    </Item>
  </itemsForSale>
</User>
```

Hibernate假设默认的XML元素名称——实体和属性名称。你也会看到集合元素被嵌入，和通过标识符（<seller>元素）解析循环引用。

可以通过把node属性添加到Hibernate映射元数据来改变这个默认的XML表示法：

```
<hibernate-mapping>

<class name="Item" table="ITEM_ENTITY" node="item">

    <id name="id" type="long" column="ITEM_ID" node="@id">
        <generator class="native"/>
    </id>

    <property name="initialPrice"
              type="big_decimal"
              column="INIT_PRICE"
              node="item-details/@initial-price"/>

    <property name="description"
              type="string"
              column="DESCRIPTION"
              node="item-details/@description"/>

    <many-to-one name="seller"
                 class="User"
                 column="USER_ID"
                 embed-xml="false"
                 node="@seller-id"/>

</class>

<class name="User" table="USERS" node="user">

    <id name="id" type="long" column="USER_ID" node="@id">
        <generator class="native"/>
    </id>

    <property name="username"
              type="string"
              column="USERNAME"
              node="@username"/>

    <bag name="itemsForSale" inverse="true" cascade="all"
         embed-xml="true" node="items-for-sale">
        <key column="USER_ID"/>
        <one-to-many class="Item"/>
    </bag>

</class>

</hibernate-mapping>
```

每一个node属性都定义了XML表示法：

- <class>映射中的node="name"属性，给该实体定义了XML元素的名称。
- 任何属性映射中的node="name"属性，指定属性内容应该表示为给定名称的XML元素文本。
- 任何属性映射中的node="@name"属性，指定属性内容应该表示为给定名称的XML属性值。
- 任何属性映射中的node="name/@attname"属性，指定属性内容应该在给定名称的一个子元素中表示为给定名称的XML属性值。

embed-xml选项被用于触发被关联实体数据的嵌入或者引用。更新过的映射产生下列你之前已经见过的相同数据的XML表示法：

```
<user id="1" username="johndoe">
  <items-for-sale>
    <item id="2" seller-id="1">
      <item-details initial-price="99"
          description="An item for auction"/>
    </item>
    <item id="3" seller-id="1">
      <item-details initial-price="123"
          description="Another item for auction"/>
    </item>
  </items-for-sale>
</user>
```

小心这个embed-xml选项——你可能很容易创建导致无限循环的循环引用！

最后，以XML表示的数据是事务的和持久化的，因此你可以修改被查询的XML元素，并让Hibernate负责更新底层的表：

```
Element itemXML =
    (Element) dom4jSession.get(Item.class, storedItemId);

itemXML.element("item-details")
        .attribute("initial-price")
         .setValue("100");
session.flush(); // Hibernate executes UPDATEs

Element userXML =
    (Element) dom4jSession.get(User.class, storedUserId);

Element newItem = DocumentHelper.createElement("item");
Element newItemDetails = newItem.addElement("item-details");
newItem.addAttribute("seller-id",
                     userXml.attribute("id").getValue() );
newItemDetails.addAttribute("initial-price", "123");
newItemDetails.addAttribute("description", "A third item");

dom4jSession.save(Item.class.getName(), newItem);

dom4jSession.flush(); // Hibernate executes INSERTs
```

对于如何处理Hiberante返回的XML，没有限制。可以用喜欢的任何方式显示、导出和转换它。更多信息请见dom4j文档。

最后，注意如果你喜欢，可以同时使用所有三种内建的实体模式。可以映射领域模型的一个静态的POJO实现，为一般的用户界面切换到动态的映射，并把数据导到XML里面。或者，可以编写一个没有任何领域类的应用，只有动态的映射和XML。可是，必须提醒你，软件行业中的原型经常意味着，客户以这个没有人想要抛弃的原型而告终——你愿意买一辆样板车吗？强烈建议你依赖静态的领域模型，如果要创建可维护的系统的话。

本书不再考虑动态的模型或者XML表示法。反之，我们把焦点放在静态的持久化类以及它

们的映射方式上。

3.5　小结

本章着重讨论了Java中富领域模型的设计和实现。

现在你了解了领域模型中的持久化类应该摆脱横切关注点（例如事务和安全性）。甚至与持久化相关的关注点也不应该渗透到领域模型实现中去。你也知道了如果要独立、轻松地执行和测试业务对象，透明的持久化是多么重要。

你已经学习了POJO、JPA实体编程模型的最佳实践和必要条件，以及它们有哪些概念与旧JavaBean规范的相同。我们深入探讨了持久化类的实现，以及如何最好地表示属性和关系。

为了给本书的后续部分做准备，并学习所有ORM选项，你要做一个有根据的决定：使用XML映射文件，还是JDK 5.0注解，或者可能是两者的结合。现在你准备好以这两种格式编写更复杂的映射了。

为了方便起见，表3-1概括了本章所讨论的Hibernate和Java Persistence有关概念之间的区别。

表3-1　第3章中Hibernate和JPA对照表

Hibernate Core	Java Persistence和EJB 3.0
持久化类需要一个无参构造函数，包含public或protected的可见性，如果使用基于代理的延迟加载的话	JPA规范强制所有的实体类都有一个无参构造函数，包含public或protected的可见性
持久化集合必须被输入到接口。Hibernate支持所有JDK接口	持久化集合必须被输入到接口。只有所有接口的一个子集（例如没有排序的集合）被认为是完全可移植的
持久化属性可以在运行时通过字段或者访问方法被访问，或者可以应用一种完全可以定制的策略	实体类的持久化属性通过字段或者访问方法被访问，但是如果需要完全的可移植性，则并非这两种都行
XML元数据格式支持所有可能的Hibernate映射选项	JPA注解涵盖了所有基础的和最高级的映射选项。怪异的映射和调优需要Hibernate Annotations
XML映射元数据可以被全局地定义，并且XML占位符被用来使元数据免于依赖	全局的元数据只有在标准的orm.xml元数据文件中声明了，才是完全可移植的

本书的后续部分将介绍所有可能的基础的和一些高级的映射技术，用于类、属性、继承、集合和关联。你将学习如何解决结构化的对象/关系不匹配。

Part 2 第二部分

映射概念和策略

这部分关于实际的ORM，从类和属性到表和列。第4章从普通的类和属性映射开始，阐述了如何映射细粒度的Java领域模型。第5章介绍如何映射更复杂的类继承层次结构，以及如何利用强大的定制映射类型系统来扩展Hibernate的功能。第6章和第7章通过许多复杂的示例介绍如何映射Java集合和类之间的关联。最后，如果需要在现有的应用程序中引入Hibernate，或者如果必须使用遗留的数据库模式和手写的SQL，就会发现第8章最值得关注。我们还在这一章讨论为模式生成定制的SQL DDL。

读完这部分，你将可以开始快速创建甚至是最复杂的映射，并且使用正确的策略。你将理解如何解决继承映射的问题，以及集合和关联如何映射。你将还能够调优和定制Hibernate，以便与任何现有的数据库模式或者应用程序整合。

本部分内容

- 第4章　映射持久化类
- 第5章　继承和定制类型
- 第6章　映射集合和实体关联
- 第7章　高级实体关联映射
- 第8章　遗留数据库和定制SQL

第4章 映射持久化类

本章内容

- 理解实体和值类型概念
- 用XML和注解映射类
- 细粒度的属性和组件映射

本章介绍基础的映射选项，阐述类和属性如何被映射到表和列。我们介绍和讨论如何处理数据库同一性和主键，以及各种其他元数据设置如何被用来定制Hibernate加载和存储对象的方式。所有映射示例都以Hibernate的原生XML格式完成，且并列使用JPA注解和XML描述符。我们还深入探讨细粒度领域模型的映射，以及如何映射属性和嵌入的组件。

首先要定义实体和值类型之间的本质区别，并阐述应该如何处理领域模型的ORM。

4.1 理解实体和值类型

实体是表述一级业务对象的持久化类型（术语“对象”在这里指它本来的含义）。换句话说，你在一个应用程序中必须处理的有些类和类型更为重要，它一般使得其他的类和类型变得比较不重要。你可能会认同这种说法：在CaveatEmptor中，`Item`是比`String`更重要的类。`User`可能比`Address`更重要。是什么使得有些东西变得重要呢？让我们换个角度来看这个问题。

4.1.1 细粒度的领域模型

Hibernate的一个主要目标是支持细粒度的领域模型，过去我们把它分离出来作为富领域模型最重要的必要条件。这是我们使用POJO的一个原因。粗略地说，细粒度意味着类比表更多。

例如，用户可能既有付款地址又有家庭住址。在数据库中，你可能有单个`USERS`表，包含列`BILLING_STREET`、`BILLING_CITY`和`BILLING_ZIPCODE`，以及`HOME_STREET`、`HOME_CITY`和`HOME_ZIPCODE`。（还记得我们在第1章讨论过的SQL类型的问题吗？）

在领域模型中，可以使用相同的方法，把两个地址表示为`User`类的6个字符串值属性。但是用`Address`类给它建立模型会更好，在这个模型中`User`有`billingAddress`和`homeAddress`属性，从而给一个表使用三个类。

这个领域模型实现了更高的凝聚力和更好的代码重用，并且它比使用固定类型系统的SQL系统更容易理解。过去，许多ORM解决方案都没有对这种映射提供特别好的支持。

Hibernate强调实现类型安全和行为的细粒度类的有效性。例如，许多人把电子邮件地址作为User的一个字符串值的属性建立模型。一种更成熟的方法是定义EmailAddress类，它添加更高级的语义和行为——它可以提供sendEmail()方法。

这个粒度问题把我们引向了ORM中极其重要的一个特征。在Java中，所有的类都是平等的——所有的对象都有它们自己的同一性和生命周期。

我们来看一个例子。

4.1.2 定义概念

两个人住在同一幢公寓里，他们都在CaveatEmptor注册了用户账号。每个账号一般都由User的一个实例表示，因此你有两个实体实例。在CaveatEmptor模型中，User类与Address类有一个homeAddress关联。两个User实例都在运行时引用同一个Address实例，还是每个User实例都引用它自己的Address？如果Address假定要支持共享的运行时引用，它就是个实体类型。如果不支持，它就可能是个值类型，因此通过一个自己的实体实例（它也提供同一性）依赖于单个引用。

我们提倡类比表多的设计：一行表示多个实例。由于数据库同一性由主键值实现，有些持久化对象将没有自己的同一性。实际上，持久化机制对有些类实现按值传递（pass-by-value）的语义！行中表示的其中一个对象有它自己的同一性，其他对象则依赖于它。在前一个例子中，包含地址信息的USERS表中的列依赖于用户的标识符，表的主键。Address的实例依赖于User的实例。

Hibernate产生了下列本质的特征：

- 实体类型的对象有它自己的数据库同一性（主键值）。对实体实例的对象引用被持久化为数据库中的引用（一个外键值）。实体有它自己的生命周期；它可以独立于任何其他的实体而存在。CaveatEmptor中的例子是User、Item和Category。
- 值类型的对象没有数据库同一性；它属于一个实体实例，并且它的持久化状态被嵌入到自身实体的表行中。值类型没有标识符或者标识符属性。值类型实例的寿命受它自己的实体实例的寿命限制。值类型不支持共享引用：如果两个用户住在同一幢公寓里，每个用户都有一个对他们自身homeAddress实例的引用。最明显的值类型是类，比如String和Integer，但是所有的JDK类都被当作值类型。用户自定义的类也可以被映射为值类型；例如，CaveatEmptor有Address和MonetaryAmount。

领域模型中实体和值类型的识别不是一项特别的任务，但是遵循一个确定的过程。

4.1.3 识别实体和值类型

你可能发现给UML类图添加固定的信息很有帮助，以便可以立即看到和区分实体和值类型。这个实践也迫使你对所有的类考虑这种区别，这是可选映射和执行良好的持久层的第一步。请见图4-1中的例子。

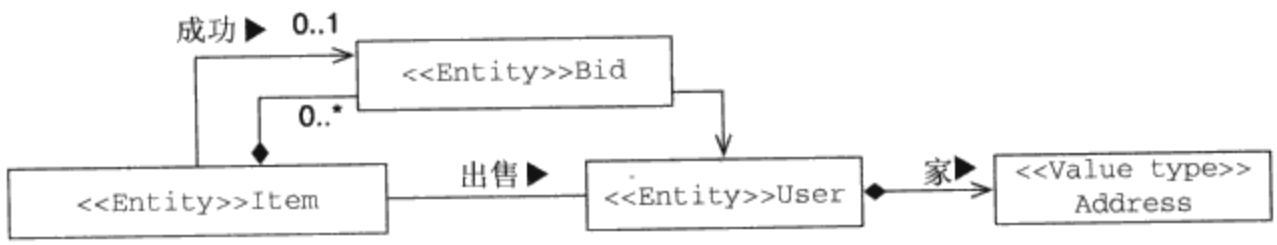


图4-1　已经把实体和值类型的固定信息添加到图中

Item和User类是明显的实体。它们各自都有自己的同一性，它们的实例有来自许多其他实例的引用（共享引用），并有独立的生命周期。

把Address视同一个值类型也很容易：一个特定的Address实例只被单个User实例引用。你知道，因为关联已经作为一个复合来创建，其中的User实例已经被设计成完全负责被引用Address实例的生命周期。因此，Address对象不能被任何其他的实例引用，也不需要它们自己的同一性。

Bid类是个问题。在面向对象的模型中，你表达一个组合（带有钻石符号的Item和Bid之间的关联），且Item管理所有Bid对象生命周期，并且Item有着对所有Bid对象的一个引用（一个引用集合）。这看起来似乎有道理，因为如果Item不再存在了，出价也就没用了。但是同时，还有另一个对Bid的关联：Item可能持有对它sucessfulBid的一个引用。这个成功的出价也必须是集合引用的其中一个出价，但是这在UML中没有表达出来。不管怎样，你必须处理对Bid实例可能的共享引用，因此Bid类必须是一个实体。它有一个依赖的生命周期，但必须有自己的同一性来支持共享引用。

你会经常发现这种混合行为；然而，你的第一反应应该是使所有的东西都成为值类型的类，并且只有当绝对必要时才把它提升为实体。试着简化关联：例如，集合有时候增加了无益的复杂性。不要映射Bid引用的一个持久化集合，而是可以编写一个查询，来获得Item的所有出价（第7章将再次回到这个话题）。

随着下一个步骤，把领域模型图拿走，并对所有实体和值类型实现POJO。你必须小心3件东西：

- *共享引用*——编写POJO类，以一种避免对值类型实例共享引用的方式。例如，确保Address对象只可以被一个User引用。例如，使它不可变，并用Address构造函数强制这种关系。
- *生命周期依赖*——如前所述，一个值类型实例的生命周期受它自己的实体实例限制。如果User对象被删除了，它的Address依赖对象也必须被删除。在Java中没有这种概念或者关键字，但是你的应用程序工作流和用户接口必须被设计成尊重和期待生命周期依赖。持久化元数据包括了所有依赖的级联规则。
- *同一性*——实体类几乎在任何情况下都需要标识符属性。用户自定义的值类型类（和JDK类）没有标识符属性，因为实例通过自己的实体被标识。

当本书稍后讨论到更高级的映射时，会回过头来讲类关联和生命周期规则。然而对象同一性是你目前必须理解的一个主题。

4.2 映射带有同一性的实体

在讨论术语（比如数据库同一性和Hibernate管理同一性的方式）之前，理解对象同一性和对象等同性之间的区别很重要。接下来探讨对象同一性和等同性如何与数据库（主键）的同一性相关联。

4.2.1 理解 Java 同一性和等同性

Java开发人员理解Java对象同一性和等同性之间的区别。对象同一性（==）是通过Java虚拟机定义的一个概念。如果两个对象引用指向相同的内存位置，它们就是同一的。

另一方面，对象等同性则是通过实现equals()方法的类定义的一个概念，有时候也被认为等值。等值意味着两个不同（非同一）的对象有着相同的值。如果String的两个不同实例表示顺序相同的字符，它们就是等值的，即使它们各自在虚拟机的内存中都有自己的位置。（如果你是Java专家，我们承认String是个特例。假设我们用了不同的类来说明同一个问题。）

持久化使事情变得复杂起来。利用对象/关系持久化，持久化对象是数据库表的一个特定行的内存表示。连同Java同一性（内存位置）和对象等同性一起，你还要选择数据库同一性（在持久化数据仓库中的位置）。现在你有3种识别对象的方法：

- 如果对象在JVM中占据着相同的内存位置，它们就是同一的。这可以通过使用 == 操作符进行检查。这个概念称作对象同一性。
- 如果对象有着相同的值，它们就是相等的，如equals(object o)方法定义的一样。不显式覆盖这个方法的类，继承了由java.lang.Object定义的实现，它比较对象同一性。这个概念称作等同性。
- 如果存储在一个关系数据库中的对象表示相同的行，或者它们共享相同的表和主键值，它们就是同一的。这个概念称作数据库同一性。

现在要看看数据库同一性如何与Hibernate中的对象同一性相关联，以及数据库同一性在映射元数据中如何表达。

4.2.2 处理数据库同一性

Hibernate以两种方式把数据库同一性公开给应用程序：

- 持久化实例的标识符属性值。
- Session.getIdentifier(Object entity)返回的值。

1. 给实体添加标识符属性

标识符属性很特殊——它的值是由持久化实例表示的数据库行的主键值。通常在领域模型图中不显示标识符属性。在示例中，标识符属性总是被命名为id。如果myCategory是Category的一个实例，调用myCategory.getId()就返回数据库中由myCategory表示的行的主键值。

来给Category类实现一个标识符属性：

```
public class Category {
    private Long id;
```

```
    ...
    public Long getId() {
        return this.id;
    }

    private void setId(Long id) {
        this.id = id;
    }
    ...
}
```

你应该使标识符属性的访问方法为私有（private）还是公有（public）范围呢？数据库标识符经常被应用程序当作对特定实例的一个方便的句柄使用，甚至在持久层之外。例如，把一个搜索屏幕的结果给用户显示为一个摘要信息的清单，这对于Web应用程序来说很常见。当用户选择一个特定的元素时，应用程序可能需要获取选中的对象，并且通常为此通过标识符来使用一个查找——你可能已经以这种方式使用过标识符，甚至在依赖JDBC的应用程序中。用一个公有标识符属性访问器来完全公开数据库同一性通常是适当的。

另一方面，通常声明`setId()`方法为私有，并让Hibernate生成和设置标识符值。或者，用直接的字段访问映射，并仅实现一个获取方法。（这个规则的例外是带有自然键的类，在这些类中，标识符的值在对象变成持久化之前由应用分配，而不是由Hibernate生成。我们将在第8章讨论自然键。）Hibernate不允许持久化实例的标识符值被第一次分配之后对它进行改变。主键值永远不变——否则该属性将不是个适当的主键备选对象！

标识符属性的Java类型，如前一个例子中的`java.lang.Long`，取决于`CATEGORY`表的主键类型，以及它在Hibernate元数据中如何被映射。

2. 映射标识符属性

用`<id>`元素在Hibernate XML文件中映射一个普通的（非复合的）标识符属性：

```
<class name="Category" table="CATEGORY">
<id name="id" column="CATEGORY_ID" type="long">
    <generator class="native"/>
</id>

...
</class>
```

标识符属性被映射到表`CATEGORY`的主键列`CATEGORY_ID`。这个属性的Hibernate类型是`long`，它在大部分数据库中映射到`BIGINT`列类型，并且已经被选择用来匹配由`native`标识符生成器生成的同一性值的类型。（下一节讨论标识符生成策略。）

对于JPA实体类，你在Java源代码中用注解映射标识符属性：

```
@Entity
@Table(name="CATEGORY")
public class Category {
    private Long id;
    ...

    @Id
```

```
    @GeneratedValue(strategy = GenerationType.AUTO)
    @Column(name = "CATEGORY_ID")
    public Long getId() {
        return this.id;
    }

    private void setId(Long id) {
        this.id = id;
    }
    ...
}
```

获取方法上的@Id注解，标志着它是标识符属性，而包含GenerationType.AUTO选项的@GenerateValue变成了一个原生的标识符生成策略，就像Hibernate XML映射中的native选项一样。注意，即使你没有定义strategy，默认也是GenerationType.AUTO，因此完全可以省略这个属性。你还指定了一个数据库列——否则Hibernate会使用属性名称。Java属性类型暗指的映射类型是java.lang.Long。

当然，也可以对所有属性使用直接的字段访问，包括数据库标识符：

```
@Entity
@Table(name="CATEGORY")
public class Category {

    @Id @GeneratedValue
    @Column(name = "CATEGORY_ID")
    private Long id;
    ...

    public Long getId() {
        return this.id;
    }
    ...

}
```

就像标准所定义的，当直接的字段访问被启用时，映射注解就被放在字段声明中。

给实体启用字段还是属性访问，这取决于强制的@Id注解的位置。在前面的例子中，它出现在字段中，因此这个类的所有属性都由Hibernate通过字段访问。在那之前的例子，在getId()方法上被注解的，使得能通过获取方法和设置方法访问所有的属性。

另一种方法，可以使用JPA XML描述符创建标识符映射：

```
<entity class="auction.model.Category" access="FIELD">
    <table name="CATEGORY"/>
    <attributes>
        <id name="id">
            <generated-value strategy="AUTO"/>
        </id>
        ...
    </attributes>
</entity>
```

除了用来测试Java对象同一性（a==b）和对象等同性（a.equals(b)）的操作之外，现在可以用a.getId().equals(b.getId())测试数据库同一性。这些概念有什么相似之处？在哪些情况下它们全部返回true？当所有都为真的时候，称作受保证的对象同一性的范围；9.2节将回到这个主题。

在Hibernate中使用数据库同一性很简单且容易理解。选择一个好的主键（和键生成策略）可能就比较困难了。接下来讨论这个问题。

4.2.3　数据库主键

Hibernate需要知道你生成主键的首选策略。不过，先要定义主键。

1．选择主键

备选的键是能够用来识别表中一个特定行的一列或者一组列。要变成主键，备选键必须满足下列属性：

- 它的值（对于备选键的任意列而言）永远不为空。
- 每一行都有唯一的值。
- 一个特定行的值永远不变。

如果一张表只有一个识别属性，根据定义，它就是主键。然而，几个列或者列的组合可能满足一张特定表的这些属性；你在备选键之间选择，给表决定最佳主键。没有被选作主键的备选键应该在数据库中声明为唯一键。

许多遗留的SQL数据模型使用自然主键。自然键是带有业务意义的键：属性或者属性的组合因其业务语义而唯一。自然键的例子有美国社会保障号码（U.S. Social Security Number）和澳大利亚税务档案号码（Australian Tax File Number）。区分自然键很简单：如果一个备选键属性在数据库上下文之外有意义，它就是自然键，无论它是否自动生成。考虑应用程序的用户：当他们谈论和使用应用程序时提到的键属性，它就是自然键。

经验表明，长远来看自然键通常会产生问题。好的主键必须是唯一、不变和必要的（从不为空或者未知）。很少有实体属性满足这些条件，并且有些确实无法被SQL数据库有效地索引（虽然这是一个实现细节，且不应该成为支持或者反对一个特定键的主要动机）。此外，你应该确定备选键定义可以在它成为主键之前的整个数据库生命周期中始终不变。改变主键的值（或者甚至定义）和所有引用它的外键，是一项令人沮丧的工作。此外，自然的备选键经常只有在复合的自然键中合并几个列才看得到。这些复合键，虽然必定适合于某些关系（比如多对多的关系中的链接表），但通常使得维护、特别查询和Schema演变变得更加困难。

基于这些原因，强烈建议你考虑合成标识符（synthetic identifier），也称代理键（surrogate key）。代理键没有业务意义——它们是由数据库或者应用程序生成的唯一值。应用程序的用户最好不要看到或者引用这些键值；它们是系统内部的一部分。一般情况下引入代理键列也合适：如果没有备选键，按定义，表不是一种由关系模型定义的关系——它允许完全相同的行——因此必须添加一个代理键列。生成代理键值有许多众所周知的方法。

2. 选择键生成器

Hibernate有几个内建的标识符生成策略。表4-1中列出了最有用的一些选项。

表4-1 Hibernate内建的标识符生成器模块

生成器名称	JPA GenerationType	选 项	描 述
native	AUTO	—	native同一性生成器挑选其他同一性生成器如identity、sequence或者hilo，取决于底层数据库的能力。使用这个生成器保证映射元数据可以移植到不同的数据库管理系统
identity	IDENTITY	—	这个生成器支持DB2、MySQL、MS SQL Server、Sybase和HypersonicSQL中的同一性列。返回的标识符类型为long、short或者int
sequence	SEQUENCE	sequence，parameters	这个生成器在DB2、PostgreSQL、Oracle、SAP DB或者Mckoi中创建一个序列；或者使用InterBase中的一个生成器。返回的标识符类型为long、short或者int。如果创建序列的其他设置要被添加到DDL，就使用sequence选项给序列定义一个目录名称（默认是hibernate_sequence）和parameters
increment	不可用	—	Hibernate启动时，这个生成器读取表的最大（数字）主键列值，并且每次插入一个新行时值就增加1。生成的标识符类型为long、short或者int。如果单服务器的Hibernate应用程序对数据库具有排它性访问时，这个生成器特别有效，但不应该被用于任何其他场合
hilo	不可用	table，column，max_lo	高/低位算法是生成long类型标识符的一种高效的方法，给定表和列（默认时，分别为hibernate_unique_key和next）作为高值的源。高/低位算法生成仅对特定数据库唯一的标识符。高值从全局的源中获取，并通过添加本地的低值被变成唯一。这种算法在标识符值的单个源必须被多个插入访问时避免拥堵。更多有关高/低位方法生成唯一标识符的信息请见“数据建模101”（Ambler，2002）。这个生成器每次获取高值时总要使用一个单独的数据库连接，因此它不受用户提供的数据库连接支持。换句话说，别把它与sessionFactory.openSession(myConnection)共用。max_lo选项定义了直到获取一个新的高值时添加了多少个低值。只有大于1的设置才有意义；默认为32767（Short.MAX_VALUE）
seqhilo	不可用	Sequence，parameters，max_lo	这个生成器的作用就像一般的hilo生成器，除了它使用一个具名的数据库序列来生成高值之外
（只适用于JPA）	TABLE	Table，catalog，Schema，PkColumnName，ValueColumnName，PkColumnValue，allocationSize	很像Hibernate的hilo策略，TABLE依赖持有最新生成的整数主键值的数据库表，每个生成器都被映射到这张表中的一行。每一行有两个列：pkColumnName和valueColumnName。pkColumnValue把每一行分配到特定的生成器，并且值列持有最新获取的主键。持久化提供程序每次都分配到allocationSize为整数为止

（续）

生成器名称	JPA GenerationType	选　项	描　述
uuid.hex	不可用	separator	这个生成器是一个128位的UUID（生成String类型的标识符的一种算法，在网络内部唯一）。IP地址与一个唯一的时间戳结合使用。UUID被编码成为长度32的十六进制数字的一个字符串，UUID表示法的每个组件之间带有一个可选的separator字符串。仅当你全局地需要唯一标识符时才使用这个生成器策略，例如当你必须规则地合并两个数据库的时候
guid	不可用	—	这个生成器在MySQL和SQL服务器中提供一个数据库生成的全局唯一的标识符字符串
select	不可用	key	这个生成器获取由数据库触发器分配的一个主键，它按某些唯一键选择行并获取主键值。这个策略需要另一个唯一的备选键列，且key选项必须被设置成唯一键列的名称

有些内建的标识符生成器可以通过选项得到配置。在原生的Hibernate XML映射中，把选项定义为键/值对：

```
<id column="MY_ID">
    <generator class="sequence">
        <para name="sequence">MY_SEQUENCE</parameter>
        <para name="parameters">
            INCREMENT BY 1 START WITH 1
        </para>
    </generator>
</id>
```

可以通过注解使用Hibernate标识符生成器，即使没有直接的注解可用：

```
@Entity
@org.hibernate.annotations.GenericGenerator(
    name = "hibernate-uuid",
    strategy = "uuid"
)
class name MyEntity {

    @Id
    @GeneratedValue(generator = "hibernate-uuid")
    @Column(name = "MY_ID")
    String id;
}
```

@GenericGenerator Hibernate扩展可以用来给Hibernate标识符生成器提供一个名称，在这个例子中为hibernate-uuid。然后这个名称被标准的generator属性引用。

生成器的这个声明和它按名称的分配，也必须通过注解应用到基于序列或基于表的标识符生成中。想象你要在所有的实体类中使用一个定制的序列生成器。因为这个标识符生成器必须是全局的，所以在orm.xml中声明：

```
<sequence-generator name="mySequenceGenerator"
    sequence-name="MY_SEQUENCE"
    initial-value="123"
    allocation-size="20"/>
```

这声明了具名MY_SEQUENCE的数据库序列，带有初始值123，可以用作数据库标识符生成的来源，并且每当持久化引擎需要标识符时，它就应该获得20个值。（但是注意，在编写本书之时，Hibernate Annotations忽略initial-Value设置。）

为了把这个标识符生成器应用到特定的实体，就利用它的名称：

```
@Entity
class name MyEntity {

    @Id @GeneratedValue(generator = "mySequenceGenerator")
    String id;
}
```

如果在实体级声明了另一个同名的生成器，且在class关键字之前，它就会覆盖全局的标识符生成器。可以用相同的方法声明和应用@TableGenerator。

你不受限于内建的策略，还可以通过实现Hibernate的IdentifierGenerator接口创建自己的标识符生成器。跟往常一样，查看现有标识符生成器的Hibernate源代码来寻求灵感是一种好策略。

甚至可能在单个领域模型中给持久化类混合标识符生成器，但是对于非遗留的数据，建议给所有实体使用相同的标识符生成策略。

对于遗留的数据和应用程序分配的标识符，情况就更复杂些。此时，应该经常坚持使用自然键，特别是复合键。复合键是由多个表列组成的自然键。因为复合标识符可能更难以使用一些，并且经常只出现在遗留的Schema中，因此我们仅在8.1节的上下文中讨论它。

从现在起，假设你已经把标识符属性添加到领域模型的实体类，并且在完成了每个实体和它的标识符属性的基础映射之后，继续映射实体的值类型属性。然而，有些特殊的选项可以简化或者增强你的类映射。

4.3 类映射选项

如果查看DTD（或者参考文档）中的<hibernate-mapping>和<class>元素，将会发现一些目前还没有讨论到的选项：

- CRUD SQL语句的动态生成；
- 实体易变性控制；
- 给查询命名实体；
- 映射包名称；
- 引用关键字和被保存的数据库标识符；
- 实现数据库命名约定。

4.3.1 动态的 SQL 生成

默认情况下，Hibernate在启动时给每个持久化类创建SQL语句。这些语句是用来读取单个行、

删除一行等的简单创建、读取、更新和删除。

Hibernate在启动时如何创建UPDATE语句？毕竟，这时候还不知道要更新的列。答案是生成的SQL语句更新所有的列，并且如果特定列的值没有被修改，这个语句就会把它设置为它的旧值。

在有些情况下，例如包含几百列的一个遗留表，在该表中，即使最简单的操作（假设只有一个列需要更新）的SQL语句也将很大，必须关闭这个启动时的SQL生成，并切换到运行时生成的动态语句。极为大量的实体也会影响启动时间，因为Hibernate必须为CRUD提前生成所有SQL语句。如果必须为数千个实体高速缓存许多语句的话，这个查询语句高速缓存的内存消耗也将很高（这通常不成问题）。

<class>映射元素中有两个属性可以禁用启动时CRUD SQL的生成：

```
<class name="Item"
   dynamic-insert="true"
   dynamic-update="true">
...
</class>
```

dynamic-insert属性告诉Hibernate是否在SQL INSERT中包括空的属性值，dynamic-update属性告诉Hibernate是否在SQL UPDATE中包括未被修改的属性。

如果使用JDK 5.0注解映射，就需要一个原生的Hibernate注解来启用动态的SQL生成：

```
@Entity
@org.hibernate.annotations.Entity(
    dynamicInsert = true, dynamicUpdate = true
)
public class Item { ...
```

来自Hibernate包的第二个@Entity注解使用额外的选项扩展了JPA注解，包括dynamicInsert和dynamicUpdate。

有时候可以避免生成任何UPDATE语句，如果持久化类被映射为不可变的话。

4.3.2　使实体不可变

一个特定类的实例可以是不可变的。例如，在CaveatEmptor中，对货品所做的Bid是不可变的。因此，在BID表中永远不需要执行UPDATE语句。Hibernate也可以进行一些其他优化，例如，如果通过把mutable属性设置为false来映射一个不可变的类，就可以避免脏检查：

```
<hibernate-mapping default-access="field">
    <class name="Bid" mutable="false">
    ...
    </class>
</hibernate-mapping>
```

如果没有公开类的任何属性的公有设置方法，POJO就是不可变的——所有值都在构造函数中设置。不用私有设置方法，而是经常更喜欢通过Hibernate对不可变的持久化进行直接的字段访问，因此不必编写没用的访问方法。可以用注解映射一个不可变的实体：

```
@Entity
@org.hibernate.annotations.Entity(mutable = false)
```

```
@org.hibernate.annotations.AccessType("field")
public class Bid { ...
```

原生的Hibernate @Entity注解再次用额外的选项扩展了JPA注解。此处我们还展现了Hibernate扩展注解@AccessType——这是个很少使用的注解。如前所述，一个特定实体类的默认访问策略，对于强制的@Id属性的位置来说是隐式的。然而，可以用@AccessType强制一个更细粒度的策略；它可以被放在类声明中（如前面的例子），或者甚至放在特定字段或者访问方法中。

我们来快速看看另一个问题：给查询命名实体。

4.3.3 给查询命名实体

默认情况下，所有类名都自动地“导入”到Hibernate查询语言（HQL）的命名空间。换句话说，可以在HQL中使用没有包前缀的短类名，这很方便。然而，如果给定的SessionFactory存在两个同名的类，这两个类可能在领域模型的不同包中，这个自动导入就可以关闭。

如果存在这种冲突，而又不改变默认的设置，Hibernate将不知道你正在HQL中引用哪个类。可以在<hibernate-mapping>根元素中设置autoimport="false"，对特定的映射文件把名称的自动导入关闭到HQL命名空间。

实体名称也可以被显式地导入到HQL命名空间。甚至可以导入非显式映射的类和接口，因此短名称可以被用在多态的HQL查询中：

```
<hibernate-mapping>
    <import class="auction.model.Auditable" rename="IAuditable"/>
</hibernate-mapping>
```

现在可以用HQL查询（如from IAuditable）来获取实现auction.model.Auditable接口的类的所有持久化实例。（如果此时你不知道这个特性是否与你有关，别着急，本书稍后会回到查询的话题。）注意<import>元素，就像<hibernate-mapping>所有其他直接的子元素一样，是一个适用于整个应用程序的声明，因此你不必（并且不能）在其他映射文件中重复它。

可以利用注解给实体一个显式的名称，如果短名称会在JPA QL或者HQL命名空间中导致冲突的话：

```
@Entity(name="AuctionItem")
public class Item { ... }
```

现在来考虑命名的另一个方面：包的声明。

4.3.4 声明包名称

CaveatEmptor应用程序的所有持久化类都在Java包auction.model中声明。然而，你并不想每当在关联、子类或者组件映射中命名这个类或者任何其他类时都重复整个包名称。替换做法是，指定一个package属性：

```
<hibernate-mapping package="auction.model">
    <class name="Item" table="ITEM">
      ...
    </class>
</hibernate-mapping>
```

现在出现在这个映射文件中的所有未限定的类名都将用被声明的包名作为前缀。假设在本书的所有映射示例中都有这个设置，并给CaveatEmptor模型类使用未限定的类名。

类和表的名称必须谨慎选择。然而，已经选择的名称可能被SQL数据库系统保存，因此这个名称必须用引号把它括起来。

4.3.5　用引号把 SQL 标识符括起来

默认情况下，Hibernate没有在生成的SQL中用引号把表和列名括起来。这使得SQL稍微更容易阅读，也允许你利用这样的事实：大部分SQL数据库在比较没有用引号括起来的标识符时都是不区分大小写的。有时候，特别是在遗留数据库中，你会遇到包含奇怪字符或者空格的标识符，或者你希望强制区分大小写。或者，如果依赖Hibernate的默认设置，Java中的类名称或者属性名称可能被自动转化为数据库管理系统中不允许的表名称或者列名称。例如，User类被映射到USER表，而它通常是SQL数据库中保留的一个关键字。Hibernate不认识任何DBMS产品的SQL关键字，因此数据库系统在启动或者运行时抛出异常。

如果在映射文档中用反引号（backtick）把表名或者列名括起来，Hibernate就会始终在生成的SQL中把这个标识符用引号括起来。下列属性声明强制Hibernate用括起来的列名"DESCRIPTION"生成SQL。Hibernate也知道Microsoft SQL Server需要变量[DESCRIPTION]，以及MySQL需要`DESCRIPTION`。

```
<property name="description"
          column="`DESCRIPTION`"/>
```

除了把所有表名和列名用反引号括起来之外，再没有其他办法可以强制Hibernate在任何地方使用括起来的标识符了。任何可能的时候，都应该考虑用保留的关键字名称重新命名表或者列。用反引号括起来对注解映射有效，但它是Hibernate的一个实现细节，而不是JPA规范的一部分。

4.3.6　实现命名约定

我们经常遇到对数据库表名和列名有严格约定的组织。Hibernate提供一种允许自动强制执行命名标准的特性。

假设CaveatEmptor中的所有表名都应该遵循模式CE_<table name>。一种解决方案是手工在映射文件中的所有<class>和集合元素上指定table属性。然而，这种方法既费时又容易遗忘。替代做法是，实现Hibernate的NamingStrategy接口，如代码清单4-1所示。

代表清单4-1　NamingStrategy实现

```
public class CENamingStrategy extends ImprovedNamingStrategy {

      public String classToTableName(String className) {
            return StringHelper.unqualify(className);
      }

      public String propertyToColumnName(String propertyName) {
          return propertyName;
    }
```

```
    public String tableName(String tableName) {
          return "CE_" + tableName;
    }

    public String columnName(String columnName) {
        return columnName;
    }

    public String propertyToTableName(String className,
                                      String propertyName) {
          return "CE_"
                      + classToTableName(className)
                      + '_'
                      + propertyToColumnName(propertyName);
    }
}
```

你扩展了ImproveNamingStrategy，它对你不想从头开始实现的NamingStrategy的所有方法提供默认的实现（请查看API文档和源代码）。仅当<class>映射没有指定显式的table名称时才调用classToTableName()方法。如果属性没有显式的column名称，就调用propertyToColumnName()方法。当声明了显式的名称时，则调用tableName()和columnName()方法。

如果启用这个CENamingStrategy，类映射声明

```
<class name="BankAccount">
```

将导致CE_BANKACCOUNT作为表的名称。

然而，如果指定了表名，像这样：

```
<class name="BankAccount" table="BANK_ACCOUNT">
```

那么CE_BANK_ACCOUNT就是表的名称。此时，BANK_ACCOUNT就被传递到tableName()方法。

NamingStrategy接口的最好特性是动态行为的潜能。为了启用一个特定的命名策略，可以在启动时把一个实例传递到Hibernate的Configuration：

```
Configuration cfg = new Configuration();
cfg.setNamingStrategy( new CENamingStrategy() );
SessionFactory sessionFactory sf =
                    cfg.configure().buildSessionFactory();
```

这允许你有多个基于相同映射文件的SessionFactory实例，每一个都使用不同的NamingStrategy。这在一个多客户端的安装中极为有用，在这里每个客户端都需要唯一的表名（但要相同的数据模型）。然而，处理这种需求的更好方法则是使用一个SQL Schema（一种命名空间），就如3.3.4节中已经讨论过的那样。

可以在Java Persistence中，用hibernate.ejb.naming_strategy选项在persistence.xml文件中设置一种命名策略实现。

既然已经讨论了实体的概念和最重要的映射，接下来就映射值类型。

4.4 细粒度的模型和映射

在花费了本章前半部分篇幅专门讨论实体和各自基本的持久化类映射选项之后，现在我们把

重点放在它们各种形式的值类型上。马上想起来的两种不同的值类型是：JDK带来的值类型的类（例如String或者基本类型），以及由应用程序的开发人员定义的值类型的类（例如Address和MonetaryAmount）。

首先，映射使用JDK类型的持久化类属性，并学习基础的映射元素和属性。然后着手定制的值类型类，并把它们映射为可嵌入的组件。

4.4.1　映射基础属性

如果映射持久化类，无论它是实体还是值类型，所有持久化属性都必须在XML映射文件中被显式地映射。另一方面，如果用注解映射类，它的所有属性都被默认为是持久化的。可以用@javax.persistence.Transient注解给属性进行标识来把它们排除，或者使用transient的Java关键字（通常只给Java序列化排除字段）。

在一个JPA XML描述符中，可以排除一个特定的字段或者属性：

```
<entity class="auction.model.User" access="FIELD">
    <attributes>
        ...
        <transient name="age"/>
    </attributes>
</entity>
```

一个典型的Hibernate属性映射定义了一个POJO的属性名称、一个数据库列名和一个Hibernate类型的名称，并且它可能经常省略类型。因此，如果description是（Java）类型java.lang.String的一个属性，Hibernate就默认使用Hibernate类型string（第5章将回到Hibernate类型系统的话题）。

Hibernate用反射决定属性的Java类型。因而，下列映射是等价的：

```
<property name="description" column="DESCRIPTION" type="string"/>
<property name="description" column="DESCRIPTION"/>
```

如果列名与属性名相同（忽略大小写），那么列名甚至可以省略。（这就是前面提到过的有意义的默认特性之一。）

对于一些更不寻常的情况，稍后会见到更多，可能要用<column>元素代替XML映射中的column属性。<column>元素提供更多的灵活性：它有更多可选择的属性，且可能不止一次地出现。（单个属性可以映射到不止一个列，这是第5章要讨论的一项技术。）下列两个属性是等价的：

```
<property name="description" column="DESCRIPTION" type="string"/>
<property name="description" type="string">
    <column name="DESCRIPTION"/>
</property>
```

<property>元素（尤其<column>元素）也定义主要应用到自动的数据库Schema生成的某些属性。如果没有正在使用hbm2ddl工具（请见2.1.4节）生成数据库Schema，就可以放心地省略它们。但是最好至少包括not-null属性，因为还没有提交给数据库时，Hibernate就能够报告非法的空属性：

```
<property name="initialPrice" column="INITIAL_PRICE" not-null="true"/>
```

JPA是基于异常模型的一个配置，因此可以依赖默认。如果持久化类的一个属性没有被注解，就应用下列规则：

- 如果属性是JDK类型，它自动就是持久化的。换句话说，它在Hibernate XML映射文件中像<property name="propertyName"/>一样处理。
- 如果属性的类被注解为@Embeddable，它就被映射为自己的类的组件。本章稍后将讨论组件的嵌入。
- 如果属性的类型是Serializable，它的值以序列化的形式保存。这通常不是你想要的，应该始终映射Java类，而不是在数据库中保存一堆字节。想象一下当几年之后应用程序消失时，维护包含这些二进制信息的数据库的情景。

如果不想依赖这些默认，就在一个特定的属性上应用@Basic注解。@Column注解相当于XML的<column>元素。以下是如何按要求声明一个属性值的例子：

```
@Basic(optional = false)
@Column(nullable = false)
public BigDecimal getInitialPrice { return initialPrice; }
```

@Basic注解把属性标识为在Java对象级上不可选。第二个设置，列映射上的nullable = false，只负责NOT NULL（非空）数据库约束的生成。Hibernate JPA实现在任何情况下对待这两个选项都一视同仁，因此你也可以只用其中一个注解达到这一目的。

在JPA XML描述符中，这个映射看起来是相同的：

```
<entity class="auction.model.Item" access="PROPERTY">
    <attributes>
        ...
        <basic name="initialPrice" optional="false">
            <column nullable="false"/>
        </basic>
    </attributes>
</entity>
```

Hibernate元数据中有许多选项可以用来声明Schema约束，例如列上的NOT NULL。然而，除了简单的可为空的能力之外，它们只用于在Hibernate从映射元数据中导出数据库Schema时生成DDL。我们将在8.3节中讨论SQL（包括DDL）的定制。另一方面，Hibernate Annotations包包括一个更高级且更复杂的数据验证框架，不仅可以用它在DDL中定义数据库Schema约束，还可以用于运行时的数据验证。第17章将讨论它。

用于属性的注解永远都在访问方法中吗？

1．定制属性访问

类的属性由持久化引擎直接（通过字段）或者间接（通过获取方法和设置方法属性访问方法）访问。在XML映射文件中，用hibernate-mapping root element的default-access="field|property|noop|custom.Class"属性给类控制默认的访问策略。被注解的实体从强制的@Id注解的位置继承默认的访问策略。例如，如果@Id已经在字段中（而不是获取方法中）声明，那么所有其他的属性映射注解（如货品description属性的列名）也在字段中声明：

```
@Column(name = "ITEM_DESCR")
private String description;

public String getDescription() { return description; }
```

这是JPA规范定义的默认行为。然而，Hibernate允许利用@org.hibernate.annotations.AccessType(<strategy>)注解灵活地定制访问策略：

- 如果在类/实体级中设置AccessType，类的所有属性都根据选中的策略访问。属性级注解是在字段还是在获取方法上，这取决于策略。这个设置覆盖来自标准的@Id注解位置的任何默认值。
- 如果给字段访问默认或者显式地设置一个实体，字段中的AccessType（"property"）注解就把这个特定的属性转换为通过属性获取方法/设置方法的运行时访问。AccessType注解的位置仍然为字段。
- 如果给属性访问默认或者显式地设置一个实体，获取方法中的AccessType（"field"）注解就把这个特定的属性转换为通过同名字段的运行时访问。AccessType注解的位置仍然为获取方法。
- 任何@Embedded类都继承默认的或者显式声明的自己根实体类的访问策略。
- 任何@MappedSuperclass属性都通过被映射实体类的默认或者显式声明的访问策略而被访问。

也可以用access属性在Hibernate XML映射中控制属性级上的访问策略：

```
<property name="description"
          column="DESCR"
          access="field"/>
```

或者，可以用default-access属性给一个根<hibernate-mapping>元素内部的所有类映射设置访问策略。

除了字段和属性访问之外，另一个可能有用的策略是noop。它映射Java持久化类中不存在的属性。这听起来有点奇怪，但是它让你在HQL查询中指向这个“虚拟”属性（换句话说，只在HQL查询中使用数据库列）。

如果没有合适的内建访问策略，也可以通过实现接口org.hibernate.property.PropertyAccessor定义自己的定制属性访问策略。在access映射属性或者@AccessType注解中设置（完全匹配的）类名。看看Hibernate源代码寻求灵感；这是个很简单的练习。

有些属性根本不映射到列。特别是衍生属性（derived property），它从SQL表达式中取值。

2. 使用衍生属性

衍生属性的值在运行时计算，通过对利用formula属性定义的表达式求值。例如，可以把totalIncludingTax属性映射到SQL表达式：

```
<property name="totalIncludingTax"
          formula="TOTAL + TAX_RATE * TOTAL"
          type="big_decimal"/>
```

这个给定的SQL公式在每次从数据库获取实体时求值（并且在任何其他时间不求值，因此如果其他的属性被修改，这个结果就可能过时）。属性没有列属性（或者子元素），并且永远不会出

现在SQL的INSERT或者UPDATE中，而只在SELECT中。公式可能指向数据库表的列，它们可以调用SQL函数，甚至包括SQL子查询。SQL表达式实际上被传递到底层的数据库；如果你不够细心，并依赖特定于供应商的操作符或者关键字，这就是把映射文件绑定到一个特定数据库产品的好机会。

公式也可使用Hibernate注解：

```
@org.hibernate.annotations.Formula("TOTAL + TAX_RATE * TOTAL")
public BigDecimal getTotalIncludingTax() {
    return totalIncludingTax;
}
```

下列示例使用一个关联的子查询来计算一件货品所有出价的平均值：

```
<property
   name="averageBidAmount"
   type="big_decimal"
   formula=
    "( select AVG(b.AMOUNT) from
       BID b where b.ITEM_ID = ITEM_ID )"/>
```

注意未限定的列名指向衍生属性所属的这个类的表列。

另一种特殊的属性依赖数据库生成的值。

3. 生成的和默认的属性值

想象类的一个特定属性有着数据库生成的值，通常在第一次插入实体行的时候。典型的数据库生成的值是创建的时间戳、对一件货品的默认价格，以及每次修改时运行的触发器。

典型的Hibernate应用程序需要刷新包含由数据库为其生成值的任何属性的对象。然而，标识属性为已生成（generated），让应用程序把这个责任委托给了Hibernate。本质上，每当Hibernate给定义了已生成属性的实体执行SQL INSERT或者UPDATE时，它在获取已生成的值之后立即执行SELECT。使用property映射中的generated开关启用这个自动刷新：

```
<property name="lastModified"
          column="LAST_MODIFIED"
          update="false"
          insert="false"
          generated="always"/>
```

标记为数据库生成的属性还必须是非可插入和非可更新的，用insert和update属性进行控制它们。如果两者都设置为false，属性的列就永远不会出现在INSERT或者UPDATE语句中——属性值是只读的。而且，通常不在类中给不可变的属性添加公有的设置方法（并切换到字段访问）。

利用注解，通过@Generated的Hibernate注解声明不可变性（和自动刷新）：

```
@Column(updatable = false, insertable = false)
@org.hibernate.annotations.Generated(
    org.hibernate.annotations.GenerationTime.ALWAYS
)
private Date lastModified;
```

可用的设置为GenerationTime.ALWAYS和GenerationTime.INSERT，并且XML映射中与之相当的选项是generated="always"和generated="insert"。

数据库生成的属性值的一个特殊情况是默认值。例如，你可能想要实现一种规则，即每件拍卖货品的成本至少为$1。首先，得把它作为INITIAL_PRICE列的默认值添加到数据库目录：

```
create table ITEM (
    ...
    INITIAL_PRICE number(10,2) default '1',
    ...
);
```

如果使用Hibernate的Schema导出工具hbm2ddl，则可以通过把default属性添加到属性映射来启用这项输出：

```
<class name="Item" table="ITEM"
       dynamic-insert="true" dynamic-update="true">
    ...
    <property name="initialPrice" type="big_decimal">
        <column name="INITIAL_PRICE"
                default="'1'"
                generated="insert"/>
    </property>
    ...
 </class>
```

注意，你还必须启用动态的插入并更新语句的生成，以便包含默认值的列不会被包括在每个语句中，如果它的值为null（否则将插入NULL，而不是默认值）。此外，已经被变成持久化但尚未清除到数据库，且尚未再次刷新的Item的一个实例，将不在对象属性上设置默认值。换句话说，你需要执行一个显式的清除：

```
Item newItem = new Item(...);
session.save(newItem);

newItem.getInitialPrice();    // is null

session.flush();              // Trigger an INSERT
// Hibernate does a SELECT automatically

newItem.getInitialPrice();    // is $1
```

因为设置了generated="insert"，Hibernate知道读取数据库生成的属性值立即需要一个额外的SELECT。

可以用注解把默认的列值映射为列的DDL定义的一部分：

```
@Column(name = "INITIAL_PRICE",
        columnDefinition = "number(10,2) default '1'")
@org.hibernate.annotations.Generated(
    org.hibernate.annotations.GenerationTime.INSERT
)
private BigDecimal initalPrice;
```

columnDefinition属性包括列DDL的完整属性，包含数据类型和所有约束。记住，一个实际上不可移植的SQL数据类型可能把注解映射绑定到一个特定的数据库管理系统。

8.3节将回到约束和DDL定制的主题。

接下来，将映射用户自定义的值类型的类。如果在两个类之间搜索一种复合关系的话，可以

很轻松地在UML类图中认出它们。它们其中一个是依赖的类，即组件。

4.4.2 映射组件

目前为止，对象模型的类已经全部成为实体类，每一个都有自己的生命周期和同一性。然而，User类与Address类有着一种特殊的关联，如图4-2所示。

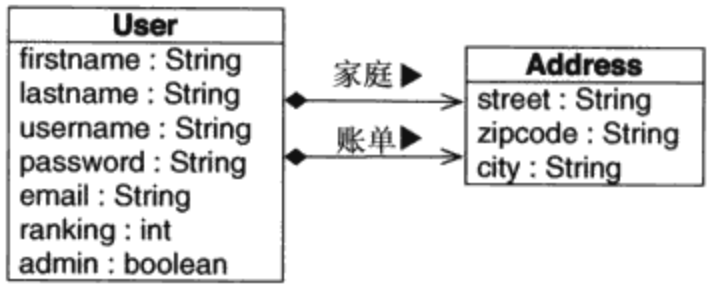


图4-2 使用复合的User和Address之间的关系

从对象建模的角度来看，这个关联是一种聚集——是整体-部分（part-of）的关系。聚集是一种强健的关联形式；它对于对象的生命周期有一些额外的语义。在这种情况下，我们有一种更强健的形式：复合（composition），在这里部分的生命周期完全依赖于整体的生命周期。

对象建模专家和UML设计师声称，就实际的Java实现而言，在这个复合和其他更弱形式的关联之间是没有区别的。但是在ORM的上下文中，则有一个很大的区别：复合的类经常是一个备选的值类型。

把Address映射为值类型，把User映射为实体。这影响POJO类的实现吗？

Java没有复合的概念——类或者属性无法被标记为组件或者复合。唯一的区别是对象标识符：组件没有独立的同一性，因此持久化组件类不需要标识符属性或者标识符映射。它是一个简单的POJO：

```
public class Address {

    private String street;
    private String zipcode;
    private String city;

    public Address() {}

    public String getStreet() { return street; }
    public void setStreet(String street) { this.street = street; }

    public String getZipcode() { return zipcode; }
    public void setZipcode(String zipcode) {
        this.zipcode = zipcode; }

    public String getCity() { return city; }
    public void setCity(String city) { this.city = city; }

}
```

User和Address之间的复合是一个元数据级的概念；你只需要告诉Hibernate，Address是映射文档中的值类型，或者是包含注解的值类型。

1. XML格式的组件映射

Hibernate对用户自定义的类使用术语component（组件），该类被持久化到与自己的实体相同的表，代码清单4-2中显示了这样的一个例子。（此处单词component的用法，与建筑级的概念没

有任何关系，而是同软件组件中的概念。）

代码清单4-2　包含组件Address的User类的映射

```
<class name="User" table="USER">

    <id name="id" column="USER_ID" type="long">
        <generator class="native"/>
    </id>

    <property name="loginName" column="LOGIN" type="string"/>

    <component name="homeAddress" class="Address">          ❶
        <property name="street" type="string"
                  column="HOME_STREET" not-null="true"/>
        <property name="city" type="string"
                  column="HOME_CITY" not-null="true"/>
        <property name="zipcode" type="string"
                  column="HOME_ZIPCODE" not-null="true"/>
    </component>

    <component name="billingAddress" class="Address">       ❷
        <property name="street" type="string"
                  column="BILLING_STREET" not-null="true"/>
        <property name="city" type="string"
                  column="BILLING_CITY" not-null="true"/>
        <property name="zipcode" type="string"
                  column="BILLING_ZIPCODE" not-null="true"/>
    </component>

    ...

</class>
```

❶ 在<component>元素的内部声明Address的持久化属性。User类的属性被命名为homeAddress。

❷ 重用同一个组件类，把这个类型的另一个属性映射到同一张表。

图4-3显示了Address类的属性如何被持久化到与User实体相同的表中。

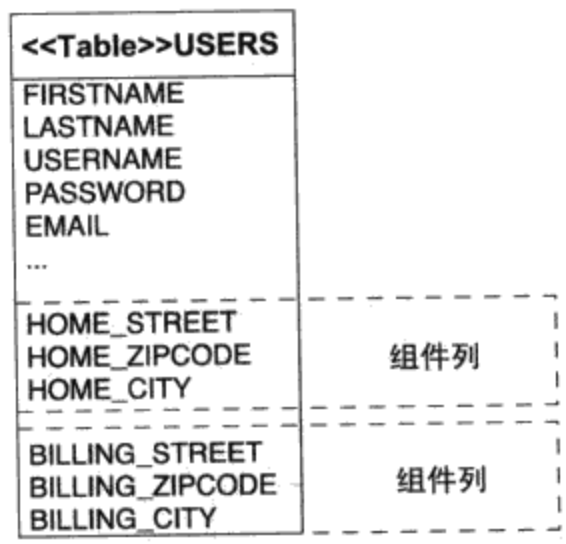


图4-3　包含Address组件的User的表属性

注意，在这个例子中，你把复合关联建模为单向的。你无法从Address导航到User。Hibernate支持单向的和双向的复合，但是单向复合要常见得多。双向映射的一个例子如代码清单4-3所示。

代码清单4-3 把后退指针添加到复合

```
<component name="homeAddress" class="Address">
    <parent name="user"/>
    <property name="street" type="string"
              column="HOME_STREET" not-null="true"/>
    <property name="city" type="string"
              column="HOME_CITY" not-null="true"/>
    <property name="zipcode" type="stringshort"
              column="HOME_ZIPCODE" not-null="true"/>
</component>
```

在代码清单4-3中，<parent>元素把类型User的属性映射到自己的实体，在这个例子中，它是具名user的那个属性。然后可以调用Address.getUser()在另一个方向进行导航。这真是一个简单的后退指针。

Hibernate组件可以拥有其他的组件，甚至关联到其他实体。这种灵活性是Hibernate支持细粒度的对象模型的基础。例如，可以创建Location类，包含关于Address所有者的家庭住址的详细信息：

```
<component name="homeAddress" class="Address">
    <parent name="user"/>

    <component name="location" class="Location">
        <property name="streetname" column="HOME_STREETNAME"/>
        <property name="streetside" column="HOME_STREETSIDE"/>
        <property name="housenumber" column="HOME_HOUSENR"/>
        <property name="floor" column="HOME_FLOOR"/>
    </component>

    <property name="city" type="string" column="HOME_CITY"/>
    <property name="zipcode" type="string" column="HOME_ZIPCODE"/>

</component>
```

Location类的设计相当于Address类。你现在有3个类、1个实体和2个值类型，全部被映射到同一张表。

现在用JPA注解映射组件。

2. 注解被嵌入的类

Java Persistence规范把组件称为被嵌入的类。为了用注解映射被嵌入的类，可以在自己的实体类中声明一个特定的属性为@Embedded，在这个例子中是User的homeAddress：

```
@Entity
@Table(name = "USERS")
public class User {

    ...
```

```
    @Embedded
    private Address homeAddress;

    ...
}
```

如果没有把一个属性声明为@Embedded，并且它不是JDK类型，Hibernate就在被关联的类中查找@Embedded注解。如果它存在，属性就自动地被映射为一个依赖的组件。

这就是被嵌入的类的样子：

```
@Embeddable
public class Address {

    @Column(name = "ADDRESS_STREET", nullable = false)
    private String street;

    @Column(name = "ADDRESS_ZIPCODE", nullable = false)
    private String zipcode;

    @Column(name = "ADDRESS_CITY", nullable = false)
    private String city;

    ...
}
```

可以在被嵌入的类中进一步定制独立的属性映射，例如用@Column注解。现在USERS表除了包含其他的之外，还包含列ADDRESS_STREET、ADDRESS_ZIPCODE和ADDRESS_CITY。任何包含组件字段的其他实体表（假设，Order类也有Address）都使用相同的列选项。也可以添加后退指针属性到Address被嵌入的类，并用@org.hibernate.annotations.Parent映射它。

有时候，你会想要从外部对一个特定的实体覆盖在被嵌入的类内部所做的设置。例如，下面是如何重命名列的例子：

```
@Entity
@Table(name = "USERS")
public class User {

    ...

    @Embedded
    @AttributeOverrides( {
        @AttributeOverride(name   = "street",
                           column = @Column(name="HOME_STREET") ),
        @AttributeOverride(name   = "zipcode",
                           column = @Column(name="HOME_ZIPCODE") ),
        @AttributeOverride(name   = "city",
                           column = @Column(name="HOME_CITY") )
    })
    private Address homeAddress;

    ...
}
```

User类中的这个新的@Column声明覆盖了被嵌入的类的设置。注意，被嵌入的@Column注解

上的所有属性都被替换，因此它们不再nullable = false。

在JPA XML描述符中，被嵌入的类和复合的映射看起来像下面这样：

```
<embeddable class="auction.model.Address access-type="FIELD"/>

<entity class="auction.model.User" access="FIELD">
    <attributes>
        ...
        <embedded name="homeAddress">
            <attribute-override name="street">
                <column name="HOME_STREET"/>
            </attribute-override>
            <attribute-override name="zipcode">
                <column name="HOME_ZIPCODE"/>
            </attribute-override>
            <attribute-override name="city">
                <column name="HOME_CITY"/>
            </attribute-override>
        </embedded>
    </attributes>
</entity>
```

对于类被映射为组件有2条重要的限制。第一，就像对于所有的值类型一样，共享引用是不可能的。组件homeAddress没有它自己的数据库同一性（主键），因此无法被除了User的包含实例之外的任何对象引用。

第二，没有优雅的方式表示对Address的空引用。作为任何优雅的方法的替代，Hibernate在组件的所有被映射列中都把空组件表示为空值。这意味着如果保存一个包含全部空属性值的组件对象，Hibernate就会在从数据库中获取自己的实体对象时返回一个空组件。

通读本书，你将发现更多的组件映射（甚至它们的集合）。

4.5 小结

本章介绍了实体和值类型之间的本质区别，以及这些概念如何影响领域模型实现为持久化的Java类。

实体是系统更粗粒度的类。它们的实例有一个独立的生命周期和它们自己的同一性，可以被许多其他的实例引用。另一方面，值类型依赖于特定的实体类。值类型的实例有着受它自己的实体实例限制的生命周期，并且它只能被一个实体引用——没有独立的同一性。

我们探讨了Java同一性、对象等同性和数据库同一性，以及是什么创造了好主键。你学习了Hibernate给主键值内建了哪些生成器，以及如何使用和扩展这个标识符系统。

你还学习了各种（主要可选的）类映射选项，最后学习了基础的属性和值类型组件在XML映射和注解中如何被映射。

为了方便起见，表4-2概括了本章讨论过的Hibernate和Java Persistence相关概念之间的区别。

表4-2　第4章的Hibernate和JPA对照表

Hibernate Core	Java Persistence和EJB 3.0
实体类型和值类型的类是支持富且细粒度的领域模型的本质概念	JPA规范也是如此，但是称值类型为“可嵌入的类”。然而，嵌套的可嵌入类被认为是一项不可移植的特性
Hibernate支持10个开箱即用的标识符生成策略	JPA标准化了4个标识符生成器的一个子集，但是允许供应商进行扩展
Hibernate可以通过字段、访问方法或者用任何定制的PropertyAccessor实现访问属性。可以对一个特定的类混合使用这些策略	JPA标准化了通过字段或者访问方法的属性访问，并且不能对没有Hibernate扩展注解的特定类混合使用这些策略
Hibernate支持公式属性和数据库生成的值	JPA不包括这些特性，需要Hibernate扩展

第5章将探讨继承，以及实体类的层次结构如何通过各种策略映射。我们也将讨论Hibernate映射类型系统，在几个实例中展现值类型的转换器。

第5章 继承和定制类型

本章内容

- 继承映射策略
- Hibernate映射类型系统
- 映射类型的定制

我们此前有意不过多讨论继承映射。把类的层次结构映射到表可能是一个复杂的问题，本章将介绍各种策略。你将学习在一个特定的场景中要选择哪种策略。

Hibernate类型系统，包含把Java值类型的属性转换为SQL数据类型的所有内建的转换器，是本章要讨论的第二大主题。

让我们从实体继承的映射开始。

5.1 映射类继承

把类映射到数据库表的一种简单的策略可能是“每个实体持久化类对应一张表”。这种方法听起来够简单，在我们遇到继承之前确实一直运行得很好。

继承是面向对象和关系世界之间如此明显的一种结构的不匹配，因为面向对象的系统模型既是is a又是has a关系。基于SQL的模型只提供实体之间的has a关系；SQL数据库管理系统不支持类型继承——甚至即使可用，通常也是私有的或者不完全的。

表示一个继承层次结构有4种不同的方法：

- 每个带有隐式多态的具体类一张表——使用非显式的继承映射和默认的运行时多态行为。
- 每个具体类一张表——完全放弃来自SQL模式的多态和继承关系。
- 每个类层次结构一张表——通过反规范化SQL模式启用多态，并利用保存类型信息的一个类型辨别标志列。
- 每个子类一张表——把is a（继承）关系表示为has a（外键）关系。

本节采用了一种自顶向下的方法；假设你正从一个领域模型开始，并试图获取一个新的SQL Schema。然而，如果你自底向上从现有的数据库表开始时，所描述的映射策略正好有关。我们将介绍一些帮助你处理不理想的表设计的技巧。

5.1.1　每个带有隐式多态的具体类一张表

假设我们坚持用建议过的最简单的方法。可以准确地给每个（非抽象的）类使用一张表。类的所有属性，包括被继承的属性，都可以被映射到这张表的列，如图5-1所示。

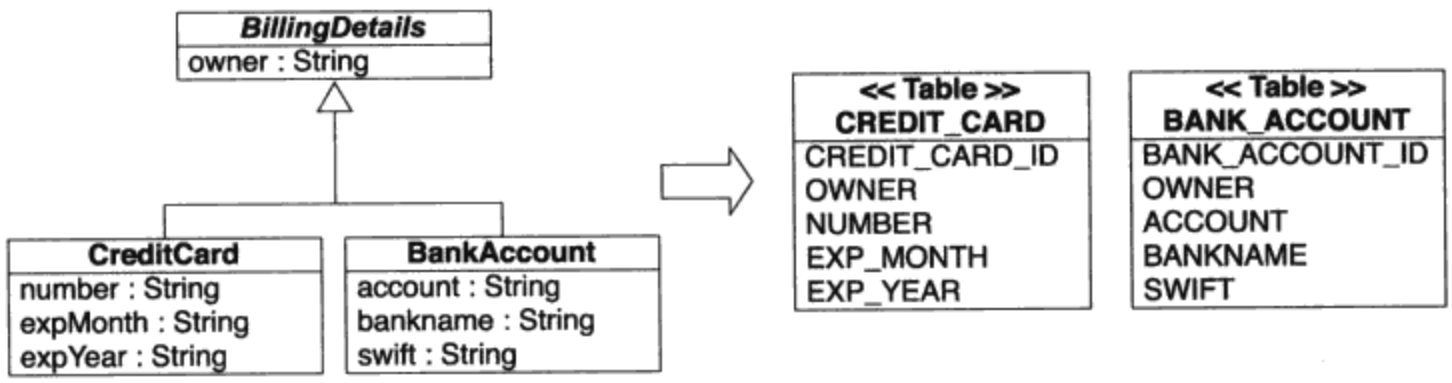


图5-1　把所有具体类映射到一张独立的表

在Hibernate中你不必做任何特别的事来启用多态行为。对CreditCard和BankAccount的映射很简单，各自都在自己的实体<class>元素中，就像我们已经对没有基类（或者持久化接口）的类所做的那样。Hibernate仍然知道这个基类（或者任何接口），因为它在启动时扫描持久化类。

这种方法的主要问题在于，它不太支持多态关联。在数据库中，关联通常被表示为外键关系。在图5-1中，如果子类全部被映射到不同的表，对它们基类（本例中为抽象的BillingDetails）的多态关联就无法被表示为一个简单的外键关系。这在我们的领域模型中会有问题，因为BillingDetails与User关联；两张子类表都需要一个对USERS表的外键引用。或者，如果User与BillingDetails有个多对一的关系，USERS表就需要单个的外键列，它必须同时引用两个具体的子类表。这用一般的外键约束是不可能的。

多态查询（它返回与被查询类的接口相匹配的所有类的对象）也有问题。针对基类的查询必须作为几个SQL SELECT执行，每个具体的子类一个。对于针对BillingDetails类的查询，Hibernate使用下列SQL：

```
select CREDIT_CARD_ID, OWNER, NUMBER, EXP_MONTH, EXP_YEAR ...
from CREDIT_CARD

select BANK_ACCOUNT_ID, OWNER, ACCOUNT, BANKNAME, ...
from BANK_ACCOUNT
```

注意每个具体的子类都需要一个单独的查询。另一方面，针对具体类的查询则十分繁琐，并且执行得很好——只需要一条语句。

（还要注意本书此处以及其他地方，都是介绍概念上与Hibernate执行的SQL一致的SQL。实际的SQL看起来可能稍有不同。）

这种映射策略进一步的概念问题在于，几张不同表的几个不同的列具有完全相同的语义。这使得模式演变更为复杂。例如，对一个基类属性的改变导致了多个列的变化。它也使得把应用到所有子类的数据库完整性约束实现起来变得更加困难。

我们推荐这种方法（仅仅）用于类层次结构的最顶层，那里通常不需要多态，且未来也不太可能修改基类。

Java Persistence接口也不支持完全的多态查询；只有被映射的实体（@Entity）可以正式成为Java Persistence查询的一部分（注意Hibernate查询接口是多态的，即使你用注解映射）。

如果依赖这个隐式多态，就像通常一样用@Entity映射具体的类。但是也必须复制基类的属性，把它们映射到所有具体的类表。默认情况下，基类的属性被忽略并且不是持久化的！你要在具体的子类表中注解超类来启用它属性的嵌入：

```
@MappedSuperclass
public abstract class BillingDetails {

    @Column(name = "OWNER", nullable = false)
    private String owner;

        ...
}
```

现在映射具体的子类：

```
@Entity
@AttributeOverride(name = "owner", column =
  @Column(name = "CC_OWNER", nullable = false)
)
public class CreditCard extends BillingDetails {

    @Id @GeneratedValue
    @Column(name = "CREDIT_CARD_ID")
    private Long id = null;

    @Column(name = "NUMBER", nullable = false)
    private String number;

    ...
}
```

可以用@AttributeOverride注解在一个子类中覆盖来自超类的列映射。在CREDIT_CARD表中重新命名OWNER列为CC_OWNER。数据库标识符也可以在超类中声明，给所有的子类使用一个共用的列名和生成器策略。

在JPA XML描述符中重复相同的映射：

```
<entity-mappings>

    <mapped-superclass class="auction.model.BillingDetails"
                       access="FIELD">
        <attributes>
            ...
        </attributes>
    </mapped-superclass>

    <entity class="auction.model.CreditCard" access="FIELD">
        <attribute-override name="owner">
            <column name="CC_OWNER" nullable="false"/>
        </attribute-override>
        <attributes>
```

```
            ...
        </attributes>
    </entity>
    ...
</entity-mappings>
```

说明　组件是值类型；因此，本章介绍的一般实体继承规则不适用。但是可以把子类映射为组件，通过把超类（或者接口）的所有属性都包括在组件映射中。有了注解，可以在正映射的可嵌入组件的超类上使用@MappedSuperclass注解，就像对实体使用时一样。注意该特性仅在Hibernate Annotations中可用，不是标准的或者可移植的。

在SQL UNION操作的帮助下，可以解决多态查询和关联的大部分问题，这些问题是这个映射策略所带来的。

5.1.2　每个带有联合的具体类一张表

首先，让我们考虑用BillingDetails把联合子类映射为抽象类（或者接口），就像前一节一样。在这种情况下，我们再次有了两张表，并在这两张表中都复制超类列：CREDIT_CARD和BANK_ACCOUNT。这里新出现的东西是包括超类的一个特殊的Hibernate映射，如代码清单5-1所示。

代码清单5-1　使用<union-subclass>继承策略

```
<hibernate-mapping>
    <class
        name="BillingDetails"                    ❶
        abstract="true">
        <id                                      ❷
            name="id"
            column="BILLING_DETAILS_ID"
            type="long">
            <generator class="native"/>
        </id>
        <property                                ❸
            name="name"
            column="OWNER"
            type="string"/>
            ...
        <union-subclass                          ❹
            name="CreditCard" table="CREDIT_CARD">
            <property name="number" column="NUMBER"/>
            <property name="expMonth" column="EXP_MONTH"/>
            <property name="expYear" column="EXP_YEAR"/>
        </union-subclass>
```

```
        <union-subclass
            name="BankAccount" table="BANK_ACCOUNT">
        ...
    </class>
</hibernate-mapping>
```

❶ 抽象的超类或者接口必须被声明为abstract="true"；否则超类的实例就需要一张单独的表。

❷ 数据库标识符映射被该层中所有的具体类共用。CREDIT_CARD和BANK_ACCOUNT表都有一个BILLING_DETAILS_ID主键列。现在数据库标识符属性必须被所有的子类共享；因而你必须把它移到BillingDetails里面，并且从CreditCard和BankAccount中把它移除。

❸ 超类（或者接口）的属性在此处声明，并被所有具体的类映射继承。这样避免了相同映射的复制。

❹ 具体的子类被映射到一张表；表继承超类（或者接口）标识符和其他的属性映射。

你可能注意到了，这种策略的第一个好处是超类（或者接口）属性的共享声明。你不再非得给所有具体的类复制这些映射了——Hibernate会替你完成。记住，SQL模式仍然不知道继承；实际上，我们已经把两个不相关的表映射到了一个更富有表现力的类结构。除了不同的主键列名之外，这两张表看起来完全相同，如图5-1所示。

在JPA注解中，这个策略称作TABLE_PER_CLASS：

```
@Entity
@Inheritance(strategy = InheritanceType.TABLE_PER_CLASS)
public abstract class BillingDetails {

    @Id @GeneratedValue
    @Column(name = "BILLING_DETAILS_ID")
    private Long id = null;

    @Column(name = "OWNER", nullable = false)
    private String owner;

    ...
}
```

数据库标识符和它的映射必须出现在超类中，在所有子类和它们的表中共享。每个子类中的@Entity注解便是所需要的全部：

```
@Entity
@Table(name = "CREDIT_CARD")
public class CreditCard extends BillingDetails {

    @Column(name = "NUMBER", nullable = false)
    private String number;

    ...
}
```

注意，TABLE_PER_CLASS在JPA标准中被指定为可选，因此并非所有的JPA实现都可以支持它。实际的实现也是由供应商决定的——在Hibernate中，它相当于XML文件中的<union-subclass>映射。

同样的映射在JPA XML描述符中看起来像这样：

```
<entity-mappings>

    <entity class="auction.model.BillingDetails" access="FIELD">
        <inheritance strategy="TABLE_PER_CLASS"/>
        ...
    </entity>

    <entity class="auction.model.CreditCard" access="FIELD"/>
    <entity class="auction.model.BankAccount" access="FIELD"/>

</entity-mappings>
```

如果超类是具体的，就需要另外一张表来存放这个类的实例。必须再次强调，数据库表之间仍然没有关系，除了它们共享一些类似的列之外。如果检验多态查询，这个映射策略的好处会更加明显。例如，对BillingDetails的查询执行下列SQL语句：

```
select
    BILLING_DETAILS_ID, OWNER,
    NUMBER, EXP_MONTH, EXP_YEAR,
    ACCOUNT, BANKNAME, SWIFT
    CLAZZ_
from
  ( select
      BILLING_DETAILS_ID, OWNER,
      NUMBER, EXP_MONTH, EXP_YEAR,
      null as ACCOUNT, null as BANKNAME, null as SWIFT,
      1 as CLAZZ_
    from
      CREDIT_CARD

    union

    select
        BILLING_DETAILS_ID, OWNER,
        null as NUMBER, null as EXP_MONTH, null as EXP_YEAR, ...
        ACCOUNT, BANKNAME, SWIFT,
        2 as CLAZZ_
      from
        BANK_ACCOUNT
  )
```

这个SELECT使用一个FROM子句的子查询，从所有具体的类表中获取BillingDetails的所有实例。这些表与一个UNION操作符结合起来，并把文字（在这个例子中是1和2）插入到中间结果中；Hibernate读取结果，通过特定行中的数据实例化正确的类。联合（union）要求被合并的查询要在相同的列上方投影；因而，我们必须用NULL填满不存在的列。你可能会问，这个查询是否真的比两个单独的声明执行得更好？这里我们可以让数据库优化器找到最好的执行计划，合并来自几张表的行，而不是在内存中合并两个结果集，就像Hibernate的多态加载器引擎所做的那样。

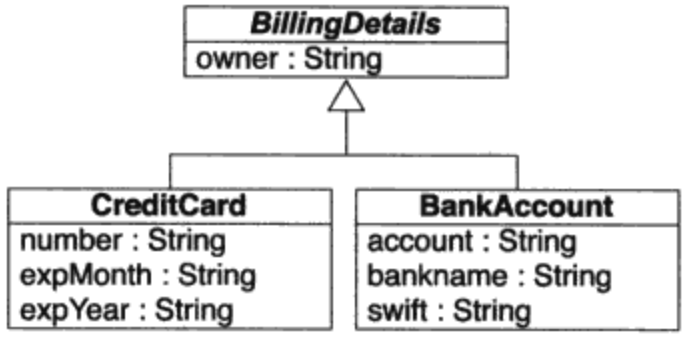
另一个更为重要的好处是处理多态关联的能力；例如，从User到BillingDetails的关联映射现在就有可能了。Hibernate可以用一个UNION查询把单张表模拟成为关联映射的目标。7.3节将

深入讨论这一主题。

目前为止，我们所讨论的继承映射策略都不需要额外考虑SQL Schema。不需要外键，并且关系也被适当地标准化。这种情况随着下一个策略而改变。

5.1.3 每个类层次结构一张表

整个类层次结构可以被映射到单张表。这张表把所有类的所有属性的列都包括在层次结构中。由特定行表示的具体子类通过一个类型辨别标志列的值进行识别。这种方法如图5-2所示。



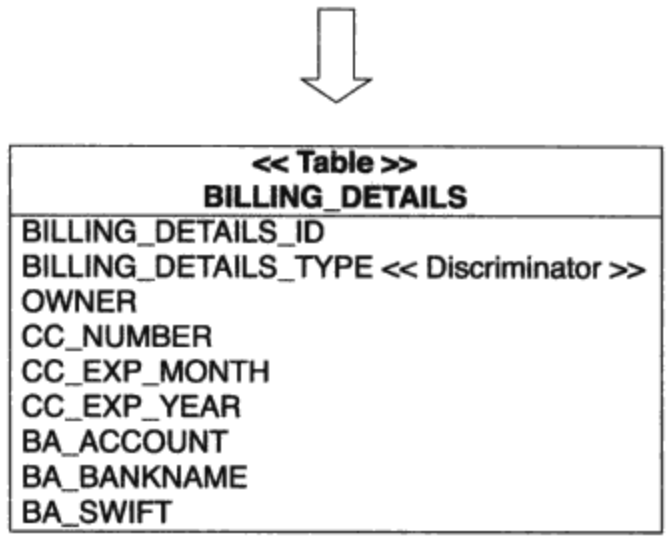


图5-2 把整个类层次结构映射到单张表

这个映射策略在性能和简单性方面都胜出一筹。它是表示多态的最佳方法——多态和非多态的查询都执行得很好——并且更易于手工实现。不用复杂的联结或者联合也有可能生成特殊的报表。Schema演变很简单。

有一个重大的问题：子类声明的属性的列必须声明为可为空。如果每个子类都定义几个不可为空的属性，从数据库完整性的角度来说，没有NOT NULL约束可能会是一个严重的问题。另一个重要的问题是标准化。我们已经创建了非键列之间的功能依赖，违背了第三范式。跟往常一样，性能的反规范化可能会令人误解，因为它为了也能通过SQL执行计划的适当优化而也可能实现的即时获取（换句话说，问问你的数据库管理员），却牺牲了数据的长期稳定性、可维护性和数据完整性。

在Hibernate中，用<subclass>元素给每个类层次结构映射创建一张表，如代码清单5-2所示。

代码清单5-2　Hibernate <subclass>映射

```
<hibernate-mapping>
    <class                                    ❶
        name="BillingDetails"
        table="BILLING_DETAILS">

        <id
            name="id"
            column="BILLING_DETAILS_ID"
            type="long">
            <generator class="native"/>
        </id>

        <discriminator                        ❷
            column="BILLING_DETAILS_TYPE"
            type="string"/>

        <property                             ❸
            name="owner"
            column="OWNER"
            type="string"/>

            ...

        <subclass                             ❹
            name="CreditCard"
            discriminator-value="CC">

            <property name="number" column="CC_NUMBER"/>
            <property name="expMonth" column="CC_EXP_MONTH"/>
            <property name="expYear" column="CC_EXP_YEAR"/>

        </subclass>

        <subclass
            name="BankAccount"
            discriminator-value="BA">
              ...

        </class>
</hibernate-mapping>
```

❶ 继承层次结构的根类BillingDetails被映射到表BILLING_DETAILS。

❷ 必须添加一个特殊的列以便在持久化类之间进行区分：辨别标志（discriminator）。这不是持久化类的属性，由Hibernate内部使用。列名为BILLING_DETAILS_TYPE，值为字符串——在这个例子中，为“CC”或者“BA”。Hibernate自动设置和获取辨别标志的值。

❸ 同往常一样，用一个简单的<property>元素映射超类的属性。

❹ 每个子类都有自己的<subclass>元素。子类的属性被映射到BILLING_DETAILS表中的列。记住不允许NOT NULL约束，因为BankAccount实例不会有expMonth属性，且该行的CC_EXP_MONTH字段必须为NULL。

<subclass>元素可以依次包含其他被嵌套的<subclass>元素，直到整个层次结构被映射到表中。

Hibernate在查询BillingDetails类时生成下列SQL：

```
select
    BILLING_DETAILS_ID, BILLING_DETAILS_TYPE, OWNER,
    CC_NUMBER, CC_EXP_MONTH, ..., BA_ACCOUNT, BA_BANKNAME, ...
from BILLING_DETAILS
```

为了查询CreditCard子类，Hibernate在辨别标志列中添加一个限制：

```
select BILLING_DETAILS_ID, OWNER, CC_NUMBER, CC_EXP_MONTH, ...
from BILLING_DETAILS
where BILLING_DETAILS_TYPE='CC'
```

JPA中也有这个映射策略，即SINGLE_TABLE：

```
@Entity
@Inheritance(strategy = InheritanceType.SINGLE_TABLE)
@DiscriminatorColumn(
    name = "BILLING_DETAILS_TYPE",
    discriminatorType = DiscriminatorType.STRING
)
public abstract class BillingDetails {

    @Id @GeneratedValue
    @Column(name = "BILLING_DETAILS_ID")
    private Long id = null;

    @Column(name = "OWNER", nullable = false)
    private String owner;

    ...
}
```

如果没有在超类中指定辨别标志列，它的名称就默认为DTYPE，且它的类型默认为字符串。继承层次结构中所有具体的类都可以有辨别标志值；在这个例子中，BillingDetails是抽象类，CreditCard则是具体的类：

```
@Entity
@DiscriminatorValue("CC")
public class CreditCard extends BillingDetails {

    @Column(name = "CC_NUMBER")
    private String number;

    ...
}
```

没有显式的辨别标志值时，如果使用Hibernate XML文件，Hibernate就默认为完全限定的类名；如果使用注解或者JPA XML文件，则默认为实体名称。注意，在Java Persistence中，没有给非字符串辨别标志类型指定默认值，每个持久化提供程序都可以有不同的默认值。

以下是JPA XML描述符中与之相当的映射：

```
<entity-mappings>
    <entity class="auction.model.BillingDetails" access="FIELD">
        <inheritance strategy="SINGLE_TABLE"/>
        <discriminator-column name="BILLING_DETAILS_TYPE"
            discriminator-type="STRING"/>
```

```
        ...
    </entity>

    <entity class="auction.model.CreditCard" access="FIELD">
        <discriminator-value>CC</discriminator-value>
        ...
    </entity>

</entity-mappings>
```

有时候，尤其在遗留的Schema中，你无权在实体表中包括一个额外的辨别标志列。此时，可以应用一个formula来计算每一行的辨别标志值：

```
<discriminator
  formula="case when CC_NUMBER is not null then 'CC' else 'BA' end"
  type="string"/>

  ...

<subclass
    name="CreditCard"
    discriminator-value="CC">
  ...
```

这个映射依赖于SQL CASE/WHEN表达式，来决定一个特定的行是表示信用卡还是银行账号（许多开发人员从来不用这种SQL表达式；如果你不熟悉它的话请查阅ANSI标准）。表达式的结果是文字，CC或者BA，依次在<subclass>映射中声明。用于辨别的公式不是JPA规范的一部分，但是可以应用Hibernate注解：

```
@Entity
@Inheritance(strategy = InheritanceType.SINGLE_TABLE)
@org.hibernate.annotations.DiscriminatorFormula(
    "case when CC_NUMBER is not null then 'CC' else 'BA' end"
)
public abstract class BillingDetails {
    ...
}
```

每个类层次结构一张表的策略的缺点对于你的设计来说可能太严重了——毕竟，反规范化的Schema长期而言会变成一个重大的负担。你的数据库管理员根本不可能喜欢它。下一个继承映射策略则不会给你造成这个问题。

5.1.4　每个子类一张表

第四种方案是把继承关系表示为相关的外键关联。声明持久化属性的每个类/子类（包括抽象类甚至接口）都有它自己的表。

不同于我们最先映射的每个具体类一张表的策略，此处的表仅仅包含了每个非继承的属性（由子类本身声明的每个属性）以及也是超类表的外键的主键的列。这种方法如图5-3所示。

如果CreditCard子类的一个实例变成持久化，由BillingDetails超类声明的属性值就被持久化到BILLING_DETAILS表的一个新行。只有子类声明的属性值被持久化到CREDIT_CARD表的一个新行。这两行通过它们的共享主键值链接在一起。随后，通过联结子类表与超类表，可以从数

据库中获取子类实例。

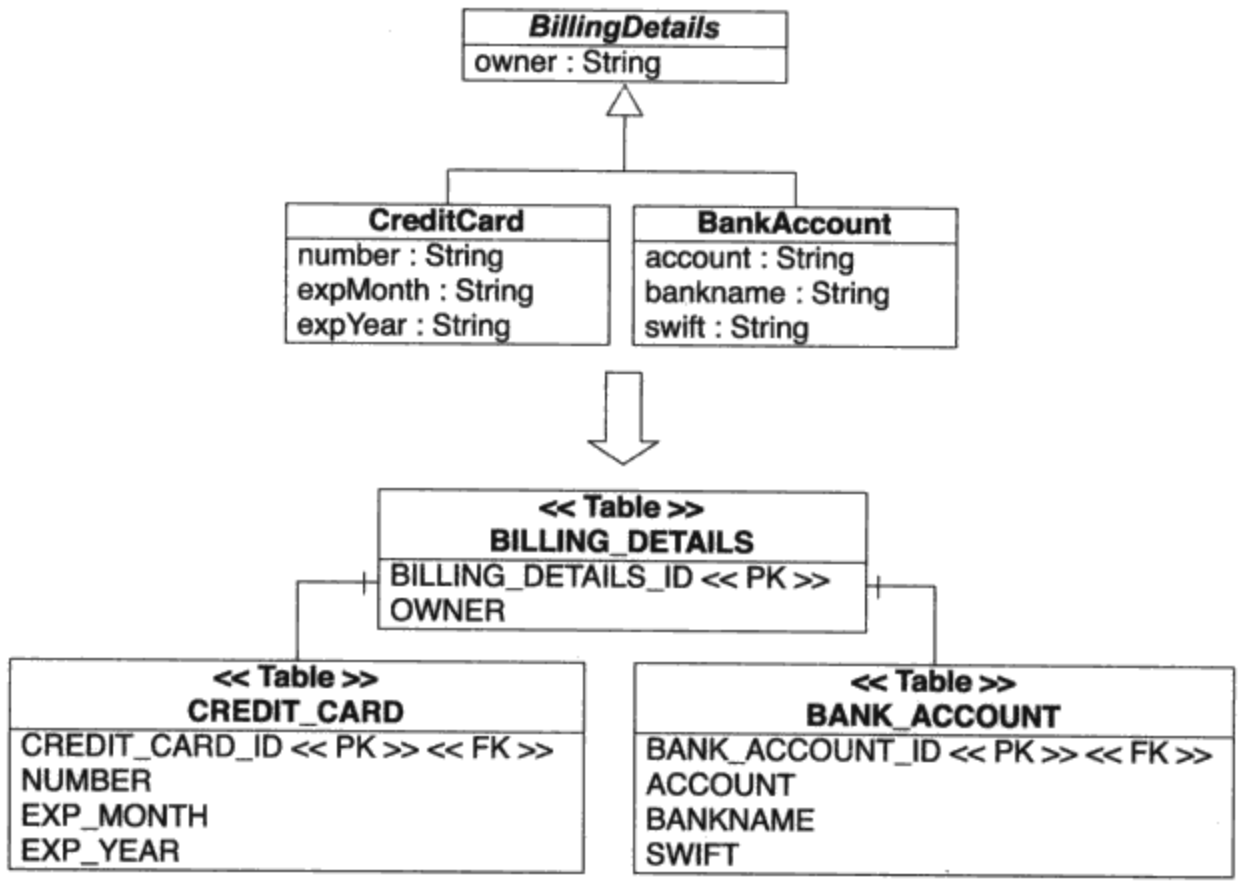


图5-3 把层次结构的所有类映射到它们自己的表

这一策略的主要好处在于，SQL Schema被标准化了。Schema演变和完整性约束定义很简单。对一个特定子类的多态关联可能被表示为引用这个特定子类的表的一个外键。

在Hibernate中，用<joined-subclass>元素给每个子类映射创建一张表。请见代码清单5-3。

代码清单5-3 Hibernate <joined-subclass>映射

```
<hibernate-mapping>
    <class                    ❶
        name="BillingDetails"
        table="BILLING_DETAILS">

        <id
            name="id"
            column="BILLING_DETAILS_ID"
            type="long">
            <generator class="native"/>
                  </id>

                  <property
                      name="owner"
                      column="OWNER"
                      type="string"/>

                      ...

                  <joined-subclass    ❷
```

```
            name="CreditCard"
            table="CREDIT_CARD">

            <key column="CREDIT_CARD_ID"/>

            <property name="number" column="NUMBER"/>
            <property name="expMonth" column="EXP_MONTH"/>
            <property name="expYear" column="EXP_YEAR"/>

        </joined-subclass>

        <joined-subclass
            name="BankAccount"
            table="BANK_ACCOUNT">
        ...

    </class>
</hibernate-mapping>
```

❶ 根类BillingDetails被映射到表BILLING_DETAILS。注意这个策略不需要辨别标志。

❷ 新的<joined-subclass>元素把子类映射到新的表——在这个例子中为CREDIT_CARD。在被联结的子类中声明的所有属性都被映射到这张表。

❸ CREDIT_CARD表需要主键。这个列也有一个对BILLING_DEAILS表的主键的外键约束。CreditCard对象查找需要这两张表的联结。<joined-subclass>元素可能包含其他嵌套的<joined-subclass>元素，直到整个层次结构已经被映射。

当查询BillingDetails类时，Hibernate依赖一个外部联结：

```
select BD.BILLING_DETAILS_ID, BD.OWNER,
       CC.NUMBER, CC.EXP_MONTH, ..., BA.ACCOUNT, BA.BANKNAME, ...
    case
        when CC.CREDIT_CARD_ID is not null then 1
        when BA.BANK_ACCOUNT_ID is not null then 2
        when BD.BILLING_DETAILS_ID is not null then 0
    end as CLAZZ_
from BILLING_DETAILS BD
    left join CREDIT_CARD CC
      on BD.BILLING_DETAILS_ID = CC.CREDIT_CARD_ID
    left join BANK_ACCOUNT BA
      on BD.BILLING_DETAILS_ID = BA.BANK_ACCOUNT_ID
```

SQL CASE语句在子类表CREDIT_CARD和BANK_ACCOUNT中侦测行的存在（或者不存在），以便Hibernate可以给BILLING_DETAILS表的一个特定行决定具体的子类。

为了缩小对子类的查询，Hibernate使用了内部联结：

```
select BD.BILLING_DETAILS_ID, BD.OWNER, CC.NUMBER, ...
from CREDIT_CARD CC
    inner join BILLING_DETAILS BD
      on BD.BILLING_DETAILS_ID = CC.CREDIT_CARD_ID
```

如你所见，这个映射策略更难以手工实现——甚至生成特殊的报告也更加复杂。如果计划把Hibernate代码和手写的SQL混合，这是一个需要考虑的重要事项。

此外，虽然这个映射策略似乎很简单，但依我们的经验，该性能无法为复杂的类层次结构所

接受。查询始终需要跨多张表的联结，或者多个连续读取。

用相同的策略和注解映射层次结构，此处称作JOINED策略：

```
@Entity
@Inheritance(strategy = InheritanceType.JOINED)
public abstract class BillingDetails {

    @Id @GeneratedValue
    @Column(name = "BILLING_DETAILS_ID")
    private Long id = null;

    ...
}
```

在子类中，如果子类表的主键列有着（或者将有）与超类表的主键列相同的名称时，就不需要指定联结列：

```
@Entity
public class BankAccount extends BillingDetails {
    ...
}
```

这个实体没有标识符属性；它自动从超类继承BILLING_DETAILS_ID属性和列，如果想要获取BankAccount的实例，Hibernate就知道如何把表联结到一起。当然，可以显式地指定列名：

```
@Entity
@PrimaryKeyJoinColumn(name = "CREDIT_CARD_ID")
public class CreditCard extends BillingDetails{
    ...
}
```

最后，下面是在JPA XML描述符中与之相当的映射：

```
<entity-mappings>

    <entity class="auction.model.BillingDetails" access="FIELD">
        <inheritance strategy="JOINED"/>
        ...
    </entity>

    <entity class="auction.model.BankAccount" access="FIELD"/>
    <entity class="auction.model.CreditCard" access="FIELD">
        <primary-key-join-column name="CREDIT_CARD_ID"/>
    </entity>

</entity-mappings>
```

在介绍何时选择何种策略之前，先思考在单个类层次结构中混合继承映射策略。

5.1.5 混合继承策略

通过嵌套<union-subclass>、<subclass>和<joined-subclass>映射元素可以映射整个继承层次结构。你无法把它们混合起来——例如，用一个辨别标志从“每个类层次结构一张表”策略转换到标准的“每个子类一张表”策略。一旦决定使用某种继承策略，就必须坚持使用它。

但这并不完全正确。使用一些Hibernate技巧，可以给一个特定的子类切换映射策略。例如，

可以把一个类层次结构映射到单张表，但是对于特定的子类，则通过外键映射策略切换到单独的表，就像使用每个子类一张表一样。使用<join>映射元素，这种转换是可能的：

```
<hibernate-mapping>
<class name="BillingDetails"
       table="BILLING_DETAILS">

    <id>...</id>

    <discriminator
        column="BILLING_DETAILS_TYPE"
        type="string"/>
        ...

    <subclass
        name="CreditCard"
        discriminator-value="CC">

        <join table="CREDIT_CARD">
            <key column="CREDIT_CARD_ID"/>

            <property name="number" column="CC_NUMBER"/>
            <property name="expMonth" column="CC_EXP_MONTH"/>
            <property name="expYear" column="CC_EXP_YEAR"/>
            ...
        </join>

    </subclass>

    <subclass
        name="BankAccount"
        discriminator-value="BA">

        <property name=account" column="BA_ACCOUNT"/>
        ...
    </subclass>

...

</class>
</hibernate-mapping>
```

<join>元素集合了一些属性，并告诉Hibernate到一个二级表中获取它们。这个映射元素有多种用途，你会在本书稍后再次见到它。在这个例子中，它把CreditCard属性从每个层次结构一张表中分离出来，放到CREDIT_CARD表中。这张表的CREDIT_CARD_ID列同时也是主键，并且它有引用该层次结构表的BILLING_DETAILS_ID的外键约束。BankAccount子类被映射到层次结构表。看看图5-4中的Schema。

运行时，Hibernate执行一个外部联结，多态地抓取BillingDetails和所有的子类实例：

```
select
    BILLING_DETAILS_ID, BILLING_DETAILS_TYPE, OWNER,
    CC.CC_NUMBER, CC.CC_EXP_MONTH, CC.CC_EXP_YEAR,
    BA_ACCOUNT, BA_BANKNAME, BA_SWIFT

from
    BILLING_DETAILS
left outer join
```

```
CREDIT_CARD CC
    on BILLING_DETAILS_ID = CC.CREDIT_CARD_ID
```

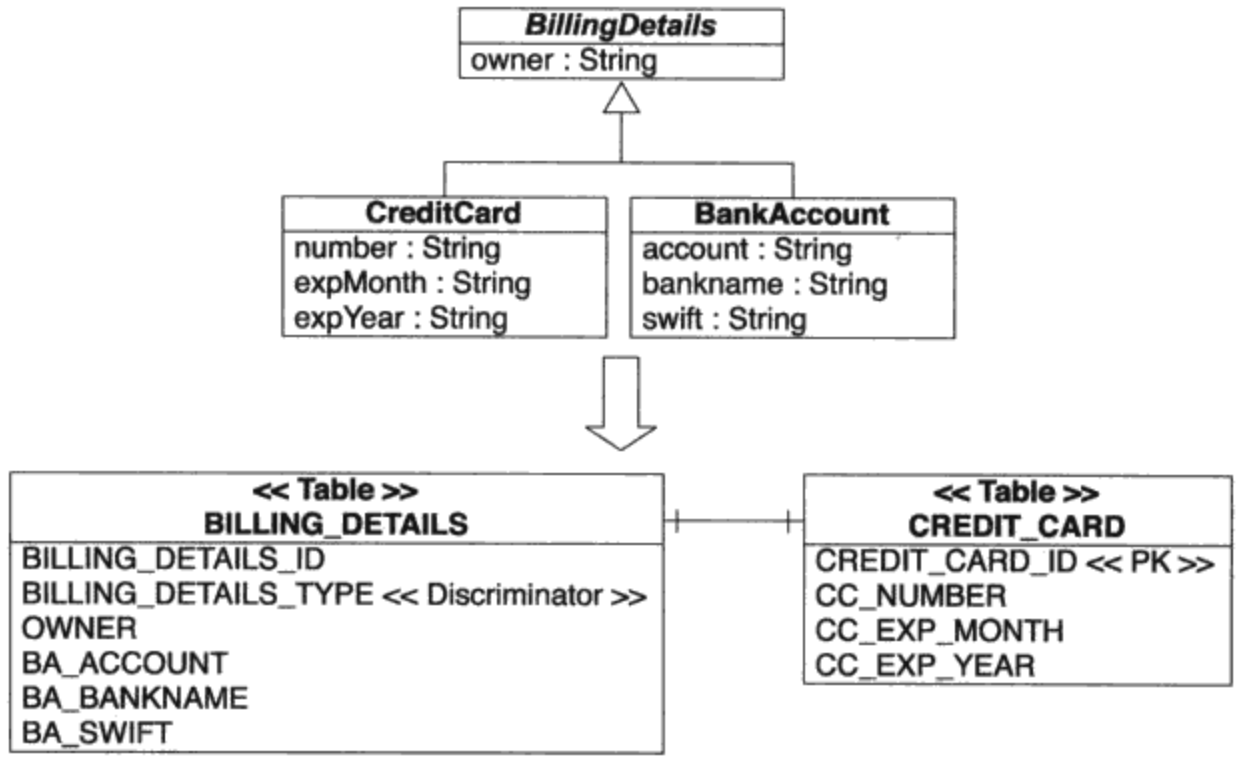


图5-4 把一个子类分到它自己的二级表中

也可以在类层次结构中对其他的子类使用<join>技巧。然而，如果你有一个宽到出乎意料的类层次结构，外部联结就可能成问题。有些数据库系统（例如Oracle）限制外部联结操作中的表数目。对于宽层次结构来说，你可能想要切换到立即执行第二选择而不是外部联结的不同抓取策略：

```
<subclass
    name="CreditCard"
    discriminator-value="CC">

    <join table="CREDIT_CARD" fetch="select">
        <key column="CREDIT_CARD_ID"/>
        ...
    </join>

</subclass>
```

Java Persistence也通过注解支持这种混合的继承映射策略。就像前面所做的那样，用InheritanceType.SINGLE_TABLE映射超类BillingDetails。现在映射你想要从单张表分到一张二级表的子类。

```
@Entity
@DiscriminatorValue("CC")
@SecondaryTable(
  name = "CREDIT_CARD",
  pkJoinColumns =  @PrimaryKeyJoinColumn(name = "CREDIT_CARD_ID")
)
public class CreditCard extends BillingDetails {

    @Column(table = "CREDIT_CARD",
```

```
                name = "CC_NUMBER",
                nullable = false)
    private String number;
    ...
}
```

如果没有给二级表指定主键联结列，就使用单张继承表的主键名称——在这个例子中为BILLING_DETAILS_ID。还要注意，你要用二级表的名称映射被移到二级表里面的所有属性。

你还想要更多关于如何给应用程序的类层次结构选择合适的映射策略组合的技巧。

5.1.6　选择策略

可以把所有的映射策略应用到抽象类和接口。接口可能没有状态，但是可能包含访问方法声明，因此可以像抽象类一样地处理它们。可以用<class>、<union-subclass>、<subclass>或者<joined-subclass>映射接口，并可以用<property>映射任何被声明或者被继承的属性。Hibernate不会试图实例化抽象类，即使你查询或者加载了它。

说明　注意JPA规范不在接口上支持任何映射注解！这在规范的未来版本中会得到解决；当你读到本书时，或许可能在使用Hibernate Annotations了。

以下是一些经验法则：

- 如果你不需要多态关联或者查询，就倾向于每个具体类一张表——换句话说，如果你从不或者很少查询BillingDetails，并且没有关联BillingDetails的类（我们的模型有）。基于UNION的显式映射应该是首选，因为随后（最优化的）多态查询和关联将成为可能。隐式多态对于利用非持久化相关的接口的查询最有用。
- 如果你一定要多态关联（对超类的关联，及由此在运行时通过具体类的动态解析对层次结构中的所有类的关联）或者查询，并且子类相对地声明几种属性（尤其当子类之间的主要区别在于它们的行为中）时，则倾向于每个类层次结构一张表。你的目标是把可为空的列数减到最少，并让你自己（和数据库管理员）确信反规范化的Schema在长期运行中不会产生问题。
- 如果你一定需要多态的关联或者查询，并且子类声明多个属性（子类的不同主要在于它们所持有的数据），则倾向每个子类一张表。或者，根据继承层次结构的宽度和深度，以及相对于联合的可能联结成本，使用每个具体类一张表。

默认情况下，仅对简单的问题选择每个类层次结构一张表。对于更复杂的案例（或者当你被一位坚持认为可为空的能力约束和标准化很重要的数据建模者否决的时候），就应该考虑每个子类一张表的策略。但这时候要问问自己：在对象模型中，重新把继承建模为委托是否未必更好。对于无关持久化或者ORM的各种原因，通常最好避免复杂的继承。Hibernate充当着领域和关系模型之间的缓冲区，但这并不意味着在设计类的时候，可以忽略持久化关注点。

当你开始考虑混合的继承策略时，记住：Hibernate中的隐式多态很聪明，足以处理更多异乎

寻常的情况。例如，考虑在应用程序中增加一个接口ElectronicPaymentOption。这是一个没有持久化方面的业务接口——除了在我们的应用程序中，例如CreditCard这样的持久化类将可能实现这个接口之外。无论你如何映射BillingDetails层次结构，Hibernate都可以正确地给from ElectronicPaymentOption回复查询。甚至不是BillingDetails层次结构一部分的其他类也可以，它们被映射为持久化，并实现这个接口。Hibernate始终知道要查询哪些表、构造哪些实例，以及如何返回一个多态的结果。

最后，也可以在单个的映射文件中使用<union-subclass>、<subclass>和<joined-subclass>映射元素（作为一级元素，而不是<class>）。然后必须声明被扩展的类，例如<subclassname="CreditCard" extends="BillingDetails">，且必须在子类映射文件之前程序化地加载基类映射（在XML配置文件中列出映射资源清单时，不必担心这个顺序）。这种技术允许扩展类层次结构，而不用修改超类的映射文件。

现在知道了有关实体、属性和继承层次结构的映射所需要知道的一切。已经可以映射复杂的领域模型了。本章后半部分讨论Hibernate用户应该熟记的另一项重要的特性：Hibernate映射类型系统。

5.2 Hibernate 类型系统

第4章首先区分了实体和值类型——Java中ORM的一个中心概念。必须详细阐述这种区别，以便你完全理解Hibernate中实体、值类型和映射类型的类型系统。

5.2.1 概述实体和值类型

实体是系统中粗粒度的类。通常按照涉及的实体定义系统的特性。用户给货品出价（the user places a bid for an item）是一个典型的特性定义；它提到了三个实体。值类型的类甚至经常不出现在业务需求中——它们通常是表示字符串、数字和货币金额的细粒度类。值类型也偶尔出现在特性定义中：用户改变账单地址（the user changes billing address）是一个例子，假设Address是一个值类型。

更正式一点的定义：实体是指其实例具有自己的持久化同一性的任何类。值类型是指没有定义某种持久化同一性的类。在实际应用中，这意味着实体类型是包含标识符属性的类，值类型的类则取决于实体。

运行时，你有一个实体实例的网络与值类型实例交错。实体实例可能处于这三种持久化生命周期状态之一：瞬时（transient）、脱管（detached）或者持久化（persistent）。我们不考虑把这些生命周期状态应用到值类型实例。（第9章再回到对象状态的讨论。）

因此，实体有自己的生命周期。Hibernate Session接口的Save()和delete()方法应用到实体类的实例，而从不应用到值类型实例。值类型实例的持久化生命周期完全由自己的实体实例的生命周期所决定。例如，保存用户时，用户名变成持久化；它从来不会独立于用户而变成持久化。

在Hibernate中，值类型可以定义关联；可能从一个值类型实例导航到一些其他的实体。但是永远不可能从其他实体导航回到值类型实例。关联始终指向实体。这意味着，当从数据库中获取值类型实例时，它完全为一个实体所有；从不共享。

在数据库级中，任何表都被当作实体。然而Hibernate提供某些构造，从Java代码中隐藏数据库级实体的存在。例如，多对多（many-to-many）的关联映射从应用程序中隐藏了中间关联表。从应用程序的观点来看，字符串的集合（更准确地说，是值类型实例的集合）表现得就像一个值类型；但它被映射到它自己的表。虽然这些特性乍看起来不错（它们简化了Java代码），但随着时间的推移，我们对此产生了怀疑。不可避免地，这些被隐藏的实体最终要随着业务需求的演变而公开给应用程序。例如，随着应用程序的成熟，多对多的关联表经常会添加额外的表。我们几乎准备推荐让每个数据库级的实体都被当作实体类公开给应用程序。例如，我们会倾向于把多对多的关联建模为对中间实体类的两个一对多的关联。但是，我们把最终的决定权留给你，并在后面的章节中回到多对多的实体关联的话题。

实体类始终利用<class>、<union-subclass>、<subclass>和<joined-subclass>映射元素被映射到数据库。值类型又如何被映射呢？

你已经见过了两种不同的值类型映射：<property>和<component>。组件（component）的值类型很明显：它是被映射为可嵌入的类。然而，属性（property）的类型是个更为一般的概念。考虑CaveatEmptor User和email地址的这个映射：

```
<property name="email"
          column="EMAIL"
          type="string"/>
```

让我们关注type="string"属性。你知道，在ORM中必须处理Java类型和SQL数据类型。这两个不同的类型系统必须被桥接起来。这是Hibernate映射类型的任务，string是内建的Hibernate映射类型的名称。

string不是Hibernate中内建的唯一映射类型。Hibernate带来了各种映射类型，它们给基本的Java类型和某些JDK类定义了默认的持久化策略。

5.2.2　内建的映射类型

Hibernate内建的映射类型通常共享它们映射的Java类型的名称。然而，一个特定的Java类型可能有不止一个Hibernate映射类型。

内建的类型不能用于执行任意的转化，例如把一个VARCHAR数据库值映射到一个Java的Integer属性值。可以给这种转化定义自己的定制值类型，如本章稍后所述。

现在讨论基本的要素：日期和时间、定位对象，以及各种其他内建的映射类型，并介绍它们处理哪些Java和SQL数据类型。

1．Java基本的映射类型

表5-1中基础的映射类型把Java基本的类型（或者它们的包装类型）映射到适当的内建SQL标准类型。

表5-1 基本的类型

映射类型	Java类型	标准的SQL内建类型
integer	int或java.lang.Integer	INTEGER
long	long或java.lang.Long	BIGINT
short	short或java.lang.Short	SMALLINT
float	float或java.lang.Float	FLOAT
double	double或java.lang.Double	DOUBLE
big_decimal	java.math.BigDecimal	NUMERIC
character	java.lang.String	CHAR(1)
string	java.lang.String	VARCHAR
byte	byte或java.lang.Byte	TINYINT
boolean	boolean或java.lang.Boolean	BIT
yes_no	boolean或java.lang.Boolean	CHAR(1) ('Y' or 'N')
true_false	boolean或java.lang.Boolean	CHAR(1) ('T' or 'F')

你可能已经注意到你的数据库不支持表5-1中提到的有些SQL类型。上面所列的名称为ANSI标准数据类型的名称。大部分数据库供应商忽略这部分SQL标准（因为他们的遗留类型系统经常把标准上的日期填早了）。然而，JDBC驱动程序提供特定于供应商的SQL数据类型的部分抽象，允许Hibernate在执行DML时使用ANSI标准类型。对于数据库专用的DDL生成，Hibernate利用对特定的SQL方言的内建支持，把ANSI标准类型转变为适当的特定于供应商的类型。（这意味着如果你正在使用Hibernate访问数据和定义SQL Schema，通常不必担心SQL数据类型。）

此外，Hibernate类型系统很聪明，可以根据值定义的长度（length）转换SQL数据类型。最明显的案例是string：如果用length属性声明字符串属性映射，Hibernate会根据选中的方言挑选正确的SQL数据类型。例如，对于MySQL，当Hibernate导出Schema时，长度达到65535时产生一个一般的VARCHAR（length）列。长度达到16777215时，使用MEDIUMTEXT的数据类型。较大的字符串映射则导致使用LONGTEXT。如果想知道这种或者其他映射类型的范围，请检查你的SQL方言（与Hibernate一起带来的源代码）。可以通过把方言作为子类，并覆盖这些设置来定制这种行为。

大部分方言也支持设置十进制SQL数据类型的比例和精度。例如，在BigDecimal映射中设置precision或者scale，给MySQL创建了一个NUMERIC（precision，scale）的数据类型。

最后，yes_no和true_false映射类型是遗留的Schema和Oracle用户最有用的转换器；Oracle DBMS产品没有内建的布尔（boolean）或者真值（truth-valued）类型（关系数据模型唯一真正需要的内建数据类型）。

2. 日期和时间映射类型

表5-2列出了与日期、时间和时间戳关联的Hibernate类型。在领域模型中，可以选择使用java.util.Date、java.util.Calendar，或者在java.sql包中定义的java.util.Date的子类，来表示日期和时间数据。这是个人偏爱的问题，我们留给你自己去决定——但是你要确保一致。（在实际应用中，绑定领域模型到来自JDBC包的类型，并非是最好的办法。）

表5-2　日期和时间类型

映射类型	Java类型	标准的SQL内建类型
date	java.util.Date或java.sql.Date	DATE
time	java.util.Date或java.sql.Time	TIME
timestamp	java.util.Date或java.sql.Timestamp	TIMESTAMP
calendar	java.util.Calendar	TIMESTAMP
calendar_date	java.util.Calendar	DATE

警告：如果用timestamp（最常见的情况）映射java.util.Date属性，Hibernate就在从数据库中加载属性时返回java.sql.Timestamp。Hibernate必须使用JDBC子类，因为它包括可能出现在数据库中的十亿分之一秒的信息。Hibernate无法只切断这个信息。如果试图把java.util.Date属性与equals()方法进行比较，就会出现问题，因为它与java.sql.Timestamp子类equals()方法不同步。首先，比较两个java.util.Date对象的正确方法（无论如何），对于任何子类也适用的，是aDate.getTime() > bDate.getTime()（用于大于的比较）。其次，可以编写一个切断数据库十亿分之一秒信息的定制映射类型，并在任何情形下都返回java.util.Date。目前（虽然这在未来可能改变），Hibernate中还没有内建这样的映射类型。

3．二进制和大值映射类型

表5-3列出了用来处理十进制数据和大值的Hibernate类型。注意只有binary作为标识符属性的类型得到支持。

表5-3　二进制和大值类型

映射类型	Java类型	标准的SQL内建类型
binary	byte[]	VARBINARY
text	java.lang.String	CLOB
clob	java.sql.Clob	CLOB
blob	java.sql.Blob	BLOB
serializable	任何实现java.io.Serializable的Java类	VARBINARY

如果持久化的Java类中的属性是byte[]类型，Hibernate就可以用二进制映射类型把它映射到一个VARBINARY列。（注意真正的SQL类型取决于方言。例如，在PostgreSQL中，SQL类型是BYTEA；在Oracle中，SQL类型是RAW。）如果持久化的Java类中的属性是java.lang.String类型，Hibernate就会用text映射类型把它映射到SQL CLOB列。

注意在这两个案例中，当加载保存着属性变量的实体实例时，Hibernate立即就初始化属性值。当你必须处理潜在的大值时，这样很不方便。

一种解决方案是按需要通过字段访问的拦截延迟加载。但是对于额外代码的注入，这种方法需要持久化类的字节码基础设施（ByteCode Instrumentation，BCI）。13.1.6节将讨论通过BCI和拦截的延迟加载。

第二种解决方案是Java类中一种不同的属性。JDBC直接支持定位器对象（Locator Object，

LOB)[①]。如果你的Java属性是java.sql.Clob或者java.sql.Blob类型，就可以用clob或者blob映射类型映射它，以便不通过BCI而获得大值的延迟加载。当加载属性的所有者时，属性值就是一个定位对象——实际上，它是一个指向了还没有被物化的真实值的指针。一旦访问了属性，值就被物化了。这种按需加载只在数据库事务打开时才起作用，因此当自己的实体实例处于持久化和事务状态，而不是处于脱管状态时，就可以访问这种类型的任何属性。现在领域模型也被绑定到JDBC了，因为需要导入java.sql包。虽然领域模型类在单独的单元测试中是可执行的，但是没有数据库连接则无法访问LOB属性。

如果依赖Java Persistence注解，包含潜在大值的映射属性则稍有不同。默认情况下，类型java.lang.String的一个属性被映射到SQL VARCHAR列（或者相当的列，取决于SQL方言）。如果想要把java.lang.String、char[]、Character[]，或者甚至java.sql.Clob类型属性映射到CLOB列，就要用@Lob注解映射它：

```
@Lob
@Column(name = "ITEM_DESCRIPTION")
private String description;
@Lob
@Column(name = "ITEM_IMAGE")
private byte[] image;
```

对于byte[]、Byte[]或者java.sql.Blob类型的任何属性也一样。注意在任何情况下，除了java.sql.Clob或者java.sql.Blob类型的属性之外，该值都会立即由Hibernate再次加载，而不是按需延迟加载。包含拦截代码的字节码再次成为透明地启用单独属性的延迟加载的一种选项。

如果领域模型中有这些属性类型，为了创建和设置java.sql.Blob或者java.sql.Clob值，就用静态的Hibernate.createBlob()和Hibernate.createClob()方法，并提供字节数组、输入流或者字符串。

最后，注意Hibernate和JPA都对Serializable（可序列化）的任何属性类型提供序列回滚。这种映射类型把属性的值转换为随后保存在VARBINARY（或者相当的）列中的字节流。当属性的所有者被加载时，属性值是反序列化的。自然地，使用这种策略要非常谨慎（数据存在得比应用程序更长久），并且它可能只对临时数据（用户偏爱、登录会话数据等）有用。

4．JDK映射类型

表5-4为JDK的其他Java类型列出了Hibernate类型，它们可以被表示为数据库中的VARCHAR。你可能已经注意到<property>不是唯一有type属性的Hibernate映射元素。

表5-4 其他JDK相关的类型

映射类型	Java类型	标准的SQL内建类型
class	java.lang.Class	VARCHAR
locale	java.util.Locale	VARCHAR
timezone	java.util.TimeZone	VARCHAR
currency	java.util.Currency	VARCHAR

① 据Jim Starkey（提出LOB概念的人）所说，术语BLOB和CLOB并没有任何意义，只是由营销部门创造出来的。可以用喜欢的任何方式来解释它们。我们更喜欢定位器对象（locator object），暗指它们的作用就像指针一样。

5.2.3　使用映射类型

所有的基础映射类型都可能出现在Hibernate映射文档、一般的属性、标识符属性和其他映射元素中的几乎任何地方。<id>、<property>、<version>、<discriminator>、<index>和<element>元素都定义了具名type的属性。

你会看到，在这个映射中，内建的映射类型对于BillingDetails类是多么有用：

```
<class name="BillingDetails" table="BILLING_DETAILS">
    <id name="id" type="long" column="BILLING_DETAILS_ID">
        <generator class="native"/>
    </id>
    <discriminator type="character" column="BILLING_DETAILS_TYPE"/>
    <property name="number" type="string"/>
    ....
</class>
```

BillingDetails类被映射为实体。它的discriminator、identifier和name属性都为值类型，并且我们使用内建的Hibernate映射类型指定变换策略。

在XML映射文档中，通常不需要显式地指定内建的映射类型。例如，如果你有Java类型java.lang.String的一个属性，Hibernate就通过反射发现它，并默认地选择string。可以轻松地简化前一个映射示例：

```
<class name="BillingDetails" table="BILLING_DETAILS">
    <id name="id" column="BILLING_DETAILS_ID">
        <generator class="native"/>
    </id>
    <discriminator type="character" column="BILLING_DETAILS_TYPE"/>
    <property name="number"/>
    ....
</class>
```

Hibernate也理解type="java.lang.String"，因此它不必使用反射。这种方法不太适用的最重要案例是java.util.Date属性。默认情况下，Hibernate认为java.util.Date是一个timestamp映射。如果不希望日期和时间信息都持久化，就要显式地指定type="time"或者type="date"。

利用JPA注解，自动侦测属性的映射类型，就像在Hibernate中一样。对于java.util.Date或者java.util.Calendar属性，Java Persistence标准要求用@Temporal注解选择精确度：

```
@Temporal(TemporalType.TIMESTAMP)
@Column(nullable = false, updatable = false)
private Date startDate;
```

另一方面，Hibernate Annotations放松了标准的规则，默认为TemporalType.TIMESTAMP——选项是TemporalType.TIME和TemporalType.DATE。

在其他极少数情况下，可能要添加@org.hibernate.annotations.Type注解到一个属性，并显式声明一个内建或者定制的Hibernate映射类型的名称。一旦开始编写自己的定制映射类型，这就立即成为一个更为常见的扩展了，本章稍后就会实践到。

与之相当的JPA XML描述符如下：

```
<entity class="auction.model.Item" access="FIELD">
    <attributes>
        ...
        <basic name="startDate">
            <column nullable="false" updatable="false"/>
            <temporal>TIMESTAMP</temporal>
        </basic>
    </attributes>
</entity>
```

对于每个内建的映射类型，都由类org.hibernate.Hibernate定义常量。例如，Hibernate.STRING表示string映射类型。这些常量对于查询参数绑定很有用，如第14章和第15章中所深入讨论的：

```
session.createQuery("from Item i where i.description like :desc")
    .setParameter("desc", d, Hibernate.STRING)
    .list();
```

注意，也可以在这个例子中用setString()实参绑定方法。类型常量对于Hibernate映射元模型的编程式操作也很有用，如第3章所述。

Hibernate不受限于内建的映射类型。把可扩展的映射类型系统当作使Hibernate变得如此灵活的核心特性之一和重要方面。

5.3 创建定制的映射类型

面向对象的语言比如Java，使得通过编写新类来定义新的类型变得很容易。这是面向对象的定义的一个基础部分。当声明持久化类的属性时，如果受限于预设的内建Hibernate映射类型，就会失去一些Java的表现力。此外，我们的领域模型实现就会与物理的数据模型紧藕合了，因为新的类型转化是不可能的。

我们已经见过的大部分ORM解决方案都支持以用户自定义的策略来执行类型转化。这些通常称作转换器。例如，用户可以创建一种新的策略，用来把JDK类型Integer的一个属性持久化到VARCHAR列。Hibernate提供一种类似的、更强大的、称作定制映射类型的特性。

你首先要理解什么时候适合编写自己的定制映射类型，以及哪种Hibernate扩展点（extension point）与你相关。然后编写一些定制的映射类型并探讨选项。

5.3.1 考虑定制的映射类型

拿前面章节的Address类的映射作为组件来举例：

```
<component name="homeAddress" class="Address">

    <property name="street" type="string" column="HOME_STREET"/>
    <property name="city" type="string"   column="HOME_CITY"/>
    <property name="zipcode" type="string" column="HOME_ZIPCODE"/>

</component>
```

这个值类型映射很简单；用户自定义的新Java类型的所有属性都被映射到内建的SQL数据类型的单独的列。然而，也可以选择利用一个定制的映射类型把它当作一个简单的属性来映射：

```
<property name="homeAddress"
          type="auction.persistence.CustomAddressType">

    <column name="HOME_STREET"/>
    <column name="HOME_CITY"/>
    <column name="HOME_ZIPCODE"/>

</property>
```

这也可能是你第一次见到单个<property>元素内部嵌有几个<column>元素。我们正把Address值类型（它甚至被随处命名）和三个指定列之间转变和转换的任务转移给一个单独的类auction.persistence.CustomAddressType。现在这个类负责加载和保存这个属性。注意在领域模型实现中，Java代码都没有改变——homeAddress属性是Address类型。

当然，在这个例子中，使用定制的映射类型替换组件映射的好处是令人置疑的。只要在加载和保存这个对象时不需要特殊的转化，现在必须编写的CustomAddressType就是额外的工作。然而，你可能已经看到定制的映射类型提供了一个额外的缓冲区——当在长期运行中需要额外的转化时，可能派得上用场的东西。当然，定制的映射类型还有更好的使用案例，你很快就会看到。（在Hibernate社区网站上可以找到许多有用的Hibernate映射类型的例子。）

来看看用于创建定制的映射类型的Hibernate扩展点。

5.3.2　扩展点

Hibernate提供定义定制的映射类型时应用程序可能使用的几个接口。这些接口减少了创建新映射类型所涉及的工作，并使定制的类型免受Hibernate核心变化的影响。这帮助你轻松地升级Hibernate，并保留现有的定制映射类型。

扩展点如下：

- org.hibernate.usertype.UserType——基础的扩展点，用于多种情况。它为定制值类型实例的加载和存储提供基础的方法。
- org.hibernate.usertype.CompositeUserType——包含比基础的UserType更多方法的一个接口，通常把有关值类型类的内部信息公开给Hibernate，例如单独的属性。然后可以在Hibernate查询中引用这些属性。
- org.hibernate.usertype.UserCollectionType——很少被用来实现定制集合的接口。实现这个接口的定制映射类型不是在属性映射中声明，而是只对定制的集合映射有用。如果想要持久化一个非JDK的集合，并持久保持额外的语义时，就必须实现这个类型。第6章将讨论集合映射和这个扩展点。
- org.hibernate.usertype.EnhanceUserType——扩展了UserType并提供额外方法的接口，这些方法用来把值类型封送到XML表示法（或者从XML表示法中封送值类型），或者启用一个定制的映射类型，在标识符和辨别标志映射中使用。

- org.hibernate.usertype.UserVersionType——扩展了UserType并提供额外方法的接口，这些方法启用用于实体版本映射的定制映射类型。
- org.hibernate.usertype.ParameterizedType——一个有用的接口，可以与所有其他的接口合并，来提供配置设置——也就是说，元数据中定义的参数。例如，可以根据映射中的一个参数，编写一个知道如何把值转变为欧元或者美元的单个MoneyConverter。

现在要创建一些定制的映射类型。不应该认为这是一个没有必要的练习，即使你对内建的Hibernate映射类型很满意。依我们的经验，每一个复杂的应用程序都会有定制映射类型的许多好的使用案例。

5.3.3 定制映射类型的案例

Bid类定义了一个amount属性，Item类定义了一个initialPrice属性；这两者都是货币值。目前为止，我们只用了一个简单的BigDecimal来表示值，通过big_decimal映射到单个NUMERIC列。

假设你要在拍卖应用程序中支持多种货币，并且必须对这个（客户驱动的）变化重构现有的领域模型。实现这一变化的一种方法是把新的属性添加给Bid、Item:amountCurrency以及initialPriceCurrency。然后可以用内建的currency映射类型，把这些新属性映射到额外的VARCHAR列。我们希望你永远不要使用这种方法！

相反，应该创建一个封装了货币和金额的新MonetaryAmount类。注意这是领域模型的一个类，不依赖任何Hibernate接口：

```
public class MonetaryAmount implements Serializable {

    private final BigDecimal amount;
    private final Currency currency;

    public MonetaryAmount(BigDecimal amount, Currency currency) {
        this.amount = amount;
        this.currency = currency;
    }

    public BigDecimal getAmount() { return amount; }

    public Currency getCurrency() { return currency; }
    public boolean equals(Object o) { ... }
    public int hashCode() { ...}

}
```

我们已经使MonetaryAmount成为了一个不可变的类。这在Java中是一个好的实践，因为它简化了代码。注意，必须实现equals()和hashCode()来完成类（此处没有什么特别的东西要考虑）。用这个新的MonetaryAmount取代Item中initialPrice属性的BigDecimal。可以并且应该把它用在任何持久化类中以及在业务逻辑中（例如，在账单系统中）所有其他的BigDecimal价格（例如Bid.amount）。

让我们通过这个新的MonetaryAmount类型，把Item的被重构的initialPrice属性映射到数据库。

5.3.4　创建 UserType

想象你正在使用一个用USD表示所有货币金额的遗留数据库。这个应用程序不再受限于单种货币(这是重构的重点)，但是它要花费数据库团队一些时间进行改变。当持久化MonetaryAmount对象时，需要把金额转化为USD。当从数据库加载时，把它转化回用户根据其偏爱所选择的货币类型。

创建一个实现Hibernate接口UserType的新MonetaryAmountUserType类。这是你的定制映射类型，如代码清单5-4所示。

代码清单5-4　给用USD为单位的货币金额定制的映射类型

```
public class MonetaryAmountUserType
        implements UserType {

    public int[] sqlTypes() {                                    ❶
        return new int[]{ Hibernate.BIG_DECIMAL.sqlType() };
    }

    public Class returnedClass() { return MonetaryAmount.class; }    ❷

    public boolean isMutable() { return false; }                 ❸

    public Object deepCopy(Object value) {  return value; }      ❹

    public Serializable disassemble(Object value)                ❺
        { return (Serializable) value; }

    public Object assemble(Serializable cached, Object owner)    ❻
        { return cached; }

    public Object replace(Object original,                       ❼
                          Object target,
                              Object owner)
        { return original; }

    public boolean equals(Object x, Object y) {                  ❽
        if (x == y) return true;
        if (x == null || y == null) return false;
        return x.equals(y);
    }

    public int hashCode(Object x) {
        return x.hashCode();
    }

    public Object nullSafeGet(ResultSet resultSet,               ❾
                              String[] names,
                              Object owner)
            throws SQLException {

        BigDecimal valueInUSD = resultSet.getBigDecimal(names[0]);
        // Deferred check after first read
        if (resultSet.wasNull()) return null;
        Currency userCurrency = User.getPreferences().getCurrency();
        MonetaryAmount amount = new MonetaryAmount(valueInUSD, "USD");
```

```
        return amount.convertTo(userCurrency);
    }

    public void nullSafeSet(PreparedStatement statement,                    ⑩
                            Object value,
                            int index)
            throws HibernateException, SQLException {

        if (value == null) {
          statement.setNull(index, Hibernate.BIG_DECIMAL.sqlType());
        } else {
            MonetaryAmount anyCurrency = (MonetaryAmount)value;
            MonetaryAmount amountInUSD =
              MonetaryAmount.convert( anyCurrency,
                                      Currency.getInstance("USD") );
            statement.setBigDecimal(index, amountInUSD.getAmount ());
        }
    }
}
```

❶ sqlTypes()方法告诉Hibernate要使用什么SQL列类型生成DDL模式。注意，这个方法返回类型代码的一个数组。UserType可以把单个属性映射到多个列，但是这个遗留的数据模型只有单个数字化的列。通过使用Hibernate.BIG_DECIMAL.sqlType()方法，你让Hibernate为指定的数据库方言确定正确的SQL数据类型。另一种方法是从java.sql.Types中返回常量。

❷ returnedClass()方法告诉Hibernate这个UserType映射哪些Java值类型。

❸ Hibernate可以对像这样的不可变类型进行一些微小的性能优化，例如，在脏检查期间比较快照的时候。isMutable()方法告诉Hibernate这个类型是不可变的。

❹ UserType首先也部分负责创建值的快照。因为MonetaryAmount是不可变的类，deepCopy()方法返回它的实参。至于可变的类型，则需要返回实参的一个副本，当作快照值使用。

❺ 当Hibernate把MonetaryAmount放进一个二级高速缓存时，调用disassemble()方法。就像稍后要介绍的，这是一种以序列化的形式保存信息的数据的高速缓存。

❻ assemble()方法与分解正好相反：它把被高速缓存的数据转变为MonetaryAmount的一个实例。如你所见，这两个子程序的实现对于不可变的类型都很容易。

❼ 实现replace()来处理脱管对象状态的合并。如本书稍后要介绍的，合并的过程涉及源对象和目标对象，它们的状态必须合并。对于不可变的值类型，再次返回第一个实参。对于可变的类型，至少返回第一个实参的深复制（deep copy）。对于带有组件字段的可变类型，你或许想要应用一个递归的合并子程序。

❽ UserType负责脏检查属性值。equals()方法把当前的属性值与之前的快照进行比较，确定属性是否为脏，并且必须被保存到数据库。两个相等的值类型实例的hashCode()必须相同。我们通常把这个方法委托给实际的值类型类——在这个例子中，是给定的MonetaryAmount对象的hashCode()方法。

❾ nullSafeGet()方法从JDBC的ResultSet获取属性值。如果要它用于转化，你也可以访问组件的所有者。所有数据库值都以USD为单位，因此把它转化为用户目前已经依其偏爱设置的

货币类型。（注意，实现这种转化和偏爱的处理取决于你。）

❿ nullSafeSet()方法把属性值写到JDBC的PreparedStatement。这个方法采用所设置的任何货币类型，并在保存之前把它转化为一个简单的BigDecimal USD金额。

现在映射Item的initialPrice属性如下：

```
<property name="initialPrice"
          column="INITIAL_PRICE"
          type="persistence.MonetaryAmountUserType"/>
```

注意你把定制的用户类型放进了persistence包；它是应用程序的持久层的一部分，而不是领域模型或者业务层的一部分。

为了在注解中使用定制类型，必须添加一个Hibernate扩展：

```
@org.hibernate.annotations.Type(
    type = " persistence.MonetaryAmountUserType"
)
@Column(name = "INITIAL_PRICE")
private MonetaryAmount initialPrice;
```

这是UserType可以执行的一种最简单的转化。更加复杂的事情也有可能。定制的映射类型可以执行验证；它可以把数据读取和写入到一个LDAP目录；它甚至可以从不同的数据库获取持久化对象。你主要受限于你的想象力。

在现实应用中，我们更喜欢在数据库中表示货币金额的金额和货币类型，尤其当Schema不是遗留的，却又可以被定义（或者快速更新）的时候。假设现在有两个列可用，并且不用多少转化就可以保存MonetaryAmount。第一种方法可能还是简单的<component>映射。然而，让我们试着用定制的映射类型来解决。

（不编写新的定制类型，而是试着把前一个例子改成两个列。这一点你可以不改变Java领域模型类就能完成——只有转换器需要为这个新需求和在映射中具名的新增列而更新。）

简单的UserType实现的缺点在于，Hibernate不知道有关MonetaryAmount内部单独属性的任何信息。它所知的只是定制类型类和列名称。Hibernate查询引擎（稍后将更深入讨论）不知道如何查询amount或者特定的currency。

如果需要Hibernate查询的全部功能，就编写CompositeUserType。这个（稍微有点复杂的）接口把MonetaryAmount的属性公开给Hibernate查询。现在再次用这个更灵活的定制接口把它映射到两个列，实际上生成了组件映射的等价物。

5.3.5　创建 CompositeUserType

为了示范定制映射类型的灵活性，不要对MonetaryAmount类（和其他持久化类）作任何改变——只改变定制映射类型，如代码清单5-5所示。

代码清单5-5　给新的数据库Schema中的货币金额定制映射类型

```
public class MonetaryAmountCompositeUserType
        implements CompositeUserType {

    // public int[] sqlTypes()...          ❶
```

```
public Class returnedClass...
public boolean isMutable...
public Object deepCopy...
public Serializable disassemble...
public Object assemble...
public Object replace...
public boolean equals...
public int hashCode...

public Object nullSafeGet(ResultSet resultSet,                    ❷
                          String[] names,
                          SessionImplementor session,
                          Object owner)
        throws SQLException {

    BigDecimal value = resultSet.getBigDecimal( names[0] );
    if (resultSet.wasNull()) return null;
    Currency currency =
        Currency.getInstance(resultSet.getString( names[1] ) );
    return new MonetaryAmount(value, currency);
}

public void nullSafeSet(PreparedStatement statement,              ❸
                        Object value,
                        int index,
                        SessionImplementor session)
        throws SQLException {
    if (value==null) {
      statement.setNull(index, Hibernate.BIG_DECIMAL.sqlType());
      statement.setNull(index+1, Hibernate.CURRENCY.sqlType());
    } else {
        MonetaryAmount amount = (MonetaryAmount) value;
        String currencyCode =
                    amount.getCurrency().getCurrencyCode();
        statement.setBigDecimal( index, amount.getAmount() );
        statement.setString( index+1, currencyCode );
    }
}

public String[] getPropertyNames() {                              ❹
    return new String[] { "amount", "currency" };
}

public Type[] getPropertyTypes() {                                ❺
    return new Type[] { Hibernate.BIG_DECIMAL,
                        Hibernate.CURRENCY };
}

public Object getPropertyValue(Object component, int property) {  ❻
    MonetaryAmount monetaryAmount = (MonetaryAmount) component;
    if (property == 0)
        return monetaryAmount.getAmount();
    else
        return monetaryAmount.getCurrency();
```

```
    }
    public void setPropertyValue(Object component,          ❼
                                 int property,
                                 Object value) {
        throw new
          UnsupportedOperationException("Immutable MonetaryAmount!");
    }
}
```

❶ CompositeUserType接口需要与你前面创建的UserType相同的内部处理方法。但是不再需要sqlTypes()方法。

❷ 现在加载一个值很简单：把结果集中的两个列值转化为新MonetaryAmount实例中的两个属性值。

❸ 保存值涉及在预编译的语句中设置两个参数。

❹ CompositeUserType通过getPropertyNames()公开值类型的属性。

❺ 每个属性都有它们自己的类型，就像getPropertyTypes()定义的那样。现在SQL列的类型对于这个方法是隐式的。

❻ getPropertyValue()方法返回MonetaryAmount的单个属性的值。

❼ setPropertyValue()方法设置MonetaryAmount的单个属性的值。

initialPrice属性现在映射到两个列，因此在映射文件中要对两者都进行声明。第一列保存值；第二列保存MonetaryAmount的货币类型：

```
<property name="initialPrice"
          type="persistence.MonetaryAmountCompositeUserType">
    <column name="INITIAL_PRICE"/>
    <column name="INITIAL_PRICE_CURRENCY"/>
</property>
```

如果通过注解映射Item，就必须给这个属性声明几个列。无法多次使用javax.persistence.Column注解，因此需要一个特定于Hibernate的新注解：

```
@org.hibernate.annotations.Type(
    type = "persistence.MonetaryAmountUserType"
)
@org.hibernate.annotations.Columns(columns = {
    @Column(name="INITIAL_PRICE"),
    @Column(name="INITIAL_PRICE_CURRENCY", length = 2)
})
private MonetaryAmount initialPrice;
```

在Hibernate查询中，现在可以引用定制类型的amount和currency属性，即使它们没有作为单独的属性出现在映射文件的任何地方：

```
from Item i
where i.initialPrice.amount > 100.0
  and i.initialPrice.currency = 'AUD'
```

我们已经用这个新的定制复合类型扩展了Java对象模型和SQL数据库模式之间的缓冲区。这两种表示法现在都更经得起变化。注意列的数目与选择UserType或者CompositeUserType无

关——只要愿意把值类型属性公开给Hibernate查询。

对于所有定制的映射类型来说，参数化是一项很有帮助的特性。

5.3.6 参数化定制类型

假设你再次面临最初的问题：当把金额保存到数据库中时把它转化为一种不同的货币类型。这些问题经常比一般的转化更为微妙；例如，可以在一些表中保存美元，而在其他表中保存欧元。你仍然想要给它编写单个定制的映射类型，使它可以进行任意的转化。如果把ParameterizedType接口添加到UserType或者CompositeUserType类，这是可能的：

```
public class MonetaryAmountConversionType
    implements UserType, ParameterizedType {

    // Configuration parameter
    private Currency convertTo;

    public void setParameterValues(Properties parameters) {
       this.convertTo = Currency.getInstance(
                            parameters.getProperty("convertTo")
                        );
    }

    // ... Housekeeping methods

    public Object nullSafeGet(ResultSet resultSet,
                              String[] names,
                              SessionImplementor session,
                              Object owner)
            throws SQLException {

       BigDecimal value = resultSet.getBigDecimal( names[0] );
       if (resultSet.wasNull()) return null;
       // When loading, take the currency from the database
       Currency currency = Currency.getInstance(
                            resultSet.getString( names[1] )
                            );
       return new MonetaryAmount(value, currency);
    }

    public void nullSafeSet(PreparedStatement statement,
                            Object value,
                            int index,
                            SessionImplementor session)
            throws SQLException {

       if (value==null) {
           statement.setNull(index, Types.NUMERIC);
       } else {
           MonetaryAmount amount = (MonetaryAmount) value;
           // When storing, convert the amount to the
           // currency this converter was parameterized with
           MonetaryAmount dbAmount =
                MonetaryAmount.convert(amount, convertTo);
           statement.setBigDecimal( index, dbAmount.getAmount() );
```

```
                statement.setString( index+1,
                                     dbAmount.getCurrencyCode() );
            }
        }
    }
```

我们在这个例子中省略了通常强制的内部处理方法。另一种重要的方法是ParameterizedType接口的setParameterValues()。Hibernate在启动时用convertTo参数调用这个方法来初始化这个类。保存MonetaryAmount时，nullSafeSet()方法就使用这个设置转化为目标货币类型。nullSafeGet()方法采用数据库中出现的这种货币，并把它留给客户去处理被加载的MonetaryAmount的货币类型（这种不对称的实现自然不是最佳办法。）

应用定制映射类型时，现在必须在映射文件中设置配置参数。一种简单的解决方案是属性中嵌套着的<type>映射：

```
<property name="initialPrice">
    <column name="INITIAL_PRICE"/>
    <column name="INITIAL_PRICE_CUR"/>
    <type name="persistence.MonetaryAmountConversionType">
        <param name="convertTo">USD</param>
    </type>
</property>
```

然而，如果在你的领域模型中有许多个货币金额时，这样很不方便，并且需要复制。一种更好的策略是使用类型的单个定义，包括所有参数，用一个可以在所有映射中重用的唯一名称。通过单个的<typedef>元素（你也可以不通过参数使用它）完成这一点：

```
<typedef class="persistence.MonetaryAmountConversionType"
         name="monetary_amount_usd">
    <param name="convertTo">USD</param>
</typedef>

<typedef class="persistence.MonetaryAmountConversionType"
         name="monetary_amount_eur">
    <param name="convertTo">EUR</param>
</typedef>
```

这里介绍的是，把带有一些参数的定制映射类型绑定到名称monetary_amount_usd和monetary_amount_eur上。这个定义可以放在映射文件中的任何位置；它是<hibernate-mapping>的一个子元素（如本书前面提到过的，较大的应用程序经常有一个或者几个没有类映射的MyCustomTypes.hbm.xml文件）。利用Hibernate扩展，可以在注解中定义带有参数的具名定制类型：

```
@org.hibernate.annotations.TypeDefs({
    @org.hibernate.annotations.TypeDef(
        name="monetary_amount_usd",
        typeClass = persistence.MonetaryAmountConversionType.class,
        parameters = { @Parameter(name="convertTo", value="USD") }
    ),
    @org.hibernate.annotations.TypeDef(
        name="monetary_amount_eur",
        typeClass = persistence.MonetaryAmountConversionType.class,
```

```
            parameters = { @Parameter(name="convertTo", value="EUR") }
        )
    })
```

这个注解元数据也是全局的，因此可以放在任何Java类声明之外（就在import语句之后），或者在一个单独的文件package-info.java中，如2.2.1节中所述。这个系统中一个很好的位置则是persistence包中的package-info.java文件。

在XML映射文件和注解映射中，现在引用被定义的类型名称，而不是定制类型完全限定的类名：

```
<property name="initialPrice"
          type="monetary_amount_usd">
    <column name="INITIAL_PRICE"/>
    <column name="INITIAL_PRICE_CUR"/>
</property>
```

```
@org.hibernate.annotations.Type(type = "monetary_amount_eur")
@org.hibernate.annotations.Columns({
  @Column(name = "BID_AMOUNT"),
  @Column(name = "BID_AMOUNT_CUR")
})
private MonetaryAmount bidAmount;
```

来看定制映射类型的一个不同的、极为重要的应用程序。这个类型安全的枚举（enumeration）设计模式几乎在所有的应用程序中都可以见到。

5.3.7 映射枚举

枚举类型是一个常见的Java惯用语，其中类有着（小）数量不变的不可变实例。在CaveatEmptor中，这可以应用到信用卡上：例如，要表达用户可以输入及应用程序提供的可能的类型（Mastercard、Visa等）。或者，可以枚举用户会在Comment中提交的有关一次特定拍卖的可能等级。

在更早版本的JDK中，你必须自己实现这些类（我们称它们为CreditCardType和Rating），遵循类型安全的枚举模式。如果没有JDK 5.0，这也仍然是实现它的一种好办法；在Hibernate社区网站上可以找到这种模式和兼容的定制映射类型。

1. 在JDK 5.0中使用枚举

如果使用JDK 5.0，可以给类型安全的枚举使用内建的语言支持。例如，Rating类看起来如下：

```
package auction.model;

public enum Rating {
    EXCELLENT, OK, BAD
}
```

Comment类有这个类型的一个属性：

```
public class Comment {
    ...
    private Rating rating;
    private Item auction;
    ...
}
```

下面是在应用程序代码中使用枚举的方式：

```
Comment goodComment =
    new Comment(Rating.EXCELLENT, thisAuction);
```

现在必须持久化这个Comment实例和它的Rating。一种方法是使用枚举的实际名称，并把它保存到COMMENTS表中的一个VARCHAR列。这个RATING列将包含EXCELLENT、OK或者BAD，具体情况取决于给定的Rating。

我们来编写Hibernate UserType，它可以加载和保存VARCHAR支持的枚举，例如Rating。

2. 编写定制的枚举处理程序

现在要介绍的不是最基础的UserType接口，而是EnhanceUserType接口。这个接口允许在XML表示模式中使用Comment实体，而不仅仅作为一个POJO（请见3.4节中对数据表示法的讨论）。此外，由于额外的ParameterizedType接口，你将要编写的实现可以支持任何VARCHAR支持的枚举，而不仅仅是Rating。

看看代码清单5-6中的代码。

代码清单5-6　给string支持的枚举定制映射类型

```
public class StringEnumUserType
        implements EnhancedUserType, ParameterizedType {

    private Class<Enum> enumClass;

    public void setParameterValues(Properties parameters) {   ❶
        String enumClassName =
            parameters.getProperty("enumClassname");
        try {
            enumClass = ReflectHelper.classForName(enumClassName);
        } catch (ClassNotFoundException cnfe) {
            throw new
              HibernateException("Enum class not found", cnfe);
        }
    }

    public Class returnedClass() {   ❷
        return enumClass;
    }

    public int[] sqlTypes() {   ❸
        return new int[] { Hibernate.STRING.sqlType() };
    }

    public boolean isMutable...   ❹
    public Object deepCopy...
    public Serializable disassemble...
    public Object replace...
    public Object assemble...
    public boolean equals...
    public int hashCode...

    public Object fromXMLString(String xmlValue) {   ❺
        return Enum.valueOf(enumClass, xmlValue);
```

```
}

public String objectToSQLString(Object value) {
    return '\'' + ( (Enum) value ).name() + '\'';
}

public String toXMLString(Object value) {
    return ( (Enum) value ).name();
}

public Object nullSafeGet(ResultSet rs,                    ❻
                          String[] names,
                          Object owner)
        throws SQLException {
    String name = rs.getString( names[0] );
    return rs.wasNull() ? null : Enum.valueOf(enumClass, name);   ❼
}

public void nullSafeSet(PreparedStatement st,
                        Object value,
                        int index)
        throws SQLException {
    if (value == null) {
        st.setNull(index, Hibernate.STRING.sqlType());
    } else {
        st.setString( index, ( (Enum) value ).name() );
    }
}
```

❶ 这个定制映射类型的配置参数是用于枚举类的名称，如Rating。

❷ 它也是从这个方法中返回的类。

❸ 数据库表中需要的单个VARCHAR列。通过让Hibernate决定SQL数据类型使它保持可移植。

❹ 这些是不可变类型一般的内部管理方法。

❺ 下列三种方法是EnchanceUserType的一部分，并用于XML封送。

❻ 加载枚举时，从数据库中获取它的名称，并创建一个实例。

❼ 保存枚举时，保存它的名称。

接下来，用这个新定制的类型映射rating属性。

3. 用XML和注解映射枚举

在XML映射中，首先创建一个定制的类型定义：

```
<typedef class="persistence.StringEnumUserType"
         name="rating">
    <param name="enumClassname">auction.model.Rating</param>
</typedef>
```

现在可以在Comment类映射中使用名为rating的类型：

```
<property  name="rating"
           column="RATING"
           type="rating"
           not-null="true"
```

```
            update="false"
            access="field"/>
```

因为等级是不可变的，你把它映射为update="false"，并启用直接的字段访问（不可变的属性没有设置方法）。如果除了Comment之外的其他类有Rating属性，就再次使用定义好的定制映射类型。

注解中这个定制映射类型的定义和声明看起来与你在前一节中所做的一样。

另一方面，可以依赖Java Persistence提供程序持久化枚举。如果在类型java.lang.Enum中的一个被注解的实体类中有一个属性（例如Comment中的rating），并且它没有标记为@Transient或者transient（Java关键字），Hibernate JPA实现必须毫无怨言地保持该属性为开箱即用；它有一个处理这个的内建类型。这个内建的映射类型在数据库中必须默认为枚举的一种表示法。两种最常见的选择是字符串表示法，就像通过定制类型或者有序表示法与原生的Hibernate实现一样。有序表示法保存选中的枚举选项的位置：例如，1为EXCELLENT，2为OK，3为BAD。数据库列也默认为一个数字列。可以在属性上通过Enumerated注解改变这个默认的枚举映射：

```
public class Comment {
    ...

    @Enumerated(EnumType.STRING)
    @Column(name = "RATING", nullable = false, updatable = false)
    private Rating rating;
    ...
}
```

现在已经切换到了基于字符串的表示法，实际上是与你的定制类型可以读写的相同的表示法。也可以使用JPA XML描述符：

```
<entity class="auction.model.Item" access="PROPERTY">
  <attributes>
    ...
    <basic name="rating">
        <column name="RATING" nullable="false" updatable="false"/>
        <enumerated>STRING</enumerated>
    </basic>
  </attributes>
</entity>
```

你可能会（理直气壮地）问，当Hibernate作为Java Persistence提供程序，可以显然开箱即用地持久化和加载枚举时，为什么必须给枚举编写自己的定制映射类型。这个秘密在于Hibernate Annotations包括了几个定制映射类型，该定制映射类型实现由Java Persistence定义的行为。可以在XML映射中使用这些定制类型；但是它们不是对用户友好的（需要许多参数），并且不是为这一目的而编写的。可以检查源代码（例如Hibernate Annotations中的org.hibernate.type.EnumType）来了解它们的参数，并决定是否要在XML中直接使用它们。

4．用定制映射类型查询

你可能遇到的进一步问题是在Hibernate查询中使用被枚举的类型。例如，考虑获取所有等级为“bad”的评语的HQL格式的下列查询：

```
Query q =
    session.createQuery(
      "from Comment c where c.rating = auction.model.Rating.BAD"
    );
```

如果把枚举持久化为字符串，虽然这个查询也有效（查询解析器用枚举值作为常量），但是如果选择了有序表示法，它就不起作用了。必须使用一个绑定参数，并对比较式通过编程来设置等级值：

```
Query q =
    session.createQuery("from Comment c where c.rating = :rating");

Properties params = new Properties();
params.put("enumClassname",
           "auction.model.Rating");

q.setParameter("rating", Rating.BAD,
               Hibernate.custom(StringEnumUserType.class, params)
              );
```

本例的最后一行使用静态的辅助方法Hibernate.custom()，来把定制的映射类型转化为Hibernate Type；这是告诉Hibernate有关枚举映射，以及如何处理Rating.BAD值的一种简单方法。注意，你还必须告诉Hibernate有关被参数化的类型可能需要的任何初始化属性。

不幸的是，Java Persistence中没有用于任意的和定制的查询参数的API，因此必须退回到Hibernate Session API，并创建一个Hibernate Query对象。

建议你要非常熟悉Hibernate类型系统，并把定制映射类型的创建当作一项基本技能——它在用Hibernate或者JPA开发的每个应用程序中都将非常有用。

5.4 小结

本章介绍了如何利用4种基础的继承映射策略，来把实体的继承层次结构映射到数据库：带有隐式多态的每个具体类一张表、带有联合的每个具体类一张表、每个类层次结构一张表和每个子类一张标准化表的策略。你已经见到了如何给一个特定的层次结构混合这些策略，以及每一种策略什么时候最适合。

我们也详细阐述了有关Hibernate实体和值类型的区别，以及Hibernate映射类型系统如何工作。你用了各种内建的类型，并利用如UserType和ParameterizedType的Hibernate扩展点编写自己的定制类型。

表5-5概括了原生的Hibernate特性和Java Persistence的对比。

表5-5 第5章的Hibernate和JPA对照表

Hibernate Core	Java Persistence和EJB 3.0
支持4种继承映射策略。混合继承策略成为可能	4种继承映射策略被标准化；在一个层次结构中混合策略被认为是不可移植的。JPA兼容的提供程序只需要每个类层次结构一张表和每个子类一张表
持久化的基类型可以是抽象类或者接口（只包含属性访问方法）	持久化的基类型可以是抽象类；被映射的接口被认为是不可移植的

（续）

Hibernate Core	Java Persistence和EJB 3.0
为值类型属性提供灵活的内建映射类型和转换器	有映射类型的自动侦测，包含对暂时的和枚举映射类型的标准覆盖。Hibernate扩展注解用于任何定制映射类型声明
强大的可扩展的类型系统	标准需要用于枚举、LOB和许多其他值类型的内建类型，必须在原生的Hibernate中为它们编写或者应用一个定制的映射类型

第6章将介绍集合映射，并讨论如何处理值类型对象的集合（例如，`String`的集合）和包含对实体实例引用的集合。

第6章

映射集合和实体关联

本章内容

- 基本的集合映射策略
- 映射值类型的集合
- 映射父/子实体关系

有两个重要（且有时难以理解）的主题——集合的映射和实体类之间关联的映射并没有出现在前面的章节中。

大多数刚接触Hibernate的开发人员在第一次尝试映射典型的父/子关系（parent/child relationship）时，都要处理集合和实体关联。本章不会立即切入正题，而是从基本的集合映射概念和简单的例子开始，之后给出实体关联中的第一个集合，当然我们在第7章会讲到更复杂的实体关联映射。为了对这部分知识获得全面的了解，建议你阅读这两章。

6.1 值类型的 set、bag、list 和 map

值类型（value type）的对象不具备数据库同一性，它属于一个实体实例，其持久化状态被嵌入到所拥有实体的表行中——至少，在实体有一个对值类型的单个实例的引用的情况下。如果实体类有一个值类型的集合（或者对值类型实例的引用的集合），就需要一张额外的表，即所谓的集合表。

在将值类型的集合映射到集合表之前，要记住，值类型的类没有标识符或者标识符属性。值类型实例的生命期限由所拥有的实体实例的生命期限决定。值类型不支持共享的引用。

Java具有一个丰富的集合API，因此可以选择最适合领域模型设计的集合接口和实现。我们来看一些最常见的集合映射。

假设CaveatEmptor的卖主能够给`Item`添加图片。图片只有通过包含的货品才能访问；它不需要支持来自系统中任何其他实体的关联。应用程序通过`Item`类管理图片集合，增加和移除元素。图片对象在集合之外没有生命，它依赖`Item`实体。

在这种情况下，把图片建模为值类型就不无道理了。接下来，需要决定使用什么集合。

6.1.1　选择集合接口

在Java领域模型中，集合属性的惯用语始终相同：

```
private <<Interface>> images = new <<Implementation>>();
...
// Getter and setter methods
```

使用接口来声明属性的类型，而不是实现。选择一种匹配的实现并立即初始化集合，这么做避免了未被初始化的集合（我们不建议在构造函数或者设置方法中太迟初始化集合）。

如果使用JDK 5.0，可能用JDK集合的一般版本进行编码。注意这不是必需的，你也可以在映射元数据中显式地指定集合的内容。以下是一个典型的一般的Set，包含类型参数：

```
private Set<String> images = new HashSet<String>();
...
// Getter and setter methods
```

开箱即用，Hibernate支持最重要的JDK集合接口。换句话说，它知道如何以持久化的方式保存JDK集合、映射和数组的语义。每个接口都有一个Hibernate支持的匹配实现，并且使用正确的组合很重要。Hibernate只包装已经在字段的声明中初始化的集合对象（或者如果不是正确的对象，有时就替换它）。

不扩展Hibernate，而是从下列集合中选择：

- 使用<set>元素映射java.util.Set。使用java.util.HashSet初始化集合。它的元素顺序没有保存，并且不允许重复元素。这在典型的Hibernate应用程序中是最常见的持久化集合。
- 可以使用<set>映射java.util.SortedSet，且sort属性可以设置成比较器或者用于内存排序的自然顺序。使用java.util.TreeSet实例初始化集合。
- 可以使用<list>映射java.util.List，在集合表中用一个额外的索引列保存每个元素的位置。使用java.util.ArrayList初始化。
- 可以使用<bag>或者<idbag>映射java.util.Collection。Java没有Bag接口或者实现；然而，java.util.Collection允许包语义（可能的重复，不保存元素顺序）。Hibernate支持持久化的包（它内部使用列表，但是忽略元素的索引）。使用java.util.ArrayList初始化包集合。
- 可以使用<map>映射java.util.Map，保存键/值对。使用java.util.HashMap初始化属性。
- 可以使用<map>元素映射java.util.SortedMap，且sort属性可以设置为比较器或者用于内存排序的自然顺序。使用java.util.TreeMap实例初始化该集合。
- Hibernate使用<primitive-array>（对于Java基本的值类型）和<array>（对于其他的一切）支持数组（array）。但是它们很少用在领域模型中，因为Hibernate无法包装数组属性。没有字节码基础设施（BCI），就失去了延迟加载，以及为持久化集合优化过的脏检查、基本的便利和性能特性。

JPA标准没有列出所有这些选项。可能的标准集合属性类型是Set、List、Collection和Map。不考虑数组。

此外，JPA规范仅仅指出集合属性持有对实体对象的引用。值类型的集合（例如简单的String实例）没有被标准化。然而，规范文件已经提及JPA的未来版本将支持可嵌入类的集合元素（换句话说，是值类型）。如果想要通过注解映射值类型的集合，将需要特定于供应商的支持。Hibernate Annotations包括了这种支持，我们期待许多其他的JPA供应商也同样支持。

如果想要映射Hibernate不直接支持的集合接口和实现，就要告诉Hibernate定制集合的语义。Hibernate中的扩展点称作PersistentCollection；通常扩展其中现有的PersistentSet、PersistentBag或者PersistentList类中的一个。定制持久化集合不太容易编写，如果你不是个经验丰富的Hibernate用户，我们也不建议这么做。示例作为Hibernate下载包的一部分，可以在Hibernate测试套件（test suite）源代码中找到。

现在快速看一下几种始终实现货品图片集合的场景。首先以XML格式映射它，然后利用Hibernate对集合注解的支持。现在，假设图片被保存在文件系统中的某个地方，并且在数据库中只保存了文件名。我们不讨论如何用这种方法保存和加载图片，而是关注映射。

6.1.2 映射 set

最简单的实现是String图片文件名的Set。首先，把集合属性添加到Item类：

```
private Set images = new HashSet();
...
public Set getImages() {
    return this.images;
}
public void setImages(Set images) {
    this.images = images;
}
```

现在，在Item的XML元数据中创建下列映射：

```
<set name="images" table="ITEM_IMAGE">

    <key column="ITEM_ID"/>

    <element type="string" column="FILENAME" not-null="true"/>
</set>
```

图片的文件名保存在具名ITEM_IMAGE的集合表中。从数据库的观点来看，这张表是单个的实体，是一张单独的表，但是Hibernate为你隐藏了这一点。<key>元素在引用自己的实体的主键ITEM_ID的集合表中，声明了外键列。<element>标签把这个集合声明为值类型实例的一个集合——在这个例子中，是字符串的集合。

集无法包含重复元素，因此ITEM_IMAGE集合表的主键是<set>声明中这两个列的复合：ITEM_ID和FILENAME。可以在图6-1中见到这个Schema。

允许用户不止一次地添加同一张图片似乎不可能，但是假设你允许这么做，那么在这种情况下，哪种映射最合适呢？

ITEM

ITEM_ID	NAME
1	Foo
2	Bar
3	Baz

ITEM_IMAGE

ITEM_ID	FILENAME
1	fooimage1.jpg
1	fooimage2.jpg
2	barimage1.jpg

图6-1　字符串集合的表结构和示例数据

6.1.3　映射标识符 bag

允许重复元素的无序集合被称作包（bag）。奇怪的是，Java Collections框架没有包括包实现。然而，java.util.Collection接口有包语义，因此只需要一种相匹配的实现。你有两种选择：

- 用java.util.Collection接口编写集合属性，并在声明中用JDK的一个ArrayList对它进行初始化。在Hibernate中用标准的<bag>或者<idbag>元素映射集合。Hibernate有一个内建的PersistentBag可以处理列表；但与包的约定一致，它忽略元素在ArrayList中的位置。换句话说，你得到了一个持久化的Collection。
- 用java.util.List接口编写集合属性，并在声明中用JDK的一个ArrayList把它初始化。像前一个选项一样映射它，但是在领域模型类中公开了一个不同的集合接口。这种方法有效，但不建议使用，因为使用这个集合属性的客户端可能认为元素的顺序会始终被保存着，其实如果把它作为<bag>或者<idbag>映射，就并非如此了。

建议使用第一个选项。把Item类中images的类型由Set改为Collection，并用ArrayList把它初始化：

```
private Collection images = new ArrayList();
...
public Collection getImages() {
    return this.images;
}

public void setImages(Collection images) {
    this.images = images;
}
```

注意，设置方法接受Collection，它可以是JDK集合接口层次结构中的任何东西。然而，Hibernate聪明到足以在持久化集合的时候替换它。（它内部也依赖ArrayList，就像你在字段的声明中所做的那样。）

还必须修改集合表以允许重复FILENAME；表需要一个不同的主键。<idbag>映射添加了一个代理键列到集合表，很像用于实体类的合成标识符：

```
<idbag name="images" table="ITEM_IMAGE">

    <collection-id type="long" column="ITEM_IMAGE_ID">
        <generator class="sequence"/>
    </collection-id>

    <key column="ITEM_ID"/>

    <element type="string" column="FILENAME" not-null="true"/>
</idbag>
```

在这个例子中，主键是生成的ITEM_IMAGE_ID，就如在图6-2中所见。注意，主键的native生成器不支持<idbag>映射，必须指定一种具体的策略。这通常不成问题，因为无论如何，现实世界的应用程序都经常使用定制的标识符生成器。也可以用占位符隔离标识符生成策略；请见3.3.4节的第3小节。

ITEM

ITEM_ID	NAME
1	Foo
2	Bar
3	Baz

ITEM_IMAGE

ITEM_IMAGE_ID	ITEM_ID	FILENAME
1	1	fooimage1.jpg
2	1	fooimage1.jpg
3	3	barimage1.jpg

图6-2 允许重复包元素的一个代理主键

还要注意，ITEM_IMAGE_ID列没有以任何方式公开给应用程序。Hibernate在内部管理它。

一种更为可能的场景是，你希望把添加图片到Item的顺序保存在那里。实现这个有许多种好方法，其中一种是使用真实的列表，而不是包。

6.1.4 映射 list

首先，更新Item类：

```
private List images = new ArrayList();
...
public List getImages() {
    return this.images;
}

public void setImages(List images) {
    this.images = images;
}
```

<list>映射需要把一个索引列（index column）新增到集合表。索引列定义元素在集合中的位置。因而，Hibernate能够保存集合元素的顺序。映射集合为<list>：

```
<list name="images" table="ITEM_IMAGE">

    <key column="ITEM_ID"/>

    <list-index column="POSITION"/>
    <element type="string" column="FILENAME" not-null="true"/>
</list>
```

（XML DTD中也有index元素，用于与Hibernate 2.x兼容。建议使用这个新的list-index；它比较不容易引起混淆，并且作用相同。）

集合表的主键是ITEM_ID和POSITION的复合。注意，现在允许重复元素（FILENAME）了，这与列表的语义一致，请见图6-3。

持久化列表的索引从0开始。可以改变它，例如在映射中使用<list-index base="1".../>。注意，如果数据库中的索引数字不连续，Hibernate就会把空元素添加到Java列表中。

ITEM

ITEM_ID	NAME
1	Foo
2	Bar
3	Baz

ITEM_IMAGE

ITEM_ID	POSITION	FILENAME
1	0	fooimage1.jpg
1	1	fooimage2.jpg
1	2	fooimage3.jpg

图6-3　集合表保存每个元素的位置

另一种方法，可以映射一个Java数组而不是列表。Hibernate支持这个；数组映射事实上与前一个例子是等同的，除了使用不同的元素和属性名称之外（<array>和<array-index>）。然而，基于前面阐述过的原因，Hibernate应用程序很少使用数组。

现在，假设一件货品的图片除了文件名之外还有用户提供的名称。在Java中对这个进行建模的一种方法是映射，使用与键相同的名称以及与映射的值一样文件名。

6.1.5　映射 map

再一次对这个Java类做小小的变化：

```
private Map images = new HashMap();
...
public Map getImages() {
    return this.images;
}
public void setImages(Map images) {
    this.images = images;
}
```

再次强调，映射<map>类似于映射列表。

```
<map name="images" table="ITEM_IMAGE">
    <key column="ITEM_ID"/>
    <map-key column="IMAGENAME" type="string"/>
    <element type="string" column="FILENAME" not-null="true"/>
</map>
```

集合表的主键是ITEM_ID和IMAGENAME的复合。IMAGENAME列保存映射的键。还是允许重复元素；请见图6-4中表的图表形式。

ITEM

ITEM_ID	NAME
1	Foo
2	Bar
3	Baz

ITEM_IMAGE

ITEM_ID	IMAGENAME	FILENAME
1	Image One	fooimage1.jpg
1	Image Two	fooimage2.jpg
1	Image Three	fooimage3.jpg

图6-4　把字符串作为索引和元素的映射表

这个映射是无序的。如果想要始终按图片的名称进行排序，该怎么办？

6.1.6 排序集合和有序集合

虽然英语单词遭到惊人的滥用，单词sorted和ordered对于Hibernate持久化集合却是指不同的东西。排序集合（sorted collection）是指用一个Java比较器在内存中进行排序。有序集合（ordered collection）则指用一个包含order by子句的SQL查询在数据库级中排列。

来把图片的映射变成一个排了序的映射。首先，要改变Java属性的初始化为java.util.TreeMap，并转换到java.util.SortedMap接口：

```
private SortedMap images = new TreeMap();
...
public SortedMap getImages() {
    return this.images;
}

public void setImages(SortedMap images) {
    this.images = images;
}
```

如果把它映射为排序，Hibernate会相应地处理这个集合：

```
<map name="images"
        table="ITEM_IMAGE"
        sort="natural">

    <key column="ITEM_ID"/>

    <map-key column="IMAGENAME" type="string"/>

    <element type="string" column="FILENAME" not-null="true"/>
</map>
```

通过指定sort="natural"，告诉Hibernate使用SortedMap，并根据java.lang.String的compareTo()方法对图片名称进行排序。如果需要一些其他的排序算法（例如反向字母顺序），可以在sort属性中指定实现java.util.Comparator的类名称。例如：

```
<map name="images"
     table="ITEM_IMAGE"
     sort="auction.util.comparator.ReverseStringComparator">

    <key column="ITEM_ID"/>

    <map-key column="IMAGENAME" type="string"/>

    <element type="string" column="FILENAME" not-null="true"/>
</map>
```

像下面这样映射java.util.SortedSet（包含一个java.util.TreeSet实现）：

```
<set name="images"
     table="ITEM_IMAGE"
     sort="natural">

    <key column="ITEM_ID"/>

    <element type="string" column="FILENAME" not-null="true"/>
</set>
```

包不可能被排序（可惜没有TreeBag），列表（list）也一样；列表元素的顺序由列表索引定义。

另一种方法，不转换到Sorted*接口（和Tree*实现），你或许想要使用一个链接映射（Linked Map），并在数据库端而不是内存中给元素排序。在Java类中保留Map/HashMap声明，并创建下列映射：

```
<map name="images"
     table="ITEM_IMAGE"
     order-by="IMAGENAME asc">
    <key column="ITEM_ID"/>
    <map-key column="IMAGENAME" type="string"/>
    <element type="string" column="FILENAME" not-null="true"/>
</map>
```

order-by属性中的表达式是SQL order by子句的一个片段。在这个例子中，在集合的加载期间，Hibernate按IMAGENAME列的升序排列集合元素。甚至可以在order-by属性中包括SQL函数调用：

```
<map name="images"
     table="ITEM_IMAGE"
     order-by="lower(FILENAME) asc">
    <key column="ITEM_ID"/>
    <map-key column="IMAGENAME" type="string"/>
    <element type="string" column="FILENAME" not-null="true"/>
</map>
```

可以按集合表的任何列进行排列。Hibernate内部使用LinkedHashMap，它是保存关键元素插入顺序的一个映射的变形。换句话说，就是在集合的加载期间，Hibernate把元素添加到集合的顺序，就是你在应用程序中见到的迭代顺序。用Set也可以完成同样的工作：Hibernate内部使用LinkedHashSet。在Java类中，属性是一般的Set/HashSet，但是Hibernate内部包含LinkedHashSet的包装再次通过order-by属性启用了排列：

```
<set name="images"
     table="ITEM_IMAGE"
     order-by="FILENAME asc">
    <key column="ITEM_ID"/>
    <element type="string" column="FILENAME" not-null="true"/>
</set>
```

也可以让Hibernate在集合的加载期间为你排列包的元素。Java集合属性是Collection/ArrayList或者List/ArrayList。Hibernate内部使用ArrayList来实现一个保存了插入-迭代顺序的包：

```
<idbag name="images"
       table="ITEM_IMAGE"
       order-by="ITEM_IMAGE_ID desc">
```

```
        <collection-id type="long" column="ITEM_IMAGE_ID">
            <generator class="sequence"/>
        </collection-id>

        <key column="ITEM_ID"/>

        <element type="string" column="FILENAME" not-null="true"/>
    </idbag>
```

Hibernate内部用于集和映射的链接集合只在JDK 1.4或者更高的版本中可用；更早的JDK没有LinkedHashMap和LinkedHashSet。有序包在所有JDK版本中都可用；内部使用ArrayList。

在真实的系统中，你可能需要保存图片名称和文件名之外的更多东西。你或许想要给这些额外的信息创建一个Image类。对于组件的集合来说，这是一个完美的使用案例。

6.2 组件的集合

可以把Image映射为实体类，并创建一个从Item到Image的一对多关系。但是这没有必要，因为Image可以建模为一个值类型：这个类的实例有一个依赖的生命周期，不需要它们自己的同一性，并且不必支持共享的引用。

作为一个值类型，Image类定义了属性name、filename、sizeX和sizeY。它与它的所有者Item实体有着单个关联，如图6-5所示。

就如你可以从复合关联风格（图中黑色菱形处）中所见，Image是Item的一个组件，Item则是负责Image实例的生命周期的实体。关联的多样性进一步把这个关联声明为多值（many-valued）——也就是说，同一个Item实例有多个（或者0个）Image实例。

来快速看一下Java中的这个实现，以及XML格式的映射。

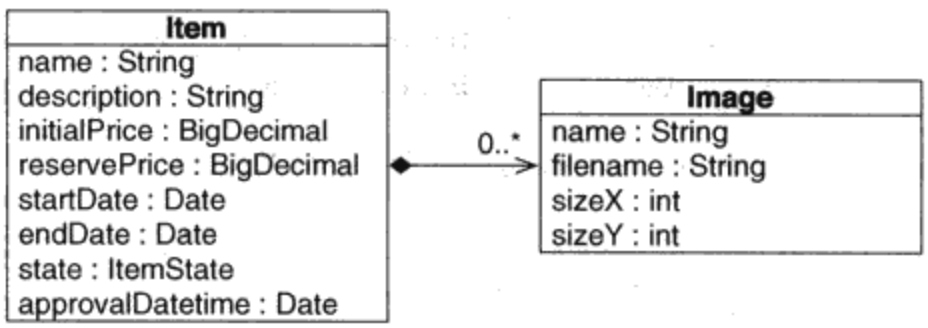


图6-5 Item中Image组件的集合

6.2.1 编写组件类

首先，把Image类实现为一般的POJO。如第4章所述，组件类没有标识符属性。你必须实现equals()（和hashCode()），并比较name、filename、sizeX和sizeY属性。Hibernate依赖这个等同性子程序检查实例中的修改。并非所有的组件类都需要equals()和hashCode()的定制实现（前面已经提到过这一点）。但是，我们建议把它用给任何组件类，因为这种实现很简单，并且“小心不出大错”可是至理名言啊。

Item类可以有图片的Set，不允许重复。我们来把这一思想映射到数据库。

6.2.2　映射集合

组件的集合被类似地映射到JDK值类型的集合。唯一的区别是用<composite-element>代替<element>标签。有序的图片集（内部使用LinkedHashSet）可以像这样映射：

```
<set name="images"
     table="ITEM_IMAGE"
     order-by="IMAGENAME asc">

    <key column="ITEM_ID"/>

    <composite-element class="Image">
       <property name="name" column="IMAGENAME" not-null="true"/>
       <property name="filename" column="FILENAME" not-null="true"/>
       <property name="sizeX" column="SIZEX" not-null="true"/>
       <property name="sizeY" column="SIZEY" not-null="true"/>
    </composite-element>
</set>
```

图6-6展现了包含示例数据的表。

ITEM

ITEM_ID	ITEM_NAME
1	Foo
2	Bar
3	Baz

ITEM_IMAGE

ITEM_ID	IMAGENAME	FILENAME	SIZEX	SIZEY
1	Foo	Foo.jpg	123	123
1	Bar	Bar.jpg	420	80
2	Baz	Baz.jpg	50	60

图6-6　组件映射的集合的示例数据表

这是一个集，因此集合表的主键是键列和所有元素列的一个复合：ITEM_ID、IMAGENAME、FILENAME、SIZEX和SIXEY。因为这些列都出现在主键中，需要用not-null="true"声明它们（或者确保它们在任何现有的Schema中都为NOT NULL）。复合主键中任何列都不可以为空——因为你无法辨别不知道的东西。这可能是这个特定映射的一个缺点。在改善这一点之前（可能你猜到了，用标识符bag），我们来启用双向导航。

6.2.3　启用双向导航

从Item到Image的关联是单向的。可以通过Item实例访问集合并迭代anItem.getImages().iterator()导航到该图片。这是可以获得这些图片对象的唯一方法；没有其他的实体保存对它们（还是值类型）的引用。

另一方面，从一张图片导航回到一件货品没有意义。但是，有时候访问后退指针就像访问anImage.getItem()一样可能会比较方便。如果把<parent>元素添加到映射，Hibernate就会为

你填入这个属性：

```
<set name="images"
     table="ITEM_IMAGE"
     order-by="IMAGE_NAME asc">

    <key column="ITEM_ID"/>

    <composite-element class="Image">
       <parent name="item"/>
       <property name="name" column="IMAGENAME" not-null="true"/>
       <property name="filename" column="FILENAME" not-null="true"/>
       <property name="sizeX" column="SIZEX" not-null="true"/>
       <property name="sizeY" column="SIZEY" not-null="true"/>
    </composite-element>
</set>
```

但是，真正的双向导航是不可能的。你无法单独获取一个Image，然后导航回到它的父Item。这是一个重要的问题：你可以通过查询它们来加载Image实例。但是当你在HQL中查询或者使用Criteria查询时，这些Image对象都没有引用它们的所有者（属性为null）。它们作为标量值（scalar value）获取。

最后，声明所有属性为not-null可能并不是你所希望的。如果任何属性列都是可为空的，IMAGE集合表就需要一个不同的主键。

6.2.4 避免非空列

类似于<idbag>提供的那个额外的代理标识符属性，现在代理键列要派上用场了。作为一种附带作用，<idset>也允许重复——明显与集的概念冲突。基于各种各样的原因（包括事实上也没有人曾经要过这一特性），Hibernate没有提供<idset>或者除了<idbag>之外的任何代理标识符集合。因而，要通过包语义把Java属性改为Collection：

```
private Collection images = new ArrayList();
...
public Collection getImages() {
    return this.images;
}

public void setImages(Collection images) {
    this.images = images;
}
```

这个集合现在也允许重复Image元素了——这是用户接口或者任何其他应用程序代码的责任，如果需要集语义，就要避免这些重复元素。映射添加了代理标识符列到集合表：

```
<idbag name="images"
       table="ITEM_IMAGE"
       order-by="IMAGE_NAME asc">

    <collection-id type="long" column="ITEM_IMAGE_ID">
        <generator class="sequence"/>
    </collection-id>
    <key column="ITEM_ID"/>
```

```
        <composite-element class="Image">
            <property name="name" column="IMAGENAME"/>
            <property name="filename" column="FILENAME" not-null="true"/>
            <property name="sizeX" column="SIZEX"/>
            <property name="sizeY" column="SIZEY"/>
        </composite-element>
    </idbag>
```

集合表的主键现在是ITEM_IMAGE_ID列，在Image类上实现equals()和hashCode()并不重要（至少Hibernate不需要）。你也不必用not-null="true"来声明属性。它们可以为空，如图6-7所示。

ITEM_IMAGE

ITEM_IMAGE_ID	ITEM_ID	IMAGENAME	FILENAME	SIZEX	SIZEY
1	1	Foo	Foo.jpg	123	123
2	1	Bar	Bar.jpg	420	80
3	2	Baz	Baz.jpg	NULL	NULL

图6-7　使用带有代理键的包的Image组件的集合

应该指出，在这个包映射和标准的父/子实体关系（如本章前面映射过的那个）之间并没有太大的区别。表是一致的。这种选择主要是个人偏爱的问题。父/子关系支持对子实体和真正的双向导航的共享引用。为此付出的代价是更复杂的对象生命周期。值类型的实例可以通过添加新元素到集合而创建，并与持久化的Item关联。也可以通过从集合中移除元素来解除关联和永远删除它们。如果Image要成为一个支持共享引用的实体类，同样的操作在应用程序中就需要更多的代码，如稍后所见。

转换到不同主键的另一种方法是map。可以从Image类中移除name属性，并用图片名称作为map的键：

```
<map name="images"
     table="ITEM_IMAGE"
     order-by="IMAGENAME asc">

    <key column="ITEM_ID"/>

    <map-key type="string" column="IMAGENAME"/>

    <composite-element class="Image">
        <property name="filename" column="FILENAME" not-null="true"/>
        <property name="sizeX" column="SIZEX"/>
        <property name="sizeY" column="SIZEY"/>
    </composite-element>
</map>
```

集合表的主键现在是ITEM_ID和IMAGENAME的复合。

像Image这样的复合元素类不是只限于像filename这样基本类型的简单属性。它可以包含其他的组件，通过<nested-composite-element>映射，甚至对实体的<many-to-one>关联。但是它无法拥有集合。包含多对一关联的复合元素很有用，第7章会回到这种映射的话题。

这样就结束了有关XML格式的基本集合映射的讨论。就像本节开头提到的，用注解映射值

类型的集合，与以XML格式映射相比有所不同；在编写本书之时，它还不是Java Persistence标准的一部分，但是在Hibernate中它还是可用的。

6.3 用注解映射集合

Hibernate Annotations包对包含值类型元素的集合映射支持非标准的注解，主要是org.hibernate.annotations.CollectionOfElements。再来快速看一些最常用的场景。

6.3.1 基本的集合映射

下列代码映射了String元素的一个简单集合：

```
@org.hibernate.annotations.CollectionOfElements(
    targetElement = java.lang.String.class
)
@JoinTable(
    name = "ITEM_IMAGE",
    joinColumns = @JoinColumn(name = "ITEM_ID")
)
@Column(name = "FILENAME", nullable = false)
private Set<String> images = new HashSet<String>();
```

集合表ITEM_IMAGE有两个列；它们一起形成了复合主键。如果使用一般的集合，Hibernate会自动侦测元素的类型。如果没有用泛型的集合编写代码，就要用targetElement属性指定元素类型——因此在前一个例子中，它是可选的。

为了映射持久化的List，要添加@org.hibernate.annotations.IndexColumn，该索引列带有索引的可选基数（默认为0）：

```
@org.hibernate.annotations.CollectionOfElements
@JoinTable(
    name = "ITEM_IMAGE",
    joinColumns = @JoinColumn(name = "ITEM_ID")
)
@org.hibernate.annotations.IndexColumn(
    name="POSITION", base = 1
)
@Column(name = "FILENAME")
private List<String> images = new ArrayList<String>();
```

如果忘记了索引列，这个列表就会被当作一个包集合处理，相当于XML格式的<bag>。

对于值类型的集合，通常用<idbag>在集合表中获得一个代理主键。值类型元素的<bag>没有真正起作用；在Java级中允许重复，但在数据库中不行。另一方面，纯粹的包对于一对多的实体关联非常棒，如你将在第7章所见。

为了映射持久化的映射，就用@org.hibernate.annotations.MapKey：

```
@org.hibernate.annotations.CollectionOfElements
@JoinTable(
    name = "ITEM_IMAGE",
    joinColumns = @JoinColumn(name = "ITEM_ID")
```

```
)
@org.hibernate.annotations.MapKey(
    columns = @Column(name="IMAGENAME")
)
@Column(name = "FILENAME")
private Map<String, String> images = new HashMap<String, String>();
```

如果忘记了映射键，这个映射的键就会自动映射到列MAPKEY。

如果该映射的键不是简单的字符串，而是可嵌入的类，就可以指定多个保存可嵌入组件的单独属性的映射键列。注意，@org.hibernate.annotations.MapKey是@javax.persistence.MapKey更强大的替代，后者不太有用（请见7.2.4节）。

6.3.2　排序集合和有序集合

集合也可以通过Hibernate注解排序或者按顺序排列：

```
@org.hibernate.annotations.CollectionOfElements
@JoinTable(
    name = "ITEM_IMAGE",
    joinColumns = @JoinColumn(name = "ITEM_ID")
)
@Column(name = "FILENAME", nullable = false)
@org.hibernate.annotations.Sort(
    type = org.hibernate.annotations.SortType.NATURAL
)
private SortedSet<String> images = new TreeSet<String>();
```

（注意，没有@JoinColumn和/或@Column，Hibernate就给Schema应用一般的命名约定和默认。）@Sort注解支持各种SortType属性，语义与XML映射选项相同。上述映射使用了一个java.util.SortedSet（带有一个java.util.TreeSet实现）和自然的排序顺序。如果启用SortType.COMPARATOR，也要把comparator属性设置为实现了比较子程序的类。映射也可以被排序；但是，就像以XML格式映射一样，没有排序的Java包或者排序列表（按定义，它有一个持久化的元素顺序）。

映射、集，甚至包，都可以在加载时由数据库排序，通过ORDER BY子句中一个SQL片段进行：

```
@org.hibernate.annotations.CollectionOfElements
@JoinTable(
    name = "ITEM_IMAGE",
    joinColumns = @JoinColumn(name = "ITEM_ID")
)
@Column(name = "FILENAME", nullable = false)
@org.hibernate.annotations.OrderBy(
    clause = "FILENAME asc"
)
private Set<String> images = new HashSet<String>();
```

特定于Hibernate的@OrderBy注解的clause属性，是被直接传递到数据库的一个SQL片段；它甚至可以包含函数调用或者任何其他原生的SQL关键字。有关排序和有序的内部实现细节请见前面的阐述；注解相当于XML映射。

6.3.3 映射嵌入式对象的集合

终于可以映射组件、用户自定义的值类型元素的集合了。假设你想要映射本章前面见过的同一个Image组件类，包含图片名称、大小等。

要在这个类上添加@Embeddable组件注解来启用嵌入：

```
@Embeddable
public class Image {

    @org.hibernate.annotations.Parent
    Item item;

    @Column(length = 255, nullable = false)
    private String name;
    @Column(length = 255, nullable = false)
    private String filename;

    @Column(nullable = false)
    private int sizeX;

    @Column(nullable = false)
    private int sizeY;

    ... // Constructor, accessor methods, equals()/hashCode()
}
```

注意，再次用Hibernate注解映射了一个后退指针，anImage.getItem()可能有用。如果不需要这个引用，可以省略这个属性。因为集合表需要所有组件列都为复合主键，把这些列映射为NOT NULL很重要。现在可以把这个组件嵌入到一个集合映射中了，甚至覆盖列定义（在下面的例子中，你覆盖了组件集合表的单个列的名称；所有其他的列都被指定为默认的策略）：

```
@org.hibernate.annotations.CollectionOfElements
@JoinTable(
    name = "ITEM_IMAGE",
    joinColumns = @JoinColumn(name = "ITEM_ID")
)
@AttributeOverride(
    name = "element.name",
    column = @Column(name = "IMAGENAME",
                     length = 255,
                     nullable = false)
)
private Set<Image> images = new HashSet<Image>();
```

为了避免不可为空的组件列，在集合表中需要一个代理主键，就像<idbag>在XML映射中提供的那样。通过注解，使用@CollectionId Hibernate扩展：

```
@org.hibernate.annotations.CollectionOfElements
@JoinTable(
    name = "ITEM_IMAGE",
    joinColumns = @JoinColumn(name = "ITEM_ID")
)
@CollectionId(
```

```
        columns = @Column(name = "ITEM_IMAGE_ID"),
        type = @org.hibernate.annotations.Type(type = "long"),
        generator = "sequence"
    )
    private Collection<Image> images = new ArrayList<Image>();
```

你现在已经使用XML映射元数据和注解，映射了所有基本的和一些更复杂的集合。转换关注点，现在思考包含非值类型、但引用其他实体实例的元素的集合。许多Hibernate用户试图映射一个典型的父/子实体关系，这涉及实体引用的集合。

6.4　映射父/子关系

在Hibernate用户社区的经验告诉我们，许多开发人员在开始使用Hibernate时要尝试的第一件事就是父/子关系的映射。这通常是你第一次遇到集合，也是你第一次必须思考实体和值类型之间的区别，否则会在ORM的复杂性中迷失。

管理类之间的关联和表之间的关系，是ORM的核心。在实现ORM解决方案中涉及的大部分难题都与关联管理有关。

前一节和本书的前面已经通过关系端的各种多样性，映射过值类型的类之间的关系。用一个简单的<property>映射一个（one）多样性，或者把它映射为<component>。多个（many）关联多样性需要值类型的集合，利用<element>或者<composite-element>映射。

现在想要映射实体类之间一值或者多值的关系。很显然，实体方面（如共享的引用和独立的生命周期）使得这个关系映射复杂化了。我们会逐步处理这些问题；而且，万一你不熟悉术语多样性（multiplicity），我们也将讨论到。

接下来的几节中要介绍的关系始终一样，即Item和Bid实体类之间的关系，如图6-8所示。

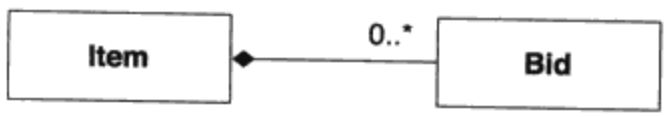


图6-8　Item和Bid之间的关系

记住这个类图。但是首先，我们要先行解释一些东西。

如果你已经用过EJB CMP 2.0，就会熟悉托管关联（或者托管关系）的概念。CMP关联被称作容器托管的关系（container managed relationship，CMR）是有理由的。CMP中的关联天生就是双向的。对关联的一侧所做的改变，会立即影响到另一侧。例如，如果调用aBid.setItem(anItem)，容器就会自动调用anItem.getBids().add(aBid)。

面向POJO的持久化引擎（如Hibernate）不实现托管关联，且POJO标准（如EJB 3.0和Java Persistence）不需要托管关联。与EJB 2.0 CMR相反，Hibernate和JPA关联天生都是单向的（unidirectional）。到Hibernate受到关注为止，从Bid到Item的关联与从Item到Bid的关联都是不同的关联！这是件好事——否则你的实体类就不可能在一个运行时容器之外使用（CMR是EJB 2.1实体之所以被认为有问题的主要原因）。

由于关联是如此重要，你需要一种精确的语言来对它们进行分类。

6.4.1 多样性

在描述和分类关联时，我们通常使用术语多样性（multiplicity）。在我们的例子中，多样性只是两小块信息：

- 一个特定的Item可以有不止一个Bid吗？
- 一个特定的Bid可以有不止一个Item吗？

看一眼领域模型（见图6-8）之后，就可以推断：从Bid到Item的关联是一个多对一的关联。回忆一下，关联是有方向性的，把从Item到Bid的反向关联归为一对多关联。

另外只有两种可能：多对多和一对一。我们将在第7章回到这个话题。

在对象持久化的上下文中，我们不关注多个（many）是否意味着两个或者最大五个或者没有限制；只关注大多数关联的选择性（optionality）；不特别关心是否需要关联实例，或者关联的另一侧是否可以为空（意味着0对多和0对0的关联）。但是，这些是影响你在关系数据Schema中选择完整性规则和在SQL DDL中定义约束的重要方面（请见8.3节）。

6.4.2 最简单的可能关联

从Bid到Item的关联（反之亦然）是一种最简单的可能实体关联的例子。两个类中有两个属性。一个是引用的集合，另一个是单个引用。

首先，这是Bid的Java类实现：

```
public class Bid {
    ...

    private Item item;

    public void setItem(Item item) {
        this.item = item;
    }

    public Item getItem() {
        return item;
    }

    ...
}
```

接下来是这个关联的Hibernate映射：

```
<class
    name="Bid"
    table="BID">
    ...
    <many-to-one
        name="item"
        column="ITEM_ID"
        class="Item"
        not-null="true"/>

</class>
```

这个映射被称作单向的多对一关联。(实际上，因为它是单向的，你不知道另一侧是什么，因此也可以称这个映射为单向的对一关联映射。)BID表中的ITEM_ID列是ITEM表主键的一个外键。

把类命名为Item，它显然是这个关联的目标。这通常是可选的，因为Hibernate可以通过反射在Java属性中确定目标类型。

添加了not-null属性，因为没有货品就不可能有出价——在SQL DDL中生成一项约束来体现这一点。BID中的外键列ITEM_ID永远不能为NULL，关联不是对0或者对一。这个关联映射的表结构如图6-9所示。

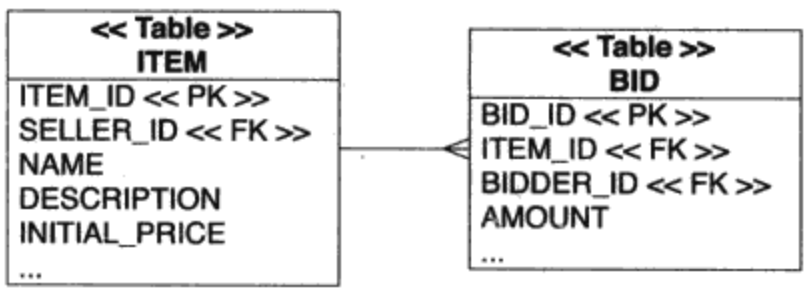


图6-9　一对多映射的表关系和主键

在JPA中，用@ManyToOne注解来映射这个关联，是在字段还是在获取方法中，这取决于实体的访问策略（由@Id注解的位置决定）：

```
public class Bid {
    ...
    @ManyToOne( targetEntity = auction.model.Item.class )
    @JoinColumn(name = "ITEM_ID", nullable = false)
    private Item item;

    ...
}
```

在这个映射中有两个可选的元素。首先，不一定要包括关联的targetEntity；它对于字段的类型来说是隐式的。显式的targetEntity属性在更复杂的领域模型中很有用。例如，在返回委托类（delegate class）——它模仿一个特定的目标实体接口——的一个获取方法中映射@ManyToOne时。

第二个可选的元素是@JoinColumn。如果没有声明外键列的名称，Hibernate会自动使用目标实体名称和目标实体的数据库标识符属性名称的一个组合。换句话说，如果没有添加@JoinColumn注解，外键列的默认名称则为item加id，用一条下划线隔开。然而，因为要使外键列为NOT NULL，所以无论如何都始终要用注解设置nullable = false。如果用Hibernate Tools生成Schema，@ManyToOne中的optional="false"属性也会在生成的列中导致一个NOT NULL约束。

这在以前也很容易。最重要的是，要意识到你可以不用任何其他东西而编写完整的应用程序。(可能经常用共享的主键一对一映射，如第7章所述。)你无需映射这个类关联的另一侧，并且可能已经映射了出现在SQL Schema（外键列）中的任何东西。如果需要为其生成特定Bid的Item实例，就利用创建过的实体关联调用aBid.getItem()。另一方面，如果需要已经对一件货品所做的所有出价，可以编写查询（用Hibernate支持的任何一种语言）。

使用像Hibernate这样的完全的ORM工具，原因之一当然在于不想编写这种查询。

6.4.3 使关联双向

为了不通过显式查询就能够为一件特定的货品轻松地抓取所有的出价，那么就要通过持久化对象的网络图进行导航和迭代。这么做最方便的方法是在Item：anItem.getBids().iterator()中使用集合属性。（注意，映射实体引用的集合还有其他很好的理由，但是不多。始终努力把这种集合映射当作一项特性，而不是必要条件。如果太难，就不要做。）

现在通过使Item和Bid之间的关系变成双向，来映射实体引用的集合。

首先把属性和脚手架代码添加到Item类：

```
public class Item {
    ...

    private Set bids = new HashSet();

    public void setBids(Set bids) {
        this.bids = bids;
    }

    public Set getBids() {
        return bids;
    }

    public void addBid(Bid bid) {
        bid.setItem(this);
        bids.add(bid);
    }
    ...
}
```

可以考虑用addBid()中的代码在对象模型中实现托管关联！（关于这些方法，我们已经在3.2节中讨论过很多。可以复习一下那里的代码示例。）

这个一对多关联的基本映射看起来像这样：

```
<class
    name="Item"
    table="ITEM">
    ...

    <set name="bids">
        <key column="ITEM_ID"/>
        <one-to-many class="Bid"/>
    </set>

</class>
```

如果拿这个与本章前面的集合映射进行比较，会发现你用一个不同的元素<one-to-many>映射了集合的内容。这表明集合没有包含值类型实例，而是包含了对实体实例的引用。现在Hibernate知道如何处理共享的引用和被关联对象的生命周期（它禁用了值类型实例的所有隐式依赖的生命周期）。Hibernate也知道用给集合的表与目标实体类被映射到的表相同——<set>映射不需要table属性。

由<key>元素定义的列映射是BID表的外键列ITEM_ID，即已经在关系的另一侧映射的同一个列。

注意，表Schema并没有改变；它与之前你映射关联的多侧一样。但是，有一点不同：not null="true"不见了。问题是现在有两个不同的单向关联映射到同一个外键列。哪一侧控制这个列呢？

在运行时，同一个外键值有两个不同的内存表示法：Bid的item属性和由Item保存的bids集合的一个元素。假设应用程序修改了关联，例如通过在addBid()方法的这个片段中对货品添加出价：

```
bid.setItem(item);
bids.add(bid);
```

这段代码很好，但是在这种情况下，Hibernate侦测到内存持久化实例的两处变化。从数据库的观点来看，只有一个值必须更新以体现这些变化：BID表的ITEM_ID列。

Hibernate没有透明地侦测到引用同一个数据库列的两处变化，因为此时还没有做任何事来表明这是一个双向的关联。换句话说，已经映射了同一个列两次（在两个映射文件中这么做没有关系），Hibernate始终需要知道这一点，因为它无法自动侦测到这个重复（没有可以处理的合理的默认方法）。

在关联映射中还需要做的一件事，就是使它成为真正的双向关联映射。inverse属性告诉Hibernate，集合是<many-to-one>关联在另一侧的一个镜像：

```
<class
    name="Item"
    table="ITEM">
    ...

    <set name="bids"
         inverse="true">

        <key column="ITEM_ID"/>
        <one-to-many class="Bid"/>

    </set>

</class>
```

在操作两个实例之间的链接时，如果没有inverse属性，Hibernate会试图执行两个不同的SQL语句，这两者更新同一个外键列。通过指定inverse="true"，显式地告诉Hibernate链接的哪一端不应该与数据库同步。在这个例子中，告诉Hibernate它应该把在关联的Bid端所做的变化传播到数据库，忽略只对bids集合所做的变化。

如果只调用anItem.getBids().add(bid)，则没有使任何变化成为持久化！只有当正确地设置了另一侧即aBid.setItem(anItem)时，才能得到想要的东西。这与Java中没有Hibernate时的行为一致：如果关联是双向的，就必须创建在两侧（而不是只在一侧）带有指针的链接。这就是我们为什么推荐像addBid()这样的便利方法的主要原因——它们在系统中负责双向引用，不用容器托管的关系。

注意，Hibernate Schema导出工具始终对SQL DDL的生成忽略关联映射的inverse侧。在这种情况下，BID表中的ITEM_ID外键列就获取了一个NOT NULL约束，因为已经对它做了与非反向

的（noninverse）<many-to-one>映射中一样的声明。

（你可以转换反向端吗？<many-to-one>元素没有inverse属性，但是可以用update="false"和insert="false"映射它，以有效地忽略掉任何UPDATE或者INSERT语句。然后集合端就是非反向的，考虑把它用于外键列的插入或者更新。第7章将会介绍这一点。）

再次通过JPA注解映射这个反向集合侧：

```
public class Item {
    ...

    @OneToMany(mappedBy = "item")
    private Set<Bid> bids = new HashSet<Bid>();

    ...

}
```

mappedBy属性相当于XML映射中的inverse属性；但是，它必须指定目标实体的反向属性。注意，这里不必再次指定外键列（它通过另一侧映射），因此不像XML那么冗长。

你现在有了一个有效的双向的（bidirectional）多对一关联（也可称为双向的一对多关联）。如果想要使它成为一个真正的父/子关系，还缺少最后一个选项。

6.4.4 级联对象状态

父子的概念意味着一个照顾另一个。实际上，这意味着只需要更少的代码行来管理父子之间的关系，因为有些事情可以自动处理。我们来探讨一下选项。

下列代码创建了一个新的Item（我们认为是父）和新的Bid实例（子）：

```
Item newItem = new Item();
Bid newBid = new Bid();

newItem.addBid(newBid); // Set both sides of the association

session.save(newItem);
session.save(newBid);
```

如果正在谈的是真正的父/子关系，对session.save()的第二次调用看起来就是多余的。保留这个想法，并再次回想一下实体和值类型：如果两个类都是实体，它们的实例就有一个完全独立的生命周期。新对象是瞬时的，如果要把它们保存在数据库中，必须使它们变成持久化。如果它们是实体，它们的关系则不影响它们的生命周期。如果Bid是值类型，Bid实例的状态则与它自己的实体的状态相同。然而，在这个例子中，Bid是一个单独的实体，有着它自己完全独立的状态。你有3种选择：

- 你自己负责独立的实例，必要时在Bid对象上执行额外的save()和delete()调用——除了管理关系所需的Java代码之外（从集合等中增加和移除引用）。
- 使Bid类成为值类型（组件）。可以用<composite-element>映射集合，并获得隐式的生命周期。但失去了实体的其他方面，例如对实例可能的共享引用。
- 需要对Bid对象的共享引用吗？目前，特定的Bid实例只被一个Item引用。但是，想象如果User实体也有一个由用户生成的bids集合。为了支持共享引用，必须把Bid映射为实

体。需要共享引用的另一个原因是，完整的CaveatEmptor模型中由Item发出的successfulBid关联。在这种情况下，Hibernate就会提供传播性持久化，它是一个可以启用的特性，用来节省代码行并让Hibernate自动管理关联实体实例的生命周期。

你不想执行超过绝对必要的更多持久化操作，并且不想改变领域模型——需要对Bid实例的共享引用。第三个选项是你要用来简化这个父/子示例的。

1. 传播性持久化

当实例化一个新的Bid并把它添加到Item时，bid应该自动变成持久化。你想要避免的是，使用一个额外的save()操作来显式地使Bid持久化。

为了启用跨关联的这个传播性状态，得把cascade选项添加到XML映射：

```
<class
    name="Item"
    table="ITEM">
    ...

    <set name="bids"
         inverse="true"
         cascade="save-update">

        <key column="ITEM_ID"/>
        <one-to-many class="Bid"/>

    </set>

</class>
```

如果特定的Bid在集合中被持久化的Item引用，cascade="save-update"属性便为Bid实例启用了传播性持久化。

cascade属性是有方向性的：它只应用到关联的一端。也可以把cascade="save-update"添加到Bid映射中的<many-to-one>关联，但是由于出价是在货品之后创建的，这么做并没有什么意义。

JPA也在关联中支持级联实体实例状态：

```
public class Item {
    ...

    @OneToMany(cascade = { CascadeType.PERSIST, CascadeType.MERGE },
               mappedBy = "item")
    private Set<Bid> bids = new HashSet<Bid>();

    ...

}
```

级联选项是你想要它变成传播性的每一个操作（per operation）。对于原生的Hibernate，用cascade="save-update"把save和update操作级联到被关联的实体。Hibernate的对象状态管理始终把这两个东西绑在一起，后面的章节中将会介绍。在JPA中，（几乎）相当的操作是persist和merge。

现在可以在原生的Hibernate中简化链接和保存Item和Bid的代码：

```
Item newItem = new Item();
Bid newBid = new Bid();

newItem.addBid(newBid); // Set both sides of the association

session.save(newItem);
```

bids集合中的所有实体现在也都变成持久化了，就像在每个Bid中手工调用save()时它们的表现一样。利用JPA EntityManager API（相当于Session），代码如下：

```
Item newItem = new Item();
Bid newBid = new Bid();

newItem.addBid(newBid); // Set both sides of the association

entityManager.persist(newItem);
```

目前不必担心update和merge操作；我们会在本书的后面回到这个话题。

常见问题 cascade在inverse中有什么作用？许多Hibernate新用户会问这个问题。答案很简单：cascade属性与inverse属性无关。它们经常出现在同一个集合映射中。如果映射实体的集合为inverse="true"，就为双向关联映射控制了SQL的生成。这个提示告诉Hibernate你映射了同一个外键列两次。另一方面，级联被用作一项便利的特性。如果决定从实体关系的一侧级联操作到被关联的实体，就省去了手工管理另一侧的状态所需的代码行。假设对象状态变成了传播性，你不仅可以级联实体的集合中的状态，还可以级联所有的实体关联映射中的状态。cascade和inverse有相似之处，它们都不出现在值类型的集合或者任何其他值类型映射中。这些规则隐含在值类型的特性中。

现在讲完了吗？嗯，可能还没有。

2. 级联删除

利用前一个映射，Bid和Item之间的关联非常松散的。目前为止，我们仅仅把使东西持久化当作一种传播性状态。那么如何删除呢？

货品的删除意味着这个货品的所有出价都删除了，这似乎有道理。事实上，这就是UML图中复合（图6-5黑色菱形处）的意思。利用当前的级联操作，你必须编写以下代码来使它发生：

```
Item anItem = // Load an item

// Delete all the referenced bids
for ( Iterator<Bid> it = anItem.getBids().iterator();
      it.hasNext(); ) {

    Bid bid = it.next();

    it.remove();                // Remove reference from collection
    session.delete(bid);        // Delete it from the database
}

session.delete(anItem);         // Finally, delete the item
```

首先通过迭代集合移除对bids的引用。删除数据库中的所有Bid实例。最后，删除Item。迭

代和移除集合中的引用似乎没有必要；毕竟，无论如何最后都会删除Item。如果能保证没有其他对象（或者任何其他表中的行）保存着对这些出价的引用，就可以使删除变成传播性。

Hibernate（和JPA）为此提供了一个级联选项。可以给delete操作启用级联：

```
<set name="bids"
     inverse="true"
     cascade="save-update, delete">
...
```

在JPA中级联的操作称作remove：

```
public class Item {
    ...
    @OneToMany(cascade = { CascadeType.PERSIST,
                           CascadeType.MERGE,
                           CascadeType.REMOVE },
               mappedBy = "item")
    private Set<Bid> bids = new HashSet<Bid>();
    ...
}
```

删除一个货品和它全部出价的相同代码被缩减如下，在Hibernate中或者用JPA：

```
Item anItem = // Load an item
session.delete(anItem);
entityManager.remove(anItem);
```

现在delete操作被级联到了集合中引用的所有实体。你不再需要担心从集合中移除，以及用手工一个个地删除那些实体了。

来思考进一步的复杂因素。你可能有对Bid实例的共享引用。就像前面建议的，User可以有对他们生成的Bid实例的一个引用集合。不先移除这些引用，就无法删除货品和它的所有出价。如果试图提交这个事务，就可能得到一个异常，因为外键约束可能遭到了破坏。

你必须追踪指针。这个过程可能不好看，如下列代码所示，它在删除出价以及最后删除货品之前，从有引用的所有用户中移除了所有引用：

```
Item anItem = // Load an item

// Delete all the referenced bids
for ( Iterator<Bid> it = anItem.getBids().iterator();
      it.hasNext(); ) {
    Bid bid = it.next();

    // Remove references from users who have made this bid
    Query q = session.createQuery(
      "from User u where :bid in elements(u.bids)"
    );
    q.setParameter("bid", bid);
    Collection usersWithThisBid = q.list();

    for (Iterator itUsers = usersWithThisBid.iterator();
         itUsers.hasNext();) {
         User user = (User) itUsers.next();
```

```
            user.getBids().remove(bid);
        }
    }

    session.delete(anItem);
    // Finally, delete the item and the associated bids
```

显然，你并不想要额外的查询（事实上是许多查询）。但是，在网络对象模型中，如果想要正确地设置指针和引用，除了像这样执行代码之外别无选择——没有持久化的垃圾收集器或者其他的自动机制。没有Hibernate级联选项可以帮忙；你必须在最终删除实体之前，追踪对它的所有引用。

（这并非全部的真相：因为表示从User到Bid关联的BIDDER_ID外键列是在BID表中，如果BID表中的一行被删除了，这些引用在数据库级别中也被自动移除。这不影响已经出现在当前工作单元内存中的任何对象，而且如果BIDDER_ID被映射到一张不同的（中间）表，它也不起作用。为了确保所有引用和外键列为空，必须在Java中追踪指针。）

另一方面，如果没有对实体的共享引用，就应该重新考虑映射，并把bids映射为集合组件（Bid映射为<composite-element>）。通过<idbag>映射，甚至表看起来也一样：

```
<class
    name="Item"
    table="ITEM">
    ...

    <idbag name="bids" table="BID">
        <collection-id type="long" column="BID_ID">
            <generator class="sequence"/>
        </collection-id>
        <key column="ITEM_ID" not-null="true"/>

        <composite-element class="Bid">
            <parent name="item"/>
            <property .../>
            ...
        </composite-element>

    </idbag>

</class>
```

Bid不再需要单独的映射。

如果真的想要使它成为一对多的实体关联，那么可以使用Hibernate提供的另一个你可能感兴趣的便利选项。

3. 启用孤儿删除

现在要阐述的级联选项有点难以理解。如果你理解了前一节的讨论，就应该不成问题了。

想象你要从数据库中删除一个Bid。注意在这个例子中没有删除父（Item）。目标是移除BID表中的一个行。看看以下代码：

```
anItem.getBids().remove(aBid);
```

如果集合让Bid映射为组件的集合，就像前一节一样，这段代码就触发了几项操作：

第十二章　西方能力本位职业教育与培训模式研究

能力本位的教育与培训(CBET)是当今世界各国职业教育与培训改革的方向。但是,不同国家对CBET的理解是不同的,这一差异不仅表现在能力观上,而且深入到了课程、教学与评价等各个层面,这样,就形成了不同的CBET模式。本章将从能力观、能力标准、课程、教学、评价几个方面详细剖析CBET几种模式之间的差异。

第一节　CBET的能力观

所谓能力观,就是人们关于"能力是什么"这个问题的基本看法。在心理学上对能力的概念早已有一个基本一致的定义,但在这里我们是把它作为一个教育概念来加以讨论。作为一个教育概念,人们对能力的看法往往殊异。然而在CBET体系中,对于能力的概念是不能不明的,因为能力观是CBET体系中最基本的核心要素,是构建CBET体系的基础。[①]不同的能力观,将产生不同的能力标准;能力标准不同,相应的课程模式、教学方式以及评价机制也就各异。当然,我们这里的探讨目的不在于、也不可能最终获得一个共识的能力概念,但是我们希望通过对观点纷纭的能力观的分析,能够有助于人们根据他们所处的环境与条件,来选择和确定与之相适宜的能力观,进而在此基础上构建一种最优化的CBET体系。

一、能力本位运动中对能力的不同理解

能力本位运动在北美地区兴起之初,由于产业界的代表在教育

---

① Ruth Nickse (Ed.) (1981) Competency-Based Education: Beyond Minimum Competency Testing. New York: Teachers College Press, p5.

```
<entity-mappings>

    <entity class="auction.model.Item" access="FIELD">
        ...
        <one-to-many name="bids" mapped-by="item">
            <cascade>
                <cascade-persist/>
                <cascade-merge/>
                <cascade-remove/>
            </cascade>
         </one-to-many>
    </entity>

    <entity class="auction.model.Bid" access="FIELD">
        ...
        <many-to-one name="item">
            <join-column name="ITEM_ID"/>
        </many-to-one>
    </entity>

</entity-mappings>
```

注意，在这个例子中，Hibernate扩展对级联孤儿删除是不起作用的。

6.5 小结

你可能被本章介绍的所有新概念搞得有点晕头转向了。你可能不得不多读几遍，并且我们鼓励你去尝试代码（并观察SQL日志）。本章介绍的许多策略和技术都是ORM的主要概念。如果你掌握了集合映射，并且一旦已经映射了第一个父/子实体关联，就克服了最大的困难。你将已经能够创建完整的应用程序了！

表6-1概括了本章讨论过的Hibernate和Java Persistence相关概念之间的区别。

表6-1 第6章的Hibernate和JPA对照表

Hibernate Core	Java Persistence和EJB 3.0
Hibernate提供支持对Set、List、Map、Bag、标识符bag和数组的映射。支持所有的JDK集合接口，并且给定制的持久化集合提供了扩展点	支持标准的持久化Set、List、Map和bag
支持值类型和组件的集合	值类型和可嵌入对象的集合需要Hibernate Annotations
支持父/子实体关系，包含每个操作在关联中的传播性状态级联	可以映射实体关联，并启用每个操作在关联中的传播性状态级联
支持孤儿实体实例的自动删除	孤儿实体实例的自动删除需要Hibernate Annotations

本章只涵盖了实体关联选项的一小部分。第7章要深入探讨的剩余选项更为少见，或者说是我们刚刚描述过的这些技术的变形。

第7章

高级实体关联映射

本章内容

- 映射一对一和多对一的实体关联
- 映射一对多和多对多的实体关联
- 多态实体关联

使用关联（association）一词时，总是指实体之间的关系。前一章示范了一个单向的多对一关联，使它变成双向，最后把它变成一个父/子关系（包含级联选项的一对多和多对一关联）。

之所以在单独一章里讨论更高级的实体映射，原因之一在于它们中大部分都比较少见，或者至少是可选的。

只使用组件映射和多对一（偶尔为一对一）的实体关联是绝对有可能的。你甚至可以编写复杂的应用程序而不映射任何集合！当然，有效而轻松地访问持久化数据，例如通过迭代集合，就是使用完全的ORM而不是简单的JDBC查询服务的原因之一了。但是有些怪异的映射特性大部分时候应该谨慎使用，甚至尽量不用。

我们会在本章指出推荐使用的和可选的映射技术，就像介绍如何映射包含各种多样性的实体关联（不管是否带有集合）。

7.1 单值的实体关联

让我们从一对一的实体关联开始。

第4章曾讨论过，User和Address之间的关系（user有billingAddress、homeAddress和shippingAddress）最好用<component>映射来表示。这通常是表示一对一关系最简单的方法，因为在这种情况下，生命周期通常是依赖的，在UML中，要么是一个聚合，要么是一个组合。

但是如果想要Address的专用表，并且把User和Address都映射为实体时该怎么办？这个模型的好处之一在于共享引用的可能性——另一个实体类（如Shipment）也可以有一个对特定Address实例的引用。如果User有对该实例的引用，作为它们的shippingAddress，那么Address实例就必须支持共享引用，并且需要有自己的同一性。

在这种情况下，User和Address类就都有一个真正的一对一关联了。看看图7-1中修改过的

类图。

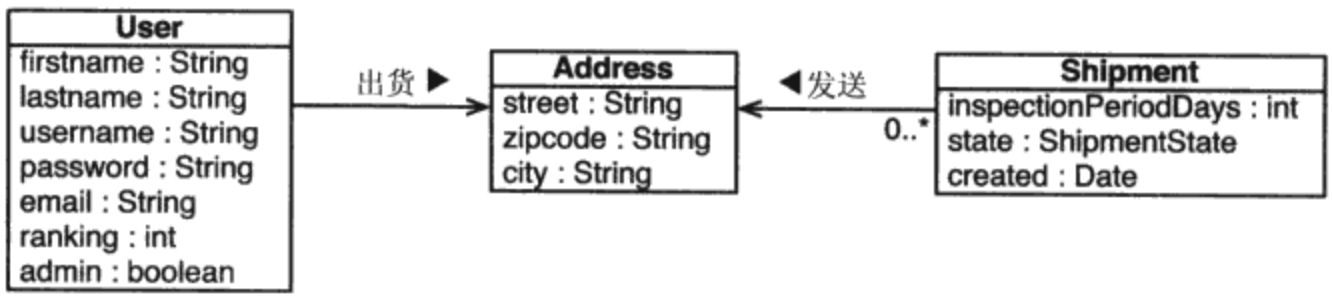


图7-1 作为实体的Address，包含了两个对相同实例引用的关联

第一个变化是Address类作为独立实体的映射：

```
<class name="Address" table="ADDRESS">
    <id name="id" column="ADDRESS_ID">
        <generator .../>
    </id>
    <property name="street" column="STREET"/>
    <property name="city" column="CITY"/>
    <property name="zipcode" column="ZIPCODE"/>
</class>
```

假设你能够轻松地利用注解创建相同的映射，或者通过标识符属性把Java类变为实体——这是必须做的唯一变化。

现在来创建从其他实体到这个类的关联映射。有几种选择，首要的是主键一对一关联。

7.1.1 共享的主键关联

由主键关联而相关的两张表中的行共享相同的主键值。这种方法的主要困难在于，确保被关联的实例在保存对象时分配到了相同的主键值。在尝试解决这个问题之前，先看看主键关联是如何映射的。

1. 用XML映射主键关联

把实体关联映射到共享主键实体的XML映射元素是<one-to-one>。首先在User类中需要一个新属性：

```
public class User {
    ...
    private Address shippingAddress;
    // Getters and setters
}
```

接下来，在User.hbm.xml中映射关联：

```
<one-to-one name="shippingAddress"
            class="Address"
            cascade="save-update"/>
```

你给这个模型添加了一个固有的级联选项：如果User实例变成持久化，通常它的shippingAddress也要变成持久化。因而，下面是保存这两个对象所需要的所有代码：

```
User newUser = new User();
Address shippingAddress = new Address();

newUser.setShippingAddress(shippingAddress);

session.save(newUser);
```

Hibernate把一行插入到USERS表中，把另一行插入到ADDRESS表中。但是慢着，这不起作用！Hibernate怎么可能知道ADDRESS表中的记录要获取与USERS行相同的主键值？在本节开始时，我们有意不在Address的映射中介绍任何主键生成器。

你需要启用一个特殊的标识符生成器。

2．外标识符生成器

如果Address实例被保存，它就需要获取User对象的主键值。你无法启用一个常规的标识符生成器，假设是数据库序列。用于Address的这个特殊的foreign标识符生成器，必须知道从哪里获取正确的主键值。

创建Address和User之间的这个标识符绑定，其第一步是双向关联。把一个新的user属性添加到Address属性：

```
public class Address {
    ...
     private User user;
    // Getters and setters
}
```

在Address.hbm.xml中映射Address的这个新user属性：

```
<one-to-one name="user"
            class="User"
            constrained="true"/>
```

这个映射不仅使关联变成双向，而且通过constrained="true"，添加了把ADDRESS表的主键链接到USERS表主键的外键约束。换句话说，数据库保证了ADDRESS行的主键引用有效的USERS主键。（作为附带作用，Hibernate现在也可以在加载出货地址时启用用户的延迟加载。外键约束意味着用户必须对于特定的出货地址而存在，因此代理可以不用命中数据库而启用。如果没有这项约束，Hibernate就必须命中数据库去发现是否有对应于这个地址的用户；代理也将成为多余的了。后面的章节会回到这个话题。）

现在可以给Address对象使用特殊的foreign标识符生成器了：

```
<class name="Address" table="ADDRESS">

    <id name="id" column="ADDRESS_ID">
        <generator class="foreign">
            <param name="property">user</param>
        </generator>
    </id>
    ...
    <one-to-one name="user"
                class="User"
                constrained="true"/>

</class>
```

这个映射乍看起来似乎很奇怪。解读如下：当Address被保存时，从user属性中提取主键值。user属性是对User对象的一个引用；因而，插入的主键值与这个实例的主键值相同。看看图7-2中的表结构。

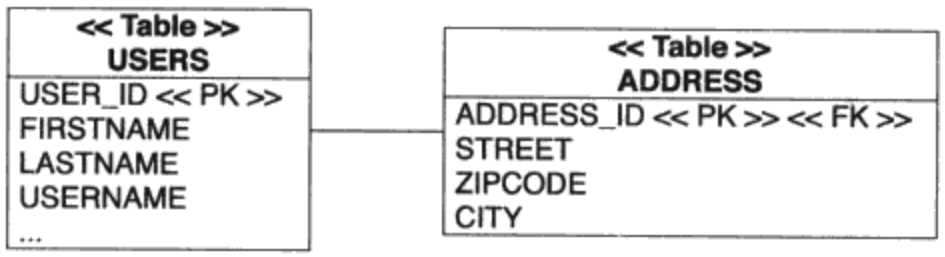


图7-2 USERS和ADDRESS表有相同的主键

保存这两个对象的代码现在必须考虑双向的关系，并且最终生效：

```
User newUser = new User();
Address shippingAddress = new Address();

newUser.setShippingAddress(shippingAddress);
shippingAddress.setUser(newUser);              // Bidirectional

session.save(newUser);
```

让我们利用注解完成同样的事情。

3. 利用注解的共享主键

JPA用@OneToOne注解支持一对一的实体关联。要映射User类中shippingAddress的关联为共享主键关联，还需要@PrimaryKeyJoinColumn注解：

```
@OneToOne
@PrimaryKeyJoinColumn
private Address shippingAddress;
```

在共享主键上创建单向的一对一关联就只需要这些。注意，如果通过复合主键映射，就要用@PrimaryKeyJoinColumns（复数）代替。在JPA XML描述符中，一对一的映射看起来像这样：

```
<entity-mappings>

    <entity class="auction.model.User" access="FIELD">
        ...
        <one-to-one name="shippingAddress">
            <primary-key-join-column/>
        </one-to-one>
    </entity>

</entity-mappings>
```

JPA规范没有包括处理共享主键生成问题的标准方法，这意味着在保存Address实例的标识符值（到被链接的User实例的标识符值）之前，你要负责正确地设置它。Hibernate有一个扩展注解，用于可以通过Address实体使用的定制标识符生成器（就像用XML格式一样）：

```
@Entity
@Table(name = "ADDRESS")
public class Address {
```

```
    @Id @GeneratedValue(generator = "myForeignGenerator")
    @org.hibernate.annotations.GenericGenerator(
        name = "myForeignGenerator",
        strategy = "foreign",
        parameters = @Parameter(name = "property", value = "user")
    )
    @Column(name = "ADDRESS_ID")
    private Long id;

    ...
    private User user;
}
```

共享的主键一对一关联并不稀奇，但是相对少见。在许多模式中，对一（to-one）的关联都用外键字段和唯一约束来表示。

7.1.2　一对一的外键关联

不共享主键，而是两行可以有一个外键关系。一张表有着引用被关联表的主键的一个外键列。[这个外键约束的源和目标甚至可以是相同的表：称作自引用（self-referencing）关系。]

我们来改变从User到Address的映射。现在不用共享主键，而是在USERS表中添加一个SHIPPING_ADDRESS_ID列：

```
<class name="User" table="USERS">

    <many-to-one name="shippingAddress"
                 class="Address"
                 column="SHIPPING_ADDRESS_ID"
                 cascade="save-update"
                 unique="true"/>

</class>
```

在XML中，这个关联的映射元素是<many-to-one>——而不是你可能期待的<one-to-one>。原因很简单：你不在乎关联的目标端是什么，因此可以像对待没有多端的对一关联一样对待它。你所要的就是表达“这个实体有一个属性，该属性是对另一个实体的实例的引用”，并使用一个外键字段表示这个关系。这个映射的数据库模式如图7-3所示。

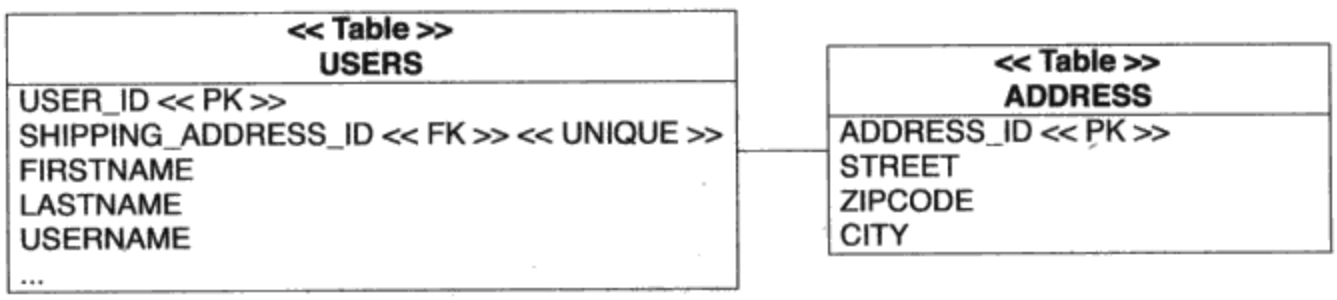


图7-3　USERS和ADDRESS之间的一对一外键关联

额外的约束强制这个关系为真正的一对一。通过使SHIPPING_ID列变成unique，声明一个特定的地址最多只能被一位用户作为出货地址使用。这不像来自共享主键关联（它允许一个特定的地址最多被一位用户引用）的保证那么有力。利用几个外键列（假设你也有唯一的HOME_

ADDRESS_ID和BILLING_ADDRESS_ID），可以多次引用相同的地址目标行。但是无论如何，两位用户无法把同一个地址用于同一目的。

我们来把从User到Address的关联变成双向。

1．反向属性引用

最后一个外键关联是用<many-to-one>从User映射到Address，并且unique约束保证了想要的多样性。可以在Address端添加什么映射元素使这个关联变成双向，以便在Java领域模型中从Address到User的访问成为可能呢？

在XML中，用属性引用（property reference）属性来创建一个<one-to-one>映射：

```
<one-to-one name="user"
            class="User"
            property-ref="shippingAddress"/>
```

你告诉Hibernate，Address类的user属性是关联另一端的一个属性的反向。现在可以调用anAddress.getUser()来访问给定出货地址的用户。没有额外的列或者外键约束；Hibernate替你管理这个指针。

应该使这个关联成为双向吗？跟往常一样，这由你决定，并取决于你是否需要在应用程序代码中的那个方向上导航对象。在这种情况下，我们或许可以推断双向关联没有多大意义。如果调用anAddress.getUser()，就等于在说“请把以这个地址作为出货地址的用户给我”，这不是一个非常合理的请求。我们建议，基于外键的一对一关联，以及在外键列中包含的唯一约束——通常要在另一端没有映射的情况下给予最好表示。

让我们通过注解重复相同的映射。

2．通过注解映射外键

JPA映射注解也支持基于外键列的实体之间的一对一关系。与本章前面的映射相比，主要区别在于用@JoinColumn代替了@PrimaryKeyJoinColumn。

首先，这是从User到Address的对一映射，在SHIPPING_ADDRESS_ID外键列中包含唯一约束。但是，它不用@ManyToOne注解，而是需要@OneToOne注解：

```
public class User {
    ...

    @OneToOne
    @JoinColumn(name="SHIPPING_ADDRESS_ID")
    private Address shippingAddress;

    ...
}
```

Hibernate现在将用唯一约束强制多样性。如果要使这个关联变成双向的，Address中还需要@OneToOne映射：

```
public class Address {
    ...

    @OneToOne(mappedBy = "shippingAddress")
    private User user;
```

```
    ...
}
```

mappedBy属性的作用与XML映射中的property-ref一样：关联的一个简单的反向声明，就在目标实体端指定了一种属性。

JPA XML描述器中与之相当的映射如下：

```
<entity-mappings>

    <entity class="auction.model.User" access="FIELD">
        ...
        <one-to-one name="shippingAddress">
            <join-column name="SHIPPING_ADDRESS_ID"/>
        </one-to-one>
    </entity>

    <entity class="auction.model.Address" access="FIELD">
        ...
        <one-to-one name="user" mapped-by="shippingAddress"/>
    </entity>

</entity-mappings>
```

你现在已经完成了两个基础的单端关联映射：第一个带有共享主键，第二个带有外键引用。我们想要讨论的最后一个选项更怪异一点：在另外一张表的帮助下映射一对一的关联。

7.1.3　用联结表映射

暂时放下复杂的CaveatEmptor模型，考虑一种不同的场景。想象你必须对表示一家公司的办公室配置规划的数据模式进行建模。一般的实体包括在办公桌前工作的人们。这似乎有道理：桌子可能被闲置着，没有分配给任何人员。另一方面，员工可能在家办公，结果也一样。你正在处理的就是Person和Desk之间可选的一对一关联。

如果应用前几节讨论的映射技术，可能会得出如下结论：Person和Desk被映射到两张表，其中一张（假设是PERSON表）有一个外键列，引用另一张带有额外唯一约束（以便两个人不会被分配到同一张桌子）的表（例如ASSIGN_DESK_ID）。如果外键列可为空，这个关系就是可选的。

第二种想法，你意识到人员和桌子之间的分配需用另一张表来表示ASSIGNMENT。在目前的设计中，这张表只有两个列：PERSON_ID和DESK_ID。这些外键列的多样性通过unique约束在这两个列上都得到了强制——特定的人员和桌子只能被分配一次，并且这样的分配只能有一次。

似乎也有可能某一天你会需要扩展这个Schema，并添加列到ASSIGNMENT表，例如当人员被分配到一张桌子时的日期。但是，只要不是这种情况，就可以利用ORM来隐藏中间的表，并创建仅仅两个类之间一对一的Java实体关联。（一旦ASSIGNMENT中引入了额外的列，这种情况就会完全改变。）

在CaveatEmptor中，这种可选的一对一关系在哪里呢?

1. CaveatEmptor用例

再次考虑CaveatEmptor中的Shipment实体，并讨论它的用途。卖主和买主在CaveatEmptor中通过在拍卖中标价和出价进行交互。货物的出货似乎在应用程序的范围之外；卖主和买主在拍卖结束之后认同一种出货和付款的方法。他们可以离线（即在CaveatEmptor之外）进行这些事。另一方面，可以在CaveatEmptor中提供一种额外的*有条件转让契约服务*（escrow service）。一旦拍卖完成，卖主可以利用这项服务创建可追踪的出货。买主可以支付拍卖货品的价钱给托管人（你），并且你要通知卖主钱已到账。一旦货物抵达，买主接受了，你就要把钱转给卖主。

如果你曾经参与过重大价值的在线拍卖，可能已经使用过这么一种有条件转让契约服务。但是在CaveatEmptor中还要更多的服务。不仅要对已完成的拍卖提供信任服务，还要允许用户给他们在拍卖之外（即CaveatEmptor之外）进行的任何交易创建可追踪的信任出货。

这种场景需要Shipment实体，包含可选的关联到Item的一对一关联。请看图7-4中这个领域模型的类图。

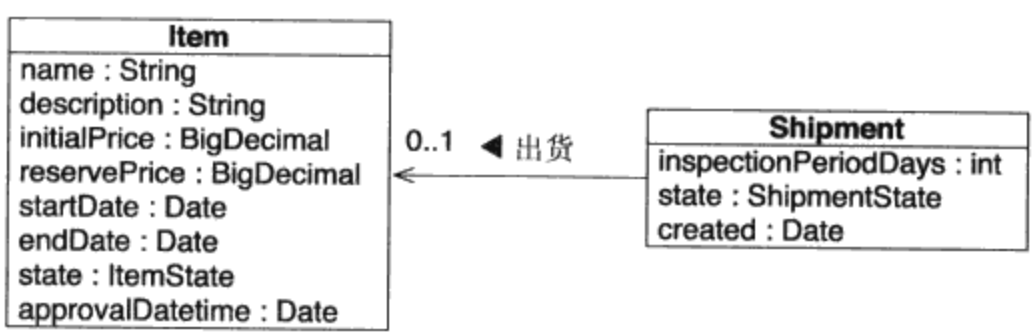


图7-4 出货与单件拍卖货品有可选的链接

在数据库Schema中，添加了一个中间的链接表，称作ITEM_SHIPMENT。这张表中的行表示在一次拍卖的上下文中产生的Shipment。这些表如图7-5所示。

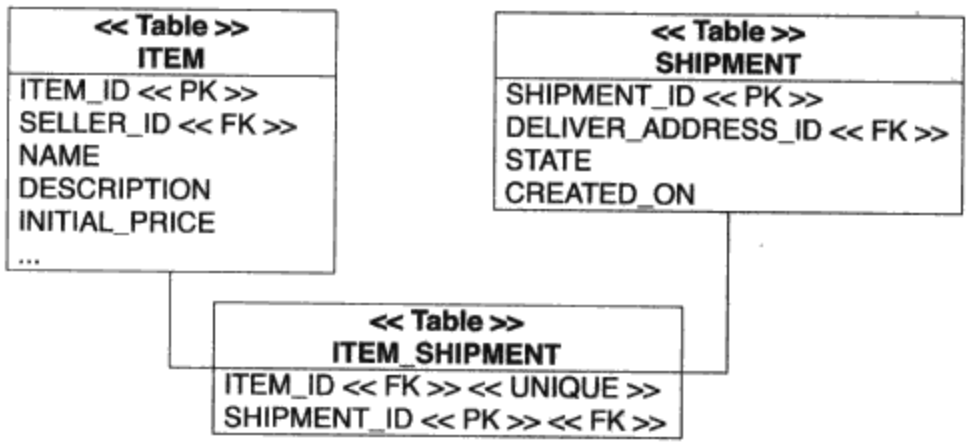


图7-5 被映射到联结表的可选的一对多关系

现在把两个类映射到三张表：先用XML映射，然后通过注解映射。

2. 用XML映射联结表

表示从Shipment到Item关联的属性称作auction：

```
public class Shipment {

    ...
```

```
    private Item auction;
    ...
    // Getter/setter methods
}
```

因为必须用一个外键列映射这个关联，因此在XML中你需要<many-to-one>映射元素。然而，外键列不在SHIPMENT表中，而在ITEM_SHIPMENT联结表中。通过<join>映射元素把它移到那里。

```
<class name="Shipment" table="SHIPMENT">

    <id name="id" column="SHIPMENT_ID">...</id>

    ...

    <join table="ITEM_SHIPMENT" optional="true">
        <key column="SHIPMENT_ID"/>
        <many-to-one name="auction"
                     column="ITEM_ID"
                     not-null="true"
                     unique="true"/>
    </join>

</class>
```

联结表有两个外键列：SHIPMENT_ID（引用SHIPMENT表的主键）和ITEM_ID（引用ITEM表）。ITEM_ID列是唯一的；特定的货品正好可以被分配给一批出货。由于联结表的主键是SHIPMENT_ID，它使得这个列也变成唯一，这样就保证了Shipment和Item之间一对一的多样性。

通过在<join>映射中设置optional="true"，告诉Hibernate它应该只有当这个映射分组的属性为非空时，才把行插入到联结表中。但是如果必须插入行（因为调用了aShipment.setAuction(anItem)），ITEM_ID列中的NOT NULL约束就得到了应用。

可以在另一端使用相同的方法，把这个关联映射为双向。然而，可选的一对一关联大多数时候都是单向的。

JPA也对实体支持作为二级表（secondary table）的关联联结表。

3. 通过注解映射二级联结表

可以通过注解把可选的一对一关联映射到一个中间的联结表：

```
public class Shipment {

    @OneToOne
    @JoinTable(
        name="ITEM_SHIPMENT",
        joinColumns = @JoinColumn(name = "SHIPMENT_ID"),
        inverseJoinColumns = @JoinColumn(name = "ITEM_ID")
    )
    private Item auction;
    ...
    // Getter/setter methods
}
```

不必指定SHIPMENT_ID列，因为它自动被认为是联结列；它是SHIPMENT表的主键列。

另一种方法，可以把JPA实体的属性映射到不止一张表，如8.1.3节所述。首先，需要给实体

声明二级表：

```
@Entity
@Table(name = "SHIPMENT")
@SecondaryTable(name = "ITEM_SHIPMENT")
public class Shipment {

    @Id @GeneratedValue
    @Column(name = "SHIPMENT_ID")
    private Long id;

    ...
}
```

注意，@SecondaryTable注解也支持用属性声明外键列名——相当于前面用XML的<key column="..."/>和@JoinTable中的joinColumn(s)。如果没有指定，就使用实体的主键列名——在这个例子中，还是SHIPMENT_ID。

auction属性映射是一个@OneToOne；并且和之前一样，引用ITEM表的外键列被移到中间的二级表：

```
...
public class Shipment {
    ...
    @ManyToOne
    @JoinColumn(table = "ITEM_SHIPMENT", name = "ITEM_ID")
    private Item auction;
}
```

用于目标@JoinColumn的表被显式地指定。为什么要使用这种方法而不是（更简单的）@JoinTable策略呢？如果不止一个属性而是多个属性必须被移到二级表的时候，给实体声明二级表就很有用。我们没有使用Shipment和Item的很好示例，但是如果你的ITEM_SHIPMENT表有额外的列，把这些列映射到Shipment实体的属性可能有用。

这样就结束了我们对一对一关联映射的讨论。总而言之，如果两个实例中有一个似乎更加重要，并且可以表现得像主键源一样，就使用共享的主键关联。在所有其他的情况下都使用外键关联，并且当一对一关联为可选时，就使用被隐藏的中间联结表。

现在来关注多值的实体关联，包括一对多以及最后是多对多映射的更多选项。

7.2 多值的实体关联

多值（many-valued）的实体关联就其定义而言，是指实体引用的一个集合。6.4节中映射过其中一种。父实体实例有一个对多个子对象的引用集合——因而，是一对多。

一对多关联是涉及集合的一种最重要的实体关联。如果简单的双向多对一或一对多能够完成任务时，目前为止我们并不鼓励使用更加怪异的关联方式。多对多关联始终可以表示为对中间类的两个多对一关联。这个模型通常更易于扩展，因此我们趋向于不在应用中使用多对多关联。也要记住：如果不想的话，你不必映射实体的任何集合；你可以始终编写显式查询来代替通过迭代的直接访问。

如果你决定要映射实体引用的集合，我们现在就来讨论几种选项以及更复杂的情况（包括多

对多的关系）。

7.2.1　一对多关联

前面映射过的父/子关系是一个双向的关联，通过<one-to-many>和<many-to-one>映射。这个关联的多（many）端在Java中用Set实现；Item类中有一个bids的集合。

来重新思考这个映射，并关注一些特殊的情况。

1. 考虑bag

对双向的一对多关联使用<bag>映射而不是set（集）是可能的。为什么要这什么做呢？

包（bag）有着可以用于双向一对多实体关联的所有集合的最有效的性能特征（换句话说，如果集合端是is inverse="true"的话）。默认情况下，Hibernate中的集合只有当它们在应用程序中第一次被访问时才被加载。因为bag不必维持其元素的索引（如list），或者检查重复的元素（如set），可以添加新元素给bag，而不触发加载。如果要映射一个可能很大的实体引用的集合，这就是一项重要的特性。另一方面，无法同时即时抓取bag类型的两个集合（例如，如果Item的bids和images都是一对多的bag）。我们会在13.1节中回到抓取策略的主题。一般来说，对于一对多的关联，我们认为bag是它最好的反向集合。

为了把双向的一对多关联映射为bag，必须用Collection和ArrayList实现替换Item持久化类中bids集合的类型。对Item和Bid之间关联的映射本质上保持不变：

```
<class name="Bid"
       table="BID">
    ...
    <many-to-one name="item"
                 column="ITEM_ID"
                 class="Item"
                 not-null="true"/>

</class>
<class name="Item"
       table="ITEM">
    ...
    <bag name="bids"
         inverse="true">
        <key column="ITEM_ID"/>
        <one-to-many class="Bid"/>
    </bag>

</class>
```

重新命名<set>元素为<bag>，没有进行其他的改变。甚至表也是一样的：BID表有ITEM_ID外键列。在JPA中，所有Collection和List属性都被认为具有包语义，因此下列代码相当于XML映射：

```
public class Item {
    ...

    @OneToMany(mappedBy = "item")
```

```
    private Collection<Bid> bids = new ArrayList<Bid>();
    ...
}
```

bag也允许重复元素，这是你之前映射过的set所没有的。结果表明，这与本例无关，因为重复意味着你已经几次把一个特定的引用添加到相同的Bid实例。不要在应用程序代码中这么做。但是即使几次把相同的引用添加到这个集合，Hibernate也会忽略它——它被反向映射。

2. 单向和双向的列表

如果你需要真正的列表来保存元素在集合中的位置，就必须在另一列保存该位置。对于一对多的映射，这也意味着应该把Item中的bids属性改为List，并通过ArrayList初始化变量（或者如果不想把这种行为公开给类的客户端，就保留来自前一节的Collection接口）。

在Item的映射中，保存对Bid实例引用的位置的额外列是BID_POSITION：

```
<class name="Item"
       table="ITEM">
    ...
    <list name="bids">
        <key column="ITEM_ID"/>
        <list-index column="BID_POSITION"/>
        <one-to-many class="Bid"/>
    </list>
</class>
```

目前为止，这些似乎都很简单；你已经把集合映射改为<list>，并添加了<list-index>列BID_POSITION到集合表（在这个例子中是BID表）。用图7-6中所示的表进行检验。

BID

BID_ID	ITEM_ID	BID_POSITION	AMOUNT	CREATED_ON
1	1	0	99.00	19.04.08 23:11
2	1	1	123.00	19.04.08 23:12
3	2	0	433.00	20.04.08 09:30

图7-6 在列表集合中保存每个出价的位置

这个映射并没有真正完成。考虑ITEM_ID外键列：它为NOT NULL（出价必须引用货品）。第一个问题在于没有在映射中指定这个约束。而且，由于这个映射是单向的（集合端为非反向），所以只好假设没有对面端被映射到同一个外键列（可以在那里声明这个约束）。需要把not-null="true"属性添加到集合映射的<key>元素中：

```
<class name="Item"
       table="ITEM">
    ...
    <list name="bids">
        <key column="ITEM_ID" not-null="true"/>
        <list-index column="BID_POSITION"/>
        <one-to-many class="Bid"/>
    </list>
</class>
```

注意，属性必须在<key>映射中，而不是在一个可能嵌套的<column>元素中。每当有实体引用的非反向集合（大多数时候是包含list、map或者array的一对多关联），并且目标表中的外键联结列不可为空时，就需要让Hibernate知道这些。Hibernate需要这个提示，以便正确地处理INSERT和UPDATE语句，避免约束冲突。

用Bid的item属性使这个变成双向。如果理解了前面章节中的例子，可能会想要在ITEM_ID外键列上添加<many-to-one>，使这个关联变成双向的，并在集合中启用inverse="true"。记住，Hibernate忽略反向集合的状态！但是这一次，集合包含了正确地更新数据库所需的信息：它的元素的位置。如果只有每个Bid实例的状态被认为是同步的，集合又是反向的并且被忽略，那么Hibernate就没有值给BID_POSITION列了。

如果通过被索引的集合映射双向的一对多实体关联（映射和数组也是这样），就必须转换反向端。无法使被索引的集合变成inverse="true"。集合变成了负责状态同步，并且一（one）端Bid必须变成反向。然而，多对一映射没有inverse="true"，因此需要在<many-to-one>中模拟这一属性：

```
<class name="Bid"
       table="BID">
    ...
    <many-to-one name="item"
                 column="ITEM_ID"
                 class="Item"
                 not-null="true"
                 insert="false"
                 update="false"/>
</class>
```

设置insert和update为false具有预期的效果。如前所述，这两个属性一起使用，实际上使属性变成了只读（read-only）。关联的这一端因此被任何写操作忽略，当内存状态与数据库同步时，集合的状态（包括元素的索引）就是相关的状态。你已经转换了关联的反向／非反向端，如果从set或者bag转换到list（或者任何其他被索引的集合），这是个必要的条件。

JPA中的等价物（双向的一对多映射中的被索引集合）如下：

```
public class Item {
    ...

    @OneToMany
    @JoinColumn(name = "ITEM_ID", nullable = false)
    @org.hibernate.annotations.IndexColumn(name = "BID_POSITION")
    private List<Bid> bids = new ArrayList<Bid>();

    ...

}
```

这个映射为非反向的，因为没有出现mappedBy属性。由于JPA不支持持久化的被索引列表（只有在加载时用@OrderBy进行排序），需要给索引支持添加一个Hibernate扩展注解。以下是Bid中关联的另一端：

```
public class Bid {
    ...

    @ManyToOne
    @JoinColumn(name = "ITEM_ID", nullable = false,
                updatable = false, insertable = false)
    private Item item;

    ...

}
```

现在再讨论一种包含一对多关系的场景：被映射到一个中间的联结表的关联。

3. 利用联结表的可选一对多关联

Item类增加了一个有用的buyer属性。可以调用anItem.getBuyer()访问出价胜出的用户。（当然，anItem.getSuccessfulBid().getBidder()可以通过不同的路径提供相同的访问。）如果变成双向的，这个关联也将有助于呈现屏幕，显示特定用户已经胜出的所有拍卖：调用aUser.getBoughtItems()来代替编写查询。

从User类的观点来看，这个关联是一对多的。类及其关系如图7-7所示。

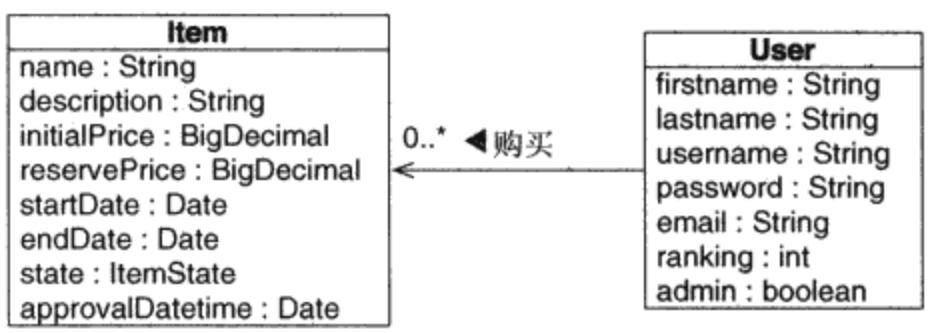


图7-7 用户可以购买货品

为什么这个关联与Item和Bid之间的不同？在UML中，多样性0..*表明引用是可选的。这不太影响Java领域模型，但是对底层的表有影响。你希望在ITEM表中有个BUYER_ID外键列。列必须是可以为空的——不可能已经购买了特定的Item（只要拍卖仍然在进行）。

可以接受外键列为NULL，并应用额外的约束（“仅当还没到拍卖终止时间或者还没有出价时才允许为NULL”）。我们总是努力在关系数据库Schema中避免可为空的列。未知的信息降低了所保存数据的质量。元组（tuple）表示那些为真（true）的命题，我们无法断言不知道的东西。并且，在实际应用程序中，许多开发人员和数据库管理员没有创建正确的约束，并依赖（经常出现bug的）应用程序代码提供数据的完整性。

可选的实体关联（一对一或者一对多）在SQL数据库中用联结表表示得最好。示例Schema请见图7-8。

本章前面给一对一关联添加过一张联结表。为了确保一对一的多样性，在联结表的两个外键列上都应用了unique约束。在目前的情形中，你拥有一对多的多样性，因此只有ITEM_BUYER表的ITEM_ID列是唯一的。一个特定的货品只可以被购买一次。

让我们用XML映射它。首先，是User类的boughtItems集合。

```
<set name="boughtItems" table="ITEM_BUYER">
    <key column="USER_ID"/>
    <many-to-many class="Item"
                  column="ITEM_ID"
                  unique="true"/>
</set>
```

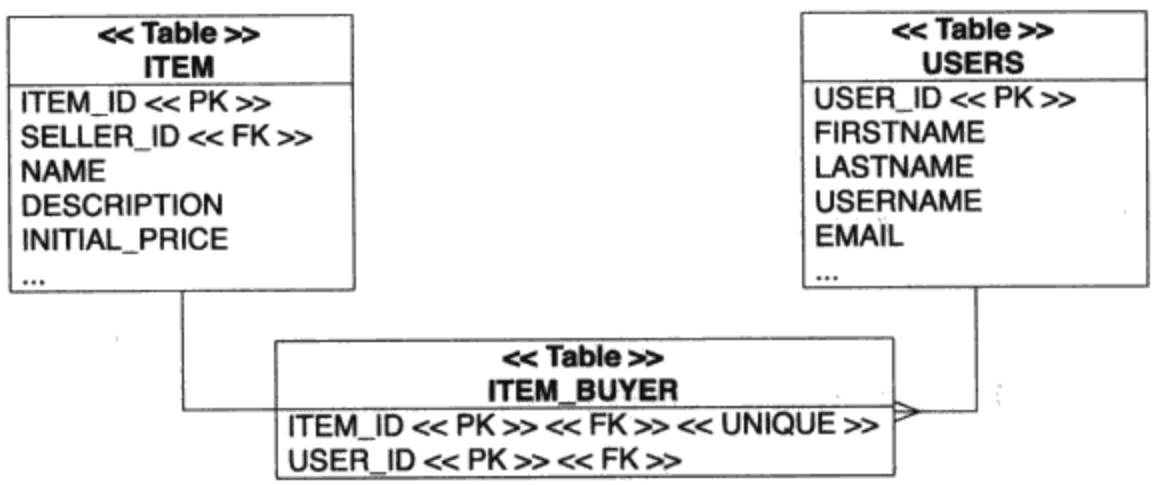


图7-8　使用联结表的可选关系避免了可为空的外键列

用Set作为集合类型。集合表是联结表ITEM_BUYER；它的主键是USER_ID和ITEM_ID的复合。你之前还没见过的新映射元素是<many-to-many>；因为常规的<one-to-many>不知道任何有关联结表的信息，因此需要它。通过在引用目标实体表的外键列上强制unique约束，有效地强制了一对多的多样性。

可以用Item的buyer属性把这个关联映射为双向的。若没有联结表，就要在ITEM表中用BUYER_ID外键列添加<many-to-one>。若有联结表，则必须把这个外键列移到联结表里面。这可能通过<join>映射：

```
<join table="ITEM_BUYER"
      optional="true"
      inverse="true">
    <key column="ITEM_ID" unique="true" not-null="true"/>
    <many-to-one name="buyer" column="USER_ID"/>
</join>
```

这里有两个重要的细节：首先，关联是可选的，你告诉Hibernate如果被分组的属性（这里只有一个属性buyer）为null，就不要把行插入到联结表。第二，这是个双向的实体关联。与往常一样，一端必须为反向的。你已经选择了<join>为反向的；现在Hibernate利用集合状态使数据库同步，并忽略Item.buyer属性的状态。只要集合不是被索引的变量（list、map或者array），就可以通过声明集合inverse="true"来反向这一点。创建购买的货品和用户对象之间链接的Java代码在这两种情况下相同：

```
aUser.getBoughtItems().add(anItem);
anItem.setBuyer(aUser);
```

可以映射JPA中的二级表来创建使用联结表的一对多关联。首先，把@ManyToOne映射到联结表：

```
@Entity
public class Item {
    @ManyToOne
    @JoinTable(
        name = "ITEM_BUYER",
        joinColumns = {@JoinColumn(name = "ITEM_ID")},
        inverseJoinColumns = {@JoinColumn(name = "USER_ID")}
    )
    private User buyer;
    ...
}
```

在编写本书之时，这个映射还有个限制：不能把它设置为optional="true"；因而，USER_ID列是可为空的。如果试图在@JoinColumn上添加nullable="false"，Hibernate Annotations就认为你要整个buyer属性永远不为null。此外，联结表的主键现在只有ITEM_ID列。这样很好，因为你不想在这个表中有重复的货品——它们只能被购买一次。

为了使这个映射变成双向的，要在User类上添加一个集合，并用mappedBy使它变成反向：

```
@OneToMany(mappedBy = "buyer")
private Set<Item> boughtItems = new HashSet<Item>();
```

我们在前一节给联结表中的一对多关联介绍过<many-to-many>XML映射元素。@JoinTable注解是注解中的等价物。来映射一个真正的多对多关联。

7.2.2 多对多关联

Category和Item之间的关联是多对多关联，如图7-9所示。

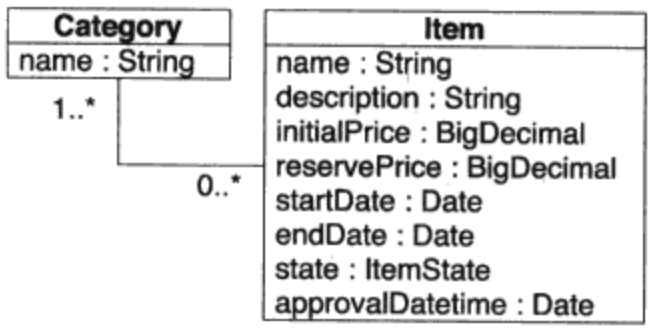


图7-9 Category和Item之间多对多的值关联

在真实的系统中，可能没有多对多关联。依我们的经验，通常有其他信息必须被附加到被关联实例之间的每一个链接（例如货品被添加到目录时的日期和时间，表示这一信息的最好方式是通过中间的关联类（association class）。在Hibernate中，可以把这个关联类映射为实体，并给任何一端映射两个一对多的关联。可能更方便的做法是，也可以映射一个复合的元素类，这是我们后面要介绍的一种方法。实现一个真正的多对多实体关联是本节的目的。让我们从一个单向的示例开始。

1. 简单的单向多对多关联

如果只需要单向导航，映射就很简单。单向的多对多关联本质上不比我们之前讨论过的值类型实例的集合更难。例如，如果Category有一组Items，你就可以创建这个映射：

```
<set name="items"
     table="CATEGORY_ITEM"
     cascade="save-update">
    <key column="CATEGORY_ID"/>
    <many-to-many class="Item" column="ITEM_ID"/>
</set>
```

联结表（也被有些开发人员称为链接表，link table）有两个列：CATEGORY和ITEM表的外键。主键是这两列的复合。完整的表结构如图7-10所示。

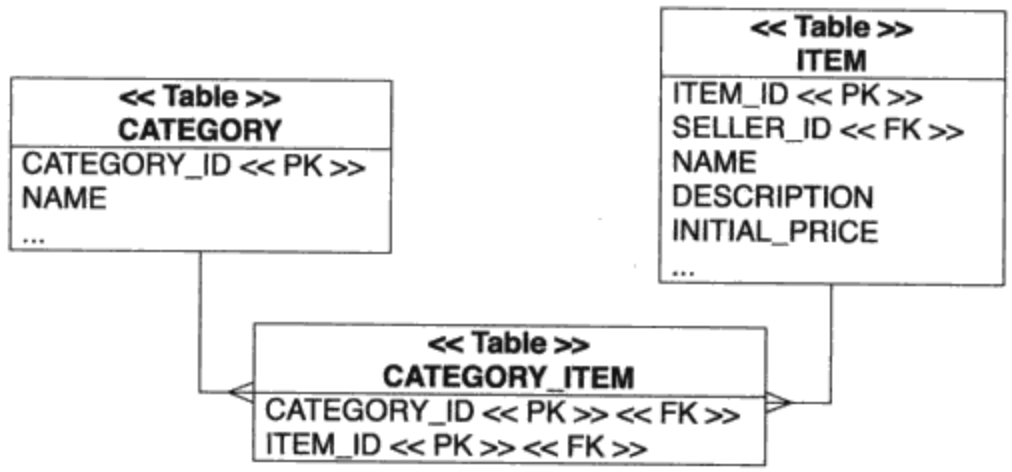


图7-10　被映射到关联表的多对多实体关联

在JPA注解中，多对多的关联通过@ManyToMany属性映射：

```
@ManyToMany
@JoinTable(
    name = "CATEGORY_ITEM",
    joinColumns = {@JoinColumn(name = "CATEGORY_ID")},
    inverseJoinColumns = {@JoinColumn(name = "ITEM_ID")}
)
private Set<Item> items = new HashSet<Item>();
```

在Hibernate XML中，也可以在联结表中通过一个单独的主键列切换到<idbag>：

```
<idbag name="items"
       table="CATEGORY_ITEM"
       cascade="save-update">
    <collection-id type="long" column="CATEGORY_ITEM_ID">
        <generator class="sequence"/>
    </collection-id>
    <key column="CATEGORY_ID"/>
    <many-to-many class="Item" column="ITEM_ID"/>
</idbag>
```

与通常使用<idbag>映射一样，主键是一个代理键列CATEGORY_ITEM_ID。因此允许重复的链接；同一个Item可以被添加到Category两次。（这似乎不是一项有用的特性。）利用注解，可以通过Hibernate @CollectionId切换到一个标识符bag：

```
@ManyToMany
@CollectionId(
    columns = @Column(name = "CATEGORY_ITEM_ID"),
    type = @org.hibernate.annotations.Type(type = "long"),
    generator = "sequence"
```

```
)
@JoinTable(
    name = "CATEGORY_ITEM",
    joinColumns = {@JoinColumn(name = "CATEGORY_ID")},
    inverseJoinColumns = {@JoinColumn(name = "ITEM_ID")}
)
private Collection<Item> items = new ArrayList<Item>();
```

使用集的一般多对多映射的JPA XML描述符（你不能给标识符bag使用Hibernate扩展）看起来像这样：

```
<entity class="auction.model.Category" access="FIELD">
    ...
    <many-to-many name="items">
        <join-table name="CATEGORY_ITEM">
            <join-column name="CATEGORY_ID"/>
            <inverse-join-column name="ITEM_ID"/>
        </join-table>
    </many-to-many>

</entity>
```

甚至可以在多对多关联中切换到一个被索引的集合（map或者list）。下列例子在Hibernate XML中映射了一个list：

```
<list name="items"
      table="CATEGORY_ITEM"
      cascade="save-update">
    <key column="CATEGORY_ID"/>
    <list-index column="DISPLAY_POSITION"/>
    <many-to-many class="Item" column="ITEM_ID"/>
</list>
```

链接表的主键是CATEGORY_ID和DISPLAY_POSITION列的复合；这个映射保证每个Item在Category中的位置都是持久化的。或者，利用注解：

```
@ManyToMany
@JoinTable(
    name = "CATEGORY_ITEM",
    joinColumns = {@JoinColumn(name = "CATEGORY_ID")},
    inverseJoinColumns = {@JoinColumn(name = "ITEM_ID")}
)
@org.hibernate.annotations.IndexColumn(name = "DISPLAY_POSITION")
private List<Item> items = new ArrayList<Item>();
```

如前所述，JPA只支持有序集合（通过一个可选的@OrderBy注解或者按主键排列），因此必须再次对被索引的集合支持使用Hibernate扩展。如果没有添加@IndexColumn，List就会通过bag语义保存（不保证元素的持久化顺序）。

创建Category和Item之间的链接很容易：

```
aCategory.getItems().add(anItem);
```

双向的多对多关联稍微难一些。

2. 双向的多对多关联

你知道，双向关联的一端必须被映射为反向，因为你已经对（一个或多个）外键列命名了两次。给双向的多对多关联应用同样的原则：链接表的每一行都由两个集合元素表示，关联的两端各一个元素。Item和Category之间的关联，在内存中由Category的items集合中Item实例表示，而且还通过Item的Categories集合中Category实例表示。

在讨论这个双向案例的映射之前，必须知道创建对象关联的代码也变了：

```
aCategory.getItems().add(anItem);
anItem.getCategories().add(aCategory);
```

如往常一样，双向关联（无论什么多样性）要求关联的两端都要设置。

映射双向的多对多关联时，必须用inverse="true"声明关联的一端，来定义哪一端的状态用来更新联结表。可以选择哪一端应该为反向。

回想前一节items集合中的这个映射：

```
<class name="Category" table="CATEGORY">
    ...
    <set name="items"
         table="CATEGORY_ITEM"
         cascade="save-update">
        <key column="CATEGORY_ID"/>
        <many-to-many class="Item" column="ITEM_ID"/>
    </set>
```

可以给双向关联的Category端重用这个映射，并映射另一端如下：

```
<class name="Item" table="ITEM">
    ...
    <set name="categories"
         table="CATEGORY_ITEM"
         inverse="true"
         cascade="save-update">
        <key column="ITEM_ID"/>
        <many-to-many class="Category" column="CATEGORY_ID"/>
    </set>
</class>
```

注意inverse="true"。这个设置再次告诉Hibernate忽略对categories集合所做的改变，并且如果在Java代码中链接实例，关联的另一端（items集合）就是应该与数据库同步的表示法。

你已经对集合的两端都启用了cascade="save-update"。我们推想这是不切实际的。另一方面，级联选项all、delete和delete-orphans对于多对多的关联都是没有意义的。（这是测试你是否理解实体和值类型的一个好机会——试着想出合理的答案，为什么这些级联选项对于多对多的关联没有意义。）

在JPA中，使用注解，可以很容易地使多对多的关联变成双向。首先是非反向端：

```
@ManyToMany
@JoinTable(
    name = "CATEGORY_ITEM",
    joinColumns = {@JoinColumn(name = "CATEGORY_ID")},
    inverseJoinColumns = {@JoinColumn(name = "ITEM_ID")}
)
private Set<Item> items = new HashSet<Item>();
```

现在是对面的反向端：

```
@ManyToMany(mappedBy = "items")
private Set<Category> categories = new HashSet<Category>();
```

如你所见，并不一定要在反向端重复联结表声明。

哪些集合类型可以用于双向的多对多关联？需要在每一端使用相同的集合类型吗？例如，给关联的非反向端映射使用<list>、给反向端映射使用<bag>是合理的。

对于反向端，可以接受<set>，下列bag映射也一样：

```
<class name="Item" table="ITEM">
    ...
    <bag name="categories"
         table="CATEGORY_ITEM"
         inverse="true"
         cascade="save-update">
        <key column="ITEM_ID"/>
        <many-to-many class="Category" column="CATEGORY_ID"/>
    </bag>
</class>
```

在JPA中，bag是没有持久化索引的一个集合：

```
@ManyToMany(mappedBy = "items")
private Collection<Category> categories = new ArrayList<Category>();
```

没有其他的映射可以用于多对多关联的反向端。被索引的集合（list和map）不行，因为如果集合为反向，Hibernate将不会初始化或者维持索引列。换句话说，无法通过被索引的集合在两端映射多对多的关联。

我们已经反对多对多关联的使用，因为联结表中的额外列通常是不可避免的。

7.2.3 把列添加到联结表

本节将讨论Hibernate用户经常问的一个问题：如果联结表有一些额外的列，而不只是两个外键列时，该怎么办？

想象在每次把Item添加到Category时都要记录一些信息。例如，可能要保存日期，以及把该货品添加到这个目录的用户名称。这在联结表中就需要额外的列，如图7-11所示。

可以用两种常用的策略把这样一种结构映射到Java类。第一种策略需要一个用于联结表的中间实体类，并通过一对多的关联而被映射。第二种策略通过给联结表使用一个值类型的类来利用组件的集合。

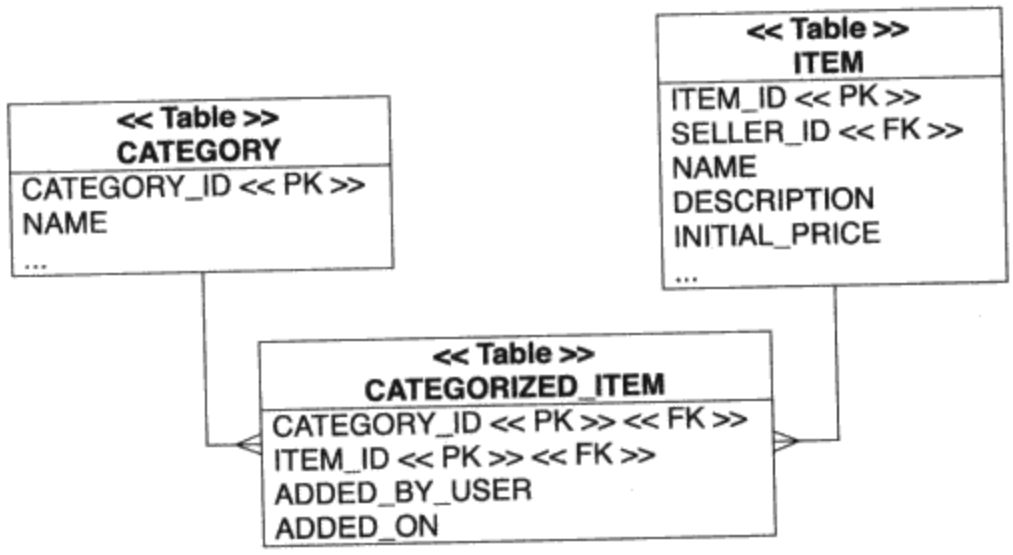


图7-11　多对多关联中联结表上的额外列

1. 把联结表映射到中间实体

我们现在讨论的第一个选项解决了Category和带有中间实体类(CategorizedItem)的Item之间的多对多关系。代码清单7-1展现了这个实体类，它在Java中表示联结表（包括JPA注解）：

代码清单7-1　表示包含额外列的链接表的实体类

```
@Entity
@Table(name = "CATEGORIZED_ITEM")
public class CategorizedItem {

    @Embeddable
    public static class Id implements Serializable {

        @Column(name = "CATEGORY_ID")
        private Long categoryId;

        @Column(name = "ITEM_ID")
        private Long itemId;

        public Id() {}

        public Id(Long categoryId, Long itemId) {
            this.categoryId = categoryId;
            this.itemId = itemId;
        }
        public boolean equals(Object o) {
            if (o != null && o instanceof Id) {
                Id that = (Id)o;
                return this.categoryId.equals(that.categoryId) &&
                       this.itemId.equals(that.itemId);
            } else {
                return false;
            }
        }
        public int hashCode() {
            return categoryId.hashCode() + itemId.hashCode();
        }
    }
```

```
@EmbeddedId
private Id id = new Id();

@Column(name = "ADDED_BY_USER")
private String username;

@Column(name = "ADDED_ON")
private Date dateAdded = new Date();

@ManyToOne
@JoinColumn(name="ITEM_ID",
            insertable = false,
            updatable = false)
private Item item;

@ManyToOne
@JoinColumn(name="CATEGORY_ID",
            insertable = false,
            updatable = false)
private Category category;
      public CategorizedItem() {}

      public CategorizedItem(String username,
                             Category category,
                             Item item) {
          // Set fields
          this.username = username;

          this.category = category;
          this.item = item;

          // Set identifier values
          this.id.categoryId = category.getId();
          this.id.itemId = item.getId();

          // Guarantee referential integrity
          category.getCategorizedItems().add(this);
          item.getCategorizedItems().add(this);
      }

      // Getter and setter methods
      ...
}
```

实体类需要一个标识符属性。联结表的主键是CATEGORY_ID和ITEM_ID的复合。因而，实体类也有一个复合键，为了方便起见，把它封装在一个静态的嵌套类中了。你还可以看到构建CategorizedItem涉及了标识符值的设置——复合键值由应用程序分配。要特别注意这个构造器，以及它如何设置字段值，并通过在关联的任意一端管理集合来确保参照完整性。

在XML中把这个类映射到联结表：

```
<class name="CategorizedItem"
       table="CATEGORY_ITEM"
       mutable="false">

    <composite-id name="id" class="CategorizedItem$Id">
        <key-property name="categoryId"
```

```
                    access="field"
                    column="CATEGORY_ID"/>

        <key-property name="itemId"
                    access="field"
                    column="ITEM_ID"/>
    </composite-id>

    <property name="dateAdded"
              column="ADDED_ON"
              type="timestamp"
              not-null="true"/>

    <property name="username"
              column="ADDED_BY_USER"
              type="string"
              not-null="true"/>

    <many-to-one name="category"
                 column="CATEGORY_ID"
                 not-null="true"
                 insert="false"
                 update="false"/>

    <many-to-one name="item"
                 column="ITEM_ID"
                 not-null="true"
                 insert="false"
                 update="false"/>
</class>
```

实体类被映射为不可变——你创建之后永远不用更新任何属性。Hibernate直接访问<composite-id>字段——在这个被嵌套的类中不需要获取方法和设置方法。这两个<many-to-one>映射实际上是只读的：insert和update被设置为false。这是必需的，因为列被映射两次：一次在复合键中（负责值的插入），另一次用于多对一的关联。

Category和Item实体（可以）有对CategorizedItem实体（一个集合）的一对多关联。例如，在Category中：

```
<set name="categorizedItems"
     inverse="true">
    <key column="CATEGORY_ID"/>
    <one-to-many class="CategorizedItem"/>
</set>
```

下面是注解的等价物：

```
@OneToMany(mappedBy = "category")
private Set<CategorizedItem> categorizedItems =
                                    new HashSet<CategorizedItem>();
```

这里没有什么特别的东西要考虑；它是个包含反向集合的一般的双向一对多关联。添加相同的集合和映射到Item来完成关联。创建和保存目录和货品之间链接的代码如下：

```
CategorizedItem newLink =
    new CategorizedItem(aUser.getUsername(), aCategory, anItem);

session.save(newLink);
```

Java对象的参照完整性受到CategorizedItem的构造器的保证，该构造器管理aCategory和anItem中的集合。移除和删除目录和货品之间的链接：

```
aCategory.getCategorizedItems().remove( theLink );
anItem.getCategorizedItems().remove( theLink );

session.delete(theLink);
```

这个策略的主要好处在于双向导航的可能性：可以通过调用aCategory.getCategorizedItems()获取目录中的所有货品，并用anItem.getCategorizedItems()从反向导航。缺点是要用更复杂的代码管理CategorizedItem实体实例，创建和移除关联——它们必须被独立地被保存和删除，并且在CategorizedItem类中需要一些基础结构，例如复合标识符。但是，可以用集合中的级联选项启用传播性持久化，从Category和Item到CategorizedItem，如12.1节中所述。

处理联结表中额外列的第二种策略不需要中间的实体类；它更为简单。

2. 把联结表映射到组件的集合

首先，简化CategorizedItem类，并使它成为值类型，不用标识符或者任何复杂的构造器：

```
public class CategorizedItem {
    private String username;
    private Date dateAdded = new Date();
    private Item item;
    private Category category;

    public CategorizedItem(String username,
                           Category category,
                           Item item) {
        this.username = username;
        this.category = category;
        this.item = item;
    }
    ...

    // Getter and setter methods
    // Don't forget the equals/hashCode methods
}
```

至于所有的值类型，这个类必须为一个实体所有。所有者是Category，它有这些组件的一个集合：

```
<class name="Category" table="CATEGORY">

    ...
    <set name="categorizedItems" table="CATEGORY_ITEM">
        <key column="CATEGORY_ID"/>
        <composite-element class="CategorizedItem">
            <parent name="category"/>

            <many-to-one name="item"
                         column="ITEM_ID"
                         not-null="true"
                         class="Item"/>

            <property name="username" column="ADDED_BY_USER"/>
            <property name="dateAdded" column="ADDED_ON"/>
```

```
        </composite-element>
    </set>

</class>
```

这是对于联结表中包含额外列的多对多关联的完整映射。<many-to-one>元素表示对Item的关联；<property>映射涵盖了联结表中的额外列。数据库表中只有一处变化：现在CATEGORY_ITEM表有一个主键，它是所有列的复合，而不仅仅是CATEGORY_ID和ITEM_ID，就像前一节的一样。因此，所有属性都永远不应该为空的——否则就无法识别联结表中的行了。除了这个变化之外，这些表看起来仍然如图7-11中所示的一样。

你可以通过对User而不只是用户名的引用来增强这个映射。这要求联结表中要有一个额外的USER_ID列，包含USERS的一个外键。这是一个三重关联（ternary association）映射：

```
<set name="categorizedItems" table="CATEGORY_ITEM">
    <key column="CATEGORY_ID"/>
    <composite-element class="CategorizedItem">
        <parent name="category"/>

        <many-to-one name="item"
                     column="ITEM_ID"
                     not-null="true"
                     class="Item"/>

        <many-to-one name="user"
                     column="USER_ID"
                     not-null="true"
                     class="User"/>

        <property name="dateAdded" column="ADDED_ON"/>

    </composite-element>
</set>
```

这是一段相当怪异的代码！

组件集合的好处无疑是链接对象的隐式生命周期。为了创建Category和Item之间的关联，就把新的CategorizedItem实例添加到集合。要解除链接，就从集合中移除该元素。不需要额外的级联设置，并且Java代码也被简化了：

```
CategorizedItem aLink =
    new CategorizedItem(aUser.getUserName(), aCategory, anItem);

aCategory.getCategorizedItems().add( aLink );

aCategory.getCategorizedItems().remove( aLink );
```

这种方法的缺点在于无法启用双向导航：根据定义，组件（例如CategorizedItem）不可能有共享的引用。不能从Item导航到CategorizedItem。但是，你可以编写一个查询来获取所需的对象。

让我们用注解进行相同的映射。首先，使组件类变成@Embeddable，并添加组件列和关联映射：

```
@Embeddable
public class CategorizedItem {

    @org.hibernate.annotations.Parent // Optional back-pointer
    private Category category;
```

```
    @ManyToOne
    @JoinColumn(name = "ITEM_ID",
                nullable = false,
                updatable = false)
    private Item item;

    @ManyToOne
    @JoinColumn(name = "USER_ID",
                nullable = false,
                updatable = false)
    private User user;

    @Temporal(TemporalType.TIMESTAMP)
    @Column(name = "ADDED_ON", nullable = false, updatable = false)
    private Date dateAdded;

    ...
    // Constructor
    // Getter and setter methods
    // Don't forget the equals/hashCode methods
}
```

现在把这个CategorizedItem类映射为Category类中组件的集合：

```
@org.hibernate.annotations.CollectionOfElements
@JoinTable(
    name = "CATEGORY_ITEM",
    joinColumns = @JoinColumn(name = "CATEGORY_ID")
)
private Set<CategorizedItem> categorizedItems =
                                    new HashSet<CategorizedItem>();
```

就是这样：你已经通过注解映射了一个三重关联。一开始看起来复杂到难以置信的东西，现在已经被减少成了几行注解元数据，它们中大部分是可选的。

我们要探讨的最后一个集合映射是实体引用的Map。

7.2.4 映射 map

上一章映射了Java Map——Map的键和值都是值类型，它们是简单的字符串。可以创建更复杂的map：不仅键可以是对实体的引用，值也可以。因此结果可以是三重关联。

1. 值对实体的引用

首先，假设只有每个映射项的值才是对另一个实体的引用。键是个值类型：long。想象Item实体有一个Bid实例的map，并且每个映射项都是一对Bid标识符与对Bid实例的引用。如果通过anItem.getBidsByIdentifier()迭代，便通过映射项如(1, <reference to Bid with PK 1>), (2,<reference to Bid with PK 2>)等进行迭代。

这个映射的底层表没有什么特别的：你再次有了ITEM和BID表，BID表中有ITEM_ID外键列。此处的目的是在应用程序中用一个Map对数据进行稍微不同的表示。

Item类中包括一个Map：

```
@MapKey(name="id")
@OneToMany(mappedBy="item")
private Map<Long,Bid> bidsByIdentifier = new HashMap<Long,Bid>();
```

此处新出现的东西是JPA的@MapKey元素——它把目标实体的一个属性映射为该映射的键。如果省略name属性，则默认为目标实体的标识符属性（因此这里的名称是多余的）。由于映射的多个键形成了一个集，因此一个特定映射的值希望是唯一的——这是对于Bid主键，但是对于Bid的任何其他属性则可能不是如此。

在Hibernate XML中，这个映射如下：

```
<map name="bidsByIdentifier"inverse="true">
    <key column="ITEM_ID"/>
    <map-key type="long" formula="BID_ID"/>
    <one-to-many class="Bid"/>
</map>
```

映射的formula键使这个列变成了只读，因此当你修改该映射时它从不更新。一种更常见的情况是三重关联中间的映射。

2. 三重关联

你现在可能有点不耐烦了，但我们保证这是最后一次介绍映射Category和Item之间关联的另一种方法了。下面来概括一下你已经了解的有关多对多关联的知识：

- 它可以利用两个集合在任意一端映射，也可以通过只有两个外键列的联结表映射。这是一个常规的多对多关联映射。
- 它可以通过表示联结表的中间实体类和那里任何额外的列映射。一对多的关联在任意一端（Category和Item）映射，并且双向的多对一相当于在中间的实体类中映射。
- 它可以被映射为单向，通过表示为值类型组件的联结表。Category实体有组件的一个集合。每个组件都有一个对它自己Category的引用以及对Item的多对一实体关联。（也可以在这段解释中切换单词Category和Item。）

你之前通过把另一个多对一实体关联添加到User，把上一个场景变成了一个三重关联。下面来用Map完成同样的事。

Category有Item实例的一个Map——每个映射项的键都是对Item的引用。每个映射项的值都是把Item添加到Category的User。如果联结表上没有额外的列，这个策略就很合适；请见图7-12中的Schema。

这一策略的好处在于不需要任何中间类，不需要实体或者值类型，就可以在Java应用程序中表示联结表的ADDED_BY_USER_ID列。

首先，这是Category中带有Hibernate扩展注解的Map属性。

```
@ManyToMany
@org.hibernate.annotations.MapKeyManyToMany(
    joinColumns = @JoinColumn(name = "ITEM_ID")
)
@JoinTable(
    name = "CATEGORY_ITEM",
    joinColumns = @JoinColumn(name = "CATEGORY_ID"),
    inverseJoinColumns = @JoinColumn(name = "USER_ID")
)
private Map<Item,User> itemsAndUser = new HashMap<Item,User>();
```

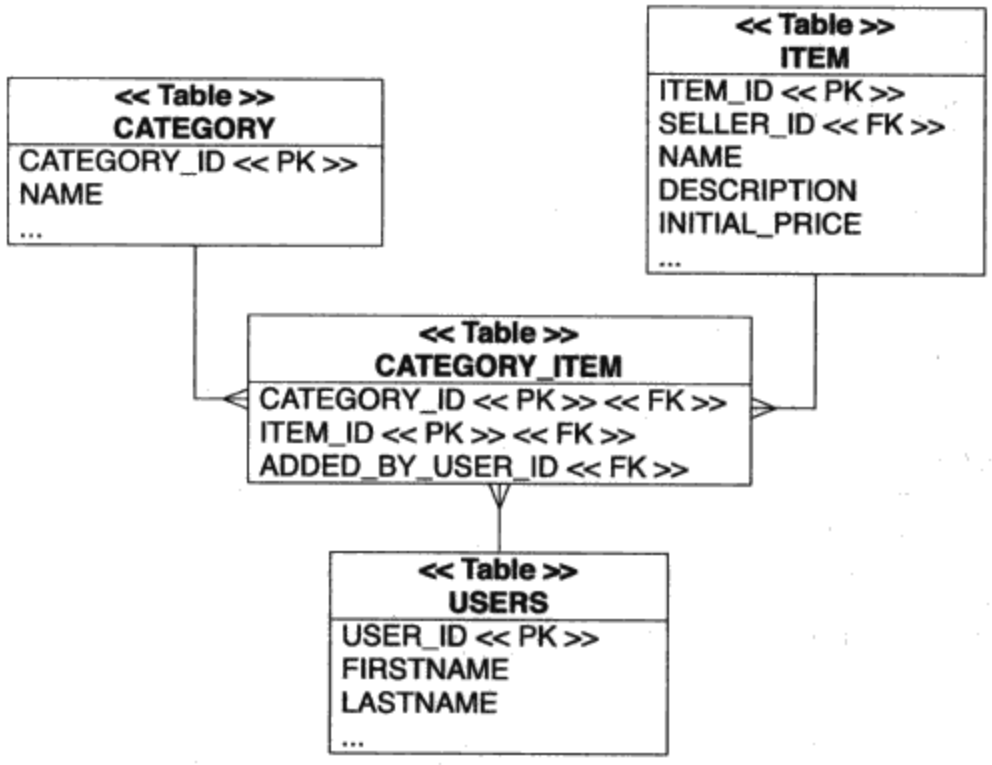


图7-12 三个实体之间使用联结表的三重关联

Hibernate XML映射包括一个新的元素<map-key-many-to-many>:

```
<map name="itemsAndUser" table="CATEGORY_ITEM">
    <key column="CATEGORY_ID"/>
    <map-key-many-to-many column="ITEM_ID" class="Item"/>
    <many-to-many column="ADDED_BY_USER_ID" class="User"/>
</map>
```

为了在所有三个实体之间创建一个链接，如果你的所有实例都已经处于持久化状态，就添加一个新的项到该映射中：

```
aCategory.getItemsAndUser().add( anItem, aUser );
```

要移除链接，就从该映射中移除该项。当作练习，你可以试着通过Item中categories的一个集合，使这个映射变成双向。

记住，这必须是一个反向的集合映射，因此它不支持被索引的集合。

既然你知道了用于一般实体的所有关联映射方法，那么我们仍然必须考虑继承，以及对各种级别的继承层次结构的关联。我们真正要的是多态的（polymorphic）行为。我们来看看Hibernate是如何处理多态的实体关联的。

7.3 多态关联

多态是面向对象语言如Java的一项定义特性。对多态关联和多态查询的支持是ORM解决方案（如Hibernate）的一项绝对基础的特性。令人惊讶的是，我们已经谈了这么多，却不需要讨论多态。甚至更为令人惊讶的是，关于这一主题并没有太多的内容可说——多态在Hibernate中是如此容易使用，以致我们无须花费太多的精力对它进行解释。

为了有一个整体的认识，首先考虑对可能有子类的一个类的多对一关联。在这种情况下，

Hibernate保证你可以创建到任何子类实例的链接，就像创建到超类的实例的链接一样。

7.3.1　多态的多对一关联

多态关联（polymorphic association）可以引用在映射元数据中显式指定的类的子类实例。对于这个例子，考虑User的defaultBillingDetails属性。它引用一个特定的BillingDetails对象，该对象在运行时可以是该类的任何具体实例。这些类如图7-13所示。

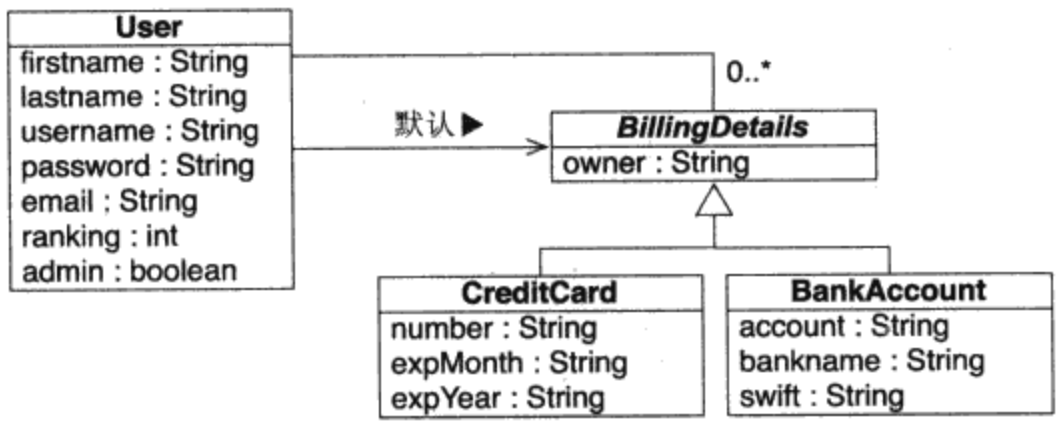


图7-13　user默认为拥有信用卡或者银行账号

在User.hbm.xml中把这个关联映射到抽象类BillingDetails如下：

```
<many-to-one name="defaultBillingDetails"
             class="BillingDetails"
             column="DEFAULT_BILLING_DETAILS_ID"/>
```

但是由于BillingDetails是抽象的，关联必须在运行时引用它其中一个子类的实例——CreditCard或者BankAccount。

不必做任何特别的事情来启用Hibernate中的多态关联；在关联映射中指定任何被映射的持久化类的名称（或者让Hibernate利用反射发现它），然后，如果该类声明了任何<union-subclass>、<subclass>或者<joined-subclass>元素，该关联就自然地成为多态。

下列代码示范了如何创建对CreditCard子类的实例的关联：

```
CreditCard cc = new CreditCard();
cc.setNumber(ccNumber);
cc.setType(ccType);
cc.setExpiryDate(ccExpiryDate);

User user = (User) session.get(User.class, userId);
user.addBillingDetails(cc); // Add it to the one-to-many association

user.setDefaultBillingDetails(cc);

// Complete unit of work
```

现在，当你在第二个工作单元中导航关联时，Hibernate就自动获取CreditCard实例：

```
User user = (User) secondSession.get(User.class, userId);

// Invoke the pay() method on the actual subclass instance
user.getDefaultBillingDetails().pay(amount);
```

只有一件事情必须小心：如果BillingDetails通过lazy="true"映射（这是默认的），Hibernate就会代理defaultBillingDetails关联目标。在这种情况下，就不能在运行时执行类型转换到具体的类CreditCard，甚至instanceof操作符也会表现得很奇怪：

```
User user = (User) session.get(User.class, userid);
BillingDetails bd = user.getDefaultBillingDetails();
System.out.println( bd instanceof CreditCard ); // Prints "false"
CreditCard cc = (CreditCard) bd; // ClassCastException!
```

在这段代码中，类型转换失败了，因为bd是一个代理实例。当一个方法在代理中被调用时，调用被委托给了被延迟抓取的CreditCard的实例（它是运行时生成的子类的一个实例，因此instanceof也失败）。直到这个初始化发生，Hibernate才知道给定实例的子类型是什么——这将需要一个数据库命中，这是你试图首先通过延迟加载要努力避免的。为了执行代理安全的类型转换，要使用load()：

```
User user = (User) session.get(User.class, userId);
BillingDetails bd = user.getDefaultBillingDetails();

// Narrow the proxy to the subclass, doesn't hit the database
CreditCard cc =
    (CreditCard) session.load( CreditCard.class, bd.getId() );
expiryDate = cc.getExpiryDate();
```

调用load()之后，bd和cc指向两个不同的代理实例，这两者都委托给同一个底层的CreditCard实例。然而，第二个代理有不同的接口，可以调用只应用给这个接口的方法（如getExpiryDate()）。

注意，可以通过避免延迟抓取来避免这些问题，就像下列代码中的那样，用一个主动抓取查询：

```
User user = (User)session.createCriteria(User.class)
    .add(Restrictions.eq("id", uid) )
    .setFetchMode("defaultBillingDetails", FetchMode.JOIN)
    .uniqueResult();

// The users defaultBillingDetails have been fetched eagerly
CreditCard cc = (CreditCard) user.getDefaultBillingDetails();
expiryDate = cc.getExpiryDate();
```

真正面向对象的代码不应该使用instanceof或者众多的类型转换。如果你发现自己使用代理时遇到了问题，就应该置疑设计，问问是否有更加多态的方法。Hibernate也提供BCI（ByteCode Instrumentation），作为通过代理延迟加载的另一种选择；我们会在13.1节中回到抓取策略的主题。

一对一的关联以同样的方式进行处理。多值的关联要如何处理呢——例如，每个User的billingDetails集合？

7.3.2 多态集合

User可以有对多个BillingDetails的引用，而不只是单个默认（默认是这多个中的一个）。用一个双向的一对多关联映射它。

BillingDetails中有下列代码：

```
<many-to-one name="user"
             class="User"
             column="USER_ID"/>
```

在Users映射中有：

```
<set name="billingDetails"
     inverse="true">
    <key column="USER_ID"/>
    <one-to-many class="BillingDetails"/>
</set>
```

添加CreditCard很容易：

```
CreditCard cc = new CreditCard();
cc.setNumber(ccNumber);
cc.setType(ccType);
cc.setExpMonth(...);
cc.setExpYear(...);

User user = (User) session.get(User.class, userId);

// Call convenience method that sets both sides of the association
user.addBillingDetails(cc);

// Complete unit of work
```

同往常一样，addBillingDetails()调用getBillingDetails().add(cc)和cc.setUser(this)，通过设置两个指针来保证关系的完整性。

可以迭代集合，并多态地处理CreditCard和BankAccount的实例（尽管你可能并不想在最终的系统中向用户开几次账单）：

```
User user = (User) session.get(User.class, userId);

for( BillingDetails bd : user.getBillingDetails() ) {
    // Invoke CreditCard.pay() or BankAccount.pay()
    bd.pay(paymentAmount);
}
```

在目前为止的这些例子中，假设BillingDetails是被显式映射的一个类，继承映射策略是每个类层次结构一张表，或者用每个子类一张表标准化。

然而，如果层次结构通过每个具体的类一张表映射（隐式多态）或者显式地通过每个带有联合的具体类一张表映射，这个场景就需要一个更为复杂的解决方案了。

7.3.3　对联合的多态关联

Hibernate支持多态的多对一和一对多关联，如前几节所示，即使类层次结构是通过每个具体类一张表的策略映射。你可能想知道这是如何进行的，因为你可能没有使用这一策略以用于超类的表；如果真是如此，就无法引用或者添加外键列到BILLING_DETAILS了。

回顾5.1.2节中对每个带有联合的具体类一张表的讨论。特别注意获取BillingDetails的实例时，Hibernate执行的多态查询。现在，考虑给User映射的BillingDetails的集合：

```
<set name="billingDetails"
     inverse="true">
    <key column="USER_ID"/>
    <one-to-many class="BillingDetails"/>
</set>
```

如果想要启用多态的联合特性，这个多态关联的必要条件是它为反向：在对面端必须有一个映射。在BillingDetails的映射中，通过<union-subclass>，必须包括<many-to-one>关联：

```
<class name="BillingDetails" abstract="true">

    <id name="id" column="BILLING_DETAILS_ID" .../>

    <property .../>

    <many-to-one name="user"
                 column="USER_ID"
                 class="User"/>

    <union-subclass name="CreditCard" table="CREDIT_CARD">
        <property .../>
    </union-subclass>

    <union-subclass name="BankAccount" table="BANK_ACCOUNT">
        <property .../>
    </union-subclass>

</class>
```

有两张表用于层次结构的两个具体类。每张表都有外键列USER_ID，引用USERS表。该Schema如图7-14所示。

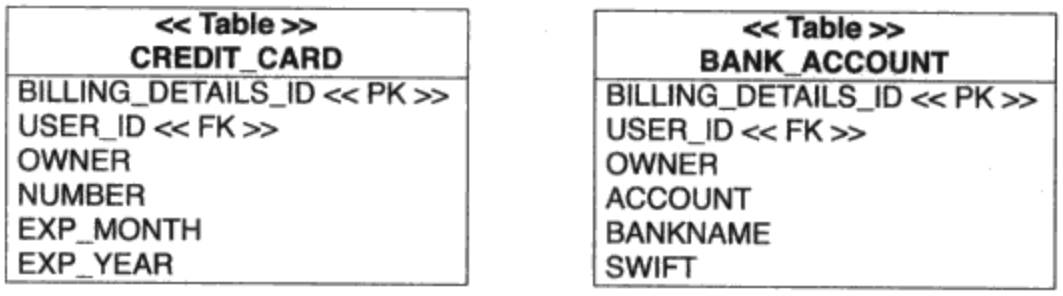


图7-14 被映射到两张单独表的两个具体类

现在，考虑下列数据访问代码：

```
aUser.getBillingDetails().iterator().next();
```

Hibernate执行UNION查询，获取在这个集合中引用的所有实例：

```
select
    BD.*
from
  ( select
      BILLING_DETAILS_ID, USER_ID, OWNER,
      NUMBER, EXP_MONTH, EXP_YEAR,
      null as ACCOUNT, null as BANKNAME, null as SWIFT,
      1 as CLAZZ
    from
      CREDIT_CARD
```

```
    union

    select
      BILLING_DETAILS_ID, USER_ID, OWNER,
      null as NUMBER, null as EXP_MONTH, null as EXP_YEAR
      ACCOUNT, BANKNAME, SWIFT,
      2 as CLAZZ
    from
      BANK_ACCOUNT
  ) BD
where
    BD.USER_ID = ?
```

FROM子句子查询是所有具体类表的一个联合，它给所有实例包括了USER_ID外键值。现在外部的select对引用特定用户的所有行在WHERE子句中包括了一项限制。

这个魔法对于数据的获取非常棒。如果操作集合和关联，非反向端被用来在具体的表中更新USER_ID列。换句话说，反向集合的修改不起作用：采用CreditCard或者BankAccount实例的user属性值。

现在再次考虑多对一的关联defaultBillingDetails，在USERS表中通过DEFAULT_BILLING_DETAILS_ID列映射。如果访问属性，Hibernate就执行一个看起来类似于前一个查询的UNION查询来获取这个实例。然而，不是在WHERE子句中限制特定的用户，而是在特定的BILLING_DETAILS_ID中进行限制。

重要提示：Hibernate不能也不会用这个策略为DEFAULT_BILLING_DETAILS_ID创建外键约束。这个引用的目标表可以是任何具体的表，它们不能被轻易地约束。你应该考虑通过一个数据库触发器为这个列编写一个定制的完整性规则。

还有一种有问题的继承策略：带有隐式多态的每个具体类一张表。

7.3.4　每个具体类一张多态表

在5.1.1节中，我们定义过每个具体类一张表的策略，并且发现这个映射策略使得表示多态关联变得很困难，因为无法把外键关系映射到抽象超类的表。没有使用这一策略时用于超类的表；只有用于具体类的表。也无法创建UNION，因为Hibernate不知道什么东西在统一这些具体的类；该超类（或者接口）不会被随处映射。

如果这个继承映射策略应用在BillingDetails层次结构中，Hibernate就不支持User中多态的billingDetails一对多集合。如果你需要多态的多对一关联来使用这一策略，则必须求助于hack。我们在本节中介绍的这种方法应该成为你最迫不得已时的选择。先试着转换到<union-subclass>映射。

假设你想要表示从User到BillingDetails的多态的多对一关联，其中BillingDetails类层次结构通过每个具体类一张表的策略和Hibernate中的隐式多态行为进行映射。你有CREDIT_CARD表和BANK_ACCOUNT表，但是没有BILLING_DETAILS表。Hibernate在USERS中需要两条信息，以便唯一地辨别被关联的默认CreditCard或者BankAccount：

- 被关联实例所在的表的名称；

- 被关联实例的标识符。

USERS表除了DEFAULT_BILLING_DETAILS_ID之外，还需要DEFAULT_BILLING_DEFAULTS_TYPE列。这个额外的列作用就像额外的辨别标志一样，并且在User.hbm.xml中还需要Hibernate <any>映射：

```
<any name="defaultBillingDetails"
     id-type="long"
     meta-type="string">
    <meta-value value="CREDIT_CARD" class="CreditCard"/>
    <meta-value value="BANK_ACCOUNT" class="BankAccount"/>
    <column name="DEFAULT_BILLING_DETAILS_TYPE"/>
    <column name="DEFAULT_BILLING_DETAILS_ID"/>
</any>
```

meta-type属性指定DEFAULT_BILLING_DEFAULTS_TYPE列的Hibernate类型；id-type属性指定DEFAULT_BILLING_DETAILS_ID列的类型（CreditCard和BankAccount必须有相同的标识符类型）。

<meta-value>元素告诉Hibernate如何解释DEFAULT_BILLING_DETAILS_TYPE列的值。在这里不需要使用完整的表名——可以使用你喜欢的任何值作为类型辨别标志。例如，可以把信息编码成两个字符：

```
<any name="defaultBillingDetails"
     id-type="long"
     meta-type="string">
    <meta-value value="CC" class="CreditCard"/>
    <meta-value value="CA" class="BankAccount"/>
    <column name="DEFAULT_BILLING_DETAILS_TYPE"/>
    <column name="DEFAULT_BILLING_DETAILS_ID"/>
</any>
```

这个表结构的例子如图7-15所示。

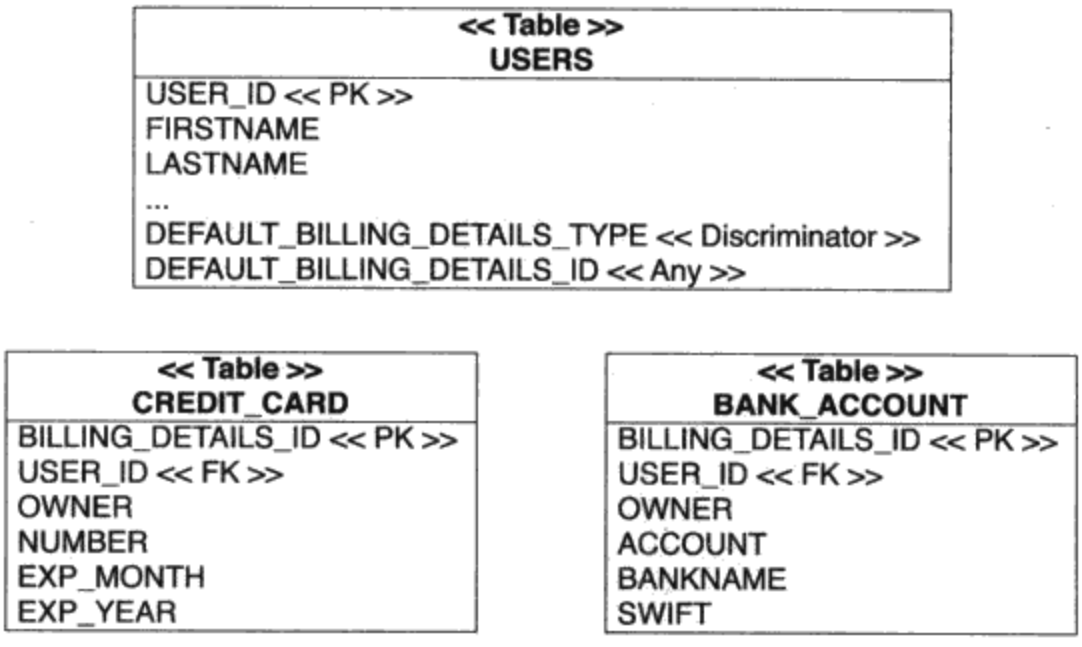


图7-15 通过任一关联使用辨别标志列

下面是使用这种关联时的第一个主要问题：你无法把外键约束添加到DEFAULT_BILLING_DETAILS_ID列，因为有些值引用BANK_ACCOUNT表，其他的则引用CREDIT_CARD表。这样，你就必须找出一些其他的方法来确保完整性（如触发器）。这与你使用<union-subclass>策略时要面对的问题相同。

此外，很难给这个关联编写SQL表联结。尤其是，Hibernate查询工具不支持这种关联映射，这个关联也不能通过外部联结而被抓取。除非最特殊的情况，否则我们不鼓励在任何情况下使用<any>关联。还要注意，这种映射技术不能通过注解或者在Java Persistence中使用（这个映射是如此罕见，因此目前为止还没有人要求注解支持它）。

如你所见，只要你不准备把关联创建到通过隐式多态映射的类层次结构，关联就很简单；通常不需要考虑它。你可能感到奇怪，前几节中竟然没有介绍任何JPA或者注解实例——运行时的行为是一样的，获得它不需要任何额外的映射。

7.4　小结

本章介绍了如何映射更复杂的实体关联。我们介绍的许多方法都是很少需要用到的，如果你可以简化类之间的关系，可能根本不需要用到它们。尤其是，多对多的实体关联通常最好表示为两个到中间实体类的一对多关联，或者利用组件的一个集合。

表7-1概括了原生的Hibernate特性和Java Persistence的比照。

表7-1　第7章中Hibernate和JPA对照表

Hibernate Core	Java Persistence和EJB 3.0
Hibernate对共享主键一对一关联映射支持键生成	支持标准的一对一映射。自动的共享主键生成需要通过Hibernate扩展才有可能
Hibernate支持跨联结表的所有实体关联映射	标准的关联映射可以跨二级表使用
Hibernate通过持久化索引支持列表的映射	持久化索引需要一个Hibernate扩展注解
Hibernate支持完全多态的行为。它对任一映射到一个通过隐式多态映射的继承层次结构的关联提供额外的支持	可用完全多态的行为，但是没有注解支持任一映射

下一章将关注遗留数据库的整合，以及如何定制让Hibernate自动为你生成的SQL。无论你是必须使用遗留的Schema，还是比如想要通过定制DDL改善新的Schema，这一章都值得关注。

第8章 遗留数据库和定制SQL

本章内容

- 遗留数据库整合和灵活映射
- SQL语句的定制
- 定制DDL改善SQL Schema

本章介绍的许多例子都是关于“困难”映射的。可能你第一次创建映射有问题时，是在使用一个不能被修改的遗留数据库Schema的时候。我们讨论在这样的场景中会遇到的典型问题，以及如何扭曲映射元数据，而不是改变应用程序或者数据库Schema。

我们还介绍如何覆盖Hibernate自动生成的SQL。这包括SQL查询、DML（创建、更新、删除）操作，以及Hibernate的自动DDL生成特性。你会看到如何映射存储过程和用户自定义的SQL函数，以及如何在数据库Schema中应用正确的完整性规则。如果你的数据库管理员需要完全的掌控，本节的内容将会特别有用（或者如果你是个数据库管理员，并想在SQL级优化Hibernate的话）。

如你所见，本章的主题很广泛，不必马上把它们全部读完。可以把本章的大部分内容当作参考资料，遇到特定的问题时再回到这里。

8.1 整合遗留数据库

我们希望本节能够涵盖处理现有的遗留数据库或者（这通常是一回事）奇怪的或者损坏的Schema时可能遇到的所有问题。然而，如果你的开发过程是自顶向下的，则可能想跳过本节。此外，建议你在尝试反向工程复杂的遗留Schema之前，先阅读有关类、集合和关联映射的所有章节。

必须提醒你：当应用程序继承现有的遗留数据库Schema时，通常应该尽可能地对现有的Schema少做一些改变。你对Schema所做的每一处改变都可能破坏访问数据库的其他现有应用程序。现有数据的迁移可能很昂贵，这也是你要考虑的东西。一般来说，构建一个新的应用程序不可能不对现有的数据模型做任何改变——新应用程序通常意味着额外的业务需求，自然需要数据库Schema的演变。

因此要考虑两种类型的问题：与改变业务需求相关的问题（不改变Schema一般无法解决）和只与你希望如何在新应用程序中表示相同业务问题相关的问题（这些问题虽不总是但大多数时候可以不用改变数据库Schema而解决）。很显然，第一种问题通常只要看看逻辑数据模型就可以看

见。第二种则更经常与逻辑数据模型实现为物理数据库Schema相关。

如果你接受这种观点，就会看到需要Schema改变的种种问题就是那些需要增加新实体、重构现有的实体、把新属性增加到现有的实体，以及修改实体之间的关联。这些可以不用改变Schema而解决的问题，通常涉及对特定实体不方便的表（或者列）的定义。本节将集中讨论这种问题。

假设你已经试过用Hibernate工具箱对现有的Schema进行反向工程，如2.3节中所述。接下来几节讨论的概念和解决方案假设你已经有基础的ORM，并且需要做些额外的变化使它生效。另一种方法，可以试着完全用手工编写映射，而不用反向工程工具。

从最明显的问题（遗留主键）开始。

8.1.1　处理主键

如前所述，我们认为自然主键可能是个馊主意。自然键经常使得在业务需求改变时很难重构数据模型。在极端的情况下，它们甚至可能影响性能。不幸的是，许多遗留Schema大量使用（自然的）复合键，并且由于我们反对使用复合键的缘故，可能很难改变遗留Schema来使用非复合的自然键或者代理键。

因此，Hibernate支持自然键的使用。如果自然键是一个复合键，支持就是通过`<composite-id>`映射。我们来映射复合的和非复合的这两种自然主键。

1．映射自然键

如果你在遗留Schema中遇到USERS表，USERNAME可能就是真正的主键。在这种情况下，就没有自动生成的代理标识符了。反之，你启用assigned标识符生成器策略，告诉Hibernate标识符是在对象被保存之前由应用分配的一个自然键：

```
<class name="User" table="USERS">
    <id name="username" column="USERNAME" length="16">
        <generator class="assigned"/>
    </id>

    ...
</class>
```

保存新User的代码如下：

```
User user = new User();
user.setUsername("johndoe"); // Assign a primary key value
user.setFirstname("John");
user.setLastname("Doe");
session.saveOrUpdate(user); // Will result in an INSERT
// System.out.println( session.getIdentifier(user) );
session.flush();
```

Hibernate如何知道saveOrUpdate()需要INSERT而不是UPDATE呢？它不知道，因此需要一个技巧：Hibernate在USERS表中查询指定的用户名。如果找到，Hibernate就更新行；如果没找到，就要插入新行。这当然不是最好的解决方案，因为它在数据库上触发了额外的瞬时干扰。

几种策略避免SELECT：

- 把`<version>`或者`<timestamp>`映射和一个属性添加到实体。Hibernate在内部通过乐观并

发控制来管理这两个值（本书稍后会讨论）。作为一项附带作用，空的时间戳（或者0或者NULL）版本表明实例是新的，并且必须插入而不是更新。

- 实现Hibernate Interceptor，并把它钩入到Session里面。这个扩展接口允许你实现方法isTransient()，它包含着你区分旧对象和新对象时可能需要的任何定制过程。

另一方面，如果喜欢显式地使用save()和update()而不是saveOrUpdate()，Hibernate就不必区分瞬时实例和脱管实例——你通过选择要调用的正确方法来进行。（实际上，这就是始终不用saveOrUpdate()的唯一原因。）

用JPA注解映射自然主键很简单：

```
@Id
private String username;
```

如果没有声明标识符生成器，Hibernate就会假设它必须应用一般的select-to-determine-state-unless-versioned[①]策略，并期待应用程序负责分配主键值。可以通过扩展包含拦截器的应用或者添加版本控制属性（版本号或者时间戳），再次避免SELECT。

复合的自然键按同样的思路进行扩展。

2. 映射复合自然键

假设USERS表的主键由USERNAME和DEPARTMENT_NR组成。可以添加一个具名departmentNr的属性到User类，并创建下列映射：

```
<class name="User" table="USERS">
    <composite-id>
        <key-property name="username"
                      column="USERNAME"/>

        <key-property name="departmentNr"
                      column="DEPARTMENT_NR"/>
    </composite-id>

    ...
</class>
```

保存新User的代码看起来如下：

```
User user = new User();

// Assign a primary key value
user.setUsername("johndoe");
user.setDepartmentNr(42);

// Set property values
user.setFirstname("John");
user.setLastname("Doe");
```

① 程序分配的标识符。如果需要应用程序分配标识符（而非Hibernate来生成），可使用assigned生成器。该生成器会使用已经分配给对象的标识符属性的标识符值。它使用自然键作为主键。这是没有指定<generator>元素时的默认行为。当选择assigned生成器时，除非有version或timestamp属性，或者你定义了Interceptor.isUnsaved()，否则需要让Hibernate使用unsavedvalue="undefined"，强制Hibernate查询数据库来确定一个实例是瞬时的，还是脱管的。——译者注

```
session.saveOrUpdate(user);
session.flush();
```

同样要记住，Hibernate执行一个SELECT来确定saveOrUpdate()应该做什么——除非你启用版本控制或者定制的Interceptor。但是调用load()或者get()时，可以（应该）使用什么对象作为标识符呢？例如，可能使用User类的一个实例：

```
User user = new User();

// Assign a primary key value
user.setUsername("johndoe");
user.setDepartmentNr(42);

// Load the persistent state into user
session.load(User.class, user);
```

在这个代码片段中，User表现得就像它自己的标识符类。定义一个只声明键属性的单独复合标识符类则更为优雅。调用这个类UserId：

```
public class UserId implements Serializable {
    private String username;
    private Integer departmentNr;

    public UserId(String username, Integer departmentNr) {
        this.username = username;
        this.departmentNr = departmentNr;
    }

    // Getters...

    public int hashCode() {
        int result;
            result = username.hashCode();
            result = 29 * result + departmentNr.hashCode();
            return result;
    }

    public boolean equals(Object other) {
        if (other==null) return false;
        if ( !(other instanceof UserId) ) return false;
        UserId that = (UserId) other;
        return this.username.equals(that.username) &&
            this.departmentNr.equals(that.departmentNr);
    }
}
```

正确地实现equals()和hashCode()很重要，因为Hibernate的高速缓存查找依赖这些方法。标识符类还需要实现Serializable接口。

现在从User和UserId属性中移除username和departmentNr属性。创建下列映射：

```
<class name="User" table="USERS">

    <composite-id name="userId" class="UserId">
        <key-property name="username"
                      column="USERNAME"/>

        <key-property name="departmentNr"
```

```
                        column="DEPARTMENT_NR"/>
    </composite-id>

    ...
</class>
```

用下列代码保存User的新实例：

```
UserId id = new UserId("johndoe", 42);

User user = new User();

// Assign a primary key value
user.setUserId(id);

// Set property values
user.setFirstname("John");
user.setLastname("Doe");

session.saveOrUpdate(user);
session.flush();
```

要使saveOrUpdate()起作用还是需要SELECT。下列代码展现了如何加载实例：

```
UserId id = new UserId("johndoe", 42);

User user = (User) session.load(User.class, id);
```

现在，假设DEPARTMENT_NR是一个引用DEPARTMENT表的外键，并且你希望在Java领域模型中把这个关联表示为多对一的关联。

3. 复合主键中的外键

建议你映射一个也是复合主键一部分的外键列，通过一般的<many-to-one>元素，并用insert="false" update="false"禁用该列的任何Hibernate插入或者更新，如下：

```
<class name="User" table="USER">

    <composite-id name="userId" class="UserId">
        <key-property name="username"
                      column="USERNAME"/>

        <key-property name="departmentId"
                      column="DEPARTMENT_ID"/>
    </composite-id>

    <many-to-one name="department"
                 class="Department"
                 column="DEPARTMENT_ID"
                 insert="false" update="false"/>
    ...
</class>
```

现在Hibernate在更新或者插入User时忽略department属性，但是你当然可以用johndoe.getDepartment()读取它。User和Department之间的关系现在通过UserId复合键类的departmentId属性得到托管：

```
UserId id = new UserId("johndoe", department.getId() );

User user = new User();

// Assign a primary key value
user.setUserId(id);
// Set property values
user.setFirstname("John");
user.setLastname("Doe");
user.setDepartment(department);

session.saveOrUpdate(user);
session.flush();
```

只有Department的标识符值对持久化状态有些影响；为了一致，调用setDepartment(department)；否则，你必须在清除之后从数据库中刷新对象来获取部门集。（在实际应用中，你可以把所有这些细节移到复合标识符类的构造器里面。）

另一种方法是<key-many-to-one>：

```
<class name="User" table="USER">

    <composite-id name="userId" class="UserId">
        <key-property name="username"
                      column="USERNAME"/>

        <key-many-to-one name="department"
                         class="Department"
                         column="DEPARTMENT_ID"/>
    </composite-id>
    ...
</class>
```

然而，在复合标识符类中有关联通常并不方便，因此除非特殊情况，否则不推荐这种方法。<key-many-to-one>构造在查询方面也有限制：你无法限制一个跨<key-many-to-one>联结的HQL或者Criteria的查询结果（虽然这些特性可能将在以后的Hibernate版本中得以实现）。

4．复合主键的外键

由于USERS有一个复合主键，任何引用外键也是复合的。例如，从Item到User（卖主）的关联现在通过复合外键映射。

通过下列从Item到User的关联映射，Hibernate可以从Java代码中隐藏这一细节：

```
<many-to-one name="seller" class="User">
    <column name="USERNAME"/>
    <column name="DEPARTMENT_ID"/>
</many-to-one>
```

User类所拥有的任何集合也都有复合外键——例如，反向关联，被这位用户出售的items：

```
<set name="itemsForAuction" inverse="true">
    <key>
        <column name="USERNAME"/>
        <column name="DEPARTMENT_ID"/>
    </key>
    <one-to-many class="Item"/>
</set>
```

注意，列的排列顺序很重要，应该与它们在User主键映射的<composite-id>元素中出现的顺序一致。

这样就完成了对Hibernate中基础复合键映射技术的讨论。利用注解映射复合键也几乎相同，但是同往常一样，微小的差别很重要。

5. 利用注解的复合键

JPA规范涵盖了处理复合键的策略。你有3种选择：

- 把标识符属性封装在一个单独的类里面，并把它标识为@Embeddable，就像一般的组件一样。在实体类中包括这个组件类型的一个属性，并利用@Id给一个应用分配的策略映射它。
- 把标识符属性封装在一个没有任何注解的单独类里面。在实体类中包括这个类型的一个属性，并用@EmbeddedId映射它。
- 把标识符属性封装在单个的类里面。现在——这与你通常在原生的Hibernate中所做的不同——复制实体类中的所有标识符属性。然后，用@IdClass注解实体类，并指定被封装的标识符类的名称。

第一种选择很简单。需要使来自前一节的UserId类变成是可嵌入的：

```
@Embeddable
public class UserId implements Serializable {
    private String username;
    private String departmentNr;
    ...
}
```

至于所有的组件映射，你可以在这个类的字段（或者获取方法）中定义额外的映射属性。为了映射User的复合键，通过省略@GenerateValue注解而对分配的应用设置生成策略：

```
@Id
@AttributeOverrides({
    @AttributeOverride(name   = "username",
                       column = @Column(name="USERNAME") ),
    @AttributeOverride(name   = "departmentNr",
                       column = @Column(name="DEP_NR") )
})
private UserId userId;
```

就像你在本书前面对一般的组件映射所做的那样，如果喜欢，可以覆盖组件类的特定属性映射。

第二种复合键映射策略不要求你给UserId主键类作标记。因此，这个类不需要@Embeddable和其他注解。在自己的实体中，你通过@EmbeddedId映射复合的标识符属性，仍然通过可选的覆盖：

```
@EmbeddedId
@AttributeOverrides({
    @AttributeOverride(name   = "username",
                       column = @Column(name="USERNAME") ),
    @AttributeOverride(name   = "departmentNr",
```

```
                        column = @Column(name="DEP_NR") )
})
private UserId userId;
```

在JPA XML描述符中，这个映射看起来如下：

```
<embeddable class="auction.model.UserId" access ="PROPERTY">
    <attributes>
        <basic name="username">
            <column name="UNAME"/>
        </basic>
        <basic name="departmentNr">
            <column name="DEPARTMENT_NR"/>
        </basic>
    </attributes>
</embeddable>

<entity class="auction.model.User" access="FIELD">
    <attributes>
        <embedded-id name="userId">
            <attribute-override name="username">
                <column name="USERNAME"/>
            </attribute-override>
            <attribute-override name="departmentNr">
                <column name="DEP_NR"/>
            </attribute-override>
        </embedded-id>
        ...
    </attributes>
</entity>
```

第三种复合键映射策略则更难理解一点，尤其对于那些经验丰富的Hibernate用户而言。首先，把所有标识符属性封装在一个单独的类里面——如前一种策略一样，这个类不需要额外的注解。现在复制实体类中的所有标识符属性：

```
@Entity
@Table(name = "USERS")
@IdClass(UserId.class)
public class User {

    @Id
    private String username;

    @Id
    private String departmentNr;

    // Accessor methods, etc.
    ...
}
```

Hibernate检查@IdClass，并挑出所有重复属性（通过比较名称和类型），作为标识符属性和主键的一部分。所有主键属性都通过@Id注解，并根据这些元素（字段或者获取方法）的位置，实体默认为字段或者属性访问。

注意，Hibernate XML映射中也有这最后一种策略；但是有点含糊：

```
<composite-id class="UserId" mapped="true">
    <key-property name="username"
                  column="USERNAME"/>
    <key-property name="departmentNr"
                  column="DEP_NR"/>
</composite-id>
```

你省略实体的标识符属性名称（因为没有），因此Hibernate内部处理标识符。利用mapped="true"，启用最后一种JPA映射策略，因此所有的键属性现在都要出现在User和UserId类中。

如果使用JPA XML描述符，这个复合标识符映射策略看起来如下：

```
<entity class="auction.model.User" access="FIELD">
    <id-class class="auction.model.UserId"/>
    <attributes>
        <id name="username"/>
        <id name="departmentNr"/>
    </attributes>
</entity>
```

由于我们没有给Java Persistence中定义的这最后一种策略找到令人信服的案例，因此只好假设它被添加到了规范中，来支持一些遗留的行为（EJB 2.x实体bean）。

复合外键也可能使用注解。先映射从Item到User的关联：

```
@ManyToOne
@JoinColumns({
  @JoinColumn(name="USERNAME", referencedColumnName = "USERNAME"),
  @JoinColumn(name="DEP_NR", referencedColumnName = "DEP_NR")
})
private User seller;
```

一般的@ManyToOne和这个映射之间的主要区别在于涉及的列数——这个顺序还是很重要，并且应该与主键列的顺序一致。然而，如果你对每个列声明referencedColumnName，顺序就不重要了，并且外键约束的源表和目标表都可以有不同的列名称。

从User到带有Item集合的反向映射甚至更为简单：

```
@OneToMany(mappedBy = "seller")
private Set<Item> itemsForAuction = new HashSet<Item>();
```

这个反向端需要mappedBy属性，就像通常对于双向的关联一样。因为这是反向端，因此它不需要任何列声明。

在遗留Schema中，外键经常不引用主键。

6. 外键引用非主键

通常，外键约束引用主键。外键约束是一个完整性规则，它保证被引用的表有一行所包含的键值与引用表和给定行中的键值相匹配。注意，外键约束可以自引用；换句话说，包含外键约束的列可以引用同一张表的主键列。（CaveatEmptor的CATEGORY表中的PAARAENT_CATEGORY就是一个例子。）

遗留Schema有时候会有不遵循简单的“外键引用主键”规则的外键约束。有时候，外键引用

非主键：一个简单的唯一列，一个自然的非主键。假设在CaveatEmptor中，你需要在USERS表中处理具名CUSTOMER_NR的遗留自然键列：

```
<class name="User" table="USERS">

    <id name="id" column="USER_ID">...</id>

    <property name="customerNr"
              column="CUSTOMER_NR"
              not-null="true"
              unique="true"/>

</class>
```

这个映射中对于你来说唯一可能陌生的是unique属性。这是Hibernate中的其中一个SQL定制选项；不在运行时使用它（Hibernate不做任何唯一性验证），而是通过hbm2dbl导出数据库Schema。如果你有个包含自然键的现有Schema，就假设它是唯一的。为了完整起见，你可以并且应该在映射元数据中重复如此重要的约束——或许有一天你会用它导出一个新的Schema。

相当于XML映射，你可以在JPA注解中声明一个列为unique：

```
@Column(name = "CUSTOMER_NR", nullable = false, unique=true)
private int customerNr;
```

在遗留Schema中可能遇到的下一个问题是，ITEM表有一个外键列SELLER_NR。在理想的世界中，你会期待用这个外键去引用USERS表的主键USER_ID。然而，在遗留Schema中，它可以引用自然的唯一键CUSTOMER_NR。需要用一个属性引用映射它：

```
<class name="Item" table="ITEM">

    <id name="id" column="ITEM_ID">...</id>

    <many-to-one name="seller" column="SELLER_NR"
                 property-ref="customerNr"/>

</class>
```

在更加怪异的Hibernate映射中你会遇到property-ref属性。它用来告诉Hibernate“这是具名属性的一个镜像”。在前一个例子中，Hibernate现在知道外键引用的目标了。你要进一步注意的是，property-ref要求目标属性要唯一，因此如前所述，这个映射需要unique="true"。

如果试图通过JPA注解映射这个关联，可以找一个相当于property-ref属性的等价物。用一个对自然键列CUSTOMER_NR的显式引用映射关联：

```
@ManyToOne
@JoinColumn(name="SELLER_NR", referencedColumnName = "CUSTOMER_NR")
private User seller;
```

现在Hibernate知道被引用的目标列是一个自然键，并相应地管理外键关系。

为了完成这个例子，利用User类中itemsForAuction集合的映射，使这两个类之间的关联映射变成双向。首先，这是以XML格式的：

```
<class name="User" table="USERS">

    <id name="id" column="USER_ID">...</id>

    <property name="customerNr" column="CUSTOMER_NR" unique="true"/>
```

```
        <set name="itemsForAuction" inverse="true">
            <key column="SELLER_NR" property-ref="customerNr"/>
            <one-to-many class="Item"/>
        </set>

    </class>
```

ITEM中的外键列再次通过一个对customerNr的属性引用而被映射。在注解中，这更加容易映射为一个反向端：

```
@OneToMany(mappedBy = "seller")
private Set<Item> itemsForAuction = new HashSet<Item>();
```

7. 复合外键引用非主键

有些遗留Schema甚至比前面讨论过的更为复杂：外键可能是一个复合键，并且设计为引用一个复合的自然的非主键！

假设USERS有一个自然的复合键，包括FIRSTNAME、LASTNAME和BIRTHDAY列。外键可以引用这个自然键，如图8-1所示。

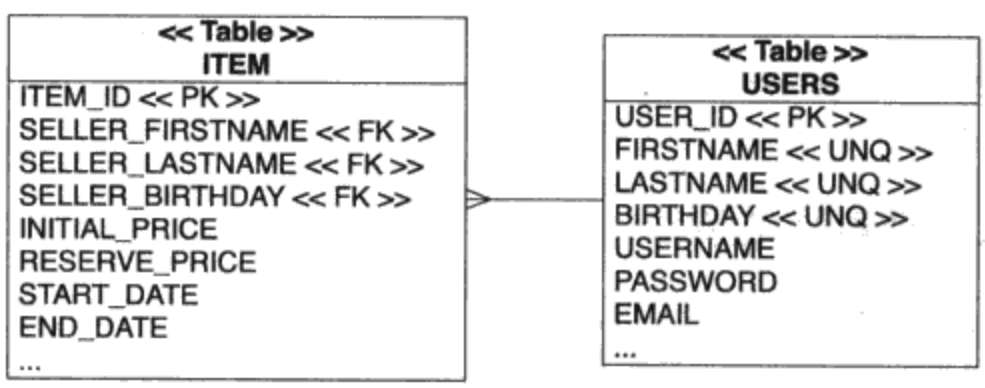


图8-1 复合外键引用复合的唯一键

为了映射这个，你需要按相同的名称给几个属性分组，否则无法在property-ref中给复合命名。应用<properties>元素来给映射分组：

```
<class name="User" table="USERS">

    <id name="id" column="USER_ID">...</id>

    <properties name="nameAndBirthday" unique="true" update="false">
        <property name="firstname" column="FIRSTNAME"/>
        <property name="lastname" column="LASTNAME"/>
        <property name="birthday" column="BIRTHDAY" type="date"/>
    </properties>

    <set name="itemsForAuction" inverse="true">
        <key property-ref="nameAndBirthday">
            <column name="SELLER_FIRSTNAME"/>
            <column name="SELLER_LASTNAME"/>
            <column name="SELLER_BIRTHDAY"/>
        </key>
        <one-to-many class="Item"/>
    </set>

</class>
```

如你所见，<properties>元素不仅可以用于给几个属性命名，而且定义了一个多列的unique约束，或者使几个属性变成不可变。对于关联映射，列的顺序仍然很重要：

```
<class name="Item" table="ITEM">

    <id name="id" column="ITEM_ID">...</id>

    <many-to-one name="seller" property-ref="nameAndBirthday">
        <column name="SELLER_FIRSTNAME"/>
        <column name="SELLER_LASTNAME"/>
        <column name="SELLER_BIRTHDAY"/>
    </many-to-one>

</class>
```

幸好，通过重构外键来引用主键的方式清除这样一个Schema经常很简单——如果你对数据库进行的改变能够不影响其他共享数据的应用。

这样就结束了我们对于在尝试映射遗留Schema时可能必须处理的自然、复合和外键相关的问题的探讨。让我们继续讨论其他值得关注的特殊映射策略。

有时候，你无法对遗留数据库进行任何改变——甚至不能创建表或者视图。Hibernate可以把类、属性，甚至关联的几个部分映射到一个简单的SQL语句或者表达式。我们称这种映射为公式映射（formula mapping）。

8.1.2　带有公式的任意联结条件

把Java工件映射到SQL表达式不仅仅对于整合遗留Schema有用。你已经创建过两个公式映射：第一个是一个简单的衍生只读属性映射，详见4.4.1节。第二个公式在一个继承映射中计算了辨别标志；请见5.1.3节。

现在将把公式应用于更为怪异的目的。记住，你现在将要见到的有些映射很复杂，在读完本书第二部分的所有章节之后可能会更容易理解它们。

1．理解用例

现在在两个实体之间映射一个文字联结条件。这听起来比实际应用程序更为复杂。看看图8-2中所示的两个类。

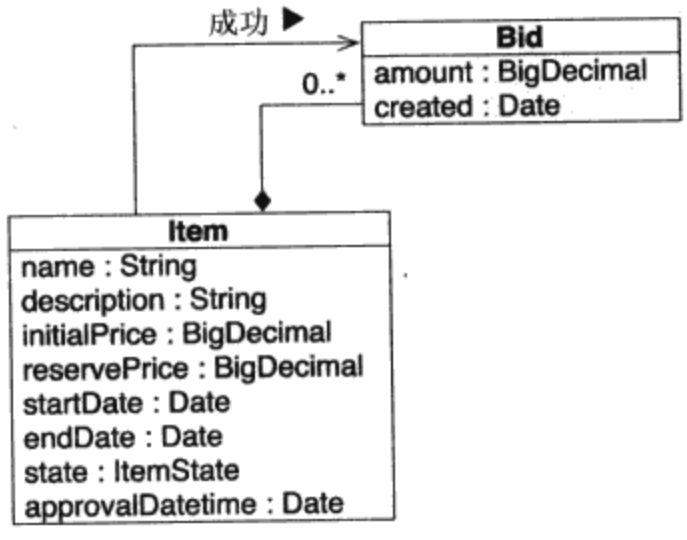


图8-2　引用多关联中一个实例的单个关联

特定的Item可以有几个Bid——这是个一对多的关联。但它不是两个类之间的唯一关联；另一个（单向的一对一关联）可用于挑选出一个特定的Bid实例作为胜出的出价。映射第一个关联，因为你想要能够通过调用anItem.getSuccessfulBid()获取一件拍卖货品的所有出价。逻辑上来说，集合中的其中一个元素也是被getSuccessfulBid()引用的成功的出价对象。

第一个关联很显然是个双向的一对多（多对一）关联，BID表中包含有一个外键ITEM_ID。（如果你之前还没有映射过它，请见6.4节。）

一对一的关联更为困难；可以用几种方式映射它。最自然的是ITEM表中一个被唯一约束的外键，它引用BID表中的一行——胜出的行，例如SUCCESSFUL_BID_ID列。

遗留Schema经常需要一个不是简单的外键关系的映射。

2. 映射公式联结条件

想象BID表中的每一行都有一个标志列，给胜出的出价做上标记，如图8-3所示。一个BID行把标记设置为true，且这件拍卖货品的所有其他行自然为false。在遗留Schema中很可能找不到对这个关系的约束或者完整性规则，但是我们现在不管它，而关注到Java类的映射。

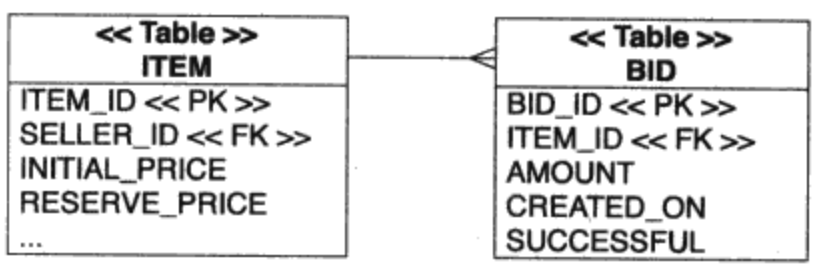


图8-3 胜出的出价用SUCCESSFUL列标志做上标记

为了使这个映射更值得关注，就假设遗留Schema没有使用SQL BOOLEAN数据类型，而是用CHAR(1)字段和值T（指true）和F（指false），模仿布尔的转换。你的目标是把这个标记列映射到Item类的一个successfulBid属性。要把它映射为对象引用，需要一个文字联结条件，因为Hibernate没有可以用于联结的外键。换句话说，对于每个ITEM行，你都需要从把SUCCESSFUL标记设置为T的BID表中联结一个行。如果没有这样的行，anItem.getSuccessfulBid()调用就返回null。

首先把Bid类和一个successful布尔属性映射到SUCCESSFUL数据库列：

```
<class name="Bid" table="BID">

    <id name="id" column="BID_ID"...

    <property name="amount"
              ...

    <properties name="successfulReference">

        <property name="successful"
                  column="SUCCESSFUL"
                  type="true_false"/>
                  ...

        <many-to-one name="item"
```

```
                    class="Item"
                    column="ITEM_ID"/>
                    ...
        </properties>

        <many-to-one name="bidder"
                     class="User"
                     column="BIDDER_ID"/>
                     ...
    </class>
```

type="true_false"属性在Java boolean基本（或者它的包装）属性和包含T/F文字值的一个简单的CHAR(1)列之间创建一个映射——它是个内建的Hibernate映射类型。你可以在其他映射中引用的名称下使用<properties>再次对若干属性进行组合。这里新出现的是，你可以组合<many- to-one>，而不只是那些基础的属性。

真正的技巧发生在另一端，对于Item类successfulBid属性的映射：

```
<class name="Item" table="ITEM">

    <id name="id" column="ITEM_ID"...

    <property name="initialPrice"
        ...

    <one-to-one name="successfulBid"
                property-ref="successfulReference">
        <formula>'T'</formula>
        <formula>ITEM_ID</formula>
    </one-to-one>

    <set name="bids" inverse="true">
        <key column="ITEM_ID"/>
        <one-to-many class="Bid"/>
    </set>

</class>
```

在这个例子中忽略<set>关联映射；这是Item和Bid之间常规的一对多双向关联，处于BID中的ITEM_ID外键列中。

说明　<one-to-one>不用于主键关联吗？通常，当两个实体表中的行有着相同的主键值时，<one-to-one>映射便是两个实体之间的主键关系。然而，通过使用一个包含property-ref的formula，你可以把它应用到外键关系。在本节所示的例子中，可以用<many-to-one>代替<one-to-one>元素，它仍然有效。

值得关注的部分是<one-to-one>映射，以及当你在使用关联的时候，它如何依赖property-ref和文字型formula值作为联结条件。

3. 使用关联

查询一件拍卖货品的获取和它的成功出价的完整的SQL看起来像这样：

```
select
    i.ITEM_ID,
    i.INITIAL_PRICE,
    ...
    b.BID_ID,
    b.AMOUNT,
    b.SUCCESSFUL,
    b.BIDDER_ID,
    ...
from
    ITEM i
left outer join
    BID b
        on 'T' = b.SUCCESSFUL
        and i.ITEM_ID = b.ITEM_ID
where
    i.ITEM_ID = ?
```

加载Item时，Hibernate现在通过应用一个涉及successfulReference属性列的联结条件，来从BID表联结一个行。因为这是个分组属性，你可以给所涉及的每个列以正确的顺序声明单独的表达式。第一个（'T'）是一个文字，如你从引号中所见到的。当Hibernate试图查出在BID表中是否有成功的行时，它现在把'T'=SUCCESSFUL包括在联结条件中。第二个表达式不是文字，而是一个列名称（没有引号）。因此，附上另一个联结条件i.ITEM_ID = b.ITEM_ID。如果需要额外的限制，可以把它扩展并添加更多的联结条件。

注意生成了一个外部联结，因为所讨论的货品不可能有成功的出价，因此给每个b.*列返回NULL。现在可以调用anItem.getSuccessfulBid()，来获取一个对成功出价的引用（或者如果不存在引用则为null）。

最后，用或者不用数据库约束，都无法只实现仅在Item实例的一个私有字段上设置值的item.setSuccessfulBid()方法。你必须在这个设置方法中实现一个小过程，负责这个特殊的关系和出价中的标记属性：

```
public class Item {
    ...

    private Bid successfulBid;
    private Set<Bid> bids = new HashSet<Bid>();

    public Bid getSuccessfulBid() {
        return successfulBid;
    }

    public void setSuccessfulBid(Bid successfulBid) {
        if (successfulBid != null) {

            for (Bid bid : bids)
                bid.setSuccessful(false);

            successfulBid.setSuccessful(true);
            this.successfulBid = successfulBid;
        }
```

```
        }
    }
```

调用setSuccessfulBid()时，你设置所有出价都不成功。这么做可能触发集合的加载——使用这一策略必须付出的代价。然后，这个新的成功出价被做上标记，并设置成一个实例变量。保存对象时，对标记的设置更新BID表中的SUCCESSFUL列。为了完成这个（并修复遗留Schema），数据库级的约束必须完成与这个方法相同的事。（本章的后面会回到这个话题。）

关于这个文字联结条件映射要记住的其中一点是，它可以被应用在许多其他的情形中，而不仅仅对于成功的或者默认的关系。每当需要把一些任意的联结条件附加到查询时，最好的选择是公式。例如，可以在<many-to-many>映射中用它创建从关联表到实体表的一个文字联结条件。

不幸的是，在编写本书之时，Hibernate Annotations还不支持用公式表达的任意联结条件。在一个引用名称下分组属性也是不可能的。我们希望这些特性一旦可用时，将十分类似于XML映射。

在遗留Schema中可能遇到的另一个问题是，它不能与类粒度很好地整合。通常建议的类要比表更多可能不起作用，并且可能你必须反其道而行之，把任意表联结到一个类里面。

8.1.3　联结任意的表

第5章的一个继承映射中已经介绍过<join>映射元素，请见5.1.5节。它帮助把特定子类的属性分到一张单独的表里面，处在主继承层次结构表之外。这项一般的功能有更多用途——但是我们必须提醒你，<join>也可能是个馊主意。任何设计适当的系统都应该是类比表更多。把单个类分离到几张单独的表中，是只有在你需要把遗留Schema中的几张表合并到单个类中时才应该做的事。

1. 把属性移到二级表中

假设在CaveatEmptor中，你没有把包含用户主要信息的用户地址信息保存在映射为组件的USERS表中，而是在一张单独的表中。如图8-4所示。注意，每个BILLING_ADDRESS都有一个外键USER_ID，反过来USER_ID也是BILLING_ADDRESS表的主键。

为了在XML映射它，需要把Address的属性组合在<join>元素中：

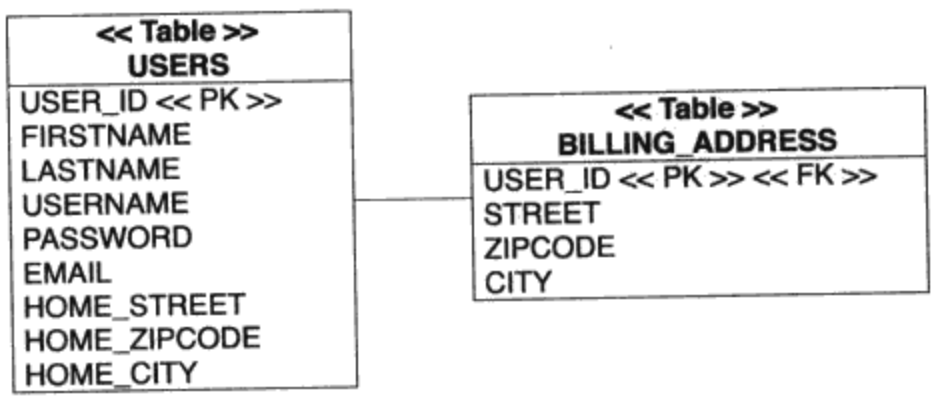


图8-4　把账单地址数据分到二级表

```
<class name="User" table="USERS">
    <id>...

    <join table="BILLING_ADDRESS" optional="true">
        <key column="USER_ID"/>
```

```
        <component name="billingAddress" class="Address">
            <property   name="street"
                        type="string"
                        column="STREET"
                        length="255"/>
            <property   name="zipcode"
                        type="string"
                        column="ZIPCODE"
                        length="16"/>
            <property   name="city"
                        type="string"
                        column="CITY"
                        length="255"/>
        </component>
    </join>

</class>
```

你不一定要联结组件；也可以联结单独的属性甚至<many-to-one>（我们在第7章中给可选的实体关联这么实现过）。通过设置optional="true"，表明组件属性对于没有billingAddress的User也可能为null，并且没有行应该被插入到这张二级表中。Hibernate还执行一个外部联结（而不是内部联结）来从二级表中获取行。如果在<join>映射中声明了fetch="select"，二级表就可以用于这一目的。

Java Persistence规范中也包括二级表的概念。首先，你必须对一个特定的实体声明一张（或者几张）二级表：

```
@Entity
@Table(name = "USERS")
@SecondaryTable(
    name = "BILLING_ADDRESS",
    pkJoinColumns = {
        @PrimaryKeyJoinColumn(name="USER_ID")
    }
)
public class User {
    ...
}
```

每张二级表都需要一个名称和一个联结条件。在这个例子中，外键列引用USERS表的主键列，就像前面在XML映射中的一样。（这是个默认的联结条件，因此可以只声明二级表名称，不需要其他东西。）你可能看到注解的语法开始有问题，并且代码也更难以阅读。所幸的是，你不必经常使用二级表。

实际的组件属性billingAddress，被映射为一个常规的@Embedded类，就像常规的组件一样。然而，需要覆盖每个组件属性列，并把它分配到User类中的二级表：

```
@Embedded
@AttributeOverrides( {
    @AttributeOverride(
        name   = "street",
        column = @Column(name="STREET",
                         table = "BILLING_ADDRESS")
```

```
        ),
        @AttributeOverride(
            name   = "zipcode",
            column = @Column(name="ZIPCODE",
                             table = "BILLING_ADDRESS")
        ),
        @AttributeOverride(
            name   = "city",
            column = @Column(name="CITY",
                             table = "BILLING_ADDRESS")
        )
    })
    private Address billingAddress;
```

这样就不再容易阅读了，但这是为在注解中使用声明元数据的映射灵活性所付出的代价。或者，可以使用JPA XML描述符：

```
<entity class="auction.model.User" access="FIELD">
    <table name="USERS"/>
    <secondary-table name="BILLING_ADDRESS">
        <primary-key-join-column
            referenced-column-name="USER_ID"/>
    </secondary-table>
    <attributes>
        ...
        <embedded name="billingAddress">
            <attribute-override name="street">
                <column name="STREET" table="BILLING_ADDRESS"/>
            </attribute-override>
            <attribute-override name="zipcode">
                <column name="ZIPCODE" table="BILLING_ADDRESS"/>
            </attribute-override>
            <attribute-override name="city">
                <column name="CITY" table="BILLING_ADDRESS"/>
            </attribute-override>
        </embedded>
    </attributes>
</entity>
```

另一种<join>元素甚至更为怪异的使用案例是被反向联结的属性或者组件。

2. 被反向联结的属性

假设在CaveatEmptor中，有具名DAILY_BILLING的遗留表。这张表包括所有开放支付，对任何拍卖都以每夜批量的方式执行。表里有一个到ITEM的外键列，如图8-5所示。

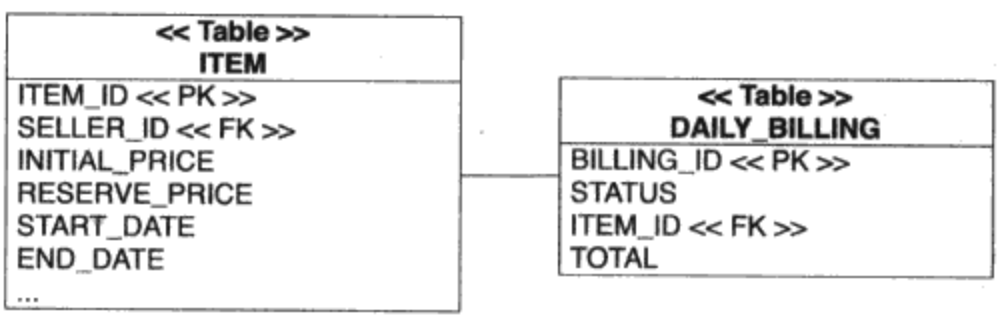


图8-5　每日账单摘要引用货品并包含总金额

每笔支付都包括TOTAL列，包含要收的总金额。在CaveatEmptor中，如果能够通过调用anItem.getBillingTotal()访问一次特定拍卖的价格就会很方便。

可以把这个列从DAILY_BILLING表映射到Item类中。然而，永远不要从这一端插入或者更新它；它是只读的。基于这个原因，把它映射为反向的——负责维持列值的另一端的简单镜像（假设你在这里没有映射它）：

```
<class name="Item" table="ITEM">
    <id>...

    <join table="DAILY_BILLING" optional="true" inverse="true">
        <key column="ITEM_ID"/>
        <property   name="billingTotal"
                    type="big_decimal"
                    column="TOTAL"/>
    </join>

</class>
```

注意，对于这个问题的另一种可以选择的解决方案是一个使用公式表达式和关联子查询的衍生属性：

```
<property name="billingTotal"
          type="big_decimal"
          formula="( select db.TOTAL from DAILY_BILLING db
                     where db.ITEM_ID = ITEM_ID )"/>
```

主要区别在于用来加载ITEM的SQL SELECT：如果启用<join fetch="select">，第一种解决方案则默认为一个外部联结，通过可选的另一个SELECT。这个衍生属性在原始查询的选择子句中产生一个嵌入的子查询。在编写本书之时，注解还不支持反向的联结映射，但是可以给公式使用Hibernate注解。

就像你或许可以从这些例子中所猜到的，<join>映射在多种情况下派上了用场。如果与公式结合，它们甚至更为强大，但是我们希望你不必经常使用这种结合。

在使用遗留数据的上下文中经常出现的进一步问题是数据库触发器。

8.1.4 使用触发器

甚至在全新的数据库中也有理由使用触发器，因此遗留数据不是触发器可能产生问题的唯一场景。使用ORM软件的触发器和对象状态管理通常是个问题，因为触发器可能在不合时机的时候运行，或者可能修改不与内存状态同步的数据。

1. 在插入时运行的触发器

假设ITEM表有一个CREATED列，被映射到类型Date的created属性，它通过一个在插入中自动执行的触发器初始化。下列映射很恰当：

```
<property name="created"
          type="timestamp"
          column="CREATED"
          insert="false"
          update="false"/>
```

注意，你映射这个属性insert="false" update="false"，表明它没有被Hibernate包括在SQL INSERT或者UPDATE中。

保存新Item之后，Hibernate不知道被触发器分配到这个列的值，因为它发生在INSERT该货品行之后。如果需要在应用程序中生成的值，就必须显式地告诉Hibernate用SQL SELECT重新加载该对象。例如：

```
Item item = new Item();
...
Session session = getSessionFactory().openSession();
Transaction tx = session.beginTransaction();

session.save(item);
session.flush(); // Force the INSERT to occur
session.refresh(item); // Reload the object with a SELECT

System.out.println( item.getCreated() );

tx.commit();
session.close();
```

涉及触发器的大部分问题都可能以这个方式解决，用一个显式的flush()强制立即执行触发器，可能接着调用refresh()来获取触发器的结果。

在把refresh()调用添加到应用程序之前，我们必须告诉你前一节的主要目标在于介绍什么时候使用refresh()。许多Hibernate的初学者不理解它真正的用途，经常不恰当地使用它。refresh()更为正式的定义是“利用数据库中的当前值，刷新处于持久化状态的内存实例。”

对于所介绍的例子，数据库触发器在插入之后填入了一个列值，这是可以使用的一种更为简单的技术：

```
<property name="created"
          type="timestamp"
          column="CREATED"
          generated="insert"
          insert="false"
          update="false"/>
```

通过注解，使用Hibernate扩展：

```
@Temporal(TemporalType.TIMESTAMP)
@org.hibernate.annotations.Generated(
    org.hibernate.annotations.GenerationTime.INSERT
)
@Column(name = "CREATED", insertable = false, updatable = false)
private Date created;
```

4.4.1节的“生成的和默认的属性值”已经深入讨论了generated属性。利用generated="insert"，Hibernate在插入之后自动执行SELECT，来获取更新过的状态。

当数据库执行触发器时，要进一步注意的问题是：脱管对象图的再关联和在每次UPDATE时运行的触发器。

2. 在更新时运行的触发器

在讨论ON UPDATE触发器与对象的重附结合使用之前，需要强调另外一项对generated属性

的设置：

```
<version name="version"
         column="OBJ_VERSION"
         generated="always"/>
...
<timestamp name="lastModified"
           column="LAST_MODIFIED"
           generated="always"/>
...
<property name="lastModified"
          type="timestamp"
          column="LAST_MODIFIED"
          generated="always"
          insert="false"
          update="false"/>
```

利用注解，与之相当的映射如下：

```
@Version
@org.hibernate.annotations.Generated(
    org.hibernate.annotations.GenerationTime.ALWAYS
)
@Column(name = "OBJ_VERSION")
private int version;

@Version
@org.hibernate.annotations.Generated(
    org.hibernate.annotations.GenerationTime.ALWAYS
)
@Column(name = "LAST_MODIFIED")
private Date lastModified;

@Temporal(TemporalType.TIMESTAMP)
@org.hibernate.annotations.Generated(
    org.hibernate.annotations.GenerationTime.ALWAYS
)
@Column(name = "LAST_MODIFIED", insertable = false, updatable = false)
private Date lastModified;
```

利用always，启用Hibernate的自动刷新，不仅针对行的插入还针对行的更新。换句话说，每当一个版本、时间戳、或者任何属性值由在UPDATE SQL语句中运行的触发器生成时，都需要启用这个选项。再次参考我们之前在4.4.1节中对生成的属性所进行的讨论。

看看如果你拥有在更新中运行的触发器可能会遇到的第二个问题。由于当脱管对象被重附到新的Session（通过update()或者saveOrUpdate()）时，没有快照可用，Hibernate可能执行不必要的SQL UPDATE语句，确保数据库状态与持久化的上下文状态同步。这可能导致不合时宜地触发一个UPDATE触发器。通过在映射中为被持久化到带有触发器的表的类启用select-before-update，可以避免这种行为。如果ITEM表有一个更新触发器，就把下列属性添加到映射中：

```
<class name="Item"
       table="ITEM"
```

```
        select-before-update="true">
    ...
</class>
```

这项设置强制Hibernate利用SQL SELECT来获取当前数据库状态的快照，如果内存Item的状态相同，则避免后面的UPDATE。你用额外的SELECT避免了不合时宜的UPDATE。

Hibernate注解启用同样的行为：

```
@Entity
@org.hibernate.annotations.Entity(selectBeforeUpdate = true)
public class Item { ... }
```

在试图映射遗留Schema之前，要注意更新之前的SELECT仅仅获取所述实体实例的状态。没有主动抓取任何集合或者关联实例，也没有任何预抓取优化是活动的。如果开始给系统中的多个实体启用selectBeforeUpdate，可能会发现通过未被优化的查询引入的性能问题是有问题的。更好的一种策略是使用合并而不是重附。然后Hibernate可以在获取数据库快照时应用一些优化（外部联结）。我们会在本书的后面更深入地讨论重附和合并之间的区别。

我们来概括一下对遗留数据模型的讨论：Hibernate提供几种轻松处理（自然的）复合键和不合时宜的列的策略。在尝试映射遗留Schema之前，我们建议要细心检查，看是否有可能改变Schema。依我们的经验，许多开发人员立即就认为改变数据库Schema太过复杂和费时，转而寻求一种Hibernate解决方案。有时候这样并不正确，应该把Schema演变当作Schema生命周期很自然的一部分。如果表改变了，那么数据导出、一些转换和导入便可能解决问题。从长期来看，一天的工作可能节省几天的工作。

遗留Schema经常也需要由Hibernate生成的SQL定制，用于数据操作（DML）或者Schema定义（DDL）。

8.2　定制 SQL

出现于20世纪70年代的SQL，直到1986年才被（ANSI）标准化。虽然SQL标准的每一次更新都带来（许多有争议的）新特性，每一个支持SQL的DBMS产品都以它自己独特的方式这么做着。可移植性的重担再次落在了数据库应用程序开发人员的肩上。这就是Hibernate有用的地方：它的内建查询机制、HQL和Criteria API，根据所配置的数据库方言生成了SQL。所有其他自动生成的SQL（例如，当集合必须按需获取时）也在方言的帮助下生成。利用一个简单的方言转换，可以在一个不同的DBMS上运行应用。

为了支持这种可移植性，Hibernate必须处理三种操作：

- 每个数据获取操作导致正被执行的SELECT语句。它可能有多种变形。例如，数据库产品可能给联结操作使用一种不同的语法，或者结果如何被限制到特定数目的行。
- 每个数据修改都需要执行DML语句，例如UPDATE、INSERT和DELECT。DML经常不像数据获取那么复杂，但是它仍然具有特定于产品的变形。
- 数据库Schema必须在可以执行DML和数据获取之前创建或者改变。使用DDL在数据库目录上工作；它包括如CREATE、ALTER和DROP这样的语句。DDL几乎完全是特定于供应商的，但是大部分产品至少有一个类似的语法结构。

我们经常使用的另一个术语是CRUD，指create（创建）、read（读取）、update（更新）和delete（删除）。Hibernate为你、为所有CRUD操作和模式定义生成所有这些SQL。这种转化基于`org.hibernate.dialect.Dialect`实现——Hibernate为所有流行的SQL数据库管理系统捆绑了方言。我们鼓励你看看正在使用的方言的源代码；它并不难读。一旦你对使用Hibernate更有经验之后，可能甚至想要扩展一种方言，或者编写自己的方言。例如，想要注册一个定制SQL函数用在HQL查询中，要用一个新的子类扩展现有的方言，并添加注册代码——还是要检查现有的源代码，找出更多有关方言系统的灵活性。

另一方面，当你要在一个更低级的抽象上工作时，有时候需要比Hibernate API（或者HQL）提供的更多控制。通过Hibernate，可以覆盖或者完全替换所有将被执行的CRUD SQL语句。如果依赖Hibernate的自动Schema导出工具（不一定），就可以定制和扩展所有定义Schema的DDL SQL语句。

此外，Hibernate允许你始终通过`session.connection()`获取简单的JDBC `Connection`对象。当没有其他东西可行，或者任何其他东西都比简单的JDBC更难时，就应该把这项特性用作最后的杀手锏。最新的Hibernate版本，是有幸十分罕见的，因为它给典型的无状态JDBC操作（例如大批量的更新和删除）内建了越来越多的特性，并且已经有了用于定制SQL的许多扩展点。

这些定制的SQL（DML和DDL）都是本节的话题。我们从用于创建、读取、更新和删除操作的定制DML开始。随后整合数据库的存储过程去做相同的工作。最后，看看给数据库Schema自动生成的DDL定制，以及如何创建表示数据库管理员优化工作的一个好起点的Schema。

注意，在编写本书之时，注解中还没有自动生成的SQL的这个详细定制；因此，接下来的例子将专门使用XML元数据。我们期待Hibernate Annotations的未来版本将更好地支持SQL定制。

8.2.1 编写定制 CRUD 语句

你要编写的第一个定制SQL用来加载实体和集合。（下列示例中的大部分代码显示了与Hibernate默认执行的几乎相同的SQL，没有多少定制——这有助于你更快地理解映射技术。）

1. 用定制SQL加载实体和集合

对于需要用定制SQL操作来加载实例的每个实体类，定义对具名查询的`<loader>`引用：

```
<class name="User" table="USERS">
    <id name="id" column="USER_ID"...
    <loader query-ref="loadUser"/>
    ...
</class>
```

`loadUser`查询现在可以被定义在映射元数据中的任何地方，从它的用途中分离出来封装。这是一个简单查询的例子，它给`User`实体实例获取数据：

```
<sql-query name="loadUser">
    <return alias="u" class="User"/>
    select
        us.USER_ID      as {u.id},
        us.FIRSTNAME    as {u.firstname},
```

```
        us.LASTNAME     as {u.lastname},
        us.USERNAME     as {u.username},
        us."PASSWORD"   as {u.password},
        us.EMAIL        as {u.email},
        us.RANKING      as {u.ranking},
        us.IS_ADMIN     as {u.admin},
        us.CREATED      as {u.created},
        us.HOME_STREET  as {u.homeAddress.street},
        us.HOME_ZIPCODE as {u.homeAddress.zipcode},
        us.HOME_CITY    as {u.homeAddress.city},
        us.DEFAULT_BILLING_DETAILS_ID as {u.defaultBillingDetails}
    from
        USERS us
    where
        us.USER_ID = ?
</sql-query>
```

如你所见，从列名称到实体属性的映射使用了一个简单的别名。在具名的加载程序中查询实体，必须SELECT以下列和属性：

- 主键列和主键属性或者属性，如果使用复合主键的话。
- 所有标量属性，必须从它们相应的列中被初始化。
- 所有必须被初始化的复合属性。可以利用下列别名语法处理单独的标量元素：{entityalias.componentProperty.scalarProperty}。
- 所有外键列，必须被获取并映射到相应的many-to-one属性。请见前一个片段中的DEFAULT_BILLING_DEFAULTS_ID实例。
- 所有标量属性、复合属性和<join>元素内部的多对一实体引用。如果所有被联结的属性永远不为NULL，就使用一个联结到二级表的内部联结；否则，用外部联结比较恰当。（注意这在本例中并没有体现出来。）
- 如果你通过字节码基础设施给标量属性启用了延迟加载，就不需要加载延迟属性。请见13.1.6节。

前一个例子中所示的{propertyName}别名并非绝对必要。如果结果中的一个列名与被映射列的名称相同，Hibernate就会自动把它们绑到一起。

甚至可以在应用程序中利用session.getNamedQuery("loadUser")，按名称调用被映射的查询。利用定制的SQL查询，更多的事情成为可能，但是我们在本节中关注对CRUD的基础SQL定制。15.2节将回到这个话题。

假设你也想定制用来加载集合的SQL——例如，被一位User售出的items。首先，在集合映射中声明一个装载程序引用：

```
<set name="items" inverse="true">
    <key column="SELLER_ID" not-null="true"/>
    <one-to-many class="Item"/>
    <loader query-ref="loadItemsForUser"/>
</set>
```

具名查询loadItemsForUser看起来几乎与实体装载程序相同：

```
<sql-query name="loadItemsForUser">
    <load-collection alias="i" role="User.items"/>
    select
        {i.*}
    from
        ITEM i
    where
        i.SELLER_ID = :id
</sql-query>
```

有两个重要的区别：一个是从别名映射到集合角色的<load-collection>；它应该是一目了然的。这个查询中新出现的东西是，从SQL表别名ITEM i到所有包含{i.*}的货品属性的自动映射。通过使用相同的别名（符号i），创建了两者之间的一个联结。此外，你现在正在使用具名参数:id，而不是包含问号标记的简单的定位参数。可以使用自己喜欢的任何一种语法。

有时候，通过外部联结，可以在单个查询中更好地加载实体实例和集合（实体可能有一个空的集合，因此无法使用内部联结）。如果想要应用这个主动抓取，就不要为集合声明装载程序引用。实体装载程序负责集合的获取：

```
<sql-query name="loadUser">
    <return alias="u" class="User"/>
    <return-join alias="i" property="u.items"/>
    select
        {u.*}, {i.*}
    from
        USERS u
        left outer join ITEM i
            on u.USER_ID = i.SELLER_ID
    where
        u.USER_ID = ?
</sql-query>
```

注意如何使用<return-join>元素把别名绑定到实体的集合属性，有效地把两个别名链接在一起。还要进一步注意，如果想要在一个原始查询中主动抓取一对一和多对一关联的实体，这个方法也有效。在这种情况下，如果被关联的实体是强制的（外键不能为NULL），你可能想要用内部联结；或者如果目标是可选的，则可能要用外部联结。可以在查询中即时获取许多个单端的关联；然而，如果（外部）联结了不止一个集合，就会创建一个笛卡儿积，有效地乘以所有的集合行。这样会产生可能比两个查询更慢的巨大结果。当我们在第13章中讨论抓取策略时，你会再次见到这一限制。

如前所述，你会在本书的后面见到更多用于对象加载的SQL选项。现在来讨论插入、更新和删除操作的定制，完成CRUD基础的讨论。

2. 定制插入、更新和删除

Hibernate在启动时生成所有琐碎的CRUD SQL。它在内部高速缓存SQL声明以备未来之用，这样避免了对最普通操作的SQL生成的任何运行时成本。你已经知道如何覆盖CRUD中的R（读取），因此让我们对CUD进行同样的讨论。

对于给每个实体或者集合，可以在<sql-insert>、<sql-delete>和<sql-update>内部相

应地定义定制的CUD SQL语句：

```
<class name="User" table="USERS">

    <id name="id" column="USER_ID"...

    ...

    <join table="BILLING_ADDRESS" optional="true">
        <key column="USER_ID"/>
        <component name="billingAddress" class="Address">
            <property ...
        </component>

        <sql-insert>
            insert into BILLING_ADDRESS
                            (STREET, ZIPCODE, CITY, USER_ID)
            values (?, ?, ?, ?)
        </sql-insert>

        <sql-update>...</sql-update>

        <sql-delete>...</sql-delete>

    </join>

    <sql-insert>
        insert into USERS (FIRSTNAME, LASTNAME, USERNAME,
                           "PASSWORD", EMAIL, RANKING, IS_ADMIN,
                           CREATED, DEFAULT_BILLING_DETAILS_ID,
                           HOME_STREET, HOME_ZIPCODE, HOME_CITY,
                           USER_ID)
            values (?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?)
    </sql-insert>

    <sql-update>...</sql-update>

    <sql-delete>...</sql-delete>

</class>
```

这个映射示例可能看起来有点复杂，但实际上很简单。你在单个映射中有两张表：主表用于实体USERS，二级表BILLING_ADDRESS来自本章前面的遗留映射。每当有用于实体的二级表时，就必须把它包括在任何定制SQL里面——因而<sql-insert>、<sql-delete>和<sql-update>元素既在映射的<class>部分，又在它的<join>部分。

下一个问题是给语句绑定参数。对于CUD SQL定制，在编写本书之时还只支持占位参数。但是什么是参数的正确顺序？Hibernate如何把实参绑定到SQL参数有一个内在的顺序。找到正确的SQL语句和参数顺序的最容易方法是让Hibernate为你生成一个。从映射文件中移除定制SQL，给org.hibernate.persister.entity包启用DEBUG日志，并观察（或者搜索）Hibernate启动日志，查找类似于这些的行：

```
AbstractEntityPersister - Insert 0: insert into USERS (FIRSTNAME,
 LASTNAME, USERNAME, "PASSWORD", EMAIL, RANKING, IS_ADMIN,
 CREATED, DEFAULT_BILLING_DETAILS_ID, HOME_STREET, HOME_ZIPCODE,
 HOME_CITY, USER_ID) values (?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?)
```

```
AbstractEntityPersister - Update 0: update USERS set
 FIRSTNAME=?, LASTNAME=?, "PASSWORD"=?, EMAIL=?, RANKING=?,
 IS_ADMIN=?, DEFAULT_BILLING_DETAILS_ID=?, HOME_STREET=?,
 HOME_ZIPCODE=?, HOME_CITY=? where USER_ID=?
...
```

你现在可以复制想要定制到映射文件中的语句，并进行必要的改变。更多有关Hibernate中日志的信息，请参考2.1.3节。

你现在已经把CRUD操作映射到了定制的SQL语句。另一方面，动态的SQL并不是获取和操作数据的唯一方法。保存在数据库中预设的和编译好的过程也可以被映射到对实体和集合的CRUD操作。

8.2.2 整合存储过程和函数

存储过程在数据库应用程序开发中很常见。把代码移近数据并在数据库内部执行它有着明显的好处。

首先，你不一定要在访问数据的每个程序中复制功能和逻辑。一种不同的观点是许多业务逻辑不应该被重复，因此它始终可以被应用。这包括保证数据完整性的过程（例如，复杂到难以声明式地实现的约束）。你通常也会在具有过程完整性规则的数据库中发现触发器。

存储过程在大量数据上的所有处理中都有优势，例如生成报表和统计分析。你应该始终努力避免在网络上以及在数据库与应用程序服务器之间设置大数据，因此存储过程是大数据操作自然的选择。或者，你可以实现一个复杂的数据获取操作，在它把最终的结果传递到应用程序客户端之前，通过几个查询把数据装配起来。

另一方面，你将经常看到利用存储过程实现甚至最基础的CRUD操作的（遗留）系统。作为它的一种变形，不允许任何直接的SQL DML但只允许存储过程调用的系统，也（且有时候仍然）有它们的用处。

可以开始给CRUD或者大量数据操作整合现有的存储过程，或者可以先开始编写自己的存储过程。

1. 编写过程

用于存储过程的编程语言通常是专有的。Oracle PL/SQL（SQL的一种过程方言）非常流行（且适用于其他数据库产品的变形）。有些数据库甚至支持用Java编写的存储过程。标准的Java存储过程是SQLJ成果的一部分，可惜还没有成功。

在本节中你将使用最常见的存储过程系统：Oracle数据库和PL/SQL。结果表明Oracle中的存储过程，就像众多的其他东西一样，始终与你期待的结果会有所不同；每当有些东西需要特别的关注时，我们会提醒你。

用PL/SQL编写的存储过程必须在数据库目录中作为源代码创建，然后编译。先来编写一个可以加载所有符合特定标准的User实体的存储过程：

```
<database-object>
    <create>
        create or replace procedure SELECT_USERS_BY_RANK
```

```
(
 OUT_RESULT out SYS_REFCURSOR,
 IN_RANK    in  int
) as
begin
 open OUT_RESULT for
 select
     us.USER_ID          as USER_ID,
     us.FIRSTNAME        as FIRSTNAME,
     us.LASTNAME         as LASTNAME,
     us.USERNAME         as USERNAME,
     us."PASSWORD"       as PASSWD,
     us.EMAIL            as EMAIL,
     us.RANKING          as RANKING,
     us.IS_ADMIN         as IS_ADMIN,
     us.CREATED          as CREATED,
     us.HOME_STREET      as HOME_STREET,
     us.HOME_ZIPCODE     as HOME_ZIPCODE,
     us.HOME_CITY        as HOME_CITY,
     ba.STREET           as BILLING_STREET,
     ba.ZIPCODE          as BILLING_ZIPCODE,
     ba.CITY             as BILLING_CITY,
     us.DEFAULT_BILLING_DETAILS_ID
                         as DEFAULT_BILLING_DETAILS_ID
 from
     USERS us
 left outer join
     BILLING_ADDRESS ba
         on us.USER_ID = ba.USER_ID
    where
        us.RANKING >= IN_RANK;
end;
</create>
<drop>
    drop procedure SELECT_USERS_BY_RANK
</drop>
</database-object>
```

你在<database-object>元素中给存储过程嵌入了DDL，用于创建和移除。这样，在数据库Schema利用hbm2ddl工具被创建和更新时，Hibernate就会自动创建和删除过程。也可以用手工在数据库目录中执行DDL。如果使用一个没有现有存储过程的非遗留系统时，把它留在映射文件（在任何适当的位置，例如在MyStoredProcedures.htm.xml）中是个好办法。我们会在本章稍后回到<database-object>映射的其他选项。

跟以前一样，这个例子中的存储过程代码很简单：一个针对基表（User类的主表和二级表）的联结查询和RANKING限制（一个输入参数到过程）。

对于在Hibernate中映射的存储过程你必须遵守一些规则。存储过程支持IN和OUT参数。如果你通过Oracle自己的JDBC驱动程序使用存储过程，Hibernate要求存储过程的第一个参数是OUT；对于要被假定用于查询的存储过程，查询结果被认为是要在这个参数中返回。在Oracle 9或者更新的版本中，OUT参数的类型必须为SYS_REFCURSOR。在Oracle更早的版本中，你必须首先定义

自己的引用光标类型，称作REF CURSOR——在Oracle产品文档中可以找到许多例子。所有其他主要的数据库管理系统（和用于并非来自Oracle的Oracle DBMS的驱动程序）与JDBC兼容，可以在存储过程中不用OUT参数直接返回结果。例如，Microsoft SQL Server中一个类似的过程看起来如下：

```
create procedure SELECT_USERS_BY_RANK
  @IN_RANK int
  as
    select
       us.USER_ID          as USER_ID,
       us.FIRSTNAME        as FIRSTNAME,
       us.LASTNAME         as LASTNAME,
    ...
from
   USERS us
where us.RANKING >= @IN_RANK
```

把这个存储过程映射到Hibernate中的具名查询。

2. 利用过程查询

用于查询的存储过程被映射为一般的具名查询，但有一些微小的区别：

```
<sql-query name="loadUsersByRank" callable="true">
    <return alias="u" class="User">
        <return-property name="id"           column="USER_ID"/>
        <return-property name="firstname"    column="FIRSTNAME"/>
        <return-property name="lastname"     column="LASTNAME"/>
        <return-property name="username"     column="USERNAME"/>
        <return-property name="password"     column="PASSWD"/>
        <return-property name="email"        column="EMAIL"/>
        <return-property name="ranking"      column="RANKING"/>
        <return-property name="admin"        column="IS_ADMIN"/>
        <return-property name="created"      column="CREATED"/>
        <return-property name="homeAddress">
            <return-column name="HOME_STREET"/>
            <return-column name="HOME_ZIPCODE"/>
            <return-column name="HOME_CITY"/>
        </return-property>
        <return-property name="billingAddress">
            <return-column name="BILLING_STREET"/>
            <return-column name="BILLING_ZIPCODE"/>
            <return-column name="BILLING_CITY"/>
        </return-property>
        <return-property name="defaultBillingDetails"
                         column="DEFAULT_BILLING_DETAILS_ID"/>
    </return>
    { call SELECT_USERS_BY_RANK(?, :rank) }
</sql-query>
```

与一般的SQL查询映射相比，第一个区别是callable="true"属性。它在Hibernate中启用了对可调用语句的支持，并纠正了存储过程的输出的处理。下列映射把在过程结果中返回的列名称绑定到User对象的属性。需要特别考虑的一个特定案例：如果在类中出现一些多列属性，包括几个组件（homeAddress），那么需要按正确的顺序映射它们的列。例如，homeAddress属性通过

三个属性映射为<component>，每个属性都映射到它自己的列。因此，存储过程映射包括绑定到homeAddress属性的三个列。

存储过程的call准备了OUT（问号标记）和具名的输入参数。如果你没有使用Oracle JDBC驱动（其他驱动或者不同的DBMS），就不需要保存第一个OUT参数；结果可以直接从存储过程中返回。

看看User类的常规类映射。注意，这个例子中被过程返回的列名与你已经映射过的列名相同。可以省略每个属性的绑定，并让Hibernate自动负责映射：

```
<sql-query name="loadUsersByRank" callable="true">
    <return class="User"/>
   { call SELECT_USERS_BY_RANK(?, :rank) }
</sql-query>
```

给所有属性以及与常规映射中同名的类的外键关联返回正确列的责任，现在转移到了存储过程代码上。因为你在存储过程中已经有别名（select ... us.FIRSTNAME as FIRSTNAME...），这很简单。或者，只要过程结果返回的列中有一些列有着与已经映射为属性的列不同的名称，就只需要声明下面这些代码：

```
<sql-query name="loadUsersByRank" callable="true">
    <return class="User">
        <return-property name="firstname"    column="FNAME"/>
        <return-property name="lastname"     column="LNAME"/>
   </return>
   { call SELECT_USERS_BY_RANK(?, :rank) }
</sql-query>
```

最后，来看看存储过程的call。这里使用的语法{ call PROCEDURE()}，定义在SQL标准中，是可移植的。对Oracle有效的一种不可移植的语法是begin PROCEDURE(); end;。建议你始终使用可移植的语法。过程有两个参数，如前所述，第一个被保存为输出参数，因此你用一个占位参数符号（?）。如果给Oracle JDBC驱动配置了一种方言，Hibernate就会负责这个参数。第二个是执行调用时必须提供的输入参数。可以只使用占位参数，或者混合使用具名和占位参数。为了方便阅读，我们更喜欢具名参数。

在应用程序中利用这个存储过程的查询看起来就像任何其他具名查询执行一样：

```
Query q = session.getNamedQuery("loadUsersByRank");
q.setParameter("rank", 12);
List result = q.list();
```

在编写本书之时，被映射的存储过程可以被启用为具名查询，如你在本节前面所做的那样，或者启用为用于实体的装载程序，类似于你在前面映射过的loadUser示例。

存储过程不仅可以查询和加载数据，还可以操作数据。第一种使用案例是大数据操作，在数据库层中执行。不应该在Hibernate中映射它，而是应该通过简单的JDBC：session.connection().prepareCallableStatement();等来执行它。在Hibernate中可以映射的数据操作是实体对象的创建、删除和更新。

3. 把CUD映射到过程

你前面为类把<sql-insert>、<sql-delete>和<sql-update>元素映射到定制SQL语句。如果不喜欢对这些操作使用存储过程，就把映射改为可调用的语句：

```
<class name="User">
    ...
    <sql-update callable="true" check="none">
        { call UPDATE_USER(?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?) }
    </sql-update>
</class>
```

使用Hibernate的当前版本时，你面临着和以前相同的问题：把值绑定到占位参数。首先，存储过程必须有与Hibernate期待相同的输入参数数量（就像前面介绍的一样，启用SQL日志，获取可以复制和粘贴的生成语句）。这些参数必须再次按照Hibernate期待的相同顺序。

考虑check="none"属性。为了正确的（且如果启用了）乐观锁，Hibernate需要知道这个定制更新操作是否成功。通常，对于动态生成的SQL，Hibernate查看从操作中返回的被更新行的数量。如果操作没有或者无法更新任何行，就出现乐观锁失败。如果你编写自己的定制SQL操作，也可以定制这个行为。

利用check="none"，Hibernate期待你的定制过程内部处理失败的更新（例如，通过给需要被更新的行做一个版本检查），并期待过程在有东西出错时抛出异常。Oracle中的这种过程如下：

```
<database-object>
<create>
   create or replace procedure UPDATE_USER
    (IN_FIRSTNAME  in varchar,
     IN_LASTNAME   in varchar,
     IN_PASSWORD   in varchar,
     ...
    )
  as
    rowcount INTEGER;
  begin

     update USERS set
       FIRSTNAME    = IN_FIRSTNAME,
       LASTNAME     = IN_LASTNAME,
       "PASSWORD"   = IN_PASSWORD,
     where
       OBJ_VERSION = ...;

     rowcount := SQL%ROWCOUNT;
     if rowcount != 1 then
         RAISE_APPLICATION_ERROR( -20001, 'Version check failed');
     end if;

  end;

</create>
<drop>
    drop procedure UPDATE_USER
```

```
        </drop>
    </database-object>
```

SQL错误被Hibernate抓住，并被转换为一个随后可以在应用程序代码中处理的乐观锁异常。检查属性的其他选项如下：

- 如果启用check="count"，Hibernate就会利用简单的JDBC API检查被修改的行数。这是默认的，在你编写动态的SQL而没用存储过程时使用。
- 如果启用check="param"，Hibernate就会保存一个OUT参数，获取存储过程调用的返回值。你需要把一个额外的问号添加到调用，并在存储过程中返回这（第一）个OUT参数中DML操作的行数。然后Hibernate为你验证被修改的行数。

对插入和删除的映射很类似；所有这些都必须声明乐观锁检查如何执行。你可以从Hibernate启动日志中复制一个模板，获得参数的正确顺序和数量。

最后，也可以在Hibernate中映射存储函数。它们有着稍微不同的语义和用例。

4．映射存储函数

存储函数只有输入参数——没有输出参数。但是它可以返回值。例如，存储函数可以返回用户的等级：

```
<database-object>
    <create>
        create or replace function GET_USER_RANK
         (IN_USER_ID int)
        return int is
         RANK int;
        begin
            select
                RANKING
            into
                RANK
            from
                USERS
            where
                USER_ID = IN_USER_ID;

            return RANK;
        end;
    </create>
    <drop>
        drop function GET_USER_RANK
    </drop>
</database-object>
```

这个函数返回一个标量数。返回标量的存储函数的主要用例在常规的SQL或者HQL查询中嵌入调用。例如，可以获取比指定用户等级更高的所有用户：

```
String q = "from User u where u.ranking > get_user_rank(:userId)";
List result = session.createQuery(q)
                .setParameter("userId", 123)
                .list();
```

这是用HQL编写的查询；由于WHERE子句中（而不是在任何其他的子句中）用于函数调用的

传递（pass-through）功能，你可以直接调用数据库中的任何存储函数。函数的返回类型应该与操作相匹配：在这个例子中，包含ranking属性（也是数字）的大于比较。

如果函数返回一个结果集游标，如前几节一样，你甚至可以把它映射为具名查询，并让Hibernate把该结果集封送到对象图。

最后记住，尤其在遗留数据库中，存储过程和函数有时候不能在Hibernate中映射；在这种情况下，必须退回到简单的JDBC。有时候你可以用另一个包含Hibernate期待的参数接口的存储过程把遗留的存储过程包起来。一般的映射工具将涵盖许多变形和特殊案例。但是Hibernate的未来版本将改进映射能力——我们期待在不久的将来，会有更好的参数处理（没有更多的问号需要计算），并能支持任意的输入和输出参数。

你现在已经完成了运行时SQL查询和DML的定制。下面来转换一下视角，给数据库Schema的创建和修改定制SQL：DDL。

8.3 改进 Schema DDL

在Hibernate应用中定制DDL，通常是只有在利用Hibernate工具集生成数据库Schema时才要考虑的东西。如果已经有一个Schema，这样的定制将不会影响Hibernate的运行时行为。

可以把DDL导出到一个文本文件，或者每当运行集成测试时直接在数据库中运行它。因为DDL主要是特定于供应商的，你放到映射元数据中的每一个选项都有可能把元数据绑定到一个特定的数据库产品——当你应用以下特性时要记住这一点。

DDL定制可分成两类：

- 在映射元数据中显式地命名自动生成的数据库对象，例如表、列和约束，而不是依赖由Hibernate从Java类和属性名称中衍生的自动命名。我们已经在4.3.5节中讨论过引用和扩展命名的内建机制和选项。接下来我们看一下可以启用以美化生成的DDL脚本的其他选项。
- 在映射元数据中处理额外的数据库对象，例如索引、约束和存储过程。在本章前面，你用<database-object>元素把任意的CREATE和DROP语句添加到了XML映射文件。也可以利用常规类内部的额外映射元素和属性映射，启用索引和约束的创建。

8.3.1 定制 SQL 名称和数据类型

在代码清单8-1中，把属性和元素添加到Item类的映射中：

代码清单8-1 对hbm2ddl的Item映射中的额外元素

```
<class name="Item" table="ITEMS">

    <id name="id" type="string">
        <column name="ITEM_ID" sql-type="char(32)"/>   ❶
        <generator class="uuid"/>
    </id>

    <property name="initialPrice" type="big_decimal">
        <column name="INIT_PRICE"                      ❷
                not-null="true"
```

```
                precision="10"
                scale="2"/>
        </property>

        <property name="description" type="string"
                  column="ITM_DESCRIPTION" length="4000"/>

        <set name="categories" table="CATEGORY_ITEM" cascade="none">
            <key>
                <column name="ITEM_ID" sql-type="char(32)"/>
            </key>
            <many-to-many class="Category">
                <column name="CATEGORY_ID" sql-type="char(32)"/>
            </many-to-many>
        </set>

        ...

    </class>
```

❶ 如果属性（甚至标识符属性）是映射类型string，hbm2ddl导出器就会生成VARCHAR类型列。你知道标识符生成器uuid永远生成32个字符的字符串；因此你转换到CHAR SQL类型，并设置它的大小固定为32个字符。这个声明需要<column>元素，因为<id>元素中没有支持SQL数据类型的属性。

❷ 对于十进制的类型，你可以声明精确度和范围。这个例子用一种Oracle方言把列创建为INIT_PRICE number(10,2)；但是，对于不支持十进制精确度类型的数据库而言，就会（用HSQL）生成一个简单的INIT_PRICE numeric。

❸ 对于描述字段，是在<property>元素而不是嵌套的<column>元素中添加DDL属性。DESCRIPTION列被生成为VARCHAR(4000)——这是Oracle数据库中变量字符字段的一个限制（在Oracle中，在DDL中是VARCHAR2(4000)，但是方言负责这项工作）。

❹ <column>元素也可以用来在关联映射中声明外键字段。否则，关联表CATEGORY_ITEM的列就会是VARCHAR(32)而不是更精确的CHAR(32)类型。

注解中也可能有同样的定制，请见代码清单8-2。

代码清单8-2　用于定制DDL导出的额外注解

```
@Entity
@Table(name = "ITEMS")
public class Item {

    @Id
    @Column(name = "ITEM_ID", columnDefinition = "char(32)")
    @GeneratedValue(generator = "hibernate-uuid.hex")
    @org.hibernate.annotations.GenericGenerator(
            name = "hibernate-uuid.hex",
            strategy = "uuid.hex"
    )
    Private String id;

    @Column(name = "INIT_PRICE", nullable = false,
```

```
        precision = 10, scale = 2)
BigDecimal initialPrice;

@Column(name = "ITM_DESCRIPTION", length = 4000)
Private String description;

@ManyToMany
@JoinTable(
    name = "CATEGORY_ITEM",
    joinColumns =
        { @JoinColumn(name = "ITEM_ID",
                      columnDefinition = "char(32)")
        },
    inverseJoinColumns =
              { @JoinColumn(name = "CATEGORY_ID",
                            columnDefinition = "char(32)")
              }
      )
      Private Set<Category> categories = new HashSet<Category>();

      ...
  }
```

必须使用一个Hibernate扩展来命名非标准的标识符生成器。生成的SQL DDL的所有其他定制都通过JPA规范的注解完成。有一个属性值得特别关注：columnDefinition与Hibernate映射文件中的sql-type不同。它更加灵活：JPA持久化提供程序在CREATE TABLE语句中的列名称后面添加整个字符串，就像在ITEM_ID char(32)中一样。

名称和数据类型的定制是绝对应该考虑的。我们建议你始终利用适当的完整性规则改进数据库Schema的质量（并最终改进被存储数据的质量）。

8.3.2 确保数据一致性

完整性规则是数据库Schema的重要组成部分。数据库最重要的责任是保护信息，并保证它永远不会处于不一致的状态。这称作一致性，它是一般被应用到事务的数据库管理系统的ACID标准的一部分。

规则是业务逻辑的一部分，因此你通常在应用程序代码和数据库中混合实现业务相关的规则。这样编写应用程序是为了避免违背任何数据库规则。但是，数据库管理系统的任务是从不允许任何错误（从业务逻辑的意义来说）信息被永久保存——例如，如果其中一个访问数据库的应用程序有bug。只在应用程序代码中确保完整性的系统倾向于数据损坏，并经常随着时间的推移而降低数据库的质量。记住，大多数业务应用程序的主要目的从长远来说就是生成有价值的业务数据。

与在过程（或者面向对象的）应用程序代码中确保数据一致性相比，数据库管理系统则允许你把完整性规则声明式地实现为数据Schema的一部分。声明规则的好处包括减少了代码中的可能错误，以及数据库管理系统优化数据访问的机会。

我们来分辨一下四个级别的规则：

- 领域约束——领域是（不严格地说，在数据库范畴中）数据库中的一种数据类型。因此，

领域约束定义了一个特定的数据类型可以处理的可能值的范围。例如，int数据类型可以用于整数值。char数据类型可以保存字符串（例如，在ASCII中定义的所有字符）。因为我们主要使用创建在数据库系统中的数据类型，因此我们依赖由供应商定义的领域约束。如果创建用户自定义的数据类型（UDT），你将必须定义它们的约束。如果它们得到你的SQL数据库的支持，就可以使用对定制领域的（有限）支持，给特定的数据类型添加额外的约束。

- *列约束*——限制一个列保存特定领域的值，相当于增加一个列约束。例如，在DDL中声明INITIAL_PRICE列保存领域MONEY的值，它内部使用数据类型number(10,2)。你大多数时候直接使用数据类型，而不用事先定义领域。SQL数据库中一个特殊的列约束是NOT NULL。
- *表约束*——应用到单行或者几行的完整性规则是表约束。典型的声明表约束是UNIQUE（检查所有行中的重复值）。只影响单行的一个样例规则是“拍卖的终止日期应该晚于开始日期”。
- *数据库约束*——如果一项规则应用给不止一张表，它就具有数据库范围。你应该已经熟悉最常用的数据库约束：外键。这个规则保证了行之间引用的完整性，通常在独立的表中，但并非永远如此（自引用的外键约束并不罕见）。

大多数（如果不是全部）SQL数据库管理系统都支持上面提到的约束级别，以及每一种约束中最重要的选项。除了简单的关键字（例如NOT NULL和UNIQUE）之外，通常也可以声明带有应用了任意SQL表达式的CHECK约束的更复杂规则。但是，完整性约束却是SQL标准的软肋之一，并且来自不同供应商的解决方案有着明显的不同。

此外，对于拦截数据修改操作的数据库触发器来说，非声明约束和过程约束是可能的。然后，触发器可以直接实现约束过程，或者调用一个现有的存储过程。就像用于存储过程的DDL一样，你可以通过包含在生成的DDL中的<database-object>元素，把触发器声明添加到Hibernate映射元数据中。

最后，当数据修改语句被执行时，可以立即检查完整性约束，或者可以延期到事务结束时才进行检查。SQL数据库中的违背响应通常是拒绝，没有任何定制的可能。

现在要深入探讨一下完整性约束的实现。

8.3.3　添加领域约束和列约束

SQL标准包括领域，但是，它不仅相当受限制，而且经常不为DBMS所支持。如果你的系统支持SQL领域，就可以用它们把约束添加到数据类型：

```
create domain EMAILADDRESS as varchar
  constraint DOMAIN_EMAILADDRESS
  check ( IS_EMAILADDRESS(value) );
```

创建表的时候，你现在可以把这个领域标识符用作列类型：

```
create table USERS (
    ...
```

```
    USER_EMAIL EMAILADDRESS(255) not null,
    ...
);
```

SQL中领域（相对次要）的好处在于把一般的约束提取到一个单个位置。当插入或者修改数据时，领域约束始终立即得到检查。为了完成前一个例子，还必须编写存储函数IS_EMAILADDRESS（可以在网上找到许多正则表达式来完成这个）。在Hibernate映射中添加新领域与sql-type一样简单：

```
<property name="email" type="string">
    <column name="USER_EMAIL"
            length="255"
            not-null="true"
            sql-type="EMAILADDRESS"/>
</property>
```

利用注解声明自己的columnDefinition：

```
@Column(name = "USER_EMAIL", length = 255,
        columnDefinition = "EMAILADDRESS(255) not null")
String email;
```

如果想要利用剩下的Schema自动创建和删除领域声明，就把它放到<database-object>映射里面。

SQL支持额外的列约束。例如，业务规则允许用户注册名中只有字母和数字：

```
create table USERS (
    ...
    USERNAME varchar(16) not null
        check(regexp_like(USERNAME,'^[[:alpha:]]+$')),
    ...
);
```

可能在你的DBMS中不能使用这个表达式，除非它支持正则表达式。可以在Hibernate映射的<column>映射元素中声明单列检查约束：

```
<property name="username" type="string">
    <column name="USERNAME"
            length="16"
            not-null="true"
            check="regexp_like(USERNAME,'^[[:alpha:]]+$')"/>
</property>
```

在注解中检查约束只有在Hibernate扩展中才是可能的：

```
@Column(name = "USERNAME", length = 16,
        nullable = false, unique = true)
@org.hibernate.annotations.Check(
    constraints = "regexp_like(USERNAME,'^[[:alpha:]]+$')"
)
private String username;
```

注意，你有一种选择：创建和使用领域，或者添加单个列约束，作用都一样。长期而言，领域通常更易于维护，并且更可能避免重复。

我们来看看下一级规则：单行和多行表约束。

8.3.4　表级约束

想象你要保证CaveatEmptor拍卖不能在它开始之前被终止。你编写应用程序代码，防止用户把Item中的startDate和endDate属性设置为错误的值。可以在用户界面或者属性的设置方法中进行。在数据库Schema中，添加单个行的表约束：

```
create table ITEM (
    ...
    START_DATE timestamp not null,
    END_DATE timestamp not null,
    ...
    check (START_DATE < END_DATE)
);
```

表约束被添加在CREATE TABLE DDL中，并且可以包含任意的SQL表达式。你把约束表达式包括在Hibernate映射文件中的<class>映射元素上：

```
<class name="Item"
       table="ITEM"
       check="START_DATE &lt; END_DATE">
```

注意，< 这个字符必须用XML的<转义。利用注解，你需要添加一个Hibernate扩展注解来声明检查约束：

```
@Entity
@org.hibernate.annotations.Check(
    constraints = "START_DATE < END_DATE"
)
public class Item { ... }
```

多行的表约束可以通过更复杂的表达式实现。表达式中可能需要一个子查询来完成这个，你的DBMS可能不支持它——先查阅产品文档。但是有一些一般的多行表约束，可以在Hibernate映射中直接作为属性添加。例如，把User的注册名称标识为系统中唯一的：

```
<property name="username" type="string">
    <column name="USERNAME"
            length="16"
            not-null="true"
            check="regexp_like(USERNAME,'^[[:alpha:]]+$')"
            unique="true"/>
</property>
```

唯一约束声明也可能在注解元数据中：

```
@Column(name = "USERNAME", length = 16, nullable = false,
        unique = true)
@org.hibernate.annotations.Check(
    constraints = "regexp_like(USERNAME,'^[[:alpha:]]+$')"
)
private String username;
```

当然，还可以在JPA XML描述符中进行（但是没有检查约束）：

```
<entity class="auction.model.User" access="FIELD">
    <attributes>
        ...
        <basic name="username">
            <column name="USERNAME"
                    length="16"
                    nullable="false"
                    unique="true"/>
        </basic>
    </attributes>
</entity>
```

被导出的DDL包括unique约束：

```
create table USERS (
    ...
    USERNAME varchar(16) not null unique
        check(regexp_like(USERNAME,'^[[:alpha:]]+$')),
    ...
);
```

唯一约束也可以跨几个列。例如，CaveatEmptor支持嵌套的Category对象树。其中一个业务规则说，一个特定的分类不能与它的任何同胞同名。因此，你需要一个保证这种唯一性的多列多行约束：

```
<class name="Category" table="CATEGORY">
    ...
    <property name="name">
      <column name="CAT_NAME"
              unique-key="unique_siblings"/>
    </property>

    <many-to-one name="parent" class="Category">
      <column name="PARENT_CATEGORY_ID"
              unique-key="unique_siblings"/>
    </many-to-one>
    ...
</class>
```

使用unique-key属性把一个标识符分配给约束，以便可以在一个类映射中多次引用它，并把相同的约束分组成列。然而，在DDL中不用标识符命名约束：

```
create table CATEGORY (
    ...
    CAT_NAME varchar(255) not null,
    PARENT_CATEGORY_ID integer,
    ...
    unique (CAT_NAME, PARENT_CATEGORY_ID)
);
```

如果想要利用注解创建一个跨几列的唯一约束，就需要在实体而不是单个列中声明它：

```
@Entity
@Table(name = "CATEGORY",
   uniqueConstraints = {
        @UniqueConstraint(columnNames =
```

```
                {"CAT_NAME", "PARENT_CATEGORY_ID"} )
    }
)
public class Category { ... }
```

利用JPA XML描述符，多列约束如下：

```
<entity class="Category" access="FIELD">
    <table name="CATEGORY">
        <unique-constraint>
            <column-name>CAT_NAME</column-name>
            <column-name>PARENT_CATEGORY_ID</column-name>
        </unique-constraint>
    </table>
...
```

完全定制约束，包括一个用于数据库目录的标识符，可以通过<database-object>元素添加到DDL：

```
<database-object>
    <create>
        alter table CATEGORY add constraint UNIQUE_SIBLINGS
            unique (CAT_NAME, PARENT_CATEGORY_ID);
    </create>
    <drop>
        drop constraint UNIQUE_SIBLINGS
    </drop>
</database-object>
```

这项功能在注解中不可用。注意，可以利用你所有的定制数据库DDL对象，在基于注解的应用程序中添加Hibernate XML元数据文件。

最后，约束的最后一个目录包括跨几张表的适用于整个数据库的规则。

8.3.5　数据库约束

可以通过联结在任何check表达式的子查询中创建跨几张表的规则。不是只引用约束在其中被声明的表，你还可以查询不同的表（通常查询特定部分的信息存不存在）。

创建适用于整个数据库的约束的另一种技术是使用定制触发器，它们在特定表中行的插入或者更新中运行。这是一种过程方法，具有已经提到的缺点，但是天生灵活。

目前为止，大部分跨几张表的常用规则都是参照完整性（referential integrity）规则。它们是广为人知的外键，是两个东西的组合：一个从相关的行中复制来的键值，和一个保证引用值存在的约束。Hibernate自动地给关联映射中的所有外键列创建外键约束。如果你查看由Hibernate生成的DDL，可能注意到这些约束也已经自动生成数据库标识符——不容易阅读并且使调试更困难的一些名称：

```
alter table ITEM add constraint FK1FF7F1F09FA3CB90
    foreign key (SELLER_ID) references USERS;
```

这个DDL给ITEM表中的SELLER_ID列声明了外键约束。它引用USERS表的主键列。可以利用foreign-key属性在Item类的<many-to-one>映射中定制约束的名称：

```
<many-to-one name="seller"
             class="User"
             column="SELLER_ID"
             foreign-key="FK_SELLER_ID"/>
```

利用注解，使用Hibernate扩展：

```
@ManyToOne
@JoinColumn(name = "SELLER_ID")
@org.hibernate.annotations.ForeignKey(name = "FK_SELLER_ID")
private User seller;
```

给多对多关联创建的外键需要一种特殊的语法：

```
@ManyToMany
@JoinTable(...)
@org.hibernate.annotations.ForeignKey(
    name = "FK_CATEGORY_ID",
    inverseName = "FK_ITEM_ID"
)
private Set<Category> categories...
```

如果想要自动生成与人类数据库管理员编写的没有区别的DDL，就在映射元数据中定制你所有的外键约束。这不仅是个好实践，而且当你必须阅读异常消息时，它的帮助十分明显。注意，hbm2ddl导出器只对已经在一个双向关联映射的非反向端设置的外键考虑约束名称。

外键约束在SQL中也有遗留Schema可能已经使用过的特性。不要立即拒绝可能违背外键约束的数据修改，SQL数据库可以把这个变化CASCADE到引用行。例如，如果父行被删除了，父行主键上包含外键约束的所有子行也可能被删除。如果必须或者想要使用这些数据库级的级联选项，就在外键映射中启用它们：

```
<class name="Item" table="ITEM">
    ...
    <set name="bids" cascade="save-update, delete">
        <key column="ITEM_ID" on-delete="cascade"/>
        <one-to-many class="Bid"/>
    </set>

</class>
```

Hibernate现在创建和依赖外键约束的数据库级ON CASCADE DELETE选项，而不是在Item实例被删除且所有出价必须被移除时，执行多个单独的DELETE语句。注意这个特性绕过了Hibernate的为了版本化数据而常用的乐观锁策略！

最后，与从业务逻辑中转变来的完整性规则不相关，数据库性能优化也成为典型的DDL定制工作的一部分。

8.3.6 创建索引

索引是优化数据库应用程序的性能时的一项关键特性。数据库管理系统中的查询优化器可以利用索引避免过度扫描数据表。因为它们只在数据库的物理实现中相关，索引不是SQL标准的一部分，并且DDL和可用的索引选项是专门用于特定产品的。然而，典型索引的最常用的DDL可以被嵌入在一个Hibernate映射中（也就是说，不用一般的<database-object>元素）。

CaveatEmptor中的许多查询都将可能涉及拍卖Item的endDate属性。可以通过对这个属性的列创建索引来加快这些查询：

```
<property    name="endDate"
             column="END_DATE"
type="timestamp"
index="IDX_END_DATE"/>
```

自动生成的DDL现在包括一条额外的语句：

```
create index IDX_END_DATE on ITEM (END_DATE);
```

注解也有相同的功能，作为Hibernate扩展：

```
@Column(name = "END_DATE", nullable = false, updatable = false)
@org.hibernate.annotations.Index(name = "IDX_END_DATE")
private Date endDate;
```

可以通过在几个属性（或者列）映射中设置相同的标识符来创建一个多列的索引。任何其他的索引选项，例如UNIQUE INDEX（它创建一个额外的多行表级约束）、索引方法（一般为btree、hash和binary）和任何存储子句（例如，在单个的表空间中创建索引）可以通过<database-object>只在完整定制的DDL中设置。

包含注解的多列索引在实体级中定义，通过把额外的属性应用到表映射的定制Hibernate注解：

```
@Entity
@Table(name="ITEMS")
@org.hibernate.annotations.Table(
  appliesTo = "ITEMS",  indexes =
    @org.hibernate.annotations.Index(
     name = "IDX_INITIAL_PRICE",
     columnNames = { "INITIAL_PRICE", "INITIAL_PRICE_CURRENCY" }
    )
)
public class Item { ... }
```

注意，@org.hibernate.annotations.Table不是@javax.persistence.Table的替代物，因此如果需要覆盖默认的表名称，仍然需要常规的@Table。

如果想要学习不同的数据库优化技术，尤其是索引如何让你更接近查询的最佳执行计划，我们建议你买本好书：Dan Tow所著的*SQL Tunning*（Tow，2003）。

本章已经多次介绍的一种映射是<database-object>。它有一些我们还没有讨论到的其他选项。

8.3.7　添加辅助的 DDL

Hibernate自动地给表和约束创建基础的DDL；如果你有特定的标识符生成器，它甚至可以创建序列。但是，存在一些Hibernate无法自动创建的DDL。这包括各种非常特定于供应商的性能选项和任何其他只与数据的物理存储（例如表空间）相关的DDL。

这种DDL没有映射元素或者注解的一个原因在于有太多的变形和可能性——没有人能够或

者想要维护超过25种数据库方言的DDL的所有可能组合。第二个更为重要的原因在于，你应该永远让数据库管理员来完成数据库Schema。例如，你知道外键列上的索引会在某些情况下损害性能，因此不用Hibernate自动生成吗？我们建议数据库管理员早点介入，并验证从Hibernate自动生成的DDL。

如果正从一个新应用程序和新数据库开始，一般的过程是在开发期间由Hibernate自动生成DDL；数据库性能关注点在这方面不应该并且通常也不扮演重要的角色。同时（或者迟一点，在测试期间），专业的数据库管理员验证并优化SQL DDL，并创建最终的数据库Schema。可以把DDL导出到文本文件，并把它交给数据库管理员。

或者可以把定制的DDL语句添加到映射元数据中（这种方法你可能已经见过多次）：

```
<database-object>
    <create>
        [CREATE statement]
    </create>
    <drop>
        [DROP statement]
    </drop>

    <dialect-scope name="org.hibernate.dialect.Oracle9Dialect"/>
    <dialect-scope name="org.hibernate.dialect.OracleDialect"/>

</database-object>
```

<dialect-scope>元素把定制的CREATE或者DROP语句限制为一个特定的被配置数据库方言组，如果你正在几个系统中部署并且需要不同的定制时，它很有用。

如果需要更多对于生成的DDL可编程式的控制，就实现AuxiliaryDatabaseObject接口。Hibernate包含一个可以子类化的便利实现；然后你可以有选择地覆盖方法：

```
package auction.persistence;

import org.hibernate.mapping.*;
import org.hibernate.dialect.Dialect;
import org.hibernate.engine.Mapping;

public class CustomDDLExtension
    extends AbstractAuxiliaryDatabaseObject {

    public CustomDDLExtension() {
        addDialectScope("org.hibernate.dialect.Oracle9Dialect");
    }

    public String sqlCreateString(Dialect dialect,
                                  Mapping mapping,
                                  String defaultCatalog,
                                  String defaultSchema) {

        return "[CREATE statement]";
    }

    public String sqlDropString(Dialect dialect,
                                String defaultCatalog,
                                String defaultSchema) {
```

```
            return "[DROP statement]";
        }
    }
```

可以通过编程来添加方言范围，甚至访问sqlCreateString()和sqlDropString()方法中的一些映射信息。这给你提供了许多关于如何创建和编写DDL语句的灵活性。必须在映射元数据中启用这个定制类：

```
<database-object>
    <definition class="auction.persistence.CustomDDLExtension"/>
    <dialect-scope name="org.hibernate.dialect.OracleDialect"/>
</database-object>
```

额外的方言范围是累积的；前面的例子全都应用两种方言。

8.4　小结

本章探讨了当你必须处理遗留数据库Schema时可能遇到的问题。自然键、复合键和外键经常不方便，需要格外小心地映射。Hibernate也提供公式以及映射文件中的一些SQL表达式，它们可以帮助你处理无法改变的遗留Schema。

通常，你也依赖Hibernate在应用程序中为所有创建、读取、更新和删除操作而自动生成的SQL。在本章中，你已经学习了如何用自己的语句定制这个SQL，以及如何把Hibernate与存储过程和存储函数整合。

最后一节探讨了数据库Schema的生成，以及如何定制和扩展映射为包括各种约束、索引和数据库管理员可能建议的任意DDL。

表8-1展现了可以用来比较原生的Hibernate特性和Java Persistence的一个概括。

表8-1　第8章中Hibernate和JPA对照表

Hibernate Core	Java Persistence和EJB 3.0
Hibernate支持任何种类的自然的和复合的主键，包括自然键的外键、复合主键和复合主键中的外键	为自然键和组合键提供标准支持，相当于Hibernate
Hibernate通过公式映射和属性引用支持任意的关联联结条件	在编写本书之时，还没有标准或者注解支持被分组的属性引用
Hibernate对特定的实体类支持二级表的基本联结	对二级表和基本联结提供标准支持
Hibernate支持触发器整合和生成的属性设置	Hibernate Annotations支持生成的属性和触发器整合
Hibernate让你利用XML映射元数据中的选项定制所有的SQL DML语句	在编写本书之时，还不支持利用注解的SQL DML定制
Hibernate让你给自动的模式生成定制SQL DDL。任意的SQL DDL语句可以被包括在XML映射元数据中	JPA标准化了基本的DDL声明，但是并非XML映射元数据的所有特性都能通过注解得到支持

你现在已经学习了关于把类映射到Schema的一切内容(我们尽可能详细地介绍了在一本书中所能够介绍的内容)。本书的下一部分，将讨论如何使用持久化管理器API来加载和保存对象，事务和会话如何实现，以及如何编写查询。

Part 3

第三部分

会话对象处理

本书的这部分将介绍如何使用持久化对象。第 9 章介绍如何利用 Hibernate 和 Java Persistence 编程接口加载和保存对象。事务和并发性控制是另一个重要的话题，在第 10 章详细讨论。然后在第 11 章实现会话，介绍这个概念如何改进系统设计。第 12 章和第 13 章则关注效率，以及在必须加载和修改大型且复杂的数据集时，Hibernate 特性如何使你变得更加轻松。查询、查询语言和 API 在第 14 章和第 15 章将得到深入的实践。在第 16 章中，我们通过设计和测试一个包含 ORM 持久化的分层应用程序，把它们集中到一起。

在读完这部分之后，你将学会如何使用 Hibernate 和 Java Persistence 编程接口，以及如何有效地加载、修改和保存对象。你将理解事务是如何工作的，以及为什么会话处理可以给应用程序设计打开新路子。你将能够优化任何对象修改场景、编写复杂的查询，并应用最佳的抓取和高速缓存策略来改善性能和增强可伸缩性。

本部分内容

- 第 9 章　使用对象
- 第 10 章　事务和并发
- 第 11 章　实现对话
- 第 12 章　有效修改对象
- 第 13 章　优化抓取和高速缓存
- 第 14 章　利用 HQL 和 JPA QL 查询
- 第 15 章　高级查询选项
- 第 16 章　创建和测试分层的应用程序
- 第 17 章　JBoss Seam 简介

第9章

使用对象

本章内容

- 对象的生命周期和状态
- 使用Hibernate API
- 使用JPA

你现在对于Hibernate和ORM如何解决对象/关系不匹配的静态方面有了一定的了解。利用你目前所知的东西，有可能解决结构不匹配的问题，但是这个问题的有效解决方案还需要更多的东西。你必须研究运行时数据访问的策略，因为它们对于应用程序的性能是至关重要的。你基本上已经学会了如何控制对象的状态。

这一章和后面的几章涵盖了对象/关系不匹配的行为方面。我们认为这些问题至少与前面章节中讨论过的结构问题一样重要。依据我们的经验，许多开发人员只是真正知道结构不匹配，但很少关注这种不匹配的更动态的行为方面。

本章讨论对象的生命周期——对象如何变成持久化，以及它如何不再被认为是持久化的——以及触发这些转变的方法调用和其他动作。Hibernate持久化管理器`Session`，负责管理对象状态，因此我们讨论如何使用这个重要的API。EJB 3.0中主要的Java Persistence接口被称作`EntityManager`，并且由于它与Hibernate API非常类似，将很容易一起学习。当然，如果你没有使用Java Persistence或者EJB 3.0，可以很快地跳过这部分资料——我们鼓励你这两种选项都读，然后决定哪种选项对你的应用程序更好。

我们从持久化对象、它们的生命周期以及触发持久化状态改变的事件开始。虽然有些资料可能过于正式，但是对持久化生命周期（persistence lifecycle）的扎实理解是必不可少的。

9.1 持久化生命周期

由于Hibernate是一种透明的持久化机制——类不知道它们自己的持久化能力——有可能编写这样的应用程序逻辑：它不知道自己所操作的对象是表示持久化状态，还是表示只存在于内存中的临时状态。应用程序没必要在意在调用对象的方法时它是否为持久化。例如，你可以在`Item`类的一个实例中调用`calculateTotalPrice()`业务方法，根本不必考虑持久化（例如，在单元

测试中）。

任何包含持久化状态的应用程序都必须与持久化服务交互，每当它需要把保存在内存中的状态传播到数据库的时候（反之亦然）。换句话说，你必须调用Hibernate（或者Java Persistence）接口来保存和加载对象。

当以这种方式与持久化机制进行交互时，应用程序必须让自己知道有关持久化的对象状态和生命周期。我们这里所谓的*持久化生命周期*指的是对象在它的生命期间经历的状态。我们也使用术语*工作单元*（unit of work），指的是把一组操作当作一个（通常是原子的）组。另一个难题是由持久化服务提供的*持久化上下文*（persistence context）。把持久化上下文当作高速缓存，它记住你在一个特定的工作单元中给对象所做的所有修改和状态改变（这有点简化了，但这是一个好的起点）。

现在来剖析一下所有这些术语：对象和实体状态、持久化上下文和托管范围。你可能更习惯于考虑使内容从数据库进出（经由JDBC和SQL）必须管理哪些*语句*。然而，成功地使用Hibernate（和Java Persistence）的关键因素之一在于对状态管理的理解，因此你要学习本节内容。

9.1.1 对象状态

不同的ORM解决方案使用不同的术语，并给持久化生命周期定义不同的状态和状态转变。甚至，内部使用的对象状态可能与那些公开给客户端应用程序的对象状态不同。Hibernate只定义4种状态，从客户端代码中隐藏了它内部实现的复杂性。

Hibernate定义的对象状态和它们在状态图中的转变如图9-1所示。你还可以看到对触发转变的持久化管理器API的方法调用。Hibernate中的这个API是`Session`。我们在本章中讨论这幅图，每当你需要整体概况时就可以参考它。

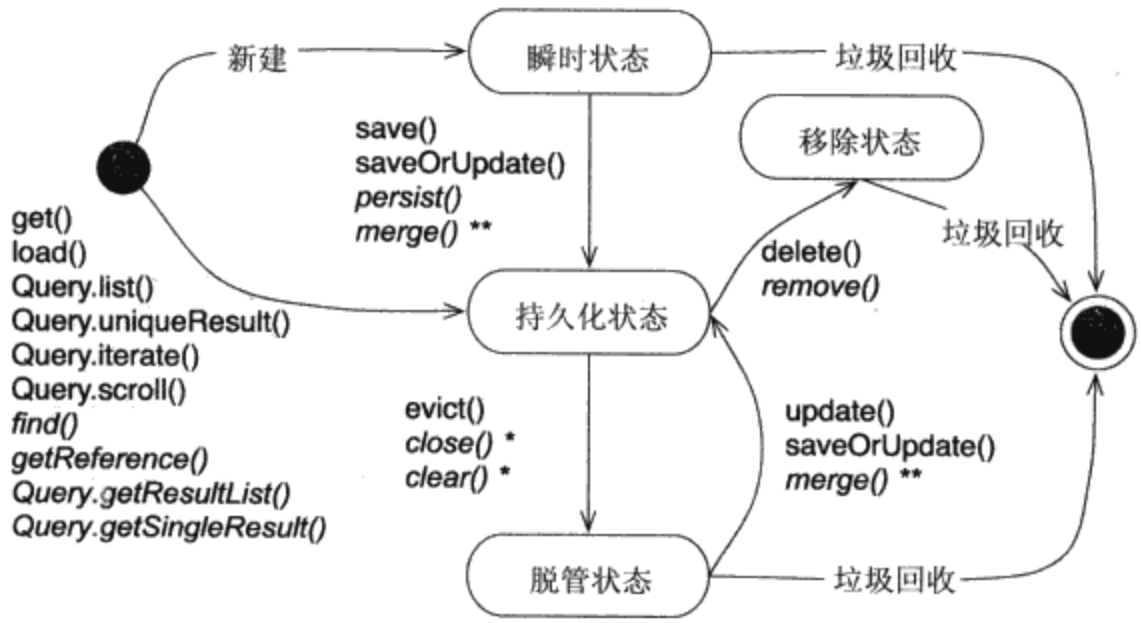


图9-1 触发持久化管理器操作时的对象状态及其转变

我们也在图9-1中包括了Java Persistence实体实例的状态。如你所见，它们几乎与Hibernate的

一样，Session的大部分方法也类似于EntityManager API（如斜体内容所示）。假设Hibernate是Java Persistence中标准化的子集提供的功能的一个超集。

有些方法在两种API中都有；例如，Session有一个persist()操作，与EntityManager的对应物有着相同的语义。其他的，如load()和getReference()，也有着相同的语义，只是方法名称不同。

在它的生命期间，对象可以从瞬时对象转变为持久化对象再到脱管对象。让我们更深入地探讨状态和转变。

1．瞬时对象

利用new操作符实例化的对象并不立即就是持久化的。它们的状态是瞬时的（transient），意味着它们不与任何数据库表行相关联，因此一旦不再被其他的对象引用时，它们的状态立即丢失。这些对象有寿命，它在那个时候有效地终止，并且变成不可访问，等待垃圾回收。Java Persistence不包括这种状态的术语；刚刚被实例化的实体对象是新的。我们将继续把它们当作是瞬时的，以强调这些实例被持久化服务变成托管的可能。

Hibernate和Java Persistence认为所有的瞬时实例都要变成非事务的；持久化上下文不知道瞬时实例的任何修改。这意味着Hibernate不给瞬时对象提供任何回滚功能。

只被其他瞬时实例引用的对象也默认为瞬时。对于从瞬时转变为持久化状态的实例，要变成托管，需要调用一个持久化管理器，或者从已经持久化的实例中创建一个引用。

2．持久化对象

持久化实例是一个包含数据库同一性的实体实例，如在4.2节中定义的一样。这意味着持久化且被托管的实例具有设置成为其数据库标识符的主键值。（当这个标识符被分配到持久化实例的时候，有一些变化形式。）

持久化实例可能是被应用程序实例化的对象，然后通过在持久化管理器上调用其中一种方法变成持久化。它们甚至可能是当从另一个已经托管的持久化对象中创建引用时变成持久化的对象。或者，持久化实例也可能是通过执行查询、标识符查找、或者开始从另一个持久化实例导航对象图，从数据库中获取的一个实例。

持久化实例始终与持久化上下文（persistance context）关联。Hibernate高速缓存它们，并且可以侦测到它们是否已经被应用程序修改。

关于这种状态以及实例在持久化上下文中如何得到托管，还有更多的内容要讨论。本章稍后会回到这个话题。

3．移除对象

可以通过几种方式删除实体实例：例如，可以用持久化管理器的一个显式操作把它移除。如果移除所有对它的引用，它可能也变成可以删除的了，这项特性只有在包含Hibernate扩展设置（实体的孤儿删除）的Hibernate或者Java Persistence中才可用。

如果一个对象已经被计划在一个工作单元结束时删除，它就是处于移除状态，但仍然由持久化上下文托管，直到工作单元完成。换句话说，移除对象不应该被重用，因为一旦工作单元完成，它就将立即从数据库中被删除。你也应该放弃在应用程序中保存着的任何对它的引用（当然，是

在你用完它之后——例如，在你已经呈现了用户看到的删除确认屏幕之后）。

4．脱管对象

要理解脱管（detached）对象，你需要考虑实例的一种典型的转变：它先是瞬时的，因为它刚刚在应用程序中创建。现在通过在持久化管理器中调用一个操作使它变成持久化。所有这些都发生在单个工作单元中，并且这个工作单元的持久化上下文在某个时间点（当产生一个SQL的INSERT时）与数据库同步。

现在工作单元完成了，持久化上下文也关闭了。但是应用程序仍然有一个句柄（handle）：对被保存实例的一个引用。只要持久化上下文是活动的，这个实例的状态就是持久化的。在工作单元结束时，持久化上下文关闭。你现在保存着对其引用的是什么对象状态，以及如何处理呢？

我们把这些对象当作脱管（detached），表示它们的状态不再保证与数据库状态同步，不再被附加到持久化上下文中，并仍然包含持久化数据（可能很快会失效）。可以继续使用脱管对象并修改它。但有时候你或许想要使那些变化变成持久化——换句话说，把脱管实例变回到持久化状态。

Hibernate提供重附（reattachment）和合并（merging）两种操作来处理这种情况。Java Persistence只对合并标准化。这些特性对于如何设计多层应用程序有着深刻的影响。从一个持久化上下文返回对象到表现层，并且随后在一个新的持久化上下文中重用它们的能力是Hibernate和Java Persistence的主要卖点。它让你能够创建跨越用户思考时间的长工作单元。我们称这种长期运行的工作单元为对话（conversation）。我们会很快回到脱管对象和对话的话题。

现在你对于对象状态以及如何发生转变应该有了初步的理解。我们的下一个主题是持久化上下文及其提供的对象管理。

9.1.2 持久化上下文

你可能认为持久化上下文会成为托管实体实例的一个高速缓存。持久化上下文不是你在应用程序中所见到的东西；它不是一个可以调用的API。在Hibernate应用程序中，假设一个Session有一个内部的持久化上下文。在Java Persistence应用程序中，EntityManager具有持久化上下文。一个工作单元中所有处于持久化状态和托管状态的实体都被高速缓存在这个上下文中。本章稍后会探讨Session和EntityManager API。现在你要知道这个（内部的）持久化上下文的作用。

持久化上下文之所以有用，基于以下几个原因：

- Hibernate可以进行自动的脏检查和事务迟写。
- Hibernate可以用持久化上下文作为一级高速缓存。
- Hibernate可以保证Java对象同一性的范围。
- Hibernate可以把持久化上下文扩展到跨整个对话。

所有这些要点对Java Persistence提供程序也有效。我们来看看每一种特性。

1．自动脏检查

持久化实例托管在一个持久化上下文中——它们的状态在工作单元结束时与数据库同步。当一个工作单元结束时，保存在内存中的状态通过SQL INSERT、UPDATE和DELETE语句（DML）的

执行被传播到数据库。这个过程也可能发生在其他时间点。例如，Hibernate可能在查询执行之前与数据库同步。这样确保了查询知道在工作单元期间之前所做的改变。

Hibernate没有于工作单元结束时在内存中更新每一个单独的持久化对象的数据库行。ORM软件必须有一个策略，用来侦测哪个持久化对象已经被应用程序修改。我们称之为*自动脏检查*（automatic dirty checking）。一个修改过的对象还没被传播到数据库时被认为是脏的（dirty）。这种状态对于应用程序不可见。利用*透明的事务级迟写*（transparent transaction-level write-behind），Hibernate尽可能迟地把状态变化传播到数据库，但是从应用程序中隐藏这个细节。通过尽可能迟地执行DML（趋向于数据库事务的结束），Hibernate试图保证数据库中的锁时间尽可能短。（DML通常在数据库中创建一直被保存到事务结束的锁。）

Hibernate能够准确地侦测哪些属性已经被修改，以便有可能只包括需要在SQL UPDATE语句中更新的列。这可能带来一些性能上的收获。但是差别通常不明显，理论上来说，这样在某些环境下会损害性能。默认情况下，Hibernate包括SQL UPDATE语句中被映射的表的所有列（因而，Hibernate可以在启动时而不是运行时生成这个基础的SQL）。如果你想要只更新被修改的列，可以通过在类映射中设置dynamic-update="true"启用动态的SQL生成。对新记录的插入实现相同的机制，并且可以用dynamic-insert="true"启用INSERT语句的运行时生成。当一张表中有特别多的列（假设，超过50列）时，我们建议考虑这种设置；有时候，无变化的字段所引起的过载的网络流量也是不容忽视的。

少数情况下，你也可能想要给Hibernate提供自己的脏检查算法。Hibernate默认把一个对象的旧快照与同步时的快照进行比较，侦测任何需要更新数据库状态的修改。可以通过org.hibernate.Interceptor给Session提供一个定制的findDirty()方法来实现自己的子程序。本书稍后将介绍拦截器（interceptor）的实现。

本章后面会回到同步过程（即清除，flushing）。

2．持久化上下文高速缓存

持久化上下文是持久化实体实例的一个高速缓存。这意味着它记住了你已经在特定的工作单元中处理过的所有持久化实体实例。自动脏检查是这个高速缓存的好处之一。另一个好处是对实体的*可重复读取*（repeatable read），以及工作范围高速缓存单元的性能优势。

例如，如果Hibernate被告知通过主键（标识符的一个查找）加载对象，它就可以先在当前的工作单元的持久化上下文中检查。如果在那里找到了实体，就不会发生数据库命中——这是对应用程序的可重复读取。如果查询通过其中一个Hibernate（或者Java Persistence）接口执行也一样。Hibernate读取查询的结果集，并封送随后返回给应用程序的实体对象。在这个过程期间，Hibernate与当前的持久化上下文交互。它试图解析这个高速缓存中的每一个实体实例（通过标识符）；只有在当前的持久化上下文中无法找到该实体时，Hibernate才会从结果集中读取剩下的数据。

持久化上下文高速缓存带来重大的性能好处，并改进工作单元中的孤立保证（免费得到了实体实例的可重复读取）。由于这个高速缓存只有工作单元的范围，它没有真正的缺点，例如对并发访问的锁管理——工作单元是在单个线程中处理的。

持久化上下文高速缓存有时候帮助避免不必要的数据库流量；但更重要的是，它确保了：

- 持久层在对象图中出现循环引用时，不会受到堆栈溢出（stack overflow）的影响。
- 工作单元结束时永远不能有相同数据库行的冲突表示法。在持久化上下文中，最多一个对象表示任何一个数据库行。对该对象进行的所有变化都可以被安全地写到数据库中。
- 同样地，在特定持久化上下文中进行的改变，也始终立即对在持久化上下文和它的工作单元内部（对实体保证的可重复读取）执行的所有其他代码可见。

不必做任何特别的事情来启用持久化上下文高速缓存。它始终开着，基于上述原因，它不可能被关闭。

本章稍后将介绍对象如何被添加到这个高速缓存（基本上是每当它们变成持久化的时候），以及如何管理这个高速缓存（通过手工从持久化上下文中脱管对象，或者通过清除持久化上下文）。

我们给持久化上下文所列的最后两项好处：受保护的同一性范围和扩展持久化上下文到跨对话的可能性，是密切相关的概念。要理解它们，就要退回一步，并从不同的角度考虑处于脱管状态的对象。

9.2 对象同一性和等同性

一个基础的Hibernate客户端/服务器端应用程序可能通过跨单个客户端请求的服务器端工作单元进行设计。当来自应用程序用户的一个请求需要数据访问时，就启动一个新的工作单元。这个工作单元在处理结束时终止，并为客户准备好了响应。这也被称作每次请求一个会话（session-per-request）策略（每当读到类似这样的东西时，可以用持久化上下文代替会话，但有点拗口）。

我们已经提到过，Hibernate可以支持一个可能长期运行的工作单元（称作对话，conversation）的实现。我们在接下来的几节中介绍对话的概念，以及对象同一性的基础和对象什么时候被认为是等同的——这些概念可能影响你如何思考和设计对话。

对话概念为什么会有用？

9.2.1 引入对话

例如，在Web应用程序中，通常不维护跨用户交互的数据库事务。用户花了很长的时间思考修改，但是由于可伸缩性的原因，必须保持数据库事务简短，并尽快释放数据库资源。每当需要引导用户通过几个屏幕完成一个工作单元（从用户的角度）时，可能面对这个问题——例如，填写一张在线的表格。在这个一般的场景中，拥有持久化服务的支持非常有用，因此可以用最少的代码和最佳的可伸缩性实现这样一个对话。

有两种策略可以在Hibernate或者Java Persistence应用程序中实现对话：利用脱管对象，或者通过扩展一个持久化上下文。这两者均各有利弊。

脱管对象状态和已经提到过的重附或者合并的特性是实现对话的方法。用户在思考时间期间对象以脱管状态保存，并且这些对象的任何修改都通过重附或者合并被手工变成持久化。这一策略也被称作利用脱管对象每次请求一个会话（session-per-request-with-detached-object）。可以在图

9-2中看到这种会话模式的图示。

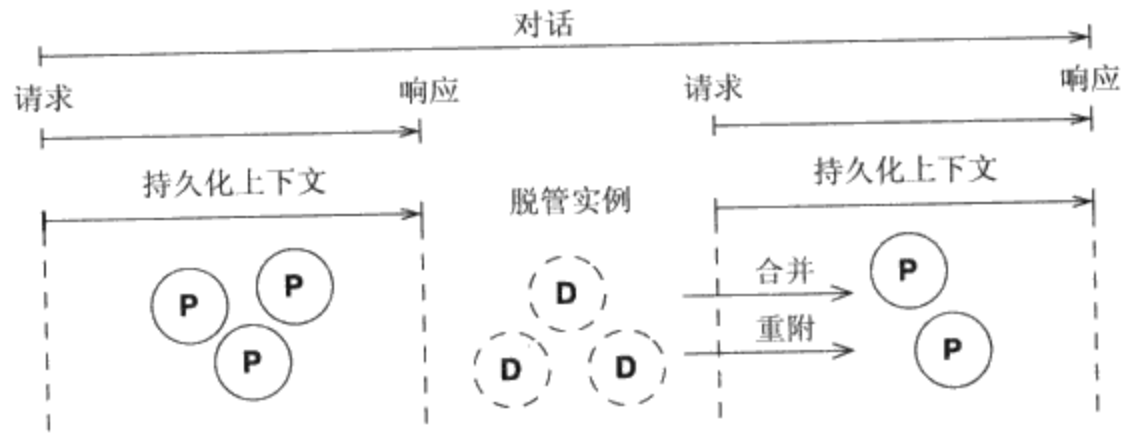


图9-2　利用脱管对象状态的对话实现

持久化上下文只跨一个特定请求的处理，对话期间应用程序手工重附和合并（且有时候脱管）实体实例。

另一种方法不需要手工重附或者合并：利用每次对话一个会话（session-per-conversation）模式，把一个持久化上下文扩展到跨整个工作单元（请见图9-3）

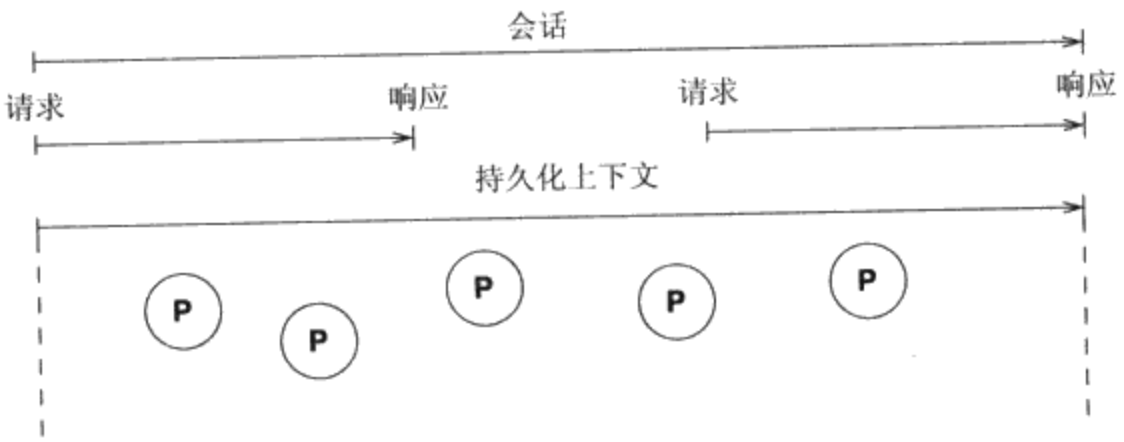


图9-3　利用被扩展持久化上下文的对话实现

首先我们来更深入地看一下脱管对象，以及在利用这种策略实现对话时会遇到的同一性问题。

9.2.2　对象同一性的范围

作为应用程序开发人员，我们用Java对象同一性（a==b）辨别对象。如果对象改变了状态，Java同一性保证与新状态一样吗？在分层应用程序中，可能不会。

为了探讨这个问题，理解Java同一性a==b和数据库同一性x.getId().euqals(y.getId())之间的关系非常重要。有时候它们是相等的；有时候不等。我们把Java同一性等价于数据库同一性的条件称作对象同一性的范围（scope of object identity）。

对于这一范围，有3种常见的选择：

- 没有同一性范围的基本持久层不保证一个行是否被访问两次，以及是否会把相同的对象实例返回给应用程序。如果应用程序修改了都在单个工作单元中表示同一行的两个不同的实例，就会有问题。（我们应该如何决定哪种状态应该被传播到数据库呢？）

- 持久层利用持久化上下文范围的同一性（persistence context-scoped identity），保证在单个持久化上下文的范围中，只有一个对象实例表示一个特定的数据库行。这样避免了前一个问题，还允许在上下文级的一些高速缓存。
- 过程范围的同一性（process-scoped identity）更进一步，保证在整个过程（JVM）中只有一个对象实例表示该行。

对于典型的Web或者企业应用程序，持久化上下文范围的同一性是首选。过程范围的同一性，的确对跨多个工作单元的实例重用，在高速缓存利用和编程模型方面提供了一些潜在的优势。但是，在一个遍布多线程的应用程序中，始终在全局的同一性映射中同步对持久化对象的共享访问成本太高，难以支付。更简单并且更可伸缩些的办法是，让每个线程在每个持久化上下文中使用一组独特的持久化实例。

假设Hibernate实现了持久化上下文范围的同一性。因此，Hibernate天生最适合多用户应用程序中高并发的数据访问。然而，我们已经提到过一些在对象不与持久化上下文关联时会遇到的问题。让我们用一个示例来讨论这个话题。

Hibernate同一性范围是持久化上下文的范围。它在包含Hibernate API的代码中的工作原理：Java Persistence代码相当于`EntityManager`，而不是`Session`。虽然我们还没有介绍更多有关这些接口的内容，但是下面的例子很简单，理解我们在`Session`中调用的方法应该不成问题。

如果你请求两个对象在同一个`Session`中使用相同的数据库标识符，结果是对相同内存实例的两个引用。代码清单9-1用几个`get()`操作在两个`Session`中进行了示范。

代码清单9-1 Hibernate中受保护的对象同一性范围

```
Session session1 = sessionFactory.openSession();
Transaction tx1 = session1.beginTransaction();

// Load Item with identifier value "1234"
Object a = session1.get(Item.class, new Long(1234) );
Object b = session1.get(Item.class, new Long(1234) );

( a==b ) // True, persistent a and b are identical

tx1.commit();
session1.close();

// References a and b are now to an object in detached state

Session session2 = sessionFactory.openSession();
Transaction tx2 = session2.beginTransaction();

Object c = session2.get(Item.class, new Long(1234) );

( a==c ) // False, detached a and persistent c are not identical

tx2.commit();
session2.close();
```

对象引用a和b，不仅有着相同的数据库同一性，而且还有相同的Java同一性，因为它们是在相同的`Session`中获得的。它们引用那个工作单元的持久化上下文中所知道的同一个持久化实

例。然而，一旦出了这个范围，Hibernate就不保证Java同一性了，因此a和c不相等。当然，测试数据库同一性的a.getId().equals(c.getId())，将仍然返回true。

如果你使用处于脱管状态的对象，就是正在处理处于受保护的对象同一性范围之外的对象。

9.2.3　脱管对象的同一性

如果对象引用离开了受保护的同一性范围，那么就称它为脱管对象引用（reference to a detached object）。在代码清单9-1中，如果只考虑数据库同一性——它们的主键值，则所有三个对象引用a、b和c都相等。然而，它们不是相等的内存对象实例。如果你在脱管状态下把它们作为相等处理，就会产生问题。例如，考虑代码的下列扩展，在session2已经终止之后：

```
...
session2.close();

Set allObjects = new HashSet();
allObjects.add(a);
allObjects.add(b);
allObjects.add(c);
```

所有三个引用都已经被添加到了一个Set。所有都是对脱管对象的引用。现在，如果你查看集合的大小、元素的数量，你希望是什么样的结果呢？

首先你必须了解Java Set的约定：在这种集合中不允许有重复元素。重复通过Set进行侦测；每当添加一个对象时，它的equals()方法就会自动被调用。被添加的对象与已经处在集合中的所有其他元素进行核对。如果equals()对已经在集合中的任何对象返回true，就不发生添加。

如果你知道对象的equals()实现，就会知道可以在Set中期待的元素数量。所有Java类都默认继承java.lang.Object的equals()方法。这个实现使用了双等号（==）比较：它在Java堆中检查两个引用是否指向同一个内存实例。

你可能猜测集合中的元素数量为两个。毕竟，a和b引用了相同的内存实例；它们已经被加载在相同的持久化上下文中。引用c在第二个Session中获得；它在堆中引用不同的实例。你有对两个实例的三个引用。然而，你知道这个，仅仅因为你已经见过加载对象的代码。在一个真实的应用程序中，你不可能知道a和b被加载在同一个Session中，c却在另一个之中。

此外，你显然希望这个集合正好只有一个元素，因为a、b和c表示相同的数据库行。

每当使用脱管状态下的对象时，特别在给同一性进行测试时（通常在一个基于散列的集合中），就需要为持久化类提供你自己的equeals()和hashCode()方法的实现。

1．理解equals()和hashCode()

在介绍如何实现你自己的等同性子程序之前，我们必须提醒两个重点。第一，依据我们的经验，许多Java开发人员在使用Hibernate（或者Java Persistence）之前永远不必覆盖equals()和hashCode()方法。传统上，Java开发人员似乎不关心这样一种实现的复杂细节。在公开的Hibernate论坛上最长的讨论是关于这个等同性问题的，并且经常把“抱怨”对准Hibernate。你应该了解根本的问题：如果默认的约定没有提供想要的语义，每一种带有基于散列的集合的面向对象编程语言都需要一个定制的等同性子程序。Hibernate应用程序中的脱管对象状态让你面对这个问题，这

可能是第一次。

另一方面，你可能不一定要覆盖equals()和hashCode()。如果你从来不比较脱管实例——也就是说，如果从来不把脱管实例放进相同的Set中，Hibernate提供的这个同一性范围保证就足够了。你可能决定设计一个不使用脱管对象的应用程序。可以给对话实现应用一个被扩展的持久化上下文策略，并从应用程序中完全消除脱管状态。这个策略也把受保护对象同一性的范围扩展到了跨整个对话。(注意，你仍然需要这条规定，不比较在两个会话中获得的脱管实例!)

假设你想要使用脱管对象，并且必须用自己的子程序测试它们的等同性。可以通过几种方式实现equals()和hashCode()。记住，在覆盖euqals()时，始终也需要覆盖hashCode()，以便这两种方法保持一致。如果两个对象相等，它们就必须有相同的散列码（hashcode)。

一种更为聪明的方法是实现equals()，只比较数据库标识符属性（通常是个代理主键）值：

```
public class User {
    ...

    public boolean equals(Object other) {
        if (this==other) return true;
        if (id==null) return false;
        if ( !(other instanceof User) ) return false;
        final User that = (User) other;
        return this.id.equals( that.getId() );
    }

    public int hashCode() {
        return id==null ?
            System.identityHashCode(this) :
            id.hashCode();
    }

}
```

注意这个equals()方法如何为还没有分配数据库标识符值的瞬时实例（如果id==null)退回到Java同一性。这是有道理的，因为它们不可能等于脱管实例，这个脱管实例有一个标识符值。

不幸的是，这种解决方案有一个大问题：标识符值直到对象变成持久化时才由Hibernate分配。如果瞬时对象在保存之前被添加到一个Set，它的散列值就可能在它被Set包含时改变，与java.util.Set的约束相反。特别是，这个问题使得级联保存（本书后面会讨论到）对于集来说变得毫无用处。我们强烈反对这种解决方案（数据库标识符等同性)。

一种更好的方法是包括持久化类的所有持久化属性在equals()比较中，远离任何数据库标识符属性。这就是大多数人知道的equals()的含义；我们称它为按值（by value）等同性。

当我们说到所有属性时，并不想包括集合。集合状态与不同的表关联，因此把它包括在内似乎错了。更重要的是，你并不想强制获取整个对象图而只是用来执行equals()。对于User而言，这意味着你不应该把boughtItems集合包括在比较式中。你可以编写下面的实现：

```
public class User {
    ...

    public boolean equals(Object other) {
        if (this==other) return true;
```

```
        if ( !(other instanceof User) ) return false;
        final User that = (User) other;
        if ( !this.getUsername().equals( that.getUsername() ) )
            return false;
        if ( !this.getPassword().equals( that.getPassword() ) )
            return false;
        return true;
    }

    public int hashCode() {
        int result = 14;
        result = 29 * result + getUsername().hashCode();
        result = 29 * result + getPassword().hashCode();
        return result;
    }

}
```

然而，这种方法还是有两个问题。第一，如果修改了其中一个（例如，如果用户改变了密码），那么来自不同Session的实例便不再相等。第二，包含不同数据库同一性的实例（表示数据库表不同行的实例）可以被认为相等，除非属性的有些组合保证唯一（数据库列有一个唯一约束）。对于这个案例中的用户而言，有一个唯一的属性：username。

这样就把我们引到了等同性检查的首选（并且语义上正确的）实现。你需要一个业务键（business key）。

2. 利用业务键实现等同性

为了开始我们推荐的这个解决方案，要先理解业务键的概念。业务键是一种属性，或者一些属性的组合，它对于每个包含相同的数据库同一性的实例来说是唯一的。本质上，它就是你要使用的自然键，如果你没有正在使用代理主键来代替的话。业务键不同于自然键，它的永远不变并非绝对的必备条件——只要它很少改变，就足够了。

我们认为，本来每个实体类都应该有一些业务键，即使它包括了类的所有属性（对于一些不可变的类而言，这很合适）。业务键是用户作为唯一辨别一个特定记录的东西，而代理键则是应用程序和数据库使用的东西。

业务键等同性意味着equals()方法只比较构成业务键的属性。这是个完美的解决方案，它避免了前面描述过的所有问题。唯一的缺点是它需要特别记着先辨别正确的业务键。无论如何，这项工作都是必要的；如果你的数据库必须经由约束检查来确保数据完整性，辨别任何唯一键就很重要了。

对于User类而言，username是一个很好的备用业务键。它永远不为空，通过数据库约束而唯一，并且即使改变也很少：

```
public class User {
    ...

    public boolean equals(Object other) {
        if (this==other) return true;
        if ( !(other instanceof User) ) return false;
        final User that = (User) other;
```

```
        return this.username.equals( that.getUsername() );
    }

    public int hashCode() {
        return username.hashCode();
    }

}
```

对于一些其他的类，业务键可能更加复杂，由一个属性的组合组成。以下是一些提示，帮助你在类中辨别业务键：

- 考虑当应用程序的用户必须辨别一个对象（在现实世界中）时，他们会引用什么属性。如果对象显示在屏幕上，用户如何把一个对象和另一个对象区分开来？这可能就是你正在寻找的业务键。
- 每个不可变的属性都可能是一个好的备选业务键。如果可变属性很少更新，或者如果当它们更新时你可以控制局势的话，可变属性也可能是好的备选对象。
- 每个包含UNIQUE数据库约束的属性都是好的备选业务键。记住，业务键的精确度必须好到足以避免重叠。
- 任何基于日期或者时间的属性，例如记录的创建时间，通常都是业务键的一个好组件。然而，System.currentTimeMillis()的准确性则取决于虚拟机和操作系统。我们建议过的安全缓冲区是50毫秒，如果基于时间的属性是业务键的单个属性时，它可能还不够精确。
- 可以用数据库标识符作为业务键的一部分。这似乎与我们之前的陈述相矛盾，但是我们不讨论指定类的数据库标识符。可能可以使用被关联对象的数据库标识符。例如，Bid类的一个备选业务键是Item的标识符，它与出价金额一起产生。你在数据库Schema中甚至可以有一个表示这个复合业务键的唯一约束。可以使用被关联Item的标识符值，因为它在Bid的生命周期期间从不改变——Bid构造器需要设置一个已经持久化的Item。

如果遵循了我们的建议，给你所有的业务类找到一个好的业务键就不难了。如果遇到很难的情况，就试着不考虑用Hibernate来解决它——毕竟，这是个纯粹的面向对象的问题。注意，在一个子类上覆盖equals()并在比较中包括另一个属性，这种做法通常都是错误的。在这种情况下，满足等同性既是对称性（symmetric）又是传递性（transitive）的要求是一个小技巧；并且更重要的是，业务键可能不符合数据库中任何定义好的备选自然键（子类属性可能被映射到不同的表）。

你可能也已经注意到，equals()和hashCode()方法始终通过获取方法访问“其他”对象的属性。这非常重要，因为作为other传递的对象实例可能是一个代理对象，而不是保存有持久化状态的真正实例。为了初始化这个代理来获得属性值，你需要用一个获取方法访问它。这是Hibernate不完全透明的一点。但是，使用获取方法而不是直接的实例变量访问，这无论如何都是个好实践。

现在转换视角，给不需要脱管对象且不公开任何脱管对象同一性问题的对话考虑一种实现策略。如果在使用脱管对象时可能公开的同一性范围问题似乎不堪重负，那么第二种对话实现策略可能就是你要找的。Hibernate和Java Persistence通过一个被扩展的持久化上下文——每个对话一

个会话（session-per-conversation）策略——支持对话的实现。

9.2.4　扩展持久化上下文

一个特定的对话为所有交互重用相同的持久化上下文。对话期间的所有请求处理都由相同的持久化上下文管理。当来自用户的请求被处理之后，持久化上下文没有关闭。在用户思考时间内，它断开与数据库的连接，并保持这种状态。当用户在对话中继续时，持久化上下文被重新连接到数据库，就可以处理下一个请求了。对话结束时，持久化上下文与数据库同步并关闭。下一个对话从一个新的持久化上下文开始，并且不重用来自前一个对话的任何实例；这一模式是可重复的。

注意，这样消除了脱管对象状态！所有实例要么是瞬时的（不为持久化上下文所知），要么是持久化的（附加到一个特定的持久化上下文）。它也不需要手工重附或合并上下文之间的对象状态，这是这个策略的好处之一。（你在对话之间仍然可能有脱管对象，但是我们认为这是你应该努力避免的一种特殊情况。）

在Hibernate中，这个策略为对话的持续使用单个的`Session`。Java Persistence对被扩展的持久化上下文有着内建的支持，甚至可以为你自动保存请求之间断开连接的上下文（在一个有状态的EJB会话bean中）。

本书后面会回到对话的话题，并介绍有关这两种实现策略的所有细节。你不必现在选择一种合适的策略，但是应该注意这些策略对于对象状态和对象同一性的影响，并且应该理解每个案例中必需的事务。

我们现在来探讨持久化管理器API，以及如何使对象状态背后的理论实际上真的可行。

9.3　Hibernate 接口

任何透明的持久化工具都包括一个持久化管理器（persistence manager）API。这个持久化管理器通常为以下内容提供服务：

- 基础的CRUD（创建、获取、更新、删除）操作；
- 查询执行；
- 事务的控制；
- 持久化上下文的管理。

持久化管理器可能通过几个不同的接口而被公开。对于Hibernate而言，它们是`Session`、`Query`、`Criteria`和`Transaction`。这些接口的实现在幕后是紧密相连的。

在Java Persistence中，你交互的主要接口是`EntityManager`；它扮演着与Hibernate的`Session`一样的角色。其他的Java Persistence接口是`Query`和`EntityTransaction`（你可能会猜测它们在原生的Hibernate中的等价物是什么）。

现在介绍如何用Hibernate和Java Persistence加载和保存对象。有时候这两者有着完全相同的语义和API，甚至方法名称也相同。因此更为重要的是注意一些微小的差别。为了使本书的这一部分更易于理解，我们决定使用一种与平时不一样的策略，即先解释Hibernate，然后再解释Java Persistence。

让我们从Hibernate开始，假设你编写一个依赖原生API的应用程序。

9.3.1 保存和加载对象

在一个Hibernate应用程序中，通过实质上改变它们的状态来保存和加载对象。你在工作单元中进行这项工作。单个工作单元是一组被认为是原子团的操作。如果你现在猜测它与事务密切相关，就对了。但是，没有必要是相同的东西。我们必须逐步解决这个问题；现在，把工作单元当作对已经组合到一起的对象进行状态改变的一个特定序列。

首先你必须开始一个工作单元。

1. 开始工作单元

在工作单元开始时，应用程序从SessionFactory处获得了Session的一个实例：

```
Session session = sessionFactory.openSession();
Transaction tx = session.beginTransaction();
```

此时，一个新的持久化上下文被初始化了，它将管理在这个Session中使用的所有对象。如果应用程序访问几个数据库，它可能会有多个SessionFactory。SessionFactory如何创建，以及你如何在应用程序代码中访问它依赖于开发环境和配置——如果遵循了2.1.3节中的设置，就应该具备了简单的HibernateUtil启动辅助类。

你永远不应该创建一个新的SessionFactory而只是服务一个特定的请求。SessionFactory的创建非常昂贵。另一方面，Session创建则非常便宜。Session甚至直到需要连接时才获得一个JDBC Connection。

前一段代码中的第二行在另一个Hibernate接口中开始一个Transaction。你在一个工作单元内部执行的所有操作都在一个事务内部发生，无论你是读取还是写入数据。但是，这个Hibernate API是可选的，你可以用喜欢的任何方式开始事务——我们将在第10章探讨这些选项。如果你使用Hibernate Transaction API，代码在所有的环境下都有效，因此你将在接下来的几节中给所有示例这么做。

打开一个新的Session和持久化上下文之后，就可以用它来加载和保存对象。

2. 使对象变成持久化

你要用Session做的第一件事就是通过save()方法使一个新的瞬时对象变成持久化（见代码清单9-2）。

代码清单9-2 使瞬时实例变成持久化

```
Item item = new Item();                                          ①
item.setName("Playstation3 incl. all accessories");
item.setEndDate( ... );

Session session = sessionFactory.openSession();                  ②
Transaction tx = session.beginTransaction();

Serializable itemId = session.save(item);                        ③

tx.commit();                                                     ④
session.close();                                                 ⑤
```

新的瞬时对象item像往常一样被实例化❶。当然，也可以在打开一个Session之后再对它进行实例化；它们目前并不相关。新的Session用SessionFactory❷打开。这样就启动了一个新事务。

调用save()❸使得Item的瞬时实例变成了持久化。它现在与当前的Session和它的持久化上下文关联。

对持久化对象所做的变化必须与数据库在某个时间点上进行同步。这发生在当你对Hibernate Transaction❹实施commit()的时候。我们说发生了清除（你也可以手工调用flush()；后面有关于这个话题的更多内容）。为了使持久化上下文同步，Hibernate获得了一个JDBC连接，并发出单个SQL INSERT语句。注意，对于插入来说，并非总是这样：Hibernate保证item对象在保存之后有一个被分配的数据库标识符，因此可能需要更早的INSERT，取决于你在映射中启用的标识符生成器。save()操作也返回持久化实例的数据库标识符。

Session最终可以关闭❺，并且持久化上下文终止。item引用现在是对于处于脱管状态的对象的一个引用。

在图9-4中可以看到相同的工作单元，以及对象如何改变状态。

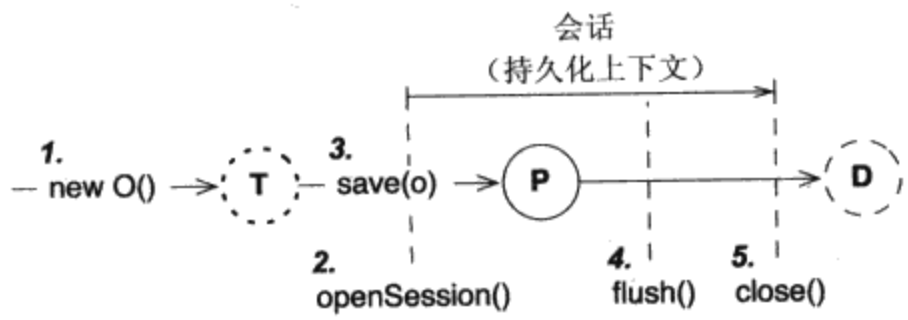


图9-4　在工作单元中使对象变成持久化

在用Session管理Item实例之前完全初始化它更好（但并非必要）。SQL INSERT语句包含调用save()时对象保存着的值。可以在调用save()之后修改对象，并且这种变化将被作为一个（额外的）SQL UPDATE传播到数据库。

Session.beginTransaction()和tx.commit()之间的一切都在一个事务中发生。现在记住，在事务范围中的所有数据库操作要么完全成功，要么完全失败。如果清除期间在tx.commit()上所做的其中一个UPDATE或者INSERT语句失败了，那么在这个事务中对持久化对象所做的所有变化便在数据库级回滚。然而，Hibernate不会把内存变化回滚到持久化对象。这是有道理的，因为事务的失败一般是不可恢复的，你必须立即放弃失败的Session。我们稍后将在第10章讨论异常处理。

3. 获取持久化对象

Session也用于查询数据库，并获取现有的持久化对象。Hibernate在这方面的功能尤其强大，你在本书稍后会体会到。它给最简单的查询提供了两种特殊的方法：通过标识符获取。get()和load()方法如代码清单9-3所示。

代码清单9-3 通过标识获取Item

```
Session session = sessionFactory.openSession();
Transaction tx = session.beginTransaction();

Item item = (Item) session.load(Item.class, new Long(1234));
// Item item = (Item) session.get(Item.class, new Long(1234));

tx.commit();
session.close();
```

可以在图9-5中看到相同的工作单元。

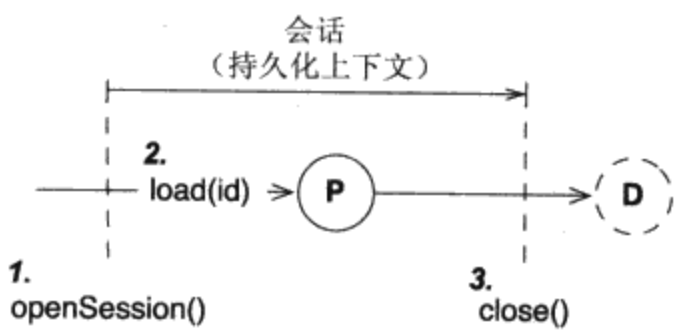


图9-5 通过标识符获取持久化对象

获取到的对象item处于持久化状态，并且一旦持久化上下文关闭，它就立即处于脱管状态。get()和load()之间的一个区别在于它们如何表明实例无法被找到。如果数据库中不存在包含给定标识符值的行，get()就返回null。load()方法则抛出一个ObjectNotFoundException。由你选择喜欢的错误处理方式。

更重要的是，load()方法可能返回一个代理（proxy），一个占位符，而不命中（hit）数据库。这个结果就是稍后你可能得到一个ObjectNotFoundException，一旦你试图访问返回的占位符，就立即强制它初始化。（这也称作*延迟加载*（lazy loading）；我们在后面的几章中讨论加载优化。）load()方法始终试图返回一个代理，如果它已经由当前的持久化上下文管理，则仅返回一个已被初始化的对象实例。在前面介绍过的例子中，根本没有发生数据库命中！另一方面，get()方法从不返回代理，它始终命中数据库。

你可能会问，这种方法为什么有用——毕竟，获取对象是要进行访问的。获得一个持久化实例，并把它作为对另一个实例的引用进行分配，这很常见。例如，假如需要item只为了一个目的：用Comment：aComment.setForAuction(item)设置一个关联。如果这就是你计划用item做的一切，那么代理会做得很好；不需要命中数据库。换句话说，当Comment被保存时，你需要item的外键值插入到COMMENT表中。item的代理正好提供包在一个占位符中的标识符值，这个占位符看起来就像真的一样。

4．修改持久化对象

由get()、load()返回的任何持久化对象，或者任何被查询的实体，都已经与当前的Session和持久化上下文关联。它可以被修改，并且其状态与数据库同步（请见代码清单9-4）。

代码清单9-4　修改持久化实例

```
Session session = sessionFactory.openSession();
Transaction tx = session.beginTransaction();

Item item = (Item) session.get(Item.class, new Long(1234));

item.setDescription("This Playstation is as good as new!");

tx.commit();
session.close();
```

图9-6展现了这个工作单元和对象转变。

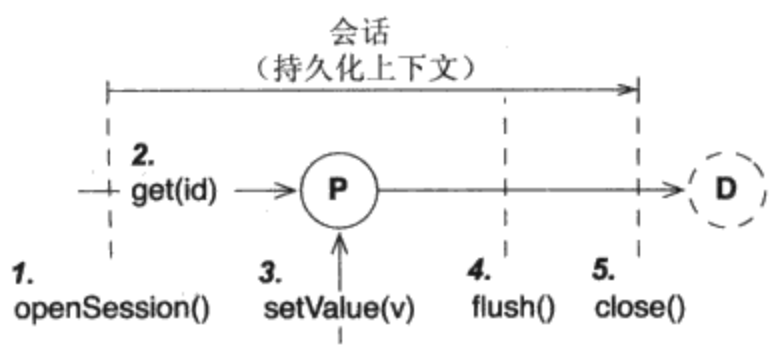


图9-6　修改持久化实例

首先，通过给定的标识符从数据库中获取对象。你修改对象，且当tx.commit()被调用时，这些修改在清除期间被传播到数据库。这种机制称作自动脏检查——意味着Hibernate追踪并保存在持久化状态中对一个对象所做的改变。一关闭Session，这个实例就被认为是脱管了。

5．使持久化对象变成瞬时

利用delete()方法可以轻松地使一个持久化对象变成瞬时（请见代码清单9-5），从数据库中移除它的持久化状态。

代码清单9-5　利用delete()使持久化对象变成瞬时

```
Session session = sessionFactory.openSession();
Transaction tx = session.beginTransaction();

Item item = (Item) session.load(Item.class, new Long(1234));

session.delete(item);

tx.commit();
session.close();
```

看看图9-7。

调用delete()之后，item对象处于移除（removed）状态；你不应该继续使用它，大多数情况下，应该确保应用程序中对它的任何引用都被移除了。只有当Session的持久化上下文在工作单元结束时与数据库同步，才执行SQL DELETE。Session关闭之后，item对象被认为是一个普通的瞬时实例。如果这个瞬时实例不再被任何其他对象引用，就会被垃圾收集器破坏。内存对象实例和持久化数据库行都将已经被移除了。

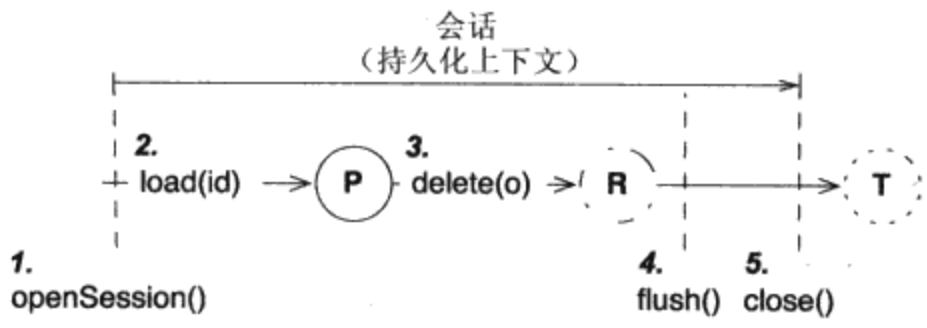


图9-7 使持久化对象变成瞬时

常见问题 必须加载要删除的对象吗？是的，对象必须加载到持久化上下文中；实例必须处于要被移除的持久化状态（注意代理就够好了）。原因很简单：你可能已经启用Hibernate拦截器，并且对象必须通过这些拦截器，完成它的生命周期。如果你直接在数据库中删除行，拦截器就不会运行了。话虽这么说，Hibernate（和Java Persistence）还是提供变成直接SQL DELETE语句的大批量操作；12.2节将讨论这些操作。

如果你启用hibernate.use_identifier_rollback配置选项，Hibernate也可以回滚已经被删除的任何实体的标识符。在前一个例子中，如果启用这个选项，Hibernate就会在删除和清除之后，设置被删除item的数据库标识符属性为null。然后它就成了一个可以在未来的工作单元中重用的干净的瞬时实例了。

6. 复制对象

目前为止，我们介绍在Session中的操作全部都是常见的；每一个Hibernate应用程序都需要它们。但是Hibernate可以用一些特殊的使用案例提供帮助——例如，当你需要从一个数据库获取对象并把它们保存在另一个数据库中的时候。这称作对象的复制（replication）。

复制采用加载在Session中的脱管对象，并在另一个Session中使它们变成持久化。这些Session通常在已经通过映射给同一个持久化类配置的两个不同的SessionFactory中打开。这里有个例子：

```
Session session = sessionFactory1.openSession();
Transaction tx = session.beginTransaction();
Item item = (Item) session.get(Item.class, new Long(1234));
tx.commit();
session.close();

Session session2 = sessionFactory2.openSession();
Transaction tx2 = session2.beginTransaction();
session2.replicate(item, ReplicationMode.LATEST_VERSION);
tx2.commit();
session2.close();
```

ReplicationMode控制复制过程的细节：

- ReplicationMode.IGNORE——当现有的数据库行包含与目标数据库中相同的标识符时忽略对象。

- ReplicationMode.OVERWRITE——覆盖任何包含与目标数据库中相同标识符的现有数据库行。
- ReplicationMode.EXCEPTION——如果现有的数据库行包含与目标数据库中相同的标识符时抛出异常。
- ReplicationMode.LATEST_VERSION——如果目标数据库的版本比对象的版本更早，则覆盖它里面的行，否则忽略对象。你需要启用的Hibernate乐观并发性控制。

当你任由数据进入不同的数据库时，当你在产品升级期间升级系统配置信息时（这经常涉及迁移到一个新的数据库实例），或者当你需要回滚在非ACID事务期间所做的改变时，都可能需要复制。

你现在知道了持久化生命周期和持久化管理器的基础操作。把这些与我们在前面章节中讨论过的持久化类映射一起使用，你就可以创建自己的小Hibernate应用程序了。映射一些简单的实体类和组件，然后在一个独立的应用程序中保存并加载对象。你不需要Web容器或者应用程序服务器：编写一个main()方法，并像我们在前一节讨论过的那样调用Session。

在接下来的几节中，我们讨论脱管对象状态，以及重附和合并持久化上下文之间的脱管对象的方法。这是你实现长工作单元（对话）所需要的基础知识。假设你熟悉本章前面所述的对象同一性范围。

9.3.2　使用脱管对象

Session关闭之后修改item，对于它在数据库中的持久化表示法没有影响。持久化上下文一关闭，item就变成了一个脱管实例。

如果想要保存对脱管对象所做的修改，必须重附或者合并它。

1. 重附被修改的脱管实例

脱管实例可以通过在托管对象上调用update()，被重附到新的Session（并由这个新的持久化上下文托管）。依据我们的经验，如果你在心里把update()改名为reattach()可能更容易理解——然而，它被称作更新也是有道理的。

update()方法在数据库中强制更新对象的持久化状态，始终计划一个SQL UPDATE。脱管对象处理的例子请见代码清单9-6。

代码清单9-6　更新脱管实例

```
item.setDescription(...); // Loaded in previous Session

Session sessionTwo = sessionFactory.openSession();
Transaction tx = sessionTwo.beginTransaction();

sessionTwo.update(item);

item.setEndDate(...);

tx.commit();
sessionTwo.close();
```

item对象是在传递到update()之前或者之后被修改都没有关系。这里重要的东西是对update()的调用把脱管实例重附到了这个新的Session（和持久化上下文）。Hibernate始终把这个对象当作脏对象来处理，并计划一个SQL UPDATE（将在清除期间执行）。可以在图9-8中看到相同的工作单元。

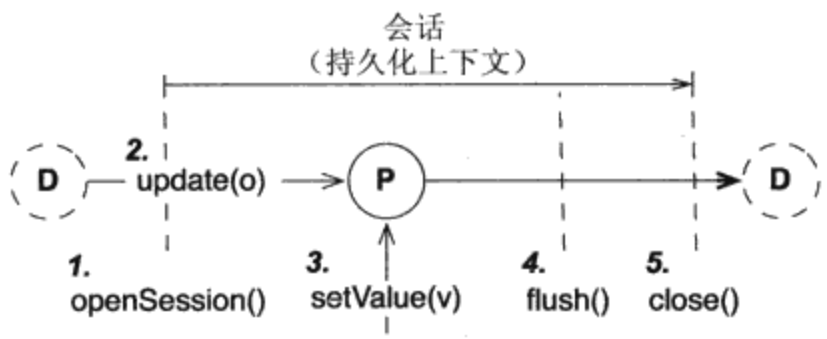


图9-8 重附脱管对象

你可能会很惊讶，或许本以为Hibernate会知道你修改了脱管item的描述（或者Hibernate应该知道你没有修改任何东西）。然而，新的Session和它的新持久化上下文没有这种信息。脱管对象也不包含已经进行的所有修改的一些内部清单。Hibernate只好假设数据库中需要UPDATE。避免这个UPDATE语句的一种方法是，通过select-before-update="true"属性配置Item的类映射。然后Hibernate通过执行一个SELECT语句，并把对象的当前状态与当前的数据库状态进行比较，确定该对象是否为脏对象。

如果确定没有修改脱管实例，你可能会更喜欢另一种重附方法：它不会总是计划一个数据库更新。

2. 重附未被修改的脱管实例

对lock()的调用把对象与Session和它的持久化上下文关联起来，不强制更新，如代码清单9-7所示。

代码清单9-7 用lock()重附脱管实例

```
Session sessionTwo = sessionFactory.openSession();
Transaction tx = sessionTwo.beginTransaction();

sessionTwo.lock(item, LockMode.NONE);

item.setDescription(...);
item.setEndDate(...);

tx.commit();
sessionTwo.close();
```

在这个例子中，变化是在对象被重附之前或者之后进行的确很重要。在调用lock()之前进行的改变没有被传播到数据库，只有当你确定脱管实例还没有被修改时才使用它。这种方法只保证对象的状态从脱管变成持久化，并保证Hibernate会再次管理持久化对象。当然，一旦你对处于托管的持久化状态的对象进行任何修改时，就需要更新数据库。

第10章会讨论Hibernate锁模式。这里通过指定LockMode.NONE，告诉Hibernate不要执行版本检查或在使对象与Session关联时重新获得任何数据库级锁。如果指定了LockMode.READ或者LockMode.UPGRADE，Hibernate就会执行SELECT语句，以便执行版本检查（并锁定数据库中的行进行更新）。

3. 使脱管对象变成瞬时

最后，可以使脱管实例变成瞬时，从数据库中删除它的持久化状态，如代码清单9-8所示。

代码清单9-8　利用delete()使脱管对象变成瞬时

```
Session session = sessionFactory.openSession();
Transaction tx = session.beginTransaction();

session.delete(item);

tx.commit();
session.close();
```

这意味着你不一定要重附（利用update()或者lock()）脱管实例来把它从数据库中删除。在这个例子中，调用delete()具有两重功能：重附对象到Session，然后计划删除对象（在tx.commit()中执行）。delete()调用之后的对象状态为移除。

脱管对象的重附是在几个Session之间传输数据的唯一可行的方法。可以用另一个选项使对脱管实例的修改与数据库同步，通过它的状态合并（merging）。

4. 合并脱管对象的状态

脱管对象的合并是另一种可以选择的方法。它可以是重附的补充，也可以取代重附。合并最初引入到Hibernate中，是用来处理重附无法再满足的特殊案例（merge()在Hibernate 2.x中的旧名称是saveOrUpdateCopy()）。看看下列代码，试着重附一个脱管对象：

```
item.getId(); // The database identity is "1234"
item.setDescription(...);

Session session = sessionFactory.openSession();
Transaction tx = session.beginTransaction();

Item item2 = (Item) session.get(Item.class, new Long(1234));

session.update(item); // Throws exception!

tx.commit();
session.close();
```

上述代码给定了一个包含数据库同一性的脱管item对象。修改它之后，你尝试把它重附到一个新的Session。但在重附之前，另一个表示相同数据库行的实例已经被加载到了这个Session的持久化上下文中。显然，通过update()进行的重附与这个已经持久化的实例相抵触，抛出NonUniqueObjectException。异常的错误信息为：persistent instance with the same database identifier is already associated with the Session!（包含相同数据库标识符的持久化实例已经与会话关联！）Hibernate无法确定哪个对象表示当前状态。

可以先通过重附item解决这种情况；然后，由于对象处于持久化状态，无需获取item2。在

像这个例子一样简单的代码中，这很容易理解，但是它或许不可能在更为复杂的应用程序中进行重构。毕竟，客户端把脱管对象发送到了持久层让它变成托管，并且客户端可能不（且不应该）知道已经处于持久化上下文中的托管实例。

可以让Hibernate自动合并item和item2：

```
item.getId() // The database identity is "1234"
item.setDescription(...);

Session session= sessionFactory.openSession();
Transaction tx = session.beginTransaction();

Item item2 = (Item) session.get(Item.class, new Long(1234));

Item item3 = (Item) session.merge(item);

(item == item2) // False
(item == item3) // False
(item2 == item3) // True

return item3;

tx.commit();
session.close();
```

❷

❸

看看图9-9中的工作单元。

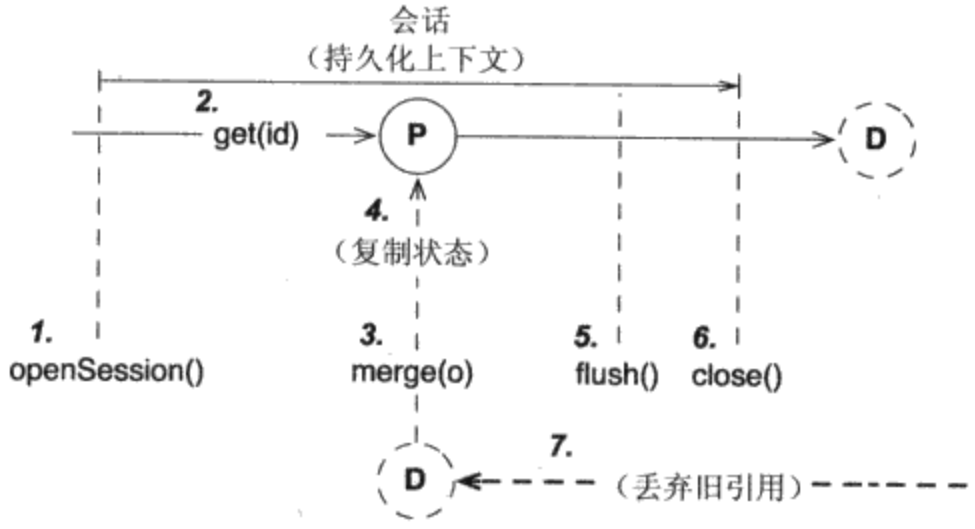


图9-9 把脱管实例合并到持久化实例

merge(item)调用❸导致了几个动作。第一，Hibernate检查持久化上下文中的持久化实例是否具有与正在合并的脱管实例相同的数据库标识符。在这个例子中，答案是肯定的：item和item2都用get()❷得到加载，它们具有相同的主键值。

如果持久化上下文中有相等的持久化实例，Hibernate把脱管实例的状态复制到持久化实例中去❹。换句话说，已经在脱管item上设置的新描述也设置在了持久化item2上。

如果持久化上下文中没有相等的持久化实例，Hibernate就从数据库中加载它（就像你用get()所做的那样，通过标识符有效地执行相同的获取），然后把脱管状态与被获取的对象的状态合并。如图9-10所示。

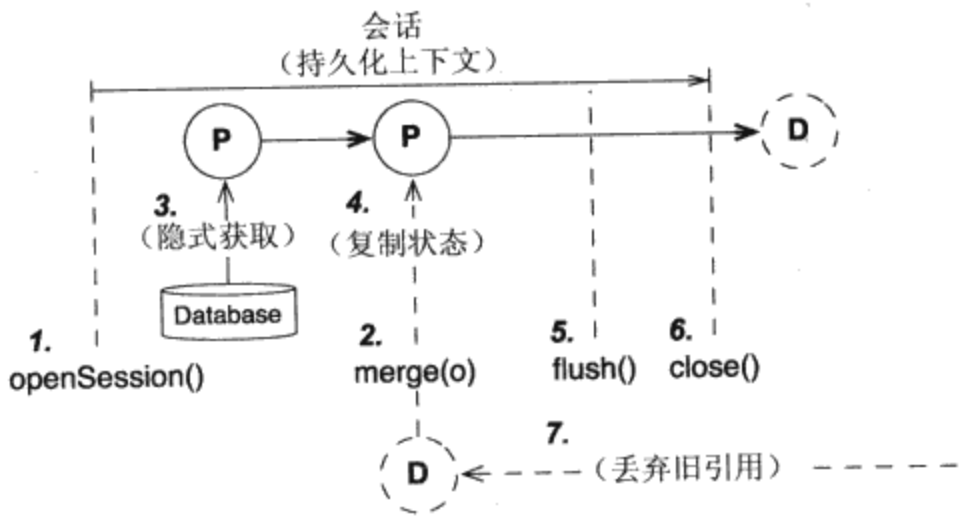


图9-10　把脱管实例合并到一个隐式加载的持久化实例

如果持久化上下文中没有相等的持久化实例，并且在数据库中的查找没有结果，就会创建新的持久化实例，并且把被合并的实例的状态复制到新实例中。然后就计划把这个新对象插入到数据库中，并通过merge()操作返回。

如果传到merge()里面的实例是一个瞬时实例，而不是脱管对象时，也会发生插入。

你可能会有下列疑问：

- 到底从item把什么复制到了item2？合并包括所有值类型的属性，以及任何集合中元素的增加和移除。
- item处于什么状态？与持久化实例合并的任何脱管对象保持脱管状态。它没有改变状态；不受合并操作的影响。因此，item和其他两个引用在Hibernate的同一性范围中是不同的。（前两个同一性在最后一个例子中进行检验）。但是item2和item3是对相同持久化内存实例的相同引用。
- 为什么item3从merge()操作中被返回？merge()操作始终返回一个句柄到它已经与之合并了状态的持久化实例。这对调用了merge()的客户端来说很方便，因为它现在可以继续使用脱管的item对象，并在需要时再次合并它；或者可以放弃这个引用，继续使用item3。区别很明显：Session完成之前，如果后来对item2或者item3所做的修改是在合并之后，客户端就完全不知道这些修改。客户端只有一个对脱管item对象的句柄，现在也过时了。但是，如果客户端决定在合并之后扔掉item，并继续使用返回的item3，在最新的状态上就有了一个新句柄。合并之后item和item2都应该被认为是废弃的。

状态的合并比重附略微复杂一些。我们认为，如果围绕脱管对象设计应用程序逻辑，它就是你有时候可能必须使用的一个基本操作。可以把这个策略作为重附的另一种可选方案，并且每一次都合并而不重附。也可以用它使任何瞬时实例变成持久化。如本章稍后所述，这是Java Persistence的标准模型；不支持重附。

目前为止，我们还没有太关注持久化上下文，以及它是如何管理持久化对象的。

9.3.3　管理持久化上下文

持久化上下文具有很多功能：自动脏检查、保证对象同一性范围等。了解它管理的一些细节，

和你有时候影响了幕后在进行的东西同等重要。

1. 控制持久化上下文高速缓存

持久化上下文是持久化对象的一个高速缓存。持久化状态中的每个对象都为持久化上下文所知，并且每个持久化实例的复制、快照（snapshot）都保存在高速缓存中。这个快照内部用于进行脏检查，侦测对持久化对象所做的任何修改。

忽视这个简单事实的许多Hibernate用户都会遇到OutOfMemoryException。这是当在一个Session中加载上千个对象却永远不想修改它们时的典型案例。Hibernate仍然必须在持久化上下文高速缓存中创建每个对象的快照，并保持对托管对象的引用，它可能导致内存耗尽。（很显然，如果修改上千个对象，就应该执行大批量数据操作（bulk data operation）——12.2节会回到这种工作单元的话题。）

持久化上下文高速缓存从不自动收缩。为了减少或者重新获得被持久化上下文在特定工作单元中消费掉的内存，必须做下列事情：

- 保持持久化上下文的大小为必需的最小尺寸。Session中的许多持久化实例经常意外出现——例如，因为你只需要几个却查询了许多个。只有当这个状态绝对需要它们时才使对象变成持久化；非常大的图表可能严重影响性能，并需要大量的内存用于状态快照。检查你的查询是否只返回需要的对象。如本书稍后所述，也可以在Hibernate中执行一个查询，返回只读状态的对象，而不用创建持久化上下文快照。
- 可以调用session.evict(object)，从持久化上下文高速缓存中手工脱管一个持久化实例。可以调用session.clear()，从持久化上下文中脱管所有持久化实例。脱管对象不用检查脏状态；它们没有被托管。
- 利用session.setReadOnly(object,true)，可以禁用对特定实例的脏检查。如果它是只读的，持久化上下文将不再维持快照。利用session.setReadOnly(object,false)，可以给一个实例重新启用脏检查，并强制重新创建快照。注意这些操作并不改变对象的状态。

工作单元结束时，你做过的所有修改都必须通过SQL DML语句与数据库同步。这个过程称作持久化上下文的清除。

2. 清除持久化上下文

Hibernate的Session实现了迟写。在持久化上下文的范围中对持久化对象所做的改变并没有立即传播到数据库。这样允许Hibernate把许多改变接合到最少的数据库请求中去，帮助把网络延时的影响减到最小。尽可能迟地（趋向于数据库事务的结束）执行DML的另一个很棒的副作用，在于缩短了数据库内部锁的持续时间。

例如，如果对象的单个属性在同一个持久化上下文中被改变两次，Hibernate就只需要执行一个SQL UPDATE。迟写有用的另一个例子是，当执行多个UPDATE、INSERT或者DELETE语句时，Hibernate能够利用JDBC批量API。

持久化上下文与数据库的同步被称作清除。Hibernate清除发生在以下几个时间点：

- 当Hibernate API中的Transaction被提交时；
- 执行查询之前；

- 应用程序显式地调用Session.flush()时。

需要在工作单元结束时把Session状态清除到数据库，是为了使变化可以持续，这是常见的案例。注意，事务提交时的自动清除是Hibernate API的一项特性！提交一个包含JDBC API的事务不会触发清除。Hibernate不会在每个查询之前清除。如果变化保存在会影响查询结果的内存中，Hibernate就会默认地先进行同步。

可以通过调用session.setFlushMode()，显式地设置Hibernate的FlushMode，来控制这个行为。默认的清除模式为FlushMode.AUTO，并启用前面所说的行为。如果你选择了FlushMode.COMMIT，在查询执行之前，持久化上下文不会被清除（只有在手工调用Transaction.commit()或者Session.flush()时，它才会被清除）。这个设置可能让你面临废弃的数据：对只处在内存中的托管对象所做的修改，可能与查询的结果冲突。通过选择FlushMode.MANUAL，可以指定只有显式地调用flush()才导致托管状态与数据库同步。

控制持久化上下文的FlushMode，在本书稍后扩展上下文为跨对话时将很有必要。

持久化上下文的重复清除经常成为性能问题的根源，因为持久化上下文中所有的脏对象都必须在清除时间被发现。一种常见的原因在于，特定的工作单元模式多次重复查询—修改—查询—修改的顺序。每一次修改都导致在每个查询之前进行一次清除和所有持久化对象的脏检查。在这种情况下，FlushMode.COMMIT可能比较恰当。

永远记着，清除过程的性能部分取决于持久化上下文的大小——它管理的持久化对象的数量。因而，我们在前一节中对管理持久化上下文所提的建议，在这里仍然适用。

现在你已经见过了最重要的策略和在Hibernate应用程序中与对象交互的一些可选的策略，以及在Hibernate Session中可以使用什么方法和操作。如果你准备只利用Hibernate API进行工作，就可以跳过下一节，直接进入第10章学习事务。如果你想利用Java Persistence和（或）EJB 3.0组件处理对象，就接着往下读。

9.4　JPA

现在我们用JPA保存和加载对象。这是你可以在Java SE应用程序中或者利用EJB 3.0组件时所用的API，作为Hibernate本地接口一个供应商独立的可选方案。

你已经读完了本章的前面部分，知道JPA定义的对象状态，以及它们如何与Hibernate的对象状态相关。因为这两个非常相似，因此本章第一部分无论选择哪种API都没有关系。它遵循你与对象交互的方法，并且操作数据库的方式也很类似。因此，假设你已经学会了前一节中的Hibernate接口（如果你跳过了前一节，就会没有印象；这里我们不再重复）。这很重要还有另一个原因：JPA提供Hibernate原生API的超集功能的一个子集。换句话说，每当你需要的时候，就有理由退回到原生的Hibernate接口。你可以期待在应用程序中需要的大多数功能都已经被这个标准涵盖了，这没有太大必要。

就像在Hibernate中一样，你通过JPA操作对象的当前状态来保存和加载对象。并且，就像在Hibernate中一样，你在工作单元中完成它，这被认为是原子的一组操作。（我们仍然还没有涵盖足够的内容来阐述所有关于事务的知识点，但我们很快就会做到。）

为了在一个Java Persistence应用程序中启动工作单元，需要有一个EntityManager（相当于

Hibernate的Session)。然而，在Hibernate应用程序中从SessionFactory打开Session的地方，可以用托管和非托管的工作单元编写Java Persistence应用程序。我们要把它保持简单，并假设你首先想要编写一个不从托管环境中的EJB 3.0组件受益的JPA应用程序。

9.4.1 保存和加载对象

术语非托管是指用Java Persistence创建持久层的可能性，它不用任何特殊的运行时环境就可以运行和工作。可以不通过应用程序服务器在运行时容器之外、简单的Java SE应用程序中使用JPA。这可以是个servlet应用程序（Web容器不提供你需要用于持久化的东西）或者简单的main()方法。另一个常见的案例是用于桌面应用程序的本地持久化，或者用于两层系统的持久化，在这里桌面应用程序访问远程的数据库层（虽然对于为什么你不能在这样的场景中利用带有EJB支持的、轻量化的模块应用程序服务器，还没有很好的解释。）

1. 在Java SE中启动工作单元

无论如何，由于你没有可以提供EntityManager的容器，因此需要手工创建一个。与Hibernate SessionFactory相当的是JPA EntityManagerFactory：

```
EntityManagerFactory emf =
    Persistence.createEntityManagerFactory("caveatemptorDatabase");
EntityManager em = emf.createEntityManager();
EntityTransaction tx = em.getTransaction();
tx.begin();
```

这段代码的第一行是系统配置的一部分。你应该为在Java Persistence应用程序中部署的每个持久化单元都创建一个EntityManagerFactory。我们已经在2.2.2节中涵盖了这一点，因此这里不再重复。接下来的三行相当于在一个独立的Hibernate应用程序中启动工作单元时要做的：首先，创建EntityManager，然后启动事务。为了让自己熟悉EJB 3.0的行话，可以称这个EntityManager为应用程序托管的（application-managed）。这里启动的事务也有一个特殊的描述：它是本地资源（resource-local）的事务。你正在控制着应用程序代码直接涉及的资源（在这个例子中指数据库）；没有运行时容器替你关照这个。

创建EntityManager时，分配了一个新的持久化上下文。你在这个上下文中保存和加载对象。

2. 使实体实例变成持久化

实体类与你其中一个Hibernate持久化类相同。当然，你通常更喜欢用注解映射实体类，取代Hibernate XML映射文件。毕竟，使用Java Persistence的（主要）原因在于标准接口和映射带来的好处。

让我们创建实体的一个新实例，并把它从瞬时状态带到持久化状态：

```
Item item = new Item();
item.setName("Playstation3 incl. all accessories");
item.setEndDate( ... );
EntityManager em = emf.createEntityManager();
EntityTransaction tx = em.getTransaction();
tx.begin();

em.persist(item);

tx.commit();
em.close();
```

如果你已经理解了本章前面几节的内容，这段代码应该看起来很熟悉了。一旦你在瞬时的item实体实例中调用persist()，它立即就变成持久化；现在它被托管在持久化上下文中。注意，persist()不返回实体实例的数据库标识符值（与Hibernate的save()方法相比的这个小差别，在11.2.3节中实现会话时，将会再次变得重要）。

常见问题　应该在Session中使用persist()吗？Hibernate的Session接口也特写了persist()方法。它有着与JPA的persist()操作相同的语义。然而关于清除，这两个操作之间有着重要的差别。在同步期间，即使用这个选项映射了一个关联，Hibernate的Session也不会把persist()操作级联到被关联的实体和集合。它只是被级联到调用persist()时可以到达的实体！如果使用Session API，则只有save()（和update()）在清除时被级联。但是在JPA应用程序中则相反：只有persist()在清除时被级联。

被托管的实体实例受到监控。你在持久化状态中对实例所做的修改在某个时间点与数据库同步（除非放弃工作单元）。由于EntityTransaction由应用程序托管，你需要手工进行commit()。应用相同的规则到应用程序控制的EntityManager：你要通过关闭它来释放所有资源。

3．获取实体实例

EntityManager也用来查询数据库，并获取持久化的实体实例。Java Persistence支持复杂的查询特性（我们将在本书稍后讨论到）。最基础的部分和往常一样，是通过标识符获取（retrieval by identifier）：

```
EntityManager em = emf.createEntityManager();
EntityTransaction tx = em.getTransaction();
tx.begin();

Item item = em.find(Item.class, new Long(1234));
tx.commit();
em.close();
```

你不需要转换find()操作的返回值；它是个一般的方法，其返回类型设置为第一个参数的一个附带作用。这是JPA一个很小但很方便的好处——Hibernate原生的方法必须与不支持泛型的更早期的JDK一起工作。

被获取的实体实例处于持久化状态，现在可以在工作单元内部被修改，或者脱管在持久化上下文之外使用。如果找不到包含指定标识符的持久化实例，find()就返回null。find()操作始终命中数据库（或者一个特定于供应商的透明高速缓存），因此实体实例始终在加载期间被初始化。可以期待稍后它所有的值都可以在脱管状态中使用。

如果不想命中数据库，因为你不确定是否需要完全初始化的实例，可以告诉EntityManager尝试获取一个占位符：

```
EntityManager em = emf.createEntityManager();
EntityTransaction tx = em.getTransaction();
tx.begin();

Item item = em.getReference(Item.class, new Long(1234));
```

```
tx.commit();
em.close();
```

这个操作返回被完全初始化的item（例如，如果当前的持久化上下文中已经有实例）或者一个代理（空的占位符）。

一旦你试图访问不是数据库标识符属性的item的任何属性时，就立即执行一个额外的SELECT来完全初始化这个占位符。这也意味着在这时候（或者甚至更早，执行getReference()的时候）应该会有一个EntityNotFoundException。合乎逻辑的结论是，如果决定脱管item引用，就不保证它会被完全初始化（当然，除非在脱管之前访问它其中一个非标识符属性）。

4．修改持久化的实体实例

处于持久化状态的实体实例由当前的持久化上下文管理。可以修改它，并期待持久化上下文在同步时清除必要的SQL DML。这是由Hibernate Session提供的相同的自动脏检查特性：

```
EntityManager em = emf.createEntityManager();
EntityTransaction tx = em.getTransaction();
tx.begin();

Item item = em.find(Item.class, new Long(1234));
item.setDescription(...);

tx.commit();
em.close();
```

持久化实体实例通过它的标识符值获取。然后，你修改其中一个被映射的属性（还没有用@Transient或者transient的Java关键字注解的一个属性）。在这个代码实例中，下一次与数据库的同步发生在本地资源事务被提交的时候。Java Persistence引擎执行必要的DML，在这个例子中是一个UPDATE。

5．使持久化实体实例变成瞬时

如果想要从数据库中移除一个实体实例的状态，就必须使它变成瞬时状态。在EntityManager中使用remove()方法：

```
EntityManager em = emf.createEntityManager();
EntityTransaction tx = em.getTransaction();
tx.begin();

Item item = em.find(Item.class, new Long(1234));

em.remove(item);

tx.commit();
em.close();
```

Java Persistence remove()方法的语义与Hibernate Session中的delete()方法相同。之前的持久化对象现在处于移除状态，你应该放弃应用程序中存有对它的任何引用。在持久化上下文的下一个同步期间执行一个SQL DELETE。JVM垃圾收集器侦测item不再被任何人引用，并最终删除该对象的最后轨迹。但是注意，不能在脱管状态中在一个实体实例上调用remove()，否则会抛出异常。你必须先合并脱管实例，然后移除被合并的对象（或者，利用相同的标识符获得一个引用，并移除之）。

6. 清除持久化上下文

对持久化实体实例所做的所有修改都在某个时间点与数据库同步，这个过程称作清除。这种迟写行为与Hibernate的相同，通过尽可能迟地执行SQL DML来保证最佳的可伸缩性。

每当EntityTransaction中的commit()被调用时，就清除EntityManager的持久化上下文。本章这一节中所有之前的代码示例都已经使用了这一策略。但是JPA实现被允许在其他的时间点上与持久化上下文同步，如果它们希望如此的话。

Hibernate作为一个JPA实现，在下列时间点同步：

- 当EntityTransaction被提交时；
- 执行查询之前；
- 当应用程序显式地调用em.flush()时。

这些规则与我们在前一节中对原生Hibernate阐述的一样。就像在原生的Hibernate中一样，可以用JPA接口FlushModeType控制这个行为：

```
EntityManager em = emf.createEntityManager();
em.setFlushMode(FlushModeType.COMMIT);
EntityTransaction tx = em.getTransaction();
tx.begin();

Item item = em.find(Item.class, new Long(1234));
item.setDescription(...);

List result = em.createQuery(...).getResultList();
tx.commit();
em.close();
```

对于EntityManager把FlushModeType转换为COMMIT，就会在查询之前禁用自动同步；它只在提交事务或者手工清除时才会发生。默认的FlushModeType是AUTO。

就像利用原生的Hibernate一样，控制持久化上下文的同步行为对于会话的实现将是很重要的功能，我们稍后会讨论到。

你现在知道了Java Persistence的基本的操作，可以继续在自己的应用程序中保存和加载一些实体实例。像2.2节中描述的一样，建立系统，并利用注解映射一些类到数据库模式中。编写使用EntityManager和EntityTransaction的main()方法；相信你很快就会明白，使用Java Persistence甚至不用EJB 3.0托管组件或者应用程序服务器有多么容易。

接下来讨论如何使用脱管的实体实例。

9.4.2　使用脱管的实体实例

假设你已经知道如何定义脱管对象（如果不知道，再读一遍本章的第一小节）。你没有必要知道如何在Hibernate中使用脱管对象，但如果使用Java Persistence的策略与原生Hibernate中的一样，我们就会让你参考之前的章节。

首先，再来看看实体实例在Java Persistence应用程序中如何变成脱管状态。

1. JPA持久化上下文范围

你已经在Java SE环境中使用过Java Persistence，利用应用程序管理的持久化上下文和事务。

每一个持久化的和托管的实体实例都在持久化上下文关闭时变成脱管状态。但是我们并没有告诉过你持久化上下文什么时候关闭。

如果你熟悉原生的Hibernate，就已经知道了答案：当Session关闭时，持久化上下文终止。EntityManager是JPA中的等价物；默认情况下，如果你创建了EntityManager，持久化上下文就会被界定到这个EntityManager实例的生命周期。

看看下列代码：

```
EntityManager em = emf.createEntityManager();
EntityTransaction tx = em.getTransaction();

tx.begin();
Item item = em.find(Item.class, new Long(1234));
tx.commit();

item.setDescription(...);

tx.begin();
User user = em.find(User.class, new Long(3456));
user.setPassword("secret");
tx.commit();

em.close();
```

在第一个事务中，你获取了Item对象。然后事务完成，但item仍然处于持久化状态。因而，在第二个事务中，你不仅加载了User对象，而且在提交第二个事务时更新了被修改的持久化item（除了对脏user实例的更新之外）。

就像在原生的Hibernate中使用Session一样，持久化上下文始于createEntityManager()，止于close()。

关闭持久化上下文不是脱管实体实例的唯一途径。

2. 实体实例的手工脱管

实体实例在它离开持久化上下文时变成脱管状态。EntityManager中的一种方法允许你清除持久化上下文，并脱管所有持久化实例：

```
EntityManager em = emf.createEntityManager();
EntityTransaction tx = em.getTransaction();

tx.begin();
Item item = em.find(Item.class, new Long(1234));

em.clear();

item.setDescription(...); // Detached entity instance!

tx.commit();
em.close();
```

获取item之后，你清除EntityManager的持久化上下文。由这个持久化上下文管理的所有实体实例现在都脱管了。脱管实例的修改不在提交期间与数据库同步。

常见问题　单独实例的清除在哪里？Hibernate Session API具备evict(object)方法，Java Persistence则没有这个能力。原因可能只有一些专家组成员才知道——我们无法解释。（注意，这种方式很好地说明了专家们不可能对该操作语义达成一致意见。）可以只完全清除持久化上下文，并脱管所有的持久化对象。如果你想要从持久化上下文中清除单独的实例，就必须退回到2.2.4节中所述的Session API。

很显然，你也想要保存在某个时间点对脱管实体所做的修改。

3. 合并脱管的实体实例

尽管Hibernate提供了重附和合并两种策略使脱管对象的任何变化与数据库同步，但Java Persistence却只提供后者。假设你已经在前一个持久化上下文中获取了一个item实体实例，并且现在想修改它并保存这些修改：

```
EntityManager em = emf.createEntityManager();
EntityTransaction tx = em.getTransaction();
tx.begin();

Item item = em.find(Item.class, new Long(1234));
tx.commit();
em.close();

item.setDescription(...); // Detached entity instance!

EntityManager em2 = emf.createEntityManager();
EntityTransaction tx2 = em2.getTransaction();
tx2.begin();

Item mergedItem = (Item) em2.merge(item);

tx2.commit();
em2.close();
```

item在第一个持久化上下文中被获取，并且在脱管状态中的修改后，合并到一个新的持久化上下文。merge()操作完成几件事情：

首先，Java Persistence引擎检查持久化上下文中的持久化实例是否与你正在合并的脱管实例有着相同的数据库标识符。因为，在我们的代码示例中，第二个持久化上下文中没有相等的持久化实例，有一个通过标识符进行查找从数据库中被获取。然后，脱管的实体实例被复制到持久化实例。换句话说，已经在脱管的item中设置的新描述也在持久化的mergeItem上设置了，它从merge()操作中返回。

如果在持久化上下文中没有相等的持久化实例，并且在数据库中通过标识符进行的查找是否定的，被合并的实例就被复制到一个新的持久化实例，然后当第二个持久化上下文与数据库同步时就被插入到数据库。

状态的合并是重附的另一种选择（与原生的Hibernate提供的一样）。参考我们前面在9.3.2节中讨论的利用Hibernate的合并；这两个API提供相同的语义，若作必要的修正（mutatis mutandis），其中的注释将适用于JPA。

你现在准备扩展Java SE应用程序，并在Java Persistence中体验使用持久化上下文和脱管对象。不是只在单个工作单元中保存和加载实体实例，而是尝试使用几个工作单元，尝试合并脱管对象的修改。别忘了观察SQL日志，看看幕后发生了什么。

一旦你掌握了使用Java SE的基础Java Persistence操作，就可能想要在托管环境中做同样的事。从完全EJB 3.0容器的JPA中得到的好处很有价值。你不再需要自己管理EntityManager和EntityTransaction。可以关注自己想要做的事：加载和保存对象。

9.5 在 EJB 组件中使用 Java Persistence

托管的运行时环境仿佛是某种容器。应用程序组件就存在这个容器里面。当今的大部分容器都利用一种拦截技术来实现，拦截对象上的方法调用，并应用需要在这个方法之前（或者之后）执行的任何代码。这对任何横切关注点（cross-cutting concern）来说都是最好的：打开和关闭EntityManager是必然的，因为你要它处在被调用的方法里面。你的业务逻辑不需要关注这个方面。事务划分是容器可以替你负责的另一个关注点。（你可能在任何应用程序中发现其他的方面。）

不像EJB 2.x时代更早的应用程序服务器，支持EJB 3.0和其他Java EE 5.0服务的容器很容易安装和使用——请参考2.2.3节，为下一节准备好系统。此外，EJB 3.0编程模型以简单的Java类为基础。如果你见到我们在本书中编写许多EJB，不应该感到惊讶；大多数时候，与简单的JavaBean的唯一区别仅仅在于一个简单的注解，一项你希望使用的由运行组件的环境所提供的一项服务的声明。如果你无法修改源代码和添加注解，可以用XML部署描述符把一个类变成一个EJB。因而，（几乎）每个类都可以成为EJB 3.0中的托管组件，使你更容易从Java EE 5.0服务中受益。

目前为止你已经创建的实体类还不足以编写一个应用程序。你还要无状态或者有状态的会话bean、可以用来封装应用程序逻辑的组件。在这些组件内部，你需要容器的服务：例如，你通常要容器注入（inject）EntityManager，以便可以加载和保存实体实例。

9.5.1 注入 EntityManager

还记得在Java SE中如何创建EntityManager的实例吗？必须手工从EntityManagerFactory中打开和关闭它。还必须用EntityTransaction接口启动和终止本地资源的事务。

在EJB 3.0服务器中，容器管理的EntityManager可以通过依赖注入（dependency injection）使用。考虑在CaveatEmptor应用程序中实现特定动作的下列EJB会话bean：

```
@Stateless
public class ManageAuctionBean implements ManageAuction {

    // Use field injection:
    @PersistenceContext
    private EntityManager em;

    // or setter injection:
    //
    // @PersistenceContext
    // public void setEntityManager(EntityManager em) {
    //     this.em = em;
```

```
    // }

    @TransactionAttribute(TransactionAttributeType.REQUIRED)
    public Item findAuctionByName(String name) {
       return (Item) em.createQuery()...
       ...
    }
}
```

这是个无状态的动作，它实现ManageAuction接口。无状态EJB的这些细节不是我们目前的关注点，值得关注的是可以在这个动作的findAuctionByName()方法中访问EntityManager。容器在动作方法执行之前，自动把EntityManager的一个实例注入到bean的em字段里面。字段的可见性对于容器来说并不重要，但是需要应用@PersistenceContext注解来表明你想要这个容器的服务。也可以给这个字段创建一个公有的设置方法，并在这个方法上应用注解。如果你还计划手工设置EntityManager，建议采用这种方法——例如，在集成测试或者功能测试期间。

被注入的EntityManager由这个容器维护。你不必清除或者关闭它，也不必启动和终止事务——你在前一个例子中告诉容器：会话bean的findAuctionByName()方法需要一个事务。(对于所有EJB会话bean方法而言，这是默认的。）当方法被客户端调用（或者新事务自动启动）时，事务必须是活动的。当方法返回时，事务要么继续，要么被提交，取决于它是否为这个方法所启动。

被注入的容器管理的EntityManager的持久化上下文由事务的范围决定。因而，当事务终止时它也自动被清除和关闭。这是个重要的区别，如果把它与前面在Java SE中介绍的例子进行比较的话。那里的持久化上下文没有被界定到事务，而是界定到你显式地关闭的EntityManager实例。事务范围的持久化上下文对于无状态bean是自然的默认，就像当你稍后关注对话实现和事务时，在接下来的章节中所看到的那样。

显然只对使用JBoss EJB 3.0有效的一种很好的技巧是Session（而不是EntityManager对象的自动注入）：

```
@Stateless
public class ManageAuctionBean implements ManageAuction {

    @PersistenceContext
    private Session session;
    ...

}
```

如果有一个托管组件要依赖Hibernate API，这最有用。

这是对两个数据库（也就是说，两个持久化单元）都有效的一种变形：

```
@Stateless
public class ManageAuctionBean implements ManageAuction {

    @PersistenceContext(unitName = "auctionDB")
    private EntityManager auctionEM;

    @PersistenceContext(unitName = "auditDB")
    private EntityManager auditEM;

    @TransactionAttribute(TransactionAttributeType.REQUIRED)
    public void createAuction(String name, BigDecimal price) {
```

```
            Item newItem = new Item(name, price);
            auctionEM.persist(newItem);
            auditEM.persist( new CreateAuctionEvent(newItem) );
            ...
        }
    }
```

unitName指向被配置和部署的持久化单元。如果你使用数据库（EntityManager或者SessionFactory），就不需要给注入声明持久化单元的名称。注意，来自两个不同持久化单元的EntityManager实例不共用相同的持久化上下文。一般地，两个都是托管实体对象的独立高速缓存，但这并不意味着它们不能参与相同的系统事务。

如果用Java Persistence编写EJB，你的选择就很清楚了：想要包含正确持久化上下文的EntityManager通过容器注入到托管组件。你很少使用的另一种方法是查找（lookup）容器托管的EntityManager。

9.5.2 查找 EntityManager

不是让容器在字段或者设置方法中注入EntityManager，而是当你需要它的时候从JNDI中查找：

```
@Stateless
@PersistenceContext(name = "em/auction", unitName = "auctionDB")
public class ManageAuctionBean implements ManageAuction {
    @Resource
    SessionContext ctx;

    @TransactionAttribute(TransactionAttributeType.REQUIRED)
    public Item findAuctionByName(String name) {
        EntityManager em = (EntityManager) ctx.lookup("em/auction");

        return (Item) em.createQuery()...
    }

}
```

在这个代码片段中发生了几件事情：第一，你声明了想要bean的组件环境中填充有EntityManager，并且假定绑定引用的名称为em/auction。JNDI中的全称为java:comp/env/em/auction——java:comp/env/部分就是所谓的bean命名上下文（bean-naming context）。在JNDI这个子上下文（subcontext）中的一切都是依赖bean。换句话说，EJB容器读取这个注解，并知道它在这个bean执行的运行时必须只给该bean绑定EntityManager，在为这个bean保存的JNDI中的命名空间下。

利用SessionContext，在bean实现中查找EntityManager。这个上下文的好处在于它自动用java:comp/env/给正在查找的名称加上前缀；因而，它尝试在bean的命名上下文中查找引用，而不是全局的JNDI命名空间。@Resource注解指示EJB容器注入SessionContext。

当EntityManager中的第一个方法被调用时，持久化上下文由容器创建；并且当事务终止（当这个方法返回）时，它被清除和关闭。

如果需要EntityManagerFactory，也可以使用注入和查找。

9.5.3　访问 EntityManagerFactory

EJB容器也允许你直接为持久化单元访问EntityManagerFactory。没有托管环境，你必须在Persistence引导类（bootstrap class）的帮助下创建EntityManagerFactory。在容器中，可以再次利用自动的依赖注入获得EntityManagerFactory：

```
@Stateless
public class ManageAuctionBean implements ManageAuction {

    @PersistenceUnit(unitName = "auctionDB")
    EntityManagerFactory auctionDB;

    @TransactionAttribute(TransactionAttributeType.REQUIRED)
    public Item findAuctionByName(String name) {
        EntityManager em = auctionDB.createEntityManager();
        ...
        Item item = (Item) em.createQuery()...
        ...
        em.flush();
        em.close();
        return item;
    }
}
```

unitName属性是可选的，只有当你有不止一个被配置的持久化单元（几个数据库）时才需要它。你从被注入的工厂中创建的EntityManager再次成为应用程序托管的——容器将不清除这个持久化上下文，也不会关闭它。你很少把容器托管的工厂与应用程序托管的EntityManager实例混合，但是如果你需要对EJB组件中EntityManager的生命周期的更多控制时，这么做就很有帮助。

可以在任何JTA事务范围之外创建EntityManager；例如，在一个不需要事务上下文的EJB方法中。然后必要时，利用joinTransaction()方法通知EntityManager一个JTA事务是活动的，这就是你的责任了。注意，这个操作没有把持久化上下文绑定或者界定到JTA事务；它只是内部转换EntityManager到事务行为的一个提示。

前面的语句还没有完成：如果close()这个EntityManager，它不会立即关闭它的持久化上下文，如果这个持久化上下文已经与一个事务关联的话。当事务结束时，持久化上下文关闭。但是，对被关闭的EntityManager的任何调用都会抛出异常（除了Java SE中的getTransaction()方法和isOpen()方法之外）。可以用hibernate.ejb.discard_pc_on_close配置设置转换这个行为。如果你永远不在事务范围之外调用EntityManager，就不必担心这一点。

访问EntityManagerFactory的另一个原因，可能是你想在这个接口中访问一个特定的供应商扩展，就像我们在2.2.4节中讨论过的一样。

如果先把EntityManagerFactory绑定到EJB的命名上下文中，也可以查找它：

```
@Stateless
@PersistenceUnit(name= "emf/auction", unitName = "auctionDB")
public class ManageAuctionBean implements ManageAuction {
```

```
    @Resource
    SessionContext ctx;

    @TransactionAttribute(TransactionAttributeType.REQUIRED)
    public Item findAuctionByName(String name) {
        EntityManagerFactory auctionDB =
            (EntityManagerFactory) ctx.lookup("emf/auction");

        EntityManager em = auctionDB.createEntityManager();
        ...
        Item item = (Item) em.createQuery()...
        ...

        em.flush();
        em.close();
        return item;
    }
}
```

如果把这个查找方法与自动的注入进行比较，会发现它还是没有特别的优势。

9.6 小结

本章已经谈到了不少内容。你现在知道了Java Persistence中的基本接口与Hibernate提供的那些没有多大区别。加载和保存对象几乎相同。但是持久化上下文的范围略有不同；在Hibernate中，默认与Session范围相同。在Java Persistence中，持久化上下文的范围很不相同，取决于你是自己创建EntityManager，还是让容器管理并它把绑定到EJB组件中的当前事务范围。

表9-1展现了可以用来比较原生的Hibernate特性和Java Persistence的一个概述。

表9-1 第9章中Hibernate和JPA对照表

Hibernate Core	Java Persistence和EJB 3.0
Hibernate定义和依赖4种对象状态：瞬时、持久化、删除和脱管	相当的对象状态在EJB 3.0中被标准化和定义
脱管对象可以被重附到一个新的持久化上下文或者被合并到持久化实例中	Java Persistence管理接口只支持合并
在清除时，save()和update()操作可以被级联到所有被关联的和可获得的实例。persist()操作只能被级联到调用时可获得的实例	在清除时，persist()操作可以被级联到所有被关联的和可获得的实例。如果退回到Session API，save()和update()也只能被级联到调用时可获得的实例
get()命中数据库；load()可能返回代理	find()命中数据库；getReference()可以返回代理
EJB中Session的依赖注入只在JBoss应用程序服务器中有效	EntityManager的依赖注入在所有EJB组件中都有效

我们已经讲到了应用程序中的对话，以及如何利用脱管对象或者利用被扩展的持久化上下文设计它们。虽然我们还没有时间非常深入地进行讨论，但你可能已经看到，使用脱管对象工作需要规定（在受保护的对象同一性范围之外）和手工重附或者合并。在实际应用程序中，依据我们多年的经验，建议把脱管对象当作次要的方法，并且先看一下包含被扩展的持久化上下文的对话实现。

遗憾的是，我们仍然没有利用所有的东西编写一个包含对话的真正复杂的应用程序。你可能特别想知道更多有关事务的信息。第10章涵盖了事务的概念和接口。

第 10 章

事务和并发

本章内容

- 数据库事务
- 使用Hibernate和Java Persistence的事务
- 非事务的数据访问

本章终于讨论事务以及如何在应用程序中创建和控制工作单元了。我们将介绍事务如何在最低级别（数据库）工作，以及在基于原生Hibernate、Java Persistence和用或者不用Enterprise JavaBeans的应用程序中如何处理事务。

事务允许你设置工作单元的范围：一个原子的操作组。它们还帮助你在多用户的应用程序中把一个工作单元从另一个中隔离出来。我们讨论并发，以及如何在应用程序中利用悲观和乐观的策略控制并发的数据访问。

最后，看一下非事务的数据访问，以及什么时候应该在自动提交模式下使用数据库。

10.1 事务本质

从一些背景信息开始。应用程序功能要求同时完成几件事情。例如，当一次拍卖结束时，CaveatEmptor应用程序必须执行3项不同的任务：

(1) 给胜出（金额最高）的出价做标记。

(2) 向卖主收取拍卖费用。

(3) 通知卖主和成功的出价人。

如果你由于外部信用卡系统的失败而无法收取拍卖费用，会发生什么事？业务需求可能规定，列出的所有动作必须要么都成功，要么都不成功。如果这样，你可以把这些步骤全部称为事务（transaction）或者工作单元。哪怕只有一个步骤失败，则整个工作单元都必定失败。这就是大家所知的原子性（atomicity），即所有操作都作为一个原子单元来执行。

此外，事务允许多个用户同时使用相同的数据，而不破坏数据的完整性和正确性；特定的事务不应该对其他同时运行的事务可见。为了完全理解这个隔离性（isolation）行为，有几个策略很重要，我们将在本章中给予探讨。

事务具有其他重要的属性，例如一致性（consistency）和持久性（durability）。一致性意味着事务始终工作在同一组数据上：从其他同时运行的事务中隐藏起来的、并在事务完成之后留在一个清洁和一致的状态中的一组数据。数据库完整性规则保证一致性。你也想要事务的正确性（correctness）。例如，业务规则规定向卖主收取一次费用，而不是两次。这是个合理的假设，但是你可能无法用数据库约束把它表达出来。因而，事务的正确性是应用程序的责任，而一致性则是数据库的责任。持久性意味着一旦事务完成，所有变化都在该事务变成持久化期间进行，即使系统后来失败了，这些变化也不会丢失。

把这些事务属性归结起来，就是大家所知的ACID标准。

数据库事务必须简短。单个的事务通常只涉及单批数据库操作。在实践中，还需要一个概念，允许你有长期运行的对话，在这里一个原子组合的数据库操作不是在一批而是几批中发生。对话允许应用程序的用户有思考时间，而仍然保证原子、隔离且一致的行为。

既然已经定义了术语，就可以讨论事务划分（demarcation）以及如何定义一个工作单元的范围了。

10.1.1 数据库和系统事务

数据库把工作单元的概念实现为一个数据库事务（database transaction）。数据库事务组合了数据库访问操作——也就是SQL操作。所有SQL语句都在一个事务内部执行；无法把SQL语句发送到数据库事务之外的数据库。事务被确保以这两种方式之一终止：要么完全被提交（commit），要么完全被回滚（roll back）。因而，我们说数据库事务是原子的。在图10-1中，你可以看到这个图示。

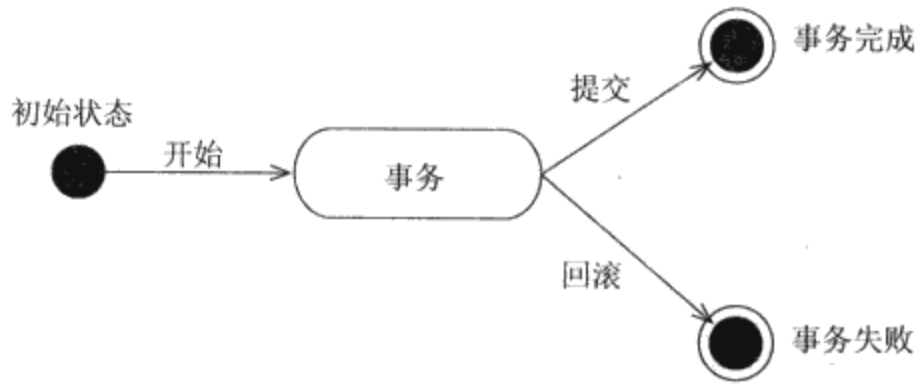


图10-1 一个原子工作单元（事务）的生命周期

为了在事务内部执行所有的数据库操作，必须给这个工作单元的范围做标记。必须启动事务，并在某个时间点提交变化。如果出现错误（在执行操作或者提交事务的时候），就必须回滚事务，把数据留在一致的状态中。这就是大家所知的事务划分，取决于你所使用的技术，它涉及更多或者更少的手工干涉。

一般来说，启动和终止事务的事务范围可以在应用程序代码中编程式地设置，或者声明式地设置。

1. 编程式的事务划分

在非托管环境中，JDBC API被用来给事务范围做标记。通过在JDBC `Connection`中调用

setAutoCommit(false)启动事务，并通过调用commit()终止它。可以在任何时候通过调用rollback()强制立即回滚。

在一个于几个数据库中操作数据的系统中，特定的工作单元涉及对不止一个资源的访问。既然如此，你就无法单独通过JDBC实现原子性。你需要可以在系统事务（system transaction）中处理几个资源的事务管理器（transaction manager）。这样的事务处理系统公开了与开发人员进行交互的Java Transaction API（JTA）。JTA中的主API是UserTransaction接口，包含begin()和commit()系统事务的方法。

此外，Hibernate应用程序中的编程式事务管理通过Hibernate Transaction接口公开给应用程序开发人员。你并没有被强制使用这个API——Hibernate也让你直接启动和终止JDBC事务，但不鼓励这种用法，因为它把代码绑定到了直接的JDBC。在Java EE环境中（或者如果你把它和Java SE应用程序一起安装了），就可以使用JTA兼容的事务管理器，因此你应该调用JTA UserTransaction接口来编程式地启动和终止事务。然而，就像你可能已经猜到的，Hibernate Transaction接口在JTA的顶层也有效。我们将介绍所有这些方法，并更详细地讨论可移植性的关注点。

利用Java Persistence的编程式事务划分，还必须在Java EE应用程序服务器的内部和外部使用。在应用程序服务器外部，利用简单的Java SE，处理本地资源事务；这是EntityTransaction接口的作用——你已经在前面几章中见过它。在应用程序服务器内部，调用JTA UserTransaction接口来启动和终止事务。

让我们概括一下这些接口，以及什么时候使用它们：

- java.sql.Connection——利用setAutoCommit(false)、commit()和rollback()进行简单的JDBC事务划分。它可以但不应该被用在Hibernate应用程序中，因为它把应用程序绑定到了一个简单的JDBC环境。
- org.hibernate.Transaction——Hibernate应用程序中统一的事务划分。它适用于非托管的简单JDBC环境，也适用于以JTA作为底层系统事务服务的应用程序服务器。但是，它最主要的好处在于与持久化上下文管理的紧密整合——例如，你提交时Session被自动清除。持久化上下文也可以拥有这个事务的范围（对会话有用，请见第11章）。如果你无法具备JTA兼容的事务服务，就使用Java SE中的这个API。
- javax.transaction.UserTransaction——Java中编程式事务控制的标准接口，它是JTA的一部分。每当你具备JTA兼容的事务服务，并想编程式地控制事务时，它就应该成为你的首选。
- javax.persistence.EntityTransaction——在使用Java Persistence的Java SE应用程序中，编程式事务控制的标准接口。

另一方面，声明式事务划分不需要额外的代码；并且从定义上来说，它解决了可移植性的问题。

2. 声明式事务划分

在应用程序中，当你希望在一个事务内部进行工作的时候要进行声明（例如，在方法中使用注解）。然后处理这个关注点就是应用部署程序和运行时环境的责任了。Java中提供声明式事务

服务的标准容器是EJB容器，这项服务也称作容器托管事务。我们将再次编写EJB会话bean，介绍Hibernate和Java Persistence如何从这项服务中受益。

在你决定一种特定的API之前，或者为了声明式事务划分，让我们一步步来探讨这些选项。首先，假设你正要在一个简单的Java SE应用程序（一个客户端/服务器端Web应用程序、桌面应用程序或者任何两层系统）中使用原生的Hibernate。之后，要把代码重构到一个托管的Java EE环境下运行（并且首先看看如何避免这种重构）。我们也顺便讨论Java Persistence。

10.1.2 Hibernate 应用程序中的事务

想象你正在编写一个必须在简单的Java中运行的Hibernate应用程序；没有容器、没有托管的数据库资源可以使用。

1. Java SE中的编程式事务

配置Hibernate为你创建一个JDBC连接池，就像在2.1.3节中所做的那样。如果你正在利用Transaction API编写Java SE Hibernate应用程序，除了连接池之外，不需要其他的配置设置：

- hibernate.transaction.factory_class选项默认为org.hibernate.transaction.JDBCTransactionFactory，这是Java SE中Transaction API以及直接的JDBC的正确工厂。
- 可以用自己的TransactionFactory实现扩展和定制Transaction接口。这几乎没有必要，但是有一些值得关注的使用案例。例如，每当启动事务时，如果你必须编写审计日志，就可以把这个日志添加到一个定制的Transaction实现。

Hibernate为你正要使用的每个Session获得一个JDBC连接：

```
Session session = null;
Transaction tx = null;

try {
    session = sessionFactory.openSession();
    tx = session.beginTransaction();

    concludeAuction(session);

    tx.commit();
} catch (RuntimeException ex) {
    tx.rollback();
} finally {
    session.close();
}
```

Hibernate Session是延迟的。这是件好事——意味着它不消费任何资源，除非绝对需要。只有当数据库事务启动时，才从连接池中获得JDBC Connection。对beginTransaction()的调用在新的JDBC Connection中转变成setAutoCommit(false)。现在Session被绑定到了这个数据库连接，并且所有的SQL语句（在这个例子中，终止拍卖所需要的所有SQL）都在该连接上发送。所有的数据库语句都在同一个数据库事务内部执行。（假设concludeAuction()方法调用指定的Session来访问数据库。）

我们已经谈到过迟写行为，因此你知道当Session的持久化上下文被清除时，大批量的SQL语句被尽可能迟地执行。默认情况下，这发生在当你在Transaction上调用commit()的时候。提交事务（或者把它回滚）之后，数据库连接被释放，并且从Session中解除绑定。用相同的Session启动一个新的事务，从连接池中获取另一个连接。

关闭Session释放出所有其他的资源（例如，持久化上下文）；所有托管的持久化实例现在都被认为是脱管的。

常见问题　回滚只读事务更快吗？如果事务中的代码读取数据但没有修改它，你应该回滚事务而不是提交它吗？这样更快吗？很显然，有些开发人员发现这在某些特殊环境下会更快，并且这种信念已经传播到了整个社区。我们用更为普及的数据库系统进行过测试，发现并没有区别。我们也没有发现显示性能差别的真实数据的任何来源。对于数据库系统为什么应该有一个并非最佳的实现——为什么它不应该内部使用最快的事务清除算法，也没有很好的解释。始终提交事务，如果提交失败就回滚。话虽这么说，但SQL标准还是包括了一个SET TRANSACTION READ ONLY（设置事务只读）的语句。Hibernate不支持启用这个设置的API，虽然你可以实现自己定制的Transaction和TransactionFactory来添加这个操作。我们建议你先查一下数据库是否支持这一点，以及可能的性能受益有什么，如果有，就可以了。

我们现在要讨论异常处理。

2. 处理异常

如果上一个例子中介绍过的concludeAuction()（或者提交期间持久化上下文的清除）抛出异常，你就必须调用tx.rollback()强制事务回滚。它会立即回滚事务，因此你发送到数据库的SQL操作对性能没有任何影响。

这似乎很简单，虽然或许你可能已经看到，每当你想要访问数据库的时候，捕捉RuntimeException都不会产生好代码。

说明　异常的历史——异常以及异常应该如何处理，始终是Java开发人员之间争论的热点。Hibernate也有一些值得瞩目的历史，这并不奇怪。在Hibernate 3.x之前，Hibernate抛出的所有异常均为checked exception（已检查异常），因此每个Hibernate API都强制开发人员捕捉和处理异常。这个策略受到JDBC影响，它也只抛出checked exception。但是很快就清楚了，这样没有意义，因为Hibernate抛出的所有异常都是致命的。在许多情况下，开发人员在这种情形之下所能做的就是清除，显示错误消息，并退出应用程序。因此，从Hibernate 3.x开始，Hibernate抛出的所有异常都是unchecked（未检查）的RuntimeException的子类型，它通常在应用程序中单个的位置进行处理。这也使得所有Hibernate模板或者包装API都变得没用了。

首先，即使我们承认你不用许多（或者成百个）try/catch块编写应用程序代码，我们介绍过的例子也是不完整的。这是一个Hibernate工作单元标准惯用语的例子，带有包含真实的异常处

理的数据库事务：

```
Session session = null;
Transaction tx = null;

try {
    session = sessionFactory.openSession();
    tx = session.beginTransaction();

    tx.setTimeout(5);

    concludeAuction(session);

    tx.commit();
} catch (RuntimeException ex) {
    try {
        tx.rollback();
    } catch (RuntimeException rbEx) {
        log.error("Couldn't roll back transaction", rbEx);
    }
    throw ex;
} finally {
    session.close();
}
```

任何Hibernate操作，包括清除持久化上下文，都可能抛出RuntimeException。甚至回滚事务也可能抛出异常！你想要捕捉这个异常并记录到日志中；否则，导致回滚的原始异常就被淹没了。

这个例子中可选的方法调用是setTimeout()，它获得允许事务运行的秒数。然而，在Java SE环境中并没有真正被监测的事务。如果在应用程序服务器外部运行这段代码（也就是说，没有事务管理器），Hibernate所能做的最好的就是给驱动器等待PreparedStatement执行的时间设置秒数（Hibernate专用的预编译语句）。如果超出限制，就抛出SQLException。

你不想用这个例子作为自己应用程序中的一个模板，因为你应该通过一般的基础结构代码来隐藏异常处理。例如，可以给RuntimeException编写单个错误处理程序，知道什么时候以及如何回滚事务。对于开启和关闭Session也一样。稍后第11章会用更真实的例子讨论它，16.1.3节也会再次讨论它。

Hibernate抛出类型（typed）异常，即帮助你辨别错误的RuntimeException的所有子类型：

- 最常见的HibernateException是个一般的错误。你必须检查异常消息，或者在通过异常中调用getCause()找出更多原因。
- JDBCException是被Hibernate的内部JDBC层抛出的任何异常。这种异常总是由一个特定的SQL语句产生，可以用getSQL()获得这个引起麻烦的语句。JDBC连接（实际上是JDBC驱动器）抛出的内部异常可以通过getSQLException()或者getCause()获得，并且通过getErrorCode()可以得到特定于数据库和特定于供应商的错误代码。
- Hibernate包括JDBCException的子类型和一个内部转换器，该转换器试图把数据库驱动抛出器的特定于供应商的错误代码变成一些更有意义的东西。内建的转换器可以给Hibernate支持的最重要的数据库方言生成JDBCConnectionException、SQLGrammarException、LockAquisitionException、DataException和ConstraintViolationException。

可以对数据库操作或者增强方言，或者插入SQLExceptionConverterFactory定制这种变换。

- Hibernate抛出的其他RuntimeException也应该终止事务。应该始终确保捕捉RuntimeException，无论你计划利用任何细粒度的异常处理策略去做什么。

你现在知道了应该捕捉什么异常，以及它们什么时候会出现。但是你心里可能有一个问题：捕捉到异常之后应该怎么办？

Hibernate抛出的所有异常都是致命的。这意味着你必须回滚数据库事务，并关闭当前的Session。不允许你继续使用抛出异常的Session。

通常，你还必须在关闭异常后面的Session之后退出应用程序，虽然有一些异常（例如StaleObjectStateException）一般导致在新Session中一个新的尝试（可能在再次与应用程序的用户交互之后）。由于这些与会话以及并发控制密切相关，我们将在稍后讨论它们。

常见问题　可以把异常用于验证吗？有些开发人员一旦知道Hibernate可以抛出许多细粒度的异常类型后就很兴奋。这样可能把你引向歧途。例如，你可能试图捕捉ConstrainViolationException用于验证。如果一个特定的操作抛出异常，为什么不显示一条（根据错误代码和文本而定制的）失败消息给应用程序的用户，让他们纠正错误呢？这个策略有两个明显的缺点。第一，依靠数据库抛出unchecked值来查看问题，对于可伸缩的应用程序而言，这并不是一种好策略。你想要至少在应用层实现一些数据完整性的验证。第二，所有的异常对于当前的工作单元而言都是致命的。然而，这并不是应用程序的用户对验证错误的理解——他们希望仍然处在工作单元内部。围绕这种不匹配编写的代码很笨拙也很困难。我们的建议是，使用细粒度的异常类型显示更好看的（致命）错误消息。这么做在开发期间有帮助（理想情况下，不应该有致命的异常出现在产品中），也帮助任何客户支持必须迅速做出决定的工程师，如果它是个应用程序错误（违背约束，执行了错误的SQL）或者如果数据库系统正在加载（无法获得锁）。

利用Hibernate Transaction接口在Java SE中编程式的事务划分，使代码保持可移植。它也可以在托管环境内部运行，当事务管理器处理数据库资源的时候。

3．使用JTA的编程式事务

能与Java EE兼容的托管运行时环境，能够为你管理资源。在大多数情况下，被管理的资源都是数据库连接，但是任何带有适配器（adaptor）的资源都可以与Java EE系统整合（如消息或者遗留系统）。在那些资源上的编程式事务划分，如果它们是事务的，就被统一并公开给使用JTA的开发人员；javax.transaction.UserTransaction是启动和终止事务的主要接口。

常见的托管运行时环境是Java EE应用程序服务器。当然，应用程序服务器提供更多的服务，而不仅仅是资源管理。许多Java EE服务是模块化的——安装应用程序服务器不是获得它们的唯一方法。如果你只需要托管的资源，可以获得一个独立的JTA提供程序。开源独立的JTA提供程序包括JBoss Transactions（http://www.jboss.com/products/transactions）、ObjectWeb JOTM（http://jotm.objectweb.org）和其他的。可以和Hibernate应用程序一起安装这样一个JTA服务（例如在Tomcat中）。它将替你管

理数据库连接池，给事务划分提供JTA接口，并通过JNDI注册提供托管的数据库连接。

下列是使用JTA的托管资源的好处，以及使用这个Java EE服务的理由：

- 事务管理服务可以统一所有的资源，无论什么类型，并用单个标准的API把事务控制公开给你。这意味着你可以替换Hibernate的`Transaction` API，并直接在任何地方使用JTA。然后，在（或者用）JTA兼容的运行时环境中安装应用程序就是应用程序部署人员的责任了。这个策略把可移植性关注点归属的位置转移了；应用程序依赖标准的Java EE接口，并且运行时环境必须提供实现。
- Java EE事务管理器可以在单个事务中获取多个资源。如果你使用几个数据库（或者不止一个资源），就可能想要一个两阶段提交（two-phase commit）协议来保证跨资源范围的事务原子性。在这样的场景中，Hibernate通过配置几个`SessionFactory`（每个数据库一个），他们的`Session`获得所有参与同一个系统事务的托管数据库连接。
- 与简单的JDBC连接池相比，JTA实现的质量通常更高。应用程序服务器和作为应用程序服务器模块的独立JTA提供程序，通常在包含大量事务的高端系统中已经进行过更多的测试。
- JTA提供程序在运行时不增加不必要的过载（一种常见的错误概念）。简单案例（单个JDBC数据库）的处理与使用简单的JDBC事务一样有效。比起与简单的JDBC一起使用的随机连接池库，JTA服务背后托管的连接池可能是更好的软件。

假设你不喜欢JTA，并想要继续使用Hibernate `Transaction` API来保证代码在Java EE中和使用托管Java EE服务时可以运行，而不用改变任何代码。为了部署前面的代码实例（它们全部在Java EE应用程序服务器上调用Hibernate `Transaction` API），你需要把Hibernate配置转换到JTA：

- `hibernate.transaction.factory_class`选项必须设置为`org.hibernate.transaction.JTATransactionFactory`。
- Hibernate需要知道你正在哪个JTA实现中部署，这出于两个原因：第一，不同的实现可能公开JTA `UserTransaction`，Hibernate现在必须以不同的名称内部调用它。第二，Hibernate必须钩进JTA事务管理器的同步过程来处理它的高速缓存。你必须设置`hibernate.transaction.manager_lookup_class`选项来配置这两项：例如，配置为`org.hibernate.transaction.JBossTransactionManagerLookup`。在类中查找最常见的JTA实现，并且应用程序服务器中附有Hibernate（需要时可以定制）。在Javadoc中检查是否有包。
- Hibernate不再负责管理JDBC连接池；它从运行时容器获得托管的数据库连接。这些连接被JTA提供程序通过JNDI这个全局的注册而被公开。你必须在JNDI上使用正确的名称对数据库资源配置Hibernate，就像在2.4.1节中所做的那样。

现在，你之前直接在JDBC顶层给Java SE编写的同一段代码，在包含托管数据源的JTA环境中也有效：

```
Session session = null;
Transaction tx = null;

try {
    session = sessionFactory.openSession();
    tx = session.beginTransaction();
```

```
        tx.setTimeout(5);

        concludeAuction(session);

        tx.commit();
    } catch (RuntimeException ex) {
        try {
            tx.rollback();
        } catch (RuntimeException rbEx) {
            log.error("Couldn't roll back transaction", rbEx);
        }
        throw ex;
    } finally {
        session.close();
    }
```

然而，数据库连接处理稍微不同。Hibernate给你正在使用的每个Session获得一个托管的数据库连接，并且还是努力尽可能地延迟。没有JTA，Hibernate将从一开始就停在一个特定的数据库连接上，直接事务终止。有了JTA配置，Hibernate甚至更为积极：获得一个连接，并只用于单个SQL语句，然后立即被返回到托管的连接池。应用程序服务器保证当另一个SQL语句再次需要连接时，它将在同一个事务期间分发出同一个连接。这个积极的连接-释放模式是Hibernate的内部行为，对于应用程序以及如何编写代码，都没有任何影响。（因而，这个代码实例每一行都与原来的一样。）

JTA系统支持全局的事务超时；它可以监控事务。因此，现在setTimeout()控制全局的JTA超时设置——相当于调用UserTransaction.setTransactionTimeout()。

Hibernate Transaction API用Hibernate配置一个简单的变化来保证可移植性。如果你想要把这个责任转移到应用部署程序上，就应该针对标准的JTA接口编写代码。为使下列代码更值得关注，你还将在同一个系统事务内部使用两个数据库（两个SessionFactory）：

```
UserTransaction utx = (UserTransaction) new InitialContext()
                        .lookup("java:comp/UserTransaction");

Session session1 = null;
Session session2 = null;

try {
    utx.begin();

    session1 = auctionDatabase.openSession();
    session2 = billingDatabase.openSession();

    concludeAuction(session1);
    billAuction(session2);

    session1.flush();
    session2.flush();

    utx.commit();
} catch (RuntimeException ex) {
    try {
        utx.rollback();
    } catch (RuntimeException rbEx) {
```

```
            log.error("Couldn't roll back transaction", rbEx);
        }
        throw ex;
    } finally {
        session1.close();
        session2.close();
    }
```

（注意，这个代码片段可以抛出一些其他的checked exception，就像从JNDI查找中抛出的NamingException。你需要对这些做相应的处理。）

首先，必须从JNDI注册上获得JTA UserTransaction上的句柄。然后，开启和关闭事务，并且所有的Hibernate Session所用的（容器提供）数据库连接都在这个事务中被自动获取。即使你没有使用Transaction API，也仍然应该给JTA和环境配置hibernate.transaction.factory_class和hibernate.transaction.manager_lookup_class，以便Hibernate可以与事务系统内部进行交互。

利用默认的设置，手工flush()每个Session，使它与数据库同步（来执行所有SQL DML）也是你的责任了。Hibernate Transaction API以前自动为你完成。你还必须手工关闭所有的Session。另一方面，可以启用hibernate.transaction.flush_before_completion和（或）hibernate.transaction.auto_close_session配置选项，并再次让Hibernate替你负责这项工作——然后，清除和关闭就成为事务管理器内部同步过程的一部分，并且发生在JTA事务终止之前（相应地，或者之后）。启用这两项设置的代码可以简化为如下：

```
UserTransaction utx = (UserTransaction) new InitialContext()
                        .lookup("java:comp/UserTransaction");

Session session1 = null;
Session session2 = null;

try {
    utx.begin();

    session1 = auctionDatabase.openSession();
    session2 = billingDatabase.openSession();

    concludeAuction(session1);
    billAuction(session2);

    utx.commit();
} catch (RuntimeException ex) {
    try {
        utx.rollback();
    } catch (RuntimeException rbEx) {
        log.error("Couldn't roll back transaction", rbEx);
    }
    throw ex;
}
```

session1和session2持久化上下文现在在UserTransaction的提交期间被自动清除，并且两者都在事务完成之后关闭。

建议尽可能地直接使用JTA。你应该始终努力把可移植性的责任转移到应用程序之外，如果

可以，要求部署在一个提供JTA的环境中。

编程式的事务划分需要针对事务划分接口而编写的应用程序代码。避免任何不可移植的代码传播到整个应用程序的一种更好的方法是，*声明式*（declarative）事务划分。

4．容器托管事务

声明式事务划分意味着容器替你负责这个关注点。你*声明*想要代码在一个事务中是否参与以及如何参与。通过应用程序部署人员，提供支持声明式事务划分容器的责任再次落到了它所属的地方。

CMT是Java EE尤其JEB的标准特性，接下来要介绍的代码是基于EJB 3.0会话bean的（只适用于Java EE），你利用注解定义事务范围。注意，如果你必须使用更早版本的EJB 2.1会话bean，实际的数据访问代码并没有改变；然而，必须用XML编写EJB部署描述符来创建事务装配——这在EJB 3.0中是可选的。

（独立的JTA实现不提供容器托管和声明式事务。但是JBoss应用程序服务器可以当作模块服务器使用，包含最少的模块，如有必要，它可以只提供JTA和EJB 3.0容器。）

假设EJB 3.0会话bean实现了一个终止拍卖的动作。你之前用编程式的JTA事务划分编写的代码被移到了一个无状态会话bean里面：

```
@Stateless
public class ManageAuctionBean implements ManageAuction {

    @TransactionAttribute(TransactionAttributeType.REQUIRED)
    public void endAuction(Item item) {
        Session session1 = auctionDatabase.openSession();
        Session session2 = billingDatabase.openSession();

        concludeAuction(session1, item);
        billAuction(session2, item);
    }
    ...
}
```

容器注意到了你的TransactionAttribute声明，并把它应用到endAuction()方法。如果调用方法时没有系统事务在运行，就会启动一个新的事务（这是REQUIRED）。一旦方法返回，并且如果调用这个方法（不是任何其他方法）时启动了事务，这个事务就提交。如果方法内部的代码抛出RuntimeException，系统事务就自动回滚。

为了方便举例，我们再给两个数据库引入两个SessionFactory。它们可以通过JNDI查找（Hibernate可以在启动时绑定它们）或者从HibernateUtil的增强版中进行分配。这两者都获得通过相同的容器托管事务而获取的数据库连接。如果容器的事务系统和资源支持它，你就再次获得一个两阶段提交协议，确保跨数据库的事务原子性。

必须用Hibernate设置一些配置选项来启用CMT：

- hibernate.transaction.factory_class选项必须被设置为org.hibernate.transaction.CMTTransactionFactory。
- 你需要给应用程序服务器设置hibernate.transaction.manager_lookup_class为正确的查找类。

还要注意，所有EJB会话bean都默认为CMT，因此如果你想要禁用CMT并在任何会话bean方法中直接调用JTA UserTransaction，就用@TransactionManagement（TransactionManagementType.BEAN）注解EJB类。然后你就是在使用bean托管事务（bean-managed transaction，BMT）了。即使它可能适用于大多数的应用程序服务器，但是Java EE规范还是不允许在单个bean中混合CMT和BMT。

CMT代码看起来已经比编程式的事务划分好多了。如果配置Hibernate来使用CMT，它就知道应该自动清除和关闭参与系统事务的Session。此外，你将很快改进这段代码，甚至移除打开Hibernate Session的那两行。

来看看Java Persistence应用程序中的事务处理。

10.1.3 使用 Java Persistence 的事务

使用Java Persistence也要进行设计抉择：是在应用程序代码中编程式地事务划分，还是由运行时容器自动处理声明式事务划分。先通过简单的Java SE探讨第一种方法，然后用JTA和EJB组件重复例子。

描述本地资源（resource-local）的事务应用到由应用程序（编程式地）控制的、且不参与全局系统事务的所有事务。它们直接变为你正在处理的资源的本地事务系统。由于你正在使用JDBC数据库，这意味着本地资源的事务变成了JDBC数据库事务。

JPA中本地资源的事务通过EntityTransaction API控制。这个接口不是为可移植性而存在，而是用来启用Java Persistence的特定特性——例如，当提交事务时，底层持久化上下文的清除。

你已经多次在Java SE中见过Java Persistence的标准习惯用语。以下代码仍然带有异常处理：

```
EntityManager em = null;
EntityTransaction tx = null;

try {
    em = emf.createEntityManager();
    tx = em.getTransaction();
    tx.begin();

    concludeAuction(em);

    tx.commit();
} catch (RuntimeException ex) {
    try {
        tx.rollback();
    } catch (RuntimeException rbEx) {
        log.error("Couldn't roll back transaction", rbEx);
    }
    throw ex;
} finally {
    em.close();
}
```

这个模式接近于它的Hibernate等价物，隐含着相同的含义：必须手工启动和终止数据库事务，并且必须保证应用程序托管的EntityManager在finaly块中关闭。（虽然我们经常介绍不处理异

常或者包在try/catch块中的代码示例，但这不是可选的。）

JPA抛出的异常是RuntimeException的子类型。任何异常都使得当前的持久化上下文无效，并且一旦抛出异常，就不允许你继续使用EntityManager。因此，我们对Hibernate异常处理所讨论的所有策略也适用于Java Persistence异常处理。此外，下列规则也适用：

- 由EntityManager接口的任何方法抛出的任何异常，都会触发当前事务的自动回滚。
- 由javax.persistence.Query接口的任何方法抛出的任何异常，都会触发当前事务的自动回滚，除了NoResultException和NonUniqueResultException之外。因此，前一个捕捉所有异常的代码示例也为这些异常执行回滚。

注意，JPA不提供细粒度的SQL异常类型。最常见的异常是javax.peristence.PersistenceException。被抛出的所有其他异常都是PersistenceException的子类型，除了NoResultException和NonUniqueResultException之外，应该把它们全部都当作是致命的。然而，可以在JPA抛出的任何异常中调用getCause()，并找出被包装的原生Hibernate异常，包括细粒度的SQL异常类型。

如果你在应用程序服务器内部或者至少提供JTA（对于Hibernate请见我们前面的讨论）的环境中使用Java Persistence，就对编程式的事务划分调用JTA接口。EntityTransaction接口只对本地资源的事务可用。

1．使用Java Persistence的JTA事务

如果Java Persistence代码是在一个可用的JTA环境中部署，并且你想要使用JTA系统事务，就需要调用JTA UserTransaction接口来编程式地控制事务范围：

```
UserTransaction utx = (UserTransaction) new InitialContext()
                        .lookup("java:comp/UserTransaction");
EntityManager em = null;

try {
    utx.begin();

    em = emf.createEntityManager();

    concludeAuction(em);

    utx.commit();
} catch (RuntimeException ex) {
    try {
        utx.rollback();
    } catch (RuntimeException rbEx) {
        log.error("Couldn't roll back transaction", rbEx);
    }
    throw ex;
} finally {
    em.close();
}
```

EntityManager的持久化上下文被界定到JTA事务。被这个EntityManager清除的所有SQL语句，都在通过事务而获取的数据库连接中的一个JTA事务内部执行。当JTA事务提交时，持久化上下文被自动清除和关闭。可以用几个EntityManager在同一个系统事务中访问几个数据库，

就像在原生的Hibernate应用程序中使用几个Session一样。

注意持久化上下文的范围改变了！现在它被界定到了JTA事务，并且一旦提交事务，事务期间原来处于持久化状态的所有对象就立即被视为脱管。

异常处理的规则相当于那些用于本地资源的事务的规则。如果在EJB中使用JTA，别忘了在类上设置@TransactionManagement（TransactionManagementType.BEAN）来启用BMT。

你不会经常通过JTA使用Java Persistence，也没有可用的EJB容器。如果你没有部署一个独立的JTA实现，Java EE 5.0应用程序服务器就将提供这两样东西。你不用编程式的事务划分，而是可能使用EJB的声明特性。

2. Java Persistence和CMT

让我们从前面只用Hibernate的示例中，把ManageAuction EJB会话bean重构到Java Persistence接口。你还让容器注入EntityManager：

```
@Stateless
public class ManageAuctionBean implements ManageAuction {

    @PersistenceContext(unitName = "auctionDB")
    private EntityManager auctionEM;

    @PersistenceContext(unitName = "billingDB")
    private EntityManager billingEM;

    @TransactionAttribute(TransactionAttributeType.REQUIRED)
    public void endAuction(Item item)
                throws AuctionNotValidException {
        concludeAuction(auctionEM, item);
        billAuction(billingEM, item);
    }
    ...
}
```

在concludeAuction()和billAuction()方法内部发生的一切与这个例子仍不相关；假设它们需要用EntityManager来访问数据库。用于endAuction()方法的TransactionAttribute要求所有的数据库访问都发生在一个事务内部。如果endAuction()调用时，没有系统事务是活动的，就会给这个方法启动一个新事务。如果方法返回，且如果事务是为这个方法启动的，它就被提交。每个EntityManager都有一个跨越事务范围、并且当事务结束时被自动清除的持久化上下文。如果方法被调用时没有活动的事务，持久化上下文就有与endAuction()方法相同的范围。

这两个持久化单元都被配置为在JTA上部署，因此两个托管的数据库连接（每个数据库一个），都在同一个事务内部获得，原子性受到应用程序服务器的事务管理器的保护。

你声明endAuction()方法可以抛出AuctionNotValidException。这是你编写的一个定制异常；在终止拍卖之前，检查是否一切都准确无误（已经到达拍卖终止时间、有一个出价等）。这是个checked exception，是java.lang.Exception的子类型。EJB容器把它当成应用程序异常（application exception）处理，如果EJB方法抛出这个异常，则不会触发任何动作。然而，容器认识系统异常（system exception），它默认为所有可能被EJB方法抛出的未检查RuntimeException。

由EJB方法抛出的系统异常强制系统事务的自动回滚。

换句话说，你不需要从Java Persistence操作中捕捉和重新抛出任何系统异常——让容器来处理它们。如果抛出应用程序异常，那么对于如何回滚事务，你有两种选择：第一，可以捕捉它，并手工调用JTA `UserTransaction`，设置它回滚。或者把`@ApplicationException(rollback=true)`注解添加到`AuctionNotValidException`的类——然后容器就知道，你希望每当EJB方法抛出这个应用程序异常时就自动回滚。

现在，你准备在应用程序服务器的内部和外部、用或者不用JTA，以及在EJB和容器管理事务的组合中使用Java Persistence和Hibernate。我们已经讨论了事务原子性的（几乎）所有方面。对于并发运行的事务之间的隔离性，你自然可能仍存疑虑。

10.2　控制并发访问

数据库（和其他的事务系统）试图确保事务隔离性（transaction isolation），这意味着，从每个并发事务的观点来看，似乎没有其他的事务在运行。传统上而言，这已经通过锁（locking）实现了。事务可以在数据库中一个特定的数据项目上放置一把锁，暂时防止通过其他事务访问这个项目。一些现代的数据库（如Oracle和PostgreSQL）通过多版本并发控制（multiversion concurrency control，MVCC）实现事务隔离性，这种多版本并发控制通常被认为是更可伸缩的。我们将讨论假设了锁模型的隔离性；但是我们的大部分论述也适用于多版本并发控制。

数据库如何实现并发控制，这在Hibernate和Java Persistence应用程序中是至关重要的。应用程序继承由数据库管理系统提供的隔离性保证。例如，Hibernate从不锁上内存中的任何东西。如果你认为数据库供应商有多年实现并发控制的经验，就会发现这种方法的好处。另一方面，Hibernate和Java Persistence中的一些特性（要么因为你使用它们，要么按照设计）可以改进远甚于数据库所提供的隔离性保证。

以几个步骤讨论并发控制。我们探讨最底层，研究数据库提供的事务隔离性保证。然后，看一下Hibernate和Java Persistence特性对于应用程序级的悲观和乐观并发控制，以及Hibernate可以提供哪些其他的隔离性保证。

10.2.1　理解数据库级并发

作为Hibernate应用程序的开发人员，你的任务是理解数据库的能力，以及如果特定的场景（或者按照数据完整性需求）需要，如何改变数据库的隔离性行为。让我们退一步来看。如果我们正在讨论隔离性，你可能假设两种情况，即隔离或者不隔离；现实世界中没有灰色区域。说到数据库事务，完全隔离性的代价就很高了。有几个隔离性级别（isolation level），它们一般削弱完全的隔离性，但提升了系统的性能和可伸缩性。

1．事务隔离性问题

首先，来看一下削弱完全事务隔离性时可能出现的几种现象。ANSI SQL标准根据数据库管理系统允许的现象定义了标准的事务隔离性级别：

如果两个事务都更新一个行，然后第二个事务异常终止，就会发生丢失更新（lost update），

导致两处变化都丢失。这发生在没有实现锁的系统中。此时没有隔离并发事务。如图10-2所示。

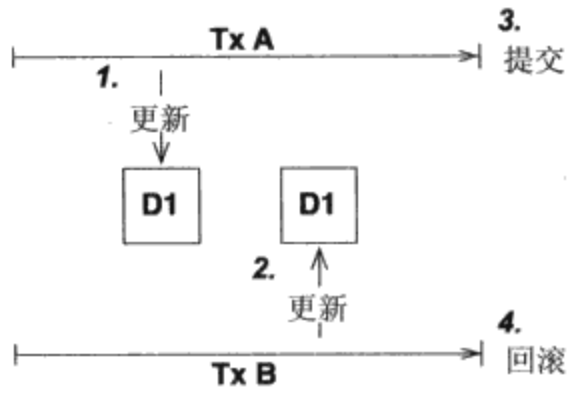


图10-2 丢失更新：两个事务更新没有加锁的同一数据

如果一个事务读取由另一个还没有被提交的事务进行的改变，就发生脏读取（dirty read）。这很危险，因为由其他事务进行的改变随后可能回滚，并且第一个事务可能编写无效的数据，如图10-3所示。

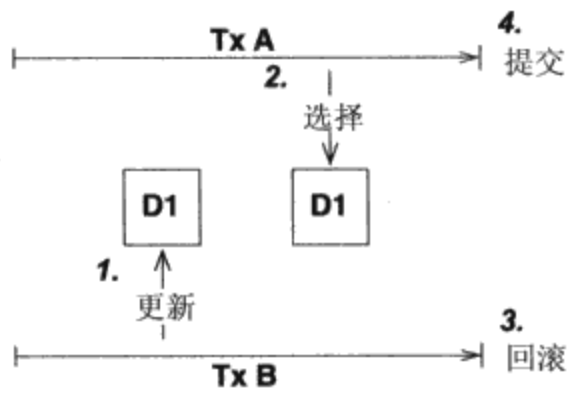


图10-3 脏读取：事务A读取没有被提交的数据

如果一个事务读取一个行两次，并且每次读取不同的状态，就会发生不可重复读取（unrepeatable read）。例如，另一个事务可能已经写到这个行，并已在两次读取之间提交，如图10-4所示。

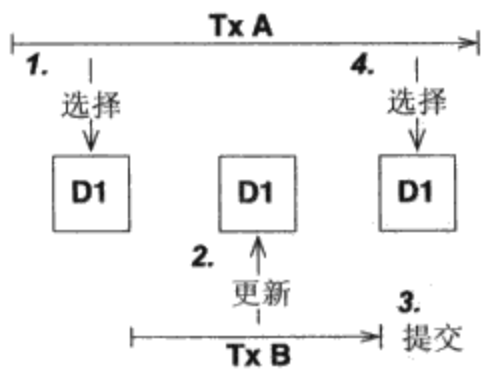


图10-4 不可重复读取：事务A执行不可重复读取两次

不可重复读取的一个特殊案例是二次丢失更新问题（second lost updates problem）。想象两个并发事务都读取一个行：一个写到行并提交，然后第二个也写到行并提交。由第一个事务所做的改变丢失了。如果考虑需要几个数据库事务来完成的应用程序对话，这个问题就特别值得关注。

我们将在稍后更深入地探讨这种情况。

幻读（phantom read）发生在一个事务执行一个查询两次，并且第二个结果集包括第一个结果集中不可见的行，或者包括已经删除的行时。（不需要是完全相同的查询。）这种情形是由另一个事务在两次查询执行之间插入或者删除行造成的，如图10-5所示。

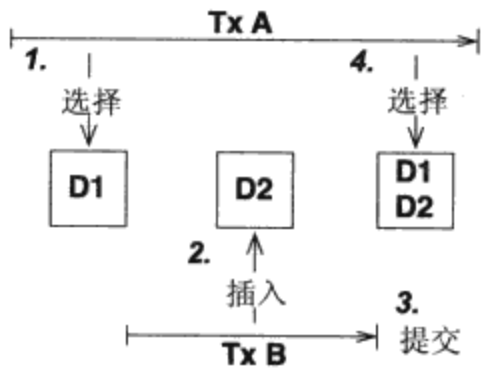


图10-5　幻读：事务A在第二次选择中读取新数据

既然已经了解所有可能发生的不好的情况，我们就可以定义事务隔离性级别了，并看看它们阻止了哪些问题。

2. ANSI事务隔离性级别

标准的隔离性级别由ANSI SQL标准定义，但是它们不是SQL数据库特有的。JTA也定义了完全相同的隔离性级别，稍后你将用这些级别声明想要的事务隔离性。隔离性级别的增加带来了更高成本以及严重的性能退化和可伸缩性：

- 允许脏读取但不允许丢失更新的系统，据说要在读取未提交（read uncommitted）的隔离性中操作。如果一个未提交事务已经写到一个行，另一个事务就不可能再写到这个行。但任何事务都可以读取任何行。这个隔离性级别可以在数据库管理系统中通过专门的写锁来实现。
- 允许不可重复读取但不允许脏读取的系统，据说要实现读取提交（read committed）的事务隔离性。这可以用共享的读锁和专门的写锁来实现。读取事务不会阻塞其他事务访问行。但是未提交的写事务阻塞了所有其他的事务访问该行。
- 在可重复读取（repeatable read）隔离性模式中操作的系统既不允许不可重复读取，也不允许脏读取。幻读可能发生。读取事务阻塞写事务（但不阻塞其他的读取事务），并且写事务阻塞所有其他的事务。
- 可序列化（serializable）提供最严格的事务隔离性。这个隔离性级别模拟连续的事务执行，好像事务是连续地一个接一个地执行，而不是并发地执行。序列化不可能只用低级锁实现。一定有一些其他的机制，防止新插入的行变成对于已经执行会返回行的查询的事务可见。

锁系统在DBMS中具体如何实现很不相同；每个供应商都有不同的策略。你应该查阅DBMS文档，找出更多有关锁系统的信息，如何逐步加强锁（例如从低级别到页面，到整张表），以及每个隔离性级别对于系统性能和可伸缩性有什么影响。

知道所有这些技术术语如何定义，这很好，但是它如何帮助你给应用程序选择隔离性级别呢？

3. 选择隔离性级别

开发人员（包括我们自己）经常不确定要在一个产品应用程序中使用哪种事务隔离性级别。隔离性太强会损害高并发应用程序的可伸缩性。隔离性不足则可能在应用程序中导致费解的、不可重现的bug，直到系统过载运行时才会发现。

注意，我们在接下来的阐述中所指的乐观锁（optimistic locking）（利用版本），是本章稍后要解释的一个概念。你可能想要跳过这一节，并且当要在应用程序中决定隔离性级别时再回来。毕竟，选择正确的隔离性级别很大程度上取决于特定的场景。把以下讨论当作建议来读，不要把它们当作金科玉律。

在数据库的事务语义方面，Hibernate努力尽可能地透明。不过，高速缓存和乐观锁影响着这些语义。在Hibernate应用程序中要选择什么有意义的数据库隔离性级别呢？

首先，消除读取未提交隔离性级别。在不同的事务中使用一个未提交的事务变化是很危险的。一个事务的回滚或者失败将影响其他的并发事务。第一个事务的回滚可能战胜其他的事务，或者甚至可能导致它们使数据库处于一种错误的状态中。甚至由一个终止回滚的事务所做的改变也可能在任何地方被提交，因为它们可以读取，然后由另一个成功的事务传播！

其次，大多数应用程序不需要可序列化隔离性（幻读通常不成问题），并且这个隔离性级别往往难以伸缩。现有的应用程序很少在产品中使用序列化隔离性，但在某些情况下，有效地强制一个操作序列化地执行相当依赖于悲观锁（请见接下来的几节内容）。

这样就把选择读取提交还是可重复读取留给你来决定了。我们先考虑可重复读取。如果所有的数据访问都在单个原子的数据库事务中执行，这个隔离性级别就消除了一个事务可能覆盖由另一个并发事务所做变化（第二个丢失更新问题）的可能性。事务持有的读锁防止了并发事务可能希望获得的任何写锁。这是一个重要的问题，但是启用可重复读取并不是唯一的解决办法。

假设你正使用版本化（versioned）的数据，这是Hibernate可以自动完成的东西。（必需的）持久化上下文高速缓存和版本控制的组合已经提供了可重复读取隔离性的大部分优良特性。特别是，版本控制防止了二次丢失更新问题，并且持久化上下文高速缓存也确保了由一个事务加载的持久化实例状态与由其他事务所做的变化隔离开来。因此，如果你使用版本化的数据，那么对于所有数据库事务来说，读取提交的隔离性是可以接受的。

可重复读取给查询结果集（只针对数据库事务的持续期间）提供了更多的可复制性；但是因为幻读仍然可能，这似乎没有多大价值。可以在Hibernate中给一个特定的事务和数据块显式地获得可重复读取的保证（通过悲观锁）。

设置事务隔离性级别允许你给所有的数据库事务选择一个好的默认锁策略。如何设置隔离性级别呢？

4. 设置隔离性级别

与数据库的每一个JDBC连接都处于DBMS的默认隔离性级别——通常是读取提交或者可重复读取。可以在DBMS配置中改变这个默认。还可以在应用程序端给JDBC连接设置事务隔离性，

通过一个Hibernate配置选项：

```
hibernate.connection.isolation = 4
```

Hibernate在启动事务之前，给每一个从连接池中获得的JDBC连接设置这个隔离性级别。对于这个选项有意义的值如下（你也可能发现它们为java.sql.Connection中的常量）：

- 1——读取未提交隔离性。
- 2——读取提交隔离性。
- 3——可重复读取隔离性。
- 4——可序列化隔离性。

注意，Hibernate永远不会改变在托管环境中从应用程序服务器提供的数据库连接中获得的连接隔离性级别！可以利用应用程序服务器的配置改变默认的隔离性级别。（如果使用独立的JTA实现也一样。）

如你所见，设置隔离性级别是影响所有连接和事务的一个全局选项。给特定的事务指定一个更加限制的锁经常很有用。Hibernate和Java Persistence依赖乐观的并发控制，并且两者都允许你通过版本检查和悲观锁，获得额外的锁保证。

10.2.2　乐观并发控制

乐观的方法始终假设一切都会很好，并且很少有冲突的数据修改。在编写数据时，乐观并发控制只在工作单元结束时才出现错误。多用户的应用程序通常默认为使用读取提交隔离性级别的乐观并发控制和数据库连接。只有适当的时候（例如，当需要可重复读取的时候）才获得额外的隔离性保证；这种方法保证了最佳的性能和可伸缩性。

1．理解乐观策略

为了理解乐观并发控制，想象两个事务从数据库中读取一个特定的对象，并且两者都对它进行修改。由于数据库连接的读取提交隔离性级别，因此没有任何一个事务会遇到任何脏读取。然而，读取仍然是不可重复的，并且更新还是可能丢失。这是当你在考虑对话的时候要面对的问题，从用户的观点来看，这些是原子的事务。请见图10-6。

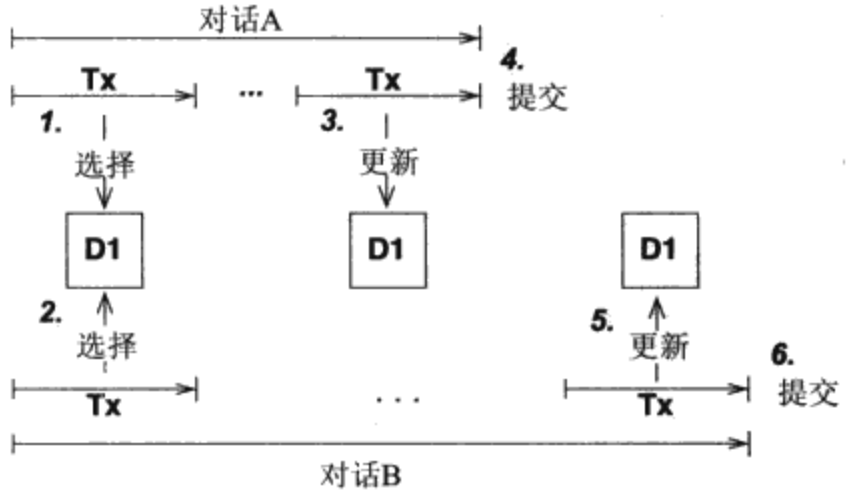


图10-6　对话B覆盖对对话A所做的改变

假设两个用户同时选择同一块代码。对话A中的用户先提交了变化，并且对话终止于第二个事务的成功提交。过了一会儿（可能只是一秒钟），对话B中的用户提交了变化。第二个事务也成功提交。在对话A中所做的改变已经丢失，并且（可能更糟的是）对话B中提交的数据修改可能已经基于失效的信息。

对于如何处理对话中这些第二个事务中的丢失更新，你有3种选择：

- 最晚提交生效（last commit wins）——两个事务提交都成功，且第二次提交覆盖第一个的变化。没有显示错误消息。
- 最先提交生效（first commit wins）——对话A的事务被提交，并且在对话B中提交事务的用户得到一条错误消息。用户必须获取新数据来重启对话，并再次利用没有失效的数据完成对话的所有步骤。
- 合并冲突更新（merge conflicting updates）——第一个修改被提交，并且对话B中的事务在提交时终止，带有一条错误消息。但是失败的对话B用户可以选择性地应用变化，而不是再次在对话中完成所有工作。

如果你没有启用乐观并发控制（默认情况为未启用），应用程序就会用最晚提交生效策略运行。在实践中，丢失更新的这个问题使得许多应用程序的用户很沮丧，因为他们可能发现他们的所有工作都丢失了，而没有收到任何错误消息。

很显然，最先提交生效更有吸引力。如果对话B的应用程序的用户提交，他就获得这样一条错误消息：有人已经对你要提交的数据提交了修改。你已经使用了失效数据。请用新数据重启对话。（Somebody already committed modifications to the data you're about to commit. You've been working with stale data. Please restart the conversation with fresh data。）设计和编写生成这条错误消息的应用程序，并引导用户重新开始对话，这就是你的责任了。Hibernate和Java Persistence用自动乐观锁协助你，以便每当事务试图提交在数据库中带有冲突的被更新状态的对象时，就会得到一个异常。

合并冲突的变化，是最先提交生效的一种变形。不显示始终强制用户返回的错误消息，而是提供一个对话框，允许用户手工合并冲突的变化。这是最好的策略，因为没有工作丢失，应用程序的用户也不会因为乐观并发失败而受挫。然而，对于开发人员来说，提供一个对话框来合并变化比显示一条错误消息并强制用户重复所有的工作来得更加费时。是否使用这一策略，由你自己决定。

乐观并发控制可以用多种方法实现。Hibernate使用自动的版本控制。

2. 在Hibernate中启用版本控制

Hibernate提供自动的版本控制。每个实体实例都有一个版本，它可以是一个数字或者一个时间戳。当对象被修改时，Hibernate就增加它的版本号，自动比较版本，如果侦测到冲突就抛出异常。因此，你给所有持久化的实体类都添加这个版本属性，来启用乐观锁：

```
public class Item {
    ...
    private int version;
    ...
}
```

也可以添加获取方法；但是不许应用程序修改版本号。XML格式的<version>属性映射必须立即放在标识符属性映射之后：

```
<class name="Item" table="ITEM">
    <id .../>

    <version name="version" access="field" column="OBJ_VERSION"/>

    ...
</class>
```

版本号只是一个计数值——它没有任何有用的语义值。实体表上额外的列为Hibernate应用程序所用。记住，所有访问相同数据库的其他应用程序也可以（并且或许应该）实现乐观版本控制，并利用相同的版本列。有时候时间戳是首选（或者已经存在）：

```
public class Item {
    ...
    private Date lastUpdated;
    ...
}

<class name="Item" table="ITEM">

    <id .../>

    <timestamp name="lastUpdated"
               access="field"
               column="LAST_UPDATED"/>
    ...
</class>
```

理论上来说，时间戳更不安全一点，因为两个并发的事务可能都在同一毫秒点上加载和更新同一件货品；但在实践中不会发生这种情况，因为JVM通常没有精确到毫秒（你应该查阅JVM和操作系统文档所确保的精确度）。

此外，从JVM处获取的当前时间在集群环境（clustered environment）下并不一定安全，该环境中的节点可能不与时间同步。可以转换为在<timestamp>映射中利用source="db"属性从数据库机器中获取当前的时间。并非所有的Hibernate SQL方言都支持这个属性（检查所配置的方言的源代码），每一次增加版本都始终会有命中数据库的过载。

我们建议新项目依赖包含版本号的版本，而不是时间戳。

一旦你把<version>或者<timestamp>属性添加到持久化类映射，就启用了包含版本的乐观锁。没有其他的转换。

Hibernate如何利用版本发现冲突？

3.* 版本控制的自动管理

涉及目前被版本控制的Item对象的每一个DML操作都包括版本检查。例如，假设在一个工作单元中，你从版本为1的数据库中加载一个Item。然后修改它的其中一个值类型属性，例如Item的价格。当持久化上下文被清除时，Hibernate侦测到修改，并把Item的版本增加到2。然后执行SQL UPDATE使这一修改在数据库中永久化：

```
update ITEM set INITIAL_PRICE='12.99', OBJ_VERSION=2
    where ITEM_ID=123 and OBJ_VERSION=1
```

如果另一个并发的工作单元更新和提交了同一个行，OBJ_VERSION列就不再包含值1，行也不会被更新。Hibernate检查由JDBC驱动器返回这个语句所更新的行数——在这个例子中，被更新的行数为0——并抛出StaleObjectStateException。加载Item时呈现的状态，清除时不再在数据库中呈现；因而，你正在使用失效的数据，必须通知应用程序的用户。可以捕捉这个异常，并显示一条错误消息，或者显示帮助用户给应用程序重启对话的一个对话框。

什么样的修改触发实体版本的增加？每当实体实例脏时，Hibernate就增加版本号（或者时间戳）。这包括实体的所有脏的值类型属性，无论它们是单值、组件还是集合。考虑User和BillingDetails之间的关系，这是个一对多的实体关联：如果CreditCard修改了，相关的User版本并没有增加。如果你从账单细节的集合中添加或者删除CreditCard（或者BankAccount），User的版本就增加了。

如果你想要禁用对特定值类型属性或者集合的自动增加，就用optimistic-lock="false"属性映射它。inverse属性在这里没有什么区别。甚至如果元素从反向集合中被添加或者移除，反向集合的所有者的版本也会被更新。

如你所见，Hibernate使得对于乐观并发控制管理版本变得难以置信地轻松。如果你正在使用遗留数据库Schema或者现有的Java类，也许不可能引入版本或者时间戳和列。Hibernate提供了另一种可选的策略。

4．没有版本号或者时间戳的版本控制

如果你没有版本或者时间戳列，Hibernate仍然能够执行自动的版本控制，但是只对在同一个持久化上下文中获取和修改的对象（即相同的Session）。如果你需要乐观锁用于通过脱管对象实现的对话，则必须使用通过脱管对象传输的版本号或者时间戳。

这种可以选择的版本控制实现方法，在获取对象（或者最后一次清除持久化上下文）时，把当前的数据库状态与没有被修改的持久化属性值进行核对。可以在类映射中通过设置optimistic-lock属性来启用这项功能：

```
<class name="Item" table="ITEM" optimistic-lock="all">
    <id .../>
    ...
</class>
```

下列SQL现在被执行，用来清除Item实例的修改：

```
update ITEM set ITEM_PRICE='12.99'
where ITEM_ID=123
  and ITEM_PRICE='9.99'
  and ITEM_DESCRIPTION="An Item"
  and ...
  and SELLER_ID=45
```

Hibernate在SQL语句的WHERE子句中，列出了所有列和它们最后知道的非失效值。如果任何并发的事务已经修改了这些值中的任何一个，或者甚至删除了行，这个语句就会再次返回被更新的行数为0。然后Hibernate抛出一个StaleObjectStateException。

另一种方法是，如果设置optimistic-lock="dirty"，Hibernate只包括限制中被修改的属性（在这个例子中，只有ITEM_PRICE）。这意味着两个工作单元可以同时修改同一个对象，并且只有当两者修改同一个值类型属性（或者外键值）时才会侦测到冲突。在大多数情况下，这对于业务实体来说并不是一种好策略。想象有两个人同时修改一件拍卖货品：一个改变价格，另一个改变描述。即使这些修改在最低级别（数据库行）没有冲突，从业务逻辑观点看它们也可能发生冲突。如果货品的描述完全改变了，还可以改变它的价格吗？如果你想要使用这个策略，还必须在实体的类映射上启用dynamic-update="true"，Hibernate无法在启动时给这些动态的UPDATE语句生成SQL。

不建议在新应用程序中定义没有版本或者时间戳列的版本控制；它更慢、更复杂，如果你正在使用脱管对象，则它不会生效。

Java Persistence应用程序中的乐观并发控制与Hibernate中的几乎如出一辙。

5. 用Java Persistence版本控制

Java Persistence规范假设并发数据访问通过版本控制被乐观处理。为了给一个特定的实体启用自动版本控制，需要添加一个版本属性或者字段：

```
@Entity
public class Item {
    ...
    @Version
    @Column(name = "OBJ_VERSION")
    private int version;
    ...
}
```

同样地，可以公开一个获取方法，但不能允许应用程序修改版本值。在Hibernate中，实体的版本属性可以是任何数字类型，包括基本类型，或者Date或者Calendar类型。JPA规范只把int、Integer、short、Short、long、Long和java.sql.Timestamp当作可移植的版本类型。

由于JPA标准没有涵盖无版本属性的乐观版本控制，因此需要Hibernate扩展，通过对比新旧状态来启用版本控制：

```
@Entity
@org.hibernate.annotations.Entity(
  optimisticLock = org.hibernate.annotations.OptimisticLockType.ALL
)
public class Item {
    ...
}
```

如果只是希望在版本检查期间比较被修改的属性，也可以转换到OptimisticLockType.DIRTY。然后你还需要设置dynamicUpdate属性为true。

Java Persistence没有对哪个实体实例修改应该触发版本增加标准化。如果你用Hibernate作为JPA提供程序，默认是一样的——每一个值类型的属性修改（包括集合元素的添加和移除）都触发版本增加。在编写本书之时，还没有在特定的属性和集合上禁用版本增加的Hibernate注解，但是已经存在一项对@OptimisticLock(excluded=true)的特性请求。你的Hibernate Annotations

版本或许包括了这个选项。

Hibernate EntityManager，像任何其他Java Persistence提供程序一样，当侦测到冲突版本时，就抛出javax.persistence.OptimisticLockException。这相当于Hibernate中原生的StaleObjectStateException，因此应该进行相应处理。

我们现在已经涵盖了数据库连接的基础隔离性级别，结论是你通常应该依赖来自数据库的读取提交保证。Hibernate和Java Persistence中的自动版本控制，在两个并发事务试图在同一块代码中提交修改时，防止了丢失更新。为了处理非可重复读取，你需要额外的隔离性保证。

10.2.3 获得额外的隔离性保证

有几种方法防止不可重复读取，并升级到一个更高的隔离性级别。

1. 显式的悲观锁

已经讨论了把所有的数据库连接转换到一个比读取提交更高的隔离性级别，但我们的结论是，当关注应用程序的可伸缩性时，这则是一项糟糕的默认。你需要更好、仅用于一个特定的工作单元的隔离性保证。还要记住，持久化上下文高速缓存为处于持久化状态的实体实例提供可重复读取。然而，这并非永远都是足够的。

例如，对标量查询（scalar query）可能需要可重复读取：

```
Session session = sessionFactory.openSession();
Transaction tx = session.beginTransaction();

Item i = (Item) session.get(Item.class, 123);

String description = (String)
        session.createQuery("select i.description from Item i" +
                            " where i.id = :itemid")
               .setParameter("itemid", i.getId() )
               .uniqueResult();

tx.commit();
session.close();
```

这个工作单元执行两次读取。第一次通过标识符获取实体实例。第二次读取标量查询，再次加载已经加载的Item实体的描述。在这个工作单元中有一个小窗口，在那里，并发运行的事务可以在两次读取之间提供一个更新过的货品描述。然后第二次读取返回这个提交数据，且变量description有一个与属性i.getDescription()不同的值。

这个例子进行过简化，但仍然足以说明：如果数据库事务隔离性级别是读取提交，那么混有实体和标量读取的工作单元有多么容易受到非可重复读取的影响。

不是把所有的数据库事务转换为一个更高的、不可伸缩的隔离性级别，而是在必要时，在Hibernate Session中使用lock()方法获得更强的隔离性保证：

```
Session session = sessionFactory.openSession();
Transaction tx = session.beginTransaction();

Item i = (Item) session.get(Item.class, 123);

session.lock(i, LockMode.UPGRADE);
```

```
String description = (String)
        session.createQuery("select i.description from Item i" +
                            " where i.id = :itemid")
                .setParameter("itemid", i.getId() )
                .uniqueResult();

tx.commit();
session.close();
```

使用LockMode.UPGRADE，给表示Item实例的（多）行，促成了在数据库中保存的悲观锁。现在没有并发事务可以在相同数据中获得锁——也就是说，没有并发事务可以在你的两次读取之间修改数据。这段代码可以被缩短成如下：

```
Session session = sessionFactory.openSession();
Transaction tx = session.beginTransaction();

Item i = (Item) session.get(Item.class, 123, LockMode.UPGRADE);
...
```

LockMode.UPGRADE导致一个SQL SELECT ... FOR UPDATE或者类似的东西，具体取决于数据库方言。一种变形LockMode.UPGRADE_NOWAIT，添加了一个允许查询立即失败的子句。如果没有这个子句，当无法获得锁时（可能由于一个并发事务已经有锁），数据库通常会等待。等待的持续时间取决于数据库，就像实际的SQL子句一样。

常见问题　可以使用长悲观锁吗？在Hibernate中，悲观锁的持续时间是单个数据库事务。这意味着你无法使用专门的锁，来阻塞比单个数据库事务更长的并发访问。我们认为这是好的一面，因为对于比如整个会话的持续时间来说，唯一的解决方案将是在内存（或者数据库中所谓的锁定表，lock table）中保存一个非常昂贵的锁。这种锁有时也称作离线（offline）锁。这通常是个性能瓶颈；每个数据访问都要对一个同步锁管理器进行额外的锁检查。然而，乐观锁是最完美的并发控制策略，并且在长运行对话中执行得很好。根据你的冲突解析（conflict-resolution）选项（即如果你有足够的时间实现合并变化），你应用程序的用户对此将会像对被阻塞的并发访问一样满意。他们也可能感激当其他人在看相同数据时，自己没有被锁在特定的屏幕之外。

Java Persistence出于同样的目的定义了LockModeType.READ，且EntityManager也有一个lock()方法。规范没有要求未被版本控制的实体支持这种锁模式；但Hibernate在所有的实体中都支持它，因为它在数据库中默认为悲观锁。

2. Hibernate锁模式

Hibernate支持下列其他LockMode：

- LockMode.NONE——别到数据库中去，除非对象不处于任何高速缓存中。
- LockMode.READ——绕过所有高速缓存，并执行版本检查，来验证内存中的对象是否与当前数据库中存在的版本相同。
- LockMode.UPGRADE——绕过所有高速缓存，做一个版本检查（如果适用），如果支持的话，就获得数据库级的悲观升级锁。相当于Java Persistence中的LockModeType.READ。如

果数据库方言不支持SELECT ... FOR UPDATE选项，这个模式就透明地退回到LockMode.READ。

- LockMode.UPGRADE_NOWAIT——与UPGRADE相同，但如果支持的话，就使用SELECT ... FOR UPDATE NOWAIT。它禁用了等待并发锁释放，因而如果无法获得锁，就立即抛出锁异常。如果数据库SQL方言不支持NOWAIT选项，这个模式就透明地退回到LockMode.UPGRADE。
- LockMode.FORCE——在数据库中强制增加对象的版本，来表明它已经被当前事务修改。相当于Java Persistence中的LockModeType.WRITE。
- LockMode.WRITE——当Hibernate已经在当前事务中写到一个行时，就自动获得它。（这是一种内部模式；你不能在应用程序中指定它。）

默认情况下，load()和get()使用LockMode.NONE。LockMode.READ对session.lock()和脱管对象最有用。这里有个例子：

```
Item item = ... ;
Bid bid = new Bid();
item.addBid(bid);
...
Transaction tx = session.beginTransaction();
session.lock(item, LockMode.READ);
tx.commit();
```

这段代码在通过级联（假设从Item到Bid的关联启用了级联）保存新Bid之前，在脱管的Item实例上执行版本检查，验证该数据库行在获取之后没有被另一个事务更新。

（注意，EntityManager.lock()不重附指定的实体实例——它只对已经处于托管持久化状态的实例有效。）

Hibernate LockMode.FORCE和Java Persistence中的LockModeType.WRITE有着不同的用途。如果默认不增加版本，就利用它们强制版本更新。

3. 强制增加版本

如果通过版本控制启用乐观锁，Hibernate会自动增加被修改实体实例的版本。然而，有时你想手工增加实体实例的版本，因为Hibernate不会把你的改变当成一个应该触发版本增加的修改。

想象你修改了CreditCard所有者的名称：

```
Session session = getSessionFactory().openSession();
Transaction tx = session.beginTransaction();

User u = (User) session.get(User.class, 123);

u.getDefaultBillingDetails().setOwner("John Doe");

tx.commit();
session.close();
```

当这个Session被清除时，被修改的BillingDetails实例（我们假设是一张信用卡）的版本通过Hibernate自动增加了。这可能并不是你想要的东西——你可能也想增加所有者（User实例）的版本。

用LockMode.FORCE调用lock()，增加一个实体实例的版本：

```
Session session = getSessionFactory().openSession();
Transaction tx = session.beginTransaction();

User u = (User) session.get(User.class, 123);

session.lock(u, LockMode.FORCE);

u.getDefaultBillingDetails().setOwner("John Doe");

tx.commit();
session.close();
```

现在，任何使用相同User行的并发工作单元都知道这个数据被修改了，即使只有被你认为是整个聚合的一部分的其中一个值被修改了。这种技术在许多情况下都有用，例如当你修改一个对象，并且想要增加聚合的根对象版本时。另一个例子是对一件拍卖货品出价金额的修改（如果这些金额是不可变的）：利用一个显式的版本增加，可以指出这件货品已经被修改，即使它的值类型属性或者集合都没有发生改变。利用Java Persistence的同等调用是em.lock(o,LockModeType.WRITE)。

现在，你具备了编写更复杂工作单元和创建对话的所有知识。但是，我们需要提及事务的最后一个方面，因为它在使用JPA的更复杂对话中变得必不可少。你必须理解自动提交如何工作，以及在实践中非事务数据访问意味着什么。

10.3　非事务数据访问

许多数据库管理员在每个新的数据库连接上默认启用所谓的自动提交模式。自动提交模式对于SQL的特殊执行很有用。

想象你用SQL控制台连接到数据库，并且运行几个查询，甚至也可能更新和删除行。这种交互式的数据访问是特殊的；大多数时候，你并不具备考虑工作单元的计划或者一系列语句。数据库连接中默认的自动提交模式对于这种数据访问是至臻完美的——毕竟，你并不想给所编写和执行的每个SQL语句都输入begin a transaction和end a transaction。在自动提交模式中，一个（短）数据库事务给对于发送到数据库的每个SQL语句进行启动和终止。你正在有效地进行非事务地工作，因为对于使用SQL控制台的会话来说，并没有原子性或者隔离性保证。（唯一的保证是单个SQL语句具有原子性。）

按照定义，应用程序始终执行计划好的一系列语句。因而你始终创建事务范围，把语句组合到原子单元中去，这似乎很合理。因此，自动提交模式在应用程序中没有用武之地。

10.3.1　揭开自动提交的神秘面纱

许多开发人员经常出于一些含糊的、不确切的理由，仍然喜欢使用自动提交模式。如果你想（或者必须）了解其中的原因的话，就让我们在介绍如何非事务地访问数据之前先揭开这其中的一部分理由吧：

- 许多应用程序的开发人员认为，他们可以在事务外部与数据库对话。这显然是不可能的；没有任何SQL语句可以被发送到数据库事务外部的数据库。术语非事务的数据访问意味着

没有显式的事务范围，没有系统事务，并且数据访问的行为处于自动提交模式。这并不意味着没有涉及实质性的数据库事务。

- 如果你的目标是利用自动提交模式改进应用程序的性能，就应该重新考虑许多小事务的含义。给每一个SQL语句启动和终止数据库事务涉及了重大的过载，它可能降低应用程序的性能。
- 如果你的目标是利用自动提交模式改进应用程序的可伸缩性，就重新考虑：对于每一个SQL语句来说，用更长运行的数据库事务代替许多小事务，可以保持更长时间的数据库锁，也可能不进行伸缩。但是，由于Hibernate持久化上下文和DML的迟写，数据库中所有写锁已经持续了很短的时间。根据你启用的隔离性级别，读锁的成本可能可以忽略。或者，你可以使用一个不需要读锁（Oracle、PostgreSQL、Informix、Firebird）的包含多版本并发的DBMS，因为默认情况下读取操作永远不会被阻塞。
- 因为你正在进行非事务地工作，不仅真的放弃了一组SQL语句的任何事务原子性，而且如果数据被同时修改，还减弱了隔离性保证。基于读锁不可能有自动提交模式的可重复读取。（持久化上下文高速缓存在此处自然很有帮助。）

在应用程序中引入非事务的数据访问时，要考虑更多的问题。我们已经注意到，引入一种新的事务类型，称作只读事务（read-only transaction），可以明显使应用程序的任何未来修改变得复杂起来。如果你引入非事务的操作也一样。

然后你在应用程序中就有3种不同的数据访问了：普通事务、只读事务以及现在又有不包含保证的非事务。想象你必须引入一项操作，把数据写到假定只读取数据的工作单元中。想象你必须把非事务的操作识别为事务操作。

我们的建议是不在应用程序中使用自动提交模式，并且只在有明显的性能好处，或者很可能要在未来改变代码时，才应用只读的事务。始终更喜欢一般的ACID事务，把数据访问操作组合起来，无论你是读还是写数据。

话虽这么说，Hibernate和Java Persistence还是允许非事务的数据访问。事实上，如果你想实现原子的长运行对话，EJB 3.0规范强制你非事务地访问数据。第11章将探讨这个主题。现在要深入到简单的Hibernate应用程序中自动提交模式所带来的结果。（注意，不要管我们负面的评论，自动提交模式还是有一些好的使用案例。依据我们的经验，自动提交经常由于错误的原因而被启用，我们本想先清除这些错误的观念。）

10.3.2 使用 Hibernate 非事务地工作

看看下列代码，它访问没有事务范围的数据库：

```
Session session = sessionFactory.openSession();

session.get(Item.class, 1231);

session.close();
```

默认情况下，在包含JDBC配置的Java SE环境中，如果执行这个片段，将发生：

(1) 打开一个新Session。此时它没有获得数据库连接。

(2) 调用get()触发一个SQL SELECT。现在Session从连接池获得了JDBC Connection。Hibernate立即默认在这个连接中用setAutoCommit(false)关闭自动提交的模式。这样有效地启动了一个JDBC事务！

(3) SELECT在这个JDBC事务内部执行。Session关闭，且连接被返回到池，并由Hibernate释放——Hibernate在JDBC Connection中调用close()。对于没有被提交的事务，发生了什么事？

这个问题的答案是：不一定！JDBC规范没有提到在一个连接上调用close()时任何未处理的事务的内容。会发生什么取决于供应商如何实现规范。例如，利用Oracle JDBC驱动器，调用close()提交了事务！大多数其他的JDBC供应商走健全的路线，并在JDBC Connection对象被关闭，且资源返回到池中时，回滚任何未处理的事务。

显然，这对于你已经执行的SELECT来说不成问题，但是看看这个变形：

```
Session session = getSessionFactory().openSession();

Long generatedId = session.save(item);

session.close();
```

这段代码促成了INSERT语句，该语句在一个从不被提交或者回滚的事务内部执行。在Oracle中，这段代码永久地插入数据；在其他数据库中，则可能不是。(这种情形稍微更复杂一些：INSERT只有在标识符生成器需要它时才执行。例如，标识符值可以从一个没有INSERT的sequence中获得。然后，持久化实体被列队等待，直到清除时插入——这在这段代码中永远不会发生。identity策略对于要生成的值，需要一个立即的INSERT。)

我们甚至还没有接触到自动提交模式，但是已经强调了一个问题，它可能出现在你试图不设置显式的事务范围进行工作时。假设你仍然认为不用事务划分进行工作是个好主意，并且想要一般的自动提交行为。首先，必须在Hibernate配置中告诉Hibernate允许自动提交的JDBC连接：

```
<property name="connection.autocommit">true</property>
```

利用这个设置，当从连接池获得JDBC连接时，Hibernate不再关闭自动提交——如果连接还没有处于该模式，就启用自动提交。现在前一个代码示例可以预期地生效了，并且JDBC驱动器把被发送到数据库的每一个SQL语句（包含我们前面列出过的那些含义）都包在一个简短的事务中。

在Hibernate中，你在什么情况下要启用自动提交模式，以便可以使用不用手工启动和终止事务的Session呢？受益于自动提交模式的系统是那些要求按需（延迟）加载数据的系统，在一个特定的Session和持久化上下文中，但是在这些系统中很难把所有可能触发按需数据获取的代码都包在事务范围中。这通常不是遵循我们在第16章中讨论的设计模式的Web应用程序中的案例。另一方面，通过Hibernate访问数据库层的桌面应用程序经常要求按需加载，没有显式的事务范围。例如，如果你在Java Swing树视图上双击一个节点，这个节点的所有子节点都必须从数据库中被加载。你最好必须把这个事件手工包在一个事务中；自动提交模式是一种更为方便的解决方案。(注意我们不是在建议按需打开和关闭Session!)

10.3.3 使用 JTA 的可选事务

前面着重讨论了自动提交模式，以及利用非托管的JDBC连接（在这里，Hibernate管理连接池）的应用程序中的非事务数据访问。现在想象你想要在Java EE环境中使用Hibernate，包含JTA，可能也有CMT。connection.autocommit配置选项在这个环境中不起作用。是否使用自动提交，取决于你的事务程序集。

想象你有一个EJB会话bean，它把特定的方法标记为非事务：

```
@Stateless
public class ItemFinder {

    @TransactionAttribute(TransactionAttributeType.NOT_SUPPORTED)
    public Item findItemById(Long id) {
        Session s = getSessionFactory().openSession();
        Item item = (Item) s.get(Item.class, id);
        s.close();
        return item;
    }
}
```

findItemById()方法生成一个返回Item实例的立即的SQL SELECT。因为该方法被标记为不支持事务上下文，所以没有为这项操作启动任何事务，并且任何现有的事务上下文在这个方法的持续时间内被暂停。SELECT在自动提交模式中被有效地执行。（自动提交的JDBC连接被内部分配用来服务于这个Session。）

最后，你需要知道当没有事务在处理时，Session的默认FlushMode改变了。默认的行为FlushMode.AUTO，导致在每个HQL、SQL或者Criteria查询之前同步。当然，这样不好，因为除了为查询执行SELECT外，DML UPDATE、INSERT和DELETE操作也执行。因为你正在自动提交模式下工作，这些修改是永久的。当你在事务范围外部使用Session时，Hibernate通过禁用自动清除防止了这一点。然后你不得不期待查询可以返回失效数据或者与当前Session中数据状态冲突的数据——当选择FlushMode.MANUAL时，必须有效地处理相同的问题。

我们将在第11章的对话讨论中，回到非事务数据访问的话题。应该把自动提交行为当作一项特性（你可能在利用Java Persistence或者EJB的对话中使用它），并且把所有数据访问事件都包在编程式的事务范围中会很困难（例如在一个桌面应用程序中）。在大多数的其他案例中，自动提交导致系统难以维护，并且没有任何性能或者可伸缩性方面的好处。（依我们之见，RDBMS供应商不应该默认启用自动提交。必要时，SQL查询控制台和工具应该在连接上启用自动提交模式。）

10.4 小结

在本章中，你学习了事务、并发、隔离性和锁。你现在知道Hibernate依赖数据库并发控制机制，但是由于自动版本控制和持久化上下文高速缓存，却在事务中提供了更好的隔离性保证。你学习了如何利用Hibernate API、JTA UserTransaction和JPA EntityTransaction接口编程式地设置事务范围。我们还探讨了包含EJB 3.0组件的事务程序集，以及你如何利用自动提交模式非事务地进行工作。

表10-1显示了可以用来比较原生的Hibernate特性和Java Persistence的一个概括。

表10-1　第10章中Hibernate和JPA的对照表

Hibernate Core	Java Persistence和EJB 3.0
可以给JDBC和JTA配置`Transaction` API	`EntityTransaction` API只对本地资源的事务有用
Hibernate可以在EJB中被配置为与JTA和容器托管事务整合	利用Java Persistence，在Java SE和Java EE之间，只有数据库连接名称这一配置变化
Hibernate默认通过自动版本控制，为最佳的可伸缩性提供乐观并发控制	Java Persistence通过自动版本控制，标准化了乐观并发控制

现在已经结束了以事务的方式对保存和加载对象所涉及的基础知识的讨论和探索。接下来将通过在用户和应用程序之间创建更为现实的对话，来把所有的知识点串联起来。

第11章 实现对话

本章内容

- 利用Hibernate实现对话
- 利用JPA实现对话
- 利用EJB 3.0组件实现的对话

在前面章节中你已经尝试过这些例子，并在事务内部保存和加载过对象。很可能你已经注意到了五行的代码范例非常有助于你理解一个特定的问题、学习API以及对象如何改变它们的状态。如果你更进一步，试着在自己的应用程序中应用所学过的知识，可能很快意识到你还少了两个重要的概念。

本章要介绍的第一个概念——持久化上下文传播（persistence context propagation），当你必须在应用程序中调用几个类来完成一个特定的动作时非常有用，它们全都需要数据库访问。目前为止，我们只用过一种内部打开和关闭持久化上下文的方法（`Session`或者`EntityManager`）。不用手工传递类和方法之间的持久化上下文，而是要介绍在Hibernate和Java Persistence中可以自动负责传播的机制。Hibernate可以帮助你创建更为复杂的工作单元。

你接下来会遇到的设计问题是，当应用程序的用户必须通过几个屏幕被引导来完成一个工作单元时的应用程序的流程问题。你必须创建代码，控制从一个屏幕到另一个屏幕的导航——但是，这是在持久化的范围之外，本章不对其做过多讨论。持久化机制的部分责任在于，为跨越可能的用户思考时间的工作单元实现数据访问的原子性和隔离性。我们把在客户端/服务器端中完成几个请求并响应的周期的工作单元称为*对话*。Hibernate和Java Persistence提供几种对话实现的策略，本章要介绍如何把这些知识点与现实的示例结合起来。

从Hibernate开始，然后在本章的后半部分讨论JPA对话。让我们先创建更复杂的数据访问示例，看看几个类如何通过自动传播来复用相同的持久化上下文。

11.1 传播 Hibernate `Session`

回忆一下前一章中介绍过的用例：触发拍卖终止的事件必须得到处理（见10.1节）。对于下面的示例，谁触发这个事件并不重要；可能是当自动定时器到达拍卖货品的终止日期和终止时间

时终止其拍卖。也可能是人工的操作员触发了该事件。

为了处理这个事件，你需要执行一系列操作：为拍卖检查胜出的出价，收取拍卖费用，通知卖主和赢家，等等。可以编写有一个大过程的单个类。更好的一种设计是，把这些步骤中每一步的责任都移到可重用的更小组件上，并按关注点把它们分开。第16章会详细讨论这个话题。现在，假设你听取了我们的建议，需要在同一个工作单元内部调用几个类，来处理拍卖的关闭。

11.1.1　Session传播的用例

看一下代码清单11-1中的代码示例，它控制事件的处理。

代码清单11-1　关闭和终止拍卖的控制器代码

```
public class ManageAuction {

    ItemDAO     itemDAO = new ItemDAO();
    PaymentDAO  paymentDAO = new PaymentDAO();

    public void endAuction(Item item) {

        // Reattach item
        itemDAO.makePersistent(item);

        // Set winning bid
        Bid winningBid = itemDAO.getMaxBid( item.getId() );
        item.setSuccessfulBid(winningBid);
        item.setBuyer( winningBid.getBidder() );

        // Charge seller
        Payment payment = new Payment(item);
        paymentDAO.makePersistent(payment);

        // Notify seller and winner
        ...
    }
    ...
}
```

ManageAuction类被称作控制器（controller）。它的任务是协调所有必需的步骤来处理一个特定的事件。当事件触发时，定时器（或者用户界面）调用endAuction()方法。控制器没有包含完成和关闭拍卖必需的所有代码；它尽可能多地委托给其他的类。首先，它需要两个无状态的服务对象（称作数据访问对象）来完成这项工作——它们直接被控制器的每个实例实例化。当endAuction()方法需要访问数据库时，它使用DAO。例如，ItemDAO用来重附脱管的item，并在数据库中查询最高的出价。PaymentDAO用来使一项瞬时的新付款持久化。甚至不需要明白如何通知拍卖的卖主和赢家——你有足够的代码证明需要上下文传播。

代码清单11-1中的代码不能正常工作。首先，没有事务划分。endAuction()中的所有代码都被当作一个原子的工作单元：它要么全部失败，要么全部成功地完成。因此，要把所有这些操作包装在一个事务中。接下来通过不同的API来完成。

更难的问题是持久化上下文。想象ItemDAO和PaymentDAO在每一个（无状态的）方法中使

用不同的持久化上下文。换句话说，itemDAO.getMaxBid()和paymentDAO.makePersistent()都打开、清除并关闭它们自己的持久化上下文（Session或者EntityManager）。这是一种应该永远避免的反模式！在Hibernate中，这就是众所周知的每个操作一个会话，这是检验应用程序设计的性能瓶颈时，作为一名优秀的Hibernate开发人员应该注意的第一件事。单个持久化上下文不应该被用来处理一个特定的操作，而是处理整个事件（它自然可能需要几项操作）。持久化上下文的范围经常与数据库事务的范围相同。这就是众所周知的每个请求一个会话，请见图11-1。

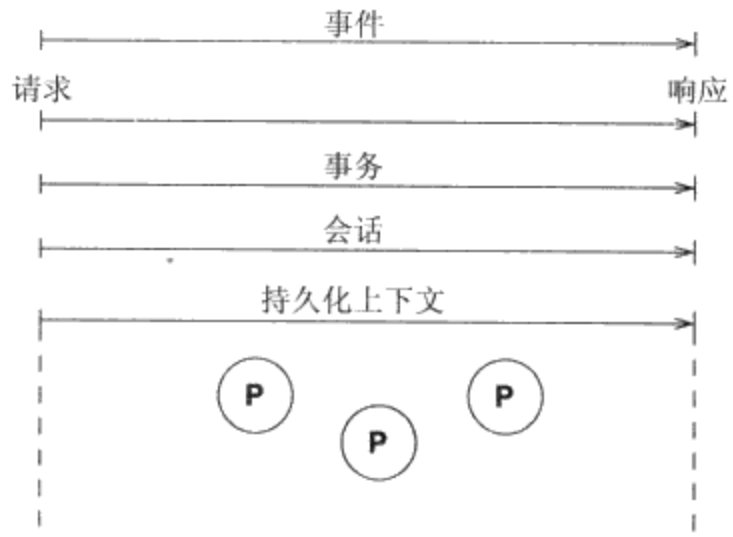


图11-1 单个持久化上下文服务于一个特定的事件

下面来把事务划分添加到ManageAuction控制器，并在数据访问类之间传播持久化上下文。

11.1.2 通过线程局部传播

Hibernate为使用简单Java SE的独立Java应用程序以及任何使用了JTA（不管是否包含EJB）的应用程序，提供了自动的持久化上下文传播。我们极力鼓励你在自己的应用程序中考虑这项特性，因为所有其他的选择方案都没有这个来得完善。

在Hibernate中，可以访问当前的Session来访问数据库。例如，考虑利用Hibernate的ItemDAO实现：

```
public class ItemDAO {

    public Bid getMaxBid(Long itemId) {
        Session s = getSessionFactory().getCurrentSession();
        return (Bid) s.createQuery("...").uniqueResult();
    }
    ...

}
```

方法getSessionFactory()返回全局的SessionFactory。它如何做到这一点则完全取决于你，取决于如何配置和部署应用程序——它可能来自于从JNDI注册中被查找或者当实例化ItemDAO时被手工注入的一个静态变量（HibernateUtil）。这种依赖管理很繁琐；SessionFactory是个线程安全的对象。

SessionFactory中的getCurrentSession()方法是我们要讨论的话题。（PaymentDAO实现

也在所有的方法中使用当前的Session。）当前的Session是什么，当前指的是什么？让我们把事务划分添加到调用ItemDAO和PaymentDAO的控制器中，请见代码清单11-2。

代码清单11-2　把事务划分添加到控制器

```
public class ManageAuction {

    ItemDAO     itemDAO = new ItemDAO();
    PaymentDAO  paymentDAO = new PaymentDAO();

    public void endAuction(Item item) {
        try {
            // Begin unit of work
            sf.getCurrentSession().beginTransaction();

            // Reattach item
            itemDAO.makePersistent(item);

            // Set winning bid
            Bid winningBid = itemDAO.getMaxBid( item.getId() );
            item.setWinningBid(winningBid);
            item.setBuyer( winningBid.getBidder() );

            // Charge seller
            Payment payment = new Payment(item);
            paymentDAO.makePersistent(payment);

            // Notify seller and winner
            ...

            // End unit of work
            sf.getCurrentSession().getTransaction().commit();

        } catch (RuntimeException ex) {
            try {
                sf.getCurrentSession().getTransaction().rollback();
            } catch (RuntimeException rbEx) {
                log.error("Couldn't roll back transaction," rbEx);
            }
            throw ex;
        }
    }
    ...
}
```

工作单元始于调用endAuction()方法的时候。如果在当前的Java线程中第一次调用sessionFactory.getCurrentSession()，就会打开和返回一个新的Session——你得到了新的持久化上下文。你通过Hibernate的Transaction接口（它在Java SE应用程序中转变为JDBC事务），在这个新的Session中立即启动一个数据库事务。

在全局共享的SessionFactory中调用getCurrentSession()的所有数据访问代码，都访问相同的当前Session——如果它在相同的线程中被调用的话。当Transaction被提交（或者回滚）时，工作单元结束。如果提交或者回滚事务，Hibernate也清除和关闭当前的Session以及它的持

久化上下文。这意味着提交或者回滚之后对getCurrentSession()的调用生成了新的Session和新的持久化上下文。

你有效地把相同的范围应用到了数据库事务和持久化上下文。你通常会想要改善这段代码，通过把事务和异常处理移到方法实现之外。一种简单的解决方案是事务拦截器（transaction interceptor），第16章中将编写一个。

Hibernate内部把当前的Session绑定到当前正在运行的Java 线程。[在Hibernate社区中，这也就是大家所知的线程本地会话（ThreadLocal Session）模式]。必须在Hibernate配置中通过把hibernate.current_session_context_class属性设置为thread来启用这个绑定。

如果用JTA部署应用程序，可以启用一种稍微不同的策略，直接把持久化上下文界定和绑定到系统事务。

11.1.3 利用 JTA 传播

在前几节中，我们始终推荐用JTA服务来处理事务，现在重申这个建议。JTA除了提供许多其他的东西之外，还提供用Hibernate接口避免代码污染的标准事务划分接口。代码清单11-3展现了利用JTA重构的ManageAuction控制器。

代码清单11-3 控制器中利用JTA的事务划分

```
public class ManageAuction {
    UserTransaction utx = null;
    ItemDAO     itemDAO = new ItemDAO();
    PaymentDAO  paymentDAO = new PaymentDAO();

    public ManageAuction() throws NamingException {
        utx = (UserTransaction) new InitialContext()
                .lookup("UserTransaction");
    }

    public void endAuction(Item item) throws Exception {
        try {
            // Begin unit of work
            utx.begin();

            // Reattach item
            itemDAO.makePersistent(item);

            // Set winning bid
            Bid winningBid = itemDAO.getMaxBid( item.getId() );
            item.setWinningBid(winningBid);
            item.setBuyer( winningBid.getBidder() );

            // Charge seller
            Payment payment = new Payment(item);
            paymentDAO.makePersistent(payment);

            // Notify seller and winner
            ...

            // End unit of work
```

```
            utx.commit();

        } catch (Exception ex) {
            try {
                utx.rollback();
            } catch (Exception rbEx) {
                log.error("Couldn't roll back transaction", rbEx);
            }
            throw ex;
        }
    }
    ...
}
```

这段代码没有任何Hibernate导入。并且更重要的是，内部使用getCurrentSession()的ItemDAO和PaymentDAO类不变。当第一次在其中一个DAO类中调用getCurrentSession()时，新的持久化上下文开始了。当前的Session被自动绑定到当前的JTA系统事务上。事务结束时，通过提交或者回滚，持久化上下文被清除，并且内部绑定的当前Session也关闭了。

与前面没有JTA的示例相比，这段代码中的异常处理略有不同，因为UserTransaction API可能抛出已检查异常（并且在构造器中的JNDI查找也可能失败）。

如果你给JTA配置Hibernate应用程序，就不必启用这个JTA绑定的持久化上下文；getCurrentSession()始终返回一个被界定和绑定到当前JTA系统事务的Session。

（注意，不能把Hibernate的Transaction接口与getCurrentSession()特性和JTA一起使用。你需要Session来调用beginTransaction()，但Session必须被绑定到当前的JTA事务上——这是个鸡和蛋的问题。这再次强调了你应该始终尽可能地使用JTA，而只有在无法使用JTA时才使用Hibernate的Transaction。）

11.1.4　利用 EJB 传播

如果把控制器编写成EJB，并应用容器托管的事务，这段代码（如代码清单11-4所示）甚至会更加简洁。

代码清单11-4　控制器中利用CMT的事务划分

```
@Stateless
public class ManageAuctionBean implements ManageAuction {

    ItemDAO     itemDAO = new ItemDAO();
    PaymentDAO  paymentDAO = new PaymentDAO();

    @TransactionAttribute(TransactionAttributeType.REQUIRED)
    public void endAuction(Item item) {

        // Reattach item
        itemDAO.makePersistent(item);

        // Set winning bid
        Bid winningBid = itemDAO.getMaxBid( item.getId() );
        item.setWinningBid(winningBid);
```

```
            item.setBuyer( winningBid.getBidder() );

            // Charge seller
            Payment payment = new Payment(item);
            paymentDAO.makePersistent(payment);

            // Notify seller and winner
            ...
        }
        ...
    }
```

当前的Session被绑定到为endAuction()方法启动的事务，当这个方法返回时，它被清除和关闭。在这个方法内部运行和调用sessionFactory.getCurrentSession()的所有代码都获得相同的持久化上下文。

如果把这个示例与代码清单11-1中第一个不能工作的示例对比，会发现你只必须添加一些注解使它生效。@TransactionAttribute甚至是可选的——它默认就是REQUIRED。这就是为什么EJB 3.0要提供一个简化的编程模型（simplified programming model）的原因。注意，目前为止，你还没有使用到JPA接口；数据访问类仍然依赖当前的Hibernate Session。稍后你可以轻松地重构它——关注点被清楚地分离了。

你现在知道了如何把持久化上下文界定到事务来服务一个特定的事件，以及如何创建更为复杂的需要在几个对象之间传播和共享持久化上下文的数据访问操作。Hibernate内部使用当前的线程或者当前的JTA系统事务，来绑定当前的Session和持久化上下文。Session和持久化上下文的范围与Hibernate Transaction或者JTA系统事务的范围相同。

我们现在关注第二个设计概念，它可以明显改善设计和创建数据库应用程序的方式。我们将实现长运行的对话，与跨越用户思考时间的应用程序用户的一系列交互。

11.2 利用 Hibernate 的对话

前面的章节已经多次介绍了对话的概念。第一次，我们说对话是跨用户思考时间的工作单元。然后探讨了必须放在一起来创建对话应用程序的基础构建块：脱管对象、重附、合并，以及使用被扩展的持久化上下文的其他可选策略。

现在，该来看看所有这些选项的实战了。我们在前一个示例（拍卖的关闭和完成）上构建，并把它变成对话。

11.2.1 提供对话保证

你已经实现过对话——只是它不长。你实现了最短的可能的对话：跨越来自应用程序用户的单个请求的对话：用户（假设现在指的是人工操作员）在CaveatEmptor管理界面单击Complete Auction（完成拍卖）按钮。然后处理这个被请求的事件，并且把表明事件成功的响应显示给操作人员。

在实践中，短对话很常见。几乎所有的应用程序都有更复杂的对话——更复杂的必须被组合

起来作为一个单元的一系列动作。例如，单击Complete Auction按钮的人工操作员这么做，因为他们确信这次拍卖应该结束。他们根据查看展现在屏幕上的数据做出决定——信息是如何到达那里的？更早的请求被发送到应用程序，并触发拍卖加载用于显示。从应用程序用户的观点来看，这个数据加载是同一个工作单元的一部分。这似乎很合理：应用程序也应该知道这两个事件——用于显示的拍卖项目的加载和拍卖的完成——假定处于同一个工作单元中。我们正在扩大工作单元的概念，并采用应用程序用户的观点。把两个事件都组合到同一个对话中。

应用程序用户希望在经过这个对话时，应用程序能有一些保证：

- 用户将要关闭和终止的拍卖在他们看着它的时候没有被修改。拍卖的完成要求这个决定所基于的数据在完成时仍然不变。否则，操作人员就是在失效的数据上工作，可能会做出错误的决定。
- 对话是原子的：用户可以随时终止对话，并且他们做过的所有变化都被回滚。这在我们当前的场景中不是个大问题，因为只有最后一个事件会进行任何永久的改变；第一个请求只加载用于显示的数据。然而，更为复杂的对话也有可能，且很常见。

作为应用程序的开发人员，你希望通过尽可能少的工作实现这些保证。

现在介绍如何利用Hibernate（无论是否带有EJB）实现长对话。在任何环境下，你必须做的第一项决定都是，要使用利用脱管对象的策略，还是使用扩展持久化上下文的策略。

11.2.2　利用脱管对象的对话

让我们用原生的Hibernate接口和脱管对象策略来创建对话。对话有两个步骤：第一步加载对象，第二步使对被加载的对象所做的变化变成持久化。这两个步骤如图11-2所示。

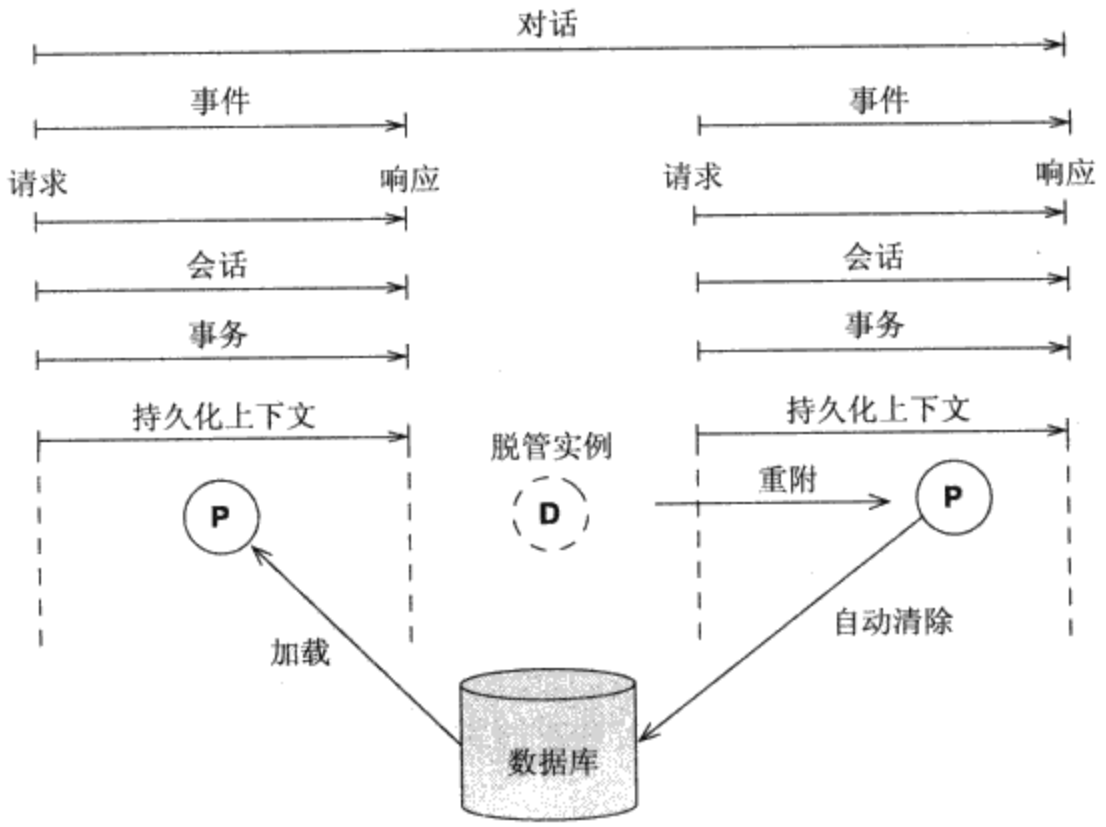


图11-2　利用脱管对象实现的两步对话

第一步需要Hibernate Session来按其标识符获取实例（假设这是个指定的参数）。你将编写另一个可以处理这个事件的ManageAuction控制器：

```
public class ManageAuction {

    public Item getAuction(Long itemId) {
        Session s = sf.getCurrentSession();

        s.beginTransaction();

        Item item = (Item) s.get(Item.class, itemId);

        s.getTransaction().commit();

        return item;
    }
    ...
}
```

对这段代码进行一些简化，避免弄乱例子——你知道异常处理实际上并不是可选的。注意，这比之前介绍的版本更为简单；我们要介绍理解对话所需的最少代码。如果你喜欢，也可以用DAO编写这个控制器。

调用getAuction()的时候，一个新的Session、持久化上下文和数据库事务开始了。对象从数据库中加载，事务提交，并且持久化上下文关闭。现在item对象处于脱管状态，并且返回到调用这个方法的客户端。客户端利用脱管对象，显示它，甚至可能允许用户修改它。

会话中的第二步是完成拍卖。这是ManageAuction控制器中另一种方法的用途。与前面的示例相比，它再次简化了endAuction()方法，避免任何不必要的复杂：

```
public class ManageAuction {

    public Item getAuction(Long itemId) ...
    ...

    public void endAuction(Item item) {
        Session s = sf.getCurrentSession();

        s.beginTransaction();

        // Reattach item
        s.update(item);

        // Set winning bid
        // Charge seller
        // Notify seller and winner
        ...

        s.getTransaction().commit();
    }

}
```

客户端调用endAuction()方法，并传回脱管的item实例——这与第一步中返回的实例相同。Session中的update()操作把脱管对象重附到持久化上下文中，并计划一个SQL UPDATE。Hibernate必须假设客户端在对象被脱管时对它进行了修改。（否则，如果你确定它没有被修改，lock()就足够了。）当对话中的第二个事务提交时，持久化上下文自动被清除，并且对曾经脱管

而现在持久化的对象所做的任何修改都与数据库同步。

在实践中，saveOrUpdate()方法比update()、save()或者lock()更有用：在复杂的对话中，你不知道item是否处于脱管状态，或者它是否为新的、瞬时的、必须保存的。当你不仅在单个实例上工作，而且还想要重附或者持久化被连接的对象图并应用级联选项时，saveOrUpdate()提供的自动状态侦测变得更有用。还要重读merge()操作的定义，以及什么时候使用合并而不是重附：详见9.3.2节。

目前为止，你还只是解决了其中一个对话实现问题：用很少的代码实现对话。然而，应用程序的用户仍然期待工作单元不仅要与并发修改隔离，而且还要有原子性。

你用乐观锁把并发的对话隔离开来。一般来说，不应该应用跨越长运行对话的悲观并发控制策略——这意味着昂贵和不可伸缩的锁定。换句话说，不能防止两个操作人员看见同一件拍卖货品。你希望这种情况很少发生：你是乐观的。但是如果发生了，你也有一种冲突的解决策略。需要给Item持久化类启用Hibernate自动的版本控制，就像在10.2.2节中所做的那样。然后，在对话期间的任何时候，每一个SQL UPDATE或者DELETE都将包括一个针对数据库中所展现的状态的版本检查。如果这个检查失败，就得到StaleObjectStateException，然后必须采取适当的行动。在这个例子中，你给用户展现了一条错误消息（“对不起，有人修改了同一件拍卖品！”），并强制从第一步开始重启对话。

如何使对话变成原子化呢？对话跨越几个持久化上下文和几个数据库事务。但是从应用程序用户的观点来看，这并不是工作单元的范围；他们认为对话会是一个原子的操作组合，要么全部失败，要么全部成功。在当前的对话中，这不成问题。因为你只在最后一步（第二步）中修改和持久化数据。任何只读取数据，并把被修改对象的重附延迟到最后一步的对话，都自动成为原子化，可以随时终止。如果对话在一个中间的步骤中把修改重附和提交到数据库，它就不再是原子的。

一种解决方案是不在提交时清除持久化上下文——也就是说，在一个假定不持久化修改的Session中设置FlushMode.MANUAL（当然，不是针对对话的最后一步）。另一种方法是使用补偿（compensation）动作，撤销（undo）任何进行永久改变的步骤，并在用户终止会话时调用适当的补偿动作。我们不过多讨论编写补偿动作；它们取决于你正在实现的对话。

接下来用不同的策略实现相同的对话，消除脱管的对象状态。你扩展持久化上下文为跨整个对话。

11.2.3　给对话扩展 Session

Hibernate Session有一个内部的持久化上下文。通过扩展持久化上下文来跨整个对话，你可以实现一个不涉及脱管对象的对话。这就是大家所知的每个对话一个会话（session-per-conversation）策略，如图11-3所示。

在对话开始的时候打开新的Session和持久化上下文。第一步，Item对象的加载在第一个数据库事务中实现。一旦提交数据库事务，Session就立即自动地从底层的JDBC Connection中断开。现在你可以在用户思考时间的过程中保持这个断开连接的Session和它内部的持久化上下

文。一旦用户继续对话，并执行下一个步骤，你就立即通过启动第二个数据库事务把Session重新连接到新的JDBC Connection。已经在这个对话中加载的任何对象都处于持久化状态：永远不会脱管。因而，一旦你在Session中调用flush()，对任何持久化对象所做的修改就立即被清除到数据库。必须通过设置FlushMode.MANUAL来禁用Session的自动清除——你应该在对话启动和Session打开的时候完成这项工作。

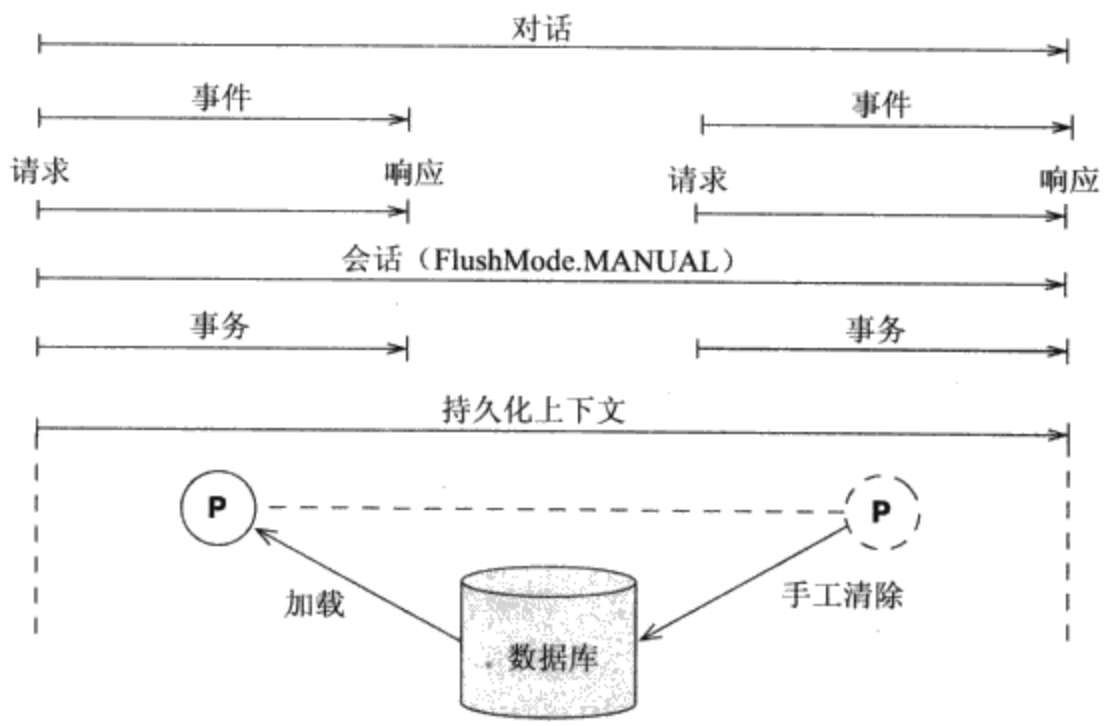


图11-3 断开连接的持久化上下文扩展为跨整个对话

在清除的期间，由于乐观锁和Hibernate的自动版本检查，在并发对话中所做的修改被隔离。如果你直到最后一步即对话结束时才清除Session，对话的原子性就得到了保证——如果关闭未清除的Session，就是有效地终止了对话。

需要详细阐述这种行为的一个例外：新实体实例的插入时间。注意，这在本例中不成问题，但它是在更复杂的对话中必须处理的东西。

1. 延迟到清除时间插入

为了理解这个问题，回想一种保存对象的方法，以及分配它们标识符值的方式。因为你没有在Complete Auction的对话中保存任何新对象，因此还没有遇到过这个问题。但是，在中间步骤中保存对象的任何对话都不可能有原子性。

Session中的save()方法要求必须返回被保存的实例的新数据库标识符。因此，标识符值必须在调用save()方法的时候生成。对于大多数标识符生成器策略来说，这不成问题；例如，Hibernate可以调用sequence，在内存中进行increment，或者向hilo生成器要一个新值。Hibernate不一定要执行SQL INSERT来在save()上返回标识符值，并把它分配到目前持久化的实例。

例外的是INSERT发生之后触发的标识符生成策略。其中一个是identity，另一个是select；两者都要求先插入一个行。如果通过这些标识符生成器映射持久化类，调用save()时就立即执行INSERT！因为你正在对话期间提交数据库事务，这个插入可能有永久的效果。

看看下列稍微不同的对话代码，它示范了这种效果：

```
Session session = getSessionFactory().openSession();
session.setFlushMode(FlushMode.MANUAL);

// First step in the conversation
session.beginTransaction();
Item item = (Item) session.get(Item.class, new Long(123) );
session.getTransaction().commit();

// Second step in the conversation
session.beginTransaction();
Item newItem = new Item();
Long newId = (Long) session.save(newItem); // Triggers INSERT!
session.getTransaction().commit();

// Roll back the conversation!
session.close();
```

你可以期待整个对话（两个步骤）可以通过关闭未清除的持久化上下文进行回滚。newItem的插入应该被延迟到你在Session上调用flush()的时候，这在本代码中永远不会发生。这是个只有在没有选择identity或者select作为标识符生成器时才会发生的情况。利用这些生成器，INSERT必须在对话的第二步中执行，并被提交到数据库。

除了关闭未被清除的持久化上下文之外，一种解决方案是使用补偿动作，执行它们用来撤销在被终止的对话期间所做的任何可能的插入。你必须手工删除插入的行。另一种解决方案是用不同的标识符生成器，如sequence，支持新标识符值的生成而不用插入。

persist()操作让你面临了相同的问题。然而，它还提供了另一种可供选择（更好）的解决方案。它可以延迟插入，如果在事务之外调用它，甚至可以用事后插入（post-insert）标识符生成：

```
Session session = getSessionFactory().openSession();
session.setFlushMode(FlushMode.MANUAL);

// First step in the conversation
session.beginTransaction();
Item item = (Item) session.get(Item.class, new Long(1));
session.getTransaction().commit();

// Second step in the conversation
Item newItem = new Item();
session.persist(newItem);

// Roll back the conversation!
session.close();
```

persist()方法可以延迟插入，因为它不必返回标识符值。注意，newItem实体在你调用persist()之后处于持久化状态，但是如果你用identity或者select生成器策略映射持久化类，它就没有分配到标识符值。当在清除时发生INSERT时，标识符值就被分配到实例。在事务之外调用persist()时，没有执行任何SQL语句。newItem对象只是在列队中等待插入。

记住，我们已经讨论的这个问题取决于所选择的标识符生成器策略——你可能不会遇上它，或者可能避免它。persist()的非事务行为在本章稍后再次变得重要，当你用JPA而不是Hibernate接口编写对话的时候。

让我们先用一个被扩展的Session完成对话的实现。利用“每个对话一个会话”策略，不再需要在代码中手工脱管和重附（或者合并）对象。必须给整个对话实现可以重用同一个Session的基础代码。

2. 管理当前的Session

前面讨论过的当前Session支持是一种可转换的机制。你已经见过两种可能的内部策略：一种是线程绑定，另一种把当前的Session绑定到JTA事务。但是，这两者都在事务结束时关闭Session。需要一个不同的Session范围用于“每个对话一个会话”的模式，但是你仍然想要能够在应用程序的代码中访问当前的Session。

第三种内建的选项正是你使用“每个对话一个会话”策略需要的东西。必须通过把hibernate.current_session_context_class配置选项设置为managed来启用它。已经讨论过的其他内建选项是thread和jta，如果你给JTA部署配置Hibernate，后者就会被隐式地启用。注意，所有这些内建的选项都是org.hibernate.context.CurrentSessionContext接口的实现；可以编写自己的实现，并在配置中命名该类。这通常没有必要，因为内建的选项涵盖了大多数的情况。

你刚刚启用的Hibernate内建实现称作托管，因为它把管理范围（启动和终止当前的Session）委托给你。你用3种静态的方法管理Session的范围：

```
public class ManagedSessionContext implements CurrentSessionContext {
    public static Session bind(Session session) { ... }
    public static Session unbind(SessionFactory factory) { ... }
    public static boolean hasBind(SessionFactory factory) { ... }
}
```

你可能已经猜到了“每个对话一个会话”策略必须做什么：

- 当对话启动时，必须用ManageSessionContext.bind()打开和绑定一个新的Session，来服务对话中的第一个请求。你还必须在这个新的Session中设置FlushMode.MANUAL，因为你不想在背后发生任何持久化上下文的同步。
- 现在调用SessionFactory.getCurrentSession()的所有数据访问代码都接收到了你所绑定的Session。
- 当对话中的请求完成时，你需要调用ManagedSessionContext.unbind()，并在某处保存现在断开连接的Session，直到对话中有了下一个请求。或者，如果这是对话中的最后一个请求，你需要清除和关闭Session。

所有这些步骤都可以在拦截器（interceptor）中实现。

3. 创建对话拦截器

你需要一个为了响应对话中的每个请求而被自动触发的拦截器。如果使用EJB（像你很快要做的那样），就可以免费获得许多这种基础结构代码。如果你编写非Java EE的应用程序，就必须编写自己的拦截器。编写拦截器有许多种方法；我们介绍一种只示范概念的抽象拦截器。可以在CaveatEmptor下载文件中的org.hibernate.ce.auction.web.filter包中找到Web应用程序有效的、已经通过测试的拦截器实现。

假设每当对话中的事件必须被处理时，拦截器就运行起来。还假设每个事件都必须通过前端控制器和它的execute()动作方法——最容易的场景。现在可以把这个方法包装在拦截器中；也就是说，编写一个在这个方法执行之前和之后调用的拦截器。如图11-4所示；从左到右阅读编了号的项目。

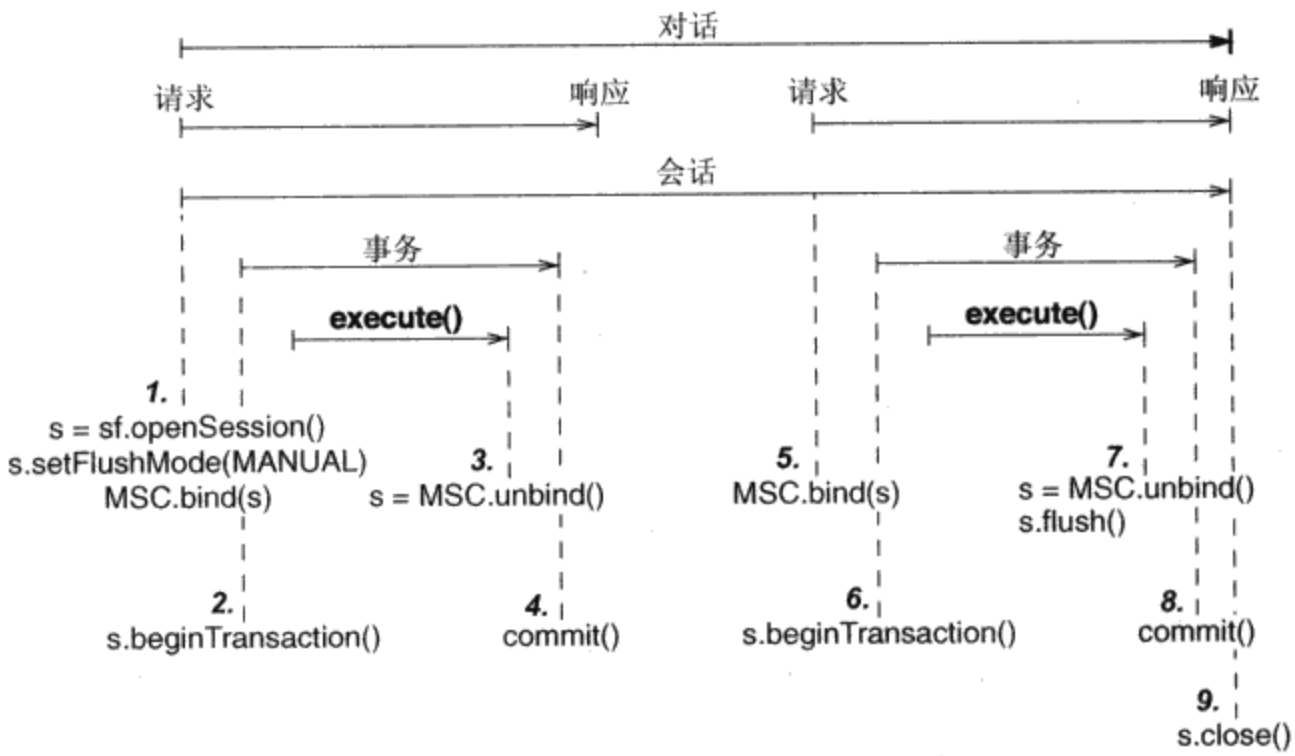


图11-4　拦截事件来管理Session的生命周期

当对话中的第一个请求命中服务器，拦截器运行并打开新的Session❶；这个Session的自动清除立即被禁用。然后这个Session被绑定到Hibernate的ManagedSessionContext。在拦截器让控制器处理事件之前启动事务❷。在这个控制器（或者任何被该控制器调用的DAO）内部运行的所有代码现在都可以调用SessionFactory.getCurrentSession()，并使用Session。当控制器完成它的工作时，拦截器再次运行，并解绑当前的Session❸。事务提交❹之后，Session自动断开连接，并且可以在用户思考时间内保存。

现在，服务器等待对话中的第二个请求。

一旦第二个请求命中服务器，拦截器就立即运行，侦测到有被保存的断开连接的Session，并把它绑定到ManagedSessionContext❺中。控制器在事务被拦截器❻启动之后处理事件。当控制器完成它的工作时，拦截器再次运行，并从Hibernate中解绑当前的Session。然而，不是断开连接并保存它，而是拦截器现在侦测到这是对话的终点，且Session需要在提交事务之前❽被清除❼。最后，对话结束，拦截器关闭Session❾。

这听起来比在代码中更为复杂。代码清单11-5是这种拦截器的一个伪实现：

代码清单11-5　拦截器实现“每个对话一个会话”策略

```
public class ConversationInterceptor {

    public Object invoke(Method method) {
```

```
// Which Session to use?
Session currentSession = null;

if (disconnectedSession == null) {
    // Start of a new conversation
    currentSession = sessionFactory.openSession();
    currentSession.setFlushMode(FlushMode.MANUAL);
} else {
    // In the middle of a conversation
    currentSession = disconnectedSession;
}

// Bind before processing event
ManagedSessionContext.bind(currentSession);

// Begin a database transaction, reconnects Session
currentSession.beginTransaction();

// Process the event by invoking the wrapped execute()
Object returnValue = method.invoke();

// Unbind after processing the event
currentSession =
    ManagedSessionContext.unbind(sessionFactory);

// Decide if this was the last event in the conversation
if ( returnValue.containsEndOfConversationToken() ) {

    // The event was the last event: flush, commit, close
    currentSession.flush();
    currentSession.getTransaction().commit();
    currentSession.close();

    disconnectedSession = null; // Clean up

} else {

    // Event was not the last event, continue conversation
    currentSession.getTransaction().commit(); // Disconnects
    disconnectedSession = currentSession;
}

return returnValue;
    }
}
```

invoke(Method)拦截器把控制器的execute()操作包装起来。每当必须处理来自应用程序的用户的请求时，这段拦截代码就会运行。当它返回时，你检查返回值是否包含一个特殊的标记或者记号。这个标记意味着这是特定对话中必须被处理的最后一个事件。现在清除Session，提交所有变化，并关闭Session。如果这不是对话的最后一个事件，你就提交数据库事务，保存断开连接的Session，并继续等待对话中的下一个事件。

这个拦截器对于调用execute()的任何客户端代码都是透明的。它对于在execute()内部运行的任何代码也是透明的：任何数据访问操作都使用当前的Session；关注点被恰当地分离。我们甚至不必展示数据访问代码，因为它不用任何数据库事务划分或者Session处理。只是利用

getCurrentSession()加载和保存对象。

你心里可能会有下列疑问：

- 当应用程序等待用户发送对话的下一个请求时，disconnectSession保存在哪里？它可以保存在HttpSession中甚至有状态的EJB中。如果没有使用EJB，这个责任就委托给了应用程序的代码。如果使用EJB 3.0和JPA，你可以把持久化上下文的范围（Session的一个等价物）绑定到一个有状态的EJB——简化编程模型的另一个优势。
- 标志对话结束的特殊标记来自哪里？在我们的抽象示例中，这个标记出现在execute()方法的返回值中。把这样的特殊信号实现到拦截器有多种方法，只要你找到一种方法把它传到那里。把它放在事件处理的结果中是一种务实的解决方案。

这样就结束了我们对利用Hibernate的持久化上下文传播和对话实现的讨论。前面的小节缩短和简化了一些示例，让你更容易理解概念。如果想用Hibernate继续实现更加复杂的工作单元，建议也先读一下第16章。

另一方面，如果没有使用Hibernate API，却想要使用Java Persistence和EJB 3.0组件，就接着往下读。

11.3　使用 JPA 的对话

现在看看利用JPA和EJB 3.0进行的持久化上下文传播和对话实现。就像使用原生的Hibernate一样，当你想要利用Java Persistence实现对话时，必须考虑三点：

- 你想要传播持久化上下文，以便持久化上下文可以在特定的请求中用于所有数据访问。在Hibernate中，这项功能内建在getCurrentSession()特性中。如果JPA在Java SE中独立部署，它就没有这项特性。另一方面，由于EJB 3.0编程模型和预设好的范围，以及事务与托管组件的生命周期，与EJB结合的JPA比原生的Hibernate更为强大。
- 如果你决定使用脱管的对象方法作为你的对话实现策略，就要使得对脱管对象的改变成为持久化。Hibernate提供重附和合并；JPA只支持合并。前一章曾深入讨论过其中的区别，但是我们想要通过更为现实的对话示例简要地对它进行重述。
- 如果决定用“每个对话一个会话”方法作为对话实现策略，就要扩展持久化上下文来跨整个对话。我们看一下JPA持久化上下文范围，并探讨如何在Java SE中使用JPA和EJB组件来实现被扩展的持久化上下文。

注意，必须再次在两种不同的环境中处理JPA：在简单的Java SE中，以及通过EJB在Java EE环境中。阅读本节时，可以对其中一个更加关注。我们之前通过先讨论上下文传播，然后讨论长对话，探讨了这个使用Hibernate的对话主题。使用JPA和EJB 3.0我们将同时探讨这两种，但是把Java SE和Java EE放在单独的小节中。

先在一个没有任何托管组件或者容器的Java SE应用程序中，通过JPA实现对话。我们经常会指出原生的Hibernate对话之间的区别，以便确定你理解了本章前面几节的内容。下面讨论之前提出过的3个问题：持久化上下文传播、脱管实例的合并和被扩展的持久化上下文。

11.3.1 Java SE 中的持久化上下文传播

再次考虑代码清单11-1中的控制器。这段代码依赖执行持久化操作的DAO。这里再现了这种包含Hibernate API的数据访问对象的实现：

```
public class ItemDAO {

    public Bid getMaxBid(Long itemId) {
        Session s = getSessionFactory().getCurrentSession();
        return (Bid) s.createQuery("...").uniqueResult();
    }
    ...

}
```

如果试图通过JPA重构，唯一的选择似乎只能是这样：

```
public class ItemDAO {

    public Bid getMaxBid(Long itemId) {
        Bid maxBid;
        EntityManager em = null;
        EntityTransaction tx = null;
        try {
            em  = getEntityManagerFactory().createEntityManager();
            tx = em.getTransaction();
            tx.begin();

            maxBid = (Bid) em.createQuery("...")
                             .getSingleResult();
            tx.commit();
        } finally {
            em.close();
        }
        return maxBid;
    }
    ...

}
```

如果在Java SE中应用程序在它自身上处理EntityManager，JPA中就没有定义任何持久化上下文传播。在Hibernate SessionFactory中，没有getCurrentSession()方法的等价物。

在Java SE中获得EntityManager的唯一方法是在工厂中利用createEntityManager()方法来实例化。也就是说，数据访问的所有方法使用它们自己的EntityManager实例——这就是我们前面指出过的“每个操作一个会话”的反模式！更糟糕的是，对于跨几个数据访问操作的事务划分，并没有明显的位置。

对于这个问题，有3种可能的解决方案：

- 可以在创建DAO时给整个DAO实例化EntityManager。这不会让你获得每个请求一个持久化上下文（persistence-context-per-request）范围，但是比“每个操作一个持久化上下文”略好一点。但是，事务划分仍然是这种策略的一个问题；所有DAO中的所有DAO操作都无法组合成为一个原子的、隔离的工作单元。

- 可以在控制器中实例化单个EntityManager，并在创建DAO（构造器注入）时，把它传递到所有的DAO里面。这样就解决了问题。处理EntityManager的代码可以与单个位置中的事务划分代码（即控制器）配对。
- 可以在拦截器中实例化单个EntityManager，并把它绑定到辅助类中的ThreadLocal变量。DAO从ThreadLocal中获取当前的EntityManager。这种策略模拟Hibernate中的getCurrentSession()功能。拦截器也可以包括事务划分，并且可以把控制器方法包装在拦截器中。不用自己编写这个基础结构代码，可以首先考虑EJB。

我们把在Java SE中要对持久化上下文传播使用哪种策略的决定权留给你。建议考虑Java EE组件、EJB，然后就可以使用强大的上下文传播了。可以轻松地给应用程序部署轻量化的EJB容器，就像在2.2.3节中所做的那样。

让我们进行第二个问题：长对话中脱管实例的修改。

11.3.2　在对话中合并脱管对象

我们已经详细阐述了脱管对象的概念，以及如何把被修改的实例重附到新的持久化上下文，或者另一种方法是，把它们合并到新的持久化上下文。因为JPA只给合并提供持久化操作，所以有必要复习一下有关利用原生的Hibernate代码合并的示例和注释（在9.3.2节中），以及9.4.2节对JPA中脱管对象的讨论。

在这里我们想要关注之前提出过的一个问题，并且从一个稍微不同的视角来看待它。这个问题就是：为什么要从merge()操作中返回持久化的实例？

你之前用Hibernate实现的长对话有两个步骤（两个事件）。在第一个事件中，获取一件拍卖货品用于显示。在第二个事件中，（可能被修改过的）货品被重附到一个新的持久化上下文中，然后关闭拍卖。

代码清单11-6展现了相同的控制器，它可以通过JPA和合并，服务于两个事件：

代码清单11-6　利用JPA合并脱管对象的控制器

```
public class ManageAuction {

    public Item getAuction(Long itemId) {
        EntityManager em = emf.createEntityManager();
        EntityTransaction tx = em.getTransaction();

        tx.begin();

        Item item = em.find(Item.class, itemId);

        tx.commit();
        em.close();

        return item;
    }

    public Item endAuction(Item item) {
        EntityManager em = emf.createEntityManager();
        EntityTransaction tx = em.getTransaction();
```

```
        tx.begin();

        // Merge item
        Item mergedItem = em.merge(item);

        // Set winning bid
        // Charge seller
        // Notify seller and winner
        // ... this code uses mergedItem!

        tx.commit();
        em.close();

        return mergedItem;
    }

}
```

这里应该没有令你感到惊讶的代码——你已经多次见过所有这些操作。考虑调用这个控制器的客户端，它通常是某种表现代码。首先，调用getAuction()方法，获取用于显示的Item实例。不久，触发第二个事件，并调用endAuction()方法。脱管的Item实例被传递到这个方法中；但是该方法也返回一个Item实例。返回的Item、mergeItem是不同的实例！客户端现在有两个Item对象：旧的和新的。

就像我们在9.3.2节中指出的，对旧实例的引用应该被客户端认为是过时的：它不表示最新的状态。只有mergeItem才是对最新状态的引用。使用合并（而不是重附），丢弃对失效对象的过时引用变成是客户端的责任。如果考虑下列客户端代码，这通常不成问题：

```
ManageAuction controller = new ManageAuction();

// First event
Item item = controller.getAuction( 12341 );

// Item is displayed on screen and modified...
item.setDescription("[SOLD] An item for sale");

// Second event
item = controller.endAuction(item);
```

最后一行代码设置了合并结果为item变量值，因此你有效地通过新引用更新了这个变量。记住，这一行只更新这个变量。表现层中仍然有对旧实例引用的任何其他代码也必须刷新变量——要多加小心。这实际上意味着表现代码必须知道重附和合并这两种策略之间的区别。

我们发现，通过被扩展的持久化上下文（extended persistence context）策略构造的应用程序经常比严重依赖脱管对象的应用程序更加容易理解。

11.3.3 在 Java SE 中扩展持久化上下文

我们已经在10.1.3节中讨论过，在Java SE中使用JPA的持久化上下文的范围。现在详细阐述这些基础的知识，并主要关注包含对话实现的一个被扩展的持久化上下文示例。

1. 默认的持久化上下文范围

在没有EJB的JPA中，持久化上下文被绑定到EntityManager实例的生命周期和范围。为了在对

话中给所有的事件重用相同的持久化上下文，只需要重用同一个EntityManager去处理所有事件。

一种简单的方法是把这个责任委托给对话控制器的客户端：

```
public static class ManageAuctionExtended {

    EntityManager em;

    public ManageAuctionExtended(EntityManager em) {
        this.em = em;
    }

    public Item getAuction(Long itemId) {
        EntityTransaction tx = em.getTransaction();

        tx.begin();

        Item item = em.find(Item.class, itemId);

        tx.commit();

        return item;
    }

    public Item endAuction(Item item) {
        EntityTransaction tx = em.getTransaction();

        tx.begin();

        // Merge item
        Item mergedItem = em.merge(item);

        // Set winning bid
        // Charge seller
        // Notify seller and winner
        // ... this code uses mergedItem!

        tx.commit();

        return mergedItem;
    }
}
```

控制器期待在它的构造器中给整个对话设置持久化上下文。客户端现在创建和关闭EntityManager：

```
// Begin persistence context and conversation
EntityManager em = emf.createEntityManager();

ManageAuctionExtended controller = new ManageAuctionExtended(em);

// First event
Item item = controller.getAuction( 12341 );

// Item is displayed on screen and modified...
item.setDescription("[SOLD] An item for sale");

// Second event
controller.endAuction(item);

// End persistence context and conversation
em.close();
```

自然地，包装有getAuction()和endAuction()方法并提供正确EntityManager实例的拦截器可能更加方便。它还避免了关注点向上渗透到表现层。如果把控制器编写成有状态的EJB会话bean，就可以轻松地获得这个拦截器。

当你试图通过跨整个对话的被扩展持久化上下文应用这种策略时，或许会遇到一个可能破坏对话原子性的问题——自动清除。

2. 防止自动清除

考虑下列对话，它添加了一个事件作为中间步骤：

```
// Begin persistence context and conversation
EntityManager em = emf.createEntityManager();

ManageAuctionExtended controller = new ManageAuctionExtended(em);

// First event
Item item = controller.getAuction( 12341 );

// Item is displayed on screen and modified...
item.setDescription("[SOLD] An item for sale");

// Second event
if ( !controller.sellerHasEnoughMoney(seller) )
    throw new RuntimeException("Seller can't afford it!");

// Third event
controller.endAuction(item);

// End persistence context and conversation
em.close();
```

看看这个新的对话客户端代码，你认为更新过的货品描述什么时候被保存在数据库中？这取决于持久化上下文的清除。你知道JPA中默认的FlushMode是AUTO，它在执行查询之前和提交事务时启用同步。对话的原子性取决于sellerHasEnoughMoney()方法的实现，以及它是执行查询还是提交事务。

假设你通过一般的事务块，把在该方法内部执行的操作包装起来：

```
public class ManageAuctionExtended {
    ...

    public boolean sellerHasEnoughMoney(User seller) {
        EntityTransaction tx = em.getTransaction();
        tx.begin();
        boolean sellerCanAffordIt = (Boolean)
            em.createQuery("select...").getSingleResult();

        tx.commit();

        return sellerCanAffordIt;
    }

    ...
}
```

这个代码片段甚至包括两个调用，它们触发EntityManager持久化上下文的清除。首先，FlushMode.AUTO意味着查询的执行触发了一次清除。其次，事务提交触发了另一次清除。这显

然不是你想要的——你想要使整个对话原子化，并在最后一个事件完成之前防止任何清除。

Hibernate提供org.hibernate.FlushMode.MANUAL，它从同步中解耦事务划分。不幸的是，由于JSR-220专家组的成员意见不一，javax.persistence.FlushMode只提供AUTO和COMMIT。在介绍“正式”的解决方案之前，先看看下列代码是如何通过退回到Hibernate API来获得FlushMode.MANUAL的：

```
// Prepare Hibernate-specific EntityManager parameters
Map params = new HashMap();
params.put("org.hibernate.flushMode," "MANUAL");

// Begin persistence context with custom parameters
EntityManager em = emf.createEntityManager(params);

// Alternative: Fall back and disable automatic flushing
((org.hibernate.Session)em.getDelegate())
   .setFlushMode(org.hibernate.FlushMode.MANUAL);

// Begin conversation
ManageAuction controller = new ManageAuction(em);

// First event
Item item = controller.getAuction( 12341 );

// Item is displayed on screen and modified...
item.setDescription("[SOLD] An item for sale");

// Second event
if ( !controller.sellerHasEnoughMoney(seller) )
    throw new RuntimeException("Seller can't afford it!");

// Third event
controller.endAuction(item);

// End persistence context and conversation
em.close();
```

别忘了em.flush()必须在第三个事件的最后一个事务中被手工调用，否则就没有任何修改会变成持久化：

```
public static class ManageAuctionExtended {
    ...
    public Item endAuction(Item item) {
        EntityTransaction tx = em.getTransaction();

        tx.begin();

        // Merge item
        ...
        // Set winning bid
        ...

        em.flush(); // Commit the conversation

        tx.commit();

        return mergedItem;
    }
}
```

正式的架构解决方案依赖非事务的行为。不是用简单的FlushMode设置，而是需要给数据访问操作编写没有事务范围的代码。专家组成员对于漏掉的FlushMode给出的其中一个原因是，“事务提交应该使所有修改永久化”。因此，你只能通过移除事务划分，给对话中的第二步禁用清除：

```
public class ManageAuction {
    ...
    public boolean sellerHasEnoughMoney(User seller) {
        boolean sellerCanAffordIt = (Boolean)
            em.createQuery("select ...").getSingleResult();
        return sellerCanAffordIt;
    }
    ...
}
```

这段代码没有触发持久化上下文的清除，因为EntityManager是在任何事务范围之外使用的。执行这个查询的EntityManager现在正在自动提交模式下工作，包含我们先前在10.3节中涵盖的所有值得关注的结果。更糟糕的是，你失去了拥有可重复读取的能力：如果同一个查询执行两次，每次查询都在自动提交模式下在它们自己的数据库连接上执行。它们可能返回不同的结果，因此数据库事务隔离级别可重复读取（repeatable read）和可序列化（serializable）都不起作用。换句话说，利用正式的解决方案，你无法获得可重复读取的数据库事务隔离性，并同时禁用自动清除。持久化上下文高速缓存可以只对实体查询（而不对标量查询）提供可重复读取。

如果用JPA实现对话，强烈建议你考虑Hibernate的FlushMode.MANUAL设置。我们也期待这个问题将在规范的未来版本中得到解决；（几乎）所有的JPA供应商都已经包括一个专有的清除模式设置，效果与org.hibernate.FlushMode.MANUAL的一样。

你现在知道如何通过脱管的实体实例和被扩展的持久化上下文编写JPA对话了。前几节为下一步打下了基础：利用JPA和EJB的对话实现。如果现在觉得JPA比Hibernate更麻烦，那么相信你定会惊讶于一旦引入EJB后的对话实现却变得非常容易。

11.4 使用 EJB 3.0 的对话

必须再看一下问题清单：持久化上下文传播、脱管对象的处理和跨整个对话的被扩展的持久化上下文。这一次，你将在其中增加EJB。

我们将不过多讨论脱管的实体实例，以及如何在对话中合并持久化上下文之间的修改——要用到的概念和API与Java SE中和利用EJB时的完全相同。

另一方面，当你引入EJB，然后依赖标准的上下文传播规则以及JPA与EJB3.0编程模型的整合时，持久化上下文传播和利用JPA的扩展的持久化上下文管理将变得更为容易。

先关注EJB调用中的持久化上下文传播。

11.4.1 使用 EJB 的上下文传播

JPA和EJB 3.0定义持久化上下文在应用程序中如何处理，以及当几个类（或者EJB组件）使用一个EntityManager时所应用的规则。使用EJB的应用程序中最常见的案例是容器托管的且被

注入的EntityManager。你通过注解，把ItemDAO变成托管的无状态EJB组件，并重写代码来使用EntityManager：

```
@Stateless
public class ItemDAOBean implements ItemDAO {

    @PersistenceContext
    private EntityManager em;

    @TransactionAttribute(TransactionAttributeType.REQUIRED)
    public Bid getMaxBid(Long itemId) {
        return (Bid) em.createQuery("...").getSingleResult();
    }
    ...

}
```

当这个bean的客户端调用getMaxBid()时，EJB容器就注入EntityManager。这个EntityManager的持久化上下文是当前的持久化上下文（很快会做更详细的讨论）。如果调用getMaxBid()时没有事务在处理，getMaxBid()返回时就会启动新的事务并提交。

说明　许多开发人员过去没有对使用EJB 2.1的DAO类使用EJB会话bean。在EJB 3.0中，所有的组件都是简单的Java对象，你确实应该通过几个简单的注解（如果你不喜欢注解，就用XML部署描述符）来获得容器的服务。

1. 连接EJB组件

既然ItemDAO是EJB组件（如果你沿用前面利用Hibernate实现对话的例子，别忘了还要重构PaymentDAO），那么可以通过依赖注入把它装配到同样被重构的ManageAuction组件，并把整个操作包在单个事务中：

```
@Stateless
public class ManageAuctionBean implements ManageAuction {

    @EJB
    ItemDAO itemDAO;

    @EJB
    PaymentDAO paymentDAO;

    @TransactionAttribute(TransactionAttributeType.REQUIRED)
    public Item endAuction(Item item) {

        // Merge item
        itemDAO.makePersistent(item);

        // Set winning bid
        Bid winningBid = itemDAO.getMaxBid( item.getId() );
        item.setWinningBid(winningBid);
        item.setBuyer( winningBid.getBidder() );

        // Charge seller
        Payment payment = new Payment(item);
        paymentDAO.makePersistent(payment);
```

```
            // Notify seller and winner
            ...
            return item;
        }
        ...
    }
```

EJB容器基于你用@EJB声明的字段，注入了想要的组件，该接口具名ItemDAO和PaymentDAO这一信息足以使容器找到必需的组件。

让我们来关注事务和应用到这个组件装配的持久化上下文传播规则。

2. 传播规则

首先，系统事务是必需的，如果客户端调用ManageAuction.endAuction()，就会启动系统事务。当这个方法返回时，事务因此被容器提交。这是系统事务的范围。被调用的以及需求或者支持事务（比DAO方法）的任何其他无状态的组件方法，都继承相同的事务上下文。如果在任何这些无状态的组件中使用EntityManager，那么你正在使用的持久化上下文就自动相同，它们都被界定到系统事务。注意，这不同于你在Java SE应用程序中使用JPA的情况：EntityManager实例定义持久化上下文的范围（我们前面也详细阐述过这一点）。

当ItemDAO和PaymentDAO，这两个无状态的组件都在系统事务内部被调用时，两者都继承界定到事务的持久化上下文。容器通过当前的持久化上下文，在幕后把EntityManager实例注入到itemDAO和paymentDAO。

（在内部，如果客户端获得ManageAuction控制器，容器就从它的无状态bean池中抓取一个闲置的ManageAuctionBean实例，注入一个闲置的无状态ItemDAOBean和PaymentDAOBean，在所有需要它的组件上设置这个持久化上下文，并把ManageAuction bean句柄返回到客户端用于方法调用。这当然有点被简化了。）

对于持久化上下文的范围和传播，有正式的规则：

- 如果容器提供的（通过注入或者通过查找获得的）EntityManager第一次被调用，持久化上下文就开始了。默认情况下，它是事务范围的，并在系统事务提交或者回滚时关闭。它在事务提交时被自动清除。
- 如果容器提供的（通过注入或者通过查找获得的）EntityManager第一次被调用，持久化上下文就开始了。如果此时没有系统事务是活动的，持久化上下文就很短，并只服务于单独的方法调用。被任何这样一种方法触发的任何SQL都在自动提交模式下在一个数据库连接中执行。在这个EntityManager调用中（可能）获取的所有实体实例都立即变成脱管状态。
- 如果无状态的组件（如ItemDAO）被调用，并且调用者有一个活动的事务并被传播到被调用的组件（因为ItemDAO方法需要或者支持事务），绑定到JTA事务的任何持久化上下文就通过事务传播。
- 如果无状态的组件（如ItemDAO）被调用，且调用者没有活动的事务（例如ManageAuction.endAuction()没有启动事务），或者事务没有传播到被调用的组件（因为ItemDAO方法不需要或者不支持事务），当在无状态的组件内部调用EntityManager时，就创建了新的持久化上下文。换句话说，如果没有事务被传播，就不会发生持久化上下文的传播。

如果只阅读正式的定义，这些规则看起来就很复杂；但是，在实践中，它们变成了一种自然的行为。持久化上下文被自动界定和绑定到JTA系统事务，如果有的话——你只需要学习事务传播的规则，了解持久化上下文如何被传播。如果没有JTA系统事务，持久化上下文就服务于单个的EntityManager调用。

目前为止，你在几乎所有的例子中都使用了TransactionAttributeType.REQUIRED。这是在事务装配中应用的最常见的属性；毕竟，EJB是事务处理的编程模型。我们只介绍过一次TransactionAttributeType.NOT_SUPPORTED，是在10.3.3节讨论利用Hibernate Session的非事务数据访问的时候。

还要记住，在JPA中，你需要用非事务的数据访问来禁用长对话中持久化上下文的自动清除——还是被漏掉的FlushMode.MANUAL的问题。

现在要更深入地探讨事务属性类型，以及如何通过EJB和被扩展的持久化上下文的手工清除来实现对话。

11.4.2　利用 EJB 扩展持久化上下文

在前一节中，你只利用了被界定到JTA系统事务的持久化上下文。容器自动被注入了EntityManager，并且透明地处理持久化上下文的清除和关闭。

如果想要利用EJB和被扩展的持久化上下文实现一个对话，有两种选择：

- 可以编写一个有状态的会话bean作为对话控制器。持久化上下文可以被自动界定到这个有状态bean的生命周期，这是一种方便的方法。当这个有状态的EJB被删除时，持久化上下文就关闭。
- 可以用EntityManagerFactory自己创建EntityManager。这个EntityManager的持久化上下文是应用程序托管的——你必须手工清除和关闭它。如果在JTA事务范围内部调用它，还必须用joinTransaction()操作通知EntityManager。使用有状态的会话bean的第一种策略通常更好。

就像以前在Java SE中实现的对话一样：必须把三个步骤当作一个原子化的工作单元来完成：获取一件拍卖货品用于显示和修改，卖主账号的流动资金检查，以及最终关闭拍卖。

必须再次决定对话期间想要如何禁用被扩展的持久化上下文的自动清除，以保持原子性。可以在这两者中选择：利用FlushMode.MANUAL的Hibernate供应商扩展，以及利用正式的非事务的操作方法。

1．通过Hibernate扩展来禁用清除

先用比较容易的Hibernate扩展编写一个有状态的EJB作为对话控制器：

```
@Stateful
@TransactionAttribute(TransactionAttributeType.REQUIRED)
public class ManageAuctionBean implements ManageAuction {

    @PersistenceContext(
      type = PersistenceContextType.EXTENDED,
      properties = @PersistenceProperty(
                    name="org.hibernate.flushMode",
```

```
                    value="MANUAL")
    )
    EntityManager em;

    public Item getAuction(Long itemId) {
        return em.find(Item.class, itemId);
    }

    public boolean sellerHasEnoughMoney(User seller) {
        boolean sellerCanAffordIt = (Boolean)
            em.createQuery("select...").getSingleResult();
        return sellerCanAffordIt;
    }

    @Remove
    public void endAuction(Item item, User buyer) {
        // Set winning bid
        // Charge seller
        // Notify seller and winner
        item.setBuyer(...);

        em.flush();
    }
}
```

这个bean实现了ManageAuction接口（我们没有必要介绍这个接口）的三种方法。首先，这是个有状态的EJB；容器给特定的客户端创建和保存一个实例。当客户端第一次获得这个EJB的句柄时，创建一个新实例，容器注入新的被扩展的持久化上下文。持久化上下文现在被绑定到EJB实例的生命周期，并在标记为@Remove的方法返回时关闭。注意你如何像读对话的历史一样一步一步地读EJB的方法。可以用@Remove注解几个方法；例如，可以添加cancel()方法来撤销所有的对话步骤。对于对话来说，这是个强大的、方便的编程模型，EJB 3.0中全部免费内建了。

接下来是自动清除的问题。ManageAuctionBean的所有方法都需要事务；在类级别上需要声明这一点。sellerHasEnoughMoney()方法（对话中的第二步），在执行查询之前清除持久化上下文，并在这个方法的事务返回时再次清除。为了防止这种情况，需要声明被注入的持久化上下文应该在FlushMode.MANUAL这个Hibernate扩展中。现在，每当你想要把列队中等待的SQL DML编写到数据库时，清除持久化上下文就是你的责任了——你只在对话结束时进行一次。

事务装配现在从持久化引擎的清除行为中解耦出来了。

2. 通过禁用事务来禁用清除

根据EJB 3.0规范，正式的解决方案混合了这两个关注点。通过使对话的所有步骤（除了最后一个之外）变成非事务来防止自动的清除：

```
@Stateful
@TransactionAttribute(TransactionAttributeType.NOT_SUPPORTED)
public class ManageAuctionBean implements ManageAuction {

    @PersistenceContext(type = PersistenceContextType.EXTENDED)
    EntityManager em;

    public Item getAuction(Long itemId) {
        return em.find(Item.class, itemId);
```

```
    }

    public boolean sellerHasEnoughMoney(User seller) {
        boolean sellerCanAffordIt = (Boolean)
            em.createQuery("select...").getSingleResult();
        return sellerCanAffordIt;
    }

    @Remove
    @TransactionAttribute(TransactionAttributeType.REQUIRED)
    public void endAuction(Item item, User buyer) {
        // Set winning bid
        // Charge seller
        // Notify seller and winner
        item.setBuyer(...);
    }
}
```

在这个实现中，你转换到一个用于所有方法的不同默认：TransactionAttributeType.NOT_SUPPORTED，并且只有endAuction()方法需要事务。这最后一个事务也在提交时清除持久化上下文。

现在不用事务调用EntityManager的所有方法都在自动提交模式（第10章讨论过）下有效地运行着。

3. 复杂的事务装配

你现在已经使用过了几个不同的TransactionAttributeType注解，请见表11-1中可用选项的完整清单。

最常用的事务属性类型是REQUIRED，这是所有无状态和有状态EJB方法的默认值。为了对有状态会话bean中的一个方法禁用被扩展的持久化上下文的自动清除，要转换到NOT_SUPPORTED甚至NEVER。

表11-1　EJB 3.0声明式事务属性类型

属性名称	描　述
REQUIRED	必须通过事务上下文调用的一种方法。如果客户端没有事务上下文，容器就会启动事务，并获取这个事务所用的所有资源（数据源等）。如果这个方法调用其他的事务组件，事务就会被传播。方法返回时，容器在结果被发送到客户端之前提交事务
NOT_SUPPORTED	如果方法在从客户端传播的事务上下文内部调用，调用者的事务就被暂停，并在方法返回时重新激活。如果调用者没有事务上下文，就没有给该方法启动任何事务。不是全部所用的资源都通过事务得到获取（发生自动提交）
SUPPORTS	如果方法在从客户端传播的事务上下文内部调用，它就把这个事务上下文与跟REQUIRED相同的结果联结起来。如果调用者没有事务上下文，就没有事务被启动，结果与NOT_SUPPORTED的相同。这个事务属性类型应该只用于可以正确处理这两种情况的方法
REQUIRES_NEW	方法始终在新的事务上下文内部执行，结果和行为与使用REQUIRED的相同。任何被传播的客户端事务都被暂停，并在方法返回且新事务完成时继续
MANDATORY	方法必须通过一个活动的事务上下文被调用。然后它联结这个事务上下文，并且在需要时进一步传播。如果调用时没有出现事务上下文，就抛出异常
NEVER	它与MANDATORY相反。如果方法通过一个活动的事务上下文被调用，就抛出异常

当利用有状态的EJB设计对话时，或者如果你想要混合无状态和有状态的组件时，必须知道事务上下文和持久化上下文的传播规则：

- 如果具有被扩展的持久化上下文的有状态会话bean调用（实际上实例化）了另一个也有持久化上下文的有状态会话bean，第二个有状态会话bean就从调用者处继承持久化上下文。持久化上下文的生命周期由第一个有状态会话bean决定；它在两个会话bean都被移除时关闭。如果涉及更多有状态的会话bean，这个行为就是递归的。这个行为还不依赖于任何事务规则和事务传播。
- 如果EntityManager用于有着绑定的被扩展持久化上下文的有状态会话bean的一个方法中，并且这个方法需要/支持客户端的JTA事务，那么如果方法的调用程序也通过它的事务传播不同的持久化上下文，就会出现异常。（这是一个罕见的设计问题。）

这些规则隐含的意思是：想象有状态的ManageAuction控制器没有直接调用EntityManager，但是它委托给其他的组件（例如数据访问对象）。它仍然有并且负责被扩展的持久化上下文，虽然EntityManager从不被直接使用。这个持久化上下文必须被传播到所有其他被调用的组件（例如，ItemDAO和PaymentDAO）里面。

如果把DAO实现为无状态的会话bean，就像你之前所做的那样，那么如果它们从有状态的控制器的一个非事务的方法中被调用，它们就不继承被扩展的持久化上下文。这又是有状态的调用DAO的控制器：

```
@Stateful
@TransactionAttribute(TransactionAttributeType.NOT_SUPPORTED)
public class ManageAuctionBean implements ManageAuction {

    @PersistenceContext(type = PersistenceContextType.EXTENDED)
    EntityManager em;

    @EJB
    PaymentDAO paymentDAO;

    public boolean sellerHasEnoughMoney(User seller) {
        return paymentDAO.checkLiquidity(seller);
    }

    ...
}
```

sellerHashEnoughMoney()方法没有启动事务，以便在对话中间的提交中自动避免持久化上下文的清除。问题是对DAO这个无状态EJB的调用。为了让持久化上下文传播到无状态EJB调用中，你需要传播一个事务上下文。如果paymentDAO.checkLiquidity()使用EntityManager，它就得到了新的持久化上下文！

第二个问题在于PaymentDAO无状态会话bean：

```
@Stateless
public class PaymentDAO {

    @PersistenceContext
    EntityManager em;
    public boolean checkLiquidity(User u) {
```

```
        boolean hasMoney = (Boolean)
            em.createQuery("select...").getSingleResult();
        return hasMoney;
    }
    ...
}
```

由于调用checkLiquidity()方法时，没有持久化上下文被传播到它里面去，因此就会创建新的持久化上下文用来服务于这个单独的操作。这是“每个操作一个会话”的反模式！更糟糕的是，现在你在一个请求和这个对话中有两个（或者更多）持久化上下文，并将遇到数据别名的问题（没有同一性范围保证）。

如果你把DAO实现为有状态的会话bean，它们就从调用有状态会话bean控制器中继承持久化上下文。在这种情况下，持久化上下文就通过实例化（而不是通过事务传播）而被传播。

如果把控制器写成一个有状态的会话bean，就把DAO写成有状态的EJB。这个问题是遗漏的FlushMode.MANUAL另一个讨厌的副作用，它会严重影响到你的设计方式和分层应用程序。建议你在EJB 3.0（或者3.1?）规范修正之前，都依赖Hibernate扩展。通过FlushMode.MANUAL，控制器不一定使用TransactionAttributeType.NOT_SUPPORTED，并且持久化上下文始终与事务一起传播（还可以轻松地混合无状态和有状态的EJB调用）。

我们会在第16章回到这个问题，那时我们将编写更加复杂的应用程序代码和DAO。

11.5　小结

在本章中，你利用Hibernate、JPA和EJB 3.0组件实现了对话。你学习了如何传播当前的Hibernate Session和持久化上下文来创建更加复杂的分层应用程序，而不渗透关注点。你也已经知道持久化上下文传播是EJB 3.0深入整合的一项特性，以及持久化上下文可以被轻松地绑定到JTA（或者CMT）事务范围。你还学习了FlushMode.MANUAL这项Hibernate特性如何独立地从事务装配中禁用持久化上下文的清除。

表11-2展现了可以用来比较原生的Hibernate特性和Java Persistence的一个概括。

表11-2　第11章中Hibernate和JPA对照表

Hibernate Core	Java Persistence和EJB 3.0
持久化上下文传播在Java SE和Java EE中可以通过线程或者JTA事务的绑定而可用。持久化上下文要么被界定到事务，要么由应用程序托管	Java Persistence只对Java EE标准化了持久化上下文传播模型（它与EJB 3.0组件进行了深入整合）。很好地定义了持久化上下文范围到事务或者有状态会话bean
Hibernate支持利用脱管对象实现对话，这些对象可以在对话期间被重附或者合并	Java Persistence标准化了脱管对象的合并，但是不支持重附
Hibernate支持用FlushMode.MANUAL选项对长对话禁用持久化上下文的自动清除	禁用被扩展的持久化上下文的自动清除，需要非事务的事件处理（对应用程序的设计和分层有严格的限制）或者需要Hibernate退回到FlushMode.MANUAL

第12章将探讨每当你要利用更复杂、更大的数据集时应该依赖的各种选项。你会了解到传播性持久化如何利用Hibernate的级联模型、如何有效地执行批量和大批量操作，以及在加载和保存对象时如何钩入和操作Hibernate的默认行为。

第12章

有效修改对象

本章内容

- 传播性状态变化
- 批量和大批量处理
- 持久化生命周期的拦截

本章介绍如何使数据操作更有效率。我们优化和减少保存对象所必需的代码量，并讨论最有效的处理选项。你应该熟悉基础的对象状态和持久化接口；要理解本章必须先阅读前面的章节。

首先，介绍传播性持久化如何使你更容易地使用复杂的对象网络。你可以在Hibernate和Java Persistence应用程序中启用的级联选项，明显减少了在没有这些级联选项时同时插入、更新或者删除几个对象所必需的代码量。

然后讨论如何最好地处理大的数据集，是通过应用程序中的批量操作，还是通过直接在数据库中执行的大批量操作。

最后，介绍数据过滤和拦截，这两者都提供透明地钩入Hibernate引擎内部加载和保存的过程。这些特性让你影响或者参与对象的生命周期，而不用编写复杂的应用程序代码，不用绑定领域模型到持久化机制。

让我们从传播性持久化开始，同时保存多个对象。

12.1 传播性持久化

真正重要的应用程序不仅仅使用单独的对象，还使用对象网络。当应用程序操作持久化的对象网络时，结果可能会是一张由持久化的、脱管的和瞬时的实例组成的对象图。传播性持久化是一种允许你把持久化自动传播到瞬时的和脱管的子图的方法。

例如，如果你把一个刚被实例化的`Category`添加到categories的已经持久化的层次结构，它应该自动变成持久化，而不必调用`save()`或者`persist()`。6.4节映射`Bid`和`Item`之间的父/子关系时，介绍了一个稍微不同的例子。在这种情况下，当出价被添加到一件货品时，它们不仅自动变成持久化，而且当胜出货品被删除时它们也自动被删除。你有效地使`Bid`变成了完全依赖另一个实体`Item`的实体（`Bid`实体不是值类型，它仍然支持共享引用）。

传播性持久化有不止一个模型。最著名的是按可到达性持久化（persistence by reachability），我们先来讨论它。虽然有些基本原理是一样的，但Hibernate还是用了它自己的更为强大的模型，稍后你会看到。对于Java Persistence也一样，它也有传播性持久化的概念，并且具有Hibernate原生提供的几乎所有选项。

12.1.1　按可到达性持久化

每当应用程序从已经持久化的实例中创建对其中一个实例的引用时，如果任何实例变成了持久化，对象持久层便是在实现按可到达性持久化。这种行为如图12-1中的对象图（注意这不是一张类图）所示。

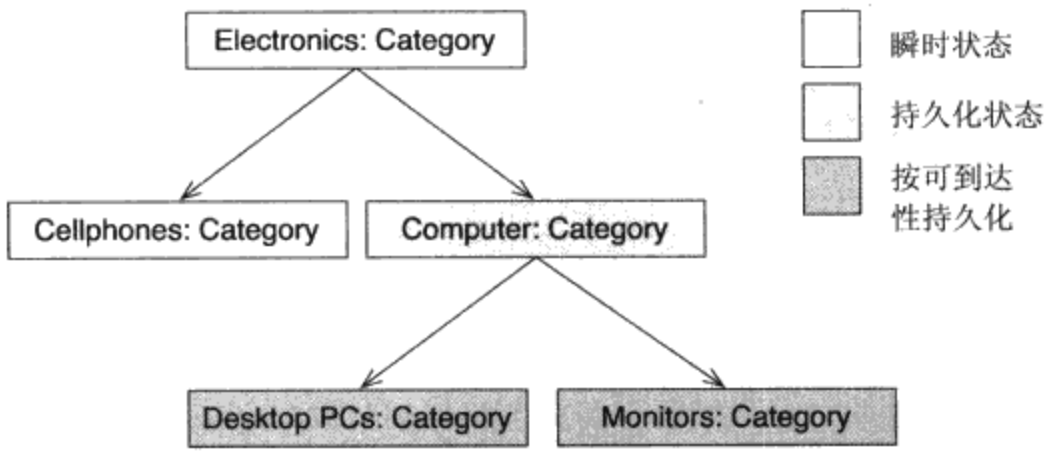


图12-1　包含根持久化对象的按可到达性持久化

在这个例子中，Computer是一个持久化对象。对象Desktop PCs和Monitors也是持久化对象：它们可以从Computer `Category`实例到达。Electronics和Cellphones是瞬时状态。注意我们假设导航只可能对子类进行，对父类则不行——例如，可以调用`computer.getChildCategories()`。按可到达性持久化是一种递归的算法。所有可以从持久化实例到达的对象都在原始实例被变成持久化时、或者就在内存状态与数据库同步之前变成持久化。

按可到达性持久化保证了参照完整性；任何对象图都可以通过加载持久化根对象被完全重新创建。应用程序可以逐个关联地穿过对象图，从来不必担心实例的持久化状态。（SQL数据库有一种不同的参照完整性的方法，它依赖声明和过程的约束来侦测行为异常的应用程序。）

在最纯粹的按可到达性持久化的形式中，数据库有一些顶级对象或者根对象，所有持久化对象都可以从那里到达。理想情况下，如果它不能通过引用从根持久化对象到达，实例应该变成瞬时状态并且就要从数据库中被删除。

Hibernate和其他的ORM解决方案都没有实现这一点——事实上，在SQL数据库中没有类似的根持久化对象，也没有持久化垃圾收集器可以侦测没有被引用的实例。面向对象的数据存储可以实现垃圾收集算法，类似于JVM给内存对象实现的那种。但是这个选项在ORM领域中不可用；扫描所有的表来查找没有被引用的行，执行起来并不令人满意。

因此，按可到达性持久化最多是个不彻底的解决方案。它帮助你使瞬时对象变成持久化，并把它们的状态传播到数据库，而不需要对持久化管理器的诸多调用。然而，至少在SQL数据库和

ORM的上下文中，它并不是使持久化对象变成瞬时（把它们的状态从数据库中移除）这个问题的完整解决方案。这样就出现了更大的难题。当你移除一个对象时，却无法移除所有可到达的实例——其他的持久化实例可能仍然持有对它们的引用（记住实体可以被共享）。你甚至无法安全地移除那些没有被内存中的任何持久化对象引用的实例；内存中的实例只在数据库中表示所有对象的一个小子集。

来看一下Hibernate更灵活的传播性持久化模型。

12.1.2 把级联应用到关联

Hibernate的传播性持久化模型使用与按可到达性持久化相同的基本概念：检查对象关联来确定传播性状态。此外，Hibernate还允许你给每个关联映射指定一种级联样式（cascade style），它为所有的状态转变提供了更多灵活性和更细粒度的控制。Hibernate读取被声明的样式，并自动把操作级联到被关联的对象。

默认情况下，当搜索瞬时的或者脱管的对象时，Hibernate并不导航关联，因此保存、删除、重附、合并等，`Category`对于被父类别的`childCategories`集合引用的任何子类别都没有影响。这与按可到达性持久化的默认行为相反。对于特定的关联而言，如果你希望启用传播性持久化，就必须在映射元数据中覆盖这个默认。

这些设置被称作级联的选项。它们使用XML和注解语法，可用于每一种实体关联映射（一对一、一对多、多对多）。请见表12-1中所有设置的清单和每种选项的描述。

在XML映射元数据中，把`cascade="..."`属性放在`<one-to-one>`或者`<many-to-one>`映射元素中，来启用传播性状态变化。所有的集合映射（`<set>`、`<bag>`、`<list>`和`<map>`）都支持`cascade`属性。然而，`delete-orphan`设置只适用于集合。很显然，你永远不必给引用值类型类的集合启用传播性持久化——这里被关联对象的生命周期是依赖和隐式的。依赖型生命周期的细粒度控制只与实体之间的关联相关，且只对它们可用。

表12-1 Hibernate和Java Persistence实体关联级联选项

XML属性	注解描述
None	（默认） Hibernate忽略关联
save-update	`Org.hibernate.annotations.CascadeType.SAVE_UPDATE` 当`Session`被清除，且对象被传递到`save()`或者`update()`时，Hibernate就导航关联，并保存刚刚被实例化的瞬时实例和把变化持久化到脱管状态的实例
persist	`javax.persistence.CascadeType.PERSIST` 当对象传递到`persist()`时，Hibernate使任何被关联的瞬时实例都变成持久化。如果使用原生的Hibernate，级联则只在调用时发生。如果使用`EntityManager`模块，这项操作则在持久化上下文清除时被级联
merge	`javax.persistence.CascadeType.MERGE` Hibernate导航关联，并在对象传递到`merge()`时，通过相当的持久化实例合并被关联的脱管实例。可到达的瞬时实例被变成持久化
delete	`org.hibernate.annotations.CascadeType.DELETE` 当对象被传递到`delete()`或者`remove()`时，Hibernate就导航关联，并删除被关联的持久化实例

（续）

XML属性	注解描述
remove	javax.persistence.CascadeType.REMOVE 当对象被传递到remove()或者delete()时，这个选项就把级联删除启用到被关联的持久化实例
lock	org.hibernate.annotations.CascadeType.LOCK 这个选项把lock()操作级联到被关联的实例，如果对象是脱管的，就把它们重附到持久化上下文。注意LockMode没有被级联；Hibernate假设你在被关联对象上不想要悲观锁——例如，因为根对象上的悲观锁已经足以避免并发修改
replicate	org.hibernate.annotations.CascadeType.REPLICATE Hibernate导航该关联，并把replicate()操作级联到被关联的对象
evict	org.hibernate.annotations.CascadeType.EVICT 当对象被传递到Hibernate Session中的evict()时，Hibernate从持久化上下文清除被关联的对象
refresh	javax.persistence.CascadeType.REFRESH 当对象被传递到refresh()时，Hibernate从数据库中重新读取被关联对象的状态
all	javax.persistence.CascadeType.ALL 这项设置包括并启用前面列出的所有级联选项
deleteorphan	org.hibernate.annotations.CascadeType.DELETE_ORPHAN 当被关联的对象从关联（即集合中）被移除时，这项额外的特殊设置启用了它们的删除。如果你在一个实体集合中启用这项设置，就等于告诉Hibernate：被关联的对象没有共享的引用，且当引用从集合中被移除时，它可以被安全地删除

常见问题　cascade和inverse之间是什么关系？没有关系；两者是不同的概念。关联的非反向端被用来在数据库中生成管理关联的SQL语句（若干外键列的插入和更新）。级联启用了跨实体类关联的传播性对象状态变化。

这里有几个在XML映射文件中的级联选项示例。注意这段代码不是来自单个实体映射或者单个类，只是用来举例说明：

```
<many-to-one name="parent"
             column="PARENT_CATEGORY_ID"
             class="Category"
             cascade="save-update, persist, merge"/>

...

<one-to-one name="shippingAddress"
            class="Address"
            cascade="save-update, lock"/>

...

<set name="bids" cascade="all, delete-orphan"
     inverse="true">
    <key column ="ITEM_ID"/>
    <one-to-many class="Bid"/>
</set>
```

如你所见，几个级联选项可以合并，并作为一个以逗号分隔的列表应用到特定的关联。还要注意delete-orphan没有包括在all中。

级联选项通过注解以两种可能的方式进行声明。第一，所有的关联映射注解（@ManyToOne、@OneToOne、@OneToMany和@ManyToMany）都支持cascade属性。这项属性的值是单个或者一列javax.persistence.CascadeType值。例如，通过注解来完成的XML示例映射看起来像下面这样：

```
@ManyToOne(cascade = { CascadeType.PERSIST, CascadeType.MERGE })
@JoinColumn(name = "PARENT_CATEGORY_ID", nullable = true)
private Category parent;

...

@OneToMany(cascade = CascadeType.ALL)
private Set<Bid> bids = new HashSet<Bid>();
```

很显然，并非所有的级联类型在标准的javax.persistence包中都可用。只有与EntityManager操作相关的级联选项，如persist()和merge()被标准化。必须使用一个Hibernate扩展注解来应用任何只适用于Hibernate的级联选项：

```
@ManyToOne(cascade = { CascadeType.PERSIST, CascadeType.MERGE })
@org.hibernate.annotations.Cascade(
    org.hibernate.annotations.CascadeType.SAVE_UPDATE
)
@JoinColumn(name = "PARENT_CATEGORY_ID", nullable = true)
private Category parent;

...

@OneToOne
@org.hibernate.annotations.Cascade({
    org.hibernate.annotations.CascadeType.SAVE_UPDATE,
    org.hibernate.annotations.CascadeType.LOCK
})
@PrimaryKeyJoinColumn
private Address shippingAddress;

...

@OneToMany(cascade = CascadeType.ALL)
@org.hibernate.annotations.Cascade(
    org.hibernate.annotations.CascadeType.DELETE_ORPHAN
)
private Set<Bid> bids = new HashSet<Bid>();
```

Hibernate扩展级联选项可以用作已经在关联注解中设置的选项的补充（第一个和最后一个例子），如果没有应用标准的选项，也可以用作一项独立的设置（第二个例子）。

Hibernate的关联级别级联样式模型比按可到达性持久化更丰富且更不安全。Hibernate没有像按可到达性持久化一样对参照完整性提供强大的保证。反之，Hibernate委托了部分参照完整性关注点给底层SQL数据库的外键约束。

这一设计决策有一个很好的解释：它允许Hibernate应用程序有效地使用脱管对象，因为你可以在关联级别控制脱管对象图的重附和合并。但是级联选项不是只可用来避免不必要的重附或者

合并：每当需要同时处理多个对象时，它们也很有用。

让我们用一些示例关联映射详细阐述传播性状态概念。我们建议你按顺序阅读接下来的小节，因为每个示例都是建立在前一个的基础上。

12.1.3　使用传播性状态

CaveatEmptor管理员能够创建新的类别，重新命名类别，并在分类层次结构中四处移动子类别。这个结构如图12-2所示。

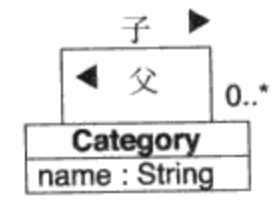


图12-2　Category类包含对自身的关联

现在，用XML映射这个类和关联：

```
<class name="Category" table="CATEGORY">
    ...
    <property name="name" column="CATEGORY_NAME"/>

    <many-to-one name="parentCategory"
                 class="Category"
                 column="PARENT_CATEGORY_ID"
                 cascade="none"/>

    <set name="childCategories"
         table="CATEGORY"
         cascade="save-update"
         inverse="true">
        <key column="PARENT_CATEGORY_ID"/>
        <one-to-many class="Category"/>
    </set>
    ...
</class>
```

这是一个递归的双向一对多关联。一值端通过<many-to-one>元素和包含<set>的Set类型属性被映射。两者都指向相同的外键列PARENT_CATEGORY_ID。所有的列都处在同一张表CATEGORY中。

1．创建新类别

假设你创建新的Category，作为Computer的子类别；请见图12-3。

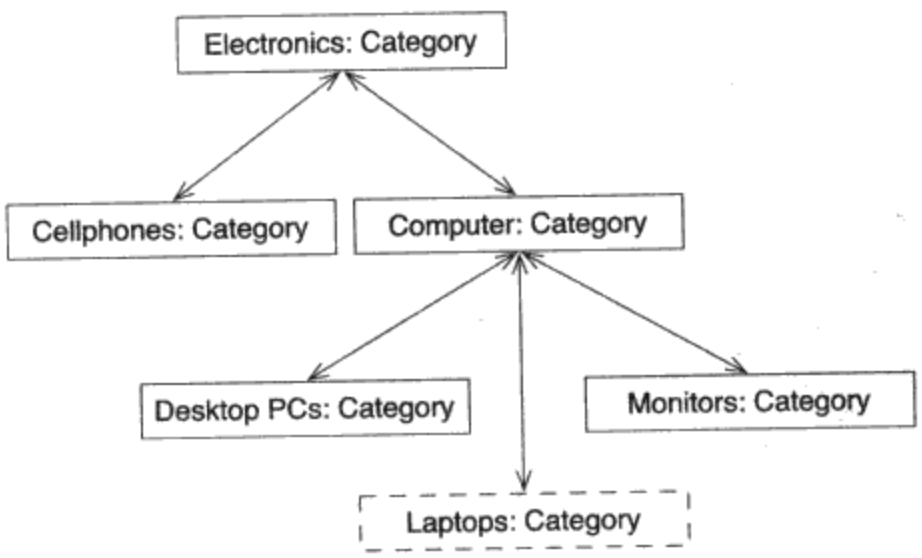


图12-3　把新的Category添加到对象图

创建这个新的Laptops对象并把它保存在数据库中有好几种方法。你可以回到数据库并获取新Laptops类别，将要所属的Computer类别添加这个新类别，并提交事务：

```
Session session = sessionFactory.openSession();
Transaction tx = session.beginTransaction();

Category computer =
            (Category) session.load(Category.class, computerId);

Category laptops = new Category("Laptops");

computer.getChildCategories().add(laptops);
laptops.setParentCategory(computer);

tx.commit();
session.close();
```

computer实例是持久化的（注意如何将load()与代理共用，并避免数据库命中），且childCategories关联启用了级联保存。因而，这段代码在tx.commit()被调用时导致新的laptops类别变成持久化，就像Hibernate把持久化状态级联到computer的childCategories集合元素一样。当持久化上下文被清除，且排列INSERT语句时，Hibernate就会检查对象的状态和它们的关系。

2. 以脱管方式创建新类别

再次完成同样的事，但这次是在持久化上下文范围之外创建Computer和Laptops之间的连接：

```
Category computer =
            (Category) session.get() // Loaded in previous Session

Category laptops = new Category("Laptops");

computer.getChildCategories().add(laptops);
laptops.setParentCategory(computer);
```

你现在有了脱管的、完全实例化（没有代理）的computer对象，它在前一个Session中加载，并与新的瞬时laptops对象关联（反之亦然）。你通过把这个新对象保存在第二个Hibernate Session——一个新的持久化上下文中，使在对象上所做的变化持久化：

```
Session session = sessionFactory.openSession();
Transaction tx = session.beginTransaction();

// Persist one new category and the link to its parent category
session.save(laptops);

tx.commit();
session.close();
```

Hibernate检查laptops.parentCategory对象的数据库标识符属性，并在数据库中正确创建对Computer类别的引用。Hibernate把父标识符值插入到CATEGORY中新Laptops行的外键字段。

在这个例子中，你无法给computer获得脱管的代理，因为computer.getChildCategories()将触发代理的初始化，你会看到LazyInitializationException: Session已经关闭。你无法在脱管状态下跨过未被初始化的边界来导航对象图。

由于你给parentCategory关联定义了cascade="none"，Hibernate忽略对层次结构

（Computer，Electronics）中任何其他类所做的改变！它不把对save()的调用级联到被这个关联引用的实体。如果在parentCategory的<many-to-one>映射中启用了cascade="save-update"，Hibernate就会在内存中导航整个对象图，使所有的实例都与数据库同步。这是你想要避免的一项明显的过载。

在这个例子中，对于parentCategory关联，你既不需要也不想要传播性持久化。

3．用传播性持久化保存几个新实例

为什么要有级联操作？如前一个例子所示，你可以不用任何级联映射保存laptop对象。那么，考虑下列案例：

```
Category computer = ... // Loaded in a previous Session

Category laptops = new Category("Laptops");
Category laptopUltraPortable = new Category("Ultra-Portable");
Category laptopTabletPCs = new Category("Tablet PCs");

laptops.addChildCategory(laptopUltraPortable);
laptops.addChildCategory(laptopTabletPCs);

computer.addChildCategory(laptops);
```

（注意，便利方法addChildCategory()在一个调用中设置了关联连接的两端，如本书前面所述。）

如果必须把这三个新类别的每一个保存在单独的新Session中，这可不是我们想要的。幸运的是，因为你用cascade="save-update"映射了childCategories关联（集合），因此不必那么辛苦。与前面所示的代码一样，它保存了单个Laptops类别，将在一个新Session中保存所有这三个新类别：

```
Session session = sessionFactory.openSession();
Transaction tx = session.beginTransaction();

// Persist all three new Category instances
session.save(laptops);

tx.commit();
session.close();
```

你或许想知道为什么级联样式被称作cascade="save-update"，而不仅仅是cascade="save"。前面刚刚使所有三个类别变成持久化，假设在接下来的事件中，你在Session外部对类别层次结构进行下面的改变（你正再一次使用脱管对象）：

```
laptops.setName("Laptop Computers");                    // Modify
laptopUltraPortable.setName("Ultra-Portable Notebooks"); // Modify
laptopTabletPCs.setName("Tablet Computers");            // Modify

Category laptopBags = new Category("Laptop Bags");
laptops.addChildCategory(laptopBags);                   // Add
```

你添加一个新类别（laptopBags）作为Laptops类别的一个子类别，并修改所有三个现有的类别。下列代码把所有这些变化传播到数据库：

```
Session session = sessionFactory.openSession();
Transaction tx = session.beginTransaction();
```

```
// Update three old Category instances and insert the new one
session.saveOrUpdate(laptops);

tx.commit();
session.close();
```

由于你在childCategories集合中指定了cascade="save-update"，Hibernate确定需要用什么把对象持久化到数据库。在这个例子中，它排列三条SQL UPDATE语句（用于laptops、laptopUltraPortable和laptopTablePCs）和一条INSERT（用于laptopBags）。saveOrUpdate()方法告诉Hibernate把实例的状态传播到数据库：如果该实例是一个新的瞬时实例，就通过创建一个新的数据库行；如果该实例是一个脱管实例，就通过更新现有的行。

更有经验的Hibernate用户专门使用saveOrUpdate()；它更容易让Hibernate决定什么是新的以及什么是旧的，尤其在混合状态的更复杂对象图中的时候。这个专门的saveOrUpdate()的唯一缺点在于，它在数据库中没有触发SELECT的情况下，有时无法猜测一个实例是旧的还是新的——例如，当类用一个自然的复合键映射，并且没有版本或者时间戳属性的时候。

Hibernate如何发现哪个实例是旧的以及哪个实例是新的呢？有许多选项可用。Hibernate假设实例是未被保存的瞬时实例，如果：

- 标识符属性为null。
- 版本或者时间戳属性（如果存在的话）为null。
- 同一个持久化类的新实例（由Hibernate内部创建）有着与指定实例相同的数据库标识符值。
- 你在映射文档中给类提供一个unsaved-value，并且标识符属性的值匹配。unsaved-value属性对于版本和时间戳映射元素也可用。
- 包含相同标识符值的实体数据不在二级高速缓存中。
- 你提供org.hibernate.Interceptor的一个实现，并在检查完你自己代码中的实例之后，从Interceptor.isUnsaved()返回Boolean.TRUE。

在CaveatEmptor领域模型中，你随处使用可为空的类型java.lang.LONG作为标识符属性类型。因为你正在使用生成的合成标识符，这样就解决了问题。新实例有一个null标识符属性值，因此Hibernate把它们当作瞬时实例对待。脱管实例有一个非空的标识符值，因此Hibernate对它们做相应的处理。

几乎没有必要定制创建到Hibernate中去的自动侦测子程序。saveOrUpdate()方法始终知道如何处理指定的对象（或者任何可到达的对象，如果给关联启用save-update的级联的话）。然而，如果你使用自然的复合键，并且实体中没有版本或者时间戳属性，Hibernate就必须用SELECT命中数据库，查看是否已经存在包含相同复合标识符的行。换句话说，建议你通常使用saveOrUpdate()而不是单独的save()或者update()方法，Hibernate很聪明，足以正确地处理，并使传播性的“无论新旧，所有这些都应该处于持久化状态”更易于处理。

现在已经讨论了Hibernate中基本的传播性持久化选项，通过尽可能少的代码行保存实例和重附脱管实例。大多数的其他级联选项都相当容易理解：persist、lock、replicate和evict完成你预期的工作——它们使特定的Session操作具有传播性。merge级联选项实际上有着与save-update一样的效果。

事实表明，对象删除更难以掌握；delete-orphan设置尤其对刚接触Hibernate的用户造成混淆。这不是因为它复杂，而是因为许多Java开发人员往往忘了他们正在使用一个指针网络。

4．考虑传播性删除

想象你想要删除Category对象。必须把这个对象传递到Session中的delete()方法；现在它处于移除状态，并且当清除或者提交持久化上下文时，将离开数据库。然而，如果任何其他的Category当时持有对被删除行的引用（可能由于它仍然作为其他类别的父类别被引用），就会得到一个外键约束违例（foreign key constraint violation）。

你有责任在删除实例之前删除对Category的所有链接。这是支持共享引用的实体的一般行为。当自己的实体实例被删除时，实体实例的任何值类型属性（或者组件）值也自动被删除。如果你从自己的集合中移除引用，值类型集合元素（例如，Item的Image对象集合）也会被删除。

在某些情况下，你想要通过从集合中移除引用来删除实体实例。换句话说，你可以保证，一旦从集合中移除对这个实体的引用，就不会存在其他的引用。因此，Hibernate可以在已经移除了最后一个引用之后安全地删除实体。Hibernate假设没有引用的孤儿实体应当被删除。在示例的领域模型中，你在Item的映射中给bids的集合启用了这个特殊的级联样式（它只对集合可用）：

```
<set name="bids"
     cascade="all, delete-orphan"
     inverse="true">
    <key column="ITEM_ID"/>
    <one-to-many class="Bid"/>
</set>
```

现在可以通过从这个集合中移除它们来删除Bid对象了——例如，在脱管状态下：

```
Item anItem = ... // Loaded in previous Session

anItem.getBids().remove(aBid);
anItem.getBids().remove(anotherBid);

Session session = sessionFactory.openSession();
Transaction tx = session.beginTransaction();

session.saveOrUpdate(anItem);

tx.commit();
session.close();
```

如果没有启用delete-orphan选项，就必须在从集合中移除最后一个对它们的引用之后，显式地删除Bid实例：

```
Item anItem = ... // Loaded in previous Session

anItem.getBids().remove(aBid);
anItem.getBids().remove(anotherBid);

Session session = sessionFactory.openSession();
Transaction tx = session.beginTransaction();

session.delete(aBid);
session.delete(anotherBid);

session.saveOrUpdate(anItem);
```

```
tx.commit();
session.close();
```

孤儿的自动删除为你节省了两行代码——不合时宜的两行代码。如果没有孤儿删除，则你必须记住想要删除的所有Bid对象——从集合中移除元素的代码，经常与执行delete()操作的代码处在不同的层中。通过启用孤儿删除，可以从集合中移除孤儿，并且Hibernate将假设它们不再被任何其他的实体引用。再次提醒，如果映射一个组件的集合，孤儿删除就是隐式的；额外的选项只与实体引用的集合相关（通常是<one-to-many>）。

Java Persistence和EJB 3.0也支持跨实体关联的传播性状态变化。标准的级联选项与Hibernate的类似，因此可以很容易地学会它们。

12.1.4 利用 JPA 的传播性关联

Java Persistence规范对于启用了级联对象操作的实体关联支持注解。就像在原生的Hibernate中一样，每个EntityManager操作都有一个相当的级联样式。例如，考虑通过注解映射的Category树（父/子关联）：

```
@Entity
public class Category {

    private String name;

    @ManyToOne
    public Category parentCategory;

    @OneToMany(mappedBy = "parentCategory",
               cascade = { CascadeType.PERSIST,
                           CascadeType.MERGE }
               )
    public Set<Category> childCategories = new HashSet<Category>();
    ...
}
```

你给persist()和merge()操作启用了标准的级联选项。现在可以在持久化和脱管状态下创建和修改Category实例，就像之前通过原生的Hibernate所做的那样：

```
Category computer = ... // Loaded in a previous persistence context

Category laptops = new Category("Laptops");
Category laptopUltraPortable = new Category("Ultra-Portable");
Category laptopTabletPCs = new Category("Tablet PCs");

laptops.addChildCategory(laptopUltraPortable);
laptops.addChildCategory(laptopTabletPCs);

computer.setName("Desktops and Laptops");
computer.addChildCategory(laptops);
EntityManager em = emf.createEntityManager();
EntityTransaction tx = em.getTransaction();
tx.begin();

computer = em.merge(computer);
```

```
tx.commit();
em.close();
```

对merge()的单个调用，使得所有修改和新增都变成了持久化。记住，merge()与重附不同：在合并之后，它返回一个新值（它应该被绑定到作为句柄的当前变量）到当前状态。

有些级联选项没有被标准化，而是Hibernate特有的。如果你正在使用Session API，或者如果你想要一个额外的设置用于EntityManager（例如org.hibernate.annotations.CascadeType.DELETE_ORPHAN），就映射这些注解（它们全部都在org.hibernate.annotations包中）。但是要注意，在其他的纯JPA应用程序中的定制级联选项引入了隐式的对象状态改变（它可能难以与不想要它们的人们进行通信）。

在前面几节中，我们已经探讨了通过Hibernate XML映射文件和Java Persistence注解的实体关联级联选项。利用传播性状态改变，你通过让Hibernate把修改导航和级联到被关联的对象，节省了代码行。建议你把领域模型所处的关联当作传播性状态改变的备选，然后通过级联选项实现它。在实践中，如果你也在领域模型［带有版型（stereotype）］的UML图中，或者在开发人员之间共享的任何其他类似文档中记下了级联选项，它是相当有帮助的。这么做改善了开发团队内部的交流，因为每个人都知道哪些操作和关联意味着级联状态的改变。

传播性持久化不是你可以通过单个操作来操作许多对象的唯一方法。许多应用程序都必须修改大对象集：例如，想象你必须在50 000个Item对象中设置标记。这是在数据库中直接执行得最好的大批量操作。

12.2　大批量和批量操作

你用ORM把数据移到应用层中，以便可以用一种面向对象的编程语言来处理这一数据。如果你正在实现一个多用户的在线事务处理应用程序，每个工作单元中都包含从少量至中等容量的数据集，那么这是一种好策略。

另一方面，需要大量数据的操作最好不要在应用层中执行。你应该把操作移近数据的位置，而不是其他地方。在SQL系统中，DML语句UPDATE和DELETE直接在数据库中执行，如果必须实现一个涉及上千行的操作，这些通常就足够了。更复杂的操作可能需要更复杂的过程在数据库内部运行；因此，你应该把存储过程当作一种可能的策略。

在Hibernate和Java Persistence应用程序中，你始终可以退回到JDBC和SQL。本节将介绍如何避免这一点，以及如何通过Hibernate和JPA执行大批量和批量操作。

12.2.1　使用 HQL 和 JPA QL 的大批量语句

HQL类似于SQL。这两者之间的区别在于HQL使用类名称而不是表名称，使用属性名称而不是列名称。它也理解继承——也就是说，无论你正在使用超类还是接口进行查询。

JPA查询语言，就像JPA和EJB 3.0定义的一样，是HQL的一个子集。因此，所有有效的JPA QL查询和语句对于HQL也是有效的。现在我们就要介绍这些语句，用于在数据库中直接执行的大批量操作，它们在JPA QL和HQL中均可用。（Hibernate采用了来自JPA的标准大批量操作。）

这些可用的语句支持在数据库中直接更新和删除对象，不需要把对象获取到内存中。也提供可以选择数据并把它当作新实体对象插入的语句。

1．直接在数据库中更新对象

前面的章节已经重申过你应该考虑对象的状态管理，而不是SQL语句如何管理。这种策略假设你正在引用的对象在内存中可用。如果执行一个直接在数据库中的行上操作的SQL语句，那么你所做的任何改变都不影响内存对象（无论它们可能处于什么状态）。换句话说，任何直接的DML语句都绕过了Hibernate持久化上下文（和所有高速缓存）。

避免这个问题的务实解决方案是一个简单的会话：首先在一个新的持久化上下文中执行任何直接的DML操作。然后，用Hibernate `Session`或者`EntityManager`加载和保存对象。这个会话保证持久化上下文不受任何之前执行的语句影响。另一种方法是，如果你知道它已经在持久化上下文的背后被修改了的话，可以选择性地使用`refresh()`操作从数据库中重载持久化对象的状态。

Hibernate和JPA提供DML操作，比简单的SQL更强大一点。让我们看看HQL和JPA QL中的第一项操作`UPDATE`：

```
Query q =
   session.createQuery("update Item i set i.isActive = :isActive");
q.setBoolean("isActive", true);
int updatedItems = q.executeUpdate();
```

这条HQL语句（或者JPA QL语句，如果通过`EntityManager`执行的话）看起来就像一条SQL语句。但是，它使用实体名称（类名称）和属性名称。它也被整合到Hibernate的参数绑定API中。返回被更新实体对象的数目——而不是被更新行的数目。另一个好处在于HQL（JPA QL）`UPDATE`语句适用于继承层次结构：

```
Query q = session.createQuery(
    "update CreditCard set stolenOn <= :now where type = 'Visa'"
);
q.setTimestamp("now", new Date() );
int updatedCreditCards = q.executeUpdate();
```

持久化引擎知道如何执行这项更新，即使必须生成几条SQL语句；它更新几张基表（因为`CreditCard`被映射到几个超类和子类表）。这个例子也没有给实体类包含别名——它是可选的。但是，如果你使用了别名，所有的属性则都必须加上别名作为前缀。还要注意，HQL（和JPA QL）`UPDATE`语句可以只引用单个实体类；例如，你无法编写单个语句来同时更新`Item`和`CreditCard`对象。`WHERE`子句中允许子查询；只有在这些子查询中才允许任何联结。

默认情况下，直接的DML操作对于被影响实体的任何版本或者时间戳值（这在Java Persistence中被标准化了）没有影响。然而，通过HQL，你可以增加直接被修改的实体实例的版本号：

```
Query q =
   session.createQuery(
        "update versioned Item i set i.isActive = :isActive"
   );
q.setBoolean("isActive", true);
int updatedItems = q.executeUpdate();
```

（如果你的版本或者时间戳属性依赖定制的org.hibernate.usertype.UserVersionType，则不允许用versioned关键字。）

引入的第二个HQL（JPA QL）大批量操作是DELETE：

```
Query q = session.createQuery(
    "delete CreditCard c where c.stolenOn is not null"
);
int updatedCreditCards = q.executeUpdate();
```

应用与UPDATE语句相同的规则：没有联结，只有单个实体类、可选的别名，WHERE子句中允许子查询。

就像SQL大批量操作一样，HQL（和JPA QL）大批量操作不影响持久化上下文，它们绕过任何高速缓存。如果你执行这其中一个例子，内存中的信用卡或者货品则不会被更新。

最后一个HQL大批量操作可以直接在数据库中创建对象。

2．在数据库中直接创建新对象

假设所有客户的Visa卡都被偷了。你编写两个大批量操作来标记被偷的日期（即发现被偷的日期），并从记录中移除其安全受到危及的信用卡的数据。因为你为一家有责任感的公司工作，必须向主管和受影响的客户报告被偷的信用卡。因此，在删除这些记录之前，提取了被偷的一切信息，并创建几百（或者几千）个StolenCreditCard对象。这是专为这一用途而编写的一个新类：

```
public class StolenCreditCard {

    private Long id;
    private String type;
    private String number;
    private String expMonth;
    private String expYear;
    private String ownerFirstname;
    private String ownerLastname;
    private String ownerLogin;
    private String ownerEmailAddress;
    private Address ownerHomeAddress;

    ... // Constructors, getter and setter methods
}
```

现在通过XML文件或者JPA注解（你自己完成这些不应该有任何问题了），把这个类映射到它自己的STOLEN_CREDIT_CARD表中。接下来，你需要一条直接在数据库中执行的语句，获取所有安全受到危及的信用卡，并创建新的StolenCreditCard对象：

```
Query q = session.createQuery(
    "insert into StolenCreditCard
           (type, number, expMonth, expYear,
            ownerFirstname, onwerLastname, ownerLogin,
            ownerEmailAddress, ownerHomeAddress)
     select
            c.type, c.number, c.expMonth, c.expYear,
            u.firstname, u.lastname, u.username,
            u.email, u.homeAddress
       from CreditCard c join c.user u
```

```
        where c.stolenOn is not null"
);

int createdObjects = q.executeUpdate();
```

这项操作完成了两件事：首先，选择CreditCard记录和对应的所有者（User）的详细信息。然后结果被直接插入到被映射到StolenCreditCard类上的表中。

注意下面几点：

- INSERT ... SELECT的目标属性（在这个例子中，是你列出的StolenCreditCard属性）必须针对一个特定的子类，而不是（抽象的）超类。由于StolenCreditCard不是继承层次结构的一部分，因此这不成问题。
- SELECT返回的类型必须与INSERT要求的类型相匹配——在这个例子中，指大量的string类型和一个组件（与用于选择和插入相同的组件类型）。
- 用于每个StolenCreditCard对象的数据库标识符，将通过标识符生成器（你使用它来映射该数据库标识符）自动生成。另一种方法是，你可以把标识符属性添加到一个被插入的属性的清单中，并提供通过选择的值。注意，标识符值的自动生成仅适用于直接在数据库内部操作的标识符生成器，如序列或者同一性字段。
- 如果生成的对象具有版本控制的类（通过version或者timestamp属性），也会生成新的版本（0或者当天的时间戳）。另一种方法是，可以选择一个版本（或者时间戳）值，并把版本（或者时间戳）属性添加到被插入的属性列表中。

最后，注意INSERT ... SELECT只可用于HQL；JPA QL没有标准化这种语句——因此，你的语句是不可移植的。

HQL和JPA QL大批量操作涵盖了许多情形，在这些情形之下你通常求助于简单的SQL。另一方面是，有时你无法在大数据操作中排除应用层。

12.2.2 利用批量处理

想象你必须操作所有的Item对象，并且必须做的改变不像设置标记这么繁琐（这个你之前已经用单条语句完成了）。也假设你无法创建SQL存储过程，不管出于什么原因（也许因为应用程序必须在不支持存储过程的数据库管理系统中运行）。你唯一的选择是在Java中编写这个过程，并把大量的数据获取到内存，通过这个过程来运行它。

你应该通过把工作批量化（batching）来执行这个过程。这意味着创建许多更小的数据集，代替不适合内存的单个数据集。

1. 利用批量更新编写过程

下列代码同时加载了100个Item对象用于处理：

```
Session session = sessionFactory.openSession();
Transaction tx = session.beginTransaction();

ScrollableResults itemCursor =
    session.createQuery("from Item").scroll();

int count=0;
```

```
while ( itemCursor.next() ) {
    Item item = (Item) itemCursor.get(0);
    modifyItem(item);
    if ( ++count % 100 == 0 ) {
        session.flush();
        session.clear();
    }
}

tx.commit();
session.close();
```

你用（简单的）HQL查询从数据库中加载所有的Item对象。但不是完全把查询的结果获取到内存，而是打开了一个在线的游标（cursor）。游标是一个指针，指向处在数据库中的结果集。可以通过ScrollableResults对象控制游标，并让它随着结果移动。get(int i)调用把单个对象获取到内存中，即游标当前所指的对象。对next()的每一次调用，都使游标朝着下一个对象前进。为了避免内存耗尽，你在下一个100个对象加载到内存中之前，flush()和clear()一次持久化上下文。

持久化上下文的清除把你对最新的100个Item对象所做的改变都写到数据库中。为了实现最好的性能，你应该把Hibernate（和JDBC）配置属性hibernate.jdbc.batch_size的大小设置为与过程批量的大小相同：100。然后在清除期间执行的所有UPDATE语句也都在JDBC级被批量化。

（注意，你应该给任何批量操作禁用二级高速缓存；否则，批量过程期间对象的每一次修改都必须为这个持久化类传播到二级高速缓存。这是一项没有必要的过载。第13章将介绍如何控制二级高速缓存。）

遗憾的是，JPA不支持基于游标的查询结果。你必须调用org.hibernate.Session和org.hibernate.Query来访问这个特性。

创建和持久化大量的对象也可以使用相同的方法。

2．批量插入许多对象

如果必须在一个工作单元中创建几百或者几千个对象，就可能遇到内存耗尽的问题。被传递到insert()或者persist()的每一个对象都被添加到这个持久化上下文高速缓存中。

一种简单的解决方案是，在一定数量的对象之后清洗和清除持久化上下文。你有效地批量插入：

```
Session session = sessionFactory.openSession();
Transaction tx = session.beginTransaction();

for ( int i=0; i<100000; i++ ) {
    Item item = new Item(...);
    session.save(item);
    if ( i % 100 == 0 ) {
        session.flush();
        session.clear();
    }
}

tx.commit();
session.close();
```

这里同时创建并持久化了100 000个对象，每次100个对象。还是要记住把hibernate.jdbc.batch_size配置属性设置为一个相等的值，并给持久化类禁用二级高速缓存。警告：如果你的实体通过identity标识符生成器被映射，Hibernate就会悄悄地禁用JDBC批量插入；许多JDBC驱动程序在这种情况下都不支持批量。

另一种完全避免持久化上下文的内存消耗的选项(通过有效地禁用它)是StatelessSession接口。

12.2.3 使用无状态的会话

持久化上下文是Hibernate和Java Persistence引擎的一项基本特性。没有持久化上下文，你将无法操作对象状态，也无法让Hibernate自动侦测到变化。许多其他的事情也将不可能。

然而，如果你喜欢通过执行语句使用数据库的话，Hibernate给你提供了另一个接口。这个面向语句的接口org.hibernate.StatelessSession，感觉和表现都像简单的JDBC，除了你从被映射的持久化类和Hibernate的数据库可移植性中受益之外。

想象你想通过这个接口执行在前面例子中编写的相同的“更新所有的货品对象”过程：

```
Session session = sessionFactory.openStatelessSession();
Transaction tx = session.beginTransaction();

ScrollableResults itemCursor =
    session.createQuery("from Item").scroll();
while ( itemCursor.next() ) {
    Item item = (Item) itemCursor.get(0);
    modifyItem(item);
    session.update(item);
}

tx.commit();
session.close();
```

在这个例子中批量不见了——你打开了StatelessSession。不再在持久化状态下使用对象，从数据库返回的一切都处于脱管状态。因此，修改Item对象之后，必须调用update()使变化变成持久化。注意，这个调用不再重附脱管的和被修改的item。它执行一个立即的SQL UPDATE；执行完命令之后，item再次处于脱管状态。

禁用持久化上下文及使用StatelessSession接口有一些其他的严重后果和概念上的限制(至少，如果你与一般的Session比较的话)：

- StatelessSession没有持久化上下文高速缓存，没有与任何其他的二级或者查询高速缓存交互。你进行的每一件事都导致立即的SQL操作。
- 对于对象的修改不会被自动侦测（没有脏检查），并且SQL操作也不会被尽可能迟地执行(没有迟写)。
- 你调用的所有对象的修改和操作都没有被级联到任何被关联的实例。你正在使用单个实体类的实例。
- 对于被映射为实体关联（一对多，多对多）的集合的任何修改都被忽略。只考虑值类型

的集合。因此，你不应该把包含集合的实体关联（而只应该把包括外键的非反向端）映射为多对一；只通过一端处理关系。编写一个查询来获得你要的数据，否则要通过迭代一个被映射的集合来获取。

- StatelessSession绕过任何被启用的org.hibernate.Interceptor，并且无法通过事件系统被拦截（本章稍后会讨论到这两项特性）。
- 你没有受保护的对象同一性范围。同一个查询生成两个不同的内存脱管实例。如果没有小心地实现持久化类的euqals()方法，则可能导致数据别名。

StetelessSession好的使用案例很少；如果通过一般的Session手工批量操作变得麻烦，你可能会更喜欢它。记住，insert()、update()和delete()操作与一般Session中相当的save()、update()和delete()操作有着天然不同的语义。（它们或许也应该有不同的名称；StetelessSession API被特别添加到Hibernate，不需要过多的计划。Hibernate开发人员团队讨论过要在Hibernate的未来版本中重新命名这个接口；你可能在正在使用的Hibernate版本中发现它们有着不同的名称。）

本章到目前为止，已经介绍了如何通过级联、大批量和批量操作，利用最有效的策略来保存和操作许多对象。现在要考虑拦截和数据过滤，以及如何以透明的方式钩入Hibernate的过程。

12.3　数据过滤和拦截

想象你不想看到数据库中的所有数据。例如，当前登录到应用程序的用户可能无权看到所有内容。通常，你把一个条件添加到查询，动态地限制结果。如果你必须处理如安全性或者临时数据（例如“只显示最近一周的数据”）这样的关注点，这就有点难度了。更难的是集合中的限制；如果你通过Category中的Item对象迭代，就会看到它们全部。

针对这个问题的一种可能的解决方案是使用数据库视图。SQL没有标准化动态的视图——可以被限制，并在运行时通过一些参数（当前登录的用户、时间周期等）移动的视图。很少有数据库提供更加灵活的视图选项，并且即使可用，也会很贵且（或）很复杂［例如，Oracle提供了一种虚拟专有数据库（Virtual Private Database，VPD）加法］。

Hibernate给动态的数据库视图提供了另一种可选的方法：运行时通过动态参数化的数据过滤器（data filter）。我们会在接下来的小节中看看数据过滤的使用案例和应用程序。

数据库应用程序中另一个常见的问题是，需要了解被保存或者被加载的数据的横切关注点。例如，想象你必须给应用程序中的每一个数据修改编写一个审计日志。Hibernate提供org.hibernate.Interceptor接口，允许你钩入Hibernate内部处理，并执行如审计日志这样的附带作用。可以利用拦截完成更多的工作，讨论完对数据过滤后，我们会给你介绍一些技巧。

Hibernate内核基于一个事件/监听器（event/listener）模型（内部最后重构的结果）。例如，如果对象必须被加载，就触发LoadEvent。Hibernate内核被实现为这些事件默认的监听器，如果你喜欢，这个系统还让你插入自己的监听器的公共接口。事件系统提供发生在Hibernate内部的任何可想象操作的完整定制，应该被当作拦截的另一种更强大的方法——我们将介绍如何编写定制的监听器，并且让你亲自处理事件。

先在一个工作单元中应用动态的数据过滤。

12.3.1 动态数据过滤

动态数据过滤的第一个使用案例与数据安全性相关。CaveatEmptor中的User有一个ranking属性。现在假设用户只可以对由同级或者更低等级的其他用户提供的货品进行出价。在业务方面，你有几组由任意等级（一个数字）定义的用户，并且用户们只能在它们的组内部进行交易。

可以用复杂的查询实现这一点。例如，假设你想要显示Category中的所有Item对象，但只是那些被同组（与登录的用户同一等级或者更低）的用户出售的货品。你得编写一个HQL或者Criteria查询获取这些货品。然而，如果用aCategory.getItems()并导航到这些对象，则所有的Item实例都将可见。

你用动态过滤解决这个问题。

1. 定义数据过滤器

动态数据过滤器用一个全局唯一的名称在映射元数据中定义。你可以在喜欢的任何XML映射文件中添加这个全局的过滤器定义，只要它是在<hibernate-mapping>元素内部：

```
<filter-def name="limitItemsByUserRank">
    <filter-param name="currentUserRank" type="int"/>
</filter-def>
```

这个过滤器被命名为limitItemByUserRank，并接受类型int的一个运行时参数。可以把相当的@org.hibernate.annotations.FilterDef注解放在你喜欢的任何类中（或者包元数据中）；它对该类的行为没有影响：

```
@org.hibernate.annotations.FilterDef(
    name="limitItemsByUserRank",
    parameters = {
        @org.hibernate.annotations.ParamDef(
            name = "currentUserRank", type = "int"
        )
    }
)
```

过滤器现在仍未激活；没有东西（可能这个名称要除外）表明它应该应用到Item对象。你必须在想要过滤的类或者集合中应用和实现过滤器。

2. 应用和实现过滤器

你想要在Item类中应用定义好的过滤器，以便如果登录用户不具备必要的等级，就看不到Item类：

```
<class name="Item" table="ITEM">
    ...
    <filter name="limitItemsByUserRank"
            condition=":currentUserRank >=
                       (select u.RANK from USER u
                        where u.USER_ID = SELLER_ID)"/>
</class>
```

<filter>元素可以对类映射设置。它把一个具名的过滤器应用到该类的实例。condition

是一个SQL表达式，直接被传递到数据库系统，因此你可以使用任何SQL操作符或者函数。如果一条记录应该通过过滤器，它的值就必须为true。在这个例子中，你使用一个子查询来获得货品卖主的等级。非限定列（如SELLER_ID）指向实体类被映射到的表。如果当前登录用户的级别没有大于或者等于由该子查询返回的等级，Item实例就被过滤掉。

下面代码与Item实体注解中的一样：

```
@Entity
@Table(name = "ITEM")
@org.hibernate.annotations.Filter(
    name = "limitItemsByUserRank",
    condition=":currentUserRank >= " +
              "(select u.RANK from USER u" +
              " where u.USER_ID = SELLER_ID)"
)
public class Item implements { ... }
```

可以应用几个过滤器，通过把它们组合在@org.hibernate.annotations.Filters注解中。一个定义好的且得到应用的过滤器，如果为特定的工作单元而被启用，就会过滤掉没有通过这个条件的任何Item实例。我们来启用它。

3. 启用过滤器

你已经定义了一个数据过滤器，并把它应用到了一个持久化类。它仍然没有过滤任何东西；它必须对特定的Session在应用程序中被启用并参数化（EntityManager不支持这个API——你必须退回到Hibernate接口来实现这项功能）：

```
Filter filter = session.enableFilter("limitItemsByUserRank");
filter.setParameter("currentUserRank", loggedInUser.getRanking());
```

你通过名称启用过滤器；这个方法返回一个Filter实例。这个对象接受运行时的参数。必须设置已经定义好的参数。其他有用的Filter方法是getFilterDefinition()（它允许遍历参数名称和类型）和validate()（如果忘了设置参数，它会抛出HibernateException）。也可以用setParameterList()设置一系列参数，如果你的SQL条件包含一个带有量词操作符（如IN操作符）的表达式，它最有用。

现在，在被过滤的Session中执行的每一个HQL或者Criteria查询都限制了返回的Item实例：

```
List<Item> filteredItems =
                session.createQuery("from Item").list();
List<Item> filteredItems =
                session.createCriteria(Item.class).list();
```

两种检索对象的方法没有被过滤：通过标识符检索和通过对Item实例的导航访问（例如通过aCategory.getItems()从Category中）检索。

通过标识符检索无法通过动态的数据过滤器限制。这在概念上来说也是错误的：如果你知道一个Item的标识符，为什么不允许看到它？解决方案是过滤标识符——也就是说，不公开首先被限制的标识符。对于多对一或者一对一关联的过滤也一样。如果多对一关联被过滤（例如，如果你调用anItem.getSeller()，就通过返回null），关联的多样性也会改变！这在概念上也是错误的，并且不是过滤器的本意。

可以通过在一个集合中应用相同的过滤器解决第二个问题：导航访问。

4. 过滤集合

目前为止，调用aCategory.getItems()返回被该Category引用的所有Item实例。这可以通过应用到集合的一个过滤器进行限制：

```
<class name="Category" table="CATEGORY">

    ...

    <set name="items" table="CATEGORY_ITEM">
        <key column="CATEGORY_ID"/>
        <many-to-many class="Item" column="ITEM_ID">

            <filter name="limitItemsByUserRank"
                    condition=":currentUserRank >=
                               (select u.RANK from USERS u where
➥ u.USER_ID = SELLER_ID)"/>
        </many-to-many>
    </set>

</class>
```

在这个示例中，你没有把过滤器应用到集合元素，而是应用到<many-to-many>。现在，子查询中的非限定SELLER_ID列引用关联的目标ITEM表，而不是关联的CATEGORY_ITEM联结表。通过注解，你可以在@ManyToMany字段或者获取方法中使用@org.hibernate.annotations.FilterJoinTable(s)，在多对多关联上应用过滤器。

如果Category和Item之间是一对多关联，你就已经创建了下列映射：

```
<class name="Category" table="CATEGORY">

    ...

    <set name="items">
        <key column="CATEGORY_ID"/>
        <one-to-many class="Item"/>

        <filter name="limitItemsByUserRank"
                condition=":currentUserRank >=
                           (select u.RANK from USERS u
➥ where u.USER_ID = SELLER_ID)"/>
    </set>

</class>
```

通过注解，你只是把@org.hibernate.annotations.Filter(s)放在正确的字段或者获取方法中，与@OneToMany或者@ManyToMany注解并排在一起。

如果现在在Session中启用过滤器，对Category的items集合的所有遍历都会被过滤。

如果有一个默认的过滤条件应用到多个实体，就用过滤器定义对它进行声明：

```
<filter-def name="limitByRegion"
            condition="REGION >= :showRegion">
    <filter-param name="showRegion" type="int"/>
</filter-def>
```

如果把这个过滤器（无论是否使用额外的条件）应用到实体或者集合，并在Session中启用

它，这个过滤器就始终把实体表中的REGION列与运行时showRegion参数进行比较。

对于动态的数据过滤，还有许多其他的优秀用例。

5. 动态数据过滤的用例

Hibernate的动态过滤器在许多情况下都很有用。唯一受限于你的想象力和使用SQL表达式的能力。典型的用例如下：

- 安全性限制（security limit）——常见的问题是对被给定的一些任意的安全性相关条件的数据访问的限制。这可以是一个用户的等级、是用户必须属于的一个特定的组，或者是用户已经被分配到的一个角色。
- 区域数据（regional data）——数据经常通过一个区域代码进行保存（例如，一个销售团队的所有业务合同）。每个销售人员都只在覆盖他们区域的一个数据集上工作。
- 临时数据（temporal data）——许多企业应用程序需要在数据上应用基于时间的视图（例如，看看上周的一个数据集）。Hibernate的数据过滤器可以提供帮助你实现这种功能的基本的临时限制。

另一个有用的概念是Hibernate内部的拦截，实现正交的关注点。

12.3.2　拦截 Hibernate 事件

假设你想要给所有的对象修改编写一个审计日志。这个审计日志保存在包含有关对其他数据所做改变的数据库表中——特别是，有关导致改变的事件。例如，你可以给拍卖Item记录有关创建和更新事件的信息。通常记录的信息有用户、事件的日期和时间、发生了什么事件类型，以及被改变的货品。

审计日志经常利用数据库触发器处理。另一方面，应用程序承担责任有时更好，尤其当不同的数据库之间需要可移植性的时候。

实现审计日志需要几个元素。第一，必须给你想对其启用审计日志的持久化类做上标记。第二，定义什么信息应该被记入日志，例如用户、日期、时间和修改的类型。第三，用自动创建审计轨迹的org.hibernate.Interceptor把所有的东西连在一起。

1. 创建标记接口

首先，创建标记接口Auditable。用这个接口给所有应该被自动审计的持久化类做上标记：

```
package auction.model;

public interface Auditable {
    public Long getId();
}
```

这个接口要求持久化实体类用一个获取方法公开它的标识符；你需要这个属性来把审计轨迹记入日志。然后给特定的持久化类启用审计日志就是小事了。把它添加到类声明中——例如对Item：

```
public class Item implements Auditable { ... }
```

当然，如果Item类没有公开一个公有的getId()方法，就需要添加它。

2. 创建和映射日志记录

现在创建新的持久化类AuditLogRecord。这个类表示你想要在审计数据库表中记入日志的

信息：

```
public class AuditLogRecord {

    public String message;
    public Long entityId;
    public Class entityClass;
    public Long userId;
    public Date created;

    AuditLogRecord() {}

    public AuditLogRecord(String message,
                          Long entityId,
                          Class entityClass,
                          Long userId) {
        this.message = message;
        this.entityId = entityId;
        this.entityClass = entityClass;
        this.userId = userId;
        this.created = new Date();
    }
}
```

不应该把这个类当成领域模型的一部分！因此，你公开所有属性作为公有的；你将不可能需要重构应用程序的这个部分。AuditLogRecord是持久层的一部分，并可能与其他持久化相关的类共用相同的包，如HibernateUtil或者定制的UserType扩展。

接下来，把这个类映射到AUDIT_LOG数据库表：

```
<hibernate-mapping default-access="field">

<class name="persistence.audit.AuditLogRecord"
       table="AUDIT_LOG" mutable="false">

    <id type="long" column="AUDIT_LOG_ID">
        <generator class="native"/>
    </id>

    <property   name="message"
                type="string"
                column="MESSAGE"
                length="255"
                not-null="true"/>

    <property   name="entityId"
                type="long"
                column="ENTITY_ID"
                not-null="true"/>

    <property   name="entityClass"
                type="class"
                column="ENTITY_CLASS"
                not-null="true"/>

    <property   name="userId"
                type="long"
                column="USER_ID"
```

```
                not-null="true"/>
    <property   name="created"
                column="CREATED"
                type="java.util.Date"
                update="false"
                not-null="true"/>
</class>
</hibernate-mapping>
```

你映射对field策略的默认访问（这个类中没有获取方法），并且，由于AuditLogRecord对象从来不更新，映射类为mutable="false"。注意你没有声明一个标识符属性名称（类也没有这样的属性）；因此Hibernate内部管理AuditLogRecord的代理键。你不准备以脱管方式使用AuditLogRecord，因此它不必包含标识符属性。然而，如果用注解映射了这个类为Java Persistence实体，就需要标识符属性。相信你在自己的代码中创建这个实体映射已经没有任何问题了。

审计日志是对引起可记入日志事件的业务逻辑的某种正交关注点。用于审计日志的逻辑可能与业务逻辑混淆，但是在许多应用程序中，审计日志最好在一段重要的代码中进行处理，对业务逻辑透明（尤其当你依赖级联选项的时候）。每当Item被修改时创建新的AuditLogRecord并保存，这必定不是你愿意手工完成的事情。Hibernate提供了Interceptor扩展接口。

3. 编写拦截器

当调用save()时，应该自动调用logEvent()方法。利用Hibernate完成这件事的最好方法是实现Interceptor接口。代码清单12-1展现了用于审计日志的拦截器。

代码清单12-1　实现用于审计日志的拦截器

```
public class AuditLogInterceptor extends EmptyInterceptor {

    private Session session;
    private Long userId;

    private Set inserts = new HashSet();
    private Set updates = new HashSet();

    public void setSession(Session session) {
        this.session=session;
    }

    public void setUserId(Long userId) {
        this.userId=userId;
    }

    public boolean onSave(Object entity,
                          Serializable id,
                          Object[] state,
                          String[] propertyNames,
                          Type[] types)
            throws CallbackException {

        if (entity instanceof Auditable)
            inserts.add(entity);
```

```
        return false;
    }

    public boolean onFlushDirty(Object entity,
                                Serializable id,
                                Object[] currentState,
                                Object[] previousState,
                                String[] propertyNames,
                                Type[] types)
            throws CallbackException {
        if (entity instanceof Auditable)
            updates.add(entity);

        return false;
    }

    public void postFlush(Iterator iterator)
                    throws CallbackException {
        try {
            for (Iterator it = inserts.iterator(); it.hasNext();) {
                Auditable entity = (Auditable) it.next();
                AuditLog.logEvent("create",
                                  entity,
                                  userId,
                                  session.connection());
            }
            for (Iterator it = updates.iterator(); it.hasNext();) {
                Auditable entity = (Auditable) it.next();
                AuditLog.logEvent("update",
                                  entity,
                                  userId,
                                  session.connection());
            }
        } finally {
            inserts.clear();
            updates.clear();
        }
    }
}
```

Hibernate `Interceptor` API的方法远远不止于这个例子中所展现的。由于你正在扩展`EntityManager`而不是直接实现这个接口，因此可以依赖你没有覆盖的所有方法的默认语义。拦截器有两个值得关注的方面。

这个拦截器需要`session`和`userId`属性进行工作；使用这个拦截器的客户端必须设置这两个属性。另一个值得关注的方面是`onSave()`和`onFlushDirty()`中的审计日志子程序：把新的、更新过的实体添加到`inserts`和`updates`集合。每当Hibernate保存实体时，就调用`onSave()`拦截器方法；每当Hibernate侦测到脏对象时，就调用`onFlushDirty()`方法。

审计轨迹的实际日志在`postFlush()`方法中完成，Hibernate在执行了把持久化上下文实现为与数据库同步的SQL之后调用它。使用静态的调用`AuditLog.logEvent()`（我们接下来要讨论的

这个类和方法）把事件记入日志。注意你无法在onSave()中把事件记入日志，因为瞬时实体的标识符值不可能在这个时候被知道。Hibernate保证在清除期间设置实体标识符，因此postFlush()是把这个信息记入日志的最好地方。

还要注意如何使用session：你把给定的Session的JDBC连接传递到对AuditLog.logEvent()的静态调用。这么做自有它的道理，我们将会进行深入的讨论。

先把它全部连在一起，看看如何启用这个新的拦截器。

4．启用拦截器

第一次打开会话时，你需要把Interceptor分配到Hibernate Session：

```
AuditLogInterceptor interceptor = new AuditLogInterceptor();
Session session = getSessionFactory().openSession(interceptor);
Transaction tx = session.beginTransaction();

interceptor.setSession(session);
interceptor.setUserId( currentUser.getId() );

session.save(newItem); // Triggers onSave() of the Interceptor

tx.commit();
session.close();
```

对于用其打开的Session来说，拦截器是活动的。

如果使用sessionFactory.getCurrentSession()，就不要控制Session的打开；它由CurrentSessionContext的其中一个Hibernate内建的实现透明地进行处理。可以编写自己的（或者扩展现有的）CurrentSessionContext实现，为打开当前的Session并给它分配一个拦截器提供自己的子程序。

另一种启用拦截器的方法是，在创建SessionFactory之前，通过setInterceptor()在Configuration上全局地设置它。然而，在Configuration中设置的并且所有Session都可以启用的任何拦截器都必须被线程安全地实现！单个Interceptor实例被并发运行的Session共享。AuditLogInterceptor实现不是线程安全的：它使用了成员变量（inserts和updates队列）。

也可以通过persistence.xml中的下列配置选项，设置一个共享的线程安全的拦截器，该拦截器有一个无参构造器，用于JPA中所有的EntityManager实例：

```
<persistence-unit name="...">
  <properties>
      <property name="hibernate.ejb.interceptor"
                value="my.ThreadSafeInterceptorImpl"/>
      ...
  </properties>
</persistence-unit>
```

我们回到拦截器中那段值得关注的Session处理代码，找出为什么要把当前Session的connection()传递到AuditLog.logEvent()。

5．使用临时的会话

AuditLogInterceptor内部为什么需要Session，这应该很清楚了。拦截器必须创建和持久化AuditLogRecord对象，因此onSave()方法的第一个尝试可能就是下列子程序：

```
if (entity instanceof Auditable) {

    AuditLogRecord logRecord = new AuditLogRecord(...);
    // set the log information

    session.save(logRecord);
}
```

这似乎很简单：利用当前运行的Session，创建新的AuditLogRecord实例并保存它。但这样并不起作用。

从Interceptor回调中调用原始的Hibernate Session是非法的。Session在拦截器调用期间处于易碎状态。在其他对象的保存期间，无法save()新对象！避免这个问题的一个好技巧是，只为保存单个AuditLogRecord对象打开新的Session。从原始的Session中复用JDBC连接。

这个临时的Session处理被封装在AuditLog类中，如代码清单12-2所示。

代码清单12-2 AuditLog辅助类使用了临时Session

```
public class AuditLog {

    public static void logEvent(
        String message,
        Auditable entity,
        Long userId,
        Connection connection) {

        Session tempSession =
                    getSessionFactory().openSession(connection);
        try {
            AuditLogRecord record =
                new AuditLogRecord(message,
                                   entity.getId(),
                                   entity.getClass(),
                                   userId );

            tempSession.save(record);
            tempSession.flush();

        } finally {
            tempSession.close();
        }
    }
}
```

logEvent()方法在同一个JDBC连接中使用了新的Session，但它从不启动或者提交任何数据库事务。它所做的就是在清除期间执行单个的SQL语句。

在同一个JDBC连接和事务中给某些操作使用临时Session的这一技巧，有时候在其他情况下也有帮助。你要记住的就是，Session只是持久化对象（持久化上下文）的高速缓存，以及使这个高速缓存与数据库同步的一系列SQL操作。

我们鼓励你去体验和尝试不同的拦截器设计模式。例如，可以重新设计审计机制，把任何实体都记入日志，而不仅仅是Auditable。Hibernate网站也有使用内嵌拦截器、甚至把一个实体的

完整历史（包括被更新的属性和集合信息）记入日志的示例。

org.hibernate.Interceptor接口也有更多可以用来钩入Hibernate过程的方法。我们认为拦截通常足以实现任何正交的关注点。

话虽这么说，Hibernate还是允许你通过它所基于的可扩展事件系统，更深地钩入到它的内核。

12.3.3　内核事件系统

Hibernate 3.x与Hibernate 2.x相比，是内核持久化引擎实现的一项重要的重新设计。新内核引擎基于一个事件/监听器的模型。例如，如果Hibernate需要保存对象，就会触发一个事件。任何监听器听到这种事件都可以捕捉它，并处理对象的保存。所有Hibernate内核功能都因此被实现为一组默认的监听器，它们可以处理所有的Hibernate事件。

这已经被设计为开放的系统：可以给Hibernate事件编写和启用自己的监听器。可以替换现有的默认监听器，或者扩展它们并执行一个附带作用或者额外的过程。替换事件监听器很少见；这么做意味着你自己的监听器实现可以负责一些Hibernate内核功能。

本质上而言，Session接口的所有方法都与一个事件关联。load()方法触发LoadEvent，并且这个事件默认通过DefaultLoadEventListener处理。

定制的监听器应该给它想要处理的事件实现适当的接口，并且（或者）扩展Hibernate提供的其中一个便利的基类，或者任何默认的事件监听器。下面是定制加载事件监听器的一个例子：

```
public class SecurityLoadListener extends DefaultLoadEventListener {

    public void onLoad(LoadEvent event,
                       LoadEventListener.LoadType loadType)
            throws HibernateException {

        if ( !MySecurity.isAuthorized(
                event.getEntityClassName(), event.getEntityId()
             )
           ) {
            throw MySecurityException("Unauthorized access");
        }

        super.onLoad(event, loadType);

    }
}
```

这个监听器调用静态的方法isAuthorized()，通过必须加载的实例的实体名称，以及该实例的数据库标识符。如果对该实例的访问被拒绝，就会抛出一个定制的运行时异常。如果没有异常抛出，过程就会被传递到超类中默认的实现。

监听器实际上应该被认为是单例（singleton），意味着它们在请求之间被共享，因而不应该把任何事务相关的状态保存为实例变量。对于原生Hibernate中所有事件和监听器接口的一个清单，请见org.hibernate.event包的API Javadoc。一个监听器实现也可以实现多个事件—监听器接口。

定制监听器可以通过Hibernate Configuration对象被编程式地注册，或者在Hibernate配置

XML中指定（不支持通过属性文件的声明性配置）。你还需要一个配置入口，告诉Hibernate除了默认的监听器之外还要使用这个监听器：

```
<session-factory>
  ...
  <event type="load">
    <listener class="auction.persistence.MyLoadListener"/>
  </event>
</session-factory>
```

监听器按照它们在你配置文件中排列的相同顺序进行注册。可以创建一个监听器的堆栈。在这个例子中，由于你正在扩展内建的DefaultLoadEventListener，因此只有一个监听器。如果没有扩展DefaultLoadEventListener，则必须把内建的DefaultLoadEventListener指定为堆栈中的第一个监听器，否则要在Hibernate中禁用加载！

另一种方法是，编程式地注册监听器堆栈：

```
Configuration cfg = new Configuration();

LoadEventListener[] listenerStack =
    { new MyLoadListener(), ...};

cfg.getEventListeners().setLoadEventListeners(listenerStack);
```

声明注册的监听器无法共享实例。如果同一个类名被用在多个<listener/>元素中，则每个引用都会产生该类的一个单独实例。如果需要在监听器类型之间共享监听器实例的能力，就必须使用编程式的注册方法。

Hibernate EntityManager也支持监听器的定制。可以在persistence.xml配置中配置共享的事件监听器，就像下面这样：

```
<persistence-unit name="...">
  <properties>
      <property name="hibernate.ejb.event.load"
                value="auction.persistence.MyLoadListener, ..."/>
      ...
  </properties>
</persistence-unit>
```

配置选项的属性名称随着你想要监听的每个事件类型（在前一个例子中是load）的改变而改变。

如果替换内建的监听器，就像MyLoadListener所做的那样，就需要扩展正确的默认监听器。在编写本书之时，Hibernate EntityManager没有绑定它自己的LoadEventListener，因此扩展org.hibernate.event.DefaultLoadEventListener的监听器仍然运行得很好。可以在参考文档以及org.hibernate.ejb.event包的Javadoc中找到完整的、最新的Hibernate EntityManager默认监听器的清单。如果想要保留Hibernate EntityManager引擎的基本行为，就扩展这些监听器中的任何一个。

几乎不必用自己的功能去扩展Hibernate内核事件系统。大多数时候，org.hibernate.Interceptor就足够灵活了。它帮助拥有更多选项，帮助以模块的方式替换任何Hibernate内核引擎。

EJB 3.0标准包括几个拦截选项，用于会话bean和实体。可以把会话bean方法调用包装在任何定制拦截器中，拦截对一个实体实例的任何修改，或者让Java Persistence服务在特定生命周期事件的bean中调用方法。

12.3.4　实体监听器和回调

EJB 3.0实体监听器是拦截实体回调事件（例如实体实例的加载和保存）的类。这类似于原生的Hibernate拦截器。可以编写定制监听器，并通过注解或者XML部署描述符中的一个绑定，把它们附加到实体。

看看下面这个简易的监听器：

```
import javax.persistence.*;

public class MailNotifyListener {

    @PostPersist
    @PostLoad
    public void notifyAdmin(Object entity) {
        mail.send("Somebody saved or loaded: " + entity);
    }

}
```

实体监听器不实现任何特定的接口；它需要一个无参构造器（在前一个例子中，是默认的构造器）。你把回调注解应用到特定事件中需要被通知的任何方法；可以在单个方法中合并几个回调。不允许在几个方法中重复相同的回调。

监听器类通过注解绑定到一个特定的实体类：

```
import javax.persistence.*;

@Entity
@EntityListeners(MailNotifyListener.class)
public class Item {
    ...

    @PreRemove
    private void cleanup() {
        ...
    }
}
```

如果需要绑定几个监听器，@EntityListeners注解可以带上一大批类。也可以把回调注解放在实体类自身中，但还是不能在单个类的方法中重复回调。然而，可以在几个监听器类中或者在监听器和实体类中实现相同的回调。

也可以为整个层次结构把监听器应用到超类，并在orm.xml配置文件中定义默认的监听器。最后，可以通过@ExcludeSuperclassListeners和@ExcludeDefaultListeners注解，给特定的实体排除超类监听器或者默认的监听器。

所有的回调方法都可以有任意的可见性，必须返回void，并且不允许抛出任何checked exception。如果抛出unchecked exception，并且有一个JTA事务正在处理，这个事务就会被

回滚。

可用的JPA回调的清单如表12-2所示。

表12-2 JPA事件回调和注解

回调注解	描 述
@PostLoad	在实体实例通过find()或者getReference()加载之后，或者在执行Java Persistence查询时触发。也在调用refresh()方法之后被调用
@PrePersist, @PostPersist	当persist()在实体中调用时，且在数据库插入之后，立即发生
@PreUpdate, @PostUpdate	在持久化上下文与数据库同步之前和之后执行——也就是说，在清除之前和之后。只在实体状态需要同步时才触发（例如，由于被认为是脏的）
@PreRemove, @PostRemove	当调用remove()时，或者当通过级联移除实体实例时，并且是在数据库删除之后触发

不像Hibernate拦截器，实体监听器是无状态的类。因此无法用实体监听器重写前一个Hibernate审计日志示例，因为你需要在本地队列中持有被修改对象的状态。另一个问题是，不允许实体监听器类使用EntityManager。确定的JPA实现，如Hibernate，让你用一个临时的另一个持久化上下文再次应用这个技巧，但你应该看看EJB 3.0拦截器中的会话bean，或许在应用程序堆栈中更高的层中编写这个审计日志的代码。

12.4 小结

本章介绍了如何有效地利用大型的复杂数据集。我们首先探讨了Hibernate的级联选项，以及如何通过Java Persistence和注解启用传播性持久化。然后，涵盖HQL和JPA QL中的大批量操作，以及如何编写在数据的子集上工作的批量过程，避免内存耗尽。

最后一节介绍了如何启用Hibernate数据过滤器，以及如何在应用程序的级别上创建动态的数据视图。最后，我们介绍了Hibernate Interceptor扩展点、Hibernate内核事件系统和标准Java Persistence实体回调机制。

表12-3展现了可以用来比较原生Hibernate特性和Java Persistence的一个概括。

第13章将转换视角，讨论如何利用最好的抓取和高速缓存策略，从数据库中获取对象。

表12-3 第12章中Hibernate和JPA对照表

Hibernate Core	Java Persistence和EJB 3.0
Hibernate通过级联选项，对所有操作支持传播性持久化	包含级联选项的传播性持久化模型相当于Hibernate。对于特殊案例请使用Hibernate注解
Hibernate在多态的HQL中支持大批量UPDATE、DELETE和INSERT...SELECT操作，它们直接在数据库中执行	JPA QL支持直接的大批量UPDATE和DELETE
Hibernate对批量更新支持查询结果游标	Java Persistence没有标准化带有游标的查询，需要退回到Hibernate API
强大的数据过滤可用于动态数据视图的创建	对于数据过滤器的映射请使用Hibernate扩展注解
扩展点可用于拦截和事件监听器	提供标准的实体生命周期回调处理器

第 13 章

优化抓取和高速缓存

本章内容

- 全局抓取策略
- 高速缓存理论
- 高速缓存实践

本章将介绍如何从数据库中获取对象，以及当你在应用程序中逐个导航对象时如何优化对象网络的加载。

之后讲述高速缓存，介绍如何在本地应用程序和分布式应用程序中加快数据的获取。

13.1 定义全局抓取计划

从数据库中获取持久化对象是使用Hibernate最值得关注的部分之一。

13.1.1 对象获取选项

Hibernate提供了下列方法从数据库中获取对象：

- 导航对象图，从一个已经加载的对象开始，通过如`aUser.getAddress().getCity()`等这样的属性访问方法访问被关联的对象。如果持久化上下文仍然是打开的，当调用访问方法时，Hibernate就会自动加载（和预加载）对象图中的节点。
- 通过标识符获取，当一个对象的唯一标识符值已知时，这是最方便的方法。
- HQL（Hibernate Query Language，Hibernate查询语言），它是一种完全面向对象的查询语言。Java QL（Java Persistence query language，JPA持久化查询语言）是HQL的一个标准子集。
- Hibernate `Criteria`接口，它提供了一种类型安全和面向对象的方式来执行查询，而不需要进行字符串操作。这种机制包括基于示例对象的查询。
- 原生的SQL查询，包括存储过程的调用（在这里Hibernate仍然把JDBC的结果集映射到持久化对象图中）。

在Hibernate或者JPA应用程序中，可以结合使用这些方法。

本章并不详细讨论每一种获取方法。我们更感兴趣的是所谓的默认抓取计划（fetch plan）和抓取策略（fetching strategy）。默认的抓取计划和抓取策略是应用到特定的实体关联或者集合的计划和策略。换句话说，它定义了当加载自己的实体对象，以及访问一个被关联的对象或者集合时，被关联的对象或者集合是否以及如何被加载。每一种获取方法都可能使用不同的计划和策略——也就是说，计划定义了持久化对象网络的哪一部分应该被获取和怎样获取。你的目标是为应用程序中的每个用例找到最好的获取方法和抓取策略；同时也要最小化SQL查询的次数，以获得最好的性能。

在探讨抓取计划选项和抓取策略之前，我们将全面概括获取方法。（我们也会偶尔提到Hibernate高速缓存系统，本章稍后会全面探讨它。）

前一章已经介绍了如何通过标识符获取对象，这里不再赘述。我们直接讨论更灵活的查询选项，HQL（等同于JPA QL）和Criteria。这两者都允许创建任意查询。

1. HQL和JPA QL

HQL是常见的数据库查询语言SQL的一种面向对象的方言。HQL与ODMG OQL极其相似，但与OQL不同的是，它适用于SQL数据库，并且更易于学习（由于它很类似于SQL），并且实现更完整（没有任何OQL的实现是完整的）。

EJB 3.0标准定义了Java Persistence查询语言。这个新的JPA QL和HQL已经融合，并且JPA QL成了HQL的一个子集。有效的JPA QL查询也始终是有效的HQL查询。HQL有更多的选项，应该被认为是标准化子集的供应商扩展。

HQL常用于对象获取，而不是更新、插入或者删除数据。对象状态同步是持久化管理器的工作，而不是开发人员的工作。但是正如前一章中讲过的，如果用例（大量数据操作）需要，HQL和JPA QL都支持直接的大批量操作，用于更新、删除和插入。

大多数情况下，只需要获取特定类的对象，并通过该类的属性进行限制。例如，下列查询通过名字获取了用户。

```
Query q = session.createQuery(
    "from User as u where u.firstname = :fname"
);
q.setString("fname", "John");
List result = q.list();
```

在准备了查询q之后，把一个值绑定到具名参数fname。结果返回User对象的List。

HQL是强大的，即使可能一直不使用更高级的特性，但对于更难的问题也会需要它们。例如，HQL支持：

- 对按引用相关或者处于集合中的被关联对象的属性，应用限制的能力（用查询语言导航对象图）。
- 只获取一个或者多个实体的属性的能力，没有加载实体本身到持久化上下文的系统开销。有时这被称作报告查询，更准确的称呼是投影。
- 对查询结果进行排序的能力。
- 对查询结果进行分页的能力。

- 与group by、having和聚集函数（如sum、min和max/min）联合使用。
- 当每一行获取多个对象时使用外部联结。
- 调用标准的和用户定义的SQL函数的能力。
- 子查询（嵌套查询）。

第14章和第15章将讨论所有这些特性，以及可选的原生SQL查询机制。

2. 按条件查询

Hibernate的按条件查询（Query By Criteria，QBC）API允许查询在运行时通过Criteria对象的操作进行创建。这样让你可以动态地指定约束，而无需直接的字符串操作，但是会失去少量的HQL的灵活性或者功能性。另一方面，用条件表达的查询通常比用HQL表达的查询更不具备可读性。

用Criteria对象根据名字获取用户很容易，如：

```
Criteria criteria = session.createCriteria(User.class);
criteria.add( Restrictions.like("firstname", "John") );
List result = criteria.list();
```

一个Criteria就是一棵Criterion实例的树。Restriction类提供了返回Criterion实例的静态工厂方法。一旦建立了想要的Criteria树，就可以依靠数据库来执行它了。

许多开发人员更喜欢按条件查询，原因是他们认为它是更面向对象的方法。他们也喜欢其查询语法是在编译时被解析和验证的，相比之下，HQL表达式只有在运行时（或者启动时，如果使用了具体命名的查询）才可被解析。

Hibernate Criteria API好的一面是，Criterion框架允许用户进行扩展，这在像HQL的查询语言中很难做到。

注意，Criteria API是Hibernate固有的，它不是Java 持久化标准的一部分。在实践中，Criteria将成为JPA应用程序中最常用的Hibernate扩展。我们期待JPA或者EJB标准的未来版本中能包含类似的编程式的查询接口。

3. 按示例查询

作为Criteria工具的一部分，Hibernate支持按示例查询（Query By Example，QBE）。按示例查询背后的思想是，应用程序通过某些属性值设置，提供被查询类的实例（到非默认的值）。查询结果返回带有匹配属性值的所有持久化实例。QBE不是特别强大的方法。然而，它对于某些应用程序很方便，尤其当它结合Criteria使用的时候：

```
Criteria criteria = session.createCriteria(User.class);

User exampleUser = new User();
exampleUser.setFirstname("John");
criteria.add( Example.create(exampleUser) );

criteria.add( Restrictions.isNotNull("homeAddress.city") );

List result = criteria.list();
```

这个示例先创建查询User对象的新的Criteria。然后只使用firstname属性设置来添加Example对象User实例。最后，在执行这个查询之前添加Restriction条件。

QBY的一个典型用例是搜索屏，允许用户指定一系列不同的属性值要与返回的结果集相匹配。这种功能可能难以用查询语言表达清楚；需要字符串操作来指定约束的动态集合。

Criteria API和按示例查询机制会在第15章中进行更深入的讨论。

你现在知道Hibernate中基本的获取选项了。本节剩下的篇幅，关注对象抓取计划和抓取策略。

让我们从“什么应该被加载到内存中去”这个定义开始。

13.1.2 延迟的默认抓取计划

Hibernate给所有的实体和集合默认一个延迟的抓取策略。这意味着Hibernate在默认情况下只加载你正在查询的对象。我们用几个示例对此进行探讨。

如果查询Item对象（假设你通过它的标识符加载），则只有这个Item被加载到内存中去：

```
Item item = (Item) session.load(Item.class, new Long(123));
```

通过标识符的这个获取形成了单个（或者可能几个，如果继承或者二级表被映射的话）获取Item实例的SQL语句。在持久化上下文中，现在你在内存中便有了这个处于持久化状态的item对象，如图13-1所示。

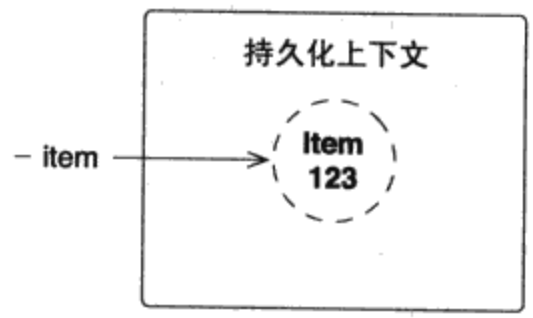


图13-1 一个未对Item实例初始化的占位符

我们骗了你。load()操作之后，内存中并没有持久化的item对象。甚至加载Item的SQL也没有被执行。Hibernate创建了一个看起来跟真的一样的代理（proxy）。

13.1.3 理解代理

代理是在运行时生成的占位符。每当Hibernate返回实体类的实例时，它就检查是否可以返回一个代理来代替，并避免数据库命中。代理是当真实对象第一次被访问时触发其加载的占位符：

```
Item item = (Item) session.load(Item.class, new Long(123));
item.getId();
item.getDescription(); // Initialize the proxy
```

这个示例中的第三行触发了把Item获取到内存的SQL的执行。只要你只访问数据库标识符属性，就没有必要初始化代理。（注意，如果用直接的字段访问映射标识符属性，就不是这样了；Hibernate当时甚至不知道getId()方法的存在。如果调用它，代理就必须被初始化。）

如果需要Item只是用来创建一个引用，代理就很有用了，例如：

```
Item item = (Item) session.load(Item.class, new Long(123));
User user = (User) session.load(User.class, new Long(1234));

Bid newBid = new Bid("99.99");
newBid.setItem(item);
newBid.setBidder(user);

session.save(newBid);
```

先加载两个对象：Item和User。Hibernate并没有命中数据库来完成这项工作，而是返回两个代理。这就是你所要的，因为你只需要Item和User来创建一个新的Bid。save(newBid)调用执行INSERT语句，通过Item和User的外键值保存BID表中的行——这是代理能够且必须提供的一

切。前一个代码片段没有执行任何SELECT语句！

如果调用get()而不是load()，就触发了一次数据库命中，且没有返回任何代理。get()操作始终命中数据库（如果实例还没有处在持久化上下文中，并且如果没有任何透明的二级高速缓存是活动的），如果无法找到对象就返回null。

JPA提供程序可以利用代理实现延迟加载。EntityManager API中相当于load()和get()这些操作的方法名称是find()和getReference()：

```
Item item = em.find(Item.class, new Long(123));

Item itemRef = em.getReference(Item.class, new Long(1234));
```

第一个调用find()，必须命中数据库以初始化Item实例。不允许代理——相当于Hibernate的get()操作。第二个调用getReference()，可能返回一个代理，但是不一定——它在Hibernate中变成load()。

由于Hibernate代理是实体类运行时生成的子类的实例，你无法通过一般的操作符得到一个对象的类。这就是辅助方法HibernateProxyHelper.getClassWithoutInitializingProxy(o)派上用场的地方。

假设你让Item实例进到内存中，要么通过显式获取，要么通过调用它的其中一个属性并强制代理初始化。持久化上下文现在包含一个完全加载的对象，如图13-2所示。

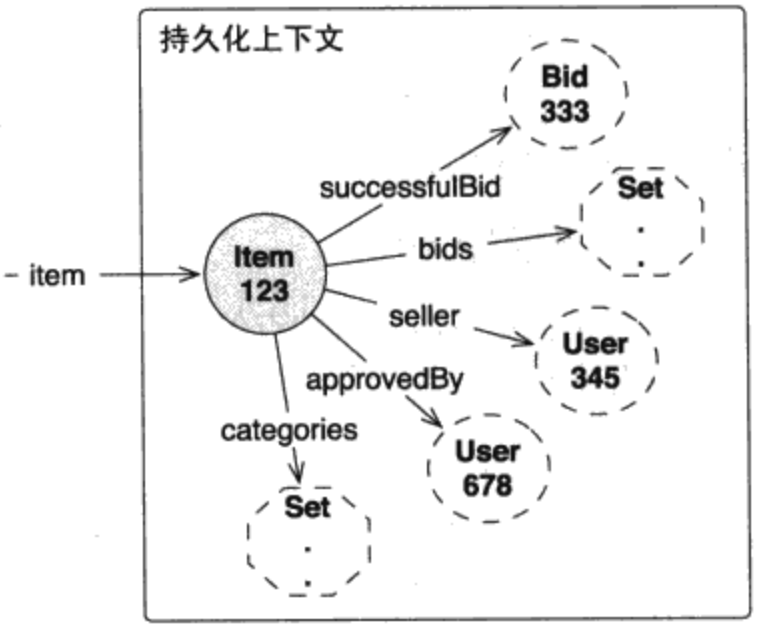


图13-2　代理和集合包装器表示被加载图的范围

在这幅图中你可以再次见到代理。这一次，这些是已经为所有*-to-one关联生成的代理。被关联的实体对象没有马上被加载；代理只提供标识符值。从一个不同的视角来看：标识符值是item行中所有的外键列。集合也没有被立即加载，但我们用术语集合包装器（collection wrapper）来描述这种占位符。Hibernate内部有一组智能的集合，可以按需要初始化它们自己。Hibernate用这些替换你的集合；这就是为什么应该只在领域模型中使用集合接口的原因。默认情况下，Hibernate给所有的关联和集合都创建占位符，并且仅仅立即获取值类型的属性和组件。（不幸的是，这并不是Java Persistence标准化的默认抓取计划；稍后会回来讨论其中的差别。）

常见问题　一对一关联的延迟加载有用吗？对一对一关联延迟加载，有时候把刚刚接触Hibernate的用户们搞糊涂了。如果你基于共享的主键来考虑一对一关联（见7.1.1节），关联只有当它为constrained="true"时才可以被代理。例如，Address始终有一个对User的引用。如果这个关联可为空并且可选，Hibernate首先必须命中数据库，决定应该应用代理还是null——延迟加载的目的是根本不命中数据库。可以通过BCI和拦截启用延迟加载，我们稍后会讨论到。

如果你调用任何不是标识符获取方法的方法，代理就被初始化，如果你通过集合的元素启动迭代，或者如果调用任何集合管理的操作，如size()和contains()，集合就被初始化。Hibernate提供一项对大集合最有用的额外设置：它们可以被映射为extra延迟。例如，考虑Item的bids集合：

```
<class name="Item" table="ITEM">
    ...
    <set name="bids"
         lazy="extra"
         inverse="true">
        <key column="ITEM_ID"/>
        <one-to-many class="Bid"/>
    </set>
</class>
```

集合包装器现在比以前更智能了。如果调用size()、contains()或者isEmpty()，集合将不再被初始化——查询数据库来获取必要的信息。如果它是一个Map或者List，containsKey()和get()操作也直接查询数据库。Hibernate扩展注解启用了同样的优化：

```
@OneToMany
@org.hibernate.annotations.LazyCollection(
    org.hibernate.annotations.LazyCollectionOption.EXTRA
)
private Set<Bid> bids = new HashSet<Bid>();
```

我们来定义不完全延迟的抓取计划。首先，可以给实体类禁用代理生成。

13.1.4 禁用代理生成

代理是个好东西：它们允许你只加载真正需要的数据。它们甚至让你创建对象之间的关联，而不用没有必要地命中数据库。有时候你会需要一种不同的计划——例如，你想要表达User对象应该永远被加载到内存中，并且不应该返回任何占位符来代替。

可以给一个特定的实体类禁用代理生成，通过XML映射元数据中的lazy="false"属性：

```
<class name="User"
       table="USERS"
       lazy="false">
       ...
</class>
```

JPA标准不要求通过代理实现；“代理”一词甚至没有出现在规范中。Hibernate是默认情况下依赖代理的JPA提供程序，因此禁用Hibernate代理的切换可以用一个供应商扩展：

```
@Entity
@Table(name = "USERS")
@org.hibernate.annotations.Proxy(lazy = false)
public class User { ... }
```

给实体禁用代理生成会有严重的后果。所有这些操作都需要数据库命中：

```
User user = (User) session.load(User.class, new Long(123));
User user = em.getReference(User.class, new Long(123));
```

User对象的load()无法返回代理。JPA操作getReference()可以不再返回代理引用。这可

能会是你想要实现的东西。然而，禁用代理对于所有引用该实体的关联来说，也有影响。例如，Item实体有一个对User的seller关联。考虑下列获取Item的操作：

```
Item item = (Item) session.get(Item.class, new Long(123));
Item item = em.find(Item.class, new Long(123));
```

除了获取Item实例之外，get()操作现在也加载Item的被链接的seller；没有给这个关联返回任何User代理。对于JPA也一样：已经用find()加载的Item没有引用seller代理。正在出售Item的User必须被立即加载。（我们稍后会解答这如何抓取的问题。）

在全局的级别上禁用代理生成经常过于粗粒度。通常，你只想禁用一个特定实体关联或者集合的延迟加载行为，来定义一个细粒度的抓取计划。你想要的正好相反：特定关联或者集合的即时加载。

13.1.5　关联和集合的即时加载

你已经知道，Hibernate默认情况下是延迟的。如果加载一个实体对象，所有被关联的实体和集合就都没有被初始化。一般来说，开发人员经常想要相反的结果：指定一个特定的实体关联或者集合应该始终被加载。你想要保证这个数据在内存中不用额外的数据库命中就可用。最重要的是，比如，即使Item实例处于脱管状态，你希望仍然可以访问Item的seller。必须定义这项抓取计划，这是你想要始终加载到内存中的一部分对象网络。

假设你始终需要Item的seller。在Hibernate XML映射元数据中，映射从Item到User的关联为lazy="false"：

```
<class name="Item" table="ITEM">
    ...
    <many-to-one name="seller"
                 class="User"
                 column="SELLER_ID"
                 update="false"
                 not-null="true"
                 lazy="false"/>
    ...
</class>
```

同样的“始终加载”保证可以被应用到集合——例如，Item的所有bids：

```
<class name="Item" table="ITEM">
    ...
    <many-to-one name="seller" lazy="false" .../>
    <set name="bids"
         lazy="false"
         inverse="true">
        <key column="ITEM_ID"/>
        <one-to-many class="Bid"/>
    </set>
    ...
</class>
```

如果现在get()一个Item（或者强制初始化一个被代理的Item），seller对象和所有的bids都作为持久化实例被加载到你的持久化上下文中去：

```
Item item = (Item) session.get(Item.class, new Long(123));
```

这个调用之后的持久化上下文如图13-3所示。

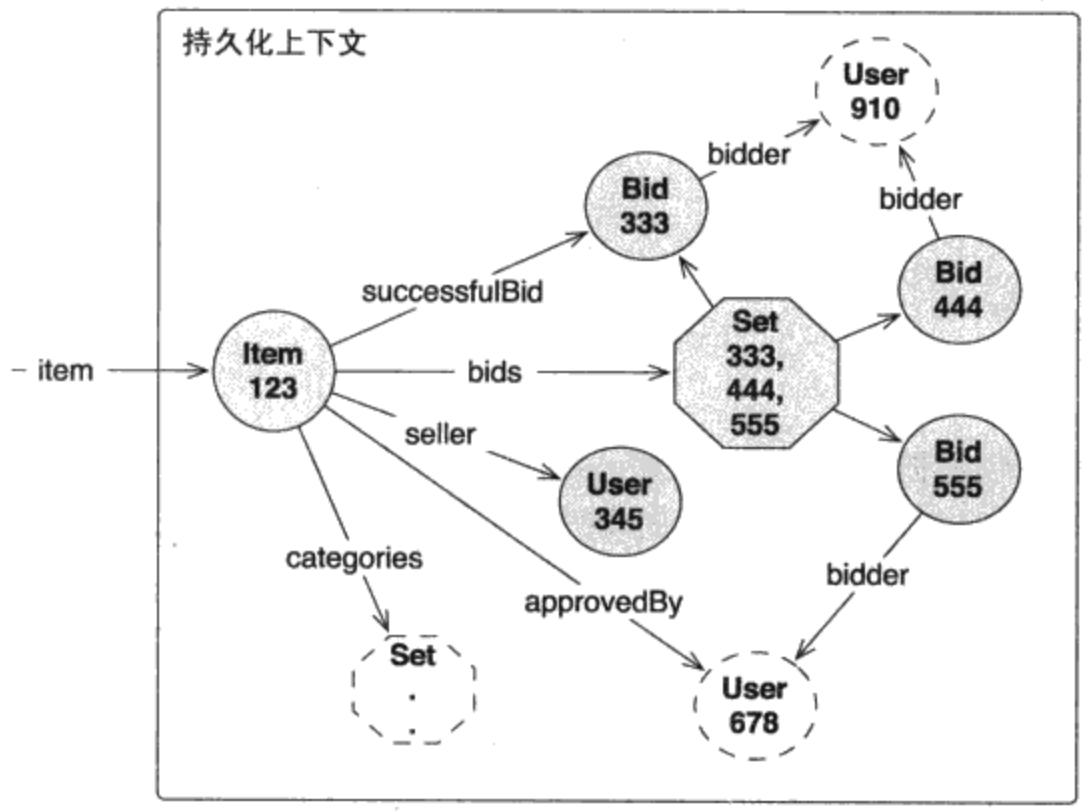


图13-3 通过禁用延迟关联和集合，即时抓取的一幅更大的图

其他延迟的被映射关联和集合（如每个Bid实例的bidder）还是没有被初始化，并且你一访问它们就立即被加载。想象你在加载Item之后关闭持久化上下文。现在可以在脱管状态下导航到Item的seller，并给这个Item遍历所有的bids。如果导航到这个Item要被分配到的categories，你就得到了LazyInitializationException！很显然，这个集合并非你抓取计划的一部分，并且没有在持久化上下文关闭之前初始化。如果试图访问一个代理（例如，核准项目的User），也会发生这种情况。（注意，访问这个代理可以用两种方法：通过approveBy和bidder引用。）

利用注解，转换一个实体关联或者集合的FetchType来得到同样的结果：

```
@Entity
public class Item {

    ...

    @ManyToOne(fetch = FetchType.EAGER)
    private User seller;

    @OneToMany(fetch = FetchType.EAGER)
    private Set<Bid> bids = new HashSet<Bid>();

    ...
}
```

在Hibernate中，FetchType.EAGER提供与lazy="false"同样的保证：被关联的实体实例必须被即时抓取，而不是延迟。我们已经注意到，Java Persistence有一项与Hibernate不同的默认抓取计划。虽然Hibernate中的所有关联都是完全延迟的，所有@ManyToOne和@OneToOne关联都默认为FetchType.EAGER！这项默认被标准化，允许Java Persistence提供程序不通过延迟加载实现（在实践中，这样的持久化提供程序并没有太大的用处）。建议你通过在“对一”的关联映射中设置FetchType.LAZY，默认为Hibernate的延迟加载计划，并且只在必要时才覆盖它：

```
@Entity
public class Item {

    ...

    @ManyToOne(fetch = FetchType.LAZY)
    private User seller;

    ...
}
```

你现在知道如何创建抓取计划了；也就是说，如何定义持久化对象网络的哪一部分应该被获取到内存中。在介绍这些对象应该如何被加载的定义方式，以及如何优化将要被执行的SQL之前，我们想要示范另一种不依赖代理的延迟加载策略。

13.1.6 通过拦截延迟加载

Hibernate提供的运行时代理生成，是透明的延迟加载的一种极好的选择。这个实现公开的唯一必要条件是一个包，或者是必须被代理的类中公有可见的无参构造器，以及非final方法和类声明。在运行时，Hibernate生成一个充当代理类的子类；这用私有的构造器或者final实体类是不可能的。

另一方面，许多其他的持久化工具没有使用运行时代理：它们使用拦截。对于你为什么要在Hibernate中使用拦截而不是运行时代理生成，没有很好的解释。非私有的构造器条件的确没什么大不了。但是有两种情况，你可能不想使用代理：

- 运行时代理不完全透明的唯一情况是通过instanceof测试的多态关联。或者，你可能想要依据类型转换一个对象，但是无法进行，因为代理是运行时生成的子类的一个实例。7.3.1节已介绍过如何避免这个问题，以及如何处理这个问题。用拦截代替代理也可以解决这些问题。
- 代理和集合包装器只能被用于延迟加载实体关联和集合。它们无法用来延迟加载单独的标量属性或者组件。我们认为这种优化将没有什么用。例如，你通常不想要延迟加载Item的initialPrice。如果没有使用（a）非常大量的可选列，或者没有使用（b）包含了必须按需获取的大值的可选列，就没有必要在SQL中被选中的单独列的级别上进行优化。大值最好用定位器对象表示；按照定义，它们提供延迟加载，无需拦截。然而，拦截（通常还有代理）还可以帮助优化列读取。

让我们通过几个示例来讨论对延迟加载的拦截。

想象你不想利用User实体类的代理，但仍然想要把一个关联延迟加载到User（例如，Item

的seller）的这一好处。用no-proxy映射这个关联：

```
<class name="Item" table="ITEM">
    ...
    <many-to-one name="seller"
                 class="User"
                 column="SELLER_ID"
                 update="false"
                 not-null="true"
                 lazy="no-proxy"/>
    ...
</class>
```

lazy属性默认为proxy。通过设置no-proxy，你在告诉Hibernate对这个关联应用拦截：

```
Item item = (Item) session.get(Item.class, new Long(123));
User seller = item.getSeller();
```

第一行把一个Item对象获取到持久化状态中去。第二行访问Item的seller。对getSeller()的这个调用被Hibernate拦截，并触发正在讨论的该User的加载。注意代理如何比拦截更延迟：可以调用item.getSeller().getId()而不强制代理的初始化。如果你只想要设置引用，这就使得拦截没有多大用处了，就像我们前面讨论过的。

也可以延迟加载通过<property>或者<component>映射的属性；在Hibernate XML映射中，这里启用拦截的属性是lazy="true"。利用注解，@Basic(fetch = FetchType.LAZY)是给Hibernate的一个提示：有一个属性或者组件应该通过拦截被延迟加载。

为了利用注解给关联禁用代理并启用拦截，必须依赖Hibernate扩展：

```
@ManyToOne
@JoinColumn(name="SELLER_ID", nullable = false, updatable = false)
@org.hibernate.annotations.LazyToOne(
    org.hibernate.annotations.LazyToOneOption.NO_PROXY
)
private User seller;
```

为了启用拦截，类的字节码必须在编译之后、运行时之前被使用。Hibernate提供一个Ant任务用于这一目的：

```
<target name="instrument" depends="compile">

    <taskdef name="instrument"
             classname=
➥ "org.hibernate.tool.instrument.cglib.InstrumentTask"
             classpathref="project.classpath"/>

    <instrument verbose="true">
        <fileset dir="${build.dir}/my/entity/package/">
            <include name="*.class"/>
        </fileset>
    </instrument>

</target>
```

你自己要决定是否想给延迟加载使用拦截——依据我们的经验，好的用例很少。

一般来说，你不仅应当定义持久化对象网络的什么部分必须被加载，还应当定义这些对象如何获取。除了创建抓取计划之外，你还应当通过正确的抓取策略对它进行优化。

13.2　选择抓取策略

Hibernate执行SQL SELECT语句把对象加载到内存。如果加载一个对象，那么就要执行单个或者几个SELECT，这取决于所涉及的表的数量，以及你所采用的抓取策略。

你的目标是把SQL语句的数量减到最少，简化SQL语句，以便尽可能地提高查询效率。你通过给每个集合或者关联应用最佳的抓取策略来做到这一点。让我们一步步地探讨不同的选项。

默认情况下，每当访问被关联的对象和集合时（如果你使用Java Persistence，假设你映射所有的“对一”关联为FetchType.LAZY），Hibernate就延迟抓取它们。看看下列这个繁琐的代码示例：

```
Item item = (Item) session.get(Item.class, new Long(123));
```

你没有配置任何关联或者集合为非延迟，也没有配置可以为所有关联生成的代理。因而，这项操作生成了下列SQL SELECT语句：

```
select item.* from ITEM item where item.ITEM_ID = ?
```

（注意，Hibernate生成的真正SQL自动包含生成的别名；为了增加可读性，我们已经在接下来的所有示例中把它们移除了。）你会看到SELECT只查询ITEM表，并获取一个特定的行。所有实体关联和集合都没有被获取。如果你访问任何被代理的关联或者未被初始化的集合，就会执行第二个SELECT以便按需获取数据。

第一个优化步骤是，减少利用默认的延迟行为时定会看到的额外按需SELECT的数量——例如，通过预抓取数据。

13.2.1　批量预抓取数据

如果每一个实体关联和集合都仅仅按需抓取，完成一个特定的过程可能就需要许多额外的SQL SELECT语句了。例如，考虑下列获取所有Item对象的查询，并访问每件货品seller的数据：

```
List allItems = session.createQuery("from Item").list();

processSeller( (Item)allItems.get(0) );
processSeller( (Item)allItems.get(1) );
processSeller( (Item)allItems.get(2) );
```

一般来说，你在这里用了一个循环，并遍历结果，但这段代码所公开的问题是一样的。你看到1个SQL SELECT获取所有的Item对象，并且你一处理Item的每一个seller时立即就有另一个SELECT。所有被关联的User对象都是代理。这是我们将要深入阐述的最糟糕的案例场景之一：n+1查询问题。SQL看起来像这样：

```
select items...

select u.* from USERS u where u.USER_ID = ?
select u.* from USERS u where u.USER_ID = ?
```

```
select u.* from USERS u where u.USER_ID = ?
...
```

Hibernate提供了一些算法，可以预抓取User对象。我们现在讨论的第一个优化称作批量抓取（batch fetching），它像这样进行：如果User的1个代理必须被初始化，就在相同的SELECT中初始化几个代理。换句话说，如果你已经知道持久化上下文中有3个Item实例，并且它们已经把一个代理应用到了它们的seller关联，那么你可以也初始化所有代理而不只是1个。

批量抓取经常被称作瞎猜优化（blind-guess optimization），因为你不知道在一个特定的持久化上下文中，有多少个未被初始化的User代理。在前一个例子中，这个数量取决于返回的Item对象的数量。你猜测并把批量大小（batch-size）抓取策略应用到User类映射：

```
<class name="User"
       table="USERS"
       batch-size="10">
...
</class>
```

你在告诉Hibernate：如果必须初始化1个代理，就在单个SQL SELECT中最多预抓取10个未被初始化的代理。之前的查询和过程所产生的SQL现在看起来可能像这样：

```
select items...

select u.* from USERS u where u.USER_ID in (?, ?, ?)
```

获取所有Item对象的第一个语句在你list()查询的时候执行。下一个语句（获取3个User对象）在你一初始化由allItems.get(0).getSeller()返回的第一个代理时，就会被触发。这个查询同时加载3个卖主——因为这就是初始查询要返回的项目数量，以及当前的持久化上下文中未被初始化的代理数量。你定义了批量大小为“最多10个”。如果返回的货品超过10个，你就会看到第二个查询如何在一批中获取10个卖主。如果应用程序命中另一个尚未被初始化的代理，就会获取到另一批的10个，等等，直到持久化上下文中再也没有未被初始化的代理，否则应用程序停止访问被代理的对象。

常见问题 什么是真正的批量抓取算法？可以回想一下前面阐述过的批量抓取，但是如果你在实践中体验过，就会发现一种稍微不同的算法。是否想了解和理解这种算法，或者你是否信任Hibernate会正确地完成，这都取决于你。举个例子，想象一个批量大小为20，未被初始化的代理总数量为119，那么它们必须以多批加载。启动时，Hibernate读取映射元数据，并在内部创建11个批量加载器。每个加载器都知道它可以初始化多少个代理：20，10，9，8，7，6，5，4，3，2，1。目标是把加载器创建的内存消耗减到最少，并创建足够的可以生成每一个可能的批量抓取的加载器。另一个目标显然是把SQL SELECT的数量减到最少。为了初始化119个代理，Hibernate执行了7批（你或许以为是6批，因为6 × 20>119）。应用程序的批量加载器有五次为20个，一次为10个，一次为9个，由Hibernate自动选择。

批量抓取也可用于集合：

```
<class name="Item" table="ITEM">
```

```
    ...
    <set name="bids"
         inverse="true"
         batch-size="10">
        <key column="ITEM_ID"/>
        <one-to-many class="Bid"/>
    </set>

</class>
```

如果现在强制初始化1个bids集合，则立即加载相同类型的集合最多10个（如果它们在当前的持久化上下文中没有被初始化）：

```
select items...

select b.* from BID b where b.ITEM_ID in (?, ?, ?)
```

在这个例子中，你再一次拥有了3个处于持久化状态的Item对象，并且接触到了其中一个未被加载的bids集合。现在所有3个Item对象都让它们的bids在单个SELECT中加载。

对实体代理和集合的批量大小设置也可以通过注解使用，但是只能作为Hibernate扩展：

```
@Entity
@Table(name = "USERS")
@org.hibernate.annotations.BatchSize(size = 10)
public class User { ... }

@Entity
public class Item {
    ...
    @OneToMany
    @org.hibernate.annotations.BatchSize(size = 10)
    private Set<Bid> bids = new HashSet<Bid>();

    ...
}
```

通过批量策略预抓取代理和集合真的是一种瞎猜。正是这种聪明的优化可以显著地减少（对于初始化所使用的所有对象是必需的）SQL语句的数量。当然，预抓取的唯一缺点是，可能预抓取了最终不需要的数据。用更少的SQL语句的代价可能是更高的内存消耗。后者经常更为重要：内存很便宜，但是可伸缩的数据库服务器则不便宜。

另一种并非瞎猜的预抓取算法是使用子查询，通过单个语句初始化多个集合。

13.2.2　通过子查询预抓取集合

最后举一个示例，并应用一种（或许）更好的预抓取优化：

```
List allItems = session.createQuery("from Item").list();

processBids( (Item)allItems.get(0) );
processBids( (Item)allItems.get(1) );
processBids( (Item)allItems.get(2) );
```

你用一个初始的SQL SELECT获取所有的Item对象，并在每个bids集合被访问的时候用一个额外的SELECT。改善这种方法的一种可能是通过批量抓取；然而，需要算出试用的最佳批量大小。

一种更好的优化是这个集合映射的子查询抓取：

```
<class name="Item" table="ITEM">
    ...
    <set name="bids"
         inverse="true"
         fetch="subselect">
        <key column="ITEM_ID"/>
        <one-to-many class="Bid"/>
    </set>

</class>
```

一旦你强制初始化1个bids集合，Hibernate现在就立即给所有被加载的Item对象初始化所有bids集合。它通过在一个子查询中重新运行（被稍微修改的）第一个初始查询来完成：

```
select i.* from ITEM i

select b.* from BID b
    where b.ITEM_ID in (select i.ITEM_ID from ITEM i)
```

在注解中，必须再次使用一个Hibernate扩展启用这个优化：

```
@OneToMany
@org.hibernate.annotations.Fetch(
    org.hibernate.annotations.FetchMode.SUBSELECT
)
private Set<Bid> bids = new HashSet<Bid>();}
```

利用子查询预抓取是一种强大的优化；稍后当我们探讨一种典型的场景时，将介绍更详细的内容。在编写本书之时，子查询抓取只可用于集合，而不可用于实体代理。还要注意，作为子查询重新运行的原始查询，只被Hibernate为一个特定Session而记住。如果你分离一个Item实例，而没有初始化bids的集合，那么重附它并启动对该集合的遍历，则不会发生其他集合的预抓取。

如果你努力减少额外SELECT（对于利用延迟加载并按需获取对象和集合，它是自然要用的）的数量，那么所有前面的抓取策略都有帮助。最后一种抓取策略与按需获取相反。你经常想要在同一个初始的SELECT中，通过一个JOIN获取被关联的对象或者集合。

13.2.3 通过联结即时抓取

延迟加载是一种极好的默认策略。另一方面，你可以经常看看领域模型和数据模型，并说“每次我需要Item的时候，也需要该Item的seller”。如果可以把它变成语句，就应该到映射元数据中去，给seller关联启用即时抓取（eager fetching），并利用SQL联结：

```
<class name="Item" table="ITEM">
    ...
    <many-to-one name="seller"
                 class="User"
                 column="SELLER_ID"
                 update="false"
                 fetch="join"/>

</class>
```

现在Hibernate在单个SQL语句中把Item和它的seller都加载了。例如：

```
Item item = (Item) session.get(Item.class, new Long(123));
```

这个操作触发了下列SQL SELECT：

```
select i.*, u.*
from ITEM i
    left outer join USERS u on i.SELLER_ID = u.USER_ID
where i.ITEM_ID = ?
```

很显然，seller不再被按需延迟加载，而是立即加载。因此，fetch="join"禁用了延迟加载。如果你只用lazy="join"启用即时抓取，会立即看到第二个SELECT。通过fetch="join"，在同一个SELECT中加载seller。这个查询的结果集如图13-4所示。

select i.*, u.* from ITEM i
left outer join USERS u on i.SELLER_ID = u.USER_ID
where i.ITEM_ID = ?

ITEM_ID	DESCRIPTION	SELLER_ID	...	USER_ID	USERNAME	...
1	Item Nr. One	2	...	2	johndoe	...

图13-4　联结两张表来即时抓取被关联的行

Hibernate读取这个行，并封送来自结果的两个对象。它利用从Item到User的一个引用即seller关联，把它们连接起来。如果Item没有卖主，所有u.*列都会被填上NULL。这就是为什么Hibernate使用外部联结的原因，因此它不仅可以获取有卖主的Item对象，而且可以获取全部Item对象。但是你知道，在CaveatEmptor中Item必须有一个seller。如果你启用<many-to-one not-null="true"/>，Hibernate就会执行一个内部联结而不是外部联结。

也可以在集合上设置即时联结抓取策略：

```
<class name="Item" table="ITEM">
    ...
    <set name="bids"
         inverse="true"
         fetch="join">
        <key column="ITEM_ID"/>
        <one-to-many class="Bid"/>
    </set>

</class>
```

如果现在加载许多个Item对象，例如通过creatCriteria(Item.class).list()，产生的SQL语句看起来像这样：

```
select i.*, b.*
from ITEM i
    left outer join BID b on i.ITEM_ID = b.ITEM_ID
```

现在这个结果集包含许多行，每个有许多个bids的Item都有重复数据，并且NULL过滤掉所有没有bids的Item对象。看看图13-5中的结果集。

select i.*, b.* from ITEM i
left outer join BID b on i.ITEM_ID = b.ITEM_ID

ITEM_ID	DESCRIPTION	...	BID_ID	ITEM_ID	AMOUNT
1	Item Nr. One	...	1	1	99.00
1	Item Nr. One	...	2	1	100.00
1	Item Nr. One	...	3	1	101.00
2	Item Nr. Two	...	4	2	4.99
3	Item Nr. Three	...	NULL	NULL	NULL

图13-5 被关联的集合元素的外部联结抓取

Hibernate创建了3个持久化的Item实例，以及4个Bid实例，并在持久化上下文中把它们全部链接在一起，以便你可以导航这个对象图，并遍历集合——甚至在持久化上下文被关闭，且所有对象被脱管的时候。

使用内部联结即时抓取集合在概念上是可能的，并且我们稍后也将在HQL查询中完成。然而，在映射元数据中去掉所有在全局抓取策略中没有bids的Item对象并没有意义，因此没有选项用于集合的全局内部联结即时抓取。

利用Java Persistence注解，通过FetchType注解属性启用即时抓取：

```
@Entity
public class Item {
    ...
    @ManyToOne(fetch = FetchType.EAGER)
    private User seller;

    @OneToMany(fetch = FetchType.EAGER)
    private Set<Bid> bids = new HashSet<Bid>();

    ...
}
```

这个映射示例看起来应该很熟悉：你前面用它来禁用关联和集合的延迟加载。Hibernate把它默认当作一个不应该通过立即的第二个SELECT（而是通过初始查询中的一个JOIN）执行的即时抓取。

可以保留FetchType.EAGER这个Java Persistence注解，但是通过显式添加一个Hibernate扩展注解，从联结抓取转换到一个立即的第二个查询：

```
@Entity
public class Item {

    ...

    @ManyToOne(fetch = FetchType.EAGER)
    @org.hibernate.annotations.Fetch(
        org.hibernate.annotations.FetchMode.SELECT
    )
    private User seller;

}
```

如果一个Item实例被加载，Hibernate将通过一个立即的第二个SELECT，即时加载这件货品的卖主。

最后，必须介绍一个全局的、可以用来控制被联结的实体关联（不是集合）最大数量的Hibernate配置设置。考虑你已经在映射元数据中设置所有多对一和一对多关联映射为fetch="join"（或者FetchType.EAGER）。假设Item有一个successfulBid关联，Bid有一个bidder，并且这个User有一个shippingAddress。如果所有这些关联都通过fetch="join"映射，那么当你加载Item时，要联结多少张表，要获取多少数据？

在这种情况下，被联结的表的数量取决于全局的hibernate.max_fetch_depth配置属性。默认情况下，没有设置限制，因此加载Item也在单个查询中获取Bid、User和Address。合理的设置很小，通常在1～5之间。你甚至可以通过设置该属性为0，给多对一和一对一的关联禁用联结抓取！（注意有些数据库方言可以预设这项属性，如MySQLDialect把它设置为2。）

如果涉及继承或者被联结的映射，SQL查询也会变得更加复杂。每当给一个特定的实体类映射二级表时，你需要考虑几个额外的优化选项。

13.2.4　给二级表优化抓取

如果你查询的对象属于继承层次结构一部分的类，SQL语句就变得更加复杂了：

```
List result = session.createQuery("from BillingDetails").list();
```

这项操作获取所有的BillingDetails实例。SQL SELECT现在取决于你已经给BillingDetails和它的子类CreditCard和BankAccount选择的继承映射策略。假设你已经把它们全部映射到一张表（每个层次结构一张表，table-per-hierarchy），这个查询与前一节介绍过的没有什么区别。然而，如果你已经利用隐式多态映射了它们，这个单个的SQL操作就可能根据每个子类的每张表产生几个SQL SELECT。

1. “每个子类一张表”层次结构的外部联结

如果以标准的方式映射层次结构（请见5.1.4节中的表和映射），那么在这个初始语句中，所有子类表都被外部联结：

```
select
    b1.BILLING_DETAILS_ID,
    b1.OWNER,
    b1.USER_ID,
    b2.NUMBER,
    b2.EXP_MONTH,
    b2.EXP_YEAR,
    b3.ACCOUNT,
    b3.BANKNAME,
    b3.SWIFT,
    case
        when b2.CREDIT_CARD_ID is not null then 1
        when b3.BANK_ACCOUNT_ID is not null then 2
        when b1.BILLING_DETAILS_ID is not null then 0
    end as clazz
from
```

```
        BILLING_DETAILS b1
    left outer join
        CREDIT_CARD b2
            on b1.BILLING_DETAILS_ID = b2.CREDIT_CARD_ID
    left outer join
        BANK_ACCOUNT b3
            on b1.BILLING_DETAILS_ID = b3.BANK_ACCOUNT_ID
```

这已经是个值得关注的查询了。它联结3张表，并利用一个CASE ...WHEN ... END表达式，用0~2之间的一个数填到clazz列中。然后Hibernate可以读取结果集，并在这个数字的基础上，决定被返回的每一行表示什么类的一个实例。

许多数据库管理系统限制可以与OUT JOIN合并的表的最大数目。如果你用标准的策略映射了一个很宽很深的继承层次结构，就可能碰到这一限制（我们正在讨论应该被重新考虑的继承层次结构以适应你毕竟正在使用的是SQL数据库的事实）。

2. 转换到额外的选择

然后在映射元数据中，你可以告诉Hibernate转换到一种不同的抓取策略。你想要继承层次结构的一些部分通过立即的额外SELECT语句而被抓取，而不是通过初始查询中的OUT JOIN。

启用这种抓取策略的唯一方式是稍微重构映射，作为使用了辨别标志列的每个层次结构一张表和使用了<join>映射每个子类一张表的混合：

```
<class name="BillingDetails"
       table="BILLING_DETAILS"
       abstract="true">

    <id name="id"
        column="BILLING_DETAILS_ID"
        .../>

    <discriminator
        column="BILLING_DETAILS_TYPE"
          type="string"/>

    ...
    <subclass name="CreditCard" discriminator-value="CC">
        <join table="CREDIT_CARD" fetch="select">
            <key column="CREDIT_CARD_ID"/>
            ...
        </join>
    </subclass>

    <subclass name="BankAccount" discriminator-value="BA">
        <join table="BANK_ACCOUNT" fetch="join">
            <key column="BANK_ACCOUNT_ID"/>
            ...

            </join>
    </subclass>

</class>
```

这个映射把CreditCard和BankAccount类分开到它们各自的表中，但是在超类表中保存辨别标志列。CreditCard对象的抓取策略是select，而BankAccount的策略则是默认的join。现

在，如果查询所有的BillingDetails，就会生成下列SQL：

```
select
    b1.BILLING_DETAILS_ID,
    b1.OWNER,
    b1.USER_ID,
    b2.ACCOUNT,
    b2.BANKNAME,
    b2.SWIFT,
    b1.BILLING_DETAILS_TYPE as clazz
from
    BILLING_DETAILS b1
    left outer join
        BANK_ACCOUNT b2
            on b1.BILLING_DETAILS_ID = b2.BANK_ACCOUNT_ID

select cc.NUMBER, cc.EXP_MONTH, cc.EXP_YEAR
from CREDIT_CARD cc where cc.CREDIT_CARD_ID = ?

select cc.NUMBER, cc.EXP_MONTH, cc.EXP_YEAR
from CREDIT_CARD cc where cc.CREDIT_CARD_ID = ?
```

第一个SQL SELECT从超类表和BANK_ACCOUNT表中获取所有的行。它也给每一行返回辨别标志值作为clazz列。Hibernate现在根据CREDIT_CARD表给第一个结果（这个结果有着CreditCard的正确辨别标志）的每一行执行一个额外的select。换句话说，两个查询意味着BILLING_DETAILS超类表中的两个行表示一个CreditCard对象（的一部分）。

这种优化几乎没有必要，但你现在也知道，每当处理<join>映射时，可以从一个默认的join抓取策略转换到一个额外的立即的select。

现在，我们已经通过你可以在映射元数据中设置的所有选项，来影响默认抓取计划和抓取策略的探讨。你学习了如何定义什么应该通过操作lazy属性进行加载，以及它应该如何通过设置fetch属性加载。在注解中，你使用FetchType.LAZY和FetchType.EAGER，并使用Hibernate扩展实现对抓取计划和策略更细粒度的控制。

知道了所有可用的选项只是实现优化的和有效的Hibernate或者Java Persistence应用程序的一小步。你还需要知道应用一种特定的策略的时机。

13.2.5　优化指导方针

默认情况下，Hibernate从不加载你没有请求的数据，这样减少了持久化上下文的内存消耗。然而，它也会让你面临所谓的n+1查询问题。如果每一个关联和集合都只按需初始化，并且没有配置其他的策略，特定的过程也可以很好地执行几十甚至几百个查询，以获得你需要的所有数据。你需要正确的策略来避免执行过多的SQL语句。

如果从默认的策略转换到通过联结即时抓取数据的查询，就可能遇到另一个问题：笛卡儿积问题。无须执行过多的SQL语句，相反现在可以（经常作为一种附带作用）创建获取过多数据的语句。

要找到这两个极端之间的中间地带：每个过程的正确抓取策略，以及你应用程序中的用例。你需要知道应该在映射元数据中设置哪种全局的抓取计划和策略，以及只对一个特定的查询应用

哪种抓取策略（通过HQL还是Criteria）。

现在介绍有关过多查询和笛卡儿积的几个基本问题，然后引导你一步步地完成优化。

1. n+1查询问题

n+1查询问题很容易通过一些示例代码理解。假设你没有在映射元数据中配置任何抓取计划或者抓取策略：所有的东西都是延迟的，且按需加载。下列示例代码试图给所有Item找到最高的Bid（当然，有许多其他的方法可以更轻松地实现这一点）：

```
List<Item> allItems = session.createQuery("from Item").list();
// List<Item> allItems = session.createCriteria(Item.class).list();

Map<Item, Bid> highestBids = new HashMap<Item, Bid>();

for (Item item : allItems) {
    Bid highestBid = null;
    for (Bid bid : item.getBids() ) { // Initialize the collection
        if (highestBid == null)
            highestBid = bid;
        if (bid.getAmount() > highestBid.getAmount())
            highestBid = bid;
    }
    highestBids.put(item, highestBid);
}
```

首先，获取所有Item实例；这在HQL和Criteria查询之间没有区别。这个查询触发1个SQL SELECT，获取ITEM表的所有行，并返回n个持久化对象。接下来，你遍历这个结果，并访问每一个Item对象。

你访问的是每个Item的bids集合。这个集合目前为止还没有被初始化，每件货品的Bid对象都必须通过一个额外的查询来加载。这整个代码片段因此产生了n+1查询。

你永远都想要避免n+1查询。

第一种解决方案可能是给集合改变你的全局映射元数据，启用批量预抓取：

```
<set name="bids"
     inverse="true"
     batch-size="10">
    <key column="ITEM_ID"/>
    <one-to-many class="Bid"/>
</set>
```

不是n+1查询，你现在看到n/10+1查询在把必要的集合获取到内存中。这项优化对于一个拍卖应用程序来说似乎很合理："只在需要一件货品的出价时才按需加载它们。但是如果出价的1个集合必须针对某件特定的货品被加载，那么就假设持久化上下文中的其他货品对象也需要初始化它们的出价集合。批量完成这项工作，因为似乎并非所有的货品对象都需要它们的出价。"

利用一个基于子查询的预抓取，可以把选择的数量减到正好两个：

```
<set name="bids"
     inverse="true"
     fetch="subselect">
    <key column="ITEM_ID"/>
    <one-to-many class="Bid"/>
</set>
```

过程中的第一个查询现在执行单个SQL SELECT，获取所有Item实例。Hibernate记住这个语句，并当你命中第一个未被初始化的集合时，再次应用它。所有的集合都通过第二个查询被初始化。这项优化的原因略有不同："只在需要一件货品出价的时候才按需加载它们。但是如果出价的1个集合必须针对某件特定货品加载，那么就假设持久化上下文中的所有其他货品对象也需要初始化它们的出价集合。"

最后，可以有效地关闭bids集合的延迟加载，并转换到只导致单个SQL SELECT的一种即时抓取策略：

```
<set name="bids"
     inverse="true"
     fetch="join">
    <key column="ITEM_ID"/>
    <one-to-many class="Bid"/>
</set>
```

这似乎是一项你不应该进行的优化。你真的能说"每当需要一件货品时，也需要它的所有出价"吗？映射元数据中的抓取策略在一个全局的级别中工作。我们不把fetch="join"当作对集合映射的一种常用优化；你始终很少需要完全被初始化的集合。除了导致更高的内存消耗之外，每一个被外部联结的集合也都导致了更严重的笛卡儿积问题，我们很快就会对它进行更深入的讨论。

在实践中，你最可能在映射元数据中给bids集合启用一个批量的或者子查询策略。如果一个特定的过程（比如这个）需要内存中每一个Item的所有bids，你修改初始的HQL或者Criteria查询，并应用一种动态的抓取策略：

```
List<Item> allItems =
    session.createQuery("from Item i left join fetch i.bids")
           .list();

List<Item> allItems =
    session.createCriteria(Item.class)
        .setFetchMode("bids", FetchMode.JOIN)
        .list();

// Iterate through the collections...
```

这两个查询通过一个OUTER JOIN，为所有Item实例都生成了获取bids的单个SELECT（就像如果你已经用fetch="join"映射了这个集合时发生的那样）。

这可能是你第一次见到如何定义一个非全局的抓取策略。你放在映射元数据中的全局抓取计划和抓取策略设置只是始终应用的全局默认。任何优化过程也都需要更细粒度的规则，即只适用于特定的过程或者用例的抓取策略和抓取计划。下一章将更详细地讨论使用HQL和Criteria的抓取。你现在需要知道的就是有这些选项的存在。

使用延迟的集合时，n+1查询问题更频繁地出现。未被初始化的代理公开了相同的行为：你可能需要许多个SELECT来初始化正在特定过程中使用的所有对象。我们已经介绍过的优化指导方针是一样的，但是有一点不同：在<many-to-one>或者<one-to-one>关联中设置的fetch="join"是一般的优化，就像@ManyToOne(fetch=FetchType.EAGER)注解（Java Persistence中的默认）一样。单端关联的即时联结抓取，不同于集合的即时外部联结抓取，不会产生笛卡儿积问题。

2. 笛卡儿积问题

与n+1查询问题相反的是抓取过多数据的SELECT语句。如果试图抓取几个“并行的”集合，少不了会出现这个笛卡儿积问题。

假设你已经决定把一个全局的fetch="join"设置应用到Item的一个bids集合（尽管我们建议只在必要时才使用全局的预抓取和动态联结抓取策略）。Item类有其他的集合（例如images）。也假设你决定每件货品的所有图片都必须即时通过fetch="join"策略始终被加载：

```
<class name="Item">
    ...
    <set name="bids"
         inverse="true"
         fetch="join">
        <key column="ITEM_ID"/>
        <one-to-many class="Bid"/>
    </set>
    <set name="images"
         fetch="join">
        <key column="ITEM_ID"/>
        <composite-element class="Image">...
    </set>
</class>
```

如果通过一个即时的外部联结抓取策略映射两个并行的集合（它们自己的实体也一样），并加载所有的Item对象，Hibernate就执行一个创建这两个集合的乘积的SQL SELECT：

```
select item.*, bid.*, image.*
  from ITEM item
    left outer join BID bid on item.ITEM_ID = bid.ITEM_ID
    left outer join ITEM_IMAGE image on item.ITEM_ID = image.ITEM_ID
```

看看这个查询的结果集，如图13-6所示。

select item.*, bid.*, image.* from ITEM item
left outer join BID bid on item.ITEM_ID = bid.ITEM_ID
left outer join ITEM_IMAGE image on item.ITEM_ID = image.ITEM_ID

ITEM_ID	DESCRIPTION	...	BID_ID	ITEM_ID	AMOUNT	...	IMAGE_NAME
1	Item Nr. One	...	1	1	99.00	...	foo.jpg
1	Item Nr. One	...	1	1	99.00	...	bar.jpg
1	Item Nr. One	...	2	1	100.00	...	foo.jpg
1	Item Nr. One	...	2	1	100.00	...	bar.jpg
1	Item Nr. One	...	3	1	101.00	...	foo.jpg
1	Item Nr. One	...	3	1	101.00	...	bar.jpg
2	Item Nr. Two	...	4	2	4.99	...	baz.jpg
3	Item Nr. Three	...	NULL	NULL	NULL	...	NULL

图13-6 乘积是带有许多行的两个外部联结的结果

这个结果集包含大量多余的数据。1号货品有3个出价和2张图片，2号货品有1个出价和1张图片，3号货品没有出价和图片。这个乘积的大小取决于正在获取的集合的大小：3次2，1次1，再加1，共8个结果行。现在想象你在数据库中有1 000件货品，每件货品有20个出价和5张图片——你会看到一个约有100 000行的结果集！这个结果的大小可能有好几兆。在数据库中创建这个结果集需要的处理时间不仅长，而且需要的内存也相当大。所有数据都必须通过网络转移。当Hibernate把结果集封送到持久化对象和集合的时候，它立即移除所有重复——跳过多余的信息。3个查询当然更快了！

如果用`fetch="subselect"`映射这个并行的集合，就会得到3个查询；这就是对并行集合所建议的优化。然而，每一种规则都有例外。只要集合很小，乘积就可能是可以接受的抓取策略。注意，通过外部联结`SELECT`即时抓取的并行的单值关联一般不会产生乘积。

最后，虽然Hibernate让你通过`fetch="join"`在2个（或者甚至更多个）并行的集合中创建笛卡儿积，但是如果你试图在并行的`<bag>`集合上启用`fetch="join"`，它就会抛出异常。乘积的结果集无法被转化为包集合，因为Hibernate无法知道哪些行包含有效的重复（包允许重复）以及哪些行不包含。如果使用包集合（它们是Java Persistence中默认的`@OneToMany`集合），就不启用导致乘积的抓取策略。给包集合的并行即时抓取使用子查询或者立即的二级查询抓取。

全局和动态的抓取策略帮助你解决n+1查询和笛卡儿积问题。Hibernate提供另一个选项来初始化一个有时候有用的代理或者集合。

3．强制初始化代理和集合

代理或者集合包装器，每当调用它的任何方法时都被自动初始化（除了标识符属性获取方法之外，它可以返回标识符值，而不抓取底层的持久化对象）。预抓取和即时联结抓取都是获取你需要的所有数据的可能的解决方案。

你有时候想要在脱管状态下使用对象网络。获取所有应该脱管的对象和集合，然后关闭持久化上下文。

在这个场景中，在关闭持久化上下文之前显式地初始化一个对象有时候有用，而不用求助于改变全局的抓取策略或者不同的查询（我们认为这是你应该始终首选的解决方案）。

可以使用静态的方法`Hibernate.initialize()`给代理进行手工初始化：

```
Session session = sessionFactory.openSession();
Transaction tx = session.beginTransaction();

Item item = (Item) session.get(Item.class, new Long(1234));

Hibernate.initialize( item.getSeller() );

tx.commit();
session.close();

processDetached( item.getSeller() );
...
```

`Hibernate.initialize()`可以传递一个集合包装器或者代理。注意，如果你把一个集合包装器传递到`initialize()`，它不会初始化这个集合所引用的目标实体对象。在前一个示例中，`Hibernate.initialize(item.getBids())`不加载该集合内部的所有`Bid`对象。它用`Bid`对象的

代理初始化该集合！

几乎没有必要用这个静态的辅助方法来显式地初始化；你应该始终首选使用HQL或者Criteria的动态抓取。

既然你知道了所有的选项、问题和可能性，让我们来完成一个典型的应用程序优化过程。

4．逐步优化

首先，启用Hibernate SQL日志。你也应该准备针对特定的数据库Schema而阅读、理解和评估SQL查询及其性能特征：单个外部联结操作比两个查询更好吗？所有的索引都使用得当吗？以及什么是数据库内部的高速缓存命中率？让数据管理员帮助你做性能评估；只要他具备关于最好的SQL执行计划的知识。（如果想变成这个领域的专家，我们推荐这本书：Dan Tow著的*SQL Tunning* [Tow，2003]。）

这两个配置属性hibernate.format_sql和hibernate.use_sql_comments，使得在日志文件中阅读和分类SQL语句变得容易多了。优化期间启用这两个。

接下来，逐个执行你应用程序的用例，并注意Hibernate执行了多少个以及哪些SQL语句。用例可以是Web应用程序的单个屏幕，或者是一系列用户对话框。这个步骤也涉及了收集你在每个用例中使用的对象获取方法：通过对象链接，通过标识符、HQL和Criteria查询获取。你的目标是通过调优元数据中的默认抓取计划和抓取策略，把每个用例的SQL语句数量（和复杂性）降下来。

现在该定义抓取计划了。默认情况下的一切都是延迟加载的。考虑转换到多对一、一对一和（有时）集合映射中的lazy="false"（或者FetchType.EAGER）。全局的抓取计划定义了始终被即时加载的对象。优化你的查询，并且如果你需要即时（不是全局）地加载对象，就启用即时抓取，但要在特定的过程（仅有的一个用例）中。

一旦定义了抓取计划，并且知道一个特定的用例所需要的数据量，就优化这个数据获取的方式。你可能遇到两个常见的问题：

- SQL语句使用太复杂且太慢的联结操作。首先和你的数据库管理员一起优化SQL执行计划。如果这样还没有解决问题，就移除集合映射中的fetch="join"（或者先不设置它）。通过考虑它们是真正需要fetch="join"策略，还是被关联的对象应该通过二级查询进行加载，来最优化你的多对一和一对一关联。也可以通过全局的hibernate.max_fetch_depth配置选项来尝试调优，但是记住这个值最好保持在1～5之间。
- 可能执行过多的SQL语句。在多对一和一对一关联映射中设置fetch="join"。在极少数情况下，如果你绝对确定，就启用fetch="join"，禁用对特定集合的延迟加载。记住，每个持久化类都有不止一个被即时抓取的集合会导致乘积。利用批量或者子查询，来评估你的用例是否能从集合的预抓取中受益。使用3～15之间的批量大小。

设置了一种新的抓取策略之后，重新运行用例，并再次检查生成的SQL。注意这些SQL语句，并进行下一个用例。优化了所有用例之后，再次检查每一个用例，看看是否任何全局优化对于其他的会有副作用。凭借一些经验，你将第一次就能够很容易地避免任何负面的影响，并得到想要的结果。

这种优化技术不仅仅对于默认的抓取策略实用；也可以用它调优HQL和Criteria查询，它们可以动态地定义抓取计划和抓取策略。你经常能够用一个新的动态查询或者现有查询的一种变形，替换全局的抓取设置——关于这些选项，下一章将进行更加深入的讨论。

下一节介绍Hibernate高速缓存系统。在应用层中高速缓存数据，是你能够在任何复杂的多用户应用程序中使用的一种补充优化。

13.3　高速缓存基本原理

我们之所以认为使用对象/关系持久层远胜于用直接的JDBC创建的应用程序，一个重要的理由在于：高速缓存的潜能。虽然我们对大多数的应用程序在不使用高速缓存的情况下应该可能实现满意的性能有严重的分歧，但对于某些类型的应用程序，尤其对于主要用于读取的应用程序或者在数据库中保存重大元数据的应用程序来说，这是毫无疑义的，因为高速缓存会对性能造成巨大的影响。此外，把一个高并发的应用程序扩展到上千个在线事务，通常需要一些高速缓存来减少（多个）数据库服务器中的负荷。

我们通过一些背景信息开始对高速缓存的探讨。这包括阐述不同的高速缓存和同一性范围，以及高速缓存在事务隔离性中的影响。这种信息和这些规则一般来说可以应用到高速缓存，并且不仅仅只对Hibernate应用程序有效。这个讨论给你提供了一个背景，让你理解为什么Hibernate高速缓存系统是这样的。然后介绍Hibernate高速缓存系统，并教你如何启用、调优和管理一级和二级Hibernate高速缓存。建议你在开始使用高速缓存之前，认真学习本节中安排的基本原理。没有这些基础，你可能很快遇到难以调试的并发问题和数据完整性的危险。

高速缓存就是性能优化，因此它自然不是Java Persistence或者EJB 3.0规范的一部分。每一个供应商都对优化提供不同的解决方案，尤其是任何二级高速缓存。本节中介绍的所有策略和选项都适用于原生的Hibernate应用程序或者依赖Java Persistence接口并使用Hibernate作为持久化提供程序的应用程序。

高速缓存使当前数据库状态的表示接近应用程序，要么在内存中，要么在应用程序服务器机器的磁盘上。高速缓存是数据的一个本地副本，处在应用程序和数据库之间，可以用来避免数据库命中，每当：

- 应用程序通过标识符（主键）执行查找的时候。
- 持久层延迟解析一个关联或者集合的时候。

也可能高速缓存查询的结果。如你将在第15章所见，高速缓存查询结果的性能收获在许多案例中都十分微小，因此这项功能很少使用。

在讨论Hibernate的高速缓存如何工作之前，先快速学习一下不同的高速缓存选项，并看看它们如何相关到同一性和并发。

13.3.1　高速缓存策略和范围

高速缓存的概念在对象/关系持久化中是如此基本，以致如果你不先知道它使用了哪（几）

种高速缓存策略，就无法理解性能、可伸缩性或者ORM实现的事务语义。有3种主要的高速缓存类型：

- 事务范围高速缓存——添加到当前的工作单元，它可能是一个数据库事务，甚至是一个对话。它只有在工作单元运行时才是有效的，才能被使用。每一个工作单元都有自己的高速缓存。这个高速缓存中的数据不会被并发访问。
- 过程范围高速缓存——在许多（可能并发的）工作单元或者事务之间共享。这意味着过程范围高速缓存中的数据被并发运行的线程访问，显然隐含着事务隔离性。
- 集群范围高速缓存——在同一台机器的多个过程之间或者一个集群中的多台机器上共享。这里，网络通信是一个值得考虑的关键点。

过程范围高速缓存可以把持久化实例本身保存在高速缓存中，或者可以用分解的格式只保存它们的持久化状态。然后，每个访问共享高速缓存的工作单元都从被高速缓存的数据中重新组合一个持久化实例。

集群范围高速缓存需要某种远程过程通信（remote process communication）来维持一致性。高速缓存信息必须被复制到集群中的所有节点。对于许多（并非所有的）应用程序来说，集群范围高速缓存的价值并不太确定，因为读取和更新高速缓存可能只是在一定程度上比直接到数据库快一些。

持久层可以提供多级高速缓存。例如，事务范围中的高速缓存丢失（cache miss，在高速缓存中查找一个项目，但是它并没有被包含在该高速缓存中）可能接着在过程范围中查找。数据库请求是最后的办法。

持久层使用的高速缓存类型影响着对象同一性的范围（Java对象同一性和数据库同一性之间的关系）。

1. 高速缓存和对象同一性

考虑事务范围的高速缓存。这个高速缓存似乎很自然地也被用作对象的同一性范围。这意味着高速缓存实现同一性处理：对于使用相同数据库标识符的对象的两个查找返回了实际上相同的Java实例。因此，如果持久化机制也提供工作单元范围的对象同一性，事务范围高速缓存就是理想的。

利用过程范围高速缓存的持久化机制可能选择实现过程范围的同一性。在这种情况下，对象同一性就相当于整个过程的数据库同一性。在两个并发运行的工作单元中，使用相同的数据库标识符的两个查找产生了相同的Java实例。另一种方法是，从过程范围的高速缓存中获取的对象可能按值（by value）被返回。此时，每个工作单元获取它自己的状态副本（考虑原始数据），并且生成的持久化实例也不相同。高速缓存的范围和对象同一性的范围就不再相同。

集群范围高速缓存始终需要远程通信，并且在面向POJO的持久化解决方案（如Hibernate）的案例中，对象始终按值被远程传递。因此，集群范围高速缓存无法保证跨越集群的同一性。

对于典型的Web或者企业应用程序的架构来说，对象同一性的范围被限制为单个工作单元最方便。换句话说，在两个并发的线程中有相同的对象既不必要也不理想。在其他种类的应用程序中（包括有些桌面型的或者胖客户端的架构），使用过程范围的对象同一性可能很恰当。这在内

存极端受限的应用程序中尤其合适——工作单元范围高速缓存的内存消耗与并发线程的数量成正比。

然而，过程范围同一性的真正缺点在于，需要同步对高速缓存中持久化实例的访问，它极可能导致死锁，以及由于锁争用而减少了可伸缩性。

2．高速缓存和并发

任何允许多个工作单元共享相同持久化实例的ORM实现，都必须提供某些对象级锁定的形式，以确保并发访问的同步。这通常结合使用读/写锁（保存在内存中）与死锁侦测共同实现。像Hibernate这样的实现，给每一个工作单元（工作单元范围的同一性）维护一个独特的实例组，很大程度地避免了这些问题。

我们的观点是，应该避免在内存中保存锁，至少对于多用户可伸缩性高于一切的Web应用程序和企业应用程序应该是如此。在这些应用程序中，通常无需通过并发的工作单元比较对象同一性；每个用户都应该完全与其他的用户隔离开来。

当底层的关系数据库实现一个多版本的并发模型（如Oracle或者PostgreSQL）时，这种观点有一个特别明显的案例。给对象/关系持久化高速缓存重新定义事务语义或者底层数据库的并发模型，这有点不像我们想要的结果。

再次考虑选项。如果你也使用工作单元范围的对象同一性，并且如果对于高并发的多用户系统来说它是最佳策略，那么事务/工作单元范围的高速缓存就是首选。这个一级高速缓存是强制的，因为它也保证同一的对象。然而，这并非是你可以使用的唯一高速缓存。对于有些数据，二级高速缓存被界定到按值返回数据的过程（或者集群）中可能会有好处。因此这个场景有两个高速缓存层；稍后你会见到Hibernate使用这种方法。

来讨论哪种数据受益于二级高速缓存——换句话说，除了强制的一级事务范围的高速缓存之外，什么时候打开过程（或者集群）范围的二级高速缓存。

3．高速缓存和事务隔离性

过程或者集群范围高速缓存使得从一个工作单位中的数据库获取到的数据对另一个工作单元可见。这对事务隔离性可能有一些讨厌的负面影响。

首先，如果应用程序有对数据库的非专有访问，就不应该使用过程范围高速缓存，除了对很少改变且可以被高速缓存期限安全地刷新的数据之外。这种数据类型经常出现在内容管理类型的应用程序中，但很少在EIS或者财务应用程序中出现。

对于非专有的访问，要小心两个主要的场景：

- 被集群的应用程序；
- 共享的遗留数据。

任何被设计为可伸缩的应用程序，都必须支持集群的操作。过程范围高速缓存不维持集群中不同机器上不同高速缓存之间的一致性。此时，应该使用集群范围（分布式的）二级高速缓存而不是过程范围高速缓存。

许多Java应用程序都通过其他的应用程序共享对它们数据库的访问。既然如此，你就不应该使用任何超出工作单元范围一级高速缓存的高速缓存类型。高速缓存系统无法知道遗留应用程序

什么时候更新了共享的数据。实际上，虽然在改变数据库时有可能实现应用级的功能来触发过程（或者集群）范围高速缓存的失效，但我们不知道实现这一点的任何标准的或者最佳的方法。的确，它从来就不是Hibernate的内建特性。如果你实现这样一种解决方案，很可能是你自己努力的结果，因为它特定于所使用的环境和产品。

在考虑了非专有的数据访问之后，你应该创建应用程序数据需要的隔离级别了。并非每一个高速缓存实现都遵循所有的事务隔离级别，并且找出需要的级别很重要。我们来看一下从过程（或者集群）范围高速缓存中最受益的数据。在实践中，当我们进行这一评估时，发现依赖数据模型图（或者类图）很有帮助。在图上标注，说明一个特定的实体（或者类）对于二级高速缓存是好还是不好的备选对象。

完全的ORM解决方案让你单独给每个类配置二级高速缓存。高速缓存的好的备选类是表示以下数据的类：

- 很少改变的数据；
- 不重要的数据（如内容管理数据）；
- 应用程序固有的而非共享的数据。

对于二级高速缓存来说，不好的备选类是：

- 经常被更新的数据；
- 财务数据；
- 通过遗留应用程序共享的数据。

通常应用的规则不是唯一的。许多应用程序都有大量包含下列属性的类：

- 少量的实例；
- 被另一个类或者几个类的许多实例引用的每一个实例；
- 很少（或者从不）更新的实例。

这种数据有时称作*引用数据*（reference data）。引用数据的例子是邮政编码、参考地址、办公地点、静态文本消息等。引用数据对于利用过程或者集群范围的高速缓存而言是极好的备选对象，并且如果这个数据被高速缓存，任何大量使用引用数据的应用程序都将受益匪浅。你允许数据在高速缓存超时到期时刷新。

前几节通过工作单元范围的一级和可选的二级过程或者集群范围高速缓存，讲述了双层高速缓存系统。这很接近Hibernate的高速缓存系统了。

13.3.2 Hibernate 高速缓存架构

就像我们前面提示过的，Hibernate有一个两级的高速缓存架构。这个系统的各种元素如图13-7所示：

- 一级高速缓存是持久化上下文高速缓存。一个Hibernate `Session`的寿命相当于单个请求（通常用一个数据库事务实现）或者单个对话。这是个强制的一级高速缓存，它保证对象的范围和数据库同一性（例外的是`StatelessSession`，它没有持久化上下文）。

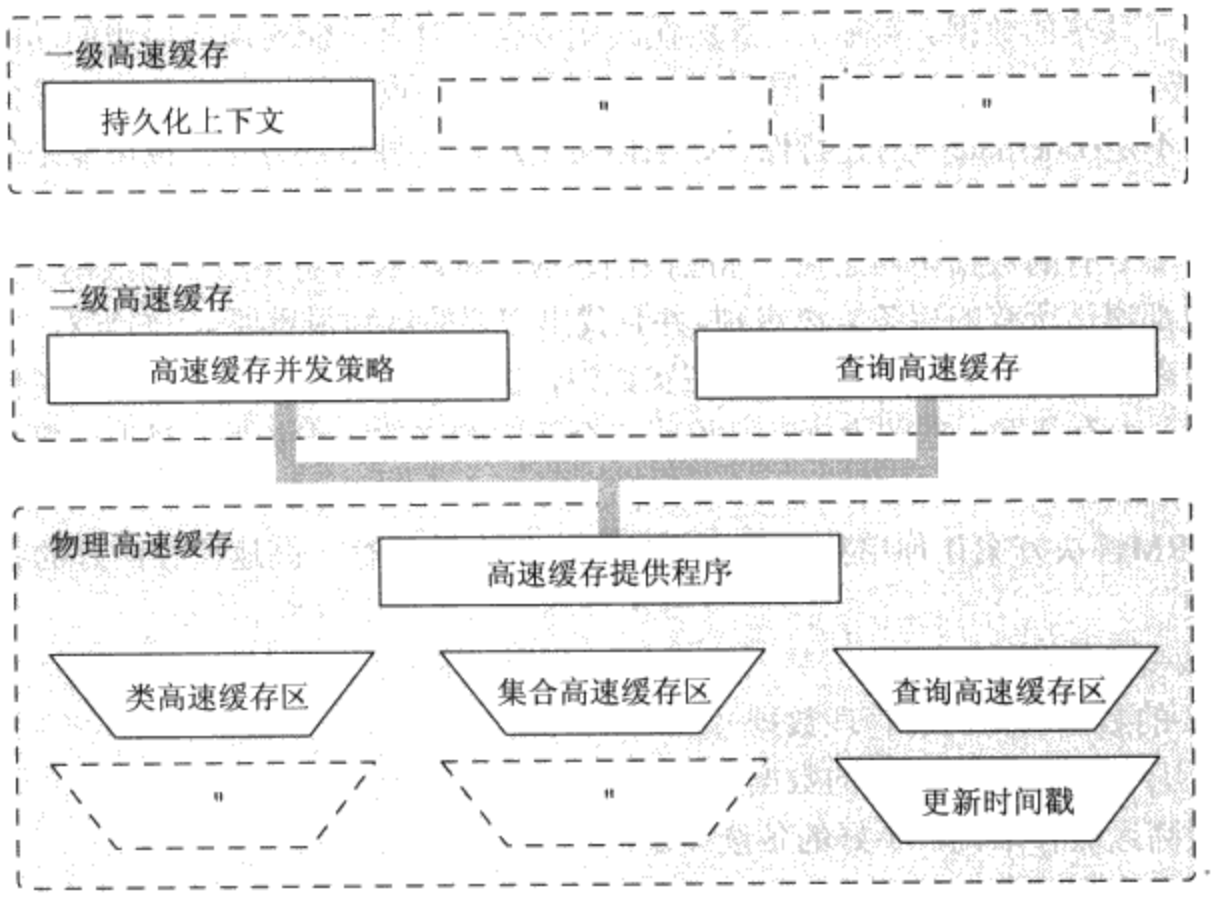


图13-7　Hibernate的二级高速缓存架构

- Hibernate中的二级高速缓存是可插拔的，并且可以被界定到过程或者集群。这是一个状态的高速缓存（按值返回），而不是实际的持久化实例的高速缓存。高速缓存并发策略给特定的数据项目定义了事务隔离细节，而高速缓存提供程序表示了物理高速缓存实现。二级高速缓存的使用是可选的，可以在按类和按集合的基础上配置——每一个这样的高速缓存都利用它自己的物理高速缓存区域。
- Hibernate也给与二级高速缓存密切整合的查询结果集实现高速缓存。这是个可选的特性；它需要两个额外的物理高速缓存区域，当表最后一次被更新时保存被高速缓存的查询结果和时间戳。我们在接下来的章节中讨论查询高速缓存，因为它的使用与正在执行的查询密切相关。

我们已经深入讨论过一级高速缓存、持久化上下文。现在直接进入可选的二级高速缓存。

1．Hibernate二级高速缓存

Hibernate的二级高速缓存有过程或者集群范围：已经从特定的`SessionFactory`开始的（或者与特定持久化单元的`EntityManager`关联的）所有持久化上下文共享同一个二级高速缓存。

持久化实例以分解的形式保存在二级高速缓存中。把分解当作一个过程，有点像序列化（但是这个算法比Java序列化要快很多）。

这个过程/集群范围高速缓存的内部实现没有什么值得关注的。更为重要的是高速缓存策略（cache policy）的正确使用——高速缓存策略和物理高速缓存提供程序。

不同种类的数据需要不同的高速缓存策略：读取与写入之比不同，数据库表的大小就不同，并且有些表与其他外部应用程序共享。二级高速缓存在一个单独的类或者集合角色的粒度中是可

配置的。例如，这样让你为引用数据类启用二级高速缓存，并对表示财务记录的类禁用它。这个高速缓存策略涉及如下设置：

- 二级高速缓存是否被启用；
- Hibernate并发策略；
- 高速缓存过期策略（如超时、LRU和内存敏感性）；
- 高速缓存的物理格式（内存、被索引的文件、集群复制）。

并非所有的类都受益于高速缓存，因此能够禁用二级高速缓存很重要。重申一下，高速缓存通常只对主要用来读取的类有用。如果你有更新比读取更经常的数据，就不要启用二级高速缓存，即使所有其他的高速缓存条件都符合！更新期间维护高速缓存的代价可能远远超出更快读取的性能受益。此外，二级高速缓存在与其他的写入应用程序共享数据库的系统中可能很危险。如前几节中所述，你在这里必须给想要启用高速缓存的每个类和集合进行小心的判断。

Hibernate二级高速缓存的创建分两步。首先，必须决定使用哪种*并发策略*（concurrency strategy）。之后，利用*高速缓存提供程序*配置高速缓存过期和物理高速缓存属性。

2. 内建的并发策略

并发策略是一个媒介：它负责在高速缓存中保存数据的项目，并从高速缓存中获取它们。这是个重要的角色，因为它也为特定的项目定义了事务隔离语义。如果想要启用二级高速缓存，就必须给每个持久化类和集合决定使用哪种并发策略。

下面4个内建的并发策略表示递减的事务隔离方面的严格级别：

- 事务（transactional）——只可用于托管环境，如有必要，它还保证完全的事务隔离直到*可重复读取*（repeatable read）。给主要用于读取的数据使用这种策略，因为在这种数据中，防止并发事务中的废弃数据最为关键，极少数情况下用于更新。
- 读/写（read-write）——这种策略利用时间戳机制，维护*读取提交*（read committed）隔离，并且只在非集群环境中可用。还是给主要用于读取的数据使用这种策略，因为在这种数据中防止并发事务中的废弃数据最为关键，极少数情况下用于更新。
- 非严格读/写（nonstrict-read-write）——不提供高速缓存和数据库之间的一致性保证。如果有可能并发访问相同的实体，你应该配置一个足够短的超时期限。否则，则可能从高速缓存中读取废弃的数据。如果数据几乎不变（几个小时、几天或者甚至一周），并且废弃的数据不可能是关键的关注点，那就使用这种策略。
- 只读（read-only）——并发策略适合于从不改变的数据。它只用于引用数据。

注意，递减的严格性带来了性能的递增。在把它用在产品中之前，必须小心评估包含完整事务隔离的集群高速缓存的性能。在许多情况下，如果绝不考虑废弃的数据，可能最好给特定的类禁用二级高速缓存！首先以禁用了二级高速缓存的应用程序为基准。为好的备选类启用它，一次一个，同时继续测试系统的可伸缩性，并评估并发策略。

通过实现`org.hibernate.cache.CacheConcurrencyStrategy`，有可能定义自己的并发策略，但这是个相对困难的任务，并且只适合于少数的优化案例。

考虑了要给高速缓存的备选类使用的并发策略之后，下一步就是挑选*高速缓存提供程序*。这

个提供程序是一个插件，是高速缓存系统的物理实现。

3．选择高速缓存提供程序

现在，Hibernate强制你给整个应用程序选择单个的高速缓存提供程序。Hibernate内建了以下开源产品的提供程序：

- EHCache是特意用于单个JVM中简单的过程范围高速缓存的高速缓存提供程序。它可以高速缓存在内存或者磁盘中，并支持可选的Hibernate查询结果高速缓存。（EHCache的最新版本现在支持被集群的高速缓存，但我们还没有对它进行过测试。）
- OpenSymphony OSCache是一项服务，它通过一组丰富的过期策略和查询高速缓存支持，支持在单个JVM中高速缓存到内存和磁盘。
- SwarmCache是基于JGroups的集群高速缓存。它使用集群的失效，但不支持Hibernate查询高速缓存。
- JBoss Cache是一个完全事务复制的集群高速缓存，也基于JGroups多播库（multicast library）。它支持复制或者失效、同步或者不同步的通信，以及乐观锁和悲观锁。支持Hibernate查询高速缓存，假设时钟在集群中同步。

通过实现`org.hibernate.cache.CacheProvider`给其他产品编写一个适配器很容易。许多商业高速缓存系统都可以通过这个接口插入到Hibernate。

并非每个高速缓存提供程序都可以与每一种并发策略兼容！表13-1中的兼容性矩阵将帮助你选择一种适当的组合。

表13-1　高速缓存并发策略支持

并发策略高速缓存提供程序	只　读	非严格读/写	读/写	事　务
EHCache	×	×	×	
OSCache	×	×	×	
SwarmCache	×	×		
JBoss Cache	×			×

创建高速缓存涉及两个步骤：首先，在映射元数据中看看持久化类和集合，并决定要给每个类和每个集合使用哪种高速缓存并发策略。然后，在全局的Hibernate配置中启用首选的高速缓存提供程序，并定制提供程序特定的设置和物理高速缓存区域。例如，如果正在使用OSCache，就编辑oscache.properties，或者给EHCache编辑类路径中的ehcache.xml。

来给CaveatEmptor的`Category`、`Item`和`Bid`类启用高速缓存。

13.4　高速缓存实践

首先，考虑每个实体类和集合，并找出哪种高速缓存并发策略可能适合。在给本地的和集群的高速缓存选择好高速缓存提供程序之后，将编写它们的（几个）配置文件。

13.4.1　选择并发控制策略

`Category`只有少量的实例并且很少更新，而且实例在许多用户之间共享。它是供二级高速缓

存使用的一个很好的备选类。

从添加Hibernate高速缓存Category实例所需的映射元素开始：

```
<class name="auction.model.Category"
       table="CATEGORY">

    <cache usage="read-write"/>

    <id ...

</class>
```

usage="read-write"属性告诉Hibernate，给auction.model.Category高速缓存使用一个读/写并发策略。每当导航到Category，或者当按标识符加载Category时，Hibernate现在就命中二级高速缓存。

如果使用注解，就需要Hibernate扩展：

```
@Entity
@Table(name = "CATEGORY")
@org.hibernate.annotations.Cache(usage =
    org.hibernate.annotations.CacheConcurrencyStrategy.READ_WRITE
)
public class Category { ... }
```

你用read-wirte代替了nonstrict-read-write，因为Category是一个高并发的类，在许多并发事务中共享。（很显然，读取提交隔离级别就足够好了。）nonstrict-read-write只依赖高速缓存过期（超时），但你更喜欢让对类别所做的改变立即可见。

类高速缓存始终对持久化类的整个层次结构而被启用。你无法只高速缓存一个特定子类的实例。

这个映射足以告诉Hibernate去高速缓存所有简单的Category属性值，而不是被关联的实体或者集合的状态。集合需要它们自己的<cache>区域。给items集合使用read-write并发策略：

```
<class name="auction.model.Category"
       table="CATEGORY">

    <cache usage="read-write"/>

    <id ...

    <set name="items">
        <cache usage="read-write"/>
        <key ...
    </set>

</class>
```

集合高速缓存的区域名称是完全匹配的类名加上集合属性的名称：auction.model.Category.items。@org.hibernate.annotations.Cache注解也可以在集合字段或者获取方法中进行声明。

调用aCategory.getItems()时，这个高速缓存设置生效——换句话说，集合高速缓存是一个包含“哪些项目处在哪些类别中”的区域。它是只有标识符的高速缓存，在该区域中没有实际的Category或者Item数据。

如果你需要Item实例本身被高速缓存，就必须启用Item类的高速缓存。读/写（read-write）策略尤其适合。用户可不想根据可能失效的Item数据做出决定（例如，发出一个出价）。更进一步，考虑bids的集合：bids集合中的特定Bid是不可变的，但是bids的集合是可变的，并且并发的工作单元需要毫不延迟地看到集合元素的添加或者移除：

```
<class name="Item"
       table="ITEM">

    <cache usage="read-write"/>

    <id ...

    <set name="bids">
        <cache usage="read-write"/>
        <key ...
    </set>

</class>
```

给Bid类应用只读（read-only）策略：

```
<class name="Bid"
       table="BID" mutable="false">

    <cache usage="read-only"/>

    <id ...

</class>
```

因此，Bid数据在高速缓存中从来不会过期，因为它只能被创建而从不更新。（出价当然可能被高速缓存提供程序终止——例如，如果达到了高速缓存中对象的最多数量时。）如果Bid实例被删除，Hibernate也从高速缓存中移除数据，但是不对这种做法提供任何事务的保证。

User是可以通过非严格读/写（nonstrict-read-wirte）策略而被高速缓存的类的一个示例，但我们不确定它对于高速缓存用户是否有意义。

下面来设置高速缓存提供程序，设置它的过期策略和高速缓存的物理区域。使用高速缓存区域单独地配置类和集合高速缓存。

13.4.2　理解高速缓存区域

Hibernate在不同的高速缓存区域（cache region）中保存不同的类/集合。区域是一个具名的高速缓存：这个句柄使你可以通过它在高速缓存提供程序配置中引用类和集合，并设置适用于该区域的过期策略。一种更为图形化的描述是，区域是一桶桶的数据，它们有两种类型：一种区域类型包含实体实例的分解数据，另一种类型只包含通过集合而被链接的实体标识符。

对于类高速缓存而言，区域的名称是类名；对于集合高速缓存而言，是类名加属性名。Category实例被高速缓存在具名auction.model.Category的区域中，而items集合则被高速缓存在具名auction.model.Category.items的区域中。

Hibernate具名hibernate.cache.region_prefix的配置属性可能被用来给特定的SessionFactory或者持久化单元指定区域名前缀。例如，如果前缀设置为dbl，Category就被

高速缓存在具名dbl.auction.model.Category的区域中。如果应用程序使用多个SessionFactory实例或者持久化单元，这项设置就是必需的。没有它，不同持久化单元的高速缓存区域名称就可能冲突。

既然了解了高速缓存区域，就可以配置auction.model.Category高速缓存的物理属性了。首先来选择一种高速缓存提供程序。假设你正在单个JVM中运行拍卖应用程序，那么你不需要集群感知的提供程序。

13.4.3 设置本地的高速缓存提供程序

需要设置选择了高速缓存提供程序的配置属性：

```
hibernate.cache.provider_class = org.hibernate.cache.EhCacheProvider
```

在这个例子中选择EHCache作为二级高速缓存。

现在，你需要指定高速缓存区域的属性。EHCache有自己的配置文件ehcache.xml，在应用程序的类路径中。Hibernate发行包中为所有绑定的高速缓存提供程序捆绑了示例配置文件，因此我们建议你阅读那些文件中的使用注释，看看详细的配置，并给我们没有明确提到的所有选项假设默认值。

echache.xml中用于Category类的高速缓存配置可能看起来像这样：

```
<cache name="auction.model.Category"
       maxElementsInMemory="500"
       eternal="true"
       timeToIdleSeconds="0"
       timeToLiveSeconds="0"
       overflowToDisk="false"
/>
```

有少量的Category实例。因此，你通过选择一个大于系统中类别数量的高速缓存大小限制、并设置eternal="true"、通过超时禁用清除。没有必要通过超时终止被高速缓存的数据，因为Category高速缓存并发策略是读/写，并且因为没有其他的应用程序直接在数据库中改变类别数据。你还禁用了基于磁盘的高速缓存溢出，因为你知道Category的实例很少，因此内存消耗不会有问题。

另一方面，Bid很小并且不可变，但是有很多，因此你必须配置EHCache来小心管理高速缓存内存消耗。你把过期超时和最大高速缓存大小限制都用上了：

```
<cache name="auction.model.Bid"
       maxElementsInMemory="50000"
       eternal="false"
       timeToIdleSeconds="1800"
       timeToLiveSeconds="100000"
       overflowToDisk="false"
/>
```

timeToIdleSeconds属性定义了以秒为单位的从一个元素最后一次在高速缓存中被访问开始的过期时间。你必须在这里设置一个有意义的值，因为你不想让没有用到的出价消耗内存。

timeToLiveSeconds属性定义了以秒为单位的从元素被添加到高速缓存中开始的最大过期时间。由于出价是不可变的，因此如果它们正被正常访问，你就不需要把它们从高速缓存中移除。因而，timeToLiveSeconds被设置成一个很大的数值。

结果，当高速缓存总大小达到最大限制50 000个元素时，那么在最近的30分钟内没有被使用或者最近使用频度最小的被高速缓存的出价就会从高速缓存中被移除。

在这个示例中，你禁用了基于磁盘的高速缓存，因为你预计应用程序服务器会被部署为与数据库相同的机器。如果所期待的物理架构不同，则可能启用基于磁盘的高速缓存来减少网络通信量。在本地磁盘上访问数据比跨过网络访问数据库更快。

如你所见，最佳的高速缓存清除策略是特定于数据和应用程序。必须考虑许多外部的因素，包括应用程序服务器机器中可用的内存、希望在数据库机器上的加载、网络延迟、遗留应用程序的存在，等等。这些因素中有一些不可能在开发时得知，因此经常需要在产品环境中或者模拟的环境中反复调试不同设置的性能影响。我们考虑通过二级高速缓存优化一些在开发期间你不会去做的事情，因为在没有实际的数据集和并发的情况下的测试，并不能代表系统的最终性能和可伸缩性。这在集群机器中带有复制的高速缓存的更复杂场景中尤其如此。

13.4.4　设置重复的高速缓存

如果应用程序部署在单个虚拟机上，EHCache就是一个极好的高速缓存提供程序。然而，支持上千个并发用户的企业应用程序则可能需要更大的计算能力，并且可伸缩的应用程序对于项目的成功可能十分关键。Hibernate应用程序天生就是可伸缩的：Hibernate没有任何方面限制应用程序要部署在哪些节点上。通过稍微改变高速缓存配置，甚至就可能使用一个集群的高速缓存系统了。

我们推荐JBoss Cache，一个基于TreeCache和JGroups多播库的集群安全的高速缓存系统。JBoss Cache是非常可伸缩的，并且集群通信可以用任何可能的方式进行调优。

现在进入对CaveatEmptor的JBoss Cache配置，一个两节点（称作节点A和节点B）的小集群。然而，我们只对这个话题做肤浅的探讨，集群配置天生复杂，并且许多设置取决于特定的场景。

首先，必须检查所有使用只读或者事务作为高速缓存并发策略的所有映射文件。这些是JBoss Cache提供程序唯一支持的策略。有一种很好的技巧有助于避免这种搜索，并在未来替换问题。不要在映射文件中放置<cache>元素，而是集中在hibernate.cfg.xml中的高速缓存配置：

```
<hibernate-configuration>
<session-factory>

    <property .../>
    <mapping .../>

    <class-cache
            class="org.hibernate.auction.model.Item"
            usage="transactional"/>

    <collection-cache
            collection="org.hibernate.auction.model.Item.bids"
            usage="transactional"/>
```

```
</session-factory>

</hibernate-configuration>
```

在这个例子中，你给Item和bids集合启用了事务高速缓存。然而，有一个重要的警告：在编写本书之时，如果你在Item的映射文件中也有<cache>元素，Hibernate就遇到了冲突。因此，你不能使用全局的配置覆盖映射文件设置。建议从一开始就使用集中的高速缓存配置，尤其当你不确定应用程序会被如何部署的时候。通过单个配置位置调优高速缓存设置也更加容易。

集群配置的下一步是JBoss Cache提供程序的配置。首先，在Hibernate配置中启用它——例如，如果你没有在hibernate.cfg.xml中使用属性：

```
<property name="cache.provider_class">
    org.hibernate.cache.TreeCacheProvider
</property>
```

JBoss Cache有它自己的配置文件treecache.xml，在应用程序的类路径中。在有些场景中，集群中的每个节点都需要不同的配置，并且你必须确保部署时把正确的文件复制到类路径中去。来看一个典型的配置文件。在一个两节点的集群（具名MyCluster）中，这个文件用在节点A上：

```
<server>

  <classpath codebase="./lib"
             archives="jboss-cache.jar, jgroups.jar"/>

  <mbean code="org.jboss.cache.TreeCache"
      name="jboss.cache:service=TreeCache">

<depends>jboss:service=Naming</depends>
<depends>jboss:service=TransactionManager</depends>

<attribute name="TransactionManagerLookupClass">
    org.jboss.cache.GenericTransactionManagerLookup
</attribute>

<attribute name="ClusterName">MyCluster</attribute>

<attribute name="NodeLockingScheme">PESSIMISTIC</attribute>
<attribute name="CacheMode">REPL_SYNC</attribute>
<attribute name="IsolationLevel">REPEATABLE_READ</attribute>

<attribute name="FetchInMemoryState">false</attribute>
<attribute name="InitialStateRetrievalTimeout">20000</attribute>
<attribute name="SyncReplTimeout">20000</attribute>
<attribute name="LockAcquisitionTimeout">15000</attribute>

<attribute name="ClusterConfig">
    <config>
      <UDP loopback="false"/>

      <PING timeout="2000"
            num_initial_members="3"
            up_thread="false"
            down_thread="false"/>

      <FD_SOCK/>

      <pbcast.NAKACK gc_lag="50"
```

```
                        retransmit_timeout="600,1200,2400,4800"
                        max_xmit_size="8192"
                        up_thread="false" down_thread="false"/>
                <UNICAST timeout="600,1200,2400"
                         window_size="100"
                         min_threshold="10"
                         down_thread="false"/>
                <pbcast.STABLE desired_avg_gossip="20000"
                               up_thread="false"
                               down_thread="false"/>
                <FRAG frag_size="8192"
                      down_thread="false"
                      up_thread="false"/>
                <pbcast.GMS join_timeout="5000"
                            join_retry_timeout="2000"
                            shun="true" print_local_addr="true"/>
                <pbcast.STATE_TRANSFER up_thread="true"
                                       down_thread="true"/>
            </config>
        </attribute>
    </mbean>
</server>
```

当然，这个配置文件乍看可能有点吓人，但它很容易理解。你必须知道，它不仅仅是JBoss Cache的一个配置文件，它还集多种角色于一身：JBoss AS部署的JMX服务配置、TreeCache的配置文件、JGroups的细粒度配置和通信库。

忽略与JBoss部署相关的前面几行，看看第一个属性`TransactionManagerLookupClass`。`GenericTransactionManagerLoopup`试图在最流行的应用程序服务器中找到事务管理器，但它也适用于没有JTA的独立环境（没有事务管理器的集群高速缓存很少见）。如果JBoss Cache在启动时抛出异常，告诉你它无法找到事务管理器，你就必须自己给JPA提供程序/应用程序服务器创建这样一个查找类。

接下来，是使用同步通信的被复制高速缓存的配置属性。这意味着发送同步消息的节点等待着，直到群组中的所有节点都获得这条消息。对于真正的被复制高速缓存来说，这是一个很好的选择；如果节点B是一个热备份（hot standby，如果节点A失败，立即取而代之的一个节点）而不是一个积极的搭档的话，异步不阻塞的通信将更合适。这是一个故障转移（failover）与计算能力的问题，两者都是配置集群的好理由。大多数的配置属性都应该一目了然，比如节点进入集群时的超时和状态抓取。

JBoss Cache也可以清除元素，防止内存耗尽。在这个例子中，你不配置清除策略，因此高速缓存慢慢开始填充所有可用的内存。对于清除策略配置，你必须参考JBoss Cache文档；Hibernate区域名称的用法和清除设置类似于EHCache。

JBoss Cache也支持失效，但不支持集群中被修改数据的复制，这是一种可能更好的执行选择。然而，Hibernate查询高速缓存需要复制。你也可以转换到OPTIMISTIC锁而不是悲观锁，再次加

强集群高速缓存的可伸缩性。这么做需要另一种Hibernate高速缓存提供程序插件org.hibernate.cache.OptimisticTreeCacheProvider。

最后，看看JGroups集群通信配置。通信协议的顺序极为重要，因此不要随便改变或者增加行。最值得关注的是第一个协议<UDP>。如果节点A是一台Microsoft Windows机器（在这个例子中不是），loopback属性就必须设置为true。

其他的JGroups属性更复杂，可以在JGroups文档中找到。它们处理发现算法（discovery algorithm），用于在群组中侦测新节点、失败侦测，以及一般来说还有群组通信的管理。

在改变持久化类的高速缓存并发策略为事务（或者只读），并为节点A创建了treecache.xml文件之后，就可以启动应用程序并观察日志输出了。建议给org.jboss.cache包启用DEBUG日志；然后你就会看到JBoss Cache如何读取配置，并报告节点A为集群中的第一个节点。为了部署节点B，你要在该节点上部署应用程序；不需要改变配置文件（如果第二个节点也不是Microsoft Windows机器）。如果启动这第二个节点，你应该在这两个节点上都看到联结消息。现在Hibernate应用程序在集群中使用完全事务的高速缓存。

还有最后一项可选的设置要考虑。对于集群高速缓存提供程序而言，把Hibernate配置选项hibernate.cache.use_minimal_puts设置为true可能更好。当启用这项设置时，Hibernate只在检查以确保该项目还没有被高速缓存之后才把它添加到高速缓存中。如果高速缓存写入（存）比高速缓存读取（取）更贵，则是这种策略执行得更好。这是针对集群中被复制高速缓存的案例，而不是针对本地高速缓存或者依赖失效（而非复制）的高速缓存提供程序。

无论使用集群还是本地高速缓存，你有时候都需要通过编程来控制它，不论为了测试，还是为了调优。

13.4.5 控制二级高速缓存

Hibernate有一些有用的方法，可以帮助你测试和调优高速缓存。给二级高速缓存hibernate.cache.use_second_level_cache考虑全局的配置转换。默认情况下，映射文件（在hibernate.cfg.xml或者注解）中的任何<cache>元素都触发二级高速缓存，并在启动时加载高速缓存提供程序。如果想要全局地禁用二级高速缓存，而不移除高速缓存映射元素或者注解，那么就设置这个配置属性为false。

就像Session和EntityManager提供通过编程来控制持久化上下文一级高速缓存的方法一样，SessionFactory对二级高速缓存也一样。在JPA应用程序中，必须访问底层的内部SessionFactory，如2.2.4节中所述。

可以调用evict()，通过指定类和对象标识符值，从二级高速缓存中移除元素：

```
SessionFactory.evict( Category.class, new Long(123) );
```

也可以通过指定一个区域名称，来清除一个特定类的所有元素，或者清除一个特定集合角色：

```
SessionFactory.evict("auction.model.Category");
```

你将很少需要用到这些控制机制。还要注意二级高速缓存的清除是非事务的，也就是说，高速缓存区域在清除期间没有被锁定。

Hibernate还提供CacheMode选项，它可以由特定的Session启用。想象你想要在Session中把许多个对象插入到数据库。需要以批量来完成这项工作，以避免内存耗尽——每一个对象都被添加到一级高速缓存中。然而，如果为实体类启用了它，它也被添加到二级高速缓存中。CacheMode控制Hibernate与二级高速缓存的交互：

```
Session session = sessionFactory.openSession();
Transaction tx = session.beginTransaction();

session.setCacheMode(CacheMode.IGNORE);

for ( int i=0; i<100000; i++ ) {
    Item item = new Item(...);
    session.save(item);
    if ( i % 100 == 0 ) {
        session.flush();
        session.clear();
    }
}

tx.commit();
session.close();
```

设置CacheMode.IGNORE告诉Hibernate不要在这个特定的Session中与二级高速缓存交互。可用的选项如下：

- CacheMode.NORMAL——默认的行为。
- CacheMode.IGNORE——Hibernate从来不与二级高速缓存交互，除了更新发生时使被高速缓存的项目失效之外。
- CacheMode.GET——Hibernate可能从二级高速缓存中读取项目，但不添加项目，除了更新发生时使项目失效之外。
- CacheMode.PUT——Hibernate从来不从二级高速缓存中读取项目，但是当它从数据库中读取项目时，会把项目添加到高速缓存。
- CacheMode.REFRESH——Hibernate从来不从二级高速缓存中读取项目，但是当它从数据库中读取项目时，会把项目添加到高速缓存。在这种模式下，hibernate.cache.use_minimal_puts的作用被忽略，以便在一个复制的集群高速缓存中强制高速缓存刷新。

除了NORMAL和IGNORE之外，好的用例很少。

这样就结束了对Hibernate应用程序中一级和二级高速缓存的讨论。我们想要重申一项在本节一开始时提到过的声明：你的应用程序不用二级高速缓存也应该执行得令人满意。如果你通过启用二级高速缓存，使应用程序中的一个特定过程在2秒而不是50秒内运行，你就只是治了标，而没有治本。抓取计划和抓取策略的定制始终是你的第一个优化步骤；然后，通过二级高速缓存使应用程序更迅速，并把它扩展到它将必须在产品中处理的并发事务加载。

13.5 小结

本章创建了一个全局的抓取计划，并定义了哪些对象和集合应该始终被加载到内存中。你定义了基于用例的抓取计划，以及想要如何访问被关联的实体，以及如何在应用程序中遍历集合。

接下来，你给抓取计划选择了正确的抓取策略。目标是把SQL语句的数量和必须执行的每个SQL语句的复杂性减到最小。你尤其想要通过各种优化策略，来避免我们详细阐述过的n+1查询和笛卡儿积问题。

本章后半部分通过高速缓存背后的理论介绍了高速缓存，并提供了一张清单，可以用它来找出哪些类和集合对于Hibernate的可选二级高速缓存是好的备选对象。然后，通过本地的EHCache提供程序和集群启用的JBoss Cache，给一些类和集合配置和启用了二级高速缓存。

表13-2展现了可以用来比较原生的Hibernate特性和Java Persistence的概括。

表13-2 第13章中Hibernate和JPA的对照表

Hibernate Core	Java Persistence和EJB 3.0
Hibernate支持通过代理或者基于拦截的延迟加载定义抓取计划	Hibernate通过代理或者基于拦截的延迟加载实现Java Persistence提供程序
Hibernate允许细粒度地控制抓取计划和抓取策略	Java Persistence给抓取计划声明标准化了注解，Hibernate扩展用于细粒度的抓取策略优化
Hibernate提供一个可选的二级类和集合数据高速缓存（配置在Hibernate配置文件或者XML映射文件中）	在实体和集合中，给高速缓存并发策略的声明使用Hibernate注解

接下来的章节专门处理查询，并介绍如何利用所有的Hibernate和Java Persistence接口来编写和执行HQL、JPA QL、SQL和Criteria查询。

第 14 章

利用HQL和JPA QL查询

本章内容

- 理解各种查询选项
- 编写HQL和JPA QL查询
- 联结、报表查询、子查询

查询是编写好的数据访问代码时最值得关注的部分。复杂的查询可能需要花很长的时间才能弄正确，因此它对应用程序的性能影响很巨大。另一方面，有了更多经验以后，编写查询就会变得更加容易，当初似乎很难的东西，只要了解了一些更高级特性，它们就不难解决了。

如果你已经使用手写的SQL多年，则可能关注ORM会拿掉一些你经常使用的表示法和灵活性。使用Hibernate和Java Persistence就没有这种问题。

Hibernate强大的查询工具允许你表达一般（或者甚至罕见地）在SQL中需要表达的几乎全部，但是从面向对象的角度——使用类和类的属性。

我们将介绍原生的Hibernate查询和Java Persistence中标准子集之间的区别。也可以把本章作为一个参考；因此有些小节写得不太详细，但是介绍了许多用于不同用例的小型代码示例。为了便于阅读，我们有时也在CaveatEmptor应用程序中跳过优化。例如，不引用`MonetaryAmount`值类型，而是在比较式中使用`BigDecimal`金额。

首先，介绍查询如何执行。别让自己被查询搞得心烦意乱；我们很快就会讨论到它们。

14.1 创建和运行查询

让我们从一些示例开始，以便你理解基础的用法。前面的章节提到过在Hibernate中有3种表达查询的方法：

- Hibernate查询语言（HQL），以及被标准化为JPA QL的子集：

```
session.createQuery("from Category c where c.name like 'Laptop%'");
entityManager.createQuery(
    "select c from Category c where c.name like 'Laptop%'"
);
```

- 用于按条件查询（QBC）和按示例查询（QBE）的`Criteria` API：

```
session.createCriteria(Category.class)
        .add( Restrictions.like("name", "Laptop%") );
```

- 直接的SQL（无论是否将结果集自动映射到对象）：

```
session.createSQLQuery(
    "select {c.*} from CATEGORY {c} where NAME like 'Laptop%'"
).addEntity("c", Category.class);
```

在执行之前，必须在应用程序代码中准备一个查询。因此，查询涉及几个独特的步骤：

(1) 创建查询，通过任何任意的限制或者你想要获取的数据的投影。

(2) 把运行时实参绑定到查询参数，该查询可以通过改变设置被重用。

(3) 执行针对数据库的预编译查询和数据获取。你可以控制查询如何执行，以及应该如何把数据获取到内存中（比如一次获取全部，还是逐个获取）。

Hibernate和Java Persistence在这些接口中提供查询接口和方法，以准备和执行任意的数据获取操作。

14.1.1 准备查询

org.hibernate.Query和org.hibernate.Criteria这两个接口都定义几种控制查询执行的方法。此外，Query还提供把具体值绑定到查询参数的方法。为了在应用程序中执行查询，需要利用Session获得其中一个接口的实例。

Java Persistence指定javax.persistence.Query接口。标准的接口不像原生的Hibernate API那么丰富，但是提供了以不同方式执行查询并把实参绑定到查询参数的所有必需的方法。不幸的是，有用的Hibernate Criteria API在Java Persistence中并没有等价物，虽然很可能在标准的未来版本中添加类似的查询接口。

1. 创建查询对象

为了创建新的Hibernate Query实例，在Session上调用createQuery()或者createSQLQuery()。createQuery()方法来准备一个HQL查询：

```
Query hqlQuery = session.createQuery("from User");
```

通过底层数据库的原生语法，可以用createSQLQuery()来创建SQL查询：

```
Query sqlQuery =
        session.createSQLQuery(
            "select {user.*} from USERS {user}"
        ).addEntity("user", User.class);
```

在这两个案例中，Hibernate都返回一个刚刚被实例化的Query对象，它可以用来指定一个特定的查询应该被如何执行，并允许执行查询。目前为止，还没有SQL被发送到数据库。

要想获得Criteria实例，就调用createCriteria()，传递你想让查询返回的对象的类。这也称作条件查询的根实体（root entity），如这个示例中的User：

```
Criteria crit = session.createCriteria(User.class);
```

Criteria实例的使用方法与Query对象一样——但它也用来构建查询的面向对象的表示法，通过添加Criterion实例并导航关联到新的Criteria。

利用JPA，你的查询起点是EntityManager。为了给JPA QL创建javax.persistence.Query实例，就调用createQuery()：

```
Query ejbQuery = em.createQuery("select u from User u");
```

为了创建原生的SQL查询，就用createNativeQuery()：

```
Query sqlQuery =
        em.createNativeQuery(
         "select u.USER_ID, u.FIRSTNAME, u.LASTNAME from USERS u",
         User.class
        );
```

你定义从原生查询返回对象的方法与Hibernate中的略有不同（这里的查询中没有占位符）。创建查询之后，要设置各种选项准备执行它。

2. 给结果进行分页

常用的方法是分页（pagination）。用户可以看到他们搜索请求（例如搜索特定的Item）的结果页面。这个页面一次显示有限的子集（假设10个Item），用户可以手工导航到后一个和前一个页面。在Hibernate中，Query和Criteria接口支持查询结果的这种分页：

```
Query query =
        session.createQuery("from User u order by u.name asc");
query.setMaxResults(10);
```

对setMaxResults(10)的调用把查询结果集限制为由数据库返回的前10个对象（行）。在这个Criteria查询中，被请求的页面从结果集的中间开始：

```
Criteria crit = session.createCriteria(User.class);
crit.addOrder( Order.asc("name") );
crit.setFirstResult(40);
crit.setMaxResults(20);
```

从第40个对象开始，获取接下来的20个对象。注意，在SQL中表达分页并没有标准的方法——Hibernate知道这些技巧，使得它能在特定的数据库中有效地工作。甚至可以把这个灵活的分页选项添加到一个SQL查询。Hibernate将给分页重写SQL：

```
Query sqlQuery =
        session.createSQLQuery("select {u.*} from USERS {u}")
                .addEntity("u", User.class);
sqlQuery.setFirstResult(40);
sqlQuery.setMaxResults(20);
```

可以通过Query和Criteria接口，使用方法链的编码风格（方法返回正接收的对象而不是void），重写前两个示例如下：

```
Query query =
        session.createQuery("from User u order by u.name asc")
                .setMaxResults(10);
Criteria crit =
        session.createCriteria(User.class)
                .addOrder( Order.asc("name") )
                .setFirstResult(40)
                .setMaxResults(20);
```

链方法调用没那么详细，许多Hibernate API都支持它。Java Persistence查询接口也通过javax.persistence.Query接口，对JPA QL和原生的SQL查询支持分页和方法链：

```
Query query =
        em.createQuery("select u from User u order by u.name asc")
          .setFirstResult(40)
          .setMaxResults(20);
```

准备查询的下一步是设置任何运行时参数。

3. 考虑参数绑定

如果没有运行时参数绑定，你只能编写很糟糕的代码：

```
String queryString =
        "from Item i where i.description like '" + search + "'";
List result = session.createQuery(queryString).list();
```

你应该永远不要编写这样的代码，因为恶意的用户可能搜索下面的项目描述——也就是说，在搜索对话框中输入search的值为：

```
foo' and callSomeStoredProcedure() and 'bar' = 'bar
```

可见，原始的queryString不再是对字符串的一个简单查询，而且还在数据库中执行一个存储过程！引号没有进行转码，因此在查询中对存储过程的调用是另一个有效的表达式。如果你编写了一个像这样的查询，允许在你的数据库中执行任意的代码，就是在应用程序中开了一个很大的安全漏洞。这就是众所周知的SQL注入（SQL injection）安全性问题。永远不要将用户输入的未被检查值传递到数据库！幸而，有一种简单的机制可以防止这种错误。

JDBC驱动包括将值安全绑定到SQL参数的功能。它准确地知道参数值中的哪些字符要转码，因此不存在前面提到的安全性问题。例如，对给定的search中的引号字符进行转码，不再作为控制符号处理，而是作为搜索字符串值的一部分。此外，使用参数的时候，数据库能够有效地高速缓存预编译的语句，显著地改善性能。

参数绑定（parameter binding）有两种方法：利用定位或者利用具名参数。Hibernate和Java Persistence都支持这两个选项，但是你无法给一个特定的查询同时使用这两个选项。

利用具名参数，可以重写这个查询为：

```
String queryString =
    "from Item item where item.description like :search";
```

参数名称后面的冒号表示这是一个具名参数。然后，对search参数绑定一个值：

```
Query q = session.createQuery(queryString)
                 .setString("search", searchString);
```

由于searchString是一个用户提供的字符串变量，你调用Query接口的setString()方法，把它绑定到具名参数（:search）。这个代码更规则、更安全且执行得更好，因为如果只是绑定参数的变化，单独编译过的SQL语句就可以被重用。

你将经常需要多个参数：

```
String queryString = "from Item item"
                   + " where item.description like :search"
                   + " and item.date > :minDate";
```

```
Query q = session.createQuery(queryString)
                 .setString("search", searchString)
                 .setDate("minDate", mDate);
```

同样的查询和代码，在Java Persistence中看起来稍有不同：

```
Query q = em.createQuery(queryString)
            .setParameter("search", searchString)
            .setParameter("minDate", mDate, TemporalType.DATE);
```

setParameter()方法是个一般的操作，它可以绑定实参的所有类型，对于临时的类型只需要一点点帮助(引擎需要知道你是只要日期、时间，还是需要完整的时间戳绑定)。Java Persistence只支持这种方法，用于参数的绑定（顺便说一下，Hibernate也有这种方法)。

另一方面，Hibernate还提供许多其他的方法，其中一部分是为了完整，其他的是为了方便，你可以用它们来把实参绑定到查询参数。

4．利用Hibernate的参数绑定

你已经调用了setString()和setDate()来把实参绑定到查询参数。原生的Hibernate Query接口提供类似的便利方法，用来绑定大多数Hibernate内建类型的参数：从setInteger()到setTimestamp()和setLocale()的一切。它们中大多数是可选的；可以依赖setParameter()方法自动找出正确的类型（除了临时的类型之外)。

特别有用的一种方法是setEntity()，它让你绑定一个持久化实体(注意，setParameter()很聪明，足以自动理解这一点)：

```
session.createQuery("from Item item where item.seller = :seller")
       .setEntity("seller", theSeller);
```

然而，还有一种允许绑定任何Hibernate类型实参的一般方法：

```
String queryString = "from Item item"
                   + " where item.seller = :seller and"
                   + " item.description like :desc";

session.createQuery(queryString)
       .setParameter( "seller",
                      theSeller,
                      Hibernate.entity(User.class) )
       .setParameter( "desc", description, Hibernate.STRING );
```

这甚至适用于定制的用户自定义的类型，如MonetaryAmount：

```
Query q = session.createQuery("from Bid where amount > :amount");
q.setParameter( "amount", givenAmount,
                Hibernate.custom(MonetaryAmountUserType.class) );
```

如果你有包含seller和description属性的JavaBean，就可以调用setProperties()方法来绑定查询参数。例如，可以在Item类自身的实例中传递查询参数：

```
Item item = new Item();
item.setSeller(seller);
item.setDescription(description);

String queryString = "from Item item"
```

```
                    + " where item.seller = :seller and"
                    + " item.description like :desccription";

session.createQuery(queryString).setProperties(item);
```

setProperties()绑定使得JavaBean属性的名称与查询字符串中的具名参数一致，内部调用setParameter()来猜测Hibernate类型并绑定值。实际上，结果表明它并没有听起来得这么有用，因为有些常用的Hibernate类型是不可猜测的（尤其是临时的类型）。

Query的参数绑定方法是可以为空（null-safe）的。因此下列代码是有效的：

```
session.createQuery("from User as u where u.username = :name")
        .setString("name", null);
```

然而，这段代码的结果几乎确定不是你想要的！生成的SQL将包含一个比较式（如USERNAME=null，它在SQL三重逻辑中始终取值为空）。反之，你必须使用is null操作符：

```
session.createQuery("from User as u where u.username is null");
```

5. 利用定位参数

如果喜欢，可以在Hibernate和Java Persistence中用定位参数代替：

```
String queryString = "from Item item"
                    + " where item.description like ?"
                    + " and item.date > ?";

Query q = session.createQuery(queryString)
                 .setString(0, searchString)
                 .setDate(1, minDate);
```

Java Persistence也支持定位参数：

```
String queryString = "from Item item"
                    + " where item.description like ?1"
                    + " and item.date > ?2";

Query q = em.createQuery(queryString)
                .setParameter(1, searchString)
                .setParameter(2, minDate, TemporalType.DATE);
```

这段代码不仅比使用具名参数的方法更难懂，而且如果稍微改变查询字符串，还有更容易破碎的弱点：

```
String queryString = "from Item item"
                    + " where item.date > ?"
                    + " and item.description like ?";
```

绑定参数位置的每一个变化都需要改变参数绑定代码。这样就产生了易碎和更需要维护的代码。建议避免使用定位参数。如果你通过编程构建复杂的查询，它可能更加方便，但是，此时Criteria API是更好的选择。

如果必须使用定位参数，要记住Hibernate是从0开始计数，而Java Persistence则是从1开始计数，必须给JPA QL查询字符串中的每一个问号都添加数字。它们有着不同的遗留根：Hibernate的根是JDBC，Java Persistence的根是更早版本的EJB QL。

除了绑定参数之外，你还会经常想要应用其他影响查询如何执行的提示（hint）。

6. 设置查询提示

假设你在执行查询之前修改了持久化对象。这些修改只出现在内存中，因此Hibernate（和Java Persistence提供程序）在执行查询之前，把持久化上下文和所有的变化清除到数据库。这样保证了查询在当前的数据中运行，并保证在查询结果和内存之间不会出现冲突。

这有时候是不现实的：例如，如果你执行一系列包含许多个查询－修改－查询－修改的操作，并且每次查询都获取一个与以前不同的数据集。换句话说，不需要在执行查询之前把修改清除到数据库，因为冲突的结果不是问题。注意，持久化上下文给实体对象提供可重复读取，因此无论如何都只有查询的标量结果才是问题。

可以在Session或者EntityManager中，利用setFlushMode()禁用持久化上下文的清除。或者，如果你想要只在特定的查询之前禁用清除，可以在Query（Hibernate和JPA）对象中设置一个FlushMode：

```
Query q = session.createQuery(queryString)
                 .setFlushMode(FlushMode.COMMIT);
Query q = em.createQuery(queryString)
            .setFlushMode(FlushModeType.COMMIT);
```

Hibernate不会在执行任何这些查询之前清除持久化上下文。

另一种优化是对特定查询结果的一个细粒度的org.hibernate.CacheMode。你在13.4.5节中用了一种高速缓存模式，控制Hibernate如何与二级高速缓存交互。如果Hibernate通过标识符获取对象，它在一级持久化上下文高速缓存中查找，如果二级高速缓存被启用，还在二级高速缓存区域查找这个实体。执行一个返回实体实例的查询时，也发生同样的事：在查询结果的封送期间，Hibernate试图先从持久化上下文高速缓存中查找它们，来解析所有的实体实例——如果实体实例处在持久化上下文高速缓存中，它就忽略查询结果的实体数据。并且，如果获取到的实体实例不在任何高速缓存中，Hibernate就会在查询结束之后把它放在高速缓存中。可以在查询中通过CacheMode控制这个行为：

```
Query q = session.createQuery("from Item")
                 .setCacheMode(CacheMode.IGNORE);

Criteria criteria = session.createCriteria(Item.class)
                           .setCacheMode(CacheMode.IGNORE);

Query q = em.createQuery(queryString)
            .setHint("org.hibernate.cacheMode",
                     org.hibernate.CacheMode.IGNORE);
```

例如，CacheMode.IGNORE告诉Hibernate不要为这个查询返回的任何实体而与二级高速缓存交互。换句话说，通过这个查询获取到的任何Item都不放在二级高速缓存中。如果你执行一个不应该更新二级高速缓存的查询，设置这种高速缓存模式就很有用，或许因为你正在获取的数据只与特定的情况相关，因此不应该耗尽高速缓存区域中的可用空间。

9.3.3节讲过持久化上下文的控制，以及如何减少内存消耗并防止很长的脏查询周期。给特定的持久化对象禁用脏查询的一种方法是设置session.setReadOnly(object,true)(EntityMa-

nager不支持这个API)。

你可以告诉Hibernate：由查询返回的所有实体对象都应该被当作是只读的（虽然不是脱管）：

```
Query q = session.createQuery("from Item")
                 .setReadOnly(true);
Query q = em.createQuery("select i from Item i")
              .setHint("org.hibernate.readOnly", true);
```

由这个查询返回的所有Item对象都处于持久化状态，但是在持久化上下文中，没有给自动脏检查启用任何快照。Hibernate不会自动持久化任何修改，除非用session.setReadOnly(object, false)禁用只读模式。

可以通过设置超时（timeout），来控制允许一个查询运行多久：

```
Query q = session.createQuery("from Item")
                 .setTimeout(60); // 1 minute

Criteria criteria = session.createCriteria(Item.class)
                           .setTimeout(60);
Query q = em.createQuery("select i from Item i")
            .setHint("org.hibernate.timeout", 60);
```

这种方法与JDBC Statement中的setQueryTimeout()方法有着相同的语义和结果。而且还与底层JDBC相关的是抓取大小（fetch size）：

```
Query q = session.createQuery("from Item")
                 .setFetchSize(50);

Criteria criteria = session.createCriteria(Item.class)
                           .setFetchSize(50);

Query q = em.createQuery("select i from Item i")
            .setHint("org.hibernate.fetchSize", 50);
```

JDBC抓取大小是对数据库驱动程序的一个优化提示；如果驱动程序没有实现这个功能，它就不能导致任何性能改善。如果它实现了，当客户端在一个查询结果（即ResultSet）中操作时，通过在一个批量中获取许多个行，可以改善JDBC客户端和数据库之间的通信。由于Hibernate幕后正在使用ResultSet，如果用list()执行一个查询，这个提示就可以改善数据获取——我们很快会实践这一点。

优化应用程序的时候，经常必须阅读复杂的SQL日志。强烈建议启用hibernate.use_sql_comments；然后Hibernate就会为它写到日志中的每条SQL语句添加一条注释。可以通过setComment()给特定的查询设置定制的注释：

```
Query q = session.createQuery("from Item")
                 .setComment("My Comment...");

Criteria criteria = session.createCriteria(Item.class)
                           .setComment("My Comment...");

Query q = em.createQuery("select i from Item i")
            .setHint("org.hibernate.comment", "My Comment...");
```

目前为止你已经设置的提示全部都与Hibernate或者JDBC处理相关。许多开发人员（和数据

库管理员）把查询提示当作是完全不同的东西。在SQL中，查询提示是SQL语句中的一条注释，它给数据库管理系统的SQL优化器包含了一条指令。例如，如果开发人员或者数据库管理员认为，数据库优化器为特定的SQL语句所选择的执行计划不是最快的，他们就用提示来强制一个不同的执行计划。Hibernate和Java Persistence的API不支持任意的SQL提示，必须退回到原生的SQL并编写自己的SQL语句——当然，你可以通过所提供的API来执行语句。

（通过一些数据库管理系统，可以在SQL语句一开始时就通过SQL注释来控制优化器。在这个例子中，是用Query.setComment()添加提示。在其他的场景中，可以编写org.hibernate.Interceptor，并在它被发送到数据库之前，在onPrepareStatement(sql)方法中操作SQL语句。）

最后，可以控制查询是否应该在数据库管理系统中强制悲观锁——这是一直持续到数据库事务结束的锁：

```
Query q = session.createQuery("from Item item")
                 .setLockMode("item", LockMode.UPGRADE);

Criteria criteria = session.createCriteria(Item.class)
                           .setLockMode(LockMode.UPGRADE);
```

这两个查询，如果得到数据库方言的支持，都会生成一个包括... FOR UPDATE操作（或者其等价物，如果得到数据库系统和方言的支持的话）的SQL声明。目前，悲观锁在Java Persistence查询接口中不可用（但它被规划为一个Hibernate扩展提示）。

如果现在准备好了查询，那么你可以运行它们了。

14.1.2　执行查询

一旦你创建并准备好了Query或者Criteria对象，就准备执行它，并把结果获取到内存中去。在一次循环中把整个结果获取到内存中去，这是执行查询的一种最常用的方法，我们称之为列表（listing）。接下来我们也要讨论一些其他的可用选项：迭代（iterating）和滚动（scrolling）。滚动的用处与迭代差不多：你很少需要这些选项。我们猜测在常规的应用程序中，有90%的查询执行都依赖list()和getResultList()方法。

首先来看一个最常见的案例。

1．列出所有结果

在Hibernate中，list()方法执行查询并返回结果为java.util.List：

```
List result = myQuery.list();
```

Criteria接口也支持这个操作：

```
List result = myCriteria.list();
```

在这两个案例中，是立即执行一个还是几个SELECT语句，这取决于你的抓取计划。如果映射任何关联或者集合为非延迟，就必须抓取它们，另外还要抓取你想要通过查询获取的数据。所有这些对象都被加载到内存中，获取到的任何实体对象都处于持久化状态，并被添加到持久化上下文。

Java Persistence用相同的语义提供了一种方法，但是用不同的名称：

```
List result = myJPAQuery.getResultList();
```

你知道，利用有些结果只是单个实例的查询——例如，如果你只想要最高的出价。如果是这样，就可以利用索引result.get(0)从结果清单中读取它。或者可以利用setMaxResult(1)限制返回的行数。然后，可以通过uniqueResult()方法执行查询，因为你知道只会返回一个对象：

```
Bid maxBid =
    (Bid) session.createQuery("from Bid b order by b.amount desc")
                 .setMaxResults(1)
                 .uniqueResult();
Bid bid = (Bid) session.createCriteria(Bid.class)
                       .add( Restrictions.eq("id", id) )
                       .uniqueResult();
```

如果查询返回不止一个对象，就抛出异常。如果查询结果为空，就返回null。这种情况在Java Persistence中也适用，只是方法的名称不同罢了（不幸的是，如果结果为空，就会抛出异常）：

```
Bid maxBid = (Bid) em.createQuery(
                       "select b from Bid b order by b.amount desc"
                     ).setMaxResults(1)
                     .getSingleResult();
```

把所有的结果获取到内存中是执行查询最常用的方法。Hibernate支持一些其他的方法，如果你想要优化查询的内存消耗和执行行为，可能会发现它们很有帮助。

2. 循环访问结果

Hibernate Query接口也提供iterate()方法来执行查询。它返回与list()一样的数据，但是依赖不同的策略来获取结果。

调用iterate()执行查询时，Hibernate在第一个SQL SELECT中只获取实体对象的主键（标识符）值，然后试图在持久化上下文高速缓存和（如果启用）二级高速缓存中查找对象的其他状态。

考虑下列代码：

```
Query categoryByName =
    session.createQuery("from Category c where c.name like :name");
categoryByName.setString("name", categoryNamePattern);
List categories = categoryByName.list();
```

这个查询导致至少一个SQL SELECT的执行，CATEGORY表的所有列都包括在该SELECT子句中：

```
select CATEGORY_ID, NAME, PARENT_ID from CATEGORY where NAME like ?
```

如果你希望类别已经被高速缓存在持久化上下文或者二级高速缓存中，那就只需要标识符值（高速缓存的键）。这样减少了从数据库中抓取的数据量。下列SQL效率更高一些：

```
select CATEGORY_ID from CATEGORY where NAME like ?
```

可以给它使用iterate()方法：

```
Query categoryByName =
    session.createQuery("from Category c where c.name like :name");
categoryByName.setString("name", categoryNamePattern);
Iterator categories = categoryByName.iterate();
```

初始的查询只获取Category主键值。然后你循环访问结果，Hibernate在当前的持久化上下文和二级高速缓存（如果启用的话）中查找每一个Category对象。如果高速缓存没有被访问到，Hibernate会给每次循环都执行一个额外的SELECT，通过Category对象的主键，从数据库中完整地获取它。

在大多数情况下，这是一项很小的优化。把行读取减到最少通常比把列读取减到最少更为重要。尽管如此，如果你的对象有很大的字符串字段，这种方法也可能有助于把网络中的数据包减到最少，因此把延迟也减到最小。应该清楚，只有当被迭代实体的二级高速缓存区域被启用时这种方法才真正有效。否则，它就会产生n+1查询问题！

Hibernate保持迭代器开着，直到你循环访问了所有的结果，或者直到Session关闭。也可以通过org.hibernate.Hibernate.close(iterator)显式地关闭它。

还要注意，在编写本书之时，Hibernate Criteria和Java Persistence还不支持这一优化。

执行查询的另一种优化方法是翻阅结果。

3．利用数据库游标滚动

简单的JDBC提供一种特性，称作可滚动的结果集（scrollable resultset）。这种方法使用一种保存在数据库管理系统中的游标。游标指向查询结果中一个特定的行，应用程序可以前后移动游标。甚至可以通过游标跳到一个特定的行。

应该翻阅查询结果（而不是把它们全部加载到内存中）的其中一种情形是结果集太大而无法载入内存。通常可以尝试通过增加查询中的条件进一步限制结果。但有时候这是不可能的，或许因为你需要所有的数据，却又想要分几步获取。

12.2.2节中已介绍过滚动，也讲解了如何实现处理批量数据的过程，因为这是它最能大展身手的地方。下列示例概述了ScrollableResults接口中其他值得关注的选项：

```
ScrollableResults itemCursor =
    session.createQuery("from Item").scroll();

itemCursor.first();
itemCursor.last();
itemCursor.get();

itemCursor.next();
itemCursor.scroll(3);
itemCursor.getRowNumber();
itemCursor.setRowNumber(5);
itemCursor.previous();
itemCursor.scroll(-3);

itemCursor.close();
```

这段代码没有太大的意义；它显示了ScrollableResults接口中最值得关注的方法。可以把游标设置到结果中的第一个和最后一个Item对象，或者使用获得游标当前正使用指向的Item。可以通过setRowNumber()跳到一个位置，到达一个特定的Item，或者通过previous()和next()前后滚动。另一种方法是利用scroll()，通过偏移前后滚动。

执行Hibernate Criteria查询也可以通过滚动而不是list()，返回的ScrollableResults游标的工作原理相同。注意，在终止数据库事务之前、用完游标之后绝对必须关闭它。下面是

Criteria示例，展现了游标的打开：

```
ScrollableResults itemCursor =
    session.createCriteria(Item.class)
            .scroll(ScrollMode.FORWARD_ONLY);

... // Scroll only forward

itemCursor.close()
```

Hibernate API的ScrollMode常量相当于简单JDBC中的常量。在这个示例中，该常量确保游标只能向前移动。这是必需的，以防万一，因为有些JDBC驱动程序并不支持向后滚动。其他可用的方法是ScrollMode.SCROLL_INSENSITIVE和ScrollMode.SCROLL_SENSITIVE。在游标开着时，迟钝的游标不会把你公开给被修改的数据（有效地保证没有脏读取、不可重复读取、或者幻读可以滑入到结果集中）。另一方面，当你在结果集中工作时，灵敏的游标会把刚刚被提交的数据和修改公开给你。注意，Hibernate持久化上下文高速缓存仍然给实体实例提供可重复读取，因此只有你在结果集中投影的被修改的标量值会受到这项设置的影响。

目前为止，我们介绍的代码示例全部都在Java代码中嵌入了查询字符串文字。对于简单的查询来说，这样做不无道理，但是一旦开始考虑必须被分成多行的复杂查询时，这就变得有点难以处理了。

14.1.3 使用具名查询

我们不想在Java代码中看见到处分散着HQL或者JPA QL字符串文字，除非真正有必要。Hibernate让你把查询字符串具体化到映射元数据，这种技术称作*具名查询*（named query）。它让你把与特定的持久化类或者一组类（该类封装了它的其他元数据）相关的所有查询都保存在一个XML映射文件中。或者，如果使用注解，就可以创建具名查询作为特定实体类的元数据，或者把它们放在一个XML部署描述符中。查询的名称被用来从应用程序代码中进行调用。

1. 调用具名查询

在Hibernate中，getNamedQuery()方法给具名的查询获得Query实例：

```
session.getNamedQuery("findItemsByDescription")
        .setString("desc", description);
```

在这个例子中，你调用了具名查询findItemsByDescription，并把一个字符串参数绑定到具名参数desc。

Java Persistence也支持具名查询：

```
em.createNamedQuery("findItemsByDescription")
   .setParameter("desc", description);
```

具名查询是全局的——也就是说，查询的名称被当作是特定的SessionFactory或者持久化单元的唯一标识符。它们如何以及在哪里定义，在XML映射文件中还是在注解中，都与应用程序代码无关。甚至查询语言也不关心。

2. 在XML元数据中定义具名查询

可以把一个具名查询放在XML元数据中任何<hibernate-mapping>元素内部。在更大的应

用程序中，建议把所有具名查询隔离并分离到它们自己的文件中去。或者，你可能想让某些查询在相同的XML映射文件中定义为一个特定的类。

<query>定义具名的HQL或者JPA QL查询：

```
<query name="findItemsByDescription"><![CDATA[
        from Item item where item.description like :desc
]]></query>
```

应该把查询文本包装在一个CDATA指令中，以便XML解析器不会被查询字符串中可能意外地被当作XML的任何字符而混淆（例如 < 操作符）。

如果你把具名查询定义放在<class>元素内部，而不是在根的内部，它就用实体类的名称作为前缀。例如，findItemsByDescription也可以称作auction.model.Item.findItemsByDescription。否则，你要确保查询的名称是全局唯一的。

你之前利用API设置的所有查询提示也可以通过声明设置：

```
<query name="findItemsByDescription"
       cache-mode="ignore"
       comment="My Comment..."
       fetch-size="50"
       read-only="true"
       timeout="60"><![CDATA[
    from Item item where item.description like :desc
]]></query>
```

具名查询不一定是HQL或者JPA QL字符串；它们甚至可以是原生的SQL查询——并且你的Java代码也不需要知道这种差别：

```
<sql-query name="findItemsByDescription">
    <return alias="item" class="Item"/>
    <![CDATA[
        select {item.*} from item where description like :desc
    ]]>
</sql-query>
```

如果你认为稍后可能想要通过调优SQL来优化查询，这就很有用。如果必须把一个遗留应用程序移植到Hibernate，其中SQL代码从手工编码的JDBC子程序中隔离开来，这也是一种很好的解决方案。利用具名查询，可以轻松地逐个地把查询移植到映射文件。第15章将更详细地讨论原生的SQL查询。

3. 利用注解定义具名查询

Java Persistence标准指定@NameQuery和@NamedNativeQuery注解。可以把这些注解放进特定类的元数据内，也可以放在JPA XML描述符文件内。注意，查询名称在任何情况下都必须是全局唯一的；不会自动加上任何类或者包名称作为前缀。

假设你考虑一个特定的具名查询属于特定的实体类：

```
package auction.model;

import ...;

@NamedQueries({
    @NamedQuery(
```

```
        name = "findItemsByDescription",
        query = "select i from Item i where i.description like :desc"
      ),
      ...
})
@Entity
@Table(name = "ITEM")
public class Item { ... }
```

一种更为常用的解决方案就是把查询封装在orm.xml部署描述符中：

```
<entity-mappings ...>
    ...
    <named-query name="findAllItems">
        <query>select i from Item i</query>
    </named-query>

    <entity class="Item">
        ...
        <named-query name="findItemsByDescription">
          <query>
               select i from Item i where i.description like :desc
          </query>
          <hint name="org.hibernate.comment" value="My Comment"/>
          <hint name="org.hibernate.fetchSize" value="50"/>
          <hint name="org.hibernate.readOnly" value="true"/>
          <hint name="org.hibernate.timeout" value="60"/>
        </named-query>
    </entity>

</entity-mappings>
```

你会看到Java Persistence描述符支持扩展点：named-query定义的hint元素。可以用它设置特定于Hibernate的提示，就像之前用Query接口通过编程进行的那样。

原生的SQL查询有它们自己的元素，也可以在一个实体映射的内部或者外部定义它们：

```
<named-native-query name="findItemsByDescription"
                    result-set-mapping="myItemResult">
    <query>select i.NAME from ITEM i where i.DESC = :desc</query>
    <hint name="org.hibernate.timeout" value="200"/>
</named-native-query>
```

嵌入原生的SQL比我们目前为止介绍过的都更强大（可以定义任意的结果集映射）。第15章将回来讨论其他的SQL嵌入选项。

是否要利用具名查询特性，这要由你自己决定。然而，我们把应用程序代码中的查询字符串（除非它们处在注解中）当作次选；如有可能，你通常应该把查询字符串具体化到外部的映射文件中去。

你现在知道如何通过Hibernate、JPA以及元数据创建、准备和执行查询了。下面应该更深入地学习查询语言和选项。我们从HQL和JPA QL开始。

14.2 基本的 HQL 和 JPA QL 查询

从一些简单的查询开始，来熟悉HQL语法和语义。我们应用选择来指定数据源，用限制使记

录与条件相匹配，并用投影选择想要从查询中返回的数据。

试一试　测试Hibernate查询——用于Eclipse IDE的Hibernate Tools支持Hibernate Console（控制台）视图。可以在控制台窗口测试查询，并立即看到生成的SQL和结果。

你还将在本节中学习JPA QL，因为它是HQL功能的一个子集——必要时我们会提到其中的区别。

本节谈到查询时，通常是指SELECT语句，即从数据库中获取数据的操作。HQL也支持UPDATE、DELETE甚至INSERT .. SELECT语句，就如12.2.1节所述。这里不再重复这些大批量操作，而是关注SELECT语句。但是记住，HQL和JPA QL之间的某些区别也可能适用于大批量操作——例如，一个特定的函数是否可移植。

HQL中的SELECT语句甚至不用SELECT子句就可以生效；只需要FROM。JPA QL则不是这样，在JPA QL中SELECT子句不是可选的。这实际上没有太大的区别；几乎所有的查询都需要SELECT子句，无论你用JPA QL还是HQL编写。然而，我们要从FROM子句开始探讨查询，因为依据我们的经验，这样更容易理解。记住，要把这些查询变成JPA QL，理论上你必须添加SELECT子句来完成语句，但是如果你忘了（假设是SELECT *），无论如何Hibernate还是会让你执行查询。

14.2.1　选择

HQL中最简单的查询是单个持久化类的一个选择（注意，这里我们不是指SELECT子句或者语句，而是指从数据被选择的位置）：

```
from Item
```

这个查询生成下列SQL：

```
select i.ITEM_ID, i.NAME, i.DESCRIPTION, ... from ITEM i
```

1. 利用别名

通常，通过使用HQL或者JPA QL选择一个类进行查询时，你需要分配一个别名给被查询的类，用作查询其他部分的一个引用：

```
from Item as item
```

as关键字通常是可选的。上面的代码相当于下列代码：

```
from Item item
```

把这个稍微想象成下列Java代码中的临时变量声明：

```
for ( Iterator i = allQueriedItems.iterator(); i.hasNext(); ) {
    Item item = (Item) i.next();
    ...
}
```

把别名item分配到Item类的被查询实例，允许你稍后在代码（或者查询）中引用它们的属性值。为了提醒自己这个类似之处，建议你给别名使用与临时变量[通常采用骆驼命名法（camelCase）]相同的命名约定。然而，本书中的某些示例可能使用更短的别名，例如用i代替

item，以保证印刷出来的代码易于阅读。

常见问题 HQL和JPA QL区分大小写吗？我们从来不用大写字母编写HQL和JPA QL关键字，也从来不用大写字母编写SQL关键字。那样看起来难看并且古板——最现代的终端机可以显示大写和小写字符。然而，HQL和JPQ QL对于关键字并不区分大小写，因此如果你喜欢，也可以编写为FROM Item AS item。

2. 多态查询

作为面向对象的查询语言，HQL和JPA QL支持多态查询（polymorphic query）——相应地查询类的实例和它子类的所有实例。你已经掌握了足够的HQL和JPA QL知识来证明这一点。考虑下列查询：

```
from BillingDetails
```

这句代码返回类型BillingDetails的对象，它是个抽象类。在这个例子中，具体的对象属于BillingDetails:CreditCard和BankAccount的子类型。如果只想要一个特定子类的实例，可以使用：

```
from CreditCard
```

在from子句中具名的类甚至不需要一定是被映射的持久化类，任何类都可以！下列查询返回所有的持久化对象：

```
from java.lang.Object
```

当然，这也适用于接口——这个查询返回所有可序列化的持久化对象：

```
from java.io.Serializable
```

同样地，下列条件查询返回所有的持久化对象（是的，可以通过这样一个查询来选择数据库的所有表）：

```
from java.lang.Object
```

注意，Java Persistence并没有标准化利用未被映射的接口的多态查询。然而，这适用于Hibernate EntityManager。

多态不仅仅应用到FROM子句中显式命名的类，还应用到多态关联，就像你将在本章稍后会见到的那样。

已经讨论过了FROM子句，现在进入HQL和JPA QL的其他部分。

14.2.2 限制

通常，你并不想要获取一个类的所有实例。你必须能够在查询返回的对象的属性值中表达约束。这称作限制。WHERE子句用来表达SQL、HQL和JPA QL中的限制。这些表达式可能跟你正在寻找的需要缩小的代码块一样复杂。注意，限制不仅应用到SELECT语句，还可用于限定UPDATE或者DELETE操作的范围。

这是一个典型的WHERE子句，它限制结果为带有指定的电子邮件地址的所有User对象：

```
from User u where u.email = 'foo@hibernate.org'
```

注意，该约束是根据User类的属性email进行表达的，并给它用了一个面向对象的概念。这个查询生成的SQL是：

```
select u.USER_ID, u.FIRSTNAME, u.LASTNAME, u.USERNAME, u.EMAIL
 from USER u
 where u.EMAIL = 'foo@hibernate.org'
```

可以在语句和条件中用单引号包括文字。HQL和JPA QL中常用的其他文字是TRUE和FALSE：

```
from Item i where i.isActive = true
```

限制利用三重逻辑进行表达。WHERE子句是一个逻辑表达式，给对象的每一个字节组（tuple）都取值为true、false或者null。利用内建的比较操作符，通过把对象的属性与其他属性或者文字值进行对比，来构建逻辑表达式。

常见问题　什么是三重逻辑？当且仅当WHERE子句取值为true时，行才被包括在SQL结果集中。在 Java 中，notNullObject==null 取值为 false，null==null 取值为 true。在 SQL 中，NOT_NULL_COLUMN=null和null=null都取值为null，而不是true。这样，SQL就需要一个特定的操作符IS NULL来测试一个值是否为null。这个三重逻辑是处理表达式的一种方法，可以应用到空的列值。不要把null当作特殊的标记，而是作为普通的值处理，这是对关系模型中常见的二重逻辑的SQL扩展。HQL和JPA QL必须利用三重操作符支持这个三重逻辑。

让我们快速看一下最常用的比较操作符。

1. 比较表达式

HQL和JPA QL支持与SQL相同的基本比较操作符。如果了解SQL，以下是一些看起来应该很熟悉的示例：

```
from Bid bid where bid.amount between 1 and 10
from Bid bid where bid.amount > 100
from User u where u.email in ('foo@bar', 'bar@foo')
```

因为底层的数据库实现了三重逻辑，测试null值需要很谨慎。记住null = null在SQL中不会取值为true，而是为null。使用null操作数的所有比较都取值为null。（这就是为什么通常在查询中看不到null文字的原因。）HQL和JPA QL提供SQL风格的IS [NOT] NULL操作符：

```
from User u where u.email is null
from Item i where i.successfulBid is not null
```

这个查询返回所有没有电子邮件地址的用户和被出售的货品。

LIKE操作符允许通配符搜索，在这里通配符符号是%和_，就像SQL中的一样：

```
from User u where u.firstname like 'G%'
```

这个表达式限制了结果为带有firstname的以大写字母G开头的用户。也可以否定LIKE操作符，例如在一个子字符串匹配表达式中：

```
from User u where u.firstname not like '%Foo B%'
```

百分比符号代表字符的任何一种顺序；下划线可以用来通配单个字符。如果想要文字百分比或者下划线，可以定义转义符：

```
from User u where u.firstname like '\%Foo%' escape='\'
```

这个查询返回带有名字的以%Foo开头的所有用户。

HQL和JPA QL支持算术表达式：

```
from Bid bid where ( bid.amount / 0.71 ) - 100.0 > 0.0
```

逻辑操作符（和用于分组的圆括号）用于组合表达式：

```
from User user
        where user.firstname like 'G%' and user.lastname like 'K%'
from User u
        where ( u.firstname like 'G%' and u.lastname like 'K%' )
        or u.email in ('foo@hibernate.org', 'bar@hibernate.org' )
```

可以在表14-1中从上到下看到操作符的优先顺序。

表14-1 HQL和JPA QL操作符优先顺序

操 作 符	描 述
.	导航路径表达式操作符
+,-	一元的正号或负号（所有无正负之分的数值均被当作正值）
*,/	数值的一般乘法和除法
+,-	数值的一般加法和减法
=, <>, <, >, >=, <=, [NOT] BETWEEN, [NOT] LIKE, [NOT] IN, IS [NOT] NULL, IS [NOT] EMPTY, [NOT] MEMBER [OR]	包含SQL语义的二元比较操作符
NOT, AND, OR	HQL和JPA QL中用于集合的二元操作符 用来排列表达式取值的逻辑操作符

这里列出的操作符和它们的优先顺序在HQL和JPA QL中是一样的。算术操作符（例如乘法和加法）都是一目了然的。你已经见过二重比较表达式如何有着与它们的SQL配对物相同的语义，以及如何通过逻辑操作符对它们进行分组和组合。我们来讨论集合处理。

2. 包含集合的表达式

前几节中的所有表达式都只包括单值的路径表达式：user.email、bid.amount等。也可以通过正确的操作符，在查询的WHERE子句中使用以集合结束的路径表达式。

例如，假设你想要通过集合的大小来限制查询结果：

```
from Item i where i.bids is not empty
```

这个查询返回它们的bids集合中有元素的所有Item实例。也可以表达你需要一个特定的元素出现在一个集合中：

```
from Item i, Category c where i.id = '123' and i member of c.items
```

这个查询返回Item和Category实例——通常添加SELECT子句，并只投影这两个实体类型的其中之一。它返回包含主键'123'（单引号中的一个文字）的Item实例，以及与这个Item实例关

联的所有Category实例。（这里使用的另一个技巧是特殊的.id路径；这个字段始终指向一个实体的数据库标识符，无论标识符属性的名称是什么。）

在HQL和JPA QL中使用集合还有许多其他的方法。例如，可以在函数调用中使用它们。

3. 调用函数

HQL一项极为强大的特性是在WHERE子句中调用SQL函数的能力。如果数据库支持用户自定义的函数（大多数都可以），不管怎样，你就可以把这一点用于各种目的。至于现在，我们要考虑使用标准的ANSI SQL函数UPPER()和LOWER()。这些可以用于区分大小写的搜索：

```
from User u where lower(u.email) = 'foo@hibernate.org'
```

另一个常用的表达式是拼接——虽然这里的SQL方言不同，但HQL和JPA QL支持可移植的concat()函数：

```
from User user
        where concat(user.firstname, user.lastname) like 'G% K%'
```

同样典型的是需要集合大小的表达式：

```
from Item i where size(i.bids) > 3
```

JPA QL标准化了最常用的函数，如表14-2中所概括的。

表14-2　标准的JPA QL函数

函　数	适　用　性
UPPER(s), LOWER(s)	字符串值。返回一个字符串值
CONCAT(s1, s2)	字符串值。返回一个字符串值
SUBSTRING(s, offset, length)	字符串值（从1开始偏移）。返回一个字符串值
TRIM([[BOTH|LEADING|TRAILING] char [FROM]] s)	如果没有指定char或者其他的规范，就去除s的两端空白。返回一个字符串值
LENGTH(s)	字符串值。返回一个数值
LOCATE(search, s, offset)	从offset开始搜索s字符串中search字符串所在的位置。返回一个数字值
ABS(n), SQRT(n), MOD(dividend, divisor)	数字值。返回一个与输入类型相同的绝对值，double类型的平方根，以及一个整数除以另一个整数之后产生的integer类型的余数
SIZE(c)	集合表达式。返回一个integer，或者如果为空则返回0

所有标准的JPA QL函数都可以用在查询的WHERE和HAVING子句中（稍后你很快会见到）。原生的HQL更灵活一点。首先，它提供其他的可移植函数，如表14-3所示。

表14-3　其他的HQL函数

函　数	适　用　性
BIT_LENGTH(s)	返回s的位数
CURRENT_DATE(), CURRENT_TIME(), CURRENT_TIMESTAMP()	返回数据库管理系统机器的日期和（或）时间
SECOND(d), MINUTE(d), HOUR(d), DAY(d), MONTH(d), YEAR(d)	从临时的实参中提取时间和日期

（续）

函 数	适 用 性
CAST(t as Type)	把指定的类型t转换为Hibernate的Type
INDEX(joinedCollection)	返回被联结的集合元素的索引
MINELEMENT(c), MAXELEMENT(c), MININDEX(c), MAXINDEX(c), ELEMENTS(c), INDICES(c)	返回被索引集合（映射、列表、数组）的元素或者索引
已经在org.hibernate.Dialect中注册的	以一种方言扩展包含其他函数的HQL

这些HQL函数中的大多数，转变成了可能你之前已经用过的SQL中的配对物。这个转换表可以通过org.hibernate.Dialect定制和扩展。检查你正用给数据库的方言的源代码；你可能在那里发现有许多已经注册的其他SQL函数，可以在HQL中立即使用。记住，没有包括在org.hibernate.Dialect超类中的每一个函数，对于其他的数据库管理系统都是不可移植的！

另一个新增到Hibernate API的是Hibernate Configuration API中的addSqlFunction()方法：

```
Configuration cfg = new Configuration();
cfg.addSqlFunction(
     "lpad",
     new StandardSQLFunction("lpad", Hibernate.STRING)
);
... cfg.buildSessionFactory();
```

这个操作把SQL函数lpad添加到HQL。请见StandardSQLFunction及其子类的Javadoc，了解更多的信息。

当你调用一个没有对SQL方言进行注册的函数时，HQL甚至尝试变得聪明些：在HQL语句的WHERE子句中调用的且不为Hibernate所知的任何函数，被作为SQL函数调用直接传递到数据库。如果你不在乎数据库的可移植性，这种方法就很棒，否则就必须注意那些不可移植的函数。

最后，在探讨HQL和JPA QL中的SELECT子句之前，看看结果是如何排序的。

4. 排序查询结果

所有的查询语言都提供一些排序查询结果的机制。HQL和JPA QL提供了ORDER BY子句（类似于SQL）。

这个查询返回所有用户，按用户名排序：

```
from User u order by u.username
```

利用asc或者desc指定按升序或者降序排序：

```
from User u order by u.username desc
```

可以按多种属性进行排序：

```
from User u order by u.lastname asc, u.firstname asc
```

你现在知道如何编写FROM、WHERE和ORDER BY子句了。你知道如何选择想要获取的实体的实例，以及选择限制和排序结果所需的表达式和操作。你现在需要的就是把这个结果的数据投影到你在应用程序中需要的东西的能力。

14.2.3　投影

SELECT子句在HQL和JPA QL中执行投影。它允许你准确指定需要查询结果中的哪些对象或者对象的哪些属性。

1．实体和标量值的简单投影

例如，考虑下列HQL查询：

```
from Item i, Bid b
```

这是有效的HQL查询，但在JPA QL中是无效的——标准要求你使用SELECT子句。尽管如此，Item和Bid的乘积所隐含的结果，同样也可以通过显式的SELECT子句生成。这个查询返回有序的Item和Bid实例对：

```
Query q = session.createQuery("from Item i, Bid b");
// Query q = em.createQuery("select i, b from Item i, Bid b");

Iterator pairs = q.list().iterator();
// Iterator pairs = q.getResultList().iterator();

while ( pairs.hasNext() ) {
    Object[] pair = (Object[]) pairs.next();
    Item item = (Item) pair[0];
    Bid bid = (Bid) pair[1];
}
```

这个查询返回Object[]的一个List。索引0处为Item，索引1处为Bid。因为这是一个乘积，结果包含了在两张底层的表中找到的Item和Bid行的每一种可能的组合。很显然，这个查询没用，但是当你收到一个Object[]的集合作为查询结果时，也不应该感到惊讶。

下列显式的SELECT子句也返回Object[]的一个集合：

```
select i.id, i.description, i.initialPrice
    from Item i where i.endDate > current_date()
```

这个查询返回的Object[]在索引0处包含了一个Long，在索引1处包含一个String，并在索引2处包含一个BigDecimal或者MonetaryAmount。这些都是标量值，而不是实体实例。因此，它们不处于任何持久化状态，就像实体实例那样。它们不是事务的，显然也不会自动对脏状态进行检查。我们称这种查询为标量查询（scalar query）。

2．获取独特的结果

当你使用SELECT子句时，结果的元素不再保证是唯一的。例如，货品描述就不是唯一的，因此下列查询不止一次地返回相同的描述：

```
select item.description from Item item
```

很难理解在一个查询结果中有两个相同的行会有什么意义，因此如果你认为可能有重复，一般就使用DISTINCT关键字：

```
select distinct item.description from Item item
```

这样就从返回的Item描述列表中消除了重复。

3．调用函数

（对于某些Hibernate SQL方言来说）也可能从SELECT子句中调用数据库特定的SQL函数。例

如，下列查询从数据库服务器（Oracle语法）中获取当前的日期和时间，以及Item的一个属性：

```
select item.startDate, current_date() from Item item
```

SELECT子句中数据库函数的技术不受限于依赖数据库的函数。它也适用于其他更多一般的（或者标准的）SQL函数：

```
select item.startDate, item.endDate, upper(item.name)
    from Item item
```

这个查询返回Object[]，包含一件货品拍卖的起始和终止日期，货品的名称全部大写。

在实践中，也可能调用SQL统计函数（aggregate function），本章稍后会讨论到。然而，注意Java Persistence标准和JPA QL不保证任何非统计函数的函数可以在SELECT子句中调用。Hibernate和HQL提供更多的灵活性，因此我们认为支持JPA QL的其他产品将在某种程度上提供同样的自主。还要注意，Hibernate不知道的函数并没有作为SQL函数调用被传递到数据库，就像在WHERE子句中一样。你必须在org.hibernate.Dialect中注册一个函数，在HQL中的SELECT子句中启用它。

前几节应该已经让你从基本的HQL和JPA QL开始了。现在该看一下更复杂的查询选项了，例如联结、动态抓取、子查询和报表查询。

14.3 联结、报表查询和子查询

很难把有些查询归类为是高级的，而把其他查询归类为基本的。很显然，我们在本章前几节中介绍过的查询还没能让你做到这一点。

至少你还需要知道联结是如何工作的。任意地联结数据的能力是关系数据访问的基础能力之一。联结数据也是基本的操作，它使你能够在单个查询中抓取几个被关联的对象和集合。我们现在要介绍基本的联结操作如何工作，以及如何使用它们编写动态抓取策略。

我们认为属于高级的其他技术，包括通过有效地对结果进行统计和分组的子查询和报表查询等各种语句的嵌套。

让我们从联结以及如何用联结进行动态抓取开始。

14.3.1 联结关系和关联

用一个联结组合两个（或者更多个）关系中的数据。例如，可以联结ITEM和BID表中的数据，如图14-1所示。（注意，并非所有的列和可能的行都显示出来；因此用省略号表示。）

ITEM

ITEM_ID	DESCRIPTION	...
1	Item Nr. One	...
2	Item Nr. Two	...
3	Item Nr. Three	...

BID

BID_ID	ITEM_ID	AMOUNT
1	1	99.00
2	1	100.00
3	1	101.00
4	2	4.99

图14-1 ITEM和BID表是联结操作明显的备选表

当人们在SQL数据库的上下文中听到联结一词时，大都认为是内部联结。内部联结是几种联结类型中最重要的一种，并且最容易理解。考虑SQL语句及图14-2。这个SQL语句是FROM子句中的ANSI类型的内部联结。

```
select i.*, b.* from ITEM i
inner join BID b on i.ITEM_ID = b.ITEM_ID
```

ITEM_ID	DESCRIPTION	...	BID_ID	ITEM_ID	AMOUNT
1	Item Nr. One	...	1	1	99.00
1	Item Nr. One	...	2	1	100.00
1	Item Nr. One	...	3	1	101.00
2	Item Nr. Two	...	4	2	4.99

图14-2　两张表的ANSI类型内部联结的结果表

如果你通过内部联结来联结表ITEM和BID，就利用它们的一般属性（ITEM_ID列），在一个新的结果表中获得所有货品和它们的出价。注意，这个操作的结果只包含有出价的货品。如果想要所有的货品，并且某种货品没有相应的出价时，就用NULL值代替其出价数据，此时就可以利用(左)外部联结，如图14-3所示。

```
select i.*, b.* from ITEM i
left outer join BID b on i.ITEM_ID = b.ITEM_ID
```

ITEM_ID	DESCRIPTION	...	BID_ID	ITEM_ID	AMOUNT
1	Item Nr. One	...	1	1	99.00
1	Item Nr. One	...	2	1	100.00
1	Item Nr. One	...	3	1	101.00
2	Item Nr. Two	...	4	2	4.99
3	Item Nr. Three	...	NULL	NULL	NULL

图14-3　两张表的ANSI类型左外部联结的结果

你可以认为表联结像下面这样工作。首先，通过取ITEM行与BID行所有可能的组合，得到两张表的一个乘积。

第二，使用联结条件过滤这些被联结的行。（任何一个好的数据库引擎都有较复杂的算法来计算联结。它通常不会构建消耗内存的乘积，然后过滤所有的行。）联结条件是一个布尔表达式，如果被联结的行会被包括在结果中，就取值为真。对于左外部联结而言，从不满足联结条件的(左)ITEM表中的每一行也都包括在结果中，给BID的所有列返回NULL值。

*右*外部联结获取所有的出价，或null（如果一个出价没有对应的货品）——此时这不是一个有意义的查询。右外部联结很少使用；开发人员始终考虑从左到右，并且先放置驱动表。

在SQL中，联结条件通常被显式指定。（不幸的是，不可能使用外键约束的名称来指定两张表如何联结。）你为ANSI类型的联结在ON子句中（或者为所谓的theta类型的联结在WHERE子句中）指定联结条件：`where I.ITEM_ID=B.ITEM_ID`。

我们现在讨论HQL和JPA QL联结选项。记住这两者都是基于SQL并转变成SQL，因此，即使语法稍有不同，你也应该始终参考前面介绍过的两个示例，并验证一下你所理解的结果SQL和结果集看起来是什么样子。

1. HQL和JPA QL联结选项

在Hibernate查询中，通常不显式指定联结条件，而是指定被映射的Java类关联的名称。这基本上与我们在SQL中想要的特性一样，通过一个外键约束名称表达联结条件。由于你已经在Hibernate中映射了大多数（如果不是全部）数据库Schema的外键关系，因此可以在查询语言中使用这些被映射关联的名称。这是真正的语法糖（syntactical sugar）[①]，但很方便。

例如，Item类有一个包含Bid类的关联具名bids。如果你在一个查询中命名这个关联，Hibernate在映射文档中就有足够的信息，然后推断出表联结表达式。这有助于使查询更为简短、更易于阅读。

事实上，HQL和JPA QL提供了4种表达（内部和外部）联结的方法：

- 隐式关联联结；
- FROM子句中的普通联结；
- FROM子句中的抓取联结；
- WHERE子句中theta类型的联结。

稍后介绍如何编写两个没有定义关联的类之间的（一个theta类型）联结，以及如何在查询的FROM子句中编写普通联结和抓取联结。

隐式关联联结是常用的缩写。（注意，我们决定要通过通常省略SELECT子句，使接下来的示例更易于阅读和理解——在HQL中有效，在JPA QL中则无效。）

2. 隐式关联联结

目前为止，你已经使用过简单的限定属性名，比如查询中的bid.amount和item.description。HQL和JPA QL支持大部分属性路径表达式，用一个点号表示两种不同的目的：

- 查询组件；
- 表达隐式关联联结。

第一种用法很简单：

```
from User u where u.homeAddress.city = 'Bangkok'
```

你用一个点号引用被映射组件Address的几个部分。在这个查询中没有联结任何表，homeAddress组件的属性与User数据一起全部被映射到相同的表。也可以在SELECT子句中编写一个路径表达式：

```
select distinct u.homeAddress.city from User u
```

这个查询返回String的一个List。由于重复没有太大的意义，因此用DISTINCT把它们排除。

大部分路径表达式的第二种用法是隐式关联联结：

```
from Bid bid where bid.item.description like '%Foo%'
```

① 指在不改变语法含义的前提下，把语法变得更加易于阅读。——编者注

这样就在从Bid到Item的多对一关联上产生了一个隐式联结——这个关联的名称是item。Hibernate知道你用BID表中的ITEM_ID外键映射了这个关联，并相应地生成SQL联结条件。隐式联结始终采用多对一或者一对一关联，从来不通过集合值的关联(无法编写item.bids.amount)。

单个路径表达式中可能存在多个联结。如果从Item到Category的关联是多对一（而不是目前的多对多），就可以这样编写：

```
from Bid bid where bid.item.category.name like 'Laptop%'
```

我们反对给更复杂的查询使用这种语法糖。SQL联结很重要，尤其当优化查询的时候，你需要能够一眼就看到有多少个联结。考虑下列查询（再次利用从Item到Category的多对一关联）：

```
from Bid bid
    where bid.item.category.name like 'Laptop%'
    and bid.item.successfulBid.amount > 100
```

在SQL中表达这个需要多少个联结？即使找到了正确答案，也花了你不少时间。答案是3个，生成的SQL看起来有点像这样：

```
select ...
from BID B
inner join ITEM I on B.ITEM_ID = I.ITEM_ID
inner join CATEGORY C on I.CATEGORY_ID = C.CATEGORY_ID
inner join BID SB on I.SUCCESSFUL_BID_ID = SB.BID_ID
where C.NAME like 'Laptop%'
and SB.AMOUNT > 100
```

如果在FROM子句中用显式的HQL和JPA QL联结来表达这个查询，它会更明显。

3. 在FROM子句中表达的联结

Hibernate区分不同的联结目的。假设你正在查询Item，你可能想要使用Bid来联结它们的原因有两种。

第一，你可能想要以应该被应用到货品Bid的一些条件为基础，来限制通过查询返回的货品。例如，你可能想要出价超过$100的所有Item；因而这需要一个内部联结。目前你还不关注没有出价的货品。

另一方面，你可能主要关注Item，但可能想要执行一个外部联结，只是因为想给在同一个SQL语句中查询的Item获取所有Bid，类似于我们之前称作即时联结抓取的东西。记住，你宁可在默认情况下延迟映射所有的关联，因此，即时的外部联结抓取查询，用于在运行时给特定的用例覆盖默认的抓取策略。

先来编写一些使用内部联结进行限制的查询。如果想要获取Item实例，并限制结果为其出价具有特定金额的货品，就必须为被联结的关联分配一个别名：

```
from Item i
    join i.bids b
        where i.description like '%Foo%'
        and b.amount > 100
```

这个查询为实体Item分配了别名i，为被联结的Item出价分配了别名b。然后可以用这两个别名在WHERE子句中表达限制条件。

生成的SQL是：

```
select i.DESCRIPTION, i.INITIAL_PRICE, ...
       b.BID_ID, b.AMOUNT, b.ITEM_ID, b.CREATED_ON
from ITEM i
inner join BID b on i.ITEM_ID = b.ITEM_ID
where i.DESCRIPTION like '%Foo%'
and b.AMOUNT > 100
```

查询以有序的配对形式返回被关联的Bid和Item的所有组合：

```
Query q = session.createQuery("from Item i join i.bids b");
Iterator pairs = q.list().iterator();
while ( pairs.hasNext() ) {
    Object[] pair = (Object[]) pairs.next();
    Item item = (Item) pair[0];
    Bid bid = (Bid) pair[1];
}
```

这个查询没有返回Item的一个List，而是返回Object[]数组的一个List。在索引0处为Item，索引1处为Bid。一个特定的Item可能出现多次，每个被关联的Bid一次。这些重复的货品是重复的内存引用，而不是重复的实例！

如果你不想要查询结果中的Bid，可以在HQL中指定一个SELECT子句（无论如何，它对于JPA QL都是强制的）。在SELECT子句中使用别名，只投影想要的对象：

```
select i
from Item i join i.bids b
where i.description like '%Foo%'
and b.amount > 100
```

现在，生成的SQL看起来像这样：

```
select i.DESCRIPTION, i.INITIAL_PRICE, ...
from ITEM i
inner join BID b on i.ITEM_ID = b.ITEM_ID
where i.DESCRIPTION like '%Foo%'
and b.AMOUNT > 100
```

查询结果只包含Item，由于它是一个内部联结，因此只有包含Bid的Item：

```
Query q = session.createQuery("select i from Item i join i.bids b");
Iterator items = q.list().iterator();
while ( items.hasNext() ) {
    Item item = (Item) items.next();
}
```

如你所见，在HQL和JPA QL中使用别名，对于直接的类和被联结的关联都是一样的。你在前面的示例中用了一个集合，但是对于单值的关联来说，其语法和语义都是相同的，例如多对一和一对一。通过命名关联，你在FROM子句中分配别名，然后在WHERE（或许也在SELECT子句）中使用别名。

HQL和JPA QL提供另一种语法，用于在FROM子句中联结集合，并给它分配一个别名。这个IN()操作符在更早版本的EJB QL中就出现了。它的语义与普通集合联结的一样。可以重写最后一个查询如下：

```
select i
from Item i in(i.bids) b
where i.description like '%Foo%'
and b.amount > 100
```

from Item i in(i.bids) b生成了与之前的示例中用from Item i join i.bids b一样的内部联结。

目前为止，你还只是编写了内部联结。外部联结主要用于动态抓取，我们很快会讨论到。有时候有你想要用外部联结而不应用动态的抓取策略，来编写一个简单的查询。例如，下列查询是第一种查询的变形，它获取金额最小的货品和出价：

```
from Item i
    left join i.bids b
        with b.amount > 100
        where i.description like '%Foo%'
```

这个语句中的第一个新事物是LEFT关键字。可以选择编写LEFT OUTER JOIN和RIGHT OUTER JOIN，但是我们通常更喜欢简短的形式。第二个变化是WITH关键字后面的额外联结条件。如果把b.amount > 100表达式放进WHERE子句中，你就可以限制结果为有出价的Item实例。但这并不是你在此处想要的：你想要获取货品和出价，甚至没有出价的货品。通过在FROM子句中添加额外的联结条件，可以限制Bid实例并且仍然获取所有的Item对象。这个查询再次返回有序的Item和Bid对象对。最后，注意包含WITH关键字的额外联结条件只在HQL中可用；JPA QL只支持由被映射的外键关联所表示的基本的外部联结条件。

外部联结在其中起着重要作用的一种更常见的场景是即时动态抓取。

4．利用联结的动态抓取策略

你在前一节见过的所有查询都有一个共同点：返回的Item实例都有一个名为bids的集合。如果映射为lazy="true"（默认），这个集合就没有被初始化，一旦你访问它，就立即触发一个额外的SQL语句。对于所有单端的关联也是一样，就像每个Item的seller。默认情况下，Hibernate生成一个代理，并只按需延迟加载被关联的User实例。

有哪些选项可以改变这个行为呢？首先，可以在映射元数据中改变抓取计划，并声明一个集合或者单值的关联为lazy="false"。然后，Hibernate执行必要的SQL，保证始终加载想要的对象网络。这也意味着单个HQL或者JPA QL语句可能产生几个SQL操作！

另一方面，通常不在映射元数据中修改抓取计划，除非绝对确定它应该全局应用。你通常给一个特定的用例编写新的抓取计划。这也是你通过编写HQL和JPA QL语句所已经做过的事；通过选择、限制和投影定义了一个抓取计划。使它更有效的唯一东西是正确的动态抓取策略（dynamic fetching strategy）。例如，你没必要用几个SQL语句来抓取所有的Item实例并初始化它们的bids集合，或者给每个Item获取seller。这可以通过一个联结操作同时完成。

在HQL和JPA QL中，可以指定被关联的实体实例或者集合应该在FROM子句中通过FETCH关键字即时抓取：

```
from Item i
    left join fetch i.bids
        where i.description like '%Foo%'
```

这个查询返回描述中包含字符串“Foo”的所有货品，以及它们在单个SQL操作中的所有出价集合。执行时，它返回Item实例的一个列表，其中它们的bids集合已被完全初始化。如果把它与前一节中的由查询返回的有序对进行对比，就会发现它完全不同！

抓取联结的目的在于性能优化：你使用这个语法，只是因为想要在单个SQL操作中即时初始化bids集合：

```
select i.DESCRIPTION, i.INITIAL_PRICE, ...
       b.BID_ID, b.AMOUNT, b.ITEM_ID, b.CREATED_ON
from ITEM i
left outer join BID b on i.ITEM_ID = b.ITEM_ID
where i.DESCRIPTION like '%Foo%'
```

额外的WITH子句在这里没有太大的意义。无法限制Bid实例：所有的集合都必须被初始化。也可以利用相同的语法预抓取多对一或者一对一关联：

```
from Bid bid
    left join fetch bid.item
    left join fetch bid.bidder
        where bid.amount > 100
```

这个查询执行下列SQL：

```
select b.BID_ID, b.AMOUNT, b.ITEM_ID, b.CREATED_ON
       i.DESCRIPTION, i.INITIAL_PRICE, ...
       u.USERNAME, u.FIRSTNAME, u.LASTNAME, ...
from BID b
left outer join ITEM i on i.ITEM_ID = b.ITEM_ID
left outer join USER u on u.USER_ID = b.BIDDER_ID
where b.AMOUNT > 100
```

如果你编写JOIN FETCH，没用LEFT，那么就通过一个内部联结获得了即时加载（如果使用INNER JOIN FETCH也一样）；例如，利用内部联结的预抓取，返回其bids集合完全初始化的Item对象，不返回没有出价的Item对象。这样的查询很少用于集合，但是可以用于不可为空的多对一关联。例如，join fetch item.seller的效果就很好。

HQL和JPA QL中的动态抓取很简单，但是你应该牢记下列警告：

- 永远不要为任何被抓取联结的关联或者集合分配别名，以便进一步限制或者投影。因此left join fetch i.bids b where b =...是无效的，而left join fetch i.bids b join fetch b.bidder却是有效的。
- 不应该并行抓取一个以上的集合；否则就产生了笛卡儿积。你喜欢抓取几个单值的关联对象就可以抓取几个，而不会产生乘积。这基本上与我们在13.2.5节中讨论过的问题一样。
- HQL和JPA QL忽略你已经在映射元数据中定义的任何抓取策略。例如，在XML中通过fetch="join"映射bids集合，对任何HQL或者JPA QL语句都没有影响。动态的抓取策略忽略全局的抓取策略（另一方面，不会忽略全局的抓取计划——保证加载每一个非延迟的关联或者集合，即使需要几个SQL查询）。
- 如果即时抓取一个集合，则可能返回重复。看看图14-3：这就是给select i from Item i join fetch i.bids HQL或者JPA QL查询执行的那个SQL操作。每个Item都在结果表的左侧重复，重复次数与相关的Bid数据出现的次数一样。HQL或者JPA QL查询返回的

List把这些重复保存为引用。如果喜欢过滤掉这些重复，需要把List包装在一个Set中（例如，用Set noDupes =new LinkedHashSet(resultList)，或者利用DISTINCT关键字：select distinct i from Item i join fetch i.bids——注意，在这个例子中，DISTINCT不在SQL级别上操作，而是在把结果封送到对象的时候，强制Hibernate过滤掉内存中的重复。很显然，重复在SQL结果中不可避免。

- 基于SQL结果行的查询执行选项（例如利用setMaxResults()/setFirstResult()的分页），如果集合被即时抓取，那么它在语义上就是错误的。如果查询中有一个即时抓取到的集合，在编写本书之时，Hibernate会退回到在内存中限制结果，而不是使用SQL。这样可能会降低效率，因此不建议通过setMaxResults()/setFirstResult()使用JOIN FETCH。如果setMaxResults()/setFirstResult()与JOIN FETCH组合使用，Hibernate的未来版本可能退回到一个不同的SQL查询策略（例如两个查询和子查询抓取）。

Hibernate就是这样实现动态关联抓取的，对于在任何应用程序中实现高性能而言，这是一个基本的强大特性。如13.2.5节中所述，通过查询调优抓取计划和抓取策略是你的第一优化，当越来越多的查询明显会有相同的必要条件时，第二优化便是映射元数据中的全局设置。

最后一个联结选项是theta类型的联结。

5. Theta类型的联结

乘积让你获取两个或者多个类的实例的所有可能的组合。这个查询返回所有有序的User和Category对象对：

```
from User, Category
```

很显然，这通常没用。它常用于一个案例：theta类型的联结。

在传统的SQL中，theta类型的联结是一个笛卡儿积和WHERE子句中的一个联结条件，在乘积上应用它用来限制结果。

在HQL和JPA QL中，当联结条件不是一个被映射到类关联的外键关系时，theta类型的语法就很有用。例如，假设你在日志记录中保存了User的名称，而不是映射一个从LogRecord到User的关联。类与类相互之间不知道对方的任何信息，因为它们没有被关联。然后可以通过下列theta类型的联结，找到所有User和它们的LogRecord：

```
from User user, LogRecord log where user.username = log.username
```

此处的联结条件是username的一个比较，在两个类中都以属性出现。如果两个类都有相同的username，就在结果中被联结（用一个内部联结）。这个查询结果由有序的对组成：

```
Iterator i =
    session.createQuery("from User user, LogRecord log" +
                        " where user.username = log.username")
           .list().iterator ();

while ( i.hasNext() ) {
    Object[] pair = (Object[]) i.next();
    User user = (User) pair[0];
    LogRecord log = (LogRecord) pair[1];
}
```

当然也可以应用SELECT子句，只投影你关注的数据。

可以不需要经常使用theta类型的联结。注意，在HQL或者JPA QL中对两张没有映射关联的表使用外部联结（theta类型的联结是内部联结）目前是不可能的。

最后，执行把主键（或者外键值）与查询参数（或者其他主或外键值）进行比较的查询，这极为常见。

6. 比较标识符

如果以更面向对象的方式考虑标识符比较，实际上就是正在比较对象引用。HQL和JPA QL支持下列查询：

```
from Item i, User u
    where i.seller = u and u.username = 'steve'
```

在这个查询中，i.seller指向ITEM表中USER表的外键（在SELLER_ID列上），并且user指向USER表的主键（在USER_ID列上）。这个查询使用一个theta类型的联结，相当于下面的首选：

```
from Item i join i.seller u
    where u.username = 'steve'
```

另一方面，下列theta类型的联结无法被重新表达为FROM子句联结：

```
from Item i, Bid b
    where i.seller = b.bidder
```

在这个例子中，i.seller和b.bidder是USER表的两个外键。注意这是应用程序中一个重要的查询，用它辨别是谁给他们自己的货品出了价。

你也可能想要把外键值与查询参数进行比较，以便从User中找到所有的Comment：

```
User givenUser = ...
Query q = session.createQuery(
                  "from Comment c where c.fromUser = :user"
                 );
q.setEntity("user", givenUser);
List result = q.list();
```

另一种偶尔会使用的方法是，你喜欢根据标识符值而不是对象引用来表达这类查询。标识符值可能被标识符属性名（如果有）或者特殊的属性名id引用。（注意，只有HQL保证id始终引用任何被任意命名的标识符属性，JPA QL则不保证。）

这些查询相当于之前的查询：

```
from Item i, User u
    where i.seller.id = u.id and u.username = 'steve'

from Item i, Bid b
    where i.seller.id = b.bidder.id
```

然而，现在可能把标识符值用作查询参数：

```
Long userId = ...
Query q = session.createQuery(
                  "from Comment c where c.fromUser.id = :userId"
                  );
q.setLong("userId", userId);
List result = q.list();
```

考虑标识符属性，在下列查询之间有着很大的区别：

```
from Bid b where b.item.id = 1

from Bid b where b.item.description like '%Foo%'
```

第二个查询使用了隐式的表联结，第一个根本没有联结！

这样就结束了我们对涉及联结的查询的讨论。你学习了如何用点号编写简单的隐式内部联结，以及如何用别名在FROM子句中编写显式的内部或者外部联结。我们也探讨了利用外部和内部联结SQL操作的动态抓取。

下一个话题是高级查询，我们认为它们主要用于报表。

14.3.2　报表查询

报表查询利用数据库执行有效的数据分组和聚集的能力。它们天生更为相关；它们并非始终返回实体。例如，不是获取处于持久化状态（和自动进行脏检查）的Item实体，报表查询可能只获取Item名称和初始的拍卖价格。如果这是你的报表屏幕所需的唯一信息(可能甚至是聚集过的、一个类别中的最高初始价格等)，就不需要事务实体实例，并且可以节省在持久化上下文中自动脏检查和高速缓存的过载。

HQL和JPA QL允许使用几个最常用于报表的SQL特性——虽然它们也用于其他目的。在报表查询中，给投影使用SELECT子句，给聚集使用GROUP BY和HAVING子句。

由于已经讨论过基本的SELECT子句，我们将直接跳到聚集和分组。

1. 利用统计函数投影

HQL识别的并在JPA QL中标准化的统计函数是count()、min()、max()、sum()和avg()。

这个查询统计所有的Item：

```
select count(i) from Item i
```

结果作为Long返回：

```
Long count =
    (Long) session.createQuery("select count(i) from Item i")
                  .uniqueResult();
```

查询的下一种变形统计了所有具有successfulBid（排除空值）的Item：

```
select count(i.successfulBid) from Item i
```

这个查询计算所有成功Bid的总数：

```
select sum(i.successfulBid.amount) from Item i
```

查询返回BigDecimal，因为amount属性是BigDecimal类型。SUM()函数也识别BigInteger属性类型，并给所有其他的数字属性类型返回Long。注意SELECT子句中隐式联结的使用：用一个点引用它，来导航从Item到Bid的关联（successfulBid)。

下一个查询给特定的Item返回最低和最高的出价金额：

```
select min(bid.amount), max(bid.amount)
        from Bid bid where bid.item.id = 1
```

结果是一个有序的BigDecimal对（BigDecimal的两个实例，在一个Object[]数组中）。

特殊的COUNT(DISTINCT)函数忽略重复：

```
select count(distinct i.description) from Item i
```

在SELECT子句中调用统计函数时，在GROUP BY子句中没有指定任何分组的情况下，就会把结果压缩到单行，包含统计好的（几个）值。这意味着（缺少GROUP BY子句）任何包含统计函数的SELECT子句都必须只包含统计函数。

对于更高级的统计和报表，需要能够执行分组。

2. 给统计结果分组

就像在SQL中一样，出现在HQL或者JPA QL中但在SELECT子句中的一个统计函数之外的任何属性或者别名，也必须出现在GROUP BY子句中。考虑下一个查询，它统计带有姓（last name）的用户数量：

```
select u.lastname, count(u) from User u
group by u.lastname
```

看看生成的SQL：

```
select u.LAST_NAME, count(u.USER_ID)
from USER u
group by u.LAST_NAME
```

在这个例子中，u.lastname不在统计函数之内，用它给结果分组。也不需要指定想要统计的属性。如果使用一个已经在FROM子句中设置的别名，生成的SQL就会自动利用主键。

下一个查询给每个货品找出平均出价金额：

```
select bid.item.id, avg(bid.amount) from Bid bid
group by bid.item.id
```

这个查询返回有序的Item标识符和平均出价金额值对。注意如何使用id这个特殊属性来指向一个持久化类的标识符，无论标识符的真实属性名称是什么。（同样，这个特殊属性在JPA QL中没有被标准化。）

下一个查询统计出价的数目，并按未售货品计算平均出价：

```
select bid.item.id, count(bid), avg(bid.amount)
from Bid bid
where bid.item.successfulBid is null
group by bid.item.id
```

该查询使用了隐式的关联联结。对于FROM子句中的显式普通联结（不是抓取联结），可以把它重新表达如下：

```
select bidItem.id, count(bid), avg(bid.amount)
from Bid bid
    join bid.item bidItem
where bidItem.successfulBid is null
group by bidItem.id
```

有时候，想要通过只选择一个分组的特定值，来进一步限制结果。

3. 用having限制分组

WHERE子句用来在行上执行限制的相关操作。HAVING子句在分组上执行限制。

例如，下一个查询统计每一个姓中以“A”开头的用户：

```
select user.lastname, count(user)
from User user
group by user.lastname
    having user.lastname like 'A%'
```

管理SELECT和HAVING子句的规则一样：只有被分了组的属性才可能出现在统计函数之外。下一个查询按未售货品统计出价的数目，只给超过10个出价的那些货品返回结果：

```
select item.id, count(bid), avg(bid.amount)
from Item item
    join item.bids bid
where item.successfulBid is null
group by item.id
    having count(bid) > 10
```

大多数报表查询都使用SELECT子句来选择一系列被投影或者统计的属性。你已经见过，当SELECT子句中列出不止一个属性或者别名时，Hibernate就返回查询结果为字节组——查询结果列表的每一行都是Object[]的一个实例。

4. 利用动态的实例化

一般用于报表查询的，字节组（tuple）特别不方便，因此HQL和JPA QL提供了SELECT NEW构造器调用。除了用这种技术动态创建新对象之外，也可以把它与聚集和分组组合起来使用。

如果用含有Long、Long和BigDecimal的构造器，定义一个称作ItemBidSummary的类，就可能用到下列查询：

```
select new ItemBidSummary(
           bid.item.id, count(bid), avg(bid.amount)
           )
from Bid bid
where bid.item.successfulBid is null
group by bid.item.id
```

在这个查询的结果中，每个元素都是ItemBidSummary的一个实例，它是一个Item的概括，包括对这件货品出价的数目以及平均出价金额。注意，此处必须编写一个完全限定的类名，带有包名，除非类已经被导入到HQL命名空间（请见4.3.3节）。这个方法是类型安全的，并且数据迁移类（如ItemBidSummary）可以轻松地对报表中值的特殊格式打印进行扩展。

ItemBidSummary类是一个Java bean，它不一定是被映射的持久化实体类。另一方面，如果通过被映射的实体类使用SELECT NEW技术，查询返回的所有实例便都处于瞬时状态——因此可以利用这个特性填充几个新对象，然后保存它们。

报表查询可能影响应用程序的性能。我们来更深入地探讨这个问题。

5. 利用报表查询提升性能

我们只在Hibernate代码中见过明显的过载（与直接JDBC查询对比）——然后只对非现实的简单测试案例——是在针对本地数据库的只读查询的特殊案例中。在这个例子中，数据库可能在

内存中完全高速缓存查询结果，并迅速响应，因此如果数据集很小，基准一般没用：简单的SQL和JDBC永远是最快的方法。

另一方面，Hibernate甚至通过一个很小的数据集，仍然必须完成把查询结果对象添加到持久化上下文高速缓存（可能二级高速缓存也要）、管理唯一性等工作。如果你曾希望避免管理持久化上下文高速缓存的过载，报表查询就为你提供了一种解决办法。Hibernate报表查询的过载，比起直接的SQL/JDBC通常是不可测量的，即使在非现实的极端案例中，就像从一个本地数据库没有网络延迟地加载一百万个对象一样。

在HQL和JPA QL中使用投影的报表查询，让你指定希望获取哪些属性。对于报表查询，不是选择托管状态下的实体，而只是选择属性或者统计值：

```
select user.lastname, count(user) from User user
group by user.lastname
```

这个查询不返回持久化的实体实例，因此Hibernate不把任何持久化对象添加到持久化上下文高速缓存。这也意味着不必关注处于脏状态的对象。

因此，报表查询导致被分配内存的更快释放，因为对象没有被保存在持久化上下文高速缓存中，直到上下文关闭——执行报表之后，一旦它们没有被应用程序引用，可能就立即被当作垃圾收集起来。

通常情况下，这些考虑极为次要，因此不必去重写所有的只读事务来使用报表查询而不是事务的、被高速缓存的和托管的对象。报表查询更为冗长并且（有争议地）更不面向对象。它们使用Hibernate高速缓存的效率也更为低下，一旦考虑到在产品系统中与数据库进行远程通信的过载，这就更加重要了。你应该等着找到一个在使用这个优化之前就真正有性能问题的案例。

利用目前已经学过的东西，已经可以创建真正复杂的HQL和JPA QL查询了。甚至更高级的查询可能包括嵌套的语句，即众所周知的子查询（subselect）。

14.3.3 利用子查询

SQL一项重要且强大的特性就是子查询。子查询是内嵌在另一个查询中的查询，一般在SELECT、FROM或者WHERE子句中。

HQL和JPA QL支持WHERE中的子查询。FROM子句中的子查询不受HQL和JPA QL的支持（虽然规范把它们列为一项可能的未来扩展），因为这两种语言都没有传递闭包。查询的结果可能不是表格，因此不能在FROM子句中重新用于选择。SELECT子句中的子查询也不受查询语言的支持，但可以通过公式（formula）映射到属性，如8.1.3节中所述。

（Hibernate支持的有些平台不实现SQL子查询。Hibernate只在SQL数据库管理系统提供这个特性时才支持子查询。）

1. 相关的和不相关的嵌套

子查询的结果可能包含单行或者多行。典型地，返回单行的子查询执行聚集。下列子查询返回被一位用户出售的货品的总数量，外部查询返回出售超过10件货品的所有用户：

```
from User u where 10 < (
    select count(i) from u.items i where i.successfulBid is not null
)
```

这是一个相关子查询——它从外部查询引用一个别名(u)。下一个子查询是一个不相关子查询：

```
from Bid bid where bid.amount + 1 >= (
    select max(b.amount) from Bid b
)
```

这个示例中的子查询返回整个系统中最高的出价金额；外部查询返回金额在那个金额一块（美元）之内的所有出价。

注意在这两个案例中，子查询都以圆括号关闭。这通常是必需的。

不相关子查询没有害处，并且方便的时候没理由不用它们，虽然它们通常可能被重写为两个查询（它们不互相引用）。你应该更慎重地考虑相关子查询对性能的影响。在一个成熟的数据库中，一个简单的关联子查询的性能成本类似于一个联结的成本。但是，不一定要用几个单独的查询重写一个相关子查询。

2. 量词

如果一个子查询返回多行，它就是与量词（quantification）组合了。ANSI SQL、HQL和JPA QL定义了下列量词：

- ALL——如果对子查询的结果中所有值的比较为真，表达式就取值为true。如果子查询结果的单个值没有通过比较测试，就取值为false。
- ANY——如果对子查询的结果中有些（任何）值的比较为真，表达式就取值为true。如果子查询结果为空或者没有值满足比较，就取值为false。关键字SOME是ANY的同义词。
- IN——这个二进制的比较操作符可以针对一个子查询的结果比较一系列值，并且如果在结果中找到所有的值，则取值为true。

例如，这个查询返回所有出价都小于100的货品：

```
from Item i where 100 > all ( select b.amount from i.bids b )
```

下一个查询返回所有其他的货品，即出价大于100的：

```
from Item i where 100 <= any ( select b.amount from i.bids b )
```

这个查询返回出价正好为100的货品：

```
from Item i where 100 = some ( select b.amount from i.bids b )
```

这个也一样：

```
from Item i where 100 in ( select b.amount from i.bids b )
```

HQL对在元素或者集合的索引上操作的子查询支持快捷语法。下列查询使用特殊的HQL elements()函数：

```
List result =
    session.createQuery("from Category c" +
                        " where :givenItem in elements(c.items)")
           .setEntity("givenItem", item)
           .list()
```

这个查询返回这件货品所属的所有分类，并相当于下列HQL（和有效的JPA QL），其中子查询更显式：

```
List result =
 session.createQuery(
"from Category c where :givenItem in (select i from c.items i)"
)
.setEntity("item", item)
.list();
```

与elements()一起，HQL还提供indices()、maxelement()、minelement()、maxindex()、minindex()和size()，它们中每一个都相当于针对已被传递的集合的某个相关子查询。参考Hibernate文档，了解更多有关这些特殊函数的信息；它们很少被使用。

子查询是一种高级的技术；你应该质疑子查询的频繁使用，因为带有子查询的查询经常可以只用联结和统计来重写。但是它们有时候很强大并且有用。

14.4 小结

你现在能够在HQL和JPA QL中编写各种各样的查询了。本章介绍了如何准备和执行查询，以及如何绑定参数。我们已经介绍了限制、投影、联结、子查询和许多你可能已经从SQL中知道的其他选项。

表14-4展现了可以用来比较原生的Hibernate特性和Java Persistence的一个概括。

表14-4 第14章中的Hibernate和JPA对照表

Hibernate Core	Java Persistence和EJB 3.0
Hibernate API通过列表、迭代和滚动支持查询执行	Java Persistence标准化了利用列表的查询执行
Hibernate支持具名的和定位的查询绑定参数	Java Persistence标准化了具名的和定位的绑定参数选项
Hibernate查询API支持应用级的查询提示	Java Persistence允许开发人员提供任意的特定于供应商的（Hibernate）查询提示
HQL支持像SQL的限制、投影、联结、子查询和函数调用	JPA QL支持像SQL的限制、投影、联结、子查询和函数调用——HQL的子集

第15章将关注更高级的查询技术，例如利用Criteria API编程式地生成复杂查询，以及原生SQL查询的嵌入。我们也将讨论查询高速缓存，以及什么时候应该启用它。

第 15 章

高级查询选项

本章内容

- 利用Criteria和Example API查询
- 嵌入原生的SQL查询
- 集合过滤器
- 可选的查询结果高速缓存

本章阐述你可能认为是可选的或者高级的所有查询选项。每当你通过编程创建更复杂的查询时，将需要本章的第一个主题：Criteria查询接口。这个API比给HQL和JPA QL通过编程生成的查询字符串更方便且更优雅。不幸的是，它也只可作为一个原生的Hibernate API使用，Java Persistence（还）没有标准化编程式的查询接口。

Hibernate和Java Persistence都支持用原生的SQL编写的查询。你可以在Java源代码中嵌入SQL和存储过程调用，或者把它们外部化到映射元数据。Hibernate可以执行你的SQL，并把结果集转变为更方便的对象，这取决于你的映射。

集合的过滤是Hibernate的一项简单便利特性——你不会经常用它。例如，如果想要在一个集合中获得对象的一个子集，它就会帮助你用一个简单的API调用和查询片段，取代更繁杂的查询。

最后，将讨论可选的查询结果高速缓存——我们已经说过，它并不是在所有的情况下都有用，因此将更深入地讨论一下高速缓存查询结果的好处，以及什么时候完美地启用这项特性。

从按条件查询和按示例查询开始。

15.1 利用条件和示例查询

Criteria和Example API都只可在Hibernate中使用；Java Persistence没有标准化这些接口。如前所述，似乎其他的供应商（不仅仅Hibernate）也支持一个类似的扩展接口，并且标准的未来版本将包括这项功能。

当查询变得更加复杂时，带有通过编程生成的条件和示例对象的查询经常是首选的解决方案。如果你必须在运行时创建查询，这尤其正确。想象你必须在应用程序中实现一个搜索掩码，利用用户可以启用的许多复选框、输入字段和开关。必须从用户的选择中创建一个数据库查询。

完成这项工作的传统方法是通过串联创建一个查询字符串，或者可能编写一个查询构建器（query builder），它可以为你构建SQL查询字符串。如果试图在这个场景中使用HQL或者JPA QL，也会遇到同样的问题。

Criteria和Example接口允许通过以正确的顺序创建和组合对象来编程式地构建查询。现在介绍如何利用这些API，以及如何表达选择、限制、联结和投影。假设你已经阅读了前一章，并且知道这些操作如何转变成SQL。即使你决定使用Criteria和Example API作为编写查询的主要方法，也要记住，由于HQL和JPA QL本来就是基于字符串的，因此始终更加灵活。

从一些基本的选择和限制示例开始吧。

15.1.1 基本的条件查询

最简单的条件查询看起来像这样：

```
session.createCriteria(Item.class);
```

它获取Item类的所有持久化实例。这也称作条件查询的根实体（root entity）。

条件查询也支持多态：

```
session.createCriteria(BillingDetails.class);
```

这个查询返回BillingDetails及其子类的实例。同样地，下列条件查询返回所有持久化对象：

```
session.createCriteria(java.lang.Object.class);
```

Criteria接口也支持通过addOrder()方法和Order条件给结果排序：

```
session.createCriteria(User.class)
        .addOrder( Order.asc("lastname") )
        .addOrder( Order.asc("firstname") );
```

你不必用开着的Session创建条件对象，DetachedCriteria可以被实例化，之后附加到Session用于执行（或者附加到另一个Criteria作为子查询）：

```
DetachedCriteria crit =
 DetachedCriteria.forClass(User.class)
             .addOrder( Order.asc("lastname") )
             .addOrder( Order.asc("firstname") );

List result = crit.getExecutableCriteria(session).list();
```

通常，你想要限制结果，并且不获取类的所有实例。

1. 应用限制

对于条件查询来说，你必须构建Criterion对象来表达约束。Restrictions类给内建的Criterion类型提供工厂方法。我们来用特定的电子邮件地址搜索User对象：

```
Criterion emailEq = Restrictions.eq("email", "foo@hibernate.org");
Criteria crit = session.createCriteria(User.class);
crit.add(emailEq);
User user = (User) crit.uniqueResult();
```

你创建了Criterion，表示对一个同一性比较的限制，并把它添加到Criteria。这个eq()方法有两个参数：第一个是属性名，然后是应该被比较的值。属性名始终作为一个字符串给定，

记住，这个名称在领域模型重构期间可以改变，并且你必须手工更新任何预设的条件查询。还要注意，条件接口不支持显式的参数绑定，因为没有必要。在前一个例子中，你绑定了字符串"foo@hibernate.org"到查询中。你可以绑定任何java.lang.Object，并让Hibernate确定如何处理它。uniqueResult()方法执行查询，并正好返回一个对象作为结果——你必须正确地转换（cast）它。

利用方法链编写这个通常不会太冗长：

```
User user =
  (User) session.createCriteria(User.class)
                .add(Restrictions.eq("email", "foo@hibernate.org"))
                .uniqueResult();
```

很显然，如果条件查询变得更加复杂时会更难以阅读——这是首选它们用于动态和编程查询生成的一个很好的理由，而给预设的查询使用外部化的HQL和JPA QL。JDK 5.0的一项新特性是静态导入（static import），它有助于使条件查询更易于阅读。例如，通过添加：

```
import static org.hibernate.criterion.Restrictions.*;
```

可以将这个条件查询限制代码缩写为：

```
User user =
  (User) session.createCriteria(User.class)
                .add( eq("email", "foo@hibernate.org") )
                .uniqueResult();
```

获得Criterion的另一种可选方法是Property对象——本节稍后讨论投影时，它将会更有用：

```
session.createCriteria(User.class)
        .add( Property.forName("email").eq("foo@hibernate.org") );
```

也可以用一般的点号命名一个组件的属性：

```
session.createCriteria(User.class)
        .add( Restrictions.eq("homeAddress.street", "Foo"));
```

Criteria API和org.hibernate.criterion包除了提供eq()之外，还提供许多其他的操作符，可以用来构造更复杂的表达式。

2. 创建比较表达式

所有一般的SQL（和HQL、JPA QL）比较操作符也可以通过Restrictions类使用：

```
Criterion restriction =
 Restrictions.between("amount",
                new BigDecimal(100),
                new BigDecimal(200) );
session.createCriteria(Bid.class).add(restriction);

session.createCriteria(Bid.class)
        .add( Restrictions.gt("amount", new BigDecimal(100) ) );

String[] emails = { "foo@hibernate.org", "bar@hibernate.org" };
session.createCriteria(User.class)
        .add( Restrictions.in("email", emails) );
```

三重逻辑操作符也可用；这个查询返回没有电子邮件地址的所有用户：

```
session.createCriteria(User.class)
        .add( Restrictions.isNull("email") );
```

还需要能够找到的确有电子邮件地址的用户：

```
session.createCriteria(User.class)
        .add( Restrictions.isNotNull("email") );
```

也可以用isEmpty()、isNotEmpty()或者它的实际大小测试集合：

```
session.createCriteria(Item.class)
        .add( Restrictions.isEmpty("bids"));

session.createCriteria(Item.class)
        .add( Restrictions.sizeGt("bids", 3));
```

或者可以比较两个属性：

```
session.createCriteria(User.class)
        .add( Restrictions.eqProperty("firstname", "username") );
```

条件查询接口对字符串匹配也有特殊的支持。

3. 字符串匹配

对于条件查询，通配符搜索可以使用与HQL和JAP QL相同的通配符符号（%和_），或者指定MatchMode。MatchMode是不用字符串操作来表达子字符串匹配的一种方便的方法。这两个查询是一样的：

```
session.createCriteria(User.class)
        .add( Restrictions.like("username", "G%") );

session.createCriteria(User.class)
        .add( Restrictions.like("username", "G", MatchMode.START) );
```

得到支持的MatchMode是START、END、ANYWHERE和EXACT。

你还经常想要执行不区分大小写的字符串匹配。求助于函数的时候，比如HQL或者JPA QL中的LOWRE()，就可以依赖Criteria API的一种方法：

```
session.createCriteria(User.class)
        .add( Restrictions.eq("username", "foo").ignoreCase() );
```

可以把表达式与逻辑操作符组合起来。

4. 组合表达式和逻辑操作符

如果把多个Criterion实例添加到同一个Criteria实例，它们被联结起来应用（利用and）：

```
session.createCriteria(User.class)
        .add( Restrictions.like("firstname", "G%") )
        .add( Restrictions.like("lastname", "K%") );
```

如果需要分开（or），有两种方法。第一种是把Restrictions.or()和Restrictions.and()一起使用：

```
session.createCriteria(User.class)
        .add(
            Restrictions.or(
                Restrictions.and(
```

```
                    Restrictions.like("firstname", "G%"),
                    Restrictions.like("lastname", "K%")
                ),
                Restrictions.in("email", emails)
            )
        );
```

第二种方法是把Restrictions.disjunction()和Restrictions.conjunction()一起使用：

```
session.createCriteria(User.class)
        .add( Restrictions.disjunction()
            .add( Restrictions.conjunction()
                .add( Restrictions.like("firstname", "G%") )
                .add( Restrictions.like("lastname", "K%") )
            )
            .add( Restrictions.in("email", emails) )
        );
```

我们认为这两种方法都很难懂，即使花上5分钟努力给它们调整格式以达到最大限度的可读性。JDK 5.0静态导入可以帮助明显改善可读性，但是即使如此，除非你正在凭空构建查询，否则HQL或者JPA QL字符串更容易理解。

你可能已经注意到许多标准的比较操作符（<、>、=等）都内建在Criteria API中，但是漏了某些操作符。例如，任何算术操作符（如加和减）都没有直接得到支持。

另一个问题是函数调用。Criteria只给最常见的案例（如字符串区分大小写）匹配内建函数。另一方面，HQL允许你在WHERE子句中调用任意的SQL函数。

Criteria API有着类似的工具：可以添加任意的SQL表达式作为Criterion。

5. 添加任意的SQL表达式

假设你想要测试一个字符串的长度，并相应地限制查询结果。Criteria API没有相当于SQL、HQL或者JPA QL中的LENGTH()函数。

但是，可以把一个简单的SQL函数表达式添加到你的Criteria：

```
session.createCriteria(User.class)
    .add( Restrictions.sqlRestriction(
            "length({alias}.PASSWORD) < ?",
            5,
            Hibernate.INTEGER
           )
    );
```

这个查询返回密码少于5个字符的所有User对象。最终的SQL中任何表别名都要加上{alias}占位符作为前缀；它总是指向根实体所映射到的表（这个例子里是USERS）。你还用了一个位置参数（这个API不支持具名参数），并指定它的类型为Hibernate.INTEGER。不用单个的绑定参数和类型，而是还可以使用支持参数和类型数组的sqlRestriction()方法的重载版本。

这个工具很强大——例如，可以用量词添加一个SQL WHERE子句子查询：

```
session.createCriteria(Item.class)
    .add( Restrictions.sqlRestriction(
            "'100' > all" +
            " ( select b.AMOUNT from BID b" +
```

```
        "    where b.ITEM_ID = {alias}.ITEM_ID )"
      )
);
```

这个查询返回出价不大于100的所有Item对象。

（Hibernate条件查询系统是可扩展的：也可以把LENGTH() SQL函数包装在自己的Criterion接口的实现中。）

最后，可以编写包括子查询的条件查询。

6. 编写子查询

条件查询中的子查询是一个WHERE子句查询。就像在HQL、JPA QL和SQL中一样，子查询的结果可能包含单行或者多行。通常情况下，返回单行的子查询执行统计。

下列子查询返回某位用户出售的货品总量；外部查询返回出售货品超过10件的所有用户：

```
DetachedCriteria subquery =
                    DetachedCriteria.forClass(Item.class, "i");

subquery.add( Restrictions.eqProperty("i.seller.id", "u.id"))
        .add( Restrictions.isNotNull("i.successfulBid") )
        .setProjection( Property.forName("i.id").count() );

Criteria criteria = session.createCriteria(User.class, "u")
                           .add( Subqueries.lt(10, subquery) );
```

这是一个相关子查询。DetachedCriteria指向别名u；这个别名在外部查询中声明。注意，外部查询使用了小于操作符，因为子查询是右操作数。还要注意i.seller.id没有生成联结，因为SELLER_ID是ITEM表中的一个列，它是脱管条件的根实体。

让我们进入下一个有关条件查询的主题：联结和动态抓取。

15.1.2 联结和动态抓取

就像在HQL和JPA QL中一样，对于你为什么要表达联结可能着不同的理由。首先，你可能想要用一个联结通过被联结类的某些属性来限制结果。例如，你可能想要获取某位特定的用户出售的所有Item实例。

当然，你也可能想要用联结来动态抓取被关联的对象或者集合，就像在HQL和JPA QL中使用fetch关键字一样。在条件查询中，也可以通过FetchMode使用相同的选项。

先来看看普通联结，以及如何表达涉及被关联类的限制。

1. 给限制联结关联

在Criteria API中表达联结有两种方法；因此你有两种给限制使用别名的方法。第一种是Criteria接口的createCriteria()方法。这个方法一般意味着你可以嵌套对createCriteria()的调用：

```
Criteria itemCriteria = session.createCriteria(Item.class);
itemCriteria.add(
    Restrictions.like("description",
                      "Foo",
                      MatchMode.ANYWHERE)
);
```

```
Criteria bidCriteria = itemCriteria.createCriteria("bids");
bidCriteria.add( Restrictions.gt( "amount", new BigDecimal(99) ) );

List result = itemCriteria.list();
```

通常编写查询如下（方法链）：

```
List result =
  session.createCriteria(Item.class)
   .add( Restrictions.like("description",
                           "Foo",
                           MatchMode.ANYWHERE)
   )
   .createCriteria("bids")
    .add( Restrictions.gt("amount", new BigDecimal(99) ) )
   .list();
```

给Item的bids创建Criteria，导致了两个类的表之间的内部联结。注意，你可以在任何一个Criteria实例中调用list()而不改变查询结果。嵌套条件不仅适用于集合（如bids），还可用于单值的关联（如seller）：

```
List result =
  session.createCriteria(Item.class)
   .createCriteria("seller")
    .add( Restrictions.like("email", "%@hibernate.org") )
   .list();
```

这个查询返回具有特定电子邮件地址模式的用户出售的所有货品。

用Criteria API表达内部联结的第二种方法是为被联结的实体分配一个别名：

```
session.createCriteria(Item.class)
        .createAlias("bids", "b")
        .add( Restrictions.like("description", "%Foo%") )
        .add( Restrictions.gt("b.amount", new BigDecimal(99) ) );
```

对于单值关联seller中的限制也一样：

```
session.createCriteria(Item.class)
        .createAlias("seller", "s")
        .add( Restrictions.like("s.email", "%hibernate.org" ) );
```

这种方法不用Criteria的第二实例；它一般与你在HQL/JPA QL语句的FROM子句中编写的别名分配机制一样。然后，被联结实体的属性必须通过在createAlias()方法中分配的别名来限定，如s.email。条件查询（Item）的根实体的属性可以被引用，而不用完全限定的别名，或者利用别名"this"：

```
session.createCriteria(Item.class)
        .createAlias("bids", "b")
        .add( Restrictions.like("this.description", "%Foo%") )
        .add( Restrictions.gt("b.amount", new BigDecimal(99) ) );
```

最后注意，在编写本书之时，只有包含了对实体的引用的被关联实体或者集合的联结（一对多和多对多），在Hibernate中通过Criteria API得到支持。下列示例试图联结组件的一个集合：

```
session.createCriteria(Item.class)
        .createAlias("images", "img")
        .add( Restrictions.gt("img.sizeX", 320 ) );
```

Hibernate失败，并有一个异常告诉你：你想要使用别名的那个属性不表示一个实体关联。我们认为在你读到本书时，这项特性可能已经实现了。

另一种语法也是无效的，但你可能有兴趣一试，它是包含点号的单值关联的一个隐式联结：

```
session.createCriteria(Item.class)
        .add( Restrictions.like("seller.email", "%hibernate.org") );
```

“seller.email”字符串不是一个属性或者组件的属性路径。创建一个别名或者嵌套的Criteria对象来联结这个实体关联。

我们来讨论被关联对象和集合的动态抓取。

2. 通过条件查询动态抓取

在HQL和JPA QL中，你用join fetch操作即时填入一个集合，或者初始化一个被映射为延迟否则将被代理的对象。可以用Criteria API完成同样的事：

```
session.createCriteria(Item.class)
        .setFetchMode("bids", FetchMode.JOIN)
        .add( Restrictions.like("description", "%Foo%") );
```

这个查询返回所有包含特定集合的Item实例，并给每个Item即时加载bids集合。

FetchMode.JOIN通过SQL外部联结启用即时抓取。如果想要使用内部联结（很少用，因为它不返回没有出价的货品），可以强制它：

```
session.createCriteria(Item.class)
        .createAlias("bids", "b", CriteriaSpecification.INNER_JOIN)
        .setFetchMode("b", FetchMode.JOIN)
        .add( Restrictions.like("description", "%Foo%") );
```

也可以预抓取多对一和一对一关联：

```
session.createCriteria(Item.class)
        .setFetchMode("bids", FetchMode.JOIN)
        .setFetchMode("seller", FetchMode.JOIN)
        .add( Restrictions.like("description", "%Foo%") );
```

但是小心。HQL和JPA QL中的警告在这里也同样适用：并行即时抓取不止一个集合（如bids和images），会造成一个SQL笛卡儿积，它或许比两个单独的查询更慢。如果你给集合使用即时抓取，那么给限制分页结果集也在内存中完成。

然而，通过Criteria和FetchMode的动态抓取与HQL和JPA QL中的稍有不同：Criteria查询不会忽略在映射元数据中定义的全局抓取策略。例如，如果利用fetch="join"或者FetchType.EAGER映射bids集合，下列查询就会造成ITEM和BID表的一个外部联结：

```
session.createCriteria(Item.class)
        .add( Restrictions.like("description", "%Foo%") );
```

被返回的Item实例初始化了它们的bids集合，并完全加载。这种情形不会发生在HQL或者JPA QL中，除非你用LEFT JOIN FETCH进行手工查询（或者，当然要映射集合为lazy="false"，这样造成第二个SQL查询）。

因而，条件查询可能返回对根实体的独特实例的重复引用，即使你没有在查询中给集合应用FetchMode.JOIN。最后一个查询实例可能返回几百个Item引用，即使你在数据库中只有十几个。

记住我们在14.3.1节中的讨论，并再看一下图14-3中的SQL语句和结果集。

可以在结果List中移除重复的引用，通过把它包装在LinkedHashSet中（常规的HashSet不保存查询结果的顺序）。在HQL和JPA QL中，也可以使用DISTINCT关键字；然而，在Criteria中，没有它的直接等价物。这就是ResultTransformer变得有用的地方。

3．应用结果转换器

结果转换器可以被应用到查询结果，以便你可以用自己的过程而不是Hibernate的默认行为过滤或者封送结果。Hibernate的默认行为是你可以替换和（或）定制的一组默认转换器。

默认情况下，所有的条件查询都只返回根实体的实例：

```
List result = session.createCriteria(Item.class)
                     .setFetchMode("bids", FetchMode.JOIN)
                     .setResultTransformer(Criteria.ROOT_ENTITY)
                     .list();

Set distinctResult = new LinkedHashSet(result);
```

Criteria.ROOT_ENTITY是org.hibernate.transform.ResultTransformer接口的默认实现。前一个查询无论是否使用这个转换器设置都生成相同的结果。它返回所有Item实例并初始化它们的bids集合。List可能（取决于每个Item的Bid数量）包含重复的Item引用。

另一种方法是，可以应用一个不同的转换器：

```
List distinctResult =
    session.createCriteria(Item.class)
           .setFetchMode("bids", FetchMode.JOIN)
           .setResultTransformer(Criteria.DISTINCT_ROOT_ENTITY)
           .list();
```

现在Hibernate在返回结果之前过滤掉重复的根实体引用——如果使用DISTINCT关键字，这实际上与在HQL或者JPA QL中发生的过滤是一样的。

如果想要在联结查询中获取有别名的实体，结果转换器也很有用：

```
Criteria crit =
        session.createCriteria(Item.class)
               .createAlias("bids", "b")
               .createAlias("seller", "s")
               .setResultTransformer(Criteria.ALIAS_TO_ENTITY_MAP);

List result = crit.list();
for (Object aResult : result) {
    Map map = (Map) aResult;
    Item item   = (Item) map.get(Criteria.ROOT_ALIAS);
    Bid  bid    = (Bid)  map.get("b");
    User seller = (User) map.get("s");
    ...
}
```

首先，创建条件查询，它用Item的bids和seller关联来联结Item。这是一个跨三张表的SQL内部联结。在SQL中，这个查询的结果是一张表，在这里每一个结果行都包含货品、出价和用户数据——几乎与图14-2中所示的一样。通过默认的转换器，Hibernate只返回Item实例。并且它通过DISTINCT_ROOT_ENTITY转换器，过滤掉重复的Item引用。似乎没有任何明显的选项——你真

正想要的是在一个映射中返回所有的信息。ALIAS_TO_ENTITY_MAP转换器可以把SQL结果封送到Map实例的一个集合。每个Map都有3个条目：Item、Bid和User。所有的结果数据都得到保存，并且可以在应用程序中被访问。(Criteria.ROOT_ALIAS是"this"的快捷方式。)

这最后一种转换器的好用例很少。注意，也可以实现自己的org.hibernate.transform.ResultTransformer。而且，HQL和原生的SQL查询也支持ResultTransformer：

```
Query q = session.createQuery(
    "select i.id as itemId," +
    "       i.description as desc," +
    "       i.initialPrice as price from Item i");
q.setResultTransformer( Transformers.aliasToBean(ItemDTO.class) );
```

这个查询现在返回ItemDTO实例的一个集合，并且这个bean的属性也通过setItemId()、setDesc()和setPrice()等设置方法被设值。

定义要从查询中返回什么数据的一种更常用的方法是投影。Hibernate条件支持SELECT子句的等价物，用于简单的投影、统计和分组。

15.1.3 投影和报表查询

在HQL、JPA QL和SQL中，编写SELECT子句给一个特定的查询定义投影。Criteria API也支持投影，当然是通过编程而不是基于字符串。可以选择在查询结果中具体需要哪些对象或者对象的哪些属性，以及可能想要如何对报表进行统计和分组结果。

1. 简单的投影列表

下列条件查询只返回仍然处在拍卖中的Item实例的标识符值：

```
session.createCriteria(Item.class)
        .add( Restrictions.gt("endDate", new Date()) )
        .setProjection( Projections.id() );
```

Criteria中的setProjection()方法，接受任何单个的被投影的属性（如在前一个示例中一样），或者接受要包括在结果中的几个属性的一个列表：

```
session.createCriteria(Item.class)
        .setProjection( Projections.projectionList()
            .add( Projections.id() )
            .add( Projections.property("description") )
            .add( Projections.property("initialPrice") )
        );
```

这个查询返回Object[]的一个List，就像HQL或者JPA QL会有一个相当的SELECT子句一样。给投影指定属性的另一种可选的方法是Property类：

```
session.createCriteria(Item.class)
        .setProjection( Projections.projectionList()
            .add( Property.forName("id") )
            .add( Property.forName("description") )
            .add( Property.forName("initialPrice") )
        );
```

在HQL和JPA QL中，可以通过SELECT NEW操作使用动态的实例化，并返回定制对象（而不是Object[]）的一个集合。Hibernate给条件查询捆绑了ResultTransformer，它可以完成几乎相同的工作（事实上，它更加灵活）。下列查询返回与前一个示例一样的结果，但被包在数据迁移对象中：

```
session.createCriteria(Item.class)
        .setProjection( Projections.projectionList()
            .add( Projections.id()
                        .as("itemId") )
    .add( Projections.property("description")
                        .as("itemDescription") )
    .add( Projections.property("initialPrice")
                        .as("itemInitialPrice") )
).setResultTransformer(
    new AliasToBeanResultTransformer(ItemPriceSummary.class)
);
```

ItemPriceSummary是一个简单的Java bean，包含设置方法或者具名itemId、itemDescription和itemInitialPrice的公共字段。它不一定是被映射的持久化类；只有属性/字段名称必须与分配到条件查询中被投影属性的别名相匹配。别名通过as()方法进行分配（可以把它当作SQL SELECT中AS关键字的等价物）。结果转换器调用设置方法或者直接填充字段，并返回ItemPriceSummary对象的一个集合。

我们来用条件进行更复杂的投影，涉及统计和分组。

2. 统计和分组

一般的统计函数和分组选项在条件查询中也可以用。有一种简单的方法统计结果中的行数：

```
session.createCriteria(Item.class)
        .setProjection( Projections.rowCount() );
```

提示　获取分页统计总数——在实际的应用程序中，你经常必须允许用户通过列表进行分页，并同时通知他们列表中总共有多少件货品。获得总数的一种方法是执行rowCount()的Criteria查询。你不用编写这个额外的查询，而是执行与利用scroll()给列表获取数据相同的Criteria。然后调用last()和getRowNumber()来跳到并得到最后一行的行数。这个数字加上一就是你所列的对象数量了。别忘了关闭光标。如果你正在使用现有的DetachedCriteria对象，并且不想重复和操作它的投影来执行rowCount()，这种方法尤其有用。它也适用于HQL或者SQL查询。

更加复杂的统计则使用统计函数。下列查询查找出价的数量，并且平均每位用户所进行的出价总金额：

```
session.createCriteria(Bid.class)
        .createAlias("bidder", "u")
        .setProjection( Projections.projectionList()
```

```
        .add( Property.forName("u.id").group() )
        .add( Property.forName("u.username").group() )
        .add( Property.forName("id").count())
        .add( Property.forName("amount").avg() )
    );
```

这个查询返回包含四个字段的Object[]的一个集合：用户id、登录名、出价数以及平均出价金额。记住，你可以再次给动态的实例化使用结果转换器，并返回数据转换器对象，而不是Object[]。产生同样结果的另一种可选方法如下：

```
session.createCriteria(Bid.class)
        .createAlias("bidder", "u")
        .setProjection( Projections.projectionList()
          .add( Projections.groupProperty("u.id") )
          .add( Projections.groupProperty("u.username") )
          .add( Projections.count("id") )
          .add( Projections.avg("amount") )
        );
```

喜欢哪种语法只是个人的偏好问题。更复杂的示例则把别名应用到被统计和被分组的属性，用于给结果排序：

```
session.createCriteria(Bid.class)
        .createAlias("bidder", "u")
        .setProjection( Projections.projectionList()
          .add( Projections.groupProperty("u.id") )
          .add( Projections.groupProperty("u.username").as("uname") )
          .add( Projections.count("id") )
          .add( Projections.avg("amount") )
        )
        .addOrder( Order.asc("uname") );
```

在编写本书之时，对HAVING的支持和对被统计结果的限制，在Hibernate条件查询中还不可用。这或许会增加到未来的版本中。

可以把原生的SQL表达式添加到条件查询中的限制，同样的特性也可用于投影。

3. 利用SQL投影

SQL投影是被添加到生成的SQL SELECT子句的一个任意片段。下列查询产生统计和分组，就像在前面的示例中一样，但是还添加了一个额外的值到结果中（货品的数量）：

```
String sqlFragment =
    "(select count(*) from ITEM i where i.ITEM_ID = ITEM_ID)" +
    " as numOfItems";

session.createCriteria(Bid.class)
        .createAlias("bidder", "u")
        .setProjection( Projections.projectionList()
            .add( Projections.groupProperty("u.id") )
            .add( Projections.groupProperty("u.username") )
            .add( Projections.count("id") )
            .add( Projections.avg("amount) )
            .add( Projections.sqlProjection(
                    sqlFragment,
                    new String[] { "numOfItems" },
```

```
                    new Type[] { Hibernate.LONG }
                )
            )
        );
```

生成的SQL如下：

```
select
        u.USER_ID,
        u.USERNAME,
        count(BID_ID),
        avg(BID_AMOUNT),
        (select
            count(*)
        from
            ITEM i
        where
            i.ITEM_ID = ITEM_ID) as numOfItems
    from
        BID
    inner join
        USERS u
            on BIDDER_ID = u.USER_ID
    group by
        u.USER_ID,
        u.USERNAME
```

这个SQL片段被嵌入到SELECT子句中。它可以包含数据库管理系统支持的任意表达式和函数调用。任何未限定列名（如ITEM_ID）都指向条件根实体（BID）的表。必须告诉Hibernate返回的SQL投影的别名numOfItems，以及它的Hibernate值映射类型Hibernate.LONG。

Criteria API的真正威力在于把任意的Criterion与示例对象组合的可能。这项特性就是大家所知的按示例查询。

15.1.4　按示例查询

通过（根据用户输入组合几个可选的条件），编程式地构建条件查询还是很常见的。例如，系统管理员可能希望按名或者姓的任何组合搜索用户，并获取按用户名排序的结果。

利用HQL或者JPA QL，你可以用字符串操作构建查询：

```
public List findUsers(String firstname,
                      String lastname) {

    StringBuffer queryString = new StringBuffer();
    boolean conditionFound = false;

    if (firstname != null) {
        queryString.append("lower(u.firstname) like :firstname ");
        conditionFound=true;
    }
    if (lastname != null) {
        if (conditionFound) queryString.append("and ");
        queryString.append("lower(u.lastname) like :lastname ");
        conditionFound=true;
```

```
    }

    String fromClause = conditionFound ?
                            "from User u where " :
                            "from User u ";

    queryString.insert(0, fromClause).append("order by u.username");

    Query query = getSession()
                      .createQuery( queryString.toString() );

    if (firstname != null)
        query.setString( "firstName",
                         '%' + firstname.toLowerCase() + '%' );
    if (lastname != null)
        query.setString( "lastName",
                         '%' + lastname.toLowerCase() + '%' );

    return query.list();
}
```

这段代码十分冗长乏味，因此我们尝试一种不同的方法。利用你目前已经学过的Criteria API看起来很有希望进行简化：

```
public List findUsers(String firstname,
                      String lastname) {

    Criteria crit = getSession().createCriteria(User.class);

    if (firstname != null) {
        crit.add( Restrictions.ilike("firstname",
                                     firstname,
                                     MatchMode.ANYWHERE) );
    }
    if (lastname != null) {
        crit.add( Restrictions.ilike("lastname",
                                     lastname,
                                     MatchMode.ANYWHERE) );
    }

    crit.addOrder( Order.asc("username") );

    return crit.list();
}
```

这段代码更短一些。注意ilike()操作符执行不区分大小写的匹配。似乎毫无疑问这就是比较好的方法了。然而，对于利用许多可选的搜索条件的搜索屏幕来说，还有一种更好的方法。

随着你增加新的搜索条件，findUsers()的参数清单也在增加，捕捉可搜索的属性作为对象将会更好。因为所有的搜索属性都属于User类，为什么不把User的实例用于这一用途？

按示例查询（QBE）依赖这一思想。你通过一些被初始化的属性提供被查询类的实例，并且查询返回包含匹配的属性值的所有持久化实例。Hibernate实现QBE作为Criteria查询API的一部分：

```
public List findUsersByExample(User u) throws {

    Example exampleUser =
      Example.create(u)
```

```
                .ignoreCase()
                .enableLike(MatchMode.ANYWHERE)
                .excludeProperty("password");

        return getSession().createCriteria(User.class)
                           .add(exampleUser)
                           .list();
    }
```

对create()的调用给User的指定实例返回Example的一个新实例。ignoreCase()方法为所有字符串值属性把实例查询放进一个不区分大小写的模式中。对enableLike()的调用指定SQL Like操作符应该用于所有的字符串值属性，并指定MatchMode。最后，可以利用excludeProperty()从搜索中排除特定的属性。默认情况下，所有的值类型属性，除了标识符属性之外，都用在比较式中。

你已经进一步明显地简化了这段代码。关于Hibernate Example查询最好的方面是，Example只是一个一般的Criterion。可以随意混合和匹配按实例查询和按条件查询。

通过进一步限制搜索结果为包含未售Items的用户，看一下这是如何进行的。因此可以把Criteria添加到示例用户中，利用它的Items的items集合来约束结果：

```
public List findUsersByExample(User u){

    Example exampleUser =
      Example.create(u)
              .ignoreCase()
              .enableLike(MatchMode.ANYWHERE);

    return getSession().createCriteria(User.class)
              .add( exampleUser )
              .createCriteria("items")
                  .add( Restrictions.isNull("successfulBid") )
              .list();
}
```

更好的是，可以在同一个搜索中组合User属性和Item属性：

```
public List findUsersByExample(User u, Item i) {

    Example exampleUser =
      Example.create(u).ignoreCase().enableLike(MatchMode.ANYWHERE);

    Example exampleItem =
      Example.create(i).ignoreCase().enableLike(MatchMode.ANYWHERE);

    return getSession().createCriteria(User.class)
              .add( exampleUser )
              .createCriteria("items")
                  .add( exampleItem )
              .list();
}
```

现在，请你退回一步，并考虑利用手工编码的SQL/JDBC实现这个搜索屏幕需要多少代码。我们不想把它复制到这里，它会增加好多页。还要注意findUsersByExample()方法的客户端不需要知道关于Hibernate的任何事，它仍然可以给搜索创建复杂的条件。

如果HQL、JPA QL甚至Criteria和Example都不够强大，难以表达特定查询，你就必须退回到原生的SQL。

15.2 利用原生的 SQL 查询

HQL、JPA QL或者条件查询，都应该足够灵活，可以执行你喜欢的几乎任何查询。它们指向被映射的对象模式，因此，如果你的映射结果不出所料，Hibernate的查询就应该提供你喜欢的任何获取数据的功能。有几种例外。如果你想包括一个原生的SQL提示，来指示数据库管理系统查询优化器（例如，你需要自己编写SQL）。HQL、JPA QL和条件查询都没有用于这个的关键字。

另一方面，你不退回到手工的SQL查询，而是始终努力扩展内建的查询机制，并给特殊的操作包括支持。这用HQL和JPA QL更难以实现，因为你必须修改这些基于字符串语言的语法。扩展Criteria API和添加新方法或者新Criterion类很容易。查看一下org.hibernate.criterion包中的Hibernate源代码，它精心设计且制成了文档。

当无法扩展内建的查询工具或者防止不可移植的手工编写的SQL时，你应该首先考虑使用Hibernate原生的SQL查询选项，就是我们现在正在介绍的。记住，你总是可以退回到简单的JDBC Connection，并自己准备任何SQL语句。Hibernate的SQL选项允许你在Hibernate API中嵌入SQL语句，并让你从能让生活变得更轻松的额外服务中受益。

最重要的是，Hibernate可以处理你SQL查询的结果集。

15.2.1 自动的结果集处理

利用Hibernate API执行SQL语句的最大好处在于，自动把表格式的结果集封送到业务对象中去。下列SQL查询返回Category对象的一个集合：

```
List result = session.createSQLQuery("select * from CATEGORY")
                     .addEntity(Category.class)
                     .list();
```

Hibernate读取SQL查询的结果集，并尝试发现你的映射元数据中所定义的列名和类型。如果返回列CATEGORY_NAME，就把它映射到Category类的name属性，Hibernate就知道如何填充该属性，并最后返回完全加载的业务对象。

SQL查询中的*投影结果集中所有选中的列。因此自动的发现机制只适用于繁琐的查询，更加复杂的查询则需要显式的投影。下一个查询返回Item对象的一个集合：

```
session.createSQLQuery("select {i.*} from ITEM i" +
                       " join USERS u on i.SELLER_ID = u.USER_ID" +
                       " where u.USERNAME = :uname")
       .addEntity("i", Item.class)
       .setParameter("uname", "johndoe");
```

SQL SELECT子句包括一个占位符，它命名表别名为i，并把这张表的所有列投影到结果中。任何其他的表别名（例如被联结的USERS表，它只与限制相关）不包括在结果集中。现在用addEntity()告诉Hibernate，别名i的占位符是指填充Item实体类所需要的所有列。列名和类型再次由Hibernate在查询执行和结果封送期间自动推测。

甚至可以在原生的SQL查询中即时抓取被关联的对象和集合：

```
session.createSQLQuery("select {i.*}, {u.*} from ITEM i" +
                       " join USERS u on i.SELLER_ID = u.USER_ID" +
                       " where u.USERNAME = :uname")
        .addEntity("i", Item.class)
        .addJoin("u", "i.seller")
        .setParameter("uname", "johndoe");
```

这个SQL查询从两个表别名中投影两组列，并且使用两个占位符。占位符i仍然是指填充了由这个查询返回的Item实体对象的列。addJoin()方法告诉Hibernate，别名u是指可以用来立即填充每个Item被关联seller的列。

自动把结果集封送到业务对象，这并不是Hibernate中原生的SQL查询特性的唯一好处。即使你想获取的只是一个简单的标量值，也可以使用它。

15.2.2　获取标量值

标量值可以是任何Hibernate值类型。最常见的是字符串、数字或者时间戳。下列SQL查询返回货品数据：

```
List result = session.createSQLQuery("select * from ITEM").list();
```

这个查询的result是Object[]的一个List，实际上是一张表。每个数组中的每个字段都是标量类型即字符串、数字或者时间戳。除了包装在一个Object[]中之外，结果与类似的简单JDBC查询结果完全一样。这显然没有太大的用处，但是Hibernate API的其中一个好处在于它抛出unchecked exception，因此你不必把查询包装在try/catch块中，就像如果你调用JDBC API时所必须做的那样。

如果没有用*投影每件东西，就需要告诉Hibernate你想要从结果中返回哪些标量值：

```
session.createSQLQuery("select u.FIRSTNAME as fname from USERS u")
        .addScalar("fname");
```

addScalar()方法告诉Hibernate，你的fname SQL别名应该作为标量值返回，并且类型应该被自动推测。查询返回字符串的一个集合。这种自动的类型发现在大多数案例中都运行得很好，但是你有时可能想要显式地指定类型——例如，想要用UserType转换值的时候：

```
Properties params = new Properties();
params.put("enumClassname", "auction.model.Rating");

session.createSQLQuery(
        "select c.RATING as rating from COMMENTS c" +
        " where c.FROM_USER_ID = :uid"
    )
    .addScalar("rating",
               Hibernate.custom(StringEnumUserType.class, params) )
    .setParameter("uid", new Long(123));
```

首先，看一下SQL查询。它选择COMMENTS表的RATING列，并限制由特定的用户所做出的评价结果。假设数据库中的这个字段包含字符串值（例如EXCELLENT、OK或者BAD）。因此SQL查询的结果是字符串值。

你一般不会把这个映射为Java中简单的字符串，而是使用枚举，或者用定制的Hibernate UserType。5.3.7节这么做过，并创建过一个可以把SQL数据库中的字符串转变为Java中任何枚举实例的StringEnumUserType。它必须用enumClassname参数化，你想要它转换值到——这个例子中的auction.model.Rating。通过在查询中用addScalar()方法设置预设的定制类型，把它启用为处理结果的转换器，这样你取回了Rating对象的集合，而不是简单的字符串。

最后，可以在同一个原生的SQL查询中，混合标量结果和实体对象：

```
session.createSQLQuery(
        "select {i.*}, u.FIRSTNAME as fname from ITEM i" +
        " join USERS u on i.SELLER_ID = u.USER_ID" +
        " where u.USERNAME = :uname"
    )
    .addEntity("i", Item.class)
    .addScalar("fname")
    .setParameter("uname", "johndoe");
```

这个查询的结果仍然是Object[]的一个集合。每一个数组有两个字段：Item实例和字符串。

你或许会认同原生的SQL查询比HQL或者JPA QL语句更难以阅读，并且把它们隔离和外部化到映射元数据中似乎会更好一些。8.2.2节对存储过程查询这么做过。这里不再重复，因为存储过程查询和简单的SQL查询之间的唯一区别只在于调用的语法或者语句——封送和结果集映射选项是一样的。

Java Persistence标准化了JPA QL，并允许退回到原生的SQL。

15.2.3 Java Persistence 中的原生 SQL

Java Persistence通过在EntityManager中用createNativeQuery()方法支持原生的SQL查询。原生的SQL查询可以返回实体实例、标量值或者两者的混合。然而，不同于Hibernate，Java Persistence中的API利用映射元数据来定义结果集处理。我们来看一些实例。

简单的SQL查询不需要显式的结果集映射：

```
em.createNativeQuery("select * from CATEGORY", Category.class);
```

结果集被自动封送到Category实例的一个集合。注意，持久化引擎希望创建Category实例所需的所有列都通过查询返回，包括所有的属性、组件和外键列——否则就会抛出异常。列在结果集中按名称搜索。你可能必须在SQL中使用别名，返回与定义在实体映射元数据中相同的列名。

如果你的原生SQL查询返回多个实体类型或者标量类型，就需要应用显式的结果集映射。例如，返回Object[]集合的查询，在这里每一个数组索引0中都是Item实例，索引1中是User实例，可以写成下面这样：

```
em.createNativeQuery("select " +
    "i.ITEM_ID, i.ITEM_PRICE, u.USERNAME, u.EMAIL " +
    "from ITEM i join USERS u where i.SELLER_ID = u.USER_ID",
    "ItemSellerResult");
```

最后一个参数ItemSellerResult，是定义在元数据中的一个结果映射的名称（在类或者全局的JPA XML级别中）：

```
@SqlResultSetMappings({
    @SqlResultSetMapping(
        name = "ItemSellerResult",
        entities = {
            @EntityResult(entityClass = auction.model.Item.class),
            @EntityResult(entityClass = auction.model.User.class)
        }
    )
})
```

这个结果集映射对于我们介绍过的查询很可能不起作用——给自动映射记住这一点，实例化Item和User对象所需的所有列都必须在SQL查询中被返回。返回的4个列不可能只表示持久化的属性。为了便于举例，假设它们的确只表示持久化的属性，而你真正的问题在于结果集中的列名，它们与被映射列的名称不匹配。首先，把别名添加到SQL语句：

```
em.createNativeQuery("select " +
    "i.ITEM_ID as ITEM_ID, i.ITEM_PRICE as ITEM_PRICE, " +
    "u.USERNAME as USER_NAME, u.EMAIL as USER_EMAIL " +
    "from ITEM i join USERS u on i.SELLER_ID = u.USER_ID",
    "ItemSellerResult");
```

接下来，在结果集映射中使用@FieldResult来把别名映射到实体实例的字段：

```
@SqlResultSetMapping(
name = "ItemSellerResult",
entities = {
    @EntityResult(
        entityClass = auction.model.Item.class,
        fields = {
          @FieldResult(name = "id", column = "ITEM_ID"),
          @FieldResult(name = "initialPrice", column = "ITEM_PRICE")
        }),
    @EntityResult(
        entityClass = auction.model.User.class,
        fields = {
          @FieldResult(name = "username", column = "USER_NAME"),
          @FieldResult(name = "email", column = "USER_EMAIL")
        })
})
```

也可以返回标量类型结果。下列查询返回拍卖货品标识符和每件货品的出价数量：

```
em.createNativeQuery("select " +
    "i.ITEM_ID as ITEM_ID, count(b.*) as NUM_OF_BIDS " +
    "from ITEM i join BIDS b on i.ITEM_ID = b.ITEM_ID " +
    "group by ITEM_ID",
    "ItemBidResult");
```

这一次，结果集映射没有包含实体结果映射，而只有列：

```
@SqlResultSetMapping(
name = "ItemBidResult",
columns = {
    @ColumnResult(name = "ITEM_ID"),
    @ColumnResult(name = "NUM_OF_BIDS")
})
```

这个查询的结果是Object[]的一个集合，有两个字段，两者都是数字类型（最可能是long）。如果想要把实体和标量类型混合成为一个查询结果，就在@SqlResultSetMapping中组合entities和columns属性。

最后，注意JPA规范并不要求原生的SQL查询支持具名参数绑定。Hibernate支持它。

接下来讨论另一项更为怪异但更方便的Hibernate特性（Java Persistence没有它的等价物）：集合过滤器（collection filter）。

15.3 过滤集合

你可能希望针对一个集合的所有元素执行一个查询。例如，你可能有一个Item，并希望按出价创建的时间获取对该特定货品的所有出价。为此可以映射一个排序的或者有序的集合，但是有一种更容易的选择。可以编写查询，你应该已经知道要如何进行了：

```
session.createQuery("from Bid b where b.item = :givenItem" +
                    " order by b.created asc")
       .setEntity("givenItem", item);
```

这个查询有效，因为出价和货品之间的关联是双向的，每一个Bid都知道它的Item。这个查询中没有任何联结，b.item指向BID表中的ITEM_ID列，因此你直接给比较设置了值。想象这个关联是单向的：Item有Bids的一个集合，但是没有从Bid到Item的反向关联。可以试试下列查询：

```
select b from Item i join i.bids b
    where i = :givenItem order by b.amount asc
```

这个查询的效率很低，它用了一个完全没有必要的联结。一种更好更优雅的解决方案是使用集合过滤器——可以应用到持久化集合（或者数组）的一种特殊查询。它常用于进一步限制或者排序结果。在一个已经加载的Item和它的出价集合中应用它：

```
List filteredCollection =
        session.createFilter( item.getBids(),
                              "order by this.created asc" ).list();
```

这个过滤器相当于本节的第一个查询，并生成相同的SQL。Session中的createFilter()方法用了两个参数：一个持久化的集合（不一定要被初始化）和一个HQL查询字符串。集合过滤器查询有一个隐式的FROM子句和一个隐式的WHERE条件。别名this暗指出价集合的元素。

Hibernate集合过滤器不是在内存中执行的。出价的集合可能在调用过滤器时没有被初始化，如果是这样，就保持未初始化。此外，不要把过滤器应用到瞬时的集合或者查询结果。它们只能应用到当前被添加到Hibernate持久化上下文的一个实体实例引用的持久化集合。术语“过滤器”有点误导，因为过滤的结果是一个全新的不同集合，并没有接触到原始的集合。

HQL查询唯一需要的子句是FROM子句。因为集合过滤器有一个隐式的FROM子句，下面是一个有效的过滤器：

```
List filteredCollection =
        session.createFilter( item.getBids(), "" ).list();
```

让大家（包括这项特性的设计者）惊喜的是，这个小小的过滤器很有用。可以用它给集合元素进行分页：

```
List filteredCollection =
    session.createFilter( item.getBids(), "" )
            .setFirstResult(50)
            .setMaxResults(100)
            .list();
```

但是，你通常会给分页查询使用ORDER BY。

即使在集合过滤器中不需要FROM子句，如果喜欢，你也可以放一个。集合过滤器甚至不需要返回正被过滤的集合的元素。下一个查询返回任何与指定集合中的一个类别同名的Category：

```
String filterString =
  "select other from Category other where this.name = other.name";

List result =
    session.createFilter( cat.getChildCategories(), filterString )
            .list();
```

下列查询返回对该货品有出价的Users的一个集合：

```
List result =
    session.createFilter( item.getBids(),
                          "select this.bidder" )
             .list();
```

下一个查询返回所有这些用户的出价（包括对其他货品的出价）：

```
List result =
    session.createFilter(
        item.getBids(),
        "select elements(this.bidder.bids)"
    ).list();
```

注意这个查询用了特殊的HQL elements()函数来投影一个集合的所有元素。

所有这些都很有趣，但是集合过滤器存在的最重要原因在于，允许应用程序获取集合的一些元素，而不用初始化整个集合。对于大集合而言，实现满意的性能很重要。下列查询获取一位用户在过去一周所做的所有出价：

```
List result =
    session.createFilter( user.getBids(),
                          "where this.created > :oneWeekAgo" )
            .setTimestamp("oneWeekAgo", oneWeekAgo)
            .list();
```

这段代码还是没有初始化User的bids集合。

无论用哪种语言或者哪个API编写的查询，都始终应该在你决定用可选的查询高速缓存给它们提速之前，把它们调优成能够按预期执行。

15.4　高速缓存查询结果

13.3节中已经讨论过二级高速缓存和Hibernate的一般高速缓存架构。你知道二级高速缓存是一个共享的数据高速缓存，每当你访问一个未加载的代理或者集合时，或者当你按标识符加载对象时（从二级高速缓存的观点来看，这些全部都是标识符查找），Hibernate就在这个高速缓存中通过查找尝试解析数据。另一方面，查询结果在默认情况下没有被高速缓存。

有些查询仍然使用二级高速缓存，这取决于你如何执行查询。例如，如果决定用iterate()执行查询（如前一章所述），只从数据库中获取实体的主键，并通过一级高速缓存查找实体数据，如果对特定的实体启用二级高速缓存，也通过二级高速缓存查找。我们还断定这个选项只有当二级高速缓存被启用时才有意义，因为列读取的优化一般不影响性能。

高速缓存查询结果是一个完全不同的问题。查询结果高速缓存在默认情况下是禁用的，并且每一个HQL、JPA QL、SQL和Criteria查询都始终先命中数据库。我们首先介绍如何启用查询结果高速缓存，以及它是如何工作的。然后讨论它为什么被禁用，以及为什么很少有查询从结果高速缓存中受益。

15.4.1 启用查询结果高速缓存

查询高速缓存必须用Hibernate配置属性来启用：

```
hibernate.cache.use_query_cache = true
```

然而，单独这个设置对于Hibernate高速缓存查询结果还不够。默认情况下，所有的查询都始终忽略高速缓存。为了给特定的查询启用查询高速缓存（允许它的结果被添加到高速缓存，并允许它从高速缓存中提取结果），那么使用org.hibernate.Query接口。

```
Query categoryByName =
    session.createQuery("from Category c where c.name = :name");
categoryByName.setString("name", categoryName);
categoryByName.setCacheable(true);
```

setCachable()方法启用了结果高速缓存。它在Criteria API上也可用。如果想给javax.persistence.Query启用结果高速缓存，就使用setHint("org.hibernate.cacheable", true)。

15.4.2 理解查询高速缓存

当某个查询被第一次执行时，它的结果被高速缓存在高速缓存区域——这个区域不同于你可能已经配置的任何其他实体或者集合高速缓存区域。这个区域的名称默认为org.hibernate.cache.QueryCache。

可以通过setCacheRegion()方法，给一个特定查询改变高速缓存区域：

```
Query categoryByName =
    session.createQuery("from Category c where c.name = :name");
categoryByName.setString("name", categoryName);
categoryByName.setCacheable(true);
categoryByName.setCacheRegion("my.Region");
```

这几乎没有必要，你只在需要一个不同的区域配置时，才会给某些查询使用不同的高速缓存区域——例如，在一个更细粒度的级别上限制查询高速缓存的内存消耗。

标准的查询结果高速缓存区域保存着SQL语句（包括所有绑定的参数）和每个SQL语句的结果集。但是，这不是完整的SQL结果集。如果结果集包含实体实例（前面的示例查询返回Category实例），那么只有标识符值被保存在结果集高速缓存中。当结果集被放进高速缓存区域

时，每个实体的数据列就被它丢弃了。因此，命中查询结果高速缓存意味着，对于前面的查询，Hibernate将找到一些Category标识符值。

高速缓存实体的状态，是二级高速缓存区域auction.model.Category（结合持久化上下文）的责任。这类似于iterate()的查询策略，如前所述。换句话说，如果你查询实体，并决定启用高速缓存，就要确保你也给这些实体启用了常规的二级高速缓存。如果没有，可能在启用查询高速缓存之后，以更多的数据库命中而告终。

如果高速缓存了不返回实体实例而只返回相同标量值（如货品名称和价格）的查询结果，这些值就被直接保存在查询结果高速缓存中。

如果查询结果高速缓存在Hibernate中被启用，另一个始终需要的高速缓存区域也出现了：org.hibernate.cache.UpdateTimestampsCache。这是Hibernate内部使用的一个高速缓存区域。

Hibernate用时间戳区域来决定被高速缓存的查询结果集是否为失效的。当你重新执行一个启用了高速缓存的查询时，Hibernate就在时间戳高速缓存中查找对被查询的（几张）表所做的最近插入、更新、或者删除的时间戳。如果找到的时间戳晚于高速缓存查询结果的时间戳，高速缓存结果就被丢弃，并产生一个新查询。这就有效地保证了：如果查询可能涉及的任何表中包含被更新的数据，Hibernate就不会使用被高速缓存的查询结果；因此高速缓存结果就可能是失效的。为了得到最佳结果，你应该配置时间戳区域，以便表的更新时间戳没有超出高速缓存期，而来自这些表的查询结果仍然被高速缓存在一个其他区域中。最容易的方法是在你二级高速缓存提供程序的配置中，关闭时间戳高速缓存区域的过期期限。

15.4.3　什么时候使用查询高速缓存

大多数查询都没有从结果高速缓存中受益。这可能有点出乎意料。毕竟，避免数据库命中听起来似乎总是件好事。比起对象导航或者按标识符获取而言，它之所以并非永远对任意的查询有用，有以下两个原因。

首先，必须问问，你会每隔多久重复执行同一个查询。毫无疑问，在你的应用程序中可能有很多已反复执行的查询（把完全相同的实参绑定到参数），以及自动生成相同的SQL语句。这种情况很罕见，但是当你确定某个查询被重复执行时，它就变成了结果高速缓存的一个很好的备选对象了。

其次，对于执行许多查询和少数插入、删除或者更新的应用程序来说，高速缓存查询可以提升性能和可伸缩性。另一方面，如果应用程序执行许多写入操作，查询高速缓存就没有得到有效的利用。当出现在被高速缓存的查询结果中的表的任何行有任何插入、更新或者删除时，Hibernate就会废弃掉被高速缓存的查询结果集。这意味着被高速缓存结果的寿命可能很短，即使重复执行查询，由于相同数据（相同表）的并发修改，也没有可以使用的高速缓存结果。

对于许多查询来说，查询结果高速缓存的好处是不存在的，至少没有达到所期待的影响程度。但是有一种特殊的查询可以从结果高速缓存中大大受益。

15.4.4 自然标识符高速缓存查找

假设你有一个包含自然键的实体。我们不讨论自然主键，而是讨论应用到你实体的单个或者复合属性的业务键（business key）。例如，如果用户的登录名称不可变，那么它可以是唯一的业务键。这是我们已经隔离作为好的equals()对象等同性子程序实现的最好的键。可以在9.2.3节中找到这种键的示例。

通常要把构成自然键的属性映射为Hibernate中的常规属性。可以在数据库级启用一个unique约束来表示这个键。例如，如果考虑User类，可能要用username和emailAddress构成实体的业务键：

```
<class name="User">
    <id name="id".../>

    <property name="username" unique-key="UNQ_USERKEY"/>
    <property name="emailAddress" unique-key="UNQ_USERKEY"/>
    ...

</class>
```

这个映射在跨越两列的数据库级上启用了一个唯一键约束。也假设业务键属性是不可变的。虽然这不太可能，因为你可能允许用户更新他们的电子邮件地址，但我们现在介绍的这项功能则只有当你处理不可变的业务键时才有意义。映射不可变性如下：

```
<class name="User">
    <id name="id".../>

    <property name="username"
              unique-key="UNQ_USERKEY"
              update="false"/>

    <property name="emailAddress"
              unique-key="UNQ_USERKEY"
              update="false"/>
    ...

</class>
```

或者，通过业务键利用高速缓存查找，可以用<natural-id>映射它：

```
<class name="User">
    <id name="id".../>

    <cache usage="read-write"/>

    <natural-id mutable="false">
        <property name="username"/>
        <property name="emailAddress"/>
    </natural-id>
    ...

</class>
```

这个分组自动启用跨越所有被组合属性的唯一键SQL约束的生成。如果mutable属性被设置成false，它也防止被映射列的更新。现在可以给高速缓存查找使用这个业务键：

```
Criteria crit = session.createCriteria(User.class);

crit.add( Restrictions.naturalId()
            .set("username", "johndoe")
            .set("emailAddress", "jd@hibernate.org")
        );
crit.setCacheable(true);

User result = (User) crit.uniqueResult();
```

这个条件查询根据业务键找到一个特定的用户对象。它通过业务键导致一个二级高速缓存查找——记住，这通常是一个通过主键的查找，并且可能只用于按主标识符的获取。业务键映射和Criteria API允许你通过业务键表达这个特殊的二级高速缓存查找。

在编写本书之时，还没有用于自然标识符映射的Hibernate扩展注解可用，并且HQL也不支持与此相当的关键字，用于通过业务键的查找。

从我们的观点来看，二级高速缓存是一项重要的特性，但它不是优化性能时的首选方案。查询设计中的错误，或者对象模型中没有必要的复杂部分，都无法通过一个“全部高速缓存”的方法得到改进。如果应用程序只用一个热高速缓存（即几个小时或者几天的运行时之后的完整高速缓存）以可接受的等级执行，那么就应该查找出严重的设计错误、损害性能的查询和n+1查询问题。在决定启用此处所述的任何一个查询高速缓存选项时，首先要按照13.2.5节中介绍的指导方针，检查和调优一下你的应用程序。

15.5　小结

在本章中，你已经通过Hibernate Criteria和Example API，有步骤地生成了查询。我们也讨论了嵌入的和外部化的SQL查询，以及如何自动地把SQL查询的结果集映射到更方便的业务对象。Java Persistence也支持原生的SQL，并标准化如何映射外部化的SQL查询的结果集。

最后，我们涵盖了查询结果高速缓存，并讨论了它为什么只在某些情况下有用。

表15-1展现了可以用来比较原生的Hibernate特性和Java Persistence的一个概括。

表15-1　第15章中Hibernate和JPA对照表

Hibernate Core	Java Persistence和EJB 3.0
Hibernate支持一个强大的Criteria和Example API，用于有步骤的查询生成	即将面世的标准版本中将有一些QBC和QBE API
Hibernate通过结果集的自动封送，具有灵活的映射选项用于嵌入式和外部化的SQL查询	Java Persistence标准化SQL嵌入和映射，并支持结果集封送
Hibernate支持集合过滤器API	Java Persistence没有标准化集合过滤器API
Hibernate可以高速缓存查询结果	特定于Hibernate的查询提示可以用来高速缓存查询结果

第16章将把所有的知识点放在一起，并关注利用Hibernate、Java Persistence和EJB 3.0组件的应用程序的设计和架构。我们也将对Hibernate应用程序进行单元测试。

第 16 章

创建和测试分层的应用程序

本章内容

- 创建分层的应用程序
- 托管的组件和服务
- 集成测试的策略

Hibernate准备只用于任何可以想象的架构场景。Hibernate可以在servlet容器内部运行，可以把它与Web应用程序框架（如Struts、WebWork或者Tapestry）一起使用，或者用在EJB容器内部，或者用在Java Swing应用程序中管理持久化数据。

甚至——尤其可能——由于所有这些选项，理解Hibernate应该如何整合到一个特定的基于Java的架构经常是有困难的。你将不可避免地需要编写基础结构代码来支持自己的应用程序的设计。本章阐述常用的Java架构，并介绍Hibernate如何整合到每一种场景中。

我们讨论如何在一个基于Web应用程序的典型的请求/响应中设计和创建分层，以及如何按功能分离代码。之后，介绍Java EE服务和EJB，并介绍托管组件如何使你的生活更加轻松，减少了没有托管组件时必需的基础结构编码。

最后，假设你也关注测试分层应用程序（无论是否包含托管组件）。如今，在开发人员的工作中，测试已经成了最重要的活动之一，并且应用正确的工具和策略是提高工作效率的根本（更不用说软件的质量了）。我们讨论并利用当前最受欢迎的测试框架TestNG进行单元、功能和集成测试。

从一个典型的Web应用程序示例开始。

16.1 Web 应用程序中的 Hibernate

第1章已经强调过规则的应用程序分层的重要性。分层有助于实现关注点的分离，通过把完成类似工作的代码分组，使得代码更易读。然而，分层要付出一些代价。每增加一层，都要增加由于实现一块简单功能而花费的代码量——而代码越多就越难以改变功能。

本节介绍如何在一个典型的分层应用程序中整合Hibernate。假设你想要用Java servlet编写一个简单的Web应用程序。我们需要CaveatEmptor应用程序的一个简单用例，来示范这些思想。

16.1.1　用例简介

当用户对一件货品进行出价时，CaveatEmptor必须在单个请求中执行下列任务：

(1) 查找大于该货品现有出价的最大金额的用户输入金额。

(2) 查找还没有结束的拍卖。

(3) 给该货品创建一个出价。

(4) 通知任务结果的用户。

如果其中任何一项查找失败，应该告知客户原因；如果两项查找都成功，则应该通知用户出价已经成功。这些查找是业务规则。如果访问数据库的时候出现失败，应该通知用户，告知目前系统不可用（一个基础结构关注点）。

下面来看如何在Web应用程序中实现这一点。

16.1.2　编写控制器

大多数Java Web应用程序使用某种MVC（Model/View/Controller，模型/视图/控制器）应用程序框架；甚至使用简单servlet的许多应用程序也遵循MVC模式，通过利用模板来实现展现代码，把应用程序控制逻辑分离到一个servlet或者多个servlet。

你现在就要编写一个这样的控制器servlet，它实现前面介绍过的用例。利用MVC方法，编写在一个具名`PlaceBidAction`的动作的`execute()`方法中实现“出价”用例的代码。假设某种Web框架，我们不探讨如何读取请求参数或者如何转到下一页。所显示的代码甚至可能是一个简单servlet的`doPost()`方法的实现。

如代码清单16-1所示，编写这种控制器的第一步是在一个地方混合所有的关注点——没有分层。

代码清单16-1　在一个execute()方法中实现一个用例

```
public void execute() {

    Long itemId = ...              // Get value from request
    Long userId = ...              // Get value from request
    BigDecimal bidAmount = ...     // Get value from request
    Transaction tx = null;

    try {

        Session session =                                          ❶
            HibernateUtil.getSessionFactory().getCurrentSession();

        tx = session.beginTransaction();

        // Load requested Item
        Item item = (Item) session.load(Item.class, itemId);       ❷

        // Check auction still valid
        if ( item.getEndDate().before( new Date() ) ) {            ❸
          ...      // Forward to error page
        }
```

```
        // Check amount of Bid                                    ❹
        Query q =
            session.createQuery("select max(b.amount)" +
                                " from Bid b where b.item = :item");
        q.setEntity("item", item);
        BigDecimal maxBidAmount = (BigDecimal) q.uniqueResult();
        if (maxBidAmount.compareTo(bidAmount) > 0) {
          ...      // Forward to error page
        }

        // Add new Bid to Item                                                ❺
        User bidder = (User) session.load(User.class, userId);
        Bid newBid = new Bid(bidAmount, item, bidder);
        item.addBid(newBid);

        ...      // Place new Bid into request context                     ❻

        tx.commit();                                                       ❼

        ...      // Forward to success page

    } catch (RuntimeException ex) {                                        ❽
        if (tx != null) tx.rollback();
        throw ex;
    }
}
```

❶ 利用当前的持久化上下文得到一个Session，然后启动一个数据库事务。2.1.3节中已介绍过HibernateUtil类，并在11.1节中讨论过持久化上下文范围。在当前的会话中启动一个新的数据库事务。

❷ 利用Item的标识符值从数据库加载它。

❸ 如果拍卖的终止日期比当前的日期早，就转到一个错误页面。通常，对于这种带有限定错误消息的异常你会想要一个更为成熟的错误处理。

❹ 利用HQL查询，查找数据库中是否有对当前货品所做的更高出价。如果有，就转到一个错误页面。

❺ 如果所有的查找都成功，就将它添加到该货品，对其出一个新价。不必手工保存它——利用传播性持久化保存它（从Item级联到Bid）。

❻ 新的Bid实例需要被保存在可以被后面的页面访问的某些变量中，这样可以把它显示给用户。可以为此在servlet请求上下文中使用一个属性。

❼ 提交数据库事务，将Session的当前状态清除到数据库，并自动关闭当前的Session。

❽ 如果抛出任何RuntimeException，就通过Hibernate或者其他服务来回滚事务，并重新抛出要在控制器外部进行适当处理的异常。

这段代码的第一个问题在于由所有事务和异常处理代码造成的混乱。因为这段代码对所有动作都是典型地一致，你想要将它集中在某个地方。一种方法是把它放在某些动作的抽象超类的execute()方法里面。*延迟初始化*（lazy initialization）也有问题，如果在success页面中访问新的出价，把它拖出请求上下文进行渲染：关闭Hibernate持久化上下文，你再也无法延迟加载集合或者

代理。

让我们开始整理这个设计，并引入分层。第一步是通过实现OSIV（Open Session In View）模式，在success页面中启用延迟加载。

16.1.3　OSIV模式

OSIV模式背后的动机在于，视图通过导航从某个脱管对象开始的对象图，从业务对象中拖出信息——例如，放在请求上下文中的刚由动作创建的Bid实例。视图（即，必须被渲染和显示的页面）访问这个脱管对象，给页面获得内容数据。

在Hibernate应用程序中，可能有未被初始化的关联（代理或者集合），它们必须在渲染视图时被驳回。在这个例子中，视图可以通过调用newBid.getBidder().getItems().iterator()，列出被出价人出售的所有货品（作为摘要屏的一部分）。这种情况很罕见，但确定是一个有效的访问。因为User的items集合只按需加载（Hibernate的延迟关联和集合默认行为），它在此时没有被初始化。无法加载未被初始化的代理和处于脱管状态的实体实例的集合。

如果Hibernate Session和相应的持久化上下文始终在动作的execute()方法结束时关闭，Hibernate就会在这个未加载的关联(或者集合)被访问时，抛出LazyInitializationException。持久化上下文不再可用，因此Hibernate无法在访问中加载延迟集合。

常见问题　如果Hibernate必须延迟加载对象，为什么它不能打开新的会话？Hibernate Session是持久化上下文，是对象同一性的范围。Hibernate保证在持久化上下文中，一个特定的数据库行最多只有一种内存表示法。按需打开Session，幕后也创建了一个新的持久化上下文，并且在这个同一性范围中加载的所有对象，都将可能与在原始的持久化上下文中加载的对象产生冲突。当对象处于受保护的对象同一性范围之外（即被脱管的时候），你将无法按需加载数据。另一方面，只要对象处于持久化状态中，且由Session管理，那么甚至在原始的事务已经被提交的时候，你都可以加载数据。在这样的场景中，必须启用自动提交模式，如10.3节中所述。建议你不要在Web应用程序中使用自动提交模式，把原始的Session和事务扩展为跨越整个请求更为容易。当对象必须在一个Session内部按需加载时，在你无法轻易启动和终止事务的系统中（例如使用Hibernate的Swing桌面应用程序），自动提交模式就很有用。

最初的解决方案是确保所有必需的关联和集合在转到视图之前被完全初始化（我们稍后讨论它），但是，在一个包含协同定位的表现层和持久层的两层架构中，一种更为方便的方法是保持持久化上下文开着，直到视图完全呈现。

OSIV模式允许你每一个请求都有一个的Hibernate持久化上下文，跨越视图的呈现，且可能跨越多个动作执行execute()。它也可以被轻松地实现——例如，通过servlet过滤器：

```
public class HibernateSessionRequestFilter implements Filter {

  private SessionFactory sf;
  private static Log log = ...;

  public void doFilter(ServletRequest request,
```

```
                        ServletResponse response,
                        FilterChain chain)
            throws IOException, ServletException {

        try {
            // Starting a database transaction
            sf.getCurrentSession().beginTransaction();

            // Call the next filter (continue request processing)
            chain.doFilter(request, response);

            // Commit the database transaction
            sf.getCurrentSession().getTransaction().commit();

        } catch (Throwable ex) {
            // Rollback only
            try {
                if (sf.getCurrentSession().getTransaction().isActive())
                    sf.getCurrentSession().getTransaction().rollback();
            } catch (Throwable rbEx) {
                log.error("Could not rollback after exception!", rbEx);
                rbEx.printStackTrace();
            }

            // Let others handle it...
            throw new ServletException(ex);
        }

    }

    public void init(FilterConfig filterConfig)
                    throws ServletException {
        sf = HibernateUtil.getSessionFactory();
    }

    public void destroy() {}

}
```

这个过滤器就像servlet请求的一个拦截器。每次请求命中服务器且必须被处理时它都运行，它在启动时需要SessionFactory，并从HibernateUtil辅助类中得到它。当请求到达时，你启动一个数据库事务，并打开新的持久化上下文。你在控制器已经执行且视图已经呈现之后，提交数据库事务。由于Hibernate的自动Session绑定和传播，这也自动成为持久化上下文的范围。

异常处理也被集中和封装在这个拦截器中。至于要给数据库事务的回滚捕捉什么异常，就取决于你了；Throwable是捕捉全部（catch-all）的变形，它意味着不仅Exception和RuntimeException，甚至抛出的Error也会触发回滚。注意实际的回滚也可能抛出错误或者异常——始终确保（例如，通过打印出堆栈轨迹）这个次要的异常不会隐藏或者淹没导致回滚的原始问题。

控制器代码现在脱离了事务和异常处理，看起来已经好多了：

```
public void execute() {

    // Get values from request
```

```
        Session session =
            HibernateUtil.getSessionFactory().getCurrentSession();

        // Load requested Item
        // Check auction still valid
        // Check amount of Bid
        // Add new Bid to Item
        // Place new Bid in scope for next page
        // Forward to success page

    }
```

由SessionFactory返回的当前Session，与现在界定到包住这个方法（以及结果页面的呈现）的拦截器的持久化上下文一样。

参考你Web容器的文档，看看如何对特定的URL将这个过滤器类启用为拦截器；建议你只把它应用到执行期间需要数据库访问的URL。否则，就会在服务器中给每一个HTTP请求启动数据库事务和Hibernate Session。这可能会耗尽你的数据库连接池，即使没有把任何SQL语句发送到数据库服务器。

可以用你喜欢的任何方法来实现这个模式，只要有能力拦截请求，并把控制器包装在代码中。许多Web框架提供原生的拦截器，应该使用你所发现的最具吸引力的拦截器。此处介绍利用servlet过滤器的实现并非没有问题。

对Session中的对象所做的改变被不定期地清除到数据库，并且在事务提交时被最终清除。事务提交可能发生在视图呈现之后。问题是servlet引擎的缓冲区大小：如果视图的内容超出缓冲区大小，缓冲区就可能被清除，并且内容被发送到客户端。当内容呈现时，缓冲区可能被多次清除，但是第一次清除也发送HTTP协议状况代码。如果Hibernate清除/提交时的SQL语句在数据库中触发了约束违例，用户可能已经看到成功的输出！无法改变状况代码（例如，用500 Internal Server Error），它已经被发送到客户端（如200 OK）。

有几种方法可以防止这种罕见的异常：调整servlet的缓冲区大小，或者在转发（forward）/重定向（redirect）到视图之前清除Session。有些Web框架没有立即用呈现的内容填充响应缓冲区——它们使用自己的缓冲区，并且只通过视图完全呈现之后的响应清除它，因此我们认为这是简单的Java servlet编程的问题。

让我们继续整理控制器并把业务逻辑抽取到业务层。

16.1.4　设计巧妙的领域模型

MVC模式背后的思想就是控制逻辑（在示例应用程序中，这是一个页面流逻辑）、视图定义，以及业务逻辑应该完全分离。目前，控制器包含一些业务逻辑——在应用程序获得新用户界面这样一个公认不可能的事件中你可能重用的代码——并且领域模型由哑数据持有（dumb data-holding）对象组成。持久化类定义状态，但是不定义行为。

建议你把业务逻辑移到领域模型中，创建一个业务层。这个层的API是领域模型API。这样增加了几行代码，但是它也增加了以后重用的可能，并且更加面向对象，因此提供各种扩展业务逻辑的方法[例如，如果你突然需要实现“最低的出价胜出”，就可以对不同的出价策略使用一个

策略模式（strategy pattern）]。也可以独立于页面流或者任何其他关注点来测试业务逻辑。

首先，把新方法placeBid()添加到Item类：

```
public class Item {
    ...
    public Bid placeBid(User bidder, BigDecimal bidAmount,
                        Bid currentMaxBid, Bid currentMinBid)
        throws BusinessException {

        // Check highest bid (TODO:Strategy pattern?)
        if (currentMaxBid != null &&
            currentMaxBid.getAmount().compareTo(bidAmount) > 0) {
            throw new BusinessException("Bid too low.");
        }

        // Auction still valid
        if ( this.getEndDate().before( new Date() ) )
            throw new BusinessException("Auction already ended");

        // Create new Bid
        Bid newBid = new Bid(bidAmount, this, bidder);
        // Place bid for this Item
        this.addBid(newBid);

        return newBid;
    }

}
```

这段代码基本上执行了所有需要业务对象的状态、但不执行数据访问代码的查找。目的在于把业务逻辑封装在领域模型的类中，而不用依赖任何持久化数据访问或者其他的基础结构。记住，这些类不应该知道关于持久化的任何东西，因为你可能在持久化上下文外部需要它们（例如，在表现层或者逻辑单元测试中）。

把代码从控制器移到领域模型，有一个值得注意的异常。来自旧控制器的代码不能像这样移动：

```
// Check amount of Bid
Query q = session.createQuery("select max(b.amount)" +
                              " from Bid b where b.item = :item");
q.setEntity("item", item);
BigDecimal maxBidAmount = (BigDecimal) q.uniqueResult();

if (maxBidAmount.compareTo(bidAmount) > 0) {
  ...      // Forward to error page
}
```

在实际的应用程序中，你经常会面对同样的情形：业务逻辑与数据访问代码甚至页面流逻辑混合在一起。有时候难以只抽取业务逻辑而不用任何依赖。如果你现在看一下解决方案，在Item.placeBid()方法上引入的currentMaxBid和currentMinBid参数，就会明白如何解决这种问题。页面流和数据访问代码保留在控制器中，但是给业务逻辑提供必要的数据：

```
public void execute() {

    Long itemId = ...            // Get value from request
    Long userId = ...            // Get value from request
```

```
BigDecimal bidAmount = ...  // Get value from request

Session session =
    HibernateUtil.getSessionFactory().getCurrentSession();

// Load requested Item
Item item = (Item) session.load(Item.class, itemId);

// Get maximum and minimum bids for this Item
Query q = session.getNamedQuery(QUERY_MAXBID);
q.setParameter("itemid", itemId);
Bid currentMaxBid = (Bid) q.uniqueResult();

q = session.getNamedQuery(QUERY_MINBID);
q.setParameter("itemid", itemId);
Bid currentMinBid = (Bid) q.uniqueResult();

// Load bidder
User bidder = (User) session.load(User.class, userId);

try {

    Bid newBid = item.placeBid(bidder,
                               bidAmount,
                               currentMaxBid,
                               currentMinBid);

    ...     // Place new Bid into request context

    ...     // Forward to success page

    } catch (BusinessException e) {
        ...     // Forward to appropriate error page
    }

}
```

控制器现在完全不知道任何业务逻辑——它甚至不知道新的出价必须比上一个价格更高还是更低。你已经把所有的业务逻辑都封装在领域模型中，现在可以把业务逻辑作为一个孤立的单元进行测试了，它不依赖于任何动作、页面流、持久化，或者其他的基础结构代码（通过在单元测试中调用`Item.placeBid()`）。

甚至可以通过捕捉和转发特定的异常设计一个不同的页面流。`BusinessException`是一个已声明的和已检查的异常，因此你必须以某种方式在控制器中对它进行处理。在这种情况下是否要回滚事务，或者是否有机会以某种方式恢复，就取决于你了。然而，处理异常时始终考虑你持久化上下文的状态：当你在应用程序异常之后重用同一个`Session`时，可能有来自之前的未被清除的修改试图出现。（当然，可以永远不重用已经抛出致命的运行时异常的`Session`。）安全的做法是始终回滚任何异常中的数据库事务，并用新的`Session`重试。

这个动作代码看起来已经很好了。你应该努力保持架构简单；隔离异常和事务处理并且抽取业务逻辑，这么做会有很大的区别。然而，动作代码现在被绑定到了Hibernate，因为它用`Session` API访问数据库。MVC模型没有提到Persistence的P应该何去何从。

16.2 创建持久层

将数据访问代码和应用程序逻辑混合，这违背了关注点分离的重点。之所以你应该考虑把Hibernate调用隐藏在facade（即所谓的持久层）之后，有几个原因：

- 持久层可以给数据访问操作提供更高级别的抽象。不用基础的CRUD和查询操作，而是公开更高级别的操作，例如`getMaximumBid()`方法。这个抽象就是你为什么想要在更大的应用程序中创建持久层的主要原因：支持重用相同的非CRUD操作。
- 持久层可以有泛型的接口，不公开实际的实现细节。换句话说，可以隐瞒你正在使用Hibernate（或者Java Persistence）从持久层的任何客户端实现数据访问操作的事实。我们认为持久层的可移植性是一个不重要的关注点，因为完全的ORM解决方案（如Hibernate）已经提供了数据库的可移植性。你不太可能在未来使用不同的软件重新编写持久层，而仍然不想改变任何客户端代码。此外，把Java Persistence当作标准的、可以完全移植的API。
- 持久层可以统一数据访问操作。这个关注点与可移植性相关，但是从一个稍微不同的角度来看。想象你必须处理混合的数据访问代码，例如Hibernate和JDBC操作。通过统一客户端看到和使用的facade，你可以从客户端隐藏这个实现细节。

如果把可移植性和统一当作创建持久层的附带作用，主要目的在于实现更高级别的抽象和改进后的可维护性，以及数据访问代码的重用。这些都是很好的理由，我们鼓励创建这样的持久层：除了最简单的应用程序之外，全部用一般的facade。系统不要过于细致（overengineer），并且首先考虑直接使用Hibernate（或者JPA）而不用任何其他分层，这一点仍然很重要。假设你想要创建一个持久层，并设计一个客户端将调用的facade。

设计持久层Facade的方法不止一种——有些小应用程序可以使用单个`PersistenceManager`对象；有些可以使用某种面向命令的设计，其他的则把数据访问操作混合到领域类（活动的记录）——但我们更喜欢DAO模式。

16.2.1 泛型的数据访问对象模式

DAO设计模式源于Sun公司的Java Blueprint。它甚至用在臭名昭著的Java Petstore演示应用程序中。DAO为与特定的持久化实体相关的持久化操作（CRUD和finder方法）定义了接口，它建议你把与该实体的持久化相关的代码都组合到一起。

使用JDK 5.0特性如泛型（generic）和变量参数，可以轻松地设计一个很好的DAO持久层。这里建议的模式基本结构如图16-1所示。

我们设计了包含两个平行层次结构的持久层：接口在一侧，实现在另一侧。基本的对象存储和对象获取操作被分在泛型的超接口（generic superinterface），以及由于利用特定的持久化解决方案（我们用Hibernate）而实现这些操作的超类中。泛型接口通过接口进行扩展，用于需要额外的业务相关的数据访问操作的特定实体。你可能再次拥有实体DAO接口的一个或者几个实现。

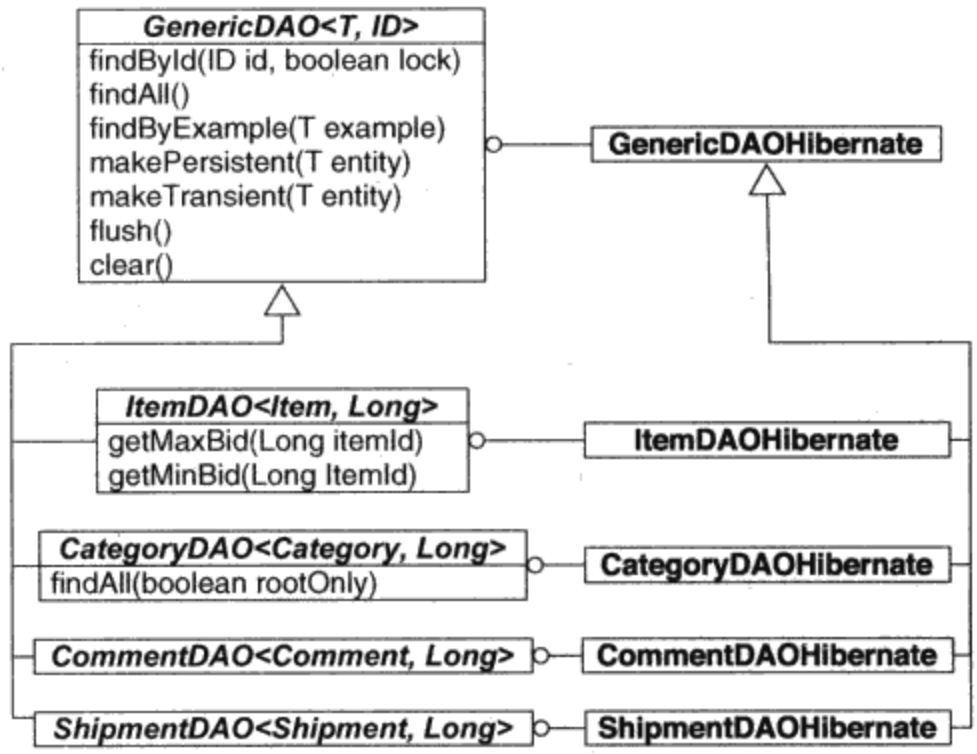


图16-1　泛型DAO接口支持任意的实现

首先考虑每个实体共享和需要的基础CRUD操作，你把这些操作组合在泛型超接口中：

```
public interface GenericDAO<T, ID extends Serializable> {

    T findById(ID id, boolean lock);

    List<T> findAll();

    List<T> findByExample(T exampleInstance,
                          String... excludeProperty);

    T makePersistent(T entity);

    void makeTransient(T entity);

    void flush();

    void clear();

}
```

GenericDAO是一个需要类型参数的接口，如果想要实现它的话。第一个参数T，是你正在为其实现DAO的实体实例。其中许多DAO方法以类型安全的方式使用这个参数来返回对象。第二个参数定义数据库标识符的类型——并非所有的实体都可以给它们的标识符属性使用相同的类型。这里值得关注的第二个东西是findByExample()方法中的变量参数，你很快就会看到它是如何为客户端改善API的。

最后，这无疑是持久层面向状态（state-oriented）工作的基础。一些方法如makePersistent()和makeTransient()改变一个对象（或者同时改变启用了级联的许多对象）的状态。flush()和clear()操作可以被客户端用来管理持久化上下文。如果你的持久层是面向语句的（statement-oriented），就得编写一个完全不同的DAO接口。例如，如果你没有用Hibernate而只用了简单的JDBC实现它。

这里介绍的持久层facade不公开任何Hibernate或者Java Persistence接口给客户端，因此理论上可以通过任何软件实现它，而不用对客户端代码做任何改变。你可能不想或者不需要持久层的可移植性，如前所述。在这种情况下，应该考虑公开Hibernate或者Java Persistence接口——例如，客户端可以用来执行任意的Hibernate Criteria查询的findByCriteria(DetachedCriteria)方法。这个由你决定。你可能觉得，公开Java Persistence接口相对于公开Hibernate接口来说，是一种更安全的选择。然而，你应该知道，虽然可能把持久层的实现从Hibernate变到Java Persistence，或者变到任何其他全特性的面向状态的ORM软件，但是几乎不可能用简单的JDBC语句重写面向状态的持久化层。

接下来，实现DAO接口。

16.2.2 实现泛型 CRUD 接口

用Hibernate API继续探讨泛型接口的可能实现：

```
public abstract class
        GenericHibernateDAO<T, ID extends Serializable>
            implements GenericDAO<T, ID> {

    private Class<T> persistentClass;
    private Session session;

    public GenericHibernateDAO() {
        this.persistentClass = (Class<T>)
            ( (ParameterizedType) getClass().getGenericSuperclass() )
                .getActualTypeArguments()[0];
     }

    public void setSession(Session s) {
        this.session = s;
    }

    protected Session getSession() {
        if (session == null)
            session = HibernateUtil.getSessionFactory()
                                   .getCurrentSession();
        return session;
    }

    public Class<T> getPersistentClass() {
        return persistentClass;
    }

    ...
```

目前为止，这是使用Hibernate实现的内部管路系统。在这个实现中，你需要访问一个Hibernate Session，因此需要DAO的客户端通过设置方法来注入当前它想要使用的Session。这在集成测试中最有用。如果客户端在使用DAO之前没有设置一个Session，那么当DAO代码需要时，你就得查找当前的Session。

DAO实现也必须知道它用于哪些持久化实体类。在构造器中要使用Java反射（Java Reflection），来查找泛型参数T的类，并把它保存在一个本地的成员中。

如果用Java Persistence编写一个泛型DAO实现，那么代码看起来几乎一样。唯一的改变是DAO需要EntityManager，而不是Session。

现在可以实现实际的CRUD操作，还是用Hibernate：

```
@SuppressWarnings("unchecked")
public T findById(ID id, boolean lock) {
    T entity;
    if (lock)
        entity = (T) getSession()
            .load(getPersistentClass(), id, LockMode.UPGRADE);
    else
        entity = (T) getSession()
            .load(getPersistentClass(), id);

    return entity;
}

@SuppressWarnings("unchecked")
public List<T> findAll() {
    return findByCriteria();
}

@SuppressWarnings("unchecked")
public List<T> findByExample(T exampleInstance,
                             String... excludeProperty) {
    Criteria crit =
        getSession().createCriteria(getPersistentClass());
    Example example =  Example.create(exampleInstance);
    for (String exclude : excludeProperty) {
        example.excludeProperty(exclude);
    }
    crit.add(example);
    return crit.list();
}

@SuppressWarnings("unchecked")
public T makePersistent(T entity) {
    getSession().saveOrUpdate(entity);
    return entity;
}

public void makeTransient(T entity) {
    getSession().delete(entity);
}

public void flush() {
    getSession().flush();
}

public void clear() {
    getSession().clear();
}

/**
 * Use this inside subclasses as a convenience method.
 */
```

```
@SuppressWarnings("unchecked")
protected List<T> findByCriteria(Criterion... criterion) {
    Criteria crit =
        getSession().createCriteria(getPersistentClass());
    for (Criterion c : criterion) {
        crit.add(c);
    }
    return crit.list();
  }

}
```

所有的数据访问操作都用getSession()获得被分配到这个DAO的Session。这些方法中大多数都很简单，在读了本书前面的章节之后，理解它们应该不成问题了。@SurpressWarning注解是可选的——Hibernate接口是给JDK 5.0之前的版本编写的，因此所有的转换都是未经检查的，否则JDK 5.0编译器就会给每一个转换都生成警告。看一下受保护的findByCriteria()方法：我们认为这是一种使其他数据访问操作的实现变得更容易的便利方法。它采用0个或者更多个Criterion参数，并把它们添加到随后要被执行的Criteria。这是JDK 5.0变量参数的一个示例。注意，我们决定不在公共的泛型DAO接口上公开这个方法；它是一个实现细节（你可能得出一个不同的结论）。

利用Java Persistence实现很简单，虽然它不支持Criteria API。你不用saveOrUpdate()，而是用merge()使任何瞬时的或者脱管的对象变成持久化，并返回合并后的结果。

你现在已经完成了持久层的基础部件和它公开给系统上层的泛型接口。下一步，你要创建实体相关的DAO接口，并通过扩展泛型的接口和实现来实现它们。

16.2.3 实现实体 DAO

假设你想要给Item业务实体实现非CRUD的数据访问操作。首先，编写一个接口：

```
public interface ItemDAO extends GenericDAO<Item, Long> {

    Bid getMaxBid(Long itemId);
    Bid getMinBid(Long itemId);

}
```

ItemDAO接口扩展了泛型的超接口，并用一个Item实体类型和一个作为数据库标识符类的Long，把它参数化。两个数据访问操作与Item实体相关：getMaxBid()和getMinBid()。

这个接口利用Hibernate的实现，扩展泛型的CRUD实现：

```
public class ItemDAOHibernate
        extends     GenericHibernateDAO<Item, Long>
        implements  ItemDAO {

    public Bid getMaxBid(Long itemId) {
        Query q = getSession().getNamedQuery("getItemMaxBid");
        q.setParameter("itemid", itemId);
        return (Bid) q.uniqueResult();
    }
```

```
    public Bid getMinBid(Long itemId) {
        Query q = getSession().getNamedQuery("getItemMinBid");
        q.setParameter("itemid", itemId);
        return (Bid) q.uniqueResult();
    }

}
```

你可以看到，有了超类提供的功能，这个实现是多么容易。查询已经被外部化到映射元数据，并且按名称被调用，这避免了弄乱代码。

建议你甚至为没有任何非CRUD数据访问操作的实体创建接口：

```
public interface CommentDAO extends GenericDAO<Comment, Long> {
    // Empty
}
```

这个实现相当简单：

```
public static class CommentDAOHibernate
        extends GenericHibernateDAO<Comment, Long>
        implements CommentDAO {}
```

我们推荐这个空接口和实现，因为你无法实例化泛型的抽象实现。此外，客户端应该依赖专用于特定实体的接口，这样避免了如果引入其他的数据访问操作而导致未来高成本的重构。但是，你可能不会接受我们的建议，从而使GenericHibernateDAO变成非抽象的。这个决定取决于你正在编写的应用程序，以及你希望未来会做的哪些改变。

把所有这些都放在一起，看看客户端如何实例化并使用DAO。

16.2.4　利用数据访问对象

如果客户端希望利用持久层，它必须实例化它需要的DAO，然后在这些DAO中调用方法。在前面引入的Hibernate Web应用程序的用例中，控制器和动作代码看起来像这样：

```
public void execute() {

    Long itemId = ...              // Get value from request
    Long userId = ...              // Get value from request
    BigDecimal bidAmount = ...     // Get value from request

    // Prepare DAOs
    ItemDAO itemDAO = new ItemDAOHibernate();
    UserDAO userDAO = new UserDAOHibernate();

    // Load requested Item
    Item item = itemDAO.findById(itemId, true);

    // Get maximum and minimum bids for this Item
    Bid currentMaxBid = itemDAO.getMaxBid(itemId);
    Bid currentMinBid = itemDAO.getMinBid(itemId);

    // Load bidder
    User bidder = userDAO.findById(userId, false);

    try {

        Bid newBid = item.placeBid(bidder,
```

```
                                        bidAmount,
                                        currentMaxBid,
                                        currentMinBid);

            ...     // Place new Bid into request context

            ...     // Forward to success page

        } catch (BusinessException e) {
            ...     // Forward to appropriate error page
        }

    }
```

几乎设法避免在控制器中依赖任何的Hibernate代码了，除了这一件事之外：你仍然需要在控制器中实例化一个特定的DAO实现。避免这种依赖的一种（并不复杂的）方法是传统的抽象工厂模式。

首先，给数据访问对象创建一个抽象的工厂：

```
public abstract class DAOFactory {

    /**
     * Factory method for instantiation of concrete factories.
     */
    public static DAOFactory instance(Class factory) {
        try {
            return (DAOFactory)factory.newInstance();
        } catch (Exception ex) {
            throw new RuntimeException(
                    "Couldn't create DAOFactory: " + factory
                    );
        }
    }

    // Add your DAO interfaces here
    public abstract ItemDAO getItemDAO();
    public abstract CategoryDAO getCategoryDAO();
    public abstract CommentDAO getCommentDAO();
    public abstract UserDAO getUserDAO();
    public abstract BillingDetailsDAO getBillingDetailsDAO();
    public abstract ShipmentDAO getShipmentDAO();

}
```

这个抽象的工厂可以构建和返回任何DAO。现在给你的Hibernate DAO实现这个工厂：

```
public class HibernateDAOFactory extends DAOFactory {

    public ItemDAO getItemDAO() {
        return (ItemDAO) instantiateDAO(ItemDAOHibernate.class);
    }

    ...

    private GenericHibernateDAO instantiateDAO(Class daoClass) {
        try {
            GenericHibernateDAO dao = (GenericHibernateDAO)
                                      daoClass.newInstance();
```

```
            return dao;
        } catch (Exception ex) {
            throw new RuntimeException(
                        "Can not instantiate DAO: " + daoClass, ex
                    );
        }
    }

    // Inline all empty DAO implementations

    public static class CommentDAOHibernate
            extends GenericHibernateDAO<Comment, Long>
            implements CommentDAO {}

    public static class ShipmentDAOHibernate
            extends GenericHibernateDAO<Shipment, Long>
            implements ShipmentDAO {}
    ...

}
```

这里发生了几件值得关注的事情。第一，工厂的实现封装了实例化DAO的方式。可以定制这个方法，并在返回DAO实例之前手工设置一个Session。

其次，把CommentDAOHibernate的实现作为公共的静态类移到工厂里面。记住，你需要这个实现（即使它是空的）来让客户端使用与实体相关的接口。然而，没有人强迫你在单独的文件中创建几十个空的实现类，可以在工厂中分组所有的空实现。如果未来必须给Comment实体引入更多的数据访问操作，就把这个实现从工厂移到它自己的文件中去。不需要改变其他的代码——客户端只依赖CommentDAO接口。

利用这个工厂模式，可以进一步简化DAO在Web应用程序控制器中的使用方法：

```
public void execute() {

    Long itemId = ...             // Get value from request
    Long userId = ...             // Get value from request
    BigDecimal bidAmount = ...    // Get value from request

    // Prepare DAOs
    DAOFactory factory = DAOFactory.instance(DAOFactory.HIBERNATE);
    ItemDAO itemDAO = factory.getItemDAO();
    UserDAO userDAO = factory.getUserDAO();

    // Load requested Item
    Item item = itemDAO.findById(itemId, true);

    // Get maximum and minimum bids for this Item
    Bid currentMaxBid = itemDAO.getMaxBid(itemId);
    Bid currentMinBid = itemDAO.getMinBid(itemId);

    // Load bidder
    User bidder = userDAO.findById(userId, false);

    try {
        ...
    }

}
```

唯一依赖Hibernate，并把持久层的真正实现公开给客户端代码的唯一一行代码，是DAOFactory的获取。你可能想要考虑把这个参数移到应用程序的外部配置中，以便可能转换DAOFactory实现，而不用改变任何代码。

提示 在一个DAO中混用Hibernate和JDBC代码——当你可以使用Hibernate时，很少需要使用简单的JDBC。记住，如果你需要JDBC Connection来执行Hibernate无法自动生成的语句，始终可以退回来使用session.connection()。因此，我们认为你不需要给少数JDBC调用使用不同的和单独的DAO。混合使用Hibernate和简单的JDBC的问题，不在于你有时候可能不得不这么做（不应该指望Hibernate会解决你的全部问题），而在于开发人员经常试图隐藏他们做过的事情。只要形成适当的文档，混合的数据访问代也没有问题。还要记住，Hibernate支持几乎所有利用原生API的SQL操作，因此你不必退回到简单的JDBC。

你现在已经创建了一个整洁、灵活且强大的持久层，它从任何客户端代码中隐藏了数据访问的细节。你可能仍然存有下列疑虑：

- 必须编写工厂吗？工厂模式是传统的，用在主要依赖无状态服务查找的应用程序中。另一种可选的（或者有时是补充的）策略是依赖注入（dependency injection）。EJB 3.0规范给托管组件标准化了依赖注入，因此本章稍后将讨论另一种DAO编写策略。
- 必须给每个领域实体创建DAO接口吗？我们的建议并不涵盖所有可能的情况。在更大的应用程序中，你可能想要按领域包来分组DAO，或者创建更深的DAO层次结构（它给特定的几个子实体提供更细粒度的特殊化）。DAO模式有许多种变形，你不应该受限于我们所建议的泛型解决方案。放心地去体验吧，并把这个模式当作一个好的起点。

你现在知道如何在传统的Web应用程序中整合Hibernate，以及如何根据最佳的实践模式创建持久层。如果必须设计和编写一个三层的应用程序，则需要考虑一个完全不一样的架构了。

16.3 命令模式简介

如果必须利用Hibernate和Java Persistence编写中小型的Web应用程序，前几节中介绍过的模式和策略就非常完美。OSIV模式适用于两层的架构，其中表现层、业务层和持久层都协同分布在同一台虚拟机上。

然而，一旦你引入第三层，并把表现层移到一台单独的虚拟机上，当前的持久化上下文就无法再保持打开状态，直到视图完全呈现。这是三层的EJB应用程序或者单独的进程中包含富客户端的架构中的典型案例。

如果表现层在一个不同的进程中运行，就要把这个进程与运行应用程序的业务层和持久层的层之间的请求减到最少。这意味着你不能使用前面的延迟方法（必要时它允许视图从领域模型对象中拖取数据）。反之，业务层必须接受抓取后面呈现视图所需的所有数据的责任。

虽然某些模式可以把远程通信减到最少[如会话外观（session facade）和数据传输对象（data transfer object，DTO）模式]，并在Java开发社区中已经得到广泛应用，但我们还是想讨论一种稍

微不同的方法。命令（Command）模式（经常被称作EJB命令）是一种复杂的解决方案，它结合了其他策略的优势。

我们来编写一个利用这种模式的三层应用程序。

16.3.1　基础接口

命令模式以命令类的层次结构思想为基础，所有命令类都实现一个简单的Command接口。看一下图16-2中的这个层次结构。

特定的Command是动作、事件或者符合类似描述的任何东西的实现。客户端代码创建命令对象，并准备执行它们。CommandHandler是一个可以执行Command对象的接口。客户端把Command对象传递到服务器层中的一个处理器，这个处理器执行它。然后Command对象被返回到客户端。

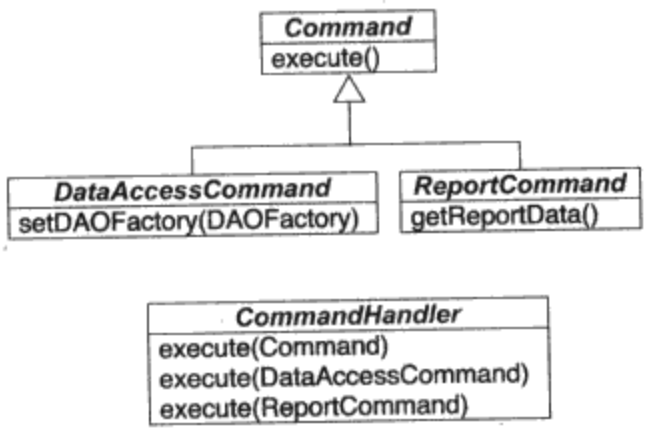


图16-2　命令模式的接口

Command接口有一个execute()方法；任何具体的命令都必须实现这个方法。任何子接口都可以添加在执行Command之前（设置方法）或者之后（获取方法）而被调用的额外方法。因此Command给特定的事件组合了输入（input）、控制器（controller）和输出（output）。

执行Command对象（即，调用它们的execute()方法）是CommandHandler实现的工作。命令的执行被多态地分派。

这些接口（和抽象类）的实现可以看作如下：

```
public interface Command {
    public void execute() throws CommandException;
}
```

Command也封装异常处理，以便执行期间抛出的任何异常都被包装在CommandException（之后可以被客户端相应处理）中。

DataAccessCommand是一个抽象类：

```
public abstract class DataAccessCommand implements Command {
    protected DAOFactory daoFactory;

    public void setDAOFactory(DAOFactory daoFactory) {
        this.daoFactory = daoFactory;
    }

}
```

任何需要访问数据库的Command都必须使用一个数据访问对象，因此DAOFactory必须在DataAccessCommand可以被执行之前进行设置。这通常是CommandHandler实现的工作，因为持久层处在服务器层中。

命令处理器的远程接口相当简单：

```
public interface CommandHandler {

    public Command executeCommand(Command c)
                    throws CommandException;

    public DataAccessCommand executeCommand(DataAccessCommand c)
                    throws CommandException;

    public Reportcommand executeCommand(ReportCommand c)
                    throws CommandException;
}
```

下面编写一些具体的实现并使用命令。

16.3.2 执行命令对象

希望执行命令的客户端需要实例化并准备一个Command对象。例如，给一次拍卖出价需要客户端上的BidForItemCommand：

```
BidForItemCommand bidForItem =
            new BidForItemCommand(userId, itemId, bidAmount);
try {
    CommandHandler handler = getCommandHandler();

    bidForItem = (BidForItemCommand)handler.execute Command (bidForItem);

    // Extract new bid for rendering
    newBid = bidForItem.getNewBid();

    // Forward to success page

} catch (CommandException ex) {
    // Forward to error page
    // ex.getCause();

}
```

对于这个动作，BidForItemCommand需要所有的输入值作为构造器参数。然后客户端查找一个命令处理器，并传递BidForItemCommand对象用于执行。处理器在执行之后返回实例，客户端从返回的对象中提取任何输出值。（如果你使用JDK 5.0，就用泛型来避免不安全的类型转换。）

命令处理器如何被查找或者实例化，这取决于命令处理器的实现，以及远程的通信如何发生。你甚至不必调用远程的命令处理器——它可以是一个本地的对象。

来看一下命令和命令处理器的实现。

1. 实现业务命令

BidForItemCommand扩展抽象类DataAccessCommand，并实现execute()方法：

```
public class BidForItemCommand extends DataAccessCommand
                               implements Serializable {

    // Input
    private Long userId;
    private Long itemId;
    private BigDecimal bidAmount;

    // Output
    private Bid newBid;

    public BidForItemCommand(Long userId,
                             Long itemId,
                             BigDecimal bidAmount) {
        this.userId = userId;
        this.itemId = itemId;
        this.bidAmount = bidAmount;
    }

    public Bid getNewBid() {
        return newBid;
    }

    public void execute() throws CommandException {

        ItemDAO itemDAO = daoFactory.getItemDAO();
        UserDAO userDAO = daoFactory.getUserDAO();

        try {

            Bid currentMaxBid = itemDAO.getMaxBid(itemId);
            Bid currentMinBid = itemDAO.getMinBid(itemId);

            Item item = itemDAO.findById(itemId, false);
            newBid = item.placeBid(userDAO.findById(userId, false),
                                   bidAmount,
                                   currentMaxBid,
                                   currentMinBid);

        } catch (BusinessException ex) {
            throw new CommandException(ex);
        }
    }

}
```

这基本上与你在本章前面修饰Web应用程序的最后阶段中所写的代码一样。然而，利用这种方法，你对于一次拍卖所需的输入和返回的输出有着清晰的约定。

因为Command实例是跨越网络进行传送，所以你必须实现Serializable（这个标识应该处在具体类中，而不是在超类或者接口中）。

我们来实现命令处理器。

2. 实现命令处理器

命令处理器可以用你喜欢的任何方式实现；它的责任很简单。许多系统只需要单个命令处理器，例如：

```
@Stateless
public class CommandHandlerBean implements CommandHandler {

    // The persistence layer we want to call
    DAOFactory daoFactory =
        DAOFactory.instance(DAOFactory.HIBERNATE);

    @TransactionAttribute(TransactionAttributeType.NEVER)
    public Command executeCommand(Command c)
            throws CommandException {
        c.execute();
        return c;
    }

    @TransactionAttribute(TransactionAttributeType.REQUIRED)
    public Command executeCommand(DataAccessCommand c)
            throws CommandException {
        c.setDAOFactory(daoFactory);
        c.execute();
        return c;
    }
}
```

这是一个实现为无状态的EJB 3.0会话bean的命令处理器。你在客户端用一个EJB查找获得对这个（本地或者远程）bean的引用，然后将Command对象传递给它用于执行。处理器知道如何准备一个特定的命令类型——例如，通过在执行之前设置对持久层的一个引用。

由于容器托管的事务和声明性事务，这个命令处理器没有包含任何Hibernate代码。当然，你也可以将这个命令处理器实现为没有EJB 3.0注解的一个POJO，并编程式地管理事务范围。另一方面，因为EJB支持开箱即用的远程通信，它们是三层架构中最佳的命令处理器选择。

这个基本的命令模式还有更多的变形。

16.3.3 命令模式的变形

首先，命令模式也并非十全十美。这种模式可能最重要的问题是，在客户端类路径中需要非表现接口。由于BidForItemCommand需要DAO，你必须在客户端的类路径中包括持久层接口（即使命令只在中间层执行）。没有真正的解决方案，因此这个问题的严重性取决于你的部署场景，以及你可以多轻松地给应用程序相应地打包。注意，客户端只需要DAO接口来实例化DataAccessCommand，因此，你或许可以在实现持久层之前稳定接口。

而且，由于你只有一个命令，命令模式似乎比传统的会话外观模式喜欢更多的工作。然而，随着系统的成长，因为横切关注点如异常处理和验证检查可以在命令处理器中实现，因此增加新命令变得更加简单。命令很容易实现，并且极易重用。你不应该受限于我们所建议的命令接口层次结构，放心地去设计更复杂且更成熟的命令接口和抽象的命令吧。你也可以利用委托把命令分组在一起——例如，DataAccessCommand可以实例化并调用ReportCommand。

对于呈现特定的视图所需的数据来说，命令是很好的装配器。不用让视图从延迟加载的业务对象中拖取信息（这需要表现层和持久层的协同定位，因此你可以留在同一个持久化上下文中），而是客户端可以准备和执行呈现特定屏幕所需的命令——每一个命令都把数据传输到它输出属

性中的表现层。命令有点像是一种带有内建装配子程序的数据迁移对象（data-transfer object）。

此外，命令模式让你能够轻松地实现任何撤销（Undo）功能。每一个命令都可以有一个undo()方法，它可以禁用execute()方法已经完成的任何永久的改变。或者，你可以在客户端排列几个命令对象，并且只在特定的对话完成时，才把它们发送到命令处理器。

如果必须实现一个桌面应用程序，命令模式也很好。例如，可以实现在数据改变时触发事件的一个命令。通过在命令处理器中注册一个监听器，所有需要被刷新的对话框都可以监听到这个事件。

可以用EJB 3.0拦截器把命令包装起来。例如，对于能够在特定类型的命令对象上透明地注入特定服务的命令处理器会话bean，可以为它编写拦截器。可以在命令处理器中组合和堆叠这些拦截器。甚至可以实现一个客户端-本地命令处理器，由于EJB拦截器，它可以透明地决定命令是否需要被发送到服务器（到另一个命令处理器），或者命令是否可以在客户端断开连接的情况下执行。

无状态会话bean不仅仅需要命令处理器。实现一个异步执行命令的基于JMS的命令处理器很容易。甚至可以在数据库中给计划好的执行保存命令。命令可以在服务器环境的外部使用——比如在一个批量处理或者单元测试案例中。

在实践中，依赖命令模式的架构运行得相当好。

下一节将讨论EJB 3.0组件如何进一步简化分层的应用程序架构。

16.4　利用 EJB 3.0 设计应用程序

本书已经关注过Java Persistence标准，并只讨论过几个其他EJB 3.0编程构造的例子。我们编写了一些EJB会话bean，启用了容器托管的事务，并使用容器注入获得EntityManager。

在EJB 3.0编程模型中，还有更多的东西等着去发现。接下来的几节要介绍如何简化前面利用EJB 3.0组件的一些模式。然而，我们仍然只讨论数据库应用程序相关的特性，因此如果你想要了解更多有关定时器、EJB拦截器或者消息驱动的EJB的信息，要参考其他的文档。

首先，你将在一个包含有状态会话bean（对话控制器）的Web应用程序中实现一个动作。然后简化数据访问对象（通过把它们变成EJB）来获得容器托管的事务和依赖注入。你也将从任何Hibernate接口转换到Java Persistence，以保持与EJB 3.0的完全兼容。

通过在Web应用程序中利用EJB 3.0组件实现对话来开始下面的内容。

16.4.1　利用有状态的 bean 实现会话

对于应用程序和用户之间可能长运行的对话来说，有状态会话bean（SFSB）是最完美的控制器。可以编写一个在一个对话（如PlaceItem对话）中实现所有步骤的SFSB：

(1) 用户输入货品信息；

(2) 用户可以给一件货品添加图片；

(3) 用户提交完成的表格。

如果必须添加的图片不止一张，可以重复执行这个对话的第2步。让我们通过一个直接使用Java Persistence和EntityManager的SFSB来实现它：

单一的SFSB实例负责整个对话。首先，下面是业务接口：

```
public interface PlaceItem {

    public Item createItem(Long userId, Map itemData);

    public void addImage(String filename);

    public void submit();
}
```

在对话的第一步中，用户输入基础的货品细节并提供用户标识符。从这里，创建一个Item实例，并保存在对话中。然后用户可以多次执行addImage()事件。最后，用户完成了表格，并调用submit()方法来终止对话。注意你如何能够阅读这个接口就像阅读对话的历史一样。

这是一种可能的实现：

```
@Stateful
@TransactionAttribute(TransactionAttributeType.NEVER)
public class PlaceItemBean implements PlaceItem {

    @PersistenceContext(type = PersistenceContextType.EXTENDED)
    private EntityManager em;

    private Item item;
    private User seller;

    public Item createItem(Long userId, Map itemData) {

        // Load seller into conversation
        seller = em.find(User.class, userId);

        // Create item for conversation
        item = new Item(itemData, seller);
        seller.addItem(item);

        return item;
    }

    public void addImage(String filename) {
        item.getImages().add(filename);
    }

    @Remove
    @TransactionAttribute(TransactionAttributeType.REQUIRED)
    public void submit() {
        em.persist(item);
    }
}
```

这个有状态会话bean的实例被绑定到一个特定的EJB客户端，因此它在对话期间也表现得像一个高速缓存。你使用一个只在submit()返回时被清除的扩展持久化上下文，因为这是在事务内部执行的唯一方法。其他方法中的所有数据访问都在自动提交模式下运行。因此em.find(User.class,userId)执行非事务的，反之em.persist(item)则是事务的。由于submit()方法也用@Remove作标识，因此当这个方法返回时，持久化上下文自动关闭，有状态会话bean也被销毁了。

这个实现的一种变形不直接调用EntityManager，而是调用数据访问对象。

16.4.2　利用 EJB 编写 DAO

数据访问对象是完美的无状态会话bean。每一个数据访问方法都不需要任何状态，它只需要EntityManager。因此，当用Java Persistence实现GenericDAO时，就需要设置EntityManager：

```
public abstract class GenericEJB3DAO<T,ID extends Serializable>
        implements GenericDAO<T, ID> {

    private Class<T> entityBeanType;

    private EntityManager em;

    public GenericEJB3DAO() {
        this.entityBeanType = (Class<T>)
          ( (ParameterizedType) getClass().getGenericSuperclass() )
                .getActualTypeArguments()[0];
    }

    @PersistenceContext
    public void setEntityManager(EntityManager em) {
        this.em = em;
    }

    protected EntityManager getEntityManager() {
        return em;
    }

    public Class<T> getEntityBeanType() {
        return entityBeanType;
    }

    ...
}
```

这实际上与前面你在16.2.2节中给Hibernate创建的实现是一样的。然而，你用@PersistenceContext给setEntityManager()方法作了标识，因此当这个bean在一个容器内部执行时，就得到正确的EntityManager的自动注入。如果它在EJB 3.0运行时容器之外执行，你可以手工设置EntityManager。

我们不介绍利用JPA的所有CRUD操作的实现，你应该能够自己实现findById()等。

下面是一个带有业务数据访问方法的具体DAO的实现：

```
@Stateless
@TransactionAttribute(TransactionAttributeType.REQUIRED)
public class ItemDAOBean extends GenericEJB3DAO<Item, Long>
                         implements ItemDAO {

    public Bid getMaxBid(Long itemId) {
        Query q = getEntityManager()
                    .createNamedQuery("getItemMaxBid");
        q.setParameter("itemid", itemId);
        return (Bid) q.getSingleResult();
    }
```

```
    public Bid getMinBid(Long itemId) {
        Query q = getEntityManager()
                    .createNamedQuery("getItemMinBid");
        q.setParameter("itemid", itemId);
        return (Bid) q.getSingleResult();
    }
    ...
}
```

这个具体的子类是个无状态EJB会话bean，被调用的所有方法（包括了从GenericDAO超类继承来的那些方法）都需要一个事务上下文。如果这个DAO的客户端调用没有活动事务的方法，就会给这个DAO方法启动一个事务。

不再需要任何DAO工厂。前面编写过的对话控制器通过依赖注入自动与DAO连接。

16.4.3 利用依赖注入

现在重构PlaceItem对话控制器，并增加一个持久层。不直接访问JPA，而是调用在运行时被容器注入到对话控制器中的几个DAO：

```
@Stateful
public class PlaceItemWithDAOsBean implements PlaceItem {

    @PersistenceContext(
        type = PersistenceContextType.EXTENDED,
        properties =
            @PersistenceProperty(
                name="org.hibernate.flushMode",
                value="MANUAL"
            )
    )
    private EntityManager em;

    @EJB ItemDAO itemDAO;
    @EJB UserDAO userDAO;

    private Item item;
    private User seller;

    public Item createItem(Long userId, Map itemData) {

        // Load seller into conversation
        seller = userDAO.findById(userId);

        // Create item for conversation
        item = new Item(itemData, seller);

        return item;
    }

    public void addImage(String filename) {
        item.getImages().add(filename);
    }

    @Remove
    public void submit() {
```

```
            itemDAO.makePersistent(item);
            em.flush();
        }
    }
```

@EJB注解给自动的依赖注入标识了itemDAO和userDAO字段。容器查找指定接口的一个实现（这个实现是依赖供应商的，但是在这个例子中，每个接口只有一个），并在字段中对它进行设置。

你还没有在这个实现中禁用事务，只是通过Hibernate org.hibernate.flushmode扩展属性禁用了自动清除。然后当SFSB中标有@Remove的方法结束，并在这个方法的事务提交之前清除持久化上下文一次。

这么做有下面两个原因：

- 你正在调用的所有DAO方法都需要一个事务上下文。如果没有在对话控制器中给每个方法启动一个事务，事务范围就是其中一个数据访问对象中的调用。然而，你想要createItem()、addImages()和submit()方法成为事务的范围，以防万一执行几个DAO操作。
- 你有一个被扩展的持久化上下文，它被自动界定和绑定到有状态会话bean。因为DAO是无状态的会话bean，所以只有当事务上下文是活动的并且也被传播时，这个单独的持久化上下文才可以被传播到所有的DAO里面。如果DAO是有状态的会话bean，你可以通过实例化（甚至在没有事务上下文用于DAO调用的时候）传播当前的持久化上下文，但是这也意味着对话控制器必须手工销毁任何有状态的DAO。

如果没有Hibernate扩展属性，你必须使DAO变成有状态会话bean，以允许在非事务的方法调用之间传播持久化上下文。那么在控制器的@Remove方法中调用每个DAO的@Remove方法，就是它自己的责任了——这两种情况都不是你想要的。你想要禁用清除，又不用编写任何非事务的方法。

EJB 3.0包括更多的注入特性，并且它们扩展到其他的Java EE 5.0规范。例如，可以在一个Java servlet容器中使用@EJB注入，或者@Resource，自动从被注入的JNDI中获得任何指定的资源。但是这些特性都不是本书讨论的范围。

既然已经创建了应用层，就需要一种方法来测试它们的正确性。

16.5　测试

测试可能是Java开发人员一天的工作当中最重要的一项活动。测试从功能以及性能和可伸缩性方面确定系统的正确性。测试执行成功意味着所有的应用程序组件和各层都正确地进行交互，并且如指定的那样一起顺利地运行。

可以用多种不同的方法测试和证明一套软件系统。在持久化上下文和数据管理中，一般最关注自动化测试。在接下来的几节中，你将创建许多种测试，你可以重复运行它们来检查应用程序的正确行为。

先来看一下不同的测试种类。功能测试、集成测试和独立的单元测试，它们全部都有着不同的目标和用途，你要知道每种策略什么时候最适当。然后我们编写测试并引入TestNG框架（http://www.testng.org）。最后，考虑压力和负载测试，以及如何查明系统是否要伸缩为大量的并发事务。

16.5.1 理解不同种类的测试

划分软件测试的种类如下：

- 验收测试（acceptance testing）——这种测试不是自动化测试必需的，并且通常不是应用程序开发人员和系统设计人员的工作。验收测试是系统测试的最后阶段，由确定系统是否满足项目需求的客户（或者任何其他方）来执行。这些测试可以包括任何准则，从功能到性能到可用性。
- 性能测试（performance testing）——压力测试或负载测试使用大量并发的用户对系统进行测试，理想情况下是等于或者高于一旦软件正式运行时所期待的负载。因为对于任何使用在线事务数据处理的应用程序来说，这个测试是如此重要的一个方面，因此我们稍后会深入讨论性能测试。
- 逻辑单元测试（logic unit testing）——这些测试考虑单项功能，经常只是一个业务方法（例如，最高的出价是否真的在拍卖系统中胜出）。如果一个组件作为单个单元进行测试，它就是独立于任何其他的组件进行测试。逻辑单元测试不涉及任何像数据库这样的子系统。
- 集成单元测试（integration unit testing）——集成测试确定软件组件、服务和子系统之间的交互是否如预期的那样工作。在事务处理和数据管理的上下文中，这可能意味着你想要测试应用程序是否正确地利用数据库进行工作（例如，刚对一件拍卖货品所做的出价是否正确地被保存在数据库中）。
- 功能单元测试（functional unit testing）——功能测试检验整个用例，以及完成这个特定的用例所需的所有应用程序组件中的公共接口。功能测试可以包括应用程序的工作流和用户界面（例如，通过模拟用户在对一件拍卖货品给出新价格之前必须如何登录）。

接下来的几节关注集成单元测试，因为当持久化数据和事务处理成为你的主要关注点时，它是最相关的一种测试。这并不意味着其他种类的测试没有它重要，我们会随时提供提示。如果你想要全面地了解单元测试，我们推荐*JUnit in Action*（Massol，2003）。

不用JUnit，而是用TestNG。这不应该对你产生什么困扰，因为这里介绍的基本原理可适用于任何测试框架。TestNG比JUnit更容易进行集成和功能单元测试，我们尤其喜欢JDK 5.0特性和基于注解的测试集合配置。

先来编写一个简单、单独的逻辑单元测试，以便你可以了解TestNG是如何工作的。

16.5.2 TestNG 简介

TestNG是一个测试框架，它有一些独特的功能，这使得它对于涉及复杂的测试（如集成或者功能测试等）的单元测试特别有用。其中有些TestNG特性是用于测试集合声明的JDK 5.0注解，支持配置参数，以及把测试灵活地分组到测试单元中去的分组，支持用于IDE和Ant的各种插件，以及通过遵循依赖以特定顺序执行测试的能力。

我们想要一步一步地接近这些特性，因此你首先要编写一个简单的逻辑单元测试，没有任何子系统的集成。

1. TestNG中的单元测试

逻辑单元测试验证单项功能，并检查特定的组件或者方法是否遵循所有的业务规则。如果理解本章前面关于巧妙的领域模型的讨论（详见16.1.4节），就会知道我们更喜欢将可以进行单元测试的业务逻辑封装在领域模型实现中。逻辑单元测试在业务层和领域模型中执行方法的测试：

```
public class AuctionLogic {

    @org.testng.annotations.Test(groups = "logic")
    public void highestBidWins() {

        // A user is needed
        User user = new User(...);

        // Create an Item instance
        Item auction = new Item(...);

        // Place a bid
        BigDecimal bidAmount = new BigDecimal("100.00");
        auction.placeBid(user, bidAmount,
                         new BigDecimal(0), new BigDecimal(0) );

        // Place another higher bid
        BigDecimal higherBidAmount = new BigDecimal("101.00");
        auction.placeBid(user, higherBidAmount,
                         bidAmount, bidAmount );

        // Assert state
        assert auction.getBids().size() == 2;
    }

}
```

AuctionLogic类是一个包含所谓测试方法的任意类。**测试方法**（**test method**）是用@Test注解作了标识的任何方法。可以选择把分组的名称分配给测试方法，以便以后可以通过合并分组来动态地装配测试单元。

测试方法highestBidWins()给“出价”用例执行部分逻辑。首先，出价需要User的实例——是否为同一个用户并非这项测试的关注点。

这项测试可能以几种方式失败，表明已经违背了业务规则。第一个出价启动了拍卖（当前最高和最低出价均为0），因此这里不会有任何失败。出第二个价格是必须成功且不抛出BusinessException的步骤，因为新的出价金额比前一个更高。最后，用Java assert关键字和一个比较操作决定拍卖状态。

你会经常想要测试失败的业务逻辑，并期望等到异常。

2. 期待测试中的失败

拍卖系统有一个相当严重的bug。如果看一下16.1.4节中Item.placeaBid()的实现，就可以看到你检查所给定的新出价金额是否比任何现有的出价金额更高。然而，你从来不把它与拍卖的初始底价进行对比。这意味着用户可以出任何价格，即使比初始价格更低。

通过测试失败来检验这一点。下列过程会产生异常：

```
public class AuctionLogic {

    @Test(groups = "logic")
    public void highestBidWins() { ... }

    @Test(groups = "logic")
    @ExpectedExceptions(BusinessException.class)
    public void initialPriceConsidered() {

        // A user is needed
        User user = new User(...);

        // Create an Item instance
        Item auction = new Item(..., new BigDecimal("200.00") );
        // Place a bid
        BigDecimal bidAmount = new BigDecimal("100.00");
        auction.placeBid(user, bidAmount,
                         new BigDecimal(0), new BigDecimal(0) );
    }

}
```

现在，值为100的出价不得不失败，因为拍卖的初始底价是200。TestNG要求这个方法抛出BusinessException——否则测试就失败。更细粒度的业务异常类型让你对业务逻辑的核心部分更精确地测试失败。

最后，考虑领域模型的执行路径的数量多少，决定了全部的业务逻辑测试覆盖度。可以使用各种工具[如cenqua clover（http://www.cenqua.com/clover/），它可以算出测试套件的代码覆盖率]，并提供关于系统质量的许多其他值得关注的细节。

让我们用TestNG和Ant来执行之前的这些测试方法。

3. 创建和运行测试套件

通过TestNG，可以用多种方法创建测试套件，并启动测试。可以通过IDE中的按钮单击来直接调用测试方法（在安装了TestNG插件之后），或者可以把常规构建中的单元测试与测试套件的Ant任务和XML描述集成在一起。

前几节中单元测试的XML测试单元描述，看起来如下：

```
<!DOCTYPE suite SYSTEM "http://testng.org/testng-1.0.dtd" >

<suite name="CaveatEmptor" verbose="2">

    <test name="BusinessLogic">
        <run><include name="logic.*"/></run>

        <packages>
            <package name="auction.test"/>
        </packages>

        <!-- Or just the class...
        <classes>
            <class name="auction.test.AuctionLogic"/>
        </classes>
            -->
    </test>

</suite>
```

测试套件是几个逻辑测试的集合——不要把这个与测试方法混为一谈。逻辑测试由TestNG在运行时确定。例如，名为BusinessLogic的逻辑测试将所有的测试方法（也就是说，用@Test作了标识的方法）都包括在auction.test包的类中。这些测试方法必须属于一个名称以logic开头的分组，注意.*是个正则表达式，意思是“任意数量的任意字符”。另一种方法是，可以列出你想要显式地考虑这个逻辑测试的一部分的测试类，而不是整个包（或者几个包）。

可以编写一些测试类和方法，以任何方便的方式安排它们，然后通过混合和匹配类、包和具名的组合来创建任意的测试集合。来自任意类和包的这个逻辑测试集合，以及利用通配符匹配分离到分组中，使得TestNG比许多其他的测试框架更为强大。

在项目的根目录下，将单元描述XML文件保存为test-logic.xml。现在，通过Ant和build.xml中的下列目标来运行这个测试套件：

```
<taskdef resource="testngtasks" classpathref="project.classpath"/>

<target name="unittest.logic" depends="compile, copymetafiles">
        description="Run logic unit tests with TestNG">

    <delete dir="${basedir}/test-output"/>
    <mkdir  dir="${basedir}/test-output"/>

    <testng outputDir="${basedir}/test-output"
            classpathref="project.classpath">
        <xmlfileset dir="${basedir}">
            <include name="test-logic.xml"/>
        </xmlfileset>
    </testng>

</target>
```

首先，TestNG Ant任务被引入到构建中。然后，unittest.logic目标通过项目根目录下的单元描述文件textlogic.xml启动一个TestNG运行。TestNG在outputDir中创建了一个HTML报告，因此你每次在运行测试之前都要清理这个目录。

调用这个Ant目标，并体验你的第一个TestNG集合。接下来讨论集成测试，以及TestNG如何通过运行时环境的灵活配置来支持你。

16.5.3　测试持久层

测试持久层意味着必须测试和检验几个组件，看看它们是否正确地进行交互。这意味着：

- 测试映射（testing mapping）——你想要测试映射的语法正确性（被映射的所有列和表是否都与属性和类相匹配）。
- 测试对象状态转变（testing object state transition）——你想要测试一个对象转变是否正确地从瞬时到持久化再到脱管状态。换句话说，你想要确保数据被正确地保存在数据库中，确保它可以被正确加载，确保对于传播性状态变化的所有潜在的级联规则都如预期般有效。
- 测试查询（testing query）——任何重要的HQL、Criteria和（可能的）SQL查询都应该测试被返回数据的正确性。

所有这些测试都要求持久层不要单独测试，而是与一个在运行的数据库管理系统集成在一

起。此外，所有其他的基础结构，诸如Hibernate SessionFactory或者JPA EntityManagerFactory，都必须可用。你需要一个运行时环境，启用你想要包含在集成测试中的任何服务。

考虑你想要在其上面运行这些测试的数据库管理系统。理想情况下，这应该与你要在产品中给应用程序部署的DBMS产品相同。另一方面，在有些情况下，开发时你却可能在一个不同的系统中运行集成测试——例如，轻量级的HSQL DB。注意，由于Hibernate的数据库可移植性，对象状态转变可以透明地进行测试。任何复杂的应用程序都有经常为特定的数据库管理系统进行量身定做的映射和查询（通过公式和原生的SQL语句），因此与非产品的数据库产品的任何集成测试都没有意义。许多DBMS供应商提供免费许可，甚至提供他们主要数据库产品的轻量级版本用于开发。在开发期间，在转换到不同的数据库管理系统之前先考虑这些。

必须先准备测试环境，并在你编写任何集成单元测试之前启用运行时基础设施。

1. 编写DBUnit超类

持久层的集成测试环境要求安装并激活数据库管理系统——我们希望你的案例中考虑到这一点。接下来，要考虑集成测试集合，以及如何以正确的顺序执行配置和测试。

首先，为了使用数据访问对象，你必须启动Hibernate——构建SessionFactory是最容易的部分。更难的是定义运行测试前后所需要的配置操作的顺序。一般的顺序是这样的：

(1) 重置数据库内容为一种大家熟知的状态。这么做最容易的方法是利用Hibernate工具箱，通过数据库Schema自动导出。然后用空的、干净的数据库开始测试。

(2) 通过将数据导入到数据库中，给测试创建任何基础数据。这可以通过各种方式来完成，比如用Java代码通过编程完成，或者利用工具如DBUnit（http://www.dbunit.org）。

(3) 创建对象，并执行你想要测试的任何状态转变，比如通过调用几个DAO，在TestNG测试方法中保存或者加载对象。

(4) 状态转变之后，通过检查Java代码中的对象，以及（或）通过执行SQL语句并验证数据库的状态，来声明状态。

考虑几个这样的集成测试。你应该始终从第1步开始，并在每一个测试方法之后都导出新的数据库Schema，然后再次导入所有基础数据吗？如果执行大量的测试，这可能很费时间。另一方面，这种方法比在每一个测试方法之后的删除和清理更为容易，这应该成为一个额外的步骤。

有一个可以帮助你给每个测试进行这些配置和准备步骤的工具是DBUnit。可以轻松地导入和管理数据集——例如，必须为每一次测试运行重置为大家熟知的状态的数据集。

即使TestNG允许你以任何可以想象的方式组合和装配测试单元，但封装所有配置和DBUnit设置操作的超类还是很方便。看一下代码清单16-2中适合于Hibernate数据访问对象的集成测试的超类。

代码清单16-2 Hibernate集成测试的超类

```
public abstract class HibernateIntegrationTest {

    protected SessionFactory sessionFactory;          ❶

    protected String dataSetLocation;                 ❷
```

```
    protected List<DatabaseOperation> beforeTestOperations
            = new ArrayList<DatabaseOperation>();
    protected List<DatabaseOperation> afterTestOperations
            = new ArrayList<DatabaseOperation>();

private ReplacementDataSet dataSet;                          ❸

@BeforeTest(groups = "integration-hibernate")                ❹
void startHibernate() throws Exception {
    sessionFactory = HibernateUtil.getSessionFactory();
}

@BeforeClass(groups = "integration-hibernate")               ❺
void prepareDataSet() throws Exception {

    // Check if subclass has prepared everything
    prepareSettings();
    if (dataSetLocation == null)
        throw new RuntimeException(
            "Test subclass needs to prepare a dataset location"
        );

    // Load the base dataset file
    InputStream input =
            Thread.currentThread().getContextClassLoader()
                    .getResourceAsStream(dataSetLocation);

    dataSet = new ReplacementDataSet(
                    new FlatXmlDataSet(input)
                );
    dataSet.addReplacementObject("[NULL]", null);
}

@BeforeMethod(groups = "integration-hibernate")              ❻
void beforeTestMethod() throws Exception {
    for (DatabaseOperation op : beforeTestOperations ) {
        op.execute(getConnection(), dataSet);
    }
}

  @AfterMethod(groups = "integration-hibernate")             ❼
void afterTestMethod() throws Exception {
    for (DatabaseOperation op : afterTestOperations ) {
        op.execute(getConnection(), dataSet);
    }
}

// Subclasses can/have to override the following methods

protected IDatabaseConnection getConnection() throws Exception {   ❽

    // Get a JDBC connection from Hibernate
    Connection con =
        ((SessionFactoryImpl)sessionFactory).getSettings()
                .getConnectionProvider().getConnection();

    // Disable foreign key constraint checking
    con.prepareStatement("set referential_integrity FALSE")
```

```
            .execute();
            return new DatabaseConnection( con);
        }

        protected abstract void prepareSettings();  ❾
    }
```

❶ 特定单元中的所有测试都使用相同的Hibernate SessionFactory。

❷ 子类可以定制在每种测试方法之前和之后执行的DBUnit数据库操作。

❸ 子类可以定制哪个DBUnit数据集应该被用于它所有的测试方法。

❹ Hibernate在测试集合的逻辑测试运行之前启动——注意，@BeforeTest仍然并非意味着是在每个测试方法之前。

❺ 对于每个测试（子）类，DBUnit数据集都必须从XML文件中加载，并且所有的空标识都必须用真正的NULL替换。

❻ 在每一种测试方法之前，用DBUnit执行必要的数据库操作。

❼ 在每一个测试方法之后，用DBUnit执行所需的数据库操作。

❽ 默认情况下，从Hibernate的ConnectionProvider中获得一个简单的JDBC连接，并把它包装在DBUnit DatabaseConnection中。你还禁用对这个连接的外键约束检查。

❾ 子类必须覆盖这个方法，并准备数据集文件位置，以及假定要在每种测试方案之前和之后运行的操作。

这个超类同时兼顾到了许多事情，并且将集成测试写成子类很容易。每个子类都可以定制它要使用哪一个DBUnit数据集（我们很快会讨论这些数据集），以及该数据集上的哪些操作（例如INSERT和DELETE）必须在特定的测试方法执行之前和之后运行。

注意，这个超类假设数据库是活动的，并且已经创建了一个有效的Schema。如果想要重新创建并自动给每个测试单元导出数据库Schema，就通过设置hibernate.hbm2ddl.auto配置选项为create来启用它。然后当创建SessionFactory时，Hibernate就删除旧的并导出新的数据库Schema。

接下来看一下DBUnit数据集。

2. 准备数据集

根据所建议的测试策略，每个测试（子）类都使用一个特定的数据集。这只不过是我们要简化超类的一个决定。你可以给每种测试方法使用一个数据集，或者为整个逻辑测试使用单个数据集，如果喜欢的话。

数据集是DBUnit可以替你维护的一个数据集合。在DBUnit中创建和使用数据集有许多种方法。我们想要介绍最容易的其中一种场景，通常这就足够了。首先，将数据集写入XML文件，用DBUnit要求的语法：

```
<?xml version="1.0"?>

<dataset>
    <USERS      USER_ID              ="1"
                OBJ_VERSION          ="0"
```

```
            FIRSTNAME                  ="John"
            LASTNAME                   ="Doe"
            USERNAME                   ="johndoe"
            PASSWORD                   ="secret"
            EMAIL                      ="jd@mail.tld"
            RANK                       ="0"
            IS_ADMIN                   ="false"
            CREATED                    ="2006-09-23 13:45:00"
            HOME_STREET                ="[NULL]"
            HOME_ZIPCODE               ="[NULL]"
            HOME_CITY                  ="[NULL]"
            DEFAULT_BILLING_DETAILS_ID ="[NULL]"
        />

    <ITEM />
</dataset>
```

对于这个文件你不需要DTD，虽然指定DTD可以让你验证数据集的语法正确性（它也意味着必须把部分数据库Schema转变为DTD）。每个数据行都有它自己的元素，包含表名称。例如，一个<USERS>元素声明USERS表中的一个行数据。注意，你用[NULL]作为被集成测试超类用真正的SQL NULL替换掉的标记。还要注意，可以给想让DBUnit维护的每一张表都增加一个空行。在此处介绍的这个数据集中，ITEM表是数据集的一部分，并且DBUnit可以删除该表中的任何数据（这一点稍后会派得上用场）。

假设这个数据集保存在auction.test.dbunit包的XML文件basedata.xml中。接下来，你要编写一个利用这个数据集的测试类。

3. 编写测试类

测试类（test class）把依赖于特定数据集的测试方法进行分组。看一下以下示例：

```
public class PersistentStateTransitions
                extends HibernateIntegrationTest {

    protected void prepareSettings() {
        dataSetLocation = "auction/test/dbunit/basedata.xml";
        beforeTestOperations.add(DatabaseOperation.CLEAN_INSERT);
    }

    ...

}
```

这是HibernateIntegrationTest的一个子类，它准备所需的数据集位置。它还要求CLEAN_INSERT操作要在任何测试方法之前运行。这个DBUnit数据库操作删除所有的行（实际上清理了USERS和ITEM表），然后插入数据集中定义的行。因此每个测试方法都有了一个干净的数据库状态。

DBUnit包括许多内建的DatabaseOperation，如INSERT、DELETE、DELETE_ALL甚至REFRESH。查看DBUnit参考文档中完整的清单，我们不在此重复。注意你可以堆叠这些操作：

```
public class PersistentStateTransitions
                extends HibernateIntegrationTest {

    protected void prepareSettings() {
```

```
        dataSetLocation = "auction/test/dbunit/basedata.xml";
        beforeTestOperations.add(DatabaseOperation.DELETE_ALL);
        beforeTestOperations.add(DatabaseOperation.INSERT);
        afterTestOperations.add(DatabaseOperation.DELETE_ALL);
    }

    ...
}
```

在每种测试方法之前，数据集表中的所有内容都被删除，然后插入。在每个测试方法之后，数据集表中的所有内部都被再次删除。这个堆叠保证每个测试方法之前和之后都有干净的数据库状态。

现在可以在这个测试类中编写实际的测试方法了。类的名称PersistentStateTransaction，暗示着你想要做的事：

```
@Test(groups = "integration-hibernate")
public void storeAndLoadItem() {

    // Start a unit of work
    sessionFactory.getCurrentSession().beginTransaction();

    // Prepare the DAOs
    ItemDAOHibernate itemDAO = new ItemDAOHibernate();
    itemDAO.setSession( sessionFactory.getCurrentSession() );

    UserDAOHibernate userDAO = new UserDAOHibernate();
    userDAO.setSession( sessionFactory.getCurrentSession() );

    // Prepare a user object
    User user = userDAO.findById(11, false);

    // Make a new auction item persistent
    Calendar startDate = GregorianCalendar.getInstance();
    Calendar endDate = GregorianCalendar.getInstance();
    endDate.add(Calendar.DAY_OF_YEAR, 3);

    Item newItem =
        new Item( "Testitem", "Test Description", user,
                  new BigDecimal(123), new BigDecimal(333),
                  startDate.getTime(), endDate.getTime() );

    itemDAO.makePersistent(newItem);

    // End the unit of work
    sessionFactory.getCurrentSession()
                  .getTransaction().commit();

    // Direct SQL query for database state in auto-commit mode
    StatelessSession s = sessionFactory.openStatelessSession();
    Object[] result = (Object[])
            s.createSQLQuery("select INITIAL_PRICE ip," +
                             "SELLER_ID sid from ITEM")
              .addScalar("ip",  Hibernate.BIG_DECIMAL)
              .addScalar("sid", Hibernate.LONG)
              .uniqueResult();
    s.close();
```

```
        // Assert correctness of state
        assert result[0].getClass() == BigDecimal.class;
        assert result[0].equals( newItem.getInitialPrice().getValue() );
        assert result[1].equals( 11 );

    }
```

这个测试方法使Item实例变成了持久化。虽然这看起来好像有很多代码，但是只有几个部分值得关注。

这个状态转变需要User实例，因此你在数据集中定义的用户数据通过Hibernate加载。必须提供与你写入数据集作为主键的相同标识符值（在这个例子中为11）。

当工作单元提交时，所有的状态转变都完成，并且Session的状态与数据库同步。最后一步是真正的测试，断言数据库内容处于预期的状态。

可以用多种方法测试数据库状态。很显然，你不会出于这个目的而使用Hibernate查询或者Session操作，因为Hibernate是你的测试和实际的数据库内容之间一个额外的层。为了确保你真的命中数据库，并看到该状态，建议用SQL查询。

Hibernate使得执行SQL查询和检查返回的值变得很容易。在这个例子中，打开Hibernate StatelessSession来创建这个SQL查询。用在这个查询中的数据库连接处于自动提交模式（hibernate.connection.autocommit设置为true），因为你没有启动事务。这是Stateless-Session的理想用例，因为它禁用了任何高速缓存、任何级联、任何拦截器，或者可能在数据库中干扰你看法的任何东西。

把所有这些都放到TestNG测试套件和Ant目标中。

4．运行集成测试

这是XML测试套件描述符：

```
<!DOCTYPE suite SYSTEM "http://testng.org/testng-1.0.dtd" >

<suite name="CaveatEmptor" verbose="2">

    <test name="PersistenceLayer">
        <groups>
            <run><include name="integration-hibernate.*"/></run>
        </groups>

        <packages>
            <package name="auction.test.dbunit"/>
         </packages>
    </test>

</suite>
```

逻辑测试PersistenceLayer包括在auction.test.dbunit包中设立的所有测试类和测试方法，如果它们的分组名称以integration-hibernate开头的话。对于任何TestNG配置方法（用@BeforeClass等作了标识的那些方法）也一样，因此你需要把任何包含配置方法的类（超类也是）放在同一个包中，并把它们添加到相同的分组。

为了用Ant运行这个测试套件，要替换掉你在16.5.2节中写在Ant目标中的XML单元描述符。

在前面的示例中，我们只是接触到了TestNG和DBUnit的一点皮毛。它们还有更多有用的选

项，例如，可以在TestNG中用套件描述符中的任意设置将测试方法参数化。可以创建一个启动EJB 3.0容器服务器的测试集合（请见2.2.3节和下载的CaveatEmptor中的EJB3IntegrationTest超类），然后测试EJB层。当你利用介绍过的策略，从基类开始构建测试环境的时候，我们推荐相应的TestNG和DBUnit文档。

你可能想知道如何测试映射和查询，因为我们只讨论了对象—状态转变的测试。首先，可以通过设置hibernate.hbm2ddl.auto为validate来轻松地测试映射。然后当构建SessionFactory时，Hibernate通过把它们与数据库目录元数据进行对比，来验证映射。第二，测试查询与测试对象状态转变一样：编写集成测试方法，并断言所返回的数据的状态。

最后，考虑负载和压力测试，以及如果你想要测试系统的性能，还必须关注的一些方面。

16.5.4 考虑性能基准

企业应用程序开发中最难的事情之一是保证应用程序的性能和可伸缩性。先来定义这些术语。

性能（performance）通常被当作是基于应用程序的请求/响应的反应时间。如果单击一个按钮，希望半秒钟之内得到响应。或者，根据用例，你希望特定的事件（或者批量操作）可以在合理的时限之内得到执行。合理的响应时间一般取决于具体情况和某些应用程序功能的使用模式。

可伸缩性（scalability）是系统在更高的负载下合理执行的能力。想象一下，假设不是1个人单击1个按钮，而是5000人单击大量的按钮。系统的可伸缩性越好，就可以在它上面捆绑越多的并发用户，而不会降低性能。

关于性能已经讨论过很多了。在我们看来，创建一个执行得很好的系统，与创建一个没有明显性能瓶颈的Hibernate / 数据库应用程序同义。性能瓶颈可以是你认为的编程错误或者设计不当的任何东西——例如，错误的抓取策略、错误的查询，或者是Session和持久化上下文的错误处理。测试系统性能的合理与否通常是验收测试的一部分。实际上，性能测试经常由实验环境中一组专门的终端用户测试员来完成，或者通过现实世界条件下的少数用户群。纯粹的自动化性能测试很少。

也可以通过自动化的可伸缩性测试发现性能瓶颈，这是终极目标。然而，我们已经在自己的职业生涯中见过许多压力和负载测试，并且它们中大多数都没有考虑下列规则中的一个或者几个：

- 用现实世界的数据集测试可伸缩性。不要用可以完全适合数据库服务器中硬盘高速缓存的数据集进行测试。使用已经存在的数据，或者使用测试数据生成器来生成测试数据（例如，TurboData:http://www.turbodata.ca/）。确保测试数据尽可能地接近系统将在投产中运行的数据，包含相同的数量、分类和选择性。
- 用并发测试可伸缩性。测量单个活动用户进行单个查询所花费时间的自动化性能测试，并非在告诉你关于投产中系统可伸缩性的任何信息。持久化服务如Hibernate在设计时考虑了高并发，因此没有并发的测试更有可能显示出乎意料的过载！一旦启用更多的并发工作单元和事务，立即就会看见如二级高速缓存这样的特性如何帮助你保持性能。
- 用真实的用例测试可伸缩性。如果你的应用程序必须处理复杂的事务（例如，根据复杂的统计模型计算股票的市值），就应该通过执行这些用例来测试系统的可伸缩性。分析你

的用例，并选择普遍的场景——许多应用程序都只有几个最关键的用例。避免编写随机保存和加载几千个对象的微基准，从这种测试中得出的数字没有任何意义。

为可伸缩性测试的自动执行创建测试环境是其中一项工作。如果遵循我们所有的规则，就要先花点时间分析数据、用例，以及你期望的系统负载。一旦有了这些信息，就可以建立自动化测试了。

典型的客户端/服务器端应用程序的可伸缩性测试，一般需要并发运行客户端的模拟，以及每个被执行操作的统计集合。你应该考虑现有的测试解决方案，商业的（如LoadRunner，http://www.mercury.com/）或者开源的［如Grinder（http://grinder.sourceforge.net/）或者JMeter（http://jakarta.apache.org/jmeter/）］。创建测试通常涉及为模拟客户端编写控制脚本，以及配置在服务器进程（例如，为了特定的事务或者统计集合的直接执行）上运行的代理。

最后，测试系统的性能和（特别是）可伸缩性一般是软件应用程序生命周期中一个独立的阶段。你不应该在开发前期测试系统的可伸缩性。也不应该启用Hibernate的二级高速缓存，直到你具备根据我们提到过的规则而创建的测试环境。

在项目的后期，可以把自动化的可伸缩性测试增加到每日的集成测试中。应该在投产之前测试系统的可伸缩性，作为常规测试周期的一部分。另一方面，我们不主张将任何性能和可伸缩性测试延迟到最后一刻。不要试图在投产之前，通过调整Hibernate二级高速缓存，一天就修复好性能瓶颈。你可能不会成功。

把性能和负载测试当作开发过程中的必需部分，包含定义明确的阶段、准则和需求。

16.6　小结

本章探讨了分层应用程序和一些重要的模式和最佳实践。我们讨论了如何用Hibernate设计Web应用程序，并实现OSIV模式。你现在知道如何创建巧妙的领域模型，以及如何从控制器代码中分离业务逻辑，灵活的命令模式是软件设计工具库中最重要的工具。我们讨论了EJB 3.0组件，以及如何通过添加一些注解，来进一步简化POJO应用程序。

最后广泛地讨论了持久层，你用检验持久层的TestNG编写了数据访问对象和集成测试。

第 17 章

JBoss Seam简介

本章内容

- 利用JSF和EJB 3.0开发Web应用程序
- 利用Seam改善Web应用程序
- 利用Hibernate Validator整合Seam
- 利用Seam管理持久化上下文

在这最后一章中，我们介绍JBoss Seam框架。Seam是一个创新的新框架，用于通过Java EE 5.0平台进行的Web应用程序开发。Seam带来了两个新的标准，即JavaServer Faces（JSF）和EJB 3.0，通过统一它们的组件和编程模型，使二者关系更加密切。最吸引那些依赖Hibernate（或者EJB 3.0中的任何Java Persistence提供程序）的开发人员的是，Seam的自动化持久化上下文管理，以及它在应用程序流中给对话的定义所提供的一级构造。如果你已经在Hibernate应用程序中见过`LazyInitializationException`，Seam也有合适的解决方案。

关于Seam有更多的内容要讲，我们鼓励你阅读本章，即使你已经决定使用一种不同的框架，或者正在编写一个Web应用程序。虽然Seam目前的目标是Web应用程序，并且也依赖JSF作为表现框架，但是其他的选项在未来应该可用（例如，你可能已经使用Ajax调用来访问Seam组件了）。此外，Seam的许多重要概念目前正被标准化，并通过Web Beans JSR 299（http://www.jcp.org/en/jsr/detail?id=299）带回到Java EE 5.0平台。

阐述Seam与学习Seam一样，都有许多种方法。本章先介绍一下Seam承诺要解决哪些问题；然后讨论各种解决方案，并强调最吸引Hibernate用户的（比如你）是哪些特性。

17.1 Java EE 5.0 编程模型

Java EE 5.0明显比它之前的版本都更容易使用，也更加强大。与Web应用程序开发人员最相关的Java EE 5.0平台的两种规范是JSF和EJB 3.0。

JSF和EJB 3.0究竟好在哪里？我们先强调每种规范中的主要概念和特性，然后用JSF和EJB 3.0编写一个小示例，并将它与用Java（如Struts和EJB 2.x）编写Web应用程序的旧方法进行比较。最后关注依然存在的问题，以及Seam如何使JSF和EJB 3.0变成一个更强大且更方便的组合。

注意，本章不可能涵盖JSF和EJB 3.0的全部内容。建议把本章与Sun Java EE 5.0教程（http://java.sun.com/javaee/5/docs/tutorial/doc/）一起读，并且如果你想了解更多关于某个特定主题的内容，就浏览这个教程。另一方面，如果你已经接触过JSF或者EJB 3.0（甚至Hibernate），很可能发现学习Seam不过是小菜一碟。

17.1.1　JSF 详解

JSF简化了在Java中对Web用户界面的构建。作为表现框架，JSF提供了以下高级特性：

- JSF给可视的组件定义了一个可扩展的组件模型，经常称作小部件（widget）。
- JSF给backing bean或者被托管的bean（它们包含应用程序逻辑）定义了组件编程模型。
- JSF定义了用户界面和应用程序逻辑之间的交互，并允许你以灵活的方式把两者绑定在一起。
- JSF允许你用XML声明性地定义导航规则——也就是说，哪一页显示应用程序逻辑中的特定结果。

让我们多花一点时间讨论这每一种特性，以及是什么使它们变得大有用处。

JSF定义了一组内建的可视组件，这是每个JSF实现都必须支持的（例如按钮和输入文本字段）。这些可视组件在页面上呈现为HTML（和Javascript）。在编写本书之时，有几个高质量的开源和商业JSF小部件库可用。对于开发人员，现成的可视组件是很棒的。不必手工对它们编码，并且最重要的是，不必维护它们或者使它们在不同的浏览器上都有效（如果需要更多使用Javascript的复杂可视组件，这尤其痛苦）。

页面可以通过任何理解JSF小部件的HTML模板引擎来创建。虽然JSP似乎是一种显而易见的选择，但就经验来看，它并不是最好的。我们发现JavaServer Facelets（http://facelets.dev.java.net/）完全适合于构建JSF视图和创建包含JSF小部件的HTML模板。（使用Facelets的另一个好处在于免费获得新的统一表达语言，甚至不用有JSP 2.1功能的servlet容器。）本章我们将在所有的JSF示例中使用Facelets。

JSF托管的应用程序组件（称作backing bean）使你的Web应用程序界面生效，它们包含应用程序代码。这些是一般的POJO，它们在JSF XML配置文件中定义并连接在一起。这种连接支持基本的依赖注入，以及backing bean实例的生命周期管理。backing bean的可用范围（它所处的位置）是当前的HTTP请求上下文、当前HTTP会话上下文和全局应用程序上下文。通过创建bean，并让JSF在其中一个上下文中管理它们的生命周期，来编写应用程序逻辑。

可以用一种表达式语言，将模型值从backing bean绑定到可视组件。例如，创建一个包含文本输入字段的页面，并把它绑定到具名的backing bean字段或者获取方法/设置方法对。然后这个backing bean名称就在JSF配置中被映射到实际的backing bean类，并声明该类的一个实例应该如何被JSF处理（在请求中，在HTTP会话中，或者在应用程序上下文中）。JSF引擎自动保持backing bean字段（或者属性）与小部件的状态同步，就像用户看到（或者操作）的一样。

JSF是事件驱动的表现框架。如果单击一个按钮，就会触发JSF `ActionEvent`，并传递到已注册的监听器。动作事件的监听器还是你在JSF配置中命名的backing bean。然后backing bean可以

对该事件做出响应——例如，通过将backing bean字段的当前值（它被绑定到一个文本输入小部件）保存到数据库中。这是对JSF用处的一种简化的阐述。在内部，来自Web浏览器的每个请求都跨过几个处理阶段。

当在JSF页面上单击一个按钮时，服务器上一个典型的请求处理顺序如下（这个过程如图17-7所示）：

(1) 所有小部件的恢复视图（Restore View）（JSF可以将小部件状态保存在服务器或者客户端上）。

(2) 应用请求参数（Apply Request Parameters），更新小部件的状态。

(3) 处理验证（Process Validations），这对于验证用户输入是必需的。

(4) 更新模型值（Update Model Values），通过调用backing bean的绑定字段和设置方法，退回小部件。

(5) 调用应用程序（Invoke Application），并将动作事件传递到监听器。

(6) 呈现响应（Render Response），客户会看到该页面。

很显然，请求可以采用不同的子程序；例如，如果验证失败，呈现响应可能发生在处理验证之后。

在已经提过的Sun Java EE 5教程的第9章中可以找到JSF生命周期和处理阶段的一个很好的图例。本章稍后也将回到JSF处理模型的话题。

哪个响应得到呈现，以及哪个页面显示给用户，这取决于所定义的导航规则，以及动作事件的结果是什么。JSF中的结果是简单的字符串，如“success”或者“failure”。这些字符串通过backing bean产生，然后在JSF XML配置文件中被映射到页面。这也被称作自由导航流（free navigation flow），例如，可以在浏览器中单击后退按钮，或者通过输入页面的URL直接跳到该页面。

如果正在寻找一种Web框架，JSF与Facelets的组合就是一种很好的解决方案。另一方面，Web应用程序的backing bean——实现应用程序逻辑的组件——通常需要访问事务的资源（大多数时候为数据库）。这就是EJB 3.0应运而生的地方。

17.1.2 EJB 3.0 详解

EJB 3.0是一个给事务的组件定义编程模型的Java EE 5.0标准。对于Web应用程序开发人员来说，EJB 3.0的下列特性最值得关注：

- EJB 3.0定义了在简单的Java 类中主要基于注解的组件编程模型。
- EJB 3.0定义了无状态、有状态和消息驱动的组件，以及运行时环境如何管理组件实例的生命周期。
- EJB 3.0定义了组件如何被连接在一起，如何获得对组件的引用，以及组件如何互相调用。
- EJB 3.0定义横切关注点如何被处理，例如事务和安全性。也可以编写定制拦截器，并用它们把组件包装起来。
- EJB 3.0标准化了Java Persistence，以及如何通过自动的和透明的ORM来访问SQL数据库。如果想要访问SQL数据库，就创建领域模型实体类（例如Item、User、Category），并通过

注解将它们从Java Persistence规范映射到数据库Schema。EJB 3.0持久化管理器API（EntityManager），现在成了数据库操作的通道。

在EJB 3.0组件中执行数据库操作——例如，有状态或者无状态会话bean。这些bean都是简单的Java类，你用一些注解将它们启用为EJB。然后在组件方法中获得容器的服务，例如自动的依赖注入（当你需要的时候就得到EntityManager）和声明性事务划分。例如，如果用户必须在应用程序的一个对话中浏览几个页面，有状态会话bean就会帮助你为特定的客户端保持状态。

可以用EJB 3.0组件和实体作为JSF动作和小部件的backing bean吗？可以将JSF文本字段小部件绑定到Item实体类中的字段吗？JSF按钮单击可以被直接发送到会话bean方法吗？

让我们用一个例子来试试。

17.1.3　用JSF和EJB 3.0编写Web应用程序

你要创建的Web应用程序很简单：它有一个搜索屏，用户可以在那里给特定的货品输入标识符，还有当在数据库中找到货品时所显示的详细信息屏幕。在这个详细信息屏幕中，用户可以编辑货品的数据，并将所做的改变保存到数据库。

（在阅读这些示例的时候不一定非要编写这个应用程序，稍后将通过引入Seam，进行重大的改进。那时候就可以开始编码了。）

从实体Item的数据模型开始。

1．创建实体类和映射

Item实体类来自CaveatEmptor。它也已经被注解和映射到了SQL数据库（见代码清单17-1）。

代码清单17-1　一个被注解和映射的实体类

```
package auction.model;
import ...;

@Entity
@Table(name = "ITEM")
public class Item implements Serializable {

    @Id @GeneratedValue
    @Column(name = "ITEM_ID")
    private Long id = null;

    @Column(name = "ITEM_NAME", length = 255,
            nullable = false, updatable = false)
    private String name;

    @ManyToOne(fetch = FetchType.LAZY)
    @JoinColumn(name="SELLER_ID",
                nullable = false, updatable = false)
    private User seller;

    @Column(name = "DESCRIPTION", length = 4000, nullable = false)
    private String description;

    @Column( name="INITIAL_PRICE", nullable = false)
```

```
    private BigDecimal initialPrice;

    Item() {}

    // Getter and setter methods...
}
```

这是CaveatEmptor Item实体的一个简化版，没有任何集合。接下来是允许用户搜索货品对象的搜索页面。

2. 用Facelets和JSF编写搜索页面

应用程序的搜索页面是用作为模板引擎的Facelets写成的一个页面，它是有效的XML。JSF小部件被嵌入在该页面中，通过它的输入字段和按钮来创建搜索表单（见代码清单17-2）。

代码清单17-2 利用Facelets编写XHTML格式的search.xhtml页面

```
<!DOCTYPE html PUBLIC                                          ❶
        "-//W3C//DTD XHTML 1.0 Transitional//EN"
        "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<html xmlns="http://www.w3.org/1999/xhtml"                     ❷
      xmlns:ui="http://java.sun.com/jsf/facelets"
      xmlns:h="http://java.sun.com/jsf/html"
      xmlns:f="http://java.sun.com/jsf/core">

<head>
    <title>CaveatEmptor - Search items</title>
    <link href="screen.css" rel="stylesheet" type="text/css"/>  ❸
</head>

<body>

<ui:include src="header.xhtml"/>                               ❹

<h:form>                                                       ❺
    <span class="errors"><h:message for="itemSearchField"/></span>   ❻

    <div class="entry">                                        ❼
        <div class="label">Enter item identifier:</div>

        <div class="input">                                    ❽
            <h:inputText id="itemSearchField"
                         size="3" required="true"
                         value="#{itemEditor.itemId}">

                <f:validateLongRange minimum="0"/>             ❾

            </h:inputText>
        </div>
    </div>

    <div class="entry">
        <div class="label"> </div>

        <div class="input">
            <h:commandButton value="Search" styleClass="button"        ❿
                             action="#{itemEditor.doSearch}"/>
        </div>
```

```
        </div>
    </h:form>
    </body>
    </html>
```

❶ 每一个有效的XHTML文件都需要正确的文档类型声明。

❷ 除了常规的XHTML命名空间之外，你还给可视的HTML组件和核心的JSF组件（例如对输入验证）导入Facelets和两个JSF命名空间。

❸ 页面布局通过外部化到独立文件的CSS进行处理。

❹ 一般的页面头模板通过<ui:import>从Facelets导入。

❺ JSF表单（注意h命名空间）是一个HTML表单，如果被提交的话，则通过JSF servlet进行处理。

❻ JSF可以输出消息，例如验证错误。

❼ 每一个 都是一个标签或者表单字段，通过CSS类label或者input定义样式。

❽ 这是呈现为HTML输入字段的JSF输入文本组件。标识符对于将它绑定到错误消息输出很有用处，其大小定义了输入字段的可视大小，并且提交这张表单时，需要用户输入。最值得关注的部分是将输入字段的值绑定到一个backing bean（具名itemEditor）和该backing bean中的一个获取方法/设置方法对（具名getItemId()/setItemId()）。这个输入字段被绑定到该数据模型，并且JSF自动地将变化同步。

❾ JSF也支持输入验证，并提供全套的内建验证器。此处你声明了用户输入不能为负数（货品标识符是正整数）。

❿ 表单的提交按钮有一个动作绑定（action binding），绑定到具名itemEditor的backing bean的方法doSearch()。动作执行之后发生什么，取决于该方法的结果。

以下是页面在浏览器中呈现后的样子（见图17-1）。

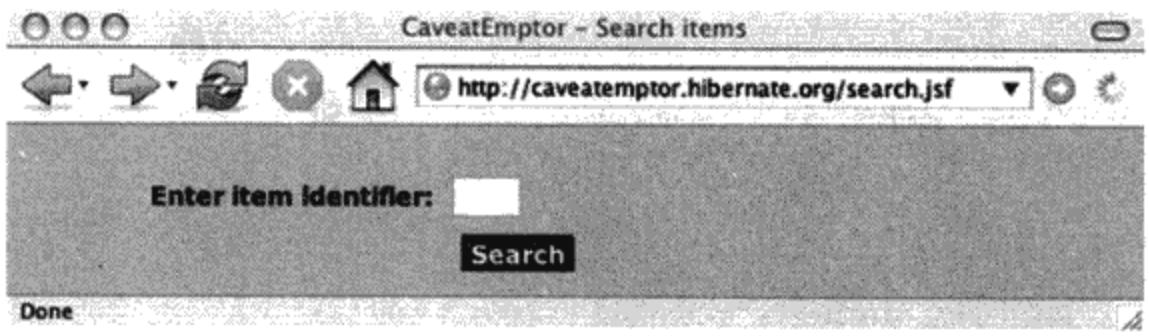


图17-1　利用JSF小部件的搜索页面

如果看一下这个URL，就会发现页面名称已经加上了.jsf的后缀，你或许原以为是search.xhtml。.jsf后缀是一个servlet映射；每当调用以.jsf结尾的URL时都会运行JSF servlet，并在安装完Facelets之后，在web.xml中将它配置为内部使用.xhtml。换句话说，search.xhtml页面通过JSF servlet得到呈现。

如果单击Search（搜索）按钮而没有输入搜索值，页面上就会出现错误消息。如果试图输入

一个非整数或者非正整数的值时，也会出现错误消息，这些全部由JSF自动进行处理。

如果输入一个有效的货品标识符值，并且backing bean在数据库中找到了该货品，就会把你转到货品编辑屏幕。（让我们在关注backing bean中的应用程序逻辑之前完成用户界面。）

3. 编写编辑页面

编辑页面显示了已经搜索到的货品详细信息，并允许用户编辑这些信息。当用户决定保存他的修改时，并在所有的验证都成功之后，应用程序再次显示搜索页面。

编辑页面的源代码如代码清单17-3所示。

代码清单17-3 包含详细表单信息的edit.xhtml页面

```
<!DOCTYPE html PUBLIC
...
<html xmlns=
...
<head>
...
<body>
...
<h2>Editing item: #{itemEditor.itemId}</h2>                    ❶

<h:form>

    <span class="errors"><h:messages/></span>
    <div class="entry">
    <div class="label">Name:</div>
    <div class="input">
        <h:inputText required="true" size="25"                 ❷
                     value="#{itemEditor.itemName}">
            <f:validateLength minimum="5" maximum="255"/>
        </h:inputText>
    </div>
</div>
<div class="entry">
    <div class="label">Description:</div>
    <div class="input">
        <h:inputTextarea cols="40" rows="4" required="true"
                         value="#{itemEditor.itemDescription}">
            <f:validateLength minimum="10" maximum="4000"/>
        </h:inputTextarea>
    </div>
</div>
<div class="entry">
    <div class="label">Initial price (USD):</div>
    <div class="input">
        <h:inputText size="6" required="true"
                     value="#{itemEditor.itemInitialPrice}" >
            <f:converter converterId="javax.faces.BigDecimal"/>
        </h:inputText>
    </div>
</div>
<div class="entry">
```

```
            <div class="label"> </div>
            <div class="input">
                <h:commandButton value="Save" styleClass="button"
                                 action="#{itemEditor.doSave}"/>
            </div>
        </div>

</h:form>

</body>
</html>
```

❶ 可以把值绑定的表达式放在任何组件之外。在这个例子中，调用了itemEditor backing bean中的getItemId()方法，且返回值最终在HTML页面中。

❷ 值绑定再次被用来将输入文本字段绑定到backing bean中的获取方法/设置方法对（或者字段）。

❸ 这个动作绑定引用了itemEditor backing bean中的doSave()方法。根据该方法的结果，要么再次显示页面（包含错误消息），要么将用户转到搜索页面。

图17-2展现了所呈现的页面。

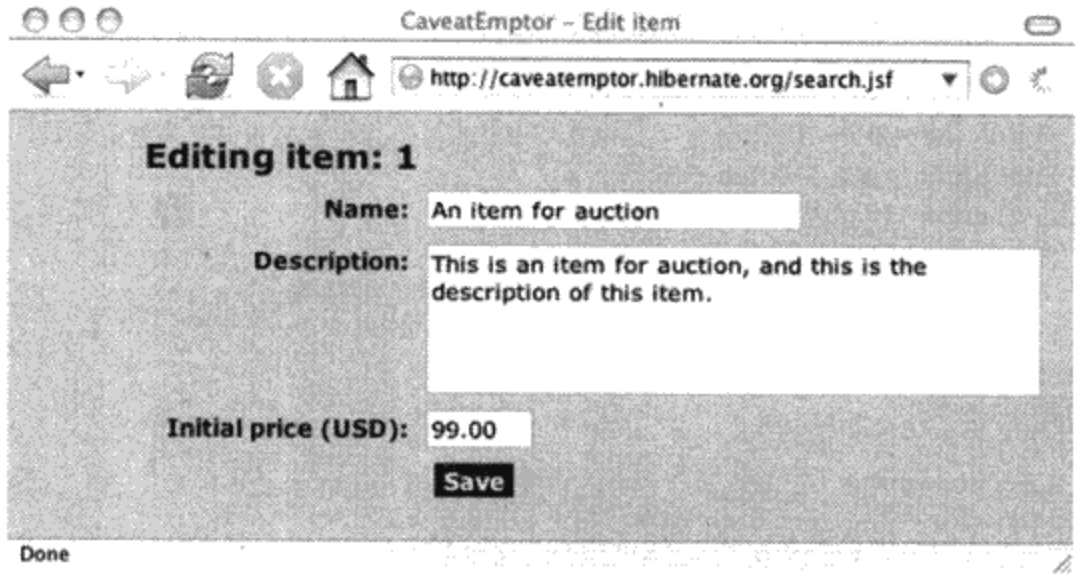


图17-2　加载有货品详细信息的编辑页面

为什么URL显示为search.jsf？它不是应该为edit.jsf吗？考虑JSF servlet的请求处理。如果用户在search.jsf页面中单击Search（搜索）按钮，backing bean的doSearch()方法就会在输入验证之后运行。如果该方法的结果触发而转到edit.xhtml页面，这个文档就通过JSF servlet得到呈现，并且HTML被发送到浏览器。URL没有改变！用户无法给编辑页面制作书签，它在这个简单的应用程序中可是很有必要的。

既然已经完成了应用程序的最顶层（视图），再来考虑访问数据库的层（你可能称它为业务层）。由于访问SQL数据库是一个事务的操作，因此要编写一个EJB。

4．在EJB中访问数据库

如果之前已经使用过EJB 2.x（和Struts），那么在无状态会话bean中，访问数据库的代码最可

能是过程代码。让我们在EJB 3.0中来完成它（代码清单17-4）。

代码清单17-4 包含数据访问facade的无状态会话bean

```
package auction.beans;

import ...;

@Stateless                                                    ❶
@TransactionAttribute(TransactionAttributeType.REQUIRED)      ❷
public class EditItemBean implements EditItem {               ❸

    @PersistenceContext                                       ❹
    EntityManager em;

    public Item findById(Long itemId) {                       ❺
        return em.find(Item.class, itemId);
    }

    public Item save(Item item) {                             ❻
        return em.merge(item);
    }

}
```

❶ @Stateless注解把这个简单的Java类变成了一个无状态的会话bean。在运行时，会准备好一个实例池，且请求会话bean的每个客户端都从该池中获得实例来执行方法。

❷ 在这个会话bean中调用的所有方法都包装在一个系统事务中，它汇集了该过程期间可能用到的所有事务资源。既使你没有注解这个bean，它也是默认的。

❸ 会话bean需要一个接口。你通常直接实现这个接口。EditItem接口有两个方法。

❹ 当运行时容器从池中分发会话bean实例时，它就注入了EntityManager，并且把（新的）持久化上下文界定到事务。

❺ 如果客户端调用findById()，就启动系统事务。EntityManager操作在该事务中执行一个SQL查询；当事务提交（方法返回）时，持久化上下文被清除并关闭。返回的Item实体实例处于脱管状态。

❻ 如果客户端调用save()，就启动系统事务。给定的脱管实例被合并到一个（新的）持久化上下文。在这个脱管的Item实例上所做的任何改变都被清除并提交到数据库。返回指向目前最新Item实例的新句柄。当方法返回时，这个新Item实例再次处于脱管状态，且持久化上下文也关闭了。

你可以称代码清单17-4中所示的会话bean为数据访问对象（data access object，DAO）。它也可以是会话外观。这个应用程序还不够复杂，难以进行清楚的区分；如果更多的非数据访问方法被添加到它的接口，会话bean就将作为业务层的一部分与传统的（主要是过程的）会话外观进行交互。

仍然漏掉了一个疑点：JSF输入小部件和按钮都有值和动作绑定到backing bean。这个backing bean与会话bean一样，还是你必须另外编写一个类呢？

5. 利用backing bean连接各个层

如果没有Seam，就必须编写一个将JSF 小部件状态和动作连接到事务的无状态会话bean的backing bean。这个backing bean有着在页面中通过表达式引用的获取方法和设置方法。它也可以与会话bean进行对话，并执行事务的操作。backing bean的代码如代码清单17-5所示。

代码清单17-5　一个JSF backing bean组件连接多个层

```
package auction.backingbeans;

import ...

public class ItemEditor {                                    ❶

    private Long itemId;                                     ❷
    private Item item;                                       ❸

    public Long getItemId() {                                ┐
        return itemId;                                       │
    }                                                        │
                                                             │
    public void setItemId(Long itemId) {                     │
        this.itemId = itemId;                                │
    }                                                        │
                                                             │
    public String getItemName() {                            │
        return item.getName();                               │
    }                                                        ❹
                                                             │
    public void setItemName(String itemName) {               │
        this.item.setName( itemName );                       │
    }                                                        │
                                                             │
    public String getItemDescription() {                     │
        return item.getDescription();                        │
    }                                                        │
                                                             │
    public void setItemDescription(String itemDescription) { ┘
        this.item.setDescription( itemDescription );
    }
    public BigDecimal getItemInitialPrice() {                ┐
        return item.getInitialPrice();                       ❹
    }                                                        ┘

    public void setItemInitialPrice(BigDecimal itemInitialPrice) {
        this.item.setInitialPrice( itemInitialPrice );
    }

    public String doSearch() {                               ❺
        item = getEditItemEJB().findById(itemId);
        return item != null ? "found" : null;
    }

    public String doSave() {                                 ❻
        item = getEditItemEJB().save(item);
        return "success";
    }
```

```
    private EditItem getEditItemEJB() {                    ❼
        try {
            return (EditItem)
                    new InitialContext()
                        .lookup("caveatemptor/EditItemBean/local");
        } catch (NamingException ex) {
            throw new RuntimeException(ex);
        }
    }
}
```

❶ 你没有实现任何接口，这是一个简单的Java类。

❷ backing bean内部利用一个字段维护货品标识符。

❸ backing bean也保存着正被用户编辑的Item实例。

❹ 这些是绑定在search.xhtml和edit.xhtml中的所有值的获取方法和设置方法。它们都是被JSF用来使backing bean内部模型状态与UI小部件状态同步的方法。

❺ doSearch()方法被绑定到一个JSF按钮的动作。它在backing bean中用EJB会话bean组件查找Item实例中当前的itemId。它的结果为字符串found或者null。

❻ doSave()方法被绑定到一个JSF按钮的动作。它用EJB会话bean组件来保存item字段的状态。[由于这是一个合并，你必须用返回的值（合并之后的状态）更新item字段。]它的结果为字符串success或者异常。

❼ 辅助方法getEditItemEJB()在EJB会话bean中获得一个句柄。如果运行时环境支持Java Servlet 2.5规范，JNDI中的这个查找可以用自动的依赖注入来替换。（在编写本书之时，Tomcat 5.5还只实现Java Servlets 2.4，Tomcat 6还处在alpha阶段。）

backing bean是一个由JSF运行时托管的组件。你在这些页面中使用的表达式通过名称itemEditor指向一个backing bean。在JSF XML配置文件中（通常是WEB-INF/faces-config.xml），你将这个名称映射到backing bean类（见代码清单17-6）。

代码清单17-6 一个JSF配置描述backing bean和导航流

```
<?xml version="1.0"?>
<!DOCTYPE faces-config PUBLIC
  "-//Sun Microsystems, Inc.//DTD JavaServer Faces Config 1.0//EN"
  "http://java.sun.com/dtd/web-facesconfig_1_1.dtd" >

<faces-config>

    <managed-bean>
        <managed-bean-name>itemEditor</managed-bean-name>
        <managed-bean-class>
            auction.backingbeans.ItemEditor
        </managed-bean-class>
        <managed-bean-scope>session</managed-bean-scope>
    </managed-bean>

    <navigation-rule>
```

```
        <from-view-id>/search.xhtml</from-view-id>
        <navigation-case>
            <from-outcome>found</from-outcome>
            <to-view-id>/edit.xhtml</to-view-id>
        </navigation-case>
    </navigation-rule>

    <navigation-rule>
        <from-view-id>/edit.xhtml</from-view-id>
        <navigation-case>
            <from-outcome>success</from-outcome>
            <to-view-id>/search.xhtml</to-view-id>
        </navigation-case>
    </navigation-rule>

</faces-config>
```

这个应用程序有1个backing bean和2个导航规则。backing bean通过itemEditor名称进行声明，并由auction.backingBeans.ItemEditor实现。JSF页面中的表达式现在可以在backing bean中以一种松散耦合的方式按名称引用方法和字段。JSF servlet管理backing bean的实例，每个HTTP会话一个实例。

更进一步：JSF页面中的表达式是一个字符串，例如#{itemEditor.itemId}。这个表达式基本上导致对具名itemEditor变量的搜索。以这样的顺序进行搜索：先是当前的请求，然后是当前的HTTP会话，最后是当前的应用程序上下文。如果JSF页面呈现且这个表达式必须取值，那么或者在HTTP会话上下文中找到包含该名称的变量，或者JSF servlet创建新backing bean实例，并将它绑定到HTTP会话上下文中。

导航规则声明哪个页面在一个动作结果之后进行呈现。这是一个从字符串（由动作返回）到页面的映射。

现在应用程序完成了，下面对它进行深入的分析。

17.1.4　分析应用程序

可能你会看着前几节中的代码想："有这么多代码要写，要把四个表单字段放到Web页面上，并把它们连接到数据库中的四个列上。"或者，如果你已经在EJB 2.x和Struts上花了很多时间，可能会说："太好了，我不必再自己管理HTTP会话了，并且所有的EJB样板代码也都不见了。"

这两种说法都对。Java EE 5.0，尤其JSF和EJB 3.0都是朝着Java Web应用程序前进的一大步，但是并非一切都完美无瑕。现在来看一下Java EE 5.0的优势，并把它与J2EE 1.4和JSF之前的Web框架进行比较。但我们也努力发现可以改善的东西、可以避免的代码，以及可以被简化的策略。这就是后来Seam出现的契机。

1．将代码与J2EE进行对比

如果你有J2EE 1.4/Struts的背景，这个JSF和EJB 3.0应用程序看起来已经比较吸引人了。要创建的东西比传统Java Web应用程序中的更少——例如，可以从会话bean外观中脱管一个Item实

例，并把它转换到JSF backing bean里面。利用EJB 2.x实体bean时，你还需要数据迁移对象（DTO）来完成这个。

代码更加简洁。利用EJB 2.x时，会话bean必须实现SessionBean接口，并实现所有的维护方法。在EJB 3.0中，这将通过简单的@Stateless注解进行解析。也没有手工将HTML表单字段的状态绑定到动作监听器中实例变量的Struts ActionForm代码。

整体而言，应用程序是透明的，没有在HTTP会话或者HTTP请求中维持值的模糊调用。JSF透明地在这些上下文中放置和查找值。

如果你考虑Item类的对象/关系映射，可能会认同说：POJO中的一些注解比EJB 2.x实体bean的部署描述符更简单。甚至，Java Persistence定义的对象/关系映射不仅比EJB 2.x实体bean更强大、更具有丰富的特性，而且还更容易使用（即使与原生的Hibernate比较）。

无法在简单的应用程序中展现JSF和EJB 3.0背后的威力。JSF是相当的灵活且可扩展，可以编写自己的HTML小部件，可以钩入JSF请求的处理阶段，甚至可以轻松地创建自己的视图层（如果你不想要Facelets的话）。EJB 3.0比EJB 2.x更容易掌控，它还具有之前标准的Java编程模型中从未有过的一些特性（例如拦截器和依赖注入）。

应用程序可以轻松地通过测试框架（如JUnit或者TestNG）进行测试。所有的类都是简单的Java类，可以实例化它们，手工设置（模拟）依赖，并运行一个测试过程。

无论如何，还是有改善的空间。

2. 改善应用程序

在这个JSF/EJB 3.0应用程序中突出的第一个东西是JSF backing bean。这个类做什么用呢？它是JSF需要的，因为你要将值和动作绑定到它的字段和方法上。但是代码没有完成任何有意义的事：它一个接一个地将任何动作传递到EJB。更糟糕的是，它是个包含最多代码行的工件。

你可能认为它从业务层解耦了视图。如果和一个不同的视图层（比如富客户端）一起使用EJB的话，这似乎有道理。但是，如果应用程序是一个简单的Web应用程序，backing bean则导致各层之间的耦合更加紧密。你对视图或者业务层所做的任何改变，也要求改变包含最多代码行的组件。如果想要改善代码，就去掉工件层，并移除backing bean。EJB没有理由不应该是backing bean。编程模型不应该强制你给应用程序分层（它也不应该限制你）。为了改善应用程序，你需要禁用工件层。

应用程序不会在几个浏览器窗口中工作。想象你在两个浏览器窗口中打开搜索页面，在第一个页面上搜索货品1，在第二个页面上搜索货品2。这两个浏览器窗口都给你显示一个编辑屏幕，包含货品1和货品2的详细信息。如果你改变了货品1并单击Save（保存），会发生什么呢？这些变化却改变了货品2！如果在第一个浏览器窗口中单击Save，你就是在HTTP会话中所呈现的状态下工作，那里是backing bean所处的地方。然而，backing bean不再保存货品1——现在的当前状态是第二个浏览器窗口（编辑货品2）的状态。换句话说，你启动了应用程序的两个会话，但是会话之间并非彼此隔离。HTTP会话不是并发对话状态的正确上下文，它在浏览器窗口之间共享。你无法轻易修正这一点。使这个（繁琐的）应用程序在几个浏览器窗口中工作，需要进行重大的架构改变。如今，用户们希望Web应用程序可以在几个浏览器窗口中进行工作。

应用程序泄露内存。当一张JSF页面首先试图解析itemEditor变量时，ItemEditor的一个新实例就被绑定到了HTTP会话上下文中的变量中。这个值从来不会被清除。即使用户在编辑屏幕上单击Save，backing bean实例也会保留在HTTP会话中，直到用户注销或者HTTP会话超时。想象一个更为复杂的应用程序，它有许多表单和许多backing bean。HTTP会话随着用户通过应用程序单击而增长，并且将HTTP会话复制到集群中的其他节点的成本，随着每一次单击而变得越来越昂贵。如果用户在用完应用程序的另一个模块之后回到货品搜索屏幕，那么表单中显示的还是旧数据。针对这个问题的一种解决方案是，在对话结束时，手工清除HTTP会话，但是完成这项工作没有很容易的方法。使用JSF和EJB 3.0时，你必须手工编写这个代码。依据我们的经验，手工处理HTTP会话中的变量和值，是那些难以追踪到的问题的一般根源。

应用程序流很难形象化和控制。如何知道单击Search按钮后会把你引向哪里？至少，你必须观察两个文件：backing bean（它返回字符串结果）和JSF XML配置文件（它定义为特定的结果显示的页面）在浏览器中也有“后退”按钮曾经出现过的问题，这是有两个以上屏幕的任何对话中一个讨厌的问题。

还要考虑业务流程（business process）。如何将页面流定义为一个更大业务流程的一部分？想象搜索和编辑一件货品只是涉及更多步骤的业务流程中的唯一任务——例如，作为预览流程的一部分。在应用程序中，没有工具或者策略可以帮助你整合业务流程管理。如今，你可以不再忽略以应用程序作为其中一部分的业务流程了，你需要一个支持业务流程管理的编程模型。

最后，这个应用程序包括了太多的XML。应用程序中的元数据没什么可说的，但是它并非要全部都在一个XML文件中。如果XML文件中的元数据独立于代码而改变，将会很棒。对于导航规则可能是这样，但是对于backing bean的声明和它们所处的上下文来说，或许不可能。这种元数据随着应用程序的大小呈线性增长——每一个backing bean都必须在XML中进行声明。反之，应该将一个注解放在类中，表明：“我是JSF的一个backing bean，我处在HTTP会话（或者任何其他的）上下文里面。”你的类将不可能突然改变它的角色，而不改变类代码。

如果你同意这种分析，就会喜欢上Seam。

17.2　用 Seam 改善应用程序

如果使用Seam，前面编写好用来搜索和编辑Web项目的Web应用程序还可以改进。从基础的Seam特性开始说起：

- Seam使JSF backing bean变得没有必要。可以直接将JSF小部件值和动作绑定到EJB有状态和无状态会话bean。Seam引入了一个统一的组件模型：所有的类都可以通过注解转变成Seam组件。组件通过字符串表达式以松散耦合的方式被连接在一起。
- Seam自动引入新的上下文，并管理组件范围。这个富上下文模型包括对应用程序有意义的逻辑上下文，例如对话或者业务流程上下文。
- Seam引入一个有状态的编程模型，它对于对话很棒。包含Seam托管对话的有状态应用程序可以在多个浏览器窗口中工作，无需额外的工作。

这是对Seam功能的一个简短罗列，稍后你还会用到更多。让我们先创建一个基本对话的、

有状态的、简单的Seam应用程序。第一步是建立和配置Seam。

如果想要沿用示例的代码，就从http://caveatemptor.hibernate.org为Seam下载CaveatEmptor包，并在IDE中打开它。如果以后想要编写自己的Seam工程，这也是一个好的起点。

17.2.1 配置 Seam

图17-3展现了在接下来的几节中，你对Web应用程序进行改变之前和之后的文件。

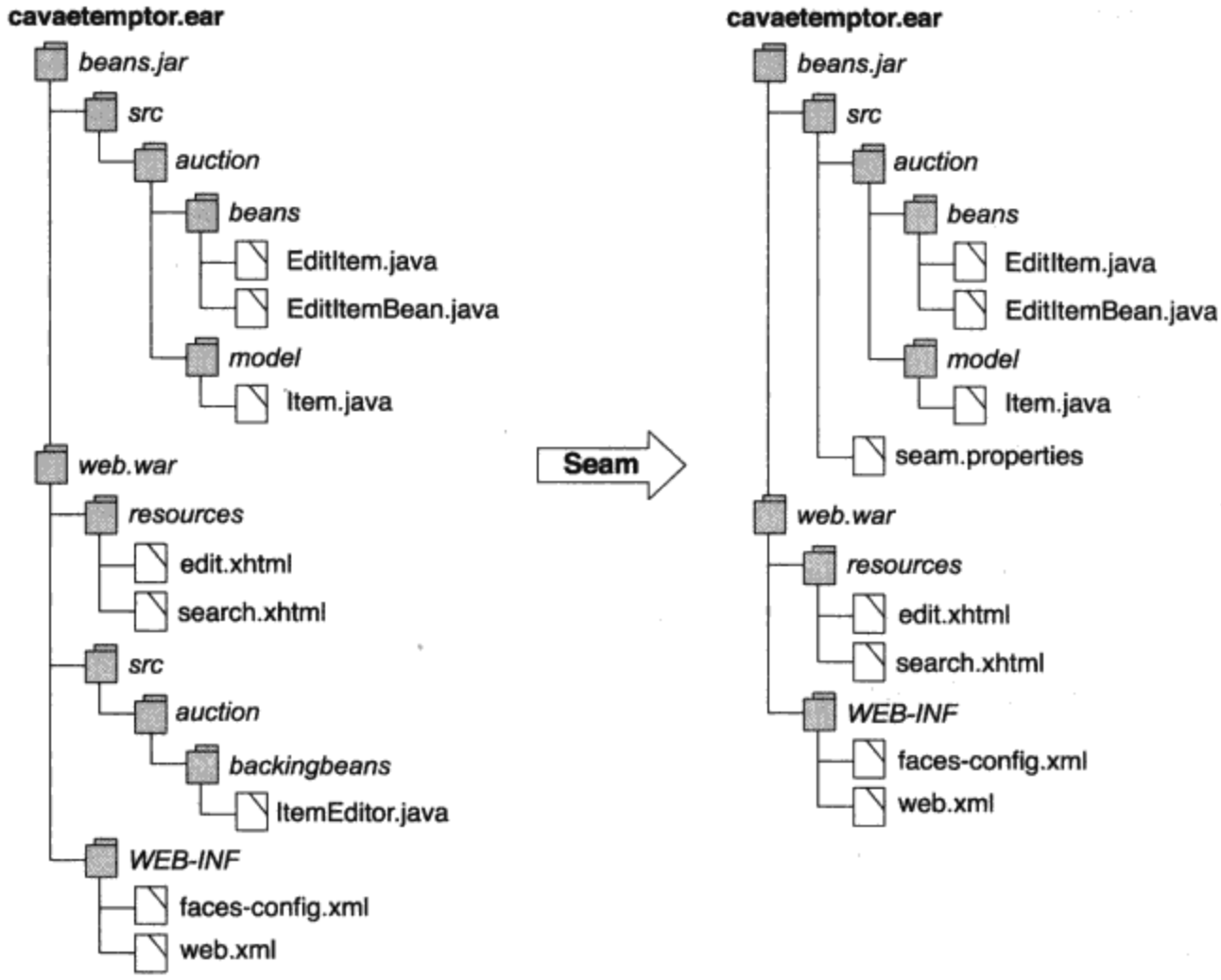


图17-3 引入Seam之前和之后的应用程序文档

发生了两处重大的变化：不再需要JSF backing bean，并且beans.jar文档有了一个新文件：seam.properties。这个文件给这个简单的应用程序包含了两个Seam配置选项（代码清单17-7）。

代码清单17-7 一个简单的seam.properties配置文件

```
org.jboss.seam.core.init.jndiPattern = caveatEmptor/#{ejbName}/local
org.jboss.seam.core.manager.conversationTimeout = 600000
```

第一个设置对于Seam与EJB 3.0容器的整合是必要的。因为现在Seam负责在运行时连接组件实例，它需要知道如何通过查找获得EJB。这里介绍的JNDI模式是用于JBoss应用程序服务器的。（Seam在任何Java EE 5.0服务器中运行，甚至可以在常规的Tomcat中运行（无论是否使用EJB 3.0）。我们认为从JBoss应用程序服务器开始最为方便，因为你不需要安装任何额外的服务。）

为了使Seam与EJB 3.0完全整合，Seam还需要拦截对EJB的所有调用。这很容易做到，因为EJB 3.0支持定制的拦截器。你不会在类的代码中看到任何拦截器，因为它们通常通过一个全局的通配符得到定义，该通配符匹配META-INF/ejb-jar.xml中的所有EJB（这里未显示出来）。如果下载了Seam实例，它就会有这个文件。

seam.properties中的第二个设置定义了Seam可以在600 000毫秒（10分钟）之后销毁非活动的用户对话。这个设置在用户决定去用餐时，释放HTTP会话中的内存。

seam.properties文件不仅是Seam的一个配置文件——它还是一个标识。当Seam启动时，它给Seam组件（包含正确注解的类）扫描类路径和所有的文档。但是，扫描所有JAR的成本太过昂贵，因此Seam只在根路径下递归地扫描包含seam.properties文件的JAR文件和目录。即使没有任何配置设置，在包含Seam组件类的文档中也要有一个空的seam.properties文件。

可以在web.xml和faces-config.xml中找到更多的Seam配置选项，以及与JSF和servlet容器的整合。稍后会回来讨论faces-config.xml，web.xml则没有什么值得关注的（请见CaveatEmptor包中注释的文件）。

Seam也可以通过components.xml文件在WAR包中的WEB-INF目录下进行配置。当你稍后需要更复杂的组件配置时会用到它。（许多Seam被写成Seam组件。字符串`org.jboss.seam.core.manager`是一个组件名称，`conversationTimeout`是你可以访问的属性，就像访问任何其他组件属性一样。）

下一步是用Seam组件替换JSF backing bean。

17.2.2　将页面绑定到有状态的 Seam 组件

search.xhtml页面根本没有改变；回顾代码清单17-2中的代码。这个页面有一个绑定到`itemEditor.itemId`的值和一个绑定到`itemEditor.doSearch`的动作。当页面由JSF servlet呈现时，这些表达式被取值，并且小部件被绑定到`itemEditor` bean中相应的方法上。

1. EJB组件接口

`itemEditor` bean现在是一个EJB。这个EJB的接口是`EditItem.java`（见代码清单17-8）。

代码清单17-8　一个有状态组件的接口

```
package auction.beans;

import ...

public interface EditItem {

    // Value binding methods
    public Long getItemId();
    public void setItemId(Long itemId);

    public Item getItem();

    // Action binding methods
    public String doSearch();
```

```
    public String doSave();

    // Cleanup routine
    public void destroy();
}
```

前两个方法是用于页面的搜索输入文本字段的值绑定的获取方法和设置方法。getItem()方法（此处你不需要设置方法）稍后将为编辑页面所用。doSearch()方法被绑定到Search按钮，doSave()将被绑定到编辑页面中的一个按钮。

这是用于有状态组件的一个接口。当有状态组件第一次被请求时就被实例化——例如，因为页面第一次呈现。每一个有状态的组件都需要一个方法，当组件被销毁时，运行时环境可以调用它。可以使用doSave()方法，并假设组件的生命周期在这个方法完成时终止，但是你很快会明白为什么独立的方法更整洁。

接下来看一下这个接口的实现。

2. EJB组件实现

EJB 3.0中标准的有状态组件是一个有状态会话bean。EditItemBean.java中的实现是一个包含几个额外注解的POJO类。在代码清单17-9中，所有的Seam注解都以粗体显示。

代码清单17-9 一个有状态组件的实现

```
package auction.beans;
import ...

@Name("itemEditor")                          ❶
@Scope(ScopeType.CONVERSATION)               ❷

@Stateful                                    ❸
public class EditItemBean implements EditItem {

    @PersistenceContext                      ❹
    EntityManager em;

    private Long itemId;                     ❺
    public Long getItemId() { return itemId; }
    public void setItemId(Long itemId) { this.itemId = itemId; }

    private Item item;
    public Item getItem() { return item; }

    @Begin                                   ❻
    public String doSearch() {
        item = em.find(Item.class, itemId);

        if (item == null)
            FacesMessages.instance().add(
                "itemSearchField",
                new FacesMessage("No item found.")
            );

        return item != null ? "found" : null;
```

```
    }
                    ❼
    @End
    public String doSave() {
        item = em.merge(item);
        return "success";
    }
                    ❽
    @Destroy
    @Remove
    public void destroy() {}
}
```

❶ Seam @Name注解给这个Seam组件声明名称。它也把这个EJB转变成一个Seam组件。现在可以用这个名称随处引用这个组件。

❷ 当需要这个组件的一个实例时，Seam就替你把它实例化。Seam把该实例放到一个以它为名的上下文中。这是正规的描述：EditItem的实例由Seam在对话上下文中托管，作为上下文变量itemEditor的一个值。

❸ POJO需要EJB 3.0 @Stateful注解，来变成一个有状态的会话bean。

❹ EJB 3.0容器用一个新的持久化上下文，在bean的任何客户端调用方法之前，将EntityManager注入到这个bean里面。当方法返回时关闭持久化上下文（假设这个方法调用也是系统事务的范围，它在这里是默认的）。

❺ 这个有状态的组件在内部的字段itemId和item中保存着状态。状态通过属性访问方法公开。

❻ Seam @Begin注解标识一个启动长运行对话的方法。如果一个JSF动作触发对这个方法的调用，Seam就跨越HTTP请求维持这个组件的状态。doSearch()方法返回字符串结果（或者null），并生成一个可以在页面上呈现的JSF消息。Seam FacesMessages辅助工具使得这个消息传递变得很容易。

❼ Seam @End注解标识一个终止长运行对话的方法。如果一个JSF动作触发对这个方法的调用，并且方法返回，Seam将销毁组件的状态，并且不再跨越HTTP请求来维持它。

❽ Seam @Destroy注解标识当组件状态必须被销毁（对话已经终止）时，由Seam调用的方法。这对于内部清除很有用（在这个例子中，没有东西可清除）。EJB 3.0 @Remove注解标识一个客户端（在这个例子中为Seam）必须调用来移除有状态会话bean实例的方法。这两个注解通常出现在同一个方法中。

为什么不用@End、@Destroy和@Remove标识doSave()呢？doSave()方法可能抛出异常，并且这个异常必须回滚任何系统事务。然而，Seam记录并吞没由它的@Destroy方法抛出的任何异常，因此你经常在有状态的Seam组件中看到空的销毁方法。此外，保存了一件货品之后，还需要组件实例一阵子，来呈现响应。

这个EJB实现封装了所有的应用程序逻辑，其他任何地方再没有Java代码（哦，还有Item实体类）。如果忽略繁琐的代码，应用程序逻辑在两个动作方法中就只有四行了。

还需要再做两处改变以使应用程序生效。edit.xhtml中的一些值绑定需要进行修改，并且定义旧JSF backing bean的XML块可以从faces-config.xml中移除。

3. 绑定值和动作

打开edit.xhtml，并改变JSF输入小部件的值绑定，如代码清单17-10中所示。

代码清单17-10 edit.xhtml页面被绑定到一个Seam组件

```
...
<h2>Editing item: #{itemEditor.itemId}</h2>

<h:form>

    <span class="errors"><h:messages/></span>

    <div class="entry">
        <div class="label">Name:</div>
        <div class="input">
            <h:inputText required="true" size="25"
                         value="#{itemEditor.item.name}">
                <f:validateLength minimum="5" maximum="255"/>
            </h:inputText>
        </div>
    </div>
    <div class="entry">
        <div class="label">Description:</div>
        <div class="input">
            <h:inputTextarea cols="40" rows="4" required="true"
                             value="#{itemEditor.item.description}">
                <f:validateLength minimum="10" maximum="4000"/>
            </h:inputTextarea>
        </div>
    </div>

    <div class="entry">
        <div class="label">Initial price (USD):</div>
        <div class="input">
            <h:inputText size="6" required="true"
                         value="#{itemEditor.item.initialPrice}" >
                <f:converter converterId="javax.faces.BigDecimal"/>
            </h:inputText>
        </div>
    </div>

    <div class="entry">
        <div class="label">&amp;#160;</div>
        <div class="input">
            <h:commandButton value="Save" styleClass="button"
                             action="#{itemEditor.doSave}"/>
        </div>
    </div>

</h:form>
...
```

改变了的绑定是关于名称、描述和初始价格输入字段的表达式。它们现在引用itemEditor.

item，后者可以被解析到Seam组件的getItem()方法。JSF在返回的Item实体中调用getName()和setName()，来使小部件的状态同步。同样的技术用于绑定和同步该货品的描述和初始价格。当用户输入新价格时，由itemEditor组件保存的Item实例的initialPrice被自动更新。

对Save按钮的动作绑定没有变——itemEditor组件的doSave()方法仍然是正确的监听器。可以看到逻辑组件名称和表达语言如何允许你轻松地耦合视图和业务层，并且不会太紧。

最后，更新faces-config.xml，如代码清单17-11所示。

代码清单17-11　没有backing bean的JSF配置文件

```
...
<faces-config>

    <navigation-rule>
        <from-view-id>/search.xhtml</from-view-id>
        <navigation-case>
            <from-outcome>found</from-outcome>
            <to-view-id>/edit.xhtml</to-view-id>
        </navigation-case>
    </navigation-rule>
    <navigation-rule>
        <from-view-id>/edit.xhtml</from-view-id>
        <navigation-case>
            <from-outcome>success</from-outcome>
            <to-view-id>/search.xhtml</to-view-id>
        </navigation-case>
    </navigation-rule>

    <!-- Integrate Seam with the JSF request processing model -->
    <lifecycle>
        <phase-listener>
            org.jboss.seam.jsf.SeamPhaseListener
        </phase-listener>
    </lifecycle>

</faces-config>
```

将这段代码与代码清单17-7中的前一个JSF配置进行对比。backing bean声明不见了（移到了EJB中的两个Seam注解）。阶段监听器是新的：Seam必须钩入JSF servlet，并监听每个HTTP请求的处理。定制的JSF阶段监听器将Seam与JSF进行整合。

上面介绍了一些新的概念，如果这是你初次接触Seam，这些概念你可能还从未见过。下面来更深入地分析一下应用程序，并看一看之前对简单的JSF和EJB 3.0应用程序所确定的问题是否已经得到解决。

17.2.3　分析 Seam 应用程序

Web应用程序的接口还没有改变，它看起来一模一样。用户唯一可能注意到的东西是他们可以在几个浏览器窗口中搜索和编辑货品，而不会重叠状态和数据修改。

Seam推出了一个强大且确定的有状态应用程序编程模型。让我们顺着应用程序流（见图17-4），看看其内部的工作原理。

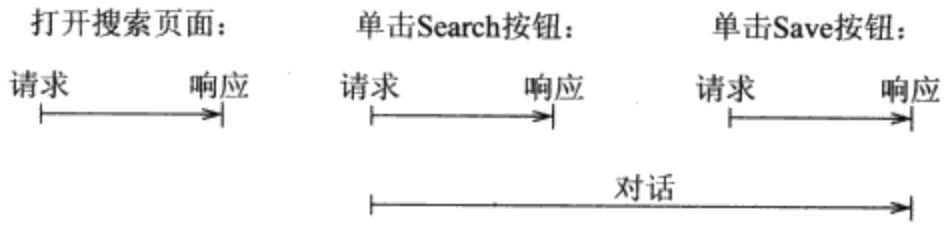


图17-4 应用程序的请求/响应流

1. 打开搜索页面

打开浏览器窗口，并输入/search.jsf URL时，一个HTTP GET请求就被发送到了JSF servlet。JSF处理生命周期开始了（见图17-5）。

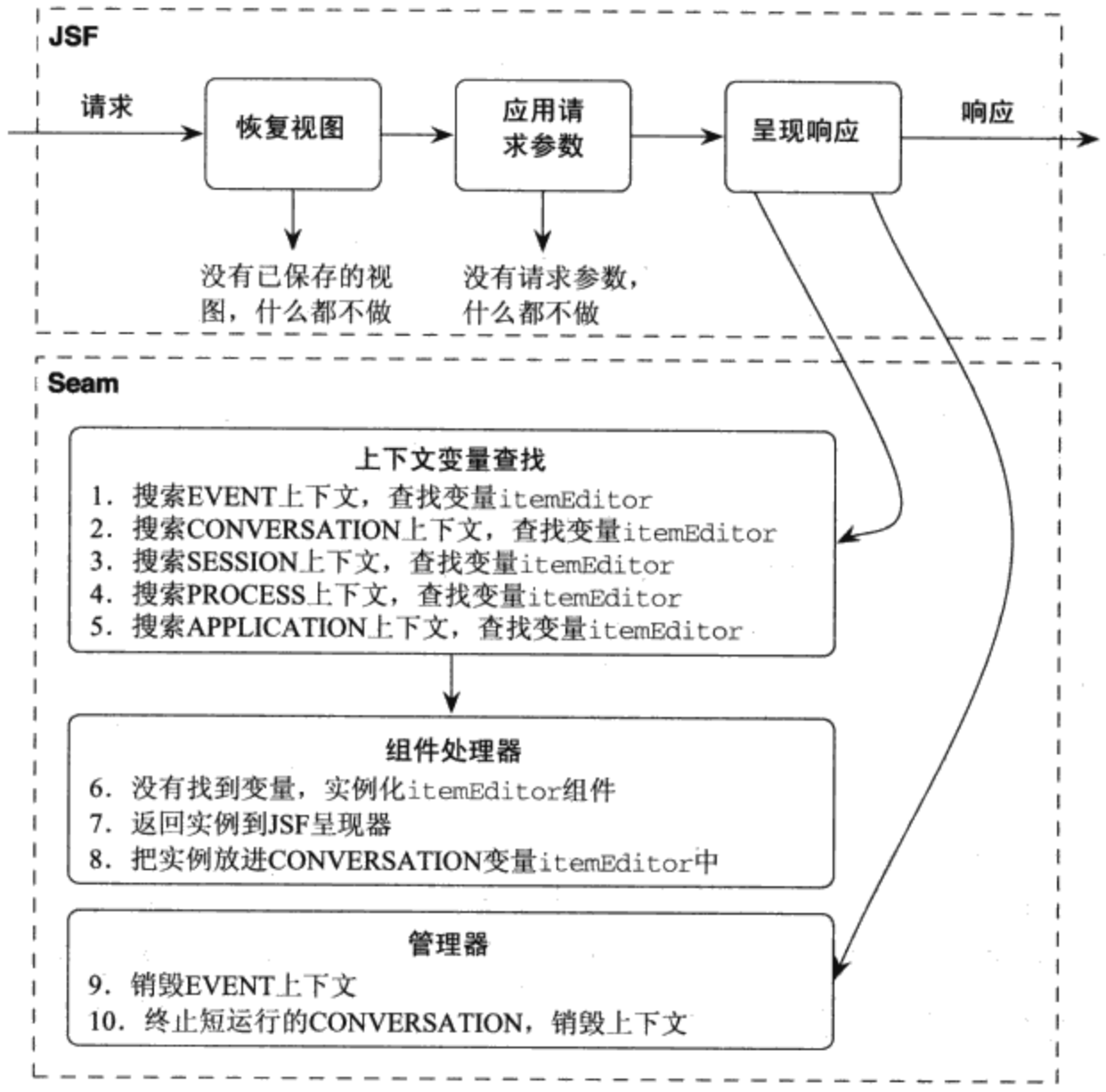


图17-5 当JSF呈现响应（搜索页面）时，Seam是活动的

直到JSF servlet进入请求处理的Render Response（呈现响应）阶段时，才会发生真正值得关注的事情。没有视图要恢复，并且没有必须应用到视图组件（小部件）的HTTP请求参数。

当响应（即search.xhtml文件）呈现时，JSF使用了一个变量解析器来给#{itemEditor.itemId}值绑定取值。Seam变量解析器比标准的JSF变量解析器更智能。它不在HTTP请求、HTTP会话和全局的应用程序上下文中查找itemEditor，而是在Seam逻辑上下文中查找。如果你认为这些逻

辑上下文是一样的就对了——我们很快会讨论关于这个话题的更多内容。至于现在，可以把Seam上下文当作被分层搜索［从范围最窄的上下文（当前的事件），到范围最宽的上下文（当前的应用程序）］的变量持有者。

无法找到变量itemEditor。因此，Seam的组件处理器开始寻找以该名字命名的Seam组件。它找到你编写过的有状态会话bean，并创建一个实例。然后这个EJB实例就被提供给JSF页面呈现器，呈现器把getItemId()的返回值放到搜索输入文本字段中去。方法返回null，因此第一次打开页面时字段为空。

Seam组件处理器也意识到有状态会话bean拥有@Scope(CONVERSATION)注解。因此实例被放入到对话上下文中，作为上下文变量itemEditor（组件的名称）的一个值。

当页面完全呈现时，Seam被再次调用（通过Seam阶段监听器）。Seam事件上下文有一个小范围：它只能在单个HTTP请求期间才能够保存变量。Seam销毁了这个上下文和其内部的一切（目前你不需要任何东西）。

Seam也销毁当前的会话上下文，以及随之刚刚创建的itemEditor变量。这可能让你感到很惊讶——你或许以为有状态会话bean对于几个请求会很好。然而，如果没有人在该请求期间把它提升为长运行对话，对话上下文的范围就是单个的HTTP请求。通过调用一个已经用@Begin做了标识的组件方法，将一个短的单请求对话提升为长运行对话。在这个请求中没有发生这一点。

搜索页面现在通过浏览器显示，应用程序等待用户输入和单击Search按钮。

2．搜索货品

当用户单击Search按钮时，一个HTTP POST请求就被发送到服务器，并由JSF（见图17-6）进行处理。必须看一下search.xhtml的源代码和EditItemBean，以便理解这个图例。

保存在前一个请求（通常在服务器的HTTP会话中）中的JSF，现在找到了一个表示视图（search.xhtml）的小部件树，并在内部对它进行重新创建。这个小部件树很小：它有一个表单、一个输入文本字段和一个提交按钮。在应用请求参数中，所有的用户输入都取自HTTP请求，并与小部件的状态同步。输入文本字段小部件现在保存着用户输入的搜索字符串。

提示　调试JSF小部件树——Facelets可以给你显示JSF 小部件树。把<ui:debug hotkey="D"/>放在你页面中的任何位置，并在浏览器中打开页面（当然是作为一个JSF URL）。现在按下Ctrl+Shift+d，打开一个包含JSF小部件/组件树的弹出式窗口。如果单击Scoped Variables，就会看到Seam把它的上下文和管理器保存在内部的什么地方（如果你不是Seam开发人员，这可能没有什么值得关注的）。

在处理验证期间，JSF验证器确保由用户输入的搜索字符串为非负整数，并且呈现输入值。如果验证失败，JSF servlet跳入呈现响应阶段，并再次利用错误消息呈现search.xhtml页面（这个阶段的处理看起来如图17-5）。

验证之后，JSF同步已经被绑定到小部件的模型对象的值。它调用itemEditor.setItemId()。这个变量由Seam解析，并在所有的Seam上下文中查找。由于没有在任何上下文中找到itemEditor变量，所以创建新的EditItemBean实例，并放入对话上下文中。setItemId()

方法在这个有状态会话bean实例中被调用。

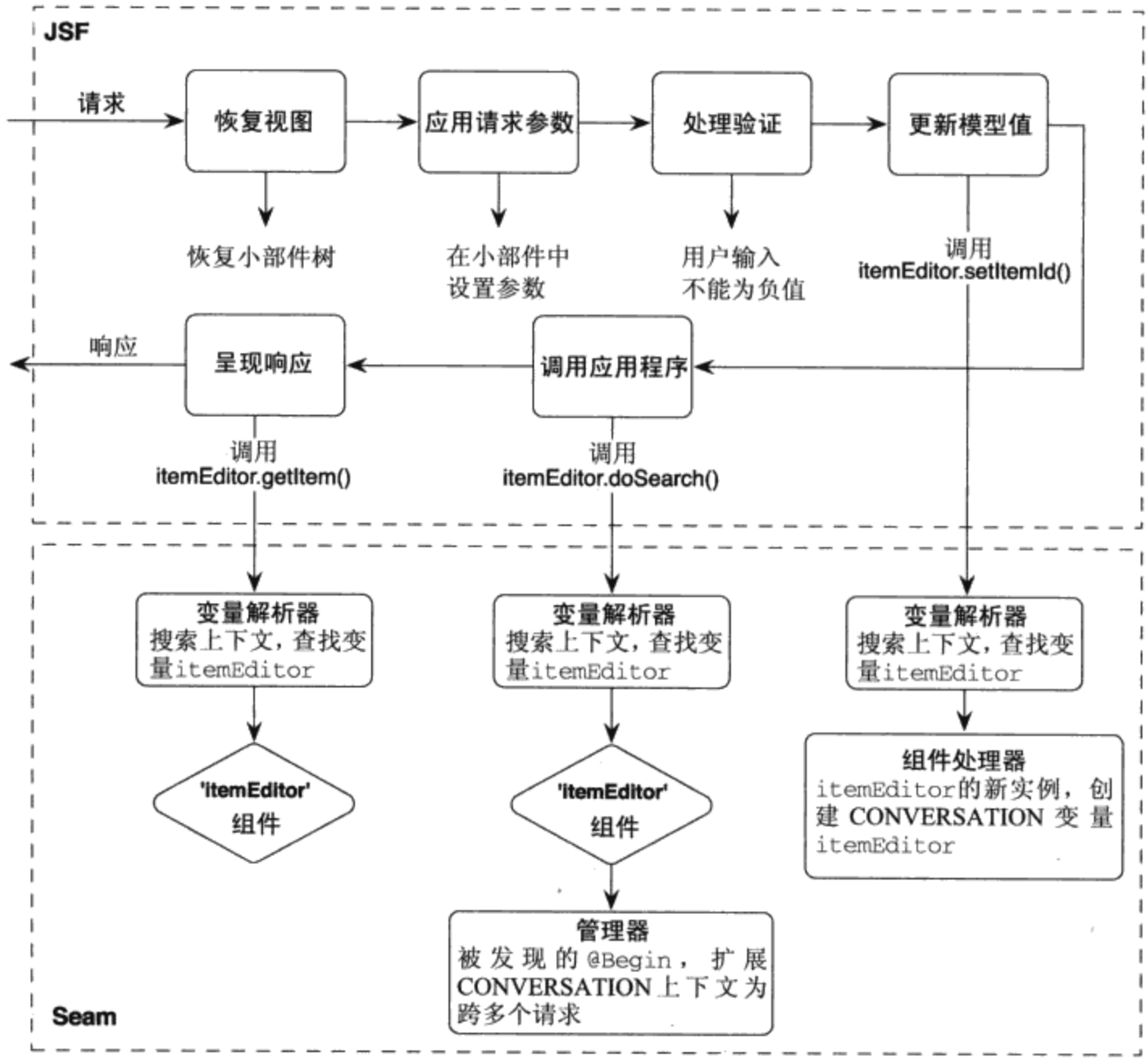


图17-6 Seam参与搜索动作的处理过程

JSF现在通过调用被绑定的方法itemEditor.doSearch执行请求的动作。Seam解析itemEditor变量，并在对话上下文中找到它。doSearch()方法在EditItemBean实例中被调用，并且EJB 3.0容器在该调用期间处理事务和持久化上下文。调用期间发生了两件事：itemEditor的item成员变量现在保存着在数据库中找到的一个Item实例（或者如果没有找到任何东西，就为null），并且@Begin注解将当前的对话提升为一个长运行对话。对话上下文由Seam保存，直到调用一个标有@End注解的方法。

doSearch()方法返回字符串found，或者null。这个结果由JSF取值，并应用来自faces-config.xml的导航规则。如果结果为null，search.xhtml页面就利用Item not found（货品没有找到）的错误消息进行呈现。如果结果为found，导航规则就声明edit.xhtml页面得到了呈现。

在edit.xhtml页面的呈现期间，变量itemEditor必须再次由JSF进行解析。Seam在对话上下文中找到itemEditor上下文变量，且JSF将页面上小部件的值（文本输出，文本输入）绑定到由

itemEditor.getItem()返回的货品实例的属性中。

提示　浏览Seam上下文——如果使用Seam调试屏幕，就可以更轻松地调试Seam应用程序。必须启用这个屏幕。要这么做，得编辑你的seam.properties文件，并添加org.jboss.seam.core.init.debug = true。现在，访问URL /debug.jsf，来浏览这个浏览器窗口的Seam上下文。可以看到当前对话中的所有变量和值：会话、过程和应用程序上下文。

请求结束时，Seam销毁它的事件上下文。对话上下文没有被销毁；应用程序的用户通过执行搜索启动长运行对话。当显示编辑页面时，应用程序等待用户输入。如果用户再一次在另一个浏览器窗口中搜索，很快就会启动并发运行的对话，并把它提升为长运行对话。这两个对话和它们的上下文自动被Seam隔离。

3. 编辑货品

当用户单击Save时，编辑表单通过一个HTTP POST请求被提交到服务器（见图17-7）。

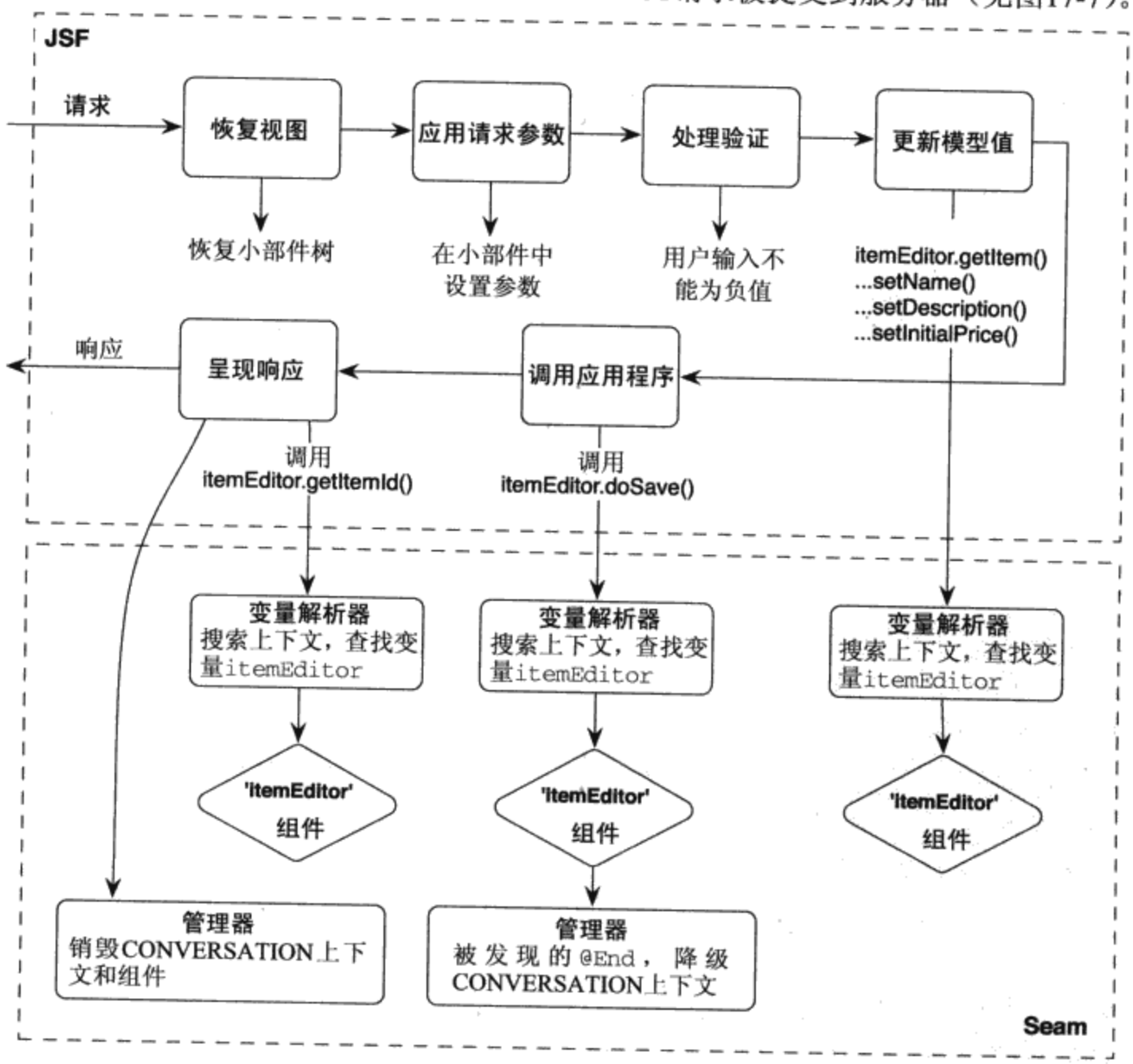


图17-7　Seam参与编辑动作的处理过程

在这个请求中恢复的视图是edit.xhtml，JSF重新创建了表单内部的一个小部件树和它所有的字段，并应用HTTP请求值。验证稍微复杂一些，你已经在edit.xhtml页面中定义了更多的JSF验证器。

验证成功之后，通过调用由itemEditor.getItem()返回的Item实例上的设置方法，JSF更新绑定的模型值。itemEditor绑定解析（通过Seam）到当前对话上下文中的一个上下文变量。Seam将会话上下文扩展到当前的请求中，因为它在前一个请求中被提升成为长运行对话了。

接下来，调用itemEditor.doSave()，变量再一次在会话上下文中被解析。EditItemBean中的代码抛出异常（如果EJB 3.0容器或者EntityManager抛出异常），或者返回字符串结果success。这个方法被标识为@End，因此Seam管理器标识在呈现响应阶段之后要清除的当前对话。

字符串结果success被映射到JSF导航规则中的/search.xhtml。在呈现响应期间，search.xhtml页面中的值绑定必须被解析。唯一的值绑定是#{itemEditor.itemId}，因此Seam再次试图在所有的上下文中找到itemEditor。使用来自（降了级但仍然活动着的）对话上下文的itemEditor，且getItemId()返回一个值。因此用户看到输入字段不是空的，而是显示了与对话开始时的输入相同的搜索值。

当呈现响应完成时，Seam移除了降了级的对话上下文，并销毁处在该上下文中的所有有状态的组件实例。destroy()方法在EditItemBean中被调用。因为它通过@Remove被标识，EJB 3.0容器也内部清除有状态会话bean。现在用户看到搜索页面，可以开始另一个对话了。

如果还从未使用过JSF，这就有很多信息需要消化了。另一方面，如果你熟悉JSF，就会看到Seam基本上监听JSF servlet的处理阶段，并用一种更强大的变形给值和动作绑定替换变量解析器。

通过这个微不足道的应用程序，我们几乎无法触及Seam的九牛一毛。下面讨论Seam的一些更值得关注和更高级的特性，它们使得通过数据库后端创建复杂的Web应用程序变得很容易。

17.3 理解上下文组件

在前几节中，你已经把基础的JSF和EJB 3.0 Web应用程序变成了一个有状态的、对话的Seam应用程序。这么做减少了代码，并改善了应用程序的功能。你不应该到此为止——Seam还提供了更多的功能。

可以以上下文的方式把Seam组件组装在一起。这是一个强大的概念，可以对你如何设计有状态的应用程序有着深刻的影响。依据我们的经验，这就是Seam应用程序具有很少的几行简洁代码的主要原因之一。为了进行示范，我们来讨论如何创建新的应用程序功能。

几乎所有的Web应用程序都有一个登录/注销特性，以及登录用户的概念。假设应用程序的第一个页面（将强制为登录屏幕）一出现，用户必须立即登录到CaveatEmptor。支持它的登录屏幕和应用程序逻辑，是学习Seam组件如何在上下文中联结在一起的绝佳场景。

17.3.1 编写登录页面

用户看到如图17-8所示的登录屏幕。

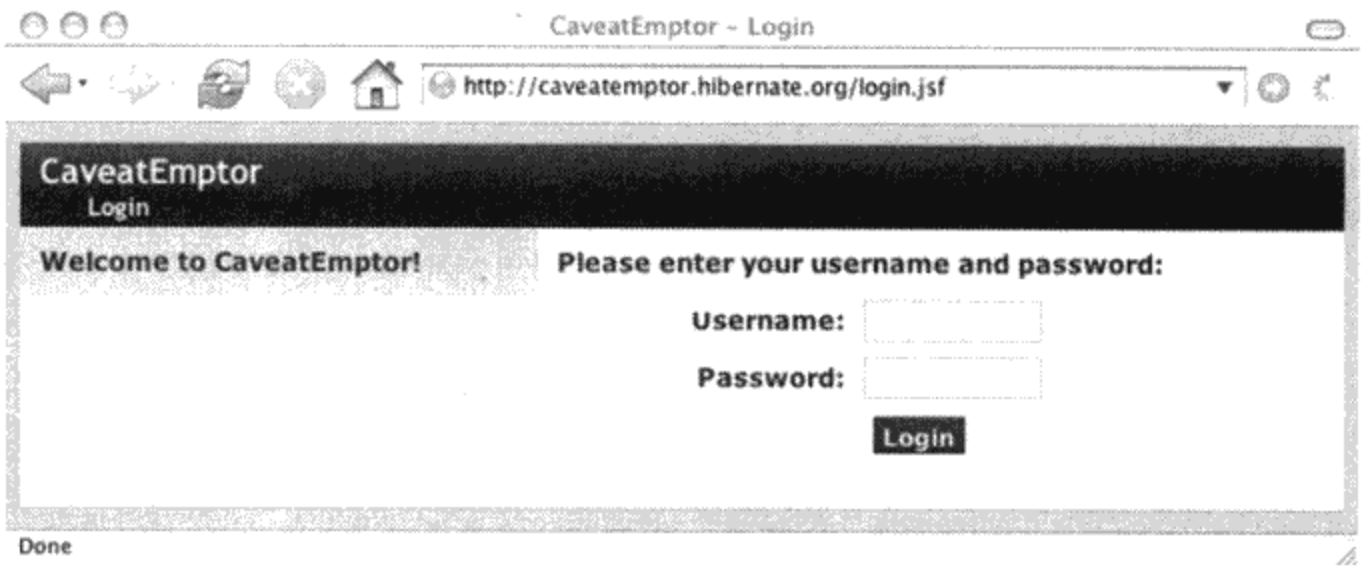


图17-8　CaveatEmptor的登录屏幕

这是一个称作login.xhtml的JSF页面，用Facelets编写的（代码清单17-12）。

代码清单17-12　login.xhtml页面源代码

```
...
<ui:composition xmlns="http://www.w3.org/1999/xhtml"
                xmlns:ui="http://java.sun.com/jsf/facelets"
                xmlns:h="http://java.sun.com/jsf/html"
                xmlns:f="http://java.sun.com/jsf/core"
                template="template.xhtml">

<ui:define name="screen">Login</ui:define>

<ui:define name="sidebar">

    <h1>Welcome to CaveatEmptor!</h1>

</ui:define>

<ui:define name="content">
<div class="section">

    <h1>Please enter your username and password:</h1>

    <h:form>

    <span class="errors"><h:messages/></span>

    <div class="entry">
        <div class="label">Username:</div>
        <div class="input">
            <h:inputText required="true" size="10"
                         value="#{currentUser.username}">
                <f:validateLength minimum="3" maximum="255"/>
            </h:inputText>
        </div>
    </div>
    <div class="entry">
        <div class="label">Password:</div>
        <div class="input">
            <h:inputSecret required="true" size="10"
```

```
                                value="#{currentUser.password}"/>
            </div>
        </div>

        <div class="entry">
            <div class="label"> </div>
            <div class="input">
                <h:commandButton value="Login" styleClass="button"
                                 action="#{login.doLogin}"/>
            </div>
        </div>

        </h:form>

    </div>
    </ui:define>

</ui:composition>
```

在这段源代码中，你可以看到更多一点的Facelets：一个全局的页面模板是如何呈现的（在<ui:composition>标签中），以及这个模板（screen、sidebar、content）支持的代码片段是如何定义的。

页面的内容是常规的JSF表单，包含到具名组件的值绑定和动作绑定。登录表单的输入字段被绑定到currentUser的属性，且Login按钮被绑定到login组件的doLogin()方法。

当登录页面第一次呈现时，JSF试图解析值和动作绑定。它用Seam变量解析器查找被引用的对象。Seam变量解析器无法在任何Seam上下文中找到对象，因此创建currentUser和login的实例。来看一下这些组件的源程序。

17.3.2 创建组件

第一个组件是currentUser。这是在CaveatEmptor中已经具有的类：User实体类。可以把它转变成一个可以由Seam通过注解进行处理的组件：

```
package auction.model;

import ...

@Name("user")
@Role(name = "currentUser", scope = ScopeType.SESSION)

@Entity
public class User implements Serializable {

    @Id @GeneratedValue
    private Long id = null;

    private String firstname;
    private String lastname;
    private String username;
    private String password;

    ...

    public User() {}

}
```

第一个Seam注解@Name，将这个POJO转变成一个Seam组件。现在每当Seam查找一个名为user的组件，并且Seam上下文都未保存具有该名称的变量时，就会由Seam创建User的一个新的空实例，并将它放到以变量名称user为名的事件上下文中。事件上下文是实体Seam组件的默认上下文（有状态和无状态的bean有着不同的默认上下文）。

对于登录功能来说，在事件上下文中并不需要User实例，因此你给这个组件定义一个额外的角色。每当Seam查找名为currentUser的组件，并且Seam上下文没有包含这个名称的变量时，Seam就实例化一个User，并将它放到会话上下文中（HTTP会话）。可以在代码和元数据中的任何位置轻松地指向currentUser，并从Seam取回一个对当前User对象的引用。

当login.xhtml呈现时，Seam创建了一个新的User对象，并将它绑定到会话上下文中。这个用户还没有登录，你需要输入用户名和密码，并单击Login按钮。

这么做就在另一个请求中执行了login组件中的doLogin()方法。这个组件实现为无状态的会话bean，如代码清单17-13所示。

代码清单17-13　无状态的Seam组件实现登录和注销过程

```
package auction.beans;

import ...

@Name("login")

@Stateless
public class LoginBean implements Login {

@In @Out
private User currentUser;

@PersistenceContext
EntityManager em;

@In
private Context sessionContext;

public String doLogin() {
  User validatedUser = null;

  Query loginQuery = em.createQuery(
                          "select u from User u where" +
                          " u.username = :uname" +
                          " and u.password = :pword"
                        );
  loginQuery.setParameter("uname", currentUser.getUsername());
  loginQuery.setParameter("pword", currentUser.getPassword());

  List result = loginQuery.getResultList();
  if (result.size() == 1) validatedUser = (User) result.get(0);

  if (validatedUser == null) {
      FacesMessages.instance().add(
        new FacesMessage("Invalid username or password!")
      );
      return null;
  } else {
```

```
            currentUser = validatedUser;

            sessionContext.set(LoggedIn.LOGIN_TOKEN, true);
            // or:
            Contexts.getSessionContext()
                    .set(LoggedIn.LOGIN_TOKEN, true);

            return "start";
        }
    }

    public String doLogout() {
        Seam.invalidateSession();
        return "login";
    }

}
```

这段代码有两个新的Seam注解：@In和@Out。这些都是变量别名，提示你用于组件联结。让我们在关注这些标签之前，先讨论剩下的代码。

doLogin()方法接受成员变量currentUser的用户名和密码，并试图在数据库中找到这个用户。如果找不到任何用户，就会等待JSF错误消息，并且结果null导致重新显示登录页面。如果找到了用户，就被分配到成员变量currentUser，替换该变量的旧值（目前为止，还没有改变HTTP会话中的任何东西）。

你还把一个令牌（一个简单的布尔）放进了会话上下文中，表明当前的User登录了。以后当需要测试当前的用户是否登录时，这个令牌很有用。稍后还将讨论doLogout()方法，它使当前的HTTP会话无效。

来看一下@In和@Out都有什么作用。

17.3.3 给上下文变量起别名

给上下文变量起别名听起来很复杂。然而，这是当你在Seam组件的代码中使用@In和@out时，对于正在进行的情况的正规描述。看一下图17-9，它显示了当用户单击Login之后，在调用应用程序请求处理阶段中所发生的事。

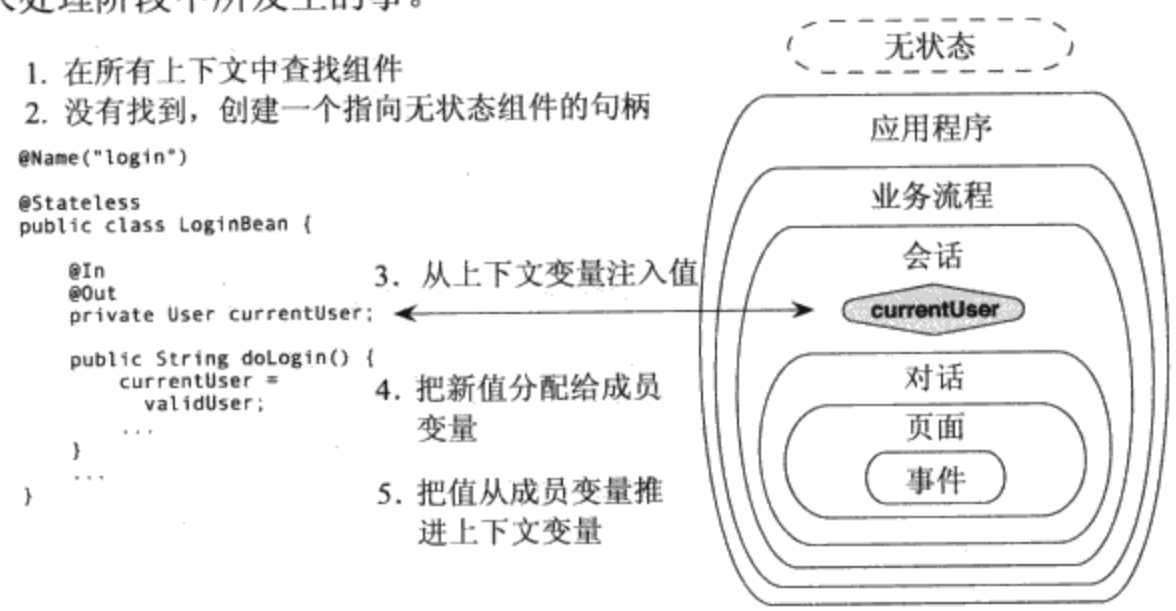


图17-9 Seam使一个成员别名与上下文变量同步

@In注解告诉Seam，你想要让一个值分配到这个组件的一个成员变量。这个值由Seam在该组件的方法被调用之前进行分配（Seam拦截每一个调用）。

这个值从哪里来的呢？Seam读取成员变量的名称，如前一个例子中的字段名称currentUser，并在它所有的上下文中查找同名的上下文变量。当doLogin()被调用时，Seam在会话上下文中找到currentUser变量。它采用这个值，并将它分配给组件的成员变量。这是对会话上下文中相同User实例的一个引用，你在组件的范围中创建了一个别名（alias）。然后可以在currentUser成员变量如getUsername()和getPassword()中调用方法。

@Out注解告诉Seam，你想要当组件的（任何）方法返回时，将一个值分配给上下文变量。上下文变量的名称是currentUser，与字段的名称相同。变量的上下文是currentUser Seam组件的默认上下文（在前一个例子中是会话上下文）。（还记得你在User类中分配的任务吗？）Seam采用成员变量的值，并将它放进上下文变量中。

再一次读取doLogin()方法。

在方法执行之前，Seam将上下文变量currentUser的值（在会话上下文中找到的）注入到同名的成员变量中去。然后方法执行并使用成员变量。成功登录之后（数据库查询），成员变量的值被替换了。这个新值必须被推回到上下文变量中去。方法执行之后，Seam将成员变量currentUser的值推进给这个组件定义的默认上下文（会话）。

不用字段，也可以用获取方法和设置方法对来起别名。例如，@In可以在setCurrentUser()中，@Out在getCurrentUser()中。在这两种情况下，被定义别名上下文变量的名称将是currentUser。

@In和@Out注解极为强大。本章后面会介绍更多的示例，但是我们需要更多的篇幅来描述你可以利用这些注解去做的所有事情。也请阅读Seam参考文档中的教程。

也可以直接利用上下文变量，而不用给它们起别名使之成为成员变量。在代码清单17-13中，doLogin()方法直接调用Contexts来设置变量值。

最后，Seam上下文构成了一个层次结构（除了伪上下文无状态之外），每当需要查找一个上下文变量时（以及当你没有显式地声明应该搜索哪个上下文时），就从该层次结构的最窄范围搜索到最宽范围。Seam参考文档中有一个上下文清单，在第2章中有它们的范围，这里不再重复。

让我们来完成登录/注销特性，并添加配置和代码中漏掉的部分。

17.3.4　完成登录/注销特性

登录/注销特性的导航规则漏掉了。这些是faces-config.xml中用于JSF的规则：

```
<navigation-rule>

    <navigation-case>
        <from-outcome>login</from-outcome>
        <to-view-id>/login.xhtml</to-view-id>
        <redirect/>
    </navigation-case>

    <navigation-case>
```

```
        <from-outcome>start</from-outcome>
        <to-view-id>/catalog.xhtml</to-view-id>
    </navigation-case>

</navigation-rule>
```

当用户成功登录时，动作的start结果把浏览器带到了应用程序的起始页面，这在CaveatEmptor中是拍卖货品的目录。

当用户注销时（通过单击一个被绑定到login.doLogout()方法的按钮），返回login结果，并且/login.xhtml得以呈现。你在这里定义的规则也等于说，这已经通过浏览器转向完成了。这个方法有两个结果：第一，用户看到/login.jsf作为浏览器的URL；第二，执行完doLogout()之后，在调用应用程序阶段之后重定向立即完成。需要用这个重定向在接下来的呈现响应阶段中启动一个新的HTTP会话。旧的HTTP会话被doLogout()标识为无效，并且在调用应用程序阶段之后被丢弃。

这个应用程序并非真正安全。虽然用户在打开应用程序时最终停止在登录页面上，但是他们可以将其他的页面做成书签（就像拍卖货品目录一样），并直接跳到一个URL。如果没有出现登录令牌，你需要保护页面，并将用户重定向到登录页面。

你还需要直接保护组件bean方法，防止用户找到一种不先呈现页面而执行动作的方法。（利用通过JavaScript公开的Seam组件，这是可能的。）用EJB 3.0拦截器保护组件方法（见代码清单17-14）。

代码清单17-14 检查登录令牌的EJB 3.0拦截器

```
package auction.interceptors;

import ...

@Name("loginInterceptor")
@Interceptor(around={BijectionInterceptor.class,
                     ValidationInterceptor.class,
                     ConversationInterceptor.class,
                     BusinessProcessInterceptor.class},
             within= RemoveInterceptor.class)
public class LoggedInInterceptor {

    @AroundInvoke
    public Object checkLoggedIn(InvocationContext invocation)
                    throws Exception {

        String loggedInOutcome = checkLoggedIn();

        if (loggedInOutcome == null) {
            return invocation.proceed();
        } else {
            return loggedInOutcome;
        }
    }

    public String checkLoggedIn() {
        boolean isLoggedIn =
```

```
                Contexts.getSessionContext()
                        .get(LoggedIn.LOGIN_TOKEN) != null;
        if (isLoggedIn) {
            return null;
        } else {
            return "login";
        }

    }
}
```

这个拦截器有两个用处。第一，它是一个EJB 3.0 `@Interceptor`，在其他EJB 3.0拦截器的中间执行。这些其他的拦截器全部来自Seam，你需要把自己的拦截器放在堆栈的正确位置上。EJB 3.0注解`@AroundInvoke`标识受保护的组件上的任何方法在调用之前和之后而被调用的方法。如果`checkLoggedIn()`方法没有返回任何东西（结果为空），被拦截组件的调用就可以继续。如果结果不为空，这个结果就被传递到JSF导航处理器，并且被拦截的组件调用不能继续进行。

拦截器类也是Seam简单的Java组件（Seam组件不一定要是EJB），名为`loginInterceptor`。JavaBean组件默认的上下文是event（事件）。你现在可以在表达式中使用这个组件名称——例如，用表达式`#{loginInterceptor.checkLoggedIn}`——不通过EJB拦截。这用来保护页面不被直接访问很有用。在Seam中，可以定义在页面呈现之前运行的动作。这些声明在WEB-INF/pages.xml中：

```
<pages>
    <page view-id="/catalog.xhtml"
          action="#{loginInterceptor.checkLoggedIn}"/>

</pages>
```

当用户直接命中/catalog.jsf URL时，`loginInterceptor.checkLoggedIn()`动作运行。如果这个动作有一个非空的结果，Seam就把结果作为常规的JSF结果进行处理，并应用导航规则。

最后，通过将拦截器应用到一个EJB类，来保护组件方法。这可以用XML（META-INF/ejb-jar.xml）完成，如果想要使用通配符并保护特定包中的所有bean的话，这个很棒。或者可以编写一个封装拦截器的辅助注解：

```
package auction.interceptors;

import ...

@Target(TYPE)
@Retention(RUNTIME)
@Documented
@Interceptors(LoggedInInterceptor.class)
public @interface LoggedIn {
    public static final String LOGIN_TOKEN  = "loggedIn";
}
```

这个注解也包含所有其他代码所引用的字符串常量，这很方便。现在，在EJB类中应用这个注解：

```
package auction.beans;

import ...

@Name("catalog")

@LoggedIn

@Stateful
public class CatalogBean implements Catalog { ... }
```

每当这个EJB的任何方法被调用时，LoggedInInterceptor就运行，并验证用户登录。如果用户没有登录，拦截器就把login结果返回给JSF。

也可以在页面上检查登录令牌——例如，如果你必须决定Logout按钮是否应该呈现：

```
<h:form>
    <h:panelGroup rendered="#{loggedIn}">
        Current user: <b>#{currentUser.username}</b>
        (<h:commandLink value="Logout" action="#{login.doLogout}"/>)
    </h:panelGroup>
</h:form>
```

表达式#{loggedIn}解析到布尔上下文变量loggedIn，它要么出现在会话上下文中，要么就不出现。

应用程序的登录/注销功能现在完成了。页面和组件方法受到了保护，只有一个登录用户可以打开和调用它们。

如何获得用户账号呢？必须填写一张申请表。必须验证表单数据，并在数据库中创建账号。

17.4 验证用户输入

在前一个示例（登录屏幕和登录/注销代码）中，你依赖标准的JSF验证器和doLogin()方法中自己的代码来验证用户输入。当用户提交登录表单时，JSF就在处理验证阶段中运行被声明的验证器(在login.xhtml中)。如果用户输入用户名和密码，且验证成功，就执行login.doLogin()方法。给定的用户名和密码被绑定到数据库查询。用户输入被验证两次：

- JSF在它将每个输入字段的值与绑定模型(Seam会话上下文中的currentUser)同步之前，验证HTML表单输入。如果稍后在动作方法中访问currentUser，就会保证页面的验证规则已经得到检查。
- JDBC驱动程序在你将用户名和密码绑定到JPA QL查询时验证用户输入。这是到常规的JDBC PreparedStatement的一个内部绑定，因此JDBC驱动程序转义用户可能已经输入的任何危险字符。

在表现层中验证用户输入，并确保不可能有任何SQL注入攻击，这对于一个简单的登录屏幕来说够好了。但是如果需要在User对象保存在数据库中之前（例如，在账号注册过程期间）对它进行验证，这时该怎么办呢？

你需要更复杂的验证：必须检查所输入的用户名的长度，看看是否使用了任何非法字符，还要验证密码的质量。所有这些都可以用更多和可能定制的验证在JSF表现层中得以解决，但是数据库Schema也必须验证所保存数据的完整性。例如，创建数据库约束，限制保存在USERNAME列

中的值的长度，或者要求一个成功的字符串模式匹配。

在任何复杂的应用程序中，输入验证都是不仅要在表现层还要在几个层甚至不同的层中进行处理的一个关注点。它是可以影响你所有代码的横切关注点。利用Hibernate Validator，可以轻松地为所有的应用层隔离和封装验证和数据完整性规则。

17.4.1　Hibernate Validator 简介

Hibernate Validator是Hibernate Annotations的一个模块。甚至可以不通过Hibernate和Seam，而只通过类路径中的hibernate3.jar和hibernate-annotations.jar，在任何Java应用程序中使用Hibernate Validator。（未来，Hibernate Validator可能分支到它自己的独立模块。这取决于在JSR 303“Bean Validation”中所做的工作，请见http://jcp.org/en/jsr/detail?id=303。）

Hibernate Validator是一组注解，可以将它应用到领域模型来声明性地定义数据验证和完整性规则。可以通过编写自己的注解，用自己的约束来扩展Hibernate Validator。

被应用的这些完整性规则和验证规则可以用于以下对象：

- 简单的Java——可以在Java代码中的任何位置调用`ClassValidator` API，并提供需要检查的对象。验证器完成验证，或者返回`InvalidValue`对象的一个数组。每个`InvalidValue`都包含关于验证失败的详细信息，例如属性名称和错误消息。
- Hibernate——在原生的Hibernate中，可以注册钩入Hibernate持久化操作内部处理的Hibernate Validator事件。通过这些事件，Hibernate可以验证你正在数据库中自动且透明地插入或者更新的任何对象。包含详细信息的`InvalidStateException`在验证失败时被抛出。
- Hibernate EntityManager——如果通过Hibernate EntityManager使用JPA，Hibernate Validator事件就被默认为激活状态，并且当你在数据库中插入或者更新对象时，所有的实体实例都要根据验证注解进行检查。
- SchemaExport——Hibernate的数据库模式生成特性可以创建数据库约束，它影响着SQL DDL中的完整性规则。SchemaExport（hbm2ddl）工具在领域模型中读取验证注解，并在SQL DDL中呈现它们。每个注解都知道SQL应该是什么样子（或者如果SQL中不存在相当的约束）。如果基于定制的过程SQL约束（触发器等）编写自己的验证注解，这个就特别强大。可以将一个定制的数据库完整性规则封装在单个Java注解中，并用Hibernate Validator在运行时检查被注解类的实例。
- Seam——通过Seam，可以将Hibernate Validator与应用程序的表现层和应用程序逻辑整合起来。Seam可以在JSF表单提交时自动调用验证API，并用任何验证错误消息装饰表单。

你已经在前几节中使用过Seam和Hibernate EntityManager。一旦将Hibernate Validator注解添加到实体类，这些完整性规则立即就在持久化上下文被清除到数据库时得到验证。

让我们将Hibernate Validator连到JSF用户界面，并给CaveatEmptor实现一项账号注册特性。

17.4.2　创建注册页面

我们将从用户界面开始。你需要一张新的页面register.xhtml，包含一个JSF表单。为了到达该

页面，必须在login.xhtml页面中提供一个链接，以便用户们知道他们可以注册：

```
<ui:define name="sidebar">

    <h1>Welcome to CaveatEmptor!</h1>

    <div>
        <h:form>
            If you don't have an account, please
            <h:commandLink action="register" immediate="true">
                register...
            </h:commandLink>
        </h:form>
    </div>

</ui:define>
```

提交这个表单，立即跳到请求过程中的呈现响应阶段（无需验证、模型绑定或者动作执行）。register字符串是一个简单的导航结果，在faces-config.xml中的导航规则中定义：

```
<navigation-rule>
    <navigation-case>
        <from-outcome>login</from-outcome>
        <to-view-id>/login.xhtml</to-view-id>
        <redirect/>
    </navigation-case>

    <navigation-case>
        <from-outcome>register</from-outcome>
        <to-view-id>/register.xhtml</to-view-id>
        <redirect/> <!-- Make this bookmark-able -->
    </navigation-case>
    ...
</navigation-rule>
```

看一下图17-10中注册页面的屏幕快照。

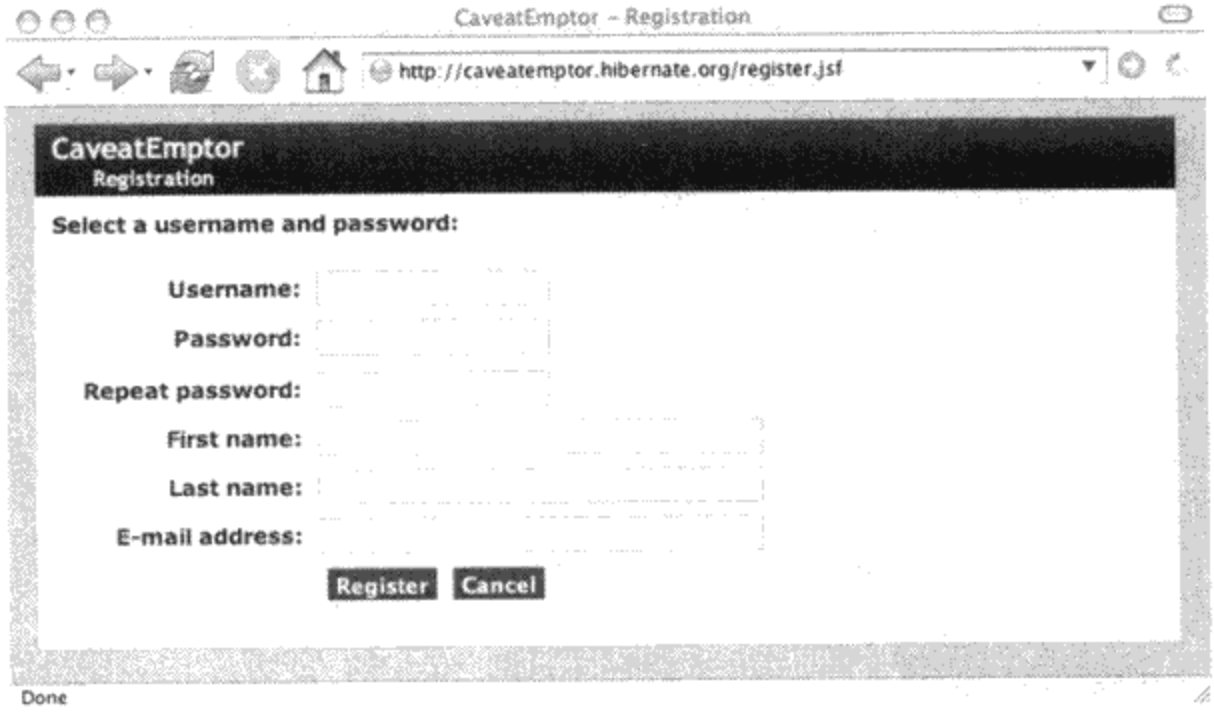


图17-10 register.xhtml页面

register.xhtml中JSF表单的代码给JSF使用了一些可视的Seam组件（可以在jboss-seam-ui.jar文件中找到这些）。

1. 用Seam标签装饰页面

现在用Seam组件把页面与Hibernate Validator整合起来（见代码清单17-15）。我们已经省略了页面的基础HTML；唯一值得关注的部分是表单，以及该表单的验证如何生效。还需要给Seam标签库（taglib）声明命名空间，来使用Facelets模板中的组件；在接下来的所有示例中使用的前缀都是s。

代码清单17-15　包含验证的registration.xhtml源代码

```
<ui:composition ...
                xmlns:s="http://jboss.com/products/seam/taglib"
                ...>
<h:form>
    <f:facet name="beforeInvalidField">                    ❶
        <h:graphicImage value="/img/attention.gif"
                        width="18" height="18"
                        styleClass="attentionImage"/>
    </f:facet>
    <f:facet name="afterInvalidField">                     ❷
        <s:message/>
    </f:facet>

    <div class="errors" align="center">                    ❸
        <h:messages globalOnly="true"/>
    </div>

<s:validateAll>                                            ❹

    <div class="entry">
        <div class="label">Username:</div>                 ❺
        <div class="input"><s:decorate>                    ❻
            <h:inputText size="16" required="true"
                         value="#{currentUser.username}"/>
        </s:decorate></div>
    </div>

    <div class="entry">
        <div class="label">Password:</div>
        <div class="input"><s:decorate>
            <h:inputSecret size="16" required="true"
                           value="#{currentUser.password}"/>
        </s:decorate></div>
    </div>

    <div class="entry">
        <div class="label">Repeat password:</div>
        <div class="input"><s:decorate>
            <h:inputSecret size="16" required="true"
                           value="#{register.verifyPassword}"/>
        </s:decorate></div>
    </div>

    <div class="entry">
```

```
            <div class="label">Firstname:</div>
            <div class="input"><s:decorate>
                <h:inputText size="32" required="true"
                             value="#{currentUser.firstname}"/>
            </s:decorate></div>
        </div>
        ...
    </s:validateAll>

    <div class="entry">
        <div class="label"> </div>

        <div class="input">
            <h:commandButton value="Register" styleClass="button"
                             action="#{register.doRegister}"/>

            <h:commandButton value="Cancel" styleClass="button"
                             action="login" immediate="true"/>
        </div>
    </div>

</h:form>
```

❶ 这个组件facet被Seam装饰器（decorator）用于错误显示。你会在具有无效值的任何输入字段之前看见它。

❷ Seam装饰器把错误消息放在无效的字段之后。

❸ 没有被分配到任何字段的全局错误消息显示在表单的顶部。

❹ <s:validateAll> Seam标签给所有的子标签启用Hibernate Validator——也就是封装在这个表单中的所有输入字段。也可以只给单个字段启用Hibernate Validator，通过用<s:validate/>将输入字段包起来。

❺ <s:decorate> Seam标签处理验证错误消息。如果出现错误，它就把输入字段包在beforeInvalidField和afterInvalidField facet里面。

❻ JSF输入小部件的可见大小为16个字符。注意，JSF并不限制用户可以输入的字符串大小，但它要求用户输入一个值。这个“非空”验证仍然是JSF的工作，而不是Hibernate Validator的工作。

❼ Register按钮有一个动作绑定到register.doRegister（一个Seam组件）。

❽ 需要Cancel按钮，把用户调整到登录页面。再次用immediate="true"跳过表单的处理。

当注册表单被提交时，Seam就参与到了JSF 处理验证阶段中，并为每个实体对象（你绑定了输入字段到它上面）调用Hibernate Validator。在这个例子中，只有单个实体实例currentUser必须被验证，Seam在它的上下文中查找它。

如果处理验证阶段完成，register.doRegister就在调用应用程序中执行。这是处在事件上下文中的一个有状态会话bean。

2. 注册Seam组件

注册表单有两个绑定到register Seam组件。第一个绑定是值绑定，通过register.verify-

Password。JSF和Seam现在把来自这个字段的用户输入与register.setVerifyPassword()和register.getVerifyPassword()方法同步。

第二个绑定是Register按钮到register.doRegister()方法的动作绑定。这个方法必须在JSF和Hibernate Validator输入验证之后，并在currentUser可以作为新账号保存在数据库中之前，实现额外的检查。请见代码清单17-16中的代码。

代码清单17-16　有状态会话bean实现注册逻辑

```
package auction.beans;

import ...

@Name("register")                                    ❶
@Scope(ScopeType.EVENT)

@Stateful
public class RegisterBean implements Register {                    ❷

    @In
    private User currentUser;

    @PersistenceContext
    private EntityManager em;

    @In(create=true)                                 ❸
    private transient FacesMessages facesMessages;

    private String verifyPassword;                                 ❹
public String getVerifyPassword() {
    return verifyPassword;
}
public void setVerifyPassword(String verifyPassword) {
    this.verifyPassword = verifyPassword;
}

public String doRegister() {                         ❺

    if (!currentUser.getPassword().equals(verifyPassword)) {       ❻
        facesMessages.add("Passwords didn't match!")
        verifyPassword = null;
        return null;
    }

    List existing =                                  ❼
        em.createQuery("select u.username from User u" +
                       " where u.username = :uname")
          .setParameter("uname", currentUser.getUsername())
          .getResultList();

    if (existing.size() != 0) {
        facesMessages.add("User exists!");
        return null;
    } else {
        em.persist(currentUser);                     ❽
        facesMessages.add("Registration complete.");
        return "login";
```

```
        }
    }

    @Remove @Destroy                    ❾
    public void destroy() {}

}
```

❶ register Seam组件由Seam创建，并且当事件上下文被销毁时它也被销毁，它是单个JSF请求的范围。

❷ Seam注入currentUser，别名取自会话上下文中的上下文变量。

❸ Seam注入（或者创建，如果无法在任何上下文中找到变量的话）FacesMessages的一个实例。如果需要发送消息给一个JSF页面，这就是一个很方便的辅助方法；你在之前没有通过注入但是通过手工查找使用过它。

❹ 这个组件的verifyPassword字段与JSF表单同步。

❺ 这个方法给新账号的注册实现了主要逻辑。它在Hibernate Validator检查完currentUser之后调用。

❻ 用户输入的两个密码必须相符，否则表单上就会显示一条错误消息。null结果利用错误消息触发登录表单的重新显示。

❼ 用户名在数据库中是唯一的。这个多行约束不能通过Hibernate Validator在内存中进行检查。需要执行数据库查询，并验证用户名。

❽ 如果所有的验证都通过，就persist()该currentUser对象，清除持久化上下文，并且当doRegister()方法返回时提交事务。结果login把用户调整回到登录页面，在这里“注册完成”的消息呈现在登录表单上。

❾ 当事件上下文被销毁的时候，Seam在JSF请求结束时调用组件的destroy()方法。EJB 3.0容器移除有状态会话bean，因为该方法用@Remove作了标识。

用户输入验证经常比在单个对象中核查单个值更为复杂。Seam给注册表单中的所有被绑定的实体实例调用Hibernate Validator。然而，所输入用户名的重复检查需要数据库访问。可以出于这个目的编写自己的Hibernate Validator扩展，但是当User对象必须进行验证时，总是在数据库中检查重复的用户名似乎是不可能的。另一方面，业务逻辑一般通过过程代码实现，而不是完全声明性地实现。

到目前为止，Hibernate Validator还没有做任何事情。如果你提交注册表单而没有输入任何值，就只有required="true"的内建JSF验证器运行。你在每个声明需要值的输入字段中得到一个内建的JSF错误消息。

3. 注解实体类

Hibernate Validator不是活动的，因为User实体类中没有完整性规则，因此所有的对象都通过验证测试。可以在实体类的字段或者获取方法中添加验证注解：

```
package auction.model;

import ...

@Name("user")
@Role(name = "currentUser", scope = ScopeType.SESSION)

@Entity
@Table(name = "USERS")
public class User implements Serializable {

    @Id @GeneratedValue
    @Column(name = "USER_ID")
    private Long id = null;

    @Column(name = "USERNAME", nullable = false, unique = true)
    @org.hibernate.validator.Length(
        min = 3, max = 16,
        message = "Minimum {min}, maximum {max} characters."
    )
    @org.hibernate.validator.Pattern(
        regex="^\\w*$",
        message = "Invalid username!"
    )
    private String username;
    @Column(name = "`PASSWORD`", length = 12, nullable = false)
    private String password;

    @Column(name = "FIRSTNAME", length = 255, nullable = false)
    private String firstname;

    @Column(name = "LASTNAME", length = 255, nullable = false)
    private String lastname;

...}
```

你只应用两个Hibernate Validator注解：@Length和@Pattern验证器。这些验证器有着如最大长度和最小长度这样的属性，或者正则表达式模式（请见java.util.regex.Pattern）。所有内建的验证注解的清单，请见Hibernate Annotations包中Hibernate Validator参考文档。也可以轻松地编写自己的注解。

所有的验证注解都有message属性。如果出现验证失败，这个消息就与表单字段显示在一起。

可以添加更多的验证注解，它们还核查密码、User的名和姓。注意，USERNAME @Column注解的length属性已经被移除了。由于长度验证注解，Hibernate的Schema导出工具现在知道VARCHAR(16)必须在数据库Schema中创建。另一方面，nullable=false属性还在，用于NOT NULL（非空）数据库列约束的生成。（可以使用一个来自Hibernate Validator的@NotNull验证注解，但是JSF已经替你检查了该字段：表单字段为required="true"。）

在将验证注解添加到User之后，提交包含不完整的值的注册表单时会显示错误消息，如图17-11所示。

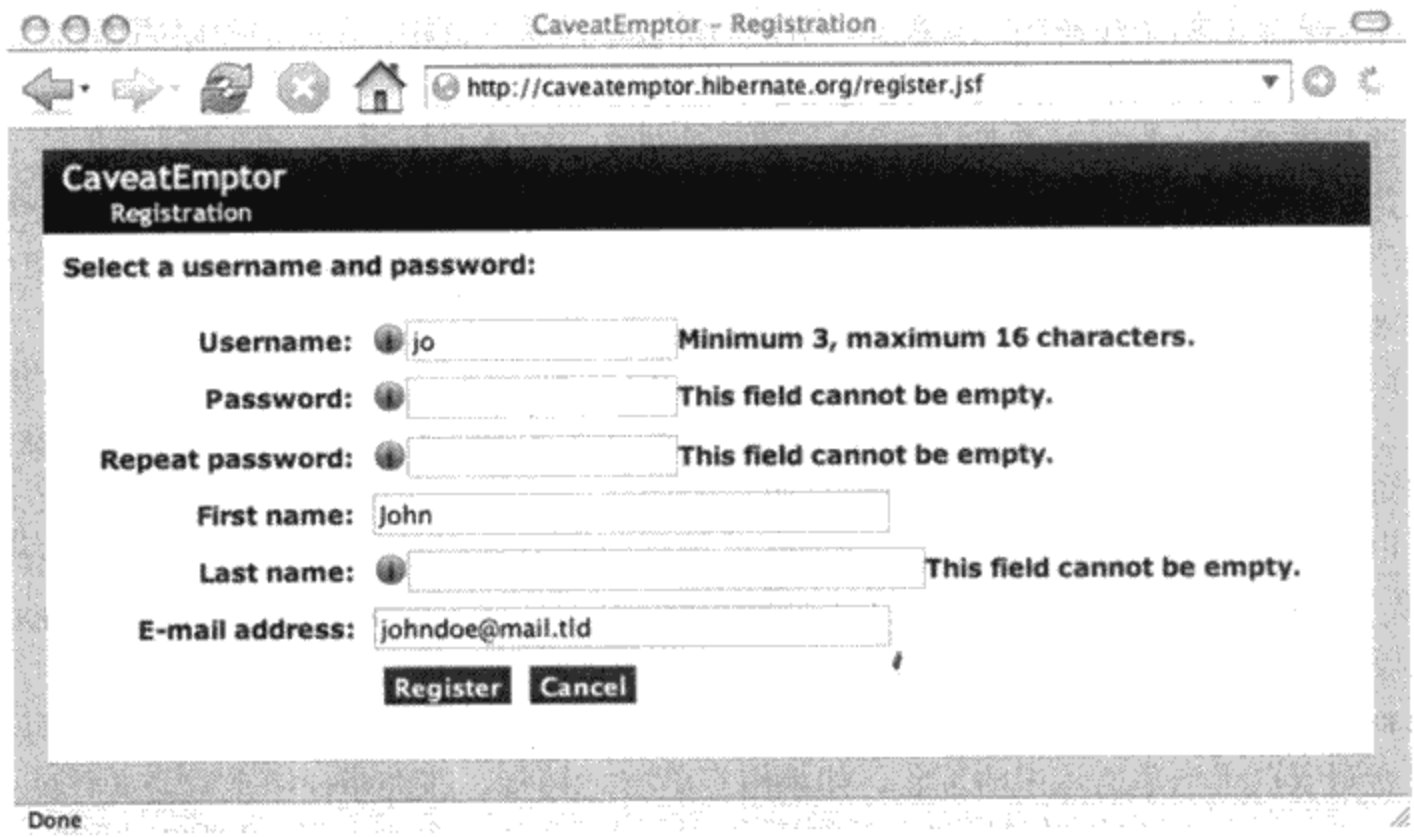


图17-11 Seam用验证错误消息装饰输入字段

注册特性现在完成了，用户们可以创建新的账号。似乎还不完美的方面在于错误消息。如果你试试这段代码，就会发现错误消息并不像图17-11所示的那么好。相反，需要输入的字段中有一个很难看的_id23: Field input is required消息。而且，把英文的错误消息放到你的实体类中（即使它们是在注解元数据中）是一种好办法吗？

不是只替换默认的JSF错误消息（它包括自动生成的小部件标识符），而是把所有的用户界面消息都隔离，并且还允许用户切换语言。

17.4.3 用 Seam 实现国际化

朝着多国语言应用程序迈出的第一步是语言切换器——假设是用户可以在应用程序的最上层菜单中单击的一个链接。Seam有一个locateSelector组件（在会话上下文中），使得这一步变得容易起来：

```
<h:form>
    <h:panelGroup>
        <h:outputText value="#{messages['SelectLanguage']}"/>:
        <h:commandLink
            value="EN"
            action="#{localeSelector.selectLanguage('en')}"/>
        |
        <h:commandLink
            value="DE"
            action="#{localeSelector.selectLanguage('de')}"/>
    </h:panelGroup>
</h:form>
```

这个小表单有两个超链接：EN和DE。用户们可以单击这些链接，在应用程序的界面上进行英语版和德语版之间的切换。链接动作通过文字参数被绑定到localeSelector.selectLanguage()方法。这些参数（en和de）都是ISO语言代码，请见Javadoc中的java.util.Locale。

但是，这还不是这里所发生的全部。呈现表单时，#{messages['SelectLanguage']}表达式被取值，并且该表达式的输出在命令链接之前被呈现为文本。这个表达式的输出有点像“Select your language:”。它是哪里来的？

很清楚，messages是一个Seam组件，它在会话上下文中。它表示被外部化消息的映射；SelectLanguage是搜索这个映射的一个键。如果该映射包含了该键的一个值，这个值就被打印出来。否则，SelectLanguage就被逐字打印。

可以在你能够编写解析Seam组件的表达式的任何地方使用messages组件（几乎是任何地方）。这个组件是指向Java资源包的一个方便的句柄，这是一个复杂的术语，意味着.property文件中的键/值对。

Seam自动地从类路径的根目录下将messages.properties读取到messages组件中。然而，实际的文件名则取决于当前所选中的位置。如果用户单击DE链接，在类路径中搜索的文件就命名为messages_de.properties。如果英语是活动的语言（它是默认的，取决于JSF配置和浏览器），被加载的文件就是messages_en.properties。

下面是messages_en.properties的一个片段：

```
SelectLanguage = Select language:
PleaseRegisterHint = Create a new account...
SelectUsernameAndPassword = Select a username and password
PasswordVerify = Repeat password
PasswordVerifyField = Controlpassword
Firstname = First name
Lastname = Last name
Email = E-mail address
TooShortOrLongUsername = Minimum 3, maximum 16 characters.
NotValidUsername = Invalid name! {TooShortOrLongUsername}
PasswordVerifyFailed = Passwords didn't match, try again.
UserAlreadyExists = A user with this name already exists.
SuccessfulRegistration = Registration complete, please log in:
DoRegister = Register
Cancel = Cancel

# Override JSF defaults
javax.faces.component.UIInput.REQUIRED = This field cannot be empty.
```

最后一行给输入字段小部件覆盖默认的JSF验证错误消息。如果想要合并消息，语法{Key}就很有帮助：TooShortOrLongUsername消息被附加到NotValidUsername消息。

现在可以用在messages Seam组件中查找键的表达式，来替换XHTML文件中的所有字符串。也可以在Java代码中使用RegistrationBean组件中的资源包的键：

```
public String doRegister() {
    if (!currentUser.getPassword().equals(verifyPassword)) {
        facesMessages
              .addFromResourceBundle("PasswordVerifyFailed");
```

```
            verifyPassword = null;
            return null;
        }

        List existing =
            em.createQuery("select u.username from User u" +
                           " where u.username = :uname")
              .setParameter("uname", currentUser.getUsername())
              .getResultList();

        if (existing.size() != 0) {
            facesMessages
                  .addFromResourceBundle("UserAlreadyExists");
            return null;
        } else {
            em.persist(currentUser);
            facesMessages
                  .addFromResourceBundle("SuccessfulRegistration");
            return "login";
        }
    }
```

最后，可以在Hibernate Validator的消息中使用资源包的键（这不是一项Seam特性——它没有Seam也可以进行）：

```
@Entity
public class User implements Serializable {

...
    @Column(name = "USERNAME", nullable = false, unique = true)
    @org.hibernate.validator.Length(
        min = 3, max = 16,
        message = "{TooShortOrLongUsername}"
    )
    @org.hibernate.validator.Pattern(
        regex="^\\w*$",
        message = "{NotValidUsername}"
    )
    private String username;

...}
```

让我们翻译一下该资源包，并把它保存为message_de.properties：

```
SelectLanguage = Sprache:
PleaseRegisterHint = Neuen Account anlegen...
SelectUsernameAndPassword = Benutzername und Passwort w\u00e4hlen
PasswordVerify = Passwort (Wiederholung)
PasswordVerifyField = Kontrollpasswort
Firstname = Vorname
Lastname = Nachname
Email = E-mail Adresse
TooShortOrLongUsername = Minimum 3, maximal 16 Zeichen.
NotValidUsername = Ung\u00fcltiger name! {TooShortOrLongUsername}
PasswordVerifyFailed = Passworte nicht gleich, bitte wiederholen.
UserAlreadyExists = Ein Benutzer mit diesem Namen existiert bereits.
SuccessfulRegistration = Registrierung komplett, bitte einloggen:
```

```
DoRegister = Registrieren
Cancel = Abbrechen

# Override JSF defaults
javax.faces.component.UIInput.REQUIRED = Eingabe erforderlich.
```

注意，你用UTF序列来表达非ASCII的字符。如果用户在应用程序中选择德语，并试图在不完成表单的情况下进行注册，那么所有的消息都以德语显示（见图17-12）。

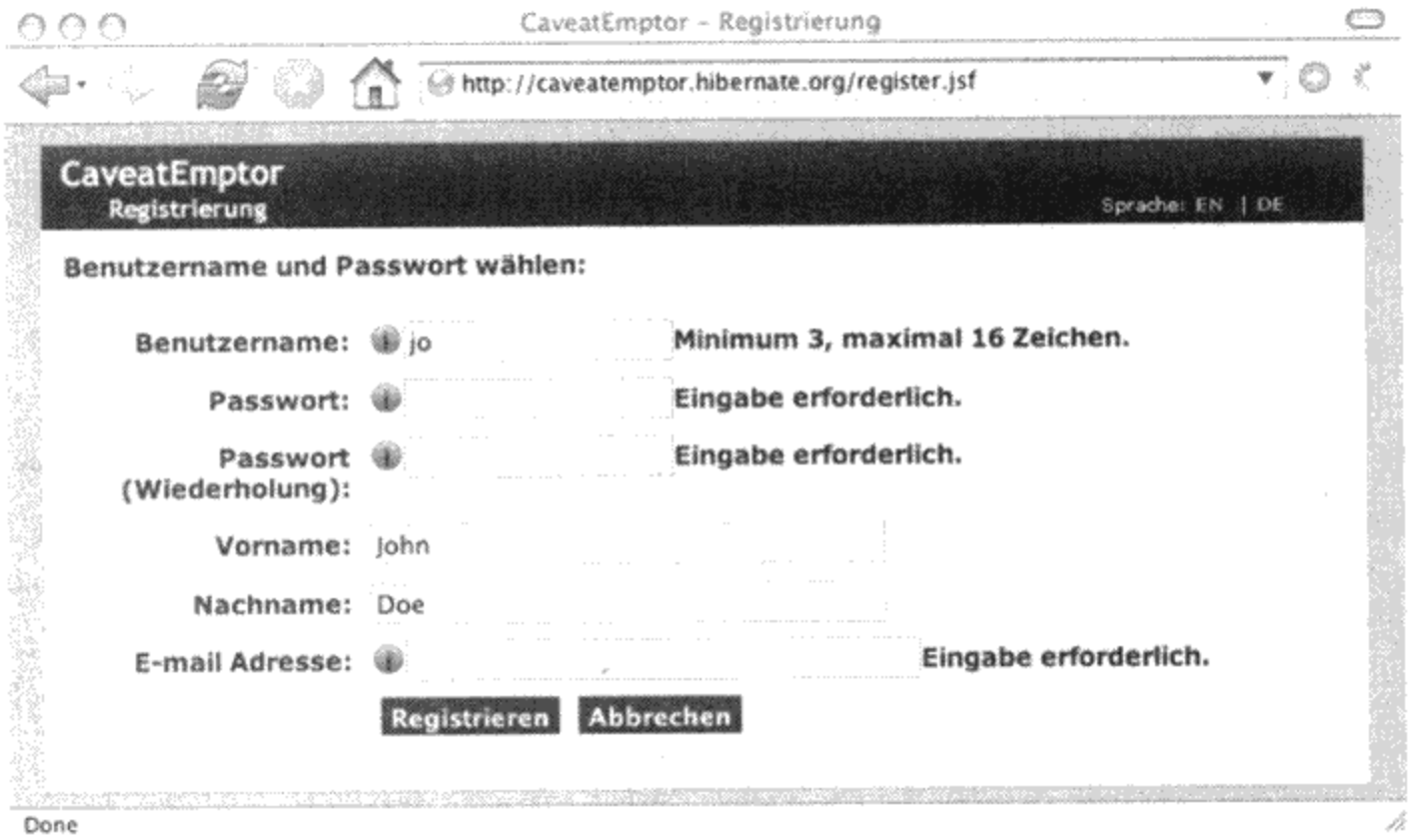


图17-12　用户界面已经被转化为德语

被选中的语言是一个会话范围的设置。它现在是活动的，直到用户注销（使HTTP会话无效）。如果你也在seam.properties中设置`localeSelector.cookieEnabled=true`开关，那么用户语言选择就会被作为一个cookie保存在Web浏览器中。

我们要示范的最后一项但并非最不重要的Seam特性是，通过Seam的自动持久化上下文处理。如果已经在Hibernate应用程序中见过`LazyInitializationException`（谁没见过？），这就是最完美的解决方案。

17.5　利用 Seam 简化持久化

本章前面的所有示例都使用`EntityManager`，由EJB 3.0容器注入。EJB中的成员字段利用`@PersistenceContext`进行注解，并且持久化上下文的范围始终是为特定的动作方法启动和提交的事务。对于Hibernate而言，Hibernate `Session`为在会话bean中调用的每一个方法而打开、清除和关闭。

当会话bean方法返回，并且持久化上下文关闭时，那么从位于那个bean方法的数据库中加载

的所有实体实例都处于脱管状态。可以在JSF页面中通过访问这些实例已初始化的属性和集合来呈现它们，但是当你试图访问一个未被初始化的关联或者集合时，就会得到LazyInitializationException。如果想要让它再次处于持久化状态，还必须重附（或者合并，通过JPA）脱管实例。此外，必须小心编写实体类的equals()和hashCode()方法代码，因为受保护的同一性范围只有事务，与（相对短的）持久化上下文范围一样。

本书前面已经几次讨论脱管对象状态的后果。通常我们断定，通过将持久化上下文和同一性范围扩展到超过事务来避免脱管状态，这是一种更为可取的解决方案。你已经见过OSIV模式，它将持久化上下文扩展为跨越整个请求。虽然这个模式是一种务实的解决方案，用于以无状态方式（其最重要的范围是请求）构建的应用程序，但是如果你编写一个包含对话的有状态Seam应用程序，就需要一种更加强大的变形。

如果让Seam将EntityManager注入到会话bean中去，并且如果你让Seam管理持久化上下文，将会得到下列结果：

- *被扩展的持久化上下文自动绑定和界定到对话*——你有一个跨越对话的受保护的同一性范围。一个特定的对话至少有一种特定数据库行的内存表示法。没有脱管对象，你可以轻松地通过双等号比较实体实例（a==b）。不必实现equals()和hashCode()，并通过业务键比较实体实例。
- *在一个对话中访问未被初始化的代理或者集合时，再也没有*LazyInitializationExceptions了——持久化上下文对于整个对话都是活动的，且持久化引擎始终可以按需抓取数据。Seam提供了一种更加强大且方便的OSIV模式的实现，它不仅在单个请求期间而且在整个对话期间，避免了游离对象。
- *自动把JSF请求包在几个系统事务中*——Seam使用几个事务来封装JSF请求生命周期中的几个阶段。稍后会讨论这个事务集合，它的其中一个好处在于，你有一个保持数据库锁定时间尽可能短的优化集合，而不用编写任何代码。

让我们用一个示例进行示范，通过把前一节的注册过程重新编写为包含被扩展的持久化上下文的对话。前一个实现基本上是无状态的：RegisterBean只被界定到单个事件。

17.5.1 实现对话

回来看一下代码清单17-16中所示的代码。这个有状态会话bean是用于CaveatEmptor中账户注册页面的backing bean。当用户打开或者提交注册页面时，就创建该bean的一个实例，并且在处理事件时处于活动状态。JSF将表单值绑定到bean中（通过verifyPassword和Seam注入的currentUser），并在必要时调用动作监听器方法。

这是一个无状态的设计。虽然你使用了一个有状态会话bean，但它的范围是单个请求（Seam中的事件上下文）。这种方法效果很好，因为用户经过的对话很是繁琐——只有包含单个表单的单张页面必须填写和提交。图17-13展现了一个更为复杂的注册过程。

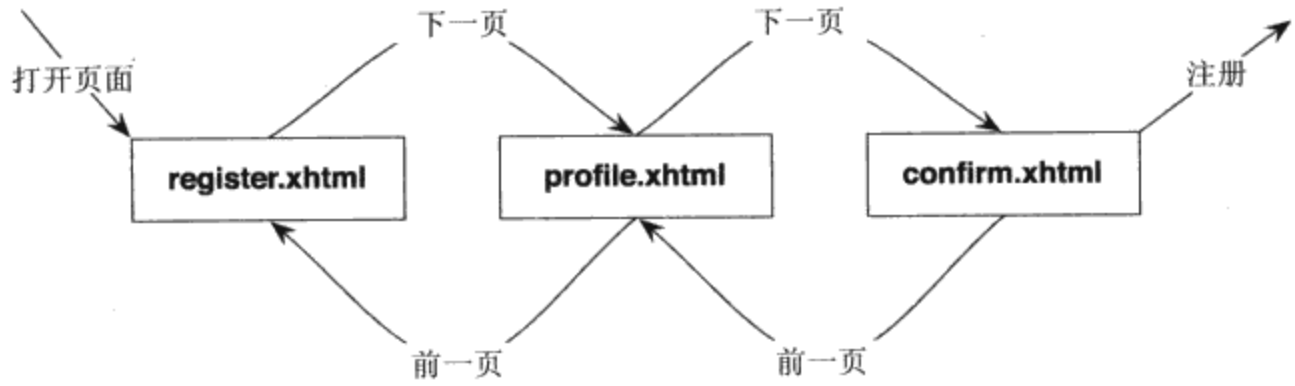


图17-13　CaveatEmptor注册向导

用户打开register.xhtml，并输入想要的用户名和密码。用户单击Next Page之后，第二个包含个性化数据（名字、电子邮件地址等）的表单出现，并且必须填写。最后一页再一次显示所有的账号和个性化数据，因此用户可以确认（或者退回并纠正）。

这个注册过程是一个向导风格的对话，包含一般的Next Page和Previous Page按钮，允许用户在对话中前进或者后退。许多应用程序都需要这种对话框。如果没有Seam，实现多页对话对于Web应用程序开发人员来说，仍然很困难。(注意，有许多其他好的会话用例，向导对话框是很常见的。)

给这个对话编写一些页面和Seam组件。

1．注册页面

register.xhtml页面看起来几乎与代码清单17-15中所示的如出一辙。移除了个性化表单字段（名、姓、电子邮件地址），并用Next Page按钮替换Register按钮：

```
...
<s:validateAll>

    <div class="entry">
        <div class="label">Username:</div>
        <div class="input">
            <s:decorate>
                <h:inputText size="16" required="true"
                             value="#{register.user.username}"/>
            </s:decorate>
        </div>
    </div>

    <div class="entry">
        <div class="label">Password:</div>

        <div class="input">
            <s:decorate>
                <h:inputSecret size="16" required="true"
                               value="#{register.user.password}"/>
            </s:decorate>
        </div>
    </div>

    <div class="entry">
        <div class="label">Repeat password:</div>
        <div class="input">
            <s:decorate>
```

```
                <h:inputSecret size="16" required="true"
                               value="#{register.verifyPassword}"/>
            </s:decorate>
        </div>
    </div>

</s:validateAll>

<div class="entry">
    <div class="label"> </div>
    <div class="input">
        <h:commandButton value="Next Page"
                         styleClass="button"
                         action="#{register.enterAccount}"/>
    </div>
</div>
```

你仍然正在引用register组件来绑定值和动作，很快会见到这个类。将表单值绑定到register.getUser()返回的User对象。currentUser不见了。你现在有了一个对话上下文，并且不再需要使用HTTP会话上下文了（如果用户试图同时在两个浏览器窗口中注册两个账户，前一个实现就不会生效）。register组件现在对话期间保存绑定到所有表单字段的User状态。

enterAccount()方法的结果把用户转到了下一页，即个性化表单。注意，你仍然依赖Hibernate Validator［由Seam在请求的处理验证阶段中进行调用（<s:validateAll/>）］进行输入验证。如果输入验证失败，页面就重新显示。

2. 个性化页面

profile.xhtml页面几乎与register.xhtml页面一样。个性化表单包括个性化字段，并且页面底部的按钮允许用户在对话中后退或者前进：

```
...

<div class="entry">
    <div class="label">E-mail address:</div>
    <div class="input">
        <s:decorate>
            <h:inputText size="32" required="true"
                         value="#{register.user.email}"/>
        </s:decorate>
    </div>
</div>

<div class="entry">
    <div class="label"> </div>

    <div class="input">
        <h:commandButton value="Previous Page"
                         styleClass="button"
                         action="register"/>

        <h:commandButton value="Next Page"
                         styleClass="button"
                         action="#{register.enterProfile}"/>
    </div>
</div>
```

当表单提交时，用户填写的任何表单字段都被应用到register.user模型。Previous Page按钮跳过调用应用程序阶段，并产生register结果——显示前一页。注意这个表单没有用<s:validateAll/>围绕起来，当用户单击Previous Page按钮时，你并不想要进行处理验证。调用Hibernate Validator现在被委托给了register.enterProfile动作。应该只在用户单击Next Page的时候验证表单输入。然而，你保留表单字段上的装饰，来显示任何验证错误的消息。

下一页概述了账户和个性化。

3. 概述页面

在confirm.xhtml中，所有的输入都呈现在概述中，允许用户在最终提交账户和个性化的详细信息用于注册之前，预览它们：

```
...
<div class="entry">
    <div class="label">Last name:</div>
    <div class="output">#{register.user.lastname}</div>
</div>

<div class="entry">
    <div class="label">E-mail address:</div>
    <div class="output">#{register.user.email}</div>
</div>

<div class="entry">
    <div class="label"> </div>
    <div class="input">
        <h:commandButton value="Previous Page"
                         styleClass="button"
                         action="profile"/>

        <h:commandButton value="Register"
                         styleClass="button"
                         action="#{register.confirm}"/>
    </div>
</div>
```

Previous Page按钮呈现了profile结果定义的响应，它就是前一个页面。当用户单击Register时，调用register.confirm方法。这个动作方法终止了对话。

最后，编写退回这个对话的Seam组件。

4. 编写对话的Seam组件

代码清单17-16中所示的RegisterBean必须被界定到对话。首先，这是该接口：

```
public interface Register {

    // Value binding methods
    public User getUser();
    public void setUser(User user);

    public String getVerifyPassword();
    public void setVerifyPassword(String verifyPassword);

    // Action binding methods
    public String enterAccount();
```

```
    public String enterProfile();
    public String confirm();

    // Cleanup routine
    public void destroy();
}
```

Seam对话模型的优势之一在于，可以像阅读会话的历史一样阅读你的接口。用户输入账户数据，之后是个性化数据。最后，确认并保存输入。

bean的实现如代码清单17-17所示。

代码清单17-17 对话范围的Seam组件

```
package auction.beans;

import ...

@Name("register")                                          ❶
@Scope(ScopeType.CONVERSATION)

@Stateful
public class RegisterBean implements Register {

    @PersistenceContext                                    ❷
    private EntityManager em;

    @In(create=true)
    private transient FacesMessages facesMessages;

    private User user;                                     ❸
    public User getUser() {
        if (user == null) user = new User();
        return user;
    }
    public void setUser(User user) {
        this.user = user;
    }

    private String verifyPassword;                         ❹
    public String getVerifyPassword() {
        return verifyPassword;
    }
    public void setVerifyPassword(String verifyPassword) {
        this.verifyPassword = verifyPassword;
    }

    @Begin(join = true)                                    ❺
    public String enterAccount() {
        if ( verifyPasswordMismatch() || usernameExists() ) {
            return null; // Redisplay page
        } else {
            return "profile";
        }
    }

    @IfInvalid(outcome = Outcome.REDISPLAY)                ❻
    public String enterProfile() {
```

```
        return "confirm";
    }

    @End(ifOutcome = "login")                    ❼
    public String confirm() {
        if ( usernameExists() ) return "register"; // Safety check
        em.persist(user);
        facesMessages.add("Registration successful!");
        return "login";
    }

    @Remove @Destroy
    public void destroy() {}

    private boolean usernameExists() {
        List existing =
            em.createQuery("select u.username from User u" +
                           " where u.username = :uname")
                .setParameter("uname",
                              user.getUsername())
                .getResultList();

        if (existing.size() != 0) {
            facesMessages.add("Username exists");
            return true;
        }
        return false;
    }

    private boolean verifyPasswordMismatch() {
        if (!user.getPassword().equals(verifyPassword)) {
            facesMessages.add("Passwords do not match");
            verifyPassword = null;
            return true;
        }
        return false;
    }
}
```

❶ 当Seam实例化这个组件时，一个实例就被绑定到了以变量register为名的会话上下文中。

❷ EJB 3.0容器注入一个事务范围的持久化上下文。稍后你会在这里用Seam注入一个会话范围的持久化上下文。

❸ user成员变量通过访问方法公开，因此JSF输入小部件可以被绑定到独立的User属性。对话期间user的状态由register组件保存。

❹ verifyPassword成员变量也通过访问方法公开给表单中的值绑定，并且在对话期间的状态被保存。

❺ 当用户在第一个屏幕中单击Next Page时，调用enterAccount()方法。当这个方法返回时，当前的对话通过@Begin被提升为长运行的对话，因此它跨越未来的请求，直到用@End进行标记的方法返回。由于用户可以退回到第一个页面，并重新提交表单，因此如果现存的对话已经在进

行，那么就需要联结它。

❻ 当用户在第二个屏幕中单击Next Page时，调用enterProfile()方法。由于它通过@IfInvalid作标记，Seam给输入验证执行Hibernate Validator。如果出现错误，就用Hibernate Validator中的错误消息重新显示页面（Outcome.REDISPLAY是一个方便的常量快捷方式）。如果没有错误，结果就是对话的最后一张页面。

❼ 当用户在最后一个屏幕中单击Register时，调用confirm()方法。当方法返回login结果时，Seam终止长运行的对话，并通过调用以@Destrory作标记的方法来销毁组件。同时，如果有其他人选择了相同的用户名，你就把用户转向到对话的第一个页面，对话上下文保持完整和活动。

本章前面已经介绍过了大多数的注解。唯一新出现的是@IfInvalid，它在调用enterProfile()方法时触发Hibernate Validator。注册对话现在完成了，一切都如预期般进行着。持久化上下文由EJB容器处理，当调用方法时，一个全新的持久化上下文就被分配到了每一个动作。

你还没有遇到任何问题，因为代码和页面没有通过从脱管的领域对象中拖取视图中的数据来按需加载数据。然而，几乎任何比注册过程更复杂的对话都将触发LazyInitialization-Exception。

17.5.2 让 Seam 管理持久化上下文

让我们来激发LazyInitializationException。当用户输入对话的最后一个屏幕（确认对话框）时，你呈现了一个拍卖分类的清单。用户可以给他们的账户选择默认的分类：他们想要浏览和从中出售项目的默认分类。分类的清单从数据库中加载，并通过获取方法公开。

1. 触发LazyInitializationException

编辑RegisterBean组件，并公开一个从数据库中加载的拍卖分类的清单：

```
public class RegisterBean implements Register {
    ...
    private List<Category> categories;
    public List<Category> getCategories() {
        return categories;
    }
    ...
    @IfInvalid(outcome = Outcome.REDISPLAY)
    public String enterProfile() {
        categories =
                em.createQuery("select c from Category c" +
                               " where c.parentCategory is null")
                  .getResultList();
        return "confirm";
    }
}
```

你还将getCategories()方法添加到组件的接口。在confirm.xhtml视图中，现在可以绑定到这个获取方法来显示分类：

```
...
<div class="entry">
    <div class="label">E-mail address:</div>
    <div class="output">#{register.user.email}</div>
</div>

<div class="entry">
    <div class="label">Default category:</div>
    <div class="input">
        <tr:tree var="cat"
                 value="#{registrationCategoryAdapter.treeModel}">
            <f:facet name="nodeStamp">
                <h:outputText value="#{cat.name}"/>
            </f:facet>
        </tr:tree>
    </div>
</div>
...
```

为了显示分类，你用了一个不同的小部件（它不在标准的JSF集中）。它是Apache Myfaces Trinidad项目中的一个可视化的树数据组件。它也需要一个将分类清单转换为树数据模型的适配器。但是这并不重要（可以在CaveatEmptor下载中为它找到那些库和配置）。

重要的是，如果分类树呈现，那么在调用enterProfile()之后，持久化上下文就已经在呈现响应阶段中被关闭。现在，哪些分类在脱管状态下完全可用？只有根类别（没有父类别的类别）已经从数据库中得以加载。如果用户单击树显示，并且想要看看一种分类是否包含任何子分类，应用程序便通过LazyInitializationException而失败。

通过Seam，可以轻松地扩展持久化上下文为跨越整个对话，而不仅仅是单个方法或者单个事件。然后，数据的按需加载在对话和任何JSF过程阶段中的任何位置便都成为可能。

2. 注入Seam持久化上下文

首先，配置Seam托管的持久化上下文。编辑（或者创建）WEB-INF目录下的文件components.xml：

```
<components>

    <component name="org.jboss.seam.core.init">

        <!-- Enable seam.debug page -->
        <property name="debug">false</property>

        <!-- How does Seam lookup EJBs in JNDI -->
        <property name="jndiPattern">
            caveatemptor/#{ejbName}/local
        </property>
    </component>

    <component name="org.jboss.seam.core.manager">

        <!-- 10 minute inactive conversation timeout -->
        <property name="conversationTimeout">600000</property>

    </component>

    <component
```

```
            name="caveatEmptorEM"
            class="org.jboss.seam.core.ManagedPersistenceContext">
            <property name="persistenceUnitJndiName">
                java:/EntityManagerFactories/caveatEmptorEMF
            </property>
        </component>

    </components>
```

你也将所有其他的Seam配置选项都移到了这个文件，因此seam.properties现在是空的（但是对于组件扫描器来说，作为标记它仍然是必需的）。

当Seam启动时，它将类ManagePersistenceContext配置为Seam组件。这就像把Seam注解放到该类中（在这个Seam绑定的类中也有注解）。组件的名称为caveatEmptorEM，它实现EntityManager接口。现在每当需要EntityManager时，就让Seam注入caveatEmptorEM。

（ManagePersistenceContext类需要知道如何获得真正的EntityManager，因此你必须在JNDI中提供EntityManagerFactory的名称。如何获取EntityManagerFactory到JNDI中，这取决于Java Persistence提供程序。在Hibernate中，可以通过在persistence.xml中的jboss.entity.manager.factory.jndi.name配置这个绑定。）

再一次修改RegisterBean，并使用Seam持久化上下文：

```
@Name("register")
@Scope(ScopeType.CONVERSATION)

@Stateful
public class RegisterBean implements Register {

    @In(create = true, value = "caveatEmptorEM")
    private EntityManager em;

    ...
```

当这个组件中的方法第一次调用时，Seam就会创建ManagedPersistenceContext的一个实例，将它绑定到对话上下文中的变量caveatEmptorEM中，并且在执行方法之前，将它注入到成员字段em里面。当对话上下文被销毁时，Seam也销毁ManagedPersistenceContext实例，这样就销毁了持久化上下文。

持久化上下文什么时候被清除呢？

3. 整合持久化上下文生命周期

每当事务提交时，Seam托管的持久化上下文就被清除。不用把你的动作方法包在事务（通过注解）中，而是让Seam也管理事务。这是JSF的不同Seam阶段监听器的任务，替换掉faces-config.xml中那个基本的任务：

```
<lifecycle>
 <phase-listener>
    org.jboss.seam.jsf.TransactionalSeamPhaseListener
 </phase-listener>
</lifecycle>
```

这个监听器使用了两个系统事务来处理一个JSF请求。一个事务在恢复视图阶段启动，并在调用应用程序阶段之后提交。这些阶段中的任何系统异常都触发事务的自动回滚。可以用一个异

常处理器准备一个不同的响应（这是JSF的弱点——必须在web.xml中用servlet异常处理器来实现这一点）。通过在动作方法执行完成之后提交第一个事务，使得动作方法中由SQL DML创建的任何数据库锁定都尽可能地短。

第二个事务跨越JSF请求的呈现响应阶段。按需拖动数据（并触发延迟加载的关联和集合的初始化）的任何视图都在这第二个事务中运行。这是一个其数据为只读的事务，因此在该阶段期间没有创建任何数据库锁定（如果你的数据库没有以可重复的读取模式运行，或者如果它有一个多版本的并发控制系统的话）。

最后，注意持久化上下文跨越对话，但是清除和提交可能发生在对话期间。因此，整个对话不具有原子性。当对话被提升为长运行时，可以用`@Begin(flushMode=FlushModeType.MANUAL)`禁用自动的清除；然后必须在对话终止时手工调用`flush()`（通常在以`@End`标识的方法中）。

持久化上下文现在可以通过Seam注入在任何组件（无状态或者有状态）中使用。对话中的持久化上下文始终相同，它表现得就像一个高速缓存和已经从数据库中加载的所有实体对象的同一性映射。

跨整个对话的被扩展持久化上下文具有一些乍看起来可能不太明显的其他好处。例如，持久化上下文不仅是同一性映射，还是对话期间已经从数据库加载的所有实体对象的高速缓存。

想象你没有保存请求之间的对话状态，而是在每个请求结束时，将每一块信息都推到数据库或者HTTP会话中（或者放到被隐藏的表单字段，或者cookie或者请求参数等中去）。当下一个请求命中服务器时，通过访问数据库、HTTP会话等再次装配状态。因为没有其他的有用上下文，也没有对话的编程模型，因此你必须给每个请求重组和分解应用程序状态。这个无状态的应用程序设计不可伸缩——无法给每一个客户端请求都命中数据库（对于伸缩最昂贵的那个层）！

开发人员试图通过启用Hibernate的二级高速缓存来解决这个问题。然而，用对话高速缓存伸缩一个应用程序，比用一个哑二级数据高速缓存伸缩它更值得关注。尤其在集群中，每当任何数据片段被任何节点修改时，二级高速缓存就强制在所有集群节点上的高速缓存更新。通过对话高速缓存，只有这个特定对话的负载均衡或者故障切换所需的节点必须参与当前对话数据的复制（在这个例子中是有状态会话bean的复制）。复制可以明显减少，因为没有全局共享的高速缓存需要同步。

我们很想讨论更多关于Seam的内容，并介绍一些其他的示例，但可惜篇幅有限。

17.6　小结

本章介绍了JSF、EJB 3.0，以及如何通过JBoss Seam框架改进利用这些标准的Web应用程序。我们讨论了Seam的上下文，以及组件如何以上下文的方式联结在一起。我们讨论了Seam与Hibernate Validator的整合，并且你了解了为什么Seam托管的持久化上下文是`LazyInitializationException`的首选解决方案。

如果你发现Seam的世界很精彩，还有更多的内容有待探索：

- Seam组件模型也支持事件/监听器概念，它允许组件通过松散耦合的观察者/可观察模式互相调用。

- 可以通过一个页面流描述符给对话启用有状态的导航流，替换无状态的JSF导航模型。这样解决了对话期间用户在浏览器中单击Back按钮时可能遇到的问题。
- Seam有一个复杂的并发模型用于服务器中的异步处理（与JMS整合），以及在对话中的并发性处理（Seam防止对话两次提交）。
- Seam允许你将对话和业务流程管理任务轻松地连在一起。它整合了工作流和JBoss jBPM的业务流程上下文（http://www.jboss.com/products/jbpm）。
- Seam整合了JBoss Rules（http://www.jboss.com/products/rules）。可以访问Seam组件中的策略，和来自规则的Seam组件。
- Seam带有JavaScript库。通过这个Remoting框架，可以轻松地从客户端代码调用Seam组件。Seam可以处理任何到你服务器的Ajax请求。
- Seam的Application Framework提供开箱即用的组件，让你能够轻捷地编写易伸缩的CRUD数据库应用程序。
- Seam组件很容易测试，无论是否使用了（可嵌入的）容器。Seam通过SeamTest超类，使得为TestNG进行集成测试和功能测试变得极为轻松；这个超类允许你编写模拟Web浏览器交互的脚本。

如果想要继续学习Seam，并探索上列没有的其他特性，请继续学习Seam参考文档中的教程。

附录A

SQL基础知识

包含着行和列的表，对于任何已经使用过SQL数据库的人来说是再熟悉不过的了。有时候你会看到表被称作关系（relation），行称作字节组（tuple），列称作属性（attribute）。这是关系数据模型（relational data model）的语言，是SQL数据库（不完全）实现的数学模型。

关系模型允许你定义数据结构和保证数据完整性的约束（例如，通过不接受不符合业务规则的值）。关系模型还定义限制、投影、笛卡儿积和关系联结[Codd，1970]的关系操作。这些操作让你利用数据去做那些有意义的事情，比如对它进行概述或者导航。

每一个操作都从一张指定的表或者表的组合中产生一张新表。SQL是用来在应用程序中表达这些操作［因此称作数据语言（data language）］，以及用来定义在其上面执行操作的基本表的一种语言。

编写SQL数据定义语言（Data Definition Language，DDL）语句来创建和管理表。我们称DDL定义了数据库模式。语句如CREATE TABLE、ALTER TABLE和CREATE SEQUENCE都属于DDL。

编写SQL数据操作语言（Data Manipulation Language，DML）语句以在运行时使用数据。让我们在CaveatEmptor应用程序的一些表的上下文中描述这些DML操作。

在CaveatEmptor中，你一般有像货品（item）、用户（user）和出价（bid）这样的实体。假设这个应用程序的SQL数据库模式包括一个ITEM表和一个BID表，如图A-1所示。这个模式的数据类型、表和约束都通过SQL DDL来创建（CREATE和ALTER操作）。

插入（insertion）是从旧表中通过添加一个行来创建一条新记录的操作。SQL数据库在某处执行这个操作，因此新行被添加到现有的表中：

```
insert into ITEM values (4, 'Fum', 45.0)
```

ITEM

ITEM_ID	DESCRIPTION	...
1	Item Nr. One	...
2	Item Nr. Two	...
3	Item Nr. Three	...

BID

BID_ID	ITEM_ID	AMOUNT
1	1	99.00
2	1	100.00
3	1	101.00
4	2	4.99

图A-1　包含示例数据的示例表

SQL update（更新）修改一个现有的行：

```
update ITEM set PRICE = 47.0 where ITEM_ID = 4
```

deletion（删除）移除一行：

```
delete from ITEM where ITEM_ID = 4
```

SQL的真正威力在于查询数据。单个查询可以在几张表中执行多个关系操作。来看一下基本的操作。

限制（restriction）是选择符合特定条件的表行的操作。在SQL中，这个条件是发生在where子句中的表达式：

```
select * from ITEM where NAME like 'F%'
```

投影（projection）是选择一张表的列、并从结果中消除重复行的操作。在SQL中，要包括的列都列在select子句中。可以通过指定distinct关键字消除重复的行：

```
select distinct NAME from ITEM
```

笛卡儿积［Cartesian product，也称作交叉联结（cross join）］生成一张新表，它由两张现有表的行的所有可能的组合构成。在SQL中，通过在from子句中列出表来表达笛卡儿积：

```
select * from ITEM i, BID b
```

关系的联结通过组合两张表的行生成新的表。对于联结条件（join condition）为真的每一对行，这张新表都包括一个行，包含来自两个被联结行的所有字段值。在ANSI SQL中，join子句指定一个表联结，联结条件在on关键字后。例如，为了获取所有包含出价的货品，可以在它们共有的ITEM_ID属性中联结ITEM和BID表：

```
select * from ITEM i inner join BID b on i.ITEM_ID = b.ITEM_ID
```

联结相当于一个有限制的笛卡儿积。因此，联结经常改为以theta风格来表达，通过from子句中的一个乘积和where子句中的联结条件。这个SQL theta风格的联结相当于前一个ANSI风格的联结：

```
select * from ITEM i, BID b where i.ITEM_ID = b.ITEM_ID
```

除了这些基本的操作之外，关系数据库还为聚合行（GROUP BY）和排序行（ORDER BY）定义操作：

```
select b.ITEM_ID, max(b.AMOUNT)
from BID b
group by b.ITEM_ID
having max(b.AMOUNT) > 15
order by b.ITEM_ID asc
```

SQL曾被称作结构查询语言，是关于所谓的子查询（subselect）的一项特性。因为每个关系操作都从一张现有的表或者几张表中生成新表，所以SQL查询可以在前一个查询的结果表中操作。SQL让你用单个查询表达这一点，通过把第一个查询嵌套在第二个里面：

```
select *
from (
    select b.ITEM_ID as ITEM, max(b.AMOUNT) as AMOUNT
    from BID b
    group by b.ITEM_ID
)
where AMOUNT > 15
order by ITEM asc
```

这个查询的结果相当于前一个结果。

子查询可以出现在SQL语句中的任何地方。在where子句中的子查询，是最值得关注的：

```
select * from BID b
   where b.AMOUNT >= (select max(c.AMOUNT) from BID c)
```

这个查询返回数据库中最大的出价。where子句子查询经常与量词（quantification）结合。下列查询是等价的：

```
select * from BID b
   where b.AMOUNT >= all(select c.AMOUNT from BID c)
```

SQL限制条件用一种复杂的表达语言表达，它支持数学表达式、函数调用、字符串匹配，甚至更复杂的特性如全文搜索：

```
select * from ITEM i
    where lower(i.DESCRIPTION) like '%gc%'
       or lower(i.DESCRIPTION) like '%excellent%'
```

附录 B

映射快速参考

许多Hibernate书都在附录中罗列了所有可能的XML映射元素和映射注解。这么做有多大用处值得质疑。首先，这些信息已经能以一种很方便的形式得到，你只需要知道如何得到它。其次，我们在这里可能增加的任何参考在几个月甚至几个星期就会过时了。核心的Hibernate映射策略不会那么经常变动，但是有一些小细节、选项和属性，则在改善Hibernate的过程中始终被修改着。

主要的原因难道不是你想要一个映射参考——所以你有所有选项的最新清单？

- 可以在Hibernate中绑定的hibernate-mapping-3.0.dtd中找到所有XML映射元素和属性的清单。在任何文本编辑器中打开这个文件，你会看到它被很好地制成了文档，非常容易阅读。如果使用XML映射文件的话，你可以把它打印出来作为一个快速参考。如果DTD的语法让你觉得困扰，就在这个文件的副本上做几个快速的搜索/替换操作，用你在打印输出中喜欢的东西替换DTD标签。
- 通过阅读Javadoc，可以找到`javax.persistence`和`org.hibernate.annotions`包所有映射注解的一个清单。Javadoc与Hibernate Annotions包捆绑在一起。例如，为了给所有的Hibernate扩展注解找到一个可单击的最新参考，请打开api/org/hibernate/annotions/package-summary.html。