Hibernate In Action

中文版

第一章 理解对象-关系持续性

我们工作的每个软件项目工程中，管理持续性数据的方法已经成为一项关键的设计决定。对于Java应用，持续性数据并不是一个新的或不寻常的需求，你也许曾经期望能够在许多相似的，已被很好构建的持续性解决方案中简单地进行选择。考虑一下Web应用框架(Jakarta Struts 对 WebWork)，GUI组件框架（Swing 对 SWT），或模版工具（JSP 对 Velocity）。每一种相互竞争的解决方案都有其优缺点，但它们至少都共享了相同的范围与总体的方法。不幸的是，这还不是持续性技术的情形，对持续性技术相同的问题有许多不同的混乱的解决方案。

在过去的几年里，持续性已经成为Java社区里一个争论的热点话题。对这个问题的范围许多开发者的意见甚至还不一致。持续性还是一个问题吗？它早已被关系技术与其扩展例如存储过程解决了。或者它是一个更一般的问题，必须使用特殊的Java组件模型例如EJB实体Bean来处理？甚至SQL和JDBC中最基本的CRUD（create, read, update, delete）操作也需要进行手工编码，还是让这些工作自动化？如果每一种数据库管理系统都有它自己的方言，我们如何达到可移植性？我们应该完全放弃SQL并采用一种新的数据库技术，例如面向对象数据库系统吗？争论仍在继续，但是最近一种称作对象-关系映射（ORM）的解决方案逐渐地被接受。Hibernate就是这样一种开源的ORM实现。

Hibernate是一个雄心勃勃的项目，它的目标是成为Java中管理持续性数据问题的一种完整的解决方案。它协调应用与关系数据库的交互，让开发者解放出来专注于手中的业务问题。Hibernate是一种非强迫性的解决方案。我们的意思是指在写业务逻辑与持续性类时，你不会被要求遵循许多Hibernate特定的规则和设计模式。这样，Hibernate就可以与大多数新的和现有的应用平稳地集成，而不需要对应用的其余部分作破坏性的改动。

本书是关于Hibernate的。我们包含了基本与高级的特征，并且描述了许多使用Hibernate开发新应用时的推荐方式。通常这些推荐并不特定于Hibernate——有时它们可能是我们关于使用持续性数据工作时处理事情的最佳方式的一些想法，只不过在Hibernate的环境中进行了介绍。然而，在我们可以开始使用Hibernate之前，你需要理解对象持续性和对象-关系映射的核心问题。本章解释了为什么像Hibernate这样的工具是必需的。

首先，我们定义了在面向对象的应用环境中持续性数据的管理，并且讨论了SQL，JDBC和Java的关系，Hibernate就是在这些基础的技术与标准之上构建的。然后我们讨论了所谓的对象-关系范例不匹配的问题和使用关系数据库进行面向对象的软件开发中所遇到的一些一般性的问题。随着这个问题列表的增长，我们需要一些工具与模式来最小化我们用在与持续性有关的代码上的时间就变得很明显了。在我们查看了可选的工具和持续性机制后，你会发现ORM在许多情况下可能是最好的解决方案。我们关于ORM的优缺点的讨论给了你一个完整的背景，在你为自己的项目选择持续性解决方案时可以作出最好的决定。

最好的学习方式并不必须是线性的。我们猜想你可能想立刻尝试一下Hibernate。如果这的确是你喜欢的方式，请跳到第2章第2.1节“开始”，在那儿我们进入并开始编写一个（小的）Hibernate应用。不读这一章你也可能理解第2章，但我们还是推荐你在循环通读本书的某一时刻再回到这一章。那样，你就可以准备好并且具有了学习其余资料所需的所有的背景概念。

1．1 什么是持续性？

几乎所有的应用都需要持续性数据。持续性在应用开发中是一个基本的概念。如果当主机停电时一个信息系统没有保存用户输入的数据，这样的系统几乎没有实际的用途。当我们讨论Java中的持续性时，我们通常是指使用SQL存储在关系数据库中的数据。我们从简单地查

看一下这项技术和我们如何在Java中使用它开始。具有了这些知识之后，我们继续关于持续性的讨论以及如何在面向对象的应用中实现它。

1．1．1 关系数据库

你，像许多其他的开发者，很可能在使用一个关系数据库进行工作。实际上，我们中的大多数每天都在使用关系数据库。关系技术是一个已知数。仅此一点就是许多组织选择它的一个充分的理由。但仅仅这样说就有点欠考虑了。关系数据库如此不易改变不是因为偶然的原因而是因为它们那令人难以置信的灵活与健壮的数据管理方法。

关系数据库管理系统既不特定于Java，也不特定于特殊的应用。关系技术提供了一种在不同的应用或者相同应用的不同技术（例如事务工具与报表工具）之间共享数据的方式。关系技术是多种不同的系统与技术平台之间的一个共同特征。因此，关系数据模型通常是业务实体共同的企业级表示。

关系数据库管理系统具有基于SQL的应用编程接口，因此我们称今天的关系数据库产品为SQL数据库管理系统，或者当我们讨论特定系统时，称之为SQL数据库。

1．1．2 理解SQL

为了有效地使用Hibernate，对关系模型和SQL扎实的理解是前提条件。你需要用你的SQL知识来调节你的Hibernate应用的性能。Hibernate会自动化许多重复的编码任务，如果你想利用现代SQL数据库的全部能力,你的持续性技术的知识必须扩充至超过Hibernate本身。记住基本目标是健壮地有效地管理持续性数据。

让我们回顾一些本书中用到的SQL术语。你使用SQL作为数据定义语言（DDL）通过CREATE与ALTER命令来创建数据库模式。创建了表（索引，序列等等）之后，你又将SQL作为数据处

理语言（DML）来使用。使用DML，你执行SQL操作来处理与取出数据。处理操作包括插入，更新和删除。你使用约束，投射，和连接操作（包括笛卡尔积）执行查询来取出数据。为了有效地生成报表，你以任意的方式使用SQL来分组，排序和合计数据。你甚至可以相互嵌套SQL命令，这项技术被称作子查询。你也许已经使用了多年的SQL并且非常熟悉用它编写的基本操作和命令。尽管如此，我们的经验还是告诉我们SQL有时难于记忆并且有些术语有不同的用法。为了理解本书，我们不得不使用相同的术语与概念；因此，如果我们提到的任何术语是新出现的或者是不清楚的，我们建议你看一下附录A。

为了进行健全的Java数据库应用开发，SQL知识是强制的。如果你需要更多的资料，找一本Dan Tow的优秀著作《SQL Tuning》。同时读一下《An Introduction to Database Systems》，学习一些（关系）数据库系统的理论，概念和思想。关系数据库是ORM的一部分，当然另一部分由你的Java应用中的对象构成，它们需要使用SQL被持续化到数据库中。

1．1．3 在Java中使用SQL

当你在Java应用中使用SQL工作时，Java代码通过Java数据库连接（JDBC）执行SQL命令来操作数据库。SQL可能被手工编码并嵌入到Java代码中，或者可能被匆忙地通过Java代码生成。你使用JDBC API将变量绑定到查询参数上，开始执行查询，在查询结果表上滚动，从结果集中取出值，等等。这些都是底层的数据访问任务，作为应用开发者，我们对需要这些数据访问的业务问题更加感兴趣。需要我们自己来关心这些单调的机械的细节，好像并不是确定无疑的。

我们真正想做的是能够编写保存和取出复杂对象（我们的类实例）的代码—从数据库中取出或者保存到数据库中，尽量为我们减少这些底层的苦差事。

因为这些数据访问任务通常都是非常单调的，我们不禁要问：关系数据模型特别是SQL是

面向对象的应用中持续性问题的正确选择吗？我们可以立即回答这个问题：是的！有许多原因使SQL数据库支配了计算行业。关系数据库管理系统是唯一被证明了的数据管理技术并且几乎在任何Java项目中都是一项需求。

然而，在过去的15年里，开发者一直在讨论范例不匹配的问题。这种不匹配解释了为什么每个企业项目都需要在持续性相关的问题上付出如此巨大的努力。这里所说的范例是指对象模型和关系模型，或者可能是面向对象编程与SQL。让我们通过询问在面向对象的应用开发环境中，持续性究竟意味着什么，来开始我们对不匹配问题的探究。首先，我们将本章开始部分声明的对持续性过分简单的定义扩展到一个较宽的范围，更成熟的理解包括维护与使用持续性数据。

1．1．4 面向对象应用中的持续性

在面向对象的应用中，持续性允许一个对象的寿命可以超过创建它的程序。这个对象的状态可能被存储到磁盘上，并且在将来的某一时刻相同状态的对象可以被重新创建。

这样的应用不仅仅限于简单的对象——关联对象的完整图形也可以被持续化并且以后可以在新的进程里被重新创建。大多数对象并不是持续性的；暂态对象只有有限的寿命，被实例化它的进程的寿命所限定。几乎所有的Java应用都在混合使用持续与暂态对象；因此，我们需要一个子系统来管理我们的持续性数据。

现代的关系数据库为持续性数据提供了一种结构化的表示方法，允许排序，检索和合计数据。数据库管理系统负责管理并发性和数据的完整性；它们负责在多个用户和多个应用之间共享数据。数据库管理系统也提供了数据级别的安全性。当我们在本书中讨论持续性时，我们考虑以下这些事情：

- 存储，组织与恢复结构化数据

- 并发性与数据完整性
- 数据共享

特别地，我们将在使用域模型的面向对象的应用环境中考虑这些问题。

使用域模型的应用并不直接使用业务实体的扁平表示进行工作，这些应用有它们自己的面向对象的业务实体模型。如果数据库中有“项目”与“竞价”表，则在这些Java应用中会定义“项目”与“竞价”类。

然后，业务逻辑并不直接在SQL结果集的行与列上进行工作，而是与面向对象的域模型进行交互，域模型在运行时表现为一个关联对象的交互图。业务逻辑从不在数据库中（作为SQL存储过程）执行，而是被实现在Java程序中。这就允许业务逻辑使用成熟的面向对象的概念，例如继承与多态。我们可以使用众所周知的设计模式例如“策略”,“中介者”和“组合” ，所有这些模式都依赖于多态方法调用。现在给你一个警告：并不是所有的Java应用都是按照这种方式设计的，并且也不打算是。对于简单的应用不使用域模型可能会更好。SQL和JDBC API可以完美地处理纯扁平数据，并且现在新的JDBC结果集已经使CRUD操作变得更简单了。使用持续性数据的扁平表示进行工作可能更直接并且更容易理解。

然而，对于含有复杂业务逻辑的应用，域模型有助于有效地提高代码的可重用性和可维护性。在本书中我们集中在使用域模型的应用上，因为通常Hibernate和ORM总是与这种类型的应用有关。

如果我们再次考虑SQL和关系数据库，我们最终会发现这两种范例的不匹配之处。

SQL操作例如投射与连接经常会导致对结果数据的扁平表示。这与在Java应用中用来执行业务逻辑的关联对象的表示完全不同！这是根本不同的模型，而不仅仅是相同模型的不同的显示方式。

带着这些认识，我们能够开始看一下这些问题——许多已经很好理解的与没有很好理解的——必须被合并使用这两种数据表示的应用所解决——一个面向对象的域模型与一个持续性的关系模型。让我们仔细地看一下。

1．2 范例的不匹配

范例不匹配的问题可以被分解成几部分，我们将依次进行分析。让我们通过一个独立于问题的简单例子开始我们的研究。随后当我们构建它时，你就会看到不匹配性问题的出现。

假设你需要设计与实现一个在线电子商务应用。在这个应用中你需要一个类来表示系统用户的信息，另一个类用来表示用户账单的详细资料（参见图**1.1**）。

Figure 1.1 A simple UML class diagram of the user and billing details entities

（图**1.1**）

看一下这个范例，你可以看到一个用户可以有多份账单资料，你可以在两个方向上对这两个类之间的关系进行导航。一开始，表示这些实体的类可能会非常简单：

```
public class User {
    private String userName;
    private String name;
    private String address;
    private Set billingDetails;
    ...
}

public class BillingDetails {
    private String accountNumber;
```

```
    private String accountName;
    private String accountType;
    private User user;
    ...
}
```

注意，我们仅对与持续性有关的实体状态感兴趣，因此我们省略了属性存取方法与业务方法（例如**getUserName()**或**billAuction()**）的实现。非常容易给这个例子提出一个好的**SQL**表设计：

```
create table USER (
    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通过外键来表示，表**BILLING_DETAILS**中的**USERNAME**就是外键。对于这个简单的对象模型，对象**-**关系的不匹配几乎不明显；你可以直接编写**JDBC**代码来插入，更新和删除用户与账单资料的信息。

现在，让我们看一下当我们考虑一个更现实的问题时会发生什么。当我们在我们的应用中加入更多的实体和实体关系时，范例不匹配的问题会变得更明显。

在我们当前的实现中最明显的问题是我们将地址设计成了一个简单的字符串值。在大多

数系统里，并不需要分别保存街道、城市、州、国家和邮政编码等信息。当然，我们可以直接在用户类里增加这些属性，但是因为系统中其它的类也非常可能含有地址信息，创建一个地址类可能更有意义。更新后的对象参见图1.2。

Figure 1.2 The User has an Address.

（图1.2）

我们同样需要增加一个地址表吗？不需要。通常只需将地址信息作为单独的列保存到用户表里。这样的设计可能执行起来效率更高，因为在一个单独的查询里我们不需要通过表连接来取得用户和地址。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，并且在用户表里使用这种类型的一个单独的列而不是许多新列。

基本上，我们有增加几列或者使用单独的一列（以一种新的SQL数据类型）这两种选择。这明显是一个粒度的问题。

1．2．1 粒度的问题

粒度是指你使用的对象的相对大小。当我们讨论Java对象和数据库表时，粒度问题意味着持续性对象可以以不同的粒度来使用表和列，而表和列本身都有粒度上固有的限制。

让我们回到我们的例子。增加一种新的数据类型，将Java地址对象在我们的数据库中保存为单独的一列，听起来好像是最好的方法。毕竟，Java中的一个新的地址类与SQL数据类型中的一个新的地址类型可以保证互用性。然而，如果你检查现在的数据库管理系统对用户定义列类型（UDT）的支持，你将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-关系扩展中的一种。不幸的是，UDT支持在

大多数SQL数据库管理系统中都是有些晦涩的特征，当然在不同的系统之间也不具有可移植性。SQL标准支持用户定义的数据类型，但是少的可怜。因为这个原因或者（任何）其它原因，此时在本项工作中使用UDT还不是一项共通的实践——并且你不可能遇到一个大量使用了UDT的遗留系统。因此我们不能将我们新的地址类的对象作为一个等价的用户定义的SQL数据类型保存到一个独立的新列中。对这个问题我们的解决办法是使用多个列，将它们定义成厂方定义的SQL类型（例如布尔，数值与字符串数据类型）。同时考虑到表的粒度，用户表通常如下定义：

```
create table USER (
    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)
)
```

这导致了下面的发现：我们的域对象模型中的类按照不同的粒度级别排列起来——从象用户这样的粗粒度的实体类到一个象地址这样的细粒度类，直到一个象邮政编码这样的简单的字符串属性值。

相比之下，在数据库的级别中只有两种粒度的级别是可见的：表例如用户和数量列例如邮政编码。这明显不如我们的Java类型系统灵活。许多简单的持续性机制未能识别这种不匹配性因此不再限制对象模型上的更不灵活的表示。我们曾经无数次地看到在用户类中包含一个邮政编码的属性。

可以证明粒度问题并不是特别难以解决。的确，我们甚至可能都不列出它，之所以列出

是因为在许多现有系统中它都非常明显这个事实。我们将在第3章，第3.5节“细粒度的对象模型”中描述这个问题的解决方案。

当我们考虑使用了继承的域对象模型时，一个更困难并且更有趣的问题就会出现。继承是面向对象设计的一个特征，我们可以用它以一种更新更有趣的方式给我们的电子商务应用的用户开账单。

1．2．2 子类型的问题

在Java中，我们使用超类与子类实现继承。为了介绍这为什么会出现不匹配性的问题，让我们继续构建我们的例子。扩展一下我们的电子商务应用以使我们现在不仅能接受银行账户账单，还能接受信用卡与借记卡。因此我们有几种方法可以给一个用户账户开账单。在我们的对象模型中反映这个改变的最普通的方式是继承账单详细资料类。

我们可能有一个抽象的账单详细资料的超类和几个具体的子类：信用卡，直接借记，支票等等。这些子类中的每一个都会定义一些稍微不同的数据（与处理这些数据的完全不同的功能）。图1.3的UML类图介绍了这个对象模型。

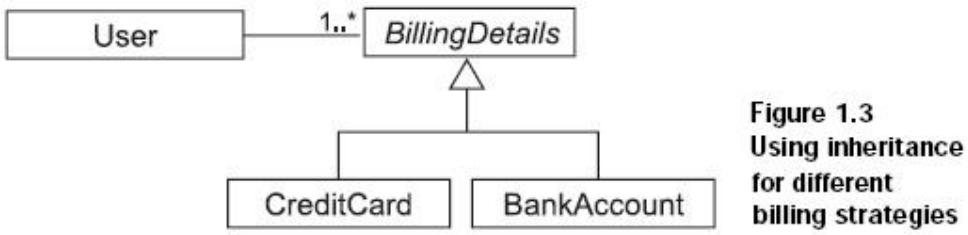

Figure 1.3
Using inheritance for different billing strategies

（图1.3）

我们立刻就会注意到SQL没有提供对继承的直接支持。我们不能够通过编写“CREATE TABLE CREDIT_CARD_DETAILS EXTENDS BILLING_DETAILS (...)”将表CREDIT_CARD_DETAILS声明为表BILLING_DETAILS的一个子类型。

在第3章第3.6节“映射类继承”里，我们讨论了象Hibernate这样的对象-关系映射解决方案如何解决将类层次持续化到数据库表中的问题。在社区里这个问题已经被很好地理解了，并且大多数的解决方案支持近似相同的功能。但是对于持续性问题我们还没有完全结束——只要我们将继承引入到对象模型中，就会出现多态的可能性。

用户类与账单详细资料超类有一个关联，这是一个多态性的关联。运行时，一个用户对象可能与账单详细资料的任何子类的实例相关联。相似地，我们希望能够针对账单详细资料类编写查询并且让它返回子类的实例。这个特征被称作多态性查询。

因为SQL数据库不支持继承的概念，它们没有明显的表示多态性关联的方式也不会太令人惊讶。一个标准的外键约束精确地引用一个表；定义一个引用多个表的外键并不是那么简单。我们可能以Java（与其它面向对象语言）的输入不如SQL那么严格来解释这个问题。幸运地是，我们将在第3章介绍两种继承映射的解决方案，它们被设计为用来解决多态性关联的表示和有效地执行多态性查询。

因此，子类型不匹配是指Java模型中的继承结构必须被持续化到SQL数据库中，而SQL数据库并没有提供一个支持继承的策略。不匹配问题的另一方面是对象的同一性问题。你可能已经注意到了我们将用户表中的用户名定义成了主键。那是一个好的选择吗？与你下面将要看到的一样，那的确不是。

1．2．3 同一性的问题

虽然对象同一性的问题一开始可能并不明显，但在我们不断增长与扩展的电子商务系统的例子中我们会经常遇到它。这个问题在我们考虑两个对象（例如，两个用户）并且检查它们是否是同一个时就会看到。有三种方法可以解决这个问题，两种使用Java，一种使用我们的SQL数据库。与你预期的一样，想让它们协同工作需要提供一些帮助。

Java对象定义了两种不同的相等性的概念：

- 对象同一性（粗略的等同于内存位置的相等，使用a==b检查）
- 通过equals()方法的实现来决定的相等性（也被称作值相等）

另一方面，数据库行的同一性使用主键值表示。象你将在第3.4节“理解对象同一性”中看到的一样，主键既不必然地等同于“equals()”也不等同于“==”。它通常是指几个对象（不相同的）同时表示了数据库中相同的行。而且，为一个持续类正确地实现equals()方法包含许多微妙的难点。

让我们通过一个例子讨论另一个关于数据库同一性的问题。在我们用户表的定义中，我们已经使用了用户名作为主键。不幸的是，这个决定使改变一个用户名变得很困难：我们不仅需要更新用户表中的用户名列，而且也要更新账单详细资料表中的外键列。因此，在本书后面的部分，我们推荐你无论什么情况如果可能的话都可以使用代替键。代替键是指对用户没有意义的主键列。例如，我们可以象这样改变我们的表定义：

```
create table USER (
    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 USER
)
```

列USER_ID和BILLING_DETAILS_ID包含系统生成的值。引入这些列纯粹是为了关系数据模型的好处。它们如何（即使从不）在对象模型中表示？我们将在第3.4节讨论这个问题并且找到一个对象-关系映射的解决方案。

在持续性环境中，同一性与系统如何处理缓存和事务密切相关。不同的持续性解决方案选择不同的策略，这已经成为混乱的一方面。我们包含了所有这些有趣的主题——并显示了他们是如何相关的——在第5章。

我们已经设计和实现的电子商务应用的构架很好地达到了我们的目的。我们识别出了映射粒度，子类型和对象同一性这些不匹配性的问题。我们几乎准备好了转移到应用的其它部分。但是首先，我们需要讨论一下关联这个重要的概念——也就是说，我们的类之间的关系是如何被映射和处理的。数据库中的外键就是我们所需要的全部吗？

1．2．4 与关联有关的问题

在我们的对象模型中，关联描述了实体间的关系。你记得用户，地址和账单详细资料类都是相互关联的。与地址不同，账单详细资料依赖于自身。账单详细资料对象保存到它们自己的表中。在任何对象持续性解决方案中，实体关联的映射与管理都是中心概念。

面向对象的语言使用对象引用和对象引用的集合表示关联。在关系世界里，关联被表示为外键列，外键是几个表的键值的拷贝。这两种表示之间有些微妙的不同。

对象引用具有固有的方向性，关联是指从一个对象到另一个对象的引用。如果对象间的关联可以在两个方向上进行导航，你需要定义两次关联，在每个关联类上分别定义一次。这你早已在我们的对象模型类中见过了：

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

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

另一方面，外键并不通过固有的方向性进行关联。事实上，对关系数据模型来说导航没有任何意义，因为你可以通过表连接和投射创建任意的数据关联。

实际上，只看Java类不可能确定单向关联的多重度。Java关联可以具有多对多的多重度。例如，我们的对象模型可能有看起来像这样的：

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

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

另一方面，表关联总是一对多或是一对一的。通过查看外键的定义你可以立刻知道多重度。下面是一个一对多的关联（或者，从另一个方向看，是多对一的）：

```
    USER_ID BIGINT FOREIGN KEY REFERENCES USER
```

这些是一对一的关联：

```
    USER_ID BIGINT UNIQUE FOREIGN KEY REFERENCES USER
    BILLING_DETAILS_ID BIGINT PRIMARY KEY FOREIGN KEY REFERENCES USER
```

如果你希望在关系数据库中表示一个多对多的关联，你必须引入一个新表，称为连接表。这个表不会在对象模型中的任何地方出现。对我们的例子来说，如果我们认为用户和用户的帐单信息之间为多对多的关系，连接表可以如下定义：

```
CREATE TABLE USER_BILLING_DETAILS (
    USER_ID BIGINT FOREIGN KEY REFERENCES USER,
    BILLING_DETAILS_ID BIGINT FOREIGN KEY REFERENCES BILLING_DETAILS
    PRIMARY KEY (USER_ID, BILLING_DETAILS_ID)
)
```

我们将会在第3和第6章非常详细地讨论关联映射。

目前为止，我们讨论的主要是结构上的问题。我们考虑系统的一个纯静态的视图就可以看到这些问题。或许在对象持续性中最困难的问题是动态的。它涉及到关联，在第1.1.4节“面向对象应用中的持续性”中，当我们描述对象图导航与表连接的区别时，就已经暗示过了。让我们更彻底地探讨这个重要的不匹配性问题。

1．2．5 对象图导航的问题

你在Java中访问对象的方式与在关系数据库中有根本的不同。在Java中，访问用户的账单信息时，你调用`aUser.getBillingDetails().getAccountNumber()`。这是最自然的面向对象数据的访问方式，通常被形容为遍历对象图。根据实例间的关联，你从一个对象导航到另一个对象。不幸地是，这不是从SQL数据库中取出数据的有效方式。

为了提高数据访问代码的性能，唯一重要的事是最小化数据库请求的次数。最明显的方式是最小化SQL查询的数量（其它方式包括使用存储过程或者JDBC批处理API）。

因此，使用SQL有效地访问关系数据通常需要在有关的表之间使用连接。在连接中包含的表的数量决定了你可以导航的对象图的深度。例如，如果我们需要取出用户，而对用户帐单的详细资料不感兴趣，我们使用这个简单查询：

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

另一方面，如果我们需要取出相同的用户并且随后访问每一个关联的账单详细资料的实例，我们使用一个不同的查询：

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

像你看到的一样，当我们取出最初的用户时，在我们开始导航对象图之前，我们需要知道我们计划访问对象图的哪一部分。

另一方面，仅当第一次访问一个对象时，所有对象持续性解决方案才提供取出关联对象数据的功能。然而，在关系数据库环境中，这种逐步的数据访问风格基本上效率是比较低的，因为，它需要在对象图的每个节点上执行一条选择语句。这就是所谓的n+1次选择的问题。

我们在Java与关系数据库中访问对象方式的不匹配，可能是Java应用中性能问题的一个最普遍的根源。虽然我们有无数的书籍与杂志论文的保佑，建议我们使用StringBuffer进行字符串连接，但好像不可能找到任何关于避免n+1次选择问题的策略。非常幸运，Hibernate提供了有效地从数据库中取出对象图的成熟的特征，使应用可以透明地访问这些图。我们将在第4和第7章讨论这些特征。

现在我们已经有了一个非常详细的对象-关系不匹配问题的列表，就像你根据经验得知的

那样，为这些问题寻找解决方案的代价可能是非常高的。这个代价经常被低估，我们认为这是大多数软件项目失败的主要原因。

1．2．6 不匹配的代价

对不匹配问题列表的全面的解决方案可能需要花费大量的时间和努力。根据我们的经验，在Java应用中编写的可能达到30%的代码，主要的目的是为了处理单调的SQL/JDBC和对象-关系范例不匹配的手工连接。不管所有这些努力，最终的结果可能仍然感觉并不完全正确。我们曾经见过一些因为数据库抽象层的复杂性和僵硬性而几乎要垮掉的项目。

一个主要的代价在模型化方面。关系与对象模型必须包含相同的业务实体。与一个有经验的关系数据建模者相比，一个面向对象的纯粹主义者可能以一种非常不同的方式模型化这些实体。对这个问题，通常的解决方案是扭曲对象模型直到它与下层的关系技术匹配为止。

这可能会成功，但代价是损失了许多对象方向的优点。记住，关系模型有关系理论所支持。对象方向上没有如此严格的数学定义和理论工作的支柱。因此，我们不能期望使用数学来解释我们应该如何融合这两种范例——没有优雅的变化等待去发现（放弃Java和SQL从头开始并不是一种深思熟虑过的优雅的方式）。

域模型不匹配的问题不是僵硬性的唯一原因，它使生产性降低并导致了更高的成本。另一个原因是JDBC API本身。JDBC和SQL提供了一个面向语句（即命令）的方法从SQL数据库中来回移动数据。至少在三个时刻（Insert，Update，Select）必须指定一个结构化关系，这增加了设计和实现所需要的时间。每种SQL数据库的独特的方言并没有改善这种情况。

最近，考虑以体系结构或基于模式的模型作为对不匹配问题的部分的解决方案已经开始流行。因此，我们有了实体bean组件模型，数据访问对象（DAO）模式，和其它实现数据访问

的实践。这些方法将以前列出的大部分或全部的问题留给了应用开发者。为了转向对对象持续性的理解，我们需要讨论应用体系结构和在典型的应用设计中持续层的角色。

1．3 持续层与可选方案

在一个大、中型的应用中，根据关系来组织类通常很有意义。持续性就是一种关系。其它的关系包括表示、工作流和业务逻辑。也有所谓的“**cross-cutting**”关系，通常这可能由框架代码实现。典型的“**cross-cutting**”关系包括日志，验证和事务划分。

一个典型的面向对象的体系结构包含了表示这些关系的层。通常，当然也是一项最佳实践，是在一个分层的系统体系结构中将所有与持续性有关的类和组件分组到一个独立的持续层中。

在本节中，我们首先查看一下这种体系结构中的层和我们为什么要使用它们。然后，我们集中到我们最感兴趣的层——持续层——和一些可以实现它的方式。

1．3．1 分层的体系结构

分层的体系结构定义了实现不同关系的代码之间的接口，允许关系实现方式的改变不会对其它层的代码造成重大的破坏。分层也决定了其间出现的中间层的类型。规则如下：

- 层由上到下进行通信。每一层仅依赖于其直接的下层。
- 除了其直接下层，每一层都不知道任何其它层。

不同的应用以不同的概念分组，因此会定义不同的层。一个典型的，已被证明的，高层的应用体系结构使用三层，分别为表示层，业务逻辑层和持续层，参见图**1.4**。

让我们仔细地看一下图中的层和元素：

■ 表示层——最高层的用户接口逻辑。负责显示、页面控制和屏幕导航的代码构成了表示层。

■ 业务层——下一层的确切形式，不同的应用可能会有很大的差异。它通常是已协商一致的，然而，作为问题域的一部分业务层应该被用户所理解，它负责实现任何业务规则或系统需求。在许多系统里，这一层有它自己的业务域实体的内部表示。在其它的系统里，它重用了表示层定义的模型。我们将在第3章再次讨论这个问题。

■ 持续层——持续层是负责从一个或多个数据仓库中存入和取出数据的一组类和组件。它必须包含一个业务域实体模型（甚至只是一个元数据模型）。

■ 数据库——数据库位于Java应用之外。它是系统状态持续性的实际表示。如果使用了一个SQL数据库，它会包含关系模式或者可能有存储过程。

■ 帮助类/工具类——每个应用都会有一组基础的帮助类或工具类，在应用的每一层它们都会被使用（例如，用于异常处理的异常类）。这些基础元素并不构成一层，因此在分层的体系结构中，它们不用遵守中间层的依赖规则。

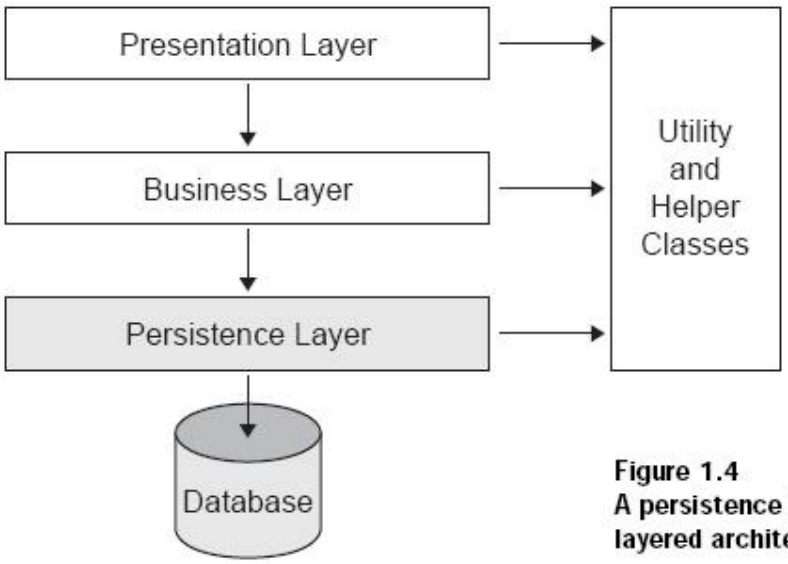

Figure 1.4
A persistence layer is the basis in a layered architecture.

（图1.4）

让我们简单地看一下Java应用实现持续层的不同方式。不必担心——我们将很快开始ORM和Hibernate。查看一下其它方法可以使我们学到很多东西。

1．3．2 使用SQL/JDBC手工编码持续层

实现Java持续性的最普遍的方式可能是应用程序员直接使用SQL和JDBC进行工作。毕竟，开发者熟悉关系数据库管理系统，理解SQL，也知道如何使用表和外键进行工作。此外，他们可以始终使用众所周知并广泛使用的DAO设计模式对业务逻辑隐藏复杂的JDBC代码和不可移植的SQL。

DAO是一种很好的模式——如此之好以至于我们甚至推荐与ORM一起使用它。然而，为域中的每个类手工编写持续性代码的工作是非常可观的，特别是需要支持多种SQL方言时。这项工作通常会消耗很大一部分的开发努力。此外，当需求改变时，一个手工编码的解决方案总是需要更多的注意和维护努力。

因此为什么不实现一个简单的ORM框架来满足你的项目的特定需求呢？这种努力的结果甚至可以在以后的项目中被重用。许多开发者已经采取了这种方法；在现今的产品系统中有许多自制的对象-关系持续层。然而，我们并不推荐这种做法。优秀的解决方案早已存在了，不仅有由商业厂商销售的工具（通常非常昂贵）而且也有使用自由许可证发布的开源项目。我们确信不论在业务或技术上，你都能够找到一种满足需求的解决方案。有可能这样的解决方案会比你在有限的时间里构建的做得更多，做得更好。

开发一个适度的包含全部特征的ORM可能需要花费很多人月。例如，Hibernate包含43,000行代码（其中许多比典型应用的代码要困难得多）和12，000行的单体测试代码。这可能远远多于你的应用。大量的细节很容易被忽略——象所有开发者从经验中得知的那样！即使一个现有的工具没有完全实现你的两三个更奇异的需求，也根本不值得自己创建。任何ORM都会处

理这些单调的普遍的情况——正是这些真正地破坏了生产性。你也可能需要针对特殊情况进行手工编码；很少应用主要由特殊情况构成。

不要被“现在尚未发明”的综合症所欺骗，就开始你自己的对象-关系映射的努力，而这只是为了避免第三方软件的学习曲线。即使如果你断定所有的ORM的材料都很狂热，并且你想以尽可能接近SQL数据库的方式工作，其它现有的持续性框架也都没有实现完全的ORM。例如，iBATIS数据库层是一个开源的持续层，它处理了许多更单调的JDBC代码但却让开发者手工编写SQL。

1．3．3 使用序列化

Java有一个内建的持续性机制：序列化提供了将对象图（应用状态）写到字节流中的能力，然后它可能被持续化到文件或数据库中。序列化也被Java的远程方法调用（RMI）使用来为复杂对象传递值语义。序列化的另一种用法是在机器集群中跨节点复制应用状态。

为什么不在持续层中使用序列化呢？很不幸，一个相互连接的对象图在序列化之后只能被当作一个整体访问，如果不反序列化整个流就不可能从流中取出任何数据。这样，结果字节流肯定会被认为不适合进行任意的检索或聚合。甚至不可能独立地访问或更新一个单独的对象或子图。为了支持高并发性，在进行系统设计时，除了在每个事务中装载与重写一次完整的对象图外没有其它的选择。

非常明显，因为当前特定的技术，序列化不适合于作为高并发性的Web和企业应用的持续性机制。在特定的环境中它被作为桌面应用的适当的持续性机制。

1．3．4 考虑EJB实体Bean

最近几年，企业JavaBeans（EJB）已经成为持续性数据的推荐方式。如果你一直在Java

企业应用领域工作，你可能已经使用过EJB特别是实体Bean了。如果没有，你也不用担心——实体Bean的流行度正在迅速地下降（尽管开发者大部分的关注还会集中在新的3.0规范上）。

实体Bean（在当前的EJB2.1规范中）非常有趣，因为与这里提到的其它解决方案相比，它们完全是通过委员会创建的。其它的解决方案（DAO模式，序列化和ORM）则是根据多年的经验提炼出来的；他们代表了经受了时间考验的方法。或许并不令人惊讶，EJB2.1实体Bean在实践中彻底地失败了。EJB规范的设计缺陷阻碍了Bean管理的持续性（BMP）实体Bean有效地执行。在EJB1.1许多明显的缺陷被纠正之后，一个边缘的稍微可接受的解决方案是容器管理的持续性（CMP）。

然而，CMP并不能表示一种对象-关系不匹配的解决方案。

这儿有六方面的原因：

■ CMP Bean与关系模型中的表是按照一对一的方式定义的。这样，它们的粒度过粗，不能够完全利用Java丰富的类型。在某种意义上，CMP强迫你的域模型成为最初通常的格式。

■ 另一方面，使用CMP Bean来实现EJB宣称的目标——可重用的软件组件，又显得粒度太细。一个可重用的组件必须是一个粒度非常粗的对象，面对数据库模式的一些小的改变它必须提供一个稳定的外部接口（是的，我们的确刚声明过CMP实体Bean的粒度既太粗又太细）。

■ 虽然EJB可以利用继承实现，但实体Bean并不支持多态的关联和查询——真正的ORM定义的特征。

■ 不管EJB规范所宣称的目标，实体Bean实际上是不可移植的。CMP引擎的性能因厂商而异，并且映射元数据也是高度特定于厂商的。许多项目选用Hibernate是因为Hibernate应用在不同的应用服务器之间具有更高的可移植性这个简单的原因。

■ 实体Bean不可序列化。我们发现当我们需要将数据传输到远程客户层时，我们必须定义额外的数据传输对象（DTO，也被称作值对象）。从客户端到远程实体Bean的实例使用细粒

度的方法调用是不可扩展的，DTO提供了一种远程数据访问的批处理方式。DTO模式导致了并行的类层次的增长，域模型中的每一个实体都要同时使用一个实体Bean和一个DTO来表示。

■ EJB是一种插入式模型；它要求一种不自然的Java风格并且在特定容器之外重用代码极其困难。对于测试驱动开发来说这是一个巨大的障碍。在需要批处理或其它离线功能的应用中，它甚至可能会引起问题。

我们不会花费更多的时间讨论EJB2.1实体Bean的优缺点。在查看了他们的持续性能力之后，我们得出了它们不适合完全的对象映射的结论。我们将拭目以待新的EJB3.0规范会有哪些提高。让我们转向另一种值得注意的对象持续性解决方案。

1．3．5 面向对象的数据库系统

因为我们使用Java对象工作，如果有一种方式根本不需要扭曲对象模型就能将这些对象存储到数据库中，那将是非常理想的。在90年代中期，新的面向对象的数据库系统引起了人们的注意。

与外部数据仓库相比，面向对象的数据库管理系统（OODBMS）更像是应用环境的扩展。通常OODBMS以一个多层实现作为主要特征，包含一个后端数据仓库，对象缓存，以及一个紧密结合的通过私有网络协议交互的客户端应用。

面向对象的数据库开发首先是宿主语言绑定的自上而下的定义，这些绑定为编程语言增加了持续性能力。因此，对象数据库提供了与面向对象应用环境无缝的集成。这不同于当今关系数据库使用的模型，它们通过中间语言（SQL）与数据库交互。

与ANSI SQL——关系数据库的标准查询接口类似，对象数据库产品也有标准。对象数据管理组（ODMG）规范定义了一组API，包括一种查询语言，一种元数据，以及C++、SmallTalk

和Java的宿主语言绑定。大多数面向对象的数据库系统对ODMG标准都提供了许多程度的支持，但据我们所知，还没有完全的实现。此外，在规范发布以后的很多年，甚至到了3.0版，还是感觉不太成熟，并且缺乏很多有用的特征，特别是基于Java环境的。ODMG也不再活跃。最近，Java数据对象（JDO）规范（发表于2002年4月）揭开了新的可能性。JDO由面向对象数据库团体的成员驱动，除了对现有的ODMG的支持之外，面向对象的数据库产品现在还经常将其作为主要的API采用。是否这些新的努力能够使面向对象数据库渗入到CAD/CAM（计算机辅助设计/建模）、科学计算以及其它市场环境中，还有待观察。

我们不会更进一步地查看为什么面向对象的数据库技术没有流行起来——我们只是简单地观察到对象数据库现在还没有被广泛地采用并且看来近期内也不太可能。根据当前行政上的实际情况（预定义的开发环境），我们确信绝大多数开发者还会有很多使用关系技术工作的机会。

1．3．6 其它选择

当然，也有其它类型的持续层。XML持续层是序列化模式的变种；这种方法通过允许工具可以很容易地访问数据结构，解决了一些字节流序列化的限制（但是它本身导致了对象/层次的不匹配）。此外，使用XML并没有其它的好处，因为它仅仅是另外一种文本文件格式。你也可以使用存储过程（甚至可以在Java里使用SQLJ编写）将这些问题移到数据库层。我们确信还有许多其它的例子，但最近没有哪一种可能会流行起来。

行政上的限制（在SQL数据库上长期的投资）与访问有价值的遗留数据的需求都要求一种不同的方法。对于我们的问题，ORM可能是最现实的解决方案。

1．4 对象-关系映射

至此，我们已经查看了对象持续性的可选技术，现在是介绍我们感觉最好的，并且是Hibernate使用的解决方案（ORM）的时候了。不管它很长的历史（第一篇研究论文发表于80年代后期），开发者对这个术语的使用也不相同。一些称它为“对象关系映射”，另一些则喜欢更简单的“对象映射”。我们统一使用术语“对象-关系映射”和它的首字母缩写ORM。中间的短线强调了当这两个领域相碰撞时出现的不匹配问题。

本节，我们首先查看一下什么是ORM。然后我们列举出一个好的ORM解决方案需要解决的问题。最后，我们讨论ORM提供的大致的好处与我们为什么推荐这种解决方案。

1．4．1 什么是ORM？

简单地说，对象-关系映射就是Java应用中的对象到关系数据库中的表的自动的（和透明的）持续化，使用元数据对对象与数据库间的映射进行了描述。本质上，ORM的工作是将数据从一种表示（双向）转换为另一种。

这意味着有一些性能损失。然而，如果ORM是作为中间件实现的，就会有许多机会可以进行优化而在手工编码的持续层中这些机会是不存在的。另外一项开销（在开发时）是对控制转换的元数据的准备与管理。而且，这个成本低于维护一个手工编码的解决方案所需的成本。相比之下，与ODMG兼容的对象数据库甚至需要大量类级别的元数据。

FAQ ORM不是一个Visio插件吗？首字母缩略语ORM也可以指对象角色建模，而且这个术语是在相关的对象-关系映射之前发明的。它描述了一种在数据库建模中使用的信息分析方法，并且主要由微软的Visio（一种图形化建模工具）支持。数据库专家使用它作为更流行的实体-关系建模的替代品或附属品。然而，如果你与Java开发者讨论ORM的话，通常是在对象-关系

映射的环境里。

ORM解决方案有以下四部分组成：

- 在持续类的对象上执行基本的CRUD操作的一组API。
- 用于指定查询的一种语言或一组API，这些查询会引用类和类属性。
- 用于指定映射元数据的工具。
- 实现ORM的一项技术，用来与事务对象交互以完成脏检查、懒关联存取和其它优化功能。

我们使用ORM这个术语包含所有可以根据元数据的描述自动生成SQL的持续层。我们不包含开发者通过编写SQL和使用JDBC手工解决对象-关系映射问题的持续层。使用ORM，应用与ORM API和根据下层SQL/JDBC抽象出来的域模型类进行交互。依赖于这些特征或特定的实现，ORM运行时也可能承担例如乐观锁、缓存等问题的职责，完全免去了应用对这些问题的关心。

让我们看一下可以实现ORM的各种不同的方式。Mark Fussel,一位ORM领域的研究者，定义了ORM品质的四个级别。

纯关系

整个应用，包括用户接口，都是围绕关系模型和基于SQL的关系操作进行设计的。如果低水平的代码重用是可以容忍的，不考虑它对大型系统的不足，这种方法对于简单的应用不失为一种优秀的解决方案。直接的SQL可以在每一方面进行微调，但是这些缺点例如缺乏可移植性和可维护性是非常重要的，特别是需要长期运行时。这种类型的应用经常大量地使用存储过程，将业务层的许多工作移动到了数据库中。

轻量对象映射

实体作为类来表示，而类又被手工地映射到关系表。手工编码的SQL/JDBC使用众所周知的设计模式对业务逻辑进行了隐藏。这种方法非常普遍并且对于那些只有少量实体的应用或者那些使用普通的元数据-驱动的数据模型的应用来说是很成功的。在这种类型的应用中存储

过程也可能有一席之地。

中等对象映射

这种应用是围绕对象模型设计的。SQL在编译时使用代码生成工具生成，或者在运行时由框架代码生成。对象间的关联由持续性机制支持，并且查询可能使用面向对象的表达式语言指定。对象被持续层缓存。很多的ORM产品和自制的持续层都至少支持这一级别的功能。它非常适合包含一些复杂事务的中等规模的应用，特别是当不同的数据库产品间的移植性非常重要时。这些应用通常不使用存储过程。

完全对象映射

完全的对象映射支持复杂的对象模型：组合、继承、多态和“可达的持续性”。持续层实现了透明的持续性；持续类不必继承任何特定的基类或实现任何特定的接口。高效的存取策略（懒惰和即时存取）和缓存策略都对应用透明地实现了。这一级别的功能很难由自制的持续层达到——它相当于数月或数年的开发时间。许多商业的和开源的Java ORM工具已经达到了这一级别的品质。这一级别满足了我们在本书中使用的ORM的定义。让我们看一下我们期望使用达到了完全对象映射级别的工具所能解决的一些问题。

1．4．2 一般的ORM问题

下面的问题列表，我们称之为对象-关系映射问题，是Java环境中完全的对象-关系映射工具解决的一些基本问题。特殊的ORM工具可能提供额外的功能（例如，积极缓存）。这是一个相当详细的概念问题的列表，而且是对象-关系映射特定的：

1 持续类像什么？它们是细粒度的JavaBean吗？或者它们是一些类似于EJB的组件模型的实例吗？持续性工具有多么透明？我们需要为业务领域的类采用一种编程模型或一些规范吗？

2 映射元数据是如何定义的？因为对象-关系转换完全由元数据控制，这些元数据的格式和定义是重要的核心问题。ORM工具应该提供一个图形化处理元数据的GUI吗？或者有定义元数据的更好的方法吗？

3 我们应该映射类的继承层次吗？这有几种标准策略。多态关联、抽象类和接口怎么映射呢？

4 对象同一性和相等性如何关联到数据库同一性（主键）？我们如何将特定类的实例映射到特定表的行。

5 在运行时持续性逻辑如何与业务域对象交互？这是一个普通的编程问题，有许多的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编译时字节码增强。这个问题的解决方案可能影响到你的构建过程（但宁可如此，你也不愿受到其它像用户那样的影响）。

6 持续性对象的生命周期是什么样的？有些对象的生命周期依赖于其它关联对象的生命周期吗？我们如何将一个对象的生命周期转化为数据库行的生命周期？

7 为排序、检索和合计提供了什么样的工具？应用可以在内存中处理其中的一些事情。但为了有效地使用关系技术有时需要通过数据库完成这些工作。

8 我们如何有效地取出关联数据？对关系数据的有效访问通常通过表连接实现。面向对象的应用通常通过导航对象图访问数据。可能的话，两种数据访问模式应该避免n+1次选择的问题，以及它的补充笛卡尔积的问题（在一次查询中取出过多的数据）。

另外，有两个问题对任何数据访问技术都是共通的。它们在ORM的设计和架构上强加了一些基本的限制。

■ 事务和并发性

■ 缓存管理（和并发性）

像你能够看到的一样，一个完全的对象映射工具需要处理一个相当长的问题列表。我们将在第3、4、5章讨论Hibernate管理这些问题的方式和一些数据访问的问题，稍后我们在本书中展开这个主题。

到此，你应该可以看到ORM的价值了。下一节，我们看一下当你使用ORM解决方案时，你将得到的一些其它好处。

1．4．3 为什么选择ORM？

ORM的实现是非常复杂的——可能不如应用服务器复杂，但要比像Struts或Tapestry这样的Web应用框架复杂得多。为什么我们要将另一个新的复杂的基础元素引入我们的系统呢？这样做值得吗？

对这些问题提供一个完全的回答将占用本书大部分的篇幅。为了避免你不耐烦，本章对大多数引人注目的好处提供了一个快速的概览。但首先，让我们快速地处理一些不是好处的问题。

ORM的一个假定的优点是对棘手的SQL保护开发者。这种观点认为面向对象的开发者不期望对SQL或关系数据库有很好的理解，而且不知为什么他们发现SQL非常令人讨厌。正好相反，我们认为Java开发者为了使用ORM工作，必须要充分熟悉——和感激——关系模型和SQL。ORM是开发者使用的一项高级技术，而且早以困难的方式实现过。为了有效地使用Hibernate，你必须能够查看和解释SQL语句的问题并能理解其性能含义。

让我们看一些ORM和Hibernate的好处。

生产性

与持续性有关的代码可能是Java应用中最乏味的代码。Hibernate去掉了很多让人心烦的工作（多于你的期望），让你可以集中到业务问题上。不论你喜欢哪种应用开发策略——自顶向下，从域模型开始；或者自底向上，从一个现有的数据库模式开始——使用Hibernate和适当的工具将会减少大量的开发时间。

可维护性

更少的代码行数（LOC）使系统更容易理解因为它们强调了业务逻辑而不是管道设备。更重要的，一个系统包含的代码越少则越容易重构。自动的对象-关系持续性充分地减少了LOC。当然，统计代码行数是度量应用复杂性的值得争议的方式。

然而，Hibernate应用更容易维护有其它方面的原因。在手工编码的持续性系统中，关系表示和对象模型之间存在一种不可避免的紧张。改变一个几乎总是包含改变其它的。并且一种表示设计经常需要妥协来适应其它的存在（实际上几乎总是发生的是域对象模型进行妥协）。ORM在这两种模型之间提供了一个缓冲，允许Java一方更优雅地进行面向对象的使用，并且每个模型都对其它模型的轻微改动进行了绝缘。

性能

一个共同的断言是手工编码的持续性与自动的持续性相比可能至少一样快，并且经常更快一些。在相同的意义上：汇编代码至少与Java代码一样快，或者手工编写的解析器至少与YACC或ANTLR生成的一样快，这的确是真的——换句话说，这有点离题了。这种断言未明确说明的含意是在实际的应用中手工编码的持续性至少应该完成得一样好。但这种含意只有在实现至少一样快的手工编码的持续性所需的努力与利用自动的解决方案包含的努力相似的情况下才是对的。实际令人感兴趣的问题是，当我们考虑时间和预算的限制时会发生什么？

对一项给定的持续性任务，可以进行许多优化。许多（例如查询提示）通过手工编码的

SQL/JDBC也很容易完成，然而，使用自动的ORM完成会更简单。在有时间限制的项目中，手工编码的持续层通常允许你利用一点时间做一些优化。Hibernate则允许你在全部的时间内做更多的优化。另外，自动的持续性给开发者提供了如此高的生产性因此你可以花更多的时间手工优化一些其余的瓶颈。

最后，ORM软件的实现人员可能有比你更多的时间来研究性能优化问题。你知道吗，例如，缓存PreparedStatement的实例对DB2的JDBC驱动导致了一个明显的性能增长但却破坏了InterBase的JDBC驱动？你了解吗，对某些数据库只更新一个表中被改变的字段可能会非常快但潜在地对其它的却很慢？在你手工编写的解决方案中，对这些不同策略之间的冲突进行试验是多么不容易呀？

厂商独立性

ORM抽象了你的应用使用下层SQL数据库和SQL方言的方式。如果工具支持许多不同的数据库（大部分如此），那么这会给你的应用带来一定程度的可移植性。你不必期望可以达到“一次编写，到处运行”，因为数据库的性能不同并且达到完全的可移植性需要牺牲更强大的平台的更多的力气。然而，使用ORM开发跨平台的应用通常更容易。即使你不需要跨平台操作，ORM依然可以帮你减小被厂商锁定的风险。另外，数据库独立性对这种开发情景也有帮助：开发者使用一个轻量级的本地数据库进行开发但实际产品需要配置在一个不同的数据库上。

1.5 总结

在本章中，我们讨论了对象持续性概念和ORM作为一项实现技术的重要性。对象持续性意味着对象个体可以比应用进程的寿命更长；它们可以被保存到数据仓库里并且以后可以被及时重建。当数据仓库是基于SQL的关系数据库管理系统时对象/关系不匹配的问题就会出现。例如，对象图不能被简单地保存到数据库表里；它必须被分解并被持续化到轻便的SQL数据类

型里。，ORM是这个问题的一个很好的解决方案，当我们考虑类型丰富的Java域模型时它特别有用。

域模型表示了在Java应用中使用的业务实体。在一个分层的系统体系结构中，域模型用来执行业务层中的业务逻辑（在Java中，而不是在数据库中）。为了装载与存储域模型中的持续性对象，业务层与其下面的持续层进行通信。ORM是持续层中管理持续性的中间件。

ORM并不是所有持续性任务的银弹；它的目标是减少开发者95%的对象持续性工作，例如使用多表连接编写复杂的SQL语句或将值从JDBC结果集中拷贝到对象或对象图中。一个包含全部特征的ORM中间件可能提供数据库间的可移植性，特定的优化技术，例如缓存和其它不容易在有限的时间里使用SQL和JDBC手工编码的可行的功能。

可能有一天会出现一种比ORM更好的解决方案。我们（和其他很多人）或许必须重新考虑我们知道的关于SQL，持续性API标准和应用集成的每一件事。现在的系统将会进化成纯关系数据库系统与面向对象的无缝的集成还仅仅只是推测。但是，我们不能等待，并且没有任何迹象表明这些问题会很快地改善（一个数十亿美元的产业不会是非常灵活的）。ORM是当前可用的最好的解决方案，它是每天都要面对对象-关系不匹配问题的开发者节省时间的一种方式。