# 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

**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。

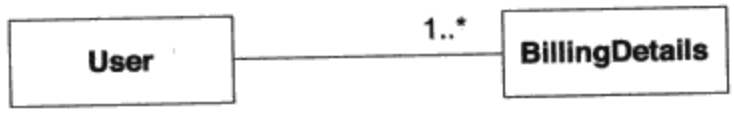


图1-1　User和BillingDetails实体的一个简单的UML类图

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。

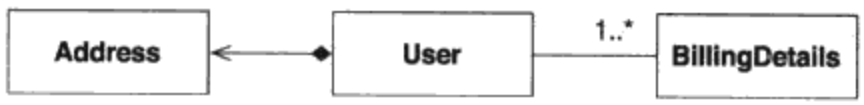


图1-2 User有一个Address

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。

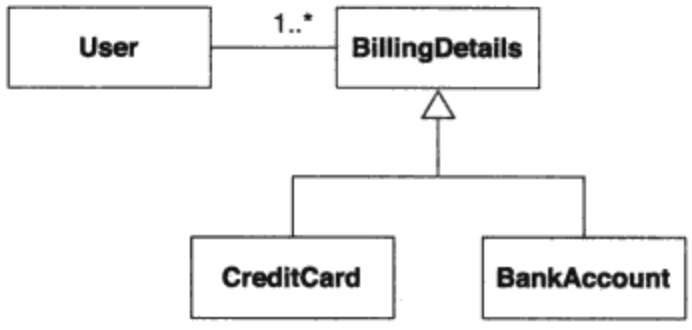


图1-3 为不同的记账策略使用继承

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。

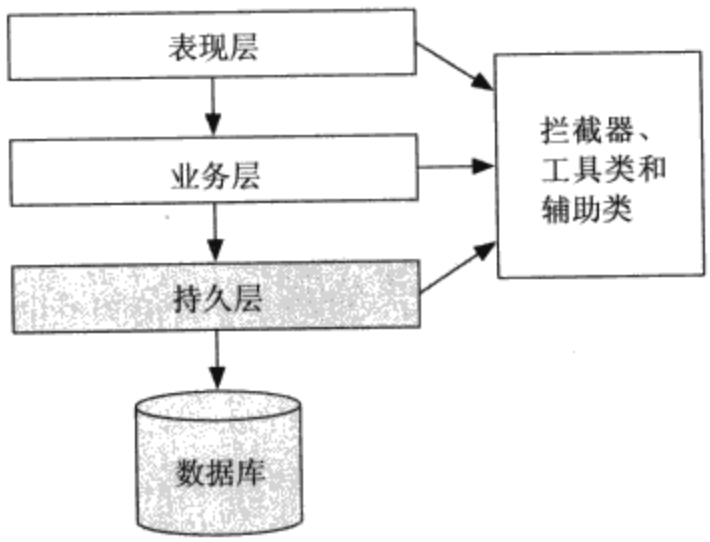


图1-4　持久层是层架构的基础

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