

## 第 3 章

# 映射到关系数据库

数据源层的作用是与应用需要的基础设施的不同部分进行通信。问题主要是和数据库的会话，换句话说，是和现今系统中广泛使用的关系数据库进行会话。当然，现在仍然有大量的数据采用比较老的存储格式，比如ISAM和VSAM文件系统。但是，现在大多数构建系统的人更担心关系数据库问题。

关系数据库之所以取得成功，最重要的原因之一就是SQL的存在，它是数据库通信标准语言。尽管不同厂商各自增强的细节给SQL带来了复杂性，但它的核心语法还是非常通俗易懂的。

### 3.1 架构模式

首先介绍的一组模式是架构模式，它要解决的问题是驱动领域逻辑访问数据库的方式。此时的选择对于设计影响深远而且难以重构，因此这个问题需要注意。同样，如何设计领域逻辑也会对这个选择产生重大影响。

尽管SQL已经在商业软件中广泛应用，但它在使用中还是存在一些缺陷。许多应用程序开发者并不能充分理解SQL，因此，不能很好地构造有效的查询语句和命令。尽管现在有很多技术可以把SQL语句嵌入到程序设计语言中，但它们多少都有点笨拙，可能使用适合程序开发语言的机制来访问数据会更好。数据库管理员（DBA）也希望能得到访问数据表的SQL语句，这样他们就能理解怎样才能最好地调整和组织索引。

基于这些原因，把SQL访问从领域逻辑中分离出来，并把它放到独立的类中，实在是明智之举。有一种方法能很好地组织这些类：让它们以数据库中的表结构为基础，这样，每一个数据库表对应一个类。这些类为数据表建立了一个入口。应用程序的其他部分根本不需要了解任何与SQL有关的事情，而且很容易就能找到所有访问数据库的SQL。这样就使专攻数据库的开发者有了一个清晰的目标。

使用入口的方法主要有两种。最显而易见的是为查询语句返回的每一行产生一个它的实例（见图3-1）。这种行数据入口就像是面向对象的方式来对待数据。

许多环境提供记录集，这是表和数据行的一种通用数据结构，用来模拟数据库的表格式属性。因为记录集是一种通用数据结构，环境可以在应用程序的很多地方使用它。GUI工具常

Person Gateway
lastname firstname numberOfDependents
insert update delete find (id) findForCompany(companyID)

图3-1 行数据入口对于一次查询逐行返回一个实例

用记录集来进行控制。如果使用记录集，对数据库中的每个表，仅仅需要一个对象<sup>⊖</sup>来管理。这种表数据入口（见图3-2）提供了查询数据库的方法，返回一个记录集。

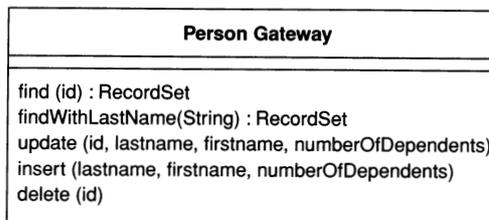


图3-2 表数据入口对每个表有一个实例

即使对于简单的应用程序，我也倾向于使用一种入口模式，浏览一下我的Ruby和Python脚本就可以看到这一点。清晰的SQL和领域逻辑分离是相当有益的。

表数据入口与记录集非常匹配，这使得它们成为使用表模块的当然选择。它也是一个组织存储过程的模式。许多设计者都喜欢通过存储过程来完成所有的数据库访问，而不是直接使用SQL语句。在这种情况下，可以考虑把存储过程的集合看成是为一个表定义的表数据入口。可能还有一个内存中的表数据入口来包装对存储过程的调用，这样就可以把存储过程的调用机制封装起来。

如果使用领域模型，还会有更多的选择。当然可以将一个行数据入口或者表数据入口与领域模型一起使用。不过，就我看来，这样要么过于间接，要么又不够间接。

在简单应用中，领域模型是一种和数据库结构相当一致的简单结构，对应每个数据库表都有一个领域类。这种领域对象的业务逻辑复杂度通常适中。在这种情况下，有必要让每个领域对象负责数据库的存取过程，这也就是活动记录（见图3-3）。从另一个角度来考虑活动记录，就是从行数据入口开始，然后把领域逻辑加入到类中，特别是在从多个事务脚本中发现了重复代码的时候。

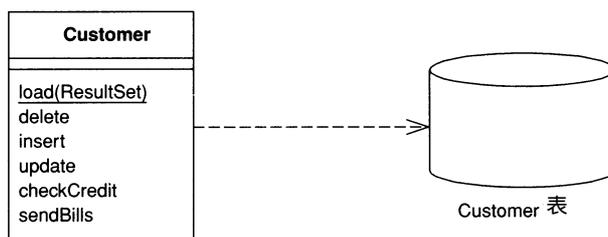


图3-3 在活动记录中，一个顾客领域对象知道如何与数据库表交互

在这种情况下，入口增加的间接性提供的价值不大。随着领域逻辑变得更加复杂，它就慢慢转变成一个大的领域模型，简单的活动记录开始不能满足要求了。领域类和表的一对一匹配也开始随着把领域逻辑放入更小的类而失效。关系数据库无法处理继承，因此使用策略模式

<sup>⊖</sup> 原书这里是“类”，根据作者在个人网站（www.martinfowler.com）上公布的勘误表，改为“对象”。——编辑注

[Gang of Four]和其他轻巧的面向对象模式非常困难。随着领域逻辑日益活跃，你会希望不用访问数据库就能随时测试它。

所有这些都迫使你随着领域模型的增大而采用间接的方式。在这种情况下，入口可以解决一些问题，但它仍然将数据库方案和领域模型耦合在一起。结果就会有一些从入口域到领域对象域转换，这种转换会使领域对象变得复杂。

一种更好的办法是把领域模型和数据库完全独立，可以让间接层完成领域对象和数据库表之间的映射。这个数据映射器（见图3-4）处理数据库和领域模型之间所有的存取操作，并且允许双方都能独立变化。这是数据库映射架构中最复杂的架构，但它的好处是把两个层完全独立了。

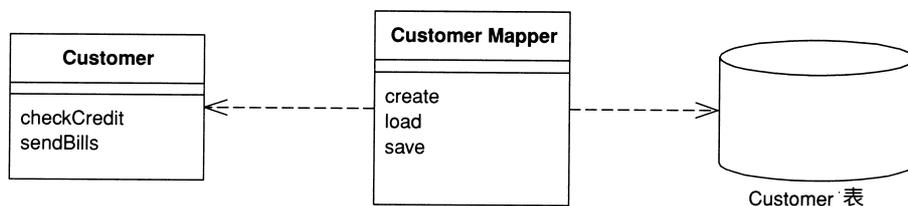


图3-4 数据映射器使领域对象和数据库彼此完全独立

我不推荐把入口用作领域模型的首选持久化机制。如果领域逻辑非常简单并且类和表十分一致，使用活动记录就足够了。如果领域逻辑比较复杂，数据映射器才是需要的。

这些模式并不是完全不能兼容的。在很多讨论中，我们关心的是首选持久化机制，通过它可以确定如何将某种内存数据模型保存到数据库中。因此，有必要从这些模式中选择一种，而不要把它们混合在一起，凌乱不堪。即使用数据映射器作为首选持久化机制，还是可以使用数据入口来封装被视为外部接口的表或服务。

在对这些思想的讨论中，无论是这里还是后面关于模式本身的讨论，我都倾向于使用“表”这个词。因为本书谈到的大多数技术对于视图、由存储过程封装的查询，甚至动态查询等都是大同小异的。但是，对于表/视图/查询/存储过程没有广泛使用的术语，所以我使用“表”这个词，仅仅因为它表达出了表型的数据结构。我通常把视图看作虚表，这也是从SQL的角度上看到的。查询视图时也使用和查询数据表同样的语法。

对于视图和查询语句来说，更新容易带来麻烦，因为不能直接对一个视图进行更新，必须去更新这个视图所对应的基础数据表。在这种情况下，实现更新逻辑的首选方法是把视图/查询用一个合适的模式来进行封装，它将使视图的使用变得更加简单可靠。

这样使用视图和查询的问题之一是：对于那些不知道视图如何组成的开发人员来说，可能会带来一些意想不到的不一致性。这些开发人员可能会对两个不同的结构执行更新，而这两个更新操作可能更新同一个基础表，这样，第二次操作就会重写第一次的操作结果。当然，如果更新逻辑做了必要的确认工作，就不会出现这样的不一致，但是这样做还是会让开发人员不好理解。

下面介绍用最简单的方法来持久化哪怕是最复杂的领域模型。在对象技术出现的早期，许多人意识到在对象和关系之间有一个“阻抗不匹配”的问题，因此随之掀起了一股面向对象数据库的研究热潮，从本质上来说，面向对象数据库是将面向对象的理论体系带到了磁盘存储领

域。使用面向对象数据库时，无需担心映射的问题。你可以使用由许多相互关联的对象组成的巨大结构，用数据库来决定何时存取对象。你还可以通过事务将更新操作分组，或支持数据存储的共享。对于程序员来说，这就好像是一个由磁盘存储器支持的无限事务性内存。

面向对象数据库的主要好处在于它们能够提高生产率。尽管我没听说过什么受控测试，但根据不严格的观察，关系数据库的映射开销大概是程序开发总开销的1/3左右，而且还会一直持续到维护阶段。

然而，大多数项目并不使用面向对象数据库。主要原因是风险。关系数据库是一种非常容易理解、并且有很多成熟的大型供应商提供长期支持的技术。SQL为所有的工具提供相关的标准接口。（如果关心性能的话，我只能说我没有看见过任何面向对象数据库与关系数据库性能比较的结论性数据。）

就算不能使用面向对象数据库，但假如有领域模型，也应该认真考虑是否购买O/R映射工具。虽然本书中的模式花了不少笔墨介绍如何构建数据映射器，但是这样做仍然很复杂。工具提供商花了多年的时间来研究这个问题，而且商业O/R映射工具能比任何相当的手工产物都要经得起检验。当然，这种工具往往也价格不菲，因此有必要权衡一下自行开发及维护数据映射层的开销与购买工具的开销，再来决定如何做。

当前，现在也有一种趋势是提供面向对象数据库风格的逻辑层，与关系数据库一同工作。JDO就是Java世界中的一个例子。但是这一技术趋势究竟能否解决问题，现在还很难说。在这方面，我没有足够的经验，不能在本书中做出任何结论性判断。

即使购买了映射工具，弄懂这些模式仍然很有用处。优秀的O/R工具会在映射到数据库的时候提供大量选项，这些模式将有助于理解何时采用不同的选择。不要认为有了工具就可以一劳永逸。工具是一个好的开始，但是，你仍然会发现，使用和调节O/R工具是很小但很有意义的一部分工作。

### 3.2 行为问题

谈到O/R映射，人们通常会关注结构方面如何把表和对象联系起来。然而，我发现：实践中最难的部分在于架构和行为方面。我已经介绍过主要的架构相关内容，接下来考虑一下行为问题。

所谓行为问题，就是如何让各种对象从数据库中读取出来以及存到数据库中。乍一看，这似乎不成问题。一个客户对象可以拥有加载和保存方法来进行这项工作。确实，用活动记录，这是一种显而易见的路线。

如果加载了一些对象到内存并且进行了修改，就必须跟踪每个修改过的对象，并保证把它们写回到数据库中。如果仅仅加载了两条记录，这是很容易的。一旦加载的对象越来越多，这就不再是一件容易的事情，尤其是在创建了某些行的同时、还修改了其他的行的时候，这是由于在修改引用它们的行之前要获得这些新建行的主键。而且这个问题虽然小，但不容易解决。

因为要读取对象并修改它们，所以就必须保证正在使用的数据库状态的一致性。如果读取了某些对象，重要的是要保证读取必须是独占的，也就是说，没有其他进程在读取的同时修改这些对象。否则，就可能在对象中得到不一致或无效的数据。这就是同步问题，一个非常棘手

的问题；我们将在第5章中详细讨论这个问题。

有一种专门解决上述问题的模式就是工作单元。工作单元会跟踪所有从数据库中读取的对象以及所有以任何形式修改过的对象。它同样负责将更新提交到数据库。应用程序的编程人员将工作交托给工作单元，而不是直接调用明确的保存方法。工作单元排列好对数据库的操作顺序，把所有复杂的提交处理放在一起。当与数据库的交互动作比较复杂的时候，工作单元是一个必要的模式。

可以这样理解工作单元，它是一个对象，充当数据库映射的控制器。在没有工作单元的情况下，一般都是由领域层充当控制器，决定何时读写数据库。工作单元就是来源于把数据库映射控制器的行为分解到它自己的对象中。

加载对象时，必须小心避免把同一个对象加载两次，否则，在内存中就有两个对象和同一个数据库行对应。对它们都进行更新就会乱套了。为了解决这个问题，可以在标识映射里记录读取的每一行。每次读入数据时，必须到标识映射里去检查一下是不是已经存在了。如果该数据已经加载，可以返回一个对它的引用。这样，所有更新操作就可以正确地组织好。还可以得到一些好处，比如可能避免一些数据库调用，因为标识映射就像一个数据库高速缓存。不过不要忘了，标识映射的主要目的是保持一致性，而不是提高性能。

如果使用了领域模型，就必须合理安排，使得关联的对象一起加载，例如，在读取一个订单对象的同时，把与之相关联的客户对象也一同加载进来。然而，如果许多对象都是连接在一起的，则读取任何对象都会从数据库中带出大批的对象。为了避免这种低效，必须设法减少带出来的东西，当然，还需要保持接口以便在以后需要的时候再来取。延迟加载的主要思想是拥有一个对象引用的占位符。可以采用几种方法，但它们的共同点都是拥有被修改对象的对象引用，它指向的是一个占位符而不是实际的对象。当且仅当想要通过链接访问的时候，才会真的去数据库中读取实际的对象。适当使用延迟加载能使每次数据库调用取得刚好够用的数据。

### 3.3 读取数据

读取数据的时候，可以把读取数据的方法看作一个查找器，它通过一个方法结构的接口来隐藏SQL查询语句。因此，可以使用诸如`find(id)`或者`findForCustomer(customer)`这样的方法。当然，如果查询语句有23个不同的子句，那么这些方法会非常低效，不过谢天谢地，这种情况非常少见。

在什么地方放置查找器方法是由使用的接口模式决定的。如果数据库交互类是基于表的，也就是说对于数据库中的每个表都有一个类的实例与之对应，那么就能把插入和更新操作也捆绑在查找器方法中。如果交互类是基于数据行的，也就是说对于数据库中的每一行都有一个类的实例与之对应，这种情况就不行了。

用基于行的类可以使查找变成静态操作，但是这样就会使数据库操作不可替代。这也就意味着不能通过服务桩在测试的时候调换数据库。为了避免这个问题，最好是创建独立的查找器对象。每一个查找器类都有很多封装了SQL语句的方法。当执行查询操作的时候，查找器对象返回一个适当的基于行的对象集合。

使用查找器方法时要注意的是：这些查找器方法工作在数据库状态下而不是对象状态下。

如果发出一个对数据库的查询语句找到一个俱乐部内所有的人，那么任何你在内存中加入到俱乐部的人都不会返回。解决这类问题的办法是在一开始就进行查询。

读取数据的时候，性能问题可能会变得比较突出。这就导致了几条经验法则。

尽量一次读回多行。实际上，最好不要为了得到多行而在同一个表上重复查询。得到的数据多往往比得到的数据少好（尽管你必须要注意通过悲观并发控制一次锁定了太多的行）。因此，考虑这种情况，需要得到50个人，这50个人可以通过领域模型中的主键确定，但是你能构造一个查询得到200个人，在这200人当中必须使用更多的逻辑判断来取得想要的50个人。不过，通过一次查询得到一些冗余的行要比进行50次独立的查询好。

另一个避免多次进入数据库的方法是使用联接（Join），这样就可以通过一次查询返回多个表。得到的记录集可能看起来比较奇怪，但是确实能加快查询速度。在这种情况下，可以建立一个入口来得到相互联接的表数据，或者通过一个数据映射器用一次调用加载多个领域对象。

然而，如果正在使用联接操作，记住数据库必须优化到在一次查询中处理3~4个联接。一旦超出这个范围，将会带来性能损失，尽管可以通过缓存视图来重新装入大部分数据。

数据库中可以进行很多优化。这些优化包括把相互关联的数据组织到一起，小心地使用索引，以及数据库的缓存功能。这些内容超过了本书的范围，但是作为一个好的DBA，需要好好掌握这些内容。

在所有的情况下，应该根据你具体的数据库和数据来对应用程序进行分析。一些通用的规则可以作为指导，但是你的特定环境往往有它们自己的变化。数据库系统和应用服务器经常有复杂的缓冲机制，很难预测具体的应用程序会发生什么情况。对于每个我用过的经验法则，我都听说过令人惊讶的异常情况，所以要用些额外的时间进行性能剖析和调整。

### 3.4 结构映射模式

当人们谈到对象 - 关系映射的时候，他们多半说的是结构映射模式，在内存对象与数据库表的映射中会用到它们。这些模式通常与表数据入口并不相关，但是可以在使用行数据入口或者活动记录的时候使用其中一些模式。在使用数据映射器的时候，所有模式可能都会需要。

#### 3.4.1 关系的映射

这里的关键问题在于对象和关系处理连接的方法不同，这会带来两个问题。首先，表现方法不同。对象是通过在运行时（内存管理环境或内存地址）中保存引用的方式来处理连接的。关系数据库则通过创建到另外一个表的键值来处理连接。其次，对象可以很容易通过集合来表示多个引用，而规范化则要求所有的关系连接都必须是单值的。这就导致对象和表之间的数据结构颠倒了。一个订单对象自然拥有一个订单项的集合，而这些订单项不需要持有订单对象的引用。然而，表结构中的各订单项必须包含一个到订单的外键，因为订单不能有一个多值域。

解决这种表现问题的方法是：通过对象中的一个标识域来保持每个对象的关系特性，并且通过查找这些值来保持对象引用和关系键之间的相互映射。这是一个乏味的过程，但是一旦理解了基本原理后却并不困难。从硬盘中读取数据的时候，使用标识映射作为从关系键到对象的查找表。每次使用表中的外键，都用外键映射（见图3-5）来得到合适的对象间引用。如果标识

映射中没有该键值，就需要到数据库中读取它或者使用延迟加载。每次存储对象时，就可以用正确的键值把它存到行中，用目标对象的ID域替换任意的对象间引用。

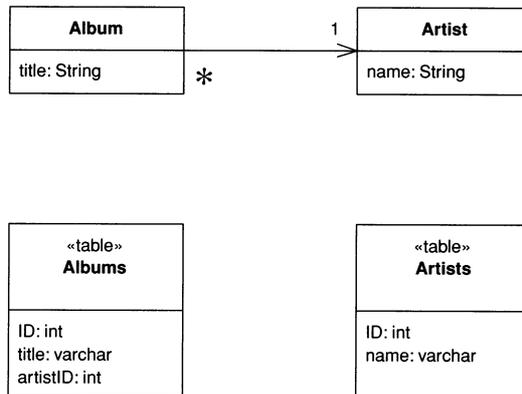


图3-5 使用外键映射来映射一个单值域

在这个基础上，集合的处理需要更复杂的外键映射版本（见图3-6）。如果对象包含一个集合，则必须构造一个新的查询来找到所有与源对象的ID相关的行（也可以通过延迟加载来避免查询）。创建每个返回的对象并加入到集合中。对这个集合的保存包括：保存其中每一个对象，并且保证它拥有一个到源对象的外键。这样非常混乱，尤其是当要检测对象加入或者移出这个集合的时候。当你熟悉了之后，就会发现这是重复性问题，这也是为什么某些基于元数据的方法会明显导致系统规模更大的原因（下文中还会详细叙述）。如果集合中的对象不在集合拥有者范围之外使用，就可以使用依赖映射来简化映射。

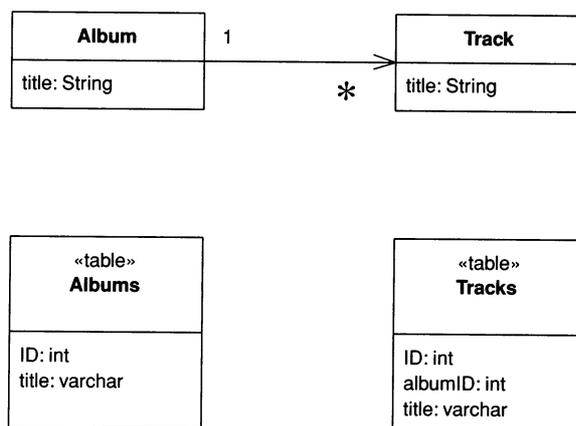


图3-6 使用外键映射来映射一个集合域

多对多的关系情况不同，这时在两边都存在集合。例如，一个人有很多种技能，并且对每种技能都要知道哪些人在使用它。关系数据库不能直接解决这种问题，因此需要使用关联表映射（见图3-7）来创建一个新的关系表，仅仅是为了解决多对多的关联问题。

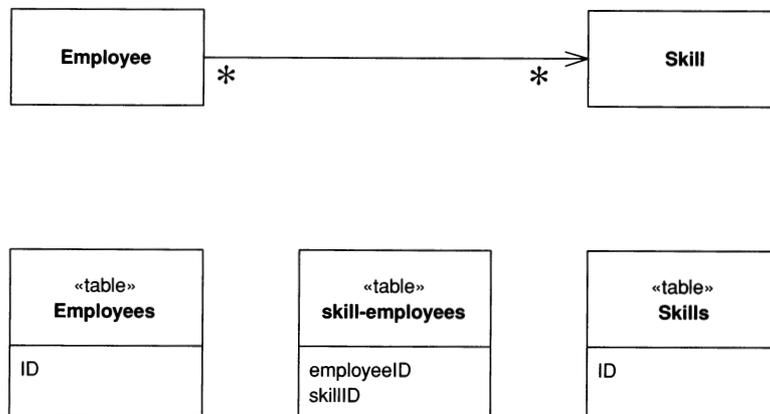


图3-7 使用关联表映射来映射一个多对多的关联

使用集合的时候，通常需要依赖集合中的排序。在面向对象语言中，通常使用数组或者列表这样的有序集合——实际上，这还可以简化测试的难度。然而，在保存到关系数据库中的时候想要维持一个绝对有序的集合是非常困难的。因此，有必要考虑使用无序集来存储集合。另一种方法是无论何时进行集合查询都要按照某种分类顺序，尽管有时候这样做代价很高。

在某些情况下，引用完整性会使得更新更加复杂。现代的系统允许把引用完整性检查延迟到交互结束的时候进行。如果有这个能力，没有道理不使用它。否则，数据库会在每次写操作的时候进行检查。在这种情况下，将不得不注意按照正确的顺序进行更新操作。具体如何操作不是本书讲述的范围，不过有一种技术是把更新操作进行拓扑分类。另一种就是确定好哪个表要按照哪种顺序写。这样有时能减少数据库中会导致事务过于频繁回滚的死锁问题。

标识域用来把对象间引用变为外键，但并不是所有的对象关系都需要用这种方法持久化。一些小的值对象（比如日期范围和钱）显然不应该描述成数据库中它们自己的表。取而代之的是，取出值对象中所有的域，并以嵌入值方式把它们嵌入到关联对象中。由于值对象含有值的语义，可以在每次执行读操作的时候创建它们，而无需担心标识映射的问题。把它们写到外面也非常容易，仅仅需要解除对象引用，并且把它的域写到它自己的表中即可。

还可以在更大规模上做类似工作，此时，可以通过序列化LOB将一组对象存储为表中的单个列。LOB代表“大对象”（Large Object），它可以是二进制（BLOB）的，也可以是文本（CLOB，字符大对象）的。将一组对象序列化为XML文档，对于层次化对象结构是显然可以采取的方法。这样就可以通过一次读取获得一整串互相关联的小对象。通常，数据库对于小的高度互连对象执行起来很低效，因为需要花费大量时间来进行许多很小的数据库调用。层次结构（诸如组织结构图表和材料账单），都能通过序列化LOB节省大量的数据库开销。

问题是SQL并不知道发生了什么事，这样就不能对应数据结构来建立可移植的查询。在这里，XML将再次发挥功效，它允许在SQL调用之中嵌入XPath查询表达式，尽管嵌入方法现在还没有标准化。因此，在不想对存储结构的内部进行查询的时候，最好使用序列化LOB。

通常，序列化LOB对于用来组成应用程序部分的相对独立对象群而言是最好的。但如果过多使用它，最终会把数据库弄得和事务文件系统差不多。

### 3.4.2 继承

前文中所述的“层次”，大多是“组合层次”，比如传统的关系系统难以处理的部件树。还有另外一种也会使得关系系统头疼的层次：通过继承关系相互连接的类层次。因为在SQL里面没有用于继承的标准方法，所以我们就必须再用到一个映射。对于任何继承结构，一般都有三种选择。可以为一个层次中的所有类建立一个表，即单表继承（见图3-8）；也可以为每个具体类建立一个表，即具体表继承（见图3-9）；或者为这个层次中每一个类建立一个表：类表继承（见图3-10）。

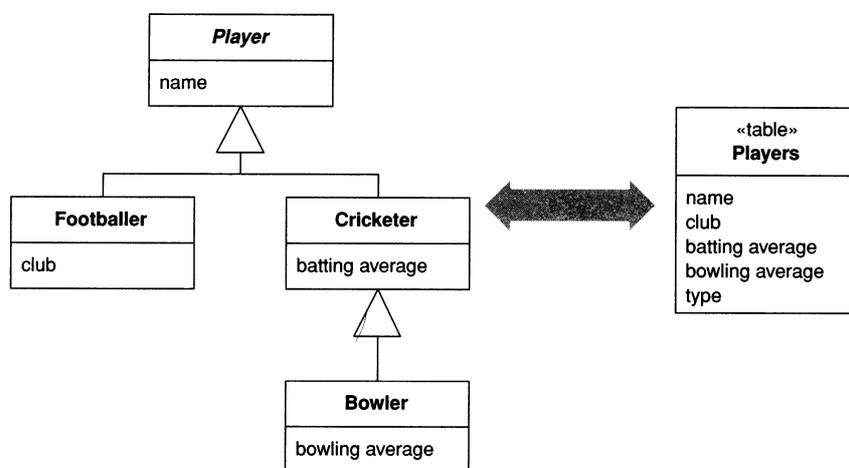


图3-8 单表继承为一个层次中的所有类建立一个表

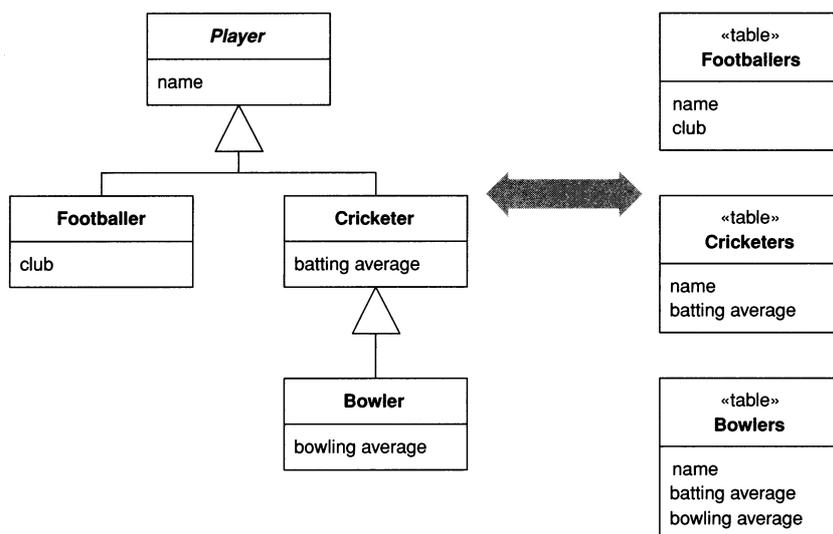


图3-9 具体表继承为一个层次中每个具体类建立一个表

在数据结构复制和访问速度之间必须进行权衡。类表继承是类和表之间最简单的关系，但

是它需要多个连接 (join) 操作来载入一个对象, 这样通常损失了性能。具体表继承避免了连接操作, 允许从一个表中取得一个对象, 但是改变起来比较困难。对超类的任何改变都不得不改变所有的表 (还有映射代码)。改变层次结构自身会带来更大的改变。缺乏超类表也能使主键管理十分可怕, 引用完整性也有问题, 尽管它能减少超类表中的锁争夺。而在某些数据库中, 单表继承最大的弊端是浪费了空间, 因为每一行都必须为每种可能的子类保留一些列, 这就导致很多空列。然而, 许多数据库都能很好地压缩浪费的表空间。单表继承的另一个问题在于它的大小将成为访问的瓶颈。它最大的好处是把所有的内容都放到一起, 这样修改起来很容易并且避免了连接操作。

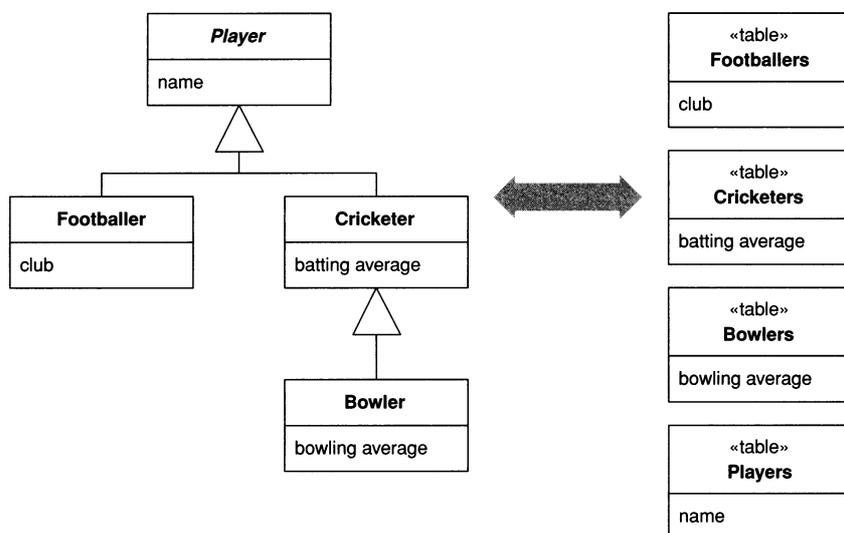


图3-10 类表继承为一个层次中每一个类建立一个表

这三个选择并不是相互排斥的, 在一个层次中可以混合几个模式。举个例子, 可以用单表继承把几个类放到一起并且使用类表继承来处理一些特殊情况。当然, 混合模式会增加复杂性。

在此, 我并不能明确地说这三个选择中哪一个是最好的。需要考虑自己的环境和偏好, 以及其他几个模式。我倾向于单表继承, 因为它比较容易实现, 并且易于重构。在需要的时候, 我喜欢用其他两个模式来帮助解决不可避免的不相关和无用列问题。最好的办法往往是直接和数据库管理员联系; 关于哪种访问方式对数据库更好, 他们通常都有好的建议。

上面的例子和模式中的例子都使用单继承。尽管多继承现在已经不那么时髦了, 大多数语言也越来越避免多继承, 但是使用接口的时候这个问题仍然在O/R映射中出现, 例如在Java和.Net中。这里, 模式并不特别描述这个主题, 但是本质上处理多继承时使用的是这三种继承模式的变种。单表继承把所有的超类和接口放到一个大表中, 类表继承为每个接口和超类建立一个独立的表, 具体表继承在每一个具体表中包含了所有的接口和超类。

### 3.5 建立映射

映射到关系数据库的时候, 一般会遇到三种情况:

- 自己选择数据库方案。
- 不得不映射到一个现有数据库方案，这个方案不能改变。
- 不得不映射到一个现有数据库方案，但这个方案是可以考虑改变的。

最简单的情况是自己选择数据库方案，并且不用迁就领域逻辑的复杂性，最终得到的结果是一个事务脚本或者表模块设计。在这种情况下，可以使用经典的数据库设计技术围绕数据来设计表。使用行数据入口或者表数据入口来把SQL从领域逻辑中剔除。

如果在使用领域模型，应该小心那种看上去像数据库设计的设计。在这种情况下，建立领域模型时不用理会数据库，这样可以简化领域逻辑。把数据库设计看作一种持久化对象数据的方法。数据映射器非常灵活，当然也带来了复杂性。如果数据库设计和领域模型同构有意义，可以考虑使用活动记录来代替。

尽管首先建立模型是一种合理的方法，但这个建议仅仅适用于短的迭代周期内。花费6个月的时间建立一个没有数据库的领域模型，并且决定一旦完成就持久化它，这是一件非常冒险的事情。危险在于，设计结果会因为迫切的性能问题而需要进行很多重构来修复。相反，应该为每一次迭代建造数据库，时间上不要超过6周并且适当地更短一些。这样就能更快和更持续地得到关于数据库交互实际上如何工作的反馈。针对特定任务，都应该首先考虑领域模型，但是这样做的时候，需要在数据库中集成领域模型的每一部分。

当已经存在一个数据库方案的时候，选择很相似，但过程却有点不同。对于简单的领域逻辑，可以建造行数据入口或者表数据入口类来模拟数据库，并在此之上构建领域逻辑。如果领域逻辑更复杂一些，将需要一个领域模型，而这个模型很可能和数据库设计不匹配。因此，应该逐步建立领域模型并包括数据映射器，把数据保存到现有的数据库中。

## 双向映射

有时，我会遇到这种情况：同样的一种数据却需要从不只一个数据源上取出来。可能有多个数据库保存相同的数据，只是由于某种复制和粘贴的重用会导致在数据库方案上的一些细微区别。（在这种情况下，头痛的数量与区别的数量成反比。）另一种可能是使用不同的存储机制，有时是数据库，有时是消息。也可能希望把类似的数据同时从XML消息、CICS事务和关系表中抽取出来。

最简单的选择是建立多个映射层，每个数据源一个。然而，如果数据非常类似的话，就会导致过多的复制。在这种情况下，可以考虑两步映射策略。第一步把数据从内存方案中转化到逻辑数据存储方案。设计逻辑数据存储方案是用来最大化数据源格式中的相似之处。第二步映射从逻辑数据存储方案到实际物理存储方案。第二步包含区别。

当有许多共同点时，额外的步骤仅仅补偿它们自身，因此你应该在有相似但又有十分头疼的不同的物理数据存储时使用它。把从逻辑数据存储到物理数据存储的映射看成是一个入口，并且使用任何映射技术从应用程序逻辑映射到逻辑数据存储。

## 3.6 使用元数据

在本书中，我绝大部分的例子都采用手写的代码。利用简单和重复性映射，这样会导致代

码简单和重复，而重复代码是设计上有问题的一个标志。可以通过委托和继承分解出通用行为，但还有一种更成熟的方法是使用元数据映射。

元数据映射基于把映射浓缩到元数据文件的方法。元数据文件详细描述数据库中的列如何映射到对象的域。这里的关键在于：一旦有了元数据，就可以通过代码生成或者反射编程来避免重复性代码。

使用元数据使我们可以用少量元数据表达很多含义。一行元数据可以像这样传递某些信息：

```
<field name = "customer" targetClass = "Customer", dbColumn = "custID", targetTable = "customers"  
lowerBound = "1" upperBound = "1" setter = "loadCustomer"/>
```

从这些信息可以定义读写代码，自动产生特别的连接操作，完成所有的SQL，加强关系的多样性，甚至可以做许多奇特的事情，比如在引用完整性存在的情况下计算写顺序等。这也就是为什么商业O/R映射工具倾向于使用元数据的原因。

当使用元数据映射的时候，有必要构造对内存对象的查询。一个查询对象允许根据内存对象和数据来构造查询，在这种方式下，开发者不需要知道SQL或关系数据库方案的细节。查询对象可以使用元数据映射把基于对象域的表达式翻译到对应的SQL。

一直使用这种方法就可以建立一个资源库，它能在很大程度上从视图隐藏数据库。任何到数据库的查询都可以做成资源库基础上的查询对象，并且开发者不用分辨对象是从内存还是从数据库中找回。资源库在有丰富领域模型的系统下工作良好。

尽管元数据有许多优点，我在本书里仍一直专注于手写的例子，因为我觉得它们首先比较容易理解。你一旦熟悉了这些模式，并且能用它们手写出自己的应用，就能领会到如何使用元数据来使问题更简单。

### 3.7 数据库连接

大多数数据库接口依赖于某些数据库连接对象，它们就好像应用程序代码和数据库之间的连接桥梁。通常，一个连接必须在能执行针对数据库的命令之前就打开。实际上，经常需要一个显式连接来建立和执行命令。在命令的整个执行过程中，该连接必须是打开的。查询的结果将返回一个记录集。某些接口提供无连接的记录集，这些记录集在连接关闭之后还能继续使用。其他的接口只提供连接的记录集，这意味着当记录集正在使用的时候连接必须一直打开。如果正在一个事务中运行，经常会把事务绑定在特定的连接上，当它运行的时候这个连接也必须一直打开。

在很多环境中，建立连接的开销相当大，这就需要建立一个连接池。在这种情况下，开发者向连接池请求一个连接并在完成以后释放，而无需即时创建和关闭。现在多数平台都会提供连接池，所以很少需要自己来实现连接池。如果必须自己实现连接池，首先就要检查连接池是不是真的能提高性能。越来越多的环境使我们可以更快地创建新的连接，因此不需要缓冲池。

提供连接池的环境经常把连接池放在一个类似创建新连接的接口后面。用这种方法，无需知道得到的是一个新创建连接还是从连接池里面分配的。那样很好，因为是否选择连接池被很好地封装起来。类似地，关闭一个连接可能并没有真关闭它，而只是把它交还给连接池以便别

人可以使用。在这里，我说的打开和关闭，也可以替换为从连接池获取和释放。

无论创建连接的代价是高还是低，连接都必须好好管理。因为它们是珍贵的资源，必须在使用完毕时立刻关闭。还有，如果正在进行一次事务，通常需要保证：在这次特定的事务中，每一个命令都是从同一个连接发出的。

最常见的建议就是用一个到连接池或者连接管理器的调用，显式得到一个连接，并且通过它来执行数据库命令。一旦执行完了，立刻把它关闭。这个建议带来两个问题：首先，保证在任何需要的地方都能得到一个连接；其次，保证不会在最后忘记关闭它。

为了保证在任何需要的地方都能得到一个连接，有两种选择。一是把这个连接作为一个直接的参数传递出去。这样做的问题在于这个连接会加入到各种各样的方法调用中去，而它的目的可能仅仅是要传递到某个在调用栈中第五层下的方法。当然，这种情况就带出了注册表。因为不希望多个线程使用同一个连接，所以将需要一个线程范围内的注册表。

如果你有我一半那么健忘，显式关闭就不是一个好主意。需要关闭的时候可能很容易就忘了。也不能通过每个命令来关闭连接，因为可能正在一次事务中运行，如果关闭了通常会导致事务回滚。

和连接差不多，内存也是一种在不用的时候需要释放的资源。现代环境提供了自动的内存管理和垃圾回收机制，因此保证连接关闭的一种方法就是使用垃圾回收器。在这种方式下，连接自身或者引用这个连接的对象会在垃圾回收期间关闭连接。这样做的好处是：它使用了和内存管理相同的机制，同样方便，也不陌生。这样做的问题是：连接的关闭只有当垃圾回收器实际收内存的时候才发生，可能离这个连接失去它最后一次引用的时间已经很久了。结果是，未被引用的连接可能会隔一段时间才被关闭。这究竟是不是一个问题要取决于特定的环境。

总的来说，我不喜欢依赖垃圾回收机制。其他的机制，甚至是显式关闭都会好一些。当然，垃圾回收机制在其他机制失败的情况下还是一种很好的后备。毕竟，让连接最终关闭总比让它们一直运行着强。

由于连接对于事务来说如此密不可分，因此管理它们的好方法就是把它们捆绑到事务中去。当开始一个事务的时候打开一个连接，当提交或者回滚的时候就关闭它。让事务知道它在使用什么样的连接，这样就可以完全不管连接而仅仅处理事务就可以了。因为事务的完成有一种可见的效果，所以即使是忘了提交，也很容易把它标识出来。工作单元很自然地适用于管理事务和连接。

如果要在事务之外处理一些事情，比如读取不可变数据，可以为每个命令使用一个新建连接。缓冲池可以处理任何创建短周期连接的问题。

如果你正在使用一个断接的记录集，可以打开一个连接，把数据放到记录集中并且在操纵记录集数据的时候关闭它。这样，数据使用结束以后，就可以打开一个新的连接和事务，把数据写出去。如果这样做，需要防止记录集正在使用的时候数据被修改。这个主题将会在同步控制中讨论。

连接管理细节描述的往往是数据库交互软件的特征，因此所使用的策略通常是由环境来决定的。

### 3.8 其他问题

细心的读者会注意到，一些代码例子用到了 `select * from` 的格式，而其他的查询语句使用了已命名的列。对于某些数据库驱动使用 `select *` 会带来一些严重的问题，如果加入了新的列或者某个列被重新排序就会失败。尽管更多的现代环境已经没有这个问题，但如果使用位置索引来从列中得到信息，那么选择 `select *` 也是不明智的，因为一个列的重新排序同样会导致代码失效。对列名索引使用 `select *` 是可以的，并且列名索引更容易读取；然而，列名索引可能会比较慢，尽管这也许并不会比SQL调用需要的时间有很大的不同。通常最好是测量一下比较保险。

如果使用列序号索引，需要保证对结果集的访问与SQL语句的定义十分接近，这样它们在列重新排序的时候就不会失去同步。因此，如果使用表数据入口，应该使用列名索引作为结果集，它们将被在入口上运行查找操作的每一段代码使用。通常为每一个使用的数据库映射结构提供简单的创建/读取/更新/删除的测试用例是值得的。这将有助于找到SQL与代码不同步的情况。

尽量使用已预先编译好的静态SQL，而不是每次都编译动态SQL。大多数平台都会提供SQL的预编译机制。一个重要的规则是避免使用字符串串联起多个SQL查询。

许多环境都提供把多个SQL查询打包到一次数据库调用的能力。对这些例子，我没有这么试过，不过这当然也是产品代码中可以使用的策略。如何做取决于平台。

对这些例子中的连接，我祈求它们可以跟在一个对“DB”对象的调用后，那就是一个注册表。如何得到一个连接将依赖于环境，因此你将用需要做的事情来代替它。我在除了关于并发之外的任何模式中都没有涉及事务。再次提醒，需要时刻考虑到环境的需要。

### 3.9 进一步阅读

对象 - 关系映射对大多数人来说是必须面对的，因此不要介意在这个主题上花费如此多的篇幅。令人奇怪的是，关于这个话题，还没有一本连贯、完整、能跟得上时代的书，这就是为什么我如此致力于研究这么一个有趣主题的原因。

好消息是关于数据库映射有许多很好的思想可以参考。重要的参考书有：[Brown and Whitenack]、[Ambler]、[Yoder]和[Keller and Coldewey]。希望读者能将资料作为本书模式的补充读物。