

iBATIS in Action

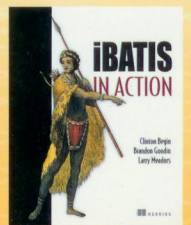
iBATIS 实战

Clinton Begin
[加] Brandon Goodin 著
Larry Meadors

叶俊 等 译

此电子文档由FerlyCreator制作，仅供学习研究之用，其它用途产生的一切后果自负！

- iBATIS之父权威著作
- 深入探讨比Hibernate更简单高效的数据库访问框架
- 同时支持Java和.NET平台



B057796



人民邮电出版社
POSTS & TELECOM PRESS

关于本书

实际上，iBATIS的使用非常简单。我曾经说过：“如果我竟要去写一本有关iBATIS的书，那肯定是多此一举了。”但是现在我却正在写这样的一本书。事实证明无论多么简单的知识，图书有时候才是学习它的最佳途径。近来有许多推测说，图书可能会被电子书或者安装在我们脑袋后面的插头所替代（通过这种插头可以在几秒钟内上传大量信息）。在我看来，这两种途径都让我觉得不太舒服。我喜欢图书，因为它便于携带且非常灵活。我可以随意地在书上写字，或者把某几页折起来，甚至可以把书脊撕开。再也没有什么比看到一本翻烂了的*iBATIS in Action*被乱丢在一间忙碌的开发人员办公室的地板上更能使我开心的了。预祝成功！

Clinton Begin

读者须知

我们希望尽量确保本书所涉及的iBATIS主题简单明了，但是有时抽象的概念需要更多的文字来描述。如果不关心相关理论，那么你可能会觉得某些章节有些冗长。但大多数章节都将迅速切入主题，遵循“原理，示例，实践”这样的结构。

本书假定读者已具备了一些相关知识。假设你已经了解Java，因此书中对Java不会着墨过多。在阅读本书之前，你应该至少已经阅读了若干本有关Java的书，也应当已经感受过使用JDBC所带来的痛苦，甚至可能体验过某些O/RM（对象关系映射）框架的缺陷。

同时，我们还假设你了解SQL。iBATIS是一个以SQL为中心的框架。iBATIS并不试图隐藏SQL，SQL语句在iBATIS框架中并非自动生成，而是完全在你自己的控制之下，因此你最好具备一些SQL开发的经验。

最后，你应该熟悉XML。尽管我们也希望有更好的解决方案，但无论如何XML对于iBATIS开发来说还是很有意义的。在创建像SQL语句这样的大块文本时，XML提供了比Java更好的支持（例如，Java不支持直接创建多行文本，必须使用字符串拼接运算），此外，XML支持丰富的标记（markup），也允许创建自定义的配置语法。iBATIS以后的版本可能会有其他的配置和开发方式，但是到目前为止还只有XML这种方式，因此你必须了解它。

读者对象

广大的开发人员是本书最主要的目标读者。建议开发人员跳过此部分，快速略读本书前几章

2 关于本书

中较高级较抽象的主题，然后直接从包含代码的部分开始深入阅读。我们期望你在阅读本书的同时进行一些编码实践。

曾经因使用O/RM受过“伤”且现在正在慢慢“康复”的O/RM用户应该会喜欢iBATIS和本书的。O/RM曾经希望能够成为一切问题的解决方案，但显然其结果不尽如人意。有太多的项目开始于O/RM却最终不免结束于SQL。iBATIS致力于在不引入新问题的前提下解决现有问题。我们并不反对使用O/RM，但是当你使用O/RM却总是遇到这样或那样的障碍时，换一种不同的方法也许会更加高效。

架构师可能会喜欢本书的高级介绍部分，该部分讨论只有iBATIS才具有的独特能力。尽管O/RM似乎被吹捧得无所不能，但架构师应该明白O/RM其实并非唯一的解决方案。他们应该学会如何画一个新的盒子，然后用几条新绘的线将它与O/RM盒子旁边的旧盒子连起来，同时也用线将这个新盒子与其他一些盒子连起来——当然上述过程应当确保不违背德米特法则（Law of Demeter）^①。

数据建模人员可能不愿再阅读本书了，但我们仍希望有人能鼓励他们阅读。创建iBATIS项目的部分原因正是因为数据库设计者们^②不愿意对他们设计的数据库进行一些适当的规范化^③（或者一些明智的反规范化）而使数据库设计总是存在这样或那样的问题。如果数据建模人员能够了解一下由于大型软件公司创建的大量遗留系统和ERP系统而带来的挑战，那么他们就应该更能体会到iBATIS的价值了。

其他适合阅读本书的人包括项目经理/主管、数据库管理员、质量保证员与测试员以及系统分析师。当然，非常欢迎大家购买本书，就算仅仅是为了本书超酷的封面。

导读

本书第一部分为iBATIS高级介绍部分，包括第1章和第2章，分别描述了iBATIS的设计理念和iBATIS的内涵。这两章为那些对iBATIS的基础感兴趣的人提供了相关的背景知识。如果你想获得有关iBATIS的比较实际的应用，并且立即开始使用它，可以直接跳到本书的第二部分。

第3章～第7章组成了本书的第二部分，该部分主要介绍基本的iBATIS应用。如果你打算利用iBATIS进行开发，该部分便是必读内容。第3章介绍如何安装iBATIS框架。第4章～第6章教你如何使用不同种类的SQL语句。作为第二部分的最后一章，第7章详细介绍了iBATIS对事务的支持，有助于确保在应用中能够正确地使用事务。第8章讨论动态SQL，它是iBATIS最重要的创新之一，要想在应用程序中引入复杂查询的能力，动态SQL必不可少。

第三部分开始讨论iBATIS的高级主题。第9章讨论有关数据高速缓存的高级主题。事实证明，高速缓存对于SQL映射框架来说是较为复杂的挑战之一，通过阅读本章你可以了解到高速缓存是

① 德米特法则通常也被表述为“只和离你最近的朋友进行交互”，用于软件设计则是说一个软件实体要尽可能地只与离它最近的实体进行通信。——译者注

② 即数据建模人员。——译者注

③ 所谓数据库规范化指的是使数据库满足2NF、3NF、BCNF或者其他范式的过程。——译者注

如何实现的。iBatis实际上包含了两个框架：SQL映射框架和DAO框架。DAO框架是完全独立的，但它的确是iBatis的一部分，因此有必要在此讨论它。第10章和第11章将详细讨论DAO框架。iBatis同时还是一个十分灵活的框架。只要有可能，你就总可以通过实现可插拔接口（pluggable interface）来将自定义的行为导入到框架中。第12章研究如何扩展iBatis以达到上述目的。

第四部分介绍iBatis的实际应用。第13章分析了多个关于如何才能更好地使用iBatis的最佳实践。第14章用一个综合性的Web应用程序JGameStore来总结全书。JGameStore应用程序的完整源代码可以从iBatis主页和Manning出版社的网站上获得。和本书中所有其他源代码一样，JGameStore得到Apache License 2.0的许可，因此你可以免费下载并随意使用它。

源代码约定及下载

代码清单中的源代码或者正文中出现的代码将以等宽字体（fixed-width font）显示。大多数代码清单中都有代码注解，用于突出重要概念。在某些情况下，还会出现一些带圈的数字标号，用于指示代码清单后面紧跟着的解释。

可以从出版社的网站下载本书中所有示例的源代码以及JGameStore应用程序的源代码，网址为www.manning.com/begin^①。

作者在线

购买*iBatis in Action*一书意味着你将同时获得免费访问Manning出版社所运营的专有Web论坛的权利，可以对本书做出评价，提出技术问题并且从作者和其他读者那里获得帮助。为访问该论坛并且订阅其内容，可以在Web浏览器中输入网址www.manning.com/begin。该网页提供了有关注册后如何登录论坛、能够获得何种类型的帮助以及在论坛上的行为准则等方面的信息。同时还提供了指向本书中示例的源代码、勘误表以及其他下载的链接。

Manning出版社对读者的承诺是，为读者之间以及读者和作者之间能够展开深入对话提供一个平台，但是并不承诺作者必须要参与到何种程度，作者对论坛的贡献是自愿的（并且是无偿的）。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！

只要本书已出版，作者在线论坛和以往讨论的存档都可以从出版社的网站上访问到。

关于书名

通过将介绍、综述和演示“如何做”的例子组合起来，Manning出版社的*In Action*系列丛书用来帮助读者学习和记忆。根据认知科学的研究，人们最容易记住的往往是他们在自发探索中发现的知识。

虽然Manning出版社并没有认知科学家，但是我们深信为了使学习变得持久，必须经历探索、

^① 网站上只能下载到JGameStore应用程序的源代码，没有书中示例的源代码。也可在图灵网站本书配套网页注册下载。——译者注

实践和（有趣地）复述所学的内容这几个阶段。只有在对新事物进行了积极的探索之后，人们才能理解和记住（即掌握）它们。人类在实践中学习。In Action系列丛书的指导思想的一个基本原则就是案例驱动。它鼓励读者去尝试新事物，实践新代码，探索新思想。

有关本书的书名，还有另一个更加世俗的理由：我们的读者都很忙。他们由于工作需要或为了解决某个问题而寻求图书的帮助。他们要求图书能够允许他们很容易地读进去，同时又能很容易地跳出来，并且仅仅当他们需要时，才学习他们真正需要的东西。他们需要能够帮助增强实战能力的图书。此系列图书就是为了这些读者而设计的。

封面插图

本书封面上的图像是“Geisiques”，非洲Horn岛Har-Geisa地区（位于今天的索马里）的一位居民。该插图来自于一本西班牙语的地区服饰集锦，它最早于1799年在马德里出版。

此书的扉页上写着：

Coleccion general de los Trages que usan actualmente todas las Naciones del Mundo desubierto, dibujados y grabados con la mayor exactitud por R.M.V.A.R. Obra muy util y en special para los que tienen la del viajero universal.

我们尽可能地照字面意思翻译如下：

世界上所有已知民族当前使用的服饰集锦，由R.M.V.A.R.精心设计和印刷。此书对于那些环球旅游爱好者尤其有用。

虽然我们对于此画的设计者、雕刻师以及那些手工为它上色的工人一无所知，但他们工作的“精确度”却很明显地表现在该画中。“Geisiques”仅仅是这本丰富多彩的集锦的众多图片中的一幅。这些多姿多彩的图片很生动地说明了200多年前世界上不同城镇和地区的独特性和个性。在那个时期，相距仅仅几十英里的两个地区的人就可以通过服饰习俗准确地识别出对方属于哪个地区。这本集锦生动地再现了那个时期（以及除了当前我们这个通信极其发达的时期之外其他所有的历史时期）的隔离感和距离感。

从那以后，服饰习俗发生了巨大变化，那个时期如此丰富多彩的地区差异也逐渐消褪了。现在甚至连区分两个不同大陆的居民都往往非常困难。也许我们应该尽量乐观地来看待这种变化，文化和视觉上的多样性已经转变成了更加丰富多彩的个人生活，或者更加多样和有趣的精神生活和技术生活。

我们使用此集锦中的图片作为本书的封面，它生动地再现了两个世纪之前不同地区丰富多彩的生活。Manning出版社的编辑们使用这种方式来赞美计算机行业的创造力和主观能动性，当然，还包括其中的乐趣。

目 录

第一部分 介绍

第1章 iBATIS 的理念.....2

1.1 一个结合了所有优秀思想的

混合型解决方案.....2

1.1.1 探索 iBATIS 的根源.....3

1.1.2 理解 iBATIS 的优势.....7

1.2 iBATIS 适合应用在何处.....10

1.2.1 业务对象模型.....11

1.2.2 表现层.....11

1.2.3 业务逻辑层.....12

1.2.4 持久层.....13

1.2.5 关系数据库.....15

1.3 使用不同类型的数据库.....17

1.3.1 应用程序数据库.....17

1.3.2 企业数据库.....18

1.3.3 私有数据库.....19

1.3.4 遗留数据库.....20

1.4 iBATIS 如何解决数据库的常见问题.....20

1.4.1 所有权与控制.....20

1.4.2 被多个分散的系统访问.....21

1.4.3 复杂的键和关系.....21

1.4.4 数据模型的去规范化或过度规范化.....22

1.4.5 瘦数据模型.....23

1.5 小结.....24

第2章 iBATIS 是什么.....26

2.1 映射 SQL 语句.....27

2.2 iBATIS 如何工作.....29

2.2.1 iBATIS 之于小型、简单系统.....30

2.2.2 iBATIS 之于大型、企业级系统.....31

2.3 为何使用 iBATIS.....31

2.3.1 简单性.....32

2.3.2 生产效率.....32

2.3.3 性能.....32

2.3.4 关注点分离.....33

2.3.5 明确分工.....33

2.3.6 可移植性: Java、.NET 及其他.....33

2.3.7 开源和诚实.....33

2.4 何时不该使用 iBATIS.....34

2.4.1 当永远拥有完全控制权时.....34

2.4.2 当应用程序需要完全动态的 SQL 时.....34

2.4.3 当没有使用关系数据库时.....35

2.4.4 当 iBATIS 不起作用时.....35

2.5 5 分钟内用 iBATIS 创建应用程序.....35

2.5.1 安装数据库.....36

2.5.2 编写代码.....36

2.5.3 配置 iBATIS (预览).....37

2.5.4 构建应用程序.....38

2.5.5 运行应用程序.....39

2.6 iBATIS 未来的发展方向.....40

2.6.1 Apache 软件基金会.....40

2.6.2 更简单、更小且依赖性更少.....40

2.6.3 更多的扩展点和插件.....41

2.6.4 支持更多的平台和语言.....41

2.7 小结.....41

第二部分 iBATIS 基础知识

第3章 安装和配置 iBATIS.....44

3.1 获得一份 iBATIS 发布.....45

3.1.1 二进制发布.....45

3.1.2 从源代码构建.....45

3.2 发布中包含的内容.....47

2 目 录

3.3 依赖性.....	48	4.5 小结.....	81
3.3.1 针对延迟加载的字节码增强.....	48	第5章 执行非查询语句.....	82
3.3.2 Jakarta Commons 数据库连接池.....	49	5.1 更新数据的基本方法.....	82
3.3.3 分布式高速缓存.....	49	5.1.1 用于非查询 SQL 语句 的 SqlMap API.....	82
3.4 将 iBATIS 添加到应用程序中.....	49	5.1.2 非查询已映射语句.....	83
3.4.1 在独立应用程序中使用 iBATIS.....	50	5.2 插入数据.....	84
3.4.2 在 Web 应用程序中使用 iBATIS.....	50	5.2.1 使用内联参数映射.....	84
3.5 iBATIS 和 JDBC.....	51	5.2.2 使用外部参数映射.....	85
3.5.1 释放 JDBC 资源.....	51	5.2.3 自动生成的键.....	86
3.5.2 SQL 注入.....	51	5.3 更新和删除数据.....	88
3.5.3 降低复杂度.....	52	5.3.1 处理并发更新.....	88
3.6 配置 iBATIS (续).....	53	5.3.2 更新或删除子记录.....	89
3.6.1 SQL Map 配置文件.....	54	5.4 运行批量更新.....	90
3.6.2 <properties>元素.....	55	5.5 使用存储过程.....	91
3.6.3 <settings>元素.....	56	5.5.1 优缺点分析.....	92
3.6.4 <typeAlias>元素.....	58	5.5.2 IN、OUT 和 INOUT 参数.....	93
3.6.5 <transactionManager>元素.....	60	5.6 小结.....	95
3.6.6 <typeHandler>元素.....	61	第6章 使用高级查询技术.....	96
3.6.7 <sqlMap>元素.....	61	6.1 在 iBATIS 中使用 XML.....	96
3.7 小结.....	62	6.1.1 XML 参数.....	96
第4章 使用已映射语句.....	63	6.1.2 XML 结果.....	98
4.1 从基础开始.....	63	6.2 用已映射语句关联对象.....	101
4.1.1 创建 JavaBean.....	64	6.2.1 复杂集合.....	101
4.1.2 SqlMap API.....	66	6.2.2 延迟加载.....	104
4.1.3 已映射语句的类型.....	67	6.2.3 避免 N+1 查询问题.....	105
4.2 使用<select>已映射语句.....	70	6.3 继承.....	107
4.2.1 使用内联参数 (用#做占位符).....	70	6.4 其他用途.....	109
4.2.2 使用内联参数 (用\$做占位符).....	71	6.4.1 使用语句类型和 DDL.....	109
4.2.3 SQL 注入简介.....	72	6.4.2 处理超大型数据集.....	109
4.2.4 自动结果映射.....	73	6.5 小结.....	115
4.2.5 联结相关数据.....	74	第7章 事务.....	116
4.3 映射参数.....	75	7.1 事务是什么.....	116
4.3.1 外部参数映射.....	75	7.1.1 一个简单的银行转账示例.....	116
4.3.2 再论内联参数映射.....	76	7.1.2 理解事务的特性.....	118
4.3.3 基本类型参数.....	78	7.2 自动事务.....	120
4.3.4 JavaBean 参数和 Map 参数.....	78	7.3 局部事务.....	121
4.4 使用内联结果映射和显式结果映射.....	78		
4.4.1 基本类型结果.....	79		
4.4.2 JavaBean 结果和 Map 结果.....	81		

7.4 全局事务	122
7.4.1 使用主动或被动事务	123
7.4.2 开始、提交以及结束事务	124
7.4.3 我是否需要全局事务	124
7.5 定制事务	125
7.6 事务划界	126
7.6.1 将事务在表现层划界	128
7.6.2 将事务在持久层划界	128
7.6.3 将事务在业务逻辑层划界	128
7.7 小结	129
第8章 使用动态 SQL	130
8.1 处理动态 WHERE 子句条件	130
8.2 熟悉动态标签	132
8.2.1 <dynamic>标签	134
8.2.2 二元标签	135
8.2.3 一元标签	136
8.2.4 参数标签	137
8.2.5 <iterate>标签	138
8.3 一个简单而完整的示例	139
8.3.1 定义如何检索和显示数据	140
8.3.2 确定将涉及哪些数据库结构	140
8.3.3 以静态格式编写 SQL	141
8.3.4 将动态 SQL 标签应用到静态 SQL 上	141
8.4 高级动态 SQL 技术	142
8.4.1 定义结果数据	142
8.4.2 定义所需的输入	143
8.4.3 以静态格式编写 SQL	144
8.4.4 将动态 SQL 标签应用到静态 SQL 上	145
8.5 动态 SQL 的其他替代方案	147
8.5.1 使用 Java 代码	147
8.5.2 使用存储过程	150
8.5.3 同 iBATIS 相比较	152
8.6 动态 SQL 的未来	152
8.6.1 简化的条件标签	152
8.6.2 表达式语言	153
8.7 小结	153

第三部分 真实世界中的 iBATIS

第9章 使用高速缓存提高性能	156
9.1 一个简单的 iBATIS 高速缓存示例	156
9.2 iBATIS 高速缓存的理念	157
9.3 理解高速缓存模型	158
9.3.1 type 属性	158
9.3.2 readOnly 属性	159
9.3.3 serialize 属性	159
9.3.4 联合使用 readOnly 属性和 serialize 属性	159
9.4 如何使用高速缓存模型中的标签	160
9.4.1 高速缓存的清除	160
9.4.2 设置高速缓存模型实现的特性	163
9.5 高速缓存模型的类型	163
9.5.1 MEMORY	163
9.5.2 LRU	164
9.5.3 FIFO	165
9.5.4 OSCACHE	166
9.5.5 你自己的高速缓存模型	166
9.6 确定高速缓存策略	166
9.6.1 高速缓存只读的长效数据	167
9.6.2 高速缓存可读写数据	169
9.6.3 高速缓存旧日的静态数据	170
9.7 小结	172
第10章 iBATIS 数据访问对象	173
10.1 隐藏实现细节	173
10.1.1 为什么要分离	174
10.1.2 一个简单示例	175
10.2 配置 DAO	177
10.2.1 <properties>元素	177
10.2.2 <context>元素	178
10.2.3 <transactionManager>元素	178
10.2.4 DAO 元素	182
10.3 配置技巧	183
10.3.1 多个服务器	183
10.3.2 多种数据库方言	184
10.3.3 运行时配置更改	185
10.4 基于 SQL Map 的 DAO 实现示例	185

4 目 录

10.4.1	配置 iBATIS DAO	186
10.4.2	创建 DaoManager 实例	187
10.4.3	定义事务管理器	187
10.4.4	加载映射	188
10.4.5	DAO 实现编码	191
10.5	小结	193
第 11 章	DAO 使用进阶	194
11.1	不是基于 SQLMap 的 DAO 实现	194
11.1.1	Hibernate 版本的 DAO 实现	194
11.1.2	JDBC 版本的 DAO 实现	199
11.2	为其他数据源使用 DAO 模式	203
11.2.1	示例：为 LDAP 使用 DAO	203
11.2.2	示例：为 Web 服务使用 DAO	208
11.3	使用 Spring DAO	209
11.3.1	编写代码	209
11.3.2	为什么使用 Spring	
	代替 iBATIS	211
11.4	创建自己的 DAO 层	211
11.4.1	从实现中分离出接口	212
11.4.2	创建一个工厂以解耦	212
11.5	小结	214
第 12 章	扩展 iBATIS	215
12.1	理解可插拔组件的设计	215
12.2	使用自定义类型处理器	217
12.2.1	实现自定义类型处理器	217
12.2.2	创建 TypeHandlerCallback	218
12.2.3	注册 TypeHandlerCallback 以供使用	221
12.3	使用 CacheController	222
12.3.1	创建 CacheController	223
12.3.2	CacheController 的放入、 获取以及清除操作	223
12.3.3	注册 CacheController 以供使用	224
12.4	配置 iBATIS 不支持的 DataSource	224
12.5	定制事务管理	225
12.5.1	理解 TransactionConfig 接口	226
12.5.2	理解 Transaction 接口	227
12.6	小结	228

第四部分 iBATIS 使用秘诀

第 13 章	iBATIS 最佳实践	230
13.1	iBATIS 中的单元测试	230
13.1.1	对映射层进行单元测试	231
13.1.2	对 DAO 进行单元测试	233
13.1.3	对 DAO 的消费层进行单元测试	235
13.2	管理 iBATIS 配置文件	237
13.2.1	将其保存在类路径上	237
13.2.2	集中放置文件	238
13.2.3	主要按返回类型来 组织映射文件	239
13.3	命名规范	239
13.3.1	语句的命名	239
13.3.2	参数映射的命名	239
13.3.3	结果映射的命名	240
13.3.4	XML 文件的命名	240
13.4	Bean、map 还是 XML	240
13.4.1	JavaBean	241
13.4.2	Map	241
13.4.3	XML	241
13.4.4	基本类型	241
13.5	小结	241
第 14 章	综合案例研究	243
14.1	设计理念	243
14.1.1	账户	243
14.1.2	目录	244
14.1.3	购物车	244
14.1.4	订单	244
14.2	选择具体的实现技术	244
14.2.1	表现层	244
14.2.2	服务层	244
14.2.3	持久层	245
14.3	调整 Struts：使用 BeanAction	245
14.3.1	BaseBean	246
14.3.2	BeanAction	246
14.3.3	ActionContext	246
14.4	JGameStore 工程结构	247
14.4.1	src 文件夹	247

14.4.2	test 文件夹	248	14.9	小结	263
14.4.3	web 文件夹	248	附录 A	iBatis.NET 快速入门	264
14.4.4	build 文件夹	248	A.1	比较 iBatis 和 iBatis.NET	264
14.4.5	devlib 文件夹	248	A.1.1	为何 Java 开发人员应该 关心 iBatis.NET	264
14.4.6	lib 文件夹	249	A.1.2	为何 .NET 开发人员应该 关心 iBatis.NET	265
14.5	配置 web.xml 文件	249	A.1.3	主要区别是什么	265
14.6	设置表现层	251	A.1.4	相似之处又在哪里	265
14.6.1	第一步	251	A.2	使用 iBatis.NET	265
14.6.2	使用表现层 bean	253	A.2.1	DLL 和依赖性	265
14.7	编写服务层代码	257	A.2.2	XML 配置文件	266
14.7.1	配置 dao.xml 文件	258	A.2.3	配置 API	267
14.7.2	事务划界	259	A.2.4	SQL 映射文件	267
14.8	编写 DAO	260	A.3	到哪里去查找更多的信息	269
14.8.1	SQLMap 配置	260			
14.8.2	SQLMap 文件	261			
14.8.3	接口和实现	262			

Part 1

第一部分

介 绍

本书从对 iBATIS 的高级介绍开始。接下来的两章将讨论 iBATIS 的理念，正是这种理念的不同使得 iBATIS 有别于其他的持久化方案。对 Java 来说，有许多种可选的持久化方案，到底在何时使用哪一种方案，可能是一个非常值得探讨的问题。通过阅读本部分的内容，你应该能够明白 iBATIS 背后的理念和它的价值，当然你也就知道应该在何时何地使用它了。

本 部 分 内 容

- 第 1 章 iBATIS 的理念
- 第 2 章 iBATIS 是什么

第 1 章

iBATIS的理念

本章内容

- iBATIS的历史
- 理解iBATIS
- 数据库的类型

SQL (Structured Query Language, 结构化查询语言) 已经存在很长一段时间了。自从Edgar F. Codd第一次提出“数据可以被规范化为一组相互关联的表”这样的思想以来, 已经超过35年了。从那时起, IT公司就投入了几十亿美元来开发RDBMS (relational database management system, 关系数据库管理系统)。很少有哪一种软件技术敢声称自己像关系数据库和SQL那样经受住了时间的考验。确实, 经过了那么长时间, 关系技术背后仍然有巨大的推动力, 并且它是世界上最大的软件公司的基础性成果。所有的迹象都表明SQL仍然会继续使用下去, 至少30年。

iBATIS的建立正是基于这样的思想: 关系数据库和SQL仍然很有价值, 在整个产业范围内对SQL投资仍然是一个非常好的主意。我们可能都曾有过这样的经历, 应用程序的源代码 (即使经历了很多版本) 随着时间的流逝最终还是过时了, 但它的数据库甚至是SQL本身却仍然很有价值。在某些情况下我们也会看到一个应用程序已经被用其他的语句重写了, 但背后的SQL和数据库却基本上保持不变。

正是基于这些原因, iBATIS并不试图去隐藏SQL或者避免使用SQL。相反, 正是iBATIS这个持久层框架广泛使用了SQL, 它使得SQL更容易使用、更容易集成到现代的面向对象软件中。最近, 有传言说数据库和SQL与我们的面向对象的设计理念不符, 但事实并不一定是这样。iBATIS能帮助我们避免这种问题。

在本章中, 我们将学习到iBATIS的历史和基本原理, 并讨论影响其创建的动力。

1.1 一个结合了所有优秀思想的混合型解决方案

在现实世界中, 混合型解决方案随处可见。将两个看上去似乎相悖的思想在“中间处”巧妙结合, 被证明是一种有效的方法, 它往往能恰到好处地解决问题, 在某些情况下甚至会导致一种

新兴行业的诞生。汽车工业就是一个最典型的例子，运输工具大部分的设计革新都来自于对不同思想的混合。将小轿车（car）与大篷货车（cargo van）结合最终就形成了我们现在的家用房车（family minivan）。把卡车与越野车（all-terrain vehicle, ATV）相结合，于是就有了现代城市人身份的象征——运动型多功能车（sport utility vehicle, SUV）。将高速汽车（hotrod）与旅行汽车相结合，于是就有了驾驶起来很舒服的家用小轿车。在汽油引擎旁边安装一个电力引擎，于是目前北美很多环境污染问题就都可以迎刃而解了。

混合型解决方案在IT领域同样被证明是非常有效的。iBATIS就是这样一个混合型的持久层解决方案。在过去的时间中，我们已经开发了大量的方法来使应用程序能够执行SQL，以便操纵其背后的数据库。iBATIS汲取了这些方法中的优秀思想，成为一个独特的持久层解决方案。让我们首先快速浏览一下这些方法。

1.1.1 探索 iBATIS 的根源

iBATIS从目前流行的关系数据库访问方法中吸收了大量的优秀特征和思想，并找出其中的协同增效作用。图1-1展示了iBATIS框架是如何吸收我们在多年使用不同方式进行数据库集成的开发过程中所学到的知识，并将其中最优秀的思想结合起来，形成这个混合型解决方案的。

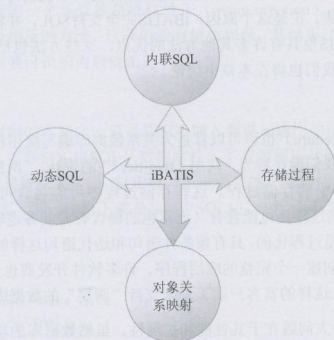


图1-1 iBATIS为简化开发过程所融合的一些思想

接下来的几节将讨论这些不同的数据库访问方法，以及iBATIS从每种方法中汲取的优秀思想。

1. SQL

iBATIS的核心是SQL。从根本上说，所有的关系数据库都支持SQL，并将它作为与数据库交互的主要方式。SQL是一种简单的、非过程化的语言，用于操纵数据库，其实SQL包含两种语言。

4 第1章 iBATIS 的理念

第一种就是数据定义语言 (Data Definition Language, DDL), 其中包含像CREATE、DROP以及ALTER这样的语句。这些语句用于定义数据库数据及其设计, 包括表 (table)、列 (column)、索引 (index)、约束 (constraint)、过程 (procedure) 以及外键关系 (foreign key relationship)。iBATIS并不直接支持DDL。虽然许多人的确通过iBATIS成功地执行了DDL, 但我们不推荐这样做, 因为DDL通常应该由某个数据库管理小组拥有并控制, 应用程序的开发人员无权操纵它。

SQL所包含的第二种语言就是数据操纵语句 (Data Manipulation Language, DML)。DML包括像SELECT、INSERT、UPDATE以及DELETE这样的用于直接操纵数据的语句。最初, SQL被设计为一种简单得足以让终端用户 (end user) 可以使用的语言。按照这种设计思想, 图像用户界面就完全不需要了, 甚至连应用程序也不需要了。当然, 这种情况得回溯到当年那个永远是“黑屏上闪着光标”的时代了, 在那个时代我们才可以对终端用户报以更多的期望。

今非昔比, 数据库已经变得太复杂了, 不能指望终端用户通过SQL来直接操纵数据库了。你能想象这样的情况吗: 将一大堆SQL语句交给会计部门, 然后对他说, “就这些了, 查找一下BSHEET表你就能找到所需要的信息。”这简直是不可想了。

仅仅SQL本身已经再也不可能作为终端用户的一个有效接口了, 但对于开发人员来说SQL仍然是一个非常强大的工具。SQL是唯一具有完整的数据库访问功能的方法, 所有其他的工具都只不过是该完整功能的一个子集。正是这个原因, iBATIS完全支持SQL, 并将它作为关系数据库访问的主要方式。同时, iBATIS也具有许多其他方法的优点, 这些方法包括存储过程和对象/关系映射工具, 有关此方面内容我们也将在本章中讨论。

2. 老式存储过程

存储过程 (stored procedure) 也许可以算是为关系数据库编写应用程序的最古老的方式了。许多遗留系统使用的都是我们现在称为“两层 (two-tier)”的设计。“两层设计”包括一个富客户端界面来直接调用数据库中的存储过程。这些存储过程中可能包含可用于操纵当前数据库的SQL。除了SQL, 这些存储过程还可能含有 (通常也的确含有) 业务逻辑。与SQL不同, 编写这些存储过程时使用的语言是过程化的, 具有条件语句和迭代语句这样的流程控制语句。事实上, 仅仅使用存储过程就可以创建一个完整的应用程序。许多软件开发商也开发了像Oracle Forms、PowerBuilder和Visual Basic这样的富客户端工具来支持“两层”的数据库应用程序的开发。

“两层应用程序”的最大问题在于其性能和扩展性。虽然数据库的功能的确非常强大, 但它们往往未必是处理成百上千甚至上百万个用户访问的最佳选择。对于现代Web应用程序来说, 对扩展性的需求并不少见。数据库在并发访问量、硬件资源以及网络套接字等方面的限制, 将使得这种“两层”架构在需要扩大规模时面临失败的危险。此外, “两层应用程序”的部署也是一场噩梦。除了通常的富客户端部署问题外, 复杂的运行时数据库引擎也需要部署到客户端机器上去。

3. 现代存储过程

在某些圈子中存储过程仍被认为是三层 (three-tier) 和N层 (N-tier) 应用程序 (例如Web应

用程序)的最佳实践。但现在存储过程通常被当作来自中层(middle tier)的远程过程调用(remote procedure call, RPC),并且许多性能方面的约束也可以通过建立间接池和数据库资源管理等方式解决了。在现代的面向对象的应用程序中,存储过程仍然是实现完整的数据访问层的一个行之有效的选择。存储过程在性能方面的确具有优势,因为它们总是能够比任何其他解决方案更快地完成数据库中的数据操作。但是,还有比性能更需要关注的问题。

将业务逻辑放在存储过程中通常都被认为是一个糟糕的设计。最主要的原因就在于存储过程的开发很难符合现代应用程序的架构。它们难以编写、难以测试、同时也难以部署。更糟糕的是,现代企业中数据库通常由其他的数据库管理小组拥有,并且受到最为严格的变更控制的保护。它们通常无法快速变更以适应现代软件开发方法学的需要。此外,要用存储过程来实现完整的业务逻辑,其自身也存在某些限制。如果业务逻辑需要访问其他的系统、资源或用户界面,存储过程很可能就无法处理所有这些逻辑了。现代应用程序非常复杂,需要更加通用的语言,而不是像存储过程这样仅仅优化了数据操纵能力的专用语言。为了解决这个问题,有些开发商在它们的数据库引擎中嵌入了像Java这样的更加强大的语言,来编写更加强壮的存储过程。但这其根本没有解决问题,它只会使得应用程序与数据库的边界变得更加模糊,并为数据库管理员带来新的负担:他们现在需要开始担心数据库中的Java和C#了。对于解决问题,这实在是一个错误的工具。

软件开发中经常出现的一个场景就是矫枉过正(overcorrection)。当发现问题时,尝试的第一个解决方案往往是完全相反的方法。这非但没有解决问题,其结果往往是引入等量的完全不同的新问题。例如我们下面要讨论的内联SQL。

4. 内联SQL

解决存储过程固有限制的方法之一就是SQL嵌入到更加通用的语言中去。与存储过程将业务逻辑移入数据库相反,内联SQL将SQL从数据库移入了应用程序代码。这就使得SQL语句可以直接与语言进行交互。从某种意义上说,SQL成为了该语言的一个特性。有很多语言具有这种“特性”,包括COBOL、C、甚至Java。以下就是Java中SQLJ的一个示例:

```
String name;  
Date hiredate;  
  
#sql {  
    SELECT emp_name, hire_date  
    INTO :name, :hiredate  
    FROM employee  
    WHERE emp_num = 28959  
};
```

内联SQL的确非常优雅,因为它做到了与语言的紧密结合。本地语言变量可以直接传递给SQL作为参数,SQL执行结果可直接赋值给类似的变量。某种意义上,SQL的确成为了该语言的一个特性。

但不幸的是,内联SQL并没有被广泛接受,一些重大问题的存在最终还是使得它无法开花结果。首先,SQL还不标准,它存在许多扩展版本,而每个版本又都只适用于某个特定的数据库。

SQL语言的分裂使得要实现一个既完备又可在各个数据库平台间移植的内联SQL分析器非常困难。内联SQL的第二个问题就是它往往并不是真的实现为语言的一个特性，而是使用一个预编译器先将内联SQL“翻译”为当前语言中的相应代码。这就给像IDE（integrated development environment，集成开发环境）这样的工具带来了问题，因为这些工具可能需要预先解释代码以启用像语法突出显示和代码完成这样的高级特性。而包含内联SQL的代码如果没有预编译器可能甚至连编译都无法通过，这种依赖性使得人们不得不担心他们的代码在将来的可维护性。

解决内联SQL这些问题的方案之一就是SQL从语言级移除，代之以应用程序中的某种数据结构（如字符串）来表现它。这种方法就是我们通常所说的动态SQL（dynamic SQL）。

5. 动态SQL

动态SQL通过避免使用预编译器来解决内联SQL存在的一些问题。作为替代，SQL被表现为一种字符串类型的数据，可以像现代语言中所有其他的字符数据一样地操纵它。因为SQL被表示为一种字符串类型，所以它就不能像内联SQL一样直接与语言进行交互了。因此，动态SQL的实现需要一组强壮的API，用来设置SQL参数，获取结果数据，等等。

动态SQL的优点就在于其具有的灵活性。SQL可以在运行时基于不同的参数或动态的应用程序功能来构建。例如，一个“按样例查询（query-by-example）”的Web表单就可能允许用户动态选择需要搜索的字段，以及希望搜索的数据。这就要求SQL语句的WHERE子句能够动态改变，使用动态SQL就非常容易做到这一点。

动态SQL是目前从现代编程语言访问关系数据库的方法中最流行的。大多数这样的现代编程语言都包括用于进行数据库访问的标准API。相信Java开发人员和.NET开发人员对各自语言（分别是JDBC和ADO.NET）中的标准API都很熟悉，这些标准SQL API通常都非常强壮，为开发人员提供了巨大的灵活性。以下就是Java语言中动态SQL的一个简单示例：

```
String name;  
Date hiredate;  
String sql = "SELECT emp_name, hire_date"  
            + " FROM employee WHERE emp_num = ? ";  
Connection conn = dataSource.getConnection();  
PreparedStatement ps = conn.prepareStatement(sql);  
ps.setInt(1, 28959);  
ResultSet rs = ps.executeQuery();  
while (rs.next()) {  
    name = rs.getString("emp_name");  
    hiredate = rs.getDate("hire_date");  
}  
rs.close();  
conn.close();
```

应该包裹在一个try-catch
块中

毫无疑问，动态SQL没有内联SQL那么优雅，甚至还比不上存储过程（以上代码中已经省略了异常处理但还是非常复杂）。这些API通常都非常复杂且冗长，就像我们从之前的示例中所看到的一样。使用这样的框架通常会带来大量的代码，而这些代码往往又具有重复性。此外，SQL语

句本身也常常因为太长而无法在一行代码中写完。这时就不得不将该SQL字符串打散为多个子串然后通过拼接(concatenat)操作组合起来。代码中的字符串拼接操作使得SQL的可读性大大降低，给维护和使用都带来了更多的困难。

那么，既然将SQL放在数据库中作为存储过程，或者放在语言中作为内联SQL，或者放在应用程序中作为一种数据结构都不是最合适的，那么到底应该如何“处置”它呢？我们回避这个问题。在现代面向对象的应用程序中，与关系数据库交互的一个最引人注目的解决方案就是对象/关系映射工具的使用。

6. O/RM

O/RM（对象/关系映射）就是被设计为用来简化对象持久化工作的，它通过将SQL完全从开发人员的职责中排除来达到这个目的。在O/RM中，SQL是生成的。一些工具在应用程序构建或编译时静态生成SQL，其他一些则在运行时动态生成。SQL是基于应用程序中的类与关系数据库表之间的映射关系而生成的。除了不用写SQL语句，使用O/RM工具的API通常也比典型的SQL API要简单得多。O/RM其实并不是一个新概念，它的历史几乎与面向对象编程语言的历史一样悠久。只是近年来的许多进展才使得O/RM又成为一个引人注目的持久化方法。

现代O/RM工具所做的决不仅仅是简单地产生几条SQL语句。它们提供了一套完整的持久化架构，可以使你的整个应用程序从中受益。任何一个优秀的O/RM工具都提供了事务管理功能，包括可用于处理本地事务以及分布式事务的非常简单的API。O/RM工具通常也会为处理各种不同类型的数据提供许多高速缓存策略以避免不必要的数据库访问。O/RM工具可以减少数据库访问的另一种方式就是数据延迟加载技术。延迟加载使得对数据的获取操作被推迟到绝对必要时，即直到它们确实需要被使用的那一刻。

虽然具有那么多优秀的特性，但O/RM工具仍然不是一颗“银弹”，它并非适用于所有的场景。O/RM工具是基于一些假设和规则的。其中最普遍的一个假设就是数据库被恰当地规范化了。正如我们将在1.4节中讨论的，那些最大的最有价值的数据库往往并没有被很好地规范化。这就会给映射带来许多麻烦，甚至需要绕些弯路，或者在设计时对效率做些折衷。没有哪一个对象/关系解决方案可以支持每一种数据库的每一个特性、每一种能力以及设计上固有的缺陷。正如我们之前说过的，SQL不是一个可靠的标准。基于这些原因，每一个O/RM工具都只是任何一个特定数据库所具有的全部功能的一个子集。

现在讨论混合型解决方案。

1.1.2 理解 iBATIS 的优势

iBATIS是一个混合型解决方案。它汲取了所有这些解决方案中最有价值的思想并将它们融会贯通。表1-1总结了iBATIS从我们之前讨论的那些方案中所汲取的思想。

表1-1 iBATIS提供的与其他解决方案相同的优点

方 案	相同的优点	解决的问题
存储过程	iBATIS对SQL进行了封装和外部化,使SQL从你的应用程序代码中分离出来。iBATIS具有与存储过程相似的API,但iBATIS的这些API是面向对象的。iBATIS也完全支持对存储过程的直接调用	业务逻辑从数据库中分离出来,应用程序更容易部署与测试,也具有更好的可移植性
内联SQL	iBATIS允许SQL以其最自然的方式书写,没有字符串拼接,没有参数“设置”,没有结果“获取”	iBATIS对应用程序代码没有任何影响。不需要任何预编译器,并且你能够完全访问SQL的所有特性,而不仅仅是一个子集
动态SQL	iBATIS提供了若干特性以支持基于参数的动态构建查询。不需要“查询构建工具”这样的API	iBATIS不要求SQL被写成一堆字符串的拼接,中间还夹杂着应用程序的代码
O/RM	iBATIS支持许多与O/RM工具一样的特性,例如延迟加载、连接抓取、高速缓存、运行时代码生成以及继承	iBATIS可用于任意数据类型与任意对象模型的组合。它对这两者中的任何一个的设计没有任何约束和要求

现在你已经理解iBATIS的根本思想了,接下来的小节将讨论iBATIS持久层所具有的两个最重要的特性:外部化SQL和封装SQL。这两个概念展现了iBATIS框架的核心价值,并启用了这个框架所能实现的许多高级特性。

1. 外部化的SQL

我们可以从过去十年来的软件开发中学习到的经验就是“总是将一个大系统设计为多个子系统,每个子系统的功能都相对集中”。应该尽可能地将那些需要由不同的开发角色处理的任务分离开,例如,用户界面设计人员专职设计用户界面,应用程序开发人员专职开发应用程序,数据库管理人员专职管理数据库。即使只有一个人来扮演所有这些角色,这种分离也能够使你拥有一个漂亮的分层设计,使你总是能专注于系统的某个特定部分。如果将SQL嵌入到Java源代码中,那么这些SQL对于数据库管理人员或者对于使用相同数据库的.NET开发人员来说就没有任何用处。外部化将SQL从应用程序的源代码中分离出来,从而使得两者都更加清晰。这样做就保证了SQL语句与任何特定的语言或平台都相对地独立。大多数现代开发语言都将SQL表现为一个字符串类型,这就使得那些较长的SQL语句需要使用字符串拼接操作。考虑下面这个简单的SQL语句:

```
SELECT
PRODUCTID,
NAME,
DESCRIPTION,
CATEGORY
FROM PRODUCT
WHERE CATEGORY = ?
```

当在现代编程语言(例如Java)中嵌入一个这样的字符串数据类型时,这个原本很简单的SQL语句就变成了一个“乱七八糟”的字符串,难以管理了。

```
String s = "SELECT"
+ " PRODUCTID,"
+ " NAME,"
+ " DESCRIPTION,"
+ " CATEGORY"
+ " FROM PRODUCT"
+ " WHERE CATEGORY = ?";
```

如果忘记在FROM子句前留一个空格，那么就会引起SQL错误。很容易想象，如果你有一条复杂的SQL语句，那将会引起多大的麻烦。

这时iBATIS的一个主要优点就体现出来了：使用iBATIS你就拥有按照最自然的方式书写SQL的能力。以下代码让你对经iBATIS映射后的SQL语句先有个感性认识：

```
SELECT
  PRODUCTID,
  NAME,
  DESCRIPTION,
  CATEGORY
FROM PRODUCT
WHERE CATEGORY = #categoryId#
```

注意该SQL不论是结构还是简单性都没有变化。它与之前的SQL最大的不同之处就在于参数的格式变为了#categoryId#，这个格式通常是特定于语言的细节，iBATIS却使得它易移植且更具可读性。

现在我们已经将SQL从源代码中分离了出来，并且放在了一个可以更自然地使用的地方，接下来只要将它重新与我们的软件建立起连接，就可以执行它了。

2. 封装化的SQL

计算机编程领域一个最古老的概念就是模块化。在一个过程化的应用程序中，代码可能被分成许多文件、函数和过程。在面向对象的应用程序中，代码常常被组织为类和方法。封装(encapsulation)是模块化的一种形式，它不仅将代码组织到一个内聚的模块中，而且还将实现细节隐藏了起来而仅仅向调用该类的代码暴露出它的接口。

封装的概念稍微扩展也可以应用到持久层中。可以通过定义SQL的输入和输出(例如它的界面)来封装它，这样应用程序的其他部分就不需要知道具体的SQL语句了。如果你是一个面向对象的软件开发人员，那么可以像理解接口与实现的分离一样来理解这种封装。如果你是一个SQL开发人员，那么可以像理解存储过程对SQL的隐藏一样来理解这种封装。

iBATIS使用XML(eXtensible Markup Language, 可扩展标记语言)来封装SQL。所以选择XML是因为它具有很好的跨平台性，并且得到了行业内的广泛采用，还有就是XML可能会像SQL一样被长期使用下去，其生命可能会比其他任何一种语言或文件格式都长。iBATIS使用XML映射SQL语句的输入和输出。大多数的SQL语句都会有一个或更多的参数，会产生一堆表格化的数据作为结果。或者说，结果被组织成了一系列的行和列。iBATIS允许你很容易地将输入输出参数映射为某些对象的特性(property)。如下例所示：

```
<select id="categoryById"
  parameterClass="string" resultClass="category">
  SELECT CATEGORYID, NAME, DESCRIPTION
  FROM CATEGORY
  WHERE CATEGORYID = #categoryId#
</select>
```

注意包围在SQL语句周围的XML元素。这就是对SQL的封装。这个简单的<select>元素定义了我们的语句的名字、输入参数类型以及输出结果类型。对于一个面向对象的软件开发人员来说，这就像是一个方法的签名。

通过对SQL的外部化和封装，我们获得了简单性和一致性。关于iBATIS的API如何使用以及SQL映射的语法细节，我们将在接下来的第2章中讨论。在此之前，我们首先应该知道，在架构应用程序中，iBATIS到底应该用在哪里，这很重要。

1.2 iBATIS 适合应用在何处

几乎所有结构良好的软件都使用了分层设计。分层设计将一个应用程序根据技术职能分为几个内聚的部分，从而将某种特定技术或接口的实现细节与其他部分分离开来。分层设计可以用任何一种强壮的编程语言（3GL/4GL）来实现。图1-2给出了一个典型的分层策略的高级视图，该图对于许多商业应用程序都是有用的。

可将图1-2中的箭头读作“依赖于”或“使用”。这种分层设计其实来源于德米特法则，该法则的一种表述方式就是，“每一层都应该只对那些与自己紧密相关的层有有限的了解。”

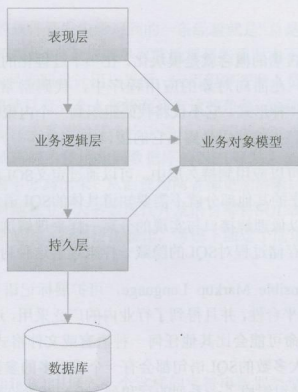


图1-2 遵循德米特法则的典型分层策略

每一层都只与自己的直接下层“交流”。这就保证了依赖流（dependency flow）只有一个方向，从而避免了那种在没有分层设计的应用程序中非常普遍的“意大利面”式的代码^①。

① 原文是“spaghetti code”，直译为“意大利面条式的代码”，可意译为“混乱代码”。——编者注

iBATIS是一个持久层框架。持久层位于应用程序的业务逻辑层和数据库之间。这种分离非常重要，它保证了持久化策略不会混入业务逻辑代码，反之亦然。这种分离的好处就在于你的代码将更加容易维护，因为它允许对象模型的演化不依赖于数据库设计。

虽然iBATIS关注的主要是持久层，但理解应用程序的整体架构中的每个层仍然是很重要的。虽然通过分层设计可以将关注点分离，从而将对任何一种特定实现的依赖都降到最低（甚至没有），但如果你认为这样就可以不管其他层的存在，不顾与其他层的交互，那就太天真了。不论应用程序设计得多么完美，你也必须明白层与层之间一定存在某些间接的行为关联。以下各节将讨论应用程序的各个层以及iBATIS与这些层的关系。

1.2.1 业务对象模型

业务对象是一个应用程序的所有其他部分的基础。它是问题域在面向对象方法学中的一种表现，因此构成业务对象模型的各个类有时也被称为领域类（domain class）。所有其他层都使用业务对象模型来表现数据和执行某些特定的业务逻辑功能。

应用程序的设计者们通常都是从业务对象模型的设计开始的。即使从较高的层次上看，业务对象模型中定义的各个类也都来源于我们的问题域中的名词。举个例子，在一个网上书店应用程序中，其业务对象模型就很可能包括一个名为Genre^①的类，该类的实例包括Science Fiction（科幻）、Mystery（神秘）和Children's（儿童）。该模型可能还包括一个名为Book的类，其实例包括The Long Walk、The Firm和Curious George。随着应用程序越来越复杂，类代表更抽象的概念（如InvoiceLineItem^②）。

业务对象模型类中当然也可能包括一些逻辑，但它们决不能包含任何用于访问其他层（特别是表现层和持久层）的代码。此外，这个业务对象模型绝对不能依赖于其他任何一层。其他层可以使用业务对象模型——但永远不能反过来。

像iBATIS这样的持久层通常会使用业务逻辑对象来表现存储在数据库中的数据。业务对象模型中的这些领域类将成为持久化方法（persistence method）的参数和返回值。也正是因为这个原因，这些类有时也被称为数据传输类（data transfer object, DTO）。虽然数据传输并不是它们唯一的作用，但从持久化框架的角度看，这个名字还是相当合适的。

1.2.2 表现层

表现层负责向最终用户展示应用程序的控制方式以及数据。它还要负责所有信息的布局和格式。今天，商业应用程序最流行的表现方式应该算是Web前端了，它使用HTML和JavaScript并通过Web浏览器来满足用户的界面外观需求。

① “类型”之意。——译者注

② 意为清单上的一个条目。——译者注

Web应用程序的优势包括跨平台兼容性、易部署和可扩展。amazon.com就是Web应用程序的一个极好的例子, 它允许你在线购书。这就是Web应用程序的一个绝佳应用, 因为不可能要求用户为了买一本书而去下载一个应用程序。

当需要高级的用户控件或者复杂的数据操纵时, Web应用程序通常就无法胜任了。在这些情况下, 使用本机操作系统小部件(如tab、table、tree view和嵌入式对象)的富客户端就体现出了它的优势。富客户端允许一个强大得多的用户界面, 但它往往更难部署, 且要达到与Web应用程序相同级别的性能 and 安全性需要开发人员花费更多的精力。富客户端技术的例子包括Java的Swing和.NET的WinForms。

最近, Web应用程序和富客户端这两个概念被混合了起来, 形成了所谓的“混合型客户端”, “混合型客户端”试图同时获得Web应用程序和富客户端两者的优点。一些非常小且使用了一些高级控件的富客户端可能通过Web浏览器被悄悄地下载到用户的桌面。这个混合型的富客户端不包含任何业务逻辑, 甚至可能连用户界面的布局都不是内建好的。相反, 应用程序的界面外观以及可用的业务功能都是通过一个Web服务, 或者把XML用作客户端与服务端接口的Web应用程序来配置的。这种方式唯一的缺点就是开发和部署这样的应用程序需要额外的软件。举个例子, Adobe Flex和来自Laszlo Systems的Laszlo可都是基于Macromedia的Flash这个浏览器插件的。

接下来当然就有了所谓混合型表现层的典型案例, 即Ajax。该术语由Jesse James Garrett提出, 它曾经是Asynchronous JavaScript and XML(异步JavaScript和XML)的首字母缩写, 但现在所有人都意识到它既不需要异步, 也并非只能使用XML, 所以现在Ajax代表的仅仅是“一种基于Web的富客户端界面, 由大量非常巧妙的JavaScript所驱动”。Ajax是使用旧技术构建内容丰富且交互性强的用户界面的一种新方法。Google很好地诠释了Ajax技术, 例如在Gmail、Google Maps和Google Calendar这样的应用程序中就充分利用了这种技术。

iBATIS既可用于Web应用程序和富客户端应用程序, 也可用于混合型应用程序。虽然表现层通常不会直接与持久化框架“交流”, 但用户界面设计时的某些决定还是会影响到持久层的需求。举个例子, 考虑一个Web应用程序, 它需要处理一个包含5 000条记录的大型列表。我们不可能需要同时显示出所有这5 000条记录, 而且如果我们不是立即需要使用它们, 那么同时从数据库中加载这5 000条记录也不是什么好主意。一个更好的方案可能是一次只加载和显示10条记录。这样的话, 持久层就需要能够在返回数据的数量上允许一定的灵活性, 甚至提供选择和获取我们希望的10条记录的能力。这样就可以避免不必要的对象创建和数据获取, 减少应用程序的网络访问量和内存需求, 进而提高应用程序的性能。iBATIS允许只查询某个特定范围内的数据, 这样的特性就可以帮助我们达到以上这些目的。

1.2.3 业务逻辑层

应用程序的业务逻辑层描述了应用程序所能提供的“粗粒度”的服务。正是这个原因, 业务逻辑层中的类有时也被称为服务类。从较高的层次来看, 任何人都应该能看懂业务逻辑层中

的类和方法进而明白系统到底要做什么。举个例子，在一个银行应用程序中，业务逻辑层可能就会包含名为TellerService的类，其中包括像openAccount()、deposit()、withdrawal()和getBalance()这样的方法。这些都是非常大的功能，涉及复杂的数据库交互甚至可能是与其他系统的交互。这些方法太重了，不适合放在领域类中，否则代码很可能马上就会变得耦合、并且通常会难以管理。解决方案就是将这些粗粒度的业务方法从与它们相关的业务对象模型中分离出来。这种业务逻辑类与对象模型类的分离有时也被称为“名词与动词的分离 (noun-verb separation)”。

纯面向对象论者可能会说，这样的设计不够面向对象，将业务方法直接放在相关的领域类中才更加面向对象。不论哪种方式更面向对象，能将关注点分离才是一个更好的设计选择。其中的主要原因还是在于业务方法通常都非常复杂。它们通常都涉及不止一个类，处理不止一种基础组件 (infrastructural component)，这些基础组件可能包括数据库、消息队列和其他系统。更重要的是，一个业务功能往往涉及许多领域类，那么该方法到底应该属于哪个类呢，的确难以决定。也正是由于这些原因，粗粒度的业务功能最好还是实现为业务逻辑层中某个类的方法。

不要害怕把那些粒度更细的业务逻辑放到相关的领域类中。业务逻辑层中那些粗粒度的服务方法可以自由地调用内建在领域类中的细粒度的纯逻辑方法。

在我们的分层架构中，业务逻辑层是持久层服务的消费者。它调用持久层的方法来获取数据和修改数据。业务逻辑层也是事务定界的最佳场所，因为其中定义的粗粒度业务功能可以供许多不同的用户界面使用，甚至还可能被像Web服务这样的一些其他接口使用。关于事务定界，我们将在第8章中详细讨论这个话题。

1.2.4 持久层

持久层是适合使用iBATIS的地方，因此它是本书的焦点。在面向对象的系统中，持久层主要关注对象（或者更精确地说应该是存储在那些对象中的数据）的存取。在企业应用程序中持久层通常用关系数据库系统来存储数据，虽然某些情况下其他持久的数据结构或者介质也可能使用。如某些系统就可能使用简单的以逗号分隔数据的平板文件 (flat file) 或XML文件。考虑到企业应用程序的持久化策略往往具有异质性，因此持久层需要关注的第二个问题就是抽象。持久层应该隐藏关于数据如何被存储以及如何被取出的所有细节。这样的细节决不能暴露给应用程序的其他层。

为更好地理解这些关注点及其如何被管理，我们将持久层又分为3层：抽象层、持久化框架以及驱动程序/接口层，如图1-3中中间偏下的那一部分。

让我们再仔细研究一下持久层细分后得到的这3层吧。

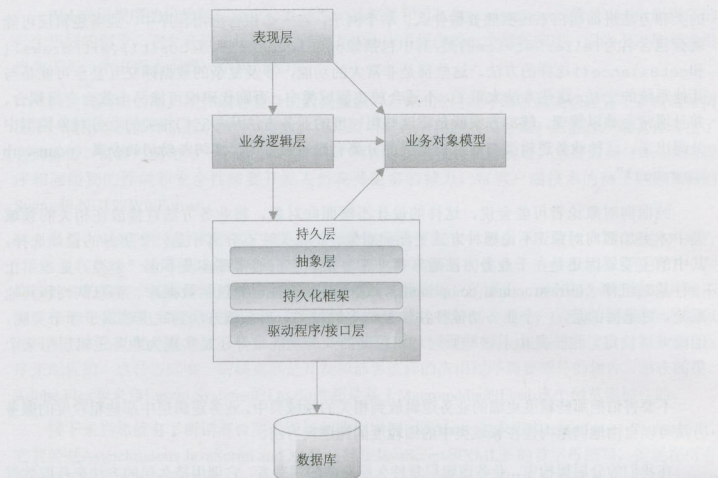


图1-3 放大持久层以展示内部的分层设计

1. 抽象层

抽象层的目的在于为持久层提供一致且有意义的接口。它是一组类和方法的集合，这些类和方法是持久层实现细节的facade^①。抽象层中的方法不能使用特定于实现的参数，也不能返回特定实现专用的类或抛出特定实现专用的异常。一旦合适的抽象层准备就绪，整个持久化方法（包括持久化API以及存储设施）的改变就不再涉及该抽象层，也不能引起其他任何依赖层的改变。有许多模式可用于帮助实现一个合适的抽象层，其中最常用的就是DAO模式。包括iBATIS在内的许多框架都为你实现了这个模式。第11章将讨论iBATIS的DAO框架。

2. 持久化框架

持久化框架负责与数据库驱动程序（或接口）的交互。持久化框架会提供用于存储、获取、更新、查找以及管理数据的方法。与抽象层不同，持久化框架通常只针对一类存储设施。例如，你可能会找到专用于处理XML文件的数据存储的持久化API。但是，对大多数现代企业应用程序来说，关系数据库通常是存储设施的首选。大多数流行的编程语言都带有用于访问关系数据库的标准API。JDBC是Java应用程序访问数据库的标准框架，而ADO.NET则是.NET应用程序访问数

^① 即“外观模式”，用于为一个子系统定义一组一致的接口。——译者注

数据库的标准持久化框架。这些标准API能够用于任何目的，因此它们的实现非常完备，只是使用起来非常冗长和繁琐。基于这些原因，人们在标准API的基础上又创建了许多框架，使之功能更加专用，因此也更加强大。iBATIS就是这样的一个持久化框架，它专用于处理任何类型的关系数据库，且以一种一致的方式同时支持Java和.NET。

3. 驱动程序/接口层

存储设施可以简单得像以一个逗号分隔数据的平板文件，也可以复杂得像一个价值几百万美元的企业数据库服务器。但不论是哪种情况，总有一个软件驱动程序在底层与存储设施通信以交换数据。一些驱动程序，如本地文件系统驱动程序，虽然在功能上是通用的，但与平台相关。你也许永远也不会看到文件I/O驱动程序，但可以肯定的是它一定存在。另一方面，数据库驱动程序总是非常复杂，并且在实现、大小和行为上存在较大的不同。因此就需要由持久化框架来与这些数据库驱动程序通信，从而将它们之间的不同简化并降低到最低点。考虑到iBATIS只支持关系数据库，因此本书将只关注关系数据库。

1.2.5 关系数据库

iBATIS的存在就是为了简化对关系数据库的访问。数据库的确非常复杂，要正确地使用它们需要做很多的工作。数据库负责管理数据和修改数据。我们使用数据库而不简简单单地使用一个平板文件的原因就在于数据库为我们提供了许多好处，特别是在数据完整性、性能以及安全性方面。

1. 数据完整性

数据完整性可能是数据库提供的最重要的好处了，因为没有数据完整性，其他一切就没有意义了。如果数据不是一致的、可靠的以及正确的，那它们又有多少价值呢。数据库通过使用强数据类型，强制约束，以及使用事务从而实现数据完整性的要求。

数据库是强类型的，即一旦创建了一个数据库表，那么它的每个列就被指定为只能存储某种特定类型的数据。数据库管理系统保证了存储在表中的数据的类型相对于列是正确的。举个例子，某个表可能将其某列定义为VARCHAR(25) NOT NULL。这就保证了存储在该列中的数据是字符且长度不会超过25，另外定义中的NOT NULL部分保证了该数据是必需的，也即你必须为该列提供一个值。

除了强数据类型，还可以对数据库表使用一些其他约束。这些约束通常在范围上更广，因为它们处理的往往不只是某一列。约束通常涉及对表中多条记录（甚至是多个表）的验证。例如UNIQUE约束就要保证表中指定列的值不重复。再如外键（FOREIGN KEY）约束，这种约束保证表中某列的取值一定来自于关联表中的相关列。外键约束是用于描述表间关系的，因此它对于关系数据库设计以及数据完整性非常重要，不可或缺。

数据库维护数据完整性最重要的方式之一就是使用事务。大多数业务功能都需要使用很多种

类型的数据, 它们往往来自不同的数据库。通常这些数据会以某种方式相互关联, 因此需要一致更新。使用事务, 数据库管理系统可以保证所有的相关数据以一种一致的方式被更新。更重要的是, 事务允许系统的多个用户同时更新数据而不造成冲突。关于事务还有许多知识需要了解, 我们将在第8章中详细讨论它们。

2. 性能

关系数据库可以帮助我们获得使用平板文件时很难获得的高性能。也就是说, 数据库性能并不是免费的, 要想获得高性能你需要大量的时间和专家经验。数据库性能可被分为3个关键因素: 设计、软件调整 (software tuning) 以及硬件。

要提高数据库的性能, 首先要考虑的因素就是设计。一个糟糕的关系数据库设计带来的低效可能用再多的软件调整和额外的硬件也无法纠正。糟糕的设计可能造成死锁、指数级的关系运算或者是几百万条记录的数据表扫描。正确的设计非常重要, 因此我们将在1.3节中对它进行详细讨论。

对大型数据库来说, 软件调整是提高性能要考虑的第二重要的因素。调整关系数据库系统需要有相应的对我们使用的特定RDBMS软件有过专门训练且富有经验的专业人士。虽然RDBMS软件的某些特征号称是跨越各个不同的产品的, 但通常每种产品都有其精妙而细微的差别, 因此需要针对该特定软件的专业人士。性能调整可以带来某些巨大的好处。例如仅仅是数据库索引的调整就可以将原本需要执行几分钟的复杂查询变为只需几秒钟。RDBMS中可调整的部分有很多, 如高速缓存、文件管理、各种索引算法, 甚至还可以考虑操作系统。同一个RDBMS软件在不同的操作系统上表现出的行为可能也是不同的, 因此针对不同的操作系统也要进行不同的调整。不用多说了, 反正调整数据库软件需要付出大量的努力。数据库到底应该如何调整已经超出了本书的讨论范围, 但是知道软件调整也是一种非常重要的提高数据库性能的因素还是很重要的。请与DBA (即数据库管理员) 好好协调。

大型关系数据库系统通常对计算机硬件的要求都比较高。也正是因为这个原因, 许多公司里性能最强大的服务器往往都是数据库服务器。在许多公司里, 数据库就是他们的“宇宙中心”, 因此针对数据库的硬件投入大量的资金也就不足为奇了。快速磁盘阵列、I/O控制器、硬件高速缓存以及网络接口, 所有这一切对于大型数据库管理系统的性能来说都是至关重要的。有了这些, 你就再不能将硬件作为糟糕的数据库设计的借口或者作为RDBMS调整的替代了。硬件不应该拿来解决性能问题——它应当用来满足性能需求。关于硬件更深入的讨论同样也超出了本书的范围, 但当你使用一个大型数据库系统时, 考虑到这个因素还是很重要的。也还请你与DBA好好协调。

3. 安全性

关系数据库系统也提供了额外的安全性。我们在日常工作中使用的大量数据都是保密的。近年来, 个人隐私越来越受到关注, 安全性已经成为所有数据都需具备的一个基本要求。基于这个原因, 甚至一个人的全名这样的简单信息也可以被认为是保密的, 因为它是潜在的唯一标识信息。

其他的信息（例如，社会保险号码和信用卡账号）需要像强加密这样的更高级别的安全保护。大多数商务性质的关系数据库都包括许多先进的安全特性，允许更细粒度的安全性以及数据加密。每个数据库都有其独特的安全需求。最重要的是你要理解它们，因为应用程序代码绝不能削弱数据库的安全策略。

不同的数据库有不同级别的数据完整性、性能和安全性。通常来说，数据库的大小、数据的价值以及数据库相关人员的多少会决定这些级别。下一节将研究不同类型的数据库。

1.3 使用不同类型的数据库

并非所有的数据库都如此复杂，需要使用昂贵的数据库管理系统以及企业级的硬件。一些数据库其实非常小，足以运行在一台老式的PC机上。所有的数据库都是不一样的。它们有各自不同的需求和不同的挑战。iBATIS可以帮助你使用几乎任何类型的关系数据库，但了解你使用的数据库究竟是哪种类型通常也是非常重要的。

数据库的划分更多是依据它与其他系统的关系，而不是依据其设计和大小。但数据库的设计和大小又往往取决于它与其他系统的关系。另一个会影响数据库设计和大小的因素就是数据库的年龄。随着时间的推移，数据库往往会以不同的方式变化，而应用程序这种变化的方式又常常不尽理想。在本节中，我们将讨论4种类型的数据库：应用程序数据库、企业数据库、私有数据库和遗留数据库。

1.3.1 应用程序数据库

应用程序数据库往往是最小、最简单、也最易于使用的数据库。这种数据库往往是我们这些开发人员通常不介意使用甚至非常乐意使用的。应用程序数据库通常与我们的应用程序处于同一个项目中，两者一齐设计和实现。正是因为这个原因，应用程序数据库的设计往往存在非常大的自由度，它也最有可能与我们的特定应用程序完美匹配。应用程序数据库的对外影响是最小的，因为它通常只有一两个对外接口。第一个接口连接到我们的应用程序，而第二个接口可能就是一个简单的报表框架或报表工具（例如Crystal Reports）。图1-4从较高的层次展示了一个应用程序数据库以及它与其他系统的关系。

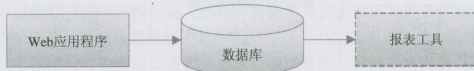


图1-4 应用程序数据库与其他系统的关系

应用程序数据库有时小到可以与应用程序直接部署在同一台服务器上。同样，使用应用程序数据库对硬件资源的要求也更加自由。

有了小型应用程序数据库，通常更容易说服公司使用那些更便宜的开源RDBMS解决方案，例如MySQL或PostgreSQL，而不需要花费大量的钱去购买Oracle或SQL Server。有些应用程序甚

至可能使用一种内嵌的应用程序数据库^①，这种数据库与应用程序运行在相同的虚拟环境中，因此连独立的SQL文件都可以不需要。

iBATIS作为一个持久化框架能很好地支持应用程序数据库。因为iBATIS非常简单，一支开发团队可以非常迅速地创建一个新的应用程序。对于简单的数据库来说，甚至可以通过使用随RDBMS自带的管理工具从数据库模式中生成SQL。同样，也有可自动产生所有iBATIS SQL映射文件的工具可用。

1.3.2 企业数据库

企业数据库比应用程序数据库更大，其外部影响也更大。它们与其他系统之间存在更多的关系，包括依赖关系和被依赖关系。这些关系可能是Web应用程序与报表工具之间的，但也很有可能的是与其他的复杂系统和数据库的接口。在企业数据库中，不仅仅存在远比应用程序数据库多得多的外部接口，而且这些接口的作用方式也大不相同。一些接口可能是用于每晚批量加载数据的接口，其他的则可能是实时事务处理接口。由于这些原因，企业数据库本身可能实际上就是由不止一个数据库组成的。图1-5从较高的层次描绘了一个企业数据库的例子。

企业数据库对于其设计和使用都加强了许多限制。对于数据完整性、性能以及安全性，企业数据库往往比应用程序数据库要考虑更多的因素。基于这个原因，企业数据库为分离关注点和分隔需求，往往会分裂为多个部分。如果试图仅创建单个的数据库来满足企业系统的所有需求，其代价将非常昂贵并且复杂，或者根本就不实际甚至不可能。

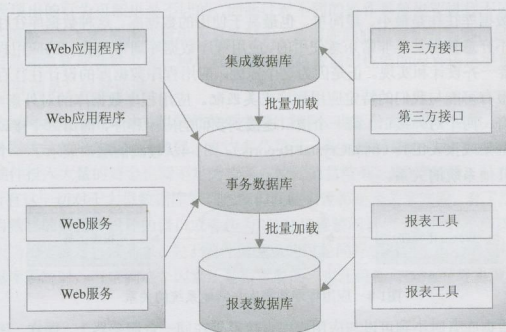


图1-5 企业数据库架构的一个示例

① 如hsqldb。——译者注

如图1-5所描绘的示例，需求按照横向的非业务需求被划分。具体来说，这些数据库被划分为集成数据库（integration database）、在线事务数据库（online transactional database）以及报表数据库（reporting database）。集成数据库和报表数据库都通过批量加载（batch load）与事务数据库交互，这也就暗示了这个系统并不要求报表必须是最新的，另外事务数据库也只要要求能够周期性地从第三方系统中更新数据。这样设计的好处就在于我们能够大大减轻事务数据库的负担，从而使一个较为简单的设计成为可能。一般来说，要设计一个同时对集成性、事务性和报表都是高效的数据库是不实际的。对以上的每一个性能都有一些设计模式可保证最佳的性能和设计。但有时我们的需求是近似于实时的集成和报表功能。对这样的需求之前的设计就无法满足了。这时你可能会发现需要将企业数据库按照业务功能纵向划分。

不管企业数据库如何设计，要理解应用程序数据库与企业数据库的不同点都非常容易。理解你的环境有哪些特别的限制，以保证你的应用程序总能高效地使用数据库并且与其他使用当前数据库的应用程序相安无事，这非常重要。

iBATIS在企业数据库环境中工作得非常出色。它具有的一些特征使得它成为了与复杂的数据库设计和大型数据集协调工作的理想工具。iBATIS用于多数据库时同样非常出色，因为它从来没有假设某个类型的对象必须仅来自一个数据库。它同样支持在单个事务中涉及多数据库的复杂事务。此外，iBATIS不仅对在线事务系统非常有用，同样对实现报表系统和集成系统也非常有用。

1.3.3 私有数据库

如果你从事软件开发工作有了一段时间的话，那么肯定听过关于“自己动手还是花钱购买”的争论。该争论是说，针对一个业务问题，我们是应该自己动手构建自己的解决方案呢，还是应该花钱购买一个声称已经解决了此问题的软件包。通常付出的代价是差不多的（否则也就没什么好争论的了），但真正的权衡其实在于“自己动手所要花费的时间”与“花钱购买来的软件包和我们要解决的问题的契合度”。自己构建的软件当然能恰到好处地符合业务需求，但实现它确实需要更多的时间。购买软件包当然非常迅速，但有时又不一定能满足我们所有的需求。基于这个原因，当决定购买软件包时，我们往往会在买回以后对它进行改造，修改它的私有数据库，扩展它所遗漏的特征，这样就可以两全其美了。

我们其实可以好好讨论一下深入别人的私有数据库并修改它是一件多么麻烦的事，但在此我想告诉你的只有一句话，私有数据库绝不是为了方便第三方修改而设计的。其设计中往往充满了假设、限制以及非标准的数据类型，还有其他像“请勿修改，否则后果自负”这样的警告信号。但企业为了省钱，往往会不顾这些警告信号而做出一些惊人的事情。因此软件开发人员就注定要深陷在这些私有数据库构成的丛林中，艰难探索了。

iBATIS在处理私有数据库时同样是一个非常优秀的持久化框架。通常这样的数据库只允许只读访问，使用iBATIS时，对这样的需求你大可放心，因为你可以限制运行的SQL的类别。当需求不允许数据库更新时，iBATIS绝不会对数据库执行任何神奇的更新操作。当需要更新时，私有数

数据库往往对数据的组织方式非常挑剔。iBATIS允许编写非常特定的更新语句以处理这种情况。

1.3.4 遗留数据库

如果说有什么东西的存在是现代面向对象软件开发人员的祸害的话，那么无疑是遗留数据库了。遗留数据库往往是曾经的企业数据库。它们具有企业数据库的各种复杂性和依赖关系。此外，它们还具有长年累月的修改、快速修正、掩饰、绕弯路、捆绑解决方案以及技术限制所带来的无尽的“伤痕”。更重要的是，遗留数据库通常是在不仅已经过时而且有时已完全不被支持的老式平台上开发出来的。因此对于现代开发人员来说可能已经没有适当的驱动程序和开发工具可用了。

iBATIS对于遗留数据库同样是有帮助的。只要你正在工作的系统有合适的数据库驱动程序可用，iBATIS就可以像对其他任何数据库那样发挥作用。事实上，iBATIS可能是处理遗留数据库问题最好的持久化框架了，因为它对数据库的设计没有任何假设，所以即使是对于最让开发人员头痛的遗留数据库，它也可以处理。

1.4 iBATIS 如何解决数据库的常见问题

在现代软件项目中数据库通常被认为是遗留组件。它们一直以来都被认为难以使用，不论是出于技术的还是非技术的原因。大多数软件开发人员宁可从头开始完全全地重建一个数据库。如果数据库是遗留下来的，相信一些开发人员会真心地希望负有此责的DBA能够去自杀©(take a long walk off a short pier)。只可惜两种情况都是不实际的，也不可能发生。不论你是否相信，数据库的存在总有它自己的理由——不论这个理由是否充分。可能是数据库变更的代价太高了，也可能是因为有其他系统依赖于该数据库。不论数据库被质疑的原因如何，我们都应该学会如何高效地与任何类型（包括饱受非议）的数据库打交道。iBATIS的开发主要就是为了对付那些设计非常复杂甚至非常糟糕的数据库。以下各节描述了数据库经常遇到的一些问题以及iBATIS是如何帮助解决这些问题的。

1.4.1 所有权与控制

在现代企业环境中，数据库存在的第一个同时也是最主要的困难其实完全不是技术问题，而是大多数现代软件企业都将数据库的所有权和责任从应用程序开发团队中分离了出来这样一个事实。数据库通常都是完全由企业中的一个独立小组所拥有。如果幸运的话，这个小组可能会与你的项目团队协同工作一起开发软件。否则，你的项目团队与数据库小组之间就可能存在一堵“墙”，你所有的需求都必须越过这堵墙传递给对方，然后期望它们能够被接收到并且得到理解。多么可悲啊，但事实就是这样，并且它一直都在发生着。

要和数据库团队打交道通常都是一件非常困难的事情。主要原因就在于他们往往都承受着巨大的压力且处理的项目不止一个。他们经常需要处理大量的需求，有时这些需求甚至可能是冲突的。数据库系统的管理的确是一项非常困难的任务，因此许多公司都认为它是一项至关重要的职

责。企业数据库系统的崩溃是会惊动该企业的CEO的。因此，数据库管理团队通常都非常谨慎。数据库系统的变更控制流程通常都比应用程序代码的变更控制流程要严格得多。数据库的某些变更可能会需要数据转移。其他一些变更则可能需要进行大量的测试以保证它们对性能不会造成影响。数据库团队难以交往看来的是有理由的，因此如果可能，能帮帮他们就帮帮他们吧。

当需要进行数据库设计以及数据库交互时，iBATIS通常能带来极大的灵活性。DBA都希望能够看到将在他们的数据库上运行的SQL，对于复杂的查询，他们甚至可能可以帮忙调整一下，而iBATIS使这种希望成为了现实。一些使用iBATIS的开发团队甚至拥有一个DBA或者数据建模人员来帮助他们直接维护iBATIS的SQL文件。数据库管理员和SQL编程人员要理解iBATIS绝对没有问题，因为背地里绝对不会发生任何意想不到的事情，他们可以看到所有的SQL语句。

1.4.2 被多个分散的系统访问

任何一个重要的数据库无疑都会拥有不止一个依赖者。即使该数据库只是简单地被两个Web应用程序所共享，也有许多事情需要考虑。假设有一个名为网上购物车（Web shopping cart）的Web应用程序，它使用了一个包含类别（category）代码的数据库。就网上购物车来说，类别代码是静态的，永远不会变化，所以该应用程序高速缓存了这些代码以提高性能。现在再假设有一个名为网上管理器（Web admin）的Web应用程序，它是用于更新类别代码的。这个网上管理器应用程序运行在一个不同的服务器上，是一个完全独立的程序。那么，设想一下，当网上管理器更新了一个类别代码时，网上购物车如何知道该刷新自己高速缓存的类别代码了呢。这是一个简单的例子，但有时确实是一个复杂的问题。

不同的系统可能会以不同的方式访问和使用数据库。一个应用程序可能是基于网络的电子商务系统，它会执行很多数据库更新和数据创建的操作。另一个应用程序则可能是一个规划好的批处理任务，定时从一个要求独占访问数据库表的第三方接口中加载数据。可能还有一个应用程序是报表引擎，它持续地要求数据库进行复杂的查询操作。可以很容易想象这样的情况会是多么的复杂。

最重要的是，一旦访问某个数据库的系统超过一个，情况就会变得有些复杂了。iBATIS在很多方面都能有所帮助。首先，iBATIS作为持久化框架对所有类型的系统（包括事务系统、批处理系统还有报表系统）都是有用的。这就是说，无论访问给定数据库的是什么系统，iBATIS都是一个非常好用的工具。其次，如果能够使用iBATIS，或者甚至是像Java这样的一致平台，那么就可以使用分布或高速缓存来在各个不同的系统之间进行通信。最后，对于最复杂的情况，你也可以很容易地禁用iBATIS高速缓存，然后自己编写能与当前情况完美契合的特定查询语句和更新语句，即使使用相同数据库的其他系统没有禁用高速缓存。

1.4.3 复杂的键和关系

设计关系数据库的本意就是需要它遵守一系列严格的设计规则。出于各种原因（不管是否正当），有时这些规则会被打破。如果某条规则被破坏、误解或者过度使用，就有可能导致出现复杂的键和关系。关系数据库设计的规则要求，表中的每一条记录都应通过一个主键来唯一地标

识。最简单的数据库设计可能会使用一个毫无意义的键作为主键。但也有一些数据库设计可能会使用一种所谓的自然键 (natural key)，此时真实数据的一部分被用作了键。即使如此，更加复杂的设计还可能会使用由两列或更多列组成的复合键。主键经常被用于在表与表之间创建关系。所以任何复杂或错误的主键定义都会因为它与其他表之间存在的关系而被传播出去。

有时，主键规则也可能没有被遵守。也就是说，有时数据根本就不存在主键。这就会使得数据库查询变得异常复杂，因为我们无法唯一地标识数据。这也会使得在表之间创建关系变得非常困难和混乱。这还会对数据库性能造成不好的影响，因为我们往往会在主键上建立索引以提高性能，且主键还被用来决定数据的物理顺序。

在另外一些情况下，主键规则又可能被过度使用了。数据库可能毫无实际理由地使用了复合自然键。这样的设计就是过于认真地遵循规则和尽可能地用最严格的方式实现规则带来的结果。使用自然键在表之间创建关系实际上会造成对一些真实数据的复制，这对于数据库的维护来说往往是一件糟糕的事情。复合键用于关系时就会造成更多的冗余，因为这会用到关联表中的好几列，只为了唯一地标识一条记录。在这两种情况下，由于自然键和复合键的使用，数据库的灵活性就丧失了，因为自然键和复合键会给数据库维护造成极大的困难，当需要进行数据迁移时更像是一场“噩梦”。

iBATIS可以处理任何类型的复杂键定义和关系。虽然最好还是将数据库设计得更合理一些，但iBATIS的确可以处理那些使用无意义键、自然键、复合键甚至根本没有键的表。

1.4.4 数据模型的去规范化或过度规范化

关系数据库的设计包括一个消除冗余的过程。冗余的消除非常重要，因为它保证了数据库能够提供较好的性能且灵活而可维护。数据模型中消除冗余的过程称为规范化 (normalization)，规范化的程度有好几个级别。没有经过任何处理的数据往往存在大量的数据冗余，因此也被认为是未规范化的。规范化是一个复杂的话题，我们在此不会讨论过多细节。

当一个数据库刚刚被设计出来时，我们常用那些未经加工的数据来分析冗余。数据库管理员、数据库建模人员甚至是开发人员都会分析这些数据，然后使用一些用于消除冗余的特定规则的集合来对它们进行规范化。没有经过规范化的数据模型会存在一些数据冗余（即在不同的表中有相同的数据），且每张表都有大量的记录和大量的列。经过规范化的数据模型的冗余就很少甚至没有了，这时模型中会存在较多的表，但每张表中只有几列和几条记录。

没有哪个级别的规范化是完美的。一个未经规范化的模型具有简单的优点，甚至有时在性能上也会具有一些优势。这种模型的数据存取速度可能比经过规范化的模型还更快。造成这种现象的原因是未经规范化的模型中需要执行的语句和关系演算更少，因此负担也就更少。也就是说，去规范化应该总是例外而不能作为一条规则。良好的数据库设计往往开始于一个“教条化”的规范化模型，然后再根据需要对它去规范化。经过规范化后再去规范化的过程总比没有规范化而需要重新规范化来得简单得多。因此，总是从一个规范化的数据模型开始数据库设计吧。

也存在数据库被过度规范化 (overnormalized) 的情况，其结果也会带来很多问题。太多的表就需要创建太多的关系，以致难以维护。过度规范化的数据库会造成的问题包括：当查询数据库时，需要执行很多表连接操作；当更新关系紧密的数据时，需要执行很多更新语句。所有这些都会对数据库性能造成负面影响。还有一点，当需要把这样的数据模型映射到对象模型时，通常也会更加困难，因为你可能不希望你的对象模型拥有与数据模型一样的如此细粒度的类。

去规范化的模型也可能存在问题，甚至可能比过度规范化的模型还要多。去规范化的模型容易具有较多的记录和较多的列。过多的记录会对性能造成负面影响，原因是查找时需要扫描的数据更多了。过多的列也存在同样的问题，因为每条记录的内容都更多了，因此每次执行更新或查询时就需要更多的资源。对这样的大表执行某些操作（如更新或查询）时要格外小心，要保证只针对那些必要的列，切忌将那些无关列也搅和进来。还要说明的一点就是，对于去规范化的模型，要创建有效的索引通常会很困难。

iBATIS对去规范化的模型和过度规范化的模型都是适用的。因为它没有对你的对象模型或数据模型的粒度做任何假设，它也没有要求两者之间必须相同或基本相似。iBATIS在分离对象模型和关系模型这件事上，恐怕是做得最好的了。

1.4.5 瘦数据模型

瘦数据模型是一种最为臭名昭著并且问题多多的对关系数据库系统的滥用。不幸的是，有时又的确需要瘦数据模型。所谓瘦数据模型，就是简单地将每张表都设计为一种通用数据结构，用于存储名值对的集合。这非常像Java中的属性文件，或者像Windows中那种旧式的INI (initialization) 文件。有时这些表也可用于存储元数据，例如期望的数据类型等。这是必要的，因为数据库只允许一列有一种类型定义。要更好地理解瘦数据模型，考虑下面这个典型的地址数据的示例，如表1-2所示。

表1-2 典型模型中的地址数据

ADDRESS_ID	STREET	CITY	STATE	ZIP	COUNTRY
1	123 Some Street	San Francisco	California	12345	USA
2	456 Another Street	New York	New York	54321	USA

很显然这个地址数据表可以进一步规范化。例如，可以创建COUNTRY (国家)、STATE (州)、CITY (城市) 和ZIP (邮编) 这样的关联表。但当前这样一个设计对于大多数应用程序来说既简单又高效，已经足够了。除非你的需求的确非常复杂，否则这个模型应该不会有任何问题。

还是使用以上的数据，如果将它们放入一个瘦数据模型对应的表中，结果将如表1-3所示。

表1-3 瘦数据模型中的地址数据

ADDRESS_ID	FIELD	VALUE
1	STREET	123 Some Street
1	CITY	San Francisco

(续)

ADDRESS_ID	FIELD	VALUE
1	STATE	California
1	ZIP	12345
1	COUNTRY	USA
2	STREET	456 Another Street
2	CITY	New York
2	STATE	New York
2	ZIP	54321
2	COUNTRY	USA

这样的设计绝对是场噩梦。首先，没有任何可能对它进一步规范化了，虽然当前的模型只能算作第一范式。其次，没有任何机会创建与COUNTRY表、CITY表、STATE表或ZIP表的关联关系了，因为我们不可能在同一列上定义多个外键。再次，如果希望执行一条涉及多个地址字段（例如，执行一个以街道和城市作为查询条件的查询语句）的“样例查询”，这样的数据实在让人头痛，它可能需要一大堆复杂的子查询。再看看更新的情况，这样的设计就性能来说也特别糟糕，仅仅是插入一个地址就需要在同一张表上执行5条插入语句。这种情况下出现锁竞争甚至死锁的可能性也大大增加了。此外，这个瘦数据模型中记录的数量整整是我们的规范化数据模型的5倍。由于记录数量过大，又缺少明确的数据定义，而且更新一条记录时需要的更新语句过多，创建有效的索引也是不可能的了。

不用再多说了，这个设计确实问题多多，我们为何要不惜一切代价避免这样的设计，原因已经再明白不过了。但话说回来，这个设计也不是毫无用处，它唯一的用武之地就在于那些需要动态字段的应用程序。有些应用程序的确有这样的需求，它允许用户对他们的记录添加额外的数据。如果用户希望能定义新的字段，然后在应用程序运行时动态地把数据插入到这些字段中，那么这样的模型就可以工作得很好。也就是说，所有的已知数据还是应该正确地规范化，而这些额外的动态字段则可以通过关联关系与这些已知数据建立父子关系。这样的设计同样存在我们之前讨论过的所有问题，但它们被最小化了，因为大部分数据（很可能是那些最重要的数据）都已经被正确地规范化了。

即使在一个企业数据库中遇到了瘦数据模型，iBATIS也可以帮助你处理它。要将若干类映射为瘦数据模型是非常困难的，甚至是根本不可能的，因为你连数据模型中可能存在哪些字段都无法明确。此时你最好将这样的类映射为一个散列表（hashtable），而幸运的是iBATIS支持这种映射。使用iBATIS，你不必将每一张表都映射为一个用户定义的类型。iBATIS允许你将关系数据映射为Java基本类型（primitive）、Map实例、XML还有用户定义类（如JavaBean）。这种巨大的灵活性使得iBATIS对于包括瘦数据模型在内的复杂数据模型非常有效。

1.5 小结

iBATIS被设计为一个混合型解决方案，它并不试图解决所有的问题，相反它只希望能解决那

些最重要的问题。iBATIS从各种数据库访问工具中汲取了大量的优秀思想。像存储过程一样，所有的iBATIS语句都有一个签名，定义了语句的名字和输入输出（封装）。与内联SQL类似，iBATIS允许SQL语句按照其最自然的方式书写，并且可以直接使用语言中的变量作为输入输出参数和结果。像动态SQL一样，iBATIS允许在运行时修改SQL。这样的查询语句可以根据用户的请求动态构建。从对象/关系映射工具中，iBATIS借用了许多概念，包括高速缓存、延迟加载，还有更高级的事务管理。

在一个应用程序的架构中，iBATIS适用于持久层。iBATIS也通过提供一些特性来支持其他层，这些特性使得对所有这些层的需求的实现都变得更加容易。例如，一个Web搜索引擎可能需要搜索结果的分页列表。iBATIS支持这种特性，因为它允许查询时指定返回结果的偏移量（如，起点）和行数量。这就使得分页操作可以在一个较低的层次上执行，同时保持数据库细节可以远离我们的应用程序。

iBATIS可用于任何大小和用途的数据库。首先，iBATIS非常适合于那些较小的应用程序数据库，因为它非常容易学习和快速上手。其次，iBATIS对大型企业应用程序也非常合适，因为它没有对数据库的设计、行为或者那些可能对我们的应用程序如何使用数据库产生影响的依赖关系做任何假设。再次，即使是对于那些在设计上存在着争议或者深陷于高层政策混乱之中的数据库，iBATIS也可以工作得很好。综上所述，iBATIS被设计得非常灵活以至于几乎可适用于任何情况了。当然，使用iBATIS也可以为你节省大量的时间，因为你再不用写那些重复的、样本一样的代码了。

本章讨论了iBATIS的理念和起源。下一章将仔细解释什么是iBATIS以及它是如何工作的。

第2章

iBATIS是什么

2

本章内容

- 何时使用iBATIS
- 何时不使用iBATIS
- 开始使用iBATIS
- iBATIS未来的方向

在第1章中我们详细地讨论了iBATIS背后的设计理念以及iBATIS框架是如何产生的。我们也说明了iBATIS是一个混合型解决方案，它从处理关系数据库的其他不同方法那里借鉴了许多思想。那么iBATIS到底是什么呢？本章将解答这个问题。

iBATIS就是我们通常所说的数据映射器（data mapper）。Martin Fowler在他的著作《企业应用架构模式》（Addison-Wesley Professional, 2002）^①中，对Data Mapper模式是这样描述的：

所谓映射器^②层，是用于在对象和数据库之间搬运数据，同时保证对象、数据库以及映射器本身都相互独立。

Martin在区分数据映射以及元数据映射上，确实做了一件非常出色的工作，元数据映射正是适合使用O/RM工具的地方。O/RM工具将数据库表及其列映射为应用程序中的类及字段。或者说，O/RM工具在数据库的元数据与类的元数据之间建立起了一种映射关系。图2-1展示了所谓的O/RM，它在一个类与数据库表之间建立了映射关系。在这种情况下，类的每一个字段都被映射为数据库中唯一的对应列。

iBATIS与O/RM不同，它不是直接把类映射为数据库表或者说把类的字段映射为数据库列，而是把SQL语句的参数与结果（也即输入和输出）映射为类。正如你将在本书的后续部分中学到的，iBATIS在类和数据库表之间建立了一个额外的间接层，这就为如何在类和数据库表之间建立映射关系带来了更大的灵活性，使得在不用改变数据模型或者对象模型的情况下改变它们的映射关系成为可能。其实我们这里讨论的这个间接层就是SQL。SQL这个额外的间接层使得iBATIS能

① 本文注释英文版即将由人民邮电出版社出版。——编者注

② 根据Martin Fowler在该书中所描述的，映射器是一个用于在两个相互独立的对象之间建立通信的对象。——译者注

够更好地隔离数据库设计和应用程序中使用的对象模型。这就使得它们两者之间的相关性降至最低。图2-2展示了iBATIS如何使用SQL映射数据。

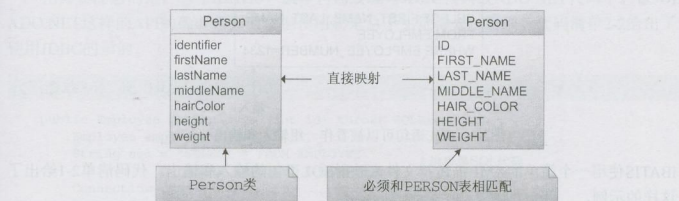


图2-1 对象/关系映射

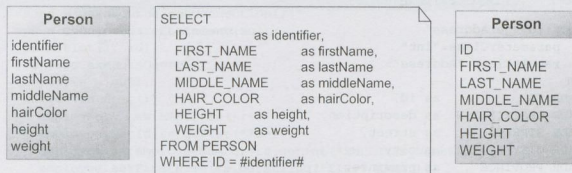


图2-2 iBATIS的SQL映射

如图2-2所示，iBATIS的映射层其实就是SQL。iBATIS让你编写SQL语句。iBATIS负责在类的特性（property）^①和数据库表的列之间映射参数和结果。基于这个原因，也考虑到与其他各种各样的映射方式的区分，为避免混淆，iBATIS团队通常将所谓的“数据映射器”称为SQL映射器（SQL mapper）。

2.1 映射 SQL 语句

任何一条SQL语句都可以看作是一组输入和输出。输入即参数（parameter），通常可以在SQL语句的WHERE子句中找到。输出则是SELECT子句中指定的那些列。图2-3描述了这个思想。

这种方式的最大优点就在于SQL语句使得开发人员能够自己把握巨大的灵活性。开发人员可以在不改变基础表设计的前提下轻松地操纵数据以适应对象模型。甚至，开发人员还可以通过数据库内建的函数和存储过程引入多表或结果集。总之，他们可以随心所欲地利用SQL的强大能力。

① 参照科学出版社在2002年出版的《计算机科学技术名词》（第2版），本书将“property”译为“特性”，将“attribute”译为“属性”。——编者注

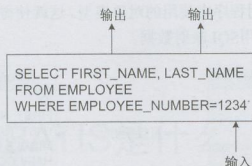


图2-3 SQL语句可以被看作一组输入和输出

iBATIS使用一个简单的XML描述符文件来映射SQL语句的输入和输出。代码清单2-1给出了一个这样的示例。

代码清单2-1 一个SQL映射描述符示例

```
<select id="getAddress"
      parameterClass="int"
      resultClass="Address">
SELECT
  ADR_ID           as id,
  ADR_DESCRIPTION  as description,
  ADR_STREET       as street,
  ADR_CITY         as city,
  ADR_PROVINCE     as province,
  ADR_POSTAL_CODE  as postalCode
FROM ADDRESS
WHERE ADR_ID = #id#
</select>
```

从以上代码中我们看到了一条SQL SELECT语句，它返回地址数据。从<select>元素中可以看出该语句使用一个Integer对象作为参数，该参数是通过WHERE子句中的#id#符号标记的。我们还可以看出该语句的结果是一个Address类的对象实例，假设Address类的所有字段名与SELECT语句中指定的各个列的别名(通过as关键字)相同，例如，ADR_ID列的别名为id，故会映射到Address类中名为id的特性上。不论你是否相信，要映射一条SQL语句使其接受一个Integer作为参数并返回一个Address对象作为输出，你要做的就是这些了。用于执行以上语句的Java代码如下：

```
Address address = (Address) sqlMap.queryForObject("getAddress",
                                                  new Integer(5));
```

SQL映射这个概念具有很好的可移植性，可应用于任何一个功能完备的编程语言。例如，基于iBATIS.NET的C#代码就与之前的Java代码几乎是一模一样的：

```
Address address = (Address) sqlMap.QueryForObject("getAddress", 5);
```

当然，iBATIS针对SQL映射还有许多高级选项，特别是针对映射的返回结果。关于这些高级选项，我们会在本书的第二部分中详细讨论。现在，我们最重要的任务就是了解iBATIS的特性和好处，以及它如何工作。

2.2 iBATIS 如何工作

尤其要注意的是，有了iBATIS，就不再需要编写JDBC代码或ADO.NET代码了。像JDBC和ADO.NET这样的API的确非常强大，但使用起来总不免觉得太过繁琐。代码清单2-2给出了一个使用JDBC的示例。

代码清单2-2 使用JDBC的代码示例

```
public Employee getEmployee (int id) throws SQLException {
    Employee employee = null;
    String sql = "SELECT * FROM EMPLOYEE " + "WHERE EMPLOYEE_NUMBER = ?";
    Connection conn = null;
    PreparedStatement ps = null;
    ResultSet rs = null;
    try {
        conn = dataSource.getConnection();
        ps = conn.prepareStatement(sql);
        ps.setInt(1, id);
        rs = ps.executeQuery();
        employee = null;
        while (rs.next()) {
            employee = new Employee();
            employee.setId(rs.getInt("ID"));
            employee.setEmployeeNumber(rs.getInt("EMPLOYEE_NUMBER"));
            employee.setFirstName(rs.getString("FIRST_NAME"));
            employee.setLastName(rs.getString("LAST_NAME"));
            employee.setTitle(rs.getString("TITLE"));
        }
    } finally {
        try {
            if (rs != null) rs.close();
        } finally {
            try {
                if (ps != null) ps.close();
            } finally {
                if (conn != null) conn.close();
            }
        }
    }
    return employee;
}
```

这里是SQL代码

从这个例子中很容易看出，JDBC API会产生许多额外的开销。尽管如此，每一行代码又都是必不可少的，所以要减少代码量还真不是一件容易的事情。最多也只不过是将其其中的一些代码挪到某个实用方法（utility method）中，最明显的就是那些关闭资源（如PreparedStatement和ResultSet）的代码。

其实，如果使用iBATIS，iBATIS在后台也是运行几乎相同的JDBC代码。iBATIS会获取数据库连接，设置其参数，执行其语句，获取执行结果，并在最后关闭所有的资源。然而，需要自己

亲自编写的代码量却大大地减少了。代码清单2-3给出了使用iBATIS运行相同的SQL语句时你需要编写的代码。

代码清单2-3 iBATIS显然比JDBC要精简得多

```
<select id="getEmployee"
      parameterClass="java.lang.Integer"
      resultClass="Employee">
  SELECT ID          as id,
         EMPLOYEE_NUMBER as employeeNumber,
         FIRST_NAME     as firstName,
         LAST_NAME      as lastName,
         TITLE          as title
  FROM EMPLOYEE
 WHERE EMPLOYEE_NUMBER = #empNum#
</select>
```

根本无需比较，iBATIS代码明显更加简洁，更容易阅读，因此也更容易维护。本章稍后会讨论更多关于使用iBATIS的好处。但是现在，你最关心的可能是如何用Java代码运行以上的语句。如前例所示，运行它仅仅需要一行简单代码：

```
Employee emp = (Employee) sqlMap.queryForObject("getEmployee",
                                                  new Integer(5));
```

无需多说，这行代码会执行相应的SQL语句，设置其参数，并以一个真实的Java对象的形式作为结果返回。SQL语句被“干干净净”地封装在Java代码之外的一个XML文件中。iBATIS负责管理幕后的所有资源，其运行的实际效果与我们之前在代码清单2-2中所见的JDBC代码示例是完全一样的。

这就引起一个问题，iBATIS对所有的系统来说是否都以一样的方式工作？或者它仅仅是适用于某一类特定的应用程序？以下几节我们将回答这个问题，首先从iBATIS是多么地适合于小型应用程序说起。

2.2.1 iBATIS 之于小型、简单系统

小型应用程序通常只涉及单个数据库，只有一些相当简单的用户界面和领域模型。它的业务逻辑非常简单，甚至对一些简单的CRUD（Create, Read, Update, Delete：增查改删）应用程序来说可能根本就不存在。iBATIS之所以非常适合于小型应用程序，有3个原因。

第一，iBATIS本身就很很小并且简单。它不需要服务器或者其他任何类型的中间件。根本不需要任何额外的基础设施（infrastructure）。iBATIS也没有任何第三方依赖。iBATIS的最简安装只需要2个JAR文件，总计不过375KB。除了需要配置几个SQL映射文件外，iBATIS不需要进行任何其他安装，因此只需要几分钟时间就可以拥有一个可工作的持久层了。

第二，iBATIS不会对应用程序或者数据库的现有设计强加任何影响。因此，如果你有一个小型系统，且已经部分实现或者甚至已经发布了，那么仍然可以轻松地重构成持久层使用iBATIS。因为iBATIS很简单，所以它根本不会使得应用程序的架构过分复杂。而如果使用对象/关系映射

工具或者代码生成工具，由于它们都对应用程序或数据库的设计做了某些假设，因此它们不可能对应用程序的架构毫无影响。

最后，只要有过软件开发的经验，相信你就不会怀疑，任何一个小软件都几乎不可避免地有一天会成长为一个大数据件。所有成功的软件都有进一步成长的趋势。这是一件好事，而我们接下来想说的就是，iBATIS同样非常适合于大型系统，它甚至可以扩展以满足企业级应用程序的需要。

2.2.2 iBATIS 之于大型、企业级系统

iBATIS当初就是为企业级应用程序而设计的。最重要的是，iBATIS在这个领域比之其他解决方案有着大量的优点。iBATIS最初的创建者都只有从大型应用到企业级应用程序系统的开发经验。这些系统通常都涉及不止一个数据库，且所有这些数据库都是不可控的。在第1章中我们讨论了各种类型的数据库，包括企业级数据库、私有数据库和遗留数据库。创建iBATIS框架一个很重要的原因就是针对这样的数据库。因此，iBATIS拥有许多适合于企业应用环境的特征。

其实iBATIS适用于大型系统中的第一个原因我们已经在其他地方说过了，不过这个原因的确很重要，所以我们再强调一下也不为过：iBATIS没有对数据库模型或对象模型的设计做任何假设。不论你的应用程序中这两个模型之间是多么不匹配，iBATIS都能适用。况且，iBATIS没有对你的企业级系统的架构做出任何假设。不论你对数据库是根据业务功能纵向划分，还是按照技术横向划分，iBATIS都允许高效地处理数据并将它整合到面向对象的应用程序中去。

第二点，iBATIS的某些特征使得它能够非常高效地处理大型数据集。iBATIS支持的行处理器（row handler）使得它能够批处理超大记录集（一次一条记录）。iBATIS也支持只获取某个范围内的结果，这就使得你可以只获取那些你当前亟需的数据。例如，假设你有10 000条记录，但只需要其中的第500条～第600条，那么就可以轻松地仅获取这些记录。iBATIS支持驱动程序提示，使得执行这样的操作非常高效。

最后一点，iBATIS允许你用多种方式建立从对象到数据库的映射关系。一个企业级系统只以一种模式工作的情况是非常少见的。许多企业级系统需要在白天执行事务性的功能，而在晚上执行批处理功能。iBATIS允许你将同一个类以多种方式映射，以保证每一种功能都能以最高效的方式执行。iBATIS同样支持多种数据获取策略。例如，可以选择对某些数据进行延迟加载，也可以将一个复杂的对象图只用一条联合查询SQL语句就同时加载完毕，从而避免严重的性能问题。

以上所说的这些好像是在自吹自擂了。那么，既然我们有兴趣，为何不继续深入研究一下需要使用iBATIS的理由呢？2.3节会做这件事情。并且为了公平起见，2.4节还会讨论一些你不应该使用iBATIS的情况。

2.3 为何使用 iBATIS

我们有充足的理由说明，你几乎可以在任何系统中使用iBATIS。正如前面小节中所介绍的，像iBATIS这样一个框架能够使你的应用程序从架构级别上受益。本节就将讨论这些益处以及使这

些益处成为可能的iBATIS特征。

2.3.1 简单性

iBATIS被广泛认为是当今可用的最简单的持久化框架之一。简单性的理念根植于iBATIS开发团队,它在iBATIS的所有开发目标中居于首位。这种简单性的取得是因为iBATIS直接构建于JDBC和SQL之上。iBATIS对于Java开发人员来说非常简单,因为它除了不用编写那么多代码外,与JDBC的工作机制非常相像。几乎你知道的关于JDBC的一切都对iBATIS同样适用。几乎可以认为,iBATIS就是以XML的形式描述的JDBC代码。也就是说,iBATIS拥有许多JDBC所没有的架构级的优点,我们随后就将讨论这些优点。iBATIS对于数据库管理员以及SQL程序员来说也非常容易理解。iBATIS配置文件几乎人人都能读懂,只要有SQL编程的经验。

2.3.2 生产效率

任何一个优秀的框架,其基本目的都是使得框架的使用者能够获得更高的生产效率。一般情况下,框架负责处理公共的任务,减少编写重复的样板代码,以及解决复杂的架构级的问题。iBATIS在给开发人员带来更高的开发效率方面做得非常成功。在意大利的Java用户组(Java Users Group)所做的一个案例研究中(参见www.jug Sardegna.org/vq/wiki/jsp/Wiki?iBatisCaseStudy),Fabrizio Gianneschi发现iBATIS减少了持久层大约62%的代码量。之所以能减少如此多的代码量,究其原因还是开发人员再不需要编写繁琐的JDBC代码了。SQL语句仍然是硬编码的,不过就像你在前面小节中所看到的,SQL不是问题——问题在于JDBC API,对于ADO.NET而言也是如此。

2.3.3 性能

性能这个话题无疑会引起框架开发人员、框架使用者以及商业软件开发商之间的一场激烈争论。事实是,从一个较低的层次来看这个问题,无疑所有的框架都会带来一定的性能损失。一般来说,如果你比较硬编码的JDBC代码和iBATIS代码,做一个1 000 000次的for循环,就会发现JDBC在性能上的确有一些优势。幸运的是,在现代应用程序开发中,以上这样的for循环带来的性能损失并不重要。真正重要的是,你如何从数据库中获取数据,何时获取,获取的频率又是多少。例如,从数据库中动态地获取记录的分页查询之所以能大大提高应用程序的性能就在于,你不会将潜在的成千上万条记录从数据库中一次取出。同样地,使用像延迟加载这样的特征可以避免加载那些当前用例并不需要的数据。另一方面,如果你确定需要加载一份复杂的对象图,涉及来自多个表的大量数据,那么使用一条SQL语句就完成所有对象的加载显然可以大大提高效率。iBATIS支持许多性能优化措施,我们将在本章的后续章节中详细讨论。就目前而言,最重要的是要知道iBATIS总是能通过一种简单的方式来配置和使用,其性能与JDBC相当,甚至更好。另一个需要重点考虑的问题就是,并不是所有的JDBC代码都是编写良好的。JDBC API非常复杂,编写正确的代码需要非常小心。不幸的是,大量的JDBC代码都编写得相当糟糕,因此从较低层次上看甚至还没有iBATIS工作得好。

2.3.4 关注点分离

在典型的JDBC代码中,在应用程序的各个层中都能找到各种数据库资源(例如连接和结果集),这并不稀奇。相信大家都见过甚至开发过这样糟糕的应用程序——所有的数据库连接和语句都在JSP页面中完成,结果被反复迭代,而且HTML代码散落其间。这样的代码简直是一场噩梦。第1章已经讨论了应用程序分层的重要性。我们已经看到了如何从较高的角度对应用程序分层,以及持久层内部又是如何进一步分层的。iBATIS通过帮助管理所有这些持久化相关的资源来支持分层,这些资源包括数据库连接(database connection)、预处理语句(prepared statement)以及结果集(result set)。iBATIS提供了一组数据库无关的接口以及API,使得应用程序的其他部分与持久化相关的资源无关。使用iBATIS,你的代码总是在直接与严格的对象打交道,再也不用管那些随意的结果集了。iBATIS实际上使得你难以违背对应用程序分层这样的最佳实践。

2.3.5 明确分工

一些数据库管理员是如此热爱他们的数据库,以至于不愿意让任何其他人为数据库编写SQL。而有一些人又是如此擅长编写SQL,以至于其他人都想让他们来做这项工作。无论是什么原因,在你的开发团队中实现人尽其才总是有利的。如果你的团队中有人尤其擅长于编写SQL,但是对于Java或C#却不那么在行,那么就可以让他们专门编写SQL。iBATIS使得这种分工成为可能。因为在iBATIS中,SQL语句在很大程度上同应用程序的源代码是分离的,SQL程序员可以按照SQL原本的方式来编写它,而不必担心有关SQL字符串拼接的问题。即使由相同的开发人员来编写Java代码和SQL,在数据库性能调整(performance tune)的过程中也会有一个来自DBA的常见请求,即“显示出SQL”。用JDBC完成此事并不容易,因为SQL常常隐藏在一系列拼接的字符串中,甚至还可能是在递归语句和条件语句的组合中动态地创建的。如果使用对象关系映射,情况就更加复杂了,因为你通常必须运行应用程序然后才能记录这些SQL语句,即使找到了它们,也根本不能改变它们。iBATIS则提供了充分的自由,使得任何人都可以开发、浏览甚至修改在数据库中执行的SQL语句。

2.3.6 可移植性: Java、.NET 及其他

iBATIS是可移植的。由于相对简单的设计,它几乎可以用任何一种语言在任何一个平台上实现。撰写本书时,iBATIS支持3种最受欢迎的开发平台:Java、Ruby和微软.NET的C#。

当前,配置文件并不完全跨平台兼容,但是我们正在计划使它们更加兼容。重要的是,概念(concept)和方法(approach)都是可移植的。这就使你能够保证所有的应用程序在设计上是一致的。和其他框架相比,iBATIS可以处理更多语言以及更多类型的应用程序,而无需考虑应用程序设计不同。如果应用程序的一致性对于你来说很重要,那么使用iBATIS就可以很好地帮助实现这一点。

2.3.7 开源和诚实

前面我们说本节有“自吹自擂”之嫌。事实上,iBATIS是免费的开源软件。无论你是否使用

它，我们都不会因此而获得哪怕是一分钱的利益。你已经买了本书，所以我们已经获得了该得的利益。也就是说，开源软件的最优点之一就是诚实。我们没有任何理由夸大事实或者撒谎。因此，下面我们将要做的事情在商业软件文档中就是非常罕见的：我们将要讨论一些你可能不应该使用iBATIS的情况，并且提出一些合适的替代方案。

2.4 何时不该使用 iBATIS

每个框架都是建立在一定的规则和约束之上的。底层的框架，如JDBC，提供了一个灵活且完备的功能集合，但是它们使用起来却更难也更繁琐。高层的框架，如对象/关系映射工具，使用起来就方便多了并且可以帮你节省大量工作，但是它们是在更多的假设和约束的基础上的，这使得它们只适用于较少的应用程序。

iBATIS是一个中层的框架。它比JDBC的层次高一些，但是相对于对象/关系映射工具，层次又要更低一些。这使得iBATIS处于一个很独特的位置上，使它能够适用于一些较为特别的应用程序。在本书前面的小节中，我们讨论了为什么iBATIS对于多种应用程序类型（包括小的客户端应用程序和较大的、企业级的Web应用程序以及任何介于这两者之间的应用程序）都是有用的。那么，何时不该使用iBATIS呢？下面几节就将详细描述不适合使用iBATIS的各种情况，并且给出有关替代方案的建议。

2.4.1 当永远拥有完全控制权时

如果能够保证拥有对应用程序设计和数据库设计的完全控制权，那你就确非常幸运。这在商业环境或者任何一个核心工作不是软件开发的行业中都是非常少见的。然而，如果你在一家软件公司工作，并且在开发一个你拥有完全设计控制权的塑封包装（shrink-wrapped）产品时，那么你可能恰好处于这种情况。

当你具有完全控制权时，就有充分理由使用一个完全的对象/关系映射方案，如Hibernate。你可以充分利用对象/关系映射工具所能提供的设计优势并提高生产效率。可能根本没有来自企业数据库小组的干扰，也不需要与遗留系统整合。此外，数据库可能是与应用程序一同部署的，这使得它属于应用程序数据库的范畴（参见第1章）。一个很好的使用Hibernate的应用程序示例是Atlassian的JIRA。他们提供了一个问题跟踪软件，作为一个他们可以完全控制的塑封包装产品。

然而，还需要考虑应用程序未来的发展。如果数据库有可能超出应用程序开发人员的控制，那么就必须仔细考虑一下使用对象/关系映射将对你的持久化策略带来的影响。

2.4.2 当应用程序需要完全动态的 SQL 时

如果应用程序的核心功能是动态生成SQL，那么iBATIS就是错误的选择。iBATIS支持非常强大的动态SQL特征，这些特征又反过来支持高级查询能力，甚至是一些动态更新功能。然而，如果系统中的每个语句都是动态地生成的，那么最好使用原始的JDBC，甚至可以创建自己的框架。

iBATIS的强大功能之一就是它允许你拥有完全的自由，可以手工编写和直接操作SQL。当应

用程序中大部分的SQL都是从某些SQL生成器类中动态地构建时，这种优势就会很快丧失。

2.4.3 当没有使用关系数据库时

对于关系数据库之外的其他数据库，也存在可用的JDBC驱动程序。对于平板文件、微软的Excel电子表格、XML，以及其他类型的数据存储平台，都有相应的JDBC驱动程序。虽然一些人在iBATIS中使用这些驱动程序也获得了成功，但是对于大多数用户我们并不推荐使用这些驱动程序。

iBATIS并不会对你的环境做出任何假设。但是它确实期望你使用的是真正的关系数据库，即支持事务、相对典型的SQL和存储过程语义的关系数据库。即使一些非常著名的数据库也可能不支持关系数据库的某些关键特征。如MySQL的早期版本就不支持事务，因此iBATIS不能很好地处理MySQL。幸运的是，当前的MySQL已支持事务并且还有一个非常符合规范的JDBC驱动程序。

如果你使用的不是真正的关系数据库，我们推荐你最好使用原始的JDBC，甚至更低层的文件I/O API。

2.4.4 当 iBATIS 不起作用时

随着社区提出的需求越来越多，iBATIS也不断地实现了许多非常好的特性。然而，iBATIS有其自己的发展方向和设计目标，因此它有时候就有可能会同一些应用程序的需求发生冲突。人们在软件帮助下可以完成很多神奇的事情，但是有时候由于需求过于复杂，软件可能完全不起作用，iBATIS也是如此。虽然我们也可以在iBATIS中添加几种特性以支持这些需求，但是这么做可能会极大地提高复杂性，甚至可能会改变iBATIS框架的适用范围。因此，我们不会修改框架。为了解决以上问题，我们将提供可插拔的接口，这样就可以扩展iBATIS以满足几乎任何需求。事实上，有时候iBATIS就是不起作用。此时，最好另寻一个更好的解决方案，而不是“削足适履”地应用程序iBATIS（或者其他任何框架）。

下面我们不再讨论使用iBATIS的是是非非了，还是来看一个简单的示例吧。

2.5 5分钟内用 iBATIS 创建应用程序

iBATIS框架实际上非常简单，要开始使用它也同样非常简单。那么究竟有多简单呢？事实上，它是如此简单，以至于你可以用iBATIS在5分钟之内创建一个完整的应用程序——不是一个大型的企业资源规划（Enterprise Resource Planning，ERP）解决方案或者一个大型电子商务网站，而是一个简单的命令行工具，用于执行iBATIS SQLMap中的一条SQL语句并且向控制台输出执行结果。下面将要给出的例子将配置一条简单的静态SQL语句，用于查询一个简单的数据库表，并且把它以如下形式输出到控制台：

```
java -classpath <...> Main
```

```
Selected 2 records.
```

```
{USERNAME=LMEADORS, PASSWORD=PICKLE, USERID=1, GROUPNAME=EMPLOYEE}
```

```
{USERNAME=JDOR, PASSWORD=TEST, USERID=2, GROUPNAME=EMPLOYEE}
```

这种数据输出方式并不是最漂亮的，但是由此你可以了解该应用程序到底要做什么。在下面几节中，我们将逐步从无到有地实现此功能。

2.5.1 安装数据库

为了满足样例应用程序的目的，我们将使用MySQL数据库。iBATIS框架可以使用任何数据库，只要该数据库具有符合规范的JDBC驱动程序。你只需要在配置文件中提供驱动程序的类名以及一个JDBC URL即可。

安装数据库服务器超出了本书所讨论的范围，因此我们假设数据库服务器已经安装好并且可用了，然后告诉你在此基础上需要做什么。以下的MySQL脚本用于构造我们将要使用的表格，并且在其中添加了一些样例数据：

```
#
# Table structure for table 'user'
#

CREATE TABLE USER_ACCOUNT (
  USERID          INT(3) NOT NULL AUTO_INCREMENT,
  USERNAME        VARCHAR(10) NOT NULL,
  PASSWORD        VARCHAR(30) NOT NULL,
  GROUPNAME       VARCHAR(10),
  PRIMARY KEY     (USERID)
);

#
# Data for table 'user'
#

INSERT INTO USER_ACCOUNT (USERNAME, PASSWORD, GROUPNAME)
VALUES ('LMEADORS', 'PICKLE', 'EMPLOYEE');
INSERT INTO USER_ACCOUNT (USERNAME, PASSWORD, GROUPNAME)
VALUES ('JDOE', 'TEST', 'EMPLOYEE');

COMMIT;
```

如果已经安装了一个不同的数据库服务器，其中包含了一些你想要在其上执行某些SQL查询的其他数据，那么可以大胆地在本示例中使用它。只需要修改SqlMap.xml文件中的查询语句，以包含你的SQL，同时还需要修改SqlMapConfig.xml文件以配置iBATIS从而使用你的数据库。为使整个示例成功运行，你还需要知道驱动程序的名称、JDBC URL以及连接时的用户名和密码。

2.5.2 编写代码

由于这个应用程序是我们给出的第一个完整示例，同时也是对使用iBATIS的一个介绍，因此它的代码将会比真正的应用程序要简单得多。关于类型安全和异常处理，我们以后将会详细讨论，因而对于这些话题此处将不予考虑。代码清单2-4给出了完整的代码：

代码清单2-4 Main.java

```
import com.ibatis.sqlmap.client.*;
import com.ibatis.common.resources.Resources;

import java.io.Reader;
import java.util.List;

public class Main {
    public static void main(String arg[]) throws Exception {
        String resource = "SqlMapConfig.xml";
        Reader reader = Resources.getResourceAsReader(resource);
        SqlMapClient sqlMap = SqlMapClientBuilder.buildSqlMapClient(reader);
        List list = sqlMap.queryForList("getAllUsers", "EMPLOYEE");
        System.out.println("Selected " + list.size() + " records.");
        for(int i = 0; i < list.size(); i++) {
            System.out.println(list.get(i));
        }
    }
}
```

配置 iBATIS

打印结果

执行语句

就是这些了！我们在大约10行Java代码中就完成了对iBATIS的配置，执行了SQL语句，并且打印了结果。以上就是一个功能完整的iBATIS应用程序所需要的全部Java代码。稍后将对其进行改进，但是现在将继续讨论有关iBATIS配置的基础知识。

2.5.3 配置 iBATIS（预览）

考虑到下一章将深入介绍如何配置iBATIS，所以此处将只是简单介绍一下。你不会找到有关配置选项的过多解释，但是我们将给出最重要的信息。

首先，让我们来研究SqlMapConfig.xml文件。它是使用iBATIS的起点，负责把所有的SQL映射文件组合在一起。代码清单2-5给出了我们的简单应用程序中使用的SqlMapConfig.xml文件。

代码清单2-5 以前编写的最简单的iBATIS应用程序中的SQL映射配置

```
<?xml version="1.0" encoding="UTF-8" ?>

<!DOCTYPE sqlMapConfig
PUBLIC "-//ibatis.apache.org/DTD SQL Map Config 2.0//EN"
"http://ibatis.apache.org/dtd/sql-map-config-2.dtd">

<sqlMapConfig>
<transactionManager type="JDBC" >
<dataSource type="SIMPLE">
<property name="JDBC.Driver"
value="com.mysql.jdbc.Driver"/>
<property name="JDBC.ConnectionURL"
value="jdbc:mysql://localhost/test"/>
<property name="JDBC.Username"
value="root"/>
<property name="JDBC.Password"
value="blah"/>
</property>
</dataSource>
</transactionManager>
</sqlMapConfig>
```

① 提供用于验证的 DOCTYPE 和 DTD

② 提供内置的事务管理器的名称


```
</dataSource>
</transactionManager>
<sqlMap resource="SqlMap.xml" /> ③ 提供SQLMaps
</sqlMapConfig>
```

你可能已经猜到，我们正是在此配置文件中告诉iBATIS如何连接数据库，以及获取哪些SQL Map文件。由于这是一个XML文档，我们需要提供doctype和DTD用于验证①。SIMPLE是一个iBATIS内置事务管理器的名字②。正是在这里你为这个事务管理器提供JDBC驱动程序的名称、JDBC URL以及允许你连接到数据库的用户名（username）和密码（password）。然后将提供你的SQLMaps③。此例中，我们只有一个SQL映射文件，但是你可以想要多少就提供多少个。在该文件中还可以做一些其他设置，下一章将详细介绍。

现在你已经看到了主配置文件，下面来看一下SqlMap.xml文件（见代码清单2-6）。这个文件包含了我们将要运行的SQL语句。

代码清单2-6 曾经是最简单的SQL映射

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE sqlMap PUBLIC "-//ibatis.apache.org//DTD SQL Map 2.0//EN"
    "http://ibatis.apache.org/dtd/sql-map-2.dtd">

<sqlMap>
  <select id="getAllUsers" parameterClass="string"
        resultClass="hashmap">
    SELECT * FROM USER_ACCOUNT WHERE GROUPNAME = #groupName#
  </select>
</sqlMap>
```

代码清单2-6的XML代码中，我们接受了一个String类型的参数（parameterClass）作为GROUPNAME参数，并且把结果（resultClass）映射到了一个hashmap上。

注意 我们并不推荐使用Map（例如HashMap和TreeMap）作为域模型，但是这样做确实显示了iBATIS所提供的映射的灵活性。不必总是映射到JavaBean——也可以直接映射到Map或者基本类型。

无论你是否相信，你现在已经看到了使用iBATIS所需要的所有代码和配置。我们故意将它分散开以便于印刷，但是即使如此，这些代码总共也只有大约50行（包括Java和XML）。而且更重要的一点是，这50行代码中有45行都是有关配置的，它们在应用程序中只需要编写一次即可，而不需要针对每条语句都写一次。正如你在前面小节中所见到的那样，使用JDBC最终需要对每条SQL语句都编写50行或者更多的代码。

2.5.4 构建应用程序

通常当构建一个大型应用程序时，你都会使用一些类似于Ant的工具来简化构建过程。由于

本例中我们只有一个类，因此就不需要麻烦地为它构建一个Ant脚本了。为了编译此应用程序，只需要在类路径上添加两个JAR文件：ibatis-common-2.jar和ibatis-sqlmap-2.jar，因此只要用命令行把它们输入给Java编译器即可：

```
javac -classpath <your-path>ibatis-common-2.jar;  
<your-path>ibatis-sqlmap-2.jar Main.java
```

当然，以上的代码应该在同一行中输入，并且应该用JAR文件的实际路径来代替<your-path>。如果一切都顺利的话，编译器应该不会产生任何输出到屏幕上，而只是在当前目录下创建一个Main.class文件。

2.5.5 运行应用程序

在运行此应用程序时，我们还需要另外一些JAR文件，但也不是那么多。为了运行我们的应用程序，只需要在类路径上添加以下JAR文件即可：ibatis-common-2.jar、ibatis-sqlmap-2.jar、commons-logging.jar以及JDBC驱动程序（在此例中，这个驱动程序就是mysql-connector-java.jar），因而下一步应该输入以下这条命令：

```
java -classpath <your-path>;mysql-connector.jar;commons-logging.jar;  
ibatis-common-2.jar;ibatis-sqlmap-2.jar;. Main
```

同样地，在编译时以上代码必须也在同一行中，同时应该用系统中的实际路径来代替<your-path>。

这个程序运行之后会告诉你选择了多少条记录，然后将它们以一种以较粗糙的格式输出，类似如下：

```
Selected 2 records.  
{USERID=1, USERNAME=LMEADORS, PASSSSWORD=PICKLE, GROUPNAME=EMPLOYEE}  
{USERID=2, USERNAME=JDOE, PASSSSWORD=TEST, GROUPNAME=EMPLOYEE}
```

iBATIS框架被设计得非常灵活。它可以是一个非常轻量且简单的框架，只执行SQL并且返回数据，当然也可以被用来做其他更多的工作。

这种灵活性的关键之一在于对框架的合理配置。在下一章中，我们将讨论两种主要的配置文件类型，然后研究通过使用配置来处理复杂情况的一些模式。

说明 配置文件都是标准的XML文档。这意味着如果你有一个现代的XML编辑器，就可以用DTD（Document Type Definition，文档类型定义）来验证你的文档，有时候甚至可以在编辑过程中提供代码完成功能。

现在你已经看到了最简单形式的iBATIS。在继续讨论其他内容之前，先来讨论一下iBATIS的未来发展方向，这样你在使用它时就可以更加有信心。

2.6 iBATIS 未来的发展方向

在过去的几个月中，iBATIS已经获得了巨大的发展动力。结果表现为：iBATIS小组发展壮大，产品得到了改进，我们也开始讨论支持新平台的问题了。下面将详细地讨论iBATIS未来的发展方向。

2.6.1 Apache 软件基金会

近期，iBATIS已经成为了Apache软件基金会（Apache Software Foundation，ASF）的一部分。之所以选择转向Apache，是因为我们相信他们的使命并且尊重他们的态度。Apache绝不仅仅是一堆服务器和基础设施的组合，它是一个系统，是开源软件真正的家。Apache更关注软件周边社区^①，而不是软件背后的技术，因为如果没有社区，软件就是一个死的项目。

对于iBATIS用户来说这意味着iBATIS并不是由某个单独的团体来指导，也不是依赖于某个单独的团体。iBATIS不属于任何个人——它属于整个社区。Apache能够始终保护iBATIS，并且确保它维持正确的方向。也就是说，Apache许可并没有像某些许可（如GPL）那样，限制对开源软件的使用。Apache许可并不具有传染性^②，这意味着可以在商业环境中自由地使用这些软件，而不用担心需要遵守许多不合理的条件。

虽然Apache并不关注基础设施，但是它们确实拥有一些非常好的基础设施。目前iBATIS使用Subversion（SVN）进行版本控制，使用Atlassian的JIRA来跟踪问题，用Atlassian的Confluence来协作撰写wiki文档，并且使用Apache的邮件列表服务器在开发团队、用户以及一般社区之间进行通信。

Apache拥有保护iBATIS所需要的一切，并且可以确保：只要仍然有人想要使用iBATIS，它就会在那儿。

2.6.2 更简单、更小且依赖性更少

和某些框架不同的是，iBATIS工程并不期望分支出新的领域，也没有任何野心要解决所有问题。iBATIS是一个目标非常集中的项目，每一次发布新版本，我们都期望它更小、更简单，并且更少地依赖第三方库。

我们相信iBATIS还有许多创新的空间。iBATIS可以从很多新的技术和设计方法中获益，以便使其配置更加简练，也更容易使用。例如，C#和Java都内置了属性[attribute，也可称为“注解（annotation）”]功能。在未来的版本中，iBATIS就有可能利用此功能来减少配置框架时所需的XML代码的数量。

① 即使用者社区。——译者注

② 传染性（viral）与copyright（版权）对应的copyleft有些类似。它是一种将程序变为自由软件的通用方法，同时也使得该程序的修改和扩充版本也成为自由软件。——编者注

在为iBatis开发支持工具方面也还有许多事情可做。iBatis的设计使得为其开发像IDE这样的图形化工具非常容易。也可考虑创建支持从数据库模式（database schema）中直接生成iBatis配置文件的工具，其实这一点上已经有相应的工具可以用了。可以在我们的网页<http://ibatis.apache.org>上看到一些工具的示例。

2.6.3 更多的扩展点和插件

iBatis已经有了许多扩展点。第12章将深入探讨有关扩展的问题。你可以实现自己的事务管理器、数据源、高速缓存控制器（cache controller），等等。但是我们期望iBatis更易于扩展。我们希望将JDBC架构的几乎每一层都设计为可扩展的，这将意味着你可以实现自己的ResultSet处理器和SQL执行引擎。这将帮助我们支持以私有方式操作的更复杂的系统或者遗留系统。它也将使开发人员能够充分地利用特定数据库或应用程序服务器的定制特性。

2.6.4 支持更多的平台和语言

正如你在第1章和第2章中所看到的那样，我们已经在.NET和Java中讨论了iBatis。本书的其余部分将主要关注Java版iBatis的API，但是大部分的信息都是可以转化成.NET平台的。另外，我们还将附录中详细讨论iBatis.NET。实际上iBatis也已经有Ruby实现了，但是Ruby是一种完全不同的语言，因此用Ruby实现的iBatis也有很大的不同。这里就不讨论其Ruby实现了。

除了Java和C#之外，iBatis团队还讨论过用其他的语言来实现iBatis，这些语言包括PHP 5和Python。我们相信iBatis对于几乎任意一种只能使用底层数据库API和高层对象/关系映射工具的平台，都可以做出巨大的贡献。iBatis可以填充中间地带，并且允许你始终用一致的方式来实现所有的应用程序。

我们也讨论了要起草一份规范，使用户可以更容易地把iBatis移植到不同的平台上，并且确保合理的一致性。当然，我们既希望iBatis能够充分利用特定语言和平台的特性，也期望它们能有一定程度的相似性，以便确保它们都能被称为iBatis，并且能够被熟悉另一种语言中的iBatis的开发人员辨认出来。

2.7 小结

在本章中，你已经了解到iBatis是一个独特的数据映射器，它使用一种被称为SQL映射的方式将对象持久化到关系数据库中。iBatis在Java和.NET两个平台上的实现是一致的，而在应用程序中始终采用一种一致的持久化方式也具有非常重要的意义。

在本章中，你还学习了iBatis是如何工作的。通常，在后台iBatis将运行已编写好的JDBC或ADO.NET代码，如手工编写这些代码将非常难以维护。我们发现，和JDBC相比，iBatis代码更加简洁，也更易于编写。

我们还讨论了为什么iBatis对于小型和大型企业级应用程序来说都是非常合适的框架，尽管

它的设计很简单。iBATIS具有很多支持企业级持久化需求的特性。如行处理器等特性可以允许高效地处理大型数据集（每次处理一条记录），这样就保证了不会耗尽系统的内存。

我们还讨论了iBATIS区别于其他竞争产品的一些很重要的特性，同时我们还构造了一个使用iBATIS的简单示例。这些特性包括：

- 简单性——iBATIS被广泛认为是最简单的一种持久化框架。
- 生产效率——简洁的代码和简单的配置使得使用iBATIS所需的代码量可以减少到相应JDBC代码的62%。
- 性能——架构级性能增强，如联合查询的使用等，加速了数据的处理。
- 关注点分离——iBATIS改进了应用程序的设计方式以确保未来的可维护性。
- 分工明确——iBATIS可以帮助细化分工，使得工作团队能够充分发挥个人的专长。
- 可移植性——iBATIS可以用任何具有完备功能的编程语言来实现。

宣传了iBATIS的优点之后，我们也承认iBATIS并不是一颗“银弹”，因为没有任何框架是万能的。我们讨论了在哪些情况下iBATIS不会是最完美的方案。例如，当你始终拥有对应用程序和数据库的完全控制权时，采用一个成熟的对象/关系映射工具可能就是更好的选择。另一方面，如果你的应用程序主要处理动态生成的SQL代码，那么直接使用JDBC可能就是更好的选择。我们也提到了iBATIS主要是为关系数据库而设计的，因此如果你使用的是平板文件、XML、Excel电子表格或者其他任何非关系数据库技术，那么最好还是使用另一种完全不同的API。

最后，我们以对iBATIS未来发展的讨论结束本章。关于iBATIS的未来，开发团队有很多非常好的设计目标，同时Apache软件基金会将会确保始终有一个充满活力的社区能够支持iBATIS未来的发展。

Part 2

第二部分

iBATIS 基础知识

正如你在本书第一部分所了解的那样，iBATIS 的基本思想就是简单。如果你熟悉 JDBC、XML 和 SQL，那么几乎就没有什么其他需要学习的了。此部分将为你集中展现 iBATIS 的基本特征，包括安装、配置、语句和事务。听起来似乎有许多东西要学习，但是 iBATIS 实际上已经大大简化了这些领域中的常见难题。通过对此部分中 5 章内容的学习，你将能够迅速熟悉和运行 iBATIS。

本 部 分 内 容

- 第 3 章 安装和配置 iBATIS
- 第 4 章 使用已映射语句
- 第 5 章 执行非查询语句
- 第 6 章 使用高级查询技术
- 第 7 章 事务
- 第 8 章 使用动态 SQL

第3章

安装和配置iBATIS

本章内容

- 获取iBATIS
- iBATIS与JDBC
- iBATIS配置基础

iBATIS的安装过程快而简单。由于iBATIS是一个库，而非应用程序，因此可以说不存在什么安装过程，但是为了能在应用程序中使用iBATIS，你确实需要完成一定的步骤。

如果熟悉Java和JDBC，你可能只需要阅读下面一两小段文字，就可以获得安装和运行iBATIS所需的所有信息。不过，为了以防万一，我们仍然会概要地告诉你完整的安装过程，本章的其余部分将给出有关安装过程的更多细节内容。

要获取iBATIS，你有两种选择：其一是下载一份其二进制发布（binary distribution），然后将它解压到某个目录下；或者从Subversion资源库（Subversion repository）中签出一份iBATIS源码的副本，然后自己构建（build）它。无论选择哪种方式，完成之后你都将获得一组相同的文件。

拥有这组文件后，只要将其中应用程序需要的JAR文件添加到应用程序的类路径中即可。如果使用的是JDK 1.4或更高版本，那么只需以下两个文件：

- ibatis-common-2.jar——共享的iBATIS类^①。
- ibatis-sqlmap-2.jar——iBATIS的SQL映射类。

这两个JAR文件中包含了最重要的iBATIS功能，因此对于使用iBATIS的几乎所有应用程序，有这两个文件基本上就足够了。注意，iBATIS没有任何必须依赖的第三方库。这样就可以避免因依赖其他框架而造成的版本冲突——对于任何在过去曾经遭受过版本冲突痛苦的人来说，这都是一个天大的喜讯。此外，iBATIS还有另外一些可选的JAR文件，可用它们获得其他一些特征，有关此方面内容将在本章稍后讨论。

① 在阅读本书时，common这个JAR文件可能已经和sqlmap JAR文件合并成单个的JAR文件了，这使得配置和使用iBATIS的过程更加简单，只需在类路径中添加对一个JAR文件的引用即可。——译者注

以上就是你需要的全部信息了！现在可以在你的应用程序中开始使用iBATIS了。当然，如果对于你来说以上信息已经足够详细，可以直接跳到3.3节，察看为使用iBATIS的其他特性而需要的其他依赖关系；或者跳到3.6节，学习如何配置iBATIS。如果以上信息你还是觉得不够详细，需要更多的信息，下面两节将给出更详细的描述。

3.1 获得一份 iBATIS 发布

如前所述，获得一份iBATIS发布（distribution）有两种方式：可以下载一份即可用的预构建的二进制发布；也可以从Subversion资源库中获取iBATIS源代码，然后从源代码构建该发布。这两种方式都可以得到相同的结果——一个可用于将iBATIS添加到你的应用程序中的完整发布，以及必要时提供可用的调试信息的完整源代码。

3.1.1 二进制发布

要开始使用iBATIS，最迅速也最容易的一种方式就是直接下载一份iBATIS二进制发布。因为二进制发布已经预构建，所以下载完后只需要解压它，就可以开始使用了。

说明 iBATIS二进制发布目前可以从网址<http://ibatis.apache.org>上获得。

iBATIS二进制发布包含了使用iBATIS所需的所有预编译的JAR文件，可用于构建iBATIS的所有相关的Java源代码，以及基本文档。

3.1.2 从源代码构建

如果对增强iBATIS框架或者修正其bug感兴趣，或者只是想从源代码中构建iBATIS框架，以便准确了解所获得的内容，那么可以从Subversion资源库中签出一份iBATIS的副本，然后自己从源代码构建一份iBATIS框架。第12章将深入讨论iBATIS框架扩展以及相应的构建问题，因此本小节介绍相对较为简略，但是依然提供了足够的信息。

说明 此处所指的Subversion（简称SVN）资源库是一个版本控制系统，所有Apache的新项目都使用它。Subversion实际上是过去许多开源项目所使用的版本控制系统（Concurrent Version System, CVS）的替代产品。SVN的目的就是提供一个环境，使得我们在修改框架时无需担心源代码版本丢失（因为每个开发人员都有一份自己的副本，而服务器则拥有很多副本）。如果想更多地了解Subversion，请参见其主页<http://subversion.tigris.org/>。
iBATIS的Subversion资源库目前的网址是<http://svn.apache.org/repos/asf/ibatis/>。

1. 深入SVN资源库

为构建整个iBATIS发布，Subversion资源库中同时包括用于Windows的批处理文件和用于

Linux的shell脚本。这就是说，一旦获得了源代码，需要做的就只是安装JDK并正确地设置好JAVA_HOME环境变量，然后运行Ant脚本即可。

iBATIS在构建时需要一些文件以满足其编译时的依赖关系，这些文件或者以JAR的形式或者以stub类的形式存在于资源库中。资源库的顶级目录结构如表3-1所示。

表3-1 来自Apache Subversion版本控制系统的iBATIS源代码目录结构

目 录 名	用 途
build	本目录中存放用于构建整个iBATIS框架的Ant脚本，以及构建过程生成的一切文件
devlib	此目录中存放当前Apache项目编译时所需的所有库文件
devsrc	此目录中存放当前Apache项目编译时需要但发布时不可用的文件，以及那些因为体积过大而不适合放入资源库的文件。对这些文件，我们在资源库中对其进行版本控制（即不控制），其实可以将它们理解为API，它们的存在仅仅是为了满足构建iBATIS的需要
doc	此目录中存放当前项目的文档
javadoc	JavaDoc是一个用于从源代码的doc注释中生成HTML格式的API帮助文档的工具，此目录是放置构建过程所需示例配置文件的地方
src	此目录中包含框架的所有实际代码。在第12章中你将更详细地看到整个iBATIS框架的结构
test	单元测试是检验代码正确性的一种方式。iBATIS框架使用JUnit在构建过程中自动运行单元测试。项目在构建过程中将运行这些测试并且基于测试结果生成测试报告。此目录包含框架的所有单元测试的源代码
tools	此目录包含那些在iBATIS的使用过程中非常有用的工具。例如，你能够在此目录中找到Abator

2. 运行build文件

要构建iBATIS，你需要运行build.bat批处理文件（针对Windows）或build.sh脚本文件（针对Linux或Macintosh）。构建过程包括如下步骤：

- (1) 清空目标文件夹，以便在随后的构建过程中放入构建得到的文件。
- (2) 编译所有的源代码。
- (3) 编织（instrument）编译后产生的类。
- (4) 运行单元测试。
- (5) 构建单元测试报告和覆盖率报告。
- (6) 构建JavaDoc生成的文档。
- (7) 构建用于发布的JAR文件。
- (8) 将以上步骤产生的所有文件打包为一个文件，即直接可用的二进制发布。

虽然现在你还不需要理解以上这些步骤，但我还是想对其简短地解释一下，这样当构建过程中出现的信息在屏幕上掠过时你也不会一无所知。

第一步用于确保以前的所有构建操作都不会对本次构建操作产生影响——每次构建都从零开始。

第二步的意图很明显（之前代码还没有编译），但第三步可能就让人有些费解了。第三步中我们用了一个叫做EMMA的覆盖率追踪工具复制并编织^①刚才编译得到的代码，后面将会对此作简短解释。

接下来，我们将对框架的各个组件运行这些单元测试。正如我们之前提到的，单元测试将在较低的层面上验证框架组件是否如预期那样运行（如，以几个已知的输入调用某个方法，然后验证其结果是否等于预期值）。

兼容性测试用于确保那些仍在使用本框架1.x版本的使用者能够在不需重写任何现有代码的情况下自然地过渡到2.x版本替换掉当前版本。

构建过程中产生的报告用于提供两类信息：第一，JUnit报告显示运行了哪些JUnit测试以及这些测试是否成功。这些报告的入口是reports\junit\index.html（相对于build目录），当你对框架做了某些修改并且想知道测试是否成功时，这份报告非常有用。位于reports\coverage\coverage.html的第二份报告，显示已运行的测试的覆盖率。代码覆盖率（code coverage）在软件测试中用于描述某程序中源代码被测试的程度。换句话说，它可以显示测试的有效程度。覆盖率报告给出了以下4个统计值：

- 类覆盖率（class coverage）——哪些类已经被测试了？
- 方法覆盖率（method coverage）——哪些方法已经被测试了？
- 代码块覆盖率（block coverage）——哪些代码块已经被测试了？
- 代码行覆盖率（line coverage）——哪些代码行已经被测试了？

如果确实对框架做了修改，请一定要检查测试报告以确保现有测试都取得了成功，确保单元测试已被修改，足以测试你所做的改变。一旦完成了这些工作，那么你就可以放心地将修改上传到Apache问题跟踪系统（JIRA）中，如果修改的确有必要被添加到框架中，那么项目负责人将在今后的某次发布中引入你的修改。

好，既然已构建了一份发布，那么你到底得到了些什么呢？继续阅读下去，谜底即将揭晓。

3.2 发布中包含的内容

不管你是如何获得这份发布的（通过解压或者自己构建），最终都会得到相同的一组JAR和ZIP文件，共计7个。JAR文件是一种Java档案文件，大多数Java库都以这种形式发布——JAR文件实际上就是一个包含了附加信息的ZIP文件。表3-2给出了iBATIS发布中最重要的文件。

^① Instrumentation是一种为大多数测试覆盖率工具采用的代码统计技术。它在产品代码的关键位置插入统计代码，具体来说有两种方式：Class Instrumentation和Source Instrumentation。前者把统计代码插入编译好的.class文件，而后者则把统计代码插入源代码并编译成新的.class文件。本文提到的EMMA使用的显然是第一种技术。——译者注

表3-2 iBATIS发布中最重要的文件

文 件	用 途
ibatis-common-2.jar	该文件包含SQL映射框架和DAO框架中都要用到的公共组件。在不久的将来(也许是本书发行时), 这些公共类可能会连同当前ibatis-sqlmap-2.jar文件中的SQL映射类一起被合并为单个的JAR文件
ibatis-sqlmap-2.jar	该文件包含SQL映射框架的所有组件
ibatis-dao-2.jar	该文件包含DAO框架的所有组件
user-javadoc.zip	该文件包含项目的一组JavaDoc文档, 这组文档仅仅是所有文档的一个子集, 是特别对那些只使用框架而不会去修改框架的人裁减而得到的
dev-javadoc.zip	该文件包含项目的所有JavaDoc文档
ibatis-src.zip	该文件包含用于构建框架JAR文件的完整源代码

从iBATIS框架需要的依赖文件数目如此之少, 你就可以看出该框架是多么的轻便。其实, 此框架还有其他一些可选功能, 可以通过在你的项目^①中包含其他框架来启用, 例如cglib[®]——公共字节码增强框架。(下一节将详细介绍有关cglib框架的知识。)

3.3 依赖性

iBATIS还有一些其他特性, 你可能想要配置它们, 例如下文中即将看到的针对延迟加载的字节码增强功能。此外, 在本书的其他部分我们还将深入研究iBATIS框架的各种功能。iBATIS的这些特性大多需要其他第三方开源或商业软件包才能正常工作。为了启用这些附加特性, 需要满足它们的相关依赖, 而且要配置iBATIS, 才能使用它们。本节将简要概述这些特性。

3.3.1 针对延迟加载的字节码增强

字节码增强是一种可以根据你所定义的配置或其他规则在运行时修改代码的技术。例如, iBATIS框架允许将某些SQL查询和其他一些SQL查询关联起来。想象这样一个场景: 你有一个客户(Customer)列表、针对每个客户的一个订单(Order)列表(作为客户对象的一部分)以及一个订单项(LineItem)列表(作为订单对象的一部分)。在这种情况下, 你就可以定义一条SQL映射, 将以上所有列表都关联起来, 从而使得对这些列表的数据库加载能够自动完成, 当然, 数据加载只发生在该数据的确被我们的应用程序请求的时候。

说明 如果你熟悉ORM工具, 可能会觉得这种功能和ORM工具所提供的是一样的。诚然, 它们的功能是很相似, 但是iBATIS框架却更加灵活。ORM工具只允许你建立数据库表和视图之间的关联, 而iBATIS框架则允许建立任意数目的数据库查询(而不仅是数据库对象)之间的关联。

以上这个功能非常有用, 并且当存在关联查询时它可以帮你节省一些编写代码的工作量。然而, 如果你有1 000个客户, 每个客户都有1 000个订单, 同时每个订单中都有25个订单项, 那么

① 即使用iBATIS的项目。——译者注

② cglib是一个强大的、高性能的、高质量的代码生成框架。它可以在运行时扩展Java类或实现Java接口。——译者注

合并后的数据将由25 000 000个对象组成。毫无疑问，这个数据太大了，根本不可能同时把它们全部加载到内存中。

延迟加载就是用来处理类似这种情况的。iBATIS可以允许你只加载那些实际需要的数据。

因此，在上述示例中，你可以重新配置SQL映射以便延迟加载那些关联列表。如此一来，当用户查看客户列表时，只有这1 000个客户对象组成的列表保存在内存中。加载其他列表所需的信息始终可用，但是直到实际需要这些数据时才真正加载。也就是说，订单信息并不是从一开始就加载，而是等到用户单击某个客户查看其订单时才加载。此时，框架才加载被客户单击的1 000个订单组成的列表，其他数据则完全没有加载。如果此时用户单击列表中的某个订单，想了解更多相关信息，那么只有被选中的那份订单的25个订单项才会被加载。

因此，通过更改配置，我们就可以在一行代码都不需要修改的情况下，把要处理的对象数目从25 000 000降低到2 025。这意味着同初始配置相比，我们的应用程序在加载对象时现在只需消耗初始配置所费时间的万分之一。

3.3.2 Jakarta Commons 数据库连接池

由于iBATIS是一个用于简化应用程序与数据库交互的工具，所以很明显应用程序必须连接到某个数据库。按需创建新的连接是非常耗时的（在某些情况下，需要好几秒才能完成）。因此，iBATIS不使用这种方法，而是使用一个连接池，连接池始终处于连接状态并且可以为应用程序的所有用户共享。

很多软件开发商所提供的驱动程序都有支持连接池化版本，但是存在一个问题，即这些连接池的特性和配置同它们的实现一样各不相同。

Jakarta Commons Database Connection Pool (DBCP) 项目正是用于解决以上问题的，它是一个包装器 (wrapper)，可以使我们轻松地使用任意的JDBC驱动程序作为连接池的一部分。

3.3.3 分布式高速缓存

多用户环境下的数据高速缓存是一个非常复杂的问题。而在多服务器环境下高速缓存数据则更加复杂。

为了解决此问题，iBATIS提供了一个使用OpenSymphony cache (OSCache) 的高速缓存实现方案。OSCache可被配置为跨多服务器集群以提供可伸缩性和错误恢复支持。

现在我们已经对iBATIS中可配置的若干特性有所了解，下面讨论如何把iBATIS添加到应用程序中！

3.4 将 iBATIS 添加到应用程序中

一旦配置了iBATIS，要想在应用程序中使用它，接下来唯一需要做的就是将它（以及你选择的所有其他依赖关系）添加到编译时和运行时的类路径中。下面先来了解一下什么是类路径。

每个计算机系统在做某项工作时都需要以一种方式来查找完成该工作所需要的所有组件。正如Linux中的\$PATH变量和Windows中的%PATH%变量一样，Java也有一个路径用来查找其所需的组件，这个路径就被称为类路径（class path）。

在Java发展的早期，通常要设置名为classpath的环境变量。虽然现在仍然可以这么做，但这种做法非常不好，因为它会被系统中运行的每个Java应用程序所共享。

Java运行时环境（Java Runtime Environment，JRE）也有一个特殊的目录lib/ext可以使用，但是一般也不推荐这么做，除非在某些特殊情况下，因此此目录下的所有类都会被使用该JRE的所有应用程序共享。建议你不要将iBATIS放在该目录下。

那么该如何在应用程序中使用iBATIS呢？有以下两种方式。

3.4.1 在独立应用程序中使用 iBATIS

对于独立应用程序，可以写一段用于启动的小脚本，在其中设置类路径。这是一种为众多独立应用程序所采用的合理方法。例如，假设你有一个基于控制台的独立应用程序，那么在Linux中就可以使用-cp命令将iBATIS的JAR文件添加到类路径上，如下所示：

```
java -cp ibatis-sqlmap-2.jar:ibatis-common-2.jar:. MyMainClass
```

如果你是在某个应用程序服务器上使用iBATIS，请参考该服务器的相关文档，以了解如何正确地iBATIS添加到你的应用程序的类路径上。

3.4.2 在 Web 应用程序中使用 iBATIS

当创建Web应用程序时，应该把iBATIS JAR文件放在Web应用程序的WEB-INF/lib目录下。

说明 你可能很想把iBATIS JAR文件放在一个可为所有Web应用程序共享的位置上。例如，对于Tomcat，这个位置可能是shared/lib或者common/lib目录。然而，把JAR文件放在共享位置上通常都很糟糕，除非由于特定原因而必须这么做（例如一个供JNDI数据源使用的JDBC驱动程序）。

所以说把JAR文件放在共享位置上很糟糕，原因之一就是这样做会使得对某个应用程序的升级变得更有风险。例如，假设你有10个应用程序共享1个JAR文件，在这样的环境中如果为了其中一个应用程序而需要升级这个JAR文件，那么你就必须对所有10个使用该文件的应用程序都进行测试。此外，还需要考虑有关类加载器（classloader）的问题。在Java中，两个不同的类加载器加载的字节码即使完全相同也被认为是两个不同的类。这就意味着静态变量是不能共享的，因此，如果试图将一个静态变量强制转换为到另一个，就会得到ClassCastException异常，即使它们属于同一类。另一个你可能遇到的有关类加载器的问题就是：类加载器如何寻找资源。例如，如果Tomcat的common/lib类加载器要加载iBATIS，那它就无法看到在使用该类加载器的Web应用程序中的配置文件了。

总之，如果你不是把iBATIS的JAR文件放在了Web应用程序的WEB-INF/lib目录下，那么无论出了什么问题都不要发邮件来提问了，请先把它移到WEB-INF/lib目录下再说吧。

3.5 iBATIS 和 JDBC

有关JDBC的深层次定义已经超出了本书的范围，但是我们仍将从较高的层次对它做一个简单介绍，以便为本书的其余部分打下良好的基础。

Sun公司的JDBC API是Java编程语言中的数据库连接标准。Java技术允许你实现“一次编写，随处运行（write once, run anywhere）”的诺言，而JDBC就是实现此诺言的一个重要部分，因为所有的数据库交互都采用JDBC来访问数据。

Sun公司对JDBC最大的贡献并不在于实现了它，而是在于定义了一套接口。软件开发商必须按照JDBC规定的接口来实现与他们的数据库的连接。如果他们不这么做，开发人员就不大可能使用他们的东西，因为至少在Java领域中，任何开发商锁定（vendor lock-in）的行为都被认为是反模式，这是所有开发人员极力避免的。

JDBC API从微软公司的ODBC API那里借鉴了许多概念，并且从1.1版（发布于1997年）开始就成为了Java的内核组件。1999年，JDBC API第2版发布；2002年，第3版发布。第4版目前正作为JCP-221的一部分处于设计中。

当前的iBATIS框架要求JDBC API至少为第2版，但是它也能够和第3版很好地兼容。

以上就是对JDBC的一个简略介绍，下面来了解一下在没有iBATIS的情况下使用JDBC需要注意的几个问题。

3.5.1 释放 JDBC 资源

使用JDBC时，很容易犯的一个错误就是获取资源之后忘记正确地释放它们。虽然垃圾收集进程最终可以释放掉这些资源，但是这种做法非常耗时，并且无法得到保证。如果这些对象没有被正确释放，那应用程序最终将会因为耗尽了资源而崩溃。iBATIS框架能帮助管理这些资源，从而减轻应用程序开发人员们的负担。如此一来，开发人员就不需要担心哪些资源需要保留，哪些需要释放，他们只需要关注所需的数据以及如何获取这些数据。当然，如果需要，开发人员仍然可以手工管理这些资源。

3.5.2 SQL 注入

另一个常见问题（在Web应用程序中更加普遍）就是SQL注入（injection）问题，也就是使得某个应用程序执行一些开发人员不曾预料到的SQL语句。例如，如果某个应用程序使用字符串拼接的方式来创建SQL语句，但是却没有正确地参数进行转义，那么恶意的用户就可以通过传

递一些特别的参数完全改变这个查询本来的意图。以下面这个查询为例：select * from product where id = 5。如果5直接来自用户，并且是被拼接到字符串select * from product where id = 之后的，那么用户就可以输入5或1=1，从而改变整个SQL语句的意图。如果用户输入的是5；delete from orders那就更糟糕了，因为该语句将忠实地按照输入值查找出那条特定的记录，然后就将你的订单表完全清除。因为灵活性往往与风险并存，所以iBATIS如果使用不正确也可能会使你的应用程序受到SQL注入攻击。然而，由于iBATIS使用了PreparedStatement，而PreparedStatement是不会受到此类攻击影响的，所以使用iBATIS可以使得你对应用程序的保护变得更加容易。

在iBATIS中，只有使用显式SQL字符串替换语法的那些语句才真正具有风险。考虑下面这个简单例子所示的语句。该语句允许动态的表名和列名：

```
SELECT * FROM $TABLE_NAMES$ WHERE $COLUMN_NAMES$ = #value#
```

在某些情况下，类似这样的语句非常的灵活并且很有用，但是它却使你的应用程序暴露给SQL注入，因此必须谨慎使用。这实际上不是一个iBATIS特有的问题，因为无论你怎么执行这样一条语句，总会遇到相同的问题。总是确保验证那些对动态构造的SQL语句有影响的输入。

3.5.3 降低复杂度

JDBC是一个非常强大的工具，但它同时也是一个非常低层的API。为了更好地了解iBATIS究竟适合于你的应用程序的哪个部分，下面先做一个类比。

几年前，要想用Java创建一个Web应用程序，必须从HTTP开始，编写一个监听端口并响应请求的应用程序。几年之后，Sun公司提供了servlet规范，从此我们可以将其作为起点，以便可以不必进行此类套接字、端口级别的编程开发了。又过了几年，Struts框架产生了，它使得利用Java进行Web开发又可以从一个更高的层次开始了。

今天大多数的Java开发人员都不会认真地考虑从HTTP协议（或者从servlet）开始编写基于Web的应用程序了。通常，他们会先找一个像Tomcat那样的servlet容器，然后使用Struts框架（或者其他类似于Spring或WebWork的东西）开始Web应用程序的开发。

回顾一下数据持久化技术的发展历史，在Java 1.0时，还没有JDBC规范。数据库开发人员必须自己解决如何通过其本地网络协议同数据库直接对话的问题。当Java 1.1发布时，JDBC从此进入了我们的视野，于是我们在处理数据库时也有了新的起点，再也不用从套接字和端口开始了。iBATIS框架对于数据库开发的作用就如同Struts对于HTTP一样。尽管可以通过打开数据库服务器连接端口的方式或者直接使用JDBC来编写应用程序，但是使用类似iBATIS这样的工具则要简单得多，因为你可以任由这些工具去处理Connection、Statement和ResultSet这些对象，而再也不用把这些对象混合在业务逻辑中了。

正如Struts所做的那样，iBATIS为你提供了一个抽象层，使得直接使用低层组件时的大量复杂操作被透明化了。但iBATIS并没有将它们从你的应用程序中完全移除，它只是帮助你避免处理它们，直到你确实需要这么做。

作为iBATIS能为你降低复杂度的一个例子，我们来看一下合理分配并确保JDBC连接释放的模式：

```
Connection connection = null;
try {
    connection = dataSource.getConnection();
    if (null == connection) {
        // kaboom
    } else {
        useConnection(connection);
    }
} catch (SQLException e) {
    // kaboom
} finally {
    if (null != connection) {
        try {
            connection.close();
        } catch (SQLException e) {
            // kaboom
        }
    }
}
```

以上将近20行的代码只是用来获取、使用然后正确地关闭一个Connection对象。除了Connection对象之外，要安全地使用Statement和ResultSet对象也需要同样的模式。只要想一想使用纯JDBC来实现诸如数据库连接、参数映射和结果映射、延迟加载以及数据高速缓存等这样的特性会有多么的复杂和繁琐，而iBATIS却能自动地为你处理所有这一切，你就会明白使用iBATIS可以帮你节省多少原本需要你费心费力去精心编写的代码。幸运的是，配置和使用iBATIS都十分简单，不信你可以阅读本章的其余部分以及接下来的几章。

3.6 配置 iBATIS (续)

第2章已经非常简单地介绍了如何配置iBATIS（如果你还没有阅读那部分也不用担心）。本节将通过创建一个SQL Map配置文件来告诉你如何构建iBATIS的基本配置。此文件是iBATIS的核心，如图3-1所示。

在此图中，SqlMapConfig文件位于最上方，我们在此文件中定义那些全局配置选项以及对各个独立SqlMap文件的引用。接下来，SqlMap文件则用于定义那些已映射语句（mapped statement），这些已映射语句结合应用程序提供的输入，来完成与数据库的交互。

下面详细考察一下如何使用这个配置文件。

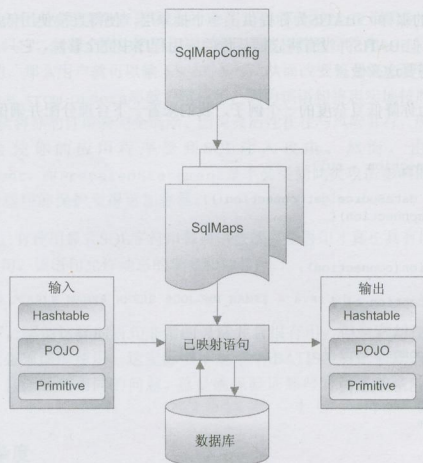


图3-1 iBATIS配置概念图 (SqlMapConfig在最上方)

3.6.1 SQL Map 配置文件

如图3-1所示的概念图，SQL Map配置文件（即图中的SqlMapConfig.xml文件）是iBATIS配置的核心（因此也称为主配置文件）。所有东西（从数据库连接到实际所用的SqlMap文件）都是通过此文件中的配置提供给框架的。

说明 主配置文件通常都被命名为SqlMapConfig.xml。虽然你不必使用这个名字，但本书还是将依照惯例使用它。

代码清单3-1给出了一个SQL Map配置文件示例，接下来的几小节将讨论它。

代码清单3-1 SqlMapConfig.xml

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE sqlMapConfig
PUBLIC "-//ibatis.apache.org//DTD SQL Map Config 2.0//EN"
"http://ibatis.apache.org/dtd/sql-map-config-2.dtd">
<sqlMapConfig>

  <properties resource="db.properties" />
```

在
配置。
章中，

3.6.2

通用。
同时将

可
性。这

便
称为资

据时引

JAR文

u
用于加

```
<settings
  useStatementNamespaces="false"
  cacheModelsEnabled="true"
  enhancementEnabled="true"
  lazyLoadingEnabled="true"
  maxRequests="32"
  maxSessions="10"
  maxTransactions="5"
/>

<transactionManager type="JDBC" >
  <dataSource type="SIMPLE">
    <property name="JDBC.Driver" value="${driver}"/>
    <property name="JDBC.ConnectionURL" value="${url}"/>
    <property name="JDBC.Username" value="${user}"/>
    <property name="JDBC.Password" value="${pword}"/>
  </dataSource>
</transactionManager>

<sqlMap resource="org/apache/mapper2/ii15/SqlMap.xml" />

</sqlMapConfig>
```

① 全局配置选项

② 事务管理器 (transaction manager)

③ 对各个Sql Map文件的引用

在下面的几小节中，我们将研究代码清单3-1中①处的全局配置项，以及②处的事务管理器配置。在接下来的3章（即第4章～第6章）中，我们将分析③处定义的已映射语句。之后在第7章中，回过头来详细讨论事务。

3.6.2 <properties>元素

<properties>元素允许在主配置文件之外提供一个名/值对列表，从而使得主配置文件更加通用。这对于应用程序部署非常有用，因为你可以把那些为所有环境共享的配置放在某个地方^①，同时将那些会根据环境而变化的值隔离在一个属性文件中。

可以使用两种方式來指定所用的属性文件，每种方式都对应于<properties>元素的一个属性。这两个属性分别是：

- resource——类路径上的一个资源（或文件）。
- url——统一资源定位符（uniform resource locator, URL）。

使用resource属性时，类加载器将会试图在应用程序的类路径上定位该资源。它之所以被称为资源（resource），就是因为我们使用类加载器来读取它。Java文档也会在以此种方式访问数据时引用这些资源，因为尽管这种数据可能只是本地文件系统中的一个文件，但也有可能是某个JAR文件中的一个输入项，甚至还有可能位于另一台电脑上。

url属性则是用java.net.URL类来处理的，因此这个属性的取值可以是该类所能理解的并可用于加载数据的任何有效的URL。

① 即主配置文件。——译者注

在之前的示例中，我们使用<properties>元素将数据库配置细节从XML文件中分离出来，并且将它们放在位于类路径上的一个简单的属性文件db.properties中。这种分离有很多好处，其中之一就是可以简化操作。假设db.properties文件包含以下4个属性定义：

```
driver=com.mysql.jdbc.Driver
url=jdbc:mysql://localhost/test
user=root
pwd=apple
```

我们使用下面这样一行代码（来自前一个示例）来引用此属性文件：

```
<properties resource="db.properties" />
```

最后，通过名字来引用这些属性，它使用了一种大多数人都熟悉的语法。

```
<property name="JDBC.Driver" value="${driver}"/>
<property name="JDBC.ConnectionURL" value="${url}"/>
<property name="JDBC.Username" value="${user}"/>
<property name="JDBC.Password" value="${pwd}"/>
```

看完以上这些代码，你可能会想：“啊呀，这看上去和其他任何工具使用属性的方式是多么一样啊！”

3.6.3 <settings>元素

<settings>元素就好像是装满了配置选项的摸彩袋（grab bag）。可以通过为<settings>元素添加若干属性来设置这些可选项。iBATIS有许多配置项可用，并且每个配置项对于这个SQL Map实例来说都是全局的。

1. lazyLoadingEnabled

延迟加载（lazy loading）是一种只加载必要信息而推迟加载其他未明确请求的数据的技术。也就是说，除非绝对需要，否则应用程序加载的数据越少越好。

在3.3节中，我们曾经讨论过一个示例，其中假设你有1 000个客户，每个客户有1 000个订单，每个订单上有25个订单项。同时加载所有这些数据将需要创建25 000 000个对象，并且要把它们都保存在内存中。使用延迟加载就可以把这个要求降低到大约2 500个对象，这个数字是原来的万分之一。

lazyLoadingEnabled配置项用于指定当存在相关联的已映射语句（6.2.2节将讨论它）时，是否使用延迟加载。其有效值为true或false，默认值为true。

在我们之前的样例文件SqlMapConfig.xml中，延迟加载是被启用的。

2. cacheModelsEnabled

数据高速缓存是一种用于提高程序性能的技术，它基于近期使用过的数据往往很快又会被用到这样一种假设，将近期使用过的数据保存在内存中。cacheModelsEnabled配置项用于指定是

不想
或fa

缓存

能。

说明

地禁

定名

已映
以及
Name
名空
不会

使用
它们

是指
我们

否想让iBATIS使用该技术。和<settings>元素中其他大多数配置项一样，它的有效值也是true或false。

在本书之前的示例中，数据高速缓存是启用的，这是该配置项的默认值。为了充分利用高速缓存技术，还必须为已映射语句配置高速缓存模型（有关此方面内容将在9.1节介绍）。

3. enhancementEnabled

enhancementEnabled配置项用于指定是否使用cglib中那些已优化的类来提高延迟加载的性能。其有效值仍然是true或false，默认值为true。

说明 cglib是一个运行时代码生成库，iBATIS可以使用它来优化其某些功能，例如JavaBeans特性的设置。同时，它可以允许你延迟加载具体类^①，这样就不需要为了能够延迟加载某个类而特意创建一个相应的接口了。可以从网址<http://cglib.sourceforge.net/>上获得cglib。和其他任何性能增强框架一样，你最好尽量避免使用它，除非确实需要。

在之前的示例中，我们允许了性能增强，但是如果cglib不在类路径上，性能增强仍会被悄悄地禁用。

4. useStatementNamespaces

useStatementNamespaces配置项用于告诉iBATIS，在引用已映射语句时，是否需要使用限定名（qualified name）。其有效值也是true或false，默认值为false。

也就是说，当你定义了SQL映射（详情请参考4.2.1节）之后，就用该SQL映射名来限定这条已映射语句。例如，假设你有一个名为Account的SQL映射，它包含名为insert、update、delete以及getAll的已映射语句。那么当你想插入一个账户且在主配置文件中启用了useStatementNamespaces配置项时，就必须用Account.insert这个名字来调用这条已映射语句。通过使用命名空间，你可以根据需要，创建任意数目名为insert的已映射语句（在其他映射文件中），而不会造成名字冲突。

虽然也可以使用类似insertAccount这样的名字来完成相同的事，但是在面对大型系统时，使用命名空间将更为有用，因为即使在已映射语句的组织毫无逻辑时，命名空间也可以帮助找到它们。

5. maxRequests (已废弃)

要理解maxRequests配置项，首先让我们定义一下什么是request。所谓request，即请求，就是指对数据库的一次SQL操作——例如插入、更新、删除，或者是调用存储程序。在这个示例中，我们修改了maxRequests的默认值512，将其缩小为32，这样我们的数据库一次最多只允许有32

① 即非接口类。——译者注

个请求处于活动状态。

6. maxSessions (已废弃)

会话 (session) 是一种线程级机制，用于跟踪关于一组相关事务和请求的信息。在这个例子中，任何时候都只允许10个会话，而不是采用其默认值128。

7. maxTransactions (已废弃)

顾名思义，事务 (transaction) 就是指数据库事务。同对maxRequests配置项的处理一样，我们也把活动事务的最大数目从默认的32减小到了5。

说明 以上3个配置项理解起来的确有点麻烦，但幸运的是，它们都被废弃。在iBATIS以后的版本中，你将不再需要手动配置它们。因此，在大多数情况下，你根本不需要去修改它们。默认设置对于大部分具有合理规模的系统都是可行的。如果确实要修改这些配置项，请确保maxRequests总是大于maxSessions，而maxSessions总是大于maxTransactions。一个简单的经验法则就是，总是保持它们间的比例值相同。同时还要注意：这3个配置项中没有一个会对连接池中的连接数或者由应用程序服务器负责管理的那些资源产生直接影响。

好，以上就是关于<settings>元素你需要知道的全部内容了。下面继续其他内容——我们几乎就要完成对SqlMapConfig.xml文件的探索了！

3.6.4 <typeAlias>元素

除非别无选择，否则没有人愿意输入像这样一长串的名字：org.apache.ibatis.jgamestore.domain.Account。<typeAlias>元素允许定义别名，这样就可以直接用Account来代替刚才的那个冗长的全限定类名 (fully qualified class name) ①了。

到目前为止，我们还没有在任何例子中实际使用过别名，但是很快就会用到它的。为了示范如何使用它，下面给出一个稍后就将在我们的SQL Map中使用的别名。

为创建此别名，需要按以下方法插入一个元素：

```
<typeAlias alias="Account"
  type="org.apache.ibatis.jgamestore.domain.Account" />
```

当在配置过程中定义了Account这个别名之后，就可以在任何时候使用它了，每当你必须提供一个数据类型时，都可以使用全限定类名或者别名来告诉iBATIS你想使用哪个数据。

iBATIS框架为若干类型定义了别名，以免除开发人员必须手动添加这些别名的麻烦，详见表3-3。

① 即在一个类名前添加了完整的层层包名作为限定。——译者注

表3-3 可以使你避免输入长类名的内置类型别名定义

别 名	类 型
事务管理器别名	
JDBC	com.ibatis.sqlmap.engine.transaction.jdbc.JdbcTransactionConfig
JTA	com.ibatis.sqlmap.engine.transaction.jta.JtaTransactionConfig
EXTERNAL	com.ibatis.sqlmap.engine.transaction.external.ExternalTransactionConfig
数据类型	
string	java.lang.String
byte	java.lang.Byte
long	java.lang.Long
short	java.lang.Short
int	java.lang.Integer
integer	java.lang.Integer
double	java.lang.Double
float	java.lang.Float
boolean	java.lang.Boolean
date	java.util.Date
decimal	java.math.BigDecimal
object	java.lang.Object
map	java.util.Map
hashmap	java.util.HashMap
list	java.util.List
arraylist	java.util.ArrayList
collection	java.util.Collection
iterator	java.util.Iterator
数据源工厂类型	
SIMPLE	com.ibatis.sqlmap.engine.datasource.SimpleDataSourceFactory
DBCP	com.ibatis.sqlmap.engine.datasource.DbcpDataSourceFactory
JNDI	com.ibatis.sqlmap.engine.datasource.JndiDataSourceFactory
高速缓存控制器类型	
FIFO	com.ibatis.sqlmap.engine.cache.fifo.FifoCacheController
LRU	com.ibatis.sqlmap.engine.cache.lru.LruCacheController
MEMORY	com.ibatis.sqlmap.engine.cache.memory.MemoryCacheController
OSCACHE	com.ibatis.sqlmap.engine.cache.oscache.OSCacheController
XML 结果的类型	
Dom	com.ibatis.sqlmap.engine.type.DomTypeMarker
domCollection	com.ibatis.sqlmap.engine.type.DomCollectionTypeMarker
Xml	com.ibatis.sqlmap.engine.type.XmlTypeMarker
XmlCollection	com.ibatis.sqlmap.engine.type.XmlCollectionTypeMarker

内置类型别名确实可以帮助我们节省大量时间，但是请记住，也可以定义自己的别名来进一步简化类名的输入。

3.6.5 <transactionManager>元素

由于iBATIS就是为了使数据库访问变得更加简单而设计的，因此它将为你处理所有的数据库事务。考虑到第8章中将详细讨论事务管理，所以在此我们只需要知道：使用iBATIS时，有哪些事务管理器实现类是可用的。可以考虑使用iBATIS预定义的那些事务管理器。<transactionManager>元素的type属性就是用于指定应该使用哪个事务管理器的。iBATIS提供了若干内建事务管理器实现，如表3-4所示。

表3-4 内置的事务管理器

名 字	描 述
JDBC	用于提供简单的基于JDBC的事务管理。大多数情况下，使用这个事务管理器就足够了
JTA	用于在你的应用程序中提供基于容器的事务管理
EXTERNAL	用于提供非事务管理，并假定管理事务的是应用程序，而不是iBATIS

说明 你可能已经发现了这些事务管理器的名字在表3-3中出现过。它们只是用于代表实现它们的全限定类名的类型别名。这在iBATIS配置文件中是一种常见的模式。

对于事务管理器，另一个可用的设置就是commitRequired属性。该属性可以被设置为true或者false，其默认值为false。它主要用于那些要求在释放某个连接之前必须提交（commit）或者回退（rollback）的情况。

对于某些操作（例如选择和存储过程调用），通常并不需要事务，因此一般会将其忽略。某些数据库驱动程序（例如Sybase驱动程序）不会轻易释放连接，直至该连接的每个已启动事务都已提交或回退。在这些情况下，就可以使用commitRequired属性来强制释放连接，即使在那些通常需要在一个事务中释放却没有释放的情况。

1. <property>元素

每个事务管理器都可以有不同的配置项。因此，iBATIS框架允许使用<property>元素来指定任意数目的名值对以提供给相应的事务管理器实现类。

2. <dataSource>元素

在Java中，使用连接池的标准方法就是通过javax.sql.DataSource对象。事务管理器的<dataSource>元素有一个类型属性，用来告诉iBATIS它应该为其数据源工厂实例化并使用哪个类。<dataSource>元素的名字容易使人产生误解，因为实际上它并不是用来定义DataSource，而是用于定义DataSourceFactory实现类，iBATIS将用这个实现类来创建实际的DataSource。

iBATIS具有3种数据源工厂实现类，每一个实现类及其简单描述都列在表3-5中。我们将在第8章中详细讨论它们。

表3-5 数据源工厂

名 字	描 述
SIMPLE	即简单的数据源工厂。它用于配置那种内置有简单连接池的数据源，因此除了实际的JDBC驱动程序之外，该数据源工厂所需的其他东西都包含在iBATIS框架中
DBCP	DBCP数据源工厂用于使用Jakarta Commons数据库连接池实现
JNDI	JNDI数据源工厂用于允许iBATIS共享通过JNDI定位的基于容器的数据源

类似于<transactionManager>元素，也可以在<dataSource>元素中内嵌<property>元素来传递任意数目的名值对以设置某特定数据源工厂的特定配置项。

3.6.6 <typeHandler>元素

iBATIS使用类型处理器 (type handler) 将数据从数据库特定的数据类型转换为应用程序中的数据类型，这样你就可以创建一个以一种尽可能透明的方式来使用数据库的应用程序。类型处理器本质上就是一个翻译器 (translator) ——它将数据库返回的结果集中的列“翻译”为相应Java Bean中的字段。

大多数情况下，这些类型处理器组件都非常简单，例如StringTypeHandler就只是调用结果集的getString方法并将其作为字符串返回。而有些情况下，则可能会有一些更复杂的转换要求。例如，假设数据库中并没有布尔数据类型，那你可能就会在数据库中用单个字符Y或N来代表true或false，但是如此一来你就必须在应用程序的类中将这样的字符转换为布尔值。

第12章将介绍更多有关如何创建自定义类型处理器以处理此类情况的技术，因此这里将不作更详细的讨论。为处理以上情况，需要创建两个类：一个自定义类型处理器和一个类型处理器回调类。

编写自定义类型处理器时，你需要告诉iBATIS如何以及何时使用它们。可以通过<typeHandler>元素并告诉iBATIS它需要在两个类型jdbcType和javaType之间转换哪些元素。此外，还需要回调类来管理类型处理器。

值得注意的是，在大多数情况下你都不需要编写自定义类型处理器。iBATIS已经预先创建了很多类型处理器，可以处理99%的情况。通常只有当你使用的数据库或数据库驱动程序非常奇特，连大部分数据库所通用的典型类型映射都不支持时，才需要编写自定义类型处理器。为确保应用程序尽可能简单，请尽量避免编写自定义类型处理器。

3.6.7 <sqlMap>元素

SqlMapConfig.xml文件的最后一部分就是我们配置<sqlMap>元素的地方。正是在此处你开始真正了解到iBATIS可以为你做的大量繁重的工作。

<sqlMap>元素实际上是此文件中最简单的元素之一，因为它只要求设置其两个属性中的一个。

如果想把SQL Map文件放在Java类路径上并且以这种方式引用它们，那么就on应该使用resource属性。通常，这是最简单的方式，因为SQL映射文件可以存储在JAR文件或WAR文件中，这时通过相对于类路径根目录的路径来引用它们就非常容易。

在其他一些情况下，你可能想要更显式地指定文件的具体位置。此时，可以使用url属性。该属性会使用java.net.URL类来解析文件的位置，因此可以使用任意的URL值，只要该值能够被java.net.URL类理解。

3.7 小结

安装iBATIS实际上非常简单。本章中讨论了获取iBATIS框架的两种方式，以及获得iBATIS框架之后如何安装它。由于iBATIS的设计目标之一就是易于使用，因此它几乎没有什么依赖性；同时，要扩展它以使用其他工具也非常的简单。

JDBC虽然是一套非常强大的API，但同时也是一套非常低层的API，因此要直接用它编写出强壮的代码是非常困难的。有了iBATIS，你就可以避开直接处理数据库组件的复杂性，而把更多的注意力放到你想要解决的业务问题上来。

本章研究了一个非常简单的iBATIS配置示例——很可能是世界上最简单的示例了！在第4章中，我们将进一步研究如何配置iBATIS，以及如何将它天衣无缝地结合到你的应用程序中去。

本章还研究了几个iBATIS配置文件。使用任何框架都必须深刻理解其原理。本章恰好提供了你所需要的对于原理的理解，或者说是一个快速参考指南。

可以在本章放置一张书签，以便你遇到问题时可以回过头来参考。即使你无法在本章中直接找到答案，你也应该能找到一个指向其他章的指针，在那里你就可以找到想要的答案。

在以后几章中，我们将会遇到的Java API其实范围十分有限，但是根据你对框架的不同配置，相应的框架也会表现出不同的行为。我们还将更加详细地讨论每一个配置项，并且把它们放在更多的上下文和代码示例中。

第 4 章

使用已映射语句

4

本章内容

- JavaBean
- 使用iBATIS API
- 已映射语句
- 使用参数映射和结果映射

在前面几章中，我们介绍了如何配置iBATIS Sql Map，并且给出了一个已映射语句（mapped statement）的示例。你现在应该已经有了一定的基础，下面就是讨论如何为应用程序构建更多的数据库访问层。

本章和第5章将更详细地研究已映射语句，并且讨论如何使用SQL映射来创建它们。本章首先从总体上探讨已映射语句以及要使用它们所必需了解的知识。然后，将解释如何使用返回具有类型的对象（即JavaBean）的已映射语句从数据库中获取数据；以及如何传递进参数以限制返回的数据。在第5章中，你还将学习如何使用已映射语句来更新数据库。

4.1 从基础开始

开始使用iBATIS之前，首先需要了解几个概念。你需要初步了解如何使用JavaBean以进行Java开发。同时还需要了解iBATIS为你提供了哪些可用的SQL语句类型，以及用于执行这些语句所需要的API。

如前所述，iBATIS并不是一种O/RM工具，它是一种查询映射工具。正因如此，我们将要介绍的API并不是你唯一可用的API。可以使用任何其他API来创建应用程序将要使用的bean或者访问数据库。因此，如果发现无法使用iBATIS完成某件事时，你完全可以直接使用JDBC API，而无需过多顾虑所带来的影响。当考虑iBATIS究竟可以为我们带来了什么的时候，请牢记以上所说的这种自由性——尽管iBATIS不可能解决所有的问题，但是它确实试图为我们简化绝大部分数据访问琐事。

4.1.1 创建 JavaBean

尽管iBATIS并不要求一定要使用JavaBean，但是强烈建议在大多数（但不是全部）情况下都使用它们。如果你是第一次接触Java，不用担心，创建JavaBean实际上非常简单：JavaBean就是一个可复用组件，你可以开发它，然后将其组合起来用于创建更加复杂的组件和应用程序。JavaBean规范（整整有114页）可以从Sun公司的网页上免费下载。只要在网页<http://java.sun.com>上进行简单搜索，即可得到该文档。

由于篇幅所限，我们将只对规范中同iBATIS密切相关的部分内容作简要介绍。

1. bean的构成

本质上而言，JavaBean规范就是一组规则，用于定义Java开发可使用的组件。这些规则使得工具创建者（例如那些创建iBATIS的人）可以知道如何与我们在应用程序中所使用的组件进行交互。规范可以看作是框架开发人员和应用程序开发人员的一个中间层公共语言。

JavaBean规范中唯一应用程序于iBATIS的规则就是关于特性命名的。在JavaBean中，特性名由一对方法定义，这对方法即规范中所谓的存取方法（accessor method）。以下就是为一个名为value的特性创建的存取方法：

```
public void setValue(ValueType newValue);  
public ValueType getValue();
```

这两个存取方法定义了一个以小写字母v开头的名为value的简单特性。Java bean特性总是以一个小写字母开始，很少有例外。以上这两个方法的类型应该始终保持一致。即是说，如果设置方法（setter）接收长整型数据，而获取方法（getter）返回一个整型数据，就会出问题。因此请始终确保它们的类型一致。

具有多个单词的特性名通常以骆驼命名规则（camelCase）的模式来命名，即用大写字母来区分单词：

```
public void setSomeValue(ValueType newValue);  
public ValueType getSomeValue();
```

为JavaBean创建特性时，必须牢记：缩略语通常被视为一个独立的单词，而不是单个的字母。例如，缩略语URL就应该对应为名为url的特性，而getUrl和setUrl则用于访问该特性。

除了关于缩略语的特殊规定之外，规范中另一个特别的地方就是，第二个字母为大写的特性名要区别对待。如果特性名的第二个字母是大写的，那么该特性对应的方法中获取或设置部分之后的名称将用作特性名，并且大小写保持不变。表4-1中阐明了这条令人费解的规则（表4-1中同时还给出了其所列的所有特性的获取方法和设置方法）。

boolean类型（基本类型）的特性允许获取方法使用isProperty，但是如果其类型是Boolean（对象类型，或盒装版本），那么就必须使用标准的getProperty名称。

表4-1 样例JavaBean特性名和方法

特性名/类型	获取方法	设置方法
xcoordinate/Double	public Double getXcoordinate()	public void setXCoordinate (Double newValue)
xCoordinate/Double	public Double getXCoordinate()	public void setxCoordinate (Double newValue)
XCoordinate/Double	Public Double getXCordinate()	public void setXCordinate (Double newValue)
Xcoordinate/Double	不允许	不允许
student/Boolean	public Boolean getStudent()	public void setStudent (Boolean newValue)
student/boolean	public boolean getStudent()	public void setStudent (boolean newValue)

已索引的特性就是指那些代表值数组的特性。例如，假设有名为Order的bean，你可能想要为与这个Order相关联的一组OrderItem对象设置一个已索引特性。根据规范，它所对应的获取和设置方法的签名将如下所示：

```
public void setOrderItem(OrderItem[] newArray);
public OrderItem[] getOrderItem();
public void setOrderItem(int index, OrderItem oi);
public OrderItem getOrderItem(int index);
```

根据我们的经验，超载bean特性（如规范所建议，也如已索引特性的示例所示范）并不被大多数的工具所支持，且经常会造成开发人员的混淆。比如，仅仅看名字就不能直接了解这两个getOrderItem()方法之间的区别。因此，我们更倾向于使用以下这种签名：

```
public void setOrderItemArray(OrderItem[] newArray);
public OrderItem[] getOrderItemArray();
public void setOrderItem(int index, OrderItem oi);
public OrderItem getOrderItem(int index);
```

以下是实现设置方法的另一种（非标准，但是功能却更强）方式：

```
public BeanType setProperty(PropType newValue) {
    this.property = newValue;
    return this;
};
```

使设置方法返回当前bean实例是，这样你就可以链式调用这些设置方法了：

```
myBean.setProperty(x)
    .setSomeProperty(y);
```

对于两个特性而言，链式调用似乎用处不大，但是假如你的bean中有很多特性，那么使用这种方法就可以帮助节省大量的输入。

2. bean导航

当使用那些可感知bean的工具时，通常可以用一种称为点记法（dot notation）的方法来引用特性，也就是说，你可以不调用获取方法和设置方法，而是直接引用它们所定义的特性。特性名根据之前所讨论的规则来确定。表4-2给出了若干示例。

表4-2 bean导航示例

Java代码	点 记 法
anOrder.getAccount().getUsername()	anOrder.account.username
anOrder.getOrderItem().get(0).getProductId()	anOrder.orderItem[0].productId
anObject.getId()	anObject.ID
anObject.getXCoordinate()	anObject.xCoordinate

假设你有一个bean，并且想要知道这个bean的所有特性的名称，那么以下这个样例方法将使用Java内置的Introspector类来输出它们：

```
public void listPropertyNames(Class c)
    throws IntrospectionException {
    PropertyDescriptor[] pd;
    pd = Introspector.getBeanInfo(c).getPropertyDescriptors();
    for(int i=0; i< pd.length; i++) {
        System.out.println(pd[i].getName()
            + " (" + pd[i].getPropertyType().getName() + ")");
    }
}
```

上面的这个例子使用了Introspector类来获得某bean的PropertyDescriptor对象列表，然后遍历该对象列表，显示出该bean所暴露的特性的名称和类型。这种方法在定位bug时非常有用。

4.1.2 SqlMap API

现在你已经了解了JavaBean究竟是什么，下面就可以开始研究iBATIS所提供的API了。SqlMapClient接口具有的方法超过30个，我们将在以后的各章中逐一介绍所有这些方法。不过现在我们只讨论几个在本章将要使用的API。

1. queryForObject()方法

queryForObject()方法用于从数据库中获取一条记录，并把它放入到一个Java对象中，它有以下两种签名：

- Object queryForObject(String id, Object parameter) throws SQLException;
- Object queryForObject(String id, Object parameter, Object result)throws SQLException;

第一种签名较为常用，只要所返回的对象有默认构造函数（default constructor），它就能为你创建它（如果没有，就会抛出运行时异常）。

第二种签名接受一个将被用作返回值的对象——运行已映射语句之后，该方法并不会创建一个新的对象，而是在该语句中设置特性。如果你请求的对象不能被轻易创建（由于构造函数受保护或者缺少默认的构造函数），那么这种方式就非常有用。

使用queryForObject()方法时需要记住的是，如果查询返回了不止一行，该方法就会抛出异常，因为它总会检查以确保仅返回一行。

2. queryForList()方法

queryForList()方法用于从数据库中获取一行或多行，并把它放入到一个Java对象列表中，

和queryForObject()方法一样，该方法也有两个版本：

- List queryForList(String id, Object parameter) throws SQLException;
- List queryForList(String id, Object parameter, int skip, int max) throws SQLException;

第一个版本的方法将已映射语句所返回的所有对象全部返回。而第二个版本的方法只返回其中的一部分——它将跳到skip参数所指定的位置上，并返回从查询结果开始的max行。例如，假设你有一条已映射语句可返回100行，但是你只需要其中开始的10行，那么就可以通过将skip设置为0，将max设置为10，来得到想要的10行。如果需要的是第二组10条记录，就可以重新调用该方法并且将skip参数设置为10。

3. queryForMap()方法

queryForMap()方法用于从数据库中返回由一行或多行组成的一个Java Map（注意不是List）。和其他查询方法一样，它也有两个版本：

- Map queryForMap(String id, Object parameter, String key) throws SQLException;
- Map queryForMap(String id, Object parameter, String key, String value) throws SQLException;

第一个版本的方法将执行一个查询，并返回由一组Java对象组成的Map，其中这些对象的键值由key参数所指定的特性来标识，而值对象则为已映射语句返回的对象。第二个版本的方法将返回一个类似的Map，但是这些对象将会是value参数所标识的对象的特性。

俗话说，“一例胜千言”。下面就给出一个例子。假设你有一条已映射语句，可以返回一组账户（account）。使用第一个版本的方法，可以创建这样一个Map：它的键为accountId特性，值为完整的账户bean；而使用第二个版本的方法，则可以创建这样一个Map：它的键为accountId特性，而值仅仅是账户名。

```
Map accountMap = sqlMap.queryForMap(
    "Account.getAll",
    null,
    "accountId");
System.out.println(accountMap);
accountMap = sqlMap.queryForMap(
    "Account.getAll",
    null,
    "accountId",
    "username");
System.out.println(accountMap);
```

作为一个iBATIS的初级使用者，你需要了解的API现在就介绍完了，下面来探讨创建已映射语句的不同方式。

4.1.3 已映射语句的类型

在开始使用这些我们刚学习到的SqlMap API之前，你还需要了解如何创建已映射语句以使得SqlMap API能够正常工作。在上一个例子中，我们调用了名为Account.getAll的已映射语句，但是当时我们并没有介绍它是怎么来的（除了第2章中的那个简单的应用程序）。

已映射语句有多种类型，每种类型都有各自不同的用途、属性集以及子元素。但一般来讲，最好使用与你所要完成的任务相匹配的语句类型（例如，要插入数据应该使用<insert>语句，而不是使用<update>语句或者更通用的<statement>类型），虽然这似乎是理所当然的，但仍请你务必遵守。所以这样要求，是因为使用特定类型更能表达特定意图，并且在某些情况下能提供更多的功能（例如在5.2节中将要介绍的<insert>语句及其子元素<selectKey>）。

表4-3包含了已映射语句（以及我们稍候将要讨论的另外两个相关的XML元素）的每一种类型，并给出了其相关信息。

表4-3 已映射语句的类型和相关XML元素

语句类型	属 性	子 元 素	用 途	更多细节
<select>	id parameterClass resultClass parameterMap resultMap cacheModel	所有动态元素	用于选择数据	4.2节； 第8章
<insert>	id parameterClass parameterMap	所有动态元素 selectKey	用于插入数据	5.2节； 第8章
<update>	id parameterClass parameterMap	所有动态元素	用于更新数据	5.3节； 第8章
<delete>	id parameterClass parameterMap	所有动态元素	用于删除数据	5.3节； 第8章
<procedure>	id parameterClass resultClass parameterMap resultMap xmlResultName	所有动态元素	用于调用存储程序	5.5节； 第8章
<statement>	id parameterClass resultClass parameterMap resultMap cacheModel xmlResultName	所有动态元素	可代表所有语句的类型， 几乎可用来执行所有的操作	6.3.1节； 第8章
<sql>	id	所有动态元素	它实际上并非已映射语句，但是可用来创建可用于已映射语句中的组件	4.2节； 第8章
<include>	refid	无	它实际上并非已映射语句，但是可用来将使用<sql>类型所创建的组件插入到已映射语句中	4.2节

本章将集中讨论<select>已映射语句的类型。表4-3中除了已映射语句的类型之外，还包括另外两个通常用来创建已映射语句的元素：<sql>元素和<include>元素。联合使用这两个元素能够创建可插入到已映射语句中的组件。当你想要复用复杂的SQL片段，却又不想复制它们时，就会发现上面这两个元素非常有用。

<sql>元素用于创建一个文本片段，而这些文本片段又可以组合起来以创建完整的SQL语句。例如，查询语句的where子句部分可能是由一组复杂的条件组成的。如果你想要使用这组复杂的条件来选出符合条件的记录数，但又不想复制这组条件，那么就可以把它们放在一个<sql>片段中，然后在计数查询以及返回实际数据的查询中包含它们即可。代码清单4-1给出了一个简单示例。

代码清单4-1 <sql>和<include>标签示例

```
<sql id="select-order">          ❶ 获取所有列
  select * from order
</sql>

<sql id="select-count">         ❷ 获取计数值
  select count(*) as value from order
</sql>

<sql id="where-shipped-after-value">  ❸ 包含发货时间在某日期之后的所有订单
  <![CDATA[
    where shipDate > #value:DATE#
  ]]>
</sql>

<select
  id="getOrderShippedAfter"      ❹ 获得发货时间在某日期之后的所有订单的所有列
  resultClass="map">
  <include refid="select-order" />
  <include refid="where-shipped-after-value" />
</select>

<select
  id="getOrderCountShippedAfter"  ❺ 获得发货时间在某日期之后的所有订单的数量
  resultClass="int">
  <include refid="select-count" />
  <include refid="where-shipped-after-value" />
</select>
```

虽然代码清单4-1只是一个很平凡的例子，但是它非常清晰地展示了<sql>元素和<include>元素的用法，且没有把事情弄得过于复杂。我们定义3个SQL片段：一个用于从表格中获取所有的列❶，另一个用于获取由查询返回的行数❷，最后一个❸用来设置数据过滤的条件。随后创建了两个使用这些片段的已映射语句：用于返回所有符合条件的对象的getOrderShippedAfter❹，以及用于返回符合条件的对象的数目的getOrderCountShippedAfter❺。在此例中，复制SQL代码可能更简单，但是当你开始使用更复杂的增删改查操作或在SQL片段中使用动态SQL（有关

动态SQL的使用，请参考第8章）时，复制过程就会变得非常容易出错。

4.2 使用<select>已映射语句

从数据库中获取数据是一个应用程序中最基本的操作之一。iBATIS框架使得编写大部分的SELECT语句都毫不费力，并且提供了很多特性，使得你几乎可以从数据库中得到任何想要的

4.2.1 使用内联参数（用#做占位符）

到目前为止，我们之前给出的所有示例都简单得没有任何实用价值，因为你通常不会想要执行一条没有任何查询条件的查询语句。内联参数（inline parameter）就是一种在已映射语句中添加查询条件的简单方式，你可以使用两个不同的方法来设置内联参数。

第一个方法是使用散列（#）符号。以下的例子就使用#符号传递一个简单的内联参数，通过accountId值来获取一个唯一的Account bean：

```
<select id="getByIdValue" resultClass="Account">
  select
    accountId,
    username,
    password,
    firstName,
    lastName,
    address1,
    address2,
    city,
    state,
    postalCode,
    country
  from Account
  where accountId = #value#
</select>
```

上面这条已映射语句中的字符串#value#是一个占位符，用于告诉iBATIS你将传递一个简单参数，iBATIS在执行该已映射语句之前会使用此参数来设置SQL。该已映射语句可以用如下方式调用：

```
account = (Account) sqlMap.queryForObject(
    "Account.getByIdValue",
    new Integer(1));
```

下面花一点时间来看一下iBATIS框架是如何处理该语句的。首先，iBATIS将查找名为Account.getByIdValue的已映射语句，并把#value#占位符变换为一个预备语句参数，见如下代码的最后一行：

```
select
  accountId,
  username,
  password,
```



```
firstName,
lastName,
address1,
address2,
city,
state,
postalCode,
country
from Account
where accountId = ?
```

然后，iBATIS将这个参数的值设置为1（即如上Java代码中传递给queryForObject()的第二个Integer类型的参数），并执行该预备语句。最后，iBATIS接收该结果行，把它映射到一个Java对象，并返回这个对象。

尽管看上去这可能像是很底层的信息，但理解此处究竟发生了什么是非常重要的。有关iBATIS最常见的问题之一就是：“如何在WHERE子句中使用LIKE？”通过上面这条语句，你就可以很容易地明白^①：为何输入的参数必须具有通配符，以及为何它们不能被轻易地插入到SQL语句中。对于这个难题，有3种可能的解决方案：

- 在所传入的参数中使用SQL通配符。
- 要搜索内容的文本必须是能够被参数化的SQL表达式的一部分。例如'%' || #value# || '%'。
- 改为使用替代语法（这正是4.2.2节的主题）。

4.2.2 使用内联参数（用\$做占位符）

使用内联参数的另一种方式就是使用替代（\$）语法，它可以用来把值直接插入到SQL语句之中（在该SQL语句被转变为参数化语句之前）。但是使用这种方式时要非常小心，因为它可能使你暴露给SQL注入，另外过度使用还可能造成性能问题。

以下是一种在iBATIS中处理LIKE运算符的方法示例：

```
<select id="getByLikeCity" resultClass="Account">
  select
    accountId,
    username,
    password,
    firstName,
    lastName,
    address1,
    address2,
    city,
    state,
    postalCode,
    country
  from Account
  where city like '%$value$%'
</select>
```

① 将其中的=换为LIKE才能明白。——译者注

以上这个语句同前一个语句的区别在于：iBatis如何处理传递给语句的参数。该语句的调用方式和之前那个语句的调用方式是完全一致的：

```
accountList = sqlMap.queryForList(  
    "Account.getByLikeCity",  
    "burg");
```

这一次，iBatis把语句转化为如下形式：

```
select  
    accountId,  
    username,  
    password,  
    firstName,  
    lastName,  
    address1,  
    address2,  
    city,  
    state,  
    postalCode,  
    country  
from Account  
where city like '%burg%'
```

此语句中并没有设置任何参数，因为这个语句已经是完整的，但是使用这种技术时需要牢记的一点就是：它使得你的应用程序更容易受到SQL注入的攻击。

4.2.3 SQL 注入简介

SQL注入攻击是指恶意的用户将特殊格式的数据传递给某个应用程序，以使该应用程序做一些不该做的事。例如，在上一个例子中，假设某个用户传递以下这样的文本：

```
burg'; drop table Account;--
```

就会把简单select语句转化成以下这样的恶意语句集合：

```
select  
    accountId,  
    username,  
    password,  
    firstName,  
    lastName,  
    address1,  
    address2,  
    city,  
    state,  
    postalCode,  
    country  
from Account  
where city like '%burg';drop table Account;--%'
```

如此一来，这位聪明的用户就可以从数据库中查询出所有以burg结尾的记录，这没有什么大不了的。但是随后他就会从数据库中删除一个表（如果他仅仅删除一个表的话，那你真是太幸运了。

的调用

因为如果他确实很聪明，他就会意识到这是一个不可多得的处理和删掉很多表的机会）。字符串最后的--告诉数据库忽略drop table语句之后的任何东西，因此数据库将不会抛出任何错误。

如果在产品开发的真实应用程序中由于你所写的代码而发生了这种问题，那么总会有你难受的一天。正如我们之前所说的，使用\$占位符请务必谨慎。

4.2.4 自动结果映射

你可能已经注意到了，在所有我们给出的示例中都没有定义任何结果映射（result map）^①，但确实定义了结果类（result class）。这种方式所以可行是因为iBATIS的自动结果映射机制，该机制会在已映射语句第一次被执行时，迅速地自动创建一个结果映射，然后将它应用于这条已映射语句。

可以有3种方式来使用这个特征：单列选择（single-column selection），固定多列选择（fixed-column list selection）和动态多列选择（dynamic-column list selection）。

要牢记

注意 如果你既没有提供结果映射，也没有提供结果类，iBATIS将执行你的语句但是却不会返回任何东西。从最早期开始，iBATIS就是这样做的，并且造成了很多人的不满。不幸的是，一些用户随意地使用select语句来进行插入和更新，尽管这种做法看起来很糟糕，但我们仍允许这样，以避免破坏现有应用程序。

序做一

如果只想从某个查询中获取单列，就可以使用别名value作为一种快捷方式来完成此目的，这样就可以不需要定义复杂的结果映射了：

```
<select id="getAllAccountIdValues"
  resultClass="int">
  select accountId as value
  from Account
</select>

List list = sqlMap.queryForList(
  "Account.getAllAccountIdValues", null);
```

该已映射语句返回Account表中所有的accountId值，作为简单的Integer对象的List。

如果需要查询多列，就可以使用自动结果映射来告诉iBATIS将列名作为bean的特性名，或者作为Map键。

当以这种方式映射到bean时，需要牢记一点：如果所选择的列在数据库中存在，但是不存在于你要映射的bean中，你将不会得到任何错误或警告，但是也得不到任何数据——这些数据只会静静地被忽略。映射到Map对象时，也有相似的问题：尽管你仍然可以得到数据，但是这些数据不会在你所期望的地方。

什么大
幸运了。

^① 关于结果映射的详细讨论，参考4.4节。——译者注

如果你想要一种更加稳健的数据映射方式，请使用外部的结果映射（见4.3.1节）。

除了上述这两个潜在的问题之外，如果希望让框架帮你完成映射，并且不在意第一次执行映射时额外的开销，那么使用自动映射还是很方便的。

如果语句中被选择的那些字段列表可以在运行时改变，那么也可以使用动态结果映射。代码清单4-2给出了一个使用动态结果映射的查询。

代码清单4-2 动态结果映射示例

```
<select id="getAccountRemapExample"
  remapResults="true"
  resultClass="java.util.HashMap" >
  select
    accountId,
    username,
    <dynamic>
      <isEqual
        property="includePassword"
        compareValue="true" >
        password,
      </isEqual>
    </dynamic>
    firstName,
    lastName
  from Account
  <dynamic prepend=" where ">
    <isEmpty property="city">
      city like #city#
    </isEmpty>
    <isNull
      property="accountId"
      prepend=" and ">
      accountId = #accountId#
    </isNull>
    </dynamic>
</select>
```

① 当已映射语句执行时重新映射结果

② 包含简单的动态SQL以演示技术

代码清单4-2中所给出的示例使用了remapResults属性①和动态SQL②来演示如何动态改变已映射语句所返回的数据。我们要到第8章才正式介绍动态SQL，但此例使用它创建了一个已映射语句，其中includePassword特性的值决定了结果中包含哪些字段。你是否可以在结果中获取密码字段，就是由includePassword特性的值决定的。需要了解的一点是：每次该动态语句执行时，确定结果映射对性能影响可能会很大，因此只有确实需要时才使用该特征。

4.2.5 联结相关数据

有时候出于制作报表或者其他的什么目的，你可能想要把多个数据表联结起来，形成一个平板型结构。iBATIS框架可以让你毫不费力地完成这项工作，因为它是将SQL语句映射为

对象，而不是将数据库中的表映射为对象，所以映射单表查询和映射多表查询几乎没有什么不同。

在第7章中，我们将讨论如何进行更高级的多表操作，以便直接从父对象得到其包含的子对象列表——例如某个订单的详细订单项列表。

但是现在，我们只是重申：映射单表查询和映射多表查询几乎没有什么不同。

我们曾说过，可以认为SQL语句和函数非常相似，它们都可以接收一些输入值，然后，根据这些输入值产生相应的输出值。下一节中，我们将研究如何提供这些输入值。

4.3 映射参数

有两种方式可以将参数映射到已映射语句中：内联映射（inline mapping）和外部映射（external mapping）。使用内联参数映射意味着：告诉iBATIS关于你想要什么的暗示，然后就让它完成细节的处理。但是外部参数映射则明确得多——可以明确地告诉iBATIS你要它做些什么。

4.3.1 外部参数映射

使用外部参数映射^①时，可以指定多达6个属性^②。如果没有指定其中的某个属性，iBATIS就会用反射来尽可能地为其确定合理的值，但是这么做比较费时并且可能不准确。表4-4列出了用于映射参数的这些属性，并且简单描述了每个属性的用法。

表4-4 参数映射的属性

属 性	描 述
property	设定一个参数 ^③ 时，property属性用于指定JavaBean中某个特性的名字 ^④ ，或者指定作为参数传递给已映射语句的Map实例的某个键值对（Map entry）的键值 ^⑤ 此名字可以不只使用一次，具体根据它在语句中所需要的次数而定 例如，假设在一个SQL UPDATE语句中，同一个特性既出现在SET子句中（以表示正被更新，又作为键的一部分 ^⑥ ）用在WHERE子句中，那么这个名字在该已映射语句中就被引用两次
javaType	设定一个参数时，javaType属性用来显式指定要设置的参数的Java特性类型 通常这个属性可以通过反射从JavaBean特性中推断出来，但是某些映射方式 ^⑦ 并不能为框架提供这样的类型。在这种情况下，如果没有设置javaType，而框架又不能通过另外的方式确定类型，就会假定类型为Object

① 具体来说，就是在sqlMap文件中定义一个parameterMap元素，然后在其中定义一条条的parameter子元素，详细信息见5.2.2节。——译者注

② 这6个属性都是parameter子元素的属性，详细信息见5.2.2节。——译者注

③ 即在parameterMap元素中定义一个parameter子元素。——译者注

④ 当父元素parameterMap的class属性取值为某个JavaBean的类名时。——译者注

⑤ 当父元素parameterMap的class属性取值为某个Map实现类时。——译者注

⑥ 即WHERE子句中name=value.name。——译者注

⑦ 例如Map和XML映射，详细信息见5.2.2节。——译者注

(续)

属 性	描 述
jdbcType	<p>设定一个参数时, jdbcType属性用来显式指定参数的数据库类型。一些JDBC驱动程序在进行某些特定操作时, 如果没有显式提供列的类型信息, 它们就不能够识别出这些列的类型。有关问题, 一个绝佳的例子就是 PreparedStatement.setNull(int parameterIndex, int sqlType)方法。该方法的第二个参数用于指定类型。如果不指定, 一些驱动程序允许其通过发送 Types.OTHER或Types.NULL而被隐式指定</p> <p>但是, 不能保证所有的驱动程序都表现一致, 有一些驱动程序就要求显式指定这个类型。对于这种情况, iBATIS允许通过parameterMap特性元素的jdbcType属性来指定其类型</p> <p>通常, jdbcType属性只有当某列允许被设置为空时才是必需的</p> <p>使用该属性的另一个原因是, 当Java类型模糊时, 可用该属性来明确地指定日期类型, 例如, Java表示时间只有一个Date值类型 (java.util.Date), 但是大部分SQL数据库通常都至少有三个不同的类型。此时, 你就需要明确地指定数据库表的列的类型是DATE还是DATETIME</p> <p>jdbcType属性可以被设置为任何同JDBC的Types类中的某个常量相匹配的字符串值</p>
nullValue	<p>设定一个参数时, nullValue属性可以根据特性类型被设置为任何有效的值</p> <p>nullValue属性用来指定外发的空值替换。也就是说, 数据库写入时, 如果在待写入的JavaBean特性或Map键值对中检测到该替换值, 就将空值写入到数据库 (反过程亦然, 即从数据库中读出一个空值时, 则将相应的JavaBean字段或Map 键值对的值设为该替换值)</p> <p>这样就能允许你在应用程序中对于那些取值不允许为空的java类型 (例如, int、double、float等类型), 设定一个“神奇”的空值。当这些类型的特性取值包含了匹配的空值时 (例如, -9999), 就将在数据库中写入空, 而不是把这个“-9999”写入数据库</p>
mode	<p>该属性专门用于支持存储过程。我们将在5.5.2节中详细讨论它</p>
typeHandler	<p>如果想要指定类型处理器 (而不是让iBATIS根据 javaType和jdbcType属性来选择类型处理器), 你就可以指定typeHandler属性</p> <p>通常, 该属性用来提供自定义的类型处理器 (我们将在第12章中详细介绍)</p>

在iBATIS中内联参数映射对于大多数的已映射语句都很好用, 但是如果你想要提高性能, 或者遇到某些不在预期较难掌控的情况, 可能你所需就需要使用外部参数映射了。在第5章中, 我们将详细讨论外部参数映射, 但就本章的目的而言, 还不需要展开讨论它。

4.3.2 再论内联参数映射

在4.2节开始时, 我们讨论了如何使用最简单形式的内联参数映射, 来告诉iBATIS我们想要在运行时代入到查询中的特性的名称。除了使用这种最简单的形式, 也可以在内联参数映射中提供一些外部参数映射所允许的特性, 例如: jdbcType (数据库类型) 以及nullValue (参数的空值占位符), 只要用冒号将参数名、数据库类型和空值占位符分隔开即可。当你的数据库中包含允许为空的列时, 数据库类型通常是必需设置。这是因为JDBC API使用以下这个方法把空值发送到数据库:

```
public void setNull(  
    int parameterIndex,  
    int sqlType);
```

如果你没有明确告诉iBATIS究竟使用何种数据库类型, 那么iBATIS将默认地使用 java.sql.Types.OTHER作为第二个参数, 但一些数据库驱动程序是不允许这么做的 (例如, Oracle驱动程序目前就不允许), 此时就会产生错误。

提示 当你使用内联参数映射却没有设置nullValue时，如果程序报错，并且在错误日志中出现了不在预期的数字1111时，那么很有可能问题就是你的驱动程序不喜欢没有显式类型的空值。OTHER这个整型值就是1111。

要在内联参数中把数据库类型告诉iBATIS，你可以把java.sql.Types常量类中代表某个数据库类型的常量的名字添加到特性名称的后面，并且用冒号将它们分隔开。下面这个示例就是一个使用内联参数方式指定数据库类型的已映射语句：

```
<select id="getOrderShippedAfter"
      resultClass="java.util.HashMap">
  select *
  from order
  where shipDate > #value:DATE#
</select>
```

说明 请记住我们使用的是XML。在之前给出的示例中，如果想要创建名为getOrderShippedBefore的语句并且设置查询条件shipDate < #value:DATE#，我们就必须使用<lt；（而不是<），因为在XML中，小于号表示你要开始一个新的元素。也可以使用CDATA，但是同时使用动态SQL（第9章）和CDATA时要特别小心，因为CDATA中的动态SQL标签不会被解析。通常，在XML中也可以直接使用>，但是为了安全起见最好使用>；来代替。

空值替换使你能够在Java代码中使用“魔力数字”^①，就像在数据库中使用空值一样。对于大部分应用程序而言，使用“魔力数字”进行开发并不是一种很好的设计（事实上，我们实在想不出在哪一种情况下这么做是适合的），它只是为了支持iBATIS初学者的不成功的模型设计。如果需要数据库具有一个空值，那么也应该在Java模型设计中把它映射为空值。

在iBATIS的当前版本中，还可以使用name=value语法来设定参数映射。例如，以下示例和前一个示例是等价的：

```
<select id="getOrderShippedAfter" resultClass="hashmap">
  select *
  from "order"
  where shipDate > #value,jdbcType=DATE#
</select>
```

请记住内联参数映射只是创建参数映射的一个简化操作。假如你的已映射语句出现错误，告诉你某参数映射YourNamespace>YourStatement-InlineParameterMap出错了，请查看命名空间YourNamespace中的语句YourStatement的内联参数映射。这就是出现错误的地方。即使你没有定义参数映射，也不一定就意味着参数映射不存在^②。

① 即数据库NULL在某个java类型中的对应值。——译者注

② iBATIS会在语句第一次执行时自动创建参数映射。——译者注

4.3.3 基本类型参数

正如我们在结果映射中所看到的，基本类型只有被包装在另外的对象中才能被使用（这里假定你所使用的不是Java5，在Java5中，由于“自动装箱”特征，你可以在任何地方传递基本类型参数）。如果你想要传递一个基本类型的值给iBATIS，可以使用bean（参见4.3.2节）或者基本类型包装类（例如，Integer、Long等）。

说明 基本类型构成的数组也可以被传递到iBATIS，但是使用数组超出了本章的范围。有关在动态SQL中使用数组的信息，请参考8.2.5节。

好，下面我们继续深入探索……

4.3.4 JavaBean 参数和 Map 参数

虽然bean参数和Map参数有些不同，但是使用它们时所用的语法是一致的。这两者的不同在于：加载参数映射时，它们的行为是不相同的。

如果使用bean创建了一个参数映射^①，并且试图引用一个该bean中实际上不存在的特性，那么，当iBATIS加载该参数映射时，就会马上报错。这其实是件好事，因为它可以帮助你避免让用户发现bug——而是由你（开发人员）自己来发现。

如果使用Map来做这件事^②，那么iBATIS就无法知道该特性是否存在，因此iBATIS将不可能帮助你确定潜在的错误。

这就引出了一个很重要的问题：及早发现错误是一件好事。应用程序发布之前就发现并且改正了的bug，不会影响你的用户。给用户的应用程序中bug越少，就意味着用户工作效率越高，这样软件的价值也就升高了。

好，现在我们已经确定了输入值，接下来我们将讨论输出值。

4.4 使用内联结果映射和显式结果映射

内联结果映射的确非常好，因为它们非常容易使用，并且在绝大多数情况下都能很顺利地地完成工作。

iBATIS中的显式结果映射（注意并不是内联结果映射）也同样有价值，因为它们可以提供更好的性能、更严格的配置验证，以及更加精确的行为。使用显式结果映射，在运行时很少会出错。表4-5描述了显式结果映射可用的属性^③。

① 即定义一个parameterMap元素，其class属性取值为某个JavaBean的全限定类名。——译者注

② 即定义一个parameterMap元素，其class属性取值为某个Map实现类的全限定类名。——译者注

③ 即resultMap元素中result子元素的属性。——译者注

表4-5 结果映射的属性

属 性	描 述
property	设定一条结果映射时 ^① ，property属性用于指定JavaBean中某个特性的名字 ^② ，或者指定作为参数传递给已映射语句的Map实例的某个键值对（Map entry）的键值 ^③ 这个名字可以只被使用一次，具体根据在结果填充时所需要的次数而定
column	设定一条结果映射时，column属性用于提供ResultSet（用于填充特性）的列的名称
columnIndex	作为一个可用于增强性能的属性，columnIndex属性是一个可选字段，用于提供列的索引以代替ResultSet中的列名 99%的应用程序下不需要为了性能而牺牲可维护性和可读性。使用这个属性是否就一定能够提高性能还要取决于你的JDBC驱动程序
jdbcType	设定一条结果映射时，jdbcType属性用于显式指定ResultSet列（用于填充特性）的数据库的列类型 虽然结果映射不像参数映射一样存在空值设置的困难，设置该类型对某些映射类型（比如Date特性）来说还是有用的 考虑到Java只有一个Date值类型，而SQL数据库则可能有多个（通常至少有3个），指定一个Date类型以保证日期（或其他类型）能够被正确设置，在某些情况下就变得十分必要 同样地，VARCHAR、CHAR以及CLOB可以用来填充String类型，这种情况下，指定一个确定的数据库类型同样十分必要 是否设置该属性，取决于你使用的数据库驱动程序
javaType	设定一条结果映射时，javaType属性用于显式指定要设置的特性的Java特性类型。通常这个属性可以通过反射从JavaBean特性中推断出来，但是某些映射方式 ^④ 并不能为框架提供这样的类型信息 在这种情况下，如果没有设置javaType，而框架又不能另外确定类型，就会假定类型为Object
nullValue	设定一条结果映射时，nullValue属性用来替换数据库中的空值 ^⑤ 这样，当从ResultSet中读出一个NULL时，JavaBean特性就会被设为由null Value属性所指定的值（而不是Null）。nullValue属性可以根据bean的特性类型被设置为任何有效的值
select	设定一条结果映射时，select属性用于描述对象之间的关系，这样iBATIS就能够自动地加载复杂的特性类型 该语句特性的值必须是另外一个已映射语句的名字 定义在同一属性元素中的作为该语句属性的数据库列（即column属性）的值将被作为这条已映射语句的参数值传递给它 因此，该列必须是iBATIS支持的简单数据类型 详细讨论请见第5章

好，现在你已经了解了有哪些属性，那么该如何使用这些属性呢？继续阅读你就可以知道了。

4.4.1 基本类型结果

Java语言中共有8个基本类型（boolean、char、byte、short、int、long、float和double），每一种基本类型都有对应的包装类（Boolean、Char、Byte、Short、Integer、Long、Float和Double）。

① 即在resultMap元素中定义一个result子元素。——译者注
② 当父元素resultMap的class属性取值为某个JavaBean时。——译者注
③ 当父元素resultMap的class属性取值为某个Map实现类时。——译者注
④ 例如Map和XML映射，详细信息见5.2.2节。——译者注
⑤ 或者说“对应值”。——译者注

尽管iBATIS不允許你直接获得基本类型的结果，但是它允許你获得基本类型的包装结果。例如，假设你想要某个客户的所有订单的数目，就可以通过Integer来获得，而不是基本类型int，如下所示：

```
Integer count = (Integer)sqlMap.queryForObject(  
    "Account.getOrderCountByAccount",  
    new Integer(1));
```

```
<select  
    id="getOrderCountByAccount"  
    resultClass="java.lang.Integer" >  
    select count(*) as value  
    from order  
    where accountId = #value#  
</select>
```

如果使用bean来获取结果，这个bean就必须有一个int特性来接收结果，如下例所示：

```
public class PrimitiveResult {  
    private int orderCount;  
    public int getOrderCount() {  
        return orderCount;  
    }  
    public void setOrderCount(int orderCount) {  
        this.orderCount = orderCount;  
    }  
}
```

```
<resultMap id="primitiveResultMapExample"  
    class="PrimitiveResult">  
    <result property="orderCount"  
        column="orderCount" />  
</resultMap>
```

```
<select id="getPrimitiveById"  
    resultMap="primitiveResultMapExample">  
    select count(*) as orderCount  
    from order  
    where accountId = #accountId#  
</select>
```

因此，底线就是：iBATIS能把结果映射为任意类型，但是由于它只返回Object实例，基本类型的值就必须以下面形式之一来包装：简单的值包装、bean或者Map。

说明 同样地，如果你使用的是J2SE 5，这就不完全对了。很多方法（例如，queryForObject()）仍然返回Object的实例。你虽然不能将Object直接转换为基本类型（int或long），但是你可以将它转换为包装类型（Integer或Long），然后编译器会将它重新拆箱为基本类型。然而，此时需要注意的是：如果所返回的Object为空，包装类型的值也将为空。当出现这种情况时，应用程序会抛出NullPointerException，这是因为待拆箱类型不能为空。

有时候（实际上是大多数时候），你可能想从SQL语句中获取不止一列数据。此时，你就要把结果放在JavaBean对象或者Map实例中。

4.4.2 JavaBean 结果和 Map 结果

除了基本包装类（例如Integer、Long等）之外，iBATIS框架还允许你使用Map实例或bean对象来做结果映射。这两种方式各有其优点和缺点，详见表4-6。

表4-6 JavaBean和Map作为数据结构的优缺点

方 式	优 点	缺 点
Bean	性能 编译时强类型 编译时名字检查 IDE中的重构支持 更少的类型转换	更多的代码（get/set）
Map	较少的代码	更慢 没有编译时检查 弱类型 较多的运行时错误 没有重构支持

通常，对于领域数据（例如，从数据库中获取一个用于编辑的账户对象或订单对象）我们推荐使用bean，而对于那些不那么关键并且更加动态的数据（例如，报表或者其他输出方法），我们推荐使用Map。

创建结果映射时，如果你将一个字段映射到一个不存在的特性上，那么一旦结果映射被加载你就会立即得到一个异常。这种在加载SQL映射时就能暴露出的问题其实是一件好事，因为它意味着可以在用户看到这个错误之前就发现它。

另一方面，如果你把一个不存在的列映射到一个确实存在的特性上，那么当试图获取此数据时，你将获得运行时错误。这就是为什么需要在DAO层进行大量单元测试的原因（第13章将详细讨论这种iBATIS最佳实践）。

4.5 小结

本章中，我们研究了有关JavaBean、SQL映射API和已映射语句的基本知识。随着你对本章所讨论内容的更加深入了解，创建已映射语句将会变得和你执行的其他开发任务一样的简单，并且和其他任何事情一样：练习得越多，就越觉得容易。

如果想要使你的应用程序更加严密并且尽可能地消除运行时错误，就请使用显式参数映射和显式结果映射，并且对于参数和结果都使用强类型的bean（而不是Map）。这也将使你的应用程序启动更快（因为这样iBATIS就不需要在应用程序加载时立即把所有这些显式映射都载入内存，而是在需要时才载入），运行更快，并且所使用的内存更少。

下一章将首先介绍数据库维护所需的所有基本操作，然后探讨iBATIS对非查询操作的支持。

第5章

执行非查询语句

本章内容

- iBATIS API详述
- 插入数据
- 更新和删除数据
- 使用存储过程

能在数据库中进行查询当然是很有用的，但是大部分应用程序时还需要能够把数据插入到数据库中。本章中，我们将研究使用iBATIS框架向数据库中插入数据的几种方式。它们将建立在第4章中介绍的概念的基础之上，因此如果你刚开始学习iBATIS并且还没有阅读那一章，建议你先快速浏览一下第4章。第4章中几乎所有的关于参数映射的知识都可以应用到非查询®类型的已映射语句中（结果映射的知识也是这样，不过范围更小一些）。

5.1 更新数据的基本方法

在第4章中，你已经学习了所有可用的语句类型以及和基本查询相关的API。下面将介绍通常用于执行非查询语句的API，并且回顾用于更新数据库的已映射语句的类型。

5.1.1 用于非查询 SQL 语句的 SqlMap API

有关数据库更新的高级方式将留到下一章介绍，本章中，我们只讨论基础的方法，即insert、update和delete这3种最常用于更新数据的方法。本章后面部分将逐一地详细介绍每种方法，不过本节将先对这3种方法作个简略介绍，这样你就可以开始使用它们了。

1. insert方法

你可能已经猜到，insert方法用来执行和SQL的insert语句相对应的已映射语句：

```
Object insert(String id, Object parameterObject)
    throws SQLException;
```

insert方法接受两个参数：要执行的已映射语句的名称，以及构建insert语句（该语句将

① 即增改删操作。——译者注

把你的数据插入到数据库中)时需要的参数对象。

在更新数据库所常用的3种方法中, insert方法与众不同的地方就是: 它返回Object对象(参见5.2.3节)。

2. update方法

update方法用来执行与SQL的update语句相对应的已映射语句:

```
int update(String id, Object parameterObject)
    throws SQLException;
```

和insert方法一样, update方法也接受两个参数: 要执行的已映射语句的名称, 以及一个参数对象(用于提供完成已映射语句时所需要的参数)。该方法的返回值是受到update语句影响的记录的数目(当特定的JDBC驱动程序支持时)。

3. delete方法

delete方法和update方法非常相似, 不同之处仅仅在于它不是用来执行SQL的update语句, 而是用于执行delete语句:

```
int delete(String id, Object parameterObject)
    throws SQLException;
```

该方法所接受的参数和其他两个方法一样: 要执行的已映射语句的名称, 以及用来完成已映射语句需要的参数对象。该方法的返回值为它所删除的记录的数目。

5.1.2 非查询已映射语句

表5-1实际上是第4章中表4-3的一部分。表中给出了用来更新数据库的3种最主要的已映射语句类型, 以及其他两个用于创建它们的顶层配置元素。

表5-1 用于更新数据的已映射语句类型(以及相关的XML元素)

已映射语句类型	属 性	子 元 素	用 途	更多细节
<insert>	id parameterClass parameterMap	所有动态元素 <selectKey>	插入数据	5.2节; 第8章
<update>	id parameterClass parameterMap	所有动态元素	更新数据	5.3节; 第8章
<delete>	id parameterClass parameterMap	所有动态元素	删除数据	5.3节; 第8章
<procedure>	id parameterClass resultClass parameterMap resultMap xmlResultName	所有动态元素	调用存储过程	5.5节; 第8章

(续)				
已映射语句类型	属 性	子 元 素	用 途	更多细节
<sql>	id	所有动态元素	它实际上并非已映射语句，但是可用来创建能在已映射语句中使用的组件	4.2节； 第8章
<include>	refid	无	它实际上并非已映射语句，但是可用来将<sql>类型所创建的组件插入到已映射语句中	4.2节

有关<sql>和<include>元素的更多信息，请参考4.1.3节。

现在我们已经有了用于构建非查询已映射语句的基本块，下面就来看下如何把它们组合在一起。

5.2 插入数据

把数据插入数据库和从数据库中查询数据并不完全相同，但是过程非常相似。不管你是使用内联参数映射还是外部参数映射（这两者我们都已经在第4章中详细讨论了，参见4.3.1节和4.3.2节），它们和其他已映射语句的工作机制是一样的。

5.2.1 使用内联参数映射

内联参数映射通过标记(markup)告诉iBATIS你想让它如何将你的输入参数映射到已映射语句中，这样你就不需要显式定义外部参数映射，从而能够快速地构建已映射语句。以下是一个使用内联参数映射的insert语句的示例：

```
<insert id="insertWithInlineInfo">
  insert into account (
    accountId,
    username, password,
    memberSince,
    firstName, lastName,
    address1, address2,
    city, state, postalCode,
    country, version
  ) values (
    #accountId:NUMBER#,
    #username:VARCHAR#, #password:VARCHAR#,
    #memberSince:TIMESTAMP#,
    #firstName:VARCHAR#, #lastName:VARCHAR#,
    #address1:VARCHAR#, #address2:VARCHAR#,
    #city:VARCHAR#, #state:VARCHAR#, #postalCode:VARCHAR#,
    #country:VARCHAR#, #version:NUMBER#
  )
</insert>
```

这就是insert类型的已映射语句，以下给出用于执行该语句的代码（来自于一个单元测试）：

```
Account account = new Account();
account.setAccountId(new Integer(9999));
account.setUsername("inline");
account.setPassword("poohbear");
account.setFirstName("Inline");
account.setLastName("Example");
sqlMapClient.insert("Account.insertWithInlineInfo", account);
```

尽管这个已映射语句能够顺利执行，但是一旦有好几个不同版本的insert语句以及一些不同版本的update语句，并且还使用十几个查询，那么使用内联参数映射就会使你的语句冗长并且难以维护一致性。此时，外部参数映射就可以帮助你简化SQLMap文件的维护。

5.2.2 使用外部参数映射

和使用内联参数映射相比，使用外部参数映射（external parameter map）除了能提供相同的功能之外，还具有更好的性能并且iBATIS能在加载它的同时对其进行验证（这意味着更多的错误在测试时就可以被发现并得到处理，而不是等用户在运行你的应用程序时才发现）。

下面给出一个使用外部参数映射的insert语句的示例。以下代码在功能上和前一个例子是一样的，只不过它使用的是外部参数映射而不是内联参数映射：

```
<parameterMap id="fullParameterMapExample" class="Account">
  <parameter property="accountId" jdbcType="NUMBER" />
  <parameter property="username" jdbcType="VARCHAR" />
  <parameter property="password" jdbcType="VARCHAR" />
  <parameter property="memberSince" jdbcType="TIMESTAMP" />
  <parameter property="firstName" jdbcType="VARCHAR" />
  <parameter property="lastName" jdbcType="VARCHAR" />
  <parameter property="address1" jdbcType="VARCHAR" />
  <parameter property="address2" jdbcType="VARCHAR" />
  <parameter property="city" jdbcType="VARCHAR" />
  <parameter property="state" jdbcType="VARCHAR" />
  <parameter property="postalCode" jdbcType="VARCHAR" />
  <parameter property="country" jdbcType="VARCHAR" />
  <parameter property="version" jdbcType="NUMBER" />
</parameterMap>

<insert id="insertWithExternalInfo"
  parameterMap="fullParameterMapExample">
  insert into account (
    accountId,
    username, password,
    memberSince
    firstName, lastName,
    address1, address2,
    city, state, postalCode,
    country, version
  ) values (
    ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?
  )
</insert>
```


尽管上面的代码看上去并不比内联版的短多少，但是当你包含更多的语句时，区别就会变得非常明显。使用外部参数映射不仅可以简化代码（因为你不需要为每个属性指定类型），而且集中式的维护意味着当你要修改参数映射时，只需要修改一次即可。

例如，无论需要在哪里输入memberSince特性，它都会被自动处理为TIMESTAMP数据库类型。假设后来我们认为DATE就足够了（因为并不需要了解账户是在哪一秒钟被创建的），就只需要在一个地方（参数映射）修改它即可。

这种方式的另一个好处就是：内联参数映射第一次被调用时不必被动态地生成。

在以上两个示例中，用来调用语句的代码都是相同的（除了示例中的已映射语句的名称之外）：

```
sqlMap.insert("Account.insertWithInlineInfo", account);  
sqlMap.insert("Account.insertWithExternalInfo", account);
```

内联参数映射和外部参数映射的区别就在于维护的代价和性能——这两者都可以通过使用外部参数映射得到提高。

5.2.3 自动生成的键

能够在数据库表中唯一识别出每一条记录，这种能力对于任何数据库都是非常重要的。几乎所有的数据库都包括一些为新插入的记录自动生成主键（primary key）的方式。尽管这非常方便，但是对于当你将一条记录插入到数据库后就立即希望知道该记录的主键值这种情况，就会有问题。

大多数数据库开发商都提供了一种方式用于确定当前会话中上一个生成的键值，他们通过使用标准SQL来调用存储过程中的这项功能。好几家数据库开发商（包括Oracle和PostgreSQL）也提供了一种生成标识值而不插入列的方式。另外，在JDBC 3.0规范中也有相应的修改，提供了在插入数据时获取主键值的API。

如果你将数据库设计为使用自动生成的主键，就可以使用iBATIS的<selectKey>元素（该元素是<insert>元素的一个专用子元素）来获取这些自动生成的主键的值并将其保存在对象中。完成这项工作可以有两种方式，具体选择何种方式由你所使用的具体的主键生成技术来确定。

第一种方式是，当你把记录插入到数据库中并且数据库为该记录自动生成了主键值之后，就立即抓取该键值。此时要注意的是，必须确保所使用的数据库驱动程序确实能返回你执行上一条insert语句所得到的键值。例如，假设两个线程几乎同时执行insert语句，执行的顺序可能是：[insert for user #1]、[insert for user #2]、[selectKey for user #1]、[selectKey for user #2]。如果驱动程序只是返回上一个键值（全局），那么[selectKey for user #1]将得到2号用户的键值，这将对应用程序产生严重破坏。对于这个问题，大多数数据库驱动程序都应该能正确处理，但是如果你并不绝对确定驱动程序是否能正确处理，请一定要做个测试。同时，请注意使用这种方式时触发器也可能造成影响。例如，对于Microsoft SQL Server而言，

@@identity的值会受到触发器的影响，因此如果你的数据插入动作会激发触发器，而该触发器也会插入一条记录并因此生成一个键值，那么@@identity所返回的值将会是触发器插入记录时所生成的键值，而不是你插入第一条记录时生成的键值。此时，你最好使用SCOPE_IDENTITY函数。

第二种方式是在插入记录之前就获取键值。如果你使用数据库交互工具来插入记录，那么这种方式需要更多的工作量，因为你必须在插入记录之前先分配一个键值。然而，这种策略可以避免因多线程同时插入数据并在记录插入之后才获取键值所带来的潜在风险，并且通常是最安全的方式，因为它几乎不需要对代码作任何假定。使用第一种方式可能会出现潜在的问题，使得应用程序无法像期望的那样顺利执行。但是使用这种方式，我们所需要的就是确保所得到的键值是唯一的。我们不需要数据库在当前会话中管理键值，只需要它生成并把键值返回给我们即可。

对于这两种方式，iBATIS都可以帮助更轻松地完成工作。<selectKey>元素使得这个任务对于你的应用程序来说完全是透明的（至少在调用代码中）。insert方法的签名如下所示：

```
Object insert(
    String id,
    Object parameterObject
) throws SQLException;
```

insert方法返回一个Object对象的原因是：使你能够得到所生成的键值。例如，假设你的应用程序中有如下这个已映射语句和代码（使用了前面所介绍的第二种方式）：

```
<insert id="insert">
  <selectKey
    keyProperty="accountId"
    resultClass="int">
    SELECT nextVal('account_accountid_seq')
  </selectKey>
  INSERT INTO Account (
    accountId, username, password
  ) VALUES (
    #accountId#, #username#, #password#)
</insert>
```

```
Integer returnValue = (Integer) sqlMap.insert(
    "Account.insert", account);
```

returnValue变量将包含所生成的键值。但是还不止这些——<selectKey>元素中的keyProperty属性会告诉iBATIS获取键值并将其设为待插入对象的相应特性®的值。这意味着如果你愿意，你甚至可以忽略返回值，因为被插入的对象的主键已经被设置为了该值。

需要记住的是：<selectKey>元素其实也定义了一个已映射语句，并且该已映射语句可以访问的参数映射与包含该已映射语句的insert语句相同。因此，在上一个示例中，如果想要选择用于记录插入的序列（sequence），就可以使用这样一个已映射语句：

```
<insert id="insertSequence">
  <selectKey keyProperty="accountId" resultClass="int">
```

① 即keyProperty属性值指向的特性。——译者注

```
SELECT nextVal (#sequence#)
</selectKey>
INSERT INTO Account (
    accountId, username, password
) VALUES (
    #accountId#, #username#, #password#
)
</insert>
```

这个已映射语句将接受名为sequence的特性，该特性包含用于插入记录的序列名称。

在上一个示例中，我们以这样的方式获取键，即从一个序列中取出下一个值，然后在插入记录前将其作为参数对象传递给已映射语句。另一方面，如果我们使用的是Microsoft SQL Server，那我们就可以使用以下这个已映射语句：

```
<insert id="insert">
    INSERT INTO Account (
        username, password
    ) VALUES (
        #username#, #password#
    )
<selectKey
    keyProperty="accountId"
    resultClass="int">
    SELECT SCOPE_IDENTITY()
</selectKey>
</insert>
```

上述示例在插入记录时由数据库创建键，然后抓取已生成的这个键并将它设置在作为参数传递给insert方法的对象中。对于你的应用程序而言，以上这两个已映射语句根本没有任何区别。

在之前的内容中，我们曾经介绍过JDBC 3.0规范中新增的用于获取已生成键的API。但是到目前为止，iBATIS并不支持这个API，因为只有极少数JDBC驱动程序支持这个API。随着越来越多JDBC驱动程序开始实现它，当你需要自动生成的键时，使用这个API也是一个不错的方案。

5.3 更新和删除数据

现在我们已经掌握了如何将数据记录插入到数据库中，知道了被插入数据的已生成键，下面就来研究更新和删除数据。

insert方法返回的是一个对象，然而update和delete方法均返回一个基本类型的整型值（或者更确切地说，一个int值），该值用来表示有多少条记录被已映射语句更新或者删除。

iBATIS框架允许你用一条SQL语句影响数据库中的一条或者多条记录（这主要取决于你的需要）。这是iBATIS和其他大多数对象关系映射工具的区别因素之一，因为其他工具通常只允许一次修改一条记录。

5.3.1 处理并发更新

iBATIS目前还没有实现的是一个功能：为记录提供某种形式的锁，以对相同数据的并发修改进行管理。不过你仍可以使用若干种技术来处理并发更新，例如使用时间戳（timestamp）或者

为数据库表中的每条记录加一个版本号——例如，假设你有一个账户表，定义如下：

```
CREATE TABLE account (  
    accountid serial NOT NULL,  
    username varchar(10),  
    passwd varchar(10),  
    firstname varchar(30),  
    lastname varchar(30),  
    address1 varchar(30),  
    address2 varchar(30),  
    city varchar(30),  
    state varchar(5),  
    postalcode varchar(10),  
    country varchar(5),  
    version int8,  
    CONSTRAINT account_pkey PRIMARY KEY (accountid)  
)
```

更新该表中的某条记录时，你需要在update语句的where子句中同时设置accountId字段和version字段，即同时根据accountId和version的值来查找需要更新的记录，然后在更新记录的同时将其version字段的值加1。这样，当执行更新时，如果正被更新的记录还没有被另一个用户修改，即版本号并没有变化，那么你的更新将获得成功，已映射语句将返回期望的记录计数1。如果返回值为0，且没有抛出任何异常，那么你就知道从数据库中读取了该数据之后又有其他人对其进行了更新。我们保证你能够得到这样的信息，至于在应用程序中如何处理它，则完全由你自己决定。

5.3.2 更新或删除子记录

一个对象包含其他对象作为自己的组件（即对象包含子对象），这种情况非常常见。例如，一个Order对象可能包含一个由多个OrderItem对象组成的列表或数组，其中OrderItem对象表示某条订单项。

由于iBATIS框架从根本上说就是一个SQL映射工具，因此它在更新数据库时并不管理对象间的这些关系。所以，你必须在应用程序的数据层而不是在iBATIS中处理对象关系。用于完成此任务的代码非常简单：

```
public void saveOrder(SqlMapClient sqlMapClient, Order order)  
    throws SQLException {  
    if (null == order.getOrderid()) {  
        sqlMapClient.insert("Order.insert", order);  
    } else {  
        sqlMapClient.update("Order.update", order);  
    }  
  
    sqlMapClient.delete("Order.deleteDetails", order);  
  
    for(int i=0;i<order.getOrderItems().size();i++) {  
        OrderItem oi = (OrderItem) order.getOrderItems().get(i);  
        oi.setOrderId(order.getOrderid());  
    }  
}
```

```
sqlMapClient.insert("OrderItem.insert", oi);
}
```

尽管上面这些代码已经足够完成任务，但是它并没有提供任何形式的事务隔离，因此如果最后一个OrderItem对象更新失败，那么所有其他数据就将处于不一致状态，究其原因，是因为此时事务是发生在每次插入或更新时。此外，每条语句执行之后就会立即被提交，这也将影响程序的性能。下一节中，你将学习如何使用批量更新来处理这两个问题。

5.4 运行批量更新

批量更新是提高iBATIS性能的一种方法。通过创建一批语句，数据库驱动程序可以以一种压缩方式执行更新任务以提高性能。

使用批处理语句（batched statement）的一个重要技巧是：将这些批处理语句打包为一个事务。如果没有这么做，那么每条语句都会开始一个新的事务，这样一来随着批处理语句规模的增长，性能将会受到严重影响。

在5.3.2节中，我们给出了一种方式用于更新包含子对象的对象。那种方式存在两个问题：性能和数据完整性。为解决第二个问题，可以简单地将该方法打包为一个事务，这样更新时如果出现异常，事务就会回滚以保证数据完整性。

这样做的确也可以改进性能，但如果我们还能将语句批量执行（通过iBATIS的startBatch和executeBatch方法），那么就可以进一步提高性能：

```
public void saveOrder(SqlMapClient sqlMapClient, Order order)
throws SQLException {
    sqlMapClient.startTransaction();
    try {
        if (null == order.getOrderID()) {
            sqlMapClient.insert("Order.insert", order);
        } else {
            sqlMapClient.update("Order.update", order);
        }
        sqlMapClient.startBatch();

        sqlMapClient.delete("Order.deleteDetails", order);

        for (int i=0;i<order.getOrderItems().size();i++) {
            OrderItem oi = (OrderItem) order.getOrderItems().get(i);
            oi.setOrderID(order.getOrderID());
            sqlMapClient.insert("OrderItem.insert", oi);
        }
        sqlMapClient.executeBatch();
        sqlMapClient.commitTransaction();
    } finally {
        sqlMapClient.endTransaction();
    }
}
```

理。
理语
入对
下所

后插
行。
递给
的孤

致时
行顺
起执

说明

来。
唯一
增加

一个
和之
项，
单及

5.5

你可能已经注意到了，我们直到父记录更新完成（或者也可能是插入完成）之后才开始批处理。这样做的原因是，当使用一批批处理语句时，直到调用executeBatch()方法执行这批次批处理语句，数据库生成键才会被产生。简单地说，这意味着如果你使用selectKey语句来更新所插入对象的系统生成键，那么它们将为所生成的键返回空值，因此事情并不会像你预期的那样。如下所示：

```
sqlMapClient.startBatch();
sqlMapClient.insert("Account.insert", account);
order.setAccountId(account.getAccountId()); // error!
sqlMapClient.insert("Order.insert", order);
sqlMapClient.executeBatch();
```

上面这个例子看上去似乎无懈可击：我们插入了一个账户，并将所生成的键置入订单对象，然后插入这个订单对象。然而，在调用executeBatch()方法之前，所有这些SQL语句都不会被执行。因此，当执行第3行代码时，账户的accountId特性仍然是空值。当执行第4行代码时，被传递给insert方法的对象就也具有一个空的accountId，这就会在订单表中创建一个没有关联账户的孤立记录。

同时还需要记住：批处理语句只有当PreparedStatement对象和以前的已映射语句完全一致时，才复用这些PreparedStatement对象。当你要执行很多已映射语句，并且这些语句的执行顺序使得它们无法被复用时，就会出现这个问题。因此，如果可能的话，把相同的语句集中起来一起执行。

说明 这种行为几乎是一个bug——虽然并非完全是，但几乎是。如果你不注意语句执行的顺序，很可能有人会攻击SqlExecutor类以使它次序颠倒地复用预备语句（prepared statement）。然而，目前iBATIS并不支持这么做。

一些人用批处理语句来插入一组记录（它们仅靠一个可以很容易确定的值就可以把彼此区分开来。一个典型示例就是：一次操作要将多达200多条票据记录插入到数据库中的系统。这些票据的唯一不同之处就是数据库生成的主键。票据号码有一个初始值，每增加一个票据，相应的号码也会增加1。可以用DAO中的一个循环来完成这项工作，但是使用存储过程将更加迅速也更有效。

在批处理模式下执行语句可能会使性能得到一些提高，而当这些语句可以很容易地被组合成一个存储过程时，使用存储过程（见5.5节）通常可以使性能得到更大的提高。例如，可以使用和之前的更新示例一样的方式来实现deleteOrder方法。然而，要删除一个订单以及相关的订单项，我们需要知道的只是该订单的标识，在此例中就是一个整数。因此，使用存储过程来删除订单及其子元素要比等效的iBATIS代码更加快速方便。

5.5 使用存储过程

存储过程是指在数据库服务器进程中运行的一段代码。尽管大多数存储过程都是使用特定的

数据库语言（通常是基于SQL的）来编写的，但是现在一些开发商也允许使用其他语言（例如，Oracle允许用Java来编写存储过程，微软计划支持用C#来编写存储过程，而PostgreSQL几乎允许使用任何语言）。

5.5.1 优缺点分析

Java开发人员常常把存储过程看作是头号大敌，因为存储过程往往是特定于平台的（特定于数据库平台，不一定特定于操作系统），这冒犯了那些始终强调通用型、跨平台性的敏感的Java开发人员。

对于那些关注问题解决胜过关注所使用的特定解决方案的开发人员，我们认为存储过程对于优化的封装那些复杂的以数据为中心的的问题的解决方案而言，都是一个引人注目的选项。

1. 千万不要走极端！

在何种使用存储过程的讨论上有两种极端。一种极端就是，有些纯Java论者认为应用程序中绝对不应该使用存储过程（并且基于这个论调，甚至有一些人认为SQL本身就不该被使用）。另一个极端是，纯数据库论者坚信所有的数据库交互都应该通过存储过程来执行。

事实上，正如一个古老的谚语所说的那样“纯粹主义者总是错误的（Purists are always wrong）”，这两种极端都是错误的。存储过程只是一种工具，因此我们既不应该高估它也不应该贬低它。作为类比，我们拿木匠为例：他有一把锤子，一把卷尺和一把锯子。尽管他也可以使用锤子来测量一块木板，但是很明显用卷尺会好得多（同样地，用锯子来钉钉子显然没有锤子那么合适）。每一项工作都有一个它最合适的工具，而每一种工具都被设计来完成一项特定的工作。用不恰当的工具来工作，是肯定做不好的。

2. 使用恰当的工具来工作

SQL、存储过程，以及程序代码也是如此。一些操作通过简单的SQL就可以很好地完成，而其他操作可能需要使用存储过程，还有一些则可能最好是代码编程。

例如，假设你为了制作一份报表而需要查询一堆表格。通过代码编程把所有表格中的所有数据“取出——过滤——联结”显然十分繁琐。同样的，为一个简单的SQL查询创建存储程序也只是没事找事，并没有带来多少价值。将这个SQL添加到iBATIS已映射语句中并用iBATIS执行它，才是这种情况下解决问题的快速、简单又高效的方案。

对于更复杂的报表，如果必须要做多个子查询，并且要从包含数百万行的表格中获取数据，此时使用存储程序就更加明智。使用存储程序，就可以有多种选项用于进行优化。

对于需要使用动态SQL的应用程序，已映射语句也很有用。在第8章中，我们将给出一个使用动态SQL的示例，其中SQL语句是通过使用Java、存储程序，以及包含动态元素的已映射语句而创建的。我们并不想破坏这个惊喜，不过如果你迫不及待想要弄清怎么回事，就可以直接跳到8.5节先睹为快。

有关存储过程另一个需要考虑的就是：可以调用它们来更新和返回数据。这时就可能需要事务，而如果事务没有提交，就可能会导致问题，因为通常用来调用存储过程的方法并不要求一定要提交。在这些情况下，事务管理器需要被配置为总是提交（即使在读操作之后也要提交），或者你必须人工管理事务，如下例所示：

```
try {
    sqlMapClient.startTransaction();
    sqlMapClient.queryForObject("Account.insertAndReturn", a);
    sqlMapClient.commitTransaction();
} finally {
    sqlMapClient.endTransaction();
}
```

有关事务的更多细节将在第7章中介绍，因此如果你有这方面的疑问，请参考第7章。

5.5.2 IN、OUT 和 INOUT 参数

到目前为止，我们所接触到的参数都只是仅仅用于输入——你把它们传递给iBATIS，并且这些参数在iBATIS的执行过程中始终保持不变（除了元素之外）。对于存储过程，你有3种类型的参数可选：OUT和INOUT。

在存储过程中使用IN参数与直接给iBATIS传递参数一样的非常简单。把一个参数传递到任何其他已映射语句中一样，可以把IN参数传递到存储过程中。以下这个简单的存储过程接受两个IN参数并且返回一个值：

```
CREATE OR REPLACE FUNCTION max_in_example
(a float4, b float4)
RETURNS float4 AS
$BODY$
BEGIN
    if (a > b) then
        return a;
    else
        return b;
    end if;
END;
$BODY$
LANGUAGE 'plpgsql' VOLATILE;
```

以下是使用这个存储过程的参数映射、已映射语句以及Java代码：

```
<parameterMap id="pm_in_example" class="java.util.Map">
    <parameter property="a" />
    <parameter property="b" />
</parameterMap>
<procedure id="in_example" parameterMap="pm_in_example"
    resultClass="int" >
    { call max_in_example(?, ?) }
</procedure>
```

```
// Call a max function
```

```
Map m = new HashMap(2);
m.put("a", new Integer(7));
m.put("b", new Integer(5));
Integer val =
    (Integer) sqlMap.queryForObject("Account.in_example", m);
```

INOUT参数是指那些被传递给一个存储过程并且可以被其修改的参数，如下例所示，该例子接受两个数字并将它们交换。以下是这个过程的代码（使用Oracle PL/SQL）：

```
create procedure swap(a in out integer, b in out integer) as
temp integer;
begin
    temp := a;
    a := b;
    b := temp;
end;
```

以下是使用这个存储过程的参数映射、已映射语句以及Java代码：

```
<parameterMap id="swapProcedureMap" class="java.util.Map">
    <parameter property="a" mode="INOUT" />
    <parameter property="b" mode="INOUT" />
</parameterMap>

<procedure id="swapProcedure" parameterMap="swapProcedureMap">
    { call swap(?, ?) }
</procedure>

// Call swap function
Map m = new HashMap(2);
m.put("a", new Integer(7));
m.put("b", new Integer(5));
Integer val =
    (Integer) sqlMap.queryForObject("Account.in_example", m);
```

OUT参数稍微有些特殊。它们和返回结果（例如resultMap结果）很相似，但是又可以像参数那样被传递给存储过程。被传递给存储过程的值将被忽略，然后被存储程序所返回的值所替代。OUT参数可以返回任何数据，从一个简单的值（如下个例子所示）到一组完整的记录（例如Oracle REF_CURSOR的情况）。

以下是一个很小的存储程序的例子，它使用了2个IN参数和1个OUT参数（Oracle PL/SQL）：

```
create or replace procedure maximum
(a in integer, b in integer, c out integer) as
begin
    if (a > b) then c := a; end if;
    if (b >= a) then c := b; end if;
end;
```

以上这个存储过程接受3个参数而不返回任何值。然而，第3个参数只是用于输出，根据另外两个参数的大小，它取值为其中较大的那个。要使用iBATIS来调用这个程序，可以创建一个参数映射集和一个已映射语句：


```
<parameterMap id="maxOutProcedureMap" class="java.util.Map">
  <parameter property="a" mode="IN" />
  <parameter property="b" mode="IN" />
  <parameter property="c" mode="OUT" />
</parameterMap>
<procedure id="maxOutProcedure"
  parameterMap="maxOutProcedureMap">
  { call maximum (?, ?, ?) }
</procedure>

// Call maximum function
Map m = new HashMap(2);
m.put("a", new Integer(7));
m.put("b", new Integer(5));
sqlMap.queryForObject("Account.maxOutProcedure", m);
// m.get("c") should be 7 now.
```

正如本章前面所提到的那样，存储过程也可以返回多行数据。这种能力可以用来极大地提高在大型数据集中进行复杂查询（此时性能无法用传统的SQL优化技术来优化）的性能。无法用传统的SQL优化技术优化的操作的例子有：外部联结（outer join）和需要进行计算的where子句。如果你计划在应用程序中使用存储程序来优化性能，那么应该确保充分理解影响性能的真正瓶颈所在，否则你非但不能节省时间反而会最终浪费时间。

5.6 小结

本章中，我们学习了使用iBATIS在数据库中修改数据的几乎所有方法。通过阅读第4章和本章，你应该已经能够创建使用iBATIS框架来维护数据的应用程序了。

第6章将继续扩展你已经学过的知识。我们将研究更高级的查询技术，以帮助你更好地利用iBATIS框架，不负你在学习数据库技巧以及数据库平台时所付出的努力。

第 6 章

使用高级查询技术

本章内容

- 使用XML
- 声明关系
- N+1问题解决方案

除了前两章介绍的简单数据库操作之外，iBATIS还可以用来执行更加高级的任务。本章将研究一些iBATIS的高级技术，它们可以帮助你减少大量的代码编写工作，并改进你的应用程序的性能、最小化发布时的应用程序的大小^①。

6.1 在 iBATIS 中使用 XML

有时候可能需要使用基于XML的数据。iBATIS框架允许你使用XML为一个数据库查询传递参数，也可以用XML从查询中返回结果。不论是作为参数还是返回值，如果并非必需，使用XML其实并不会给你带来什么价值，相反使用POJO (plain old Java object) 在大多数情况下也许更加有效。

此外，上面这个特征很有可能会在iBATIS的下一个主要版本中被删除，这两个原因。其中一个原因随着本节中对此功能性的深入介绍就会变得很明显。另一个原因则是，此特征和iBATIS框架的基本理念不符，其实iBATIS框架的基本理念很简单，就是为了简化对对象的映射查询。

不过本节仍然会讨论一下这个特征，因为可能在某些系统中你必须使用它；但是我们也将介绍一些变通方式，这样即使它确实消失了，你也不至于很无助地去寻找一种使应用程序仍能继续正常工作的方式。

6.1.1 XML 参数

要使用XML来向已映射语句传递参数，既可以通过String值也可以通过DOM对象来完成，这两种方式具有完全相同的结构。

^① 原文为footprint of your application，意为应用程序部署时留下的痕迹，此处意义为应用程序的大小。——译者注

使用XML作为参数时其结构并非是严格的XML，而只是一段格式良好的XML片段。在这种结构中，参数元素包装了要传递的名值对，名值对的“值”被包装在表示“名”的元素中。例如：

```
<parameter><accountId>3</accountId></parameter>
```

这个例子中，已映射语句将获得名为accountId的参数，它的值为3。以下示例就使用我们刚才介绍的XML字符串来向已映射语句传递参数：

```
<select id="getByXmlId" resultClass="Account" parameterClass="xml">
  select
    accountId,
    username,
    password,
    firstName,
    lastName,
    address1,
    address2,
    city,
    state,
    postalCode,
    country
  from Account
  where accountId = #accountId#
</select>
```

```
String parameter = "<parameter><accountId>3</accountId></parameter>";
Account account = (Account) sqlMapClient.queryForObject(
    "Account.getByXmlId",
    parameter);
```

同样地，也可以使用DOM对象来给iBATIS传递参数，并且得到完全相同的结果：

```
<select id="getByDomId" resultClass="Account" parameterClass="dom">
  select
    accountId,
    username,
    password,
    firstName,
    lastName,
    address1,
    address2,
    city,
    state,
    postalCode,
    country
  from Account
  where accountId = #accountId#
</select>
```

```
Document parameterDocument = DocumentBuilderFactory.newInstance()
    .newDocumentBuilder().newDocument();
Element paramElement = parameterDocument
    .createElement("parameterDocument");
Element accountIdElement = parameterDocument
```



```
.createElement("accountId");
accountIdElement.setTextContent("3");
paramElement.appendChild(accountIdElement);
parameterDocument.appendChild(paramElement);
Account account = (Account) sqlMapClient.queryForObject(
    "Account.getByXmlId", parameterDocument);
```

正如我们之前所提到的那样，仅仅是将你的参数转换成XML格式就需要大量的代码。然而，如果你正在使用像Cocoon^①这样的工具，或者你正在在一个无法将XML转变为对象的框架中编写Web服务，那么此时iBATIS能够处理XML这一特征就显得非常有用。根据你开始时的XML结构，也许使用XSL将它转变成iBATIS所要求的结构，然后作为XML参数直接交给iBATIS处理，比起由你亲自处理XML并将它转变为Java对象，再作为bean参数交给iBATIS处理（这是你在这种情况下下的另一种可选方案），前者要比后者来得更加简单。

6.1.2 XML 结果

iBATIS框架也允许你从已映射语句中创建XML格式的结果。当执行一条返回XML的已映射语句时，iBATIS会为每一个返回对象返回一份完整的XML文档。

要使用该特征，你需要创建一个具有xml结果类的已映射语句。以下就是一个简单示例：

```
<select id="getByIdValueXml" resultClass="xml"
    xmlResultName="account">
    select
      accountId,
      username,
      password
    from Account
    where accountId = #value#
</select>

String xmlData = (String) sqlMap.queryForObject(
    "Account.getByIdValueXml",
    new Integer(1));
```

此时返回的结果将如下所示（实际返回的结果可能与此稍有不同，我们添加了一些空格和换行以使其更容易阅读）：

```
<?xml version="1.0" encoding="UTF-8"?>
<account>
  <accountId>1</accountId>
  <username>lmeadors</username>
  <password>blah</password>
</account>
```

如果希望获取的记录只有一条，那么以XML文档的方式获取这条记录还是非常方便的。但

① Cocoon是Stefano Mazzocchi于1999年1月创建的ASF之下的开放源代码项目。其目标是帮助分离内容格式、逻辑和对基于XML网站的管理功能。Cocoon使用XML，XSLT——Extensible Stylesheet Language Transformations，以及SAX——Simple API for XML技术，以帮助创建、部署和维护XML服务器应用程序。它支持大多数类型的数据源，包括RDBMS、LDAP和文件系统。——译者注

如果想要获取多个对象，如下所示（其实它与仅获取一条记录在形式上是一样的），也许事情就不再那么简单了。

```
<select id="getAllXml" resultClass="xml" xmlResultName="account">
  select
    accountId,
    username,
    password,
    firstName,
    lastName,
    address1,
    address2,
    city,
    state,
    postalCode,
    country
  from Account
</select>
```

```
List xmlList = sqlMap.queryForList("Account.getAllXml", null);
```

以上这个示例中得到的结果列表是一个XML文档（或者说字符串）的列表。这恰好是你所需要的，是吗？实际上，有时候你可能的确希望得到这种结果列表；但是在大多数情况下并不是这样的。你希望得到的是单个的XML文档（或者说字符串），其中包含很多的账户元素，而不是像前一个结果的字符串列表，这意味着如果你要把这些字符串都拼接成单个的文档，你就必须手工完成。这显然不是最优方案。

要解决这个问题，方案之一就是不要使用iBATIS来得到XML结果，而是使用一个普通的iBATIS已映射语句（它返回一个普通的bean集合），然后从该集合中手工创建XML。以下是一个用于创建XML的辅助方法，如果你也使用bean来获取结果，就可以创建一个类似的方法：

```
public String toXml(){
    StringBuffer returnValue = new StringBuffer("");
    returnValue.append("<account>");
    returnValue.append("<accountId>" + getAccountId() + "</accountId>");
    returnValue.append("<username>" + getUsername() + "</username>");
    returnValue.append("<password>" + getPassword() + "</password>");
    returnValue.append("</account>");
    return returnValue.toString();
}
```

该问题的另一个解决方案就是创建一个实用类，利用反射将bean自动转换为XML。可以自己试着写一个这样的方法。以下这个简洁的实用方法（utility method）就可以完成此任务，可以以它作为参考。虽然我们为了节省篇幅而对代码作了简化，但是它仍然足够展示此技术。

```
public class XmlReflector {
    private Class sourceClass;
    private BeanInfo beanInfo;
    private String name;

    XmlReflector(Class sourceClass, String name) throws Exception {
```

```
this.sourceClass = sourceClass;
this.name = name;
beanInfo = Introspector.getBeanInfo(sourceClass);
}

public String convertToXml(Object o) throws Exception {
    StringBuffer returnValue = new StringBuffer("");
    if (o.getClass().isAssignableFrom(sourceClass)) {
        PropertyDescriptor[] pd = beanInfo.getPropertyDescriptors();
        if (pd.length > 0) {
            returnValue.append("<" + name + ">");
            for (int i = 0; i < pd.length; i++) {
                returnValue.append(getProp(o, pd[i]));
            }
            returnValue.append("</" + name + ">");
        } else {
            returnValue.append("<" + name + "/>");
        }
    } else {
        throw new ClassCastException("Class " + o.getClass().getName() +
            " is not compatible with " + sourceClass.getName());
    }
    return returnValue.toString();
}

private String getProp(Object o, PropertyDescriptor pd)
    throws Exception {
    StringBuffer propValue = new StringBuffer("");
    Method m = pd.getReadMethod();
    Object ret = m.invoke(o);
    if (null == ret) {
        propValue.append("<" + pd.getName() + "/>");
    } else {
        propValue.append("<" + pd.getName() + ">");
        propValue.append(ret.toString());
        propValue.append("</" + pd.getName() + ">");
    }
    return propValue.toString();
}
}
```

使用以上这个样例类可以很容易地接受一个bean并将它转换为一个XML片段，而不是一个完整的XML文档。以下是这个类的使用示例：

```
XmlReflector xr = new XmlReflector(Account.class, "account");
xmlList = sqlMap.queryForList("Account.getAll", null);
StringBuffer sb = new StringBuffer(
    "<?xml version='1.0' encoding='UTF-8' ?><accounts>");
for (int i = 0; i < xmlList.size(); i++) {
    sb.append(xr.convertToXml(xmlList.get(i)));
}
sb.append("</accounts>");
```

从内存消耗的角度考虑，如果需要处理的记录数量巨大，那么使用以上技术将耗费大量的内

存——内存中将保存已映射语句返回的对象列表中的所有对象，并且还要使用字符串高速缓存来构建XML文档。在6.3节中，在介绍更有效的方式以处理大型结果集时，我们将重新讨论这个问题。

6.2 用已映射语句关联对象

iBatis框架也提供了多种方式用于关联复杂对象，例如订单及其明细（以及它们对应的产品、顾客等）关联。就像大多数事情一样，每种方法都有各自的优缺点，没有一个解决方案是绝对正确的。应该根据需求来选择最恰当的那一个。

说明 为了简洁，本章其余的示例中，将省略bean中那些对于展现技术而非必需的数据属性。例如，当获取顾客时，我们将不再获取该顾客的所有字段，而只是获取它的主键和外键。

6.2.1 复杂集合

在第4章中，你已经学习了如何使用SELECT语句从数据库中获取数据。在那些示例中，我们在结果中只使用了一种对象类型，即使是在联结多个表的情况下。其实也可以使用iBatis来加载更复杂的对象。

如果你想让应用程序的对象模型看上去就像数据模型^①一样，iBatis这种加载复杂对象的能力就显得非常有用。有了这种能力，使用iBatis时就可以根据相关联的对象来定义数据模型，并且让iBatis立即加载到它们。例如，假设你有一个数据库，其中Account记录和Order记录存在关联，而Order记录又关联OrderItem记录，建立起这些关联关系之后，请求Account记录时，就会同时获得所有关联的Order对象以及OrderItem对象。代码清单6-1显示了应该如何定义SQL映射以完成这项任务。

代码清单6-1 映射复杂集合^②

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE sqlMap
PUBLIC "-//ibatis.apache.org//DTD SQL Map 2.0//EN"
"http://ibatis.apache.org/dtd/sql-map-2.dtd">
```

① 数据模型即关系模型，数据模型的概念可参考任何一本有关数据库原理的图书。——译者注

② 代码清单6-1中第8行代码之所以为account.accountId而不是account.Id，读者推断是因为AccountInfo的代码如下：

```
package org.apache.mapper2.examples.bean;
import org.apache.ibatis.jgamestore.domain.Account;
import org.apache.ibatis.jgamestore.domain.Order;
import java.util.ArrayList;
public class AccountInfo {
    private Account account;
    private List<Order> orderList = new ArrayList<Order>();
    // account和OrderList的getter和setter
    // 其他属性
}
```

```
<sqlMap namespace="Ch6">

  <resultMap id="ResultAccountInfoMap"
    class="org.apache.mapper2.examples.bean.AccountInfo">
    <result property="account.accountId"
      column="accountId" />
    <result property="orderList"
      select="Ch6.getOrderInfoList"
      column="accountId" />
  </resultMap>

  <resultMap id="ResultOrderInfoMap"
    class="org.apache.mapper2.examples.bean.OrderInfo">
    <result property="order.orderId" column="orderId" />
    <result property="orderItemList" column="orderId"
      select="Ch6.getOrderItemList" />
  </resultMap>

  <resultMap id="ResultOrderItemMap"
    class="org.apache.mapper2.examples.bean.OrderItem">
    <result property="orderId" column="orderId" />
    <result property="orderItemId" column="orderItemId" />
  </resultMap>

  <select id="getAccountInfoList"
    resultMap="ResultAccountInfoMap">
    select accountId
    from Account
  </select>

  <select id="getOrderInfoList"
    resultMap="ResultOrderInfoMap">
    select orderId
    from orders
    where accountId = #value#
  </select>

  <select id="getOrderItemList"
    resultMap="ResultOrderItemMap">
    select
      orderId,
      orderItemId
    from orderItem
    where orderId = #value#
  </select>
</sqlMap>
```

如果研究一下结果映射（ResultAccountInfoMap ①、ResultOrderInfoMap ②和ResultOrderItemMap ③），你就会发现前两个结果映射为每一个映射特性都使用了select属性。这个属性告诉IBATIS使用另一个已映射语句（该已映射语句的名称就由select属性的值来确定）的结果来设置该特性的值。例如，当运行getAccountInfoList这个已映射语句时④，在ResultAccountInfoMap结果映射中就有<result property="orderList" select="Ch6.

getOrderInfoList" column="accountId" />。这段 XML 告诉 iBATIS 通过运行“Ch6.getOrderInfoList”这条已映射语句⑥来获取orderList特性的值（将accountId列的值作为参数传递给已映射语句，然后将返回值放入orderList）。同样地，在结果映射ResultOrderInfoMap②中，也需要通过执行getOrderItemList⑥已映射语句来获取orderItemList特性的值。

尽管使用select属性的确给我们带来了一些方便，但是也会出现两个问题。第一，创建包含大量对象的列表将消耗大量的内存。第二，由于一种被称为“N+1查询”问题（有关此问题稍后将作详细讨论）的现象，这种方式可能会很快地造成数据库I/O问题。iBATIS框架提供了针对每一个问题的解决方案，但是还没有能同时解决这两个问题的方案。

1. 数据库I/O

数据库I/O可以很好地度量你的数据库的使用情况，同时也是影响数据库性能的主要瓶颈之一。当从数据库中读数据或者往里写数据时，数据必须从磁盘转移到内存或者从内存转移到磁盘，这种操作非常耗时。使用高速缓存来避免数据库I/O可以使你的应用程序更加快速，但是使用这种策略如果不小心就会出问题。有关何时以及如何使用iBATIS提供的高速缓存机制，请参考第9章。

为了说明你在使用关联数据时可能遇到的数据库I/O问题，我们假设你有1 000条Account记录，每条记录和1 000条Order记录关联，而每条Order记录又关联25条OrderItem记录。如果试图把所有这些数据都加载到内存中，就需要执行超过1 000 000条SQL语句（1条语句查找所有账户，查找订单需1 000条语句，查找关联订单项OrderItem需要1 000 000条语句），并且需要创建大约2 500万个Java对象——你要真是这么做的话，系统管理员肯定要气得扇你的耳光了！

2. “N+1查询”问题

N+1查询问题是由于试图加载和父记录列表相关的子记录而造成的。因此，如果执行一个查询语句并获取N条父记录，那么为了获得所有这些父记录的子记录，你就必须再执行N个查询，这就产生了“N+1查询”。

3. 这些问题的解决方案

延迟加载（6.2.2节中有更详细的讨论）可以通过将加载过程分割成更小也更易管理的小过程，来减轻内存的负担。然而，这样做仍然没有解决数据库I/O问题，因为在最坏情况下它仍会同非延迟加载版本一样在加载数据时使用N+1查询，从而对你的数据库造成同样的冲击。6.2.3节中会给出一种解决方案，当我们解决了N+1查询问题，从而减小数据库I/O，尽管可以使用单条数据库查询同时加载所有的关联对象，但我们的内存将有一大块被这25 000 000条记录所占据。

既然“数据库I/O”或“N+1查询”两个问题必有其一，你现在是否在考虑不使用复杂特性了呢？其实这需要根据你对数据库的理解以及应用程序使用数据库的方式来决定。如果你正确使用本节所描述的技术，就可以省去大量的编程工作，但是如果误用这些技术，就可能把事情弄得一团糟。在下面的两节中，我们将研究如何根据你的目的来决定使用何种策略。

首先要问一个问题：我们的例子设计为将账户和订单关联，同时订单又和订单项关联，这是一个展示何时需要将数据关联起来的好例子吗？实际上，不是的——订单到订单项的关系是固定的，但是账户到订单的关系却并不是必需的。

原因就是，订单项如果没有对应的订单就不是一个完整的实体，然而账户却是完整的实体。从你使用它们的方式这一角度来考虑一下。通常，对于没有订单项的订单，你不可能用它来做些什么，反过来说，没有对应订单的订单项从某种程度上说是毫无意义的。但是，账户就可以被看作是一个完整的实体对象。

然而，对于我们例子的目的而言，目前的这种关系使用了大家所熟知的可以理解的概念（账户具有订单，而订单具有订单项），已经很好地展示了我们的技术，因此我们将继续使用这个关系。

6.2.2 延迟加载

我们将讨论的第一种解决方案就是延迟加载。如果不是同时需要所有相关联的数据时，延迟加载就会非常有用。例如，如果应用程序调用了—个Web页来显示所有的账户，某个销售代表（应用程序的用户）可能会单击某个账户以便查看该账户的所有订单，然后再单击某个订单以便查看该订单的所有细节。这种情况下，任何时候我们所需要的都只是一个列表。此时使用延迟加载就非常合理。

要使用延迟加载，需要编辑SqlMapConfig.xml文件，通过在<setting>元素中将lazyLoadingEnabled属性改为true来启用它。如果想要使用延迟加载的cglib增强版，则必须下载并将其添加到应用程序的类路径中，同时也必须将<setting>元素中的enhancementEnabled属性改为true，必须注意的是，这是一个全局设置，因此如果启用了这些特性，SQL映射中所有的已映射语句都会使用延迟加载。

一旦启用了延迟加载功能，就可以使需要创建的对象数目和数据库I/O次数更加合理。对于一个深入到订单细节的用户，只需要3次查询（一次针对于账户，另一次针对订单，最后一次针对订单项），并且应用程序将仅仅创建2 025个对象（1 000个账户，1 000个订单，以及25个订单项）。根本不需要修改应用程序的任何代码，只需要稍微修改iBATIS的XML配置即可完成此任务。

在我们进行的一个完全非科学测试中，使用非延迟加载关系加载数据以获取第一个列表所花的时间是延迟加载版本的3倍。然而，如果要获取所有的数据，延迟加载版本所耗费的时间要比非延迟加载多20%。很明显，消耗的时间很大程度上取决于要加载的数据量。因此，是否使用延迟加载，和多数事情一样，经验是最好的导师。

有时候可能你并不想推迟所有数据的加载，而是希望第一次请求数据时就加载所有的数据。此时，可以使用下一节将要描述的技术，那种技术恰好可以完成此任务：一次查询就完成所有数据的加载，而不需要多次查询。同时，这种方式避免了N+1查询问题。

6.2.3 避免 N+1 查询问题

有两种方式可以避免“N+1查询”问题。一种方式是通过使用iBATIS提供的groupBy属性，另一种是通过使用一个称为RowHandler的自定义组件。

使用groupBy属性和前面的技术非常相似。简单地讲就是，使用结果映射来定义数据间的关联关系，然后将最顶层的结果映射与某个已映射语句关联起来。以下示例构建了一个和之前的延迟加载示例完全相同的结构，但是由于使用了联结查询（join query），此例只会在数据库服务器上执行一条SQL语句。

这里包含了3个结果映射：一个用于获取账户信息，一个用于获取订单信息，还有一个用于获取订单项信息。

用于获取账户信息的结果映射具有以下3个功能：

- 映射账户对象中的特性。
- 告诉iBATIS根据哪个特性来创建新的账户对象。
- 告诉iBATIS如何映射关联对象集合，此处就是指与账户相关的一组订单对象。

此处需要注意的一件很重要的事就是，groupBy属性值应该在引用一个特性名而不是一个列名。

用于获取订单信息的结果映射也具有以下3个相同的功能：

- 将订单数据映射到订单对象。
- 告诉iBATIS哪个特性指示一个新订单。
- 告诉iBATIS哪个结果映射将被用于任何子记录。

最后，用于获取订单项信息的结果映射是一个标准的结果映射，它只是用来将订单项映射到对象。代码清单6-2展示了此例中的映射。

代码清单6-2 使用N+1查询方案

```
<resultMap id="ResultAccountInfoNMap" <-- ① 声明用于获取账户数
    class="AccountInfo"
    groupBy="account.accountId" >
    <result property="account.accountId"
        column="accountId" />
    <result property="orderList"
        resultMap="Ch6.ResultOrderInfoNMap" />
</resultMap>

<resultMap id="ResultOrderInfoNMap" <-- ② 声明用于获取订单数
    class="OrderInfo"
    groupBy="order.orderId" >
    <result property="order.orderId" column="orderId" />
```

```
<result property="orderItemList"
      resultMap="Ch6.ResultOrderItemNMap" />
</resultMap>

<resultMap id="ResultOrderItemNMap"
      class="OrderItem">
  <result property="orderId"
        column="orderId" />
  <result property="orderItemId"
        column="orderItemId" />
</resultMap>

<select id="getAccountInfoListN"
      resultMap="ResultAccountInfoNMap">
  select
    account.accountId as accountId,
    orders.orderid as orderid,
    orderitem.orderitemid as orderitemid
  from account
  join orders on account.accountId = orders.accountId
  join orderitem on orders.orderId = orderitem.orderId
  order by accountId, orderid, orderitemid
</select>
```

3 声明用于获取订单
项信息的结果映射

4 在<select>元素
中使用结果映射

通过调用getAccountInfoListN已映射语句④，可以获取和前两个示例中所得的数据（一个账户列表，每个账户都有一个订单列表，而每个订单又具有相关的订单项作为列表属性）相同的数据，但是由于我们现在执行的只有一条SQL语句，因此它将快得多。执行getAccountInfoListN已映射语句④时，通过使用了groupBy属性的ResultAccountInfoNMap结果映射①来映射结果。groupBy这个属性可以告诉iBATIS，只有当account.accountId特性改变时，它才需要创建一个新的AccountInfo实例。另外，由于orderList特性被映射到ResultOrderInfoNMap结果映射②上，故该特性将被填充为所执行的查询得到记录。由于ResultOrderInfoNMap结果映射也使用了groupBy属性，因此iBATIS的整个处理过程完全相同，也就是说，由于orderItemList字段被映射到ResultOrderItemNMap结果映射③上，orderItemList将被赋以所执行的查询得到结果集合。

使用之前做的完全非科学测试，对于小数据集，使用含groupBy属性的结果映射集，性能可以提高到原来的七倍。但我们怀疑，对于我们一开始所举的（包含2 500万记录的）例子而言，不论使用select属性还是groupBy属性，程序都无法正常运行。

必须牢记，使用groupBy属性尽管性能得到改善，但内存的消耗仍然和非延迟加载版本是一样的。所有的记录都被一次性读入内存中，因此，即使它获取对象列表的速度的确更快了，但内存的大量消耗仍然是一个问题。

底线就是，根据需要，某种技术可能会帮助你解决问题。那么该如何选择呢？表6-1提供了指导。

表6-1 延迟加载和N+1查询方案的差异

延 迟 加 载	N+1查询方案
此方式适用于获取那些虽然较为大型但并非其每条记录都会被用到的那些数据集	此方式适用于小型数据集或者所有数据都肯定会被使用的数据集
前期性能可以得到提高,但之后你可能会为付出一定的代价	整体性能得到提高

因此，以上就是关于如何映射复杂结果的所有内容。下面讨论iBATIS的一些其他用途。

6.3 继承

继承是面向对象编程中的一个基本概念。继承允许从基类中扩展出新的类，从而有效地继承基类的字段以及方法，甚至从多个基类中创建一个继承体系。扩展得到的新类可以覆盖基类已有的方法，以便增强或者替换其功能。继承，作为面向对象的语言（例如Java）的特征，有许多意义，包括：

- 代码复用——可以构建一个包含大量公共逻辑的抽象的基类，但是该基类本身又是不完整的。扩展该基类的子类可以复用基类的公共特征，以使自己功能性完备。因此，你可以为某个特征提供多个实现，同时不需要重写或者复制公共的部分。
- 功能的增强与特殊化——有时候，你可能决定扩展一个类以便添加更有用的特征。扩展集合类（collection class）通常就是基于这个原因。例如，你可能想要扩展ArrayList类，以得到一个只支持字符串的集合类StringArray。这样你就可以在其中添加新的特征，例如根据正则表达式进行搜索的功能。
- 公共接口——虽然使用一个实际的接口而不是一个抽象类通常是更好的选择，但是在一个框架或者其他可扩展系统中使用基类作为公共接口也是可能的。

继承还有很多其他的好处，但是它也存在着风险。例如对商业领域问题的建模，继承显得不够灵活，原因是商业领域的问题通常太过复杂，仅仅使用单个的继承体系通常无法表达。尤其是在Java语言中，只有单继承，没有多重继承，你只能从单个超类中扩展。但即使你所选择的编程语言支持多重继承，使用继承可能仍然不是最佳选择。

近些年出现的许多模式以及最佳实践为继承机制提供了许多很有帮助的帮助方案。这些方案包括：

- 使用组合（composition）而不是继承——一个很好的例子就是人所扮演的角色。通常一个不好的设计会从Employee类派生得到Manager类，而Employee类本身又是从Person类继承而来。也就是说，Manager（经理）是一个Employee（雇员），而Employee（雇员）又是一个Person（人）。这当然是正确的，但是如果同一个人又是一个Customer（顾客），那该怎么办呢？例如，一位Manager（经理）决定在自己的商店里买东西。当一个人既是Customer又是Employee时，它将处于我们的这个继承体系中的什么地方呢？当然了，Customer不一定是一个Employee，而Employee也不一定是一个Customer。因此，之前的继承体系无法表达我们的需求。其实，除了使用继承机制建模之外，还可以考虑使用组合。

通过在Person类中保存一个Roles（角色）集合来把是一个（is a）关系改为有一个（has a）关系。如此一来，一个人就可以拥有所描述的所有角色的任意组合。

- 使用接口而不是抽象类——当想要用一个公共接口来描述某组功能时，最好直接使用实际的接口类型，而不是抽象类。要完成代码复用以及接口和实现之间的分离这两项任务，一个很好的方式就是三层设计。在三层设计中，你仍然使用抽象类来获得某种程度的代码复用，但却设计使用抽象类来实现某个接口，然后仅仅把这个接口暴露给公有的API。最后，扩展抽象类得到具体类，这样这些具体类就继承实现该接口。

映射继承

iBatis通过使用一个特殊的被称为鉴别器（discriminator）的结果映射来支持继承体系。使用鉴别器你可以根据数据库中某列的值来确定要实例化的类的类型。鉴别器是结果映射的一部分，它和switch语句的工作原理很相似，例如：

```
<resultMap id="document" class="testdomain.Document">
  <result property="id" column="DOCUMENT_ID"/>
  <result property="title" column="TITLE"/>
  <result property="type" column="TYPE"/>
  <discriminator column="TYPE" javaType="string">
    <subMap value="Book" resultMap="book"/>
    <subMap value="Newspaper" resultMap="news"/>
  </discriminator>
</resultMap>
```

以上这个鉴别器可以这样来理解：

如果TYPE列取值为Book，那么就使用名为book的结果映射；否则，如果TYPE取值为Newspaper，那么就使用名为news的结果映射。

在以上的例子中我们看到的子映射（sub map）就是一些标准的结果映射，可以通过名字来引用（如下例所示）。如果鉴别器不能找到一个值来匹配其中的某个子映射，那么就会应用父结果映射。如果找到了合适的值，那么就仅仅应用子映射——此时，所定义的父亲映射集将不会被应用，除非子映射明确地扩展了这个父映射，如下例所示。

```
<resultMap id="book" class="testdomain.Book" extends="document">
  <result property="pages" column="DOCUMENT_PAGENUMBER"/>
</resultMap>
```

结果映射的extends属性告诉iBatis复制其指向的结果映射中的所有结果映射。然而，它与我们的类层次体系没有任何关系。请牢记，iBatis不是一个O/RM框架。它只是一个SQL Mapper。它与O/RM的区别就在于iBatis不知道也不关心类和数据库表之间的映射。因此，iBatis不要求子映射一定要使用一个从父结果映射所引用的类而扩展得到的子类。也就是说，你可以随意地使用鉴别器，只要使用起来看上去很自然。当然了，继承是一个很明显的的应用，但是你也可能会发现在某些其他情况下使用鉴别器也会很方便。

6.4

<S

6.4.

<del

示了：
用它

模式
变数

来执

<

s

手

写创

特征

6.4.2

有

其实，
一些技

例

格式输
据我们
有的数
案例中

①

②

6.4 其他用途

iBATIS框架的设计意图就是要灵活。当无法使用其他类型的已映射语句时，也许就可以使用<statement>已映射语句。

6.4.1 使用语句类型和 DDL

<statement>类型的已映射语句有一点奇怪，它和其他类型（例如<insert>、<update>、<delete>以及<select>）的已映射语句不同的地方就是，它没有对应的方法^①可以调用。这就暗示了：我们不鼓励使用<statement>类型的已映射语句，只有在别无选择的情况下才可以考虑使用它。

DDL（Data Definition Language，数据定义语言）是SQL语言的一个子集，用来定义数据库模式（database schema）的结构。可以使用DDL来定义表和索引，并且可以用它来执行那些不改变数据而是改变数据结构^②的操作。

虽然iBATIS并没有正式支持DDL，但是数据库可能会允许你通过iBATIS的<statement>类型来执行DDL语句。例如，PostgreSQL数据库就允许用<statement>语句创建和删除数据库表：

```
<statement id="dropTable">
    DROP TABLE Account CASCADE;
</statement>

sqlMap.update("Account.dropTable", null);
```

我们并不能保证所有的数据库都支持以这种方式运行DDL语句，但是如果它支持的话，当编写创建或修改数据库结构的程序时，这种方式就非常方便。

下一节将讨论一个使得iBATIS显得非常灵活的特征，只是很可惜它常常为使用者忽略，这个特征就是行处理器（row handler）。

6.4.2 处理超大型数据集

有时候，你可能会发现为了满足某些应用程序的需求似乎必须使用大型数据集（dataset）。其实，在多数情况下，这样的需求总是可以通过其他方式得到满足的，例如，可以针对需求提出一些探索性的问题，通过对这些问题的回答，你就有可能透过需求的表面而揭露出需求的本质。

例如，假设交给你一份需求文档，要求将一份有30 000行记录的数据表的全部内容以HTML格式输出给用户浏览。这时你首先应该问的一个问题就是，“用户确实需要所有的数据么？”根据我们的经验，于情于理答案通常都不会是“yes”。尽管我们并不怀疑用户确实可能需要浏览所有的数据，但是我们一定要弄清楚他们是否会真的需要一次性浏览这所有的数据。几乎在所有的案例中，我们都发现用某种方式过滤一下返回的数据，会比那种仅仅把“select * from table”的

① 如insert方法、update方法。——译者注

② 此处即指数据库模式。——译者注

查询结果直接“倾倒”到显示屏上的方式能取得更好的效果。

如果数据库查询的结果不是为了输出，而是用于处理，那么就必须认真考虑一下使用存储过程是否会比写Java代码更好些。虽然存储过程常常被看作是对Java目标“一次编写，随处运行”的一种背叛，但是我们也曾在应用程序中遇到过使用纯Java需要10~15分钟才能完成的任务，而使用存储过程却只需要不到10秒的情况。一个系统的用户并不会关心应用程序是否是用纯Java编写的，他们关心的只是是否能使用你的系统来完成工作。

1. 不再避开此问题……

好，到目前为止我们一直在试图回避如何处理大型数据集这个问题，但现在我们决定确实需要解决它，下面来看一下iBATIS提供了什么方式来解决大型数据集问题：RowHandler接口就是用来解决这个问题的。

RowHandler接口是一个非常简单的接口，它允许你在某个已映射语句的结果集的处理中插入自己的动作。此接口只有一个方法：

```
public interface RowHandler {  
    void handleRow(Object valueObject);  
}
```

对于已映射语句的返回的结果集中的每一条记录，iBATIS都会调用一次handleRow方法。使用这个接口，就可以处理大型数据集，而不需要一次性地将它们全部加载到内存中。一次只把一条记录加载到内存中，之后调用代码，废弃这个对象，此过程将不断重复直到结果集中所有记录都处理完毕。

如果需要的话，RowHandler对象可以帮助加速处理大型数据集。这是处理大型数据集问题的最后一根“救命稻草”，但它同时也是iBATIS的“瑞士军刀”。几乎可以用RowHandler来做任何事。

在6.1.2节中，我们介绍了iBATIS中XML结果生成的功能，但发现它在某些方面是不足的——尤其是当你想要为对象列表或者一个复杂对象创建一份（而不是多份）XML文档时。在6.1.2节中，我们曾说过要告诉你如何使用更少的内存来获取XML数据，而不是通过加载整个列表或者对象图然后循环处理它。使用RowHandler，我们仍然需要循环处理这些对象，但这时只需要把列表中的一个元素加载到内存中即可。以下例子给出了一个行处理器，它构建了一份同时包含多个<account>元素的XML文档。

```
public class AccountXmlRowHandler implements RowHandler {  
    private StringBuffer xmlDocument = new StringBuffer("<AccountList>");  
    private String returnValue = null;  
  
    public void handleRow(Object valueObject) {  
        Account account = (Account) valueObject;  
        xmlDocument.append("<account>");  
  
        xmlDocument.append("<accountId>");
```

```
xmlDocument.append(account.getId());
xmlDocument.append("</accountId>");

xmlDocument.append("<username>");
xmlDocument.append(account.getUsername());
xmlDocument.append("</username>");

xmlDocument.append("<password>");
xmlDocument.append(account.getPassword());
xmlDocument.append("</password>");

xmlDocument.append("</account>");
}

public String getAccountListXml(){
    if (null == returnValue){
        xmlDocument.append("</AccountList>");
        returnValue = xmlDocument.toString();
    }
    return returnValue;
}
```

使用这个行处理器的代码非常简单，只需要一个能返回Account对象列表的已映射语句（之前的代码中它就已经存在了），创建一个RowHandler的实例，然后调用queryWithRowHandler方法，将要运行已映射语句、已映射语句所需的参数以及行处理器实例都传递给该方法即可。以下示例利用某个已映射语句返回的所有XML格式的账户创建了一份XML文档：

```
AccountXmlRowHandler rh = new AccountXmlRowHandler();
sqlMapClient.queryWithRowHandler("Account.getAll", null, rh);
String xmlData = rh.getAccountListXml();
```

如果以上这个关于XML的例子还不足以向你展现行处理器的强大能力，那么也许下面的示例可以帮助你了解行处理器是多么有用。

2. 有关RowHandler的另一个更有趣的示例

另一个说明如何使用行处理器的示例是，如何从多个角度处理多表关系。例如，在我们的样例数据库中，我们有账户（或者说顾客），账户可以拥有多个订单，而订单又可以拥有多个订单项，每个订单项都含一种产品，每种产品都有相应的制造商。图6-1给出了这个关系的数据模型。

想象一个这样的需求：我们需要提供一份被定购的产品的列表，以及一份定购这些产品的账户列表。我们还需要一份账户列表，列表中的每一个账户关联一份它定购的产品的制造商列表。如果能有有一个Map对象，利用它我们就能够快速根据ID查找到对应的账户或者产品，那就更好了。

虽然可以使用之前的groupBy属性和queryForMap方法以及4条已映射语句达到目的，但这种方法需要执行4条独立的查询语句（意味着更多的数据库I/O），并且会潜在地返回同一对象的多个副本——第一条已映射语句返回的顾客对象和第二条已映射语句（意味着更多的内存消耗）

返回的那些就可能不是同一个对象。其实使用行处理器可以做得更好！

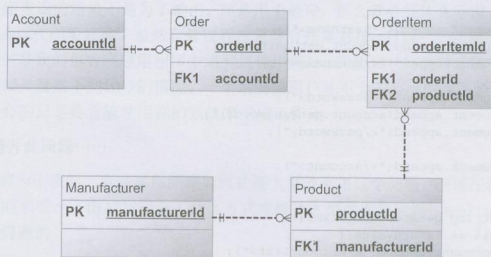


图6-1 示例的实体关系图 (ER图)

使用RowHandler，可以仅用一条SQL语句就满足以上的需求。在这种方法中，我们仅仅需要处理一次结果集就可以同时获得主列表及其相关列表、Map，以及没有重复对象的其他所有东西。使用这种方式，你需要多写一些代码，但可以节省许多处理时间、数据库I/O以及内存空间。

要实现以上方法，我们首先要将相关数据联结起来，然后查看返回的每一条记录，将新的项添加到相应的列表（产品列表或账户列表）和map中。听起来很简单，看看代码吧。

首先，创建一条已映射语句。它会联结数据库表，获取相关的数据，然后将它映射到一个复合的3个不同对象中去，该复合对象包含我们所关心的所有3件事情。

```
<resultMap id="AmprHExample"
class="org.apache.mapper2.examples.chapter6.AccountManufacturerProduct">
  <result property="account.accountId" column="accountId" />
  <result property="manufacturer.manufacturerId"
column="manufacturerId" />
  <result property="product.productId" column="productId" />
</resultMap>
```

```
<select id="AMPRowHandlerExample" resultMap="AmprHExample">
  select distinct
    p.productId as productId,
    o.accountId as accountId,
    m.manufacturerId as manufacturerId
  from product p
  join manufacturer m
    on p.manufacturerId = m.manufacturerId
  join orderitem oi
    on oi.productId = p.productId
  join orders o
    on oi.orderId = o.orderId
  order by 1,2,3
</select>
```


AccountManufacturerProduct类要说也不算复杂，它包含3个特性：account、manufacturer和product。结果映射会自动地为这些字段赋值，将像我们手动创建数据的一份简单视图时所要做的“工作”一样。

然后，行处理器处理返回的每一个AccountManufacturerProduct对象，将它们根据productId、accountId和manufacturerId分类并装入相应map。当它第一次遇到某个特定账户或产品时（根据ID），就会把它们相应地添加到账户列表或产品列表中；否则，以map中的既有对象代替当前从数据库中刚刚加载的那个对象。

最后，当得到了特定账户对象、制造商对象或产品的单个实例后，行处理器会将它们添加到某些合适的对象^①中去，如代码清单6-3所示。

代码清单6-3 一个非常强大的行处理器

```
public class AMPRowHandler implements RowHandler {
    private Map<Integer, AccountManufacturers> accountMap
        = new HashMap<Integer, AccountManufacturers>();
    private Map<Integer, Manufacturer> manufacturerMap
        = new HashMap<Integer, Manufacturer>();
    private Map<Integer, ProductAccounts> productMap
        = new HashMap<Integer, ProductAccounts>();
    private List<ProductAccounts> productAccountList
        = new ArrayList<ProductAccounts>();
    private List<AccountManufacturers> accountManufacturerList
        = new ArrayList<AccountManufacturers>();

    public void handleRow(Object valueObject) {
        AccountManufacturerProduct amp;
        amp = (AccountManufacturerProduct) valueObject;
        Account currentAccount = amp.getAccount();
        Manufacturer currentMfgr = amp.getManufacturer();
        AccountManufacturers am;
        ProductAccounts pa;
        Product currentProduct = amp.getProduct();
        if (null == accountMap.get(currentAccount.getId())) {
            // this is the first time we have seen this account
            am = new AccountManufacturers();
            am.setAccount(currentAccount);
            accountMap.put(currentAccount.getId(), am);
            accountManufacturerList.add(am);
        } else {
            // Use the account from the account map
            am = accountMap.get(currentAccount.getId());
            currentAccount = am.getAccount();
        }
        // am is now the current account / manufacturer list
        if (null ==
```

① 只包含必需的方法

② 验证重复账户

① 如AccountManufacturers对象或ProductAccounts对象。——译者注

```
        manufacturerMap.get(currentMfgr.getManufacturerId())) {  
            // we have not seen this manufacturer yet  
            manufacturerMap.put(  
                currentMfgr.getManufacturerId(),  
                currentMfgr);  
        }  
        else {  
            // we already have this manufacturer loaded, reuse it  
            currentMfgr = manufacturerMap.get(  
                currentMfgr.getManufacturerId());  
        }  
        am.getManufacturerList().add(currentMfgr);  
  
        if (null == productMap.get(currentProduct.getProductId())) {  
            // this is a new product  
            pa = new ProductAccounts();  
            pa.setProduct(currentProduct);  
            productMap.put(currentProduct.getProductId(), pa);  
            productAccountList.add(pa);  
        }  
        else {  
            // this product has been loaded already  
            pa = productMap.get(currentProduct.getProductId());  
        }  
        // pa is now the current product's product / account list  
        pa.getAccountList().add(currentAccount);  
        am.getManufacturerList().add(currentMfgr);  
    }  
  
    public List<ProductAccounts> getProductAccountList() {  
        return productAccountList;  
    }  
  
    public List<AccountManufacturers> getAccountManufacturerList() {  
        return accountManufacturerList;  
    }  
  
    public Map<Integer, ProductAccounts> getProductMap() {  
        return productMap;  
    }  
  
    public Map<Integer, AccountManufacturers> getAccountMap() {  
        return accountMap;  
    }  
}
```

验证重复
制造商 ③

验证重
复产品 ④

获得ProductAccount
bean列表 ⑤

获得AccountManufacturer
bean列表 ⑥

获得产品map ⑦

获得账户map ⑧

虽然此处的代码看起来非常复杂，但其原因是它要完成的工作同样十分复杂。我们从接口需要实现的唯一方法开始①，在此方法中我们处理查询返回的数据，构建包含没有重复的账户对象②、制造商对象③以及产品对象④的列表和map，然后在行处理器中通过获取方法⑤、⑥、⑦和⑧将这些列表和map暴露给调用方。另外，我们提供了对象标识，因此这里的账户、产品和制造商实例都是相同的对象。也就是说，ID值为1的账户在所有的数据结构中都是这个账户类的相同对象——这样，对该对象的任何修改都会在引用它的任何地方反映出来。

6.5 小结

在本章中，你学习了如何将XML数据映射到iBatis中，以及如何使用iBatis在你的结果中生成XML数据。我们还讨论了使用多条SQL语句关联多个数据库表，或者使用一条语句将所有相关数据联系起来。

从第4章～第6章，你已经大致了解了在映射SQL语句时iBatis几乎所能做的所有事情——从简单的到特殊的。在下一章中，你将学习如何在一个更加面向事务的环境中使用iBatis。

第7章

事务

7

本章内容

- 事务介绍
- 自动事务、局部事务和全局事务
- 自定义事务
- 事务划界（demarcation）

事务是使用关系数据库时必须理解的最重要的概念之一。很少有其他的决定能够比事务对系统的稳定性、性能和数据完整性造成更大的影响了。知道如何在你要构建的系统中识别出事务并为其划定边界，对于一个开发人员来说是非常重要的。本章将讨论事务究竟是什么，以及如何处理它们。

7.1 事务是什么

用最简单的话来说，事务就是一项通常包含若干步骤的工作单元，这些步骤必须作为整体来执行，无论成功还是失败。也就是说，如果一个事务中的任何一个步骤失败了，那么所有其他步骤就必须回滚^①，以保证数据仍处于一致的状态。下面我们再用一个经典的示例来解释一下事务。

7.1.1 一个简单的银行转账示例

一个解释事务重要性的最常见的示例就是银行转账业务。考虑两个银行账户，分别由Alice和Bob拥有，初始余额（balance）如表7-1所示。

表7-1 初始余额

Alice的账户		Bob的账户	
余额	\$5 000.00	余额	\$10 000.00

下面假设从Alice的账户转了\$1 000.00到Bob的账户（见表7-2）。

^① 即撤销其操作产生的影响。——译者注

表7-2 期望的结果

Alice的账户		Bob的账户	
余额	\$5 000.00	余额	\$10 000.00
提款	\$1 000.00		
		存款	\$1 000.00
余额	\$4 000.00	余额	\$11 000.00

对Alice账户的提款（withdrawal）操作必须和对Bob账户的存款（deposit）操作在同一个事务中完成。否则，如果存款操作不成功，那么Alice账户的余额将是\$4 000.00，而Bob账户的余额仍然只是\$10 000.00（见表7-3）。转账的\$1 000.00将会凭空消失：

表7-3 存款失败

Alice的账户		Bob的账户	
余额	\$5 000.00	余额	\$10 000.00
提款	\$1 000.00		
		存款（失败）	\$1-000.00
余额	\$4 000.00	余额	\$10 000.00

如果将以上操作定义为一个事务，那么就可以保证即使存款失败，提款操作也可以回滚。提款操作回滚后，数据将仍然停留在事务开始之前的状态，这样就避免了\$1 000.00的凭空消失（见表7-4）。

表7-4 存款失败

Alice的账户		TX	Bob的账户	
余额	\$5 000.00		余额	\$10 000.00
提款（回滚）	\$1-000.00	TX1		
		TX1	存款（失败）	\$1-000.00
余额	\$5 000.00		余额	\$10 000.00

这是一个非常简单的示例。正如你可以想象的，真实世界中银行系统的转账业务会比这个例子复杂得多。基于这个原因，事务被分为许多类型以保证它可以覆盖真实世界中的各种情况，从而解决真实世界中的各种问题。

事务可以非常简短而基本，可能仅仅由几条SQL语句组成，这些语句只改变一个数据库中一张表上的数据；然而，事务也可以非常庞大而复杂，例如一个商务领域的事务（business-to-business transaction）甚至可能超出计算机领域，而涉及人机交互（例如需要某人的数字签名）。其实仅就事务而言，写一本书详细地讨论它根本不足为奇，因此本书将仅仅讨论iBATIS支持的4种范围的事务。

- 自动（automatic）事务——针对那些简单的仅由一条语句构成的事务，这种事务不需要显式地划定事务边界。

- 局部 (local) 事务——指那种简单的范围较窄的事务，这种事务虽然的确涉及很多语句，但只作用于一个数据库。
- 全局 (global) 事务——指那种复杂的范围较广的事务，这种事务往往涉及许多语句和不止一个数据库，甚至有可能涉及潜在的其他具有事务能力的资源，比如JMS (Java Messaging Service) 队列或JCA (J2EE Connector Architecture) 连接等。
- 定制 (custom) 事务——iBATIS支持由用户提供的连接，这样用户就可以随心所欲地管理事务。

对于以上这4种范围的事务，本章将用专门的篇幅分别讨论。在深入探讨这4种事务之前，我们首先谈一谈事务的几个特性。

7.1.2 理解事务的特性

对某个系统来说，如果它宣称自己能够处理事务，那么它就必须具备一些特性或者说特征。几乎所有的现代关系数据库都支持事务，那些不支持事务的数据库是不应该作为企业级应用程序的解决方案的。支持事务的数据库需要具备的特性通常被称为ACID，即所谓的原子性 (atomicity)、一致性 (consistency)、隔离性 (isolation) 和持久性 (durability)

1. 原子性

原子性用于保证事务中的所有步骤作为一个整体，或者全部成功，或者完全失败。没有原子性，如果事务中的某个步骤失败，数据库就有可能处于一种不正确的状态。我们用一个数学加法的例子来简单说明一下这个问题。我们将几个数字相加，并认为这是一个事务，其中的每次加运算都是事务中的一个步骤（见表7-5）。

表7-5 期望的事务

事务 状态	运 算
初始状态	10
第一步	+30
第二步	+45
第三步	+15
最终状态	=100

好，现在假设其中的某一次加运算失败了，比如说第三步（见表7-6）

表7-6 不具备原子性的事务

事务 状态	运 算
初始状态	10
第一步	+30
第二步	+45
失败的第三步	+15
最终状态	=85

因为不具备原子性，当事务中的某一步失败时，数据将处于一种不正确的状态（见表7-6）。

表7-7 具备原子性的事务

事务状态	运 算
初始状态	10
第一步	+30
第二步	+45
失败的第三步	+15
回滚到最初状态	=10

对于一个保证原子性的系统，如果事务中的某一步失败，则所有的操作都不会被执行，这样数据库中的数据就不会受到任何影响（见表7-7）。

2. 一致性

好的数据库模式往往会定义很多约束（constraint）以保证数据完整性（integrity）和数据一致性。被称为一致性的数据库特征要求数据库在事务开始前和结束后都处于一种一致状态。所谓“一致状态”是指所有的约束（包括完整性约束、外键约束和唯一性约束）都得到满足的状态。

3. 隔离性

数据库往往是一种集中共享的资源，会同时被许多用户使用。到底有多少用户并不重要，重要的是，如果你有多个用户，应该如何保证他们各自的事务不会相互冲突。ACID中用于预防这种冲突的特征即被称为隔离性。再考虑一下之前的那个数学加法的例子。如果在计算过程中的某个时刻有另一个用户要求获取计算的值，会发生什么事情呢？（见表7-8）

表7-8 隔离性的事务

用 户 1	运 算	用 户 2
初始状态	10	
第一步	+30	
第二步	+45	
中间状态	=85	<< 要求读取数据
第三步	+15	
最终状态	=100	

其实这个问题并没有一个明确的答案，因为数据库可能会支持各种不同的隔离级别（isolation level）。隔离级别的选择往往需要与性能作权衡。要想获得较高的隔离级别，就要以牺牲较多的性能（尤其是并发访问的性能）为代价。具体的隔离级别如下：

- 读未提交数据（read uncommitted）——这是最低的隔离级别，或者说其实根本就没有任何隔离。在此隔离级别下，可以随意地从数据库中读取数据，即使它是一个未完成事务的结果。因此，对于我们的例子来说，在此隔离级别下你的读数据请求会返回结果85。

- 读已提交数据 (read committed) —— 这个级别的隔离能够保证不返回^①未提交的数据。但尽管如此，如果在事务的进行中，其他用户要访问你正在访问的数据并修改它，你也完全不能阻拦它。这也就是说，这个级别的隔离不能保证一个事务在开始时和结束时读取的数据一定是相同的。
- 可重复读 (repeatable read) —— 这个级别的隔离除了保证只读取已提交数据外，还会获取数据库表中被请求记录的只读锁 (read-lock) 以保证它们在当前事务提交前不会被其他用户修改。但尽管如此，这个级别的隔离保护完全取决于所使用的查询 (query) 的类型。如果请求中包含一个范围子句 (range clause，例如 BETWEEN)，那么该级别的隔离并不会去获取范围锁 (range lock)，这样数据就仍然可能被别的用户通过范围查询获得并修改，最终导致同一事务中对相同数据的前后两次范围查询所获取的数据并不相同 [这种现象通常被称为幻读 (phantom read)]。
- 串行化 (serializable) —— 这可能是获得的最高级别的隔离了。本质上来说其实它让所有的事务都串行执行，一个接一个，因此完全不可能存在冲突。很显然，这个级别的隔离对高并发访问系统会造成巨大的性能损失，因为最终所有的人都必须排队站好一个接一个地完成自己的工作。

4. 持久性

数据库如果不能将其储存的数据持久化，就犹如一座大桥无法通行车辆一样，没有任何用处。ACID 中的持久性特征要求，一旦数据库报告其某个事务成功结束，就意味着事务的执行结果已经安全，即使事务结束之后发生了系统错误，这些数据仍然是安全的。

现在你应该已经熟悉了事务的基础以及那些使事务之所以成为事务的特性，下面我们讨论如何使用 iBATIS 中各种类型的事务。

7.2 自动事务

与 JDBC 还有所谓的“自动提交”模式^②不同，iBATIS 处理的只有事务，它根本就没有在一个事务的范围之外工作的概念，因此它对“自动提交”的支持其实完全是间接的。作为“自动提交”的替代，iBATIS 支持所谓的自动事务。自动事务允许使用单个的方法调用来运行单个的更新语句或查询语句，而完全不用担心如何划分事务，iBATIS 自动将 API 调用所涉及的 SQL 语句划定为一个事务。该语句仍然会在事务中运行，但是你不需要显式地开始 (start)、提交 (commit) 或者结束 (end) 它。

当使用自动事务时，运行一条语句也没有任何需要特别注意的地方——调用相应的 iBATIS API 执行它就可以了。使用自动事务时的配置与使用局部事务（我们将在下一节讨论）时没有任何区别。代码清单 7-1 给出了一个如何使用自动事务的例子。注意其中每条语句都是一个独立的

① 即读取。——译者注

② 即将每条 SQL 语句都看作一个事务，执行完毕后立即提交。——译者注

事务，具体来说就是对queryForObject()和update()两个API的调用。

代码清单7-1 自动事务

```
public void runStatementsUsingAutomaticTransactions()
{
    SqlMapClient sqlMapClient =
        SqlMapClientConfig.getSqlMapClient();
    Person p = (Person)
        sqlMapClient.queryForObject("getPerson",
                                    new Integer(9));
    p.setLastName("Smith");
    sqlMapClient.update("updatePerson", p);
}
```

看完代码你可能会问，事务在哪里？呵呵，既然是自动事务，一切当然都是隐式地自动完成的，但是事务的确存在！在很多情况下，自动事务就已经足够了，但不排除有某些情况需要进行更加细粒度的控制，在这些情况下，你可能就需要使用更加显式的局部事务或全局事务了。

7.3 局部事务

局部事务是最常用的事务类型，在任何涉及关系数据库的项目中，你至少都需要使用这种事务。即使是在上一节中讨论的自动事务，其实也不过是局部事务的一种较简洁的形式罢了。所谓局部事务，它仅仅涉及一个应用程序、一种资源（例如关系数据库），并且一次只能处理一个事务。如图7-1所示。

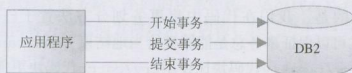


图7-1 局部事务范围

局部事务在iBATIS的SQL Map配置文件（XML格式）中被配置为一个JDBC类型的事务管理器。代码清单7-2显示了事务管理器配置可能的读取方式。

代码清单7-2 局部事务管理器的配置

```
<transactionManager type="JDBC">
  <dataSource type="SIMPLE">
    <property .../>
    <property .../>
    <property .../>
  </dataSource>
</transactionManager>
```

type="JDBC" 属性告诉iBATIS使用标准的JDBC Connection API来管理事务。使用

SqlMapClient API来划分事务其实非常简单。代码清单7-3展示了一种使用iBATIS进行事务划分的典型模式。

代码清单7-3 局部事务

```
public void runStatementsUsingLocalTransactions() {
    SqlMapClient sqlMapClient =
        SqlMapClientConfig.getSqlMapClient();
    try {
        sqlMapClient.startTransaction();
        Person p =
            (Person)sqlMapClient.queryForObject
                ("getPerson", new Integer(9));
        p.setLastName("Smith");
        sqlMapClient.update("updatePerson", p);

        Department d =
            (Department)sqlMapClient.queryForObject
                ("getDept", new Integer(3));
        p.setDepartment(d);
        sqlMapClient.update("updatePersonDept", p);
        sqlMapClient.commitTransaction();
    } finally {
        sqlMapClient.endTransaction();
    }
}
```

代码清单7-3中的两个更新语句将在同一个事务中运行，因此如果任何一个更新失败，则两个更新同时失败。

这里需要重点指出的是，我们在代码中将事务划分方法置于try/finally中，而不是try/catch中。这种模式可以保证事务总是能够被恰当地结束（endTransaction），即使在发生了错误的情况下。使用try/finally代码块比使用try/catch代码块更加简单和高效，因为它不需要你去捕捉那些其实你根本不会（也无法）去处理的异常。

7.4 全局事务

全局事务定义了一个比局部事务所定义的事务范围大得多的事务范围。它可以包括其他的数据库，包括消息队列，甚至包括其他的应用程序。图7-2展示了一个使用全局事务的系统，从图中可以清楚地看出全局事务会使系统变得多么的复杂。

幸运的是，只要使用iBATIS，全局事务就不会比局部事务复杂。但是，你还必须明白，就像做其他选择一样，只有通过反复试验，才能找到最佳答案。

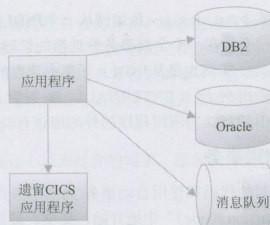


图7-2 全局事务范围示例

7.4.1 使用主动或被动事务

iBATIS可以用以下两种方式之一来参与全局事务：主动参与或被动参与。当配置为主动参与时，iBATIS会查找全局事务上下文，然后试图以某种恰当的方式管理它。这就是说，iBATIS会查看现存事务的状态并且在必要的时候主动开始一个事务。它也可以在发生错误时将事务的状态设置为“rollback-only”，这样就可以通知参与此事务的所有其他资源当前事务不应被提交。

当配置为被动参与一个全局事务时，iBATIS会忽略当前应用程序中所有关于开始事务、提交事务以及结束事务的指令代码。它在出现错误时不是主动回滚而是抛出异常，因为iBATIS假设抛出异常就可以引起事务的回滚。

到底使用主动方式还是被动方式参与全局事务其实并没有什么标准，有时候只能通过反复试验来确定。对于不同的应用程序服务器、不同的体系结构，这两种方式各有千秋。而使用iBATIS的好处就在于，可以在这两种方式之间方便地来回切换。代码清单7-4展示了如何将全局事务配置为主动参与和被动参与。

代码清单7-4 全局事务管理器配置选项

```
<transactionManager type="JTA">
  <property name="UserTransaction"
    value="java:ctx/oon/someUserTransaction"/>
  <dataSource type="JNDI">
    <property name="DataSource"
      value="java:comp/env/jdbc/someDataSource"/>
  </dataSource>
</transactionManager>

<transactionManager type="EXTERNAL">
  <dataSource type="JNDI">
    <property name="DataSource"
      value="java:comp/env/jdbc/someDataSource"/>
  </dataSource>
</transactionManager>
```

主动参与

被动参与

被动参与

注意以上配置中在两种方式下DataSource是如何从一个JNDI上下文中获得的。这是一个很实际的需求，因为你需要的连接必须在一个全局事务管理器的管理范围之内。在使用JTA事务管理器的情况下，UserTransaction实例也是从JNDI上下文中获得的，这样你的应用程序就可以主动参与全局事务了。在使用外部（EXTERNAL）事务管理器的情况下，不再需要UserTransaction实例，因为iBATIS认为应用程序的外部应该有其他的系统负责管理事务。

7.4.2 开始、提交以及结束事务

使用全局事务时，需要编写的代码和使用自动事务/局部事务几乎是一样的。同样需要在一个“内部事务范围（inner transaction scope）”中地开始、提交、最后结束事务。你可能会想，既然是全局定义的事务，为什么仍然与其他普通事务一样呢？有两个理由。第一，这可以帮助iBATIS管理像数据库连接这样的资源，这样你就再不需要亲自编写代码去向数据源不断地请求连接和返回连接了。第二，这种方式允许你在不用改变任何代码的情况下在全局事务和局部事务之间来回切换。阅读代码清单7-5所示的使用全局事务的代码你就会发现，它与之前的局部事务示例没有任何区别，这样不论使用的是全局事务还是局部事务，对iBATIS来说都没有任何区别。

代码清单7-5 全局事务

```
public void runStatementsUsingGlobalTransactions() {
    SqlMapClient sqlMapClient =
        SqlMapClientConfig.getSqlMapClient();
    try {
        sqlMapClient.startTransaction();
        Person p =
            (Person)sqlMapClient.queryForObject
                ("getPerson", new Integer(9));
        p.setLastName("Smith");
        sqlMapClient.update("updatePerson", p);

        Department d =
            (Department)sqlMapClient.queryForObject
                ("getDept", new Integer(3));
        p.setDepartment(d);
        sqlMapClient.update("updatePersonDept", p);
        sqlMapClient.commitTransaction();
    } finally {
        sqlMapClient.endTransaction();
    }
}
```

好的，到目前为止你已经知道如何实现局部事务和全局事务了，那么应该如何决定到底使用哪一种事务呢？继续阅读看看是否能帮助做出决定。

7.4.3 我是否需要全局事务

对于这个问题，答案是：“可能不。”在大多数情况下，使用全局事务给应用程序带来更多的额外负担。其原因主要是因为全局事务往往是分布的，这就需要比局部事务更多的网络带宽和状

态管理。如果网络带宽和状态管理没有任何开销，那么也许对所有的应用程序我们都会使用全局事务。除了性能损失，全局事务存在的问题还包括它更加难以建立，它需要更多的基础设施、更多的软件 and 更多的资源。所以即使你使用容器管理的事务，也只有当完全确定你确实需要全局事务时才使用它。大多数好的应用程序服务器都能够通过非常简单的配置选项切换来启用（enable）或者禁用（disable）分布事务^①。

如果以上这些配置选项都还不能满足你的需求，那么下面就看看还有什么其他可用的选项吧。

7.5 定制事务

正如你已经看到的，iBATIS支持以多种方式管理事务。如果以上的这些事务管理机制都还不能满足你的要求，那不妨自己管理事务吧，iBATIS为你提供了许多选项可以达到这个目的。第一种方法就是使用iBATIS的某些接口，自己写一个事务管理器，然后把它插入到SQL Map配置文件中。这种方法会在第12章中详细讨论。第二种方式就是由你为iBATIS传递一个要使用JDBC Connection实例，从而允许你拥有对连接和事务的完全控制权。给iBATIS的SqlMapClient实例传递一个Connection实例的方法有两种，第一种就是使用SqlMapClient类的setUserConnection(Connection)方法，如代码清单7-6所示。

代码清单7-6 使用setUserTransaction()实现定制事务控制

```
public void runStatementsUsingSetUserConnection() {
    SqlMapClient sqlMapClient =
        SqlMapClientConfig.getSqlMapClient();
    Connection conn = null;
    try {
        conn = dataSource.getConnection();
        conn.setAutoCommit(false);
        sqlMapClient.setUserConnection(conn);
        Person p =
            (Person)sqlMapClient.queryForObject(
                "getPerson", new Integer(9));
        p.setLastName("Smith");
        sqlMapClient.update("updatePerson", p);

        Department d =
            (Department)sqlMapClient.queryForObject(
                "getDept", new Integer(3));
        p.setDepartment(d);
        sqlMapClient.update("updatePersonDept", p);
        conn.commit();
    } finally {
        sqlMapClient.setUserConnection(null);
        if (conn != null) conn.close();
    }
}
```

^① 即全局事务。——译者注

第二种方式就是使用SqlMapClient类的openSession (Connection)方法。这是我们推荐的方法，因为使用这种方式iBATIS能够更好地管理资源。代码清单7-7展示了如何通过openSession()方法实现定制事务控制。

代码清单7-7 用openSession()方法实现定制事务控制

```
public void runStatementsUsingSetUserConnection() {
    SqlMapClient sqlMapClient =
        SqlMapClientConfig.getSqlMapClient();
    Connection conn = null;
    SqlMapSession session = null;
    try {
        conn = dataSource.getConnection();
        conn.setAutoCommit(false);
        session = sqlMapClient.openSession(conn);
        Person p =
            (Person)session.queryForObject("getPerson",
                                           new Integer(9));
        p.setLastName("Smith");
        session.update("updatePerson", p);

        Department d =
            (Department)session.queryForObject
                ("getDept", new Integer(3));
        p.setDepartment(d);
        session.update("updatePersonDept", p);
        conn.commit();
    } finally {
        if (session != null) session.close();
        if (conn != null) conn.close();
    }
}
```

使用这种方式，代码看上去的确不那么优雅，这也是为什么有时你需要写一个自己的事务管理器的原因所在。此外，我们仍然需要定义一个EXTERNAL类型的事务管理器，并至少提供一个SIMPLE类型的DataSource；否则，像延迟加载这样的特征就无法正常工作。

尽管相比于第一种方式，我们推荐使用这种方式，但如果可能，你最好还是连这种方式也尽量避免使用，而考虑写一个自己的事务管理器。

7.6 事务划界

到目前为止，我们已经告诉了你很多种开始和结束事务的方式了，那么你可能会问，“到底应该在哪里开始和结束事务呢？”通常来说，回答一个有关where的问题要比回答一个有关how的问题更加困难。就像许多其他难以回答的问题一样，这个问题的答案是：“要具体问题具体分析。”所谓“仁者见仁，智者见智”，去问不同的人，就可以得到不同的答案。

在何处划分事务将直接决定这个事务处于“打开”状态的时间会有多长，也就决定了在一个

事务的范围内到底包含多少工作。明白了这些，相信你也就不难理解一个事务的范围如果过宽会
给性能造成多大的影响了。但话说回来，大范围的事务又的确更加安全，因为它保证了一组可能
相关的工作（这些工作如果不相关，那它们为什么会在对同一个请求的响应中发生呢）的完整性。
因为要总结一条放之四海而皆准的规则的确很困难，所以此处我们只给出一条简单的经验性规则：
因此：

事务的范围越宽越好,但绝不能超过一次用户动作的范围。

举个例子，在一个Web应用程序中，当用户单击某个按钮提交一份表单时，相应的事务范围就应该马上开始，但是当响应页面呈现在用户浏览器中时，这个事务就应该结束了。在一个富客户端应用程序中，以上规则同样适用。一个事务应该包含一次用户操作（通常表现为按钮的再次单击）中的所有工作。也可以这样考虑这个问题：用户一旦离开电脑去做别的事，事务就不应再处于打开状态，或者未完成状态。

那么，到底应该在何处开始和结束事务呢？理想情况下，事务根本就不应该由开发人员来启动。换句话说，应该让容器来启动事务。因此，你应该声明式地配置应用程序，这样事务的划分就可以由容器代劳了。不论是在使用一个支持无状态会话bean的应用程序服务器，还是一个像Spring Framework这样的轻量级容器，都可以通过声明文件来配置事务。容器可以保证事务正确地启动、提交和结束。iBATIS为其支持的各种范围的事务使用了一个统一的编程模型，因此即使容器支持事务划分，你也可以也应该在代码中始终使用 `startTransaction()`、`commitTransaction()` 和 `endTransaction()`。iBATIS所以使用这种方式的原因在于它使得你的持久化代码可以不依赖于任何容器，这样即使在没有容器的环境下，代码也可以表现出相对一致的行为。其实当事务管理器的类型被配置为EXTERNAL时，iBATIS会将事务的管理权完全交给外部的容器。

```

graph TD
    A[表现层] <--> B[业务逻辑层]
    B --> C[业务对象模型]
  
```

如果不允许由容器来管理事务,那么也可以亲自管理它们。在分层体系结构中,究竟在哪里开始和结束事务,可以有多种选择。但不论做出什么选择,都要始终如一、坚持到底,这才是最重要的。在如图7-3所示的分层体系结构中,将事务在表现层、业务逻辑层、持久层划界似乎都是可以的。

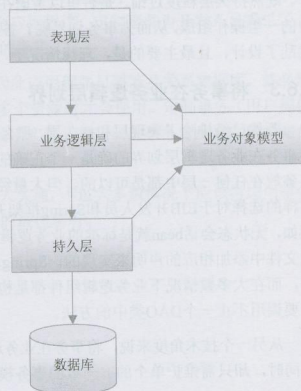


图7-3 分层体系结构

让我们仔细研究一下如何将事务在这三层划界。

7.6.1 将事务在表现层划界

将事务在表现层划界会使得一个事务变得非常巨大，范围极宽，持续时间极长。然而，这样的事务非常安全，它们提供了最好的数据完整性约束，并且最容易实现。可以为你的表现层框架使用一个servlet（过滤器）或一个插件程序来开始、提交和结束事务。这种方式的问题就是，性能将因为事务持续时间过长而受到较大的影响。表现层通常并不确切知道为何要开始一个事务，因为它离持久层太远了。因此，许多事务往往不必要地持续了过长的时间才结束。另一个问题就是，如果你的应用程序存在多个用户界面，比如说网站、Web服务API或者富客户端，那么就要分别为这些用户界面重新实现事务范围，这就可能造成多个用户界面在事务上的表现不一致。

既然表现层与持久层太远，因此不是放置事务的合理位置，那么我们何不直接将事务放到持久层中去呢？

7.6.2 将事务在持久层划界

很多人可能想当然地认为将事务在持久层划界是最为自然的。但是，其实这是一个非常糟糕的选择。原因就在于一个设计良好的持久层通常都是由作用范围狭小、松散耦合、高内聚的数据库操作组成的。实际上很少有人会在一个事务中仅仅使用这些操作中的单个操作，因为这样做将使事务毫无用处。大多数情况下，要完成一项有意义的业务功能，都需要执行持久层中的一组操作。既然持久层粒度过细，是否可以考虑在持久层之上再构建一层，让该层中的方法都由持久层中的一组操作组成，从而为事务划界呢？答案是否定的，因为这种方法徒添了大量不必要的工作、扰乱了设计，且最主要的是，理想情况下，持久层应该是完全透明的。

7.6.3 将事务在业务逻辑层划界

既然已经排除了表现层和持久层，那么现在就只剩下业务逻辑层了。是的，正如你所猜测的，将事务在业务逻辑层划界的确是一个正确的选择。虽然之前所说，根据具体应用程序的需求，将事务放在任何一层中都是可以的。但大量经验表明，将事务在业务逻辑层划界往往是最佳选择。这样的选择对于EJB开发人员和Spring框架开发人员来说也许都是再熟悉不过、再正常不过的了。例如，无状态会话bean就是标准的业务逻辑组件，而应用程序中对于事务的需求都是通过在其配置文件中添加相应的声明来实现的。Spring以类似的方式几乎允许为任何方法（method）声明事务，而在大多数情况下业务逻辑组件都是构建在DAO^①类之上的，因为通常一个业务逻辑操作都需要调用不止一个DAO类中的方法。

从另一个技术角度来说，将事务在业务逻辑层划界可以使你在为应用程序提供多种用户界面^②的同时，却只需维护单个的、一致的事务模型。

那么你现在怎么想呢？将事务在业务逻辑层划界感觉自然吗？我想答案一定是“非常自然”

① 详见第10章。——译者注

② 对比7.6.1节。——译者注

吧。架构师、数据库管理员和开发人员通常都把事务当作一个技术概念，认为事务仅仅与数据库有关，是“数据库要处理的东西”。不过我们应该从一个更高的层次来思考事务，事务范围应该包括业务操作或者业务功能。它绝不仅仅是数据库才具有的特征。其实，数据库决不是支持事务的唯一基础设施元素。我们的业务操作中就有可能包括通过连接器到大型机的调用、发布消息到一个消息队列，甚至可能把人类的介入（想象一下工作流）也作为事务的一部分所有这些可能的确支持事务。

将事务在业务逻辑层划界是一个完美的选择，不论从逻辑上说还是从技术上说都是如此。

7.7 小结

本章讨论了事务，具体包括什么是事务以及如何在iBATIS中使用事务。我们讨论了对于事务管理器来说必不可少的ACID特性。原子性保证了事务中的所有步骤要么全部成功要么一齐失败，总之必须是一个整体，不可分割。一致性保证所有的数据库约束在事务前后都是满足的。隔离性保证并发事务不会相互冲突，造成意想不到的结果。持久性保证事务一旦执行成功，其结果就一定是安全的了。

我们讨论了iBATIS支持的几种范围的事务，包括自动事务、局部事务、全局事务和定制事务。自动事务的范围最窄，仅包括一条语句，但绝对不要怀疑该语句是在一个事务的范围中的。局部事务的范围稍宽，可以包括许多更新语句，但也仅仅涉及一个应用程序、一个数据库。全局事务则要复杂得多，它允许事务跨越多个数据库、资源甚至应用程序。定制事务为开发人员提供了一种全面控制数据库连接进而控制iBATIS所使用的事务的手段。

局部事务，包括自动事务，是范围最窄的事务，因此应用程序只要涉及关系数据库，就必然会使用这两种事务。全局事务则只有当应用程序涉及多个资源时才会使用。定制事务（用户提供连接）的使用则必须更加谨慎，如果可能，尽量自己动手实现一个定制的事务管理器实现来避免使用它们。

选择在应用程序的哪一层划界事务有时候的确让人头痛，不过大多数情况下应该考虑用事务来包裹业务功能，因此，业务逻辑层应该是划界事务的最佳场所。

第 8 章

使用动态SQL

8

本章内容

- 动态SQL介绍
- 动态SQL简单示例
- 高级动态SQL
- 未来的方向

在第4章中我们讨论了如何编写简单的静态SQL。静态SQL只要求通过特性语法（`#...#`）或者文字（literal）语法（`$...$`）来赋值。虽然在iBATIS中你编写的大多数SQL都是静态的，但也有些情况可能不会那么简单。例如，你可能很快就会碰到像这样一个棘手的情况，你需要遍历一个列表，将其中所有的值都填充到一个IN子句中去，以便为用户提供由他们决定显示哪些列的能力^①，或者根据参数对象的不同状态设置不同的WHERE子句作为查询条件。iBATIS提供一组动态SQL标签，可以将它们用在已映射语句中，以提高SQL的可复用性和灵活性。

本章将为你解释动态SQL究竟是什么，并使你理解它的有效性以及何时使用它最佳。同时，我们还将给出一些关于你可能会使用的用于处理动态SQL问题的解决方案的比较背景。最终你将充分理解如何把动态SQL添加到自己的工具库（即解决问题的技术库）中。

在详细讨论动态SQL所使用的标签之前，让我们先用一个需要使用动态SQL的最常见示例来说明它的价值，这个例子就是动态构建的WHERE子句。

8.1 处理动态 WHERE 子句条件

在下面将要给出的这个示例中，我们要从购物车应用程序中查询名为Category的表。该表的parentCategoryId列是一个自引用列。也就是说，parentCategoryId引用同一个Category表中的categoryId，如图8-1所示。

① 详见8.2.5节。——译者注

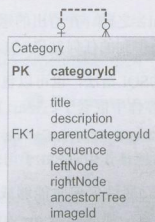


图8-1 Category表示图

需求非常简单。如果传递进来的Category对象的parentCategoryId特性为空，这就意味着我们需要查询所有的顶层类别^①。如果传递进来的Category对象的parentCategoryId特性有一个非空数值，这意味着我们需要查询该parentCategoryId值对应的Category对象的所有子Category对象^②。

在SQL中，等号(=)不能用来比较NULL值是否相等。要比较NULL值的相等性，需要用IS关键字。既然我们想要使用相同的SQL语句来同时处理NULL和非NULL值的比较，就必须使用动态SQL，以在一条已映射语句中来完成此任务。我们将通过对这条已映射语句的分析来剖析动态SQL（见代码清单8-1）。

代码清单8-1 动态WHERE子句示例

```

...
<select id="getChildCategories" parameterClass="Category"
    resultClass="Category">
SELECT *
FROM category
<dynamic prepend="WHERE ">
    <isNull property="parentCategoryId">
        parentCategoryId IS NULL
    </isNull>
    <isNotNull property="parentCategoryId">
        parentCategoryId=#parentCategoryId#
    </isNotNull>
</dynamic>
</select>
...

```

我们对动态SQL的剖析从最外层的父标签开始。一个父标签可以是任意一个动态SQL标签。在此例中，我们使用<dynamic>标签作为父标签。<dynamic>标签不像其他动态SQL标签，它不会去评测（evaluate）任何值或状态。它通常只是使用prepend属性，该属性值将作为前缀被加入到

① 即没有父类别的类别。——译者注

② 即传递进来的Category对象的所有兄弟。——译者注

<dynamic>标签的内容体（body content）之前。在给出的示例中，WHERE值都会作为前缀添加到由嵌套的动态SQL标签经iBATIS处理而得到的任何结果SQL之前。

父标签的内容体中可以包含简单的SQL语法，也可以包含其他的动态SQL标签。在代码清单8-1中可以看到，<dynamic>标签的内容体中嵌套了<isNull>和<isNotNull>两个标签。此处我们所关心的是如何根据参数类Category(parameterClass="Category")的parentCategoryId特性是否为空，来确定正确的SQL是否应该作为WHERE子句条件的一部分。

此处需要重点指出的是，prepend属性值何时能够成为前缀。如果<dynamic>标签的内容体经iBATIS处理后没有产生任何文本，那么prepend值将被忽略。要使prepend属性值真正能够成为前缀，就必须要有可为其添加前缀的结果SQL产生。在我们的例子中，总是会有结果SQL产生的^①。在其他情况下，如果内容体没有产生任何内容，那么prepend属性所取的WHERE值将被忽略。

使用动态SQL的好处之一就是，它可以提高SQL代码的可复用性。假设此例没有使用动态SQL，就必须编写两个SELECT语句来处理这种情况。当然，还可以将对类别对象的parentCategoryId特性的检查放在DAO层中（见第10章），根据parentCategoryId的值是否为空来调用相应的SELECT语句。尽管对这个简单的例子而言，不使用动态SQL也不会给我们带来多大的不便，但是当特性的不同组合会导致SQL语句的可能性成指数增长时，动态SQL的价值就非常明显了。通过使用动态SQL，我们可以提高已映射语句的可复用性，从而避免编写一大堆静态SQL语句。

现在已经充分了解了动态SQL的使用背景和强大功能，接下来让我们继续深入研究动态SQL的所有标签和属性。

8.2 熟悉动态标签

iBATIS提供了一组功能强大的标签以支持动态SQL，这组动态标签可用于评测包裹了被传递到已映射语句的参数对象的各种条件是否成立。了解现有标签的所有范围及其在生成正确的SQL输出中所扮演的各种角色非常重要。下面我们将所有标签分成5类：<dynamic>标签、二元标签、一元标签、参数标签以及<iterate>标签。每一组标签都包含一个或多个共享相同属性的相关标签。在开始研究这些不同的标签组之前，首先看一下为所有的动态SQL标签所共享的一些属性和行为。

所有的动态标签都有prepend、open和close这3个属性。open和close属性在每个标签中的功能是一样的。它们无条件地将其属性值放在标签的结果内容的开始处或者结束处。除了<dynamic>标签之外，prepend属性在所有其他标签中的功能也都是是一样的。<dynamic>标签在内容体的结果SQL非空时，总是将其prepend属性值添加为该结果SQL的前缀。没有什么方式可

① 因为parentCategoryId特性取值如果不是空就必然是非空值，<isNull>和<isNotNull>两者必有一评测结果为真。——译者注

以阻止<dynamic>标签将该值加为前缀^①。代码清单8-2给出了一些动态SQL标签的使用示例。

代码清单8-2 用于释义removeFirstPrepend属性示例

```
...
<dynamic prepend="WHERE "> ① 起始<dynamic>标签
...
  <isNotEmpty property="y"> ② 简单的isNotEmpty标签
    y=#y#
  </isNotEmpty>

  <isNotNull property="x" removeFirstPrepend="true"
    prepend="AND" open="(" close=")"> ③ 更复杂的isNotEmpty标签
    <isNotEmpty property="x.a" prepend="OR">
      a=#x.a#
    </isNotEmpty>

    <isNotEmpty property="x.b" prepend="OR"> ④ 嵌套的动态标签
      a=#x.b#
    </isNotEmpty>

    <isNotEmpty property="x.c" prepend="OR">
      a=#x.c#
    </isNotEmpty>

  </isNotNull>
</dynamic>
...
```

在①处，起始<dynamic>标签具有隐式删除（或者说忽略）其第一个产生内容的子标签上的prepend功能^②。本例中，如果<isNotEmpty>标签被评测为true^②，那么作为第一个产生内容的子标签，其prepend属性值将被忽略^③，而与其同一层的所有其他标签的prepend值将生效，作为前缀添加在相应的位置。<isNotNull>标签^③设定了removeFirstPrepend属性。open和close属性的值将把标签的内容体最终产生的内容包裹起来。在④处，由于父标签设定了removeFirstPrepend属性，第一个产生内容的<isNotEmpty>子标签将不会把它的prepend值OR加为前缀。这样最终究①处就能产生正确的SQL语句，且由于open和close属性，该SQL语句将被包裹在一对括号中。

“移除第一个prepend”（即removeFirstPrepend）这个功能在所有的标签中都得到了显式或者隐式的支持。<dynamic>标签隐式地支持此功能。而所有其他标签则通过removeFirstPrepend属性显式地支持此功能。“删除第一个prepend”功能移除生成内容的子标签的第一个prepend。即使第一个生成内容的子标签并没有指定prepend属性，该子标签仍然会被计算

① 这是相对于其他标签的prepend属性而言的，其他标签的prepend属性值并不一定总会被加为前缀，详情见后。

——译者注

② 即隐式包含一个取值为true的removeFirstPrepend属性。——译者注

③ 此例中该标签没有指定prepend属性。——译者注

在内，这样其后所有的内容生成子标签都将会把各自的prepend属性值作为前缀加到结果内容体中。

需要注意的最后一个共享功能就是，所有标签都可以相互独立使用。这意味着你并不需要将所有的动态SQL标签嵌套在<dynamic>标签中。可以从<iterate>标签开始，然后可以很容易地把<isNull>标签嵌套在其中，就像你可以把这两个标签都封装在<dynamic>标签中一样简单。提供这种功能是因为，只有当你想要向结果内容体使用open、close或prepend值时，才必须使用<dynamic>标签。

下面来逐类分析每一组标签。

8.2.1 <dynamic>标签

<dynamic>标签是最顶层标签；这意味着它不能被嵌套。该标签用来划分一个动态SQL片段。该标签用于提供一种将公共的prepend、open或close值作为前缀添加到其内容体的结果内容中的方式。<dynamic>标签的属性如表8-1所示。

表8-1 <dynamic>标签的属性

prepend（可选的）	该值用于作为前缀添加到标签的结果内容体前。但是当标签的结果内容体为空时，prepend值将不起作用
Open（可选的）	该值用于作为前缀添加到标签的结果内容体前。如果结果内容体为空，open值将不会被附加到其前面。open值将在prepend属性值被添加前缀之前先被添加前缀。例如，假设prepend="WHEN"，而open="("，则最终得到的组合前缀将会是"OR ("
close（可选的）	该值用于作为后缀附加到标签的结果内容体后。如果标签的结果内容体为空，close值将不起作用

现在，当你不明白如何使用<dynamic>标签的各个属性时，就可以参考表8-1了，代码清单8-3说明了如何使用<dynamic>标签。

代码清单8-3 <dynamic>标签使用示例

```
...
<select id="getChildCategories" parameterClass="Category"
      resultClass="Category">
SELECT *
FROM category
<dynamic prepend="WHERE ">
  <isNull property="parentCategoryId">
    parentCategoryId IS NULL
  </isNull>
  <isNotNull property="parentCategoryId">
    parentCategoryId=#parentCategoryId#
  </isNotNull>
</dynamic>
</select>
...
```

代码清单8-3中，我们使用动态SQL为SELECT语句构建了一个动态WHERE子句，该语句查看传入对象的parentCategoryId特性并依此构建SQL。

8.2.2 二元标签

二元标签 (binary tag) 用于将参数特性的某个值同另外一个值或者参数特性做比较。如果比较结果为true，那么结果SQL中就包含其内容体。所有的二元标签都共享compareProperty特性、compareValue属性。property属性用于设置被比较的基本类型值，而compareProperty属性和compareValue属性则用于设置比较的参考值。compareProperty属性会指定参数对象中的一个特性，该字段的取值会作为比较时的参考值。compareValue属性则会指定一个静态值，用于比较基本类型值。各个标签的名称实际上就已经指出了应该如何进行比较这些值。二元标签的属性如表8-2所示。

表8-2 二元标签的属性

property (必需的)	参数对象中用于同compareValue或者compareProperty相比较的特性
prepend (可选的)	该值用于作为前缀附加到标签的结果内容体前。只有在以下3种情况下，prepend值才不会被加为前缀：(a) 当标签的结果内容体为空时；(b) 如果该标签第一个产生内容体，并且它被嵌套在一个removeFirstPrepend属性被设置为true的父标签中时；(c) 如果此标签是跟在prepend属性取值非空的<dynamic>标签后的第一个生成内容体的标签
Open (可选的)	该值用于作为前缀附加到标签的结果内容体前。如果标签的结果内容体为空，open值将不会被附加到其前面。open值将在prepend属性值被添加为前缀之前先被添加前缀。例如，假设prepend="OR"，而open="("，则最终得到的组合前缀将会是"OR ("
close (可选的)	该值用于作为后缀附加到标签的结果内容体后。如果标签的结果内容体为空，则close值将不起作用
removeFirstPrepend (可选的)	该值用于决定第一个嵌套的内容生成标签是否移除其prepend值 (prepend是可选属性)
compareProperty (如果没有指定compareValue，则它是必需的)	该值指定参数对象中的一个特性用来同property属性所指定的特性相比较
compareValue (如果没有指定compareProperty，则它是必需的)	该值指定一个静态比较值用于同property属性所指定的特性相比较

所有的二元动态标签都可以共享表8-2给出的这些属性，而二元标签本身如表8-3所示。

表8-3 iBATIS二元动态标签

<isEqual>	将property属性同compareProperty属性或compareValue属性相比较，确定它们是否相同
<isNotEqual>	将property属性同compareProperty属性或compareValue属性相比较，确定它们是否不同
<isGreaterThan>	确定property属性是否大于compareProperty属性或compareValue属性
<isGreaterEqual>	确定property属性是否大于等于compareProperty属性或compareValue属性
<isLessThan>	确定property属性是否小于compareProperty属性或compareValue属性
<isLessEqual>	确定property属性是否小于等于compareProperty属性或compareValue属性

虽然表格便于参考，但是它们不能给出好的例子，因此代码清单8-4给出了一个使用这些标签的示例。

代码清单8-4 二元标签示例

```
<select id="getShippingType" parameterClass="Cart"
      resultClass="Shipping">
  SELECT * FROM Shipping
  <dynamic prepend="WHERE">
    <isGreaterEqual property="weight" compareValue="100">
      shippingType='FREIGHT'
    </isGreaterEqual>
    <isLessThan property="weight" compareValue="100">
      shippingType='STANDARD'
    </isLessThan>
  </dynamic>
</select>
```

在代码清单8-4中，我们创建了一条SELECT语句，该SELECT语句根据weight特性的值来判定应该使用何种航运类型——小于100时，则使用标准航运；而大于或等于100时，则使用货运。

8.2.3 一元标签

一元标签（unary tag）用于直接考察参数对象中某个bean特性的状态，而不需要与其他值进行比较。如果参数对象的状态结果为真，那么结果SQL中就会包含其内容体。所有的一元标签都共享property属性。一元标签的property属性用于指定参数对象上用来考察状态的特性。标签的名称就可以指出当前要考察的状态类型。一元标签的属性如表8-4所示。

表8-4 一元标签属性

property（必需的）	参数对象中用于状态比较的特性
prepend（可选的）	该值用于作为前缀附加到标签的结果内容体前。只有在以下3种情况下，prepend值才不会被加为前缀：(a)当标签的结果内容体为空时；(b)如果该标签第一个产生内容体，并且它被嵌套在一个removeFirstPrepend属性被设置为true的父标签中时；(c)如果此标签是跟在prepend属性取值非空的<dynamic>标签后的第一个生成内容体的标签
Open（可选的）	该值用于作为前缀附加到标签的结果内容体前。如果结果内容体为空，open值将不会被附加到其前面。open值将在prepend属性值被添加前缀之前先被添加前缀。例如，假设prepend="OR"，而open="("，则最终得到的组合前缀将会是"OR ("
close（可选的）	该值用于作为后缀附加到标签的结果内容体后。如果结果内容体为空，则close值将不起作用
removeFirstPrepend（可选的）	该属性值用于决定第一个产生内容体的嵌套子标签是否移除其prepend值

表8-4中给出的所有属性对于表8-5中所列的一元动态SQL标签都是可用的。

表8-5 一元标签

<isPropertyAvailable>	确定参数对象中是否存在所指定的字段。对于bean，它寻找一个特性；而对于map，它寻找一个键
<isNotPropertyAvailable>	确定参数中是否不存在所指定的字段。对于bean，它寻找一个特性；而对于map，它寻找一个键
<isNull>	确定所指定的字段是否为空。对于bean，它寻找获取方法特性的返回值；而对于map，它寻找一个键，若这个键不存在，则返回true
<isNotNull>	确定所指定的字段是否非空。对于bean，它寻找获取方法特性的返回值；而对于map，它寻找一个键，若这个键不存在，则返回false
<isEmpty>	确定所指定的字段是否为空、空字符串、空集合或者空的String.valueOf()
<isNotEmpty>	确定所指定的字段是否非空、非空字符串、非空集合或者非空的String.valueOf()

代码清单8-5显示了如何使用一元动态SQL标签。

代码清单8-5 一元标签示例

```
...
<select id="getProducts" parameterClass="Product"
      resultClass="Product">
  SELECT * FROM Products
  <dynamic prepend="WHERE ">
    <isNotEmpty property="productType">
      productType=#productType#
    </isNotEmpty>
  </dynamic>
</select>
...
```

在代码清单8-5中，我们创建了一个简单的SELECT已映射语句，然后使用一个动态SQL标签根据productType特性有选择地过滤结果。

8.2.4 参数标签

考虑到iBATIS允许定义没有参数的已映射语句。参数标签（parameter tag）就是用来检查某个特定参数是否被传递给了已映射语句。该标签的属性如表8-6所示。

表8-6 参数标签的属性

prepend（可选的）	该值用于作为前缀附加到标签的结果内容体前。只有在以下3种情况下，prepend值才不会被加为前缀：(a)当标签的结果内容体为空时；(b)如果该标签第一个产生内容体，并且它被嵌套在一个removeFirstPrepend属性被设置为true的父标签中时；(c)如果此标签是跟在prepend属性取值非空的<dynamic>标签后的第一个生成内容体的标签
Open（可选的）	该值用于作为前缀附加到标签的结果内容体前。如果结果内容体为空，open值将不会被附加到其前面。open值将在prepend属性值被添加前缀之前先被添加前缀。例如，假设prepend="OR"，而open="("，则最终得到的组合前缀将会是"OR ("
close（可选的）	该值用于作为后缀附加到标签的结果内容体后。如果结果内容体为空，则close值将不起作用

(续)

removeFirstPrepend (可选的)	该值用于决定第一个产生内容的嵌套子标签是否删除其prepend值
--------------------------	----------------------------------

表8-6中所有属性对于表8-7中的标签都是可用的。

表8-7 参数标签

<isParameterPresent>	确定参数对象是否出现
<isNotParameterPresent>	确定参数对象是否存在

代码清单8-6显示了如何在SELECT语句中使用参数标签。

代码清单8-6 参数标签示例

```
<select id="getProducts" resultClass="Product">
  SELECT * FROM Products
  <isParameterPresent prepend="WHERE ">
    <isNotEmpty property="productType">
      productType=#{productType#}
    </isNotEmpty>
  </ isParameterPresent >
</select>
```

在此示例中，我们创建了一个简单的SELECT语句，不过这一次我们根据productType参数来选择性地创建WHERE子句以过滤结果。

8.2.5 <iterate>标签

<iterate>标签以一个集合或数组类型的特性作为其property属性值，iBATIS通过遍历这个集合（数组）来从一组值中重复产生某种SQL小片段。这些小片段以conjunction属性值作为分隔符连接起来，从而形成一个有意义的SQL语句片段，open属性值将作为所呈现的值列表的前缀，close属性值将作为所呈现的值列表的后缀，最终动态形成一个完整合法的SQL。<iterate>标签属性如表8-8所示。

表8-8 <iterate>标签属性

property (必需的)	包含列表的参数的特性
prepend (可选的)	该值用于作为前缀附加到标签的结果内容体前。只有在以下3种情况下，prepend值才不会被加为前缀：(a)当标签的结果内容体为空时；(b)如果该标签第一个产生内容体，并且它被嵌套在一个removeFirstPrepend属性被设置为true的父标签中时；(c)如果此标签是跟在prepend属性取值非空的<dynamic>标签后的第一个生成内容体的标签
Open (可选的)	该值用于作为前缀附加到标签的结果内容体前。如果结果内容体为空，open值将不会被附加到其前面。open值将在prepend属性值被添加前缀之前先被添加前缀。例如，假设prepend="OR"，而open="("，则最终得到的组合前缀将会是"OR ("
close (可选的)	该值用于作为后缀附加到标签的结果内容体后。如果结果内容体为空，则close值将不起作用

(续)

conjunction (可选的)	该值用于连接iBATIS遍历集合(数组)时重复产生那些SQL小片段
removeFirstPrepend (可选的)	该值用于决定第一个产生内容的嵌套子标签是否移除其prepend值

代码清单8-7显示了如何使用<iterate>标签来为SQL语句构建更加复杂的WHERE子句。

代码清单8-7 <iterate>标签示例

```

...
<select id="getProducts" parameterClass="Product"
      resultClass="Product">
  SELECT * FROM Products
  <dynamic prepend="WHERE productType IN ">
    <iterate property="productTypes"
      open="(" close=")"
      conjunction=",">
      productType=#productType[]#
    </iterate>
  </dynamic>
</select>
...

```

在此例中，我们创建了一个SELECT语句，然后遍历一组产品类型（product type）以为其构造了一个更加复杂的过滤器。

8.3 一个简单而完整的示例

现在你已经了解了关于动态SQL的基本知识，下面就利用动态SQL来实现一个简单的搜索功能，此搜索功能将在某个应用程序中被反复使用，这个应用程序就是JGameStore（见图8-2），我们将在第14章中更为正式地介绍这个应用程序。在给出本节中这个例子的过程中，我们将使用一种简单的方式，以帮助你消化动态SQL的概念并装配出自己的动态SQL。

在介绍此例子之前，先来研究一下构建动态SQL的过程。这个过程本身实际上非常简单，然而随着你的应用程序逐步成熟，过程中各个步骤的使用顺序可能会有不同，且为实施某个步骤所需付出的精力也会变化。正所谓万事开头难，做每件事时在开始时总是需要稍微多做一点工作，因为你必须构建基础设施。一旦基础打好了，在基础之上继续工作就不会那么复杂了。

这个过程由若干个基本步骤组成：

- (1) 描述数据将如何被检索和显示。
- (2) 确定将涉及哪些数据库结构。
- (3) 以静态格式编写SQL。
- (4) 将动态SQL标签应用到静态SQL上。

图8-2是一个非常简单的页面，但是正如你所知道的那样，它肯定不止看上去那么简单，其背后往往隐藏着更多的东西。因此，以下的几节将研究使这个页面能够正常工作的代码。

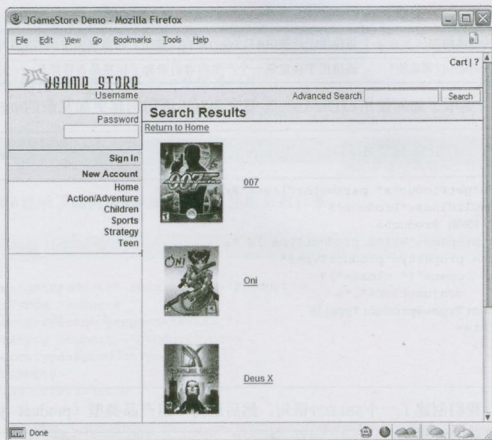


图8-2 JGameStore应用程序中的搜索结果屏幕

8.3.1 定义如何检索和显示数据

在JGameStore应用程序中的每个页面上，我们都想要实现一个用于搜索的文本输入框和按钮。搜索输入框中的输入通过空格来分隔搜索项。例如，如果输入Adventure Deus，那么表示输入两个搜索项，分别是Adventure和Deus。一旦单击了搜索按钮，将搜索每个产品的categoryId、名称和描述^①，看它们是否包含某个搜索项。所有的搜索结果将每页4个地列在搜索结果页面上。

8.3.2 确定将涉及哪些数据库结构

下一步我们就要定义查询应该涉及哪些表结构。既然我们只搜索categoryId、名称和描述，那么只需要Product表就可以满足需求（见图8-3）。Product表包含了我们在搜索时和显示时所需要的关于各种产品的全部信息。

Product	
PK	productid
	categoryId
	name
	description
	image

图8-3 简单产品查询所涉及的表

^① 这3个字段都是字符串类型。——译者注

8.3.3 以静态格式编写 SQL

对新手来说，首先，我们将构造一些静态SQL，这些静态SQL将选出所有需要在搜索结果中显示的必要字段。代码清单8-8给出了我们为完成产品信息显示而创建的静态查询。

代码清单8-8 静态SQL模型

```
SELECT
  PRODUCTID,
  NAME,
  DESCRIPTION,
  IMAGE,
  CATEGORYID
FROM PRODUCT
WHERE
  lower(name) like 'adventure%' OR
  lower(categoryid) like 'adventure%' OR
  lower(description) like 'adventure%' OR
  lower(name) like 'Deus%' OR
  lower(categoryid) like 'deus%' OR
  lower(description) like 'deus%'
```

既然已经定义了输入、输出以及查询涉及的表，下面就可以开始构造我们的SQL语句了。一旦有了之前这些信息，要构造出满足需求的SQL语句是非常简单的。SELECT语句就可以满足你的需要，返回一个产品列表。我们在其WHERE子句设置查询条件，根据所提供的搜索项在名称、categoryid以及描述列中搜索是否存在匹配。

8.3.4 将动态 SQL 标签应用到静态 SQL 上

下面来研究一下之前构造的SQL语句，解析它以确定需要在哪里引入动态标签。由于SELECT子句是静态的，我们将不需要对它进行任何动态化的改造。WHERE子句才是我们需要进行动态化调整的地方。代码清单8-9给出了我们将要研究的这个动态SQL。

代码清单8-9 动态SQL^①

```
<select id="searchProductList" resultClass="product" >
  SELECT
    PRODUCTID,
    NAME,
    DESCRIPTION,
    IMAGE,
    CATEGORYID
  FROM PRODUCT
  <dynamic prepend="WHERE">
    <iterate property="keywordList" conjunction="OR">
      lower(name) like lower('#keywordList[#]') OR
```

① 代码清单8-9第11行中“[]”符号放在iterate标签的property属性值之后，用于表示当前遍历中的列表元素，这是iterate标签的语法，但作者在本书中却始终没有提及，真是遗憾。——译者注

```
lower(categoryid) like lower(#keywordList[#]) OR  
lower(description) like lower(#keywordList[#])  
</iterate>  
</dynamic>  
</select>
```

既然动态标签使用了传递给已映射语句的参数，此处需要考虑一下这个参数是什么，以及如何对它使用动态标签。将要传递给已映射语句的参数会是一个字符串列表。由于我们直接使用一个简单的List对象作为参数，所以此处将使用<iterate>标签。<iterate>标签将遍历List对象中的所有字符串，以之构造查询条件作为其内容体，在categoryId、名称和描述列中寻找指定的搜索项。必须将conjunction属性值设置为OR。该值将用于把针对每一个搜索项设置的查询条件连接起来。这里想要强调的一点经验就是关于已映射语句的命名问题：总是使你的已映射语句的命名能够充分描述其包含的SQL的功能。在此例中，我们将已映射语句命名为searchProductList。这样当你阅读该已映射语句的名称时，就可以立即明白该语句的用途。

```
sqlMap.queryForPaginatedList(  
    "searchProducts", parameterObject, PAGE_SIZE);
```

为满足最后一个需求，需要确保一次只返回4个记录。为了完成此任务，我们将使用queryForPaginatedList()方法来调用已映射语句，该方法接受一个pageSize参数。这可以允许控制查询返回的记录数。

现在我们已经从头到尾研究了一个简单的例子，学习了如何计划并开发动态SQL。必须牢记的一点是，一条动态SQL必须只用于唯一——个目的。对于那些希望达到多个目的情况，不应该使用动态SQL。在我们的这个例子中，这个唯一的目的就是帮助用户选择产品。请始终牢记，发明动态SQL是为了使复杂的东西变得更加简单，而不是让简单的东西变得复杂。下面继续讨论动态SQL的另一个更加高级的用法。

8.4 高级动态 SQL 技术

在此例中，我们将使用一个购物车应用程序，该应用程序需要为客户提供一种能够使用更多细节搜索产品的方法。我们将在之前的例子所给出的一些结构的基础上，逐步增加复杂性。这里我们将再一次使用你在之前的例子中已经学过的分析方法。

8.4.1 定义结果数据

下面用更一般的术语来定义我们的结果数据^①。购物车应用程序要求当某个用户输入一组查询条件之后，可以搜索并显示出所有相符的产品构成的列表（见图8-4）。输出结果取决于输入的那些特定的查询条件，具体包括产品种类（category）、产品（product）和制造商（manufacturer）等。搜索得到的产品列表4个一组地依次显示在页面上。动态SQL必须能够生成分页的产品列表，并且能够处理用户可能输入的复杂多变的查询项。

① 即需求。——译者注

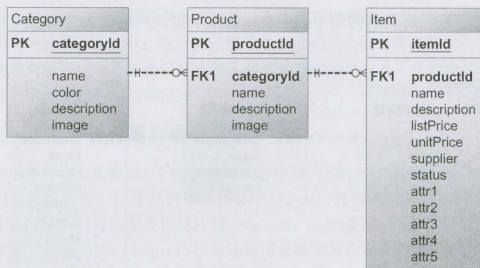


图8-4 搜索表单

那么，搜索页面是如何同数据库交互的呢？代码清单8-10给出了此问题的部分答案。

代码清单8-10 SearchCriteria.java搜索参数类

```
public class ProductSearchCriteria {

    private String[] categoryIds;
    private String productName;
    private String productDescription;
    private String itemName;
    private String itemDescription;

    ...
    // setters and getters
}
```

从本质上说，搜索页面上的每个输入框都将被映射到ProductSearchCriteria类的一个特性上，该类实际上就是一个简单的JavaBean。

8.4.2 定义所需的输入

下面我们将打破之前的示例中所采用的工作模式，在确定完成工作所需涉及的数据库结构之前，首先确定对输入的需求。用户将根据5种查询条件来进行搜索。这些查询条件分别是：产品类别（categoryId）、产品名称（product name）、产品描述（product description）、条目名称（item name）以及条目描述（item description）。产品名称和产品描述允许用户使用标准数据库百分号（%）语法进行通配符搜索。页面中还将有一个多选的下拉列表，以帮助用户把搜索范围限定在某些类别中。有了这些复杂的输入需求，我们还需要了解应该使用哪些数据库结构以满足这些要求。图8-5给出了在本例中我们将要处理的数据库表。

下面我们来定义有关的表结构。为满足需求我们需要使用3个数据库表（见图8-5）。Product表和前面例子中给出的是一致的。此例中引入的另外两个新表分别是Category和Item。Category

是一个用来定义产品所属类别的简单表。Item表用于定义产品的各个变种（various product permutations）。

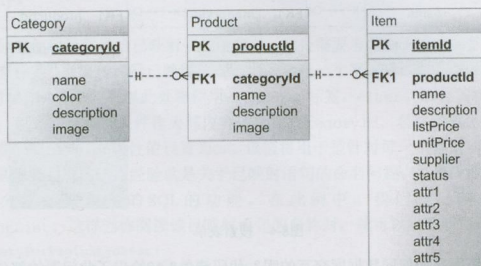


图8-5 搜索产品列表时所涉及的各个表及其关系

8.4.3 以静态格式编写 SQL

既然我们已经定义了输入和输出，下面就可以用简单的静态格式先编写一个SQL，并在查询工具中运行它就可以了。代码清单8-11给出了一个静态SQL语句，我们将从它开始，并且最终把它转变为一个动态语句。

代码清单8-11 静态SQL模型

```
SELECT
  p.PRODUCTID AS PRODUCTID,
  p.NAME AS NAME,
  p.DESCRPTION AS DESCRIPTION,
  p.IMAGE AS IMAGE,
  p.CATEGORYID AS CATEGORYID
FROM Product p
INNER JOIN Category c ON
  c.categoryId=p.categoryId
INNER JOIN Item i ON
  i.productId = p.productId
WHERE
  c.categoryId IN ('ACTADV')
AND
  p.name LIKE '007'
AND
  p.description LIKE '007'
AND
  i.name LIKE 'PS2'
AND
  i.description LIKE 'PS2'
```

现在我们已经知道了输入需要以及相关的表，下面就可以构造静态SQL查询了。在这个静态SQL语句示例（代码清单8-11）中，你可能已经注意到我们的WHERE查询条件已经被扩展以包含尽可能多的查询条件组合。为了能够更好地使人感知动态需要是什么，我们最好能把静态SQL构造得尽可能完备。下面开始构造动态SQL。

8.4.4 将动态 SQL 标签应用到静态 SQL 上

既然已经构造了静态SQL模型，下面就应用动态标签。正如你在代码清单8-12中所见的那样，动态SQL显得稍微复杂一些。注意此例中使用了<isEqual>作为顶级标签。此处没有必要使用<dynamic>作为父标签，因为不需要使用任何prepend、open或close值。我们需要的只是显示内容或者不显示它。相同的<isEqual>标签还被用来确定是否需要包含LEFT JOIN。

代码清单8-12 用于高级搜索的动态SQL

```
<select id="searchProductsWithProductSearch"
  parameterClass="productSearch"
  resultClass="product" >
SELECT DISTINCT
  p.PRODUCTID,
  p.NAME,
  p.DESCRPTION,
  p.IMAGE,
  p.CATEGORYID
FROM Product p
<isEqual property="itemProperties" compareValue="true">
  INNER JOIN Item i ON i.productId=p.productId
</isEqual>
<dynamic prepend="WHERE">
  <iterate
    property="categoryIds"
    open="p.categoryId IN (" close=")"
    conjunction="," prepend="BOGUS">
      #categoryIds[]#
    </iterate>

    <isNotEmpty property="productName" prepend="AND">
      p.name LIKE #productName#
    </isNotEmpty>

    <isNotEmpty property="productDescription" prepend="AND">
      p.description LIKE #productDescription#
    </isNotEmpty>

    <isNotEmpty property="itemName" prepend="AND">
      i.name LIKE #itemName#
    </isNotEmpty>
```

① 起始select标签

② 最小的SQL语句

③ 根据itemProperties字段是否为true确定是否需要使用表联结查询

④ 起始dynamic标签

⑤ 评测categoryIds特性

评测productName特性

评测productDescription特性

评测itemName特性


```
<isNotEmpty property="itemDescription" prepend="AND">
  i.description LIKE #itemDescription#
</isNotEmpty>
</dynamic>
</select>
```

评测itemDescription特性

起始select标签①将参数类(parameter class)的类型定义为productSearch^①，将结果类(result class)的类型定义为product。SQL片段②是该SQL中能被使用的最小部分——例如，假设用户只是想要执行一个返回所有产品的搜索。<isEqual>标签③被用在动态片段之外，用于确定是否需要使用关联查询^②然后我们使用了一个起始动态标签④，其prepend属性值为WHERE。如果其内容体中产生的结果内容为空，那么prepend属性值WHERE将被忽略。值得注意的是，动态标签也会隐式地移除其嵌套的第一个产生了内容的子标签的prepend值。还必须注意<iterate>标签⑤，因为它和动态标签④联合作用，使我们获得了可以将搜索限定在某些特定类别内的所有SQL组件。由于iBATIS当前存在的一个问题，对<iterate>标签来说必须要有prepend属性。第一个嵌套的标签所产生的内容不会被认为是第一个prepend，并将导致下一个标签的prepend属性被忽略。因此，作为规则，<iterate>标签一定要指定prepend属性^③，即使该标签可能根本不需要它。4个<isNotEmpty>标签分别用于评测productName、productDescription、itemName以及itemDescription特性，以确定它们的值不是零长度字符串或NULL。

那么，该如何调用以上这个庞然大物呢？

```
queryForPagedList(
    "Product.searchProductsWithProductSearch",
    productSearch, PAGE_SIZE);
```

和前面的示例一样，最后一个需求是保证一次只返回4条记录。为了完成此任务，我们使用queryForPagedList方法来调用已映射语句，该方法接受一个指示每页记录数的参数。有关如何使用queryForPagedList方法，请参考前面的示例。

到目前为止，我们已经研究了一个比较复杂的示例，并且学习了如何执行动态SQL。请牢记之前示例给出的所有建议和忠告，动态SQL只有一个唯一的目的——即根据用户输入的搜索查询条件搜索产品列表——现在我们已经完成了这个目的。虽然这个示例可能看上去有点复杂，但是如果直接用Java代码在没有任何框架帮助的情况下编写它，那将会复杂得多。至此，相信你應該已经有了足够的准备知识，可以独立构造自己的动态SQL了。

理解iBATIS动态SQL和其他动态SQL解决方案的异同点是非常重要的。下面来简单看一下动态SQL还可以用哪些其他的方式来完成。

① 这是ProductSearchCriteria类的别名。——译者注

② 查看JGameStore应用程序的源代码，可以发现ProductSearchCriteria类中并不存在itemProperties特性，但存在一个isItemProperties()方法，该方法判断itemName特性或itemDescription特性是否有同一个同时不为空和全由空格组成的字符串，是则返回true，否则返回false。——译者注

③ 示例中此处的BOGUS为“伪造的”之意。——译者注

8.5 动态 SQL 的其他替代方案

动态SQL绝不是一个新概念。根据条件构造不同的SQL查询，这样一个复杂的需求对开发人员来说一直都是一个不小的挑战。在过去，为了在执行动态SQL的同时兼顾效率，我们不得不处理使用了存储过程的私有数据库内容方式。在其他情况下，通过在更加健壮的编程语言中构造动态SQL而牺牲了性能，然后通过数据库驱动程序将查询传递到数据库中去执行。在以上这两种情况中，构造简单的SQL字符串的方式都不实用并且非常的复杂。如果你已经是iBATIS用户，本节内容将帮助简单回顾并且提醒一些你可能忘记的东西。如果你是iBATIS新手，本节可能是一个新鲜的介绍性比较。不管你是怎样的背景，我们都希望本节能够为你的实践提供全新的视角，并且使你确信iBATIS能够极大地减少你可能经历的复杂工作。

我们将用一个Java代码示例和一个存储过程示例来完成相同的动态SQL。代码清单8-13是一个不算太复杂的SELECT语句，编写它应该非常容易。假设其IN子句的内容可根据用户输入动态变化，这就成为了一个动态SQL问题。在介绍了使用Java代码和存储过程是如何解决该动态SQL问题之后，我们将把这两种方式同iBATIS进行简单的比较。

代码清单8-13 静态SQL模型

```
SELECT *
FROM Category
WHERE
categoryId IN ('ACTADV','SPORTS','STRATEGY') AND
name LIKE ('N%')
```

8.5.1 使用 Java 代码

在Java中编程是非常容易的。但是需要把Java和SQL混合在一起时，就必须小心地编写代码以保持其清晰性了。随着需求变得越来越复杂，你很可能就会把代码逐渐弄得一团糟。下面就看一个略微有点复杂的示例，该示例直接使用JDBC来动态组装一条动态SQL语句，然后把它传递给数据库。代码清单8-14给出了我们将用于构建SQL语句的查询条件。

代码清单8-14 CategorySearchCriteria.java

```
public class CategorySearchCriteria implements Serializable {

    private String    firstLetter;
    private List      categoryIds;

    ...
    // setters and getters
}
```

这个不算太复杂的SQL语句将从CategorySearchCriteria的categoryIds特性接收到类别ID的未知数量这些类别ID将被用来填充IN语句，firstLetter特性包含一个字符串前缀，用来针对该类别名的第一个字母进行搜索。在这个示例中，我们的焦点集中在与JDBC交互以及各种动态SQL解决方案的比较上。所以，我们并不详述与此无关的任何代码。代码清单8-15展示了如

有仅仅使用Java代码来构建这个动态SQL。

代码清单8-15 CategorySearchDao.java

```
public class CategorySearchDao {  
    ...  
    public List searchCategory(  
        CategorySearchCriteria categorySearchCriteria) {  
  
        List retVal = new ArrayList();  
        try {  
  
            Connection conn =  
                ConnectionPool.getConnection("MyConnectionPool");  
            // 获取JDBC资源  
  
            PreparedStatement ps = null;  
            ResultSet rs = null;  
  
            List valueList = new ArrayList();  
  
            StringBuffer sql = new StringBuffer("");  
            // 开始构建SQL查询  
            sql.append("SELECT * ");  
            sql.append("FROM Category ");  
            // 开始构造动态部分  
  
            if (categorySearchCriteria.getCategoryIds() != null &&  
                categorySearchCriteria.getCategoryIds().size() > 0) {  
  
                Iterator categoryIdIt =  
                    categorySearchCriteria.getCategoryIds().iterator();  
  
                sql.append("WHERE ");  
                sql.append("categoryId IN (");  
  
                if (categoryIdIt.hasNext()) {  
                    Object value = categoryIdIt.next();  
                    valueList.add(value);  
                    sql.append("?");  
                }  
  
                while (categoryIdIt.hasNext()) {  
                    Object value = categoryIdIt.next();  
                    valueList.add(value);  
                    sql.append(",?");  
                }  
  
                sql.append(")");  
  
                if (categorySearchCriteria.getFirstLetter() != null  
                    &&  
                    !categorySearchCriteria.getFirstLetter().trim().equals(""))
```



```
{
    if(valueList.size() != 0) { ← 不要忘记AND!
        sql.append("AND ");
    }

    sql.append("name LIKE (?)");
    valueList.add(categorySearchCriteria.getFirstLetter()
        + "%");
    }

    ps = conn.prepareStatement(sql.toString()); ← 准备待执行语句并且
                                                设置其参数

    Iterator valueListIt =
        valueList.iterator();

    int indexCount = 1;

    while(valueListIt.hasNext()) {
        ps.setObject(indexCount, valueListIt.next());
        indexCount++;
    }

    rs = ps.executeQuery(); ← 运行查询语句

    while(rs.next()) { ← 循环处理结果并创建用于返回
                        的对象
        Category category = new Category();
        category.setCategoryId(rs.getInt("categoryId"));
        category.setTitle(rs.getString("title"));
        category.setDescription(rs.getString("description"));
        category.setParentCategoryId(
            rs.getInt("parentCategoryId"));
        category.setSequence(rs.getInt("sequence"));

        retVal.add(category);
    }
} catch (SQLException ex) {
    logger.error(ex.getMessage(), ex.fillInStackTrace());
} finally {
    if (rs != null) ← 回收资源的“快乐”
        try { rs.close(); }
        catch (SQLException ex)
        {logger.error(ex.getMessage(), ex.fillInStackTrace());}
    if (ps != null)
        try { ps.close(); }
        catch (SQLException ex)
        {logger.error(ex.getMessage(), ex.fillInStackTrace());}
    if (conn != null)
        try { conn.close(); }
        catch (SQLException ex)
        {logger.error(ex.getMessage(), ex.fillInStackTrace());}
}

return retVal;
}
```

```
--  
}
```

使用Java代码创建动态SQL，你需要亲自处理所有的基础性任务（且这些任务往往具有重复性），例如获得连接、准备参数、迭代结果集，以及填充返回对象等。除了管理普通任务之外，在此例中我们还要处理由于需要动态创建IN语句而引入的复杂性。为了处理此任务，我们创建了一组参数以填充IN语句。即使在编写了所有这些代码之后，但与理想情况相比仍然缺少一项功能。你可能已经注意到了，PreparedStatement只能使用setObject来设置参数，而最理想的情况是，我们需要能够指定参数的类型。要实现这个需求，相信我们的代码还会庞大得多。

最终，我们的Java代码还非常难以理解。还记得小时候的游戏吗，我们要在杂志的大幅图画中找出那些被巧妙隐藏的图画。是不是觉得我们的Java代码与这个游戏有几分相似呢。是啊，阅读我们的Java代码需要面对的一个挑战就是努力从中找出隐藏的SQL。在这个示例中，直接使用JDBC显然让人难以忍受。让我们看看如果使用存储过程，情况又会怎样吧。

8.5.2 使用存储过程

对于很多任务来说，存储过程可以很好地解救我们于危困之中，应该感激存储过程所能做的一切。但是当需要使用存储过程来创建动态SQL时，它们却常常遇到和用Java编码的动态SQL一样的问题。代码清单8-16显示了一个用Oracle的PL/SQL编写的存储过程，让我们看看它是如何处理动态SQL构造的。

代码清单8-16 Oracle存储过程（由Sven Boden提供）

```
create or replace package category_pkg  
as  
    type ref_cursor is ref cursor;  
  
    function get_category(  
        categoryid varchar default null,  
        name category.name&type default null)  
        return ref_cursor;  
end;  
/  
  
create or replace package body category_pkg  
as  
    function get_category(  
        categoryid varchar default null,  
        name category.name&TYPE default null)  
        return ref_cursor  
is  
    return_cursor ref_cursor;  
    sqltext      varchar(4000);  
    first        char(1) default 'Y';  
begin  
    sqltext :=  
        'select c.categoryid, c.title, c.description, ' ||  
-- 开始创建SQL
```

```

        'c.sequence ' ||
        ' from category c ';;
    if ( categoryid is not null ) then <—— 在SQL中添加待查询的那些类别ID
        if ( first = 'Y' ) then
            sqltext := sqltext ||
                'where c.categoryid in ( ' ||
                    categoryid || ' ) ';;
            first := 'N';
        end if;
    end if;

    if ( name is not null ) then <—— 在SQL中添加名称
        if ( first = 'Y' ) then
            sqltext := sqltext || 'where ';;
        else
            sqltext := sqltext || 'and ';;
        end if;
        sqltext := sqltext || 'c.name like ' || ' ' ||
            name || '% ';;
        first := 'N';
    end if;

    open return_cursor for sqltext; <—— 执行SQL

    return return_cursor; <—— 返回结果
end get_category;

end;
/

```

必须承认，代码清单8-16中给出的示例打破了使用存储过程时一个非常有价值的准则：不要使用参数绑定，否则会带来SQL注入的可能性并降低性能。在存储过程中使用参数，我们不但不能减小复杂度，甚至可能会增加复杂度。因此，避免复杂度总是我们必须遵循的规则么？当然不是的！但是对于动态SQL而言，我们很难找到一个理由利用存储过程来执行此任务。

通常我们使用存储过程的两大原因是安全性和性能，但是对于动态SQL而言，这两者都不适用。在Java中或者在存储过程中使用参数化的SQL，性能和安全性都没有任何差别。而当我们转而关注易读性和可维护性时，Java代码的表现则同样让人气馁。存储过程比Java示例要更加易读，这或许是由于它没有受到参数化SQL的包裹。有关可维护性，情况就复杂了。对于存储过程，在部署应用程序时，我们必须依赖数据库管理员来配置DDL脚本。而对于Java示例来说，SQL是完全由开发人员控制的，它们将和代码库的其他部分一样地被部署。

使用存储过程的好处不尽相同，这完全取决于所使用的数据库，以及该数据库的内部语言在多大程度上适合于你的复杂任务。使用存储过程来创建动态SQL时，最终你将得到与使用Java代码一样的复杂度——不会增加安全性，也不会提升性能，并且部署更加复杂。另一件需要注意的事就是，我们甚至还没有包括用来第一次调用存储过程的Java代码。如果硬要作一个综合评估的话，我们可能很难选择究竟是使用Java代码还是使用存储过程。既然如此，为何不考虑使用iBATIS呢。

8.5.3 同 iBATIS 相比较

研究了如何直接使用Java代码和存储程序来创建动态SQL之后，我们迫切需要一种能够同时为我们提供高性能、高可靠性和高生产效率的解决方案。代码清单8-17给出了使用iBATIS的SqlMap框架创建的和代码清单8-15以及代码清单8-16功能一样的动态SQL。

代码清单8-17 iBATIS动态SQL

```
<select id="getCategories" parameterClass="SearchClass"
  resultClass="CategorySearchCriteria">
  SELECT *
  FROM Category
  <dynamic prepend="WHERE">
    <iterate prepend=" categoryId IN"
      open="(" close=")" conjunction=",">
      #categoryIds[]#
    </iterate>
    <isNotEmpty property="categoryName" prepend="AND">
      name LIKE ( #categoryName# || '%' )
    </isNotEmpty>
  </dynamic>
</select>
```

以上就是该已映射语句的代码了，可以用如下代码调用该已映射语句：

```
queryForList("getCategories", searchObject);
```

从这个例子可以看出，我们仅仅使用了大约14行代码就完成了用Java代码或存储过程需要几倍的代码量才能完成的工作。由于iBATIS内在地使用PreparedStatement，因此我们能够避免SQL注入带来的安全问题，同时又获得参数化SQL的执行性能。通过把SQL保存在Java源代码之外的一个简单的XML文件中，我们也可以更易于维护SQL并且可以将它们与应用程序一齐部署。如果要对iBATIS、Java代码、存储过程三者进行比较分析，那么谁是胜利者相信将毫无疑问。

8.6 动态 SQL 的未来

iBATIS已经开始展望未来，并且已经采取行动改进动态SQL了。你在本章中所学到几乎所有的东西，在未来的动态SQL中仍将是恰切的。但是，了解iBATIS将朝着哪个方向发展它的动态SQL是很重要的。

有关动态SQL标签集的最初想法是在iBATIS 1.x版中发展起来的。所以建立动态SQL标签集，主要是借鉴了Struts建立的标签库（更多细节请参考Ted Husted的*Struts in Action*, [Manning, 2002]）的思想。但随着Java社区中相关标准的改进，iBATIS现在试图借鉴更加标准的Java概念，并且将这些概念融入到iBATIS中。有两个领域需要改进：一是提供一个更简化更健壮的标签集，二是提供一个可以和标签联合使用的简单的表达式语言。下面来看一下iBATIS将如何改进动态SQL。

8.6.1 简化的条件标签

目前iBATIS以拥有16个健壮的可用于构建动态SQL的标签而感到自豪。然而，这些标签太特

殊了。在为提供更加通用功能的条件标签集所作的努力中，iBATIS团队计划引入一个简化标签集，以便补充现有的标签。我们的目标是使新一代的动态SQL标签能够最终完全替代旧的标签。新一代动态SQL将模仿JSTL (Java Standard Tag Library, Java标准标签库)。这将使我们现有的16个标签减少为6个。书写本书时，所提议的新标签分别是<choose>、<when>、<otherwise>、<if>、<foreach>以及<while>。在大多数情况下，这些标签和它们在JSTL中对应的标签具有完全相同的功能，不同之处仅在于它们将包含一些额外的属性，如prepend、open、close以及removeFirstPrepend等。

8.6.2 表达式语言

因为新的动态SQL标签集将更加通用，所以需要一种简单的表达式语言。iBATIS小组决定仿造Java的J2EE表达式语言 (Expression Language, EL) 来建立自己的表达式语言。这将更好地支持在一个单一的评测中完成更多的条件分析。当前的动态SQL标签集不支持类似于“and”和“or”这样的布尔操作。当通用标签同强大的表达式语言组合起来后，就可以更容易地实现复杂的动态SQL要求了。

8.7 小结

iBATIS的动态SQL是集成库中的一个功能强大的工具。充分理解它在开发数据库交互中的位置非常重要。使用iBATIS的动态SQL时请务必牢记，必须简化你的目标。

在本章中，你学习了如何使用Java和PL/SQL代码来编写简单的和复杂的动态SQL，以及如何如何在iBATIS中完成相同的任务。尽管有时候iBATIS的动态SQL可能不能完全满足你的需求，但是假如如设置了其他的替代方案，动态SQL仍然不失为一种好方法，它可以满足90%的查询需求。

Part 3

第三部分

真实世界中的 iBATIS

iBATIS 框架使用了一种分层架构，这使你可以更容易地只使用其中你需要的组件，而不必管那些你不需要的特征。然而，有时候你需要的不仅仅是基本功能。第三部分将介绍更高级的 iBATIS，并且将向你展示如何使用 iBATIS 的高级特征。你将学习到有关 iBATIS 的高速缓存机制和数据层抽象的知识，同时我们还将告诉你当所有其他方法都失效时如何扩展 iBATIS 以满足你的特别需求。

本部分 内容

- 第 9 章 使用高速缓存提高性能
- 第 10 章 iBATIS 数据访问对象
- 第 11 章 DAO 使用进阶
- 第 12 章 扩展 iBATIS

第9章

使用高速缓存提高性能

本章内容

- 高速缓存的理念
- 高速缓存的配置
- 高速缓存的策略

通常意义的高速缓存含义非常广泛。例如，在一个包含表现层、服务层和数据访问层的传统Web应用程序中，对其中任何一层或者所有层进行高速缓存都是合理的。但iBATIS的高速缓存只关注在持久层对结果进行高速缓存。因此，iBATIS的高速缓存与服务层以及表现层都无关，并且不依赖于对象标识。

本章将介绍如何配置高速缓存、优化高速缓存以及如何扩展iBATIS高速缓存实现。

9.1 一个简单的 iBATIS 高速缓存示例

iBATIS的强健且简单的高速缓存机制是完全基于配置的，因此避免了直接管理高速缓存的负担。在深入介绍应该在何时使用iBATIS高速缓存、为什么要使用它，以及如何使用它等问题之前，我们先简单介绍一下iBATIS高速缓存。代码清单9-1给出了一个简单的高速缓存配置以及一个使用此配置的已映射语句。

代码清单9-1 基本高速缓存示例

```
<cacheModel id="categoryCache" type="MEMORY">
  <flushOnExecute statement="insert"/>
  <flushOnExecute statement="update"/>
  <flushOnExecute statement="delete"/>
  <property name="reference-type" value="WEAK"/>
</cacheModel>

<select
  id="getCategory" parameterClass="Category"
  resultClass="Category" cacheModel="categoryCache">
```

```
SELECT *  
FROM Category  
WHERE categoryId=#categoryId,  
</select>
```

在代码清单9-1的示例中，可以看到两个主要的组件：高速缓存模型和已映射语句（select 语句）。高速缓存模型定义了如何在高速缓存中存储新的查询结果以及清除失效数据。需要使用高速缓存的已映射语句只需要通过<select>和<procedure>标签的cacheModel属性来引用它。

在代码清单9-1中，高速缓存模型指定其高速缓存类型为MEMORY。这是iBATIS内置的一种高速缓存方式，可以把查询结果直接存储在内存中。这通常也是iBATIS中最常使用的一种高速缓存方式。高速缓存模型的内容体中有一对标签。<flushOnExecute>标签用于指定当某个特定高速缓存被访问时，其存储结果将被清除。注意，清除意味着所有的高速缓存内容都将被清空。因此，如果你有若干条使用了相同高速缓存模型的已映射语句，那么，这些已映射语句的所有结果都将被清除。最后一个要注意的标签为<property>。每一种类型的高速缓存模型都有一些专用于它自己配置的特性，<property>标签就是用于完成这些专用特性的设置的。在<property>的name属性中指定该高速缓存模型将要设置的特性的名称，value指定该预定义特性的值。

正如你在之前的代码中所看到的，iBATIS高速缓存非常简单。你需要做的主要工作可能就是，发现每一类高速缓存模型都可用的所有特性。本章在快速介绍完iBATIS高速缓存的基本理念之后，我们将解释有哪些可用的高速缓存模型，并介绍何时以及如何使用它们。

9.2 iBATIS 高速缓存的理念

开发人员使用高速缓存，大部分都是用它来存储那些长效的、似乎不会发生改变的数据。例如你在下拉菜单或者选择框列表中看到的就是这样一类数据。类似于州、城市和国家名这样的数据最适合使用这种类型的高速缓存。但是，高速缓存并不仅仅只能用于高速缓存那些长效的只读数据，它也可以用于高速缓存那些可修改的对象。

此时的难点就在于，你需要手动检查被请求的数据是否存在于当前高速缓存中，如不存在，需要自己将它保存到高速缓存中。有关高速缓存的另一个难点就是，如何判断高速缓存中所保存的数据是否已经过时。编写类似于定时高速缓存清除这样的简单规则来保证高速缓存的有效性是非常简单的，但是当存在多个进程，且它们的执行都会造成高速缓存内容的失效时，问题就不那么简单了。一旦正在执行的代码之间的依赖性对于维护高速缓存对象的完整性变得十分重要时，情况就变得更加复杂了。通常当高速缓存达到这样的复杂度时，性能-代价比（performance-to-effort ratio，即所获得的性能与所付出的代价的比例）将变得不那么令人信服，或者至少看上去将是一项很麻烦的任务。这就是为何iBATIS只专注于数据访问层的高速缓存实现和高速缓存策略。只针对数据访问层，使得框架通过配置文件来管理高速缓存，而配置文件的管理是非常容易的。

现在考虑一下iBATIS高速缓存框架的基本理念，以及它和其他持久层解决方案的区别。这对于那些习惯于用传统的O/RM解决方案执行高速缓存的人来说常常是一个痛苦的过程。iBATIS的思想是建立SQL语句到对象间的映射，而不是建立数据库表到对象间的映射。要特别注意这两者

之间的差异。传统的ORM工具主要关注数据库表到对象间的映射，这也会影响它们的高速缓存。例如，传统的ORM高速缓存会为其管理的每个对象维护一个OID^①，就像数据库需要管理其表中每条记录的唯一性一样。这意味着，如果两个不同的结果都返回同一个对象，那么该对象将只被高速缓存一次。但是iBatis就不是这样的。因为iBatis是一个以数据为中心的框架，它关注的是SQL语句的执行结果，所以我们不会根据对象的唯一性来高速缓存它们iBatis高速缓存返回的所有结果，而不考虑所标识的对象是否存在于高速缓存中。

下面将深入讨论什么是高速缓存模型，以及它有哪些公共的组件。我们将看到高速缓存模型如何能够帮助避免因手工管理高速缓存结果及其依赖性而造成的大量繁琐的工作。

9.3 理解高速缓存模型

有关高速缓存模型，最简单的描述就是，它是一种高速缓存配置。或者更确切地说，它是定义所有的iBatis高速缓存实现的基础。高速缓存模型的配置是在SQL Map配置文件中定义的，并且可以被一个或者更多的查询已映射语句所使用。

高速缓存的配置是通过<cacheModel>标签来定义的，此标签包含了表9-1中所列出的这些属性。

表9-1 <cacheModel>标签属性

id (必需的)	该值用于指定一个唯一的ID，便于为需要使用此高速缓存模型所配置的高速缓存的查询已映射语句所引用
Type (必需的)	此属性用于指定高速缓存模型所配置的高速缓存的类型。其有效值包括MEMORY、LRU、FIPO和OSCACHE。该属性也可取值为某个自定义CacheController实现的全限定类名
readOnly (可选的)	将该值设置为true，就表示高速缓存将仅仅被用作只读高速缓存。从只读高速缓存中读出的对象的特性值不允许更改
serialize (可选的)	该属性用于指定在读取高速缓存内容时是否要进行“深复制”

id属性用于标识高速缓存，这样iBatis就能知道哪些已映射语句在该高速缓存中存储了数据。下面让我们更详细地学习其他的属性。

9.3.1 type 属性

iBatis提供了4个默认的高速缓存实现，高速缓存模型可以直接使用它们。这4个类型如表9-2所示。

表9-2 内置的各种高速缓存模型类型

高速缓存模型类型	描 述
MEMORY	这个模型简单地将高速缓存数据保存在内存中，直至垃圾收集器将它移除
FIFO	这个模型中，高速缓存的数据量是固定的，使用“先进先出（first in first out）”算法来移除高速缓存中的数据
LRU	这个模型中，高速缓存的数据量也是固定的，使用“最近最少使用（least recently used）”算法来移除高速缓存中的数据
OSCACHE	这个模型使用OpenSymphony（简称OS）高速缓存

① object identification，对象标识，用于唯一地标识一个对象。——译者注

要使用何种高速缓存实现，是通过将它们相应的默认关键字（MEMORY、LRU、FIFO以及OSCACHE）添加到<cacheModel>标签的type属性中来指定的。9.5节将详细讨论这4种默认的高速缓存实现。

也可以自己实现CacheController接口，然后在type属性中指定其全限定类名，以提供一套自己的高速缓存方案（见第12章）。

9.3.2 readOnly 属性

<cacheModel>标签提供了一个readOnly属性。该属性仅仅是一个为高速缓存模型提供指令的指示器，用于告诉高速缓存模型应该如何检索和保存已高速缓存对象。将该属性值设置为true，并不能保证从高速缓存中检索出的对象内容不被改变。指定一个高速缓存为只读时，也就是告诉高速缓存模型允许其返回对存在于其高速缓存中的某个对象的引用，因为该对象将不会被正在请求它的应用程序所改变。如果readOnly属性被设置为false，就可以确保不会出现多个用户同时访问某个已高速缓存对象的同一引用实例的情况。readOnly属性通常与serialize属性联合使用。理解这两个属性是如何联合工作的，这一点非常重要。

9.3.3 serialize 属性

serialize属性用于指示已高速缓存对象应该被如何返回。当该属性被设置为true时，高速缓存中所请求的每个对象都将作为一个深副本被返回。这就意味着从高速缓存中检索出的对象只是具有相同的值，但并不是同一个实例。这就可以确保存储在高速缓存中的实际数据永远不会被改变。需要注意的是，此处所谓的串行化（serialization）并不是大家通常所理解的串行化，它并不是将对象串行化到磁盘中。这里的串行化是基于内存的串行化，用于创建内存中已高速缓存对象的深副本。

9.3.4 联合使用 readOnly 属性和 serialize 属性

现在你已经理解了这两个属性，它们看上去在功能性上似乎存在一些重叠。事实上，它们需要紧密协同才能正常工作。当你把这两个属性进行不同的组合时，必须理解幕后究竟发生了什么。表9-3中给出了4种所有可能的组合，并且分析其中每一种组合的优点（或者缺点）。

表9-3 联合使用readOnly和serialize属性概要

readOnly	serialize	结 果	原 因
True	False	好	可以最快速地检索出已高速缓存对象。返回已高速缓存对象的一个共享实例，若使用不当可能会导致问题
False	True	好	能快速检索出已高速缓存对象。返回已高速缓存对象的一个深副本
False	False	警告！	对于此种组合，高速缓存仅仅同调用线程的会话的生命周期相关，且不能被其他线程所使用
True	True	坏	这种组合同readOnly=false而serialize=true的组合作用一致，否则它在语义上没有任何意义

以上这两个属性的默认组合是`readOnly=true`和`serialize=false`。这种组合告诉高速缓存模型始终返回其高速缓存中所包含的同一个引用。使用这种组合时，已高速缓存对象有可能被更改。这就会产生问题，因为这些对象是全局共享的，所有使用相同参数通过查询已映射语句访问已高速缓存对象的用户，检索得到的都是相同的一些已高速缓存对象，但这些对象可能已被另一个会话不恰当地更改了。

当你确实想要更改已高速缓存对象时，就应该把`readOnly`属性设置为`false`。这将强制高速缓存返回一个特定于该会话的实例。把设置为`false`的`readOnly`属性同设置为`true`的`serialize`属性组合起来时，就可以得到一个已高速缓存对象的深副本。这时对所检索对象的更改，就可以同调用会话（即发起检索的会话）隔离开来了。

另一个可用的组合是，把`readOnly`属性设置为`false`，同时把`serialize`属性设置为`false`。这种组合是有用的，但其适用的场合却非常少。把这两个属性都设置为`false`，就会要求高速缓存为调用线程生成被请求对象的唯一实例。由于`serialize`属性被设置为`false`，因而不采用深复制策略。相反地，高速缓存仅仅在该会话的生命周期中被创建和使用。这意味着如果在同一会话中若干次调用同一条查询已映射语句，那么你就可以享受到高速缓存的好处。然而，每个会话在第一次调用某条已高速缓存的查询已映射语句时，数据库访问仍然不可避免。

最后一种组合是`readOnly=true`和`serialize=true`。这种组合同`readOnly`设置为`false`而`serialize`设置为`true`的组合相比较，功能性是一致的。此处的问题是它所表达的语义。创建一个可能需要串行化的只读（`readOnly`）结果，是没有任何意义的。设置`serialize`属性为`true`表明你期望（或计划）高速缓存中的对象能够以某种方式被复制（如深复制），以保证对象更改不会造成问题，而设置`readOnly=true`又表明你不期望高速缓存中的对象可以被更改。因此，串行化一个只读的高速缓存是非常荒唐的。

好了，现在你已经从理论上理解了高速缓存，下面进一步深入讨论如何设置并使用它。

9.4 如何使用高速缓存模型中的标签

在探讨几种不同类型的高速缓存模型实现之前，应该首先熟悉用于`<cacheModel>`标签内容体内的那几个子标签。这些子标签可用于定义清除高速缓存的通用动作，也可用于指定仅与某个特定高速缓存模型实现相关的特性值。

9.4.1 高速缓存的清除

每个高速缓存实现都可使用一组公用的标签，用于清除其高速缓存的内容。在设计高速缓存模型时，请先仔细考虑以确定如何以及何时需要从高速缓存中移除对象。默认情况下，每种类型的高速缓存模型都有一种自己的方式，用于在某个粒度级别上管理其高速缓存数据。它们可以根据内存、近期访问率或者以某个周期来移除高速缓存中的对象。除了每个高速缓存模型的固有行为之外，还可以进一步指示它们应该在何时清除其所有的内容。可以利用清除标签来提供这种功能。

共有两个清除标签，如表9-4所示。

表9-4 清除标签，用于定义从高速缓存中清理数据的规则

标签名称	用 途
<flushOnExecute>	定义查询已映射语句，其执行将引起相关高速缓存的清除
<flushInterval>	定义一个时间间隔，高速缓存将以此间隔定期清除

下面更深入地研究这两个清除标签。

1. <flushOnExecute>标签

<flushOnExecute>标签只有一个属性statement，当某条已映射语句执行时，将触发相应高速缓存的清除。这个功能在你拥有应当在底层数据库数据发生改变而更新的结果时显得尤其有用，因为此时你必须更新高速缓存以保证与数据库的一致。例如，假设你有一个高速缓存，其中存储了类别列表，每当有一个新的Category对象插入数据库时，就可以使用<flushOnExecute>标签来清除高速缓存。

如前所述，每次动作整个高速缓存将会被完全清除，因此对于那些经常更改的数据，应该非常谨慎地决定是否对其进行高速缓存，以避免创建已映射语句的清除依赖性。因为频繁的清除和填充下，你的高速缓存看起来没有任何意义。代码清单9-2定义了一个高速缓存模型，作为示例说明在把新数据添加到数据库时如何使用<flushOnExecute>标签来使高速缓存失效。

代码清单9-2 flushOnExecute高速缓存示例

```
<sqlMap namespace="Category">
  ...
  <cacheModel id="categoryCache" type="MEMORY">
    ...
    <flushOnExecute statement="Category.insert"/>
    ...
  </cacheModel>
  ...
  <select
    id="getCategory" parameterClass="Category"
    resultClass="Category" cacheModel="categoryCache">
    SELECT *
    FROM Category
    WHERE parentCategoryId=#categoryId#
  </select>
  ...
  <insert id="insert" parameterClass="Category" >
    INSERT INTO Category
    (title,description,sequence)
    VALUES
    (#title#,#description#,#sequence#)
  </insert>
  ...
</sqlMap>
```


要使用<flushOnExecute>标签，需要通过statement属性来指定某已映射语句的名称，该已映射语句的执行将触发高速缓存的清除。如果此已映射语句包含在一个指定了namespace属性的sqlMap文件中，那么statement属性也必须包含完整的名称空间。即使statement属性引用的是同一个sqlMap配置文件所限定的已映射语句，也必须使用完整的名称空间符号。当指定的已映射语句存在于另一个sqlMap配置文件中时，还必须确保所依赖的sqlMap文件在statement属性引用它之前就已经被加载了。

2. <flushInterval>标签

另一个用于管理高速缓存内容清除的标签就是<flushInterval>。<flushInterval>标签比<flushOnExecute>标签稍微简单一些，因为除了时间之外它没有任何配置上的依赖性^①。<flushInterval>标签每隔一定的时间间隔就会清除一次高速缓存。此时间间隔从高速缓存被创建（发生在加载配置期间）开始算起，一直持续到应用程序被关闭时结束。<flushInterval>标签允许你以小时、分钟、秒，甚至毫秒为单位指定时间间隔，如表9-5所示。

表9-5 <flushInterval>标签属性

属 性	描 述
hours (可选的)	每次清除高速缓存前应该经过的小时数
minutes (可选的)	每次清除高速缓存前应该经过的分钟数
seconds (可选的)	每次清除高速缓存前应该经过的秒数
milliseconds (可选的)	每次清除高速缓存前应该经过的毫秒数

为了避免任何可能的混淆，<flushInterval>不允许指定特定的时间点来清除高速缓存。它是完全基于时间间隔的。代码清单9-3给出了一个使用<flushInterval>标签的示例，它指定高速缓存对象的生命期为12小时。

代码清单9-3 <flushInterval>高速缓存示例

```
<sqlMap namespace="Category">
  ...
  <cacheModel id="categoryCache" type="MEMORY">
    ...
    <flushInterval hours="12" />
    ...
  </cacheModel>
  ...
  <select
    id="getCategory" parameterClass="Category"
    resultClass="Category" cacheModel="categoryCache">
      SELECT *
      FROM Category
      WHERE parentCategoryId=#categoryId#
    </select>
    ...
  </sqlMap>
```

① 如<flushOnExecute>标签就依赖于某条已映射语句的存在。——译者注

使用<flushInterval>标签必须牢记的是，它只允许指定一个属性。因此如果想要每隔12小时10分10秒5毫秒清除一次高速缓存，就必须把这个时间间隔计算为毫秒，并以该毫秒值作为输入。

9.4.2 设置高速缓存模型实现的特性

由于高速缓存模型只是一些可以插入到iBatis框架中的组件，它甚至允许用户自己定制，因此必须有一种方式能够为这些组件提供任意的值。<property>标签就是用来完成此任务的。该标签的属性如表9-6所示。

表9-6 <property>标签的属性

Name (必需的)	所设置的特性的名称
value (必需的)	所设置的特性的值

name和value属性都是必需的，它们被用来构建一个可传递给高速缓存模型组件以供其初始化的Properties对象。

因此，通常来说，高速缓存的配置与所使用的高速缓存类型并不是完全无关的。所以，了解高速缓存模型有哪些可用的类型以及每种类型所特有的配置项是必需的。

9.5 高速缓存模型的类型

正如9.3.1节所述，iBatis为应用程序提供了4种类型的高速缓存模型：

- MEMORY
- LRU
- FIFO
- OSCACHE

以下4节将分别介绍它们。

9.5.1 MEMORY

MEMORY高速缓存是一种基于引用的高速缓存（参考java.lang.ref包的Java文档）。高速缓存中的每个对象都被赋予一个引用类型。此引用类型为垃圾收集器提供了线索，指导它如何处理相应的对象。和java.lang.ref包中的类一样，MEMORY高速缓存也提供WEAK和SOFT两种类型的引用。当指定引用类型为WEAK或SOFT时，垃圾收集器就会根据内存约束和（或）对已高速缓存对象的当前访问，来确定应该保存哪些数据以及删除哪些数据。而当你使用的是STRONG类型的引用时，则无论发生什么情况高速缓存中的对象都会一直保留，直至到达指定的清除间隔^①。

MEMORY高速缓存模型对于那些更关注内存的管理策略而不是对象的访问策略的应用程序而言是完美的。有了STRONG、SOFT和WEAK这三种引用类型，就可以确定哪些结果应该比其

① 参见<flushInterval>标签。——译者注

他结果保留更长的时间。表9-7展示了每一种引用类型的功能，以及不同的类型是如何确定对象在内存中被高速缓存的时间的。

表9-7 MEMORY高速缓存引用类型

WEAK	WEAK引用类型将很快地废弃已高速缓存的对象。这种引用类型不会阻止对象被垃圾收集器收集。它仅提供一种方式来访问高速缓存中的对象，该对象在垃圾收集器的第一遍收集中就会被移除。这是MEMORY高速缓存默认的引用类型，如果保存在高速缓存中的所有对象都会以非常一致的方式被访问，那么使用这种引用类型就非常合适。由于已高速缓存对象被废弃的速度比较快，可以确保你的高速缓存不会超过内存限制。然而，使用这种引用类型时，数据库访问的频率会很高
SOFT	SOFT引用类型也适合于那些将满足内存约束看得很重要，必要时就会释放高速缓存中对象的情况。这种引用类型在满足内存约束的前提下，将尽可能地保留已高速缓存对象。此时，垃圾收集器始终不会收集对象，除非确定需要更多的内存。SOFT引用也确保不会超过内存限制，并且和WEAK引用类型相比，其数据库访问频率会低一些
STRONG	STRONG引用类型不管内存约束，其中的已高速缓存对象永远不会被废弃，除非到达了指定的清除时间间隔。STRONG类型的高速缓存应该用于存放那些静态的小对象，并且对于这些小对象的访问应该有一定规律。这个引用类型可以通过减少数据库访问频率提高性能，但当高速缓存中积累的数据越来越多时存在内存耗尽的风险

MEMORY高速缓存类型只有一个特性，即reference-type，它用于指定期望使用的引用类型。

代码清单9-4给出了一个简单的MEMORY高速缓存模型，该模型使用WEAK引用类型来高速缓存数据，且指定对象的高速缓存时间最长不超过24小时，每当有插入、更新或删除已映射语句被执行时，高速缓存都会被清除。

代码清单9-4 MEMORY高速缓存模型样例

```
<cacheModel id="categoryCache" type="MEMORY">
  <flushInterval hours="24"/>
  <flushOnExecute statement="insert"/>
  <flushOnExecute statement="update"/>
  <flushOnExecute statement="delete"/>
  <property name="reference-type" value="WEAK"/>
</cacheModel>
```

需要在应用程序中高速缓存数据时，MEMORY高速缓存类型是一种虽然简单但非常有效的方式。

9.5.2 LRU

LRU类型的高速缓存模型使用最近最少使用策略来管理高速缓存。该高速缓存的内部机制会在后台记录哪些对象最近最少被访问，一旦超过高速缓存大小限制就会废弃它们。只有当高速缓存超过大小限制这个约束条件时，才会废弃高速缓存的对象。大小限制定义了高速缓存中可以包含的对象数目。因此一定要避免将那些大的内存对象^①放置在此类高速缓存中，否则内存会很快

① 即占有内存较多的对象。——译者注

耗尽。

LRU高速缓存非常适用于那些需要根据对某些特定对象的访问频率来管理高速缓存的情况。通常这种高速缓存策略适用于那些需要高速缓存用于分页结果或关键搜索结果的对象应用程序中。

对于LRU高速缓存类型来说，在使用<property>标签时唯一能被指定的特性就是size，它用于指定能够保存在高速缓存中的对象的最大数目。

代码清单9-5展示了一个简单的LRU高速缓存模型，该模型将最近的200个已高速缓存对象在内存中至多保存24小时，并且每当调用插入、更新或删除已映射语句时高速缓存就会被清除。

代码清单9-5 LRU高速缓存模型样例

```
<cacheModel id="categoryCache" type="LRU">
  <flushInterval hours="24"/>
  <flushOnExecute statement="insert"/>
  <flushOnExecute statement="update"/>
  <flushOnExecute statement="delete"/>
  <property name="size" value="200"/>
</cacheModel>
```

LRU高速缓存对于那些数据的不同子集都用在某段时间内的应用程序非常有用。

9.5.3 FIFO

FIFO高速缓存模型采用先进先出的管理策略。FIFO是一种基于时间的策略，它总是将最老的已高速缓存对象先移除。只有当高速缓存超过大小限制这个约束条件时，才会废弃高速缓存中最老的对象。大小限制定义了高速缓存中最多能够包含的对象数目。再一次强调，一定要避免把大的内存消耗对象放置在此类高速缓存中，否则内存将很快耗尽。

由于FIFO采用的是基于时间的策略，因此这种高速缓存非常适合于放置那些初放入时使用频率较高、随着时间流逝访问频率就会逐渐降低（但仍有可能被访问）的对象。这种类型的高速缓存对于那些强调时效性的报表应用程序非常有用。例如你要报告股票价格，这些价格在当前时刻是非常重要的，但其重要性将会随着时间的流逝而逐渐降低。

对于FIFO高速缓存类型来说，使用<property>标签时唯一能被指定的特性就是size，它用于指定能够保存在高速缓存中的对象的最大数目。

代码清单9-6展示了一个简单的FIFO高速缓存模型，该模型将最近的1 000个已高速缓存的对象在内存中至多保存24小时，并且每当调用插入、更新或删除已映射语句时高速缓存就会被清除。

代码清单9-6 FIFO高速缓存模型样例

```
<cacheModel id="categoryCache" type="FIFO">
  <flushInterval hours="24"/>
  <flushOnExecute statement="insert"/>
  <flushOnExecute statement="update"/>
```

```
<flushOnExecute statement="delete"/>
<property name="size" value="1000"/>
</cacheModel>
```

FIFO高速缓存模型适用于那些数据具有时效性的应用程序，例如购物车应用程序。

9.5.4 OSCACHE

OSCACHE高速缓存模型采用OpenSymphony公司的产品——OSCache 2.0 (www.opensymphony.com/oscache/)。OSCache是一个非常健壮的高速缓存框架，它可以提供很多同iBATIS在其高速缓存模型中所提供的高速缓存策略一致的策略。使用OSCACHE高速缓存模型，你的应用程序就存在对OSCache JAR文件的依赖了。此时，需要将OSCache JAR包含在工程之中。这种高速缓存的配置方式与标准OSCache安装时的配置方式是一样的。这就意味着，你将需要把oscache.properties文件放在类路径的根目录下，以便OSCache读取。有关如何安装和配置OSCache的更多信息，请访问在线文档，网址为www.opensymphony.com/oscache/documentation.action。代码清单9-7给出的示例显示了OSCACHE高速缓存模型的外观。

代码清单9-7 OSCACHE高速缓存模型样例

```
<cacheModel id="categoryCache" type="OSCACHE">
  <flushInterval hours="24"/>
  <flushOnExecute statement="insert"/>
  <flushOnExecute statement="update"/>
  <flushOnExecute statement="delete"/>
</cacheModel>
```

9.5.5 你自己的高速缓存模型

之前曾提到高速缓存模型实际上是框架的可插入组件，你可能会猜想应该如何创建一个属于自己的高速缓存模型，或者你可能只是好奇应该如何做。

只需要记住两点。首先，iBATIS所提供4种类型的高速缓存模型实际上都是com.ibatis.sqlmap.engine.cache.CacheController接口的具体实现。其次，它们的名称实际上只是映射到这些实现的全限定名的别名。

现在你已经有了高速缓存模型，下面看看应该如何使用这些高速缓存模型，让它们恰如其分地为我们工作。

9.6 确定高速缓存策略

在确定高速缓存策略前，首先必须弄清楚需求。iBATIS为数据访问层提供了之前提到的那一组高速缓存策略。这些高速缓存策略可以（并且很可能将）在你的高速缓存策略中起作用，但是它们可能并不能面面俱到地满足你的高速缓存需求。如果要找一个面面俱到的高速缓存策略，需要考虑很多东西，有关此问题的讨论超出了本书的范围。然而，确定iBATIS高速缓存能在你那“面面俱到”的高速缓存策略中起到怎样的作用，这是十分重要的。

在数据访问层上高速缓存数据时, 可以将那些需要处理的数据库分为专用数据库 (dedicated database) 和共享数据库 (shared database)。对于专用数据库, 所有对该数据库的访问都将通过你正在开发的应用程序进行。而对于共享数据库, 可能会有多个应用程序访问此数据库, 并且这些应用程序可以同时更改其中的数据。

如果应用程序所连接的数据库同时被很多的其他应用程序所访问并且允许它们修改自身数据, 就应尽量避免高速缓存其中的数据。如果底层数据已经过期很久, 那么使用 `<flushOnExecute>` 就不是一个有效的高速缓存策略, 因为其他应用程序可能已经更改了你的数据库。不过你仍然可以利用高速缓存策略来对那些更静态的、不会对你读/写数据造成任何影响的只读数据进行高速缓存。此类例子包括: 不会更改的基于时间的报表数据, 购物车页面的数据, 以及静态的下拉列表中的数据。

另一方面, 如果你要高速缓存的数据库只能通过一个应用程序来访问, 那么就可以更大胆地使用 iBATIS 高速缓存了。在这样一个专用应用程序 (dedicated application) 中, `<flushOnExecute>` 就变得更有价值了。因为我们知道哪些已映射语句的执行会导致高速缓存数据变得无效, 并因此影响你的清除策略。

假定你需要能够更细粒度地控制高速缓存中的某些特定项的释放时间, 或者你需要一个分布式高速缓存。iBATIS 自身并没有提供这种方式的高速缓存, 但是它同 OSCACHE 高速缓存类型 (见 9.5.4 节) 相结合就可以提供这类功能, 且很健壮。然而, 此处也要遵循和前面所述的同样的规则。即当你知道数据库只能被一个应用程序访问时, 就可以更大胆地使用高速缓存。但是当应用程序并不是数据库的专用入口时, 就必须谨慎地使用高速缓存。

下面的几节将通过案例研究的方式来讨论高速缓存使用, 并给出一些代码片段来说明如何实现它们。

9.6.1 高速缓存只读的长效数据

通常只读的长效数据是我们高速缓存的理想数据。因为此类数据不经常发生改变, 所以非常易于高速缓存。我们可以把此类数据放在高速缓存中, 然后根本不用管它们, 直到定期清除到来或者某个已映射语句的执行触发其清除。下面回顾一下我们的购物车应用程序, 然后逐步地创建一个只读高速缓存。

回顾本书之前章节中所给出的购物车类别, 我们将在此处再次使用这个类。当用户选择了某个类别之后, 与其相关的子类别也会显示出来。由于这些类别数据会被经常访问, 但却不常变化, 因此非常适合为它们使用长效的只读高速缓存。

当思考如何设置类别高速缓存时, 一定要考虑其结果将被如何查询, 以及哪种高速缓存模型最适合该数据的访问模式。对于高速缓存子类别列表的情况, 用户通常会根据相关的父类别来查询于类别列表。用 SQL 术语来表达此意思就是, `WHERE` 子句将会基于 `parentCategoryId` 和某个输入参数的相等性比较。另一个需要考虑的问题就是, 高速缓存应该多长时间清除一次, 以及应

该使用哪一种高速缓存策略。在此例中，通常用户和某些类别的交互会比其他的多，因此，我们将选择LRU策略（见代码清单9-8）。该高速缓存策略将保存那些在近期被经常访问的数据，而废弃高速缓存中不常被访问的旧数据。

代码清单9-8 LRU高速缓存模型样例

```
<cacheModel id="categoryCache" type="LRU">
  <flushInterval hours="24"/>
  <flushOnExecute statement="insert"/>
  <flushOnExecute statement="update"/>
  <flushOnExecute statement="delete"/>
  <property name="size" value="50"/>
</cacheModel>
```

在代码清单9-8中，我们首先创建了一个高速缓存模型。并且将其类型属性指定为LRU。在建立将会使用高速缓存的查询已映射语句时，id属性提供了一个要引用的唯一标识。通过使用<flushInterval>标签，我们可以确保高速缓存中的数据不会超过24小时。<flushInterval>将会清除所有保存在标识为categoryCache的高速缓存中的结果。通过<flushOnExecute>标签，我们指定当所标识的插入、更新或删除已映射语句被调用时，也将触发对标识为categoryCache的高速缓存中所有结果的清除。最后，通过<property>标签和<size>特性，以设置高速缓存中最多能够保存的项数。一旦高速缓存中的数据超过50个，它就开始删除最近最少使用的数据。

在代码清单9-9中，查询已映射语句getChildCategories通过设置cacheModel属性将categoryCache指定为与其关联的高速缓存。当用户需要细读购物车应用程序中的某个类别的详细信息时，getChildCategories查询已映射语句就会被调用。这将使得该类别的所有子类别都被高速缓存。当高速缓存数据达到最大限度50时，高速缓存就会开始移除旧数据。这样就可以使经常被访问的子类别列表在高速缓存中保存更长的时间，并为用户提供更好的性能。如果某个管理员对类别列表执行了插入、更新或者删除操作，就会触发高速缓存的清除，并且使得已高速缓存的结果全部开始重建。这种清除和选择性删除的组合使得高速缓存中的子类别列表能始终保持与数据库的一致并且是近期最常使用的，同时还可以减低由于毫无必要的数据库访问而造成的性能损失。

代码清单9-9 使用了categoryCache的查询已映射语句

```
<select id="getChildCategories" parameterClass="Category"
resultClass="Category" cacheModel="categoryCache">
  SELECT *
  FROM category
  <dynamic prepend="WHERE ">
    <isNull property="categoryId">
      parentCategoryId IS NULL
    </isNull>
    <isNotNull property="categoryId">
      parentCategoryId=#categoryId:INTEGER:0#
    </isNotNull>
  </dynamic>
  ORDER BY sequence, title
</select>
```

LRU高速缓存的移除过程可以通过增大或减小size特性的值以高速缓存更多或更少的类别来进行调整。请记住LRU的目的就是仅仅高速缓存那些最近常被访问的数据。因此不要将size的设置得太大。如果将该值设置过大，实际上就把LRU高速缓存变成了STRONG类型的MEMORY高速缓存——这就会破坏LRU高速缓存的整个意图。

本节讨论了一种用于高速缓存长效只读数据的有效方式，下面考虑另一种你经常需要面对的情况：可读写数据的高速缓存。是啊，对这样的数据到底是高速缓存还是不高速缓存呢？

9.6.2 高速缓存可读写数据

假设你想要高速缓存的对象天性易变，就必须谨慎地考虑是否应该高速缓存。如果环境是高事务性（high-transaction）的，你就会发现高速缓存的负担往往过重从而实际上已经不起作用了。出现此种情况的原因是，试图高速缓存高事务性的数据就必然会要求频繁地清除高速缓存。过于频繁地清除高速缓存会给系统带来双重的负担。第一，每当有数据库请求时，应用程序就必须检查、清除并且重新填充高速缓存。因此，如果高速缓存经常性地被清除，就意味着应用程序必须经常进行数据库访问以获取新的结果。此时，要想提高应用程序的性能，与其费神费力为其创建高速缓存，还不如使用像索引（indexing）以及数据库表内常驻留（table pinning）这样的数据库技术。

高速缓存那些易变数据时需要小心谨慎，但高速缓存那些易变性相对稍弱一些的数据时，谨慎同样是很有意义的。在JGameStore应用程序中，我们在高速缓存产品时就发现了一个很好的示例。在该应用程序中，店面的管理员有时需要输入新的产品，或更新现有的产品，或将某些产品标记为已售出，或者其他一些相似的任务。此类动作将产生轻微的不稳定性。但由于这并不是一个高事务性的环境，因此高速缓存对于改善应用程序的整体性能还是起到了一定的作用。只要高速缓存能有时间建立起来，并在一段时间内为用户提供了更好的性能，就可以避免之前所描述的困境。

考察你想要高速缓存的数据的特性时，需要考虑如下因素：

- 产品的数量很可能非常大。
- 产品的具体信息具有易变性。
- 最经常被访问的产品每天都会变化，取决于消费者的习惯。

在此例中，我们决定使用WEAK内存高速缓存，因为它是一种限制性较少的高速缓存。它不像LRU那样，需要一定程度的预见性以确定应该高速缓存的结果的数量，WEAK内存高速缓存会在内存达到事先预定的极限之前，就决定哪些数据应该被保留，哪些可以被废弃。由于高速缓存使用了java.lang.ref.Reference实现来保存数据，因此它会根据内部分析来决定移除或保留结果。当把WEAK引用类型和MEMORY高速缓存结合使用时，数据库查询结果会首先被包装在WeakReference（java.lang.ref.WeakReference）中然后保存在高速缓存中。之后，垃圾收集器就可以根据其内部机制来处理这些被包装的结果了。

下面来配置<cacheModel>标签。正如大家设想的那样，cacheModel的类型属性被设置为

MEMORY。注意，我们是在一个读写环境中将<cacheModel>标签的readOnly属性设置为了true。此外，我们还将serialize属性设置为了false，以消除深复制的负担。这意味着，从高速缓存中检索出的对象可能会被改变。这是一种安全的方式，原因如下：第一，只有管理购物车的人才可能更改产品对象，实际进行购物的用户其动作将不会导致产品对象的更改；第二，无论何时发生了产品的更新，高速缓存都将被清除；最后，把reference-type属性设置为WEAK将保证产品在高速缓存中不会保存过长时间，因为它会根据垃圾收集器的判断来废弃这些产品数据。代码清单9-10给出了我们的高速缓存模型配置示例。

代码清单9-10 productCache的高速缓存模型

```
<cacheModel id="productCache" type="MEMORY"
  readOnly="true" serialize="false">
  <flushOnExecute statement="Product.add" />
  <flushOnExecute statement="Product.edit" />
  <flushOnExecute statement="Product.remove" />
  <property name="reference-type" value="WEAK" />
</cacheModel>
```

现在就可以在一个查询已映射语句中使用这个定义好的<cacheModel>了。我们通过将productCache指定在cacheModel中，告诉getProductById查询已映射语句使用productCache高速缓存。无论何时应用程序调用getProductById查询，已高速缓存的产品对象都会根据productCache高速缓存模型的规范被检索。代码清单9-11给出了一个简单的示例，说明<select>语句可以如何利用已定义的<cacheModel>。

代码清单9-11 使用productCache的查询已映射语句

```
<select id="getProductById" resultClass="Product"
  parameterClass="Product" cacheModel="productCache">
  SELECT * FROM Product WHERE productId=#productId#
</select>
```

9.6.3 高速缓存旧的静态数据

最后一个案例有一点不太常见，但它却是一个很有意思的测试。此案例的情况发生在当你需要处理那些随着时间逐渐变得不重要的小型静态数据时。通常此类高速缓存同基于时间的分析有一定的关联。基于时间的分析的例子有：呼叫中心应用程序中的性能统计，或者提供以前几小时/天/月内的统计数据的股票行情自动收录器。总结这类数据的属性，我们可以说，它们随着时间的流逝而逐渐变得不重要。它不是一个大数据集；这些数据在开始时被频繁地访问，之后随着时间的流逝被访问的次数逐渐降低。这些数据并没有发生改变，或者说它们是“静态的”。

JGameStore应用程序并没有这类需求，但是我们将继续使用购物车应用程序来进行类比。假定我们需要收集关于每小时内顾客购买的最多5件产品的统计数据。然后我们将在购物车主页上使用该数据，以便告诉购物者每小时的热门商品是什么。也可以假定我们提供了一个下拉列表，允许用户选择之前的时间以便观察每个时段的最热门商品。一旦产品售出了，这些数据就不会发生改变。如果

在过去的一个小时之内, 5款最新的3D blockbuster售出了, 那么这个数据在将来仍然是真实的, 不会改变。考虑了这些简单的要求之后, 下面来看一下应该如何使用iBATIS高速缓存来确保性能增强。

考虑到这些数据随着时间的流逝而逐渐变得不再重要, 具有鲜明的时效性, FIFO就成了首选的高速缓存策略。我们将FIFO高速缓存描述为一个时效性高速缓存。任何保存在高速缓存中的数据都将随着高速缓存大小限制的超过而“年老淡出”。因此, 随着时间的流逝, 以及新的数据逐渐添加到高速缓存中, 那些不是最新的产品列表的访问次数将会变得越来越少。随着新的产品都被高速缓存, 那些数据最终将被废弃。假如我们将FIFO的高速缓存大小设置为24, 那么我们的列表将仅仅保存排名前24位的最畅销商品, 25位之后的所有“畅销”商品都将被废弃。下面讨论FIFO高速缓存是如何满足其他要求的。

由于产品的购买长期都是保持一致的, 我们就可以安全地假定它们是静态的。在FIFO高速缓存中保存静态的数据是很合适的。静态数据最终会从高速缓存中“年老淡出”, 因此并不需要经常清除, 也不会打乱数据“老化”的过程。只有当产品有更新时, 我们才需要清除高速缓存。而由于这种情况并不经常出现, 因此我们完全可以让一切顺其自然, 通过FIFO高速缓存来存储数据, 发生更新时自动清除。

我们每小时都要存储的5件产品应该具有较小的内存消耗。由于FIFO只有当高速缓存超过指定的大小时才废弃数据, 因此在高速缓存中保存大量消耗内存的对象是不合适的。当内存约束变得紧张时, FIFO高速缓存并不会废弃对象, 这就可能最终造成内存耗尽异常。不过在我们的产品列表例子中, 将完全不存在此种异常的可能性, 我们是非常安全的。

对于高速缓存, 高频率访问和低频率访问是必须考虑的问题。只要某组数据的访问次数有可能超过一次, 那么高速缓存这组数据就能带来实际的好处, 不论它是否真的被超过一次地访问。我们只需要关心如何微调高速缓存的大小。根据访问当前最畅销产品或者过去最畅销产品的人数, 就可以把高速缓存的大小设置为最恰当的值。

现在已经讨论了各种需求, 以及FIFO高速缓存是如何满足这些需求的, 下面讨论高速缓存模型的配置。我们的高速缓存模型非常简单。如代码清单9-12所示, 我们想要为已售出的产品显示正确的产品信息。这就要求必须首先设置恰当的<flushOnExecute>标签, 以便在产品数据改变时清除hotProductsCache。只需要指定Product.update或Product.delete来清除高速缓存即可。此处并没有包含插入语句, 因为我们并不关心不存在的产品。如果一个产品被添加后成为了畅销商品, 完全可以很容易地把它添加到hotProductsCache中。因此, 只有当产品被更新或者被删除时, 才需要考虑清除的问题。此外, 我们不会想要高速缓存已经不出售的(例如, 被删除了的)产品, 也不会想要显示已经更新过价格的产品的新价格。

代码清单9-12 hotProductsCache的高速缓存模型和已映射语句

```
<cacheModel id="hotProductCache" type="FIFO">
  <flushOnExecute statement="Product.update"/>
  <flushOnExecute statement="Product.delete"/>
</cacheModel>
```

```
<property name="size" value="12"/>
</cacheModel>

<select
  id="getPopularProductsByPurchaseDate"
  parameterClass="Product"
  resultClass="Product" cacheModel="hotProductsCache">
  SELECT count(productId) countNum, productId
  FROM productPurchase
  WHERE
    purchaseDate
      BETWEEN
        #startDate#
      AND
        #endDate#
  GROUP BY productId
  ORDER BY countNum DESC
  LIMIT 5
</select>
```

设置好高速缓存清除配置之后，继续设置FIFO高速缓存的大小特性。把size特性设置为12，可以允许高速缓存最多保存12个结果（见代码清单9-12）。一旦高速缓存中已经有了12个结果，最老的数据就将被废弃，而最新的数据将被添加到高速缓存的开始处。由于我们使用的SQL语句每隔一个小时甚至更长时间才给高速缓存提供一个新的结果，因此高速缓存应该调整得恰到好处以便数据恰当地“年老淡出”。当旧的结果被废弃时，新的结果应该总是占据高速缓存队列的前端。

9.7 小结

iBatis为持久层高速缓存选项提供了丰富的高速缓存选择。做出正确的选择需要谨慎的思考。必须牢记的是，iBatis高速缓存是基于数据值的（相对于OID而言），并且它主要用于处理持久层上的高速缓存。

如果默认的iBatis高速缓存选项不能满足需求，可以扩展iBatis（见第12章）以便提供一个能够满足需求的高速缓存。本章中的示例可以帮助你初步提高性能，经验往往是最好的指导。花一些时间来了解不同类型的高速缓存以及它们的各种选项，熟悉每一类高速缓存各自的优缺点，这些知识将帮助你更好地确定哪种选择可以满足你特定的要求。

第 10 章

iBatis数据访问对象

本章内容

- DAO基本原理
- 配置
- SQLMapDAO示例

应用程序经常需要访问来自多种数据源的数据，例如关系数据库、文件系统、目录服务（directory service）、Web服务，以及其他提供程序。每种数据存储都具有各自不同的API以便访问其底层的存储机制，也都有一个完整的特性集合。

DAO（Data Access Object，数据访问对象）模式用于隐藏这些API各自的独特实现。而为应用程序的开发人员提供了一套简单并且公共的API，因此数据的使用者就不必担心数据访问API的复杂性了。

iBatis的DAO子项目就是J2EE中同名核心模式的一个简单实现。

10.1 隐藏实现细节

面向对象编程的重要原则之一就是，将实现和它的公共接口分离并且封装起来。DAO模式所做的正是这样一件事情。在深入讨论DAO模式之前，先看一下图10-1中所示的DAO模式。

如果认为图10-1看上去不像DAO，反而更像一个JDBC，那么其实你是半对半错。它确实是一个JDBC，但是Java中的JDBC API正是DAO模式应用于实践的一个很好的例子。

图的顶层是一个“工厂”（DataSource接口），它用于创建实现了Connection接口的对象的实例。一旦你获得了某个Connection，就可以通过它获得PreparedStatement对象或者CallableStatement对象，它们接下来又可以提供ResultSet对象。你根本不需要知道DataSource是如何创建连接的，而连接又是如何创建PreparedStatement的，以及PreparedStatement是如何将参数绑定到查询上的，甚至也不需要知道ResultSet是如何被构造的。所有的细节都是无关紧要的，你只需要知道：调用Connection的prepareStatement()方法时，你

就可以得到一个PreparedStatement对象。

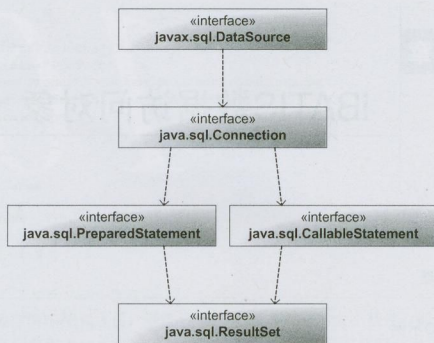


图10-1 一个简化的DAO（或者说JDBC）

10.1.1 为何要分离

通过将数据访问的具体实现和数据访问接口分离开，就可以为异质（heterogeneous）数据源提供一套同源（homogenous）的接口。应用程序可以访问多种数据库中的数据，虽然每种数据库都具有其特定的底层数据访问机制，但应用程序中使用的是一套一致的数据访问API。

JDBC API仅仅是DAO模式的一种类型。JDBC定义了一组接口，数据库开发商负责实现这组接口，这样在绝大多数情况下应用程序的底层数据库就可相互替换。在有些情况下，开发商认为接口没有为他们提供让他们想做做的事情的能力，他们就会在接口之外进行工作。

为了使你了解DAO模式在多大程度上简化了JDBC API的使用（是的，尽管JDBC真的很复杂，但我们确实还是想要说“简化了”），来看一下JDBC驱动程序的实现。例如，某个主要数据库开发商为实现图10-1中所示的5个接口，首先实现了9个该开发商自己定义的接口，然后才以此为为基础实现了这5个接口（这并不是在计算真实的实现类之外的类或接口的数量）。实际上作者之一确实为此画了一个UML图，并打算将它放在此处作为示例，但是本章已经太长了，这样我们又不得不添一页纸，所以就算了。是的，它就是那么复杂。

DAO模式是使得类似于iBATIS的SQL映射和ORM工具这样的数据库工具成为可能的原因之一。因为几乎所有的数据库开发商都非常精确地实现了JDBC API，所以可以编写直接处理接口的工具，而不需要处理底层的具体实现。大多数情况下，这些工具都可以处理任何开发商的JDBC实现，你只需要在创建DataSource时引用该开发商的软件即可——从此以后，你就只需要处理公共API。

这也是iBatis DAO的目的之一：帮助你提供一组接口给应用程序，以便隐藏数据访问的底层实现。因此，如果你有一个使用DAO模式的基于Hibernate的应用程序，就可以用基于SQLMap的实现或者基于JDBC的实现去代替它，而不需要重新编写整个应用程序。事实上，所有需要更改的地方就是应用程序所使用的接口实现。只要DAO接口被正确地实现了，应用程序就会如你所料地继续正常工作。

作为规则，DAO不允许暴露来自java.sql或javax.sql包的任何接口或对象。这意味着DAO是应用程序中实际进行数据源整合的唯一的层，而访问DAO层的所有其他层都无需考虑底端数据源的细节了。这还意味着DAO模式除了能够改变数据访问机制（例如，SqlMap、Hibernate和JDBC），还可以允许你以相似的方式更改数据源。由于接口是以一种数据源不可知的方式创建的，因此应用程序根本不需要知道数据到底是来自Oracle还是PostgreSQL——甚至是来自一个不是基于SQL的数据库。应用程序所必须处理的一切就是JavaBean。至于这些bean来自哪里，对于应用程序来说无关紧要。

进行这种分离的另一个好处就是，它使得测试变得更加容易，因为你再也不用去处理特定于DAO所使用的数据库访问方法的接口——而是只需要处理更加常见的对象，如List和Map，以及针对你的应用程序的bean。

10.1.2 一个简单示例

让我们从一个简单示例开始研究如何配置及使用iBatis DAO。在给出这个例子之前，首先来看一下图10-2，这就是我们将要配置的DAO。

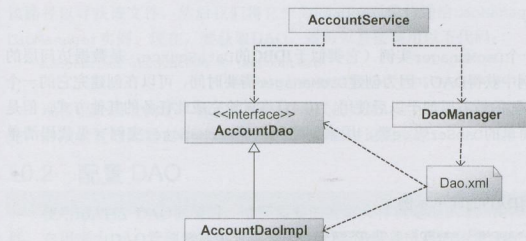


图10-2 一个更简单的DAO（确实非常简单）

这个DAO（它比JDBC示例要简单得多）是由一个接口（AccountDao）和相应的实现（AccountDaoImpl）组成的。另外两个类展示了接口的一种用法，而DaoManager类是iBatis用来创建DAO的工厂类。DaoManager类是通过读取Dao.xml配置文件来完成配置。

1. Dao.xml配置文件

要配置DaoManager，首先必须从一个XML配置文件开始，该文件通常被命名为dao.xml，

它包含了告诉 iBATIS 如何构造 DAO 所需的信息。代码清单 10-1 给出了一个用于配置基于 SQLMap 的 DAO 层的 XML 配置文件示例。

代码清单 10-1 一个简单的 dao.xml 示例

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE daoConfig
PUBLIC
"-//ibatis.apache.org/DTD DAO Configuration 2.0//EN"
"http://ibatis.apache.org/dtd/dao-2.dtd">
<daoConfig>
  <context id="example">
    <transactionManager type="SQLMAP">
      <property
        name="SqlMapConfigResource"
        value="examples/SqlMapConfig.xml"/>
    </transactionManager>
    <dao
      interface="examples.dao.AccountDao"
      implementation="examples.dao.impl.AccountDao"/>
    </context>
  </daoConfig>
```

① DAO 上下文
② 事务管理器
③ 配置中定义的唯一一个 DAO

代码清单 10-1 定义了一个名为 example 的 DAO 上下文，它将使用 SQLMaps 来管理事务（在第 7 章中已经介绍过）并且其中只包含一个 Account DAO。10.2 节将更详细地讨论此文件的内容，因此即使你现在并不完全清楚，也没有关系。

2. 创建 DaoManager

接下来，需要创建一个 DaoManager 实例（它类似于 JDBC 的 DataSource，是数据访问层的起始点），可以从这个实例中获得 DAO。因为创建 DaoManager 需要时间，可以在创建完它的一个实例后将它保存在一个已知的地方以便于以后使用。10.4 节将讨论完成此任务的其他方式，但是我们现在仅仅创建一个简单的 DaoService 类，用来创建和存储 DaoManager 实例（见代码清单 10-2）。

代码清单 10-2 一个使用 DAO 的简单示例

```
package org.apache.mapper2.examples.chapter10.dao;

import com.ibatis.dao.client.DaoManager;
import com.ibatis.dao.client.DaoManagerBuilder;
import com.ibatis.common.resources.Resources;
import java.io.Reader;
import java.io.IOException;

public class DaoService {
  private static DaoManager daoManager;

  public static synchronized DaoManager getDaoManager() {
```



```
String daoXmlResource = "dao.xml"; <1 指定配置文件的位置
Reader reader;
if (null == dacManager) {
    try {
        reader =
            Resources.getResourceAsReader(daoXmlResource); <2 为构建器获取一个阅读器
        dacManager =
            DaoManagerBuilder.buildDaoManager(reader); <3 构造DAO管理器
        return dacManager;
    } catch (IOException e) {
        throw new RuntimeException(
            "Unable to create DAO manager.", e);
    }
} else {
    return dacManager;
}

public static Dao getDao(Class interfaceClass){
    return getDaoManager().getDao(interfaceClass);
}
}
```

来简单看一下以上的代码，因为其中确实有一部分代码稍候将被证明很重要。最重要的是 daoXmlResource 变量^①，它包含了我们之前在代码清单10-1中所见到的 dao.xml 文件的位置。这非常重要，因为它并不是一个路径，而是位于类路径上的一个资源的位置。例如，如果你正在开发一个Web应用程序，那么该资源就应该在WEB-INF/classes目录下。Resources类^②将搜索该路径以寻找该文件，然后我们将它作为Reader对象传递给DaoManagerBuilder^③以获取一个DaoManager实例。现在，要获取DAO，就可以直接调用以下代码：

```
AccountDao accountDao = (AccountDao)
    DaoService.getDao(AccountDao.class);
```

现在已经通过例子了解了使用DAO模式的好处，你可能想要知道应该如何使用这种模式。首先，必须设置好配置文件，这就是接下来将要讨论的主题。

10.2 配置 DAO

使用iBATIS DAO框架时，唯一需要的配置文件就是dao.xml文件。这是一个非常简单的文件，它用来为DAO管理器提供管理有关DAO类的事务所需的信息，并且告诉DAO管理器如何为DAO接口提供DAO实现。

首先来看一下有哪些可用的配置元素，然后我们将深入研究如何使用这些配置元素来解决常见的问题。

10.2.1 <properties>元素

<properties>元素和它在SqlMapConfig.xml文件中的使用方式是一样的。它用来指定一个特性文件，该文件中所列出的所有特性都可以通过\${name}语法使用在DAO层的配置文件中。

这种方式非常有用，尤其是当你针对开发和产品分别拥有不同的服务器（甚至还有可能针对不同阶段和测试也分别拥有不同的服务器）时。此时，你就可以在同一个配置文件中完成对所有DAO类的配置，而把那些由于环境不同而相异的项放在特性文件中。如此一来，当你需要在某个特定环境中部署应用程序时，所需的全部改变就是这个特性文件。

10.2.2 <context>元素

一个DAO上下文就是指一组相关的DAO配置信息和一组DAO实现：

```
<context id="example">
```

通常，上下文只和数据源相关联，例如关系数据库或者平板文件（flat file）。通过配置多个上下文，可以很容易地将对多个数据库的访问配置集中起来。

10.3节将探讨使用上下文来创建多个采用不同数据访问模型的DAO组——其中一个使用的是SQL映射（这是当然的！），另一个使用Hibernate，而最后一个则直接使用JDBC。

每个上下文都有各自的事务管理器以及一组DAO实现。在接下来的两节中，你将学习如何配置上下文中的每一个项。

10.2.3 <transactionManager>元素

顾名思义，事务管理器用来为DAO类管理事务（有关事务详见第7章）。<transactionManager>元素提供一个DaoTransactionManager实现的名字。iBATIS DAO框架提供了7种不同的事务管理器，它们可与不同的数据访问工具协同工作，如表10-1所示。

表10-1 iBATIS DAO框架中可用的事务管理器

类型别名	事务管理器/特性	注 释
EXTERNAL	ExternalDaoTransactionManager	一个什么都不做的事务管理器，让你在iBATIS DAO框架之外管理你自己的事务
HIBERNATE	HibernateDaoTransactionManager	委托给Hibernate的事务管理功能
JDBC	JdbcDaoTransactionManager * DataSource * JDBC.Driver * JDBC.ConnectionURL * JDBC.Username * JDBC.Password * JDBC.DefaultAutoCommit	使用JDBC通过DataSource API来提供数据库连接池服务。支持3种DataSource实现：SIMPLE，DBC和JNDI SIMPLE是iBATIS的SimpleDataSource接口的实现，如果希望获得最小的开销和最少的依赖性，那么SIMPLE是一个理想的选择 DBC是一个使用了Jakarta的DBCP DataSource的实现 最后，JNDI是一个从JNDI目录中获取DataSource引用的实现。这是一个最通用和灵活的配置，因为它允许通过应用程序服务器来集中配置你的应用程序
JTA	JtaDaoTransactionManager * DBJndiContext * UserTransaction	通过JTA（Java Transaction Architecture）API来管理事务 需要通过JNDI来获取DataSource实现，并且User-Transaction实例也要通过JNDI来访问

(续)

类型别名	事务管理器/特性	注 释
OJB	OjbBrokerTransactionManager	委托给OJB的事务管理功能
SQLMAP	SqlMapDaoTransactionManager	委托给SQL映射的事务管理功能
TOPLINK	ToplinkDaoTransactionManager	委托给TOPLINK的事务管理功能

好，现在你已经知道有哪些选项了，下面让我们进一步讨论每一种选项都有哪些独特的地方。

1. EXTERNAL事务管理器

EXTERNAL事务管理器最易于配置（它没有特性需要设置，任何被传递给它的数据都将被忽略），但是却很可能是最难使用的，因为你必须100%地为应用程序中的所有事务管理负责。

2. HIBERNATE事务管理器

HIBERNATE事务管理器也很容易配置。它接受传递给它的所有特性，并且直接把它们传递给Hibernate会话工厂。另外，在传递给HIBERNATE事务管理器的所有特性中，任何名字以class.开头的特性都被认为是应该受Hibernate管理的类，并将被添加到用于构建会话工厂的配置对象中。同样地，传递进来的任何名字以map.开头的特性都被认为是映射文件，也将被添加到配置对象中。

3. JDBC事务管理器

JDBC事务管理器可能是最难配置的。对于这种事务管理器来说，DataSource特性是必需的，并且该特性取值必须是以下3种中的一种：SIMPLE、DBCP或JNDI。

● SIMPLE数据源

SIMPLE数据源是iBATIS的SimpleDataSource接口的一个实现，如果你希望获得最小的开销和最少的依赖性，那么SIMPLE是一个理想的选择。它有5个必需的特性：

- JDBC.Driver——这是在DAO上下文中用来管理事务的JDBC驱动程序的全限定名。
- JDBC.ConnectionURL——这是当前DAO上下文用于连接到数据库的JDBCURL。
- JDBC.Username——这是当前DAO上下文用于连接到数据库的用户名。
- JDBC.Password——这是当前DAO上下文用于连接到数据库的用户密码。
- JDBC.DefaultAutoCommit——如果该值为true（或者是Java中任意可被Boolean类解释为true的表达式），那么在这个DAO上下文中，这个数据源返回的连接的autoCommit特性将被设置为true。

除了以上这些必需属性之外，还有8个可选特性也可以用来配置连接池：

- Pool.MaximumActiveConnections——这是池中一次能够同时拥有的活跃连接的数目。该特性的默认值为10个连接。
- Pool.MaximumIdleConnections——该特性指定了池中一次能拥有的空闲连接的数目。该特性的默认值为5个连接。

- `Pool.MaximumCheckoutTime`——它指一个连接在被赋给另一个请求之前能够被保留的最大毫秒数。该特性的默认值为20 000毫秒，即20秒。
- `Pool.TimeToWait`——它代表了发出请求却无法获得连接时需要等待的毫秒数。该特性的默认值为20 000毫秒，即20秒。
- `Pool.PingEnabled`——如果该值为`true`（或者是Java中任意可被`Boolean`类解释为`true`的表达式），那么当连接被确定具有失效的危险时，它们将使用在`Pool.PingQuery`特性中所定义的查询来测试。该特性的默认值为`false`，这意味着连接在使用之前决不会被查验。以下的3个特性是用来确定连接的查验行为的。
- `Pool.PingQuery`——这是当连接需要被测试以便确定它是否仍然有效时而执行的查询。请确保将该查询构造为一个可以快速返回的语句。例如，在Oracle中，类似于`select 0 from dual`的语句就可以很快地返回结果。
- `Pool.PingConnectionsOlderThan`——该值代表了连接被认为有失效危险的时间间隔（以毫秒为单位）。该特性的默认值为0，这意味着如果查验是被启用的，那么每当连接被使用时它都会被检查。在一个高事务性环境中，这将对性能造成严重的影响。
- `Pool.PingConnectionsNotUsedFor`——这是考虑连接具有失效的“危险”之前能够允许它空闲的毫秒数。

● DBCP数据源

DBCP数据源是对Jakarta CDBCP（Commons Database Connection Pooling，公共数据库连接池）工程的数据源实现的一个包装，其目的是允许你以增加一条依赖性为代价使用一个更强健的实现。这种事务管理器的特性要给予不同的处理，根据哪些特性的当前可用性。设置DBCP事务管理器的老方式（这种方式目前仍然被支持）就是设置以下这8个特性：

- `JDBC.Driver`——指定DAO上下文所用的JDBC驱动程序。
- `JDBC.ConnectionURL`——这是要连接到当前DAO上下文数据库所要使用的JDBCURL。
- `JDBC.Username`——这是连接到数据库时所要使用的用户名。
- `JDBC.Password`——这是连接到数据库时所要使用的用户密码。
- `Pool.ValidationQuery`——用于验证数据库连接的查询。
- `Pool.MaximumActiveConnections`——该特性指定了要置于连接池中的活跃连接的最大数目。
- `Pool.MaximumIdleConnections`——该特性指定了要置于连接池中的空闲连接的最大数目。
- `Pool.MaximumWait`——该特性指定了在放弃某个连接之前的最大等待时间（以毫秒为单位）。

配置DBCP数据源的新方式要更加灵活，因为它将其简单地看作一个bean，因此通过获取/设置方法暴露出来的所有属性都可以在iBATIS中使用。例如，要设置数据源的`driverClassName`，可以编写以下代码：

```
<property
  name="driverClassName"
  value="com.mysql.jdbc.Driver"/>
```

说明 想了解更多有关使用和配置DBC数据源的信息，请访问Jakarta工程的官方网站，网址为：
<http://jakarta.apache.org/commons/dbcp/>。

● JNDI数据源

JNDI数据源的作用就是它允许利用你的应用程序容器可能提供的任何JNDI上下文。它可能是设置起来最为简单的数据源了。它仅有一个必需的特性，即DBJndiContext，用来提供包含数据源的上下文名称。

该数据源也允许你在配置文件中通过在其包含的任意属性名前加上前缀context.，从而把其他特性传递给InitialContext构造器。例如，为了传递名为someProperty的特性，就可以使用以下的语法：

```
<property name="context.someProperty" value="someValue"/>
```

4. JTA事务管理器

JTA (Java Transaction API) 事务管理器允许你为多个数据库提供分布式事务管理。这意味着你可以在多个数据库中进行更改提交或者更改回退，这种操作就像在单个数据库中那样简单。

在这个事务管理器中，只有两个特性需要配置，因为大量的配置工作都已经在JNDI中完成了。第一个特性DBJndiContext，用于指定一个包含某数据源的上下文的名称，该数据源将被事务管理器使用。另一个需要配置的特性是UserTransaction，用于提供包含用户事务的上下文名称。

5. OJB事务管理器

OJB (ObjectRelationalBridge) 是为使用关系数据库的Java对象提供持久化帮助的一种ORM工具。OJB事务管理器是对OJB所提供的事务管理接口的一个包装。

OJB事务管理器的所有配置的设置与iBATIS DAO无关，不受其任何影响。

说明 想了解更多有关OJB工具的信息，并且了解如何配置OJB事务管理器，请访问网页<http://db.apache.org/ojb/>。

6. SQLMAP事务管理器

SQLMAP可能是使用iBATIS DAO时最常用的一种事务管理器。SQLMAP事务管理器需要设置SqlMapConfigURL特性或者SqlMapConfigResource特性。它和SQLMap使用同样的事务管理器。

SqlMapConfigURL特性取值应该是一个字符串，要求java.net.URL类能够解析此字符串并

且从http:和file:协议中检索资源。SqlMapConfigResource特性用于引用一个存在在当前类路径上的资源。

7. TOPLINK事务管理器

Oracle 的产品TopLink是另一个O/RM工具。TOPLINK事务管理器的唯一必需的特性是session.name, 它用来获取将要用在DAO上下文中的会话。

8. 使用你自己的或者另外的事务管理器

除了以上这些事务管理器实现之外, DaoTransactionManager是一个你可以自己实现的接口, 并且你可以通过在事务管理器配置元素的type属性中提供实现的全限定名, 从而将其插入到DAO配置中。在<transactionManager>元素的内容体中所列出的任何特性都会被传递给该事务管理器类的configure()方法:

```
<transactionManager
  type="com.mycompany.MyNiftyTransactionManager">
  <property name="someProp" value="aValue"/>
  <property name="someOtherProp" value="anotherValue"/>
</transactionManager>
```

在以上这个例子中, iBATIS DAO框架将创建MyNiftyTransactionManager类的一个实例, 并且分别把名为someProp和someOtherProp的Properties对象传递给它。我们将在第11章中深入剖析一个既有的DaoTransactionManager实现。

10.2.4 DAO 元素

一旦事务管理器被选定并且配置好之后, 就可以将DAO元素添加到DAO上下文中, 用于定义应用程序中可用的DAO接口和相应实现。

<dao>元素仅有两个属性: interface和implementation。

说明 由于interface属性名可用于标识DAO实现, 因此使用iBATIS DAO时接口实际上并不是必需的。在之前的例子中, 这两个属性(interface和implementation)可取值为同一个实现类的全限定名。尽管这可能看上去是一种减少代码量的简单方式(不再需要特意写一个接口了), 但是强烈建议你不要这么做, 因为它从本质上消除了DAO层所带来的价值——即接口和实现的分离。至于减少代码, 几乎所有的IDE现在都提供了重构工具, 可以帮助你从一个实现中提取出接口, 这意味着你可以在不创建接口的前提下先编写并测试你的DAO实现, 然后只需要点击几下鼠标就可以创建出接口。

interface属性用来在DAO映射中标识DAO, 它通常以如下方式使用:

```
<dao interface="com.mycompany.system.dao.AccountDao"
  implementation=
    "com.mycompany.system.dao.impl.AccountDaoImpl"/>
```


下面来看一个示例。设想有一个类似于图10-2中描述的关系。在图10-2中，有一个AccountDao接口和一个用来实现该接口的AccountDaoImpl类。要通过DaoManager获取DAO，应该使用以下代码：

```
AccountDao accountDao = (AccountDao)
    daoManager.getDao(AccountDao.class);
```

在此行代码中，我们声明了AccountDao变量，并利用接口名从DaoManager实例中请求它，然后将该实例强制转换为AccountDao接口（因为DAO管理器仅仅返回Object）。

在DAO以前的版本中，允许输入一个字符串，而不是一个接口类。但是在2.x版中，此功能被撤销了，因为这样可以消除一个可能的出错点。通过强制使用类名来标识DAO实现，就可以确保在Java环境中不会出现拼写错误，因为如果接口名称被拼错，代码将不会通过编译。早点发现错误总是一件好事。

好，现在你已经完全深入地理解了能够使用iBATIC DAO框架来做些什么，下面就可以开始讨论其他更高级的用法了。

10.3 配置技巧

虽然DAO配置表面上看起来非常简单，但是它仍然能提供极大的灵活性。通过创造性地配置DAO管理器，你可以用一些相当巧妙的方式来解决一些公共问题。下面就来讨论一些配置技巧。

10.3.1 多个服务器

如前所述，大部分开发室都同时拥有大量服务器可分别用于开发、QC测试、UA测试以及产品的环境，这是非常普遍的。

在这种情况下，将特定于环境的信息从dao.xml文件中移出并把它们放在一个外部文件中，就会非常有用。properties元素就是用来完成此类任务的。代码清单10-3给出了一个样例dao.xml文件，该文件就使用了这种技术，使得与JDBC设置都放在了dao.xml文件之外。

代码清单10-3 dao.xml文件样例，其中JDBC设置通过<properties />元素插入

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE daoConfig
PUBLIC
"-//ibatis.apache.org//DTD DAO Configuration 2.0//EN"
"http://ibatis.apache.org/dtd/dao-2.dtd">
<daoConfig>
  <properties resource="server.properties"/>
  <context>
    <transactionManager type="JDBC">
      <property name="DataSource" value="SIMPLE"/>
      <property name="JDBC.Driver" value="${jdbcDriver}" />
      <property name="JDBC.ConnectionURL"
        value="${jdbcUrl}" />
```

```
<property name="JDBC.Username" value="${jdbcUser}" />
<property name="JDBC.Password"
           value="${jdbcPassword}" />
<property name="JDBC.DefaultAutoCommit"
           value="${jdbcAutoCommit}" />
</transactionManager>
<dao interface="..." implementation="..." />
</context>
</daoConfig>
```

在此例中，所有的属性值都被保存在名为 `server.properties` 的文件中，该文件将从类路径的根目录下被加载。

这是一种我们喜欢使用的方式，因为所有的文件都可以处于版本控制之下，并且特性文件的不同版本可以根据环境来命名（例如，`server-production.properties`、`server-user.properties`等），因此构造程序可以自动地把正确的版本复制到正确的位置上。

这种方式在更灵敏的环境中也很好用，在此种环境下对配置中属性的设置过程被认为更加安全，因为它们没有处在版本控制之下。此外，这样的环境还使得手动配置更加简单，因为会随着环境而改变的配置文件总是同一个文件。

10.3.2 多种数据库方言

针对不同的数据库平台，你可能分别需要不同的代码（例如，使用没有存储过程的MySQL，或者具有存储过程的Oracle），如果要使用DAO模式来支持这种情况，那么处理方式与我们之前处理JDBC设置时有点相似，只要把包名也作为特性文件的一部分（见代码清单10-4）即可。

代码清单10-4 使用<properties />元素插入实现信息的dao.xml文件

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE daoConfig
PUBLIC
  "-//ibatis.apache.org//DTD DAO Configuration 2.0//EN"
  "http://ibatis.apache.org/dtd/dao-2.dtd">
<daoConfig>
  <properties resource="config.properties"/>
  <context>
    <transactionManager type="JDBC">
      <property name="DataSource" value="SIMPLE"/>
      <property name="JDBC.Driver" value="${jdbcDriver}" />
      <property name="JDBC.ConnectionURL"
                value="${jdbcUrl}" />
      <property name="JDBC.Username" value="${jdbcUser}" />
      <property name="JDBC.Password"
                value="${jdbcPassword}" />
      <property name="JDBC.DefaultAutoCommit"
                value="${jdbcAutoCommit}" />
    </transactionManager>
    <dao interface="com.company.system.dao.AccountDao"
          implementation="${impl}.AccountDaoImpl"/>
  </context>
</daoConfig>
```

```
</context>
</daoConfig>
```

在代码清单10-4中，所有的服务器设置和数据访问实现都是在主配置文件之外设置的。

10.3.3 运行时配置更改

假如这还不够灵活，DAO管理器也可以使用在运行时确定并在创建DAO管理器时传入的特性来构建。

我们在10.1.1节所看到的传入运行时配置信息的代码其实还可以有第二种形式^①，它可能提供一种类似于代码清单10-5的方法。

代码清单10-5 创建一个具有运行时特性的DaoManager

```
public static DaoManager getDaoManager(Properties props)
{
    String daoXml = "/org/apache/mapper2/examples/Dao.xml";
    Reader reader;
    DaoManager localDaoManager;

    try {
        reader = Resources.getResourceAsReader(daoXml);
        localDaoManager =
            DaoManagerBuilder.buildDaoManager(reader, props);
    } catch (IOException e) {
        throw new RuntimeException(
            "Unable to create DAO manager.", e);
    }

    return localDaoManager;
}
```

代码清单10-5中给出的代码将创建一个动态配置的DAO管理器，其特性将在运行时被传入，而不是通常会被返回的共享特性^②。尽管这样可以提供更多的灵活性，它也要求该DAO管理器的使用者保存它的一个副本^③，而不是每次需要时都创建一次。

下面将探讨如何使用iBATIS DAO框架，并创建一些iBATIS DAO框架将为我们提供管理的DAO类。

10.4 基于 SQL Map 的 DAO 实现示例

DAO模式其实说穿了就是关于如何在接口后面隐藏实际的数据访问实现的，但是你仍然必须构造底层的实现。在本节中，我们为DAO接口创建了一个SQL Map实现。你将在第11章中学习

① 其实是指DaoManagerBuilder.buildDaoManager方法还有一个重载形式。——译者注

② 即在配置文件中硬编码的属性文件中指定的属性。——译者注

③ Singleton模式。——译者注

更多有关如何使用DAO模式的知识，在那里我们将使用不同的数据访问技术再次实现此接口：一个用Hibernate技术实现，另一个直接用JDBC实现。

在开始构建具体实现之前，先构建DAO接口（见代码清单10-6）。

代码清单10-6 DAO的接口

```
package org.apache.mapper2.examples.chapter10.dao;

import org.apache.mapper2.examples.bean.Account;
import org.apache.mapper2.examples.bean.IdDescription;

import java.util.List;
import java.util.Map;

public interface AccountDao {
    public void insert(Account account);
    public void update(Account account);
    public int delete(Account account);
    public int delete(Integer accountId);
    public List<Account> getAccountListByExample(
        Account account);
    public List<Map<String, Object>>
        getMapListByExample(Account account);
    public List<IdDescription>
        getIdDescriptionListByExample(Account account);
    public Account getId(Integer accountId);
    public Account getById(Account account);
}
```

这对于账户表来说是一个合理的接口——它包括了所有基本的CRUD操作，以及另外一些便利的方法使得该API更易于使用。由于主要讨论的是iBATIS，因此我们将首先使用前面所定义的基于SQL Map的版本来实现这个DAO接口。首先，来看一下用于描述iBATIS配置的dao.xml文件。

10.4.1 配置 iBATIS DAO

代码清单10-7给出了一个DAO配置文件，其中包含一个基于SQL Map的DAO。我们定义了名为sqlmap的上下文，该上下文将定义在一个位于类路径（例如，如果此上下文用在Web应用程序中，那么该文件的路径将是/WEB-INF/classes/）上的SqlMapConfig.xml文件中。

代码清单10-7 iBATIS dao.xml样例

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE daoConfig
PUBLIC
    "-//ibatis.apache.org/DTD DAO Configuration 2.0//EN"
    "http://ibatis.apache.org/dtd/dao-2.dtd">
<daoConfig>
  <context id="sqlmap">
    <transactionManager type="SQLMAP">
```

```
<property name="SqlMapConfigResource"
value="SqlMapConfig.xml"/>
</transactionManager>
<dao interface="com.mycompany.system.dao.AccountDao"
implementation=
"com.mycompany.system.dao.sqlmap.AccountDaoImpl"/>
</context>
</daoConfig>
```

dao.xml文件将被DaoManagerBuilder用来创建DaoManager实例。下面就来讨论如何创建它。

10.4.2 创建 DaoManager 实例

为了能够使用已经定义的DAO管理器，需要使用DaoManagerBuilder来创建一个它的实例。代码清单10-8给出了一个代码片段，可以用它来创建一个DaoManager实例。

代码清单10-8 构建DaoManager的样例代码

```
private DaoManager getDaoManager() {
    DaoManager tempDaoManager = null;
    Reader reader;
    try {
        reader = Resources.getResourceAsReader("Dao.xml");
        tempDaoManager =
            DaoManagerBuilder.buildDaoManager(reader);
    } catch (Exception e) {
        e.printStackTrace();
        fail("Cannot load dao.xml file.");
    }
    return tempDaoManager;
}
```

现在我们已经有一些代码和配置元素，下面将深入考察这些代码的作用。

以上代码寻找一个名为Dao.xml的资源，该资源位于类加载器（classloader）将会检查的某个根目录下。例如，在Tomcat中，它可能位于你的Web应用程序的WEB-INF/classes目录下，或者在WEB-INF/lib目录下的一个JAR文件中（只要它位于JAR文件的顶层即可）。

一旦上述代码拥有了配置文件，就会将数据传递给DaoManagerBuilder，并且要求它构建一个DaoManager实例。

以上代码来自我们用于构造和测试当前所讨论的DAO实现的JUnit测试，因此它的异常处理非常简单。在真正的产品应用程序中，你不可能只进行这样的异常处理。

10.4.3 定义事务管理器

接下来，将定义事务管理器，该事务管理器将基于我们在SQL Map配置文件中定义的事务管理器，而SQL Map配置文件是通过嵌套在<transactionManager>元素中的SqlMapConfigResource特性来定义的。所有的事务管理功能，只要是在直接使用SQL Maps时可用，那么在我

们的DAO实现类中也一样可用。代码清单10-9给出了此例中所使用的SQLMapConfig.xml文件。

代码清单10-9 SQLMapConfig.xml样例文件

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE sqlMapConfig
PUBLIC
"-//ibatis.apache.org/DTD SQL Map Config 2.0//EN"
"http://ibatis.apache.org/dtd/sql-map-config-2.dtd">
<sqlMapConfig>
  <properties resource="SqlMapConfig.properties" />
  <settings
    errorTracingEnabled="true"
    cacheModelsEnabled="true"
    enhancementEnabled="true"
    lazyLoadingEnabled="true"
    maxRequests="32"
    maxSessions="10"
    maxTransactions="5"
    useStatementNamespaces="true"
  />
  <transactionManager type="JDBC" >
    <dataSource type="SIMPLE">
      <property name="JDBC.Driver" value="${driver}"/>
      <property name="JDBC.ConnectionURL"
        value="${connectionUrl}"/>
      <property name="JDBC.Username" value="${username}"/>
      <property name="JDBC.Password" value="${password}"/>
    </dataSource>
  </transactionManager>
  <sqlMap
    resource=
      "com/mycompany/system/dao/sqlmap/Account.xml" />
</sqlMapConfig>
```

此文件中所有的设置都已经在第4章中详细介绍过了，因此不再赘述。

10.4.4 加载映射

除了定义事务管理之外，所有在SQL Map配置文件中定义的映射也会被加载。例如这样一个Account.xml样例文件，它为我们的DAO类定义了所有的已映射语句，如代码清单10-10所示。

代码清单10-10 SQL Map样例文件

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE sqlMap
PUBLIC "-//ibatis.apache.org/DTD SQL Map 2.0//EN"
"http://ibatis.apache.org/dtd/sql-map-2.dtd">
<sqlMap namespace="Account">

  <typeAlias alias="Account"
    type="${BeanPackage}.Account" />
```



```
<typeAlias alias="IdDescription"
            type="{BeanPackage}.IdDescription" />

<insert id="insert" parameterClass="Account">
  <selectKey keyProperty="accountId" resultClass="int">
    SELECT nextVal('account_accountid_seq')
  </selectKey>
  INSERT INTO Account (
    accountId,
    username,
    password,
    firstName,
    lastName,
    address1,
    address2,
    city,
    state,
    postalCode,
    country
  ) VALUES (
    #accountId#,
    #username:varchar#,
    #password:varchar#,
    #firstName:varchar#,
    #lastName:varchar#,
    #address1:varchar#,
    #address2:varchar#,
    #city:varchar#,
    #state:varchar#,
    #postalCode:varchar#,
    #country:varchar#
  )
</insert>

<update id="update">
  update Account set
    username = #username:varchar#,
    password = #password:varchar#,
    firstName = #firstName:varchar#,
    lastName = #lastName:varchar#,
    address1 = #address1:varchar#,
    address2 = #address2:varchar#,
    city = #city:varchar#,
    state = #state:varchar#,
    postalCode = #postalCode:varchar#,
    country = #country:varchar#
  where accountId = #accountId#
</update>

<delete id="delete">
  delete from Account
  where accountId = #accountId#
</delete>
```

```
<sql id="allFields">
  accountId as "accountId",
  username,
  password,
  firstName as "firstName",
  lastName as "lastName",
  address1,
  address2,
  city,
  state,
  postalCode as "postalCode",
  country
</sql>

<sql id="whereByExample">
  <dynamic prepend=" where ">
    <isNotEmpty property="city">
      city like #city#
    </isNotEmpty>
    <isNotNull property="accountId" prepend=" and ">
      accountId = #accountId#
    </isNotNull>
  </dynamic>
</sql>

<sql id="getByExample">
  select
  <include refid="allFields" />
  from Account
  <include refid="whereByExample" />
</sql>

<select id="getAccountListByExample"
  resultClass="Account">
  <include refid="getByExample" />
</select>

<select id="getMapListByExample" resultClass="hashmap">
  <include refid="getByExample" />
</select>

<select id="getIdDescriptionListByExample"
  resultClass="IdDescription">
  select
    accountId as id,
    COALESCE(firstname, '(no first name)')
    ||
    || COALESCE(lastname, '(no last name)')
    as description
  from Account
  <include refid="whereByExample" />
</select>

<select id="getId" resultClass="Account">
```

① 字段列表SQL片段

② WHERE子句SQL片段

③ SQL片段组合

④ 用于获取bean的已映射语句

⑤ 用于获取映射的已映射语句

⑥ 用于获取名/值对列表的已映射语句

```
select
  <include refid="allFields" />
from Account
where accountId = #value#
</select>

</sqlMap>
```

在这个SQL映射中，我们定义了一个SQL片段①用来列出所有的字段。此时，由于我们所使用的数据库驱动程序对列名不区分大小写，因此我们使用显式的列别名，以确保iBATIS在进行隐式属性映射时能够正确地处理它们。另一个SQL片段②用于定义一个我们将要使用的较复杂的WHERE子句。第3个SQL片段③用来将前两个SQL片段组合成为一个片段，然后我们就可以在两个查询语句中重复使用它——一次用于获取bean列表④，另一次用于获取Map列表⑤。在getIdDescriptionListByExample已映射语句⑥中，我们再一次使用了这个复杂的WHERE子句（SQL片段）以获取由不同类型的bean所组成的列表。

10.4.5 DAO 实现编码

最后，我们开始实际的DAO实现。如前所述，要创建DAO，需要同时提供接口和实现。此时，接口被定义为com.mycompany.system.dao.AccountDao，而实现则被定义为com.mycompany.system.dao.sqlmap.AccountDaoImpl。

我们已经在10.3节中给出了这个接口，故此处将不再重复，下面来看一下DAO实现类（见代码清单10-11）。

代码清单10-11 Account DAO的实现

```
package org.apache.mapper2.examples.chapter10.dao.sqlmap;

import com.ibatis.dao.client.DaoManager;
import com.ibatis.dao.client.template.SqlMapDaoTemplate;
import org.apache.mapper2.examples.bean.Account;
import org.apache.mapper2.examples.bean.IdDescription;
import
  org.apache.mapper2.examples.chapter10.dao.AccountDao;

import java.util.List;
import java.util.Map;

public class AccountDaoImpl extends SqlMapDaoTemplate
  implements AccountDao {
  public AccountDaoImpl(DaoManager daoManager) {
    super(daoManager);
  }

  public Integer insert(Account account) {
    return (Integer) insert("Account.insert", account);
  }
}
```



```
public int update(Account account) {
    return update("Account.update", account);
}

public int delete(Account account) {
    return delete(account.getAccountId());
}

public int delete(Integer accountId) {
    return delete("Account.delete", accountId);
}

public List<Account> getAccountListByExample(
    Account account) {
    return queryForList("Account.getAccountListByExample",
        account);
}

public List<Map<String, Object>>
    getMapListByExample(Account account) {
    return queryForList("Account.getMapListByExample",
        account);
}

public List<IdDescription>
    getIdDescriptionListByExample(
        Account account) {
    return
        queryForList("Account.getIdDescriptionListByExample",
            account);
}

public Account getId(Integer accountId) {
    return (Account)queryForObject("Account.getId",
        accountId);
}

public Account getId(Account account) {
    return getId(account.getAccountId());
}
}
```

以上代码从表面上看，似乎没有什么了不起的。在类声明中，我们看到该类实现了 AccountDao 接口，并扩展了 SqlMapDaoTemplate 类。

通过将所有的 SQL Map API 组件封装在一个简洁的包中，SqlMapDaoTemplate 类为我们完成了很多相当繁琐的工作。此外，该类还提供了很多局部方法，这些方法为我们把调用委托给 SqlMapExecutor，这样，就不需要获取 SqlMapClient 或 SqlMapExecutor 实例，而可以直接调用它们的方法，就好像它们是我们的 DAO 类的一部分一样。

到目前为止，仅仅为了将我们的 DAO 类从其实现中分离出来似乎就要做大量的工作，但现在

如果还需要创建其他的DAO类，则仅仅需要创建一个接口、一个实现和一份SQL Map文件，最后再在Dao.xml文件中添加一行代码即可^①。在下一章中，我们将直接使用Hibernate和JDBC来实现相同的DAO接口。所有这3种实现使用的都是相同的API（即AccountDao接口），尽管用来访问数据库的底层技术完全不同。

10.5 小结

在本章中，我们解释了在应用程序中使用数据访问层的基本原理，以及如何设置iBatis DAO框架。另外我们还研究了一些配置iBatis DAO框架的更高级的方式，以及如何创建基于SQLMap的DAO。

在下一章中，我们将讨论如何设置其他的DAO类型，以及通过构建一个非SQL的DAO实现来介绍有关iBatis DAO框架的更高级用法。同时还将研究利用Spring框架来在应用程序中实现DAO层的方式，并且提出一些你在从头创建自己的DAO层时需要注意的事项。

^① 见代码清单10-7的<dao>元素和代码清单10-9的<sqlMap>元素。——译者注

第 11 章

DAO使用进阶

本章内容

- DAO的更多样例
- 使用Spring替代iBATIS
- 从头开始创建DAO层

正如第10章所述，DAO模式可用于隐藏那些与数据相关的API独特实现特征，而为应用程序开发人员^①提供一套简单而通用的API。这个模式的确非常强大，而且它并非iBATIS所特有的，其他一些项目也创建了自己的DAO实现，可以通过iBATIS使用它们。

本章将介绍另外一些基于SQL的DAO实现，以及一些针对其他数据源（LDAP和Web服务）的DAO实现。然后尝试研究DAO层的一些其他选项，例如Spring框架中卓越的DAO实现。本章也将告诉你如何创建自己的定制DAO层。

11.1 不是基于SQLMap的DAO实现

第10章为Account DAO定义了接口，然后为它构建了一个基于SQLMAP的DAO实现。接下来的两节将分别用Hibernate和JDBC再次实现这个接口，以向你展示在基于iBATIS的应用程序中，DAO模式是如何使你以非常容易地使用不同的数据库访问技术。

11.1.1 Hibernate 版本的DAO实现

Hibernate版本的DAO实现与SQLMap版本的DAO实现差异很大，但由于它们实现的DAO接口相同，所以它的使用方式甚至使用它的应用程序代码几乎都是一样的。

1. 定义DAO上下文

代码清单11-1展示了一个在Dao.xml文件中需要的XML片段，以描述将会使用Hibernate的DAO上下文。

^① 指数数据访问层之上的其他开发人员。——译者注

代码清单11-1 XML片段，定义了使用Hibernate的DAO上下文

```
<context id="hibernate">
  <transactionManager type="HIBERNATE">
    <property name="hibernate.connection.driver_class"
      value="org.postgresql.Driver" />
    <property name="hibernate.connection.url"
      value="jdbc:postgresql:ibatisdemo" />
    <property name="hibernate.connection.username"
      value="ibatis" />
    <property name="hibernate.connection.password"
      value="ibatis" />
    <property name="hibernate.connection.pool_size"
      value="5" />
    <property name="hibernate.dialect"
      value="net.sf.hibernate.dialect.PostgreSQLDialect" />
    <property name="map.Account"
      value="
        ${DaoHomeRes}/hibernate/Account.hbm.xml" />
  </transactionManager>
  <dao interface="${DaoHome}.AccountDao"
    implementation="
      ${DaoHome}.hibernate.AccountDaoImpl"/>
</context>
```

正如我们在第10章（见10.2.2节）中讨论的那样，HIBERNATE类型的事务管理器期望你将那些通常会放在hibernate.properties文件中的特性列为<transactionManager>元素的特性。

因为期望源代码树保持清晰，所以我们没有将Account实体bean的Hibernate映射文件（Account.hbm.xml）与Account实体bean放在同一个包中，而是通过使用一个map.特性将它添加到Hibernate配置中。记住，所有这一切都只是为了保证我们的数据访问实现与接口分离。

2. 映射Account表

代码清单11-2中的映射文件非常简单，因为我们直接从特性文件中映射到具有同名的列中，且没有其他关联项。

代码清单11-2 Account表的Hibernate映射文件

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping
  PUBLIC "-//Hibernate/Hibernate Mapping DTD//EN"
  "http://hibernate.sourceforge.net/hibernate-mapping-2.0.dtd">
<hibernate-mapping>
  <class
    name="org.apache.mapper2.examples.bean.Account" ① 将表映射到类
    table="Account">
    <id name="accountId" type="int" column="accountid"> ② 连接ID生成序列
      <generator class="sequence">
        <param
```

```
name="sequence">account_accountid_seq</param>
</generator>
</id>
<property name="username" />
<property name="password" />
<property name="firstName" />
<property name="lastName" />
<property name="address1" />
<property name="address2" />
<property name="city" />
<property name="state" />
<property name="postalCode" />
<property name="country" />
</class>
</hibernate-mapping>
```

如果你以前使用过Hibernate，那么代码清单11-2所示的文件对你来说就应该非常容易理解，它仅仅是一个简单的表映射。如果以前没有使用过Hibernate，那么这个映射文件可能就不好理解了。简单地说，映射文件所做的就是将我们的Account实体bean中的特性映射到数据库账户表中的列上^①。同时，它还告诉Hibernate我们希望如何在bean中为新创建的数据库记录产生id特性^②。

3. 实际的DAO实现

DAO实现的Java源代码较为冗长，因为我们要将Account表的内容映射为好几个不同的对象类。这种情况对于Hibernate来说就显得有些繁琐了，如代码清单11-3所示。

代码清单11-3 Account DAO接口的Hibernate实现

```
public class AccountDaoImpl
    extends HibernateDaoTemplate
    implements AccountDao {

    private static final Log log =
        LoggerFactory.getLog(AccountDaoImpl.class);

    public AccountDaoImpl(DaoManager daoManager) {
        super(daoManager);
        if (log.isDebugEnabled()) {
            log.debug("Creating instance of " + getClass());
        }
    }

    public Integer insert(Account account) { ◀— ① 插入新的账户
        try {
            getSession().save(account);
        } catch (HibernateException e) {
            log.error(e);
            throw new DaoException(e);
        }
        return account.getAccountid();
    }
}
```

```
}

public int update(Account account) { ◀— ② 更新账户
    try {
        getSession().save(account);
    } catch (HibernateException e) {
        log.error(e);
        throw new DaoException(e);
    }
    return 1;
}

public int delete(Account account) { ◀— ③ 删除账户
    try {
        getSession().delete(account);
    } catch (HibernateException e) {
        log.error(e);
        throw new DaoException(e);
    }
    return 1;
}

public int delete(Integer accountId) { ◀— ④ 删除账户
    Account account = new Account();
    account.setAccountId(accountId);
    return delete(account);
}

public List<Account> getAccountListByExample( ◀— ⑤ 获得bean列表
    Account acct) {
    List accountList;
    Session session = this.getSession();

    Criteria criteria =
        session.createCriteria(Account.class);
    if (!isEmpty(acct.getCity())) {
        criteria.add(
            Expression.like("city", acct.getCity())
        );
    }
    if (!isEmpty(acct.getAccountId())) {
        criteria.add(
            Expression.eq("accountId", acct.getAccountId())
        );
    }

    try {
        accountList = criteria.list();
    } catch (HibernateException e) {
        log.error(
            "Exception getting list: " +
            e.getMessage(), e);
        throw new DaoException(e);
    }
}
```

理解，
不好理
解账户
生id

的对象


```
        return (List<Account>)accountList;
    }

    public List<Map<String, Object>> getMapListByExample( ◀ 6 获得Map列表
        Account account
    )
    {
        List<Account> accountList =
            getAccountListByExample(account);
        List<Map<String, Object>> mapList =
            new ArrayList<Map<String, Object>>();
        for (Account acctToAdd : accountList) {
            Map<String, Object> map =
                new HashMap<String, Object>();
            map.put("accountId", acctToAdd.getAccountId());
            map.put("address1", acctToAdd.getAddress1());
            map.put("address2", acctToAdd.getAddress2());
            map.put("city", acctToAdd.getCity());
            map.put("country", acctToAdd.getCountry());
            map.put("firstName", acctToAdd.getFirstName());
            map.put("lastName", acctToAdd.getLastName());
            map.put("password", acctToAdd.getPassword());
            map.put("postalCode", acctToAdd.getPostalCode());
            map.put("state", acctToAdd.getState());
            map.put("username", acctToAdd.getUsername());
            mapList.add(map);
        }
        return mapList;
    }

    public List<IdDescription> getIdDescriptionListByExample(
        Account exAcct ◀ 7 获得bean的不同列表
    )
    {
        List<Account> acctList =
            getAccountListByExample(exAcct);
        List<IdDescription> idDescriptionList =
            new ArrayList<IdDescription>();
        for (Account acct : acctList) {
            idDescriptionList.add(
                new IdDescription(
                    acct.getAccountId(),
                    acct.getFirstName() + " " + acct.getLastName()
                )
            );
        }
        return idDescriptionList;
    }

    public Account getId(Integer accountId) { ◀ 8 获得单个账户
        Session session = this.getSession();
        try {
            return (Account) session.get(
                Account.class, accountId);
        } catch (HibernateException e) {
```

```
        log.error(e);
        throw new DaoException(e);
    }

    public Account getById(Account account) { ◀——⑨ 获得单个账户
        return getById(account.getAccountId());
    }
}
```

比之SQLMap实现，这个Hibernate版本实现的代码量显然多了许多。对于那些处理Account对象的情况（①~⑤以及⑧~⑨），代码是相当简单的，但当我们返回一列Map对象⑥或IdDescription对象⑦时，事情就变得复杂起来了。因为Hibernate就是被设计为用于将一张数据库表映射到一个Java实体类的，当需要将同一张表映射到多个不同的类时，事情就不那么容易了。

在接下来的这个DAO实现示例中，我们将不再使用任何映射工具了，而是直接使用JDBC来构建DAO。

11.1.2 JDBC 版本的 DAO 实现

为我们之前的DAO接口所构建的最后一个实现将直接使用JDBC。基于JDBC的实现，其最大的好处就在于它需要的配置文件最少且灵活性最强。

代码清单11-4给出了这个dao.xml配置文件。

代码清单11-4 XML片段，定义了使用JDBC的DAO上下文

```
<context id="jdbc">
  <transactionManager type="JDBC">
    <property name="DataSource"
      value="SIMPLE"/>
    <property name="JDBC.Driver"
      value="org.postgresql.Driver" />
    <property name="JDBC.ConnectionURL"
      value="jdbc:postgresql:ibatisdemo" />
    <property name="JDBC.Username"
      value="ibatis" />
    <property name="JDBC.Password"
      value="ibatis" />
    <property name="JDBC.DefaultAutoCommit"
      value="true" />
  </transactionManager>
  <dao interface="${DaoHome}.AccountDao"
    implementation="${DaoHome}.jdbc.AccountDaoImpl"/>
</context>
```

这就是所有的配置文件了——剩下的所有工作就都在接下来的代码中了，稍后你就会看到。这个DAO的JDBC版本实现，其代码量相当于Hibernate版本代码的两倍还多，几乎相当于SQL映

射版本的7倍。如果再加上配置文件的代码，那么“代码行数”统计将如表11-1所示。即使不算配置文件，最终我们的代码编写的工作量也翻了一番。我们并不是说使用JDBC开发应用程序是一个馊主意，我们想强调的仅仅是：为了换取更大的灵活性和最少的配置，那么你需要付出的代价是编写的代码量将大大增加。

表11-1 各种DAO实现的代码及配置文件的行数

实现方式	配置文件	代 码	总 计
iBATIS	118+8=126	53	179
Hibernate	23+20=43	141	184
JDBC	18	370	388

看看JDBC版的DAO实现

由于JDBC版的DAO实现代码量过大，我们无法在本章中给出该类的完整代码，而仅仅只关注构建这类所用的一些技巧。

需要做的第一件事就是扩展iBATIS所提供的JdbcDaoTemplate类来构建DAO。通过扩展该类，我们接下来的工作就可以轻松很多，因为该类可以为我们管理连接，这就意味着我们不需要在自己的代码中关闭它了。

虽然这似乎是一件小事，但在一个大型系统中，如果没有正确地管理好连接将会使系统在几个小时（如果不是几分钟的话）后就陷入瘫痪。这就将我们带入了下面这个减少代码的小技巧——总是创建可帮助管理所使用资源的辅助方法。

```
private boolean closeStatement(Statement statement) {  
    boolean returnValue = true;  
    if (null != statement) {  
        try {  
            statement.close();  
        } catch (SQLException e) {  
            log.error("Exception closing statement", e);  
            returnValue = false;  
        }  
    }  
    return returnValue;  
}
```

考虑到关闭Statement时有可能抛出SQLException，所以上方法在关闭Statement时捕获SQLException，并将它封装为DaoException抛出^①。有了如上方法，我们就再不需要反复写这段相同的代码，而仅仅需要简单地调用closeStatement()，它就会记录原始异常的内容并抛出DaoException。基于同样的原因，关闭ResultSet对象也可使用类似的方法。

我们构建的下一个快捷方法用于从JDBC返回的ResultSet对象中提取出数据结构（如

① 从代码来看，这里仅仅是捕获了SQLException异常，但并没有封装为DaoException抛出，作者此处说明与代码不符。——译者注

Account对象或者Map对象):

```
private Account extractAccount(ResultSet rs
) throws SQLException {
    Account accountToAdd = new Account();
    accountToAdd.setAccountId(rs.getInt("accountId"));
    accountToAdd.setAddress1(rs.getString("address1"));
    accountToAdd.setAddress2(rs.getString("address2"));
    accountToAdd.setCity(rs.getString("city"));
    accountToAdd.setCountry(rs.getString("country"));
    accountToAdd.setFirstName(rs.getString("firstname"));
    accountToAdd.setLastName(rs.getString("lastname"));
    accountToAdd.setPassword(rs.getString("password"));
    accountToAdd.setPostalCode(rs.getString("postalcode"));
    accountToAdd.setState(rs.getString("state"));
    accountToAdd.setUsername(rs.getString("username"));
    return accountToAdd;
}

private Map<String, Object> accountAsMap(ResultSet rs
) throws SQLException {
    Map<String, Object> acct =
        new HashMap<String, Object>();
    acct.put("accountId", rs.getInt("accountId"));
    acct.put("address1", rs.getString("address1"));
    acct.put("address2", rs.getString("address2"));
    acct.put("city", rs.getString("city"));
    acct.put("country", rs.getString("country"));
    acct.put("firstName", rs.getString("firstname"));
    acct.put("lastName", rs.getString("lastname"));
    acct.put("password", rs.getString("password"));
    acct.put("postalCode", rs.getString("postalcode"));
    acct.put("state", rs.getString("state"));
    acct.put("username", rs.getString("username"));
    return acct;
}

private IdDescription accountAsIdDesc(ResultSet rs
) throws SQLException {
    return new IdDescription(
        new Integer(rs.getInt("id")),
        rs.getString("description"));
}
```

因为需要在很多地方执行这种提取操作，因此构建专门的方法来简化这个映射（即提取）过程可以使我们以后在创建实际方法^①节省大量的时间也减少了出错的可能性。

接下来的这个方法为DAO接口中那些QBE（query-by-example，按样例查询）方法创建了PreparedStatement。这个方法非常复杂，它更使我们强烈地感受到SQLMap实现看起来是多么吸引人。其他那些辅助方法虽然冗长乏味但还不失简单，而这个方法则是纯粹的冗长乏味、容易出错、难以测试——总之，不到万不得已不要写这样的方法。

① 即实现DAO接口的那些方法。——译者注

```
private PreparedStatement prepareQBStatement(  
    Account account,  
    Connection connection,  
    PreparedStatement ps,  
    String baseSql  
) throws SQLException {  
    StringBuffer sqlBase = new StringBuffer(baseSql);  
    StringBuffer sqlWhere = new StringBuffer("");  
    List<Object> params = new ArrayList<Object>();  
  
    String city = account.getCity();  
    if (!isEmpty(city)) {  
        sqlWhere.append(" city like ?");  
        params.add(account.getCity());  
    }  
  
    Integer accountId = account.getAccountId();  
    if (!isEmpty(accountId)) {  
        if (sqlWhere.length() > 0) {  
            sqlWhere.append(" and");  
        }  
        sqlWhere.append(" accountId = ?");  
        params.add(account.getAccountId());  
    }  
  
    if (sqlWhere.length() > 0) {  
        sqlWhere.insert(0, " where");  
        sqlBase.append(sqlWhere);  
    }  
  
    ps = connection.prepareStatement(sqlBase.toString());  
    for (int i = 0; i < params.size(); i++) {  
        ps.setObject(i+1, params.get(i));  
    }  
    return ps;  
}
```

好，现在所有的辅助方法都已准备就绪，可以开始构建DAO接口中的公共方法了。insert方法应该算是其中最为复杂的了，因为它同时需要查询操作和插入操作。

```
public Integer insert(Account account) {  
    Connection connection = this.getConnection();  
    Statement statement = null;  
    PreparedStatement ps = null;  
    ResultSet rs = null;  
    Integer key = null;  
    if (null != connection) {  
        try {  
            statement = connection.createStatement();  
            rs = statement.executeQuery(sqlGetSequenceId);  
            if (rs.next()) {  
                key = new Integer(rs.getInt(1));  
                account.setAccountId(key);  
                if (log.isDebugEnabled()) {
```

```
        log.debug("Key for inserted record is " + key);
    }
}

ps = connection.prepareStatement(sqlInsert);
int i = 1;
ps.setObject(i++, account.getAccountId());
ps.setObject(i++, account.getUsername());
ps.setObject(i++, account.getPassword());
ps.setObject(i++, account.getFirstName());
ps.setObject(i++, account.getLastName());
ps.setObject(i++, account.getAddress1());
ps.setObject(i++, account.getAddress2());
ps.setObject(i++, account.getCity());
ps.setObject(i++, account.getState());
ps.setObject(i++, account.getPostalCode());
ps.setObject(i++, account.getCountry());
ps.executeUpdate();
} catch (SQLException e) {
    log.error("Error inserting data", e);
    throw new DaoException(e);
} finally {
    closeStatement(ps);
    closeResources(statement, rs);
}
}

return key;
}
```

这个方法中，我们首先要为将插入的新记录获取一个新id，在要传入的bean上设置它，然后将这个bean插入到数据库。代码非常容易理解，只是与iBATIS版本和Hibernate版本的对应方法比较起来显得有些冗长。

该版本实现中的其他方法也是同样的直接，所以这里就不详述它们了。

11.2 为其他数据源使用 DAO 模式

DAO模式与Gateway模式非常相似，这使得它对于许多其他数据源同样适用，如LDAP和Web服务。

你可能对Gateway模式这个名称并不熟悉，事实上这个模式通常也被称为Wrapper，因为这个模式所做的正是“包装”。它将一个API包装一下，使得它看起来像一个简单的对象，如图11-1所示，其中WebServiceGateway接口隐藏了底层的具体实现。

你可能会觉得Gateway模式似曾相识，没错，因为Gateway模式的思想就是DAO模式背后的思想，DAO只不过是一个专门针对数据库的入口，它可用于帮助你管理事务、连接池以及其他特定于数据库的问题。

11.2.1 示例：为 LDAP 使用 DAO

LDAP是一个用于存储具有层次结构的数据的强大工具，它经常被网络管理员用于追踪用户、组成员以及其他类似数据之间的从属关系。举个例子，NDS（Novell Directory Services）和微软

的ActiveDirectory都是基于LDAP的，且都给出了LDAP的API。

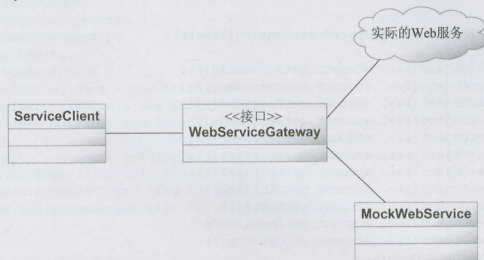


图11-1 DAO模式和Gateway模式非常相似

使用DAO模式来访问一个使用了LDAP的目录，能够很好地使你在编写应用程序代码时不用去考虑JNDI编程的细微差别。通过创建小型的专用类集，可以构建出一些轻量的、可测试的JNDI组件，然后将它们隐藏到你的DAO实现中，从而不需暴露出实际的数据源。

1. 了解LDAP术语

在开始构建一个完整的基于LDAP目录的DAO实现之前，首先回顾一下有关LDAP的术语。LDAP故意设计得非常模糊，因为它的目的就是为访问具有层次结构的数据集合提供一个非常灵活通用的协议。

构成LDAP目录的基本单元称为记录项(entry)，它可以包含被称为属性的数据，或者其他记录项，或两者同时包含。每个记录项只有一个父记录项，并且被一个DN (Distinguished Name) 唯一标识，该标识在整个目录中是唯一的。记录项中的数据元素被一个或多个该记录项代表的对象类别所定义。

存储在LDAP目录中的数据由若干属性组成，一个属性就是一个名/值对，实际上与Java中的Map非常相似。记录项所属的对象类将决定该记录项可以包含哪些可选属性，以及哪些属性则是它必须有的。

举个例子，如果想用Java创建一个联系管理器(contact manager)来管理普通的LDAP记录项，那么应该定义一个bean来表示一条记录项，该bean如下所示：

```
public class Contact {
    private String userId;
    private String mail;
    private String description;
    private String firstName;
    private String lastName;
    // Getters and Setters to make properties...
}
```

要将这个对象存储到一个LDAP目录中，一种方式就是简单地将这个Java对象序列化到LDAP目录中。在我们的例子中，却并不打算采用这种方式，原因有两个。其一是我们希望LDAP目录能够与其他系统互操作，而这些系统有可能是非Java的^①。另一个原因则是，我们希望充分利用基于LDAP的查询所带来的巨大的性能优势——LDAP就是被设计为这样使用的。

2. 从Java到LDAP记录项之间的映射

正如之前提到的，每一个LDAP记录项都表示一个或多个对象类。这些对象类定义属性组。由于这些属性与Java的Map接口之间的相似性，创建一个使用Map来隐藏JNDI属性结构的DAO的基于Map就显得非常容易。尽管如此，本节仍将创建一个DAO，用于将前一节的那个bean映射到一个使用了表11-2中映射的LDAPinetOrgPerson的记录项：

表11-2 从JavaBean到LDAP属性的映射

Bean特性	LDAP属性
userId	uid
mail	mail
description	description
lastName	sn
firstName	givenName

这个映射会在我们的DAO实现中通过一些方法来完成，这些方法会根据一个Attributes对象创建一个bean，或者反过来根据一个bean创建一个Attributes对象。虽然为此创建一个基于反射的映射机制是可能的，但是为了使DAO实现尽量简单，我们还是决定使用硬编码^②的方式来实现这个映射。代码清单11-5包含了DAO实现中的3个方法，由这3个方法完成映射。

代码清单11-5 针对LDAP DAO实现的辅助方法

```
private Attributes getAttributes(Contact contact){
    Attributes returnValue = new BasicAttributes();
    returnValue.put("mail", contact.getMail());
    returnValue.put("uid", contact.getUserId());
    returnValue.put("objectClass", "inetOrgPerson");
    returnValue.put(
        "description", contact.getDescription());
    returnValue.put("sn", contact.getLastName());
    returnValue.put("cn", contact.getUserId());
    returnValue.put("givenName", contact.getFirstName());
    return returnValue;
}

private Contact getContact(Attributes attributes) {
    Contact contact = new Contact();
    contact.setDescription(
        getAttributeValue(attributes, "description"));
}
```

① 如果使用Java的对象序列化技术就可能会破坏这种跨平台性。——译者注
② 所谓硬编码，是指代码直接引用了映射两端的字段名和属性名。——译者注

```
contact.setLastName(  
    getAttributeValue(attributes, "sn");  
contact.setFirstName(  
    getAttributeValue(attributes, "givenName");  
contact.setMail(getAttributeValue(attributes, "mail"));  
contact.setUserId(  
    getAttributeValue(attributes, "uid"));  
return contact;  
}  
  
private String getAttributeValue(  
    Attributes attributes, String attrID  
) {  
    Attribute attribute = attributes.get(attrID);  
    try {  
        return (null==attribute?"":(String)attribute.get());  
    } catch (NamingException e) {  
        throw new DaoException(e);  
    }  
}
```

Attributes接口来自于Sun的JDK的JNDI包，该包中的BasicAttributes类实现了这个接口。Contact类就是我们希望映射到LDAP目录中去的那个bean。在代码的最后，getAttributeValue()方法是一个辅助方法，通过自动处理空值以及转换特定于JNDI的异常为DaoException来简化整个映射过程。

就像其他的DAO实现一样，我们需要决定何时以及如何访问数据库。如果正在使用可以提供JNDI上下文的J2EE容器，那么你可能会非常希望能够使用它。如果它能够满足需求，那为什么不呢？但是，你也需要为此付出一定的代价。虽然使用这种方式可以简化代码，但它也会使得测试变得更加困难。如果确实需要，这种牺牲可能还是可接受的。

在这个例子中，我们希望让代码尽量地可测试，所以使用了基于构造函数的注入以便在运行时配置DAO类。因为iBATIS DAO框架不支持这种方式，我们也创建了一个默认构造函数，它使用所需要的两个值作为默认值。11.3节将讨论为DAO层使用Spring框架，这个框架允许通过配置文件的方式注入依赖关系，但目前为止，我们只能使用默认构造函数这个方法。

第二个构造函数需要两个参数以完成两项设置，这两项设置在默认构造函数中都是硬编码设置的。第一项设置用于确定LDAP中我们的联系bean的DN属性。这个属性与数据库表的主键十分类似，只不过它需要在整个目录中保证唯一，而不像主键，只需要在一张表中保证唯一即可。接下来的这个方法在DAO实现中用于为我们的联系bean创建一个唯一的DN：

```
private String getDn(String userId){  
    return MessageFormat.format(this.dnTemplate, userId);  
}
```

我们需要的第二项设置用于获取初始的目录上下文。DAO实现的默认构造函数，使用了一些硬编码的特性来连接到LDAP目录，但第二个构造函数则允许注入一些自定义特性以达到其他的目的。这些特性都是用于获取初始的目录上下文的。


```
private DirContext getInitialContext() {
    DirContext ctx = null;
    try {
        ctx = new InitialDirContext(env);
    } catch (NamingException e) {
        log.error("Exception getting initial context", e);
        throw new DaoException(e);
    }
    return ctx;
}
```

好，现在所有必需的基础设施代码都已准备就绪，可以处理bean映射以及LDAP目录连接，那么我们可以开始构建DAO实现了。

首先来看一个最简单的方法——通过userId查找到相应的联系对象。以下就是实现这个方法代码：

```
public Contact getById(String id) {
    DirContext ctx = getInitialContext();
    Attributes attributes;
    try {
        attributes = ctx.getAttributes(getDn(id));
    } catch (NamingException e) {
        throw new DaoException(e);
    }
    return getContact(attributes);
}
```

在这个方法中，首先获取目录上下文，然后通过它并基于所传入的userId值的DN来获得相应的联系记录项的所有属性。一旦获得了这些属性，就可以调用之前的辅助方法将这些属性转换为一个联系bean并返回。如果有特定于LDAP的NamingException抛出，则将它替换为DaoException来重新抛出，以保证方法签名不指定具体的数据源。

用LDAP的术语来说，插入操作称为绑定（binding）。应用程序不会知道这些术语，因为它会将它封装在DAO中，而仍然称之为insert。

```
public Contact insert(Contact contact) {
    try {
        DirContext ctx = getInitialContext();
        ctx.bind(getDn(
            contact.getUserId(),
            null,
            getAttributes(contact));
    } catch (Exception e) {
        log.error("Error adding contact", e);
        throw new DaoException(e);
    }
    return contact;
}
```

同样地，更新方法和删除方法也使用相同的技术，调用LDAP的特定类来完成工作并在方法中重新抛出异常。用JNDI的术语来说更新被称为重绑定（rebinding），删除被称为解绑定

(unbinding)。但我们使用了DAO模式，这些术语在应用程序中绝不会出现，我们的应用程序对于在DAO实现中创建的LDAP依赖是毫无知觉的。

LDAP绝不是我们处理过的唯一的较为陌生的数据源，而要在一章中研究完所有这样的数据源是不可能的（即使是一整本书也不可能），下一节将研究Web服务，你在今后的工作中很有可能会遇到它。

11.2.2 示例：为 Web 服务使用 DAO

为Web服务使用DAO模式是一个非常好的主意。为Web服务创建某种抽象层能够简化对使用它的组件的测试。举个例子，如果希望使用信用卡处理服务或者其他一些远程服务，而调用这些服务需要花费时间去连接、执行以及处理结果，那么这就会严重阻碍你对应用程序的测试，因为测试过程总是需要不断地停下来等待（可能是几秒钟甚至更长）。此外，如果每次调用服务都需要付费，或者服务是公共的（如Amazon、Google或者eBay）但每次返回的数据都在变化且无法预期，那么除了在系统集成时或者在用户的验收测试时，以上这样的数据在其他任何目的下都可能是很难使用的，因为费用很高或者返回数据不一致。直接调用Web服务的另一个问题就是，随着时间的变化，你需要为相同的测试准备的数据也不尽相同——而对于单元测试来说，你需要的是合理且静态（或易于预测）的数据。

好，现在假设你正在开发一个系统并且希望给用户在应用程序中进行Google搜索的能力。用于调用Google的Web服务的API非常容易使用，但考虑到现在其他搜索引擎也在提供相似的API，所以我们希望能创建一个更加通用的搜索API来包装Google实现，并从应用程序中调用这个API。

首先，需要提出一个搜索接口以及返回搜索结果的数据结构，就从一个简单的bean以及一个接口开始吧。

```
public class SearchResult {
    private String url;
    private String summary;
    private String title;
    // getters and setters omitted...
}

public interface WebSearchDao {
    List<SearchResult> getSearchResults(String text);
}
```

我们的bean有3个特性，而接口中则只有一个方法，该方法返回一个类型化的列表。以下是这个搜索API的实现，使用的是Google API。

```
public class GoogleDaoImpl implements WebSearchDao {
    private String googleKey;

    public GoogleDaoImpl() {
        this("insert-your-key-value-here");
    }
}
```

```
public GoogleDaoImpl(String key){
    this.googleKey = key;
}

public List<SearchResult> getSearchResults(String text){
    List<SearchResult> returnValue = new
        ArrayList<SearchResult>();
    GoogleSearch s = new GoogleSearch();
    s.setKey(googleKey);
    s.setQueryString(text);
    try {
        GoogleSearchResult gsr = s.doSearch();
        for (int i = 0; i < gsr.getResultElements().length;
            i++){
            GoogleSearchResultElement sre =
                gsr.getResultElements()[i];
            SearchResult sr = new SearchResult();
            sr.setSummary(sre.getSummary());
            sr.setTitle(sre.getTitle());
            sr.setUrl(sre.getURL());
            returnValue.add(sr);
        }
        return returnValue;
    } catch (GoogleSearchFault googleSearchFault) {
        throw new DaoException(googleSearchFault);
    }
}
```

观察以上代码可以发现，Google API与DAO接口非常相似，其实这没什么好奇怪的，但Google API需要一个键才能工作。这在示例中不是问题，因为我们可以将这个键硬编码到实现中。但在一个产品应用程序中，你可能就希望有一种更好的方式了，例如提供一种方式来使用一个特定于用户的键而不是一个共享键。

这也暴露出了iBATIS的DAO层的另一个局限性。因为它不能为一个DAO类创建多种实例而只能使用其默认构造函数，所以要让这个DAO像我们期望的那样工作就需要想一些办法来跳过这些障碍。

下一节将研究如何使用Spring框架来创建一个能力更强的DAO层，这种DAO允许我们通过配置来执行一些漂亮的高级技巧。

11.3 使用 Spring DAO

在应用程序的数据访问层中使用Spring框架可以有多种不同的方式，即使这个应用程序与iBATIS没有任何关系。本节中，你将学习如何利用Spring框架对iBATIS的支持来构建一个数据访问层。

11.3.1 编写代码

Spring框架通过一个针对数据访问对象的模板模式来支持iBATIS，这也就是说，DAO实现可以从扩展Spring框架中一个现成的SqlMapClientTemplate开始。使用这项技术，基于Spring

的AccountDao实现将如代码清单11-6所示。

代码清单11-6 SQL Maps的Account DAO的Spring版本

```
public class AccountDaoImplSpring
    extends SqlMapClientTemplate
    implements AccountDao
{
    public Integer insert(Account account) {
        return (Integer) insert("Account.insert", account);
    }

    public int update(Account account) {
        return update("Account.update", account);
    }

    public int delete(Account account) {
        return delete(account.getAccountId());
    }

    public int delete(Integer accountId) {
        return delete("Account.delete", accountId);
    }

    public List<Account> getAccountListByExample(
        Account account) {
        return queryForList("Account.getAccountListByExample",
            account);
    }

    public List<Map<String, Object>>
        getMapListByExample(Account account) {
        return queryForList("Account.getMapListByExample",
            account);
    }

    public List<IdDescription>
        getIdDescriptionListByExample(Account account) {
        return
            queryForList("Account.getIdDescriptionListByExample",
                account);
    }

    public Account getById(Integer accountId) {
        return (Account) queryForObject("Account.getById",
            accountId);
    }

    public Account getById(Account account) {
        return (Account) queryForList("Account.getById",
            account);
    }
}
```

机敏的读者可能已经注意到了，以上的代码与我们在代码清单 10-11 中看到的几乎没有任何区别，唯一的不同之处就在于这两个实现所具体扩展的类。类中的所有东西都是完全相同的。现在，你需要知道究竟何时应该使用这两个类中的哪一个。继续阅读寻找答案吧。

11.3.2 为什么使用 Spring 代替 iBATIS

这个问题问得非常好——我们的书是关于 iBATIS 的，那么这里为什么讨论为 DAO 层使用其他的技术呢？Spring 和 iBATIS 都有它们各自的优点和缺点，究竟使用哪个需要你理解它们的优点和缺点以及应用程序的具体需求。

iBATIS 的 DAO 层的优点就在于它是一个快速简单的解决方案。只要下载了 iBATIS 的 SQLMap 框架，你就同时拥有了 iBATIS 的 DAO 框架。如果需要的仅仅是事务以及连接管理，那么它是一个比 Spring 简单得多的框架。在这种情况下，iBATIS 的 DAO 框架对你的应用程序来说可能就是一个恰当的解决方案了。

但是，简单性同时也是 iBATIS 的 DAO 框架一个最大的缺点：一旦开始使用 DAO 模式，享受接口与实现的分离（即所谓的解耦）所带来的良好的可测试性，你可能就会希望能够在应用程序的其他地方也能使用相同的方式。

举个例子，在一个 Struts 应用程序中，我们可能就会希望能够在 Action 类和业务逻辑类之间也能使用这种方式^①，就像在业务逻辑类和 DAO 类之间已经使用的这种方式一样。在代码中看到的并不是该段代码需要的某个具体的实现类，而只是它需要的某个接口，而接口的具体实现类都是通过配置文件插入的。这就使得 Action 类能够非常简单，并且每一层的测试也非常容易。

除了能够管理这种接口与实现的分离，Spring 也能够用来管理你的数据库连接和事务，就像 iBATIS 的 DAO 层所做的一样。Spring 的最大优势就在于它不仅仅仅是针对 DAO 类的，它能够管理应用程序的所有部分。

11.4 创建自己的 DAO 层

有时候，不论是 iBATIS 的 DAO 支持还是 Spring 的 DAO 支持都不能精确地满足你的需求。在这种情况下，你就需要创建自己的 DAO 层。

从头开始创建 DAO 层听起来像是一个让人望而生畏的任务。然而你会惊讶于实现模式事实上的直接性。本质上，要创建一个有效的 DAO 层，需要依次完成以下 3 件事情：

- (1) 从实现中分离出接口。
- (2) 用一个可外部配置的工厂来解耦实现。
- (3) 提供事务管理和数据库连接管理。

为此，我们将关注完成前两步需要做些什么事情，至于事务管理以及数据库连接管理，你需

^① 即所有的业务逻辑类也都有一个接口，Action 类只依赖于这些接口而与具体实现无关。——译者注

要参考第7章，从那里开始。

11.4.1 从实现中分离出接口

你希望能够将DAO分离为一个接口和一个实现的原因有两方面。首先，你希望当需要支持数据访问的不同类型时能够使用不同的实现类。其次，分离能使得测试变得更加简单和快速，因为你可以插入一个模拟DAO（mock DAO）来模拟数据访问，从而不需要真的去访问数据库。

如果使用IDE，你可能会认为以上步骤是整个过程中最容易的部分了。因为现在大多数开发环境都支持重构工具，这样你就可以利用工具自动从类中提取出接口。但是，这仅仅是提取接口这一步骤所需工作的一小部分。

如果仅仅是刚开始接触DAO模式，那么你的接口很可能会暴露出特定于JDBC或iBATIS或处理数据库的其他一些工具类。接口中的这一部分将难以管理。这样的接口所带来的问题就是，应用程序将更多地与数据访问的具体实现绑定而不仅仅是与DAO绑定。虽然修改接口以避免这个问题并不是一个多大的困难，但它也绝不是一件轻松的工作。

将使用结果集的代码改为使用集合类（例如一个bean列表）是一种非常直接的修改措施，但就像其他任何代码修改一样，我们需要对这种修改进行充分测试，并且取决于代码在应用程序中的什么位置，测试过程可能并不那么简单。举个例子，如果Web应用程序使用了“快速道读取器（Fast Lane Reader）”模式来提供关于大型数据集的轻量级报表，你的JDBC代码就可能需要直接与视图层（view layer）交互。这就使得测试难以进行，因为任何需要与人交互的操作都非常费时。此外，重写代码使它能执行得和初始时一样好，这也非常困难。一种解决方案就是，编写使用回调函数（callback）的代码来加速数据访问（因此，在这个例子中，可以考虑响应数据的视图请求的RowHandler）。

修改那些直接访问SQLMapAPI的应用程序，让它们访问被更好地封装了的API，这也是一件相当繁琐的事情。举个例子，如果开发了一个类，它直接调用SqlMapClient对象的queryForList()方法，那么你就需要把这个调用重构到DAO类，然后从它返回列表对象，这样你的数据使用者就只需要同DAO打交道了。

11.4.2 创建一个工厂以解耦

到目前为止，已经分离了接口和实现，但我们并不希望把这两者引进到使用DAO的那些类中，因为这样的话就根本没有移除实现上的依赖，而只是添加了接口上的依赖。

我们想说的是，如果现在有了一个DAO接口，也有了针对这个接口的一个实现，那么你应该如何使用这个实现呢？如果不使用工厂，那么你很可能会这样做：

```
AccountDao accountDao = new AccountDaoImpl();
```

发现问题了吗？虽然我们已经为DAO分离出了接口和实现，但我们仍然在同一个地方同时引用了它们。除非把DAO传递到整个应用程序，否则分离不会带来任何价值。一个更好的模式可能

是使用如下方式（或与之类似的方式）：

```
AccountDao accountDao =  
    (AccountDao) DaoFactory.get(AccountDao.class);
```

以上代码看不到实现类了。它在哪里呢？我们既不知道也不关心，因为DaoFactory会处理所有的一切。我们关心的仅仅是DaoFactory项将返回一个对象（这个对象实现了AccountDao接口）。我们并不关心这个对象实例使用的是LDAP、JDBC还是其他任何技术，只要它实现了AccountDao接口，那就足够了。

创建抽象工厂是一件有趣而容易的事情。即使不那么有趣，但绝对容易。本节会构建一个简单的工厂，并讨论为什么你也应该为那些DAO类使用一个类似的工厂。

那么，这个DaoFactory到底是怎样的呢？可能令你感到惊讶的是，它的代码并不多，如代码清单11-7所示。

代码清单11-7 一个超级简单的DAO工厂

```
public class DaoFactory {  
    private static DaoFactory instance = new DaoFactory();  
    private final String defaultConfigLocation =  
        "DaoFactory.properties";  
    private Properties daoMap;  
    private Properties instanceMap;  
  
    private String configLocation = System.getProperty(  
        "dao.factory.config",  
        defaultConfigLocation  
    );  
  
    ① 声明一个私有的构造函数——这里  
    使用了单例（Singleton）模式  
    private DaoFactory() {  
        daoMap = new Properties();  
        instanceMap = new Properties();  
        try {  
            daoMap.load(getInputStream(configLocation));  
        } catch (IOException e) {  
            throw new RuntimeException(e);  
        }  
    }  
  
    private InputStream getInputStream(String configLocation)  
    {  
        return Thread.  
            .currentThread().  
            .getContextClassLoader().  
            .getResourceAsStream(configLocation);  
    }  
  
    ② 声明一个简单的工厂方法  
    public static DaoFactory getInstance() {  
        return instance;  
    }  
}
```

```
    }

    public Object getDao(Class daoInterface) { ← ③ 获得一个DAO
        if (instanceMap.containsKey(daoInterface)) {
            return instanceMap.get(daoInterface);
        }
        return createDao(daoInterface);
    }

    private synchronized Object createDao( ← ④ 保证每种类型只有一个DAO
        Class daoInterface
    ) {
        Class implementationClass;
        try {
            implementationClass = Class.forName((String)
                daoMap.get(daoInterface));

            Object implementation =
                implementationClass.newInstance();
            instanceMap.put(implementationClass, implementation);
        } catch (Exception e) {
            throw new RuntimeException(e);
        }
        return instanceMap.get(daoInterface);
    }
}
```

很明显，这并不是一个最复杂的工厂，但它非常小而实用。它的公共接口只包括两个方法：getInstance()和getDao()。私有构造函数①加载配置文件，本例中它只是一个由名/值对组成的特性文件，包含接口名和该接口对应的实现名。以上这个工厂类是一个自包含的单例，所以其getInstance()方法②只会返回该类的一个实例。而getDao()方法③则用于返回接口的实现。DAO的创建只在确实被请求时才由代码中较后部分的createDao()方法④完成。

11.5 小结

本章针对DAO模式进行了探索。通过阅读本章，相信你已经学会了如何使用除iBATIS的Sql Map之外的其他数据访问工具。你同时也看到了DAO模式如何用作gateway模式的类型，以及如何调整DAO模式以便应用于非典型数据源，如LDAP目录以及Web服务。

我们根据iBATIS所提供的DAO的创建情况，研究了iBATIS DAO框架的局限性。你也看到了如何通过使用Spring框架，使得创建和配置DAO类成为可能的。

我们也简要介绍了如何构建一个自定义的DAO层，甚至给出了一个简单的DAO工厂，它可作为你构建自定义DAO层的一个简单的起点。

到目前为止，我们已经学完了iBATIS代码中包含的所有东西了。第12章将介绍如何扩展iBATIS框架以便完成一些仅仅通过配置是不可能完成的事情。

第 12 章

扩展iBATIS

12

本章内容

- 自定义类型处理器
- 高速缓存控制器
- 自定义数据源

没有哪一个框架可以满足所有人的开箱即用的需求，这就是为什么框架必须提供扩展点以便用户修改框架的行为——换句话说，框架必须是可插入（pluggable）的，这一点非常重要。

尽管iBATIS是一个开源软件，其用户可以很容易地修改代码以便处理他们所需要做的任何事情，但是提供一种一致的、为iBATIS支持的扩展框架的方式仍然十分重要。否则，每当iBATIS发布一个新的版本，那些已经修改了其iBATIS副本的用户就必须将其修改合并到新版本中，然后重新编译整个框架。

iBATIS的定制（customization）可分为好几个级别。首先，作为一种常用的最佳实践，iBATIS在框架的各个设计层之间全面使用了接口。这意味着，即使框架不支持一种标准的扩展方式，终端用户所需要做的大部分工作，就是自己实现接口并用它来代替标准实现。我们将在本章稍后部分讨论SqlMapClient接口的定制时，给出一个相关的例子。

对于那些在每个应用程序、每个平台上都较有可能需要定制的功能组件，诸如类型处理器、事务管理器、数据源以及高速缓存控制器等，iBATIS提供了一种标准的可插拔组件架构。

下面几节将逐一对这些功能组件进行详细的探讨。首先来讨论一下插件架构（plug-in architecture）中的一些基本概念。

12.1 理解可插拔组件的设计

一个可插拔组件的设计通常由3部分组成：

- 接口

- 实现
- 工厂

接口描述了预定义的功能，它是所有实现必须遵循的规范，它描述了该功能组件要做什么。

实现是用来描述功能组件如何工作的具体行为。它可能依赖于第三方框架，甚至大型的基础设施，例如一个高级的高速缓存系统或者应用程序服务器。

工厂负责根据配置将实现和接口绑定。整体想法就是要确保应用程序只依赖于框架暴露出的那些单一且一致的接口。如果应用程序仍然需要依赖于第三方的实现细节，那么框架就不能很好地完成其工作。

图12-1描述了这一点。其中的箭头可以被看作“依赖于”，或者至少是“知道”。

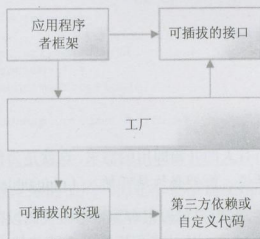


图12-1 可插拔框架设计示例

如前所述，iBATIS对很多功能组件都采用了可插拔组件设计。但是这究竟意味着什么呢？通常，一个插件就是对某个应用程序或框架的一个扩展，它可以带来以前不曾存在的功能，也可以用另一种不同的功能替换现存的功能。大多数情况下，iBATIS扩展都涉及现存功能的替换。

iBATIS的设计采用了分层架构，并且在每一层中都可能提供了一些扩展点，以便于你插入自己的功能。表12-1描述了iBATIS的几个扩展点，并且从总体上总结了每层的可扩展性。

表12-1 分层扩展摘要

可扩展的功能组件	描述
TypeHandlerCallback	实现你自己的类型处理逻辑，以便处理非标准数据库、驱动程序和（或）数据类型
CacheController	用你自己的高速缓存代码实现CacheController，或者为第三方高速缓存方案提供支持
DataSourceFactory	提供任何标准的JDBC DataSource实现
TransactionConfig	实现一个自定义事务管理器，以便最佳地适应环境

下面几节将详细描述表中的每一项。首先从最常见的扩展点TypeHandlerCallback开始。

12.2 使用自定义类型处理器

尽管我们都希望所有的关系数据库管理系统（RDBMS）是标准化的，但可惜的是，它们都不是标准化的。它们往往都实现了各自的SQL扩展，以及自己的数据类型。虽然大部分关系数据库现在都支持那些公共数据类型，例如二进制大对象（binary large object，BLOB）和字符大对象（character large object，CLOB），但是通常每个数据库驱动程序都以不同的方式来处理它们。因此iBATIS框架很难通过一个类型处理器实现来支持所有的数据库。为了应对这种情况，iBATIS支持自定义类型处理器，它可以允许你自定义特定类型的处理方式。使用自定义类型处理器，可以告诉iBATIS如何把某个关系数据库类型映射到Java类型。甚至还可以覆盖内置的类型处理器。本节将解释如何完成此项任务。

12.2.1 实现自定义类型处理器

要实现自定义TypeHandler，你只需要实现部分功能即可。此功能定义在一个被称为TypeHandlerCallback的简单接口中。如代码清单12-1所示。

代码清单12-1 TypeHandlerCallback

```
public interface TypeHandlerCallback {

    public void setParameter(
        ParameterSetter setter, Object parameter)
        throws SQLException;

    public Object getResult(ResultGetter getter)
        throws SQLException;

    public Object valueOf(String s);

}
```

下面逐步分析TypeHandlerCallback的实现过程。为了更好地说明此例，假设有一个数据库使用单词“YES”和“NO”来代表布尔值（例如，分别代表true和false）。表12-2给出了这个例子。

表12-2 使用YES和NO来代表布尔值

UserID	Username	PasswordHashcode	Enabled
1	asmith	1190B32A35FACBEF	YES
2	brobertson	35FACBEFAF35FAC2	YES
3	cjohnson	AF35FAC21190B32A	NO

假设我们将此表映射到如下所示的一个类上：

```
public class User {
    private int id;
    private String username;
    private String passwordHashcode;
    private boolean enabled;
```

```
// assume compliant JavaBeans properties  
// (getters/setters) below  
}
```

注意此处数据类型是不匹配的。在数据库中，Enabled列是一个VARCHAR类型，存储值为YES和NO，而在Java类中却是一个布尔类型。我们不能直接把一个YES或NO值设置为布尔类型。因此，需要将它进行转换。由JDBC驱动程序为我们完成这种转换当然是可能的，但现在假定事实并非如此。

TypeHandlerCallback接口就是用来处理这种情况的。因此下面就来编写一个完成此任务
TypeHandlerCallback接口实现。

12.2.2 创建 TypeHandlerCallback

正如我们所见，TypeHandlerCallback接口实际上非常简单。所需要做的就是创建一个实现该接口的类。我们将把这个新类取一个很好的描述性的名字，同时包含一对私有常量：

```
public class YesNoTypeHandlerCallback  
    implements TypeHandlerCallback {  
  
    private static final String YES = "YES";  
    private static final String NO = "NO";  
  
    public void setParameter(  
        ParameterSetter setter, Object parameter)  
        throws SQLException {  
    }  
  
    public Object getResult(ResultGetter getter)  
        throws SQLException {  
    }  
  
    public Object valueOf(String s) {  
    }  
}
```

以上仅仅是类型处理器的一个骨架实现，下面几节将逐步地充实它。

1. 设置参数

向数据库传送数据时，它必须是YES或者NO。此时，null是无效的。从Java类，我们将得到一个强类型布尔值true或者false。因此需要把true转换为YES值，并且把false转换为NO值。可以使用类似下面这个简单的方法来完成此任务：

```
private String booleanToYesNo(Boolean b) {  
    if (b == null) {  
        throw new IllegalArgumentException (  
            "Could not convert null to a boolean value. " +  
            "Valid arguments are 'true' and 'false'.");  
    } else if (b.booleanValue()) {  
        return YES;  
    } else {  
        return NO;  
    }  
}
```



```
}  
}
```

现在就可以在设置参数之前使用这个方法来进行参数值的转换。设置参数非常简单。TypeHandlerCallback接口的setParameter()方法接受两个参数。第一个参数ParameterSetter，使你能够访问很多设置方法，每一种方法适用于不同的数据类型。例如，setString()方法、setInt()方法以及setDate()方法。此类方法很多，这里没有完整地列出，但是请放心，几乎任何一种你所熟悉的Java数据类型都有相对应的设置方法。在我们的例子中，数据库表中的数据类型是VARCHAR，因此我们将使用ParameterSetter的setString()方法。

第二个参数是传递到数据库中的需要被转换的值。在上例中，我们将从User类的enabled特性中接收到一个布尔值。以下就是setParameter()方法的代码，该方法使用了之前所编写的便利的booleanToYesNo()方法：

```
public void setParameter(  
    ParameterSetter setter, Object parameter  
) throws SQLException {  
    setter.setString(booleanToYesNo((Boolean) parameter));  
}
```

此方法的内容体仅仅使用了ParameterSetter来设置由我们的便利方法转换得到的字符串值。我们必须将输入参数强制转换为Boolean，因为Type Handler Callback是可以支持任意类型的接口。

这非常简单，不是吗？不过正如你将在下一节中所看到的，获取结果也一样的简单。

2. 获取结果

当从数据库中得到YES或者NO值时，就需要把它转化为布尔值true或者false。这就是我们在设置参数时所进行的操作的反向操作。因此，为何不以相同的方式开始呢？下面来构建一个方法用于将字符串类型转化为布尔类型，如下所示：

```
private Boolean yesNoToBoolean(String s) {  
    if (YES.equalsIgnoreCase(s)) {  
        return Boolean.TRUE;  
    } else if (NO.equalsIgnoreCase(s)) {  
        return Boolean.FALSE;  
    } else {  
        throw new IllegalArgumentException (  
            "Could not convert " + s +  
            " to a boolean value. " +  
            "Valid arguments are 'YES' and 'NO'." );  
    }  
}
```

现在就可以使用这个方法将数据库返回的字符串结果转换为我们所需的布尔值了。为此，可以在TypeHandlerCallback的getResult()方法中调用这个新的转换方法。getResult()方法只有一个参数ResultGetter，该参数包含了很多用于获取不同类型值的方法。在此例中，我们需要获取一个字符串值。以下就是实现getResult()方法的代码：

```
public Object getResult(ResultGetter getter)
    throws SQLException {
    return yesNoToBoolean(getter.getString());
}
```

在上面的代码中，我们调用ResultGetter的getString()方法，以便将数据库值作为字符串返回。然后将该返回值传递给便利转换方法，该方法将返回我们最终在User类的enabled特性中设置的布尔值。

3. 处理空值：valueOf()方法究竟是干什么用的

iBATIS有一个空值转换功能，允许你在数据库中使用一个可为空的列，但却不需要对象模型中的可为空类型。当你对对象模型或者数据库没有完全的设计控制权但又必须为它们两者建立映射关系时，这个功能就非常有价值。举个例子，如果你的Java类有一个int类型的特性，该特性显然不允许为空，但你如果必须将它映射到一个可为空的数据库列，那么就必须用一个常量来代表这个空值。通常这个常量被称为“神奇数字（magic number）”，一般说来，使用“神奇数字”是一个非常糟糕的设计。但有时候，你没有任何选择——并且在某些情况下它还的确非常有意义。

因为iBATIS是使用XML文件进行配置的，因此数据库空值替代被指定为一个字符串。例如：

```
<result property="enabled" column="Enabled" nullValue="NO"/>
```

因此，必须为该替代值执行一些转换以使其变为一个实际的Java类型。iBATIS通过TypeHandlerCallback接口的valueOf()方法完成以上转换。在这个例子中，我们需要将字符串NO转换为布尔类型的false。幸运的是，这样一个转换通常与之前在获取结果时所做的转换是相似的。事实上，对YesNoTypeHandlerCallback来说，这个转换是完全一样的。因此该方法的实现如下：

```
public Object valueOf(String s) {
    return yesNoToBoolean(s);
}
```

就是这些了。我们已经完成了自定义类型处理器。代码清单12-2包含完整的源代码。

代码清单12-2 一个自定义类型处理器

```
public class YesNoTypeHandlerCallback
    implements TypeHandlerCallback {

    public static final String YES = "YES";
    public static final String NO = "NO";

    public void setParameter(
        ParameterSetter setter, Object parameter
    )
        throws SQLException {
        setter.setString(booleanToYesNo((Boolean)parameter));
    }

    public Object getResult(ResultGetter getter) {
```

数据库中YES和NO对应的常量

使用类型处理器设置参数

使用类型处理器获取结果

```

        throws SQLException {
            return yesNoToBoolean(getter.getString());
        }

        public Object valueOf(String s) { ◀—— 将字符串转换为我们的类型
            return yesNoToBoolean(s);
        }

        private Boolean yesNoToBoolean(String s) { ◀—— 将字符串转换为布尔类型
            if (YES.equalsIgnoreCase(s)) {
                return Boolean.TRUE;
            } else if (NO.equalsIgnoreCase(s)) {
                return Boolean.FALSE;
            } else {
                throw new IllegalArgumentException (
                    "Could not convert " + s +
                    " to a boolean value. " +
                    "Valid arguments are 'YES' and 'NO'." );
            }
        }

        private String booleanToYesNo(Boolean b) { ◀—— 将布尔类型转换为字符串
            if (b == null) {
                throw new IllegalArgumentException (
                    "Could not convert null to a boolean value. " +
                    "Valid arguments are 'true' and 'false'." );
            } else if (b.booleanValue()) {
                return YES;
            } else {
                return NO;
            }
        }
    }
}

```

现在完成了TypeHandlerCallback实现，下一节将讨论如何对它进行注册，因为只有经过注册，自定义类型处理器才能使用。

12.2.3 注册 TypeHandlerCallback 以供使用

要使用TypeHandlerCallback，我们需要执行一些步骤以指定在哪里以及何时使用它。共有3个可选项：

- 在SqlMapConfig.xml文件中注册，以使它全局可用。
- 在某个SqlMap.xml文件中注册，以使它局部可用。
- 专门针对某个结果映射或参数映射进行注册。

要将TypeHandlerCallback注册为全局可用，只需在SqlMapConfig.xml文件中简单地添加一个<typeHandler>元素。以下代码给出了<typeHandler>元素完整的示例：

```

<typeHandler
    callback="com.domain.package.YesNoTypeHandlerCallback"
    javaType="boolean" jdbcType="VARCHAR" />

```


<typeHandler>元素接受两个或三个属性。第一个属性是TypeHandlerCallback类本身，可以指定全限定名，或者如果你喜欢，也可以使用类型别名以使你的配置文件可读性更强。第二个是javaType属性，该参数指定了当前TypeHandlerCallback需要处理的Java类型。最后，第三个属性（是可选的），允许你指定当前TypeHandlerCallback需要应用到的JDBC（例如，数据库）类型。因此在这个例子中，我们使用的Java类型是布尔，JDBC类型是VARCHAR。如果没有指定数据类型，那么该类型处理器将被所有布尔类型所使用，但不会覆盖那些更明确地指定了JDBC类型的处理器。因此那些注册得越明确，同时匹配Java类型和JDBC类型的类型处理器将被使用。

自定义类型处理器目前来说是iBATIS最常见的扩展形式。这主要是因为关系数据库系统支持的非标准功能和非标准数据类型实在不少。在后面几节中，我们将讨论其他形式的扩展，这些扩展虽然很少使用，但了解它们仍然是非常有用的。

12.3 使用 CacheController

iBATIS包括好几个内建的高速缓存实现。之前的几章中已经讨论过这些实现。但为了唤醒你的记忆，表12-3对这几个高速缓存实现进行了总结。

表12-3 高速缓存实现总结

类	描 述
LruCacheController	最近最少使用的高速缓存会追踪最近访问的高速缓存项。当需要为新的缓存项腾空间时，最近最少访问的那条缓存项将先被移除。
FifoCacheController	当需要为新的缓存项腾空间时，先进先出缓存简单地会将高速缓存中最先进入的那条缓存项移除。
MemoryCacheController	内存缓存项由Java内存模型和垃圾收集器来决定缓存项应该在何时被移除。
OSCacheController	OpenSymphony高速缓存是一个称为OSCache的非常先进的第三方缓存解决方案的适配器。OSCache支持许多自己的缓存项模型以及像分布缓存这样的特性。

iBATIS提供了一个称为CacheController的接口，以允许你实现一个完全属于自己的自定义高速缓存解决方案，或者为框架插入一个现存的第三方缓存解决方案。CacheController接口非常简单，如下所示：

```
public interface CacheController {
    public void configure(Properties props);
    public void putObject(CacheModel cacheModel,
                        Object key, Object object);
    public Object getObject(CacheModel cacheModel,
                        Object key);
    public Object removeObject(CacheModel cacheModel,
                        Object key);
    public void flush(CacheModel cacheModel);
}
```

以下几节将带着你学习一个高速缓存实现示例。这并不意味着我们要教会你如何写一个用于企业应用程序的高速缓存控制器。我们的这个简单的高速缓存将仅仅使用一个Map来存储高速

缓存数据。使用自定义高速缓存一个更加普遍的情况是，插入一个自身就具有高级功能的第三方高速缓存。我们不推荐你将此例作为高速缓存策略的备选项。

12.3.1 创建 CacheController

CacheController实现往往都是从配置开始的。配置信息是通过实现configure()方法来获取的，这个方法以一个Java的Properties实例作为参数，该实例中包含所有的相关配置信息。对我们的高速缓存来说，我们并不需要任何配置特性，但需要一个用来存储高速缓存对象的映射。以下代码是我们实现CacheController的一个起点：

```
public class MapCacheController {  
  
    private Map cache = new HashMap();  
  
    public void configure(Properties props) {  
        // There is no configuration necessary, and therefore  
        // this cache will depend upon external  
        // flush policies (e.g. time interval)  
    }  
  
    // other methods implied ...  
}
```

好了，我们现在已经有了一个高速缓存模型的骨架了，开始丰富它的血肉吧。

12.3.2 CacheController 的放入、获取以及清除操作

现在可以开始思考如何向高速缓存中添加对象了。iBATIS的CacheModel会替我们管理所有的键，确定如何区分不同的语句调用以及结果集。所以，要向高速缓存中放入对象，你要做的就是将键和相应对象传递给所选择的高速缓存实现。

举个例子，以下就是我们的放入、获取和移除方法：

```
public void putObject(CacheModel cacheModel, Object key,  
                    Object object) {  
    cache.put (key, object);  
}  
  
public Object getObject(CacheModel cacheModel,  
                      Object key) {  
    return cache.get(key);  
}  
  
public Object removeObject(CacheModel cacheModel,  
                          Object key) {  
    return cache.remove(key);  
}
```

注意每个方法是如何提供控制该缓存的CacheModel实例的访问的，这就使你可以访问CacheModel中所可能需要的任何特性。参数key是一个CacheKey的实例，CacheKey是iBATIS中

一个特别的类，用于比较传递到语句中的参数集。在大多数情况下，我们不需要对它进行任何操作。就putObject()方法而言，参数object包含了将要放入高速缓存的实例或对象集合。

CacheModel描述的最后一个方法就是flush()方法。这个方法简单地将整个高速缓存清空。

```
public void flush(CacheModel cacheModel) {  
    cache.clear();  
}
```

以上就是一个虽然简单，但是却功能完备的CacheController实现。下面需要学习如何使用这个CacheController。

12.3.3 注册 CacheController 以供使用

像所有其他的iBATIS配置一样，CacheModel和CacheController也是以XML配置文件的形式进行配置的。开始使用CacheModel的最简单方法就是，首先为这个新类声明一个类型别名。这会为你节省很多随后的代码输入工作。

```
<typeAlias alias="MapCacheController"  
            type="com.domain.package.MapCacheController"/>
```

这样我们就为自己节省了大量的代码输入工作了。在定义<cacheModel>时，可以像使用其他的高速缓存模型类型一样应用这个高速缓存控制器类型。如下所示：

```
<cacheModel id="PersonCache" type="MapCacheController" >  
    <flushInterval hours="24"/>  
    <flushOnExecute statement="updatePerson"/>  
    <flushOnExecute statement="insertPerson"/>  
    <flushOnExecute statement="deletePerson"/>  
</cacheModel>
```

以上就是完成了一个自定义高速缓存的实现。但是请记住，这只是一个非常简单的示例而已。写一个自定义高速缓存所花费的时间可能比写一个应用程序还要多，因此，我建议你还是好好地研究其他第三方高速缓存方案，找出满足你需求的方案然后将其插入到iBATIS中。

12.4 配置 iBATIS 不支持的 DataSource

iBATIS包括对那些最常用的DataSource方案的支持，如JNDI（应用程序服务器管理的DataSource），Apache的DBCP，以及一个内建的名为SimpleDataSource的DataSource实现。也可以添加对其他DataSource实现的支持。

要配置一个新的DataSource实现，你需要为iBATIS提供一个工厂，这个工厂可以为框架提供DataSource实例。这个工厂类必须实现DataSourceFactory接口，如下所示：

```
public interface DataSourceFactory {  
  
    public void initialize(Map map);  
    public DataSource getDataSource();  
  
}
```


DataSourceFactory接口中仅仅包含两个方法：一个用于初始化DataSource，另一个用于访问该DataSource。框架会为initialize()方法提供一个Map实例，该实例包括各种配置信息，诸如JDBC驱动程序名称、数据库URL，以及用户名和密码。

getDataSource()方法仅仅需要返回已配置妥当的数据源。DataSourceFactory是一个简单的接口，其实现的复杂程度与你插入到其中的DataSource实现相当。以下是一个来自iBATIS源代码的示例。这是针对SimpleDataSource实现的一个DataSourceFactory。如你所见，它确实非常简单。

```
public class SimpleDataSourceFactory
    implements DataSourceFactory {

    private DataSource dataSource;

    public void initialize(Map map) {
        dataSource = new SimpleDataSource(map);
    }

    public DataSource getDataSource() {
        return dataSource;
    }
}
```

正像之前说过的那样，复杂的DataSource实现可能会需要更多的工作，不过我们希望这不会成为你需要担心的问题。

关于iBATIS扩展，我们将讨论的最后一个主题就是如何定制事务管理。

12.5 定制事务管理

正如前几章所介绍的，iBATIS提供了很多事务选项。然而，今天的应用程序服务器以及事务管理的定制手段所涉及的范围非常广，使得事务管理仍然有需要定制的空间。从外部看，事务似乎非常简单，它仅仅提供了几个功能：开始、提交、回滚以及结束。但从内部来看，事务是非常复杂的，每个应用程序服务器的行为都稍有不同，且与标准存在一定的偏离。正因为如此，iBATIS允许你定制自己的事务管理系统。如果有过开发事务管理系统的经验，那么你可能一想到它就会觉得浑身发颤——的确如此。要想正确地实现事务管理器是一件非常非常困难的事情。因此，我们不会在本书中给出一个真实的实现，而只是详细地讨论这个接口，如果你有一个实现该接口的任务，那么以下的内容对你来说至少是一个很好的开始。如果你确实希望能看到一个接口实现的示例，那么可以参考iBATIS中的3个实现：JDBC、JTA和EXTERNAL。表12-4总结了这几个实现，以防你在之前几章中错过了它们。

表12-4 内建的事务管理器配置

实 现	描 述
JdbcTransactionConfig	使用了JDBC Connection API提供的事务基础设施
JtaTransactionConfig	或者开始一个全局事务，或者加入一个现存的全局事务
ExternalTransactionConfig	提交和回滚的无操作实现，从而将它们留给了外部的事务管理器

大多数情况下，表12-4中的选项应该就能够满足需求了。但是如果使用的应用程序服务器或者事务管理器不是标准的（或存在bug），iBATIS也提供了两个接口TransactionConfig和Transaction，以便于你创建自己的事务管理适配器。通常需要同时实现这两个接口以构成一个完整的实现，除非你的情况允许从其他实现中复用某个现成的Transaction类。

12.5.1 理解 TransactionConfig 接口

TransactionConfig接口可以被认为是一种工厂，但它主要负责为事务实现配置事务设施（transaction facility）。该接口如下所示：

```
public interface TransactionConfig {  
  
    public void initialize(Properties props)  
        throws SQLException, TransactionException;  
  
    public Transaction newTransaction(int  
        transactionIsolation)  
        throws SQLException, TransactionException;  
  
    public int getMaximumConcurrentTransactions();  
  
    public void setMaximumConcurrentTransactions(int max);  
  
    public DataSource getDataSource();  
  
    public void setDataSource(DataSource ds);  
  
}
```

第一个方法是initialize()。正如我们在扩展框架的其他部分时所看到的那样，该方法用于配置事务设施。它以一个Properties实例作为唯一的参数，该Properties实例中包括了所有的配置选项。例如，JTA实现需要一个从某个JNDI树中检索到的UserTransaction实例。因此传递给JTA实现的特性之一就是指向该实现所需的UserTransaction的JNDI路径。

第二个方法是newTransaction()方法。这是一个用于创建新的事务实例的工厂方法。它有一个int类型（为保证类型安全，其实此处应该使用一个枚举类型）的参数，描述了该事务应该在其中表现出的事务隔离等级。可用的事务隔离级别定义于JDBC Connection类中，包括如下常量：

- TRANSACTION_READ_UNCOMMITTED
- TRANSACTION_READ_COMMITTED
- TRANSACTION_REPEATABLE_READ
- TRANSACTION_SERIALIZABLE
- TRANSACTION_NONE

以上所有这些隔离级别在JDBC Connection API中都有文档描述，相关的更多内容请参考第7章。这里需要强调的一点就是，如果你的事务管理器实现不支持以上级别中的一个或几个，就

应该抛出相应的异常以便让开发人员了解此情况。否则，用户可能会遇到很难调试的意外后果。

接下来的一对方法是`getDataSource()`和`setDataSource()`。这两个方法描述了与`TransactionConfig`实例关联的`DataSource`的Java Beans特性。通常你不需要对`DataSource`做任何特别的事，但是你可以在需要时用额外的行为去修饰（decorate）它。许多事务管理器实现都为其提供的`DataSource`对象和`Connection`对象进行了包装，为它们加上了事务相关的功能。

最后一对方法描述了与`TransactionConfig`实例关联的另一个Java Beans特性，该特性允许框架对其支持的最多数目的并发事务进行配置。你的实现可能是（也可能不是）可配置的，如果它不可配置，那么当并发事务的最大数目超过了系统所能处理的上限时，请确保要抛出一个适当的异常，这非常重要。

12.5.2 理解 Transaction 接口

回想一下上一节中讨论的`TransactionConfig`类上的`newTransaction()`工厂方法。该方法返回值是一个`Transaction`实例。相应的`Transaction`接口描述了在iBATIS框架中要支持事务所需的行为。该接口定义的功能集合非常典型，任何人只要以前使用过事务，肯定就会对它非常熟悉。`Transaction`接口如下所示：

```
public interface Transaction {  
  
    public void commit() throws SQLException,  
        TransactionException;  
  
    public void rollback() throws SQLException,  
        TransactionException;  
  
    public void close() throws SQLException,  
        TransactionException;  
  
    public Connection getConnection()  
        throws SQLException, TransactionException;  
  
}
```

这个典型的接口实在没有什么特别的地方。只要使用过事务，就应该对它非常熟悉。正如你所期望的那样，`commit()`方法用于提交事务，使得对工作单元所做的全部更改永久化。另一方面，`rollback()`方法用于回滚事务，使得已经在工作单元中生效的全部更改同时撤销。`close()`方法负责释放所有分配或预留给当前事务的资源。

最后一个方法`getConnection()`，是你可能没有想到会出现在该接口中的一个方法。从设计上来说，iBATIS是JDBC API之上的一个高层框架。不那么严格地说，在JDBC中，数据库连接就是事务。至少，事务是在JDBC数据库连接这样一个级别上被管理、控制以及理解的。基于这样的原因，大多数事务的实现都会与一个数据库`Connection`实例绑定。这很有用，因为iBATIS需要访问与当前事务关联的连接。

12.6 小结

本章中我们探索了扩展iBATIS框架的几种方式。使用标准的方式进行扩展非常重要，即使是对于开源软件，因为只有这样才能避免那些不受控的框架定制行为（这些行为可能是iBATIS框架不期望的，因此可能不具有可维护性）。iBATIS支持许多不同的扩展，包括TypeHandler-Callback、CacheController、DataSourceFactory和TransactionConfig。

TypeHandlerCallback可能是一种最常见的扩展方式了，因为当你使用了一些私有数据类型时，可以用它来解决因此造成的常见问题。实现TypeHandlerCallback接口非常简单，它只需要实现几个方法以允许在Java类型和JDBC类型之间进行映射的定制。简单地说，TypeHandlerCallback负责为已映射语句设置参数，从结果集中获取结果，以及转换空值替换以将可为空的数据库类型映射为不可为空的Java类型。

CacheController为把第三方高速缓存解决方案集成到iBATIS中提供了一种非常简单的方式。当然，你也可以写一个自己的高速缓存解决方案，但要写一个漂亮的高速缓存是相当困难的。CacheController接口包括很多方法，分别用于配置实现，将实现项放置在高速缓存中，从高速缓存中获取实现项，以及从高速缓存中移除或清除实现项等。

DataSourceFactory负责为标准的JDBC DataSource实现进行配置并提供对它的访问。推荐你实现这个接口来配置某些第三方DataSource以供使用；除非你有非常充足的理由，否则还是建议你尽量避免自己编写DataSource实现。DataSourceFactory只有两个方法：一个用于配置DataSource，另一个用于提供对该DataSource的访问。

TransactionConfig和Transaction接口实现起来最为复杂，幸好通常很少使用它们。需要一个自定义TransactionConfig的情况应该很少出现，不过万一确实需要，iBATIS也允许你自己写一个。

以上就是iBATIS所支持的标准扩展点。iBATIS在设计时大量使用了接口，以允许你替换现有功能。我们不可能在本章中向你展示所有的扩展点，这即使是一本书的篇幅也不足够，不过如果你查看代码，就会发现iBATIS在设计时为大多数的层提供了接口与实现的得体的分离。大多数情况下，iBATIS的设计将如我们在本章中所描述的一样。

Part 4

第四部分

iBATIS 使用秘诀

到此为止，你应该已经完全掌握了 iBATIS 的基本特征和高级特征，并且深刻理解应该如何以及何时使用这些特征。本书的第四部分（即最后一部分）将讨论最佳实践，并且在一个完整的样例应用中展示这些最佳实践，以此来总结全书。如果说一幅画可以抵得上千言万语，那么一个样例应用就可以抵得上成千上万行代码。我们希望你喜欢这个样例应用。

本部分 内容

- 第 13 章 iBATIS 最佳实践
- 第 14 章 综合案例研究

第 13 章

13

iBATIS最佳实践

本章内容

- 单元测试
- 配置管理
- 命名规范
- 数据结构

iBATIS就是有关最佳实践的——从很大程度上说，最佳实践正是创建iBATIS的原因所在。对于初学者而言，iBATIS可以帮助你保持应用程序和持久层之间的分离。它还可以帮助你避免把Java代码和SQL代码混淆在一起，从而避免了把代码弄得一团乱。此外，iBATIS还帮助你将面向对象的领域模型设计同数据库设计时的关系模型分离开。本章将讨论一些最佳实践，它们可以帮助你最大程度地利用iBATIS。

13.1 iBATIS 中的单元测试

单元测试已经成为了现代软件开发方法中的一个非常重要的组成部分。即使不赞成极限编程（cXtreme Programming，XP）或者其他敏捷方法能够带来好处，单元测试也应该成为你的软件开发生命周期中的一个基础实践。

从概念上说，持久层可以分为3层，而iBATIS使得对这些不同的层进行单元测试都变得非常简单，如图13-1所示。

iBATIS至少在以下3个方面可以使得你对这些不同层进行单元测试变得更容易：

- 测试映射层（mapping layer）本身，包括各个映射、所有的SQL语句，以及这些SQL语句被映射到的那些领域对象。

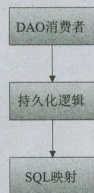


图13-1 图中都是和持久化直接相关的典型层（和持久性无关的层没有显示在此图中）

- 测试DAO层，这使你可以对DAO层中的任何特定于持久化的逻辑进行测试。
- 在DAO的消费层^①中进行测试。

13.1.1 对映射层进行单元测试

对映射层所进行的单元测试，可能是在大部分应用程序中所发生的最低层次的单元测试了。此过程包括对SQL语句以及这些语句所映射到的领域对象进行测试。这意味着我们将需要一个用于进行测试的数据库实例。

1. 测试用数据库实例

测试用数据库实例可能是创建于你实际使用的数据库管理系统（例如，Oracle或者微软的SQL Server）中的一个真实实例。如果你的环境对于单元测试是友好的，那么只需要简单地更改一下配置就可以运行单元测试了。如果打算使用非标准数据库特征（例如，存储过程），那么就可能有必要使用真实的数据库实例。存储过程和其他一些非可移植的数据库设计时选择，会使得对数据库进行单元测试变得很难，除非使用真实的数据库实例。

使用真实的数据库实例的缺点是，只有连接到网络才能进行单元测试。或者也可以使用某个真实数据库的一个本地实例，但这意味着单元测试在运行之前将需要额外的本地环境设置。无论使用这两种方法中的哪一种，你都将面对同一个问题，即每次测试都必须重建测试数据，甚至可能需要重建测试套件（test suite）之间的模式，或者是每个单元测试之间的模式。即使是在大型的企业级的数据库服务器上，要完成以上任务也需要花费很长时间。另一个问题是，由于使用的数据库是集权式的，多个开发人员同时进行单元测试时就可能会导致冲突。所以，必须使用不同的数据库模式来隔离每一个开发人员。正如你所见，这种方法的普遍问题就是，单元测试取决于相当多的基础设施，而这对于大多数经验丰富的测试驱动程序的开发人员来说是不够完美的。

Java开发人员是非常幸运的，因为他们至少有一种非常棒的内存（in-memory）数据库可以使用，这种数据库可以使对相对标准的数据库设计进行单元测试变得非常简单。HSQLDB是一个完全用Java写成的内存数据库。它既不需要磁盘上的任何文件也不需要连接网络就能够正常工作。此外，它还能够重新生成来自典型数据库（例如Oracle和微软的SQL Server）的大部分数据库设计。即使由于设计过于复杂（例如使用了存储过程）而导致HSQLDB不能重建整个数据库，它也能够重新生成该数据库的绝大部分。HSQLDB允许快速重建数据库，包括数据库模式和测试数据。iBATIS自己的单元测试套件就是使用HSQLDB在每个单独的测试之间重建数据库模式和测试数据。我们亲自使用HSQLDB测试了由将近1 000个数据库相关的测试构成的测试套件，运行时间不到30秒。

有关HSQLDB的更多信息，请访问网页<http://hsqldb.sourceforge.net/>。另外，可能会让微软

① 即使用层。——译者注

的.NET高兴的一个消息是，已经有人发起对HSQLDB的移植了，同时也有人开始创建其他的内存数据库了。

2. 数据库脚本

现在已经有了数据库实例，那么数据库模式和测试数据应该如何构造和创建呢？你可能已经有了可以用来创建数据库模式以及测试数据的数据库脚本了。理想情况下，你应该将这些脚本纳入到版本控制系统（例如CVS或者Subversion）中。这些脚本应该和应用程序中的其他代码一样被同等地对待。即使你对自己使用的数据库没有控制权，你也应该定期从拥有控制权的人那里获得相应的更新。应用程序的源代码与数据库脚本应该始终保持同步，并且单元测试本来就是确保它们同步的。每当运行单元测试套件时，你还应该运行这些脚本来重新创建数据库模式。使用这种方法，可以很容易地将数据库创建脚本需要的新集提交给版本控制系统，然后运行单元测试以便确定脚本更新是否给应用程序带来了问题。这是最理想的情况。如果使用内存关系数据库（例如HSQLDB）来运行测试，则可能需要另外一个步骤来转换数据库模式。可以考虑使这个转换过程自动化，以便避免手工编程可能出现的错误，加快集成的速度。

3. iBATIS配置文件（例如SqlMapConfig.xml）

为了进行单元测试，你可能想要使用一个独立的iBATIS配置文件。配置文件用于控制数据源和事务管理器的配置，它在测试环境和产品环境中可能会完全不同。例如，产品环境可能会是一个像J2EE应用程序服务器这样的受管理环境。在这样的环境下，一个受管理的DataSource实例可能是从JNDI中检索得到的。你还可能会在产品环境中利用全局事务。然而，在测试环境中，你的应用程序可能不会运行在服务器中；而只是配置了一个简单DataSource，使用的也是局部事务。分别进行测试环境配置和产品环境配置的最简单的方式就是，使用不同的iBATIS配置文件，这两份配置文件引用相同的一组SQL映射文件。

4. iBATIS SqlMapClient单元测试

现在所有的先决条件都已经准备好了，这些先决条件包括数据库实例、自动构建数据库的脚本，以及用于测试的配置文件，接下来可以开始创建单元测试了。代码清单13-1即是一个使用JUnit来创建简单单元测试的例子。

代码清单13-1 SqlMapClient单元测试示例

```
public class PersonMapTest extends TestCase {
    private SqlMapClient sqlMapClient;

    public void setup () {
        sqlMapClient = SqlMapClientBuilder.
            build SqlMapClient (Resources.getResource As Reader
            ("maps/TestSqlMapConfig.xml"));
        runSqlScript ("scripts/drop-person-schema.sql");
        runSqlScript ("scripts/create-person-schema.sql");
    }
}
```

① 设置单元测试和测试数据

```
runSqlScript("scripts/create-person-test-data.sql");
}

public void testShouldGetPersonWithIdOfOne() {
    Person person = (Person) sqlMapClient.
        queryForObject("getPerson", new Integer(1));
    assertNotNull("Expected to find a person.", person);
    assertEquals("Expected person ID to be 1.",
        new Integer(1), person.getId());
}
```

② 利用主键值测试单个
person对象的检索

代码清单13-1中的示例使用了针对Java的JUnit单元测试框架。（可以在www.junit.org上找到更多关于JUnit的信息。对于.NET Framework，也有相似的工具，包括NUnit，可以从www.nunit.org上下载得到。）在我们的设置方法中①，首先删除了测试涉及的那些数据库表，然后再重建它们并对它们重新填充测试数据。为每一个测试都重建所有的东西可以确保各个测试间相互独立，但是如果在像Oracle或者SQL Server这样的RDBMS上这样删删建建地做测试那就太慢了。在这种情况下，可以使用类似于HSQldb这样的内存关系数据库。在我们实际的测试案例中②，我们从数据库中读取一条记录，将其映射到一个bean上，然后断言（assert）bean中各字段的值都是所预期的值。

以上就是测试映射层所需做的全部工作了。接下来要测试的层是DAO层，假定你的应用程序有DAO层。

13.1.2 对 DAO 进行单元测试

DAO层是一个抽象层，因此根据其本质，DAO应该非常容易测试。DAO也使得对DAO层用户的测试变得更加简单。本节，将讨论测试DAO本身。DAO通常被分离为一个接口和一个实现。由于我们将直接测试DAO，因此接口将不起作用。我们将直接对DAO实现进行测试。这可能同DAO模式的工作方式是相反的，但是这恰好体现了单元测试的好处——它把那些坏习惯清除出我们的系统！

如果可能的话，对DAO层的测试应该不涉及数据库以及底层的基础设施。DAO层是持久化实现的一个接口，但是在测试DAO层时，我们更感兴趣的是测试DAO层内部的东西，而不是测试DAO层之外的东西。

测试DAO的复杂度仅仅取决于DAO实现。例如，测试一个JDBC DAO可能非常困难。你需要一个很好的模拟框架来代替所有典型的JDBC组件，如Connection、ResultSet和PreparedStatement等。即使是这样，要利用模拟对象来管理这样复杂的API也非常麻烦。而模拟iBATIS SqlMapClient接口则简单得多。下面就来试一试。

1. 利用模拟对象来对DAO进行单元测试

模拟对象是指为了进行单元测试而用来替换实际实现的对象。模拟对象通常没有很强的功能

性：它们只用于满足某个单一的情况，以使得单元测试仅仅关注其应该关注的部分，而不需要担心复杂度的增加。我们将在下面的例子中使用这些模拟对象来示范一种测试DAO层的方法。

在我们的示例中，将使用一个简单的DAO。我们将不考虑iBATIS DAO框架，因此就不需要担心事务以及诸如此类的东西了。这个示例的目的就是，示范如何测试DAO层，无论你使用的是哪一种DAO框架（只要确实使用了它）。

首先，来考察一下要进行测试的DAO。代码清单13-2给出了一个SqlMapPersonDao实现，它调用了和一个和13.1.1节中给出的示例相似的SQL映射文件。

代码清单13-2^① 测试一个简单的DAO

```
public class SqlMapPersonDao implements PersonDao {  
  
    private SqlMapClient sqlMapClient;  
  
    public SqlMapPersonDao(SqlMapClient sqlMapClient) {  
        this.sqlMapClient = sqlMapClient;  
    }  
  
    public Person getPerson (int id) {  
        try {  
            return (Person)  
                sqlMapClient.queryForObject("getPerson", id);  
        } catch (SQLException e) {  
            throw new DaoRuntimeException(  
                "Error getting person. Cause: " + e, e);  
        }  
    }  
}
```

请注意在代码清单13-2中我们是如何把SqlMapClient注入到DAO的构造函数中的。这为对DAO进行单元测试提供了一种简易的方式，因为我们只需要模拟SqlMapClient接口就可以了。显然，这是一个非常简单的示例，没有对其进行太多的测试，但是每一个测试都非常重要。代码清单13-3显示了用来模拟SqlMapClient并且测试getPerson()方法的单元测试。

代码清单13-3 含有模拟SqlMapClient的PersonDao单元测试

```
public void testShouldGetPersonFromDaoWithDoOne() {  
    final Integer PERSON_ID = new Integer(1);  
  
    Mock mock = new Mock(SqlMapClient.class);  
    mock.expects(once())  
        .method("queryForObject")  
        .with(eq("getPerson"), eq(PERSON_ID))
```

① 原书代码有错。代码第6行~第7行应改为：`sqlMapClient = SqlMapClientBuilder.buildSqlMapClient(Resources.getResourceAsReader("maps/TestSqlMapConfig.xml"));`。——译者注

```
.will(returnValue(new Person (PERSON_ID)));

PersonDao daoSqlMap =
    new SqlMapPersonDao((SqlMapClient) mock.proxy());
Person person = daoSqlMap.getPerson(PERSON_ID);

assertNotNull("Expected non-null person instance.",
    person);
assertEquals("Expected ID to be " + PERSON_ID,
    PERSON_ID, person.getId());
}
```

代码清单13-3中给出的示例使用了JUnit以及JMock这个Java对象模拟框架。正如代码清单13-3中加粗的部分所显示的那样，利用JMock来模拟SqlMapClient接口实现，使我们能够单独测试DAO的行为，而无需顾虑实际的SqlMapClient实现，也就不需要考虑与其相关的SQL语句、XML文件，还有数据库了。JMock是一个非常好用的工具，可以在www.jmock.org上找到更多有关它的信息。你可能已经猜到了，还有一个针对.NET的模拟框架，称为NMock，有关它的更多信息请参考<http://nmock.org>。

13.1.3 对 DAO 的消费层进行单元测试

应用程序中那些使用DAO层的其他层称为DAO层的消费者（consumer）。DAO模式使得你可以在不依赖于持久层的任何功能的情况下测试这些消费者的功能。一个好的DAO实现应该有一个能够很好地描述其可用功能的接口。测试消费层的关键在于获得此接口。考察代码清单13-4中的接口，你就会发现上一节中所描述的getPerson()方法。

代码清单13-4 简单的DAO接口

```
public interface PersonDao extends Dao {
    Person getPerson(Integer id);
}
```

要开始测试DAO层的消费者，所需要的就只是代码清单13-4中所给出的接口。我们甚至根本不需要一个完整的实现。利用JMock，我们能够很轻松地模拟getPerson()方法所预期的行为。考虑如下这个使用了PersonDao接口的服务（见代码清单13-5）。

代码清单13-5 使用PersonDao接口的服务

```
public class PersonService {

    private PersonDao personDao;

    public PersonService(PersonDao personDao) {
        this.personDao = personDao;
    }

    public Person getValidatedPerson(Integer personId) {
```

```
Person person = personDao.getPerson(personId);

validateAgainstPublicSystems(person);
validateAgainstPrivateSystems(person);
validateAgainstInternalSystems(person);

return person;
}
}
```

我们的单元测试的目标并不是DAO——而是getValidatedPerson()方法中的业务逻辑，例如它所执行的各种验证。方法中的每一个验证可能都是一个私有方法，为了便于讨论，假设此处我们只测试私有接口。

多亏了之前的PersonDao接口，在没有数据库的情况下测试getValidatedPerson()方法也很容易。所需要做的只是模拟PersonDao接口实现，然后将该模拟实现传递给服务的构造函数，最后调用getValidatedPerson()方法即可。代码清单13-6给出了完成上述工作的单元测试。

代码清单13-6 使用模拟而不是真实的DAO，以避免访问数据库

```
public void testShouldRetrieveAValidatedPerson () {
    final Integer PERSON_ID = new Integer(1);

    Mock mock = new Mock(PersonDao.class);
    mock.expects(once())
        .method("getPerson")
        .with(eq(PERSON_ID))
        .will(returnValue(new Person(PERSON_ID)));

    PersonService service =
        new PersonService((PersonDao)mock.proxy());
    service.getValidatedPerson(
        new Integer(PERSON_ID));

    assertNotNull("Expected non-null person instance.",
        person);
    assertEquals("Expected ID to be " + PERSON_ID,
        PERSON_ID, person.getId());
    assertTrue("Expected valid person.",
        person.isValid());
}
```

我们再一次同时使用了JUnit和JMock。正如你在代码清单13-6中所见到的那样，这种测试方式在应用程序的各个层都是一致的。这是很有好处的，因为它可以带来易于维护的凝练的单元测试。

有关iBATIS中的单元测试，我们就介绍到这。实际上网上有很多很好的关于单元测试资源。使用Google搜索一下“unit test（单元测试）”，就可以找到很多相关的资源，它们可以帮助你迅速

提高单元测试的能力，甚至你还可能发现比本书使用的方法更好的单元测试方法。

13.2 管理 iBATIS 配置文件

相信阅读到此，大家都已经很清楚iBATIS是使用XML文件来进行配置以及语句映射的了。但随着文件的增多，它们可能很快就会变得难以处理。本节将讨论一些最佳实践，以帮助组织SQL映射文件。

13.2.1 将其保存在类路径上

位置透明（location transparency）是应用程序可维护性的一个重要方面。它可以简化应用程序的测试和部署。保持位置透明的一个方法就是，不要在应用程序中使用静态的文件路径，例如 `/usr/local/myapp/config/` 或者 `C:\myapp\`。虽然iBATIS允许使用特定的文件路径，但是最好还是使用类路径。当你不希望应用程序中出现任何特定的文件路径时，就可以使用Java类路径。可以把类路径看作一个微型文件系统，应用程序可以通过使用类加载器（classloader）在内部访问它。类加载器能够从类路径上读取资源，这些资源包括类和其他文件。下面将给出一个示例。首先假设类路径上有如下一个文件结构：

```
/org
  /example
    /myapp
      /domain
      /persistence
      /presentation
      /service
```

在这个结构下，可以使用全限定类路径 `org/example/myapp/persistence` 来访问持久包。一个很好地放置映射文件的位置就是 `org/example/myapp/persistence/sqlmaps`，该位置在此结构中以下的形式呈现：

```
/org
  /example
    /myapp
      /domain
      /persistence
        /sqlmaps
          SqlMapConfig.xml
          Person.xml
          Department.xml
      /presentation
      /service
```

另外，如果想为配置文件使用一个更浅的结构，可以考虑把这些映射文件放在一个普通的配置包中。例如，可以使用 `config/sqlmaps`，它对应的结构如下所示：

```
/config
  /sqlmaps
    SqlMapConfig.xml
    Person.xml
```

```
Department.xml
/org
/example
/myapp
/domain
/persistence
/presentation
/service
```

将映射文件存储在类路径上之后，iBATIS就可以通过Resources实用类^①来加载这些文件，这非常简单。Resources实用类包含了类似getResourceAsReader()这样的方法，该方法与SqlMapClientBuilder协同工作就可以得到一个SqlMapClient实例因此给定之前的类路径，就可以通过以下代码来加载SqlMapConfig.xml：

```
Reader reader = Resources
    .getResourceAsReader("config/maps/SqlMapConfig.xml");
```

如果发现所处的环境要求你必须将数据库资源配置文件放在一个集中位置（centralized location）上，例如一个固定的文件路径上，那么仍然应该将所有的SQL映射文件存储在类路径上。也就是说，应该使用一种混合方式：将SqlMapConfig.xml存储在某固定文件路径上，而仍然将SQL映射文件存储在类路径上。例如如下这个结构：

```
C:\common\config\
/sqlmaps
    SqlMapConfig.xml

/config
/sqlmaps
    Person.xml
    Department.xml
/org
/example
/myapp
/domain
/persistence
/presentation
/service
```

即使SqlMapConfig.xml是在一个固定的位置上，但在内部它仍然可以引用类路径上的XML映射文件。这使你可以将大部分的资源仍存储在你希望的地方，并且减少了映射文件部署不当的可能性。

13.2.2 集中放置文件

应该将所有的映射文件集中放置，避免将它们分散在类路径上。千万不要把它们放置在它们所处理的类的旁边，也不要将它们存储在单独的包中。这么做会使得你的配置变得复杂，并且使人很难弄清楚究竟有哪些映射文件可用。映射文件的内部结构使得对其进行进一步分类变得完全没有

① 该类是iBATIS框架提供的，jre中没有这个类。——译者注

必要。采用灵活的文件名，并且将所有文件放置在单个的目录下。最好避免把类放在同一个目录（即包）下，当然也不要把类和其他XML文件混淆在一起！

这种方法使你能够更容易地定位映射文件，从而总体上更好地定位整个工程。你把文件放在什么地方对于iBATIS框架来说，是没有任何区别的，但是对于你的合作开发人员来说区别就大了。

13.2.3 主要按返回类型来组织映射文件

有关映射文件的组织，最常见的问题就是按照什么来组织它们。是应该用数据库表来组织它们呢，还是根据类来组织，或者是根据语句的类型？

这个问题的答案取决于你的具体环境。虽然此问题没有一个“绝对正确的”答案，但你也不能因此就毫无依据地随便组织。iBATIS是非常灵活的，因此你总是可以将已映射语句到处移动，这非常简单。

刚开始时，最好利用语句的返回类型以及语句接受的参数类型来组织你的映射文件。这样组织得到的映射文件就可以根据希望查找的类型很快地进行定位。例如，可以在一个名为Person.xml的映射文件中，你应该期望找到那些返回Person对象（或者Person对象集合）的已映射语句，以及将Person对象作为参数的已映射语句（例如insertPerson或者updatePerson）。

13.3 命名规范

在iBATIS中有很多东西需要命名：语句、结果映射、参数映射、SQL映射文件，以及XML文件都需要命名。因此，最好能有一种命名规范。本书将讨论一种命名规范，但是你完全可以使用自己的规范。只要你在应用程序中始终遵循一致的命名规范，就不会有任何问题。

13.3.1 语句的命名

语句的命名通常应该遵循你所使用的编程语言中方法的命名规范。也就是说，在Java应用程序中，语句应该使用类似于loadPerson或者getPerson这样的名字。而在C#应用程序中，就应该使用类似于SavePerson或者UpdatePerson这样的名字。使用同样的命名规范将帮助你保持一致性，同时它也可以有助于方法绑定特征和代码生成工具。

13.3.2 参数映射的命名

大多数时候都不需要命名参数映射，这是因为我们通常采用的都是内联参数映射，而不是显式地定义一个参数映射。由于SQL映射的本质，参数映射的可复用性通常是有限的，一般情况下，对于INSERT语句和UPDATE语句，就无法使用同一个参数映射。因此，如果你确实使用了一个显式定义的参数映射，建议在使用该参数映射的语句的名字后添以Param后缀作为该参数映射的名字。如下所示（注意其中的粗体部分）：

```
<select id="getPerson" parameterMap="getPersonParam" ... >
```


13.3.3 结果映射的命名

结果映射通常是与某特定类类型绑定的，具有较好的可复用性。因此建议使用结果映射所绑定到的类型名字添加以Result后缀来作为结果映射的名字。例如：

```
<resultMap id="PersonResult" type="com.domain.Person">
```

13.3.4 XML 文件的命名

在iBATIS中有两类XML文件。第一类是主配置文件，第二类则是SQL映射文件。

1. 主配置文件

主配置文件可以命名为任何你喜欢的名字，但推荐SqlMapConfig.xml。如果应用程序的不同模块有不同的配置文件，那么将该模块的名字作为前缀添加到配置文件名中去。例如，如果你的应用程序有一个Web客户端和一个GUI客户端，它们需要不同的配置，那么就可以使用WebSqlMapConfig.xml和GuiSqlMapConfig.xml这两个名字。可能有多个不同的环境需要部署，如产品环境和测试环境。在这种情况下，可以在文件名前添加环境的类型作为前缀。继续之前那个示例，就可以使用ProductionWebSqlMapConfig.xml和TestWebSqlMapConfig.xml这两个名字。这些名字都是描述性的，其一致性可以使得不同环境下的自动构建成为可能。

2. SQL映射文件

如何命名SQL映射文件很大程度上取决于你的已映射语句的组织方式。之前曾推荐你根据已映射语句的返回类型和参数将已映射语句划分到不同的XML文件中。如果你的确是那么做的，那么在返回类型或参数后命名文件就可以了。例如，如果一个映射文件包含的SQL语句涉及Person类，那么将该文件命名为Person.xml就非常合适。在大多数应用程序中使用这种命名方式就可以了。但也有一些其他情况需要考虑。

某些应用程序可能需要对同一个已映射语句做多个实现以匹配不同的数据库。大多数情况下，我们编写的SQL语句都是可移植的。例如，最初那个使用了iBATIS的JPetStore应用程序可以与11个不同的数据库兼容。但是，有时数据库的某些特性的确是不可移植的，而它对我们的解决方案来说又是最理想的。在类似这样的情况下，命名你的映射文件以包含我们特别编写的数据库就是可接受甚至非常重要的了。举个例子，如果我们有一个特定于Oracle的person文件，那么就可以将其命名为OraclePerson.xml。还有一种方式就是针对每种数据库使用一个在其后面命名的不同目录。不过千万别滥用这种方式。仅仅用这种方式命名那些确有需要文件和目录，并且保证该映射文件中的确有足够多的Oracle相关的东西以使它名副其实。如果文件中仅有一条与Oracle相关的语句，那么你还不如仅在该语句的名字中包含单词Oracle。

13.4 Bean、map 还是 XML

iBATIS支持多种类型的参数映射和结果映射。可以选择使用JavaBean、Map（如HashMap）、XML，当然还有基本类型（primitive）。到底应该将你的语句映射为哪种类型呢。默认情况下总

是推荐JavaBean。

13.4.1 JavaBean

JavaBean具有最好的性能、最大的灵活性和类型安全性。JavaBean所以快是因为它在特性映射中使用的是最简单的底层方法调用。JavaBean在你为其添加了更多的特性时并不会因此降低性能，并且它比其他的各选项具有更好的内存效率。一个更重要的原因就在于JavaBean是类型安全的。这种类型安全使得iBATIS可以知道从数据库中返回的值应该是什么类型从而将值与类型紧紧绑定。你使用map或者XML时这种工作是猜测性的，但使用JavaBean时绝不是。使用JavaBean时还可以有更大的灵活性，因为你可以定制获取/设置方法从而对数据进行调优。

13.4.2 Map

iBATIS所以支持map有两个原因。其一，iBATIS使用map作为传递多个复杂参数给已映射语句的一种机制。其二，有时数据库中的表确实什么都不是，它根本没有什么对应的领域类，而仅仅是一堆名值对而已。

尽管如此，在领域模型中使用map仍然是一件非常可怕的事情，因此决不能用map来代表业务对象。这个建议绝不特定于iBATIS；不论你的持久层是什么，都不应该在领域模型中使用map。map非常的低效，它们不具类型安全性，且比JavaBean消耗更多的内存，它们可能会变得不可预测、难以维护。使用map要谨慎。

13.4.3 XML

iBATIS支持直接从数据库中存取XML，不论是以DOM的形式或者简单地以字符串的形式。使用XML的确有其价值，尽管该价值十分有限；但是，在一个只需要以一种可移植、可解析的格式快速转换数据的简单应用程序中，XML可能就是有用的。

但是，与map一样，XML决不应该是领域模型的首选。XML是所有类型中最慢的、最不安全的，并且最消耗内存的。它与数据的最终状态（例如，通常是HTML）是最接近的，但这种优势是因其难以操纵和难以维护作为代价的。与map一样，使用XML要谨慎。

13.4.4 基本类型

基本类型可直接作为参数类型和结果类型，这是为iBATIS直接支持的。以两种方式使用基本类型都没有任何问题，基本类型高效而（类型）安全。很显然，使用基本类型时数据不可能过于复杂；但是，对于像计算一个给定查询返回的记录数这样简单的需求来说，一个基本类型的integer就足够了。只要基本类型能够满足你的需求，就放心地使用它吧。

13.5 小结

使用iBATIS并不难，但是和其他任何框架一样，如果能遵循我们推荐的最佳实践，那么你总

能改进所得的结果。

本章讨论了如何正确地测试应用程序中的持久层。通过使用两个流行的单元测试框架JUnit和JMock，我们总是能够以一种简单并且一致的方式来测试应用程序的三个不同的层。我们还讨论了如何正确地设立测试数据库，以确保测试不需要网络连接，也不需要如关系数据库管理系统这样复杂的基础设施。

此外，我们还讨论了管理XML文件的最佳方式。位置透明是简化配置、确保测试的简单性和未来的可维护性的关键。在Java应用程序中，位置透明可以通过将所有的SQL映射文件都放置在类路径上来实现。然而某些情况可能不允许你把所有的SQL映射文件都放置在类路径上，因此我们还讨论了一种分离配置的方式，以便SQL映射配置文件保存在一个集中位置上，同时将所有其他的SQL映射文件保存在你所期望的类路径上。

接下来，我们讨论了如何为iBATIS涉及的所有组件命名。对组件的命名与对组件的组织同样重要，因为这样才能保证你的映射文件易读、易理解。使SQL语句的命名规范与方法的命名规范保持一致可使得代码看上去遵循一种一致的范型（paradigm），因为已映射语句和方法其实的确没什么不同。如果使用参数映射，那么它的名字应该基于使用它的已映射语句的名字，因为参数映射复用的可能性很小。另一方面，结果映射具有很好的可复用性，因此其命名应该基于其绑定的类的名字。

选择映射为何种类型是本章关注的最后一个问题。应该在领域模型中使用JavaBean、map，还是XML呢？答案是明确的。应该使用JavaBean作为领域模型。Map和XML存在相同的问题，除了糟糕的性能以及不具备类型安全性外，两者对于将来的维护工作来说可能都是一场噩梦。

第 14 章

综合案例研究

本章内容

- 选择工具
- 设立工程
- 各种组件的综合应用

iBATIS不是一个孤岛，它本来就是一个供人使用从而成为某个应用程序一部分的框架。任何一个需要访问SQL数据库的应用程序，都可以使用iBATIS作为数据访问工具。虽然有许多各不相同的应用程序都可能会访问SQL数据库，但在这里我们将集中焦点，仅仅讨论最流行的Web应用程序。Web应用程序在幕后往往都需要访问一个SQL数据库。为了在一个有意义的上下文中使用iBATIS，本章将给出一个购物车应用程序的创建过程。我们试图做一点原创性的东西，创建一个游戏商店，而不再是什么宠物商店了。好，现在开始这个应用程序的创建过程吧。

14.1 设计理念

总是带着某种目的性开始一个应用程序的创建是一件好事情。对要创建的应用程序究竟要完成什么功能有一个总体了解是非常重要的。说得专业一点，就是要了解应用程序的需求。要描述一个购物车应用程序的需求是比较简单的，因为这个需求已经不知被描述过多少次了。虽然你可能觉得厌烦了，但这次我们还是再重复一遍。（嗨，老兄，打起精神来，这次好歹不是宠物商店了！）

我们将尽量保持购物车应用程序设计的简单性，并且将工作集中在四个主要组件的设计上：account（账户），catalog（目录），cart（购物车），还有order（订单）。我们将忽略应用程序的所有管理部分，因为这会使得需求过于复杂，并且这么做也并不会为本书带来什么新的内容。定义了应用程序的四个重要组件之后，下面深入讨论它们各自的需求。

14.1.1 账户

账户用于保存用户的相关信息。一个账户会包括某个用户的地址（address）、偏好（preference）等私人信息。用户应该能够创建和编辑自己的账户。该账户也用于处理用户登录时的安全信息。

14.1.2 目录

目录中将包含大量我们需要为之编码的东西。类别(category)、产品(product)和产品项(item)三个领域对象都将在此使用。类别、产品和产品项的深度都只有一级，一个类别将包含一些产品，一个产品又将包含一些产品项。产品项代表同一个产品的多个变种。例如，假设有一个名为Action的类别，它就可能包含一个像Doom这样的游戏。而该游戏产品又可能包含像PC、PlayStation、XBox以及其他类似的变种等产品项。

14.1.3 购物车

购物车用于维护用户选择的所有产品。它应该能够计算出该购物车中当前所有产品项的总价，以便为最终形成订单做准备。

14.1.4 订单

应用程序的订单部分将用于结账。一旦消费者选择完他们需要的产品项并且希望购买它们，他们就会选择对当前购物车结账。购物车将带着他们完成一系列的过程，包括购物信息确认、付款、形成账单、账单发送，以及最后的确认。一旦订单完成，用户就可以在他们的订单历史中浏览到它。

14.2 选择具体的实现技术

现在我们已经对需求的某些方面有所理解了，下面需要做的就是决定我们应该使用什么技术以满足需求中的这些功能。考虑到这是一个Web应用程序，我们需要看看应用程序的各个层都有哪些技术可供选择。一个标准的Web应用程序可分成如下部分。

- 表现层：应用程序中特定于Web的部分。
- 服务层：大部分业务逻辑都存在于此层。
- 持久层：我们将在此层中处理特定于数据库访问的元素。

14.2.1 表现层

对表现层来说，有许多技术可供选择。顶层框架如Struts、JSF、Spring和WebWork。所有这些框架都有它们自己的积极拥护者，并且在其适用范围内的确工作得非常出色。但在所有这些框架中，我们使用Struts。原因有三条：首先，Struts框架非常稳定，具有可预测性；第二，Struts框架至今仍然保持着一种非常上进的姿态；最后，Struts框架对于开发新的应用程序和更改现存应用程序都非常合适。在本章之后的各节中我们将假设你对Struts框架有一个中等程度的了解。否则，建议你阅读Ted N. Husted、Cedric Dumoulin、George Franciscus和David Winterfeldt所著的Struts in Action一书（Manning，2002），该书是学习Struts非常好的资料。

14.2.2 服务层

服务层将非常直接。因为本书是关于iBATIS的，所以我们将服务层中使用iBATIS的DAO

框架，用它来获取DAO对象实例为服务类中的实例变量。这样我们就可以将服务类与DAO实现类隔离开。此外，iBATIS的DAO框架还将用于事务划界，这样我们就可以在服务类中将持久层的细粒度数据访问方法聚集起来，形成粒度更粗的业务方法。

14.2.3 持久层

如果我要在持久层中使用iBATIS的SQL Map，相信你一定不会吃惊吧。iBATIS的SQL Map框架将负责完成包括管理SQL，持久化高速缓存以及执行数据库调用在内的一些任务。我们将避免在本章中对这三个主题进行过于细致的讨论，因为本书原本就是一本关于这些主题的书了。

14.3 调整 Struts: 使用 BeanAction

最近，Web应用程序框架正在经历一场革命。如状态管理、基于bean的表现层类、增强的GUI组件，还有复杂事件模型（sophisticated Event model）等等这样的特征正在被越来越多地引入到Web应用程序框架中，从而使得Web开发变得更加容易。但即使像这样处于呼之欲出的新一代框架的夹缝中，Struts仍然是一个深受欢迎、被大量使用的框架。经过对各个框架认真地评估，我们还是决定为JGameStore使用Struts，但要求其能够保证我们的应用程序与新一代框架的前向适应性。考虑到这个因素，我们决定使用一项所谓的BeanAction的技术：BeanAction技术允许标准Struts应用程序的开发人员非常容易的掌握iBATIS如何适用于标准的Struts应用程序。同时，使用像JSF、Wicket、Tapestry这样的新一代框架的开发人员也能够很容易地理解BeanAction技术的语意。最后，要说明的是，我们并不打算使Struts有何不同；我们使用BeanAction只是希望应用程序能够为更多背景的读者所理解。

BeanAction技术成功地将Action和ActionForm的职责展平，融入到一个类中。它也可以使你从对session、request这样特定于Web的组件的直接访问中解脱出来。这种类型的架构不禁让我们想起了WebWork和JSF。BeanAction技术所以能够将Action和ActionForm的职责展平，是通过一些关键组件完成的：首先是BeanAction类，它扩展了Struts的Action类、BaseBean类，以及ActionContext类。理解这些组件对于理解BeanAction技术如何工作非常重要，它们的关系如图14-1。

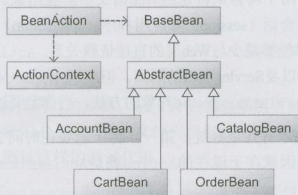


图14-1 BeanAction架构的UML图

14.3.1 BaseBean

在深入研究ActionContext类和BeanAction类之前，理解BaseBean类的功用非常重要。BaseBean扩展了ValidatorActionForm以允许进行标准的Struts验证。有了BaseBean，在自己的应用程序中就只需要扩展它，而再不需要去直接扩展ActionForm了。应该在BaseBean扩展类中包含你需要的属性，就像一个ActionForm通常所作的那样。Struts会像填充ActionForm一样地去填充BaseBean，因为BaseBean就是ActionForm。唯一的区别就在于你的BaseBean扩展类中的页面行为处理方法可以使用一种更为简单的签名：public String methodName()。

14.3.2 BeanAction

我们将要介绍的下一个类就是BeanAction类。该类有两个职责。第一，它负责填充ActionContext。其次，它负责将页面行为调用路由到你的BaseBean扩展类中对应的页面行为处理方法上，并且将该方法返回的字符串转化为一个Struts的ActionForward。这也就是为何BaseBean的页面行为处理方法的签名中可以不出现在特定于Struts组件的原因所在。当需要决定调用你的BaseBean扩展类中的哪个页面行为处理方法时，BeanAction类会在两个不同的地方查找^①。首先，它查看Struts配置文件中的动作映射（action mapping）的parameter属性值，以确定你是否显式指定了页面行为与处理方法的对应关系。如果parameter取值非空且不为*（即取值为一个有意义的字符串），则调用BaseBean扩展类中方法名为parameter取值的方法；如果parameter取值为*，则不调用BaseBean扩展类中的任何处理方法，而是直接使用name取值为success的forward子元素；如果parameter取值为空或不存在，那么BeanAction会查看path，然后使用其最后一个“/”后的字符串作为BaseBean扩展类中的页面行为处理方法名并调用该方法。例如，对如下代码，就将调用orderBean中的checkout方法：

```
<action path="/cart/checkout" type="com.ibatis.struts.BeanAction"
name="orderBean" scope="session" validate="false">
  <forward name="success" path="/order/ViewOrder.jsp"/>
</action>
```

14.3.3 ActionContext

最后，ActionContext用于将你从特定的Web语义中解脱出来。它使你请求（request）、参数（parameter）、cookies、会话（session）和应用程序（application）这些对象的访问都能够通过一个Map接口完成。这就使你能够减少与Web层的直接依赖关系。ActionContext中的大多数方法可以使你成功地实现与Struts以及Servlet API的隔离。但ActionContext也提供了直接访问HttpServletRequest和HttpServletResponse对象的方法，当你需要访问它们的时候可以使用。

使用ActionContext技术有几个好处。第一，再不需要花时间去将一个ActionForm对象转换为其某个扩展类型了。原因就在于现在的Action就是ActionForm。你只需要直接访问Base-

① 以下文字及示例都是译者给出的，原书中语焉不详且存在小小错误，故译者分析源代码后采用了自己的表述，没有遵循原文。——译者注

Bean扩展类的实例中的各个特性。其次，那些页面行为处理方法的复杂性大大降低了。通常，Struts的页面行为处理方法（即Action类的execute方法）需要接收4个参数：HttpServletRequest、HttpServletResponse、ActionForm和ActionMapping，并且要求返回一个ActionForward。而BeanAction的页面行为处理方法则远远不需要那么复杂，它不需要任何参数，只需要返回一个字符串。第三，单元测试一个简单的bean毕竟要比单元测试一个ActionForm和Action简单得多。当然可以使用MockObject和StrutsTestCase来对Action类进行彻底的测试。但是，仅仅测试一个bean肯定还是要简单得多。最后，因为BeanAction架构与现存的Struts应用程序协同工作得很好，而BeanAction技术的语意与新一代框架是一致的，这就使你能够轻松地将被应用程序的架构从现在的Struts转换为某个新一代框架而不会破坏你之前所有的辛苦劳动。

14.4 JGameStore 工程结构

好，现在开始构建开发环境。我们不会将注意力集中在那些仅用于开发基于iBATIS的应用程序的特定工具上，但却想花一些时间来提供一个对大家来说都非常有用的应用程序的基本结构。合理地组织源代码树对于编写出优秀的、清晰的、简单的代码来说是非常重要的。如果你喜欢，可以一边阅读本章一边对照查看iBATIS JGameStore应用程序中的源代码。篇幅所限，我们不可能在书中覆盖之前提到的所有需求。但是，如果愿意的话，你可以通过查看JGameStore的源代码来了解那些我们没能在本章中覆盖的内容。

从创建一个名为jgamestore的基(base)文件夹开始吧。它将成为工程的文件夹并包含源代码树。可以在喜欢的任意IDE中创建这个文件夹并且将相应工程命名为jgamestore，也可以简单地在操作系统中手动创建这个文件夹。在工程文件夹下再创建如下文件夹：src、test、web、build、devlib和lib：

```
/jgamestore
 /src
 /test
 /web
 /build
 /devlib
 /lib
```

下面仔细研究一下每个文件夹。

14.4.1 src 文件夹

文件夹名src是source（源代码）的简称。src文件夹将包括所有的Java源代码文件，以及需要存在于类路径上的特性文件和XML文件。这些文件对我们的分布式应用程序来说都是有用的。src目录不应该包含任何像单元测试这样的测试代码。

所有的源代码都包含在org.apache.ibatis.jgamestore基包中。基包下面的每一个包可以认为是JGameStore的所有组件的一个分类。

这些子包如下：

- domain包——该包中包含所有的DTO/POJO类，这些类的对象都是应用程序中的瞬时对象（transient object）。这些对象将在应用程序的其他各层之间传递和使用。
- persistence包——该包用于存放数据访问接口和实现，以及SQL Map XML文件。这些数据访问实现将使用iBATIS SQLMap API。
- presentation包——该包中包含所有的表现层bean。这些类将包含与Web应用程序中各个不同场景相关的所有特性和行为。
- service包——该包中包含业务逻辑。这些粗粒度的服务类用于将对持久层中的细粒度的数据访问方法的调用组织起来。

14.4.2 test 文件夹

test目录包含所有的单元测试代码。该包的结构将与src目录完全一致。包中的单元测试用于测试位于src目录的姊妹包中的类。所以要将测试目录与源代码目录分开，原因有很多，其中大部分与代码的安全性和可测试性有关。

14.4.3 web 文件夹

web文件夹包含所有与Web相关的制品，如JSP、图标、Struts配置文件以及类似的文件。

web文件夹的结构如下：

- pages目录——其下的account、cart、catalog、common和order目录全都包含应用程序相关部分的JSP。这些目录的名字就让我们一目了然地知道其中包含的JSP的用途。
- css目录——该目录包含应用程序中使用的所有CSS。如果你对CSS不熟悉，只要在网上搜索一下，就可以发现很多的资源。
- images目录——该目录包含应用程序中用到的所有图标或图像。
- WEB-INF目录——该目录下包含与servlet规范和Struts相关的所有配置文件。

14.4.4 build 文件夹

build文件夹包含Ant脚本以及相关的起帮助作用的Linux shell脚本和Windows BAT脚本，这些脚本可以使得构建过程更加轻松。

14.4.5 devlib 文件夹

该文件夹包含那些编译时需要但在发布时的WAR包中不需要的JAR文件。

开发时需要的库（library）包括：

- ant.jar
- ant-junit.jar
- ant-launcher.jar

- cgilib-nodep-2.1.3.jar
- emma_ant.jar
- emma.jar
- jmock-1.0.1.jar
- jmock-cglib-1.0.1.jar
- junit.jar
- servlet.jar

14.4.6 lib 文件夹

lib 文件夹包含编译和作为 WAR 包发布时都需要的所有 JAR 文件。

运行时和发布时都需要的库 (library) 包括：

- antlr.jar
- beanaction.jar
- commons-beanutils.jar
- commons-digester.jar
- commons-fileupload.jar
- commons-logging.jar
- commons-validator.jar
- hsqldb.jar
- ibatis-common-2.jar
- ibatis-dao-2.jar
- ibatis-sqlmap-2.jar
- jakarta-oro.jar
- struts.jar

基本的源代码树结构建立完毕后，就可以开始创建应用程序了。考虑到应用程序的 catalog 部分是用户最先查看的部分，我们就从这个部分开始吧。

14.5 配置 web.xml 文件

web.xml 的设置是非常直接的。我们将在其中设置 Struts ActionServlet，然后再简单地设置一些安全选项以避免用户可以直接访问 JSP 页面。

代码清单 14-1 web.xml 中的 ActionServlet 配置

```
<servlet>
  <servlet-name>action</servlet-name>
  <servlet-class>
    org.apache.struts.action.ActionServlet</servlet-class>
  <init-param>
    <param-name>config</param-name>
```

```
<param-value>/WEB-INF/struts-config.xml</param-value>
</init-param>
<init-param>
  <param-name>debug</param-name>
  <param-value>2</param-value>
</init-param>
<init-param>
  <param-name>detail</param-name>
  <param-value>2</param-value>
</init-param>
<load-on-startup>2</load-on-startup>
</servlet>

<servlet-mapping>
  <servlet-name>action</servlet-name>
  <url-pattern>*.shtml</url-pattern>
</servlet-mapping>
```

使用<servlet>标签将ActionServlet设置为请求的处理器。对ActionServlet的设置就是典型的Struts设置，没有任何特别的地方。<servlet-class>标签中指定的ActionServlet就是标准的ActionServlet。其他一切都取标准情况设置：struts-config.xml在标准路径上，调试级别取为标准值2，细节级别取为标准值2，load-on-startup也取为标准值2。

注意<servlet-mapping>标签。为了让我们觉得自己似乎很聪明，我们决定不使用标准的.do扩展来把请求映射到ActionServlet，而使用.shtml。这样做的原因纯粹是为了好玩，让人觉得我们似乎在使用古老的SHTML技术。谁知道呢——说不定这会使得那些黑客在试图攻击我们的网站时迟疑一下。

当使用Struts时，非常重要的一点就是避免用户对JSP页面的直接访问。JGameStore使用的所有JSP页面都被放在pages目录中。既然所有的JSP页面都存放在了一个目录中，那么只要简单地避免对该目录的直接访问就可以了，如代码清单14-2所示。这样就可以保证对JSP页面的所有访问都是通过Struts ActionServlet进行的。

代码清单14-2 web.xml中的安全设置

```
<security-constraint>
  <web-resource-collection>
    <web-resource-name>
      Restrict access to JSP pages
    </web-resource-name>
    <url-pattern>/pages/*</url-pattern>
  </web-resource-collection>
  <auth-constraint>
    <description>
      With no roles defined, no access granted
    </description>
  </auth-constraint>
</security-constraint>
```

设置完web.xml文件，我们就可以将注意力集中到Struts表现层中的那些特定类和配置上了。在此我们也将更加详细地描述Struts BeanAction技术。

14.6 设置表现层

因为应用程序的catalog部分是在用户在购物时最先查看的部分，所以我们接下来就将精力集中到表现层中catalog对应部分的设置上。

14.6.1 第一步

当访客来到JGameStore时，我们的初始页面将对他致以问候。在Web应用程序中使用Struts框架时有一条重要规则就是，总是通过Struts控制器（ActionServlet）转发请求。为了让用户能通过Struts框架进入初始页面，必须创建一个简单的index.jsp，该jsp中只包含一条简单的转发语句，指定了我们希望转发到的URL，该URL定义在struts-config.xml文件中，查看该文件我们就可以看到该转发语句执行成功后请求将继续转发到index.tiles，该tiles定义在tiles-defs.xml文件中，其中调用了我们的访客最终将到达的初始JSP页面——catalog目录下的Main.jsp。

为使index.jsp转发成功，首先需要为访客设置初始页面（见图14-2）。

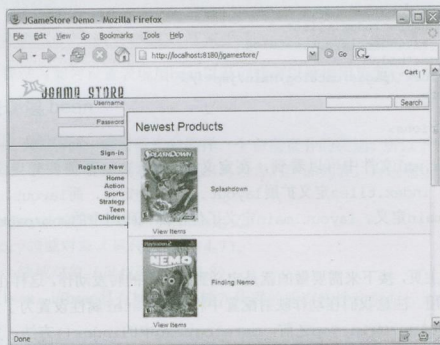


图14-2 Main.jsp页面是访客将看到的第一个页面

该页面位于源代码树的web/catalog/目录下，名为Main.jsp。考虑到catalog是访客首先访问的东西，我们将主页放在catalog目录下。

此外，还需要在tiles-defs.xml文件中删除一条对该页面的定义语句（见代码清单14-3）。Struts的tiles使得创建公共模版页面和复用页面变得非常容易，这就避免了重复的JSP包含语句。

要了解更多关于tiles的知识，请参考*Struts in Action*一书。

代码清单14-3 tiles定义配置

```
<tiles-definitions>
...
<definition name="layout.main" path="/pages/main.jsp" >
  <put name="header"
    value="/pages/common/header.jsp" />
  <put name="footer"
    value="/pages/common/footer.jsp" />
  <put name="left"
    value="/pages/common/left-blank.jsp" />
</definition>

<definition name="layout.catalog"
  extends="layout.main" >
  <put name="header"
    value="/pages/common/header.jsp" />
  <put name="footer"
    value="/pages/common/footer.jsp" />
  <put name="left"
    value="/pages/common/left.jsp" />
</definition>

<definition name="index.tiles"
  extends="layout.catalog" >
  <put name="body"
    value="/pages/catalog/Main.jsp" />
</definition>
...
</tiles-definitions>
```

从tiles-defs.xml文件中可以看到，在定义index.tiles前需要先定义layout.main和layout.catalog。index.tiles定义扩展layout.catalog定义，而layout.catalog定义又进一步扩展layout.main定义。layout.main定义了位于源代码树中的web/pages/main.jsp上的基本模板JSP。

一旦设置好了主页，接下来需要做的就是定义到该主页的转发动作，这样在index.jsp中设置的转发才能起作用。注意我们在动作映射配置中将parameter属性设置为了index。因此，在请求被转发到/catalog/Main.jsp之前，catalogBean中的index()方法（见代码清单14-4）将先被调用。

代码清单14-4 Struts配置文件中的动作映射

```
<action-mappings>
  <action path="/index"
    type="org.apache.struts.beanaction.BeanAction"
    name="catalogBean" parameter="index"
    validate="false">
```

```
<forward name="success" path="index.tiles"/>
</action>
</action-mappings>
```

在JGameStore应用程序中，catalogBean的index()方法调用了CatalogService，来填充每个类别中的最新产品列表。考虑到稍后我们会提供一个更好的例子，所以现在就不做过多的解释了。目前，我们只要知道index()方法将返回SUCCESS字符串，从而请求将转发到catalog目录中的Main.jsp（见代码清单14-5）。

代码清单14-5 catalogBean中的index()方法

```
public String index() {
    ...
    return SUCCESS;
}
```

接下来，可以使用index.jsp来通过Struts控制器从而成功地将请求转发到主页。这对于index.jsp来说非常简单，只要添加一条forward语句就可以了。一定要保证你转发的URL以.shtml作为扩展名，这样你的servlet容器才能将请求路由过Struts控制器。

代码清单14-6 JSP转发

```
<jsp:forward page="/index.shtml"/>
```

接下来，你需要学习如何设置表现层bean以及创建可以被调用页面行为处理方法。

14.6.2 使用表现层 bean

当访客开始浏览或购物时，他们首先会选择一个希望查看的类别。所以下一步工作就是要获取访问者选择的类别，并在随后的页面中展现出其包含的产品列表。为达到以上目的，需要完成如下步骤：

- 在tiles-defs.xml文件中添加一条tiles定义。
- 创建Category领域对象（见代码清单14-7）。
- 创建Product领域对象（见代码清单14-8）。
- 创建持久层bean，该bean具有捕捉用户输入所需的所有特性和页面行为处理方法（见代码清单14-9）。
- 一个列出指定类别下所有产品的JSP页面（见代码清单14-10）。

然后需要在Struts配置文件中添加一条动作映射。

考虑到随后的代码中会使用类别，首先在org.apache.ibatis.jgamestore.domain包中创建一个Category.java文件。Category领域对象（见代码清单14-7）是一个非常简单的对象，仅仅由categoryId、名称、描述、图像这4项组成。考虑到类别的深度绝不会出现超过一层，故我们不需要为类别添加什么文特性。

代码清单14-7 Category领域对象

```
...
public class Category implements Serializable {

    private String categoryId;
    private String name;
    private String description;
    private String image;

    // simple setters and getters
...

```

我们现在编写的这些代码中也会使用Product领域对象（见代码清单14-8）：JSP页面将加载一系列产品对象并显示它们。这也意味着目前产品还不会被任何Java代码直接访问。虽然如此，为周全起见，在此完成该领域对象的定义还是明智的。在org.apache.ibatis.jgamestore.domain包中添加一个Product.java文件。Product对象由productId、categoryId、名称、描述和图像组成。除Product领域对象中包含一个关联的categoryId外，Product与Category领域对象应该说没有什么差别。

代码清单14-8 Product领域对象

```
...
public class Product implements Serializable {

    private String productId;
    private String categoryId;
    private String name;
    private String description;
    private String image;

    // simple setters and getters
...

```

好，现在已经将领域对象建立完毕了，开始组装表现层bean吧。查看CatalogBean的源代码（见代码清单14-9）可以让我们对BeanAction如何应用于真实世界有一个初步的了解。我们需要使用创建一个viewCategory页面行为处理方法CatalogBean。该方法将使用BeanAction风格的行为签名：public String <behaviorName> ()。viewCategory行为非常简单。它的工作就是获取一个与当前选中的类别相关的产品列表，并将当前Category对象填充完整，然后将请求转发到某个视图。在viewCategory的方法体中，通过调用CatalogService类来填充productList和category。我们稍后会仔细研究服务类。但目前来说，只要这些服务类能够正确地返回所需要的对象就足够了。

代码清单14-9 CatalogBean表现层类（与我们的关注点相关的部分）

```
...
private String categoryId;
private Category category;

```



```

...
private PaginatedList productList;
...
public String viewCategory() {
    if (categoryId != null) {
        productList =
            catalogService.getProductListByCategory(categoryId);
        category = catalogService.getCategory(categoryId);
    }
    return SUCCESS;
}
...
// category setter/getter
...
// productList setter/getter
...

```

要使CatalogBean类能够编译通过，需要在org.apache.ibatis.jgamestore.service包中创建一个简单的CatalogService接口，并在其中添加两个必要的方法：public List getProductListByCategory(Integer categoryId)和public Category getCategory(Integer categoryId)。然后，可以继续编写代码而不需要担心该接口应如何实现了。

viewCategory方法编写完后，它使用了一个名为SUCCESS的公共静态字符串作为其返回值。SUCCESS变量的值是success。该返回字符串用于为BeanAction提供接下来调用的动作转发的名称。这就导致Category.jsp页面被展示出来，如代码清单14-10。

代码清单14-10 Product列表——/catalog/Category.jsp^①

```

...
<:set var="category"
value="${catalogBean.category}"/> ❶ 将Category对象和productList
                                  设置在页面范围内
<:set var="productList"
value="${catalogBean.productList}"/>

<table width="100%">
<tr>
<td colspan="2" class="PageHeader" align="left">
<:out value="${category.name}"/>
</td>
</tr>
<tr>
<td colspan="2" align="left">
<html:link page="/index.shtml" styleClass="BackLink">
    Return to Main Page
</html:link>
</td>
</tr>
<:forEach var="product" items="${productList}"> ❷ 遍历productList

```

① 下载的JGameStore源代码中该页面与此有微小差别。——译者注

```
<tr>
  <td width="128" align="center"
    style="border-bottom: 1px solid #ccc">
    <html:link paramId="productId"
      paramName="product"
      paramProperty="productId"
      page="/viewProduct.shtml">
      <c:out value="{product.image}"
        escapeXml="false"/>
    </html:link>
    <html:link paramId="productId"
      paramName="product"
      paramProperty="productId"
      page="/viewProduct.shtml">
      View Items
    </html:link>
  </td>
  <td align="left"
    style="border-bottom: 1px solid #ccc">
    <html:link styleClass="ZLink"
      paramId="productId"
      paramName="product"
      paramProperty="productId"
      page="/viewProduct.shtml">
      <c:out value="{product.name}"/>
    </html:link>
  </td>
</tr>
</c:forEach>
```

③ 呈现链接和显示文本

③ 呈现链接和显示文本

代码一开始，用<c:set>标签①将Category对象和productList对象在页面范围内暴露了出来，这样随后需要它们的标签就可以使用它们了，如随后的JSTL核心标签（core tag）②和Struts的<html>标签中③就使用了Category和productList，以完成相应对象的展现。<c:out>标签首先用于展现Category对象中的name特性。<c:forEach>标签用于遍历productList以展现所有的产品信息，<html:link>嵌套在<c:forEach>标签内，一次展现一种产品并创建到展现该产品详细信息的页面的链接。在<html:link>标签的内容体中我们再次使用了<c:out>标签来呈现产品名称的值。

Struts标签使用通常的命名法来处理对象。例如，对于<html:link>标签，paramName属性就用于指向一个暴露在某个范围内的对象。例如，假设你将某个对象暴露给使用了名称category的页面，那么category就是paramName属性可以引用的值。与paramName属性相对应的一个属性是paramProperty属性，该属性使你可以访问paramName指向的对象的某特定特性。

与之相对应的，是JSTL中的EL（Expression Language，表达式语言）。EL比Struts的相应标签功能更加强大。EL用于为某个特定的JSTL属性暴露某个范围内的对象。我们在此不详细讨论EL，如果你对它有兴趣，推荐你阅读Shawn Bayern的*JSTL in Action*一书（Manning，2002）。

编写完所有这些必要组件的代码之后，就可以在struts-config.xml文件（见代码清单14-11）中添加一条动作映射了，这样应用程序就可以使用这些组件了。首先需要指定CatalogBean为一个形式bean，通过将它命名为catalogBean。并为catalogBean提供全限定类名的一种类型来做到这一点。然后我们就可以在动作映射中使用这个形式bean了。

代码清单14-11 针对/viewCategory的动作映射

```
<form-bean
  name="catalogBean"
  type=
    "org.apache.ibatis.jgamestore.presentation.CatalogBean"/>
...
<action
  path="/viewCategory"
  type="org.apache.struts.beanaction.BeanAction"
  name="catalogBean" scope="session" validate="false">
  <forward name="success" path="/catalog/Category.jsp"/>
</action>
```

最后，配置动作映射，如代码清单14-11所示。该映射需要我们指定一个路径，在这里，路径为/viewCategory。然后需要使用type属性来标识一个用来处理请求的Action类的全限定类名。在本例中，我们的类型为BeanAction。BeanAction会将请求交给位于动作映射所使用的形式bean中的页面行为方法去处理。该形式bean是由动作映射的name属性值指定的。此处即为我们使用稍早配置的catalogBean。然后用scope属性来指定将该形式bean驻留在会话范围内。将validate属性值设为false，因为不存在对输入值进行的任何验证。

最后，是<action>标签的子标签<forward>，它用于指定请求将转发到哪个页面。其name属性与表现层bean中的页面行为处理方法返回的字符串值相对应。在这个例子中，因为页面行为处理方法总是返回success字符串，因此请求就被转发到了/catalog/Category.jsp页面。

下面，开始着手编写服务层的代码。

14.7 编写服务层代码

服务层由两部分组成：服务接口和实现。服务类原本就是一些粗粒度的类，由它们来将持久层中更细粒度的数据访问方法组装起来。这听起来似乎非常简单，但服务类的内容却不一定那么简单。考虑到要将我们的服务类与任何特定于数据库的信息都隔离开，我们需要采取一些额外的措施以保证合适的抽象。

因为服务层需要调用数据访问层并处理事务划分，所以简单地在服务层获取一个数据库连接并使用这个数据库连接来管理事务划分似乎将更加容易。但如果这样做的话，我们就会将特定于JDBC的语义引入到服务层中。这就是说服务层现在就会意识到我们使用的数据存储实现。我们不允许出现这种情况，因为它违背了我们的目标。

正如你在第10章中学到的，iBatis提供了一个很小的称为iBatis DAO的框架，我们在此可以使用这个框架。iBatis DAO在此将承担两个重要责任。首先，它是我们的DAO工厂。其次，我们会使用iBatis DAO进行事务划分，从而减少我们的服务层对底层数据访问技术的依赖关系。在本节中，我们将继续之前的例子，并且你还将学习到如何使用iBatis DAO来设置服务层。

14.7.1 配置 dao.xml 文件

不像之前构建表现层时的流程，我们对服务层的构建过程将首先从iBatis DAO框架的配置开始。这样的过程能够让我们更容易理解服务层中涉及到的各个组件。iBatis DAO框架允许我们通过配置文件管理必要的抽象，所以从配置文件开始最为合适。

iBatis DAO中需要配置的第一个组件就是事务管理器（见代码清单14-12）。该事务管理器用于在对数据访问层的调用之上处理事务划分。我们的例子将使用一个SQLMAP类型的事务管理器，这个类型与iBatis SQL Map框架进行了集成。虽然很难将iBatis DAO框架和iBatis SQL Map框架区分开来，但它们的确是两个不同的框架。SQLMAP事务管理器非常漂亮，使用起来也容易。除非你在使用iBatis DAO框架时联合使用了其他的持久层，否则SQLMAP在大多数情况下就是最好的选择了。需要为SQLMAP事务管理器指定的唯一特性就是SqlMapConfigResource，通过在<transactionManager>标签的内容体中使用一个<property>标签来指定。SqlMapConfigResource特性的值指向sql-map-config配置文件，该文件中包含所有必要的数据库连接配置信息。任何时候需要调用事务的开始（start）、提交（commit）和结束（end）方法时，iBatis DAO框架都会悄悄地调用隐藏在背后的数据库连接对象，该连接对象在SQL Map配置文件中指定。

代码清单14-12 dao.xml中的transactionManager配置

```
<transactionManager type="SQLMAP">
  <property name="SqlMapConfigResource"
    value=
"org/apache/ibatis/jgamestore/persistence/sqlmapdao/sql/sql-map-
config.xml"/>
</transactionManager>
```

下面就需要在dao.xml中配置DAO接口和实现的对应关系了（见代码清单14-13）。做这件事非常容易——你只需要<dao>标签为对应于全限定接口名的interface属性提供一个值，实现就会成为该接口的一个全限定类名实现。如果配置的实现类没有使用指定的接口，iBatis DAO框架会在运行时给出提示信息的。

代码清单14-13 dao.xml文件中的DAO配置

```
<dao
interface=
"org.apache.ibatis.jgamestore.persistence.iface.ProductDao"
implementation=
"org.apache.ibatis.jgamestore.persistence.sqlmapdao.ProductSqlMapDao"/>
```

14.7.2 事务划分

当使用iBatis DAO框架中的SQLMAP类型的事务处理器时，你可以使用隐式或显式事务管理。默认情况下，如果没有在代码中显式指定一个事务，事务将在每次方法调用时自动开始。有很多方式避免这种情况，可以在第4章和第10章中找到相关内容。

使用SQLMAP类型的隐式事务管理的确非常简单——只需要调用DAO中的方法（见代码清单14-14），事务管理将自动执行。对于查询语句来说，我们可能并不需要什么事务，但有一个事务也不会带来什么问题。

代码清单14-14 隐式事务管理示例

```
public PaginatedList getProductListByCategory(  
    String categoryId  
) {  
    return productDao.getProductListByCategory(categoryId);  
}
```

显式事务管理（见代码清单14-15）就有点复杂了。只有当我们需要不止一次地调用DAO时才需要显式事务管理。这些代码必须放在一个try块中，可以首先调用daoManager.startTransaction()，然后调用一次或多次DAO。当对DAO调用完毕，再通过调用daoManager.commitTransaction()来提交事务。如果以上调用过程有任何一次失败了，我们都需要调用finally块中的daoManager.endTransaction()。这就会将事务回滚，避免对数据库中已存储的数据造成任何破坏。对我们正在执行的这个简单的查询语句，任何级别的事务管理都不需要。当然，如果你喜欢，也可以为其加上事务管理。

代码清单14-15 显式事务管理示例

```
public PaginatedList getProductListByCategory(  
    String categoryId  
) {  
    PaginatedList retVal = null;  
    try {  
        // Get the next id within a separate transaction  
        daoManager.startTransaction();  
        retVal = productDao  
            .getProductListByCategory(categoryId);  
        daoManager.commitTransaction();  
    } finally {  
        daoManager.endTransaction();  
    }  
    return retVal;  
}
```

现在，针对简单的视图类别示例，服务层中需要完成的工作我们都已经看到了，接下来将DAO层中剩余的工作组装进来，以完成这个完整的例子。

14.8 编写 DAO

数据访问层是我们的Java代码真正与数据库接触的地方。我们在此使用iBATIS SQL Map框架以使对SQL的处理更加容易。创建一个使用iBATIS SQL Map框架的数据访问层需要完成三部分工作：首先需要创建一个SQLMap配置文件，然后要创建所有相关的SQLMap文件，最后还需要编写DAO。

下面来看看如何在视图类别示例中应用它们，以获取相应的产品列表。

14.8.1 SQLMap 配置

我们将使用一个名为sql-map-config.xml的配置文件来指定数据库特性，设置事务管理器，并将所有的SQLMap文件联系在一起（见代码清单14-16）。<properties>标签将指向一个database.properties文件，其中包含一系列的键值对，这些键值对用于替换sql-map-config.xml文件中形如\${...}的项。必须保证database.properties文件中包含对于所选的数据库来说是正确的数据库驱动程序（driver）、URL、用户名（username）和密码（password）。

代码清单14-16 SQLMap事务管理器文件

```
<sqlMapConfig>
<properties resource="properties/database.properties"/>
<transactionManager type="JDBC">
  <dataSource type="SIMPLE">
    <property value="${driver}" name="JDBC.Driver"/>
    <property value="${url}" name="JDBC.ConnectionURL"/>
    <property value="${username}" name="JDBC.Username"/>
    <property value="${password}" name="JDBC.Password"/>
  </dataSource>
</transactionManager>

<sqlMap resource="-CCC
"org/apache/ibatis/jgamestore/persistence/sqlmapdao/sql/Product.xml"/>
</sqlMapConfig>
```

紧接着，就需要配置事务管理器了。针对我们的目标，我们将使用最简单的JDBC类型的事务管理器。JDBC类型指定SQLMap框架将使用标准的数据库Connection对象的提交和回滚方法。考虑到我们是在服务层处理事务划界，以上配置非常重要。此外，为使在iBATIS DAO中配置的事务管理器正常工作，此处的事务管理器配置也是必需的。

<transactionManager>标签内的<dataSource>标签定义了一个JDBC数据源，事务管理器将使用该数据源来获取连接。将type指定为SIMPLE是因为我们将让iBATIS来处理数据源连接池。<property>标签在此用于指定数据库驱动程序（driver）、连接URL、用户名（username），还有密码（password）。每个<property>标签都使用了\${...}语法并从database.properties文件中获取值。

需要设置的最后一个元素就是<sqlMap>标签。该标签用于指定SQLMap文件的位置。无论何

时iBatis被第一次访问所配置的，SQLMap以及其内容都将被加载到内存中，这样其中的SQL语句就可以根据需要随时执行。

14.8.2 SQLMap 文件

需要创建一个SQLMap文件来存储类别产品列表所需的SQL调用。在该文件(命名为product.xml)中，我们会为Product对象定义一个type Alias，定义一个高速缓存模型来高速缓存查询结果，并定义一个<select>来存储我们的查询SQL(见代码清单14-17)。

代码清单14-17 SQLMap文件——Product.xml(部分)

```
<sqlMap namespace="Product">

  <typeAlias
    alias="product"
    type="org.apache.ibatis.jgamestore.domain.Product"/>

  <cacheModel id="productCache" type="LRU">
    <flushInterval hours="24"/>
    <property name="size" value="100"/>
  </cacheModel>

  ...

  <select
    id="getProductListByCategory" resultClass="product"
    parameterClass="string" cacheModel="productCache">
    SELECT
      PRODUCTID,
      NAME,
      DESCRIPTION,
      IMAGE,
      CATEGORYID
    FROM PRODUCT
    WHERE CATEGORYID = #value#
  </select>

  ...
</sqlMap>
```

<typeAlias>标签用于为Product领域对象的全限定名定义一个别名。在此，我们指定这个别名为product。每当需要引用Product领域对象时，这个别名都可以为我们节省很多敲键盘的时间。

高速缓存模型也非常简单，我们将其类型配置为LRU并命名为productCache。考虑到LRU中对象有可能会持续较长时间，我们通过指定<flushInterval>标签以保证高速缓存不会“悬挂(suspend)”超过24小时。这将使得高速缓存中的对象保持相对清新。将LRU的size属性设置为100将允许在productCache高速缓存模型中存储100条不同的结果。如果网站访问量较高，一天至少一次的强制刷新将仍可以保证一个较好的性能。

接下来就是查询语句了。我们将该查询的id指定为一个可以表现该SQL实际目的的值。不要

害怕id取值过于冗长，越冗长越能清晰地表明包含在查询语句中的SQL的确切目的。在这个例子中我们将id取值为getProductListByCategory。不用怀疑其中的SQL将基于所提供的类别返回一个列表。

利用之前定义的typeAlias，我们将查询语句的resultClass指定为product。注意，虽然这条查询语句的执行将返回一个product列表，但我们并没有将resultClass指定为一个List。原因就在于该语句也将被用于返回单个的Product对象。这可能显得有些荒谬，因为我们明明将该语句命名为getProductListByCategory，但的确有这样的情况，一条查询语句是多用途的，它既可以返回一个单一的对象，也可以返回一个对象列表。

该查询语句的parameterClass也使用了别名string（iBATIS默认定义）。如你所猜测的一样，这个别名代表String对象。当然parameterClass属性也可以使用用户自定义的别名。

查询语句的最后一个属性就是cacheModel，我们用它来引用之前定义的productCache高速缓存模型。指定cacheModel保证了所有被查询到的类别产品列表将被高速缓存。这就可以为应用程序提供更好的性能，因为不需要对数据库进行不必要的重复访问。

下一步就是在<select>标签的内容体中定义SQL语句了。查询语句将从Product表中取出一个结果集，并将它们映射为一个产品对象列表，如<select>标签配置所指定的那样。表中所有的字段都将轻松地映射为Product对象中的相应特性，因为列名与特性名是一一对应的。

一旦完成sql-map-config.xml、别名、高速缓存模型以及查询语句的配置，我们就可以在Java代码中使用iBATIS API了。我们将通过为ProductDao接口编写一个实现来使用iBATIS的SQLMap。

14.8.3 接口和实现

当一个应用程序被分为多个层时，层与层之间总是通过接口进行通信，这是一种非常优秀的设计。针对我们的示例来说，目前我们正在服务层和数据访问层之间工作。服务层应该总是与DAO接口交互而不知道任何具体的DAO实现。我们的示例也是这样，如下所示：

```
public interface ProductDao {
    PaginatedList getProductListByCategory(
        String categoryId);
    ...
}
```

我们有一个ProductDao接口，该接口将被CatalogService类使用。因为CatalogService只与ProductDao接口交互，因此它并不关心实际的实现。在ProductDao中，我们需要定义一个CatalogService能够使用的，getProductListByCategory方法。该方法的返回类型是PaginatedList，其方法签名仅包含一个String类型的categoryId。

```
public class ProductSqlMapDao
    extends BaseSqlMapDao
    implements ProductDao {
```

```
...
    public ProductSqlMapDao(DaoManager daoManager) {
        super(daoManager);
    }

    ...

    public PaginatedList getProductListByCategory(
        String categoryId
    ) {
        return queryForPaginatedList(
            "Product.getProductListByCategory",
            categoryId, PAGE_SIZE);
    }
}
...
}
```

ProductDao接口的实现是ProductSqlMapDao,它位于org.apache.ibatis.jgamestore.persistence.sqlmapdao包中。这个类扩展了BaseSqlMapDao,而BaseSqlMapDao又扩展了SqlMapDaoTemplate。SqlMapDaoTemplate是iBATIS SQLMap的一个基础类,该类中包括可用于调用定义在SQLMap XML文件中的SQL的很多方法。我们将在ProductSqlMapDao类的getProductListByCategory方法实现的体中使用queryForPaginatedList方法。当调用queryForPaginatedList方法时,我们传进了希望调用的命名空间及语句名(例如Product.getProductListByCategory),以及查询所针对的categoryId,还有我们希望一次展现的返回列表的长度。

14.9 小结

就这些了。我们已经将所有的组件在一个简单的应用程序中组合了起来。我们对应用程序的各个层都过了一遍,包括表现层、服务层和数据方法层。每一层都有一组属于该层的类和框架需要研究。我们研究了iBATIS、BeanAction、iBATIS DAO框架和iBATIS SQLMap框架,但你仍然有很多试验可做。我们并没有涉及更新、插入、删除和需要使用动态SQL的查询操作。所有这些在JGameStore示例工程中都有涉及。利用你在本章中学到的东西去好好研究一下完整的JGameStore吧,相信对你会非常有帮助的。

附录 A

iBATIS.NET快速入门

在本书较早的时候我们就说过iBATIS是一个跨平台的概念。在iBATIS 2.0发布后不久，我们就成立了一个新的开发小组，致力于将iBATIS移植到.NET平台之上。本附录将使你对.NET平台上的iBATIS有一个快速而总体性的了解。

A.1 比较 iBATIS 和 iBATIS.NET

开源项目通常都对平台移植性缺乏热情，移植性的项目往往会在开始后不久就归于失败。但iBATIS.NET却不会这样。

我们的iBATIS.NET开发小组非常地勤勉，总是时刻更新着iBATIS.NET的核心特征集，以保证其与Java版的一致性。iBATIS.NET与Java版iBATIS同在一个项目之下，其开发团队与我们属于同一个小组，这是一个非常重要的优势，因为不同平台的开发人员始终能够在一起交流，我们每天都可以从彼此那儿学到很多东西。这不论是对Java还是.NET平台上的开发都是非常好的，因为任何一个平台上的创新都可以立即为另一个平台所借鉴。

A.1.1 为何 Java 开发人员应该关心 iBATIS.NET

事实是，.NET的存在就一定有其理由。作为软件开发人员，总是要和各自异构的环境打交道。如果你够专业，并且希望能够长期从事软件开发这个行业，那么就需要开拓视野，而不能仅仅局限在Java上。这并不是说Java或者说.NET没有前途。如果你是一个独立咨询师，那么.NET可能会是你50%的市场，忽略这样的市场无疑是一个巨大的损失。如果你是某大型软件公司的全职员工，那么就会发现.NET总会以这样或那样的方式渗透到你的环境中。.NET是一个非常有活力的平台，它总会在你的企业中占有一席之地的。

作为一个学习过iBATIS的Java开发人员，你的优势就在于可以将Java版iBATIS中学到的所有概念都应用到iBATIS.NET的持久层中。这些概念在这两个平台之间没有任何区别，也不应该有任何区别。.NET拥有许多非常巧妙的特性，例如被认为是快速而肮脏（quick-and-dirty）的软件开发方法最有力论据的DataSet，但对于一个实际的企业应用程序来说，最好还是从一个领域模型开始。正如你在本书中已经学到的，始终能够将领域模型映射到企业应用程序中的数据库，这种能力正是使用iBATIS的优势所在。

A.1.2 为何.NET 开发人员应该关心 iBATIS.NET

许多.NET开发人员都对开源感到很陌生。.NET是一个商业产品，因此许多推荐的第三方解决方案也是商业的闭源产品。有人可能会说开源的免费软件与.NET社区的文化氛围不符。如果真是这样的话，那么只能说这样的文化需要改变，相信也即将改变。

免费的开源解决方案正在.NET社区中越来越流行起来。部分原因可能在于开源的构建和测试工具正在不断地增长，而它们正是商业软件长期缺乏的。像Mono、SharpDevelop、NAnt、NUnit、NHibernate以及CruiseControl.NET这样的项目正在强烈冲击着传统的.NET开发人员，这些开发人员可能一直以来就仅仅使用过微软的工具。.NET社区也越来越认识到了开源软件的价值。即使是微软，随着www.CodePlex.com的发布，也开始在开源领域进行投资了。很高兴迎来.NET开源热潮！

A.1.3 主要区别是什么

聪明的iBATIS.NET开发人员们为了保持iBATIS.NET与iBATIS的一致性进行了大量卓越的工作，因此它们之间只有一些微小的差异。这些差异主要是由于Java和.NET平台本身在设计理念上的不同所造成的。对于像类、接口以及方法的命名规范等方面的明显差异，我们尊重.NET平台自己的约定。此外，XML文件的结构也略有不同，稍后会对此做更为详细的讨论。

A.1.4 相似之处又在哪儿

iBATIS.NET保持了与Java版本相同的所有理念和价值。除了将简单性作为首要目标之外，为了保证能够应用于尽可能多的应用程序，iBATIS.NET也具有大量的灵活性。此外，iBATIS.NET具有很少的依赖性，并且对应用程序的架构也几乎没有任何前提假设。这就是说，我们早先在本书中讨论的Java版iBATIS所具有的全部优点以及设计时的考虑，对iBATIS.NET同样适用。

本附录的剩余部分将介绍iBATIS.NET的Data Mapper框架的基本用法。iBATIS.NET也有一个称为IbatisNet.DataAccess的DAO框架。但是，本附录中并不打算详细介绍iBATIS.NET的所有特征集（例如DAO框架）。

A.2 使用 iBATIS.NET

如果曾有过iBATIS的使用经验，那么你应该能很快地适应iBATIS.NET。本节讨论使用iBATIS.NET时你需要理解的关键点。首先从iBATIS.NET的依赖（DLL）和配置开始，然后利用已有的关于iBATIS的知识，示范各种SQL映射文件的使用。

A.2.1 DLL 和依赖性

值得庆幸的是，你已经阅读了本书的其他部分，因此在此你需要知道的东西已经非常少了。因为我们正在使用.NET，当然就再不需要什么JAR文件了。相反，iBATIS.NET的Data Mapper是

作为DLL（动态链接库）文件部署的，因此你需要在程序集（assembly）中添加对该DLL的引用。与它的Java兄弟一样，iBATIS.NET版本所需要添加的依赖非常少，实际上，只有3个DLL是必需的，如表A-1所示。

表A-1 使用iBATIS.NET只需依赖的3个程序集

DLL文件名	目 的
IbatisNet.Common.dll	该程序集包含iBATIS.NET框架所有的公用工具类。通常这些公共类是为Data Mapper框架和Data Access框架所共享的
IbatisNet.DataMapper.dll	该程序集包含iBATIS.NET的Data Mapper框架的所有核心类。这些类将是你与iBATIS.NET交互时最常使用的类
Castle.DynamicProxy.dll	该程序集是iBATIS.NET仅有的两个第三方依赖之一。Castle Dynamic Proxy提供了在运行时对类的动态扩展和接口实现的支持功能。iBATIS.NET在某些情况下需要使用这些代理来支持像延迟加载和自动事务管理这样的特性

A.2.2 XML 配置文件

iBATIS.NET也需要一个XML配置文件，就像其Java兄弟一样。该文件的结构与Java版稍有不同，但已非常相似了，相信你绝对可以理解它。所有的配置文件都可以通过XSD模式进行验证。如果在Visual Studio中为这些文件安装了相应的XSD，那么你还能获得IDE的智能感知（IntelliSense）支持，这将使得这些XML文件的编写工作变得容易得多。关于如何安装XSD文件，请参考Visual Studio的文档。代码清单A-1为一个简单应用程序给出了一份样例iBATIS.NET配置文件。

代码清单A-1 SqlMap.config XML配置文件

```
<?xml version="1.0" encoding="utf-8"?>
<sqlMapConfig
  xmlns="http://ibatis.apache.org/dataMapper"
  xmlns: xsi="http://www.w3.org/2001/XMLSchema-instance">

  <providers resource="providers.config"/>

  <!-- Database connection information -->
  <database>
    <provider name="sqlServer2.0"/>
    <dataSource name="Northwind"
      connectionString="server=localhost,1403;database=Northwind;
      user id=sa;password=sa;connection reset=false;connection
      lifetime=5;min pool size=1; max pool size=50"/>
  </database>

  <sqlMaps>
    <sqlMap resource="Employee.xml"/>
  </sqlMaps>

</sqlMapConfig>
```

如果你熟悉Java版iBATIS，那么代码清单A-1中的配置文件看起来应该有点熟悉吧。它从对

提供程序的声明开始，提供程序中包含对各种数据库驱动的配置。ADO.NET数据库驱动模型需要的初始化设置要比JDBC多一点。该配置包含在providers.config文件中，而该文件则包含在iBATIS.NET发布中。注意对数据库连接的配置是完全不同的。文件并没有声明iBATIS.NET将使用ADO.NET，而Java版iBATIS将使用JDBC。所有这些底层API都使用不同的驱动模型和不同的连接字符串——因此XML中的这一部分在结构上有所不同。但是，配置文件仍然保持了其简单性。通常，你仅仅需要指定一个提供程序、一个数据源名字和一个连接字符串（connection string）。

提供程序告诉iBATIS你想要连接到的数据库类型。提供程序是iBATIS.NET的可插拔组件，对其支持的每种类型的数据库都有一个相应的提供程序。尽管如此，你可能还是需要自己查看一下providers.config文件，并禁用或移除所有你没有为其提供驱动的提供程序；否则，当试图在没有这些驱动的情况下运行应用程序时，你将得到一个运行时错误。

数据库名（由name属性指定）和连接字符串（由connectionString属性指定）使用信任状（credential）和你提供的其他信息（如用户名和密码）来建立数据库连接。如果你习惯于Java，那么这种方式与典型的JDBC连接URL没有什么不同。

配置文件的最后一部分是<sqlMaps>，其中列出了所有的SQL映射文件（包含SQL语句、结果映射以及其他的iBATIS元素）。在本例中，我们只有一个名为Employee.xml的SQL映射文件。

正如你可能已经注意到的，所有这些示例使用的都是微软的SQL Server和Access数据库管理系统自带的Northwind数据库。这个数据库经常被作为样例使用，如本附录中列出的代码清单所示。

A.2.3 配置 API

代码清单A-1所示的配置文件用于配置一个SqlMapper实例，你将使用该实例来调用在SQL映射文件中定义的已映射语句（mapped statement）。SqlMapper实例是由一个名为DomSqlMapBuilder的工厂类创建得到的，该工厂类会通读你定义的XML配置文件来构建SqlMapper。该配置的外观示例如下：

```
ISqlMapper sqlMap =  
new DomSqlMapBuilder().Configure("SqlMap.config");
```

这行代码（在此由于排版问题被分割为了两截）就是你需要的全部代码了。

现在我们已经有了一个配置好的ISqlMapper实例，利用该实例就可以调用Employee.xml中的SQL语句了。下一节会看看Employee.xml文件中到底有些什么。

A.2.4 SQL 映射文件

与代码清单A-1所示的配置文件一样，iBATIS.NET的SQL映射文件也稍有不同，但对于任何一个iBATIS用户来说相信这点微小的区别都不足以影响他们对文件的理解。iBATIS.NET支持的语句类型（包括存储过程）完全相同。下面是来自Employee.xml中的一条简单查询语句：

```
<select id="SelectEmployee" parameterClass="int"
      resultClass="Employee">
  select
    EmployeeID as ID,
    FirstName,
    LastName
  from
    Employees
  where
    EmployeeID = #value#
</select>
```

该查询语句使用一个整数作为参数，并且返回一个由Northwind数据库的Employees表的数据所填充的Employee对象。你应该能够分辨得出来，这条特定的语句使用的是自动映射。即是说，我们没有指定任何结果映射。自动映射将自动根据数据库表列的名字把它们映射到实体类中的相应特性上。注意我们是如何设置EmployeeID列的别名为ID的，因为这是Employee类中相应特性的名字。我们当然也可以显式地定义一个结果映射，但这将改变下面的代码清单：

```
<resultMap id="EmployeeResult" class="Employee">
  <result property="ID" column="EmployeeId"/>
  <result property="FirstName" column="FirstName"/>
  <result property="LastName" column="LastName" />
</resultMap>
<select id="SelectEmployee" parameterClass="int"
      resultMap="EmployeeResult">
  select *
  from
    Employees
  where
    EmployeeID = #value#
</select>
```

注意我们的SQL语句是如何被简化的，但这是以额外的称为<resultMap>的XML元素作为代价的。这是你经常需要做出的权衡，但对于Java版本来说，我们通常都会使用<resultMap>，这是因为它能够为你的结果处理提供额外的功能。

现在已经完成了对SQL语句的映射，接下来就可以用C#代码来调用该语句了，如下所示：

```
Employee emp =
    sqlMap.QueryForObject<Employee>("SelectEmployee", 1);

// You can get a sense of the result by writing out to the
// console... which should return "1: Nancy Davolio"
Console.WriteLine(emp.ID + ": " + emp.FirstName + " " +
    emp.LastName);
```

非查询语句在iBATIS.NET中没有什么不同。以下示例展示了我们应该如何定义一条插入语句。

首先，看看这段代码：

```
<insert id="InsertEmployee" parameterClass="Employee">
  insert into Employees
```

```
( FirstName, LastName )
values
( #FirstName#, #LastName# )
<selectKey resultClass="int" property="ID" >
    select @@IDENTITY
</selectKey>
</insert>
```

这段代码中的大部分只是一条简单的插入语句。但是，注意该语句中的**粗体**字部分。内嵌的<selectKey>元素对于任何一个iBATIS用户来说都应该是非常熟悉的：这是一种用于获取生成的主键列值的机制。

使用C#来调用这个语句时，Insert()方法将返回该生成值，同时它也将被用来设置传递给Insert()方法的Employee实例的ID特性的值。为清晰起见，再看看C#示例代码：

```
Employee employee = new Employee();
employee.FirstName = "Clinton";
employee.LastName = "Begin";
Object id = sqlMap.Insert("InsertEmployee", employee);
```

Northwind数据库为Employees表的EmployeeId主键使用了一个IDENTITY列。因此我们不需要为插入语句传递一个主键值^①。我们传入的Employee实例的ID特性将被更新为生成的主键值。该生成值将同时作为Insert()方法的返回值。

就像Java版iBATIS一样，一旦你了解了以上两种语句，剩下的事情就变得非常明显了。更新语句和删除语句的实现与插入语句非常相似，但分别有它们自己的<update>和<delete>元素。

A.3 到哪里去查找更多的信息

如你所见，Java版iBATIS与iBATIS.NET并没有太多的区别。它们的区别主要在于那些与它们各自的语言相关的特征上，但它们仍然维持了一种一致的用户体验。iBATIS.NET还有许多需要学习的知识点，我们在网上为你准备了许多可用的资源。访问<http://ibatis.apache.org>，你可以找到iBATIS.NET用户手册以及NPetShop样例应用程序。

^① 即不需要设定传入的Employee实例的ID特性。——译者注