

Broadview®
www.broadview.com.cn

iBatis——目前主流的 ORM框架
Java 软件设计师、架构师案头必备参考用书

iBatis

框架源码剖析

任钢 著



电子工业出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY
<http://www.phei.com.cn>

欲穷iBATIS框架源码的全貌,
探索iBATIS框架的主要思路 and 核心实现方式,
请走进

iBATIS

框架源码剖析

本书特色:

- ◎ 全面介绍iBATIS框架的体系结构;
- ◎ 深入阐述iBATIS的持久层框架:SQL Maps和DAO;
- ◎ 采用代码注释、UML分析、GoF设计模式抽象和归类、代码跟踪和案例的讲解等说明方式;
- ◎ 配套光盘提供了iBATIS框架源码解读的主要思路及实现案例。



责任编辑:高洪霞
责任美编:李玲



本书贴有激光防伪标志,凡没有防伪标志者,属盗版图书。

上架建议:软件开发/Java

ISBN 978-7-121-10872-3



9 787121 108723 >

定价:79.00元(含光盘1张)

IBATIS

iBATIS

框架源码剖析

任钢 著

电子工业出版社

Publishing House of Electronics Industry

北京·BEIJING

PDF

内 容 简 介

iBatis 是一种比较流行的 ORM 框架, 本书全面介绍其结构体系并分析其源程序代码, 该框架的核心包括两个组件, 一个是 iBatis DAO, 另一个是 iBatis SQL Map。

本书分为三个部分, 第一部分介绍 iBatis 的一些基础知识; 第二部分介绍 iBatis DAO 的框架结构及其实现; 第三部分针对 iBatis 的底层平台 iBatis SQL Map 进行分析。其中第三部分是主要内容: 首先剖析了 SQL Map 是如何读取配置信息的; 其次说明了 SQL Map 引擎的实现, 勾画出 iBatis SQL Map 的框架结构, 描述其核心实现机制和主要实现步骤; 再次说明 SQL Map 如何用来实现数据库处理, 包括事务管理、数据库连接池, 以及 SQL Map 中 Mapping 的实现, 这也是 iBatis 不同于其他 ORM 框架的独创性实现; 最后是一些常用的实现, 如 TypeHandler 类型转化和 iBatis 常用工具的实现。

在源码剖析过程中, 本书采用了代码注释、UML 分析和设计、GoF 设计模式抽象和归类、代码跟踪和案例的讲解和说明。目的是让读者全方位地了解 iBatis 的实现框架和实现手段。一方面让读者理解开发者的思路, 另一方面也是帮助读者在实际工作中能应用这些策略、方法和编程技巧。

本书适用于软件设计师、架构师和一些有较好 Java 基础的开发人员, 既可以作为 iBatis 的学习指南, 也可以给软件架构师在设计方面进行参考。

未经许可, 不得以任何方式复制或抄袭本书之部分或全部内容。
版权所有, 侵权必究。

图书在版编目 (CIP) 数据

iBatis 框架源码剖析 / 任钢著. —北京: 电子工业出版社, 2010.6
ISBN 978-7-121-10872-3

I. ①i… II. ①任… III. ①关系数据库—数据库管理系统—软件工具 IV. ①TP311.138

中国版本图书馆 CIP 数据核字 (2010) 第 087570 号

责任编辑: 高洪霞

特约编辑: 顾慧芳

印 刷: 北京东光印刷厂

装 订: 三河市皇庄路通装订厂

出版发行: 电子工业出版社

北京市海淀区万寿路 173 信箱 邮编 100036

开 本: 787×1092 1/16 印张: 32.75 字数: 786 千字

印 次: 2010 年 6 月第 1 次印刷

印 数: 3500 册 定价: 79.00 元 (含光盘 1 张)

凡所购买电子工业出版社图书有缺损问题, 请向购买书店调换。若书店售缺, 请与本社发行部联系, 联系及邮购电话: (010) 88254888。

质量投诉请发邮件至 zltts@phei.com.cn, 盗版侵权举报请发邮件至 dbqq@phei.com.cn。

服务热线: (010) 88258888。

前言

搞 IT 技术已经有十多年的历史了，接触 Java 语言也有一定的时间了。为什么到现在才要写这本书呢？几年前我开发过一个 ORM 模型框架，当时的思路居然与 iBATIS 框架有一些类似（可见英雄所见略同）。于是，为了更好地实现这个 ORM 框架，我仔细阅读了 iBATIS 框架的源码。在阅读过程中，由于 iBATIS 框架代码层层叠叠、峰回路转、跌宕起伏，为了理清框架的主要思路 and 核心实现方式、加快理解速度和加深理解深度，我用 PowerDesigner 画了一些 UML 图，并做了一些阅读笔记和备忘录。一个月下来，基本上从总体架构上了解了 iBATIS 框架的实现。这时候阅读笔记和备忘录已大约积累了好几万字。我想，如果能把这些笔记和备忘录进行系统化、简单化、章节化的整理，就可以给更多 iBATIS 爱好者使用。同样，这些学习心得对软件架构师、软件开发工程师等都非常有价值，所谓它山之石，可以攻玉。于是，我决定写一本关于 iBATIS 框架源码剖析的书籍。而在实际操作中，我觉得在讲述 iBATIS 源码的同时，已经涉及很多关于 ORM 的内容，也有一些 Java 的基础处理和编程技巧，甚至还包括一些经典的设计模式。

在国内介绍和讲述开源软件的书可谓是琳琅满目，不胜枚举。但这些书基本上都归纳为应用型或工具型，更趋向于软件的使用说明或使用指南之类。而且，在全球这么多开源框架代码中，我国做出的贡献还是非常少的。分析原因，主要是我们热衷于拿来主义，直接就用，能解决问题就行。而对于源码，也许只有在使用过程中遇到了障碍，为了解决问题才做一些源码阅读和分析，这也是国内许多人很少去分析开源框架源码的原因，而且介绍开源软件实现的书籍也是凤毛麟角。我写这本书的目的，就是希望在这方面能与大家多分享一些学习心得和体会。

对于开源代码，能读懂并搞明白是一回事。但是理解了源代码，把这些东西用文字表述出来，让别人也能理解却是另一回事。我觉得后者的难度远远大于前者。当然，如果仅仅是简单地介绍 iBATIS 框架，我相信只要有几句话就能说清楚。但是要把一个实现框架说得条理清晰、层次鲜明，这不仅仅要求有一定的技术背景，还要有文字语言的表达和掌控能力。我用一个月时间就基本上搞明白了 iBATIS 框架的内容，但是要把它写出来，的确非常头疼。有的时候要非常细致地去推敲，因为很多琐碎的细节决定了整个框架的核心，这需要有一定的耐心和抽象能力。同时阅读和理解程序源码是一种实践性非常强的工作，所谓读书破万卷，下笔如有神，这是一个真理。但若阅读并理解了几十万行程序代码后，再来编写程序代码，那基本上就可以非常有章法并有一定的深度了。事实上，我国大学目

前计算机教育的水平是基础性质的，对于如何去阅读源码没有相关的课程来进行讲授。对于源码的分析有什么办法、手段和策略，即如何把那些复杂的程序代码用简单的语言表达出来，让别人能迅速地理解并掌握，我国的教育还是做得不够的，也没有专人或学者来搞这方面的研究。但笔者觉得这项工作是非常重要的，而且也是很有意义的。事实上，每个开发人员和设计人员在实践中都有自己的一套读取和分析源码方法。当然这也是仁者见仁、智者见智的事。我的方法和手段只是其中之一而已。所以说我在编写这本书时对源码的剖析只是做了一些非常粗浅但是有意义的尝试，以试图弥补国内软件行业在这方面的缺陷。

当然，阅读本书也要具备一定的基础知识，否则，有些术语和解释还是比较难以理解的。

我在本书的编写过程中花费了大量的时间。而且，这些工作都是在业余时间内完成的，每天都要照常上班，只有到了晚上或者节假日，才有闲暇写这些东西。一般技术人员都喜欢新鲜事物，如果说去阅读和理解开源代码，也许还有一些挑战性，大家可能都有兴趣去做这件事情，但是要把这些东西用文字表述出来并给别人讲述，则许多人就不太愿意干了，原因是这是一项非常枯燥的任务，写作的艰辛旁人是很难理解的。很高兴的是我还是坚持下来了，度过了无数个寂寞和孤单的晚上终于把这本书奉献给了读者。

在这期间，我要感谢我的家人对我的理解和支持。我要感谢我的妻子在这期间承担了全部的家务，同时也要感谢我女儿在我撰写书稿的时候没给我添太多的麻烦，她一直没有搞明白爸爸总是待在电脑前做什么。我要感谢我的父母，他们肯定不知道我写的内容，但是无论我做什么，有没有成绩，他们都是一如既往地给予鼓励。我把这里的一切都献给他们。

我还要感谢本书的编辑高洪霞和顾慧芳。没有她们耐心的指导和完善，这本书也许只是一个读书笔记，我也要感谢本书的策划编辑袁金敏，是她给我信心让我继续下去，否则这些资料只能是束之高阁。我还要感谢那些在编写本书作出贡献的所有人，他们都是默默无闻的后台工作者。

当然，由于笔者水平有限，书中的错误和缺点在所难免，希望读者能给予批评和指正。笔者的联系方式：rengang66@sina.com。

编者

2009年12月于深圳

目 录

第一部分 iBATIS 的基础知识

第 1 章 iBATIS 概述 2

1.1 iBATIS 概论 2

1.2 ORM 模型介绍 4

1.2.1 什么是 ORM 4

1.2.2 ORM 的实现方式 4

1.2.3 常用的 ORM 框架 8

1.2.4 ORM 模型和持久层框架 9

1.3 iBATIS 的组件和实现的功能 10

1.3.1 iBATIS 的 DAO 组件 10

1.3.2 iBATIS SQL Map 组件 11

第 2 章 相关的技术背景和基础知识 13

2.1 面向对象和 UML 基本知识 13

2.1.1 面向对象基础 13

2.1.2 UML 基础知识 15

2.1.3 UML 图 16

2.1.4 类和接口以及之间的关系 18

2.2 Java 基础知识 26

2.2.1 Java 的 I/O 操作 27

2.2.2 Java 解析 XML 文档 27

2.2.3 Java 的线程管理 29

2.2.4 Java 的反射机制 31

2.2.5 Java 的动态 Proxy 32

2.2.6 JDBC 和 JDBC 扩展 33

2.2.7 JavaBean 34

2.2.8 JNDI 35

2.3 数据库相关基础知识 37

2.3.1 SQL 37

2.3.2 数据库事务管理 38

2.4 Java EE 规范相关知识 39

2.5 开源 ORM 框架 40

2.5.1 Hibernate 40

2.5.2 TopLink 42

2.5.3 Apache OJB 42

2.6 其他开源框架 43

2.6.1 与 Log 相关的开源框架 43

2.6.2 OSCache 44

2.6.3 Commons-DBCP 数据库连

接池 45

2.7 GoF 的 23 种设计模式 45

第 3 章 安装和配置 iBATIS 源码 48

3.1 安装和配置 iBATIS SQL Map 源码环境 48

3.2 安装和配置 iBATIS DAO 源码环境 50

3.3 安装和配置 iBATIS JPetStore 源码环境 51

3.3.1 iBATIS JPetStore 源码环境 配置 51

3.3.2 创建 iBATIS JPetStore 的 应用 53

3.3.3 安装 iBATIS JPetStore 的 MySQL 数据库 53

3.3.4 安装 MySQL 数据库的管理工具	58
3.3.5 配置成功的标志	60

第二部分 iBATIS DAO 框架 源码剖析

第 4 章 iBATIS DAO 体系结构和实现	64
4.1 iBATIS DAO 基本结构	64
4.1.1 Java EE 核心设计模式——DAO 模式介绍	65
4.1.2 iBATIS DAO 包文件和组件结构	66
4.1.3 使用 iBATIS DAO 工作流程	67
4.2 iBATIS DAO 外部接口和实现	68
4.2.1 iBATIS DAO 框架外部接口	68
4.2.2 iBATIS DAO Template API 结构和说明	69
4.3 DAO 配置文件读取	72
4.3.1 dao.xml 的格式说明	72
4.3.2 dao.xml 文件的读取过程	73
4.3.3 如何验证 dao.xml 文件	82
4.3.4 dao.xml 配置文件实例说明	84
4.4 iBATIS DAO 引擎实现	87
4.4.1 DAO 业务实现的序列图和说明	87
4.4.2 iBATIS DAO 组件管理	90
4.4.3 iBATIS DAO 事务管理实现	94
4.5 基于 iBATIS DAO SqlMap 的实例说明	124

4.6 读取源码的收获	132
-------------------	-----

第三部分 iBATIS 的底层平台 ——iBATIS SQL Map 的分析

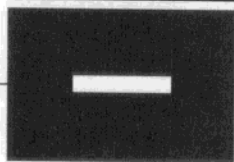
第 5 章 iBATIS SQL Map 体系结构和剖析	134
5.1 SQL Map 实现的功能和原理	134
5.2 SQL Map 组件的包结构和文件结构	136
5.3 SQL Map 的组件结构	137
第 6 章 SQL Map 配置信息的读取	139
6.1 XML 文件的验证处理	139
6.1.1 XML 验证处理的通用模式	139
6.1.2 iBATIS SQL Map 的 XML 验证	142
6.2 解析 SQL Map 配置文件	145
6.2.1 SqlMapConfig.xml 的格式说明	145
6.2.2 SqlMapConfig.xml 文件读取总体说明	147
6.2.3 基于设计模式中策略模式的数据执行	152
6.2.4 基于递归和路径来实现配置文件的全部遍历	157
6.2.5 XmlParserState 对象在解析 SQL Map XML 配置文件的协调者角色	159
6.2.6 配置的一级门面 SqlMapConfiguration 实例化对象	162
6.2.7 一级应用门面 SqlMapExecutorDelegate 实例化对象	164

6.2.8 SQL Map 配置文件中节点 解析的通用处理	165	7.3.1 业务实现类	243
6.2.9 数据库事务节点的解析和 转化	167	7.3.2 配置信息类	254
6.2.10 配置文件其他节点的 解析和转化	170	7.3.3 运行状态信息类	256
6.3 解析 SQL Map 映射文件	178	7.4 业务实现分析	258
6.3.1 SQL Map XML 映射 文件格式	178	7.4.1 业务实现两个阶段的 分析	258
6.3.2 SQL Map XML 映射文件 读取总体说明	182	7.4.2 查询类业务实现过程	259
6.3.3 XmlParserState 对象解析 SQL Map 映射文件的 协调者角色	185	7.4.3 单事务业务操作实现 过程	264
6.3.4 cacheModel 节点的解析 和转化	187	7.4.4 联合事务处理实现过程	266
6.3.5 parameterMap 节点的解析 和转化	194	7.4.5 存储过程的处理	272
6.3.6 resultMap 节点的解析 和转化	200	7.4.6 批处理及其实现	275
6.3.7 statement 类型节点的解析 和转化	212	7.4.7 全局 JTA 事务的处理	277
6.3.8 对 SQL 的处理	223	7.4.8 全局外部事务的处理	278
6.4 抽象出通用的 XML 解析 框架	229	7.4.9 用户自定义数据库 Connection 处理	279
6.5 读取源码的收获	235	7.5 读取源码的收获	280
第 7 章 SQL Map 引擎实现框架	236	第 8 章 SQL Map 数据库处理	281
7.1 SQL Map 引擎实现框架的 组成	236	8.1 SQL Map 的 transaction Manager	282
7.2 业务运行过程和介绍	239	8.1.1 Java 事务简介	282
7.2.1 总体业务运行程序 列图	239	8.1.2 SQL Map 的 transaction Manager 概述	282
7.2.2 系统总体运行简化说 明图	240	8.1.3 SQL Map 事务管理的 设计模式	283
7.3 业务实现类的分析	242	8.2 系统如何调用事务管理和 SQL Map 事务策略	285
		8.2.1 SQL Map 如何调用事务	285
		8.2.2 Java 事务类型	286
		8.2.3 SQL Map 中 JDBC 事务 实现	290
		8.2.4 SQL Map 中 JTA 事务 实现	293
		8.2.5 SQL Map 的 External 事务 实现	297

8.2.6 SQL Map 的用户事务实现	298	9.2 ResultMap 框架及其说明	338
8.3 SQL Map 的 DataSource 策略	298	9.2.1 ResultMap 框架介绍	338
8.3.1 关于 DataSource 的说明	298	9.2.2 ResultMap 框架说明	339
8.3.2 SQL Map 的 DataSource 结构和内容	300	9.2.3 ResultMap 中的类说明	340
8.3.3 SIMPLE 策略的实现	302	9.2.4 ResultMap 框架是如何工作的	341
8.3.4 DBCP 策略实现	302	9.2.5 如何实现子查询	342
8.3.5 JNDI 策略实现	304	9.2.6 延迟加载的实现	345
8.4 SQL Map 自定义 DataSource 实现	306	9.3 Statement 框架及其说明	348
8.4.1 DataSource 接口的结构	306	9.3.1 Statement 介绍	348
8.4.2 实现 DataSource 的设计思路	306	9.3.2 Statement 框架总体结构	349
8.4.3 SimpleDataSource 设计和实现	308	9.3.3 Statement 组件中的类介绍	350
8.5 SQL Map 扩展 DataSource 为 C3P0	322	9.3.4 MappedStatement 是如何工作的	354
8.6 SQL Map 如何进行批处理	324	9.3.5 Statement 缓存的实现	361
8.7 SQL Map 事务隔离的实现	327	9.3.6 自动生成的主键	363
8.7.1 JDBC 事务隔离概述	327	9.4 Sql 框架及其说明	367
8.7.2 SQL Map 的事务隔离的实现	328	9.4.1 Sql 接口框架	367
8.8 SQL Map 事务状态的实现	329	9.4.2 SqlChild 接口框架	368
8.9 读取源码的收获	330	9.4.3 Sql 接口方法	368
第 9 章 SQL Map 中 Mapping 实现	332	9.4.4 静态 SQL 的实现	369
9.1 ParameterMap 框架及其说明	333	9.4.5 简单动态 SQL 的实现	370
9.1.1 ParameterMap 总体框架说明	333	9.4.6 动态 SQL 语言的实现	372
9.1.2 ParameterMap 组件中各个类介绍	334	9.5 数据对象转换框架及其说明	379
9.1.3 ParameterMap 框架如何工作	335	9.5.1 DataExchange 组件作用、内容和设计模式	380
		9.5.2 Accessplan 组件的设计模式	393
		9.5.3 DataExchange 和 Accessplan 在系统中如何实现	399
		9.6 读取源码的收获	404
第 10 章 SQL Map 缓存管理和实现	405		
10.1 SQL Map 缓存结构和组成	406		

10.2 系统如何使用缓存	407	12.1.2 实例化类并缓存	445
10.2.1 缓存实现的序列图和说明	407	12.2 Bean 管理	447
10.2.2 CacheModel 类缓存的实现	409	12.2.1 ClassInfo 类	447
10.2.3 唯一性 CacheKey 对象的产生	411	12.2.2 Probe 接口及其实现	453
10.3 缓存策略的程序实现	412	12.3 Log 管理	468
10.3.1 FIFO 缓存实现	413	12.4 调试信息工具	472
10.3.2 LRU 缓存实现	415	12.5 ScriptRunner 的应用	472
10.3.3 MEMORY 缓存实现	417	12.6 读取源码的收获	476
10.3.4 OSCACHE 缓存实现	420		
10.4 扩展缓存策略——增加先进后出缓存策略	422	附录一 第 4 章 dao-2.dtd	478
10.5 读取源码的收获	425	附录二 第 5 章 SqlMapConfig.xml 的 DTD 结构	479
第 11 章 TypeHandler 类型转化	426	附录三 第 5 章 SqlMapConfig.xml 的 XSD 结构	484
11.1 Java 的数据类型的说明	426	附录四 第 5 章 SqlMapMapping.xml 的 DTD 结构	486
11.2 TypeHandler 组件的框架结构	427	附录五 第 5 章 SqlMapMapping.xml 的 XSD 结构	500
11.3 TypeHandlerFactory 的结构、作用和实现	428	附录六 第 11 章 JDBC Types Mapped to Java Types	503
11.3.1 TypeHandlerFactory 的别名处理	428	附录七 第 11 章 Java Types Mapped to JDBC Types	504
11.3.2 TypeHandlerFactory 容器的数据类型转化	430	附录八 第 11 章 JDBC Types Mapped to Java Object Types	505
11.4 TypeHandler 的实现	431	附录九 第 11 章 Java Object Types Mapped to JDBC Types	506
11.4.1 一般类型的处理	433	附录十 第 11 章 JDBC Types Mapped to Database-specific SQL Types	507
11.4.2 Sql 类型的处理	434	参考文献	509
11.4.3 通用类型的处理	436		
11.4.4 定制数据类型的转化	438		
11.5 读取源码的收获	440		
第 12 章 iBATIS 常用工具的实现	441		
12.1 Resources 工具	441		
12.1.1 资源加载	441		

PART



第 1 部分

基础 2ITAB

第一部分 iBATIS 的基础知识

第 1 章 iBATIS 概述

第 2 章 相关的技术背景和基础知识

第 3 章 安装和配置 iBATIS 源码



第 1 章

iBATIS 概述

本章内容：

1. 首先对 iBATIS 进行总体性的介绍。
2. 对 ORM 模式和持久层进行了分析说明。
3. 说明 iBATIS 主要的组件和实现方式。

在信息系统的开发过程中，由于绝大多数业务模型都涉及关系数据库，在采用 Java 作为系统的开发语言时，传统应用开发方法是直接用 JDBC 与数据库交互。但在这种模式下开发和维护工作量都很大并且维护调试也非常不方便，并且一旦业务逻辑稍微有一些变更，就需要大量地更改这些 JDBC 中的 SQL 语句，因此不管是开发还是维护系统都很不方便。由于 Java 的面向对象性和关系型数据库的关系型结构相差甚远，所以很有必要引入一种在对象与关系型数据库之间的直接映射机制。这种映射应该尽量地多使用配置文档，以便今后业务逻辑更改后能通过修改映射文件而不是 Java 源代码，从而出现了 O/R 映射模式。有很多开源项目都实现了 Java O/R 映射，而 iBATIS 是其中最为优秀的实现架构之一。

本书主要分析 iBATIS 的源码，一方面是为用户更好地理解和掌握 iBATIS，另一方面也是让一些高层次的开发人员从理论和实践上都有一个层次的提高。

1.1 iBATIS 概论

iBATIS Database Layer 架构自 2001 年发展以来，至今已经成为 Apache 的官方项目。在 iBATIS 创始人的《iBATIS 实战》一书中已经对 iBATIS 的定位做了一个明确的说明。iBATIS 是一种 Data Mapper，那什么又是 Data Mapper 呢？Martin Fowler 在他的《Patterns of Enterprise Application Architecture》一书中是这样描述 Data Mapper 的：一个映射层，在对象和数据库间传递数据，并保持两者与映射层本身相独立。所以说，Mapper 是在两个独立对象间建立通信关系的一种对象。

通过对程序源码的分析, iBATIS 也具备了 ORM 框架的一些基本特性, 但实际上 iBATIS 更像是一个 SQL 工具。同时, iBATIS 不是直接在类与数据表或字段与列之间进行关联, 而是把 SQL 语句的参数(parameter)和返回结果(result)映射至实体类(或 JavaBean)。iBATIS 是处于实体类和数据表之间的一个中间层, 这使得它在类和数据表之间进行映射时更加灵活, 而不需要数据库模型或对象模型(Object Model)的任何修改。我们所说的中间层实际上就是 SQL 映射层, 它使得 iBATIS 能够更好地分离数据库和对象模型的设计, 这样就相对减少了两者的耦合。

相对 Hibernate 和 Apache OJB 等“一站式”ORM 对象关系映射解决方案而言, iBATIS 是一种“半自动化”的 ORM 实现。这里要说明一下“全自动化”和“半自动化”在实现 ORM 模式上的区别。

Hibernate 和 Apache OJB 都是对数据库结构提供了较为完整的封装, 提供了从 POJO (Plain Old Java Objects 普通 Java 对象) 到数据库表的全套映射机制。软件开发人员往往只需定义好了 POJO 到数据库表的映射关系, 即可通过 Hibernate 或者 OJB 提供的方法完成持久层操作, 软件开发人员甚至不需要对 SQL 的熟练掌握。Hibernate、Apache OJB 会根据制定的存储逻辑, 自动生成对应的 SQL 并调用 JDBC 接口去执行。我们把这种模式称为“全自动化”模式。

“半自动化”ORM 框架是相对上述提到的 Hibernate 等提供了全面的数据库封装机制的“全自动化”ORM 实现而言, “半自动化”ORM 框架重点在于 POJO 与 SQL 之间的映射关系。也就是开发人员编写 SQL 语句, 通过映射配置文件, 将 SQL 所需的参数, 以及返回的结果字段映射到指定的 POJO。这些过程全是手工来进行操作。iBATIS 就属于“半自动化”ORM。

“全自动”ORM 机制在大多数情况下有很大的优势。但是, 也有一些特定的环境, 这种一站式的解决方案却未必是最佳的选择。在现实中, 这些特定的环境条件具有如下的特点:

① 数据库系统对于开发人员并不是完全控制的。处于安全考虑, 也许只有部分数据或者局部权限开放给开发人员。只对开发团队提供几条 Select SQL 或存储过程以获取所需数据, 具体的表结构不予公开。

② 为了提高系统的性能和实现分层开发, 必须由数据库层的存储过程来实现所有的业务逻辑部分。

③ 对于一些系统, 由于表与表之间主键和外键的约束关系复杂, 这造成了生成标准 POJO 对象之间的关系也比较复杂, 但是可以通过多种复合 SQL 可以编写出比较简洁高效的语句来实现多表关联查询。

④ 由于系统数据处理量巨大, 对性能要求极为苛刻, 这往往意味着我们必须通过经过高度优化的 SQL 语句或存储过程才能达到系统性能设计指标。

面对这样的需求。如果再使用 Hibernate 这种数据访问策略来实现对数据库的访问就显得很不明智了。此时“半自动化”的 iBATIS, 却刚好解决了这个问题。

当然，iBATIS 的 ORM 模式也并不是没有缺陷。首先，对于开发人员是增加工作量。iBATIS 并不会为开发人员在运行期自动生成 SQL 语句执行，具体的 SQL 语句需要开发人员编写。其次，对开发人员的要求要高一点，毕竟要求开发人员对 SQL 语句要有一定的要求。最后，iBATIS 支持的 SQL 是标准规范的 SQL，可以在所有支持标准 SQL 的数据库系统中移植和运行。但是针对一些对标准 SQL 有扩展的 SQL，如 T-SQL、PL SQL 等，则缺乏对扩展部分的支持。

在 iBATIS 创始人的《iBATIS 实战》一书中，也专门提到了 iBATIS 的不适合环境。其中 iBATIS 不适合的三种环境为：① 当对数据库永远拥有完全控制权；② 当应用程序需要完全动态的 SQL 语句；③ 当数据是非关系数据库时。

1.2 ORM 模型介绍

当 Java 作为一门语言出现时，一个比较重要的工作任务就是要操作数据库。这就不可避免地要涉及数据库的接口。Sun 公司从 1996 年就把 JDBC 加入了 Java 平台的 1.1 版本之中，将其作为 RDBMS 资源管理和数据访问的标准低级抽象层。但是，直接使用 JDBC API 是相当麻烦的，并且开发的效率也比较低。这样，Java ORM 模型就横空出世了。

1.2.1 什么是 ORM

O/R Mapping 全称 Object Relation Mapping，即对象关系映射，把对数据表映射为对象类，将在数据库中直接进行的原始操作演变为对类的属性和方法的操作，而间接更改数据表的数据。

通常，实现 ORM 框架一般包括以下四部分：

- ① 对映射类进行 CRUD（新增、查询、修改和删除）操作的 API；
- ② 规定 Object 与 Relational 之间的映射规则，一般采用 metadata 进行表示；
- ③ 规定类和类属性相关的查询规则；
- ④ 实现 ORM 中对数据库操作的事务管理。

实现 ORM 框架也有很多其他的扩展功能，包括缓存处理、分布式事务管理、多种数据库之间的无缝迁徙等等。

1.2.2 ORM 的实现方式

关系数据模型则基于数学原理，特别是集合论的原理，这些原理都能在数学理论上得到完全的证明。而对象模型基于程序设计的一些原则，类同于一种解决问题的工具，并没有得到公理或数学理论的支持。面向对象设计的目标是通过把一个业务过程分解成具有状

态和行为的对象来进行建模的业务过程,而关系数据库的设计目标是规范化数据以消除数据冗余度。两种不同的理论基础导致对象模型与关系数据模型之间的阻抗不匹配。面向对象模型和关系数据模型的阻抗不匹配使得进行对象关系映射时存在着许多问题。关系数据库不支持诸如类、继承、封装和多态等面向对象的概念。对象关系映射是在对象与数据库之间进行映射的一种技术,对象间的关系、类及其属性必须以某种方式映射为关系数据库中的数据库关系、表和字段。

针对对象模型与关系数据模型间的阻抗不匹配,人们提出了一些解决方案来实现对象模型向关系数据库模型的映射,由于数据库是以二维表为基本管理单元的,所以在使用关系数据库的面向对象软件应用中对象模型最终是由二维表以及表间关系来描述的,而这其实就是一个类与数据库表的变换过程。从某些方面看来,在对象与关系数据库表的行之间有着很多共同点。对象是类的实例,它的内部数据的结构以及其他一些特征由某个类定义。类似地,关系数据库的脚本定义了数据库表的结构,例如表包含哪些字段等。类的实例与数据的行以相似的方式存储数据。通常一个类可以映射为一个或一个以上的关系数据库的表,类属性将映射成关系数据库中的字段,而对象间的关系可以通过使用关系数据库中的外键来维护。对象关系映射的解决方案一般是把每个对象映射到数据库表的单个行上,这一行通常来自一个表,也可以由一个表的连接操作产生。

1. ORM 中对对象与数据库表之间的映射机制

ORM 的实现方式也就是 O/R Mapping 的映射机制,一般有以下几种情况。

(1) 类属性和数据库的数据表与列建立一种随机的映射关系

也就是说,对象类和数据库表并非一一对应,同时对象类中的属性也不是与数据库表中的列并非一一对应。一个类属性可对应 1 或多个实体表的字段。同样,一个实体表可以对应 1 个或多个实体类的属性。这种实现模式主要还是根据业务逻辑来划分对象的,一方面一个业务逻辑类可以获取一个数据库表的一部分字段,同时还要获取另一个或几个数据库表的部分或全部字段。另一方面一个数据库表可以映射成为多个对象,分别应用在多个业务实体类中。这种方式的好处有两点,一是映射的对象结构简单、易于使用,二是避免每次构造对象的时候传送大量的不相关的数据。

当然,基于这种模式,存在实现对象在数据库中的唯一标识的问题。其实这有两种解决方案,第一种解决方案还是采用数据库主键来实现唯一标识。如果一个实体类映射的是一个数据库表,可以采用数据库表的主键来形成实体类的唯一标识。如果一个实体类映射的是多个数据库表,可以把多个数据库表的主键组合形成一个实体类的唯一标识。第二种解决方案是采用无业务意义的字段 OID 作为各个实体对象的主键。这样 OID 也作为类与数据库映射时的对象的唯一标识。

(2) 实体类和数据库表一一映射

这是一种最简单的实现模式,即数据库表与对象一对一,即一个数据库表映射为一个 Java 对象。所有的表字段(field)映射为 Java 对象的属性(attribute)。但是不同层次的实

体类映射到数据表时，应根据数据库表的关系来进行。这样也有三种模式。

① 一个类层次对应一个数据表

这里所说的一个类层次，指的是父类及其所有子类。将父类和子类中有持久性需求的属性设置为同一数据表的字段。此方法实现起来简单，并且较好地支持多态。因为不同的子类可以用一个标志位加以区分。子类实例的转换比较容易实现，但是，类与数据库之间的耦合程度高。由于子类所特有的属性被所有类层次中的对象所共有，数据库中冗余字段较多。

② 一个实体类对应一个数据表

各个子类所特有的属性，联合从父类中继承的公共属性，构成表的结构。父类不映射为数据库中的实体表，它只作为子类公共属性的载体。这种映射模型使得类属性值的保存和对对象还原实现方便。缺点同样是类结构与数据库的耦合程度高，特别在父类属性变列时，所有从此继承下来的子类都需要进行变更。此外，对多态性的支持也较差。子类的角色转换需要在子类对应的数据表之间准确地传递适当的属性值，同时，需要赋予新的 OID。这种情况下，就不如第一种映射模型，用同一个 OID，在同一张表内就可以实现不同子类之间的转换。

③ 一个类对应一个数据表

无论是父类还是子类，只要类中的属性有持久性保存的需要，就将类映射到数据表。子类的表以父类类表的 OID 为外键。在关系数据库中最大程度地实现了类的多态性。与前两者比较，对象与数据库的耦合程度是最低的，某一个类属性的变更引起的表结构变动最少。这种方法的缺点在于生成子类实例时，继承层次多的子类的属性值还原很困难。由于子类的公共属性包含在父类对应的数据表中，当需要单独获取子类实例时，需要从多个数据表中获取数据合成完整的实例。当类的层次较多时，子类的访问可能会成为系统与数据库交互的瓶颈。

(3) 实体类和数据库视图映射

第 3 种映射方式是根据数据库视图来构造对象，数据库视图是一种虚拟表，它可以合并多个数据库表，然后再把这个数据库视图映射成一个实体对象。最为常见的是在数据库中体现为主从表的结构，映射成一个实体对象。

2. 类间关系映射为键值

数据库表的约束决定了实体类关系中的关联和聚合。

一对一、一对多的关联是通过在关联的某一方（一般在关联角色多重性 ≥ 1 的一方）引用对方的 OID，并根据关系的紧密程度为外键的字段加上非空以及唯一性的约束。在生成相关联的类实例时，可根据外键自动获取另一方的实例。多对多的关联需要创建关联表。关联表是统一的以 OID 作为主键，同时以关联角色双方的 OID 作为联合外键。

聚合关系映射到数据库中通常所说的主附表结构类似，在聚合关系中的子类对应的数据表中含有指明父类的 OID 的域。特别是强制型聚合（或称之为组合），子类单独存在

是没有意义的，必须与父类同时存在、同时消亡。当然，数据库键值本身只是这一关系的体现。

数据库表之间主键外键关联映射的关系对象模型中存在着三种关系，分别是一对一、一对多、多对多。体现在数据库中分别是一个主键对应一个外键，一个主键对应多个外键，中间表（join table）。

① 一对一：这种情况可以直接映射成在对象之间保持一个引用关系，一个对象持有对另一个对象的引用。具体体现在对象中实现一个方法，该方法的返回值即它的一对一关联对象。如果这种关系是双向的，那么必须在两个对象中都实现这样一个方法，如果关系是单向的，那么由方向性决定在哪个对象中实现。

② 一对多：这种关系有两种类型，分别是关联（association）和聚合（aggregation）。对于关联，仍然体现为引用关系。不同于一对一的是，一方对象持有的是一个集合的引用，该集合中的元素是多方对象，多方对象持有的是一方对象的引用。一方对象实现的方法的返回值是一个集合，集合中的元素是多方对象；多方对象实现的方法的返回值就是一方对象。当然可根据业务需求决定关系的方向性，从而决定在哪个对象中实现对应的方法。对于聚合，需要在一方对象中增加一个集合的属性，该集合中的元素为多方对象。同时一方对象的增、改、删、查也要加入相应的操作以实现多方对象的对应操作。而多方对象的实现可以根据实际需求来决定是否需要实现独立的增、改、删、查方法。

③ 多对多：这种关系可以看作是一个双向的一对多，都把自己看作是一方，对方看作是多方。两个对象分别持有一个集合的引用，集合中的元素即为对方对象。如果将数据库的中间表映射成一个对象，那么可以将多对多关系的两个对象分别实现为对应中间表对象的一对多，即这两个对象都看作是一方，而中间表对象则看作是多方。从而利用一对多的实现方式去实现这种关系。

3. 面向对象操作映射为数据库操作

ORM 框架的一项重要工作就是将数据库的操作封装成实体类的方法。其好处就在于封装了操作的细节，开发人员根本不用关心如何去连接数据库，如何发送 SQL 语句，如何取各个字段，他只需调用一个 CRUD（Create、Read、Update 和 Delete）方法，该方法完成一系列的底层操作，返回的是一个构造好的对象，然后他就可调用相应的 get 方法取得他所需要的字段的数值。

ORM 框架的方法也就是增、改、删、查等基本操作，而实体对象属性也是由数据库表中的字段所构成的。ORM 框架在构造过程中，实际上把 CRUD 方法，根据对象属性与数据库表的映射字段转化为数据库操作中的 insert、select、update 和 delete 等 SQL 语句。

4. 事务管理

由于 ORM 框架封装了数据库的存储操作，软件开发人员不能处理底层存储细节，没有利用数据库的事务管理机制的可能。可以通过两种办法解决这个问题，一是采用独立的

事务处理机制，不采用数据库本身的事务管理。二是在映射对象中重载多个保存方法，仍然采用数据库的事务管理机制，使得某一个重载的方法可以提供给软件开发人员事务处理的能力。在 Java 语言中，可以由软件开发人员提供一个 `java.sql.Connection` 接口，将这个接口作为一个参数传给保存的方法，从而可以使得软件开发人员有手工控制事务的可能。

5. ORM 的性能

ORM 框架需要频繁地跟数据库交互，以下几个方面对性能有影响。

① 数据库的连接：数据库的连接对象是非常昂贵的资源，不同的获取数据库连接的方式将对性能产生极大的影响。这可以采用连接池的办法来解决，每次从连接池中获取数据库连接，将极大地提高性能。

② 大量数据的传输：由于应用程序使用的是对象，而对已有数据库记录的对象生成需要构造它的所有属性，也就是要将该条记录所有字段的数值赋给这个对象。而业务逻辑有可能仅仅关心其中几个字段的数值，其他字段数值的传输可以说是浪费的，因而可以采用分段获取的模式。

③ 对象的频繁获取：每次应用程序使用已有的持久化对象都要查询一次数据库，然后构造这个对象，对于需要频繁使用的对象来说，每次都需要查询数据库。对于这种情况，可以采用缓冲的机制，将一些频繁使用的对象缓冲起来，再次使用的时候先从缓冲池里面查找，如果没有找到，然后再查询数据库。

1.2.3 常用的 ORM 框架

常见的 ORM 框架包括 Hibernate、iBatis、TopLink、Castor JDO、Apache OJB 等，分别介绍如下。

1. Hibernate

Hibernate 是一个开放源代码的 O/R Mapping（对象关系映射框架），它对 JDBC 进行了轻量级的对象封装，使 Java 程序员可以随心所欲地使用对象编程思维来操纵数据库。其官方网址：<http://www.hibernate.org>。

2. iBatis

iBatis 也是开放源代码的 O/R Mapping，但这是一种“半自动化”的 ORM 实现。所谓“半自动”，iBatis 以 SQL 开发的工作量和数据库移植性方面的让步，为系统设计提供了更大的自由空间。其官方网址：<http://ibatis.apache.org/>。

3. TopLink

TopLink 是 Java 对象关系可持续性体系结构，原属于 WebGain 公司的产品，现在被 Oracle 收购，并重新包装为 Oracle AS TopLink。TopLink 为在关系数据库表中存储 Java 对

象和企业 Java 组件 (EJB) 提供了高度灵活和高效的机制。TopLink 提供了一个持久性基础架构, 使开发人员能够将来自多种体系结构的数据 (包括 EJB、CMP 和 BMP)、POJO、servlet、JSP、会话 Bean 和消息驱动 (Bean) 集成在一起。

4. Entity Bean

Entity Bean 它提供了一个持久性数据的面向对象的表示。不同于对象关系映射, Entity Bean 对于关系数据库没有限制; 它描述的持久性信息可以来自一个企业信息系统 (EIS) 或者其他的存储设备。

5. Castor JDO

Castor JDO 是 ExoLab Group 下面的一个开放源代码的项目, 它最大的特色就是实现了大部分的 ODMG OQL 规范, 其原理是通过 Java 反射 API 去实现属性的设置和读取。它的主要 API 和数据接口为: JDO-like、SQL、OQL、JDBC、LDAP、XML、DSML。它支持分布式目录事务处理和时间; 提供处理 XML、Directory、XADirectory 的类库, 提供从 XML 到 Java 类的转换机制。其官方网址: <http://castor.exolab.org>。

6. Apache OJB

Apache OJB (Object Relational Bridge) 是 Apache 下面的一个开放源代码的项目。Apache OJB 是一种对象关系映射工具, 能够完成从 Java 对象到关系数据库的透明存储。OJB 使用基于 XML 的对象关系映射, 映射发生在一个动态的元数据层, 使得通过一个简单的元对象协议 (MOP) 在运行时就可以操作元数据层去改变存储内核。其官方网址: <http://db.apache.org/ojb/>。

7. Torque

Apache Torque 是一个使用关系数据库作为存储手段的 Java 应用程序持久化工具。Torque 是 Apache 下面的一个开源项目, 由 Web 应用程序框架 Jakarta Apache Turbine 发展而来, 但现在已完全独立于 Turbine。

1.2.4 ORM 模型和持久层框架

现在很多书籍都把 ORM 模型和持久层作为一个概念来进行说明。笔者认为这两者还是有区别的。这里有几个基本概念要区分, 包括持久化、持久层、ORM 等。

持久化, 英文即 “Persistence”, 就是把内存中的数据对象保存到可永久保存的存储设备中。持久主要应用是将内存中的数据存储在关系型的数据库中, 当然也可以指存储在磁盘文件等。持久层就是专注于实现数据持久化应用领域的某个特定系统的一个逻辑层面, 将数据使用者和数据实体相关联。

O/R Mapping 是把对数据表映射为对象类, 或者把对象类映射为数据表。

在细微的比较上两者还是有一点差异的。

① ORM 模型主要是实现对象和关系数据直接的映射。持久层是对象处理和关系数据库处理中的一个交互层,在这个交互层主要是实现内存数据与硬盘数据的一致性和统一性。

② ORM 模型是一种实现手段,而持久层是一种实现技术。

③ 持久层框架的概念要比 ORM 模型框架要大,即持久层框架一般都是包括 ORM 框架,而 ORM 只是实现持久化技术中的一种。

从严格意义上讲,iBatis 应该趋向于是 ORM 框架。这样才归属持久层框架。在这里简单地介绍一下几种持久层框架的实现方式。

① 主动域的对象模式

主动域的对象模式是在实现中封装了关系数据模型和数据访问细节的一种形式。在 J2EE 架中,EJB 组件分为会话 EJB 和实体 EJB。会话 EJB 通常实现业务逻辑,而实体 EJB 表示业务实体。实体 EJB 又分为两种:由 EJB 本身管理持久化,即 BMP (Bean-Managed Persistence);有 EJB 容器管理持久化,即 CMP (Container-Managed Persistence)。BMP 就是主动域对象模式的一个例子,BMP 表示由实体 EJB 自身管理数据访问细节。主动域对象本身位于业务逻辑层,因此采用主动域对象模式时,整个应用仍是三层应用结构,并没有从业务逻辑层分离出独立的持久化层。

② JDO 模式

JDO (Java Data Objects) 规范是 Sun 公司制定的描述对象持久化语义的标准。严格地说,JDO 并不是对象-关系映射接口,因为它支持把对象持久化到一种存储系统中,包括关系数据库、面向对象的数据库、基于 XML 的数据库,以及其他专有存储系统。由于关系数据库是目前最流行的存储系统,故许多 JDO 的实现都包含了对象-关系映射服务。

③ CMP 模式

在 J2EE 架构中,CMP (Container-Managed Persistence) 表示由 EJB 容器来管理实体 EJB 的持久化,EJB 容器封装了对象-关系的映射及数据访问细节。CMP 和 ORM 的相似之处在于,两者都提供对象-关系映射服务,都把对象持久化的任务从业务逻辑中分离出来。区别在于 CMP 负责持久化实体 EJB 组件,而 ORM 负责持久化 POJO,它是基于普通的 JavaBean 形式的实体域对象。

1.3 iBatis 的组件和实现的功能

iBatis 框架主要包含两类组件:SQL Map 组件和 DAO 组件。

1.3.1 iBatis 的 DAO 组件

iBatis DAO 组件的主要功能是帮助开发人员基于 DAO 设计模式设计和开发 Java EE

应用程序。DAO 框架的主要目标是抽象化应用程序的数据访问层和持久层的表示方式和位置，使它远离应用程序的业务逻辑。同时 DAO 框架允许在应用程序中定义负责数据中心操作的接口，使用 DAO 可以动态地配置应用程序，从而使用不同的持久性机制和隐藏持久性层的实现细节。例如，如果应用程序通过 JDBC 来获取持久性，则 DAO 框架的目标就是抽象 Connection、PreparedStatement 和 ResultSet 等类和接口的使用，并下移到持久层；如果应用程序使用 get 和 post 来获得和存储数据，则 DAO 框架的用途是实现抽象化的使用，使它们远离应用程序的业务层。然后应用程序可以使用 DAO 接口在数据上执行操作，这样就可以从数据库、Web 服务或其他任何数据源中获得数据。iBATIS DAO 实现如图 1-1 所示。

iBATIS DAO 实现的目的如下：

- ① 为二次开发人员实现核心 Java EE 设计模式的 DAO 模式提供了一个基础平台。
- ② 抽象出数据访问方式，隐藏了实现细节；
- ③ 封装了多种 ORM 模型，屏蔽了持久层。使得 iBATIS DAO 支持 iBATIS SQL Map、Hibernate、Apache Ojb、Toplink、JDBC 和 JTA。
- ④ 由于对数据库的操作都是由 DAO 代理实现的，这样可以使系统更具可维护性。
- ⑤ 当进行多种 ORM 模型组合时，不用修改代码。
- ⑥ 增强了缓存处理，提高了安全性。

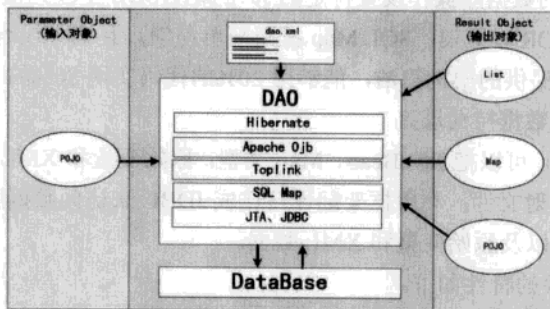


图 1-1 iBATIS DAO 实现图

1.3.2 iBATIS SQL Map 组件

iBATIS SQL Map 组件是 iBATIS Database Layer 框架的重要组成部分，是整个 iBATIS Database Layer 的核心所在。SQL Map 能够大大减少访问数据库的代码。iBATIS SQL Map 使用简单的 XML 配置文件将 JavaBean、XML、Map 映射成 SQL 语句，通过 SQL 语句的执行而获得 Java Bean、XML、Map、List 等对象。iBATIS SQL Map 实现如图 1-2 所示。

iBATIS SQL Map 提供了一个简洁的框架。输入参数使用简单的 XML 配置文件映射成 JDBC 的查询对象（Statement）——用来绑定要执行的操作的对象，然后生成结果。iBATIS SQL Map 的实现的功能如下。

① 该组件基于 XML 配置文件，实现底层数据操作的 SQL 可配置化，可以控制最终的数据操作方式，通过 SQL 的优化获得最佳的数据库执行效能，在系统设计上具有比较大的自由空间。这在 SQL 自动生成的“全自动化”ORM 机制中是所难以实现的。

② SQL 语句的输入和输出参数可以是基本类型的包装类和简单类（如 Integer、String 等），也可以是 Map、JavaBean、XML 文件等，还可以直接使用应用中更为复杂的类（例如值对象 VO、数据传输对象 DTO 等）。

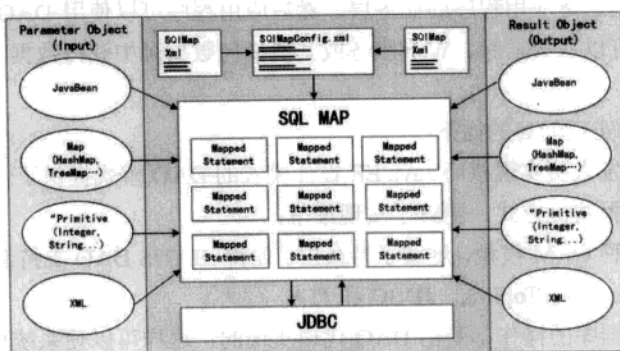


图 1-2 iBATIS SQL Map 实现图

③ 可针对特定的数据库操作设置特定的 Java 属性/SQL 字段列值映射。

④ 相对于其他 ORM 框架，SQL Map 框架简单易学，且编程代码简练。

⑤ SQL Map 所提供的架构简洁，能够用 20% 的代码实现 80% 的 JDBC 的功能，也就能够非常方便地实现数据持久层。

从图 1-2 中看到，可以把 JavaBean、Map 类型，原始变量和 XML 数据作为传入对象；通过配置文件载入映射文件；利用框架翻译转化成 JDBC 来访问数据库；执行结果可以是 JavaBean、Map 类型以及原始变量和 XML 数据。

iBATIS 几个主要的特性如下。

① 能够管理对象持续性。JavaBean 到数据库表的映射，以对象的方式存取数据。O/R Mapping 的定义都是基于 XML，具有很好的扩展性和通用性。

② 支持多种数据输入和输出类型，包括 Java 基本类型、JavaBean、Map、XML 等。

③ 支持静态 SQL 语言、参数 SQL 语言和动态 SQL 语言。

④ 支持新增、修改、删除、查询和存储过程的操作。

⑤ 支持显式事务和隐式事务，支持 JDBC、JTA 和容器事务。

⑥ 允许用户使用定制的 SQL 来提高查询的性能，提供了多种 SQL 自动策略开关。

⑦ 提供了灵活的 Cache 缓冲机制，以及延迟加载、复合查询的策略，保证一般应用的性能不会低于相应的数据集应用。

⑧ 使用 iBATIS 框架可实现数据库平台无关性，可以随时切换开发及数据库发布平台，方便移植。这些数据库包括 Oracle、MS SQL Server、MySQL 等。

第 2 章

相关的技术背景和基础知识

本章内容:

1. 介绍面向对象和 UML 基础知识, UML 是分析源码的主要工具, 尤其是通过 UML 图来进行演示。同时, 对于类与类之间、接口与接口之间、类和接口之间的关系进行了详细的描述, 这也是为将来进行 iBATIS 各个组件的框架分析提供基础知识。
2. 介绍 iBATIS 所涉及的 Java 基础知识, 这些基础知识包括 Java 的 I/O 操作、XML 操作、线程管理、反射机制、动态代理、JDBC、JavaBean 和 JNDI 等。
3. 介绍 iBATIS 所涉及的数据库基础知识, 主要是 SQL 语句和事务管理。
4. 介绍 iBATIS 所涉及的 Java EE 规范, 主要是 JTA 规范。
5. 介绍 iBATIS 所涉及的开源 ORM 框架, 包括 Hibernate、TopLink、Apache OJB 等。
6. 介绍 iBATIS 所涉及的其他开源框架, 主要是 Log 相关的开源框架、OSCache、commons-dbcp 等。
7. 介绍 iBATIS 在设计框架中涉及的 GoF 设计模式。

要了解 iBATIS, 首先还要具备一些基础知识。这些基础知识在以后阅读本书是非常有帮助的, 现在简单介绍如下。

2.1 面向对象和 UML 基本知识

面向对象 (Object Oriented, 简称 OO) 是当前计算机界关心的重点, 是目前软件开发方法的主流。本节主要介绍对象、UML、UML 图和 UML 的类图。

2.1.1 面向对象基础

对象是人们要进行研究的任何事物, 从最简单的整数到复杂的宇宙飞船均可看作对

象，它不仅能表示具体的事物，还能表示抽象的规则、计划或事件。

1. 面向对象分析的特征

面向对象分析（Object Oriented Analyzing，简称 OOA）的核心是对象的概念，它通过对所研究的事物进行高度的抽象得到对象，每个对象都真实地反映了它所对应的事物，事物的行为用其显露的函数接口来表征，具有相同结构、相同操作并遵守相同约束规则的对象聚合成对象类。对象类间具有层次结构关系，处于上层的对象类称为父类或基类，下层的类称为子类。OOA 方法具有抽象性、封装性、继承性、多态性四大特点。

① 抽象性（Abstract）

抽象是指强调实体的本质、内在的属性。在系统开发中，抽象指的是在决定如何实现对象之前，对对象的意义和行为进行概括。使用抽象可以尽可能避免过早考虑一些细节。类实现了对对象的数据（即状态）和行为的抽象。同时抽象忽略一个主题中与当前目标无关的那些方面，以便充分地注意与当前目标有关的方面。抽象包括两个方面：一是过程抽象，二是数据抽象。过程抽象是指任何一个明确定义功能的操作都可被使用者当作单个实体看待，尽管这个操作实际上可能由一系列更低级的操作来完成。数据抽象定义了数据类型和施加于该类型对象上的操作，并限定了对象的值只能通过使用这些操作进行修改和观察。

② 继承性（Inheritance）

继承是一种联结类的层次模型，能直接获得已有的特性而不必重新定义，并且允许和鼓励类的重用。它提供了一种明确表述共性的方法，在定义一种新的对象时，只需指明它具有哪些类定义以外的新特性，而不必定义新对象的全部特性，保证了软件的可重用性，使对象之间具有传递作用，子类能继承各层父类的全部语义特性。也就是说，一个类的数据和方法可以传给另一个类，对象的一个新类可以从现有的类中派生，这个过程称为类继承，它是面向对象语言中可重复使用的基础。派生类继承了原始类（基类）的特性，同时可以修改或增加新的特性。这体现了一般和特殊的关系。继承性很好地解决了软件的可重用性问题，有利于提高软件的开发效率。

③ 封装性（Encapsulation）

封装是面向对象的特征之一，是对象和类概念的主要特性。封装是把过程和数据包装起来，实现信息的隐蔽，禁止其他对象直接访问对象的内部状态，保证了对单元数据的封装和隐藏，使得每个单元对外部都有清晰的边界，我们只看到封装界面的信息，对数据的访问只能通过已定义的界面。具体地说，封装就是隐藏信息，就好比我们会操作电脑，无须知道电脑内部结构的组成。封装保证了模块具有较好的独立性，使得程序维护修改较为容易。由于数据和代码封装在对象中，不易破坏，封装的对象保证系统模块化不会互相影响。

封装性是保证软件部件具有优良模块性的基础。面向对象的类是封装良好的模块，类定义将其说明（用户可见的外部接口）与实现（用户不可见的内部实现）显式地分开，其内部实现按其具体定义的作用域提供保护。对象是封装的最基本单位。封装防止程序相互依赖性而带来的变动影响。面向对象的封装比传统语言的封装更为清晰、更为有力。

④ 多态性 (Polymorphism)

多态性是指允许不同类的对象对同一消息做出响应。多态性的一个操作可以被类层次中上下成员所共享,层次中每个类都以适合自己的方式实现这个操作,可在基类中定义所有相同的特性,由派生类实现某处特定的操作,允许每个对象以适合自身的方法去响应共同的消息。多态性包括参数化多态性和包含多态性。多态性语言具有灵活、抽象、行为共享、代码共享的优势,很好地解决了应用程序的函数同名问题。例如,一个应用程序包括许多对象,这些对象也具有同一类型的工作,但是即以不同的做法来实现,不必为每个对象的过程取一过程名,造成复杂化,使得同一类型的工作有相同的过程名,使过程名复用。从而增强了对对象行为的透明性和可维护性,实现软件的简洁性和一致性。

2. 对象和类的属性

对象的状态和行为。对象具有状态,一个对象用数据值来描述它的状态。对象还有操作,用于改变对象的状态,对象及其操作就是对象的行为。这样,对象实现了数据和操作的结合,使数据和操作封装于对象的统一体中。

面向对象中一个重要的概念就是类。所谓类,就是具有相同或相似性质的对象的抽象。因此,对象的抽象是类,类的具体化就是对象,也可以说对象是类的实例。类具有属性,它是对象的状态的抽象,用数据结构来描述类的属性。类具有操作,它是对象的行为的抽象,用操作名和实现该操作的方法来描述。

类是有结构的,在客观世界中有若干类,这些类之间有一定的结构关系。通常有两种主要的关系,即一般—具体结构关系,整体—部分结构关系。一般—具体结构称为分类结构,也可以说是“或”关系,或者是“is a”关系。整体—部分结构称为组装结构,它们之间的关系是一种“与”关系,或者是“has a”关系。

面向对象还有两个概念,即消息和方法。对象之间进行通信的结构叫做消息。在对象的操作中,当将一个消息发送某个对象时,消息包含了驱动接收对象去执行某种操作的信息。发送一条消息至少要包括说明接受消息的对象名、发送给该对象的消息名(即对象名、方法名)。一般还要对参数加以说明,参数可以是认识该消息的对象所知道的变量名,或者所有对象都知道的全局变量名。

2.1.2 UML 基础知识

UML 统一建模语言 (Unified Modeling Language) 是一种建模语言,是第三代用来为面向对象开发系统的产品进行说明可视化和编制文档的方法。它是由信息系统 Information System 和面向对象领域的三位著名的方法学家 Grady Booch、James Rumbaugh 和 Ivar Jacobson (称为三个好朋友 the Three Amigos) 提出的一种建模语言,得到了 UML 伙伴联盟 (Unified Modeling Language) 的应用与反馈,该组织于 1996 年由 Rational 公司创立。对象管理组织 (OMG) 于 1997 年 11 月采纳了 UML 规范。UML 规范目前最新的版本是

UML2.2。UML 是多种方法相互借鉴、相互融合、趋于一致、走向标准化的产物。这样的统一建模语言将为软件开发商及其用户带来诸多便利。美国、日本等计算机技术发达国家已有大量的软件开发组织开始用 UML 进行系统建模。学习和使用 UML 已经成为一种潮流。

通过把这些先进的面向对象思想统一起来，UML 为公共的、稳定的、表达能力很强的、面向对象的开发方法提供了基础。UML 是一种标准的图形化建模语言，它是面向对象分析与设计的一种标准表示。它不是一种可视化的程序设计语言，而是一种可视化的建模语言；它不是工具或知识库的规格说明，而是一种建模语言规格说明，是一种表示的标准。它不是过程也不是方法，但允许任何一种过程和方法使用它。UML 的目标是以面向对象图的方式来描述任何类型的系统，具有很广阔的应用领域；其中最常用的是建立软件系统的模型，但它同样可以用于描述非软件领域的系统，如机械系统、企业机构或业务过程，以及处理复杂数据的信息系统、具有实时要求的工业系统或工业过程等。而且，UML 适用于系统开发的不同阶段，从需求规格描述直至系统完成后的测试和维护；因此，它是一个通用的标准建模语言，可以对任何具有静态结构和动态行为的系统进行建模。

UML 的目标是：

- 易于使用、表达能力强、进行可视化建模；
- 与具体的实现无关，可应用于任何语言平台和工具平台；
- 与具体的过程无关，可应用于任何的软件开发过程；
- 简单并且可扩展，具有扩展和专有化机制，便于扩展，无须对核心概念进行修改；
- 为面向对象的设计与开发中涌现出的高级概念，例如协作框架模式和组件提供支持，强调在软件开发中对架构框架模式和组件的重用；
- 与最好的软件工程实践经验集成；
- 可升级，具有广阔的适用性和可用性；
- 有利于面向对象工具的市场成长。

2.1.3 UML 图

UML 图 (diagram) 是一组元素的图形表示，大多数情况下把图画成顶点 (代表事物) 和弧 (代表关系) 的连通图。为了对系统进行可视化，可以从不同的角度画图；这样，图是对系统的投影。除了非常微小的系统外，图是系统组成元素的省略视图。UML 包含了九种这样的图，它们是类图、对象图、用例图、顺序图、协作图、状态图、活动图、构件图和实施图，下面对它们进行一一介绍。

(1) 类图

类图 (Class Diagram) 展现了一组对象、接口、协作和它们之间的关系。在对面向对象系统的建模中所建立的最常见的图就是类图。类图给出系统的静态设计图，包含主动类的类图给出了系统的静态进程视图。由于类图描述系统中类的静态结构，它不仅定义系统

中的类，表示类之间的联系，如关联、依赖、聚合等，也包括类的内部结构（类的属性和操作）。类图描述的是一种静态关系，在系统的整个生命周期内都是有效的。

（2）对象图

对象图（Object Diagram）是类图的实例，几乎使用与类图完全相同的标识。对象图和类图一样，给出系统的静态设计视图或静态进程视图，但它们是从真实的或原型案例的角度创立的。对象图和类图的不同点还在于对象图显示类的多个对象实例，而不是实际的类。一个对象图是类图的一个实例。由于对象存在生命周期，因此对象图只能在系统某一时间段内存在。

（3）用例图

用例图（Use Case Diagram）展现了一组用例、参与者（一种特殊的类）及其它它们之间的关系。用例图给出系统的静态用例视图，这些图对于系统行为进行组织和建模是非常重要的。

（4）顺序图

顺序图（Sequence Diagram）是一种交互图（Interaction Diagram）。交互图展现了一种交互，它由一组对象和它们之间的关系组成，包括在它们之间可能发送的消息。顺序图显示对象之间的动态合作关系，它强调对象之间消息发送的顺序，同时显示对象之间的交互。

（5）协作图

协作图（Collaboration Diagram）也是一种交互图，描述对象间的协作关系，协作图跟顺序图相似，显示对象间的动态合作关系。除显示信息交换外，协作图还显示对象以及它们之间的关系，它强调收发消息的对象的结构组织。如果强调时间和顺序，则使用顺序图；如果强调上下级关系，则选择合作图。这两种图合称为交互图，顺序图和协作图是同构的，这就意味着它们是可以相互转化的。

（6）状态图

状态图（Statechart Diagram）展现了一个状态机，它由状态、转换、事件和活动组成，它描述类的对象所有可能的状态以及事件发生时状态的转移条件。它专注于系统的动态视图，对于接口、类或协作的行为建模非常重要。通常，状态图是对类图的补充。在实用上并不需要为所有的类画状态图，仅为那些有多个状态且行为受外界环境的影响并且发生改变的类画状态图。

（7）活动图

活动图（Active Diagram）是一种特殊的状态图，它展示了在系统内从一个活动到另一个活动的流程。活动图专注于系统的动态视图，它对于系统的功能建模特别重要，并强调对象间的控制流程。

（8）构件图

构件图（Component Diagram）展现了一组构件之间的组织和依赖。构件图专注于系统的静态实现图，描述代码部件的物理结构及各部件之间的依赖关系，它与类图相关，通常把构件映射成一个或多个类、接口和协作。一个构件可能是一个资源代码构件、一个二

进制构件或一个可执行构件，它包含逻辑类或实现类的有关信息。构件图有助于分析和理解构件之间的相互影响程度。

(9) 配置图

配置图 (Deployment Diagram) 展现了对运行时处理节点以及其中的构件的配置。配置图给出了体系结构的静态实施视图，定义系统中软硬件的物理体系结构，显示实际的计算机和设备 (用节点表示) 以及它们之间的连接关系，也可显示连接的类型及构件之间的依赖性。它与构件图相关，通常一个节点包含一个或多个构件，在节点内部，放置可执行部件和对象以显示节点跟可执行软件单元的对应关系。

2.1.4 类和接口以及之间的关系

UML 的类图中类与类、类和接口的关系有很多种说法，有的书籍上说这些关系分为关联、聚合/组合、依赖、泛化 (继承)，或者说分类标准有泛化、关联、依赖、聚合。也有的书说是依赖、关联、泛化和实现，还有的书说关系包括一般化关系 (泛化)、关联关系、聚合关系、合成关系和依赖关系。综合上述的说法，类关系应该有依赖、泛化、关联、聚合、组合、继承、实现等。通过进行统一规范有助于理解和记忆 UML 类的规范。所以，本节主要介绍 UML 中类与类之间以及类与接口之间的一般化关系 (泛化)、关联关系、聚合关系、合成关系和依赖关系等。然后通过 UML 图和代码来表示这些关系，并说明这些关系之间的内部联系。

1. UML 类关系的图示和说明

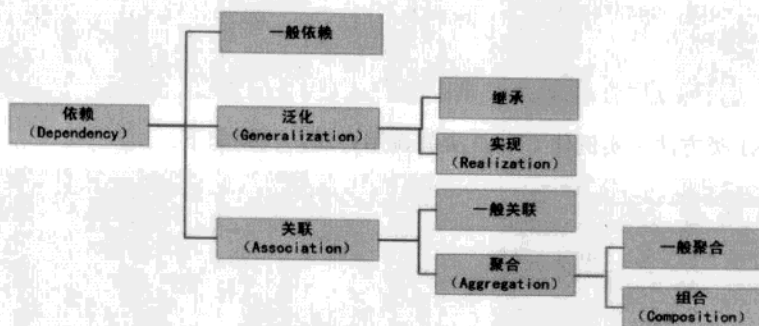
按照标准定义，依赖 (Dependency) 是两个事物之间的语义关系，其中一个事物 (独立事物) 发生变化会影响另一个事物 (依赖事物) 的语义。由于泛化 (Generalization) 和关联 (Association) 都有这种含义，所以，依赖包括泛化关系和关联关系。

泛化 (Generalization) 是一种特殊/一般关系，特殊元素 (子元素) 的对象可以代替一般元素 (父元素) 的对象。用这种方法，子元素可共享父元素的结构和行为。对于类与类之间或者接口与接口之间这种关系叫继承关系，对于接口和类之间，这种应该叫实现 (Realization) 关系。所以，泛化包括继承关系和实现关系。

关联 (Association) 是一种结构关系，它是类与类之间的连接，它使一个类知道另一个类的属性和方法。聚合 (Aggregation) 是一种特殊类型的关联，它描述了整体和部分之间的关系。组合关系 (Composition)，有的书籍也叫合成关系，是一种特殊的聚合关系，它要求普通的聚合关系中代表整体的对象负责代表部分对象的生命周期，组合关系不能共享。

整个 UML 类和接口之间的关系，还是用一个图来说明，具体如图 2-1 所示。

在图 2-1 中，可以明确看出类关系最开始的节点是依赖，如果一个类连依赖关系都不存在，则如何存在什么泛化和关联。下面分别介绍这些关系。



2. 依赖 (Dependency)

(1) 依赖 (Dependency) 定义

依赖关系是类与类之间的连接，表示一个类依赖于另一个类的定义。例如如果 A 依赖于 B，则 B 体现为局部变量，方法的参数、或静态方法的调用。依赖关系有如下三种情况：

① A 类是 B 类的一个成员变量；② A 类是 B 类方法中的一个参数；③ A 类向 B 类发送消息，从而影响 B 类发生变化。注意，要避免双向依赖。一般来说，不应该存在双向依赖。

依赖 (Dependency) 图示的表示方法，可使用带实心箭头的虚线表示，如图 2-2 所示。



图 2-2 依赖关系的图示

由图 2-2 可见，依赖关系表示为虚线+箭头，箭头指向依赖的类，说明 Class1 依赖 Class2。

(2) 依赖 (Dependency) 代码

主要还是采用 Java 代码来描述。

① Class2 类是 Class1 类的一个成员变量，这是一种关联关系。一般情况下，都按照关联关系进行标识，代码如下。

```
public class class1 {
    public class2 association1;
}
```

② Class2 类是 Class1 类方法中的一个参数，代码如下。

```
public class class1 {
    public int operation1(class2 parameter1) {
        return 0;
    }
}
```

③ Class1 类方法中调用 Class2 类的方法，代码如下。

```
public class class1 {
```

```
public void operation1() {
    class2.method1();
}
```

④ Class1 类方法中实例化 Class2 类并调用方法，代码如下。

```
public class class1 {
    public void operation1() {
        class2 Object1 = new class2();
        Object1.method1();
    }
}
```

(3) 依赖 (Dependency) 扩展说明

依赖关系包括很多，有 bind、call、derive、extend、friend、import、include、instantiate、refine、sameFile、trace、use 等。call 关系如图 2-3 所示。



图 2-3 依赖关系中的 call 关系

图 2-3 表示 Class1 通过 call 依赖 Class2，即 Class1 调用 Class2。

3. 泛化 (Generalization)

泛化 (Generalization) 是一种特殊/一般关系，特殊元素 (子元素) 的对象可以代替一般元素 (父元素) 的对象。用这种方法，子元素共享了父元素的结构和行为。具体表现为类与类之间的继承关系，接口与接口之间的继承，类对接口的实现关系。

(1) 泛化 (Generalization) 关系图示

表示方法：使用带实心箭头的实线或虚线表示。如果父类是类，则用一个空心箭头+实线，箭头指向父类。如果父类是接口，则用一个空心箭头+虚线，箭头指向父类。有三种表示方式，其中的一种模式还是实现模式。

类与类之间的泛化关系也就是继承关系，如图 2-4 所示。表示为空心箭头+实线，箭头指向父类。图 2-4 说明类 Class1 继承类 Class2。

接口与接口之间的泛化关系同样也是继承关系，如图 2-5 所示。表示为空心箭头+实线，箭头指向父接口。图 2-5 说明接口 Interface1 继承接口 Interface2。

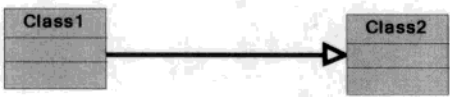


图 2-4 类与类之间的泛化关系



图 2-5 接口与接口之间的泛化关系

接口与类之间的泛化关系是实现关系，如图 2-6 所示。表示为空心箭头+虚线，箭头指

向父接口。图 2-6 说明类 Class1 实现接口 Interface1, Interface1 是父接口, Class1 是子类。

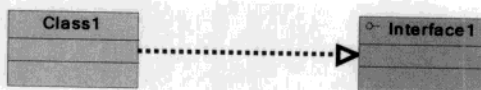


图 2-6 接口与类之间的泛化关系

(2) 泛化 (Generalization) 关系的 Java 代码

① 类与类之间的继承关系的 Java 代码如下。

```
public class Class1 extends Class2 {}
```

② 接口与接口之间的继承关系的 Java 代码如下。

```
public interface Interface1 extends Interface2 {}
```

③ 接口与类之间的实现关系的 Java 代码如下。

```
public class Class1 implements Interface1 {}
```

(3) 泛化 (Generalization) 关系扩展

泛化关系的三个要求:

- ① 子类与父类应该完全一致, 父类所具有的属性、操作, 子类应该都有;
- ② 子类中除了与父类一致的信息以外, 还包括额外的信息;
- ③ 可以使用父类实例的地方, 也可以使用子类的实例。

4. 关联 (Association) 关系

关联关系: 类与类之间的联接, 它使一个类知道另一个类的属性和方法。关联是一种结构关系, 它是类与类之间的连接, 它使一个类知道另一个类的属性和方法。聚合是一种特殊类型的关联, 它描述了整体和部分之间的关系。

关联关系包括类之间的关系和类与接口之间的关系, 接口与接口之间没有关联关系。

关联关系是一种结构关系, 它描述了一组链, 链是对象之间的连接。聚合关系是一种特殊类型的关联关系, 它描述了整体和部分之间的关系。

关联从关联导航可以分为双向关联、单向关联和自身关联三种

(1) 关联关系图示

使用带实心箭头的虚线表示。表示方法: 用实线+箭头, 箭头指向被使用的类, 如图 2-7 所示。

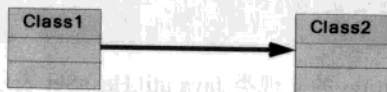


图 2-7 类之间的关联关系

由图 2-7 可见 Class1 关联 Class2, 即 Class1 包含了 Class2。关联关系的 Java 代码描述

如下。

```
public class class1 {
    public class2 association1;
}
```

(2) 关联关系扩展

关联关系分类有两种，一种是多重性分类，另一种是导向分类。

关联关系涉及的对象个数称为关联的重数，反映关联的多重性。按多重性来划分关联可分为以下三种。

① 一对一关联

表示 Class1 的一个对象和 Class2 的一个对象关联，这两个对象具有相同的生命周期，这种关联关系依赖于两个对象而存在，如图 2-8 所示。

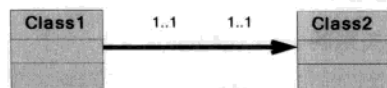


图 2-8 类之间的一对一关联关系

一对一关联的 Java 代码描述如下。

```
public class class1 {
    public class2 association1;
}
```

② 一对多关联

表示类 Class1 的一个对象和类 Class2 的多个对象关联，这种关联关系依赖于一个类 Class1 的对象和一个以上类 Class2 的对象而存在。但类 Class2 的对象可以具有关联动态性，即不仅数目可变，对象次序也可变。而类 Class1 的对象具有确定性，如图 2-9 所示。

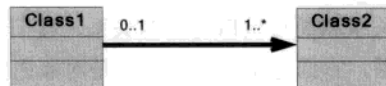


图 2-9 类之间的一对多关联关系

图 2-9 说明 Class2 实例的对象包含了多个 Class1 实例的对象。

一对多关联的 Java 代码有几种情形，一种可能是数组表示，其 Java 代码描述如下。

```
public class class1 {
    public class2[] association1;
}
```

也可能是 Java.util.Collection 的实现类 Java.util.HashSet 表示，其 Java 代码描述如下。

```
public class class1 {
    public Java.util.Collection association1;
}
```

```

public Java.util.Collection getAssociation1() {
    if (association1 == null)    association1 = new Java.util.HashSet();
    return association1;
}

public Java.util.Iterator getIteratorAssociation1() {
    if (association1 == null)    association1 = new Java.util.HashSet();
    return association1.iterator();
}

public void setAssociation1(Java.util.Collection newAssociation1) {
    removeAllAssociation1();
    for (Java.util.Iterator iter = newAssociation1.iterator(); iter.hasNext();)
        addAssociation1((class2)iter.next());
}

public void addAssociation1(class2 newClass2) {
    if (newClass2 == null)    return;
    if (this.association1 == null) this.association1 = new Java.util.HashSet();
    if (!this.association1.contains(newClass2)) this.association1.add(newClass2);
}

public void removeAssociation1(class2 oldClass2) {
    if (oldClass2 == null)    return;
    if (this.association1 != null)
        if (this.association1.contains(oldClass2)) this.association1.remove(oldClass2);
}

public void removeAllAssociation1() {
    if (association1 != null)    association1.clear();
}
}

```

也可能是 `Java.util.Collection` 的实现类 `Java.util.List` 表示，其 Java 代码描述如下。

```

public class class1 {
    public Java.util.List association1;

    public Java.util.List getAssociation1() {
        if (association1 == null)    association1 = new Java.util.ArrayList();
        return association1;
    }
}

```

也可能是 `Java.util.Map` 的实现类 `Java.util.HashMap` 表示，其 Java 代码描述如下。

```

public class class1 {
    public Java.util.Map association1;

    public Java.util.List getAssociation1() {
        if (association1 == null)    association1 = new Java.util.HashMap ();
        return association1;
    }
}

```

③ 多对多关联

表示多个类 Class1 的对象和多个类 Class2 的对象具有关联关系。该关联所涉及的对象可以具有关联动态性。当两种对象都具有关联动态性时，这种关联关系较上述几种关系要复杂，实现起来也要困难得多。所以，在实现时，应尽量限制关联动态性，也可以在一定条件下将其转换为多个一对多或多对一关联，如图 2-10 所示。



图 2-10 类之间的多对多关联关系

这种情况发生得比较少。

(3) 关联的导向分类

关联的导向有双向关联、单向关联和自身关联（反身关联）三种。

① 双向关联

Class1- Class2: 指双方都知道对方的存在，都可以调用对方的公共属性和方法。

双向关联在代码的表现双方都拥有对方的一个指针，当然也可以是引用或者是值，如图 2-11 所示。



图 2-11 类之间的双向关联关系

② 单向关联

单向关联: Class1→Class2: 表示相识关系，指 Class1 知道 Class2，Class1 可以调用 Class2 的公共属性和方法。它没有生命期的依赖，一般表示为一种引用，如图 2-12 所示。

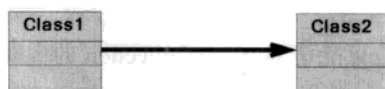


图 2-12 类之间的单向关联关系

单向关联的代码就表现为 Class1 有 Class2 的指针，而 Class2 对 Class1 一无所知。

③ 单向多关联

单向多关联: Class1→Class2: 首先是一种单向关联，但是由于 Class1 多次引用 Class2，而且每次引用都是一个独立的处理，这样就形成了多关联，如图 2-13 所示。

图 2-13 中单向多关联表现为 Class1 有三次引用 Class2 的指针。

④ 自身关联（反身关联）

自身关联（反身关联）: 自己引用自己，带着一个自己的引用。就是在自己的内部有着一个自身的引用，如图 2-14 所示。

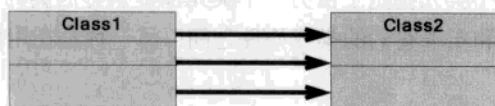


图 2-13 类之间的单向关联关系

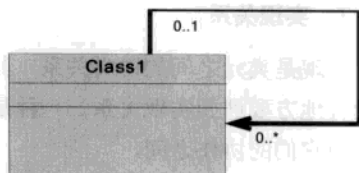


图 2-14 类之间的自身关联关系

自身关联 Java 代码实现如下：

```
public class class1 {
    public class1[] association1B;
}
```

5. 聚合 (Aggregation) 关系

聚合关系是关联关系的一种，是强的关联关系。聚合关系是整体和个体的关系。关联关系的两个类处于同一层次上，而聚合关系两个类处于不同的层次上，一个是整体，一个是部分。

(1) 聚合 (Aggregation) 关系图示

类之间的聚合关系使用带实心箭头的虚线表示。表示方法：空心菱形+实线+箭头，箭头指向部分，如图 2-15 所示。



图 2-15 类之间的聚合关系

由图 2-15 可见，Class2 实例的对象包含了多个 Class1 实例的对象。

(2) 聚合 (Aggregation) 关系代码

聚合关系的代码实现与一对多的关联关系 Java 代码实现是相同的。

(3) 聚合 (Aggregation) 关系扩展

聚合是一种特殊的关联，聚合更明确指出聚合的主体具有整体-部分关系。组合是一种特殊的聚合，组合中的某个主体控制着另外一个主体的生命周期，而且它们还存在整体-部分关系。当类之间有整体-部分关系的时候，我们就可以使用组合或者聚合。

6. 组合 (Composition) 关系

组合关系，也叫合成关系，是关联关系的一种，是比聚合关系强的关系。它要求普通的聚合关系中代表整体的对象负责代表部分的对象的生命周期，组合关系不能共享。

(1) 组合关系图示

类之间的组合关系使用带实心箭头的虚线表示。表示方法：实心菱形+实线+箭头，如图 2-16 所示。

图 2-16 表示 Class2 实例的对象包含了多个 Class1 实例的对象。

(2) 组合关系代码

组合关系的实现与一对多的关联关系 Java 代码实现是相同的。

7. 实现关系

实现是类元之间的语义关系，其中的一个类元指定了由另一个类元保证执行的契约。在两种地方要遇到实现关系。一种是在接口和实现它们的类或构件之间；另一种是在用例和实现它们的协作之间。

实现关系是用来规定接口和实线接口的类或者构建结构的关系，接口是操作的集合，而这些操作就用于规定类或者构建的一种服务。

(1) 实现关系图示

使用带空心箭头的虚线表示。如图 2-17 所示。

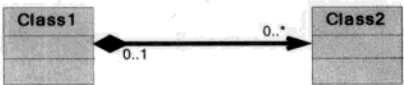


图 2-16 类之间的组合关系



图 2-17 接口和类之间的实现关系

图 2-17 说明 Class1 实现接口 Interface1。

(2) 实现关系代码

实现关系的代码实现与实现关系代码实现是相同的。

8. 几种关系的内容联系

(1) 依赖和聚合/组合、关联等的区别

关联是类之间的一种关系，例如老师教学生、领导和下属、书籍与书橱等就是一种关系。这种关系是非常明显的，在问题领域中通过分析直接就能得出。

依赖是一种弱关联，只要一个类用到另一个类，但是和另一个类的关系不是太明显的时候（可以说是“uses”了那个类），就可以把这种关系看成是依赖，依赖也可说是一种偶然的关系，而不是必然的关系。

组合是一种整体-部分的关系，在问题域中这种关系很明显，直接分析就可以得出。例如轮胎是车的一部分、树叶是树的一部分、手脚是身体的一部分这种的关系，非常明显的整体-部分关系。

(2) 泛化、关联和依赖

一般关系表现为继承或实现关系（belong to），关联关系表现为变量（is a 或 has a），依赖关系表现为函数中的参数（use a）。

2.2 Java 基础知识

Java 语言是美国 Sun 公司于 1995 年正式推出的纯面向对象的、多线程的、与平台无关的程序设计语言。它的诸多优点越来越受到程序开发人员的钟爱。现在已经发展到了 1.6

版本。本节主要介绍与 iBATIS 源码所要涉及的 Java 内容,包括 Java 的 I/O 操作,XML 操作、线程管理、反射机制、动态代理、JDBC、JavaBean 和 JNDI 等。

2.2.1 Java 的 I/O 操作

Java 的 I/O 操作是一个庞大的软件包,包含了大量的接口、类和方法。这些接口、类和方法都定义在 JDK 的 java.io 包内。由于 Java 的 I/O 过于庞大,所以,Java 定义了两个 I/O 系统,分别是字节 I/O 和字符 I/O。关于字节和字符的区别,主要是字符集的不同。Java 中支持 Unicode 字符集和 ASCII 字符集。Unicode 字符集用 16 位来表示一个字符,ASCII 字符集用 8 位来表示一个字符。Unicode 字符集可表示的符号显然要比 ASCII 字符集多得多,基本上可以表示世界上大多数语言的符号。正是针对这两种字符集,Java 有两种 I/O 实现模型:字节 I/O 实现 ASCII 字符集,而字符 I/O 实现 Unicode 字符集。

字节流 I/O 类和字符流 I/O 类的命名有其对应关系。字节输入流类的名字以“InputStream”结尾,而字符输入流类的名字以“Reader”结尾。字节输出流类的名字后缀为“OutputStream”,而字符输出流类的名字后缀为“Writer”。为了在适当的时候能把这两种流类联系起来,Java I/O API 中设置了两个类,充当两者的桥梁。InputStreamReader 根据特定的编码规则从字节流创建相应的字符流,而 OutputStreamWriter 则根据编码规则从字符流读取字符,把它们转化为字节,写入字节流中。

在 iBATIS 中读取配置文件,有时到本地目录去找寻 class 路径。这些都是属于 I/O 操作。这些问题在一般编程中都还算是比较简单的,但是在应用服务器环境中,尤其是既不知道是什么操作系统,也不知道采用了什么编码。这都需要采用一种通用的处理过程,所以 iBATIS 读取配置文件要同时支持字节 I/O 和字符 I/O。

2.2.2 Java 解析 XML 文档

用 Java 解析 XML 文档,XML 在不同的语言里解析方式都是一样的,只不过实现的语法不同而已。最常用的有两种方法:使用基于事件的 XML 简单 API (Simple API for XML) 称为 SAX 和基于树与节点的文档对象模型 (Document Object Model) 称为 DOM。SAX 是基于事件流的解析,DOM 是基于 XML 文档树结构的解析。Sun 公司提供了 Java API for XML Parsing (JAXP) 来同时支持 SAX 和 DOM,通过 JAXP,我们可以使用任何与 JAXP 兼容的 XML 解析器。

1. DOM 生成和解析 XML 文档

什么是 DOM? DOM 是 W3C (万维网联盟) 的推荐标准。DOM 定义了访问诸如 XML 和 XHTML 文档的标准。“W3C 文档对象模型 (DOM) 是一个使程序和脚本有能力动态地访问和更新文档的内容、结构,以及样式的平台和语言中立的接口。”

W3C DOM 被分为三个不同的部分/级别 (parts / levels): ① 核心 DOM; 用于任何结

构化文档的标准模型；② XML DOM：用于 XML 文档的标准模型；③ HTML DOM：用于 HTML 文档的标准模型。

DOM 定义了所有文档元素的对象和属性，以及访问它们的方法（接口）。XML DOM 定义了访问和处理 XML 文档的标准方法。XML DOM 是 XML Document Object Model 的缩写，即 XML 文档对象模型。DOM 为 XML 文档的已解析版本定义了一组接口。解析器读入整个文档，接着构建一个驻留内存的树结构，然后代码就可以使用 DOM 接口来操作这个树结构。优点是整个文档树在内存中，便于操作；支持删除、修改、重新排列等多种功能；缺点是将整个文档调入内存（包括无用的节点），浪费时间和空间，所以 DOM 的使用场合一般是一旦解析了文档还需多次访问这些数据，其次还要硬件资源充足（内存、CPU）。

org.w3c.dom 软件包为文档对象模型（DOM）提供接口，该模型是 Java API for XML Processing 的组件 API。该 Document Object Model Level 2 Core API 允许程序动态访问和更新文档的内容和结构。

2. SAX 生成和解析 XML 文档

为解决 DOM 在解析 XML 中的性能问题，出现了 SAX。SAX 基于事件驱动，当解析器发现元素开始、元素结束、文本、文档的开始或结束等时，发送事件，开发人员编写响应这些事件的代码，保存数据。SAX 的优点是：不用事先调入整个文档，占用资源少；SAX 解析器代码比 DOM 解析器代码小，适于下载。缺点：① 非持久性；事件过后，若没保存数据，那么数据就丢了；② 无状态性；从事件中只能得到文本，但不知该文本属于哪个元素。所以 DOM 的使用场合一般是 Applet，或者只需 XML 文档的少量内容、很少回头访问，再者就是电脑内存少等场所。

org.XML.sax 软件包提供了核心 SAX API。

3. Java API for XML Processing (JAXP)

Java API for XML Processing (JAXP) 允许使用几种不同的 API 来验证、解析和转换 XML。JAXP 既提供了使用方便性，又提供了开发商中立性。

在 Java 平台的早期版本中，JAXP 是核心平台中单独的下载。在 Java 5.0 中，JAXP 已经是 Java 语言的主要产品。有了 JAXP，您可以用 Sun 的 XML 解析器、Apache 的 Xerces XML 解析器和 Oracle 的 XML 解析器来处理相同的代码。因为使用特定于 Sun 的类会违反使用 JAXP 的要点。

JAXP 是 API，更准确地说是抽象层。它没有提供解析 XML 的新方法，没有添加到 SAX 或 DOM，也没有为 Java 和 XML 处理提供新功能。但是，JAXP 使得使用 DOM 和 SAX 来处理一些困难任务变得更容易，它还允许以开发商以中立方式处理一些在使用 DOM 和 SAX API 时可能遇到的特定开发商的任务。

JAXP 接口包含了三个包：

- ① org.w3c.dom: W3C 推荐的用于 XML 标准规划文档对象模型的接口；

- ② org.XML.sax: 用于对 XML 进行语法分析的事件驱动的 XML 简单 API (SAX);
 - ③ Javax.XML.parsers: 解析器工厂工具, 开发人员获得并配置特殊的特殊语法分析器。
- JAXP (Java API for XML Parsing) 的最新规范可以到 Sun 公司的官方网站 <http://java.sun.com/javase/6/docs/technotes/guides/xml/jaxp/index.html> 去下载。

2.2.3 Java 的线程管理

多线程程序设计是 Java 中比较精彩的部分之一。Java 对多线程的支持是语言级的, 即能够使用该语言直接编写同时执行多个任务的应用程序。这使得应用程序开发人员只需要关注应用程序本身, 而不用了解后台操作系统的线程机制。这也是为了最大限度地提高可移植性。

Java 的多线程系统建立在 Thread 类和它相应接口 Runnable 基础之上, Thread 封装了一个执行线程。为了调用多线程程序, 开发人员必须扩展 Thread 类或实现 Runnable 接口。扩展 Thread 类和 Runnable 接口的区别是扩展 Thread 类必须重载 Thread 类的 run 方法。

在 iBATIS 框架多次使用多线程设计来完成操作, 主要涉及的线程内容包括 Synchronized、线程之间的通信和 ThreadLocal。

1. 关键字 Synchronized

使用关键字 Synchronized 就可以解决多线程共享数据同步问题。关键字 Synchronized 用于保护共享数据。每个对象都有一个锁标志, 当一个线程访问该对象时, 被 Synchronized 修饰的数据将被“上锁”, 阻止其他线程访问。当前线程访问完这部分数据后释放锁标志, 其他线程就可以访问了。

Synchronized 关键字可以作为函数的修饰符, 也可作为函数内的语句, 也就是平时说的同步方法和同步语句块。同时, Synchronized 关键字还可以直接作用于 Object 对象、instance 变量、Object Reference (对象引用)、static 函数和 class literals 等。这里有几个基本说明: 无论 Synchronized 关键字加在方法上还是对象上, 它取得的锁都是对象, 而不是把一段代码或函数当作锁——而且同步方法很可能还会被其他线程的对象访问。当 Synchronized 关键字加在方法上时, 那么这个方法归属的对象就是要进行加锁的对象。

每个对象只有一个锁 (lock) 与之相关联。实现同步是要以很大的系统开销作为代价的, 甚至可能造成死锁, 所以要尽量避免无谓的同步控制。

2. 线程之间的通信

线程之间的通信主要是 wait、notify、notifyAll 方法, 这是任何一个 java 对象都具有的方法。这三个方法用于协调多个线程对共享数据的存取, 所以必须在 Synchronized 语句块内使用这三个方法。前面说过 Synchronized 这个关键字用于保护共享数据, 阻止其他线程对共享数据的存取。但是这样程序的流程就很不灵活了, 如何才能在当前线程还没退出 Synchronized 数据块时让其他线程也有机会访问共享数据呢? 就用这三个方法来灵活控制。

`wait` 方法使当前线程暂停执行并释放对象锁标志, 让其他线程可以进入 `Synchronized` 数据块, 当前线程被放入对象等待池中。当调用 `notify` 方法后, 将从对象的等待池中移走一个任意的线程并放到锁标志等待池中, 只有锁标志等待池中的线程能够获取锁标志; 如果锁标志等待池中没有任何线程, 则方法 `notify` 不起作用。方法 `notifyAll` 则从对象等待池中移走所有等待那个对象的线程并放到锁标志等待池中。

在 iBATIS 中处理多线程同步时候多次用到了 `Synchronized` 和线程之间的通信。

3. ThreadLocal 的说明

早在 JDK 1.2 的版本中就提供 `java.lang.ThreadLocal`, `ThreadLocal` 为解决多线程程序的并发问题提供了一种新的思路。使用这个工具类可以很简洁地编写出优美的多线程程序。

`ThreadLocal` 不是本地的 `Thread`, 而是 `Thread` 的局部变量, 该类提供了线程局部变量。这些变量不同于它们的普通对应物, 因为访问一个变量 (通过其 `get` 或 `set` 方法) 的每个线程都有自己的局部变量, 它独立于变量的初始化副本。`ThreadLocal` 实例通常是类中的私有静态字段, 它们希望将状态与某一个线程 (例如, 用户 ID 或事务 ID) 相关联。

当使用 `ThreadLocal` 维护变量时, `ThreadLocal` 为每个使用该变量的线程提供独立的变量副本, 所以每一个线程都可以独立地改变自己的副本, 而不会影响到其他线程所对应的副本。

从线程的角度看, 目标变量就像是线程的本地变量, 这也是类名中 “Local” 所要表达的意思。

`ThreadLocal` 类接口很简单, 只有 4 个方法, 我们先来了解一下:

- ① `public void set (Object value)` 方法设置当前线程的线程局部变量的值;
- ② `public Object get()` 方法返回当前线程所对应的线程局部变量;
- ③ `public void remove()` 方法将当前线程局部变量的值删除, 目的是为了减少内存的占用, 该方法是 JDK 5.0 新增的方法。需要指出的是, 当线程结束后, 对应该线程的局部变量将自动被垃圾回收, 所以显式调用该方法清除线程的局部变量并不是必需的操作, 但可以加快内存回收的速度;
- ④ `protected Object initialValue()` 方法返回该线程局部变量的初始值, 该方法是一个 `protected` 的方法, 显然是为了让子类覆盖而设计的。这个方法是一个延迟调用方法, 在线程第 1 次调用 `get()` 或 `set (Object)` 时才执行, 并且仅执行 1 次。`ThreadLocal` 中的默认实现直接返回一个 `null`。

在 JDK5.0 中, `ThreadLocal` 已经支持泛型, 该类的类名已经变为 `ThreadLocal<T>`。API 方法也相应进行了调整, 新版本的 API 方法分别是 `void set(T value)`、`T get()` 以及 `T initialValue()`。

`ThreadLocal` 是如何做到为每一个线程维护变量的副本的呢? 其实实现的思路很简单: 在 `ThreadLocal` 类中有一个 `Map`, 用于存储每一个线程的变量副本, `Map` 中元素的键为线程对象, 而值对应线程的变量副本。

在 iBATIS 的 `session` 中使用了 `ThreadLocal`。

2.2.4 Java 的反射机制

反射 (Reflection) 的概念是由 Smith 在 1982 年首次提出的, 主要是指程序可以访问、检测和修改它本身状态或行为的一种能力。这一概念的提出很快引发了计算机科学领域关于应用反射性的研究。它首先被程序语言的设计领域所采用, 并在 Lisp 和面向对象方面取得了成绩。Java 就是基于反射机制的语言。

Java 反射机制由元层体系结构的两部分内容来共同作用: 元层 (Meta Level) 和基层 (Base Level)。元层由元对象组成, 用于描述类本身而不是它们的用法。元层对象由元对象协议 MOP (Meta Object Protocol) 来规约; 基层用于定义应用逻辑, 使用元对象来实现。Java 并不属于动态语言, 但 Java 的反射机制在一定程度上实现了动态功能。

Java 反射是 Java 语言的一个很重要的特征, 它使得 Java 具体化了“动态性”。Reflection 是 Java 被视为动态 (或准动态) 语言的一个关键性质。这个机制允许程序在运行时透过 Reflection APIs 取得任何一个已知名称的 class 的内部信息, 包括其 modifiers (诸如 Public、Static 等)、superclass (例如 Object)、实现之 interfaces, 也包括 fields 和 methods 的所有信息, 并可于运行时改变 fields 内容或调用 methods。简单地说, 就是对于任意一个类在运行状态中, 我们都能够知道这个类的所有属性和方法; 对于任意一个对象, 都能够调用它的任意一个方法; 这种动态获取的信息以及动态调用对象的方法的功能称为 Java 语言的反射机制。

Java 反射机制主要提供了以下功能。

- 在运行时判断任意一个对象所属的类。
- 在运行时构造任意一个类的对象。
- 在运行时判断任意一个类所具有的成员变量和方法。
- 在运行时调用任意一个对象的方法。

在 JDK 中, 主要由 Class、Field、Method、Constructor、Array 等类来实现 Java 反射机制, 这些类都包含在 Java.lang.reflect 包中。

- Class 类: 代表一个类。
- Field 类: 代表类的成员变量 (成员变量也称为类的属性)。
- Method 类: 代表类的方法。
- Constructor 类: 代表类的构造方法。
- Array 类: 提供了动态创建数组, 以及访问数组的元素的静态方法。

Java 语言反射提供一种动态链接程序组件的多功能方法。它允许程序创建和控制任何类的对象 (根据安全性限制), 无须提前硬编码目标类。这些特性使得反射特别适用于创建以非常普通的方式与对象协作的库。Java 的这一特性非常强大, 并且是其他一些常用语言, 如 C、C++、Fonran 或者 Pascal 等都不具备的。

iBATIS 在处理 bean 的时候, 会多次应用到 Java 的反射机制。尤其是针对从数据库的

一条结果集 `ResultSet` 中的每条记录分别提取出表列值并赋值给域对象相应的字段，而对不同的域对象由于字段不同而造成具体的读写方法（`read write method`）也不同，为此在 `iBATIS` 设计加载一条结果集时需要考虑动态调用不同对象具体字段的读写方法。

2.2.5 Java 的动态 Proxy

代理模式就是给某一对象提供一个代理，由该代理对象控制对被代理对象的访问。它定义了代理对象、被代理对象和它们公共的接口三个角色。动态代理是一种特殊的代理模式，其代理对象是在程序运行期间动态创建的。它的主要特点是不需要定义代理对象、程序代码高度抽象以及对象之间耦合弱等。

动态 Proxy 是 Java 中比较精彩的部分之一。自从 `JDK1.3` 以来，Java 语言通过在 `java.lang.reflect` 包中提供下面三个类直接支持代理模式：Proxy 类、InvocationHandler 接口和 Method 接口。其中 Proxy 类使得设计师能够在运行时间创建代理，当系统有了一个代理对象后，对原对象的方法调用会首先被分配给一个调用处理器（Invocation Handler）。调用处理器需要实现 InvocationHandler 接口，该接口就包含一个 `invoke` 方法。在反身映射的基础上，程序可以在调用处理器的 `yoke()` 方法中截获这个调用，进行额外的操作。

`java.lang.reflect` 包中的 Proxy 类和 InvocationHandler 接口提供了对动态代理的支持。其中 Proxy 类是代理类，提供用于创建动态代理类和实例的静态方法，它还是由这些方法创建的所有动态代理类的超类。它定义的主要方法如下：

```
Public Static Object newProxyInstance(ClassLoader loader , Class[] classArray ,
InvocationHandler handler )throws IllegalArgumentException
```

InvocationHandler 接口是调用处理器接口，它声明的方法：

```
Public Object invoke(Object Object, Method method, Object Objects ) throws Throwable
```

动态代理的实现机制：由 Proxy 类的静态方法 `newProxyInstance()` 在程序运行期间根据参变量 `loader`（类加载器）、`classArray`（代理接口数组）和 `handler`（实现了 InvocationHandler 接口的调用处理器）来创建一个代理对象，而对该代理对象的方法调用都会被 JVM 截获并将该代理对象 `Object`、方法 `method` 和方法参数数组 `Objects` 传递给调用处理器 `handler` 的 `invoke()` 方法中相应参变量，程序流程转入 `invoke()` 方法。

Java 动态代理可以看作是一个在运行期实现一组指定接口的特殊类，它具有三个特点：

- ① 运行期动态生成；
- ② 实现一组指定接口，可以直接被造型成指定的对象；
- ③ 对动态代理所暴露接口的方法调用将被转换为一种统一的调用方式，然后回调到它的调用处理类 InvocationHandler 上，由 InvocationHandler 完成具体的处理。

创建动态代理的具体步骤和要求如下：

- ① 定义若干代理接口，这些代理接口中不能有冲突的方法名；

- ② 定义实现对应代理接口的若干被代理类;
- ③ 定义调用处理器类, 在 `invoke()` 方法中利用 Java 反射机制实现对被代理对象的方法调用;
- ④ 编写创建代理对象、调用对象方法等程序代码。

iBATIS 在架构 DAO 框架中和 iBATIS SQL Map 的延迟加载多次应用到 Java 的动态代理机制。

2.2.6 JDBC 和 JDBC 扩展

JDBC (Java Base Connectivity) 是 Java 的开发者 Sun 公司制定的 Java 数据库连接技术的简称, 是为各种常用数据库提供无缝链接的技术。这是 Java 语言为了支持 SQL 功能而提供的与数据库相连的用户接口, JDBC 中包括了一组由 Java 语言书写的接口和类, 它们都是独立于特定的 DBMS, 或者说它们可以和各种数据相关联。有了 JDBC 以后, 开发人员方便地在 Java 语言中使用 SQL 语言, 从而实现 Java 应用程序可以对分布在网络上的各种关系数据库的访问。使用了 JDBC 以后, 开发人员可以将精力集中于上层的功能实现, 而不必关心底层与具体的 DBMS 的连接和访问过程。

JDBC API 是一个标准统一的 SQL 数据存取接口, 由一组 Java 语言编写的类和接口组成, 使用内嵌式的 SQL。JDBC 的作用与 ODBC 在 Windows 系列中的作用类似。ODBC (Open Data Base Connectivity), 称为开放式数据库互联技术, 是由微软公司倡导并得到业界普遍响应的一门数据库连接技术。开发人员在使用 JDBC 时可以不关心它所要操作的数据库是哪个厂家的产品, 从而提高了软件的通用性, 只要系统上安装了正确的驱动器组, JDBC 的应用程序就可以访问其相关的数据库, 主要实现三方面的功能: 建立与数据库的连接、执行 SQL 声明, 以及处理 SQL 执行结果。JDBC 支持基本的 SQL 功能, 使用它可方便地与不同的关系型数据库建立连接, 进行相关操作, 并无须再为不同的 DBMS 分别编写程序。JDBC 现在可以连接的数据库几乎覆盖了所有的关系数据库。

JDBC 的工作机制: JDBC 定义了 Java 语言同 SQL 数据之间的程序设计接口。JDBC 有一个非常独特的动态连接结构, 它使得系统模块化。使用 JDBC 来完成对数据库的访问包括以下四个主要组件: Java 的应用程序、JDBC 驱动器管理器、驱动器和数据源。

JDBC 的实现过程: 使用 JDBC API 处理关系数据库的第一步是安装驱动程序, 然后利用 `DriverManager` 类的 `getConnection()` 方法创建一个 `Connection` 对象来建立与关系数据库的连接, 连接一旦建立, 就可用来向数据库传送 SQL 语句, 而 `Statement` 对象用于将 SQL 语句发送到数据库中, `Statement` 对象可用 `Connection` 的 `createStatement()` 方法产生, SQL 语句的执行结果可由 `ResultSet` 对象获得, 这个对象通过一套 `get` 方法提供了对结果记录的访问。

JDBC 包含几个核心对象, JDBC 中最重要的部分是定义了一系列的抽象接口, 通过这些接口, JDBC 实现了三个基本的功能: 建立与数据的连接、执行 SQL 声明和处理执行结果。主要的接口和功能实现关系如表 2-1 所示。

表 2-1 JDBC 的核心对象

序号	核心对象	功能说明
1	JDBC 的 DriverManager	java.sql.DriverMagnager: 管理驱动器, 支持驱动器与数据连接的创建
2.	JDBC 的 Connection 接口	通过 Connection 对象建立数据库连接。代表与某一数据库的连接, 支持 SQL 声明的创建
3.	JDBC 的 DatabaseMetaData 接口	通过 DatabaseMetaData 对象了解数据库的信息
4.	JDBC 的 Statement 接口	java.sql.Statement: 在连接中执行一静态的 SQL 声明并取得执行结果
5.	JDBC 的结果集 ResultSet 接口	java.sql.ResultSet: 代表执行 SQL 声明后产生的数据结果
6.	JDBC 的 PreparedStatement 接口	java.sql.PreparedStatement: Statement 的子类, 代表预编译的 SQL 声明
7.	JDBC 的 CallableStatement 接口	java.sql.CallableStatement: Statement 的子类, 代表 SQL 的存储过程

JDBC 的优点如下:

- ① JDBC API 与 ODBC 十分相似, 有利于用户理解;
- ② JDBC 使得编程人员从复杂的驱动器调用命令和函数中解脱出来, 可以致力于应用程序中的关键地方;
- ③ JDBC 支持不同的关系数据库, 使得程序的可移植性大大加强;
- ④ 用户可以使用 JDBC-ODBC 桥驱动器将 JDBC 函数调用转换为 ODBC;
- ⑤ JDBC API 是面向对象的, 可以让用户把常用的方法封装为一个类, 便于进行高层的耦合。

JDBC 的缺点如下:

- ① 使用 JDBC, 访问数据记录的速度会受到一定程度的影响;
- ② JDBC 结构中包含了不同厂家的产品, 这就给更改数据源带来了很大的麻烦。

JDBC (Java Data Base Connectivity) 的最新规范可以到 Sun 公司的官方网站 <http://java.sun.com/products/jdbc/download.html> 去下载。

2.2.7 JavaBean

JavaBean 是 Java 技术的一个发展。Java 语言的设计者们于 1996 年底推出了 JavaBean 应用程序接口的正式说明书。引入 JavaBean 的目的是为 Java 提供一个新的软件组件 (Component) 模型, 以便使用 Java 开发的软件可以很方便地被重复使用, 或组装起来以适应新的需要。简单地说, 意图是让开发者开发出来的程序 (JavaBean) 相对独立, 具有统一的接口, 如同积木一般可以装配。尽管这一概念尚不完善, 但已引起许多人士的注意。“JavaBean”的发展将为软件复用提供一种新的解决方案。所以说, JavaBean 是用 Java 语言描述的软件组件 (Component) 模型, 实际上是一个类, 并且是一种特殊的类。按照 JavaBean 规范接口说明, JavaBean 是一种可重用的软件组件, 它可以在创建工具中被可视化地操纵。

JavaBean 具有如下的典型特征, 正是这些特征保证了它的功能:

- ① “自省”功能 (Introspection), 即提供相应的机制, 使创建工具能够分析它的行为

和性质:

- ② 可以定制 (Customize)。即某些属性 (Properties) 可以通过创建工具来修改;
- ③ 可以被组合。这是通过事件机制来完成的;
- ④ 可以存储、装入。这使得对 Bean 编辑和连接的结果可以被录入保存,并在适当的时候取出来应用。

JavaBean 的三大要素是属性 (Properties)、允许其他组件调用的方法集和能够激发的事件集。

属性值可以通过调用相应的方法来修改或获取。如用 `get` 获取值, `set` 设置值等。

`public` 方法一般来说都可以被其他组件调用,但 JavaBean 可以限定只有部分 `public` 方法可以被调用。

事件是组件之间联系的手段, JDK1.1 版本和 BDK 中引入了一种有别于过去的事件处理的模式,即代理 (Delegation) 模式。在这种模式中,存在事件源与事件监听者。两者通过“登记”联系起来,一旦事件源处发生了相应的事件,对应的监听者的方法就被调用来处理它。

JavaBean 严格遵循 JavaBean 命名规范,定义存取类状态信息方法的命名规则。JavaBean 的属性由它的方法定义 (而不是由字段定义)。以“`set`”为名称开始的方法是可写的属性,而以“`get`”为名称开始的方法是可读的属性。对于“`boolean`”类型的字段,可读的方法名称也可以用“`is`”开始。“`set`”方法不应拥有返回类型 (即必须为 `void`),并且只能有一个参数,参数的数据类型必须和属性的数据类型一致。“`get`”方法应返回合适的类型并且不允许有参数。虽然通常并不强制,但“`set`”方法参数的数据类型和“`get`”方法的返回类型应一致。JavaBean 还应实现 `Serializable` 接口。Java Bean 还支持其他特性 (例如事件接口等)。

特点:易于维护、使用、编写;可实现代码的重用性;可移植性强、但仅限于 Java 工作平台;便于传输,不限于本地环境还是网络环境;可以以其他部件的模式进行工作。

iBATIS 作为一个 ORM 框架模型,毫不犹豫地要使用和支持 JavaBean。

JavaBean (JavaBeans Component API) 的最新规范可以到 Sun 公司的官方网站 <http://java.sun.com/javase/6/docs/technotes/guides/beans/index.html> 去下载。

2.2.8 JNDI

JNDI (Java Naming and Directory Interface) 是 Java SE 规范中定义的一组标准 API,通过提供一组接口、类和关于命名空间的概念,为 Java SE 应用部署的资源提供命名与目录服务的统一访问接口,以简化应用程序对资源的引用方法。

JNDI 是 provider-based 的技术,暴露了一个 API 和一个服务供应接口 (SPI)。这意味着:只要 JNDI 支持这项技术,任何基于名字的技术都能通过 JNDI 而提供服务。JNDI 目前所支持的技术包括 LDAP、CORBA Common Object Service (COS) 名字服务、RMI、

NDS、DNS、Windows 注册表等。很多 Java EE/J2EE 技术，包括 EJB 都依靠 JNDI 来组织和定位实体。

JNDI 通过绑定的概念将对象和名称联系起来。在一个文件系统中，文件名被绑定给文件。在 DNS 中，一个 IP 地址绑定一个 URL。在目录服务中，一个对象名被绑定给一个对象实体。

JNDI 中的一组绑定作为上下文来引用。每个上下文暴露的一组操作是一致的。例如，每个上下文提供了一个查找操作，返回指定名字的相应对象，每个上下文都提供了绑定和撤除绑定名字到某个对象的操作。JNDI 使用通用的方式来暴露命名空间，即使用分层上下文以及使用相同命名语法的子上下文。

JNDI 分为命名服务和目录服务两部分。命名服务将名称和对象联系起来，使我们可以用名称访问对象。而命名服务中的对象是可以多样化的，它可以是域名系统中的名称、应用服务器中的组件或某些必要的环境变量等。目录服务是命名服务的自然扩展，两者之间的关键差别是目录服务中的对象可以有属性，而命名服务中的对象没有属性。JNDI 的结构如图 2-18 所示。

JNDI 从结构上提供了标准的独立于命名系统的 API，这些 API 构建在与命名系统有关的驱动之上。这一层有助于将应用与实际数据源分离，而不管应用访问的是 LDAP (Lightweight Directory Access Protocol)、RMI (Remote Method Invoke)、DNS，还是其他的目录服务。换句话说，JNDI 是独立于目录服务的具体体现，只要存在目录的服务提供接口或驱动，就可以使用目录。

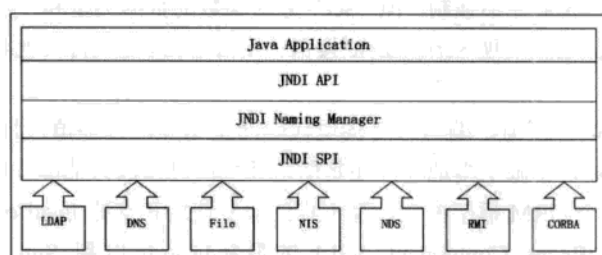


图 2-18 JNDI 的结构

JNDI 由两组 API 组成：API (Application Programming Interface) 和 SPI (Service Provider Interface)。JNDI API 的定义不依赖于任何特定的命名和目录服务，这使得对于各种不同的命名和目录服务，都可以使用这组通用和统一的接口调用。JNDI SPI 可以使得各种不同的命名和目录服务（如 LDAP、DNS、NIS 等）透明地加入到 JNDI 体系结构中，从而使 Java 应用程序能够通过 JNDI API 访问这些服务。

实际上，Java EE 规范要求所有 Java EE 容器都要提供 JNDI 规范的实现。JNDI 在 Java EE 中的角色就是“交换机”——Java EE 组件在运行时间间接地查找其他组件、资源或服务的通用机制。在多数情况下，提供 JNDI 供应者的容器可以充当有限的数据存储，这样管

理员就可以设置应用程序的执行属性,并让其他应用程序引用这些属性。Java 管理扩展, (Java Management Extensions, JMX) 也可以用作这个目的。JNDI 在 Java EE 应用程序中的主要角色就是提供间接层,这样组件就可以发现所需要的资源,而不用了解这些间接性。在 Java EE 中, JNDI 是把 Java EE 应用程序合在一起的粘合剂, JNDI 提供的间接寻址允许跨企业交付可伸缩的、功能强大且很灵活的应用程序。

ibatis 在支持 EJB 和外部事务容器方面用到了 JNDI。

JNDI (Java Name and Directory Interface) 的最新规范可以到 Sun 公司的官方网站 <http://java.sun.com/javase/6/docs/technotes/guides/jndi/index.html> 去下载。

2.3 数据库相关基础知识

数据库相关内容覆盖面非常宽,这里只简单介绍与 ibatis 接触最多的 SQL 语句和数据库事务。

2.3.1 SQL

关于 SQL 的解释,可能有很多种说法。SQL 是用于访问和处理数据库的标准的计算机语言,SQL 指结构化查询语言,SQL 使我们有能力访问数据库,SQL 是一种 ANSI 的标准计算机语言。总之,不能一言以蔽之。

SQL 全名是结构化查询语言 (Structured Query Language),是用于数据库的标准数据查询语言,IBM 公司最早将其用在开发的数据库系统中。1986 年 10 月,美国 ANSI 对 SQL 进行了规范,以此作为关系式数据库管理系统的标准语言 (ANSI X3.135-1986),1987 年得到国际标准组织的支持,成为了国际标准。不过各种通行的数据库系统在其实过程中都对 SQL 规范做了某些编改和扩充。实际上,不同数据库系统之间的 SQL 语言是不能完全相互通用的。

SQL 是一个历史悠久,生命力强大的语言。在 20 世纪 70 年代初,由 IBM 公司加州圣荷赛研究实验室的 E.F.Codd 发表了将资料组成表格的应用原则 (Codd's Relational Algebra),这是关系数据库系统思想的雏形。1974 年,同一实验室的 D.D.Chamberlin 和 R.F. Boyce 将 Codd's Relational Algebra 用于研制关系数据库管理系统 System R,研制出了一套规范语言——SEQUEL (Structured English QUery Language),并在 1976 年 11 月的 IBM Journal of R&D 上公布新版本的 SQL 语言 (叫 SEQUEL/2)。1980 年改名为 SQL。1979 年 ORACLE 公司首先提供商用的 SQL,IBM 公司在 DB2 和 SQL/DS 数据库系统中也实现了 SQL。1986 年 10 月,美国 ANSI 采用 SQL 作为关系数据库管理系统的标准语言 (ANSI X3.135-1986),后为国际标准化组织 (ISO) 采纳为国际标准。

SQL 是高级的非过程化编程语言,它允许用户在高层数据结构上工作。它不要求用户

指定对数据的存放方法，也不需要用户了解其具体的数据存放方式。而它的界面，能使具有底层结构完全不同的数据库系统和不同数据库之间，使用相同的 SQL 语言作为数据的输入与管理。它以记录项目 (records) 的合集 (set) (项集, record set) 作为操纵对象，所有 SQL 语句接受项集作为输入，回送出的项集作为输出，这种项集特性允许一条 SQL 语句的输出作为另一条 SQL 语句的输入，所以 SQL 语句可以嵌套，这使它拥有极大的灵活性和强大的功能。在多数情况下，在其他编程语言中需要用一大段程序才可实践的一个单独事件，而其在 SQL 上只需要一个语句就可以被表达出来。这也意味着用 SQL 语言可以写出非常复杂的语句。

现在有两种扩展性非常好的 SQL，分别是 Transact-SQL 和 PL-SQL。Transact-SQL 是微软 MS SQL-Server，以及 Sybase Adaptive Server 系列数据库所用的 SQL 语言。PL-SQL 是 Oracle 数据库所使用的 SQL 语言。

iBATIS 的 SQL Map 就是以 SQL 为基础，但由于采用了通用实现模式，故 iBATIS 的 SQL Map 并没有针对性地实现 Transact-SQL 和 PL-SQL。

2.3.2 数据库事务管理

与数据库打交道，不可避免的就是事务管理。关于数据库事务的概念，已经超过了本书的讨论范围。这里只是简单地说明一下事务的基本概念和内容。

事务是作为单个逻辑工作单元执行的一系列操作。一个逻辑工作单元必须有四个属性，称为原子性、一致性、隔离性和持久性 (ACID) 属性，只有这样才能成为一个事务，具体说明如下：

(1) 原子性：事务必须是原子工作单元；对于其数据修改，要么全都执行，要么全都不执行；

(2) 一致性：事务在完成时，必须使所有的数据都保持一致状态。在相关数据库中，所有规则都必须应用于事务的修改，以保持所有数据的完整性。事务结束时，所有的内部数据结构（如 B 树索引或双向链表）都必须是正确的；

(3) 隔离性：由并发事务所做的修改必须与任何其他并发事务所做的修改隔离。事务识别数据时数据所处的状态，要么是另一并发事务修改它之前的状态，要么是第二个事务修改它之后的状态，事务不会识别中间状态的数据。这称为可串行性，因为它能够重新装载起始数据，并且重播一系列事务，以使数据结束时的状态与原始事务执行的状态相同；

(4) 持久性：事务完成之后，它对于系统的影响是永久性的。该修改即使出现系统故障也将一直保持。

对于数据库事务而言，系统中所有任务的事务状态分为三种：开始提交、正在提交和事务空闲（没有开始事务）。这要求事务处理包括开始事务 (Start)，提交事务 (Commit) 和回滚事务 (Rollback) 和事务恢复。

数据库系统要涉及并发控制，这主要表现为一个事务对数据库系统中数据修改，然后又有一个事务对同一数据库系统的数据进行修改，形成冲突。先有事务读操作，然后有另一事务进行写操作，形成冲突。解决冲突的方式是将事务可串行化调度。例如一个调度的动作组成一个事务的所有动作，而没有动作的混合。通常，如果不管数据库初态怎样，一个调度对数据库状态的影响都和某个串行调度相同，我们就说这个调度是可串行化的。可串行性的实现要有几种机制来保障、有锁机制、时间戳机制、有效性确认机制等。

(1) 锁机制：事务获得在它所访问的数据库元素上的锁，以防止其他事务几乎在同一时间访问这些元素并因而引起的非可串行性的可能。

(2) 时间戳：我们给每个事务分配一个“时间戳”，记录最后读写每个数据库元素的事务的时间戳，并比较这些值以保证事务的时间戳排序的串行调度等价于事务的实际调度。

(3) 有效性确认：当事务将要提交时，我们检查事务的时间戳和数据库元素，这个过程称为事务的“有效性确认”。按照有效性确认时间对事务进行排序的串行调度必须等价于实际的调度。

在分布式计算环境中，一个事务可以访问多个不同的独立资源，它们可能分布在不同的应用中，或者不同的进程中，甚至不同的计算机上，称这种事务为分布式事务。分布式事务所涉及的整个系统也具有 ACID 的特性。分布式事务跨越两个或多个称为资源管理器的服务器，称为事务管理器的服务器组件必须在资源管理器之间协调事务管理。

分布式事务处理中也要涉及事务提交、事务恢复、事务日志、并发控制及死锁等方面的问题，但是分布式事务在处理这些问题的时候比较复杂，其中涉及的关键技术包括事务管理器、两阶段提交协议和实现、分布式事务恢复机制等。关于这些内容，已经超过了本书的讨论范围。有兴趣的读者，可以去找一些专业书籍来参考。

2.4 Java EE 规范相关知识

还好，在 iBATIS 中只用到了 Java EE 的一个规范，即 JTA。

JTA 介绍

JTA (Java Transaction API) 是事务管理器和分布式事务处理系统所涉及的其他组件之间的一个接口规范。JTA 也定义了一种标准的 API，通过使用接口，不必使用事务管理器的特有 API 就可以单独地划分事务，应用系统由此可以访问各种事务监控。

关于 Java EE 的事务规范，Java EE 定义了两套规范，分别是 Java Transaction API (JTA) 和 Java Transaction Service (JTS)。JTA 是用来定义事务管理器和一个分布式事务系统各组件间的接口。JTS 则是事务管理器的实现定义，定义了参与到一个分布式事务的基本 DTP 模型和所涉及的各个组件，以及各个组件的责任和功能。

JTA 具有的三个主要的接口，分别是 UserTransaction 接口、TransactionManager 接口和 Transaction 接口。这些接口共享公共的事务操作，例如 commit 和 rollback，但是也包含

特殊的事务操作，例如 `suspend`、`resume` 和 `enlist`，它们只出现在特定的接口上，以便在实现中允许一定程度的访问控制。例如，`UserTransaction` 能够执行事务划分和基本的事务操作，而 `TransactionManager` 能够执行上下文管理。应用程序可以调用 `UserTransaction.begin()` 方法开始一个事务，该事务与应用程序在其中运行的当前线程相关联。底层的事务管理器实际处理线程与事务之间的关联。`UserTransaction.commit` 方法终止与当前线程关联的事务（提交）。`UserTransaction.rollback` 方法将放弃与当前线程关联的当前事务（回滚）。

JTA 只是一组 Java 接口，用于描述 Java EE 框架中事务管理器与应用程序、资源管理器，以及应用服务器之间的事务通信。它主要包括：高层接口，即面向应用程序的接口；`XAResource` 接口，即面向资源的接口；以及事务管理器的接口。值得注意的是，JTA 只提供了接口，没有具体的实现。JTS 是服务 OTS 的 JTA 的实现。简单地说，JTS 实现了 JTA 接口，并且符合 OTS 的规范。资源管理器只要其提供给事务管理器的接口符合 XA 接口规范，就可以被事务管理器处理。所以，JTA 可以处理任何提供符合 XA 接口的资源，包括：数据库、JMS，商业对象，等等。

iBATIS 在支持事务管理时需要 JTA。

JTA（Java Transaction API）的最新规范可以到 Sun 公司的官方网站 <http://java.sun.com/javaee/technologies/jta/> 去下载。

2.5 开源 ORM 框架

由于 iBATIS DAO 的实现涉及多种开源 ORM 框架的集成，所以，在这里简单地介绍一下 iBATIS 所涉及的开源 ORM 框架。

2.5.1 Hibernate

1. Hibernate 简介

Hibernate 是一个开放源代码的 O/R Mapping（对象关系映射框架），它对 JDBC 进行了轻量级的对象封装，使 Java 开发人员可以随心所欲地使用对象编程思维来操纵数据库。Hibernate 是一个成熟的基于 Java 环境的对象-关系映射器，把数据表示由对象模型映射到关系模型、SQL-based 结构的一种技术策略。Hibernate 不仅包含 Java 类到数据表的映射，也提供数据查询和获取的便捷方法。

Hibernate 对 JDBC 做了轻量级封装，也就是说，它没有完全封装 JDBC，Java 应用程序可以通过 Hibernate API 访问数据库，还可以绕过 Hibernate API，接通过 JDBC 来访问数据库。Hibernate 的工作原理是：它通过对关系映射的描述文件在值对象和数据库表之间建立起一个映射关系，这样我们只需要通过操作这些值对象和 Hibernate 提供的一些基本类，就可以达到访问数据库的目的。

Hibernate 提供了实现持久化层的一种模式, 它采用元数据来描述对象-关系的映射文件, 在 Java 应用的业务逻辑层和数据库层之间充当了桥梁, Hibernate 应用的结构如图 2-19 所示。

2. Hibernate 的核心接口

Hibernate 的核心接口有 5 个, 如图 2-20 所示。所有的 Hibernate 应用都会访问 Hibernate 的这 5 个核心接口, 通过接口, 可以存储和获得持久对象, 进行事务控制。这些接口如下。

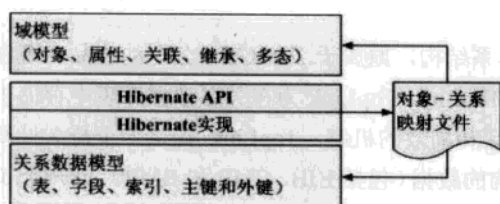


图 2-19 Hibernate 应用的结构图

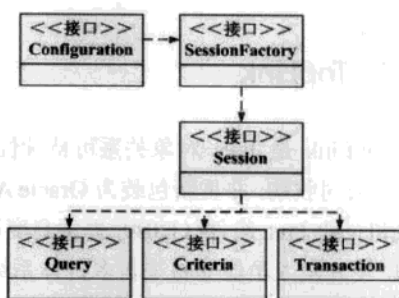


图 2-20 Hibernate 核心接口

① Configuration 接口。Configuration 接口用于配置和启动 Hibernate。Hibernate 应用通过 Configuration 实例来指定对象-关系映射文件的位置或者动态配置 Hibernate 的属性, 然后创建 SessionFactory 实例。

② SessionFactory 接口: SessionFactory 实例对应一个数据存储源, 是 Session 的工厂, 并且提供事务之间可重用的缓存。

③ Session 接口。Session 接口是 Hibernate 应用最广泛的接口, Session 也被称为持久化管理器。它提供与持久化相关的操作, 如添加、更新、删除、加载和查询对象。

④ Transaction 接口。Transaction 接口是 Hibernate 的数据库事务接口, 对底层的 JDBC、JTA、COBRA 进行封装和抽象。

⑤ Query 和 Criteria 接口。是 Hibernate 的查询接口, 用于向数据库查询对象, 以及控制执行查询的过程。Query 实例包装了一个 HQL (Hibernate Query Language) 查询语句。HQL 是完全面向对象的, 它应用类名及类的属性名、而不是表名和表的字段名。

3. Hibernate 的实现模式

Hibernate 的实现模式是先把关系数据库表及其关系模型转化为 Java 中的 JavaBean (Plain Ordinary Java Object, POJO) 和一个描述表和关系的 XML 文件。通过 Hibernate 提供了实现持久化层的一种模式, 它采用元数据来描述对象-关系的映射细节。

实现过程是 Configuration 接口创建 SessionFactory 接口实例化对象, SessionFactory 接口实例化对象创建 Session 接口实例化对象。Session 接口实例化后充当一个容器, 实现 POJO 的增加、修改、查询和删除功能。同时实例化后的 Session 可以创建出 Query 实例化对象和 Criteria 实例化对象, 主要用于查询。Session 也可以创建出 Transaction 接口实例化

对象，主要用于实现事务处理和并发控制。如图 2-21 所示。

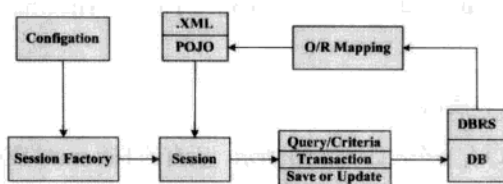


图 2-21 Hibernate 的实现模式

2.5.2 TopLink

TopLink 是 Java 对象关系可持续性体系结构，原属于 WebGain 公司的产品，现在被 Oracle 公司收购，并重新包装为 Oracle AS TopLink。TopLink 为在关系数据库表中存储 Java 对象和企业 Java 组件（EJB）提供高度灵活和高效的机制。TopLink 提供一个持久性基础架构，使开发人员能够将来自多种体系结构的数据（包括 EJB、CMP 和 BMP）、常规的 Java 对象、Servlet、JSP、会话 bean 和消息驱动 bean）集成在一起。

TopLink 的特点：

- 丰富的查询框架，该框架支持面向对象的表现框架、范例式查询（QBE）、EJB QL、SQL 以及存储过程。
- 一个对象级、层次的事务框架。
- 快速的存取能力，保证对象能被一致性地辨识。
- 具有直接映射与关联映射。
- Object 到 XML 的映射，此外也支持 JAXB。
- 支持 EIS / JCA 等数据来源。
- 可视化的映射编辑平台——Mapping Workbench。

2.5.3 Apache OJB

Apache OJB（Apache ObjectRelationalBridge）是一种对象关系映射工具，它能够完成从 Java 对象到关系数据库的透明存储。OJB 使用基于 XML 的对象关系映射。映射发生在一个动态的元数据层，使得在运行时通过一个简单的元对象协议（MOP）就可以操作元数据层去改变存储内核。其官方网站：<http://db.apache.org/ojb/>。

Apache OJB 的特性包括三个方面：

- ① 灵活性，OJB 支持多种持久性 API，为用户提供多种 API 的选择。
 - PersistenceBroker API 充当 OJB 持久性的内核，OTM、ODMG 和 JDO 的实现都是建立在这个内核上的。应用程序可以不需要完全对象层事务而直接使用该 API。
 - OJB 完全兼容 ODMG 3.0 API。

- OJB 兼容 JDO API。现阶段只提供了一个 JDO 参考实现 (RI) 的插件。结合 JDO 的 RI 和 OJB 的插件可以实现一个标准的 JDO 1.0 的 O/R 解决方案。
 - OJB2.0 完全兼容 JDO。
 - 公共对象事务管理层 (OTM) 提供了 JDO 和 ODMG 所有的特征
- ② 可扩展性
- OJB 支持广泛的应用范围：从嵌入式系统到基于 Java EE 架构的多层富客户端应用程序。
 - OJB 可以平滑地集成到 Java EE 应用服务器。它支持 JNDI、JTA 和 JCA。也可以使用 JSP、Servlet 和 SessionBeans。OJB 对 EJB 的 BMP 也提供特别的支持。
- ③ 功能性
- OJB 使用基于对象/关系映射的 XML。映射驻留在一个动态的元数据层，可以在运行时通过一个简单的元数据对象协议 (MOP) 来进行操作并改变持久层的行为。
 - OJB 提供了一些先进的 O/R 特征：如对象缓存，基于虚拟代理和分布式锁管理的事务隔离级别的懒加载、乐观锁定、悲观锁定等。
 - OJB 提供了一个灵活的配置和插件机制，既允许选择预定义设置，也允许自定义插件。

2.6 其他开源框架

2.6.1 与 Log 相关的开源框架

Log 管理是开源框架都不可缺少的部分，iBATIS 也不例外。SQL Map 框架使用 Jakarta Commons Logging (JCL) 记录日志信息。JCL 使用独立于具体实现的方式提供日志服务。通过插件可以实现 Log4J 和 JDK1.4 Logging API 等不同的日志服务。

1. Jakarta Commons Logging

Jakarta Commons Logging 是 Apache 的一个开放源代码项目。

Jakarta Commons Logging (JCL) 提供一个日志 (Log) 接口 (interface)，同时兼顾轻量级和不依赖于具体的日志实现工具。它提供给中间件/日志工具开发者一个简单的日志操作抽象，允许程序开发人员使用不同的具体日志实现工具。用户被假定已熟悉某种日志实现工具的更高级别的细节。JCL 提供的接口，对其他一些日志工具，包括 Log4J、Avalon LogKit 和 JDK 1.4 等，进行了简单的包装，此接口更接近于 Log4J 和 LogKit 的实现。

JCL 通用日志包提供一组通用的日志接口，用户可以自由选择实现日志接口的第三方软件。通用日志目前支持以下的日志实现：① Log4J 日志器；② JDK1.4Logging；③ SimpleLog 日志器；④ NoOpLog 日志器。

JCL 有两个基本的抽象类: Log (基本记录器) 和 LogFactory (负责创建 Log 实例)。当 commons-logging.jar 被加入到 CLASSPATH 之后, 它会合理地选择用户比较喜欢的日志工具, 然后进行自我设置, 用户根本不需要做任何设置。

默认的 LogFactory 是按照下列的顺序和步骤去发现并决定哪个日志工具将被使用的:

- ① 寻找当前 factory 中名叫 org.apache.commons.logging.Log 配置属性的值;
- ② 寻找系统中属性名叫 org.apache.commons.logging.Log 的值;
- ③ 如果应用程序的 classpath 中有 Log4J, 则使用相关的包装类 (Log4JLogger);
- ④ 如果应用程序运行在 Jdk1.4 的系统中, 使用相关的包装类 (Jdk14Logger);
- ⑤ 使用简易日志包装类 (SimpleLog)。

2. Log4J 日志

Log4J 是 Apache 的一个开放源代码项目。Log4J 可以控制日志信息输送的目的地是控制台、文件、GUI 组件, 甚至是套接口服务器、NT 的事件记录器、UNIX Syslog 守护进程等; Log4J 也可以控制每一条日志的输出格式; 通过定义每一条日志信息的级别, Log4J 能够更加细致地控制日志的生成过程。Log4J 通过一个配置文件来灵活地进行配置, 而不需要修改应用的代码。

Log4J 由三大组件构成:

- ① **Logger:** 负责生成日志, 并能根据配置的日志器级别来决定什么日志消息应该被输出。
- ② **Appender:** 定义日志消息输出的目的地, 指定日志消息应该被输出到什么地方, 这些地方可以是控制台、文件和网络设备等。
- ③ **Layout:** 指定日志消息的输出格式。

2.6.2 OSCache

OSCache 是 OpenSymphony 组织提供的一个 Java EE 架构中的缓存技术实现组件。它主要用于处理短时间或一定时间内不会发生变化的一些或局部数据, 缓存在内存或磁盘, 可以充分减轻服务器的压力, 实现负载平衡。这是一种用于提高系统响应速度、改善系统运行性能的技术。OSCache 的结构如图 2-22 所示。

OSCache 最新的稳定版本是 2.0, 其主要特征如下。

- ① **缓存任何对象:** 任何 Java 对象都可以被缓存。
- ② **可选的缓存区:** 可以使用内存、硬盘空间、同时使用内存和硬盘, 或者提供自己的其他资源 (需要自己提供适配器) 作为缓存区。
- ③ **永久缓存:** 缓存能被配置写入硬盘, 因此允许在应用服务器的多次生命周期间, 缓存创建开销昂贵的数据。
- ④ **灵活的缓存系统:** 开发人员可以根据不同的需求、不同的环境选择不同的缓存级别。

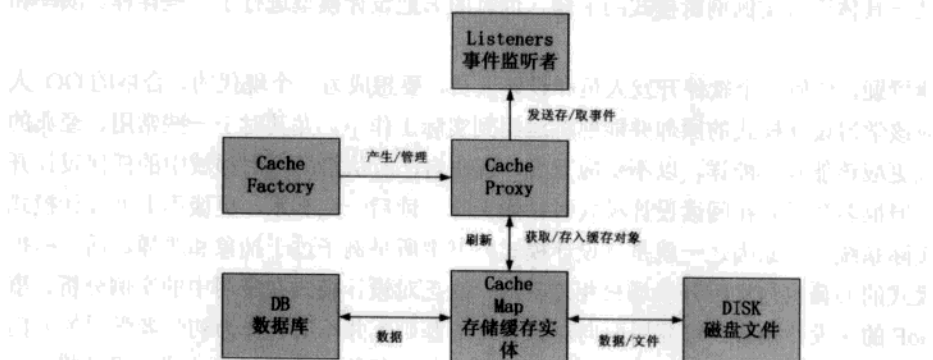


图 2-22 OSCache 结构图

- ⑤ 容错：采用了缓存，有一定的缓冲时间，便于容错。
- ⑥ 支持集群：集群缓存数据能被单个地进行参数配置，不需要修改代码。
- ⑦ 缓存主动刷新：可以有最大限度地控制缓存对象的过期，包括可插入式的刷新策略。
- ⑧ 拥有全面的 API：OSCache 允许编程人员通过编程的方式控制所有缓存系统。

2.6.3 Commons-DBCP 数据库连接池

Commons-DBCP 是 Apache 的一个开放源代码项目。这是一个经典的数据库连接池，它已经被嵌入到 Apache Tomcat Web 服务器中，因此十分的流行。为了使用 Commons-DBCP，需要 Commons 中的另外两个子项目：Commons-Collections 和 Commons-Pool。

Commons-Collections 项目提供了一组扩展 Java 集合框架的类库，它提供了许多十分有用的数据结构，以帮助提高 Java 应用程序开发的效率。Collections 虽然依靠 Java 集合框架，但它也提出了许多新的接口或原接口的新实现，以及许多有用的其他工具类。

Apache 的 Commons-Pool 提供了编写对象池的 API，将用完的对象返回对象池中以便下次利用，从而减少了对对象创建时间。这对于创建对象相对耗时的应用来说，能够提高应用的性能。Commons-DBCP 数据库连接池正是使用 Commons-Pool 来创建和数据库之间的连接对象，在对象池中保存这些对象，从而减少了频繁建立连接对象所造成的性能损耗。

2.7 GoF 的 23 种设计模式

自从 GoF (Erich Gamma、Richard Helm、Ralph Johnson 和 John Vlissides 四位博士) 的《设计模式：可复用面向对象软件的基础》问世以来，在全世界的开发人员形成了一个学习、使用设计模式的热潮。类似书籍也出版了无数，有深入挖掘、扩展各种模式的书籍，

有结合某一具体语言实例剖析模式的书籍，也有的书把设计模型进行了一些诠释、演绎和案例说明。

毋庸置疑，任何一个软件开发人员和设计人员，要想成为一个现代的、合格的 OO 人士，都应该学习设计模式的原理并能熟练运用到实际工作中。尤其对于一些常用、经典的设计模式更应该能耳熟能详，以不变应万变，解决自己所从事的应用领域中的任何设计开发问题。但很多初学者在阅读设计模式时枯燥无味，读后一头雾水，更谈不上把设计模式应用到实际系统中，原因之一就是《设计模式》书中所举例子过于抽象和难懂，而一些扩展设计模式的书籍也与实际开发场景相差甚远，缺乏对设计模式在应用中的实例分析。事实上，GoF 的《设计模式：可复用面向对象软件的基础》并不适合作为初学者学习的入门教材。在《设计模式》一书中，基本都采用了统一信息描述结构模板格式描述设计模式，每一个模式根据标准的模板被分成若干部分，包括模式名称、分类、意图、别名、动机、适用性、结构、参与者、协作、效果、代码示例、已知应用和相关模式 13 个方面来说明。这样太理论化和抽象化，与广大应用开发者的实际应用还是有一定的距离。这种学习模式主要是针对那些有丰富的开发设计经验的实践大师和有深厚 OO 思想的理论专家。他们学习的目的是在是要进行更高层次的抽象、演绎和归纳。但是一般的开发人员和设计人员遵循这种方式去学习，则收效甚微，反而让人望文生畏。同时《设计模式》中案例的实现代码也主要采用了比较旧版的 Smalltalk 和 C++ 等编程语言。而现在是一批更为成熟和高级的当代 OO 编程语言（如 Java、C# 等）占据编程的主流，并且 UML 思想和技术也已经在实际应用中深入人心。这一切都说明我们应该把 GoF 的 23 种设计模式通俗化，大众化，让它服务于和收益于广大的软件开发人员和设计人员。而通过实际的源码分析，把这些设计模式在源码中揭示出来，分析其实现模式和优势，这样无疑能让广大的软件开发人员和设计人员能清晰地理解源码，同时也能迅速地掌握这些设计模式的实质。

GoF 设计模式总共 23 个，其分类有很多种不同的方法。依据设计模式的工作目的不同，模式可分为创建型模式、结构型模式和行为模式三类。创建型模式与对象的创建有关；结构型模式处理类和对象的组合，将一组对象组合成一个大的结构；行为模式描述类或对象的交互和职责分配，定义对象间的通信和复杂程序中的流控。具体介绍如下：

① 创建型模式 (Creational Patterns)：主要负责对象的创建工作，其应用过程不再是简单的直接实例化对象。创建型模式将对象的部分创建工作延迟到子类，而创建型对象模式则将它延迟到另一个对象中。创建型模式包括工厂方法 (Factory Method) 模式、抽象工厂 (Abstract Factory) 模式、构造器 (Builder) 模式、原型 (Prototype) 模式和单例 (Singleton) 模式 5 种模式。

② 结构型模式 (Structural Patterns)：结构型模式处理类或对象的组合，即描述类和对象之间怎样组织起来形成大的结构，从而实现新的功能。结构型模式采用继承机制来组合类。一般用于复杂的用户界面和统计数据，它描述了如何组合类和对象以形成更大的结构。结构型模式包括适配器 (Adapter) 模式、桥梁 (Bridge) 模式、组合 (Composite) 模式、装饰 (Decorator) 模式、门面 (Facade) 模式、享元 (Flyweight) 模式和代理 (Proxy)

模式 7 种模式。

③ 行为型模式 (Behavioral Patterns): 行为型设计模式描述算法以及对象之间的任务 (职责) 分配, 它所描述的不仅仅是类或对象的设计模式, 还有它们之间的通信模式。这些模式刻画了在运行时时刻难以跟踪的复杂的控制流。主要用于精确定义系统中对象之间的通信流程, 以及在一些相当复杂的程序中控制该流程的方法。行为型模式通常可以分为解释器 (Interpreter) 模式、模板 (Template) 模式、职责链 (Chain of Responsibility) 模式、命令 (Command) 模式、迭代器 (Iterator) 模式、调停者 (Mediator) 模式、备忘录 (Memento) 模式、观察者 (Observer) 模式、状态 (State) 模式、策略 (Strategy) 模式和访问者 (Visitor) 模式 11 种模式。

iBATIS 框架整体设计非常巧妙, 框架易于使用和扩展, 具有更大的灵活性、可扩展性和更好的可重用性。因为该框架灵活的使用了多种 GoF 设计模式。这些典型设计模式中对象间的耦合度小, 易于变化和扩展, 通过揭示源码中隐藏的设计模式的提取, 对其使用到的设计模式进行分析, 能更加清晰地看清楚 iBATIS 框架的构成和实现。

iBATIS 框架涉及的设计模式包括: 工厂方法 (Factory Method) 模式, 单例 (Singleton) 模式、桥梁 (Bridge) 模式、装饰 (Decorator) 模式、门面 (Facade) 模式、享元 (Flyweight) 模式、代理 (Proxy) 模式、模板 (Template) 模式、调停者 (Mediator) 模式、状态 (State) 模式、策略 (Strategy) 模式等。在后面章节的源码分析中会一一进行详细描述、分析和解释。

第 3 章

安装和配置 iBATIS 源码

本章内容：

1. 如何安装和配置 iBATIS SQLMap 源码环境。
2. 如何安装和配置 iBATIS DAO 源码环境。
3. 如何安装和配置 iBATIS JPetStore 源码环境，包括 JPetStore Web 程序的创建和发布，MySQL 数据库和 Navicat 数据库管理工具的安装等。

要学习源程序代码，首先要配置好 iBATIS 的这些开源代码，这样才能进行浏览、查阅和读取。笔者的 iBATIS 源码配置环境的编辑工具采用 Eclipse 3.2。当然，Eclipse 的版本愈高愈好。按照三个项目来进行配置。第一个是配置 iBATIS SQL Map 项目。第二个是配置 iBATIS DAO 项目，第三个是配置 iBATIS JPetStore 项目。

3.1 安装和配置 iBATIS SQL Map 源码环境

有几种渠道可以获得 iBATIS SQL Map 的源代码。可以从网站 <http://ibatis.apache.org/> 中获取，也可以从本书配套的光盘中获取（本书配套光盘的源码有一些注释）。

iBATIS SQL Map 最新的版本是 SQL Map 2.3.4，对 JDK 的要求必须是版本 1.5 以上。

下载打包文件后直接解压文件并打开，会出现如图 3-1 所示的 iBATIS SQL Map 目录。

iBATIS SQL Map 目录中包括 doc、lib、simple_example 和 src 目录。其中 doc 目录有 user-Javadoc 和 dev-Javadoc，主要都是 Javadoc 文档，对 iBASIT 的 SQL Map 中的包、类和接口 API 的描述。Lib 目录是 SQL Map 形成的二进制发布包。Simple_example 目录是一个非常简单的 SQL Map 例子程序。src 目录主要就是

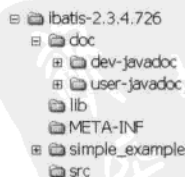


图 3-1 iBATIS SQL Map 目录内容

SQL Map 的源程序代码。

SQL Map 的 JAR 文件和依赖性很少。iBATIS 的一个主要的特点就是管理并降低 JAR 文件的依赖性，需要的基本运行库包含 rt.rar、cglib、commons-logging、Log4J、commons-dbc、oscache、Jboss-J2ee 等。它们根据功能来分类，说明何时需要使用可选 JAR 类库的状况，如表 3-1 所示。

表 3-1 iBATIS SQL Map 依赖的包说明

序号	名 称	内 容	库 名 称	备 注
1	rt	包含了 Jdk 的基础类库，也就是在 Java API 里面看到的所有的类的 class 文件。覆盖了所有的 Java EE 的规范和扩展规范	rt.Jar	也就是第 2 章中 Java 基础知识覆盖的 Java 实现
2	cglib	cglib-2.1.3.Jar，全称是 Code Generation Library，它可以用来动态继承 Java 类或者实现接口，可以实现字节码的动态生成	cglib-2.1.3.Jar	
3	commons-logging	日志包	commons-logging-1.0.4.Jar	
4	Log4J	日志包，可以更加详细地记录日志。可选的	Log4J-1.2.11.Jar	
5	commons-dbc	一种开源的数据库连接池	commons-dbc-1.2.2.Jar	
6	oscache	一种开源的缓存组件	oscache-2.1.Jar	
7	Jboss-J2ee	包括 JTA 包的内容	Jboss-J2ee.Jar	

采用 Jboss-J2ee 的原因是因为里面实现了 JTA（Java Transaction API）的功能，当然也可以替换成任何已经实现了 JTA 规范的 Java 包。

1. 创建 iBATIS SQLMap-2.3.4.726 的 Java 项目

解压源程序包，可以在 Eclipse 中创建一个 Java 项目，并把 SQL Map 的源程序直接复制到 src 目录中去。

2. iBATIS SQLMap-2.3.4.726 项目基本属性配置

iBATIS SQLMap-2.3.4.726 的 Java 项目要做一些基本设置，信息为 UTF-8，如图 3-2 所示。

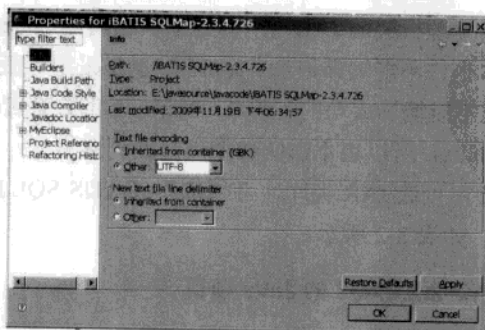


图 3-2 iBATIS SQLMap-2.3.4.726 项目的 Text File Encoding 属性

iBATIS SQLMap-2.3.4.726 项目引用的库文件，如图 3-3 所示。
iBATIS SQLMap-2.3.4.726 项目采用 JDK1.5 进行编译，其编译器属性用 5.0，如图 3-4 所示。

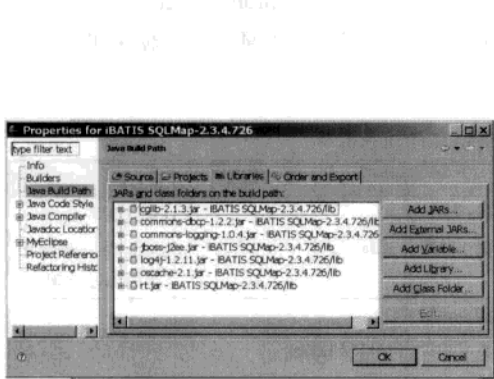


图 3-3 iBATIS SQLMap-2.3.4.726 项目 Libraries 属性

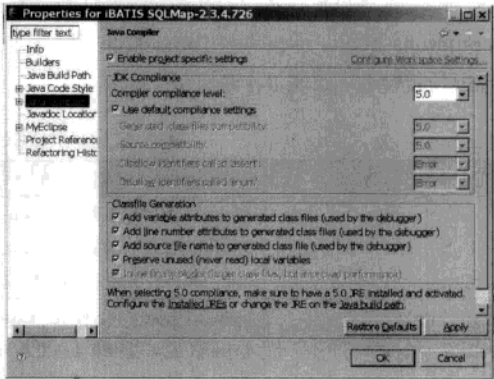


图 3-4 iBATIS SQLMap-2.3.4.726 项目 Java Compiler 属性

通过上述的配置，iBATIS SQLMap-2.3.4.726 就能成功地编译通过。现在，就可以用 Eclipse 编辑器看 iBATIS SQLMap 的源程序代码了。

3.2 安装和配置 iBATIS DAO 源码环境

接着我们来配置 iBATIS DAO 项目。有几种渠道可以获得 iBATIS DAO 的源代码。一方面可从网站 <http://ibatis.apache.org/> 中获取，也可以从本书配套的光盘中获取。

当前 iBATIS_DAO 的最新版本是 iBATIS_DAO-2.2.0.638，对 JDK 的要求同样是 1.5 以上。

下载打包文件后直接解压文件并打开，目录里面其他内容都不重要，主要有一个 src 目录，该目录就是 iBATIS DAO 的源程序代码。

iBATIS DAO 的 JAR 文件比 SQLMap 要多。首先要包含 SQLMap 所依赖的 JAR 包，其次要包含 SQL Map 的 JAR 包，最后由于 iBATIS DAO 集成了 Hibernate、TopLink 和 Apache OJB，所以也引入这三个开源框架的 JAR 包（在这里，引用的 Hibernate 包为 2.0 版本。如果采用 Hibernate3.0 版本也许不能编译通过。TopLink 也是采用老版本，而我使用的是新版本的 TopLink，新版本是不能编译成功的。所以，TopLink 部分有可能编译不过，但是由于这部分代码本来就已经陈旧，所以也不影响整个项目的阅读）。

iBATIS DAO 需要使用可选 JAR 类库，必须依赖 iBATIS SQL Map 已经依赖的包。这里只是介绍新加入的包，如表 3-2 所示。

1. 创建 iBATIS_DAO-2.2.0.638 的 Java 项目

解压源程序包、可以在 Eclipse 中创建一个 Java 项目，并把 iBATIS_DAO-2.2.0.638 的

源程序直接复制到 src 目录中去，这里要提醒一下，由于在 SQLMap 包中已经包含了 com.ibatis.common 包的内容，所以，这里的源代码只有 com.ibatis.dao 包的内容。

表 3-2 iBATIS DAO 所依赖的包说明

序 号	名 称	内 容	库 名 称	备 注
1	Apache OJB	Apache OJB 映射框架的实现包	db-ojb-1.0.4.Jar	
2	Hibernate	Hibernate2 映射框架的实现包	hibernate2. Jar	
3	iBATIS	iBATIS 映射框架的实现包	ibatis-2.3.4.726. Jar	
4	Toplink	Toplink 映射框架的实现包	TopLink-api. Jar	
5	Toplink	Toplink 映射框架的实现包	TopLink-essentials. Jar	
6	Jakarta-ORO	正则表达式的实现包	jakarta-oro. Jar	

2. iBATIS_DAO-2.2.0.638 项目基本属性配置

iBATIS_DAO-2.2.0.638 项目的基本配置与 iBATIS SQLMap-2.3.4.726 项目相同，只有引用包有所区别，如图 3-5 所示。

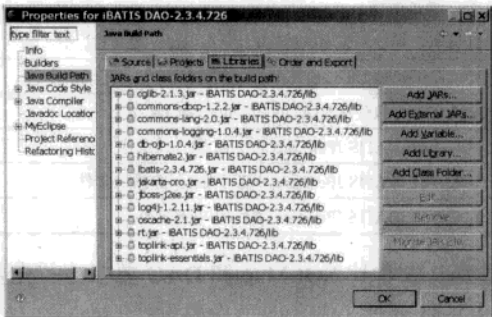


图 3-5 iBATIS_DAO-2.2.0.638 项目 Libraries 属性

通过上述的配置，iBATIS_DAO-2.2.0.638 项目就能成功地编译通过。现在，就可以用 Eclipse 编辑器看 iBATIS DAO 的源程序代码了。

3.3 安装和配置 iBATIS JPetStore 源码环境

由于本文中的大部分源码分析的案例都是基于这个程序。所以，配置 iBATIS JPetStore 项目是比较重要的。

3.3.1 iBATIS JPetStore 源码环境配置

有几种渠道可以获得 iBATIS JPetStore 的源代码。可以从网站 <http://ibatis.apache.org/> 上获取，也可以从本书配套的光盘中获取。

iBatis JPetStore 当前的最新版本是 JPetStore-5.0,对 JDK 的要求必须是 1.5 以上版本。
下载打包文件后直接解压文件并打开,会出现如图 3-6 所示的目录:



图 3-6 iBatis JPetStore 目录内容

iBatis JPetStore 目录中的内容如表 3-3 所示。

表 3-3 iBatis JPetStore 目录内容说明

序 号	目录名称	内 容 描 述	备 注
1	build	本目录中存放由于构建整个 iBatis JPetStore 的 Ant 脚本, 包括构建过程中生成的一切文件	
2	devlib	此目录用于存放开发 iBatis JPetStore 的 Jar 库文件	
3	doc	此目录用于存放 License 和 Versions 说明	
4	lib	此目录用于存放使用 iBatis JPetStore 的 Jar 库文件	
5	src	此目录存放 iBatis JPetStore 的 Java 源代码程序文件	
6	test	此目录存放 iBatis JPetStore 中基于 Junit 的测试文件	
7	web	此目录存放 iBatis JPetStore 项目的 Web 文件	

iBatis JPetStore 的 JAR 文件比 SQLMap 和 DAO 要多。首先要包含 SQLMap 所依赖的 JAR 包,其次要包含 SQLMap 的 JAR 包。再次要包括 DAO 的 JAR 包。最后由于 iBatis DAO 集成了 Struts 等,还要包括 Struts 方面的包。总结何时需要使用可选 JAR 类库的情况(不包含 iBatis DAO 包含的内容),如表 3-4 所示。

表 3-4 iBatis JPetStore 依赖的包说明

序号	名 称	内 容	库 名 称	备注
1	antlr	antlr.jar 语法生成工具包	antlr. Jar	
2	beanaction	iBatis 用于集成 Struts 的框架基础包	beanaction. Jar	
3	c3p0-0.9.0.Jar	c3p0 数据库连接池包	c3p0-0.9.0. Jar	可选
4	commons-digester. Jar	Bean 和 XML 转化的包, 被 Struts 调用	commons-digester. Jar	
5	hsqldb.Jar	Hsqldb 数据库接口包	hsqldb. Jar	可选
6	iBatis-common-2. Jar	iBatis 基础实现包	iBatis-common-2. Jar	
7	iBatis -sqlmap-2. Jar	iBatis SQL Map 实现包	iBatis -sqlmap-2. Jar	
8	iBatis -dao-2. Jar	iBatis DAO 实现包	iBatis -dao-2. Jar	
9	mysql. Jar	MySQL 数据库接口包	mysql. Jar	可选
10	struts. Jar	Struts 实现的包	struts. Jar	

3.3.2 创建 iBATIS JPetStore 的应用

创建 iBATIS JPetStore 的应用就是要创建 iBATIS JPetStore-5.0 的 Web 项目和部署环境。这需要一个 Web 的应用服务器。

1. 创建 iBATIS JPetStore-5.0 的 Web 项目

解压源程序包、可以在 Eclipse 中创建一个 Web 项目，名称叫 iBATIS JPetStore-5.0。并把 iBATIS JPetStore 的 src 源程序直接复制到 iBATIS JPetStore-5.0 Web 项目的 src 目录中去。把 iBATIS JPetStore 的 Web 目录中内容直接复制到 iBATIS JPetStore-5.0 web 项目中的 WebRoot 目录中。

2. iBATIS SQLMap-2.3.4.726 项目基本属性配置

iBATIS JPetStore-5.0 web 项目基本属性配置与 iBATIS SQLMap-2.3.4.726 项目相同，只有引用包有所区别。如图 3-7 所示。

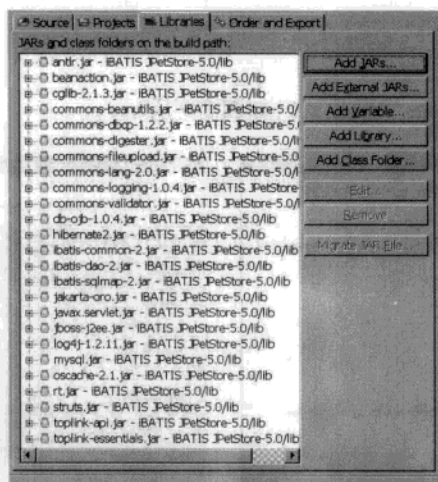


图 3-7 iBATIS JPetStore-5.0 项目 Libraries 属性

通过上述的配置，接着 iBATIS JPetStore-5.0 Web 项目就能成功地编译通过。通过 Eclipse 部署到 Tomcat 服务器上，再安装 MySQL 数据库，就可以调试出这个应用程序。

3.3.3 安装 iBATIS JPetStore 的 MySQL 数据库

在下载的程序默认采用的是 HSQLDB 数据库，本文采用了 MySQL 数据库，在 ddl 目录中有生成 MySQL 数据库的代码，同时要加入 MySQL 的驱动 Java 包——mysql.jar。

只需要把以前基于 hsqldb 的参数修改为基于 MySQL 的参数, hsqldb 的参数如下:

```
driver=org.hsqldb.jdbcDriver
url=jdbc:hsqldb:mem:jpetstore
username=sa
password=
```

修改后的结果如下:

```
driver=com.mysql.jdbc.Driver
url=jdbc:mysql://localhost:3306/jpetstore
username=jpetstore
password=
```

本应用采用 MySQL 来做数据库。在这里简单介绍一些 MySQL 的使用, MySQL 的下载网址: <http://dev.mysql.com/downloads/>。

安装 MySQL 的步骤如下:

1. 启动安装程序

界面如图 3-8 所示。

2. 选择安装类型

一般情况下选择 Custom 安装模式, 如图 3-9 所示。

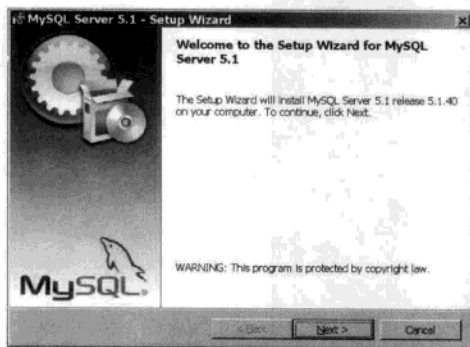


图 3-8 MySQL 安装的启动界面

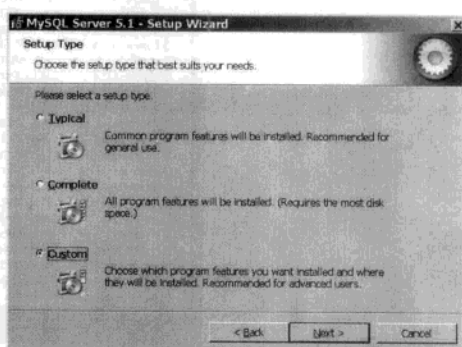


图 3-9 MySQL 安装中选择安装类型

3. 选择 MySQL 应用程序的安装目录

可根据需要选择 MySQL 应用程序的安装目录, 一般不要放到系统盘上。本例选择放到 D 盘上安装, 如图 3-10 所示。

4. 选择 MySQL 应用程序的数据目录

可根据需要选择 MySQL 应用程序的数据目录, 一般不要放到系统盘上。本例选择放到 D 盘上安装, 如图 3-11 所示。

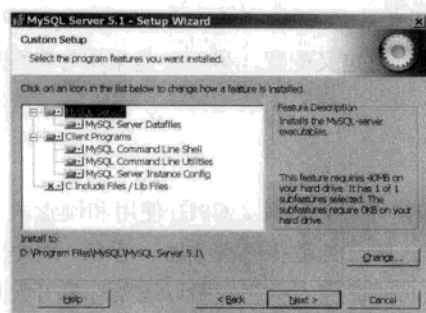


图 3-10 MySQL 安装中选择 MySQL 应用程序的安装目录

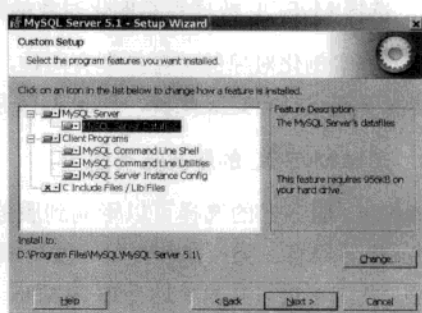


图 3-11 MySQL 安装中选择 MySQL 应用程序的数据目录

5. 开始安装和安装成功表示

接下来就开始安装 MySQL 应用程序，如图 3-12 所示。

我们能做的就是等待，直至安装完成。

6. 进行数据库实例化对象的配置

安装程序成功后，接下来就要对数据库实例进行配置，所谓数据库实例，就是一个数据库的服务，这样 MySQL 才能给客户端进行使用。其数据库实例化对象的配置启动如图 3-13 所示。

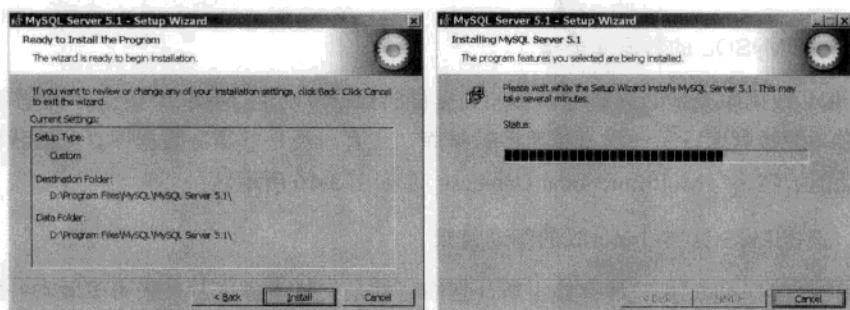


图 3-12 始安装 MySQL

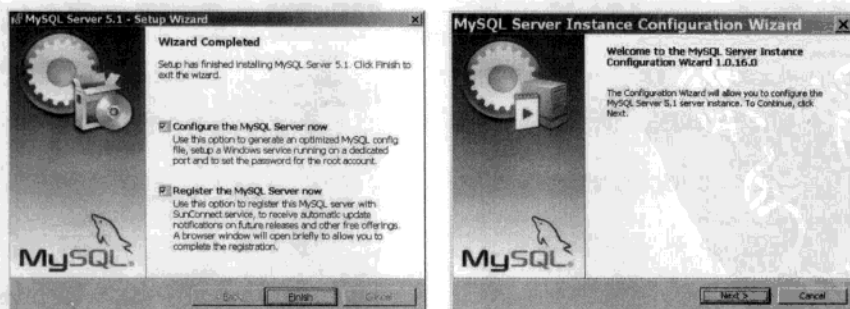


图 3-13 MySQL 安装中

7. 选择 MySQL 数据库实例化对象的配置类型

数据库实例化对象的配置类型有两种，一种是自定义配置，另一种是采用标准配置。一般情况下都选择自定义配置，如图 3-14 所示。

8. 选择 MySQL 的服务器类型

MySQL 服务器的选择可以影响到服务器的内存容量分配、CPU 使用和硬盘大小。MySQL 服务器有三类：第一类是开发用服务器，第二类是普通服务器，还有一类是专业服务器。我们这里只是简单地介绍 iBATIS JPetStore 的应用程序，所以还是选择开发用服务器（Developer Machine），如图 3-15 所示。

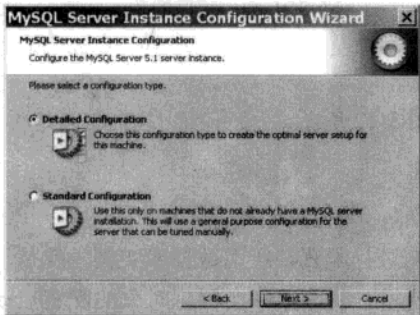


图 3-14 安装 MySQL 中选择数据库实例化对象的配置类型

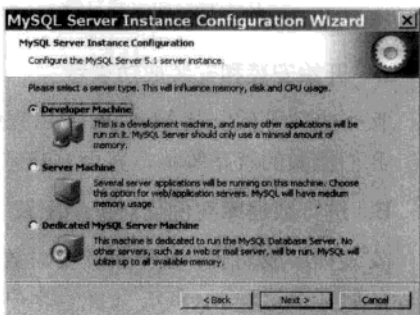


图 3-15 安装 MySQL 中选择服务器类型

9. 选择 MySQL 的数据库类型

MySQL 数据库类型的选择可以影响到提供的数据库功能。MySQL 数据库类型有三类：一类是多功能数据库，另一类是单事务数据库，还有一类是非事务数据库。一般我们还是选择多功能数据库（Multifunctional Database），如图 3-16 所示。

10. 选择 MySQL 中 InnoDB 的驱动目录

MySQL 的 InnoDB 是一种存储引擎，它的特性是支持事务，并且采用多版本并发控制的方式来提高并发度。在这里一般采用默认选择，如图 3-17 所示。

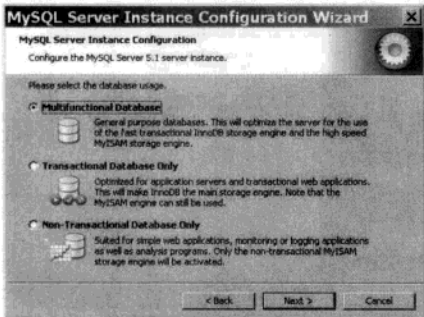


图 3-16 安装 MySQL 中选择数据库类型

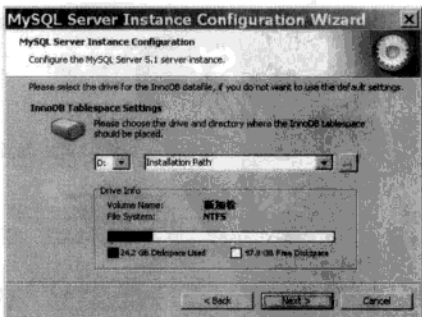


图 3-17 安装 MySQL 中选择 InnoDB 的驱动目录

11. 选择 MySQL 中数据库连接数量

MySQL 的数据库连接数量有三种：第一种是 DDS/OLAP 类型，这种的连接支持最多为 20 个。第二种是 OLTP 类型，这种连接支持可达 500 个。我们一般采用手工来定义数据库连接数并定义为 100 个，如图 3-18 所示。

12. 选择 MySQL 中网络协议和 SQL 模式

选择 TCP/IP 协议是毫无疑问的。MySQL 默认的 TCP 端口是 3306。对于 SQL 模式的采用系统默认模式，如图 3-19 所示。

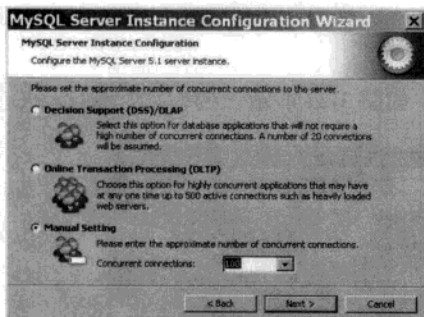


图 3-18 安装 MySQL 中选择数据库连接数量

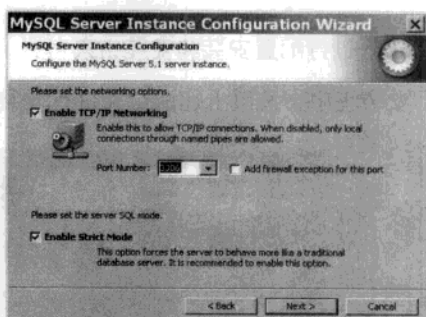


图 3-19 安装 MySQL 中选择网络协议和 SQL 模式

13. 选择 MySQL 中字符集

这个选项也采用模式方式，一般情况下是 UTF-8 字符集模式，如图 3-20 所示。

14. 选择 MySQL 是否支持 Windows Services

针对 Windows Services，一般情况下选择支持，如图 3-21 所示。

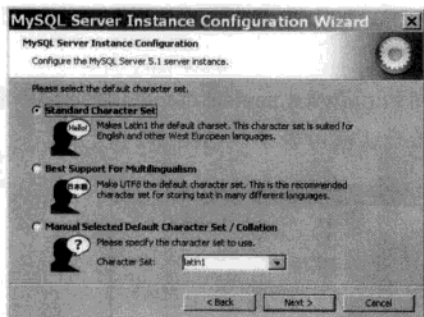


图 3-20 安装 MySQL 中选择字符集

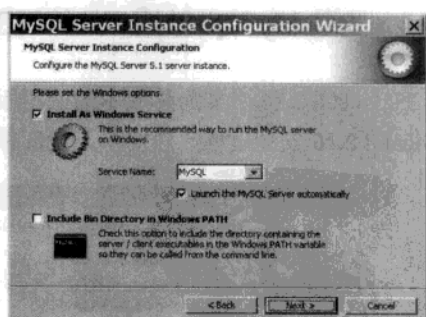


图 3-21 安装 MySQL 中选择是否支持 Windows Services

15. 设置 MySQL 的系统管理员口令

在这里配置 root 的口令，包括是否创建一个匿名账号，如图 3-22 所示。

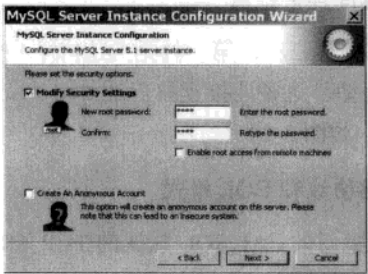


图 3-22 安装 MySQL 中设置系统管理员口令

16. 进行配置安装和安装结束

安装程序会按照上述的定义进行安装和配置，如图 3-23 所示。

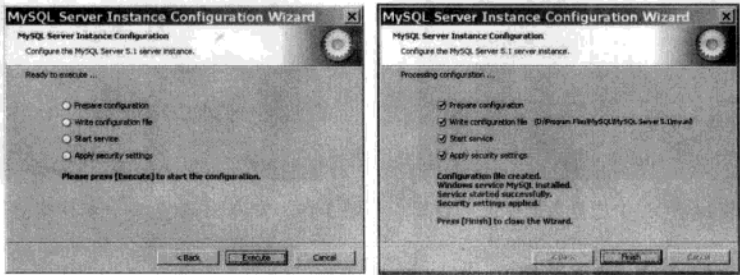


图 3-23 成功安装 MySQL

MySQL 安装成功后，可以考虑创建 JPetStore 的数据库。由于采用 MySQL 的管理工具比较麻烦，笔者采用了一个有试用期的商业软件——Navicat 8.2.16 来做管理工作。

3.3.4 安装 MySQL 数据库的管理工具

Navicat 8.2.16 中文简体版本的试用版本可以到 <http://www.navicat.com/cn> 地址去下载。Navicat 8.2.16 安装过程比较简单，全部都可以直接确认就可以了，如图 3-24 所示。

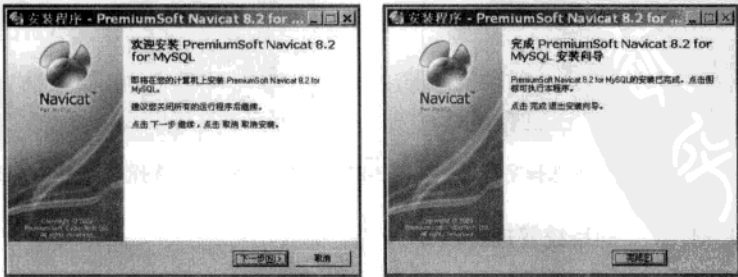


图 3-24 安装 Navicat 的启动和安装成功界面

创建和初始化 JPetStore 的数据库。

1. 首先进入 Navicat

Navicat 应用程序界面如图 3-25 所示。

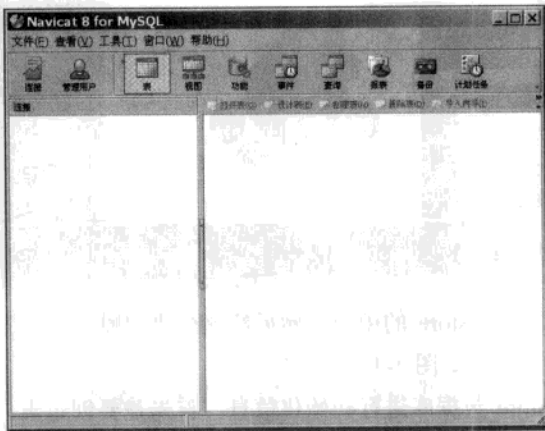


图 3-25 Navicat 应用程序界面

2. 新建一个数据库连接

在界面的空白处单击鼠标右键, 出现如图 3-26 所示的菜单。然后选择菜单“创建连接”, 出现如图 3-27 所示的数据库连接参数设置。

连接名称可以随便定义为 `manager`, 后面几个参数比较重要, 首先主机名/IP 地址要输入 `localhost`, 端口为默认的 `3306`, 用户名为 `root`, 密码就是安装 MySQL 定义的口令。

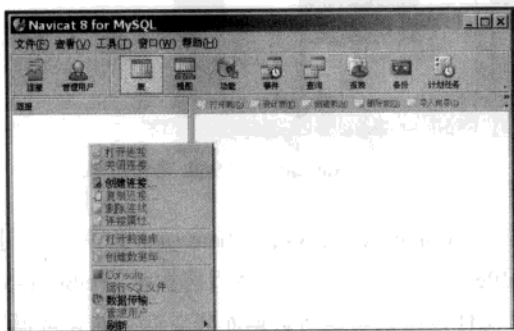


图 3-26 Navicat 创建数据库的菜单选项

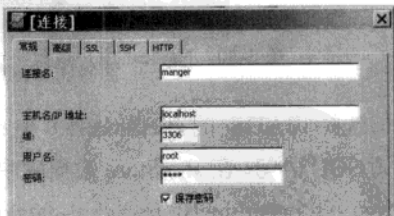


图 3-27 Navicat 数据库连接信息输入

定义了 `manager` 数据库连接, 然后启动 `manager` 连接(确认 MySQL 数据库已经启动)。在 `manager` 连接上单击鼠标右键, 出现弹出式菜单, 然后选择“创建数据库”, 出现如图 3-28 所示的内容。

这样可以创建一个名为 JPetStore 的数据库。然后在 `manager` 连接上单击鼠标右键, 出

现弹出式菜单，然后选择“管理用户”，然后在弹出的窗口中单击“添加用户”，出现如图 3-29 所示的内容。

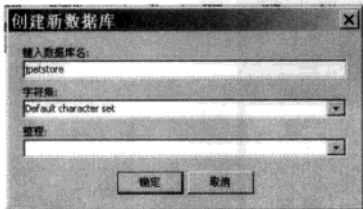


图 3-28 Navicat 创建数据库信息输入

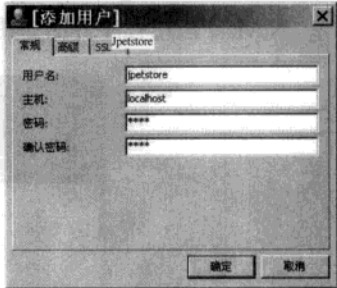


图 3-29 Navicat 添加用户信息输入

可以添加一个名为 JPetStore 的用户。然后给这个用户赋权，即把 JPetStore 数据库的权限赋值给 JPetStore 用户，如图 3-30 所示。

接着开始对 JPetStore 数据库进行初始化信息。首先是要创建表。选择 JPetStore 数据库节点，然后单击鼠标右键，出现如图 3-31 所示菜单。

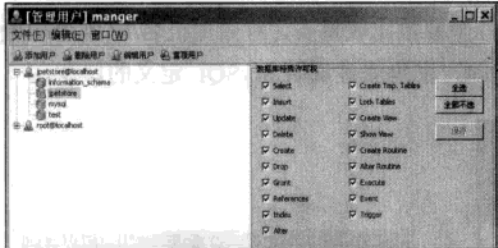


图 3-30 Navicat 用户赋权

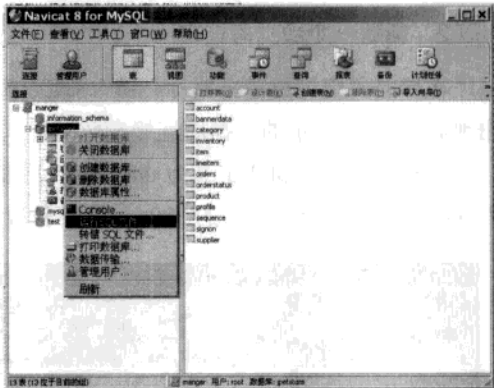


图 3-31 Navicat 运行 SQL 语句菜单选项

选择 iBATIS 的 JPetStore-5.0 中 src\ddl\mysql\jpetstore-mysql-schema.sql 文件，然后执行该 SQL 语句。在 JPetStore 数据库中创建 JPetStore 的数据库表。然后再执行 JPetStore-5.0 中 src\ddl\mysql\jpetstore-mysql-dataload.sql 文件，可以对这些数据库表进行初始化数据。这样数据库配置便全部完成了。

3.3.5 配置成功的标志

本例采用的应用服务区为 Tomcat，修改配置信息文件（主要是数据库连接的配置信息）。然后直接把光盘的程序发布到 Tomcat。部署成功后，启动 Tomcat，在浏览器上输入：

<http://localhost:18080/iBATIS-JPetStore/shop/index.shtml>。得到界面如图 3-32 所示。

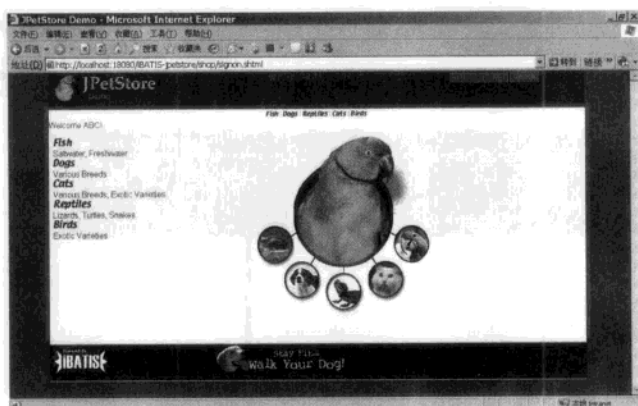


图 3-32 iBATIS JPetStore 的启动界面

然后里面的每个页面都能运行（这个很重要），这说明 iBATIS JPetStore-5.0 Web 项目安装和配置成功。至少能出现如图 3-33 所示的界面，说明应用程序和数据库都配置成功。

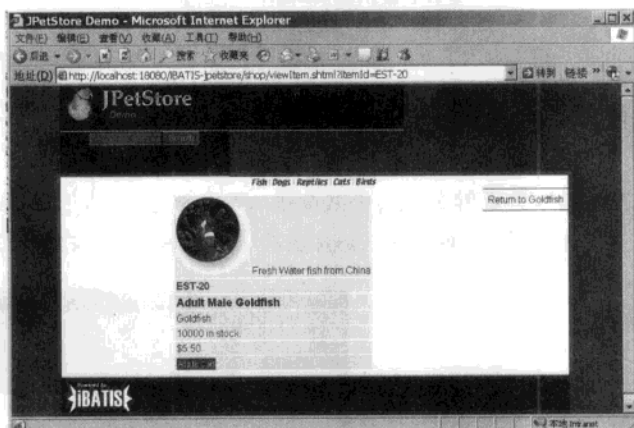
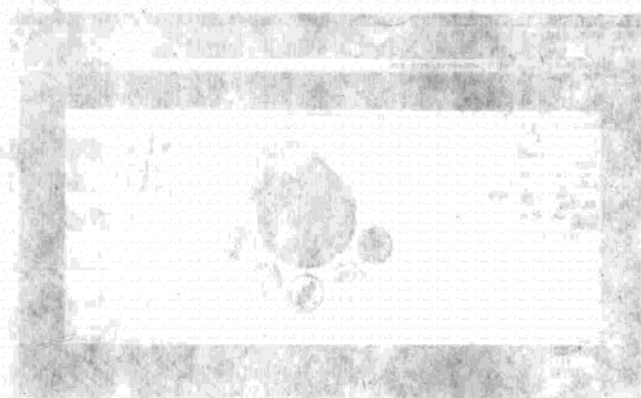


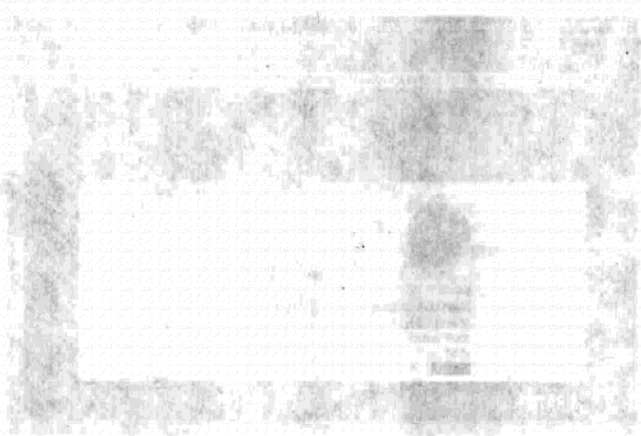
图 3-33 iBATIS JPetStore 成功安装的界面





第一屆全國人民代表會議第一次會議

第一屆全國人民代表會議第一次會議，於十月一日在中華人民共和國首都北京隆重開幕。會議由毛澤東主席主持，並由朱德、周恩來、林彪、劉少奇、宋慶齡等領導人出席。會議的主要任務是討論和通過《中華人民共和國憲法草案》，並選舉中央人民政府委員會。



第一屆全國人民代表會議第一次會議

第一屆全國人民代表會議第一次會議，於十月一日在中華人民共和國首都北京隆重開幕。會議由毛澤東主席主持，並由朱德、周恩來、林彪、劉少奇、宋慶齡等領導人出席。會議的主要任務是討論和通過《中華人民共和國憲法草案》，並選舉中央人民政府委員會。

第一屆全國人民代表會議第一次會議，於十月一日在中華人民共和國首都北京隆重開幕。會議由毛澤東主席主持，並由朱德、周恩來、林彪、劉少奇、宋慶齡等領導人出席。會議的主要任務是討論和通過《中華人民共和國憲法草案》，並選舉中央人民政府委員會。

第一屆全國人民代表會議第一次會議，於十月一日在中華人民共和國首都北京隆重開幕。會議由毛澤東主席主持，並由朱德、周恩來、林彪、劉少奇、宋慶齡等領導人出席。會議的主要任務是討論和通過《中華人民共和國憲法草案》，並選舉中央人民政府委員會。

第一屆全國人民代表會議第一次會議，於十月一日在中華人民共和國首都北京隆重開幕。會議由毛澤東主席主持，並由朱德、周恩來、林彪、劉少奇、宋慶齡等領導人出席。會議的主要任務是討論和通過《中華人民共和國憲法草案》，並選舉中央人民政府委員會。

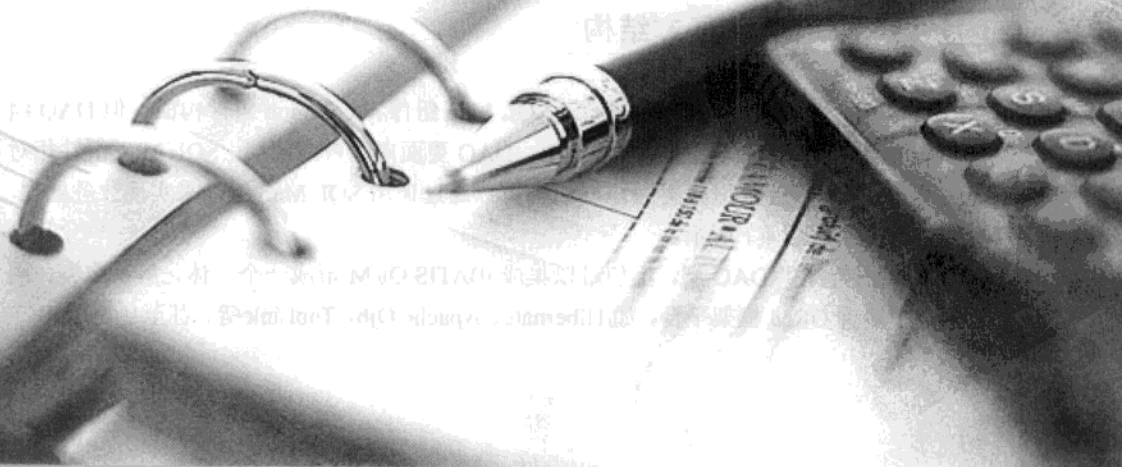
第一屆全國人民代表會議第一次會議，於十月一日在中華人民共和國首都北京隆重開幕。會議由毛澤東主席主持，並由朱德、周恩來、林彪、劉少奇、宋慶齡等領導人出席。會議的主要任務是討論和通過《中華人民共和國憲法草案》，並選舉中央人民政府委員會。

第一屆全國人民代表會議第一次會議，於十月一日在中華人民共和國首都北京隆重開幕。會議由毛澤東主席主持，並由朱德、周恩來、林彪、劉少奇、宋慶齡等領導人出席。會議的主要任務是討論和通過《中華人民共和國憲法草案》，並選舉中央人民政府委員會。



★ CAD CITAB

DATE _____



第 4 章

iBatis DAO 体系结构和实现

本章内容:

1. 首先介绍 iBatis DAO 基本结构, 包括 DAO 的设计模式, iBatis DAO 包文件和使用 iBatis DAO 的工作流程。
2. 介绍 iBatis DAO 外部接口和实现。
3. 介绍 iBatis DAO 的配置文件 dao.xml 的读取过程。说明了 dao.xml 的文件格式和 XSD, iBatis DAO 如何读取和解析 dao.xml。包括 iBatis DAO 如何验证 dao.xml 满足 dtd 的文件格式。
4. 重点介绍 iBatis DAO 引擎的实现, 包括 DAO 业务实现的序列图和说明, iBatis DAO 组件管理, iBatis DAO 事务管理实现。
5. 说明基于 iBatis DAO SQL Map 的实例

iBatis DAO 框架是 iBatis 组件之一, 其主要功能是帮助开发人员基于 DAO 设计模式设计和开发 J2EE 应用程序。iBatis DAO 框架总共有 2000 行程序代码, 从代码量来看还是比较少的。但由于其涉及的范围比较广, 功能实现比较抽象, 因此在分析 iBatis DAO 框架程序代码时还有一定的难度。

4.1 iBatis DAO 基本结构

iBatis 框架平台主要由 DAO 组件、SQL Map 组件和 common 组件构成, 但 DAO 组件与 SQL Map 组件并不是在同一个层次上。DAO 更面向用户应用层, SQL Map 组件相对于要底层一些, 更靠近数据源层。DAO 组件可以通过调用 SQL Map 组件来实现业务处理, 两者的结构层次如图 4-1 所示。

iBatis 框架提供 DAO 层, 不但可以集成 iBatis ORM 形成一个一体化解决方案, 同时还融合了多种 ORM 框架平台, 如 Hibernate、Apache Ojb、TopLink 等, 甚至包括 JDBC、

JTA 等都可以集成到 DAO 中。

在这里要澄清一个概念。iBATIS 的 DAO 组件并不是一种 J2EE 核心设计模式，而是为了实现 J2EE 核心设计模式——DAO 设计模式而开发的一些辅助工具或者基础平台。它的目的是为了让二次开发人员更好地支持 DAO 模式的开发。

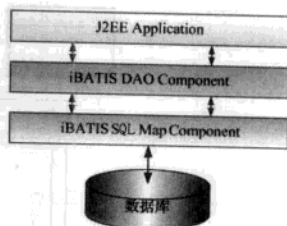


图 4-1 iBATIS DAO 和 SQL Map 层次结构

4.1.1 Java EE 核心设计模式——DAO 模式介绍

研究 iBATIS DAO 组件，首先要简单地了解一下 DAO 模式。DAO(Data Access Objects) 模式是一种核心 Java EE 设计模式。它将底层数据访问操作和上层的业务逻辑分开，使用 DAO 模式可以通过创建简单配置来实现对不同的数据访问模式，且不仅仅限制在数据库类型，还可以延伸到实现数据访问的方式。DAO 模式是 Data Accessor 模式和 ADO(Active Domain Object) 模式两个模式的组合。Data Accessor 模式的实质就是封装了对数据库访问的实现机制，仅对应用程序公开逻辑操作。实现 Data Accessor 的功能复杂程度由开发的封装细节的程度来决定。从外部来看，Data Accessor 提供了黑盒式的数据存取接口。而 ADO(Active Domain Object) 模式实现了业务数据的对象化封装，将数据模型的细节也封装在单一组件中，对应用程序仅公开与业务逻辑相关的通用操作。通过引入 DAO 模式，业务逻辑更加清晰，且富于形象性和描述性，更利于系统日后的维护和升级。图 4-2 是 DAO 模式的类图，该类图表明了 DAO 模式的各个参与者之间的关系。

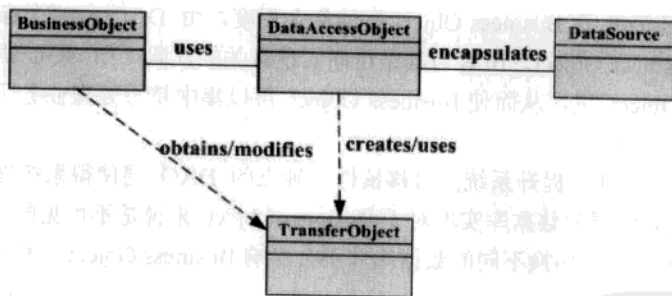


图 4-2 Data Access Objects (DAO) 设计模式

在 DAO 模式中有 Business Object、DataAccessObject、DataSource 和 TransferObject 几个角色。BusinessObject 可以直接调用 DataAccessObject 对象，也可以转化为 TransferObject 对象再去调用 DataAccessObject 对象，DataAccessObject 对象就是处于 BusinessObject 和 DataSource 的中间层。DAO 模式的序列图如图 4-3 所示，该序列图则显示模式的各个对象之间的协作和实现。DAO 模式可以提供更好的解耦，将业务逻辑层与持久层访问层技术分离，使业务逻辑层无须关注底层数据库访问的实现。

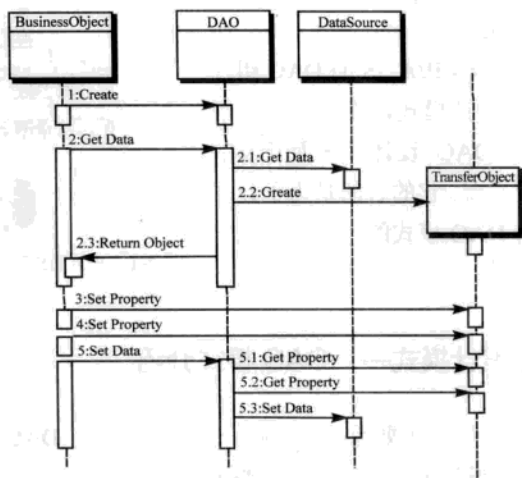


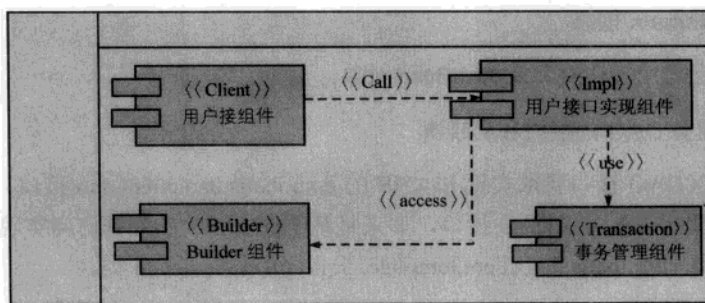
图 4-3 Data Access Objects (DAO) 序列图

使用 DAO 模式主要有如下优势。

- (1) DAO 模式可抽象出数据访问方式，在 Business Object 访问数据源时，完全不涉及数据源。Business Object 与数据源完全解耦。
- (2) DAO 模式采用统一的抽象化的 API。业务组件只调用 DAO 提供的接口就可以实现数据的访问。
- (3) DAO 将数据访问集中在独立的一层。因为所有的数据访问都由 DAO 代理，这层独立的 DAO 就将数据访问的实现与系统的其余部分剥离，将数据访问集中使得系统更具可维护性。
- (4) DAO 还降低了 Business Object 层的复杂程度。由 DAO 管理复杂的数据访问，从而简化了 Business Object。所有与数据访问实现有关的代码（如 SQL 语言等）都不用写在 Business Object 里，从而使 Business Object 可以集中精力处理业务逻辑，提高了代码的可读性和生产率。
- (5) DAO 还有助于提升系统的可移植性。独立的 DAO 层使得系统能在不同的数据库之间轻易切换，底层的数据库实现对于 Business Object 来说是不可见的。数据移植时影响的仅仅是 DAO 层，切换不同的数据库并不会影响 Business Object，因此提高了系统的可复用性。

4.1.2 iBatis DAO 包文件和组件结构

iBatis DAO 核心组件文件主要是在 com.ibatis.dao 包下的 34 个文件。iBatis DAO 组件结构包括四个部分，第一部分为 Client 部分，主要提供给外部的接口部分。第二部分是配置文件生成的 Builder 组件，第三部分是具体实现的 Impl 组件。第四部分是用于事务管理的 Transaction 组件。这四部分之间的关系如图 4-4 所示。



DAO 组件框架

图 4-4 DAO 组件框架

用户接口部分（Client）主要是给外部二次开发人员进行调用的 API，是 DAO 与外部的接口部分。iBATIS DAO 中所有的业务 dao 类都要实现 Client 里的 Dao 接口。DaoManagerBuilder 是 DaoManager 的工厂类，负责生产 DaoManager 的实例化对象。DaoTransaction 实例化对象在 dao 类中定义，可以通过 DaoManager 对象来获取。

Builder 组件只有一个类 XmlDaoManagerBuilder，其功能是把 dao.xml 配置文件转化成 engine 组件中用户接口实现组件的类的各个实例化对象，主要包括 StandardDaoManager 类实例化对象、DaoContext 类实例化对象、DaoImpl 类实例化对象和 DaoProxy 类实例化对象。

客户接口实现（Impl）组件主要包括 StandardDaoManager 类、DaoContext 类、DaoImpl 类和 DaoProxy 类，客户接口实现组件是业务实现的核心。为了搞清楚这几者的关系，可以按照以下思路去理解，StandardDaoManager 类实例化对象主要是总体信息载体，在 StandardDaoManager 实例化对象充当门面类，其属性中包含多个 DaoContext 类实例化对象、多个 DaoImpl 类实例化对象和多个 DaoProxy 类实例化对象，任何操作都要经过 StandardDaoManager 对象来进行调度处理。关于 StandardDaoManager 类的介绍在引擎实现中有详细描述。DaoContext 类实例化对象主要是针对当前的环境。一个 DaoContext 类实例化对象对应一个数据库连接和数据库环境，它可以操作多个 Dao 对象。DaoImpl 类实例化对象主要是针对 Dao 接口，但并不是其实现的对象；它的作用应该是一个中转站，或者是一个 Dao 接口的临时居住地。在这里有一个 DaoProxy 类，它是实例化对象代理 Dao 接口的实现，而 Dao 接口主要是配置文件中设置的接口。

Transaction 组件主要是针对不同的数据源或者是 ORM 接口而实现的事务处理。Transaction 组件通过 TransactionManager 和 Transaction 接口来实现客户的业务事务管理。Transaction 组件包括 SQL Map 组件、Hibernate 组件、JDBC 组件、JTA 组件、EXTERNAL 组件、Apache OJB 组件内容和 Toplink 组件等。同时也是映射这些不同的事务实现策略。

4.1.3 使用 iBATIS DAO 工作流程

一般情况下，我们用 iBATIS DAO 进行开发的基本步骤如下。

1. 创建 domain 信息

这主要是创建 POJO 对象或 JavaBean 对象。

2. 创建业务 DAO 的接口和实现类

业务 X X X DAO 接口要求实现 iBATIS 的 com.ibatis.dao.client.dao 接口。X X X DAO 实现类一方面要实现 X X X DAO 接口，如果是基于 SQLMap 的数据访问实现，那么 DAO 实现类要求继承 com.ibatis.dao.client.template. SqlMapDaoTemplate 类。

3. 编写配置文件——dao.xml

在 dao.xml 文件中，把业务 Dao 接口和 Dao 实现类关联起来。这样实现了接口和实现的松耦合。

4. 创建基础信息处理类，调用应用程序

接着就可以生成基本的 daoManager 实例化对象。

```
String resource = "com.ibatis/jpetstore/persistence/dao.xml";
Reader reader = Resources.getResourceAsReader(resource);
DaoManger daoManager = DaoManagerBuilder.buildDaoManager(reader);
```

对于 daoManager 实例化对象，可以注入到 SqlMapDaoTemplate 类中。这样一来，X X X DAO 实现类就可以直接进行 CRUD 操作了。

4.2 iBATIS DAO 外部接口和实现

iBATIS DAO 外部接口主要包括在 com.ibatis.dao.client 内的 11 个文件，覆盖 3 个接口文件、6 个抽象类和 2 个具体类，也是 iBATIS DAO 的 client 组件。

4.2.1 iBATIS DAO 框架外部接口

iBATIS DAO 框架的外部接口主要由以下五个部分组成，如图 4-5 所示。

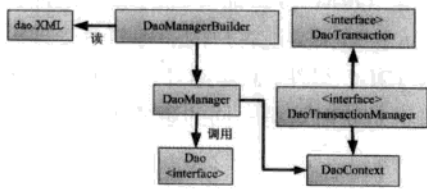


图 4-5 iBATIS DAO 框架的外部接口

下面就这五个部分做一个简单说明。

1. DaoManager

① 通过 buildDaoManager()方法完成 dao.xml 文件的读取，负责 DAO 框架的配置。

② 当系统需要调用某一个特定的 DAO 的时候，DaoManager 就会提供相对应的 DAO，它隐

以统一的形式来实现。

2. DaoTransaction 接口

事务连接的接口，通常是通过 JDBC 连接对象来实现的。

3. DaoException 异常类

所有 Dao 的方法和类成员在处理中抛出异常时，Dao 处理转向 DaoException，更有利于开发人员发现自己程序中的错误。

4. Dao 接口和 DaoTemplate 抽象类

Dao 接口是系统设计中所有的 Dao 都必须继承的接口。它提供了访问关系数据库系统所需的所有操作的接口，其目的是将底层数据访问操作与高层业务逻辑分离开，对上层提供面向对象的数据访问接口。

DaoTemplate 抽象类实现 Dao 接口，它是系统设计中所有 Dao 都必须继承的抽象类。它的子类分别提供了访问关系数据库系统所需的不同的 ORM 模型的操作。

5. DaoManagerBuilder 类

通过调用 DaoManagerBuilder 的 buildDaoManager 方法，获得实现 DaoManager 接口的实例化对象。

4.2.2 iBATIS DAO Template API 结构和说明

iBATIS DAO API 可以支持 5 种与数据交互的方式，分别为 JDBC、iBATIS SQLMap、Hibernate、Apache OJB、TopLink。iBATIS DAO API 用 JDBC 和 JTA 实现直接与数据库交互，也可以通过 iBATIS 本身的 SQLMap 实现持久层，当然，在这些过程中还可以使用第三方的 ORM 实现模式，如 Hibernate、Apache OJB、TopLink 等。这些实现方式都是通过 iBATIS DAO Template API 来实现的。

iBATIS DAO Template API 采用 GoF 中的模板方法设计（Template Method Pattern）模式。iBATIS DAO API 的外部接口和类的类结构如图 4-6 所示。在图 4-6 中，所有具体模板抽象类都继承抽象类 DaoTemplate，而抽象类 DaoTemplate 是 Dao 接口的实现。

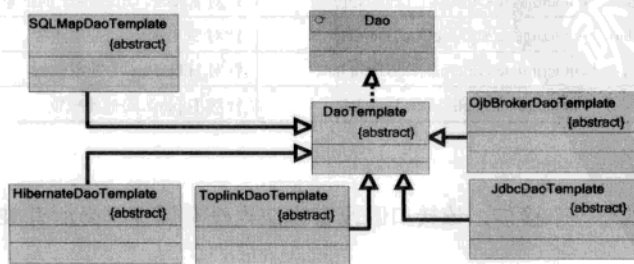


图 4-6 iBATIS DAO Template API 的外部接口的类图

关于 GoF 设计模式中的模板方法（Template Method）模式，就是定义一个操作中的算法的骨架，而将一些步骤延迟到子类中。Template Method 使得子类可以不改变一个算法的结构即可重定义该算法的某些特定步骤。模板方法（Template Method）模式属于行为型设计模式。用 Java 语言的实现结构来说，模板方法模式准备一个抽象类，将部分逻辑以具体方法及具体构造子的形式实现，然后声明一些抽象方法来迫使子类实现剩余的逻辑。不同的子类可以使用不同的方式实现这些抽象方法，从而对剩余的逻辑有不同的实现。也就是说，先制定一个顶级逻辑框架，而将逻辑的细节留给具体的子类去实现。模板的方法（Template Method）模式的结构如图 4-7 所示，其角色包括抽象模板（Abstract Class）角色

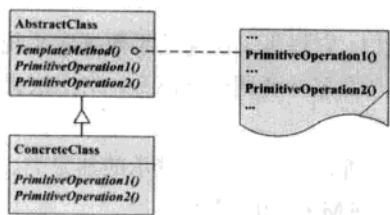


图 4-7 模板方法模式结构

和具体模板（Concrete Class）角色，说明如下：

① 抽象模板（Abstract Class）角色有如下的责任：定义了一个或多个抽象操作，以便让子类实现。这些抽象操作叫做基本操作，它们是一个顶级逻辑的组成步骤，定义并实现了一个模板方法。这个模板方法一般是一个具体方法，给出了一个顶级逻辑的骨架，而逻辑的组成步骤在相应的抽象操作中推迟到子类实现。顶级逻辑也有可能调用一些具体方法。

② 具体模板（Concrete Class）角色有如下的责任：实现父类所定义的一个或多个抽象方法，它们是一个顶级逻辑的组成步骤。

每一个抽象模板角色都可以有任意多个具体模板角色与之对应，而每一个具体模板角色都可以给出这些抽象方法（也就是顶级逻辑的组成步骤）的不同实现，从而使得顶级逻辑的实现各不相同。

对于 iBATIS DAO Template API 而言，其中 DaoTemplate、JdbcDaoTemplate、OjbBrokerDaoTemplate、SqlMapDaoTemplate、HibernateDaoTemplate、ToplinkDaoTemplate 都是抽象模板，而 DaoTemplate 是其他各个抽象模板的再抽象模板。具体模式即是应用程序中继承这些抽象模板的实现类，其抽象模板类的说明如表 4-1 所示。

表 4-1 iBATIS DAO 抽象模板类列表说明

序 号	类结构和名称	作 用	备 注
1	com.ibatis.dao.client.template.JdbcDaoTemplate	直接对 Jdbc 操作的抽象模板	
2	com.ibatis.dao.client.template.OjbBrokerDaoTemplate	针对 Apache OjbBroker 的抽象模板	
3	com.ibatis.dao.client.template.SqlMapDaoTemplate	针对 SQL Map 的抽象模板	
4	com.ibatis.dao.client.template.HibernateDaoTemplate	针对 Hibernate 的抽象模板	
5	com.ibatis.dao.client.template.ToplinkDaoTemplate	针对 Toplink 的抽象模板	

1. Dao 接口

Dao 是接口只是一个接口，该接口既没有方法也没有属性，是所有 iBATIS 的 Dao 的祖先接口。

2. DaoTemplate 抽象类

DaoTemplate 抽象类是实现 Dao 接口的抽象类，是其他具体各个实现框架的祖先类。它所起的作用是引入了 protected DaoManager。DaoManager 是 iBATIS DAO 组件的接口。通过 DaoTemplate 抽象类，使得 Client 组件与 iBATIS DAO 组件及其 iBATIS DAO 事务组件结合起来使用。

3. 各个具体框架抽象类

在 iBATIS DAO 具体支持的抽象框架有 JDBC、Apache Ojb、Hibernate、SQL Map、Toplink 等。

针对 SQLMap 的实现模式，在 SqlMapDaoTemplate 类中有实现 CRUD 的基本方法。在初始化的时候获得 DaoManager 对象。在进行业务操作时通过 DaoManager 对象的 getTransaction 方法获得 SqlMapDaoTransaction 对象。通过 SqlMapDaoTransaction 对象的 getSqlMap 方法获得 SqlMapExecutor 对象。而 SqlMapExecutor 对象实际上就是 SQL Map 的 Client。所以，一切对 SqlMapDaoTemplate 的操作，最终转嫁到在 SQL Map 上 Client 的同样操作，其序列图如图 4-8 所示。

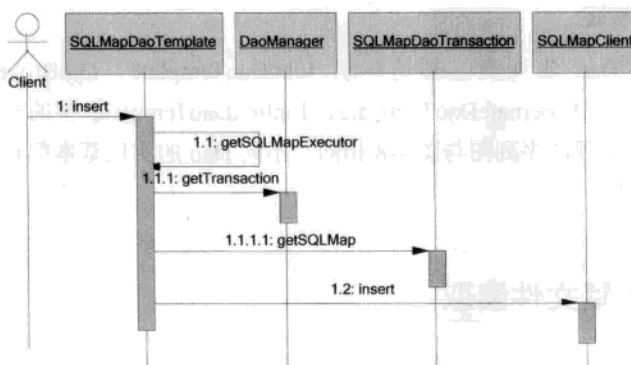


图 4-8 iBATIS DAO SqlMapDaoTemplate 实现序列图

对 SqlMapDaoTemplate 类进行 Insert 操作的程序代码如下：

```

public Object insert(String id, Object parameterObject) {
    try {
        return getSqlMapExecutor().insert(id, parameterObject);
    } catch (SQLException e) {
        throw new DaoException("Failed to insert - id ["
            + id + "], parameterObject [" + parameterObject + "]. Cause: " + e, e);
    }
}

```

通过 getSqlMapExecutor()方法获得 SqlMapExecutor 对象，再调用 SqlMapExecutor 对

象的 insert 方法。获得的 SqlMapExecutor 对象代码如下。

```
protected SqlMapExecutor getSqlMapExecutor() {
    SqlMapDaoTransaction trans = (SqlMapDaoTransaction) daoManager.getTransaction(
        this);
    return trans.getSqlMap();
}
```

而 trans.getSqlMap 的实现代码如下。

```
...
private SqlMapClient client;

public SqlMapClient getSqlMap() {
    return client;
}
...
```

SqlMapDaoTemplate 在进行 CRUD 操作中做了两项工作，第一是封装了 SqlMapExecutor 对象的获得过程，第二是对异常进行了封装，把本应该是 SQLException 转化为了 DaoException 异常。

4. 二次开发实现

任何外部的 Dao 实现类，都是继承 JdbcDaoTemplate、OjbBrokerDaoTemplate、SqlMapDaoTemplate、HibernateDaoTemplate、ToplinkDaoTemplate 中的一个类然后再实现 Dao 接口。所以其实现的序列图与图 4-8 相同。不同 Dao 的实现基本都是类似的，唯一的区别就是 Dao 不同而已。

4.3 DAO 配置文件读取

iBATIS DAO 框架配置文件的信息包括：① DAO Context 信息；② 对于每个 Context 的事务管理器执行情况；③ TransactionManager 性能配置属性；④ 一个 DAO Context 内部的一组相关 Dao 接口和 Dao 实现的配置信息。

4.3.1 dao.xml 的格式说明

dao.xml 是 iBATIS DAO 框架的信息配置载体。dao.xml 配置文件的 XSD 如图 4-9 所示。也可以用 dao-2.dtd 来表示，参见附录一。其结构描述如下。

daoConfig 是根节点。Context 是第一层子节点，包含相关的 DAO Context 信息。TransactionManager 节点是属于 Context 节点下的第 2 层节点，包含当前 context 的事务处理相关信息，即与数据库连接的参数、事务参数等信息。dao 节点也属于 Context 节点下的

第 2 层节点, dao 节点包含系统所有的业务 dao 接口及其对应的实现类。
properties 节点是包含其他的属性信息。

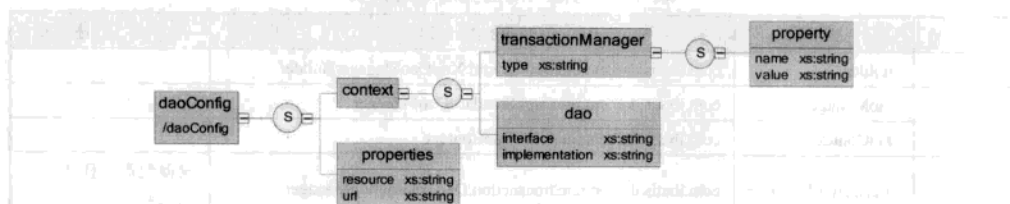


图 4-9 iBATIS DAO 的 dao.xml 的 XSD 结构

4.3.2 dao.xml 文件的读取过程

1. dao.xml 文件读取总体说明

配置文件读取非常简单, 只有一个 XmlDaoManagerBuilder 类就实现了对配置文件 dao.xml 进行读取, 而且采用的是从 daoConfig 根节点开始。为了理解方便, 可分为下列四步。

第 1 步: 对 DaoConfig 进行处理。daoConfig 的信息创建 StandardDaoManager 对象。对 context 子节点进行循环读取, Context 子节点信息创建 DaoContext 对象。

第 2 步: 对 Context 子节点内部进行处理。接着对 context 子节点下的 transactionManager 孙节点和 Dao 孙节点进行循环读取。每次读取时, 同时处理相关信息并形成相关的对象。

第 3 步: 由 transactionManager 孙节点创建实现基于接口 DaoTransactionManager 的实例化对象。Dao 孙节点形成 DaoImpl 对象和 DaoProxy 对象。

第 4 步: 把基于接口 DaoTransactionManager 的实例化对象、DaoImpl 对象和 DaoProxy 对象都保存在 DaoContext 对象中, 然后把所有的 DaoContext 对象保存在 StandardDaoManager 对象中。这样就完成了配置文件的读取。

由于在这里会涉及 StandardDaoManager 对象、DaoContext 对象、transactionManager 对象、Dao 接口 Class 对象、DaoImpl 对象等。其类结构如图 4-10 所示。

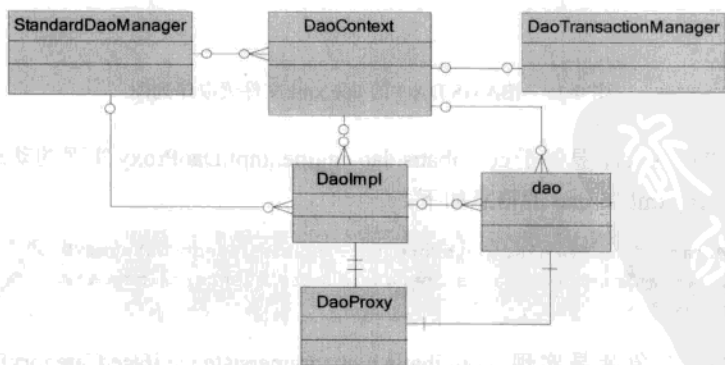


图 4-10 iBATIS DAO 的核心接口和类的类图结构

序列图表明都是对象，其对象对应的类如表 4-2 所示。

表 4-2 对象与对应类的映射列表

序 号	对象名称	对应的类和接口	备 注
1	Builder	com.ibatis.dao.engine.builder.xml.XmlDaoManagerBuilder	
2	DaoManager	com.ibatis.dao.engine.impl.StandardDaoManager	
3	DaoContext	com.ibatis.dao.engine.impl.DaoContext	
4	TransactionManager	com.ibatis.dao.engine.transaction.DaoTransactionManager	根据配置信息针对不同的类
5	DaoImpl	com.ibatis.dao.engine.impl.DaoImpl	
6	DaoProxy	com.ibatis.dao.engine.impl.DaoProxy	
7	*dao	com.ibatis.dao.engine.impl.DaoProxy	根据配置针对不同 dao 接口的代理类

备注说明：对于编号 4 中的 TransactionManager 对象是实现 com.ibatis.dao.engine.transaction.DaoTransactionManager 的实例化对象。针对的实现类可根据 dao.xml 配置信息的不同而不同，具体如图 4-11 所示。

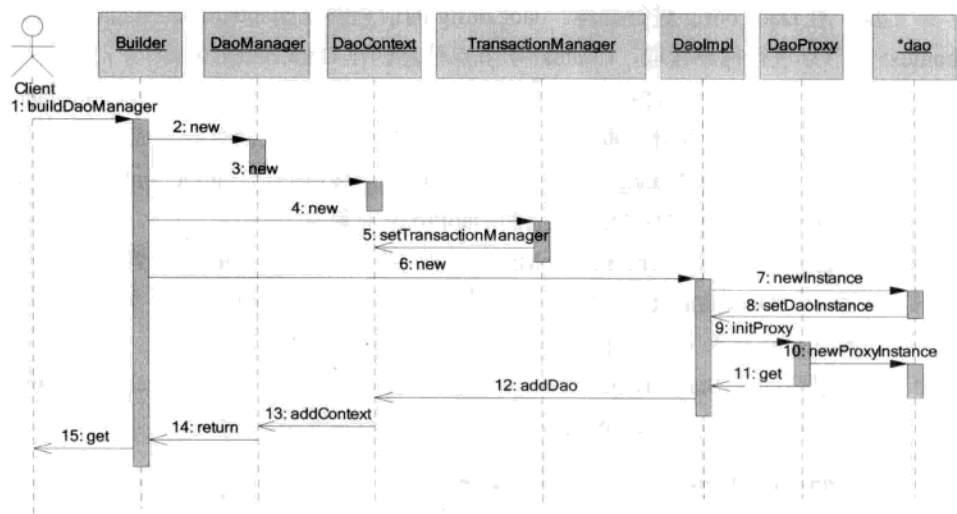


图 4-11 iBATIS DAO 的 dao.xml 文件读取序列图

对于编号 7 中的 *dao 是基于 com.ibatis.dao.engine.impl.DaoProxy 实现的动态代理对象。例如配置文件 dao.xml 中 dao 的信息如下。

```
<dao interface="com.ibatis.jpetstore.persistence.iface.CategoryDao"
      implementation="com.ibatis.jpetstore.persistence.sqlmapdao.
      CategorySqlMapDao"/>
```

那么 DAO 对象就是实现 com.ibatis.jpetstore.persistence.iface.CategoryDao 接口的 com.ibatis.dao.engine.impl.DaoProxy 对象。

这样，我们理解代码就可从最基本的 daoConfig 开始，接着进入 context 节点处理，最后对 transactionManager 节点和 DAO 节点进行读取处理。

上面的步骤解释如下。

第 1 步骤：客户端调用 XmlDaoManagerBuilder 对象的 buildDaoManager 方法。

第 2 步骤：XmlDaoManagerBuilder 对象创建一个 StandardDaoManager 对象。

第 3 步骤：读取配置文件中 daoConfig 节点下 context 的个数，XmlDaoManagerBuilder 对象循环创建 1 个或多个 DaoContext 对象。

第 4 步骤：根据配置文件中 context 节点下 transactionManager 的个数，XmlDaoManagerBuilder 对象创建 TransactionManager 对象。

第 5 步骤：把 TransactionManager 对象赋值给 DaoContext 对象。

第 6 步骤：根据配置文件中 context 节点下 DAO 的个数，XmlDaoManagerBuilder 对象实例化多个 daoImpl 对象。

第 7 步骤：每个 daoImpl 实例化对象根据获得的 dao 节点中的 implementation 属性，采用 constructor.newInstance 方法动态实例化 dao 对象。

第 8 步骤：把动态的 DAO 通过 setDaoInstance 方法赋值给 daoImpl 对象。

第 9 步骤：调用 daoImpl 对象的 initProxy()方法，调用 DaoProxy 对象的静态方法 newInstance(this)。

第 10 步骤：通过系统的 Proxy.newProxyInstance 调用，实现动态代理类的生成。动态代理类映射的就是在第 7 步骤创建的 dao 实例化对象。

第 11 步骤：把动态代理类赋值给 daoImpl 对象。

第 12 步骤：把所有的 daoImpl 对象通过循环加入到 DaoContext 对象的 Map 变量中。

第 13 步骤：把所有的 DaoContext 对象通过循环加入到 StandardDaoManager 对象的 Map 变量中。

第 14 步骤：XmlDaoManagerBuilder 对象获得 StandardDaoManager 对象。

第 15 步骤：客户端获得 StandardDaoManager 对象。

这些程序代码详细解释如下。

2. 对 daoConfig 节点读取

对 daoConfig 节点读取的代码代码如下（已经增加了注解）。

```
public DaoManager buildDaoManager(Reader reader) throws DaoException {
    //创建一个 StandardDaoManager 实例化对象
    StandardDaoManager daoManager = new StandardDaoManager();

    try {
        //装载配置文件并获得根节点
        Document doc = getDoc(reader);
        Element root = (Element) doc.getLastChild();

        String rootname = root.getNodeName();
```

```

        if (!DAO_CONFIG_ELEMENT.equals(rootname)) {
            throw new IOException("Error while configuring DaoManager. The root tag of
the DAO configuration XML " +
                "document must be '" + DAO_CONFIG_ELEMENT + "'");
        }
        //对根节点下的第一层子节点进行循环读取
        NodeList children = root.getChildNodes();
        for (int i = 0; i < children.getLength(); i++) {
            Node child = children.item(i);
            if (child.getNodeType() == Node.ELEMENT_NODE) {
                if (CONTEXT_ELEMENT.equals(child.getNodeName())) {
                    //解析 context 子节点的信息
                    DaoContext daoContext = parseContext((Element) child, daoManager);
                    daoManager.addContext(daoContext);
                } else if (PROPERTIES_ELEMENT.equals(child.getNodeName())) {
                    //解析 Properties 子节点的信息
                    Properties attributes = parseAttributes(child);
                    if (attributes.containsKey("resource")) {
                        String resource = attributes.getProperty("resource");
                        if (properties == null) {
                            properties = Resources.getResourceAsProperties(resource);
                        } else {
                            Properties tempProps = Resources.getResourceAsProperties(resource);
                            tempProps.putAll(properties);
                            properties = tempProps;
                        }
                    } else if (attributes.containsKey("url")) {
                        String url = attributes.getProperty("url");
                        if (properties == null) {
                            properties = Resources.getUrlAsProperties(url);
                        } else {
                            Properties tempProps = Resources.getUrlAsProperties(url);
                            tempProps.putAll(properties);
                            properties = tempProps;
                        }
                    }
                }
            }
        }
    } catch (Exception e) {
        throw new DaoException("Error while configuring DaoManager. Cause: " +
            e.toString(), e);
    }
    return daoManager;
}

```

3. 对 DaoContext 节点读取

对 DaoContext 节点读取的代码代码如下（已经增加了注解）。

```
private DaoContext parseContext(Element contextElement, StandardDaoManager
```

```

daoManager) throws DaoException {
    //创建一个新的 DaoContext 对象
    DaoContext daoContext = new DaoContext();

    daoContext.setDaoManager(daoManager);
    String id = contextElement.getAttribute("id");
    if (id != null && id.length() > 0) {
        daoContext.setId(id);
    }

    NodeList children = contextElement.getChildNodes();
    //对 DaoContext 子节点开始读取
    for (int i = 0; i < children.getLength(); i++) {
        Node child = children.item(i);
        if (child.getNodeType() == Node.ELEMENT_NODE) {
            if (TRANS_MGR_ELEMENT.equals(child.getNodeName())) {
                //获取 DaoTransactionManager 实例化对象并加到 DaoContext 对象中
                DaoTransactionManager txMgr = parseTransactionManager((Element) child);
                daoContext.setTransactionManager(txMgr);
            } else if (DAO_ELEMENT.equals(child.getNodeName())) {
                //获取 DaoImpl 实例化对象并加到 DaoContext 对象中
                DaoImpl daoImpl = parseDao((Element) child, daoManager, daoContext);
                daoContext.addDao(daoImpl);
            }
        }
    }
    return daoContext;
}

```

4. 对 transactionManager 节点读取

对 transactionManager 节点读取的代码代码如下（已经增加了注解）。

```

private DaoTransactionManager parseTransactionManager(Element transPoolElement)
throws DaoException {
    DaoTransactionManager txMgr = null;

    Properties attributes = parseAttributes(transPoolElement);
    //获取 transactionManager 节点的 type 内容
    String implementation = attributes.getProperty("type");
    //根据 type 内容获取全名, 如 SQLMAP 的全名是 SqlMapDaoTransactionManager.class.getName()
    //HIBERNATE 的全名是 HibernateDaoTransactionManager.class.getName()
    implementation = resolveAlias(implementation);

    try {
        //对类 com.ibatis.dao.engine.transaction.sqlmap.SqlMapDaoTransactionManager 进
        //行实例化
        txMgr = (DaoTransactionManager) Resources.classForName(implementation).
        newInstance();
    } catch (Exception e) {
        throw new DaoException("Error while configuring DaoManager. Cause: " +

```



```
e.toString(), e);
    }

    Properties props = properties;
    if (props == null) {
        props = parsePropertyElements(transPoolElement);
    } else {
        props.putAll(parsePropertyElements(transPoolElement));
    }

    // 对 SqlMapDaoTransactionManager 实例化对象赋初值，初值的信息由 transactionManager 节点信息获取
    txMgr.configure(props);

    if (txMgr == null) {
        throw new DaoException("Error while configuring DaoManager. Some unknown condition caused the " +
            "DAO Transaction Manager to be null after configuration.");
    }
    return txMgr;
}
```

在上述代码中，有一句创建 DaoTransactionManager 的实例化对象。

```
txMgr = (DaoTransactionManager) Resources.className(implementation).newInstance();
```

这个实例化对象可根据不同的配置信息生成不同的事务处理管理器。分别为 JDBC、Apache Ojb、Toplink、JTA、SQL Map、Hibernate 和 EXTERNAL。在 XmlDaoManagerBuilder 类的构造就赋了初值，代码如下。

```
public XmlDaoManagerBuilder() {
    typeAliases.put("EXTERNAL", ExternalDaoTransactionManager.class.getName());
    typeAliases.put("HIBERNATE", HibernateDaoTransactionManager.class.getName());
    typeAliases.put("JDBC", JdbcDaoTransactionManager.class.getName());
    typeAliases.put("JTA", JtaDaoTransactionManager.class.getName());
    typeAliases.put("APACHE_OJB", "com.ibatis.dao.engine.transaction.Apache Ojb. OjbBrokerTransactionManager");
    typeAliases.put("SQLMAP", SqlMapDaoTransactionManager.class.getName());
    typeAliases.put("TOPLINK", "com.ibatis.dao.engine.transaction.toplink. Toplink DaoTransactionManager");
}
```

表 4-3 说明各个配置所产生的事务管理器实例化对象。

表 4-3 默认启动别名与实现类的对应关系

序 号	配置信息	实例化对象
1	JDBC	com.ibatis.dao.engine.transaction.jdbc.JdbcDaoTransactionManager
2	Apache Ojb	com.ibatis.dao.engine.transaction.Apache Ojb.OjbBrokerTransactionManager

续表

序 号	配置信息	实例化对象
3	Toplink	com.ibatis.dao.engine.transaction.toplink.ToplinkDaoTransactionManager
4	JTA	com.ibatis.dao.engine.transaction.jta.JtaDaoTransactionManager
5	SQL Map	com.ibatis.dao.engine.transaction.sqlmap.SqlMapDaoTransactionManager
6	Hibernate	com.ibatis.dao.engine.transaction.hibernate.HibernateDaoTransactionManager
7	EXTERNAL	com.ibatis.dao.engine.transaction.external.ExternalDaoTransactionManager

对于 JDBC、JTA 和 EXTERNAL 的事务实例化说明与 SQL Map 的事务实例化是基本相同的, 将在“第 8 章 SQL Map 数据库处理”章节中具体加以说明。

对于 Apache Ojb、Toplink 和 Hibernate 的事务实例化说明将在“dao 事务管理实现”节中说明。在这里只介绍 SqlMapDaoTransactionManager 实例化对象赋初值。

```
txMgr.configure(props);
```

这个初值是如何进入的呢?

```
public void configure(Properties properties) {
    try {
        Reader reader = null;
        if (properties.containsKey("SqlMapConfigURL")) {
            reader = Resources.getUrlAsReader((String) properties.get("SqlMapConfigURL"));
        } else if (properties.containsKey("SqlMapConfigResource")) {
            reader = Resources.getResourceAsReader((String) properties.get("SqlMapConfigResource"));
        } else {
            throw ....;
        }
        //创建出 Client 客户端
        client = SqlMapClientBuilder.buildSqlMapClient(reader, properties);
    } catch (IOException e) {
        throw ...;
    }
}
```

在这里有一行代码 `client = SqlMapClientBuilder.buildSqlMapClient(reader, properties)`。实际上, 程序就已经转入到了 SQLMap 的调用。

5. dao 节点读取

对 dao 节点读取的代码如下 (已经增加了注解)。

```
private DaoImpl parseDao(Element element, StandardDaoManager daoManager, DaoContext daoContext) {
    //新实例化一个 DaoImpl 对象
    DaoImpl daoImpl = new DaoImpl();
    if (element.getNodeType() == Node.ELEMENT_NODE) {
        if (DAO_ELEMENT.equals(element.getNodeName())) {
```

```

        //获取 dao 节点下面的属性
        Properties attributes = parseAttributes(element);

        try {
            //获取 dao 节点的 interface 内容
            String iface = attributes.getProperty("interface");
            //获取 dao 节点的 implementation 内容
            String impl = attributes.getProperty("implementation");
            daoImpl.setDaoManager(daoManager);
            daoImpl.setDaoContext(daoContext);
            //根据 dao 节点 interface 内容创建一个 interface 的接口
            daoImpl.setDaoInterface(Resources.classForName(iface));
            //根据 dao 节点 implementation 内容创建一个 implementation 的类
            daoImpl.setDaoImplementation(Resources.classForName(impl));

            Class daoClass = daoImpl.getDaoImplementation();
            Dao dao = null;

            try {
                //根据创建的 implementation 的类, 采用构造方法实例化, 形成 implementation 类的实例
                Constructor constructor = daoClass.getConstructor(new Class[]
                {DaoManager.class});
                dao = (Dao) constructor.newInstance(new Object[]{daoManager});
            } catch (Exception e) {
                //根据创建的 implementation 的类, 直接形成 implementation 类的实例化对象
                dao = (Dao) daoClass.newInstance();
            }

            daoImpl.setDaoInstance(dao);
            //创建 dao 接口的代理对象
            daoImpl.initProxy();
        } catch (Exception e) {
            throw new DaoException("Error configuring DAO. Cause: " + e, e);
        }
    }
    return daoImpl;
}

```

上面的语句中有一句创建 dao 接口的代理对象, 即如下代码要详细说明一下。

```
daoImpl.initProxy();
```

我们看看其实现代码, 在 DaoImpl 类中实现、转移到 DaoProxy 类的静态方法。

```

public void initProxy() {
    proxy = DaoProxy.newInstance(this);
}

```

DaoProxy 类是一个实现 InvocationHandler 接口的代理类, 我们可以看一下 newInstance

这个静态方法的实现过程。

```
public static Dao newInstance(DaoImpl daoImpl) {
    return (Dao) Proxy.newProxyInstance(daoImpl.getDaoInterface().getClassLoader(),
        new Class[] { Dao.class, daoImpl.getDaoInterface() },
        new DaoProxy(daoImpl));
}
```

在这个静态方法中，采用动态代理类方式，即实现创建类时在运行中指定的接口列表的类。Proxy.newProxyInstance 方法的第一个参数表示加载的类加载器，在这里是得到配置信息中 dao 的 interface 接口的类加载器。第二个参数是要实现的接口数组，包括 dao 接口和配置信息中 dao 的 interface 接口。第三个参数是实现的代理类实例化对象，也就是 DaoProxy 类本身的实例化对象。

实际上如下写的程序代码就比较清晰了。

```
public static Dao newInstance(DaoImpl daoImpl) {
    InvocationHandler handler = new DaoProxy (daoImpl);
    Class proxyClass = Proxy.getProxyClass(
        daoImpl.getDaoInterface().getClassLoader(), new Class[] { Dao.class,
        daoImpl.getDaoInterface() });
    Dao d = (Dao) proxyClass.getConstructor(new Class[] { InvocationHandler.
    class }).
        newInstance(new Object[] { handler });
}
```

所以，在一个新建的 DaoImpl 实例化对象中，有以下几个私有变量。

```
private StandardDaoManager daoManager;
private DaoContext daoContext;
private Class daoInterface;
private Class daoImplementation;
private Dao daoInstance;
private Dao proxy;
```

对这些变量进行分析说明如下。

变量 daoManager 是 StandardDaoManager 的实例化对象，即由 dao.xml 中本 dao 节点的父亲节点 Context 上的父亲节点 daoConfig 创建的 StandardDaoManager 对象。

变量 daoContext 是 DaoContext 的实例化对象，即由 dao.xml 中本 dao 节点的父亲节点 Context 创建的 DaoContext 对象。

变量 daoInterface 是 Class 变量，即由 dao.xml 中本 dao 节点的 interface 属性的接口。

变量 daoImplementation 是 Class 变量，即由 dao.xml 中本 dao 节点的 implementation 属性的类。

变量 daoInstance 是 Dao 实例化对象，即由 dao.xml 中本 dao 节点的 implementation 属性的实例化对象。

变量 proxy 是 Dao 实例化对象，即由 dao.xml 中本 dao 节点的 interface 属性的动态代

理对象。

这样，就可以响应外部的事件，进行业务的处理。

4.3.3 如何验证 dao.xml 文件

XML 在不同的语言里解析方式都是一样的，只不过实现的语法不同而已。基本的解析方式有两种：一种叫 SAX，另一种叫 DOM。SAX 是基于事件流的解析，DOM 是基于 XML 文档树结构的解析。在对 dao.xml 的解析过程中，采用了两种解析方式相结合的实现模式。

对于 dao.xml 是否遵循正确的格式，即是否拥有正确语法的 XML，这一点需要进行验证。而验证的工具就是 dao.xml 的 DTD 文件。DTD 文件验证时会包括一些常用的 XML 语法规则，如 XML 文档必须有根元素、XML 文档必须有关闭标签、XML 标签对大小写敏感、XML 元素内容是否正确、XML 元素必须被正确的嵌套、XML 属性必须加引号，等等。

dao.xml 包含对应的 DTD 声明，即 dao-2.dtd。该 dtd 文件用来定义该 XML 文档中的格式。dao.xml 中用到的 DTD 信息如下。

```
<!DOCTYPE daoConfig
PUBLIC "-//ibatis.apache.org//DTD DAO Configuration 2.0//EN"
"http://ibatis.apache.org/dtd/dao-2.dtd">
```

当开发人员定义了 dao.xml 文件时，但不清楚这个文件是否满足上述的 dtd，这需要进行验证。验证的目的方面要保证应用程序读取的 dao.xml 文件是符合既定的文件格式，这样才能获得正确的配置信息；另一方面，对于文件格式的错误，也可以进行异常处理，让二次开发人员能迅速找到错误的位置并能正确改正。

一般而言，dtd 文件定义格式都放到互联网上。当我们解析这个 XML 的时候，如果当前计算机已联网，那么解析的速度比较慢，因为它要到互联网上去寻找定义的 dtd 文件。如果当前计算机未联网，则会报无法连接主机的异常。因此问题就在于：如果我们的计算机不能保证时时都连在网上，那么怎么保证解析过程中不出错呢？

iBATIS 首先自定义了 DaoClasspathEntityResolver 解析器类，DaoClasspathEntityResolver 类是实现 org.XML.sax.EntityResolver 接口的实现类。在 DaoClasspathEntityResolver 类有一个 static final Map 属性，其保存内容如下（该技术是采用 SAX 来实现的）。

```
private static final String DTD_PATH_DAO = "com/ibatis/dao/engine/builder/xml/dao-2.dtd";

static {
    doctypeMap.put("http://www.ibatis.com/dtd/dao-2.dtd", DTD_PATH_DAO);
    doctypeMap.put("http://ibatis.apache.org/dtd/dao-2.dtd", DTD_PATH_DAO);
    doctypeMap.put("-//IBATIS.com//DTD DAO Configuration 2.0", DTD_PATH_DAO);
    doctypeMap.put("-//IBATIS.com//DTD DAO Config 2.0", DTD_PATH_DAO);
}
```

把所有的关于 dtd 的信息都转化为本地目录的 dao-2.dtd 文件。而 dao-2.dtd 文件格式（见附录一）即是 dao.xml 的 XML 文件格式。该类的目的就是把本来从 <http://www.ibatis.com/dtd/dao-2.dtd>、<http://ibatis.apache.org/dtd/dao-2.dtd>、<http://ibatis.com/DTD> DAO Configuration 2.0、<http://ibatis.com/DTD> DAO Config 2.0 等地址中读取 dtd 的信息改为了在当前类路径中读取，也就是说，[com.ibatis.dao.engine.builder.XML/dao-2.dtd](http://ibatis.com/DTD)，通过重新定义 `InputSource` 来返回 dao-2.dtd 数据流，从而让 XML 解析器不从网上获取 DTD 信息。当然，这样做的前提是事先必须把 dao-2.dtd 文件保存在类路径所在的目录中，以便自定义的 `EntityResolver` 可以读取到。

对于 `DaoClasspathEntityResolver` 类必须实现方法 `resolveEntity(String publicId, String systemId)`。`iBATIS DAO` 的实现代码如下。

```
public InputSource resolveEntity(String publicId, String systemId) throws
SAXException {
    InputSource source = null;

    try {
        String path = (String) doctypeMap.get(publicId);
        source = getInputSource(path, source);
        if (source == null) {
            path = (String) doctypeMap.get(systemId);
            source = getInputSource(path, source);
        }
    } catch (Exception e) {
        throw new SAXException(e.toString());
    }

    return source;
}
```

其作用是转化公共的 DTD 为本地的 DTD。

```
private InputSource getInputSource(String path, InputSource source) {
    if (path != null) {
        InputStream in = null;
        try {
            in = Resources.getResourceAsStream(path);
            source = new InputSource(in);
        } catch (IOException e) {
            ...
        }
    }
    return source;
}
```

采用 DOM 来解析 dao.xml 文件的程序代码：

```
private Document getDoc(Reader reader) {
    try {
```



```

// Configuration
DocumentBuilderFactory dbf = DocumentBuilderFactory.newInstance();
//指定由此代码生成的解析器将提供对 XML 名称空间的支持。
dbf.setNamespaceAware(false);
//开启验证属性。
dbf.setValidating(true);
//指定由此代码生成的解析器将忽略注释。
dbf.setIgnoringComments(true);
//指定由此工厂创建的解析器在解析 XML 文档时，必须删除元素内容中的空格
dbf.setIgnoringElementContentWhitespace(true);
//指定由此代码生成的解析器将把 CDATA 节点转换为 Text 节点，并将其附加到相邻的 Text 节点。
dbf.setCoalescing(false);
//指定由此代码产生的解析器将扩展实体引用节点。
dbf.setExpandEntityReferences(false);

OutputStreamWriter errorWriter = new OutputStreamWriter(System.err);

DocumentBuilder db = dbf.newDocumentBuilder();
//注册错误事件处理程序。
db.setErrorHandler(new SimpleErrorHandler(new PrintWriter(errorWriter, true)));
//注册自定义的实体解析器。
db.setEntityResolver(new DaoClasspathEntityResolver());

// Parse input file
//解析 XML 文档
Document doc = db.parse(new ReaderInputStream(reader));
return doc;
} catch (Exception e) {
    throw new RuntimeException("XML Parser Error. Cause: " + e);
}
}

```

由于在解析 XML 文件时，指定了要解析的命名空间支持，这些支持也是由本地的 dtd 文件组成的，通过注册自定义的实体解析器，可以按照规定的格式来进行处理。当出现解析问题时，就可以获得错误文件格式的异常。

4.3.4 dao.xml 配置文件实例说明

1. dao.xml 内容

dao.xml 配置文件如下：

```

<?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>
        <transactionManager type="SQLMAP">
            <property name="SqlMapConfigResource"

```



```

        value="com.ibatis.jpetstore.persistence.sqlmapdao/sql/sql-map-config.xml"/>
    </transactionManager>
    <dao interface="com.ibatis.jpetstore.persistence.iface.ItemDao"
        implementation="com.ibatis.jpetstore.persistence.sqlmapdao.ItemSqlMapDao"/>
    <dao interface="com.ibatis.jpetstore.persistence.iface.SequenceDao"
        implementation="com.ibatis.jpetstore.persistence.sqlmapdao.
SequenceSqlMapDao"/>
    <dao interface="com.ibatis.jpetstore.persistence.iface.AccountDao"
        implementation="com.ibatis.jpetstore.persistence.sqlmapdao.
AccountSqlMapDao"/>
    <dao interface="com.ibatis.jpetstore.persistence.iface.CategoryDao"
        implementation="com.ibatis.jpetstore.persistence.sqlmapdao.
CategorySqlMapDao"/>
    <dao interface="com.ibatis.jpetstore.persistence.iface.ProductDao"
        implementation="com.ibatis.jpetstore.persistence.sqlmapdao.
ProductSqlMapDao"/>
    <dao interface="com.ibatis.jpetstore.persistence.iface.OrderDao"
        implementation="com.ibatis.jpetstore.persistence.sqlmapdao.
OrderSqlMapDao"/>
</context>
</daoConfig>

```

在这个 dao.xml 中有一个 daoConfig 节点, 在 daoConfig 根节点下有一个 context 节点。在 context 节点下有一个 transactionManager 和 6 个 dao 节点。

在这个配置文件下, 在服务器端形成对应的内存对象结构。

1. 总体说明

解析生成 1 个 StandardDaoManager 实例化对象, 1 个 DaoContext 实例化对象, 1 个 DaoTransactionManager 实例化对象, 6 个 DaoImpl 实例化对象和 6 个 DaoProxy 实例化对象。

2. StandardDaoManager 实例化对象说明

1 个 StandardDaoManager 实例化对象的内部私有变量有两个 HashMap: typeContextMap 和 daoImplMap, 其初始化程序代码如下。

```

public void addContext(DaoContext context) {
    // Add context ID mapping, 在本例中没有执行
    if (context.getId() != null && context.getId().length() > 0) {
        if (idContextMap.containsKey(context.getId())) {
            throw new DaoException("There is already a DAO Context with the ID '" +
context.getId() + "'.");
        }
        idContextMap.put(context.getId(), context);
    }
    //增加类型 context 映射
    Iterator i = context.getDaoImpls();
    while (i.hasNext()) {
        DaoImpl daoImpl = (DaoImpl) i.next();
        // Don't associate a default DAO when multiple DAO impls are registered.
        if (typeContextMap.containsKey(daoImpl.getDaoInterface())) {

```

```

        typeContextMap.put(daoImpl.getDaoInterface(), null);
    } else {
        typeContextMap.put(daoImpl.getDaoInterface(), context);
    }
    daoImplMap.put(daoImpl.getProxy(), daoImpl);
    daoImplMap.put(daoImpl.getDaoInstance(), daoImpl);
}
}

```

变量 `typeContextMap` 存储了 6 个数据，分别是配置文件 `dao` 节点的 `interface` 内容形成的接口和当前的 `DaoContext` 实例化对象。`daoImplMap` 存储了 12 个数据，其中 6 个是配置文件 `dao` 节点的 `interface` 属性内容形成的实现 `dao` 接口的 `DaoProxy` 动态代理对象和对应的 `DaoImpl` 实例化对象，另外 6 个是配置文件 `dao` 节点的 `implementation` 属性内容形成的实现 `dao` 接口的实例化对象和对应的 `DaoImpl` 实例化对象。

3. DaoContext 实例化对象说明

`DaoContext` 实例化对象的内部私有变量有 1 个 `StandardDaoManager` 变量 `daoManager`、1 个 `DaoTransactionManager` 变量 `transactionManager` 和 1 个 `HashMap` 变量 `typeDaoImplMap`。

`StandardDaoManager` 变量就是上述的 `StandardDaoManager` 实例化对象。`transactionManager` 变量就是 `dao.xml` 配置文件中 `transactionManager` 节点形成的 `SqlMapDaoTransactionManager` 实例化对象。变量 `typeDaoImplMap` 存储了 6 个数据，分别是配置文件 `dao` 节点的 `interface` 内容形成的接口和对应的 `DaoImpl` 实例化对象。

4. DaoTransactionManager 实例化对象

`DaoTransactionManager` 实例化对象就是 `dao.xml` 配置文件中 `transactionManager` 节点形成的 `SqlMapDaoTransactionManager` 实例化对象。其私有变量只有 1 个 `SqlMapClient` 变量 `client`。这个 `SqlMapClient` 变量非常重要，是 `dao` 结合 `SqlMap` 的接口（其内容在下面会有详细的介绍）。

5. DaoImpl 实例化对象

`DaoImpl` 实例化对象就是在 `dao.xml` 配置文件中每个 `dao` 节点都形成一个针对本节点的 `DaoImpl` 实例化对象。每个 `DaoImpl` 实例化对象有 1 个 `StandardDaoManager` 变量 `daoManager`、1 个 `DaoContext` 变量 `daoContext`、1 个 `Class` 变量 `daoInterface`、1 个 `Class` 变量 `daoImplementation`、1 个 `Dao` 变量 `daoInstance` 和 1 个 `Dao` 变量 `proxy`。

`StandardDaoManager` 变量就是上述的 `StandardDaoManager` 实例化对象，`DaoContext` 变量就是上述的 `DaoContext` 实例化对象。`Class` 类型 `daoInterface` 变量是配置文件 `dao` 节点的 `interface` 内容形成的接口。`Class` 类型 `daoImplementation` 变量是配置文件 `dao` 节点的 `implementation` 内容形成的类。基于 `Dao` 接口的实例化变量 `daoInstance` 是配置文件 `dao` 节点的 `implementation` 内容形成的实例化对象。基于 `Dao` 接口的实例化变量 `proxy` 是基于配置文件 `dao` 节点的 `interface` 属性内容形成的接口并通过 `DaoProxy` 动态代理类实现的实例化对象。

6. DaoProxy 实例化对象

DaoProxy 实例化对象就是基于配置文件 dao 节点的 interface 属性内容形成的接口并通过 DaoProxy 动态代理类实现的实例化对象。在每个 DaoProxy 实例化对象有 1 个 DaoImpl 变量 daoImpl。

4.4 iBATIS DAO 引擎实现

dao 引擎实现包括 com.ibatis.dao.engine.impl 包和 com.ibatis.dao.engine.transaction 包下的所有程序文件。在这里要涉及两个 Manager，一个是 Dao 的 Manager，还有一个是 Transaction 的 Manager。顾名思义，StandardDaoManager 主要负责管理 Dao 对象接口及其相关的实现，而 TransactionManager 是管理 Transaction 对象接口及其相关的实现。

4.4.1 DAO 业务实现的序列图和说明

这个实现过程有点扑朔迷离。但是只要掌握了基本框架和实现过程，这些都还是很容易理解的。所用的 Dao 接口操作，根据配置文件对应的实现类信息转化为相应的 Dao 实现类对象。Dao 实现类由于通过代理 DaoProxy 来实现，所以对象的操作要转移到 DaoProxy 的 invoke 方法。DaoProxy 对象并不直接进行业务处理，而是调用 DaoManager 容器来进行处理；DaoManager 容器获得当前 Dao 的 DaoContext，然后通过 DaoContext 的事务处理方法来进行业务处理。DaoContext 的事务处理方法也是继续转移给下层的事务管理器 DaoTransactionManager 来实现的，DaoTransactionManager 对象也是交给不同的 DaoTransaction 实例化对象来进行 Dao 的最后事务操作。Dao 框架的序列如图 4-12 所示（为了便于说明，主要包括一些核心类和对象）。

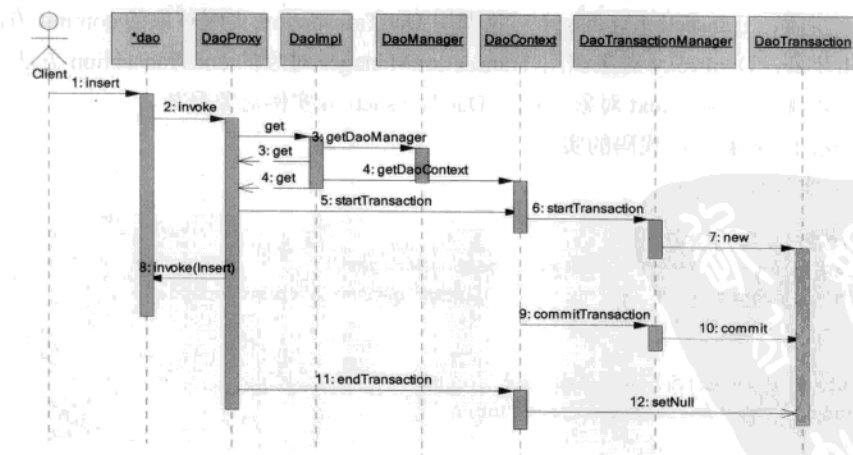


图 4-12 iBATIS DAO 业务实现的序列图

这些对象的实现类说明如下表 4-4 所示。

表 4-4 iBATIS DAO 业务实现序列图中对象与实际类对应说明

序 号	对象名称	对应的类和接口	备 注
1	*dao	com.ibatis.dao.engine.impl.DaoProxy	根据配置针对 不同 dao 接口 的代理类
2	DaoProxy	com.ibatis.dao.engine.impl.DaoProxy	
3	DaoImpl	com.ibatis.dao.engine.impl.DaoImpl	
4	DaoManager	com.ibatis.dao.engine.impl.StandardDaoManager	
5	DaoContext	com.ibatis.dao.engine.impl.DaoContext	
6	TransactionManager	com.ibatis.dao.engine.transaction.sqlmap.SqlMapDaoTransactionManager	
7	SqlMapClient	com.ibatis.sqlmap.client.SqlMapClient	
8	DaoTransaction	com.ibatis.dao.engine.transaction.sqlmap.SqlMapDaoTransaction	

图 4-12 的实现步骤解释如下：

第 1 步骤：客户端调用某个 DAO 对象的 Insert 方法。

第 2 步骤：由于 DAO 对象是继承 DAO 接口，其转移到了 DaoProxy 对象的 invoke 方法。

第 3 步骤：DaoProxy 对象的 invoke 调用 DaoImpl 对象，并从 DaoImpl 对象中获得 DaoManager 实例对象。

第 4 步骤：DaoImpl 对象从 DaoManager 实例对象中获得 DaoContext 对象。

第 5 步骤：DaoImpl 对象调用 DaoContext 对象的 startTransaction 方法。

第 6 步骤：DaoContext 对象调用 TransactionManager 对象的 startTransaction 方法。

第 7 步骤：TransactionManager 对象创建一个 DaoTransaction 实例对象。

第 8 步骤：DAO 对象调用 invoke(Insert)方法。

第 9 步骤：DaoContext 对象调用 TransactionManager 对象的 commitTransaction 方法。

第 10 步骤：TransactionManager 对象调用 DaoTransaction 实例对象的 commit 方法。

第 11 步骤：DaoProxy 对象调用 TransactionManager 对象的 endTransaction 方法。

第 12 步骤：DaoContext 对象直接把 DaoTransaction 实例对象释放。

下面我们来看看程序代码的实现。

Dao 的实现代码如下：

```
public AccountService() {
    DaoManager daoMgr = DaoConfig.getDaoManager();
    this.accountDao = (AccountDao) daoMgr.getDao(AccountDao.class);
}

public void insertAccount(Account account) {
    accountDao.insertAccount(account);
}
```

从配置里面我们知道，daoMgr.getDao(AccountDao.class)是返回实现 AccountDao 接口

的 DaoProxy 动态代理对象。当 accountDao 调用 insertAccount 方法时，实际上是转到了 DaoProxy 动态代理对象的 invoke 方法上。DaoProxy 的 invoke 实现代码如下：

```
public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {
    Object result = null;
    //这是处理一些常规的方法，如 equals、getClass 等方法。
    if (PASSTHROUGH_METHODS.contains(method.getName())) {
        try {
            result = method.invoke(daoImpl.getDaoInstance(), args);
        } catch (Throwable t) {
            throw ClassInfo.unwrapThrowable(t);
        }
    } else {
        //获得 StandardDaoManager 实例化对象
        StandardDaoManager daoManager = daoImpl.getDaoManager();
        DaoContext context = daoImpl.getDaoContext();

        if (daoManager.isExplicitTransaction()) {
            // Just start the transaction (explicit)
            //针对调用了 daoManager 的 startTransaction 方法的，不能采取一次性提交，只是启动事务
            try {
                context.startTransaction();
                result = method.invoke(daoImpl.getDaoInstance(), args);
            } catch (Throwable t) {
                throw ClassInfo.unwrapThrowable(t);
            }
        } else {
            //针对单个的 dao，一次性进行启动事务，提交事务和结束事务
            try {
                context.startTransaction();
                result = method.invoke(daoImpl.getDaoInstance(), args);
                context.commitTransaction();
            } catch (Throwable t) {
                throw ClassInfo.unwrapThrowable(t);
            } finally {
                context.endTransaction();
            }
        }
    }
    return result;
}
```

操作在业务 Dao 对象、DaoProxy 动态代理对象、DaoImpl 对象、StandardDaoManager 对象、DaoContext 对象，SqlMapClient 实例化对象等之间进行传递。就像一个接力棒，一棒一棒往下传，最后交到最后一个对象。这个对象就是 SqlMapClient 实例化对象。无论是事务的启动、对象的修改和事务的提交，最后都落实到 SqlMapClient 接口的实例化对象上来实现。

SqlMapClient 接口继承 SqlMapExecutor 接口，所以返回 SqlMapExecutor 也就是返回了 SqlMapClient 的实例化对象。

其中在 DaoManager 对象上和 DaoContext 对象上进行了线程安全性的处理, 保证操作都基于一个线程来实现。在 DaoManager 对象中, 启用了两种不同作用的线程, 一种是针对 Context 对象, 在不同的线程开启不同的 DaoContext 对象; 还有一种是用于事务管理的, 即是否开启事务。

而 DaoContext 对象的线程主要是处理不同的事务状态。也启用了 2 种不同作用的线程, 一种是针对 DaoTransaction 对象, 还有一种是判断 DaoTransaction 对象的事务状态。

4.4.2 iBATIS DAO 组件管理

1. DAO Impl 的类图结构和说明

StandardDaoManager 是调度中心, 把配置信息和运行信息结合起来。一个 StandardDaoManager 实例的对象包含多个 DaoContext 实例的对象, 而一个 DaoContext 实例又包括一个基于 DaoTransactionManager 接口实例的 TransactionManager 对象和多个 DaoImpl 对象。其结构类图如图 4-13 所示。

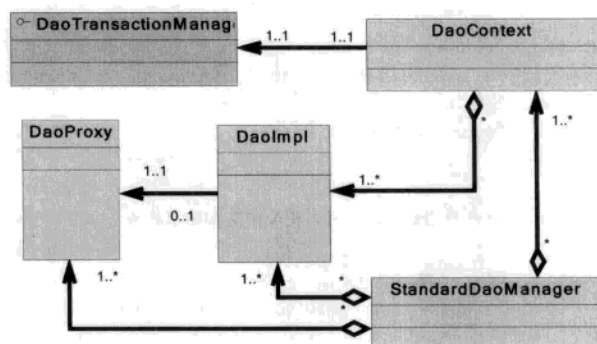


图 4-13 dao 引擎的类结构图

要理解 iBATIS DAO 的引擎实现, 必须了解这四个类及其变量, 同时还要了解 DaoTransactionManager 接口及其实现类。

(1) StandardDaoManager 类

StandardDaoManager 类是全部 DAO 的管理类, 更确切地说, 是容器类。在 StandardDaoManager 类有如下的属性:

```

private Map idContextMap = new HashMap();
private Map typeContextMap = new HashMap();
private Map daoImplMap = new HashMap();

```

其中 idContextMap 是放置当前 DaoManager 中所有的 DaoContext 实例化对象。typeContextMap 是存放当前 Manager 中所有的 Dao 接口类。daoImplMap 是存放当前 DaoManager 中所有的 Dao 代理类和接口类。这样 StandardDaoManager 类实际上是充当了

一个容器，把所有 DaoContext 和 Dao 都包容在里面。

同时 StandardDaoManager 类有具体操作方法，如 startTransaction、commitTransaction、endTransaction 等。但是这些方法都转移给内部的 DaoContext 去处理了。

(2) DaoContext 类

是当前上下文的 Dao 管理器，在 DaoContext 类有如下属性：

```
private Map typeDaoImplMap = new HashMap();
```

其中 typeDaoImplMap 属性存放当前 DaoContext 中所有的 Dao 实例化对象。这样 DaoContext 类也是充当了一个容器，把其所有的 Dao 都包容在里面了。

在 DaoContext 类有一个 DaoTransactionManager 属性，其主要是负责事务管理内容。

```
private DaoTransactionManager transactionManager;
```

与 StandardDaoManager 类一样，DaoContext 类也有 startTransaction、commitTransaction、endTransaction 等方法，他是接受 StandardDaoManager 类的调用，它也是调用 DaoTransactionManager 类的 transactionManager 属性的对应方法，如 DaoContext 类的 commitTransaction 方法就要调用 transactionManager 对象的 commitTransaction 方法。

(3) DaoImpl 和 DaoProxy 类

DaoImpl 和 DaoProxy 类是同时出现的。实际上 DaoImpl 的作用只是一个信息的存储体，或者是一个 Javabean，用于保存对应 Dao 的信息，而业务操作都是通过 DaoProxy 代理类来实现的。DaoProxy 代理类并不是代理 DaoImpl 类，而是代理 Dao 接口或者实现 Dao 接口的 Dao 实现类。

(4) DaoTransactionState

DaoTransactionState 类主要是描述 Dao 的事物状态，采用自身关联（反身关联），即自己引用自己，带着一个自己的引用也就是在自己的内部有一个自身的引用；而且这样的引用还是四个，如图 4-14 所示。

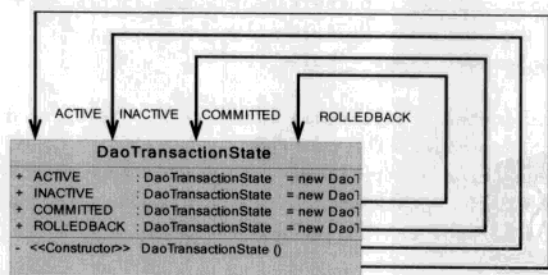


图 4-14 DaoTransactionState 类结构图

这样做的好处是便于直接调用，不用进行对象的实例化。

对于 iBATIS DAO 组件可以打这样一个比喻：DaoManager 是一栋大厦，DaoContext 是大厦中的一层楼，而 DaoTransactionManager 就是大厦每一层楼的会议室或者活动室。

而 DaoImpl 就是大楼中的工作人员，DaoProxy 就是大楼中每个工作人员的任务编号，而且一个任务编号只能对应一个工作人员。分配工作了，首先是获得了各个任务编号，按照编号找到相关的工作人员。如果这个工作比较简单，不需要多个工作人员协作，那么每个工作人员就自己完成自己的工作。如果这个工作比较复杂，需要多人协调合作完成，那就把这些工作人员都集中到会议室或者活动室，统一进行管理；要么这些人同时完成自己的工作并使整个工作完成，要么就是有某个人或某一些人没有完成自己的工作而造成整个工作不能完成。

单个人就能完成的工作，这是单事务处理，而多个人协同完成工作就是多事务处理。

2. iBATIS DAO 中对象的转化

对于配置文件中 transactionManager 的属性设置，系统通过实现 DaoTransactionManager 接口的实例化对象找到实现 DaoTransaction 接口的实例化对象，最终转化为具体的实例化类名而采取的实例化对象去处理。如 SQLMAP 的类名称就是 com.ibatis.SqlMap.Client.SqlMapClient。同理，针对不同的 transactionManager 配置，都转化为不同的实例化对象去处理，其对应关系如表 4-5 所示。

表 4-5 transactionManager 配置信息与最终实例化对象的对应关系

序 号	配置名称	transactionManager 最终实例化对象	备 注
1	JDBC	java.sql.Connection 实例化对象	
2	SQLMAP	com.ibatis.sqlmap.client.SqlMapClient 实例化对象	
3	hibernate	org.hibernate.Session 实例化对象实例化对象	
4	Apache Ojb	org.apache.Apache Ojb.broker.PersistenceBroker 实例化对象	
5	toplink	oracle.toplink.sessions.Session 实例化对象	

下面说明 SQLMAP 如何转化为 com.ibatis.sqlmap.client.SqlMapClient 实例化对象。

在配置信息中，对 transactionManager 节点读取的代码如下（已经增加了注解）：

```
private DaoTransactionManager parseTransactionManager(Element transPoolElement)
throws DaoException {
    DaoTransactionManager txMgr = null;

    Properties attributes = parseAttributes(transPoolElement);
    //获取 transactionManager 节点的 type 内容
    String implementation = attributes.getProperty("type");
    //根据 type 内容获取全名，SQLMAP 的全名是 SqlMapDaoTransactionManager.class.getName()
    implementation = resolveAlias(implementation);

    try {
        //对类 com.ibatis.dao.engine.transaction.sqlmap.SqlMapDaoTransactionManager 进行实例化
        txMgr = (DaoTransactionManager) Resources.classForName(implementation).
        newInstance();
    } catch (Exception e) {
        throw new DaoException("Error while configuring DaoManager. Cause: " +
```

```

e.toString(), e);
    }

    Properties props = properties;
    if (props == null) {
        props = parsePropertyElements(transPoolElement);
    } else {
        props.putAll(parsePropertyElements(transPoolElement));
    }
    // 对 SqlMapDaoTransactionManager 实例化对象赋初值, 初值的信息由 transactionManager 节点信息获取
    txMgr.configure(props);

    if (txMgr == null) {
        throw new DaoException("Error while configuring DaoManager. Some unknown condition caused the " +
            "DAO Transaction Manager to be null after configuration.");
    }
    return txMgr;
}

```

在上述代码中, 有一句创建 DaoTransactionManager 的实例化对象。

```
txMgr = (DaoTransactionManager) Resources.classForName(implementation).newInstance();
```

当配置文件中的 type 是 SQLMAP, 那么根据表 4-5 就获得 com.ibatis.dao.engine.transaction.sqlmap.SqlMapDaoTransactionManager 实例化对象, 然后对这个对象进行赋初值, 代码如下:

```

public void configure(Properties properties) {
    try {
        Reader reader = null;
        if (properties.containsKey("SqlMapConfigURL")) {
            reader = Resources.getUrlAsReader((String) properties.get("SqlMapConfigURL"));
        } else if (properties.containsKey("SqlMapConfigResource")) {
            reader = Resources.getResourceAsReader((String) properties.get("SqlMapConfigResource"));
        } else {
            throw new DaoException("SQLMAP transaction manager requires either 'SqlMapConfigURL' or 'SqlMapConfigResource' to be specified as a property.");
        }
        client = SqlMapClientBuilder.buildSqlMapClient(reader, properties);
    } catch (IOException e) {
        throw new DaoException("Error configuring SQL Map. Cause: " + e);
    }
}

```

其中黑体的这一句, 就说明已经转化为 SQLMap 的 SqlMapClient 了。关于 SqlMapClient 的用法, 在下面的章节中有详细的描述。

4.4.3 iBATIS DAO 事务管理实现

iBATIS DAO 事务管理组件不但要管理事务连接池，同时也要管理多个 ORM 的事务处理。iBATIS DAO 框架重要的因素之一是它全面的事务支持。iBATIS DAO 框架是对通用事务管理的一个包装，向开发人员提供了一致的事务管理抽象。iBATIS DAO 目前支持如下 7 种实现事务的管理框架：JDBC、Apache Ojb、Toplink、JTA、SQLMap、Hibernate 和 EXTERNAL 事务。

JDBC 的事务管理器：用 DataSource API 实现连接池服务。现阶段支持三种 DataSource，分别为 Simple、DBCP 和 JNDI。Simple 是采用 iBATIS 自己实现的一个 SimpleDataSource，其理想运行场所是对可靠性要求低并且比较独立的事务处理。DBCP 是采用 Jakarta DBCP 的 DataSource。JNDI 事务是基于检索 JNDI 目录而生成的 DataSource。

JTA 事务管理器管理事务使用 JTA 的 API 来实现，实现模式与上述基本一致，还是要通过检索 JNDI 目录而生成的 DataSource。但这个 JTA 事务管理器允许开发人员对 UserTransaction 实例进行控制。

SQLMap 事务管理器基于 iBATIS SQLMap 框架平台的事务处理。iBATIS DAO 事务管理在此只起到一个二传手的作用。Hibernate 事务管理器、Apache Ojb 事务管理器和 Toplink 事务管理器基本与 SQLMap 事务管理器类似，都是转移给相关平台的事务管理来进行处理的。

1. iBATIS DAO 事务管理的实现和配置

DAO 本身是没有进行事务处理的。在事务处理方面，DAO 充当了一个二传手。它把事务处理都转移给各个持久层来进行控制，实现二传手的工具就是 DaoContext。DAO 所有的事务处理都提交到 DaoContext 接口方法，然后 DaoContext 接口通过调用持久层接口的相应方法来实现。

2. DAO 事务管理的结构说明

iBATIS DAO 框架提供了事务管理模块。而这个事务管理可以应用到很多场合，包括 JDBC、Hibernate、JTA、SQLMap 等。其中 JtaDaoTransaction 类、JdbcDaoTransaction 类、SqlMapDaoTransaction 类和 HibernateDaoTransaction 类实现 ConnectionDaoTransaction 接口。ConnectionDaoTransaction 接口继承 DaoTransaction 接口。而 OjbBrokerDaoTransaction 类和 ToplinkDaoTransaction 类直接实现 DaoTransaction 接口。OjbBrokerTransactionManager 类、JtaDaoTransactionManager 类、JdbcDaoTransactionManager 类、HibernateDaoTransactionManager 类、SqlMapDaoTransactionManager 类、ToplinkDaoTransactionManager 类实现 DaoTransactionManager 接口。具体的类结构图如图 4-15 所示。

DaoTransactionManager 接口和 DaoTransactionDao 接口的关系很奇特。TransactionManager 接口只是依赖 DaoTransaction 接口，两者并没有建立关联。但是，这种依赖关系可以等同于一种关联关系，即 DaoTransactionManager 接口只有在 DaoTransaction

接口实现的条件下，才能进行正常的工作。这种依赖就好像一个桥梁，把 DaoTransactionManager 接口和 DaoTransaction 关联了起来。

深层次地抽象两者关系，这其实是一种变形的桥梁模式。也就是说，我们可以用桥梁模式的思路去理解这种结构模式。桥梁（Bridge）模式属于结构型设计模式。它将有关联的两个事务的抽象化与实现化脱耦，使得两者可以独立地变化，也就是说，将它们之间的强关联变成弱关联，是指在一个软件系统的抽象化和实现化之间使用组合/聚合关系而不是继承关系，从而使两者可以独立地变化。

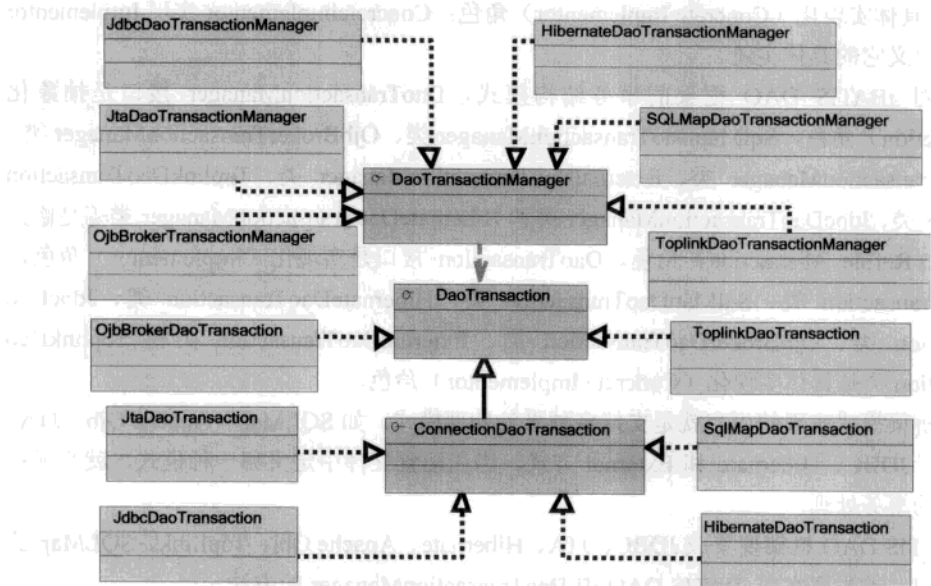


图 4-15 iBATIS DAO 事务管理类结构图

Bridge 结构如图 4-16 所示。Bridge 角色包括抽象化（Abstraction）角色、修正抽象化（Refine Abstraction）角色、实现化（Implementor）角色和具体实现化（Concrete Implementor）角色。

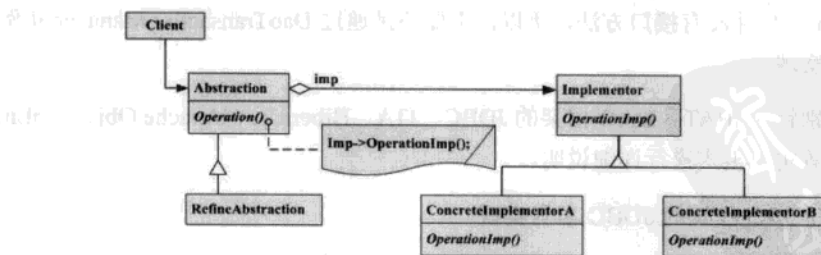


图 4-16 Bridge 设计模式结构

① 抽象化（Abstraction）角色：Abstraction 定义抽象类的接口。维护一个指向

Implementor 类型对象的指针。抽象化 (Abstraction) 角色可以是接口, 也可以是抽象类。属于不可缺少角色。

② 修正抽象化 (Refine Abstraction) 角色: RefinedAbstraction 扩充由 Abstraction 定义的接口 Implementor 定义实现类的接口。该接口不一定要与 Abstraction 的接口完全一致。

③ 实现化 (Implementor) 角色: Implementor 接口仅提供基本操作。这个抽象类规范具体实现化角色, 规定出具体实现化角色当有的 (非私有) 方法和属性。实现化 (Implementor) 角色可以是接口, 也可以是抽象类, 属于不可缺少角色。

④ 具体实现化 (Concrete Implementor) 角色: Concreteimplementor 实现 Implementor 接口并定义它的具体实现。

针对 iBATIS DAO 框架的事务结构模式, DaoTransactionManager 接口是抽象化 (Abstraction) 角色, SqlMapDaoTransactionManager 类、OjbBrokerTransactionManager 类、JtaDaoTransactionManager 类、ExternalDaoTransactionManager 类、ToplinkDaoTransactionManager 类、JdbcDaoTransactionManager 类和 HibernateDaoTransactionManager 类等是修正抽象化 (Refine Abstraction) 角色。DaoTransaction 接口是实现化 (Implementor) 角色, JtaDaoTransaction 类、SqlMapDaoTransaction 类、HibernateDaoTransaction 类、JdbcDaoTransaction 类、OjbBrokerDaoTransaction 类、ExternalDaoTransaction 类和 ToplinkDaoTransaction 类是具体实现化 (Concrete Implementor) 角色。

该桥梁模式实现的内容就是支持多种事务处理模式, 如 SQLMap、Apache Ojb、JTA、Toplink、JDBC、Hibernate 和 External 方式。当在配置文件中定义哪一种模式, 就实现那种模式的事务处理。

iBATIS DAO 框架要集成 JDBC、JTA、Hibernate、Apache Obj、TopLink、SQLMap 的事务框架, 首先要实现 iBATIS DAO 中 DaoTransactionManager 的方法。

```
public void configure(Properties properties);  
public DaoTransaction startTransaction();  
public void commitTransaction(DaoTransaction trans);  
public void rollbackTransaction(DaoTransaction trans);
```

同时, 在实现这些方法的同时, 会创建不同的 DaoTransaction 实例化对象。由于 DaoTransaction 本身并没有接口方法, 所以, 主要还是通过 DaoTransactionManager 实例化对象的方法来实现。

下面就分别针对 iBATIS DAO 框架的 JDBC、JTA、Hibernate、Apache Obj、TopLink、SQLMap 的事务实现模式进行详细说明。

3. iBATIS DAO 框架的 JDBC 事务实现

基于 JDBC 的数据库操作中, 一项事务是由一条或是多条表达式所组成的一个不可分割的工作单元。关于事务操作的方法都位于接口 java.sql.Connection 中。我们通过调用 Connection 的 commit 方法来提交事务, 用调用 rollback 方法来回滚或结束事务。一般情况下, JDBC 的

事务操作默认是自动提交。也就是说，一条对数据库的更新表达式代表一项事务操作，操作成功后，系统将自动调用 `commit` 来提交，否则将调用 `rollback` 来回滚。但是可以通过调用 `Connection` 的 `setAutoCommit(false)` 来禁止自动提交。之后就可以把多个数据库操作的表达式作为一个事务，在操作完成后调用 `commit` 来进行整体提交，倘若其中一个表达式操作失败，都不会执行到 `commit`，并且将产生响应的异常；此时就可以在异常捕获时调用 `rollback` 进行回滚。这样做可以保持多次更新操作后，相关数据的一致性，代码示例如下：

```

...
try {
    conn = DriverManager.getConnection(jdbc:oracle:thin:@host:1521:SID; username;,
userpwd);
    conn.setAutoCommit(false); //禁止自动提交，设置回滚点
    stmt = conn.createStatement();
    stmt.executeUpdate("alter table ..."); //数据库更新操作 1
    stmt.executeUpdate("insert into table ..."); //数据库更新操作 2
    conn.commit(); //事务提交
} catch (Exception ex) {
    ex.printStackTrace();
    try {
        conn.rollback(); //操作不成功则回滚
    } catch (Exception e) {
        e.printStackTrace();
    }
}
}

```

iBATIS DAO JDBC 组件还包含了 JDBC 的 `Connection` 等内容，总共由 5 个接口，5 个类组成。其中 `DaoTransaction` 接口、`DaoTransactionManager` 接口和 `ConnectionDaoTransaction` 接口是通用接口，也是外部调用的 API 接口。`JdbcDaoTransactionManager` 类实现 `DaoTransactionManager` 接口并关联 JDBC 的 `DataSource` 接口，同时它也依赖 `JdbcDaoTransaction` 类。`JdbcDaoTransaction` 类实现 `ConnectionDaoTransaction` 接口（该接口继承 `DaoTransaction` 接口）并关联 JDBC 的 `Connection` 接口。具体类结构如图 4-17 所示。

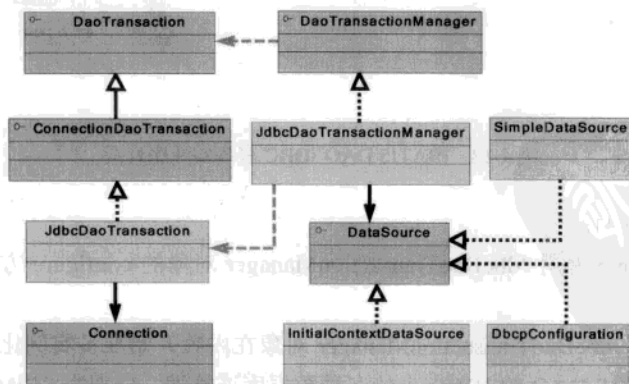


图 4-17 iBATIS DAO JDBC 事务管理类结构图

iBatis DAO 的 JDBC 事务处理基本上是按照上述过程来进行的。其实现过程为：JdbcDaoTransactionManager 对象创建出一个 DataSource 对象。这个 DataSource 对象可以有三种类型，分别为“SIMPLE”、“DBCP”和“JNDI”。当调用 JdbcDaoTransactionManager 对象的 startTransaction 方法时，该对象的 DataSource 对象为参数实例化一个 JdbcDaoTransaction 对象。所以，JdbcDaoTransaction 对象的 DataSource 属性都来自于 JdbcDaoTransactionManager 对象的属性。

当对 JdbcDaoTransactionManager 对象进行 commitTransaction、rollbackTransaction 等业务操作时，通常都通过 JdbcDaoTransactionManager 对象转移给 JdbcDaoTransaction 对象，而 JdbcDaoTransaction 对象最后转换给 Connection 对象的 commit 和 rollback 等方法来进行处理。iBatis DAO 的 JDBC 事务处理的序列图如图 4-18 所示。

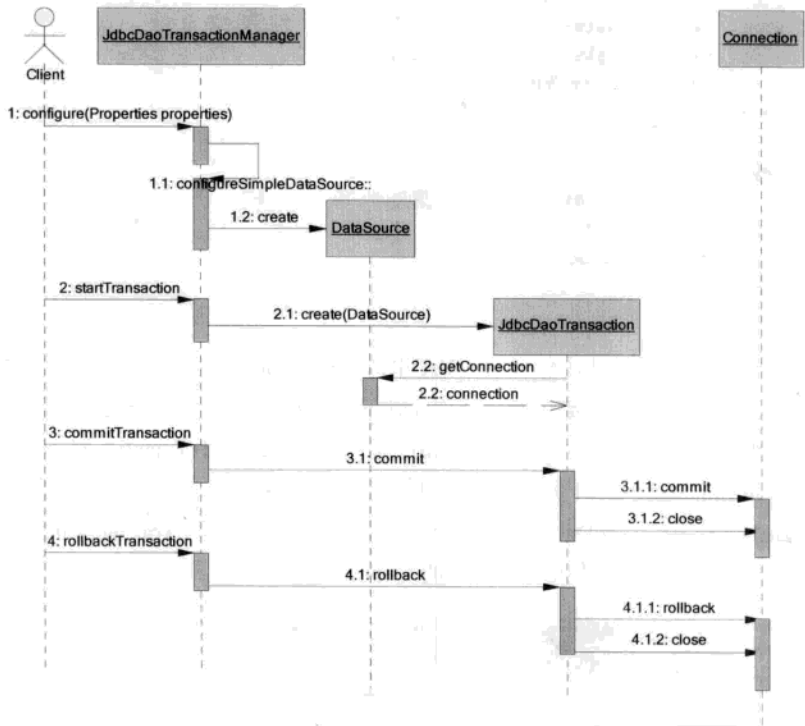


图 4-18 iBatis DAO JDBC 事务实现序列图

其实现步骤说明：

第 1 步骤：外部调用 JdbcDaoTransactionManager 对象的 configure 方法，并把配置信息参数传递进来。

第 1.1 步骤：JdbcDaoTransactionManager 对象在内容开始准备实例化 DataSource。这有三种选择，iBatis DAO 的 JDBC 支持三种数据库连接池，分别为“SIMPLE”、“DBCP”和“JNDI”。关于这几种连接池的应用，在 SQLMap 中会有更详细的描述。

第 1.2 步骤: JdbcDaoTransactionManager 对象在内容开始实例化 DataSource 连接池。

第 2 步骤: 外部调用 JdbcDaoTransactionManager 对象的 startTransaction 方法。

第 2.1 步骤: JdbcDaoTransactionManager 对象创建 JdbcDaoTransaction 对象, 并把参数 DataSource 传递过去。

第 2.2 步骤: JdbcDaoTransaction 对象调用 DataSource 的 getConnection 方法, 获得数据库的 Connection。

第 3 步骤: 外部调用 JdbcDaoTransactionManager 对象的 commitTransaction 方法。

第 3.1 步骤: JdbcDaoTransactionManager 对象调用 JdbcDaoTransaction 对象的 commit 方法。

第 3.1.1 步骤: JdbcDaoTransaction 对象调用 Connection 对象的 commit 方法。

第 3.1.2 步骤: JdbcDaoTransaction 对象接着调用 Connection 对象的 close 方法, 关闭数据库连接。

第 4 步骤: 外部调用 JdbcDaoTransactionManager 对象的 rollbackTransaction 方法。

第 4.1 步骤: JdbcDaoTransactionManager 对象调用 JdbcDaoTransaction 对象的 rollback 方法。

第 4.1.1 步骤: JdbcDaoTransaction 对象调用 Connection 对象的 rollback 方法。

第 4.1.2 步骤: JdbcDaoTransaction 对象接着调用 Connection 对象的 close 方法, 关闭数据库连接。

实现代码如下, 在代码中已经有详细的注解说明。

(1) JdbcDaoTransactionManager 的 configure 方法

按照总体部署, 首先是实现 configure。

在 JdbcDaoTransactionManager 的 configure 初始化参数, 并创建出 DataSource 实例。根据配置参数, 映射不同的 DataSource 实例。如 SIMPLE 则为:

```
public void configure(Properties properties) {
    if (properties.containsKey("DataSource")) {
        String type = (String) properties.get("DataSource");
        //根据 DataSource 配置值的不同, 分别去实例化不同的 DataSource 对象
        if ("SIMPLE".equals(type)) {
            configureSimpleDataSource(properties);
        } else if ("DBCP".equals(type)) {
            configureDhcp(properties);
        } else if ("JNDI".equals(type)) {
            configureJndi(properties);
        } else {
            throw new DaoException("DAO Transaction Manager properties must include a
value for 'DataSource' of SIMPLE, DBCP or JNDI.");
        }
    } else {
        throw new DaoException("DAO Transaction Manager properties must include a value
for 'DataSource' of SIMPLE, DBCP or JNDI.");
    }
}
```

```

    }

    //实例化 iBATIS 自己编写的 DataSource
    private void configureSimpleDataSource(Map properties) {
        dataSource = new SimpleDataSource(properties);
    }

    //实例化 Apache dbcp 的 DataSource
    private void configureDbcp(Map properties) {
        DbcpConfiguration dbcp = new DbcpConfiguration(properties);
        dataSource = dbcp.getDataSource();
    }

    //实例化 JNDI 定义的 DataSource
    private void configureJndi(Map properties) {
        try {
            Hashtable contextProps = getContextProperties(properties);
            InitialContext initCtx = null;
            if (contextProps == null) {
                initCtx = new InitialContext();
            } else {
                initCtx = new InitialContext(contextProps);
            }
            dataSource = (DataSource) initCtx.lookup((String) properties.get ("DBJndi
Context"));
        } catch (NamingException e) {
            throw new DaoException("There was an error configuring the DataSource from JNDI.
Cause: " + e, e);
        }
    }
}

```

在这里，为了理解方便，可以参看配置文件信息，配置的信息如下所示：

```

<transactionManager type="JDBC">
  <property name="DataSource" value="SIMPLE"/>
  <property name="JDBC.Driver" value="${driver}"/>
  <property name="JDBC.ConnectionURL" value="${url}"/>
  <property name="JDBC.Username" value="${username}"/>
  <property name="JDBC.Password" value="${password}"/>
  <property name="JDBC.DefaultAutoCommit" value="true" />
  <property name="Pool.MaximumActiveConnections" value="10"/>
  <property name="Pool.MaximumIdleConnections" value="5"/>
  <property name="Pool.MaximumCheckoutTime" value="120000"/>
</transactionManager>

```

(2) JdbcDaoTransactionManager 的 startTransaction()方法

JdbcDaoTransactionManager 的 startTransaction()方法是开启事务，代码如下：

```

public DaoTransaction startTransaction() {
    // 创建一个 JdbcDaoTransaction 对象
    return new JdbcDaoTransaction(dataSource);
}

```

让我们看看 JdbcDaoTransaction 类的构造函数。

```
public JdbcDaoTransaction(DataSource dataSource) {
    try {
        //获取数据库连接
        connection = dataSource.getConnection();
        if (connection == null) {
            throw new DaoException("Could not start transaction. Cause: The DataSource
returned a null connection.");
        }
        if (connection.getAutoCommit()) {
            //设置不为自动提交
            connection.setAutoCommit(false);
        }
        ....
    } catch (SQLException e) {
        throw new DaoException("Error starting JDBC transaction. Cause: " + e);
    }
}
```

创建出 JDBC 的 Connection 对象。

(3) JdbcDaoTransactionManager 的 commitTransaction 方法

JdbcDaoTransactionManager 的 commitTransaction 方法是提交事务，代码如下：

```
public void commitTransaction(DaoTransaction trans) {
    ((JdbcDaoTransaction) trans).commit();
}
```

让我们看看 JdbcDaoTransaction 类的 commit 方法，还是转移到交到 JDBC 中 Connection 对象的 commit 方法，完成任务后关掉数据库连接。

```
public void commit() {
    try {
        try {
            connection.commit();
        } finally {
            connection.close();
        }
    } catch (SQLException e) {
        throw new DaoException("Error committing JDBC transaction. Cause: " + e);
    }
}
```

(4) JdbcDaoTransactionManager 的 rollbackTransaction 方法

JdbcDaoTransactionManager 的 rollbackTransaction 方法是回滚事务，代码如下：

```
public void rollbackTransaction(DaoTransaction trans) {
    ((JdbcDaoTransaction) trans).rollback();
}
```

让我们看看 `JdbcDaoTransaction` 类的 `rollback` 方法，处理基本与 `commit` 相似。

```
public void rollback() {
    try {
        try {
            connection.rollback();
        } finally {
            connection.close();
        }
    } catch (SQLException e) {
        throw new DaoException("Error ending JDBC transaction. Cause: " + e);
    }
}
```

4. iBATIS DAO 框架的 JTA 事务实现

JTA (Java Transaction API) 主要用于 J2EE 分布式的多个数据源的两阶段提交的事务。JTA 事务因为其分布式和多数据源的特性，不可能由任何一个数据源实现事务，因此 JTA 中的事务是由事务管理器实现的，它会在多个数据源之间统筹事务，具体使用的技术就是所谓的两阶段提交事务，要用 JTA 进行事务界定，应用程序要调用 `Javax.transaction.UserTransaction` 接口中的方法。代码实现如下：

```
...
InitialContext ctx = new InitialContext();
Object txObj = ctx.lookup("java:comp/UserTransaction");
UserTransaction utx = (UserTransaction) txObj;
utx.begin();
...
DataSource ds = obtainXADataSource();
Connection conn = ds.getConnection();
pstmt = conn.prepareStatement("UPDATE MOVIES ...");
pstmt.setString(1, "Spinal Tap");
pstmt.executeUpdate();
// ...
utx.commit();
// ...
```

iBATIS DAO JTA 组件同样包含了 JTA 相关 `InitialContext`、`UserTransaction`、`DataSource`、`Connection` 等内容，总共由 3 个接口和 6 个类组成。其中 `DaoTransaction` 接口、`DaoTransactionManager` 接口和 `ConnectionDaoTransaction` 接口是通用接口，也是外部调用的 API 接口。`JtaDaoTransactionManager` 类实现 `DaoTransactionManager` 接口并关联 JDBC 的 `UserTransaction` 接口和 `DataSource` 接口，同时它也依赖 `JtaDaoTransaction` 类。`JtaDaoTransaction` 类实现 `ConnectionDaoTransaction` 接口（该接口继承 `DaoTransaction` 接口）并关联 JTA 的 `UserTransaction` 接口、`DataSource` 接口、`Connection` 接口，具体的类结构如图 4-19 所示。

iBATIS DAO 的 JTA 事务处理基本上还是按照上述过程来进行的，实现说明：`JtaDaoTransactionManager` 对象创建出一个 `InitialContext` 对象。通过 `InitialContext` 对象的 `lookup` 方法获得 `UserTransaction` 对象。由于要支持数据连接池，故再通过 `InitialContext`

对象的 lookup 方法获得 DataSource 对象。当调用 JtaDaoTransactionManager 对象的 startTransaction 方法时, 该对象以 UserTransaction 对象和 DataSource 对象为参数实例化一个 JtaDaoTransaction 对象。所以, JtaDaoTransaction 对象的 UserTransaction 属性和 DataSource 属性都是来自于 JtaDaoTransactionManager 对象的同样属性。

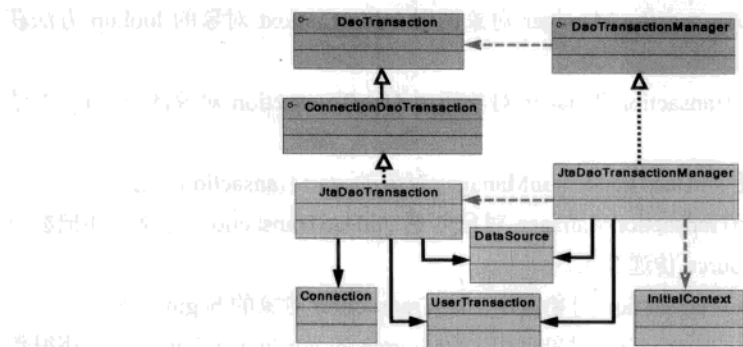


图 4-19 iBATIS DAO JTA 事务管理类结构图

要对 JtaDaoTransactionManager 对象进行 commitTransaction、rollbackTransaction 等业务操作, 可通过 JtaDaoTransactionManager 对象转移给 JtaDaoTransaction 对象的 commit 和 rollback 等方法。而 JtaDaoTransaction 对象也是最后转换给 UserTransaction 对象的 commit 和 rollback 等方法进行处理的, 序列图如图 4-20 所示。

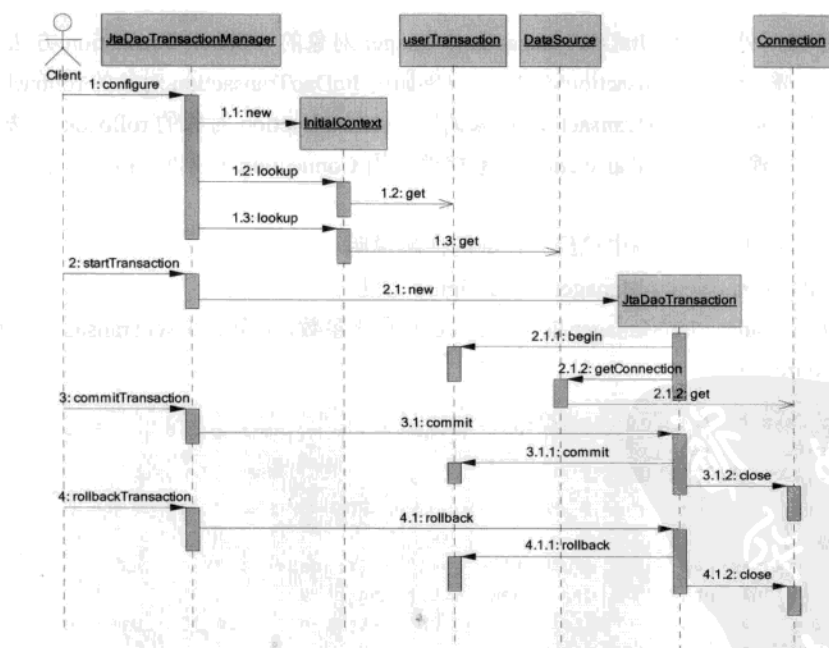


图 4-20 iBATIS DAO JTA 事务实现序列图

其实现的步骤如下:

第 1 步骤: 外部调用 JtaDaoTransactionManager 对象的 configure 方法, 并把配置信息参数传递进来。

第 1.1 步骤: JtaDaoTransactionManager 对象创建一个 InitialContext 对象。

第 1.2 步骤: JtaDaoTransactionManager 对象调用 InitialContext 对象的 lookup 方法获取 UserTransaction 对象。

第 1.3 步骤: JtaDaoTransactionManager 对象调用 UserTransaction 对象的 lookup 方法获取 DataSource 对象。

第 2 步骤: 外部调用 JtaDaoTransactionManager 对象的 startTransaction 方法。

第 2.1 步骤: JtaDaoTransactionManager 对象创建 JtaDaoTransaction 对象, 并把参数 UserTransaction 和 DataSource 传递过去。

第 2.1.1 步骤: JtaDaoTransaction 对象调用 UserTransaction 对象的 begin 方法。

第 2.1.2 步骤: JtaDaoTransaction 对象调用 DataSource 的 getConnection 方法, 获得数据库的 Connection。

第 3 步骤: 外部调用 JtaDaoTransactionManager 对象的 commitTransaction 方法。

第 3.1 步骤: JtaDaoTransactionManager 对象调用 JtaDaoTransaction 对象的 commit 方法。

第 3.1.1 步骤: JtaDaoTransaction 对象调用 UserTransaction 对象的 commit 方法。

第 3.1.2 步骤: JtaDaoTransaction 对象接着调用 Connection 对象的 close 方法, 关闭数据库连接。

第 4 步骤: 外部调用 JtaDaoTransactionManager 对象的 rollbackTransaction 方法。

第 4.1 步骤: JtaDaoTransactionManager 对象调用 JtaDaoTransaction 对象的 rollback 方法。

第 4.1.1 步骤: JtaDaoTransaction 对象调用 UserTransaction 对象的 rollback 方法。

第 4.1.2 步骤: JtaDaoTransaction 对象接着调用 Connection 对象的 close 方法, 关闭数据库连接。

实现代码如下, 在代码中已经有详细的注解说明。

(1) JtaDaoTransactionManager 的 configure 方法

在 JtaDaoTransactionManager 的 configure 初始化参数, 并创建 UserTransaction 对象和获得 DataSource 对象。

```
public void configure(Properties properties) {
    String utxName = null;
    String dsName = null;
    String contextMessage = "Error creating JNDI context.";
    try {
        utxName = (String) properties.get("UserTransaction");
        InitialContext initCtx = new InitialContext();
        contextMessage = "Error looking up user transaction '" + utxName + "'.";
        // 创建 UserTransaction 对象
        userTransaction = (UserTransaction) initCtx.lookup(utxName);
        dsName = (String) properties.get("DBJndiContext");
    } catch (Exception e) {
        throw new RuntimeException(contextMessage, e);
    }
}
```



```

        contextMessage = "Error looking up data source '" + dsName + "'.";
// 获得 DataSource 对象
        dataSource = (DataSource) initCtx.lookup(dsName);
    } catch (Exception e) {
        throw new DaoException(contextMessage + " Cause: " + e);
    }
}

```

在这里, 为了理解方便, 可以参看配置文件信息, 配置信息如下所示。

```

<transactionManager type="JTA">
    <property name="DBJndiContext" value="java:comp/env/jdbc/MyDataSource"/>
    <property name="UserTransaction" value="java:comp/env/UserTransaction"/>
</transactionManager>

```

(2) JtaDaoTransactionManager 的 startTransaction 方法

JtaDaoTransactionManager 的 startTransaction 方法是开启事务, 代码如下。

```

public DaoTransaction startTransaction() {
    return new JtaDaoTransaction(userTransaction, dataSource);
}

```

让我们看看 JtaDaoTransaction 类的构造函数。

```

public JtaDaoTransaction(UserTransaction utx, DataSource ds) {
    // Check parameters
    userTransaction = utx;
    dataSource = ds;
    if (userTransaction == null) {
        throw new DaoException("JtaTransaction initialization failed. UserTransaction
was null.");
    }
    if (dataSource == null) {
        throw new DaoException("JtaTransaction initialization failed. DataSource was
null.");
    }

    // 开始 JTA 事务
    try {
        newTransaction = userTransaction.getStatus() == Status.STATUS_NO_TRANSACTION;
        if (newTransaction) {
            userTransaction.begin();
        }
    } catch (Exception e) {
        throw new DaoException("JtaTransaction could not start transaction. Cause: ", e);
    }

    try {
        // 打开 JDBC Connection
        connection = dataSource.getConnection();
        if (connection == null) {
            throw new DaoException("Could not start transaction. Cause: The DataSource

```



```

returned a null connection.");
    }
    if (connection.getAutoCommit()) {
        connection.setAutoCommit(false);
    }
    .....
} catch (SQLException e) {
    throw new DaoException("Error opening JDBC connection. Cause: " + e, e);
}
}

```

创建出 JDBC 的 Connection 对象。

(3) JtaDaoTransactionManager 的 commitTransaction 方法

JtaDaoTransactionManager 的 commitTransaction 方法是提交事务，代码如下。

```

public void commitTransaction(DaoTransaction trans) {
    ((JtaDaoTransaction) trans).commit();
}

```

让我们看看 JtaDaoTransaction 类的 commit 方法，还是转移到 JDBC 中 Connection 对象的 commit 方法，完成任务后关掉数据库连接。

```

public void commit() {
    if (committed) {
        throw new DaoException("JtaTransaction could not commit because this transaction has already been committed.");
    }
    try {
        try {
            if (newTransaction) {
                userTransaction.commit();
            }
        } finally {
            close();
        }
    } catch (Exception e) {
        throw new DaoException("JtaTransaction could not commit. Cause: ", e);
    }
    committed = true;
}

```

(4) JdbcDaoTransactionManager 的 rollbackTransaction 方法

JdbcDaoTransactionManager 的 rollbackTransaction 方法是回滚事务，代码如下。

```

public void rollbackTransaction(DaoTransaction trans) {
    ((JtaDaoTransaction) trans).rollback();
}

```

让我们看看 JdbcDaoTransaction 类的 rollback 方法，处理基本与 commit 相似。

```

public void rollback() {

```

```

if (!committed) {
    try {
        try {
            if (userTransaction != null) {
                if (newTransaction) {
                    userTransaction.rollback();
                } else {
                    userTransaction.setRollbackOnly();
                }
            }
        } finally {
            close();
        }
    } catch (Exception e) {
        throw new DaoException("JTA transaction could not rollback. Cause: ", e);
    }
}
}

```

5. iBATIS DAO 框架的 Hibernate 事务实现

要理解 iBATIS DAO 如何集成 Hibernate 的事务实现，首先要知道原生态 Hibernate 是如何实现事务管理的。原生态 Hibernate 的事务实现序列如图 4-21 所示。

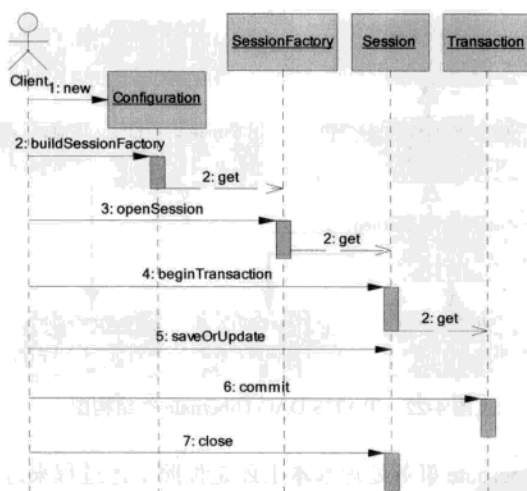


图 4-21 原生态 Hibernate 的事务实现序列图

在正常情况下，原生态 Hibernate 的 JDBC 事务实现代码如下：

```

Configuration config = new Configuration("hibernate.xml");
SessionFactory factory = config.buildSessionFactory();
Session session = factory.openSession();
Transaction tx = null;

```

```

try {
    tx = session.beginTransaction();
    // do some work
    ...
    tx.commit();
}
catch (RuntimeException e) {
    if (tx != null) tx.rollback();
    throw e; // or display error message
}
finally {
    session.close();
}

```

iBATIS DAO Hibernate 组件封装了 Hibernate 相关 SessionFactory、Session、Transaction 等内容，总共由三个接口、五个类组成。其中 DaoTransaction 接口、DaoTransactionManager 接口和 ConnectionDaoTransaction 接口是通用接口，也是外部调用的 API 接口。HibernateDaoTransactionManager 类实现 DaoTransactionManager 接口并关联 Hibernate 的 SessionFactory 接口，同时它也依赖 HibernateDaoTransaction 类。HibernateDaoTransaction 类实现 ConnectionDaoTransaction 接口（该接口继承 DaoTransaction 接口）并关联 Hibernate 的 SessionFactory、Session、Transaction 接口。具体类结构如图 4-22 所示。

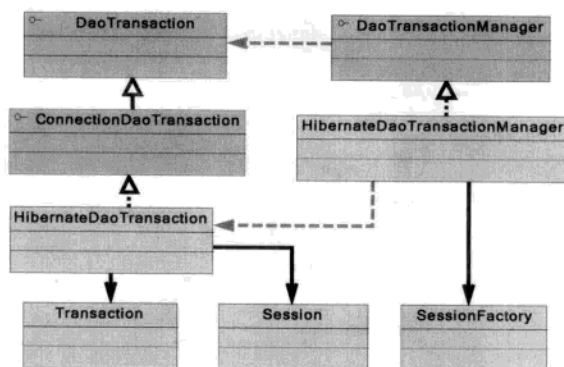


图 4-22 iBATIS DAO Hibernate 类结构图

iBATIS DAO 的 Hibernate 事务处理基本上还是按照上述过程来进行的。其实现过程为：初始化时，HibernateDaoTransactionManager 对象创建 Hibernate 的 SessionFactory 对象。当调用 HibernateDaoTransactionManager 对象的 startTransaction 方法时，该 SessionFactory 对象为参数实例化一个 HibernateDaoTransaction 对象。HibernateDaoTransaction 对象通过传递的 SessionFactory 对象创建 Session 对象，并以此 Session 对象创建 Transaction 对象。

要对 HibernateDaoTransactionManager 对象进行 commitTransaction、rollbackTransaction 等业务操作，需通过 HibernateDaoTransactionManager 对象转移给 HibernateDaoTransaction 对象的 commit 和 rollback 等方法。而 HibernateDaoTransaction 对象也是最后转移给

Transaction 对象的 commit 和 rollback 等方法进行处理的, 完成处理后调用 Session 对象的 close 方法, 关闭当前 Hibernate 会话。iBATIS DAO Hibernate 事务处理的序列如图 4-23 所示。

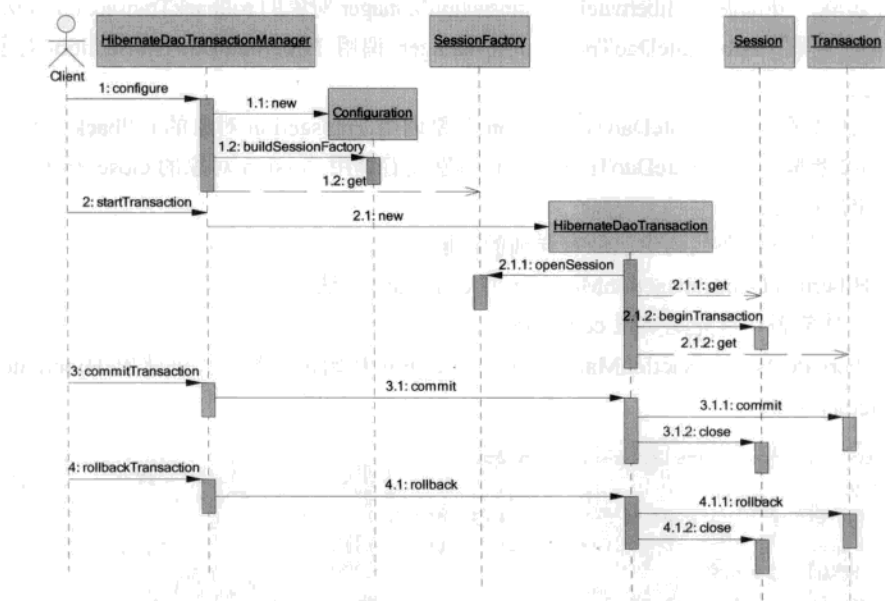


图 4-23 iBATIS DAO Hibernate 事务实现序列图

其实现步骤说明:

第 1 步骤: 外部调用 HibernateDaoTransactionManager 对象的 configure 方法, 并把配置信息参数传递进来。

第 1.1 步骤: HibernateDaoTransactionManager 对象创建 Hibernate 的 Configuration 对象。

第 1.2 步骤: HibernateDaoTransactionManager 对象调用 Configuration 对象的 buildSessionFactory 方法, 生成 SessionFactory 对象。

第 2 步骤: 外部调用 HibernateDaoTransactionManager 对象的 startTransaction 方法。

第 2.1 步骤: HibernateDaoTransactionManager 对象创建 HibernateDaoTransaction 对象, 并把参数 SessionFactory 对象传递过去。

第 2.1.1 步骤: HibernateDaoTransaction 对象调用 SessionFactory 对象的 openSession 方法, 获得 Hibernate 的 Session 对象。

第 2.1.2 步骤: HibernateDaoTransaction 对象调用 Session 对象的 beginTransaction 方法, 获得 Hibernate 的 Transaction 对象。

第 3 步骤: 外部调用 HibernateDaoTransactionManager 对象的 commitTransaction 方法。

第 3.1 步骤: HibernateDaoTransactionManager 调用 HibernateDaoTransaction 对象的 commit 方法。

第 3.1.1 步骤: HibernateDaoTransaction 对象调用 Transaction 对象的 commit 方法。

第 3.1.2 步骤: HibernateDaoTransaction 对象接着调用 Session 对象的 close 方法, 关闭 Hibernate 的当前会话。

第 4 步骤: 外部调用 HibernateDaoTransactionManager 对象的 rollbackTransaction 方法。

第 4.1 步骤: HibernateDaoTransactionManager 调用 HibernateDaoTransaction 对象的 rollback 方法。

第 4.1.1 步骤: HibernateDaoTransaction 对象调用 Transaction 对象的 rollback 方法。

第 4.1.2 步骤: HibernateDaoTransaction 对象接着调用 Session 对象的 close 方法, 关闭 Hibernate 的当前会话。

实现代码如下, 在代码中已经有详细的注解说明。

(1) HibernateDaoTransactionManager 的 configure 方法

按照总体部署, 首先是实现 configure。

在 HibernateDaoTransactionManager 的 configure 初始化参数, 并创建出 Hibernate 的 SessionFactory 实例。

```
public void configure(Properties properties) {
    try {
        Configuration config = new Configuration();
        Iterator it = properties.keySet().iterator();
        //读取配置参数信息
        while (it.hasNext()) {
            String key = (String) it.next();
            String value = (String) properties.get(key);
            if (key.startsWith("class.")) {config.addClass(Resources.classForName(value));}
            if (key.startsWith("map.")) { config.addResource(value); }
        }

        Properties props = new Properties();
        props.putAll(properties);
        config.setProperties(props);
        //根据配置参数创建 Hibernate 的 SessionFactory 实例对象
        factory = config.buildSessionFactory();
    } catch (Exception e) { throw new DaoException("Error configuring Hibernate.
Cause: " + e); }
}
```

在这里, 为了理解方便, 可以参看配置文件信息, 配置信息如下所示。

```
<transactionManager type="HIBERNATE">
  <property name="hibernate.dialect" value="net.sf.hibernate.dialect.PostgreSQL
Dialect"/>
  <property name="hibernate.connection.driver_class" value="${driver}"/>
  <property name="hibernate.connection.url" value="${url}"/>
  <property name="hibernate.connection.username" value="${username}"/>
  <property name="hibernate.connection.password" value="${password}"/>
  <property name="class.1" value="com.domain.Person"/>
</transactionManager>
```

```

<property name="class.2" value="com.domain.Business"/>
<property name="class.3" value="com.domain.Account"/>
</transactionManager>

```

(2) HibernateDaoTransactionManager 的 startTransaction()方法

HibernateDaoTransactionManager 的 startTransaction()方法是开启事务,代码如下:

```

public DaoTransaction startTransaction() {
    return new HibernateDaoTransaction(factory);
}

```

让我们看看 HibernateDaoTransaction 类的构造函数。

```

public HibernateDaoTransaction(SessionFactory factory) {
    try {
        this.session = factory.openSession();
        this.transaction = session.beginTransaction();
    } catch (HibernateException e) { throw new DaoException("Error starting Hibernate
transaction. Cause: " + e, e); }
}

```

创建出 Hibernate 的 Session 对象和 Transaction。

(3) HibernateDaoTransactionManager 的 commitTransaction 方法

HibernateDaoTransactionManager 的 commitTransaction 方法是提交事务,代码如下:

```

public void commitTransaction(DaoTransaction trans) {
    ((HibernateDaoTransaction) trans).commit();
}

```

让我们看看 HibernateDaoTransaction 类的 commit 方法。

```

public void commit() {
    try {
        try { transaction.commit();
        } finally { session.close(); }
    } catch (HibernateException e) { throw new DaoException("Error committing
Hibernate transaction. Cause: " + e);}
}

```

(4) HibernateDaoTransactionManager 的 rollbackTransaction 方法

HibernateDaoTransactionManager 的 rollbackTransaction 方法是回滚事务,代码如下:

```

public void rollbackTransaction(DaoTransaction trans) {
    ((HibernateDaoTransaction) trans).rollback();
}

```

让我们看看 HibernateDaoTransaction 类的 rollback 方法。

```

public void rollback() {
    try {
        try {
            transaction.rollback();
        } finally { session.close(); }
    }
}

```

```
        } catch (HibernateException e) {throw new DaoException("Error ending Hibernate transaction. Cause: " + e); }  
    }
```

6. iBatis DAO 框架的 Apache OJB 事务实现

Apache OJB 对象关系桥 (ObjectRelationalBridge) 是一种对象关系映射工具，它能够完成从 Java 对象到关系数据库的透明存储。要理解 iBatis DAO 如何集成 Apache OJB 的事务实现，首先要知道原生态 Apache OJB 是如何实现事务管理的。Apache OJB 支持单独模式和 C/S 模式，本例采用单独模式，也是 Apache OJB 的默认模式。使用 Apache OJB 的好处就是不需要进行初始化配置。

原生态 Apache OJB 的事务实现代码如下。

```
PersistenceBroker broker = null;  
try {  
    // create a broker and ask it to retrieve the Product collection  
    broker = PersistenceBrokerFactory.defaultPersistenceBroker();  
    // start broker transaction  
    broker.beginTransaction();  
    // do some work  
    ...  
    // 5. abort transaction (because we don't want to change anything)  
    broker.abortTransaction();  
}  
finally{  
    // 6. we should not hold onto the broker instance  
    if (broker != null){  
        broker.close();  
    }  
}
```

iBatis DAO OJB 组件封装了 Apache OJB 相关 PersistenceBroker 等内容，总共由两个接口，三个类组成。其中 DaoTransaction 接口和 DaoTransactionManager 接口是通用接口，也是外部调用的 API 接口。OjbBrokerTransactionManager 类实现 DaoTransactionManager 接口并依赖 OjbBrokerDaoTransaction 类。OjbBrokerDaoTransaction 类实现 DaoTransaction 接口并关联 Apache OJB 的 PersistenceBroker 接口。具体类结构如图 4-24 所示。

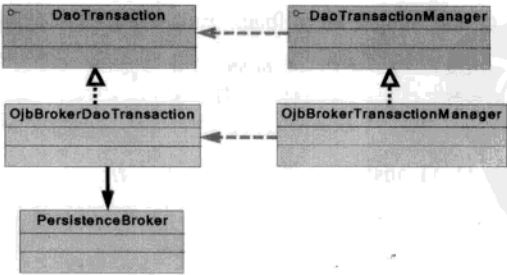


图 4-24 iBatis DAO Ojb 类结构图

iBATIS DAO 的 Obj 事务处理的实现过程：初始化时，在 ObjBrokerTransactionManager 的 configure 初始化参数，若没有内容，就不需要初始化。当调用 ObjBrokerTransactionManager 对象的 startTransaction 方法时，ObjBrokerTransactionManager 对象调用 PersistenceBrokerFactory 的静态方法 defaultPersistenceBroker 创建 PersistenceBroker 对象。然后，ObjBrokerTransactionManager 对象用新建的 PersistenceBroker 对象作为参数，实例化一个 ObjBrokerDaoTransaction 对象。

要对 ObjBrokerTransactionManager 对象进行 commitTransaction、rollbackTransaction 等业务操作，需通过 ObjBrokerTransactionManager 对象转移给 ObjBrokerDaoTransaction 对象的 commit 和 rollback 等方法。而 ObjBrokerDaoTransaction 对象也是最后转换给 PersistenceBroker 对象的 commitTransaction 和 abortTransaction 等方法进行处理的，完成后调用 PersistenceBroker 对象的 close 方法，关闭当前 ObjBroker 会话。在其事务处理的序列如图 4-25 所示。

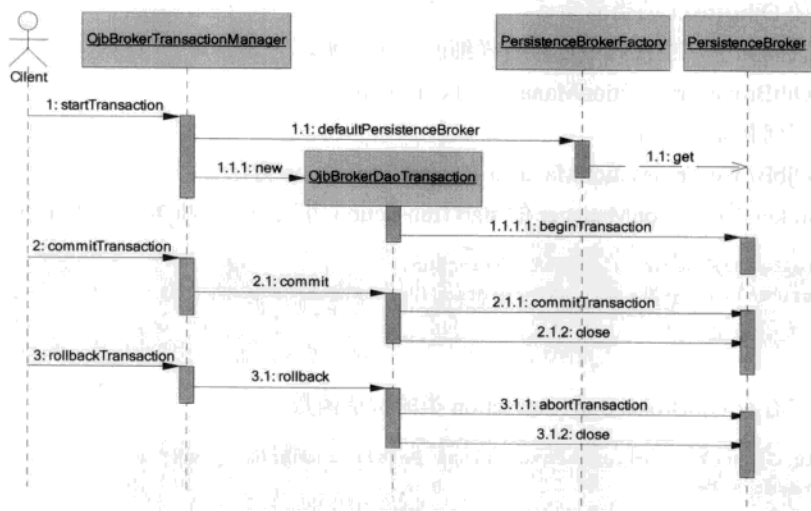


图 4-25 iBATIS DAO Obj 事务实现序列图

其实现步骤说明：

第 1 步骤：外部调用 ObjBrokerTransactionManager 对象的 startTransaction 方法。

第 1.1 步骤：ObjBrokerTransactionManager 对象调用 PersistenceBrokerFactory 的静态方法 defaultPersistenceBroker 创建 PersistenceBroker 对象。

第 1.1.1 步骤：ObjBrokerTransactionManager 对象把 PersistenceBroker 对象作为参数，实例化一个 ObjBrokerDaoTransaction 对象。

第 1.1.1.1 步骤：ObjBrokerDaoTransaction 对象调用 PersistenceBroker 对象的 beginTransaction 方法。

第 2 步骤：外部调用 ObjBrokerTransactionManager 对象的 commitTransaction 方法。

第 2.1 步骤: OjbBrokerTransactionManager 对象调用 OjbBrokerDaoTransaction 对象的 commit 方法。

第 2.1.1 步骤: OjbBrokerDaoTransaction 对象调用 PersistenceBroker 对象的 commit Transaction 方法。

第 2.1.2 步骤: OjbBrokerDaoTransaction 对象接着调用 PersistenceBroker 对象的 close 方法, 关闭 OjbBroker 的当前会话。

第 3 步骤: 外部调用 OjbBrokerTransactionManager 对象的 rollbackTransaction 方法。

第 3.1 步骤: OjbBrokerTransactionManager 对象调用 OjbBrokerDaoTransaction 对象的 rollback 方法。

第 3.1.1 步骤: OjbBrokerDaoTransaction 对象调用 PersistenceBroker 对象的 abort Transaction 方法。

第 3.1.2 步骤: OjbBrokerDaoTransaction 对象接着调用 PersistenceBroker 对象的 close 方法, 关闭 OjbBroker 的当前会话。

实现代码如下, 在代码中已经有详细的注解说明。

(1) OjbBrokerTransactionManager 的 configure 方法

不做任何初始化操作。

(2) OjbBrokerTransactionManager 的 startTransaction 方法

OjbBrokerTransactionManager 的 startTransaction 方法是开启事务, 代码如下:

```
public DaoTransaction startTransaction() {
    return new OjbBrokerDaoTransaction(PersistenceBrokerFactory. defaultPersistence
    Broker());
}
```

让我们看看 OjbBrokerDaoTransaction 类的构造函数。

```
public OjbBrokerDaoTransaction(final PersistenceBroker brk) {
    broker = brk;
    try {
        broker.beginTransaction();
    } catch (final Throwable t) {throw new DaoException("Error starting OJB broker
    transaction. Cause: " + t, t); }
}
```

创建出 Hibernate 的 Session 对象和 Transaction。

(3) OjbBrokerTransactionManager 的 commitTransaction 方法

OjbBrokerTransactionManager 的 commitTransaction 方法是提交事务, 代码如下:

```
public void commitTransaction(final DaoTransaction trans) {
    ((OjbBrokerDaoTransaction) trans).commit();
}
```

让我们看看 OjbBrokerDaoTransaction 类的 commit 方法。

```

public void commit() {
    try {
        broker.commitTransaction();
    } catch (final Throwable t) {
        throw new DaoException("Error committing APACHE OJB broker transaction. Cause: "
            + t);
    } finally {
        if (broker != null) {
            broker.close();
        }
    }
}

```

(4) OjbBrokerTransactionManager 的 rollbackTransaction 方法

OjbBrokerTransactionManager 的 rollbackTransaction 方法是回滚事务，代码如下：

```

public void rollbackTransaction(final DaoTransaction trans) {
    ((OjbBrokerDaoTransaction) trans).rollback();
}

```

让我们看看 OjbBrokerDaoTransaction 类的 rollback 方法。

```

public void rollback() {
    try {
        broker.abortTransaction();
    } catch (final Throwable t) {
        throw new DaoException("Error ending APACHE OJB broker transaction. Cause: "
            + t);
    } finally {
        if (broker != null) {
            broker.close();
        }
    }
}

```

7. iBATIS DAO 框架的 TopLink 事务实现

要理解 iBATIS DAO 如何集成 TopLink 的事务实现，首先要知道原生态 TopLink 是如何实现事务管理的。原生态 TopLink 的事务实现序列如图 4-26 所示。

原生态 TopLink 的事务实现。在正常情况下，TopLink 的事务实现代码如下。

```

SessionManager manager = SessionManager.getManager();
ServerSession session = (ServerSession) manager.getSession("ServerSession");
UnitOfWork uow = session.acquireUnitOfWork();
uow.beginEarlyTransaction();
PetOwner existingPetOwnerClone = (PetOwner)uow.readObject(PetOwner.class);

```

```

uow.commit();
uow.release();
session.release();

```

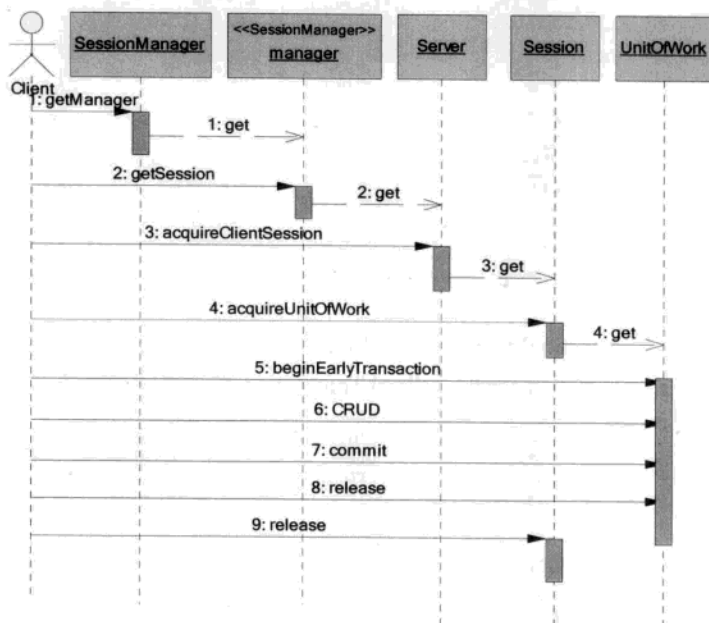


图 4-26 原生态 TopLink 的事务实现序列图

iBATIS DAO TopLink 组件封装了 TopLink 相关的 Server、Session、UnitOfWork 等内容，总共由两个接口、五个类组成。其中 DaoTransaction 接口和 DaoTransactionManager 接口是通用接口，也是外部调用的 API 接口。ToplinkDaoTransactionManager 类实现 DaoTransactionManager 接口并关联 Toplink 的 Server 接口和 UnitOfWork 接口，同时它也依赖于 ToplinkDaoTransaction 类。ToplinkDaoTransaction 类实现 DaoTransaction 接口并关联 Toplink 的 Server、UnitOfWork、Session 接口。具体类结构如图 4-27 所示。

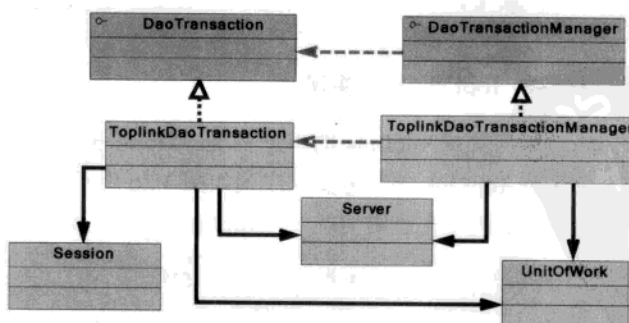


图 4-27 iBATIS DAO TopLink 事务管理类结构图

iBATIS DAO Toplink 事务实现说明：初始化时，ToplinkDaoTransactionManager 对象创建 Toplink 的 Server 对象。当调用 ToplinkDaoTransactionManager 对象的 startTransaction 方法时，该 Server 对象为参数实例化一个 ToplinkDaoTransaction 对象。ToplinkDaoTransaction 对象通过传递的 Server 对象创建 Toplink 的 Session 对象，并以此 Session 对象创建 UnitOfWork 对象。

要对 ToplinkDaoTransactionManager 对象进行 commitTransaction、rollbackTransaction 等业务操作，需通过 ToplinkDaoTransactionManager 对象转移给 ToplinkDaoTransaction 对象的 commit 和 rollback 等方法。而 ToplinkDaoTransaction 对象也是最后转移给 UnitOfWork 对象的 commit 和 revertAndResume 等方法进行处理的，完成处理后调用 UnitOfWork 对象的 release 方法，终结当前 UnitOfWork 的事务处理，接着再调用 Session 对象的 release 方法，关闭当前的 Toplink 会话。iBATIS DAO TopLink 事务实现的序列如图 4-28 所示。

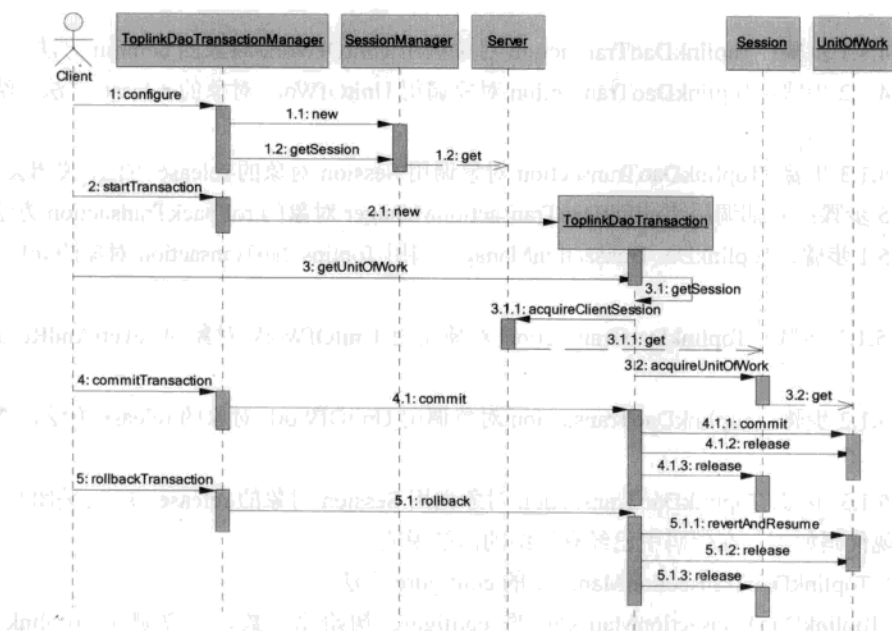


图 4-28 iBATIS DAO TopLink 事务实现序列图

其实现步骤说明：ToplinkDaoTransactionManager 和 ToplinkDaoTransaction

第 1 步骤：外部调用 ToplinkDaoTransactionManager 对象的 configure 方法，并把配置信息参数传递进来。

第 1.1 步骤：ToplinkDaoTransactionManager 对象调用 SessionManager 类的静态方法 getManager，实例化一个 SessionManager 对象。

第 1.2 步骤：ToplinkDaoTransactionManager 对象调用 SessionManager 对象的 getSession 方法，获得一个 Server 对象。

第2步骤：外部调用 ToplinkDaoTransactionManager 对象的 startTransaction 方法。

第2.1步骤：ToplinkDaoTransactionManager 对象创建 ToplinkDaoTransaction 对象，并把参数 Server 对象传递过去。

第3步骤：外部调用 ToplinkDaoTransaction 对象的 getUnitOfWork 方法。

第3.1步骤：ToplinkDaoTransaction 对象调用自身的 getSession 方法。

第3.1.1步骤：ToplinkDaoTransaction 对象调用 Server 对象的 acquireClientSession 方法，获得 Session 对象。

第3.1.2步骤：ToplinkDaoTransaction 对象调用 Session 对象的 acquireUnitOfWork 方法，获得一个 UnitOfWork 对象。

第4步骤：外部调用 ToplinkDaoTransactionManager 对象的 commitTransaction 方法。

第4.1步骤：ToplinkDaoTransactionManager 调用 ToplinkDaoTransaction 对象的 commit 方法。

第4.1.1步骤：ToplinkDaoTransaction 对象调用 UnitOfWork 对象的 commit 方法。

第4.1.2步骤：ToplinkDaoTransaction 对象调用 UnitOfWork 对象的 release 方法，结束事务。

第4.1.3步骤：ToplinkDaoTransaction 对象调用 Session 对象的 release 方法，关闭会话。

第5步骤：外部调用 ToplinkDaoTransactionManager 对象的 rollbackTransaction 方法。

第5.1步骤：ToplinkDaoTransactionManager 调用 ToplinkDaoTransaction 对象的 rollback 方法。

第5.1.1步骤：ToplinkDaoTransaction 对象调用 UnitOfWork 对象的 revertAndResume 方法。

第5.1.2步骤：ToplinkDaoTransaction 对象调用 UnitOfWork 对象的 release 方法，结束事务。

第5.1.3步骤：ToplinkDaoTransaction 对象调用 Session 对象的 release 方法，关闭会话。实现代码如下，在代码中已经有详细的注解说明。

(1) ToplinkDaoTransactionManager 的 configure 方法

在 ToplinkDaoTransactionManager 的 configure 初始化参数，并创建出 Toplink 的 SessionManager 实例。

```
public void configure(Properties properties) throws DaoException {
    // Get the name of the session and create it
    String sessionName = null;
    try {
        SessionManager manager = SessionManager.getManager();
        // Get the name of the session and create it
        sessionName = properties.getProperty("session.name");
        server = (Server) manager.getSession(sessionName,
            ToplinkDaoTransactionManager.class.getClassLoader());
    } catch (Exception e) {
        throw new DaoException(
```

```

        "Error configuring Toplink environment for session: "
        + sessionName);
    }
}

```

(2) ToplinkDaoTransactionManager 的 startTransaction()方法

ToplinkDaoTransactionManager 的 startTransaction()方法是开启事务，代码如下：

```

public DaoTransaction startTransaction() throws DaoException {
    ToplinkDaoTransaction trans = new ToplinkDaoTransaction(uow, server);
    return trans;
}

```

让我们看看 ToplinkDaoTransaction 类的构造函数。

```

public ToplinkDaoTransaction(UnitOfWork uow, Server server) throws DaoException {
    // Check arguments
    if (server == null) {throw new DaoException("Toplink Server not available"); }
    // Set the server settings
    this.unitOfWork = uow;
    this.server = server;
}

```

创建出 Hibernate 的 Session 对象和 Transaction。

(3) ToplinkDaoTransactionManager 的 commitTransaction 方法

ToplinkDaoTransactionManager 的 commitTransaction 方法是提交事务，代码如下：

```

public void commitTransaction(DaoTransaction transaction) throws DaoException {
    ((ToplinkDaoTransaction) transaction).commit();
}

```

让我们看看 ToplinkDaoTransaction 类的 commit 方法。

```

public void commit() throws DaoException {
    // Make sure the transaction has not been committed
    if (committed) {throw new DaoException("Transaction already committed");}
    // Now try to commit the transaction
    try {
        // Check for UnitOfWork
        if (unitOfWork != null) {
            // UnitOfWork was not lazily loaded
            unitOfWork.commit();
            unitOfWork.release();
            session.release();
        }
    } catch (TopLinkException e) {
        throw new DaoException("Error committing transaction", e);
    }
    committed = true;
}

```

(4) ToplinkDaoTransactionManager 的 rollbackTransaction 方法

ToplinkDaoTransactionManager 的 rollbackTransaction 方法是回滚事务，代码如下：

```
public void rollbackTransaction(DaoTransaction transaction) throws DaoException {
    ((ToplinkDaoTransaction) transaction).rollback();
}
```

让我们看看 ToplinkDaoTransaction 类的 rollback 方法。

```
public void rollback() throws DaoException {
    // Commit the transaction if it has not been committed
    if (!committed) {
        try {

            // Make sure the UOW is still active before rollback
            if ((unitOfWork != null) && unitOfWork.isActive()) {
                unitOfWork.revertAndResume();
                unitOfWork.release();
                session.release();
            }
        } catch (TopLinkException e) {throw new DaoException("Error rolling
back transaction", e); }
    }
}
```

8. iBATIS DAO 框架的 SQLMap 事务实现

要理解 iBATIS dao 如何集成 iBATIS SQLMap 的事务实现，首先要知道原生态 SQLMap 是如何实现事务管理的。

原生态 SQLMap 的事务实现序列如图 4-29 所示。

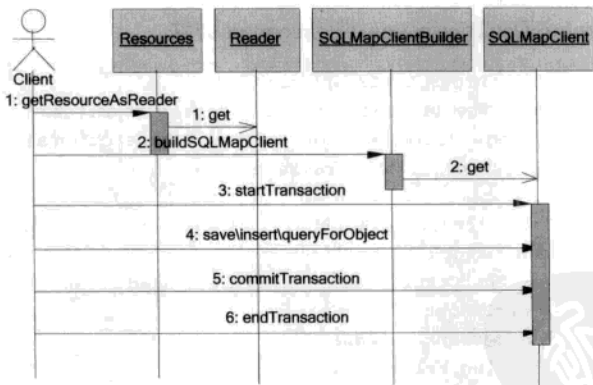


图 4-29 原生态 SQLMap 的事务实现序列图

原生态 iBATIS SQLMap 的事务实现。在正常情况下，iBATIS SQLMap 的事务实现代码如下。

```
Reader reader = Resources.getResourceAsReader ("sql-map-config.xml");
```

```

try {
    SqlMapClient sqlMap = SqlMapClientBuilder.buildSqlMapClient(reader);
    sqlMap.startTransaction();
    // do some work
    ...
    sqlMap.commitTransaction();
} catch (Exception e) {
    throw new RuntimeException (e);
}
finally {
    sqlMap.endTransaction();
}

```

iBATIS DAO SQLMap 组件封装了 iBATIS SQLMap 相关 SqlMapClient 等内容, 总共由两个接口、三个类组成。其中 DaoTransaction 接口、DaoTransactionManager 接口和 ConnectionDaoTransaction 接口是通用接口, 也是外部调用的 API 接口。SqlMapDaoTransactionManager 类实现 DaoTransactionManager 接口并关联 iBATIS SQLMap 的 SqlMapClient 接口, 同时它也依赖 SqlMapDaoTransaction 类。SqlMapDaoTransaction 类实现 ConnectionDaoTransaction 接口 (该接口继承 DaoTransaction 接口) 并关联 iBATIS SQLMap 的 SqlMapClient 接口。具体类结构如图 4-30 所示。

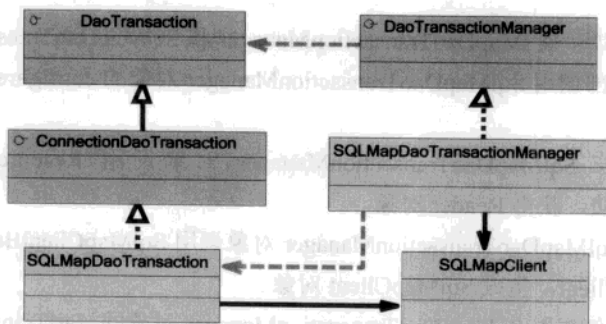


图 4-30 iBATIS DAO SQLMap 事务管理类结构图

iBATIS DAO SQLMap 事务管理实现说明: (1) 初始化时, SqlMapDaoTransactionManager 对象创建 SqlMapClient 对象。(2) 当调用 SqlMapDaoTransactionManager 对象的 startTransaction 方法时, 该 SqlMapClient 对象为参数实例化一个 SqlMapDaoTransaction 对象, 并调用 SqlMapClient 对象的 startTransaction 方法。(3) 当对 SqlMapDaoTransactionManager 对象进行 commitTransaction、rollbackTransaction 等业务操作, SqlMapDaoTransactionManager 对象转移给 SqlMapDaoTransaction 对象的 commit 和 rollback 等方法。而 SqlMapDaoTransaction 对象也是最后转换给 SqlMapClient 对象的 commitTransaction 或 endTransaction 等方法进行处理的, 完成处理后调用 SqlMapClient 对象的 endTransaction 方法, 结束当前事务处理。iBATIS DAO SQLMap 事务处理的序列如图 4-31 所示。

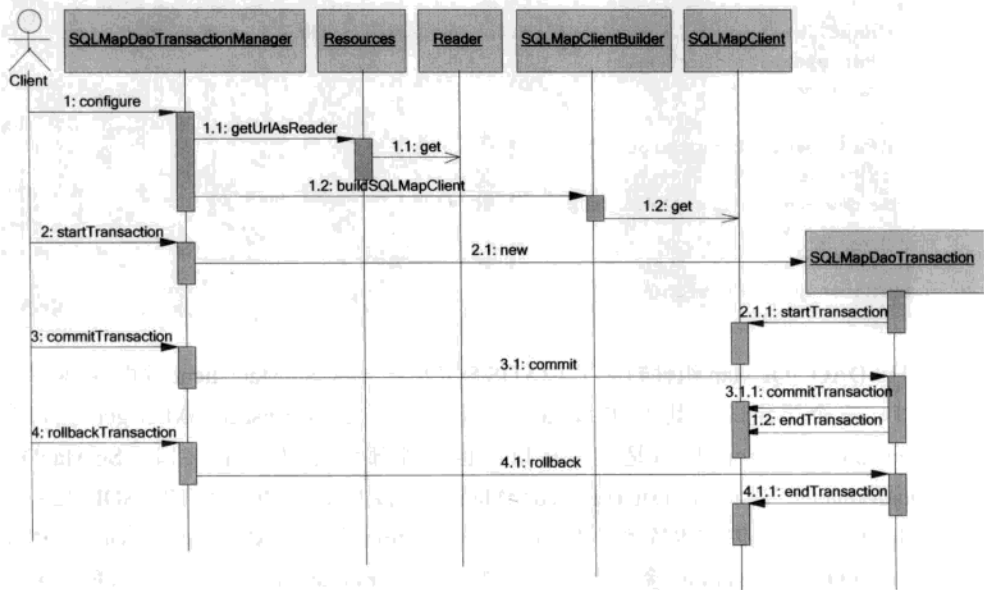


图 4-31 iBATIS DAO SQLMap 事务实现序列图

- 其实现步骤说明：SqlMapDaoTransactionManager 和 SqlMapDaoTransaction
- 第 1 步骤：外部调用 SqlMapDaoTransactionManager 对象的 configure 方法，并把配置信息参数传递进来。
- 第 1.1 步骤：SqlMapDaoTransactionManager 对象调用 Resources 的静态方法 getUrlAsReader 方法，获得 Reader 对象。
- 第 1.2 步骤：SqlMapDaoTransactionManager 对象调用 SqlMapClientBuilder 对象的静态方法 buildSqlMapClient，生成 SqlMapClient 对象。
- 第 2 步骤：外部调用 SqlMapDaoTransactionManager 对象的 startTransaction 方法。
- 第 2.1 步骤：SqlMapDaoTransactionManager 对象创建 SqlMapDaoTransaction 对象，并把参数 SqlMapClient 对象传递过去。
- 第 2.1.1 步骤：SqlMapDaoTransaction 对象调用 SqlMapClient 对象的 startTransaction 方法。
- 第 3 步骤：外部调用 SqlMapDaoTransactionManager 对象的 commitTransaction 方法。
- 第 3.1 步骤：SqlMapDaoTransactionManager 对象调用 SqlMapDaoTransaction 对象的 commit 方法。
- 第 3.1.1 步骤：SqlMapDaoTransaction 对象调用 SqlMapClient 对象的 commitTransaction 方法。
- 第 3.1.2 步骤：SqlMapDaoTransaction 对象调用 SqlMapClient 对象的 endTransaction 方法，关闭 SQLMap 的当前事务。

第 4 步骤：外部调用 `SqlMapDaoTransactionManager` 对象的 `commitTransaction` 方法。

第 4.1 步骤：`SqlMapDaoTransactionManager` 对象调用 `SqlMapDaoTransaction` 对象的 `commit` 方法。

第 4.1.1 步骤：`SqlMapDaoTransaction` 对象调用 `SqlMapClient` 对象的 `endTransaction` 方法，关闭 `SQLMap` 的当前事务。

实现代码如下，在代码中已经有详细的注解说明。

(1) `SqlMapDaoTransactionManager` 的 `configure` 方法

在 `SqlMapDaoTransactionManager` 的 `configure` 初始化参数，并创建出 `SQLMap` 的 `SqlMapClient` 实例化对象。

```
public void configure(Properties properties) {
    try {
        Reader reader = null;
        if (properties.containsKey("SqlMapConfigURL")) {
            reader = Resources.getUrlAsReader((String) properties.get("SqlMapConfigURL"));
        } else if (properties.containsKey("SqlMapConfigResource")) {
            reader = Resources.getResourceAsReader((String) properties.get("SqlMapConfigResource"));
        } else {
            throw new DaoException("SQLMAP transaction manager requires either 'SqlMapConfigURL' or 'SqlMapConfigResource' to be specified as a property.");
        }
        client = SqlMapClientBuilder.buildSqlMapClient(reader, properties);
    } catch (IOException e) {
        throw new DaoException("Error configuring SQL Map. Cause: " + e);
    }
}
```

在这里，为了理解方便，可以参看配置文件信息，配置信息如下所示。

```
<transactionManager type="SQLMAP">
<property name="SqlMapConfigResource" value="com/domain/dao/sqlmap/SqlMapConfig.xml"/>
</transactionManager>
```

(2) `SqlMapDaoTransactionManager` 的 `startTransaction()` 方法

`SqlMapDaoTransactionManager` 的 `startTransaction()` 方法是开启事务，代码如下：

```
public DaoTransaction startTransaction() {
    return new SqlMapDaoTransaction(client);
}
```

让我们看看 `SqlMapDaoTransaction` 类的构造函数。

```
public SqlMapDaoTransaction(SqlMapClient client) {
    try {
        client.startTransaction();
        this.client = client;
    }
```

```

    } catch (SQLException e) {
        throw new DaoException("Error starting SQL Map transaction. Cause: " + e, e);
    }
}

```

(3) SqlMapDaoTransactionManager 的 commitTransaction 方法

SqlMapDaoTransactionManager 的 commitTransaction 方法是提交事务，代码如下：

```

public void commitTransaction(DaoTransaction trans) {
    ((SqlMapDaoTransaction) trans).commit();
}

```

让我们看看 SqlMapDaoTransaction 类的 commit 方法。

```

public void commit() {
    try {
        client.commitTransaction();
        client.endTransaction();
    } catch (SQLException e) {
        throw new DaoException("Error committing SQL Map transaction. Cause: " + e, e);
    }
}

```

(4) SqlMapDaoTransactionManager 的 rollbackTransaction 方法

SqlMapDaoTransactionManager 的 rollbackTransaction 方法是回滚事务，代码如下：

```

public void rollbackTransaction(DaoTransaction trans) {
    ((SqlMapDaoTransaction) trans).rollback();
}

```

让我们看看 SqlMapDaoTransaction 类的 rollback 方法。

```

public void rollback() {
    try {
        client.endTransaction();
    } catch (SQLException e) {
        throw new DaoException("Error ending SQL Map transaction. Cause: " + e, e);
    }
}

```

4.5 基于 iBATIS DAO SqlMap 的实例说明

那我们可以从一个具体的实现例子来展现 iBATIS DAO 框架的实现过程。例子程序为 iBATIS 经典的 jpetstore 例子。

iBATIS DAO 架构要求应用层次要分为四层，分别为域数据层、持久接口层、持久实现层和服务层。域数据层主要存放 JavaBean (POJO 类)。持久接口层主要存放业务实现的 dao 接口，持久实现层存放业务实现的 Dao 实现类。服务层存放上层的业务服务类，当然

也可以是接口。

jpetstore 例子代码中实现 AccountService 实现过程的应用场景。外部调用利用 Account 对象，通过调用 AccountDao 接口的 insertAccount 方法，实现对 Account 对象针对数据库表 ACCOUNT 新增记录操作。应用场景如图 4-32 所示。

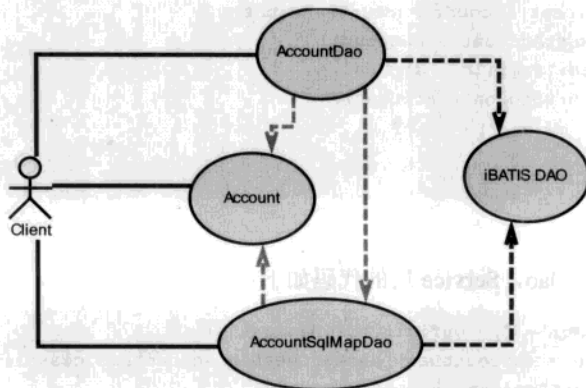


图 4-32 iBATIS DAO SqlMap 事务实现序列图

图 3-34 说明 Client 访问 Account 对象、AccountDao 接口和 AccountSqlMapDao 实现类。AccountDao 接口和 AccountSqlMapDao 实现类都依赖 Account 对象，同时 AccountDao 接口和 AccountSqlMapDao 都是在 iBATIS DAO 组件下工作的。

dao.xml 配置文件中针对 AccountDao 接口和 AccountSqlMapDao 实现类配置信息如下。

```

<daoConfig>
  <context>
    <transactionManager type="SQLMAP">
      <property name="SqlMapConfigResource"
        value="com/ibatis/jpetstore/persistence/sqlmapdao/sql/sql-map-config.xml"/>
    </transactionManager>
    ...
    <dao interface="com.ibatis.jpetstore.persistence.iface.AccountDao"
      implementation="com.ibatis.jpetstore.persistence.sqlmapdao.AccountSqlMapDao"/>
    ...
  </context>
</daoConfig>
  
```

其中 AccountDao 接口的 insertAccount 方法代码如下。

```

public interface AccountDao {
  ...
  void insertAccount(Account account);
  ...
}
  
```

AccountSqlMapDao 实现类的 insertAccount 方法代码如下。

```

public class AccountSqlMapDao extends BaseSqlMapDao implements AccountDao {

    public AccountSqlMapDao(DaoManager daoManager) {
        super(daoManager);
    }
    ....
    public void insertAccount(Account account) {
        update("insertAccount", account);
        update("insertProfile", account);
        update("insertSignon", account);
    }
    ....
}

```

实现步骤说明:

(1) 当调用某个 dao, Service 层的代码如下:

```

DaoManager daoMgr = DaoConfig.getDaoManager();
this.accountDao = (AccountDao) daoMgr.getDao(AccountDao.class);
accountDao.insertAccount(account);

```

(2) 程序转到了 DaoProxy 动态代理对象的 invoke 方法, 代码如下:

```

public Object invoke(Object proxy, Method method, Object[] args)
    throws Throwable {
    Object result = null;
    //这是处理一些常规的方法, 如 equals、getClass 等方法。
    if (PASSTHROUGH_METHODS.contains(method.getName())) {
        try {
            result = method.invoke(daoImpl.getDaoInstance(), args);
        } catch (Throwable t) {
            throw ClassInfo.unwrapThrowable(t);
        }
    } else {
        StandardDaoManager daoManager = daoImpl.getDaoManager();
        DaoContext context = daoImpl.getDaoContext();

        if (daoManager.isExplicitTransaction()) {
            // Just start the transaction (explicit)
            //针对已经调用了 daoManager 的 startTransaction 方法的, 没有采取一次性提交, 只是启动事务
            try {
                context.startTransaction();
                result = method.invoke(daoImpl.getDaoInstance(), args);
            } catch (Throwable t) {
                throw ClassInfo.unwrapThrowable(t);
            }
        } else {
            //针对单个的 dao, 一次性进行 Start, commit and end the transaction (autocommit)
            try {
                //启动事务
                context.startTransaction();

```



```

//业务处理
result = method.invoke(daoImpl.getDaoInstance(), args);
//提交事务
context.commitTransaction();
} catch (Throwable t) {
throw ClassInfo.unwrapThrowable(t);
} finally {
context.endTransaction();
}
}
}
return result;
}

```

在程序代码中，获得了基于配置文件 dao 节点 DaoImpl 实例化对象，接着获得了 StandardDaoManager 实例化对象和 DaoContext 实例化对象。

```

StandardDaoManager daoManager = daoImpl.getDaoManager();
DaoContext context = daoImpl.getDaoContext();

```

下面是进行业务的处理，代码如下：

```

...
context.startTransaction();
result = method.invoke(daoImpl.getDaoInstance(), args);
context.commitTransaction();
...

```

(1) 我们先来分析 context.startTransaction 最后转化成了什么代码

context 用来启动事务，即上述代码的 context.startTransaction，我们来看看这些代码，代码如下，实际上转移到了 DaoTransaction 实例化对象的 startTransaction 方法。

```

public void startTransaction() {
if (state.get() != DaoTransactionState.ACTIVE) {
DaoTransaction trans = transactionManager.startTransaction();
transaction.set(trans);
state.set(DaoTransactionState.ACTIVE);
daoManager.addContextInTransaction(this);
}
}

```

我们知道这个 DaoTransaction 实例化对象就是 SqlMapDaoTransactionManager 实例化对象。然后我们去看看它的 startTransaction 方法。

```

public DaoTransaction startTransaction() {
return new SqlMapDaoTransaction(client);
}

```

SqlMapDaoTransactionManager 对象创建一个对象并返回，我们转移到 SqlMapDaoTransaction 构造函数，去看看到底发生了什么情况，代码如下：

```
public SqlMapDaoTransaction(SqlMapClient client) {
    try {
        client.startTransaction();
        this.client = client;
    } catch (SQLException e) {
        throw new DaoException("Error starting SQL Map transaction. Cause: " + e, e);
    }
}
```

在上述代码中，client 变量就是 SqlMapClient 实例化变量。这个变量是如何产生的呢？就是在解析配置文件 dao.xml 中的 transactionManager 节点，产生 SqlMapDaoTransactionManager 实例化对象时，在 SqlMapDaoTransactionManager 的内部函数中产生的一行代码如下。

```
client = SqlMapClientBuilder.buildSqlMapClient(reader, properties)
```

实际上进了这么多重的转化，最后是实现了如下的代码：

```
sqlMapClient.startTransaction();
```

(2) 然后我们分析 result = method.invoke(daoImpl.getDaoInstance(), args) 实现了什么。当进行业务提交，即调用 method.invoke(daoImpl.getDaoInstance(), args) 方法。针对本 DaoProxy 实例化代理对象，其 daoImpl.getDaoInstance 返回的是 AccountSqlMapDao 实例化对象。调用的方法是 insertAccount，参数是对象 account。

那我们去看看 AccountSqlMapDao 的 insertAccount 方法内容。

```
public void insertAccount(Account account) {
    update("insertAccount", account);
    update("insertProfile", account);
    update("insertSignon", account);
}
```

转移到父类 SqlMapDaoTemplate 的 update 方法，我们接着过去看看。

```
public int update(String id, Object parameterObject) {
    try {
        return getSqlMapExecutor().update(id, parameterObject);
    } catch (SQLException e) {
        throw new DaoException("Failed to update - id [" +
            id + "] - parameterObject [" + parameterObject + "]. Cause: " + e, e);
    }
}
```

而其中的 getSqlMapExecutor() 到底返回什么呢？我们继续往下看。

```
protected SqlMapExecutor getSqlMapExecutor() {
    SqlMapDaoTransaction trans = (SqlMapDaoTransaction) daoManager.getTransaction(
        this);
    return trans.getSqlMap();
}
```

我们看看 `daoManager.getTransaction(this)`。

```
public DaoTransaction getTransaction(Dao dao) {
    DaoImpl impl = (DaoImpl) daoImplMap.get(dao);
    return impl.getDaoContext().getTransaction();
}
```

这个 `daoImplMap.get(dao)` 返回的就是配置文件 `dao` 节点（如下）所形成的 `DaoImpl` 实例化对象。

```
<dao interface="com.ibatis.jpetstore.persistence.iface.AccountDao"
    implementation="com.ibatis.jpetstore.persistence.sqlmapdao.
AccountSqlMapDao"/>
```

这个对象的 `getDaoContext()` 就是配置文件 `context` 节点（如下）实例化的 `DaoContext` 对象。

```
<context>
    <transactionManager type="SQLMAP">
        <property name="SqlMapConfigResource"
            value="com/ibatis/jpetstore/persistence/sqlmapdao/sql/sql-map-config.xml"/>
    ...
</context>
```

这个 `DaoContext` 对象也就是第一步骤中 `context` 变量，这个 `SqlMapDaoTransaction` 实例化对象同样也是在第一步骤中。我们来看看 `DaoContext` 对象的 `getTransaction()`。

```
public DaoTransaction getTransaction() {
    startTransaction();
    return (DaoTransaction) transaction.get();
}

public void startTransaction() {
    if (state.get() != DaoTransactionState.ACTIVE) {
        DaoTransaction trans = transactionManager.startTransaction();
        transaction.set(trans);
        state.set(DaoTransactionState.ACTIVE);
        daoManager.addContextInTransaction(this);
    }
}
```

如果 `DaoContext` 对象已经启动了事务，那么就调用 `transactionManager.startTransaction()`，获得新创建的 `DaoTransaction` 实例化对象并返回，否则，就采用 `transaction` 线程上的 `DaoTransaction` 实例化对象。这是采用的单例设计模式。

而 `SqlMapDaoTransaction` 实例化对象是返回什么呢？

```
public SqlMapClient getSqlMap() {
    return client;
}
```

结果还是由 `SqlMapDaoTransactionManager` 实例化对象时, 在 `SqlMapDaoTransactionManager` 的内部函数中产生基于 `SqlMapClient` 接口实现的实例化对象。

而这个 `SqlMapClient` 实例化对象与第一步骤产生的 `SqlMapClient` 实例化对象是同一个对象。

`DaoProxy` 实例化代理对象的这一系列处理, 最终还是转化成如下操作。

```
sqlMapClient.update(id, parameterObject);
```

(3) 再然后我们看看 `context.commitTransaction()` 又实现了什么

`context` 用来启动事务, 即上述代码的 `context.commitTransaction()`, 该代码实际上转移到了 `DaoTransaction` 实例化对象的 `commitTransaction` 方法。同时对 `state` 线程进行了修改。

```
public void commitTransaction() {
    DaoTransaction trans = (DaoTransaction) transaction.get();
    if (state.get() == DaoTransactionState.ACTIVE) {
        transactionManager.commitTransaction(trans);
        state.set(DaoTransactionState.COMMITTED);
    } else {
        state.set(DaoTransactionState.INACTIVE);
    }
}
```

我们知道, 这个 `DaoTransaction` 实例化对象就是 `SqlMapDaoTransactionManager` 实例化对象。然后我们去看看它的 `commitTransaction` 方法。

```
public void commitTransaction(DaoTransaction trans) {
    ((SqlMapDaoTransaction) trans).commit();
}
```

程序转移到 `SqlMapDaoTransaction` 的 `commit`, 代码如下:

```
public void commit() {
    try {
        client.commitTransaction();
        client.endTransaction();
    } catch (SQLException e) {
        throw new DaoException("Error committing SQL Map transaction. Cause: " + e, e);
    }
}
```

又回到了 `SqlMapClient` 实例化对象的处理。

实际上进行了这么多重的转化, 最后是实现了如下的代码:

```
sqlMapClient.commitTransaction();
sqlMapClient.endTransaction();
```

(4) 最后我们看看 context.endTransaction()做了什么工作

主要是最后清理现场,把当前 transaction 线程内容清空,把当前 state 线程的内容赋为 INACTIVE。

```
public void endTransaction() {
    DaoTransaction trans = (DaoTransaction) transaction.get();
    if (state.get() == DaoTransactionState.ACTIVE) {
        try {
            transactionManager.rollbackTransaction(trans);
        } finally {
            state.set(DaoTransactionState.ROLLEDBACK);
            transaction.set(null);
        }
    } else {
        state.set(DaoTransactionState.INACTIVE);
        transaction.set(null);
    }
}
```

这就是整个业务的处理过程。为了更加清楚地说明这里面发生的一切,可用协作图来说明。把这整个过程分为三个部分,主要是转化给 DaoProxy 动态代理对象的 invoke 方法。第一部分是 DaoProxy invoke 方法启动事务,第二部分是 DaoProxy invoke 方法进行业务操作,第三部分是 DaoProxy invoke 方法提交事务。其实可以用三个协作图来说明到底发生了什么事情。

对于第 1 步骤的协作图如图 4-33 所示。DaoProxy 对象调用 context 对象的 startTransaction 方法,一直最后到 SqlMapClient 对象的 startTransaction 方法。

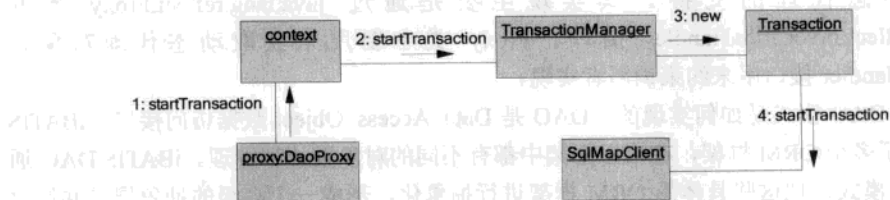


图 4-33 DaoProxy 启动事务协作图

第 2 步骤协作图如图 4-34 所示。DaoProxy 对象调用 daoImpl 对象的 getDaoInstance 方法,获得外部 Dao 的实现类,然后调用外部 Dao 的实现类的 insertAccount 方法,经过了 3、4、5、6 等步骤,一直到获取外部 Dao 当前的 SqlMapClient 对象,最后调用 SqlMapClient 对象的 update 方法,实现业务操作。

第 3 步骤协作图如图 4-35 所示。DaoProxy 对象调用 context 对象的 commitTransaction 方法,最后到 SqlMapClient 对象的 endTransaction 方法。

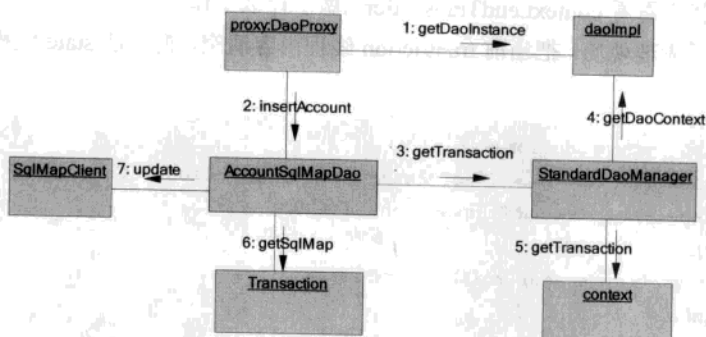


图 4-34 DaoProxy 业务操作协作图

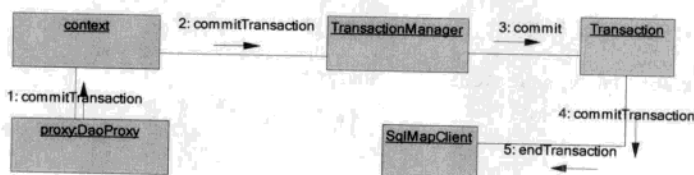


图 4-35 DaoProxy 提交事务协作图

4.6 读取源码的收获

通过我们对 iBATIS DAO 源码的理解和分析，收获如下。

第一，如何实现接口和实现分离。在应用程序中，为了达到松耦合，常常要把业务的表述和具体的实现相分离。在 iBATIS DAO 中是通过动态代理来实现的。Java 开发包中包含了对动态代理的支持。其实现主要是通过 `java.lang.reflect.Proxy` 类和 `java.lang.reflect.InvocationHandler` 接口。Proxy 类主要用来获取动态代理对象，InvocationHandler 接口用来约束调用者实现。

第二，DAO 模式是如何实现的。DAO 是 Data Access Object 数据访问接口。iBATIS DAO 集成了多个 ORM 框架，在这里框架中都有不同的对象设计和管理。iBATIS DAO 通过模板设计模式，把这些具体的 ORM 框架进行抽象化，形成一个高级的抽象层，并针对各个具体 ORM 框架定义模板。

第三，如何集成多个 ORM 框架的事务管理和异常处理。iBATIS DAO 提供的 DAO（数据访问对象）支持的主要目的是便于以标准的方式使用不同的数据访问技术，如 JDBC、JTA、Hibernate、Apache OJB 或者 TopLink 等。它不仅可以在这些持久化技术间切换，而且开发人员在编码的时候不用考虑处理各种技术中特定异常。尤其是对于事务管理的抽象，源码也显示了如何通过接口来分离事务的不同实现。在这个实现中采用了桥梁模式。

第四，如何对 XML 文件进行验证。通过 iBATIS DAO DTD 文件进行了验证的 `dao.xml` 才是合法的 XML 文件。iBATIS DAO 集合 DOM 和 SAX 来解析和验证 XML 文件，这些技术都值得我们借鉴。

PART

三

第三部分 iBATIS 的底层平台—— iBATIS SQL Map 的分析

第 5 章 iBATIS SQL Map 体系结构和剖析

第 6 章 SQL Map 配置信息的读取

第 7 章 SQL Map 引擎实现框架

第 8 章 SQL Map 数据库处理

第 9 章 SQL Map 中 Mapping 实现

第 10 章 SQL Map 缓存管理和实现

第 11 章 TypeHandler 类型转化

第 12 章 iBATIS 常用工具的实现



第 5 章

iBATIS SQL Map 体系结构和剖析

本章内容:

1. 介绍 SQL Map 实现的功能和原理;
2. 介绍 SQL Map 的包结构和文件结构;
3. 介绍 SQL Map 的组件结构。

SQL Map 是 iBATIS Database Layer 框架的重要组成部分,是整个 iBATIS Database Layer 的核心价值所在。iBATIS SQL Map 提供一个简单的框架,它的最大优势就是简单。SQL Map 使用一个简单的 XML 文件来实现从实体到 SQL Statements 的映射,在连接表或复杂查询时只要自由地使用 SQL 语句即可解决,在 SQL Map 应用编程接口上也只需建立对象和 PreparedStatement 参数或者 resultMap 之间的映射。本章主要介绍 iBATIS SQL Map 的体系结构。

5.1 SQL Map 实现的功能和原理

按照 SQL Map 的标准定义,SQL Map 提供了一个简洁的框架,使用简单的 XML 描述文件将 Java Bean、Map 实现和基本数据类型的包装类 (String、Integer 等) 映射成 JDBC 的输入参数,然后进行 SQL 处理,处理的结果也是按照 Java Bean、Map 实现和基本数据类型的包装类 (String、Integer 等) 来进行输出。

SQL Map 的实现框架结构如图 5-1 所示。外部 Java Application 调用 SqlMapClient 接口,经过 SqlMapSession、SqlMapExecutorDelegate 等实现类,通过 PersistentObject、ParameterMap、ResultMap、MappedStatement、Transaction 等,基于 JDBC、JTA 等事务操作,最后调用 SqlExecutor 操作类对数据库进行操作。

再详细一点的解释就是,SQL Map 框架可以专门用于 O/R 映射,O/R 映射是 JavaBean 对象到数据库中关系表的映射。SQL Map 框架使用 XML 描述符将 JavaBean 映射到 SQL

语言, 执行 SQL 并将结果映射返回对象, 其核心功能是围绕 MappedStatement 进行的。MappedStatement 实质上是一个 XML 节点元素, 可以拥有 Parameter Map 属性、ResultMap 属性和 SQL 文本。而 ParameterMap 和 ResultMap 实质上是一个 XML 节点元素, 其中, ParameterMap 为 MappedStatement 提供数据输入参数, ResultMap 为 MappedStatement 提供数据输出, 且 ResultMap 允许将 MappedStatements 映射成为 JavaBean 对象的方式。在实现 O/R 时, 通常是通过配置 SqlMapConfig.xml 来定义总体变量、事务管理模式、连接数据库信息, 同时, 指定包含 MappedStatements、ResuMap、ParameterMap 的映射文件位置等内容。然后, 通过 SQL Map API 将 JavaBeans 对象映射到 PreparedStatement 参数和 ResultSet, 并完成持久化操作。

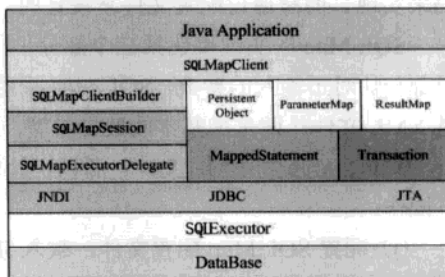


图 5-1 SQL Map 框架结构

SQL Map 生命周期流程的描述是将一个对象作为参数 (对象可以是 Java Bean、Map 实现和基本类型的包装类), 而参数对象则将为 SQL 修改语句和查询语句设定参数值。具体过程是:

① 执行 mapped statement。这是 SQL Maps 最重要的步骤。SQL Map 框架将创建一个 PreparedStatement 实例, 用参数对象为 PreparedStatement 实例设定参数, 执行 PreparedStatement 并从 ResultSet 中创建结果对象。

② 执行 SQL 的更新数据语句时, 返回受影响的数据行数。执行查询语句时, 将返回一个结果对象或对象的集合。与参数对象一样, 结果对象可以是 Java Bean、Map 实现和基本数据类型的包装类。

基于 SQL Map 的数据持久层的三个基本实现步骤如下:

① 提供一个或多个参数, 这些参数将作为 SQL 语句执行或存储过程运行时的输入值。

② 通过传送的参数和配置文件 SqlMap.xml 中的描述来执行映射。框架将会准备 SQL 声明或者存储过程, 传送的参数设置为运行时的数据值, 执行 SQL 语句或者存储过程, 返回结果。

③ 返回一个结果对象或对象的集合, 与参数对象一样, 结果对象可以是 Java Bean 等多种类型。

也可以这么来理解 iBATIS 的 SQL Map。SQL Map 就是一个简化版本的 SQL 工具, 其实现与 JDBC 的实现内容是相似的。比如, 我们用 JDBC 来对数据库进行操作; 必须要进行如下的操作。

① 加载 (注册) 适当的 JDBC 驱动程序;

② 建立数据库连接;

③ 建立符合 JDBC 规范的 SQL 语句和输入参数;

④ 执行 SQL 语句;

- ⑤ 处理结果集;
- ⑥ 关闭数据库连接。

SQL Map 对数据库进行操作的步骤也是一样的。不过, 参数信息并不是通过硬编码来实现、而是通过配置文件来设置的。

SQL Map 的实现要包括三个部分: ① SQL 语句部分; ② 输入参数部分; ③ 输出结果部分。对于输入的参数, SQL Map 采用了四种数据格式。一是 JavaBean, 二是 Map, 三是 XML 文件, 四是最基本的数据类型。SQL Map 在执行操作后有三种输出格式, 分别为 JavaBean、Map 和 XML 文件。所以, 针对上述 JDBC 操作, 转化为 SQL Map 就变成了如下操作:

- ① 配置 SQL Map 配置文件, 载入 JDBC 驱动程序和数据库连接 (包括事务管理模式和 DataSource 模式) 等信息;
- ② 配置 SQL Map 映射文件, 包括 parameterMap、resultMap、SQL 等信息。其中: parameterMap 是输入参数部分, resultMap 是输出结果部分, sql 是 SQL 语句部分;
- ③ 根据 SQL Map 配置文件配置信息, 加载配置的 JDBC 驱动程序;
- ④ 根据 SQL Map 配置文件, 建立数据库连接;
- ⑤ 根据 SQL Map 映射文件中的 sql 信息和 parameterMap 信息, 建立符合 JDBC 规范的 SQL 语句和输入参数;
- ⑥ 执行 SQL 语句;
- ⑦ 处理结果集, 把数据库结果集转化为 SQL Map 映射文件中的 resultMap 格式内容;
- ⑧ 关闭数据库的连接。

5.2 SQL Map 组件的包结构和文件结构

SQL Map 组件包括两个大包, client 包和 engine 包。Client 包主要是客户端接口的核心库。engine 包是 SQL Map 组件实现映射功能的核心库。SQL Map 组件在使用过程中要依赖 com.ibatis.common 包, SQL Map 组件覆盖的包名称如表 5-1 所示。

表 5-1 SQL Map 的包名称及说明

序 号	包 名 称	说 明
1	com.ibatis.sqlmap.engine.accessplan	Java 对象和数据库数据的转化处理
2	com.ibatis.sqlmap.engine.builder	解析配置文件, 形成配置信息
3	com.ibatis.sqlmap.engine.cache	缓存实现
4	com.ibatis.sqlmap.engine.config	配置文件的转化实现
5	com.ibatis.sqlmap.engine.datasource	数据库的 datasource 处理
6	com.ibatis.sqlmap.engine.exchange	Java 对象和数据库数据的转化处理
7	com.ibatis.sqlmap.engine.execution	对数据库操作的执行类
8	com.ibatis.sqlmap.engine.impl	用户调用接口的实现类

续表

序 号	包 名 称	说 明
9	com.ibatis.sqlmap.engine.mapping	映射处理
10	com.ibatis.sqlmap.engine.scope	用户进行调用的接口的状态包
11	com.ibatis.sqlmap.engine.transaction	数据库的事务处理
12	com.ibatis.sqlmap.engine.type	数据类型转化

5.3 SQL Map 的组件结构

基于组件的开发是框架程序开发的变体。分层是从逻辑上将功能子系统划分成许多组件集合，而层间关系的形成要遵循一定的规则。通过分层，可以限制功能子系统间的依赖关系，使系统以更松散的方式耦合，从而更易于建设、维护和进化。组件也是一种组合关系，由大颗粒组件包容中颗粒组件，中颗粒组件再包含小颗粒组件。这样形成一个有机、层次、条理分明的组件群体。

SQLMap 也采用了面向构件设计的原则，其组件结构包括了 8 个组件群，其中 Common 是通用的基础组件库，其他任何组件库都要对这个组件库进行调用。其他 7 个组件群有机地联系在一块，每个组件群里包含实现特定业务功能的组件，按照高内聚低耦合的设计原则进行组合。组件群以 Java 包为单位。

第 1 个组件群是 Client 接口部分，提供给外部的接口部分，也就是我们使用 SQLMap 所面对的 API 或者接口。

第 2 个组件群是 Configuration 配置组件，其功能主要是读取配置文件和形成配置信息的组件，包括 Builder 组件和 config 组件。

第 3 个组件群是 implement 组件，其功能是针对 Client 接口的具体实现 Impl 组件，包含 impl 包和 scope 包的内容。

第 4 个组件群是用于数据库处理部分，包含事务管理的 transaction 组件、数据源 datasource。

第 5 个组件群是 Map 组件群。功能主要是覆盖 SQL 的映射部分，包括 mapping 组件和 exchange 组件。

第 6 个组件群是缓存处理部分，主要是 Cache 组件，以实现对象的缓存管理。

第 7 个组件群是数据类型处理组件，主要是 type 组件，其主要的功能是对数据库数据类型、JDBC 数据类型和。

第 8 个组件群是 Common 组件，在这个组件群中包括了一些最基本的应用工具，如 I/O 操作、Util 工具、Beans 管理、日志管理、XML 处理和资源管理等。

这 8 个组件覆盖的包内容如表 5-1 所示。这 8 个组件之间的关系如图 5-2 所示。

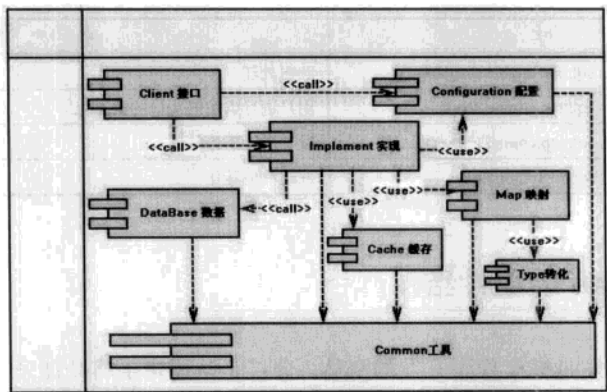


图 5-2 SQL Map 组件结构

在这里要对上述图 5-2 进行描述：外部应用程序调用 SQLMap 组件，主要是与 Client 接口组件交互而实现具体的业务功能。Client 接口只是充当了 SQLMap 组件的门户。Client 组件调用 Configuration 配置组件和实现类组件——implement 组件。implement 组件是整个 SQLMap 组件的核心，该组件属于中心调度组件，由它来协调其他各个组件的任务顺序和工作安排。首先该组件通过 Configuration 配置组件获得要实现业务的引擎信息，然后通过调用 DataBase 数据库组件来对数据库和数据库事务进行管理，通过调用 Map 映射组件来实现 SQL 语句的生成和执行，并转化执行后获得的结果。通过调用 Cache 缓存组件来实现对象的缓存管理。Map 映射组件调用 Type 数据类型转化组件来实现各种数据类型的转换。同时 Implement 实现组件、Configuration 配置组件、DataBase 数据库组件、Map 映射组件、Cache 缓存组件、Type 数据类型转化组件都依赖 Common 基础组件，都是基于该组件提供的基础功能来实现特定的任务。组件和 Java package 的对应关系如表 5-2 所示。

表 5-2 SQL Map 各个组件与包的对应关系

序 号	组 件	覆盖的包	说 明
1	Client 接口组件	com.ibatis.sqlmap.client	用户进行调用的接口
2	Implement 实现组件	com.ibatis.sqlmap.engine.impl com.ibatis.sqlmap.engine.scope	用户调用接口的实现类
3	Configuration 配置组件	com.ibatis.sqlmap.engine.builder.XML com.ibatis.sqlmap.engine.config	解析配置文件，形成配置信息
4	DataBase 数据库组件	com.ibatis.sqlmap.engine.datasource com.ibatis.sqlmap.engine.transaction com.ibatis.common.jdbc	数据库处理，包括事务和 SQL 内容
5	Map 映射组件	com.ibatis.sqlmap.engine.mapping com.ibatis.sqlmap.engine.exchange	映射处理
6	Cache 缓存组件	com.ibatis.sqlmap.engine.Cache	缓存处理
7	Type 数据类型转化组件	com.ibatis.sqlmap.engine.type	数据类型转换和转化
8	Common 基础组件	com.ibatis.common	通用工具和基础类

第 6 章

SQL Map 配置信息的读取

本章内容:

1. XML 文件的验证概述和处理说明;
2. SQL Map 如何读取 SQL Map XML 配置文件;
3. SQL Map 如何读取 SQL Map XML 映射文件;
4. 用一个实例来抽象出通用的 XML 解析框架。

任何一个框架,为了提高其适应性和扩展性,一般都提供可以编辑的配置文件作为其可扩充性的手段和方式。配置文件成为了一个灵活性程序框架必不可少的组成部分。由于现在 XML 文件格式的普及,所以一般情况下程序框架都采用基于 XML 格式来定义的配置文件。iBATIS SQL Map 也不例外。对于 SQL Map 配置信息的读取,主要涉及三个方面的话题:采用什么技术?如何进行验证?读取内容的方式?对于 SQL Map 配置信息文件的读取,采用了 Java API for XML Processing (JAXP),包括里面覆盖的 DOM 和 SAX 技术。iBATIS SQL Map 的配置 XML 文件都要求支持对 DTD 的验证。

6.1 XML 文件的验证处理

6.1.1 XML 验证处理的通用模式

验证是可以快速检查输入 XML 文档是否大体上符合预期的形式,它可以立刻拒绝与处理目标相距甚远的文档。如果数据中存在问题,及早发现总是有益于后期使用的。对于可扩展标记语言(XML)来说,验证就是采用各种模式语言为文档内容编写详细的规范,这些模式语言包括万维网联盟(W3C)的 XML Schema Language (XSD)、RELAX NG、文档类型定义(DTD)和 Schematron 等。有时候验证在解析的同时进行,有时候在解析完成后立刻进行。但一般情况下,都在对输入的其他处理之前完成。

XML 验证的具体应用程序编程接口(API)随着模式语言和解析器的不同而不同。DTD 和 XSD 是 Simple API for XML (SAX)、文档对象模型(DOM)和 Java API for XML Processing (JAXP) 常见的配置选项。RELAX NG 需要自定义的库和 API。Schema 可以使用 Transformations API for XML (TrAX), 还有其他模式也要求程序员学习更多的 API, 尽管执行的操作基本相同。

XML 验证是优秀文档创作的基准。给出 XML 文档含意的关键就是拥有一组控制文档的约束以及确保遵循这些约束, 这也是验证的要点。默认情况下, XML 语法分析器中的验证通常都是关闭的, 这是因为很多不严谨的 XML 作者都不编写约束, 将 XML 语法分析器验证关闭可帮助避免产品环境中的冗长处理。要打开验证, 必须明确请求验证开启。

基于语法的验证语言, 例如 XML Schema 和 DTD, 可以很好地确保 XML 文档遵从定义良好的消息结构。这样可确保接收 XML 消息的应用程序能够正确地处理接收到的 XML 消息, 但是不能保证包含在消息中的数据是有效的。例如, 基于语法的验证语言的这些局限性意味着, 必须使用不同的方法来验证变量和外部数据集上的同现约束(co-occurrence constraints) 和其他约束。

Java API for XML Processing (JAXP) 一直是一种稳定、可靠的 XML 处理 API。在一般的 JAXP 处理中, 都是从工厂开始的。SAXParserFactory 用于 SAX 解析, DocumentBuilderFactory 则用于 DOM 解析。这两种工厂都使用静态方法 newInstance() 创建。对工厂设置的选项影响该工厂创建的所有解析器。如果用 true 调用 setValidating(), 则要明确地告诉工厂创建的所有解析器都必须是进行验证的。

虽然 SAXParserFactory 和 DocumentBuilderFactory 有分别适合 SAX 和 DOM 的不同特性和性质, 但是对验证来说它们都有一个共同的方法: setValidating。要打开验证, 只须把 true 传递给该方法。但是使用工厂是为了创建解析器而不是直接解析文档。创建工厂之后就可以调用 newSAXParser (SAX) 或 newDocumentBuilder (DOM)。无论哪种情况, 都将得到一个能够解析 XML 并在解析过程中验证 XML 的对象 (SAXParser 或 DocumentBuilder)。这样仅限于 DTD 解析。setValidating(true)调用对基于 XML 的解析完全没有作用。

错误处理模式的默认响应方式是, 如果遇到问题则抛出 SAXException, 否则什么也不做。但是, 可以提供 SAX ErrorHandler 来接收关于文档问题的更详尽的信息。比方说, 假设要记录所有验证错误, 但又不希望遇到错误时停止处理。由于 iBATIS 采用 XML DTD 文档结构模式, 所以要采用 XML DTD 验证。其通用验证 DTD 的做法有个以下四个步骤。

- ① 首先把要标准的 DTD 文件放置在某一个可以找到的位置。
- ② 定义一个实现类, 实现接口 EntityResolver。目的是实现把 Internet 网的 DTD 转化到本地。
- ③ 在读取 XML 文档的时候 (无论是采取 DOM、SAX), 把要验证的标志加入到阅读器中。
- ④ 错误处理模式, 一般采用 SAX 获得异常, 然后对异常进行处理。

这里要简单地介绍一下 SAX EntityResolver 接口。在进行 XML 分析时, 分析器使用

DTD 或 XML 模式中指定的位置, 解析 XML 文档中的外部实体引用。解析期间, 分析器会找到被引用的内容并将它插入 XML 中。需要一个活动的网络连接以使解析能够正确进行, 因为有许多被引用的实体都引用了位于某处的一个远程 URL。解析 (打开连接、下载内容、关闭连接等) 也会减慢分析进程的速度。开发者可能想知道是否存在一种方法在本地提供了被引用内容的高速缓存副本, 或有另一种方法能绕过实体解析进程。

SAX 定义了一个接口 `org.xml.sax.EntityResolver`, 这个接口只定义了一个方法, 代码如下所示:

```
package org.xml.sax;

public interface EntityResolver {

    public InputSource resolveEntity(String publicID, String systemID) throws
    SAXException;

}
```

这个接口中的唯一方法 `resolveEntity()` 提供了一种开始实体解析进程的方法。由于每个外部实体引用在指定如何解析内容的 DTD 中都有一个公共标识或一个系统标识, 或同时拥有这两者, 所以可以在这个方法中匹配它们并实现自己的行为。例如, 下面的 DTD 部分定义了 `copyright` 外部实体引用。DTD 中定义的 `copyright` 外部实体引用代码如下。

```
<!ENTITY copyright SYSTEM "http://www.ibm.com/developerworks/copyright.xml">
```

这里, 没有公共标识, 系统标识是 `http://www.ibm.com/developerworks/copyright.XML`。所以, 可以创建一个名为 `CopyrightEntityResolver` 的类, 代码如下所示。

清单 3. 实现 EntityResolver

```
package com.ibm.developerWorks;
import org.xml.sax.EntityResolver;
import org.xml.sax.InputSource;
import org.xml.sax.SAXException;

public class CopyrightResolver implements EntityResolver {

    public InputSource resolveEntity(String publicID, String systemID) throws
    SAXException {

        if (systemID.equals("http://www.ibm.com/developerworks/copyright.xml")) {
            // Return local copy of the copyright.xml file
            return new InputSource("/usr/local/content/localCopyright.xml");
        }

        // If no match, returning null makes process continue normally
        return null;
    }

}
```

在这个简单的实现中, 在每次解析实体时都会调用 `resolveEntity` 方法。如果实体的系统标识匹配该方法中的 URL, 则返回本地的 XML 文档 (`localCopyright.xml`), 而不是返回位于提供的系统标识处的任何资源。使用这种方法, 可以“缩短”处理的进程, 并向给定的公共标识或系统标识提供开发者自己的数据。当不匹配时, 开发者会希望始终确保返回 `null`, 这样实体解析在非特殊情况下都可以正常进行。以上就是对实体引用解析的介绍。

可以将实体解析器注册在分析器上，代码如下所示。

清单 4. 实现 EntityResolver

```
// Get an XML Reader - this code not detailed here
XMLReader reader = XMLReaderFactory.createXMLReader();
reader.setEntityResolver(new CopyrightResolver());
reader.parse(new InputSource("article.xml"));
```

这样一来就行了。如果可以获取实体引用内容的本地副本，或者需要用自己的内容替代实体引用，可以使用 SAX EntityResolver 接口，这将有助于提高应用程序的速度并增加 XML 文档的灵活性。

6.1.2 iBATIS SQL Map 的 XML 验证

在 iBATIS SQL Map 读取 XML 文件采用了三种模式，即 org.w3c.dom 软件包、org.xml.sax 软件包和 JAXP 包。从技术角度来讲，org.w3c.dom 软件包主要是对文档和节点进行处理，org.xml.sax 软件包主要是进行验证和从外部 DTD 文件中获取相关的配置信息。

一般比较正式的 XML 信息中都会包含对应的 DTD 声明，用来定义该 XML 文档中的格式。SQL Map 的 SqlMapConfig.xml 配置文件中有这么一句。

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

同理，在 SQL Map 的映射文件中有这么一句，道理与上述的 SqlMapConfig.xml 是相同的。

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

这需要对 SQL Map 的 SqlMapConfig.xml 中内容进行解析时，需要--//ibatis.apache.org//DTD SQL Map Config 2.0//EN"或者是 http://ibatis.apache.org/dtd/sql-map-config-2.dtd 来处理。但如果我们的计算机不能保证时都连在网上，那么怎么能保证解析过程不出错呢？在 iBATIS 中就是采用 org.xml.sax 软件包来实现的，这样同样存在于 SQL Map 的映射文件中。

这说明既包括了公共标识，也有系统标识。公共标识为"--//ibatis.apache.org//DTD SQL Map Config 2.0//EN"和"--//ibatis.apache.org//DTD SQL Map 2.0//EN"，系统标识为"http://ibatis.apache.org/dtd/sql-map-config-2.dtd" 和

SQL Map 采用了 SqlMapClasspathEntityResolver 类和预先将 DTD 文件放置在本地目录来实现。

SqlMapClasspathEntityResolver 类用于实现 EntityResolver 接口，其初始化变量代码如下。

```
private static final String SQL_MAP_CONFIG_DTD = "com/ibatis/sqlmap/engine/builder
/xml/sql-map-config-2.dtd";
```

```

private static final String SQL_MAP_DTD = "com/ibatis/sqlmap/engine/builder/xml/sql-
map-2.dtd";

private static final Map doctypeMap = new HashMap();

static {
    doctypeMap.put("http://www.ibatis.com/dtd/sql-map-config-2.dtd".toUpperCase(), SQL_MAP_CONFIG_DTD);
    doctypeMap.put("http://ibatis.apache.org/dtd/sql-map-config-2.dtd".toUpperCase(), SQL_MAP_CONFIG_DTD);
    doctypeMap.put("-//IBATIS.com//DTD SQL Map Config 2.0//EN".toUpperCase(), SQL_MAP_CONFIG_DTD);
    doctypeMap.put("-//ibatis.apache.org//DTD SQL Map Config 2.0//EN".toUpperCase(), SQL_MAP_CONFIG_DTD);

    doctypeMap.put("http://www.ibatis.com/dtd/sql-map-2.dtd".toUpperCase(), SQL_MAP_DTD);
    doctypeMap.put("http://ibatis.apache.org/dtd/sql-map-2.dtd".toUpperCase(), SQL_MAP_DTD);
    doctypeMap.put("-//IBATIS.com//DTD SQL Map 2.0//EN".toUpperCase(), SQL_MAP_DTD);
    doctypeMap.put("-//ibatis.apache.org//DTD SQL Map 2.0//EN".toUpperCase(), SQL_MAP_DTD);
}

```

这样可以把 SqlMapConfig.xml 中出现的要解析的 DTD 文件都映射到本地目录 com/ibatis/sqlmap/engine/builder/XML/指定的 iBATIS 的 DTD 文件中。

首先在 com/ibatis/sqlmap/engine/builder/xml/ 目录下有 sql-map-config-2.dtd 文件和 sql-map-2.dtd 这两个文件。把所有关于 DTD 的信息都转化为本地目录的 sql-map-config-2.dtd 的文件和 sql-map-2.dtd。而 sql-map-config-2.dtd 和 sql-map-2.dtd 的文件格式见附录二。该类的目的就是要把本来从 http://www.ibatis.com/dtd/sql-map-config-2.dtd、http://ibatis.apache.org/dtd/sql-map-config-2.dtd、-//IBATIS.com//DTD SQL Map Config 2.0//EN、-//ibatis.apache.org//DTD SQL Map Config 2.0//EN 等地址中读取 DTD 的信息改为了从当前类路径中读取，也就是 com/ibatis/sqlmap/engine/builder/xml/sql-map-config-2.dtd，通过重新定义 InputSource 来返回 sql-map-2.dtd 数据流，从而让 xml 解析器不必从网上获取 DTD 信息。

sql-map-2.dtd 也是采用同样方式来实现的，其实现接口 resolveEntity 方法的程序代码如下。

```

public InputSource resolveEntity(String publicId, String systemId) throws
SAXException {
    if (publicId != null) publicId = publicId.toUpperCase();
    if (systemId != null) systemId = systemId.toUpperCase();
    InputSource source = null;
    try {
        String path = (String) doctypeMap.get(publicId);
        source = getInputSource(path, source);
        if (source == null) {

```

```

        path = (String) doctypeMap.get(systemId);
        source = getInputSource(path, source);
    }
} catch (Exception e) {
    throw new SAXException(e.toString());
}
return source;
}

private InputSource getInputSource(String path, InputSource source) {
    if (path != null) {
        InputStream in = null;
        try {
            in = Resources.getResourceAsStream(path);
            source = new InputSource(in);
        } catch (IOException e) {
            // ignore, null is ok
        }
    }
    return source;
}
}

```

在启动 XML 的 doc 是载入 SqlMapClasspathEntityResolver 实例化对象。

```
parser.setEntityResolver(new SqlMapClasspathEntityResolver());
```

在创建 JAXP Document 是加载，代码如下。

```

private Document createDocument(InputStream inputStream) throws
    ParserConfigurationException, FactoryConfigurationError, SAXException,
    IOException {
    DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();
    factory.setValidating(validation);

    factory.setNamespaceAware(false);
    factory.setIgnoringComments(true);
    factory.setIgnoringElementContentWhitespace(false);
    factory.setCoalescing(false);
    factory.setExpandEntityReferences(true);

    DocumentBuilder builder = factory.newDocumentBuilder();
    builder.setEntityResolver(entityResolver);
    builder.setErrorHandler(new ErrorHandler() {
        public void error(SAXParseException exception) throws SAXException {
            throw exception;
        }

        public void fatalError(SAXParseException exception) throws SAXException {
            throw exception;
        }

        public void warning(SAXParseException exception) throws SAXException {

```

```

    }
    });

    return builder.parse(new InputSource(inputStream));
}

```

在 Java 5.0 (和 JAXP 1.3) 中, JAXP 引进一种新方法来自验证文档。不是仅仅在 SAX 或 DOM 工厂上使用 `setValidating()` 方法, 而是将验证划分到新 `javax.xml.validation` 软件包中的几个类中, 通过 `org.xml.sax.InputSource` 来进行文档的验证。

坦白地说, JAXP 验证的验证是 SAX 和 DOM 的结合。所以验证也可以使用 DOM 来整体解析 XML, 并结合 SAX 的 `ErrorHandler` 类, 通过巧妙的编程也能对验证错误进行即时处理。但是这需要对 SAX 有充分的了解, 需要很多时间去测试和调试并且仔细地管理内存 (如果最终创建 DOM Document 对象的话)。这正是 JAXP 验证 API 闪光的地方。它提供了一种经过认真测试的、可以随时使用的解决方案, 而不仅仅是是否启用模式验证的一个开关。它很容易与已有的 JAXP 代码结合在一起, 增加模式验证非常简单。

6.2 解析 SQL Map 配置文件

对 SQL Map XML 配置文件的读取, 我们分几个步骤来说明。首先介绍 SQL Map XML 配置文件的格式, 然后说明 iBATIS 是如何来读取配置文件, 并把配置文件的配置信息转化成应用系统中的对象和变量的。

6.2.1 SqlMapConfig.xml 的格式说明

`SqlMapConfig.xml` 是 SQL Map 的主配置文件, 一般配置的样例如下。

```

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

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

<sqlMapConfig>
    <properties
        resource="examples/sqlmap/maps/SqlMapConfigExample.properties" />
    <settings cacheModelsEnabled="true" enhancementEnabled="true"
        lazyLoadingEnabled="true" maxRequests="32" maxSessions="10"
        maxTransactions="5" useStatementNamespaces="false" />
    <typeAlias alias="order" type="testdomain.Order" />
    <transactionManager type="JDBC">
        <dataSource type="SIMPLE">
            <property name="JDBC.Driver" value="${driver}" />
            <property name="JDBC.ConnectionURL" value="${url}" />
            <property name="JDBC.Username" value="${username}" />

```

```

<property name="JDBC.Password" value="{password}" />
<property name="JDBC.DefaultAutoCommit" value="true" />
<property name="Pool.MaximumActiveConnections" value="10" />

<property name="Pool.MaximumIdleConnections" value="5" />
<property name="Pool.MaximumCheckoutTime" value="120000" />
<property name="Pool.TimeToWait" value="500" />
<property name="Pool.PingQuery"
    value="select 1 from ACCOUNT" />
<property name="Pool.PingEnabled" value="false" />
<property name="Pool.PingConnectionsOlderThan" value="1" />
<property name="Pool.PingConnectionsNotUsedFor" value="1" />
</dataSource>
</transactionManager>

<sqlMap resource="examples/sqlmap/maps/Person.xml" />
</sqlMapConfig>

```

SqlMapConfig.xml 采用 DTD 格式来进行验证，其 DTD 格式参考附录二。XML DTD 是以前使用最广泛的老式 XML 模式，而如今的 XML Schema 则已经成为 W3C 的正式推荐标准，是目前使用最广泛的 XML 模式，并有替代 XML DTD 的趋势。本文仍采用 XSD 来描述其结构。

SqlMapConfig.xml 的 XSD 组成如图 6-1 所示，其 XSD 的代码参考附录三。

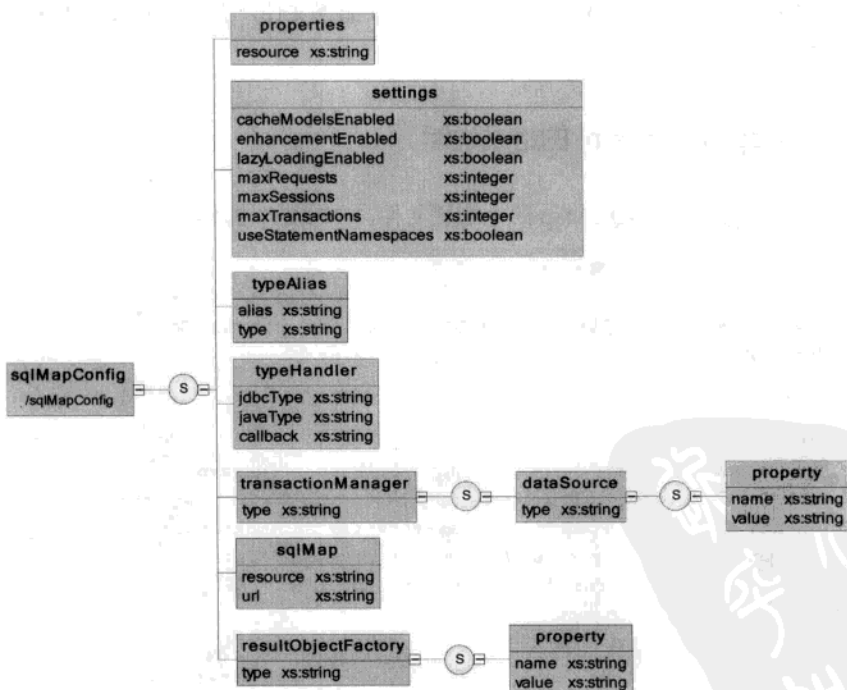


图 6-1 SQL Map 主配置文件的 XSD 模型

在图 6-1 中, SqlMapConfig.xml 文件的根节点是 sqlMapConfig 节点。sqlMapConfig 根节点包括 7 个子节点, 分别是 properties、settings、typeAlias、typeHandler、transactionManager、SQLMap 和 resultObjectFactory 等。在这 7 个子节点中, 稍微复杂一点的是 transactionManager 节点。transactionManager 节点包括了 dataSource 节点, 而 dataSource 节点还包含了 property 节点。这些节点的基本属性简单描述如下。

(1) <properties>元素: SQL Map 配置文件拥有唯一的<properties>元素, 用于在配置文件中使用的标准的 Java 属性文件 (name=value)。这样做后, 在属性文件中定义的属性可以作为变量在 SQL Map 配置文件及其包含的所有的 SQL Map 映射文件引用。

(2) <settings>元素: <setting>元素用于配置和优化 SqlMapClient 实例的各选项。<setting>元素本身及其所有的属性都是可选的。

(3) <typeAlias>元素: <typeAlias>元素可以为一个通常较长的、全限定类名指定一个较短的别名。

(4) <typeHandler>元素, 在官方的用户手册中没有介绍, 该节点用于创建新的数据库数据类型、Java 数据类型和 jdbc 数据类型的映射。

(5) <transactionManager>元素可以为 SQL Map 配置事务管理服务。属性 type 指定所使用的事务管理器类型。这个属性值可以是一个类名, 也可以是一个别名。包含在框架的三个事务管理器分别是: JDBC、JTA 和 EXTERNAL。

(6) <datasource>元素: <datasource>是<transactionManager>的一部分, 为 SQL Map 数据源设置了一系列参数。目前 SQL Map 架构只提供三个 DataSource Factory, 包括 SimpleDataSourceFactory、DbcpDataSourceFactory、JndiDataSourceFactory。但也可以添加自己的实现。

(7) <sqlMap>元素: <sqlMap>元素用于包括 SQL Map 映射文件和其他的 SQL Map 配置文件。每个 SqlMapClient 对象使用的所有 SQL Map 映射文件都要在此声明。映射文件作为 stream resource 从类路径或 URL 读入。必须在这里指定所有的 sql Map 文件。

(8) <resultObjectFactory>元素: 在官方的用户手册中没有介绍, iBATIS 的使用了这个接口创造了一个语句的执行结果对象的实现。如果要使用该元素, 在 sqlMapConfig 元素类型 “resultObjectFactory” 指定一个实现类。如果要对这个接口实现, 必须有一个 public 无参数的构造方法。

6.2.2 SqlMapConfig.xml 文件读取总体说明

一般过程的 XML 文件解析。我们在 iBATIS DAO 模式上已经了解到 Java 解析 XML 文件的一般方式。基本上的实现思路是这样的。采用 DOM 方式, 从 XML 的根节点开始, 然后循环查找根节点下面的第一个子节点, 如果子节点还有子子节点, 再通过循环查找子子节点。采用这种递归方式来遍历所有的节点和节点属性。但是, SQL Map 对于 XML 文件的解析, 有一套特殊的方法和实现手段。

SQL Map 解析 XML 文件主要是由 SqlMapConfigParser、NodeletParser、XmlParserState、Nodelet、SqlMapConfiguration 等几个核心类来实现的。其中 NodeletParser 类是一个通用处理 XML 文件节点的解析类,而 SqlMapConfigParser 是主要处理 SQL Map 主配置文件的类。在解析过程中,SqlMapConfigParser 采用了一个匿名内部类的方式来实现。这是一种 Gof 23 设计模式中的策略模式,读取的时候是按照递归算法来实现的。NodeletParser 类增加了内部静态 Path 来创建、生产和定位路径。根据节点的路径来解析 XML 节点,当访问到一个节点时,即处理一个节点内属性及其下属的节点信息。

对 SqlMapConfig.xml 文件会涉及这些类,包括 SqlMapClientBuilder 类、SqlMapConfigParser 类、NodeletParser 类、XmlParserState 类、Nodelet 类、SqlMapConfiguration 类、SqlMapClientImpl 类、SqlMapExecutorDelegate,其类结构如图 6-2 所示。由图 6-2 可见,SqlMapClientBuilder 类,依赖 SqlMapConfigParser 类和 SqlMapClientImpl 类。其依赖性表现为 SqlMapClientBuilder 类,调用 SqlMapConfigParser 的方法而获得 SqlMapClientImpl 的实例化对象。SqlMapConfigParser 类关联 NodeletParser 类,NodeletParser 类,用于文件的 I/O 操作,其关联 SqlMapClasspathEntityResolver 类,主要用于文件 DTD 验证。SqlMapConfigParser 类关联 XmlParserState 类,XmlParserState 类,又关联 SqlMapConfiguration 类。SqlMapConfiguration 类关联 SqlMapExecutorDelegate 类,最后 SqlMapExecutorDelegate 类关联 SqlMapClientImpl 类。而 SqlMapClientImpl 类的实例化对象 SqlMapClientImpl 对象正是客户端所需要获得 Object。

通过类结构图的关联关系分析,由一个 SqlMapConfigParser 对象可以获取一个 XmlParserState 对象,而 XmlParserState 对象可以获取一个 SqlMapConfiguration 对象,而 SqlMapConfiguration 对象可以获取一个 SqlMapExecutorDelagate 对象,最后 SqlMapExecutorDelergate 对象可以获取一个 SplMapClientImpl 对象。这种方式是参照对象关系模式来进行推导的。

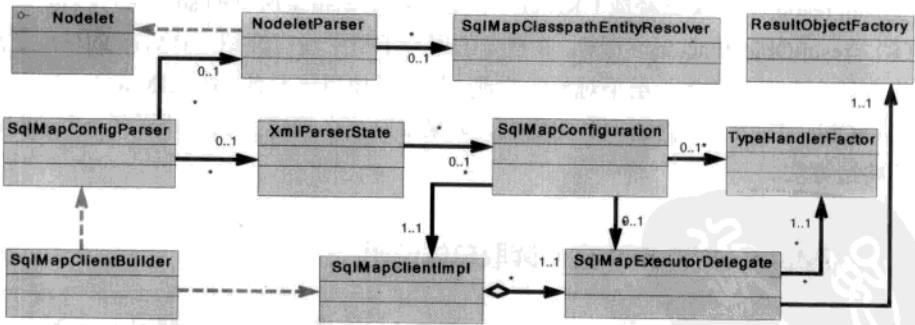


图 6-2 SQL Map 解析主配置 XML 的类结构图

配置文件在读取的过程中,基本上创建要进行业务处理的其他全部类和对象。所以,创建配置信息过程就是一个建立整个系统的初始化过程,为了便于说明,这里只是列出其中涉及的比较重要的对象。其读取过程如图 6-3 所示。

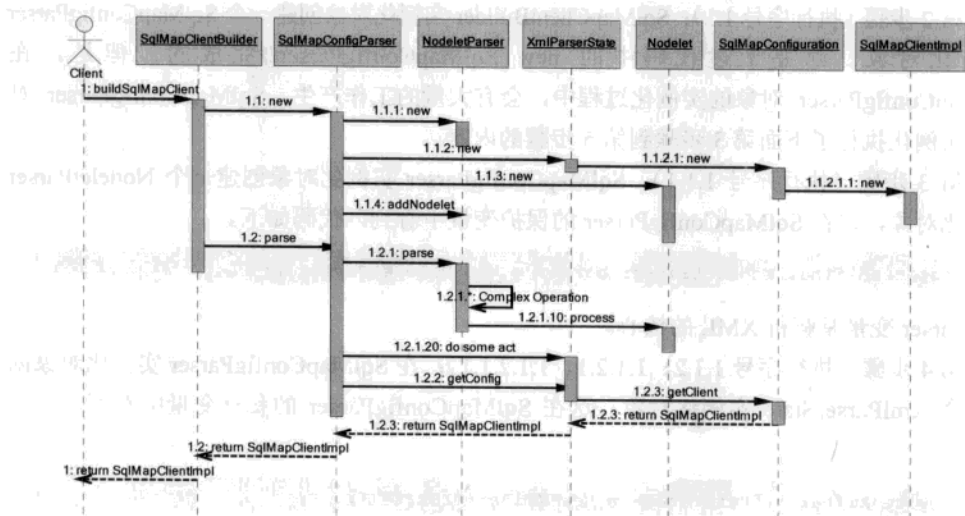


图 6-3 SQL Map 解析主配置 XML 的序列图

这里只是列出主要的类和核心操作。这是按照对象产生的先后顺序来描述整个应用程序执行过程的。客户端调用 `SqlMapClientBuilder` 实例化对象的 `buildSqlMapClient` 方法，`SqlMapClientBuilder` 的 `buildSqlMapClient` 方法创建出 `SqlMapConfigParser` 实例化对象，`SqlMapConfigParser` 对象生成中创建 `NodeletParser` 对象和 `XmlParserState` 对象，在其构造过程中调用了很多方法，但核心是实例化很多 `Nodelet` 对象，同时调用了 `NodeletParser` 对象的 `addNodelet` 方法，把 `Nodelet` 对象等加到 `NodeletParser` 对象的 `HashMap` 变量 `letMap` 中。`SqlMapClientBuilder` 的 `buildSqlMapClient` 方法再次调用 `SqlMapConfigParser` 实例化对象的 `parse` 方法。而 `SqlMapConfigParser` 实例化对象的 `parse` 方法首先是调用 `NodeletParser` 对象的 `parse` 方法。当然，`NodeletParser` 对象 `parse` 方法的执行内容比较复杂，但是其一定要完成两項工作，一项工作是要让实例化的 `Nodelet` 对象执行自身的 `process` 方法，另一项工作是把执行过程中获得的信息转移给 `XmlParserState` 对象。`SqlMapConfigParser` 实例化对象然后再调用 `XmlParserState` 对象的 `getConfig` 方法获得 `SqlMapConfiguration` 对象，接着调用 `SqlMapConfiguration` 对象的 `getClient` 方法获得 `SqlMapClientImpl` 对象。这样，这个 `SqlMapClientImpl` 对象就是客户端调用 `SqlMapClientBuilder` 实例化对象想要获得的结果。

上面的描述从一个总体框架来说明应用程序的执行过程，但是总的来说还是比较抽象的，下面分别对其进行详细说明：

第 1 步骤（执行序号 1）：客户端调用 `SqlMapClientBuilder` 实例化对象的 `buildSqlMapClient` 方法，`SqlMapClientBuilder` 类中代码如下。

```
public static SqlMapClient buildSqlMapClient(Reader reader) {
    return new SqlMapConfigParser().parse(reader);
}
```

第2步骤(执行序号 1.1): `SqlMapClientBuilder` 实例化对象创建一个 `SqlMapConfigParser` 实例化对象。这是上述代码中的 `new SqlMapConfigParser()` 完成的。但是, 在 `SqlMapConfigParser` 对象的实例化过程中, 会有大量的工作产生, `SqlMapConfigParser` 对象的实例化执行了下面第3步骤到第5步骤的内容。

第3步骤(执行序号 1.1.1): `SqlMapConfigParser` 实例化对象创建一个 `NodeletParser` 实例化对象, 这在 `SqlMapConfigParser` 的保护变量中看到, 代码如下。

```
protected final NodeletParser parser = new NodeletParser();
```

`parser` 变量是解析 XML 的核心。

第4步骤(执行序号 1.1.2、1.1.2.1、1.1.2.1.1): 在 `SqlMapConfigParser` 实例化对象创建一个 `XmlParserState` 实例化对象, 这在 `SqlMapConfigParser` 的私有变量中看到, 代码如下。

```
private XmlParserState state = new XmlParserState();
```

`state` 变量充当一个中间变量的容器, 或者是一个配置信息的门户。`XmlParserState` 实例化对象在生成过程中, 会创建 `SqlMapConfiguration` 对象。这在 `XmlParserState` 类的属性中看到, 代码如下。

```
private SqlMapConfiguration config = new SqlMapConfiguration();
```

在 `SqlMapConfiguration` 对象实例化过程中, 它又创建了 `SqlMapClientImpl` 实例化对象。这在 `SqlMapConfiguration` 类的属性中和构造函数中看到, 代码如下。

```
...
private SqlMapExecutorDelegate delegate;
private SqlMapClientImpl client;
...

public SqlMapConfiguration() {
    ...
    delegate = new SqlMapExecutorDelegate();
    client = new SqlMapClientImpl(delegate);
    ...
}
```

第5步骤(执行序号 1.1.3 和 1.1.4): 在 `SqlMapConfigParser` 实例化对象的构造函数中有如下的方法。

```
public SqlMapConfigParser() {

    parser.setValidation(true);
    parser.setEntityResolver(new SqlMapClasspathEntityResolver());

    addSqlMapConfigNodelets();
    addGlobalPropNodelets();
}
```

```

addSettingsNodelets();
addTypeAliasNodelets();
addTypeHandlerNodelets();
addTransactionManagerNodelets();
addSqlMapNodelets();
addResultObjectFactoryNodelets();
}

```

前面两个方法的用途是为了进行 DTD 文件规范化验证。

后面方法是针对每个节点进行处理，以 `addSQLMapConfigNodelets` 方法为例。

```

private void addSqlMapConfigNodelets() {
    parser.addNodelet("/sqlMapConfig/end()", new Nodelet() {
        public void process(Node node) throws Exception {
            state.getConfig().finalizeSqlMapConfig();
        }
    });
}

```

这些方法最终是要实现创建一个 `Nodelet` 对象（匿名对象），同时调用 `parser` 对象的 `addNodelet` 方法，如上述代码中的黑体所示。

第 6 步骤（执行序号 1.2）：`SqlMapClientBuilder` 实例化对象调用 `SqlMapConfigParser` 实例化对象的 `parse` 方法。

第 7 步骤（执行序号 1.2）：`SqlMapConfigParser` 实例化对象调用 `NodeletParser` 实例化对象 `parser` 的 `parse` 方法，代码如下。

```

public SqlMapClient parse(InputStream inputStream) {
    try {
        usingStreams = true;
        parser.parse(inputStream);
        return state.getConfig().getClient();
    } catch (Exception e) { throw new RuntimeException("Error occurred. Cause: "
        + e, e);
    }
}

```

第 8 步骤（执行序号 1.2.1.*）：`NodeletParser` 实例化对象的 `parse` 方法执行的内容非常的丰富。这里只是说明其中比较重要的两个操作，一个是调用 `Nodelet` 对象的 `process` 方法。在 XML 配置文件中，有多少个节点，就会调用多少个针对该节点新建 `Nodelet` 对象的 `process` 方法。另一个是调用 `XmlParserState` 对象的数据赋值方法。在调用 `Nodelet` 对象的 `process` 方法，其主要的功能还是通过 `XmlParserState` 对象来进行数据的传输和赋值。而 `XmlParserState` 对象也只是一个中转站，其目标数据汇总地还是实例化的 `SqlMapConfiguration` 对象。

第 9 步骤（执行序号 1.2.2）：`SqlMapConfigParser` 对象调用 `XmlParserState` 对象的 `getConfig` 方法，获得 `SqlMapConfiguration` 对象。

第 10 步骤（执行序号 1.2.3）：`XmlParserState` 对象调用 `SqlMapConfiguration` 对象的

getClient 方法，获得 SqlMapClientImpl 对象。然后 SqlMapConfiguration 对象把 SqlMapClientImpl 对象传递给 XmlParserState 对象，XmlParserState 对象接着把 SqlMapClientImpl 对象传递给 SqlMapConfigParser 对象，SqlMapConfigParser 对象接着把 SqlMapClientImpl 对象传递给 SqlMapClientBuilder 对象，然后 SqlMapClientBuilder 对象把 SqlMapClientImpl 对象传递给客户端。也就是说，要调用函数所得到的 SQLMapClient 对象。

在这个步骤中的第 4 步和第 10 步是非常重要的处理步骤，也是这里面最难理解的地方，下面做详细的说明。

读取和解析 SqlMapConfig.xml 文件要涉及的接口或类，如表 6-1 所示。

表 6-1 解析 SqlMapConfig.xml 文件要涉及的接口或类

接口或类	功能描述
com.ibatis.sqlmap.client.SqlMapClientBuilder	客户端获得 SqlMapClient 对象的工厂类
com.ibatis.sqlmap.engine.builder.xml.SqlMapConfigParser	SQL Map 配置文件的主解析类
com.ibatis.sqlmap.engine.builder.xml.SqlMapClasspathEntityResolver	SQL Map 配置文件和映射文件的 EntityResolver 类，实现 EntityResolver 接口，用于把验证时采用本地的 dtd 文件
com.ibatis.common.xml.NodeletParser	iBATIS 的通用解析 XML 节点的节点解析类，属工具类范畴
com.ibatis.sqlmap.engine.builder.xml.XmlParserState	用于解析 XML 节点时的连接类，主要是进行节点内节点之间的信息关联，属于存储中间状态的数据结构类，类似于 Javabean
com.ibatis.common.xml.Nodelet	一个 XML 节点的接口
com.ibatis.sqlmap.engine.config.SqlMapConfiguration	配置的一级门面对象，在 SQL Map 配置文件的信息基本上都通过它与具体应用做关联。其主要的作用还是向 SQLMap ExecutorDelegate 对象传递配置数据信息
com.ibatis.sqlmap.engine.impl.SqlMapClientImpl	SqlMapClientImpl 是实现外部调用 SqlMapClient 接口的实现类，但是其本身不做具体代码操作，所有工作都转移到 SQL MapExecutorDelegate 对象
com.ibatis.sqlmap.engine.impl.SqlMapExecutorDelegate	应用的一级门面对象，在 iBATIS SQL Map 的所有应用操作基本上都要在这里交汇、调度、转移。该对象保留了所有的 SQL Map 配置文件和映射文件的信息

6.2.3 基于设计模式中策略模式的数据执行

在数据实例化过程中，采用了 Gof 23 设计模式中的策略模式。下面简单阐述一下 Gof 23 设计模式中的策略模式。

策略 (Strategy) 模式属于对象的行为模式，其标准定义是：定义一系列的算法，把它们一个个封装起来，并且使它们可相互替换。本模式使得算法可独立于使用它的客户而变化，通过分析 Strategy 模式可以发现：策略模式针对一组算法，将每一个算法封装到具有共同接口的独立的类中，从而使得它们可以相互替换。策略模式使得算法可以在不影响客户端的情况下发生变化，还可把行为和环境分开。环境类负责维持和查询行为类，各种算法在具体的策略类中提供。由于算法和环境独立开来，算法的增减和修改都不会影响到环

境和客户端。

在软件设计中，经常会遇到因为一开始考虑不周而导致后面遇到很大的麻烦。开发者只是想尽快地解决问题而没考虑到后期的维护或者根本没法预料到将来会有什么样的变化干脆置之不理。针对这个问题，Strategy 模式提供了一个解决问题的方案：面向接口编程而不是实现，发现其中会改变的部分作为扩展点，然后把它封装起来。

Strategy 结构如图 6-4 所示，其角色包括环境（Context）角色、抽象策略（Strategy）角色和具体策略（Concrete Strategy）角色。这些角色说明如下：

- ① 环境（Context）角色：拥有一个 Strategy 接口的引用。环境（Context）角色可以是接口、具体类或抽象类。该角色不可缺少。
- ② 抽象策略（Strategy）角色：这是一个抽象角色，此角色给出所有的具体策略类所需的统一接口，或者是定义所有支持的算法的公共接口。通常由一个接口或抽象类实现。
- ③ 具体策略（Concrete Strategy）角色：包装了相关的算法或行为。具体策略（Concrete Strategy）角色都是由具体类来实现的，以 Strategy 接口实现某具体算法。

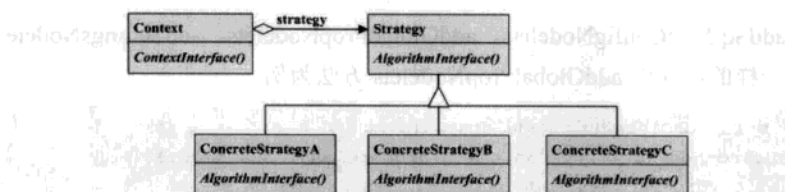


图 6-4 策略模式结构

当客户调用 Context 时，它将客户的请求转发给它的 Strategy。Context 将该算法所需的所有数据传递给 Strategy，或者将自身作为参数传递给 Strategy，使 Strategy 可以回调 Context，由 ConcreteStrategy 实现具体算法。

在本系统中涉及两个类和一个接口。SqlMapConfigParser 类、NodeletParser 类和 Nodelet 接口。其中 Nodelet 接口就是抽象策略（Strategy）角色。NodeletParser 类可以算作一个环境（Context）角色。SqlMapConfigParser 类中新创建的匿名类就是具体策略（Concrete Strategy）角色。SqlMapConfigParser 类属于 Client 角色。为与上述的标准策略模式结构图相对应起见，做出了一个 Sql Map 的配置文件实现结构类图（如图 6-5 所示）。

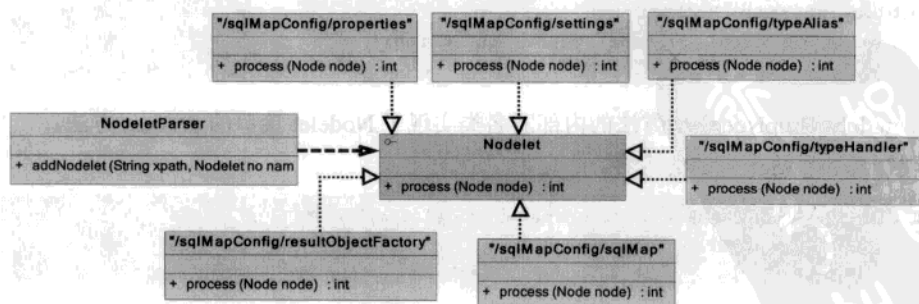


图 6-5 SQL Map 的策略设计模式类结构图

策略设计模式具体是如何实现的呢？

NodeletParser 类中有一个 HashMap 类型的私有变量 letMap。SqlMapConfigParser 类在实例化构造中，有如下方法。

```
public SqlMapConfigParser() {
    parser.setValidation(true);
    parser.setEntityResolver(new SqlMapClasspathEntityResolver());

    addSqlMapConfigNodelets();
    addGlobalPropNodelets();
    addSettingsNodelets();
    addTypeAliasNodelets();
    addTypeHandlerNodelets();
    addTransactionManagerNodelets();
    addSqlMapNodelets();
    addResultObjectFactoryNodelets();
}
```

其中，addSqlMapConfigNodelets、addGlobalPropNodelets、addSettingsNodelets 等方法的结构都是一样的，现以 addGlobalPropNodelets 方法为例。

```
private void addGlobalPropNodelets() {
    parser.addNodelet("/sqlMapConfig/properties", new Nodelet() {
        public void process(Node node) throws Exception {
            Properties attributes = NodeletUtils.parseAttributes(node, state.getGlobal
Props());
            String resource = attributes.getProperty("resource");
            String url = attributes.getProperty("url");
            state.setGlobalProperties(resource, url);
        }
    });
}
```

在这个方法中，由 NodeletParser 类实例化的 parser 变量调用了 addNodelet 方法。addNodelet 方法中的第二个参数是基于 Nodelet 接口创建了一个内部匿名类。我们先看看 Nodelet 接口内容。

```
public interface Nodelet {
    void process (Node node) throws Exception;
}
```

addGlobalPropNodelets 方法的内部匿名类实现了 Nodelet 接口的 process 方法。现在，看看这个方法的内容。

```
public void addNodelet(String xpath, Nodelet nodelet) {
    letMap.put(xpath, nodelet);
}
```

第 1 个参数 xpath，表示一个路径，针对节点的不同，分别为：

- 当节点内容是节点，则 Path 结构为：/rootElement/childElement/theElement;
- 当节点内容是属性，则 Path 结构为：/rootElement/childElement/@theAttribute;
- 当节点内容是文本，则 Path 结构为：/rootElement/childElement/text();
- 当节点已经完成，则 Path 结构为：/rootElement/childElement/end()。

第 2 个参数是实现 Nodelet 的实例化对象，也就是 NodeletParser 类中方法创建的内部匿名类。

把这个路径参数和内部匿名类保存到 HashMap 类型的变量 letMap 中。

当应用程序在调用如下方法的时候。

```
addSqlMapConfigNodelets();
```

实际上就是在 letMap 变量中加入一条数据。该数据的 key 为 Path 内容，该数据的 value 为 Nodelet 的实例化对象。在 letMap 变量进行初始化配置，该配置就是把不同的节点有不同的处理策略结合起来，形成一个标准的节点信息处理模板。而这些节点如果来执行匿名类的 process 方法，则还要到工具类 NodeletParser 去看看如何实现。

首先要回到递归来处理 XML 文件，其代码如下。

```
private void process(Node node, Path path) {
    if (node instanceof Element) {
        // Element
        String elementName = node.getNodeName();
        path.add(elementName);
        processNodelet(node, path.toString());
        processNodelet(node, new StringBuffer("/").append(elementName).toString());

        // Attribute
        NamedNodeMap attributes = node.getAttributes();
        int n = attributes.getLength();
        for (int i = 0; i < n; i++) {
            Node att = attributes.item(i);
            String attrName = att.getNodeName();
            path.add("@ " + attrName);
            processNodelet(att, path.toString());
            processNodelet(node, new StringBuffer("/@").append(attrName).toString());
            path.remove();
        }

        // Children
        NodeList children = node.getChildNodes();
        for (int i = 0; i < children.getLength(); i++) {
            process(children.item(i), path);
        }
        path.add("end()");
        processNodelet(node, path.toString());
        path.remove();
        path.remove();
    } else if (node instanceof Text) {
```

```
// Text
path.add("text()");
processNodelet(node, path.toString());
processNodelet(node, "//text()");
path.remove();
}
}
```

要理解这个方法，还要结合一个实现方法 `processNodelet`。

```
private void processNodelet(Node node, String pathString) {
    Nodelet nodelet = (Nodelet) letMap.get(pathString);
    if (nodelet != null) {
        try {
            nodelet.process(node);
        } catch (Exception e) {
            throw new RuntimeException("Error parsing XPath '" + pathString + "'. Cause: " + e, e);
        }
    }
}
```

即是在对应的每个节点，都要执行下列的方法。

```
String elementName = node.getNodeName();
path.add(elementName);
processNodelet(node, path.toString());
processNodelet(node, new StringBuffer("/").append(elementName).toString());

//属性其他方法
// Attribute
....

//下级节点处理
....

path.add("end()");
processNodelet(node, path.toString());
path.remove();
path.remove();
```

对于 `Path` 类，我已经在前面做了说明，主要是实现路径。

当执行 `path.add(elementName)` 时，在 `path` 类的变量中加入了这个节点的名称。

当执行 `processNodelet(node, path.toString())` 时，调用 `Nodelet nodelet = (Nodelet) letMap.get(pathString)`。

这个 `nodelet` 就是初始化时候实例化的内部匿名对象，它是按照路径来进行保存的。在 `SqlMapConfigParser` 实例化对象的 `addGlobalPropNodelets` 方法中，`path` 是 `/sqlMapConfig/properties`，而 `nodelet` 变量是如下代码。

```
public void process(Node node) throws Exception {
```

```

Properties attributes = NodeletUtils.parseAttributes(node, state.getGlobalProps());
String resource = attributes.getProperty("resource");
String url = attributes.getProperty("url");
state.setGlobalProperties(resource, url);
}

```

接着调用 `nodelet.process(node)` 方法。`Node` 就是 XML 配置文件中的 `/sqlMapConfig/properties` 节点。然后根据这个节点来进行 `process` 方法。

这就是整个实现过程。看上去非常难以理解。但是我们采用设计模式中的策略模式就好说明了。首先定义了一个策略接口，这个接口只有一个方法，即 `process` 方法。然后根据 XML 配置文件不同的节点，采用不同的处理方式，这些处理方式就是 `SqlMapConfigParser` 实例化对象各个方法中的内部匿名对象。

6.2.4 基于递归和路径来实现配置文件的全部遍历

基于递归和路径来实现配置文件的全部遍历这一点主要是在类 `NodeletParser` 中，代码（为了阅读清晰，简化了部分代码）如下：

```

private void process(Node node, Path path) {
    if (node instanceof Element) {
        // 对 Element 进行处理
        ....
        // 对 Attribute 进行处理
        ....
        // Children
        NodeList children = node.getChildNodes();
        for (int i = 0; i < children.getLength(); i++) {
            //进行递归
            process(children.item(i), path);
        }
        // 对当前节点进行处理
        ....
    } else if (node instanceof Text) {
        // 对 Text 进行处理
        ....
    }
}

```

上述方法也是基于 DOM 技术解析 XML 文档的典型方法。

在递归过程中，通过 `NodeletParser` 类内部的静态 `Path` 类来实现不同的路径。我们来分析一下静态 `Path` 类的代码。

```

private static class Path {
    private List nodeList = new ArrayList();
    public Path() {}

    public Path(String path) {

```

```

StringTokenizer parser = new StringTokenizer(path, "/", false);
while (parser.hasMoreTokens()) {
    nodeList.add(parser.nextToken());
}

}

public void add(String node) { nodeList.add(node); }
public void remove() { nodeList.remove(nodeList.size() - 1); }
public String toString() {
    StringBuffer buffer = new StringBuffer("/");
    for (int i = 0; i < nodeList.size(); i++) {
        buffer.append(nodeList.get(i));
        if (i < nodeList.size() - 1) {
            buffer.append("/");
        }
    }
    return buffer.toString();
}
}

```

在 Path 类中有一个 ArrayList 类型的私有变量 nodeList。

当构造时传入了一个路径，就根据路径的前后顺序按照规则是以“/”当作分隔符存入到 nodeList 中的。比如，传入的参数是/sqlMapConfig/transactionManager/dataSource。那在 nodeList 中就加入了三个 String 变量，分别是 sqlMapConfig、transactionManager 和 dataSource。

而方法 add(String node)是加入一个路径节点。按照上例中 nodeList 已经有 sqlMapConfig、transactionManager 和 dataSource 等三个变量，如果再进行加入，采用如下方法。

```
Path.add("end()")
```

那么在 nodeList 中就用了四个 String 变量，分别是 sqlMapConfig、transactionManager、dataSource 和 end()。

而 remove 方法是删掉最后一个路径节点。按照上例中 nodeList 已经有 sqlMapConfig、transactionManager、dataSource 和 end 四个变量，如果再进行删除，则采用如下方法。

```
Path.remove()
```

那么在 nodeList 中就回到了三个 String 变量，分别是 sqlMapConfig、transactionManager 和 dataSource。

方法 toString 是把 nodeList 中的节点生成路径。比如有 nodeList 中有三个 String 变量，分别是 sqlMapConfig、transactionManager 和 dataSource。调用 toString 方法，则返回 /sqlMapConfig/transactionManager/dataSource。

6.2.5 XmlParserState 对象在解析 SQL Map XML 配置文件的协调者角色

XmlParserState 实例化对象在整个解析配置文件过程中，充当了协调者角色。关于调停者（Mediator）模式，我们可以参考 GoF 23 种设计模式的调停者（Mediator）模式。

调停者（Mediator）模式标准定义：用一个中介对象来封装一系列的对象交互。中介者使各对象不需要显式地相互引用，从而使其耦合松散，而且可以独立地改变它们之间的交互。

调停者（Mediator）模式也叫中介者模式，属于对象行为型模式。调停者模式包装了一系列对象相互作用的方式，使得这些对象不必相互明显作用。从而使他们可以松散耦合。当某些对象之间的作用发生改变时，不会立即影响其他一些对象之间的作用，保证这些作用可以彼此独立地变化。调停者模式将多对多地相互作用转化为一对多的相互作用，将类与类之间复杂的相互关系封装到一个调停者类中。调停者模式将对象的行为和协作抽象化，把对象在小尺度的行为上与其他对象的相互作用分开处理。

调停者（Mediator）模式结构如图 6-6 所示，其角色包括抽象调停者（Mediator）角色、具体调停者（Concrete Mediator）角色、抽象同事类（Colleague）角色和具体同事类（Concrete Colleague）角色等。这些角色说明如下：

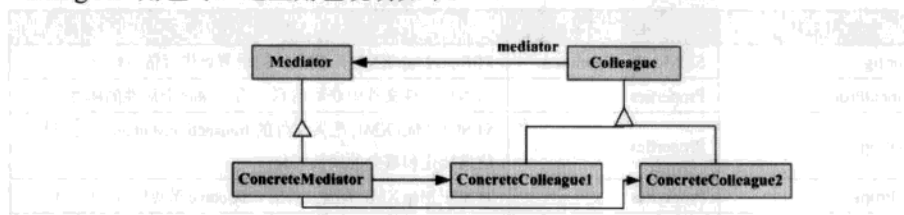


图 6-6 调停者模式结构

① 抽象调停者（Mediator）角色：定义出同事对象到调停者对象的接口，其中主要的方法是一个（或者多个）事件方法。在有些情况下，这个抽象对象可以省略。一般而言，这个角色由一个抽象类或者接口实现。

② 具体调停者（Concrete Mediator）角色：从抽象调停者继承而来，实现了抽象超类所声明的事件方法。具体调停者知晓所有的具体同事类，它从具体同事对象接收消息、向具体同事对象发出命令。一般而言，这个角色由一个具体类实现。

③ 抽象同事类（Colleague）角色：定义出调停者到同事对象的接口。同事对象只知道调停者而不知道其余的同事对象。一般而言，这个角色由一个抽象类或者对象实现。

④ 具体同事类（Concrete Colleague）角色：所有的具体同事类均从抽象同事类继承而来。每一个具体同事类都很清楚它自己在小范围的行为，而不知道它在大范围内的目的。在示意性的类图中，具体同事类是 Colleague1 和 Colleague2。一般而言，这个角色由一个具体类实现。

我们来看看一级门面 XmlParserState 的属性和构造方法吧。

```
public class XmlParserState {
    private SqlMapConfiguration config = new SqlMapConfiguration();
    private Properties globalProps = new Properties();
    private Properties txProps = new Properties();
    private Properties dsProps = new Properties();
    private Properties cacheProps = new Properties();
    private boolean useStatementNamespaces = false;
    private Map sqlIncludes = new HashMap();

    private ParameterMapConfig paramConfig;
    private ResultMapConfig resultMapConfig;
    private CacheModelConfig cacheConfig;

    private String namespace;
    private DataSource dataSource;
    .....
}
```

对 XmlParserState 的属性做一个简单的介绍，具体如表 6-2 所示。

表 6-2 XmlParserState 的属性列表和说明

序 号	变量名称	类 别	说 明
1	Config	SqlMapConfiguration	对 SQL Map XML 配置文件的根节点进行信息转化
2	GlobalProps	Properties	从本地属性文件中获取信息，用于保存全局性的属性
3	TxProps	Properties	对 SQL Map XML 配置文件的 transactionManager 节点进行信息转化和事务的属性转化
4	DsProps	Properties	对 SQL Map XML 配置文件的 datasource 节点进行属性转化
5	CacheProps	Properties	对 SQL Map XML 映射文件的 cacheModel 节点进行属性转化
6	useStatementNamespaces	boolean	
7	SqlIncludes	HashMap	
8	ParamConfig	ParameterMapConfig	对 SQL Map XML 映射文件的 parameterMap 节点进行转化对象转化
9	ResultConfig	ResultMapConfig	对 SQL Map XML 映射文件的 resultMap 节点进行转化对象转化
10	CacheConfig	CacheModelConfig	对 SQL Map XML 映射文件的 cacheModel 节点进行转化对象转化
11	Namespace	String	
12	DataSource	DataSource	暂时存放解析生成的数据库 DataSource

XmlParserState 的中介者表现如图 6-7 所示。其中 XmlParserState 是中介者，SqlMapConfigParser 是配置文件的主解析器，SqlMapParser 是映射文件的主解析器，SqlStatementParser 是映射文件中 Statement 类型节点（包括 select、insert、update、delete 和 procedure 等节点）的解析器。SqlMapConfiguration 是配置的一级门面。

ParameterMapConfig、ResultMapConfig、CacheModelConfig 和 MappedStatementConfig 是二级门面。

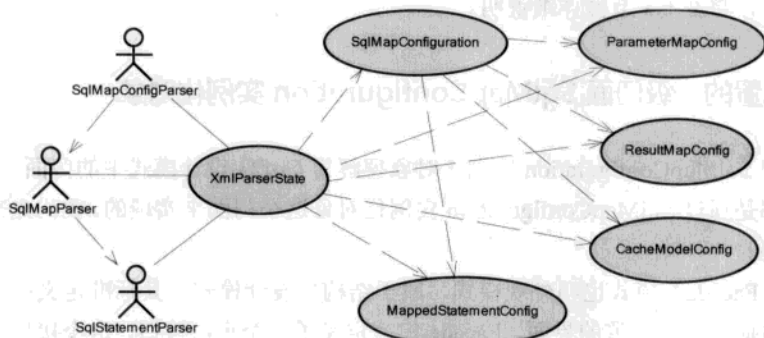


图 6-7 XmlParserState 在 SQLMap 配置文件解析的中介者角色图

所有配置文件的读取信息，都是通过 XmlParserState 实例化对象转化到业务系统中的。其中 SqlMapConfiguration 实例化对象主要是处理关于系统中全局性配置信息，最后转移到两个最重要的实例化对象——SqlMapExecutorDelegate 的实例化对象和 SqlMapClientImpl 实例化对象。

为什么要有中介者呢？主要是各个节点之间本身还有关系，这样需要一个中间者来把这种关系沟通起来，达到通用解析的目的。例如，对于 transactionManager 节点和 dataSource 节点的解析，按照策略设计模式，这两种节点的解析方式是不同的。但是，transactionManager 节点是 dataSource 节点的父节点，当解析 dataSource 节点把 dataSource 节点生成的 dataSource 对象缓存到 XmlParserState 对象中时，解析 transactionManager 节点就会把暂存在 XmlParserState 对象由 transactionManager 节点生成 dataSource 对象赋值到 transactionManager 节点生成的 transactionManager 对象，这样也体现了一个包含关系。

XmlParserState 中介者的实现过程如下：

① SqlMapConfigParser 实例化对象对 SqlMap 配置文件进行解析，然后通过中介者 XmlParserState 实例化对象转移数据给第一级配置门户 SqlMapConfiguration 实例化对象。由 SqlMapConfiguration 实例化对象最后传递到两个最重要的实例化对象——SqlMapExecutorDelegate 的实例化对象和 SqlMapClientImpl 实例化对象。

② 当 SqlMap 配置文件的节点是 properties 时，SqlMapConfigParser 利用 XmlParserState 对象作为中介者把 properties 节点信息转移给 SqlMapConfiguration 对象。

③ 当 SqlMap 配置文件的节点是 settings、typeAlias、typeHandler、transactionManager 和 resultObjectFactory 时，SqlMapConfigParser 利用 XmlParserState 对象作为中介者把上述节点信息转移给 SqlMapConfiguration 对象，同时还有一些继续转移，最后数据的终点站是 SqlMapExecutorDelegate 对象。

④ 当 SqlMapConfigParse 解析到 SqlMap 节点的时候，就开始对 SqlMap 映射文件进

行处理,这时候就创建出 `SqlMapParser` 实例化对象,由 `SqlMapParser` 实例化对象对 `SqlMap` 映射文件进行解析,但是其中的参数传递还是 `XmlParserState` 对象。关于对 `SqlMap` 映射文件的解析,将在 6.3 节做详细说明。

6.2.6 配置的一级门面 `SqlMapConfiguration` 实例化对象

可以把 `SqlMapConfiguration` 实例化对象理解为 Gof23 设计模式中的门面模式。即各种配置信息都是通过 `SqlMapConfiguration` 实例化对象这个门面来实现的。我们先来看看门面模式的定义。

门面 (Facade) 模式也叫外观模式,属于结构型设计模式。其标准定义:为子系统的一组接口提供一个一致的界面,Facade 模式定义了一个高层接口,这个接口使得这一子系统更加容易使用。实际上,外部与一个子系统的通信必须通过一个统一的门面对象进行。于是,门面模式提供了一个高层次的接口,使得子系统更易于使用。每一个子系统只有一个门面类,而且此门面类只有一个实例。也就是说,它是一个单例模式,但整个系统可以有多个门面类。

Facade 结构如图 6-8 所示。其角色包括门面 (Facade) 角色和子系统 (Subsystem) 角色,说明如下。

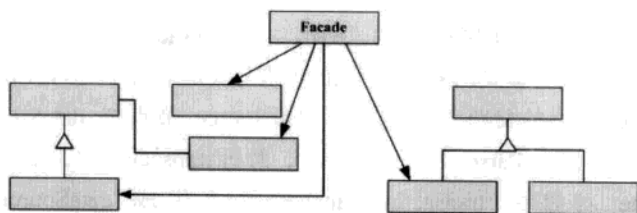


图 6-8 门面模式结构

① 门面 (Facade) 角色: 客户端可以调用这个方法。此角色知晓相关的 (一个或者多个) 子系统的功能和责任。在正常情况下,本角色会将所有从客户端发来的请求委派到相应的子系统去。

② 子系统 (Subsystem) 角色: 可以同时有一个或者多个子系统。每一个子系统都不是一个单独的类,而是一个类的集合。每一个子系统都可以被客户端直接调用,或者被门面角色调用。子系统并不知道门面的存在,对于子系统而言,门面仅仅是另外一个客户端而已。

我们来看看门面 `SqlMapConfiguration` 的属性和构造方法吧。

```
public class SqlMapConfiguration {
    private static final Probe PROBE = ProbeFactory.getProbe();
    private ErrorContext errorContext;
    private SqlMapExecutorDelegate delegate;
    private TypeHandlerFactory typeHandlerFactory;
```

```

private SqlMapClientImpl client;
private Integer defaultStatementTimeout;

public SqlMapConfiguration() {
    errorContext = new ErrorContext();
    delegate = new SqlMapExecutorDelegate();
    typeHandlerFactory = delegate.getTypeHandlerFactory();
    client = new SqlMapClientImpl(delegate);
    registerDefaultTypeAliases();
}
//其他内容
.....

private void registerDefaultTypeAliases() {
    // TRANSACTION ALIASES
    typeHandlerFactory.putTypeAlias("JDBC", JdbcTransactionConfig.class.getName());
    typeHandlerFactory.putTypeAlias("JTA", JtaTransactionConfig.class.getName());
    typeHandlerFactory.putTypeAlias("EXTERNAL", ExternalTransactionConfig.class.
getName());

    // DATA SOURCE ALIASES
    typeHandlerFactory.putTypeAlias("SIMPLE", SimpleDataSourceFactory.class.getName());
    typeHandlerFactory.putTypeAlias("DBCP", DbcpDataSourceFactory.class.getName());
    typeHandlerFactory.putTypeAlias("JNDI", JndiDataSourceFactory.class.getName());

    // CACHE ALIASES
    typeHandlerFactory.putTypeAlias("FIFO", FifoCacheController.class.getName());
    typeHandlerFactory.putTypeAlias("LRU", LruCacheController.class.getName());
    typeHandlerFactory.putTypeAlias("MEMORY", MemoryCacheController.class.getName());
    // use a string for OSCache to avoid unnecessary loading of properties upon init
    typeHandlerFactory.putTypeAlias("OSCACHE",
"com.ibatis.sqlmap.engine.cache.oscache.OSCacheController");

    // TYPE ALIASES
    typeHandlerFactory.putTypeAlias("dom", DomTypeMarker.class.getName());
    typeHandlerFactory.putTypeAlias("domCollection", DomCollectionTypeMarker.class.
getName());
    typeHandlerFactory.putTypeAlias("xml", XmlTypeMarker.class.getName());
    typeHandlerFactory.putTypeAlias("xmlCollection", XmlCollectionTypeMarker.class.get
Name());
}
}

```

在构造方法中，创建了 `SqlMapExecutorDelegate` 的实例化对象 `delegate` 和 `SqlMapClientImpl` 实例化对象 `client`，同时也创建了 `TypeHandlerFactory` 实例化对象 `typeHandlerFactory`。并且调用了 `registerDefaultTypeAliases`，初始化了最基本的 `TypeAliases`，这些初始化的 `Aliases` 都是基本的别名处理。`SqlMapConfiguration` 类的属性如表 6-3 所示。

表 6-3 SqlMapConfiguration 类的属性列表和说明

序号	变量名称	类 别	说 明
1	PROBE	Probe	
2	errorContext	ErrorContext	用于查找错误信息
3	delegate	SqlMapExecutorDelegate	关联 SqlMapExecutorDelegate 对象。
4	typeHandlerFactory	TypeHandlerFactory	数据类型转化的工厂类
5	client	SqlMapClientImpl	
6	defaultStatementTimeout	Integer	

而我们在后面会说明，SqlMapExecutorDelegate 实例化对象是业务逻辑的核心，也是业务处理的门面。任何操作都是通过接受 SqlMapExecutorDelegate 实例化对象来进行管理并最后实现的。在 SqlMapConfiguration 实例化对象这个配置门面中创建 SqlMapExecutorDelegate 实例化对象业务门面，实际上就是把配置信息向业务引擎信息进行了转化。当外部在调用各种业务的时候，它所使用的基础信息就是这样传递过去的。

6.2.7 一级应用门面 SqlMapExecutorDelegate 实例化对象

如果说 SqlMapConfiguration 实例化对象是 iBATIS SQL Map 的一级配置门面，那么 SqlMapExecutorDelegate 就是 iBATIS SQL Map 的一级应用门面。配置门面把所有的配置信息都转移到应用门面上。这样，应用系统才能根据可定制的配置信息来调度相应的执行组件。

在这里简单地讲解一下 SqlMapExecutorDelegate 类的内容。SqlMapExecutorDelegate 的属性和初始化如下。

```
public class SqlMapExecutorDelegate {

    private static final Probe PROBE = ProbeFactory.getProbe();

    private boolean lazyLoadingEnabled;
    private boolean cacheModelsEnabled;
    private boolean enhancementEnabled;
    private boolean useColumnLabel = true;
    private boolean forceMultipleResultSetSupport;

    private TransactionManager txManager;

    private HashMap mappedStatements;
    private HashMap cacheModels;
    private HashMap resultMaps;
    private HashMap parameterMaps;

    protected SqlExecutor sqlExecutor;
    private TypeHandlerFactory typeHandlerFactory;
    private DataExchangeFactory dataExchangeFactory;
```

```

private ResultObjectFactory resultObjectFactory;
private boolean statementCacheEnabled;

/**
 * Default constructor
 */
public SqlMapExecutorDelegate() {
    mappedStatements = new HashMap();
    cacheModels = new HashMap();
    resultMaps = new HashMap();
    parameterMaps = new HashMap();

    sqlExecutor = new SqlExecutor();
    typeHandlerFactory = new TypeHandlerFactory();
    dataExchangeFactory = new DataExchangeFactory(typeHandlerFactory);
} //其他内容
.....
}

```

SqlMap 配置 Settings 节点属性和 SqlMapExecutorDelegate 对象属性映射关系如表 6-4 所示。

表 6-4 SqlMap 配置 Settings 节点和 SqlMapExecutorDelegate 对象属性映射关系

序号	SqlMap 配置节点下属性节点	SqlMapExecutorDelegate 对象属性	类 型
1	lazyLoadingEnabled="true"	lazyLoadingEnabled	boolean
2	CacheModelsEnabled="true"	CacheModelsEnabled	boolean
3	enhancementEnabled="true"	enhancementEnabled	boolean
4	useColumnLabel	useColumnLabel	boolean
5	forceMultipleResultSetSupport	forceMultipleResultSetSupport	boolean
6	statementCachingEnabled	statementCacheEnabled	boolean

在这里进行了一些转化。这些变量全都是全局性或者是重量级变量，关于这些变量的介绍，可以参考 SqlMapExecutorDelegate 对象属性。

6.2.8 SQL Map 配置文件中节点解析的通用处理

在每个 SQL Map 配置文件都有这么一个方法，即在每个节点处理中都有如下的预处理语句。

```
Properties attributes = NodeletUtils.parseAttributes(node, state.getGlobalProps());
```

其功能主要是针对要读取配置的属性文件，把属性文件中的内容作为全局变量来处理。预处理实际调用的代码如下。

```

public static Properties parseAttributes(Node n, Properties variables) {
    Properties attributes = new Properties();

```

```

NamedNodeMap attributeNodes = n.getAttributes();
for (int i = 0; i < attributeNodes.getLength(); i++) {
    Node attribute = attributeNodes.item(i);
    String value = parsePropertyTokens(attribute.getNodeValue(), variables);
    attributes.put(attribute.getNodeName(), value);
}
return attributes;
}

public static String parsePropertyTokens(String string, Properties variables) {
    final String OPEN = "${";
    final String CLOSE = "}";

    String newString = string;
    if (newString != null && variables != null) {
        int start = newString.indexOf(OPEN);
        int end = newString.indexOf(CLOSE);

        while (start > -1 && end > start) {
            String prepend = newString.substring(0, start);
            String append = newString.substring(end + CLOSE.length());
            String propName = newString.substring(start + OPEN.length(), end);
            String propValue = variables.getProperty(propName);
            if (propValue == null) {
                newString = prepend + propName + append;
            } else {
                newString = prepend + propValue + append;
            }
            start = newString.indexOf(OPEN);
            end = newString.indexOf(CLOSE);
        }
    }
    return newString;
}

```

这个预处理主要是实现在各个属性内容中是否有“\${}”，如果有这样的内容，就用全局 Properties 中的内容进行替换，比如在 XML 配置文件中如下配置信息。

```
<property name="JDBC.Driver" value="${driver}"/>
```

预处理会用 properties 节点中所获得的全局 propertie 来替换里面的内容，如果全局 propertie 文件有这么一个设置，

```
driver=org.hsqldb.jdbcDriver
```

那么最后转化的信息为：

```
<property name="JDBC.Driver" value="org.hsqldb.jdbcDriver"/>
```

6.2.9 数据库事务节点的解析和转化

数据库和事务节点主要是 transactionManager 节点和 datasource 节点的解析。transactionManager 节点为 SQL Map 配置事务管理服务, 属性 type 指定所使用的事务管理器类型。而 transactionManager 节点是基于数据库的 datasource 的事务管理器。

datasource 节点主要是数据库的 datasource。

1. datasource 节点的解析

先看看 datasource 节点实现的代码, 首先是解析 datasource 节点下的属性节点。SqlMapConfig.xml 文件对 datasource 节点的处理, 首先调用 SqlMapConfigParser 类的 addTransactionManagerNodelets 方法, 在 addTransactionManagerNodelets 方法中对 datasource 节点的处理分为两个部分, 第一部分是对 datasource 节点下 property 的处理, 通过调用 parser.addNodelet("/sqlMapConfig/transactionManager/dataSource/property", new Nodelet()) 来实现。第二部分是对 datasource 节点自身属性的处理, 通过调用 parser.addNodelet("/sqlMapConfig/transactionManager/dataSource/end()", new Nodelet()) 来实现。

其次我们来进行对 datasource 节点下 property 的处理, 通过调用 parser.addNodelet("/sqlMapConfig/transactionManager/dataSource/end()", new Nodelet()) 中形成的 Nodelet 实例化对象的 process 方法, 其实现代码如下:

```
public void process(Node node) throws Exception {
    Properties attributes = NodeletUtils.parseAttributes(node, state.getGlobal
    Props());
    String name = attributes.getProperty("name");
    String value = NodeletUtils.parsePropertyTokens(attributes.getProperty
    ("value"), state.getGlobalProps());
    state.getDsProps().setProperty(name, value);
}
```

当 datasource 节点下的属性节点解析完成后, 把 datasource 节点下的属性节点信息存放到 XmlParserState 实例化对象的 Properties 类型的 dsProps 变量中。接着解析 datasource 节点信息, 调用 parser.addNodelet("/sqlMapConfig/transactionManager/dataSource/end()", new Nodelet()) 中 Nodelet 实例化对象的 process 方法代码如下。

```
public void process(Node node) throws Exception {
    //获得 XmlParserState 实例化对象中 Properties 类型的 dsProps 变量
    Properties attributes = NodeletUtils.parseAttributes(node, state.getGlobal
    Props());
    String type = attributes.getProperty("type");
    Properties props = state.getDsProps();
    type = state.getConfig().getHandlerFactory().resolveAlias(type);
```



```
try {
    state.getConfig().getErrorContext().setMoreInfo("Check the data source
type or class.");
    //基于类型创建 DataSourceFactory 实例化对象
    DataSourceFactory dsFactory = (DataSourceFactory) Resources.instantiate
(type);
    state.getConfig().getErrorContext().setMoreInfo("Check the data source
properties or configuration.");
    // DataSourceFactory 实例化对象进行初始化
    dsFactory.initialize(props);
    //把 DataSource 实例化对象赋值给 XmlParserState 实例化对象
    state.setDataSource(dsFactory.getDataSource());
    state.getConfig().getErrorContext().setMoreInfo(null);
} catch (Exception e) {
    if (e instanceof SqlMapException) {
        throw (SqlMapException) e;
    } else {
        throw new SqlMapException("..." + e, e);
    }
}
```

根据 datasource 节点创建 DataSourceFactory 实例化对象，再获得 XmlParserState 实例化对象中 Properties 类型的 dsProps 变量，然后基于这个信息初始化 DataSourceFactory 实例化对象，关于初始化内容，可参见第 7 章中的内容。接着把 DataSourceFactory 实例化对象中的 DataSource 对象存放到 XmlParserState 实例化对象中 DataSource 类型的 dataSource 变量中。DataSource 类型参数和对应类的关系如表 6-5 所示。

表 6-5 DataSource 类型参数和对应类的关系

序 号	类型参数	实例化对象	说 明
1	SIMPLE	com.ibatis.sqlmap.engine.datasource.SimpleDataSourceFactory	iBATIS 自己实现的一个 DataSource
2	DBCP	com.ibatis.sqlmap.engine.datasource.DbcpDataSourceFactory	采用 org.apache.commons.dbcp 的 DataSource
3	JNDI	com.ibatis.sqlmap.engine.datasource.JndiDataSourceFactory	采用 JNDI 的 DataSource

关于具体的 DataSource 的实现，将在后面的章节中做详细描述。

2. transactionManager 节点解析

SqlMapConfig.xml 文件对 transactionManager 节点的处理，首先调用 SqlMapConfig Parser 类的 addTransactionManagerNodelets 方法，但是在具体解析时候与解析 Datasource 节点是一样的，都分为两个部分，第一部分是解析 transactionManager 节点下属的 property 节点信息，通过调用 parser.addNodelet("/sqlMapConfig/transactionManager/property",new Nodelet())方法来实现。第二部分才是解析 transactionManager 节点信息,通过调用 parser.addNodelet("/sqlMapConfig/transactionManager/end()",new Nodelet())来实现。

首先，我们来看解析 transactionManager 节点下属的 property 节点信息，通过调用

parser.addNodelet("/sqlMapConfig/transactionManager/end()",new Nodelet())中形成的 Nodelet 实例化对象的 process 方法来实现,代码如下。

```
public void process(Node node) throws Exception {
    Properties attributes = NodeletUtils.parseAttributes(node, state.getGlobal
Props());
    String name = attributes.getProperty("name");
    String value = NodeletUtils.parsePropertyTokens(attributes.getProperty
("value"), state.getGlobalProps());
    state.getTxProps().setProperty(name, value);
}
```

主要是获得节点上的信息,其中 transactionManager 节点的“type”支持三种类别,分别是 JDBC、JTA 和 EXTERNAL,以便对整个 transactionManager 节点的处理。接着解析 transactionManager 节点信息,调用 parser.addNodelet("/sqlMapConfig/transactionManager/end()",new Nodelet())中 Nodelet 实例化对象的 process 方法代码如下。

```
public void process(Node node) throws Exception {
    Properties attributes = NodeletUtils.parseAttributes(node, state.getGlobal
Props());
    String type = attributes.getProperty("type");
    boolean commitRequired = "true".equals(attributes.getProperty("commitRequired"));

    state.getConfig().getErrorContext().setActivity("configuring the transact
tion manager");
    type = state.getConfig().getTypeHandlerFactory().resolveAlias(type);
    TransactionManager txManager;
    try {
        state.getConfig().getErrorContext().setMoreInfo("Check the transaction
manager type or class.");
        TransactionConfig config = (TransactionConfig) Resources.instantiate(type);
        config.setDataSource(state.getDataSource());
        state.getConfig().getErrorContext().setMoreInfo("Check the transactio
nmanager properties or configuration.");
        config.setProperties(state.getTxProps());
        config.setForceCommit(commitRequired);
        config.setDataSource(state.getDataSource());
        state.getConfig().getErrorContext().setMoreInfo(null);
        txManager = new TransactionManager(config);
    } catch (Exception e) {
        if (e instanceof SqlMapException) {
            throw (SqlMapException) e;
        } else {
            throw new SqlMapException(".... " + e, e);
        }
    }
    state.getConfig().setTransactionManager(txManager);
}
```

以上代码实现的功能是根据配置信息首先实例化一个 TransactionConfig 对象。这个实例化对象是按照下面的配置信息来实例化的。事务类型参数与实现类之间的对应关系如表 6-6 所示。

表 6-6 事务类型参数与实现类之间的对应关系

序号	类型参数	实例化对象	说 明
1	JDBC	com.ibatis.sqlmap.engine.transaction.jdbc.JdbcTransactionConfig	针对 JdbcTransaction 的配置转化类
2	JTA	com.ibatis.sqlmap.engine.transaction.jta.JtaTransactionConfig	针对 JtaTransaction 的配置转化类
3	EXTERNAL	com.ibatis.sqlmap.engine.transaction.external.ExternalTransactionConfig	针对 ExternalTransaction 的配置转化类

然后把上面生成的 DataSource 赋值给这个 TransactionConfig 对象，接着用这个 TransactionConfig 对象外参数实例化一个 TransactionManager 实例化对象。最后把这个 TransactionManager 实例化对象保存为 SqlMapExecutorDelegate 实例化对象的 txManager 属性。transactionManager 节点的信息转移如图 6-9 所示。

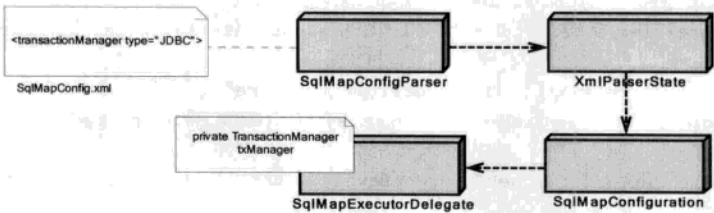


图 6-9 SqlMapConfig.xml 的 transactionManager 节点的最后结果

6.2.10 配置文件其他节点的解析和转化

本节说明各个节点的信息最终都转化到了哪里。这些节点都是在 SqlMapConfigParser 类的各个解析方法中处理的。现在分别介绍 SqlMapConfig.xml 文件中每个节点的解析和转化过程，包括解析的最后终点。

1. properties 节点的处理

SqlMapConfig.xml 文件对 properties 节点的处理，首先调用 SqlMapConfigParser 类的 addGlobalPropNodelets 方法，在 addGlobalPropNodelets 方法中调用 Nodelet 实例化对象的 process 方法，其实现的代码如下：

```
public void process(Node node) throws Exception {
    Properties attributes = NodeletUtils.parseAttributes(node, state.getGlobal
Props());
    String resource = attributes.getProperty("resource");
    String url = attributes.getProperty("url");
    state.setGlobalProperties(resource, url);
}
```

获得配置文件中 properties 的 properties 文件目录信息。然后调用 XmlParserState 实例化对象的 setGlobalProperties 方法。

```
public void setGlobalProperties(String resource, String url) {
    config.getErrorContext().setActivity("loading global properties");
    try {
        Properties props;
        if (resource != null) {
            config.getErrorContext().setResource(resource);
            props = Resources.getResourceAsProperties(resource);
        } else if (url != null) {
            config.getErrorContext().setResource(url);
            props = Resources.getUrlAsProperties(url);
        } else {
            throw new RuntimeException("The " + "properties" + " element requires either
a resource or a url attribute.");
        }

        // Merge properties with those passed in programmatically
        if (props != null) {
            props.putAll(globalProps);
            globalProps = props;
        }
    } catch (Exception e) {
        throw new RuntimeException("Error loading properties. Cause: " + e, e);
    }
}
```

该方法通过配置文件目录信息，然后去打开这个属性文件，获得全局的配置信息，并把这些信息都保存在 XmlParserState 实例化对象中 Properties 类型的私有变量 globalProps 中，为将来进行各个节点预处理提供基础数据。转化过程如图 6-10 所示。

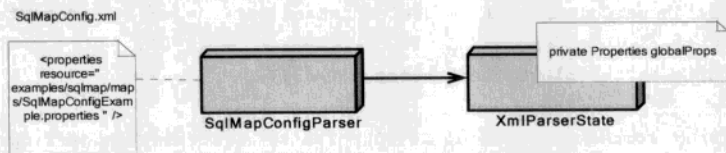


图 6-10 SqlMapConfig.xml 的 properties 节点的最后结果

2. settings 节点的处理

SqlMapConfig.xml 文件对 settings 节点的处理，首先调用 SqlMapConfigParser 类的 addSettingsNodelets 方法，在 addSettingsNodelets 方法中调用 Nodelet 实例化对象的 process 方法，其实现的代码如下：

```
public void process(Node node) throws Exception {
    Properties attributes = NodeletUtils.parseAttributes(node, state.getGlobal
```

```
Props());  
    SqlMapConfiguration config = state.getConfig();  
  
    String classInfoCacheEnabledAttr = attributes.getProperty("classInfoCache  
Enabled");  
    boolean classInfoCacheEnabled =  
        (classInfoCacheEnabledAttr == null || "true".equals(classInfoCacheEnabledAttr));  
    config.setClassInfoCacheEnabled(classInfoCacheEnabled);  
  
    String lazyLoadingEnabledAttr = attributes.getProperty("lazyLoadingEnabled");  
    boolean lazyLoadingEnabled = (lazyLoadingEnabledAttr == null || "true".  
equals(lazyLoadingEnabledAttr));  
    config.setLazyLoadingEnabled(lazyLoadingEnabled);  
  
    String statementCachingEnabledAttr = attributes.getProperty("statementCa  
chingEnabled");  
    boolean statementCachingEnabled =  
        (statementCachingEnabledAttr == null || "true".equals(statementCachingEnabledAttr));  
    config.setStatementCachingEnabled(statementCachingEnabled);  
  
    String cacheModelsEnabledAttr = attributes.getProperty("cacheModelsEnabled");  
    boolean cacheModelsEnabled = (cacheModelsEnabledAttr == null || "true".  
equals(cacheModelsEnabledAttr));  
    config.setCacheModelsEnabled(cacheModelsEnabled);  
  
    String enhancementEnabledAttr = attributes.getProperty("enhancementEnabled");  
    boolean enhancementEnabled = (enhancementEnabledAttr == null || "true".  
equals(enhancementEnabledAttr));  
    config.setEnhancementEnabled(enhancementEnabled);  
  
    String useColumnLabelAttr = attributes.getProperty("useColumnLabel");  
    boolean useColumnLabel = (useColumnLabelAttr == null || "true".equals(use  
ColumnLabelAttr));  
    config.setUseColumnLabel(useColumnLabel);  
  
    String forceMultipleResultSetSupportAttr = attributes.getProperty("force  
MultipleResultSetSupport");  
    boolean forceMultipleResultSetSupport = "true".equals(forceMultipleResult  
SetSupportAttr);  
    config.setForceMultipleResultSetSupport(forceMultipleResultSetSupport);  
  
    String defaultTimeoutAttr = attributes.getProperty("defaultStatementTimeout");  
    Integer defaultTimeout = defaultTimeoutAttr == null ? null : Integer.valueOf  
(defaultTimeoutAttr);  
    config.setDefaultStatementTimeout(defaultTimeout);  
  
    String useStatementNamespacesAttr = attributes.getProperty("useStatement
```

```
Namespaces");
    boolean useStatementNamespaces = "true".equals(useStatementNamespacesAttr);
    state.setUseStatementNamespaces(useStatementNamespaces);
}
```

配置文件中 `sqlMapConfig/setting` 节点的 `classInfoCacheEnabled` 属性，最后到达 `ClassInfo` 类的静态属性 `CacheEnabled`。这主要是用于是否把载入的 `JavaBean` 对象进行缓存处理。`SqlMapConfig.xml` 的转化过程如图 6-11 所示。

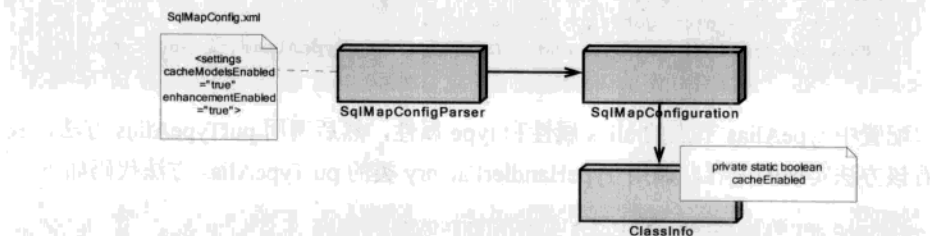


图 6-11 `SqlMapConfig.xml` 的 `classInfoCacheEnabled` 属性的最终结果

配置文件中 `sqlMapConfig/setting` 节点的 `lazyLoadingEnabled` 属性、`statementCachingEnabled` 属性、`cacheModelsEnabled` 属性、`enhancementEnabled` 属性、`useColumnLabel` 属性、`forceMultipleResultSetSupport` 属性都通过配置门面 `SqlMapConfiguration` 实例化对象转化成业务处理门面 `SqlMapExecutorDelegate` 实例化对象的各个私有属性，它们都是 `Boolean` 类型变量。在数据转化的最终结果如图 6-12 所示（可以参看表 6-2 的内容）。

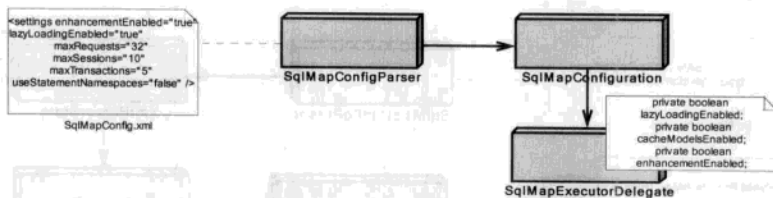


图 6-12 `SqlMapConfig.XML` 的 `setting` 节点其他属性的最终结果

配置文件中 `sqlMapConfig/setting` 节点的 `defaultStatementTimeout` 属性和 `useStatementNamespaces` 属性转化成 `XmlParserState` 实例化对象的私有属性，其属性的最终结果如图 6-13 所示。

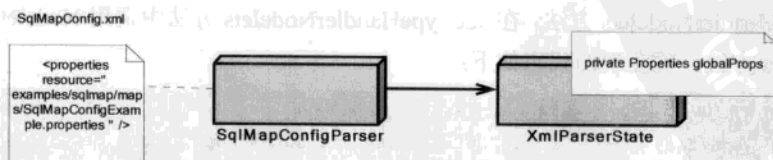


图 6-13 `setting` 节点的 `defaultStatementTimeout` 和 `useStatementNamespaces` 属性的最终结果

3. typeAlias 节点

SqlMapConfig.xml 文件对 typeAlias 节点的处理，首先调用 SqlMapConfigParser 类的 addTypeAliasNodelets 方法，在 addTypeAliasNodelets 方法中调用 Nodelet 实例化对象的 process 方法，其实现的代码如下：

```
public void process(Node node) throws Exception {
    Properties prop = NodeletUtils.parseAttributes(node, state.getGlobalProps());
    String alias = prop.getProperty("alias");
    String type = prop.getProperty("type");
    state.getConfig().getHandlerFactory().putTypeAlias(alias, type);
}
```

获取配置中 typeAlias 节点的 alias 属性和 type 属性，然后调用 putTypeAlias 方法，我们来看看该方法实现的内容，调用 TypeHandlerFactory 类的 putTypeAlias 方法代码如下。

```
public void putTypeAlias(String alias, String value) {
    String key = null;
    if(alias != null) key = alias.toLowerCase();
    if (typeAliases.containsKey(key) && !typeAliases.get(key).equals(value)) {
        throw new SqlMapException("...");
    }
    typeAliases.put(key, value);
}
```

最后转化到 TypeHandlerFactory 类的 HashMap 类型变量 typeAliases 的一个值，其转化如图 6-14 所示。

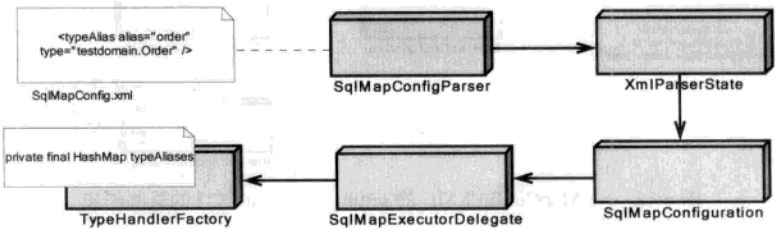


图 6-14 typeAlias 节点的最后结果

4. typeHandler 节点

SqlMapConfig.xml 文件对 typeHandler 节点的处理，首先调用 SqlMapConfigParser 类的 addTypeHandlerNodelets 方法，在 addTypeHandlerNodelets 方法中调用 Nodelet 实例化对象的 process 方法，其实现的代码如下：

```
public void process(Node node) throws Exception {
    Properties prop = NodeletUtils.parseAttributes(node, state.getGlobalProps());
    String jdbcType = prop.getProperty("jdbcType");
    String javaType = prop.getProperty("javaType");
}
```



```

String callback = prop.getProperty("callback");

javaType = state.getConfig().getTypeHandlerFactory().resolveAlias(javaType);
callback = state.getConfig().getTypeHandlerFactory().resolveAlias(callback);

state.getConfig().newTypeHandler(Resources.className(javaType),
jdbcType, Resources.instantiate(callback));
}

```

其中, `state.getConfig().newTypeHandler` 的实现代码如下:

```

public void newTypeHandler(Class javaType, String jdbcType, Object callback) {
    try {
        ...
        TypeHandlerFactory typeHandlerFactory = client.getDelegate().getTypeHandler
Factory();
        TypeHandler typeHandler;
        if (callback instanceof TypeHandlerCallback) {
            typeHandler = new CustomTypeHandler((TypeHandlerCallback) callback);
        } else if (callback instanceof TypeHandler) {
            typeHandler = (TypeHandler) callback;
        } else {
            throw new RuntimeException("...");
        }
        ...
        if (jdbcType != null && jdbcType.length() > 0) {
            typeHandlerFactory.register(javaType, jdbcType, typeHandler);
        } else {
            typeHandlerFactory.register(javaType, typeHandler);
        }
    } catch (Exception e) {
        throw new SqlMapException("Error registering occurred. Cause: " + e, e);
    }
    errorContext.setMoreInfo(null);
    errorContext.setObjectId(null);
}

```

`typeAlias` 节点中配置信息最终转化的结果, 如图 6-15 所示。

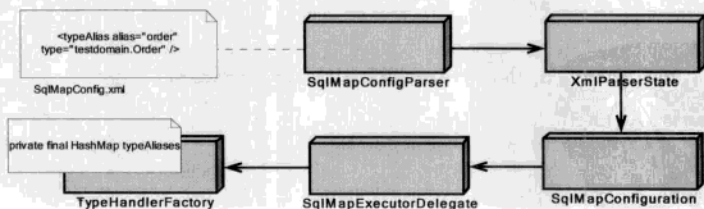


图 6-15 `typeAlias` 节点的最终结果

5. sqlmap 节点的处理

`SqlMapConfig.xml` 文件对 `sqlmap` 节点的处理, 首先调用 `SqlMapConfigParser` 类的

addSqlMapNodelets 方法，在 addSqlMapNodelets 方法中调用 Nodelet 实例化对象的 process 方法，其实现的代码如下。

```
protected void addSqlMapNodelets() {
    parser.addNodelet("/sqlMapConfig/sqlMap", new Nodelet() {
        public void process(Node node) throws Exception {
            state.getConfig().getErrorContext().setActivity("loading the SQL Map
resource");

            Properties attributes = NodeletUtils.parseAttributes(node, state.get
GlobalProps());
            String resource = attributes.getProperty("resource");
            String url = attributes.getProperty("url");

            if (usingStreams) {
                InputStream inputStream = null;
                if (resource != null) {
                    state.getConfig().getErrorContext().setResource(resource);
                    inputStream = Resources.getResourceAsStream(resource);
                } else if (url != null) {
                    state.getConfig().getErrorContext().setResource(url);
                    inputStream = Resources.getUrlAsStream(url);
                } else {
                    throw new SqlMapException(
                        "The <sqlMap> element requires either a resource or a url
attribute.");
                }
                new SqlMapParser(state).parse(inputStream);
            } else {
                Reader reader = null;
                if (resource != null) {
                    state.getConfig().getErrorContext().setResource(resource);
                    reader = Resources.getResourceAsReader(resource);
                } else if (url != null) {
                    state.getConfig().getErrorContext().setResource(url);
                    reader = Resources.getUrlAsReader(url);
                } else {
                    throw new SqlMapException(
                        "The <sqlMap> element requires either a resource or
a url attribute.");
                }
                new SqlMapParser(state).parse(reader);
            }
        }
    });
}
```

主要功能是从配置文件中得到各个映射对象名称和映射文件录路径信息，然后基于这些信息新建一个 SqlMapParser 实例化对象并调用该对象的 parse 方法。关于 SqlMapParser 对象如何解析 SQL Map 映射文件，我们在 6.3 节做详细的说明。

6. resultObjectFactory 节点的处理,

SqlMapConfig.xml 文件对 resultObjectFactory 节点的处理, 首先调用 SqlMapConfigParser 类的 addResultObjectFactoryNodelets 方法, 在 addResultObjectFactoryNodelets 方法中对于 resultObjectFactory 节点的处理包括两部分, 第一部分是解析 resultObjectFactory 节点自身的属性, 通过调用 parser.addNodelet("/sqlMapConfig/resultObjectFactory", new Nodelet()) 方法来实现; 第二部分是解析 resultObjectFactory 节点下 property 节点的属性, 通过调用 parser.addNodelet("/sqlMapConfig/resultObjectFactory/property", new Nodelet()) 方法来实现。

首先调用 parser.addNodelet("/sqlMapConfig/resultObjectFactory", new Nodelet()) 生成的 Nodelet 实例化对象的 process 方法, 其实现的代码如下。

```
public void process(Node node) throws Exception {
    Properties attributes = NodeletUtils.parseAttributes(node, state.
getGlobalProps());
    String type = attributes.getProperty("type");

    state.getConfig().getErrorContext().setActivity("configuring the
Result Object Factory");
    ResultObjectFactory rof;
    try {
        rof = (ResultObjectFactory) Resources.instantiate(type);
        state.getConfig().setResultObjectFactory(rof);
    } catch (Exception e) {
        throw new SqlMapException("Error instantiating resultObject
Factory: " + type, e);
    }
}
```

上述代码获得配置信息中 resultObjectFactory 节点属性 type 的字符信息, 然后用这个字符信息实例化成 ResultObjectFactory 对象, 并把这个对象放到 SqlMapExecutorDelegate 对象中。

接着解析 resultObjectFactory 节点下 property 节点的信息, 调用 parser.addNodelet("/sqlMapConfig/resultObjectFactory/property", new Nodelet()) 中 Nodelet 实例化对象的 process 方法代码如下。

```
public void process(Node node) throws Exception {
    Properties attributes = NodeletUtils.parseAttributes(node, state.getGlobalProps());
    String name = attributes.getProperty("name");
    String value = NodeletUtils.parsePropertyTokens(attributes.getProperty("value"),
state.getGlobalProps());
    state.getConfig().getDelegate().getResultObjectFactory().setProperty(name, value);
}
```

其实现内容是为 ResultObjectFactory 对象进行属性赋值, 其属性转化如图 6-16 所示。

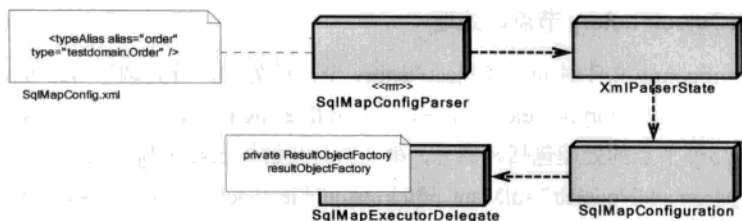


图 6-16 resultObjectFactory 节点的最后结果

6.3 解析 SQL Map 映射文件

在解析 SQL Map XML 映射文件时，首先阐述 SQL Map XML 映射文件的 XML 文件格式，包括里面节点的大致用途。其次讲解 SQL Map 是如何读取 XML 映射文件的，包括 cacheModel 节点、parameterMap 节点、resultMap 节点、statement 类型节点等。

6.3.1 SQL Map XML 映射文件格式

SqlMapMapping.xml 是 SQL Map 的映射文件，一般配置的样例如下。

```

<?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="com.ibatis.jpetstore.domain.Account" />

  <cacheModel id="AccountCache" type="LRU">
    <flushOnExecute statement="24" />
    <flushInterval hours="24" />
    <property name="size" value="1000" />
  </cacheModel>

  <parameterMap id="productParam" class="product">
    <parameter property="id" />
  </parameterMap>

  <resultMap id="productResult" class="product" extends="extends" xmlName="xmlName"
    groupBy="groupBy">
    <result property="id" nullValue="null" jdbcType="jdbcType"
      javaType="javaType" column="PRD_ID" columnIndex="int" select=""
      resultMap="name Of resultMap" typeHandler="" notNullColumn="" />
    <discriminator nullValue="" jdbcType="" javaType=""
      column="PRD_ID" columnIndex="" typeHandler="">
      <subMap value="" resultMap="" />
    </discriminator>
  </resultMap>

```

```

</discriminator>
</resultMap>

<statement id="updateProfile" cacheModel="name Of Cache"
    parameterMap="name Of ParameterMap" parameterClass="some.class.Name"
    resultMap="name Of ResultMap" resultClass="some.class.Name">
    UPDATE PROFILE SET LANGPREF = #languagePreference#, FAVCATEGORY
    = #favouriteCategoryId#, MYLISTOPT = #listOption#, BANNEROPT =
    #bannerOption# WHERE USERID = #username#
</statement>

<select id="getAccountByUsername" resultClass="account"
    parameterClass="string" cacheModel="AccountCache">
    SELECT SIGNON.USERNAME, ACCOUNT.EMAIL, ACCOUNT.FIRSTNAME,
    ACCOUNT.LASTNAME, ACCOUNT.STATUS, ACCOUNT.ADDR1 AS address1,
    ACCOUNT.ADDR2 AS address2, ACCOUNT.CITY, ACCOUNT.STATE,
    ACCOUNT.ZIP, ACCOUNT.COUNTRY, ACCOUNT.PHONE, PROFILE.LANGPREF AS
    languagePreference, PROFILE.FAVCATEGORY AS favouriteCategoryId,
    PROFILE.MYLISTOPT AS listOption, PROFILE.BANNEROPT AS
    bannerOption, BANNERDATA.BANNERNAME FROM ACCOUNT, PROFILE,
    SIGNON, BANNERDATA WHERE ACCOUNT.USERID = #username# AND
    SIGNON.USERNAME = ACCOUNT.USERID AND PROFILE.USERID =
    ACCOUNT.USERID AND PROFILE.FAVCATEGORY = BANNERDATA.FAVCATEGORY
</select>

<update id="updateAccount" parameterClass="account">
    UPDATE ACCOUNT SET EMAIL = #email#, FIRSTNAME = #firstName#,
    LASTNAME = #lastName#, STATUS = #status#, ADDR1 = #address1#,
    ADDR2 = #address2:VARCHAR#, CITY = #city#, STATE = #state#, ZIP
    = #zip#, COUNTRY = #country#, PHONE = #phone# WHERE USERID =
    #username#
</update>

<insert id="insertAccount" parameterClass="account">
    INSERT INTO ACCOUNT (EMAIL, FIRSTNAME, LASTNAME, STATUS, ADDR1,
    ADDR2, CITY, STATE, ZIP, COUNTRY, PHONE, USERID) VALUES
    (#email#, #firstName#, #lastName#, #status#, #address1#,
    #address2:VARCHAR#, #city#, #state#, #zip#, #country#, #phone#,
    #username#)
</insert>

<delete id="deleteAccount">
    DELETE INTO ACCOUNT WHERE ACCOUNT.USERID=#uiId#
</delete>

<procedure id="updateAccountProcedure" parameterClass="account"
    resultClass="int">
    {call out_int_account(?)}
</procedure>
</sqlMap>

```

SQL MapMapping XML 采用 DTD 格式来进行验证，其 DTD 格式可参见附录四。本文还是采用 XSD 来描述其结构，SqlMapConfig.xml 的 XSD 组成如图 6-17 所示。

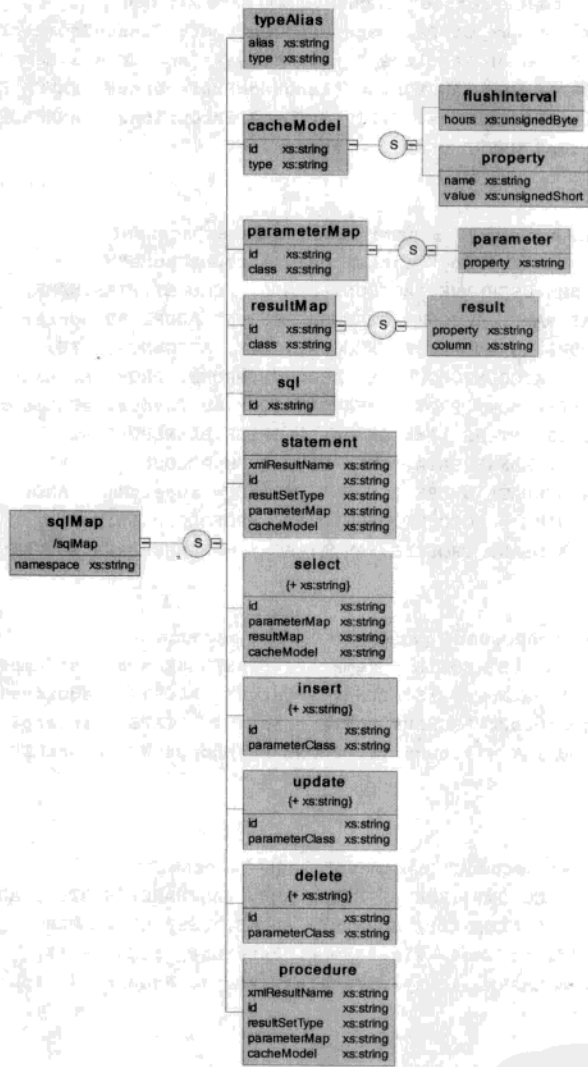


图 6-17 SqlMapMapping.xml 的 XSD 结构图

SQL MapMapping XML 的 XSD 的代码参见附录五。SqlMapMapping.xml 文件的根节点是 sqlMap 节点，sqlMap 根节点包括 11 个子节点，分别是 typeAlias、cacheModel、parameterMap、resultMap、sql、statement、select、insert、update、delete 和 procedure。在这 11 个子节点中，都是比较粗粒度地进行说明的，而每个节点里还有很多要涉及属性等的内容。这些节点的基本属性简单描述如下：

<typeAlias>元素是在 SQL Map XML 文件中只能定义一个，一般是当前 Mapping 映

射的 JavaBean。具有唯一定义。其功能是把类别名和类全名建立一种映射关系。与 SqlMapConfig.xml 文件 typeAlias 节点的作用是一样的。

<cacheModel>元素是在 SQL Map XML 文件中定义的可配置缓存模式。

< parameterMap >元素是在 SQL Map XML 文件中定义的参数变量，可以是 Map、JavaBean、简单数据变量。

< resultMap >元素是在 SQL Map XML 文件中定义的返回值变量，可以是 map、javabean、list、简单数据变量。

< sql >元素是在 SQL Map XML 文件中定义的通用 SQL 语句。

< statement >元素是 SQL Map XML 文件中的通用 SQL 映射声明，可以用于任何类型的 SQL 语句。

< select >元素是在 SQL Map XML 文件中基于查询的 SQL 映射声明。

< insert >元素是在 SQL Map XML 文件中基于插入的 SQL 映射声明。

< update >元素是在 SQL Map XML 文件中基于修改的 SQL 映射声明。

< delete >元素是 SQL Map XML 文件中基于删除的 SQL 映射声明。

< procedure >元素是 SQL Map XML 文件中基于存储过程的 SQL 映射声明。

对于 statement、select、insert、update、delete 和 procedure 来说，它们都是 sql 的处理，其中 statement 是通用的，而 select、insert、update、delete 和 procedure 是根据各个具体情况专用的 SQL 语句。其关系如表 6-7 所示。

表 6-7 各种 statement 类型及其属性和子元素

Statement 类型	属 性	子 元 素
<statement>	id parameterClass resultClass parameterMap resultMap CacheModel XMLResultName	所有的动态元素
<insert>	id parameterClass parameterMap	所有的动态元素 <selectKey>
<update>	id parameterClass parameterMap	所有的动态元素
<delete>	id parameterClass parameterMap	所有的动态元素
<select>	id parameterClass resultClass parameterMap resultMap CacheModel	所有的动态元素

续表

Statement 类型	属 性	子 元 素
<procedure>	id parameterClass resultClass parameterMap resultMap XMLResultName	所有的动态元素

在这里可以了解到，其实<insert>、<update>、<delete>三者是可以换用的。而 statement 可以替换所有的操作命名，在以后的代码分析中也能得出同样的结论。

6.3.2 SQL Map XML 映射文件读取总体说明

解析 SQL Map XML 映射文件的程序实现结构与解析 SQL Map XML 配置文件基本上是一致的，采用的技术手段和实现方式也类似，都是基于递归和路径来实现配置文件的全部遍历，都是采用基于匿名类的策略设计模式。不同在于 SQL Map XML 配置文件的解析器是 SqlMapConfigParser 实例化对象，而 SQL Map XML 映射文件的解析器是 SqlMapParser 实例化对象。

解析 SQL Map XML 映射文件要涉及 SqlMapConfigParser 类、SqlMapParser 类、NodeletParser 类、XmlParserState 类、Nodelet 接口、SqlMapConfiguration 类、CacheModelConfig 类、MappedStatementConfig 类、ParameterMapConfig 类、ResultMapConfig 类等，其类结构如图 6-18 所示。

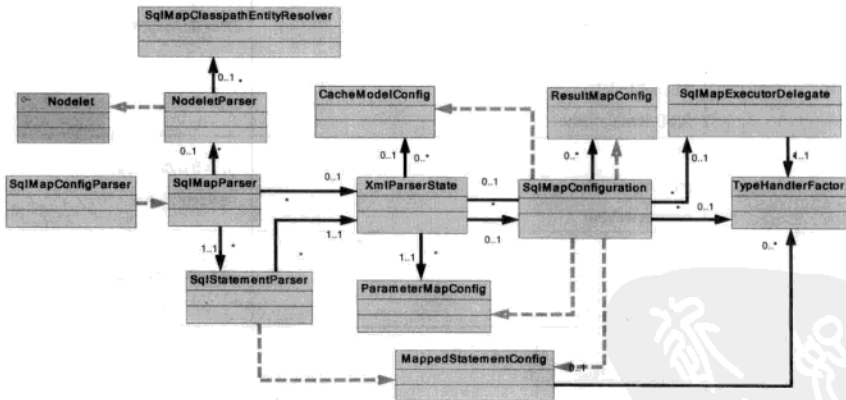


图 6-18 SQL Map XML 映射解析总体结构类图

SqlMapConfigParser Class 依赖 SqlMapParser 类，而由 SqlMapParser 类、Nodelet Parser 类、SqlMapClasspathEntityResolver 类、Nodelet 接口组合的结构与 SQL Map 中解析配置文件是相同的，处理也是一致的，但对于针对映射文件中具体节点的处理是不一样的。首先，

SqlMapParser 类要关联 SqlStatementParser 类, 该类主要是解析映射文件中的 SQL 声明节点, 如 Statement、Insert、delete、update 等。其次, SqlMapParser 类要关联 XmlParserState 类, 这种处理与 SQL Map 中解析配置文件是相同的。再次, XmlParserState 类是整个映射文件解析中的中转站, 它关联 SqlMapConfiguration (与 SQL Map 中解析配置文件是相同的), 同时还要关联 CacheModelConfig 类、ResultMapConfig 类、ParameterMapConfig 类等。最后, 作为配置的中心和枢纽的 SqlMapConfiguration 类, 它要依赖所有的配置类, 覆盖 CacheModelConfig 类、ResultMapConfig 类、ParameterMapConfig 类和 MappedStatementConfig 类。这就是 SQL Map XML 映射解析总体结构类的概括说明。

同时, 由于在解析过程中还有针对各个节点的对象实例化过程中, 这还要涉及 CacheModel 类、ParameterMap 类、ResultMap 类、MappedStatement 类、DeleteStatement 类、InsertStatement 类、ProcedureStatement 类、SelectStatement 类和 UpdateStatement 类。这些都只有到各个具体实现时再进行详细说明。

图 6-19 是 SQL Map XML 映射文件解析前半段过程的序列图。还是只列出主要的对象和主要的方法。其中*Config 包括 CacheModelConfig 类、MappedStatementConfig 类、ParameterMapConfig 类、ResultMapConfig 类等。

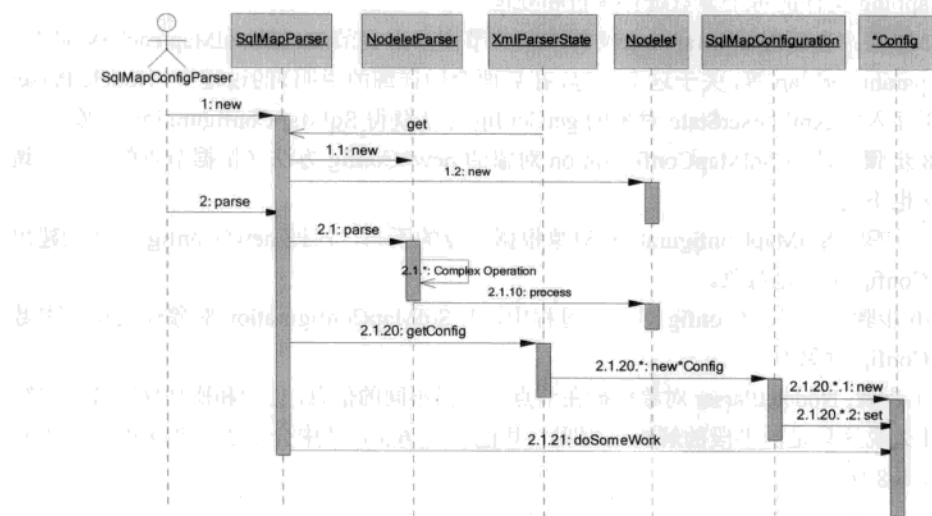


图 6-19 SQL Map XML 映射文件解析的序列图

SqlMapConfigParser 对象在解析 SqlMap 配置文件中 sqlmap 节点的时候创建 SqlMapParser 实例化对象, 并把本身的 XmlParserState 对象注入到 SqlMapParser 实例化对象中。SqlMapParser 实例化对象在实例化过程中, 新建 NodeletParser 对象, 并在构造函数中按照 SqlMap 映射文件的节点开始创建多个实现 Nodelet 接口的内部匿名对象。SqlMapParser 实例化对象在获得 SqlMapConfigParser 对象的 parse 消息后调用 NodeletParser 对象的 parse 方法。然后 NodeletParser 对象开始采用递归循环方式解析 SqlMap 映射文件的节点信息。在

解析的过程中,要通过 XmlParserState 对象来传递参数,并通过 SqlMapConfiguration 对象作为门面来创建各个节点的对应配置对象,如 cacheModel 节点对应 CacheModelConfig 类,parameterMap 节点对应 ParameterMapConfig 类。resultMap 节点对应 ResultMapConfig 类。statement 节点(包括 select、insert、update、delete 和 procedure 等节点)对应 SqlMapParser 类和 MappedStatementConfig 类。最后对各个 Config 对象进行赋值。

由于上述的过程比较抽象,下面对这个过程进行分步骤的解释。

第 1 步骤: SqlMapConfigParser 对象在对 SqlMap 配置文件的 sqlmap 节点解析中创建 SqlMapParser 实例化对象,由 SqlMapParser 实例化对象负责对 sqlmap 节点所映射的 SqlMapping 文件进行解析。SqlMapConfigParser 对象在创建 SqlMapParser 实例化对象过程中,通过构造方法把 XmlParserState 对象注入到 SqlMapParser 实例化对象中。

第 2 步骤: SqlMapParser 实例化对象,创建一个新的 NodeletParser 对象。

第 3 步骤: SqlMapParser 对象在构造阶段,创建出多个实现 Nodelet 接口的内部匿名对象。

第 4 步骤: SqlMapConfigParser 对象调用 SqlMapParser 对象的 parse 方法。

第 5 步骤: SqlMapParser 对象调用 NodeletParser 对象的 parse 方法。

第 6 步骤:通过 NodeletParser 对象在解析 XML 文件的递归循环 process 方法,开始对 SqlMapping 文件的每个节点进行解析和处理。

第 7 步骤:当处理 SqlMapping 文件中某些节点(这些节点包括/sqlMap/cacheModel/、/sqlMap/parameterMap/等,关于这个内容,在后面会做详细的说明)的过程中,NodeletParser 对象调用注入的 XmlParserState 对象的 getConfig 方法获得 SqlMapConfiguration 对象。

第 8 步骤:调用 SqlMapConfiguration 对象的 new*Config 方法(根据节点的不同,调用的方法也不同)。

第 9 步骤:SqlMapConfiguration 对象根据节点的不同,通过 new*Config 方法创建出不同的*Config 实例化对象。

第 10 步骤:在创建*Config 对象的过程中,把 SqlMapConfiguration 对象作为构造参数注入到*Config 对象中。

第 11 步骤:NodeletParser 对象依据在节点上获得不同的信息,处理和操作*Config 对象。

为什么说这只是前半段的解析过程呢?其他已在 6.3.1 节中介绍了。相关的处理类和接口如表 6-8 所示。

表 6-8 解析 SqlMap 映射文件要涉及的接口和类

接口或类	功能描述
com.ibatis.sqlmap.engine.builder.xml.SqlMapParser	SQL Map 映射文件的主解析类
com.ibatis.common.xml.NodeletParser	iBATIS 的通用解析 XML 节点的节点解析类,所以属工具类范畴
com.ibatis.sqlmap.engine.builder.xml.XmlParserState	用于解析 XML 节点时的连接类,主要是进行节点内节点之间的信息关联,属于存储中间状态的数据结构类,类似于 javabean
com.ibatis.common.xml.Nodelet	一个 XML 节点的接口,

续表

接口或类	功能描述
com.ibatis.sqlmap.engine.config.SqlMapConfiguration	与表 6-1 中的 com.ibatis.sqlmap.engine.config.SqlMapConfiguration 作用相同
com.ibatis.sqlmap.engine.impl.SqlMapClientImpl	与表 6-1 中的 com.ibatis.sqlmap.engine.config.SqlMapConfiguration 作用相同
com.ibatis.sqlmap.engine.impl.SqlMapExecutorDelegate	与表 6-1 中的 com.ibatis.sqlmap.engine.config.SqlMapConfiguration 作用相同
com.ibatis.sqlmap.engine.config.CacheModelConfig	缓存的配置信息载体
com.ibatis.sqlmap.engine.Cache.CacheModel	缓存的实现类
com.ibatis.sqlmap.engine.config.ParameterMapConfig	参数 Map 的配置信息载体
com.ibatis.sqlmap.engine.mapping.parameter.ParameterMap	参数 Map 的实现类
com.ibatis.sqlmap.engine.config.ResultMapConfig	结果集 Map 的配置信息载体
com.ibatis.sqlmap.engine.mapping.result.ResultMap	结果集 Map 的实现类
com.ibatis.sqlmap.engine.config.MappedStatementConfig	SQLMap 中通用 SQL 声明配置类
com.ibatis.sqlmap.engine.mapping.statement.MappedStatement	SQLMap 中通用 SQL 声明实现类
com.ibatis.sqlmap.engine.mapping.statement.DeleteStatement	SQLMap 中 Delete 的实现类
com.ibatis.sqlmap.engine.mapping.statement.InsertStatement	SQLMap 中 Insert 的实现类
com.ibatis.sqlmap.engine.mapping.statement.ProcedureStatement	SQLMap 中 Procedure 的实现类
com.ibatis.sqlmap.engine.mapping.statement.SelectStatement	SQLMap 中 Select 的实现类
com.ibatis.sqlmap.engine.mapping.statement.UpdateStatement	SQLMap 中 Update 的实现类

6.3.3 XmlParserState 对象解析 SQL Map 映射文件的协调者角色

XmlParserState 对象在解析 SQL Map XML 映射文件时同样也是充当协调者角色。不过这次协调的主要是 SQL Map XML 映射文件中的节点，其中介者角色如图 6-20 所示。

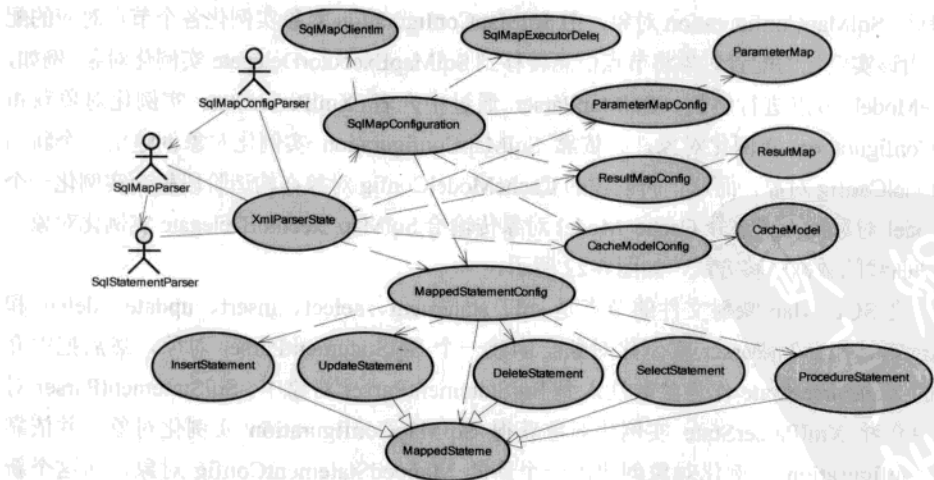


图 6-20 XmlParserState 在 SQLMap 映射文件解析的中介者角色图

在 6.2 节中 XmlParserState 充当了 SqlMap 配置文件的中介者角色。在对映射文件的解析过程中, XmlParserState 仍旧是中介者, SqlMapConfiguration 是配置的一级门面。ParameterMapConfig、ResultMapConfig、CacheModelConfig 和 MappedStatementConfig 是二级门面。

其实现步骤: 所有映射文件的读取信息, 都是通过 XmlParserState 实例化对象转化到业务系统中的。在 6.3.2 节已经说明了 XmlParserState 在解析配置文件的作用, 现在说明 XmlParserState 在解析映射文件中的作用。其实现要根据节点情况不同而采用的实现方式有所区别。

(1) 当 SQL Map 映射文件的节点是 typeAlias 时, SqlMapParser 实例化对象对 SqlMap 映射文件进行解析与进行 SqlMap 映射文件解析是一样的, 通过中介者 XmlParserState 实例化对象和 SqlMapConfiguration 实例化对象转移数据到 TypeHandlerFactory 对象。由 SqlMapConfiguration 实例化对象最后传递给 SqlMapExecutorDelegate 实例化对象。其实现过程如图 6-21 所示。

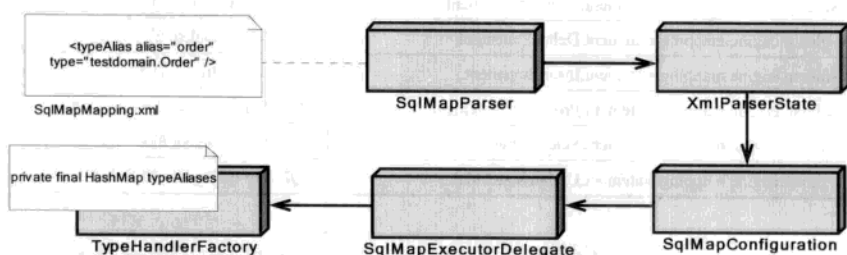


图 6-21 XmlParserState 在 typeAlias 节点解析的中介者角色图

(2) 当 SQL Map 映射文件的节点是 CacheModel、parameterMap、resultMap 时, SqlMapParser 实例化对象对 SqlMap 映射文件进行解析要经过几个环节, 首先是通过 XmlParserState 中介者角色转移到 SqlMapConfiguration 对象, 由 SqlMapConfiguration 对象实例化各个节点对应的配置对象。由该实例化的配置对象将节点信息转移到 SqlMapExecutorDelegate 实例化对象。例如, 对 cacheModel 节点进行解析, SqlMapParser 通过中介者 XmlParserState 实例化对象获得 SqlMapConfiguration 实例化对象, 并依靠 SqlMapConfiguration 实例化对象创建出一个新的 CacheModelConfig 对象, 而这个新创建的 CacheModelConfig 对象在构造阶段就会实例化一个 CacheModel 对象, 再把这个 Cache Model 对象传输给 SqlMapExecutorDelegate 实例化对象。这是一种曲线的数据转移方式, 如图 6-22 所示。

(3) 当 SQL Map 映射文件的节点是 sql、statement、select、insert、update、delete 和 procedure 时, SqlMapParser 实例化对象会创建一个 SqlStatementParser 对象, 然后把中介者角色的 XmlParserState 作为参数注入到 SqlStatementParser 对象中。SqlStatementParser 对象通过中介者 XmlParserState 实例化对象获得 SqlMapConfiguration 实例化对象, 并依靠 SqlMapConfiguration 实例化对象创建出一个新的 MappedStatementConfig 对象, 而这个新

创建的 `MappedStatementConfig` 对象内部有一个已经实例化的 `MappedStatement` 对象, 再把这个 `MappedStatement` 对象传输给 `SqlMapExecutorDelegate` 实例化对象。这也是一种曲线的数据转移方式, 如图 6-23 所示。

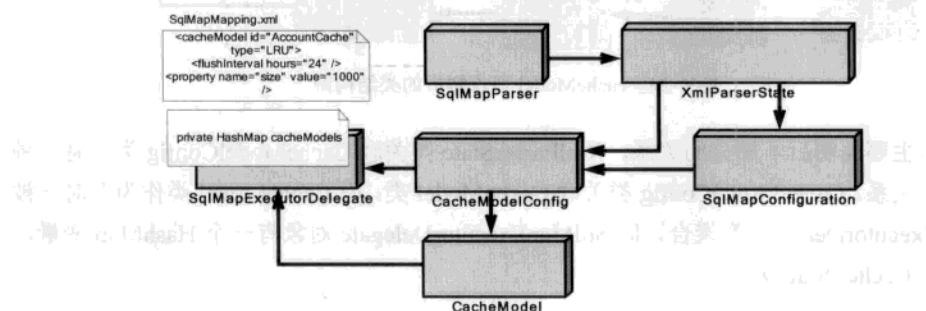


图 6-22 XmlParserState 在 cacheModel 节点解析的中介者角色图

在后面会对 `cacheModel` 节点、`parameterMap` 节点、`resultMap` 节点、`statement` 类型节点做详细的说明。

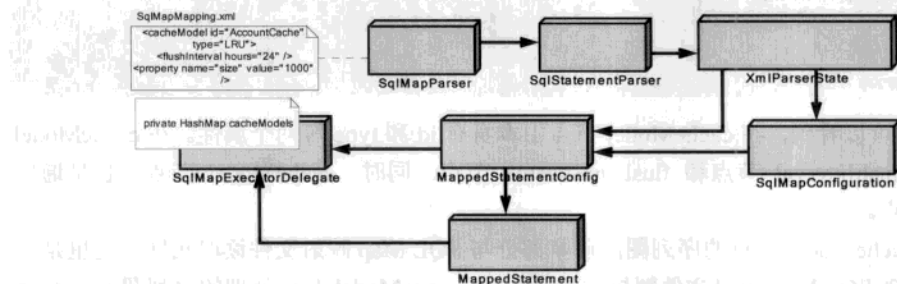


图 6-23 XmlParserState 在 MappedStatement 节点解析的中介者角色图

6.3.4 cacheModel 节点的解析和转化

解析 `cacheModel` 节点的前半部分内容与 SQL Map XML 映射文件读取总体说明是相同的, 也就是说, 本节的内容主要还是衔接上面的总体部分, 更加细化后面的处理部分。对 `cacheModel` 节点的解析和转化说明如下。

1. 解析 cacheModel 节点的类结构图

对 `cacheModel` 节点的解析要涉及如下这些类: `SqlMapConfigParser` 类、`SqlMap Parser` 类、`NodeletParser` 类、`Nodelet` 接口、`CacheModelConfig` 类、`CacheModel` 类和 `SqlMapExecutorDelegate` 类, 其类结构如图 6-24 所示。

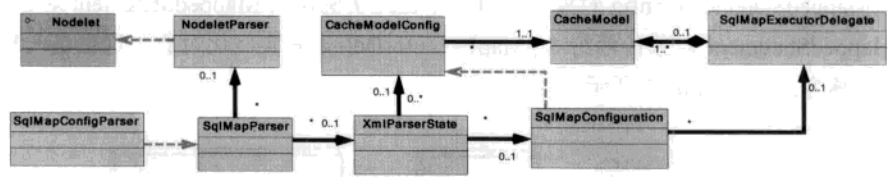


图 6-24 cacheModel 节点解析的类结构图

这里主要说明后半部分的关系。XmlParserState 类关联 CacheModelConfig 类，是一种一对多的关系。CacheModelConfig 类关联 CacheModel 类，而 CacheModel 类作为多对一被 SqlMapExecutorDelegate 类聚合，即 SqlMapExecutorDelegate 对象有一个 HashMap 变量，包含多个 CacheModel 对象。

2. cacheModel 节点处理的总体序列图和代码实现图

解析 SQL Map 的 cacheModel 节点，其 cacheModel 节点的结果内容如下所示。

```
<?xml version="1.0" encoding="UTF-8" ?>
<sqlMap namespace="Account">
  <CacheModel id="AccountCache" type="LRU">
    <flushInterval hours="24" />
    <property name="size" value="1000" />
  </CacheModel>
</sqlMap>
```

由上面可以看到，在 cacheModel 节点中本身有 id 和 type 等两个属性。在 cacheModel 节点下有 flushInterval 节点和 flushOnExecute 节点。同时，cacheModel 节点还包括通用 property 节点。

解析 cacheModel 节点的序列图的前半部分与 SQL Map 映射文件读取相同，这里是后半部分，衔接 SQL Map 映射文件解析的前半部分。cacheModel 节点处理的序列图如图 6-25 所示，其实现步骤如下：

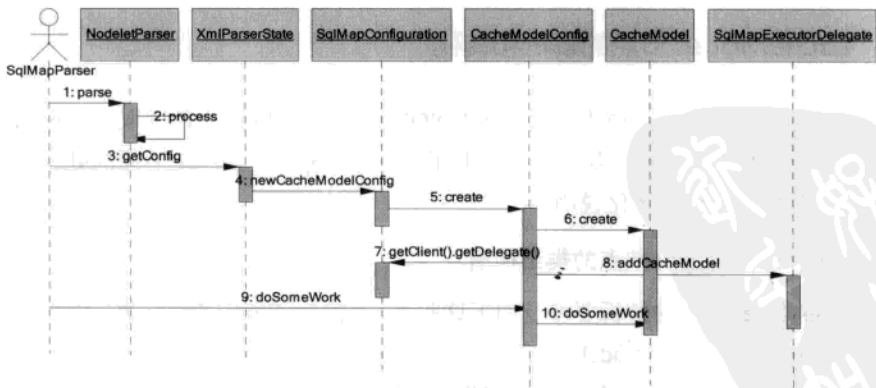


图 6-25 解析 cacheModel 节点的序列图

(1) SqlMapParser 对象调用 NodeletParser 对象, 对 SQL Map XML 映射文件中的 cacheModel 节点进行解析和处理;

(2) NodeletParser 对象调用 process 方法, 对 SQL Map XML 映射文件中的 cacheModel 节点本身、属性和子节点进行解析和处理;

(3) SqlMapParser 对象调用 XmlParserState 对象的 getConfig 方法, 获得 SqlMapConfiguration 对象;

(4) 调用 SqlMapConfiguration 对象的新 CacheModelConfig 方法;

(5) 再调用 SqlMapConfiguration 对象的新 CacheModelConfig 方法, 创建 CacheModelConfig 对象;

(6) CacheModelConfig 对象在实例化过程中, 创建 CacheModel 对象;

(7) 同时 CacheModelConfig 对象在实例化过程中, 通过 getClient().getDelegate()方法获得 SqlMapConfiguration 对象中的 SqlMapExecutorDelegate 对象;

(8) CacheModelConfig 对象调用 SqlMapExecutorDelegate 对象的 addCacheModel 方法, 把 CacheModel 实例化对象加入到 SqlMapExecutorDelegate 对象的 HashMap 中;

(9) SqlMapParser 对象对 CacheModelConfig 对象进行各种操作, 主要内容是把节点的信息转化为 CacheModelConfig 对象的信息或者对 CacheModelConfig 对象进行处理;

(10) CacheModelConfig 对象对 CacheModel 对象进行各种操作, 根据其数据来源与节点的配置信息来对 CacheModel 对象进行处理。

对于上述的第 9 个步骤和第 10 个步骤, 在这里再进行详细的解释和说明。

下面采用一个实现图 (如图 6-26 所示) 来说明这个代码的执行过程。

针对图 6-26 的代码实现过程, 再解析 cacheModel 节点, 首先从 SqlMapParser 类的 addCacheModelNodelets 方法开始, addCacheModelNodelets 方法实现节点解析的内容和说明如表 6-9 所示。

表 6-9 addCacheModelNodelets 方法实现节点解析内容和说明

序号	方法名称	实现功能
1	parser.addNodelet("/sqlMap/CacheModel", new Nodelet())	解析和处理 cacheModel 节点属性
2	parser.addNodelet("/sqlMap/CacheModel/end()", new Nodelet())	完成整个 cacheModel 节点的解析
3	parser.addNodelet("/sqlMap/CacheModel/property", new Nodelet())	解析和处理 cacheModel 节点下的属性节点
4	parser.addNodelet("/sqlMap/CacheModel/flushOnExecute", new Nodelet())	解析和处理 cacheModel 节点下的 flushOnExecute 节点
5	parser.addNodelet("/sqlMap/CacheModel/flushInterval", new Nodelet())	解析和处理 cacheModel 节点下的 flush Interval 节点

首先解析和处理解析 cacheModel 节点的属性, 根据获得的信息生成 CacheModelConfig 对象, CacheModelConfig 对象中创建 CacheModel 对象, 把所有生成的 CacheModel 对象添加到 SqlMapExecutorDelegate 对象中。接着是解析 cacheModel 节

点下的属性节点，根据获得的信息赋值到 `XmlParserState` 对象，作为一个中间变量。最后是解析 `cacheModel` 节点下的 `flushOnExecute` 节点和 `flushInterval` 节点，把获取的信息赋值到 `cachemodel` 对象中。

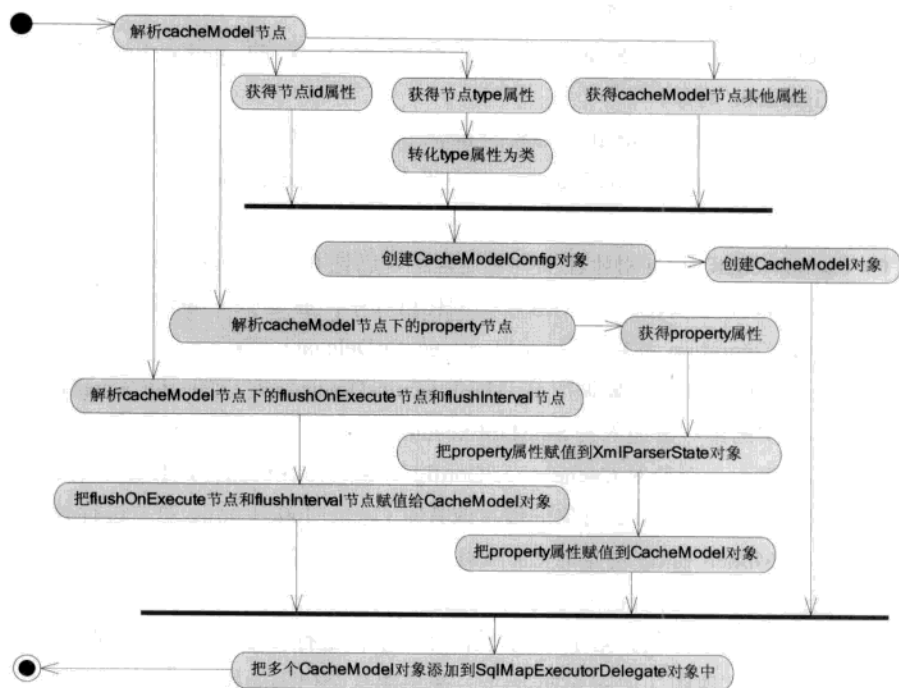


图 6-26 解析 cacheModel 节点的过程图

3. 解析 cacheModel 节点的实例代码说明和解释

下面进行代码说明。

(1) cacheModel 节点属性的解析和处理

调用 `addCacheModelNodelets` 方法的 `parser.addNodelet("/sqlMap/CacheModel", new Nodelet()`方法，实际上是最后调用 `Nodelet` 匿名类的 `process` 方法，代码如下。

```

public void process(Node node) throws Exception {
    Properties attributes = NodeletUtils.parseAttributes(node, state.getGlobal
Props());
    String id = state.applyNamespace(attributes.getProperty("id"));
    String type = attributes.getProperty("type");
    String readOnlyAttr = attributes.getProperty("readOnly");
    Boolean readOnly = readOnlyAttr ==
        null || readOnlyAttr.length() <= 0 ? null : new Boolean("true".equals
(readOnlyAttr));
    String serializeAttr = attributes.getProperty("serialize");
    Boolean serialize = serializeAttr ==
        null || serializeAttr.length() <= 0 ? null : new Boolean("true".equals

```

```

(serializeAttr));
    type = state.getConfig().getTypeHandlerFactory().resolveAlias(type);
    Class clazz = Resources.classForName(type);
    if (readOnly == null) {readOnly = Boolean.TRUE; }
    if (serialize == null) { serialize = Boolean.FALSE; }
    CacheModelConfig cacheConfig = state.getConfig().newCacheModelConfig(id,
    (CacheController) Resources.instantiate(clazz), readOnly.booleanValue(), serialize.
    booleanValue());
    state.setCacheConfig(cacheConfig);
}

```

上述代码实现获取 cacheModel 节点属性信息, 包括 id、clazz、readOnly、serialize 等, 然后用这些参数创建一个 CacheModelConfig 对象。让我们看看 CacheModelConfig 对象的构造方法。

```

CacheModelConfig(SqlMapConfiguration config, String id, CacheController controller,
boolean readOnly, boolean serialize) {
    this.errorContext = config.getErrorContext();
    this.cacheModel = new CacheModel();
    SqlMapClientImpl client = config.getClient();
    errorContext.setActivity("building a cache model");
    cacheModel.setReadOnly(readOnly);
    cacheModel.setSerialize(serialize);
    errorContext.setObjectId(id + " cache model");
    errorContext.setMoreInfo("Check the cache model type.");
    cacheModel.setId(id);
    cacheModel.setResource(errorContext.getResource());
    try {
        cacheModel.setCacheController(controller);
    } catch (Exception e) {
        throw new RuntimeException("Error setting Cache Controller Class.
Cause: " + e, e);
    }
    errorContext.setMoreInfo("Check the cache model configuration.");
    if (client.getDelegate().isCacheModelsEnabled()) {
        client.getDelegate().addCacheModel(cacheModel);
    }
    errorContext.setMoreInfo(null);
    errorContext.setObjectId(null);
}

```

在该构造方法中, 首先实例化一个 CacheModel 对象, 接着对 CacheModel 对象进行初始化赋值工作。然后通过 SqlMapConfiguration 对象来获取 SqlMapExecutorDelegate 对象, 最后把 CacheModel 对象赋值到 SqlMapExecutorDelegate 对象中, 完成解析工作。

(2) cacheModel 节点下属性节点的解析和处理

调用 addCacheModelNodelets 方法的 parser.addNodelet("/sqlMap/CacheModel/property"、new Nodelet()方法, 实质调用是 Nodelet 匿名类的 process 方法, 代码如下。

```

public void process(Node node) throws Exception {

```

```

        state.getConfig().getErrorContext().setMoreInfo("Check the cache model
properties.");
        Properties attributes = NodeletUtils.parseAttributes(node, state.getGlobal
Props());
        String name = attributes.getProperty("name");
        String value = NodeletUtils.parsePropertyTokens(attributes.getProperty
("value"), state.getGlobalProps());
        state.getCacheProps().setProperty(name, value);
    }

```

其实现的内容是把 cacheModel 节点下属性节点的信息传递到 XmlParserState 实例化对象。我们知道，一旦进入这个对象，主要是用于数据转化的中转站。

(3) cacheModel 节点下 flushOnExecute 和 flushInterval 节点的解析和处理

调用 addCacheModelNodelets 方法的 parser.addNodelet("/sqlMap/CacheModel/flushOnExecute", new Nodelet()) 方法来解析 flushOnExecute 节点，实质调用是 Nodelet 匿名类的 process 方法，代码如下。

```

public void process(Node node) throws Exception {
    Properties childAttributes = NodeletUtils.parseAttributes(node, state.get
GlobalProps());
    String statement = childAttributes.getProperty("statement");
    state.getCacheConfig().addFlushTriggerStatement(statement);
}

```

其实现的内容是获取 flushOnExecute 节点信息，通过 CacheModelConfig 实例化对象赋值到 CacheModel 对象中。

调用 addCacheModelNodelets 方法的 parser.addNodelet("/sqlMap/CacheModel/flushInterval", new Nodelet()) 方法来解析 flushInterval 节点的处理，实质调用是 Nodelet 匿名类的 process 方法，代码如下。

```

public void process(Node node) throws Exception {
    Properties childAttributes = NodeletUtils.parseAttributes(node, state.get
GlobalProps());
    try {
        int milliseconds = childAttributes.getProperty("milliseconds") == null ? 0 :
Integer.parseInt(childAttributes.getProperty("milliseconds"));
        int seconds =
            childAttributes.getProperty("seconds") == null ? 0 : Integer.parseInt
(childAttributes.getProperty("seconds"));
        int minutes =
            childAttributes.getProperty("minutes") == null ? 0 : Integer.parseInt
(childAttributes.getProperty("minutes"));
        int hours =
            childAttributes.getProperty("hours") == null ? 0 : Integer.parseInt
(childAttributes.getProperty("hours"));
        state.getCacheConfig().setFlushInterval(hours, minutes, seconds, milliseconds);
    } catch (NumberFormatException e) {
        throw new RuntimeException

```

```

        ("Error building cache in '" + "resourceNAME" +
         "'. Flush interval milliseconds must be a valid long integer
         value. Cause: " + e, e);
    }
}

```

其实现的内容是获取 flushInterval 节点的节点信息，通过 CacheModelConfig 实例化对象赋值到 CacheModel 对象中。

4. cacheModel 节点解析和处理后的结果

对 SQL Map XML 映射文件中 cacheModel 节点的处理结果是：实例化一个 CacheModel 对象，并最终保存在 SqlMapExecutorDelegate 实例化对象中 HashMap 类型的属性 cacheModels 中。cacheModels 是一个 HashMap，所以保存的格式按照主键是 cacheModel 节点的 id 内容，而保存的对象就是用这个节点信息实例化的 CacheModel 对象。其对象转移过程路线图如图 6-27 所示。cacheModel 节点属性在经过了几次的短途跋涉，最后转化为 CacheModel 对象的属性，其对应关系如表 6-10 所示。

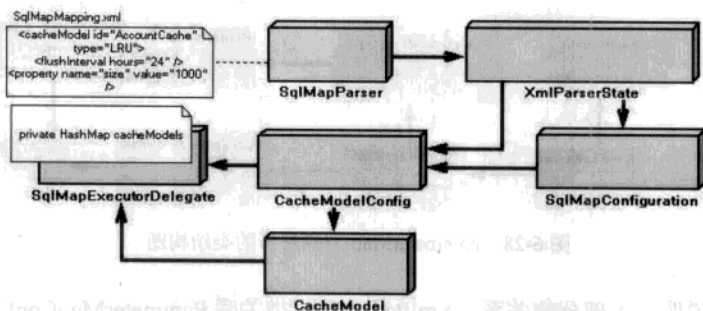


图 6-27 CacheModel 对象转移过程路线图

表 6-10 cacheModel 节点属性和 CacheModel 对象属性映射关系

序号	cacheModel 节点下属性节点	CacheModel 对象属性	类 型
1	id	id	String
2	type	controller	CacheController
3	readOnly	readOnly	boolean
4	serialize	Serialize	boolean
5		lastFlush	long
6	flushInterval	flushInterval	long
7	flushInterval	flushIntervalSeconds	long
8		flushTriggerStatements	Set
9		resource	String

在这里进行了一些转化。其中有一些字符类型转化为实例化对象，转化如表 6-11 所示。

表 6-11 字符类型的节点到实例化对象的转化

序号	节点值	类别	备注
1	FIFO	com.ibatis.sqlmap.engine.cache.fifo.FifoCacheController	
2	LRU	com.ibatis.sqlmap.engine.cache.lru.LruCacheController	
3	MC	com.ibatis.sqlmap.engine.cache.memory.MemoryCacheController	
4	OSCache	com.ibatis.sqlmap.engine.cache.oscache.OSCacheController	

6.3.5 parameterMap 节点的解析和转化

解析 parameterMap 节点与 cacheModel 节点基本是相同的。

1. 解析 parameterMap 节点的类结构图

对 parameterMap 节点的解析要涉及以下这些类: SqlMapConfigParser 类、SqlMap Parser 类、NodeletParser 类、Nodelet 接口、ParameterMapConfig 类、ParameterMap 类、ParameterMapping 类和 SqlMapExecutorDelegate Class 类, 其类结构如图 6-28 所示。

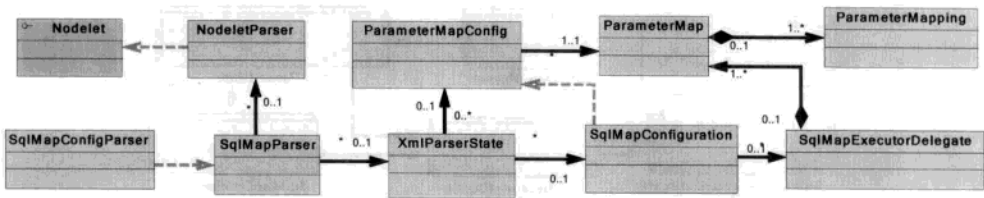


图 6-28 parameterMap 节点解析的类结构图

这里主要说明后半部分的关系。XmlParserState 类关联 ParameterMapConfig 类, 是一种一对多的关系。ParameterMapConfig 类关联 ParameterMap 类。ParameterMap 类组合多个 ParameterMapping 类, 即一个 ParameterMap 对象会包含多个 ParameterMapping 对象。而 ParameterMap 类作为多对一被 SqlMapExecutorDelegate 类聚合, 即 SqlMapExecutorDelegate 对象有一个 HashMap 变量, 包含多个 ParameterMap 对象。

2. parameterMap 节点处理的总体序列图和活动图

解析 SQL Map 的 parameterMap 节点, 其 parameterMap 节点的结果内容如下所示。

```
<?xml version="1.0" encoding="UTF-8" ?>
<sqlMap namespace="Account">
  <parameterMap id="productParam" class="product">
    <parameter property="id" jdbcType="id" typeName="id" javaType="id"
resultMap="id"
      nullValue="id" mode="id" typeHandler="id" numericScale="id"/>
  </parameterMap>
</sqlMap>
```

由上面可以看到, 在 `parameterMap` 节点中本身有 `id` 和 `class` 等两个属性。在 `parameterMap` 节点下只有 `parameter` 节点, 该节点有 `property`、`jdbcType`、`typeName`、`javaType`、`resultMap`、`nullValue`、`mode`、`typeHandler`、`numericScale` 等属性。对 `parameterMap` 节点的解析和转化如图 6-29 所示。

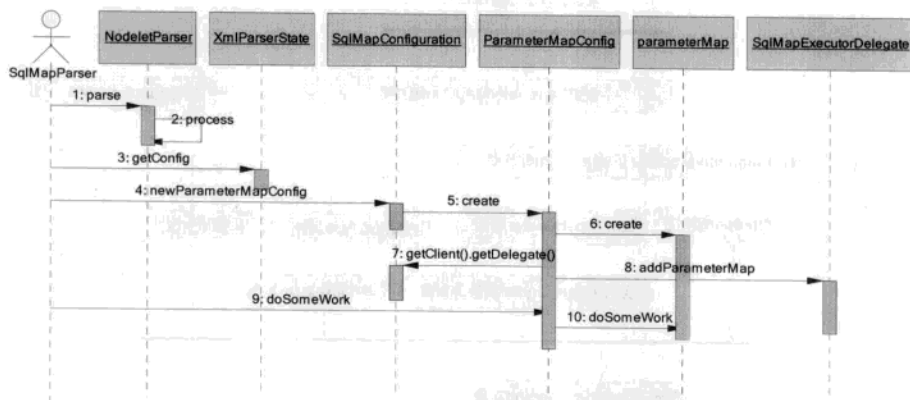


图 6-29 解析 `ParameterMap` 节点的序列图

其处理过程与 `cacheModel` 节点的解析和转化基本相同, 只是配置对象为 `ParameterMap Config` 对象, 实例化对象是 `ParameterMap` 对象。

但是其中的第 9 步骤和第 10 步骤, `SqlMapParser` 对象对 `parameterMap` 节点的属性和 `parameterMap` 节点下的属性节点进行处理。由于都是在 `SqlMapParser` 对象、`ParameterMap Config` 对象、`ParameterMap` 对象、`ParameterMapping` 对象和 `SqlMapExecutorDelegate` 对象之间来回处理, 如果用序列图恐怕比较难以理解, 故改用活动图来进行说明并有必要来解释一下。

在解析 `parameterMap` 节点, 首先是从 `SqlMapParser` 类的 `addParameterMapNodelets` 方法开始的。`addParameterMapNodelets` 方法实现节点解析的内容及说明如表 6-12 所示。其实现如图 6-30 所示。这是一个活动图, 首先解析和处理解析 `ParameterMap` 节点, 根据获得的信息生成 `parameterMap` 对象, 接着是解析 `ParameterMap` 节点下的 `parameter` 节点, 根据获得的信息生成创建 `ParameterMapping` 对象。其次把所有生成的 `ParameterMapping` 对象添加到 `parameterMap` 对象中, 最后把所有生成的 `parameterMap` 对象添加到 `SqlMapExecutorDelegate` 对象中。

表 6-12 `addParameterMapNodelets` 方法实现节点解析内容及说明

序号	方法名称	实现功能
1	<code>parser.addNodelet("/sqlMap/parameterMap", new Nodelet())</code>	解析和处理 <code>parameterMap</code> 节点属性
2	<code>parser.addNodelet("/sqlMap/parameterMap/end()", new Nodelet())</code>	完成整个 <code>parameterMap</code> 节点的解析
3	<code>parser.addNodelet("/sqlMap/parameterMap/parameter", new Nodelet())</code>	解析和处理 <code>parameterMap</code> 节点下的 <code>parameter</code> 节点

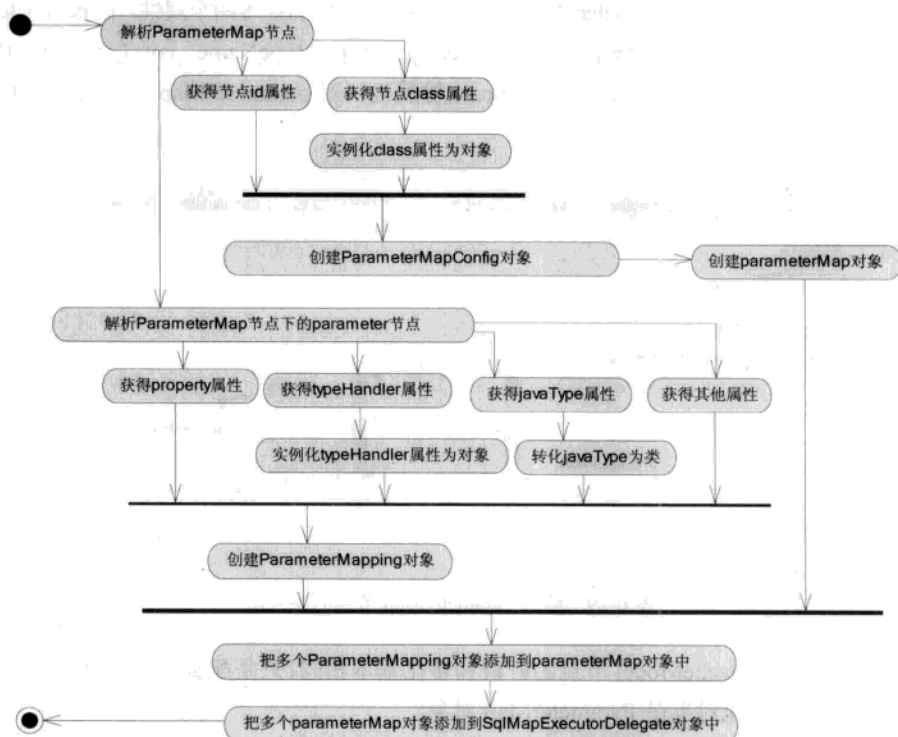


图 6-30 解析 ParameterMap 节点的实现图

3. 解析的步骤和代码说明

下面进行代码说明。

(1) SqlMapParser 对象对 parameterMap 节点的属性进行处理

调用 addParameterMapNodelets 方法的 parser.addNodelet("/sqlMap/parameterMap", new Nodelet()方法, 实际上是最后调用 Nodelet 匿名类的 process 方法, 代码如下。

```

public void process(Node node) throws Exception {
    Properties attributes = NodeletUtils.parseAttributes(node, state.getGlobal
    Props());
    String id = state.applyNamespace(attributes.getProperty("id"));
    String parameterClassName = attributes.getProperty("class");
    parameterClassName = state.getConfig().getHandlerFactory().resolve
    Alias(parameterClassName);
    try {
        state.getConfig().getErrorContext().setMoreInfo("Check the parameter
        class.");
        ParameterMapConfig paramConf =
            state.getConfig().newParameterMapConfig(id,
            Resources.classForName(parameterClassName));
        state.setParamConfig(paramConf);
    } catch (Exception e) {
    }
}
    
```



```

        throw new SqlMapException("..." + e, e);
    }
}

```

获取 parameterMap 节点中属性 id 和 class 的内容, 并调用 newParameterMapConfig 方法实例化一个 ParameterMapConfig 对象, 把属性 id 和有属性 class 实例化的对象作为参数传递过去。

ParameterMapConfig 对象对 ParameterMap 对象进行处理。

```

ParameterMapConfig(SqlMapConfiguration config, String id, Class parameterClass) {
    this.config = config;
    this.errorContext = config.getErrorContext();
    this.client = config.getClient();
    errorContext.setActivity("building a parameter map");
    parameterMap = new ParameterMap(client.getDelegate());
    parameterMap.setId(id);
    parameterMap.setResource(errorContext.getResource());
    errorContext.setObjectId(id + " parameter map");
    parameterMap.setParameterClass(parameterClass);
    errorContext.setMoreInfo("Check the parameter mappings.");
    this.parameterMappingList = new ArrayList();
    client.getDelegate().addParameterMap(parameterMap);
}

```

创建 ParameterMap 实例化对象, 并把这个实例化对象添加到 SqlMapExecutorDelegate 对象中的 HashMap 中。

(2) SqlMapParser 对象对 parameterMap 节点的 parameter 节点进行处理

调用 addParameterMapNodelets 方法的 parser.addNodelet("/sqlMap/parameterMap/parameter", new Nodelet() 方法, 实际上最后调用是 Nodelet 匿名类的 process 方法, 代码如下。

```

public void process(Node node) throws Exception {
    Properties childAttributes = NodeletUtils.parseAttributes(node, state.getGlobalProps());
    String propertyName = childAttributes.getProperty("property");
    String jdbcType = childAttributes.getProperty("jdbcType");
    String type = childAttributes.getProperty("typeName");
    String javaType = childAttributes.getProperty("javaType");
    String resultMap = state.applyNamespace(childAttributes.getProperty("resultMap"));
    String nullValue = childAttributes.getProperty("nullValue");
    String mode = childAttributes.getProperty("mode");
    String callback = childAttributes.getProperty("typeHandler");
    String numericScaleProp = childAttributes.getProperty("numericScale");

    callback = state.getConfig().getTypeHandlerFactory().resolveAlias(callback);
    Object typeHandlerImpl = null;
    if (callback != null) {
        typeHandlerImpl = Resources.instantiate(callback);
    }
}

```

```

        javaType = state.getConfig().getTypeHandlerFactory().resolveAlias(javaType);
        Class javaClass = null;
        try {
            if (javaType != null && javaType.length() > 0) {
                javaClass = Resources.classForName(javaType);
            }
        } catch (ClassNotFoundException e) {
            throw new RuntimeException("Error setting javaType on parameter mapping.
Cause: " + e);
        }

        Integer numericScale = null;
        if (numericScaleProp != null) {
            numericScale = new Integer(numericScaleProp);
        }

        state.getParamConfig().addParameterMapping(propertyName, javaClass, jdbc
Type, nullValue, mode, type, numericScale, typeHandlerImpl, resultMap);
    }

```

获取 parameterMap 节点下属性节点的各种属性，并调用 ParameterMapConfig 对象的 addParameterMapping 方法，把这些获得的属性作为参数传递过去。

(3) ParameterMapConfig 对象对 parameterMap 节点的属性传递参数的处理

```

    public void addParameterMapping(String propertyName, Class javaClass, String jdbc
Type, String nullValue, String mode, String outParamType, Integer numericScale, Object
typeHandlerImpl, String resultMap) {
        errorContext.setObjectId(propertyName + " mapping of the " + parameterMap.getId()
+ " parameter map");

        TypeHandler handler;
        if (typeHandlerImpl != null) {
            errorContext.setMoreInfo("...");
            if (typeHandlerImpl instanceof TypeHandlerCallback) {
                handler = new CustomTypeHandler((TypeHandlerCallback) typeHandlerImpl);
            } else if (typeHandlerImpl instanceof TypeHandler) {
                handler = (TypeHandler) typeHandlerImpl;
            } else {
                throw new RuntimeException
                    ("The class '" + typeHandlerImpl + "' is not a valid implementation of
TypeHandler or TypeHandlerCallback");
            }
        } else {
            errorContext.setMoreInfo("Check the parameter mapping property type or
name.");
            handler =
                config.resolveTypeHandler(client.getDelegate().getTypeHandlerFactory
(), parameterMap.getParameterClass(), propertyName, javaClass, jdbcType);
        }
        ParameterMapping mapping = new ParameterMapping();
        mapping.setPropertyName(propertyName);
    }

```

```
mapping.setJdbcTypeName(jdbcType);
mapping.setTypeNames(outParamType);
mapping.setResultMapName(resultMap);
mapping.setNullValue(nullValue);
if (mode != null && mode.length() > 0) {
    mapping.setMode(mode);
}
mapping.setTypeHandler(handler);
mapping.setJavaType(javaClass);
mapping.setNumericScale(numericScale);
parameterMappingList.add(mapping);
parameterMap.setParameterMappingList(parameterMappingList);
```

4. 解析和处理后的结果

最终，对 SQL Map XML 映射文件中 parameterMap 节点的处理结果如下。

(1) 对 parameterMap 节点解析的数据转移图和结果

实例化一个 `ParameterMap` 对象，并最终保存在 `SqlMapExecutorDelegate` 实例化对象中 `HashMap` 类型的属性 `parameterMaps`。`parameterMaps` 是一个 `HashMap` 类型的变量，所以保存的格式按照主键是 `ParameterMap` 节点的 `id` 内容，而保存的对象就是用这个节点信息实例化 `ParameterMap` 对象。数据转移路线如图 6-31 所示。

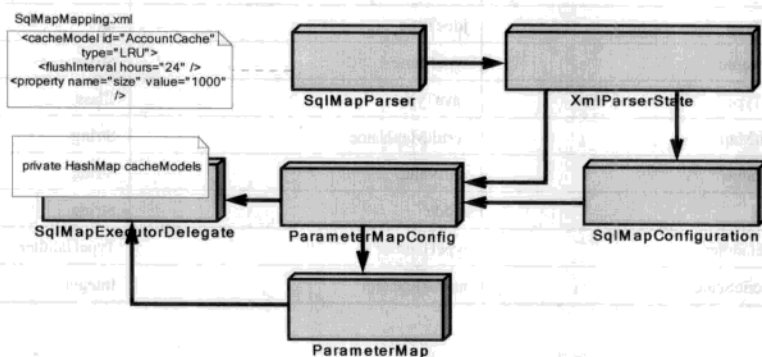


图 6-31 parameterMap 节点解析的数据转移路线图

parameterMap 节点属性和 ParameterMap 对象属性的映射关系如表 6-13 所示。

表 6-13 parameterMap 节点属性和 ParameterMap 对象属性映射关系

序号	parameterMap 节点属性	ParameterMap 对象属性	类 型
1	id	id	String
2	class	parameterClass	Class
3	parameterMap 节点下属性节点	parameterMappings	ParameterMapping 数组

(2) 对 parameterMap 节点下 parameter 节点解析的数据转移图和结果对 parameterMap

节点下的属性节点就是实例化 ParameterMapping 对象，如图 6-32 所示。

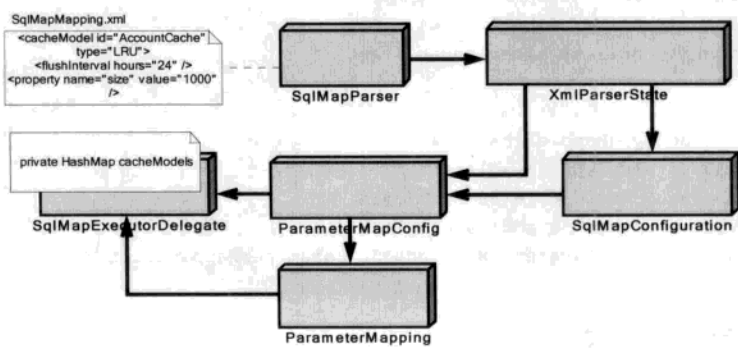


图 6-32 parameter 节点解析的数据转移路线图

parameterMap 节点下属性节点与 ParameterMapping 对象属性的映射关系如表 6-14 所示。

表 6-14 parameterMap 节点下属性节点与 ParameterMapping 对象属性映射关系

序号	parameterMap 节点下属性节点	ParameterMapping 对象属性	类 型
1	property	propertyName	String
2	jdbcType	jdbcType	int
3	typeName	typeName	String
4	JavaType	JavaType	Class
5	resultMap	resultMapName	String
6	nullValue	nullValue	String
7	mode	mode	String
8	typeHandler	typeHandler	TypeHandler
9	numericScale	numericScale	Integer

6.3.6 resultMap 节点的解析和转化

解析 resultMap 节点与 parameterMap 节点基本是相同的。但是由于变量更多，所以比解析 parameterMap 节点更复杂。按照 resultMap 节点处理的总体序列图、活动图、各个解析步骤和最后实现的结果来进行说明。

1. 解析 resultMap 节点的类结构图

对 resultMap 节点的解析要涉及以下这些类：SqlMapConfigParser 类、SqlMapParser 类、NodeletParser 类、Nodelet 接口、ResultMapConfig 类、ResultMap 类、ResultMapping 类、Discriminator 类和 SqlMapExecutorDelegate 类，其类结构如图 6-33 所示。

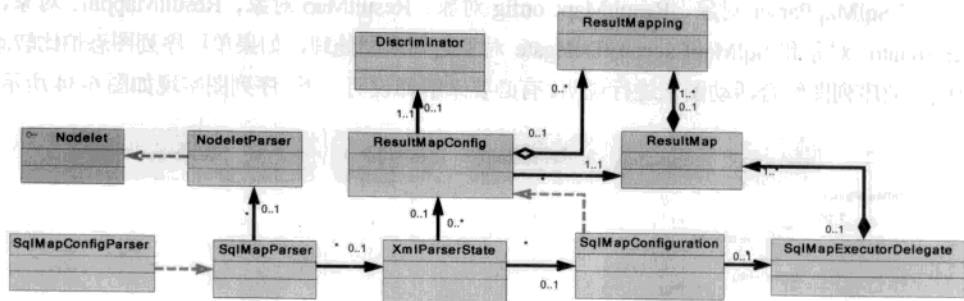


图 6-33 resultMap 节点解析的类结构图

主要说明后半部分的关系。XmlParserState 类关联 ResultMapConfig 类，是一种一对多关系。ResultMapConfig 类关联 ResultMap 类。ResultMap 类组合多个 ResultMapping 类，即一个 ResultMap 对象会包含多个 ResultMapping 对象。而 ResultMap 类作为多对一被 SqlMapExecutorDelegate 类聚合，即 SqlMapExecutorDelegate 对象有一个 HashMap 变量，包含多个 ResultMap 对象。

2. resultMap 节点处理的总体序列图和活动图

解析 SQL Map 的 resultMap 节点，其 resultMap 节点的结果内容如下。

```

<?xml version="1.0" encoding="UTF-8" ?>
<sqlMap namespace="Account">
  <resultMap id="productResult" class="product" extends="" xmlName=""
    groupBy="">
    <result property="id" nullValue="" jdbcType="" javaType=""
      column="PRD_ID" columnIndex="" select="" resultMap="" typeHandler=""
      notNullColumn="" />
    <discriminator nullValue="" jdbcType="" javaType=""
      column="PRD_ID" columnIndex="" typeHandler="">
      <subMap value="" resultMap="" />
    </discriminator>
  </resultMap>
</sqlMap>

```

由上面可以看到，在 resultMap 节点中本身有 id、class、extends、xmlName 和 groupBy 等属性。在 resultMap 节点有两类节点，一类节点是 result 节点，该节点有 property、nullValue、jdbcType、JavaType、column、columnIndex、select、resultMap、typeHandler 和 notNullColumn 等属性。另一类节点是 discriminator 节点，该节点有 nullValue、jdbcType、JavaType、column、columnIndex 和 typeHandler 等属性。discriminator 节点下包括 subMap 节点，subMap 节点有 value 和 resultMap 属性。对 resultMap 节点的解析和转化如图 6-33 所示。

其处理过程与 parameterMap 节点的解析和转化基本相同，只是配置对象为 ResultMapConfig 对象，实例化对象是 ResultMap 对象。

但是其中第 9 步骤和第 10 步骤，SqlMapParser 对象对 resultMap 节点的属性进行处理。由

于都是在 SqlMapParser 对象、ResultMapConfig 对象、ResultMap 对象、ResultMapping 对象、Discriminator 对象和 SqlMapExecutorDelegate 对象之间来回处理，如果单用序列图恐怕比较难以理解，故序列图结合活动图来进行说明。有必要来解释说明一下。序列图实现如图 6-34 所示。

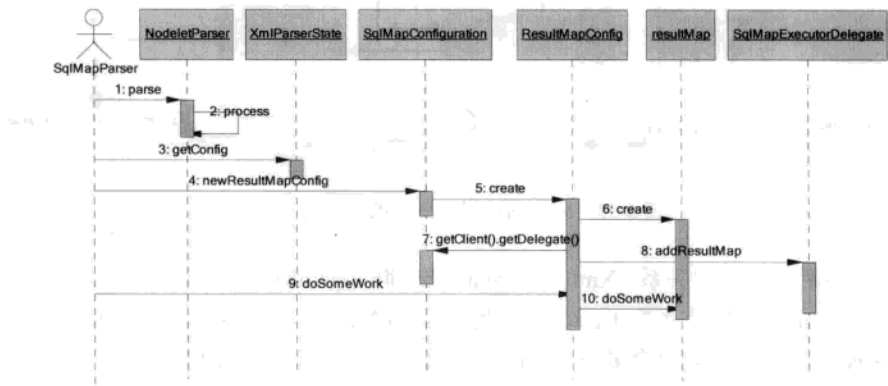


图 6-34 resultMap 节点解析的序列图

在解析 resultMap 节点，首先是从 SqlMapParser 类的 addResultMapNodelets 方法开始的。addResultMapNodelets 方法实现节点解析如表 6-15 所示。

表 6-15 addResultMapNodelets 方法实现节点解析方法及其功能列表

序号	方法名称	实现功能
1	parser.addNodelet("/sqlMap/resultMap/end()", new Nodelet())	完成整个 resultMap 节点的解析
2	parser.addNodelet("/sqlMap/resultMap", new Nodelet())	解析和处理 resultMap 节点属性
3	parser.addNodelet("/sqlMap/resultMap/result", new Nodelet())	解析和处理 resultMap 节点下 result 节点
4	parser.addNodelet("/sqlMap/resultMap/discriminator/subMap", new Nodelet())	解析和处理 resultMap 节点下 discriminator 节点下 subMap 节点
5	parser.addNodelet("/sqlMap/resultMap/discriminator", new Nodelet())	解析和处理 resultMap 节点下 discriminator 节点属性

图 6-35 是一个活动图，首先解析和处理解析 resultMap 节点的属性，根据获得的信息生成 ResultMapConfig 对象，在 ResultMapConfig 对象中创建 ResultMap 对象，把所有生成的 ResultMap 对象添加到 SqlMapExecutorDelegate 对象中。接着是解析 resultMap 节点下 discriminator 节点的 subMap 节点，根据获得的信息赋值到 XmlParserState 对象，作为一个中间变量。最后是解析 resultMap 节点下的 discriminator，把获取的信息赋值到 ResultMapConfig 对象中。

3. resultMap 节点解析的步骤和代码说明

下面进行代码说明。分别按照解析和处理 resultMap 节点属性，resultMap 节点下 result 节点，resultMap 节点下 discriminator 节点下 subMap 节点，resultMap 节点下 discriminator 节点属性等顺序来说明。

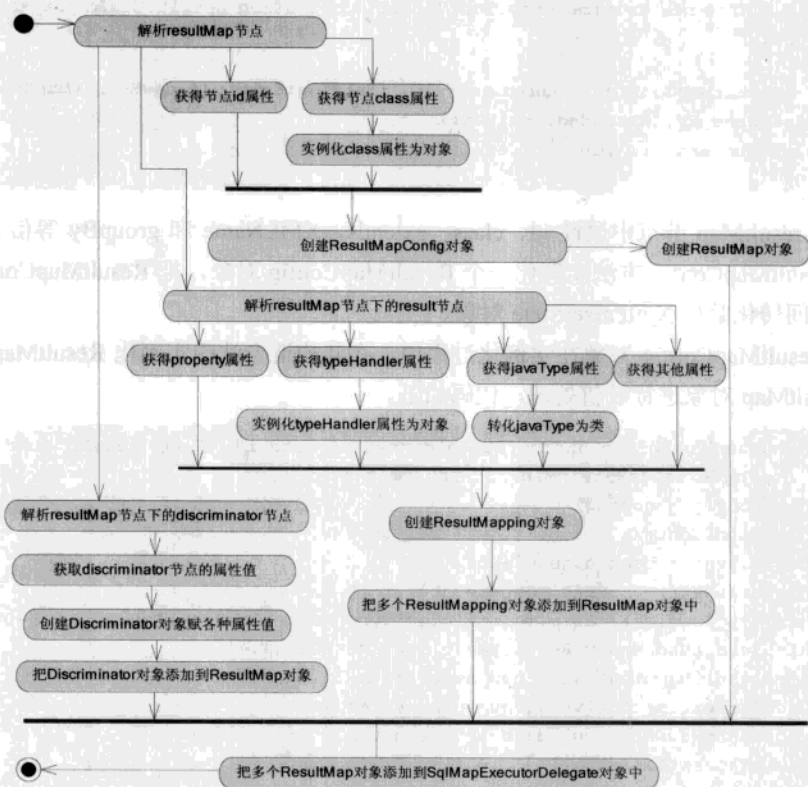


图 6-35 resultMap 节点解析的实现图

(1) SqlMapParser 对象对 resultMap 节点属性进行解析。

调用 addResultMapNodelets 方法的 parser.addNodelet("/sqlMap/resultMap", new Nodelet()) 方法, 实际上是最后调用 Nodelet 匿名类的 process 方法, 代码如下。

```

public void process(Node node) throws Exception {
    Properties attributes = NodeletUtils.parseAttributes(node, state.getGlobal
    Props());

    String id = state.applyNamespace(attributes.getProperty("id"));
    String resultClassName = attributes.getProperty("class");
    String extended = state.applyNamespace(attributes.getProperty("extends"));
    String xmlName = attributes.getProperty("xmlName");
    String groupBy = attributes.getProperty("groupBy");

    resultClassName = state.getConfig().getHandlerFactory().resolveAlias
    (resultClassName);
    Class resultClass;
    try {
        state.getConfig().getErrorContext().setMoreInfo("Check the result class.");
        resultClass = Resources.classForName(resultClassName);
    } catch (Exception e) {

```



```

        throw new RuntimeException("Error configuring Result.Could not set Result
Class.Cause: " + e, e);
    }
    resultMapConfig resultConf = state.getConfig().newResultMapConfig(id,
resultClass,groupBy, extended, xmlName);
    state.setResultConfig(resultConf);
}

```

获取 resultMap 节点中属性 id、class、extends、XMLName 和 groupBy 等信息，并调用 newResultMapConfig 方法实例化一个 resultMapConfig 对象，把 resultMapConfig 对象传递给中间转化数据 XmlParserState 对象。

当 resultMapConfig 对象在实例化过程中，在其构造方法中实例化 resultMap 对象，并对 resultMap 对象进行赋值处理，代码如下。

```

ResultMapConfig(SqlMapConfiguration config, String id, Class resultClass, String
groupBy, String extendsResultMap, String xmlName) {
    this.config = config;
    this.errorContext = config.getErrorContext();
    this.client = config.getClient();
    this.delegate = config.getDelegate();
    this.typeHandlerFactory = config.getTypeHandlerFactory();
    this.resultMap = new ResultMap(client.getDelegate());
    this.resultMappingList = new ArrayList();
    errorContext.setActivity("building a result map");
    errorContext.setObjectId(id + " result map");
    resultMap.setId(id);
    resultMap.setXmlName(xmlName);
    resultMap.setResource(errorContext.getResource());
    if (groupBy != null && groupBy.length() > 0) {
        StringTokenizer parser = new StringTokenizer(groupBy, ",", false);
        while (parser.hasMoreTokens()) {
            resultMap.addGroupByProperty(parser.nextToken());
        }
    }
    resultMap.setResultClass(resultClass);
    errorContext.setMoreInfo("Check the extended result map.");
    if (extendsResultMap != null) {
        ResultMap extendedResultMap = (ResultMap) client.getDelegate().getResultMap
(extendsResultMap);
        ResultMapping[] resultMappings = extendedResultMap.getResultMappings();
        for (int i = 0; i < resultMappings.length; i++) {
            resultMappingList.add(resultMappings[i]);
        }
        List nestedResultMappings = extendedResultMap.getNestedResultMappings();
        if (nestedResultMappings != null) {
            Iterator iter = nestedResultMappings.iterator();
            while (iter.hasNext()) {
                resultMap.addNestedResultMappings((ResultMapping) iter.next());
            }
        }
    }
}

```

```

        if (groupBy == null || groupBy.length() == 0) {
            if (extendedResultMap.hasGroupBy()) {
                Iterator i = extendedResultMap.groupByProps();
                while (i.hasNext()) {
                    resultMap.addGroupByProperty((String) i.next());
                }
            }
        }
    }

    errorContext.setMoreInfo("Check the result mappings.");
    resultMappingIndex = resultMappingList.size();
    resultMap.setResultMappingList(resultMappingList);
    client.getDelegate().addResultMap(resultMap);
}

```

(2) SqlMapParser 对象对 result 节点属性进行处理。

调用 addResultMapNodelets 方法的 parser.addNodelet("/sqlMap/resultMap/result", new Nodelet()方法, 实际上是最后调用 Nodelet 匿名类的 process 方法, 即对 ResultMap 进行实例化。代码如下。

```

public void process(Node node) throws Exception {
    Properties childAttributes = NodeletUtils.parseAttributes(node, state.getGlobalProps());

    String propertyName = childAttributes.getProperty("property");
    String nullValue = childAttributes.getProperty("nullValue");
    String jdbcType = childAttributes.getProperty("jdbcType");
    String javaType = childAttributes.getProperty("javaType");
    String columnName = childAttributes.getProperty("column");
    String columnIndexProp = childAttributes.getProperty("columnIndex");
    String statementName = childAttributes.getProperty("select");
    String resultMapName = childAttributes.getProperty("resultMap");
    String callback = childAttributes.getProperty("typeHandler");
    String notNullColumn = childAttributes.getProperty("notNullColumn");

    state.getConfig().getErrorContext().setMoreInfo("Check the result mapping property type or name.");
    Class javaClass = null;
    try {
        javaType = state.getConfig().getTypeHandlerFactory().resolveAlias(javaType);
        if (javaType != null && javaType.length() > 0) {
            javaClass = Resources.className(javaType);
        }
    } catch (ClassNotFoundException e) {
        throw new RuntimeException("Error setting java type on result discriminator mapping. Cause: " + e);
    }

    state.getConfig().getErrorContext().setMoreInfo("Check the result mapping typeHandler attribute '" + callback + "' (must be a TypeHandler or TypeHandlerCallback implementation).");
    Object typeHandlerImpl = null;
}

```

```

        try {
            if (callback != null && callback.length() > 0) {
                callback = state.getConfig().getHandlerFactory().resolveAlias(callback);
                typeHandlerImpl = Resources.instantiate(callback);
            }
        } catch (Exception e) {
            throw new RuntimeException("Error occurred during custom type handler
configuration. Cause: " + e, e);
        }

        Integer columnIndex = null;
        if (columnIndexProp != null) {
            try {
                columnIndex = new Integer(columnIndexProp);
            } catch (Exception e) {
                throw new RuntimeException("Error parsing column index. Cause: " + e, e);
            }
        }

        state.getResultConfig().addResultMapping(propertyName, columnName, column
Index, javaClass, jdbcType, nullValue, notNullColumn, statementName, resultMapName,
typeHandlerImpl);
    }
}

```

Nodelet 匿名类的 process 方法获得获取 resultMap 节点下 result 节点中属性 property、nullValue、jdbcType、JavaType、column、columnIndex、select、resultMap、typeHandler 和 notNullColumn 等信息，对这些信息进行加工处理，并通过中间转化数据 XmlParserState 对象获得 ResultMapConfig 对象，调用 ResultMapConfig 对象的 addResultMapping 方法进行进一步处理。调用 addResultMapping 方法的代码如下。

```

    public void addResultMapping(String propertyName, String columnName, Integer
columnIndex, Class javaClass, String jdbcType, String nullValue, String notNullColumn,
String statementName, String resultMapName, Object impl) {
        errorContext.setObjectId(propertyName + " mapping of the " + resultMap.getId()
+ " result map");
        TypeHandler handler;
        if (impl != null) {
            if (impl instanceof TypeHandlerCallback) {
                handler = new CustomTypeHandler((TypeHandlerCallback) impl);
            } else if (impl instanceof TypeHandler) {
                handler = (TypeHandler) impl;
            } else {
                throw new RuntimeException("The class '" + impl + "' is not a valid
implementation of TypeHandler or TypeHandlerCallback");
            }
        } else {
            handler = config.resolveTypeHandler(client.getDelegate().getHandler
Factory(), resultMap.getResultClass(), propertyName, javaClass, jdbcType, true);
        }
        ResultMapping mapping = new ResultMapping();
    }
}

```

```

mapping.setPropertyName(propertyName);
mapping.setColumnNames(columnNames);
mapping.setJdbcTypeName(jdbcType);
mapping.setTypeHandler(handler);
mapping.setNullValue(nullValue);
mapping.setNotNullColumn(notNullColumn);
mapping.setStatementName(statementName);
mapping.setNestedResultMapName(resultMapName);
if (resultMapName != null && resultMapName.length() > 0) {
    resultMap.addNestedResultMappings(mapping);
}
mapping.setJavaType(javaClass);
if (columnIndex != null) {
    mapping.setColumnIndex(columnIndex.intValue());
} else {
    resultMappingIndex++;
    mapping.setColumnIndex(resultMappingIndex);
}
resultMappingList.add(mapping);
resultMap.setResultMappingList(resultMappingList);
}

```

ResultMapConfig 对象的 addResultMapping 方法继续对获取信息进行处理, 创建一个 ResultMapping 实例化对象, 然后把这些信息转移给 ResultMapping 对象, 并把这个 ResultMapping 对象加入到 List 类型的变量 resultMappingList 中, 最后把变量 resultMappingList 赋值给 resultMap 对象。

(3) SqlMapParser 对象对 discriminator 节点的 subMap 节点进行解析

调用 addResultMapNodelets 方法的 parser.addNodelet("/sqlMap/resultMap/discriminator/subMap", new Nodelet()) 方法, 实际上是最后调用 Nodelet 匿名类的 process 方法, 代码如下。

```

public void process(Node node) throws Exception {
    Properties childAttributes = NodeletUtils.parseAttributes(node, state.getGlobalProps());
    String value = childAttributes.getProperty("value");
    String resultMap = childAttributes.getProperty("resultMap");
    resultMap = state.applyNamespace(resultMap);
    state.getResultConfig().addDiscriminatorSubMap(value, resultMap);
}

```

获取 subMap 节点的 value 和 resultMap 信息并进行处理后, 通过 XmlParserState 对象获得 ResultMapConfig 对象并调用 ResultMapConfig 对象的 addDiscriminatorSubMap 方法。其方法代码如下。

```

public void addDiscriminatorSubMap(Object value, String resultMap) {
    if (discriminator == null) {
        throw new RuntimeException("...");
    }
    discriminator.addSubMap(value.toString(), resultMap);
}

```

```
}
```

调用 Discriminator 对象的 addSubMap 方法，代码如下。

```
public void addSubMap(String discriminatorValue, String resultMapName) {
    if (subMaps == null) {
        subMaps = new HashMap();
    }
    subMaps.put(discriminatorValue, resultMapName);
}
```

在 Discriminator 对象的 Map 类型变量 subMaps 加入一组数值而已。

(4) SqlMapParser 对象对 discriminator 节点属性进行解析

调用 addResultMapNodelets 方法的 parser.addNodelet("/sqlMap/resultMap/discriminator", new Nodelet()方法，实际上是最后调用 Nodelet 匿名类的 process 方法，代码如下。

```
public void process(Node node) throws Exception {
    Properties childAttributes = NodeletUtils.parseAttributes(node, state.getGlobalProps());

    String nullValue = childAttributes.getProperty("nullValue");
    String jdbcType = childAttributes.getProperty("jdbcType");
    String javaType = childAttributes.getProperty("javaType");
    String columnName = childAttributes.getProperty("column");
    String columnIndexProp = childAttributes.getProperty("columnIndex");
    String callback = childAttributes.getProperty("typeHandler");

    state.getConfig().getErrorContext().setMoreInfo("Check the discriminator type or name.");
    Class javaClass = null;
    try {
        javaType = state.getConfig().getTypeHandlerFactory().resolveAlias(javaType);
        if (javaType != null && javaType.length() > 0) {
            javaClass = Resources.className(javaType);
        }
    } catch (ClassNotFoundException e) {
        throw new RuntimeException("Error setting java type on result discriminator mapping. Cause: " + e);
    }

    state.getConfig().getErrorContext().setMoreInfo("Check the result mapping discriminator typeHandler attribute '" + callback + "' (must be a TypeHandlerCallback implementation).");
    Object typeHandlerImpl = null;
    try {
        if (callback != null && callback.length() > 0) {
            callback = state.getConfig().getTypeHandlerFactory().resolveAlias(callback);
            typeHandlerImpl = Resources.instantiate(callback);
        }
    } catch (Exception e) {
        throw new RuntimeException("Error occurred during custom type handler");
    }
}
```



```

configuration. Cause: " + e, e);
    }

    Integer columnIndex = null;
    if (columnIndexProp != null) {
        try {
            columnIndex = new Integer(columnIndexProp);
        } catch (Exception e) { throw new RuntimeException("Error parsing column
index. Cause: " + e, e);
        }
    }

    state.getResultConfig().setDiscriminator(columnName, columnIndex, java
Class,jdbcType,nullValue, typeHandlerImpl);
}

```

获取 discriminator 节点的 nullValue、jdbcType、JavaType、column、columnIndex 和 typeHandler 信息并进行处理后, 通过 XmlParserState 对象获得 ResultMapConfig 对象并调用 ResultMapConfig 对象的 setDiscriminator 方法。其方法代码如下。

```

public void setDiscriminator(String columnName, Integer columnIndex, Class
javaClass, String jdbcType, String nullValue, Object typeHandlerImpl) {
    TypeHandler handler;
    if (typeHandlerImpl != null) {
        if (typeHandlerImpl instanceof TypeHandlerCallback) {
            handler = new CustomTypeHandler((TypeHandlerCallback) typeHandlerImpl);
        } else if (typeHandlerImpl instanceof TypeHandler) {
            handler = (TypeHandler) typeHandlerImpl;
        } else {
            throw new RuntimeException("The class '' is not a valid implementation of
TypeHandler or TypeHandlerCallback");
        }
    } else {
        handler = config.resolveTypeHandler(client.getDelegate().getTypeHandler
Factory(), resultMap.getResultClass(), "", javaClass, jdbcType, true);
    }

    ResultMapping mapping = new ResultMapping();
    mapping.setColumnName(columnName);
    mapping.setJdbcTypeName(jdbcType);
    mapping.setTypeHandler(handler);
    mapping.setNullValue(nullValue);
    mapping.setJavaType(javaClass);
    if (columnIndex != null) {
        mapping.setColumnIndex(columnIndex.intValue());
    }

    discriminator = new Discriminator(delegate, mapping);
    resultMap.setDiscriminator(discriminator);
}

```

该方法获得传递过来的参数并进行处理, 接着赋值给一个新创建的 ResultMapping 对

象，然后以 ResultMapping 对象为构造参数创建一个 Discriminator 对象，最后把 Discriminator 对象赋值给 ResultMap 对象。

4. resultMap 节点解析和处理后的结果

对 SQL Map XML 映射文件中 resultMap 节点的处理结果是：实例化一个 ResultMap 对象，并最终保存在 SqlMapExecutorDelegate 实例化对象中 HashMap 类型的属性 resultMap 中。

(1) 对 resultMap 节点的解析

实例化一个 ResultMap 对象，并最终保存在 SqlMapExecutorDelegate 实例化对象的 HashMap 类型的属性 resultMap 中。resultMaps 是一个 HashMap 类型的变量，所以保存的格式是按照主键是 resultMap 节点的 id 内容，而保存的对象就是用这个节点信息实例化的 ResultMap 对象。数据转移如图 6-36 所示。

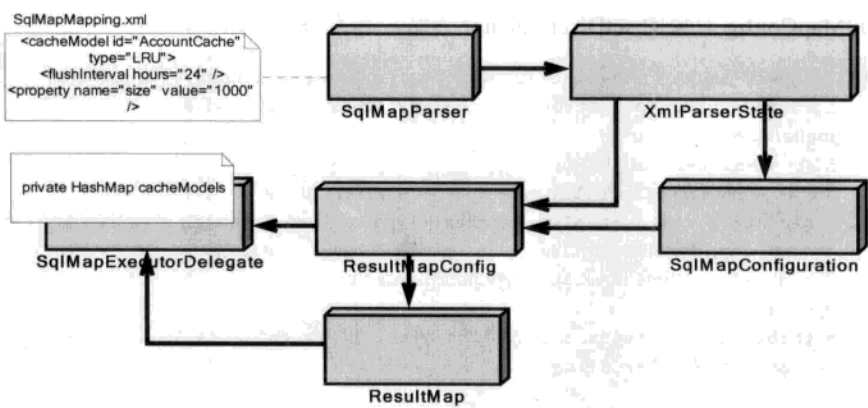


图 6-36 resultMap 节点解析的数据转移路线图

resultMap 节点属性和 ResultMap 对象属性的映射关系如表 6-16 所示。

表 6-16 resultMap 节点属性和 ResultMap 对象属性的映射关系

序号	resultMap 节点属性	ResultMap 对象属性	类 型
1	id	id	String
2	class	parameterClass	Class
3	extends	nestedResultMappings	List
4	xmlName	xmlName	String
5	groupBy	groupByProps	Set

(2) 对 resultMap 节点下的 result 节点的解析

对 resultMap 节点下的 result 节点就是实例化 ResultMapping 对象，数据转移如图 6-37 所示。

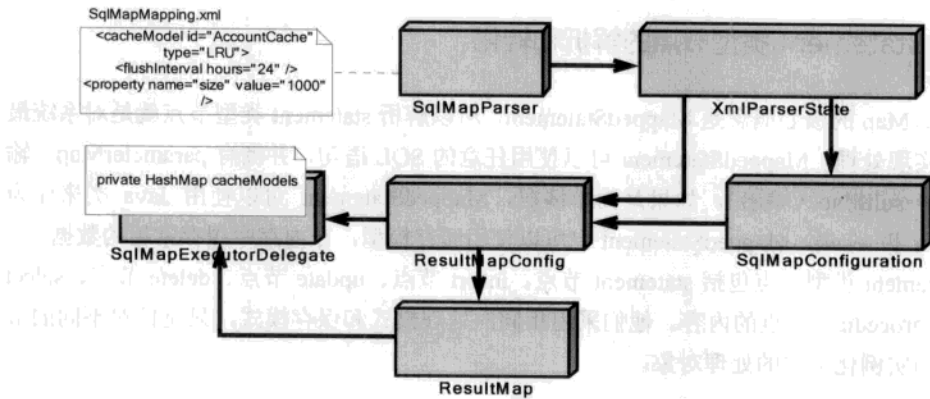


图 6-37 resultMap 节点解析的数据转移路线图

result 节点下属性节点与 ResultMapping 对象属性的映射关系如表 6-17 所示。

表 6-17 result 节点下属性节点与 ResultMapping 对象属性的映射关系

序号	result 节点属性	ResultMapping 对象属性	类 型
1	property	propertyName	String
2	jdbcType	jdbcType	int
3	typeName	typeName	String
4	JavaType	JavaType	Class
5	resultMap	resultMapName	String
6	nullValue	nullValue	String
7	mode	mode	String
8	typeHandler	typeHandler	TypeHandler
9	numericScale	numericScale	Integer

(3) 对 resultMap 节点下的 discriminator 节点就是实例化 Discriminator 对象，discriminator 节点下属性节点与 Discriminator 对象属性的映射关系如表 6-18 所示。

表 6-18 discriminator 节点下属性节点与 Discriminator 对象属性的映射关系

序号	discriminator 节点属性	Discriminator 对象属性	类 型
1	property	propertyName	String
2	jdbcType	jdbcType	int
3	typeName	typeName	String
4	JavaType	JavaType	Class
5	resultMap	resultMapName	String
6	nullValue	nullValue	String
7	mode	mode	String
8	typeHandler	typeHandler	TypeHandler
9	numericScale	numericScale	Integer

6.3.7 statement 类型节点的解析和转化

SQL Map 的核心概念是 MappedStatement，所以解析 statement 类型节点就是对系统最核心的实现处理。MappedStatement 可以使用任意的 SQL 语句，并拥有 parameterMap（输入）和 resultMap（输出）。如果是简单情况，MappedStatement 可以使用 Java 类来作为 parameter 和 result。MappedStatement 也可以使用缓存模型，在内存中缓存常用的数据。

statement 类型节点包括 statement 节点、insert 节点、update 节点、delete 节点、select 节点和 procedure 节点的内容。他们采用相同的处理模式和保存模式，只是针对不同的节点类型而实例化不同的处理对象。

1. 解析 statement 类型节点的类结构图

解析 statement 类型节点是这些节点中覆盖面最广的，基本上要涉及 SQL Map 中所有的类。解析 statement 类型节点也是把前面几个解析内容结合起来，是 SQL Map 所有节点解析的综合。

解析要涉及以下这些类：SqlMapConfigParser 类、SqlMapParser 类、NodeletParser 类、Nodelet 接口、SqlStatementParser 类、CacheModel 类、ParameterMap 类、ResultMap 类、MappedStatementConfig 类、MappedStatement 类、InsertStatement 类、UpdateStatement 类、DeleteStatement 类、SelectStatement 类、ProcedureStatement 类、CachingStatement 类、Sql 接口和 SqlMapExecutorDelegate 类。其类结构如图 6-37 所示。

这个类结构图有点复杂，左端的内容在前面已经做了描述和说明。而 SqlMapExecutorDelegate 类聚合 CacheModel 类、ParameterMap 类、ResultMap 类在前面也已经做了说明，在图 6-32 中为了不把图搞得更复杂，所以都没有标记。图 6-38 主要还是以 MappedStatementConfig 类、MappedStatement 类为中心来进行标识的。

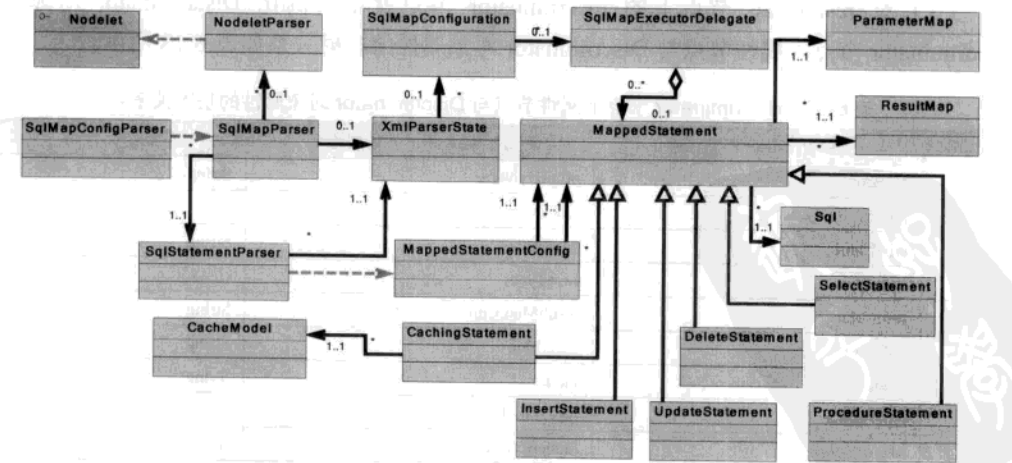


图 6-38 statement 节点解析的类结构图

SqlMapParser 类关联 SqlStatementParser 类, 表示 SqlMapParser 对象在解析 statement 类型节点的时候把这项工作转移给了 SqlStatementParser 对象。

SqlStatementParser 类关联 XmlParserState 类, 表示 SqlStatementParser 对象还要用 XmlParserState 对象作为中间转移体变量角色。SqlStatementParser 类依赖 MappedStatement 类, 表明 SqlStatementParser 是解析 statement 类型节点的主体, 并将信息传递到 MappedStatement 对象。

MappedStatement 类关联 ParameterMap 类、ResultMap 类、Sql 接口。当 MappedStatement 类关联 ParameterMap 类, 该 ParameterMap 类实例化的 ParameterMap 对象是 MappedStatement 实例化对象的参数, 并实现 MappedStatement 对象中输出对象的处理。当 MappedStatement 类关联 ResultMap 类时, 该 ResultMap 类实例化的 ResultMap 对象是 MappedStatement 实例化对象的输出对象, 并实现 MappedStatement 对象中输出对象的处理。当 MappedStatement 类关联 Sql 接口时, 该 Sql 接口实例化的 Sql 对象是 MappedStatement 实例化对象的 Sql 对象, 并实现 MappedStatement 对象中 Sql 对象的处理。

MappedStatement 类作为多对一被 SqlMapExecutorDelegate 类所聚合。这与 SqlMapExecutorDelegate 类聚合 CacheModel 类、ParameterMap 类、ResultMap 类性质上是一样的。因为 SqlMapExecutorDelegate 是应用的门面。所有的 SQL Map 映射文件节点内容都要到这里集结。

InsertStatement 类、UpdateStatement 类、DeleteStatement 类、SelectStatement 类、ProcedureStatement 类, 继承 MappedStatement 类, 它们都是 MappedStatement 根据不同的操作类型的变体类。

CachingStatement 类, 继承 MappedStatement 类并关联 CacheModel 类, 这是 MappedStatement 类, 中缓存的实现。

2. 对 statement 节点的解析和转化

解析 SQL Map 的 statement 节点, 其 statement 节点的结果内容如下所示。

```
<?xml version="1.0" encoding="UTF-8" ?>
<sqlMap namespace="Account">
  <statement id="updateProfile" cacheModel="name Of Cache"
    parameterMap="name Of ParameterMap" parameterClass="some.class.Name"
    resultMap="name Of ResultMap" resultClass="some.class.Name">
    UPDATE PROFILE SET LANGPREF = #languagePreference#, FAVCATEGORY
    = #favouriteCategoryId#, MYLISTOPT = #listOption#, BANNEROPT =
    #bannerOption# WHERE USERID = #username#
  </statement>
</sqlMap>
```

由上面可以看到, 在 statement 节点中本身有 id、cacheModel、parameterMap、parameterClass、resultMap 和 resultClass 等属性。statement 节点下面没有其他节点, 只有

一个文本内容，即要处理的 SQL 语句。对 resultMap 节点的解析和转化如图 6-39 所示。

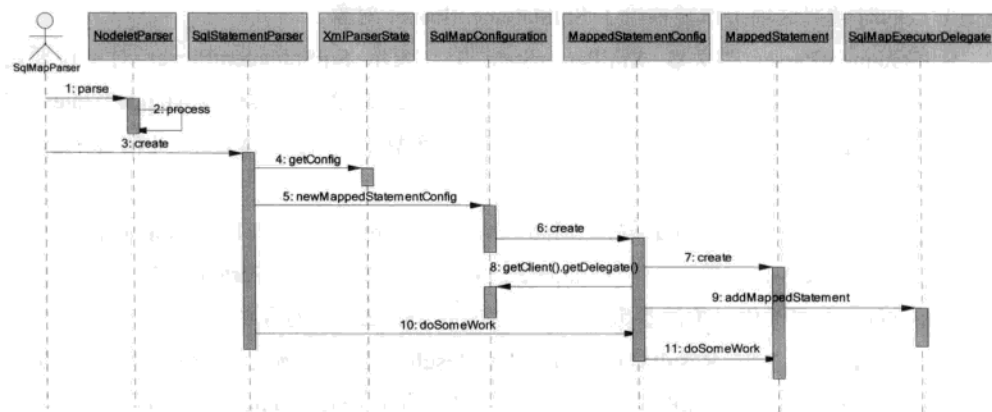


图 6-39 statement 节点解析的整体序列图

对 SQL Map XML 映射文件中 statement 节点的处理结果是：实例化一个 MappedStatement 对象，并最终保存在 SqlMapExecutorDelegate 实例化对象属性 mappedStatements 中。mappedStatements 是一个 HashMap 类型的变量，所以保存的格式按照主键是 statement 节点的 id 内容，而保存的对象就是用这个节点信息实例化的 MappedStatement 对象。实现步骤如下。

(1) SqlMapParser 对象调用 NodeletParser 对象，对 SQL Map XML 映射文件中的节点进行解析和处理。

(2) NodeletParser 对象调用 process 方法，对 SQL Map XML 映射文件中的节点本身、属性和子节点进行解析和处理。

(3) 当解析到 statement 节点的时候，SqlMapParser 对象创建一个新的实例化对象 SqlStatementParser，负责对 statement 节点及其相同的节点（如 insert 节点、update 节点、delete 节点、select 节点和 procedure 节点等）进行解析。

(4) SqlStatementParser 对象调用 XmlParserState 对象的 getConfig 方法，获得 SqlMapConfiguration 对象。

(5) 调用 SqlMapConfiguration 对象的新MappedStatementConfig 方法。

(6) 再调用 SqlMapConfiguration 对象的新MappedStatementConfig 方法，创建 MappedStatementConfig 对象。

(7) MappedStatementConfig 对象在实例化过程中，创建 MappedStatement 对象。

(8) 同时 MappedStatement 对象在实例化过程中，通过 getClient().getDelegate()方法获得 SqlMapConfiguration 对象中的 SqlMapExecutorDelegate 对象。

(9) MappedStatement 对象调用 SqlMapExecutorDelegate 对象的 addMappedStatement

方法,把 MappedStatement 实例化对象加入到 SqlMapExecutorDelegate 对象的 HashMap 变量中。

(10) SqlStatementParser 对象对 MappedStatementConfig 对象进行各种操作,主要内容是把节点的信息转化为 MappedStatementConfig 对象的信息或者对 MappedStatementConfig 对象进行操作处理。

(11) MappedStatementConfig 对象对 MappedStatement 对象进行各种操作,其数据来源于节点的配置信息,并用这些信息对 MappedStatement 对象进行处理。

上述的总体步骤中对于其中的第 10 步和第 11 步是非常复杂的。这里要专门列出来进行说明。解析 statement 节点时,首先是从 SqlMapParser 类的 addStatementNodelets 方法开始的。用 addStatementNodelets 方法实现的节点解析如表 6-19 所示。其实现如图 6-40 所示。这是一个活动图,首先解析和处理解析 statement 节点的属性,接着解析 insert、update、delete、select 和 procedure 等节点的属性。在解析 statement 节点的时候,创建出 SqlStatementParser 对象来进行解析,SqlStatementParser 通过 XmlParserState 对象和 SqlMapConfiguration 对象来新建一个 MappedStatementConfig 对象,MappedStatementConfig 对象在实例化过程中,创建 MappedStatement 对象。把所有生成的 MappedStatement 对象添加到 SqlMapExecutorDelegate 对象中。

表 6-19 addStatementNodelets 方法实现节点解析方法和功能的列表

序号	方法名称	实现功能
1	parser.addNodelet("/sqlMap/statement", new Nodelet())	解析和处理 statement 节点属性
2	parser.addNodelet("/sqlMap/insert", new Nodelet())	解析和处理 insert 节点属性
3	parser.addNodelet("/sqlMap/update", new Nodelet())	解析和处理 update 节点属性
4	parser.addNodelet("/sqlMap/delete", new Nodelet())	解析和处理 delete 节点属性
5	parser.addNodelet("/sqlMap/select", new Nodelet())	解析和处理 select 节点属性
6	parser.addNodelet("/sqlMap/procedure", new Nodelet())	解析和处理 procedure 节点属性

statement 类型节点的解析和转化的整个处理过程的代码主要包括三个方面的步骤和内容:① SqlMapParser 对象对 statement 节点的解析和转化;② SqlStatementParser 对象对 statement 节点的解析和转化;③ MappedStatementConfig 对象对 MappedStatement 对象的处理。

① SqlMapParser 对象对 statement 节点的解析和转化调用 SqlMapParser 对象 addStatementNodelets 方法的 parser.addNodelet("/sqlMap/statement", new Nodelet())方法,实际上是最后调用是 Nodelet 匿名类的 process 方法,代码如下:

```
public void process(Node node) throws Exception {
    statementParser.parseGeneralStatement(node, new MappedStatement());
}
```

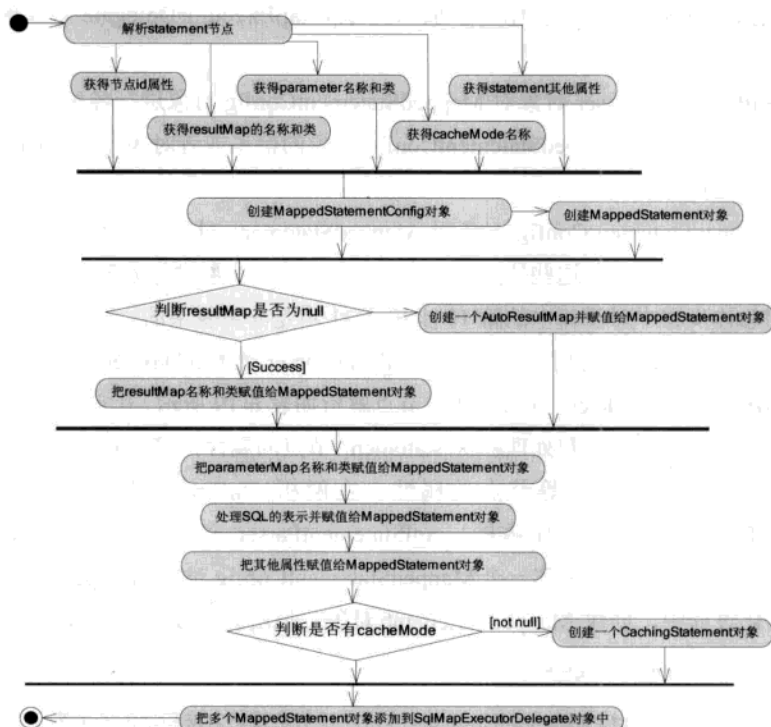


图 6-40 statement 节点解析的实现图

② SqlstatementParser 对象对 statement 节点的解析和转化

转到了 SqlStatementParser 对象的 parseGeneralStatement 方法，我们来看看这个方法的内容，代码如下：

```

public void parseGeneralStatement(Node node, MappedStatement statement) {

    // get attributes
    Properties attributes = NodeletUtils.parseAttributes(node, state.getGlobalProps());
    String id = attributes.getProperty("id");
    String parameterMapName = state.applyNamespace(attributes.getProperty(
        "parameterMap"));
    String parameterClassName = attributes.getProperty("parameterClass");
    String resultMapName = attributes.getProperty("resultMap");
    String resultClassName = attributes.getProperty("resultClass");
    String cacheModelName = state.applyNamespace(attributes.getProperty("cache
        Model"));
    String xmlResultName = attributes.getProperty("xmlResultName");
    String resultSetType = attributes.getProperty("resultSetType");
    String fetchSize = attributes.getProperty("fetchSize");
    String allowRemapping = attributes.getProperty("remapResults");
    String timeout = attributes.getProperty("timeout");

```

```

    if (state.isUseStatementNamespaces()) {
        id = state.applyNamespace(id);
    }
    String[] additionalResultMapNames = null;
    if (resultMapName != null) {
        additionalResultMapNames = state.getAllButFirstToken(resultMapName);
        resultMapName = state.getFirstToken(resultMapName);
        resultMapName = state.applyNamespace(resultMapName);
        for (int i = 0; i < additionalResultMapNames.length; i++) {
            additionalResultMapNames[i] = state.applyNamespace(additionalResultMapNames[i]);
        }
    }

    String[] additionalResultClassNames = null;
    if (resultClassName != null) {
        additionalResultClassNames = state.getAllButFirstToken(resultClassName);
        resultClassName = state.getFirstToken(resultClassName);
    }
    Class[] additionalResultClasses = null;
    if (additionalResultClassNames != null) {
        additionalResultClasses = new Class[additionalResultClassNames.length];
        for (int i = 0; i < additionalResultClassNames.length; i++) {
            additionalResultClasses[i] = resolveClass(additionalResultClassNames[i]);
        }
    }

    state.getConfig().getErrorContext().setMoreInfo("Check the parameter class.");
    Class parameterClass = resolveClass(parameterClassName);

    state.getConfig().getErrorContext().setMoreInfo("Check the result class.");
    Class resultClass = resolveClass(resultClassName);

    Integer timeoutInt = timeout == null ? null : new Integer(timeout);
    Integer fetchSizeInt = fetchSize == null ? null : new Integer(fetchSize);
    boolean allowRemappingBool = "true".equals(allowRemapping);

    MappedStatementConfig statementConf = state.getConfig().newMappedStatementConfig(id, statement,
        new XmlSqlSource(state, node), parameterMapName, parameterClass, resultMapName, additionalResultMapNames,
        resultClass, additionalResultClasses, resultSetType, fetchSizeInt, allowRemappingBool, timeoutInt, cacheModelName,
        xmlResultName);

    findAndParseSelectKey(node, statementConf);
}

```

该方法首先获取 statement 节点的相关信息，然后进行一系列的处理和转化。接着通过

XmlParserState 对象获得 SqlMapConfiguration 对象,最后通过调用 SqlMapConfiguration 对象的 newMappedStatementConfig 方法来创建一个 MappedStatementConfig 对象,并且把大量的配置信息作为参数传入。代码如下:

```
public MappedStatementConfig newMappedStatementConfig(String id, MappedStatement
statement,
    SqlSource processor, String parameterMapName,
    Class parameterClass, String resultMapName, String[] additionalResultMap
Names,
    Class resultClass, Class[] additionalResultClasses, String resultSetType,
Integer fetchSize,
    boolean allowRemapping, Integer timeout, String cacheModelName, String xmlResult
Name) {
    return new MappedStatementConfig(this, id, statement, processor,
parameterMapName, parameterClass, resultMapName, additionalResultMapNames, resultClass,
additionalResultClasses, cacheModelName, resultSetType, fetchSize,
    allowRemapping, timeout, defaultStatementTimeout, xmlResultName);
}
```

③ MappedStatementConfig 对象对 MappedStatement 对象的处理

SqlMapConfiguration 对象的 newMappedStatementConfig 方法实际上就是实例化 MappedStatementConfig 对象,接受了传入的参数。让我们来看看 MappedStatementConfig 对象的实例化过程,代码如下。

```
MappedStatementConfig(SqlMapConfiguration config, String id, MappedStatement
statement, SqlSource processor,
    String parameterMapName, Class parameterClass, String result
resultName,
    String[] additionalResultMapNames, Class resultClass, Class[]
additionalResultClasses,
    String cacheModelName, String resultSetType, Integer fetch
Size, boolean allowRemapping,
    Integer timeout, Integer defaultStatementTimeout, String xml
ResultName) {
    this.errorContext = config.getErrorContext();
    this.client = config.getClient();
    SqlMapExecutorDelegate delegate = client.getDelegate();
    this.typeHandlerFactory = config.getTypeHandlerFactory();
    errorContext.setActivity("parsing a mapped statement");
    errorContext.setObjectId(id + " statement");
    errorContext.setMoreInfo("Check the result map name.");
    if (resultMapName != null) {
        statement.setResultMap(client.getDelegate().getResultMap(resultMapName));
        if (additionalResultMapNames != null) {
            for (int i = 0; i < additionalResultMapNames.length; i++) {
                statement.addResultMap(client.getDelegate().getResultMap(additionalResultMapNames[i]));
            }
        }
    }
}
```

```

    }
}
errorContext.setMoreInfo("Check the parameter map name.");
if (parameterMapName != null) {
    statement.setParameterMap(client.getDelegate().getParameterMap
(parameterMapName));
}
statement.setId(id);
statement.setResource(errorContext.getResource());
if (resultSetType != null) {
    if ("FORWARD_ONLY".equals(resultSetType)) {
        statement.setResultSetType(new Integer(ResultSet.TYPE_FORWARD_ONLY));
    } else if ("SCROLL_INSENSITIVE".equals(resultSetType)) {
        statement.setResultSetType(new Integer(ResultSet.TYPE_SCROLL_INSENSITIVE));
    } else if ("SCROLL_SENSITIVE".equals(resultSetType)) {
        statement.setResultSetType(new Integer(ResultSet.TYPE_SCROLL_SENSITIVE));
    }
}
if (fetchSize != null) {
    statement.setFetchSize(fetchSize);
}

// set parameter class either from attribute or from map (make sure to match)
ParameterMap parameterMap = statement.getParameterMap();
if (parameterMap == null) {
    statement.setParameterClass(parameterClass);
} else {
    statement.setParameterClass(parameterMap.getParameterClass());
}

// process SQL statement, including inline parameter maps
errorContext.setMoreInfo("Check the SQL statement.");
Sql sql = processor.getSql();
setSqlForStatement(statement, sql);

// set up either null result map or automatic result mapping
ResultMap resultMap = (ResultMap) statement.getResultMap();
if (resultMap == null && resultClass == null) {
    statement.setResultMap(null);
} else if (resultMap == null) {
    resultMap = buildAutoResultMap(allowRemapping, statement, resultClass, xml
ResultName);
    statement.setResultMap(resultMap);
    if (additionalResultClasses != null) {
        for (int i = 0; i < additionalResultClasses.length; i++) {
            statement.addResultMap(buildAutoResultMap(allowRemapping, statement,
additionalResultClasses[i], xmlResultName));
        }
    }
}
}

```

```

statement.setTimeout(defaultStatementTimeout);
if (timeout != null) {
    try {
        statement.setTimeout(timeout);
    } catch (NumberFormatException e) {
        throw new SqlMapException("Specified timeout value for statement " +
statement.getId() + " is not a valid integer");
    }
}
errorContext.setMoreInfo(null);
errorContext.setObjectId(null);
statement.setSqlMapClient(client);
if (cacheModelName != null && cacheModelName.length() > 0 && client.
getDelegate().isCacheModelsEnabled()) {
    CacheModel cacheModel = client.getDelegate().getCacheModel(cacheModelName);
    mappedStatement = new CachingStatement(statement, cacheModel);
} else {
    mappedStatement = statement;
}
rootStatement = statement;
delegate.addMappedStatement(mappedStatement);
}

```

MappedStatementConfig 对象的构造方法才是整个配置信息处理的核心。由于在 **SqlMapParser** 对象已经创建并实例化了一个 **MappedStatement** 对象，所以后面的工作就是针对这个 **MappedStatement** 对象进行赋值，要完成了下列几个工作。

第一是要对输入参数进行处理，即获取 **ParameterMap** 对象并转入到 **MappedStatement** 对象中。

第二是对配置的 SQL 语句进行转化，即处理 **Sql** 对象。

第三是对输出对象进行处理，即获取 **ResultMap** 对象并转入到 **MappedStatement** 对象中。

第四是对缓存对象进行处理，即如果 **CacheModel** 对象存在，则把 **MappedStatement** 对象转化为 **CachingStatement** 对象。

第五是把 **MappedStatement** 对象添加到 **SqlMapExecutorDelegate** 对象的 **HashMap** 类型的属性 **mappedStatements** 中。

3. 形成的节点和对象映射表

实例化一个 **MappedStatement** 对象，并最终保存在 **SqlMapExecutorDelegate** 实例化对象 **HashMap** 类型的属性 **mappedStatements** 中。**mappedStatements** 是一个 **HashMap** 类型的变量，所以保存的格式按照主键是 **statement** 节点的 **id** 内容，而保存的对象就是用这个节点信息实例化的 **MappedStatement** 对象。**statement** 节点属性和 **MappedStatement** 对象属性的映射关系如表 6-20 所示。

表 6-20 statement 节点属性和 MappedStatement 对象属性的映射关系

序号	statement 节点属性	MappedStatement 对象属性	类 型
1	id	id	String
2	parameterClass	parameterClass	Class
3	resultClass		Class
4	parameterMap	parameterMap	ParameterMap
5	resultMap	resultMap	ResultMap
6	cacheModel		
7	Sqltext	sql	Sql

4. 对 insert、update、delete、select 和 procedure 节点的解析和转化

(1) 对 insert 节点的解析和转化

调用 addStatementNodelets 方法的 parser.addNodelet("/sqlMap/insert", new Nodelet()) 方法, 实际上最后调用是 Nodelet 匿名类的 process 方法, 代码如下。

```
public void process(Node node) throws Exception {
    statementParser.parseGeneralStatement(node, new InsertStatement());
}
```

实现模式与调用 statement 是一样的。只是实例化是一个 InsertStatement 对象, 最终保存在 SqlMapExecutorDelegate 实例化对象属性 mappedStatements 中。mappedStatements 是一个 HashMap 类型的变量, 所以保存的格式按照主键是 insert 节点的 id 内容, 而保存的对象就是用这个节点信息实例化的 InsertStatement 对象。针对的配置文件如下所示。

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

<sqlMap namespace="Account">
    <insert id="insertAccount" cacheModel="name Of Cache"
        parameterMap="name Of ParameterMap" parameterClass="some.class.Name"
        resultMap="name Of ResultMap" resultClass="some.class.Name">
        INSERT INTO ACCOUNT (EMAIL, FIRSTNAME, LASTNAME, STATUS, ADDR1,
        ADDR2, CITY, STATE, ZIP, COUNTRY, PHONE, USERID) VALUES
        (#email#, #firstName#, #lastName#, #status#, #address1#,
        #address2:VARCHAR#, #city#, #state#, #zip#, #country#, #phone#,
        #username#)
    </insert>
</sqlMap>
```

(2) 对 update 节点的解析和转化

调用 addStatementNodelets 方法的 parser.addNodelet("/sqlMap/update", new Nodelet()) 方法, 实际上最后调用是 Nodelet 匿名类的 process 方法, 代码如下。

```
public void process(Node node) throws Exception {
    statementParser.parseGeneralStatement(node, new UpdateStatement());
}
```

实现模式与调用 statement 是一样的。只是实例化是一个 UpdateStatement 对象, 最终

保存在 `SqlMapExecutorDelegate` 实例化对象属性 `mappedStatements` 中。`mappedStatements` 是一个 `HashMap` 类型的变量，所以保存的格式按照主键是 `update` 节点的 `id` 内容，而保存的对象就是用这个节点信息实例化的 `UpdateStatement` 对象。针对的配置文件如下所示。

```
<?xml version="1.0" encoding="UTF-8" ?>
<sqlMap namespace="Account">
  <update id="updateAccount" cacheModel="name Of Cache"
    parameterMap="name Of ParameterMap" parameterClass="some.class.Name"
    resultMap="name Of ResultMap" resultClass="some.class.Name">
    UPDATE ACCOUNT SET EMAIL = #email#, FIRSTNAME = #firstName#,
    LASTNAME = #lastName#, STATUS = #status#, ADDR1 = #address1#,
    ADDR2 = #address2:VARCHAR#, CITY = #city#, STATE = #state#, ZIP
    = #zip#, COUNTRY = #country#, PHONE = #phone# WHERE USERID =
    #username#
  </update>
</sqlMap>
```

(3) 对 `delete` 节点的解析和转化

调用 `addStatementNodelets` 方法的 `parser.addNodelet("/sqlMap/delete", new Nodelet())` 方法，实际上最后调用是 `Nodelet` 匿名类的 `process` 方法，代码如下。

```
public void process(Node node) throws Exception {
    statementParser.parseGeneralStatement(node, new DeleteStatement());
}
```

实现模式与调用 `statement` 是一样的。只是实例化是一个 `DeleteStatement` 对象，最终保存在 `SqlMapExecutorDelegate` 实例化对象属性 `mappedStatements` 中。`mappedStatements` 是一个 `HashMap` 类型的变量，所以保存的格式按照主键是 `delete` 节点的 `id` 内容，而保存的对象就是用这个节点信息实例化的 `DeleteStatement` 对象。针对的配置文件如下所示。

```
<?xml version="1.0" encoding="UTF-8" ?>
<sqlMap namespace="Account">
  <delete id="deleteAccount" cacheModel="name Of Cache"
    parameterMap="name Of ParameterMap" parameterClass="some.class.Name"
    resultMap="name Of ResultMap" resultClass="some.class.Name">
    DELETE INTO ACCOUNT WHERE ACCOUNT.USERID=#uiId#
  </delete>
</sqlMap>
```

(4) 对 `select` 节点的解析和转化

调用 `addStatementNodelets` 方法的 `parser.addNodelet("/sqlMap/select", new Nodelet())` 方法，实际上最后调用是 `Nodelet` 匿名类的 `process` 方法，代码如下。

```
public void process(Node node) throws Exception {
    statementParser.parseGeneralStatement(node, new SelectStatement());
}
```

实现模式与调用 `statement` 是一样的。只是实例化是一个 `SelectStatement` 对象，最终保

存在 `SqlMapExecutorDelegate` 实例化对象属性 `mappedStatements` 中。`mappedStatements` 是一个 `HashMap` 类型的变量，所以保存的格式按照主键是 `select` 节点的 `id` 内容，而保存的对象就是用这个节点信息实例化的 `SelectStatement` 对象。针对的配置文件如下所示。

```
<?xml version="1.0" encoding="UTF-8" ?>
<sqlMap namespace="Account">
  <select id="getAccountByUsername" cacheModel="name Of Cache"
    parameterMap="name Of ParameterMap" parameterClass="some.class.Name"
    resultMap="name Of ResultMap" resultClass="some.class.Name">
    SELECT SIGNON.USERNAME, ACCOUNT.EMAIL, ACCOUNT.FIRSTNAME,
    ACCOUNT.LASTNAME, ACCOUNT.STATUS, ACCOUNT.ADDR1 AS address1,
    ACCOUNT.ADDR2 AS address2, ACCOUNT.CITY, ACCOUNT.STATE,
    ACCOUNT.ZIP, ACCOUNT.COUNTRY, ACCOUNT.PHONE, PROFILE.LANGPREF AS
    languagePreference, PROFILE.FAVCATEGORY AS favouriteCategoryId,
    PROFILE.MYLISTOPT AS listOption, PROFILE.BANNEROPT AS
    bannerOption, BANNERDATA.BANNERNAME FROM ACCOUNT, PROFILE,
    SIGNON, BANNERDATA WHERE ACCOUNT.USERID = #username# AND
    SIGNON.USERNAME = ACCOUNT.USERID AND PROFILE.USERID =
    ACCOUNT.USERID AND PROFILE.FAVCATEGORY = BANNERDATA.FAVCATEGORY
  </select>
</sqlMap>
```

(5) 对 `procedure` 节点的解析和转化

调用 `addStatementNodelets` 方法的 `parser.addNodelet("/sqlMap/procedure", new Nodelet())` 方法，实际上最后调用是 `Nodelet` 匿名类的 `process` 方法，代码如下。

```
public void process(Node node) throws Exception {
    statementParser.parseGeneralStatement(node, new ProcedureStatement());
}
```

实现模式与调用 `statement` 是一样的。只是实例化是一个 `ProcedureStatement` 对象，最终保存在 `SqlMapExecutorDelegate` 实例化对象属性 `mappedStatements` 中。`mappedStatements` 是一个 `HashMap` 类型的变量，所以保存的格式按照主键是 `procedure` 节点的 `id` 内容，而保存的对象就是用这个节点信息实例化的 `ProcedureStatement` 对象。

```
<?xml version="1.0" encoding="UTF-8" ?>
<sqlMap namespace="Account">
  <procedure id="updateAccountProcedure" cacheModel="name Of Cache"
    parameterMap="name Of ParameterMap" parameterClass="some.class.Name"
    resultMap="name Of ResultMap" resultClass="some.class.Name">
    {call out_int_account(?)}
  </procedure>
</sqlMap>
```

6.3.8 对 SQL 的处理

在对 `statement` 类型节点进行解析时，必须要对每个节点内的 `sqltext` 做解析。例如下

面列表中的黑体部分。

```
<?xml version="1.0" encoding="UTF-8" ?>
<sqlMap namespace="Account">
  <update id="updateAccount" cacheModel="name Of Cache"
    parameterMap="name Of ParameterMap" parameterClass="some.class.Name"
    resultMap="name Of ResultMap" resultClass="some.class.Name">
    UPDATE ACCOUNT SET EMAIL = #email#, FIRSTNAME = #firstName#,
    LASTNAME = #lastName#, STATUS = #status#, ADDR1 = #address1#,
    ADDR2 = #address2:VARCHAR#, CITY = #city#, STATE = #state#, ZIP
    = #zip#, COUNTRY = #country#, PHONE = #phone# WHERE USERID =
    #username#
  </update>
</sqlMap>
```

这些全部都是文本内容，解析只是针对文本进行处理，然后对输入的参数进行转化即可。这个内容其实很简单，但是 iBATIS 的功能不仅仅是这样，它还支持动态的 SQL 处理。如下面的例子。

```
<?xml version="1.0" encoding="UTF-8" ?>
<sqlMap namespace="Account">
  <select id="dynamicGetAccountList" cacheModel="account-cache" resultMap=
    "account-result">
    select * from ACCOUNT
    <isGreaterThan prepend="and" property="id" compareValue="0">
      where ACC_ID = #id#
    </isGreaterThan>
    order by ACC_LAST_NAME
  </select>
</sqlMap>
```

上面的例子中，根据参数 bean “id” 属性的不同情况，可创建两个可能的语句。如果参数 “id” 大于 0，将创建下面的语句：

```
select * from ACCOUNT where ACC_ID = ?
```

或者，如果 “id” 参数小于等于 0，将创建下面的语句：

```
select * from ACCOUNT
```

在更复杂的例子中，动态 MappedStatement 的用处更明显。如下面比较复杂的例子：

```
<?xml version="1.0" encoding="UTF-8" ?>
<sqlMap namespace="Account">
  <statement id="someName" resultMap="account-result">
    select * from ACCOUNT
    <dynamic prepend="where">
      <isGreaterThan prepend="and" property="id"
        compareValue="0">
        ACC_ID = #id#
```



```

        </isGreaterThan>
        <isNotNull prepend="and " property="lastName">
            ACC_LAST_NAME = #lastName#
        </isNotNull>
    </dynamic>
    order by ACC_LAST_NAME
</statement></sqlMap>

```

上面的例子中，<dynamic>元素划分出 SQL 语句的动态部分。动态部分可以包含任意多的条件标签元素，条件标签决定是否在语句中包含其中的 SQL 代码。所有的条件标签元素将根据传给动态查询 Statement 的参数对象的情况来工作。<dynamic>元素和条件元素都有“prepend”属性，它是动态 SQL 代码的一部分，在必要情况下，可以被父元素的“prepend”属性覆盖。上面的例子中，prepend 属性“where”将覆盖第一个为“真”的条件元素。这对于确保生成正确的 SQL 语句是有必要的。例如，在第一个为“真”的条件元素中，“AND”是不需要的，事实上，加上它肯定会出错。

这要涉及 iBATIS 的一元条件元素和二元条件元素。这里面对文本的处理和语法分析往往会涉及 SQL Map 的处理能力，而且也会牵涉到多个对象。针对 sqltext 解析和处理，iBATIS 有专门的实现。

由于 SQL 有三种类型，分别为动态 SQL 语句，静态 SQL 语句和简单动态 SQL 语句。

1. 解析 SQL 文本的类结构图

解析 SQL 文本要涉及如下这些类：SqlMapConfigParser 类、SqlMapParser 类、NodeletParser 类、Nodelet 接口、SqlStatementParser 类、MappedStatementConfig 类、MappedStatement 类、SqlSource 接口、XmlSqlSource 类、Sql 接口、RawSql 类、DynamicSql 类、SimpleDynamicSql 类、StaticSql 类和 SqlMapExecutorDelegate 类，其类结构如图 6-41 所示。

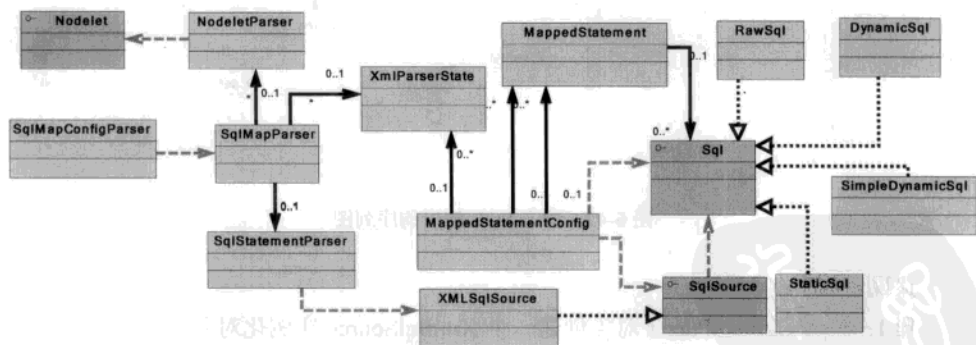


图 6-41 SQL 文本解析的类结构图

左端的内容与前面基本上无异。SqlStatementParser 类依赖 XmlSqlSource 类，MappedStatementConfig 类依赖 SqlSource 接口（实际上就是 XmlSqlSource 类），同时也依赖 Sql 接口，MappedStatement 关联 Sql 接口，而 Sql 接口由 RawSql 类、DynamicSql 类、

SimpleDynamicSql 类和 StaticSql 类来实现。

2. 静态 SQL 语句的解析

静态 SQL 语句的解析比较简单，由于静态 SQL 的内容比较少，只是把 SQL 语句和参数分离出来。其解析的例子如下。

```
<?xml version="1.0" encoding="UTF-8" ?>
<sqlMap namespace="Account">
  <statement id="updateProfile" cacheModel="name Of Cache"
    parameterMap="name Of ParameterMap" parameterClass="some.class.Name"
    resultMap="name Of ResultMap" resultClass="some.class.Name">
    UPDATE PROFILE SET LANGPREF = #languagePreference#, FAVCATEGORY
    = #favouriteCategoryId#, MYLISTOPT = #listOption#, BANNEROPT =
    #bannerOption# WHERE USERID = #username#
  </statement>
</sqlMap>
```

静态 SQL 语句包括两个部分，一个部分是 SQL 的文本内容，还有一个部分就是参数，如用#包括的内容，其实现的序列如图 6-42 所示。

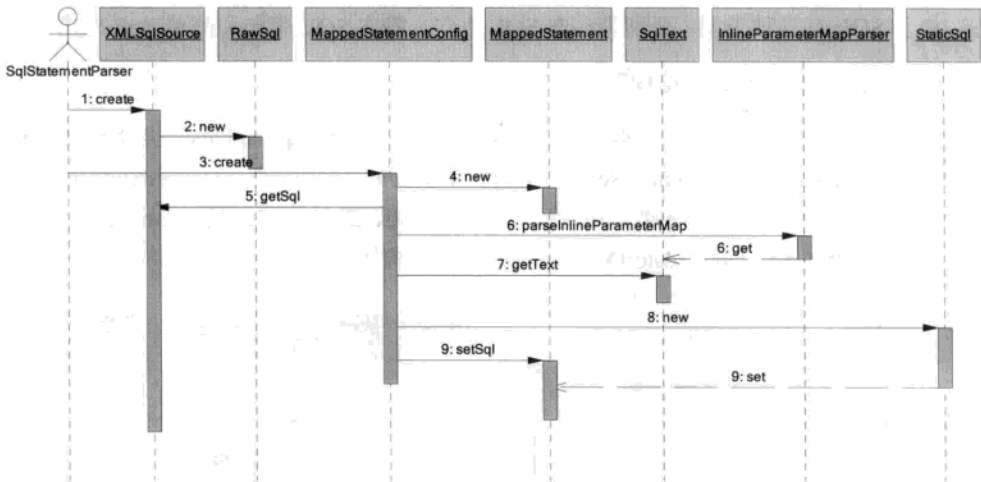


图 6-42 静态 SQL 解析的序列图

步骤说明如下：

- 步骤 1: SqlStatementParser 对象创建一个 XMLSqlSource 实例化对象。
- 步骤 2: XMLSqlSource 对象创建一个 RawSql 对象。
- 步骤 3: SqlStatementParser 对象创建一个 MappedStatementConfig 实例化对象。
- 步骤 4: MappedStatementConfig 实例化对象内部创建一个 MappedStatement 实例化对象。
- 步骤 5: MappedStatementConfig 调用 XMLSqlSource 实例化对象的 getSql 方法，获得实现接口 Sql 的对象（实际上是一个 RawSql 对象）。

步骤 6: XmlSqlSource 对象调用 InlineParameterMapParser 对象的 parseInline Parameter Map 方法, 获得解析后的 SqlText 对象。

步骤 7: XmlSqlSource 对象调用 SqlText 对象 getText 方法, 获得 String 变量的 sql 语句。

步骤 8: XmlSqlSource 对象创建一个 StaticSql 对象, 并把 String 变量的 sql 语句初始化给 StaticSql 对象。

步骤 9: XmlSqlSource 对象调用 MappedStatement 实例化对象的 setSql 方法, 把 StaticSql 对象作为变量赋值给 MappedStatement 对象。

3. 简单动态 SQL 语句的解析

其解析的例子如下。

```
<?xml version="1.0" encoding="UTF-8" ?>
<sqlMap namespace="Account">
  <statement id="getProduct" resultMap="get-product-result">
    SELECT * FROM PRODUCT
    <dynamic prepend="WHERE">
      <isNotEmpty property="description">
        PRD_DESCRIPTION $operator$ #description#
      </isNotEmpty>
    </dynamic>
  </statement>
</sqlMap>
```

简单动态 sql 语句解析的序列如图 6-43 所示。

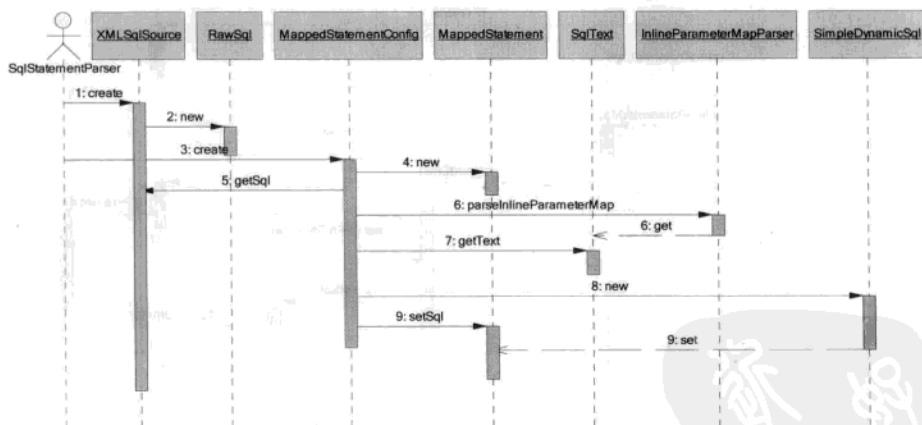


图 6-43 简单动态 SQL 语句解析的序列图

简单动态 SQL 语句的解析与静态 SQL 语句的解析基本一致, 就是在最后的时候创建的是一个 SimpleDynamicSql 对象。前面 7 个步骤都与静态 SQL 语句的解析一样, 其后为: 步骤 8: XmlSqlSource 对象创建一个 SimpleDynamicSql 对象, 并把 String 变量的 SQL 语

句初始化给 SimpleDynamicSql 对象。

步骤 9: XmlSqlSource 对象调用 MappedStatement 实例化对象的 setSql 方法，把 SimpleDynamicSql 对象作为变量赋值给 MappedStatement 对象。

4. 动态 SQL 语句的解析实现

图 6-44 列出动态 SQL 语句所要使用的核心对象并对业务进行说明。

动态 SQL 语句解析的步骤说明如下：

步骤 1: SqlStatementParser 对象创建一个 XmlSqlSource 实例化对象，参数如下。

```
new XmlSqlSource(state, node)
```

步骤 2: XmlSqlSource 内部实例化一个静态 InlineParameterMapParser 对象，变量名称是 PARAM_PARSER，代码如下。

```
private static final InlineParameterMapParser PARAM_PARSER = new InlineParameterMapParser();
```

步骤 3: SqlStatementParser 对象创建一个 MappedStatementConfig 实例化对象。

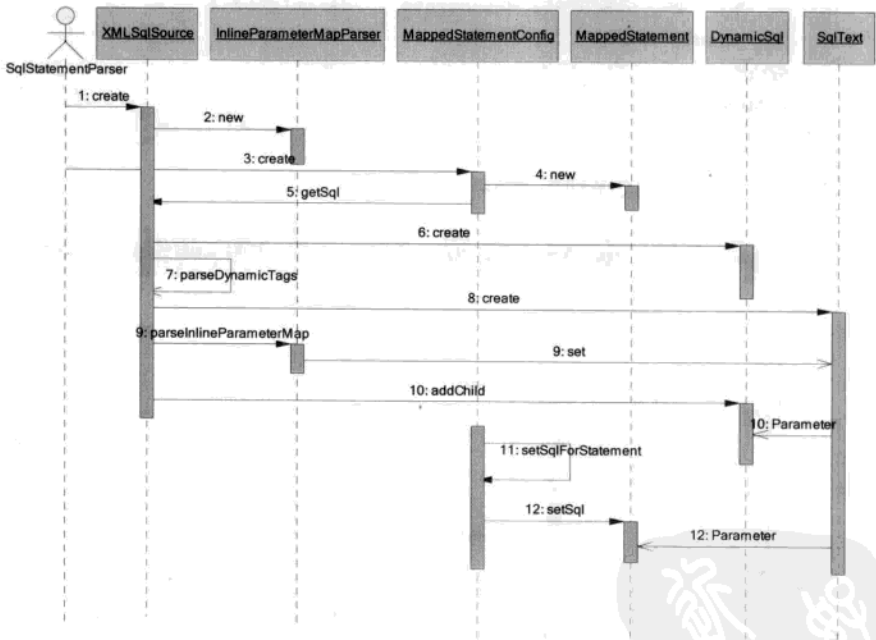


图 6-44 动态 SQL 语句解析的序列图

步骤 4: MappedStatementConfig 实例化对象内部创建一个 MappedStatement 实例化对象。

步骤 5: MappedStatementConfig 调用 XmlSqlSource 实例化对象的 getSql 方法，获得实现接口 Sql 的对象（实际上是一个 DynamicSql 对象）。

步骤 6: XmlSqlSource 对象在运行 getSql 方法时, 创建一个 DynamicSql 对象。

步骤 7: XmlSqlSource 对象调用自身的 parseDynamicTags 方法。

步骤 8: XmlSqlSource 对象调用自身的 parseDynamicTags 方法, 创建一个 SqlText 对象。

步骤 9: XmlSqlSource 对象调用 InlineParameterMapParser 对象的 parseInlineParameterMap 方法, 获得解析后的 SqlText 对象。

步骤 10: XmlSqlSource 对象调用 DynamicSql 对象的 addChild 方法, 把 SqlText 对象加入到 DynamicSql 对象中。

步骤 11: MappedStatementConfig 对象调用自身的 setSqlForStatement 方法。

步骤 12: MappedStatementConfig 对象在调用自身的 setSqlForStatement 方法时, 调用 MappedStatement 对象的 setSql 方法, 把 DynamicSql 对象加载到 MappedStatement 对象中。

6.4 抽象出通用的 XML 解析框架

在 iBATIS 中配置文件和映射文件都是采用同一个 XML 解析框架来处理的。根据对配置文件的处理, 我们可以抽象出一个通用的 XML 解析框架。

为了便于说明, 还可用一个实践例子来说明。由于不失一般通用性, 可以随便选择了一个 XML 文件来进行解析。于是就采用本项目的 ant build 生成的 build.xml 文件来进行解析说明。

build.xml 样例的 XML 文件格式如下。

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- WARNING: Eclipse auto-generated file.
      Any modifications will be overwritten.
      To include a user specific buildfile here, simply create one in the same
      directory with the processing instruction <?eclipse.ant.import?>
      as the first entry and export the buildfile again. -->
<project basedir="." default="build" name="iBATIS-XmlParser">
  <property environment="env"/>
  <property name="ECLIPSE_HOME" value="../../Eclipse3.2"/>
  <property name="debuglevel" value="source,lines,vars"/>
  <property name="target" value="1.5"/>
  <property name="source" value="1.5"/>
  <path id="iBATIS-XmlParser.classpath">
    <pathelement location="bin"/>
  </path>
  <target name="init">
    <mkdir dir="bin"/>
    <copy includeemptydirs="false" todir="bin">
      <fileset dir="src" excludes="**/*.launch, **/*.java"/>
    </copy>
  </target>
  <target name="clean">
    <delete dir="bin"/>
  </target>
```

```

<target depends="clean" name="cleanall"/>
<target depends="build-subprojects,build-project" name="build"/>
<target name="build-subprojects"/>
<target depends="init" name="build-project">
  <echo message="${ant.project.name}: ${ant.file}"/>
  <javac debug="true" debuglevel="${debuglevel}" destdir="bin" source="${
{source}" target="${target}"/>
    <src path="src"/>
    <classpath refid="ibatis-XmlParser.classpath"/>
  </javac>
</target>
<target description="Build all projects which reference this project. Useful to
propagate changes." name="build-refprojects"/>
<target description="copy Eclipse compiler jars to ant lib directory" name=
"init-eclipse-compiler">
  <copy todir="${ant.library.dir}">
    <fileset dir="${ECLIPSE_HOME}/plugins" includes="org.eclipse.jdt.core_
*.jar"/>
  </copy>
  <unzip dest="${ant.library.dir}">
    <patternset includes="jdtCompilerAdapter.jar"/>
    <fileset dir="${ECLIPSE_HOME}/plugins" includes="org.eclipse.jdt.core_
*.jar"/>
  </unzip>
</target>
<target description="compile project with Eclipse compiler" name="build-
eclipse-compiler">
  <property name="build.compiler" value="org.eclipse.jdt.core.JDTCompiler
Adapter"/>
  <antcall target="build"/>
</target>
</project>

```

该文件的 XSD 格式如图 6-45 所示。

在该 XSD 中，我们知道 build.xml 的根节点是 project，包括 basedir、default、name 等属性，第一级子节点是 property、path、target 节点。其中 property 节点有 environment、name 和 value 属性。path 节点有 id 属性。target 节点有 name、depends 和 description 属性。path 节点下只有 pathelement 节点。而 target 节点比较复杂，包括了多级节点，其中下一级节点就有 property、antcall、echo、javac、delete、mkdir、copy 和 unzip。

在本项目实现的内容就是把各个节点都实例化为对象，节点的属性转化为对象中以 String 类型的属性。如果节点下有节点，那在父节点形成的对象中有一个 Map 变量，包括所有同类型子节点实例化的对象。例如，本样例 XML 文件的根节点是 project，所以实例化一个 Project 对象。根节点有 basedir、default、name 等属性，变成了 Project 对象中的私有 String 变量 basedir、default、name。根节点下面的一级节点有 property、path、target。这些节点都要形成 Property 对象、Path 对象、Target 对象，并在 Project 对象有 Map 类型的 propertyMap、pathMap、targetMap，分别存储这些 Property 对象、Path 对象、Target 对象的集合。

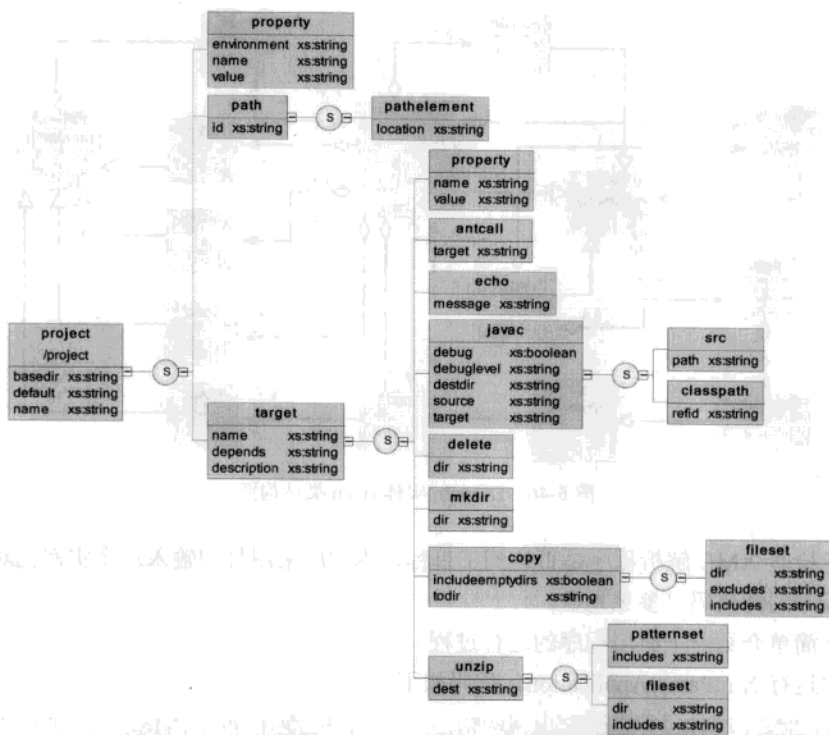


图 6-45 build.xml 的 XSD 结构图

应用程序包括三个部分，第一个是 Xml 解析工具组件，第二个是 XML 解析程序，第三个是 Domain，主要是各个节点实例化的 JavaBean 类。

Xml 解析工具组件可以直接从 iBATIS 的源码中获取，主要是 com.ibatis.common.xml 包的全部内容，采取拿来主义，基本上不做修改。

XML 解析程序是根据具体的业务要求，即要对 XML 文件做什么操作和处理而编写的程序。内容包括两个类——ConfigParser 类和 XmlParserState 类。ConfigParser 类是执行体，即对各个节点要采取什么操作，都是在 ConfigParser 类中的方法来实现的。而 XmlParserState 主要是进行信息转化，主要是针对 XML 文件中的父子节点进行关联。

Domain 中的实体 Bean，按照根节点生成 Project 对象，Project 对象属性对应节点的属性，同时，对于根节点的子节点，就转化为 Project 类的 HashMap 属性。按照这个道理往下推，其他对象也是按照这个规则来进行的。其类结构如图 6-46 所示。

在图 6-46 中为了表现清晰，把 Pojo 类分为三个类，分别为 Pojo:1 类、Pojo:2 类和 Pojo:3 类。这三个类在图中有三个标识符，但实际上都只有一个 Pojo 类。图中除 Pojo 类外所有的实体 Bean 都继承抽象类 Pojo。Project 类聚合 Property 类、Path 类、Target 类。Path 类聚合 Pathelement 类。Target 类聚合 Antcall 类、Mkdir 类、Copy 类、Delete 类、Echo 类、Javac 类和 Unzip 类。


```

        project.setDefaultType(defaultType);
        state.setProject(project);
    }
});

//解析 project/property 节点
parser.addNodelet("/project/property", new Nodelet() {
    public void process(Node node) throws Exception {
        Property property = new Property();
        Properties attributes = NodeletUtils.parseAttributes(node,
            state.getGlobalProps());
        String name = attributes.getProperty("name");
        String value = attributes.getProperty("value");

        property.setName(name);
        property.setValue(value);
        state.getProject().addProperty(name, property);
    }
});
}
}

```

在 `addProjectConfigNodelets` 方法代码中，有多少类型的节点，`NodeletParser` 对象就有多少个 `addNodelet` 方法。其方法的传入参数有两个，第 1 个参数是节点的位置，第 2 个参数是创建一个 `Nodelet` 对象。其 `Nodelet` 对象为匿名类，其实现的方法是 `process`。为不失一般性，我们用两个节点来说明，一个是根节点 `project`，另一个是 `path` 节点。

根节点 `project` 的解析代码如下。

```

public void process(Node node) throws Exception {
    Project project = new Project();
    Properties attributes = NodeletUtils.parseAttributes(node, state.getGlobal
Props());
    String name = attributes.getProperty("name");
    String basedir = attributes.getProperty("basedir");
    String defaultType = attributes.getProperty("default");
    project.setName(name);
    project.setBasedir(basedir);
    project.setDefaultType(defaultType);
    state.setProject(project);
}
}

```

根节点 `project` 的 `process` 方法首先创建一个 `Project` 对象，然后通过 `NodeletUtils` 从 `node` 中获取节点 `project` 的属性。这些节点获得的信息赋值给 `Project` 对象对应的属性。由于 `Project` 对象是一个父对象，所以要把这个对象传递给 `XmlParserState` 实例化对象。供给节点 `project` 下的子节点使用。

我们来看看子 `path` 节点的解析代码。

```

public void process(Node node) throws Exception {
    Path path = new Path();
    Properties attributes = NodeletUtils.parseAttributes(node, state.getGlobalProps());
    String id = attributes.getProperty("id");
    path.setId(id);
    state.setPath(path);
    state.getProject().addPath(id, path);
}

```

节点 `path` 的 `process` 方法首先创建一个 `Path` 对象，然后通过 `NodeletUtils` 从 `node` 中获取节点 `path` 的属性。这些节点获得的信息赋值给 `Path` 对象对应的属性。由于 `path` 节点是 `project` 节点的子节点，`Path` 对象也是 `Project` 对象中 `HashMap` 的一个值，所以要从 `XmlParserState` 实例化对象获得父节点形成的 `Project` 对象，并把自己赋值进去。同时由于 `Path` 对象是一个父对象，所以要把这个对象传递给 `XmlParserState` 实例化对象，供给节点 `path` 下的子节点使用。

经过所有节点的 `addNodelet` 方法，是给每个节点的处理进行了工作的安排，实际上这些操作并没有执行，只是已经知道要做什么工作。

我们再回头来看看 `XmlParserApplication` 对象的 `viewProject` 方法，下面执行第二步骤。

```

public static void viewProject(){
    Project project = configParser.buildProject("build.xml");
}

```

看看 `ConfigParser` 对象的 `buildProject` 方法，代码如下：

```

public Project buildProject(String resource) {
    InputStream inputStream = null;
    try {
        inputStream = getResourceAsStream(resource);
        usingStreams = true;
    } catch (Exception e) {
        System.out.println(e);
    }
    return parse(inputStream);
}

```

在 `buildProject` 方法中，通过 `getResourceAsStream` 可以获得一个 `inputStream` 对象，然后把这个 `inputStream` 对象作为参数调用 `parse` 方法。我们来看看这个方法的代码。

```

private Project parse(InputStream inputStream) {
    try {
        parser.parse(inputStream);
        return state.getProject();
    } catch (Exception e) {
        throw new RuntimeException("Error occurred. Cause: " + e, e);
    }
}

```

ConfigParser 对象的 parse 方法调用 parse 方法, 这才是整个程序实现的核心。当调用这个方法的时候, 系统会按照刚才所安排的任务规则开始解析各个节点。然后形成各个层次的对象。

最后执行 XmlParserApplication 对象的 viewProject 方法第三步骤。

```
project.showContent();
```

这样就可以看到我们的实现结果。

6.5 读取源码的收获

通过理解 iBATIS SQL Map 对配置文件的读取和解析过程, 应该有几个收获。

第一, 如何对 XML 文件进行验证。通过 iBATIS SQL Map DTD 文件进行了验证的 dao.xml 才是合法的 XML 文件。iBATIS SQL Map 集合 DOM 和 SAX 来解析和验证 XML 文件, 这些技术都值得我们借鉴。

第二, 对 XML 文件中不同的节点采用不同的处理策略模式的实现方式。采用开闭合设计原则, 可以实现多种 XML 文件和复杂节点的解析和处理。

第三, 抽象出一个通用的 XML 解析框架。在这种模式中, 基于匿名类的策略模式, 可以非常方便地进行扩展和延伸。

第四, 面对多个对象处理, 采用中介者模式。由一个统一的接口和处理, 这也是遵循设计原则中的迪米特法则。

第五, 为了提高应用程序的扩展性, 可以采用对 Bean 进行动态加载技术, 即在配置文件中维护相关的 Bean 类, 在程序应用中提供接口。当需要重新修改和装载特定的实现类时, 可以通过加载类名称的方式提供应用平台的扩展性。

第 7 章

SQL Map 引擎实现框架

本章内容:

1. SQL Map 引擎实现框架组成。说明 SQL Map 的四个组件组成部分和相关的核心类和接口。

2. 业务运行过程和介绍, 分别说明总体业务运行过程序列图和简化说明。

3. 业务实现类的分析。针对三种类型的类进行进一步说明, 包括业务实现类、配置信息类、运行状态信息类。业务实现类包括 `SqlMapClientImpl` 类、`SqlMapSessionImpl` 类、`SqlMapExecutorDelegate` 类、`TransactionManager` 类、`Transaction` 接口、`MappedStatement` 类、`SqlExecutor` 类。配置信息类包括 `ParameterMap` 类、`ResultMap` 类、`Sql` 接口。运行状态信息类包括 `SessionScope` 类和 `StatementScope` 类。

4. 业务实现分析。把整个 SQL Map 引擎分为两个部分。对于第一个部分是通用内容。对于第二个部分, 根据具体业务实现的不同, 进行了细化说明。这些具体业务过程包括: ① 查询类业务实现过程; ② 单事务业务操作实现过程; ③ 联合事务处理实现过程; ④ 存储过程的处理; ⑤ 批处理及其实现; ⑥ 全局 JTA 事务的处理; ⑦ 全局外部事务的处理; ⑧ 用户自定义数据库 Connection 处理。

上面讲解了配置文件读取的实现, 现在根据读取的配置文件来进行业务处理。SQL Map 引擎起到一个承上启下的作用。

7.1 SQL Map 引擎实现框架的组成

SQL Map 引擎覆盖的包主要有 `com.ibatis.sqlmap.client` 包和 `com.ibatis.sqlmap.engine.impl` 包。当然也可以引申到其他实现的包内。但是这两个包, 一个是接口包, 还有一个是接口实现包。其中涉及的组件群有四个, 第 1 个组件群是 Client 接口部分; 第 2 个组件群是 `implement` 组件; 第 3 个组件群是用于数据库处理部分; 第 4 个组件群是 Map 组件群。

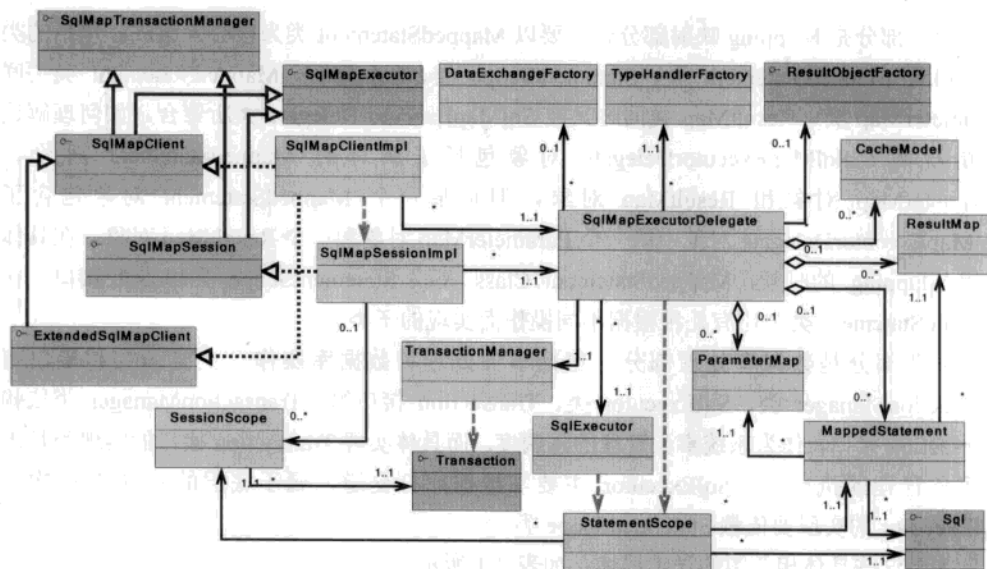


图 7-1 SQL Map 引擎类结构图

在这个类结构图中，基本上描述了 SQL Map 的整体核心框架。如果搞清楚这个框架的结构，基本上也就了解了整个 SQL Map 的结构体系。关于图 7-1 的描述，我想还是分为三个部分来说明。第一部分是接口和实现部分。第二部分是中间引擎部分。第三部分是 Mapping 映射部分。第四部分是数据库操作部分。

第一部分由图 7-1 的左上角组成，其主要功能是把外部的接口转化为内部的实现，主要由 `SqlMapTransactionManager` 接口、`SqlMapExecutor` 接口、`SqlMapClient` 接口、`SqlMapSession` 接口、`ExtendedSqlMapClient` 接口、`SqlMapClientImpl` 接口、`SqlMapSessionImpl` 接口、`SessionScope` 接口组成。`SqlMapClient` 接口继承 `SqlMapTransactionManager` 接口和 `SqlMapExecutor` 接口并由 `SqlMapClientImpl` 类实现。`SqlMapClientImpl` 类关联 `SqlMapExecutorDelegate` 类同时依赖 `SqlMapSessionImpl` 类。`SqlMapSessionImpl` 类实现 `SqlMapSession` 接口，同时关联 `SessionScope` 类，而且也关联 `SqlMapExecutorDelegate` 类。

第二部分是中间引擎部分，主要是以 `SqlMapExecutorDelegate` 类为核心来组成的。包括 `SqlMapExecutorDelegate` 类、`TransactionManager` 类、`MappedStatement` 类、`SqlExecutor` 类、

CacheModel 类、ParameterMap 类、ResultMap 类等。在这个部分中, SqlMapExecutorDelegate 是星型模式, 采取一种发散状的结构。SqlMapExecutorDelegate 类关联 DataExchange Factory 类、TypeHandlerFactory 类、ResultObjectFactory 接口、TransactionManager 类、SqlExecutor 类, 同时也聚合 CacheModel 类、ParameterMap 类、ResultMap 类和 MappedStatement 类。当然, SqlMapExecutorDelegate 类作为中间引擎与外部进行系统内容的实现, 也充当了门面角色。所以, SqlMapExecutorDelegate 类被 SqlMapClientImpl 类和 SqlMapSessionImpl 类所关联。

第三部分是 Mapping 映射部分, 主要以 MappedStatement 类为核心来组成。包括的类和接口有 MappedStatement 类、StatementScope 类、Sql 接口。MappedStatement 类关联 ParameterMap 类、ResultMap 类同时又被 SqlMapExecutorDelegate 类所聚合。如何理解这种情况呢? SqlMapExecutorDelegate 对象包括了所有的 MappedStatement 对象、ParameterMap 对象和 ResultMap 对象, 但是某一个 MappedStatement 对象包含了 SqlMapExecutorDelegate 对象中的一个 ParameterMap 对象和一个 ResultMap 对象。在具体运用 Mapping 的时候, MappedStatement Class 关联 StatementScope 类和 Sql 接口。在 MappedStatement 类中还有几种根据不同操作而实现的子类。

第四部分是数据库处理部分, 包括事务处理和数据库操作。主要的类和接口有 TransactionManager 类、SqlExecutor 类、Transaction 接口等。TransactionManager 类依赖 Transaction 接口, 作为系统事务管理的总调度。而具体实现 Transaction 接口的实现类是进行事务管理的执行体。SqlExecutor 主要与数据库打交道, 属于底层的 JDBC 层次。SqlExecutor 的实现要依赖 StatementScope 类。

关于这些具体相关类的详细描述, 如表 7-1 所示。

表 7-1 SQL Map 的整体核心接口和类及其功能说明列表

接口或类	功能或用途描述	备注
com.ibatis.sqlmap.client.SqlMapTransactionManager	此接口声明了 SQL Map 的事务方法	
com.ibatis.sqlmap.client.SqlMapExecutor	此接口声明了 SQL Map 的执行语句和批处理的方法	
Scom.ibatis.sqlmap.client.SqlMapClient	SQL Maps 中一个线程安全性的客户端接口, 该接口继承了 SqlMapTransactionManager 的事务方法和 SqlMapExecutor 执行语句和批处理的方法	
com.ibatis.sqlmap.client.SqlMapClientBuilder	客户端获得 SqlMapClient 对象的工厂类	
com.ibatis.sqlmap.client.SqlMapSession	SQL Maps 中一个单线程安全性的客户端接口, 该接口继承了 SqlMapTransactionManager 的事务方法和 SqlMapExecutor 执行语句和批处理的方法	
com.ibatis.sqlmap.engine.impl.ExtendedSqlMapClient	这个类是不必要的, 应尽快删除。目前, spring 框架集成依赖它	
com.ibatis.sqlmap.engine.impl.SqlMapClientImpl	SqlMapClientImpl 是实现外部调用 SqlMapClient 接口的实现类, 但其本身不做具体代码操作, 所有工作都转移到 SqlMapSession Impl 对象	

续表

接口或类	功能或用途描述	备注
com.ibatis.sqlmap.engine.impl.SqlMapSessionImpl	SqlMapSessionImpl 是实现当前会话，但其本身不做具体代码操作，所有工作都转移到 SqlMapExecutorDelegate 对象	
com.ibatis.sqlmap.engine.scope.SessionScope	SQL Map 用于会话的范围接口实现	
com.ibatis.sqlmap.engine.impl.SqlMapExecutorDelegate	应用的一级门面对象，基本上在 iBATIS SQL Map 的所有应用操作都要在这里交汇、调度、转移。该对象保留了基本上所有 SQL Map 配置文件和映射文件的信息	
com.ibatis.sqlmap.engine.transaction.TransactionManager	SQL Map 的事务管理	
com.ibatis.sqlmap.engine.transaction.Transaction	SQL Map 的事务操作接口	
com.ibatis.sqlmap.engine.exchange.DataExchangeFactory	iBATIS 数据交换对象的工厂类	
com.ibatis.sqlmap.engine.type.TypeHandlerFactory	iBATIS 数据类型转化的工厂类	
com.ibatis.sqlmap.engine.mapping.result.ResultObjectFactory	iBATIS 用这个工厂类是对 ResultMap 的补充。主要是执行语句后，返回结果为一个对象的情况。当然，这要在 SQLMapConfig 做好配置	
com.ibatis.sqlmap.engine.cache.CacheModel	iBATIS 缓存的包装类	
com.ibatis.sqlmap.engine.mapping.result.ResultMap	SQL Map 输出结构的包装类	
com.ibatis.sqlmap.engine.mapping.parameter.ParameterMap	SQL Map 输入参数的包装类	
com.ibatis.sqlmap.engine.mapping.statement.MappedStatement	SQL Map 用于映射的语句处理类	
com.ibatis.sqlmap.engine.scope.StatementScope	SQL Map 用于 Statement 请求的范围接口实现	
com.ibatis.sqlmap.engine.mapping.SQL.SQL	SQL 语句的处理接口	
com.ibatis.sqlmap.engine.execution.SqlExecutor	负责执行 SQL 语句的执行类	

7.2 业务运行过程和介绍

总体业务描述主要是让读者从一个整体框架上了解系统的实现结构。在每个实现的过程中都会有这样的选择和那样的处理，这样，很容易让代码阅读者了解细节而忘掉了整体。

通过两种方式来说明，一种是通过类和接口的序列图，另一种是用一种比较简洁的说明来阐述这个系统的实现过程。

7.2.1 总体业务运行过程序列图

业务的执行主要是在这几个类之间进行操作，但是不同的业务操作的细节是不同的。因此笔者先用一个统一的类来说明整体的操作，然后根据不同的情况来进行不同的说明。总体框架序列图如图 7-2 所示。

实现步骤如下：

(1) Client 调用 SqlMapClientImpl 实例对象的 CRUD 方法。

(2) SqlMapClientImpl 实例对象获得 SqlMapSessionImpl 对象，如果没有则创建一个 SqlMapSessionImpl 对象，

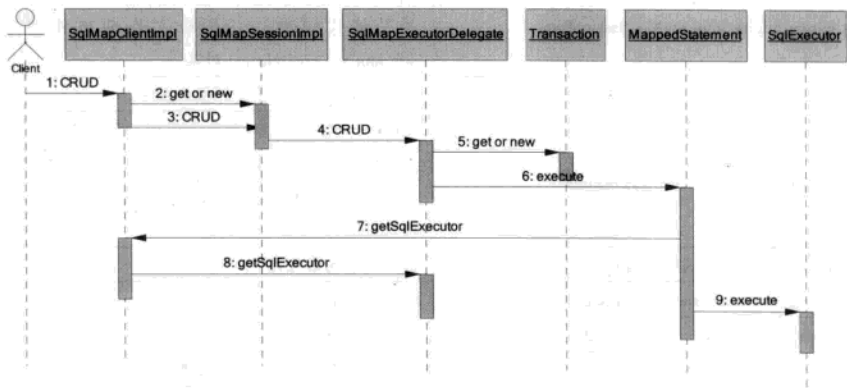


图 7-2 SQL Map 引擎的序列图

- (3) SqlMapClientImpl 对象向 SqlMapSessionImpl 对象调用 CRUD 方法。
 - (4) SqlMapSessionImpl 对象调用 SqlMapExecutorDelegate 对象的 CRUD 方法。
 - (5) SqlMapExecutorDelegate 对象获得 Transaction 对象，如果没有则创建一个 Transaction 对象。
 - (6) SqlMapExecutorDelegate 对象调用 MappedStatement 对象的 execute 方法。
 - (7) MappedStatement 对象调用 SqlMapClientImpl 对象的 getSqlExecutor 方法，返回 SqlExecutor 对象。
 - (8) SqlMapClientImpl 对象调用 SqlMapExecutorDelegate 对象的 getSqlExecutor 方法，返回 SqlExecutor 对象。
 - (9) MappedStatement 对象调用 SqlExecutor 对象的 execute 方法。完成业务的操作。
- 上述的过程只是一个主要的、核心的过程。在这个过程中，还进行了很多的处理。
- SqlMapExecutorDelegate 在进行查询和业务处理的时候会获取该查询和业务处理的 id，并调用对应的 DeleteStatement、InsertStatement、ProcedureStatement、SelectStatement 或 UpdateStatement 实例化对象。新建一个针对该对象的状态对象——StatementScope 实例化对象。
- 然后，Statement 对象带着这个 StatementScope 状态对象去处理业务。Statement 对象会调用 Sql Executor 对象去对数据库进行操作。如果是增加、修改和删除操作，SqlExecutor 对象会返回一个返回值，并清理处理现场。如果是查询，那么 SqlExecutor 对象还要把获得的 ResultSet 对象通过 ResultMap 对象进行处理，转化成 JavaBean、Map 获得 XML 对象，并返回出来。

7.2.2 系统总体运行简化说明图

系统总体运行基本上可以用图 7-3 来说明。为了形象说明，在这里主要有五类角色。

分别是 SqlMapClientImpl 角色、SqlMapSessionImpl 角色、SqlMapExecutorDelegate 角色、MappedStatement (包括 InsertStatement 角色、UpdateStatement 角色、DeleteStatement 角色、SelectStatement 角色和 ProcedureStatement 角色) 角色和 SqlExecutor 角色。

(1) 接口层

客户端通过 SqlMapClient 接口进行调用, 首先是获得了 SqlMapClientImpl 角色。由于应用程序要适应多线程操作, 客户端进行调用就要给它形成一个独立的会话, 所以 SqlMapClientImpl 角色为某个客户端建立该客户独立的会话角色, 即 SqlMapSessionImpl 角色。

(2) 会话层

SqlMapSessionImpl 角色在形成的过程中保持自己的状态, 所以它要附带一个 SessionScope 的状态参数实体。该状态参数实体要起到一个非常重要的功能就是实现事务状态的控制。这也是由多线程的操作转化为单线程的事务操作所必须要考虑的问题。SqlMapSessionImpl 角色并不能处理业务, 它转交给 SqlMapExecutorDelegate 角色。在转交过程中把 SessionScope 的状态参数实体也作为变量传递过去。

(3) 调度层

SqlMapExecutorDelegate 角色是整个应用程序的中枢和门面, 所有的配置信息最后都汇总到 SqlMapExecutorDelegate 角色, 在 SqlMapExecutorDelegate 角色有一个容器, 里面包括了所有配置信息中所定义的操作名称容器、参数实体容器、输入实体容器、缓存实体容器等。SqlMapExecutorDelegate 角色首先要根据 SqlMapSessionImpl 角色传递的状态信息判断当前的事务环境, 然后调整自己的应付策略, 如果用户已经启动了事务, 那就不对事务进行处理; 如果用户没有启动事务, 那说明这次操作的整个事务都由 SqlMapExecutorDelegate 角色来掌握, 于是 SqlMapExecutorDelegate 角色自主地启动事务, 然后操作完成后, 自主地进行提交和回滚。当然, 在调用这些角色的过程中, 还要给当前的 MappedStatement 角色保持状态, 这样要增加一个 StatementScope 的状态参数实体。该状态参数实体主要是保持当前 MappedStatement 角色的一些状态信息, 甚至是 MappedStatement 角色本身。

(4) 映射层

在生成 MappedStatement 角色时, SqlMapExecutorDelegate 角色还要根据 SqlMapSessionImpl 角色提交过来的信息判断该操作的编号、需要什么参数资源、需要什么输出资源、是否要缓存或者要缓存的编号等。然后 SqlMapExecutorDelegate 角色在自己的容器中按图索骥, 如果是新增数据操作, 则从容器中调用 InsertStatement 角色; 如果是修改数据操作, 则从容器中调用 UpdateStatement 角色; 如果是删除数据操作, 则从容器中调用 DeleteStatement 角色; 如果是查询数据操作, 则从容器中调用 SelectStatement 角色; 如果是要执行存储过程, 则从容器中调用 ProcedureStatement 角色。当然, 如果操作都不属于上述类型, 那就调用 MappedStatement 角色。同时还要检查该 MappedStatement 角色是否需要缓存, 否则就调用继承 MappedStatement 角色的 CachingStatement 角色。

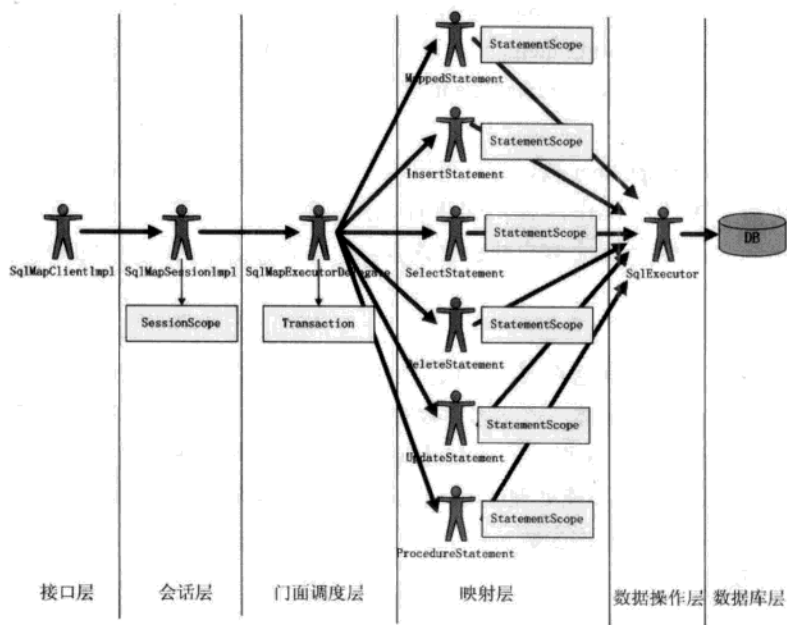


图 7-3 SQL Map 引擎的实现简化图

(5) 数据库操作层

在这里转了一个弯弯，当 `SqlMapExecutorDelegate` 角色为执行操作而调用 `MappedStatement` 角色时，`MappedStatement` 角色只是把相关的操作信息（包括自己）放到了 `StatementScope` 状态参数实体，然后 `MappedStatement` 角色把 `StatementScope` 状态参数实体作为参数从 `SqlMapExecutorDelegate` 角色中获得 `SqlExecutor` 角色，最后调用 `SqlExecutor` 角色去执行。`SqlExecutor` 角色基本上是 JDBC 的实现，算是 iBATIS 对数据库操作的最后一道防线，它通过 `StatementScope` 状态参数实体来获取相关的 SQL 语句、包括传入的参数，就可以直接对数据库进行操作了。操作完成后的返回值，也是通过 `StatementScope` 状态参数实体来获得返回格式说明来进行组装返回对象，并把这些返回值对象回传给 `SqlMapExecutorDelegate` 角色，`SqlMapExecutorDelegate` 角色把返回值对象回传给 `SqlMapSessionImpl` 角色，`SqlMapSessionImpl` 角色把返回值对象回传给 `SqlMapClientImpl` 角色，最后 `SqlMapClientImpl` 角色把返回值对象传给客户端，这样便完成了一个业务的操作。

(6) 数据库层

最终的数据库系统，主要就是关系数据库。

7.3 业务实现类的分析

从一个平台业务实现过程的这些类而言，主要包括三种类型的类，分别为有业务实现

的类，有配置信息的类，也有运行状态信息的类。

其中业务实现的类主要是 SqlMapClientImpl、SqlMapSessionImpl、SqlMapExecutorDelegate、TransactionManager、Transaction、MappedStatement 和 SqlExecutor 等。业务实现类主要是一些有具体的操作，在核心流程环节中充当一定的角色和承担一定的实现功能。

配置信息的类和接口有 ParameterMap 类、ResultMap 类和 Sql 接口等，主要用于数据缓存，即配置文件信息在内存中的体现。

而运行状态信息的类有两个：SessionScope 类和 StatementScope 类，主要是保持业务实现类的状态，会随着业务进展的变化而发生变化。

7.3.1 业务实现类

业务实现类是 iBATIS 平台的核心实现体系。主要由 SqlMapClientImpl 类、SqlMapSessionImpl 类、SqlMapExecutorDelegate 类、TransactionManager 类、MappedStatement 类、SqlExecutor 类等组成。

1. SqlMapClientImpl 的属性、方法和作用

当我们进行解析 SQLMap 配置文件的时候，在形成 SqlMapConfiguration 实例化对象的同时，就创建了一个 SqlMapClientImpl。其类结构如图 7-4 所示。

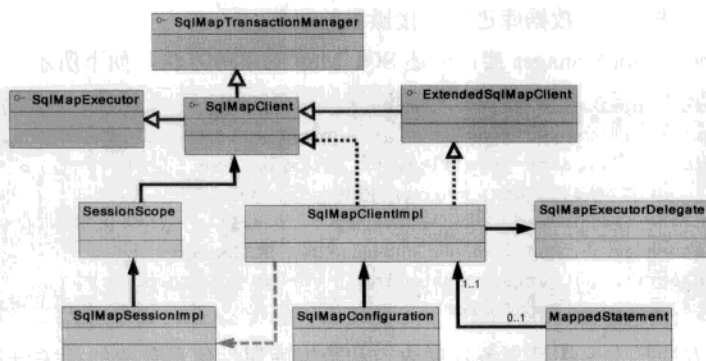


图 7-4 SqlMapClientImpl 类结构图

关于 SqlMapClientImpl 的内容在前面已经做了简单描述，这里进行更详细的说明。由于 SqlMapClientImpl 实现了 SqlMapClient 接口，而 SqlMapClient 又继承 SqlMapExecutor 接口和 SqlMapTransactionManager 接口。所以，有必要简单地了解一下 SqlMapExecutor 接口和 SqlMapTransactionManager 接口。

SqlMapExecutor 接口描述 SqlMap 的执行语句和批处理的方法，如下所示。

```

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

```

```

int update(String id) throws SQLException;
int delete(String id, Object parameterObject) throws SQLException;
int delete(String id) throws SQLException;
Object queryForObject(String id, Object parameterObject) throws SQLException;
Object queryForObject(String id) throws SQLException;
Object queryForObject(String id, Object parameterObject, Object resultObject)
throws SQLException;
List queryForList(String id, Object parameterObject) throws SQLException;
List queryForList(String id) throws SQLException;
List queryForList(String id, Object parameterObject, int skip, int max) throws
SQLException;
List queryForList(String id, int skip, int max) throws SQLException;
void queryWithRowHandler(String id, Object parameterObject, RowHandler rowHandler)
throws SQLException;
void queryWithRowHandler(String id, RowHandler rowHandler) throws SQLException;
PaginatedList queryForPaginatedList(String id, Object parameterObject, int
pageSize) throws SQLException;
PaginatedList queryForPaginatedList(String id, int pageSize) throws SQLException;
Map queryForMap(String id, Object parameterObject, String keyProp) throws
SQLException;
Map queryForMap(String id, Object parameterObject, String keyProp, String valueProp)
throws SQLException;
void startBatch() throws SQLException;
int executeBatch() throws SQLException;
List executeBatchDetailed() throws SQLException, BatchException;

```

以上这些方法都是对数据库进行直接操纵的语句。

SqlMapTransactionManager 接口描述 **SQL Map** 的事务方法，如下所示。

```

public void startTransaction() throws SQLException;
public void startTransaction(int transactionIsolation) throws SQLException;
public void commitTransaction() throws SQLException;
public void endTransaction() throws SQLException;
public void setUserConnection(Connection connection) throws SQLException;
public Connection getUserConnection() throws SQLException;
public Connection getCurrentConnection() throws SQLException;
public DataSource getDataSource();

```

以上这些方法基本上都是事务处理内容和数据库属性信息获取的相关语句。

SqlMapClient 接口还有一些自己的方法，如下所示。

```

public SqlMapSession openSession();
public SqlMapSession openSession(Connection conn);
public SqlMapSession getSession();
public void flushDataCache();
public void flushDataCache(String cacheId);

```

SqlMapClient 接口方法主要是与 **session** 相关的，实际上也就是与线程是有联系的。

SqlMapClient 接口继承了 **SqlMapExecutor** 接口和 **SqlMapTransactionManager** 接口。在 **SQL Map** 中是一个中心接口。一方面，这个接口可以执行映射语句（如新增、修改、删除

和查询),同时也定义了事务处理,还包括了批处理等。在 SqlMapClient 接口基本上可以实现 SQL Map 上所有的功能,而且。也就是要求 SqlMapClient 接口的实现类既要实现对数据库的操作,又要实现与事务相关的管理。同时,还要针对自身的线程属性进行处理。

SqlMapExecutor 接口的定义是 SQL Map 的各种 SQL 操作和批处理操作,如新增、修改、删除、存储过程等。而 SqlMapTransactionManager 接口的定义是与数据库事务管理相关的操作,如开启事务、提交事务、回滚事务和结束事务等。SqlMapClient 同时继承 SqlMapExecutor 接口和 SqlMapTransactionManager 接口,这样既包括 SQL 和批处理操作,又包含事务操作。而 ExtendedSqlMapClient 接口不是必须要的,应尽快删除。目前, Spring 集成 iBATIS 时依赖于它,但这个接口可以作为一个扩展接口来考虑。

SqlMapClientImpl 是 SqlMapClient 接口和 ExtendedSqlMapClient 接口的实现类。所以,所有 SqlMapClient 的方法都要在 SqlMapClientImpl 中得到实现。

SqlMapClientImpl 关联 SqlMapExecutorDelegate,主要是应用在配置信息时使用。

SqlMapClientImpl 依赖 SqlMapSessionImpl 类才是这个程序的核心内容,这样可以实现单个会话的单线程操作。关于如何实现多线程,在下面会详细描述。

SqlMapClientImpl 类是实现外部的接口,是与用户接口的实现,其属性如表 7-2 所示。

表 7-2 SqlMapClientImpl 类属性列表及其说明

属性名称	类 型	功能和用途	备注
Delegate	public SqlMapExecutorDelegate	这是在 SqlMapClientImpl 对象中的核心类	
localSqlMapSession	protected ThreadLocal	这是用于转化多个线程为单线程的 ThreadLocal	

SqlMapClientImpl 类,方法列表及其说明如表 7-3 所示。

表 7-3 SqlMapClientImpl 类方法列表及其说明

方法名称	参 数	返 回 值	功能和用途	备注
queryForObject	String id, Object parameterObject)	Object	根据一个参数对象进行查询,返回一个查询结果的对象。处理过程转移给 SqlMapSessionImpl 相应方法	
queryForObject	String id	Object	直接查询,返回一个查询结果的对象。处理过程转移给 SqlMapSessionImpl 相应方法	
queryForObject	String id, Object parameterObject, Object resultObject	Object	根据一个参数对象进行查询,返回一个已经采用输出格式封装好的查询对象。处理过程转移给 SqlMapSessionImpl 相应方法	
queryForList	String id, Object parameterObject	List	根据一个参数对象进行查询,返回一个查询结果的列表。处理过程转移给 SqlMapSessionImpl 相应方法	
queryForList	String id	List	直接查询,返回一个查询结果的列表。处理过程转移给 SqlMapSessionImpl 相应方法	
queryForList	String id, Object parameterObject, int skip, int max	List	根据一个参数对象进行查询,按照开始序号和最大返回值的规则返回一个查询结果的列表。处理过程转移给 SqlMapSessionImpl 相应方法	

续表

方法名称	参 数	返 回 值	功能和用途	备注
QueryWithRow Handler	String id, Object parameterObject, RowHandler rowHandler	无	根据一个参数对象进行查询, 按照 Row Handler 格式回传, 没有返回值。处理过 程转移给 SqlMapSessionImpl 相应方法	
QueryWithRow Handler	String id, RowHandler rowHandler	无	直接查询, 按照 RowHandler 格式回传, 没有返回值。处理过程转移给 SqlMapSessionImpl 相应方法	
queryForMap	String id, Object parameterObject, String keyProp	Map	根据一个参数对象进行查询, 返回一个 查询结果的 Map 对象。处理过程转移给 SqlMapSessionImpl 相应方法	
queryForMap	String id, Object parameterObject, String keyProp, String valueProp	Map	根据一个参数对象进行查询, 返回一个 查询结果的 Map 对象。处理过程转移给 SqlMapSessionImpl 相应方法	
insert	String id, Object parameterObject)	无	根据一个参数对象插入一条记录。处理 过程转移给 SqlMapSessionImpl 相应方法	
insert	String id	无	直接插入一条记录。处理过程转移给 SqlMapSessionImpl 相应方法	
update	String id, Object parameterObject	无	根据一个参数对象修改一条记录。处理 过程转移给 SqlMapSessionImpl 相应方法	
update	String id	无	直接修改一条记录。处理过程转移给 SqlMapSessionImpl 相应方法	
delete	String id, Object parameterObject	无	根据一个参数对象删除一条记录。处理 过程转移给 SqlMapSessionImpl 相应方法	
delete	String id	无	直接删除一条记录。处理过程转移给 SqlMapSessionImpl 相应方法	
startTransaction	无	无	启动一个事务。处理过程转移给 SqlMapSessionImpl 相应方法	
startTransaction	int transactionIsolation	无	按照一定的事务隔离级别启动一个事 务。处理过程转移给 SqlMapSessionImpl 相应方法	
commitTransaction	无	无	提交一个事务。处理过程转移给 SqlMapSessionImpl 相应方法	
endTransaction	无	无	结束一个事务。处理过程转移给 SqlMapSessionImpl 相应方法	

这些方法大致可以分为两类, 一类用于事务管理, 另一类用于 CRUD 操作。这些方法全是外部可以调用的方法。

在 SqlMapClientImpl 要求实现事务线程的安全性。如何实现线程安全性?
用 ThreadLocal 来实现 SqlMapClientImpl 的线程安全性。当使用 ThreadLocal 维护变量时, ThreadLocal 为每个使用该变量的线程提供独立的变量副本, 所以每一个线程都可以独立地改变自己的副本, 而不会影响其他线程所对应的副本。

这里采用的设计模式是单例模式。所谓单例模式，就是保证一个类仅有一个实例，并提供一个访问它的全局访问点。单例模式属于创建型模式，单例模式确保某一个类只有一个实例，而且自行实例化并向整个系统提供这个实例单例模式，也就是采取一定的方法保证在整个软件系统中，对某个类只能存在一个对象实例，并且其他类可以通过某种方法访问该实例。单例模式只应在有真正的“单一实例”的需求时才可使用。Singleton 结构如图 7-5 所示，单例模式只有一个角色，就是要进行单例的类。

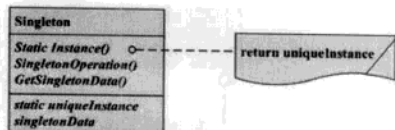


图 7-5 单例模式结构

实现如果如下：

在 SqlMapClientImpl 有变量，代码如下：

```
protected ThreadLocal localSqlMapSession = new ThreadLocal();
```

采用单例模式，实现本线程的副本。

```
protected SqlMapSessionImpl getLocalSqlMapSession() {
    SqlMapSessionImpl sqlMapSession = (SqlMapSessionImpl) localSqlMapSession.get();
    if (sqlMapSession == null || sqlMapSession.isClosed()) {
        sqlMapSession = new SqlMapSessionImpl(this);
        localSqlMapSession.set(sqlMapSession);
    }
    return sqlMapSession;
}
```

其方法是外部调用的全部方法实现，包括各种 CRUD 操作，其功能是实现与外部的接口，是一个边界类。

单例模式有两种实现方式，一种叫懒汉式单例模式，另一种叫饿汉式单例模式。懒汉式单例模式表示当外部需要对象。才去查找是否有对象，如果有，就返回该对象；如果没有，就创建一个并返回对象。饿汉式单例模式表示当不管外部是否需要对象，只要类加载就提供实例对象。本代码实现模式是采用懒汉式单例模式，当需要 SqlMapSessionImpl 对象的时候，便到线程池去寻找，如果找不到存在的 SqlMapSessionImpl 对象，就开始创建一个新的 SqlMapSessionImpl 对象。

SqlMapClientImpl 类主要是联系外部的接口。

2. SqlMapSessionImpl 的属性、方法和作用

SqlMapSessionImpl 就是 SqlMapClientImpl 的变量副本，也就是 SqlMapClientImpl 对象的单线程实现。在 SqlMapSessionImpl 中有一个 SessionScope 类型的变量 sessionScope，主要是为不同的 session 提供不同的状态。实际上，对 SqlMapSessionImpl 的利用就是对 sessionScope 对象中的属性的利用。

SqlMapSessionImpl 相关的类结构如图 7-6 所示。

SqlMapSession 接口与 SqlMapClient 接口一样都继承了 SqlMapTransactionManager 和 SqlMapExecutor 接口，所以它同时也具有事务处理和 SQL 操作处理的方法。

SqlMapSessionImpl 类实现 SqlMapSession 接口，也即实现了事务处理和 SQL 操作处理的方法。同时关联 SessionScope 类。这样可以保持当前会话的信息。而 SessionScope 类却关联 SqlMapTransactionManager、SqlMapExecutor 接口和 SqlMapClient 接口。SqlMapSessionon Impl 类属性列表如表 7-4 所示。

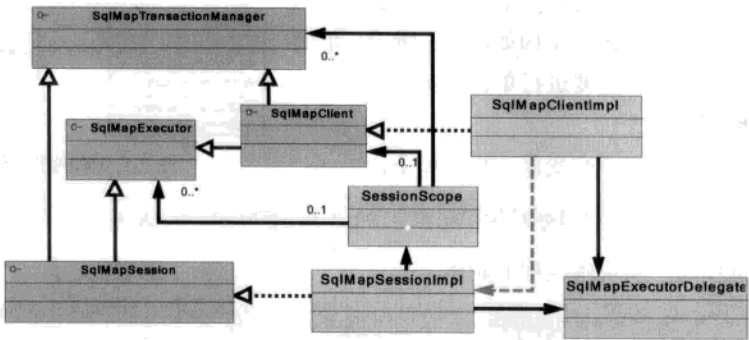


图 7-6 SqlMapSessionImpl 类结构图

表 7-4 SqlMapSessionImpl 类属性列表

属性名称	类型	功能和用途	备注
delegate	protected SqlMapExecutorDelegate	当前会话的门户调度实例化对象	
sessionScope	protected SessionScope	当前会话的环境变量	
closed	protected boolean	用于关闭当前会话	

SqlMapSessionImpl 方法与 SqlMapClientImpl 基本相同，甚至名称都一样，包括各种 CRUD 操作。但是 SqlMapSessionImpl 类的主要功能是实现装载一个保存 SessionI 状态信息的对象。SqlMapSessionImpl 与 SqlMapClientImpl 一样，都属于一个边界类。即把 SqlMapClientImpl 调用的方法，加载一个环境变量，然后调用门面业务类——SqlMapExecutorDelegate。例如其中的一个 update 方法，代码如下：

```
public int update(String id, Object param) throws SQLException {
    return delegate.update(sessionScope, id, param);
}
```

在这里就不一一列举里面的方法了。

3. SqlMapExecutorDelegate 的属性、方法和作用

SqlMapExecutorDelegate 是业务处理门户。所有的操作，都是通过 SqlMapExecutorDelegate 来实现的，在 SqlMapExecutorDelegate 实现过程中，由于带有不同 SqlMapSessionImpl 对象的 SessionScope 对象信息，这样可以保证线程的安全性，其类结构图 7-7 所示。

SqlMapExecutorDelegate 是一个大容器，是大部分配置信息的汇总点，同时也是所有操作的接口门面。SqlMapExecutorDelegate 这些变量全都是全局性或者是重量级变量，是

实现平台的门面接口，是与用户接口和后台实现的中转站的实现，其属性如表 7-5 所示。

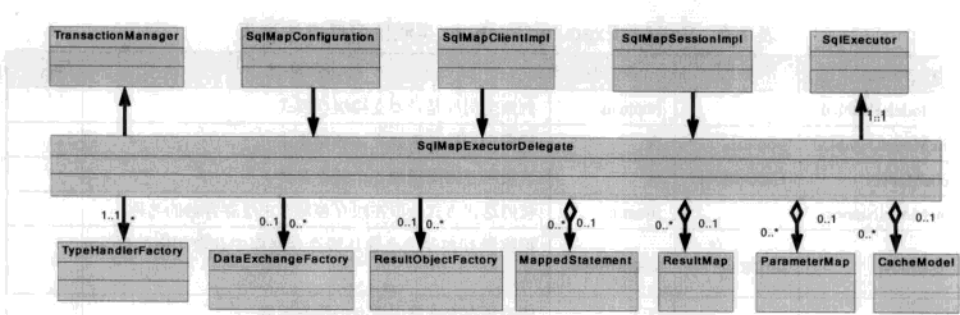


图 7-7 SqlMapExecutorDelegate 类结构图

表 7-5 SqlMapExecutorDelegate 类属性列表

属性名称	类型	功能和用途	备注
lazyLoadingEnabled	boolean	全局性地启用或禁用 SqlMapExecutorDelegate 的所有延迟加载	
cacheModelsEnabled	boolean	全局性地启用或禁用 SqlMapExecutorDelegate 的所有缓存 model	
enhancementEnabled	boolean	全局性地启用或禁用运行时字节码增强，以优化访问 Java Bean 属性的性能，同时优化延迟加载的性能	
useColumnLabel	boolean		
forceMultipleResultSetSupport	boolean		
statementCacheEnabled	boolean	全局性地启用或禁用 SqlMapExecutorDelegate 的 statement 是否可以缓存	
txManager	TransactionManager	用于事务处理	
mappedStatements	HashMap	用于映射语句处理的 Map	
cacheModels	HashMap	用于缓存处理的 Map	
resultMaps	HashMap	用于显示结果的 Map	
parameterMaps	HashMap	用于参数处理的 Map	
SqlExecutor	SqlExecutor	最终的 SQL 执行体对象	
typeHandlerFactory	TypeHandlerFactory	数据类型转化的工厂对象	
dataExchangeFactory	DataExchangeFactory	数据交换的工厂对象	
resultObjectFactory	ResultObjectFactory	结果对象的工厂对象	

SqlMapExecutorDelegate 类的方法可以分为三类：第一类是属于进行赋值变量方法，第二类是执行语句和批处理的方法，第三类是操纵事务语句。

对于第一类赋值变量方法，由于 SqlMapExecutorDelegate 类是一个配置信息的总汇聚地，其中配置文件的全局变量、事务信息和映射文件的缓存信息、输入参数信息、输出格式信息、映射语句信息等都要通过解析器传送过来。所以，第一类方法更趋向把 SqlMapExecutorDelegate 类作为 JavaBean 来进行处理，这些方法名称也多是 set、get 或 is 前缀，也有一些 add 前缀，主要是针对 SqlMapExecutorDelegate 类中的 Map 变量增加对象。

例如，表 7-6 是针对 CacheModel 缓存处理的方法。

表 7-6 SqlMapExecutorDelegate 类赋值变量方法列表

方法名称	参数	返回值	功能和用途	备注
isCacheModelsEnabled	无	boolean	判断全局性是否要支持缓存模式	
setCacheModelsEnabled	boolean	无	设置全局性是否要支持缓存模式	
addCacheModel	CacheModel	无	给缓存容器库增加一个缓存容器	
getCacheModelNames		Iterator	按照迭代方式获取在缓存容器库中缓存容器的名称	
getCacheModel	String id	CacheModel	根据缓存容器的名称从缓存容器库中获取相应的缓存容器	
flushDataCache		无	清空缓存容器库中所有的缓存容器	
flushDataCache	String id	无	清空缓存容器库中指定名称的缓存容器	

对于类似的方法还包括输入参数信息 (ParameterMaps)、输出格式信息 (ResultMaps)、映射语句信息 (MappedStatement)、事务信息 (TransactionManager) 等，在这里就不一一列举了。

SqlMapExecutorDelegate 类的第二类方法是执行语句和批处理的方法，实际上就是类似于实现 SqlMapExecutor 接口的方法。SqlMapExecutor 接口的方法经过了 SqlMapClient 接口、SqlMapSessionImpl 类和 SqlMapClientImpl 等几轮转化，最后都转移到了 SqlMapExecutorDelegate 类上。这些方法包括如表 7-7 所示的内容。

表 7-7 SqlMapExecutorDelegate 类中执行语句和批处理方法列表

方法名称	参数	返回值	功能和用途	备注
insert	SessionScope, String, Object	Object	根据当前 Session 环境、映射内码和参数对象，处理插入的映射语句	
update	SessionScope, String, Object	int	根据当前 Session 环境、映射内码和参数对象，处理修改的映射语句	
delete	SessionScope, String id, Object	int	根据当前 Session 环境、映射内码和参数对象，处理删除的映射语句	
queryForObject	SessionScope, String id, Object, Object	Object	根据当前 Session 环境、映射内码、参数对象和输出对象，处理查询的映射语句，返回一个遵照输出对象格式的对象	
queryForList	SessionScope, String id, Object, int skip, int max	List	根据当前 Session 环境、映射内码、参数对象、输出的第几位，输出最大数量，处理查询的映射语句，返回一个遵照输出对象格式的对象组成的列表	
queryWithRowHandler	SessionScope, String, Object, RowHandler	无	根据当前 Session 环境、映射内码、参数对象、RowHandler 对象，处理查询的映射语句，无返回值	

续表

方法名称	参 数	返回值	功能和用途	备注
queryForMap	SessionScope , String , Object, String, String	Map	根据当前 Session 环境、映射内码、参数对象、输出对象的属性名称，处理查询的映射语句，返回一个遵照输出对象的值组成的 Map	
startBatch	SessionScope	无	根据当前 Session 环境开始一个批处理	
executeBatch	SessionScope	int	根据当前 Session 环境执行一个批处理	
executeBatchDetailed	SessionScope	List	根据当前 Session 环境执行一个批处理，返回一个列表	

SqlMapExecutorDelegate 类的第三类方法是操纵事务方法，实际上就是类似于实现 SqlMapTransactionManager 接口的方法。SqlMapTransactionManager 接口的方法经过了 SqlMapClient 接口、SqlMapSessionImpl 类和 SqlMapClientImpl 等几轮转化，最后都转移到 SqlMapExecutorDelegate 类上。这些方法包括如表 7-8 所示的内容。

表 7-8 SqlMapExecutorDelegate 类中操纵事务方法列表

方法名称	参 数	返回值	功能和用途	备注
startTransaction	SessionScope	无	根据当前 Session 环境开始一个事务	
startTransaction	SessionScope, int	无	根据当前 Session 环境和事务隔离性质开始一个事务	
commitTransaction	SessionScope	无	根据当前 Session 环境提交一个事务	
endTransaction	SessionScope	无	根据当前 Session 环境结束一个事务	

4. TransactionManager 类和 Transaction 接口的作用

TransactionManager 类和 Transaction 接口都是用来处理事务管理的，其类结构如图 7-8 所示。

关于 TransactionManager 类和 Transaction 接口的详细说明和描述，可参见第 8 章的内容。

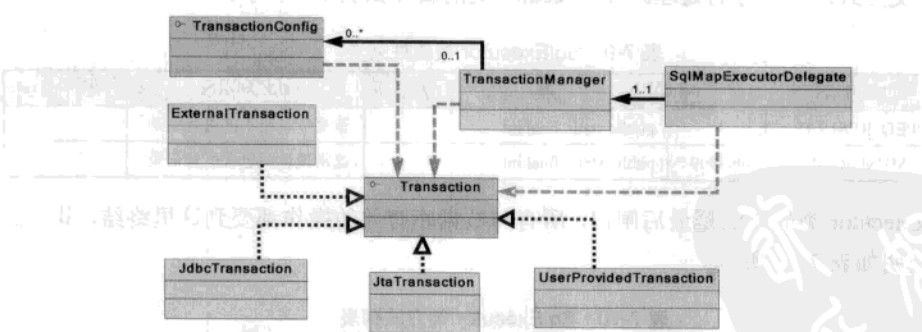


图 7-8 Transaction 类结构图

5. MappedStatement 类的属性、方法和功能

对于 SQLMap 的操作，尤其是 SQL 语句、输入参数和输出结果的的处理，都是通过

MappedStatement 实例化的对象来实现,或者是继承 MappedStatement 类的其他实例化对象来实现的。MappedStatement 类结构如图 7-9 所示。

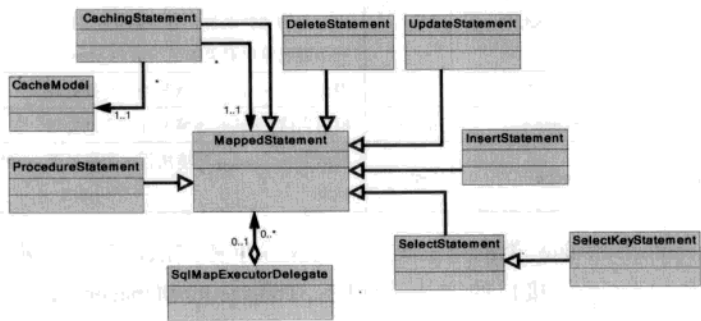


图 7-9 MappedStatement 类结构图

MappedStatement 类覆盖 CachingStatement、DeleteStatement、UpdateStatement、InsertStatement、ProcedureStatement、SelectStatement、SelectKeyStatement 等。其中, Delete Statement、UpdateStatement、InsertStatement、ProcedureStatement、SelectStatement、SelectKey Statement 都继承 MappedStatement。

CachingStatement 继承 MappedStatement 并关联 MappedStatement,同时也关联了 CacheModel。这样可以实现 MappedStatement 的缓存功能。SelectKeyStatement 继承 SelectStatement,其作用是实现自动增长值。

关于 MappedStatement 类的详细说明和描述,可参见第 9 章的内容。

6. SqlExecutor 的属性、方法和功能

SqlExecutor 类在实际操作中更具备工具类的性质。该类的实例化对象主要是对 JDBC 进行操作,也就是说,通过上述的转化,最终由 SqlExecutor 对象转化为 SQL 语言,按照 JDBC 提交方式进行事务的处理。SqlExecutor 类的属性如表 7-9 所示。

表 7-9 SqlExecutor 类属性列表

属性名称	类型	功能和用途	备注
NO_SKIPPED_RESULTS = 0;	public static final int	该常量说明不能进行跳动	
NO_MAXIMUM_RESULTS = -999999	public static final int	该常量说明包含所有记录	

SqlExecutor 类的方法是最后闸口,所有与数据库有关的操作都要到这里终结,其方法列表和说明如表 7-10 所示。

表 7-10 SqlExecutor 类方法列表

方法名称	参数	返回值	功能和用途	备注
executeQuery	StatementScope Connection conn, String SQL,	无	执行 SQL 的查询语句	

续表

方法名称	参 数	返回值	功能和用途	备注
	Object[] parameters, int skipResults, int maxResults, RowHandlerCallback callback			
executeUpdate	StatementScope statementScope, Connection connn, String SQL, Object[] parameters	public int	执行 SQL 的新增、删除和 修改语句	
executeQueryProcedure	StatementScope statementScope, Connection connn, String SQL, Object[] parameters, int skipResults, int maxResults, RowHandlerCallback callback	无	调用并执行存储过程	
executeUpdateProcedure	StatementScope statementScope, Connection connn, String SQL, Object[] parameters	public int	调用并执行修改数据的存 储过程	
addBatch	StatementScope statementScope, Connection connn, String SQL, Object[] parameters	无	增加批处理命令	
executeBatch	SessionScope sessionScope	public int	执行批处理	
executeBatchDetailed	SessionScope sessionScope	public List	执行批处理并得到 JDBC 返回的所有信息	

对这七个方法的功能和参数进行详细说明。

```
public void executeQuery(StatementScope statementScope, Connection conn,String
sql,Object[] parameters, int skipResults, int maxResults,RowHandlerCallback
callback)throws SQLException
```

该方法执行 SQL 的查询语句。其中第 1 个参数表示当前 Statement 的 StatementScope 对象,该对象包括了几乎关于当前 Statement、session 和事务环境的一切信息。第 2 个参数表示数据库 Connection 对象,第 3 个参数表示要查询的 SQL 语句。第 4 个参数表示执行 SQL 查询的参数列表。第 5 个参数表示从指定的行编号开始。第 6 个参数表示查询语句的最大行数。第 7 个参数表示返回集的包装类。

```
public int executeUpdate(StatementScope statementScope, Connection connn, String
sql, Object[] parameters) throws SQLException
```

该方法执行 SQL 的新增、删除和修改语句。其中第 1 个参数表示当前 Statement 的 StatementScope 对象,该对象包括了几乎关于当前 Statement、session 和事务环境的一切信

息。第 2 个参数表示数据库 Connection 对象，第 3 个参数表示要新增、删除或修改的 SQL 语句。第 4 个参数表示执行 SQL 查询的参数列表。

```
public void executeQueryProcedure(StatementScope statementScope, Connection
conn, String sql, Object[] parameters,
int skipResults, int maxResults, RowHandlerCallback callback) throws SQLException
```

该方法是通过存储过程执行 SQL 的查询语句。其中第 1 个参数表示当前 Statement 的 StatementScope 对象，该对象包括了几乎关于当前 Statement、session 和事务环境的一切信息。第 2 个参数表示数据库 Connection 对象，第 3 个参数表示要查询的 SQL 语句。第 4 个参数表示执行 SQL 查询的参数列表。第 5 个参数表示从指定的行编号开始。第 6 个参数表示查询语句的最大行数。第 7 个参数表示返回集的包装类。

```
public int executeUpdateProcedure(StatementScope statementScope, Connection conn,
String sql, Object[] parameters)
throws SQLException
```

该方法通过存储过程执行 SQL 的新增、删除和修改语句。其中第 1 个参数表示当前 Statement 的 StatementScope 对象，该对象包括了几乎关于当前 Statement、session 和事务环境的一切信息。第 2 个参数表示数据库 Connection 对象，第 3 个参数表示要新增、删除或修改的 SQL 语句。第 4 个参数表示执行 SQL 查询的参数列表。

```
public void addBatch(StatementScope statementScope, Connection conn, String
sql, Object[] parameters)
throws SQLException
```

该方法是增加批处理命令。其中第 1 个参数表示当前 Statement 的 StatementScope 对象，该对象包括了几乎关于当前 Statement、session 和事务环境的一切信息。第 2 个参数表示数据库 Connection 对象，第 3 个参数表示要新增、删除、查询或修改的 SQL 语句。第 4 个参数表示执行 SQL 查询的参数列表。

```
public int executeBatch(SessionScope sessionScope) throws SQLException
```

该方法是执行批处理命令。其中参数表示表示当前 Statement 的 StatementScope 对象，该对象包括了几乎关于当前 Statement、session 和事务环境的一切信息。

```
public List executeBatchDetailed(SessionScope sessionScope) throws SQLException,
BatchException
```

该方法是执行批处理命令。其中参数表示表示当前 Statement 的 StatementScope 对象，该对象包括了几乎关于当前 Statement、session 和事务环境的一切信息。

7.3.2 配置信息类

1. ParameterMap 信息类

ParameterMap 类以及 ParameterMapping 类都是实现传入参数，由前面的配置信息解析

知道, ParameterMap 类主要是由 SQL Map 文件中的节点来形成的。ParameterMap 类在核心框架中所起的作用如图 7-10 所示。

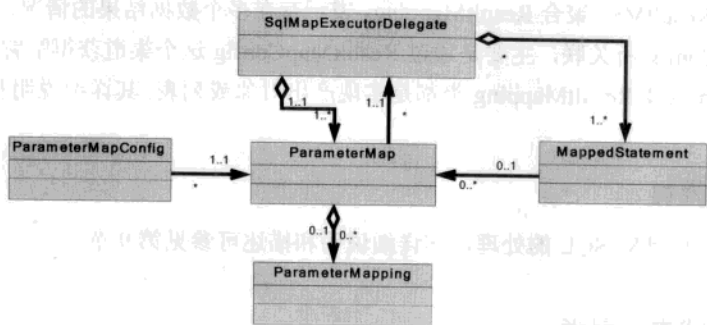


图 7-10 ParameterMap 类结构图

SQL Map 映射文件中的所有 parameterMap 节点都转化到 SqlMapExecutorDelegate 对象 Map 类型的 parameterMaps 变量中。每个 MappedStatement 对象都拥有 0 个或 1 个 ParameterMap 对象来作为参数输入对象。

在核心框架中, ParameterMap 被 SqlMapExecutorDelegate 所聚合, 也被 MappedStatement 所关联。同时 ParameterMap 聚合 ParameterMapping, 表示存在多个参数的情况。而 ParameterMap 被 ParameterMapConfig 所关联, 主要是通过 ParameterMapConfig 这个渠道获得配置信息的。

关于 ParameterMap 实现的详细说明和描述可参见第 9 章。

2. ResultMap 信息类

ResultMap 类以及 ResultMapping 类都是实现输出结果, 由前面的配置信息解析知道, ResultMap 类主要是由 SQL Map 文件中的 resultMap 节点来形成的。ResultMap 类在核心框架中所起的作用如图 7-11 所示。

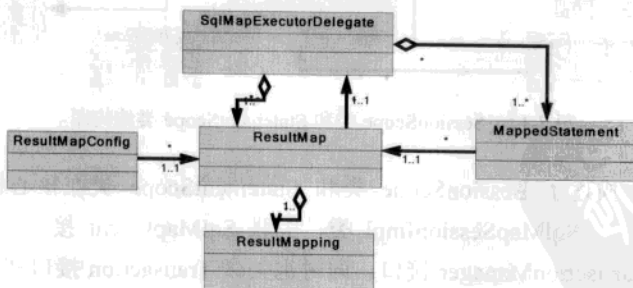


图 7-11 ResultMap 类结构图

SQL Map 映射文件中的所有 resultMap 节点都转化到 SqlMapExecutorDelegate 对象 Map 类型的 parameterMaps 变量中。每个 MappedStatement 对象都拥有 0 个或 1 个 ResultMap

对象来作为参数输入对象。

在核心框架中，ResultMap 被 SqlMapExecutorDelegate 所聚合，也被 MappedStatement 所关联。同时 ResultMap 聚合 ResultMapping，表示存在多个数据结果的情况。而 ResultMap 被 ResultMapConfig 所关联，主要是通过 ResultMapConfig 这个渠道获得配置信息的。

ResultMap 以及 ResultMapping 类都是实现产出对象或列表，其详细说明和描述可参见第 9 章。

3. Sql 信息类

Sql 接口主要针对 SQL 的处理，其详细说明和描述可参见第 9 章。

7.3.3 运行状态信息类

状态信息主要是应用程序在运行中的中间状态。这些状态会影响到具体业务的操作或者是方法的选择。SessionScope 类和 StatementScope 类都属于运行中的状态信息。SessionScope 主要是当前的 Session 信息，而 StatementScope 更多是处理当前的 Statement 信息。两者的类关系如下。

他们在核心框架中的作用如图 7-12 所示。

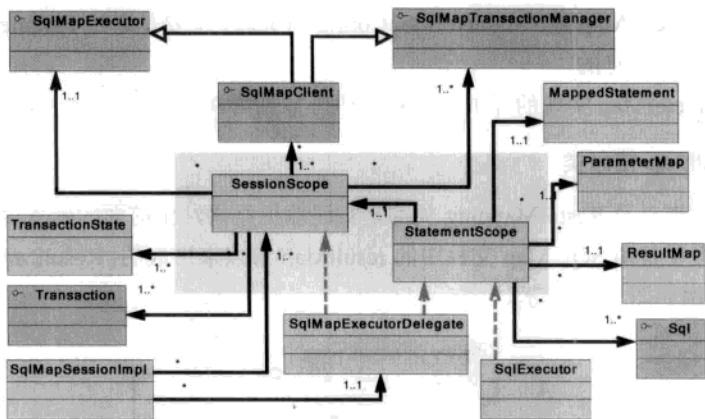


图 7-12 SessionScope 类和 StatementScope 类结构图

在图 7-12 中描述了 SessionScope 类和 StatementScope 类在核心框架中的作用。SessionScope 类基于 SqlMapSessionImpl 类，关联 SqlMapClient 接口、SqlMapExecutor 接口和 SqlMapTransactionManager 接口。同时也关联 Transaction 接口和 TransactionState 类，说明 SessionScope 类主要是用于控制会话的事务状态信息。而且，核心类 SqlMapExecutorDelegate 也依赖 SessionScope 类。

StatementScope 类首先要依赖 SessionScope 类，这样在 SessionScope 对象中的属性信息都可以被 StatementScope 对象所获取和利用。StatementScope 类关联 MappedStatement

类、ParameterMap 类、ResultMap 类和 Sql 接口。这样也让 StatementScope 对象拥有当前 MappedStatement 对象、ParameterMap 对象、ResultMap 对象和 Sql 接口的实现化对象的信息。SqlMapExecutorDelegate 类和 SqlExecutor 类依赖 StatementScope 类，这样可以让 StatementScope 类在这几个对象中充当信息和状态的传递者而进行相互之间的调用。

1. SessionScope 类

SessionScope 类依附在 SqlMapSessionImpl 类，是作为当前会话的一个环境变量，其属性列表和功能用途如表 7-11 所示。

表 7-11 SessionScope 属性列表和功能用途说明

属性名称	类型	功能和用途	备注
nextId	static long	是个静态 Long 变量，形成自增长序列	
id	long	该 SessionScope 实例化对象 id 标示，具有唯一性	
SqlMapClient	SqlMapClient	该 SessionScope 对象对应实现 SqlMapClient 接口的实例化对象，实际上是 SqlMapClientImpl 对象	
SqlMapExecutor	SqlMapExecutor	该 SessionScope 对象对应实现 SqlMapExecutor 接口的实例化对象，实际上是 SqlMapExecutorDelegate 对象	
SqlMapTxMgr	SqlMapTransactionManager	该 SessionScope 对象对应实现 SQLMapTransaction Manager 接口的实例化对象，实际上还是 SqlMapClientImpl 对象	
requestStackDepth	int	用于进行深度的判断	
transaction	Transaction	由 TransactionManager 调用，这是当前 SessionScope 处理的事务	
transactionState	TransactionState	由 TransactionManager 调用	
savedTransactionState	TransactionState	针对外部的事务处理。被 SqlMapExecutorDelegate 的 setUserProvidedTransaction 方法所用，表示当前 SessionScope 事务的状态	
inBatch	boolean	是否要进行批处理	
batch	Object	由 SqlExecutor 来进行调用进行批处理的对象	
commitRequired	boolean	是否提交标识	
preparedStatements	Map	按照 SQL 语句为主键，存放 Java.SQL.PreparedStatement 实例化对象的 Map	

2. StatementScope 类

StatementScope 主要针对每个 Statement 处理的状态，其主要属性列表和功能用途如表 7-12 所示。

表 7-12 StatementScope 主要属性列表和功能用途

属性名称	类型	功能和用途	备注
statement	MappedStatement	本 StatementScope 对应的 MappedStatement 对象	
sessionScope	SessionScope	针对 MappedStatement 对象的 session 状态信息	
parameterMap	ParameterMap	本 StatementScope 对应的 ParameterMap 对象	

续表

属性名称	类型	功能和用途	备注
resultMap	ResultMap	本 StatementScope 对应的 ResultMap 对象	
SQL	SQL	本 StatementScope 对应的 SQL 对象	
dynamicParameterMap	ParameterMap	动态 SQL 使用的 ParameterMap 对象	
dynamicSQL	String	本 StatementScope 对应的动态 SQL 语句的字符串	
resultSet	ResultSet	处理查询中的 N+1 问题	
uniqueKeys	Map	处理查询中的 N+1 问题	
rowDataFound	boolean	处理查询中的 N+1 问题，用于判断是否查询到数据	
currentNestedKey	String	处理查询中的 N+1 问题	

7.4 业务实现分析

在总体业务运行的介绍中已经对业务实现做了概要说明。下面就是把这个概要说明进行细化。

7.4.1 业务实现两个阶段的分析

外部调用接口层也是支持多种调用模式，比如查询、修改、事务操作等。对于每种业务的调用，按照业务运行的总体过程来看，都是从接口层到会话层，再到调度层，然后是映射层，最后是数据库操作层。这些层次的转移，基本上在前半部分都是相似的，调用的对象也是基本一样的。这样，为了讲解和表示的方便，可以把系统分为两个阶段来进行讲解。因为第一个阶段基本上都是一致的，主要的差别是在第二阶段的实现上。第一个阶段的序列图如图 7-13 所示。

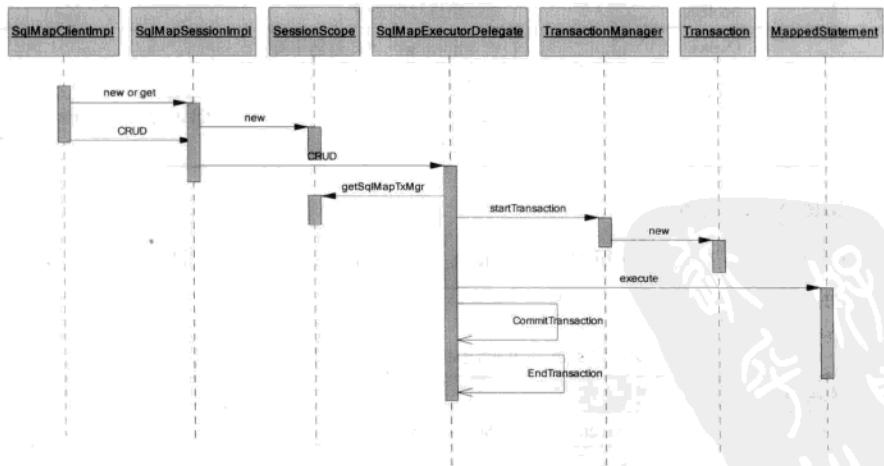


图 7-13 业务总体实现前半部分序列图

下一步就是细化后第二阶段具体的处理过程，同时进行源码实现分析。第二阶段由于不同的操作，其运行情况完全不一样，大致可以分为三种情况来讲解：第一种是查询业务，第二种是直接执行事务业务，第三种是开启事务后执行业务。

业务分析主要是按照事务的标准来进行，首先是划分事务标准，事务主要是 JDBC 事务、JTA 事务、外部事务和用户自定义数据库 Connection。其次是按照单事务操作和联合事务操作进行划分。最后是划分其中的 CUD 操作是否采用批处理。

JDBC 事务由 Connection 管理，也就是说，事务管理实际上是在 JDBC Connection 中实现的。事务周期限于 Connection 的生命周期。同样，对于基于 JDBC 的 iBATIS 事务管理机制而言，事务管理在 SqlMapClient 所依托的 JDBC Connection 中实现，事务周期限于 SqlMapClient 的生命周期。

7.4.2 查询类业务实现过程

iBATIS 的查询有几种模式，一种是直接查询，另一种是带传入参数的查询。查询后返回值也有几种类型，一种是直接返回一个对象，如 JavaBean。另一种是返回一个对象的列表 List，还有一种是返回对象的 Map。

查询基本上不涉及事务处理。外部主要的接口调用方法如下。

```
SqlMapClient.queryForObject(String id, Object parameterObject)
SqlMapClient.queryForObject(String id)
SqlMapClient.queryForObject(String id, Object parameterObject, Object result Object)

SqlMapClient.queryForList(String id, Object parameterObject)
SqlMapClient.queryForList(String id)
SqlMapClient.queryForList(String id, Object parameterObject, int skip, int max)
SqlMapClient.queryForList(String id, int skip, int max)
SqlMapClient.queryWithRowHandler(String id, Object parameterObject, RowHandler
rowHandler)
SqlMapClient.queryWithRowHandler(String id, RowHandler rowHandler)
SqlMapClient.queryForMap(String id, Object parameterObject, String keyProp)
SqlMapClient.queryForMap(String id, Object parameterObject, String keyProp, String
valueProp)
```

上述方法的实现，主要都是针对 SQLMap 映射文件的 select 节点来进行的。select 节点的信息主要包括 id、parameterMap、SqlText 和 resultMap。当然，也有简化模式，即 select 节点的信息只有 id、parameterClass、SqlText 和 resultClass，说明如下。

```
<select id="getPerson" parameterClass="int" resultClass="examples.domain.Person">
    SELECT PER_ID as id, PER_FIRST_NAME as firstName, PER_LAST_NAME as lastName,
    PER_BIRTH_DATE as birthDate, PER_WEIGHT_KG as weightInKilograms, PER_HEIGHT_
M as heightInMeters
    FROM PERSON
    WHERE PER_ID = #value#
</select>
```


而从数据库获得的程序代码如下：

```
...
SqlMapClient sqlMap = MyAppSqlMapConfig.getSqlMapInstance(); // as coded above
...
Integer personPk = new Integer(5);
Person person = (Person) sqlMap.queryForObject ("getPerson", personPk);
```

iBATIS 平台具体是如何实现的呢？

我们由第 6 章知道，iBATIS 平台在解析 SQLMap 映射文件的 select 节点时，会把节点内的信息转移到 SqlMapExecutorDelegate 对象的 HashMap 类型的 mappedStatements 变量中保存。其中 select 节点会被实例化成为一个 SelectStatement 对象。在这个 SelectStatement 对象中，有一个实现 SQL 接口的 SQL 变量，在 SQL 变量中有一个 Sqltext 的变量。

在这里要涉及 SqlMapExecutorDelegate 对象、SelectStatement 对象、StatementScope 对象、DefaultRowHandler 对象、Sql 接口实例化对象、RowHandlerCallback 对象、SqlExecutor 对象、PreparedStatement 对象、ParameterMap 对象、ParameterMapping 对象、ResultMap 对象、ResultMapping 对象等多个对象。

为了清晰地了解整个过程，在这里也分为两个过程来实现。第一个过程是 SelectStatement 对象进行处理后给 SqlExecutor 对象。第二个过程是 SqlExecutor 对象进行处理后返回值给 SelectStatement 对象。为了简化说明，把事务处理作为第一部分。

第一部分的序列如图 7-14 所示。

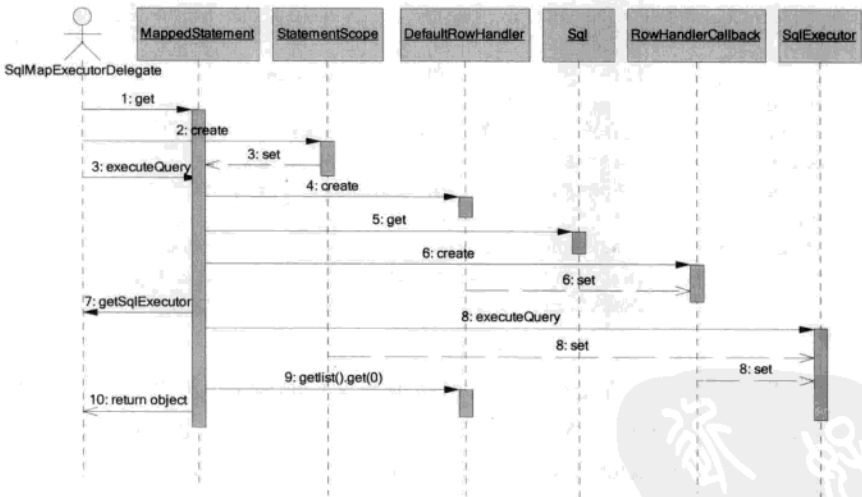


图 7-14 查询类业务实现第一部分序列图

实现步骤说明：

第 1 步骤：当外部调用 SqlMapExecutorDelegate 对象的 executeQuery 方法时，传入的参数主要是 MappedStatement 的 id 和 Sessionscope 对象。SqlMapExecutorDelegate 对象根

据 id 从 HashMap 类型的变量 mappedStatements 中获取 MappedStatement 实例化对象，如果是查询的话，一般都是 SelectStatement 对象。

第 2 步骤：SqlMapExecutorDelegate 对象用传入的 SessionScope 对象作为传入参数创建一个 StatementScope 对象。

第 3 步骤：SqlMapExecutorDelegate 对象用 SessionScope 变量作为参数，调用 MappedStatement 的 executeQuery 方法。

第 4 步骤：MappedStatement 对象创建一个 DefaultRowHandler 对象。

第 5 步骤：MappedStatement 对象获得当前 MappedStatement 对象实现 Sql 接口的 Sql 对象（如果是动态语言，则是 DynamicSql 对象，如果简单赋值，则是 SimpleDynamicSql 对象，默认情况下是 StaticSql 对象）。

第 6 步骤：MappedStatement 对象用 DefaultRowHandler 对象作为参数创建一个 RowHandlerCallback 对象。

第 7 步骤：MappedStatement 对象通过调用 SqlMapExecutorDelegate 对象的 getSqlExecutor 方法，获得 SqlExecutor 对象。

第 8 步骤：MappedStatement 对象以 StatementScope 对象和 RowHandlerCallback 对象为参数，调用 SqlExecutor 对象的 executeQuery 方法。

第 9 步骤：SqlExecutor 对象在执行 executeQuery 方法的时候 RowHandlerCallback 对象以及 RowHandlerCallback 对象内部的 DefaultRowHandler 对象进行处理。当执行方法完成后，MappedStatement 对象调用 RowHandlerCallback 对象内部 DefaultRowHandler 对象的 getList().get() 方法，获得一个 Object 对象。

第 10 步骤：把 Object 对象返回给 SqlMapExecutorDelegate 对象。而 Object 对象实际上就是查询所得到的 JavaBean。

上述 10 个步骤就是实现的查询的第一过程。在这个阶段中，主要是在 MappedStatement 对象进行程序的处理，还没有涉及 SqlExecutor 对象的具体操作。我们将在第二过程中详细描述 SqlExecutor 对象是如何获取参数、操作数据库，同时对获得数据集进行处理的。

程序在实现过程中，最终都是按照交到 MappedStatement 对象或 SelectStatement 对象的 queryForList 方法来处理的。事实上，SelectStatement 类是继承 MappedStatement 类，但基本上没有自己的方法。所以说，MappedStatement 对象和 SelectStatement 对象基本上是一致的。区别就在于用户在定义 SQLMap 的时候，有的喜欢用 Statement 作为节点，有的喜欢用 Select 而已。

不失一般性，我们可以以调用接口的 queryForObject 方法作为例子来说明，就是把 queryForList 方法返回的 List 的第一个对象返回即可。

```
public Object executeQueryForObject(StatementScope statementScope, Transaction
trans, Object parameterObject, Object resultObject) throws SQLException {
    try {
        Object object = null;

        DefaultRowHandler rowHandler = new DefaultRowHandler();
```

```

        executeQueryWithCallback(statementScope, trans.getConnection(), parameterObject,
resultObject,
        rowHandler, SqlExecutor.NO_SKIPPED_RESULTS, SqlExecutor.NO_MAXIMUM_
RESULTS);
        List list = rowHandler.getList();

        if (list.size() > 1) { throw new SQLException("Error: executeQueryForObject
returned too many results.");
        } else if (list.size() > 0) { object = list.get(0); }

        return object;
    } catch (TransactionException e) {
        throw new NestedSQLException("Error getting Connection from Transaction.
Cause: " + e, e);
    }
}

```

对于 List 和 Map 的处理基本相同，只是返回值的时候不同而已。在 iBATIS SQL Map 平台里有专门把 List 转化为 Map 的程序代码，代码如下。

```

public Map queryForMap(SessionScope sessionScope, String id, Object paramObject,
String keyProp, String valueProp) throws SQLException {
    Map map = new HashMap();
    List list = queryForList(sessionScope, id, paramObject);
    for (int i = 0, n = list.size(); i < n; i++) {
        Object object = list.get(i);
        Object key = PROBE.getObject(object, keyProp);
        Object value = null;
        if (valueProp == null) { value = object; }
        else { value = PROBE.getObject(object, valueProp); }
        map.put(key, value);
    }
    return map;
}

```

第二过程实现的序列图如图 7-15 所示。该过程要实现三个任务，而且这些任务还要按照一定的顺序来执行：（1）查询前准备条件和参数准备；（2）对数据库操作进行查询；（3）查询后对 Resultset 的处理。

第一个任务的工作是查询前的准备，主要是准备好数据库连接、查询语句和查询参数。

第二个任务是对数据库进行操作，直接执行 SQL 语句并返回 Resultset 内容。

第三个任务是把获得的 Resultset 内容按照定义的格式赋值给 ResultMap（如果定义了的话），如果没有定义 ResultMap，而是定义 JavaBean、或者 Map，或者 Dom 文档。那就把 Resultset 内的数据赋值给这些对象。

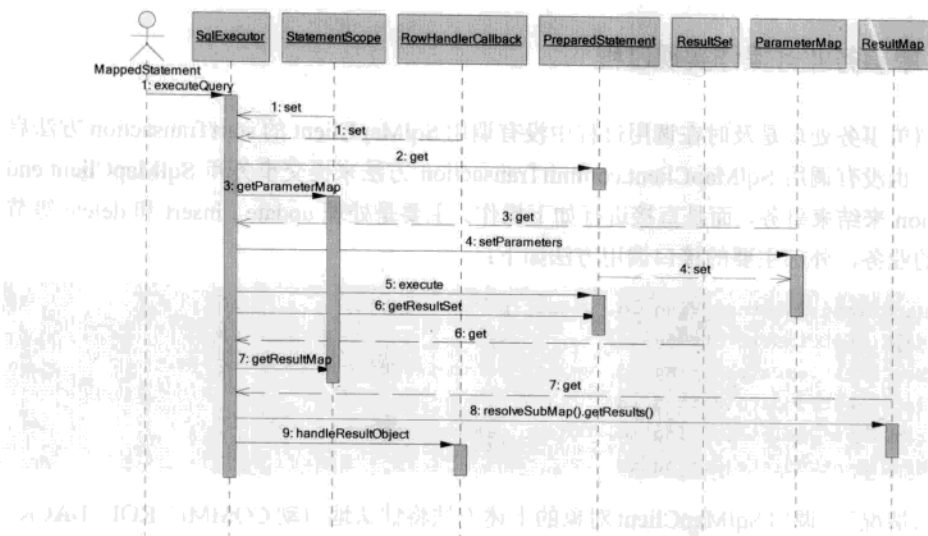


图 7-15 查询类业务实现第二部分序列图

实现步骤说明如下。

第 1 步骤: MappedStatement 对象以 StatementScope 对象和 RowHandlerCallback 对象为参数, 调用 SqlExecutor 对象的 executeQuery 方法。

第 2 步骤: SqlExecutor 对象通过 Sessionscope 对象获得数据库连接, 在这个数据库连接上创建一个 PreparedStatement 对象。

第 3 步骤: SqlExecutor 对象通过调用 StatementScope 对象的 getParameterMap 方法获得当前的 ParameterMap 对象。

第 4 步骤: SqlExecutor 对象以 PreparedStatement 对象为参数, 调用 ParameterMap 对象的 setParameters 方法, 实际上是对 PreparedStatement 对象进行参数赋值。

第 5 步骤: SqlExecutor 对象调用 PreparedStatement 对象的 execute 方法, 执行对数据库的查询操作。

第 6 步骤: 从 PreparedStatement 对象获得查询后得到 ResultSet 的结果。

第 7 步骤: SqlExecutor 对象通过调用 StatementScope 对象的 getResultMap 方法获得当前的 ResultMap 对象。

第 8 步骤: SqlExecutor 对象循环调用 ResultMap 对象的 resolveSubMap().getResults() 方法, 实现把 ResultSet 的内容都转化到 Object 对象。

第 9 步骤: SqlExecutor 对象循环调用 RowHandlerCallback 对象的 handleResultObject 方法, 把第 8 个步骤产生的 Object 对象赋值到 RowHandlerCallback 对象中 DefaultRowHandler 对象的 List。

当然这个过程还不包括一些特殊处理, 如缓存处理、动态语言支持等。

7.4.3 单事务业务操作实现过程

所谓单事务处理是及时在调用过程中没有调用 `SqlMapClient` 的 `startTransaction` 方法启动事务，也没有调用 `SqlMapClient.commitTransaction` 方法来提交事务和 `SqlMapClient.endTransaction` 来结束事务，而是直接进行如下操作。主要是处理 `update`、`insert` 和 `delete` 等节点形成的业务，外部主要的接口调用方法如下：

```
SqlMapClient.insert(String id, Object parameterObject)
SqlMapClient.insert(String id)
SqlMapClient.update(String id, Object parameterObject)
SqlMapClient.update(String id)
SqlMapClient.delete(String id, Object parameterObject)
SqlMapClient.delete(String id)
```

默认情况下，调用 `SqlMapClient` 对象的上述方法将默认地自动 `COMMIT/ROLLBACK`。这意味着每次调用上述方法都是一个独立的事务。

平台在 `SQLMap` 映射信息的时候，会解析 `update`、`insert` 和 `delete` 节点，并在 `SqlMapExecutorDelegate` 对象的 `HashMap` 类型的 `mappedStatements` 变量中保存。其中 `update` 节点会被实例化成为了一个 `UpdateStatement` 对象，`insert` 节点会被实例化成为了一个 `InsertStatement` 对象，`delete` 节点会被实例化成为了一个 `DeleteStatement` 对象。

这些对象的基本结构包括 `id`、`parameterClass` 和 `sql`，由于一般情况下没有返回值，所以没有 `resultClass`，其实现过程序列如图 7-16 所示。

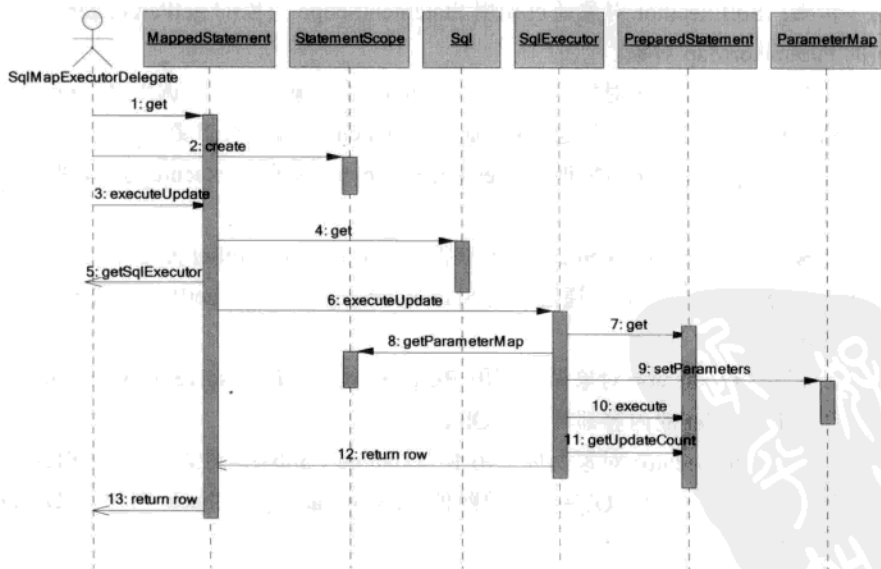


图 7-16 业务总体实现前半部分序列图

实现步骤说明:

第1步骤: 当外部调用 `SqlMapExecutorDelegate` 对象的 `executeUpdate` 方法, 传入的参数主要是 `MappedStatement` 的 `id` 和 `SessionScope` 对象。`SqlMapExecutorDelegate` 对象根据 `id` 从 `HashMap` 类型的变量 `mappedStatements` 中获取 `MappedStatement` 实例化对象。如果是删除操作, 为 `DeleteStatement` 对象; 如果是修改操作, 为 `UpdateStatement` 对象; 如果是新增, 为 `InsertStatement` 对象。

第2步骤: `SqlMapExecutorDelegate` 对象用传入的 `SessionScope` 对象作为传入参数, 创建一个 `StatementScope` 对象。

第3步骤: `SqlMapExecutorDelegate` 对象用 `SessionScope` 变量为参数, 调用 `MappedStatement` 对象的 `executeUpdate` 方法。

第4步骤: `MappedStatement` 对象获得当前 `MappedStatement` 对象的实现 `Sql` 接口的 `Sql` 对象 (如果是动态语言, 则是 `DynamicSql` 对象; 如果简单赋值, 则是 `SimpleDynamicSql` 对象; 一般情况下是 `StaticSql` 对象)。

第5步骤: `MappedStatement` 对象通过调用 `SqlMapExecutorDelegate` 对象的 `getSqlExecutor` 方法, 获得 `SqlExecutor` 对象。

第6步骤: `MappedStatement` 对象以 `StatementScope` 对象为参数, 调用 `SqlExecutor` 对象的 `executeUpdate` 方法。

第7步骤: `SqlExecutor` 对象通过 `StatementScope` 对象获得数据库连接, 在这个数据库连接上创建一个 `PreparedStatement` 对象。

第8步骤: `SqlExecutor` 对象通过调用 `StatementScope` 对象的 `getParameterMap` 方法, 获得当前的 `ParameterMap` 对象。

第9步骤: `SqlExecutor` 对象以 `PreparedStatement` 对象为参数, 调用 `ParameterMap` 对象的 `setParameters` 方法, 实际上是对 `PreparedStatement` 对象进行参数赋值。

第10步骤: `SqlExecutor` 对象调用 `PreparedStatement` 对象的 `execute` 方法, 执行对数据库的操作。

第11步骤: `SqlExecutor` 对象调用 `PreparedStatement` 对象的 `getUpdateCount` 方法, 获得操作后的返回值 `row`。

第12步骤: `SqlExecutor` 对象把 `row` 返回给 `MappedStatement` 对象。

第13步骤: `MappedStatement` 对象把 `row` 返回给 `SqlMapExecutorDelegate` 对象。

上述的处理主要是针对 `Update` 和 `Delete` 的操作。对于 `InsertStatement` 对象在处理时候要稍微复杂一点, 因为用户在插入数据的时候, 需要自动形成键值。很多数据库支持自动生成主键的数据类型。不过这通常 (并不总是) 是一个私有的特性。SQL Map 通过 `<insert>` 的子元素 `<selectKey>` 来支持自动生成的键值。它同时支持预生成 (如 Oracle) 和后生成两种类型 (如 MS-SQL Server)。

对于自动生成内码的 `insert` 的业务处理, 其序列图如图 7-17 所示。

其他步骤与单事务处理基本上一致, 也就是说, 是在 `SqlMapExecutorDelegate` 对象调

用 InsertStatement 对象之前，要创建一个 SelectKeyStatement 对象，然后自动生成的主键，并把这些作为参数传递到 InsertStatement 对象。关于 InsertStatement 对象如何具体生成主键，在后面进行详细描述。

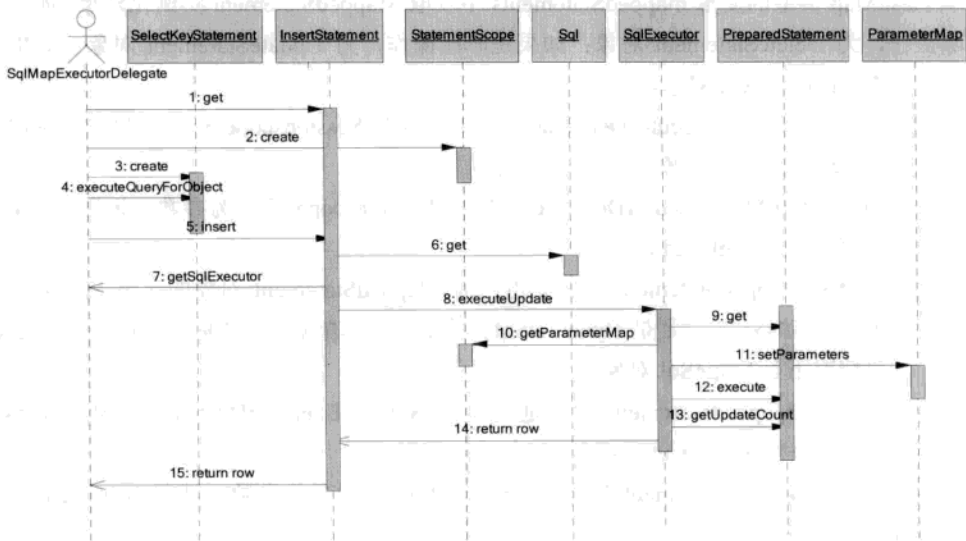


图 7-17 业务总体实现前半部分序列图

7.4.4 联合事务处理实现过程

联合事务比较复杂，在调用过程中首先要启动事务，然后进行业务处理，最后进行业务提交。如果提交不成功，还要进行回滚操作，最后还要清理事务现场。联合事务处理包括新增、修改、删除和查询操作，外部主要的接口调用方法如下。

```
SqlMapClient.insert(String id, Object parameterObject)
SqlMapClient.insert(String id)
SqlMapClient.update(String id, Object parameterObject)
SqlMapClient.update(String id)
SqlMapClient.delete(String id, Object parameterObject)
SqlMapClient.delete(String id)
SqlMapClient.queryForObject(String id, Object parameterObject)
SqlMapClient.queryForObject(String id)
SqlMapClient.queryForObject(String id, Object parameterObject, Object resultObject)

SqlMapClient.queryForList(String id, Object parameterObject)
SqlMapClient.queryForList(String id)
SqlMapClient.queryForList(String id, Object parameterObject, int skip, int max)
SqlMapClient.queryForList(String id, int skip, int max)
SqlMapClient.queryWithRowHandler(String id, Object parameterObject, RowHandler
rowHandler)
```



```

SqlMapClient.queryWithRowHandler(String id, RowHandler rowHandler)
SqlMapClient.queryForMap(String id, Object parameterObject, String keyProp)
SqlMapClient.queryForMap(String id, Object parameterObject, String keyProp, String
valueProp)

SqlMapClient.startTransaction()
SqlMapClient.startTransaction(int transactionIsolation)
SqlMapClient.commitTransaction()
SqlMapClient.endTransaction()

```

SqlMapClient 对象拥有让您定义事务范围的方法。使用下面 SqlMapClient 类的方法，可以开始、提交和/或回退事务。

```

public void startTransaction () throws SQLException
public void commitTransaction () throws SQLException
public void endTransaction () throws SQLException

```

开始一个事务，意味着您从连接池中得到一个链接，打开它并执行查询和更新 SQL 操作。使用事务处理的例子如下：

```

private Reader reader = new Resources.getResourceAsReader("com/ibatis/example/
sqlMapconfig.xml");
private SqlMapClient sqlMap = XmlSqlMapBuilder.buildSqlMap(reader);
public updateItemDescription (String itemId, String newDescription) throws
SQLException {
    try {
        sqlMap.startTransaction ();
        Item item = (Item) sqlMap.queryForObject ("getItem", itemId);
        item.setDescription (newDescription);
        sqlMap.update ("updateItem", item);
        sqlMap.commitTransaction ();
    } finally {
        sqlMap.endTransaction ();
    }
}

```

SqlMapClient 事务处理使用 Java 的 ThreadLocal 保存事务对象。这意味着在处理事务时，每个调用 startTransaction 的线程，将得到一个唯一的 Connection 对象。将一个 Connection 对象返回数据源（或关闭链接）唯一的方法是调用 commitTransaction 或 rollbackTransaction 方法。否则，会用光链接池中的链接并导致死锁。

SqlMapClient 事务是按照 Session 来进行管理的，即外部的 startTransaction、commitTransaction 或 rollbackTransaction 都是转化到唯一的 SqlMapSessionImpl 线程基础上的。其总体序列图如图 7-18 所示。

但是由于外部调用是分阶段来进行的，故可按照外部调用来进行详细说明。下面分别说明外部的 startTransaction、commitTransaction 或 rollbackTransaction 在内部是如何实现的。

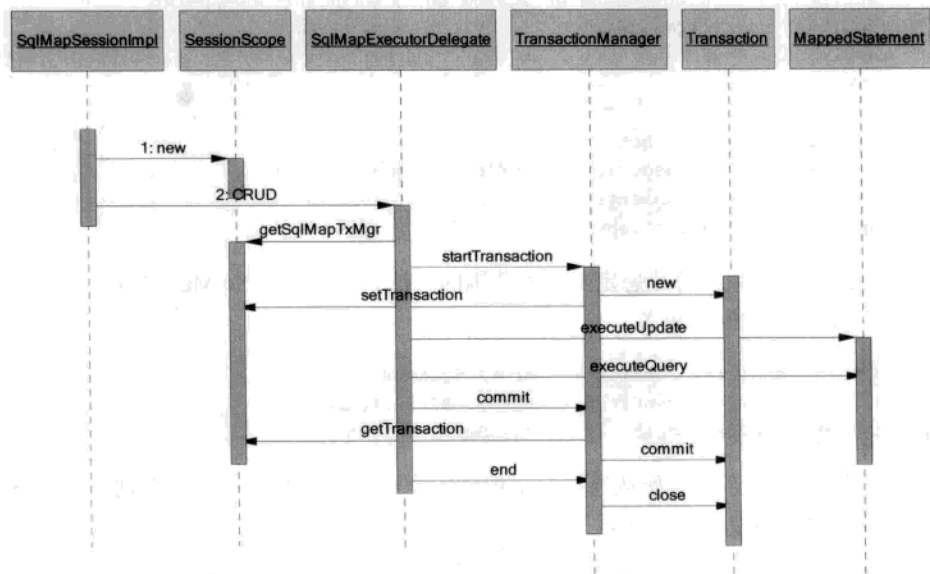


图 7-18 业务总体实现前半部分序列图

即上述的程序代码，分别按照四个阶段来说明。

```
private Reader reader = new Resources.getResourceAsReader("com/ibatis/example/
sqlMapconfig.xml");
private SqlMapClient sqlMap = XmlSqlMapBuilder.buildSqlMap(reader);
public updateItemDescription (String itemId, String newDescription) throws SQL
Exception {
    try {
        //第1阶段
        sqlMap.startTransaction ();//第1阶段
        //第2阶段
        Item item = (Item) sqlMap.queryForObject ("getItem", itemId);
        item.setDescription (newDescription);
        sqlMap.update ("updateItem", item);

        //第3阶段
        sqlMap.commitTransaction ();
    } finally {
        //第4阶段
        sqlMap.endTransaction ();
    }
}
```

第一阶段：客户端调用 SqlMapClient 的 startTransaction 方法

SqlMapClient 的 startTransaction 方法，转化为单一线程 SqlMapSessionImpl 的 startTransaction，其实现的序列图如图 7-19 所示。

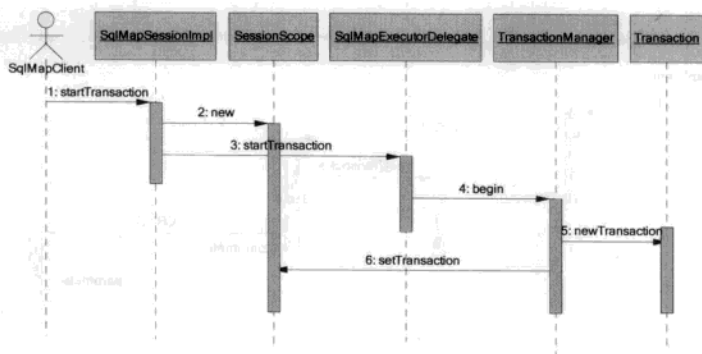


图 7-19 业务总体实现前半部分序列图

实现步骤说明:

第 1 步骤: 外部调用 SqlMapClient 的 startTransaction 方法, 将被 SqlMapClient 转化为调用 SqlMapSessionImpl 对象的 startTransaction 方法。

第 2 步骤: SqlMapSessionImpl 对象创建一个 SessionScope 对象, 把当前的状态信息都传递给 SessionScope 对象。

第 3 步骤: SqlMapSessionImpl 对象调用 SqlMapExecutorDelegate 对象的 start Transaction 方法。

第 4 步骤: SqlMapExecutorDelegate 对象在处理 startTransaction 方法时, 调用本身属性 TransactionManager 对象的 begin 方法。

第 5 步骤: 在 TransactionManager 对象的 begin 方法中, 创建一个实现 Transaction 接口的 Transaction 事务对象。

第 6 步骤: TransactionManager 对象把 Transaction 事务对象传递给 SessionScope 对象。

上述步骤就是外部的 SqlMapClient 的 startTransaction 方法在平台内容的实现过程。第 1 阶段产生了 Transaction 事务对象, 并在 SessionScope 对象中表明该事务已经开始启动。

第二阶段: 客户端调用 SqlMapClient 的 CRUD 方法

在客户端的代码开始调用 SqlMapClient 的 CRUD 方法, 这些方法包括 queryForObject、queryForList、queryForMap、insert、update、delete 等方法, 其序列图如图 7-20 所示。

实现步骤说明:

第 1 步骤: 外部调用 SqlMapClient 的 CRUD 方法, 这些方法将被 SqlMapClient 转化为调用 SqlMapSessionImpl 对象的 queryForObject、queryForList、queryForMap、insert、update、delete 等方法。

第 2 步骤: SqlMapSessionImpl 对象把调用自己的 queryForObject、queryForList、queryForMap、insert、update、delete 转化为调用 SqlMapExecutorDelegate 对象的 queryForObject、queryForList、queryForMap、insert、update、delete 方法。。

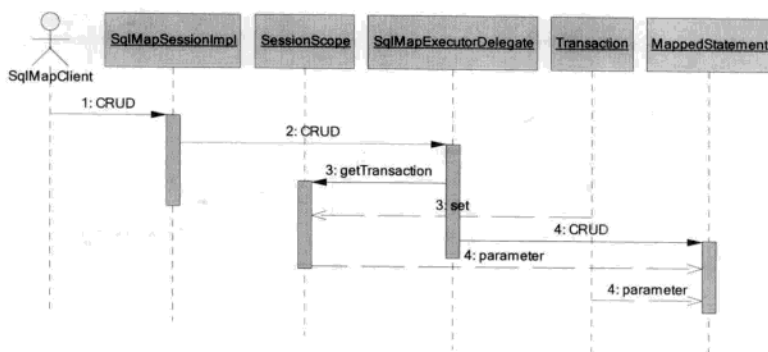


图 7-20 业务总体实现前半部分序列图

第 3 步骤: SqlMapExecutorDelegate 对象调用 SessionScope 的 getTransaction 方法, 通过 startTransaction 方法可获得 Transaction 事务对象。

第 4 步骤: SqlMapExecutorDelegate 对象在传入 SessionScope 对象和 Transaction 对象为参数时, 调用 MappedStatement 对象的 executeUpdate、executeQueryForList、executeQueryForObject、executeQueryWithRowHandler 方法。

但是由于当前存在一个实例化的 Transaction 事务对象, 步骤到此结束, 不进行提交操作。但是第 2 阶段主要是对数据库进行的 excute 操作。

第三个阶段: 客户端调用 SqlMapClient 的 commitTransaction 方法

下一步就准备开始进行事务提交的处理, 其序列图如图 7-21 所示。

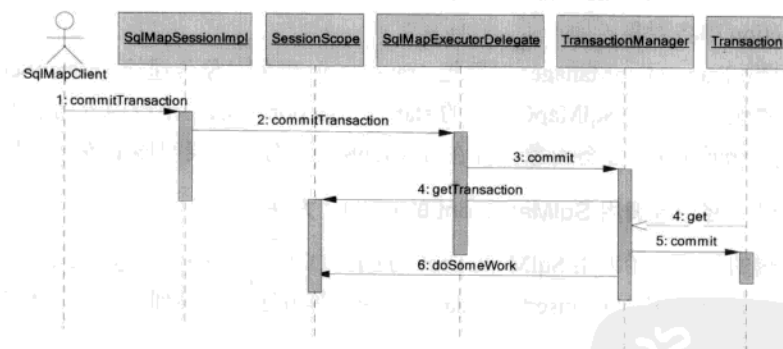


图 7-21 业务总体实现前半部分序列图

实现步骤说明。

第 1 步骤: 外部调用 SqlMapClient 的 commitTransaction 方法将被 SqlMapClient 转化为调用 SqlMapSessionImpl 对象的 commitTransaction 方法。

第 2 步骤: SqlMapSessionImpl 对象调用 SqlMapExecutorDelegate 对象的 commitTransaction 方法。

第 3 步骤: SqlMapExecutorDelegate 对象调用 TransactionManager 对象的 commit 方法。

第 4 步骤: TransactionManager 对象调用 SessionScope 对象的 getTransaction 方法, 通过 startTransaction 方法可获得 Transaction 事务对象。。

第 5 步骤: TransactionManager 对象调用 Transaction 事务对象的 commit 方法。

第 6 步骤: TransactionManager 对象对 SessionScope 对象做一些善后处理, 主要是修改事务状态等。

第 3 阶段主要是对 Transaction 事务对象进行 commit 操作, 也就是对数据库进行 commit 操作。

第四个阶段: 客户端调用 SqlMapClient 的 endTransaction 方法

第四个阶段是最后来结束事务, 完成一次事务操作的管理, 其序列图如图 7-22 所示。

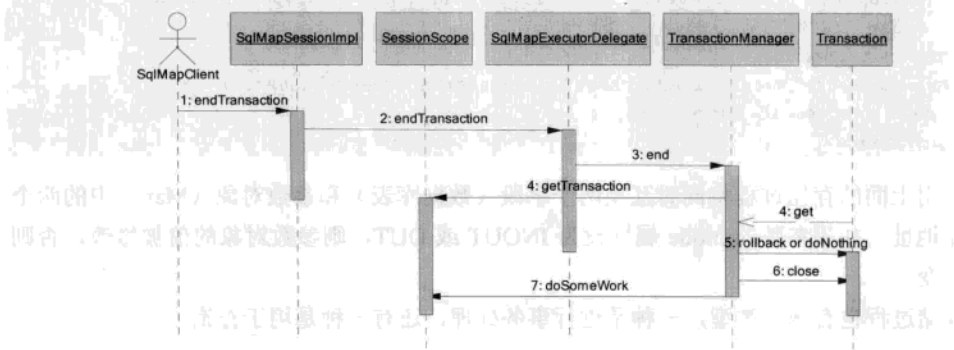


图 7-22 业务总体实现前半部分序列图

实现步骤说明。

第 1 步骤: 外部调用 SqlMapClient 的 endTransaction 方法将被 SqlMapClient 转化为调用 SqlMapSessionImpl 对象的 endTransaction 方法。

第 2 步骤: SqlMapSessionImpl 对象调用 SqlMapExecutorDelegate 对象的 endTransaction 方法。

第 3 步骤: SqlMapExecutorDelegate 对象调用 TransactionManager 对象的 end 方法。

第 4 步骤: TransactionManager 对象调用 SessionScope 对象的 getTransaction 方法, 通过 startTransaction 方法获得 Transaction 事务对象。

第 5 步骤: TransactionManager 对象调用 Transaction 事务对象的 rollback 方法或者什么都不用做。

第 6 步骤: TransactionManager 对象调用 Transaction 事务对象的 close 方法。

第 7 步骤: TransactionManager 对象对 SessionScope 对象做一些善后处理, 主要是删掉事务对象等。

第 4 阶段主要是销毁 Transaction 事务对象和改变 SessionScope 对象中表明该事务状

态，回到了最初没有进行 `startTransaction` 方法的现场环境中。

关于 Transaction 如何实现 `new`、`commit` 和 `rollback` 的问题，我们将在本书 7.4.7 节“全局 JTA 事务的处理”中进行详细讲解。

7.4.5 存储过程的处理

SQL Map 通过 `<procedure>` 元素支持存储过程。下面的例子说明如何使用具有输出参数的存储过程。

```
<parameterMap id="swapParameters" class="map" >
  <parameter property="email1" jdbcType="VARCHAR" javaType="java.lang.String"
    mode= "INOUT"/>
  <parameter property="email2" jdbcType="VARCHAR" javaType="java.lang.String"
    mode= "INOUT"/>
</parameterMap>
<procedure id="swapEmailAddresses" parameterMap="swapParameters" >
  {call swap_email_address (?, ?)}
</procedure>
```

调用上面的存储过程将同时互换两个字段（数据库表）和参数对象（Map）中的两个 E-mail 地址。如果参数的 `mode` 属性设为 `INOUT` 或 `OUT`，则参数对象的值被修改，否则保持不变。

存储过程也有两种类型，一种是进行事务处理，还有一种是用于查询。

1. Update 处理

Update 处理主要是那些基于增加、修改和删除操作的存储过程。在这个过程中，没有数据集的生成，其序列操作顺序如图 7-23 所示。

实现步骤说明。

第 1 步骤：当外部调用 `SqlMapExecutorDelegate` 对象的 `executeUpdate` 方法，传入的参数主要是 `ProcedureStatement` 的 `id` 和 `Sessionscope` 对象。`SqlMapExecutorDelegate` 对象根据 `id` 从 `HashMap` 类型的变量 `mappedStatements` 中获取 `ProcedureStatement` 实例化对象。

第 2 步骤：`SqlMapExecutorDelegate` 对象用传入的 `Sessionscope` 对象为传入参数创建一个 `StatementScope` 对象。

第 3 步骤：`SqlMapExecutorDelegate` 对象用 `Sessionscope` 变量作为参数，调用 `ProcedureStatement` 对象的 `executeUpdate` 方法。

第 4 步骤：`ProcedureStatement` 对象获得当前 `ProcedureStatement` 对象实现 `Sql` 接口的 `Sql` 对象（如果是动态语言，则是 `DynamicSql` 对象，如果简单赋值，则是 `SimpleDynamicSql` 对象，一般情况下是 `StaticSql` 对象）。

第 5 步骤：`ProcedureStatement` 对象通过调用 `SqlMapExecutorDelegate` 对象的 `getSqlExecutor` 方法，获得 `SqlExecutor` 对象。

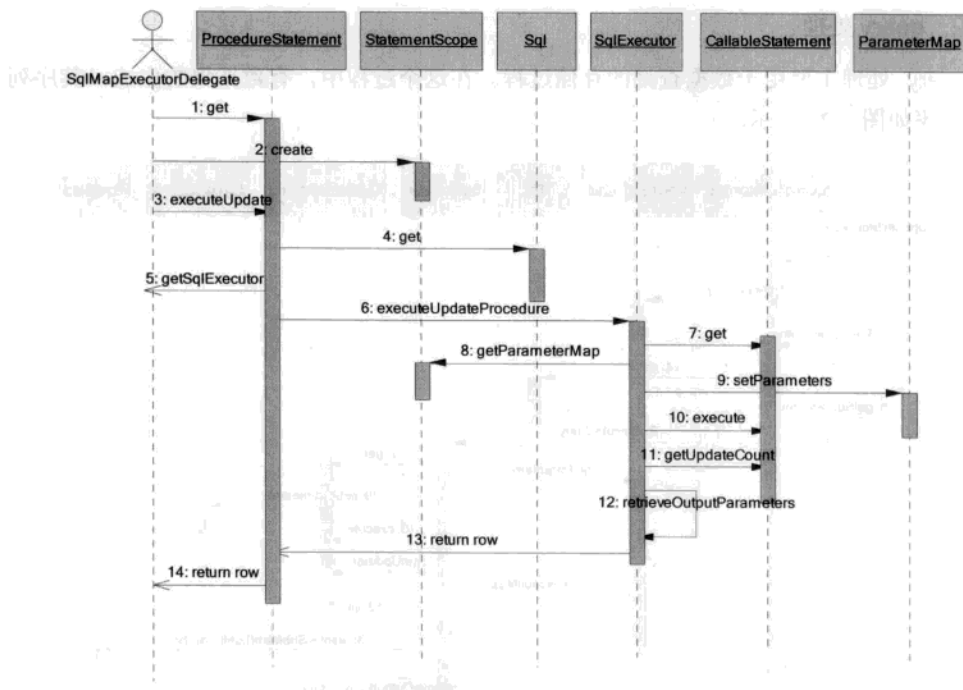


图 7-23 业务总体实现前半部分序列图

第 6 步骤: ProcedureStatement 对象以 StatementScope 对象为参数, 调用 SqlExecutor 对象的 executeUpdateProcedure 方法。

第 7 步骤: SqlExecutor 对象通过 StatementScope 对象获得数据库连接, 在这个数据库连接上创建一个 PreparedStatement 对象。

第 8 步骤: SqlExecutor 对象通过调用 StatementScope 对象的 getParameterMap 方法获得当前的 ParameterMap 对象。

第 9 步骤: SqlExecutor 对象以 PreparedStatement 对象为参数, 调用 ParameterMap 对象的 setParameters 方法, 实际上是对 PreparedStatement 对象进行参数赋值。

第 10 步骤: SqlExecutor 对象调用 PreparedStatement 对象的 execute 方法, 执行对数据库的操作。

第 11 步骤: SqlExecutor 对象调用 PreparedStatement 对象的 getUpdateCount 方法, 获得操作后的返回值 row。

第 12 步骤: SqlExecutor 对象调用自身的 retrieveOutputParameters 方法, 实现存储过程的输出变量。

第 13 步骤: SqlExecutor 对象把 row 返回给 ProcedureStatement 对象。

第 14 步骤: ProcedureStatement 对象把 row 返回给 SqlMapExecutorDelegate 对象。

2. query 处理

query 处理主要用于数据查询的存储过程。在这个过程中，有数据集的生成，其序列操作顺序如图 7-24 所示。

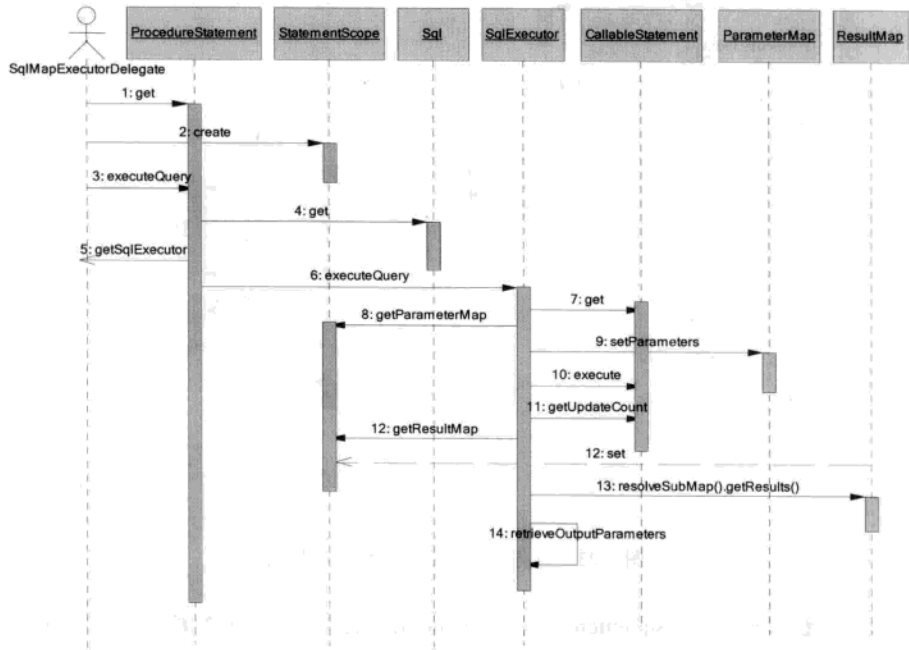


图 7-24 业务总体实现前半部分序列图

实现步骤说明。

第 1 步骤：当外部调用 SqlMapExecutorDelegate 对象的 executeUpdate 方法时，传入的参数主要是 ProcedureStatement 的 id 和 Sessionscope 对象。SqlMapExecutorDelegate 对象根据 id 从 HashMap 类型的变量 mappedStatements 中获取 ProcedureStatement 实例化对象。

第 2 步骤：SqlMapExecutorDelegate 对象用传入的 Sessionscope 对象作为传入参数创建一个 StatementScope 对象。

第 3 步骤：SqlMapExecutorDelegate 对象用 Sessionscope 变量作为参数，调用 ProcedureStatement 对象的 executeQuery 方法。

第 4 步骤：ProcedureStatement 对象获得当前 ProcedureStatement 对象实现 Sql 接口的 Sql 对象（如果是动态语言，则是 DynamicSql 对象；如果简单赋值，则是 SimpleDynamicSql 对象；一般情况下是 StaticSql 对象）。

第 5 步骤：ProcedureStatement 对象通过调用 SqlMapExecutorDelegate 对象的 getSqlExecutor 方法，获得 SqlExecutor 对象。

第 6 步骤：ProcedureStatement 对象以 StatementScope 对象作为参数，调用 SqlExecutor

对象的 `executeQueryProcedure` 方法。

第 7 步骤: `SqlExecutor` 对象通过 `StatementScope` 对象获得数据库连接, 在这个数据库连接上创建一个 `PreparedStatement` 对象。

第 8 步骤: `SqlExecutor` 对象通过调用 `StatementScope` 对象的 `getParameterMap` 方法获得当前的 `ParameterMap` 对象。

第 9 步骤: `SqlExecutor` 对象以 `PreparedStatement` 对象为参数, 调用 `ParameterMap` 对象的 `setParameters` 方法, 实际上是对 `PreparedStatement` 对象进行参数赋值。

第 10 步骤: `SqlExecutor` 对象调用 `PreparedStatement` 对象的 `execute` 方法, 执行对数据库的操作。

第 11 步骤: `SqlExecutor` 对象调用 `PreparedStatement` 对象的 `getUpdateCount` 方法, 获得操作后的返回值 `row`。

第 12 步骤: `SqlExecutor` 对象通过调用 `StatementScope` 对象的 `getResultMap` 方法, 获得当前的 `ResultMap` 对象。

第 13 步骤: `SqlExecutor` 对象调用 `ResultMap` 对象的 `resolveSubMap().getResults()`, 把 `resultset` 的数值转移给 `ResultMap` 对象对应的 `ClassObject`。

第 14 步骤: `SqlExecutor` 对象调用自身的 `retrieveOutputParameters` 方法, 实现存储过程的输出变量。

7.4.6 批处理及其实现

如果要执行很多非查询 (`insert/update/delete`) 的语句, 开发人员可能将它们作为一个批处理来执行, 以减少网络通信流量, 并让 `JDBC Driver` 进行优化 (例如压缩)。SQL Map API 使用批处理很简单, 可以使用两个简单的方法划分批处理的边界。

```
sqlMap.startBatch();
//...execute statements in between
sqlMap.executeBatch();
```

当调用 `endBatch` 方法时, 所有的批处理语句将通过 `JDBC Driver` 来执行。

对于批处理过程, 也要分为三个阶段。第 1 个阶段就是 `startBatch` 阶段, 告诉平台要开始进入批处理业务。第 2 个阶段是进行正常的业务操作, 包括 `insert`、`update`、`delete` 等方法的调用, 事实上, 这些操作只是进入了准备阶段, 并没有真正地对数据库进行操作。第 3 个阶段是 `executeBatch` 阶段, 就是通知平台把第 2 阶段的业务操作统一地进行批处理, 在这个阶段, 就是把第 2 阶段输入的全部操作都转化为 SQL 语句, 进行统一的数据库批处理操作并提交。为了描述方便, 我们把所有这三个阶段都放到一个序列图中进行讲解说明。

iBatis 平台批处理操作如图 7-25 所示。

`SqlMapClient` 调用 `startBatch` 方法, `SqlMapClient` 会创建或者获得当前线程的 `SqlMapSessionImpl` 对象, 然后调用 `SqlMapSessionImpl` 对象的 `startBatch` 方法。

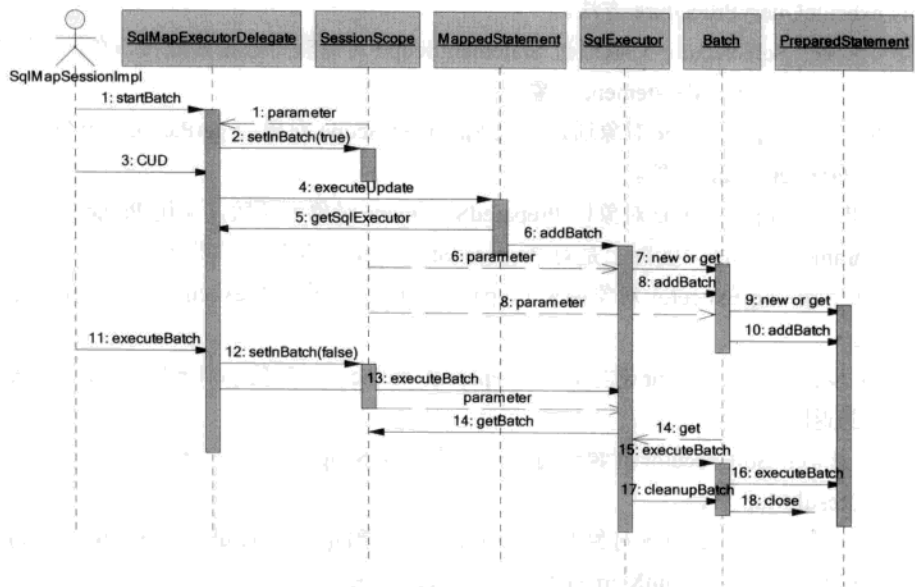


图 7-25 业务总体实现前半部分序列图

实现步骤说明：

第 1 个阶段——准备阶段。

第 1 步骤：在外部调用 SqlMapSessionImpl 对象的 startBatch 方法，SqlMapSessionImpl 对象转而以它的属性 SessionScope 对象为参数，调用 SqlMapExecutorDelegate 对象的 startBatch 方法。

第 2 步骤：SqlMapExecutorDelegate 对象调用 SessionScope 对象的 setInBatch 方法，把当前的 Session 状态赋值为 True，即进入了批处理工作状态。

第 2 个阶段开始——形成批处理阶段。

第 3 步骤：当外部对 SqlMapSessionImpl 对象执行 CUD 操作，即 insert、update、delete 等方法时。由 SqlMapSessionImpl 对象转移到调用 SqlMapExecutorDelegate 对象的 CUD 操作，即 insert、update、delete 等方法。

第 4 步骤：SqlMapExecutorDelegate 对象以 SessionScope 对象为基础创建或获取 StatementScope 对象，并把 SessionScope 对象的状态信息转移给 StatementScope 对象。对于调用 SqlMapExecutorDelegate 对象 CUD 操作，都会被 SqlMapExecutorDelegate 以 StatementScope 对象为参数调用 MappedStatement 对象的 executeUpdate 方法。

第 5 步骤：MappedStatement 对象调用 SqlMapExecutorDelegate 对象的 getSqlExecutor 方法，获得 SqlExecutor 对象。

第 6 步骤：MappedStatement 对象调用 SqlExecutor 对象的 addBatch 方法，并把带有 SessionScope 对象的 StatementScope 对象作为参数传递过去。

第 7 步骤: SqlExecutor 对象根据传递过来的 StatementScope 对象, 创建或者获得一个 Batch 对象。

第 8 步骤: SqlExecutor 对象调用 Batch 对象 addBatch 方法, 把要操作的语句都放在 Batch 对象的 List 属性 statementList 中。

第 9 步骤: Batch 对象创建或者获得一个 PreparedStatement 对象。

第 10 步骤: Batch 对象调用 PreparedStatement 对象的 addBatch 方法。

第 3 个阶段——执行阶段

第 11 步骤: 当外部对 SqlMapSessionImpl 对象执行 executeBatch 操作时, 由 SqlMapSessionImpl 对象转移到调用 SqlMapExecutorDelegate 对象的 executeBatch 方法。

第 12 步骤: SqlMapExecutorDelegate 对象调用 SessionScope 对象的 setInBatch 方法, 把当前的 Session 状态赋值为 False, 即进入了批处理工作完成状态。

第 13 步骤: SqlMapExecutorDelegate 对象调用 SqlExecutor 对象的 executeBatch 方法。

第 14 步骤: SqlExecutor 对象调用 SessionScope 对象的 getBatch 方法, 获得 Batch 对象。

第 15 步骤: SqlExecutor 对象调用 Batch 对象的 executeBatch 方法。

第 16 步骤: Batch 对象调用 PreparedStatement 对象的 executeBatch 方法, 执行完毕后返回。

第 17 步骤: SqlExecutor 对象调用 Batch 对象的 cleanupBatch 方法, 清除批处理现场和状态信息。

7.4.7 全局 JTA 事务的处理

上面所描述都是基于 JDBC 的事务实现, 如果要涉及多个事务的处理, 这就要求通过 JTA 来实现。SQL Map 框架支持全局事务。全局事务也叫分布式事务, 它可以允许在同一事务中更新多个数据库 (或其他符合 JTA 规范的资源), 即同时成功或失败。而且这些事务都是在 SQL Map 做好了连接配置的事务。如果一个事务要涉及外部, 或者是事务对象是由外部引入时, SQL Map 框架也可以管理全局事务。要支持受管理的全局事务, 必须在 SQL Map 配置文件中设定 <transactionManager> 的 type 属性为 JTA, 并设定 “UserTransaction” 属性为 JNDI 的全名, 以使 SqlMapClient 实例能找到 UserTransaction 对象。

JTA 事务管理则由 JTA 容器实现, JTA 容器对当前加入事务的众多 Connection 进行调度, 实现其事务性要求。JTA 的事务周期可横跨多个 JDBC Connection 生命周期。同样, 对于基于 JTA 事务的 iBATIS 而言, JTA 事务可横跨多个 SqlMapClient。

在应用层面上, 调用 JTA 和调用 JDBC 的实现基本上是相同的。只是在配置文件上有不同。在 JTA 事务管理中, 可以支持有多个 SqlMap 配置文件的情况, 多个配置文件可以采用同一个 JTA 事务管理器。

但对于容器管理的资源, 需要像下面的例子一样配置, 让它能和全局事务一起工作:

```
<transactionManager type="JTA" >
```

```

<property name="UserTransaction" value="java:/ctx/con/UserTransaction"/>
<dataSource type="JNDI">
    <property name="DataSource" value="java:comp/env/jdbc/jpetstore"/>
</dataSource>
</transactionManager>

```

注意，UserTransaction 属性指向 UserTransaction 实例所在的 JNDI 位置。JTA 事务管理需要它，以使 SQL Map 能够参与涉及其他数据库和事务资源的范围更大的事务。

例如在下面示例中，同时将 user 对象更新到两个不同的数据库：

```

User user = new User();
user.setId(new Integer(1));
user.setName("Erica");
user.setSex(new Integer(0));
sqlMap1 = xmlBuilder.buildSqlMap(db1Config);
sqlMap2 = xmlBuilder.buildSqlMap(db2Config);
try{
    sqlMap1.startTransaction();
    sqlMap1.update("User.updateUser", user);
    sqlMap2.update("User.updateUser", user);
    sqlMap1.commitTransaction();
}finally{
    sqlMap1.endTransaction();
}

```

上面的代码中，两个针对不同数据库的 SqlMapClient 实例，在同一个 JTA 事务中对 user 对象所对应的数据库记录进行更新。外层的 sqlMap1 启动了一个全局事务，此事务将涵盖本线程内 commitTransaction 之前的所有数据库操作。只要其间发生了异常，则整个事务都将被回滚。

7.4.8 全局外部事务的处理

上面所描述的事务都是在 SqlMap 做好了数据库连接配置的事务。如果一个事务要涉及外部，或者是事务对象是由外部引入，这就需要进行外部事务的处理。

设定<transactionManager>的 type 属性为 EXTERNAL。使用外部管理的全局事务。

SQL Map 事务控制方法变得有些多余，因为事务的开始、提交和回退都是由外部的事务管理器来控制的。但是，使用 SqlMapClient 的 startTransaction、commitTransaction 或 rollbackTransaction 来划分事务范围，对提高性能是有帮助的。继续使用这些方法，可以保持编程规范的一致性。由于把事务接口都暴露出来，所以开发的时候可以直接调用事务方法来进行事务提交，如下面所示。

对于外部事务管理，我们需要在配置文件中进行如下设定。

```

<transactionManager type="EXTERNAL">
    <dataSource type="JNDI">
        <property name="DataSource" value="java:comp/env/jdbc/jpetstore"/>
    </dataSource>
</transactionManager>

```

```
</dataSource>
</transactionManager>
```

下面是一个外部事务管理的典型示例：

```
.....
sqlMap1 = xmlBuilder.buildSqlMap(db1Config);
sqlMap1.update("User.updateUser", user);
```

此时，在 iBATIS 平台只是启动了数据库链接，但不负责提交、回滚等操作。具体提交和回滚等操作交到外部的容器（如 EJB）等来进行管理。同时，iBATIS 平台支持关闭数据库链接。当然，这种模式也支持没有事务管理的数据库。

还有一点要说明的是，可以通过 `startTransaction`、`commitTransaction` 或 `rollbackTransaction` 等方法来划分事务隔离范围。

7.4.9 用户自定义数据库 Connection 处理

上面所有内容都是在 SQL Map 做好了链接配置的事务，但平台也支持那种不用定义事务，只是在应用中直接调用数据库链接的处理。在这个环节中，可以省略到 SQL Map 的事务管理器。但是外部自定义 Connection 不支持 `startTransaction` 方法、`commitTransaction` 方法和 `endTransaction` 方法，同时，外部自定义 Connection 也不支持事务隔离，其实现的序列如图 7-26 所示。

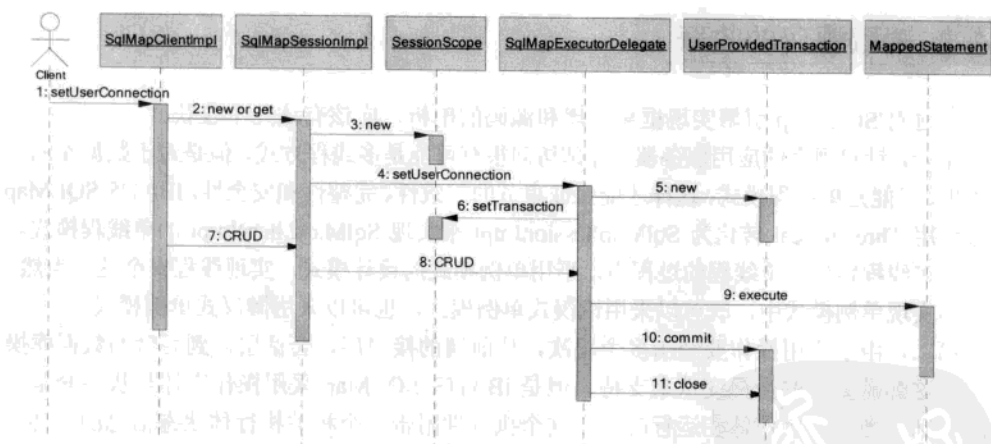


图 7-26 业务总体实现前半部分序列图

实现步骤说明：

第 1 阶段——设置数据库 Connection

第 1 步骤：外部建立好数据库 Connection，然后调用 `SqlMapClientImpl` 对象的 `setUserConnection` 方法，把数据库 Connection 作为参数传递进来。

第2步骤: SqlMapClientImpl 对象创建或者获得本地线程的 SqlMapSessionImpl 对象。

第3步骤: SqlMapSessionImpl 对象创建一个 SessionScope 对象。

第4步骤: SqlMapSessionImpl 对象调用 SqlMapExecutorDelegate 对象的 setUserConnection 方法, 把数据库 Connection 作为参数传递进来。

第5步骤: SqlMapExecutorDelegate 对象用获得的数据库 Connection 创建一个 UserProvidedTransaction 实例化对象。

第6步骤: SqlMapExecutorDelegate 对象调用 SessionScope 对象的 setTransaction 方法, 把创建的 UserProvidedTransaction 对象放在 SessionScope 对象变量中。

第2阶段——进行 CRUD 操作

第7步骤: SqlMapClientImpl 对象把外部的 CRUD 操作方法转换给 SqlMapSessionImpl 对象的 CRUD 操作方法。

第8步骤: SqlMapSessionImpl 对象把 CRUD 操作方法转换给 SqlMapExecutorDelegate 对象的 CRUD 操作方法。

第9步骤: SqlMapExecutorDelegate 对象把 CRUD 操作方法转换给 MappedStatement 对象的 execute 方法。

第10步骤: SqlMapExecutorDelegate 对象调用 UserProvidedTransaction 对象的 commit 方法, 进行事务提交。

第11步骤: SqlMapExecutorDelegate 对象调用 UserProvidedTransaction 对象的 close 方法, 进行清理事务现场。

7.5 读取源码的收获

通过对 SQL Map 引擎实现框架实现和源码的剖析, 应该有这几个收获。

第一, 针对现在的应用服务器, 外部访问很有可能是多线程方式, 但是对于数据库的事务处理, 只能是单线程模式, 这样才能保证事务的一致性、完整性和安全性。iBATIS SQLMap 通过使用 ThreadLocal 转化为 SqlMapSessionImpl 来实现 SqlMapClientImpl 的单线程模式。在这种多线程转化为单线程的过程中, 采用单例模式的设计模式, 实现线程安全性。当然, 在具体实现单例模式中, 既可以采用饿汉式单例模式, 也可以采用懒汉式单例模式。

第二, 由于应用操作要经历多个层次, 从前端的接口层, 会话层, 到后台的数据库操作层, 这都需要一些全局变量来支持, 但是 iBATIS SQLMap 采用操作实体与状态变量分离的原则。当一个操作体在运行时, 对这个执行性附带一个相关执行体状态信息的实体。而对外部操作时候, 主要用状态实体来进行传递, 这样才能达到更好的效果。

第三, 采用大集中的中心调度模式。对于复杂业务, 有的时候要采用统一的接口来进行调度和管理。这种汇总的中心就是引擎中心。也可以在引擎中心前端再封装一个门面接口, 这样应用系统的条理就更加清晰。

第四, 采用分层结构模式, 基本上按照接口层、会话层、调度层、映射层和数据库操作层组成。各个层次各司其职, 这样能统一地完成复杂的功能。

第 8 章

SQL Map 数据库处理

本章内容:

1. 说明 SQL Map 中的事务管理, 包括简单介绍一下 Java 事务, 事务管理的设计模型和事务管理的实现。
2. 说明 SQL Map 中事务管理的四种策略模式, 分别为 JDBC、JTA、EXTERNAL、用户事务等。
3. SQL Map 的 DataSource 策略, 包括 SIMPLE 策略、DBCP 策略、JNDI 策略等。
4. SQL Map 自定义 DataSource 实现, 主要是 SimpleDataSource 设计和实现。
5. 将 SQL Map 的 DataSource 扩展为支持 C3p0。
6. SQL Map 如何进行批处理。
7. SQL Map 事务隔离的实现。
8. 介绍 SQL Map 事务状态的实现。

SQL Map 作为一个 ORM 模型, 与数据库打交道是不可避免的。Java 与数据库交互无一例外地要涉及 JDBC、事务等概念。SQL Map 也不例外, SQL Map 数据库相关的内容主要是如何获得数据库连接、如何进行事务管理和如何进行批处理操作。其中获得数据库连接要涉及数据库池, 这里要讲解到 DataSource。对于数据库事务管理部分要涉及 SQL Map 事务策略。批处理的实现要涉及 JDBC 的批处理内容。

SQL Map 数据库处理覆盖的包主要有 `com.ibatis.sqlmap.engine.datasource` 包和 `com.ibatis.sqlmap.engine.transaction` 包。当然那也可以引申到其他实现的包内。涉及的组件包含事务管理的 `transaction` 组件、数据源 `datasource` 组件。数据库处理部分包括事务处理和数据库操作。主要的类和接口有 `TransactionManager` 类、`TransactionConfig` 接口、`TransactionState` 类、`Transaction` 接口等。`TransactionManager` 类依赖 `Transaction` 接口, 作为系统事务管理的总调度。而具体实现 `Transaction` 接口的实现类是进行事务管理的执行体。

8.1 SQL Map 的 transactionManager

8.1.1 Java 事务简介

由数据库事务我们可以知道，事务是作为工作的单一逻辑单元而执行的一系列操作，在执行过程中要么所有的操作都执行，要么都不执行。事务必须服从 ISO/IEC 所制定的 ACID 原则。ACID 是原子性 (atomicity)、一致性 (consistency)、隔离性 (isolation) 和持久性 (durability) 的缩写。事务的原子性表示事务执行过程中的任何失败都将导致事务所做的任何修改失效。一致性表示当事务执行失败时，所有被该事务影响的数据都应该恢复到事务执行前的状态。隔离性表示在事务执行过程中对数据的修改，在事务提交之前对其他事务不可见。持久性表示已提交的数据在事务执行失败时，数据的状态都应该正确。分布式事务处理是指事务的参与者、支持事务的服务器、资源服务器以及事务管理器分别位于不同的分布式系统的不同节点之上，事务的实现由这些分布式组件完成。一个事务处理可能涉及多个资源的参与，这些资源可以是数据库和面向消息的中间件产品。

Java EE 开发者有两个事务管理的选择：全局事务和本地事务。全局事务由应用服务器管理，使用 JTA，可以用于多个事务性的资源。局部事务是和资源相关的，主要通过 JDBC 就可以实现。在实际情况中，还有一种容器管理事务，容器事务主要是由 Java EE 应用服务器提供的，容器事务大多基于 JTA 完成，事实上这是在容器中覆盖了 JDBC 事务和 JTA 事务。

8.1.2 SQL Map 的 transactionManager 概述

SQL Map 可以自行配置事务管理服务。属性 type 指定所使用的事务管理器类型。这个属性值可以是一个类名，也可以是一个别名。包含在框架中的三个事务管理器分别是：JDBC、JTA 和 EXTERNAL。

- JDBC：通过常用的 Connection commit() 和 rollback() 方法，让 JDBC 管理事务。
- JTA：本事务管理器使用一个 JTA 全局事务，使 SQL Map 的事务包括在更大的事务范围内，这个更大的事务范围可能包括了其他的数据库和事务资源。这个配置需要一个 UserTransaction 属性，以便从 JNDI 获得一个 UserTransaction。
- EXTERNAL：这个配置可以让开发者自己管理事务。开发者仍然可以配置一个数据源，但事务不再作为框架生命周期的一部分被提交或回退。这意味着 SQL Map 外部应用的一部分必须自己管理事务。这个配置也可以用于没有事务管理的数据库（例如只读数据库）。

8.1.3 SQL Map 事务管理的设计模式

SQL Map 的事务管理要达到两个目的。第一是要从配置文件中获取不同的事务实现，第二是 JDBC、JTA 和外部事务等多种事务方式的具体实现。SQL Map 的事务管理类结构如图 8-1 所示。

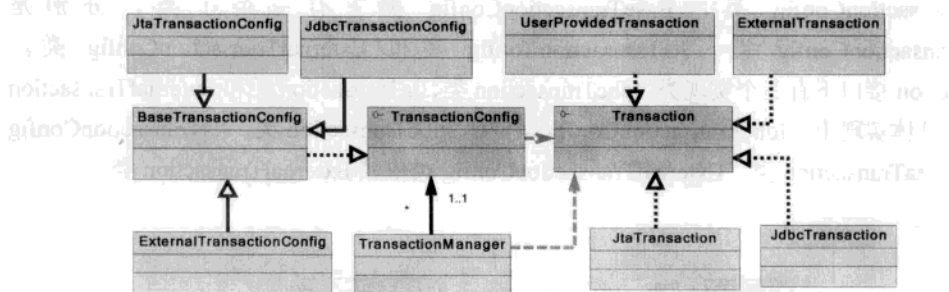


图 8-1 transactionManager 的类结构图

TransactionManager 类关联 TransactionConfig 接口并依赖 Transaction 接口。TransactionConfig 依赖 Transaction 接口并由 BaseTransactionConfig 类实现。BaseTransactionConfig 类由 JtaTransactionConfig、ExternalTransactionConfig 类和 JdbcTransactionConfig 类所继承。Transaction 接口由 UserProvidedTransaction 类、ExternalTransaction 类、JtaTransaction 类和 JdbcTransaction 类来实现。

这种设计模式是一种变形的桥梁模式。首先我们来看看桥梁模式的结构。

桥梁（Bridge）模式属于结构型设计模式。其标准定位为：将抽象部分与它的实现部分分离，使它们都可以独立地变化。具体地说，它将抽象化与实现化脱耦，使得两者可以独立地变化，也就是说，将他们之间的强关联变成弱关联，是指在一个软件系统的抽象化和实现化之间使用组合/聚合关系而不是继承关系，从而使两者可以独立地变化。

Bridge 结构如图 8-2 所示。Bridge 模式中的角色包括抽象化（Abstraction）角色、修正抽象化（Refine Abstraction）角色、实现化（Implementor）角色和具体实现化（Concrete Implementor）角色。这几个角色的说明如下。

① 抽象化（Abstraction）角色：Abstraction 定义抽象类的接口。维护一个指向 Implementor 类型对象的指针。抽象化（Abstraction）角色可以是接口，也可以是抽象类。属于不可缺少角色。

② 修正抽象化（Refine Abstraction）角色：RefinedAbstraction 扩充由 Abstraction 定义的接口 Implementor 定义实现类的接口。该接口不一定要与 Abstraction 的接口完全一致。

③ 实现化（Implementor）角色：Implementor 接口仅提供基本操作。这个抽象类规范具体实现化角色，规定出具体实现化角色当有的（非私有）方法和属性。实现化（Implementor）角色可以是接口，也可以是抽象类。属于不可缺少角色。

④ 具体实现化（Concrete Implementor）角色：Concreteimplementor 实现 Implementor 接口并定义它的具体实现。

SQL Map 的事务是一个变形的桥梁模式，其中两个桥墩是 TransactionConfig 接口和 Transaction 接口。但是这两者不是一种关联关系，而是一种依赖关系，即 TransactionConfig 接口依赖 Transaction 接口。同时，TransactionManager 类关联 TransactionConfig 接口并依赖 Transaction 接口，这也是一种结合两者的桥梁。TransactionConfig 接口下有抽象类 BaseTransactionConfig 类，BaseTransactionConfig 类下有三个子类，分别是 JdbcTransactionConfig 类、JtaTransactionConfig 类和 ExternalTransactionConfig 类。Transaction 接口下有三个实现类：JdbcTransaction 类、JtaTransaction 类和 ExternalTransaction 类。在具体实现中，JdbcTransactionConfig 类依赖 JdbcTransaction 类，JtaTransactionConfig 类依赖 JtaTransaction 类，ExternalTransactionConfig 类依赖 ExternalTransaction 类。

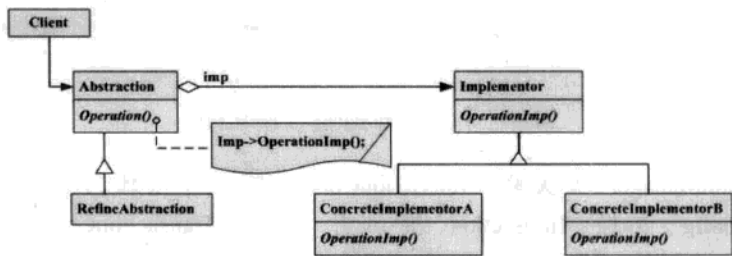


图 8-2 桥梁模式结构

按照桥梁模式的角色定义，把桥梁模式的角色映射到 SQL Map 事务设计的类和对象。

抽象化（Abstraction）角色——TransactionConfig 接口。

修正抽象化（Refine Abstraction）角色——BaseTransactionConfig 类、JdbcTransactionConfig 类、JtaTransactionConfig 类和 ExternalTransactionConfig 类。

实现化（Implementor）角色——Transaction 接口。

具体实现化（Concrete Implementor）角色——JdbcTransaction 类、JtaTransaction 类和 ExternalTransaction 类。

这样，可以比较好理解 SQL Map 事务管理的设计模式，所涉及的接口和类的功能说明如表 8-1 所示。

表 8-1 SQL Map 事务管理的类和接口说明

接口或类	功能描述
com.ibatis.sqlmap.engine.transaction.TransactionManager	用于事务管理的统一调度类
com.ibatis.sqlmap.engine.transaction.TransactionConfig	事务管理配置信息的转化接口
com.ibatis.sqlmap.engine.transaction.BaseTransactionConfig	事务管理配置信息的转化抽象类
com.ibatis.sqlmap.engine.transaction.jta.JtaTransactionConfig	Jta 事务配置信息的转化类
com.ibatis.sqlmap.engine.transaction.jdbc.JdbcTransactionConfig	Jdbc 事务配置信息的转化类
com.ibatis.sqlmap.engine.transaction.external.ExternalTransactionConfig	外部事务配置信息的转化类

续表

接口或类	功能描述
com.ibatis.sqlmap.engine.transaction.Transaction	事务实现的接口
com.ibatis.sqlmap.engine.transaction.user.UserProvidedTransaction	用户事务实现类
com.ibatis.sqlmap.engine.transaction.external.ExternalTransaction	外部事务实现类
com.ibatis.sqlmap.engine.transaction.jta.JtaTransaction	Jta 事务实现类
com.ibatis.sqlmap.engine.transaction.jdbc.JdbcTransaction	Jdbc 事务实现类

8.2 系统如何调用事务管理和 SQL Map 事务策略

8.1 节说明了 SQL Map 事务的设计结构。现在介绍平台如何调用事务和各个事务策略的实现。

8.2.1 SQL Map 如何调用事务

SQL Map 事务调用分为两个阶段,第 1 个阶段是通过配置文件形成 Transaction 的实例化对象,但是还是没有启动与数据库的连接。第 2 个阶段是外部开始调用,事务处理最终都转化到 Transaction 实例化对象的获得、提交和回滚。关于 Transaction 实例化对象的这些功能,在下面的事务策略中会依次说明。

第 1 个阶段在前面已经做了描述。通过对配置文件的解析,已经形成了 Transaction Manager 对象和其内部属性 DataSource 对象。

当外部开始进行事务管理时,首先要调用 TransactionManager 对象的 begin 方法。

```
public void begin(SessionScope sessionScope, int transactionIsolation) throws
SQLException, TransactionException {
    Transaction trans = sessionScope.getTransaction();
    TransactionState state = sessionScope.getTransactionState();
    if (state == TransactionState.STATE_STARTED) { throw new TransactionException
("....");
    } else if (state == TransactionState.STATE_USER_PROVIDED) { throw new
TransactionException("....");
    }

    trans = config.newTransaction(transactionIsolation);
    sessionScope.setCommitRequired(false);

    sessionScope.setTransaction(trans);
    sessionScope.setTransactionState(TransactionState.STATE_STARTED);
}
```

TransactionManager 对象去调用 DataSource 对象的新 Transaction 方法。为不失一般性,我们假定 DataSource 对象是一个 JDBC 的 DataSource 对象,实现代码如下。

```

    public Transaction newTransaction(int transactionIsolation) throws SQLException,
        TransactionException {
        return new JdbcTransaction(dataSource, transactionIsolation);
    }

```

而构造一个 JdbcTransaction 的代码如下：

```

    public JdbcTransaction(DataSource ds, int isolationLevel) throws Transaction
        Exception {
        // Check Parameters
        dataSource = ds;
        if (dataSource == null) {
            throw new TransactionException("JdbcTransaction initialization failed.Data
            Source was null.");
        }
        this.isolationLevel.setIsolationLevel(isolationLevel);
    }

```

这样就返回了一个新的 JdbcTransaction 对象，在 TransactionManager 对象调用 SessionScope 对象的 setTransaction(trans)方法，把 JdbcTransaction 对象载入到当前的 session 环境变量中。

后面的工作就是通过 session 环境变量作为一个变量容器，调用 JdbcTransaction 对象的处理方法，比如 getConnection 方法来获取一个数据库 Connection，commit 方法来实现数据库 Connection 的提交事务，rollback 方法来实现数据库 Connection 的回滚事务，close 方法来关闭和销毁当前的这个数据库 Connection。

关于实现 Transaction 接口的实例化对象的内容，在下面的事务策略章节中将有详细说明。

8.2.2 Java 事务类型

按照 Java 事务的类型，主要分为三种类型：JDBC 事务、JTA（Java Transaction API）事务和容器事务。下面分别介绍 Java 事务的三种类型。

1. JDBC 事务

JDBC 事务是用 java.sql.Connection 控制的。java.sql.Connection 提供了以下控制事务的方法：

```

    public void setAutoCommit(boolean)
    public boolean getAutoCommit()
    public void commit()
    public void rollback()

```

JDBC 事务的例子实现：

```

    java.sql.Connection conn = null;

```

```

try{
    javax.sql.DataSource ds = (javax.sql.DataSource) context.lookup("java:/ OracleDS");
    conn = ds.getConnection();
    conn.setAutoCommit(false);
    java.sql.Statement statement = conn.createStatement();
    conn.commit();
} catch (Exception e) {
    if(conn!=null)
        try(conn.rollback());catch(Exception e1){out.println("....");}
}finally{
    if(conn!=null)
        try(conn.close());catch(Exception e1){out.println("...");}
}

```

使用 JDBC 事务界定时, 可以将多个 SQL 语句结合到一个事务中。JDBC 事务的一个缺点是事务的范围局限于一个数据库 Connection。即一个 JDBC 事务不能跨越多个数据库。

JDBC Connection 接口 (java.sql.Connection) 提供了两种事务模式: 自动提交和手工提交。

在 JDBC 中, 事务操作默认是自动提交。也就是说, 一条对数据库的更新表达式代表一项事务操作, 操作成功后, 系统将自动调用 commit 来提交, 否则将调用 rollback 来回滚。

同时在 JDBC 中, 也可以通过调用 setAutoCommit(false)来禁止自动提交。之后就可以把多个数据库操作的表达式作为一个事务, 在操作完成后调用 commit 来进行整体提交, 倘若其中一个表达式操作失败, 都不会执行 commit, 并且将产生响应的异常; 此时就可以在异常捕获时调用 rollback 进行回滚。这样做可以保持多次更新操作后, 相关数据的处理就能保持一致性。

2. JTA (Java Transaction API) 事务

JTA 是一种高层的, 与实现无关联, 与协议无关的 API, 应用程序和应用服务器可以使用 JTA 来访问事务。JTA 为 Java EE 平台提供了分布式事务服务。一个分布式的事务涉及一个事务管理器和一个或者多个资源管理器, 如图 8-3 所示。一个资源管理器是任何类型的持久性数据存储。事务管理器负责协调所有事务参与者之间的通信。

JTA 允许应用程序执行分布式事务处理——在两个或多个网络计算机资源上访问并且更新数据, 这些数据可以分布在多个数据库上。JDBC 驱动程序的 JTA 支持极大地增强了数据访问能力。

如果要用 JTA 进行事务界定, 应用程序要调用 javax.transaction.UserTransaction 接口中的方法。用 JTA 界定事务, 那么就需要有一个实现 javax.sql.XADataSource、javax.sql.XAConnection 和 javax.sql.XAResource 接口的 JDBC 驱动程序。一个实现了这些接口的驱

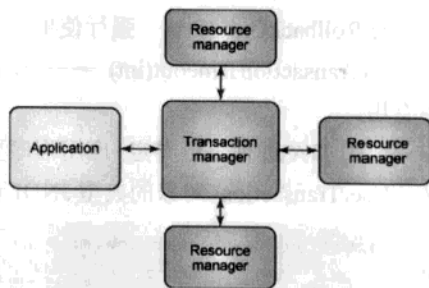


图 8-3 一个事务管理器和资源管理器

动程序将可以参与 JTA 事务。一个 XADataSource 对象就是一个 XAConnection 对象的工厂。XAConnections 是参与 JTA 事务的 JDBC 连接。要使用 JTA 事务，必须使用 XADataSource 来产生数据库连接，产生的连接为一个 XA 连接。XA 连接 (javax.sql.XAConnection) 和非 XA (java.sql.Connection) 连接的区别在于：XA 可以参与 JTA 的事务，而且不支持自动提交。Oracle、MS SQL Server、Sybase, DB2 等大型数据库才支持 XA，并支持分布事务。MySQL 只支持本地事务。

JTA 方式的实现过程：JTA 事务实现方式用 JNDI 查询数据源。一旦应用程序找到了数据源对象，它就调用 javax.sql.DataSource.getConnection() 以获得数据库的连接。XA 连接与非 XA 连接不同。一定要记住 XA 连接参与了 JTA 事务。这意味着 XA 连接不支持 JDBC 的自动提交功能。同时，应用程序一定不要对 XA 连接调用 java.sql.Connection.commit 或者 java.sql.Connection.rollback。相反，应用程序应该使用 UserTransaction.begin、UserTransaction.commit 和 UserTransaction.rollback。

javax.transaction.UserTransaction 接口提供了以下的事务控制方法。

begin()—— 创建一个新的事务，并将该事务与当前线程关联起来。

commit()—— 提交与当前线程有关联的事务。

rollback()—— 强行回滚与当前线程有关联的事务。

应用程序调用 begin 开始事务。应用程序调用 commit 或者 rollback 结束事务。在 begin() 和 commit() 之间发生的所有更新都是在一个事务中完成的。

UserTransaction 接口上的其他方法包括：

getStatus()—— 检索与当前线程有关的事务的状态。返回值是 javax.transaction.Status 接口上定义的常数。

setRollbackOnly()—— 强行使事务终止。

setTransactionTimeout(int) —— 设置事务中止前能运行的最大次数。这在避免死锁时很有用。

要用 JTA 进行事务界定，应用程序要调用 javax.transaction.UserTransaction 接口中的方法。UserTransaction 对象的典型 JNDI 查询代码如下：

```
import javax.transaction.*;
import javax.naming.*;
// ...
InitialContext ctx = new InitialContext();
Object txObj = ctx.lookup("java:comp/UserTransaction");
UserTransaction utx = (UserTransaction) txObj;
```

当应用程序找到了 UserTransaction 对象后，就可以开始事务了，代码如下所示。

```
utx.begin();
// ...
DataSource ds = obtainXADataSource();
Connection conn = ds.getConnection();
pstmt = conn.prepareStatement("UPDATE MOVIES ...");
```

```

pstmt.setString(1, "Spinal Tap");
pstmt.executeUpdate();
// ...
utx.commit();
// ...

```

当应用程序调用 `commit` 时，事务管理器用一个两阶段的提交协议结束事务。

JTA 事务工作流程，Java EE 应用（比如：WebLogic Server）将根据以下条件返回不同种类的包装器：

1. 所使用的 JDBC 驱动程序类是否支持 XA；
2. 是从 `DataSource` 还是从 `TxDataSource` 获得连接；
3. 调用 `getConnection()` 时是否在事务内运行；
4. 是否通过 RMI 从远程获得连接。

```

javax.transaction.UserTransaction tx = null;
java.sql.Connection conn = null;
try{
    //取得 JTA 事务，本例中是由 Jboss 容器管理
    tx = (javax.transaction.UserTransaction) context.lookup("java:comp/ UserTransaction");
    //取得数据库连接池，必须有支持 XA 的数据库、驱动程序
    javax.sql.DataSource dataSource = (javax.sql.DataSource) context.lookup("java:XAOracleDS");
    tx.begin();
    conn = dataSource.getConnection();
    java.sql.Statement statement = conn.createStatement();
    String sql = "....";
    int insert = statement.executeUpdate(sql);
    tx.commit();
} catch (Exception e) {
    if(tx!=null)
        try(tx.rollback();){catch(Exception e1){out.println("....");}}
}finally{
    if(conn!=null)
        try(conn.close();){catch(Exception e1){out.println(".....");}}
}

```

3. 容器事务

容器事务主要由 Java EE 应用服务器提供，容器事务大多是基于 JTA 完成的，这是一个基于 JNDI 的、相当复杂的 API 实现。相对编码实现 JTA 事务管理，我们可以通过 EJB 容器提供的容器事务管理机制（CMT）完成同一个功能，这项功能由 Java EE 应用服务器提供。这使得我们可以简单地指定将哪个方法加入事务，一旦指定，容器将负责事务管理任务。这是我们土建的解决方式，因为通过这种方式我们可以将事务代码排除在逻辑编码之外，同时将所有困难交给 Java EE 容器去解决。使用 EJB CMT 的另外一个好处就是程序员无需关心 JTA API 的编码，不过，理论上我们必须使用 EJB。

4. 三种事务差异

- (1) JDBC 事务控制的局限性在一个数据库连接内，但是其使用非常简单。
- (2) JTA 事务的功能强大，事务可以跨越多个数据库或多个事务，使用也比较复杂。
- (3) 容器事务，主要是指 Java EE 应用服务器提供的事务管理，局限于 EJB 应用使用。

针对这三种类型的事务，iBATIS 平台提供三个事务策略，分别是：JDBC、JTA 和 EXTERNAL。下面分别介绍 iBATIS 平台中这三种事务策略的实现过程。

8.2.3 SQL Map 中 JDBC 事务实现

iBATIS SQL Map 平台中 JDBC 事务操作的类结构如图 8-4 所示。

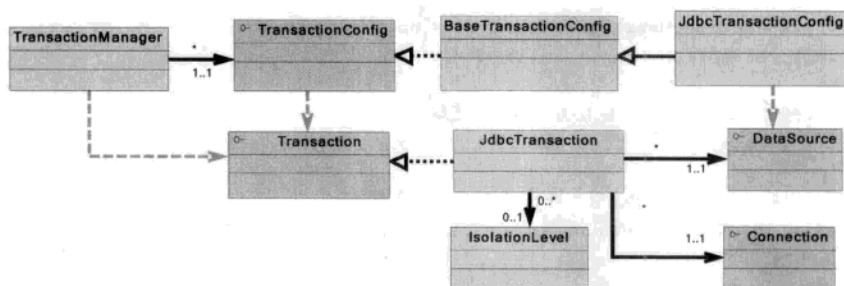


图 8-4 SQL Map 中 JDBC 事务类结构图

TransactionManager 类关联 TransactionConfig 接口，TransactionConfig 接口依赖 Transaction 接口。BaseTransactionConfig 抽象类实现 TransactionConfig 接口，而 JdbcTransactionConfig 继承 BaseTransactionConfig 抽象类，实际上也就是实现了 TransactionConfig 接口，同时 JdbcTransactionConfig 类依赖 JDK 的 DataSource 接口。JdbcTransaction 类实现 Transaction 接口的同时关联 IsolationLevel 类、DataSource 接口和 Connection 接口。

调用 TransactionManager 对象的 begin 方法、commit 方法和 end 方法最终都转化到 JDBC，其实现的序列如图 8-5 所示。

图 8-5 外部 sqlDelegate 为 sqlMapExecutorDelegate 实例化对象。外部 Statement 为 MappedStatement、DeleteStatement、UpdateStatement、InsertStatement、ProcedureStatement、SelectStatement 中一个类的实例化对象。

实现步骤说明。

第 1 阶段——Transaction 产生阶段。

第 1 步骤：sqlDelegate 对象调用 TransactionManager 对象的 begin 方法。

第 2 步骤：TransactionManager 对象调用 JdbcTransactionConfig 对象的新 Transaction 方法。

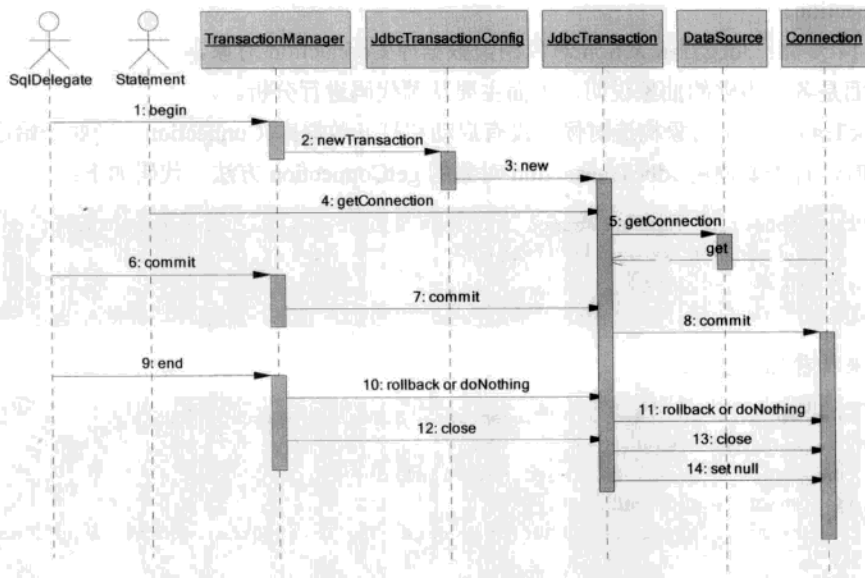


图 8-5 SQL Map 中 JDBC 事务实现序列图

第 3 步骤: JdbcTransactionConfig 对象创建一个 JdbcTransaction 实例化对象。

第 2 阶段——业务处理阶段

第 4 步骤: Statement 对象调用 JdbcTransaction 对象的 getConnection 方法, 获得数据库 Connection。

第 5 步骤: JdbcTransaction 对象调用 DataSource 对象的 getConnection 方法, 获得数据库 Connection 并返回。

第 3 阶段——事务提交阶段

第 6 步骤: SqlDelegate 对象调用 TransactionManager 对象的 commit 方法。

第 7 步骤: TransactionManager 对象调用 JdbcTransaction 对象的 commit 方法。

第 8 步骤: JdbcTransaction 对象调用数据库 Connection 对象的 commit 方法, 实现数据库的 commit 事务。

第 4 阶段——事务结束阶段

第 9 步骤: SqlDelegate 对象调用 TransactionManager 对象的 end 方法。

第 10 步骤: TransactionManager 对象根据情况, 如果事务处理有异常, 则调用 JdbcTransaction 对象的 rollback 方法, 否则什么也不做。

第 11 步骤: 如果 TransactionManager 对象调用 JdbcTransaction 对象的 rollback 方法, 则 JdbcTransaction 对象调用 Connection 对象的 rollback 方法, 否则什么也不做。

第 12 步骤: TransactionManager 对象调用 JdbcTransaction 对象的 close 方法。

第 13 步骤: JdbcTransaction 对象调用数据库 Connection 对象的 close 方法, 关闭数据

库 Connection。

第 14 步骤: JdbcTransaction 对象销毁数据库 Connection 对象。

上面是各个步骤的抽象说明,下面主要从源代码进行分析。

JdbcTransaction 对象构造时候并没有启动和打开数据库 Connection。当要开始进行事务处理时,首先要调用 JdbcTransaction 对象的 getConnection 方法。代码如下:

```
public Connection getConnection() throws SQLException, TransactionException {
    if (connection == null) { init(); }
    return connection;
}
```

再看看 init 方法。

```
private void init() throws SQLException, TransactionException {
    // Open JDBC Transaction
    connection = dataSource.getConnection();
    if (connection == null) {
        throw new TransactionException("Cause: The DataSource returned a null connection.");
    }
    // Isolation Level
    isolationLevel.applyIsolationLevel(connection);
    // AutoCommit
    if (connection.getAutoCommit()) {
        connection.setAutoCommit(false);
    }
    // Debug
    if (connectionLog.isDebugEnabled()) {
        connection = ConnectionLogProxy.newInstance(connection);
    }
}
```

从 dataSource 启动一个数据库 Connection 并赋值为当前数据库 Connection 属性。当进行 commit 操作时,代码如下。

```
public void commit() throws SQLException, TransactionException {
    if (connection != null) { connection.commit(); }
}
```

当进行 rollback 操作时,代码如下。

```
public void rollback() throws SQLException, TransactionException {
    if (connection != null) { connection.rollback(); }
}
```

当进行 close 操作时,代码如下。

```
public void close() throws SQLException, TransactionException {
    if (connection != null) {
        try { isolationLevel.restoreIsolationLevel(connection);

```

```

    } finally {
        connection.close();
        connection = null;
    }
}
}

```

8.2.4 SQL Map 中 JTA 事务实现

iBATIS SQL Map 平台中 JTA 事务操作的类如图 8-6 所示。

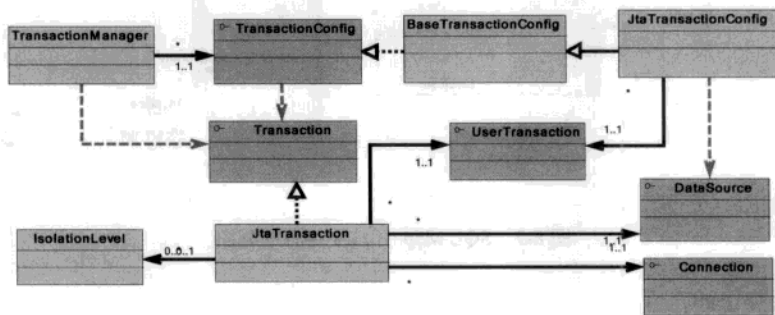


图 8-6 SQL Map 中 JTA 事务类结构图

JtaTransactionConfig 继承 BaseTransactionConfig 抽象类，实际上也就是实现了 TransactionConfig 接口，同时 JtaTransactionConfig 类依赖 JDK 的 DataSource 接口。JtaTransactionConfig 类关联 JDK 的 UserTransaction 接口。JtaTransaction 类实现 Transaction 接口，同时关联 IsolationLevel 类、DataSource 接口、Connection 接口和 UserTransaction 接口。

在 JTA 的配置文件内容如下所列。

```

<transactionManager type="JTA" >
    <property name="UserTransaction" value="java:/ctx/con/UserTransaction"/>
    <dataSource type="JNDI">
        <property name="DataSource" value="java:comp/env/jdbc/jpetstore"/>
    </dataSource>
</transactionManager>

```

在这里一定要有 UserTransaction 属性，否则再配置就不能生成 JTA 的 UserTransaction 对象。

调用 TransactionManager 对象的 begin 方法、commit 方法和 end 方法最终都转化到 JTA 事务。其实现的序列如图 8-7 所示。

在图 8-7 中外部 SqlDelegate 为 SqlMapExecutorDelegate 实例化对象。外部 Statement 为 MappedStatement、DeleteStatement、UpdateStatement、InsertStatement、ProcedureStatement、SelectStatement 中的一个类的实例化对象。

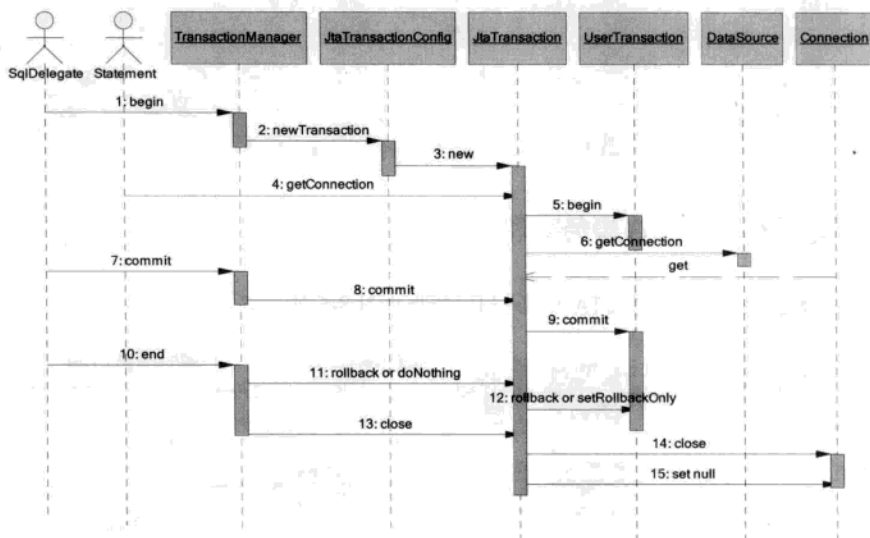


图 8-7 SQL Map 中 JTA 事务序列图

实现步骤说明:

第 1 阶段——Transaction 产生阶段。

第 1 步骤: SqlDelegate 对象调用 TransactionManager 对象的 begin 方法。

第 2 步骤: TransactionManager 对象调用 JtaTransactionConfig 对象的 newTransaction 方法。

第 3 步骤: JtaTransactionConfig 对象创建一个 JtaTransaction 实例化对象, 并把 UserTransaction 对象作为参数传递进去。

第 2 阶段——业务处理阶段

第 4 步骤: Statement 对象调用 JtaTransaction 对象的 getConnection 方法。

第 5 步骤: JtaTransaction 对象调用 UserTransaction 对象的 begin 方法。

第 6 步骤: JtaTransaction 对象调用 DataSource 对象的 getConnection 方法, 获得数据库 Connection 并返回。

第 3 阶段——事务提交阶段

第 7 步骤: SqlDelegate 对象调用 TransactionManager 对象的 commit 方法。

第 8 步骤: TransactionManager 对象调用 JtaTransaction 对象的 commit 方法。

第 9 步骤: JtaTransaction 对象调用数据库 UserTransaction 对象的 commit 方法, 实现数据库的 commit 事务。

第 4 阶段——事务结束阶段

第 10 步骤: SqlDelegate 对象调用 TransactionManager 对象的 end 方法。

第 11 步骤: TransactionManager 对象根据情况, 如果事务处理有异常, 则调用

JtaTransaction 对象的 rollback 方法, 否则什么也不做。

第 12 步骤: 如果事务处理有异常, 则 JtaTransaction 对象调用 UserTransaction 对象的 rollback 方法, 否则 JtaTransaction 对象调用 UserTransaction 对象的 setRollbackOnly 方法。

第 13 步骤: TransactionManager 对象调用 JtaTransaction 对象的 close 方法。

第 14 步骤: JtaTransaction 对象调用数据库 Connection 对象的 close 方法, 关闭数据库 Connection。

第 15 步骤: JtaTransaction 对象销毁数据库 Connection 对象。

在对事务处理时, JDBC 和 JTA 的区别就表现出来了。

JtaTransactionConfig 在配置的时候, 会有 setProperties 方法, 代码如下。

```
public void setProperties(Properties props) throws SQLException, TransactionException
{
    String utxName = null;
    try {
        utxName = (String) props.get("UserTransaction");
        InitialContext initCtx = new InitialContext();
        userTransaction = (UserTransaction) initCtx.lookup(utxName);
    } catch (NamingException e) {
        throw new SqlMapException("Error initializing JtaTransactionConfig while
looking up UserTransaction (" + utxName + "). Cause: " + e);
    }
}
```

这个方法实际上就创建了一个 userTransaction 对象。JtaTransaction 对象构造时代码如下:

```
public JtaTransaction(UserTransaction utx, DataSource ds, int isolationLevel)
throws TransactionException {
    // Check parameters
    userTransaction = utx;
    dataSource = ds;
    if (userTransaction == null) {
        throw new TransactionException("JtaTransaction initialization failed. User
Transaction was null.");
    }
    if (dataSource == null) {
        throw new TransactionException("JtaTransaction initialization failed. Data
Source was null.");
    }
    this.isolationLevel.setIsolationLevel(isolationLevel);
}
```

JtaTransaction 对象构造时获得了在配置时创建的 userTransaction 对象, 但还没有启动和打开数据库 Connection。当要开始进行事务处理时, 首先要调用 JtaTransaction 对象的 getConnection() 方法。代码如下。

```
public Connection getConnection() throws SQLException, TransactionException {
    if (connection == null) { init(); }
```

```

    return connection;
}

```

我们来看看 init() 方法。

```

private void init() throws TransactionException, SQLException {
    // Start JTA Transaction
    try {
        newTransaction = userTransaction.getStatus() == Status.STATUS_NO_TRANSACTION;
        if (newTransaction) {
            userTransaction.begin();
        }
    } catch (Exception e) {
        throw new TransactionException("JtaTransaction could not start transaction.
Cause: ", e);
    }

    // Open JDBC Connection
    connection = dataSource.getConnection();
    if (connection == null) {
        throw new TransactionException(" Cause: The DataSource returned a null
connection.");
    }
    // Isolation Level
    isolationLevel.applyIsolationLevel(connection);
    // AutoCommit
    if (connection.getAutoCommit()) {
        connection.setAutoCommit(false);
    }
    // Debug
    if (connectionLog.isDebugEnabled()) {
        connection = ConnectionLogProxy.newInstance(connection);
    }
}

```

从 dataSource 启动一个数据库 Connection 并赋值为当前数据库 Connection 属性。当进行 commit 操作时，代码如下。

```

public void commit() throws SQLException, TransactionException {
    if (connection != null) {
        if (committed) { throw new TransactionException("JtaTransaction could not
commit."); }
        try {
            if (newTransaction) {userTransaction.commit();}
        } catch (Exception e) {
            throw new TransactionException("JtaTransaction could not commit. Cause: ", e);
        }
        committed = true;
    }
}

```

当进行 rollback 操作时，代码如下。

```

public void rollback() throws SQLException, TransactionException {
    if (connection != null) {
        if (!committed) {
            try {
                if (userTransaction != null) {
                    if (newTransaction) { userTransaction.rollback(); }
                    else {
                        userTransaction.setRollbackOnly();
                    }
                }
            } catch (Exception e) {
                throw new TransactionException("JtaTransaction could not rollback. Cause: ", e);
            }
        }
    }
}

```

当进行 close 操作时，代码如下。

```

public void close() throws SQLException, TransactionException {
    if (connection != null) {
        try {
            isolationLevel.restoreIsolationLevel(connection);
        } finally {
            connection.close();
            connection = null;
        }
    }
}

```

8.2.5 SQL Map 的 External 事务实现

iBatis SQL Map 平台中 External 事务操作的类结构如图 8-8 所示。

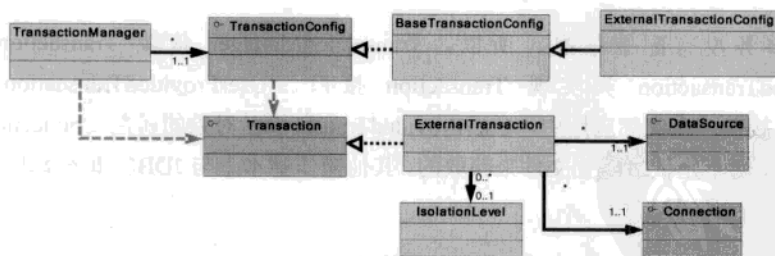


图 8-8 SQL Map 平台中 External 事务的类结构图

ExternalTransactionConfig 继承 BaseTransactionConfig 抽象类，实际上也就是实现了 TransactionConfig 接口，同时 ExternalTransaction 类依赖 JDK 的 DataSource 接口。JtaTransaction 类实现 Transaction 接口同时关联 IsolationLevel 类。

External 实现与 JDBC 事务实现基本相似，就是 commit 和 rollback 没有内容，主要都交到外部管理器去处理了，其序列图如图 8-9 所示。

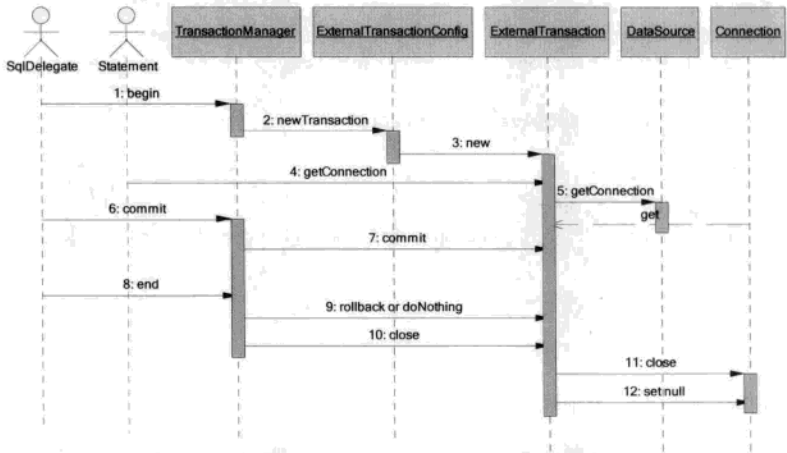


图 8-9 SQL Map 平台中 External 事务序列图

8.2.6 SQL Map 的用户事务实现

iBATIS SQL Map 平台中用户事务操作的类图结构如图 8-10 所示。

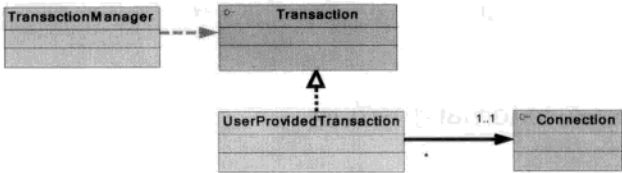


图 8-10 SQL Map 平台中用户事务操作的类结构图

用户事务没有配置信息，所以，TransactionManager 依赖 Transaction 接口。UserProvidedTransaction 类实现 Transaction 接口。UserProvidedTransaction 类关联 Connection 接口。在实际应用中，UserProvidedTransaction 对象的属性 Connection 对象是由外部传入、而不是依靠配置文件来获取的，其他操作基本上与 JDBC 事务实现基本相似。

8.3 SQL Map 的 DataSource 策略

8.3.1 关于 DataSource 的说明

在软件开发过程中，可能需要管理一些“瓶颈资源”，所谓瓶颈资源，就是这些资源的

使用在时间或者空间上都极大地影响着软件的使用效率。如网络连接、内存、进程等。为了充分利用瓶颈资源,有一个基本的管理原则是:尽可能地推迟资源的分配,并尽可能早地解除资源分配。因此软件工程人员采用了共享模式,将这些资源在时间或空间上尽量复用。

在数据库应用程序中,数据库的连接就是一种瓶颈资源。网络环境中,对于数据的访问往往都是非常频繁的,这使得数据库需要处理量的请求。一般的数据库连接的工作过程是:用户发送请求;建立数据库连接;用户使用数据库连接;用户退出;关闭数据库连接;该连接被销毁。再如此反复。由于用户操作的随意性比较大,即用户可能会十分频繁地要求建立连接,然后又将其关闭。如果这一应用是在高速局域网内,并且用户数并不多的情况下,不会有性能上的较大差别。但如果这一应用是在 Web 应用,并且有大量访问的情况下,那就会发现问题的严重性了。数据库连接的生成需要耗费系统开销,频繁地打开、连接或者仅是不断地生成连接都会使服务器不堪重负。而且在有些情况下,数据库连接需要使用网络连接,而一些平台会限制连接的数量。这就需要数据库连接的对象池——数据库连接池来解决这一问题。

数据库连接池则是数据库连接对象的对象池,它的基本模型如图 8-2 所示。当用户请求访问数据库时,数据库连接池会分配一个数据库连接(Connection)给该用户;当用户不再访问数据库时,它将使用的数据库连接(Connection)被重新放回到连接池中,而不会销毁该数据库连接(Connection)。

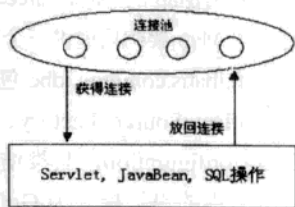


图 8-11 连接池模型

由于被重新放回池中的数据库连接在以后会被重复使用,这样就没有必要再去创建一个新的数据库连接(Connection)。

在 Java 语言中,DataSource 充当了数据库连接池的作用,DataSource 对象本身就是 java.sql.Connection 对象的工厂。相对于另一种获得 Connection 对象的机制——使用 Driver Manager 类,使用 DataSource 是一种更好的方法。实现了 DataSource 接口的对象,一般会被注册为基于 JNDI(Java Naming nadDireeto Interface) API 的命名服务。

DataSource 接口由 JDBC 驱动提供商来实现,共有三种典型的实现方法:

- ① 基本实现:产生一个标准的 Connection 对象;
- ② Connection 的池化实现:产生一个将自动参与到连接池中的 Connection 对象,这种实现将伴随着一个处于中间层的连接池管理器来使用;
- ③ 分布式事务实现:产生一个可能会被用于分布式事务的 Connection 对象,并且该对象几乎肯定会参与到连接池中。这种实现将伴随着一个处于中间层的事务管理器(几乎肯定还会有一个连接池管理器)来使用。

对于一般的 Java EE 数据库应用程序可以不采用 DataSource 对象。但是,对于那些需要用的数据库连接池或者分布式事务的应用程序系统,就必须使用 DataSource 对象来获得 Connection。使用数据库连接池时,当需要连接的时候,可以从连接池中取出一个连接分配给需要的用户。当对这个连接的使用完毕的时候,不必把它销毁,而是将它返回到连接

池中，以供一下次再使用。这样就可以大大地减少开销，提高响应速度，增强在多线程数据库应用程序的使用效率。

主流的商业数据库，如 Oracle、IBM DB2 等都有自己的数据库连接池。一些开源数据库，如 PostgreSQL 等都有它们自己的数据库连接池。其中 PostgreSQL 不仅有一般常见的前台连接池，还具有后台连接池。另外，主流的应用服务器（或 Web 容器），如 IBM Websphere、BEA WebLogic、Apache Tomcat 等，也都集成有数据库连接池。但是必须有其第三方的专用类方法支持连接池的用法。

8.3.2 SQL Map 的 DataSource 结构和内容

SQL Map 的 DataSource 是数据库管理的一部分，为 SQL Map 数据源设置了一系列参数。目前 SQL Map 架构只提供三个 DataSource Factory。涉及的包为 com.ibatis.sql map.engine.datasource 和 com.ibatis.common.jdbc 包。包括的类和接口有 DataSource Factory、JndiDataSource Factory、SimpleDataSource Factory、DataSource、Simple DataSource、Dbcp DataSource Factory 和 DbcpConfiguration，其类结构如图 8-12 所示。

采用的设计模式为 Gof23 设计模式中的工厂方法模式。具体工厂的类如下：

工厂（Factory Method）模式属于创建性的设计模式，其标准定义为定义一个用于创建对象的接口，让子类决定实例化哪一个类。Factory Method 使一个类的实例化延迟到其子类。它要求工厂类和产品类分开。由一个工厂类可以根据传入的参量决定创建出哪一种产品类的实例，但这些不同的实例有共同的父类。Factory Method 把创建这些实例的具体过程封装起来了。当一个类无法预料将要创建哪种类的对象，或是一个类需要由子类来指定创建的对象时，我们就需要用到 Factory Method 模式了。

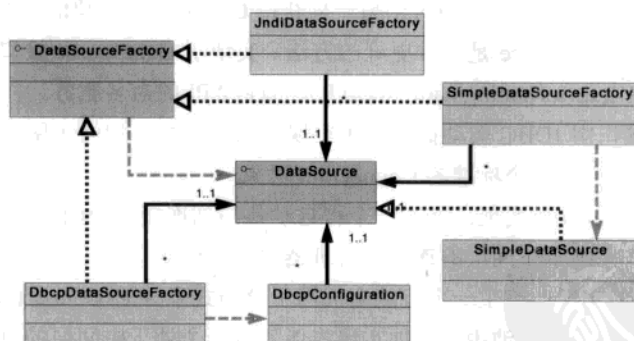


图 8-12 SQL Map 平台中 DataSource 类结构图

工厂（Factory Method）模式结构如图 8-13 所示。工厂模式涉及抽象工厂角色、具体工厂角色、抽象产品角色以及具体产品角色等。角色说明如下。

① 抽象工厂（Creator）角色：任何在模式中创建对象的工厂类必须实现这个接口。

在实际的系统中，这个角色也常常使用抽象类或接口实现。

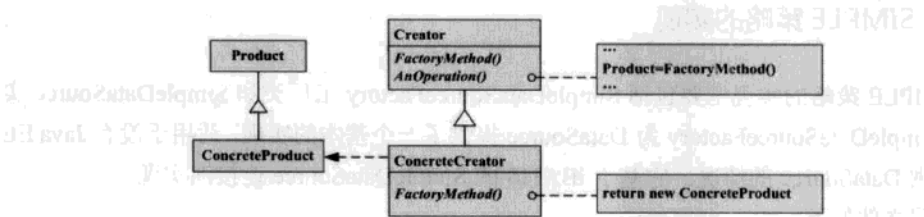


图 8-13 工厂方法模式结构

② 具体工厂（Concrete Creator）角色：担任这个角色的是实现了抽象工厂接口的具体类。具体工厂角色含有与应用密切相关的逻辑，并且受到应用程序的调用以创建产品对象。在实际的系统中这个角色使用具体类实现。

③ 抽象产品（Product）角色：工厂方法模式所创建的对象的上类型，也就是产品对象共同父类或共同拥有的接口。在实际的系统中，这个角色也常常使用抽象类或者接口来实现。

④ 具体产品（Concrete Product）角色：这个角色实现了抽象产品角色所声明的接口。工厂方法模式所创建的每一个对象都是某个具体产品角色的实例。这个角色使用具体类来实现。

对应于 SQL Map 的 DataSource，其中 DataSourceFactory 接口是抽象工厂（Creator）角色，SimpleDataSourceFactory、DbcpDataSourceFactory、JndiDataSourceFactory 是具体工厂（Concrete Creator）角色。DataSource 接口是抽象产品（Product）角色。SimpleDataSource 是具体产品（Concrete Product）角色，而 DbcpConfiguration 其实为转化过来的具体产品（Concrete Product）角色。

SOL MAP 的 DataSource。涉及的接口和类如表 8-2 所示。

表 8-2 SQL Map 的 DataSource 组件中的类和接口及其功能列表

接口或类	功能描述	备 注
com.ibatis.sqlmap.engine.datasource.DataSourceFactory	数据源工厂接口	
com.ibatis.sqlmap.engine.datasource.SimpleDataSourceFactory	iBATIS 自定义数据源工厂实现类	
com.ibatis.sqlmap.engine.datasource.DbcpDataSourceFactory	Dbcp 数据源工厂实现类	
com.ibatis.sqlmap.engine.datasource.JndiDataSourceFactory	Jndi 数据源工厂实现类	
com.ibatis.common.jdbc.SimpleDataSource	iBATIS 自定义数据源实现类	
com.ibatis.common.jdbc.DbcpConfiguration	Dbcp 数据源实现类	

在解析配置文件的时候，代码如下。

```
DataSourceFactory dsFactory = (DataSourceFactory) Resources.instantiate(type);
dsFactory.initialize(props);
```

实际上，这就已经创建出了 DataSource。不同的 DataSourceFactory 创建不同的 DataSource。下面分别介绍这几种策略。

8.3.3 SIMPLE 策略的实现

SIMPLE 策略的实现主要包括 SimpleDataSourceFactory 工厂类和 SimpleDataSource 实现类。SimpleDataSourceFactory 为 DataSource 提供了一个基本的实现,适用于没有 Java EE 容器提供 DataSource 的情况。它基于 iBATIS 的 SimpleDataSource 连接池实现。

配置文件如下。

```
<dataSource type="SIMPLE">
  <property name="JDBC.Driver" value="org.postgresql.Driver"/>
  <property name="JDBC.ConnectionURL" value="jdbc:postgresql://server:5432/dbname"/>
  <property name="JDBC.Username" value="user"/>
  <property name="JDBC.Password" value="password"/>
  <!-- OPTIONAL PROPERTIES BELOW -->
  <property name="Pool.MaximumActiveConnections" value="10"/>
  <property name="Pool.MaximumIdleConnections" value="5"/>
  <property name="Pool.MaximumCheckoutTime" value="120000"/>
  <property name="Pool.TimeToWait" value="10000"/>
  <property name="Pool.PingQuery" value="select * from dual"/>
  <property name="Pool.PingEnabled" value="false"/>
  <property name="Pool.PingConnectionsOlderThan" value="0"/>
  <property name="Pool.PingConnectionsNotUsedFor" value="0"/>
</dataSource>
```

让我们来看看 SimpleDataSourceFactory 的 initialize 方法。

```
public void initialize(Map map) {
    dataSource = new SimpleDataSource(map);
}
```

initialize 方法中构造一个 SimpleDataSource 实例,其中 SimpleDataSource 类是 iBATIS 平台实现 DataSource 接口的实现类。SimpleDataSource 的构造方法就是把配置文件中的属性进行转化成 SimpleDataSource 的属性。

关于 SimpleDataSource 的构造和内部使用的内容在 8.4 节中有说明。

8.3.4 DBCP 策略实现

DBCP 策略主要包括 DbcpDataSourceFactory 工厂类和 DbcpConfiguration 实现类。DbcpDataSourceFactory 实现使用 Jakarta DBCP (Database Connection Pool) 的 DataSource API 提供连接池服务,适用于应用/Web 容器不提供 DataSource 服务的情况,或执行一个单独的应用。DbcpDataSourceFactory 中必须要配置参数例子如下:

```
<dataSource type="SIMPLE">
  <property name="JDBC.Driver" value="org.postgresql.Driver"/>
  <property name="JDBC.ConnectionURL" value="jdbc:postgresql://server:5432/dbname"/>
```

```

<property name="JDBC.Username" value="user"/>
<property name="JDBC.Password" value="password"/>
<!-- OPTIONAL PROPERTIES BELOW -->
<property name="Pool.MaximumActiveConnections" value="10"/>
<property name="Pool.MaximumIdleConnections" value="5"/>
<property name="Pool.MaximumCheckoutTime" value="120000"/>
<property name="Pool.TimeToWait" value="10000"/>
<property name="Pool.PingQuery" value="select * from dual"/>
<property name="Pool.PingEnabled" value="false"/>
<property name="Pool.PingConnectionsOlderThan" value="0"/>
<property name="Pool.PingConnectionsNotUsedFor" value="0"/>
</dataSource>

```

让我们来看看 DbcpDataSourceFactory 工厂类的 initialize 方法。

```

public void initialize(Map map) {
    DbcpConfiguration dbcp = new DbcpConfiguration(map);
    dataSource = dbcp.getDataSource();
}

```

实例化一个 DbcpConfiguration 对象，我们来看一下 DbcpConfiguration 的构造方法。

```

public DbcpConfiguration(Map properties) {
    try {
        dataSource = legacyDbcpConfiguration(properties);
        if (dataSource == null) { dataSource = newDbcpConfiguration(properties); }
    } catch (Exception e) {
        throw new RuntimeException("Error initializing DbcpDataSourceFactory. Cause: "
            + e, e);
    }
}

```

其实，在这里只有 legacyDbcpConfiguration 方法就足够了，增加一个 newDbcpConfiguration 是更保险。

```

private BasicDataSource legacyDbcpConfiguration(Map map) {
    BasicDataSource basicDataSource = null;
    if (map.containsKey("JDBC.Driver")) {
        basicDataSource = new BasicDataSource();
        String driver = (String) map.get("JDBC.Driver");
        String url = (String) map.get("JDBC.ConnectionURL");
        String username = (String) map.get("JDBC.Username");
        String password = (String) map.get("JDBC.Password");
        String validationQuery = (String) map.get("Pool.ValidationQuery");
        String maxActive = (String) map.get("Pool.MaximumActiveConnections");
        String maxIdle = (String) map.get("Pool.MaximumIdleConnections");
        String maxWait = (String) map.get("Pool.MaximumWait");

        basicDataSource.setUrl(url);
        basicDataSource.setDriverClassName(driver);
        basicDataSource.setUsername(username);
        basicDataSource.setPassword(password);
    }
}

```

```

        if (notEmpty(validationQuery)) { basicDataSource.setValidationQuery (validation
Query); }
        if (notEmpty(maxActive)) { basicDataSource.setMaxActive(Integer.parseInt (max
Active)); }
        if (notEmpty(maxIdle)) {basicDataSource.setMaxIdle(Integer. ParseInt (max
Idle)); }
        if (notEmpty(maxWait)) { basicDataSource.setMaxWait(Integer. ParseInt (max
Wait)); }
        Iterator props = map.keySet().iterator();
        while (props.hasNext()) {
            String propertyName = (String) props.next();
            if (propertyName.startsWith(ADD_DRIVER_PROPS_PREFIX)) {
                String value = (String) map.get(propertyName);
                basicDataSource.addConnectionProperty(propertyName.substring(ADD_
DRIVER_PROPS_PREFIX_LENGTH), value);
            }
        }
        return basicDataSource;
    }
}

```

在这里创建一个 Jakarta DBCP 的 BasicDataSource 对象，并把配置文件的参数初始化该对象。

8.3.5 JNDI 策略实现

JNDI 策略只有 JndiDataSourceFactory 工厂类。JndiDataSourceFactory 在应用容器内部从 JNDI Context 中查找 DataSource 实现。当使用应用服务器，并且服务器提供了容器管理的连接池和相关 DataSource 实现的情况下，可以使用 JndiDataSourceFactory。使用 JDBC DataSource 的标准方法是通过 JNDI 来查找。JndiDataSourceFactory 必须要配置的属性如下：

```

<dataSource type="JNDI">
    <property name="DataSource" value="java:comp/env/jdbc/jpetstore"/>
</dataSource>

```

注意，UserTransaction 属性指向 UserTransaction 实例所在的 JNDI 位置。JTA 事务管理需要它，以使 SQL Map 能够参与涉及其他数据库和事务资源的范围更大的事务。

让我们来看看 DbcpDataSourceFactory 工厂类的 initialize 方法。

```

public void initialize(Map properties) {
    try {
        InitialContext initCtx = null;
        Hashtable context = getContextProperties(properties);

        if (context == null) { initCtx = new InitialContext();
        } else { initCtx = new InitialContext(context);
        }
    }
}

```

```

        if (properties.containsKey("DataSource")) {
            dataSource = (DataSource) initCtx.lookup((String) properties.get ("Data
Source"));
        } else if (properties.containsKey("DBJndiContext")) { // LEGACY --Backward
compatibility
            dataSource = (DataSource) initCtx.lookup((String) properties.get ("DBJndi
Context"));
        } else if (properties.containsKey("DBFullJndiContext")) { // LEGACY --Backward
compatibility
            dataSource = (DataSource) initCtx.lookup((String) properties.get ("DBFull
JndiContext"));
        } else if (properties.containsKey("DBInitialContext")
            && properties.containsKey("DBLookup")) { // LEGACY --Backward compati-
bility
            Context ctx = (Context) initCtx.lookup((String) properties.get ("DBInitial
Context"));
            dataSource = (DataSource) ctx.lookup((String) properties.get("DBLookup"));
        }
    } catch (NamingException e) {
        throw new SqlMapException("There was an error configuring JndiDataSource
TransactionPool. Cause: " + e, e);
    }
}

```

直接通过 JNDI 查询来获得一个 DataSource 对象。

为什么要采用 JNDI，这样可以实现系统的通用性。这里简单地了解一下 JNDI。

开发人员进行数据库开发，都需要对 JDBC 驱动程序类的引用进行编码，并通过使用适当的 JDBC URL 连接到数据库，代码如下。

```

Connection conn=null;
Class.forName("com.mysql.jdbc.Driver", true, Thread.currentThread().getContext
ClassLoader());
conn=DriverManager. getConnection("jdbc:mysql://MyDBServer?user=qingfeng&password=
mingyue");
.....
conn.close();

```

这样采用的是硬编码，其存在的问题如下。

① 数据库服务器名称、用户名和口令都可能需要改变，由此引发 JDBC URL 需要修改，造成代码也要重新编译。

② 数据库可能改用别的产品，如改用 DB2 或者 Oracle，引发 JDBC 驱动程序包和类名需要修改。

③ 随着实际使用终端的增加，原配置的链接池参数可能需要调整。

正是由于存在这些问题，JNDI 就应这而生。使用 JNDI 之后，只需要在配置文件中配置 JNDI 参数，定义一个数据源，也就是 JDBC 引用参数，给这个数据源设置一个名称；然后，在程序中，通过数据源名称引用数据源从而访问后台数据库。配置非常简便，而且

可维护性还很强。

8.4 SQL Map 自定义 DataSource 实现

iBATIS 实现了一个简单的 DataSource 叫 SimpleDataSource。SimpleDataSource 经过简单修改后可以完全应用到其他系统中。要分析 SimpleDataSource 的实现，首先要了解一下 DataSource 的结构和一些常用的实现模式。

8.4.1 DataSource 接口的结构

标准 JDBC 接口没有提供 DataSource 资源的管理方法。开发人员自己负责管理数据库资源。虽然在 JDBC 规范中，多次提及资源的关闭/回收及其他的合理运用。但最稳妥的方式，还是为应用提供有效的管理手段。所以，JDBC2.0 用一种替代的方法，引入了 DataSource 的实现，为第三方应用服务器提供了一个由数据库厂家实现的管理标准接口。一个 DataSource 对象代表了一个真正的数据源。根据 DataSource 的实现方法，数据源既可以从关系数据库，也可以是电子表格，还可以是一个表格形式的文件。当一个 DataSource 对象注册到名字服务中时，应用程序就可以通过名字服务获得 DataSource 对象，并用它来产生一个与 DataSource 代表的数据源之间的连接。javax.sql.DataSource 的接口方法如表 8-3 所示。

表 8-3 javax.sql.DataSource 接口的方法和参数列表

序 号	返 回 值	方法和参数	说 明
1	Connection	getConnection()	尝试建立与此 DataSource 对象表示的数据源的链接
2	Connection	getConnection(String username, String password)	尝试建立与此 DataSource 对象表示的数据源的链接
3	int	getLoginTimeout()	获取此数据源尝试链接到某一数据库时可以等待的最长时间，以秒为单位
4	PrintWriter	getLogWriter()	获得此 DataSource 对象的日志 writer
5	void	setLoginTimeout(int seconds)	设置数据源尝试链接到某一数据库时将等待的最长时间，以秒为单位
6	void	setLogWriter(PrintWriter out)	将此 DataSource 对象的日志 writer 设置为给定的 java.io.PrintWriter 对象

8.4.2 实现 DataSource 的设计思路

正常情况下编写 Java 程序实现一个 javax.sql.DataSource 接口的设计思路包括下面几个方面。

- ① 进行 DataSource 的属性初始化。建立一个静态的 Connection 池，即在系统初始化

时向连接池中分配一定数目的 Connection，并且不能随意关闭。预先放入的 Connection 数是根据具体应用的设定来决定的，并且要设定一个连接池中容纳 Connection 数的最大值、最小值、等待时间等。

② 分配及释放。连接池的分配与释放，对系统的性能有很大的影响。合理的分配与释放，可以提高连接的复用度，从而降低建立新连接的开销，同时还可以加快用户的访问速度。当用户请求数据库 Connection 时，首先看链接池中是否有空闲 Connection，即那些没有被分配出去的 Connection。如果有空闲 Connection，则为用户分配一个 Connection。如果没有空闲 Connection，就看 Connection 池已经提供地连接数是否达到了最大值，如果没有，就新建一个连接分配给用户；如果已经达到最大值，就只能等待其他用户释放 Connection，这需要在初始化的时候设定一个最大等待时间。

③ 并发处理。在多线程环境中，必须保证 Connection 池管理自身数据库 Connection 的一致性和 Connection 内部数据的一致性。也就是说，对于可能被多个用户访问的数据，在同一个时刻只能让一个用户进行操作。这可以利用 Java 提供的多线程机制和同步语句来实现。使用 synchronized 关键字即可确保线程是同步的。使用方法为直接在类方法前面加上 synchronized 关键字，如：`public synchronized Connection getConnection`。

④ 事务处理：JDK 的 Connection 类本身提供了对事务的支持，可以通过设置 Connection 的 AutoCommit 属性为 false，然后显式的调用 commit 或 rollback 方法来实现。但要高效地进行 Connection 复用，就必须提供相应的事务支持机制。可采用每一个事务独占一个连接来实现，这种方法可以大大降低事务管理的复杂性。

⑤ close 处理。对于数据库连接，如果采用 close 方法，那将销毁当前对象，这实际上就已经没有连接池的概念了。故连接池中连接的 close 方法必须将本 connection 放回数据库连接池。为了接管数据库连接的 close 方法，必须采用一种拦截器技术，把本应该有 connection 接口实现的方法转化到自己定义的方法上来。在 Java 中同样也有这样一个机制，即动态代理。Java 提供了一个 Proxy 类和一个 InvocationHandler，这两个类都在 java.lang.reflect 包中。当外部调用一个 Proxy 实例的方法时会触发 InvocationHandler 的 invoke 方法。这种动态代理机制只能接管接口的方法，而对一般的类无效，考虑到 java.sql.Connection 本身也是一个接口，由此就找到了解决如何接管 close 方法的出路。

⑥ 连接池的配置与维护：连接池中到底应该放置多少连接，才能使系统的性能最佳？系统可采取设置最小连接数（minConn）和最大连接数（maxConn）来控制连接池中的连接。最小连接数是系统启动时连接池所创建的连接数。如果创建过多，则系统启动就慢，但创建后系统的响应速度会很快；如果创建过少，则系统启动得很快，响应起来却慢。这样，可以在开发时，设置较小的最小连接数，开发起来会快，而在系统实际使用时设置较大的，因为这样对访问客户来说速度会快些。最大连接数是连接池中允许连接的最大数目，具体设置多少，要看系统的访问量，可通过反复测试，找到最佳点。如何确保连接池中的最小连接数呢？有动态和静态两种策略。动态即每隔一定时间就对连接池进行检测，如果发现连接数量小于最小连接数，则补充相应数量的新连接，以保证连接池的正常运转。静

态是发现空闲连接不够时再去检查。

对于一个 `javax.sql.DataSource` 接口的实现类结构，如图 8-14 所示。

在数据库连接池容器要创建几个连接池，一是正在使用连接的活动连接池，另一个是已经使用过连接后的连接池，也叫做空闲池。空闲池中连接的来源是外部已经采用了 `close` 方法的连接。在这种情况下，获得 `Connection` 的方式是：每当用户请求一个连接时，系统首先检查空闲池内有没有空闲连接。如果有就把建立时间最长（通过容器的顺序存放实现）的那个连接分配给他（实际是先做连接是否有效的判断，如果可用就分配给用户，如不可用就把这个连接从空闲池删掉，重新检测空闲池是否还有连接）；如果在空闲池没有连接，则检查当前所开连接池是否达到连接池所允许的最大连接数（`maxConn`），如果没有达到，就新建一个连接，并把这个连接放到活动连接池中。如果已经达到，就等待一定的时间（`timeout`）。如果在等待的时间内有连接被释放出来就可以把这个连接分配给等待的用户，如果等待时间超过预定时间 `timeout`，则返回空值（`null`）。系统对已经分配出去正在使用的连接只做计数，当使用完后再返还给空闲池。对于空闲连接的状态，可开辟专门的线程定时检测，这样会花费一定的系统开销，但可以保证较快的响应速度。也可采取不开辟专门线程，只是在分配前检测的方法。

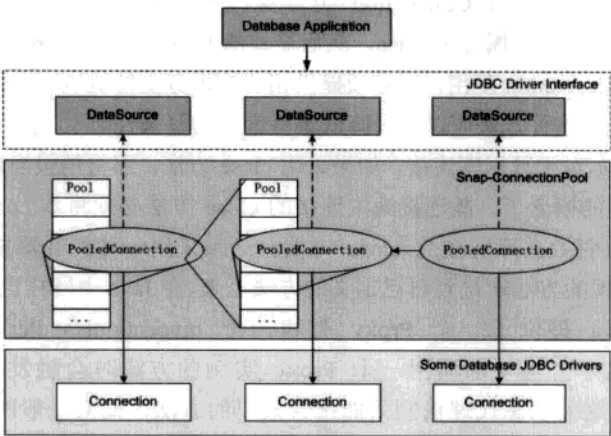


图 8-14 `javax.sql.DataSource` 接口的实现类结构

8.4.3 SimpleDataSource 设计和实现

在 iBATIS 中实现了一个简化版本的 `DataSource`，该 `DataSource` 支持数据库连接池，但是不支持分布式。该 `DataSource` 由 `SimpleDataSource` 类来实现，在 `SQL Map` 配置文件中进行配置的方式如下。

```
<transactionManager type="JDBC">
  <dataSource type="SIMPLE">
    <property name="JDBC.Driver" value="org.postgresql.Driver"/>
  </dataSource>
</transactionManager>
```



```

<property name="JDBC.ConnectionURL" value="jdbc:postgresql://server:
5432/dbname"/>
<property name="JDBC.Username" value="user"/>
<property name="JDBC.Password" value="password"/>
<!-- OPTIONAL PROPERTIES BELOW -->
<property name="Pool.MaximumActiveConnections" value="10"/>
<property name="Pool.MaximumIdleConnections" value="5"/>
<property name="Pool.MaximumCheckoutTime" value="120000"/>
<property name="Pool.TimeToWait" value="10000"/>
<property name="Pool.PingQuery" value="select * from dual"/>
<property name="Pool.PingEnabled" value="false"/>
<property name="Pool.PingConnectionsOlderThan" value="0"/>
<property name="Pool.PingConnectionsNotUsedFor" value="0"/>
</dataSource>
</transactionManager>

```

在 SQL Map 配置文件的 `dataSource` 节点中,属性是可以进行任意定义的。在上述文件,上半部分的属性定义主要是针对 JDBC 数据库连接的属性信息。下半部分的可选属性是定义关于连接池的初始化属性。

调用 `SimpleDataSource` 的代码如下。

```

DataSource dataSource = new SimpleDataSource(props); //properties usually loaded
from a file
Connection conn = dataSource.getConnection();
//.....database queries and updates
conn.commit();
conn.close(); //connections retrieved from SimpleDataSource will return to the pool
when closed

```

`SimpleDataSource` 组件其实就只有一个类,即 `SimpleDataSource` 类,但是 `SimpleDataSource` 类包含了一个内部静态类。其组件的类结构模式如图 8-15 所示。

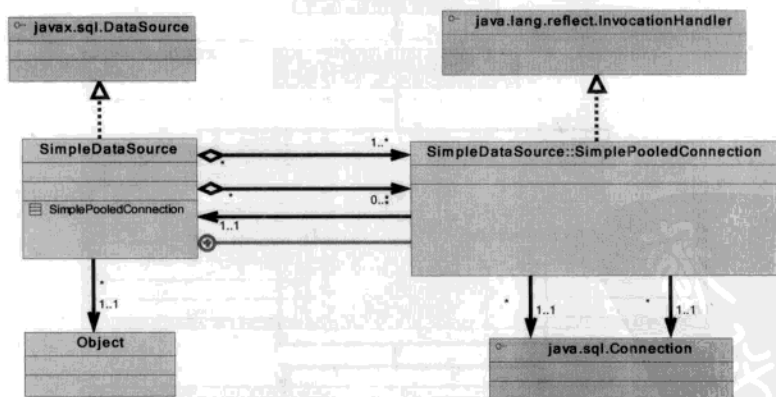


图 8-15 `SimpleDataSource` 组件类结构图

`SimpleDataSource` 类实现 `javax.sql.DataSource` 接口。`SimplePooledConnection` 类实现

java.lang.reflect.InvocationHandler 接口，充当动态代理。SimplePooledConnection 类是 SimpleDataSource 类的内部类。同时 SimpleDataSource 类两次聚合 SimplePooledConnection 内部类。而 SimplePooledConnection 内部类关联 SimpleDataSource 类并且两次关联 java.sql.Connection。

SimpleDataSource 类的属性如表 8-4 所示。

表 8-4 SimpleDataSource 类的属性列表

属性名称	类 型	功能和用途	备 注
POOL_LOCK	Object	多线程同步的锁对象	
idleConnections	List	空闲的连接池列表，内部对象为 SimplePooledConnection 实例化对象	
activeConnections	List	活动的连接池列表，内部对象为 SimplePooledConnection 实例化对象	
requestCount	long	请求连接的总数	
accumulatedRequestTime	long	累计请求时间	
accumulatedCheckoutTime	long	累计检查时间	
claimedOverdueConnectionCount	long	统计超时并被回收的 SimplePooledConnection 实例化对象的个数	
accumulatedCheckoutTimeOfOverdueConnections		统计超时并被回收的 SimplePooledConnection 实例化对象的时间	

由于 SimpleDataSource 是实现 javax.sql.DataSource 接口的实现类，必须有 getConnection()、getConnection(String username, String password)、getLoginTimeout()、getLogWriter()、setLoginTimeout(int seconds)、setLogWriter(PrintWriter out) 等方法。其框架结构如图 8-16 所示。

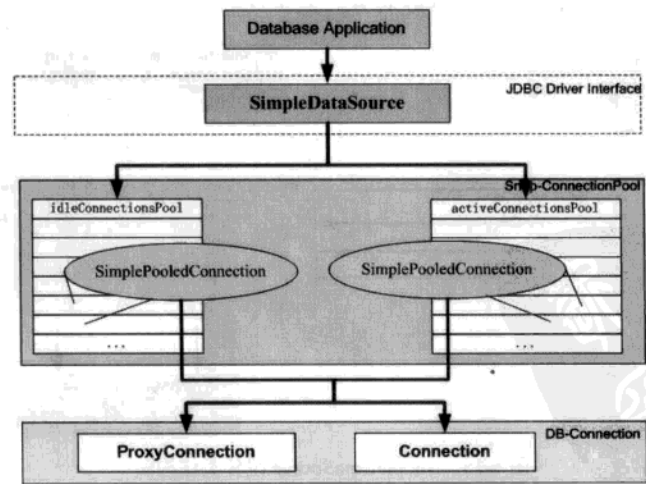


图 8-16 SimpleDataSource 实现框架

系统初始化的时候,把连接数据库的信息都保持到 SimpleDataSource 内部。在图 8-23 中 SimpleDataSource 类有两个 List 变量,第一个 List 变量叫 activeConnections 充当连接代理池容器。连接代理池容器是正在激活使用的连接池,叫活动连接代理池容器,第二个 List 变量叫 idleConnections 充当已经空闲的连接代理,叫空闲连接代理池。但是这个容器内装的并不是真正的数据库连接,而是 SimpleDataSource 类的子类 SimplePooledConnection 实例化对象。而 SimplePooledConnection 实例化对象只是充当了一个代理角色,其代理的内容才是真正的数据库连接。这里采用了 GoF 设计模式中的代理模式,而且还是 Java 特有的动态代理。

这里简单介绍一下代理设计模式。代理 (Proxy) 模式属于结构型设计模式,其标准定义是为其他对象提供一种代理以控制对这个对象的访问。代理就是一个人或一个机构代表另一个人或者一个机构采取行动。某些情况下,客户不想或者不能够直接引用一个对象,代理对象可以在客户和目标对象中直接起到中介的作用。客户端分辨不出代理主题对象与真实主题对象。代理模式可以不知道真正的被代理对象,而仅仅持有有一个被代理对象的接口,这时候代理对象不能够创建被代理对象,被代理对象必须有系统的其他角色代为创建并传入。代理 (Proxy) 模式结构如图 8-17 所示,其角色包括抽象主题 (Subject) 角色、代理主题 (Proxy) 角色和真实主题 (Real Subject) 角色。

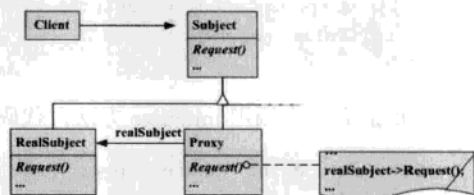


图 8-17 代理模式结构

① 抽象主题 (Subject) 角色: 声明了真实主题和代理主题的共同接口,这样一来在任何可以使用真实主题的地方都可以使用代理

主题。该类的实现中保存一个引用使得代理可以访问实体,提供一个与实体接口相同的代理接口,这样代理就可以用来替代实体来控制对实体的存取,并可能负责创建和删除实体。

② 代理主题 (Proxy) 角色: 代理主题角色内部含有对真实主题的引用,从而可以在任何时候操作真实主题对象;代理主题角色提供一个与真实主题角色相同的接口,以便可以在任何时候都可以替代真实主体;控制对真实主题的引用,负责在需要的时候创建真实主题对象 (和删除真实主题对象);代理角色通常在将客户端调用传递给真实的主题之前或者之后,都要执行某个操作,而不是单纯地将调用传递给真实主题的对象。

③ 真实主题 (Real Subject) 角色: 定义了代理角色所代表的真实对象,包含实体具体实现的代码。

在本例中, javax.sql.DataSource 接口是一个抽象主题 (Subject) 角色, SimpleDataSource 类 SimplePooledConnection 内部类充当代理主题 (Proxy) 角色,真正实现的数据库 Connection 实例化对象是真实主题 (Real Subject) 角色。

每当用户请求一个数据库 Connection 时,采用同步线程方式,进入获取数据库连接 Proxy 的循环,即获得了活动的数据库连接 Proxy 才退出循环。系统首先检查空闲数据库连接 Proxy 池内有没有空闲连接 Proxy。如果有就把建立时间最长 (通过容器的顺序存放

实现)的那个连接 Proxy 分配给它(实际是先做连接是否有效的判断,如果可用就分配给用户,如不可用就把这个连接从空闲池删掉,重新检测空闲池是否还有连接);如果在空闲池没有连接 Proxy,则检查当前活动的连接 Proxy 池是否达到连接池所允许的最大连接数(maxConn),如果没有达到,就新建一个连接 Proxy,并把这个连接 Proxy 放到活动连接池中。如果已经达到,就等待一定的时间(timeout)。如果在等待的时间内有连接被释放出来就可以把这个连接 Proxy 分配给等待的用户,如果等待时间超过预定时间 timeout,则返回空值(null)。系统对已经分配出去正在使用的连接 Proxy 只做计数,当使用完后再返还给空闲池。对于空闲连接 Proxy 的状态,可开辟专门的线程定时检测,这样会花费一定的系统开销,但可以保证较快的响应速度。也可采取不开辟专门线程、只是在分配前检测的方法。

下面分别从几个方面来说明 SimpleDataSource 类的实现模式。

1. SimpleDataSource 类的初始化

SimpleDataSource 构造中把相关的参数传入,代码如下(屏蔽掉一些非核心代码):

```
public SimpleDataSource(Map props) {
    initialize(props);
}
```

在构造方法中调用 initialize 方法,代码如下:

```
private void initialize(Map props) {
    try {
        String prop_pool_ping_query = null;

        if (props == null) {
            throw new RuntimeException("SimpleDataSource: The properties map passed to the initializer was null.");
        }

        //对 jdbcDriver、jdbcUrl、jdbcUsername、jdbcPassword 赋值

        if (!(props.containsKey(PROP_JDBC_DRIVER)
            && props.containsKey(PROP_JDBC_URL)
            && props.containsKey(PROP_JDBC_USERNAME)
            && props.containsKey(PROP_JDBC_PASSWORD))) {
            throw new RuntimeException("SimpleDataSource: Some properties were not set.");
        } else {
            jdbcDriver = (String) props.get(PROP_JDBC_DRIVER);
            jdbcUrl = (String) props.get(PROP_JDBC_URL);
            jdbcUsername = (String) props.get(PROP_JDBC_USERNAME);
            jdbcPassword = (String) props.get(PROP_JDBC_PASSWORD);

            //对 pool 的相关参数赋值
            poolMaximumActiveConnections =
```

```

        props.containsKey(PROP_POOL_MAX_ACTIVE_CONN)
        ? Integer.parseInt((String) props.get(PROP_POOL_MAX_ACTIVE_CONN))
        : 10;

        .....

        //判断是否有用户的驱动程序, 如果有, 装载到 Properties 变量 driverProps 中
        useDriverProps = false;
        Iterator propIter = props.keySet().iterator();
        driverProps = new Properties();
        driverProps.put("user", jdbcUsername);
        driverProps.put("password", jdbcPassword);
        while (propIter.hasNext()) {
            String name = (String) propIter.next();
            String value = (String) props.get(name);
            if (name.startsWith(ADD_DRIVER_PROPS_PREFIX)) {
                driverProps.put(name.substring(ADD_DRIVER_PROPS_PREFIX_LENGTH), value);
                useDriverProps = true;
            }
        }

        expectedConnectionTypeCode = assembleConnectionTypeCode(jdbcUrl, jdbcUsername,
        jdbcPassword);

        //这是核心代码, 根据 JDBC 驱动名称, 实例化数据库驱动的对象, 如同 Class.forName
        (jdbcDriver);
        Resources.instantiate(jdbcDriver);
    }
    } catch (Exception e) {
        log.error("SimpleDataSource: Error while loading properties. Cause: " +
        e.toString(), e);
        throw new RuntimeException("SimpleDataSource: Error while loading properties.
        Cause: " + e, e);
    }
}

```

在 SimpleDataSource 类内部有两个 ArrayList 变量, 主要用于记录 Connection 池中的 Connection 情况。idleConnections 保存已经闲置的 Connection, activeConnections 保存正在使用的 Connection。

```

private List idleConnections = new ArrayList();
private List activeConnections = new ArrayList();

public static class SimplePooledConnection implements InvocationHandler {
}

```

2. 两个核心方法

DataSource 接口的主要方法时, getConnection() 和 getConnection(String username, String password), SimpleDataSource 实现类在实现这两个方法的时候, 转化为 popConnection

方法，同时，对于外部对 Connection 的 close 方法，SimpleDataSource 也转化为 pushConnection 方法。所以，SimpleDataSource 有两个核心方法，一个是获取数据库连接，另一个是退出数据库连接。实现方法与数据结构的堆栈相类似。采用入栈方法和出栈方法来分析。入栈方法是 pushConnection 方法，出栈方法是 popConnection 方法。popConnection 方法的活动实现如图 8-18 所示。

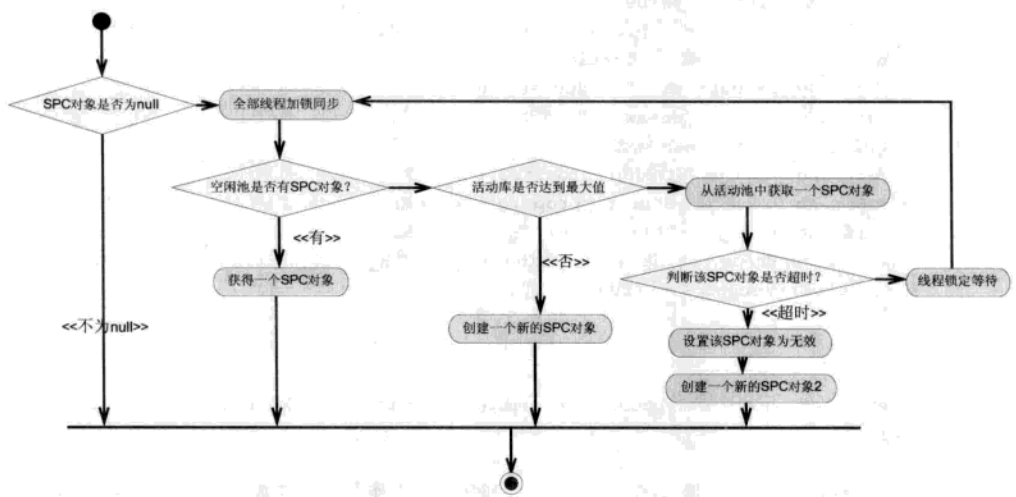


图 8-18 SimpleDataSource 类的 popConnection 方法实现图

在图 8-18 中，把 SimplePooledConnection 对象简称为 SPC 对象
popConnection 方法的程序代码如下：

```
private SimplePooledConnection popConnection(String username,String password)
throws SQLException {
    boolean countedWait = false;
    SimplePooledConnection conn = null;
    long t = System.currentTimeMillis();
    int localBadConnectionCount = 0;

    while (conn == null) {
        // 全部线程同步
        synchronized (POOL_LOCK) {
            if (idleConnections.size() > 0) {
                // Pool has available connection
                // 在连接池是否有空闲的 SimplePooledConnection 实例化对象
                conn = (SimplePooledConnection) idleConnections.remove(0);
                .....
            } else {
                // Pool does not have available connection
                // 空闲连接池没有可以使用的 connection
                if (activeConnections.size() < poolMaximumActiveConnections) {
                    // Can create new connection
```



```

// 创建一个新的 SimplePooledConnection 实例化对象
if (useDriverProps) {
    conn = new SimplePooledConnection(DriverManager.
        getConnection(jdbcUrl, driverProps), this);
} else {
    conn = new SimplePooledConnection(DriverManager.
        getConnection(jdbcUrl, jdbcUsername,
            jdbcPassword), this);
}
Connection realConn = conn.getRealConnection();
if (realConn.getAutoCommit() != jdbcDefaultAutoCommit) {
    realConn.setAutoCommit(jdbcDefaultAutoCommit);
}
.....
} else {
    // Cannot create new connection
    // 不能创建一个新的 SimplePooledConnection 实例化对象

    // 从活动连接池获取一个 SimplePooledConnection 实例化对象
    SimplePooledConnection oldestActiveConnection = (Simple
PooledConnection) activeConnections
        .get(0);
    long longestCheckoutTime = oldestActiveConnection
        .getCheckoutTime();

    // 判断 SimplePooledConnection 实例化对象是否超时

    if (longestCheckoutTime > poolMaximumCheckoutTime) {
        // Can claim overdue connection
        claimedOverdueConnectionCount++;
        accumulatedCheckoutTimeOfOverdueConnections +=
longestCheckoutTime;

        accumulatedCheckoutTime += longestCheckoutTime;
        activeConnections.remove(oldestActiveConnection);
        if (!oldestActiveConnection.getRealConnection()
            .getAutoCommit()) {
            oldestActiveConnection.getRealConnection().
rollback();
        }
        conn = new SimplePooledConnection(
            oldestActiveConnection,
                this);
        oldestActiveConnection.invalidate();
        .....
    } else {
        // Must wait
        // 进入等待时间
        try {
            if (!countedWait) {
                hadToWaitCount++;
            }
        } catch (InterruptedException e) {
            // Ignored
        }
    }
}

```



```

        countedWait = true;
    }
    .....

    // 在设定的时间内锁住线程, 不让程序执行下去
    long wt = System.currentTimeMillis();
    POOL_LOCK.wait(poolTimeToWait);
    accumulatedWaitTime += System.currentTimeMillis()

- wt;

        } catch (InterruptedException e) {
            break;
        }
    }
}

if (conn != null) {
    if (conn.isValid()) {
        // 对于超时的、或者无效的 SimplePooledConnection 实例化对象,
        // 对于内部的数据库 Connection 进行回滚
        if (!conn.getRealConnection().getAutoCommit()) {
            conn.getRealConnection().rollback();
        }
        conn.setConnectionTypeCode(assembleConnectionTypeCode(
            jdbcUrl, username, password));
        conn.setCheckoutTimestamp(System.currentTimeMillis());
        conn.setLastUsedTimestamp(System.currentTimeMillis());
        activeConnections.add(conn);
        requestCount++;
        accumulatedRequestTime += System.currentTimeMillis()
            - t;
    } else {
        .....
        badConnectionCount++;
        localBadConnectionCount++;
        conn = null;
        if (localBadConnectionCount > (poolMaximumIdleConnections
+ 3)) {
            .....
            throw new SQLException(".....");
        }
    }
}

if (conn == null) {
    .....
    throw new SQLException(".....");
}

```

```
return conn;
```

SimpleDataSource 类的 pushConnection 方法的活动实现如图 8-19 所示。主要实现向连接池中加入数据库 Connection。

图 8-19 中的把 SimplePooledConnection 对象简称为 SPC 对象。

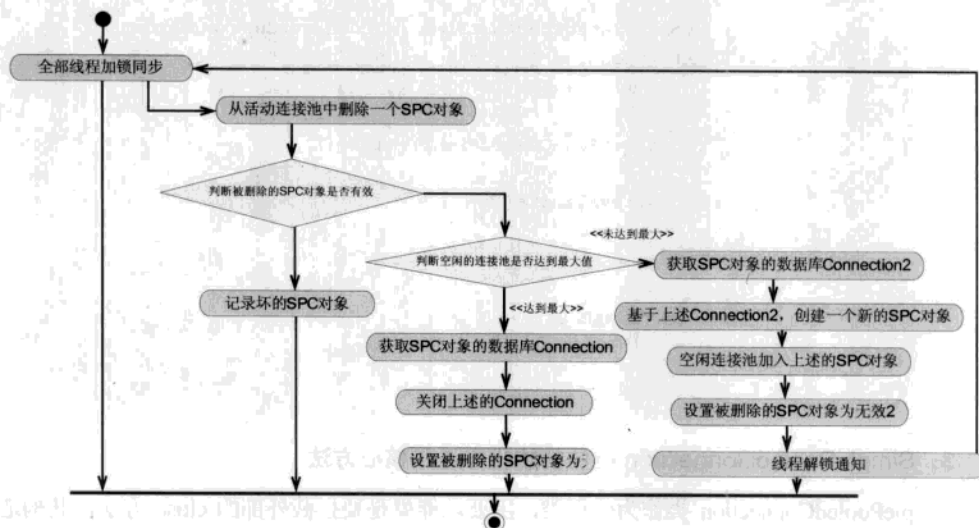


图 8-19 SimpleDataSource 类的 pushConnection 方法实现图

pushConnection 方法的程序代码如下。

```
private void pushConnection(SimplePooledConnection conn) throws SQLException {
    // 全部线程同
    synchronized (POOL_LOCK) {
        // 从活动连接池中删除一个 SimplePooledConnection 对象
        activeConnections.remove(conn);
        if (conn.isValid()) {
            // 当
            if (idleConnections.size() < poolMaximumIdleConnections
                && conn.getConnectionTypeCode() == getExpectedConnection
                TypeCode()) {
                accumulatedCheckoutTime += conn.getCheckoutTime();
                if (!conn.getRealConnection().getAutoCommit()) {
                    conn.getRealConnection().rollback();
                }
                SimplePooledConnection newConn = new SimplePooledConnection(
                    conn.getRealConnection(), this);
            }
        }
    }
}
```

```
        idleConnections.add(newConn);
        newConn.setCreatedTimestamp(conn.getCreatedTimestamp());
        newConn.setLastUsedTimestamp(conn.getLastUsedTimestamp());
        conn.invalidate();
        .....

        // 当可以获得新的 SimplePooledConnection 对象, 解锁程序
        POOL_LOCK.notifyAll();
    } else {
        accumulatedCheckoutTime += conn.getCheckoutTime();
        if (!conn.getRealConnection().getAutoCommit()) {
            conn.getRealConnection().rollback();
        }
        conn.getRealConnection().close();
        .....
        conn.invalidate();
    }
} else {
    .....
    badConnectionCount++;
}
}
```

3. SimplePooledConnection 类的初始化方法和核心方法

SimplePooledConnection 类作为代理类, 主要功能就是要拦截外部的 close 方法, 其构造方法是用于创建代理的数据库连接。其核心方法主要是 invoke 方法。SimplePooledConnection 内部类的属性说明如表 8-5 所示。

表 8-5 SimplePooledConnection 内部类的属性列表

属性名称	类 型	功能和用途	备 注
IFACES	Class[]	用于生成动态代理的 Connection 对象而定义的 Connection 类	
hashCode	private int	/本 SimplePooledConnection 的哈希码	
dataSource	SimpleDataSource	当前 SimplePooledConnection 对象对应的 SimpleDataSource 实例化对象	
realConnection	Connection	真正的数据库连接	
proxyConnection	Connection	代理的数据库连接	
checkoutTimestamp	long	检查的时间点	
createdTimestamp	long	创建时间点	
lastUsedTimestamp	long	最后使用的时间点	
connectionTypeCode	int	//	
valid	boolean	判断当前 SimplePooledConnection 对象是否有效	

由于 SimplePooledConnection 内部类是 java.sql.Connection 接口的代理实现类, 必须实现 java.sql.Connection 接口方法。所以, SimplePooledConnection 内部类的方法主要由三类组成。第 1 类是实现 InvocationHandler 接口的一个方法, 即 invoke 方法。第 2 类是实现

java.sql.Connection 接口的方法, 包括 close()、commit()、createStatement()、getAutoCommit() 等, 第 3 类是 SimplePooledConnection 内部类自有的方法。

SimplePooledConnection 类的构造方法如下。

```
public SimplePooledConnection(Connection connection, SimpleDataSource dataSource) {
    this.hashCode = connection.hashCode();
    this.realConnection = connection;
    this.dataSource = dataSource;
    this.createdTimestamp = System.currentTimeMillis();
    this.lastUsedTimestamp = System.currentTimeMillis();
    this.valid = true;

    proxyConnection = (Connection) Proxy.newProxyInstance(
        Connection.class.getClassLoader(), IFACES, this);
}
```

在构造方法中, 首先对 SimplePooledConnection 对象的属性进行初始化。关键是创建了一个动态代理对象。

SimplePooledConnection 的核心 invoke 方法的代码实现如下。

```
public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {
    String methodName = method.getName();
    if (CLOSE.hashCode() == methodName.hashCode() && CLOSE.equals(methodName)) {
        dataSource.pushConnection(this);
        return null;
    } else {
        try {
            return method.invoke(getValidConnection(), args);
        } catch (Throwable t) {
            throw ClassInfo.unwrapThrowable(t);
        }
    }
}
```

实现了拦截 close 方法, 转去调用 SimpleDataSource 的 pushConnection 方法。

对于 SimplePooledConnection 实现 java.sql.Connection 接口的方法, 包括 close()、commit()、createStatement()、getAutoCommit() 等, 除了 close 方法的代码如下。

```
public void close() throws SQLException {
    dataSource.pushConnection(this);
}
```

其余的基本上都是调用 getValidConnection() 来获得真正的数据库 Connection, 然后再对真正的数据库 Connection 执行对应的方法, 例如 createStatement() 方法, 代码如下。

```
public Statement createStatement() throws SQLException {
    return getValidConnection().createStatement();
}
```

4. 通过线程锁机制来控制多线程应用

通过用线程的锁机制来实现 Connection 池中 Connection 的获取。当要获取一个 SimplePooledConnection 对象时，看看 idleConnections 列表中是否有空闲的 SimplePooledConnection 对象，如果有，那么就获得这个 SimplePooledConnection 对象。如果没有，就到 activeConnections 看看是否已经到达 Connection 池的最大容量，如果还没有到达，那就可以创建一个新的 SimplePooledConnection 对象并返回。如果已经到达了 Connection 池的最大容量，那么就从 activeConnections 列表中获取一个 SimplePooledConnection 对象，看看它是否超时，如果它已经超时，那就新建一个 SimplePooledConnection 对象，并把从 activeConnections 列表获取的这个 SimplePooledConnection 对象的真正 Connection 转移给新建的这个 SimplePooledConnection 对象。同时让 activeConnections 列表的 SimplePooledConnection 对象无效。如果从 activeConnections 列表获得的 SimplePooledConnection 对象没有超时，那就按照连接池设定的时间等待，并且让后面的全部停止。

这些都是通过变量 POOL_LOCK 来进行控制的。POOL_LOCK 在 SimpleDataSource 类的赋值。

```
private final Object POOL_LOCK = new Object();
```

POOL_LOCK 进行线程控制主要通过两个方法。一个是获取的 popConnection 方法，一个是把 SimplePooledConnection 对象存储起来的方法。

```
private SimplePooledConnection popConnection(String username,String password)
throws SQLException {
    .....
    while (conn == null) {
        //进入线程的并发状态
        synchronized (POOL_LOCK) {
            .....
            //在设定的时间内锁住线程，不让程序执行下去
            POOL_LOCK.wait(poolTimeToWait);
            .....
        }
        .....
    }
}
```

这个是进行线程的释放，主要是有闲置的 SimplePooledConnection 对象。

```
private void pushConnection(SimplePooledConnection conn) throws SQLException {
    .....
    synchronized (POOL_LOCK) {
        .....
        //当可以获得新的 SimplePooledConnection，解锁程序。
        POOL_LOCK.notifyAll();
        ... ..
    }
}
```

5. 通过动态代理机制实现对 Connection 的控制

SimpleDataSource 类有一个静态类 SimplePooledConnection, 该静态类是一个动态代理类。实现的方法都是通过这个代理类来进行转化。

让我们来看看内部静态类 SimplePooledConnection 类的情况, 首先是 SimplePooledConnection 类的主要私有变量及其含义。

```
public static class SimplePooledConnection implements InvocationHandler {

    //这是一个类组变量, 在后面形成代理类要调用
    private static final Class[] IFACES = new Class[] { Connection.class };

    private SimpleDataSource dataSource; //这是内部的 SimpleDataSource 对象
    private Connection realConnection; //这是真实的数据库连接
    private Connection proxyConnection; //这是代理的数据库连接
    private long checkoutTimestamp; // proxyConnection 时间
    private long createdTimestamp; // proxyConnection 创建时间
    private long lastUsedTimestamp; // proxyConnection 使用时间
    private boolean valid; //判断 proxyConnection 是否有效
```

我们再来看看 SimplePooledConnection 类的构造函数, 传入的参数是一个真正的数据库 Connection 和 SimpleDataSource 对象。根据传入的参数, 创建一个动态 Proxy。

```
public SimplePooledConnection(Connection connection, SimpleDataSource dataSource) {
    this.hashCode = connection.hashCode();
    this.realConnection = connection;
    this.dataSource = dataSource;
    this.createdTimestamp = System.currentTimeMillis();
    this.lastUsedTimestamp = System.currentTimeMillis();
    this.valid = true;

    proxyConnection = (Connection) Proxy.newProxyInstance(
        Connection.class.getClassLoader(), IFACES, this);
}
```

这个动态 Proxy 基于 java.sql.Connection 接口并代理当前实例化的 SimplePooledConnection 对象。所以, proxyConnection 可以实现 java.sql.Connection 接口的所有方法。但是这些方法的具体实现, 要通过对 SimplePooledConnection 的操作来完成。其操作代码如下。

```
public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {
    String methodName = method.getName();
    if (CLOSE.hashCode() == methodName.hashCode() && CLOSE.equals(methodName)) {
        dataSource.pushConnection(this);
        return null;
    } else {
        try {
```



```

        return method.invoke(getValidConnection(), args);
    } catch (Throwable t) {
        throw ClassInfo.unwrapThrowable(t);
    }
}
}

```

SimplePooledConnection 对象调用 getValidConnection() 获得真正的数据库 Connection 对象, 也就是初始化传入的那个数据库 Connection 对象, 并通过 method.invoke 来执行。在这里还可以看到, 当外部调用的方法是 close 方法, 本对象并不调用真正的数据库 Connection 对象的 close 方法, 而是调用了 dataSource.pushConnection(this) 方法, 把 SimplePooledConnection 对象从连接池的 activeConnections 列表中转移到 idleConnections 列表中。

8.5 SQL Map 扩展 DataSource 为 C3P0

C3P0 是一个开放源代码的 JDBC 连接池, 包括了实现 jdbc3 和 jdbc2 扩展规范说明的 Connection 和 Statement 池的 DataSourcees 对象。该项目主页: <http://sourceforge.net/projects/c3p0>。

要在 iBATIS 中要扩展 DataSourcees, 让其支持 C3P0, 可以参照其他 DataSource 策略来进行编写。

扩展 C3P0 策略必须要增加 C3p0DataSourceFactory 工厂类和 C3p0Configuration 实现类。C3p0DataSourceFactory 是用于配合配置信息的解析和 Transaction 接口的调用, C3p0DataSourceFactory 工厂类必须实现 DataSourceFactory 接口。C3p0Configuration 实现类是具体产生 DataSourcees 对象。

C3p0DataSourceFactory 中必须要配置的参数例子如下:

```

<sqlMapConfig>
  <properties resource="properties/database.properties" />
  <typeAlias alias="C3P0"
    type="com.ibatis.sqlmap.engine.datasource.C3p0DataSourceFactory" />

  <transactionManager type="JDBC">
    <dataSource type="C3P0">
      <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="com/ibatis/jpetstore/persistence/sqlmapdao/sql/Account.xml" />

```



```
</sqlMapConfig>
```

由于动态地增加了 C3p0DataSourceFactory 工厂类, 要把这个信息告诉给 iBATIS 的解析程序, 所以增加了一个 typeAlias。同时对于 transactionManager 节点下的 dataSource 节点, 其属性要修改为定义的 Alias, 上述的是 C3P0。

让我们来看看 C3p0DataSourceFactory 工厂类的 initialize 方法。

```
public void initialize(Map map) {
    C3p0Configuration C3p0 = new C3p0Configuration(map);
    dataSource = C3p0.getDataSource();
}
```

实例化一个 C3p0Configuration 对象, 我们来看一下 C3p0Configuration 的构造方法。

```
public C3p0Configuration(Map properties) {
    try {
        dataSource = buildDataSource(properties);
    } catch (Exception e) {
        throw new RuntimeException( "Error initializing C3p0Configuration.
Cause: " + e, e);
    }
}
```

然后查看 buildDataSource 方法。

```
//根据配置信息, 创建一个 c3p0 的 DataSource
private DataSource buildDataSource(Map map) throws Exception {
    DataSource thisDataSource = null;
    String jdbcDriver = null;
    String jdbcUrl = null;
    String username = null;
    Properties connectionProps = new Properties();

    try {
        if (map.containsKey("JDBC.Driver")) {
            //对配置信息进行初始化处理, 并判断是否有错
            jdbcDriver = (String) map.get("JDBC.Driver");
            try {
                Class.forName(jdbcDriver);
            } catch (ClassNotFoundException cnfe) {
                throw new Exception("Could not instantiate C3P0 jdbcDriver", cnfe);
            }

            if (map.get("JDBC.ConnectionURL") == null) {throw new Exception
("jdbcUrl is null.");}
            if (map.get("JDBC.Username") == null) {throw new Exception
("Username is null.");}

            jdbcUrl = (String) map.get("JDBC.ConnectionURL");
            username = (String) map.get("JDBC.Username");
        }
    }
```

```

String password = (String) map.get("JDBC.Password");

connectionProps.setProperty("user", username);
connectionProps.setProperty("password", password);

//初始化连接池的信息
PoolConfig poolConfig = new PoolConfig();

if (notEmpty((String) map.get("Pool.MaximumActiveConnections"))) {
    poolConfig.setMaxPoolSize(Integer.valueOf((String) map.
        get("Pool.MaximumActiveConnections")));
} else {
    poolConfig.setMaxPoolSize(20);
}

if (notEmpty((String) map.get("Pool.MaximumWait"))) {
    poolConfig.setMaxIdleTime(Integer.valueOf((String) map.
        get("Pool.MaximumWait")));
} else {
    poolConfig.setMaxIdleTime(60000);
}

//获取一个没有连接池的 DataSource
DataSource unpooled = DataSources.unpooledDataSource(jdbcUrl,
connectionProps);

// 获取一个有连接池的 DataSource
thisDataSource = DataSources.pooledDataSource(unpooled, pool
Config);

} else {
    throw new Exception("JDBC.Driver is null.");
}
} catch (Exception e) {
    throw new Exception("Could not instantiate C3P0 connection pool", e);
}
return thisDataSource;
}

```

在这里根据配置文件的信息，创建一个 C3p0 的 DataSource 对象。

本扩展 DataSource 的源程序代码可参见光盘，且该代码已经通过了测试。

8.6 SQL Map 如何进行批处理

要在 JDBC 中进行批处理，首先要定义一个 Statement 对象，然后通过 addBatch 给对象加入要批处理的内容，然后通过调用对象的 executeBatch 方法来执行批处理。代码列出如下：

```

Connection conn = DriverManager.getConnection(URL, USER, PASS)
Statement stmt = conn.createStatement();

```

```

    stmt.addBatch("insert into authors(firstName , lastName) values('fegor',
    'hack')");
    stmt.addBatch("insert into authors(firstName , lastName) values('fegors',
    'hacks')");
    stmt.addBatch("insert into authors(firstName , lastName) values('fegorsr',
    'hacksr')");
    stmt.executeBatch();
    stmt.close();
    conn.close();

```

这是针对没有参数的 SQL，如果 SQL 带有参数，则标准代码如下。

```

Connection conn = DriverManager.getConnection(URL,USER,PASS);
PreparedStatement ps=conn.prepareStatement("insert into authors(firstName, lastName)
values(?,?)");

    ps.setString(1,"string1");//设置参数
    ps.setString(2," string 2");
    ps.addBatch();           //把语句加入批处理队列

    ps.setString(1," string3");
    ps.setString(2," string4");
    ps.addBatch();

    ps.setString(1," string5");
    ps.setString(2," string6");
    ps.addBatch();

    ps.executeBatch();      //执行批处理

    ps.close();             //最后关闭
    conn.close();

```

从上面的例子中，JDBC 使用批处理功能涉及两个方法：一个是 `addBatch(String)` 方法。该方法可以接受一段标准的 SQL（如果使用一个 `Statement`）作为参数，也可以什么参数都不带。另一个是 `executeBatch` 方法，就是直接执行批处理。

JDBC 使用批处理也涉及两种 `Statement` 对象和 `PreparedStatement` 对象。如果是直接执行 SQL，就可以采用 `Statement` 类来处理，如果是涉及传入参数，则要用 `PreparedStatement`。

iBATIS 平台的批处理就是采用 `PreparedStatement` 来进行处理的。

在 `SQL Executor` 类中有一个静态类 `Batch`，主要是处理批处理操作。在静态类 `Batch` 中的属性如表 8-6 所示。

表 8-6 静态类 `Batch` 中的属性列表

属性名称	类 型	功能和用途	备 注
current Sql	private String	当前执行的 SQL 语句	
statementList	private ArrayList	要求批处理执行的 SQL 语句列表	
batchResultList	private ArrayList	要求批处理执行的 SQL 语句的返回结果列表	
size	private int	要求处理批处理执行的个数	

静态类 Batch 中的方法如表 8-7 所示。

表 8-7 静态类 Batch 中的方法列表

方法名称	参 数	返 回 值	功能和用途	备 注
addBatch	StatementScope Connection, String SQL, Object[]	无	增加批处理内容	
executeBatch	无	int	执行所有当前的批处理操作	

静态类 Batch 的两个方法，addBatch 和 executeBatch，分别介绍如下。

addBatch 方法主要是增加批处理内容，其代码如下。

```
public void addBatch(StatementScope statementScope, Connection conn, String sql,
Object[] parameters)
    throws SQLException {
    PreparedStatement ps = null;
    if (currentSql != null && currentSql.equals(sql)) {
        int last = statementList.size() - 1;
        ps = (PreparedStatement) statementList.get(last);
    } else {
        ps = prepareStatement(statementScope.getSession(), conn, sql);
        setStatementTimeout(statementScope.getStatement(), ps);
        currentSql = sql;
        statementList.add(ps);
        batchResultList.add(new BatchResult(statementScope.getStatement().getId(),
sql));
    }
    statementScope.getParameterMap().setParameters(statementScope, ps, parameters);
    ps.addBatch();
    size++;
}
```

executeBatch 是执行所有当前的批处理操作，其代码如下。

```
public int executeBatch() throws SQLException {
    int totalRowCount = 0;
    for (int i = 0, n = statementList.size(); i < n; i++) {
        PreparedStatement ps = (PreparedStatement) statementList.get(i);
        int[] rowCounts = ps.executeBatch();
        for (int j = 0; j < rowCounts.length; j++) {
            if (rowCounts[j] == Statement.SUCCESS_NO_INFO) {
                // do nothing
            } else if (rowCounts[j] == Statement.EXECUTE_FAILED) {
                throw new SQLException("The batched statement at index " + j + " failed
to execute.");
            } else {
                totalRowCount += rowCounts[j];
            }
        }
    }
}
```

```
    }  
    return totalRowCount;  
}
```

其实我们从源码分析，iBATIS 并没有采用针对一个 Statement 对象加入批处理，而是对不同 Statement 对象加入批处理。当执行批处理的时候，同时执行这些所有 Statement 对象的批处理操作。

8.7 SQL Map 事务隔离的实现

SQL Map 的事务隔离还是遵循 JDBC 事务隔离规则，先简单介绍一下 JDBC 事务隔离。

8.7.1 JDBC 事务隔离概述

为了解决与“多个线程请求相同数据”相关的问题，事务之间用锁相互隔开。多数主流的数据库支持不同类型的锁；因此，JDBC API 支持不同类型的事务，它们由 Connection 对象指派或确定。在 JDBC API 支持五个事务的隔离级别，如表 8-8 所示。

表 8-8 JDBC 事务隔离级别

序 号	JDBC 事务隔离级别等级	功能或用途描述
1	TRANSACTION_NONE	说明不支持事务
2	TRANSACTION_READ_UNCOMMITTED	说明在提交前一个事务可以看到另一个事务的变化。这样脏读、不可重复的读和虚读都是允许的
3	TRANSACTION_READ_COMMITTED	说明读取未提交的数据是不允许的。这个级别仍然允许不可重复的读和虚读产生
4	TRANSACTION_REPEATABLE_READ	说明事务保证能够再次读取相同的数据而不会失败，但虚读仍然会出现
5	TRANSACTION_SERIALIZABLE	是最高的事务级别，它防止脏读、不可重复的读和虚读

为了在性能与一致性之间寻求平衡才出现了上面的几种级别。事务保护的级别越高，性能损失就越大。假定数据库和 JDBC 驱动程序支持这个特性，则给定一个 Connection 对象，可以明确地设置想要的事务级别。

```
conn.setTransactionLevel(TRANSACTION_SERIALIZABLE);  
    可以通过下面的方法确定当前事务的级别：  
    int level = conn.getTransactionIsolation();  
    if(level == Connection.TRANSACTION_NONE)  
        System.out.println("TRANSACTION_NONE");  
    else if(level == Connection.TRANSACTION_READ_UNCOMMITTED)  
        System.out.println("TRANSACTION_READ_UNCOMMITTED");  
    else if(level == Connection.TRANSACTION_READ_COMMITTED)  
        System.out.println("TRANSACTION_READ_COMMITTED");  
    else if(level == Connection.TRANSACTION_REPEATABLE_READ)
```

```

System.out.println("TRANSACTION REPEATABLE READ");
else if(level == Connection.TRANSACTION_SERIALIZABLE)
System.out.println("TRANSACTION_SERIALIZABLE");

```

以上的五个事务隔离级别都是在 `Connection` 类中定义的静态常量，使用 `setTransactionIsolation(int level)` 方法可以设置事务隔离级别。

8.7.2 SQL Map 的事务隔离的实现

为了实现 JDBC 中的事务隔离，在 iBATIS SQL Map 平台中有一个 `IsolationLevel` 类，负责实现事务隔离。默认的事务级别是-9999。`IsolationLevel` 类有三个方法，如表 8-9 所示。

表 8-9 `IsolationLevel` 类的方法列表

方法名称	参 数	返 回 值	功能和用途	备 注
<code>setIsolationLevel</code>	<code>int isolationLevel</code>	无	设置事务隔离级别	
<code>applyIsolationLevel</code>	<code>Connection conn</code>	无	设置数据库 <code>Connection</code> 的事务隔离级别	
<code>restoreIsolationLevel</code>	<code>Connection conn</code>	无	恢复数据库 <code>Connection</code> 的事务隔离级别	

外部调用时，只调用 `IsolationLevel` 的 `setIsolationLevel` 方法，即可把事务级别设置为外部事务需要的事务级别。然后调用 `applyIsolationLevel` 方法，把数据库 `Connection` 的事务级别变成可以应用。`applyIsolationLevel` 方法的实现代码如下。

```

public void applyIsolationLevel(Connection conn) throws SQLException {
    if (isolationLevel != UNSET_ISOLATION_LEVEL) {
        originalIsolationLevel = conn.getTransactionIsolation();
        if (isolationLevel != originalIsolationLevel) {
            conn.setTransactionIsolation(isolationLevel);
        }
    }
}

```

`restoreIsolationLevel` 是恢复事务隔离级别为默认状态。

```

public void restoreIsolationLevel(Connection conn) throws SQLException {
    if (isolationLevel != originalIsolationLevel) {
        conn.setTransactionIsolation(originalIsolationLevel);
    }
}

```

外部如果想使用 Java 的事务隔离，可以创建 `IsolationLevel` 对象，然后对 `IsolationLevel` 进行赋值操作。如 JDBC 的调用方法如下。

```

private IsolationLevel isolationLevel = new IsolationLevel();

public JdbcTransaction(DataSource ds, int isolationLevel) throws TransactionException {
    // Check Parameters
    dataSource = ds;
}

```



```

if (dataSource == null) { throw new TransactionException(".....");}
this.isolationLevel.setIsolationLevel(isolationLevel);
}

```

8.8 SQL Map 事务状态的实现

SQL Map 平台中有一个事务状态变量类 `TransactionState`，其功能主要就是对 SQL Map 的事务处理状态进行分类。`TransactionState` 类结构如图 8-20 所示。

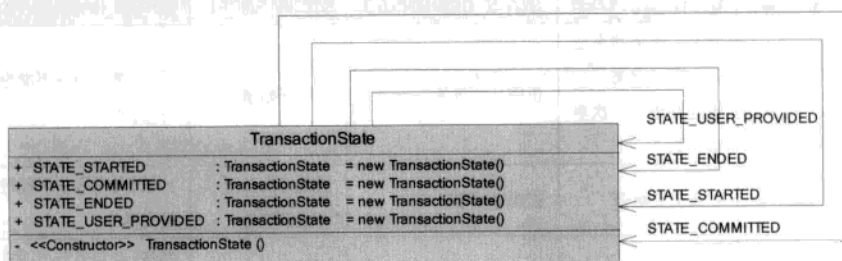


图 8-20 `TransactionState` 类结构图

这是一种自身关联（反身关联）：自己引用自己，带着一个自己的引用。就是在自己的内部有着一个自身的引用。`TransactionState` 的程序代码如下。

```

public class TransactionState {

    public static final TransactionState STATE_STARTED = new TransactionState();
    public static final TransactionState STATE_COMMITTED = new TransactionState();
    public static final TransactionState STATE_ENDED = new TransactionState();
    public static final TransactionState STATE_USER_PROVIDED = new TransactionState();

    private TransactionState() {
    }
}

```

首先是定义了四种事务状态。`STATE_STARTED` 表示事务已经开始。`STATE_COMMITTED` 表示事务提交。`STATE_ENDED` 表示事务已经结束。`STATE_USER_PROVIDED` 表示是外部事务时。其中当该事务状态是外部事务时，其他状态基本上都不起作用。外部事务主要是在 `TransactionManager` 管理和控制事务状态。

`TransactionManager` 是事务控制的执行体。`TransactionManager` 在执行事务处理的时候，首先要判断当前事务状态，其状态如图 8-21 所示。

具体代码可以参看 `TransactionManager` 的 `begin`、`commit`、`end` 方法。表 8-10 说明 `TransactionManager` 执行事务方法 `begin`、`commit`、`end` 时，根据其当前的事务状态而采取的下一步操作。



图 8-21 TransactionManager 的状态实现图

表 8-10 TransactionManager 的状态操作方法

TransactionManager	STATE_STARTED	STATE_COMMITTED	STATE_ENDED	STATE_USER_PROVIDED
begin	异常。事务状态已经是开始状态了，不能在在进行二次事务开始	正确开始事务	正确开始事务	异常。外部事务不能进行事务开始操作
commit	正确提交事务	正确提交事务	异常。没有可以使用的事务	异常。外部事务不能进行事务提交操作
end	正确结束事务	正确结束事务	正确结束事务	异常。外部事务不能进行事务结束操作

8.9 读取源码的收获

通过我们对 SQL Map 数据库处理源码的理解和分析，应该有这几个收获。

第一，了解 Java 的事务类型，知道 Java 的事务类型包括 JDBC、JTA 和容器事务，以及这些事务的原生态是如何实现的。

第二，采用桥梁模式把事务的配置信息和事务的实现分离开来。这样一方面可以实现配置文件中获取不同的事务配置信息，另一方面也可以针对性地实现 JDBC、JTA 和外部事务。事务配置信息和事务实现独立操作，可以在各自的领域内进行扩展而不影响到两者的合作关系。这种设计模式和实现方式值得我们学习。

第三，SQL Map 的 DataSource 策略采用了工厂方法模式，同时也结合了动态类加载技术。这样可以保证软件系统的稳定性、延续性和扩张性。我们在学习这种设计策略的同时，也给 SQL Map 的 DataSource 策略增加了一种新的 DataSource 策略——C3p0。这也说明这种设计方式在延续性和扩张性的优势。

第四，如何编写一个 javax.sql.DataSource 接口的实现类。在 SQL Map 中实现了一个自定义的 SimpleDataSource 实现类。在分析和理解这个实现类的过程中，我们也对 DataSource 接口的方法和属性有了一个较深入的了解。同时应该在这几个方面也有所体会：

- ① 如何进行一个 DataSource 中 Connection 池容器的创建、包括这个 Connection 池中 Connection 的分配及释放。
- ② Connection 池中如何进行并发处理，系统采用多线程的锁机制来控制 and 协调多线程应用的 wait 和 notify 方法，采用线程同步方法来实现它们之间的和

诸并避免死锁。③ 通过动态代理的拦截器技术来屏蔽 Connection 接口的 close 方法，这样可以达到对 Connection 池中的 Connection 进行动态控制。④ 通过内部类来实现动态代理。

第五，Java 事务隔离有五种类型。iBATIS SQL Map 是如何实现事务隔离？

第六，SQL Map 平台中有四种事务状态变量。首先采用自关联模式实现这四种事务状态，同时还采用了状态设计模式来对事务进行控制。

第 9 章

SQL Map 中 Mapping 实现

本章内容:

1. ParameterMap 的实现框架及其实现说明。包括 ParameterMap 总体框架、ParameterMap 组件覆盖的内容, ParameterMap 类和 ParameterMapping 类的内部实现, 以及 ParameterMap 如何融入到 SQL Map 中 Mapping 的实现。

2. resultMap 的实现框架及其实现说明。包括 resultMap 总体框架、resultMap 组件覆盖的内容, resultMap 类和 resultMap 类的内部实现, 以及 resultMap 如何融入到 SQL Map 中 Mapping 的实现。

3. Statement 框架及其说明。包括 Statement 总体框架、Statement 组件覆盖的内容, MappedStatement 是如何工作的, Statement 缓存的实现, Statement 自动生成的主键的实现, 以及 Statement 如何结合 ParameterMap 和 resultMap 并实现 SQL Map 中 Mapping。

4. Sql 框架及其说明。包括 Sql 框架、Sql 组件覆盖的内容, 静态 SQL 的实现, 简单动态 SQL 的实现和动态 SQL 语言的实现。

5. 数据对象转换框架及其说明。覆盖 DataExchange 组件和 Accessplan 组件的框架、作用、设计模式, 代码实现和如何融合 iBATIS 平台。

SQL Map 中的 Mapping 主要涉及三种 Map, 即 ParameterMap、resultMap 和 StatementMap。一个数据交换 DataExchange。对于 statement 类型比较复杂一点, 首先是 statement 类型包括几种类型, 其次是对 SQL 的处理, 因为对 SQL 的处理就会涉及上面的 parameterMap 和 resultMap。

本章从 ParameterMap 和 resultMap 进行说明, 接着讲述 StatementMap 和 SQL 框架, 最后说明一下数据对象转换框架。

9.1 ParameterMap 框架及其说明

9.1.1 ParameterMap 总体框架说明

ParameterMap 的基本思想是定义一系列有次序的参数系列，用于匹配 JDBC PreparedStatement 的值符号。例如：

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

<sqlMap namespace="Account">
  <parameterMap id="productParam" class="product">
    <parameter property="id" jdbcType="id" typeName="id" javaType="id"
resultMap = "id"
      nullValue="id" mode="id" typeHandler="id" numericScale="id"/>
  </parameterMap>
</sqlMap>

<statement id="insertProduct" parameterMap="insert-product-param">
  insert into PRODUCT (PRD_ID, PRD_DESCRIPTION) values (?,?);
</statement>
```

上面的例子中，parametermap 的两个参数按次序匹配 SQL 语句中的值符号（?）。因此，第一个“?”号将被“id”属性的值替换，而第二个“?”号将被“description”属性的值替换。

ParameterMap 比较简单，其内容主要在 com.ibatis.sqlmap.engine.mapping.parameter 包内。由 InlineParameterMapParser 类、NoParameterMap 类、ParameterMap 类、ParameterMapping 类这四个类组成。其中 ParameterMap 最重要。ParameterMap 结构如图 9-1 所示。

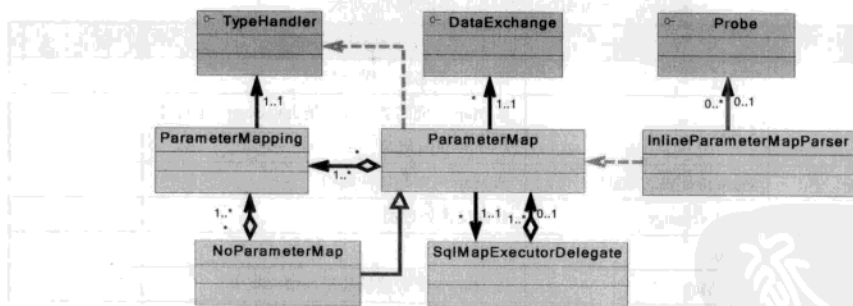


图 9-1 ParameterMap 类结构图

ParameterMap 类是本组件的核心，其关联 SqlMapExecutorDelegate 类，又被 SqlMapExecutorDelegate 类所聚合，这是一种双向关联关系。ParameterMap 类同时聚合 ParameterMapping 类，表明 ParameterMap 对象可以对应多个 ParameterMapping 对象。

ParameterMap 类也关联 DataExchange 接口,用于处理数据对象的转化。同时 ParameterMap 类依赖 TypeHandler 接口,主要是进行数据类型的转化和别名的翻译。NoParameterMap 类继承 ParameterMap 类同时又聚合 ParameterMapping 类,但在应用程序中没有什么实际意义。ParameterMapping 类关联 TypeHandler 接口,主要是进行数据类型的转化和别名的翻译。InlineParameterMapParser 类是一个工具类,其依赖 ParameterMap 类同时也关联 Probe 接口,Probe 接口用于处理 bean、DOM 对象和其他对象的转化。ParameterMap 组件相关接口和类说明如表 9-1 所示。

表 9-1 ParameterMap 组件相关接口和类说明

接 口 或 类	功能或用途描述	备注
com.ibatis.sqlmap.engine.mapping.parameter.ParameterMap	输入参数的包装类	
com.ibatis.sqlmap.engine.mapping.parameter.NoParameterMap	一个附加的参数的包装类,在系统中没有作用	
com.ibatis.sqlmap.engine.mapping.parameter.ParameterMapping	每个具体输入参数的包装类	
com.ibatis.sqlmap.engine.mapping.parameter.InlineParameterMapParser	语句解析的工具类	

InlineParameterMapParser 类不属于 ParameterMap 内容,它是一个独立的参数处理工具类。

9.1.2 ParameterMap 组件中各个类介绍

1. ParameterMap 类的介绍

这是把节点参数映射过来的数据类,属于数据存储类。其数据来源是 XML 映射文件的 parameterMap 节点,其对应关系如表 6-12 所示。ParameterMap 是一个数据存储类,其属性参数说明如表 9-2 所示。

表 9-2 ParameterMap 类的属性列表

属 性 名 称	类 型	功能和用途	备 注
id	String	该 Map 的名称,唯一标识	
parameterClass	Class	该 Map 对应的类,可能是 JavaBean,也可能是标准 Java 对象	
parameterMappings	ParameterMapping[]	该 Map 中参数属性数组	
dataExchange	DataExchange	数据交换处理类	
resource	String		
parameterMappingIndex	HashMap	该 Map 中参数属性数组的排序列表	
delegate	SqlMapExecutorDelegate		

2. NoParameterMap 类介绍

这是一个没有任何参数的 ParameterMap,其他内容全部与 ParameterMap 相同,也是属于数据存储类。

3. ParameterMapping 类介绍

这是一个存储 ParameterMap 参数的类。如上述中的例子，就会产生 2 个 ParameterMapping 实例对象，分别是 id 和 description。其数据来源是 XML 映射文件的 parameterMap 节点下的 parameter 节点，其对应关系如表 6-16 所示。ParameterMapping 类是一个数据存储类，其参数说明如表 9-3 所示。

表 9-3 ParameterMapping 类的属性列表

属性名称	类 型	功能和用途	备 注
propertyName	String	属性的名称，唯一标识	
typeHandler	TypeHandler	对应的 TypeHandler 变量	
typeName	String	用于自定义的 REF 类型	
jdbcType	int	该参数对应的 jdbcType 数值	
jdbcTypeName	String	该参数对应的 jdbcType 名称	
nullValue	String	是否空值	
mode	String	实现模式	
inputAllowed	boolean	是否允许输入	
outputAllowed	boolean	是否允许输出	
javaType	Class	对应的 javaType 类型	
resultMapName	String	对应的 resultMapName 变量	
numericScale	Integer		

4. InlineParameterMapParser 介绍

InlineParameterMapParser 支持 JavaBean 的属性名称嵌在 MappedStatement 的定义中（即直接写在 SQL 语句中）。默认情况下，任何没有指定 ParameterMap 的 MappedStatement 都会被解析成 Inline Parameter（内嵌参数）。InlineParameterMapParser 类是一个处理类。

InlineParameterMapParser 类主要就一个核心方法——parseInlineParameterMap，该方法实现的功能是处理 SQL 文本中含有#内字符的内容。其参数及其含义，typeHandlerFactory 表示可以找寻类名解释的工厂，sqlStatement 表示要处理的 SQL 语句，parameterClass 表示采用何种处理模式，有四种可以选择，分别为 DomTypeMarker、java.util.Map、TypeHandler 和 JavaBean。

9.1.3 ParameterMap 框架如何工作

当进入一个查询、修改或者删除的时候，要调用参数，就要使用本框架。

平台在 SQL Map 映射文件中的 parameter Map 节点的时候，会把实例化一个 ParameterMap 对象，最终保存在 SqlMapExecutorDelegate 实例化对象中 HashMap 类型的属性 parameterMaps。并且把 parameterMap 节点下的多个属性节点实例化成为 ParameterMapping 对象并统一赋值给 ParameterMap 对象的 parameterMappings 数组属性。

当用户调用外部查询和业务处理操作的时候，如果这些查询和业务处理操作在映射文件中要求有传入参数并有对应的 `ParameterMap` 名称，则直接可以从 `SqlMapExecutorDelegate` 的 `parameterMaps` 哈希变量组中获得对应的 id。如果这些查询和业务处理操作在映射文件中有要求有传入参数但是没有对应的 `ParameterMap` 名称，而只是 `ParameterObject`，那么系统会自动创建一个 `ParameterMap` 对象。

由于平台在处理过程中，会把 `ParameterMap` 对象放到 `StatementScope` 对象中，而 `StatementScope` 对象是贯穿全部处理过程的一个状态类。所以，在 `SqlExecutor` 对象进行 `executeUpdate` 方法和 `executeQuery` 方法的时候，都会调用下述方法。

```
statementScope.getParameterMap().setParameters(statementScope, ps, parameters);
```

实际上，调用上述方法就是调用 `ParameterMap` 的 `setParameters(statementScope, ps, parameters)` 方法，把参数赋值给 `PreparedStatement`。下面我们来看详细的程序代码。

```
public void setParameters(StatementScope statementScope, PreparedStatement ps,
    Object[] parameters)
    throws SQLException {
    .....
    if (parameterMappings != null) {
        for (int i = 0; i < parameterMappings.length; i++) {
            ParameterMapping mapping = parameterMappings[i];
            errorContext.setMoreInfo(mapping.getErrorString());
            if (mapping.isInputAllowed()) {
                setParameter(ps, mapping, parameters, i);
            }
        }
    }
}
```

上述代码根据有多少个参数，循环地调用 `setParameter` 方法。让我们来看看 `setParameter` 方法的代码。

```
protected void setParameter(PreparedStatement ps, ParameterMapping mapping, Object[]
parameters, int i) throws SQLException {
    Object value = parameters[i];
    // Apply Null Value
    String nullValueString = mapping.getNullValue();
    if (nullValueString != null) {
        TypeHandler handler = mapping.getTypeHandler();
        if (handler.equals(value, nullValueString)) {
            value = null;
        }
    }

    // Set Parameter
    TypeHandler typeHandler = mapping.getTypeHandler();
    if (value != null) {
        typeHandler.setParameter(ps, i + 1, value, mapping.getJdbcTypeName());
    }
}
```



```

    } else if (typeHandler instanceof CustomTypeHandler) {
        typeHandler.setParameter(ps, i + 1, value, mapping.getJdbcTypeName());
    } else {
        int jdbcType = mapping.getJdbcType();
        if (jdbcType != JdbcTypeRegistry.UNKNOWN_TYPE) {
            ps.setNull(i + 1, jdbcType);
        } else {
            ps.setNull(i + 1, Types.OTHER);
        }
    }
}
}

```

上述处理是通用处理，对于存储过程的处理有一些不同。我们来看看 `SqlExecutor` 对象的 `executeUpdateProcedure` 方法和 `executeQueryProcedure` 方法是如何来进行参数处理(主要列出核心代码)。

```

public void executeQueryProcedure(StatementScope statementScope, Connection conn,
String sql, Object[] parameters, int skipResults, int maxResults, RowHandlerCallback
callback) throws SQLException {
    .....
    CallableStatement cs = null;
    ResultSet rs = null;
    setupResultObjectFactory(statementScope);
    try {
        Integer rsType = statementScope.getStatement().getResultSetType();
        if (rsType != null) {
            cs = prepareCall(statementScope.getSession(), conn, sql, rsType);
        } else {
            cs = prepareCall(statementScope.getSession(), conn, sql);
        }
        setStatementTimeout(statementScope.getStatement(), cs);
        Integer fetchSize = statementScope.getStatement().getFetchSize();
        if (fetchSize != null) {
            cs.setFetchSize(fetchSize.intValue());
        }
        ParameterMap parameterMap = statementScope.getParameterMap();
        ParameterMapping[] mappings = parameterMap.getParameterMappings();
        registerOutputParameters(cs, mappings);
        parameterMap.setParameters(statementScope, cs, parameters);
        cs.execute();

        // Begin ResultSet Handling
        rs = handleMultipleResults(cs, statementScope, skipResults, maxResults,
callback);
        // End ResultSet Handling
        retrieveOutputParameters(statementScope, cs, mappings, parameters, callback);

    } finally {
        try {
            closeResultSet(rs);
        } finally {

```

```

        closeStatement(statementScope.getSession(), cs);
    }
}

private void registerOutputParameters(CallableStatement cs, ParameterMapping[]
mappings) throws SQLException {
    for (int i = 0; i < mappings.length; i++) {
        ParameterMapping mapping = ((ParameterMapping) mappings[i]);
        if (mapping.isOutputAllowed()) {
            if (null != mapping.getTypeName() && ! mapping.getTypeName().equals(""))
            { //@@added
                cs.registerOutParameter(i + 1, mapping.getJdbcType(), mapping.getTypeName());
            } else {
                if (mapping.getNumericScale() != null &&
                    (mapping.getJdbcType() == Types.NUMERIC || mapping.getJdbcType() == Types.DECIMAL))
                {
                    cs.registerOutParameter(i + 1, mapping.getJdbcType(), mapping.getNumeric
Scale().intValue());
                } else {
                    cs.registerOutParameter(i + 1, mapping.getJdbcType());
                }
            }
        }
    }
}

```

在处理过程中，由于存储过程可能会要求有输出 Output。所以在这里的处理是直接获得 ParameterMapping，然后进行输出的注册处理。

9.2 resultMap 框架及其说明

9.2.1 resultMap 框架介绍

在 SQL Map 框架中，ResultMap 在执行查询 MappedStatement 时，resultMap 负责将结果集的列值映射成 JavaBean 的属性值。其结构如下所示。

```

<resultMap id="productResult" class="product" extends="extends"
    xmlName="xmlName" groupBy="groupBy">
    <result property="id" nullValue="null" jdbcType="jdbcType"
        javaType="javaType" column="PRD_ID" columnIndex="int" select=""
        resultMap="name Of resultMap" typeHandler="" notNullColumn="" />
    <discriminator nullValue="" jdbcType="" javaType=""
        column="PRD_ID" columnIndex="" typeHandler="">
        <subMap value="" resultMap="" />
    </discriminator>
</resultMap>

```

resultMap 的 id 属性是 statement 的唯一标识。ResultMap 的 class 属性用于指定 Java 类的全限定名（即包括包的名称）。该 Java 类初始化并根据定义填充数据。extends 是可选的属性，可设定成另外一个 resultMap 的名字，并以它为基础来进行随后的操作。和在 Java 中继承一个类相似，父 resultMap 的属性将作为子 resultMap 的一部分。父 resultMap 的属性总是加到子 resultMap 属性的前面，并且父 resultMap 必须要在子 resultMap 之前定义。父 resultMap 和子 resultMap 的 class 属性不一定要一致，它们可以没有任何关系。

resultMap 可以包括任意多的属性映射，将查询结果集的列值映射成 JavaBean 的属性。属性的映射按它们在 resultMap 中定义的顺序进行。相关的 JavaBean 类必须符合 JavaBean 规范，每一属性都必须拥有 get/set 方法或者 is 方法。

9.2.2 resultMap 框架说明

ResultMap 的内容主要在 com.ibatis.sqlmap.engine.mapping.result 包内。由 ResultMap、ResultMapping 等类组成。其中 ResultMap 和 ResultMapping 最重要。ResultMap 组件类结构如图 9-2 所示。

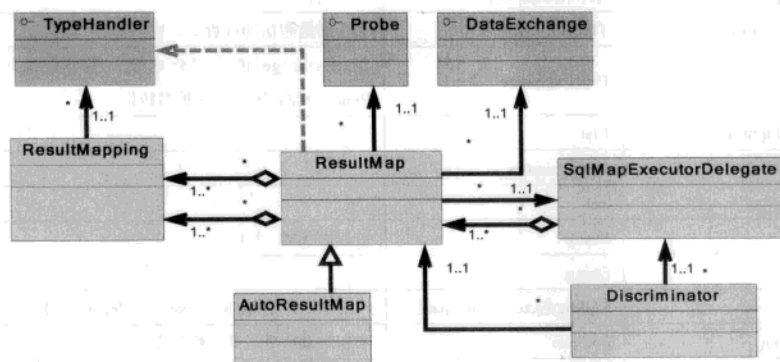


图 9-2 resultMap 的类结构图

ResultMap 类是本组件的核心，其关联 SqlMapExecutorDelegate 类，又被 SqlMapExecutorDelegate 类所聚合，这是一种双向关联关系。与 ParameterMap 和 SqlMapExecutorDelegate 的关系相同，ResultMap 类同时两次聚合 ResultMapping 类，表明 resultMap 对象可以对应多个 ResultMapping 对象。ResultMap 类也关联 DataExchange 接口，用于处理数据对象的转化。同时 resultMap 类依赖 TypeHandler 接口，主要是进行数据类型的转化和别名的翻译。ResultMapping 类关联 TypeHandler 接口，主要是进行数据类型的转化和别名的翻译。AutoResultMap 类依赖 resultMap 类。Discriminator 类关联 SqlMapExecutorDelegate 类和 resultMap 类，其主要作用是使用文件、设置、文件范本来更改模板。

ResultMap 组件相关接口和类说明如表 9-4 所示。

表 9-4 resultMap 组件相关接口和类说明

接 口 或 类	功能或用途描述	备注
com.ibatis.sqlmap.engine.mapping.result.ResultMap	输出结果的包装类	
com.ibatis.sqlmap.engine.mapping.result.AutoResultMap	针对简单对象的输出结果的包装类	
com.ibatis.sqlmap.engine.mapping.result.ResultMapping	输出结果中各个具体参数的包装类	
com.ibatis.sqlmap.engine.mapping.result.Discriminator	用文件、集合和文件模板去改变当面模板	

9.2.3 resultMap 中的类说明

1. resultMap 类说明

ResultMap 类内部属性如表 9-5 所示。

表 9-5 resultMap 类的属性和说明列表

属 性 名 称	类 型	功能和用途	备注
id	String	标识内码	
resultClass	Class	对应的类，可能是 JavaBean，也可能是 Map	
resultMappings	ResultMapping[]	参数属性数组	
remappableResultMappings	ThreadLocal	用于参数数组的线程安全性	
dataExchange	DataExchange	DataExchange 的实例化对象，主要处理与 Bean、XML 等对象的数据转化	
nestedResultMappings	List		
discriminator	Discriminator		
groupByProps	Set		
XMLName	String	对应的 XML 名称	
resource	String		
delegate	SqlMapExecutorDelegate	产生本对象的 SqlMapExecutorDelegate 对象	
allowRemapping	boolean	判断对参数数组采用线程安全性	
Object NO_VALUE	Object	判断是否有值	

2. ResultMapping 类说明

ResultMapping 类内部属性如表 9-6 所示。

表 9-6 resultMap 类的方法和说明列表

属 性 名 称	类 型	功能和用途	备注
propertyName	String	属性的名称，唯一标识	
columnName	String	对应数据库表的列名	
columnIndex	int	对应数据库表的索引号	
typeHandler	TypeHandler	对应 TypeHandler 的类型	
jdbcType	int	对应 jdbcType 的类型	
jdbcTypeName	String	对应 jdbcType 的名称	
nullValue	String	判断是否空值	

续表

属性名称	类 型	功能和用途	备注
notNullColumn	String	判断是否空列	
statementName	String	对应 statementName 的名称	
JavaType	Class	对应 Java 的数据类型	
nestedResultMapName	String		

9.2.4 ResultMap 框架是如何工作的

在 ResultMap 实例化对象中，主要是实现把从数据库中获得的 `java.sql.ResultSet` 实例化对象转化成 ResultMap 对象中的 Bean 或者 Map 对象。

ResultMap 框架主要应用与查询相关的操作。

平台在 SqlMap 映射文件中的 resultMap 节点的时候，会实例化一个 ResultMap 对象，最终保存在 SqlMapExecutorDelegate 实例化对象中 HashMap 类型的属性 resultMaps 中，同时把 resultMap 节点下的多个属性节点实例化成为 ResultMapping 对象并统一赋值给 ResultMap 对象的 resultMappings 数组属性。

当用户调用外部查询操作的时候，如果查询操作在映射文件中要求有对应的 resultMap 节点，则直接可以从 SqlMapExecutorDelegate 的 resultMaps 哈希变量组中获得对应的 id。如果这些查询操作在映射文件中没有对应的 resultMap 名称，而只是 resultSet，那系统会根据 resultSet 来自动创建一个 AutoResultMap 对象。而 AutoResultMap 类是继承 ResultMap 类的，所以同样这也是一个 ResultMap 对象。

由于平台在处理过程中，会把 ResultMap 对象放到 StatementScope 对象中，而 StatementScope 对象是贯穿全部处理过程的一个状态类。所以，在 SqlExecutor 对象进行 executeQuery 方法的时候，对于获得的 ResultSet 要进行处理，要调用如下程序代码。

```
while ((maxResults == SqlExecutor.NO_MAXIMUM_RESULTS || resultsFetched < maxResults)
    && rs.next()) {
    Object[] columnValues = resultMap.resolveSubMap(statementScope, rs).
getResults(statementScope, rs);
    callback.handleResultObject(statementScope, columnValues, rs);
    resultsFetched++;
}
```

实际上，这个代码是调用了 ResultMap 对象的 getResults 方法，代码如下。

```
public Object[] getResults(StatementScope statementScope, ResultSet rs)
    throws SQLException {
    ErrorContext errorContext = statementScope.getErrorContext();
    errorContext.setActivity("applying a result map");
    errorContext.setObjectId(this.getId());
    errorContext.setResource(this.getResource());
    errorContext.setMoreInfo("Check the result map.");
}
```

```

boolean foundData = false;
Object[] columnValues = new Object[getResultMappings().length];
for (int i = 0; i < getResultMappings().length; i++) {
    ResultMapping mapping = (ResultMapping) getResultMappings()[i];
    errorContext.setMoreInfo(mapping.getErrorString());
    if (mapping.getStatementName() != null) {
        if (resultClass == null) {
            throw new SqlMapException("results for ResultMap named " + getId() + ".");
        } else if (Map.class.isAssignableFrom(resultClass)) {
            Class javaType = mapping.getJavaType();
            if (javaType == null) {
                javaType = Object.class;
            }
            columnValues[i] = getNestedSelectMappingValue(statementScope, rs, mapping,
javaType);
        } else if (DomTypeMarker.class.isAssignableFrom(resultClass)) {
            Class javaType = mapping.getJavaType();
            if (javaType == null) {
                javaType = DomTypeMarker.class;
            }
            columnValues[i] = getNestedSelectMappingValue(statementScope, rs, mapping,
javaType);
        } else {
            Probe p = ProbeFactory.getProbe(resultClass);
            Class type = p.getPropertyTypeForSetter(resultClass, mapping.getProperty
Name());
            columnValues[i] = getNestedSelectMappingValue(statementScope, rs, mapping,
type);
        }
        foundData = foundData || columnValues[i] != null;
    } else if (mapping.getNestedResultMapName() == null) {
        columnValues[i] = getPrimitiveResultMappingValue(rs, mapping);
        if (columnValues[i] == null) {
            columnValues[i] = doNullMapping(columnValues[i], mapping);
        } else {
            foundData = true;
        }
    }
}
statementScope.setRowDataFound(foundData);
return columnValues;
}

```

这样就可以实现实例化一个 `Javabean` 并把 `resultset` 的值传递过去。

9.2.5 如何实现子查询

对于一些 `resultMap` 节点，其 `result` 节点中还有 `select` 方法属性，如下所示。

```
<resultMap id="get-product-result" class="com.ibatis.example.Product">
```



```

<result property="id" column="PRD_ID"/>
<result property="description" column="PRD_DESCRIPTION"/>
<result property="category" column="PRD_CAT_ID" select="getCategory"/>
</resultMap>

<resultMap id="get-category-result" class="com.ibatis.example.Category">
  <result property="id" column="CAT_ID"/>
  <result property="description" column="CAT_DESCRIPTION"/>
</resultMap>

<statement id="getProduct" parameterClass="int" resultMap="get-product-result">
  select * from PRODUCT where PRD_ID = #value#
</statement>

<statement id="getCategory" parameterClass="int" resultMap="get-category-result">
  select * from CATEGORY where CAT_ID = #value#
</statement>

```

resultMap 的 id 是 “get-product-result”，在其子节点 result 的属性是 “category”，而该节点有一个 select 属性，内容是 “getCategory”，而 getCategory 又是一个 statement 节点的内码。根据 getCategory 的 statement，可以知道要产生的 resultMap 是 “get-category-result” 的 ResultMap 对象。

这种情况就是要实现子查询模式，即在一个 ResultMap 对象中还有一个 ResultMap 对象。

实现方式是这样的，首先在读取配置信息的时候，根据 result 节点是否有 select 属性，进行处理 String statementName = childAttributes.getProperty("select");

在 ResultMap 类的 getResult(StatementScope statementScope, ResultSet rs)方法中，判断 mapping.getStatementName() 是否为空，如果不为空，则执行处理下一级查询的方式，代码如下。

```

for (int i = 0; i < getResultMappings().length; i++) {
    ResultMapping mapping = (ResultMapping) getResultMappings()[i];
    errorContext.setMoreInfo(mapping.getErrorString());
    if (mapping.getStatementName() != null) {
        if (resultClass == null) { throw new SqlMapException(".....");
        } else if (Map.class.isAssignableFrom(resultClass)) {
            Class javaType = mapping.getJavaType();
            if (javaType == null) {
                javaType = Object.class;
            }
            columnValues[i] = getNestedSelectMappingValue(
                statementScope, rs, mapping, javaType);
        } else if (DomTypeMarker.class.isAssignableFrom(resultClass)) {
            Class javaType = mapping.getJavaType();
            if (javaType == null) {
                javaType = DomTypeMarker.class;
            }
            columnValues[i] = getNestedSelectMappingValue(
                statementScope, rs, mapping, javaType);
        }
    }
}

```



```

        } else {
            Probe p = ProbeFactory.getProbe(resultClass);
            Class type = p.getPropertyTypeForSetter(resultClass,
                mapping.getPropertyName());
            columnValues[i] = getNestedSelectMappingValue(
                statementScope, rs, mapping, type);
        }
        foundData = foundData || columnValues[i] != null;
    } else if (mapping.getNestedResultMapName() == null) {
        columnValues[i] = getPrimitiveResultMappingValue(rs, mapping);
        if (columnValues[i] == null) {
            columnValues[i] = doNullMapping(columnValues[i], mapping);
        } else {
            foundData = true;
        }
    }
}
}
}

```

调用了方法 `getNestedSelectMappingValue` 进行细化处理。

```

protected Object getNestedSelectMappingValue(StatementScope statementScope,
    ResultSet rs, ResultMapping mapping, Class targetType) throws
SQLException {
    try {
        TypeHandlerFactory typeHandlerFactory = getDelegate()
            .getTypeHandlerFactory();

        String statementName = mapping.getStatementName();
        SqlMapClientImpl client = (SqlMapClientImpl) statementScope
            .getSession().getSqlMapClient();

        MappedStatement mappedStatement = client
            .getMappedStatement(statementName);
        Class parameterType = mappedStatement.getParameterClass();
        Object parameterObject = null;

        if (parameterType == null) {
            parameterObject = prepareBeanParameterObject(statementScope,
                rs, mapping, parameterType);
        } else {
            if (typeHandlerFactory.hasTypeHandler(parameterType)) {
                parameterObject = preparePrimitiveParameterObject(rs,
                    mapping, parameterType);
            } else if (DomTypeMarker.class.isAssignableFrom(parameterType)) {
                parameterObject = prepareDomParameterObject(rs, mapping);
            } else {
                parameterObject = prepareBeanParameterObject(
                    statementScope, rs, mapping, parameterType);
            }
        }
    }
}

```

```

Object result = null;
if (parameterObject != null) {
    Sql sql = mappedStatement.getSql();
    ResultMap resultMap = sql.getResultMap(statementScope,
        parameterObject);
    Class resultClass = resultMap.getResultClass();

    if (resultClass != null
        && !DomTypeMarker.class.isAssignableFrom(targetType)) {
        if (DomCollectionTypeMarker.class
            .isAssignableFrom(resultClass)) {
            targetType = DomCollectionTypeMarker.class;
        } else if (DomTypeMarker.class.isAssignableFrom(resultClass)) {
            targetType = DomTypeMarker.class;
        }
    }

    result = ResultLoader.loadResult(client, statementName,
        parameterObject, targetType);

    String nullValue = mapping.getNullValue();
    if (result == null && nullValue != null) {
        TypeHandler typeHandler = typeHandlerFactory
            .getTypeHandler(targetType);
        if (typeHandler != null) {
            result = typeHandler.valueOf(nullValue);
        }
    }
}
return result;
} catch (InstantiationException e) {
    throw new NestedSQLException(
        "Error setting nested bean property. Cause: " + e, e);
} catch (IllegalAccessException e) {
    throw new NestedSQLException(
        "Error setting nested bean property. Cause: " + e, e);
}
}

```

这样就可以实现子查询了。

9.2.6 延迟加载的实现

我们知道，Java 类加载的一个重要特点就是懒惰加载 (lazy load)，即只有当要用到这个类时，系统才会加载这个类。Java 类是由类加载器负责加载的，ClassLoader 类就是一个基本的类加载器，它是用 Java 语言写成的，所以可以通过继承它并重载类加载的方法来实现一种自定义的类加载方式，大大提高了程序的灵活性。

在 iBATIS 中也有这种懒惰加载 (lazy load)，叫做延迟加载。所谓延迟加载，即一般情

况下系统已经获得了检索对象，但是为了不影响系统性能，并不把当前没有使用到的对象加载到内存中。只有当调用到相关对象的时候，系统才会加载这个对象，实际上也就是懒汉式加载。

iBATIS SQL Maps 中通过配置文件在全局配置中加入这个参数来配置延迟加载模式。

```
<settings enhancementEnabled="true" lazyLoadingEnabled="true" />
```

iBATIS 实现延迟加载的技术有两种模式，一种是正常的延迟加载，另一种是调用外部的 CGLib 库的延迟加载，这主要是为了提高性能。

iBATIS 的延迟加载主要涉及 `com.ibatis.sqlmap.engine.mapping.result.loader` 包下的 `ResultLoader`、`EnhancedLazyResultLoader`、`LazyResultLoader` 三个类。其中 `ResultLoader` 类是调用 `ResultMap` 对象的加载方法，`LazyResultLoader` 是正常的延迟加载实现，而 `EnhancedLazyResultLoader` 是加强的延迟加载。延迟加载的核心思想还是动态代理模式，其对象主要是 `Collection` 类型数据，如果只是一个对象，基本上也不必采用延迟加载的方式。

当 iBATIS 进行加载时，要调用 `ResultLoader` 类的 `loadResult` 方法，代码如下。

```
public static Object loadResult(SqlMapClientImpl client, String statementName,
    Object parameterObject, Class targetType)
    throws SQLException {
    Object value = null;
    if (client.isLazyLoadingEnabled()) {
        if (client.isEnhancementEnabled()) {
            //采用加强的延迟加载
            EnhancedLazyResultLoader lazy = new EnhancedLazyResultLoader(client,
                statementName, parameterObject, targetType);
            value = lazy.loadResult();
        } else {
            // 采用正常的延迟加载
            LazyResultLoader lazy = new LazyResultLoader(client, statementName,
                parameterObject, targetType);
            value = lazy.loadResult();
        }
    } else {
        //立即加载
        value = getResult(client, statementName, parameterObject, targetType);
    }
    return value;
}
```

根据配置信息中延迟加载的类型不同，来调用不同的延迟加载对象。对于 `LazyResultLoader` 延迟加载模式来说，首先是创建一个 `LazyResultLoader` 对象，然后再调用 `LazyResultLoader` 对象的 `loadResult` 方法。

那么 `LazyResultLoader` 类是如何实现动态代理的呢？`LazyResultLoader` 类是一个动态代理类，要实现 `InvocationHandler` 接口，其必须实现的方法是 `invoke` 方法。

当调用 `LazyResultLoader` 对象的 `loadResult` 方法时，代码如下。

```

public Object loadResult() throws SQLException {
    if (Collection.class.isAssignableFrom(targetType)) {
        InvocationHandler handler = new LazyResultLoader(client, statementName,
parameterObject, targetType);
        ClassLoader cl = targetType.getClassLoader();
        if (Set.class.isAssignableFrom(targetType)) {
            // 基于 Set.class 的动态代理类
            return Proxy.newProxyInstance(cl, SET_INTERFACES, handler);
        } else {
            // 基于 List.class 的动态代理类
            return Proxy.newProxyInstance(cl, LIST_INTERFACES, handler);
        }
    } else {
        return ResultLoader.getResult(client, statementName, parameterObject, target
Type);
    }
}

```

代码含义比较简单，如果当前对象的返回值是 `Collection.class`，那么创建一个 `InvocationHandler` 实现类，而该实现类实际上是 `LazyResultLoader` 类。再调用 `Proxy.newProxyInstance(cl, SET_INTERFACES, handler)` 转化为一个动态代理类，这个动态类代理的是 `List` 或者 `Map`。当调用这个动态代理对象的 `loadObject` 方法时，即调用 `invoke` 方法，代码如下。

```

public Object invoke(Object o, Method method, Object[] objects) throws Throwable {
    if ("finalize".hashCode() == method.getName().hashCode()
        && "finalize".equals(method.getName())) {
        return null;
    } else {
        loadObject();
        if (resultObject != null) {
            try {
                return method.invoke(resultObject, objects);
            } catch (Throwable t) {
                throw ClassInfo.unwrapThrowable(t);
            }
        } else {
            return null;
        }
    }
}

```

方法是在执行过程中先去调用一个同步线程方法 `loadObject`，然后才去获取数据，代码如下。

```

private synchronized void loadObject() {
    if (!loaded) {
        try {
            loaded = true;
            // 真正开始加载数据
            resultObject = ResultLoader.getResult(client, statementName, parameter

```

```
Object, targetType);
    } catch (SQLException e) {
        throw new RuntimeException("Error lazy loading result. Cause: " + e, e);
    }
}
}
```

获得对象后，也许是 List、Map 或是其他对象，再通过反射机制调用该对象的方法。

EnhancedLazyResultLoader 类的实现方式与 LazyResultLoader 类相同，只是在 loadResult 方法中调用 Enhancer.create 方法，而不是 LazyResultLoader 类的 Proxy.newProxyInstance 方法。EnhancedLazyResultLoader 类的 loadResult 方法代码如下。

```
public Object loadResult() throws SQLException {
    if (DomTypeMarker.class.isAssignableFrom(targetType)) {
        return ResultLoader.getResult(client, statementName, parameterObject,
            targetType);
    } else if (Collection.class.isAssignableFrom(targetType)) {
        if (Set.class.isAssignableFrom(targetType)) {
            return Enhancer.create(Object.class, SET_INTERFACES, this);
        } else {
            return Enhancer.create(Object.class, LIST_INTERFACES, this);
        }
    } else if (targetType.isArray() || ClassInfo.isKnownType(targetType)) {
        return ResultLoader.getResult(client, statementName, parameterObject,
            targetType);
    } else {
        return Enhancer.create(targetType, this);
    }
}
```

9.3 Statement 框架及其说明

9.3.1 Statement 介绍

SQL Map 的核心概念是 MappedStatement。MappedStatement 可以使用任意的 SQL 语句，并拥有 Parameter map（输入）和 Result map（输出）。如果是简单情况，MappedStatement 可以使用 Java 类来作为 Parameter 和 Result。MappedStatement 也可以使用缓存模型，即在内存中缓存常用的数据。MappedStatement 的结构如下所示：

```
<statement id="updateProfile" cacheModel="name Of Cache"
    parameterMap="name Of ParameterMap" parameterClass="some.class.Name"
    resultMap="name Of ResultMap" resultClass="some.class.Name">
    UPDATE PROFILE SET LANGPREF = #languagePreference#, FAVCATEGORY
    = #favouriteCategoryId#, MYLISTOPT = #listOption#, BANNEROPT =
    #bannerOption# WHERE USERID = #username#
</statement>
```

其中<statement>元素是一个通用声明，可以用于任何类型的 SQL 语句。通常，使用具体的 statement 类型是个好主意。具体的 statement 类型能提供更直观的 XML DTD，并拥有某些<statement>元素没有的特性。

9.3.2 Statement 框架总体结构

在 Statement 组件中主要内容在 com.ibatis.sqlmap.engine.mapping.statement 包内。由 Statement、ExecuteListener 等类组成。其中 MappedStatement 最重要，处于一个核心地位，其他实现 MappedStatement 的实现类都是继承该 MappedStatement 类。Statement 组件类的结构如图 9-3 所示。

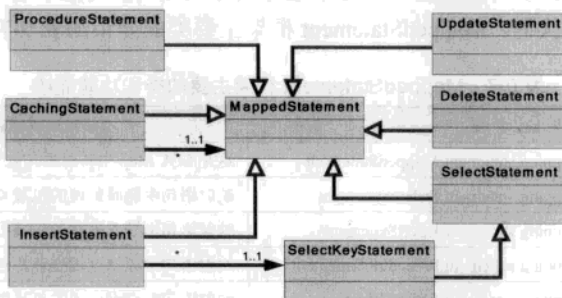


图 9-3 Statement 框架类结构图

MappedStatement 类处于父类地位，InsertStatement 类、SelectStatement 类、Delete Statement 类、UpdateStatement 类、ProcedureStatement 类、CachingStatement 类等都继承 MappedStatement 类。CachingStatement 类还关联 MappedStatement 类，这形成了一种装饰模式。SelectKeyStatement 类继承 SelectStatement 类，并被 InsertStatement 类关联。

MappedStatement 类在 Statement 组件中处于领导地位，在与其他 mapping 组件的协作上也是处于中心地位。在合作模式上，MappedStatement 是几个实现结合点，其类结构如图 9-4 所示。

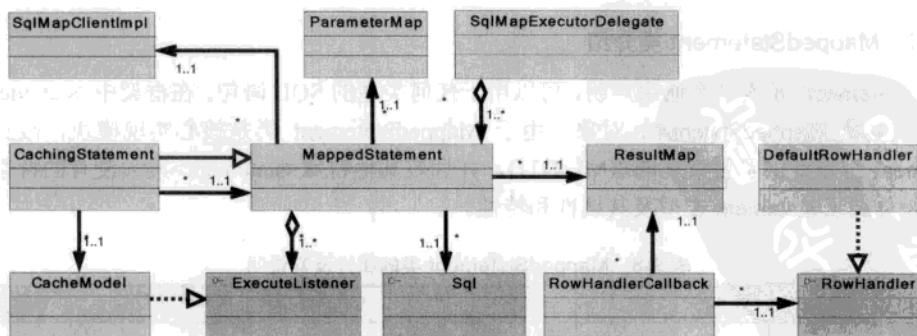


图 9-4 Statement 框架在 Mapping 组件的类结构图

图 9-4 说明 MappedStatement 类被 SqlMapExecutorDelegate 类所聚合，同时关联 SqlMapClientImpl 对象。这样实际上与 SqlMapExecutorDelegate 对象形成双向关联，以实现通过 MappedStatement 对象可以获得 SqlMapExecutorDelegate 对象，而对于 SqlMapExecutorDelegate 对象而言，MappedStatement 对象只是 SqlMapExecutorDelegate 对象中的一个实例化对象。MappedStatement 类关联 ParameterMap 类，这样，通过 MappedStatement 对象就获得该对象输入参数的 ParameterMap 对象。MappedStatement 类关联 ResultMap 类，这样通过 MappedStatement 对象就获得该对象输出结果的 ResultMap 对象。MappedStatement 类关联 Sql 接口，这样通过 MappedStatement 对象就获得该对象执行 SQL 语句的 Sql 实例化对象。CachingStatement 类继承并关联 MappedStatement 类，这是 MappedStatement 处理缓存的装饰设计模式，关于缓存处理在 Statement 缓存实现会进行详细的说明。MappedStatement 类聚合 ExecuteListener 接口，这是为了将来进行缓存处理（如清空该 MappedStatement 对象的缓存对象）留下的伏笔。MappedStatement 框架主要类的介绍如表 9-7 所示。

表 9-7 MappedStatement 框架主要的类和功能描述

接 口 或 类	功能或用途描述	备注
com.ibatis.sqlmap.engine.mapping.statement.MappedStatement	这是 SQL Map 中映射语句处理的核心包装类	
com.ibatis.sqlmap.engine.mapping.statement.DeleteStatement	映射语句中删除处理的包装类	
com.ibatis.sqlmap.engine.mapping.statement.InsertStatement	映射语句中插入处理的包装类	
com.ibatis.sqlmap.engine.mapping.statement.ProcedureStatement	映射语句中存储过程处理的包装类	
com.ibatis.sqlmap.engine.mapping.statement.SelectStatement	映射语句中查询处理的包装类	
com.ibatis.sqlmap.engine.mapping.statement.UpdateStatement	映射语句中修改处理的包装类	
com.ibatis.sqlmap.engine.mapping.statement.SelectKeyStatement	映射语句中主键生成处理的包装类	
com.ibatis.sqlmap.engine.mapping.statement.CachingStatement	映射语句中缓存处理的包装类	
com.ibatis.sqlmap.engine.mapping.statement.DefaultRowHandler	映射语句返回行结果	
com.ibatis.sqlmap.engine.mapping.statement.ExecuteListener	缓存接口	
com.ibatis.sqlmap.engine.mapping.statement.RowHandlerCallback	映射语句处理返回行的处理类	
com.ibatis.sqlmap.engine.mapping.statement.StatementType	映射语句类型的定义类	

9.3.3 Statement 组件中的类介绍

1. MappedStatement 类介绍

<statement>元素是个通用声明，可以用于任何类型的 SQL 语句。在框架中<statement>元素会生成 MappedStatement 对象。由于 MappedStatement 类是核心实现模块，故具体 statement 对象提供了更直观的 XML DTD，并拥有其他特殊<statement>元素没有的特性。表 9-8 总结了 statement 类型及其属性和特性。

表 9-8 MappedStatement 类的属性及其说明

属 性 名 称	类 型	功能和用途	备注
id	String	MappedStatement 对象的全局唯一内码	
resultSetType	Integer	MappedStatement 处理返回的 resultSet 的类型	

续表

属性名称	类型	功能和用途	备注
fetchSize	Integer	MappedStatement 为 JDBC 驱动程序提供关于需要更多行时应该从数据库获取的行数	
resultMap	ResultMap	MappedStatement 对象拥有的输出结果对象	
parameterMap	ParameterMap	MappedStatement 对象拥有的输入参数对象	
parameterClass	Class	MappedStatement 对象拥有的输入参数的 Class 类型	
SQL	SQL	MappedStatement 对象拥有的 SQL 语句对象	
baseCacheKey	int	MappedStatement 对象形成唯一缓存 Hash 码的最基本的内码	
SQLMapClient	SQLMapClientImpl	MappedStatement 对象对应的外部 SQLMapClientImpl 对象	
timeout	Integer	耗时标志	
additionalResultMaps	ResultMap[]	附加的输出结果对象集合	
executeListeners	ArrayList	MappedStatement 对象包含的缓存对象列表	
resource	String		

MappedStatement 类的方法如表 9-9 所示。

表 9-9 MappedStatement 类的方法及其说明

方法名称	参数	返回值	功能和用途	备注
getStatementType	无	public StatementType	获得当前 MappedStatement 对象的类型	
executeUpdate	StatementScope, Transaction, Object	public int	执行修改方法	
executeQueryForObject	StatementScope statementScope, Transaction trans, Object parameterObject, Object resultObject)	public Object	执行查询方法, 获得一个对象	
executeQueryForList	StatementScope statementScope Transaction trans, Object parameterObject, int skipResults, int maxResults	public List	执行查询方法, 获得一组对象	
executeQueryWithRow Handler	StatementScope statementScope, Transaction trans, Object parameterObject, RowHandler rowHandler)	无	执行查询方法, 形成 RowHandler 列表	

关于上述方法, 在 MappedStatement 是如何工作的会有详细说明。

2. StatementType 类介绍说明

Statement 组件中有一个映射类型变量类 StatementType, 其功能主要就是对 Statement 组件的类别进行分类。StatementType 类结构如图 9-5 所示。

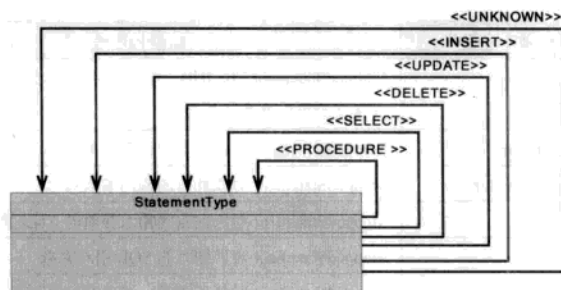


图 9-5 StatementType 类结构图

这是一种自身关联（反身关联）：自己引用自己，带着一个自己的引用，也就是说在自己的内部有着一个自身的引用。StatementType 的程序代码如下。

```
public final class StatementType {
    public static final StatementType UNKNOWN = new StatementType();
    public static final StatementType INSERT = new StatementType();
    public static final StatementType UPDATE = new StatementType();
    public static final StatementType DELETE = new StatementType();
    public static final StatementType SELECT = new StatementType();
    public static final StatementType PROCEDURE = new StatementType();

    private StatementType() {
    }
}
```

该代码定义了六种 Statement 类型：UNKNOWN 表示未知 Statement 类型，INSERT 表示插入数据的 Statement 类型，UPDATE 表示修改数据的 Statement 类型，DELETE 表示删除数据的 Statement 类型，SELECT 表示选择数据的 Statement 类型，PROCEDURE 表示存储过程的 Statement 类型。

3. DeleteStatement、InsertStatement、ProcedureStatement、SelectStatement、UpdateStatement 类介绍

MappedStatement Class 与 MappedStatement Class、InsertStatement Class、Update Statement Class、DeleteStatement Class、SelectStatement Class、ProcedureStatement Class 的类关系如图 9-3 所示。可以采用 GoF 的设计模式中的状态设计模式去理解这个结构。

所谓状态（State）模式标准，即允许一个对象在其内部状态改变时改变它的行为，从对象看起来似乎修改了它的类。状态（State）模式属于对象行为型模式，它允许一个对象在其内部状态改变的时候改变行为。这个对象看上去象是改变了它的类一样，可理解为在不同的上下文中，相同的动作导致的结果不同。状态模式把所研究的对象的行为包装在不同的状态对象里，每一个状态对象都属于一个抽象状态类的一个子类。状态模式的意图是让一个对象在其内部状态改变的时候，其行为也随之改变。状态模式需要对每一个系统可

能取得的状态创建一个状态类的子类。当系统的状态变化时，系统便改变所选的子类。

使用 State 模式可以将不同状态下的行为分隔开来，这样做的好处是很容易增加新的状态并实现状态转换而不影响已存在的状态和上下文环境，同时还避免了操作中庞大的条件分支语句，使代码更容易维护。状态 (State)

模式结构如图 9-6 所示，其角色包括抽象状态 (State) 角色、具体状态 (Concrete State) 角色和环境 (Context) 角色。其角色介绍如下。

① 抽象状态 (State) 角色：定义一个接口，用以封装环境 (Context) 对象的一个特定的状态所对应的行为。

② 具体状态 (Concrete State) 角色：每一个具体状态类都实现了环境 (Context) 的一个状态所对应的行为。

③ 环境 (Context) 角色：定义客户端所感兴趣的接口，并且保留一个具体状态类的实例。这个具体状态类的实例给出此环境对象的现有状态。

在本例中，可以把 MappedStatement 类理解为抽象状态 (State) 角色，把 MappedStatement 类、InsertStatement 类、UpdateStatement 类、DeleteStatement 类、SelectStatement 类、ProcedureStatement 类理解为具体状态 (Concrete State) 角色。而 StatementType 类就是不同的状态类型，当处于不同的状态类型时，具体的实现类有不同的实现方法，也就是实例化对象的多态性。而环境 (Context) 角色可以理解为调用方式。

表 9-10 总结了 Statement 组件各个类的 StatementType 属性和方法：

表 9-10 Statement 组件各个类的 StatementType 属性和方法

类和对象	StatementType 属性	操作方法	正确或异常
MappedStatement	UNKNOWN	insert	正确
		update	正确
		delete	正确
		所有的查询方法	正确
InsertStatement Class	INSERT	insert	正确
		update	正确
		delete	正确
		所有的查询方法	异常
UpdateStatement Class	UPDATE	insert	正确
		update	正确
		delete	正确
		所有的查询方法	异常
DeleteStatement Class	DELETE	insert	正确
		update	正确
		delete	正确
		所有的查询方法	异常

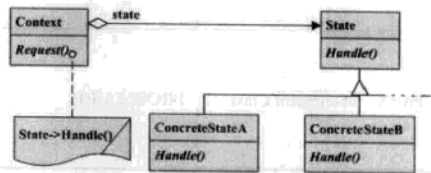


图 9-6 状态模式结构

续表

类和对象	StatementType 属性	操作方法	正确或异常
SelectStatement Class	SELECT	Insert	异常
		update	异常
		delete	异常
		所有的查询方法	正确
ProcedureStatement Class	PROCEDURE	insert	正确
		update	正确
		delete	正确
		所有的查询方法	正确

不失一般性，我们可用 DeleteStatement 类来看看，代码如下。

```

public class DeleteStatement extends MappedStatement {

    public StatementType getStatementType() {
        return StatementType.DELETE;
    }

    public Object executeQueryForObject(StatementScope statementScope, Transaction
trans, Object parameterObject, Object resultObject) throws SQLException {
        throw new SQLException("Delete statements cannot be executed as a query.");
    }

    public List executeQueryForList(StatementScope statementScope, Transaction trans,
Object parameterObject, int skipResults, int maxResults) throws SQLException {
        throw new SQLException("Delete statements cannot be executed as a query.");
    }

    public void executeQueryWithRowHandler(StatementScope statementScope, Transactiontrans,
Object parameterObject, RowHandler rowHandler) throws SQLException {
        throw new SQLException("Delete statements cannot be executed as a query.");
    }
}

```

4. RowHandlerCallback

用于管理 Row 记录的一个类。通过调用 DefaultRowHandler 对象来处理实现的 com.ibatis.sqlmap.client.event.RowHandler 接口。

9.3.4 MappedStatement 是如何工作的

MappedStatement 对象在实例化过程中，一般会有自己的状态对象。当外部 SqlMapExecutorDelegate 对象调用 MappedStatement 对象的方法时，转化为 MappedStatement 对象的内部处理，主要是参数和 SQL 的处理，然后再途经 SqlMapExecutorDelegate 对象去调用 SqlExecutor 对象的方法。

在 `MappedStatement` 类中，以下几个方法是最重要的，如 `executeUpdate`、`executeQueryForObject`、`executeQueryForList`、`executeQueryWithRowHandler`、`executeQueryWithCallback` 等。

在实际应用中，如果外部要做新增、删除和修改操作，都通过 `SqlMapExecutorDelegate` 对象调用 `MappedStatement` 对象的 `executeUpdate` 方法。

如果是做查询操作，都通过 `SqlMapExecutorDelegate` 对象调用 `MappedStatement` 对象的 `executeQueryForObject`、`executeQueryForList` 和 `executeQueryWithRowHandler` 方法。然后由 `MappedStatement` 对象再调用自身的 `executeQueryWithCallback` 方法。

1. `MappedStatement` 类的 `executeUpdate` 方法实现

`MappedStatement` 对象的 `executeUpdate` 方法主要是进行新增、删除和修改操作，其代码实现的框架如下，实现过程如图 9-7 所示。

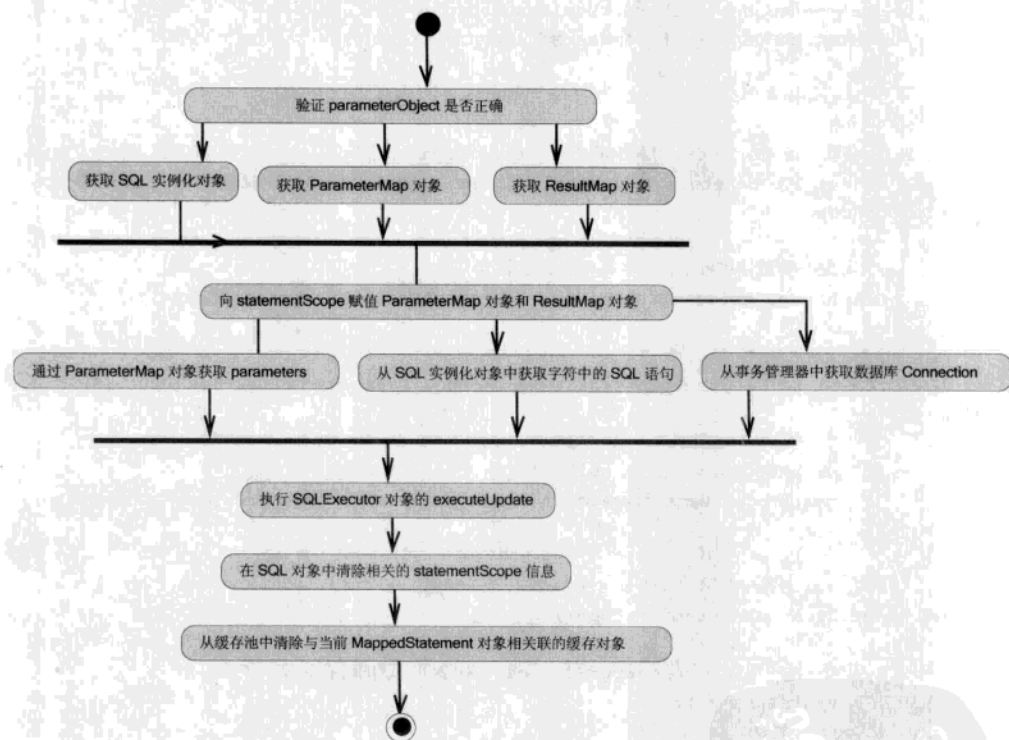


图 9-7 `MappedStatement` 类的 `executeUpdate` 方法实现图

实现代码如下（只保留了核心代码）。

```

public int executeUpdate(StatementScope statementScope, Transaction trans, Object
parameterObject)
    throws SQLException {

    statementScope.getSession().setCommitRequired(true);
  
```

```
try {
    //验证 parameterObject 是否正确
    parameterObject = validateParameter(parameterObject);

    //获取当前 MappedStatement 对象配置形成的 Sql 实例化对象
    Sql sql = getSql();

    //获取当前 MappedStatement 对象配置形成的 ParameterMap 实例化对象
    ParameterMap parameterMap = sql.getParameterMap(statementScope, parameterObject);

    //获取当前 MappedStatement 对象配置形成的 ResultMap 实例化对象
    ResultMap resultMap = sql.getResultMap(statementScope, parameterObject);

    //把 ParameterMap 对象和 ResultMap 对象赋值给状态变量 statementScope
    statementScope.setResultMap(resultMap);
    statementScope.setParameterMap(parameterMap);

    int rows = 0;

    //获取 ParameterMap 对象中的 ParameterMapping 对象组
    Object[] parameters = parameterMap.getParameterObjectValues(statementScope, parameterObject);

    //从 Sql 实例化对象中获取字符串的 Sql 语句
    String sqlString = sql.getSql(statementScope, parameterObject);

    //提交执行语句
    rows = sqlExecuteUpdate(statementScope, trans.getConnection(), sqlString, parameters);

    if (parameterObject != null) {
        postProcessParameterObject(statementScope, parameterObject, parameters);
    }

    //在 sql 对象中清除相关的 statementScope 信息
    sql.cleanup(statementScope);

    //从缓存池中清除与当前 MappedStatement 对象相关联的缓存对象
    notifyListeners();
    return rows;
} catch (SQLException e) {
    errorContext.setCause(e);
    throw new NestedSQLException(errorContext.toString(), e.getSQLState(), e.getErrorCode(), e);
} catch (Exception e) {
    errorContext.setCause(e);
    throw new NestedSQLException(errorContext.toString(), e);
}
```

调用了自身的 `sqlExecuteUpdate` 方法，代码如下。

```
protected int sqlExecuteUpdate(StatementScope statementScope, Connection conn,
String sqlString, Object[] parameters) throws SQLException {
    //判断是否采用了批处理操作
    if (statementScope.getSession().isInBatch()) {
        //调用 SqlExecutor 对象的 addBatch 方法
        getSqlExecutor().addBatch(statementScope, conn, sqlString, parameters);
        return 0;
    } else {
        //调用 SqlExecutor 对象的 executeUpdate 方法
        return getSqlExecutor().executeUpdate(statementScope, conn, sqlString,
parameters);
    }
}
```

完成后调用 `Sql` 对象的 `cleanup(statementScope)` 方法，该方法主要是针对 `Void cleanup`。代码如下。

```
public void cleanup(StatementScope statementScope) {
    statementScope.setDynamicSql(null);
    statementScope.setDynamicParameterMap(null);
}
```

完成后调用 `notifyListeners` 方法，代码如下：

```
public void notifyListeners() {
    for (int i = 0, n = executeListeners.size(); i < n; i++) {
        ((ExecuteListener) executeListeners.get(i)).onExecuteStatement(this);
    }
}
```

而实现 `ExecuteListener` 接口的实例化对象是 `CacheModel`。所以该方法是执行 `CacheModel` 对象的 `onExecuteStatement(this)` 方法，代码如下。

```
public void onExecuteStatement(MappedStatement statement) {
    flush();
}
```

实际上是清除了当前的存储对象。

```
public void flush() {
    synchronized (this) {
        controller.flush(this);
        lastFlush = System.currentTimeMillis();
        ...
    }
}
```

2. MappedStatement 类的 executeQueryForObject 方法实现

`MappedStatement` 对象的 `executeQueryForObject` 方法主要是进行查询并返回一个对象

的操作。其实现过程如图 9-8 所示。

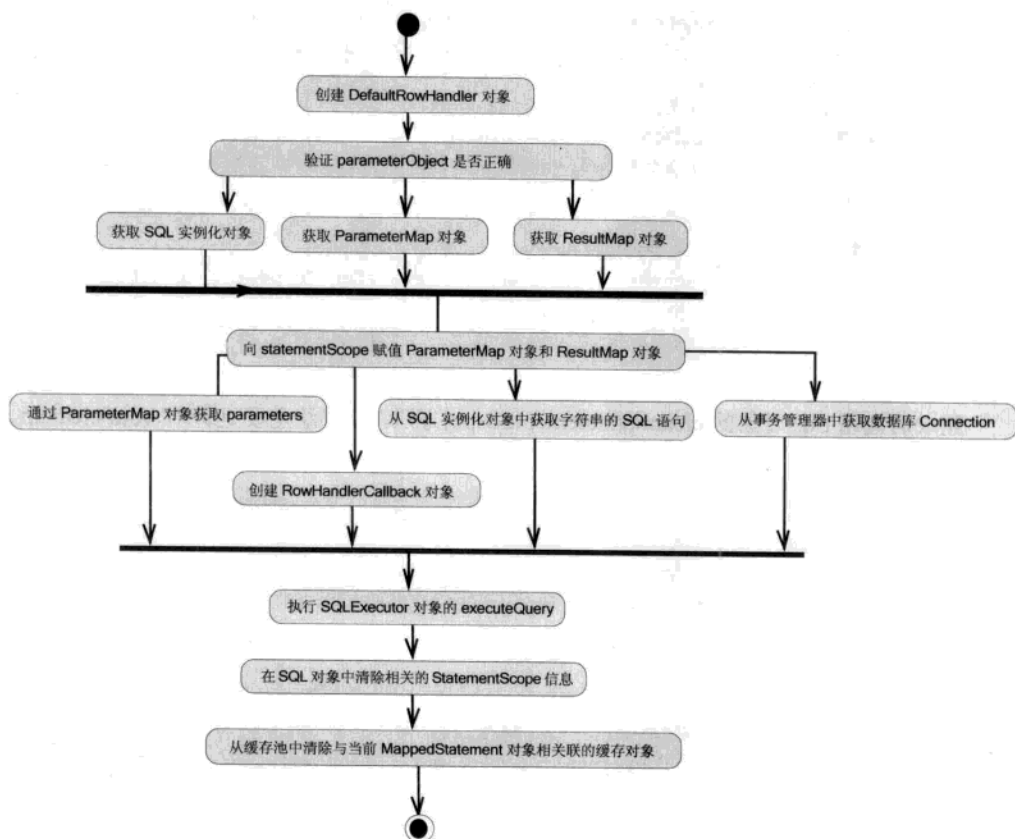


图 9-8 MappedStatement 类的 executeQueryForObject 方法实现图

实现代码如下（只保留了核心代码）。

```

public Object executeQueryForObject(StatementScope statementScope, Transaction
trans, Object parameterObject, Object resultObject) throws SQLException {
    try {
        Object object = null;

        DefaultRowHandler rowHandler = new DefaultRowHandler();
        executeQueryWithCallback(statementScope, trans.getConnection(), parameter
Object, resultObject, rowHandler, SqlExecutor.NO_SKIPPED_RESULTS, SqlExecutor.NO_
MAXIMUM_RESULTS);
        List list = rowHandler.getList();

        if (list.size() > 1) {
            throw new SQLException("Error: executeQueryForObject returned too many
results.");
        } else if (list.size() > 0) {

```

```

        object = list.get(0);
    }

    return object;
} catch (TransactionException e) {
    throw new NestedSQLException("Error getting Connection from Transaction.
Cause: " + e, e);
}
}

```

主要是调用 `executeQueryWithCallback` 方法，我们来看看代码。

```

protected void executeQueryWithCallback(StatementScope statementScope, Connection
conn, Object parameterObject, Object resultObject, RowHandler rowHandler, int skip
Results, int maxResults)
    throws SQLException {

    try {
        //验证 parameterObject 是否正确
        parameterObject = validateParameter(parameterObject);

        //获取当前 MappedStatement 对象配置形成的 Sql 实例化对象
        Sql sql = getSql();

        //获取当前 MappedStatement 对象配置形成的 ParameterMap 实例化对象
        ParameterMap parameterMap = sql.getParameterMap(statementScope, parameter
Object);

        //获取当前 MappedStatement 对象配置形成的 ResultMap 实例化对象
        ResultMap resultMap = sql.getResultMap(statementScope, parameterObject);

        //把 ParameterMap 对象和 ResultMap 对象赋值给状态变量 statementScope
        statementScope.setResultMap(resultMap);
        statementScope.setParameterMap(parameterMap);

        //获取 ParameterMap 对象中的 ParameterMapping 对象组
        Object[] parameters = parameterMap.getParameterObjectValues(statementScope,
parameterObject);

        //从 Sql 实例化对象中获取字符串的 Sql 语句
        String sqlString = sql.getSql(statementScope, parameterObject);

        //提交执行语句
        RowHandlerCallback callback = new RowHandlerCallback(resultMap, result
Object, rowHandler);
        sqlExecuteQuery(statementScope, conn, sqlString, parameters, skipResults,
maxResults, callback);

        if (parameterObject != null) {
            postProcessParameterObject(statementScope, parameterObject, parameters);
        }
    }
}

```

```
//在 sql 对象中清除相关的 statementScope 信息
sql.cleanup(statementScope);

//从缓存池中清除与当前 MappedStatement 对象相关联的缓存对象
notifyListeners();
} catch (SQLException e) {
    errorContext.setCause(e);
    throw new NestedSQLException(errorContext.toString(), e.getSQLState(), e.get
        ErrorCode(), e);
} catch (Exception e) {
    errorContext.setCause(e);
    throw new NestedSQLException(errorContext.toString(), e);
}
}
```

调用了自身的 `sqlExecuteQuery` 方法，代码如下。

```
protected void sqlExecuteQuery(StatementScope statementScope, Connection conn,
    String sqlString, Object[] parameters, int skipResults, int maxResults, RowHandler
    Callback callback) throws SQLException {
    getSqlExecutor().executeQuery(statementScope, conn, sqlString, parameters,
        skipResults, maxResults, callback);
}
```

3. MappedStatement 类的 executeQueryForList 方法实现

`MappedStatement` 类的 `executeQueryForList` 方法实现只是 `executeQueryForObject` 方法的一部分，其实现代码如下（只保留了核心代码）。

```
public List executeQueryForList(StatementScope statementScope, Transaction trans,
    Object parameterObject, int skipResults, int maxResults)
    throws SQLException {
    try {
        DefaultRowHandler rowHandler = new DefaultRowHandler();
        executeQueryWithCallback(statementScope, trans.getConnection(), parameter
            Object, null, rowHandler, skipResults, maxResults);
        return rowHandler.getList();
    } catch (TransactionException e) {
        throw new NestedSQLException("Error getting Connection from Transaction.
            Cause: " + e, e);
    }
}
```

后面的代码实现与 `executeQueryForObject` 方法类似。

4. MappedStatement 类的 executeQueryWithRowHandler 方法实现

`MappedStatement` 类的 `executeQueryWithRowHandler` 方法实现只是 `executeQueryForObject` 方法的一部分，实现代码如下（只保留了核心代码）。

```

public void executeQueryWithRowHandler(StatementScope statementScope, Transaction
trans, Object parameterObject, RowHandler rowHandler) throws SQLException {
    try {
        executeQueryWithCallback(statementScope, trans.getConnection(), parameter
Object, null, rowHandler, SqlExecutor.NO_SKIPPED_RESULTS, SqlExecutor.NO_MAXIMUM_
RESULTS);
    } catch (TransactionException e) {
        throw new NestedSQLException("Error getting Connection from Transaction.
Cause: " + e, e);
    }
}

```

后面的代码实现与 `executeQueryForObject` 方法类似。

9.3.5 Statement 缓存的实现

在 iBATIS 中, 缓存主要是针对查询。当对一个对象查询完成后, 把查询结果保存起来。当下一次再进行查询的时候, 如果是同样的对象, 而且是同样的参数, 就到缓存中去寻找相同的记录, 如果有这个记录就返回给记录, 如果没有这个记录就到数据库中去查询。

SQL Map 中 Statement 的缓存的类结构如图 9-9 所示。

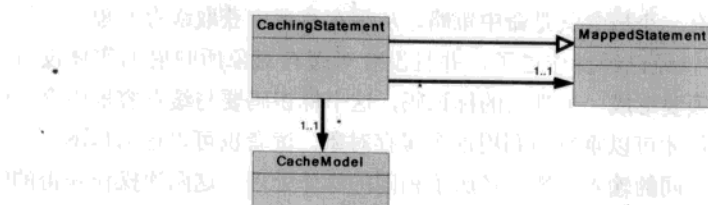


图 9-9 SQL Map 中 Statement 的缓存的类结构

这种设计模式采用的是装饰设计模式。即 `CachingStatement` 类即继承 `MappedStatement` 类, 同时又关联 `MappedStatement` 类。在这里做一个装饰设计模式的结构说明, 装饰 (Decorator) 模式属于结构型设计模式, 其标准定义是动态地给一个对象添加一些额外的职责。就增加功能来说, Decorator 模式相比生成子类更为灵活。在实际应用中, 装饰模式用于动态地为对象附加额外的职责, 以达到扩展其功能的效果; 同时, 它也是继承关系的一个替代方案, 提供比继承更多的灵活性。动态给一个对象增加功能, 这些功能可以再动态地撤消。增加的是由一些基本功能的排列组合而产生的非常大量的功能。装饰 (Decorator) 模式属于结构型设计模式。

装饰 (Decorator) 模式结构如图 9-10 所示, 角色包括抽象构件 (Component) 角色、具体构件 (Concrete Component) 角色、装饰 (Decorator) 角色和具体装饰 (Concrete Decorator) 角色。其角色介绍如下:

- ① 抽象构件 (Component) 角色: 给出一个抽象接口, 以规范准备接收附加责任的对

象。抽象构件角色作为整个装饰模式类图体系的超类，其子类均可以被增添附加责任。

② 具体构件（Concrete Component）角色：定义一个将要接收附加责任的类。

③ 装饰（Decorator）角色：持有一个构件（Component）对象的实例，并定义一个与抽象构件接口一致的接口。

④ 具体装饰（Concrete Decorator）角色：负责给构件对象增添附加的责任。

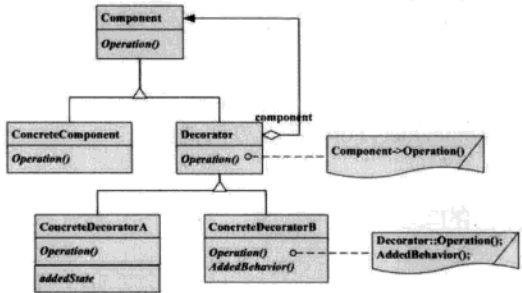


图 9-10 装饰模式结构

在本例中省去了抽象构件角色和装饰角色，而 MappedStatement 充当具体构件（Concrete Component）角色，CachingStatement 充当具体装饰（Concrete Decorator）角色。

但是要实现缓存，还有一个概念就是命中策略。从缓存容器中获取缓存对象，必须的一个条件就是当前的操作曾经已经操作过了，并且没有对缓存对象所映射的实体没有变化。所以，当前的操作必须要形成一个唯一的标识码，这个标识码要与缓存容器中存在的缓存对象标识码相同，我们才可以重复地利用这个缓存对象。读者也可以这么理解，两个操作，必须是相同的语句，同的输入参数，经历了相同的操作处理，这两次操作获得的输出对象才应该是相同的。这个条件有些苛刻，但这是必须的。

iBATIS SQL Map 中的实现缓存主要就是两个对象，一个是 Statement 对象，另一个是 ParameterMap。这两个对象能形成唯一性，可以通过 CacheKey 来转化为唯一的数字，并保存到缓存容器中。这就是缓存的基础，这些实现都是 CachingStatement 类中的查询方法。下面来介绍这个 CacheKey 形成的过程。

也就是要形成唯一标识在缓存容器中的缓存对象唯一码。

CachingStatement 的 CacheKey 的形成过程，要经历 CachingStatement 对象、ParameterMap 对象、Sql 对象处理和 BaseDataExchange 处理。在这么多对象的过程中进行哈希算法，这些过程要求 CachingStatement 对象 id 和 baseCacheKey 要一样，SQL 语句要一样，传入的参数要一样等。而且实现的过程步骤和顺序都要一样，最后得出并形成一个唯一的 CacheKey 对象。

在 BaseDataExchange 对象中创建 CacheKey 对象。

由于缓存主要是针对查询处理，所以我们还要了解基于查询的方法，如 executeQueryForObject 方法等。下面来看看其实现代码，当要进行缓存处理的时候，调用

CachingStatement 的 executeQueryForObject 方法, 代码如下。

```
public Object executeQueryForObject(StatementScope statementScope, Transaction
trans, Object parameterObject, Object resultObject) throws SQLException {
    //获取当前操作的唯一码
    CacheKey cacheKey = getCacheKey(statementScope, parameterObject);
    //唯一码的再次处理
    cacheKey.update("executeQueryForObject");
    //从缓存容器中获取对应唯一码的缓存对象
    Object object = cacheModel.getObject(cacheKey);
    if (object == CacheModel.NULL_OBJECT){
        // This was cached, but null
        object = null;
    }else if (object == null) {
        //缓存容器中没有唯一码的缓存对象, 到数据库去查询, 然后把查询结果保存到缓存容器中作为缓存对象
        object = statement.executeQueryForObject(statementScope, trans, parameter
Object, resultObject);
        cacheModel.putObject(cacheKey, object);
    }
    return object;
}
```

CacheKey 对象的 update 方法是通过哈希算法来实现标记过程的唯一码的。关于这个内容的介绍见本书第 10.2.3 节的“唯一性 CacheKey 对象的产生”的内容。

在开始, 通过调用 MappedStatement 类的 getCacheKey 方法, 获取唯一码的代码如下。

```
public CacheKey getCacheKey(StatementScope statementScope, Object parameterObject) {
    Sql sql = statementScope.getSql();
    ParameterMap pmap = sql.getParameterMap(statementScope, parameterObject);
    CacheKey cacheKey = pmap.getCacheKey(statementScope, parameterObject);
    cacheKey.update(id);
    cacheKey.update(baseCacheKey);
    cacheKey.update(sql.getSql(statementScope, parameterObject));
    return cacheKey;
}
```

对于最开始的唯一码的形成, 主要来源于三个特性: 第一是传入的参数及其数据形成的哈希码, 第二是当前 MappedStatement 的 id 和 baseCacheKey 变量的哈希码, 第三是进行操作的 SQL 语句的哈希码。

关于这个唯一码的形成, 可以进行专门的说明。

9.3.6 自动生成的主键

很多数据库支持自动生成主键的数据类型。SQL Map 通过 <insert> 的子元素 <selectKey> 来支持自动生成的键值, 它同时支持预生成 (如 Oracle) 和后生成两种类型 (如 MS-SQL Server)。Oracle 的自动主键生成的例子如下。


```

!-Oracle SEQUENCE Example -->
<insert id="insertProduct-ORACLE" parameterClass="com.domain.Product">
  <selectKey resultClass="int" keyProperty="id" >
    SELECT STOCKIDSEQUENCE.NEXTVAL AS ID FROM DUAL
  </selectKey>
  insert into PRODUCT (PRD_ID,PRD_DESCRIPTION) values (#id#,#description#)
</insert>

```

MS-SQL Server 的自动主键生成的例子配置信息如下。

```

<!-- Microsoft SQL Server IDENTITY Column Example -->
<insert id="insertProduct-MS-SQL" parameterClass="com.domain.Product">
  insert into PRODUCT (PRD_DESCRIPTION) values (#description#)
  <selectKey resultClass="int" keyProperty="id" >
    SELECT @@IDENTITY AS ID
  </selectKey>
</insert>

```

在解析 insert 节点过程中,对于 InsertStatement 对象,在 SqlStatementParser 对象中有这么一个私有方法 findAndParseSelectKey,其主要就是处理自动生成主键。其程序代码如下。

```

private void findAndParseSelectKey(Node node, MappedStatementConfig config) {
    state.getConfig().getErrorContext().setActivity("parsing select key tags");
    boolean foundSQLFirst = false;
    NodeList children = node.getChildNodes();
    for (int i = 0; i < children.getLength(); i++) {
        Node child = children.item(i);
        if (child.getNodeType() == Node.CDATA_SECTION_NODE
            || child.getNodeType() == Node.TEXT_NODE) {
            String data = ((CharacterData) child).getData();
            if (data.trim().length() > 0) {
                foundSQLFirst = true;
            }
        } else if (child.getNodeType() == Node.ELEMENT_NODE
            && "selectKey".equals(child.getNodeName())) {
            Properties attributes = NodeletUtils.parseAttributes(child, state.
                getGlobalProps());
            String keyPropName = attributes.getProperty("keyProperty");
            String resultClassName = attributes.getProperty("resultClass");
            String type = attributes.getProperty("type");
            config.setSelectKeyStatement(new XmlSqlSource(state, child), result
                ClassName, keyPropName, foundSQLFirst, type);
            break;
        }
    }
    state.getConfig().getErrorContext().setMoreInfo(null);
}

```

当判断在 insert 节点下有 selectKey 时,便对 selectKey 节点进行解析,并把相关内容都转化到 InsertStatement 对象中。在这里稍微解释一下,如果是先写 selectKey 节点,再写 insert

节点的 SQL 语句,那么就属于先调用模式,即先获得主键,再进行 Insert 操作。如果是先写 insert 节点的 SQL 语句,再写 selectKey 节点,那么就是属于后调用模式,即在进行 Insert 操作时才获取主键。调用 MappedStatementConfig 对象的 setSelectKeyStatement 方法的代码如下。

```
public void setSelectKeyStatement(SqlSource processor, String resultClassName,
String keyPropName, boolean runAfterSQL, String type) {
    if (rootStatement instanceof InsertStatement) {
        InsertStatement insertStatement = ((InsertStatement) rootStatement);
        Class parameterClass = insertStatement.getParameterClass();
        errorContext.setActivity("parsing a select key");
        SelectKeyStatement selectKeyStatement = new SelectKeyStatement();
        resultClassName = typeHandlerFactory.resolveAlias(resultClassName);
        Class resultClass = null;

        // get parameter and result maps
        selectKeyStatement.setSqlMapClient(client);
        selectKeyStatement.setId(insertStatement.getId() + "-SelectKey");
        selectKeyStatement.setResource(errorContext.getResource());
        selectKeyStatement.setKeyProperty(keyPropName);
        selectKeyStatement.setRunAfterSQL(runAfterSQL);
        // process the type (pre or post) attribute
        if (type != null) {
            selectKeyStatement.setRunAfterSQL("post".equals(type));
        }
        try {
            if (resultClassName != null) {
                errorContext.setMoreInfo("Check the select key result class.");
                resultClass = Resources.classForName(resultClassName);
            } else {
                if (keyPropName != null && parameterClass != null) {
                    resultClass = PROBE.getPropertyTypeForSetter(parameterClass,
selectKeyStatement.getKeyProperty());
                }
            }
        } catch (ClassNotFoundException e) {
            throw new SqlMapException("Error. Could not set result class. Cause:
" + e, e);
        }
        if (resultClass == null) {
            resultClass = Object.class;
        }

        // process SQL statement, including inline parameter maps
        errorContext.setMoreInfo("Check the select key SQL statement.");
        Sql sql = processor.getSql();
        setSqlForStatement(selectKeyStatement, sql);
        ResultMap resultMap;
        resultMap = new AutoResultMap(client.getDelegate(), false);
        resultMap.setId(selectKeyStatement.getId() + "-AutoResultMap");
        resultMap.setResultClass(resultClass);
    }
}
```

```

        resultMap.setResource(selectKeyStatement.getResource());
        selectKeyStatement.setResultMap(resultMap);
        errorContext.setMoreInfo(null);
        insertStatement.setSelectKeyStatement(selectKeyStatement);
    } else {
        throw new SqlMapException("You cant set a select key statement on statement
named " + rootStatement.getId() + " because it is not an InsertStatement.");
    }
}

```

这样形成的结果是,在 InsertStatement 对象中的私有属性 selectKeyStatement 是存在的,而且是一个 SelectKeyStatement 类型的实例化对象。

当应用程序开始执行带自动主键的 Insert 操作时,在 SqlMapExecutorDelegate 对象的 insert 方法中,有如下的程序代码。

```

SelectKeyStatement selectKeyStatement = null;
if (ms instanceof InsertStatement) {
    selectKeyStatement = ((InsertStatement) ms).getSelectKeyStatement();
}

// Here we get the old value for the key property. We'll want it later if for
some reason the
// insert fails.
Object oldKeyValue = null;
String keyProperty = null;
boolean resetKeyValueOnFailure = false;
if (selectKeyStatement != null && !selectKeyStatement.isRunAfterSQL()) {
    keyProperty = selectKeyStatement.getKeyProperty();
    oldKeyValue = PROBE.getObject(param, keyProperty);
    generatedKey = executeSelectKey(sessionScope, trans, ms, param);
    resetKeyValueOnFailure = true;
}

```

当然,如果在解析中 InsertStatement 对象存在私有属性 SelectKeyStatement 对象,那就执行判断内的方法,内容如下:

```

private Object executeSelectKey(SessionScope sessionScope, Transaction trans,
MappedStatement ms, Object param) throws SQLException {
    Object generatedKey = null;
    StatementScope statementScope;
    InsertStatement insert = (InsertStatement) ms;
    SelectKeyStatement selectKeyStatement = insert.getSelectKeyStatement();
    if (selectKeyStatement != null) {
        statementScope = beginStatementScope(sessionScope, selectKeyStatement);
        try {
            generatedKey = selectKeyStatement.executeQueryForObject(statementScope,
trans, param, null);
            String keyProp = selectKeyStatement.getKeyProperty();
            if (keyProp != null) {
                PROBE.setObject(param, keyProp, generatedKey);
            }
        }
    }
}

```

```

    }
    } finally {
        endStatementScope(statementScope);
    }
}
return generatedKey;
}

```

这样，通过一个执行 SQL 语句并获得返回值，就取得了自动生成的主键。

9.4 Sql 框架及其说明

在 iBATIS SQL Map 中，Sql 框架中的 Sql 语句可以使用 JDBC 支持的任意的 SQL 语句，甚至是多条语句。Sql 框架主要是由 Sql 接口及其实现类和 SqlChild 接口及其实现类组成的。它主要存在 com.ibatis.sqlmap.engine.mapping.sql 包中。

9.4.1 Sql 接口框架

Sql 接口框架包括 Sql 接口和该接口的实现类，这些实现类有 SimpleDynamicSql 类、DynamicSql 类、RawSql 类和 StaticSql 类。Sql 接口及其实现类结构类如图 9-11 所示。

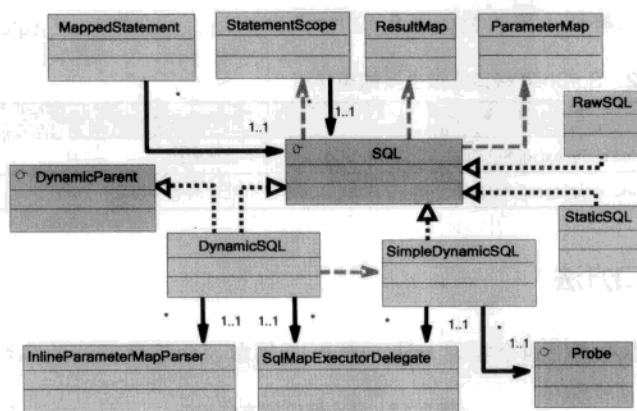


图 9-11 Sql 接口类结构图

Sql 接口是核心。Sql 接口被 MappedStatement 类关联，然后依赖 ParameterMap 类、ResultMap 类和 StatementScope 类。这也使得 Sql 接口的实现类也依赖 ParameterMap 类、ResultMap 类和 StatementScope 类。RawSql 类、StaticSql 类、SimpleDynamicSql 类、DynamicSql 类实现 Sql 接口。SimpleDynamicSql 类关联 Probe 接口，主要是用于处理一些 bean、DOM 对象和其他对象的转化。SimpleDynamicSql 类关联 SqlMapExecutorDelegate 类，主要是实现 SqlMapExecutorDelegate 对 Sql 接口的实现类引用。DynamicSql 类关联

InlineParameterMapParser 类，主要是用于处理一些基本 SQL 语句（即是静态 SQL 语句要处理的内容）。DynamicSql 类关联 SqlMapExecutorDelegate 类，道理与 SimpleDynamicSql 类关联 SqlMapExecutorDelegate 类相同。DynamicSql 类依赖 SimpleDynamicSql 类，主要是用于处理一些附加的 sql 语句（即是简单动态 SQL 的实现要处理的内容）。

Sql 接口框架主要的类介绍如表 9-11 所示。

表 9-11 Sql 接口框架主要的类介绍

接 口 或 类	功能或用途描述	备 注
com.ibatis.sqlmap.engine.mapping.sql.Sql	映射语句 SQL 文本的接口	
com.ibatis.sqlmap.engine.mapping.sql.RawSql	映射语句 SQL 文本的抽象实现类，在系统中基本上没有使用，该类更趋向测试	
com.ibatis.sqlmap.engine.mapping.sql.StaticSql	映射语句的 SQL 文本的静态语句实现类	
com.ibatis.sqlmap.engine.mapping.sql.SimpleDynamicSql	映射语句的 SQL 文本的简单动态语句实现类	
com.ibatis.sqlmap.engine.mapping.sql.DynamicSql	映射语句的 SQL 文本的动态语言实现类	

9.4.2 SqlChild 接口框架

SqlChild 接口主要对 SQL 文本进行一些粗加工，对 SQL 语句去掉一些回车、表格等内容。SqlChild 接口及其实现类结构类如图 9-12 所示。

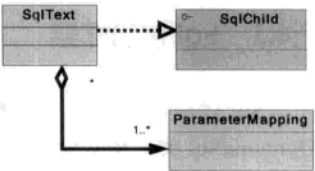


图 9-12 SqlChild 接口类结构图

SqlChild 接口框架主要的类如表 9-12 所示。

表 9-12 SqlChild 接口框架主要的类介绍

接 口 或 类	功能或用途描述	备 注
com.ibatis.sqlmap.engine.mapping.sql.SqlChild	SQL 文本简单处理的接口	
com.ibatis.sqlmap.engine.mapping.sql.SqlText	SQL 文本简单处理的实现类	

9.4.3 Sql 接口方法

Sql 接口主要是调用四个实现方法，表 9-13 简单说明这些方法的内容。

表 9-13 Sql 接口的方法说明

方 法 名 称	参 数	返 回 值	功能和用途
getSql	StatementScope , Object	String	通过传入的状态参数 statementScope 和参数对象，获得当前 Sql 对象的 SQL 字符串。
getParameterMap	StatementScope , Object	ParameterMap	通过传入的状态参数 statementScope 和参数对象，获得当前 Sql 对象的 ParameterMap 对象
getResultMap	StatementScope , Object parameterObject	ResultMap	获得一个 Javabeans 的属性的 set 方法的参数类型。
cleanup	StatementScope	无	清除状态参数 StatementScope 对象的关于本 Sql 对象的信息。

对这四个方法的说明:

```
public String getSql(StatementScope statementScope, Object parameterObject);
```

该方法通过传入的状态参数 `statementScope` 和参数对象, 获得当前 `Sql` 对象的 `SQL` 字符串。其中第一个参数表示状态参数 `StatementScope` 对象, 第二个参数表示当前对象对应的参数对象。

```
public ParameterMap getParameterMap(StatementScope statementScope, Object parameterObject);
```

该方法通过传入的状态参数 `statementScope` 和参数对象, 获得当前 `Sql` 对象的 `ParameterMap` 对象。其中第一个参数表示状态参数 `StatementScope` 对象, 第二个参数表示当前对象对应的参数对象。

```
public ResultMap getResultMap(StatementScope statementScope, Object parameterObject);
```

该方法获得一个 `JavaBean` 属性的 `set` 方法的参数类型。其中第一个参数表示状态参数 `StatementScope` 对象, 第二个参数表示当前对象对应的参数对象。

```
public void cleanup(StatementScope statementScope)
```

该方法清除状态参数 `StatementScope` 对象的关于本 `Sql` 对象的信息。传入参数表示一个状态参数 `StatementScope` 对象。该方法只对 `DynamicSql` 类有用。

在对 `Statement` 节点的解析过程中, 有专门对 `SQL` 语句的解析。这种 `SQL` 语句分为三种类型: 一种是动态 `SQL` 语句, 另一种是静态 `SQL` 语句, 还有一种是动态 `SQL` 语句的简化版本, 即简单动态 `SQL` 语句。下面分别介绍这几个具体类的实现过程。

9.4.4 静态 SQL 的实现

`StaticSql` 类主要针对静态 `SQL` 语句, 在 `com.ibatis.sqlmap.engine.mapping.sql.stat` 包中, 主要内容是实现 `SQL` 里的 `#` 号处理, 一般处理如下配置的语句。

```
<statement id="insertProduct" >
    insert into PRODUCT (PRD_ID, PRD_DESCRIPTION) values (#id#, #description#);
</statement>
```

静态 `SQL` 语言实现比较简单, 通过 `XML` 的解析, 再通过 `InlineParameterMapParser` 的转化, 上述例子的 `SQL` 语句变成了如下形式。

```
insert into PRODUCT (PRD_ID, PRD_DESCRIPTION) values (?, ?)
```

这样的语句, 基本上是 `JDBC` 中可以接受的 `SQL` 语句了。

由于在解析过程中已经调用了 `InlineParameterMapParser` 转化, 所以下面来看看保存在 `StaticSql` 对象中的代码是如何实现的。

(1) getSql 方法的实现

StaticSql 类 getSql 方法代码如下。

```
private String sqlStatement;
public String getSql(StatementScope statementScope, Object parameterObject) {
    return sqlStatement;
}
```

直接获得当前 StaticSql 实例化对象的字符串变量 sqlStatement。

(2) getParameterMap 方法的实现

StaticSql 类 getParameterMap 方法代码如下。

```
public ParameterMap getParameterMap(StatementScope statementScope, Object parameterObject) {
    return statementScope.getParameterMap();
}
```

获取 StatementScope 对象的 ParameterMap 属性。

(3) getResultMap 方法的实现

StaticSQL 类 getResultMap 语句代码如下。

```
public ResultMap getResultMap(StatementScope statementScope, Object parameterObject) {
    return statementScope.getResultMap();
}
```

获取 StatementScope 对象的 ResultMap 属性。

(4) cleanup 方法的实现

StaticSQL 类 getResultMap 方法没有执行代码。

9.4.5 简单动态 SQL 的实现

SimpleDynamicSql 类主要处理简单动态 SQL，在 com.ibatis.sqlmap.engine.mapping.sql.simple 包中，SQL statement 和 statement 都可以包含简单的动态 SQL 元素。简单动态 SQL 主要针对 SQL 里有标志\$符号，一般处理如下配置的语句。

```
<statement id="getProduct" resultMap="get-product-result">
    select * from PRODUCT order by $preferredOrder$
</statement>
```

上面的例子中，preferredOrder 动态元素将被参数对象的 preferredOrder 属性值替换（象 parameter map 一样）。

(1) getSQL 方法的实现

StaticSQL 类 getSQL 方法代码如下。

```
public String getSql(StatementScope statementScope, Object parameterObject) {
```



```
return processDynamicElements(sqlStatement, parameterObject);
}
```

SimpleDynamicSql 在对 SQL 语句的处理前已经通过了 InlineParameterMapParser 的转化,也就是已经处理标识符为#的 Sql 语句。接着我们看看调用 SimpleDynamicSql 类的 processDynamicElements 方法。

```
private static final String ELEMENT_TOKEN = "$";
private String processDynamicElements(String sql, Object parameterObject) {
    StringTokenizer parser = new StringTokenizer(sql, ELEMENT_TOKEN, true);
    StringBuffer newSql = new StringBuffer();

    String token = null;
    String lastToken = null;
    while (parser.hasMoreTokens()) {
        token = parser.nextToken();

        if (ELEMENT_TOKEN.equals(lastToken)) {
            if (ELEMENT_TOKEN.equals(token)) {
                newSql.append(ELEMENT_TOKEN);
                token = null;
            } else {
                Object value = null;
                if (parameterObject != null) {
                    if (delegate.getTypeHandlerFactory().hasTypeHandler(parameterObject.
getClass())) {
                        value = parameterObject;
                    } else {
                        value = PROBE.getObject(parameterObject, token);
                    }
                }
                if (value != null) {
                    newSql.append(String.valueOf(value));
                }

                token = parser.nextToken();
                if (!ELEMENT_TOKEN.equals(token)) {
                    throw new SqlMapException("Unterminated dynamic element in sql
(" + sql + ").");
                }
                token = null;
            }
        } else {
            if (!ELEMENT_TOKEN.equals(token)) {
                newSql.append(token);
            }
        }
        lastToken = token;
    }
}
```



```

        return newSql.toString();
    }

```

该方法实现的功能是对 SQL 中\$中的字符串进行处理，首先是通过循环来进行字符串的遍历。调用 `delegate.getTypeHandlerFactory().hasTypeHandler(parameterObject.getClass())` 判断是否存在当前的 Bean，如果存在，则获得当前 bean 的属性。如果不是一个 bean，则判断是否是在 `TypeHandlerFactory` 中已经存在的数据类型，然后直接赋值。

(2) `getParameterMap` 方法的实现

`StaticSQL` 类 `getParameterMap` 方法的代码如下，与 `StaticSQL` 类相同。

```

public ParameterMap getParameterMap(StatementScope statementScope, Object parameter
Object) {
    return statementScope.getParameterMap();
}

```

获取 `StatementScope` 对象的 `ParameterMap` 属性。

(3) `getResultMap` 方法的实现

`StaticSQL` 类 `getResultMap` 语句代码如下，与 `StaticSQL` 类相同。

```

public ResultMap getResultMap(StatementScope statementScope, Object parameter
Object) {
    return statementScope.getResultMap();
}

```

获取 `StatementScope` 对象的 `ResultMap` 属性。

(4) `cleanup` 方法的实现

`StaticSql` 类的 `getResultMap` 方法没有执行代码，与 `StaticSql` 类相同。

9.4.6 动态 SQL 语言的实现

我们知道，数据库的 SQL 语句主要还是结构化语句，没有一些判断、循环等处理语句。但是一些复杂的处理也许要加入判断、循环的处理，除非在数据库中采用存储过程，或者再采用嵌入式 SQL 语句。

iBATIS SQL Map 也支持动态 SQL 语句。这也是 SQL Map 比较复杂的部分。在 SQL 中用 `<dynamic>` 元素划分出 SQL 语句的动态部分。动态部分可以包含任意多的条件标签元素，条件标签决定是否在语句中包含其中的 SQL 代码。所有的条件标签元素将根据传给动态查询 `Statement` 的参数对象的情况来工作。`<dynamic>` 元素和条件元素都有“prepend”属性，这也是动态 SQL 代码的一部分。

动态 SQL 主要针对 SQL 里有标志 `<dynamic>` 符号，一般处理如下配置的语句。

```

...
< 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。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到<dynamic>标签的内容体(body content)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。

1. SQL Map 的动态 SQL 组件结构

SQL Map 的动态 SQL 组件包括在 com.ibatis.sqlmap.engine.mapping.sql.simple 包内。对于这个框架内的分析，主要分为两个部分，一个是按照总体框架，还有一个是按照条件来进行分类。其类结构如图 9-13 所示。

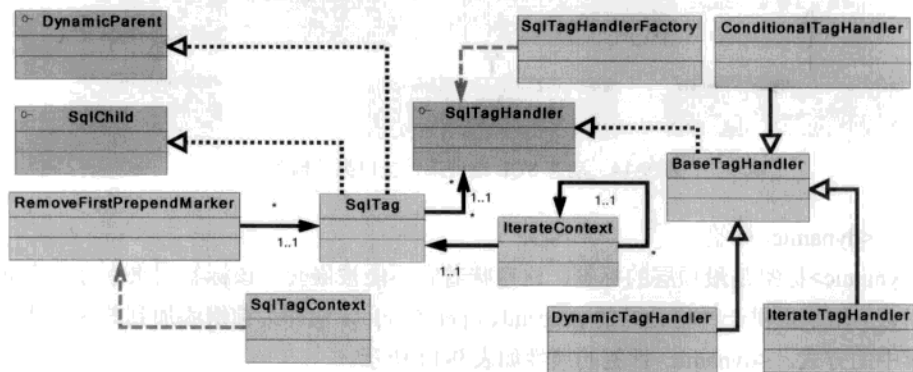


图 9-13 动态 SQL 组件整体接口类结构图

条件判断是动态语言的主要内容，SQL Map 的动态语言对于条件判断的类结构如图 9-14 所示。

由于具体的动态语言的解析和实现非常的繁琐和复杂，也不是本文的重点，所涉及的内容也是基于文本的解释和分析，所以在这里就不做详细分析说明，只是做一个简要说明。

2. SQL Map 的动态 SQL 语句的标签介绍

要搞明白 SQL Map 的动态 SQL 语句，首先要熟悉动态标签。

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五类：<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。所有的动态标签都有 prepend、open 和 close 这三个属性。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。没有什么方式可以阻止<dynamic>标签将该值加为前缀。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。

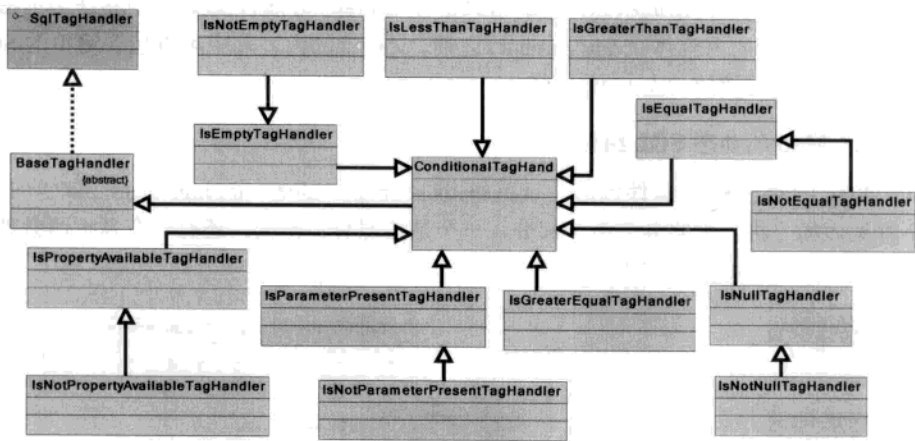


图 9-14 动态 SQL 组件条件接口类结构图

(1) <dynamic>标签

<dynamic>标签是最顶层的标签，这意味着它不能被嵌套。该标签用来划分一个动态 SQL 片段，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 9-14 所示。

表 9-14 <dynamic>标签的属性

序 号	节点属性名称	含 义
1	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
2	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（”
3	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

(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如何进行比较这些值, 二元标签的属性如表 9-15 所示。

表 9-15 二元标签的属性

序 号	节点属性名称	含 义
1	property (必需的)	参数对象中用于同 compareValue 或者 compareProperty 相比较的特性
2	prepend (可选的)	该值用于作为前缀附加到标签的结果内容体前。只有在以下三种情况下, 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> 标签后的一个生成内容体的标签
3	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)”
4	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
5	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)
6	compareProperty	(如果没有指定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, 则它是必需的)
7	compareValue	(如果没有指定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 compareProperty, 则它是必需的)

二元标签的类型有几种类型, 如表 9-16 所示。

表 9-16 二元标签的类型

序 号	标 签 名 称	含 义
1	<isEqual>	将 property 属性同 compareProperty 属性或 compareValue 属性相比较, 确定它们是否相同
2	<isNotEqual>	将 property 属性同 compareProperty 属性或 compareValue 属性相比较, 确定它们是否不同
3	<isGreaterThan>	确定 property 属性是否大于 compareProperty 属性或 compareValue 属性
4	<isGreaterEqual>	确定 property 属性是否大于等于 compareProperty 属性或 compareValue 属性
5	<isLessThan>	确定 property 属性是否小于 compareProperty 属性或 compareValue 属性
6	<isLessEqual>	确定 property 属性是否小于等于 compareProperty 属性或 compareValue 属性

(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 9-17 所示。

表 9-17 一元标签的属性

序 号	节点属性名称	含 义
1	property (必需的)	参数对象中用于状态比较的特殊性
2	prepend (可选的)	该值用于作为前缀附加到标签的结果内容体前。只有在以下三种情况下，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
3	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)”
4	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
5	removeFirstPrepend (可选的)	该属性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

一元标签的类别如表 9-18 所示。

表 9-18 一元标签的类别

序 号	标 签 名 称	含 义
1	<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
2	<isNotPropertyAvailable>	确定参数中是否不存在所指字的字段。对于 bean，它寻找一个特性；而对于 map，它寻找一个键
3	<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
4	<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
5	<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()
6	<isNotEmpty>	确定所指定的字段是否为非空、非空字符串、非空集合或者非空的 String.valueOf()

(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。参数标签的属性如表 9-19 所示。

表 9-19 参数标签的属性

序 号	节点属性名称	含 义
1	prepend (可选的)	该值用于作为前缀附加到标签的结果内容体前。只有在以下三种情况下，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
2	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")"
3	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
4	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

该标签有几种类型，如表 9-20 所示。

表 9-20 参数标签的类别

序 号	标 签 名 称	含 义
1	<isParameterPresent>	确定参数对象是否出现
2	<isNotParameterPresent>	确定参数对象是否存在

(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 9-21 所示。

表 9-21 iterate 标签的属性

序 号	节点属性名称	含 义
1	property (必需的)	包含列表的参数的特性
2	prepend (可选的)	该值用于作为前缀附加到标签的结果内容体前。只有在以下三种情况下，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
3	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")"
4	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
5	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
6	removeFirstPrepend	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

3. SQL Map 的动态 SQL 语言的实现

SQL Map 的动态 SQL 语言主要还是实现 Sql 接口的四个方法。

(1) getSql 方法的实现

DynamicSql 类 getSql 方法代码如下。

```
public String getSql(StatementScope statementScope, Object parameterObject) {
    String sql = statementScope.getDynamicSql();
    if (sql == null) {
        process(statementScope, parameterObject);
        sql = statementScope.getDynamicSql();
    }
    return sql;
}
```

调用 process 方法，其方法代码如下。

```
private void process(StatementScope statementScope, Object parameterObject) {
    SqlTagContext ctx = new SqlTagContext();
    List localChildren = children;
    processBodyChildren(statementScope, ctx, parameterObject, localChildren.iterator());

    ParameterMap map = new ParameterMap(delegate);
    map.setId(statementScope.getStatement().getId() + "-InlineParameterMap");
    map.setParameterClass(((MappedStatement) statementScope.getStatement()).getParameterClass());
    map.setParameterMappingList(ctx.getParameterMappings());

    String dynSql = ctx.getBodyText();

    // Processes $substitutions$ after DynamicSql
    if (SimpleDynamicSql.isSimpleDynamicSql(dynSql)) {
        dynSql = new SimpleDynamicSql(delegate, dynSql).getSql(statementScope, parameterObject);
    }

    statementScope.setDynamicSql(dynSql);
    statementScope.setDynamicParameterMap(map);
}
```

首先要实例化一个 SqlTagContext 对象，然后通过解析中生成的 XmlSqlSource 对象对 SqlTagContext 对象进行处理，然后通过调用 SqlTagContext 对象的 getBodyText 方法获得处理后的 Sql 语句。再然后交给 SimpleDynamicSql 去处理。

SimpleDynamicSql 对 SQL 的处理前已经通过了 InlineParameterMapParser 的转化，也就是已经处理标识符为#的 SQL 语句。接着再看看调用 SimpleDynamicSql 类的 processDynamicElements 方法。

该方法实现的功能是对 SQL 中 \$ 中的字符串进行处理，首先是通过循环来进行字符串的遍历。调用 `delegate.getTypeHandlerFactory().hasTypeHandler(parameterObject.getClass())` 判断是否存在当前的 Bean，如果存在，则获得当前 bean 的属性。如果不是一个 bean，则判断是否是 `TypeHandlerFactory` 中已经存在的数据类型，然后直接赋值。

(2) `getParameterMap` 方法的实现

`DynamicSql` 类 `getParameterMap` 方法代码如下，与 `StaticSql` 类相同。

```
public ParameterMap getParameterMap(StatementScope statementScope, Object parameterObject) {
    ParameterMap map = statementScope.getDynamicParameterMap();
    if (map == null) {
        process(statementScope, parameterObject);
        map = statementScope.getDynamicParameterMap();
    }
    return map;
}
```

获取 `StatementScope` 对象的 `ParameterMap` 对象。

(3) `getResultMap` 方法的实现

`DynamicSql` 类 `getResultMap` 语句代码如下，与 `StaticSql` 类相同。

```
public ResultMap getResultMap(StatementScope statementScope, Object parameterObject) {
    return statementScope.getResultMap();
}
```

获取 `StatementScope` 对象的 `ResultMap` 对象。

(4) `cleanup` 方法的实现

`DynamicSql` 类 `getResultMap` 方法没有执行代码如下，清空环境变量。

```
public void cleanup(StatementScope statementScope) {
    statementScope.setDynamicSql(null);
    statementScope.setDynamicParameterMap(null);
}
```

9.5 数据对象转换框架及其说明

数据对象转化框架主要包括 `DataExchange` 组件和 `Accessplan` 组件。由于在 iBATIS 中，传入的参数可能是 `Map`、`List`、`Object`、`JavaBean` 等。这些参数要经过转化才能被 JDBC 的 SQL 语法所理解。同样，在执行完成 SQL 查询后，形成的数据要经过类型的转化才能重新生成 `Map`、`List`、`Object`、`JavaBean` 等。而这项工作，主要就是由 `DataExchange` 组件和 `Accessplan` 组件来完成的。

9.5.1 DataExchange 组件作用、内容和设计模式

DataExchange 主要用于数据对象的转化，其内容主要在 com.ibatis.sqlmap.engine.exchange 包内。这些对象包括 DOM 对象、List 对象、Map 对象、beans 对象等。DataExchange 组件采用的是工厂方法设计模式，类结构如图 9-15 所示。

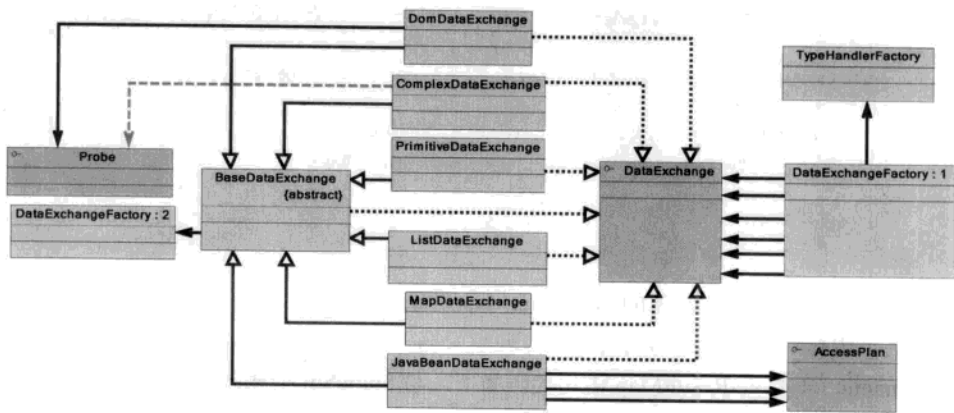


图 9-15 DataExchange 组件类结构图

ComplexDataExchange 类、DomDataExchange 类、JavaBeanDataExchange 类、ListDataExchange 类、MapDataExchange 类、PrimitiveDataExchange 类都是继承 BaseDataExchange 抽象类并实现 DataExchange 接口。DataExchangeFactory 类五次关联 DataExchange 接口，分别映射 ComplexDataExchange 类、DomDataExchange 类、ListDataExchange 类、MapDataExchange 类和 PrimitiveDataExchange 类。同时 BaseDataExchange 抽象类关联 DataExchangeFactory 类。JavaBeanDataExchange 类三次关联 AccessPlan 接口。在图 9-15 中，DataExchangeFactory:1 和 DataExchangeFactory:2 是同一个类，即 DataExchangeFactory 类。DataExchangeFactory 类外部关联 TypeHandlerFactory 类，而 BaseDataExchange 抽象类外部关联 DataExchangeFactory 类。在这里把 TypeHandler 组件、DataExchange 组件、AccessPlan 组件和 Probe 组件结合在一块了。

DataExchange 所涉及的相关类和接口如表 9-22 所示。

表 9-22 DataExchange 组件主要的类和接口列表

接口或类	功能或用途描述	备注
com.ibatis.sqlmap.engine.exchange.DataExchangeFactory	DataExchange 的工厂类	类
com.ibatis.sqlmap.engine.exchange.DataExchange	基于参数 Map 或结果 Map 和相关对象进行数据交换的接口	接口
com.ibatis.sqlmap.engine.exchange.BaseDataExchange	DataExchange 实现的抽象类	抽象类
com.ibatis.sqlmap.engine.exchange.ComplexDataExchange	JavaBean 集合的 DataExchange 实现	类

续表

接口或类	功能或用途描述	备 注
com.ibatis.sqlmap.engine.exchange.DomDataExchange	Dom 对象的 DataExchange 实现	类
com.ibatis.sqlmap.engine.exchange.BeanDataExchange	JavaBean 对象的 DataExchange 实现	类
com.ibatis.sqlmap.engine.exchange.ListDataExchange	List 对象的 DataExchange 实现	类
com.ibatis.sqlmap.engine.exchange.MapDataExchange	Map 对象的 DataExchange 实现	类
com.ibatis.sqlmap.engine.exchange.PrimitiveDataExchange	基本数据对象的 DataExchange 实现	类

1. DataExchangeFactory 类介绍

对 DataExchange 的应用都是通过调用 DataExchangeFactory 来创建的，DataExchangeFactory 的构造方法代码如下。

```
public DataExchangeFactory(TypeHandlerFactory typeHandlerFactory) {
    this.typeHandlerFactory = typeHandlerFactory;
    domDataExchange = new DomDataExchange(this);
    listDataExchange = new ListDataExchange(this);
    mapDataExchange = new ComplexDataExchange(this);
    primitiveDataExchange = new PrimitiveDataExchange(this);
    complexDataExchange = new ComplexDataExchange(this);
}
```

当 ParameterMap 对象要实现 DataExchange 接口的具体实现类时，要依据 ParameterMap 对象本身的 Class 私有属性内容。不同的 Class 创建不同 DataExchange 实现类。程序代码在 DataExchangeFactory 的 getDataExchangeForClass 方法，代码如下：

```
public DataExchange getDataExchangeForClass(Class clazz) {
    DataExchange dataExchange = null;
    if (clazz == null) {
        //当用户没有定义 Class 时，返回 complexDataExchange 对象
        dataExchange = complexDataExchange;
    } else if (DomTypeMarker.class.isAssignableFrom(clazz)) {
        dataExchange = domDataExchange;
    } else if (List.class.isAssignableFrom(clazz)) {
        dataExchange = listDataExchange;
    } else if (Map.class.isAssignableFrom(clazz)) {
        dataExchange = mapDataExchange;
    } else if (typeHandlerFactory.getTypeHandler(clazz) != null) {
        dataExchange = primitiveDataExchange;
    } else {
        dataExchange = new BeanDataExchange(this);
    }
    return dataExchange;
}
```

这种调用方式同样也适合 resultMap 对象和 resultMap 对象的 Class 私有属性。在 DataExchange 接口实现过程中，ParameterMap 对象只需要如下三个方法。

```
public void initialize(Map properties);
public Object[] getData(StatementScope statementScope, ParameterMap parameterMap,
Object parameterObject);
public Object setData(StatementScope statementScope, ParameterMap parameterMap,
Object parameterObject, Object[] values);
```

而 ResultMap 对象只需要如下两个方法。

```
public void initialize(Map properties);
public Object setData(StatementScope statementScope, ResultMap resultMap, Object
resultObject, Object[] values);
```

2. DataExchange 接口方法

DataExchange 组件还有一个重要的功能就是实现 DataExchange 接口。DataExchange 接口的方法列表和说明如表 9-23 所示。

表 9-23 DataExchange 接口的方法列表和说明

方法名称	参数	返回值	功能和用途	备注
initialize	Map	无	初始化 exchange 实例化对象	
getData	StatementScope ParameterMap Object	Object[]	实例化对象中获取所有的 Parameter 对象并形成对象数组	
setData	StatementScope ResultMap Object Object[] values	Object	从对象数组中设置一个对象到 resultMap 实例化对象中	
setData	StatementScope ParameterMap Object Object[]	Object	从对象数组中设置一个对象到 ParameterMap 实例化对象中	
getCacheKey	StatementScope ParameterMap Object	CacheKey	Returns an object capable of being a unique cache key for a parameter object	

后面分别说明 DomDataExchange 类、MapDataExchange 类和 JavaBeanDataExchange 类的实现。但是 JavaBeanDataExchange 类的最终实现，还要与 Accessplan 组件相结合，这里只是介绍一下 JavaBeanDataExchange 调用 Accessplan 接口的方法，关于 Accessplan 组件的实现部分放到 Accessplan 组件设计模式进行说明。由于 DomDataExchange 类、MapDataExchange 类和 JavaBeanDataExchange 类都继承 BaseDataExchange 抽象类，所以还是先从 BaseDataExchange 抽象类进行说明。

3. BaseDataExchange 抽象类的实现

BaseDataExchange 抽象类主要是实现了 getCacheKey 方法，其代码如下：

```

public CacheKey getCacheKey(StatementScope statementScope, ParameterMap parameter
Map, Object parameterObject) {
    CacheKey key = new CacheKey();
    Object[] data = getData(statementScope, parameterMap, parameterObject);
    for (int i = 0; i < data.length; i++) {
        if (data[i] != null) {
            key.update(data[i]);
        }
    }
    return key;
}

```

创建一个缓存标识码的 `CacheKey` 对象，然后把当前 `StatementScope` 对象中 `ParameterMap` 的 `parameter` 参数内容转化为一个对象数组。然后进行循环调用 `update` 方法，这样可以形成一个只有唯一内码的缓存标识码。

4. PrimitiveDataExchange 类的实现

`PrimitiveDataExchange` 类主要是基本数据类型的转换，实现最简单。

(1) `getData(StatementScope statementScope, ParameterMap parameterMap, Object parameterObject)` 方法实现

该 `getData` 方法是根据简单数据类型对 `ParameterMap` 对象进行取值处理。

```

public Object[] getData(StatementScope statementScope, ParameterMap parameterMap,
Object parameterObject) {
    ParameterMapping[] mappings = parameterMap.getParameterMappings();
    Object[] data = new Object[mappings.length];
    for (int i = 0; i < mappings.length; i++) {
        data[i] = parameterObject;
    }
    return data;
}

```

按照 `ParameterMapping` 的名称和数量，通过循环直接赋值并形成对象数组。

(2) `setData(StatementScope statementScope, ResultMap resultMap, Object resultObject, Object[] values)` 方法实现

该 `setData` 方法是根据简单数据类型对 `ResultMap` 对象进行赋值处理。

```

public Object setData(StatementScope statementScope, ResultMap resultMap, Object
resultObject, Object[] values) {
    return values[0];
}

```

没有赋值操作。

(3) `setData(StatementScope statementScope, ParameterMap parameterMap, Object parameterObject, Object[] values)` 方法实现

该 `setData` 方法是根据简单数据类型对 `ParameterMap` 对象进行赋值处理。

```

    public Object setData(StatementScope statementScope, ResultMap resultMap, Object
resultObject, Object[] values) {
        return values[0];
    }

```

没有赋值操作。

5. ListDataExchange 类的实现

ListDataExchange 类是处理 List 对象。

(1) getData(StatementScope statementScope, ParameterMap parameterMap, Object parameterObject)方法实现

该 getData 方法是根据 List 对象和 ParameterMap 对象来进行取值处理。

```

    public Object[] getData(StatementScope statementScope, ParameterMap parameterMap,
Object parameterObject) {
        ParameterMapping[] mappings = parameterMap.getParameterMappings();
        Object[] data = new Object[mappings.length];
        for (int i = 0; i < mappings.length; i++) {
            String propName = mappings[i].getPropertyName();
            // parse on the '.' notation and get nested properties
            String[] propertyArray = propName.split("\\.");
            if (propertyArray.length > 0) {
                // iterate list of properties to discover values
                Object tempData = parameterObject;
                for (int x=0; x<propertyArray.length; x++) {
                    // is property an array reference
                    int arrayStartIndex = propertyArray[x].indexOf('[');
                    if (arrayStartIndex == -1) {
                        // is a normal property
                        tempData = ProbeFactory.getProbe().getObject(tempData, property
Array[x]);
                    } else {
                        int index = Integer.parseInt(propertyArray[x].substring(array
StartIndex + 1, propertyArray[x].length() - 1));
                        tempData = ((List) tempData).get(index);
                    }
                }
                data[i] = tempData;
            } else {
                int index = Integer.parseInt((propName.substring(propName.indexOf('[')
+ 1, propName.length() - 1)));
                data[i] = ((List) parameterObject).get(index);
            }
        }
        return data;
    }

```

按照 ParameterMapping 的名称和数量，直接获得对应于 List 中的 Value，进行循环取值，形成一个对象数组。但是有一个小插曲，就是要 ParameterMapping 的名称是否有“.”，

否则要进行循环取值。

(2) setData(StatementScope statementScope, ResultMap resultMap, Object resultObject, Object[] values) 方法实现

该 setData 方法是根据 List 对象和 ResultMap 对象来进行的赋值处理。

```
public Object setData(StatementScope statementScope, ResultMap resultMap, Object
resultObject, Object[] values) {
    ResultMapping[] mappings = resultMap.getResultMappings();
    List data = new ArrayList();
    for (int i = 0; i < mappings.length; i++) {
        String propName = mappings[i].getPropertyName();
        int index = Integer.parseInt((propName.substring(1, propName.length() -
1)));
        data.set(index, values[i]);
    }
    return data;
}
```

按照 ResultMap 的名称和数量，直接获得对应于 List 中的 Value，进行循环赋值。

(3) setData(StatementScope statementScope, ParameterMap parameterMap, Object parameterObject, Object[] values)方法实现

该 setData 方法是根据 List 对象和 ParameterMap 对象来进行的赋值处理。

```
public Object setData(StatementScope statementScope, ParameterMap parameterMap,
Object parameterObject, Object[] values) {
    ParameterMapping[] mappings = parameterMap.getParameterMappings();
    List data = new ArrayList();
    for (int i = 0; i < mappings.length; i++) {
        if (mappings[i].isOutputAllowed()) {
            String propName = mappings[i].getPropertyName();
            int index = Integer.parseInt((propName.substring(1, propName.length()
- 1)));
            data.set(index, values[i]);
        }
    }
    return data;
}
```

按照 ParameterMap 的名称和数量，直接获得对应于 List 中的 Value，进行循环赋值。

6. ComplexDataExchange 类的实现

DomDataExchange 类是处理复杂 Object。

(1) getData(StatementScope statementScope, ParameterMap parameterMap, Object parameterObject)方法实现

该 getData 方法是根据复杂 Object 和 ParameterMap 对象来进行的取值处理。

```
public Object[] getData(StatementScope statementScope, ParameterMap parameterMap,
```



```

Object parameterObject) {
    TypeHandlerFactory typeHandlerFactory = getDataExchangeFactory().getTypeHandlerFactory();
    if (parameterObject == null) {
        return new Object[1];
    } else {
        if (typeHandlerFactory.hasTypeHandler(parameterObject.getClass())) {
            ParameterMapping[] mappings = parameterMap.getParameterMappings();
            Object[] data = new Object[mappings.length];
            for (int i = 0; i < mappings.length; i++) {
                data[i] = parameterObject;
            }
            return data;
        } else {
            Object[] data = new Object[parameterMap.getParameterMappings().length];
            ParameterMapping[] mappings = parameterMap.getParameterMappings();
            for (int i = 0; i < mappings.length; i++) {
                data[i] = PROBE.getObject(parameterObject, mappings[i].getProperty
Name());
            }
            return data;
        }
    }
}

```

按照 ParameterMapping 的名称和数量, 直接获得对应于复杂 Object 中的各个属性, 通过反射机制进行循环取值, 形成一个对象数组。但是有 1 个小插曲, 就是当前复杂对象是属于 TypeHandlerFactory 中已经存在的 Class, 即形成一个只有复杂对象的对象数组。如果当前复杂对象是不属于 TypeHandlerFactory 中已经存在的 Class, 即形成一个只有复杂对象各个属性的对象数组。

(2) setData(StatementScope statementScope, ResultMap resultMap, Object resultObject, Object[] values) 方法实现

该 setData 方法是根据复杂 Object 和 ParameterMap 对象来进行取值处理。

```

public Object setData(StatementScope statementScope, ResultMap resultMap, Object
resultObject, Object[] values) {
    TypeHandlerFactory typeHandlerFactory = getDataExchangeFactory().getTypeHandlerFactory();
    if (typeHandlerFactory.hasTypeHandler(resultMap.getResultClass())) {
        return values[0];
    } else {
        Object object = resultObject;
        if (object == null) {
            try {
                object = ResultObjectFactoryUtil.createObjectThroughFactory (resultMap.getResultClass());
            } catch (Exception e) {
                throw new RuntimeException("JavaBeansDataExchange could not instant-

```

```

        tiate result class. Cause: " + e, e);
    }
}
ResultMapping[] mappings = resultMap.getResultMappings();
for (int i = 0; i < mappings.length; i++) {
    PROBE.setObject(object, mappings[i].getPropertyName(), values[i]);
}
return object;
}
}

```

按照 resultMap 的名称和数量, 直接获得复杂 Object 中各个属性对应数组对象的 Value, 进行循环赋值。

(3) setData(StatementScope statementScope, ParameterMap parameterMap, Object parameterObject, Object[] values)方法实现

该 setData 方法是根据复杂 Object 和 ParameterMap 对象来进行取值处理。

```

public Object setData(StatementScope statementScope, ParameterMap parameterMap,
Object parameterObject, Object[] values) {
    TypeHandlerFactory typeHandlerFactory = getDataExchangeFactory().getType
HandlerFactory();
    if (typeHandlerFactory.hasTypeHandler(parameterMap.getParameterClass())) {
        return values[0];
    } else {
        Object object = parameterObject;
        if (object == null) {
            try {
                object = ResultObjectFactoryUtil.createObjectThroughFactory
(parameterMap.getParameterClass());
            } catch (Exception e) {
                throw new RuntimeException("JavaBeansDataExchange could not
instantiate result class. Cause: " + e, e);
            }
        }
        ParameterMapping[] mappings = parameterMap.getParameterMappings();
        for (int i = 0; i < mappings.length; i++) {
            if (mappings[i].isOutputAllowed()) {
                PROBE.setObject(object, mappings[i].getPropertyName(), values[i]);
            }
        }
        return object;
    }
}
}

```

按照 ParameterMap 的名称和数量, 直接获得复杂 Object 中各个属性对应数组对象的 Value, 进行循环赋值。

7. DomDataExchange 类的实现

DomDataExchange 类是处理 Dom 对象。

(1) `getData(StatementScope statementScope, ParameterMap parameterMap, Object parameterObject)`方法实现

该 `getData` 方法是根据 `Document` 对象和 `ParameterMap` 来进行取值处理。

```
public Object[] getData(StatementScope statementScope, ParameterMap parameterMap,
Object parameterObject) {
    Probe probe = ProbeFactory.getProbe(parameterObject);

    ParameterMapping[] mappings = parameterMap.getParameterMappings();
    Object[] values = new Object[mappings.length];

    for (int i = 0; i < mappings.length; i++) {
        values[i] = probe.getObject(parameterObject, mappings[i].getPropertyName());
    }

    return values;
}
```

首先从 `ProbeFactory` 工厂中创建出 `DomProbe` 实例化对象。然后按照 `ParameterMapping` 对象的名称和数量, 通过 `DomProbe` 的 `getObject` 方法, 得到其对应名称的 `Element` 的值, 并形成一个对象数组。

(2) `setData(StatementScope statementScope, ResultMap resultMap, Object resultObject, Object[] values)`方法实现

该 `setData` 方法是根据 `Document` 对象和 `ResultMap` 对象来进行赋值处理。

```
public Object setData(StatementScope statementScope, ResultMap resultMap, Object
resultObject, Object[] values) {

    String name = ((ResultMap) resultMap).getXmlName();
    if (name == null) {name = "result";}

    if (resultObject == null) {
        try {
            Document doc = DocumentBuilderFactory.newInstance().newDocument
Builder().newDocument();
            doc.appendChild(doc.createElement(name));
            resultObject = doc;
        } catch (ParserConfigurationException e) {
            throw new SqlMapException("Error creating new Document for DOM result.
Cause: " + e, e);
        }
    }

    Probe probe = ProbeFactory.getProbe(resultObject);

    ResultMapping[] mappings = resultMap.getResultMappings();

    for (int i = 0; i < mappings.length; i++) {
```

```

        if (values[i] != null) {
            probe.setObject(resultObject, mappings[i].getPropertyName(), values[i]);
        }
    }

    return resultObject;
}

```

首先创建 Document 对象, 然后从 ProbeFactory 工厂中创建出 DomProbe 实例化对象。再然后按照 ResultMapping 对象的名称和数量, 通过 DomProbe 对象的 setObject 方法, 按照 Document 对象中对应名称的 Element 把对象数组一个个地进行赋值, 完成对整个 Document 对象的填充。

(3) setData(StatementScope statementScope, ParameterMap parameterMap, Object parameterObject, Object[] values)方法实现

该 setData 方法是根据 Document 对象和 ParameterMap 对象来进行赋值处理。

```

public Object setData(StatementScope statementScope, ParameterMap parameterMap,
Object parameterObject, Object[] values) {
    Probe probe = ProbeFactory.getProbe(parameterObject);
    ParameterMapping[] mappings = parameterMap.getParameterMappings();
    for (int i = 0; i < mappings.length; i++) {
        if (values[i] != null) {
            if (mappings[i].isOutputAllowed()) {
                probe.setObject(parameterObject, mappings[i].getPropertyName(),
values[i]);
            }
        }
    }
    return parameterObject;
}

```

首先从 ProbeFactory 工厂中创建出 DomProbe 实例化对象。然后按照 ParameterMapping 的名称和数量, 通过 DomProbe 的 setObject 方法, 按照 Document 对象中对应名称的 Element 把对象数组一个个地进行赋值, 完成对整个 Document 对象的填充。

8. MapDataExchange 类的实现

(1) getData(StatementScope statementScope, ParameterMap parameterMap, Object parameterObject)方法实现

该 getData 方法是针对 Map 的 ResultMap 对象来进行取值操作。

```

public Object[] getData(StatementScope statementScope, ParameterMap parameterMap,
Object parameterObject) {
    if (!(parameterObject instanceof Map)) {
        throw new RuntimeException("Error. Object passed into MapDataExchange was
not an instance of Map.");
    }
}

```

```

Object[] data = new Object[parameterMap.getParameterMappings().length];
Map map = (Map) parameterObject;
ParameterMapping[] mappings = parameterMap.getParameterMappings();
for (int i = 0; i < mappings.length; i++) {
    data[i] = map.get(mappings[i].getPropertyName());
}
return data;
}

```

处理方式比较简单, 按照 `ParameterMapping` 的名称和数量, 直接获得对应于 `Map` 中 key 的 Value, 进行循环取值, 形成一个对象数组。

(2) `setData(StatementScope statementScope, ResultMap resultMap, Object resultObject, Object[] values)` 方法实现

该 `setData` 方法是针对 `Map` 的 `ResultMap` 对象来进行赋值操作。

```

public Object setData(StatementScope statementScope, ResultMap resultMap, Object
resultObject, Object[] values) {
    if (!(resultObject == null || resultObject instanceof Map)) {
        throw new RuntimeException("Error. Object passed into MapDataExchange was
not an instance of Map.");
    }

    Map map = (Map) resultObject;
    if (map == null) {map = new HashMap();}

    ResultMapping[] mappings = resultMap.getResultMappings();
    for (int i = 0; i < mappings.length; i++) {
        map.put(mappings[i].getPropertyName(), values[i]);
    }

    return map;
}

```

处理方式比较简单, 按照 `ResultMap` 的名称和数量, 直接获得对应于 `Map` 中 key 的 Value, 进行循环赋值。

(3) `setData(StatementScope statementScope, ParameterMap parameterMap, Object parameterObject, Object[] values)` 方法实现

该 `setData` 方法是针对 `Map` 的 `ParameterMap` 对象来进行赋值操作。

```

public Object setData(StatementScope statementScope, ParameterMap parameterMap,
Object parameterObject, Object[] values) {
    if (!(parameterObject == null || parameterObject instanceof Map)) {
        throw new RuntimeException("Error. Object passed into MapDataExchange was
not an instance of Map.");
    }

    Map map = (Map) parameterObject;
    if (map == null) {map = new HashMap();}
}

```



```

ParameterMapping[] mappings = parameterMap.getParameterMappings();
for (int i = 0; i < mappings.length; i++) {
    if (mappings[i].isOutputAllowed()) {
        map.put(mappings[i].getPropertyName(), values[i]);
    }
}

return map;
}

```

处理方式比较简单, 按照 `ParameterMapping` 的名称和数量, 直接获得对应于 `Map` 中 key 的 Value, 进行循环赋值。

9. `JavaBeanDataExchange` 类的实现

`JavaBeanDataExchange` 类是处理 `JavaBean` 对象。该类在实例化的时候, 要进行初始化赋值, 代码如下:

```

public void initialize(Map properties) {
    Object map = properties.get("map");
    if (map instanceof ParameterMap) {
        ParameterMap parameterMap = (ParameterMap) map;
        if (parameterMap != null) {
            ParameterMapping[] parameterMappings = parameterMap.get
ParameterMappings();
            String[] parameterPropNames = new String[parameterMappings.length];
            for (int i = 0; i < parameterPropNames.length; i++) {
                parameterPropNames[i] = parameterMappings[i].getPropertyName();
            }
            parameterPlan = AccessPlanFactory.getAccessPlan(parameterMap.getParameter
Class(), parameterPropNames);

            // OUTPUT PARAMS
            List outParamList = new ArrayList();
            for (int i = 0; i < parameterPropNames.length; i++) {
                if (parameterMappings[i].isOutputAllowed()) {
                    outParamList.add(parameterMappings[i].getPropertyName());
                }
            }
            String[] outParams = (String[]) outParamList.toArray(new String[out
ParamList.size()]);
            outParamPlan = AccessPlanFactory.getAccessPlan(parameterMap.getParameter
Class(), outParams);
        }
    } else if (map instanceof ResultMap) {
        ResultMap resultMap = (ResultMap) map;
        if (resultMap != null) {
            ResultMapping[] resultMappings = resultMap.getResultMappings();
            String[] resultPropNames = new String[resultMappings.length];
            for (int i = 0; i < resultPropNames.length; i++) {

```

```

        resultPropNames[i] = resultMappings[i].getPropertyName();
    }
    resultPlan = AccessPlanFactory.getAccessPlan(resultMap.getResultClass(), resultPropNames);
}
}
}

```

根据传入参数的不同,分别从 AccessPlanFactory 工厂类中创建出实现 AccessPlan 接口的 parameterPlan 变量对象和 outParamPlan 变量对象或 resultPlan 变量对象。

(1) getData(StatementScope statementScope, ParameterMap parameterMap, Object parameterObject)方法实现

该 getData 方法是根据 JavaBean 对象和 ParameterMap 对象来进行处理的。

```

public Object[] getData(StatementScope statementScope, ParameterMap parameterMap,
Object parameterObject) {
    if (parameterPlan != null) {
        return parameterPlan.getProperties(parameterObject);
    } else {
        return NO_DATA;
    }
}

```

调用初始化生成的 parameterPlan 变量对象的 getProperties 方法来完成,关于这个方法的实现,在后面的 Accessplan 组件的设计模式中会进行详细说明。

(2) setData(StatementScope statementScope, ResultMap resultMap, Object resultObject, Object[] values) 方法实现

该 setData 方法是根据 JavaBean 对象和 ResultMap 对象来进行处理的。

```

public Object setData(StatementScope statementScope, ResultMap resultMap, Object
resultObject, Object[] values) {
    if (resultPlan != null) {
        Object object = resultObject;
        ErrorContext errorContext = statementScope.getErrorContext();

        if (object == null) {
            errorContext.setMoreInfo("The error occurred while instantiating the
result object");
            try {
                object = ResultObjectFactoryUtil.createObjectThroughFactory
(resultMap.getResultClass());
            } catch (Exception e) {
                throw new RuntimeException("JavaBeansDataExchange could not
instantiate result class. Cause: " + e, e);
            }
        }
        errorContext.setMoreInfo("The error happened while setting a property on
the result object.");
        resultPlan.setProperties(object, values);
    }
}

```



```

        return object;
    } else {
        return null;
    }
}

```

调用初始化生成的 resultPlan 变量对象的 setProperties 方法来完成, 关于这个方法的实现, 在后面的 Accessplan 组件的设计模式中会详细说明。

(3) setData(StatementScope statementScope, ParameterMap parameterMap, Object parameterObject, Object[] values)方法实现

该 setData 方法是根据 JavaBean 对象和 ParameterMap 对象来进行的处理的。

```

public Object setData(StatementScope statementScope, ParameterMap parameterMap,
Object parameterObject, Object[] values) {
    if (outParamPlan != null) {
        Object object = parameterObject;
        if (object == null) {
            try {
                object = ResultObjectFactoryUtil.createObjectThroughFactory(
parameterMap.getParameterClass());
            } catch (Exception e) {
                throw new RuntimeException("JavaBeansDataExchange could not
instantiate parameter class. Cause: " + e, e);
            }
        }
        values = getOutputParamValues(parameterMap.getParameterMappings(), values);
        outParamPlan.setProperties(object, values);
        return object;
    } else {
        return null;
    }
}

```

调用初始化生成的 outParamPlan 变量对象的 setProperties 方法来完成, 关于这个方法的实现, 在后面的 Accessplan 组件的设计模式中会详细说明。

9.5.2 Accessplan 组件的设计模式

Accessplan 组件主要用于 beans 属性类型的转化, 其内容主要在 com.ibatis.sqlmap.engine.accessplan 包内。在 DataExchange 接口支持的 DOM 对象、List 对象、Map 对象、beans 对象中, Accessplan 组件只是为实现 beans 对象的 JavaBeanDataExchange 对象服务的。Accessplan 组件支持三种属性的转化, 第一种是 Javabean 中的 get 和 set 方法的属性转化。第二种是 Accessplan 组件的设计模式与 DataExchange 组件基本一致, 都是采用工厂方法设计模式, 类结构如图 9-16 所示。

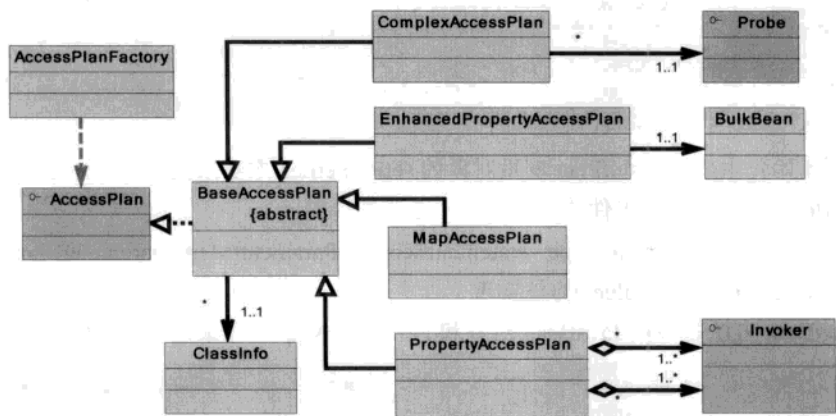


图 9-16 Accessplan 组件类结构图

BaseAccessPlan 类、ComplexAccessPlan 类、PropertyAccessPlan 类、EnhancedPropertyAccessPlan 类、MapAccessPlan 类继承 BaseAccessPlan 抽象类并实现 AccessPlan 接口。同时 AccessPlanFactory 类依赖 BaseAccessPlan 抽象类。

Accessplan 组件所涉及的相关类和接口如表 9-24 所示。

表 9-24 Accessplan 组件主要的类和接口列表

接口或类	功能或用途描述	备 注
com.ibatis.sqlmap.engine.accessplan.AccessPlanFactory	获得一个访问 Bean 的 Accessplan 实例化对象的工厂类	类
com.ibatis.sqlmap.engine.accessplan.AccessPlan	访问 Bean、Map 等对象的 Accessplan 接口	接口
com.ibatis.sqlmap.engine.accessplan.BaseAccessPlan	访问 Bean、Map 等对象的基本 Accessplan 实现类	抽象类
com.ibatis.sqlmap.engine.accessplan.ComplexAccessPlan	访问复杂对象的基本 Accessplan 实现类	类
com.ibatis.sqlmap.engine.accessplan.PropertyAccessPlan	访问复杂对象属性的基本 Accessplan 实现类	类
com.ibatis.sqlmap.engine.accessplan.EnhancedPropertyAccessPlan	访问复杂对象属性的基本 Accessplan 加强实现类（用 CG 库）	类
com.ibatis.sqlmap.engine.accessplan.MapAccessPlan	访问 Maps 对象的基本 Accessplan 实现类	类

1. AccessPlanFactory 类介绍

对 Accessplan 组件的应用都是通过调用 AccessPlanFactory 来创建的，当 JavaBeanDataExchange 对象要实现 Accessplan 接口的具体实现类时，要依据 JavaBeanDataExchange 对象所对应的 ParameterMap 对象本身的 Class 私有属性内容。不同的 Class 创建不同的 Accessplan 实现类。程序代码在 AccessPlanFactory 的 getAccessPlan 方

法,代码如下:

```
public static AccessPlan getAccessPlan(Class clazz, String[] propertyNames) {
    AccessPlan plan;
    boolean complex = false;

    if (clazz == null || propertyNames == null) {
        complex = true;
    } else {
        for (int i = 0; i < propertyNames.length; i++) {
            if (propertyNames[i].indexOf('.') > -1 || propertyNames[i].
indexOf('.') > -1) {
                complex = true;
                break;
            }
        }

        if (complex) {
            plan = new ComplexAccessPlan(clazz, propertyNames);
        } else if (Map.class.isAssignableFrom(clazz)) {
            plan = new MapAccessPlan(clazz, propertyNames);
        } else {
            if (bytecodeEnhancementEnabled) {
                try {
                    plan = new EnhancedPropertyAccessPlan(clazz,
propertyNames);
                } catch (Throwable t) {
                    try {
                        plan = new PropertyAccessPlan(clazz, property
Names);
                    } catch (Throwable t2) {
                        plan = new ComplexAccessPlan(clazz, property
Names);
                    }
                }
            } else {
                try {
                    plan = new PropertyAccessPlan(clazz, property Names);
                } catch (Throwable t) {
                    plan = new ComplexAccessPlan(clazz, propertyNames);
                }
            }
        }
    }
    return plan;
}
```

根据 Class 的情况,创建出不同的 AccessPlan 实例化对象。

2. Accessplan 的接口方法

Accessplan 组件还有一个重要的功能就是实现 Accessplan 接口。AccessPlan 接口是用

于访问任何数据类型的一致性资源。Accessplan 接口的方法如表 9-25 所示。

表 9-25 Accessplan 接口的方法列表

方法名称	参 数	返 回 值	功能和用途	备 注
setPropertyies	Object Object[]	无	给 bean 的所有属性赋值	
getPropertyies	Object	Object[]	获取 bean 的所有属性	

后面举两个例子来进行详细说明，分别是 DomDataExchange 类和 MapDataExchange 类的实现。关于 JavaBeanDataExchange 类的实现，放到 Accessplan 组件中进行说明。由于 DomDataExchange 类和 MapDataExchange 类都继承 BaseDataExchange 抽象类，所以还是先从 BaseDataExchange 抽象类进行说明。

3. ComplexAccessPlan 类的方法实现

通过调用 Probe 组件来实现。

(1) setPropertyies 方法的实现

ComplexAccessPlan 类 setPropertyies 方法的程序代码：

```
private static final Probe PROBE = ProbeFactory.getProbe();

public void setPropertyies(Object object, Object[] values) {
    for (int i = 0; i < propertyNames.length; i++) {
        PROBE.setObject(object, propertyNames[i], values[i]);
    }
}
```

通过 ProbeFactory 创建出一个 GenericProbe 实例化对象，然后根据传入的对象数组，采用循环过程用 GenericProbe 对象的 setObject 方法赋值到 Bean 对象中。GenericProbe 对象的 setObject 方法见本书第 12.2 节的“Probe 接口及其实现”内容。

(2) getPropertyies 方法的实现

ComplexAccessPlan 类 getPropertyies 方法的程序代码：

```
private static final Probe PROBE = ProbeFactory.getProbe();

public Object[] getPropertyies(Object object) {
    Object[] values = new Object[propertyNames.length];
    for (int i = 0; i < propertyNames.length; i++) {
        values[i] = PROBE.getObject(object, propertyNames[i]);
    }
    return values;
}
```

采用循环过程用 GenericProbe 对象的 getPropertyies 方法从 Bean 对象中获取各个属性值并赋值给对象数组。GenericProbe 对象的 setObject 方法见本书 12.2 节中的“Probe 接口及其实现”内容。

4. EnhancedPropertyAccessPlan 类的方法实现

主要是利用 CGLIB 库来实现 JavaBean 的属性方法。EnhancedPropertyAccessPlan（用 CGLib 处理 beans）。关于 CGLIB 包的介绍见第 1 章。

(1) setProperties 方法的实现

EnhancedPropertyAccessPlan 类 setProperties 方法的程序代码：

```
private BulkBean bulkBean;

public void setProperties(Object object, Object[] values) {
    bulkBean.setPropertyValues(object, values);
}
```

EnhancedPropertyAccessPlan 在构造的时候，对 BulkBean 对象进行了实例化。

```
EnhancedPropertyAccessPlan(Class clazz, String[] propertyNames) {
    super(clazz, propertyNames);
    bulkBean = BulkBean.create(clazz, getGetterNames(propertyNames), getSetterNames(propertyNames), getTypes(propertyNames));
}
```

所以，当调用 EnhancedPropertyAccessPlan 类的 setProperties 方法时，转到 BulkBean 对象的 setProperties 方法。

(2) getProperties 方法的实现

EnhancedPropertyAccessPlan 类 getProperties 方法的程序代码。

```
public Object[] getProperties(Object object) {
    return bulkBean.getPropertyValues(object);
}
```

当调用 EnhancedPropertyAccessPlan 类的 getProperties 方法时，转到 BulkBean 对象的 getProperties 方法。

5. MapAccessPlan 类的方法实现

对 Map 对象进行赋值和取值操作。

(1) setProperties 方法的实现

MapAccessPlan 类 setProperties 方法的程序代码：

```
public void setProperties(Object object, Object[] values) {
    Map map = (Map) object;
    for (int i = 0; i < propertyNames.length; i++) {
        map.put(propertyNames[i], values[i]);
    }
}
```

代码比较简单，通过循环直接对 Map 对象进行赋值。

(2) getProperties 方法的实现

MapAccessPlan 类 getProperties 方法的程序代码。

```
public Object[] getProperties(Object object) {
    Object[] values = new Object[propertyNames.length];
    Map map = (Map) object;
    for (int i = 0; i < propertyNames.length; i++) {
        values[i] = map.get(propertyNames[i]);
    }
    return values;
}
```

循环对 Map 对象进行取值，然后转化为对象数组输出。

6. PropertyAccessPlan 类的方法实现

通过反向反射机制来实现对一个对象的赋值和取值。该类在使用过程中与 BaseAccessPlan 结合起来。

(1) setProperties 方法的实现

PropertyAccessPlan 类 setProperties 方法的程序代码：

```
public void setProperties(Object object, Object[] values) {
    int i = 0;
    try {
        Object[] arg = new Object[1];
        for (i = 0; i < propertyNames.length; i++) {
            arg[0] = values[i];
            try {
                setters[i].invoke(object, arg);
            } catch (Throwable t) {
                throw ClassInfo.unwrapThrowable(t);
            }
        }
    } catch (Throwable t) {
        throw new RuntimeException("..." + t, t);
    }
}
```

在程序中调用 setters[i].invoke(Object, arg)，最后还是调用 ClassInfo 类的反射方法来实现。

(2) getProperties 方法的实现

PropertyAccessPlan 类的 getProperties 方法的程序代码。

```
public Object[] getProperties(Object object) {
    int i = 0;
    Object[] values = new Object[propertyNames.length];
    try {
        for (i = 0; i < propertyNames.length; i++) {
            try {
```

```

        values[i] = getters[i].invoke(object, NO_ARGUMENTS);
    } catch (Throwable t) {
        throw ClassInfo.unwrapThrowable(t);
    }
}
} catch (Throwable t) {
    throw new RuntimeException("..." + t, t);
}
return values;
}

```

当调用 `PropertyAccessPlan` 类的 `getProperties` 方法时，转到 `getters[i].invoke` 方法。最后还是通过 `ClassInfo` 类的反射调用来实现。

9.5.3 DataExchange 和 Accessplan 在系统中如何实现

在这里只是针对 beans 进行讲解，其他的 Map、Object 和 List，由于都比较简单，读者可以自己去看看。对于 `JavaBean` 的讲解也分为两种情况，一种是 `JavaBean` 作为参数进行传入的情况，还有一种是当执行查询后，用 `JavaBean` 作为输出对象的情况。

1. `JavaBean` 作为 `ParameterMap` 进行输入

当外部配置信息为如下内容时，

```

<parameterMap id="insert-product-param" class="com.domain.Product">
    <parameter property="id"/>
    <parameter property="description"/>
</parameterMap>
<statement id="insertProduct" parameterMap="insert-product-param">
    insert into PRODUCT (PRD_ID, PRD_DESCRIPTION) values (?,?);
</statement>

```

在 `SqlMapParser` 类的 `addParameterMapNodelets` 方法中解析 `ParameterMap` 节点。调用的程序代码如下：

```

state.getParamConfig().addParameterMapping(propertyName, javaClass, jdbcType,
nullValue, mode, type, numericScale, typeHandlerImpl, resultMap);

```

在形成 `ParameterMap` 对象后，处理 `ParameterMap` 的 `ParameterMapping` 数组时，会调用 `setParameterMappingList` 方法，代码如下。

```

public void setParameterMappingList(List parameterMappingList) {
    this.parameterMappings = (ParameterMapping[])
        parameterMappingList.toArray(new ParameterMapping[parameterMapping
List.size()]);
    parameterMappingIndex.clear();
    for (int i = 0; i < parameterMappings.length; i++) {
        parameterMappingIndex.put(parameterMappings[i].getPropertyName(), new

```



```

Integer(i));
    }
    Map props = new HashMap();
    props.put("map", this);

    dataExchange = delegate.getDataExchangeFactory().getDataExchangeForClass
(parameterClass);
    dataExchange.initialize(props);
}

```

按照上面的例子，由于 class 是 com.domain.Product，故它实际上是一个 JavaBean。所以，dataExchange 变量实际上是一个 JavaBeanDataExchange 的实例化对象。后面执行的代码为 initialize 方法。代码如下。

```

public void initialize(Map properties) {
    Object map = properties.get("map");
    if (map instanceof ParameterMap) {
        ParameterMap parameterMap = (ParameterMap) map;
        if (parameterMap != null) {
            ParameterMapping[] parameterMappings = parameterMap.get
ParameterMappings();
            String[] parameterPropNames = new String[parameterMappings.length];
            for (int i = 0; i < parameterPropNames.length; i++) {
                parameterPropNames[i] = parameterMappings[i].getPropertyName();
            }

            parameterPlan = AccessPlanFactory.getAccessPlan(parameterMap.get
ParameterClass(), parameterPropNames);

            // OUTPUT PARAMS
            List outParamList = new ArrayList();
            for (int i = 0; i < parameterPropNames.length; i++) {
                if (parameterMappings[i].isOutputAllowed()) {
                    outParamList.add(parameterMappings[i].getPropertyName());
                }
            }
            String[] outParams = (String[]) outParamList.toArray(new String[out
ParamList.size()]);
            outParamPlan = AccessPlanFactory.getAccessPlan(parameterMap.get
ParameterClass(), outParams);
        }

    } else if (map instanceof ResultMap) {
        .....
    }
}

```

由于 parameterMap.getParameterClass()返回的还是 com.domain.Product 类。在上面的 parameterPlan 变量实际上是 PropertyAccessPlan 的实例化对象。当 parameterPlan 创建并被实例化时，其代码如下：

```

PropertyAccessPlan(Class clazz, String[] propertyNames) {
    super(clazz, propertyNames);
    setters = getSetters(propertyNames);
    getters = getGetters(propertyNames);
}

```

其中 `getGetters` 方法实现的内容如下。

```

protected Invoker[] getGetters(String[] propertyNames) {
    Invoker[] methods = new Invoker[propertyNames.length];
    for (int i = 0; i < propertyNames.length; i++) {
        methods[i] = info.getGetInvoker(propertyNames[i]);
    }
    return methods;
}

```

返回一个 `get` 的方法数组。这样初始化就结束了。

当外部开始调用具体实现的数值时，在系统中会采用如下方法。

```

Object[] parameters = parameterMap.getParameterObjectValues(statementScope,
parameterObject);

```

`ParameterMap` 对象的 `getParameterObjectValues` 方法如下。

```

public Object[] getParameterObjectValues(StatementScope statementScope, Object
parameterObject) {
    return dataExchange.getData(statementScope, this, parameterObject);
}

```

`JavaBeanDataExchange` 对象的 `getData` 方法如下。

```

public Object[] getData(StatementScope statementScope, ParameterMap parameterMap,
Object parameterObject) {
    if (parameterPlan != null) {
        return parameterPlan.getProperties(parameterObject);
    } else { return NO_DATA; }
}

```

`PropertyAccessPlan` 对象的 `getProperties` 方法如下。

```

public Object[] getProperties(Object object) {
    int i = 0;
    Object[] values = new Object[propertyNames.length];
    try {
        for (i = 0; i < propertyNames.length; i++) {
            try {
                values[i] = getters[i].invoke(object, NO_ARGUMENTS);
            } catch (Throwable t) { ..... }
        }
    } catch (Throwable t) {
        .....
    }
    return values;
}

```

通过反射调用，获取 JavaBean 的属性值。

2. JavaBean 作为 resultMap 进行输出结果

当外部配置信息为如下内容时。

```
<resultMap id="get-product-result" class="com.ibatis.example.Product">
  <result property="id" column="PRD_ID"/>
  <result property="description" column="PRD_DESCRIPTION"/>
</resultMap>
<statement id="getProduct" resultMap="get-product-result">
  select * from PRODUCT
</statement>
```

在 SqlMapParser 类的 addResultMapNodelets 方法中解析 resultMap 节点。外部程序代码如下。

```
state.getResultConfig().addResultMapping(propertyName, columnName, columnIndex,
javaClass, jdbcType, nullValue, notNullColumn, statementName, resultMapName, type
HandlerImpl);
```

在形成 resultMap 对象后，处理 resultMap 的 resultMaping 数组时，会调用 setResultMappingList 方法，代码如下。

```
public void setResultMappingList(List resultMapingList) {
    if (allowRemapping) {
        this.remappableResultMappings.set((ResultMapping[])
resultMappingList.toArray(new ResultMapping[resultMappingList.size()]));
    } else {
        this.resultMappings = (ResultMapping[]) resultMappingList.toArray(new
ResultMapping[resultMappingList.size()]);
    }

    Map props = new HashMap();
    props.put("map", this);
    dataExchange = getDelegate().getDataExchangeFactory().getDataExchangeFor
Class(resultClass);
    dataExchange.initialize(props);
}
```

按照上面的例子，由于 class 是 com.ibatis.example.Product，它实际上是一个 JavaBean。所以，dataExchange 变量实际上是一个 JavaBeanDataExchange 的实例化对象。后面执行的代码为 initialize 方法，与上面的 ParameterMap 代码相同。

```
public void initialize(Map properties) {
    Object map = properties.get("map");
    if (map instanceof ParameterMap) {
        .....
    } else if (map instanceof ResultMap) {
        ResultMap resultMap = (ResultMap) map;
        if (resultMap != null) {
```

```

        ResultMapping[] resultMappings = resultMap.getResultMappings();
        String[] resultPropNames = new String[resultMappings.length];
        for (int i = 0; i < resultPropNames.length; i++) {
            resultPropNames[i] = resultMappings[i].getPropertyName();
        }
        resultPlan = AccessPlanFactory.getAccessPlan(resultMap.getResultClass(), resultPropNames);
    }
}

```

由于 resultMap.getResultClass()返回的还是 com.ibatis.example.Product 类, 故在上面的 resultPlan 变量实际上是 PropertyAccessPlan 的实例化对象。当 resultPlan 创建并被实例化时, 其代码如下:

```

PropertyAccessPlan(Class clazz, String[] propertyNames) {
    super(clazz, propertyNames);
    setters = getSetters(propertyNames);
    getters = getGetters(propertyNames);
}

```

其中 getSetters 方法实现的内容如下。

```

protected Invoker[] getSetters(String[] propertyNames) {
    Invoker[] methods = new Invoker[propertyNames.length];
    for (int i = 0; i < propertyNames.length; i++) {
        methods[i] = info.getSetInvoker(propertyNames[i]);
    }
    return methods;
}

```

返回一个 set 的方法数组。这样初始化就结束了。

当外部开始调用具体实现的数值时, 在系统中会采用如下方法。

```

resultObject = dataExchange.setData(statementScope, this, resultObject, values);

```

JavaBeanDataExchange 对象的 setData 方法如下。

```

public Object setData(StatementScope statementScope, ResultMap resultMap, Object resultObject, Object[] values) {
    if (resultPlan != null) {
        Object object = resultObject;
        ....
        if (object == null) {
            ....
            try {
                object = ResultObjectFactoryUtil.createObjectThroughFactory(resultMap.getResultClass());
            } catch (Exception e) {
                ....
            }
        }
    }
}

```

```

.....
    resultPlan.setProperties(object, values);
    return object;
} else {
    return null;
}
}

```

PropertyAccessPlan 对象的 setProperties 方法如下。

```

public void setProperties(Object object, Object[] values) {
    int i = 0;
    try {
        Object[] arg = new Object[1];
        for (i = 0; i < propertyNames.length; i++) {
            arg[0] = values[i];
            try {
                setters[i].invoke(object, arg);
            } catch (Throwable t) {
                throw ClassInfo.unwrapThrowable(t);
            }
        }
    } catch (Throwable t) {..... }
}

```

通过反射调用，给 JavaBean 的属性赋值。

9.6 读取源码的收获

通过我们对 SQL Map 中 Mapping 实现源码的理解和分析，应该有这么几个收获。

第一，在实现 resultMap 类中，通过动态代理来实现延迟加载 resultMap 实例化对象。这样也是为了提高系统性能的一种手段。

第二，用状态设计模式来实现 MappedStatement 类以及 InsertStatement 类、SelectStatement 类、DeleteStatement 类、UpdateStatement 类、ProcedureStatement 类的关系。这样的实现方式还是值得借鉴的。通过这种设计，至少可以实现抽象层和最终实现层的耦合，在具体类的实现上有更大的自由度。但是我觉得还是有一点代码繁琐的感觉。

第三，采用装饰模式来实现 MappedStatement 类和 CachingStatement 类的关系。这样在不影响 MappedStatement 对象的情况下，以动态、透明的方式给单个对象添加缓存职责。这种方式比单独的继承要灵活得多，让代码更加紧凑，而且实现效率也比较高。

第四，对于数据库支持自动生成主键，iBATIS 实现了基于 MS SQL Server 和 Oracle 数据库的功能。在了解了这些源码后，为我们在编写数据库应用系统中有同样问题提供了一个可以借鉴的参考资料和代码。

第五，通过对 DataExchange 和 Accessplan 组件的学习，应该了解如何来处理松耦合的 JavaBean、Map 和 XML 与数据的交换。这里面要涉及对象名称、对象属性的获取和赋值，还有数据类型的转换。

第 10 章

SQL Map 缓存管理和实现

本章内容:

1. SQL Map 缓存结构和组成, 主要是采用简单工厂模式和动态加载来实现扩展性。
2. 系统如何使用缓存, 说明 CacheModel 类、CacheKey 类及其实现。
3. 缓存策略的程序实现。主要是缓存控制器的实现说明, 包括 FIFO 策略、LRU 策略、MEMORY 策略和 OSCACHE 策略。
4. 增加一个 SQL Map 的扩展缓存控制器, 即 FILO 的实现。

在软件开发过程中, 可能需要管理一些“稀有资源”, 如网络链接、内存、程序等。这些资源的使用在时间或者空间上都极大地影响着软件的使用效率。对于这些稀有资源可以采用共享模式, 将这些资源在时间或空间上尽量复用。

当前, 以数据库为中心的信息检索系统的应用日益普及, 而应用系统中的信息海量且检索频繁, 采用合理有效的技术手段来提高检索效率就显得非常必要。在一定的软硬件环境条件下, 从数据库或外部存储器频繁获取数据, 势必会降低整个系统的响应性能。因而, 合理有效地减少与数据库服务器的交互次数, 已经成为提高运行性能的重要手段之一。因此, 缓存技术便应运而生, 即客户端软件一次从数据库服务器或外设将符合要求的数据读入内存, 然后多次呈现到应用系统界面中与用户完成实时交互。缓存技术是“以空间换时间”设计理念的继承和发展, 是利用内存空间资源来提高检索速度的有效方法之一。

数据缓存是数据库数据或外存中的文件数据在内存中的临时存储印象, 是应用程序运行到一定阶段数据信息的内存副本。在实际应用环境中, 有的数据在整个运行阶段更新频率较低, 没有必要频繁地读取, 应用系统启动后一次性读入内存供使用即可; 有的数据更新相对较频繁, 查询时由于算法规则的复杂性导致响应性能不能满足实际需求, 因而需要将这类数据周期性地存入内存, 供后续操作或其他用户直接从内存中间接读取使用, 以减少直接读取次数、加快响应速度。内存资源存取速度虽快, 但容量非常有限且成本较高, 只有建立良好的缓存策略才能充分使用内存。科学合理的缓存策略是获得高效检索功能的重要保证, 提高软件性能方面应考虑数据在内存中的存储结构、缓存项的更新淘汰算法、

内存数据与主数据（数据源数据）的同步策略、缓存命中情况以及命中率计算评估等方面的内容。缓存技术的实质是把在一定时间范围内不发生改变的数据读入内存，以一种易于快速检索的形式（比如哈希表）进行存储；数据请求时，总是先查看内存中是否存在符合要求的数据副本，如果存在（称之为缓存命中），则直接从内存中取得，否则，从实际数据源中读取，并对相应数据进行缓存。如此循环往复，会大大地提高整个应用系统的响应性能。

缓存技术（cache）的工作原理是基于程序访问的局部性原理。根据局部性原理，可以在数据库（硬盘）和内存之间设置一个高速的容量相对较小的存储器。cache 外部对系统进行数据请求时，通常先访问内存 cache。在内存 cache 中找到有用的数据被称为命中，但是局部性原理不能保证所请求的数据百分之百地在 cache 中，这时称为未命中，需要访问内存。这里存在一个命中率的问题，即 CPU 在任一时刻从 cache 中可靠获取数据的几率。

基于缓存技术实现主要是完成两个核心任务：就是命中率策略和淘汰算法策略。什么是命中率？就是计算机的局部性访问原理，即采取什么技术手段实现缓存对象的唯一性。所谓淘汰算法策略，即缓存容量达到最大，采取什么样的策略和方法淘汰多余的缓存对象。

10.1 SQL Map 缓存结构和组成

SQL Map 的 CacheModel 使用插件方式来支持不同的缓存算法。SQL Map 的缓存主要在 com.ibatis.sqlmap.engine.cache 包内。涉及的类和接口有 ExecuteListener 接口、CacheController 接口、CacheModel 类、FifoCacheController 类、LruCacheController 类、MemoryCacheController 类、OSCacheController 类等，其缓存结构的类结构如图 10-1 所示。

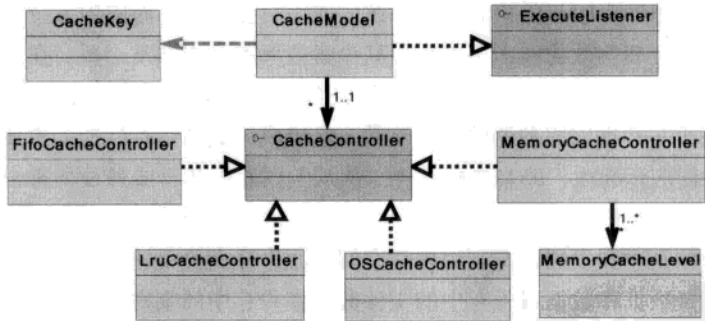


图 10-1 SQL Map 平台中缓存类结构图

下面对 SQLMap 平台中缓存类结构做一个简单的说明。CacheModel 类实现 ExecuteListener 接口并关联 CacheController 接口。同时，CacheModel 类还要依赖 FifoCacheController 类、LruCacheController 类、MemoryCacheController 类、OSCacheController 类实现 CacheController 接口。这是一种简单的工厂设计模式，iBATIS 缓存的接口和类组成说明如表 10-1 所示。

表 10-1 iBATIS 缓存的接口和类列表说明

接口或类	功能或用途描述	备 注
com.ibatis.sqlmap.engine.cache.CacheModel	Caches 的包装类	类
com.ibatis.sqlmap.engine.cache.CacheController	Cache 控制器接口	接口
com.ibatis.sqlmap.engine.cache.CacheKey	生成唯一 Hash 码的键值类	类
com.ibatis.sqlmap.engine.cache.fifo.FifoCacheController	先进先出的 Cache 控制器的实现类	类
com.ibatis.sqlmap.engine.cache.lru.LruCacheController	“近期最少使用”的 Cache 控制器的实现类	类
com.ibatis.sqlmap.engine.cache.memory.MemoryCacheController	MemoryCache 的 Cache 控制器的实现类	类
com.ibatis.Sqlmap.engine.cache.oscache.OSCacheController	OSCache2.0 缓存引擎在 iBATIS 中的实现	类

其中 CacheModel 类是核心类，这是一个 Cache 的包装类，但是 CacheModel 并不放置或保存缓存对象，它通过缓存控制器（CacheController）来保存缓存对象（Cache Object）。CacheKey 类是用于生成唯一 Hash 码的对象，也是基于 Hash 码生成的工具。这是为命中率做准备的。

10.2 系统如何使用缓存

当外部要使用缓存时，首先要创建缓存对象，同时要有个唯一标示这个缓存变量的 key。唯一标识这个缓存变量用 CacheKey 对象。这里主要从四个方面来进行说明，第一是缓存实现的过程，第二是缓存实现的核心类 CacheModel 类的介绍。第三是如何实现唯一的 CacheKey 对象，第四是缓存实现的策略。

10.2.1 缓存实现的序列图和说明

缓存实现主要涉及 CacheModel 类、CacheKey 类和实现 CacheController 接口的实现类。具体实现的过程如图 10-2 所示。

当一个对象要进行缓存处理时，一般要经过上述的 9 个步骤，分别解释实现步骤如下。

步骤 1：创建一个要进行缓存处理的 CacheObject 对象。

步骤 2：创建一个标识对象的 CacheKey 实例化对象。

步骤 3：传入参数（即由 CacheObject 的哈希等值形成的一个唯一标识值），调用 CacheKey 对象的 update 方法，产生一个唯一标识的 CacheKey 对象。该对象的内部属性 hashcode 是一个 int 值，唯一标识这个对象。

步骤 4：创建一个缓存对象 CacheModel。

步骤 5：为 CacheModel 对象设置缓存策略，iBATIS 平台支持四种缓存策略，分别为 FIFO（“先进先出”）策略、LRU（“近期最少使用”）策略、MEMORY 策略和 OSCACHE 策略。

步骤 6: 调用 CacheModel 对象的 putObject 方法, 传入 CacheKey 对象和 CacheObject 对象参数, 把要缓存的对象载入到缓存容器中。

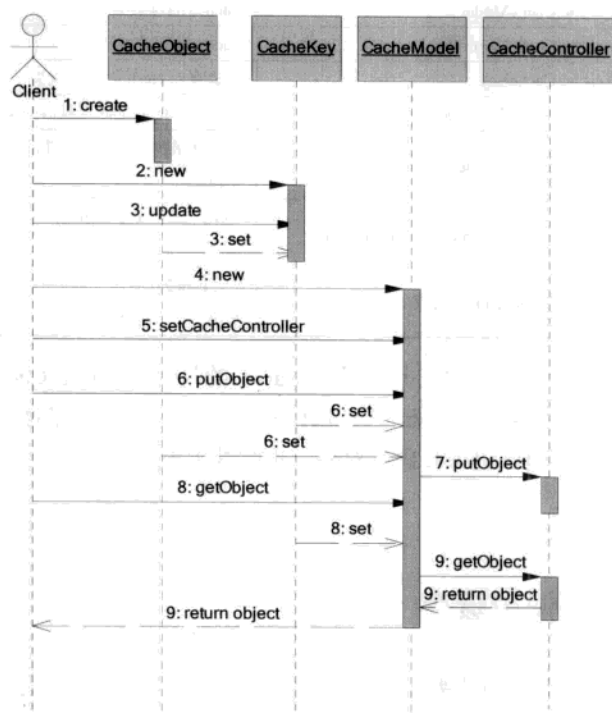


图 10-2 SQL Map 平台中缓存实现的序列图

步骤 7: CacheModel 对象对于上述的步骤的实现是调用其内部 CacheController 对象的 putObject 方法, 把缓存的对象保存在一个 Map 中。

步骤 8: 当要获得一个缓存对象时, 调用 CacheModel 对象的 getObject 方法, 传入 CacheKey 对象, 从 CacheModel 对象中获得缓存对象。

步骤 9: CacheModel 对象对于上述步骤的实现是调用其内部 CacheController 对象的 getObject 方法, 从 Map 中获取缓存对象并传递出来。

一般都是在 com.ibatis.sqlmap.engine.mapping.statement 包中的 CachingStatement 类实现缓存, 实现从缓存容器中获取缓存对象的缓存代码如下。

```

public Object executeQueryForObject(StatementScope statementScope, Transaction
trans, Object parameterObject, Object resultObject) throws SQLException {
    CacheKey cacheKey = getCacheKey(statementScope, parameterObject);
    cacheKey.update("executeQueryForObject");
    Object object = cacheModel.getObject(cacheKey);
    if (object == CacheModel.NULL_OBJECT){
        // This was cached, but null
        object = null;
    }
}

```

```

    }else if (object == null) {
        object = statement.executeQueryForObject(statementScope, trans, parameterObject,
resultObject);
        cacheModel.putObject(cacheKey, object);
    }
    return object;
}

```

实现从缓存容器中获取缓存 List 对象的代码如下。

```

public List executeQueryForList(StatementScope statementScope, Transaction trans,
Object parameterObject, int skipResults, int maxResults)
    throws SQLException {
    CacheKey cacheKey = getCacheKey(statementScope, parameterObject);
    cacheKey.update("executeQueryForList");
    cacheKey.update(skipResults);
    cacheKey.update(maxResults);
    Object listAsObject = cacheModel.getObject(cacheKey);
    List list;
    if(listAsObject == CacheModel.NULL_OBJECT){
        // The cached object was null
        list = null;
    }else if (listAsObject == null) {
        list = statement.executeQueryForList(statementScope, trans, parameterObject,
skipResults, maxResults);
        cacheModel.putObject(cacheKey, list);
    }else{
        list = (List) listAsObject;
    }
    return list;
}

```

在这些步骤中，其中的两个步骤比较重要，一个是由缓存对象生成唯一标识这个缓存对象的 CacheKey 码值，还有一个就是保存缓存对象和获得缓存对象。

10.2.2 CacheModel 类缓存的实现

CacheModel 类是 SQL Map 实现缓存的中心调度类。其初始化信息从 SQL Map 映射文件 CacheModel 节点中获取，其转化信息如表 6-2 所示。

1. CacheModel 类属性含义

这里列出 CacheModel 类的属性信息、数据类型及其用途，如表 10-2 所示。

表 10-2 CacheModel 类的属性列表

属性名称	类 型	功能和用途	备 注
id	String	缓存对象的唯一代码	
controller	CacheController	采用何种缓存控制器实现方式	

续表

属性名称	类 型	功能和用途	备 注
readOnly	Boolean	缓存对象是否只读	
serialize	boolean	是否对缓存对象采用序列化操作	
lastFlush	long	最后一次刷新时间	
flushInterval	long	刷新缓存的时间间隔，以微秒为单位	
flushIntervalSeconds	long	刷新缓存的间隔时间，以秒为单位	
flushTriggerStatements	Set	要进行缓存操作的方法，是一个 Set 表	

CacheModel 对象的初始化过程基本上就会对这些属性赋初值。

2. 载入和获得缓存对象

载入和获得缓存对象主要是在 CacheModel 对象实现。载入缓存对象的过程代码如下。

```
public void putObject(CacheKey key, Object value) {
    if (null == value) value = NULL_OBJECT;
    synchronized ( this ) {
        if (serialize && !readOnly && value != NULL_OBJECT) {
            try {
                ByteArrayOutputStream bos = new ByteArrayOutputStream();
                ObjectOutputStream oos = new ObjectOutputStream(bos);
                oos.writeObject(value);
                oos.flush();
                oos.close();
                value = bos.toByteArray();
            } catch (IOException e) {
                throw new RuntimeException("Error caching serializable object. Cause: "
+ e, e);
            }
        }
        controller.putObject(this, key, value);
        if ( log.isDebugEnabled() ) {
            log("stored object", true, value);
        }
    }
}
```

在载入缓存对象的时候，进行了序列化处理，接着对缓存对象进行保存。

3. 获得缓存对象

获取缓存对象的过程代码如下。

```
public Object getObject(CacheKey key) {
    Object value = null;
    synchronized (this) {
        if (flushInterval != NO_FLUSH_INTERVAL
        && System.currentTimeMillis() - lastFlush > flushInterval) {
            flush();
        }
    }
}
```

```

    }

    value = controller.getObject(this, key);
    if (serialize && !readOnly &&
        (value != NULL_OBJECT && value != null)) {
        try {
            ByteArrayInputStream bis = new ByteArrayInputStream((byte[]) value);
            ObjectInputStream ois = new ObjectInputStream(bis);
            value = ois.readObject();
            ois.close();
        } catch (Exception e) {
            throw new RuntimeException("Error caching serializable object. " +
                                     " Cause: " + e, e);
        }
    }
    requests++;
    if (value != null) {
        hits++;
    }
    if (log.isDebugEnabled()) {
        if (value != null) {
            log("retrieved object", true, value);
        }
        else {
            log("cache miss", false, null);
        }
    }
    return value;
}

```

载出缓存对象的时候，进行了反序列化处理，获得值对象。

10.2.3 唯一性 CacheKey 对象的产生

针对一个要缓存对象的 CacheKey 对象的形成很简单，调用 CacheKey 对象的 update 方法，并把缓存对象自身的一些属性值传入即可。

```

public CacheKey update(Object object) {
    int baseHashCode = object.hashCode();
    count++;
    checksum += baseHashCode;
    baseHashCode *= count;
    //典型的哈希算法
    hashCode = multiplier * hashCode + baseHashCode;

    paramList.add(object);

    return this;
}

```

由于 Java 中 int 类型的范围是:-2147483648 到 2147483648 之间，所以不用考虑 hashCode 是否会有重复。而且还重写了 equals 方法。我们可以看看 equals 方法的内容，基本上还有重码的可能性非常小。

```
public boolean equals(Object object) {
    if (this == object) return true;
    if (!(object instanceof CacheKey)) return false;

    final CacheKey cacheKey = (CacheKey) object;
    //判断哈希码是否相同
    if (hashCode != cacheKey.hashCode) return false;
    //判断哈希码是否相同，每次 update 都重新赋值
    if (checksum != cacheKey.checksum) return false;
    //判断计数是否相同，每次 update 都进行计数
    if (count != cacheKey.count) return false;
    //判断 paramList 是否相同，每次 update 都进行修改
    for (int i=0; i < paramList.size(); i++) {
        Object thisParam = paramList.get(i);
        Object thatParam = cacheKey.paramList.get(i);
        if(thisParam == null) {
            if (thatParam != null) return false;
        } else {
            if (!thisParam.equals(thatParam)) return false;
        }
    }
    return true;
}
```

10.3 缓存策略的程序实现

CacheModel 使用插件方式来支持不同的缓存算法。它的实现在 cacheModel 中用 type 属性来指定。指定的实现类必须实现 CacheController 接口。CacheModel 实现的其他配置参数通过 CacheModel 的 property 元素来设置。iBATIS 支持四种缓存策略。所谓的缓存策略，主要是淘汰算法，即采用什么方式删除在缓存容器中多余的缓存对象。这四种策略分别是 FIFO、LRU、MEMORY、OSCACHE。对于每种缓存策略的实现，都要实现五个方法，即 setProponties putObject、getObject、removeObject、flush。其方法和用途如表 10-3 所示。

表 10-3 CacheController 接口的方法列表

序 号	方法名称	作用和用途	备 注
1	setProperties	设置缓存控制器初始化变量	
2	putObject	放置一个缓存对象到缓存容器池	
3	getObject	从缓存容器池获得一个缓存对象	
4	removeObject	从缓存容器池获得一个缓存对象	
5	flush	清空整个缓存容器池	

对这五个方法的作用、用途和参数说明如下：

```
public void setProperties(Properties props);
```

该方法设置缓存控制器初始化变量。其中参数表示一个 `Properties` 对象，包含要配置的信息集合。

```
public void putObject(CacheModel cacheModel, Object key, Object object);
```

该方法加入缓存对象到缓存控制器中。其中第一个参数表示包含缓存控制器的缓存包装类容器，第二个参数表示要加入缓存对象的内码，第三个参数为要加入的缓存对象。

```
public Object getObject(CacheModel cacheModel, Object key);
```

该方法根据缓存内码从缓存控制器中获取缓存对象。其中第一个参数表示包含缓存控制器的缓存包装类容器，第二个参数表示要获取缓存对象的内码。

```
public Object removeObject(CacheModel cacheModel, Object key);
```

该方法从缓存控制器中删除缓存对象。其中第一个参数表示包含缓存控制器的缓存包装类容器，第二个参数表示要删除缓存对象的内码。

```
public void flush(CacheModel cacheModel);
```

该方法清空缓存控制器内全部的缓存对象。其中参数表示包含缓存控制器的缓存包装类容器。

下面分别介绍 FIFO、LRU、MEMORY、OSCACHE 的缓存实现。

10.3.1 FIFO 缓存实现

FIFO Cache 实现用“先进先出”原则来确定如何从 Cache 中清除对象。当 Cache 溢出时，最先进入 Cache 的对象将从 Cache 中清除，如同数据结构中的队列结构。其实现类是 `com.ibatis.sqlmap.engine.cache.fifo` 的 `FifoCacheController` 类。对于短时间内持续引用特定的查询而后很可能不再使用的情况，FIFO Cache 是很好的选择。

1. FIFO Cache 的配置

在映射文件中，FIFO Cache 的配置如下所示。

```
<cacheModel id="product-cache" type="FIFO">
  <flushInterval hours="24"/>
  <flushOnExecute statement="insertProduct"/>
  <flushOnExecute statement="updateProduct"/>
  <flushOnExecute statement="deleteProduct"/>
  <property name="size" value="1000" />
</cacheModel>
```


FIFO Cache 实现只认可一个 `property` 元素。其名为“Cache-size”的属性值必须是整数，代表同时保存在 Cache 中对象的最大数目。值得注意的是，这里指的对象可以是任意的，从单一的 `String` 对象到 Java Bean 的 `ArrayList` 对象都可以。因此，不要让 Cache 有太多的对象，以免内存不足。

2. FIFO Cache 初始化

在这里，`Map` 类型变量 `Cache` 一定要 `synchronized`，保证线程的安全性。即所有不同的访问产生的线程，在这里都同步成为一个线程。变量 `keyList` 也是同理。

```
private int cacheSize;
private Map cache;
private List keyList;

public FifoCacheController() {
    this.cacheSize = 100;
    this.cache = Collections.synchronizedMap(new HashMap());
    this.keyList = Collections.synchronizedList(new LinkedList());
}
```

上面代码说明当前缓存容器的最大量是 100，当然这个最大量也是可以修改的，可以通过 `setProperties` 方法来实现。

然后采用同步来实现多线程缓存容器（即 `Map` 变量 `Cache`）线程安全性。我们知道，如果多个线程同时访问一个 `HashMap` 变量，而其中至少一个线程从结构上修改了该 `HashMap`，则它必须保持外部同步。（结构上的修改是指添加或删除一个或多个映射关系的任何操作，仅改变与实例已经包含的键关联的值不是结构上的修改。）这一般通过使用 `Collections.synchronizedMap` 方法来“包装”该 `HashMap`。并且最好在创建时完成这一操作，以防止对映射进行意外的非同步访问，如上面代码所示。由所有此类的“collection 视图方法”所返回的迭代器都是快速失败的：在迭代器创建之后，如果从结构上对映射进行修改，除非通过迭代器本身的 `remove` 方法，其他任何时间任何方式的修改，迭代器都将抛出 `ConcurrentModificationException`。因此，面对并发的修改，迭代器很快就会完全失败，而不会冒将来不确定的时间发生任意不确定行为的风险。

`Map` 数据类型变量 `cache` 用于保存缓存对象的键值和缓存对象。而 `LinkedList` 数据类型变量 `keyList` 则用于保存缓存对象的排序码和键值，为什么用 `LinkedList` 数据类型呢？这是因为 `LinkedList` 具有排序功能。

3. putObject 方法

`FifoCacheController` 类 `putObject` 方法实现把缓存对象载入到同步线程的 `HashMap` 中。

```
public void putObject(CacheModel cacheModel, Object key, Object value) {
    cache.put(key, value);
    keyList.add(key);
    if (keyList.size() > cacheSize) {
```

```

try {
    //当达到了最高值时，删掉载入的第一个，这就是所谓的先进先出
    Object oldestKey = keyList.remove(0);
    cache.remove(oldestKey);
} catch (IndexOutOfBoundsException e) {
    //ignore
}
}
}

```

代码的关键是先加入缓存对象，然后判断整个缓存容器池是否已经超过了最大值。当超过了最大值，从 keyList 变量中获取最先进入的缓存对象键值，先从 keyList 变量中删除该排序对象，并从缓存容器中删掉这个缓存对象。

4. getObject 方法

FifoCacheController 类 getObject 方法就是从缓存容器池获得缓存对象。

```

public Object getObject(CacheModel cacheModel, Object key) {
    return cache.get(key);
}

```

按照缓存对象键值来获取缓存对象。

5. removeObject 方法

FifoCacheController 类 removeObject 方法就是从缓存容器池删除缓存对象。

```

public Object removeObject(CacheModel cacheModel, Object key) {
    keyList.remove(key);
    return cache.remove(key);
}

```

按照缓存对象键值来删除缓存对象。

6. flush 方法

清空缓存容器池里的全部缓存对象。

```

public void flush(CacheModel cacheModel) {
    cache.clear();
    keyList.clear();
}

```

包括两个步骤，第一是从缓存容器池清空所有数据，第二是从键值 List 中清空所有数据。

10.3.2 LRU 缓存实现

LRU Cache 实现用“近期最少使用”原则来确定如何从 Cache 中清除对象。当 Cache 溢

出时,最近最少使用的对象将被从 Cache 中清除。其实现类是 `com.ibatis.sqlmap.engine.cache.lru` 包的 `LruCacheController` 类。使用这种方法,如果一个特定的对象总是被使用,它将保留在 Cache 中,而且被清除的可能性最小。对于在较长的期间内,某些用户经常使用某些特定对象的情况,LRU Cache 是一个不错的选择。

1. LRU Cache 的配置

在映射文件中,LRU Cache 的配置如下所示。

```
<cacheModel id="product-cache" type="LRU">
  <flushInterval hours="24"/>
  <flushOnExecute statement="insertProduct"/>
  <flushOnExecute statement="updateProduct"/>
  <flushOnExecute statement="deleteProduct"/>
  <property name="size" value="1000" />
</cacheModel>
```

LRU Cache 实现只认可一个 `property` 元素,其名为“`cache-size`”的属性值必须是整数,代表同时保存在 Cache 中对象的最大数目。值得注意的是,这里指的对象可以是任意的,从单一的 `String` 对象到 Java Bean 的 `ArrayList` 对象都可以。

2. LRU Cache 初始化

与 FIFO Cache 初始化完全相同。实现三个任务。第一:设置缓存容器的最大量;第二,多线程同步缓存容器池;第三,多线程同步键值的排序码列表。

3. putObject 方法

与 FIFO Cache 完全相同,也是采用先进先出。

4. getObject 方法

获得一个对象的同时把这个对象删除掉再重新载入。实际上是最新获得的对象是最后进入。

```
public Object getObject(CacheModel cacheModel, Object key) {
    Object result = cache.get(key);
    keyList.remove(key);
    if (result != null) {
        keyList.add(key);
    }
    return result;
}
```

上面代码的实现过程就是先按照对象的键值在缓存容器池中寻找,同时也在键值排序列表中按照对象键值删除当前对象。当在缓存容器池中存在有该缓存对象时,则在键值排序列表加入该缓存对象的键值。实际上这也是最新进入的对象。

5. removeObject 方法

与 FIFO Cache 完全相同。

6. flush 方法

与 FIFO Cache 完全相同。

10.3.3 MEMORY 缓存实现

MEMORY Cache 实现使用 Reference 类型来管理 Cache 的行为。垃圾收集器可以根据 Reference 类型判断是否要回收 Cache 中的数据。MEMORY 实现适用于没有统一的对象重用模式的应用，或内存不足的应用。MEMORY Cache 的实现包括 com.ibatis.sqlmap.engine.cache.memory 包的 MemoryCacheController 类和 MemoryCacheLevel 类。

这是采用了 JDK 的垃圾回收机制。用到了 JDK 中的 java.lang.ref.SoftReference 类和 java.lang.ref.WeakReference；同时在 MEMORY 内部有一个内嵌类 StrongReference。同时用一个自关联 MemoryCacheLevel 类定义了三种缓存状态，分别针对 STRONG、SOFT 和 WEAK 三者其一，其类结构如图 10-3 所示。

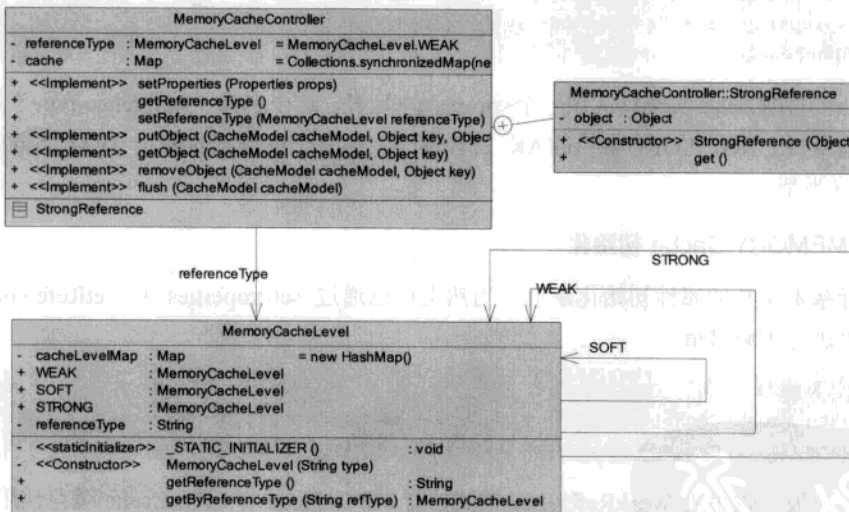


图 10-3 MEMORY 缓存实现的类结构图

MEMORY cache 实现只认识一个<property>元素。这个名为“Reference-type”属性的值必须是 STRONG、SOFT 和 WEAK 三者其一。这三个值分别对应于 JVM 不同的内存 Reference 类型。表 10-4 介绍了在 MEMORY 实现中不同的 Reference 类型。要更好地理解 Reference 类型，请参考 JDK 文档中的 java.lang.ref，以获得更多关于“reachability”的信息。

表 10-4 MEMORY 实现中不同的 Reference 类型

序 号	方 式	介 绍
1	WEAK (默认)	大多数情况下，WEAK 类型是最佳选择。如果不指定类型，默认类型就是 WEAK。它能大大提高常用查询的性能。但是对于当前不被使用的查询结果数据，将被清除以释放内存用来分配其他对象
2	SOFT:	在查询结果对象数据不被使用、同时需要内存分配其他对象的情况下，SOFT 类型将减少内存不足的可能性。然而，这不是最具侵入性的 Reference 类型，结果数据依然可能被清除
3	STRONG:	STRONG 类型可以确保查询结果数据一直保留在内存中，除非 Cache 被刷新（例如，到了刷新的时间或执行了更新数据的操作）。对于下面的情况，这是理想的选择：1) 结果内容数据很少，2) 完全静态的数据和 3) 频繁使用的数据。优点是对于这类查询性能非常好。缺点是，如果需要分配其他对象，内存无法释放（可能是更重要的数据对象）

1. MEMORY Cache 的配置

在映射文件中，MEMORY Cache 的配置如下所示。

```
<cacheModel id="product-cache" type="MEMORY">
  <flushInterval hours="24"/>
  <flushOnExecute statement="insertProduct"/>
  <flushOnExecute statement="updateProduct"/>
  <flushOnExecute statement="deleteProduct"/>
  <property name="reference-type" value="WEAK" />
</cacheModel>
```

MEMORY Cache 实现只认识一个<property>元素。这个名为“Reference-type”属性的值必须是 STRONG、SOFT 和 WEAK 三者其一。这三个值分别对应于 JVM 不同的内存 Reference 类型。

2. MEMORY Cache 初始化

进行基本参数的属性初始化赋值。当然也可以通过 setProperties 和 setReferenceType 等方法来进行重新赋值。

```
//默认为 WEAK
private MemoryCacheLevel referenceType = MemoryCacheLevel.WEAK;
private Map cache = Collections.synchronizedMap(new HashMap());
```

默认垃圾回收采用 WeakReference 模式，与 FIFO Cache 和 LRU Cache 缓存使用一样，都是采用线程安全性的缓存容器池。

3. putObject 方法

MemoryCacheController 类的 putObject 方法实现把基于缓存对象的 Reference 对象载入到同步线程的 HashMap 中。

```
public void putObject(CacheModel cacheModel, Object key, Object value) {
```

```

Object reference = null;
if (referenceType.equals(MemoryCacheLevel.WEAK)) {
    reference = new WeakReference(value);
} else if (referenceType.equals(MemoryCacheLevel.SOFT)) {
    reference = new SoftReference(value);
} else if (referenceType.equals(MemoryCacheLevel.STRONG)) {
    reference = new StrongReference(value);
}
cache.put(key, reference);
}

```

首先判断是何种的 Reference，如果是 WEAK 的类型就实例化 WeakReference 对象。如果是 SOFT 类型，就实例化 SoftReference 对象。如果是 STRONG 类型，就实例化 StrongReference 对象。然后把当前实例化的 Reference 对象按照键值编码加入到 HashMap 数据类型的缓存容器池中。

4. getObject 方法

MemoryCacheController 类的 getObject 方法从缓存容器池中获取对应的缓存对象。

```

public Object getObject(CacheModel cacheModel, Object key) {
    Object value = null;
    Object ref = cache.get(key);
    if (ref != null) {
        if (ref instanceof StrongReference) {
            value = ((StrongReference) ref).get();
        } else if (ref instanceof SoftReference) {
            value = ((SoftReference) ref).get();
        } else if (ref instanceof WeakReference) {
            value = ((WeakReference) ref).get();
        }
    }
    return value;
}

```

首先从缓存容器中获得 Reference 对象，然后根据 Reference 对象的类型进行类型转化，再然后从 Reference 对象获得缓存对象。

5. removeObject 方法

MemoryCacheController 类的 removeObject 方法是从缓存容器池中删除缓存对象。

```

public Object removeObject(CacheModel cacheModel, Object key) {
    Object value = null;
    Object ref = cache.remove(key);
    if (ref != null) {
        if (ref instanceof StrongReference) {
            value = ((StrongReference) ref).get();
        } else if (ref instanceof SoftReference) {
            value = ((SoftReference) ref).get();
        }
    }
}

```

```

    } else if (ref instanceof WeakReference) {
        value = ((WeakReference) ref).get();
    }
}
return value;
}

```

实际上是从缓存容器池中删除缓存对象的键值。

6. flush 方法

MemoryCacheController 类的 flush 方法是清空缓存容器池的缓存对象。

```

public void flush(CacheModel cacheModel) {
    cache.clear();
}

```

10.3.4 OSCACHE 缓存实现

OSCACHE Cache 实现是 OSCache2.0 缓存引擎的一个 Plugin。它具有高度的可配置性、分布式和高度的灵活性。OSCACHE Cache 的实现包括 com.ibatis.sqlmap.cache.oscache 包的 OSCacheController 类。主要用到的 GeneralCacheAdministrator 方法如下。

public Object getFromCache (String key) throws NeedsRefreshException;——从缓存中获取一个 key 标识的对象。

public Object getFromCache (String key, int refreshPeriod) throws NeedsRefreshException;——从缓存中获取一个 key 标识的对象。refreshPeriod 刷新周期，标识此对象在缓存中保存的时间（单位:秒）。

public void putInCache (String key, Object content) ——存储一个由 Key 标识的缓存对象。

public void putInCache (String key, Object content, String[] groups) ——存储一个由 Key 标识的属于 groups 中所有成员的缓存对象。

public void flushEntry (String key) ——更新一个 Key 标识的缓存对象。

public void flushGroup (String group) ——更新一组属于 groupr 标识的所有缓存对象。

public void flushAll()——更新所有缓存。

public void cancelUpdate (String key) ——取消更新，只用于在处理捕获的 Needs RefreshException 异常并尝试生成新缓存内容失效的时候。

public void removeEntry (String key) ——从缓存中移除一个 key 标识的对象。

public void clear()——清除所有缓存。

1. OSCACHE 的配置

在映射文件中，OSCACHE 的配置如下所示。


```
<cacheModel id="product-cache" type="OSCACHE">
  <flushInterval hours="24"/>
  <flushOnExecute statement="insertProduct"/>
  <flushOnExecute statement="updateProduct"/>
  <flushOnExecute statement="deleteProduct"/>
</cacheModel>
```

OSCACHE 实现不使用 `property` 元素，而是在类路径的根路径中使用标准的 `oscache.properties` 文件进行配置。在 `oscache.properties` 文件中，可以配置 Cache 的算法，Cache 的大小，持久化方法（内存，文件等）和集群方法。

2. OSCACHE Cache 初始化

OSCACHE 首先实例化一个 `GeneralCacheAdministrator` 对象，而且这个对象还是一个静态的和最终的变量。

```
private static final GeneralCacheAdministrator CACHE = new GeneralCacheAdministrator();
```

3. putObject 方法

OSCACHE 类的 `putObject` 方法是调用 `CACHE.putInCache(keyString, object, new String[]{CacheModel.getId()})` 方法。存储一个由 `keyString` 标识的属于 `groups` 中所有成员的缓存对象 `Object`。

```
public void putObject(CacheModel cacheModel, Object key, Object object) {
    String keyString = key.toString();
    CACHE.putInCache(keyString, object, new String[]{cacheModel.getId()});
}
```

把对 OSCACHE 类的操作转移给静态的 `GeneralCacheAdministrator` 对象的方法。

4. getObject 方法

OSCACHE 类的 `getObject` 方法，采用 `getFromCache(String key, int refreshPeriod) throws NeedsRefreshException` 方法来获得缓存对象，即从缓存中获取一个 `key` 标识的对象。`refreshPeriod` 刷新周期，标识此对象在缓存中保存的时间（单位：秒）。

```
public Object getObject(CacheModel cacheModel, Object key) {
    String keyString = key.toString();
    try {
        int refreshPeriod = (int) (cacheModel.getFlushIntervalSeconds());
        return CACHE.getFromCache(keyString, refreshPeriod);
    } catch (NeedsRefreshException e) {
        //如果在这个期间没有这个对象，从缓存中删除这个对象的主键
        CACHE.cancelUpdate(keyString);
        return null;
    }
}
```

5. removeObject 方法

OSCACHE 类的 removeObject 方法

```
public Object removeObject(CacheModel cacheModel, Object key) {
    Object result;
    String keyString = key.toString();
    try {
        int refreshPeriod = (int) (cacheModel.getFlushIntervalSeconds());
        Object value = CACHE.getFromCache(keyString, refreshPeriod);
        if (value != null) {
            CACHE.flushEntry(keyString);
        }
        result = value;
    } catch (NeedsRefreshException e) {
        try {
            CACHE.flushEntry(keyString);
        } finally {
            CACHE.cancelUpdate(keyString);
            result = null;
        }
    }
    return result;
}
```

6. flush 方法

OSCACHE 类的 flush 方法是清空这个组的对象。

```
public void flush(CacheModel cacheModel) {
    CACHE.flushGroup(cacheModel.getId());
}
```

10.4 扩展缓存策略——增加先进后出缓存策略

可以按照 iBATIS 缓存的设计框架来扩展缓存策略,比如我想增加一种缓存控制方式。这种方式是按照先进入的缓存对象后出的思路来进行的。

在 iBATIS 框架的编程方面,只需要增加一个缓存控制器类即可。但是这个缓存控制器类必须实现 com.ibatis.sqlmap.engine.cache.CacheController 接口。该接口包括的方法如下。

```
public interface CacheController {

    //清空缓存控制器内全部缓存对象
    public void flush(CacheModel cacheModel);

    //根据缓存内码从缓存控制器中获取缓存对象
    public Object getObject(CacheModel cacheModel, Object key);
}
```

```

//缓存控制器中删除缓存对象
public Object removeObject(CacheModel cacheModel, Object key);

//加入缓存对象到缓存控制器中
public void putObject(CacheModel cacheModel, Object key, Object object);

//设置缓存控制器初始化变量
public void setProperties(Properties props);

}

```

其创建的先进后出缓存控制器的代码如下。

```

/** FILO 先进后出缓存控制器实现*/
public class FiloCacheController implements CacheController {
    private int cacheSize;
    private Map cache;
    private List keyList;
    public FiloCacheController() {
        this.cacheSize = 100;
        this.cache = Collections.synchronizedMap(new HashMap());
        this.keyList = Collections.synchronizedList(new LinkedList());
    }

    public int getCacheSize() {return cacheSize;}

    public void setCacheSize(int cacheSize) {this.cacheSize = cacheSize;}

    /** 初始化缓存变量 */
    public void configure(Properties props) {
        setProperties(props);
    }

    public void setProperties(Properties props) {
        String size = props.getProperty("cache-size");
        if (size == null) {size = props.getProperty("size");}
        if (size != null) {cacheSize = Integer.parseInt(size); }
    }

    /**给缓存容器加入一个缓存对象*/
    public void putObject(CacheModel cacheModel, Object key, Object value) {
        // 当达到了缓存容器的最大值，删除最后的一个，即最后进入的那个缓存对象
        if (keyList.size() == cacheSize) {
            try {
                Object oldestKey = keyList.remove(keyList.size() - 1);
                cache.remove(oldestKey);
            } catch (IndexOutOfBoundsException e) {
                // ignore
            }
        }
        cache.put(key, value);
    }
}

```

```

        keyList.add(key);
    }

    /**从缓存容器中获得一个缓存对象*/
    public Object getObject(CacheModel cacheModel, Object key) {
        return cache.get(key);
    }

    /**从缓存容器中删除一个缓存对象 */
    public Object removeObject(CacheModel cacheModel, Object key) {
        keyList.remove(key);
        return cache.remove(key);
    }

    /** 清空缓存容器/
    public void flush(CacheModel cacheModel) {
        cache.clear();
        keyList.clear();
    }
}

```

在调用该缓存控制器的时候，要在两个地方设置，一个是把该缓存控制器的类告诉给 iBATIS 系统，通过加入别名的方式。配置文件信息如下。

```

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

<sqlMapConfig>
    .....
    <typeAlias alias="FILO" type="com.ibatis.sqlmap.engine.cache.filo.FiloCacheController"
/>
    .....
</sqlMapConfig>

```

在具体使用该缓存策略的地方还要配置一下。在映射文件中配置如下：

```

<sqlMap namespace="Account">
    <typeAlias alias="account" type="com.ibatis.jpetstore.domain.Account" />
    <cacheModel id="account-cache" type="FILO">
        <flushInterval hours="24" />
        <flushOnExecute statement="getAccountByUsername" />
        <property name="size" value="1000" />
    </cacheModel>

    <select id="getAccountByUsername" resultClass="account"
        parameterClass="string">
        SELECT SIGNON.USERNAME, ACCOUNT.EMAIL, ACCOUNT.FIRSTNAME,
        ACCOUNT.LASTNAME, ACCOUNT.STATUS, ACCOUNT.ADDR1 AS address1,
        ACCOUNT.ADDR2 AS address2, ACCOUNT.CITY, ACCOUNT.STATE,
        ACCOUNT.ZIP, ACCOUNT.COUNTRY, ACCOUNT.PHONE, PROFILE.LANGPREF AS
        languagePreference, PROFILE.FAVCATEGORY AS favouriteCategoryId,

```

```

        PROFILE.MYLISTOPT AS listOption, PROFILE.BANNEROPT AS
        bannerOption, BANNERDATA.BANNERNAME FROM ACCOUNT, PROFILE,
        SIGNON, BANNERDATA WHERE ACCOUNT.USERID = #username# AND
        SIGNON.USERNAME = ACCOUNT.USERID AND PROFILE.USERID =
        ACCOUNT.USERID AND PROFILE.FAVCATEGORY = BANNERDATA.FAVCATEGORY
    </select>
    .....
</sqlMap>

```

这样，就可以使用自定义的缓存控制器了。

本扩展缓存控制器的源程序代码参见本书的配套光盘，该盘已经通过了测试。

10.5 读取源码的收获

通过我们对 iBATIS DAO 源码的理解和分析，应该有以下这几个收获。

第一，实现缓存的核心任务是什么？主要是完成两个核心任务：就是命中率策略和淘汰算法策略。

第二，学习 SQL Map 缓存处理 CacheModel 包装类的实现，同时在如下几个方面也有所体会：① 如何进行一个缓存池容器的创建、包括这个缓存池中缓存对象的分配及释放。② 缓存池中如何进行并发处理，采用线程同步方法来实现缓存处理的唯一性。③ CacheModel 包装类采用了工厂方法模式实现多种缓存控制器策略，包括一些常用的缓存淘汰算法处理，如 FIFO Cache（先进先出）、LRU Cache（近期最少使用）、MEMORY Cache（调用垃圾收集器来实现缓存）和 OSCACHE Cache（OSCache2.0 缓存引擎的调用）。

第三，唯一性 CacheKey 对象的产生，这是通过不停地进行各个对象 HashCode 处理最后形成一个 CacheKey 对象，而且该 CacheKey 对象还具有可逆性和追溯性。

第四，MEMORY Cache 的实现，让我们更好地了解垃圾收集器使用的原理，包括使用 Reference 类型来回收 Cache 中的数据。Reference 类型有 WEAK（默认）、SOFT 和 STRONG 等。

第五，SQL Map 的缓存控制器策略采用了工厂方法模式，同时也结合了动态类加载技术。这样可以保证软件系统的稳定性、延续性和扩张性。我们在学习这种设计策略的同时，也给 SQL Map 的缓存控制器策略增加了一种新的缓存策略——先进后出缓存策略。这也说明这种设计方式在延续性和扩张性方面的优势。

第 11 章

TypeHandler 类型转化

本章内容:

1. Java 的数据类型说明
2. TypeHandler 组件的框架结构
3. TypeHandlerFactory 的结构、作用和实现
4. TypeHandler 的实现

在进行 ORM 框架编写中,首先要面对数据类型的转化。因为这要面临三种数据类型。首先是数据库有自己的数据类型,其次是 JDBC 也有自己独有的数据类型,再者就是 Java 语言本身的数据类型。对于 Java 语言本身也有两种类型,一种是 Java 基本数据类型,还有一种是 Java 的对象类型。由于在 SQL 和 Java 编程语言数据类型的数据类型不相同,所以在实现 ORM 时需要使用 Java 之间的类型和数据库使用 SQL 类型进行数据类型转换机制。

11.1 Java 的数据类型的说明

在 iBATIS 平台要涉及很多的类型转化。比如 Java 数据类型和 JDBC 数据类型的转化。所以,平台有专门的一个组件来处理这方面的内容。作者本来想把 JavaBean、XML、JDBC 等类型处理都一块来解决。但是现在的版本好像只是处理了 JavaBean 数据类型和 JDBC 数据类型之间的转化。

JDBC 定义一个通用的 SQL 类 `java.sql.Types` 的类型标识符集。这些类型标识符代表最常用的 SQL 类型。在 JDBC API 编程中,开发人员通常能够使用这些 JDBC 类型,而不是特定数据库的 SQL 类型。

其中,JDBC 数据类型映射到 Java 基本数据类型见附录六:JDBC Types Mapped to Java Types。

其中,Java 基本数据类型映射到 JDBC 数据类型见附录七:Java Types Mapped to JDBC

Types。

其中，JDBC 数据类型映射到 Java 对象类型见附录八：JDBC Types Mapped to Java Object Types。

其中，Java 对象类型映射到 JDBC 数据类型见附录九：Java Object Types Mapped to JDBC Types。

其中，JDBC 数据类型映射到特定数据库 SQL 类型见附录十：JDBC Types Mapped to Database-specific SQL Types。

关于 JDBC 数据类型的区别详见《Getting Started with the JDBC API》，网址：
<http://java.sun.com/j2se/1.3/docs/guide/jdbc/getstart/GettingStartedTOC.fm.html>。

11.2 TypeHandler 组件的框架结构

上面描述了关于 Java 的数据类型转化，在 iBATIS SQL Map 中有专门的组件来解决这个问题，这个组件就是 TypeHandler 组件。TypeHandler 组件框架主要是在 com.ibatis.sqlmap.engine.type 包内。其设计模式采用简单工厂模式和模板模式相结合。

简单工厂模式主要体现在 TypeHandler 的实例化对象都是通过 TypeHandlerFactory 工厂类来创建的，这是一种工厂创建的方式。其结构如图 11-1 所示，其中 *TypeHandler 表示在 com.ibatis.sqlmap.engine.type 包中多个以 TypeHandler 结尾的类。

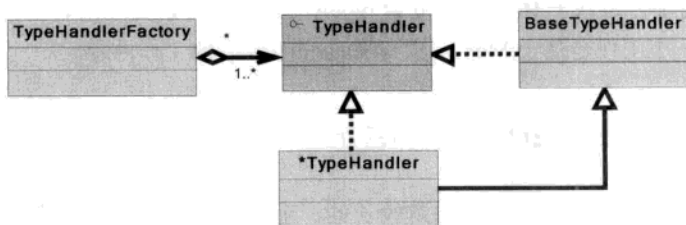


图 11-1 简单工厂模式类结构图

TypeHandlerFactory 类对应多个 TypeHandler 实例化对象。而 BaseTypeHandler 抽象类实现 TypeHandler 接口被 *TypeHandler 类继承。这是一种模板模式，即一个实现类继承抽象类并实现一个接口。TypeHandler 组件中要实现的接口是 TypeHandler 接口。抽象类是 BaseTypeHandler 类，其实现类分别是 LongTypeHandler 类、StringTypeHandler 类、EnumTypeHandler 类、DateTypeHandler 类、DoubleTypeHandler 类等。类型转化的类结构如图 11-2 所示。

按照上面的分类，TypeHandler 组件框架的内容包括两个方面，一方面是基于 TypeHandlerFactory 类的处理，另一方面是具体实现数据转化的 TypeHandler 类。下面就按照这两个方面来进行讲解。

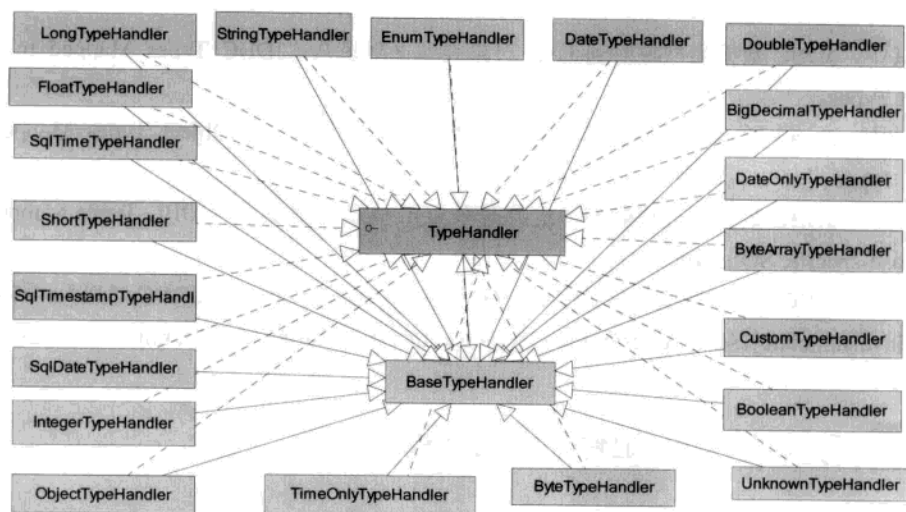


图 11-2 模板模式类结构图

11.3 TypeHandlerFactory 的结构、作用和实现

首先要搞明白 TypeHandlerFactory 类。TypeHandlerFactory 类实际上要处理两个部分的工作，一部分的工作是基于 HashMap 变量 typeAliases，主要是处理别名和类名之间的映射对应关系。另一个工作是基于 Map 变量 typeHandlerMap 的工作内容。

11.3.1 TypeHandlerFactory 的别名处理

什么是别名处理？就是为一个通常较长的、全限定类名指定一个较短的别名。基于 HashMap 变量 typeAliases 的工作比较简单，例如在配置文件中有下列一些信息。

```
<typeAlias alias="shortname" type="com.long.class.path.Class"/>
```

即可实现当调用 shortname 时，可以去调用 com.long.class.path.Class。

在 iBATIS 平台中，TypeHandlerFactory 有默认的别名转化。在 TypeHandlerFactory 的初始化方法中就有这样的程序代码。

```
putTypeAlias("string", String.class.getName());
putTypeAlias("byte", Byte.class.getName());
putTypeAlias("long", Long.class.getName());
putTypeAlias("short", Short.class.getName());
putTypeAlias("int", Integer.class.getName());
putTypeAlias("integer", Integer.class.getName());
putTypeAlias("double", Double.class.getName());
putTypeAlias("float", Float.class.getName());
```

```

putTypeAlias("boolean", Boolean.class.getName());
putTypeAlias("date", Date.class.getName());
putTypeAlias("decimal", BigDecimal.class.getName());
putTypeAlias("object", Object.class.getName());
putTypeAlias("map", Map.class.getName());
putTypeAlias("hashmap", HashMap.class.getName());
putTypeAlias("list", List.class.getName());
putTypeAlias("arraylist", ArrayList.class.getName());
putTypeAlias("collection", Collection.class.getName());
putTypeAlias("iterator", Iterator.class.getName());
putTypeAlias("cursor", java.sql.ResultSet.class.getName());

```

同时在 SqlMapConfiguration 实例化对象的初始化方法中就已经做了这样的处理。

```

private void registerDefaultTypeAliases() {
    // TRANSACTION ALIASES
    typeHandlerFactory.putTypeAlias("JDBC", JdbcTransactionConfig.class.getName());
    typeHandlerFactory.putTypeAlias("JTA", JtaTransactionConfig.class.getName());
    typeHandlerFactory.putTypeAlias("EXTERNAL", ExternalTransactionConfig.class.
getName());

    // DATA SOURCE ALIASES
    typeHandlerFactory.putTypeAlias("SIMPLE", SimpleDataSourceFactory.class.getName());
    typeHandlerFactory.putTypeAlias("DBCP", DbcDataSourceFactory.class.getName());
    typeHandlerFactory.putTypeAlias("JNDI", JndiDataSourceFactory.class.getName());

    // CACHE ALIASES
    typeHandlerFactory.putTypeAlias("FIFO", FifoCacheController.class.getName());
    typeHandlerFactory.putTypeAlias("LRU", LruCacheController.class.getName());
    typeHandlerFactory.putTypeAlias("MEMORY", MemoryCacheController.class.getName());
    // use a string for OSCache to avoid unnecessary loading of properties upon init
    typeHandlerFactory.putTypeAlias("OSCACHE", "com.ibatis.sqlmap.engine.cache.
oscache.OSCacheController");

    // TYPE ALIASES
    typeHandlerFactory.putTypeAlias("dom", DomTypeMarker.class.getName());
    typeHandlerFactory.putTypeAlias("domCollection", DomCollectionTypeMarker.class.
getName());
    typeHandlerFactory.putTypeAlias("xml", XmlTypeMarker.class.getName());
    typeHandlerFactory.putTypeAlias("xmlCollection", XmlCollectionTypeMarker.class.
getName());
}

```

最终实现的这样的别名和全限定类名的映射，如表 11-1 所示。

表 11-1 默认别名与全限定类名的映射

别 名	类 名 称	作 用
JDBC	com.ibatis.sqlmap.engine.transaction.jdbc.JdbcTransactionConfig	Jdbc 的配置信息类
JTA	com.ibatis.sqlmap.engine.transaction.jta.JtaTransactionConfig	Jdbc 的配置信息类
EXTERNAL	com.ibatis.sqlmap.engine.transaction.external.ExternalTransactionConfig	Jdbc 的配置信息类

续表

别 名	类 名 称	作 用
SIMPLE	com.ibatis.sqlmap.engine.datasource.SimpleDataSourceFactory	iBATIS 实现的一个 DataSource 类
DBCP	com.ibatis.sqlmap.engine.datasource.DbcpDataSourceFactory	基于 Dbcp 方式的 DataSource 实现类
JNDI	com.ibatis.sqlmap.engine.datasource.JndiDataSourceFactory	基于 Jndi 方式的 DataSource 实现类
FIFO	com.ibatis.sqlmap.engine.Cache.fifo.FifoCacheController	先进先出的缓存控制器实现类
LRU	com.ibatis.sqlmap.engine.Cache.lru.LruCacheController	近期最少使用的缓存控制器实现类
MEMORY	com.ibatis.sqlmap.engine.Cache.memory.MemoryCacheController	MEMORY 的缓存控制器实现类
OSCACHE	com.ibatis.sqlmap.engine.Cache.osCache.OSCacheController	OSCACHE 的缓存控制器实现类
dom	com.ibatis.sqlmap.engine.type.DomTypeMarker	Dom 类型数据转化包装类
domCollection	com.ibatis.sqlmap.engine.type.DomCollectionTypeMarker	DomCollection 类型数据转化包装类
XML	com.ibatis.sqlmap.engine.type.XmlTypeMarker	XML 类型数据转化包装类
XML Collection	com.ibatis.sqlmap.engine.type.XmlCollectionTypeMarker	XML Collection 类型数据转化包装类

11.3.2 TypeHandlerFactory 容器的数据类型转化

TypeHandlerFactory 类的主要内容还是第二部分工作。即以 Map 变量 typeHandlerMap 核心的数据类型转化工作。首先看一下变量 typeHandlerMap 的数据结构。

typeHandlerMap 的 key 是类对象，即一个类，是 JDK 中的标准数据类型。value 是一个 HashMap。对于 value 的 HashMap 的数据结构：key 是 jdbcType 的名称，value 是一个实现 TypeHandler 接口的实例化对象。这样就可以形成一个父子模式的树形数据结构。即一个 Java 标准类，可以对应多个 jdbcType 名称的 TypeHandler 接口的实例化对象。由于 HashMap 允许 key 放入 null。这样第二个 HashMap 中就可以放入 null 的 key，然后保持这样的数据结构。

在 TypeHandlerFactory 对象的初始化过程中，把 Java 标准数据类型和 jdbc 的数据类型以及 TypeHandler 实例化对象结合关联起来。

Java 提供两种不同的类型：引用类型和原始类型（或内置类型）。Int 是 Java 的原始数据类型，Integer 是 Java 为 Int 提供的封装类。Java 为每个原始类型提供了封装类，

如表 11-2 所示。

表 11-2 Java 原始类型和封装类

序 号	原始类型	封 装 类
1	boolean	Boolean
2	char	Character
3	byte	Byte
4	short	Short
5	int	Integer
6	long	Long
7	float	Float
8	double	Double

引用类型和原始类型的行为完全不同，并且它们具有不同的语义。引用类型和原始类型具有不同的特征和用法，它们包括：大小和速度问题，这种类型以哪种类型的数据结构存储，当引用类型和原始类型用作某个类的实例数据时所指定的默认值时。对象引用实例变量的默认值为 null，而原始类型实例变量的默认值与它们的类型有关。

11.4 TypeHandler 的实现

下一步我们看看 TypeHandler 能实现什么内容？TypeHandler 要实现的数据类型转化，包括有 BigDecimalTypeHandler 类、BooleanTypeHandler 类、ByteArrayTypeHandler 类、ByteTypeHandler 类、CustomTypeHandler 类等，内容如表 11-3 所示。

表 11-3 TypeHandler 类和实现功能列表

类 名 称	功能和用途
BigDecimalTypeHandler	BigDecimal 数据类型处理和转化的实现类
BooleanTypeHandler	Boolean 数据类型处理和转化的实现类
ByteArrayTypeHandler	byte[]数据类型处理和转化的实现类
ByteTypeHandler	Byte 数据类型处理和转化的实现类
CustomTypeHandler	定制化数据类型处理和转化的实现类
DateOnlyTypeHandler	Date 数据类型处理和转化的实现类
DateTypeHandler	Date (和 time) 数据类型处理和转化的实现类
DoubleTypeHandler	Double 数据类型处理和转化的实现类
EnumTypeHandler	String 数据类型处理和转化的实现类
FloatHandler	Float 数据类型处理和转化的实现类
IntegerTypeHandler	Integer Decimal 数据类型处理和转化的实现类
LongTypeHandler	Long 数据类型处理和转化的实现类
ObjectTypeHandler	Object 数据类型处理和转化的实现类
ShortTypeHandler	Short 数据类型处理和转化的实现类
Sql DateTypeHandler	Sql Date 数据类型处理和转化的实现类

续表

类 名 称	功能和用途
Sql TimestampTypeHandler	SQL timestamp 数据类型处理和转化的实现类
Sql TimeTypeHandler	SQL time 数据类型处理和转化的实现类
StringTypeHandler	String 数据类型处理和转化的实现类
TimeOnlyTypeHandler	Time (only) 数据类型处理和转化的实现类
UnknownTypeHandler	通用数据类型处理和转化的实现类

TypeHandler 接口用于说明其要求实现的内容，其方法列表如表 11-4 所示。

表 11-4 TypeHandler 接口的方法列表

方法名称	参 数	返 回 值	功能和用途	备 注
setParameter	PreparedStatement ps, int i, Object parameter, String jdbcType	Void	给 PreparedStatement 对象设置一个参数	
getResult	ResultSet rs, String columnName	Object	从 resultSet 获得一个列值	
getResult	ResultSet rs int columnIndex	Object	从 resultSet 获得一个列值	
getResult	CallableStatement cs int columnIndex	Object	从 callable statement 获得一个列值	
valueOf	String s	Object	转换字符串到处理程序处理的类型	
equals	Object object String string	boolean	比较两个值（即此程序处理）是否相等	

这六个方法的功能和参数进行详细说明如下。

```
public void setParameter(PreparedStatement ps, int i, Object parameter, String
jdbcType) throws SQLException;
```

该方法给 PreparedStatement 对象设置一个参数。其中第 1 个参数表示要设置的 PreparedStatement 对象，第 2 个参数表示要赋值的位数，第 3 个参数表示要赋值的对象，第 4 个参数表示要赋值的 jdbcType 的类型。

```
public Object getResult(ResultSet rs, String columnName) throws SQLException;
```

该方法表示从 resultSet 获得一个列值。其中第 1 个参数表示要获得值的 resultSet 对象，第 2 个参数表示要获得值的列名称。

```
public Object getResult(ResultSet rs, int columnIndex) throws SQLException;
```

该方法表示从 resultSet 获得一个列值。其中第 1 个参数表示要获得值的 resultSet 对象，第 2 个参数表示要获得值的列位。

```
public Object getResult(CallableStatement cs, int columnIndex) throws SQLException;
```

该方法表示从 callable statement 获得一个列值。其中第 1 个参数表示要获得值的 CallableStatement 对象，第 2 个参数表示要获得值的列位。

```
public Object valueOf(String s);
```

该方法表示转换字符串到处理程序处理的类型。其中参数表示要转化的数据类型的名称。

```
public boolean equals(Object object, String string);
```

该方法比较两个值（即此程序处理）是否相等。其中第 1 个参数表示一个对象，第 2 个参数表示数据类型的名称。

针对处理的类型，我们可以分为四类，一类是一般类型，第二类是 SQL 数据类型，第三类是定制类型，还有一类是不可知类型。下面分别用程序代码来解释这些数据转化是如何实现的。

11.4.1 一般类型的处理

一般数据类型包括 BigDecimal、Boolean、ByteArray、Byte、DateOnly、Date、Double、Enum、Float、Integer、Long、Object、Short、String 等。

不失一般性，我们以 StringTypeHandler 来举例说明。

1. StringTypeHandler 类的 setParameter 方法实现

代码如下，非常简单，直接调用 PreparedStatement 对象的 setString 方法进行赋值。

```
public void setParameter(PreparedStatement ps, int i, Object parameter, String
jdbcType) throws SQLException {
    ps.setString(i, ((String) parameter));
}
```

2. StringTypeHandler 类的 getResult(ResultSet rs, String columnName)方法的实现

代码如下，也非常简单，直接调用 ResultSet 对象的 getString 方法获得返回值。

```
public Object getResult(ResultSet rs, String columnName) throws SQLException {
    Object s = rs.getString(columnName);
    if (rs.wasNull()) { return null;}
    else { return s; }
}
```

3. StringTypeHandler 类的 getResult(ResultSet rs, int columnIndex)方法的实现

代码如下，还是非常简单，直接调用 ResultSet 对象的 getResult 方法获得返回值。

```
public Object getResult(ResultSet rs, int columnIndex) throws SQLException {
    Object s = rs.getString(columnIndex);
    if (rs.wasNull()) { return null;}
    else { return s; }
}
```

4. StringTypeHandler 类的 getResult(CallableStatement cs, int columnIndex)方法的实现

代码如下，还是非常简单，直接调用 CallableStatement 对象的 getString 方法获得返回值。

```
public Object getResult(CallableStatement cs, int columnIndex) throws SQLException {
    Object s = cs.getString(columnIndex);
    if (cs.isNull()) { return null; }
    else { return s; }
}
```

5. StringTypeHandler 类的 valueOf 方法的实现

由于都是字符串变量，直接返回参数值。

```
public Object valueOf(String s) {
    return s;
}
```

6. StringTypeHandler 类的 equals 方法的实现

StringTypeHandler 类的 equals 方法调用的是祖先 BaseTypeHandler 抽象类的 equals 方法。

```
public boolean equals(Object object, String string) {
    if (object == null || string == null) {
        return object == string;
    } else {
        Object castedObject = valueOf(string);
        return object.equals(castedObject);
    }
}
```

11.4.2 Sql 类型的处理

一般数据类型包括 SqlDate、SqlTimestamp、SqlTime、TimeOnly 等。不失一般性，我们以 SqlDateTypeHandler 来举例说明。

1. SqlDateTypeHandler 类的 setParameter 方法的实现

代码如下。但是在赋值的时候进行了数据类型的转化。赋值参数转化为了 java.sql.Date 数据类型。

```
public void setParameter(PreparedStatement ps, int i, Object parameter, String
jdbcType) throws SQLException {
    ps.setDate(i, (java.sql.Date) parameter);
}
```


2. SqlDateTypeHandler 类的 getResult(ResultSet rs, String columnName)方法的实现

代码如下。直接调用 ResultSet 对象的 getDate 方法获得返回值。返回值的数据类型是 java.sql.Date。

```
public Object getResult(ResultSet rs, String columnName) throws SQLException {
    Object sqlDate = rs.getDate(columnName);
    if (rs.wasNull()) { return null;}
    else {return sqlDate; }
}
```

3. SqlDateTypeHandler 类的 getResult(ResultSet rs, int columnIndex)方法的实现

代码如下。直接调用 ResultSet 对象的 getDate 方法获得返回值。返回值数据类型是 java.sql.Date。

```
public Object getResult(ResultSet rs, int columnIndex) throws SQLException {
    Object s = rs.getString(columnIndex);
    if (rs.wasNull()) { return null;}
    else { return s; }
}
```

4. SqlDateTypeHandler 类的 getResult(CallableStatement cs, int columnIndex)方法的实现

代码如下。直接调用 ResultSet 对象的 getDate 方法获得返回值。返回值的数据类型是 java.sql.Date。

```
public Object getResult(CallableStatement cs, int columnIndex) throws
SQLException {
    Object sqlDate = cs.getDate(columnIndex);
    if (cs.wasNull()) { return null;}
    else { return sqlDate; }
}
```

5. SqlDateTypeHandler 类的 valueOf 方法的实现

通过时间格式转化得到返回值。

```
public Object valueOf(String s) {
    return SimpleDateFormat.format(DATE_FORMAT, s);
}
```

创建 SimpleDateFormat 对象来转化格式。

```
public class SimpleDateFormat {
    public static Date format(String format, String datetime) {
        try {
            return new SimpleDateFormat(format).parse(datetime);
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

```

    } catch (ParseException e) {
        throw new SqlMapException("Error parsing default null value date. Format must
be '" + format + "'. Cause: " + e);
    }
}

```

6. SqlDateTypeHandler 类的 equals 方法的实现

SqlDateTypeHandler 类的 equals 方法调用的是祖先 BaseTypeHandler 抽象类的 equals 方法。

11.4.3 通用类型的处理

通用数据类型处理主要就是 UnknownTypeHandler 类的处理。

1. UnknownTypeHandler 类的 setParameter 方法的实现

UnknownTypeHandler 类的 setParameter 方法的实现代码如下。

```

public void setParameter(PreparedStatement ps, int i, Object parameter, String
jdbcType)
    throws SQLException {
    Class searchClass = parameter.getClass();
    if ( usingJavaPre5 ) {
        try {
            searchClass = getBaseClass(searchClass);
        }
        catch ( Exception ex ) {
            searchClass = null;
        }
    }
    if ( searchClass == null ) {
        searchClass = parameter.getClass();
    }
    TypeHandler handler = factory.getTypeHandler(searchClass, jdbcType);
    handler.setParameter(ps, i, parameter, jdbcType);
}

```

调用 TypeHandlerFactory 的 getTypeHandler(searchClass, jdbcType)方法，代码如下：

```

public TypeHandler getTypeHandler(Class type, String jdbcType) {
    Map jdbcHandlerMap = (Map) typeHandlerMap.get(type);
    TypeHandler handler = null;
    if (jdbcHandlerMap != null) {
        handler = (TypeHandler) jdbcHandlerMap.get(jdbcType);
        if (handler == null) {
            handler = (TypeHandler) jdbcHandlerMap.get(null);
        }
    }
    if (handler == null && type != null && Enum.class.isAssignableFrom(type)) {

```

```

        handler = new EnumTypeHandler(type);
    }
    return handler;
}

```

最后调用 TypeHandler 的 setParameter 方法。

2. UnknownTypeHandler 类的 getResult(ResultSet rs, String columnName)方法的实现

UnknownTypeHandler 类的 getResult(ResultSet rs, String columnName)方法的实现代码如下。

```

public Object getResult(ResultSet rs, String columnName) throws SQLException {
    Object object = rs.getObject(columnName);
    if (rs.wasNull()) {
        return null;
    } else {
        return object;
    }
}

```

直接调用 ResultSet 对象的 getObject 方法获得返回值。

3. UnknownTypeHandler 类的 getResult(ResultSet rs, int columnIndex)方法的实现

UnknownTypeHandler 类的 getResult(ResultSet rs, int columnIndex)方法与 getResult(ResultSet rs, String columnName)方法处理雷同，只是采用的序号不一样。

4. UnknownTypeHandler 类的 getResult(CallableStatement cs, int columnIndex)方法的实现

UnknownTypeHandler 类的 getResult(CallableStatement cs, int columnIndex)方法与 getResult(ResultSet rs, int columnIndex)方法处理雷同，只是传入的参数不一样。

5. UnknownTypeHandler 类的 valueOf 方法的实现

UnknownTypeHandler 类的 valueOf 方法的实现代码如下。

```

public Object valueOf(String s) {
    return s;
}

```

6. UnknownTypeHandler 类的 equals 方法的实现

UnknownTypeHandler 类的 equals 方法的实现代码如下。

```

public boolean equals(Object object, String string) {
    if (object == null || string == null) {
        return object == string;
    } else {
        TypeHandler handler = factory.getTypeHandler(object.getClass());
    }
}

```

```

    Object castedObject = handler.valueOf(string);
    return object.equals(castedObject);
}
}

```

11.4.4 定制数据类型的转化

在这里，有几个特殊的类型转化。主要是针对数据库中的大文本对象，包括 CLOB、LONGVARCHAR、BLOB、LONGVARBINARY 等，这些大字段都是通过 CustomTypeHandler 类来实现的。涉及的接口和类的类结构如图 11-3 所示。

为不失一般性，我们通过处理 Blob 对象来看看其实现过程。其主要调用的类有 CustomTypeHandler 类，BlobTypeHandlerCallback 类。CustomTypeHandler 对象把基本所有的操作都转移给 BlobTypeHandlerCallback 的同类方法去处理。

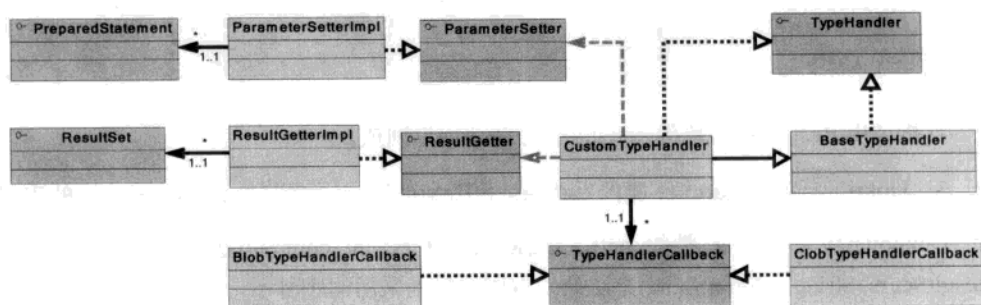


图 11-3 类结构图

1. CustomTypeHandler 类的 setParameter 方法的实现

CustomTypeHandler 类的 setParameter 方法，代码如下。

```

public void setParameter(PreparedStatement ps, int i, Object parameter, String
jdbcType) throws SQLException {
    ParameterSetter setter = new ParameterSetterImpl(ps, i);
    callback.setParameter(setter, parameter);
}

```

当 ParameterSetter 对象实例化时，其代码如下。

```

public ParameterSetterImpl(PreparedStatement statement, int columnIndex) {
    this.ps = statement;
    this.index = columnIndex;
}

```

而 BlobTypeHandlerCallback 类的 setParameter(setter, parameter) 方法，代码如下。

```

public void setParameter(ParameterSetter setter, Object parameter) throws SQLException {
    if (null != parameter) {

```

```

        byte[] bytes = (byte[]) parameter;
        ByteArrayInputStream bis = new ByteArrayInputStream(bytes);
        setter.setBinaryStream(bis, (int) (bytes.length));
    } else {
        setter.setNull(Types.BLOB);
    }
}

```

调用了 `ParameterSetter` 对象的 `setBinaryStream` 方法，代码如下。

```

public void setBinaryStream(InputStream x, int length) throws SQLException {
    ps.setBinaryStream(index, x, length);
}

```

2. CustomTypeHandler 类的 getResult (ResultSet rs, String columnName) 方法的实现

`CustomTypeHandler` 类的 `getResult(ResultSet rs, String columnName)` 方法，代码如下。

```

public Object getResult(ResultSet rs, String columnName) throws SQLException {
    ResultGetter getter = new ResultGetterImpl(rs, columnName);
    return callback.getResult(getter);
}

```

当 `ResultGetter` 对象实例化，其代码如下。

```

public ResultGetterImpl(ResultSet resultSet, int columnIndex) {
    this.rs = resultSet;
    this.index = columnIndex;
}

```

`ResultGetter` 是 `ResultSet` 处理的包装类。接着调用 `BlobTypeHandlerCallback` 类的 `getResult(getter)` 方法，其代码如下。

```

public Object getResult(ResultGetter getter) throws SQLException {
    Blob blob = getter.getBlob();
    byte[] returnValue;
    if (!getter.wasNull()) {
        returnValue = blob.getBytes(1, (int) blob.length());
    } else {
        returnValue = null;
    }
    return returnValue;
}

```

3. CustomTypeHandler 类的 getResult (ResultSet rs, int columnIndex) 方法的实现

基本与 `CustomTypeHandler` 类的 `getResult(ResultSet rs, String columnName)` 方法的实现相同。采用的是序号。这里就不做详细介绍了。

4. CustomTypeHandler 对象的 getResult 方法的实现

基本与 CustomTypeHandler 类的 getResult(ResultSet rs, int columnIndex)方法的实现相同。只是传入的参数是 CallableStatement 对象。这里就不做详细介绍了。

5. CustomTypeHandler 对象的 valueOf 方法的实现

CustomTypeHandler 类的 getResult 方法，代码如下。

```
public Object valueOf(String s) {  
    return callback.valueOf(s);  
}
```

接着调用 BlobTypeHandlerCallback 类的 valueOf 方法，其代码如下。

```
public Object valueOf(String s) {  
    return s;  
}
```

6. CustomTypeHandler 类的 equals 方法的实现

CustomTypeHandler 类的 equals 方法是调用祖先 BaseTypeHandler 抽象类的 equals 方法。

11.5 读取源码的收获

通过我们对 iBATIS TypeHandler 类型转化源码的理解和分析，应该有这几个收获。

第一，知道 Java 复杂的数据类型，尤其还要针对数据库处理。这些数据类型要涉及 Java 的基本数据类型、Java 的类数据类型，JDBC 的数据类型和各个厂家数据库的数据类型。

第二，iBATIS 采用工厂模式来实现 Java 数据类型的转化。对于所涉及的所有数据类型都采用专门封装的包装类，平台有很强的灵活性和扩展性。就算是再加一种新的数据类型，基本上可以不用编写 TypeHandler 包内的内容就可以直接实现了。

第 12 章

iBATIS 常用工具的实现

本章内容:

1. Resources 工具类的使用方法, 包括资源加载和实例化并缓存对象;
2. Bean 管理的实现, 包括 ClassInfo 类和 Probe 接口及其实现;
3. Log 组件类的使用和实现;
4. 调试信息工具;
5. ScriptRunner 的应用。

常用工具包括 Resources 工具类、Bean 管理、Log 管理等。这些程序代码都具有很大的移植性, 基本上稍微做一些修改就可以应用在其他系统上。

12.1 Resources 工具

Resources 类主要在 com.ibatis.common.resources 包内, 它为从类路径中加载资源, 提供了更加方便和简化的使用方法。Resources 类的方法全部都是静态方法, 所以, Resources 类是一个纯粹的工具类。Resources 类在 iBATIS 框架中主要实现两个方面的功能, 第一是进行资源的加载, 第二是实例化类并结合 ClassInfo 类进行缓存。

在 iBATIS 中, Resources 类常用于以下几种情况: ① 从类路径加载 SQL Map 配置文件 (如 SqlMap-config.xml), ② 从类路径加载 DAO Manager 配置文件 (如 dao.XML), ③ 从类路径加载各种.properties 文件。

对于字符转化, Resources 类中有一个 java.nio.charset.Charset 类型的静态变量 charset, 其主要实现在字节和 Unicode 字符之间转换的 charset、解码器和编码器。

12.1.1 资源加载

由于处理 ClassLoader 比较复杂, 容易出错, 尤其是在应用服务器/容器的情况下。因

此 `Resources` 类试图简化这些工作，为类路径中加载资源，提供了易于使用的方法。其调用方法如表 12-1 所示。

表 12-1 `Resources` 类的调用方法

序 号	加载方法	功能描述
1	<code>Reader getResourceAsReader(String resource);</code>	对于简单的只读文本数据，加载为 <code>Reader</code>
2	<code>Stream getResourceAsStream(String resource);</code>	对于简单的只读二进制或文本数据，加载为 <code>Stream</code>
3	<code>File getResourceAsFile(String resource);</code>	对于可读写的二进制或文本文件，加载为 <code>File</code>
4	<code>Properties getResourceAsProperties(String resource);</code>	对于只读的配置属性文件，加载为 <code>Properties</code>
5	<code>Url getResourceAsUrl(String resource);</code>	对于只读的通用资源，加载为 <code>URL</code>

1. 二进制或文本数据加载实现

对于简单的只读二进制或文本数据，调用 `Stream getResourceAsStream(String resource)` 方法加载为 `Stream`。其程序代码：

```
Stream stream = Resources.getResourceAsStream(String resource);
```

`Resources` 类的 `getResourceAsStream` 方法如下。

```
public static InputStream getResourceAsStream(String resource) throws IOException {
    return getResourceAsStream(getClassLoader(), resource);
}
```

要先调用 `getClassLoader()` 的代码如下：

```
private static ClassLoader getClassLoader() {
    //使用缺省的 ClassLoader，如果缺省的 ClassLoader 不存在，获取当前的执行线程上的
    ClassLoader
    if (defaultClassLoader != null) {
        return defaultClassLoader;
    } else {
        return Thread.currentThread().getContextClassLoader();
    }
}
```

使用默认的 `ClassLoader`，如果默认的 `ClassLoader` 不存在，获取当前的执行线程上的 `ClassLoader`，然后看 `getResourceAsStream(ClassLoader loader, String resource)` 的实现代码，代码如下：

```
public static InputStream getResourceAsStream(ClassLoader loader, String resource)
throws IOException {
    InputStream in = null;
    if (loader != null) in = loader.getResourceAsStream(resource);
    //采用系统的 ClassLoader 来获取资源。
    if (in == null) in = ClassLoader.getResourceAsStream(resource);
    if (in == null) throw new IOException("Could not find resource " + resource);
    return in;
}
```

如果当前线程的 `ClassLoader` 可用, 那么从线程的 `ClassLoader` 路径去寻找资源, 如果找到则把资源转化为 `Stream` 对象。如果没有找到资源则返回 `Stream` 对象为空。根据判断, `Stream` 对象是空则继续寻找, 这次是到系统的 `ClassLoader` 来获取资源, 然后把资源转化为 `Stream` 对象并返回出来。以上方法的 `resource` 参数名称应该是全限定名。

2. 只读文本加载实现

对于简单的只读文本数据, 这是一种字符数据流模式, 调用 `Reader getResourceAsReader(String resource)` 方法加载为 `Reader`。其程序代码:

```
Reader reader= Resources.getResourceAsReader(String resource);
```

`Resources` 类的 `getResourceAsReader` 方法如下。

```
public static Reader getResourceAsReader(String resource) throws IOException {
    Reader reader;
    if (charset == null) {
        reader = new InputStreamReader(getResourceAsStream(resource));
    } else {
        reader = new InputStreamReader(getResourceAsStream(resource), charset);
    }
    return reader;
}
```

根据 `Charset` 字符集类型来进行判断, 如果系统默认是不使用字符集, 就把流模式的数据传输转化成字符模式的数据传输。

3. 只读的通用资源加载实现

对于只读的通用资源, 调用 `Url getResourceAsUrl(String resource)`; 加载为 `URL`, 其程序代码如下。

```
Url url = Resources.getResourceAsUrl(String resource);
```

`Resources` 类的 `getResourceAsUrl` 方法如下。

```
public static URL getResourceURL(String resource) throws IOException {
    return getResourceURL(getClassLoader(), resource);
}
```

这里要先调用 `getClassLoader` 方法, 该方法首先使用默认的 `ClassLoader`, 如果默认的 `ClassLoader` 不存在, 则获取当前执行线程上的 `ClassLoader`。`getResourceURL(getClassLoader(), resource)` 实现代码的内容如下。

```
public static URL getResourceURL(ClassLoader loader, String resource) throws
IOException {
    URL url = null;
    if (loader != null) url = loader.getResource(resource);
    if (url == null) url = ClassLoader.getResource(resource);
}
```

```

        if (url == null) throw new IOException("Could not find resource " + resource);
        return url;
    }

```

如果当前线程的 `ClassLoader` 可用, 那么从线程的 `ClassLoader` 路径去寻找资源, 如果找到则把资源转化为 `URL` 对象。如果没有找到资源则返回 `URL` 对象为空。根据判断, `URL` 对象是空则继续寻找, 这次是到系统的 `ClassLoader` 来获取资源, 然后把资源转化为 `URL` 对象并返回出来。以上方法的 `resource` 参数名称应该是全限定名, 要求加上全文件/资源名。

4. 二进制或文本文件加载实现

对于可读写的二进制或文本文件, 调用 `File getResourceAsFile(String resource)`; 加载为 `File`。其程序代码如下。

```
File file = Resources.getResourceAsFile(String resource);
```

`Resources` 类的 `getResourceAsFile` 方法如下。

```

public static File getResourceAsFile(String resource) throws IOException {
    return new File(getResourceURL(resource).getFile());
}

```

通过通用资源加载, 获得 `Url` 对象, 然后调用 `Url` 对象的 `getFile` 方法, 转化为 `File` 对象。

5. 只读的配置属性文件加载实现

对于只读的配置属性文件, 调用 `Properties getResourceAsProperties(String resource)`; 加载为 `Properties`, 其程序代码如下。

```
Properties properties = Resources.getResourceAsProperties(resource);
```

`Resources` 类的 `getResourceAsProperties` 方法如下。

```

public static Properties getResourceAsProperties(String resource) throws
IOException {
    Properties props = new Properties();
    InputStream in = null;
    String propfile = resource;
    in = getResourceAsStream(propfile);
    props.load(in);
    in.close();
    return props;
}

```

创建一个 `Properties` 对象, 然后调用 `getResourceAsStream(propfile)` 方法而获取二进制 `Stream` 对象, 然后将 `Properties` 对象载入 `Stream` 对象, 实现 `Properties` 文件的载入。

在以上每个方法中, 加载资源和加载 `Resources` 类的为同一个 `ClassLoader`, 或者, 如果失败, 则将使用系统的 `ClassLoader`。实现的程序代码如下, 在源程序代码上添加了解析。

```
public static InputStream getResourceAsStream(String resource) throws IOException {
```

```
return getResourceAsStream(getClassLoader(), resource);
}
```

以上方法的 `resource` 参数名称应该是全限定名。例如，如果在类路径中有资源“`com.domain.mypackage.MyPropertiesFile.properties`”，则使用下面的代码加载资源为 `Properties`（注意，资源名前面不需要斜杠(/)）。

```
String resource = "com/domain/mypackage/MyPropertiesFile.properties";
Properties props = Resources.getResourceAsProperties(resource);
```

同样地，假设 `SQLMap` 配置文件在类路径的 `properties` 目录下（`properties/sqlMap-config.xml`），则可以从类路径将其加载为一个 `Reader`。

```
String resource = "properties/sqlMap-config.xml";
Reader reader = Resources.getResourceAsReader(resource);
SqlMapClient sqlMap = XmlSqlMapBuilder.buildSqlMap(reader);
```

12.1.2 实例化类并缓存

`Resources` 类可以对类进行实例化。同时其与 `ClassInfo` 类结合使用，可以把类的 `ClassInfo` 对象载入缓存列表，下一次再实例化时候就不用重新创建对象了。

`Resources` 类可以通过一个 `String` 类型的类名称，实例化成类名称的对象。外部调用该方法的代码如下（例如根据 `DataSourceFactory` 名称实例化相应的对象）。

```
DataSourceFactory dsFactory = (DataSourceFactory) Resources.instantiate(type);
```

`Resources` 类的 `instantiate` 方法实现的代码如下。

```
public static Object instantiate(String className)
    throws ClassNotFoundException, InstantiationException, IllegalAccessException {
    return instantiate(classForName(className));
}
```

先把 `String` 类型的类名称进行 `classForName` 方法处理，然后列出 `Resources` 类的 `classForName` 方法如下。

```
public static Class classForName(String className) throws ClassNotFoundException {
    Class clazz = null;
    try {
        clazz = getClassLoader().loadClass(className);
    } catch (Exception e) {
        // Ignore. Failsafe below.
    }
    if (clazz == null) {
        clazz = Class.forName(className);
    }
    return clazz;
}
```

通过名称先转化为对应名称的类。接着的代码是调用 `Resources` 类的 `instantiate(Class clazz)` 方法，代码如下。

```
public static Object instantiate(Class clazz) throws InstantiationException,
    IllegalAccessException {
    try {
        return ClassInfo.getInstance(clazz).instantiateClass();
    } catch (Exception e) {
        return clazz.newInstance();
    }
}
```

`ClassInfo` 类的 `getInstance(clazz)` 如下。

```
public static ClassInfo getInstance(Class clazz) {
    if (cacheEnabled) {
        synchronized (clazz) {
            ClassInfo cached = (ClassInfo) CLASS_INFO_MAP.get(clazz);
            if (cached == null) {
                cached = new ClassInfo(clazz);
                CLASS_INFO_MAP.put(clazz, cached);
            }
            return cached;
        }
    } else {
        return new ClassInfo(clazz);
    }
}
```

上面代码主要是看实现是已经保持在缓存中，采用享元设计模式。关于其实现，在讲解 `ClassInfo` 类的时候进行过详细说明。

在 `ClassInfo` 类的 `instantiateClass()` 如下。

```
public Object instantiateClass() {
    // defaultConstructor 是缺省的构造方法
    if (defaultConstructor != null) {
        try {
            return defaultConstructor.newInstance(null);
        } catch (Exception e) {
            throw new RuntimeException("Error instantiating class. Cause: " + e, e);
        }
    } else {
        throw new RuntimeException("Error instantiating class. There is no default constructor for class " + className);
    }
}
```

12.2 Bean 管理

SQL Map 架构需要对 JavaBean 有深刻的理解。Bean 管理主要负责 JavaBean 类的处理，包括两个部分，一个部分是 ClassInfo 类为中新的基于 Bean 存储的内容，ClassInfo 类是每个 JavaBean 都要实现存储的类。还有一个部分是外部调用方法的内容。Probe 接口及其形成的实现类主要用于处理 bean、DOM 对象和其他的对象。

JavaBean 是一种特殊的 Java 类，它严格遵循 JavaBean 命名规范，定义存取类状态信息方法的命名规则。JavaBean 的属性由它的方法定义（而不是由字段定义）。以“set”为名称开始的方法是可写的属性，而以“get”为名称开始的方法是可读的属性。对于“boolean”类型的字段，可读的方法名称也可以用“is”开始。“set”方法不应拥有返回类型（即必须为 void），并且只能有一个参数，参数的数据类型必须和属性的数据类型一致。“get”方法应返回合适的类型并且不允许有参数。虽然通常并不强制，但“set”方法参数的数据类型和“get”方法的返回类型应一致。JavaBean 应实现 Serializable 接口，还支持其他的特性。

12.2.1 ClassInfo 类

Bean 管理主要是 ClassInfo 类。ClassInfo 类在实现管理 Bean 上有双重功能，第一种功能是充当缓存 JavaBean 的容器池，其意义是，当我们在配置文件中配置了多个 JavaBean，这些 JavaBean 最终都会实例化为对象并保持在 ClassInfo 类 MAP 数据类型的变量 CLASS_INFO_MAP 中。第二种功能是担任某个独立的 JavaBean 的对象实体容器，其意义是每个 JavaBean 进行实例化后，这些 JavaBean 的一些本身特性，如 JavaBean 名称、构造方法、属性、get 方法、set 方法等一些信息，全部都转化为一个 ClassInfo 实例化对象的各种属性。

从这个类的构造函数来看，实际上它主要是管理 JavaBean 类，把一个 JavaBean 类的构造方法、get 方法、set 方法和 Field 全部保存到当前的私有变量中，其类结构如图 12-1 所示。

ClassInfo 类依赖 Invoker 接口。MethodInvoker 类实现 Invoker 接口并关联 java.lang.reflect.Method 抽象类，主要用于处理 JavaBean 的方法内容和方法类型，是 Method 类型的包装类。GetFieldInvoker 类和 SetFieldInvoker 类实现 Invoker 接口并关联 java.lang.reflect.Field 抽象类，主要用于处理 JavaBean 的 Field 内容和方法类型，是 Field 类型的包装类。

同时，ClassInfo 类是一个比较有意思的类，不仅包括静态变量和静态方法，而且还可以自己进行实例化。由于这种错综复杂的结构，容易给读者造成混淆。所以，对 ClassInfo 类要一分为二地进行分析。事实上，该 ClassInfo 类可以编写成两个类，一个类是 ClassInfo 类的静态方法和处理类，另一个类是可以实例化的对象类。这样就比较清晰一些。

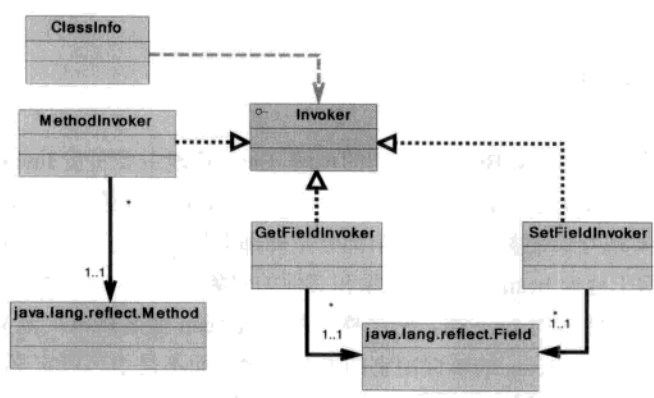


图 12-1 Bean 管理中 ClassInfo 类的相关类结构图

1. ClassInfo 类的静态信息和方法

ClassInfo 类的静态信息如表 12-2 所示。

表 12-2 ClassInfo 类的静态属性列表

属性名称	类 型	功能和用途	备 注
CacheEnabled	boolean	用于确定全局的 JavaBean 是否都要进行缓存	
EMPTY_STRING_ARRAY	String[]	空数组	
SIMPLE_TYPE_SET	Set	存储一些 Java 数据类型的 set	
CLASS_INFO_MAP	Map	存储已经针对 JavaBean 类的 ClassInfo 实例化对象	

ClassInfo 类的 Public 的静态方法如表 12-3 所示。

表 12-3 ClassInfo 类的静态方法列表

方法名称	参 数	返 回 值	功能和用途	备 注
isKnownType	Class	boolean	了解提交的类是否是一个通用类型	
getInstance	Class	ClassInfo	根据 ClassInfo 获得一个实例化对象	
unwrapThrowable	Throwable t	Throwable	解释一个 Throwable 对象并得到其根本原因	

其他内容都比较简单，主要是介绍静态变量 CLASS_INFO_MAP 变量和静态方法 getInstance。

在 ClassInfo 类中定义了如下一个静态变量。

```
private static final Map CLASS_INFO_MAP = Collections.synchronizedMap(new HashMap());
```

对于这个静态变量，CLASS_INFO_MAP 采用了线程安全性；即所有的线程处理在这里都要合成一个处理模式。其处理的静态方法代码如下。

```
public static ClassInfo getInstance(Class clazz) {
    if (cacheEnabled) {
        synchronized (clazz) {
```



```

ClassInfo cached = (ClassInfo) CLASS_INFO_MAP.get(clazz);
if (cached == null) {
    cached = new ClassInfo(clazz);
    CLASS_INFO_MAP.put(clazz, cached);
}
return cached;
}
} else {
    return new ClassInfo(clazz);
}
}

```

这样可以在 ClassInfo 类中通过静态方法来创建 ClassInfo 对象。CLASS_INFO_MAP 的内部数据类型如下, key 是 JavaBean 类, value 是基于 key 的这个 JavaBean 类的 ClassInfo 实例化对象。

如果在 CLASS_INFO_MAP 中有这个 JavaBean 的 ClassInfo 对象, 就获得这个 ClassInfo 对象; 如果没有这个 ClassInfo 对象, 则基于传入的 JavaBean 实例化一个 ClassInfo 对象。通过这种方式, 针对一个 JavaBean 只用实例化一次 ClassInfo 对象。

这里采用的设计模式是享元模式。所谓享元 (Flyweight) 模式, 就是运用共享技术有效地支持大量细粒度的对象。一般来说, 享元模式属于结构型设计模式。该模式以共享的方式高效地支持大量的细粒度对象, 能做到共享的关键是区分内蕴状态和外蕴状态。内蕴状态存储在享元内部, 不会随环境的改变而有所不同。外蕴状态是随环境的改变而改变的。外蕴状态不能影响内蕴状态, 它们是相互独立的。将可以共享的状态和不可以共享的状态从常规类中区分开来, 将不可以共享的状态从类里剔除出去。客户端不可以直接创建被共享的对象, 而应当使用一个工厂对象负责创建被共享的对象。享元模式可大幅度地降低内存中对象的数量。

Flyweight 结构如图 12-2 所示。其角色包括抽象享元 (Flyweight) 角色、具体享元 (Concrete Flyweight) 角色、复合享元 (Unsharable Flyweight) 角色和享元工厂 (Flyweight Factory) 角色等。下面简单介绍一下这些角色的含义。

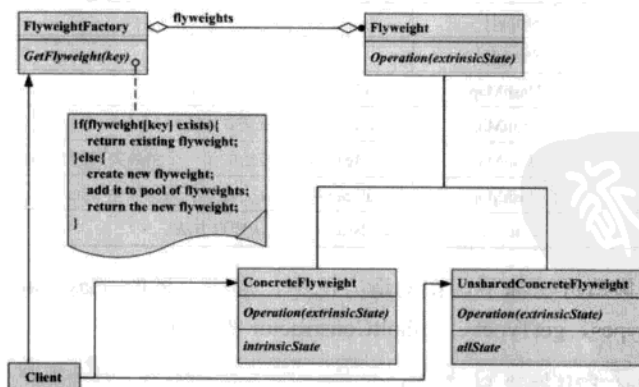


图 12-2 享元模式结构

① 抽象享元 (Flyweight) 角色：此角色是所有具体享元类的超类，为这些规定出需要实现的公共接口。那些需要外蕴状态 (External State) 的操作可以通过方法的参数传入。抽象享元的接口使得享元变得可能，但是并不强制子类实行共享，因此并非所有的享元对象都是可以共享的。

② 具体享元 (Concrete Flyweight) 角色：实现抽象享元角色所规定的接口。如果有内蕴状态的话，必须负责为内蕴状态提供存储空间。享元对象的内蕴状态必须与对象所处的周围环境无关，从而使得享元对象可以在系统内共享。有时候具体享元角色又叫做单纯具体享元角色，因为复合享元角色是由单纯具体享元角色通过复合而成的。

③ 复合享元 (Unsharable Flyweight) 角色：复合享元角色所代表的对象是不可以共享的，但是一个复合享元对象可以分解成多个本身是单纯享元对象的组合。复合享元角色又称为不可共享的享元对象。

④ 享元工厂 (Flyweight Factory) 角色：本角色负责创建和管理享元角色，且必须保证享元对象可以被系统适当地共享。当一个客户端对象请求一个享元对象的时候，享元工厂角色需要检查系统中是否已经有一个符合要求的享元对象，如果已经有了，享元工厂角色就应当提供这个已有的享元对象；如果系统中没有一个适当的享元对象，那么享元工厂角色就应当创建一个新的合适的享元对象。

⑤ 客户端 (Client) 角色：本角色还需要自行存储所有享元对象的外蕴状态。
针对上述共享模式的描述和具体的 ClassInfo 实现，我们来做一个对应。ClassInfo 类既是享元工厂 (Flyweight Factory) 角色，又是具体享元 (Concrete Flyweight) 角色。

2. ClassInfo 类实例化对象

ClassInfo 类的实例化对象信息如表 12-4 所示。

表 12-4 ClassInfo 类的实例化对象的属性列表

属性名称	类 型	功能和用途	备 注
className	String	JavaBean 的类名称	
readablePropertyNames	String[]		
writablePropertyNames	String[]		
setMethods	HashMap	JavaBean 类的 set 方法 Map	
getMethods	HashMap	JavaBean 类的 get 方法 Map	
setTypes	HashMap	JavaBean 类 set 方法的第一个参数类型的 Map	
getTypes	HashMap	JavaBean 类的 get 方法返回数据类型 Map	
defaultConstructor;	Constructor	JavaBean 类的默认构造方法	

ClassInfo 的构造方法如下，在构造方法中，形成上述的 className、setMethods、getMethods、setTypes、getTypes、defaultConstructor 等变量。

```
private ClassInfo(Class clazz) {
    className = clazz.getName();
    addDefaultConstructor(clazz);
}
```

```

        addGetMethods(clazz);
        addSetMethods(clazz);
        addFields(clazz);
        readablePropertyNames = (String[]) getMethods.keySet().toArray(new String
[getMethods.keySet().size()]);
        writeablePropertyNames = (String[]) setMethods.keySet().toArray(new String
[setMethods.keySet().size()]);
    }

```

可能读者会感觉到奇怪, 这个 `ClassInfo` 类的构造函数居然是私有构造方法。其实, 这就是为了不让这个类在外部构造, 只能在 `ClassInfo` 类中的静态方法中构造。代码参见 `ClassInfo` 类静态方法 `getInstance`。

针对 `defaultConstructor` 变量的赋值情况, 代码如下。

```

private void addDefaultConstructor(Class clazz) {
    Constructor[] consts = clazz.getDeclaredConstructors();
    for (int i = 0; i < consts.length; i++) {
        Constructor constructor = consts[i];
        if (constructor.getParameterTypes().length == 0) {
            if (canAccessPrivateMethods()) {
                try {
                    constructor.setAccessible(true);
                } catch (Exception e) {
                    // Ignored. This is only a final precaution, nothing we can do.
                }
            }
            if (constructor.isAccessible()) {
                this.defaultConstructor = constructor;
            }
        }
    }
}

```

针对 `getMethods` 变量的赋值代码如下。

```

private void addGetMethods(Class cls) {
    Method[] methods = getClassMethods(cls);
    for (int i = 0; i < methods.length; i++) {
        Method method = methods[i];
        String name = method.getName();
        if (name.startsWith("get") && name.length() > 3) {
            if (method.getParameterTypes().length == 0) {
                name = dropCase(name);
                addGetMethod(name, method);
            }
        } else if (name.startsWith("is") && name.length() > 2) {
            if (method.getParameterTypes().length == 0) {
                name = dropCase(name);
                addGetMethod(name, method);
            }
        }
    }
}

```

```

    }
    }
    }

    private void addGetMethod(String name, Method method) {
        getMethods.put(name, new MethodInvoker(method));
        getTypes.put(name, method.getReturnType());
    }

```

Map 变量 `getMethods` 的结构是: key 是 JavaBean 的属性名称, value 是基于 `get` 该属性的 Method 方法类的 `MethodInvoker` 实例化对象。

Map 变量 `getTypes` 的结构是: key 是 JavaBean 的属性名称, value 是基于 `get` 该属性的 Method 方法的返回值类型。

针对 `setMethods` 变量的赋值代码如下。

```

private void addSetMethods(Class cls) {
    Map conflictingSetters = new HashMap();
    Method[] methods = getClassMethods(cls);
    for (int i = 0; i < methods.length; i++) {
        Method method = methods[i];
        String name = method.getName();
        if (name.startsWith("set") && name.length() > 3) {
            if (method.getParameterTypes().length == 1) {
                name = dropCase(name);
                ///-----
                addSetterConflict(conflictingSetters, name, method);
                // addSetMethod(name, method);
                ///-----
            }
        }
    }
    resolveSetterConflicts(conflictingSetters);
}

private void addSetMethod(String name, Method method) {
    setMethods.put(name, new MethodInvoker(method));
    setTypes.put(name, method.getParameterTypes()[0]);
}

```

由上述代码, 可知 Map 变量 `setMethods` 的结构如下: key 是 JavaBean 的属性名称, value 是基于 `set` 该属性的 Method 方法类的 `MethodInvoker` 实例化对象。

Map 变量 `setTypes` 的结构如下: key 是 JavaBean 的属性名称, value 是基于 `set` 该属性的 Method 方法的第一个参数类型。

`addFields(clazz)` 方法的内容如下。

```

private void addFields(Class clazz) {
    Field[] fields = clazz.getDeclaredFields();

```

```

for (int i = 0; i < fields.length; i++) {
    Field field = fields[i];
    if (canAccessPrivateMethods()) {
        try {
            field.setAccessible(true);
        } catch (Exception e) {
            // Ignored. This is only a final precaution, nothing we can do.
        }
    }
    if (field.isAccessible()) {
        if (!setMethods.containsKey(field.getName())) {
            addSetField(field);
        }
        if (!getMethods.containsKey(field.getName())) {
            addGetField(field);
        }
    }
}
if (clazz.getSuperclass() != null) {
    addFields(clazz.getSuperclass());
}
}

```

遍历类的所有定义的 Field, 然后对这些 Field 进行处理, 包括调用 `addSetField(field)` 方法和 `addGetField(field)` 方法来处理。最后通过调用 `addFields(clazz.getSuperclass())` 来实现对类中的内部类的递归处理。简单地浏览一下 `addSetField(field)` 方法和 `addGetField(field)` 方法。

```

private void addSetField(Field field) {
    setMethods.put(field.getName(), new SetFieldInvoker(field));
    setTypes.put(field.getName(), field.getType());
}

private void addGetField(Field field) {
    getMethods.put(field.getName(), new GetFieldInvoker(field));
    getTypes.put(field.getName(), field.getType());
}

```

12.2.2 Probe 接口及其实现

Probe 接口及其形成的实现类主要用于处理 Bean、DOM 对象和其他对象的转化, 实际上也是针对 `ClassInfo` 类的处理工具, 或者说是 `ClassInfo` 类的解释器。对 `ClassInfo` 类处理的方法都是通过 Probe 接口对 `ClassInfo` 对象进行操作的, 包括赋值或者获取属性、属性值等内容, 其类结构如图 12-3 所示, 其设计模式是基于简单工厂模式。

ProbeFactory 类三次关联 Probe 接口, 实际上形成了 DomProbe 对象、ComplexBeanProbe 对象和 GenericProbe 对象。BaseProbe 是一个抽象类, 其实现 Probe 接口, 然后由 DomProbe

类、ComplexBeanProbe 类和 GenericProbe 类来继承。

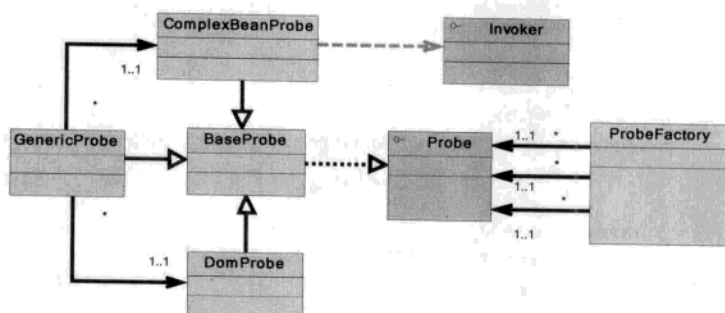


图 12-3 Bean 管理中 Probe 接口及其类结构图

1. ProbeFactory 类的使用

ProbeFactory 类采用简单工厂模式，而且加载模式为饿汉式加载。ProbeFactory 采用简单工厂模式，可以获得三种类型的 Probe，分别是 DomProbe 对象、ComplexBeanProbe 对象和 GenericProbe 对象。属性就把三种类型进行了静态实例化对象，形成了静态变量 DomProbe 对象、ComplexBeanProbe 对象和 GenericProbe 对象。

工厂方法有两个，一个是不带参数的创建 Probe 对象模式，代码如下：

```
private static final Probe GENERIC = new GenericProbe();
public static Probe getProbe() {
    return GENERIC;
}
```

默认采用的是 GenericProbe 实例化对象。另一个是带有参数，则根据要求处理的对象类型来获得返回 Probe 实例化对象。

```
private static final Probe DOM = new DomProbe();
private static final Probe BEAN = new ComplexBeanProbe();

public static Probe getProbe(Object object) {
    if (object instanceof org.w3c.dom.Document) {
        return DOM;
    } else {
        return BEAN;
    }
}
```

如果是 Document 对象，则返回 DomProbe 实例化对象。如果是 Map、Bean 等其他对象，则返回 ComplexBeanProbe 实例化对象。

2. Probe 的接口方法

Probe 接口主要是调用六个实现方法，其说明见表 12-5 所示。

表 12-5 Probe 接口的方法列表

方法名称	参 数	返 回 值	功能和用途
getObject	Object object, String name	Object	该方法获得一个 JavaBean 的属性值
setObject	Object object, String name, Object value	无	该方法设置一个 JavaBean 的属性值
getPropertyTypeForSetter	Object object, String name	Class	该方法获得一个 JavaBean 的属性的 set 方法的参数类型
getPropertyTypeForGetter	Object object, String name	Class	该方法获得一个 JavaBean 的属性的 get 方法的返回值类型
hasWritableProperty	Object object, String propertyName	boolean	该方法判断 JavaBean 对象的某个属性是否可写
hasReadableProperty	Object object, String propertyName	boolean	该方法判断 JavaBean 对象的某个属性是否可读

对这六个方法的说明：

```
public Object getObject(Object object, String name)
```

该方法获得一个 JavaBean 的属性值。其中第一个参数表示一个 JavaBean 对象，第二个参数表示属性名称。

```
public void setObject(Object object, String name, Object value);
```

该方法设置一个 JavaBean 的属性值。其中第一个参数表示一个 JavaBean 对象，第二个参数表示属性名称，第三个参数是设置 JavaBean 的内容。

```
public Class getPropertyTypeForSetter(Object object, String name)
```

该方法获得一个 JavaBean 的属性的 set 方法的参数类型。其中第一个参数表示一个 JavaBean 对象，第二个参数表示属性名称。

```
public Class getPropertyTypeForGetter(Object object, String name)
```

该方法获得一个 JavaBean 的属性的 get 方法的返回值类型。其中第一个参数表示一个 JavaBean 对象，第二个参数表示属性名称。

```
public boolean hasWritableProperty(Object object, String propertyName)
```

该方法判断 Javabeen 对象的某个属性是否可写。其中第一个参数表示一个 Javabeen 对象，第二个参数表示属性名称。

```
public boolean hasReadableProperty(Object object, String propertyName);
```

该方法判断 JavaBean 对象的某个属性是否可读。其中第一个参数表示一个 JavaBean 对象，第二个参数表示属性名称。

ComplexBeanProbe 类主要是处理 Bean 对象或其他标准对象（如 Map、List）的属性和方法。DomProbe 类主要是处理 Dom 对象。GenericProbe 类是集成 ComplexBeanProbe

类和 DomProbe 类的实现方式。下面分别介绍这三个类的实现方式。

3. DomProbe 类的实现

DomProbe 类主要处理 Dom 对象。我们还是来看看 DomProbe 类是如何实现这六个方法的。

(1) DomProbe 类的 getObject 方法实现

DomProbe 类的 Object getObject(Object object, String name)方法如下。

```
public Object getObject(Object object, String name) {
    Object value = null;
    Element element = findNestedNodeByName(resolveElement(object), name, false);
    if (element != null) {
        value = getElementValue(element);
    }
    return value;
}
```

先调用 resolveElement(Object)方法来获得 Element 对象。

```
private Element resolveElement(Object object) {
    Element element = null;
    if (object instanceof Document) {
        element = (Element) ((Document) object).getLastChild();
    } else if (object instanceof Element) {
        element = (Element) object;
    } else {
        throw new ProbeException(".....");
    }
    return element;
}
```

上述代码说明，程序是根据传递进来的参数来决定返回 Element 对象变量的。当传递对象为 Element 对象，那就通过转化为 Element 类型并直接返回该对象。如果传递的对象是 Document 对象，则返回该 Document 的根节点 Element 对象。

当程序获得了 Element 对象后，调用 findNestedNodeByName(resolveElement(Object), name, false)来获得一个需要的 Element 对象。findNestedNodeByName 代码如下。

```
private Element findNestedNodeByName(Element element, String name, boolean create) {
    Element child = element;
    StringTokenizer parser = new StringTokenizer(name, ".", false);
    while (parser.hasMoreTokens()) {
        String childName = parser.nextToken();
        if (childName.indexOf('.') > -1) {
            String propName = childName.substring(0, childName.indexOf('.'));
            int i = Integer.parseInt(childName.substring(childName.indexOf('.') + 1, childName.indexOf('.')));
            child = findNodeByName(child, propName, i, create);
        } else {

```

```

        child = findNodeByName(child, childName, 0, create);
    }
    if (child == null) { break; }
}
return child;
}

```

该程序首先是要处理名称中带有的“.”内容，通过字符串分隔解析来进行循环处理。在获取 child 的过程中，调用了 findNodeByName 方法，代码如下：

```

private Element findNodeByName(Element element, String name, int index, boolean
create) {
    Element prop = null;

    // Find named property element
    NodeList propNodes = element.getElementsByTagName(name);
    if (propNodes.getLength() > index) {
        prop = (Element) propNodes.item(index);
    } else {
        if (create) {
            for (int i = 0; i < index + 1; i++) {
                prop = element.getOwnerDocument().createElement(name);
                element.appendChild(prop);
            }
        }
    }
    return prop;
}

```

上述代码实现的功能，要达到的目标，根据元素名称获得 Node 的 element 节点并返回。其中代码中关于循环处理的内容并没有执行。

最后调用 getElementValue(element)，获得当前 Element 对象的值，代码如下：

```

private Object getElementValue(Element element) {
    StringBuffer value = null;

    Element prop = element;

    if (prop != null) {
        // Find text child elements
        NodeList texts = prop.getChildNodes();
        if (texts.getLength() > 0) {
            value = new StringBuffer();
            for (int i = 0; i < texts.getLength(); i++) {
                Node text = texts.item(i);
                if (text instanceof CharacterData) {
                    value.append(((CharacterData) text).getData());
                }
            }
        }
    }
}

```

```

    }

    //convert to proper type
    //value = convert(value.toString());

    if (value == null) {
        return null;
    } else {
        return String.valueOf(value);
    }
}

```

(2) DomProbe 类的 setObject 方法实现

DomProbe 类的 setObject 方法代码如下。

```

public void setObject(Object object, String name, Object value) {
    Element element = findNestedNodeByName(resolveElement(object), name, true);
    if (element != null) {
        setElementValue(element, value);
    }
}

```

调用了 setElementValue 方法，代码如下。

```

private void setElementValue(Element element, Object value) {
    CharacterData data = null;
    Element prop = element;

    if (value instanceof Collection) {
        Iterator items = ((Collection) value).iterator();
        while (items.hasNext()) {
            Document valdoc = (Document) items.next();
            NodeList list = valdoc.getChildNodes();
            for (int i = 0; i < list.getLength(); i++) {
                Node newNode = element.getOwnerDocument().importNode(list.item(i), true);
                element.appendChild(newNode);
            }
        }
    } else if (value instanceof Document) {
        Document valdoc = (Document) value;
        Node lastChild = valdoc.getLastChild();
        NodeList list = lastChild.getChildNodes();
        for (int i = 0; i < list.getLength(); i++) {
            Node newNode = element.getOwnerDocument().importNode(list.item(i), true);
            element.appendChild(newNode);
        }
    } else if (value instanceof Element) {
        Node newNode = element.getOwnerDocument().importNode((Element) value, true);
        element.appendChild(newNode);
    } else {
        // Find text child element
    }
}

```

```

NodeList texts = prop.getChildNodes();
if (texts.getLength() == 1) {
    Node child = texts.item(0);
    if (child instanceof CharacterData) {
        // Use existing text.
        data = (CharacterData) child;
    } else {
        // Remove non-text, add text.
        prop.removeChild(child);
        Text text = prop.getOwnerDocument().createTextNode(String.valueOf(value));
        prop.appendChild(text);
        data = text;
    }
} else if (texts.getLength() > 1) {
    // Remove all, add text.
    for (int i = texts.getLength() - 1; i >= 0; i--) {
        prop.removeChild(texts.item(i));
    }
    Text text = prop.getOwnerDocument().createTextNode(String.valueOf(value));
    prop.appendChild(text);
    data = text;
} else {
    // Add text.
    Text text = prop.getOwnerDocument().createTextNode(String.valueOf(value));
    prop.appendChild(text);
    data = text;
}
data.setData(String.valueOf(value));
}
}

```

按照要赋值的类型进行分类处理，赋值有三种类型：分别是 Collection、其次是 Document 对象、再次是 Element 对象。

对于 Collection 对象的处理就是形成两次循环。首循环是以 Collection 列表为基础，处理单个 Collection，即 Document 对象；第二个循环是以 Document 对象中的 Element 为基础，处理单个的 Element 元素。在要赋值 Element 节点下面增加子节点，然后一个个地赋值。

对于 Document 对象的处理就是形成一次循环，循环是以 Document 对象中的 Element 为基础、处理单个的 Element 元素。在要赋值 Element 节点下面增加子节点，然后一个个地赋值。

对于 Element 元素的处理就是直接进行赋值操作。

(3) DomProbe 类的 getPropertyTypeForSetter 方法实现
DomProbe 类的 getPropertyTypeForSetter 方法代码如下。

```

public Class getPropertyTypeForSetter(Object object, String name) {
    Element e = findNestedNodeByName(resolveElement(object), name, false);
    //todo alias types, don't use exceptions like this
    try {

```

```

        return Resources.classForName(e.getAttribute("type"));
    } catch (ClassNotFoundException el) {
        return Object.class;
    }
}

```

根据 `findNestedNodeByName` 方法获得 `Element` 对象，然后得到 `Element` 对象的“type”属性内容，再然后通过 `Resources` 工具生成类对象并返回。

(4) `DomProbe` 类的 `getPropertyTypeForGetter` 方法实现
`DomProbe` 类的 `getPropertyTypeForGetter` 方法代码如下。

```

public Class getPropertyTypeForGetter(Object object, String name) {
    Element e = findNestedNodeByName(resolveElement(object), name, false);
    //todo alias types, don't use exceptions like this
    try {
        return Resources.classForName(e.getAttribute("type"));
    } catch (ClassNotFoundException el) {
        return Object.class;
    }
}

```

根据 `findNestedNodeByName` 方法获得 `Element` 对象，然后得到 `Element` 对象的“type”属性内容，再然后通过 `Resources` 工具生成类对象并返回。

(5) `DomProbe` 类的 `hasWritableProperty` 方法实现
`DomProbe` 类的 `hasWritableProperty` 方法代码如下。

```

public boolean hasWritableProperty(Object object, String propertyName) {
    return findNestedNodeByName(resolveElement(object), propertyName, false) != null;
}

```

根据 `findNestedNodeByName` 方法来判断是否能获得一个 `Element` 对象。

(6) `DomProbe` 类的 `hasReadableProperty` 方法实现
`DomProbe` 类的 `hasReadableProperty` 方法代码如下。

```

public boolean hasReadableProperty(Object object, String propertyName) {
    return findNestedNodeByName(resolveElement(object), propertyName, false) != null;
}

```

根据 `findNestedNodeByName` 方法来判断是否能获得一个 `Element` 对象。

4. `ComplexBeanProbe` 类的实现

(1) `ComplexBeanProbe` 类的 `getObject` 方法实现
`ComplexBeanProbe` 类的 `getObject` 方法代码如下。

```

public Object getObject(Object object, String name) {
    if (name.indexOf('.') > -1) {
        StringTokenizer parser = new StringTokenizer(name, ".");
        Object value = object;
    }
}

```

```

while (parser.hasMoreTokens()) {
    value = getProperty(value, parser.nextToken());
    if (value == null) {break; }
}
return value;
} else {
    return getProperty(object, name);
}
}

```

调用 `getProperty(Object, name)` 方法的代码如下:

```

protected Object getProperty(Object object, String name) {
    try {
        Object value = null;
        if (name.indexOf('[') > -1) {
            value = getIndexedProperty(object, name);
        } else {
            if (object instanceof Map) {
                int index = name.indexOf('.');
                if (index > -1) {
                    String mapId = name.substring(0, index);
                    value = getProperty(((Map) object).get(mapId), name.substring(index + 1));
                } else {
                    value = ((Map) object).get(name);
                }
            } else {
                int index = name.indexOf('.');
                if (index > -1) {
                    String newName = name.substring(0, index);
                    value = getProperty(getObject(object, newName), name.substring(index + 1));
                } else {
                    ClassInfo classCache = ClassInfo.getInstance(object.getClass());
                    Invoker method = classCache.getGetInvoker(name);
                    if (method == null) {
                        throw new NoSuchMethodException("...")
                    }
                    try {
                        value = method.invoke(object, NO_ARGUMENTS);
                    } catch (Throwable t) {
                        throw ClassInfo.unwrapThrowable(t);
                    }
                }
            }
        }
        return value;
    } catch (ProbeException e) { throw e; }
    } catch (Throwable t) {
        if (object == null) {
            throw new ProbeException("Could not get property '" + name + "' from null reference. Cause: " + t.toString(), t);
        }
    }
}

```



```

    } else {
        throw new ProbeException("....." + t.toString(), t);
    }
}
}

```

按照要取值的类型进行分类处理, 赋值有两种类型, 一种是 Map, 还有一种是 Bean。如果是 Map, 就按照 key 去获得 Map 的 value。如果是 Bean, 通过反射机制去获取相关属性的值。

(2) ComplexBeanProbe 类的 setObject 方法实现

ComplexBeanProbe 类的 setObject 方法代码如下。

```

public void setObject(Object object, String name, Object value) {
    if (name.indexOf('.') > -1) {
        StringTokenizer parser = new StringTokenizer(name, ".");
        String property = parser.nextToken();
        Object child = object;
        while (parser.hasMoreTokens()) {
            Class type = getPropertyTypeForSetter(child, property);
            Object parent = child;
            child = getProperty(parent, property);
            if (child == null) {
                if (value == null) {
                    return; // don't instantiate child path if value is null
                } else {
                    try {
                        child = ResultObjectFactoryUtil.createObjectThroughFactory(type);
                        setObject(parent, property, child);
                    } catch (Exception e) {
                        throw new ProbeException("....." + e.toString(), e);
                    }
                }
            }
            property = parser.nextToken();
        }
        setProperty(child, property, value);
    } else {
        setProperty(object, name, value);
    }
}

```

该程序首先要处理名称中带有的“.”内容, 通过字符串分隔解析来进行循环处理。然后调用 setProperty(Object, name, value)方法, 其程序代码如下:

```

protected void setProperty(Object object, String name, Object value) {
    ClassInfo classCache = ClassInfo.getInstance(object.getClass());
    try {
        if (name.indexOf('[') > -1) {
            setIndexedProperty(object, name, value);
        }
    }
}

```



```

    } else {
        if (object instanceof Map) {
            ((Map) object).put(name, value);
        } else {
            Invoker method = classCache.getSetInvoker(name);
            if (method == null) {
                throw new NoSuchMethodException(".....");
            }
            Object[] params = new Object[1];
            params[0] = value;
            try {
                method.invoke(object, params);
            } catch (Throwable t) {
                throw ClassInfo.unwrapThrowable(t);
            }
        }
    }
} catch (ProbeException e) {
    throw e;
} catch (Throwable t) {
    if (object == null) {
        throw new ProbeException(".....", t);
    } else {
        throw new ProbeException(".....", t);
    }
}
}
}

```

按照要赋值的类型进行分类处理, 赋值有两种类型, 一种是 Map, 还有一种是 Bean。如果是 Map, 就按照 key 去给 Map 赋值。如果是 Bean, 通过反射机制去给 Bean 赋值。

(3) ComplexBeanProbe 类的 getPropertyTypeForSetter 方法实现

ComplexBeanProbe 类的 getPropertyTypeForSetter 方法代码如下。

```

public Class getPropertyTypeForSetter(Object object, String name) {
    Class type = object.getClass();

    if (object instanceof Class) {
        type = getClassPropertyTypeForSetter((Class) object, name);
    } else if (object instanceof Map) {
        Map map = (Map) object;
        Object value = map.get(name);
        if (value == null) {
            type = Object.class;
        } else {
            type = value.getClass();
        }
    } else {
        if (name.indexOf('.') > -1) {
            StringTokenizer parser = new StringTokenizer(name, ".");
            while (parser.hasMoreTokens()) {

```

```

        name = parser.nextToken();
        type = ClassInfo.getInstance(type).getSetterType(name);
    }
    } else {
        type = ClassInfo.getInstance(type).getSetterType(name);
    }
}

return type;
}

```

按照要取值的类型进行分类处理，赋值有两种类型，一种是 Map，还有一种是 Bean。如果是 Map，就按照 key 去给 Map 赋值。如果是 Bean，通过反射机制去给 Bean 赋值。

(4) ComplexBeanProbe 类的 getPropertyTypeForGetter 方法实现

ComplexBeanProbe 类的 getPropertyTypeForGetter 方法代码如下。

```

public Class getPropertyTypeForGetter(Object object, String name) {
    Class type = object.getClass();

    if (object instanceof Class) {
        type = getClassPropertyTypeForGetter((Class) object, name);
    } else if (object instanceof Map) {
        Map map = (Map) object;
        Object value = map.get(name);
        if (value == null) {
            type = Object.class;
        } else {
            type = value.getClass();
        }
    } else {
        if (name.indexOf('.') > -1) {
            StringTokenizer parser = new StringTokenizer(name, ".");
            while (parser.hasMoreTokens()) {
                name = parser.nextToken();
                type = ClassInfo.getInstance(type).getGetterType(name);
            }
        } else {
            type = ClassInfo.getInstance(type).getGetterType(name);
        }
    }

    return type;
}

```

按照要获得 Getter 类型进行分类处理，一类是 Class，另一类是 Map，还有就是其他类。如果是 Class，就调用 getClassPropertyTypeForGetter(Class type, String name)来活动区 Class，代码如下。

```

private Class getClassPropertyTypeForGetter(Class type, String name) {

```

```

    if (name.indexOf('.') > -1) {
        StringTokenizer parser = new StringTokenizer(name, ".");
        while (parser.hasMoreTokens()) {
            name = parser.nextToken();
            type = ClassInfo.getInstance(type).getGetterType(name);
        }
    } else {
        type = ClassInfo.getInstance(type).getGetterType(name);
    }

    return type;
}

```

该程序首先要处理名称中带有的“.”内容，通过字符串分隔解析来进行循环处理。通过 ClassInfo 的静态方法 getInstance 方法获得一个实例化的 ClassInfo 对象，再调用 ClassInfo 实例化对象的 getGetterType 方法来获得 Class。

如果是 Map，就按照 key 获得 Map 的 value。然后返回 value 的 Class 类型。

如果是其他类型，处理基本与 Class 模式相同。该程序还是先要处理名称中带有的“.”内容，通过字符串分隔解析来进行循环处理。通过 ClassInfo 的静态方法 getInstance 方法获得一个实例化的 ClassInfo 对象，再调用 ClassInfo 实例化对象的 getGetterType 方法来获得 Class。

(5) ComplexBeanProbe 类的 hasWritableProperty 方法实现

ComplexBeanProbe 类的 hasWritableProperty 方法代码如下。

```

public boolean hasWritableProperty(Object object, String propertyName) {
    boolean hasProperty = false;
    if (object instanceof Map) {
        hasProperty = true;
    } else {
        if (propertyName.indexOf('.') > -1) {
            StringTokenizer parser = new StringTokenizer(propertyName, ".");
            Class type = object.getClass();
            while (parser.hasMoreTokens()) {
                propertyName = parser.nextToken();
                type = ClassInfo.getInstance(type).getGetterType(propertyName);
                hasProperty = ClassInfo.getInstance(type).hasWritableProperty(propertyName);
            }
        } else {
            hasProperty = ClassInfo.getInstance(object.getClass()).hasWritableProperty(propertyName);
        }
    }
    return hasProperty;
}

```

按照要处理的 Object 分为两类处理，一类是 Map，另一类是其他对象类型。如果是 Map 对象，直接放回 True。如果是其他对象类型，该程序要先处理名称中带有的“.”内

容，通过字符串分隔解析来进行循环处理。通过 `ClassInfo` 的静态方法 `getInstance` 方法获得一个实例化的 `ClassInfo` 对象，再调用 `ClassInfo` 实例化对象的 `hasWritableProperty` 方法来判断 `True` 或 `false`。

(6) `ComplexBeanProbe` 类的 `hasReadableProperty` 方法实现
`ComplexBeanProbe` 类的 `hasReadableProperty` 方法代码如下。

```
public boolean hasReadableProperty(Object object, String propertyName) {
    boolean hasProperty = false;
    if (object instanceof Map) {
        hasProperty = true; // ((Map) object).containsKey(propertyName);
    } else {
        if (propertyName.indexOf('.') > -1) {
            StringTokenizer parser = new StringTokenizer(propertyName, ".");
            Class type = object.getClass();
            while (parser.hasMoreTokens()) {
                propertyName = parser.nextToken();
                type = ClassInfo.getInstance(type).getGetterType(propertyName);
                hasProperty = ClassInfo.getInstance(type).hasReadableProperty(propertyName);
            }
        } else {
            hasProperty = ClassInfo.getInstance(object.getClass()).hasReadableProperty(propertyName);
        }
    }
    return hasProperty;
}
```

按照要处理的 `Object` 分为两类处理，一类是 `Map`，另一类是其他对象类型。如果是 `Map` 对象，直接放回 `True`。如果是其他对象类型，该程序要先处理名称中带有的“.”内容，通过字符串分隔解析来进行循环处理。通过 `ClassInfo` 的静态方法 `getInstance` 方法获得一个实例化的 `ClassInfo` 对象，再调用 `ClassInfo` 实例化对象的 `hasWritableProperty` 方法来判断 `true` 或 `false`。

5. `GenericProbe` 类的实现

(1) `GenericProbe` 类的 `getObject` 方法实现
`GenericProbe` 类的 `getObject` 方法代码如下。

```
public Object getObject(Object object, String name) {
    if (object instanceof org.w3c.dom.Document) {
        return DOM_PROBE.getObject(object, name);
    } else if (object instanceof List) {
        return BEAN_PROBE.getIndexProperty(object, name);
    } else if (object instanceof Object[]) {
        return BEAN_PROBE.getIndexProperty(object, name);
    } else if (object instanceof char[]) {
        return BEAN_PROBE.getIndexProperty(object, name);
    }
}
```

```

    } else if (object instanceof boolean[]) {
        return BEAN_PROBE.getIndexProperty(object, name);
    } else if (object instanceof byte[]) {
        return BEAN_PROBE.getIndexProperty(object, name);
    } else if (object instanceof double[]) {
        return BEAN_PROBE.getIndexProperty(object, name);
    } else if (object instanceof float[]) {
        return BEAN_PROBE.getIndexProperty(object, name);
    } else if (object instanceof int[]) {
        return BEAN_PROBE.getIndexProperty(object, name);
    } else if (object instanceof long[]) {
        return BEAN_PROBE.getIndexProperty(object, name);
    } else if (object instanceof short[]) {
        return BEAN_PROBE.getIndexProperty(object, name);
    } else {
        return BEAN_PROBE.getObject(object, name);
    }
}

```

根据来的对象类型，转化给相应的 DomProbe 对象或 ComplexBeanProbe 对象去处理。

(2) GenericProbe 类的 setObject 方法实现

GenericProbe 类的 setObject 方法代码如下。

```

public void setObject(Object object, String name, Object value) {
    if (object instanceof org.w3c.dom.Document) {
        DOM_PROBE.setObject(object, name, value);
    } else {
        BEAN_PROBE.setObject(object, name, value);
    }
}

```

根据来的对象类型，转化给相应的 DomProbe 对象或 ComplexBeanProbe 对象去处理。

(3) GenericProbe 类的 getPropertyTypeForSetter 方法实现

GenericProbe 类的 getPropertyTypeForSetter 方法代码如下。

```

public Class getPropertyTypeForSetter(Object object, String name) {
    if (object instanceof Class) {
        return getClassPropertyTypeForSetter((Class) object, name);
    } else if (object instanceof org.w3c.dom.Document) {
        return DOM_PROBE.getPropertyTypeForSetter(object, name);
    } else {
        return BEAN_PROBE.getPropertyTypeForSetter(object, name);
    }
}

```

根据来的对象类型，转化给相应的 DomProbe 对象或 ComplexBeanProbe 对象去处理。

(4) GenericProbe 类的 getPropertyTypeForGetter 方法实现

GenericProbe 类的 getPropertyTypeForGetter 方法代码如下。

```

public Class getPropertyTypeForGetter(Object object, String name) {

```

```

    if (object instanceof Class) {
        return getClassPropertyTypeForGetter((Class) object, name);
    } else if (object instanceof org.w3c.dom.Document) {
        return DOM_PROBE.getPropertyTypeForGetter(object, name);
    } else if (name.indexOf('.') > -1) {
        return BEAN_PROBE.getIndexedType(object, name);
    } else {
        return BEAN_PROBE.getPropertyTypeForGetter(object, name);
    }
}

```

根据来的对象类型，转化给相应的 DomProbe 对象或 ComplexBeanProbe 对象去处理。

(5) GenericProbe 类的 hasWritableProperty 方法实现

GenericProbe 类的 hasWritableProperty 方法代码如下。

```

public boolean hasWritableProperty(Object object, String propertyName) {
    if (object instanceof org.w3c.dom.Document) {
        return DOM_PROBE.hasWritableProperty(object, propertyName);
    } else {
        return BEAN_PROBE.hasWritableProperty(object, propertyName);
    }
}

```

根据来的对象类型，转化给相应的 DomProbe 对象或 ComplexBeanProbe 对象去处理。

(6) GenericProbe 类的 hasReadableProperty 方法实现

GenericProbe 类的 hasReadableProperty 方法代码如下。

```

public boolean hasReadableProperty(Object object, String propertyName) {
    if (object instanceof org.w3c.dom.Document) {
        return DOM_PROBE.hasReadableProperty(object, propertyName);
    } else {
        return BEAN_PROBE.hasReadableProperty(object, propertyName);
    }
}

```

根据是 Document 类型还是 Bean 类型，转移给不同的实例化对象去处理。

12.3 Log 管理

iBATIS 框架使用 Jakarta Commons Logging (JCL) 记录日志信息。JCL 使用独立于具体实现的方式提供日志服务。可以“plug-in”包括 Log4J 和 JDK1.4 Logging API 等不同的日志服务实现。

iBATIS 框架的 Log 管理主要是在包 com.ibatis.common.logging 内。其日志有四种模式，一种是 Jakarta Commons Logging (JCL) 记录日志信息。第二种是 Log4J，第三种是 JDK1.4 Logging API，还有一种就是无日志模式。由 Jakarta Commons Logging (JCL) 使用独立于

具体实现的方式提供日志服务。

iBATIS 采用了简单工厂模式对日志做了一定的封装，其类结构如图 12-4 所示。

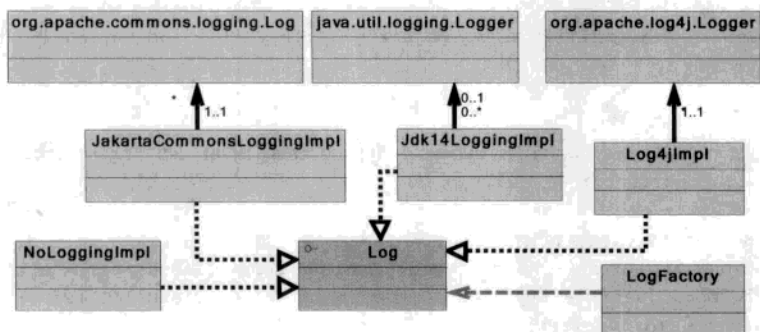


图 12-4 iBATIS 平台的 Log 类结构图

LogFactory 是日志的工厂类，其依赖 Log 接口。JakartaCommonsLoggingImpl、Jdk14LoggingImpl、Log4jImpl 和 NoLoggingImpl 都实现 Log 接口。但是这几种类实现日志的渊源都不同，JakartaCommonsLoggingImpl 类关联 org.apache.commons.logging.Log 接口，是 org.apache.commons.logging 实现的包装类。Log4jImpl 关联 org.apache.log4j.Logger 接口，是 org.apache.log4j 实现的包装类。Jdk14LoggingImpl 关联 Java.util.logging.Logger，是 JDK 日志实现的包装类。

外部调用日志非常简单，在需要调用的 Class 中加入如下代码。

```
import com.ibatis.common.logging.Log;
import com.ibatis.common.logging.LogFactory;

public class CommonLog {
    private static final Log log = LogFactory.getLog(CommonLog.class);
    // ...
    public static void main(String[] args) {
        log.error("ERROR");
        log.debug("DEBUG");
        log.warn("WARN");
        System.out.println(log.getClass());
    }
}
```

LogFactory 启动的时候加载 Jakarta Commons Logging，虽然后面还有几个 tryImplementation 方法，但是都不执行了。因为已经加载了 com.ibatis.common.logging.jakarta.JakartaCommonsLoggingImpl 类。变量 logConstructor 已经存在了，LogFactory 启动方法的代码如下。

```
static {
    tryImplementation("org.apache.commons.logging.LogFactory", "com.ibatis.common.
logging.jakarta.JakartaCommonsLoggingImpl");
}
```



```

        tryImplementation("org.apache.log4j.Logger", "com.ibatis.common.logging.log4j.
Log4jImpl");
        tryImplementation("java.util.logging.Logger", "com.ibatis.common.logging.jdk14.
Jdk14LoggingImpl");
        tryImplementation("java.lang.Object", "com.ibatis.common.logging.nologging.
NoLoggingImpl");
    }

    private static void tryImplementation(String testClassName, String implClassName) {
        if (logConstructor == null) {
            try {
                Resources.classForName(testClassName);
                Class implClass = Resources.classForName(implClassName);
                logConstructor = implClass.getConstructor(new Class[] {Class.class});
            } catch (Throwable t) {
            }
        }
    }
}

```

当调用日志的时候，用 `JakartaCommonsLoggingImpl` 类的构造函数实例化 `JakartaCommonsLoggingImpl` 对象，（代码如下）。

```

public static Log getLog(Class aClass) {
    try {
        return (Log)logConstructor.newInstance(new Object[] {aClass});
    } catch (Throwable t) {
        throw new RuntimeException("Error creating logger for class " + aClass + ".
Cause: " + t, t);
    }
}

```

`JakartaCommonsLoggingImpl` 实例化对象返回 `com.ibatis.common.logging.Log` 对象实例。但引用的是 `Jakarta Commons Logging` 的内容。所以，在 `JakartaCommonsLoggingImpl` 实例化对象中的日志对象还是 `org.apache.commons.logging.Log` 的实例化对象，（代码如下）。

```

public JakartaCommonsLoggingImpl(Class clazz) {
    log = LogFactory.getLog(clazz);
}

```

为了更清楚地理解程序代码，代码可修改如下，效果是一样的。

```

public JakartaCommonsLoggingImpl(Class clazz) {
    org.apache.commons.logging.Log log;
    log = org.apache.commons.logging.LogFactory.getLog(clazz);
}

```

在这里把 `Jakarta Commons Logging (JCL)` 做一个详细说明。

`Jakarta Commons Logging (JCL)` 的 `Logging` 组件实现方式是将记录日志的功能封装为一组标准的 API，然后通过接口来提供服务，但是其底层实现可以任意修改和变换。所

以开发人员可以利用这个 API 来执行记录日志信息的命令,由 API 来决定把这些命令传递给适当的底层实现。因此,对于开发人员而言,Logging 组件对于任何具体的底层实现都是透明的。

使用 Commons 的 Logging API 非常简单。只需导入 Logging 的两个必须类、创建一个 Log 的静态实例,这部分操作的代码展示如下:

```
import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;

public class CommonLog {
    private static Log log = LogFactory.getLog(CommonLog.class);
    // ...
}
```

当调用 LogFactory.getLog 方法时。调用该函数会启动一个发现过程,即找出必需的底层日志记录功能的实现,具体的发现过程在下面列出。注意,不管底层的日志工具是怎么找到的,它都必须是一个实现了 Log 接口的类,且必须在 CLASSPATH 之中。Commons Logging API 直接提供对下列底层日志记录工具的支持: Jdk14Logger、Log4Jlogger、LogKitLogger、NoOpLogger (直接丢弃所有日志信息),还有一个 SimpleLog。

① Commons 的 Logging 首先在 CLASSPATH 中查找 commons-logging.properties 文件。这个属性文件至少定义 org.apache.commons.logging.Log 属性,它的值应该是上述任意 Log 接口实现的完整限定名称。如果找到 org.apache.commons.logging.Log 属性,则使用该属性对应的日志组件。结束发现过程。

② 如果上面的步骤失败(文件不存在或属性不存在),Commons 的 Logging 接着检查系统属性 org.apache.commons.logging.Log。如果找到 org.apache.commons.logging.Log 系统属性,则使用该系统属性对应的日志组件。结束发现过程。

③ 如果找不到 org.apache.commons.logging.Log 系统属性,Logging 接着在 CLASSPATH 中寻找 log4j 的类。如果找到了,Logging 就假定应用要使用的是 log4j。不过这时 log4j 本身的属性仍要通过 log4j.properties 文件正确配置。结束发现过程。

④ 如果上述查找均不能找到适当的 Logging API,但应用程序正运行在 JRE 1.4 或更高版本上,则默认使用 JRE 1.4 的日志记录功能。结束发现过程。

⑤ 最后,如果上述操作都失败,则应用将使用内建的 SimpleLog。SimpleLog 把所有日志信息直接输出到 System.err。结束发现过程。

获得适当的底层日志工具之后,接下来就可以开始记录日志信息。作为一种标准的 API,Commons Logging API 主要的好处是在底层日志机制的基础上建立了一个抽象层,通过抽象层把调用转换成与具体实现有关的日志记录命令。

iBATIS 框架的日志管理具有很大的移植性。只要稍微修改一下代码就可以运用到其他系统中去了。

12.4 调试信息工具

在 iBATIS 源码中,常常会用到 `ErrorContext` 类,这实际上是一个调试错误信息的工具。主要放在 `SQLMapConfiguration` 和 `StatementScope` 对象中进行使用。这个错误的上下文对象有助于我们理解有意义的错误信息。

在 `SQLMapConfiguration` 主要是进行解析文件的时候用,而 `StatementScope` 对象主要是进行业务处理的时候来用。

12.5 ScriptRunner 的应用

`ScriptRunner` 应用是 iBATIS 的 SQL 语句执行程序。可以读取一个 SQL 文件,然后批量进行 SQL 语句的执行。本功能在 `com.ibatis.common.jdbc` 包内的 `ScriptRunner` 类进行实现。

外部可以调用 `ScriptRunner` 执行的 SQL 文件格式如下(针对 MySQL 数据库)。

```
CREATE DATABASE JPETSTORE;
USE JPETSTORE;
DROP TABLE IF EXISTS account;

create table account (
    userid varchar(80) not null, email varchar(80) not null, firstname varchar(80) not null,
    lastname varchar(80) not null, status varchar(2) null, addr1 varchar(80) not null,
    addr2 varchar(40) null, city varchar(80) not null, state varchar(80) not null,
    zip varchar(20) not null, country varchar(20) not null, phone varchar(80) not null,
    constraint pk_account primary key (userid)
);
```

外部调用 `ScriptRunner` 的方式有多种,使用现成的数据库连接代码如下。

```
Connection conn = getConnection(); //some method to get a Connection
ScriptRunner runner = new ScriptRunner ();
runner.runScript(conn, Resources.getResourceAsReader("com/some/resource/path/
initialize.sql"));
conn.close();
```

使用新的数据库连接代码如下。

```
ScriptRunner runner = new ScriptRunner ("com.some.Driver", "jdbc:url://db",
"login", "password");
runner.runScript(conn, new FileReader("/usr/local/db/scripts/ initialize-db.sql"));
```

使用新创建的数据库连接代码如下。

```
Properties props = getProperties (); // some properties from somewhere
ScriptRunner runner = new ScriptRunner (props);
```

```
runner.runScript(conn, new FileReader("/usr/local/db/scripts/ initialize-db.sql"));
```

使用新的数据库连接代码如下。

```
ScriptRunner runner = new ScriptRunner ("com.some.Driver", "jdbc:url://db",
"login", "password");
runner.runScript(conn, new FileReader("/usr/local/db/scripts/ initialize-db.sql"));
```

在这里，先简单地介绍一下 ScriptRunner 的属性。ScriptRunner 的属性涉及好几个方面，如有关于数据库连接方面的，有关于错误处理方面的，有关于日志信息处理的，还有关于执行 SQL 的格式处理的。其属性如表 12-6 所示。

表 12-6 ScriptRunner 的属性列表

变量名称和类型	功能和作用
private Connection connection; private String driver; private String url; private String username; private String password;	数据库连接和数据库连接参数
private boolean stopOnError;	是否忽略 SQL 执行错误
private boolean autoCommit;	是否自动提交
private PrintWriter logWriter = new PrintWriter(System.out);	进行日志信息输出
private PrintWriter errorLogWriter = new PrintWriter(System.err);	进行错误信息输出
private String delimiter = DEFAULT_DELIMITER;	SQL 语句执行中的分行标志
private boolean fullLineDelimiter = false;	SQL 语句执行中是否不分行

了解这些属性，ScriptRunner 就可以实现执行 SQL 语句了。由上面的调用可以知道，都要先调用 runScript(Reader reader)方法，该方法的执行代码如下。

```
public void runScript(Reader reader) throws IOException, SQLException {
    try {
        if (connection == null) {
            DriverManager.registerDriver((Driver) Resources.instantiate
(driver));
            Connection conn = DriverManager.getConnection(url, username,
password);
            try {
                if (conn.getAutoCommit() != autoCommit) {
                    conn.setAutoCommit(autoCommit);
                }
                runScript(conn, reader);
            } finally { conn.close();}
        } else {
            boolean originalAutoCommit = connection.getAutoCommit();
            try {
                if (originalAutoCommit != this.autoCommit) {
                    connection.setAutoCommit(this.autoCommit);
                }
            }
        }
    }
}
```

```

        runScript(connection, reader);
    } finally {
        connection.setAutoCommit(originalAutoCommit);
    }
}
} catch (IOException e) { throw e;
} catch (SQLException e) {throw e;
} catch (Exception e) {throw new RuntimeException("Error running script.
Cause: " + e, e);
}
}
}

```

该程序代码首先连接数据库,获得数据库 Connection,然后执行 runScript(conn, reader),最后关闭数据库连接。runScript(conn, reader)方法的代码如下。

```

private void runScript(Connection conn, Reader reader) throws IOException,
SQLException {
    StringBuffer command = null;
    try {
        LineNumberReader lineReader = new LineNumberReader(reader);
        String line = null;
        while ((line = lineReader.readLine()) != null) {
            if (command == null) {command = new StringBuffer();}
            String trimmedLine = line.trim();
            if (trimmedLine.startsWith("--")) {println(trimmedLine);
            } else if (trimmedLine.length() < 1
                || trimmedLine.startsWith("//")) {
                // Do nothing
            } else if (trimmedLine.length() < 1
                || trimmedLine.startsWith("--")) {
                // Do nothing
            } else if (!fullLineDelimiter && trimmedLine.endsWith (getDelimiter()))
                || fullLineDelimiter && trimmedLine.equals(getDelimiter())) {
                ....
            } else {
                command.append(line);
                command.append(" ");
            }
        }
        if (!autoCommit) {
            conn.commit();
        }
    } catch (SQLException e) {
        e.fillInStackTrace();
        printlnError("Error executing: " + command);
        printlnError(e);
        throw e;
    } catch (IOException e) {
        e.fillInStackTrace();
        printlnError("Error executing: " + command);
        printlnError(e);
    }
}

```

```

        throw e;
    } finally {
        conn.rollback();
        flush();
    }
}

```

上述代码实现的功能是，先要通过 `LineNumberReader` 对象来打开 `reader` 数据流，这样可以采用行模式来读取文档。因为 `java.io.LineNumberReader` 类是 `java.io.BufferedReader` 类的扩展，它封装了处理行号的额外能力。程序在分析每行的时候，首先看行标，如果是“--”，则说明这只是一句注释。其中最复杂的是下面的这个判断，即具备了什么条件，可以去执行一句 SQL 语句。

```

(!fullLineDelimiter && trimmedLine.endsWith(getDelimiter())) || fullLineDelimiter &&
trimmedLine.equals(getDelimiter())

```

该判断说明，当本 SQL 语句需要有分行标识，同时行结尾存在 SQL 语句分行标识。那么，这个就是一句完整的 SQL。当然，这可能有两种情况。一种情况是 SQL 文件中某行直接就是一条 SQL 执行语句。还有一种情况是 SQL 文件中几行才能组合成一条 SQL 执行语句。所以，当不能满足该判断，则要在 `command` 上加入当前的行内容。由于是采用循环实现，这样就可以保证到了有 SQL 语句分行标识才能最后形成完整的 SQL 语句。

```

command.append(line);
command.append(" ");

```

当程序满足判断要求时，就获得了一句完整的 SQL 语句，接着执行对数据库的操作，代码如下。

```

command.append(line.substring(0, line.lastIndexOf(getDelimiter())));
command.append(" ");
Statement statement = conn.createStatement();
println(command);
boolean hasResults = false;
if (stopOnError) {
    hasResults = statement.execute(command.toString());
} else {
    try {
        statement.execute(command.toString());
    } catch (SQLException e) {
        e.fillInStackTrace();
        printlnError("Error executing: " + command);
        printlnError(e);
    }
}

if (autoCommit && !conn.getAutoCommit()) {conn.commit();}

```



```

ResultSet rs = statement.getResultSet();
if (hasResults && rs != null) {
    ResultSetMetaData md = rs.getMetaData();
    int cols = md.getColumnCount();
    for (int i = 0; i < cols; i++) {
        String name = md.getColumnLabel(i);
        print(name + "\t");
    }
    println("");
    while (rs.next()) {
        for (int i = 0; i < cols; i++) {
            String value = rs.getString(i);
            print(value + "\t");
        }
        println("");
    }
}

command = null;
try {
    statement.close();
} catch (Exception e) {
    // Ignore to workaround a bug in Jakarta DBCP
}

Thread.yield();

```

根据获得的 SQL 语句，创建一个 Statement 对象。如果 ScriptRunner 应用设置了忽略 SQL 执行错误属性，那么就调用 Statement 对象的 execute 方法，直接对数据库进行操作，否则获取异常并继续操作。如果 ScriptRunner 应用设置了自动提交，那就立刻把当前这个 SQL 语句进行事务提交。

最后进行线程 thread.yield()。这是在多线程程序中，为了防止某线程独占 CPU 资源（这样其他的线程就得不到“响应”了），可以让当前执行的线程“休息”一下。但是这种 thread.yield() 调用，并不保证下一个运行的线程就一定不是该线程。

12.6 读取源码的收获

通过对 iBATIS 常用工具源码的理解和分析，应该有以下几个收获。

第一，iBATIS 的 com.ibatis.common.resources.Resources 类为从类路径中加载资源，提供了更加方便和简化的使用方法，这对于我们在处理文件、资源加载时候可以不用去考虑当前的环境，同时结合 ClassInfo 类可以实现类名的动态加载。把一些复杂的、要全方位处理的问题简单化。这也是值得我们去学习的编程技巧。同时，这个类基本上在不做大修改

的情况下可以完全重用。

第二. iBATIS 的 Bean 管理中 ClassInfo 类的设计非常有意思,一方面它充当了一个静态的 Bean 容器,基于享元设计模式来处理各种各样的 Bean。另一方面,它也充当了一个 Class 的 Info 实例化对象。可把这两种角色合并到一个类来处理,这种设计方式有优点,也有缺点。优点是简化了代码,而且 ClassInfo 的构造方法是私有方法,这样保证了只有 ClassInfo 类中静态方法才能进行构造和实例化,强化了安全性。缺点是一个类处理两个角色,代码读起来有些难解。两种角色有时会有一些冲突。

第三. iBATIS 的 Probe 接口组件主要用于处理 Bean、DOM 对象和其他对象的转化,也是采用简单工厂设计模式,对于我们在编写 XML 和 Bean 处理方面有很多可以借鉴的源码内容。

第四. iBATIS 框架采用简单工厂模式对 Jakarta Commons Logging (JCL)、Log4J 和 JDK1.4 Logging API 等记录日志进行了封装。这样把几种日志集成一个统一的日志管理,将来可以采用插件方式融合更多的日志实现。这种设计模式值得我们学习,因这个封装的日志包基本上在不做大修改的情况下可以完全重用。

资源分享网
PDG

附录一

第 4 章 dao-2.dtd

```
<?xml version="1.0" encoding="UTF-8"?>
<!ELEMENT daoConfig (properties? , context+)>

<!ELEMENT context (transactionManager, dao*)+>
<!ATTLIST context
id CDATA #IMPLIED
>

<!ELEMENT properties EMPTY>
<!ATTLIST properties
resource CDATA #IMPLIED
url CDATA #IMPLIED
>

<!ELEMENT transactionManager (property*)>
<!ATTLIST transactionManager
type CDATA #REQUIRED
>

<!ELEMENT property EMPTY>
<!ATTLIST property
name CDATA #REQUIRED
value CDATA #REQUIRED
>

<!ELEMENT dao EMPTY>
<!ATTLIST dao
interface CDATA #REQUIRED
implementation CDATA #REQUIRED
>
```

附录二

第 5 章 SqlMapConfig.xml 的 DTD 结构

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

<!ELEMENT sqlMapConfig (properties?, settings?, resultObjectFactory?, typeAlias*,
typeHandler*, transactionManager?, sqlMap+)>
<!-- ATTLIST sqlMapConfig
xmlns:fo CDATA #IMPLIED
-->

<!-- ELEMENT properties EMPTY
-->
<!-- ATTLIST properties
resource CDATA #IMPLIED
url CDATA #IMPLIED
-->

<!-- ELEMENT settings EMPTY
-->
<!-- ATTLIST settings
classInfoCacheEnabled (true | false) #IMPLIED
lazyLoadingEnabled (true | false) #IMPLIED
statementCachingEnabled (true | false) #IMPLIED
cacheModelsEnabled (true | false) #IMPLIED
enhancementEnabled (true | false) #IMPLIED
errorTracingEnabled (true | false) #IMPLIED
useStatementNamespaces (true | false) #IMPLIED
useColumnLabel (true | false) #IMPLIED
forceMultipleResultSetSupport (true | false) #IMPLIED
maxSessions CDATA #IMPLIED
maxTransactions CDATA #IMPLIED
maxRequests CDATA #IMPLIED
defaultStatementTimeout CDATA #IMPLIED
-->

<!--The <transactionManager> element allows you to configure the transaction
```

management services for an

SQL Map. The type attribute indicates which transaction manager to use. The value can either be a class

name or a type alias. The three transaction managers included with the framework are: JDBC, JTA and

EXTERNAL.

JDBC - This allows JDBC to control the transaction via the usual Connection commit() and

rollback() methods.

JTA - This transaction manager uses a JTA global transaction such that the SQL Map activities can

be included as part of a wider scope transaction that possibly involves other databases or

transactional resources. This configuration requires a UserTransaction property set to locate the

user transaction from a JNDI resource.

EXTERNAL-This allows you to manage transactions on your own. You can still configure a data source, but transactions will not be committed or rolled back as part of the framework

lifecycle. This means that some part of your application external to Data Mapper must manage the

transactions. This setting is also useful for non-transactional databases (e.g. Read-only).

The <transactionManager> element also allows an optional attribute commitRequired that can be true or

false. Normally iBATIS will not commit transactions unless an insert, update, or delete operation has been

performed. This is true even if you explicitly call the commitTransaction() method. This behavior

creates problems in some cases. If you want iBATIS to always commit transactions, even if no insert,

update, or delete operation has been performed, then set the value of the commitRequired attribute to true.

Examples of where this attribute is useful include:

1. If you call a stored procedures that updates data as well as returning rows. In that case you would

call the procedure with the queryForList() operation - so iBATIS would not normally commit the

transaction. But then the updates would be rolled back.

2. In a WebSphere environment when you are using connection pooling and you use the JNDI

<dataSource> and the JDBC or JTA transaction manager. WebSphere requires all transactions on

pooled connections to be committed or the connection will not be returned to the pool.

Note that the commitRequired attribute has no effect when using the EXTERNAL transaction manager.

```

<br><br>
Some of the transaction managers allow extra configuration properties.-->
<!--ELEMENT transactionManager (property*,dataSource)>
<!--ATTLIST transactionManager
type CDATA #REQUIRED
commitRequired (true | false) #IMPLIED
>

```

<!--Included as part of the transaction manager configuration is a dataSource element and a set of properties to configure a DataSource for use with your SQL Map. There are currently three datasource factories

provided with the framework, but you can also write your own.

```
<br><br>
```

```
<b>SimpleDataSourceFactory</b><br>
```

The SimpleDataSource factory provides a basic implementation of a pooling DataSource that is

ideal for providing connections in cases where there is no container provided DataSource. It is

based on the iBATIS SimpleDataSource connection pool implementation.

```
<br><br>
```

```
<b>DbcpDataSourceFactory</b><br>
```

This implementation uses Jakarta DBCP (Database Connection Pool) to provide connection

pooling services via the DataSource API. This DataSource is ideal where the application/web

container cannot provide a DataSource implementation, or you're running a standalone application. IBATIS provides direct access to setting the properties of a DBCP datasource by

allowing you to specify any DBCP property name you desire in the configuration.

```
<br><br>
```

```
<b>JndiDataSourceFactory</b><br>
```

This implementation will retrieve a DataSource implementation from a JNDI context from within

an application container. This is typically used when an application server is in use and a

container managed connection pool and associated DataSource implementation are provided. The

standard way to access a JDBC DataSource implementation is via a JNDI context.

JndiDataSourceFactory provides functionality to access such a DataSource via JNDI. The

configuration parameters that must be specified in the datasource stanza are as follows:

```
<br><br>
```

```
<!--transactionManager type="JDBC" --><br>
```

```
  <!--dataSource type="JNDI"--><br>
```

```
    <!--property name="DataSource" value="java:comp/env/jdbc/jpetstore"/--><br>
```

```
  <!--/dataSource--><br>
```

```
<!--/transactionManager--><br>
```

```
<br>
```

The above configuration will use normal JDBC transaction management. But with a

container

managed resource, you might also want to configure it for global transactions as follows:

```
<br><br>
<transactionManager type="JTA" ><br>
    <property name="UserTransaction" value="java:/comp/UserTransaction"/><br>
    <dataSource type="JNDI"><br>
        <property name="DataSource" value="java:comp/env/jdbc/jpetstore"/><br>
    </dataSource><br>
</transactionManager><br>
<br>
```

Notice the UserTransaction property that points to the JNDI location where the UserTransaction

instance can be found. This is required for JTA transaction management so that your SQL Map

take part in a wider scoped transaction involving other databases and transactional resources.

JNDI context properties can be added before the lookup by specifying additional properties with a

```
prefix of "context."-->
<!ELEMENT dataSource (property*)>
<!-- ATTLIST dataSource
type CDATA #REQUIRED
-->
```

```
<!-- Defines a standard Java property. Is used by various elements to define settings.
-->
```

```
<!ELEMENT property EMPTY>
<!-- ATTLIST property
name CDATA #REQUIRED
value CDATA #REQUIRED
-->
```

<!-- The sqlMap element is used to explicitly include an SQL Map or another SQL Map Configuration file.

Each SQL Map XML file that is going to be used by this SqlMapClient instance, must be declared. The

SQL Map XML files will be loaded as a stream resource from the classpath or from a URL. You must

```
specify any and all Data Mapper (as many as there are). -->
<!ELEMENT sqlMap EMPTY>
<!-- ATTLIST sqlMap
resource CDATA #IMPLIED
url CDATA #IMPLIED
-->
```

<!--The typeAlias element simply allows you to specify a shorter name to refer to what is usually a long, fully qualified classname. For example:

```
<br><br>
<typeAlias alias="shortname" type="com.long.class.path.Class"/>-->
```

```

<!ELEMENT typeAlias EMPTY>
<!--ATTLIST typeAlias
alias CDATA #REQUIRED
type CDATA #REQUIRED
-->

```

<!--This element is used to declare a custom TypeHandler in iBATIS. This is necessary for iBATIS to know how to handle translations between the stated java type and jdbc type.-->

```

<!ELEMENT typeHandler EMPTY>
<!--ATTLIST typeHandler
javaType CDATA #REQUIRED
jdbcType CDATA #IMPLIED
callback CDATA #REQUIRED
-->

```

<!--The resultObjectFactory element allows you to specify a factory class for creating objects resulting from

the execution of SQL statements. This element is optional - if you don't specify the element, iBATIS will

use internal mechanisms to create result objects (class.newInstance()).

iBATIS creates result objects in these cases:

1. When mapping rows returned from a ResultSet (the most common case)

2. When you use a nested select statement on a result element in a resultMap. If the nested select

statement declares a parameterClass, then iBATIS will create and populate an instance of the class

before executing the nested select

3. When executing stored procedures - iBATIS will create objects for OUTPUT parameters

4. When processing nested result maps. If the nested result map is used in conjunction with the

groupBy support for avoiding N+1 queries, then the object will typically be an implementation of

type Collection, List, or Set. You can provide custom implementations of these interfaces through

the result object factory if you wish. In a 1:1 join with a nested result map, then iBATIS will

create an instance of the specified domain object through this factory.

If you choose to implement a factory, your factory class must implement the interface com.ibatis.sqlmap.engine.mapping.result.ResultObjectFactory, and your class must have a public

default constructor. The ResultObjectFactory interface has two-->

```

<!ELEMENT resultObjectFactory (property*)>

```

```

<!--ATTLIST resultObjectFactory

```

```

type CDATA #REQUIRED

```

```

>

```


附录三

第 5 章 SqlMapConfig.xml 的 XSD 结构

```
<?xml version="1.0" encoding="utf-8"?>
<xs:schema attributeFormDefault="unqualified"
  elementFormDefault="qualified" xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="sqlMapConfig">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="properties">
          <xs:complexType>
            <xs:attribute name="resource" type="xs:string" use="required" />
          </xs:complexType>
        </xs:element>
        <xs:element name="settings">
          <xs:complexType>
            <xs:attribute name="cacheModelsEnabled" type="xs:boolean" use="required" />
            <xs:attribute name="enhancementEnabled" type="xs:boolean" use="required" />
            <xs:attribute name="lazyLoadingEnabled" type="xs:boolean" use="required" />
            <xs:attribute name="maxRequests" type="xs:unsignedByte" use="required" />
            <xs:attribute name="maxSessions" type="xs:unsignedByte" use="required" />
            <xs:attribute name="maxTransactions" type="xs:unsignedByte" use="required" />
            <xs:attribute name="useStatementNamespaces" type="xs:boolean" use="required" />
          </xs:complexType>
        </xs:element>
        <xs:element name="typeAlias">
          <xs:complexType>
            <xs:attribute name="alias" type="xs:string" use="required" />
            <xs:attribute name="type" type="xs:string" use="required" />
          </xs:complexType>
        </xs:element>
        <xs:element name="transactionManager">
          <xs:complexType>
            <xs:sequence>
```

```
<xs:element name="dataSource">
  <xs:complexType>
    <xs:sequence>
      <xs:element maxOccurs="unbounded" name="property">
        <xs:complexType>
          <xs:attribute name="name" type="xs:string" use="required" />
          <xs:attribute name="value" type="xs:string" use="required" />
        </xs:complexType>
      </xs:element>
    </xs:sequence>
    <xs:attribute name="type" type="xs:string" use="required" />
  </xs:complexType>
</xs:element>
</xs:sequence>
<xs:attribute name="type" type="xs:string" use="required" />
</xs:complexType>
</xs:element>
<xs:element name="sqlMap">
  <xs:complexType>
    <xs:attribute name="resource" type="xs:string" use="required" />
  </xs:complexType>
</xs:element>
</xs:sequence>
</xs:complexType>
</xs:element>
</xs:schema>
```

PDF
PDG

附录四

第 5 章 SqlMapMapping.xml 的 DTD 结构

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

<!--
  Copyright 2004 Clinton Begin

  Licensed under the Apache License, Version 2.0 (the "License");
  you may not use this file except in compliance with the License.
  You may obtain a copy of the License at

      http://www.apache.org/licenses/LICENSE-2.0

  Unless required by applicable law or agreed to in writing, software
  distributed under the License is distributed on an "AS IS" BASIS,
  WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
  See the License for the specific language governing permissions and
  limitations under the License.
-->

<!--The root element of an iBATIS SQL Map file.
  A single SQL Map XML file can contain as many cache models, parameter maps, result
  maps and
  statements as you like. Use discretion and organize the statements and maps
  appropriately for your
  application (group them logically).-->
<!ELEMENT sqlMap (typeAlias* | cacheModel* | resultMap* | parameterMap* | sql* |
statement* | insert* | update* | delete* | select* | procedure*)+>
<!--ATTLIST sqlMap
xmlns:fo CDATA #IMPLIED
namespace CDATA #IMPLIED
>

<!--The parameterMap is responsible for mapping JavaBeans properties to the
```

parameters of a statement.

The parameterMap itself only requires a id attribute that is an identifier that statements will use to refer to it.

The class attribute is optional but highly recommended. Similar to the parameterClass attribute of a statement,

the class attribute allows the framework to validate the incoming parameter as well as optimize the engine

for performance. The parameterMap can contain any number of parameter mappings that map directly to the

parameters of a statement.-->

```
<!ELEMENT parameterMap (parameter+)>
```

```
<!ATTLIST parameterMap
```

```
id CDATA #REQUIRED
```

```
class CDATA #REQUIRED
```

```
>
```

```
<!-- The <parameter> tag is used to describe the parameters of a parameterMap.-->
```

```
<!ELEMENT parameter EMPTY>
```

```
<!ATTLIST parameter
```

```
property CDATA #REQUIRED
```

```
javaType CDATA #IMPLIED
```

```
jdbcType CDATA #IMPLIED
```

```
typeName CDATA #IMPLIED
```

```
nullValue CDATA #IMPLIED
```

```
mode (IN | OUT | INOUT) #IMPLIED
```

```
typeHandler CDATA #IMPLIED
```

```
resultMap CDATA #IMPLIED
```

```
numericScale CDATA #IMPLIED
```

```
>
```

<!--Result maps are an extremely important component of Data Mapper. The resultMap is responsible for

mapping JavaBeans properties to the columns of a ResultSet produced by executing a query mapped statement.

The resultMap itself has a id attribute that statements will use to refer to it. The resultMap also has a

class attribute that is the fully qualified (i.e. full package) name of a class or a type alias. This class will

be instantiated and populated based on the result mappings it contains. The extends attribute can be

optionally set to the name of another resultMap upon which to base a resultMap. This is similar to

extending a class in Java, all properties of the super resultMap will be included as part of the sub

resultMap. The properties of the super resultMap are always inserted before the sub resultMap

properties and the parent resultMap must be defined before the child. The classes for the super/sub

resultMaps need not be the same, nor do they need to be related at all (they can each use any class).

The resultMap element also supports the attribute groupBy. The groupBy attribute is used to specify a list of properties in this resultMap that are used to identify unique rows in the returned result set. Rows with equal values for the specified properties will only generate one result object. Use groupBy in combination with nested resultMaps to solve the N+1 query problem (see following discussion for examples).

The resultMap can contain any number of result mappings that map JavaBean properties to the columns of a ResultSet. These property mappings will be applied in the order that they are defined in the document. The associated class must be a JavaBeans compliant class with appropriate get/set methods for each of the

```

properties, a Map or XML.-->
<!ELEMENT resultMap (result*, discriminator?)>
<!--ATTLIST resultMap
id CDATA #REQUIRED
class CDATA #REQUIRED
extends CDATA #IMPLIED
xmlName CDATA #IMPLIED
groupBy CDATA #IMPLIED
-->
<!--Results map JavaBean properties to the columns of a ResultSet. These property
mappings will be applied
in the order that they are defined in the document. The associated class must be
a JavaBeans compliant
class with appropriate get/set methods for each of the properties, a Map or XML.-->
<!ELEMENT result EMPTY>
<!--ATTLIST result
property CDATA #REQUIRED
javaType CDATA #IMPLIED
column CDATA #IMPLIED
columnIndex CDATA #IMPLIED
jdbcType CDATA #IMPLIED
nullValue CDATA #IMPLIED
notNullColumn CDATA #IMPLIED
select CDATA #IMPLIED
resultMap CDATA #IMPLIED
typeHandler CDATA #IMPLIED
-->
<!-- TODO: add documentation. Could not find anything about this element in the
official ibatis doc.-->
<!ELEMENT discriminator (subMap+)>
<!--ATTLIST discriminator
javaType CDATA #REQUIRED
column CDATA #IMPLIED
columnIndex CDATA #IMPLIED
jdbcType CDATA #IMPLIED
nullValue CDATA #IMPLIED
-->

```

```

typeHandler CDATA #IMPLIED
>
<!-- TODO: add documentation. Could not find anything about this element in the
official iabtis doc. -->
<!ELEMENT subMap EMPTY>
<!ATTLIST subMap
value CDATA #REQUIRED
resultMap CDATA #REQUIRED
>
<!--A cacheModel is used to describe a cache for use with a query mapped statement.
Each query mapped statement
can use a different cacheModel, or the same one. The following example will demonstrate
how it looks related to a statement:
<br><br>
<cacheModel id="product-cache" type="LRU"><br>
    <flushInterval hours="24"/><br>
    <flushOnExecute statement="insertProduct"/><br>
    <flushOnExecute statement="updateProduct"/><br>
    <flushOnExecute statement="deleteProduct"/><br>
    <property name="size" value="1000" /><br>
</cacheModel><br>
<br>
<select id="getProductList" parameterClass="int" cacheModel="product-cache"><br>
    select * from PRODUCT where PRD_CAT_ID = #value#<br>
</select><br>

```

In the above example, a cache is defined for products that uses a WEAK reference type and flushes every

24 hours or whenever associated update statements are executed.-->

```
<!ELEMENT cacheModel (flushInterval?, flushOnExecute*, property*)>
```

```
<!ATTLIST cacheModel
```

```
id CDATA #REQUIRED
```

```
type CDATA #REQUIRED
```

```
readOnly (true | false) #IMPLIED
```

```
serialize (true | false) #IMPLIED
```

```
>
```

<!--Defines the interval of when the cache will be flushed. There can be only one flush interval element and it

can be set using hours, minutes, seconds or milliseconds.-->

```
<!ELEMENT flushInterval EMPTY>
```

```
<!ATTLIST flushInterval
```

```
milliseconds CDATA #IMPLIED
```

```
seconds CDATA #IMPLIED
```

```
minutes CDATA #IMPLIED
```

```
hours CDATA #IMPLIED
```

```
>
```

<!--Defines that the cache will be flushed when the specified statement is executed. There can

be any number of "flush on execute" elements specified for a cache.-->

```
<!ELEMENT flushOnExecute EMPTY>
```

```
<!ATTLIST flushOnExecute
```

```

statement CDATA #REQUIRED
>

<!--Defines a standard Java property. Is used by various elements to define
settings.-->
<!ELEMENT property EMPTY>
<!ATTLIST property
name CDATA #REQUIRED
value CDATA #REQUIRED
>

<!--The typeAlias element simply allows you to specify a shorter name to refer to
what is usually a long, fully
qualified classname. For example:

<typeAlias alias="shortname" type="com.long.class.path.Class"/>-->
<!ELEMENT typeAlias EMPTY>
<!ATTLIST typeAlias
alias CDATA #REQUIRED
type CDATA #REQUIRED
>

<!--The <sql> tag can be used to eliminate duplication of SQL code. You can
define define a part of SQL code via
the use of the <sql> tag and use it in a statement by using the <include> tag.
Here is a short example:

<br><br>
Example:<br>
<sql id="selectItem_fragment"><br>
    FROM items<br>
    WHERE parentid = 6<br>
</sql><br>
<br>
<select id="selectItemCount" resultClass="int"><br>
    SELECT COUNT(*) AS total<br>
    <include refid="selectItem_fragment"/><br>
</select><br>
<br>
<select id="selectItems" resultClass="Item"><br>
    SELECT id, name<br>
    <include refid="selectItem_fragment"/><br>
</select>-->
<!ELEMENT sql (#PCDATA | include | dynamic | iterate | isParameterPresent |
isNotParameterPresent | isEmpty | isNotEmpty | isNotNull | isNull | isEqual |
isGreaterThan | isGreaterEqual | isLessThan | isLessEqual | isPropertyAvailable |
isNotPropertyAvailable)*>
<!ATTLIST sql
id CDATA #REQUIRED
>

<!--The <include> tag can be used to insert SQL code fragments defined by the <sql>

```



```
isNotParameterPresent | isEmpty | isNotEmpty | isNotNull | isNull | isNotEqual | isEqual
| isGreaterThan | isGreaterEqual | isLessThan | isLessEqual | isPropertyAvailable |
isNotPropertyAvailable)*>
```

```
<!ATTLIST select
  id CDATA #REQUIRED
  parameterMap CDATA #IMPLIED
  parameterClass CDATA #IMPLIED
  resultMap CDATA #IMPLIED
  resultClass CDATA #IMPLIED
  cacheModel CDATA #IMPLIED
  resultSetType (FORWARD_ONLY | SCROLL_INSENSITIVE | SCROLL_SENSITIVE) #IMPLIED
  fetchSize CDATA #IMPLIED
  xmlResultName CDATA #IMPLIED
  remapResults (true|false) #IMPLIED
  timeout CDATA #IMPLIED
>
```

<!--Statement used for insert queries.

Supports all dynamic elements, <selectKey> and the query methods insert, update and delete.-->

```
<!ELEMENT insert ({PCDATA | include | dynamic | selectKey | iterate |
isParameterPresent | isNotParameterPresent | isEmpty | isNotEmpty | isNotNull | isNull
| isNotEqual | isEqual | isGreaterThan | isGreaterEqual | isLessThan | isLessEqual |
isPropertyAvailable | isNotPropertyAvailable})*>
```

```
<!ATTLIST insert
  id CDATA #REQUIRED
  parameterMap CDATA #IMPLIED
  parameterClass CDATA #IMPLIED
  timeout CDATA #IMPLIED
>
```

<!--Many relational database systems support auto-generation of primary key fields. This feature of the

RDBMS is often (if not always) proprietary. Data Mapper supports auto-generated keys via the

<selectKey> stanza of the <insert> element. Both pre-generated keys (e.g. Oracle) and post-generated

(MS-SQL Server) keys are supported.

The selectKey statement is executed before the insert statement if it is placed before the insert SQL,

otherwise the selectKey statement is executed after the insert statement. In the previous examples, the

Oracle example shows that the selectKey will be executed before the insert statement (as is appropriate for

a sequence). The SQL Server example shows that the selectKey statement will be executed after the insert

statement (as is appropriate for an identity column).

With iBATIS versions 2.2.0 and later, you can explicitly state the order of execution of the statements if

you wish. The selectKey element supports an attribute type that can be used to explicitly set the execution

order. The value of the type attribute can be either "pre" or "post" - meaning that the statement will be executed before or after the insert statement. If you specify the type attribute, then the value you specify

will be used regardless of the position of the selectKey element.-->

```
<!ELEMENT selectKey (#PCDATA | include)*>
```

```
<!ATTLIST selectKey
```

```
resultClass CDATA #IMPLIED
```

```
keyProperty CDATA #IMPLIED
```

```
type (pre|post) #IMPLIED
```

```
>
```

```
<!--Statement used for update queries.
```

```
Supports all dynamic elements and the query methods insert, update and delete.-->
```

```
<!ELEMENT update (#PCDATA | include | dynamic | iterate | isParameterPresent |
isNotParameterPresent | isEmpty | isNotEmpty | isNotNull | isNull | isNotEqual | isEqual
| isGreaterThan | isGreaterEqual | isLessThan | isLessEqual | isPropertyAvailable |
isNotPropertyAvailable)*>
```

```
<!ATTLIST update
```

```
id CDATA #REQUIRED
```

```
parameterMap CDATA #IMPLIED
```

```
parameterClass CDATA #IMPLIED
```

```
timeout CDATA #IMPLIED
```

```
>
```

```
<!--Statement used for delete queries.
```

```
Supports all dynamic elements and the query methods insert, update and delete.-->
```

```
<!ELEMENT delete (#PCDATA | include | dynamic | iterate | isParameterPresent |
isNotParameterPresent | isEmpty | isNotEmpty | isNotNull | isNull | isNotEqual | isEqual
| isGreaterThan | isGreaterEqual | isLessThan | isLessEqual | isPropertyAvailable |
isNotPropertyAvailable)*>
```

```
<!ATTLIST delete
```

```
id CDATA #REQUIRED
```

```
parameterMap CDATA #IMPLIED
```

```
parameterClass CDATA #IMPLIED
```

```
timeout CDATA #IMPLIED
```

```
>
```

<!--The <procedure> statement element is used for Stored Procedures. The following example shows

how a stored procedure would be used with output parameters:

```
<br><br>
```

```
&lt;parameterMap id="swapParameters" class="map" &gt;<br>
```

```
&lt;parameter property="email1" jdbcType="VARCHAR" javaType="java.lang.
String" mode="INOUT"/&gt;<br>
```

```
&lt;parameter property="email2" jdbcType="VARCHAR" javaType="java.lang.
String" mode="INOUT"/&gt;<br>
```

```
&lt;/parameterMap&gt;<br>
```

```
<br>
```

```
&lt;procedure id="swapEmailAddresses" parameterMap="swapParameters" &gt;<br>
```

```
(call swap_email_address (?, ?))<br>
```

```

</procedure><br>
<br>
    Calling the above procedure would swap two email addresses between two columns
(database table) and
    also in the parameter object (Map). The parameter object is only modified if the
parameter mappings mode
    attribute is set to "INOUT" or "OUT". Otherwise they are left unchanged. Obviously
immutable parameter
    objects (e.g. String) cannot be modified.-->
    <!ELEMENT procedure (#PCDATA | include | dynamic | iterate | isParameterPresent |
isNotParameterPresent | isEmpty | isNotEmpty | isNotNull | isNull | isNotEqual | isEqual
| isGreaterThan | isGreaterEqual | isLessThan | isLessEqual | isPropertyAvailable |
isNotPropertyAvailable)*>
    <!ATTLIST procedure
    id CDATA #REQUIRED
    parameterMap CDATA #IMPLIED
    parameterClass CDATA #IMPLIED
    resultMap CDATA #IMPLIED
    resultClass CDATA #IMPLIED
    cacheModel CDATA #IMPLIED
    fetchSize CDATA #IMPLIED
    xmlResultName CDATA #IMPLIED
    remapResults (true|false) #IMPLIED
    timeout CDATA #IMPLIED
    >

    <!-- -----
          DYNAMIC ELEMENTS
    ----- -->

    <!--Wrapper tag that allows for an overall prepend, open and close.-->
    <!ELEMENT dynamic (#PCDATA | include | iterate | isParameterPresent |
isNotParameterPresent | isEmpty | isNotEmpty | isNotNull | isNull | isNotEqual | isEqual
| isGreaterThan | isGreaterEqual | isLessThan | isLessEqual | isPropertyAvailable |
isNotPropertyAvailable)*>
    <!ATTLIST dynamic
    prepend CDATA #IMPLIED
    open CDATA #IMPLIED
    close CDATA #IMPLIED
    >

    <!--Checks if a property is not null.-->
    <!ELEMENT isNotNull (#PCDATA | include | iterate | isParameterPresent |
isNotParameterPresent | isEmpty | isNotEmpty | isNotNull | isNull | isNotEqual | isEqual
| isGreaterThan | isGreaterEqual | isLessThan | isLessEqual | isPropertyAvailable |
isNotPropertyAvailable)*>
    <!ATTLIST isNotNull
    prepend CDATA #IMPLIED
    open CDATA #IMPLIED
    close CDATA #IMPLIED
    property CDATA #IMPLIED

```

```

removeFirstPrepend (true|false) #IMPLIED
>

<!--Checks if a property is null.-->
<!ELEMENT isNull (#PCDATA | include | iterate | isParameterPresent |
isNotParameterPresent | isEmpty | isNotEmpty | isNotNull | isNull | isNotEqual | isEqual
| isGreaterThan | isGreaterEqual | isLessThan | isLessEqual | isPropertyAvailable |
isNotPropertyAvailable)*>
<!--ATTLIST isNull
prepend CDATA #IMPLIED
open CDATA #IMPLIED
close CDATA #IMPLIED
property CDATA #IMPLIED
removeFirstPrepend (true|false) #IMPLIED
>

<!--Checks if a property is unavailable (i.e not a property of the parameter bean).-->
<!ELEMENT isNotPropertyAvailable (#PCDATA | include | iterate | isParameterPresent
| isNotParameterPresent | isEmpty | isNotEmpty | isNotNull | isNull | isNotEqual | isEqual
| isGreaterThan | isGreaterEqual | isLessThan | isLessEqual | isPropertyAvailable |
isNotPropertyAvailable)*>
<!--ATTLIST isNotPropertyAvailable
prepend CDATA #IMPLIED
open CDATA #IMPLIED
close CDATA #IMPLIED
property CDATA #REQUIRED
removeFirstPrepend (true|false) #IMPLIED
>

<!--Checks if a property is available (i.e is a property of the parameter bean).-->
<!ELEMENT isPropertyAvailable (#PCDATA | include | iterate | isParameterPresent |
isNotParameterPresent | isEmpty | isNotEmpty | isNotNull | isNull | isNotEqual | isEqual
| isGreaterThan | isGreaterEqual | isLessThan | isLessEqual | isPropertyAvailable |
isNotPropertyAvailable)*>
<!--ATTLIST isPropertyAvailable
prepend CDATA #IMPLIED
open CDATA #IMPLIED
close CDATA #IMPLIED
property CDATA #REQUIRED
removeFirstPrepend (true|false) #IMPLIED
>

<!--Checks the equality of a property and a value, or another property.-->
<!ELEMENT isEqual (#PCDATA | include | iterate | isParameterPresent |
isNotParameterPresent | isEmpty | isNotEmpty | isNotNull | isNull | isNotEqual | isEqual
| isGreaterThan | isGreaterEqual | isLessThan | isLessEqual | isPropertyAvailable |
isNotPropertyAvailable)*>
<!--ATTLIST isEqual
prepend CDATA #IMPLIED
open CDATA #IMPLIED
close CDATA #IMPLIED

```



```

property CDATA #IMPLIED
removeFirstPrepend (true|false) #IMPLIED
compareProperty CDATA #IMPLIED
compareValue CDATA #IMPLIED
>

<!--Checks the inequality of a property and a value, or another property.-->
<!ELEMENT isNotEqual (#PCDATA | include | iterate | isParameterPresent |
isNotParameterPresent | isEmpty | isNotEmpty | isNotNull | isNull | isNotEqual | isEqual
| isGreaterThan | isGreaterEqual | isLessThan | isLessEqual | isPropertyAvailable |
isNotPropertyAvailable)*>
<!ATTLIST isNotEqual
prepend CDATA #IMPLIED
open CDATA #IMPLIED
close CDATA #IMPLIED
property CDATA #IMPLIED
removeFirstPrepend (true|false) #IMPLIED
compareProperty CDATA #IMPLIED
compareValue CDATA #IMPLIED
>

<!--Checks if a property is greater than a value or another property.-->
<!ELEMENT isGreaterThan (#PCDATA | include | iterate | isParameterPresent |
isNotParameterPresent | isEmpty | isNotEmpty | isNotNull | isNull | isNotEqual | isEqual
| isGreaterThan | isGreaterEqual | isLessThan | isLessEqual | isPropertyAvailable |
isNotPropertyAvailable)*>
<!ATTLIST isGreaterThan
prepend CDATA #IMPLIED
open CDATA #IMPLIED
close CDATA #IMPLIED
property CDATA #IMPLIED
removeFirstPrepend (true|false) #IMPLIED
compareProperty CDATA #IMPLIED
compareValue CDATA #IMPLIED
>

<!--Checks if a property is greater than or equal to a value or another property.-->
<!ELEMENT isGreaterEqual (#PCDATA | include | iterate | isParameterPresent |
isNotParameterPresent | isEmpty | isNotEmpty | isNotNull | isNull | isNotEqual | isEqual
| isGreaterThan | isGreaterEqual | isLessThan | isLessEqual | isPropertyAvailable |
isNotPropertyAvailable)*>
<!ATTLIST isGreaterEqual
prepend CDATA #IMPLIED
open CDATA #IMPLIED
close CDATA #IMPLIED
property CDATA #IMPLIED
removeFirstPrepend (true|false) #IMPLIED
compareProperty CDATA #IMPLIED
compareValue CDATA #IMPLIED
>

```

```

<!--Checks if a property is less than a value or another property.-->
<!ELEMENT isLessThan (#PCDATA | include | iterate | isParameterPresent |
isNotParameterPresent | isEmpty | isNotEmpty | isNotNull | isNull | isNotEqual | isEqual
| isGreaterThan | isGreaterEqual | isLessThan | isLessEqual | isPropertyAvailable |
isNotPropertyAvailable)*>
  <!ATTLIST isLessThan
    prepend CDATA #IMPLIED
    open CDATA #IMPLIED
    close CDATA #IMPLIED
    property CDATA #IMPLIED
    removeFirstPrepend (true|false) #IMPLIED
    compareProperty CDATA #IMPLIED
    compareValue CDATA #IMPLIED
  >

<!--Checks if a property is less than or equal to a value or another property.
<br>
Example Usage:<br>
&lt;isLessEqual prepend="AND" property="age" compareValue="18"&gt;<br>
  ADOLESCENT = 'TRUE'<br>
&lt;/isLessEqual&gt;-->
<!ELEMENT isLessEqual (#PCDATA | include | iterate | isParameterPresent |
isNotParameterPresent | isEmpty | isNotEmpty | isNotNull | isNull | isNotEqual | isEqual
| isGreaterThan | isGreaterEqual | isLessThan | isLessEqual | isPropertyAvailable |
isNotPropertyAvailable)*>
  <!ATTLIST isLessEqual
    prepend CDATA #IMPLIED
    open CDATA #IMPLIED
    close CDATA #IMPLIED
    property CDATA #IMPLIED
    removeFirstPrepend (true|false) #IMPLIED
    compareProperty CDATA #IMPLIED
    compareValue CDATA #IMPLIED
  >

<!--Checks to see if the value of a Collection, String or String.valueOf() property
is null or empty (" or size() < 1).-->
<!ELEMENT isEmpty (#PCDATA | include | iterate | isParameterPresent |
isNotParameterPresent | isEmpty | isNotEmpty | isNotNull | isNull | isNotEqual | isEqual
| isGreaterThan | isGreaterEqual | isLessThan | isLessEqual | isPropertyAvailable |
isNotPropertyAvailable)*>
  <!ATTLIST isEmpty
    prepend CDATA #IMPLIED
    open CDATA #IMPLIED
    close CDATA #IMPLIED
    property CDATA #IMPLIED
    removeFirstPrepend (true|false) #IMPLIED
  >

<!--Checks to see if the value of a Collection, String or String.valueOf() property
is not null and not empty (" or size() < 1).

```



```

<br><br>
Example Usage:<br>
<lt;isEmpty prepend="AND" property="firstName" >><br>
    FIRST_NAME=#firstName#<br>
<lt;/isEmpty><br>
<ELEMENT isEmpty (#PCDATA | include | iterate | isParameterPresent |
isEmptyParameterPresent | isEmpty | isNotEmpty | isNotNull | isNull | isNotEqual | isEqual
| isGreaterThan | isGreaterEqual | isLessThan | isLessEqual | isPropertyAvailable |
isNotPropertyAvailable)*>
    <!ATTLIST isEmpty
    prepend CDATA #IMPLIED
    open CDATA #IMPLIED
    close CDATA #IMPLIED
    property CDATA #IMPLIED
    removeFirstPrepend (true|false) #IMPLIED
    >

    <!--Checks to see if the parameter object is present (not null).-->
    <ELEMENT isParameterPresent (#PCDATA | include | iterate | isParameterPresent |
isEmptyParameterPresent | isEmpty | isNotEmpty | isNotNull | isNull | isNotEqual | isEqual
| isGreaterThan | isGreaterEqual | isLessThan | isLessEqual | isPropertyAvailable |
isNotPropertyAvailable)*>
    <!ATTLIST isParameterPresent
    prepend CDATA #IMPLIED
    open CDATA #IMPLIED
    close CDATA #IMPLIED
    removeFirstPrepend (true|false) #IMPLIED
    >

    <!--Checks to see if the parameter object is not present (null). -->
    <ELEMENT isNotParameterPresent (#PCDATA | include | iterate | isParameterPresent
| isEmptyParameterPresent | isEmpty | isNotEmpty | isNotNull | isNull | isNotEqual | isEqual
| isGreaterThan | isGreaterEqual | isLessThan | isLessEqual | isPropertyAvailable |
isNotPropertyAvailable)*>
    <!ATTLIST isNotParameterPresent
    prepend CDATA #IMPLIED
    open CDATA #IMPLIED
    close CDATA #IMPLIED
    removeFirstPrepend (true|false) #IMPLIED
    >

    <!--Iterates over a property that is an implementation java.util.Collection, or
java.util.Iterator, or is an array.
<br><br>
Example Usage:<br>
<lt;iterate prepend="AND" property="userNameList" open="(" close=")" conjunction=
"OR"><br>
    username=#userNameList[]#<br>
<lt;/iterate><br>
<br>
    It is also possible to use the iterate when the collection is passed in as a

```

parameter to your mapped statement.

Example Usage:


```
<iterate prepend="AND" open="(" close=")" conjunction="OR"><br>
    username=#{}<br>
</iterate><br>
```

Note: It is very important to include the square brackets[] at the end of the property name when using the Iterate element. These brackets distinguish this object as a collection to keep the parser from simply outputting the collection as a string.-->

```
<!ELEMENT iterate (#PCDATA | include | iterate | isParameterPresent |
isNotParameterPresent | isEmpty | isNotEmpty | isNotNull | isNull | isNotEqual | isEqual
| isGreaterThan | isGreaterEqual | isLessThan | isLessEqual | isPropertyAvailable |
isNotPropertyAvailable)*>
<!ATTLIST iterate
prepend CDATA #IMPLIED
property CDATA #IMPLIED
removeFirstPrepend (true|false|iterate) #IMPLIED
open CDATA #IMPLIED
close CDATA #IMPLIED
conjunction CDATA #IMPLIED
>
```

附录五

第 5 章 SqlMapping.xml 的 XSD 结构

```
<?xml version="1.0" encoding="utf-8"?>
<xs:schema attributeFormDefault="unqualified"
elementFormDefault="qualified" xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="sqlMap">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="cacheModel">
          <xs:complexType>
            <xs:sequence>
              <xs:element name="flushInterval">
                <xs:complexType>
                  <xs:attribute name="hours" type="xs:unsignedByte" use="required" />
                </xs:complexType>
              </xs:element>
              <xs:element name="property">
                <xs:complexType>
                  <xs:attribute name="name" type="xs:string" use="required" />
                  <xs:attribute name="value" type="xs:unsignedShort" use="required"
/>
                </xs:complexType>
              </xs:element>
            </xs:sequence>
            <xs:attribute name="id" type="xs:string" use="required" />
            <xs:attribute name="type" type="xs:string" use="required" />
          </xs:complexType>
        </xs:element>
        <xs:element name="typeAlias">
          <xs:complexType>
            <xs:attribute name="alias" type="xs:string" use="required" />
            <xs:attribute name="type" type="xs:string" use="required" />
          </xs:complexType>
        </xs:element>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

```

<xs:element name="parameterMap">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="parameter">
        <xs:complexType>
          <xs:attribute name="property" type="xs:string" use="required" />
        </xs:complexType>
      </xs:element>
    </xs:sequence>
    <xs:attribute name="id" type="xs:string" use="required" />
    <xs:attribute name="class" type="xs:string" use="required" />
  </xs:complexType>
</xs:element>
<xs:element name="resultMap">
  <xs:complexType>
    <xs:sequence>
      <xs:element maxOccurs="unbounded" name="result">
        <xs:complexType>
          <xs:attribute name="property" type="xs:string" use="required" />
          <xs:attribute name="column" type="xs:string" use="required" />
        </xs:complexType>
      </xs:element>
    </xs:sequence>
    <xs:attribute name="id" type="xs:string" use="required" />
    <xs:attribute name="class" type="xs:string" use="required" />
  </xs:complexType>
</xs:element>
<xs:element name="select">
  <xs:complexType>
    <xs:simpleContent>
      <xs:extension base="xs:string">
        <xs:attribute name="id" type="xs:string" use="required" />
        <xs:attribute name="parameterMap" type="xs:string" use="required" />
        <xs:attribute name="resultMap" type="xs:string" use="required" />
        <xs:attribute name="cacheModel" type="xs:string" use="required" />
      </xs:extension>
    </xs:simpleContent>
  </xs:complexType>
</xs:element>
<xs:element name="insert">
  <xs:complexType>
    <xs:simpleContent>
      <xs:extension base="xs:string">
        <xs:attribute name="id" type="xs:string" use="required" />
        <xs:attribute name="parameterClass" type="xs:string" use="required" />
      </xs:extension>
    </xs:simpleContent>
  </xs:complexType>
</xs:element>
<xs:element name="update">
  <xs:complexType>

```

```
<xs:simpleContent>
  <xs:extension base="xs:string">
    <xs:attribute name="id" type="xs:string" use="required" />
    <xs:attribute name="parameterClass" type="xs:string" use="required" />
  </xs:extension>
</xs:simpleContent>
</xs:complexType>
</xs:element>
<xs:element name="delete">
  <xs:complexType>
    <xs:simpleContent>
      <xs:extension base="xs:string">
        <xs:attribute name="id" type="xs:string" use="required" />
        <xs:attribute name="parameterClass" type="xs:string" use="required" />
      </xs:extension>
    </xs:simpleContent>
  </xs:complexType>
</xs:element>
</xs:sequence>
<xs:attribute name="namespace" type="xs:string" use="required" />
</xs:complexType>
</xs:element>
</xs:schema>
```

附录六

第 11 章 JDBC Types Mapped to Java Types

| 序 号 | JDBC Type | Java Type |
|-----|---------------|----------------------------|
| 1 | CHAR | String |
| 2 | VARCHAR | String |
| 3 | LONGVARCHAR | String |
| 4 | NUMERIC | java.math.BigDecimal |
| 5 | DECIMAL | java.math.BigDecimal |
| 6 | BIT | boolean |
| 7 | TINYINT | byte |
| 8 | SMALLINT | short |
| 9 | INTEGER | int |
| 10 | BIGINT | long |
| 11 | REAL | float |
| 12 | FLOAT | double |
| 13 | DOUBLE | double |
| 14 | BINARY | byte[] |
| 15 | VARBINARY | byte[] |
| 16 | LONGVARBINARY | byte[] |
| 17 | DATE | java.sql.Date |
| 18 | TIME | java.sql.Time |
| 19 | TIMESTAMP | java.sql.Timestamp |
| 20 | CLOB | Clob |
| 21 | BLOB | Blob |
| 22 | ARRAY | Array |
| 23 | DISTINCT | mapping of underlying type |
| 24 | STRUCT | Struct |
| 25 | REF | Ref |
| 6 | JAVA_OBJECT | underlying Java class |

附录七

第 11 章 Java Types Mapped to JDBC Types

| 序 号 | Java Types | JDBC Types |
|-----|----------------------|-------------------------------------|
| 1 | String | CHAR, VARCHAR, or LONGVARCHAR |
| 2 | java.math.BigDecimal | NUMERIC |
| 3 | boolean | BIT |
| 4 | byte | TINYINT |
| 5 | short | SMALLINT |
| 6 | int | INTEGER |
| 7 | long | BIGINT |
| 8 | float | REAL |
| 9 | double | DOUBLE |
| 10 | byte[] | BINARY, VARBINARY, or LONGVARBINARY |
| 11 | java.sql.Date | DATE |
| 12 | java.sql.Time | TIME |
| 13 | java.sql.Timestamp | TIMESTAMP |
| 14 | Clob | CLOB |
| 15 | Blob | BLOB |
| 16 | Array | ARRAY |
| 17 | Struct | STRUCT |
| 18 | Ref | REF |
| 19 | Java class | JAVA_OBJECT |

附录八

第 11 章 JDBC Types Mapped to Java Object Types

| 序 号 | JDBC Type | Java Object Type |
|-----|---------------|--------------------------------|
| 1 | CHAR | String |
| 2 | VARCHAR | String |
| 3 | LONGVARCHAR | String |
| 4 | NUMERIC | java.math.BigDecimal |
| 5 | DECIMAL | java.math.BigDecimal |
| 6 | BIT | Boolean |
| 7 | TINYINT | Integer |
| 8 | SMALLINT | Integer |
| 9 | INTEGER | Integer |
| 10 | BIGINT | Long |
| 11 | REAL | Float |
| 12 | FLOAT | Double |
| 13 | DOUBLE | Double |
| 14 | BINARY | byte[] |
| 15 | VARBINARY | byte[] |
| 16 | LONGVARBINARY | byte[] |
| 17 | DATE | java.sql.Date |
| 18 | TIME | java.sql.Time |
| 19 | TIMESTAMP | java.sql.Timestamp |
| 20 | DISTINCT | Object type of underlying type |
| 21 | CLOB | Clob |
| 22 | BLOB | Blob |
| 23 | ARRAY | Array |
| 24 | STRUCT | Struct or SQLData |
| 25 | REF | Ref |
| 26 | JAVA_OBJECT | underlying Java class |

附录九

第 11 章 Java Object Types Mapped to JDBC Types

| 序 号 | Java Object Type | JDBC Type |
|-----|----------------------|-------------------------------------|
| 1 | String | CHAR, VARCHAR, or LONGVARCHAR |
| 2 | java.math.BigDecimal | NUMERIC |
| 3 | Boolean | BIT |
| 4 | Integer | INTEGER |
| 5 | Long | BIGINT |
| 6 | Float | REAL |
| 7 | Double | DOUBLE |
| 8 | byte[] | BINARY, VARBINARY, or LONGVARBINARY |
| 9 | java.sql.Date | DATE |
| 10 | java.sql.Time | TIME |
| 11 | java.sql.Timestamp | TIMESTAMP |
| 12 | Clob | CLOB |
| 13 | Blob | BLOB |
| 14 | Array | ARRAY |
| 15 | Struct | STRUCT |
| 16 | Ref | REF |
| 17 | Java class | JAVA_OBJECT |

附录十

第 11 章 JDBC Types Mapped to Database-specific SQL Types

| JDBC Type Name | Oracle 8.1 | Sybase 11.9 | Informix 9.2 | IBM DB2 5.2(Unix,NT) | Microsoft SQL Server 7.0 | Microsoft Access 7.0 | Sybase SQL Anywhere 6.0 |
|----------------|---------------------------|----------------------|----------------------|-----------------------|--------------------------|----------------------|--------------------------|
| BIT | | BIT | | | BIT | BIT | BIT |
| TINYINT | | TINYINT | SMALLINT | | TINYINT | BYTE | TINYINT |
| SMALLINT | SMALLINT, NUMBER(38,0) | SMALLINT | SMALLINT | SMALLINT | SMALLINT | SMALLINT | SMALLINT |
| INTEGER | INTEGER, NUMBER(38,0) | INTEGER | INTEGER, SERIAL | INTEGER | INTEGER | INTEGER, COUNTER | INTEGER |
| BIGINT | | | INT8,SERIAL8 | BIGINT | | | |
| REAL | REAL,NUMBER | REAL | REAL | REAL | REAL | REAL | REAL |
| FLOAT | FLOAT,NUMBER | FLOAT | FLOAT | FLOAT | FLOAT | | FLOAT |
| DOUBLE | DOUBLE PRECISION, NUMBER | DOUBLE PRECISION | DOUBLE PRECISION | FLOAT | DOUBLE PRECISION | DOUBLE | DOUBLE PRECISION |
| NUMERIC(p,s) | NUMERIC(p,s), NUMBER(p,s) | NUMERIC(p,s) | NUMERIC(p,s) | NUMERIC(p,s) | NUMERIC(p,s) | CURRENCY | NUMERIC(p,s) |
| DECIMAL(p,s) | DECIMAL(p,s), NUMBER(p,s) | DECIMAL(p,s),MONEY | DECIMAL(p,s) | DECIMAL(p,s) | DECIMAL(p,s),MONEY | | DECIMAL(p,s) |
| CHAR(n) | CHAR(n)
n<=2000 | CHAR(n)
n<=255 | CHAR(n)
n<=32767 | CHAR(n)
n<=254 | CHAR(n)
n<=8000 | CHAR(n)
n<=255 | CHAR(n)
n<=32,767 |
| VARCHAR(n) | VARCHAR2(n)
n<=4000 | VARCHAR(n)
n<=255 | VARCHAR(n)
n<=255 | VARCHAR(n)
n<=4000 | VARCHAR(n)
n<=8000 | VARCHAR(n)
n<=255 | VARCHAR2(n)
n<=32,767 |

续表

| JDBC Type Name | Oracle 8.1 | Sybase 11.9 | Informix 9.2 | IBM DB2 5.2(Unix,NT) | Microsoft SQL Server 7.0 | Microsoft Access 7.0 | Sybase SQL Anywhere 6.0 |
|----------------|------------------------------|----------------------------|---------------------------|---|----------------------------|-----------------------------------|-----------------------------------|
| LONGYARBAR | LONG Limit is 2 Gigobytes | TEXT Limit is 2 Gigobytes | TEXT Limit is 2 Gigobytes | LONGVARCH AR Limit is 32,700 bytes | TEXT Limit is 2 Gigobytes | LONGCHAR Limit is 2.0 Gigobytes | LONGVARCH AR Limit is 2 Gigobytes |
| BINARB(n) | | BINARY(n) n<=255 | BYTE | CHAR(n)FORB IT DATA n<=254 | BINARY(n) n<=8000 | BINARY(n) n<=255 | BINARY n<=32,767 |
| VARBINARP | RAW(n) n<=2000 | VARBINARY (n) n<=255 | BYTE | VARCHAR(n)F OR BIT DATA n<=4000 | VARBINARY (n) n<=8000 | VARBINARY (n) n<=255 | |
| LONGYARB INARY | LONGRAW Limit is 2 Gigobytes | IMAGE Limit is 2 Gigobytes | BYTE Limit is 2 Gigobytes | LONG VARCH AR FORBIT DATA Limit is 32,700 bytes | IMAGE Limit is 2 Gigobytes | LONGBINARY Limit is 1.0 Gigobytes | IMAGE Limit is 2 Gigobytes |
| DATE | | | DATE | DATE | | | DATE |



参考文献

- [1] iBATIS Data Access Objects Developer Guide(Version 2.0), February 18, 2006
- [2] iBATIS Data Mapper Developer Guide (Version 2.0). November 30, 2006
- [3] Clinton Begin 著 刘涛译. iBATIS SQL Maps 开发指南 (Version 2.0). 2004.6.17
- [4] Clinton Begin (加)等著, 叶俊等译. iBATIS 实战. 北京: 人民邮电出版社, 2008.05
- [5] E.Gamma, R.Helm, R.Johnson, and Vlissides. Design Patterns Elements of Reusable Object Oriented Software. Addison-Wesley, 1995
- [6] E.Gamma, R.Helm, R.Johnson, and Vlissides. 著, 李英军等译, 设计模式: 可复用面向对象软件的基础, 北京: 机械工业出版社. 2000.9
- [7] 布莱特(美), 皮瑞拉尼(美) 著, 宋今, 赵丰年 译. 面向对象的建模与设计在数据库中的应用[M]. 北京: 北京理工大学出版社, 2001.
- [8] Rosanna Lee (美), scott Sehgan(美) . JNDI API Tutorial and Reference. 出版社: Addison-Wesley Longman Publishing Co. IncBoston, MA, USA(2000-2)
- [9] Herbert Schildt 著 张玉清 吴溥峰等 译. Java(TM) 2 参考大全 (第四版) [M] . 北京: 清华大学出版社, 2002
- [10] 王珊, 萨师煊 著. 数据库系统概论(第4版)[M]. 北京: 高等教育出版社, 2006
- [11] 罗摩克里希纳(美), 格尔基(美) 著. 数据库管理系统(第2版) [M]. 北京: 清华大学出版社, 2004.3
- [12] 马丁(Martin,R.C.)(美) 著, 黄晓春 译. UML: Java 程序员指南(双语版) [M]. 北京: 清华大学出版社, 2004
- [13] 王钱 王蓉 张利. 基于 iBATIS 的通用数据持久层的研究与设计. 微计算机信息, 2007. (04X): 172-173,128.
- [14] 周相兵 杨小平. 基于 iBATIS 的持久性层次框架研究. 科学技术与工程, 2007.16 (7): 4062-4066.
- [15] 李澎林 朱国清 吴斌. 基于 iBATIS SQL Map 的数据持久层实现应用研究. 浙江工业

- 大学学报, 2008.36 (1): 72-76.
- [16] 刘军 戴金山. 基于 Spring MVC 与 iBATIS 的轻量级 Web 应用研究. 计算机应用, 2006.26 (4): 840-843.
- [17] 封小钰 王飞. 基于 iBATIS 数据库访问技术的研究与应用. 电脑开发与应用, 2008.21 (6): 51-53.
- [18] 吴建设. 基于数据持久层的 iBATIS DAO 研究与应用. 黄石理工学院学报, 2007.23 (4): 19-22.
- [19] 何铮 陈志刚. 对象 / 关系映射框架的研究与应用. 计算机工程与应用, 2003.39 (26): 188-191,194.
- [20] 黄丽娟 郑雪峰 罗涛. 对象映射关系型数据库技术研究. 计算机工程与设计, 2004.25 (11): 1994-1995,1998.
- [21] 张凯. JNDI 技术及其使用方法. 科技信息, 2009 (04X): 153-153.
- [22] 齐勇 马莉 等. 分布式事务处理技术及其模型. 计算机工程与应用, 2001.37(9): 60-62.
- [23] 孙勇强 李续武. 一种通用的 EJB 方法调用框架的设计与实现. 计算机与现代化, 2005 (2): 100-102.
- [24] 李志勇 尤淑辉. 基于 Java 动态代理实现分布式网管 F 口调用设计. 计算机与现代化, 2004 (10): 43-46.
- [25] 吴继栋. 浅论计算机缓存的工作机制. 科技信息, 2007 (33): 91-91,102.
- [26] 卢成均. 缓存机制及其在数据存取层中的应用模型研究. 计算机应用与软件, 2008.25 (12): 172-174,201.
- [27] 宋善德 郭飞. 基于 Java 的 Web 数据库连接池技术的研究. 计算机工程与应用, 2002.38 (8): 201-203,206.
- [28] 张佳阳. J2EE 项目中数据访问策略的选择. 电脑知识与技术: 学术交流, 2007 (6): 1207-1208,1213.
- [29] Apache iBATIS 官方网站. <http://ibatis.apache.org/>
- [30] Hibernate 官方网站. <http://www.hibernate.org>
- [31] Apache ojb 官方网站. <http://db.apache.org/ojb/>
- [32] Getting Started with the JDBC API. <http://java.sun.com/j2se/1.3/docs/guide/jdbc/getstart/GettingStartedTOC.fm.html>
- [33] commons-dbcp 官方网站. <http://commons.apache.org/dbcp/>



电子工业出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY
http://www.phei.com.cn

Broadview®
WWW.BROADVIEW.COM.CN

Csdn.NET

技术凝聚实力 专业创新出版

博文视点 (www.broadview.com.cn) 资讯有限公司是电子工业出版社、CSDN.NET、《程序员》杂志联合打造的专业出版平台, 博文视点致力于——IT专业图书出版, 为IT专业人士提供真正专业、经典的好书。

请访问 www.dearbook.com.cn (第二书店) 购买优惠价格的博文视点经典图书。

请访问 www.broadview.com.cn (博文视点的服务平台) 了解更多更全面的出版信息; 您的投稿信息在这里将会得到迅速的反馈。

博文本版精品汇聚



加密与解密 (第三版)

段钢 编著
ISBN 978-7-121-06644-3
定价: 69.00元
畅销书升级版, 出版一月销售10000册。
看雪软件安全学院众多高手, 合力历时4年精心打造。



疯狂Java讲义

新东方IT培训广州中心
软件教学总监 李刚 编著
ISBN 978-7-121-06646-7
定价: 99.00元 (含光盘1张)
用案例驱动, 将知识点融入实际项目的开发。
代码注释非常详细, 几乎每两行代码就有一行注释。



Windows驱动开发技术详解

张帆 等编著
ISBN 978-7-121-06846-1
定价: 65.00元 (含光盘1张)
原创经典, 威盛一线工程师倾力打造。
深入驱动核心, 剖析操作系统底层运行机制。



Struts 2权威指南

李刚 编著
ISBN 978-7-121-04853-1
定价: 79.00元 (含光盘1张)
可以作为Struts 2框架的权威手册。
通过实例演示Struts 2框架的用法。



你必须知道的.NET

王涛 著
ISBN 978-7-121-05891-2
定价: 69.80元
来自于微软MVP的最新技术心得和感悟。
将技术问题以生动易懂的语言展开, 层层深入, 以例说理。



Oracle数据库精讲与疑难解析

赵振平 编著
ISBN 978-7-121-06189-9
定价: 128.00元
754个故障重现, 件件源自工作的经验教训。
为专业人士提供的速查手册, 遇到故障不求人。



SOA原理·方法·实践

IBM资深架构师毛新生 主编
ISBN 978-7-121-04264-5
定价: 49.8元
SOA技术巅峰之作!
IBM中国开发中心技术经典呈现!



VC++深入详解

孙鑫 编著
ISBN 7-121-02530-2
定价: 89.00元 (含光盘1张)
IT培训专家孙鑫经典畅销力作!

博文视点资讯有限公司

电话: (010) 51260888 传真: (010) 51260888-802
E-mail: market@broadview.com.cn (市场)
editor@broadview.com.cn jnj@phei.com.cn (投稿)
通信地址: 北京市万寿路173信箱 北京博文视点资讯有限公司
邮编: 100036

电子工业出版社发行部

发行部: (010) 88254055
门市部: (010) 68279077 68211478
传真: (010) 88254050 88254060
通信地址: 北京市万寿路173信箱
邮编: 100036

博文视点 · IT出版旗舰品牌



电子工业出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY
http://www.phei.com.cn

Broadview®
WWW.BROADVIEW.COM.CN

Csdn.NET

技术凝聚实力 专业创新出版

博文视点 (www.broadview.com.cn) 资讯有限公司是电子工业出版社、CSDN.NET、《程序员》杂志联合打造的专业出版平台, 博文视点致力于——IT专业图书出版, 为IT专业人士提供真正专业、经典的好书。

请访问 www.dearbook.com.cn (第二书店) 购买优惠价格的博文视点经典图书。

请访问 www.broadview.com.cn (博文视点的服务平台) 了解更多更全面的出版信息; 您的投稿信息在这里将会得到迅速的反馈。

博文典藏



博文视点资讯有限公司

电话: (010) 51260888 传真: (010) 51260888-802

E-mail: market@broadview.com.cn (市场)

editor@broadview.com.cn jsj@phei.com.cn (投稿)

通信地址: 北京市万寿路173信箱 北京博文视点资讯有限公司

邮编: 100036

电子工业出版社发行部

发行部: (010) 88254055

门市部: (010) 68279077 68211478

传真: (010) 88254050 88254060

通信地址: 北京市万寿路173信箱

邮编: 100036

博文视点 · IT出版旗舰品牌



《iBATIS 框架源码剖析》读者交流区

尊敬的读者：

感谢您选择我们出版的图书，您的支持与信任是我们持续上升的动力。为了使您能通过本书更透彻地了解相关领域，更深入的学习相关技术，我们将特别为您提供一系列后续的服务，包括：

1. 提供本书的修订和升级内容、相关配套资料；
2. 本书作者的见面会信息或网络视频的沟通活动；
3. 相关领域的培训优惠等。

请您抽出宝贵的时间将您的个人信息和需求反馈给我们，以便我们及时与您取得联系。

您可以任意选择以下三种方式与我们联系，我们都将记录和保存您的信息，并给您提供不定期的信息反馈。

1. 短信

您只需编写如下短信：B10872+您的需求+您的建议

发送到1066 6666 789（本服务免费，短信资费按照相应电信运营商正常标准收取，无其他信息收费）

为保证我们对您的服务质量，如果您在发送短信24小时后，尚未收到我们的回复信息，请直接拨打电话（010）88254369。

2. 电子邮件

您可以发邮件至jsj@phei.com.cn或editor@broadview.com.cn。

3. 信件

您可以写信至如下地址：北京万寿路173信箱博文视点，邮编：100036。

如果您选择第2种或第3种方式，您还可以告诉我们更多有关您个人的情况，及您对本书的意见、评论等，内容可以包括：

- （1）您的姓名、职业、您关注的领域、您的电话、E-mail地址或通信地址；
- （2）您了解新书信息的途径、影响您购买图书的因素；
- （3）您对本书的意见、您读过的同领域的图书、您还希望增加的图书、您希望参加的培训等。

如果您在后期想退出读者俱乐部，停止接收后续资讯，只需发送“B10872+退订”至10666666789即可，或者编写邮件“B10872+退订+手机号码+需退订的邮箱地址”发送至邮箱：market@broadview.com.cn 亦可取消该项服务。

同时，我们非常欢迎您为本书撰写书评，将您的切身感受变成文字与广大书友共享。我们将挑选特别优秀的作品转载在我们的网站（www.broadview.com.cn）上，或推荐至CSDN.NET等专业网站上发表，被发表的书评的作者将获得价值50元的博文视点图书奖励。

我们期待您的消息！

博文视点愿与所有爱书的人一起，共同学习，共同进步！

通信地址：北京万寿路 173 信箱 博文视点（100036）

电话：010-51260888

E-mail：jsj@phei.com.cn， editor@broadview.com.cn

www.phei.com.cn

www.broadview.com.cn

反侵权盗版声明

电子工业出版社依法对本作品享有专有出版权。任何未经权利人书面许可，复制、销售或通过信息网络传播本作品的行为；歪曲、篡改、剽窃本作品的行为，均违反《中华人民共和国著作权法》，其行为人应承担相应的民事责任和行政责任，构成犯罪的，将被依法追究刑事责任。

为了维护市场秩序，保护权利人的合法权益，我社将依法查处和打击侵权盗版的单位和个人。欢迎社会各界人士积极举报侵权盗版行为，本社将奖励举报有功人员，并保证举报人的信息不被泄露。

举报电话：(010) 88254396；(010) 88258888

传 真：(010) 88254397

E-mail: dbqq@phei.com.cn

通信地址：北京市万寿路 173 信箱

电子工业出版社总编办公室

邮 编：100036

数字出版
PDG

[General Information]

书名=iBATIS框架源码剖析

作者=任钢著

页码=524

定价=¥ 79.00

SS号=

dxNumber=000006916045

出版时间=2010.06

出版社=电子工业出版社

试读地址=<http://book.duxiu.com/bookDetail.jsp?dxNumber=000006916045&d=2855805AB7FC21558F2957566840EF8B&fenlei=181704&sw=iBATIS%BF%F2%BC%DC%D4%B4%C2%EB%C6%CA%CE%F6>