iBATIS SQL Maps

开发指南

Version 2.0

2004 年 6 月 17 日

Clinton Begin 著

刘 涛 译

目 录

简介

使用 SQL Map，能够大大减少访问关系数据库的代码。SQL Map 使用简单的 XML 配置文件将 Java Bean 映射成 SQL 语句，对比其他的数据库持续层和 ORM 框架（如 JDO 的实现，Hibernate 等），SQL Map 最大的优点在于它简单易学。要使用 SQL Map，只要熟悉 Java Bean，XML 和 SQL，就能使您充分发挥 SQL 语句的能力。

关于本文

本文讨论了 **iBATIS SQL Map** 最重要的特性。本文中没有提及的其他特性，可能以后不再支持或不久将会修改，并且修改时不作通告，因此最好不要使用它们。本文将随着 iBatis SQL Map 的修改而变更。如果您发现其中的错误，或是觉得某些地方难以理解，请发 email 至 clinton.begin@ibatis.com 。

本文是《iBatis SQL Maps Developer Guide》的中文版，仅供读者参考，最权威的应以 Clinton Begin 的官方文档为准。如果中文翻译有错误，请通知译者（email：toleu@21cn.com，Blog：http://starrynight.blogdriver.com/）。

SQL Maps (com.ibatis.sqlmap.*)

SQL Map 的概念

SQL Map API 让开发人员可以轻易地将 Java Bean 映射成 PreparedStatement 的输入参数和 ResultSet 结果集。开发 SQL Map 的想法很简单：提供一个简洁的架构，能够用 20%的代码实现 80%JDBC 的功能。

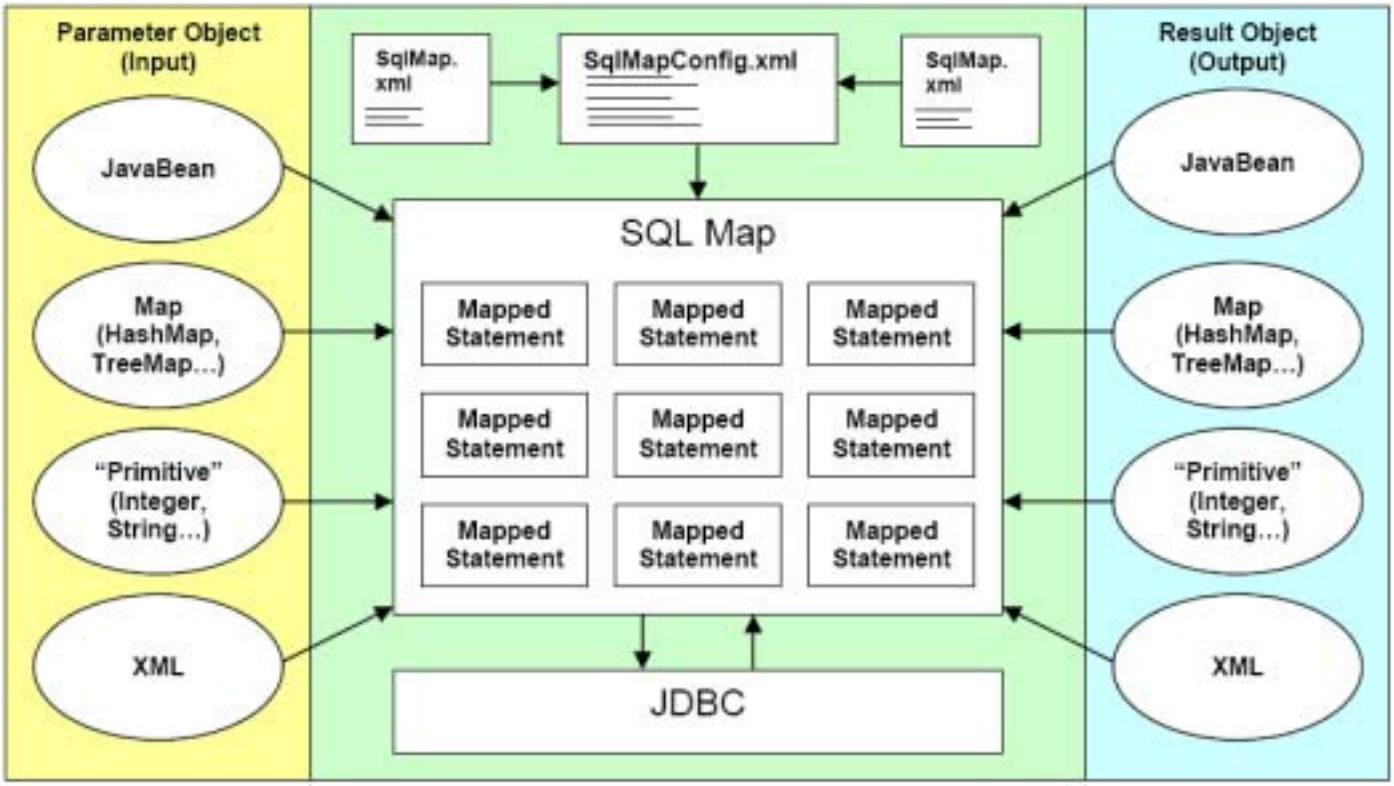
SQL Map 如何工作？

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 PreparedStatement。以下流程描述了 SQL Maps 的高层生命周期：

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），参数对象将为 SQL 修改语句和查询语句设定参数值。

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。和参数对象一样，结果对象可以是 Java Bean，Map 实现和基本数据类型的包装类。

下图描述了以上的执行流程。



安装 SQL Maps

安装 SQL Maps 很简单，只要把相关的 JAR 文件复制到类路径下即可。类路径或者是 JVM 启动是指定的类路径（java 命令参数），或者是 Web 应用中的/WEB-INF/lib 目录。Java 类路径的详尽讨论超出了本文的范围，如果您是 Java 的初学者，请参考以下的资源：

http://java.sun.com/j2se/1.4/docs/tooldocs/win32/classpath.html

http://java.sun.com/j2se/1.4.2/docs/api/java/lang/ClassLoader.html

http://java.sun.com/j2se/1.4.2/docs/

安装 iBatis 需要在类路径下放置以下 JAR 文件：

文件名	描述	是否必需
ibatis-common.jar	IBATIS 公用的工具类	是
ibatis-sqlmap.jar	IBATIS SQL Maps 框架	是
ibatis-dao.jar	IBATIS DAO 框架	否

JAR 文件和依赖性

如果一个框架依赖于太多的 JAR 文件，就很难与其他的应用和框架集成。IBATIS 2.0 的一个主要的关注点是管理并降低 JAR 文件的依赖性。因此，如果您用的是 JDK1.4，IBATIS

仅仅依赖于 Jakarta Commons Logging 框架。可选的 JAR 文件放在发布版的/lib/optional 目录下。它们根据功能来分类。下面列表总结了何时需要使用可选 JAR 类库。

描述	何时使用	目录
1.4 以前 JDK 版本支持	如果您使用的 JDK 版本低于 1.4，并且您的应用服务器不提供这些 JAR 文件，您将需要这些可选的 JAR 文件	/lib/optional/**jdbc** /lib/optional/**jta** /lib/optional/**xml**
IBATIS 的向后兼容	如果您使用旧的 IBATIS（1.x）DAO 框架，或旧的 SQL Maps（1.x）	/lib/optional/**compatibility**
运行时字节码增强	如果您需要使用 CGLIB2.0 字节码增强来提高 lazy loading 和 reflection 的性能	/lib/optional/**enhancement**
DataSource 实现	如果您使用 Jakarta DBCP 连接池	/lib/optional/**dbcp**
分布式缓存	如果您使用 OSCache 来支持集中或分布式缓存	/lib/optional/**caching**
Log4J 日志	如果您需要使用 Log4J	/lib/optional/**logging**

从 1.x 版本升级

是否应该升级

判断您是否需要升级的最好办法是尝试它。下面是几种升级的方法。

1. 版本 2.0 几乎完全保持和 1.x 版本的向后兼容，因此某些情况下只需用新的 JAR 文件替代旧的即可。这个升级方法带来的好处最少，但最简单。您无需修改 XML 文件或 Java 代码，但会存在某些不兼容的问题。
2. 第二种方法是把 1.x 的 XML 文件转换成 2.0 规范，但仍使用 1.x 的 Java API。除了 XML 映射文件存在着细微的不兼容之处外，这是个安全的方法。SQL Map 框架包括了用来转换 XML 映射文件的 ANT Task（参见下节）。
3. 第三种方法是转换 XML 文件（和第二种方法相同）和 Java 代码。因为没有转换 Java 代码的工具，必须手工进行。
4. 第四种方法是不必升级。如果您升级有困难，可以让应用继续使用 1.x 版本。让旧应用继续使用 1.x 版，在新应用中使用 2.0 版是个不错的主意。

转换XML配置文件（从1.x到2.0）

框架的 2.0 版本包含了一个可以在 ANT 构建环境中使用的 XML 文件转换器。虽然转换

XML 配置文件是可选的，但将 1.x 的配置文件转换成 2.0 仍然是个好主意。你几乎不会遇到不兼容的文件，并且还可以使用 2.0 版本新的特性（即使您继续使用 1.x 的 Java API）。

XML 配置文件转换器在 build.xml 文件中的例子如下：

```
<taskdef name="convertSqlMaps"
    classname="com.ibatis.db.sqlmap.upgrade.ConvertTask"
    classpathref="classpath"/>
<target name="convert">
    <convertSqlMaps todir="D:/targetDirectory/" overwrite="true">
        <fileset dir="D/sourceDirectory/">
            <include name="**/maps/*.xml"/>
        </fileset>
    </convertSqlMaps>
</target>
```

就像您看到的一样，它和 Ant 的 copy task 很相似。事实上它就是 Ant 的 copy task 类的子类，因此您可以用这个 task 完成任何 copy task 的功能（详细信息请参考 Ant 的 Copy task 文档）

使用新的JAR文件

要升级到 2.0，最好删除 iBatis 原有旧的 JAR 文件及其依赖 JAR 类库，并用新的 JAR 文件替代。但要主要不要删除其他组件或框架还需要的文件。请参考上节关于 JAR 类库及其依赖性的讨论。

下表总结了旧文件及其相应的新文件。

旧文件	新文件
ibatis-db.jar 1.2.9b 以后的版本，这个文件被分拆成一下 3 个文件 ibatis-common.jar ibatis-dao.jar ibatis-sqlmap.jar	ibatis-common.jar（必需） ibatis-sqlmap.jar（必需） ibatis-dao.jar（可选）
commons-logging.jar commons-logging-api.jar commons-collection.jar commons-dbcp.jar commons-pool.jar oscache.jar jta.jar jdbc2_0-stdext.jar xercesImpl.jar xmlParserAPIs.jar	commons-logging-1-0-3.jar（必需） commons-collection-2-1.jar（可选） commons-dbcp-1-1.jar（可选） commons-pool-1-1.jar（可选） oscache-2-0-1.jar（可选） jta-1-0-1a.jar（可选） jdbc2_0-stdext.jar（可选） xercesImpl-2-4-0.jar（可选） xmlParserAPIs-2-4-0.jar（可选） xalan-2-5-2.jar（可选）

jdom.jar	log4j-1.2.8.jar（可选） cglib-full-2-0-rc2.jar（可选）

本文余下部分将介绍如何使用 SQL Maps 框架。

SQL Map XML 配置文件

SQL Map 使用 XML 配置文件统一配置不同的属性，包括 DataSource 的详细配置信息，SQL Map 和其他可选属性，如线程管理等。以下是 SQL Map 配置文件的一个例子：

SqlMapConfig.xml

```
<?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">
  <!-- Always ensure to use the correct XML header as above! -->
  <sqlMapConfig>
  <!-- The properties (name=value) in the file specified here can be used placeholders in this
  config file (e.g. “${driver}”. The file is relative to the classpath and is completely optional. -->
  <properties resource=" examples/sqlmap/maps/SqlMapConfigExample.properties " />

  <!-- These settings control SqlMapClient configuration details, primarily to do with transaction
  management. They are all optional (more detail later in this document). -->
  <settings
        cacheModelsEnabled="true"
        enhancementEnabled="true"
        lazyLoadingEnabled="true"
        maxRequests="32"
        maxSessions="10"
        maxTransactions="5"
        useStatementNamespaces="false"
  />
  <!-- Type aliases allow you to use a shorter name for long fully qualified class names. -->
  <typeAlias alias="order" type="testdomain.Order"/>

  <!-- Configure a datasource to use with this SQL Map using SimpleDataSource.
  Notice the use of the properties from the above resource -->
  <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>

    <!-- Identify all SQL Map XML files to be loaded by this SQL map. Notice the paths
    are relative to the classpath. For now, we only have one… -->
    <sqlMap resource="examples/sqlmap/maps/Person.xml" />
</sqlMapConfig>
```

以下详细讨论 SQL Map 配置文件的各组成部分。

<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 SQL Map 映射文件中引用。例如，如果属性文件中包含属性：

```
driver=org.hsqldb.jdbcDriver
```

SQL Map 配置文件及其每个映射文件都可以使用占位符${driver}来代表值 org.hsqldb.jdbcDriver。例如：

```
<property name="JDBC.Driver" value="${driver}"/>
```

这个元素在开发，测试和部署各阶段都很有用。它可以使在多个不同的环境中重新配置应用和使用自动生成工具（如 ANT）变得容易。属性文件可以从类路径中加载（使用 resource 熟悉)，也可以从合法的 URL 中加载（使用 url 属性)。例如，要加载固定路径的属性文件，使用：

```
<properties url=”file:///c:/config/my.properties” />
```

<setting>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。下表列出了<setting>元素支持的属性及其功能：

maxRequests	同时执行 SQL 语句的最大线程数。大于这个值的线

	程将阻塞直到另一个线程执行完成。不同的 DBMS 有不同的限制值，但任何数据库都有这些限制。通常这个值应该至少是 maxTransactions（参见以下）的 10 倍，并且总是大于 maxSessions 和 maxTranactions。减小这个参数值通常能提高性能。 例如：maxRequests=“256” 缺省值：512
maxSessions	同一时间内活动的最大 session 数。一个 session 可以是代码请求的显式 session，也可以是当线程使用 SqlMapClient 实例（即执行一条语句）自动获得的 session。它应该总是大于或等于 maxTransactions 并小于 maxRequests。减小这个参数值通常能减少内存使用。 例如：maxSessions=“64” 缺省值：128
maxTransactions	同时进入 SqlMapClient.startTransaction()的最大线程数。大于这个值的线程将阻塞直到另一个线程退出。不同的 DBMS 有不同的限制值，但任何数据库都有这些限制。这个参数值应该总是小于或等于 maxSessions 并总是远远小于 maxRequests。减小这个参数值通常能提高性能。 例如：maxTransactions=“16” 缺省值：32
cacheModelsEnabled	全局性地启用或禁用 SqlMapClient 的所有缓存 model。调试程序时使用。 例如：cacheModelsEnabled=“true” 缺省值：true（启用）
lazyLoadingEnabled	全局性地启用或禁用SqlMapClient的所有延迟加载。调试程序时使用。 例子：lazyLoadingEnabled=“true” 缺省值：true（启用）
enhancementEnabled	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。 例子：enhancementEnabled=“true” 缺省值：false（禁用）
useStatementNamespaces	如果启用本属性，必须使用全限定名来引用 mapped statement。Mapped statement 的全限定名由 sql-map 的名称和 mapped-statement 的名称合成。例如：

	queryForObject(“sqlMapName.statementName”); 例如：useStatementNamespaces= “ false ” 缺省值：false（禁用）

<typeAlias>元素

<typeAlias>元素让您为一个通常较长的、全限定类名指定一个较短的别名。例如：

```
<typeAlias alias="shortname" type="com.long.class.path.Class"/>
```

在 SQL Map 配置文件预定义了几个别名。它们是：

事务管理器别名	
JDBC	com.ibatis.sqlmap.engine.transaction.jdbc.JdbcTransactionConfig
JTA	com.ibatis.sqlmap.engine.transaction.jta.JtaTransactionConfig
EXTERNAL	com.ibatis.sqlmap.engine.transaction.external.ExternalTransactionConfig
Data Source Factory 别名	
SIMPLE	com.ibatis.sqlmap.engine.datasource.SimpleDataSourceFactory
DBCP	com.ibatis.sqlmap.engine.datasource.DbcpDataSourceFactory
JNDI	com.ibatis.sqlmap.engine.datasource.JndiDataSourceFactory

<transactionManager>元素

1.0 转换注意事项：SQL Map 1.0 允许配置多个数据源。这引起了一些不好的实践。因此，2.0 版本只允许一个数据源。要使用多个部署/配置参数，您最好使用多个属性文件，不同的系统使用不同的属性文件，或在创建 SQL Map 时传入不同的属性文件（参见下面的 Java API）。

<transationManager>元素让您为 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。

- 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。*参见以下 JNDI 数据源的例子。*
- EXTERNAL：这个配置可以让您自己管理事务。您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)。

<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，但您也可以添加自己的实现。下面详细地讨论 DataSourceFactory 的三个实现及其例子。

SimpleDataSourceFactory

SimpleDataSourceFactory 为 DataSource 提供了一个基本的实现，适用于在没有 J2EE 容器提供 DataSource 的情况。它基于 iBatis 的 SimpleDataSource 连接池实现。

```
<transactionManager type="JDBC">
  <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>
</transactionManager>
```

DbcpDataSourceFactory

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 中必须要配置的参数例子如下：

```
<transactionManager type="JDBC">
  <dataSource type="DBCP">
    <property name="JDBC.Driver" value="${driver}"/>
    <property name="JDBC.ConnectionURL" value="${url}"/>
    <property name="JDBC.Username" value="${username}"/>
    <property name="JDBC.Password" value="${password}"/>
    <!-- OPTIONAL PROPERTIES BELOW -->
    <property name="Pool.MaximumActiveConnections" value="10"/>
```

```
      <property name="Pool.MaximumIdleConnections" value="5"/>
      <property name="Pool.MaximumWait" value="60000"/>
      <!-- Use of the validation query can be problematic.
      If you have difficulty, try without it. -->
      <property name="Pool.ValidationQuery" value="select * from ACCOUNT"/>
      <property name="Pool.LogAbandoned" value="false"/>
      <property name="Pool.RemoveAbandoned" value="false"/>
      <property name="Pool.RemoveAbandonedTimeout" value="50000"/>
    </datasource>
</transactionManager>
```

JndiDataSourceFactory

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：

```
<transactionManager type="JDBC" >
  <dataSource type="JNDI">
  <property name="DataSource" value="java:comp/env/jdbc/jpetstore"/>
  </dataSource>
</transactionManager>
```

以上配置使用了常用的 JDBC 事务管理。但对于容器管理的资源，您可能需要象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。

<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。您必须在这里指定所有的 SQL Map 文件。例子如下：

```
    <!-- CLASSPATH RESOURCES -->
```

```
<sqlMap resource="com/ibatis/examples/sql/Customer.xml" />
<sqlMap resource="com/ibatis/examples/sql/Account.xml" />
<sqlMap resource="com/ibatis/examples/sql/Product.xml" />
<!-- URL RESOURCES -->
<sqlMap url="file:///c:/config/Customer.xml " />
<sqlMap url="file:///c:/config/Account.xml " />
<sqlMap url="file:///c:/config/Product.xml" />
```

以下几个章节详细讨论 SQL Map XML 映射文件的结构。

SQL Map XML 映射文件

在上面的例子中，只使用了 SQL Map 最简单的形式。SQL Map 的结构中还有其他更多的选项。这里是一个 mapped statement 较复杂的例子，使用了更多的特性。

```
<sqlMap id="Product">
  <cacheModel id="productCache" type="LRU">
      <flushInterval hours="24"/>
      <property name="size" value="1000" />
  </cacheModel>
  <typeAlias alias="product" type="com.ibatis.example.Product" />
  <parameterMap id="productParam" class="product">
      <parameter property="id"/>
  </parameterMap>
  <resultMap id="productResult" class="product">
      <result property="id" column="PRD_ID"/>
      <result property="description" column="PRD_DESCRIPTION"/>
  </resultMap>
  <select id="getProduct" parameterMap="productParam" resultMap="productResult"
  cacheModel="product-cache">
      select * from PRODUCT where PRD_ID = ?
  </select>
</sqlMap>
```

太麻烦？虽然框架为您做了很多工作，为了完成一个简单的查询操作，依然需要做很多。别担心，下面是一个简洁版本。

```
<sqlMap id="Product">
  <select id="getProduct" parameterClass=" com.ibatis.example.Product"
          resultClass="com.ibatis.example.Product">
      select
      PRD_ID as id,
      PRD_DESCRIPTION as description
      from PRODUCT
      where PRD_ID = #id#
  </select>
</sqlMap>
```

但是简洁版本的行为和前一个声明的行为不太一样。首先，简洁版本没有定义缓存，因此每一个请求都要读取数据库。其次，简洁版本使用了框架的自动映射特性，这将带来一些副作用。但是，这两者在 Java 代码中的执行方式完全一致，因此您开始可以先使用一个简

单的方案，等将来需要时再换成更好的版本。首先应用最简单的方案，是很多现代方法学的最佳实践。

注意！一个 SQL Map XML 映射文件可以包含任意多个 Mapped Statement，Parameter Map 和 Result Map。按照它们之间的逻辑关系，为您的应用合理地组织 Mapped Statement，Parameter Map 和 Result Map。

注意！SQL Map 的名称是全局性的，在所有的 SQL Map 文件中名称必须是唯一的。

Mapped Statements

SQL Map 的核心概念是 Mapped Statement。Mapped Statement 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，Mapped Statement 可以使用 Java 类来作为 parameter 和 result。Mapped Statement 也可以使用缓存模型，在内存中缓存常用的数据。Mapped Statement 的结构如下所示：

```
<statement  id="statementName"
    [parameterClass="some.class.Name"]
    [resultClass="some.class.Name"]
    [parameterMap="nameOfParameterMap"]
    [resultMap="nameOfResultMap"]
    [cacheModel="nameOfCache"]
>
    select * from PRODUCT where PRD_ID = [?|#propertyName#]
    order by [$simpleDynamic$]
</statement>
```

在上面的表达式中，括号[]里的部分时可选的属性，并且在某些情况下只有特定的组合才是合法的。因此下面这个简单的例子也是正确的：

```
<statement id="insertTestProduct" >
    insert into PRODUCT (PRD_ID, PRD_DESCRIPTION) values (1, "Shih Tzu")
</statement>
```

上面的例子不太可能出现，但如果只是简单地想用 SQL Map 架构来执行任意地的 SQL 语句，这种写法就派上用场了。但常见的用法还是通过 Parameter Map 和 Result Map，来实现 Java Bean 映射的特性，这是 SQL Map 真正有价值的地方。

Statement的类型

<statement>元素是个通用声明，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 statement 类型提供了更直观的 XML DTD，并拥有某些

<statement>元素没有的特性。下表总结了 statement 类型及其属性和特性：

Statement 类型	属性	子元素	方法
<statement>	id parameterClass resultClass parameterMap resultMap cacheModel xmlResultName	所有的动态元素	insert update delete 所有的查询方法
<insert>	id parameterClass parameterMap	所有的动态元素 <selectKey>	insert update delete
<update>	id parameterClass parameterMap	所有的动态元素	insert update delete
<delete>	id parameterClass parameterMap	所有的动态元素	insert update delete
<select>	id parameterClass resultClass parameterMap resultMap cacheModel	所有的动态元素	所有的查询方法
<procedure>	id parameterClass resultClass parameterMap resultMap xmlResultName	所有的动态元素	insert update delete 所有的查询方法

SQL 语句

SQL 显然是 mapped statement 中最重要的部分，可以使用对于数据库和 JDBC Driver 合法的任意 SQL 语句。只要 JDBC Driver 支持，可以使用任意的函数，甚至是多条语句。因为 SQL 语句是嵌在 XML 文档中的，因此有些特殊的字符不能直接使用，例如大于号和小于号（<>）。幸运的是，解决的办法很简单，只需将包含特殊字符的 SQL 语句放在 XML 的 CDATA 区里面就可以了。例如：

```
<statement id="getPersonsByAge" parameterClass=”int”
  resultClass="examples.domain.Person">
    <![CDATA[
```

```
    SELECT *
    FROM PERSON
    WHERE AGE > #value#
    ]]>
</statement>
```

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。不过这通常（并不总是）是个私有的特性。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）。下面是两个例子：

```
<
!—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>
<!— 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>
```

存储过程

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 email

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。

注意！要确保始终只使用 JDBC 标准的存储过程语法。参考 JDBC 的 CallableStatement 文档以获得更详细的信息。

parameterClass

parameterClass 属性的值是 Java 类的全限定名（即包括类的包名）。parameterClass 属性是可选的，但强烈建议使用。它的目的是限制输入参数的类型为指定的 Java 类，并优化框架的性能。如果您使用 parameterMap，则没有必要使用 parameterClass 属性。例如，如果要只允许 Java 类“ examples.domain.Product ”作为输入参数，可以这样作：

```
<statement id=”statementName” parameterClass=” examples.domain.Product”>
    insert into PRODUCT values (#id#, #description#, #price#)
</statement>
```

重要提示：虽然 2.0 向后兼容，但强烈建议使用 parameterClass（除非没必要）。通过提供 parameterClass，您可以获得更好的性能，因为如果框架事先知道这个类，就可以优化自身的性能。

如果不指定 parameterClass 参数，任何带有合适属性（get/set 方法）的 Java Bean 都可以作为输入参数。

parameterMap

属性 parameterMap 的值等于一个预先定义的<parameterMap>元素的名称。parameterMap 属性很少使用，更多的是使用上面的 parameterClass 和 inline parameter（接下来会讨论）。

注意！动态 mapped statement 只支持 inline parameter，不支持 parameter map。

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：

```
<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>
```

上面的例子中，parameter map 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。Parameter Map 及其选项将在以后详细讨论。

Inline Parameter简介

现在简单介绍一下 inline parameter，详细讨论见后面章节。Inline parameter 可以嵌在 mapped statement 内部使用。例如：

```
<statement id=”insertProduct” >
    insert into PRODUCT (PRD_ID, PRD_DESCRIPTION)
    values (#id#, #description#);
</statement>
```

以上例子中，内嵌的参数是#id#和#description#。每个参数代表一个 Java Bean 属性，用于给 SQL 语句中相应的位置赋值。上面例子中，Product 对象的 id 和 description 属性的值将会替换 SQL 语句中相应的符号。因此，对于 id=5，description=‘ dog ’的 Product 对象，SQL 语句变为：

```
insert into PRODUCT (PRD_ID, PRD_DESCRIPTION)
values (5, ‘dog’);
```

resultClass

resultClass 属性的值是 Java 类的全限定名（即包括类的包名）。resultClass 属性可以让您指定一个 Java 类，根据 ResultSetMetaData 将其自动映射到 JDBC 的 ResultSet。只要是 Java Bean 的属性名称和 ResultSet 的列名匹配，属性自动赋值给列值。这使得查询 mapped statement 变得很短。例如：

```
<statement 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#
</statement>
```

在上面的例子中，Person 类拥有属性包括 id，firstName，lastName，birthDate，weightInKilograms 和 heightInMeters。每一个属性对应 SQL 查询语句一个列的别名（使用‘ as ”关键字－标准的 SQL 语法）。一般情况下，列名和属性名称不匹配，就需要使用“ as ”关键字。当执行 mapped statement 时，Person 类将被初始化，从结果集中得到的列值将根据属性名和列名映射成 Person 对象的属性值。

正如以前所说，使用 resultClass 的自动映射存在一些限制，无法指定输出字段的数据类

型（如果需要的话），无法自动装入相关的数据（复杂属性），并且因为需要 ResultSetMetaData 的信息，会对性能有轻微的不利影响。但使用 resultMap，这些限制都可以很容易解决。

resultMap

resultMap 是最常用和最重要的属性。ResultMap 属性的值等于预先定义的 resultMap 元素的 name 属性值（参照下面的例子）。使用 resultMap 可以控制数据如何从结果集中取出，以及哪一个属性匹配哪一个字段。不象使用 resultClass 的自动映射方法，resultMap 属性可以允许指定字段的数据类型，NULL 的替代值复杂类型映射（包括其他 Java Bean，集合类型和基本类型包装类）。

关于 resultMap 的详细讨论放在以后的章节，这里只给出一个相关 statement 的 resultMap 的例子。

```
<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>
```

上面的例子中，通过 resultMap 的定义，查询语句得到的 ResultSet 被映射成 Product 对象。resultMap 定义的“ id ”属性值将赋予“ PRO_ID ”字段值，而“ description ”属性值将赋予“ PRD_DESCRIPTION ”字段值。注意 resultMap 支持“ select * ”，并不要求定义 ResultSet 所有返回字段的映射。

cacheModel

cacheModel 的属性值等于指定的 cacheModel 元素的 name 属性值。属性 cacheModel 定义查询 mapped statement 的缓存。每一个查询 mapped statement 可以使用不同或相同的 cacheModel。详细讨论见后面的章节，以下只给出个例子。

```
<cacheModel id="product-cache" imlementation="LRU">
    <flushInterval hours="24"/>
    <flushOnExecute statement="insertProduct"/>
    <flushOnExecute statement="updateProduct"/>
    <flushOnExecute statement="deleteProduct"/>
    <property name="size" value="1000" />
</cacheModel>
<statement id="getProductList" parameterClass="int" cacheModel="product-cache">
    select * from PRODUCT where PRD_CAT_ID = #value#
```

```
</statement>
```

上面例子中，“ getProductList ”的缓存使用 WEAK 引用类型，每 24 小时刷新一次，或当更新的操作发生时刷新。

xmlResultName

当直接把查询结果映射成 XML document 时，属性 xmlResultName 的值等于 XML document 根元素的名称。例如：

```
<select id="getPerson" parameterClass=”int” resultClass="xml" xmlResultName=”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>
```

上面的查询结果将产生一个 XML document，结构如下：

```
<person>
    <id>1</id>
    <firstName>Clinton</firstName>
    <lastName>Begin</lastName>
    <birthDate>1900-01-01</birthDate>
    <weightInKilograms>89</weightInKilograms>
    <heightInMeters>1.77</heightInMeters>
</person>
```

Parameter Map 和 Inline Parameter

如上所述，parameterMap 负责将 Java Bean 的属性映射成 statement 的参数。虽然 parameterMap 的外部形式很少使用，理解它如何工作对于理解 inline parameter 还是很有帮助。本节以下部分将详细讨论 inline parameter。

```
<parameterMap id=”parameterMapName” [class=”com.domain.Product”]>
    <parameter property =”propertyName” [jdbcType=”VARCHAR”] [javaType=”string”]
            [nullValue=”NUMERIC”] [null=”-9999999”]/>
    <parameter …… />
    <parameter …… />
</parameterMap>
```

括号[]是可选的属性。parameterMap 元素只要求属性 ***id*** 作为唯一标识。属性 ***class*** 是可

选的但强烈推荐使用。和 parameterClass 类似，class 属性可以框架检查输入参数的类型并优化性能。

<parameter>元素

一个 parameterMap 可包含任意多的 parameter 元素。以下讨论 parameter 的各个属性。

- **property**

属性 property 是传给 statement 的参数对象的 Java Bean 属性名称。该名称根据需要，可以在 statement 中多次出现（即在 SQL 语句 SET 子句中被更新的属性，也可以作为条件出现在 WHERE 子句中）。

- **jdbcType**

属性 jdbcType 用于显式地指定给本属性（property）赋值的数据库字段的数据类型。对于某些特定的操作，如果不指定字段的数据类型，某些 JDBC Driver 无法识别字段的数据类型。一个很好的例子是 PreparedStatement.setNull(int parameterIndex, int sqlType)方法，要求指定数据类型。如果不指定数据类型，某些 Driver 可能指定为 Types.Other 或 Types.Null。但是，不能保证所有的 Driver 都表现一致。对于这种情况，SQL Map API 允许使用 parameterMap 子元素 parameter 的 jdbcType 属性指定数据类型。

正常情况下，只有当字段可以为 NULL 时才需要 jdbcType 属性。另一需要指定 jdbcType 属性的情况是字段类型为日期时间类型的情况。因为 Java 只有一个 Date 类型（java.util.Date），而大多数 SQL 数据库有多个－通常至少有 3 种。因此，需要指定字段类型是 DATE 还是 DATETIME。

属性 jdbcType 可以是 JDBC Types 类中定义的任意参数的字符串值。虽然如此，还是有某些类型不支持（即 BLOB）。本节的稍后部分会说明架构支持的数据类型。

注意！大多数 JDBC Driver 只有在字段可以为 NULL 时需要指定 jdbcType 属性。因此，对于这些 Driver，只是在字段可以为 NULL 时才需要指定 type 属性。

注意！当使用 Oracle Driver 时，如果没有给可以为 NULL 的字段指定 jdbcType 属性，当试图给这些字段赋值 NULL 时，会出现“ Invalid column type ”错误。

- **javaType**

属性 javaType 用于显式地指定被赋值参数 Java 属性的类名。正常情况下，这可以通过反射从 Java Bean 的属性获得，但对于某些映射（例如 Map 和 XML document），框架不能通过这种方法来获知。如果没有设置 javaType，同时框架也不能获知类型信息，类型将被假定

为 Object。

- **nullValue**

属性 nullValue 的值可以是对于 property 类型来说任意的合法值，用于指定 NULL 的替换值。就是说，当 Java Bean 的属性值等于指定值时，相应的字段将赋值 NULL。这个特性允许在应用中给不支持 null 的数据类型（即 int，double，float 等）赋值 null。当这些数据类型的属性值匹配 null 值（即匹配-9999）时，NULL 将代替 null 值写入数据库。

- **<parameterMap>的例子**

以下是一个完整的例子。

```
<parameterMap id="insert-product-param" class="com.domain.Product">
    <parameter property="id" jdbcType="NUMERIC" javaType="int" nullValue="-9999999"/>
    <parameter property="description" jdbcType="VARCHAR" nullValue="NO_ENTRY"/>
</parameterMap>
<statement id="insertProduct" parameterMap="insert-product-param">
    insert into PRODUCT (PRD_ID, PRD_DESCRIPTION) values (?,?);
</statement>
```

上面的例子中，Java Bean 的属性“ id ”和“ description ”将按顺序地替换 Mapped Statement “ insertProduct ”的值符号（?）。因此，“ id ”赋值给第一个参数而“ description ”赋值给第二个参数。如果将顺序倒过来，XML 配置将变成：

```
<parameterMap id="insert-product-param" class="com.domain.Product">
    <parameter property="description" />
    <parameter property="id"/>
</parameterMap>
<statement id="insertProduct" parameterMap="insert-product-param">
    insert into PRODUCT (PRD_DESCRIPTION, PRD_ID) values (?,?);
</statement>
```

注意！parameterMap 并不自动地绑定到特定的 Java 类。因此在上面的例子中，任何拥有“ id ”和“ description ”属性的 Java Bean 对象，都可以作为 parameterMap 的输入。如果需要将输入绑定到特定的 Java 类，可以使用 mapped-statement 的 resultClass 属性。

注意！Parameter Map 的名称（name）局部的，只在定义它的 SQL Map XML 文件中有效。不过，加上 SQL Map 的名称（即在<sqlMap>根元素中的名称）作为前缀，您可以在另一个 SQL Map XML 文件中引用它。例如，要在另一个文件中引用以上的 parameterMap，可以使用名称“ Product.insert-product-param ”。

Inline Parameter Map

parameterMap 的语法虽然简单，但很繁琐。还有一种更受欢迎更灵活的方法，可以大

大简化定义和减少代码量。这种方法把 Java Bean 的属性名称嵌在 Mapped Statement 的定义中(即直接写在 SQL 语句中)。缺省情况下,任何没有指定 parameterMap 的 Mapped Statement 都会被解析成 inline parameter（内嵌参数)。用上面的例子（即 Product）来说，就是：

```
<statement id="insertProduct" parameterClass="com.domain.Product">
    insert into PRODUCT (PRD_ID, PRD_DESCRIPTION)
    values (#id#, #description#);
</statement>
```

在内嵌参数中指定数据类型可以用下面的语法：

```
<statement id="insertProduct" parameterClass="com.domain.Product">
    insert into PRODUCT (PRD_ID, PRD_DESCRIPTION)
    values (#id:NUMERIC#, #description:VARCHAR#);
</statement>
```

在内嵌参数中指定数据类型和 NULL 的替代值可以用这样的语法：

```
<statement id="insertProduct" parameterClass="com.domain.Product">
    insert into PRODUCT (PRD_ID, PRD_DESCRIPTION)
    values (#id:NUMERIC:-999999#, #description:VARCHAR:NO_ENTRY#);
</statement>
```

注意！在内嵌参数中，要指定 NULL 的替代值，必须要先指定数据类型。

注意 !如需要在查询时也使用 NULL 替代值,必须同时在 resultMap 中定义(如下说明)。

注意 !如果您需要指定很多的数据类型和 NULL 替代值,可以使用外部的 parameterMap 元素，这样会使代码更清晰。

基本类型输入参数

假如没必要写一个 Java Bean 作为参数，可以直接使用基本类型的包装类（即 String，Integer，Date 等）作为参数。例如：

```
<statement id="insertProduct" parameter="java.lang.Integer">
    select * from PRODUCT where PRD_ID = #value#
</statement>
```

假设 PRD_ID 的数据类型是 NUMERIC，要调用上面的 mapped statement，可以传入一个 java.lang.Integer 对象作为参数。Integer 对象的值将代替#value#参数。当使用基本类型包装类代替 Java Bean 时，切记要使用#value#作为参数。Result Map（参见以下章节）也支持使用基本类型作为结果参数。

为了使定义更简洁，基本类型可以使用别名。例如，可以用“ int ”来代替“ java.lang.Integer ”。这些别名参见下面的“ 支持 Parameter Map 和 Result Map 的数据类型 ”表格。

Map类型输入参数

假如没必要写一个 Java Bean 作为参数，而要传入的参数又不只一个时，可以使用 Map 类（如 HashMap，TreeMap 等）作为参数对象。例如：

```
<statement id="insertProduct" parameterClass="java.util.Map">
    select * from PRODUCT
    where PRD_CAT_ID = #catId#
    and PRD_CODE = #code#
</statement>
```

可以注意到 mapped statement 的形式完全没有区别！上面的例子中，如果把 Map 对象作为输入参数去调用 mapped statement，Map 对象必须包含键值“ catId ”和“ code ”。键值引用的对象必须是合适的类型，以上面的例子来说，必须是 Integer 和 String。Result Map（参见以下章节）也支持使用 Map 类型作为结果参数。要获得更多信息，请参见“ Result Map ”和“ 使用 SQL Map API 编程 ”部分。

Map 类型也可以使用别名。例如，可以用“ map ”来代替“ java.util.Map ”。这些别名参见下面的“ 支持 Parameter Map 和 Result Map 的数据类型 ”表格。

Result Map

在 SQL Map 框架中，Result Map 是极其重要的组件。在执行查询 Mapped Statement 时，resultMap 负责将结果集的列值映射成 Java Bean 的属性值。resultMap 的结构如下：

```
<resultMap id="resultMapName" class="some.domain.Class" [extends="parent-resultMap"]>
    <result property="propertyName" column="COLUMN_NAME"
        [columnIndex="1"] [javaType="int"] [jdbcType="NUMERIC"]
        [nullValue="-999999"] [select="someOtherStatement"]
    />
    <result ...../>
    <result ...../>
    <result ...../>
</resultMap>
```

括号[]中是可选的属性。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。和在 Java 中继承一个类相似，父 resultMap 的属性将作为子 resutlMap 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 Java Bean 的属性。属性的映射按它们在 resultMap 中定义的顺序进行。相关的 Java Bean 类必须符合 Java Bean 规范，每一属性都必须拥有 get/set 方法。

注意！ResultSet 的列值按它们在 resultMap 中定义的顺序读取（这特性会在某些实现得不是很好的 JDBC Driver 中派上用场）。

下面几个小节讨论 resultMap 的 result 元素各个属性：

- **property**

属性 property 的值是 mapped statement 返回结果对象的 Java Bean 属性的名称（get 方法）。

- **column**

属性 column 的值是 ResultSet 中字段的名称，该字段赋值给 names 属性指定的 Java Bean 属性。同一字段可以多次使用。注意，可能某些 JDBC Driver（例如，JDBC/ODBC 桥）不允许多次读取同一字段。

- **columnIndex**

属性 columnIndex 是可选的，用于改善性能。属性 columnIndex 的值是 ResultSet 中用于赋值 Java Bean 属性的字段次序号。在 99%的应用中，不太可能需要牺牲可读性来换取性能。使用 columnIndex，某些 JDBC Driver 可以大幅提高性能，某些则没有任何效果。

- **jdbcType**

属性 type 用于指定 ResultSet 中用于赋值 Java Bean 属性的字段的数据库数据类型（而不是 Java 类名）。虽然 resultMap 没有 NULL 值的问题，指定 type 属性对于映射某些类型（例如 Date 属性）还是有用的。因为 Java 只有一个 Date 类型，而 SQL 数据库可能有几个（通常至少有 3 个），为保证 Date（和其他）类型能正确的赋值，某些情况下指定 type 还是有必要的。同样地，String 类型的赋值可能来自 VARCHAR，CHAR 和 CLOB，因此同样也有必要指定 type 属性（取决于 JDBC Driver）。

- **javaType**

属性 javaType 用于显式地指定被赋值的 Java Bean 属性的类型。正常情况下，这可以通过反射从 Java Bean 的属性获得，但对于某些映射（例如 Map 和 XML document），框架不能通过这种方法来获知。如果没有设置 javaType，同时框架也不能获知类型信息，类型将被假定为 Object。

- **nullValue**

属性 nullValue 指定数据库中 NULL 的替代值。因此，如果从 ResultSet 中读出 NULL 值，Java Bean 属性将被赋值属性 null 指定的替代值。属性 null 的值可以指定任意值，但必须对于 Java Bean 属性的类型是合法的。

如果数据库中存在 NULLABE 属性的字段，但您需要用指定的常量代替 NULL，您可以这样设置 resultMap：

```
<resultMap id="get-product-result" class="com.ibatis.example.Product">
    <result property="id" column="PRD_ID"/>
    <result property="description" column="PRD_DESCRIPTION"/>
    <result property="subCode" column="PRD_SUB_CODE" nullValue="-999"/>
</resultMap>
```

上面的例子中，如果 PRD_SUB_CODE 的值是 NULL，subCode 属性将被赋值-999。这让您在 Java 类中用基本类型的属性映射数据库中的 NULLABE 字段。记住，如果您要在查询和更新中同样使用这个功能，必须同时在 parameterMap 中指定 nullValue 属性。

- **select**

属性 select 用于描述对象之间的关系，并自动地装入复杂类型（即用户定义的类型）属性的数据。属性 select 的值必须是另外一个 mapped statement 元素的名称。在同一个 result 元素中定义的数据库字段（column 属性）以及 property 属性，将被传给相关的 mapped statement 作为参数。因此，字段的数据类型必须是 SQL Map 支持的简单数据类型。关于简单数据类型和复杂类型之间映射/关系的信息，参照后面章节更详细的讨论。

隐式的Result Map

假如您有简单的映射，不需要重用定义好的 resultMap，有一个快速的方法，就是通过设定 mapped statement 的 resultClass 属性来隐式地指定 result map。诀窍在于，保证返回的 ResultSet 的字段名称（或标签或别名）和 Java Bean 中可写入属性的名称匹配。例如，考虑以上的 Product 类，可以创建一个带有隐式 result map 的 mapped statement 如下：

```
<statement id="getProduct" resultClass="com.ibatis.example.Product">
    select
    PRD_ID as id,
    PRD_DESCRIPTION as description
    from PRODUCT
    where PRD_ID = #value#
</statement>
```

上面的 mapped statement 定义了 resultClass 属性，并为每个字段指定了别名，用于匹配 Product 类的属性名称。这样就可以了，不需要 result map。缺点在于，您无法指定字段的数

据类型（通常不是 NULLABLE 字段不需要），或 NULL 替代值（或<result>别的属性）。另外还要记住，数据库很少是大小写敏感的，因此隐式 result map 对大小写也不敏感。假如您的 Java Bean 有两个属性，一个是 firstName，另一个是 firstname，数据库会把两者看作同一个属性，因而不能使用隐式的 result map（这也可以看作是 Java Bean 设计的一个潜在问题）。此外，使用 resultClass 的自动映射也对性能有轻微的不利影响。因为读取 ResultSetMetaData 信息会使某些 JDBC Driver 变慢。

基本类型的Result（即String，Integer，Boolean）

除了支持符合 Java Bean 规范的 Java 类，Result Map 还可以给基本类型包装类如 String，Integer，Boolean 等赋值。使用下面章节讨论的 API（参见 executeQueryForList()），Result Map 还可以得到基本类型包装类的集合。基本类型可以象 Java Bean 一样映射，只是要记住一个限制，基本类型只能有一个属性，名字可以任意取（常用“ value ”或“ val ”）。例如，如果您要获得所有产品描述的一个列表而不是整个 Product 类，Result Map 如下：

```
<resultMap id="get-product-result" class="java.lang.String">
    <result property="value" column="PRD_DESCRIPTION"/>
</resultMap>
```

更简单方法是，在 mapped statement 中使用 resultClass 属性（使用“ as ”关键字给字段取别名“ value ”）：

```
<statement id="getProductCount" resultClass="java.lang.Integer">
    select count(1) as value
    from PRODUCT
</statement>
```

Map类型的Result

Result Map 也可以方便为一个 Map（如 HashMap 或 TreeMap）对象赋值。使用下面讨论的 API（参见 executeQueryForList()），还可以得到 Map 对象的集合（即 Map 的 List）。Map 对象与 Java Bean 同样的方式映射，只是使用 name 属性值作为 Map 的键值，用它来索引相应的数据库字段值，而不是象 Java Bean 一样给属性赋值。例如，如果您要将 Product 对象的数据装入 Map，可以这样做：

```
<resultMap id="get-product-result" class="java.util.HashMap">
    <result property="id" column="PRD_ID"/>
    <result property="code" column="PRD_CODE"/>
    <result property="description" column="PRD_DESCRIPTION"/>
    <result property="suggestedPrice" column="PRD_SUGGESTED_PRICE"/>
</resultMap>
```

上面的例子会创建一个 HashMap 的实例并用 Product 的数据赋值。Property 的 name 属性值（即“ id ”）作为 HashMap 的键值，而列值则作为 HashMap 中相应的值。

当然，可以把 Map 类型 Result 和隐式的 Result Map 一起使用。例如：

```
<statement id=”getProductCount” resultClass=”java.util.HashMap”>
    select * from PRODUCT
</statement>
```

上面的例子中，结果集将以 Map 类型返回。

复杂类型属性（即自定义类型的属性）

因为 mapped statement 知道如何装入合适的数据和 Java 类，通过将 resultMap 的 property 和相应的 mapped statement 联系起来，可以自动地给复杂类型（即用户创建的类）的属性赋值。复杂类型用以表示在数据库中相互关系为一对一，一对多的数据。对于一对多的数据关系，拥有复杂类型属性的类作为“ 多 ”的一方，而复杂属性本身则作为“ 一 ”的一方。考虑下面的例子：

```
<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>
```

上面的例子中，Product 对象拥有一个类型为 Category 的 category 属性。因为 category 是复杂类型（用户定义的类型），JDBC 不知道如何给它赋值。通过将 category 属性值和另一个 mapped statement 联系起来，为 SQL Map 引擎如何给它赋值提供了足够的信息。通过执行“ getProduct ”，“ get-product-result ” Result Map 使用 PRD_CAT_ID 字段的值去调用“ getCategory ”。“ get-category-result ” Result Map 将初始化一个 Category 对象并赋值给它。然后整个 Category 对象将赋值给 Product 的 category 属性。

避免 N＋1 Select（1:1）

上面的方法存在一个问题，就是无论何时加载一个 Product，实际上都要执行两个 SQL 语句（分别加载 Product 和 Category）。只加载一个 Product 的情况下，这个问题似乎微不足道。但在执行一个获得 10 个 Product 的查询时，每得到一个 Product 都要分别执行一个加载 Category 的 SQL 语句。结果共执行了 11 次查询：一次用于得到一个 Product List，每得到一个 Product 对象都要执行另外一次查询，以获得相应的 Category 对象（N＋1，这个例子是 10＋1＝11）。

解决方法是，使用一个联合查询和嵌套的属性映射来代替两个查询 statement。上面例子的解决方案是：

```
<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.id" column="CAT_ID" />
    <result property="category.description" column="CAT_DESCRIPTION" />
</resultMap>
<statement id="getProduct" parameterClass="int" resultMap="get-product-result">
    select *
    from PRODUCT, CATEGORY
    where PRD_CAT_ID=CAT_ID
    and PRD_ID = #value#
</statement>
```

延迟加载 VS 联合查询（1:1）

必须要声明的是，使用联合查询的方案并不总是最好的。假如很少有必要访问相关的对象（如 Product 对象的 Category 属性），则不用联合查询加载所有的 Categor 属性可能更快。对于牵涉到外部连接或没有索引字段的数据库设计时，更是如此。在这种情况下，使用延迟加载和字节码增强选项的子查询，可能性能会更好。基本的原则是，如果您需要访问相关的对象，则使用联合查询。否则，使用延迟加载和字节码增强选项的子查询。

如果您不知道选择哪种方法，别担心。您可以随时更改选择而不会影响到 Java 代码。上面两个例子都得到相同的结果，并使用同样的调用方法。唯一要考虑的是，如果您要缓存查询结果，则使用子查询（而不是联合查询）来缓存查询结果。

复杂类型集合的属性

Result Map 还可以装入代表复杂类型对象集合（List）的属性，用以表示在数据库中相互关系为多对多或一对多的数据。拥有集合属性的类作为“一”的一方，而在集合中的对象作为“多”的一方。用来装入对象集合的 mapped statement 和上面例子一样。唯一的不同是，让 SQL Map 架构装入复杂类型集合（List）的业务对象的属性必须是 java.util.List 或 java.util.Collection 类型。例如，如果 Category 拥有 Product 对象的 List，mapped-statement 就像下面的例子（假设 Category 类有一个叫 productList 的属性，类型是 java.util.List）：

```
<resultMap id="get-category-result" class="com.ibatis.example.Category">
    <result property="id" column="CAT_ID"/>
    <result property="description" column="CAT_DESCRIPTION"/>
    <result property="productList" column="CAT_ID" select=" getProductsByCatId"/>
</resultMap>
<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="getCategory" parameterClass="int" resultMap="get-category-result">
    select * from CATEGORY where CAT_ID = #value#
</statement>
<statement id="getProductsByCatId" parameterClass="int" resultMap="get-product-result">
    select * from PRODUCT where PRD_CAT_ID = #value#
</statement>
```

避免 N＋1 Select（1:M 和 M:N）

类似于上面 1:1 的情况，但因为可能涉及到更大量的数据，因此问题甚至会更严重。上面例子存在的问题是，加载一个 Category 将执行两个 SQL 语句（分别加载一个 Category 和一个相关 Product 的 List）。只加载一个 Category 的情况下，这个问题似乎微不足道。但在执行一个获得 10 个 Category 的查询时，每得到一个 Category 都要分别执行一个加载 Product List 的 SQL 语句。一次用于得到一个 Category List，每得到一个 Category 对象都要执行另外一次查询，以获得相应的 Product 对象 List（N＋1，这个例子是 10＋1＝11）。

1:M 和 M:N 的解决方案：*目前这个问题还没有解决方案。它将在近期的版本提供。*

组合键值或多个复杂参数属性

您可能注意到在上面的例子中，resultMap 的 column 属性只指定了一个键值。这意味着只有一个字段和相关的 mapped statement 关联。但是，还有一种语法能把多个字段传给相关的 mapped statement。当对象之间的关系存在组合键值关系，或只是简单地使用一个其他名称而不是#value#的属性时，它就能派上用场了。这种关于 column 属性的语法很简单，写成{param1=column1，param2=column2，…，paramN=columnN}。考虑下面的例子，其中 PAYMENT 表的主键是 CustomerID 和 OrderID：

```
<resultMap id="get-order-result" class="com.ibatis.example.Order">
    <result property="id" column="ORD_ID"/>
    <result property="customerId" column="ORD_CST_ID"/>
    …
    <result property="payments" column="{itemId=ORD_ID, custId=ORD_CST_ID}"
        select=" getOrderPayments"/>
</resultMap>
<statement id="getOrderPayments" resultMap="get-payment-result">
    select * from PAYMENT
    where PAY_ORD_ID = #itemId#
    and PAY_CST_ID = #custId#
</statement>
```

只要字段的次序和 statement 参数的次序相同，您也可以只指定字段名称。例如：

```
{ORD_ID, ORD_CST_ID}
```

但通常这样做会对性能有轻微的影响，并影响到可读性和可维护性。

重要提示！目前 SQL Map 架构无法自动解决 resultMap 之间的双向关系。这在处理“父/子”双向关系的 resultMap 时尤其要注意。一个简单的办法是，为其中一种情况再定义一个不装入父对象的 resultMap（反之亦然）。

注意！某些 JDBC Driver（如嵌入式 PointBase）不支持同时打开多个 ResultSet。这些 Driver 支持复杂类型映射，因为 SQL Map 引擎要求同时打开多个 ResultSet。

注意！Result Map 的名称是局部的，只在定义它的 SQL Map XML 文件中有效。如果要在别的 QL Map XML 文件引用它，需要在 Result Map 名称前加上 SQL Map 的名称（在<sqlMap>根元素中定义的名称）作为前缀。

如果使用微软的SQL Server 2000 JDBC驱动程序，为了在手工事务模式中执行多条语句，需要在数据库连接url加上SelectMethod=Cursor。参见微软知识库Article 313181：
http://support.microsoft.com/default.aspx?scid=kb%3Ben-us%3B313181

支持 Parameter Map 和 Result Map 的数据类型

对于 Parameter Map 和 Result Map，iBATIS 框架支持的 Java 类型如下表：

Java Type	JavaBean/Map Property Mapping	Result Class/ Parameter Class***	Type Alias**
boolean	YES	NO	boolean
Java.lang.Boolean	YES	YES	boolean
byte	YES	NO	byte
Java.lang.Byte	YES	YES	byte
short	YES	NO	short
Java.lang.Short	YES	YES	short
int	YES	NO	int/integer
Java.lang.Integer	YES	YES	int/integer
long	YES	NO	long
Java.lang.Long	YES	YES	long
float	YES	NO	float
Java.lang.Float	YES	YES	float
double	YES	NO	double
Java.lang.Double	YES	YES	double
Java.lang.String	YES	YES	string
Java.util.Date	YES	YES	date
Java.math.BigDecimal	YES	YES	decimal
*Java.sql.Date	YES	YES	N/A
*Java.sql.Time	YES	YES	N/A
*Java.sql.Timestamp	YES	YES	N/A

* 建议不要使用 java.sql.Date，最好使用 java.util.Date 代替。

** 当指定 parameterClass 和 resultClass 时，可以使用类型别名。

*** 基本数据类型如 int，boolean 和 float 不能直接使用。因为 iBATIS 是面向对象的解决方案。因此所有的参数和结果必须是对象。

缓存 Mapped Statement 结果集

通过在查询 statement 中指定 cacheModel 属性，可以缓存 Mapped Statement 中得到的查询结果。Cache model 是在 SQL Map XML 文件中定义的可配置缓存模式，可以使用 cacheModel 元素来配置。例子如下：

```
<cacheModel id="product-cache" type ="LRU" readOnly=”true” serialize=”false”>
    <flushInterval hours="24"/>
    <flushOnExecute statement="insertProduct"/>
```

```
    <flushOnExecute statement="updateProduct"/>
    <flushOnExecute statement="deleteProduct"/>
    <property name="cache-size" value="1000" />
</cacheModel>
```

上面的 cache model 创建了一个名为‘ product-cache ’的缓存，使用‘ 近期最少使用 ”(LRU) 实现。Implemention 属性的名称要么是全限定的类名，要么是缓存实现的别名（参见下面的内容）。根据 cacheModel 中 flush 元素的内容，上面的例子每 24 小时刷新一次。一个 cacheModel 只能有一个 flushInteval 元素，它可以使用 hours，minutes，seconds 或 milliseconds 来设定。另外，当 insertProduct，updateProduct 或 deleteProduct 的 Mapped Statement 执行时，缓存也被刷新。cacheModel 可以定义任意多的 flushOnExecute 元素。某些 cache model 的实现可能需要另外的属性，如上面的 “ cache-size ” 属性。在 LRU cache model 中，cache-size 指定了缓存储存的项数。一旦配置了 cache model，您可以指定 mapped statement 使用的 cache model，例如：

```
<statement id="getProductList" cacheModel="product-cache">
    select * from PRODUCT where PRD_CAT_ID = #value#
</statement>
```

只读 VS 可读写

框架同时支持只读和可读写缓存。只读缓存可供所有用户共享，因此性能更好。但是，只读缓存的数据不应该被修改。相反，要更新数据，必须从数据库（或读写缓存）中读出数据。因此，如果要读出数据并修改，则需要可读写缓存。要使用只读缓存，在 cacheModel 设置 readOnly=“ true ”。要使用可读写缓存，则设置 readOnly=“ false ”。缺省为只读缓存 (true)。

Serializable可读写缓存

正如您所知道的，只对当前 Session 有效的缓存对整体应用性能的提高作用有限。Serializable 可读写缓存可以提高整体应用（而不仅仅是每个 Session）的性能。这种缓存为每一个 Session 返回缓存对象不同的实例（复本）。因此每一个 Session 都可以安全修改返回的对象。不同之处在于，通常您希望从缓存中得到同一个对象，但这种情况下得到的是不同的对象。还有，每一个缓冲在 Serializable 缓存的对象都必须是 Serializable 的。这意味着不能同时使用 Serializable 缓存和延迟加载，因为延迟加载代理不是 Serializable 的。想知道如何把 Serializable 缓存，延迟加载和联合查询结合起来使用，最好的方法是尝试。要使用 Serializable 缓存，设置 readOnly= “ false ” 和 serialize= “ true ”。缺省情况下，缓存是只读模式，不使用 Serializable 缓存。只读缓存不需要 Serializable。

缓存类型

Cache Model 使用插件方式来支持不同的缓存算法。它的实现在 cacheModel 的用 type 属性来指定（如上所示）。指定的实现类必须实现 CacheController 接口，或是下面 4 个别名中的其中之一。Cache Model 实现的其他配置参数通过 cacheModel 的 property 元素来设置。目前包括以下的 4 个实现：

- **“MEMORY” (com.ibatis.db.sqlmap.cache.memory.MemoryCacheController)**

  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 实现可以这样配置：

```
<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。

下面的表格介绍了在 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。

WEAK（缺省）	大多数情况下，WEAK 类型是最佳选择。如果不指定类型，缺省类型就是 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。
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。
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）。

- **“LRU” (com.ibatis.db.sqlmap.cache.lru.LruCacheController)**

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。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（例如，在 PaginatedList 和常用的查询关键字结果集中翻页），LRU Cache 是一个不错的选择。

LRU Cache 实现可以这样配置：

```
<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。因此，不要 Cache 太多的对象，以免内存不足。

- **“FIFO” (com.ibatis.db.sqlmap.cache.fifo.FifoCacheController)**

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。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。

FIFO Cache 可以这样配置：

```
<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 Cache 太多的对象，以免内存不足。

- **“OSCACHE” (com.ibatis.db.sqlmap.cache.oscache.OSCacheController)**

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，高度的灵活性。

OSCACHE 实现可以这样配置：

```
<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，您可以配置 Cache 的算法（和上面讨论的算法很类似），Cache 的大小，持久化方法（内存，文件等）和集群方法。

要获得更详细的信息，请参考 OSCache 文档。OSCache 及其文档可以从 OpenSymphony 网站上获取：

http://www.opensymphony.com/oscache/

动态 Mapped Statement

直接使用 JDBC 一个非常普遍的问题是动态 SQL。使用参数值、参数本身和数据列都是动态的 SQL，通常非常困难。典型的解决方法是，使用一系列 if-else 条件语句和一连串讨厌的字符串连接。对于这个问题，SQL Map API 使用和 mapped statement 非常相似的结构，提供了较为优雅的方法。这里是一个简单的例子：

```
<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>
```

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 Mapped Statement 的用处更明显。如下面比较复杂的例子：

```
<statement id="dynamicGetAccountList" resultMap="account-result" >
    select * from ACCOUNT
    <dynamic prepend="WHERE">
        <isNotNull prepend="AND" property="firstName">
            (ACC_FIRST_NAME = #firstName#
            <isNotNull prepend="OR" property="lastName">
                ACC_LAST_NAME = #lastName#
            </isNotNull>
            )
        </isNotNull>
        <isNotNull prepend="AND" property="emailAddress">
            ACC_EMAIL like #emailAddress#
        </isNotNull>
        <isGreaterThan prepend="AND" property="id" compareValue="0">
            ACC_ID = #id#
        </isGreaterThan>
    </dynamic>
    order by ACC_LAST_NAME
```

```
</statement>
```

根据不同的条件，上面动态的语句可以产生 16 条不同的查询语句。使用 if-else 结构和字符串，会产生上百行很乱的代码。

而使用动态 Statement，和在 SQL 的动态部位周围插入条件标签一样容易。例如：

```
<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>
```

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。以下小节讨论不同的条件元素，包括二元条件元素，一元条件元素和其他动态元素。

二元条件元素

二元条件元素将一个属性值和一个静态值或另一个属性值比较，如果条件为“ 真 ”，元素体的内容将被包括在查询 SQL 语句中。

- **二元条件元素的属性：**

prepend － 可被覆盖的 SQL 语句组成部分，添加在语句的前面（可选）

property － 被比较的属性（必选）

compareProperty － 另一个用于和前者比较的属性（必选或选择 compareValue）

compareValue － 用于比较的值（必选或选择 compareProperty）

<isEqual>	比较属性值和静态值或另一个属性值是否相等。
<isNotEqual>	比较属性值和静态值或另一个属性值是否不相等。

<isGreaterThan>	比较属性值是否大于静态值或另一个属性值。
<isGreaterEqual>	比较属性值是否大于等于静态值或另一个属性值。
<isLessThan>	比较属性值是否小于静态值或另一个属性值。
<isLessEqual>	比较属性值是否小于等于静态值或另一个属性值。 例子： **<isLessEqual prepend="AND" property="age" compareValue="18">** **ADOLESCENT = 'TRUE'** **</isLessEqual>**

一元条件元素

一元条件元素检查属性的状态是否符合特定的条件。

- **一元条件元素的属性：**

prepend － 可被覆盖的 SQL 语句组成部分，添加在语句的前面（可选）

property － 被比较的属性（必选）

<isPropertyAvailable>	检查是否存在该属性（存在 parameter bean 的属性）。
<isNotPropertyAvailable>	检查是否不存在该属性（不存在 parameter bean 的属性）。
<isNull>	检查属性是否为 null。
<isNotNull>	检查属性是否不为 null。
<isEmpty>	检查 Collection.size()的值，属性的 String 或 String.valueOf()值，是否为 null 或空（" " 或 size() < 1）。
<isNotEmpty>	检查 Collection.size()的值，属性的 String 或 String.valueOf()值，是否不为 null 或不为空（" " 或 size() > 0）。 例子： **<isNotEmpty prepend="AND" property="firstName" >** **FIRST_NAME=#firstName#** **</isNotEmpty>**

其他元素

- **Parameter Present：**这些元素检查参数对象是否存在。

Parameter Present 的属性：

prepend － 可被覆盖的 SQL 语句组成部分，添加在语句的前面（可选）

<isParameterPresent>	检查是否存在参数对象（不为 null）。
<isNotParameterPresent>	检查是否不存在参数对象（参数对象为 null）。 例子： **<isNotParameterPresent prepend="AND">** **EMPLOYEE_TYPE = 'DEFAULT'** **</isNotParameterPresent>**

- **Iterate**：这属性遍历整个集合，并为 List 集合中的元素重复元素体的内容。

Iterate 的属性：

prepend － 可被覆盖的 SQL 语句组成部分，添加在语句的前面（可选）

property － 类型为 java.util.List 的用于遍历的元素（必选）

open － 整个遍历内容体开始的字符串，用于定义括号（可选）

close －整个遍历内容体结束的字符串，用于定义括号（可选）

conjunction － 每次遍历内容之间的字符串，用于定义 AND 或 OR（可选）

<iterate>	遍历类型为 java.util.List 的元素。 例子： **<iterate prepend="AND" property="userNameList"** **open="(" close=")" conjunction="OR">** **username=#userNameList[]#** **</iterate>** **注意：**使用<iterate>时，在List元素名后面包括方括号[]非常重要，方括号[]将对象标记为List，以防解析器简单地将List输出成String。

简单的动态SQL元素

虽然动态 Mapped Statement API 功能强大，但有时仅需要一小部分的动态 SQL 即可。为此，SQL statement 和 statement 都可以包含简单的动态 SQL 元素，以帮助实现动态的 order by 子句，动态的查询字段或 SQL 语句的其他动态部分。简单动态 SQL 元素的概念有点象 inline parameter 的映射，但使用了稍微不同的语法。考虑下面的例子：

```
<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）。不同的是，它从根本上改变了 SQL 语句本身，比仅仅简单地改变参数值严重得多。在这样的动态 SQL 语句中，错误可能会引起安全，性能和稳定性的风险。因此，应细心检查，以确保简单动态 SQL 元素使用的正确。另外，还要留意您的设计，以防数据库细节对业务逻辑对象模型造成不好的影响。例如，您不应因为要使用 order by 子句，而将一个数据字段名作为业务对象的属性，或作为 JSP 页面的一个域的值。

简单动态元素可以包含在<dynamic-mapped-statement>中，当要修改 SQL 语句本身时它

能派上用场。例如：

```
<statement id="getProduct" resultMap="get-product-result">
    SELECT * FROM PRODUCT
    <dynamic prepend="WHERE">
        <isNotEmpty property="description">
            PRD_DESCRIPTION $operator$ #description#
        </isNotEmpty>
    </dynamic>
</statement>
```

上面的例子中，参数对象的 operator 属性将用于替代符号$operator$。因此，假如 operator 属性等于“ like ”，description 属性等于“ %dog% ”，生成的 SQL 语句如下：

```
SELECT * FROM PRODUCT WHERE PRD_DESCRIPTION LIKE '%dog%'
```

使用 SQL Map API 编程

SQL Map API 力求简洁。它为程序员提供 4 种功能 配置一个 SQL Map，执行 SQL update 操作，执行查询语句以取得一个对象，以及执行查询语句以取得一个对象的 List。

配置 SQL Map

一旦您创建了 SQL Map XML 定义文件和 SQL Map 配置文件，配置 SQL Map 就是一件极其简单的事情。SQL Map 使用 XmlSqlMapClientBuilder 来创建。这个类有一个静态方法叫 buildSqlMap。方法 buildSqlMap 简单地用一个 Reader 对象为参数，读入 sqlMap-config.xml 文件（不必是这个文件名）的内容。

```
String resource = “com/ibatis/example/sqlMap-config.xml”;
Reader reader = Resources.getResourceAsReader (resource);
SqlMapClient sqlMap = SqlMapClientBuilder.buildSqlMap(reader);
```

事务处理

缺省情况下，调用 SqlMapClient 对象的任意 executeXxxx()方法将缺省地自动 COMMIT/ROLLBACK。这意味着每次调用 executeXxxx()方法都是一个独立的事务。这确实很简单，但对于需要在同一个事务中执行多个语句的情况（即只能同时成功或失败），并不适用。这正是事务处理要关心的事情。

如果您在使用 Global Transaction（在 SQL Map 配置文件中设置），您可以使用自动提交并且可以得到在同一事务中执行的效果。但为了提高性能，最好是明确地划分事务的范围，因为这样做可以减少连接池的通讯流量和数据库连接的初始化。

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连接，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 ();
    }
}
```

注意！事务不能嵌套。在调用 commit（）或 rollback（）之前，从同一线程多次调用.startTransaction，将引起抛出例外。换句话说，对于每个 SqlMap 实例，每个线程最多只能打开一个事务。

注意！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（或关闭连接）唯一的方法是调用 commitTransaction（）或 rollbackTransaction（）方法。否则，会用光连接池中的连接并导致死锁。

自动的事务处理

虽然极力推荐使用明确划分的事务范围，在简单需求（通常使只读）的情况下，可以使用简化的语法。如果您没有使用 startTransaction（），commitTransation（）和 rollbackTransaction（）方法来明确地划分事务范围，事务将会自动执行。例如：

```
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 {
      Item item = (Item) sqlMap.queryForObject ("getItem", itemId);
      item.setDescription ("TX1");
      //No transaction demarcated, so transaction will be automatic (implied)
      sqlMap.update ("updateItem", item);
      item.setDescription (newDescription);
      item.setDescription ("TX2");
      //No transaction demarcated, so transaction will be automatic (implied)
      sqlMap.update("updateItem", item);
```

```
    } catch (SQLException e) {
        throw (SQLException) e.fillInStackTrace();
    }
}
```

注意！使用自动事务处理要非常小心。虽然看起来很有吸引力，但如果有多个数据更新操作要在同一事务中处理时，您会遇到麻烦。在上面的例子中，如果第二个“ updateItem ”操作失败，第一个“ updateItem ”操作仍然会执行，description 会更新成“ TX1 ”。

全局（分布式）事务

SQL Map 框架支持全局事务。全局事务也叫分布式事务，它可以允许您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。

- **External/Programmatic Global 事务**

您可以选择外部管理或手工编程管理全局事务，或实现一个象 EJB 一样的架构。使用 EJB，您可以通过使用 EJB 的描述文件定义事务范围。更多的细节超出了本文的范围。要支持外部管理或手工编程管理全局事务，必须在 SQL Map 配置文件中(参见前面章节的内容)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的多余，因为事务的开始、提交和回退都由外部的事务管理器来控制。但是，使用 SqlMapClient 的 startTransaction（），commitTransaction（）或 rollbackTransaction（）来划分事务范围（相对于自动的事务处理），还是对提高性能有帮助。继续使用这些方法，可以保持编程规范的一致性。另一个好处是，在某些情况下，您可能需要改变关闭资源的顺序（不幸的是，不同的应用服务器和事务管理器具有不同的规则）。除了这些考虑，要使用全局事务，不需要改变您的 SQL Map 代码。

- **受管理的（Managed）全局事务**

SQL Map 框架也可以为您管理全局事务。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。更详细的配置信息，参见前面关于<transactionManager>的说明。

使用全局事务编程，代码没有多大的不同，但有几个小小的地方要注意。例如：

```
try {
    orderSqlMap.startTransaction();
    storeSqlMap.startTransaction();
    orderSqlMap.insertOrder(…);
    orderSqlMap.updateQuantity(…);
```

```
        storeSqlMap.commitTransaction();
        orderSqlMap.commitTransaction();
} finally {
        try {
                storeSqlMap.endTransaction()
        } finally {
                orderSqlMap.endTransaction()
        }
}
```

上面的例子中，假设我们通过两个 SqlMapClient 来使用两个不同的数据库。第一个开始事务的 SqlMapClient（orderSqlMap）同时也开始了一个全局事务。在这之后，所有其他的操作将被看作是这个全局事务的一部分，直到同一个 SqlMapClient（orderSqlMap）调用 commitTransaction（）或 rollbackTransaction（），届时全局事务被提交并完成其他所有的操作。

警告！虽然这些看起来很简单，但记住不要滥用全局（分布式）事务，这点很重要。这样做既有性能方面的考虑，同时也是因为全局事务会让应用服务器和数据库驱动程序的设置变得更复杂。虽然看起来简单，您可能还是会遇到一些困难。记住，EJB 拥有更多厂商和工具支持。对于需要分布式事务的应用，最好还是使用 Session EJB。在 www.ibatis.com 的 JPetstore，就是一个使用 SQL Map 全局事务的例子。

批处理

如果要执行很多非查询（insert/update/delete）的语句，您可能喜欢将它们作为一个批处理来执行，以减少网络通讯流量，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。

用 SqlMapClient 执行 SQL 语句

SqlMapCient 类提供了执行所有 mapped statement 的 API。这些方法如下：

```
public int insert(String statementName,
        Object parameterObject)
        throws SQLException
```

```
public int update(String statementName,
        Object parameterObject)
        throws SQLException

public int delete(String statementName,
        Object parameterObject)
        throws SQLException

public Object queryForObject(String statementName,
        Object parameterObject)
        throws SQLException
public Object queryForObject(String statementName,
        Object parameterObject, Object resultObject)
        throws SQLException

public List queryForList(String statementName,
        Object parameterObject)
        throws SQLException

public List queryForList(String statementName, Object parameterObject,
        int skipResults, int maxResults)
        throws SQLException

public List queryForList (String statementName,
        Object parameterObject, RowHandler rowHandler)
        throws SQLException

public PaginatedList queryForPaginatedList(String statementName,
        Object parameterObject, int pageSize)
        throws SQLException

public Map queryForMap (String statementName, Object parameterObject,
        String keyProperty)
        throws SQLException

public Map queryForMap (String statementName, Object parameterObject,
        String keyProperty, String valueProperty)
        throws SQLException
```

在上面的每个方法中，Mapped Statement 的名称作为第一个参数。这个名称要对应<statement>的名称属性。另外，第二个参数总是参数对象。如果不需要参数对象，可以为null。除了这些之外，上面的方法就没有相同的了。下面简单地介绍一下它们的不同之处。

- **insert( )，update( )，delete( )**

这些方法用于数据更新（即非查询语句）。这就是说，使用下面的查询方法来执行数据更新操作并不是不可能。但这显得很奇怪，并明显依赖于 JDBC 的驱动程序。执行 update()，返回受影响数据记录的数目。

- **queryForObject( )**

queryForObject()方法有两个版本。一个返回查询得到的新对象，另一个用一个事先生成的对象作为参数。后者对于使用多个查询为对象赋值很有用。

- **queryForList( )**

queryForList()方法也有三个版本。第一个执行一个查询并返回所有的查询结果。第二个允许指定跳过结果的数目（即开始点）和返回结果的最大数目。对于查询一个数量很大的结果集，并且不想返回所有的数据时很有用。

最后一个版本的 queryForList()方法需要一个行处理器（row handler）作为输入参数。这个方法可以让您使用结果对象，而不是数据表的行和列来逐行地处理查询结果集。这个方法的参数除了例行的名称和参数对象外，还包括了一个实现了 RowHandler 接口的类的实例。RowHandler 接口只有一个方法：

```
public void handleRow (Object object, List list);
```

对于每一行从数据库中返回的数据，RowHandler 接口的方法都会被执行。这种处理数据的方法，简洁而具有扩展性。使用 RowHandler 的例子，请参见下面的章节。输入参数 list 是 List 接口的一个实例，并从 queryForList()方法中再次返回。您可以向 list 对象添加零或多个结果对象。显然，如果您处理上百万的数据行，把它们全部放入一个 list 对象里并不是个好方法。

- **queryForPaginatedList( )**

对于要返回一个可以向前和向后翻页的数据子集，queryForPaginatedList()方法很有用，它返回一个可管理的 List 对象。通常用于只显示一部分查询结果的用户界面。一个常见的例子是，搜索引擎找到了 10000 条结果，但每次只显示其中的 100 条。PaginatedList 接口包含了向前和向后翻页的方法（nextPage()，previousPage()，gotoPage()），并提供了检查翻页状态的方法（isFirstPage()，isMiddlePage()，isLastPage()，isPreviousPageAvailable()，getPageIndex()，getPageSize()）。虽然不能从 PaginatedList 接口得到查询结果集的总数，但这个总数可以再执行一个简单的语句 count()来得到。否则，PaginatedList 接口会大大的降低性能。

- **queryForMap( )**

上面的方法将查询结果集放在 List 对象中，而 queryForMap()方法将结果集放在一个 Map 对象中，这个 Map 对象用一个传入参数 keyProperty 作为 key 值。例如，要读入一批 Employee 对象，您可以将这些 Employee 对象放在一个用 employeeNumber 属性作为 key 值的 Map 对象中。Map 对象的值可以是整个 Employee 对象，也可以是 Employee 对象的另一个属性，属性的名称由第二个参数 valueProperty 指定。例如，您可能只是需要一个 Map 对象，用员工号作为 key 值，员工姓名作为 value 值。**不要**把它和用 Map 作为结果对象的概念混淆。这个方法可以使用 Java Bean 和 Map（或基本类型的包装类，但不可能这样用）作为结果对象。

代码例子

例子1：执行update（insert，update，delete）

```
sqlMap.startTransaction();
Product product = new Product();
product.setId (1);
product.setDescription (“Shih Tzu”);
int rows = sqlMap.insert (“insertProduct”, product);
sqlMap.commitTransaction();
```

例子2：查询成对象（select）

```
sqlMap.startTransaction();
Integer key = new Integer (1);
Product product = (Product)sqlMap.queryForObject (“getProduct”, key);
sqlMap.commitTransaction();
```

例子3：用预赋值的结果对象查询成对象（select）

```
sqlMap.startTransaction();
Customer customer = new Customer();
sqlMap.queryForObject(“getCust”, parameterObject, customer);
sqlMap.queryForObject(“getAddr”, parameterObject, customer);
sqlMap.commitTransaction();
```

例子4：查询成对象List（select）

```
sqlMap.startTransaction();
List list = sqlMap.queryForList (“getProductList”, null);
sqlMap.commitTransaction();
```

例子5：自动提交

```
//当没调用 startTransaction 的情况下，statements 会自动提交。

//没必要 commit/rollback。

int rows = sqlMap.insert (“insertProduct”, product);
```

例子6：用结果集边界查询成对象List（select）

```
sqlMap.startTransaction();
List list = sqlMap.queryForList (“getProductList”, null, 0, 40);
sqlMap.commitTransaction();
```

例子7：用RowHandler执行查询（select）

```
public class MyRowHandler implements RowHandler {
  public void handleRow (Object object, List list) throws SQLException {
      Product product = (Product) object;
      product.setQuantity (10000);
      sqlMap.update (“updateProduct”, product);
      // Optionally you could add the result object to the list.
      // The list is returned from the queryForList() method.
  }
}
sqlMap.startTransaction();
RowHandler rowHandler = new MyRowHandler();
List list = sqlMap.queryForList (“getProductList”, null, rowHandler);
sqlMap.commitTransaction();
```

例子8：查询成Paginated List（select）

```
PaginatedList list =
sqlMap.queryForPaginatedList (“getProductList”, null, 10);
list.nextPage();
list.previousPage();
```

例子9：查询成Map（select）

```
sqlMap.startTransaction();
Map map = sqlMap.queryForMap (“getProductList”, null, “productCode”);
sqlMap.commitTransaction();
Product p = (Product) map.get(“EST-93”);
```

用 Jakarta Commons Logging 记录 SQL Map 日志

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。您可以“ plug-in ”包括 Log4J 和 JDK1.4 Loging API 等不同的日志服务实现。Jakarta Commons Logging，Log4J 和 JDK1.4 Logging API 的详细规范超出了本文档的范围。但是下面的例子可以提供一个开始使用它们的起点。如果您想知道得更多，可以参考下面的 URL：

Jakarta Commons Logging

- http://jakarta.apache.org/commons/logging/index.html

Log4J

- http://jakarta.apache.org/log4j/docs/index.html

JDK 1.4 Logging API

- http://java.sun.com/j2se/1.4.1/docs/guide/util/logging/

配置日志服务

配置 Commons Logging 日志服务非常简单，只需加入几个配置文件（例如 log4j.properties）和 JAR 文件（例如 log4j.jar）即可。下面的配置例子使用 Log4J 作为日志服务的实现。分为两个步骤：

第一步：添加 Log4J JAR 文件

因为要使用 Log4J，我们要确保应用能使用它的 JAR 文件。记住，Commons Logging 是一个抽象的 API，并不提高日志的实现。因此，要使用 Log4J，必须将它的 JAR 文件加入应用的类路径中。您可以从上面的 URL 下载 Log4J。对于 Web 应用，可以将 log4.jar 添加到 WEB-INF/lib 目录下，或对于独立的应用，可以在 JVM 的-classpath 启动参数中添加。

第二步：配置 Log4J

配置 Log4J 也很简单。象 Commons Logging 一样，将配置文件添加到类路径的根目录中（即，不要放在 package 里）。这个配置文件是 log4j.properties，例子如下：

log4j.properties

```
1 # Global logging configuration
```

```
2 log4j.rootLogger=ERROR, stdout
3
4 # SqlMap logging configuration...
5 #log4j.logger.com.ibatis=DEBUG
6 #log4j.logger.com.ibatis.common.jdbc.SimpleDataSource=DEBUG
7 #log4j.logger.com.ibatis.common.jdbc.ScriptRunner=DEBUG
8 #log4j.logger.com.ibatis.sqlmap.engine.impl.SqlMapClientDelegate=DEBUG
9 #log4j.logger.java.sql.Connection=DEBUG
10 #log4j.logger.java.sql.Statement=DEBUG
11 #log4j.logger.java.sql.PreparedStatement=DEBUG
12 #log4j.logger.java.sql.ResultSet=DEBUG
13
14 # Console output...
15 log4j.appender.stdout=org.apache.log4j.ConsoleAppender
16 log4j.appender.stdout.layout=org.apache.log4j.PatternLayout
17 log4j.appender.stdout.layout.ConversionPattern=%5p [%t] - %m%n
```

上面的配置文件仅在日志中记录错误（error）信息。文件的第 2 行指定在标准输出的 Appender 中只记录 error 信息。Appender 是用于收集输出（例如，控制台，文件和数据库等）的组件。我们可以把第 2 行改为：

```
log4j.rootLogger=DEBUG, stdout, fileout
```

这样，Log4J 现在将把所有的日志事件记录到“ stdout ”Appender（控制台）和“ fileout ”滚动日志文件 Appender 中。如果要更好地控制日志输出的级别，可以象文件中“ SqlMap logging configuration ”一样配置每个类的日志输出级别（配置文件中被注释的第 5 行到第 9 行）。因此，如果您要记录 PreparedStatement 类（SQL 语句）DEBUG 级别的日志到控制台，可以将第 9 行改为（注意，已删除注释）：

```
log4j.logger.java.sql.PreparedStatement=DEBUG, stdout
```

配置文件 log4j.properties 其余部分用来设置 Appender，关于 Appender 的设置已超出本文的范围，不再详述。您可以在 Log4J 网站找到更多的信息。或者，您可以通过作试验来熟悉它。

Java Bean 简易教程

SqlMap 架构需要对 Java Bean 有坚实的理解。幸运的是，和 SqlMap 有关系的 Java Bean API 的并不是很多。因此，如果您以前没有接触过 Java Bean，下面是一个简单的介绍。

Java Bean 是什么呢？Java Bean 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。Java Bean 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。Java Bean 还应实现 Serializable 接口。Java Bean 还支持其他特性（如事件等）。但这些特性 SQL Map 和 Web 应用中并不重要。

下面是 Java Bean 的一个例子：

```
public class Product implements Serializable {
    private String id;
    private String description;
    public String getId() {
        return id;
    }
    public void setId(String id) {
        this.id = id;
    }
    public String getDescription() {
        return description;
    }
    public void setDescription(String description) {
        this.description = description;
    }
}
```

注意！对于特定的属性，不要混淆“ Get ”方法和“ Set ”方法的数据类型。例如，对于数值类型的“ account ”属性，确保 getter 方法和 setter 方法使用同一数值类型，如下面的例子：

```
public void setAccount (int acct) {….}
public int getAccount () {….}
```

注意到两个方法都使用了“ int ”类型。如果从“ get ”方法返回“ long ”类型，会出现问题。

注意！类似的，您必须只有一个“ get ”方法和“ set ”方法。Be judicious with polymorphic methods. You’re better off naming them more specifically anyway.

注意！对于“ boolean ”类型的属性，“ get ”方法还可以有别的语法，可以是“ isXxxx() ”格式。确保您只有一个“ is ”方法或“ get ”方法，不能同时有两种写法。

恭喜，您完成了这个 Java Bean 教程。

Side Bar: Object Graph Navigation (JavaBeans Properties, Maps, Lists)

如果您熟悉 Struts 或其他符合 Java Bean 规范的框架，那么您可能注意到在文档使用一种特别的语法来访问对象。SQL Map 允许通过 JavaBean 的属性，Map（key/value）和 List 来遍历 Object Graph：

```
Employee emp = getSomeEmployeeFromSomewhere();
((Address) ( (Map)emp.getDepartmentList().get(3) ).get (“address”)).getCity();
```

可以使用下面的方法，在一个 SqlMapClient 属性（ResultMap，ParameterMap 等）中遍历 employee 对象的属性（假设 employee 对象的结构如上所示）：

```
“departmentList[3].address.city”
```

Resources（com.ibatis.common.resource.*）

Resources 类为从类路径中加载资源，提供了易于使用的方法。处理 ClassLoader 是一项富于挑战的工作，尤其是应用服务器/容器的情况下。Resources 类试图简化这些工作。

Resources 类常用于以下几种情况：

- 从类路径加载 SQL Map 配置文件（如 sqlMap-config.xml）。
- 从类路径加载 DAO Manager 配置文件（如 dao.xml）。
- 从类路径加载各种.properties 文件。

加载一个资源有很多方式，包括：

- 对于简单的只读文本数据，加载为 Reader。
- 对于简单的只读二进制或文本数据，加载为 Stream。
- 对于可读写的二进制或文本文件，加载为 File。
- 对于只读的配置属性文件，加载为 Properties。
- 对于只读的通用资源，加载为 URL。

按以上的顺序，Resources 类加载资源的方法如下：

```
Reader getResourceAsReader(String resource);
Stream getResourceAsStream(String resource);
File getResourceAsFile(String resource);
Properties getResourceAsProperties(String resource);
Url getResourceAsUrl(String resource);
```

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，将使用系统的 ClassLoader。在某些环境下（比如某些应用服务器），ClassLoader 可能是个麻烦事，您可以指定所使用的 ClassLoader（比如使用加载应用的 ClassLoader）。上面每个方法都有相应把 ClassLoader 作为参数的方法。它们是：

```
Reader getResourceAsReader (ClassLoader classLoader, String resource);
Stream getResourceAsStream (ClassLoader classLoader, String resource);
File getResourceAsFile (ClassLoader classLoader, String resource);
Properties getResourceAsProperties (ClassLoader classLoader, String resource);
Url getResourceAsUrl (ClassLoader classLoader, String resource);
```

以上方法的 resource 参数名称应该是全限定名，加上全文件/资源名。例如，如果在类路径中有资源“ com.domain.mypackage.MyPropertiesFile.properties ”，您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，您可以从类路径加载 SQL Map 配置文件为一个 Reader。假设它在类路径的 properties 目录下（properties.sqlMap-config.xml）。

```
String resource = “properties/sqlMap-config.xml”;
Reader reader = Resources.getResourceAsReader(resource);
SqlMapClient sqlMap = XmlSqlMapBuilder.buildSqlMap(reader);
```

SimpleDataSource （com.ibatis.common.jdbc.*）

SimpleDataSource 类是 JDBC2.0 的 DataSource 接口的一个简单实现。它的使用方法和其他的 JDBC DataSource 实现完全相同。请参考 JDBC 标准扩展 API（http://java.sun.com/products/jdbc/jdbc20.stdext.javadoc/）。

注意！SimpleDataSource 是一个方便而高效的 DataSource 实现。但是，对于大型的企业级应用或关键应用，建议使用企业级的 DataSource 实现（例如应用服务器或商业 O/R 映射工具自带的 DataSource 实现）。

SimpleDataSource 的构造器使用一个 Properties 作为参数，其中包含了一系列的配置参数。下表说明这些配置参数。必需的参数仅包括以“ JDBC. ”开头的参数。

属性名称	是否必须	缺省值	说明
JDBC.Driver	是	N/A	JDBC 驱动程序类名
JDBC.ConnectionURL	是	N/A	数据库URL
JDBC.Username	是	N/A	数据库用户名
JDBC.Password	是	N/A	数据库用户密码
JDBC.DefaultAutoCommit	否	取决于驱动程序	连接池中所有连接缺省的自动提交参数
Pool.MaximumActiveConnections	否	10	数据库连接池可维持的最大容量
Pool.MaximumIdleConnections	否	5	数据库连接池中允许的挂起（idle）连接数
Pool.MaximumCheckoutTime	否	20000	数据库联接池中，连接被某个任务所允许占用的最大时间，如果超过这个时间限定，连接将被强制收回（单位：毫秒）。
Pool.TimeToWait	否	20000	当线程试图从连接池中获取连接时，连接池中无可用连接可供使用，此时线程将进入等待状态，直到池中出现空闲连接。此参数设定了线程所允许等待的最长时间（单位：毫秒）。
Pool.PingQuery	否	N/A	数据库连接状态检测语句。某些数据库在连接在某段时间持续处于空闲状

			态时会将其断开。而连接池管理器将通过此语句检测池中连接是否可用。它对性能的影响较大，应小心使用。检测语句应该是一个最简化的无逻辑 SQL，如：select 1 from dual
Pool.PingEnabled	否	false	是否允许检测连接状态
Pool.PingConnectionsOlderThan	否	0	对持续连接时间超过设定值（毫秒）的连接进行检测
Pool.PingConnectionsNotUsedFor	否	0	对空闲超过设定值（毫秒）的连接进行检测
*Driver.**	否	N/A	某些 JDBC 驱动程序支持别的配置参数。为了给驱动程序发送这些参数，可以在参数名前面加上“ Driver. ”前缀。 例如，如果驱动程序支持配置参数“ compressionEnabled ”，您可以在 SimpleDataSource 把它设置为“ Driver.compressionEnabled=true ”

使用 SimpleDataSource 类的例子：

```
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
```

ScriptRunner (com.ibatis.common.jdbc.*)

ScriptRunner 类用于执行 SQL 语句，例如创建数据库 schema，或插入缺省或测试数据等。从下面的例子可以认识到它的易用性：

例子脚本：initialize-db.sql

```
-- Creating Tables – Double hyphens are comment lines
CREATE TABLE SIGNON (USERNAME VARCHAR NOT NULL, PASSWORD VARCHAR
NOT
NULL, UNIQUE(USERNAME));
-- Creating Indexes
CREATE UNIQUE INDEX PK_SIGNON ON SIGNON(USERNAME);
-- Creating Test Data
INSERT INTO SIGNON VALUES('username','password');
```

例子 1：使用现成的数据库连接

```
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();
```

例子 2：使用新的数据库连接

```
ScriptRunner runner = new ScriptRunner (“com.some.Driver”, “jdbc:url://db”, “login”,
“password”);
runner.runScript(conn, new FileReader("/usr/local/db/scripts/ initialize-db.sql"));
```

例子 3：使用新创建的数据库连接

```
Properties props = getProperties (); // some properties from somewhere
ScriptRunner runner = new ScriptRunner (props);
runner.runScript(conn, new FileReader("/usr/local/db/scripts/ initialize-db.sql"));
```

上面例子的属性文件必须包含以下属性：

```
driver=org.hsqldb.jdbcDriver
url=jdbc:hsqldb:.
username=dba
password=whatever
stopOnError=true
```

还有一些您可能会用到的方法：

```
// if you want the script runner to stop running after a single error
scriptRunner.setStopOnError (true);
// if you want to log output to somewhere other than System.out
scriptRunner.setLogWriter (new PrintWriter(…));
```