**使用 ibatis 处理复杂对象数据关系的实例**

**如何使用 ibatis 处理复杂对象数据关系**

iBatis 是一个开源的对象关系映射程序，其工作是将对象映射到 SQL 语句。和其它 O/R Mapping 框架不同，iBatis 开发者需要自己编写和维护 SQL 语句，这给开发带来了很多的灵活性的同时，也带来了很大的复杂度与工作量。 在一个数据库中，**常见的对象关系**有：1 对 1，1 对多，多对多，单表映射，多表映射，单主键，多主键以及对象主键等种种情况。在使用 ibatis 处理数据映射时，需要根据不同的情况，写出不同的 sql 语句和 sql map 的实现，特别是在复杂关系跟复杂主键时需要编写大量的复杂 sql。本文将详细介绍如何使用 ibatis 来处理数据库中的复杂数据对象关系，根据数据关系的分类，给出具体的实现代码模板与解释。通过本文读者可以了解 ibatis 处理复杂关系时可以使用方式，在具体编写代码过程中只需要套用本文中列出的这些模板，这样可以大大简化开发和加快开发的速度。本文的读者需要对 ibatis 已经有一些基本的使用经验。

6http://dw1.s81c.com/developerworks/i/v17/dw-cmts-arrow.png [评论：](http://www.ibm.com/developerworks/cn/opensource/os-cn-ibatis/#icomments)

[顾 翊](http://www.ibm.com/developerworks/cn/opensource/os-cn-ibatis/#authorN1001E), 高级软件工程师, IBM

2010 年 9 月 02 日

* [+](http://www.ibm.com/developerworks/cn/opensource/os-cn-ibatis/#toggle)**内容**

**ibatis 基本介绍**

起源于 2001 年的开放源代码项目 ibatis，是一个基于 Java 的持久层框架。与 Hibernate, Toplink 等持久化框架不同，ibatis 是一个 “半自动化”的 ORM 实现。ibatis 没有对数据库结构提供了较为完整的封装，而是提供了一个从 POJO 到数据库表的全套映射机制。这使得在开发 ibatis 的时候，需要手动的编写 sql 来提过数据库与类对象之间的映射，这在给开发提供了很大的灵活性的同时，大大增加了开发的工作量。Ibatis 主要有以下的这些特性：

1. 简单性

Ibatis 是一个非常简单易用的工具包。这个简单性不仅仅体现在开发库的轻量小巧上，iBATIS 对于 Java 开发者来说也非常简单。因为它除了不用编写那么多代码外与 JDBC 的工作机制非常相像，iBATIS 就是以 XML 的形式来描述的 JDBC 代码。iBATIS 对于数据库管理员以及 SQL 程序员来 说也非常容易理解。iBATIS 配置文件几乎人人都能读懂，只要他有 SQL 编程的经验。

2. 效率与性能

iBATIS 通过一种简单的方式来配置和使用，其性能与 JDBC 相当。通过直接编写 SQL 查询，开发人员可以直接控制 SQL 语句来优化执行效率，甚至可以在其中调用存储过程，为开发人员提供了巨大的 SQL 优化空间。

3. 代码分离

在 iBATIS 中，SQL 语句在很大程度上同应用的源代码是分离 的，SQL 程序员可以按照 SQL 原本的方式来编写它，而不必担心有关 SQL 字符串连接的问题。iBATIS 提供了充分的自由，使得任何人都可以开发、观察并且修改在数据库中执行的 SQL 语句。

4. 可移植性

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

**常见的对象映射与实现**

由于在使用 ibatis 进行项目开发中，需要根据数据库的表结构与 Java 类之间的关系写出实现的 SQL 代码，不同的关系会对应不同 SQL 实现。这些 SQL 代码将会和其他 mapping 信息一起放在 ibatis 的 SQL mapping 文件中。下面就对这些每一种映射进行详细分析并给出实现示例。

**单表对象绑定与多表对象绑定**

在 ibatis 中最简单的关系就是一个对象对单个表的绑定，这也是最简单的类与数据库的映射关系。在没有任何外键关系的时候，在 ibatis 中直接将类的字段对应到表的列即可：

**清单 1. 单个对象表绑定的实现**

<typeAlias alias="Bank" type="com.example.domain.Bank"/>

<resultMap id="getBankResult" class="Bank">

<result property="id" column="B\_ID"/>

<result property="name" column="B\_NAME"/>

</resultMap>

<select id="getBankById" resultMap="getBankResult" parameterClass="string"/>

SELECT `BANKL`.`BANKID` AS `B\_ID`, `BANKL`.`NAME` AS `B\_NAME`

FROM `BANKL`

WHERE `BANKID` = #id#

</select/>

在某些特殊的对象中，有的时候需要将一个对象映射到多个表中，在这个时候就需要在映射时加入对两张表的选择。因此，需要对上面的简单的 sql 进行一些改动，加入一个两张表的 join 操作，以下是示例：

**清单 2. 单个对象多表绑定的实现**

<typeAlias alias="Customer" type="com.example.domain.Customer"/>

<resultMap id="getCustomerResult" class="Customer">

<result property="id" column="C\_ID"/>

<result property="name" column="C\_NAME"/>

<result property="birthday" column="C\_BIRTHDAY"/>

<result property="gender" column="C\_GENDER"/>

<result property="identityType" column="C\_IDENTITYTYPE"/>

<result property="identityId" column="C\_IDENTITYID"/>

<result property="consumePoint" column="C\_CONSUMEPOINT"/>

<result property="lastGetDate" column="C\_LASTGETDATE"/>

<result property="lastUseDate" column="C\_LASTUSEDATE"/>

</resultMap>

<select id="getCustomerById" resultMap="getCustomerResult" parameterClass="string"/>

SELECT

`CUMAS`.`ID` AS `C\_ID`, `CUMAS`.`NAME` AS `C\_NAME`, `CUMAS`.`BIRTHDAY` AS

`C\_BIRTHDAY`, `CUMAS`.`GENDER` AS `C\_GENDER`, `CUMAS`.`IDTYPE` AS `C\_IDENTITYTYPE`,

`CUMAS`.`IDNO` AS `C\_IDENTITYID`, `CUSPT`.`POINT` AS `C\_CONSUMEPOINT`, `CUSPT`.`LASTGET`

AS `C\_LASTGETDATE`, `CUSPT`.`LASTUSE` AS `C\_LASTUSEDATE`

FROM `CUMAS`, `CUSPT` WHERE **`ID` = #id# AND `CUMAS`.`ID` = `CUSPT`.`CUSID`**

<select/>

在上面这个例子中，Customer 这个对象被映射到 CUMAS 与 CUSPT 两个表中，每个表中包含了一部分字段，所以在选择的时候需要将两张表 join 起来。在进行删除，修改等操作时，需要注意的是要创建两条语句并在 java code 中对两条语句都进行调用，否则会导致数据不一致的现象发生：

**清单 3. 单个对象多表绑定的删除语句**

<delete id="deleteCustomer" parameterClass="string">

DELETE FROM `CUMAS` WHERE `ID` = #id#

</delete>

<delete id="deleteCustomer2" parameterClass="string">

DELETE FROM `CUSPT` WHERE `CUSID` = #id#

</delete>

**单主键与多主键**

在数据库中，一般需要对每条记录定义一个主键，大部分情况下采用一个自增长的 ID 来实现。这个时候有两种实现方法，第一种是用数据库自带的 id 增长功能，如何在 ibatis 中实现这种情况已在 iBatis 帮助文档中有详细说明，这里就不在赘述。第二种方法是采用类似 Toplink 中的实现方式，在 Java 中控制 ID 的增长，这种方法使用一张 sequence 表来存放所有的 id 记录，该表有两个字段，第一个为表名，第二个为当前最新的 id 值，表中的每一行对应一个自增长的 id。与该表对应的为一个 Java 的 Sequence 类与 iBatis 映射文件。在查询每个 sequence 时，通过 sequence name 将当前 id 值取出并设置到对象的 id 中。

在获取新的 sequence 时，需要注意的是对并发的控制，在同一时间只能有一个线程更新 sequence，否则将会导致主键重复。

**清单 4. Sequence 的实现**

**Sequence 的 Java 类**

public class Sequence{

protected String name;

protected String nextId;

public String getName() {

return name;

}

public void setName(String name) {

this.name = name;

}

public String getNextId() {

return nextId;

}

public void setNextId(String nextId) {

this.nextId = nextId;

}

}

**Sequence 的 SQL 映射**

<sqlMap namespace="Sequence">

<typeAlias alias="Sequence" type="com.example.Sequence"/>

<resultMap id="getSequenceResult" class="Sequence">

<result property="name" column="S\_NAME"/>

<result property="nextId" column="S\_NEXTID"/>

</resultMap>

<select id="getSequenceById" resultMap="getSequenceResult" parameterClass="string" >

SELECT

`SEQ`.`SEQNAME` AS `S\_NAME`, `SEQ`.`SEQVALUE` AS `S\_NEXTID` FROM `SEQ`

WHERE `SEQNAME` = #name#

</select>

<update id="updateSequence" parameterClass="Sequence">

UPDATE `B2CSEQ` SET `SEQVALUE`=#nextId# WHERE `SEQNAME` = #name#

</update>

</sqlMap>

**通过 Sequence 获取 ID 的方法**

public static synchronized String getNextId(String name) throws SQLException {

SqlMapSession sequenceSession=SessionManager.retrieveSession();

ISequence sequence = (ISequence) sequenceSession.queryForObject(

"getSequenceById", name);

if (sequence == null) {

throw new SQLException("Error: A null sequence was returned from the database" +

" (could not get next "+ name + " sequence).");

}

String id = sequence.getNextId();

sequence.setNextId((new BigDecimal(id).add(new BigDecimal(5))).toString());

sequenceSession.startTransaction();

sequenceSession.update("updateSequence", sequence);

sequenceSession.commitTransaction();

sequenceSession.close();

return String.valueOf(sequence.getNextId());

}

**常见的类关系与实现**

对于类与类之间的引用关系，最常见的是一对一，一对多与多对多。下面就讲述对这三种关系的实现。

**一对一映射**

在类与类之间关系中，一对一是最简单的一种。在每个类中有一个对其他类的引用，反之亦然。此时只需在 ibatis 的 result map 中建立对对象的引用关系即可：

**清单 5. 一对一的引用**

<result property="belongedCatalog" column="A\_BELONGEDCATALOG"

select="getArticleCatalogById"/>

在上面的 mapping 中，A\_BELONGEDCATALOG 为查询结果中的一个字段，而 getArticleCatalogById 为一个返回结果为 Catalog 的查询。因此 ibatis 在调用时，将会把 A\_BELONGEDCATALOG 作为参数传入 getArticleCatalogById 来生成一个 Catalog 实例的引用 , 并将改引用赋值给 belongedCatalog 属性。

**一对多映射**

一对多关系与 1 对 1 关系的区别之处在于类中引用的对象为一个列表而非单个对象，因此在 resultmap 中的 mapping 是一样的，唯一的区别在于关联的 select 返回的为一个列表而非单个对象（上例中的 getArticleCatalogById 的返回结果为一个对象，而对于一对多关系此方法将返回一个列表）。因此在这里不再给出实现。

**多对多**

在实现多对多关系时，每个类都可能包含多个对对方类的引用列表，通常需要建立一张关系表用来存储关系映射，如下所示：

**清单 6. 多对多的关系表**

CREATE TABLE `ARTICLE\_GROUP\_RELATION` (

`GROUPID` varchar(10) not null ,

`ARTICLEID` varchar(10) not null );

在这里 Article 与 Group 两个对象之间存在多对多关系，GROUPID 与 ARTICLEID 字段分别对应与 Group 表与 Article 表的主键。若在 Group 对象中有一个对 Article 列表的引用，首先需要在 Group 的 resultmap 中定义一个 List 的属性。这个属性会映射到一条根据 GroupID 查出 Article 的语句 getArticleByArticleGroupId，该查询需要将上面定义的关系表与 Article 表进行连接，返回结果为一个 Article 的列表 :

**清单 7. 多对多的关系映射**

<result property="articleList" column="AG\_ID" select="getArticleByArticleGroupId"/>

<select id="getArticleByArticleGroupId" resultMap="getArticleResult"

parameterClass="string">

SELECT

`ARTICLE`.`ID` AS `A\_ID`, `ARTICLE`.`NAME` AS `A\_NAME`, `ARTICLE`.`INDUSTRY`

AS `A\_ISINDUSTRY`, `ARTICLE`.`UPDATEDATE` AS `A\_UPDATEDATE`

FROM ` ARTICLE `, ` ARTICLE\_GROUP\_RELATION `

WHERE `ARTICLE\_GROUP\_RELATION`.`GROUPID` = #id# AND

`ARTICLE\_GROUP\_RELATION`.`ARTICLEID` = `ARTICLE`.`ID`

</select>

需要注意的是，在对应的增加 / 删除 / 修改操作中，也需要调用一个 SQL 语句来更新关系表，否则会导致表中的数据不一致，以删除为例：

**清单 8. 多对多的删除语句**

<delete id="deleteArticleGroupArticleList" parameterClass="string" >

DELETE FROM `ARTICLE\_GROUP\_RELATION` WHERE `GROUPID` = #id#

</delete>

**复合关系的分类与实现**

上一章介绍了类之间的关系的实现，本章在上一章的基础上，介绍了几种对象主键与继承类关系的实现。

**对象主键**

对象主键是一种非常特殊的主键实现，此时不是使用一个自动增长的 ID 作为类的主键，而是将类中所引用到的某个对象作为主键。因此，在建表的时候就需要将主键对象表中的主键字段作为该表中的主键字段保存，同样在进行 resultmap 的定义时需要建立一个对与其他对象的引用：

在下面这个例子中，UserCredit 对象中包含一个 createUser 对象并以该对象作为主键，而 User 对象本身包含了两个主键：用户 ID 与注册系统。因此在对 UserCredit 建表时将 USERID 跟 REGISTEREDSYSTEM 作为两个字段存放，对应的在映射时需要将这两个字段一起映射到 user 对象的查询上。对应的字段映射与查询语句如下：

**清单 9. 对象主键的关系映射**

<result property="createUser" column="userId= USERID ,

registeredSystem= REGISTEREDSYSTEM" select="getUserById"/>

<select id="getUserById" resultMap="getUserResult"

parameterClass="java.util.HashMap" >

SELECT

`SYUSR`.`USERID` AS `U\_USERID`, `SYUSR`.`SYSCODE` AS

`U\_REGISTEREDSYSTEM`, `SYUSR`.`NAME` AS `U\_NAME`,

`SYUSR`.`SYSNAME` AS `U\_DISPNAME`, `SYUSR`.`GENDER` AS `U\_GENDER`,

`SYUSR`.`MOBILE` AS `U\_MOBILE`, `SYUSR`.`EMAIL` AS `U\_EMAIL

FROM `SYUSR`

WHERE

`USERID` = #userId# AND `SYSCODE` = #registeredSystem#

</select>

<select id="getUserCreditByUser" resultMap="getUserCreditResult"

parameterClass="java.util.HashMap">

SELECT `NUMBER`, `NUMBERREMAIN`, ` USERID `, `SYSCODE`

FROM `CREDIT`

WHERE `CREDIT`.` USERID ` = #userId# AND `CREDIT`.`SYSCODE` = #registeredSystem#

</select>

需要注意的是，在 getUserById 中接受的参数为一个 HahsMap，这个 HashMap 中存放了两个字段：userId 和 registeredSystem。因此在映射是需要写成 column="userId= USERID , registeredSystem= REGISTEREDSYSTEM"的方式来传入一个 Map 参数。

对于对象主键的查询语句，可以将参数设为一个对象，也可以将参数设为一个 Map，在 Map 中放入 userId 与 registeredSystem，在查询时使用这两个值作为查询条件。 类似的在其他增加 / 删除 / 修改操作时也可以传入一个 HashMap 作为参数。

**一对多与多对多时的对象主键**

在上一小节的基础上，如果某个以对象为主键的类被其他类所引用并建立了一对多或者多对多的关系，此时在 mapping 时需要把前一章中使用的关联的 id 改成对象的主键。下面是一对多的示例，首先在 resultmap 中定义属性映射，然后定义了对应的查询方法。

**清单 10. 一对多对象主键的关系映射**

<result property="groupList" column="userId= USERID ,

registeredSystem= REGISTEREDSYSTEM" select=" getAttrListByUser "/>

<select id="getAttrListByUser" resultMap="getAttrResult"

parameterClass="java.util.HashMap">

SELECT `ATTRIBUTENAME`, `ATTRIBUTEVALUE`, ` USERID `, `SYSCODE`

FROM `ATTRIBUTE`

WHERE `ATTRIBUTE`.`USERID` = #userId# AND

`ATTRIBUTE`.`SYSCODE` = #registeredSystem#

</select>

将上面的对象主键与前面提到的多对多关系结合起来，可以得到最复杂的多对多的对象主键的实现。首先在关系表中需要加入对于对象主键的定义。假设 Article 对象为多主键对象，在 Article 中有一个 GroupList 的属性，Article 与 Group 有多对多的关系，以下为表的定义与 resultmap 的实现：

**清单 11. 一对多对象主键的关系映射**

CREATE TABLE `ARTICLE\_GROUP\_RELATION` (

`GROUPID` varchar(10) not null ,

`ARTICLETITLE` varchar(10) not null,

`ARTICLEUSER` varchar(10) not null );

<result property="groupList" column=" articleTitle=TITLE,

articleUser=USERID" select="getGroupByArticle"/>

<select id="getGroupByArticle" resultMap="getGroupResult"

parameterClass="java.util.HashMap">

SELECT

`GROUP`.`NAME` AS `G\_NAME`, `GROUP`.`ID` AS `G\_ID`

FROM ` GROUP `, ` ARTICLE\_GROUP\_RELATION `

WHERE `ARTICLE\_GROUP\_RELATION`.`GROUPID` = GROUP.`ID` AND

`ARTICLE\_GROUP\_RELATION`.`ARTICLETITLE` = #articleTitle# AND

`ARTICLE\_GROUP\_RELATION`.`ARTICLEUSER` = #articleUser#

</select>

<delete id="deleteArticleGroupArticleList" parameterClass="java.util.HashMap" >

DELETE FROM `ARTICLE\_GROUP\_RELATION`

WHERE `GROUPID` = #groupId# AND

`ARTICLE\_GROUP\_RELATION`.`ARTICLETITLE` = #articleTitle# AND

`ARTICLE\_GROUP\_RELATION`.`ARTICLEUSER` = #articleUser#

</delete>

上面的 getGroupByArticle 为一条根据 Article 查出 Group 的语句，该查询需要将关系表与 Groupe 表 join，返回结果为一个 Group 的列表。同样需要注意的是，在对应的增加 / 删除 / 修改操作中，也需要调用一个 SQL 语句来更新关系表，上面是删除的例子。

**类的继承的特殊处理**

类的继承关系是一种非常常见的类关系，在一般的应用中都会涉及。一般会将一些公有的属性与方法放在一个父类中，然后不同的子类继承父类。在进行数据库的映射时，对于继承关系可以有两种处理方法。 第一种是对于父类并不单独建表与之对应，而是分别对每个子类分别建表与做映射，所有父类的属性都在子类中保存有一份。这种方法的好处是在进行映射时非常简单，与一般类的处理相同，坏处是是并利用类的继承的特点，子类的表中会有大量重复的属性。 第二种方法是对于父类建立一张表与之对应，对于每一个子类也单独建立一张表与之对应，但是在子类当中并不保存父类的属性，父类的属性只保存在父类的表当中。对于所有的父类与子类，都采用一套相同的 id，这就是说如果子类的表中有一行 ID 值为 10 的数据，在父类表中也会有一行 ID 值为 10 的数据。这两行数据分别存放子类与父类中的属性值。为了区分具体的值是属于哪个子类，在父类中需要定义一个特殊字段 CLASSKEY，这个字段的值表示该行是属于哪个字段。这种方法也是在很多 O/R mapping 工具中常用的方法，下面将详细讲述这种方法的实现。 在下面的这里实现示例中有三个类：Customer（客户），ComanyCustomer（公司客户）， PersonCustomer（个人客户）。 Customer 为一个父类，ComanyCustomer 与 PersonCustomer 继承了 Customer。在数据库中对应的有三张表 CUSTOMER, COMPANYCUSTOMER 和 PERSONCUSTOMER。 在下面的实现中，“COM”为 ComanyCustome 类所对应的 Class Keyr，“PERSON”为 PersonCustomer 对应的 Class Key。在进行查询的时候需要将子表与父表相 join，在进行增加 / 删除 / 修改操作的时候，也需要同时更改父类表与子类表，如增加中需要同时调用下面的两条增加语句。

**清单 12. 继承关系的实现**

<select id="getCompanyCustomerByID" resultMap="getCompanyCustomerResult"

parameterClass="string">

SELECT

`CUSTOMER`.`NAME` AS `C\_NAME`, `COMPANYCUSTOMER`.`ID`

AS `C\_ID`, `COMPANYCUSTOMER`.`LICENSE` AS `C\_ LICENSE`

FROM `COMPANYCUSTOMER`, `CUSTOMER`

WHERE `COMPANYCUSTOMER`.`ID` = `CUSTOMER`.`ID` AND

`CUSTOMER`.`CLASSKEY` = ‘ COM ’ AND

`COMPANYCUSTOMER`.`ID` = #id#

</select>

<insert id="insertCustomer" parameterClass="Customer">

INSERT INTO `CUSTOMER`

(`CUSTOMER`.`ID`, `CUSTOMER`.`CLASSKEY`, `CUSTOMER`.` NAME `)

VALUES ( #id#, #classKey#，#name#)

</insert>

<insert id="insertComanyCustomer" parameterClass="ComanyCustomer">

INSERT INTO `COMPANYCUSTOMER`

(`COMPANYCUSTOMER`.`ID`, `COMPANYCUSTOMER`.` NAME `)

VALUES ( #id#, #license#)

</insert>

<select id="getClassKeyByID" resultClass="string” parameterClass="string">

SELECT

`CUSTOMER`.` CLASSKEY` AS `C\_ CLASSKEY`

FROM `CUSTOMER`

WHERE `CUSTOMER`.`ID` = #id#

</select>

继承关系比其他关系复杂的实现是在 java 类也需要做一些特殊的处理，在引用 Customer 类的时候，可以直接引用父类或子类。引用父类时可能指向的是一个 ComanyCustomer 的实例，也可能是一个 PersonalCustomer 的实例。然而在仅仅知道 id 的时候无法判断是哪一个具体的子类，在这个时候，就需要根据 ClassKey 来判读具体是哪一个实例了。首先需要有一个方法获取 ClassKey，然后根据 ClassKey 来调用获取具体子类的语句。

**清单 13. 继承关系的 Java 实现**

public Customer getCustomerById(String id)

{

String classKey=(String) sqlSession.queryForObject("getClassKeyByID", id);

if(“COM”.equals(classKey))

return (Customer) sqlSession.queryForObject("getCompanyCustomerByID", id);

else

if(“PERSON”.equals(classKey))

return (Customer) sqlSession.queryForObject("getPersonCustomerByID", id);

}

**总结**

本文介绍了关于 iBatis 开发中常用的映射的实现，由于 iBatis 的所有语句必须靠开发人员生成，上面描述的种种对象关系和给出的实现实例将对开发 iBatis 的应用具有非常大的指导意义。