OpenDoc Series'

iBATIS 2.0 开发指南

V1.0

作者：夏昕 xiaxin(at)gmail.com

So many open source projects. Why not **Open** your **Doc**uments? **J**

文档说明

参与人员：

作者	联络
夏昕	xiaxin(at)gmail.com

(at) 为 email @ 符号

发布记录

版本	日期	作者	说明
0.0	2004.8.1	夏昕	第一版
1.0	2004.9.1	夏昕	补充 ibatis in Spring 部分

OpenDoc 版权说明

如果本文对您有些许帮助，表达谢意的最好方式，是将您发现的问题和文档改进意见及时反馈给作者。当然，倘若有时间和能力，能为技术群体无偿贡献自己的所学为最好的回馈。

另外，笔者近来试图就日本、印度的软件开发模式进行一些调研。如果诸位可以赠阅日本、印度软件研发过程中的需求、设计文档以供研究，感激不尽！

ibatis 开发指南

ibatis 开发指南

相对 **Hibernate** 和 **Apache OJB** 等“一站式”**ORM** 解决方案而言，**ibatis** 是一种“半自动化”的 **ORM** 实现。

所谓“半自动”，可能理解上有点生涩。纵观目前主流的 **ORM**，无论 **Hibernate** 还是 **Apache OJB**，都对数据库结构提供了较为完整的封装，提供了从 **POJO** 到数据库表的全套映射机制。程序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。程序员甚至不需要对 **SQL** 的熟练掌握，**Hibernate/OJB** 会根据制定的存储逻辑，自动生成对应的 **SQL** 并调用 **JDBC** 接口加以执行。

大多数情况下（特别是对新项目，新系统的开发而言），这样的机制无往不利，大有一统天下的势头。但是，在一些特定的环境下，这种一站式的解决方案却未必灵光。

在笔者的系统咨询工作过程中，常常遇到以下情况：

1. 系统的部分或全部数据来自现有数据库，处于安全考虑，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. 开发规范中要求，所有牵涉到业务逻辑部分的数据库操作，必须在数据库层由存储过程实现（就笔者工作所面向的金融行业而言，工商银行、中国银行、交通银行，都在开发规范中严格指定）
3. 系统数据处理量巨大，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，再次举起 **Hibernate** 大刀，却发现刀锋不再锐利，甚至无法使用，奈何？恍惚之际，只好再摸出 **JDBC** 准备拼死一搏.....，说得未免有些凄凉，直接使用 **JDBC** 进行数据库操作实际上也是不错的选择，只是拖沓的数据库访问代码，乏味的字段读取操作令人厌烦。

“半自动化”的 **ibatis**，却刚好解决了这个问题。

这里的“半自动化”，是相对 **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，“全自动”**ORM** 实现了 **POJO** 和数据库表之间的映射，以及 **SQL** 的自动生成和执行。而 **ibatis** 的着力点，则在于 **POJO** 与 **SQL** 之间的映射关系。也就是说，**ibatis** 并不会为程序员在运行期自动生成 **SQL** 执行。具体的 **SQL** 需要程序员编写，然后通过映射配置文件，将 **SQL** 所需的参数，以及返回的结果字段映射到指定 **POJO**。

使用 **ibatis** 提供的 **ORM** 机制，对业务逻辑实现人员而言，面对的是纯粹的 **Java** 对象，这一层与通过 **Hibernate** 实现 **ORM** 而言基本一致，而对于具体的数据操作，**Hibernate** 会自动生成 **SQL** 语句，而 **ibatis** 则要求开发者编写具体的 **SQL** 语句。相对 **Hibernate** 等“全自动”**ORM** 机制而言，**ibatis** 以 **SQL** 开发的工作量和数据库移植性上的让步，为系统设计提供了更大的自由空间。作为“全自动”**ORM** 实现的一种有益补充，**ibatis** 的出现显得别具意义。

ibatis Quick Start

准备工作

1. 下载 ibatis 软件包（http://www.ibatis.com）。
2. 创建测试数据库，并在数据库中创建一个 t_user 表，其中包含三个字段：
   - Ø id(int)
   - Ø name(varchar)
   - Ø sex(int)。
3. 为了在开发过程更加直观，我们需要将 ibatis 日志打开以便观察 ibatis 运作的细节。ibatis 采用 Apache common_logging，并结合 Apache log4j 作为日志输出组件。在 CLASSPATH 中新建 log4j.properties 配置文件，内容如下：

```
log4j.rootLogger=DEBUG, stdout

log4j.appender.stdout=org.apache.log4j.ConsoleAppender
log4j.appender.stdout.layout=org.apache.log4j.PatternLayout
log4j.appender.stdout.layout.ConversionPattern=%c{1} - %m%n

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

构建 ibatis 基础代码

ibatis 基础代码包括：

1. **ibatis** 实例配置

一个典型的配置文件如下（具体配置项目的含义见后）：

```
<?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>

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

  <transactionManager type="JDBC">
```

```
    <dataSource type="SIMPLE">
      <property name="JDBC.Driver"
value="com.p6spy.engine.spy.P6SpyDriver"/>
      <property name="JDBC.ConnectionURL"
value="jdbc:mysql://localhost/sample"/>
      <property name="JDBC.Username" value="user"/>
      <property name="JDBC.Password" value="mypass"/>
      <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="com/ibatis/sample/User.xml"/>

</sqlMapConfig>
```

2. **POJO**（**P**lain Ordinary **J**ava **O**bject）

下面是我们用作示例的一个 **POJO:**

```
public class User implements Serializable {

    private Integer id;

    private String name;

    private Integer sex;

    private Set addresses = new HashSet();
    /** default constructor */
    public User() {
    }

    public Integer getId() {
        return this.id;
```

```
    }

    public void setId(Integer id) {
        this.id = id;
    }

    public String getName() {
        return this.name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public Integer getSex() {
        return this.sex;
    }

    public void setSex(Integer sex) {
        this.sex = sex;
    }
}
```

3. 映射文件

与 **Hibernate** 不同。因为需要人工编写 **SQL** 代码，**ibatis** 的映射文件一般采用手动编写（通过 **Copy/Paste**，手工编写映射文件也并没想象中的麻烦）。

针对上面 **POJO** 的映射代码如下：

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

<!DOCTYPE sqlMap
    PUBLIC "-//iBATIS.com//DTD SQL Map 2.0//EN"
    "http://www.ibatis.com/dtd/sql-map-2.dtd">

<sqlMap namespace="User">

 <typeAlias alias="user" type="com.ibatis.sample.User"/>
  <select id="getUser"
    parameterClass="java.lang.String"
    resultClass="user">
    <![CDATA[
    select
      name,
      sex
    from t_user
```

```
    where name = #name#
    ]]>
  </select>

  <update id="updateUser"
    parameterClass="user">
    <![CDATA[
    UPDATE t_user
        SET
            name=#name#,
            sex=#sex#
    WHERE id = #id#
    ]]>
  </update>

  <insert id="insertUser"
    parameterClass="user"
    >
        INSERT INTO t_user (
            name,
            sex)
        VALUES (
            #name#,
            #sex#
        )
  </insert>

  <delete id="deleteUser"
    parameterClass="java.lang.String">
    delete from t_user
    where id = #value#
  </delete>

</sqlMap>
```

从上面的映射文件可以看出，通过<insert>、<delete>、<update>、<select>四个节点，我们分别定义了针对 **TUser** 对象的增删改查操作。在这四个节点中，我们指定了对应的 **SQL** 语句，以 **update** 节点为例：

```
.....
  <update id="updateUser"                    (1)
    parameterClass="user">                   (2)
    <![CDATA[                                (3)
    UPDATE t_user                            (4)
        SET (
```

```
            name=#name#,                    (5)
            sex=#sex#                       (6)
            )
    WHERE id = #id#                         (7)
    ]]>
  </update>
……
```

(1) ID

指定了操作 ID，之后我们可以在代码中通过指定操作 id 来执行此节点所定义的操作，如：

```
sqlMap.update("updateUser",user);
```

ID 设定使得在一个配置文件中定义两个同名节点成为可能（两个 update 节点，以不同 id 区分）

(2) parameterClass

指定了操作所需的参数类型，此例中 update 操作以 com.ibatis.sample.User 类型的对象作为参数，目标是将提供的 User 实例更新到数据库。

parameterClass="user"中，user 为“com.ibatis.sample.User”类的别名，别名可通过 typeAlias 节点指定，如示例配置文件中的：

```
<typeAlias alias="user" type="com.ibatis.sample.User"/>
```

(3) <![CDATA[……]]>

通过<![CDATA[……]]>节点，可以避免 SQL 中与 XML 规范相冲突的字符对 XML 映射文件的合法性造成影响。

(4) 执行更新操作的 SQL，这里的 SQL 即实际数据库支持的 SQL 语句，将由 ibatis 填入参数后交给数据库执行。

(5) SQL 中所需的用户名参数,“#name#”在运行期会由传入的 user 对象的 name 属性填充。

(6) SQL 中所需的用户性别参数“#sex#”，将在运行期由传入的 user 对象的 sex 属性填充。

(7) SQL 中所需的条件参数“#id#”，将在运行期由传入的 user 对象的 id 属性填充。

对于这个示例，ibatis 在运行期会读取 id 为“updateUser”的 update 节点的 SQL 定义，并调用指定的 user 对象的对应 getter 方法获取属性值，并用此属性值，对 SQL 中的参数进行填充后提交数据库执行。

此例对应的应用级代码如下，其中演示了 ibatis SQLMap 的基本使用方法：

```
String resource ="com/ibatis/sample/SqlMapConfig.xml";
Reader reader;
```

```
reader = Resources.getResourceAsReader(resource);

XmlSqlMapClientBuilder xmlBuilder =
new XmlSqlMapClientBuilder();
SqlMapClient sqlMap = xmlBuilder.buildSqlMap(reader);
  //sqlMap系统初始化完毕，开始执行update操作
try{
  sqlMap.startTransaction();

  User user = new User();
  user.setId(new Integer(1));
  user.setName("Erica");
  user.setSex(new Integer(1));


  sqlMap.update("updateUser",user);

  sqlMap.commitTransaction();
finally{
  sqlMap.endTransaction();
}
```

其中，SqlMapClient 是 ibatis 运作的核心，所有操作均通过 SqlMapClient 实例完成。

可以看出，对于应用层而言，程序员面对的是传统意义上的数据对象，而非 JDBC 中烦杂的 ResultSet，这使得上层逻辑开发人员的工作量大大减轻，同时代码更加清晰简洁。
数据库操作在映射文件中加以定义，从而将数据存储逻辑从上层逻辑代码中独立出来。
而底层数据操作的 SQL 可配置化，使得我们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，这在依赖 SQL 自动生成的“全自动”ORM 机制中是所难以实现的。

ibatis 配置

结合上面示例中的 ibatis 配置文件。下面是对配置文件中各节点的说明：

```
<?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>

  <settings                                                    (1)
    cacheModelsEnabled="true"
    enhancementEnabled="true"
    lazyLoadingEnabled="true"
    errorTracingEnabled="true"
    maxRequests="32"
    maxSessions="10"
    maxTransactions="5"
    useStatementNamespaces="false"
    />

  <transactionManager type="JDBC">                             (2)
    <dataSource type="SIMPLE">                                 (3)
      <property name="JDBC.Driver"
value="com.p6spy.engine.spy.P6SpyDriver"/>
      <property name="JDBC.ConnectionURL"
value="jdbc:mysql://localhost/sample"/>
      <property name="JDBC.Username" value="user"/>
      <property name="JDBC.Password" value="mypass"/>
      <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="com/ibatis/sample/User.xml"/>   (4)
  <sqlMap resource="com/ibatis/sample/Address.xml"/>

</sqlMapConfig>
```

(1) Settings 节点

参数	描述
cacheModelsEnabled	是否启用 **SqlMapClient** 上的缓存机制。 建议设为"true"
enhancementEnabled	是否针对 **POJO** 启用字节码增强机制以提升 **getter/setter** 的调用效能，避免使用 **Java Reflect** 所带来的性能开销。 同时，这也为 **Lazy Loading** 带来了极大的性能提升。 建议设为"true"
errorTracingEnabled	是否启用错误日志，在开发期间建议设为"true"以方便调试
lazyLoadingEnabled	是否启用延迟加载机制，建议设为"true"
maxRequests	最大并发请求数（**Statement** 并发数）
maxTransactions	最大并发事务数
maxSessions	最大 **Session** 数。即当前最大允许的并发 **SqlMapClient** 数。 **maxSessions** 设定必须介于 **maxTransactions** 和 **maxRequests** 之间，即 maxTransactions<maxSessions=<maxRequests
useStatementNamespaces	是否使用 **Statement** 命名空间。 这里的命名空间指的是映射文件中，**sqlMap** 节点的 **namespace** 属性，如在上例中针对 **t_user** 表的映射文件 **sqlMap** 节点： <sqlMap namespace="User"> 这里，指定了此 **sqlMap** 节点下定义的操作均从属于"User"命名空间。 在 useStatementNamespaces="true"的情况下，**Statement** 调用需追加命名空间，如：

	sqlMap.update("User.updateUser",user);
	否则直接通过 Statement 名称调用即可，如：
	sqlMap.update("updateUser",user);
	但请注意此时需要保证所有映射文件中，**Statement** 定义无重名。

⑵ transactionManager 节点

transactionManager 节点定义了 ibatis 的事务管理器，目前提供了以下几种选择：

- Ø JDBC
  通过传统 JDBC Connection.commit/rollback 实现事务支持。
- Ø JTA
  使用容器提供的 JTA 服务实现全局事务管理。
- Ø EXTERNAL
  外部事务管理，如在 EJB 中使用 ibatis，通过 EJB 的部署配置即可实现自动的事务管理机制。此时 ibatis 将把所有事务委托给外部容器进行管理。此外，通过 Spring 等轻量级容器实现事务的配置化管理也是一个不错的选择。关于结合容器实现事务管理，参见“高级特性”中的描述。

⑶ dataSource 节点

dataSource 从属于 transactionManager 节点，用于设定 ibatis 运行期使用的 DataSource 属性。

type 属性： dataSource 节点的 type 属性指定了 dataSource 的实现类型。可选项目：

- Ø **SIMPLE**：
  SIMPLE 是 ibatis 内置的 dataSource 实现，其中实现了一个简单的数据库连接池机制，对应 ibatis 实现类为 com.ibatis.sqlmap.engine.datasource.SimpleDataSourceFactory。
- Ø **DBCP**：
  基于 Apache DBCP 连接池组件实现的 DataSource 封装，当无容器提供 DataSource 服务时，建议使用该选项，对应 ibatis 实现类为 com.ibatis.sqlmap.engine.datasource.DbcpDataSourceFactory。
- Ø JNDI：
  使用 J2EE 容器提供的 DataSource 实现，DataSource 将通过指定的 JNDI Name 从容器中获取。对应 ibatis 实现类为 com.ibatis.sqlmap.engine.datasource.JndiDataSourceFactory。

dataSource 的子节点说明（SIMPLE&DBCP）：

参数	描述
JDBC.Driver	**JDBC** 驱动。 如：org.gjt.mm.mysql.Driver
JDBC.ConnectionURL	数据库 **URL**。 如：jdbc:mysql://localhost/sample 如果用的是 SQLServer JDBC Driver，需要在 url 后追加 SelectMethod=Cursor 以获得 JDBC 事务的多 Statement 支持。
JDBC.Username	数据库用户名
JDBC.Password	数据库用户密码
Pool.MaximumActiveConnections	数据库连接池可维持的最大容量。
Pool.MaximumIdleConnections	数据库连接池中允许的挂起（**idle**）连接数。

以上子节点适用于 **SIMPLE** 和 DBCP 模式，分别针对 **SIMPLE** 和 DBCP 模式的 DataSource 私有配置节点如下：

SIMPLE:

参数	描述
Pool.MaximumCheckoutTime	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。（毫秒）
Pool.TimeToWait	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。（毫秒）
Pool.PingQuery	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。 检测语句应该是一个最简化的无逻辑 **SQL**。 如“**select 1 from t_user**”，如果执行此语句成功，连接池管理器将认为此连接处于可用状态。
Pool.PingEnabled	是否允许检测连接状态。
Pool.PingConnectionsOlderThan	对持续连接时间超过设定值（毫秒）的连接进行检测。

Pool.PingConnectionsNotUsedFor	对空闲超过设定值（毫秒）的连接进行检测。

DBCP:

参数	描述
Pool.MaximumWait	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。（毫秒）
Pool.ValidationQuery	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。 检测语句应该是一个最简化的无逻辑 **SQL**。 如“**select 1 from t_user**”，如果执行此语句成功，连接池管理器将认为此连接处于可用状态。
Pool.LogAbandoned	当数据库连接被废弃时，是否打印日志。
Pool.RemoveAbandonedTimeout	数据库连接被废弃的最大超时时间
Pool.RemoveAbandoned	当连接空闲时间超过 RemoveAbandonedTimeout 时，是否将其废弃。

JNDI 由于大部分配置是在应用服务器中进行，因此 ibatis 中的配置相对简单，下面是分别使用 JDBC 和 JTA 事务管理的 JDNI 配置：

使用 JDBC 事务管理的 JNDI DataSource 配置

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

```
<transactionManager type="JTA" >
  <property name="UserTransaction"
     value="java:/ctx/con/UserTransaction"/>
  <dataSource type="JNDI">
     <property name="DataSource"
     value="java:comp/env/jdbc/myDataSource"/>
  </dataSource>
```

```
</transactionManager>
```

⑷ **sqlMap 节点**

sqlMap 节点指定了映射文件的位置，配置中可出现多个 sqlMap 节点，以指定项目内所包含的所有映射文件。

ibatis 基础语义

XmlSqlMapClientBuilder

XmlSqlMapClientBuilder 是 ibatis 2.0 之后版本新引入的组件,用以替代 1.x 版本中的 XmlSqlMapBuilder。其作用是根据配置文件创建 SqlMapClient 实例。

SqlMapClient

SqlMapClient 是 ibatis 的核心组件，提供数据操作的基础平台。SqlMapClient 可通过 XmlSqlMapClientBuilder 创建:

```
String resource ="com/ibatis/sample/SqlMapConfig.xml";
Reader reader;

reader = Resources.getResourceAsReader(resource);

XmlSqlMapClientBuilder xmlBuilder =
new XmlSqlMapClientBuilder();

SqlMapClient sqlMap = xmlBuilder.buildSqlMap(reader);
```

"com/ibatis/sample/SqlMapConfig.xml"指明了配置文件在 CLASSPATH 中的相对路径。XmlSqlMapClientBuilder 通过接受一个 Reader 类型的配置文件句柄，根据配置参数，创建 SqlMapClient 实例。

SqlMapClient 提供了众多数据操作方法，下面是一些常用方法的示例，具体说明文档请参见 ibatis java doc，或者 ibatis 官方开发手册。

SqlMapClient 基本操作示例

以下示例摘自 ibatis 官方开发手册，笔者对其进行了重新排版以获得更好的阅读效果。

例 1: 数据写入操作(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 2: 数据查询 (select)

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

例 3: 在指定对象中存放查询结果(select)

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

例 4: 执行批量查询 (select)

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

例 5: 关于 AutoCommit

```
//没有预先执行startTransaction时，默认为auto_commit模式
int rows = sqlMap.insert ("insertProduct", product);
```

例 6:查询指定范围内的数据

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

例7: 结合RowHandler进行查询(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);
      }
}
sqlMap.startTransaction();
RowHandler rowHandler = new MyRowHandler();
List list = sqlMap.queryForList ("getProductList", null,
rowHandler);
sqlMap.commitTransaction();
```

例8: 分页查询 (select)

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

例9: 基于Map的批量查询 (select)

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

OR 映射

相对 Hibernate 等 ORM 实现而言，ibatis 的映射配置更为简洁直接，下面是一个典型的配置文件。

```
<!DOCTYPE sqlMap
    PUBLIC "-//iBATIS.com//DTD SQL Map 2.0//EN"
    "http://www.ibatis.com/dtd/sql-map-2.dtd">

<sqlMap namespace="User">
    <!--模块配置-->
    <typeAlias alias="user" type="com.ibatis.sample.User"/>

    <cacheModel id="userCache" type="LRU">
        <flushInterval hours="24"/>
        <flushOnExecute statement=" updateUser"/>
        <property name="size" value="1000" />
    </cacheModel>

  <!—Statement配置-->
  <select id="getUser"
    parameterClass="java.lang.String"
    resultClass="user"
    cacheModel="userCache"
    >
    <![CDATA[
    select
      name,
      sex
    from t_user
    where name = #name#
    ]]>
  </select>

  <update id="updateUser"
    parameterClass="user">
    UPDATE t_user
        SET
            name=#name#,
            sex=#sex#
    WHERE id = #id#
  </update>
```

```
</sqlMap>
```

可以看到，映射文件主要分为两个部分：模块配置和 Statement 配置。
模块配置包括：

Ø typeAlias 节点：

定义了本映射文件中的别名，以避免过长变量值的反复书写，此例中通过 typeAlias 节点为类"com.ibatis.sample.User"定义了一个别名"user"，这样在本配置文件的其他部分，需要引用"com.ibatis.sample.User"类时，只需以其别名替代即可。

Ø cacheModel 节点

定义了本映射文件中使用的 Cache 机制：

```
<cacheModel id="userCache" type="LRU">
  <flushInterval hours="24"/>
  <flushOnExecute statement="updateUser"/>
  <property name="size" value="1000" />
</cacheModel>
```

这里申明了一个名为"userCache"的 cacheModel，之后可以在 Statement 申明中对其进行引用：

```
<select id="getUser"
    parameterClass="java.lang.String"
    resultClass="user"
    cacheModel="userCache"
    >
```

这表明对通过 id 为"getUser"的 Select statement 获取的数据，使用 cacheModel "userCache"进行缓存。之后如果程序再次用此 Statement 进行数据查询，即直接从缓存中读取查询结果，而无需再去数据库查询。

cacheModel 主要有下面几个配置点：

- flushInterval：
  设定缓存有效期，如果超过此设定值，则将此 CacheModel 的缓存清空。

- size：
  本 CacheModel 中最大容纳的数据对象数量。

- flushOnExecute：
  指定执行特定 Statement 时，将缓存清空。如 updateUser 操作将更新数据库中的用户信息，这将导致缓存中的数据对象与数据库中的实际数据发生偏差，因此必须将缓存清空以避免脏数据的出现。

关于 Cache 的深入探讨，请参见“高级特性”中的相关章节。

Statement 配置：

Statement 配置包含了数个与 SQL Statement 相关的节点，分别为：

- statement
- insert
- delete
- update
- select
- procedure

其中，statement 最为通用，它可以替代其余的所有节点。除 statement 之外的节点各自对应了 SQL 中的同名操作（procedure 对应存储过程）。
使用 statement 定义所有操作固然可以达成目标，但缺乏直观性，建议在实际开发中根据操作目的，各自选用对应的节点名加以申明。一方面，使得配置文件更加直观，另一方面，也可借助 DTD 对节点申明进行更有针对性的检查，以避免配置上的失误。

各种类型的 Statement 配置节点的参数类型基本一致，区别在于数量不同。如 insert、update、delete 节点无需返回数据类型定义（总是 int）。

主要的配置项如下：

statement:

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

select:

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

Insert:

```
<insert id="statementName"
    [parameterClass="some.class.Name"]
    [parameterMap="nameOfParameterMap"]
>
    insert into t_user
        (name,sex)
        values
        ([?|#propertyName#],[?|#propertyName#])
</insert>
```

Update：

```
<update id="statementName"
    [parameterClass="some.class.Name"]
    [parameterMap="nameOfParameterMap"]
>
        UPDATE t_user
            SET
                name=[?|#propertyName#],
                sex=[?|#propertyName#]
        WHERE id = [?|#propertyName#]

</update>
```

Delete：

```
<delete id="statementName"
    [parameterClass="some.class.Name"]
    [parameterMap="nameOfParameterMap"]
>
    delete from t_user
        where id = [?|#propertyName#]
</delete>
```

其中以“[]”包围的部分为可能出现的配置栏目。

参数	描述
parameterClass	参数类。指定了参数的完整类名（包括包路径）。 可通过别名避免每次重复书写冗长的类名。
resultClass	结果类。指定结果类型的完整类名（包括包路径） 可通过别名避免每次重复书写冗长的类名。
parameterMap	参数映射，需结合 **parameterMap** 节点对映射关系加以定义。 对于存储过程之外的 **statement** 而言，建议使用 parameterClass 作为参数配置方式，一方面避

	免了参数映射配置工作，另一方面其性能表现也更加出色。
resultMap	结果映射，需结合 resultMap 节点对映射关系加以定义。
cacheModel	**statement** 对应的 **Cache** 模块。

对于参数定义而言，尽量使用 parameterClass，即直接将 POJO 作为 statement 的调用参数，这样在 SQL 中可以直接将 POJO 的属性作为参数加以设定，如：

```
<update id="updateUser"
  parameterClass="com.ibatis.sample.User">
  UPDATE t_user
      SET
          name=#name#,
          sex=#sex#
  WHERE id = #id#
</update>
```

这里将 com.ibatis.sample.User 类设定为 update statement 的参数，之后，我们即可在 SQL 中通过#propertyName#对 POJO 的属性进行引用。如上例中的：

```
SET name=#name#, sex=#sex# WHERE id=#id#
```

运行期，ibatis 将通过调用 User 对象的 getName、getSex 和 getId 方法获得相应的参数值，并将其作为 SQL 的参数。

如果 parameterClass 中设定的是 jdk 的中的简单对象类型，如 String、Integer，ibatis 会直接将其作为 SQL 中的参数值。

我们也可以将包含了参数数据的 Map 对象传递给 Statement，如：

```
<update id="updateUser"
  parameterClass="java.util.Map">
  UPDATE t_user
      SET
          name=#name#,
          sex=#sex#
  WHERE id = #id#
</update>
```

这里传入的参数就是一个 Map 对象，ibatis 将以 key "name"、"sex"、"id"从中提取对应的参数值。

同样的原理，我们也可以在 resultMap 中设定返回类型为 map。

```
  <select id="getUser"
  parameterClass="java.lang.String"
  resultClass="java.util.Map">
```

```
    <![CDATA[
    select
          id,
          name,
          sex
        from t_user
    where id = #id#
        ]]>
    </select>
```

返回的结果将以各字段名为 key 保存在 Map 对象中返回。

在 SQL 中设定参数名时，可以同时指定参数类型，如：

```
SET  name=#name:VARCHAR#,sex=#sex:NUMERIC# WHERE
id=#id:NUMERIC#
```

对于返回结果而言，如果是 select 语句，建议也采用 resultClass 进行定义，如：

```
<select id="getUser"
    parameterClass="java.lang.String"
    resultClass="user">
    <![CDATA[
    select
      name,
      sex
    from t_user
    where name = #name#
    ]]>
</select>
```

ibatis 会自动根据 select 语句中的字段名，调用对应 POJO 的 set 方法设定属性值，如上例中，ibatis 会调用 setName，setSex 方法将 Select 语句返回的数据赋予相应的 POJO 实例。

有些时候，数据库表中的字段名过于晦涩，而为了使得代码更易于理解，我们希望字段映射到 POJO 时，采用比较易读的属性名，此时，我们可以通过 Select 的 as 字句对字段名进行转义，如（假设我们的书库中对应用户名的字段为 xingming，对应性别的字段为 xingbie）：

```
    select
      xingming as name,
      xingbie as sex
    from t_user
    where id = #id#
```

ibatis 会根据转义后的字段名进行属性映射（即调用 POJO 的 setName 方法而不是 setXingming 方法）。

parameterMap 和 resultMap 实现了 POJO 到数据库字段的映射配置，下面是一个例子：

```
<resultMap id="get_user_result" class="user">
    <result property="name" column="xingming"
    jdbcType="VARCHAR" javaType="java.lang.String"/>
    <result property="sex" column="xingbie"
    jdbcType="int" javaType="java.lang.Integer"/>
    <result property="id" column="id"
    jdbcType="int" javaType="java.lang.Integer"/>
</resultMap>

<parameterMap id="update_user_para" class="redemption" >
    <parameter property="name"
       jdbcType="VARCHAR"
       javaType="java.lang.String"
       nullValue=""
    />
    <parameter property="sex"
       jdbcType="int"
       javaType="java.lang.Integer"
       nullValue=""
    />
</parameterMap>
```

Parameter 的 nullValue 指定了如果参数为空（null）时的默认值。

之后我们即可在 statement 申明中对其进行引用，如：

```
  <procedure id="getUserList"
    resultMap="get_user_result"
    >
    {call sp_getUserList()}
  </procedure>

  <procedure id="doUserUpdate"
    parameterMap="update_user_para"
    >
    {call sp_doUserUpdate(#id#,#name#,#sex#)}
  </procedure>
```

一般而言，对于 insert、update、delete、select 语句，优先采用 parameterClass 和 resultClass。

parameterMap 使用较少，而 resultMap 则大多用于嵌套查询以及存储过程的

处理，之所以这样，原因是由于存储过程相对而言比较封闭（很多情况下需要调用现有的存储过程，其参数命名和返回的数据字段命名往往不符合 Java 编程中的命名习惯，并且由于我们难以通过 Select SQL 的 as 子句进行字段名转义，无法使其自动与 POJO 中的属性名相匹配）。此时，使用 resultMap 建立字段名和 POJO 属性名之间的映射关系就显得非常有效。另一方面，由于通过 resultMap 指定了字段名和字段类型，ibatis 无需再通过 JDBC ResultSetMetaData 来动态获取字段信息，在一定程度上也提升了性能表现。

ibatis 高级特性

数据关联

至此，我们讨论的都是针对独立数据的操作。在实际开发中，我们常常遇到关联数据的情况，如 User 对象拥有若干 Address 对象，每个 Address 对象描述了对应 User 的一个联系地址，这种情况下，我们应该如何处理？

通过单独的 Statement 操作固然可以实现（通过 Statement 用于读取用户数据，再手工调用另外一个 Statement 根据用户 ID 返回对应的 Address 信息）。不过这样未免失之繁琐。下面我们就看看在 ibatis 中，如何对关联数据进行操作。

ibatis 中，提供了 Statement 嵌套支持，通过 Statement 嵌套，我们即可实现关联数据的操作。

一对多关联

下面的例子中，我们首选读取 t_user 表中的所有用户记录，然后获取每个用户对应的所有地址信息。

配置文件如下：

```
<sqlMap namespace="User">
  <typeAlias alias="user" type="com.ibatis.sample.User"/>
  <typeAlias alias="address" type="com.ibatis.sample.Address"/>

  <resultMap id="get-user-result" class="user">
      <result property="id" column="id"/>
      <result property="name" column="name"/>
      <result property="sex" column="sex"/>
      <result property="addresses" column="id"
              select="User.getAddressByUserId"/>
  </resultMap>

  <select id="getUsers"
      parameterClass="java.lang.String"
      resultMap="get-user-result">
      <![CDATA[
      select
        id,
```

```
        name,
        sex
      from t_user
      where id = #id#
      ]]>
  </select>

  <select id="getAddressByUserId"
      parameterClass="int"
      resultClass="address">
       <![CDATA[
       select
         address,
         zipcode
    from t_address
    where user_id = #userid#
    ]]>
  </select>
</sqlMap>
```

对应代码：

```
String resource ="com/ibatis/sample/SqlMapConfig.xml";
Reader reader;

reader = Resources.getResourceAsReader(resource);

XmlSqlMapClientBuilder xmlBuilder = new XmlSqlMapClientBuilder();
sqlMap = xmlBuilder.buildSqlMap(reader);
//sqlMap系统初始化完毕

List userList = sqlMap.queryForList("User.getUsers", "");

for (int i = 0; i < userList.size(); i++) {

  User user = (User)userList.get(i);
  System.out.println("==>" + user.getName());

  for (int k = 0; k < user.getAddresses().size(); k++) {
      Address addr = (Address) user.getAddresses().get(k);
      System.out.println(addr.getAddress());
  }

}
```

这里通过在 resultMap 中定义嵌套查询 `getAddressByUserId`，我们实现了关联数据的读取。

实际上，这种方式类似于前面所说的通过两条单独的 Statement 进行关联数据的读取，只是将关联关系在配置中加以描述，由 ibatis 自动完成关联数据的读取。

需要注意的是，这里有一个潜在的性能问题，也就是所谓“n+1”Select 问题。

注意上面示例运行过程中的日志输出：

```
……
PreparedStatement - {pstm-100001} PreparedStatement: select id, name, sex from t_user
……
PreparedStatement - {pstm-100004} PreparedStatement: select address, zipcode from t_address where user_id = ?
……
PreparedStatement - {pstm-100007} PreparedStatement: select address,zipcode from t_address where user_id = ?
```

第一条 PreparedStatement 将 t_user 表中的所有数据读取出来（目前 t_user 表中有两条测试数据），随即，通过两次 Select 操作，从 t_address 表中读取两个用户所关联的 Address 记录。

如果 t_user 表中记录较少，不会有明显的影响，假设 t_user 表中有十万条记录，那么这样的操作将需要 100000＋1 条 Select 语句反复执行才能获得结果，无疑，随着记录的增长，这样的开销将无法承受。

之所以在这里提及这个问题，目的在于引起读者的注意，在系统设计中根据具体情况，采用一些规避手段（如使用存储过程集中处理大批量关联数据），从而避免因为这个问题而引起产品品质上的缺陷。

一对一关联

一对一关联是一对多关联的一种特例。这种情况下，如果采用上面的示例将导致 1+1 条 SQL 的执行。

对于这种情况，我们可以采用一次 Select 两张表的方式，避免这样的性能开销（假设上面示例中，每个 User 只有一个对应的 Address 记录）：

```
<resultMap id="get-user-result" class="user">
    <result property="id" column="id"/>
    <result property="name" column="name"/>
    <result property="sex" column="sex"/>
    <result property="address" column="t_address.address"/>
    <result property="zipCode" column="t_address.zipcode"/>
</resultMap>

<select id="getUsers"
    parameterClass="java.lang.String"
    resultMap="get-user-result">
    <![CDATA[
    select
```

```
        *
    from t_user,t_address
    where t_user.id=t_address.user_id
    ]]>
</select>
```

与此同时，应该保证 User 类中包含 address 和 zipCode 两个 String 型属性。

延迟加载

在运行上面的例子时，通过观察期间的日志输出顺序我们可以发现，在我们执行`sqlMap.queryForList("User.getUsers", "")`时，实际上 ibatis 只向数据库发送了一条 select id, name, sex from t_user SQL。而用于获取 Address 记录的 SQL，只有在我们真正访问 address 对象时，才开始执行。

这也就是所谓的延迟加载（Lazy Loading）机制。即当真正需要数据的时候，才加载数据。延迟加载机制能为我们的系统性能带来极大的提升。

试想，如果我们只需要获取用户名称和性别数据，在没有延迟加载特性的情况下，ibatis 会一次将所有数据都从数据库取回，包括用户信息及其相关的地址数据，而此时，关于地址数据的读取操作没有意义，也就是说，我们白白在地址数据的查询读取上浪费了大量的系统资源。延迟加载为我们妥善的处理了性能与编码上的平衡（如果没有延迟加载，我们为了避免无谓的性能开销，只能专门为此再增加一个不读取地址信息的用户记录检索模块，无疑增加了编码上的工作量）。

回忆之前“ibatis 配置”中的内容：

```
<settings                                                    (1)
    ......
  enhancementEnabled="true"
  lazyLoadingEnabled="true"
  ......
  />
```

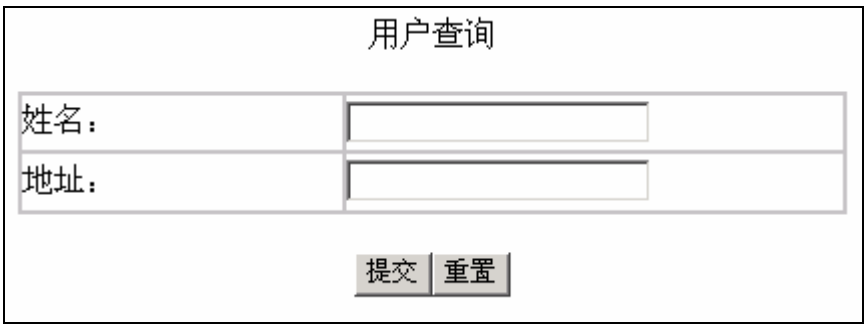
Settings 节点有两个与延迟加载相关的属性 `lazyLoadingEnabled` 和 `enhancementEnabled`，其中 `lazyLoadingEnabled` 设定了系统是否使用延迟加载机制，`enhancementEnabled` 设定是否启用字节码强化机制（通过字节码强化机制可以为 `Lazy Loading` 带来性能方面的改进。

为了使用延迟加载所带来的性能优势，这两项都建议设为`"true"`。

动态映射

在复杂查询过程中，我们常常需要根据用户的选择决定查询条件，这里发生变化的并不只是 SQL 中的参数，包括 Select 语句中所包括的字段和限定条件，都可能发生变化。典型情况，如在一个复杂的组合查询页面，我们必须根据用户的选择和输入决定查询的条件组合。

一个典型的页面如下：



对于这个组合查询页面，根据用户选择填写的内容，我们应为其生成不同的查询语句。

如用户没有填写任何信息即提交查询请求，我们应该返回所有记录：

Select * from t_user;

如用户只在页面上填写了姓名“Erica”，我们应该生成类似：

Select * from t_user where name like ‘%Erica%’；

的 SQL 查询语句。

如用户只在页面上填写了地址“Beijing”，我们应该生成类似：

Select * from t_user where address like ‘%Beijing%”;

的 SQL。

而如果用户同时填写了姓名和地址（”Erica”&’Beijing’），则我们应生成类似：

Select * from t_user where name like ‘%Erica%’ and address like ‘%Beijing%”

的 SQL 查询语句。

对于 ibatis 这样需要预先指定 SQL 语句的 ORM 实现而言，传统的做法无非通过 if-else 语句对输入参数加以判定，然后针对用户选择调用不同的 statement 定义。对于上面这种简单的情况（两种查询条件的排列组合，共 4 种情况）而言，statement 的重复定义工作已经让人不厌其烦，而对于动辄拥有七八个查询条件，乃至十几个查询条件的排列组合而言，琐碎反复的 statement 定义实在让人不堪承受。

考虑到这个问题，ibatis 引入了动态映射机制，即在 statement 定义中，根据不同的查询参数，设定对应的 SQL 语句。

还是以上面的示例为例：

```
<select id="getUsers"
    parameterClass="user"
    resultMap="get-user-result">
```

```
    select
      id,
      name,
      sex
    from t_user

    <dynamic prepend="WHERE">
        <isNotEmpty prepend="AND" property="name">
            (name like #name#)
        </isNotEmpty>

        <isNotEmpty prepend="AND" property="address">
            (address like #address#)
        </isNotEmpty>
    </dynamic>
</select>
```

通过 dynamic 节点，我们定义了一个动态的 WHERE 子句。此 WHERE 子句中将可能包含两个针对 name 和 address 字段的判断条件。而这两个字段是否加入检索取决于用户所提供的查询条件（字段是否为空[isNotEmpty]）。

对于一个典型的 Web 程序而言，我们通过 HttpServletRequest 获得表单中的字段名并将其设入查询参数，如：

```
user.setName(request.getParameter("name"));
user.setAddress(request.getParameter("address"));
sqlMap.queryForList("User.getUsers", user);
```

在执行 queryForList("User.getUsers", user)时，ibatis 即根据配置文件中设定的 SQL 动态生成规则，创建相应的 SQL 语句。

上面的示例中，我们通过判定节点 isNotEmpty，指定了关于 name 和 address 属性的动态规则：

```
<isNotEmpty prepend="AND" property="name">
    (name like #name#)
</isNotEmpty>
```

这个节点对应的语义是，如果参数类的"name"属性非空（isNotEmpty，即非空字符串""），则在生成的 SQL Where 字句中包括判定条件(name like #name#)，其中#name#将以参数类的 name 属性值填充。

Address 属性的判定生成与 name 属性完全相同，这里就不再赘述。

这样，我们通过在 statement 定义中引入 dynamic 节点，很简单的实现了 SQL 判定子句的动态生成，对于复杂的组合查询而言，这将带来极大的便利。

判定节点的定义可以非常灵活，我们甚至可以使用嵌套的判定节点来实现复杂的动态映射，如：

```
<isNotEmpty prepend="AND" property="name">
    (   name=#name#
    <isNotEmpty prepend="AND" property="address">
        address=#address#
    </isNotEmpty>
    )
</isNotEmpty>
```

这段定义规定，只有用户提供了姓名信息时，才能结合地址数据进行查询（如果只提供地址数据，而将姓名信息忽略，将依然被视为全检索）。

Dynamic 节点和判定节点中的 prepend 属性，指明了本节点中定义的 SQL 子句在主体 SQL 中出现时的前缀。

如：

```
<dynamic prepend="WHERE">
    <isNotEmpty prepend="AND" property="name">
        (name like #name#)
    </isNotEmpty>

    <isNotEmpty prepend="AND" property="address">
        (address like #address#)
    </isNotEmpty>
</dynamic>
```

假设"name"属性的值为“Erica”，"address"属性的值为“Beijing”，则会生成类似下面的 SQL 子句（实际运行期将生成带占位符的 PreparedStatement，之后再为其填充数据）：

```
WHERE (name like 'Beijing') AND (address like 'Beijing')
```

其中 WHERE 之后的语句是在 dynamic 节点中所定义，因此以 dynamic 节点的 prepend 设置（"WHERE"）作为前缀，而其中的"**AND**"，实际上是 address 属性所对应的 isNotEmpty 节点的 prepend 设定，它引领了对应节点中定义的 SQL 子句。至于 name 属性对应的 isNotEmpty 节点，由于 ibatis 会自动判定是否需要追加 prepend 前缀，这里(name like #name#)是 WHERE 子句中的第一个条件子句，无需 AND 前缀，所以自动省略。

判定节点并非仅限于 isNotEmpty，ibatis 中提供了丰富的判定定义功能。

判定节点分两类：

Ø 一元判定

一元判定是针对属性值本身的判定，如属性是否为 NULL，是否为空值等。

上面示例中 isNotEmpty 就是典型的一元判定。

一元判定节点有：

节点名	描述
<isPropertyAvailable>	参数类中是否提供了此属性
<isNotPropertyAvailable>	与<isPropertyAvailable>相反
<isNull>	属性值是否为 NULL
<isNotNull>	与<isNull>相反
<isEmpty>	如果属性为 Collection 或者 String，其 size 是否<1，如果非以上两种类型，则通过 String.valueOf(属性值) 获得其 String 类型的值后，判断其 size 是否<1
<isNotEmpty>	与<isEmpty>相反。

Ø 二元判定

二元判定有两个判定参数，一是属性名，而是判定值，如

```
<isGreaterThan prepend="AND" property="age"
        compareValue="18">
    (age=#age#)
</isGreaterThan>
```

其中，property="age"指定了属性名"age"，compareValue="18"指明了判定值为"18"。

上面判定节点 isGreaterThan 对应的语义是：如果 age 属性大于 18(compareValue)，则在 SQL 中加入(age=#age#)条件。

二元判定节点有：

节点名	属性值与 compareValues 的关系
<isEqual>	相等。
<isNotEqual>	不等。
<isGreaterThan>	大于
<isGreaterEqual>	大于等于
<isLessThan>	小于
<isLessEqual>	小于等于

事务管理

基于 JDBC 的事务管理机制

ibatis 提供了自动化的 JDBC 事务管理机制。

对于传统 JDBC Connection 而言，我们获取 Connection 实例之后，需要调用 `Connection.setAutoCommit` 设定事务提交模式。

在 AutoCommit 为 true 的情况下，JDBC 会对我们的操作进行自动提交，此时，每个 JDBC 操作都是一个独立的任务。

为了实现整体事务的原子性，我们需要将 AutoCommit 设为 false，并结合 Connection.commit/rollback 方法进行事务的提交/回滚操作。

ibatis 的所谓“自动化的事务提交机制”，即 ibatis 会根据当前的调用环境，自动判断操作是否需要自动提交。

如果代码没有显式的调用 `SqlMapClient.startTransaction()`方法，则 ibatis 会将当前的数据库操作视为自动提交模式（AutoCommit=true），如：

```
sqlMap = xmlBuilder.buildSqlMap(reader);
User user = new User();
user.setId(new Integer(1));
user.setName("Erica");
user.setSex(new Integer(0));
sqlMap.update("User.updateUser", user);

User user2 = new User();
user2.setId(new Integer(2));
user2.setName("Kevin");
user2.setSex(new Integer(1));
sqlMap.update("User.updateUser", user2);
```

在执行 sqlMap.update 的时候，ibatis 会自动判定当前的运行环境，这里 update 操作并没有相对应的事务范围（`startTransaction` 和 `endTransaction` 代码块），于是 ibatis 将其作为一个单独的事务，并自动提交。对于上面的代码，update 执行了两次，与其相对应，事务也提交了两次（即每个 update 操作为一个单独的事务）。

不过，值得注意的是，这里的所谓“自动判定”，可能有些误导，ibatis 并没有去检查当前是否已经有事务开启，从而判断当前数据库连接是否设定为自动提交。

实际上，在执行 update 语句时，sqlMap 会检查当前的 Session 是否已经关联了某个数据库连接，如果没有，则取一个数据库连接，将其 AutoCommit 属性设为 true，然后执行 update 操作，执行完之后又将这个连接释放。这样，上面两次 update 操作实际上先后获取了**两个**数据库连接，而不是我们通常所认为的两次 update 操作都基于同一个 JDBC Connection。这点在开发时需特别注意。

对于多条 SQL 组合而成的一个 JDBC 事务操作而言，必须使用 `startTransaction`、`commit` 和 `endTransaction` 操作以实现整体事务的原子性。

如：

```
try{
  sqlMap = xmlBuilder.buildSqlMap(reader);
  sqlMap.startTransaction();

  User user = new User();
  user.setId(new Integer(1));
  user.setName("Erica");
  user.setSex(new Integer(0));
  sqlMap.update("User.updateUser", user);

  User user2 = new User();
  user2.setId(new Integer(2));
  user2.setName("Kevin");
  user2.setSex(new Integer(1));
  sqlMap.update("User.updateUser", user2);

  sqlMap.commitTransaction();
}finally{
     sqlMap.endTransaction();
}
```

如果 user1 或者 user2 的 update 操作失败，整个事务就会在 `endTransaction` 时回滚，从而保证了两次 update 操作的原子性。

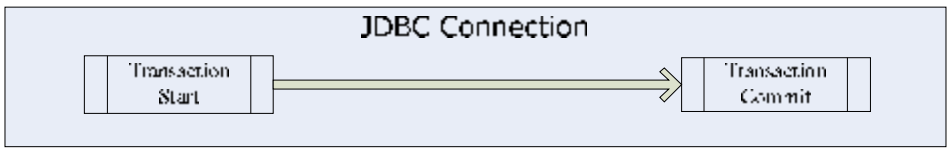
基于 JTA 的事务管理机制

JTA 提供了跨数据库连接（或其他 JTA 资源）的事务管理能力。这一点是与 JDBC Transaction 最大的差异。

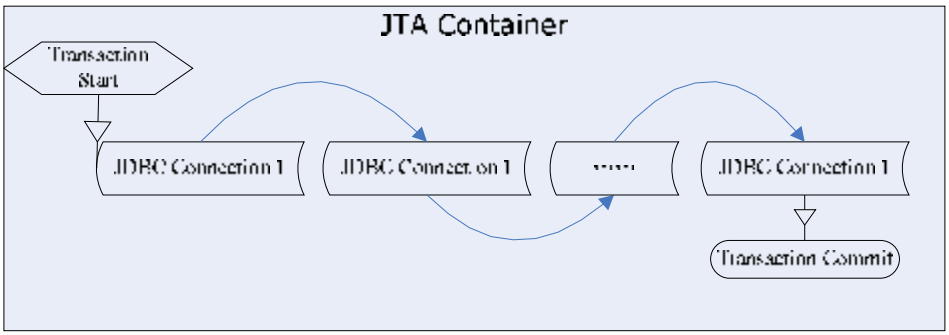
JDBC 事务由 Connnection 管理，也就是说，事务管理实际上是在 JDBC Connection 中实现。事务周期限于 Connection 的生命周期。同样，对于基于 JDBC 的 ibatis 事务管理机制而言，事务管理在 SqlMapClient 所依托的 JDBC Connection 中实现，事务周期限于 SqlMapClient 的生命周期。

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 事务横跨可横跨多个 SqlMapClient。

下面这幅图形象的说明了这个问题：



JDBC Transaction



JTA Transaction

为了在 ibatis 中使用 JTA 事务管理，我们需要在配置文件中加以设定：

```
<transactionManager type="JTA">
    ......
</transactionManager>
```

在实际开发中，我们可能需要面对分布式事务的处理，如系统范围内包含了多个数据库，也许还引入了 JMS 上的事务管理（这在 EAI 系统实现中非常常见）。我们就需要引入 JTA 以实现系统范围内的全局事务，如下面示例中，我们同时将 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。

外部事务管理

基于 JTA 的事务管理还有另外一个特殊情况，就是利用外部事务管理机制。

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">
    ……
</transactionManager>
```

下面是一个外部事务管理的典型示例：

```
UserTransaction tx = new InitialContext().lookup("……");
……
sqlMap1 = xmlBuilder.buildSqlMap(db1Config);
sqlMap2 = xmlBuilder.buildSqlMap(db2Config);
sqlMap1.update("User.updateUser", user);
sqlMap2.update("User.updateUser", user);
……
tx.commit();
```

此时，我们借助 JTA UserTransaction 实例启动了一个全局事务。之后的 ibatis 操作（`sqlMap1.update` 和 `sqlMap2.update`）全部被包含在此全局事务之中，当 UserTransaction 提交的时候，ibatis 操作被包含在事务中提交，反之，如果 UserTransaction 回滚，那么其间的 ibatis 操作也将作为事务的一部分被回滚。这样，我们就实现了 ibatis 外部的事务控制。

另一种外部事务管理方式是借助 EJB 容器，通过 EJB 的部署配置，我们可以指定 EJB 方法的事务性

下面是一个 Session Bean 的 doUpdate 方法，它的事务属性被申明为“Required”，EJB 容器将自动维护此方法执行过程中的事务：

```
/**
 * @ejb.interface-method
 *    view-type="remote"
 *
 * @ejb.transaction type = "Required"
**/
public void doUpdate(){
  //EJB环境中，通过部署配置即可实现事务申明，而无需显式调用事务
  ……
  sqlMap1 = xmlBuilder.buildSqlMap(db1Config);
  sqlMap2 = xmlBuilder.buildSqlMap(db2Config);
  sqlMap1.update("User.updateUser", user);
  sqlMap2.update("User.updateUser", user);
  ……
}//方法结束时，如果没有异常发生，则事务由EJB容器自动提交。
```

上面的示例中，`ibatis` 数据操作的事务管理将全部委托给 `EJB` 容器管理，由 `EJB` 容器控制其事务调度。

在上面 JTA 事务管理的例子中，为了保持清晰，我们省略了 startTransaction、commitTransaction 和 endTransaction 的编写，在这种情况下，调用 ibatis 的事务管理方法并非必须，不过在实际开发时，请酌情添加 startTransaction、commitTransaction 和 endTransaction 语句，这样可以获得更好的性能（如果省略了 startTransaction、commitTransaction 和 endTransaction 语句，ibatis 将为每个数据操作获取一个数据连接，就算引入了数据库连接池机制，这样的无谓开销也应尽量避免，具体请参见 JDBC 事务管理中的描述），并保持代码风格的统一。

Cache

在特定硬件基础上（同时假设系统不存在设计上的缺漏和糟糕低效的 SQL 语句）Cache 往往是提升系统性能的最关键因素）。

相对 Hibernate 等封装较为严密的 ORM 实现而言（因为对数据对象的操作实现了较为严密的封装，可以保证其作用范围内的缓存同步，而 ibatis 提供的是半封闭的封装实现，因此对缓存的操作难以做到完全的自动化同步）。

ibatis 的缓存机制使用必须**特别谨慎**。特别是 flushOnExecute 的设定（见“ibatis 配置”一节中的相关内容），需要考虑到所有可能引起实际数据与缓存数据不符的操作。如本模块中其他 Statement 对数据的更新，其他模块对数据的更新，甚至第三方系统对数据的更新。否则，脏数据的出现将为系统的正常运行造成极大隐患。如果不能完全确定数据更新操作的波及范围，建议避免 Cache 的盲目使用。

结合 cacheModel 来看：

```
<cacheModel
  id="product-cache"
  type ="LRU"
  readOnly="true"
  serialize="false">
</cacheModel>
```

可以看到，Cache 有如下几个比较重要的属性：

- readOnly
- serialize
- type

readOnly

readOnly 值的是缓存中的**数据对象**是否只读。这里的只读并不是意味着数据对象一旦放入缓存中就无法再对**数据**进行修改。而是当**数据对象**发生变化的时候，如数据对象的某个属性发生了变化，则此数据对象就将被从缓存中废除，下次需要重新从数据库读取数据，构造新的数据对象。

而 readOnly="false"则意味着缓存中的数据对象可更新，如 user 对象的 name 属性发生改变。

只读 Cache 能提供更高的读取性能，但一旦数据发生改变，则效率降低。系统设计时需根据系统的实际情况（数据发生更新的概率有多大）来决定 Cache 的读写策略。

serialize

如果需要全局的数据缓存，CacheModel 的 serialize 属性必须被设为 true。否则数据缓存只对当前 Session（可简单理解为当前线程）有效，局部缓存对系统的整体性能提升有限。

在 `serialize="true"`的情况下，如果有多个 Session 同时从 Cache 中读取某个数据对象，Cache 将为每个 Session 返回一个对象的复本，也就是说，每个 Session 将得到包含相同信息的不同对象实例。因而 Session 可以对其从 Cache 获得的数据进行存取而无需担心多线程并发情况下的同步冲突。

Cache Type:

与 hibernate 类似，ibatis 通过缓冲接口的插件式实现，提供了多种 Cache 的实现机制可供选择：

1. MEMORY
2. LRU
3. FIFO
4. OSCACHE

MEMORY 类型 Cache 与 WeakReference

MEMORY 类型的 Cache 实现，实际上是通过 Java 对象引用进行。ibatis 中，其实现类为 com.ibatis.db.sqlmap.cache.memory.MemoryCacheController，MemoryCacheController 内部，使用一个 HashMap 来保存当前需要缓存的数据对象的引用。

这里需要注意的是 Java2 中的三种对象引用关系：

a SoftReference
b WeakReference
c PhantomReference

传统的 Java 对象引用，如：

```
public void doSomeThing(){
    User user = new User()
    ……
}
```

当 doSomeThing 方法结束时，user 对象的引用丢失，其所占的内存空间将由 JVM 在下次垃圾回收时收回。如果我们将 user 对象的引用保存在一个全局的 HashMap 中，如：

```
Map map = new HashMap();

public void doSomeThing(){
    User user = new User();
    map.put("user",user);
}
```

此时，user 对象由于在 map 中保存了引用，只要这个引用存在，那么 JVM 永远也不会收回 user 对象所占用的内存。

这样的内存管理机制相信诸位都已经耳熟能详，在绝大多数情况下，这几乎是一种完美

的解决方案。但在某些情况下，却有些不便。如对于这里的 Cache 而言，当 user 对象使用之后，我们希望保留其引用以供下次需要的时候可以重复使用，但又不希望这个引用长期保存，如果每个对象的引用都长期保存下去的话，那随着时间推移，有限的内存空间将立即被这些数据所消耗殆尽。最好的方式，莫过于有一种引用方式，可以在对象没有被垃圾回收器回收之前，依然能够访问此对象，当垃圾回收器启动时，如果此对象没有被其他对象所使用，则按照常规对其进行回收。

SoftReference、WeakReference、PhantomReference 为上面的思路提供了有力支持。

这三种类型的引用都属于“非持续性引用”，也就是说，这种引用关系并非持续存在，它们所代表的引用的生命周期与 JVM 的运行密切相关，而非与传统意义上的引用一样依赖于编码阶段的预先规划。

先看一个 SoftReference 的例子：

```
SoftReference ref;

public void doSomeThing(){
 User user = new User();
 ref = new SoftReference(user);
}

public void doAnotherThing(){
 User user = (User)ref.get();//通过SoftReference获得对象引用
 System.out.println(user.getName());
}
```

假设我们先执行了 doSomeThing 方法，产生了一个 User 对象，并为其创建了一个 SoftReference 引用。

之后的某个时刻，我们调用了 doAnotherThing 方法，并通过 SoftReference 获取 User 对象的引用。

此时我们是否还能取得 user 对象的引用？这要看 JVM 的运行情况。对于 SoftReference 而言，只有当目前内存不足的情况下，JVM 在垃圾回收时才会收回其包含的引用（JVM 并不是只有当内存不足时才启动垃圾回收机制，何时进行垃圾回收取决于各版本 JVM 的垃圾回收策略。如某这垃圾回收策略为：当系统目前较为空闲，且无效对象达到一定比率时启动垃圾回收机制，此时的空余内存倒可能还比较充裕）。这里可能出现两种情况，即：

- JVM 目前还未出现过因内存不足所引起的垃圾回收，user 对象的引用可以通过 SoftReference 从 JVM Heap 中收回。
- JVM 已经因为内存不足启动了垃圾回收机制，SoftReference 所包含的 user 对象的引用被 JVM 所废弃。此时 `ref.get` 方法将返回一个空引用（`null`），对于上面的代码而言，也就意味着这里可能抛出一个 `NullPointerException`。

WeakReference 比 SoftReference 在引用的维持性上来看更加微弱。无需等到内存不足的情况，只要 JVM 启动了垃圾回收机制，那么 WeakReference 所对应的对象就将被 JVM 回收。

也就是说，相对 SoftReference 而言，WeakReference 被 JVM 回收的概率更大。

PhantomReference 比 WeakReference 的引用维持性更弱。与 WeakReference 和 SoftReference 不同，PhantomReference 所引用的对象几乎无法被回收重用。它指向的对象实际上已经被 JVM 销毁（finalize 方法已经被执行），只是暂时还没被垃圾回收器收回而已。PhantomReference 主要用于辅助对象的销毁过程，在实际应用层研发中，几乎不会涉及。

MEMORY 类型的 Cache 正是借助 SoftReference、WeakReference 以及通常意义上的 Java Reference 实现了对象的缓存管理。

下面是一个典型的 MEMORY 类型 Cache 配置：

```
<cacheModel id="user_cache" type="MEMORY">
    <flushInterval hours="24"/>
    <flushOnExecute statement="updateUser"/>
    <property name="reference-type" value="WEAK" />
</cacheModel>
```

其中 flushInterval 指定了多长时间清除缓存，上例中指定每 24 小时强行清空缓存区的所有内容。

reference-type 属性可以有以下几种配置：

1. STRONG

   即基于传统的 Java 对象引用机制，除非对 Cache 显式清空（如到了 flushInterval 设定的时间；执行了 flushOnExecute 所指定的方法；或代码中对 Cache 执行了清除操作等），否则引用将被持续保留。

   此类型的设定适用于缓存常用的数据对象，或者当前系统内存非常充裕的情况。

2. SOFT

   基于 SoftReference 的缓存实现，只有 JVM 内存不足的时候，才会对缓冲池中的数据对象进行回收。

   此类型的设定适用于系统内存较为充裕，且系统并发量比较稳定的情况。

3. WEAK

   基于 WeakReference 的缓存实现，当 JVM 垃圾回收时，缓存中的数据对象将被 JVM 收回。

   一般情况下，可以采用 WEAK 的 MEMORY 型 Cache 配置。

LRU 型 Cache

当 Cache 达到预先设定的最大容量时，ibatis 会按照“最少使用”原则将使用频率最少的对象从缓冲中清除。

```
<cacheModel id="userCache" type="LRU">
    <flushInterval hours="24"/>
    <flushOnExecute statement="updateUser"/>
    <property name="size" value="1000" />
</cacheModel>
```

可配置的参数有：

- flushInterval

  指定了多长时间清除缓存，上例中指定每 24 小时强行清空缓存区的所有内容。

- size

Cache 的最大容量。

FIFO 型 Cache

先进先出型缓存，最先放入 Cache 中的数据将被最先废除。可配置参数与 LRU 型相同：

```
<cacheModel id="userCache" type="FIFO">
    <flushInterval hours="24"/>
    <flushOnExecute statement="updateUser"/>
    <property name="size" value="1000" />
</cacheModel>
```

OSCache

与上面几种类型的 Cache 不同，OSCache 来自第三方组织 Opensymphony。可以通过以下网址获得 OSCache 的最新版本（http://www.opensymphony.com/oscache/）。

在生产部署时，建议采用 OSCache，OSCache 是得到了广泛使用的开源 Cache 实现（Hibernate 中也提供了对 OSCache 的支持），它基于更加可靠高效的设计，更重要的是，最新版本的 OSCache 已经支持 Cache 集群。如果系统需要部署在集群中，或者需要部署在多机负载均衡模式的环境中以获得性能上的优势，那么 OSCache 在这里则是不二之选。

Ibatis 中对于 OSCache 的配置相当简单：

```
<cacheModel id="userCache" type="OSCACHE">
    <flushInterval hours="24"/>
    <flushOnExecute statement="updateUser"/>
    <property name="size" value="1000" />
</cacheModel>
```

之所以配置简单，原因在于，OSCache 拥有自己的配置文件（oscache.properties）**J**。下面是一个典型的 OSCache 配置文件：

```
#是否使用内存作为缓存空间
cache.memory=true

#缓存管理事件监听器，通过这个监听器可以获知当前 Cache 的运行情况
cache.event.listeners=com.opensymphony.oscache.plugins.clustersupport.JMSBroadcastingListener

#如果使用磁盘缓存（cache.memory=false），则需要指定磁盘存储接口实现
#cache.persistence.class=com.opensymphony.oscache.plugins.diskpersistence.DiskPersistenceListener

# 磁盘缓存所使用的文件存储路径
# cache.path=c:\\myapp\\cache

# 缓存调度算法，可选的有 LRU,FIFO 和无限缓存（UnlimitedCache）
```

```
# cache.algorithm=com.opensymphony.oscache.base.algorithm.FIFOCache
# cache.algorithm=com.opensymphony.oscache.base.algorithm.UnlimitedCache
cache.algorithm=com.opensymphony.oscache.base.algorithm.LRUCache

#内存缓存的最大容量
cache.capacity=1000

# 是否限制磁盘缓存的容量
# cache.unlimited.disk=false

# 基于 JMS 的集群缓存同步配置
#cache.cluster.jms.topic.factory=java:comp/env/jms/TopicConnectionFactory
#cache.cluster.jms.topic.name=java:comp/env/jms/OSCacheTopic
#cache.cluster.jms.node.name=node1

# 基于 JAVAGROUP 的集群缓存同步配置
#cache.cluster.properties=UDP(mcast_addr=231.12.21.132;mcast_port=45566;ip_
ttl=32;mcast_send_buf_size=150000;mcast_recv_buf_size=80000):PING(timeout
=2000;num_initial_members=3):MERGE2(min_interval=5000;max_interval=10000
):FD_SOCK:VERIFY_SUSPECT(timeout=1500):pbcast.NAKACK(gc_lag=50;retransm
it_timeout=300,600,1200,2400,4800):pbcast.STABLE(desired_avg_gossip=20000):
UNICAST(timeout=5000):FRAG(frag_size=8096;down_thread=false;up_thread=fal
se):pbcast.GMS(join_timeout=5000;join_retry_timeout=2000;shun=false;print_loc
al_addr=true)
#cache.cluster.multicast.ip=231.12.21.132
```

配置好之后，将此文件放在 **CLASSPATH** 中，**OSCache** 在初始化时会自动找到此文件并根据其中的配置创建缓存实例。

ibatis in Spring

这里我们重点探讨**Spring**框架下的**ibatis**应用，特别是在容器事务管理模式下的**ibatis**应用开发。

关于**Spring Framework**，请参见笔者另一篇文档：
《**Spring** 开发指南》**http://www.xiaxin.net/Spring_Dev_Guide.rar**

针对**ibatis**，**Spring**配置文件如下：

Ibatis-Context.xml:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
"http://www.springframework.org/dtd/spring-beans.dtd">

<beans>

  <bean id="dataSource"
      class="org.apache.commons.dbcp.BasicDataSource"
      destroy-method="close">
      <property name="driverClassName">
          <value>net.sourceforge.jtds.jdbc.Driver</value>
      </property>
      <property name="url">
          <value>jdbc:jtds:sqlserver://127.0.0.1:1433/Sample</value>
      </property>
      <property name="username">
          <value>test</value>
      </property>
      <property name="password">
          <value>changeit</value>
      </property>
  </bean>

  <bean id="sqlMapClient"
  class="org.springframework.orm.ibatis.SqlMapClientFactoryBean">
      <property name="configLocation">
          <value>SqlMapConfig.xml</value>
      </property>
  </bean>

    <bean id="transactionManager"
    class="org.springframework.jdbc.datasource.DataSourceTransactio
nManager">
```

```
        <property name="dataSource">
            <ref local="dataSource"/>
        </property>
    </bean>

  <bean id="userDAO" class="net.xiaxin.dao.UserDAO">
      <property name="dataSource">
          <ref local="dataSource" />
      </property>
      <property name="sqlMapClient">
          <ref local="sqlMapClient" />
      </property>
  </bean>

  <bean id="userDAOProxy"
  class="org.springframework.transaction.interceptor.TransactionPro
xyFactoryBean">

      <property name="transactionManager">
          <ref bean="transactionManager" />
      </property>

      <property name="target">
          <ref local="userDAO" />
      </property>

      <property name="transactionAttributes">
          <props>
              <prop key="insert*">PROPAGATION_REQUIRED</prop>
              <prop key="get*">PROPAGATION_REQUIRED,readOnly</prop>
          </props>
      </property>
  </bean>
</beans>
```

可以看到：

1. **sqlMapClient**节点
   sqlMapClient节点实际上配置了一个**sqlMapClient**的创建工厂类。
   configLocation属性配置了**ibatis**映射文件的名称。

2. **transactionManager**节点
   transactionManager采用了**Spring**中的**DataSourceTransactionManager**。

3. **userDAO**节点

对应的，**UserDAO**需要配置两个属性，**sqlMapClient**和**DataSource**，**sqlMapClient**将从指定的**DataSource**中获取数据库连接。

本例中**Ibatis**映射文件非常简单：

sqlMapConfig.xml:

```
<sqlMapConfig>
 <sqlMap resource="net/xiaxin/dao/entity/user.xml"/>
</sqlMapConfig>
```

net/xiaxin/dao/entity/user.xml

```
<sqlMap namespace="User">
    <typeAlias alias="user" type="net.xiaxin.dao.entity.User" />

    <insert id="insertUser" parameterClass="user">
        INSERT INTO users ( username, password) VALUES ( #username#,
#password# )
    </insert>

</sqlMap>
```

UserDAO.java:

```
public class UserDAO extends SqlMapClientDaoSupport implements
IUserDAO {

    public void insertUser(User user) {
        getSqlMapClientTemplate().update("insertUser", user);
    }
}
```

SqlMapClientDaoSupport（如果使用**ibatis 1.x**版本，对应支持类是**SqlMapDaoSupport**）是**Spring**中面向**ibatis**的辅助类，它负责调度**DataSource**、**SqlMapClientTemplate**（对应**ibatis 1.x**版本是**SqlMapTemplate**）完成**ibatis**操作，而**DAO**则通过对此类进行扩展获得上述功能。上面配置文件中针对**UserDAO**的属性设置部分，其中的属性也是继承自于这个基类。

SqlMapClientTemplate对传统**SqlMapClient**调用模式进行了封装，简化了上层访问代码。

User.java:

```
public class User {

    public Integer id;

    public String username;
```

```
    public String password;

    public Integer getId() {
        return id;
    }

    public void setId(Integer id) {
        this.id = id;
    }
    public String getPassword() {
        return password;
    }

    public void setPassword(String password) {
        this.password = password;
    }

    public String getUsername() {
        return username;
    }

    public void setUsername(String username) {
        this.username = username;
    }
}
```

测试代码：

```
InputStream is = new FileInputStream("Ibatis-Context.xml");
XmlBeanFactory factory = new XmlBeanFactory(is);
IUserDAO userdao = (IUserDAO)factory.getBean("userDAOProxy");

User user = new User();
user.setUsername("Sofia");
user.setPassword("mypass");

userdao.insertUser(user);
```

对比前面 **ibatis** 程序代码，我们可以发现 **UserDAO.java** 变得异常简洁，这也正是 **Spring Framework** 为我们带来的巨大帮助。

Spring 以及其他 **IOC** 框架对传统的应用框架进行了颠覆性的革新，也许这样的评价有点过于煽情，但是这确是笔者第一次跑通 **Spring HelloWorld** 时的切身感受。有兴趣的读者可以研究一下 **Spring framework**，**enjoy it**！