Hibernate的缺陷

Hibernate作为全表映射框架，举个例子来说，如果我们有张财务表（按年份分表），比如2015年表名为bill2015，到了2016年，表命名为bill2016，要动态加映射关系，Hibernate需要破坏底层封装才能做到。又比如说，一些财务信息往往需要和某些对象关联起来，不同的对象有不同的列，因此列名也是无法确定的，显然我们没有办法配置XML去完成映射规则。再者如果使用存储过程，Hibernate也是无法适应的。这些都不是致命的，最为致命的问题是性能。Hibernate屏蔽了SQL，那就意味着只能全表映射，但是一张表可能有几十到上百个字段，而你感兴趣的只有2个，这是Hibernate无法适应的。尤其是在大型网站系统，对传输数据有严格规定，不能浪费带宽的场景下就更为明显了。有很复杂的场景需要关联多张表，Hibernate全表逐级去对象的方法也只能作罢，写SQL还需要手工的映射去数据，这带来了很大的麻烦。此外，如果我们需要优化SQL，Hibernate也是无法做到的。

总结一下Hibernate的缺点：

* 全表映射带来的不便，比如更新时需要发送所有的字段。
* 取法根据不用的条件组装不同的SQL。
* 对夺标关联和复杂SQL查询支持较差，需要自己写SQL，返回后，需要自己将数据组装为POJO。
* 不能有效支持存储过程。
* 虽然有HQL，但是性能较差。大型互联网系统往往需要优化SQL，而Hibernate做不到。

Mybatis

为了解决Hibernate的不足，一个半自动映射的框架mybatis应运而生。之所以称它为半自动，是因为它需要手工匹配提供POJO、SQL和映射关系，而全表映射的Hibernate只需要提供POJO和映射关系便可。

在Mybatis里面，你需要自己编写SQL，虽然比Hibernate配置得多，但是MyBatis可以配置动态SQL，这就解决了Hibernate的表名根据时间变化，不同的条件下列名不一样的问题。同时你也可以优化SQL，通过配置决定你的SQL映射规则，也能支持存储过程，所以对于一些复杂的和需要优化性能SQL的查询它更加方便，Mybatis几乎能做到JDBC所能做到的所有事情。MyBatis具有自动映射功能。换句话说，在注意一些规则的基础上，MyBatis可以给我们完成自动映射，而无需再写任何的映射规则，这大大提高了开发效率和灵活性。

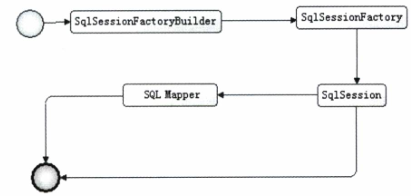
第2章

Mybatis的基本构成

Mybatis由一下几个核心组件构成：

* SqlSessionFactoryBuilder（构造器）：它会根据配置信息或者代码来生成SqlSessionFactory。
* SqlSessionFactory（工厂接口）：依靠工厂来生成SqlSession。
* SqlSession（会话）：是一个既可以发送SQL去执行并返回结果，也可以获取Mapper的接口。
* SQL Mapper：它是Mybatis新设计的组件，它是由一个Java接口和XML文件（或注解）构造成的。需要给出对应的SQL和映射规则，它负责发送SQL去执行，并返回结果。

用一个图表达它们之间的关系：



# 配置

以下是Mybatis配置XML文件的层次结构：

*<?***xml version="1.0" encoding="UTF-8"** *?>*<**configuration**>  
 <**properties**/>*<!--属性-->* <**settings**/>*<!--设置-->* <**typeAliases**/>*<!--类型命名-->* <**typeHandlers**/>*<!--类型处理器-->* <**objectFactory**/>*<!--对象工厂-->* <**plugins**/>*<!--插件-->* <**environments**>*<!--配置环境-->* <**environment**>  
 <**transactionManager**/>*<!--事务管理器-->* <**dataSource**/>*<!--数据源-->* </**environment**>  
 </**environments**>  
 <**databaseIdProvider**/>*<!--数据库厂商标识-->* <**mappers**/>*<!--映射器-->*</**configuration**>

注意，这些层次是不能颠倒顺序的，如果颠倒顺序，mybatis在解析XML文件的时候回出现异常。

## <properties/>

properties是一个配置属性的元素，能让我们在配置文件的上下中使用配置的属性，Mybatis提供了3种配置方式：

* property子元素
* .properties文件配置
* 程序参数传递

### 子元素配置

使用子元素配置的方式如下：

<**properties**>  
 <**property name="driver" value="com.mysql.jdbc.Driver"**/>  
 <**property name="url" value="jdbc:mysql://localhost:3306/test?useUnicode=true&amp;characterEncoding=utf-8"**/>  
 <**property name="username" value="root"**/>  
 <**property name="password" value="123456"**/>  
</**properties**>

这样我们就可以使用${}占位符的形式使用这些属性，如：

<**dataSource type="POOLED"**>  
 <**property name="driver" value="${driver}"** />  
 <**property name="url" value="${url}"** />  
 <**property name="username" value="${username}"** />  
 <**property name="password" value="${password}"** />  
</**dataSource**>

### .properties文件配置

我们也可以使用使用properties配置文件来配置属性值，以方便在多个配置文件中重复使用它们，也方便日后维护修改，我们将上面的数据源信息配置在.properties文件中，如：

*#数据库配置信息***driver**=**com.mysql.jdbc.Driver  
url**=**jdbc:mysql://localhost:3306/test?useUnicode=true&characterEncoding=utf-8  
username**=**root  
password**=**123456**

### 程序参数传递

在实际工作中，系统是有运维人员去配置的，生成数据库的用户密码对于开发者而言是保密的，而且为了安全，运维人员要求对配置文件中的数据库用户和密码进行加密，这样我们的配置文件中往往配置的加密过后的数据库信息，而无法通过加密字符串去连接数据库，这个时候可以通过编码的形式来满足我们遇到的场景。

下面假设jdbc.properties文件中的username和password两个属性使用了加密的字符串，这个时候我们需要在生成SQLSessionFactory之前将它转化为明文，而系统已经提供了解密的方法decode(Str），以下是使用代码的方式来完成SQLSessionFactory的创建：

InputStream cfgStream = **null**;  
Reader cfgReader = **null**;  
InputStream proStream = **null**;  
Reader proReader = **null**;  
Properties properties = **null**;  
**try**{  
 *// 读取配置文件* cfgSteam = Resources.getReasourceAsStream(**"mybatis-config.xml"**);  
 cfgReader = **new** InputStreamReader(cfgStream);  
 *// 读入属性文件* proStream = **new** inputStreamReader(proStream);  
 properties = **new** Properties();  
 proerties.load(proRdader);  
 *// 解密为明文* properties.setProperty(**"username"**,decode(properties.getProperty(**"username"**)));  
 properties.setProperty(**"password"**,decode(properties.getProperty(**"password"**)));  
}**catch**(IOException ex){}  
  
*// 创建SqlSessionFactory*sqlSessionFactory = **new** SqlSessionFactoryBuilder().build(cfg,Reader, properties);

### 三种配置方式的优先级

如果属性在不只一个地方进行了配置，那么 MyBatis 将按照下面的顺序来加载：

* 在 properties 元素体内指定的属性首先被读取。
* 然后根据 properties 元素中的 resource 属性读取类路径下属性文件或根据 url 属性指定的路径读取属性文件，并覆盖已读取的同名属性。
* 最后读取作为方法参数传递的属性，并覆盖已读取的同名属性。

因此，**通过方法参数传递的属性具有最高优先级，resource/url 属性中指定的配置文件次之，最低优先级的是 properties 属性中指定的属性。实际操作中我们推荐使用properties文件的方式，尽量避免使用混合的方式。**

## <settings/>

这是 MyBatis 中极为重要的调整设置，它们会改变 MyBatis 的运行时行为。下表描述了设置中各项的意图、默认值等。

| **设置参数** | **描述** | **有效值** | **默认值** |
| --- | --- | --- | --- |
| cacheEnabled | 该配置影响的所有映射器中配置的缓存的全局开关。 | true | false | true |
| lazyLoadingEnabled | 延迟加载的全局开关。当开启时，所有关联对象都会延迟加载。  特定关联关系中可通过设置fetchType属性来覆盖该项的开关  状态。 | true | false | false |
| aggressiveLazyLoading | 当开启时，任何方法的调用都会加载该对象的所有属性。  否则，每个属性会按需加载（参考lazyLoadTriggerMethods). | true | false | false (true in ≤3.4.1) |
| multipleResultSetsEnabled | 是否允许单一语句返回多结果集（需要兼容驱动）。 | true | false | true |
| useColumnLabel | 使用列标签代替列名。不同的驱动在这方面会有不同的表现，  具体可参考相关驱动文档或通过测试这两种不同的模式来观察  所用驱动的结果。 | true | false | true |
| useGeneratedKeys | 允许 JDBC 支持自动生成主键，需要驱动兼容。 如果设置为  true 则这个设置强制使用自动生成主键，尽管一些驱动不能兼  容但仍可正常工作（比如 Derby）。 | true | false | False |
| autoMappingBehavior | 指定 MyBatis 应如何自动映射列到字段或属性。 NONE 表  示取消自动映射；PARTIAL 只会自动映射没有定义嵌套结果  集映射的结果集。 FULL 会自动映射任意复杂的结果集（无论  是否嵌套）。 | NONE, PARTIAL, FULL | PARTIAL |
| autoMapping-  UnknownColumnBehavior | 指定发现自动映射目标未知列（或者未知属性类型）的行为。  NONE: 不做任何反应  WARNING: 输出提醒日志 ('org.apache.ibatis.session.AutoMappingUnknownColumn  Behavior' 的日志等级必须设置为 WARN)  FAILING: 映射失败 (抛出 SqlSessionException) | NONE, WARNING, FAILING | NONE |
| defaultExecutorType | 配置默认的执行器。SIMPLE 就是普通的执行器；REUSE 执行器会重用预处理语句（prepared statements）； BATCH 执行器将重用语句并执行批量更新。 | SIMPLE REUSE BATCH | SIMPLE |
| defaultStatementTimeout | 设置超时时间，它决定驱动等待数据库响应的秒数。 | 任意正整数 | Not Set (null) |
| defaultFetchSize | 为驱动的结果集获取数量（fetchSize）设置一个提示值。此参数只可以在查询设置中被覆盖。 | 任意正整数 | Not Set (null) |
| safeRowBoundsEnabled | 允许在嵌套语句中使用分页（RowBounds）。 If allow, set the false. | true | false | False |
| safeResultHandlerEnabled | 允许在嵌套语句中使用分页（ResultHandler）。 If allow, set the false. | true | false | True |
| mapUnderscoreToCamelCase | 是否开启自动驼峰命名规则（camel case）映射，即从经典数据库列名 A\_COLUMN 到经典 Java 属性名 aColumn 的类似映射。 | true | false | False |
| localCacheScope | MyBatis 利用本地缓存机制（Local Cache）防止循环引用（circular references）和加速重复嵌套查询。 默认值为 SESSION，这种情况下会缓存一个会话中执行的所有查询。 若设置值为 STATEMENT，本地会话仅用在语句执行上，对相同 SqlSession 的不同调用将不会共享数据。 | SESSION | STATEMENT | SESSION |
| jdbcTypeForNull | 当没有为参数提供特定的 JDBC 类型时，为空值指定 JDBC 类型。 某些驱动需要指定列的 JDBC 类型，多数情况直接用一般类型即可，比如 NULL、VARCHAR 或 OTHER。 | JdbcType enumeration. Most common are: NULL, VARCHAR and OTHER | OTHER |
| lazyLoadTriggerMethods | 指定哪个对象的方法触发一次延迟加载。 | A method name list separated by commas | equals,clone,hashCode,toString |
| defaultScriptingLanguage | 指定动态 SQL 生成的默认语言。 | A type alias or fully qualified class name. | org.apache.ibatis.scripting.xmltags.  XMLLanguageDriver |
| callSettersOnNulls | 指定当结果集中值为 null 的时候是否调用映射对象的 setter（map 对象时为 put）方法，这对于有 Map.keySet() 依赖或 null 值初始化的时候是有用的。注意基本类型（int、boolean等）是不能设置成 null 的。 | true | false | false |
| returnInstanceForEmptyRow | 当返回行的所有列都是空时，MyBatis默认返回null。 当开启这个设置时，MyBatis会返回一个空实例。 请注意，它也适用于嵌套的结果集 (i.e. collectioin and association)。（从3.4.2开始） | true | false | false |
| logPrefix | 指定 MyBatis 增加到日志名称的前缀。 | Any String | Not set |
| logImpl | 指定 MyBatis 所用日志的具体实现，未指定时将自动查找。 | SLF4J | LOG4J | LOG4J2 | JDK\_LOGGING | COMMONS\_LOGGING | STDOUT\_LOGGING | NO\_LOGGING | Not set |
| proxyFactory | 指定 Mybatis 创建具有延迟加载能力的对象所用到的代理工具。 | CGLIB | JAVASSIST | JAVASSIST (MyBatis 3.3 or above) |
| vfsImpl | 指定VFS的实现 | 自定义VFS的实现的类全限定名，以逗号分隔。 | Not set |
| useActualParamName | 允许使用方法签名中的名称作为语句参数名称。 为了使用该特性，你的工程必须采用Java 8编译，并且加上-parameters选项。（从3.4.1开始） | true | false | true |
| configurationFactory | Specifies the class that provides an instance of Configuration. The returned Configuration instance is used to load lazy properties of deserialized objects. This class must have a method with a signature static Configuration getConfiguration(). (Since: 3.2.3) | A type alias or fully qualified class name. | Not set |

一个配置完整的 settings 元素的示例如下：

<**settings**>  
 <**setting name="cacheEnabled" value="true"**/>  
 <**setting name="lazyLoadingEnabled" value="true"**/>  
 <**setting name="multipleResultSetsEnabled" value="true"**/>  
 <**setting name="useColumnLabel" value="true"**/>  
 <**setting name="useGeneratedKeys" value="false"**/>  
 <**setting name="autoMappingBehavior" value="PARTIAL"**/>  
 <**setting name="autoMappingUnknownColumnBehavior" value="WARNING"**/>  
 <**setting name="defaultExecutorType" value="SIMPLE"**/>  
 <**setting name="defaultStatementTimeout" value="25"**/>  
 <**setting name="defaultFetchSize" value="100"**/>  
 <**setting name="safeRowBoundsEnabled" value="false"**/>  
 <**setting name="mapUnderscoreToCamelCase" value="false"**/>  
 <**setting name="localCacheScope" value="SESSION"**/>  
 <**setting name="jdbcTypeForNull" value="OTHER"**/>  
 <**setting name="lazyLoadTriggerMethods" value="equals,clone,hashCode,toString"**/>  
</**settings**>

## <typeAliases/>

别名是一个指代的名称。因为有时候我们遇到的类权限定名过长，所以我们希望使用一个简短的名称去指代它，而这个名称可以在mybatis上下文中使用。别名在Mybatis中分为系统别名和自定义别名两类。**注意，别名是不区分大小写的。**

### 系统别名

Mybatis系统定义了一些经常使用的类型的别名，例如，数值、字符串、日期和集合等，我们可以在Mybatis中直接使用它们，在使用时不要重复定义把它们给覆盖了。以下是Mybatis已经定义好的别名，支持数值类型的只要加“[]”即可使用，比如Date数组别名可以使用date[]代替。

|  |  |
| --- | --- |
| **别名** | **映射的类型** |
| \_byte | byte |
| \_long | long |
| \_short | short |
| \_int | int |
| \_integer | int |
| \_double | double |
| \_float | float |
| \_boolean | boolean |
| string | String |
| byte | Byte |
| long | Long |
| short | Short |
| int | Integer |
| integer | Integer |
| double | Double |
| float | Float |
| boolean | Boolean |
| date | Date |
| decimal | BigDecimal |
| bigdecimal | BigDecimal |
| object | Object |
| map | Map |
| hashmap | HashMap |
| list | List |
| arraylist | ArrayList |
| collection | Collection |
| iterator | Iterator |

### 自定义别名

系统定义的别名往往是不够用的，因为不同的应用有着不同的需要，所以Mybatis允许自定义别名。如：

<**typeAliases**>  
 <**typeAlias alias="employeer" type="com.whz.entity.Employeer"** />  
</**typeAliases**>

这样employee可以在任何需要使用com.whz.entity.Employee的地方去提换它，如：

*<!--<select id="findEmployeerByID" parameterType="int" resultType="com.whz.entity.Employeer">-->*<**select id="findEmployeerByID" parameterType="int" resultType="employeer"**>  
 select *\** from `t\_employeer`  
 where employeer\_id = #{employeer\_id}  
</**select**>

也可以使用@Alias注解来定义别名，如：

@Alias(**"employeer"**)  
**public class** Employeer {  
 ...  
}

如果POJO过多的时候，配置也会非常多，Mybatis为解决该问题，支持使用自动扫描包的功能，将扫描到的类装载到上下文中，如：

<**typeAliases**>  
 <**package name="com.whz.entity"**/>  
</**typeAliases**>

注意，配置了包扫描功能后，**没有@Alias的也会装载，Mybatis将把你的类名的第一个字母变为小写来作为别名**，所以需要特别注意避免出现重名的场景，建议使用部分包名加类名的限定。

## <typeHandlers/>

Mybatis在预处理语句（PreparedStatement）中设置一个参数时，或从结果集（ResultSet）中取出一个值时，都会用到typeHandler进行处理。typeHandler的作用就是将参数从JavaType转为jdbcType，或者从数据库取出结果时把jdbcType转为JavaType。

由于数据库可能来自于不同的厂商，不同的厂商设置的参数可能所有不同，同时数据库也可以自定义数据类型，typeHandler允许根据项目的需要自定义设置Java传递到数据库的参数中，或者从数据库读出数据，我们也需要进行特殊的处理，这些都可以在定义的typeHandler中处理，尤其是在使用枚举的时候我们常常需要使用typeHandler进行转换。

typeHandler也分为系统和用于自定义两种，一般来说，使用Mybatis系统定义的typeHandler就可以实现大部分的功能。

### 系统定义的typeHandler



### 用户定义的typeHandler

**需求背景**

在做开发时，我们经常会遇到这样一些问题，比如我有一个[Java](http://lib.csdn.net/base/java" \o "Java 知识库" \t "http://blog.csdn.net/u012702547/article/details/_blank)中的Date数据类型，我想将之存到[数据库](http://lib.csdn.net/base/mysql" \o "MySQL知识库" \t "http://blog.csdn.net/u012702547/article/details/_blank)的时候存成一个1970年至今的毫秒数，怎么实现？

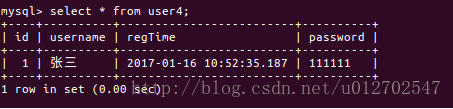
假设我现在创建一张表，如下：

**create table user**(  
 **id integer primary key** auto\_increment,  
 **username varchar**(32),  
 **password varchar**(64),  
 regTime **varchar(64)**)**default character set**=utf8;

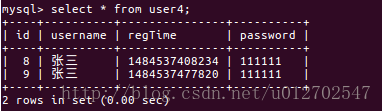
然后我再在Java中定义一个实体类：

**public class** User {  
 **private** Long **id**;  
 **private** String **username**;  
 **private** String **password**;  
 private **Date** regTime;  
 *//省略getter/setter*}

这个JavaBean中也有一个regTime字段，不同的是这里的数据类型是Date。如果我不做任何特殊处理，直接向数据库插入数据，也是可以插入成功的，但是插入成功后是这样：



这个当然不是我想要的，我希望存到数据库里的是这样的：



就是我直接向数据库写数据，要写的是一个Date对象，但是写到数据库之后这个Date对象就变成了Date对象所描述的时间到1970年的秒数了，然后当我从数据库读取这个秒数之后，系统又会自动帮我将这个秒数转为Date对象，就是这样两个需求。

**自定义typeHandler**

这个时候，我们要做的事情其实很简单，那就是自定义typeHandler，自定义typeHandler我们有两种方式，一种是实现TypeHandler接口，还有一种简化的写法就是继承自BaseTypeHandler类，我这里先以第二种为例来进行说明。

@MappedJdbcTypes({JdbcType.***VARCHAR***})  
@MappedTypes({Date.**class**})  
**public class** MyDateTypeHandler **extends** BaseTypeHandler<Date> {  
 **public void** setNonNullParameter(PreparedStatement preparedStatement, **int** i, Date date, JdbcType jdbcType) **throws** SQLException {  
 preparedStatement.setString(i, String.*valueOf*(date.getTime()));  
 }  
  
 **public** Date getNullableResult(ResultSet resultSet, String s) **throws** SQLException {  
 **return new** Date(resultSet.getLong(s));  
 }  
  
 **public** Date getNullableResult(ResultSet resultSet, **int** i) **throws** SQLException {  
 **return new** Date(resultSet.getLong(i));  
 }  
  
 **public** Date getNullableResult(CallableStatement callableStatement, **int** i) **throws** SQLException {  
 **return** callableStatement.getDate(i);  
 }  
}

**关于这个类我说如下几点：**

1. @MappedJdbcTypes定义的是JdbcType类型，这里的类型不可自己随意定义，必须要是枚举类org.apache.ibatis.type.JdbcType所枚举的数据类型。
2. @MappedTypes定义的是JavaType的数据类型，描述了哪些Java类型可被拦截。
3. 在我们启用了我们自定义的这个TypeHandler之后，数据的读写都会被这个类所过滤
4. 在setNonNullParameter方法中，我们重新定义要写往数据库的数据。
5. 在另外三个方法中我们将从数据库读出的数据类型进行转换。

**注册typeHandler**

我们需要在我们的mybatis配置文件中注册typeHandler，注册有两种不同的方式，可以像下面这样一个类一个类的注册：

<**typeHandlers**>  
 <**typeHandler handler="com.whz.MyDateTypeHandler"**/>  
</**typeHandlers**>

也可以直接注册一个包中所有的typeHandler，系统在启动时会自动扫描包下的所有文件，如下：

<**typeHandlers**>  
 <**package name="com.whz"**/>  
</**typeHandlers**>

这样配置完成之后，我们的目的就达到了，当我们进行数据库的读取操作的时候，秒数就会自动转为Date对象。

**查询时将jdbcType转为JavaType**

<**resultMap id="userResultMap" type="org.sang.bean.User"**>  
 <**result typeHandler="org.sang.db.MyDateTypeHandler"   
 column="regTime" jdbcType="VARCHAR"  
 property="regTime" javaType="java.util.Date"**/>  
</**resultMap**>

<**select id="getUser" resultMap="userResultMap"**>  
 select \* from user  
</**select**>

**插入时将JavaType转为jdbcType**

*<!--方法一：-->*<**insert id="insertUser1" parameterType="org.sang.bean.User"**>  
 INSERT INTO user4(username,password,regTime)  
 VALUES (#{username},#{password},#{regTime,javaType=Date,jdbcType=VARCHAR,typeHandler=org.sang.db.MyDateTypeHandler})  
</**insert**>  
*<!--方法二：-->*<**insert id="insertUser2"**>  
 INSERT INTO user4(username,password,regTime)  
 VALUES (#{username},#{password},#{regTime,javaType=Date,jdbcType=VARCHAR})  
</**insert**>  
*<!--方法三：-->*<**insert id="insertUser3"**>  
 INSERT INTO user4(username,password,regTime)  
 VALUES (#{username},#{password},#{regTime,typeHandler=org.sang.db.MyDateTypeHandler})  
</**insert**>

这三种效果都是一样的，都是在插入的时候将数据Date对象转为秒数。

### 枚举类型typeHandler

在Java中，我们经常使用枚举类型来对一些字段进行建模，比如性别，然而在数据库中，我们通常将性别字段设置为tinyint等类型，这时我们就需要使用到枚举类型的typeHandler，关于这个的使用示例，在网上一大堆，我就不举例了。

## <objectFactory/>

MyBatis 每次创建结果对象的新实例时，它都会使用一个对象工厂（ObjectFactory）实例来完成。 默认的对象工厂需要做的仅仅是实例化目标类，要么通过默认构造方法，要么在参数映射存在的时候通过参数构造方法来实例化。在大部分场景下我们都不用修改，如果想覆盖对象工厂的默认行为，则可以通过创建自己的对象工厂并添加相应的配置即可。

**public class** ExampleObjectFactory **extends** DefaultObjectFactory {  
   
 @Override   
 **public** Object create(Class type) {  
 **if**(type.equals(User.**class**)){  
 User user = (User)**super**.create(type);   
 *// 这里可做一些操作* **return** user;   
 }   
 **return super**.create(type);   
 }  
   
 @Override   
 **public void** setProperties(Properties properties) {  
 Iterator iterator= properties.keySet().iterator();  
 **while**(iterator.hasNext()){   
 *//迭代器 输出配置文件定义的数据* String value = String.*valueOf*(iterator.next());   
 System.***out***.println(properties.getProperty(value));   
 }   
 *//传入property* **super**.setProperties(properties);   
 }   
   
 @Override   
 **public** <T> **boolean** isCollection(Class<T> type) {  
 **return** Connection.**class**.isAssignableFrom(type);  
 }   
   
}

**objectFactory配置**

<**objectFactory type="com.whz.ExampleObjectFactory"**>  
 <**property name="someProperty" value="100"**/>  
</**objectFactory**>

ObjectFactory 接口很简单，它包含两个创建用的方法，一个是处理默认构造方法的，另外一个是处理带参数的构造方法的。 最后，setProperties 方法可以被用来配置 ObjectFactory，在初始化你的 ObjectFactory 实例后， objectFactory 元素体中定义的属性会被传递给 setProperties 方法。

## <plugins/>

。。。

## <environments>

配置环境可以注册多个数据源，每一个数据源分为两大部分：一个是数据库源的配置，另一个是数据库事物的配置。如：

<**environments default="development"**>  
 <**environment id="development"**>  
 <**transactionManager type="JDBC"** >  
 <**property name="autoCommit" value="false"**/>  
 </**transactionManager**>  
 <**dataSource type="POOLED"**>  
 <**property name="driver" value="${driver}"** />  
 <**property name="url" value="${url}"** />  
 <**property name="username" value="${username}"** />  
 <**property name="password" value="${password}"** />  
 </**dataSource**>  
 </**environment**>  
</**environments**>

* environments中的属性default，标明在缺省的情况下，我们将启动哪个数据源配置。
* environment元素是配置一个数据源的开始，属性id是设置这个数据源的标志，以便Mybatis上下文使用它。

**数据库事务配置：**

* transactionManager配置的是数据库事务，其中type属性有3种配置方式：

1. JDBC，采用JDBC方式管理事务，在独立编码中我们常常使用。
2. MANAGED，采用容器方式管理事务，在JNDI数据源中常用。
3. 自定义，由使用者自定义数据库事务管理办法，适用于特殊应用。

* property元素则是可以配置数据源的各类属性，我们这配置了autoCommit=false，这时要求数据源不自动提交。

**数据源配置：**

* dataSource标签，是配置数据源连接的信息，type属性是提供我们对数据库连接方式的配置，同样MyBatis提供这么几种配置方式：

1. UNPOOLED，非连接池数据库。
2. POOLED，连接池数据库。
3. JNDI，JNDI数据源。
4. 自定义数据源。

其中，配置的property元素，就是定义数据库的各类参数。

## <databaseIdProvider/>

...

## <mappers>

既然 MyBatis 的行为已经由上述元素配置完了，我们现在就要定义 SQL 映射语句了。但是首先我们需要告诉 MyBatis 到哪里去找到这些语句。 Java 在自动查找这方面没有提供一个很好的方法，所以最佳的方式是告诉 MyBatis 到哪里去找映射文件。你可以使用相对于类路径的资源引用， 或完全限定资源定位符（包括 file:/// 的 URL），或类名和包名等。例如：

*<!-- 使用相对于类路径的资源引入 -->*<**mappers**>  
 <**mapper resource="org/mybatis/builder/AuthorMapper.xml"**/>  
 <**mapper resource="org/mybatis/builder/BlogMapper.xml"**/>  
 <**mapper resource="org/mybatis/builder/PostMapper.xml"**/>  
</**mappers**>  
*<!-- 使用完全限定资源定位符引入 -->*<**mappers**>  
 <**mapper url="file:///var/mappers/AuthorMapper.xml"**/>  
 <**mapper url="file:///var/mappers/BlogMapper.xml"**/>  
 <**mapper url="file:///var/mappers/PostMapper.xml"**/>  
</**mappers**>  
*<!-- 使用类注册引入 -->*<**mappers**>  
 <**mapper class="org.mybatis.builder.AuthorMapper"**/>  
 <**mapper class="org.mybatis.builder.BlogMapper"**/>  
 <**mapper class="org.mybatis.builder.PostMapper"**/>  
</**mappers**>  
*<!-- 使用报名引入 -->*<**mappers**>  
 <**package name="org.mybatis.builder"**/>  
</**mappers**>

# 常用案例参考

## 1、动态条件查询

**public interface** IEmployeerMapper {  
 List<Map> findEmployeerByCondition1(Map condition);  
 List<Employeer> findEmployeerByCondition2(Map condition);  
 List<Employeer> findEmployeerByCondition3(Employeer condition);  
}

<**sql id="whereCondition"**>  
 <**trim suffixOverrides="and"**>  
 <**where**>  
 <**if test="name!=null"**> employeer\_name = #{name} and </**if**>  
 <**if test="age!=null"**> employeer\_age = #{age} and </**if**>  
 <**if test="department!=null"**> employeer\_department = #{department} and </**if**>  
 <**if test="worktype!=null"**> employeer\_worktype = #{worktype} </**if**>  
 </**where**>  
 </**trim**>  
</**sql**>  
<**select id="findEmployeerByCondition1" parameterType="map" resultType="map"**>  
 select *\** from `t\_employeer`  
 <**include refid="whereCondition"**/>  
</**select**>  
<**select id="findEmployeerByCondition2" parameterType="map" resultType="alias\_Employeer"**>  
 select *\** from `t\_employeer`  
 <**include refid="whereCondition"**/>  
</**select**>  
*<!--使用这种方法时，<if test="xxx"/>里的xxx必须对应持久化对象的字段名称（而不是setXxx方法）-->*<**select id="findEmployeerByCondition3" parameterType="alias\_Employeer" resultType="alias\_Employeer"**>  
 select *\** from `t\_employeer`  
 <**trim suffixOverrides="and"**>  
 <**where**>  
  
 <**if test="employeer\_name!=null"**> employeer\_name = #{employeer\_name} and </**if**>  
 <**if test="employeer\_name!=null"**> employeer\_age = #{age} and </**if**>  
 <**if test="employeer\_name!=null"**> employeer\_department = #{employeer\_name} and </**if**>  
 <**if test="employeer\_name!=null"**> employeer\_worktype = #{employeer\_name} </**if**>  
 </**where**>  
 </**trim**>  
</**select**>

测试：

Map condition = **new** HashMap<String,String>();  
condition.put(**"age"**,30);  
List<Map> employeerList = session.getMapper(IEmployeerMapper.**class**).findEmployeerByCondition1(condition);  
List<Employeer> employeerList = session.getMapper(IEmployeerMapper.**class**).findEmployeerByCondition2(condition);  
  
Employeer condition = **new** Employeer();  
condition.setEmployeer\_age(30);  
List<Employeer> employeerList = session.getMapper(IEmployeerMapper.**class**).findEmployeerByCondition3(condition);

## Insert返回自增主键

**public interface** IEmployeerMapper {  
 **int** addEmployeer(Employeer employeer);  
}

*<!-- useGeneratedKeys设置为"true"表明要MyBatis获取由数据库自动生成的主键；keyProperty="id"指定把获取到的主键值注入到Employeer的id属性 -->*<**insert id="addEmployeer" parameterType="alias\_Employeer" useGeneratedKeys="true" keyProperty="employeer\_id"**>  
 insert into `t\_employeer`(employeer\_name,employeer\_age,employeer\_department,employeer\_worktype)  
 values(#{employeer\_name},#{employeer\_age},#{employeer\_department},#{employeer\_worktype})  
</**insert**>

测试：

Employeer employeer = **new** Employeer();  
employeer.setEmployeer\_age1(56);  
employeer.setEmployeer\_name(**"王五"**);  
**int** resultCount = session.insert(**"com.whz.mapperinterface.IEmployeerMapper.addEmployeer"**, employeer );  
System.***out***.printf(**"获取自增主键employeer\_id :%d "** , employeer.getEmployeer\_id()); *//获取插入对象的id*