

-

整理自互联网，仅供学习交流。



2018-10-30

[公司名称]

[公司地址]

目录

[一、 课程目标 1](#_Toc2538050)

[二、 Mybatis快速入门 1](#_Toc2538051)

[1.1 2.1 Mybatis介绍 1](#_Toc2538052)

[1.2 2.2 Mybatis环境搭建 1](#_Toc2538053)

[1.2.1 添加Maven坐标 1](#_Toc2538054)

[1.2.2 2.2.2建表 1](#_Toc2538055)

[1.2.3 添加mybatis配置文件 2](#_Toc2538056)

[1.2.4 定义表的实体类 2](#_Toc2538057)

[1.2.5 定义userMapper接口 2](#_Toc2538058)

[1.2.6 定义操作users表的sql映射文件userMapper.xml 2](#_Toc2538059)

[1.2.7 2.2.7mybatis.xml文件中加载配置文件 3](#_Toc2538060)

[1.2.8 2.2.8mybatis.xml测试方法 3](#_Toc2538061)

[1.3 增加案例 3](#_Toc2538062)

[1.4 2.2.4 删除 4](#_Toc2538063)

[三、 sql注入案例 4](#_Toc2538064)

[1.5 3.1创建表+测试数据 4](#_Toc2538065)

[1.6 3.2 jdbc进行加载 4](#_Toc2538066)

[1.7 3.3将username的值设置为 5](#_Toc2538067)

[1.8 3.4sql注入解决办法 5](#_Toc2538068)

[1.9 3.4 mybatis中#与$区别 6](#_Toc2538069)

[1.9.1 3.4.1创建UserTable 7](#_Toc2538070)

[1.9.2 3.4.2创建UserTable 7](#_Toc2538071)

[1.9.3 3.4.3userTableMapper.xml 7](#_Toc2538072)

[1.9.4 3.4.4 测试SQL注入 7](#_Toc2538073)

[1.9.5 3.4.5 总结 8](#_Toc2538074)

[四、 Mybatis 注解使用 8](#_Toc2538075)

[1.10 4.1 建立注解Mapper 8](#_Toc2538076)

[1.11 4.2 加入mybatis.xml 8](#_Toc2538077)

[1.12 4.3 运行测试 8](#_Toc2538078)

[二、 五、Generator使用 9](#_Toc2538079)

# 我的

## MyBatis简介

### MyBatis

MyBatis 本是[apache](http://baike.baidu.com/view/28283.htm)的一个开源项目[iBatis](http://baike.baidu.com/view/628102.htm), 2010年这个项目由apache software foundation 迁移到了google code，并且改名为MyBatis，实质上Mybatis对ibatis进行一些改进。

MyBatis是一个优秀的持久层框架，它对jdbc的操作数据库的过程进行封装，使开发者只需要关注 SQL 本身，而不需要花费精力去处理例如注册驱动、创建connection、创建statement、手动设置参数、结果集检索等jdbc繁杂的过程代码。

对jdbc的封装框架有哪些：Hibernate,dbutils,jdbcTemplate[spring]，mybatis

原理：Mybatis通过**xml或注解**的方式将要执行的各种statement（statement、preparedStatemnt、CallableStatement）配置起来，并通过java对象和statement中的sql进行映射生成最终执行的sql语句，最后由mybatis框架执行sql并将结果映射成java对象并返回。

### jdbc程序代码

|  |
| --- |
|  |

**存在的问题**

* 数据库连接频繁开启和关闭，会严重影响数据库的性能。
* 代码中存在硬编码，分别是数据库部分的硬编码和SQL执行部分的硬编码。

### MyBatis的框架核心

|  |
| --- |
| 1. mybatis配置文件，包括**Mybatis全局配置文件和Mybatis映射文件**，其中全局配置文件配置了数据源、事务等信息；映射文件配置了SQL执行相关的 信息。 2. mybatis通过读取配置文件信息（全局配置文件和映射文件），构造出**SqlSessionFactory，**即会话工厂。 3. 通过SqlSessionFactory，可以创建**SqlSession**即会话。Mybatis是通过SqlSession来操作数据库的。 4. SqlSession本身不能直接操作数据库，它是通过底层的**Executor**执行器接口来操作数据库的。Executor接口有两个实现类，一个是普通执行器，一个是**缓存执行器（默认）**。 5. Executor执行器要处理的SQL信息是封装到一个底层对象**MappedStatement**中。该对象包括：SQL语句、输入参数映射信息、输出结果集映射信息。其中输入参数和输出结果的映射类型包括**HashMap集合对象、POJO对象类型**。 |

## 逆向工程

### 简介

**简单点说，就是通过数据库中的单表，自动生成java代码。**

Mybatis官方提供了逆向工程

可以针对单表自动生成mybatis代码（mapper.java\mapper.xml\po类）

企业开发中，逆向工程是个很常用的工具。

### 下载逆向工程

<https://github.com/mybatis/generator/releases/tag/mybatis-generator-1.3.2>

### 使用方法

1. 创建简单的java项目
2. 导入jar包,创建generator配置文件；
3. 使用java类来执行逆向工程；
4. 把生成的代码拷贝到项目中。
5. 在正式项目中使用逆向工程生成的代码

#### 第一步：创建generator配置文件

在classpath下，创建generator.xml配置文件：（文件内容可以从逆向工程的jar包中docs目录下的index.html中找到相关代码）

|  |
| --- |
| <?xml version=*"1.0"* encoding=*"UTF-8"*?>  <!DOCTYPE generatorConfiguration  PUBLIC "-//mybatis.org//DTD MyBatis Generator Configuration 1.0//EN"  "http://mybatis.org/dtd/mybatis-generator-config\_1\_0.dtd">  <generatorConfiguration>    <context id=*"mysqlTable"* targetRuntime=*"MyBatis3"*>  <!-- 1.数据连接参数 -->  <jdbcConnection driverClass=*"com.mysql.jdbc.Driver"*  connectionURL=*"jdbc:mysql://localhost:3306/mybatis"*  userId=*"root"*  password=*"123456"*>  </jdbcConnection>    <!-- 2.默认false，把JDBC DECIMAL 和 NUMERIC 类型解析为 Integer，为 true时把JDBC DECIMAL  和 NUMERIC 类型解析为java.math.BigDecimal -->  <javaTypeResolver >  <property name=*"forceBigDecimals"* value=*"false"* />  </javaTypeResolver>  <!-- 3.生成模型的位置 -->  <javaModelGenerator targetPackage=*"com.gyf.backoffice.domain"* targetProject=*".\src"*>  <!-- enableSubPackages:是否让schema作为包的后缀 -->  <property name=*"enableSubPackages"* value=*"true"* />  <property name=*"trimStrings"* value=*"true"* />  </javaModelGenerator>  <!-- 4.targetProject:mapper映射文件生成的位置 -->  <sqlMapGenerator targetPackage=*"com.gyf.backoffice.mapper"* targetProject=*".\src"*>  <!-- enableSubPackages:是否让schema作为包的后缀 -->  <property name=*"enableSubPackages"* value=*"true"* />  </sqlMapGenerator>  <!-- 5. targetPackage：mapper接口生成的位置 -->  <javaClientGenerator type=*"XMLMAPPER"* targetPackage=*"com.gyf.backoffice.mapper"*  targetProject=*".\src"*>  <property name=*"enableSubPackages"* value=*"true"* />  </javaClientGenerator>  <!-- 6.要生成的表 -->  <table tableName=*"items"*/>  <table tableName=*"orderdetail"*/>  <table tableName=*"orders"*/>  <table tableName=*"user"*/>  </context>  </generatorConfiguration> |

#### 第二步：使用java类来执行逆向工程

需要导入mysql的驱动包和mybatis的逆向工程包

|  |
| --- |
| public class Generator {  public static void main(String[] args) throws Exception{  List<String> warnings = new ArrayList<String>();  boolean overwrite = true;  File configFile = new File("config/generator.xml");  ConfigurationParser cp = new ConfigurationParser(warnings);  Configuration config = cp.parseConfiguration(configFile);  DefaultShellCallback callback = new DefaultShellCallback(overwrite);  MyBatisGenerator myBatisGenerator = new MyBatisGenerator(config,  callback, warnings);  myBatisGenerator.generate(null);  }  } |

#### 第三步：把生成的代码拷贝到项目中

如果正式项目中已经有po类所在的包了，那么就只需要拷贝po类到指定包下就可以。

如果正式项目中没有po包，那么就把逆向工程中整个po类的包拷贝过去。

Mapper.xml和mapper.java的拷贝与po类一样。

|  |
| --- |
|  |

#### 第四步：测试

逆向工程提供了很多查询方法，可以不用写sql,这个根hibernate有点类似

|  |
| --- |
|  |

## SSM 使用 mybatis 分页插件 pagehepler 实现分页

https://www.cnblogs.com/ainyi/p/8625672.html

### 依赖：

<!--分页插件-->

<dependency>

<groupId>com.github.pagehelper</groupId>

<artifactId>pagehelper</artifactId>

<version>5.0.0</version>

</dependency>

### 配置分页拦截器插件

官方文档有两种配置分页拦截器插件

1. 在 MyBatis 配置 xml 中配置拦截器插件

<!--

plugins在配置文件中的位置必须符合要求，否则会报错，顺序如下:

properties?, settings?,

typeAliases?, typeHandlers?,

objectFactory?,objectWrapperFactory?,

plugins?,

environments?, databaseIdProvider?, mappers?

-->

<plugins>

<!-- com.github.pagehelper为PageHelper类所在包名 -->

<plugin interceptor="com.github.pagehelper.PageInterceptor">

<!-- 使用下面的方式配置参数，后面会有所有的参数介绍 -->

<property name="param1" value="value1"/>

</plugin>

</plugins>

#### 2. 在 Spring 配置文件中配置拦截器插件

使用 spring 的属性配置方式，可以使用 plugins 属性像下面这样配置：

<bean id="sqlSessionFactory" class="org.mybatis.spring.SqlSessionFactoryBean">

<!-- 注意其他配置 -->

<property name="plugins">

<array>

<bean class="com.github.pagehelper.PageInterceptor">

<property name="properties">

<!--使用下面的方式配置参数，一行配置一个 -->

<value>

params=value1

</value>

</property>

</bean>

</array>

</property>

</bean>

### 调用方法

mapper 层 sql 语句按照一般查询写法，不用写分页：

<!-- 分页查询 -->

<select id="finds" resultType="country">

SELECT

id,

countryname,

countrycode

FROM

country

</select>

### mapper 层接口：

public List<Country> finds();

### 基础请求类

public class BaseRequest implements Serializable {

private static final long serialVersionUID = 1193444819529643410L;

private Integer pageNum;//页数

private Integer pageSize;//每页行数

private Boolean count;//是否查询总条数

public Integer getPageNum() {

return pageNum;

}

public void setPageNum(Integer pageNum) {

this.pageNum = pageNum;

}

public Integer getPageSize() {

return pageSize;

}

public void setPageSize(Integer pageSize) {

this.pageSize = pageSize;

}

public Boolean getCount() {

return count;

}

public void setCount(Boolean count) {

this.count = count;

}

@Override

public String toString() {

return "BaseRequest{" +

"pageNum=" + pageNum +

", pageSize=" + pageSize +

'}';

}

}

### service 业务层接口：

service 业务层实现类：PageHelper.startPage(1, 10);第一个参数表示第几页，第二个参数表示每页显示的记录数

执行完 PageHelper.startPage(1, 10);语句后，紧跟着的第一个select方法会被分页：List<Country> blogs = countryMapper.finds();

然后再用 PageInfo 对查询结果进行包装，PageInfo<Country> pageInfo = new PageInfo<Country>(blogs);

并将 pageInfo 返回到控制层

/\*\*

\* 查询

\*/

public PageInfo<Country> finds(Params params) {

//查询

int pageNo = params.getPageNo();

int pageSize = params.getPageSize();

PageHelper.startPage(pageNo, pageSize);

List<Country> blogs = countryMapper.finds();

//用PageInfo对结果进行包装

PageInfo<Country> pageInfo = new PageInfo<Country>(blogs);

return pageInfo;

}

## mybatis之mybatis-config.xml 配置中的标签详解

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

<!DOCTYPE configuration PUBLIC "-//mybatis.org//DTD Config 3.0//EN" "http://mybatis.org/dtd/mybatis-3-config.dtd">

<configuration>

<!-- 参数设置 -->

<settings>

<!-- 这个配置使全局的映射器启用或禁用缓存 -->

<setting name="cacheEnabled" value="true" />

<!-- 全局启用或禁用延迟加载。当禁用时，所有关联对象都会即时加载 -->

<setting name="lazyLoadingEnabled" value="true" />

<!-- 当启用时，有延迟加载属性的对象在被调用时将会完全加载任意属性。否则，每种属性将会按需要加载 -->

<setting name="aggressiveLazyLoading" value="true" />

<!-- 允许或不允许多种结果集从一个单独的语句中返回（需要适合的驱动） -->

<setting name="multipleResultSetsEnabled" value="true" />

<!-- 使用列标签代替列名。不同的驱动在这方便表现不同。参考驱动文档或充分测试两种方法来决定所使用的驱动 -->

<setting name="useColumnLabel" value="true" />

<!-- 允许JDBC支持生成的键。需要适合的驱动。如果设置为true则这个设置强制生成的键被使用，尽管一些驱动拒绝兼容但仍然有效（比如Derby） -->

<setting name="useGeneratedKeys" value="true" />

<!-- 指定MyBatis如何自动映射列到字段/属性。PARTIAL只会自动映射简单，没有嵌套的结果。FULL会自动映射任意复杂的结果（嵌套的或其他情况） -->

<setting name="autoMappingBehavior" value="PARTIAL" />

<!--当检测出未知列（或未知属性）时，如何处理，默认情况下没有任何提示，这在测试的时候很不方便，不容易找到错误。

NONE : 不做任何处理 (默认值)

WARNING : 警告日志形式的详细信息

FAILING : 映射失败，抛出异常和详细信息

-->

<setting name="autoMappingUnknownColumnBehavior" value="WARNING"/>

<!-- 配置默认的执行器。SIMPLE执行器没有什么特别之处。REUSE执行器重用预处理语句。BATCH执行器重用语句和批量更新 -->

<setting name="defaultExecutorType" value="SIMPLE" />

<!-- 设置超时时间，它决定驱动等待一个数据库响应的时间 -->

<setting name="defaultStatementTimeout" value="25000" />

<!--设置查询返回值数量，可以被查询数值覆盖 -->

<setting name="defaultFetchSize" value="100"/>

<!-- 允许在嵌套语句中使用分页-->

<setting name="safeRowBoundsEnabled" value="false"/>

<!--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。-->

<setting name="mapUnderscoreToCamelCase" value="false"/>

<!--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。-->

<setting name="localCacheScope" value="SESSION"/>

<!-- 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、VARCHAROTHER。-->

<setting name="jdbcTypeForNull" value="OTHER"/>

<!-- 指定哪个对象的方法触发一次延迟加载。-->

<setting name="lazyLoadTriggerMethods" value="equals,clone,hashCode,toString"/>

</settings>

<!-- 别名定义 -->

<typeAliases>

<typeAlias alias="pageAccessURL" type="com.lgm.mybatis.model.PageAccessURL" />

</typeAliases>

<!--自定义类型处理器 -->

<typeHandlers>

<!-- <typeHandler handler="com.xhm.util.BooleanTypeHandlder" /> -->

<!--扫描整个包下的自定义类型处理器-->

<package name="com.xhm.util"/>

</typeHandlers>

<!--plugins插件之 分页拦截器 -->

<plugins>

<plugin interceptor="com.xhm.util.PageInterceptor"></plugin>

</plugins>

<!--配置environment环境-->

<environments default="development">

<!-- 环境配置1，每个SqlSessionFactory对应一个环境 -->

<environment id="development1">

<!-- 事务配置 type= JDBC、MANAGED 1.JDBC:这个配置直接简单使用了JDBC的提交和回滚设置。它依赖于从数据源得到的连接来管理事务范围。

2.MANAGED:这个配置几乎没做什么。它从来不提交或回滚一个连接。而它会让容器来管理事务的整个生命周期（比如Spring或JEE应用服务器的上下文）。 默认情况下它会关闭连接。然而一些容器并不希望这样，因此如果你需要从连接中停止它，将closeConnection属性设置为false -->

<transactionManager type="JDBC" />

<!-- <transactionManager type="MANAGED">

<property name="closeConnection" value="false"/>

</transactionManager> -->

<!-- 数据源类型：type = UNPOOLED、POOLED、JNDI 1.UNPOOLED：这个数据源的实现是每次被请求时简单打开和关闭连接。它有一点慢，这是对简单应用程序的一个很好的选择，因为它不需要及时的可用连接。

不同的数据库对这个的表现也是不一样的，所以对某些数据库来说配置数据源并不重要，这个配置也是闲置的 2.POOLED：这是JDBC连接对象的数据源连接池的实现，用来避免创建新的连接实例时必要的初始连接和认证时间。

这是一种当前Web应用程序用来快速响应请求很流行的方法。 3.JNDI：这个数据源的实现是为了使用如Spring或应用服务器这类的容器，容器可以集中或在外部配置数据源，然后放置一个JNDI上下文的引用 -->

<dataSource type="UNPOOLED">

<property name="driver" value="com.mysql.jdbc.Driver" />

<property name="url" value="jdbc:mysql://localhost:3306/xhm" />

<property name="username" value="root" />

<property name="password" value="root" />

<!-- 默认连接事务隔离级别 <property name="defaultTransactionIsolationLevel" value=""

/> -->

</dataSource>

</environment>

<!-- 环境配置2 -->

<environment id="development2">

<transactionManager type="JDBC" />

<dataSource type="POOLED">

<property name="driver" value="com.mysql.jdbc.Driver" />

<property name="url" value="jdbc:mysql://localhost:3306/xhm" />

<property name="username" value="root" />

<property name="password" value="root" />

<!-- 在任意时间存在的活动（也就是正在使用）连接的数量 -->

<property name="poolMaximumActiveConnections" value="10" />

<!-- 任意时间存在的空闲连接数 -->

<property name="poolMaximumIdleConnections" value="5" />

<!-- 在被强制返回之前，池中连接被检查的时间 -->

<property name="poolMaximumCheckoutTime" value="20000" />

<!-- 这是给连接池一个打印日志状态机会的低层次设置，还有重新尝试获得连接，这些情况下往往需要很长时间（为了避免连接池没有配置时静默失败） -->

<property name="poolTimeToWait" value="20000" />

<!-- 发送到数据的侦测查询，用来验证连接是否正常工作，并且准备接受请求。 -->

<property name="poolPingQuery" value="NO PING QUERY SET" />

<!-- 这是开启或禁用侦测查询。如果开启，你必须用一个合法的SQL语句（最好是很快速的）设置poolPingQuery属性 -->

<property name="poolPingEnabled" value="false" />

<!-- 这是用来配置poolPingQuery多次时间被用一次。这可以被设置匹配标准的数据库连接超时时间，来避免不必要的侦测 -->

<property name="poolPingConnectionsNotUsedFor" value="0" />

</dataSource>

</environment>

<!-- 环境配置3 -->

<environment id="development3">

<transactionManager type="JDBC" />

<dataSource type="JNDI">

<property name="data\_source" value="java:comp/env/jndi/mybatis" />

<property name="env.encoding" value="UTF8" />

<!-- <property name="initial\_context" value=""/> <property name="env.encoding"

value="UTF8"/> -->

</dataSource>

</environment>

</environments>

<!-- 映射文件，mapper的配置文件 -->

<mappers>

<!--直接映射到相应的mapper文件-->

<mapper resource="com/xhm/mapper/UserMapper.xml"/>

<!--扫描包路径下所有xxMapper.xml文件-->

<package name="com.xhm.mapper"/>

</mappers>

</configuration>

# 面试题

## mybatis与hibernate的区别

### Mybatis技术特点：

好处：

1. 通过直接编写SQL语句，可以直接对SQL进行性能的优化；
2. 学习门槛低，学习成本低。只要有SQL基础，就可以学习mybatis，而且很容易上手；
3. 由于直接编写SQL语句，所以灵活多变，代码维护性更好。

缺点：

1. 不能支持数据库无关性，即数据库发生变更，要写多套代码进行支持，移植性不好。
   1. Mysql:limit
   2. Oracle:rownum
2. 需要编写结果映射。

### Hibernate技术特点：

好处：

1. 标准的orm框架，程序员不需要编写SQL语句。
2. 具有良好的数据库无关性，即数据库发生变化的话，代码无需再次编写。
   1. 以后,mysql数据迁移到oracle，只需要改方言配置

缺点:

1. 学习门槛高，需要对数据关系模型有良好的基础，而且在设置OR映射的时候，需要考虑好性能和对象模型的权衡。
2. 程序员不能自主的去进行SQL性能优化。

### Mybatis应用场景：

需求多变的互联网项目，例如电商项目。

### Hibernate应用**场**景：

需求明确、业务固定的项目，例如OA项目、ERP项目等。

## mybatis实现原理

dom4j+反射+xml

使用java解析xml技术进行mybatis.xml

读取配置文件中的mybatis.xml，获取到jdbc值。进行反射创建数据源。

创建会话工厂，获取到某个会话工厂的连接。

企业中没有删除语句，一般都是逻辑删除。

一般在表字段上加一个字段，0显示，1不显示。

查询的时候只查询字段为0的，定时备份值为1的数据。

# 常见问题

## There is no getter for property named 'id' in 'class java.lang.Integer'

当mapper的方法参数是基础类型且只有一个时，在mapper.xml文件中使用${}取值时，会报There is no getter for property named 'id' in 'class java.lang.Integer'异常。

解决方法

1、@Param注解

public AreaDict selectById(@Param("id") Integer id);

select \* from area\_dict where area\_dict\_id = ${id}

2、将${}换成#{}取值，#{}能防止sql注入，${}只是简单的字符串替换，#{}先进行预处理

select \* from area\_dict where area\_dict\_id = #{id}

3、通过${value}或${\_parameter}取值

select \* from area\_dict where area\_dict\_id = ${\_parameter}

select \* from area\_dict where area\_dict\_id = ${value}

## mapper.xml文件自动扫描注入的方法

1、配置好项目后在applicationContext-dao.xml中的 session 工厂配置里加入 mapperLocations 项 即可自动扫描目录下所有xml映射文件

<!-- spring和MyBatis完美整合 -->

<bean id="sqlSessionFactory" class="org.mybatis.spring.SqlSessionFactoryBean">

<!-- 指定数据源 -->

<property name="dataSource" ref="dataSource"/>

<!-- 具体指定xml文件，可不配 -->

<property name="configLocation" value="classpath:mybatis-config.xml"/>

<!-- 自动扫描mapping.xml文件，\*\*表示迭代查找 ,,也可在mybatis-config.xml中单独指定xml文件 -->

<property name="mapperLocations" value="classpath: com/yaosiyuan/dao/\*\*/\*.xml" />

<!-- 配置单个xml文件 -->

<!--<property name="mapperLocations" value="classpath:com/yaosiyuan/dao/xml/MsMerchantMapper.xml"/> -->

<!-- 配置多个xml文件 -->

<!--

<property name="mapperLocations">

<list>

<value>classpath:com/yaosiyuan/dao/xml/\*.xml</value>

</list>

<array>

<value>classpath:com/yaosiyuan/dao/xml/MsMerchantMapper.xml</value>

<value>classpath:com/yaosiyuan/dao/xml/MsMerchantMapper.xml</value>

</array>

</property>

-->

</bean>

2、mybatis-config.xml可以直接这样写个空白的

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

<!DOCTYPE configuration PUBLIC "-//mybatis.org//DTD Config 3.0//EN" "http://mybatis.org/dtd/mybatis-3-config.dtd">

<configuration>

<!-- 在这个文件放置一些全局性的配置

<typeAliases>

<typeAlias type="实体类" alias="别名"/>

</typeAliases> -->

<settings>

<setting name="cacheEnabled" value="false" />

</settings>

<!-- 以下内容不再需要手动配置 已经自动映射 -->

<mappers>

<!-- <mapper resource="com/gfan/api/framework/model/sqlMapper/TestuserMapper.xml" /> -->

</mappers>

</configuration>

mapperLocations方式直接配置dao层和mapper对应的映射关系

configLocation方式以中间代理文件(mybatis-config.xml)的形式, 配置dao层和mapper对应的映射关系

## Result Maps collection already contains value for…BaseResultMap

1.1 异常信息

mybatis出现Result Maps collection already contains value for…BaseResultMap的错误，

1.2 场景描述

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

1.3 异常原因

既有可能是generatorSqlmapCustom逆向工程热的货。

假如使用generatorSqlmapCustom逆向工程生成代码，即生成dao文件和mapper.xml文件，

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

1.4 解决方案

如果是再次生成代码，必须先将已经生成的代码删除，否则会在原文件中追加，仔细检查一下mapper.xml文件

看是否存在两个id为BaseResultMap的resultMap，删除多余的，只保留一个就可以解决问题。

## nested exception is org.mybatis.spring.MyBatisSystemException: nested exception is org.apache.ibatis.reflection.ReflectionException: There is no getter for property named '？' in '？'

## [Invalid bound statement (not found)错误的可能原因](https://www.cnblogs.com/liaojie970/p/8034525.html)

**其他原因导致此问题解决参考：**

**1.**检查xml文件所在package名称是否和Mapper interface所在的包名

<mapper namespace="me.tspace.pm.dao.UserDao">

 mapper的namespace写的不对！！！注意系修改。

**2.**UserDao的方法在UserDao.xml中没有，然后执行UserDao的方法会报此

**3.** UserDao的方法返回值是List<User>,而select元素没有正确配置ResultMap,或者只配置ResultType!

**4.**如果你确认没有以上问题,请任意修改下对应的xml文件,比如删除一个空行,保存.问题解决

5.看下mapper的XML配置路径是否正确

# 蚂蚁

## Mybatis快速入门

### Mybatis介绍

MyBatis是支持***普通SQL查询***，***存储过程***和***高级映射***的优秀持久层框架。MyBatis消除了几乎所有的JDBC代码和参数的手工设置以及对结果集的检索封装。MyBatis可以使用简单的XML或注解用于配置和原始映射，将接口和Java的POJO（Plain Old Java Objects，普通的Java对象）映射成数库中的记录.**JDBC-** 🡪**MyBatis-**🡪**Hibernate**

### Mybatis环境搭建

#### 添加Maven坐标

|  |
| --- |
| <dependencies>  <!-- https://mvnrepository.com/artifact/org.mybatis/mybatis -->  <dependency>  <groupId>org.mybatis</groupId>  <artifactId>mybatis</artifactId>  <version>3.4.4</version>  </dependency>  <!-- https://mvnrepository.com/artifact/mysql/mysql-connector-java -->  <dependency>  <groupId>mysql</groupId>  <artifactId>mysql-connector-java</artifactId>  <version>5.1.21</version>  </dependency>  </dependencies> |

#### 建表

|  |
| --- |
| CREATE TABLE users(id INT PRIMARY KEY AUTO\_INCREMENT, NAME VARCHAR(20), age INT);  INSERT INTO users(NAME, age) VALUES('Tom', 12);  INSERT INTO users(NAME, age) VALUES('Jack', 11); |

#### 添加mybatis配置文件

|  |
| --- |
| <?xml version=*"1.0"* encoding=*"UTF-8"*?>  <!DOCTYPE configuration PUBLIC "-//mybatis.org//DTD Config 3.0//EN" "http://mybatis.org/dtd/mybatis-3-config.dtd">  <configuration>  <environments default=*"development"*>  <environment id=*"development"*>  <transactionManager type=*"JDBC"* />  <dataSource type=*"POOLED"*>  <property name=*"driver"* value=*"com.mysql.jdbc.Driver"* />  <property name=*"url"* value=*"jdbc:mysql://localhost:3306/test"* />  <property name=*"username"* value=*"root"* />  <property name=*"password"* value=*"root"* />  </dataSource>  </environment>  </environments>  </configuration> |

#### 定义表的实体类

|  |
| --- |
| package com.entity;  public class User {  private int id;  private String name;  private int age;  //get,set方法  } |

#### 定义userMapper接口

|  |
| --- |
| **package** com.itmayiedu.mapper;  **import** com.itmayiedu.entity.User;  **public** **interface** UserMapper {  **public** User getUser(**int** id);  } |

#### 定义操作users表的sql映射文件userMapper.xml

|  |
| --- |
| <?xml version=*"1.0"* encoding=*"UTF-8"* ?>  <!DOCTYPE mapper PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN" "http://mybatis.org/dtd/mybatis-3-mapper.dtd">  <mapper namespace=*"com.itmayiedu.mapper.UserMapper"*>  <select id=*"getUser"* parameterType=*"int"* resultType=*"com.itmayiedu.entity.User"*>  SELECT \*  FROM users where id =#{id}  </select>  </mapper> |

#### mybatis.xml文件中加载配置文件

|  |
| --- |
| <mappers>  <mapper resource=*"mapper/userMapper.xml"* />  </mappers> |

#### mybatis.xml测试方法

|  |
| --- |
| import java.io.File;  import java.io.IOException;  import java.io.Reader;  import org.apache.ibatis.io.Resources;  import org.apache.ibatis.session.SqlSession;  import org.apache.ibatis.session.SqlSessionFactory;  import org.apache.ibatis.session.SqlSessionFactoryBuilder;  import com.itmayiedu.entity.User;  public class TestMybatis {  public static void main(String[] args) throws IOException {  String resource = "mybatis.xml";  // 读取配置文件  Reader reader = Resources.*getResourceAsReader*(resource);  // 获取会话工厂  SqlSessionFactory sqlSessionFactory = new SqlSessionFactoryBuilder().build(reader);  SqlSession openSession = sqlSessionFactory.openSession();  // 查询  String sql = "com.itmayiedu.mapper.UserMapper.getUser";  // 调用api查询  User user = openSession.selectOne(sql, 1);  System.*out*.println(user.toString());  }  } |

### 增加案例

Xml:

|  |
| --- |
| <insert id="addUser" parameterType="com.itmayiedu.entity.User" >    INSERT INTO users(NAME, age) VALUES(#{name}, #{age});  </insert> |

代码:

|  |
| --- |
| static public void add() throws IOException{  String resource = "mybatis.xml";  // 读取配置文件  Reader reader = Resources.getResourceAsReader(resource);  // 获取会话工厂  SqlSessionFactory sqlSessionFactory = new SqlSessionFactoryBuilder().build(reader);  SqlSession openSession = sqlSessionFactory.openSession();  // 查询  String sql = "com.itmayiedu.mapper.UserMapper.addUser";  // 调用api查询  User userPa = new User();  userPa.setAge(19);  userPa.setName("张三");  int reuslt = openSession.insert(sql, userPa);  System.out.println(reuslt);  } |

### 删除

Xml:

|  |
| --- |
| <delete id=*"delUser"* parameterType=*"int"* >  delete from users where id=#{id}  </delete> |

代码:

|  |
| --- |
| static public void delUser() throws IOException{  String resource = "mybatis.xml";  // 读取配置文件  Reader reader = Resources.*getResourceAsReader*(resource);  // 获取会话工厂  SqlSessionFactory sqlSessionFactory = new SqlSessionFactoryBuilder().build(reader);  SqlSession openSession = sqlSessionFactory.openSession();  // 查询  String sql = "com.itmayiedu.mapper.UserMapper.delUser";  int reuslt = openSession.delete(sql,1);  System.*out*.println(reuslt);  } |

## sql注入案例

### 创建表+测试数据

|  |
| --- |
| create table user\_table(  id int Primary key,  username varchar(30),  password varchar(30)  );  insert into user\_table values(1,'yushengjun-1','12345');  insert into user\_table values(2,'yushengjun-2','12345'); |

### jdbc进行加载

|  |
| --- |
| String username = "yushengjun-1";  String password = "12345";  String sql = "SELECT id,username FROM user\_table WHERE " + "username='" + username + "'AND " + "password='"  + password + "'";  Class.forName("com.mysql.jdbc.Driver");  Connection con = DriverManager.getConnection("jdbc:mysql://localhost:3306/test", "root", "root");  PreparedStatement stat = con.prepareStatement(sql);  System.out.println(stat.toString());  ResultSet rs = stat.executeQuery();  while (rs.next()) {  String id = rs.getString(1);  String name = rs.getString(2);  System.out.println("id:" + id + "---name:" + name);  } |

### 将username的值设置为

username='  OR 1=1 --

因为--表示SQL注释，因此后面语句忽略；

因为1=1恒成立，因此 username='' OR 1=1  恒成立，因此SQL语句等同于：

### 3.4sql注入解决办法

第一步：编译sql

第二步：执行sql

优点：能预编译sql语句

|  |
| --- |
| String username = "username=' OR 1=1 -- ";  String password = "12345";  // String sql = "SELECT id,username FROM user\_table WHERE " +  // "username='" + username + "'AND " + "password='"  // + password + "'";  String sql = "SELECT id,username FROM user\_table WHERE username=? AND password=?";  Class.*forName*("com.mysql.jdbc.Driver");  Connection con = DriverManager.*getConnection*("jdbc:mysql://localhost:3306/test", "root", "root");  PreparedStatement stat = con.prepareStatement(sql);  stat.setString(1, username);  stat.setString(2, password);  System.*out*.println(stat.toString());  ResultSet rs = stat.executeQuery();  while (rs.next()) {  String id = rs.getString(1);  String name = rs.getString(2);  System.*out*.println("id:" + id + "---name:" + name);  } |

### 3.4 mybatis中#与$区别

动态 sql 是 mybatis 的主要特性之一，在 mapper 中定义的参数传到 xml 中之后，在查询之前 mybatis 会对其进行动态解析。mybatis 为我们提供了两种支持动态 sql 的语法：#{} 以及 ${}。

在下面的语句中，如果 username 的值为 zhangsan，则两种方式无任何区别：

select \* from user where name = #{name};

select \* from user where name = ${name};

其解析之后的结果均为

select \* from user where name = 'zhangsan';

　但是 #{} 和 ${} 在预编译中的处理是不一样的。#{} 在预处理时，会把参数部分用一个占位符 ? 代替，变成如下的 sql 语句：

select \* from user where name = ?;

而 ${} 则只是简单的字符串替换，在动态解析阶段，该 sql 语句会被解析成

select \* from user where name = 'zhangsan';

以上，#{} 的参数替换是发生在 DBMS 中，而 ${} 则发生在动态解析过程中。

那么，在使用过程中我们应该使用哪种方式呢？

答案是，优先使用 #{}。因为 ${} 会导致 sql 注入的问题。看下面的例子：

　select \* from ${tableName} where name = #{name}

在这个例子中，如果表名为

 user; delete user; --

　　则动态解析之后 sql 如下：

select \* from user; delete user; -- where name = ?;

之后的语句被注释掉，而原本查询用户的语句变成了查询所有用户信息+删除用户表的语句，会对数据库造成重大损伤，极大可能导致服务器宕机。

但是表名用参数传递进来的时候，只能使用 ${} ，具体原因可以自己做个猜测，去验证。这也提醒我们在这种用法中要小心sql注入的问题。

#### 3.4.1创建UserTable

|  |
| --- |
| **package** com.itmayiedu.entity;  **public** **class** UserTable {  **private** **int** id;  **private** String userName;  **private** String passWord;  } |

#### 3.4.2创建UserTable

|  |
| --- |
| **pa*ckage com.itmayiedu.mapper;***  ***import com.itmayiedu.entity.UserTable;***  ***public interface UserTableMapper {***  ***public UserTable login(UserTable userTable);***  ***}*** |

#### 3.4.3userTableMapper.xml

|  |
| --- |
| <?xml version=*"1.0"* encoding=*"UTF-8"* ?>  <!DOCTYPE mapper PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN" "http://mybatis.org/dtd/mybatis-3-mapper.dtd">  <mapper namespace=*"com.itmayiedu.mapper.UserTableMapper"*>  <select id=*"login"* parameterType=*"com.itmayiedu.entity.UserTable"*  resultType=*"com.itmayiedu.entity.UserTable"*>  SELECT id ,username as userName FROM user\_table WHERE  username=${userName} AND password=${passWord}  </select>  </mapper> |

#### 3.4.4 测试SQL注入

|  |
| --- |
| **public class TestLoginMybatis3 {**  **public static void main(String[] args) throws IOException, SQLException, ClassNotFoundException {**  **String resource = "mybatis.xml";**  **// 读取配置文件**  **Reader reader = Resources.*getResourceAsReader*(resource);**  **// 获取会话工厂**  **SqlSessionFactory sqlSessionFactory = new SqlSessionFactoryBuilder().build(reader);**  **SqlSession openSession = sqlSessionFactory.openSession();**  **// 查询**  **String sql = "com.itmayiedu.mapper.UserTableMapper.login";**  **// 调用api查询**  **UserTable userTable = new UserTable();**  **userTable.setUserName("'' OR 1=1 -- ");**  **userTable.setPassWord("12345");**  **List<UserTable> listUserTable = openSession.selectList(sql, userTable);**  **for (UserTable ub : listUserTable) {**  **System.*out*.println(ub.getUserName());**  **}**  **}**  **}** |

#### 3.4.5 总结

优先使用 #{}。因为 ${} 会导致 sql 注入的问题

## Mybatis 注解使用

Mybatis提供了增删改查注解、@select @delete @update

### 建立注解Mapper

|  |
| --- |
| **import** org.apache.ibatis.annotations.Param;  **import** org.apache.ibatis.annotations.Select;  **import** com.itmayiedu.entity.User;  **public** **interface** UserTestMapper {  @Select("select \* from users where id = ${id};")  public User getUser(@Param("id") String id);  } |

### 加入mybatis.xml

|  |
| --- |
| <mapper class="com.itmayiedu.mapper.UserTestMapper" /> |

### 运行测试

|  |
| --- |
| public class TestMybatis3 {  public static void main(String[] args) throws IOException {  String resource = "mybatis.xml";  // 读取配置文件  Reader reader = Resources.*getResourceAsReader*(resource);  // 获取会话工厂  SqlSessionFactory sqlSessionFactory = new SqlSessionFactoryBuilder().build(reader);  SqlSession openSession = sqlSessionFactory.openSession();  // 调用api查询  UserTestMapper userTestMapper=openSession.getMapper(UserTestMapper.class);  System.*out*.println(userTestMapper.getUser("2"));  }  } |

## Generator使用

Generator 逆向生成 使用

#### 前言

最近做的项目（SSM+Shiro）的数据库表已经创建完成，一共有15张表，如果我们一个个去写pojo/bean的代码以及各种sql语句的话未免太过麻烦而且很容易出错，这个时候我们就需要MyBatis逆向工程去为我们生成这些基本的东西。

先来简单的了解一下什么是逆向工程。

### 什么是逆向工程

**官网解释浓缩版**：

MyBatis逆向工程需要用到的就是MyBatis官方提供的MyBatis Generator（MBG）。MBG是MyBatis和iBATIS的代码生成器，它将为所有版本的MyBatis以及版本2.2.0之后的iBATIS版本生成代码。MBG对简单CRUD（增删改查）的大部分数据库操作产生重大影响。但是您仍然需要为连接查询或存储过程手动编写SQL和对象代码。

**简要概括版：**

创建好数据库表之后，MBG可以根据数据库表自动为您生成pojo类、example类(用于添加条件，相当where语句后面的部分 )、mapper文件。

关于Mapper接口以及Example的实例的讲解可以查看这一篇文章：《Mapper接口以及Example的实例》  
<https://blog.csdn.net/biandous/article/details/65630783>

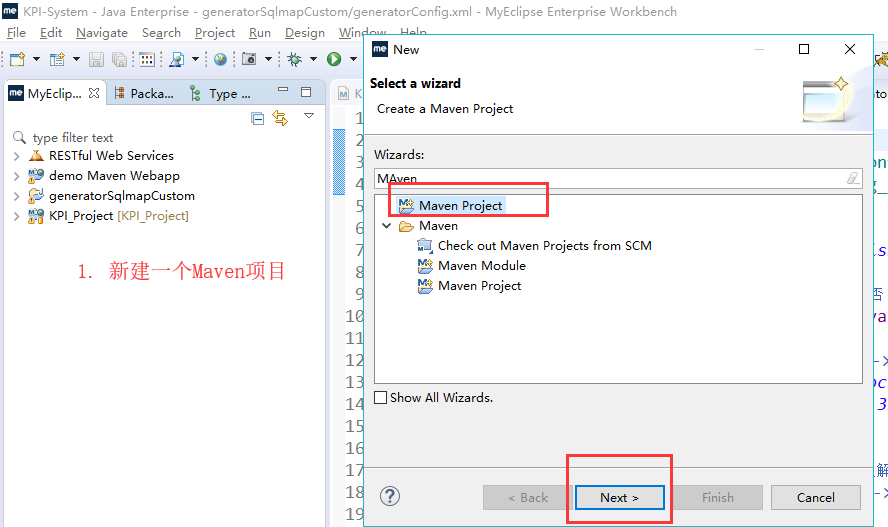
介绍完逆向工程是个什么东西后，接下来我们就要开始学习如何使用它了?

### 二 使用MyBatis逆向工程

考虑到大家的基础可能不同，所以我尽可能详细一点。

#### 新建Maven项目并添加相关依赖

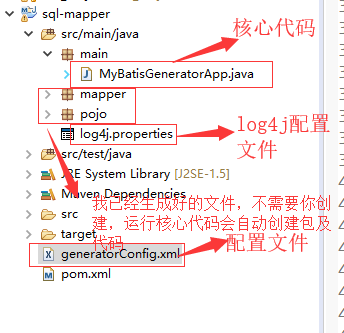
**1.新建Maven项目(普通的Java项目)：**



**2.添加相关依赖(pom.xml)：**

|  |
| --- |
| **<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"**  **xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">**  **<modelVersion>4.0.0</modelVersion>**  **<groupId>mybatis-generator</groupId>**  **<artifactId>sql-mapper</artifactId>**  **<version>0.0.1-SNAPSHOT</version>**  **<packaging>jar</packaging>**  **<name>sql-mapper</name>**  **<url>http://maven.apache.org</url>**  **<properties>**  **<project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>**  **</properties>**  **<dependencies>**  **<!--log4j -->**  **<!-- https://mvnrepository.com/artifact/log4j/log4j -->**  **<dependency>**  **<groupId>log4j</groupId>**  **<artifactId>log4j</artifactId>**  **<version>1.2.17</version>**  **</dependency>**  **<!--MyBatis Generator -->**  **<!-- https://mvnrepository.com/artifact/org.mybatis.generator/mybatis-generator-core -->**  **<dependency>**  **<groupId>org.mybatis.generator</groupId>**  **<artifactId>mybatis-generator-core</artifactId>**  **<version>1.3.5</version>**  **</dependency>**  **<!--mysql-connector-java -->**  **<!-- https://mvnrepository.com/artifact/mysql/mysql-connector-java -->**  **<dependency>**  **<groupId>mysql</groupId>**  **<artifactId>mysql-connector-java</artifactId>**  **<version>5.1.38</version>**  **</dependency>**  **<!--mybatis -->**  **<dependency>**  **<groupId>org.mybatis</groupId>**  **<artifactId>mybatis</artifactId>**  **<version>3.2.1</version>**  **</dependency>**  **</dependencies>**  **</project>** |

2.2 项目创建完成之后的目录结构



#### 2.3 MyBatis逆向工程代码编写

核心代码官网已经给出，地址为：<http://www.mybatis.org/generator/running/runningWithJava.html>

核心代码示例(推荐读取xml配置文件的形式，还有一种基于Java的配置这里就不做演示)：

List<String> warnings = new ArrayList<String>();

boolean overwrite = true;

//读取xml配置文件，推荐使用这种方式

File configFile = new File("generatorConfig.xml");

ConfigurationParser cp = new ConfigurationParser(warnings);

Configuration config = cp.parseConfiguration(configFile);

DefaultShellCallback callback = new DefaultShellCallback(overwrite);

MyBatisGenerator myBatisGenerator = new MyBatisGenerator(config, callback, warnings);

myBatisGenerator.generate(null);

本项目中使用代码MyBatisGeneratorApp.java：

|  |
| --- |
| public class MyBatisGeneratorApp {  public void generator() throws Exception{  List<String> warnings = new ArrayList<String>();  boolean overwrite = true;  //指定 逆向工程配置文件  File configFile = new File("generatorConfig.xml");  ConfigurationParser cp = new ConfigurationParser(warnings);  Configuration config = cp.parseConfiguration(configFile);  DefaultShellCallback callback = new DefaultShellCallback(overwrite);  MyBatisGenerator myBatisGenerator = new MyBatisGenerator(config,  callback, warnings);  myBatisGenerator.generate(null);  }  public static void main(String[] args) throws Exception {  try {  MyBatisGeneratorApp generatorSqlmap = new MyBatisGeneratorApp();  generatorSqlmap.generator();  } catch (Exception e) {  e.printStackTrace();  }    }  } |

#### 2.4 MyBatis逆向工程配置文件以及log4j配置文件编写

MyBatis GeneratorXML参考配置文件官网已经给出，地址为：<http://www.mybatis.org/generator/configreference/xmlconfig.html>

MyBatis逆向工程配置文件如下（注释已经解释的很详细了，我这里就不多做解释了）：

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

<!DOCTYPE generatorConfiguration

PUBLIC "-//mybatis.org//DTD MyBatis Generator Configuration 1.0//EN"

"http://mybatis.org/dtd/mybatis-generator-config\_1\_0.dtd">

<generatorConfiguration>

<context id="testTables" targetRuntime="MyBatis3">

<commentGenerator>

<!-- 是否去除自动生成的注释 true：是 ： false:否 -->

<property name="suppressAllComments" value="true" />

</commentGenerator>

<!--数据库连接的信息：驱动类、连接地址、用户名、密码 ,加上“useSSL=false”是因为我SSL连接数据库出现了错误 -->

<jdbcConnection driverClass="com.mysql.jdbc.Driver"

connectionURL="jdbc:mysql://localhost:3306/kpi\_project?useSSL=false"

userId="root" password="xxx">

</jdbcConnection>

<!-- 默认false，把JDBC DECIMAL 和 NUMERIC 类型解析为 Integer，为 true时把JDBC DECIMAL

和 NUMERIC 类型解析为java.math.BigDecimal -->

<javaTypeResolver>

<property name="forceBigDecimals" value="false" />

</javaTypeResolver>

<!-- targetProject:生成pojo类的位置 -->

<javaModelGenerator targetPackage="pojo"

targetProject=".\src\main\java">

<!-- enableSubPackages:是否让schema作为包的后缀 -->

<property name="enableSubPackages" value="false" />

<!-- 从数据库返回的值被清理前后的空格 -->

<property name="trimStrings" value="true" />

</javaModelGenerator>

<!-- targetProject:mapper映射文件生成的位置 -->

<sqlMapGenerator targetPackage="mapper" targetProject=".\src\main\java">

<!-- enableSubPackages:是否让schema作为包的后缀 -->

<property name="enableSubPackages" value="false" />

</sqlMapGenerator>

<!-- targetPackage：mapper接口生成的位置 -->

<javaClientGenerator type="XMLMAPPER"

targetPackage="mapper" targetProject=".\src\main\java">

<!-- enableSubPackages:是否让schema作为包的后缀 -->

<property name="enableSubPackages" value="false" />

</javaClientGenerator>

<!-- 指定数据库表 -->

<table schema="" tableName="com\_role"></table>

<table schema="" tableName="com\_rule"></table>

<table schema="" tableName="kpi\_assess"></table>

<table schema="" tableName="kpi\_assign"></table>

<table schema="" tableName="kpi\_complain"></table>

<table schema="" tableName="kpi\_index"></table>

<table schema="" tableName="kpi\_institute"></table>

<table schema="" tableName="kpi\_peformance"></table>

<table schema="" tableName="kpi\_result"></table>

<table schema="" tableName="kpi\_score"></table>

<table schema="" tableName="sys\_permission"></table>

<table schema="" tableName="sys\_role"></table>

<table schema="" tableName="sys\_role\_perm"></table>

<table schema="" tableName="sys\_user"></table>

<table schema="" tableName="sys\_user\_role"></table>

</context>

</generatorConfiguration>

我的log4j的配置文件如下：

log4j.rootLogger=DEBUG, Console

#Console

log4j.appender.Console=org.apache.log4j.ConsoleAppender

log4j.appender.Console.layout=org.apache.log4j.PatternLayout

log4j.appender.Console.layout.ConversionPattern=%d [%t] %-5p [%c] - %m%n

log4j.logger.java.sql.ResultSet=INFO

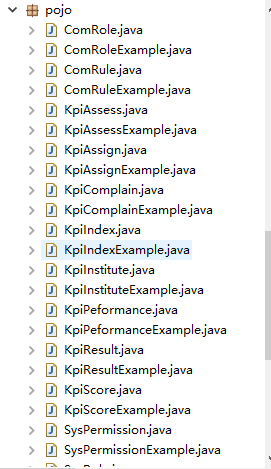
log4j.logger.org.apache=INFO

log4j.logger.java.sql.Connection=DEBUG

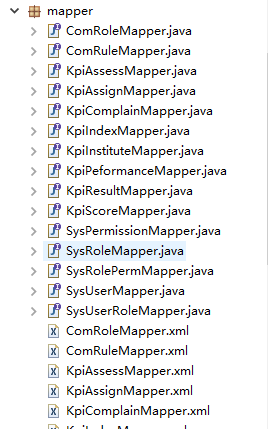
log4j.logger.java.sql.Statement=DEBUG

log4j.logger.java.sql.PreparedStatement=DEBUG

#### 2.5 运行结果

**pojo及example类：**  


**mapper接口及mapper.xml:**



我们已经介绍了MyBatis逆向工程以及它的使用，下面我们简单的总结与补充一下所学的知识。

三 总结

**什么是逆向工程？**

MBG可以根据数据库表自动为您生成pojo类、example类(用于添加条件，相当where语句后面的部分 )、mapper文件`。

**如何使用逆向工程？**

pom文件（添加相关jar包）->逆向工程核心代码编写->创建generatorConfig.xml配置文件(可以加上log4j日志打印配置文件)

### Mybatis generator不生成example

Mybatis Generator默认设置会生成一大堆罗哩罗嗦的Example类,主要是用各种不同的条件来操作数据库,大部分是用不到的,用到的时候手工修改mapper和接口文件就行了。

|  |
| --- |
| <table schema="general" tableName="tb\_table\_name" domainObjectName="EntityName"  enableCountByExample="false" enableUpdateByExample="false" enableDeleteByExample="false"  enableSelectByExample="false" selectByExampleQueryId="false" >  <property name="useActualColumnNames" value="true"/>  </table> |

注意：

domainObjectName：表名

设置驼峰命名法 ：<property name="useActualColumnNames" value="true"/>

Example类用于构造复杂的筛选条件。

基本概念

Criterion

Criterion是最基本,最底层的Where条件，用于字段级的筛选，feild用于指代字段名字,列举如下:

只有一个条件,不需要其他参考值

feild IS NOLL

feild IS NOT NULL

与一个参考值进行算数运算

feild > value

feild >= value

feild = value

feild <> value

feild <= value

feild < value

与一个参考值进行模糊查询,参值中的%,?只能在构造查询条件时手动指定。

feild LIKE value

feild NOT LIKE value

介于两个参考值之间

feild BETWEEN value AND secondValue

在或不在一个参考值集合中,item来自于value集合

feild IN (item,item,item,...)

feild NOT IN (item,item,item,...)

MyBatis Generator会为每个字段产生如上的Criterion，如果表的字段比较多,产生的Example类会十分庞大。理论上通过Example类可以构造你想到的任何筛选条件。

Criteria

Criteria包含一个Cretiron的集合,每一个Criteria对象内包含的Cretiron之间是由AND连接的,是逻辑与的关系。

oredCriteria

Example内有一个成员叫oredCriteria,是Criteria的集合,就想其名字所预示的一样，这个集合中的Criteria是由OR连接的，是逻辑或关系。oredCriteria就是ORed Criteria。

# 18黑马

## Mybatis入门

### 环境准备

#### 创建数据库表

|  |
| --- |
|  |

#### 下载MyBatis

mybaits的代码由github.com管理，下载地址：<https://github.com/mybatis/mybatis-3/releases>

#### 创建项目导包

导入下面的包

|  |
| --- |
|  |
|  |

#### 添加log4j.properties

Mybatis使用的日志包是log4j的，所以需要添加log4j.properties。

在classpath下创建log4j.properties如下：【文件内容可以从mybatis-3.2.7.pdf中拷贝】

# Global logging configuration

log4j.rootLogger=DEBUG, stdout

# Console output...

log4j.appender.stdout=org.apache.log4j.ConsoleAppender

log4j.appender.stdout.layout=org.apache.log4j.PatternLayout

log4j.appender.stdout.layout.ConversionPattern=%5p [%t] - %m%n

日志级别在开发阶段设置成DEBUG，在生产阶段设置成INFO或者ERROR。

### 开发步骤

|  |
| --- |
| 1. 创建PO（model）类，根据需求创建； 2. 创建全局配置文件SqlMapConfig.xml； 3. 编写映射文件； 4. 加载映射文件，在SqlMapConfig.xml中进行加载； 5. 编写测试程序，即编写Java代码，连接并操作数据库。   思路：   1. 读取配置文件； 2. 通过SqlSessionFactoryBuilder创建SqlSessionFactory会话工厂。 3. 通过SqlSessionFactory创建SqlSession。 4. 调用SqlSession的操作数据库方法。 5. 关闭SqlSession。 |

#### 创建PO类

|  |
| --- |
|  |

#### 创建SqlMapConfig.xml

在classpath(src)下，创建SqlMapConfig.xml文件【SqlMapConfig.xml（文件头可以从mybatis-3.2.7.pdf文档的2.1.2小节中拷贝）】

|  |
| --- |
| <?xml version=*"1.0"* encoding=*"UTF-8"* ?>  <!DOCTYPE configuration  PUBLIC "-//mybatis.org//DTD Config 3.0//EN"  "http://mybatis.org/dtd/mybatis-3-config.dtd">  <configuration>  <!-- 配置mybatis的环境信息 -->  <environments default=*"development"*>  <environment id=*"development"*>  <!-- 配置JDBC事务控制，由mybatis进行管理 -->  <transactionManager type=*"JDBC"*></transactionManager>  <!-- 配置数据源，采用dbcp连接池 -->  <dataSource type=*"POOLED"*>  <property name=*"driver"* value=*"com.mysql.jdbc.Driver"*/>  <property name=*"url"* value=*"jdbc:mysql://localhost:3306/mybatis?useUnicode=true&amp;characterEncoding=utf8"*/>  <property name=*"username"* value=*"root"*/>  <property name=*"password"* value=*"123456"*/>  </dataSource>  </environment>  </environments>  </configuration> |

#### 映射文件

在classpath下，创建sqlmap文件夹。在sqlmap目录下，创建User.xml映射文件。

**【Mybatis的映射文件头（可以从mybatis-3.2.7.pdf文件中拷贝）】**

|  |
| --- |
| <?xml version=*"1.0"* encoding=*"UTF-8"* ?>  <!DOCTYPE mapper  PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"  "http://mybatis.org/dtd/mybatis-3-mapper.dtd">  <!--  namespace：命名空间，它的作用就是对SQL进行分类化管理，可以理解为SQL隔离  注意：使用mapper代理开发时，namespace有特殊且重要的作用  -->  <mapper namespace=*"test"*>  <!--  [id]：statement的id，要求在命名空间内唯一  [parameterType]：入参的java类型  [resultType]：查询出的单条结果集对应的java类型  [#{}]： 表示一个占位符?  [#{id}]：表示该占位符待接收参数的名称为id。注意：如果参数为简单类型时，#{}里面的参数名称可以是任意定义  -->  <select id=*"findUserById"* parameterType=*"int"* resultType=*"com.gyf.domain.User"*>  SELECT \* FROM USER WHERE id = #{id}  </select>  </mapper> |

#### 配置文件加载映射文件

|  |
| --- |
|  |

#### 测试类

|  |
| --- |
|  |

### 2.3 更多案例讲解

#### 模糊查询用户信息

|  |
| --- |
| <!--  [${}]：表示拼接SQL字符串  [${value}]：表示要拼接的是简单类型参数。  注意：  1、如果参数为简单类型时，${}里面的参数名称必须为value  2、${}会引起SQL注入，一般情况下不推荐使用。但是有些场景必须使用${}，比如order by ${colname}  -->  <select id=*"findUserByName"* parameterType=*"String"* resultType=*"com.gyf.domain.User"*>  SELECT \* FROM USER WHERE username like '%${value}%'  </select> |
|  |

#### 插入用户信息

|  |
| --- |
|  |
|  |

#### 删除用户

|  |
| --- |
|  |
|  |

#### 更新用户

|  |
| --- |
|  |
|  |

#### 主键返回之MySQL自增主键

思路：

* MySQL自增主键，是指在insert之前MySQL会自动生成一个自增的主键。
* 我们可以通过MySQL的函数获取到刚插入的自增主键:

LAST\_INSERT\_ID()

* 这个函数是在insert语句之后去调用。

|  |
| --- |
| <insert id="insertUser" parameterType="com.gyf.domain.User">  <!--  [selectKey标签]：通过select查询来生成主键  [keyProperty]：指定存放生成主键的属性  [resultType]：生成主键所对应的Java类型  [order]：指定该查询主键SQL语句的执行顺序，相对于insert语句  [last\_insert\_id]：MySQL的函数，要配合insert语句一起使用 -->  <selectKey keyProperty="id" resultType="int" order="AFTER">  SELECT LAST\_INSERT\_ID()  </selectKey>  <!-- 如果主键的值是通过MySQL自增机制生成的，那么我们此处不需要再显示的给ID赋值 -->  INSERT INTO USER (username,sex,birthday,address)  VALUES(#{username},#{sex},#{birthday},#{address})  </insert> |

#### 主键返回之MySQL自增UUID

|  |
| --- |
| <insert id="insertUser" parameterType="com.gyf.domain.User">  <selectKey keyProperty="id" resultType="String" order="BEFORE">  SELECT UUID()  </selectKey>  INSERT INTO USER (username,sex,birthday,address)  VALUES(#{username},#{sex},#{birthday},#{address})  </insert> |

#### ORCLE主键

SELECT user\_seq.nextval() FROM dual

#### 小结：

|  |
| --- |
| **parameterType和resultType**  parameterType指定输入参数的java类型，可以填写别名或Java类的全限定名。  resultType指定输出结果的java类型，可以填写别名或Java类的全限定名。  **#{}和${}**  #{}：相当于预处理中的占位符？。  #{}里面的参数表示接收java输入参数的名称。  #{}可以接受HashMap、POJO类型的参数。  当接受简单类型的参数时，#{}里面可以是value，也可以是其他。  #{}可以防止SQL注入。  ${}：相当于拼接SQL串，对传入的值不做任何解释的原样输出。  ${}会引起SQL注入，所以要谨慎使用。  ${}可以接受HashMap、POJO类型的参数。  当接受简单类型的参数时，${}里面只能是value。  **selectOne和selectList**  selectOne：只能查询0或1条记录，大于1条记录的话，会报错：  selectList：可以查询0或N条记录 |

### 2.4 MyBatis的Dao编写 【一般不用，有更多好方式】

#### dao

|  |
| --- |
|  |
|  |

#### 测试

|  |
| --- |
|  |

### MyBatis的Dao编写【mapper代理方式实现】

Mapper代理的开发方式，程序员只需要编写mapper接口（相当于dao接口）即可。Mybatis会自动的为mapper接口生成动态代理实现类。

**不过要实现mapper代理的开发方式，需要遵循一些开发规范。**

#### 开发规范

|  |
| --- |
| 1. mapper接口的全限定名要和mapper映射文件的namespace的值相同。 2. mapper接口的方法名称要和mapper映射文件中的statement的id相同； 3. mapper接口的方法参数只能有一个，且类型要和mapper映射文件中statement的parameterType的值保持一致。 4. mapper接口的返回值类型要和mapper映射文件中statement的resultType值或resultMap中的type值保持一致；   通过规范式的开发mapper接口，可以解决原始dao开发当中存在的问题：   * 模板代码已经去掉； * 剩下去不掉的操作数据库的代码，其实就是一行代码。这行代码中硬编码的部分，通过第一和第二个规范就可以解决。 |

#### 编写步骤

#### 第一步：【】

重新写个UserMapper配置文件和定义mapper映射文件UserMapper.xml（内容同Users.xml，**除了namespace的值**），放到新创建的目录mapper下。

|  |
| --- |
|  |

#### 第二步：【添加映射配置文件】

|  |
| --- |
|  |

#### 第三步：测试

|  |
| --- |
|  |

## 全局配置文件其它配置

### properties数据库文件配置

#### 在src下配置个db.properties文件

|  |
| --- |
|  |

#### 修改全局的配置文件

|  |
| --- |
|  |

### 3.2 setting【了解】

|  |
| --- |
|  |

#### 具体配置详解

|  |
| --- |
|  |

### 3.3 typeAliases

别名是使用是为了在映射文件中，更方便的去指定参数和结果集的类型，不再用写很长的一段全限定名。

#### mybatis支持的别名

|  |  |  |
| --- | --- | --- |
| 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 |

#### 自定义别名

|  |
| --- |
|  |

### mappers

#### <mapper resource=’’/>

使用相对于类路径的资源

如：<mapper resource="sqlmap/User.xml" />

#### <mapper url=’’/> 【不用】

使用完全限定路径

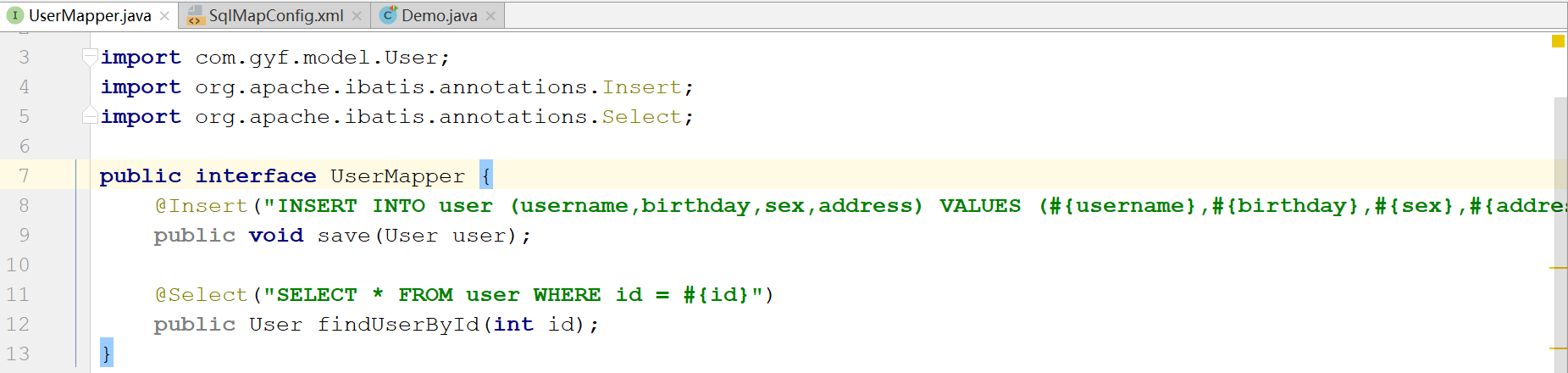
如：<mapper url="file:///D:\workspace\_spingmvc\mybatis\_01\config\sqlmap\User.xml" />

#### <mapper class=’’/>

使用mapper接口的全限定名

如：<mapper class="cn.gyf.mybatis.mapper.UserMapper"/>

也可使用注解开发，把xml文件删除



**注意：此种方法要求mapper接口和mapper映射文件要名称相同，且放到同一个目录下**；

#### <package name=’’/>（推荐）

注册指定包下的所有映射文件

如：<package name="cn.gyf.mybatis.mapper"/>

**注意：此种方法要求mapper接口和mapper映射文件要名称相同，且放到同一个目录下**；

## Mybatis的映射文件

### 输入映射ParameterType

指定输入参数的java类型，可以使用别名或者类的全限定名。它可以接收**简单类型,POJO对象、HashMap**。

#### 传递简单类型

根据用户ID查询用户信息。

|  |
| --- |
|  |

#### 传递POJO对象

|  |
| --- |
|  |

#### 传递POJO包装对象

开发中通过pojo传递查询条件 ，查询条件是综合的查询条件，不仅包括用户查询条件还包括其它的查询条件（比如将用户购买商品信息也作为查询条件），这时可以使用包装对象传递输入参数。

**需求**

综合查询用户信息，需要传入查询条件复杂，比如（用户信息、订单信息、商品信息）。

vo:键值对对象，相对于kv

po:persist object 持久化对象

pojo:简单的java对象

entity:实体

##### 定义POJO包装类

|  |
| --- |
|  |

##### 修改UserMapper.java

|  |
| --- |
|  |

##### 修改UsrMappler.xml

|  |
| --- |
|  |

##### 测试

|  |
| --- |
|  |

#### 传递Map对象

|  |
| --- |
| 修改UserMapper.java |
|  |
| 修改UserMapper.xml |
|  |
| 测试 |
|  |

### 4.2 输出映射 resultType/resultMap

#### resultType

使用resultType进行结果映射时，查询的列名和映射的pojo属性名完全一致，该列才能映射成功。

如果查询的列名和映射的pojo属性名全部不一致，则不会创建pojo对象；

如果查询的列名和映射的pojo属性名有一个一致，就会创建pojo对象。

##### 输出简单类型

当输出结果只有一列时，可以使用ResultType指定简单类型作为输出结果类型。

**案例：输出一个count(\*)**

|  |
| --- |
| **UserMapper.java** |
|  |
| **UserMapper.xml** |
|  |
| 测试 |
|  |

##### 输出POJO单个对象

|  |
| --- |
|  |
|  |

##### 输出POJO列表

|  |
| --- |
|  |
|  |

**总结：**

输出单个pojo对象和pojo列表时，mapper映射文件中的resultType的类型是一样的，mapper接口的方法返回值不同。

同样的mapper映射文件，返回单个对象和对象列表时，mapper接口在生成动态代理的时候，会根据返回值的类型，决定调用selectOne方法还是selectList方法。

#### resultMap

如果查询出来的列名和属性名不一致，通过定义一个**resultMap将列名和pojo属性名**之间作一个映射关系。

1. 定义resultMap

2、使用resultMap作为statement的输出映射类型

|  |
| --- |
| UserMapper.java |
|  |
| UserMapper.xml |
|  |
| 测试 |
|  |

## 动态SQL

### 5.1 if和where

* If标签：作为判断入参来使用的，如果符合条件，则把if标签体内的SQL拼接上。

**注意：用if进行判断是否为空时，不仅要判断null，也要判断空字符串‘’；**

* Where标签：会去掉条件中的第一个and符号。

|  |
| --- |
|  |

### SQL片断

Mybatis提供了SQL片段的功能，可以提高SQL的**可重用性**。

|  |
| --- |
|  |

### foreach 遍历

#### 案例：查询指定id的用户

【SELECT \* FROM `user` where id in (31,32,33); 】

|  |
| --- |
| UserQueryVO.java |
|  |
| UserMapper.xml |
|  |
| 测试： |
|  |

#### 另一种遍历，参数直接传入id集合【自己练】

|  |
| --- |
|  |

## 关联查询

### 案例：用户和定单

|  |
| --- |
|  |

|  |
| --- |
| * **user和orders：**   User 与orders：一个用户可以创建多个订单，一对多  Orders 与 user：多个订单只由一个用户创建，多对一   * **orders和orderdetail：**   Orders 与 orderdetail：一个订单可以包括 多个订单明细，因为一个订单可以购买多个商品，每个商品的购买信息在orderdetail记录，一对多关系  orderdetail 与orders：多个订单明细包括在一个订单中, 多对一   * **orderdetail和items：**   Orderdetail 与 items：多个订单明细只对应一个商品信息，多对一  Items 与 orderdetail:一个商品可以包括在多个订单明细 ，一对多 |

### 需求：

**根据商品ID查找定单信息，包括用户名和地址**

|  |
| --- |
| #查找id为10的所有定单  SELECT  orders.id, orders.number,orders.createtime,orders.note,`user`.username,`user`.address  FROM  orders ,`user`  WHERE  orders.user\_id = `user`.id AND `user`.id = 10; |

### 一对一 resultType实现

复杂查询时，单表对应的po类已不能满足输出结果集的映射。所以要根据需求建立一个**扩展类来作为resultType的类型**。

|  |
| --- |
| #查找某个定单id的信息，包括用户名字和地址  SELECT o.\*,u.username,u.address FROM orders o,user u  WHERE o.user\_id = u.id AND o.id = 3 |

#### 第一步：写个定单的扩展类

|  |
| --- |
|  |

#### 第二步：声明定单接口

|  |
| --- |
|  |

#### 第三步：声明定单配置文件

|  |
| --- |
|  |

#### 第四步：加载映射文件

|  |
| --- |
|  |

#### 第五步：测试

|  |
| --- |
|  |

#### 一对一 resultMap实现

**掌握association的使用**

|  |
| --- |
| OrdersMapper.java添加一个方法 |
|  |
| OrdersMapper.xml |
|  |
| 测试 |
|  |

#### 总结

**resultType**：使用resultType实现较为简单，如果pojo中没有包括查询出来的列名，需要增加列名对应的属性，即可完成映射。**如果没有查询结果的特殊要求建议使用resultType。**

**resultMap**：需要单独定义resultMap，实现有点麻烦，如果对查询结果有特殊的要求，使用resultMap可以完成将关联查询映射pojo的对象属性中。

**resultMap可以实现延迟加载，resultType无法实现延迟加载。**

### 一对多

#### 需求：

根据定单ID查找定单信息、用户信息和定单明细信息

|  |
| --- |
| Select  orders.id,  orders.user\_id,  orders.number,  orders.createtime,  orders.note,  user.username,  user.address,  orderdetail.id detail\_id,  orderdetail.items\_id,  orderdetail.items\_num  from  orders,user,orderdetail  where  orders.user\_id = user.id  and orders.id = orderdetail.orders\_id  and orders.id = #{?}; |
| SELECT  o.\*,  u.username,  u.address,  od.id detail\_id,  od.items\_id,  od.items\_num  FROM  orders o,  user u,  orderdetail od  WHERE  o.user\_id = u.id  AND o.id = od.orders\_id  AND o.id = 3 |

目标：掌握collection的使用

#### 第一步：在Orders中添加定单明细

|  |
| --- |
|  |

#### 第二步：Mapper接口

|  |
| --- |
|  |

#### 第三步：OrderMapper.xml

resultMap中有个extends属性，可以继承【自行研究】

|  |
| --- |
|  |

#### 第四步：测试

|  |
| --- |
|  |

#### 总结

mybatis使用resultMap的collection对关联查询的多条记录映射到一个list集合属性中。

**使用resultType实现：**

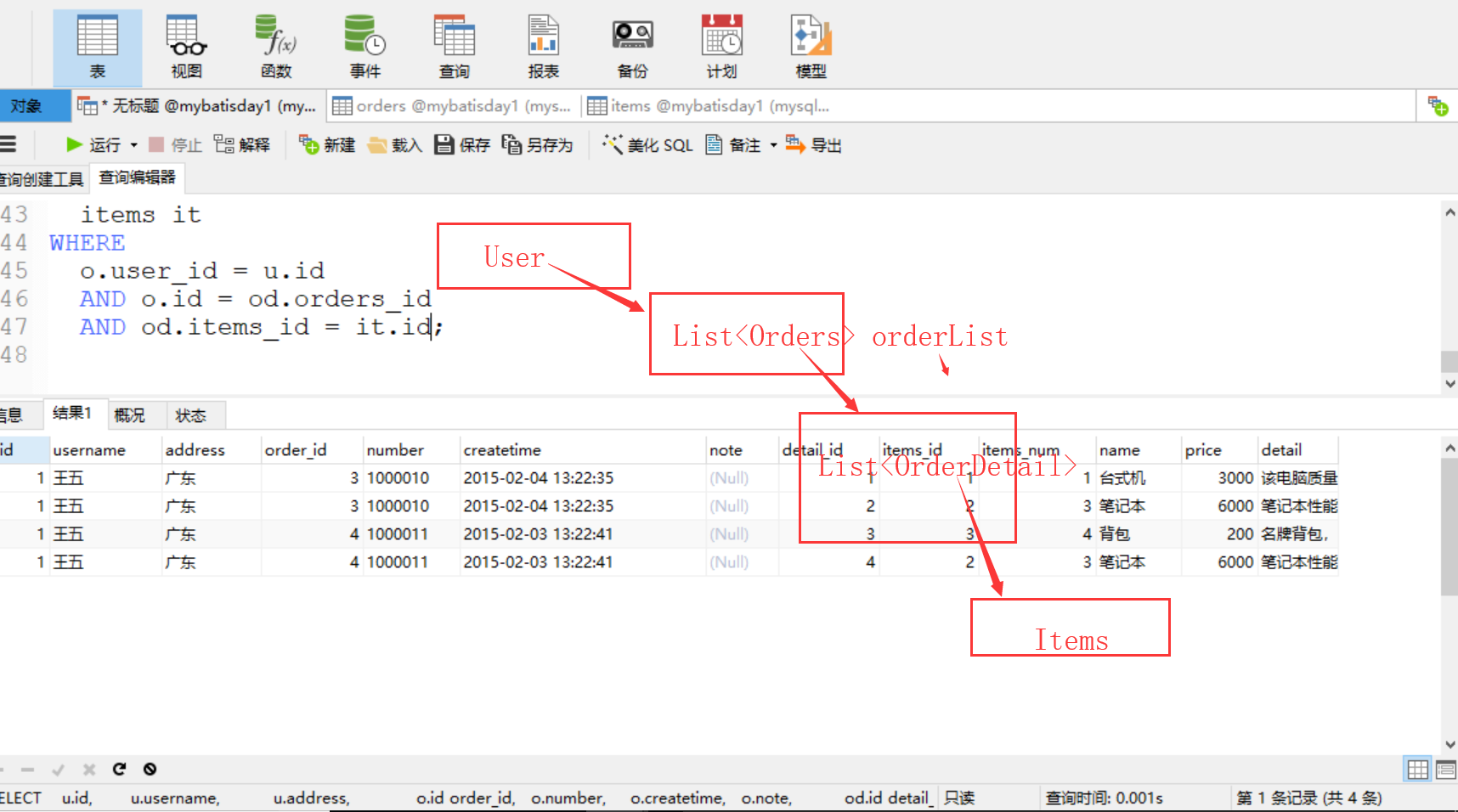
需要对结果集进行二次处理。

将订单明细映射到orders中的orderdetails中，需要自己处理，使用双重循环遍历，去掉重复记录，将订单明细放在orderdetails中。

### 多对多

#### 需求

查询用户信息及用户购买的商品信息，**要求将关联信息映射到主pojo的pojo属性中**



#### Sql

|  |
| --- |
| Select  user.id,  user.username,  user.address,  orders.id orders\_id,  orders.user\_id,  orders.number,  orders.createtime,  orders.note,  orderdetail.id detail\_id,  orderdetail.items\_id,  orderdetail.items\_num,  items.name items\_name,  items.detail items\_detail  FROM  USER,orders,orderdetail,items  WHERE user.`id` = orders.`user\_id`  AND orders.`id` = orderdetail.`orders\_id`  AND orderdetail.`items\_id` = items.`id` |
| SELECT  u.id,  u.username,  u.address,  o.id order\_id,  o.number,  o.createtime,  o.note,  od.id detail\_id,  od.items\_id,  od.items\_num,  it.name,  it.price,  it.detail  FROM  user u,  orders o,  orderdetail od,  items it  WHERE  o.user\_id = u.id  AND o.id = od.orders\_id  AND od.items\_id = it.id; |

#### 映射思路

* 将用户信息映射到user中。
* 在user类中添加订单列表属性List<Orders> orderslist，将用户创建的订单映射到orderslist
* 在Orders中添加订单明细列表属性List<Orderdetail> detailList，将订单的明细映射到detailList
* 在Orderdetail中添加Items属性，将订单明细所对应的商品映射到Items

#### 第一步：UserMapper.java

|  |
| --- |
|  |

#### 第二步：User/Orders/Orderdetail.java

|  |
| --- |
|  |
|  |
|  |

#### 第三步：UserMapper.xml

|  |
| --- |
|  |

#### 第四步：测试

|  |
| --- |
|  |
| 打印效果 |

#### 总结

resultType：将查询结果按照sql列名pojo属性名一致性映射到pojo中。

resultMap：使用association和collection完成一对一和一对多高级映射（对结果有特殊的映射要求）。

association：将关联查询信息映射到一个pojo对象中。

collection：将关联查询信息映射到一个list集合中。

## 延时加载

### 延迟加载

延迟加载又叫懒加载，也叫按需加载。也就是说先加载主信息，在需要的时候，再去加载从信息。

在mybatis中，resultMap标签 的association标签和collection标签具有延迟加载的功能。

### 案例：

#### Mapper.java

|  |
| --- |
|  |
|  |
|  |

#### Mapper.xml

|  |
| --- |
| UserMappler.xml |
|  |
| OrdersMapper.xml |
|  |

#### 测试

|  |
| --- |
|  |

#### 配置懒加载

|  |
| --- |
|  |

## 查询缓存

### Mybatis的缓存理解

Mybatis的缓存，包括一级缓存和二级缓存，一级缓存是默认使用的。二级缓存需要手动开启。

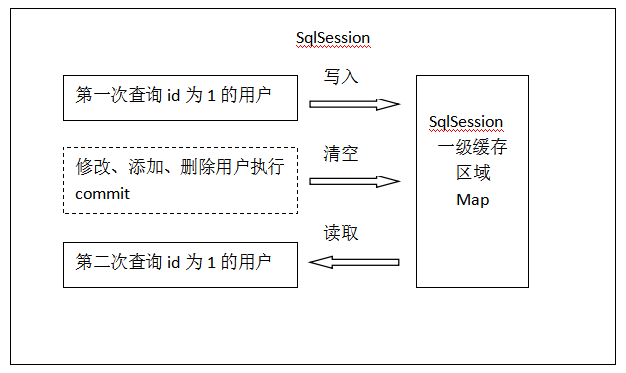
一级缓存指的就是sqlsession，在sqlsession中有一个数据区域，是map结构，这个区域就是一级缓存区域。一级缓存中的key是由sql语句、条件、statement等信息组成一个唯一值。一级缓存中的value，就是查询出的结果对象。

二级缓存指的就是同一个namespace下的mapper，二级缓存中，也有一个map结构，这个区域就是一级缓存区域。一级缓存中的key是由sql语句、条件、statement等信息组成一个唯一值。一级缓存中的value，就是查询出的结果对象。

|  |
| --- |
|  |

### 一级缓存

#### 原理：



#### 测试1

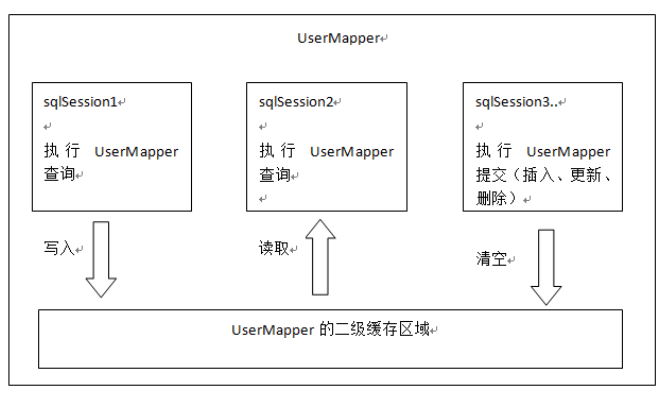
|  |
| --- |
|  |

#### 测试2

|  |
| --- |
|  |

### 二级缓存

#### 原理：



#### 使用：

##### 开启二级缓存总开关

|  |
| --- |
|  |

##### UserMapper中配置二级缓存

|  |
| --- |
|  |

##### User系列化

|  |
| --- |
|  |

##### 测试

|  |
| --- |
|  |

##### 禁用指定方法二级缓存

|  |
| --- |
|  |

##### 刷新缓存

|  |
| --- |
|  |

### 3.4 整合ehcache

Mybatis本身是一个持久层框架，它不是专门的缓存框架，所以它对缓存的实现不够好，不能支持分布式。

Ehcache是一个分布式的缓存框架。

##### 什么是分布式

系统为了提高性能，通常会对系统采用分布式部署（集群部署方式）

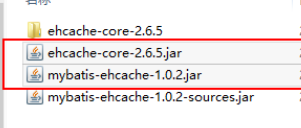
|  |
| --- |
|  |

##### 整合思路

Cache是一个接口，它的默认实现是mybatis的PerpetualCache。如果想整合mybatis的二级缓存，那么实现Cache接口即可。

|  |
| --- |
|  |

##### 添加jar包



##### 设置映射文件中cache标签

的type值为ehcache的实现类

|  |
| --- |
|  |

##### 在src下添加ehcache的配置文件

|  |
| --- |
|  |
| * maxElementsInMemory :设置基于内存的缓存中可存放的对象最大数目 * eternal:设置对象是否为永久的,true表示永不过期,此时将忽略 * timeToIdleSeconds 和 timeToLiveSeconds属性; 默认值是false * timeToIdleSeconds:设置对象空闲最长时间,以秒为单位, 超过这个时间,对象过期。当对象过期时,EHCache会把它从缓存中清除。如果此值为0,表示对象可以无限期地处于空闲状态。 * timeToLiveSeconds:设置对象生存最长时间,超过这个时间,对象过期。如果此值为0,表示对象可以无限期地存在于缓存中. 该属性值必须大于或等于 timeToIdleSeconds 属性值 * overflowToDisk:设置基于内在的缓存中的对象数目达到上限后,是否把溢出的对象写到基于硬盘的缓存中 * diskPersistent 当jvm结束时是否持久化对象 true false 默认是false * diskExpiryThreadIntervalSeconds 指定专门用于清除过期对象的监听线程的轮询时间   memoryStoreEvictionPolicy - 当内存缓存达到最大，有新的element加入的时候， 移除缓存中element的策略。**默认是LRU（最近最少使用），可选的有LFU（最不常使用）和FIFO（先进先出）** |

##### 测试：用上面二级缓存例子即可

##### 二级缓存应用场景

使用场景：**对于访问响应速度要求高，但是实时性不高的查询，可以采用二级缓存技术**。

注意：在使用二级缓存的时候，要设置一下刷新间隔（cache标签中有一个flashInterval属性）来定时刷新二级缓存，这个刷新间隔根据具体需求来设置，比如设置30分钟、60分钟等，单位为毫秒。

##### 局限性

**Mybatis二级缓存对细粒度的数据，缓存实现不好。**

场景：

对商品信息进行缓存，由于商品信息查询访问量大，但是要求用户每次查询都是最新的商品信息，此时如果使用二级缓存，就无法实现当一个商品发生变化只刷新该商品的缓存信息而不刷新其他商品缓存信息，因为二级缓存是mapper级别的，当一个商品的信息发送更新，所有的商品信息缓存数据都会清空。

解决此类问题，需要在业务层根据需要对数据有针对性的缓存。

比如可以对经常变化的 数据操作单独放到另一个namespace的mapper中。

## mybaties整合spring

SSH:struts2+spring+hibernate

SSM:SpringMVC + Spring + MyBatis

Spring 3.2

Mybaties 3.2.7

### 4.1 创建工程导包

#### 导入mybaties包

|  |
| --- |
| mybaties核心包 |
| mybatis依赖包 |

#### 导入mysql数据库驱动

|  |
| --- |
|  |

#### 数据库dbcp连接池

|  |
| --- |
|  |

#### 导入spring+mvc包

|  |
| --- |
|  |

#### Mybatis-spring整合包

|  |
| --- |
|  |

### 4.2 配置mybatis的核心配置文件

核心配置文件、创建User模型、映射文件，

|  |
| --- |
|  |
|  |

### 4.3 spring的数据源

|  |
| --- |
| <beans xmlns="http://www.springframework.org/schema/beans"  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:mvc="http://www.springframework.org/schema/mvc"  xmlns:context="http://www.springframework.org/schema/context"  xmlns:aop="http://www.springframework.org/schema/aop" xmlns:tx="http://www.springframework.org/schema/tx"  xsi:schemaLocation="http://www.springframework.org/schema/beans  http://www.springframework.org/schema/beans/spring-beans-3.2.xsd  http://www.springframework.org/schema/mvc  http://www.springframework.org/schema/mvc/spring-mvc-3.2.xsd  http://www.springframework.org/schema/context  http://www.springframework.org/schema/context/spring-context-3.2.xsd  http://www.springframework.org/schema/aop  http://www.springframework.org/schema/aop/spring-aop-3.2.xsd  http://www.springframework.org/schema/tx  http://www.springframework.org/schema/tx/spring-tx-3.2.xsd ">      <!-- 1.配置数据库，dbcp数据库连接池 -->  <bean id="datasourse" class="org.apache.commons.dbcp.BasicDataSource">  <property name="driverClassName" value="xxx"/>  <property name="url" value="xxx"/>  <property name="username" value="xxx/>  <property name="password" value="xxx"/>  <!-- 最大连接 -->  <property name="maxActive" value="10"/>  <!--最大空闲数 -->  <property name="maxIdle" value="5"/>  </bean>  </beans> |

### spring配置SqlSessionFactory

|  |
| --- |
| <!-- 3.配置SqlSessionFactory -->  <bean id=*"sqlSessionFactoryBean"* class=*"org.mybatis.spring.SqlSessionFactoryBean"*>  <property name=*"configLocation"* value=*"SqlMapConfig.xml"*></property>  <property name=*"dataSource"* ref=*"dataSourse"*></property>  </bean> |

### 编写个UserDaoImpl,接口省略

|  |
| --- |
|  |

### spring中配置daobean

|  |
| --- |
|  |

### 4.7测试

|  |
| --- |
|  |

### 4.8 换成Mapper接口整合dao

#### 创建Mapper映射文件

|  |
| --- |
| UserMapper |
|  |
| UserMapper.xml |
|  |

#### 核心配置文件加载映射文件

|  |
| --- |
|  |

#### Spring配置MapperFactoryBean

使用工厂Bean生成userMapper对象

|  |
| --- |
| <!-- 4.配置 -->  <bean id = *"userMapper"* class=*"org.mybatis.spring.mapper.MapperFactoryBean"*>  <property name=*"mapperInterface"* value=*"com.gyf.backoffice.mapper.UserMapper"*/>  <property name=*"sqlSessionFactory"* ref=*"sqlSessionFactoryBean"*/>  </bean> |

#### 测试

|  |
| --- |
|  |

#### 用MapperScannerConfigurer批量扫描创建代理对象

【上面的代码麻烦，每一个mappler就创建一个工厂bean】

|  |
| --- |
| <!-- mapper代理开发方式之批量mapper配置 ,bean的名字默认为mapper接口类名的首字母小写  注意：  1.jdk1.8 用这种方式，bean不能创建成功 ，改成jdk1.7的即可  2.或者spring我换成spring3.2.9或以上就OK了  -->  <bean class=*"org.mybatis.spring.mapper.MapperScannerConfigurer"*>  <property name=*"basePackage"* value=*"com.gyf.backoffice.mapper"*/>  <property name=*"sqlSessionFactoryBeanName"* value=*"sqlSessionFactoryBean"*/>  </bean> |

测试与上一个测试一样

# 秒杀系统

## test

### 1、环境搭建

|  |
| --- |
| <dependency>  <groupId>junit</groupId>  <artifactId>junit</artifactId>  <version>4.12</version>  <scope>provided</scope>  </dependency>    <dependency>  <groupId>log4j</groupId>  <artifactId>log4j</artifactId>  <version>1.2.17</version>  </dependency>    <dependency>  <groupId>org.mybatis</groupId>  <artifactId>mybatis</artifactId>  <version>3.1.1</version>  </dependency>  <dependency>  <groupId>mysql</groupId>  <artifactId>mysql-connector-java</artifactId>  <version>5.1.38</version>  </dependency> |

MyBatis传统的方式写mybatis的应用程序

### 2、添加实体

public class {

private int id;

private String name; // 姓名

private String sex; // 性别

private Date birthday; // 生日

private String address; // 地址

}

### 3、添加映射文件person.xml

上课代码

|  |
| --- |
| <?xml version="1.0" encoding="UTF-8" ?>  <!DOCTYPE mapper  PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"  "http://mybatis.org/dtd/mybatis-3-mapper.dtd">  <mapper namespace="com.yaosiyuan.mapper.PersonMapper">  <sql id="consutomsql">  <if test="customPerson !=null">  <if test="customPerson.name != null and customPerson.name !=''">  and name = #{customPerson.name}  </if>  <if test="customPerson.birthday !=null and customPerson.birthday !=''">  and birthday = #{customPerson.birthday}  </if>  </if>  </sql>  <select id = "queryPersonByVo" parameterType="PersonVo" resultType="Person">  select \* from Person  <where>  <include refid="consutomsql"></include>  </where>  </select>  <select id="querypersonbyid" parameterType="int" resultType="Person">  select id,name,age,birthday,address from Person where id= #{value}  </select>      <!-- 模糊查询 like "%${value}%" -->  <select id="querypersonbyName" parameterType="java.lang.String" resultType="Person">  select id,name,age,birthday,address from Person where name like "%${value}%"  </select>      <insert id="insertPerson" parameterType="Person">  <selectKey keyProperty="id" order="AFTER" resultType="java.lang.Integer">  SELECT LAST\_INSERT\_ID()  </selectKey>  insert into Person(name,age,address,birthday) value(#{name},#{age},#{address},#{birthday})  </insert>    <delete id="deletePersonById" parameterType="java.lang.Integer">  <!-- 删除 用户 根据id删除用户，需要输入 id值 -->  delete from person where  id=#{id}  </delete>    <delete id ="upadtePersonById" parameterType="Person">  update Person set name=#{name},age=#{age},birthday=#{birthday},address=#{address} where id = #{id}  </delete>    </mapper> |

备课代码

|  |
| --- |
| <!-- 在 映射文件中配置很多sql语句 -->  <!-- 将sql语句封装到mappedStatement对象中，所以将id称为statement的id -->  <!-- parameterType：指定输入 参数的类型，这里指定int型 #{}表示一个占位符号 -->  <!-- #{id}：其中的id表示接收输入 的参数，参数名称就是id，如果输入 -->  <!-- 参数是简单类型，#{}中的参数名可以任意，可以value或其它名称 -->  <!-- resultType：指定sql输出结果 的所映射的java对象类型，select指定resultType表示将单条记录映射成的java对象。 -->  <select id="findPersonById" parameterType="int"  resultType="person">  <!-- 需求：通过select执行id查询用户表的记录 -->  SELECT \* FROM Person WHERE id=#{value}  </select>    <!-- resultType：指定就是单条记录所映射的java对象类型 -->  <!-- ${}:表示拼接sql串，将接收到参数的内容不加任何修饰拼接在sql中。 使用${}拼接sql，引起 sql注入 -->  <!-- ${value}：接收输入 参数的内容，如果传入类型是简单类型，${}中只能使用value -->  <!-- 这个mapper文件person对象已经用别名代替 -->  <select id="findPersonByName" parameterType="java.lang.String"  resultType="person">  <!-- 根据用户名称模糊查询用户信息，可能返回多条 -->  SELECT \* FROM Person WHERE name LIKE '%${value}%'  </select>      <!-- parameterType：指定输入 参数类型是pojo（包括 用户信息）-->  <!-- #{}中指定pojo的属性名，接收到pojo对象的属性值，mybatis通过OGNL获取对象的属性值 -->  <insert id="insertPerson" parameterType="person">  <!-- SELECT LAST\_INSERT\_ID()：得到刚insert进去记录的主键值，只适用与自增主键 -->  <!-- keyProperty：将查询到主键值设置到parameterType指定的对象的哪个属性 -->  <!-- order：SELECT LAST\_INSERT\_ID()执行顺序，相对于insert语句来说它的执行顺序 -->  <!-- resultType：指定SELECT LAST\_INSERT\_ID()的结果类型 -->  <selectKey keyProperty="id" order="AFTER" resultType="java.lang.Integer">  <!-- 将插入数据的主键返回，返回到person对象中 -->  SELECT LAST\_INSERT\_ID()  </selectKey>  insert into person(name,birthday,sex,address)  value(#{name},#{birthday},#{sex},#{address})  <!-- 使用mysql的uuid（）生成主键 执行过程： -->  <!-- 首先通过uuid()得到主键，将主键设置到pseron对象的id属性中 其次在insert执行时，从person对象中取出id属性值 -->  <!-- <selectKey keyProperty="id" order="BEFORE" resultType="java.lang.String"> -->  <!-- SELECT uuid() </selectKey> insert into person(id,name,birthday,sex,address) -->  <!-- value(#{id},#{name},#{birthday},#{sex},#{address}) -->  </insert>      <delete id="deletePerson" parameterType="java.lang.Integer">  <!-- 删除 用户 根据id删除用户，需要输入 id值 -->  delete from person where  id=#{id}  </delete>      <!-- 需要传入用户的更新信息 parameterType指定person对象，包括 id和更新信息， -->  <!-- 注意：id必须存在 -->  <!-- #{id}：从输入 person对象中获取id属性值 -->  <update id="person" parameterType="person">  <!-- 根据id更新用户 分析： 需要传入用户的id -->  update person set  name=#{name},birthday=#{birthday},sex=#{sex},address=#{address}  where id=#{id}  </update>    </mapper> |

### 4、添加SqlMappingConfig.xml文件

|  |
| --- |
| <?xml version="1.0" encoding="UTF-8" ?>  <!DOCTYPE configuration  PUBLIC "-//mybatis.org//DTD Config 3.0//EN"  "http://mybatis.org/dtd/mybatis-3-config.dtd">  <configuration>    <!-- 加载属性文件 -->  <properties resource="db.properties">  <!--properties中还可以配置一些属性名和属性值 -->  <!-- <property name="jdbc.driver" value=""/> -->  </properties>    <typeAliases>  <!-- 别名定义 -->  <typeAlias type="cn.youfan.pojo.Person" alias="Person" />  </typeAliases>    <environments default="development">  <environment id="development">  <!-- 使用jdbc事务管理，事务控制由mybatis -->  <transactionManager type="JDBC" />  <!-- 数据库连接池，由mybatis管理 -->  <dataSource type="POOLED">  <property name="driver" value="${jdbc.driver}" />  <property name="url" value="${jdbc.url}" />  <property name="username" value="${jdbc.username}" />  <property name="password" value="${jdbc.password}" />  </dataSource>  </environment>  </environments>      <!-- 加载 映射文件 -->  <mappers>      <!--通过resource方法一次加载一个映射文件 -->  <!--注意这里的路径和xml文件 -->  <mapper resource="Person.xml" />    </mappers>    </configuration> |

### 5、数据库创建表

### 6、编写测试类进行测试

上课代码

|  |
| --- |
| /\*\*  \* @description: 功能描述：()  \* @copyright: Copyright (c) 2019  \* @company: 昭阳科技  \* @author:  \* @version: 2.0  \* @date: 2019 2019年5月2日 下午6:07:47  \*/  package com.yaosiyuan.entity;  import java.io.IOException;  import java.io.InputStream;  import java.util.List;  import org.apache.ibatis.io.Resources;  import org.apache.ibatis.session.SqlSession;  import org.apache.ibatis.session.SqlSessionFactory;  import org.apache.ibatis.session.SqlSessionFactoryBuilder;  import org.junit.Test;  import com.yaosiyuan.mapper.PersonMapper;  import com.yaosiyuan.paran.CustomPerson;  import com.yaosiyuan.vo.PersonVo;  /\*\*  \* @description: 功能描述 ()  \* @copyright: Copyright (c) 2019  \* @company: 昭阳科技  \* @author:  \* @version: 2.0  \* @date: 2019 2019年5月2日 下午6:07:47  \*/  public class TestMybatis {  public SqlSessionFactory getFactory() throws IOException {  String filePath = "SqlMappingConfig.xml";  InputStream resourceAsStream = Resources.getResourceAsStream(filePath);  SqlSessionFactory sqlSessionFactory = new SqlSessionFactoryBuilder().build(resourceAsStream);  return sqlSessionFactory;  }  @Test  public void testInsert() throws IOException {  SqlSessionFactory factory = this.getFactory();  SqlSession openSession = factory.openSession();  PersonMapper personMapper = openSession.getMapper(PersonMapper.class);  Person person = new Person();  person.setName("张三");  person.setBirthday("11-11");  person.setAge(12);  person.setAddress("河南郑州");  personMapper.insertPerson(person);  // openSession.insert("test1.insertPerson", person);  System.out.println("id:" + person.getId());  openSession.commit();  openSession.close();  }  @Test  public void testQueryUserById() throws IOException {  SqlSessionFactory factory = this.getFactory();  SqlSession openSession = factory.openSession();  PersonMapper personMapper = openSession.getMapper(PersonMapper.class);  Person person = personMapper.querypersonbyid(1);  // Person person = openSession.selectOne("querypersonbyid", 1);  System.out.println(person);  }  @Test  public void testQueryUserByName() throws IOException {  SqlSessionFactory factory = this.getFactory();  SqlSession openSession = factory.openSession();  PersonMapper personMapper = openSession.getMapper(PersonMapper.class);  List<Person> personList = personMapper.querypersonbyName("张");  // List<Person> personList = openSession.selectList("querypersonbyName",  // "姚");  for (int i = 0; i < personList.size(); i++) {  System.out.println(personList.get(i));  }  openSession.commit();  openSession.close();  }  @Test  public void testdeletePersonById() throws IOException {  SqlSessionFactory factory = this.getFactory();  SqlSession openSession = factory.openSession();  PersonMapper personMapper = openSession.getMapper(PersonMapper.class);  personMapper.deletePersonById(3);  // openSession.delete("deletePersonById", 2);  openSession.commit();  openSession.close();  }  @Test  public void testupadtePersonById() throws IOException {  SqlSessionFactory factory = this.getFactory();  SqlSession openSession = factory.openSession();  PersonMapper personMapper = openSession.getMapper(PersonMapper.class);  Person person = new Person();  person.setId(1);  person.setName("1111张三111");  person.setBirthday("1112-12");  person.setAge(12);  person.setAddress("1111");  personMapper.upadtePersonById(person);  // openSession.update("upadtePersonById", person);  openSession.commit();  openSession.close();  }  @Test  public void queryPersonByVo() throws IOException {  SqlSessionFactory factory = this.getFactory();  SqlSession openSession = factory.openSession();  PersonMapper personMapper = openSession.getMapper(PersonMapper.class);  PersonVo personVo = new PersonVo();  CustomPerson customPerson = new CustomPerson();  customPerson.setName("张三");  customPerson.setBirthday("11-11");  personVo.setCustomPerson(customPerson);  List<Person> personList = personMapper.queryPersonByVo(personVo);  for (int i = 0; i < personList.size(); i++) {  System.out.println(personList.get(i));  }  openSession.commit();  openSession.close();  }  } |

备课代码

|  |
| --- |
| public class MybatisFirst {    public SqlSessionFactory getSqlSessionFactory() throws IOException {  // mybatis配置文件  String resource = "config/SqlMapConfig.xml";  // 得到配置文件流  InputStream inputStream = Resources.getResourceAsStream(resource);  // 创建会话工厂，传入mybatis的配置文件信息  SqlSessionFactory sqlSessionFactory = new SqlSessionFactoryBuilder().build(inputStream);  return sqlSessionFactory;  }        // 根据id查询用户信息，得到一条记录结果  @Test  public void findPersonByIdTest() throws IOException {    // 通过工厂得到SqlSession  SqlSession sqlSession = this.getSqlSessionFactory().openSession();    // 通过SqlSession操作数据库  // 第一个参数：映射文件中statement的id，等于=namespace+"."+statement的id  // 第二个参数：指定和映射文件中所匹配的parameterType类型的参数  // sqlSession.selectOne结果 是与映射文件中所匹配的resultType类型的对象  // selectOne查询出一条记录  Person Person = sqlSession.selectOne("test.findPersonById", 1);    System.out.println(Person);    // 释放资源  sqlSession.close();    }    // 根据用户名称模糊查询用户列表  @Test  public void findPersonByNameTest() throws IOException {    // 通过工厂得到SqlSession  SqlSession sqlSession = this.getSqlSessionFactory().openSession();  // list中的Person和映射文件中resultType所指定的类型一致  List<Person> list = sqlSession.selectList("test.findPersonByName", "小明");  System.out.println("信息：" + list);  sqlSession.close();    }    // 添加用户信息  @Test  public void insertPersonTest() throws IOException {    // 通过工厂得到SqlSession  SqlSession sqlSession = this.getSqlSessionFactory().openSession();  // 插入用户对象  Person Person = new Person();  Person.setPersonname("王小军");  Person.setBirthday(new Date());  Person.setSex("1");  Person.setAddress("河南郑州");    sqlSession.insert("test.insertPerson", Person);    // 提交事务  sqlSession.commit();    // 获取用户信息主键  System.out.println(Person.getId());  // 关闭会话  sqlSession.close();    }    // 根据id删除 用户信息  @Test  public void deletePersonTest() throws IOException {    // 通过工厂得到SqlSession  SqlSession sqlSession = this.getSqlSessionFactory().openSession();    // 传入id删除 用户  sqlSession.delete("test.deletePerson", 49);    // 提交事务  sqlSession.commit();    // 关闭会话  sqlSession.close();    }    // 更新用户信息  @Test  public void updatePersonTest() throws IOException {    // 通过工厂得到SqlSession  SqlSession sqlSession = this.getSqlSessionFactory().openSession();  // 更新用户信息    Person Person = new Person();  // 必须设置id  Person.setId(41);  Person.setPersonname("王大军");  Person.setBirthday(new Date());  Person.setSex("2");  Person.setAddress("河南郑州");    sqlSession.update("test.updatePerson", Person);    // 提交事务  sqlSession.commit();    // 关闭会话  sqlSession.close();    } |