# 单独使用操作

## 第一步 导包

必须包

mybatis-3.4.6(核心包)、mysql-connector-java-5.1.4(mysql驱动包)

功能包(可有可无)

1、日志包(用于打印sql)：mybatis本身就无打印sql功能，但整合slf4j日志接口：

还需核心xml中配置：<setting name="logImpl" value="STDOUT\_LOGGING" />

log4j和slf4j融合需要3个jar：

slf4j-log4j12-1.7.25:log4j和slf4j中间包，因为log4j没有直接对接slf4j标准接口

slf4j-api-1.7.25：slf4j标准接口jar包

log4j-1.2.17:log4j核心包

留意：不用commons-logging-1.2,应该只有用了spring框架才需要

maven工程导入：

此maven一次性导入log4j的3个jar包

<dependency>

<groupId>org.slf4j</groupId>

<artifactId>slf4j-log4j12</artifactId>

<version>1.7.25</version>

</dependency>

## 第二步 核心xml

小知识：Eclipse引入本地DTD文件

windows –>preferences–>搜索xml–>xml catalog–> add–>fileSystem找到的DTD文件–>key选择:URI–>key:就是绝对地址–>保存

以下mybatis-config.xml内容 这边只是整合好以后复习：注意标签的顺序(有约束)

1）头部：SYSTEM本地 PUBLIC：网络

引入本地dtd文件头：<!DOCTYPE mapper SYSTEM "F:\学习资料\MyBatis\mybatis-3-mapper.dtd">

引入网路上的文件头：<!DOCTYPE configuration PUBLIC "-//mybatis.org//DTD Config 3.0//EN""http://mybatis.org/dtd/mybatis-3-config.dtd">

2）配置数据源

<!—1、 引入jdbc.properties读取driver，url，username，password的值 -->

<properties resource="jdbc.properties"/>

<!--2 environments 指定数据源环境(可以配置多个)，default指的是使用哪个数据源 -->

<environments default="development">

<environment id="development">

<!-- type="JDBC" 表示事务由jdbc连接管理，type="MANAGED"表示事务由容器来管理 -->

<transactionManager type="JDBC"/>

<!-- type="POOLED" mybatis的连接池，type="UNPOOLED"表示不使用连接池 jndi：基于tomcat的数据源 -->

<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>

<!-- 配置另一个数据源 -->

<environment id="test">

<transactionManager type="JDBC"/>

<dataSource type="POOLED">

<property name="driver" value="${jdbc-driver}"/>

<property name="url" value="${jdbc-driver}"/>

<property name="username" value="${jdbc-driver}"/>

<property name="password" value="${jdbc-driver}"/>

</dataSource>

</environment>

</environments>

3）sql映射文件xml扫描

<mappers>

<package name="com.wwl.dao"/>

</mappers>

4)功能的引用(如 控制打印sql语句、别名的配置、一级缓存的配置等)

<!-- 打印查询语句 -->

<setting name="logImpl" value="STDOUT\_LOGGING" />

## 第三步 接口和mapper

接口类和mapper.xml放在同一包下、相同的名字

接口类

public interface UserDao {

public int insertUser(User user); //添加操作，返回值int ; (在mapper.xml指定参数实体类类型)

public List<User> selectUser(); //查询操作，返回值List<User>

public int deleteUser(Integer id); //删除操作：基本数值类型，

public int updateUser(@Param("user")User user, @Param("id")Integer id); //修改操作，两个参数传参操作

}

mapper.xml

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

<!DOCTYPE mapper SYSTEM "F:\java资源包\MyBatis\mybatis-3-mapper.dtd" >

<mapper namespace="com.wwl.dao.UserDao"> 第一步：和接口类关联

插入操作：依次参数：方法名，useGeneratedKeys：开启自增长主键id返回，实体类的id，表中的主键id，传入类型参数

<insert id="insertUser" useGeneratedKeys="true" keyColumn="id" keyProperty="id" parameterType="com.wwl.entity.User">

insert into test(name, age) values(#{name},#{age})

</insert>

查询操作：依次参数：接口中对应方法名，返回类型(不用指定list)

<select id="selectUser" resultType="com.wwl.entity.User">

select \* from test

</select>

删除操作：传入的是基本类型：使用param 1，如果有两个参数就是： param 2，注意顺序，基本数值类型返回类型可以不指定

<delete id="deleteUser">

delete from test where id=#{ param 1}

</delete>

修改操作：两个参数传入：有引用类型和基本数据类型，使用注解传参

<update id="updateUser">

update test

<set> 使用set标签可以智能的去除参数后面的逗号：如name=#{user.name},

<if test="user.name != null">

name=#{user.name}, //接口中指定实体类User：@param(“user”)

</if>

<if test="user.age != null">

age=#{user.age}

</if>

</set>

where id=#{id} //接口中指定基本类型Integer：@param(“id”)

</update>

</mapper>

## 第四步 调用

public static void main(String[] args) throws IOException {

//1、加载核心配置文件：使用核心类mybatis中：org.apache.ibatis.io.Resources

InputStream inputStream = Resources.getResourceAsStream("MyBatis-config.xml");

//2、建立sqlSessionFactory使用类SqlSessionFactoryBuilder创建工厂(只建立工厂类，后续就可丢弃)

SqlSessionFactory sessionFactory = new SqlSessionFactoryBuilder().build(inputStream);

//3、创建会话：每次创建不同会话都需要通过工厂类开启连接数据库

SqlSession sqlSession = sessionFactory.openSession();

4、获取动态代理接口Dao对象：传入和mapper.xml对应的类字节码

UserDao userDao = sqlSession.getMapper(UserDao.class);

5、增删改查方法：注意：增删改必须要最后进行commit提交事务操作，不然mybatis不会提交到数据库

//insert(sqlSession, userDao);

delete(sqlSession, userDao, 4);

//update(sqlSession, userDao, 5);

select(sqlSession, userDao);

sqlSession.close(); // 和spring整合后由spring管理

}

private static void insert(SqlSession sqlSession, UserDao userDao) {

User user=new User(34,"温文亮");

userDao.insertUser(user);

sqlSession.commit(); //注意需commit操作

System.out.println(user.getId()+"----");

}

private static void select(SqlSession sqlSession, UserDao userDao) {

List<User> list = userDao.selectUser();

for(User user:list) {

System.out.println(user);

}

}

private static void delete(SqlSession sqlSession, UserDao userDao,Integer id) {

int i = userDao.deleteUser(id);

if(i>0) {

System.out.println("删除成功");

}else {

System.out.println("删除失败");

}

sqlSession.commit();

}

private static void update(SqlSession sqlSession, UserDao userDao,Integer id) {

User user= new User();

user.setName("wen35");

//user.setAge(34);

int i = userDao.updateUser(user, 5);

sqlSession.commit();

}

第二种方式的调用(了解即可)

Users u=sqlSession.selectOne("users.getUsersById", 15); //和第一种方式的方法不一样：直接传xml中的id名称，和值

System.out.println(u.getUserName()); //有值

System.out.println(u.getId()); //null 数据库的字段和pojo的字段不一致

}

# SSM整合步骤

## mapper代理模式开发

### 第一步 导入jar包

1、mybatis-3.4.6(核心包)、mysql-connector-java-5.1.4(mysql驱动包)

2、springmvc是spring框架中的子框架(web层框架)所以不用整合，完美融合

3、spring的框架包：aop、jdbc、tx(用于mybatis事务需要)、comtext、core、web、等等

4、日志Log4j：1.2.16和commons-logging-1.2还有配置文件log4j.properties(放到src下)

### 第二步 web中的配置

<!--1、配置监听器ContextLoaderListener用于：配置数据源、mapper扫描、mybatis相关配置-->

说明：如果是单工程(不是分布式工程)监听器ContextLoaderListener不用配置，下面servlet已经都有这功能

<!-- 加载application-context.xml文件 -->

<context-param>

<param-name>contextConfigLocation</param-name>

<param-value>classpath:application-\*.xml</param-value>

</context-param>

<listener>

<listener-class>org.springframework.web.context.ContextLoaderListener</listener-class>

</listener>

<!--2、配置springmvc的servlet：用于配置扫描注册bean、mvc的注解驱动 -->

<servlet>

<servlet-name>springmvc</servlet-name>

<servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>

<init-param>

<!-- springmvc配置文件加载 -->

<param-name>contextConfigLocation</param-name>

<!-- 单个工程也是需要配置多个xml的：写成springmvc-\*.xml -->

<param-value>classpath:springmvc-servlet.xml</param-value>

</init-param>

<!-- 让此servlet在服务启动就加载 (默认是第一次请求访问此servlet才初始化)-->

<load-on-startup>1</load-on-startup>

</servlet>

<servlet-mapping>

<servlet-name>springmvc</servlet-name>

<!-- 设置成为默认的servlet(tomcat就有全局的默认寻找静态资源、后缀.jsp的解析器) -->

<url-pattern>/</url-pattern>

</servlet-mapping>

<!--3、 配置post请求编码：此过滤器jar文件在spring.web的filter包下 -->

<filter>

<filter-name>uncode</filter-name>

<filter-class>org.springframework.web.filter.CharacterEncodingFilter</filter-class>

<init-param>

<param-name>encoding</param-name>

<param-value>utf8</param-value>

</init-param>

</filter>

<filter-mapping>

<filter-name>uncode</filter-name>

<url-pattern>/\*</url-pattern>

</filter-mapping>

### 第三步 Spring的xml配置

#### controller层配置

springmvc-web.xml的配置 和前端浏览器交互一些配置

<!--1、配置注册扫描bean 此标签包含了<context:annotation-config>功能 -->

<context:component-scan base-package="com.wwl"/>

<!-- 2、此标签配置：3.1以后新版本注解映射器和注解适配器(不配置也可以有默认加载旧的)-->

<mvc:annotation-driven/>

<!--3、 配置视图解析器进行优化：前缀和后缀。(系统默认加载的进行配置是为了优化功能) -->

<bean class="org.springframework.web.servlet.view.InternalResourceViewResolver">

<!-- 配置逻辑视图的前缀 -->

<property name="prefix" value="/WEB-INF/jsp/"/>

<!-- 配置逻辑视图的后缀 -->

<property name="suffix" value=".jsp"/>

</bean>

<!--4、 配置静态资源不被拦截 -->

<mvc:resources location="/img/" mapping="/img/\*\*"/>

<mvc:resources location="/js/" mapping="/js/\*\*"/>

<mvc:resources location="/css/" mapping="/css/\*\*"/>

#### service事务配置

事务是可不用配置的不像hibernate要在事务中才能操作，但是企业开发事务还是要配置的

application-service.xml配置

<!-- 扫描service包下所有使用注解的类型 -->

<context:component-scan base-package="com.wwl.service" />

<!-- 配置事务管理器：使用到的是jdbc的注解 -->

<bean id="transactionManager" class="org.springframework.jdbc.datasource.DataSourceTransactionManager">

<property name="dataSource" ref="dataSource"/> <!-- 注入数据库连接池 -->

</bean>

<!-- 开启注解的事务：配置基于注解的声明式事务-->

<tx:annotation-driven transaction-manager="transactionManager"/>

<!—开启xml的注解事务操作 -->

<!-- 通知 -->

<tx:advice id="txAdvice" transaction-manager="transactionManager">

<tx:attributes>

<!-- 传播行为 -->

<tx:method name="save\*" propagation="REQUIRED" />

<tx:method name="insert\*" propagation="REQUIRED" />

<tx:method name="add\*" propagation="REQUIRED" />

<tx:method name="create\*" propagation="REQUIRED" />

<tx:method name="delete\*" propagation="REQUIRED" />

<tx:method name="update\*" propagation="REQUIRED" />

<tx:method name="find\*" propagation="SUPPORTS" read-only="true" />

<tx:method name="select\*" propagation="SUPPORTS" read-only="true" />

<tx:method name="get\*" propagation="SUPPORTS" read-only="true" />

</tx:attributes>

</tx:advice>

<!-- 切面 -->

<aop:config>

<aop:advisor advice-ref="txAdvice"

pointcut="execution(\* cn.itcast.core.service.\*.\*(..))" />

</aop:config>

#### 持久层Dao配置

springmvc-dao.xml的配置：数据源、mybatis整合配置

<!-- 导入properties文件 ：注意key命名要有一定的特殊规则：不要单user命名,jdbc.user -->

<context:property-placeholder location="classpath:mysql-config.properties"/>

<!--第一步 c3p0连接池数据源 -->

<!-- c3p0数据源的配置 需要配置下面的数据，不然会导致8小时后没用过 出现错误-->

<bean id="dataSource" class="com.mchange.v2.c3p0.ComboPooledDataSource" destroy-method="close">

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

<property name="jdbcUrl" value="jdbc:mysql:///wwl ?useUnicode=true&characterEncoding=utf8"/>开启编码，使用编码utf8

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

<property name="password" value="123456"/>

<!-- 指定连接池中保留的最大连接数. Default:15 -->

<property name="maxPoolSize" value="20"/>

<!-- 指定连接池中保留的最小连接数-->

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

<!-- 指定连接池的初始化连接数  取值应在minPoolSize 与 maxPoolSize 之间.Default:3-->

<property name="initialPoolSize" value="8"/>

<!-- 最大空闲时间,60秒内未使用则连接被丢弃。若为0则永不丢弃。 Default:0-->

<property name="maxIdleTime" value="120"/>

<!-- 当连接池中的连接耗尽的时候c3p0一次同时获取的连接数. Default:3-->

<property name="acquireIncrement" value="3"></property>

<!-- 每60秒检查所有连接池中的空闲连接.Default:0 -->

<property name="idleConnectionTestPeriod" value="60"/></bean>

<!-- 第二步 配置sqlSessionFactoryBean <!-- Mybatis的工厂用于获取mybatis核心配置等 -->

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

<!-- 注入数据库连接池 -->

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

<!-- 配置MyBaties核心配置文件:mybatis-config.xml -->

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

<!-- 扫描entity包使用别名 (注意在mybaties中配置了这边就不要配置了) -->

<property name="typeAliasesPackage" value=" com.wwl.entity " />

<!-- 扫描sql配置文件:mapper需要的xml文件 (注意:mapper.xml和接口类在同一包中就不要配置了) -->

<property name="mapperLocations" value="classpath:mapper/\*.xml" />

</bean>

<!-- 第三步 配置Mapper动态代理开发 扫描 如果需要扫描多个包，中间使用英文逗号隔开 -->

<bean class="org.mybatis.spring.mapper.MapperScannerConfigurer">

<!-- 此标签sqlSessionFactoryBeanName可以不配置看[困扰问题](#_困扰问题)，数据源有多个才配置 -->

<property name="sqlSessionFactoryBeanName" value="sqlSessionFactory" />

<property name="basePackage" value="com.wwl.mapper"/>

</bean>

### 第四步 mybatis核心配置

mybatis-config.xml 的配置：配置别名，sql语句控制输出

1、配置mapper：由于使用spring和mybatis的整合包进行mapper扫描，这里不需要配置了

但是必须遵循：mapper.xml和mapper.java文件同名且在一个目录(application-dao.xml的工厂中可以配置不同的目录)

2、配置别名 (注意：spring-mvc.xml的工厂中也可以配置别名(任选其一))

<!-- 1、配置别名:包其子包下所有类有效，大小写不敏感 作用:mapper.xml下parameterType、resultType不用写全路径类名-->

<typeAliases>

<package name="com.wwl.entity"/>

</typeAliases>

3、配置全局settiing的配置：根据需要添加

<settings>

<!-- 使用jdbc的getGeneratedKeys获取数据库自增主键值(不配置都可以的) 只有insert插入才有效返回-->

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

<!-- 使用列标签(字段名)替换列别名(sql语句设置的别名) 默认:true 了解都默认是ture了还配置啥-->

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

<!-- 开启驼峰命名转换:Table{create\_time} -> Entity{createTime} 默认:true -->

<setting name="mapUnderscoreToCamelCase" value="true" /> <!-- 这个实用当然默认也是true-->

<!-- 打印查询语句 -->

<setting name="logImpl" value="STDOUT\_LOGGING" />

</settings>

### 第五步 接口和mapper

参照单独使用操作[第三步](#_第三步_接口和mapper) 一样的

### 困扰问题

扫描mapper.xml配置问题：MapperScannerConfigurer

问题点：有些老师配置<property name="sqlSessionFactoryBeanName" value="sqlSessionFactory" />

<bean class="org.mybatis.spring.mapper.MapperScannerConfigurer">

<property name="basePackage" value="org.mybatis.spring.sample.mapper" />

<property name="sqlSessionFactoryBeanName" value="sqlSessionFactory" /> 有些老师配置有些不配置

</bean>

basePackage 属性是让你为映射器接口文件设置基本的包路径。 你可以使用分号或逗号 作为分隔符设置多于一个的包路径。每个映射器将会在指定的包路径中递归地被搜索到。

解决问题点

注 意 , 没 有 必 要 去 指 定 SqlSessionFactory 或 SqlSessionTemplate , 因 为：

MapperScannerConfigurer 将会创建 MapperFactoryBean,之后自动装配。但是,如果你使用了一个以上的 DataSource ,那么自动装配可能会失效 。这种情况下 ,你可以使用 sqlSessionFactoryBeanName 或 sqlSessionTemplateBeanName 属性来设置正确的 bean 名 称来使用。这就是它如何来配置的,

注意 bean 的名称是必须的,而不是 bean 的引用,因此,value 属性在这里替代通常的 ref:

<property name="sqlSessionFactoryBeanName" value="sqlSessionFactory" />

经测试mybatis核心配置文件：

<setting name="useGeneratedKeys" value="true" />这个可以不用配置

直接在标签<insert>中写即可

插入操作：依次参数：方法名，useGeneratedKeys：开启自增长主键id返回，实体类的id，表中的主键id，传入类型参数

<insert id="insertUser" useGeneratedKeys="true" keyColumn="id" keyProperty="id" parameterType="com.wwl.entity.User">

insert into test(name, age) values(#{name},#{age})

</insert>

## 传统Dao开发

这种开发都是以了解为主，了解原理

### 第一步 导入jar包

看代理模式

### 第二步 web中的配置

看代理模式

### 第三步 spring的xml配置

# 1、输入、输出映射

## 输入参数传递

### 传递基本数值类型

参数只有一个

1、只要是8大基本数值类型都可以省略parameterType 属性设置

2、参数写法：#{这里可以随便写}

Xml写法

<select id="selectTestById" resultType="user" parameterType="int" > //parameterType="int"可省略

select \* from test where id=#{fffffff}

</select>

接口类

public User selectTestById(Integer id);

参数两个以上

1、只要是8大基本数值类型都可以省略parameterType属性设置

2、有两种写法：

1) 接口类未定义任何方式：#{ param1}、#{ param2}按接口类从左到右顺序写即可

2)在接口类使用注解方式：@Param("age") 也适用于实体类

Xml写法

<select id="likeTest" resultType="test" >

select \* from user

<where>

<if test="name !=null">

and age=#{age} //接口没定义：另一种方式：#{ param2}

</if>

<if test="name !=null">

and name like #{name} //接口没定义：另一种方式：#{ param1}

</if>

</where>

</select>

接口类

第一种 定义注解方式

public List<User> likeTest(@Param("name")String name,@Param("age")Integer age);

第二种 未定义任何方式

public List<User> likeTest(String name, Integer age);//xml使用#{ param1}代替第一个name参数、#{ param2}代替第二

### 传递pojo类

1、映射xml中写入parameterType="com.wwl.pojo.User"

2、参数就是成员变量名：#{age}: age就是User的成员变量名，必须一致

Xml写法

<insert id="insertTest" parameterType="user"> //使用了别名：对大小写不敏感

insert into test(name, age) values(#{name},#{age})

</insert>

接口类

public int insertTest(User user);

### 传递pojo包装对象

意思就是：对象中又有对象

实体类(pojo)

public class Qu{

//维护者用户对象

private User user;

public User getUser() {

return user;

}

public void setUser(User user) {

this.user = user;

}

}

Xml写法

1、resultType:如果要返回数据集合，只需设定为每一个元素的数据类型

2、包装的pojo取值通过 "."来获取

<select id="getUserByQu" parameterType="qu" resultType="com.itheima.mybatis.pojo.User">

SELECT \* FROM USER WHERE username LIKE '%${user.username}%' //根据成员变量来 点 获取user对象中属性值

</select>

接口类

Public User getUserByQu(QU qu) 参数是pojo包装对象

## 输出映射结果

### resultType(输出类型)

#### 输出简单数值类型

XMl写法

统计表的总数

1、输出类型不能省略返回类型，不像输入的基本数值类型可以省略

2、容易搅浑插入、修改操作返回的影响行(这个int类型是可以省略的)

<!-- 查询用户总记录数，演示返回简单类型 -->

<select id="getUserCount" resultType="int"> //必须要有resultType标签 不能省略

SELECT COUNT(1) FROM USER

</select>

接口类写法

public int getUserCount ();

#### 输出pojo对象

XMl写法

根据id查询表

<select id="selectTestById" resultType="User">

select \* from user where id=#{id}

</select>

接口类写法

public User selectTestById(Integer id); //只会返回一条记录，User接收

#### 输出pojo对象列表

Sql语句查询有时不止返回一条对象信息，而是很多，接口类用List<pojo>接收即可

XMl写法

模糊查询

<select id="likeTest" resultType="user"> 这里还是类型返回值即可

select \* from user

<where>

<if test="name !=null">

and age=#{ param2}

</if>

<if test="name !=null">

and name like #{ param1}

</if>

</where>

</select>

接口类写法

public List<User> likeTest(String name, Integer age);//用List<User>返回值接收即可

### resultMap(输出类型)

1数据库的表名和实体类属性名不一致可以使用(也可以使用别名解决问题)

1）单表为了解决名称不一致问题(属性名与表字段)只写不一致的即可

2）涉及到联表查询：建议写全pojo属性，因为查询的结果会有重复的列名(使用sql语句的别名进行区分)

2 具体写法请查看[联表查询](#_4、联表查询)

注意 resulType无法实现延迟加载，resultMap(association、collection才有延迟加载)

# 2、文件配置

## 核心配置

核心配置文件 ：1数据源、2事务管理 、3找映射文件、4配置别名

配置顺序 必须是下面顺序来不然报错

properties（属性）

settings（全局配置参数）

typeAliases（类型别名）

typeHandlers（类型处理器）

objectFactory（对象工厂）

plugins（插件）

environments（环境集合属性对象）

environment（环境子属性对象）

transactionManager（事务管理）

dataSource（数据源）

mappers（映射器）

<mappers>加载映射文件

<mappers>

1使用mapper接口类路径：此方法要求mapper接口名称和mapper映射文件名称相同，且放在同一个目录中

<mapper class="com.itheima.mybatis.mapper.UserMapper" /> //指定接口类

2、注册指定包下的所有mapper接口：此方法要求mapper接口名称和mapper映射文件名称相同，且放在同一个目录中

<package name="com.itheima.mybatis.mapper"/>

3、使用相对于类路径的资源：此配置没有硬性要求

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

</mappers>

<typeAliases>配置别名

<typeAliases>

<typeAlias type="com.wwl.pojo.Users" alias="h"/> //单个匹配类

<package name="com.wwl.pojo"/>!-- 批量别名定义，扫描整个包下的类，别名为类名（大小写不敏感） -->

</typeAliases>

然后在resultType或：parameterType中就可以不用使用全类路径的名字了

settiing配置文件

<settings>

<!-- 使用jdbc的getGeneratedKeys获取数据库自增主键值(不配置都可以的) 只有insert插入才有效返回-->

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

<!-- 使用列标签(字段名)替换列别名(sql语句设置的别名) 默认:true 了解都默认是ture了还配置啥-->

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

<!-- 开启驼峰命名转换:Table{create\_time} -> Entity{createTime} 默认:true -->

<setting name="mapUnderscoreToCamelCase" value="true" /> <!-- 这个实用当然默认也是true-->

<!-- 打印查询语句 -->

<setting name="logImpl" value="STDOUT\_LOGGING" />

</settings>

## 映射配置

1、根标签<mapper>配置

1、属性namespace：命名空间，用于隔离sql -->

2、还有一个很重要的作用，使用动态代理开发DAO， namespace必须和Mapper接口类路径一致 -

<mapper namespace="cn.itcast.mybatis.mapper.UserMapper">

2、查询标签<select>

1、属性id必须和Mapper接口方法名一致

2、属性parameterType必须和接口方法参数类型一致

3、属性resultType必须和接口方法返回值类型一致

<select id="queryUserById" parameterType="int"

resultType="cn.itcast.mybatis.pojo.User">

select \* from user where id = #{id}

</select>

3、resultMap标签的配置

1)如果sql查询字段名和pojo的属性名不一致，可以通过resultMap将字段名和属性名作一个对应关系 ，resultMap实质上还需要将查询结果映射到pojo对象中。

2)resultMap可以实现将查询结果映射为复杂类型的pojo，比如在查询结果映射对象中包括pojo和list实现一对一查询和一对多查询。

要明白：这个标签其实就是让实体类属性和表字段去对应：如sql语句中使用了别名就要和别名去对应，如果属性和表字段有相同的就不用写(包括id标签)，一般用于多表关联。

<resultMap type="order" id="orderResultMap">

<!-- 定义主键 ,非常重要。如果是多个字段,则定义多个id -->

<id property="id" column="id" />

<!-- 定义普通属性 -->

<result property="userId" column="user\_id" /> <!-- property：主键在pojo中的属性名 column：主键在数据库中的列名-->

<result property="number" column="number" />

<!-- 定义一对一association标签-->

<!-- association : property参数是：实体类UserChild对应的属性名，可以直接resultMap或javaType -->

<association property="user2" javaType="com.wwl.entity.User2" >…. </association> //类型使用javaType或resultMap

<!-- 定义一对多collection标签-->

<collection property="orders" ofType=" com.wwl.entity.Order">…. </ collection > //使必须使用ofType标签，实体类的参数是List<Order> orders

</resultMap>

实体类属性和表字段都一致：可以简单写：下边是定义多对多写法

<resultMap type="order" id="orderResultMap">

<!-- 定义一对一association标签-->

<association property="user2" javaType="com.wwl.entity.User2" >

<!-- association标签里可以进行定义一对一-->

<association property="user3" javaType="com.wwl.entity.User2" >…. </association>

<!--association标签里可以定义一对多-->

<collection property="orders" ofType=" com.wwl.entity.Order">…. </ collection >

</association>

</resultMap>

4、<sql>标签：sql片段

1)引入本身XML的操作

<mapper namespace="com.wwl.mapper.UsersMapper">

<sql id="userSql"> 第一步 声明一个sql片段

select \* from users

</sql>

<select id="getByUserId" parameterType="Integer" resultType="Users">

<include refid="userSql"/> 第二步 引入sql片段

where userId=#{v}

</select>

</mapper>

2)引入另一个XML中的sql片段的操作：在refid前面加上对应的Mapper.xml的namespace

<include refid="com.wwl.mapper.AbcMapper.AbcSql"/

5、<where>语句标签和if语句标签

where标签可以自动添加where，同时处理sql语句中第一个and关键字

<!-- 根据条件查询用户 -->

<select id="queryUserByWhere" parameterType="user" resultType="user">

SELECT id, username, birthday, sex, address FROM `user`

<!-- where标签可以自动添加where，同时处理sql语句中第一个and关键字 -->

<where>

<if test="sex != null">

AND sex = #{sex}

</if>

<if test="username != null and username != ''">注意字符串类型的数据需要要做不等于空字符串校验。

AND username LIKE

'%${username}%'

</if>

</where>

</select>

6、<foreach>标签的使用

collection属性：参数是list、array(数组)、ids(map)3个，遍历的集合

item属性：遍历，值可以随便写，，但是和后面的#{}里面要一致

open属性：循环开启前的内容

close属性：循环结束后输出的内容

separator属性：指定遍历的元素之间使用的分隔符，如 逗号，等

<foreach collection="list" item="item" open="id IN (" close=")"separator=",">

#{item} 一致

</foreach>

# 3、增删改查操作

注意事项:1）增删改操作:必须要调用sqlSession.commit(),mybatis默认是开启事务的

2）或者在sqlSessionFactory.openSession(true);true：交由自己来管理事务

## insert

### 主键id返回

第一种 方式获取主键id

核心配置文件：测试结果其实都不用配置的不知道是不是版本问题

<!-- 使用jdbc的getGeneratedKeys获取数据库自增主键值(不配置都可以的) 只有insert插入才有效返回-->

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

XML操作

依次参数：方法名，useGeneratedKeys：开启自增长主键id返回，实体类的id，表中的主键id，传入类型参数

<insert id="insertUser" useGeneratedKeys="true" keyColumn="id" keyProperty="id" parameterType="com.wwl.entity.User">

insert into test(name, age) values(#{name},#{age})

</insert>

代码部分

User user=new User(34,"温文亮");

userDao.insertUser(user);

sqlSession.commit(); //进行事务提交

sqlSession.close(); //整合后spring管理

System.out.println(user.getId()+"----"); //这样就获取返回的主键id

第二种 获取主键

XML操作

AFTER的主键id操作：它是先执行完insert插入，后返回主键id

<!-- 获取插入数据后的id值，一个表插入数据后，因为id主键是自增长的，插入数据后需要获取到ID，可下面的方式 -->

<!-- keyProperty：主键对应的pojo中的属性 -->

<insert id="insertUser" parameterType="Users">

<selectKey keyProperty="id" resultType="Integer" order="AFTER"> AFTER用于主键自增长方式(先数据insert后插入后再返回主键封装到对象中)

select LAST\_INSERT\_ID()<!-- 是MySqlne内置函数方法获取上一次 insert 进去后自增主键的值-->

</selectKey>

insert into user(username,birthdat,address,sex)

values(#{username},#{birthdat},#{address},#{sex})

</insert>

BEFORE的设置返回主键id操作：先返回主键值封装后，再insert插入数据

BEFORE：先是先执行selectKey返回主键UUID的值封装到Users中，后才执行插入操作

<insert id="insertUser" parameterType="Users">

//参数：指定类中的属性接收封装 表中的类型(UUID都是字符串)，所以设定string类型

<selectKey keyProperty="id" resultType="string" order="BEFORE"> 先生成主键BEFORE返回Users中再进行insert语句插入

SELECT UUID() MySQL内置函数得到UUID的值

</selectKey>

insert into user(username,birthdat,address,sex)

values(#{username},#{birthdat},#{address},#{sex})

</insert>

代码部分

public static void insertUser() {

InputStream is = Resources.getResourceAsStream("mybatis-config.xml");

SqlSessionFactory s= new SqlSessionFactoryBuilder().build(is);

SqlSession sqlSession=s.openSession();

UsersMapper usersMapper=sqlSession.getMapper(UsersMapper.class);

Users user= new Users(“曾老婆”, "15326584545");

usersMapper.insertUser(user); //插入成功后，就会使用mysql内置函数调回自增长的id

System.out.println(user.getId());//然后user中就会有增长id返回

}

## update

修改(upadte)

XML操作 参数是两个以上的可以使用注解@Param，或者嫌麻烦可以按顺序写param1....2..3

下面修改例子是通用写法：表中字段很多，使用判断标签<if>配合<set>标签这两个就可以写出字段哪一个要修改，哪一个字段不修改

(小注意：以前搞晕了:调用逆向工程代码为啥我只修改一个字段，其它都变成null呢，请明白最后都是看sql语句的)

<update id="updateUser"> //不用写返回值类型，

update test

<set>使用set智能判断多条件参数后的逗号name=#{user.name}**,**是否去掉

<if test="user.name != null">

name=#{user.name},

</if>

<if test="user.age != null">

age=#{user.age}

</if>

</set>

where id=#{param2} //接口类参数在第二个位置：所以可以这样写

</update>

代码部分调用

public int updateUser(@Param("user")User user,Integer id);

User user= new User();

user.setName("wen35");

//user.setAge(34);

int i = userDao.updateUser(user, 5);

sqlSession.commit();

## select

### where

两个条件 根据年龄和名字查询用户

XMl操作

写这个例子是让你明白模糊查询语句的两种参数传参方式：使用${}或#{}

<select id="likeUser" resultType="com.wwl.entity.User">

select \*from test

<where> <!-- where标签可以自动添加where，智能判断添加或取消第一个and -->

<if test="param1 !=null">

and age=#{param1}

</if>

<if test="param2 !=null">

and name like ‘%${param2}%’ 有单引号 //或and name like #{param2} 代码参数中带%%

</if>

</where>

</select>

代码操作调用

public List<User> likeUser(Integer age,String name);

Integer age=null;

String name="温";//或String name="%温%";

userDao.likeUser(age, name);

### choose

判断语句跟if差不多

使用此例子是要告诉你：特定场景下如果都不匹配(为了安全性)，什么信息都没有

<where>

<choose>

<when test="a !=null"></when>

<when test="b !=null "></when>

<when test=" c !=null "></when>

<otherwise>and 1=2</otherwise> //语句写成这样就什么都查询不了

</choose>

</where>

上面意思是：所有when都不成立，才会执行otherwise语句

### <foreach>

根据多个id查询：select \* from users where userid in (1,2,3,….)

第一种 list传参：public List<Users> selectUserByIds(List<Integer> list); 传入参数是list集合

代码操作调用

这个例子：条件语句中使用的是in语句,或者有多个值传参进来(list)需要在这个范围内找的

List<Integer> list= new ArrayList<Integer>();

lis.add(1);lis.add(15);

List<Users> result=userMapper.selectUserByIds(list);//调用操作

System.out.println(result);

XMl操作

使用foreach标签，进行遍历，此标签支持3种集合：list、array、ids<map>

<!-- collection：遍历的集合:list(List)、array(数组)、ids(map) -->

<!-- item：遍历的内容，值可以随便写，但是和后面的#{}里面要一致 -->

<!-- open：循环前输出的内容，close：循环结束后输出的内容 -->

<!-- separator：指定遍历的元素之间使用的分隔符 -->

<select id="selectUserByIds" resultType="com.wwl.entity.User">

select \* from test where id in

<foreach collection="list" item="a" separator="," open="(" close=")">// 也可以open="userid in (" close=")

#{a}

</foreach>

</select>

第二种 参数是数组：public List<User> selectUserByIds(Integer[] i);

XMl操作 和list不同之处：collection="array"其它都一样

sql语句 ：select \* from users where userid in (1,2,3)

<select id="selectUserByIds" resultType="com.wwl.entity.User">

select \* from test where id in

<foreach collection="array" item="a" separator="," open="(" close=")">

#{a}

</foreach>

</select>

代码操作调用

Integer[] array= {3,5};

List<user> list=userDao.selectUserByIds(array);

第三种 参数是Map：public List<User> selectUserByIds(Map map);差不多的，这里不重复了

# 4、联表查询

上面的内容都是基于单表查询的

## 一对一

### ResultType封装数据

一对一:使用的sql语句一般都是内连接inner join，附带where条件,查询出来的数据必须是一条

代码部分

实体类

新建UserChild的pojo，继承自User。

public class UserChild extends User{

private String username; //添加联表查询少的字段属性

private String address; //添加联表查询少的字段属性

…….get,set

}

dao接口类

public UserChild innerJoinUser(@Param("id")Integer id);

xml写法

<!-- 一对一关联查询，使用resultType -->

<select id="getOrderUser" resultType="UserChild"> 使用UserChild接收

SELECT o.`user\_id`, o.`number`,o.`createtime`,u.`username`,u.`address`

FROM `user` o inner JOIN `aaa` u ON u.id = o.`user\_id` where u.id=#{id}

</select>

总结：你就想象是单表:只要sql语句查出来的字段能跟实体类属性名一一对应就能封装

### ResultMap封装数据

一对一:使用的sql语句一般都是内连接inner join，附带where条件,查询出来的数据必须是一条

代码部分

1、实体类

public class User {

private Integer id;

private String name;

private Integer age;

private User2 user2; //关联user2，实际开发不会在此类写

setget省略

}

实际开发：不要动本类：继承上面类

public class UserChild extends User{

private User2 user2; //这里关联user2

setget省略(必须要写，mybatis是通过getset映射)

}

2、dao接口类

public UserChild innerJoinUser(@Param("id")Integer id); //实际开发：返回参数是继承User

xml写法

1、select查询方法

<select id="innerJoinUser" resultMap="UserMap">返回值resultMap类型

select u\*, u2.\* from test u INNER JOIN test2 u2 WHERE u.user2\_id=u2.id and u.user2\_id=#{id}

</select>

2、resultMap写法

要明白：这个标签其实就是让实体类属性和表字段去对应：如sql语句中使用了别名就要和别名去对应，如果属性和表字段有相同的就不用写(包括id标签)，一般用于多表关联。

<resultMap type="com.wwl.entity.UserChild" id="UserMap">

<id column="id" property="id"/> <!-- id：标识唯一的，如要多个标识配置多个id -->

<result column="name" property="name"/> column：数据库表中的字段（列名）property：对象中的属性名

<result column="age" property="age"/>

<!-- association : property参数是：实体类UserChild对应的属性名，可以直接resultMap或javaType -->

<association property="user2" javaType="com.wwl.entity.User2" >//这里也可以是resultMap： resultMap="User2Map"调用下面resultMap

<id column="id" property="userId2"/>

<result column="address" property="address"/>

</association>

</resultMap>

User2Map的resultMap

<resultMap type="com.wwl.entity.User2" id="User2Map">

<id column="id" property="userId2"/>

<result column="address" property="address"/>

</resultMap>

注意resulType无法实现延迟加载，resultMap的association、**collection**：可以实现延迟加载

调用

Integer id=1;

UserChild user = userDao.innerJoinUser(id);

String address = user.getUser2().getAddress();

System.out.println(address);

## 一对多

也可以用ResultType这里不写了

ResultMap封装数据

代码部分

User实体类中添加：List<Order> orders

Public class User{

Private Integer id;

private String username;

private String set;

private List<Order> orders; //添加此属性

setget省略 必须要有getset

}

实际开发不会动原装pojo 继承此pojo

Public class UserChild extends User{

private List<Order> orders; //添加此属性

setget省略 必须要有getset

}

XMl部分

要注意的地方：

1、sql语句使用了别名，那么就要用别名：<id property="id" column="oid"/>这个oid是SQL语句别名

2、一对多的ResultMap的标签<collection property="orders" ofType="order">

<!-- 一对多关联查询 -->

<resultMap type="user" id="user\_order\_map">

<id property="id" column="id" />

<result property="birthday" column="birthday" />

<result property="address" column="address" />

<result property="sex" column="sex" />

<result property="uuid2" column="uuid2" />

<!-- collection:配置一对多关系

property:用户下的order属性

ofType:property的数据类型，支持别名-->

<collection property="orders" ofType="order">

<id property="id" column="oid"/> //查询语句中id使用了别名，这里必须写别名

<!-- 使用result绑定普通字段 -->

<result property="userId" column="id"/>

<result property="number" column="number"/>

<result property="createtime" column="createtime"/>

</collection>

</resultMap>

一对多关联查询：使用LEFT JOIN左连接查询

<select id="getUserOrder" resultMap="user\_order\_map">

SELECT u.`birthday`,u.`sex`,u.`address`,u.`uuid2`,o.`id` oid, o.`number`,o.`createtime`

FROM `user` u LEFT JOIN `order` o ON o.`user\_id` = u.`id`

</select>

# 5、延迟加载/缓存

## 延迟加载

**延迟加载(懒加载、按需加载)：在mybatis中，resultMap标签的association标签和collection标签具有延迟加载的功能**

前提

两表关系：demo和demo2

demo表有外键demo2\_id维护着 demo2表的主键id

实体类关系: Demo和Demo2

Demo是维护着Demo2的对象：private Demo2 demo2;

第一步 关键：主查询要返回多个结果sql语句(有测试过)才会有效

<select id="getDemoByIdMapList" resultMap="DemoMap"> 返回类型是resultMap

select \* from demo 此语句能查询出多个结果

</select>

接口类

public List<Demo> getDemoByIdMapList(); //返回的是要list集合

第二步 需要ResultMap才能支持懒加载

这里要注意：因为我的两表字段名和两个实体类属性名都是一样的，就没有写任何属性名(这样是可以的)

参数 property:Demo类中的Demo2对象属性名

column:是表demo的外键字段名，以此查询出的外键字段作为参数查询下一条sql语句

select：是根据id 查询表demo2的select标签引用

<resultMap type="com.wwl.entity.Demo" id="DemoMap">

<association property="demo2" column="test2\_id" select="com.wwl.dao.Demo2Dao.getDemo2ById" >

</association>

</resultMap>

下面是根据id 查询表demo2

<mapper namespace="com.wwl.dao.Demo2Dao">

<!-- 根据id查询 test2 -->

<select id="getDemo2ById" resultType="com.wwl.entity.Demo2">

select \* from demo2 where id=#{id}

</select>

</mapper>

第三步 配置懒加载

<!-- 打开延迟加载的开关 -->

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

<!-- 将积极加载改为消极加载，即延迟加载 -->

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

第四步 测试

InputStream inputStream = Resources.getResourceAsStream("MyBatis-config.xml");

SqlSessionFactory sqlSessionFactory = new SqlSessionFactoryBuilder().build(inputStream);

SqlSession sqlSession = sqlSessionFactory.openSession();

DemoDao demoDao = sqlSession.getMapper(DemoDao.class);

执行到这一步：只会有这一条主语句：select \* from demo，后续没有代码就不会有别的sql语句

List<Demo> mapList = demoDao.getDemoByIdMapList();

执行到这一步：会执行一条sql语句：select \* from demo2 where id=#{id}

mapList.get(0).getDemo2();

执行到这一步：因为缓存机制保存了外键值只有这条语句select \* from demo2 where id=#{id}

mapList.get(1).getDemo2();

sqlSession.close();

## 缓存

### 一级缓存

一级缓存 指的是同一个sqlSession

工作原理：

第一次查询id为1的用户 ，先去找缓存是否有id为1的用户信息，没有从数据库查询

注意 ：如果sqlSession去执行commit操作（插入，删除，修改），会清空一级缓存（避免赃读）

第二次查询id如果有，先去找缓存，就不从数据库查

测试 ：默认支持一级缓存，不用去配置

设置两次查询一样的id，进行跟踪控制台信息， 第二次，中间加入一个插入功能

InputStream inputStream = Resources.getResourceAsStream("MyBatis-config.xml");

SqlSessionFactory sqlSessionFactory = new SqlSessionFactoryBuilder().build(inputStream);

SqlSession sqlSession = sqlSessionFactory.openSession(true); //事务关闭(也可以理解为交由自己管理)

Demo2Dao demo2Dao = sqlSession.getMapper(Demo2Dao.class);

Integer id=1;

Demo demo= demoDao.getDemo2ById(id);//执行第一次：语句select \* from demo2 where id=1

Demo byId = demoDao.getDemo2ById(1);//执行到此步：值还是1，直接从缓存拿，因为之前没有增删改操作

demoDao.insertDemo(demo4); //进行添加操作

demoDao.getDemo2ById(1); //缓存全部被清空了，又会和mysql交互语句select \* from demo2 where id=1

sqlSession.close();

其实呢：一般情况下在同一sqlSession很少调用同一个值的

### 二级缓存

二级缓存 (一般不会用mybatis的二级缓存(redis代替))

一级缓存是生命周期是在同一个sqlSession的

二级缓存是在同一个映射文件：<mapper namespace="com.wwl.dao.DemoDao">

第一步 核心配置开启二级缓存(默认关闭)

<!-- 二级缓存的总开关 -->

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

第二步 因为生命周期是mapper.xml的，还需要指定哪一个mapper.xml要开启

<mapper namespace="com.wwl.dao.DemoDao">

<!-- 开启二级缓存 cache默认使用了二级缓存的type="org.apache.ibatis.cache.impl.PerpetualCache"-->

<cache />

</mapper>

第三步 还需要把关联到的实体类序列化

不好用原因：缓存在map意思也是在内存中容易造成溢出，会进行序列化到硬盘

第四步 测试

测试 有增删改也会清空二级缓存

InputStream inputStream = Resources.getResourceAsStream("MyBatis-config.xml");

SqlSessionFactory sqlSessionFactory = new SqlSessionFactoryBuilder().build(inputStream);

//创建sqlSession对象

SqlSession sqlSession = sqlSessionFactory.openSession();

SqlSession sqlSession1 = sqlSessionFactory.openSession();

SqlSession sqlSession2 = sqlSessionFactory.openSession();

//获取demoDao对象：一级缓存是生命周期是在同一个sqlSession的，这个二级缓存是在同一个DemoDao.class接口类

DemoDao demoDao = sqlSession.getMapper(DemoDao.class);

DemoDao demoDao1 = sqlSession1.getMapper(DemoDao.class);

DemoDao demoDao2 = sqlSession2.getMapper(DemoDao.class);

demoDao.getDemoById(1);

sqlSession.close(); //在close的时候，才会将数据写入到二级缓存中

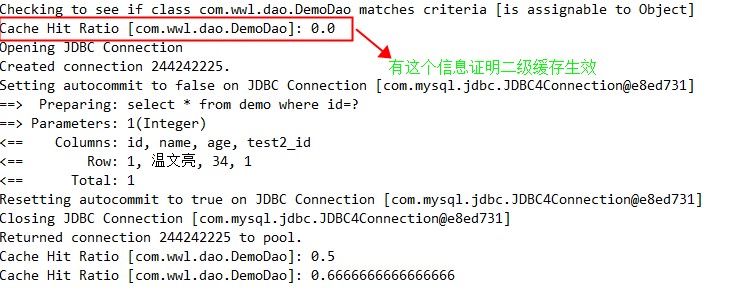
demoDao1.getDemoById(1);

sqlSession1.close();

demoDao2.getDemoById(1);

sqlSession2.close();

控制台信息



设置mapper.xml中下的某个方法不使用二级缓存

<mapper namespace="com.wwl.dao.DemoDao">

<!-- 开启二级缓存 cache默认使用了二级缓存type="org.apache.ibatis.cache.impl.PerpetualCache"-->

<cache />

<select id="getDemoById" resultType="com.wwl.entity.Demo" useCache="false"> 默认为true，false不使用二级缓存

select \* from demo where id=#{id}

</select>

</mapper>

刷新缓存

<mapper namespace="com.wwl.dao.DemoDao">

<!-- 开启二级缓存 cache默认使用了二级缓存-->

<cache type="org.apache.ibatis.cache.impl.PerpetualCache"/>

刷新缓存：在select语句中，默认值是false，在增删改语句中，默认值是true，如果增删改设置为false(那么就不会清空缓存)

<select id="getDemoById" resultType="com.wwl.entity.Demo" flushCache="true">

</mapper>

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级别的缓存实现不好。

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。

#### 整合ehcache

ehcache、memcache、redis三大缓存

ehcache：可以单独使用，下面介绍是整合到mybatis二级缓存中没实用的

二级缓存使用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。

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。

步骤

第一步 导包 核心包 ehcache-core-2.6.5.jar | 整合包：mybatis-ehcache

<dependency>

<groupId>net.sf.ehcache</groupId>

<artifactId>ehcache-core</artifactId>

<version>2.6.11</version>

</dependency>

<dependency>

<groupId>org.mybatis.caches</groupId>

<artifactId>mybatis-ehcache</artifactId>

<version>1.1.0</version>

</dependency>

第二步 在classpath下添加ehcache.xml

<ehcache xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:noNamespaceSchemaLocation="../config/ehcache.xsd">

<!-- 缓存数据要存放的磁盘地址 -->

<diskStore path="F:\develop\ehcache" />

<!-- diskStore：指定数据在磁盘中的存储位置。  defaultCache：当借助CacheManager.add("demoCache")创建Cache时，EhCache便会采用<defalutCache/>指定的的管理策略

以下属性是必须的：  maxElementsInMemory - 在内存中缓存的element的最大数目  maxElementsOnDisk

- 在磁盘上缓存的element的最大数目，若是0表示无穷大  eternal - 设定缓存的elements是否永远不过期。如果为true，则缓存的数据始终有效，如果为false那么还要根据timeToIdleSeconds，timeToLiveSeconds判断

 overflowToDisk - 设定当内存缓存溢出的时候是否将过期的element缓存到磁盘上 以下属性是可选的：  timeToIdleSeconds

- 当缓存在EhCache中的数据前后两次访问的时间超过timeToIdleSeconds的属性取值时，这些数据便会删除，默认值是0,也就是可闲置时间无穷大

 timeToLiveSeconds - 缓存element的有效生命期，默认是0.,也就是element存活时间无穷大 diskSpoolBufferSizeMB

这个参数设置DiskStore(磁盘缓存)的缓存区大小.默认是30MB.每个Cache都应该有自己的一个缓冲区.  diskPersistent

- 在VM重启的时候是否启用磁盘保存EhCache中的数据，默认是false。  diskExpiryThreadIntervalSeconds

- 磁盘缓存的清理线程运行间隔，默认是120秒。每个120s，相应的线程会进行一次EhCache中数据的清理工作  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） -->

<defaultCache maxElementsInMemory="1000"

maxElementsOnDisk="10000000" eternal="false" overflowToDisk="false"

timeToIdleSeconds="120" timeToLiveSeconds="120"

diskExpiryThreadIntervalSeconds="120" memoryStoreEvictionPolicy="LRU">

</defaultCache>

</ehcache>

第三步 mapper映射文件配置cache的type属性

<mapper namespace="com.wwl.dao.DemoDao">

<!-- 开启二级缓存 cache默认使用了二级缓存的type="org.apache.ibatis.cache.impl.PerpetualCache"-->

<cache type="org.mybatis.caches.ehcache.EhcacheCache" />//这里使用EhcacheCache整合包的类

</mapper>

第四步 其它操作看二级缓存的相关操作，请点击[跳转](#_二级缓存)

# 6、逆向工程

根据表结构最后会生成：

持久层：mapper和接口类

实体类：和表对应的实体类和扩招类

Mybatis逆向工程生成的Mapper所进行的操作都是针对单表的

## 添加

插入:只有两个方法，方法传入的参数都是POJO，返回值都是int类型的受影响的行数。

1)insert：会插入所有的信息，如果传入的对象某一属性为空，则插入空,数据库中设置了默认值，默认值就失效了。

2)insertSelective：只会插入含有数据的属性，对于为空的属性，不予以处理

AMapper aMapper = sqlSession.getMapper(AMapper.class);

A a= new A("",2);

//int i = aMapper.insert(a);

int i =aMapper.insertSelective(a);

sqlSession.commit();

sqlSession.close();

注意 如果插入不成功也要看看mysql关系表(外键约束)

## 修改

更新在这里有6个方法，可以分为2组

第一组：根据特定限制条件进行更新

参数1：A  record  ->  要更新的对象

参数2：AExample example  ->  生成一个Criteria对象来设置查询条件

方法1：updateByExample(A  record, AExample example)

 作用：根据特定的限制条件进行更新除了text类型(数据库)的所有列。

sql语句：update a set id = ?, address = ?, b\_id = ? WHERE ( id = ? )会有主键id，一般都不会更新主键

方法2：updateByExampleSelective(A  record, AExample example)这个才实用

作用：根据特定的限制条件更新所有设置了值的列。

最终sql语句：update a SET address = ?, b\_id = ? WHERE ( id = ? )

创建两个实体类：A和扩展类Aexample类

A a= new A("www",2); //要修改的属性值

AExample ae=new AExample();//要修改的where条件

Criteria criteria = ae.createCriteria();//创建内部类的条件类

criteria.andIdEqualTo(7); //设置条件id=7 where id=7

传入两个实体类

aMapper.updateByExample(a, ae);

sqlSession.commit();

sqlSession.close();

方法3：updateByExampleWithBLOBs(A  record, AExample example)

作用：根据特定的限制条件进行更新所有列

第二组：根据ID进行更新

参数：A  record  ->  要更新的对象

方法1：updateByPrimaryKey(A  record)

作用：通过ID更新除了text类型(数据库)的所有列

方法2：updateByPrimaryKeySelective(A  record)

最终sql语句：update a set address = ?, b\_id = ? where id = ?

AMapper aMapper = sqlSession.getMapper(AMapper.class);

A a= new A("sdf",2);

a.setId(7);

aMapper.updateByPrimaryKey(a);

sqlSession.commit();

sqlSession.close();

作用：通过ID更新所有设置了值的列。

 方法3：updateByPrimaryKeyWithBLOBs(A  record)

作用：通过ID进行更新所有列。

## 删除

方法1：根据特定限制条件删除，具体使用的方法和查询的时候是一样的。

方法2：根据主键删除。

## 查询

方法1：selectByExample(AExample example)

返回值：List<A>

作用：通过特定限制条件查询信息，example用于生成一个Criteria对象来设置查询条件

AExample ae=new AExample(); //A类扩展类

Criteria criteria = ae.createCriteria(); //创建条件类对象：有很多的条件方法

criteria.andIdEqualTo(1); //设置类中属性id=1

List<A> list = aMapper.selectByExample(ae);

System.out.println(list);

sqlSession.close();

方法2：selectByExampleWithBLOBs(AExample example)

返回值：List<A>

作用：根据特定限制条件查询，返回值包含类型为text的列(方法一不会查询表类型是text)。example用于生成一个Criteria对象来设置查询条件，使用方法和方法1是一样的，唯一的不同就是返回值是所有列。

方法3：selectByPrimaryKey(Long Id)

返回值：A类

作用：通过主键查询

# 通配符${} #{}

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 ${}。

如下：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';

区别

${}是Properties文件中的变量占位符 在mybatis sql 语句中是字符串拼接，不能防止sql注入.通常用来传表名

select \* from ${tableName}

#{}是sql的参数占位符，Mybatis会将sql中的#{}替换为?号 传入用户输入的参数

select \* from user where id=#{id}

# 核心类

SqlSession的使用范围

SqlSession中封装了对数据库的操作，如：查询、插入、更新、删除等。

SqlSession通过SqlSessionFactory创建。

SqlSessionFactory是通过SqlSessionFactoryBuilder进行创建。

SqlSession是一个面向用户的接口，sqlSession中定义了数据库操作方法。

每个线程都有它自己的SqlSession实例。SqlSession的实例不能共享使用，它是线程不安全的。因此最佳的范围是请求或方法范围。绝对不能将SqlSession实例的引用放在一个类的静态字段或实例字段中。

打开一个 SqlSession；使用完毕就要关闭它。通常把这个关闭操作放到 finally 块中以确保每次都能执行关闭。如下：

SqlSession session = sqlSessionFactory.openSession();

try {

// do work

} finally {

session.close();

}

SqlSessionFactoryBuilder 可以作为工具类，创建完就不需要了

SqlSessionFactoryBuilder用于创建SqlSessionFacoty，SqlSessionFacoty一旦创建完成就不需要SqlSessionFactoryBuilder了，因为SqlSession是通过SqlSessionFactory创建的。所以可以将SqlSessionFactoryBuilder当成一个工具类使用，最佳使用范围是方法范围即方法体内局部变量。

SqlSessionFactory 适用范围运行期间、重复使用、设置为单例

SqlSessionFactory是一个接口，接口中定义了openSession的不同重载方法，SqlSessionFactory的最佳使用范围是整个应用运行期间，一旦创建后可以重复使用，通常以单例模式管理SqlSessionFactory。

# 分页插件

第一步 导包 mybatis-3.4.3.jar jsqlparser-0.9.4.jar pagehelper-4.1.1.jar

注意了：mybatis版本问题，测试了很久才发现这个这三个版本合起来可以用

初始化 dialect [mysql]时出错:null

那就是sql解析工具--jsqlparser.jar4.1.0及以后版本需要0.9.4版本

http://repo1.maven.org/maven2/com/github/jsqlparser/jsqlparser/0.9.4/

4.1.0以前版本需要0.9.1版本

http://repo1.maven.org/maven2/com/github/jsqlparser/jsqlparser/0.9.1/

<https://blog.csdn.net/weixin_41382994/article/details/80205865?utm_source=copy>

第二步 配置 mybatis.config.xml

<!-- —1、配置别名： 包其子包下所有类有效，作用：第三步mapper.xml下parameterType、resultType不用写全路径类名-->

<typeAliases>

<package name="com.wwl.pojo"/>

</typeAliases>

<!-- 配置分页插件 -->

<plugins>

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

<!-- 设置数据库类型 Oracle,Mysql,MariaDB,SQLite,Hsqldb,PostgreSQL六种数据库-->

<property name="dialect" value="mysql"/>

</plugin>

</plugins>

-------------------------------------------------网上找到的-------------------------------------------------------------------------------------------

<plugins>

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

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

<!-- 4.0.0以后版本可以不设置该参数 -->

<property name="dialect" value="mysql"/>

<!-- 该参数默认为false -->

<!-- 设置为true时，会将RowBounds第一个参数offset当成pageNum页码使用 -->

<!-- 和startPage中的pageNum效果一样-->

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

<!-- 该参数默认为false -->

<!-- 设置为true时，使用RowBounds分页会进行count查询 -->

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

<!-- 设置为true时，如果pageSize=0或者RowBounds.limit = 0就会查询出全部的结果 -->

<!-- （相当于没有执行分页查询，但是返回结果仍然是Page类型）-->

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

<!-- 3.3.0版本可用 - 分页参数合理化，默认false禁用 -->

<!-- 启用合理化时，如果pageNum<1会查询第一页，如果pageNum>pages会查询最后一页 -->

<!-- 禁用合理化时，如果pageNum<1或pageNum>pages会返回空数据 -->

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

<!-- 3.5.0版本可用 - 为了支持startPage(Object params)方法 -->

<!-- 增加了一个`params`参数来配置参数映射，用于从Map或ServletRequest中取值 -->

<!-- 可以配置pageNum,pageSize,count,pageSizeZero,reasonable,orderBy,不配置映射的用默认值 -->

<!-- 不理解该含义的前提下，不要随便复制该配置 -->

<property name="params" value="pageNum=pageHelperStart;pageSize=pageHelperRows;"/>

<!-- 支持通过Mapper接口参数来传递分页参数 -->

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

<!-- always总是返回PageInfo类型,check检查返回类型是否为PageInfo,none返回Page -->

<property name="returnPageInfo" value="none"/>

</plugin>

</plugins>

第三步 代码中的使用

先按没有分页的正常查询：如下面模糊查询，它是先查询完后才给你分页的

把分页步骤部署上去：参数分页的两个参数pageNo,pageSize

//分页插件的使用:共两步，第一步

PageHelper.startPage(Integer.parseInt(pageNo),Integer.parseInt(pageSize));

UsersExample example=new UsersExample();

Criteria criteria = example.createCriteria();

//设置模糊查询username

if(username ==null) {

username="%%";

}else {

username="%"+username+"%";

}

criteria.andUsernameLike(username);

List<Users> list = usersMapper.selectByExample(example);

if(!Judge.isNullAndEmpty(list)) {

//分页插件的使用:共两步，第二步,把模糊查询的结果当参数放进去

PageInfo<Users> pageInfo = new PageInfo<>(list);

List<Users> rlist = pageInfo.getList();

customResult.setListData(rlist);

customResult.setStatus(200);

}else {

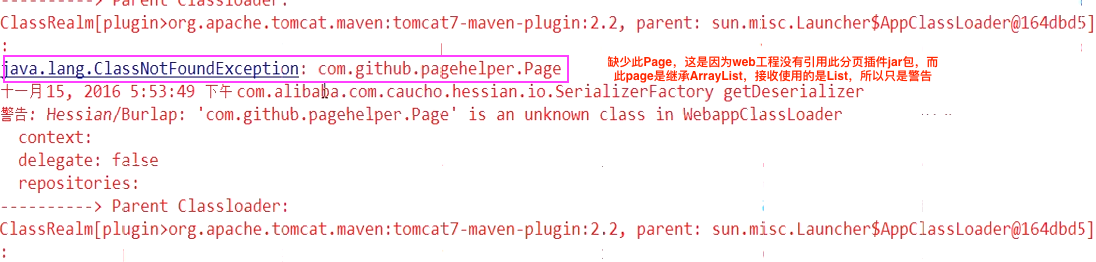
customResult.setStatus(404);

}

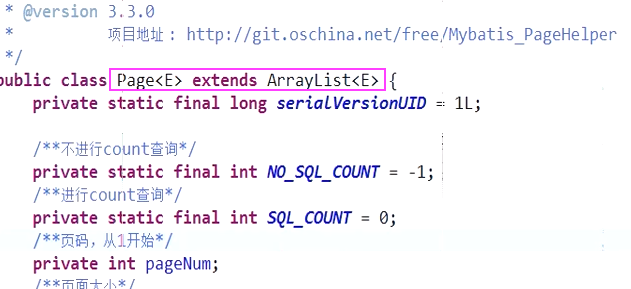
return customResult;

## 异常问题

在32期第二天使用分页插件出现警告问题

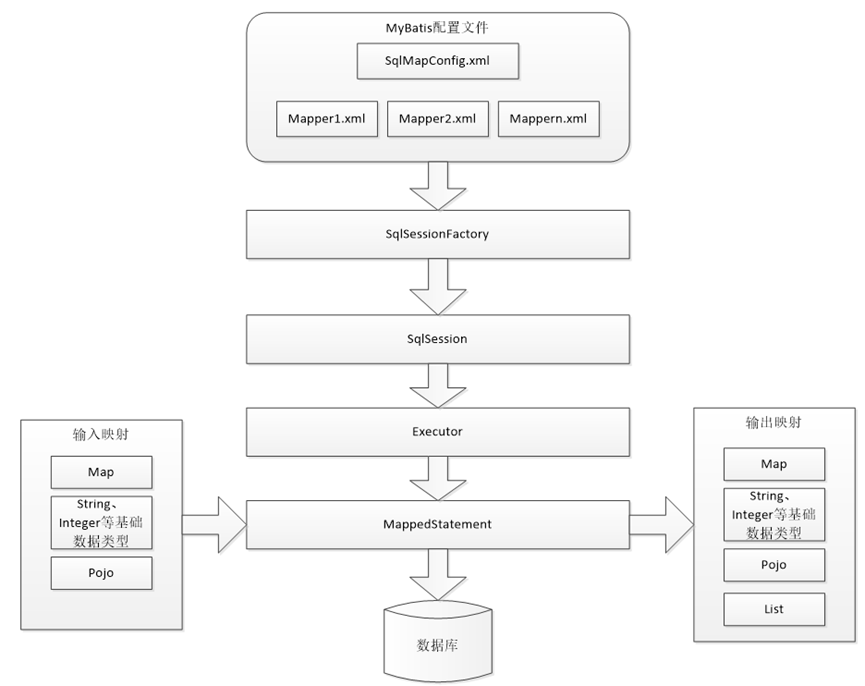


查看源码可得知：此缺少的包是继承了ArrayList的，web工程没有引入



# 拦截器

# 结构体系



# 其它

## Eclipse引入本地DTD文件

<!DOCTYPE mapper SYSTEM "F:\学习资料\MyBatis\mybatis-3-mapper.dtd">

要引入本地的：SYSTEM（引入网络的：PUBLIC，）”绝对地址”

Eclipse操作

windows –>preferences–>搜索xml–>xml catalog–> add–>fileSystem找到的DTD文件–>key选择:URI–>key:就是绝对地址–>保存

创建

右键工程–>new–>选择xml–>下一步选择DTD–>选择selectXML–>拉到最后选中自己DTD–>保存

## MySQL :after和before

after :自增长用：如先把数据插入数据库，然后再主键生成

before：先用UUID生成主键ID，再把数据插入数据库

select UUID() MySQL内置函数

## CDATA的使用

CDATA 部件以"<![CDATA[" 标记开始，以"]]>"标记结束。在"<![CDATA["和"]]>"之间 的特殊字符的意义都不起作用，而转变为普通字符串内容。

在 MyBatis 的 XML 映射语句配置文件中，如果 SQL 语句有特殊字符，使用CDTA 部件括起来，如：

<select id= "selectBlog\_use\_collection"

resultMap= "blogResult" >

<![CDATA[ SELECT id , title, author\_id as authored FROM BLOG WHERE ID > 0 and ID < 10 ]]>

## 实体类属性名和表名不一致

1、在sql语句中使用别名

如User 的属性名userId 表中的是user\_id

Select user\_id as userId,age,name from user

好多东西不要搞混了，这明明就是mysql的知识啊，使用别名查出来不就是显示别名的

2、使用ResultMap跟表一一对应

1、只是单表查询(为了解决名称不一致问题)、只写不一致的即可

2、但是涉及到关联表查询，必须要写全属性

<resultMap type="com.wwl.entity.user" id="userMap">

<id column="user\_id" property="userId"/>

<result column="age" property="age"/> //只要是单表查询不是关联查询

</resultMap>

3、在核心配置中使用配置

<!-- 开启驼峰命名转换:Table{create\_time} -> Entity{createTime} -->

<setting name="mapUnderscoreToCamelCase" value="true" /> <!-- 这个实用-->