# ***Mybatis***

## ***原生态JDBC程序中存在的问题***

public class Demo {

@Test

public void fun(){

//数据库连接

Connection connection = null;

//预编译Statement，可以提高数据库性能

PreparedStatement preparedStatement = null;

//结果集

ResultSet resultSet = null;

try {

//加载数据库驱动

Class.forName("com.mysql.jdbc.Driver");

//通过驱动管理类获取数据库链接

connection = DriverManager.getConnection("jdbc:mysql://localhost:3306/mybatis?characterEncoding=utf-8", "root", "123");

//使用占位符定义sql模板

String sql = "select \* from user where username = ?";

//获取预处理statement

preparedStatement = connection.prepareStatement(sql);

//设置参数

preparedStatement.setString(1, "王五");

//向数据库发出sql执行查询，查询出结果集

resultSet = preparedStatement.executeQuery();

//遍历查询结果集

while(resultSet.next()){

System.out.println(resultSet.getString("id")+" "+resultSet.getString("username"));

}

} catch (Exception e) {

e.printStackTrace();

}finally{

//释放资源

try {

if(resultSet != null)resultSet.close();

if(preparedStatement != null) preparedStatement.close();

if(connection != null) connection.close();

} catch (SQLException e) {

throw new RuntimeException(e);

}

}

}

}

1.频繁创建关闭连接，资源浪费！ 2.sql语句硬编码在java代码中，不利于项目维护

3.占位符位置和参数值硬编码在java代码中，不利于项目维护 4.遍历结果集数据时，存在硬编码，不利于项目维护

## ***Mybatis概述***

1.mybatis是一个持久层框架，是apache下的顶级项目。先被托管到goolecode，后来又被托管到github

2.mybatis使程序员将主要精力放在sql语句上，通过mybatis提供的映射方式可以灵活地生成满足需要的半自动化sql语句

3.mybatis可以将输入的参数自动进行输入映射，并将查询结果集灵活地映射成java对象，也就是输出映射

4.下载地址：<https://github.com/mybatis/mybatis-3/releases>

5.hibernate是一个标准ORM框架，sql语句是自动生成的，所以对sql语句进行优化或修改比较困难，适用于需求变化不多的中小型项目

mybatis是一个不完全的ORM框架，由程序员编写sql语句，也可以实现映射（输出映射，输入映射），对sql语句进行优化或修改比较方便，适用于需求变化较多的项目

6.mybatis的执行流程：

SqlMapConfig.xml（mybatis的全局配置文件，名称固定）

配置了数据源、事务等mybatis运行环境

配置了很多映射文件mapper.xml（配置sql语句）

SqlSessionFactory，根据配置文件创建的工厂，用于创建SqlSession

SqlSession，面向程序员的一个接口，可以通过执行器操作数据库

Executor（执行器），是一个接口，它有两个实现类（基本执行器、缓存执行器）

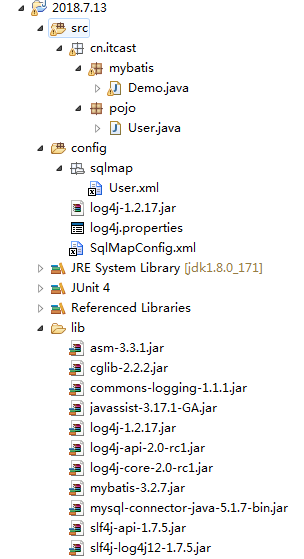
mapped statement（底层封装对象），封装了操作数据库的数据，包括 sql语句、输入参数类型、输出结果类型

输入参数类型和输出参数类型可以是：java简单类型、hashmap、pojo自定义类型

数据库

## Mybatis入门程序

### 第一步：导入 mybatis核心jar包及其依赖jar包、数据库驱动jar包、添加log4j日志文件



log4j.properties

log4j.rootLogger=DEBUG, stdout

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

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

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

SqlMapConfig.xml

<?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>

<!-- 和spring整合后 environments配置将被废除-->

<environments default="development">

<environment id="development">

<!--由mybatis管理的jdbc事务管理-->

<transactionManager type="JDBC" />

<!-- 由mybatis管理的数据库连接池-->

<dataSource type="POOLED">

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

<property name="url" value="jdbc:mysql://localhost:3306/mybatis?characterEncoding=utf-8" />

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

<property name="password" value="123" />

</dataSource>

</environment>

</environments>

</configuration>

### 第二步：创建pojo类

public class User implements Serializable {

private int id;

private String username;

private String sex;

private Date birthday;

private String address;

public int getId() {

return id;

}

public void setId(int id) {

this.id = id;

}

public String getUsername() {

return username;

}

public void setUsername(String username) {

this.username = username;

}

public String getSex() {

return sex;

}

public void setSex(String sex) {

this.sex = sex;

}

public Date getBirthday() {

return birthday;

}

public void setBirthday(Date birthday) {

this.birthday = birthday;

}

public String getAddress() {

return address;

}

public void setAddress(String address) {

this.address = address;

}

@Override

public String toString() {

return "User [id=" + id + ", username=" + username + ", sex=" + sex + ", birthday=" + birthday + ", address=" + address + "]";

}

}

### 第三步：配置pojo类对应的映射文件User.xml，在mapper代理开发时，需要命名为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">

<!--namespace命名空间可以对sql进行分类管理，也就是sql隔离，在mapper代理中，namespace还有特殊且重要的作用-->

<mapper namespace="test">

<!--id：映射文件中可以配置很多sql语句，所以id就是sql语句的标识，sql语句的数据信息会被封装到mappedStatement对象中，所以将id也称为statementId-->

<!--parameterType：输入参数的类型-->

<!--resultType：输出结果的类型，单条！-->

<!--#{}：占位符，用来接收输入参数，对于输入参数类型是简单类型，占位符中的参数可以任意的-->

<select id="findUserById" parameterType="int" resultType="cn.itcast.pojo.User">

SELECT \* FROM user Where id = #{id}

</select>

<!--${}：拼接字符串，可能遭遇sql攻击，不建议使用，对于输入参数类型是简单类型，其传入的参数只能是value-->

<select id="findUserByUsername" parameterType="java.lang.String" resultType="cn.itcast.pojo.User">

SELECT \* FROM user Where username Like '%${value}%'

</select>

<insert id="insertUser" parameterType="cn.itcast.pojo.User">

<!--自增主键返回，返回到插入的对象中，不设置默认返回默认值给对象-->

<!--SELECT LAST\_INSERT\_ID()：刚插入数据的自增主键值-->

<!--keyProperty：指定将主键值设置给parameterType类型对象的哪个属性-->

<!--order：相对sql语句返回主键的执行顺序，自增是先插入数据后才有主键值-->

<!--resultType：指定主键的输出结果类型-->

<selectKey keyProperty="id" order="AFTER" resultType="java.lang.Integer">

SELECT LAST\_INSERT\_ID()

</selectKey>

<!--#{}中指定pojo的属性名，用来接收pojo对象的属性值，mybatis会通过OGNL获取对象的属性值-->

INSERT INTO user(username,birthday,sex,address) VALUES(#{username},#{birthday},#{sex},#{address})

<!--非自增主键返回，例如SELECT uuid()，需要在sql语句中添加"id和#{id}"，并且相对sql语句返回主键的执行顺序为BEFORE-->

</insert>

<!--可以不需要输出结果类型-->

<delete id="deleteUser" parameterType="java.lang.Integer">

DELETE FROM user WHERE id=#{id}

</delete>

<update id="updateUser" parameterType="cn.itcast.pojo.User">

UPDATE user SET username=#{username},birthday=#{birthday},sex=#{sex},address=#{address} WHERE id=#{id}

</update>

</mapper>

### 第四步：在SqlMapConfig.xml中加载映射文件

</environments>

<mappers>

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

</mappers>

</configuration>

### 第五步：测试代码

public class Demo {

@Test

public void fun() {

SqlSession sqlSession = null;

try {

//根据配置文件创建的工厂

SqlSessionFactory sqlSessionFactory = new SqlSessionFactoryBuilder().build(Resources.getResourceAsStream("SqlMapConfig.xml"));

//通过工厂创建会话

sqlSession = sqlSessionFactory.openSession();

//1.查询，第一个参数值等于"namespace + '.' + statementId"，第二个参数指定sql语句中parameterTyp的参数值，使用selectOne()方法查询出多条语句时会报错

User user = sqlSession.selectOne("test.findUserById", 10);

List<User> list = sqlSession.selectList("test.findUserByUsername", "小明");

System.out.println(user);

System.out.println(list);

//2.添加，可以不为对象设置数据库表中的自增主键

user = new User();

user.setUsername("李小明");

user.setBirthday(new Date());

user.setSex("1");

user.setAddress("四川成都");

sqlSession.insert("test.insertUser", user);

//删除

sqlSession.delete("deleteUser",28);

//修改

user = new User();

user.setId(10);

user.setUsername("李大明");

user.setBirthday(new Date());

user.setSex("2");

user.setAddress("成都温江");

sqlSession.update("updateUser", user);

//添加、删除、修改操作需要提交事务

sqlSession.commit();

} catch (Exception e) {

e.printStackTrace();

} finally {

if (sqlSession != null)sqlSession.close();

}

}

}

## Mybatis开发Dao的两种方法

### 1.原始（Ibatis）

SqlSessionFactory一旦创建，就一直使用，所以在mybatis和spring整合时，可以使用单例模式管理SqlSessionFactory，

进而，可以把SqlSessionFactoryBuilder当做一个工具类，在需要创建SqlSessionFactory时候，只需要new一次SqlSessionFactoryBuilder即可

SqlSession是一个提供了很多操作数据库的方法的接口，因为在其实现类中还有数据域属性，导致其线程不安全，所以其应该使用在方法体中

public interface UserDao {

public User findUserById(int id) throws Exception;

public List<User> findUserByUsername(String name) throws Exception;

public void insertUser(User user) throws Exception;

public void deleteUser(int id) throws Exception;

public void updateUser(User user) throws Exception;

}

public class UserDaoImpl implements UserDao {

private SqlSessionFactory sqlSessionFactory;

public UserDaoImpl(SqlSessionFactory sqlSessionFactory) {

this.sqlSessionFactory = sqlSessionFactory;

}

@Override

public User findUserById(int id) throws Exception {

SqlSession sqlSession = sqlSessionFactory.openSession();

User user = sqlSession.selectOne("test.findUserById", id);

sqlSession.close();

return user;

}

@Override

public List<User> findUserByUsername(String name) throws Exception {

SqlSession sqlSession = sqlSessionFactory.openSession();

List<User> list = sqlSession.selectList("test.findUserByUsername", name);

sqlSession.close();

return list;

}

@Override

public void insertUser(User user) throws Exception {

SqlSession sqlSession = sqlSessionFactory.openSession();

sqlSession.insert("test.insertUser", user);

sqlSession.commit();

sqlSession.close();

}

@Override

public void deleteUser(int id) throws Exception {

SqlSession sqlSession = sqlSessionFactory.openSession();

sqlSession.delete("test.deleteUser", id);

sqlSession.commit();

sqlSession.close();

}

@Override

public void updateUser(User user) throws Exception {

SqlSession sqlSession = sqlSessionFactory.openSession();

sqlSession.update("test.updateUser", user);

sqlSession.commit();

sqlSession.close();

}

}

public class Demo {

private UserDao userDao;

@Before

public void setSqlSessionFactory() throws IOException{

userDao = new UserDaoImpl(new SqlSessionFactoryBuilder().build(Resources.getResourceAsStream("SqlMapConfig.xml")));

}

@Test

public void testFindUserById() throws Exception {

System.out.println(userDao.findUserById(1));

}

@Test

public void testfindUserByUsername() throws Exception {

System.out.println(userDao.findUserByUsername("小明"));

}

@Test

public void testInsertUser() throws Exception {

User user = new User();

user.setUsername("赵六");

user.setBirthday(new Date());

user.setSex("2");

user.setAddress("成都温江");

userDao.insertUser(user);

}

@Test

public void testDeleteUser() throws Exception {

userDao.deleteUser(32);

}

@Test

public void testUpdateUser() throws Exception {

User user = new User();

user.setId(1);

user.setUsername("阿龙");

user.setBirthday(new Date());

user.setSex("1");

user.setAddress("成都简阳");

userDao.updateUser(user);

}

}

原始Dao开发方法需要写Dao接口和Dao实现类，并在Dao实现类中注入SqlSessionFactory

存在的问题：1.Dao接口实现类方法中存在大量重复代码

2.调用SqlSession方法时将statementId硬编码了

3.由于SqlSession方法使用泛型，即使变量类型传入错误，在编译阶段也不报错，不利于程序员开发

### 2.Mapper代理

Mapper代理Dao开发方法只需要写Mapper接口，它相当于Dao接口

开发规范： 1.映射文件命名为xxxMapper.xml，并且其中namespace属性值为接口类路径

2.Mapper接口中方法的名称必须与statementId一致

3.Mapper接口中方法的参数类型必须和parameterType一致

4.Mapper接口中方法的返回值类型必须和resultType一致

public interface UserMapper {

//如果方法返回单个对象，代理对象自动调用selectOne()方法

public User findUserById(int id) throws Exception;

//如果方法返回多个对象，代理对象自动调用selectList()方法

public List<User> findUserByUsername(String name) throws Exception;

public void insertUser(User user) throws Exception;

public void deleteUser(int id) throws Exception;

public void updateUser(User user) throws Exception;

}

public class Demo {

private SqlSession sqlSession;

private UserMapper userMapper;

@Before

public void setSqlSessionFactory() throws IOException{

sqlSession = new SqlSessionFactoryBuilder().build(Resources.getResourceAsStream("SqlMapConfig.xml")).openSession();

userMapper = sqlSession.getMapper(UserMapper.class);

}

@Test

public void testFindUserById() throws Exception {

System.out.println(userMapper.findUserById(1));

sqlSession.close();

}

@Test

public void testfindUserByUsername() throws Exception {

System.out.println(userMapper.findUserByUsername("小明"));

sqlSession.close();

}

@Test

public void testInsertUser() throws Exception {

User user = new User();

user.setUsername("赵雷");

user.setBirthday(new Date());

user.setSex("2");

user.setAddress("成都温江");

userMapper.insertUser(user);

sqlSession.commit();

sqlSession.close();

}

@Test

public void testDeleteUser() throws Exception {

userMapper.deleteUser(38);

sqlSession.commit();

sqlSession.close();

}

@Test

public void testUpdateUser() throws Exception {

User user = new User();

user.setId(1);

user.setUsername("阿飞");

user.setBirthday(new Date());

user.setSex("1");

user.setAddress("成都简阳");

userMapper.updateUser(user);

sqlSession.commit();

sqlSession.close();

}

}

## SqlMapConfig.xml

### Properties（属性）

可以用来引入配置文件，方便对信息统一管理，还可以定义属性

db.properties文件

jdbc.driver=com.mysql.jdbc.Driver

jdbc.url=jdbc:mysql://localhost:3306/mybatis?characterEncoding=utf-8

jdbc.username=root

jdbc.password=123

使用resource属性或url属性引入db.properties文件、定义属性，并加载db.properties文件

<configuration>

<properties resource="db.properties">

<property name="xxx" value="XXX"></property>

</properties>

<environments default="development">

<environment id="development">

<transactionManager type="JDBC" />

<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>

注意！MyBatis按照顺序加载属性，同名属性的属性值会被覆盖：在properties元素体内定义的属性首先被读取

由properties元素引入文件中的属性  由parameterType传递的属性

### Settings（全局参数）

可以配置mybatis全局参数来影响mybatis的运行行为







### typeAliases（别名）

mybatis提供了别名，也可以自定义别名

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

<typeAliases>

<!--单个别名定义，type指类路径，alias指别名-->

<typeAlias alias="user" type="cn.itcast.mybatis.po.User"/>

<!--批量别名定义，包中所有类的别名都为类名的首字母大写或小写，这种方式不允许指定范围内存在同名类，扩大范围只能写多个package或减少包的层级，不能使用通配符-->

<package name="cn.itcast.pojo"/>

<package name="cn.itcast.dao"/>

</typeAliases>

### mappers（映射器）

<mappers>

<!--加载相对类路径下的映射文件-->

<mapper resource="cn/itcast/mapper/UserMapper.xml" />

<!--加载完全限定路径的映射文件-->

<mapper url="file:///F:/development/workspaces/2018.7.13/cn/itcast/mapper/UserMapper.xml" />

<!--通过Mapper接口加载映射文件，要求映射文件和Mapper接口在同一个包下，且名称相同-->

<mapper class="cn.itcast.mapper.UserMapper"/>

<!—扫描包中所有Mapper接口并加载对应的映射文件，同样要求映射文件和Mapper接口在同一个包下，且名称相同-->

<package name="cn.itcast.mapper"/>

</mappers>

## 输入映射

pojo包装类：parameterType只有一个，当一个pojo简单类型包含的属性不足以满足输入的条件时，可以写一个类来包含所有需要的输入参数，这就是pojo包装类。

需要先定义一个pojo扩展类，其通过继承得到pojo简单类属性，再封装扩展数据。

pojo包装类则只需通过封装多个扩展类即可，再将其设置为parameterType，就可以通过OGNL找到需要的输入参数

当parameterType使用hashmap类型时，需要给hashmap对象添加查询数据的键值对，key为查询的列名称，value为指定查询的数据

## 输出映射

### resultType

对于pojo类型，查询出来的列名必须和pojo中的属性名一致，才能映射成功，并且只会映射需要查询的内容

只要查询出来的列名和pojo的属性有一个一致，就会创建pojo对象

只有查询出来的列名和pojo的属性名全部不一致，才不会创建pojo对象，但是注意！对于查询多条记录，因为查询到了数据，返回的集合的长度仍然存在，

对于hashmap类型，输出的结果是hashmap对象，其key为查询的列名称，value为查询出的数据

### 2.resultMap

对于pojo类型，如果查询出来的列名与pojo中的属性名不一致，还可以使用resultMap来完成映射

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

<!--需要先定义resultMap，给定pojo类路径和其唯一标识-->

<resultMap type="user" id="userResultMap">

<!—id用于指定查询列中的唯一标识，如果多个列共同组成一个唯一标识，那么需要配多个id-->

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

<!--对应普通列-->

<result column="\_username" property="username"/>

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

</resultMap>

<!--使用resultMap，对于本命空间中定义的resultMap，"namespace."可以省略不些-->

<select id="findUserByUsername" parameterType="string" resultMap="userResultMap">

SELECT id \_id,username \_username,birthday \_birthday FROM user Where username Like '%${value}%'

</select>

</mapper>

## 动态Sql

根据parameterType对象的属性是否存在，动态添加条件语句

<?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="cn.itcast.mapper.UserMapper">

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

SELECT \* FROM user WHERE username = #{username} AND sex = #{sex}

<where>

<if test="username != null">

AND username = #{username}

</if>

<if test="sex != null">

AND sex = #{sex}

</if>

</where>

</select>

</mapper>

对于大量重复使用的sql片段，可以将其抽取出来，在需要使用时引用其即可

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

<!--定义sql片段，需要给定唯一标识-->

<sql id="query\_user\_where">

<if test="username != null">

AND username = #{username}

</if>

<if test="sex != null">

AND sex = #{sex}

</if>

</sql>

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

SELECT \* FROM user

<where>

<include refid="query\_user\_where"/>

</where>

</select>

</mapper>

对于IN和OR这种sql片段，可以使用foreach对其遍历

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

<sql id="query\_user\_or">

<!--这里的ids是pojo中的一个集合成员变量-->

<if test="ids!=null">

<!--collection指需要遍历的集合，item指临时变量，open和close指遍历开始前和结束后需要添加的sql片段，separator指遍历间添加的sql片段-->

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

#{id}

</foreach>

</if>

</sql>

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

SELECT \* FROM user

<where>

<include refid="query\_user\_or"/>

</where>

</select>

</mapper>

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

<sql id="query\_user\_or">

<if test="ids!=null">

<foreach collection="ids" item="id" open="id=" separator="OR" >

#{id}

</foreach>

</if>

</sql>

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

SELECT \* FROM user

<where>

<include refid="query\_user\_or"/>

</where>

</select>

</mapper>

## 高级映射

### 1.一对一查询

#### 1.使用resultType

1. 创建pojo扩展类

public class OrdersCustom extends Orders{

private String username;

private String sex;

private String address;

public String getUsername() {

return username;

}

public void setUsername(String username) {

this.username = username;

}

public String getSex() {

return sex;

}

public void setSex(String sex) {

this.sex = sex;

}

public String getAddress() {

return address;

}

public void setAddress(String address) {

this.address = address;

}

}

1. 配置sql语句

<mapper namespace="cn.itcast.mapper.OrdersMapperCustom">

<select id="selectOrdersCustoms" resultType="ordersCustom">

SELECT o.\*,u.username,u.sex,u.address FROM orders o,USER u WHERE o.user\_id=u.id

</select>

</mapper>

3.创建Mapper接口

public interface OrdersMapperCustom {

public List<OrdersCustom> selectOrdersCustoms() throws Exception;

}

4.测试代码

public class Demo {

@Test

public void testSelectOrdersCustoms() throws Exception {

SqlSession sqlSession = new SqlSessionFactoryBuilder().build(Resources.getResourceAsStream("SqlMapConfig.xml")).openSession();

System.out.println(sqlSession.getMapper(OrdersMapperCustom.class).selectOrdersCustoms());

sqlSession.close();

}

}

#### 2.使用resultMap

1. 在Orders类中封装User属性，将关联查询出来的用户信息直接封装到orders对象的user属性中

private User user;

public User getUser() {

return user;

}

public void setUser(User user) {

this.user = user;

}

2.配置sql语句

<mapper namespace="cn.itcast.mapper.OrdersMapper">

<resultMap type="orders" id="ordersUserResultMap">

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

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

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

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

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

<!—association可以将关联查询信息映射到一个pojo对象中，javaType指定pojo类路径-->

<association property="user" javaType="user">

<!--id用于指定关联查询对象的唯一标识 -->

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

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

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

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

</association>

</resultMap>

<select id="selectOrders" resultMap="ordersUserResultMap">

SELECT o.\*,u.username,u.sex,u.address FROM orders o,USER u WHERE o.user\_id=u.id

</select>

</mapper>

3.创建Mapper接口

public interface OrderssMapper {

public List<Orders> selectOrders() throws Exception;

}

4.测试代码

public class Demo {

@Test

public void testSelectOrdersCustoms() throws Exception {

SqlSession sqlSession = new SqlSessionFactoryBuilder().build(Resources.getResourceAsStream("SqlMapConfig.xml")).openSession();

System.out.println(sqlSession.getMapper(OrdersMapper.class).selectOrders());

sqlSession.close();

}

}

### 2.一对多查询

#### 1.使用resultType

使用resultType实现一对多与实现一对一无任何区别

#### 2.使用resultMap

1.在Orders类中封装Orderdetails属性，将关联查询出来的所有订单明细直接封装到orders对象的List<Orderdetail>属性中

private List<Orderdetail> orderdetails;

public List<Orderdetail> getOrderdetails() {

return orderdetails;

}

public void setOrderdetails(List<Orderdetail> orderdetails) {

this.orderdetails = orderdetails;

}

2.配置sql语句

<mapper namespace="cn.itcast.mapper.OrdersMapper">

<resultMap type="orders" id="ordersUserResultMap">

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

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

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

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

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

<association property="user" javaType="user">

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

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

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

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

</association>

</resultMap>

<!--这里使用了extends，相当于将ordersUserResultMap中的内容写到了这个resultMap中的最前面-->

<resultMap type="orders" id="ordersUserOrderdetailResultMap" extends="ordersUserResultMap">

<!—collection可以将关联查询信息映射到一个集合中，ofType指定集合中的pojo类路径-->

<collection property="orderdetails" ofType="orderdetail">

<id column="orderdetail\_id" property="id"/>

<result column="items\_num" property="itemsNum"/>

</collection>

</resultMap>

<select id="selectOrders" resultMap="ordersUserOrderdetailResultMap">

SELECT o.\*,u.username,u.sex,u.address,s.id orderdetail\_id,s.items\_num FROM orders o,USER u,orderdetail s WHERE o.user\_id=u.id AND o.id=s.orders\_id

</select>

</mapper>

3.创建Mapper接口

public interface OrdersMapper {

public List<Orders> selectOrders() throws Exception;

}

4.测试代码

public class Demo {

@Test

public void testSelectOrdersCustoms() throws Exception {

SqlSession sqlSession = new SqlSessionFactoryBuilder().build(Resources.getResourceAsStream("SqlMapConfig.xml")).openSession();

System.out.println(sqlSession.getMapper(OrdersMapper.class).selectOrders());

sqlSession.close();

}

}

### 3.一对多查询

#### 1.使用resultType

使用resultType实现多对多与实现一对一无任何区别

#### 2.使用resultMap

1. 在User类中封装ordersList属性，将关联查询出来的所有订单直接封装到user对象的List<Orders>属性中

在Orders类中封装Orderdetails属性，将关联查询出来的所有订单明细直接封装到orders对象的List<Orderdetail>属性中

在Orderdetail类中封装items属性，将关联查询出来的商品直接封装到orderdetail对象的items属性中

private List<Orders> ordersList;

public List<Orders> getOrdersList() {

return ordersList;

}

public void setOrdersList(List<Orders> ordersList) {

this.ordersList = ordersList;

}

private List<Orderdetail> orderdetails;

public List<Orderdetail> getOrderdetails() {

return orderdetails;

}

public void setOrderdetails(List<Orderdetail> orderdetails) {

this.orderdetails = orderdetails;

}

private Items items;

public Items getItems() {

return items;

}

public void setItems(Items items) {

this.items = items;

}

2.配置sql语句

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

<resultMap type="orders" id="ordersUserResultMap">

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

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

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

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

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

<association property="user" javaType="user">

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

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

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

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

</association>

</resultMap>

<resultMap type="user" id="userOrdersOrderdetailItemsResultMap">

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

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

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

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

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

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

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

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

<collection property="orderdetails" ofType="orderdetail">

<id column="orderdetail\_id" property="id"/>

<result column="items\_num" property="itemsNum"/>

<association property="items" javaType="items">

<id column="items\_id" property="id"/>

<result column="price" property="pic"/>

</association>

</collection>

</collection>

</resultMap>

<select id="selectUsers" resultMap="userOrdersOrderdetailItemsResultMap">

SELECT o.\*,u.username,s.id orderdetail\_id,s.items\_id,s.items\_num,i.price FROM orders o,USER u,orderdetail s,items i WHERE o.user\_id=u.id AND o.id=s.orders\_id AND i.id=s.items\_id

</select>

</mapper>

3.创建Mapper接口

public interface UserMapper {

public List<User> selectUsers() throws Exception;

}

4.测试代码

public class Demo {

@Test

public void testSelectOrdersCustoms() throws Exception {

SqlSession sqlSession = new SqlSessionFactoryBuilder().build(Resources.getResourceAsStream("SqlMapConfig.xml")).openSession();

System.out.println(sqlSession.getMapper(OrdersMapper.class).selectUsers());

sqlSession.close();

}

}

### 3.小结

resultType实现关联查询较为简单，如果pojo中没有属性对应需要查询出来的列，只需定义pojo扩展类或包装类，即可完成映射

resultMap则较为复杂，但可以将关联查询信息映射到属性中

## 延迟加载

1.延迟加载：先查询简单信息、在需要时进一步关联查询所需要的信息，能够大大提高数据库性能，这就是延迟记在

使用resultMap的association和collection可以实现一对一和一对多的延迟加载，也可以在service层中根据需求来判断以实现延迟加载

2.使用resultMap实现延迟加载

1.配置延迟加载

<configuration>

<properties resource="db.properties"/>

<settings>

<!--打开延迟加载-->

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

<!--将积极加载更改为消极加载，即按需加载-->

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

</settings>

<typeAliases>

2.在Orders类中封装User属性，将关联查询出来的用户信息直接封装到orders对象的user属性中

private User user;

public User getUser() {

return user;

}

public void setUser(User user) {

this.user = user;

}

3.配置sql语句

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

<select id="findUserById" parameterType="\_int" resultType="user">

SELECT \* FROM USER WHERE ID=#{id}

</select>

</mapper>

<mapper namespace="cn.itcast.mapper.OrdersMapper">

<resultMap type="orders" id="OrdersLazyLoadingUserResultMap">

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

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

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

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

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

<!--select用来指定延迟加载是需要执行的sql语句的statementId，column指定关联查询时需要的条件列-->

<association property="user" javaType="user" select="cn.itcast.mapper.UserMapper.findUserById" column="user\_id"/>

</resultMap>

<select id="selectOrdersLazyLoadingUser" resultMap="OrdersLazyLoadingUserResultMap">

SELECT \* FROM Orders

</select>

</mapper>

1. 创建Mapper接口

public interface UserMapper {

public User findUserById(int id) throws Exception;

}

4.测试代码

public class Demo {

@Test

public void testSelectOrdersCustoms() throws Exception {

SqlSession sqlSession = new SqlSessionFactoryBuilder().build(Resources.getResourceAsStream("SqlMapConfig.xml")).openSession();

for(Orders orders: sqlSession.getMapper(OrdersMapper.class).selectOrdersLazyLoadingUser()){

System.out.println(orders.getUser());

}

sqlSession.close();

}

}

## 查询缓存

### 1.查询缓存

Mybatis提供了一级缓存，和二级缓存来减轻数据压力，提高数据库性能，Mybatis默认支持一级缓存，但二级缓存需要通过配置开启

### 2.一级缓存

1. 一级缓存是SqlSession级别的缓存，在操作数据库时需要创建SqlSession，其内部有一个HashMap数据结构用于存储缓存数据，多个SqlSession之间缓存数据区域分隔

2.查询数据时，会先去一级缓存中看否存在对应的数据，如果没有，再去数据库查询，找到后返回数据的同时还会将数据保存到一级缓存中

但如果有，就直接从一级缓存返回数据

3.注意！一旦执行commit操作，一级缓存就会被清空，以保证查询到最新的数据，关闭SqlSession时，一级缓存也会被清空

4.将Mybatis和Spring整合后，事务控制在service中，每个service方法在开始执行时会开启事务并创建SqlSession对象，在方法结束时关闭SqlSession

在方法中多次调用的mapper方法共用一个一级缓存

### 3.二级缓存

1. 二级缓存是Mapper级别的缓存，每一个namespace有一个缓冲区域，一个Mapper的sql语句可以由多个SqlSession去操作，所以多个SqlSession共用二级缓存，

即二级缓存是跨SqlSession的

2.开启二级缓存时，不但需要在SqlMapConfig.xml中开启，还需要在mapper.xml中开启，mapper存储缓冲数据的数据结构也是HashMap

<!--开启二级缓存-->

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

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

<!--开启本namespace的二级缓存-->

<cache/>

3.由于二级缓存数据的存储介质多种多样，不一定只在内存中，为了能够将缓存数据取出执行反序列化操作，需要pojo实现序列化接口

4.在关闭SqlSession时，才能把SqlSession中的数据存储到二级缓存中，与一级缓存一样，一旦执行commit操作，二级缓存也会被清空，并且两者查询原理相同

5.二级缓存的参数

<!--禁止当前select语句的二级缓存，useCache的默认值为true，即使用二级缓存-->

<select id="findUserById" resultType="user" useCache="false">

<!--不刷新缓存，insert、delete、update的flushCache默认值为true，即在执行commit()后清空缓存-->

<update id="updateUserSex" parameterType="int" flushCache="false">

6. Mybatis整合Encache

在不使用分布式缓存时，缓存数据在各个服务器单独存取，不方便系统开发，所以需要使用分布式缓存对缓存数据进行集中管理

Mybatis无法实现分布式管理，需要和其它的分布式缓存框架进行整合

Mybatis提供了一个cache接口，其默认实现类是PerpetualCache，在Mybatis和Encache整合后，只需要更改Mybatis的默认实现类即可

<!--需要添加encache.xml配置文件并引入jar包：encache-core、mybatis-encache-->

<cache type="org.mybatis.caches.ehcache.EhcacheCache"/>

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

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

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

<defaultCache

maxElementsInMemory="1000"

maxElementsOnDisk="10000000"

eternal="false"

overflowToDisk="false"

timeToIdleSeconds="120"

timeToLiveSeconds="120"

diskExpiryThreadIntervalSeconds="120"

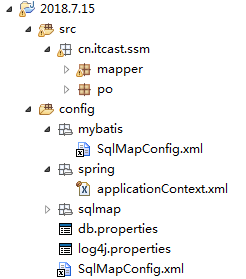
memoryStoreEvictionPolicy="LRU">

</defaultCache>

</ehcache>

## Mybatis和Spring整合

需要先在先前的基础上还需要导入spring与mybatis整合的jar包和spring的jar包，并引入spring的核心配置文件



applicationContext.xml

<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 ">

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

<bean id="dataSource" class="org.apache.commons.dbcp.BasicDataSource" destroy-method="close">

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

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

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

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

</bean>

<!--配置SqlSessionFactory，该类在mybatis与spring的整合包下-->

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

<!-- 加载mybatis的配置文件 -->

<property name="configLocation" value="mybatis/SqlMapConfig.xml" />

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

</bean>

</beans>

### 1.原始开发Dao方式

<mapper namespace="test">

<select id="findUserById" parameterType="int" resultType="cn.itcast.ssm.po.User">

SELECT \* FROM USER WHERE id=#{id}

</select>

</mapper>

public interface UserDao {

public User findUserById(int id) throws Exception;

}

//这里继承SqlSessionDaoSupport，它类似于Hibernate框架的HibernateDaoSupport

public class UserDaoImpl extends SqlSessionDaoSupport implements UserDao {

@Override

public User findUserById(int id) throws Exception {

return getSqlSession().selectOne("test.findUserById", id);

}

}

<bean id="userDao" class="cn.itcast.ssm.dao.UserDaoImpl">

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

</bean>

public class UserDaoImplTest {

@Test

public void testFindUserById() throws BeansException, Exception {

System.out.println(((UserDao)new ClassPathXmlApplicationContext("spring/applicationContext.xml").getBean("userDao")).findUserById(1));

}

}

### 2.Mapper代理开发

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

<select id="findUserById" parameterType="int" resultType="user">

SELECT \* FROM USER WHERE id=#{value}

</select>

</mapper>

public interface UserMapper {

public User findUserById(int id) throws Exception;

}

<!--该类可根据Mapper接口生成代理对象-->

<bean id="userMapper" class="org.mybatis.spring.mapper.MapperFactoryBean">

<!--需要指定Mapper接口-->

<property name="mapperInterface" value="cn.itcast.ssm.mapper.UserMapper"/>

<!--该代理对象类继承SqlSessionDaoSupport，所以也需要注入sqlSessionFactory-->

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

</bean>

public class UserMapperTest {

@Test

public void testFindUserById() throws BeansException, Exception {

System.out.println(((UserMapper)new ClassPathXmlApplicationContext("spring/applicationContext.xml").getBean("userMapper")).findUserById(1));

}

}

升级！

<!--mapper批量扫描：该类是一个Mapper扫描器，它能从指定包中扫描出所有mapper接口，并自动创建代理对象，扫描出来的mapper的bean的id为接口名称首字母小写-->

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

<!--指定需要被扫描的包，不能使用通配符，只能用","将多个包隔开-->

<property name="basePackage" value="cn.itcast.ssm.mapper"/>

<!--由于扫描器会比加载配置文件的组件先执行，导致找不到sqlSessionFactory，所以这里使用sqlSessionFactoryBeanName和value属性-->

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

</bean>

</beans>

<!--这时，这一步也可以省略了，但仍然要求映射文件和Mapper接口在同一个包下，且名称相同-->

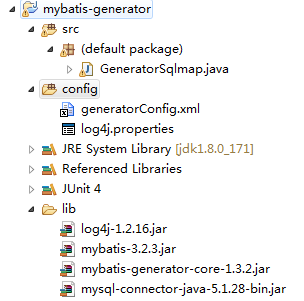
<mappers>

<package name="cn.itcast.ssm"/>

</mappers>

## Mybatis逆向工程

针对数据库单表自动生成mybatis执行所需要的代码，这里使用java程序的xml文件的方式单独创建项目来生成代码，生成的代码尽量不要修改！需要时拷贝！



<?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 -->

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

connectionURL="jdbc:mysql:///mybatis" userId="root"

password="123">

</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PO类的位置 -->

<javaModelGenerator targetPackage="cn.itcast.ssm.po"

targetProject=".\src">

<!-- 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="cn.itcast.ssm.mapper"

targetProject=".\src">

<!-- 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="cn.itcast.ssm.mapper"

targetProject=".\src">

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

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

</javaClientGenerator>

<!-- 指定数据库表 -->

<table tableName="items"></table>

<table tableName="orders"></table>

<table tableName="orderdetail"></table>

<table tableName="user"></table>

</context>

</generatorConfiguration>

public class GeneratorSqlmap {

@Test

public void generator() throws Exception{

try{

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

boolean overwrite = true;

File configFile = new File("config/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);

}catch(Exception e){

e.printStackTrace();

}

}

}

