## Mybatis入门

### 从一个jdbc程序开始

**public** **static** **void** main(String[] args) {

Connection connection = **null**;

PreparedStatement preparedStatement = **null**;

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", "mysql");

//定义sql语句 ?表示占位符

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

//获取预处理statement

preparedStatement = connection.prepareStatement(sql);

//设置参数，第一个参数为sql语句中参数的序号（从1开始），第二个参数为设置的参数值

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

**if**(resultSet!=**null**){

**try** {

resultSet.close();

} **catch** (SQLException e) {

// **TODO** Auto-generated catch block

e.printStackTrace();

}

}

**if**(preparedStatement!=**null**){

**try** {

preparedStatement.close();

} **catch** (SQLException e) {

// **TODO** Auto-generated catch block

e.printStackTrace();

}

}

**if**(connection!=**null**){

**try** {

connection.close();

} **catch** (SQLException e) {

// **TODO** Auto-generated catch block

e.printStackTrace();

}

}

}

}

上边使用jdbc的原始方法（未经封装）实现了查询数据库表记录的操作。

#### jdbc操作步骤总结如下：

1. 加载数据库驱动
2. 创建并获取数据库链接
3. 创建jdbc statement对象
4. 设置sql语句
5. 设置sql语句中的参数(使用preparedStatement)
6. 通过statement执行sql并获取结果
7. 对sql执行结果进行解析处理
8. 释放资源(resultSet、preparedstatement、connection)

#### jdbc问题总结如下：

1. 数据库链接创建、释放频繁造成系统资源浪费从而影响系统性能，如果使用数据库链接池可解决此问题。
2. Sql语句写在代码中造成代码不易维护，实际应用sql变化的可能较大，sql变动需要改变java代码。
3. 向sql语句传参数麻烦，因为sql语句的where条件不一定，可能多也可能少，占位符需要和参数一一对应。
4. 对结果集解析麻烦，sql变化导致解析代码变化，且解析前需要遍历，如果能将数据库记录封装成pojo对象解析比较方便。

### MyBatis介绍

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。

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。

Mybatis通过xml或注解的方式将要执行的statement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。

### Mybatis架构



1. mybatis配置

SqlMapConfig.xml，此文件作为mybatis的全局配置文件，配置了mybatis的运行环境等信息。

mapper.xml文件即sql映射文件，文件中配置了操作数据库的sql语句。此文件需要在SqlMapConfig.xml中加载。

1. 通过mybatis环境等配置信息构造SqlSessionFactory即会话工厂
2. 由会话工厂创建sqlSession即会话，操作数据库需要通过sqlSession进行。
3. mybatis底层自定义了Executor接口操作数据库，Executor接口有两个实现，一个是基本实现、一个是缓存实现。
4. Mapped Statement也是mybatis一个底层对象，它包装了mybatis配置信息及sql映射信息等。mapper.xml文件中一个sql对应一个Mapped Statement对象，sql的id即是Mapped statement的id。
5. Mapped Statement对sql执行输入参数进行定义，包括HashMap、基本类型、pojo，Executor通过 Mapped Statement在执行sql前将输入的java对象映射至sql中，输入参数映射就是jdbc编程中对preparedStatement设置参数。
6. Mapped Statement对sql执行输出结果进行定义，包括HashMap、基本类型、pojo，Executor通过 Mapped Statement在执行sql后将输出结果映射至java对象中，输出结果映射过程相当于jdbc编程中对结果的解析处理过程。

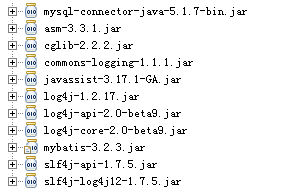
### Mybatis第一个程序

#### 第一步：创建java工程

使用eclipse创建java工程，jdk使用1.6。

#### 第二步：加入jar包

加入mybatis核心包、依赖包、数据驱动包。



#### 第三步：log4j.properties

在classpath下创建log4j.properties如下：

# 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

mybatis默认使用log4j作为输出日志信息。

#### 第四步：SqlMapConfig.xml

在classpath下创建SqlMapConfig.xml，如下：

<?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>

<!-- <properties resource=""></properties> -->

<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/mybatis?characterEncoding=utf-8"* />

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

<property name=*"password"* value=*"mysql"* />

</dataSource>

</environment>

</environments>

</configuration>

SqlMapConfig.xml是mybatis核心配置文件，上边文件的配置内容为数据源、事务管理。

#### 第五步：po类

Po类作为mybatis进行sql映射使用，po类通常与数据库表对应，User.java如下：

**public** **class** User {

**private** **int** id;

**private** String username;// 用户姓名

**private** String sex;// 性别

**private** Date birthday;// 出生日期

**private** String address;// 地址

**private** String detail;// 详细信息

**private** Float score;// 成绩

get/set……

#### 第六步：sql映射文件

在classpath下的sqlmap目录下创建sql映射文件User.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=*"test"*>

<!-- 根据id获取用户信息 -->

<select id=*"selectUserById"* parameterType=*"int"* resultType=*"cn.itcast.mybatis.po.User"*>

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

</select>

<!-- 获取用户列表 -->

<select id=*"selectUserList"* resultType=*"cn.itcast.mybatis.po.User"*>

select \* from user

</select>

<!-- 添加用戶 -->

<insert id=*"insertUser"* parameterType=*"cn.itcast.mybatis.po.User"*>

insert into user(username,birthday,sex,address,detail,score)

values(#{username},#{birthday},#{sex},#{address},#{detail},#{score});

</insert>

<!-- 更新用戶 -->

<update id=*"updateUser"* parameterType=*"cn.itcast.mybatis.po.User"*>

update user set username=#{username},birthday=#{birthday},sex=#{sex},address=#{address},detail=#{detail},score=#{score}

where id=#{id}

</update>

<!-- 刪除用戶 -->

<delete id=*"deleteUser"* parameterType=*"cn.itcast.mybatis.po.User"*>

delete from user where id=#{id}

</delete>

</mapper>

namespace ：命名空间，用于隔离sql语句，后面会讲另一层非常重要的作用。

parameterType ：定义输入到sql中的映射类型，#{id}表示使用preparedstatement设置占位符号并将输入变量id传到sql。

resultType：定义结果映射类型。

#### 第七步：将User.xml添加在SqlMapConfig.xml

在SqlMapConfig.xml中添加mappers如下：

<mappers>

<mapper resource=*"sqlmap/user.xml"* />

</mappers>

这里即告诉mybatis Sql映射文件在哪里。

#### 第八步：程序编写

##### 查询

/\*\*

\* 第一个mybatis程序

\*

\* **@author** Thinkpad

\*

\*/

**public** **class** Mybatis\_select {

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

//mybatis配置文件

String resource = "sqlMapConfig.xml";

InputStream inputStream = Resources.*getResourceAsStream*(resource);

//使用SqlSessionFactoryBuilder创建sessionFactory

SqlSessionFactory sqlSessionFactory = **new** SqlSessionFactoryBuilder()

.build(inputStream);

//通过session工厂获取一个Sqlsession，sqlsession中包括了对数据库操作的sql方法

SqlSession session = sqlSessionFactory.openSession();

**try** {

//通过sqlsession调用selectOne方法获取一条结果集

//参数1：指定定义的statement的id,参数2：指定向statement中传递的参数

User user = session.selectOne("test.selectUserById", 1);

System.*out*.println(user);

} **finally**{

session.close();

}

}

}

##### 添加

**public** **class** Mybatis\_insert {

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

//mybatis配置文件

String resource = "sqlMapConfig.xml";

InputStream inputStream = Resources.*getResourceAsStream*(resource);

//使用SqlSessionFactoryBuilder创建sessionFactory

SqlSessionFactory sqlSessionFactory = **new** SqlSessionFactoryBuilder()

.build(inputStream);

//通过session工厂获取一个Sqlsession，sqlsession中包括了对数据库操作的sql方法

SqlSession session = sqlSessionFactory.openSession();

**try** {

User user = **new** User();

user.setUsername("张三");

user.setBirthday(**new** Date());

user.setSex("1");

user.setAddress("北京市");

user.setDetail("好同志");

user.setScore(99.8f);

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

session.commit();

} **finally**{

session.close();

}

}

}

###### 主键返回

通过修改sql映射文件，可以将mysql自增主键返回：

<insert id=*"insertUser"* parameterType=*"cn.itcast.mybatis.po.User"*>

<!-- selectKey将主键返回，需要再返回 -->

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

select LAST\_INSERT\_ID()

</selectKey>

insert into user(username,birthday,sex,address,detail,score)

values(#{username},#{birthday},#{sex},#{address},#{detail},#{score});

</insert>

添加selectKey实现将主键返回

keyProperty:返回的主键存储在pojo中的哪个属性

order：selectKey的执行顺序，是相对与insert语句来说，由于mysql的自增原理执行完insert语句之后才将主键生成，所以这里selectKey的执行顺序为after

resultType:返回的主键是什么类型

LAST\_INSERT\_ID():是mysql的函数

##### 删除

**public** **class** Mybatis\_delete {

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

//mybatis配置文件

String resource = "sqlMapConfig.xml";

InputStream inputStream = Resources.*getResourceAsStream*(resource);

//使用SqlSessionFactoryBuilder创建sessionFactory

SqlSessionFactory sqlSessionFactory = **new** SqlSessionFactoryBuilder()

.build(inputStream);

//通过session工厂获取一个Sqlsession，sqlsession中包括了对数据库操作的sql方法

SqlSession session = sqlSessionFactory.openSession();

**try** {

session.delete("test.deleteUser", 4);

session.commit();

} **finally**{

session.close();

}

}

}

##### 修改

**public** **class** Mybatis\_update {

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

//mybatis配置文件

String resource = "sqlMapConfig.xml";

InputStream inputStream = Resources.*getResourceAsStream*(resource);

//使用SqlSessionFactoryBuilder创建sessionFactory

SqlSessionFactory sqlSessionFactory = **new** SqlSessionFactoryBuilder()

.build(inputStream);

//通过session工厂获取一个Sqlsession，sqlsession中包括了对数据库操作的sql方法

SqlSession session = sqlSessionFactory.openSession();

**try** {

User user = **new** User();

user.setId(4);

user.setUsername("李四");

user.setBirthday(**new** Date());

user.setSex("1");

user.setAddress("北京市");

user.setDetail("好同志");

user.setScore(99.8f);

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

session.commit();

} **finally**{

session.close();

}

}

}

##### 步骤总结：

1. 创建SqlSessionFactory
2. 通过SqlSessionFactory创建SqlSession
3. 通过sqlsession执行数据库操作
4. 调用session.commit()提交事务
5. 调用session.close()关闭会话

### Mybatis解决jdbc编程的问题

1. 数据库链接创建、释放频繁造成系统资源浪费从而影响系统性能，如果使用数据库链接池可解决此问题。

解决：在SqlMapConfig.xml中配置数据链接池，使用连接池管理数据库链接。

1. Sql语句写在代码中造成代码不易维护，实际应用sql变化的可能较大，sql变动需要改变java代码。

解决：将Sql语句配置在XXXXmapper.xml文件中与java代码分离。

1. 向sql语句传参数麻烦，因为sql语句的where条件不一定，可能多也可能少，占位符需要和参数一一对应。

解决： Mybatis自动将java对象映射至sql语句。

1. 对结果集解析麻烦，sql变化导致解析代码变化，且解析前需要遍历，如果能将数据库记录封装成pojo对象解析比较方便。

解决：Mybatis自动将sql执行结果映射至java对象。

### 与hibernate主要不同

Mybatis和hibernate不同，它不完全是一个ORM框架，因为MyBatis需要程序员自己编写Sql语句，不过mybatis可以通过XML或注解方式灵活配置要运行的sql语句，并将java对象和sql语句映射生成最终执行的sql，最后将sql执行的结果再映射生成java对象。

Mybatis学习门槛低，简单易学，程序员直接编写原生态sql，可严格控制sql执行性能，灵活度高，非常适合对关系数据模型要求不高的软件开发，例如互联网软件、企业运营类软件等，因为这类软件需求变化频繁，一但需求变化要求成果输出迅速。但是灵活的前提是mybatis无法做到数据库无关性，如果需要实现支持多种数据库的软件则需要自定义多套sql映射文件，工作量大。

Hibernate对象/关系映射能力强，数据库无关性好，对于关系模型要求高的软件（例如需求固定的定制化软件）如果用hibernate开发可以节省很多代码，提高效率。但是Hibernate的缺点是学习门槛高，要精通门槛更高，而且怎么设计O/R映射，在性能和对象模型之间如何权衡，以及怎样用好Hibernate需要具有很强的经验和能力才行。

总之，按照用户的需求在有限的资源环境下只要能做出维护性、扩展性良好的软件架构都是好架构，所以框架只有适合才是最好。

## SqlSession

SqlSession中封装了对数据库的sql操作，如：查询、插入、更新、删除等。

通过SqlSessionFactory创建SqlSession，而SqlSessionFactory是通过SqlSessionFactoryBuilder进行创建。

### SqlSessionFactoryBuilder

SqlSessionFacoty是通过SqlSessionFactoryBuilder进行创建，SqlSessionFactoryBuilder只用于创建SqlSessionFactory，可以当成一个工具类，在使用时随时拿来使用不需要特殊处理为共享对象。

### SqlSessionFactory

SqlSessionFactory是一个接口，接口中定义了openSession的不同方式，SqlSessionFactory一但创建后可以重复使用，实际应用时通常设计为单例模式。

### SqlSession

SqlSession是一个接口，默认使用DefaultSqlSession实现类，sqlSession中定义了数据库操作。

执行过程如下：

1. 加载数据源等配置信息

Environment environment = configuration.getEnvironment();

1. 创建数据库链接
2. 创建事务对象
3. 创建Executor，SqlSession所有操作都是通过Executor完成，mybatis源码如下：

**if** (ExecutorType.*BATCH* == executorType) {

executor = **new** BatchExecutor(**this**, transaction);

} **else** **if** (ExecutorType.*REUSE* == executorType) {

executor = **new** ReuseExecutor(**this**, transaction);

} **else** {

executor = **new** SimpleExecutor(**this**, transaction);

}

**if** (cacheEnabled) {

executor = **new** CachingExecutor(executor, autoCommit);

}

1. SqlSession的实现类即DefaultSqlSession，此对象中对操作数据库实质上用的是Executor

结论：每个线程都应该有它自己的SqlSession实例。SqlSession的实例不能共享使用，它也是线程不安全的。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甚至是实例字段中。

## Namespace的作用

命名空间除了对sql进行隔离，mybatis中对命名空间有特殊的作用，用于定义mapper接口地址。

### 问题：

没有使用接口编程，java是面向接口编程语言，对数据库的操作应该定义一些操作接口，如：用户添加、用户删除、用户查询等，调用dao接口完成数据库操作。

#### 解决：

**public** **interface** UserDao {

**public** User getUserById(**int** id) **throws** Exception;

**public** **void** insertUser(User user) **throws** Exception;

}

**public** **class** UserDaoImpl **implements** UserDao {

**public** UserDaoImpl(SqlSessionFactory sqlSessionFactory){

**this**.setSqlSessionFactory(sqlSessionFactory);

}

**private** SqlSessionFactory sqlSessionFactory;

@Override

**public** User getUserById(**int** id) **throws** Exception {

SqlSession session = sqlSessionFactory.openSession();

User user = **null**;

**try** {

//通过sqlsession调用selectOne方法获取一条结果集

//参数1：指定定义的statement的id,参数2：指定向statement中传递的参数

user = session.selectOne("selectUserById", 1);

System.*out*.println(user);

//获取List

List<User> list = session.selectList("selectUserList");

System.*out*.println(list);

} **finally**{

session.close();

}

**return** user;

}

@Override

**public** **void** insertUser(User user) **throws** Exception {

SqlSession session = sqlSessionFactory.openSession();

**try** {

session.insert("insertUser", user);

session.commit();

} **finally**{

session.close();

}

}

**public** SqlSessionFactory getSqlSessionFactory() {

**return** sqlSessionFactory;

}

**public** **void** setSqlSessionFactory(SqlSessionFactory sqlSessionFactory) {

**this**.sqlSessionFactory = sqlSessionFactory;

}

}

### 问题：

第一个例子中，在访问sql映射文件中定义的sql时需要调用sqlSession的selectOne方法，并将sql的位置(命名空间+id)和参数传递到selectOne方法中，且第一个参数是一个长长的字符串，第二个参数是一个object对象，这对于程序编写有很大的不方便，很多问题无法在编译阶段发现。

虽然上边对提出的面向接口编程问题进行解决，但是dao实现方法中仍然是调用sqlSession的selectOne方法，重复代码多。

#### 改为mapper 接口实现：

##### 第一步：定义mapper.xml

Mapper.xml文件不变还用原来的。

##### 第二步：定义mapper 接口

/\*\*

\* 用户管理mapper

\* **@author** Thinkpad

\*

\*/

**public** **interface** UserMapper {

**public** User selectUserById(**int** id) **throws** Exception;

**public** List<User> selectUserList() **throws** Exception;

**public** **void** insertUser(User user) **throws** Exception;

**public** **void** updateUser(User user) **throws** Exception;

**public** **void** deleteUser(**int** id) **throws** Exception;

}

**接口定义有如下特点：**

1. Mapper接口方法名和mapper.xml中定义的每个sql的id相同
2. Mapper接口方法的输入参数类型和mapper.xml中定义的每个sql 的parameterType的类型相同
3. Mapper接口方法的输出参数类型和mapper.xml中定义的每个sql的resultType的类型相同

##### 第三步：修改namespace

Mapper.xml映射文件中的namepace改为如下：

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

**修改后namespace即是mapper接口的地址。**

##### 第四步：通过mapper接口调用statement

**public** **class** UserMapperTest **extends** TestCase {

**private** SqlSessionFactory sqlSessionFactory;

**protected** **void** setUp() **throws** Exception {

//mybatis配置文件

String resource = "sqlMapConfig.xml";

InputStream inputStream = Resources.*getResourceAsStream*(resource);

//使用SqlSessionFactoryBuilder创建sessionFactory

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

}

**public** **void** testSelectUserById() **throws** Exception {

//获取session

SqlSession session = sqlSessionFactory.openSession();

//获取mapper接口实例

UserMapper userMapper = session.getMapper(UserMapper.**class**);

//通过mapper接口调用statement

User user = userMapper.selectUserById(1);

System.*out*.println(user);

//关闭session

session.close();

}

**public** **void** testSelectUserList() **throws** Exception {

//获取session

SqlSession session = sqlSessionFactory.openSession();

//获取mapper接口实例

UserMapper userMapper = session.getMapper(UserMapper.**class**);

//通过mapper接口调用statement

List<User> list = userMapper.selectUserList();

System.*out*.println(list);

//关闭session

session.close();

}

**public** **void** testInsertUser() **throws** Exception {

//获取session

SqlSession session = sqlSessionFactory.openSession();

//获限mapper接口实例

UserMapper userMapper = session.getMapper(UserMapper.**class**);

//要添加的数据

User user = **new** User();

user.setUsername("张三");

user.setBirthday(**new** Date());

user.setSex("1");

user.setAddress("北京市");

user.setDetail("好同志");

user.setScore(99.8f);

//通过mapper接口添加用户

userMapper.insertUser(user);

//提交

session.commit();

//关闭session

session.close();

}

**public** **void** testUpdateUser() **throws** Exception {

//获取session

SqlSession session = sqlSessionFactory.openSession();

//获限mapper接口实例

UserMapper userMapper = session.getMapper(UserMapper.**class**);

//要更新的数据

User user = **new** User();

user.setId(7);

user.setUsername("李四");

user.setBirthday(**new** Date());

user.setSex("1");

user.setAddress("北京市");

user.setDetail("好同志");

user.setScore(99.8f);

//通过mapper接口调用statement

userMapper.updateUser(user);

//提交

session.commit();

//关闭session

session.close();

}

**public** **void** testDeleteUser() **throws** Exception {

//获取session

SqlSession session = sqlSessionFactory.openSession();

//获限mapper接口实例

UserMapper userMapper = session.getMapper(UserMapper.**class**);

//通过mapper接口删除用户

userMapper.deleteUser(6);

//提交

session.commit();

//关闭session

session.close();

}

}

session.getMapper(UserMapper.**class**)生成一个代理对象作为UserMapper的接口实现对象。

### 总结：

使用mapper接口不用写接口实现类即可完成数据库操作，简单方便，此方法为官方推荐方法。

使用mapper接口调用必须具备如下条件：

1. Mapper接口方法名和mapper.xml中定义的每个sql的id相同
2. Mapper接口方法的输入参数类型和mapper.xml中定义的每个sql 的parameterType的类型相同
3. Mapper接口方法的输出参数类型和mapper.xml中定义的每个sql的resultType的类型相同
4. Mapper.xml文件中的namespace即是mapper接口的类路径。

至此，mybatis的mapper包括mapper.xml和mapper接口两种文件。

## SqlMapConfig.xml

### 配置内容

SqlMapConfig.xml中配置的内容和顺序如下：

properties（属性）

settings（配置）

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）

### properties（属性）

SqlMapConfig.xml可以引用java属性文件中的配置信息如下：

在classpath下定义db.properties文件，

driver=com.mysql.jdbc.Driver

url=jdbc:mysql://localhost:3306/mybatis

username=root

password=mysql

SqlMapConfig.xml引用如下：

<properties resource=*"db.properties"* />

<environments default=*"development"*>

<environment id=*"development"*>

<transactionManager type=*"JDBC"* />

<dataSource type=*"POOLED"*>

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

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

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

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

</dataSource>

</environment>

</environments>

### settings（配置）

mybatis全局配置参数，全局参数将会影响mybatis的运行行为。

详细参见“学习资料/mybatis-settings.xlsx”文件



### typeAliases（类型别名）

#### mybatis支持别名：

|  |  |
| --- | --- |
| 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：

##### 在SqlMapConfig.xml中配置：

<typeAliases>

<!-- 单个别名定义 -->

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

<!-- 批量别名定义，扫描整个包下的类 -->

<package name=*"cn.itcast.mybatis.po"*/>

</typeAliases>

### typeHandlers（类型处理器）

类型处理器在将java类型和sql映射文件进行映射时使用，如下：

<select id=*"selectUserById"* parameterType=*"int"* resultType=*"user"*>

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

</select>

parameterType：指定输入数据类型为int，即向statement设置值

resultType：指定输出数据类型为自定义User，即将resultset转为java对象

mybatis自带的类型处理器基本上满足日常需求，不需要单独定义。

mybatis支持类型处理器：

|  |  |  |
| --- | --- | --- |
| 类型处理器 | **Java**类型 | **JDBC**类型 |
| BooleanTypeHandler | Boolean，boolean | 任何兼容的布尔值 |
| ByteTypeHandler | Byte，byte | 任何兼容的数字或字节类型 |
| ShortTypeHandler | Short，short | 任何兼容的数字或短整型 |
| IntegerTypeHandler | Integer，int | 任何兼容的数字和整型 |
| LongTypeHandler | Long，long | 任何兼容的数字或长整型 |
| FloatTypeHandler | Float，float | 任何兼容的数字或单精度浮点型 |
| DoubleTypeHandler | Double，double | 任何兼容的数字或双精度浮点型 |
| BigDecimalTypeHandler | BigDecimal | 任何兼容的数字或十进制小数类型 |
| StringTypeHandler | String | CHAR和VARCHAR类型 |
| ClobTypeHandler | String | CLOB和LONGVARCHAR类型 |
| NStringTypeHandler | String | NVARCHAR和NCHAR类型 |
| NClobTypeHandler | String | NCLOB类型 |
| ByteArrayTypeHandler | byte[] | 任何兼容的字节流类型 |
| BlobTypeHandler | byte[] | BLOB和LONGVARBINARY类型 |
| DateTypeHandler | Date（java.util） | TIMESTAMP类型 |
| DateOnlyTypeHandler | Date（java.util） | DATE类型 |
| TimeOnlyTypeHandler | Date（java.util） | TIME类型 |
| SqlTimestampTypeHandler | Timestamp（java.sql） | TIMESTAMP类型 |
| SqlDateTypeHandler | Date（java.sql） | DATE类型 |
| SqlTimeTypeHandler | Time（java.sql） | TIME类型 |
| ObjectTypeHandler | 任意 | 其他或未指定类型 |
| EnumTypeHandler | Enumeration类型 | VARCHAR-任何兼容的字符串类型，作为代码存储（而不是索引）。 |

### mappers（映射器）

Mapper配置的几种方法：

#### <mapper resource=" " />

使用相对于类路径的资源

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

#### <mapper url=" " />

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接口类路径

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

**注意：此种方法要求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=""/>

注册指定包下的所有mapper接口

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

**注意：此种方法要求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.xml

Mapper.xml映射文件中定义了操作数据库的sql，每个sql是一个statement，映射文件是mybatis的核心。

### parameterType (输入类型)

#### #{}与${}

#{}实现的是向prepareStatement中的预处理语句中设置参数值，sql语句中#{}表示一个占位符即?。

<!-- 根据id查询用户信息 -->

<select id=*"selectUserById"* parameterType=*"int"* resultType=*"user"*>

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

</select>

使用占位符#{}可以有效防止sql注入，在使用时不需要关心参数值的类型，mybatis会根据参数值的类型调用不同的statement设置参数值的方法。可以想象为：如果参数值是一个字符串则自动映射生成的sql中参数值两边自动有单引号，如果参数值是一个数字型则自动映射生成的sql中参数值两边没有单引号。

**注意：当传递单个值时#{}中的参数名称通常和mapper接口的形参名称相同，也可以设置成任意值。**

${}和#{}不同，${}是将参数值不加修饰的拼在sql中，相当中用jdbc的statement拼接sql，使用${}不能防止sql注入，但是有时用${}会非常方便，如下的例子：

<!-- 根据名称模糊查询用户信息 -->

<select id=*"selectUserByName"* parameterType=*"string"* resultType=*"user"*>

select \* from user where username like '%${value}%'

</select>

如果本例子使用#{}则传入的字符串中必须有%号，而%是人为拼接在参数中，显然有点麻烦，如果采用${}在sql中拼接为%的方式则在调用mapper接口传递参数就方便很多。

//如果使用占位符号则必须人为在传参数中加%

List<User> list = userMapper.selectUserByName("%管理员%");

//如果使用${}原始符号则不用人为在参数中加%

List<User> list = userMapper.selectUserByName("管理员");

再比如order by排序，如果将列名通过参数传入sql，根据传的列名进行排序，应该写为：

ORDER BY ${columnName}

如果使用#{}将无法实现此功能。

**注意：**${}不能防止sql注入，对系统安全性有很大的影响，如果使用${}建议传入参数尽量不让用户自动填写，即使要用户填写也要对填写的数据进行校验，保证安全性。

另外，当传递单个值时${}中填写的参数名称经过测试填写value不错报。

#### 传递简单类型

参考上边的例子。

#### 传递pojo对象

Mybatis使用ognl表达式解析对象字段的值，如下例子：

<!—传递pojo对象综合查询用户信息 -->

<select id=*"selectUserByUser"* parameterType=*"user"* resultType=*"user"*>

select \* from user where id=#{id} and username like '%${username}%'

</select>

上边红色标注的是user对象中的字段名称。

测试：

**public** **void** testselectUserByUser()**throws** Exception{

//获取session

SqlSession session = sqlSessionFactory.openSession();

//获限mapper接口实例

UserMapper userMapper = session.getMapper(UserMapper.**class**);

//构造查询条件user对象

User user = **new** User();

user.setId(1);

user.setUsername("管理员");

//传递user对象查询用户列表

List<User> list = userMapper.selectUserByUser(user);

//关闭session

session.close();

}

异常测试：

Sql中字段名输入错误后测试，username输入dusername测试结果报错：

org.apache.ibatis.exceptions.PersistenceException:

### Error querying database. Cause: org.apache.ibatis.reflection.ReflectionException: There is no getter for property named 'dusername' in 'class cn.itcast.mybatis.po.User'

### Cause: org.apache.ibatis.reflection.ReflectionException: There is no getter for property named 'dusername' in 'class cn.itcast.mybatis.po.User'

#### 传递hashmap

Sql映射文件定义如下：

<!-- 传递hashmap综合查询用户信息 -->

<select id=*"selectUserByHashmap"* parameterType=*"hashmap"* resultType=*"user"*>

select \* from user where id=#{id} and username like '%${username}%'

</select>

上边红色标注的是hashmap的key。

测试：

**public** **void** testselectUserByHashmap()**throws** Exception{

//获取session

SqlSession session = sqlSessionFactory.openSession();

//获限mapper接口实例

UserMapper userMapper = session.getMapper(UserMapper.**class**);

//构造查询条件Hashmap对象

HashMap<String, Object> map = **new** HashMap<String, Object>();

map.put("id", 1);

map.put("username", "管理员");

//传递Hashmap对象查询用户列表

List<User> list = userMapper.selectUserByHashmap(map);

//关闭session

session.close();

}

异常测试：

传递的map中的key和sql中解析的key不一致。

测试结果没有报错，只是通过key获取值为空。

### resultType(输出类型)

#### 输出简单类型

参考getnow输出日期类型，看下边的例子输出整型：

Mapper.xml文件

<!-- 获取用户列表总数 -->

<select id=*"selectUserCount"* parameterType=*"user"* resultType=*"int"*>

select count(1) from user

</select>

Mapper接口

public int selectUserCount(User user) throws Exception;

调用：

**public** **void** testselectUserCount() **throws** Exception{

//获取session

SqlSession session = sqlSessionFactory.openSession();

//获取mapper接口实例

UserMapper userMapper = session.getMapper(UserMapper.**class**);

User user = **new** User();

user.setUsername("管理员");

//传递Hashmap对象查询用户列表

**int** count = userMapper.selectUserCount(user);

//使用session实现

//int count = session.selectOne("cn.itcast.mybatis.mapper.UserMapper.selectUserCount", user);

//关闭session

session.close();

}

**总结：**

输出简单类型必须查询出来的结果集有一条记录，最终将第一个字段的值转换为输出类型。

使用session的selectOne可查询单条记录。

#### 输出pojo

##### 输出pojo对象

参考selectUserById的定义：

Mapper.xml

<!-- 根据id查询用户信息 -->

<select id=*"selectUserById"* parameterType=*"int"* resultType=*"user"*>

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

</select>

Mapper接口：

public User selectUserById(int id) throws Exception;

测试：

**public** **void** testSelectUserById() **throws** Exception {

//获取session

SqlSession session = sqlSessionFactory.openSession();

//获限mapper接口实例

UserMapper userMapper = session.getMapper(UserMapper.**class**);

//通过mapper接口调用statement

User user = userMapper.selectUserById(1);

System.*out*.println(user);

//关闭session

session.close();

}

使用session调用selectOne查询单条记录。

##### 输出pojo列表

参考selectUserByName的定义：

Mapper.xml

<!-- 根据名称模糊查询用户信息 -->

<select id=*"selectUserByName"* parameterType=*"string"* resultType=*"user"*>

select \* from user where username like '%${value}%'

</select>

Mapper接口：

public List<User> selectUserByName(String username) throws Exception;

测试：

**public** **void** testselectUserByName()**throws** Exception{

//获取session

SqlSession session = sqlSessionFactory.openSession();

//获限mapper接口实例

UserMapper userMapper = session.getMapper(UserMapper.**class**);

//如果使用占位符号则必须人为在传参数中加%

//List<User> list = userMapper.selectUserByName("%管理员%");

//如果使用${}原始符号则不用人为在参数中加%

List<User> list = userMapper.selectUserByName("管理员");

//关闭session

session.close();

}

使用session的selectList方法获取pojo列表。

##### 总结：

输出pojo对象和输出pojo列表在sql中定义的resultType是一样的。

返回单个pojo对象要保证sql查询出来的结果集为单条，使用session.selectOne方法调用，mapper接口使用pojo对象作为方法返回值。

返回pojo列表表示查询出来的结果集可能为多条，只能使用session.selectList方法调用，mapper接口使用List<pojo>对象作为方法返回值。

#### 输出hashmap

输出pojo对象可以改用hashmap输出类型，将输出的字段名称作为map的key，value为字段值。

总结：如果方法输出单个对象，必须保证sql查询出来的记录为单条，如果为多条则报错，因为内部是使用的session.selectOne方法查询。

### resultMap

当输出pojo的字段和sql查询出来的字段名称不对应时而还想用这个pojo类作为输出类型这时就需要使用resultMap了。

另外，resultMap也解决了一对一关联查询、一对多关联查询等常见需求。

#### 创建Person类

**public** **class** Person {

**private** **int** id;

**private** String name;// 用户姓名，名称和User表的字段名称不一样

**private** String sex;// 性别

**private** Date birthday;// 出生日期

**private** String addr;// 地址，名称和User表的字段名称不一样

**private** String detail;// 详细信息

**private** Float score;// 成绩

get/set。。。。

#### 定义resultMap

在mapper.xml文件中定义resultMap：

<!-- resultMap定义 -->

<resultMap type=*"cn.itcast.mybatis.po.Person"* id=*"personmap"*>

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

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

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

</resultMap>

<id />：此属性表示查询结果集的唯一标识，非常重要。如果是多个字段为复合唯一约束则定义多个<id />。

Property：表示person类的属性。

Column：表示sql查询出来的字段名。

Column和property放在一块儿表示将sql查询出来的字段映射到指定的pojo类属性上。

<result />：普通结果，即pojo的属性。

这里只将sql查询出来的字段与pojo属性名不一致的进行了定义，通过后边的测试pojo属性名和sql字段相同的自动进行映射。

#### Mapper.xml定义

<!-- 获取用户列表返回resultMap -->

<select id=*"selectUserListResultMap"* resultMap=*"personmap"*>

select \* from user

</select>

使用resultMap指定上边定义的personmap。

#### Mapper接口定义

public List<Person> selectUserListResultMap() throws Exception;

实际返回的类型是Person类型。

#### 测试：

**public** **void** testselectUserListResultMap() **throws** Exception{

//获取session

SqlSession session = sqlSessionFactory.openSession();

//获限mapper接口实例

UserMapper userMapper = session.getMapper(UserMapper.**class**);

User user = **new** User();

user.setUsername("管理员");

//查询用户列表返回resultMap

List<Person> list = userMapper.selectUserListResultMap();

System.*out*.println(list);

//关闭session

session.close();

}

### 动态sql

Mybatis提供使用ognl表达式动态生成sql的功能。

#### If

<!-- 传递pojo综合查询用户信息 -->

<select id=*"selectUserByUser"* parameterType=*"user"* resultType=*"user"*>

select \* from user

where 1=1

<if test=*"id!=null and id!=''"*>

and id=#{id}

</if>

<if test=*"username!=null and username!=''"*>

and username like '%${username}%'

</if>

</select>

**注意要做不等于空字符串校验。**

#### Where

上边的sql也可以改为：

<select id=*"selectUserByUser"* parameterType=*"user"* resultType=*"user"*>

select \* from user

<where>

<if test=*"id!=null and id!=''"*>

and id=#{id}

</if>

<if test=*"username!=null and username!=''"*>

and username like '%${username}%'

</if>

</where>

</select>

<where /> 可以自动处理第一个and。

#### foreach

向sql传递数组或List，mybatis使用foreach解析，如下：

##### 传递List

传递List类型在编写mapper.xml没有区别，唯一不同的是只有一个List参数时它的参数名为list。

如下：

###### Mapper.xml

<select id=*"selectUserByList"* parameterType=*"java.util.List"* resultType=*"user"*>

select \* from user

<where>

<!-- 传递List，List中是pojo -->

<if test=*"list!=null"*>

<foreach collection=*"list"* item=*"item"* open=*"and id in("* separator=*","* close=*")"* >

#{item.id}

</foreach>

</if>

</where>

</select>

###### Mapper接口

public List<User> selectUserByList(List userlist) throws Exception;

###### 测试：

**public** **void** testselectUserByList()**throws** Exception{

//获取session

SqlSession session = sqlSessionFactory.openSession();

//获限mapper接口实例

UserMapper userMapper = session.getMapper(UserMapper.**class**);

//构造查询条件List

List<User> userlist = **new** ArrayList<User>();

User user = **new** User();

user.setId(1);

userlist.add(user);

user = **new** User();

user.setId(2);

userlist.add(user);

//传递userlist列表查询用户列表

List<User> list = userMapper.selectUserByList(userlist);

//关闭session

session.close();

}

##### 传递数组（数组中是pojo）：

请阅读文档学习。

###### Mapper.xml

<!-- 传递数组综合查询用户信息 -->

<select id=*"selectUserByArray"* parameterType=*"Object[]"* resultType=*"user"*>

select \* from user

<where>

<!-- 传递数组 -->

<if test=*"array!=null"*>

<foreach collection=*"array"* index=*"index"* item=*"item"* open=*"and id in("* separator=*","* close=*")"* >

#{item.id}

</foreach>

</if>

</where>

</select>

sql只接收一个数组参数，这时sql解析参数的名称mybatis固定为array ，如果数组是通过一个pojo传递到sql则参数的名称为pojo中的属性名。

index：为数组的下标。

item：为数组每个元素的名称，名称随意定义

open：循环开始

close：循环结束

separator：中间分隔输出

###### Mapper接口：

public List<User> selectUserByArray(Object[] userlist) throws Exception;

###### 测试：

**public** **void** testselectUserByArray()**throws** Exception{

//获取session

SqlSession session = sqlSessionFactory.openSession();

//获限mapper接口实例

UserMapper userMapper = session.getMapper(UserMapper.**class**);

//构造查询条件List

Object[] userlist = **new** Object[2];

User user = **new** User();

user.setId(1);

userlist[0]=user;

user = **new** User();

user.setId(2);

userlist[1]=user;

//传递user对象查询用户列表

List<User> list = userMapper.selectUserByArray(userlist);

//关闭session

session.close();

}

##### 传递数组（数组中是字符串类型）：

请阅读文档学习。

###### Mapper.xml

<!-- 传递数组综合查询用户信息 -->

<select id=*"selectUserByArray"* parameterType=*"Object[]"* resultType=*"user"*>

select \* from user

<where>

<!-- 传递数组 -->

<if test=*"array!=null"*>

<foreach collection=*"array"* index=*"index"* item=*"item"* open=*"and id in("* separator=*","* close=*")"* >

#{item}

</foreach>

</if>

</where>

</select>

如果数组中是简单类型则写为#{item} ，不用再通过ognl获取对象属性值了。

###### Mapper接口：

public List<User> selectUserByArray(Object[] userlist) throws Exception;

###### 测试：

**public** **void** testselectUserByArray()**throws** Exception{

//获取session

SqlSession session = sqlSessionFactory.openSession();

//获限mapper接口实例

UserMapper userMapper = session.getMapper(UserMapper.**class**);

//构造查询条件List

Object[] userlist = **new** Object[2];

userlist[0]=”1”;

userlist[1]= ”2”;

//传递user对象查询用户列表

List<User> list = userMapper.selectUserByArray(userlist);

//关闭session

session.close();

}

#### set

参考pdf文档自行学习

### Sql片段

#### 需求

Sql中可将重复的sql提取出来，使用时用include引用即可，最终达到sql重用的目的，如下：

<!-- 传递pojo综合查询用户信息 -->

<select id=*"selectUserByUser"* parameterType=*"user"* resultType=*"user"*>

select \* from user

<where>

<if test=*"id!=null and id!=''"*>

and id=#{id}

</if>

<if test=*"username!=null and username!=''"*>

and username like '%${username}%'

</if>

</where>

</select>

#### 将where条件抽取出来：

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

<if test=*"id!=null and id!=''"*>

and id=#{id}

</if>

<if test=*"username!=null and username!=''"*>

and username like '%${username}%'

</if>

</sql>

#### 使用include引用：

<select id=*"selectUserByUser"* parameterType=*"user"* resultType=*"user"*>

select \* from user

<where>

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

</where>

</select>

注意：如果引用其它mapper.xml的sql片段，则在引用时需要加上namespace，如下：

<include refid=*"*namespace.sql片段”/>

### 一对一查询

案例：查询所有订单信息，订单信息中显示下单人信息。

注意：因为一个订单信息只会是一个人下的订单，所以从查询订单信息出发关联查询用户信息为一对一查询。如果从用户信息出发查询用户下的订单信息则为一对多查询，因为一个用户可以下多个订单。

#### 方法一：

使用resultType，定义订单信息po类，此po类中包括了订单信息和用户信息：

##### Sql语句：

SELECT

orders.\*,

user.username,

user.address

FROM

orders,

USER

WHERE orders.user\_id = user.id

##### 定义po类

Po类中应该包括上边sql查询出来的所有字段，如下：

**public** **class** UserOrder **extends** Orders {

**private** String username;// 用户姓名

**private** String address;// 地址

get/set。。。。

UserOrder类继承Orders类后UserOrder类包括了Orders类的所有字段，只需要定义用户的信息字段即可。

##### Mapper.xml

<!-- 查询所有订单信息 -->

<select id=*"findOrdersList"* resultType=*"cn.itcast.mybatis.po.UserOrder"*>

SELECT

orders.\*,

user.id user\_id,

user.username,

user.address

FROM

orders, USER

WHERE orders.user\_id = user.id

</select>

##### Mapper接口：

public List<UserOrder> findOrdersList() throws Exception;

##### 测试：

**public** **void** testfindOrdersList()**throws** Exception{

//获取session

SqlSession session = sqlSessionFactory.openSession();

//获限mapper接口实例

UserMapper userMapper = session.getMapper(UserMapper.**class**);

//查询订单信息

List<UserOrder> list = userMapper.findOrdersList();

System.*out*.println(list);

//关闭session

session.close();

}

##### 总结：

定义专门的po类作为输出类型，其中定义了sql查询结果集所有的字段。此方法较为简单，企业中使用普遍。

#### 方法二：

使用resultMap，定义专门的resultMap用于映射一对一查询结果。

##### Sql语句：

SELECT

orders.\*,

user.username,

user.address

FROM

orders,

USER

WHERE orders.user\_id = user.id

##### 定义po类

在Orders类中加入User属性。

##### Mapper.xml

<select id=*"findOrdersList2"* resultMap=*"userordermap"*>

SELECT

orders.\*,

user.username,

user.address

FROM

orders, USER

WHERE orders.user\_id = user.id

</select>

这里resultMap指定*userordermap。*

##### 定义resultMap

<!-- 订单信息resultmap -->

<resultMap type=*"cn.itcast.mybatis.po.Orders"* id=*"userordermap"*>

<!-- 这里的id，是mybatis在进行一对一查询时将user字段映射为user对象时要使用，必须写 -->

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

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

<result property=*"order\_number"* column=*"order\_number"* />

<association property=*"user"* javaType=*"cn.itcast.mybatis.po.User"*>

<!-- 这里的id为user的id，如果写上表示给user的id属性赋值 -->

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

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

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

</association>

</resultMap>

association：表示进行关联查询单条记录

property：表示关联查询的结果存储在cn.itcast.mybatis.po.Orders的user属性中

javaType：表示关联查询的结果类型

<id property=*"id"* column=*"user\_id"* />：查询结果的user\_id列对应关联对象的id属性，这里是<id />表示user\_id是关联查询对象的唯一标识。

<result property=*"username"* column=*"username"* />：查询结果的username列对应关联对象的username属性。

##### Mapper接口：

public List<Orders> findOrdersList2() throws Exception;

##### 测试：

**public** **void** testfindOrdersList2()**throws** Exception{

//获取session

SqlSession session = sqlSessionFactory.openSession();

//获限mapper接口实例

UserMapper userMapper = session.getMapper(UserMapper.**class**);

//查询订单信息

List<Orders> list = userMapper.findOrdersList2();

System.*out*.println(list);

//关闭session

session.close();

}

##### 总结：

此种方法使用了mybatis的association标签用于一对一关联查询，将查询结果映射至对象中。

### 一对多查询

案例：查询所有订单信息及订单下的订单明细信息。

订单信息与订单明细为一对多关系，一个订单包括多个商品信息。

使用resultMap实现如下：

##### Sql语句：

SELECT

orders.\*,

user.username,

user.address,

orderdetail.id orderdetail\_id,

orderdetail.item\_id,

orderdetail.item\_num,

orderdetail.item\_price

FROM

orders,USER ,orderdetail

WHERE orders.user\_id = user.id

AND orders.id = orderdetail.orders\_id

##### 定义po类

在Orders类中加入User属性。

在Orders类中加入List<Orderdetail> orderdetails 属性

##### Mapper.xml

<select id=*"findOrdersDetailList"* resultMap=*"userorderdetailmap"*>

SELECT

orders.\*,

user.username,

user.address,

orderdetail.id orderdetail\_id,

orderdetail.item\_id,

orderdetail.item\_num,

orderdetail.item\_price

FROM orders,USER ,orderdetail

WHERE orders.user\_id = user.id

AND orders.id = orderdetail.orders\_id

</select>

##### 定义resultMap

<!-- 订单信息resultmap -->

<resultMap type=*"cn.itcast.mybatis.po.Orders"* id=*"userorderdetailmap"*>

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

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

<result property=*"order\_number"* column=*"order\_number"* />

<association property=*"user"* javaType=*"cn.itcast.mybatis.po.User"*>

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

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

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

</association>

<collection property=*"orderdetails"* ofType=*"cn.itcast.mybatis.po.Orderdetail"*>

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

<result property=*"item\_id"* column=*"item\_id"* />

<result property=*"item\_num"* column=*"item\_num"* />

<result property=*"item\_price"* column=*"item\_price"* />

</collection>

</resultMap>

黄色部分和上边一对一查询订单及用户信息定义的resultMap相同，

collection部分定义了查询订单明细信息。

collection：表示关联查询结果集

property=*"orderdetails"：*关联查询的结果集存储在cn.itcast.mybatis.po.Orders上哪个属性。

ofType=*"cn.itcast.mybatis.po.Orderdetail"：*指定关联查询的结果集中的对象类型即List中的对象类型。

<id />及<result/>的意义同一对一查询。

##### Mapper接口：

public List<Orders> findOrdersDetailList () throws Exception;

##### 测试：

**public** **void** testfindOrdersDetailList()**throws** Exception{

//获取session

SqlSession session = sqlSessionFactory.openSession();

//获限mapper接口实例

UserMapper userMapper = session.getMapper(UserMapper.**class**);

//查询订单信息

List<Orders> list = userMapper.findOrdersDetailList();

System.*out*.println(list);

//关闭session

session.close();

}

##### 总结：

此种方法使用了mybatis的collection标签用于一对多关联查询，将查询结果映射至集合对象中。

##### resultMap使用继承

上边定义的resultMap中黄色部分和一对一查询订单信息的resultMap相同，这里使用继承可以不再填写重复的内容，如下：

<resultMap type=*"cn.itcast.mybatis.po.Orders"* id=*"userorderdetailmap2"* extends=*"userordermap"*>

<collection property=*"orderdetails"* ofType=*"cn.itcast.mybatis.po.Orderdetail"*>

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

<result property=*"item\_id"* column=*"item\_id"* />

<result property=*"item\_num"* column=*"item\_num"* />

<result property=*"item\_price"* column=*"item\_price"* />

</collection>

</resultMap>

使用extends继承订单信息*userordermap。*

### 多对多查询

案例：查询所有订单信息及订单明细的商品信息。

订单信息与商品信息为多对多关系，因为一个订单包括多个商品信息，一个商品可以在多个订单中存在，订单信息与商品信息的多对多关系是通过订单明细表进行关联。

##### Sql语句：

SELECT

orders.\*,

user.username,

user.address,

orderdetail.id orderdetail\_id,

orderdetail.item\_id,

orderdetail.item\_num,

orderdetail.item\_price,

items.item\_name,

items.item\_detail

FROM

orders,USER ,orderdetail,items

WHERE orders.user\_id = user.id

AND orders.id = orderdetail.orders\_id

AND orderdetail.item\_id = items.id

##### 定义po类

在Orders类中加入User属性。

在Orders类中加入List<Orderdetail> orderdetails 属性，存储订单明细信息

在Orderdetail类中加入Items items 属性存储商品信息

##### 定义resultMap

<!-- 订单商品信息resultmap -->

<resultMap type=*"cn.itcast.mybatis.po.Orders"* id=*"userorderitemsmap"*

extends=*"userordermap"*>

<collection property=*"orderdetails"* ofType=*"cn.itcast.mybatis.po.Orderdetail"*>

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

<result property=*"item\_id"* column=*"item\_id"* />

<result property=*"item\_num"* column=*"item\_num"* />

<result property=*"item\_price"* column=*"item\_price"* />

<!-- 商品信息 -->

<association property=*"items"* javaType=*"cn.itcast.mybatis.po.Items"*>

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

<result property=*"item\_name"* column=*"item\_name"* />

<result property=*"item\_detail"* column=*"item\_detail"* />

</association>

</collection>

</resultMap>

在collection中加入association通过订单明细表关联查询商品信息

##### Mapper.xml

<select id=*"findOrdersItemsList"* resultMap=*"userorderitemsmap"*>

SELECT

orders.\*,

user.username,

user.address,

orderdetail.id orderdetail\_id,

orderdetail.item\_id,

orderdetail.item\_num,

orderdetail.item\_price,

items.item\_name,

items.item\_detail

FROM

orders,USER ,orderdetail,items

WHERE orders.user\_id = user.id

AND orders.id = orderdetail.orders\_id

AND orderdetail.item\_id = items.id

</select>

##### Mapper接口：

public List<Orders> findOrdersItemsList () throws Exception;

##### 测试：

**public** **void** findOrdersItemsList()**throws** Exception{

//获取session

SqlSession session = sqlSessionFactory.openSession();

//获限mapper接口实例

UserMapper userMapper = session.getMapper(UserMapper.**class**);

//查询订单信息

List<Orders> list = userMapper.findOrdersItemsList();

System.*out*.println(list);

//关闭session

session.close();

}

##### 总结：

所谓一对多查询、多对多查询都对于具体的业务分析来说，使用mybatis提交的collection和association可以完成不同的关联查询需求，通常在实际应用时association用自定义pojo方式代替，关联查询结果集使用collection完成。

### 延迟加载

需要查询关联信息时，使用mybatis延迟加载特性可有效的减少数据库压力，首次查询只查询主要信息，关联信息等用户获取时再加载。

#### 打开延迟加载开关

在mybatis核心配置文件中配置：

lazyLoadingEnabled、aggressiveLazyLoading

|  |  |  |  |
| --- | --- | --- | --- |
| 设置项 | 描述 | 允许值 | 默认值 |
| lazyLoadingEnabled | 全局性设置懒加载。如果设为‘false’，则所有相关联的都会被初始化加载。 | true | false | false |
| aggressiveLazyLoading | 当设置为‘true’的时候，懒加载的对象可能被任何懒属性全部加载。否则，每个属性都按需加载。 | true | false | true |

<settings>

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

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

</settings>

#### 一对一查询延迟加载

##### Sql语句：

SELECT

orders.\*

FROM

orders

##### 定义po类

在Orders类中加入User属性。

##### 定义resultMap

<!-- 订单信息resultmap -->

<resultMap type=*"cn.itcast.mybatis.po.Orders"* id=*"userordermap2"*>

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

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

<result property=*"order\_number"* column=*"order\_number"* />

<association property=*"user"* javaType=*"cn.itcast.mybatis.po.User"* select=*"selectUserById"* column=*"user\_id"* />

</resultMap>

association：

select=*"selectUserById"*：指定关联查询sql为*selectUserById*

column=*"user\_id"*：关联查询时将user\_id列的值传入*selectUserById*

最后将关联查询结果映射至*cn.itcast.mybatis.po.User。*

##### Mapper.xml

<select id=*"findOrdersList3"* resultMap=*"userordermap2"*>

SELECT

orders.\*

FROM

orders

</select>

##### Mapper接口：

public List<Orders> findOrdersList3() throws Exception;

##### 测试：

**public** **void** testfindOrdersList3()**throws** Exception{

//获取session

SqlSession session = sqlSessionFactory.openSession();

//获限mapper接口实例

UserMapper userMapper = session.getMapper(UserMapper.**class**);

//查询订单信息

List<Orders> list = userMapper.findOrdersList3();

System.out.println(list);

//开始加载，通过orders.getUser方法进行加载

for(Orders orders:list){

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

}

//关闭session

session.close();

}

##### 总结：

使用延迟加载提高数据库查询性能，默认不查询关联数据，按需要发出sql请求关联查询信息。

## 缓存

### 一级缓存

Mybatis一级缓存的作用域是同一个SqlSession。

#### 第一个例子：

//获取session

SqlSession session = sqlSessionFactory.openSession();

//获限mapper接口实例

UserMapper userMapper = session.getMapper(UserMapper.**class**);

//第一次查询

User user1 = userMapper.selectUserById(1);

System.*out*.println(user1);

//第二次查询，由于是同一个session则不再向数据发出语句直接从缓存取出

User user2 = userMapper.selectUserById(1);

System.*out*.println(user2);

##### 原理：

Mybatis首先去缓存中查询结果集，如果没有则查询数据库，如果有则从缓存取出返回结果集就不走数据库。

Mybatis内部存储缓存使用一个HashMap，key为hashCode+sqlId+Sql语句。value为从查询出来映射生成的java对象

#### 第二个例子：

//获取session

SqlSession session = sqlSessionFactory.openSession();

//获限mapper接口实例

UserMapper userMapper = session.getMapper(UserMapper.**class**);

//第一次查询

User user1 = userMapper.selectUserById(1);

System.*out*.println(user1);

//在同一个session执行更新

User user\_update = **new** User();

user\_update.setId(1);

user\_update.setUsername("李奎");

userMapper.updateUser(user\_update);

session.commit();

//第二次查询，虽然是同一个session但是由于执行了更新操作session的缓存被清空，这里重新发出sql操作

User user2 = userMapper.selectUserById(1);

System.*out*.println(user2);

##### 原理

该例子与第一个例子不同的是在两次查询中间加入了更新，更新操作执行后mybatis执行了清除缓存即清空HashMap。

### 二级缓存

Mybatis的二级缓存即查询缓存，它的作用域是一个mapper的namespace，即在同一个namespace中查询sql可以从缓存中获取数据。

二级缓存是可以跨SqlSession的。

#### 开启二级缓存：

1. 在核心配置文件SqlMapConfig.xml中加入

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

|  |  |  |  |
| --- | --- | --- | --- |
|  | 描述 | 允许值 | 默认值 |
| cacheEnabled | 对在此配置文件下的所有cache 进行全局性开/关设置。 | true false | true |

1. 要在你的Mapper映射文件中添加一行：  **<cache />**
2. 在select语句中useCache=false可以禁用当前的语句的二级缓存，即每次查询夸session 的查询都会发出sql去查询，默认情况是true，即该sql使用二级缓存。

#### 例子：

注意：将查询结果的pojo对象进行序列化实现 java.io.Serializable接口

//获取session1

SqlSession session1 = sqlSessionFactory.openSession();

UserMapper userMapper = session1.getMapper(UserMapper.**class**);

//使用session1执行第一次查询

User user1 = userMapper.selectUserById(1);

System.*out*.println(user1);

//关闭session1

session1.close();

//获取session2

SqlSession session2 = sqlSessionFactory.openSession();

UserMapper userMapper2 = session2.getMapper(UserMapper.**class**);

//使用session2执行第二次查询，由于开启了二级缓存这里从缓存中获取数据不再向数据库发出sql

User user2 = userMapper2.selectUserById(1);

System.*out*.println(user2);

//关闭session2

session2.close();

#### 刷新缓存

在mapper的同一个namespace中，如果有其它insert、update、delete操作数据后需要刷新缓存，如果不执行刷新缓存会出现脏读。

4. sql中的 flushCache=*"true"* 属性，默认情况下为true即刷新缓存，如果改成false则不会刷新。使用缓存时如果手动修改数据库表中的查询数据会出现脏读。

如下：

<insert id="insertUser" parameterType="cn.itcast.mybatis.po.User" flushCache="true">

#### cache  的其它参数：

flushInterval（刷新间隔）可以被设置为任意的正整数，而且它们代表一个合理的毫秒形式的时间段。默认情况是不设置，也就是没有刷新间隔，缓存仅仅调用语句时刷新。

size（引用数目）可以被设置为任意正整数，要记住你缓存的对象数目和你运行环境的可用内存资源数目。默认值是1024。

readOnly（只读）属性可以被设置为true或false。只读的缓存会给所有调用者返回缓存对象的相同实例。因此这些对象不能被修改。这提供了很重要的性能优势。可读写的缓存会返回缓存对象的拷贝（通过序列化）。这会慢一些，但是安全，因此默认是false。

如下例子：

<cache  eviction="FIFO"  flushInterval="60000"  size="512"  readOnly="true"/>

这个更高级的配置创建了一个 FIFO 缓存,并每隔 60 秒刷新,存数结果对象或列表的 512 个引用,而且返回的对象被认为是只读的,因此在不同线程中的调用者之间修改它们会 导致冲突。可用的收回策略有, 默认的是 LRU:

1. LRU – 最近最少使用的:移除最长时间不被使用的对象。
2. FIFO – 先进先出:按对象进入缓存的顺序来移除它们。
3. SOFT – 软引用:移除基于垃圾回收器状态和软引用规则的对象。
4. WEAK – 弱引用:更积极地移除基于垃圾收集器状态和弱引用规则的对象。

### 二级缓存使用Ehcache

Mybatis与缓存框架ehcache进行了整合，采用ehcache框架管理缓存数据。

#### 第一步：引入缓存的依赖包



#### 第二步：引入缓存配置文件

ehcache.xml

defaultCache配置说明：

maxElementsInMemory 内存中最大缓存对象数.当超过最大对象数的时候,ehcache会按指定的策略去清理内存  
eternal 缓存对象是否永久有效,一但设置了,timeout将不起作用.  
timeToIdleSeconds 设置Element在失效前的允许闲置时间.仅当element不是永久有效时使用,可选属性,默认值是0,也就是可闲置时间无穷大.  
timeToLiveSeconds：设置Element在失效前允许存活时间.最大时间介于创建时间和失效时间之间.仅当element是永久有效时使用,默认是0.,也就是element存活时间无穷大.  
overflowToDisk 配置此属性,当内存中Element数量达到maxElementsInMemory时,Ehcache将会Element写到磁盘中.  
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.  
maxElementsOnDisk 磁盘中最大缓存对象数,若是0表示无穷大.  
diskPersistent 是否在重启服务的时候清楚磁盘上的缓存数据.true不清除.  
diskExpiryThreadIntervalSeconds 磁盘失效线程运行时间间隔.  
memoryStoreEvictionPolicy：当达到maxElementsInMemory限制时,Ehcache将会根据指定的策略去清理内存.默认策略是LRU(最近最少使用).你可以设置为FIFO(先进先出)或是LFU(较少使用).

#### 第三步：修改mapper文件中缓存类型

在cache中指定type。

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

## Mybatis与springmvc整合

### Dao

#### Spring配置文件：

##### applicationContext.xml

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

<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-3.1.xsd*

*http://www.springframework.org/schema/mvc*

*http://www.springframework.org/schema/mvc/spring-mvc-3.1.xsd*

*http://www.springframework.org/schema/context*

*http://www.springframework.org/schema/context/spring-context-3.1.xsd*

*http://www.springframework.org/schema/aop*

*http://www.springframework.org/schema/aop/spring-aop-3.1.xsd*

*http://www.springframework.org/schema/tx*

*http://www.springframework.org/schema/tx/spring-tx-3.1.xsd "*>

<!-- 引用配置文件 -->

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

<bean id=*"dataSource"* class=*"org.apache.commons.dbcp.BasicDataSource"*

destroy-method=*"close"*>

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

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

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

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

<property name=*"maxActive"* value=*"30"* />

<property name=*"maxIdle"* value=*"5"* />

</bean>

</beans>

##### applicationContext-dao.xml

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

<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-3.1.xsd*

*http://www.springframework.org/schema/mvc*

*http://www.springframework.org/schema/mvc/spring-mvc-3.1.xsd*

*http://www.springframework.org/schema/context*

*http://www.springframework.org/schema/context/spring-context-3.1.xsd*

*http://www.springframework.org/schema/aop*

*http://www.springframework.org/schema/aop/spring-aop-3.1.xsd*

*http://www.springframework.org/schema/tx*

*http://www.springframework.org/schema/tx/spring-tx-3.1.xsd "*>

<!-- 会话工厂 -->

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

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

<!-- 加载mybatis的配置文件 -->

<property name=*"configLocation"* value=*"classpath:mybatis/sqlMapConfig.xml"*></property>

</bean>

<!-- mapper扫描器，这里由于没有在sqlMapConfig配置mapper，所以必须保证mapper和dao接口在同一个目录且同名 -->

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

<property name=*"basePackage"* value=*"yycg.\*\*.dao.mapper"*></property>

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

</bean>

<!-- 如果采用自动扫描器则不用手动设置工厂bean

<bean id="useryyMapper2" class="org.mybatis.spring.mapper.MapperFactoryBean">

<property name="mapperInterface"

value="yycg.dao.mapper.UserMapper" />

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

</bean> -->

</beans>

##### 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>

<!—使用自动扫描器时，mapper.xml文件如果和mapper.java接口在一个目录则此处不用定义mappers -->

<mappers>

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

</mappers>

</configuration>

##### Mapper编写的三种方法

###### 接口实现类继承SqlSessionDaoSupport

使用此种方法需要编写mapper接口，mapper接口实现类、mapper.xml文件

1. 在sqlMapConfig.xml中配置mapper.xml的位置

<mappers>

<mapper resource=*"mapper.xml文件的地址"* />

<mapper resource=*"mapper.xml文件的地址"* />

</mappers>

1. 定义mapper接口
2. 实现类集成SqlSessionDaoSupport

mapper方法中可以this.getSqlSession()进行数据增删改查。

1. spring 配置

<bean id=*" "* class=*"mapper接口的实现"*>

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

</bean>

###### 使用org.mybatis.spring.mapper.MapperFactoryBean

1. 在sqlMapConfig.xml中配置mapper.xml的位置

如果mapper.xml和mappre接口的名称相同且在同一个目录，这里可以不用配置

<mappers>

<mapper resource=*"mapper.xml文件的地址"* />

<mapper resource=*"mapper.xml文件的地址"* />

</mappers>

1. 定义mapper接口

**注意**

**1、mapper.xml中的namespace为mapper接口的地址**

**2、mapper接口中的方法名和mapper.xml中的定义的statement的id保持一致**

1. Spring中定义

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

<property name=*"mapperInterface"* value=*"mapper接口地址"* />

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

</bean>

###### 使用mapper扫描器

1. mapper.xml文件编写，

**注意：**

**mapper.xml中的namespace为mapper接口的地址**

**mapper接口中的方法名和mapper.xml中的定义的statement的id保持一致**

**如果将mapper.xml和mapper接口的名称保持一致则不用在sqlMapConfig.xml中进行配置**

1. 定义mapper接口

**注意mapper.xml的文件名和mapper的接口名称保持一致，且放在同一个目录**

1. 配置mapper扫描器

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

<property name=*"basePackage"* value=*"mapper接口包地址"*></property>

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

</bean>

1. 使用扫描器后从spring容器中获取mapper的实现对象

扫描器将接口通过代理方法生成实现对象，要spring容器中自动注册，名称为mapper 接口的名称。

### Service

##### UserManager接口

编写UserManagerService接口，如下：

**public** **interface** UserManagerService {

/\*\*

\* 根据id查询用户

\*/

**public** User findUserById(String id) **throws** Exception;

}

**public** **class** UserManagerServiceImpl **implements** UserManagerService {

@Autowired

UserMapper userMapper;

@Override

**public** User findUserById(**int** id) **throws** Exception {

**return** userMapper.selectUserById(id);

}

}

##### Spring配置文件：

将*userManager*在spring配置文件进行配置

###### applicationContext--service.xml

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

<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-3.1.xsd*

*http://www.springframework.org/schema/mvc*

*http://www.springframework.org/schema/mvc/spring-mvc-3.1.xsd*

*http://www.springframework.org/schema/context*

*http://www.springframework.org/schema/context/spring-context-3.1.xsd*

*http://www.springframework.org/schema/aop*

*http://www.springframework.org/schema/aop/spring-aop-3.1.xsd*

*http://www.springframework.org/schema/tx*

*http://www.springframework.org/schema/tx/spring-tx-3.1.xsd "*>

<!-- 用户管理-->

<bean id=*"userManagerService"* class=*"cn.itcast.mybatis.service.impl.UserManagerServiceImpl"* />

</beans>

##### Serivce测试：

ApplicationContext applicationContext;

**protected** **void** setUp() **throws** Exception {

applicationContext = **new** ClassPathXmlApplicationContext(

**new** String[]{

"spring/applicationContext.xml",

"spring/applicationContext-dao.xml",

"spring/applicationContext-service.xml"

}

);

}

**public** **void** testFindUserById() **throws** Exception {

UserManagerService userManagerService = (UserManagerService)applicationContext.getBean("userManagerService");

System.*out*.println(userManagerService.findUserById(1));

}

##### 事务控制：

###### 配置

在applicaitonContext.xml中配置事务管理器

<!-- 事务控制 -->

<bean id=*"txManager-base"*

class=*"org.springframework.jdbc.datasource.DataSourceTransactionManager"*>

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

</bean>

<tx:advice id=*"txAdvice-base"* transaction-manager=*"txManager-base"*>

<tx:attributes>

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

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

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

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

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

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

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

</tx:attributes>

</tx:advice>

<aop:config proxy-target-class=*"true"*>

<aop:advisor

pointcut=*"execution(\* cn.itcast.\*\*.service.impl.\*.\*(..))"*

advice-ref=*"txAdvice-base"* />

</aop:config>

###### 事务测试

在一个service方法中先执行更新，再执行插入，插入一个违反唯一约束的记录，如果数据不回滚则说明事务没有控制。

### Action

#### spingmvc.xml配置文件

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

<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-3.1.xsd*

*http://www.springframework.org/schema/mvc*

*http://www.springframework.org/schema/mvc/spring-mvc-3.1.xsd*

*http://www.springframework.org/schema/context*

*http://www.springframework.org/schema/context/spring-context-3.1.xsd*

*http://www.springframework.org/schema/aop*

*http://www.springframework.org/schema/aop/spring-aop-3.1.xsd*

*http://www.springframework.org/schema/tx*

*http://www.springframework.org/schema/tx/spring-tx-3.1.xsd "*>

<!-- 注解驱动 -->

<mvc:annotation-driven/>

<!-- 组件扫描，用于控制层 -->

<context:component-scan base-package=*"cn.itcast.mybatis.action"* />

<!-- 视图解析器 -->

<bean

class=*"org.springframework.web.servlet.view.InternalResourceViewResolver"*>

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

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

</bean>

<!-- 拦截器 -->

<!-- <mvc:interceptors>

多个拦截器,顺序执行

<mvc:interceptor>

<mvc:mapping path="/\*\*" />

<bean class="cn.itcast.project.yycg.base.filter.LoginInterceptor"></bean>

</mvc:interceptor>

<mvc:interceptor>

<mvc:mapping path="/\*\*" />

<bean class="cn.itcast.project.yycg.base.filter.PermissionInterceptor"></bean>

</mvc:interceptor>

</mvc:interceptors> -->

</beans>

#### 编写UserAction.java

/\*\*

\* 用户管理

\* **@author** Thinkpad

\*

\*/

@Controller

@RequestMapping("/user")

**public** **class** UserAction {

@Autowired

UserManagerService userManagerService;

/\*\*

\* 用户修改

\* **@param** model

\* **@param** id

\* **@return**

\* **@throws** Exception

\*/

@RequestMapping("/useredit")

**public** String useredit(Model model,**int** id)**throws** Exception{

User user = userManagerService.findUserById(id);

model.addAttribute("user", user);

**return** "useredit";

}

/\*\*

\* 用户修改提交

\* **@param** user

\* **@return**

\* **@throws** Exception

\*/

@RequestMapping("/usereditsubmit")

**public** String usereditsubmit(User user)**throws** Exception{

userManagerService.saveUser(user);

**return** "success";

}

//其它方法略

//……

}

注意：学会如果在action中调用service，处理结果返回用户。

#### web.xml

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

<web-app xmlns:xsi=*"http://www.w3.org/2001/XMLSchema-instance"* xmlns=*"http://java.sun.com/xml/ns/javaee"* xsi:schemaLocation=*"http://java.sun.com/xml/ns/javaee http://java.sun.com/xml/ns/javaee/web-app\_3\_0.xsd"* id=*"WebApp\_ID"* version=*"3.0"*>

<display-name>mybatis\_03</display-name>

<context-param>

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

<param-value>/WEB-INF/classes/spring/applicationContext.xml,/WEB-INF/classes/spring/applicationContext-\*.xml</param-value>

</context-param>

<listener>

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

</listener>

<filter>

<filter-name>SpringCharacterEncodingFilter</filter-name>

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

<init-param>

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

<param-value>UTF-8</param-value>

</init-param>

</filter>

<filter-mapping>

<filter-name>SpringCharacterEncodingFilter</filter-name>

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

</filter-mapping>

<servlet>

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

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

<init-param>

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

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

</init-param>

</servlet>

<servlet-mapping>

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

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

</servlet-mapping>

<welcome-file-list>

<welcome-file>index.jsp</welcome-file>

</welcome-file-list>

</web-app>

#### 测试

将工程部署在tomcat运行，输入：http://localhost:8080/mybatis\_03/user/useredit.aciton?id=1，进入首页

## Mybatis逆向工程

使用官方网站的mapper自动生成工具mybatis-generator-core-1.3.2来生成po类和mapper映射文件。

### 第一步mapper生成配置文件：

在generatorConfig.xml中配置mapper生成的详细信息， 注意改下几点：

1. 添加要生成的数据库表
2. po文件所在包路径
3. mapper文件所在包路径

配置文件如下：

详见generatorSqlmapCustom工程

### 第二步使用java类生成mapper文件：

**public** **void** generator() **throws** Exception{

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

**boolean** overwrite = **true**;

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** {

GeneratorSqlmap generatorSqlmap = **new** GeneratorSqlmap();

generatorSqlmap.generator();

} **catch** (Exception e) {

e.printStackTrace();

}

}

### 第三步：拷贝生成的mapper文件到工程中指定的目录中

#### Mapper.xml

Mapper.xml的文件拷贝至mapper目录内

#### Mapper.java

Mapper.java的文件拷贝至mapper 目录内

注意：mapper xml文件和mapper.java 文件在一个目录内且文件名相同。

### 第四步Mapper接口测试

学会使用mapper自动生成的增、删、改、查方法。

//删除符合条件的记录

**int** deleteByExample(UserExample example);

//根据主键删除

**int** deleteByPrimaryKey(String id);

//插入对象所有字段

**int** insert(User record);

//插入对象不为空的字段

**int** insertSelective(User record);

//自定义查询条件查询结果集

List<User> selectByExample(UserExample example);

//根据主键查询

User selectByPrimaryKey(String id);

//根据主键将对象中不为空的值更新至数据库

**int** updateByPrimaryKeySelective(User record);

//根据主键将对象中所有字段的值更新至数据库

**int** updateByPrimaryKey(User record);

#### 注意：

##### Mapper文件内容不覆盖而是追加

XXXMapper.xml文件已经存在时，如果进行重新生成则mapper.xml文件内容不被覆盖而是进行内容追加，结果导致mybatis解析失败。

解决方法：删除原来已经生成的mapper xml文件再进行生成。

Mybatis自动生成的po及mapper.java文件不是内容而是直接覆盖没有此问题。

##### Table schema问题

下边是关于针对oracle数据库表生成代码的schema问题：

Schma即数据库模式，oracle中一个用户对应一个schema，可以理解为用户就是schema。

当Oralce数据库存在多个schema可以访问相同的表名时，使用mybatis生成该表的mapper.xml将会出现mapper.xml内容重复的问题，结果导致mybatis解析错误。

解决方法：在table中填写schema，如下：

<table schema="XXXX" tableName=" " >

XXXX即为一个schema的名称，生成后将mapper.xml的schema前缀批量去掉，如果不去掉当oracle用户变更了sql语句将查询失败。

快捷操作方式：mapper.xml文件中批量替换：“from XXXX.”为空

Oracle查询对象的schema可从dba\_objects中查询，如下：

select \* from dba\_objects