# 课程计划

第一天：

1. mybatis的介绍
2. Mybatis的入门
   1. 使用jdbc操作数据库存在的问题
   2. Mybatis的架构
   3. Mybatis的入门程序
3. Dao的开发方法
   1. 原始dao的开发方法
   2. 动态代理方式
4. SqlMapConfig.xml文件说明

# MyBatis介绍

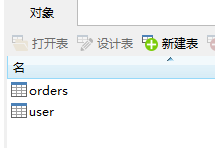
MyBatis 本是apache的一个开源项目iBatis, 2010年这个项目由apache software foundation 迁移到了google code，并且改名为MyBatis 。2013年11月迁移到Github。 MyBatis是一个优秀的持久层框架，它对jdbc的操作数据库的过程进行封装，使开发者只需要关注 SQL 本身，而不需要花费精力去处理例如注册驱动、创建connection、创建statement、手动设置参数、结果集检索等jdbc繁杂的过程代码。

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

# 使用jdbc编程问题总结

## 创建mysql数据库

先导入创建数据库的sql脚本导入到数据库中。



## 创建工程

1. 创建一个java工程。
2. 导入jar包。此时需要mysql 的数据库驱动。

## jdbc编程步骤：

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程序

**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", "root");

//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, "王保长");

//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语句在代码中硬编码，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. 使用preparedStatement向占有位符号传参数存在硬编码，因为sql语句的where条件不一定，可能多也可能少，修改sql还要修改代码，系统不易维护。
4. 对结果集解析存在硬编码（查询列名），sql变化导致解析代码变化，系统不易维护，如果能将数据库记录封装成pojo对象解析比较方便。

# 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执行器接口操作数据库，Executor接口有两个实现，一个是基本执行器、一个是缓存执行器。
4. Mapped Statement也是mybatis一个底层封装对象，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入门程序

## mybatis下载

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



mybatis-3.2.7.jar----mybatis的核心包

lib----mybatis的依赖包

mybatis-3.2.7.pdf----mybatis使用手册

## 需求

实现以下功能：

根据用户id查询一个用户信息

根据用户名称模糊查询用户信息列表

添加用户

更新用户

删除用户

## 工程搭建

### 第一步：创建java工程

使用eclipse创建java工程

### 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>

<!-- 和spring整合后 environments配置将废除-->

<environments default=*"development"*>

<environment id=*"development"*>

<!-- 使用jdbc事务管理-->

<transactionManager type=*"JDBC"* />

<!-- 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=*"root"* />

</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;// 生日

**private** String address;// 地址

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Users.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"*>

</mapper>

namespace ：命名空间，用于隔离sql语句。

### 第七步：加载映射文件

mybatis框架需要加载映射文件，将Users.xml添加在SqlMapConfig.xml，如下：

<mappers>

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

</mappers>

## 根据id查询用户信息

### 映射文件：

在user.xml中添加：

<!-- 根据id获取用户信息 -->

<select id=*"findUserById"* parameterType=*"int"* resultType=*"cn.qf.mybatis.po.User"*>

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

</select>

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。

### 测试程序：

**public** **class** Mybatis\_first {

//会话工厂

**private** SqlSessionFactory sqlSessionFactory;

@Before

**public** **void** createSqlSessionFactory() **throws** IOException {

// 配置文件

String resource = "SqlMapConfig.xml";

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

// 使用SqlSessionFactoryBuilder从xml配置文件中创建SqlSessionFactory

sqlSessionFactory = **new** SqlSessionFactoryBuilder()

.build(inputStream);

}

// 根据 id查询用户信息

@Test

**public** **void** testFindUserById() {

// 数据库会话实例

SqlSession sqlSession = **null**;

**try** {

// 创建数据库会话实例sqlSession

sqlSession = sqlSessionFactory.openSession();

// 查询单个记录，根据用户id查询用户信息

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

// 输出用户信息

System.out.println(user);

} **catch** (Exception e) {

e.printStackTrace();

} **finally** {

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

sqlSession.close();

}

}

}

}

## 根据用户名查询用户信息

### 映射文件：

在user.xml中添加：

<!-- 自定义条件查询用户列表 -->

<select id=*"findUserByUsername"* parameterType=*"java.lang.String"*

resultType=*"cn.qf.mybatis.po.User"*>

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

</select>

parameterType：定义输入到sql中的映射类型，${value}表示使用参数将${value}替换，做字符串的拼接。

注意：如果是取简单数量类型的参数，括号中的值必须为value

resultType：定义结果映射类型。

### 测试程序：

// 根据用户名称模糊查询用户信息

@Test

**public** **void** testFindUserByUsername() {

// 数据库会话实例

SqlSession sqlSession = **null**;

**try** {

// 创建数据库会话实例sqlSession

sqlSession = sqlSessionFactory.openSession();

// 查询单个记录，根据用户id查询用户信息

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

System.*out*.println(list.size());

} **catch** (Exception e) {

e.printStackTrace();

} **finally** {

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

sqlSession.close();

}

}

}

## 小结

### 5.6.1 parameterType和resultType

parameterType指定输入参数的java类型，可以填写别名或Java类的全限定名。

resultType指定输出结果的java类型，可以填写别名或Java类的全限定名。

### #{}和${}

#{}表示一个占位符号，通过#{}可以实现preparedStatement向占位符中设置值，自动进行java类型和jdbc类型转换。#{}可以有效防止sql注入。 #{}可以接收简单类型值或pojo属性值。 如果parameterType传输单个简单类型值，#{}括号中可以是value或其它名称。

${}表示拼接sql串，通过${}可以将parameterType 传入的内容拼接在sql中且不进行jdbc类型转换， ${}可以接收简单类型值或pojo属性值，如果parameterType传输单个简单类型值，${}括号中只能是value。

### selectOne和selectList

selectOne查询一条记录，如果使用selectOne查询多条记录则抛出异常：

org.apache.ibatis.exceptions.TooManyResultsException: Expected one result (or null) to be returned by selectOne(), but found: 3

at org.apache.ibatis.session.defaults.DefaultSqlSession.selectOne(DefaultSqlSession.java:70)

selectList可以查询一条或多条记录。

## 添加用户

### 映射文件：

在SqlMapConfig.xml中添加：

<!-- 添加用户 -->

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

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

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

</insert>

### 测试程序：

// 添加用户信息

@Test

**public** **void** testInsert() {

// 数据库会话实例

SqlSession sqlSession = **null**;

**try** {

// 创建数据库会话实例sqlSession

sqlSession = sqlSessionFactory.openSession();

// 添加用户信息

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

user.setUsername("张小明");

user.setAddress("河南郑州");

user.setSex("1");

user.setPrice(1999.9f);

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

//提交事务

sqlSession.commit();

} **catch** (Exception e) {

e.printStackTrace();

} **finally** {

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

sqlSession.close();

}

}

}

### mysql自增主键返回

通过修改sql映射文件，可以将mysql自增主键返回：

<insert id=*"insertUser"* parameterType=*"cn.qf.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>

<!—selectKeyUUID主键（注意插入的时候加上 主键名称UID） -->

<selectKey keyProperty="address" order="BEFORE" resultType="java.lang.String">

select replace(uuid(),'-','')

</selectKey>

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

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

</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，返回auto\_increment自增列新记录id值。

### Mysql使用 uuid实现主键

需要增加通过select uuid()得到uuid值

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

<selectKey resultType="java.lang.String" order="BEFORE"

keyProperty="id">

select replace(uuid(),'-','')

</selectKey>

insert into user(id,username,birthday,sex,address)

values(#{id},#{username},#{birthday},#{sex},#{address})

</insert>

注意这里使用的order是“BEFORE”

## 删除用户

### 映射文件：

<!-- 删除用户 -->

<delete id=*"deleteUserById"* parameterType=*"int"*>

delete from user where id=#{id}

</delete>

### 测试程序：

// 根据id删除用户

@Test

**public** **void** testDelete() {

// 数据库会话实例

SqlSession sqlSession = **null**;

**try** {

// 创建数据库会话实例sqlSession

sqlSession = sqlSessionFactory.openSession();

// 删除用户

sqlSession.delete("test.deleteUserById",18);

// 提交事务

sqlSession.commit();

} **catch** (Exception e) {

e.printStackTrace();

} **finally** {

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

sqlSession.close();

}

}

}

## 修改用户

### 映射文件

<!-- 更新用户 -->

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

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

where id=#{id}

</update>

### 测试程序

// 更新用户信息

@Test

**public** **void** testUpdate() {

// 数据库会话实例

SqlSession sqlSession = **null**;

**try** {

// 创建数据库会话实例sqlSession

sqlSession = sqlSessionFactory.openSession();

// 添加用户信息

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

user.setId(16);

user.setUsername("张小明");

user.setAddress("河南郑州");

user.setSex("1");

user.setPrice(1999.9f);

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

// 提交事务

sqlSession.commit();

} **catch** (Exception e) {

e.printStackTrace();

} **finally** {

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

sqlSession.close();

}

}

}

## 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，通过statement中的parameterType定义输入参数的类型。

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，通过statement中的resultType定义输出结果的类型。

## mybatis与hibernate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。

# Dao开发方法

使用Mybatis开发Dao，通常有两个方法，即原始Dao开发方法和Mapper接口开发方法。

## 需求

将下边的功能实现Dao：

根据用户id查询一个用户信息

根据用户名称模糊查询用户信息列表

添加用户信息

## SqlSession的使用范围

SqlSession中封装了对数据库的操作，如：查询、插入、更新、删除等。

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

### SqlSessionFactoryBuilder

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

### SqlSessionFactory

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

### SqlSession

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

每个线程都应该有它自己的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或实例字段中。

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

SqlSession session = sqlSessionFactory.openSession();

try {

// do work

} finally {

session.close();

}

## 原始Dao开发方式

原始Dao开发方法需要程序员编写Dao接口和Dao实现类。

### 映射文件

<?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=*"findUserById"* parameterType=*"int"* resultType=*"cn.qf.mybatis.po.User"*>

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

</select>

<!-- 添加用户 -->

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

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

select LAST\_INSERT\_ID()

</selectKey>

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

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

</insert>

</mapper>

### Dao接口

**Public interface** UserDao {

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

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

}

**Public class** UserDaoImpl **implements** UserDao {

//注入SqlSessionFactory

**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("test.findUserById", 1);

System.out.println(user);

} **finally**{

session.close();

}

**return** user;

}

@Override

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

SqlSession sqlSession = sqlSessionFactory.openSession();

**try** {

sqlSession.insert("insertUser", user);

sqlSession.commit();

} **finally**{

session.close();

}

}

}

### Dao测试

创建一个JUnit的测试类，对UserDao进行测试。

|  |
| --- |
| **private** SqlSessionFactory sqlSessionFactory;    @Before  **public** **void** init() **throws** Exception {  SqlSessionFactoryBuilder sessionFactoryBuilder = **new** SqlSessionFactoryBuilder();  InputStream inputStream = Resources.*getResourceAsStream*("SqlMapConfig.xml");  sqlSessionFactory = sessionFactoryBuilder.build(inputStream);  }  @Test  **public** **void** testGetUserById() {  UserDao userDao = **new** UserDaoImpl(sqlSessionFactory);  User user = userDao.getUserById(22);  System.***out***.println(user);  }  } |

### 问题

原始Dao开发中存在以下问题：

* Dao方法体存在重复代码：通过SqlSessionFactory创建SqlSession，调用SqlSession的数据库操作方法
* 调用sqlSession的数据库操作方法需要指定statement的id，这里存在硬编码，不利于开发维护。

## Mapper动态代理方式

### 开发规范

Mapper接口开发方法只需要程序员编写Mapper接口（相当于Dao接口），由Mybatis框架根据接口定义创建接口的动态代理对象，代理对象的方法体同上边Dao接口实现类方法。

Mapper接口开发需要遵循以下规范：

1. Mapper.xml文件中的namespace与mapper接口的类路径相同。
2. Mapper接口方法名和Mapper.xml中定义的每个statement的id相同
3. 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
4. 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

### Mapper.xml(映射文件)

定义mapper映射文件UserMapper.xml（内容同Users.xml），需要修改namespace的值为 UserMapper接口路径。将UserMapper.xml放在classpath 下mapper目录 下。

<?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.qf.mybatis.mapper.UserMapper"*>

<!-- 根据id获取用户信息 -->

<select id=*"findUserById"* parameterType=*"int"* resultType=*"cn.qf.mybatis.po.User"*>

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

</select>

<!-- 自定义条件查询用户列表 -->

<select id=*"findUserByUsername"* parameterType=*"java.lang.String"*

resultType=*"cn.qf.mybatis.po.User"*>

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

</select>

<!-- 添加用户 -->

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

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

select LAST\_INSERT\_ID()

</selectKey>

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

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

</insert>

</mapper>

### Mapper.java(接口文件)

/\*\*

\* 用户管理mapper

\*/

**Public interface** UserMapper {

//根据用户id查询用户信息

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

//查询用户列表

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

//添加用户信息

**public** **void** insertUser(User user)**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statement的id相同
2. Mapper接口方法的输入参数类型和mapper.xml中定义的statement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statement的resultType的类型相同

### 加载UserMapper.xml文件

修改SqlMapConfig.xml文件：

<!-- 加载映射文件 -->

<mappers>

<mapper resource=*"mapper/UserMapper.xml"*/>

</mappers>

### 测试

**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** testFindUserById() **throws** Exception {

//获取session

SqlSession session = sqlSessionFactory.openSession();

//获取mapper接口的代理对象

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

//调用代理对象方法

User user = userMapper.findUserById(1);

System.out.println(user);

//关闭session

session.close();

}

@Test

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

SqlSession sqlSession = sqlSessionFactory.openSession();

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

List<User> list = userMapper.findUserByUsername("张");

System.*out*.println(list.size());

}

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

//获取session

SqlSession session = sqlSessionFactory.openSession();

//获取mapper接口的代理对象

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");

//通过mapper接口添加用户

userMapper.insertUser(user);

//提交

session.commit();

//关闭session

session.close();

}

}

### 小结

* selectOne和selectList

动态代理对象调用sqlSession.selectOne()和sqlSession.selectList()是根据mapper接口方法的返回值决定，如果返回list则调用selectList方法，如果返回单个对象则调用selectOne方法。

* namespace

mybatis官方推荐使用mapper代理方法开发mapper接口，程序员不用编写mapper接口实现类，使用mapper代理方法时，输入参数可以使用pojo包装对象或map对象，保证dao的通用性。

# SqlMapConfig.xml配置文件

## 配置内容

SqlMapConfig.xml中配置的内容和顺序如下：

**properties**（属性）

settings（全局配置参数）

**typeAliases**（类型别名）

typeHandlers（类型处理器）

objectFactory（对象工厂）

plugins（插件）

environments（环境集合属性对象）

environment（环境子属性对象）

transactionManager（事务管理）

dataSource（数据源）

**mappers**（映射器）

## 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，

jdbc.driver=com.mysql.jdbc.Driver

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

jdbc.username=root

jdbc.password=root

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=*"${jdbc.driver}"*/>

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

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

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

</dataSource>

</environment>

</environments>

注意： MyBatis 将按照下面的顺序来加载属性：

* 在 properties 元素体内定义的属性首先被读取。
* 然后会读取properties 元素中resource或 url 加载的属性，它会覆盖已读取的同名属性。

## 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 |
| map | Map |

### 自定义别名：

在SqlMapConfig.xml中配置：

<typeAliases>

<!-- 单个别名定义 -->

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

<!-- 批量别名定义，扫描整个包下的类，别名为类名（首字母大写或小写都可以） -->

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

<package name=*"其它包"*/>

</typeAliases>

## 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 class=" " />

使用mapper接口类路径

如：<mapper class="com.qf.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.qf.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。**

1. 总结：

hibernate和mybatis区别:

hibernate:它是一个标准的orm框架,比较重量级,学习成本高.

优点:高度封装,使用起来不用写sql,开发的时候,会减低开发周期.

缺点:sql语句无法优化

应用场景:oa(办公自动化系统), erp(企业的流程系统)等,还有一些政府项目,

总的来说,在用于量不大,并发量小的时候使用.

mybatis:它不是一个完全的orm框架, 它是对jdbc的轻量级封装, 学习成本低,比较简单

有点:学习成本低, sql语句可以优化, 执行效率高,速度快

缺点:编码量较大,会拖慢开发周期

应用场景: 互联网项目,比如电商,P2p等

总的来说是用户量较大,并发高的项目.

**Mybatis第二天**

**框架课程**

# 课程计划

1. 输入映射和输出映射
   1. 输入参数映射
   2. 返回值映射
2. 动态sql
   1. If
   2. Where
   3. Foreach
   4. Sql片段
3. 关联查询
   1. 一对一关联
   2. 一对多关联
4. Mybatis整合spring

# 输入映射和输出映射

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)

### 传递简单类型

#{变量名} 变量名可以随便取名

参考第一天内容。

### 传递pojo对象

Mybatis使用ognl表达式解析对象字段的值，#{}或者${}括号中的值为pojo属性名称。

### 传递pojo包装对象

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

Pojo类中包含pojo。

需求：根据用户名查询用户信息，查询条件放到QueryVo的user属性中。

#### QueryVo

|  |
| --- |
| **public** **class** QueryVo {  **private** User user;  **public** User getUser() {  **return** user;  }  **public** **void** setUser(User user) {  **this**.user = user;  }      } |

#### Map

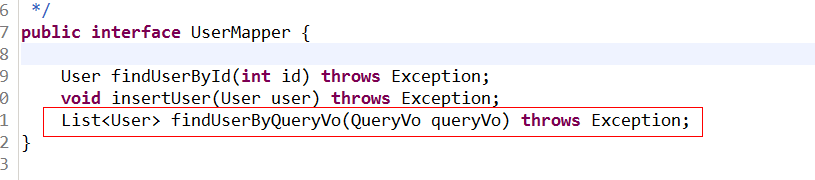
#### Sql语句

SELECT \* FROM user where username like '%张%'

#### Mapper文件

|  |
| --- |
| <!-- 使用包装类型查询用户  使用ognl从对象中取属性值，如果是包装对象可以使用.操作符来取内容部的属性  -->  <select id=*"findUserByQueryVo"* parameterType=*"queryvo"* resultType=*"user"*>  SELECT \* FROM user where username like '%${**user.username**}%'  </select> |

#### 接口



#### 测试方法

|  |
| --- |
| @Test  **public** **void** testFindUserByQueryVo() **throws** Exception {  SqlSession sqlSession = sessionFactory.openSession();  //获得mapper的代理对象  UserMapper userMapper = sqlSession.getMapper(UserMapper.**class**);  //创建QueryVo对象  QueryVo queryVo = **new** QueryVo();  //创建user对象  User user = **new** User();  user.setUsername("刘");  queryVo.setUser(user);  //根据queryvo查询用户  List<User> list = userMapper.findUserByQueryVo(queryVo);  System.*out*.println(list);  sqlSession.close();  } |

## resultType(输出类型)

### 输出简单类型

Mapper.xml文件

<!-- 获取用户列表总数 -->

<select id=*"findUserCount"* resultType=*"int"*>

select count(1) from user

</select>

Mapper接口

public int findUserCount() throws Exception;

调用：

**Public void** testFindUserCount() **throws** Exception{

//获取session

SqlSession session = sqlSessionFactory.openSession();

//获取mapper接口实例

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

//传递Hashmap对象查询用户列表

**int** count = userMapper.findUserCount();

//关闭session

session.close();

}

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对象

### 输出pojo列表

## resultMap

resultType可以指定pojo将查询结果映射为pojo，但需要pojo的属性名和sql查询的列名一致方可映射成功。

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

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

### Mapper.xml定义



使用resultMap指定上边定义的personmap。

### 定义resultMap

由于上边的mapper.xml中sql查询列和Users.java类属性不一致，需要定义resultMap：userListResultMap将sql查询列和Users.java类属性对应起来



<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User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。

### Mapper接口定义

public List<User> findUserListResultMap() throws Exception;

# 动态sql

通过mybatis提供的各种标签方法实现动态拼接sql。

## If

<!-- 传递pojo综合查询用户信息 -->

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

select \* from user

where 1=1

<if test="id!=null">

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=*"findUserList"* 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 或者or。

## foreach

向sql传递数组或List，mybatis使用foreach解析，如下：

* 需求

传入多个id查询用户信息，用下边两个sql实现：

SELECT \* FROM USERS WHERE username LIKE '%张%' AND (id =10 OR id =89 OR id=16)

SELECT \* FROM USERS WHERE username LIKE '%张%' id IN (10,89,16)

* 在pojo中定义list属性ids存储多个用户id，并添加getter/setter方法



* mapper.xml

<if test=*"ids!=null and ids.size>0"*>

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

#{id}

</foreach>

</if>

* 测试代码：

List<Integer> ids = **new** ArrayList<Integer>();

ids.add(1);//查询id为1的用户

ids.add(10); //查询id为10的用户

queryVo.setIds(ids);

List<User> list = userMapper.findUserList(queryVo);

## Sql片段

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=*"findUserList"* 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=*"findUserList"* 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”/>

## choose—when--when--otherwise

如果传入了某个参数值，那就只查询这个字段，如果没传入，就看下一个字段是否传入，如果这些字段值都没有传入，那就按默认的条件查询。这是choose--when—otherwise

例如：查询书籍信息，如果传入了图书的id号，那就只查这个字段，没传入的话，就看书名，有的话就查书名，如果id号和书名都没有，那就把douban评分大于7分的查出来.

|  |
| --- |
| <!-- 动态sql:choose when when otherwize -->    <select id="selectUserChoose" parameterType="user" resultMap="myMap" >  <!-- 通过include标签引入sql片段 -->  <include refid="select\_users"/>  where  <choose>    <when test="uid!=null">uid=#{uid}</when>  <when test="uname!=null and uname!=''">uname\_=#{uname}</when>  <otherwise>uid=10</otherwise>    </choose>      </select> |

## Set标签

|  |
| --- |
| <!-- 动态sql set标签 -->    <update id=*"updateUserSet"* parameterType=*"user"*>  <!-- 更新之前做个判断，==null或者=='' 就不更新 set会自动去掉最后一个逗号,-->  update user <!-- set upwd=#{upwd} -->    <set>  <if test=*"uname1!=null and uname1!='' "*>uname=#{uname1},</if>  <if test=*"upwd!=null and upwd!='' "*>upwd=#{upwd},</if>  <if test=*"money!=null "*>money=#{money},</if>  </set>    where uid=#{uid}  </update> |

|  |
| --- |
| //测试动态更新  @Test  **public** **void** findUpdate() {  SqlSession sqlSession = sqlSessionFactory.openSession();  UserMapper mapper = sqlSession.getMapper(UserMapper.**class**);  User user =**new** User();  user.setUid(2L);  //user.setUname1("rose");  //user.setUpwd("rose123");  user.setMoney(28800000.0);      **int** c=mapper.updateUserSet(user);  sqlSession.commit();  System.***out***.println(c);  } |

## Trim 标签

trim标记是一个格式化的标记，可以完成set或者是where标记的功能

**代替where**

|  |
| --- |
| <!-- 动态标签 trim -->    <select id="selectTrim" parameterType="user" resultMap="MyMap">  <include refid="userSql"></include>  <!-- select \* from user where uname=#{uname1} and upwd =#{upwd} -->    <!-- 都没有的话就查询所有 -->  <trim prefix="where" prefixOverrides="and|or">  <if test="uname1!=null and uname1!=''">and uname=#{uname1}</if>  <if test="upwd!=null and upwd!=''">and upwd=#{upwd}</if>  </trim>  </select> |

|  |
| --- |
| //trim select where  @Test  public void findTrim() {  SqlSession sqlSession = sqlSessionFactory.openSession();  UserMapper mapper = sqlSession.getMapper(UserMapper.class);  User user =new User();  user.setUid(2L);  //user.setUname1("rose");  //user.setUpwd("rose123");  user.setMoney(28800000.0);      List<User> selectTrim = mapper.selectTrim(user);  sqlSession.commit();  System.out.println(selectTrim);  } |

**代替 set**

|  |
| --- |
| <!-- trim实现动态更新 -->  <update id=*"updateTrim"* parameterType=*"user"*>  update user  <trim prefix=*"set"* suffixOverrides=*","* suffix=*"where uid= #{uid}"*>  <if test=*"uname1!=null and uname1!=''"*>uname=#{uname1},</if>  <if test=*"upwd!=null and upwd!=''"*>upwd=#{upwd},</if>  <if test=*"money!=null and money!=''"*>money=#{money},</if>  </trim>  </update> |

|  |
| --- |
| //update updateTrim    @Test  **public** **void** updateTrim() {  SqlSession sqlSession = sqlSessionFactory.openSession();  UserMapper mapper = sqlSession.getMapper(UserMapper.**class**);  User user =**new** User();  user.setUid(2L);  user.setUname1("rose123");  user.setUpwd("rose123");  user.setMoney(88800000.0);  **int** c= mapper.updateTrim(user);  sqlSession.commit();  System.***out***.println(c);  } |

# 关联查询

## 商品订单数据模型

用户表：user

记录了购买商品的用户信息

Id：唯一标识一个用户

订单表：orders

记录了用户创建的订单

创建用户:user\_id(外键)

订单号

创建时间

订单状态

一对一：一个订单只能由一个用户创建

一对多：一个用户可以创建多个订单

## 一对一查询

案例：查询所有订单信息，关联查询下单用户信息。

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,

userss.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** OrdersCustom **extends** Orders {

**private** String username;// 用户名称

**private** String address;// 用户地址

get/set。。。。

OrdersCustom类继承Orders类后OrdersCustom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.qf.mybatis.po.OrdersCustom"*>

SELECT

orders.\*,

user.username,

user.address

FROM

orders, user

WHERE orders.user\_id = user.id

</select>

#### Mapper接口：

public List<*OrdersCustom*> 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<*OrdersCustom*> list = userMapper.findOrdersList();

System.*out*.println(list);

//关闭session

session.close();

}

#### 小结：

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

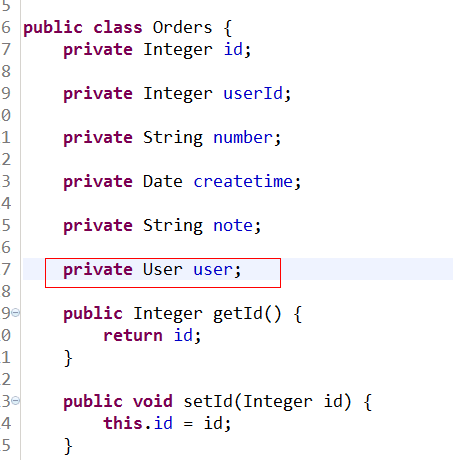
orders,

user

WHERE orders.user\_id = user.id

#### 定义po类

在Orders类中加入User属性，user属性中用于存储关联查询的用户信息，因为订单关联查询用户是一对一关系，所以这里使用单个User对象存储关联查询的用户信息。



#### Mapper.xml

|  |
| --- |
| <!-- 查询订单关联用户信息使用resultmap -->  <resultMap type=*"cn.qf.po.Orders"* id=*"orderUserResultMap"*>  <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"*/>  <!-- 一对一关联映射 -->  <!--  property:Orders对象的user属性  javaType：user属性对应 的类型  -->  <**association** property=*"user"* javaType=*"cn.qf.po.User"*>  <!-- column:user表的主键对应的列 property：user对象中id属性-->  <id column=*"user\_id"* property=*"id"*/>  <result column=*"username"* property=*"username"*/>  <result column=*"address"* property=*"address"*/>  </association>  </resultMap>  <select id=*"findOrdersWithUserResultMap"* resultMap=*"orderUserResultMap"*>  SELECT  o.id,  o.user\_id,  o.number,  o.createtime,  o.note,  u.username,  u.address  FROM  orders o  JOIN `user` u ON u.id = o.user\_id  </select> |

这里resultMap指定orderUserResultMap*。*

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.qf.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> findOrdersListResultMap() throws Exception;

#### 测试：

**Public void** testfindOrdersListResultMap()**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();

}

#### 小结：

使用association完成关联查询，将关联查询信息映射到pojo对象中。

## 一对多查询

案例：查询所有用户信息及用户关联的订单信息。

用户信息和订单信息为一对多关系。

使用resultMap实现如下：

### Sql语句：

SELECT

u.\*, o.id oid,

o.number,

o.createtime,

o.note

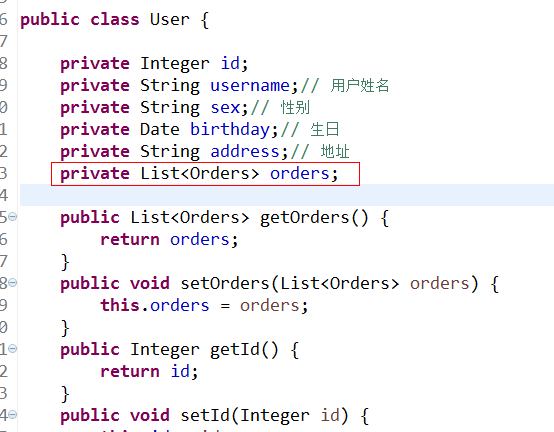
FROM

`user` u

LEFT JOIN orders o ON u.id = o.user\_id

### 定义po类

在User类中加入List<Orders> orders属性



### Mapper.xml

|  |
| --- |
| <resultMap type=*"cn.qf.po.user"* id=*"userOrderResultMap"*>  <!-- 用户信息映射 -->  <id property=*"id"* column=*"id"*/>  <result property=*"username"* column=*"username"*/>  <result property=*"birthday"* column=*"birthday"*/>  <result property=*"sex"* column=*"sex"*/>  <result property=*"address"* column=*"address"*/>  <!-- 一对多关联映射 -->  <collection property=*"orders"* ofType=*"cn.qf.po.Orders"*>  <id property=*"id"* column=*"oid"*/>  <!--用户id已经在user对象中存在，此处可以不设置-->  <!-- <result property="userId" column="id"/> -->  <result property=*"number"* column=*"number"*/>  <result property=*"createtime"* column=*"createtime"*/>  <result property=*"note"* column=*"note"*/>  </collection>  </resultMap>  <select id=*"getUserOrderList"* resultMap=*"userOrderResultMap"*>  SELECT  u.\*, o.id oid,  o.number,  o.createtime,  o.note  FROM  `user` u  LEFT JOIN orders o ON u.id = o.user\_id  </select> |

collection部分定义了用户关联的订单信息。表示关联查询结果集

property=*"orders"：*关联查询的结果集存储在User对象的上哪个属性。

ofType=*"orders"：*指定关联查询的结果集中的对象类型即List中的对象类型。此处可以使用别名，也可以使用全限定名。

<id />及<result/>的意义同一对一查询。

### Mapper接口：

List<User> getUserOrderList();

### 测试

|  |
| --- |
| @Test  **public** **void** getUserOrderList() {  SqlSession session = sqlSessionFactory.openSession();  UserMapper userMapper = session.getMapper(UserMapper.**class**);  List<User> result = userMapper.getUserOrderList();  **for** (User user : result) {  System.***out***.println(user);  }  session.close();  } |

# Mybatis整合spring

## 整合思路

1. SqlSessionFactory对象应该放到spring容器中作为单例存在。
2. 传统dao的开发方式中，应该从spring容器中获得sqlsession对象。
3. Mapper代理形式中，应该从spring容器中直接获得mapper的代理对象。
4. 数据库的连接以及数据库连接池事务管理都交给spring容器来完成。

## 整合需要的jar包

1. spring的jar包
2. Mybatis的jar包
3. Spring+mybatis的整合包。
4. Mysql的数据库驱动jar包。
5. 数据库连接池的jar包。

## 整合的步骤

第一步：创建一个java工程。

第二步：导入jar包。（上面提到的jar包）

第三步：mybatis的配置文件sqlmapConfig.xml

第四步：编写Spring的配置文件

1、数据库连接及连接池

2、事务管理（暂时可以不配置）

3、sqlsessionFactory对象，配置到spring容器中

4、mapeer代理对象或者是dao实现类配置到spring容器中。

第五步：编写dao或者mapper文件

第六步：测试。

### 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>  <typeAliases>  <package name=*"cn.qf.mybatis.pojo"*/>  </typeAliases>  <mappers>  <mapper resource=*"sqlmap/User.xml"*/>  </mappers>  </configuration> |

### applicationContext.xml

|  |
| --- |
| <?xml version=*"1.0"* encoding=*"UTF-8"*?>  <beans xmlns=*"http://www.springframework.org/schema/beans"*  xmlns:context=*"http://www.springframework.org/schema/context"* xmlns:p=*"http://www.springframework.org/schema/p"*  xmlns:aop=*"http://www.springframework.org/schema/aop"* xmlns:tx=*"http://www.springframework.org/schema/tx"*  xmlns:xsi=*"http://www.w3.org/2001/XMLSchema-instance"*  xsi:schemaLocation=*"http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans-4.0.xsd*  *http://www.springframework.org/schema/context http://www.springframework.org/schema/context/spring-context-4.0.xsd*  *http://www.springframework.org/schema/aop http://www.springframework.org/schema/aop/spring-aop-4.0.xsd http://www.springframework.org/schema/tx http://www.springframework.org/schema/tx/spring-tx-4.0.xsd*  *http://www.springframework.org/schema/util http://www.springframework.org/schema/util/spring-util-4.0.xsd"*>  <!-- 加载配置文件 -->  <context:property-placeholder location=*"classpath:db.properties"* />  <!-- 数据库连接池 -->  <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}"* />  <property name=*"maxActive"* value=*"10"* />  <property name=*"maxIdle"* value=*"5"* />  </bean>  <!-- mapper配置 -->  <!-- 让spring管理sqlsessionfactory 使用mybatis和spring整合包中的 -->  <bean id=*"sqlSessionFactory"* class=*"org.mybatis.spring.SqlSessionFactoryBean"*>  <!-- 数据库连接池 -->  <property name=*"dataSource"* ref=*"dataSource"* />  <!-- 加载mybatis的全局配置文件 -->  <property name=*"configLocation"* value=*"classpath:mybatis/SqlMapConfig.xml"* />  </bean>  </beans> |

### db.properties

|  |
| --- |
| jdbc.driver=com.mysql.jdbc.Driver  jdbc.url=jdbc:mysql://localhost:3306/mybatis?characterEncoding=utf-8  jdbc.username=root  jdbc.password=root |

## Dao的开发

三种dao的实现方式：

1、传统dao的开发方式

2、使用mapper代理形式开发方式

3、使用扫描包配置mapper代理。

### 传统dao的开发方式

接口+实现类来完成。需要dao实现类需要继承SqlsessionDaoSupport类

#### Dao实现类

|  |
| --- |
| **public** **class** UserDaoImpl **extends** SqlSessionDaoSupport **implements** UserDao {  @Override  **public** User findUserById(**int** id) **throws** Exception {  SqlSession session = getSqlSession();  User user = session.selectOne("test.findUserById", id);  //不能关闭SqlSession，让spring容器来完成  //session.close();  **return** user;  }  @Override  **public** **void** insertUser(User user) **throws** Exception {  SqlSession session = getSqlSession();  session.insert("test.insertUser", user);  session.commit();  //session.close();  }  } |

#### 配置dao

把dao实现类配置到spring容器中

|  |
| --- |
| <!-- 配置UserDao实现类 -->  <bean id=*"userDao"* class=*"cn.qf.dao.UserDaoImpl"*>  <property name=*"sqlSessionFactory"* ref=*"sqlSessionFactory"*/>  </bean> |

#### 测试方法

初始化:

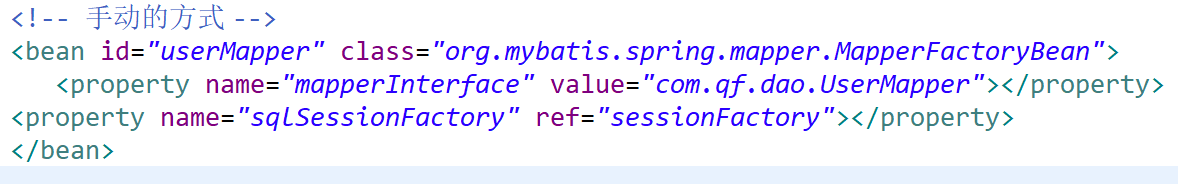
|  |
| --- |
| **private** ApplicationContext applicationContext;  @Before  **public** **void** setUp() **throws** Exception{  String configLocation = "classpath:spring/ApplicationContext.xml";  //初始化spring运行环境  applicationContext = **new** ClassPathXmlApplicationContext(configLocation);  } |

测试:

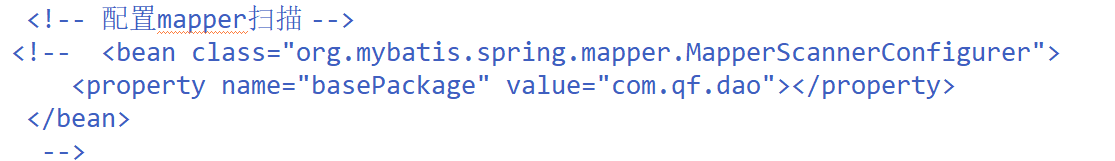
|  |
| --- |
| @Test  **public** **void** testFindUserById() **throws** Exception {  UserDao userDao = (UserDao) applicationContext.getBean("userDao");  User user = userDao.findUserById(1);  System.*out*.println(user);  } |

### Mapper代理形式开发dao

**Mapper代理对象：org.mybatis.spring.mapper.MapperFactoryBean 手动给每个mapper接口进行代理 生成代理类**



**Mapper扫描：org.mybatis.spring.mapper.MapperScannerConfigurer （推荐使用）**



#### 开发mapper接口

开发mapper文件

#### 配置mapper代理

|  |
| --- |
| <!-- 配置mapper代理对象 -->  <bean class=*"org.mybatis.spring.mapper.MapperFactoryBean"*>  <property name=*"mapperInterface"* value=*"cn.qf.mybatis.mapper.UserMapper"*/>  <property name=*"sqlSessionFactory"* ref=*"sqlSessionFactory"*></property>  </bean> |

#### 测试方法

|  |
| --- |
| **public** **class** UserMapperTest {  **private** ApplicationContext applicationContext;  @Before  **public** **void** setUp() **throws** Exception {  applicationContext = **new** ClassPathXmlApplicationContext("classpath:spring/applicationContext.xml");  }  @Test  **public** **void** testGetUserById() {  UserMapper userMapper = applicationContext.getBean(UserMapper.**class**);  User user = userMapper.getUserById(1);  System.***out***.println(user);  }  } |

### 扫描包形式配置mapper

|  |
| --- |
| <!-- 使用扫描包的形式来创建mapper代理对象 -->  <bean class=*"org.mybatis.spring.mapper.MapperScannerConfigurer"*>  <property name=*"basePackage"* value=*"cn.qf.mybatis.mapper"*></property>  </bean> |

每个mapper代理对象的id就是类名，首字母小写

并且不需要在mybatis.xml配置文件中配置<mapper>标签了

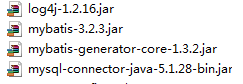
# Mybatis逆向工程

# 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。

作用:mybatis官方提供逆向工程,可以使用它通过数据库中的表来自动生成Mapper接口和映射文件(单表增删改查)和Po类.

导入的jar包有:



## 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：

|  |
| --- |
| <?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://localhost:3306/mybatis"* userId=*"root"*  password=*"admin"*>  </jdbcConnection>  <!-- <jdbcConnection driverClass="oracle.jdbc.OracleDriver"  connectionURL="jdbc:oracle:thin:@127.0.0.1:1521:yycg"  userId="yycg"  password="yycg">  </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.qf.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.qf.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.qf.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>  <!-- <table schema="" tableName="sys\_user"></table>  <table schema="" tableName="sys\_role"></table>  <table schema="" tableName="sys\_permission"></table>  <table schema="" tableName="sys\_user\_role"></table>  <table schema="" tableName="sys\_role\_permission"></table> -->    <!-- 有些表的字段需要指定java类型  <table schema="" tableName="">  <columnOverride column="" javaType="" />  </table> -->  </context>  </generatorConfiguration> |

## 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

UserselectByPrimaryKey(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);

## 逆向工程注意事项

### 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：

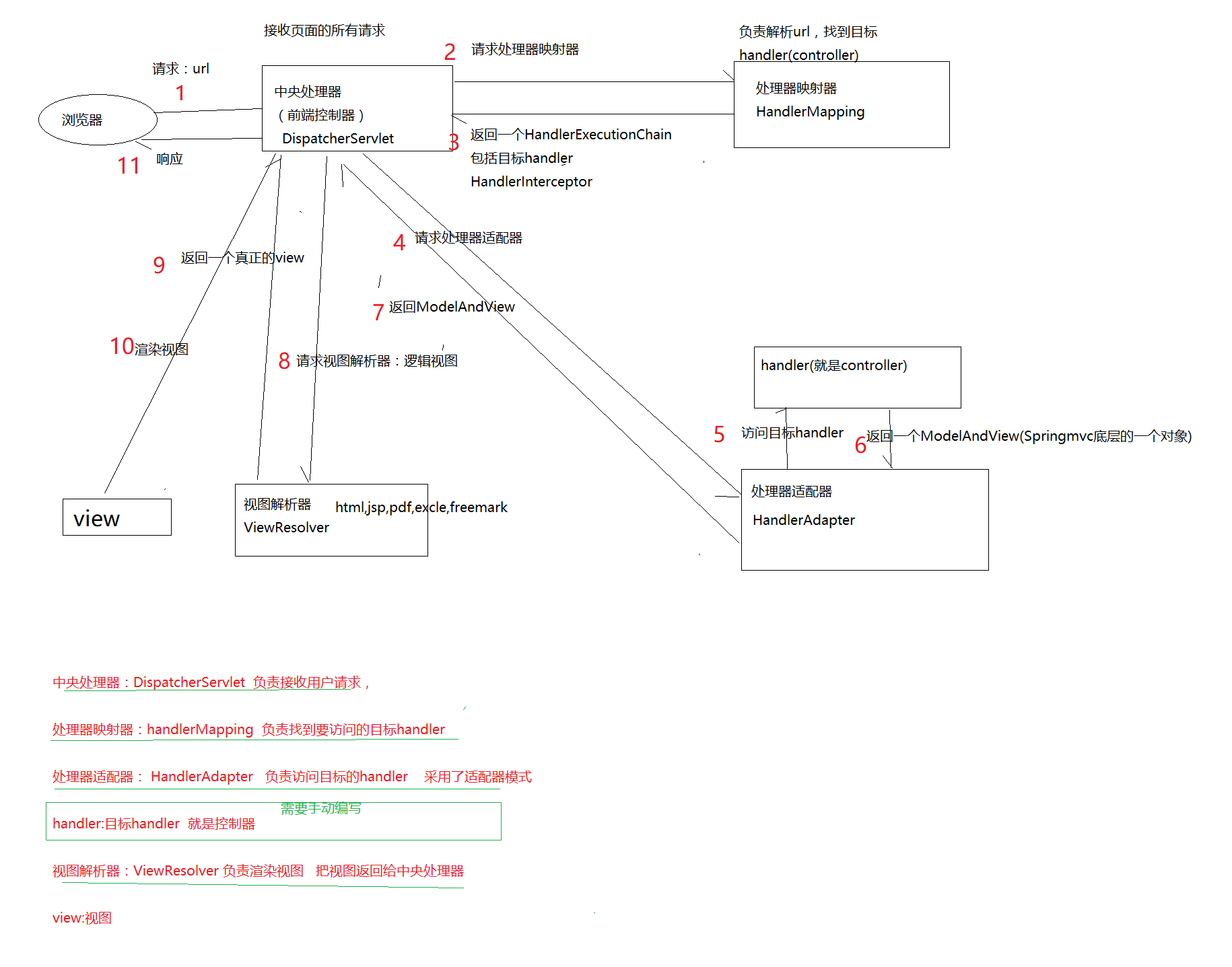
<table schema="XXXX" tableName=" " >

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



**Spring web mvc**

**框架课程**

# SpringMVC架构

## Spring web mvc介绍

Spring web mvc和Struts2都属于表现层的框架,它是Spring框架的一部分,我们可以从Spring的整体结构中看得出来：



## Web MVC

mvc设计模式在b/s系统下应用：

1发起请求

C 控制器controller

作用：接收请求，响应处理结果，没有业务逻辑

5 响应结果

M 模型

pojo

service

dao

2请求业务处理

V 视图 View

将模型数据展示给用户

4 向view中填充模型数据

3返回模型

注意：b/s下模型无法将数据设置到视图中

1. 用户发起request请求至控制器(Controller)

控制接收用户请求的数据，委托给模型进行处理

1. 控制器通过模型(Model)处理数据并得到处理结果

模型通常是指业务逻辑

1. 模型处理结果返回给控制器
2. 控制器将模型数据在视图(View)中展示

web中模型无法将数据直接在视图上显示，需要通过控制器完成。如果在C/S应用中模型是可以将数据在视图中展示的。

1. 控制器将视图response响应给用户

通过视图展示给用户要的数据或处理结果。

## Spring web mvc 架构

### 架构图



### 架构流程

1. 用户发送请求至前端控制器DispatcherServlet
2. DispatcherServlet收到请求调用HandlerMapping处理器映射器。
3. 处理器映射器根据请求url找到具体的处理器，生成处理器对象及处理器拦截器(如果有则生成)一并返回给DispatcherServlet。
4. DispatcherServlet通过HandlerAdapter处理器适配器调用处理器
5. 执行处理器(Controller，也叫后端控制器)。
6. Controller执行完成返回ModelAndView
7. HandlerAdapter将controller执行结果ModelAndView返回给DispatcherServlet
8. DispatcherServlet将ModelAndView传给ViewReslover视图解析器
9. ViewReslover解析后返回具体View
10. DispatcherServlet对View进行渲染视图（即将模型数据填充至视图中）。
11. DispatcherServlet响应用户

### 组件说明

以下组件通常使用框架提供实现：

* DispatcherServlet：前端控制器

用户请求到达前端控制器，它就相当于mvc模式中的c，dispatcherServlet是整个流程控制的中心，由它调用其它组件处理用户的请求，dispatcherServlet的存在降低了组件之间的耦合性。

* HandlerMapping：处理器映射器

HandlerMapping负责根据用户请求找到Handler即处理器，springmvc提供了不同的映射器实现不同的映射方式，例如：配置文件方式，实现接口方式，注解方式等。

* HandlAdapter：处理器适配器

通过HandlerAdapter对处理器进行执行，这是适配器模式的应用，通过扩展适配器可以对更多类型的处理器进行执行。

* Handler：处理器

Handler 是继DispatcherServlet前端控制器的后端控制器，在DispatcherServlet的控制下Handler对具体的用户请求进行处理。

由于Handler涉及到具体的用户业务请求，所以一般情况需要程序员根据业务需求开发Handler。

* View Resolver：视图解析器

View Resolver负责将处理结果生成View视图，View Resolver首先根据逻辑视图名解析成物理视图名即具体的页面地址，再生成View视图对象，最后对View进行渲染将处理结果通过页面展示给用户。 springmvc框架提供了很多的View视图类型，包括：jstlView、freemarkerView、pdfView等。

一般情况下需要通过页面标签或页面模版技术将模型数据通过页面展示给用户，需要由程序员根据业务需求开发具体的页面。

# 商品订单业务说明

本教程在通过商品订单业务学习使用springmvc进行功能开发。

## 业务流程

1、管理员维护商品信息

2、用户挑选商品，购买，创建订单

## 数据库环境

## 商品订单数据模型

用户表：users

记录了购买商品的用户信息

Id：唯一标识一个用户

订单表：orders

记录了用户创建的订单

创建用户:users\_id(外键)

订单号

创建时间

订单状态

外键：

users\_id

订单明细表：orderdetail

记录了用户购买信息

所属订单：orders\_id（外键）

商品id:items\_id（外键）

商品数量

商品购买价格

外键：orders\_id

商品信息：items

记录了所有商品信息

商品id：id（主键）

商品名称：

商品介绍

商品价格

外键：

items\_id

一对多

用户可以创建多个订单

一对一：

一个订单只能由一个用户创建

一对多

一个订单包括多个购买明细

一对一

一个订单明细只能

属于一个订单

一对一

一个明细对应一个商品

一对多

一个商品对应多个订单明细

通过订单明细表订单表和商品表建立 关系

一个订单对应多个商品

一个商品对应多个订单

订单表和商品表是多对多关系

# SpringMVC入门

## 需求

实现商品查询列表功能。

## 第一步:建立一个Web项目

在eclipse下创建动态web工程springmvc\_first。

## 第二步:导入spring的jar包

## 第三步：前端控制器配置

在WEB-INF\web.xml中配置前端控制器，

<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:springmvc.xml</param-value>

</init-param>

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

</servlet>

<servlet-mapping>

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

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

</servlet-mapping>

load-on-startup：表示servlet随服务启动；

url-pattern：\*.action的请交给DispatcherServlet处理。

contextConfigLocation：指定springmvc配置的加载位置，如果不指定则默认加

载WEB-INF/[DispatcherServlet 的Servlet 名字]-servlet.xml。

### Servlet拦截方式

详情看http://blog.sina.com.cn/s/blog\_54829a240102vj0z.html

1、拦截固定后缀的url，比如设置为 \*.do、\*.action， 例如：/user/add.action

此方法最简单，不会导致静态资源（jpg,js,css）被拦截。

2、拦截所有，设置为/，例如：/user/add /user/add.action

此方法可以实现REST风格的url，很多互联网类型的应用使用这种方式。不拦截jsp

但是此方法会导致静态文件（jpg,js,css）被拦截后不能正常显示。需要特殊处理。

注解中使用 <mvc:default-servlet-handler/>

3、拦截所有，设置为/\*，此设置方法错误，因为请求到Action，当action转到jsp时再次被拦截，提示不能根据jsp路径mapping成功。

## 第四步：springmvc配置文件

Springmvc默认加载WEB-INF/[前端控制器的名字]-servlet.xml，也可以在前端控制器定义处指定加载的配置文件，如下：

<init-param>

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

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

</init-param>

如上代码，通过contextConfigLocation加载classpath下的springmvc.xml配置文件。

## 第五步：配置处理器适配器

在springmvc.xml文件配置如下：

<bean class=*"**org.springframework.web.servlet.mvc.SimpleControllerHandlerAdapter"/*>

SimpleControllerHandlerAdapter：即简单控制器处理适配器，所有实现了org.springframework.web.servlet.mvc.Controller 接口的Bean作为

Springmvc的后端控制器。

## 第六步：处理器开发

**public** **class** ItemList1 **implements** Controller {

@Override

**public** ModelAndView handleRequest(HttpServletRequest request,

HttpServletResponse response) **throws** Exception {

//商品列表

List<Items> itemsList = **new** ArrayList<Items>();

Items items\_1 = **new** Items();

items\_1.setName("联想笔记本");

items\_1.setPrice(6000f);

items\_1.setDetail("ThinkPad T430 联想笔记本电脑！");

Items items\_2 = **new** Items();

items\_2.setName("苹果手机");

items\_2.setPrice(5000f);

items\_2.setDetail("iphone6苹果手机！");

itemsList.add(items\_1);

itemsList.add(items\_2);

//创建modelAndView准备填充数据、设置视图

ModelAndView modelAndView = **new** ModelAndView();

//填充数据

modelAndView.addObject("itemsList", itemsList);

//视图

modelAndView.setViewName("*/WEB-INF/jsp/*order/itemsList.jsp");

**return** modelAndView;

}

}

org.springframework.web.servlet.mvc.Controller：处理器必须实现Controller 接口。

ModelAndView：包含了模型数据及逻辑视图名

## 第七步：配置处理器映射器

<!-- 处理器映射器 -->

<!-- 根据bean的name进行查找Handler 将action的url配置在bean的name中 -->

<bean

class=*"org.springframework.web.servlet.handler.BeanNameUrlHandlerMapping"* />

BeanNameUrlHandlerMapping：表示将定义的Bean名字作为请求的url，需要将编写的controller在spring容器中进行配置，且指定bean的name为请求的url，且必须以.action结尾。

在springmvc.xml文件配置如下：

<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 "*>

<!-- 处理器映射器 -->

<!-- 根据bean的name进行查找Handler 将action的url配置在bean的name中 -->

<bean

class=*"**org.springframework.web.servlet.handler.BeanNameUrlHandlerMapping"* />

## 第八步：处理器配置

在springmvc.xml文件配置如下：

//id名为 处理器类名首字母小写

<!-- controller配置 -->

<bean name=*"/items1.action"* id=*"itemList1"*

class=*"com.qf.springmvc.controller.first.**ItemList1"*/>

name="/items1.action"：前边配置的处理器映射器为BeanNameUrlHandlerMapping，如果请求的URL 为“上下文/items1.action”将会成功映射到ItemList1控制器。

## 第九步：配置视图解析器

在springmvc.xml文件配置如下：

<!-- ViewResolver -->

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

//使用注解时使用

<property name=*"viewClass"* value=*"org.springframework.web.servlet.view.JstlView"*/>

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

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

</bean>

InternalResourceViewResolver：支持JSP视图解析

viewClass：JstlView表示JSP模板页面需要使用JSTL标签库，所以classpath中必须包含jstl的相关jar 包；

prefix 和suffix：查找视图页面的前缀和后缀，最终视图的址为：

前缀+**逻辑视图名**+后缀，逻辑视图名需要在controller中返回ModelAndView指定，比如逻辑视图名为hello，则最终返回的jsp视图地址 “WEB-INF/jsp/hello.jsp”

## 第十步：视图开发

创建/WEB-INF/jsp/order/itemsList.jsp视图页面：

<%@ page language=*"java"* contentType=*"text/html; charset=UTF-8"*

pageEncoding=*"UTF-8"*%>

<%@ taglib uri=*"http://java.sun.com/jsp/jstl/core"* prefix=*"c"* %>

<%@ taglib uri=*"http://java.sun.com/jsp/jstl/fmt"* prefix=*"fmt"*%>

<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN" "http://www.w3.org/TR/html4/loose.dtd">

<html>

<head>

<meta http-equiv=*"Content-Type"* content=*"text/html; charset=UTF-8"*>

<title>查询商品列表</title>

</head>

<body>

商品列表：

<table width=*"100%"* border=*1*>

<tr>

<td>商品名称</td>

<td>商品价格</td>

<td>商品描述</td>

</tr>

<c:forEach items=*"*${itemsList }*"* var=*"item"*>

<tr>

<td>${item.name }</td>

<td>${item.price }</td>

<td>${item.detail }</td>

</tr>

</c:forEach>

</table>

</body>

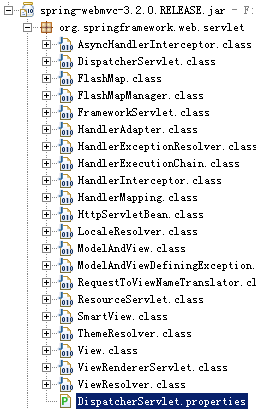
</html>

## 第十一步：部署在tomcat测试

通过请求：http://localhost:8080/springmvc\_first/items1.action，如果页面输出商品列表就表明我们成功了！

## DispatcherServlet

DispathcerServlet作为springmvc的中央调度器存在，DispatcherServlet创建时会默认从DispatcherServlet.properties文件加载springmvc所用的各各组件，如果在springmvc.xml中配置了组件则以springmvc.xml中配置的为准，DispatcherServlet的存在降低了springmvc各各组件之间的耦合度。



## HandlerMapping处理器映射器

HandlerMapping 负责根据request请求找到对应的Handler处理器及Interceptor拦截器，将它们封装在HandlerExecutionChain 对象中给前端控制器返回。

### BeanNameUrlHandlerMapping

BeanNameUrl处理器映射器，根据请求的url与spring容器中定义的bean的name进行匹配，从而从spring容器中找到bean实例。

<!—beanName Url映射器 -->

<bean class="org.springframework.web.servlet.handler.BeanNameUrlHandlerMapping"/>

### SimpleUrlHandlerMapping

simpleUrlHandlerMapping是BeanNameUrlHandlerMapping的增强版本，它可以将url和处理器bean的id进行统一映射配置。

<!—简单url映射 -->

<bean class=*"org.springframework.web.servlet.handler.SimpleUrlHandlerMapping"*>

<property name=*"mappings"*>

<props>

<prop key=*"/items1.action"*>controller的bean id</prop>

<prop key=*"/items2.action"*>controller的bean id</prop>

</props>

</property>

</bean>

## HandlerAdapter处理器适配器

HandlerAdapter会根据适配器接口对后端控制器进行包装（适配），包装后即可对处理器进行执行，通过扩展处理器适配器可以执行多种类型的处理器，这里使用了适配器设计模式。

### SimpleControllerHandlerAdapter

SimpleControllerHandlerAdapter简单控制器处理器适配器，所有实现了org.springframework.web.servlet.mvc.Controller 接口的Bean通过此适配器进行适配、执行。

适配器配置如下：

<bean class="org.springframework.web.servlet.mvc.SimpleControllerHandlerAdapter" />

### HttpRequestHandlerAdapter

HttpRequestHandlerAdapter，http请求处理器适配器，所有实现了org.springframework.web.HttpRequestHandler 接口的Bean通过此适配器进行适配、执行。

适配器配置如下：

<bean class="org.springframework.web.servlet.mvc.HttpRequestHandlerAdapter"/>

Controller实现如下：

**public** **class** ItemList2 **implements** HttpRequestHandler {

@Override

**public** **void** handleRequest(HttpServletRequest request,

HttpServletResponse response) **throws** ServletException, IOException {

// 商品列表

List<Items> itemsList = **new** ArrayList<Items>();

Items items\_1 = **new** Items();

items\_1.setName("联想笔记本");

items\_1.setPrice(6000f);

items\_1.setDetail("ThinkPad T430 联想笔记本电脑！");

Items items\_2 = **new** Items();

items\_2.setName("苹果手机");

items\_2.setPrice(5000f);

items\_2.setDetail("iphone5 苹果手机！");

itemsList.add(items\_1);

itemsList.add(items\_2);

// 填充数据

request.setAttribute("itemsList", itemsList);

// 视图

request.getRequestDispatcher("/WEB-INF/jsp/order/itemsList.jsp").forward(request, response);

}

}

从上边可以看出此适配器器的handleRequest方法没有返回ModelAndView，可通过response修改定义响应内容，比如返回json数据：

response.setCharacterEncoding("utf-8");

response.setContentType("application/json;charset=utf-8");

response.getWriter().write("json串");

## 注解映射器和适配器

### Controller的代码

@Controller

**public** **class** ItemList3 {

@RequestMapping("/queryItem.action")

**public** ModelAndView queryItem() {

// 商品列表

List<Items> itemsList = **new** ArrayList<Items>();

Items items\_1 = **new** Items();

items\_1.setName("联想笔记本");

items\_1.setPrice(6000f);

items\_1.setDetail("ThinkPad T430 联想笔记本电脑！");

Items items\_2 = **new** Items();

items\_2.setName("苹果手机");

items\_2.setPrice(5000f);

items\_2.setDetail("iphone6苹果手机！");

itemsList.add(items\_1);

itemsList.add(items\_2);

// 创建modelAndView准备填充数据、设置视图

ModelAndView modelAndView = **new** ModelAndView();

// 填充数据

modelAndView.addObject("itemsList", itemsList);

// 视图

modelAndView.setViewName("order/itemsList");

**return** modelAndView;

}

}

### 组件扫描器

使用组件扫描器省去在spring容器配置每个controller类的繁琐。使用<context:component-scan自动扫描标记@controller的控制器类，配置如下：

<!-- 扫描controller注解,多个包中间使用半角逗号分隔 -->

<context:component-scan base-package=*"com.qf.springmvc.controller.first"*/>

### RequestMappingHandlerMapping

注解式处理器映射器，对类中标记@ResquestMapping的方法进行映射，根据ResquestMapping定义的url匹配ResquestMapping标记的方法，匹配成功返回HandlerMethod对象给前端控制器，HandlerMethod对象中封装url对应的方法Method。

从spring3.1版本开始，废除了DefaultAnnotationHandlerMapping的使用，推荐使用RequestMappingHandlerMapping完成注解式处理器映射。

配置如下：

<!--注解映射器 -->

<bean class=*"org.springframework.web.servlet.mvc.method.annotation.RequestMappingHandlerMapping"*/>

注解描述：

@RequestMapping：定义请求url到处理器功能方法的映射

### RequestMappingHandlerAdapter

注解式处理器适配器，对标记@ResquestMapping的方法进行适配。

从spring3.1版本开始，废除了AnnotationMethodHandlerAdapter的使用，推荐使用RequestMappingHandlerAdapter完成注解式处理器适配。

配置如下：

<!--注解适配器 -->

<bean class=*"org.springframework.web.servlet.mvc.method.annotation.RequestMappingHandlerAdapter"*/>

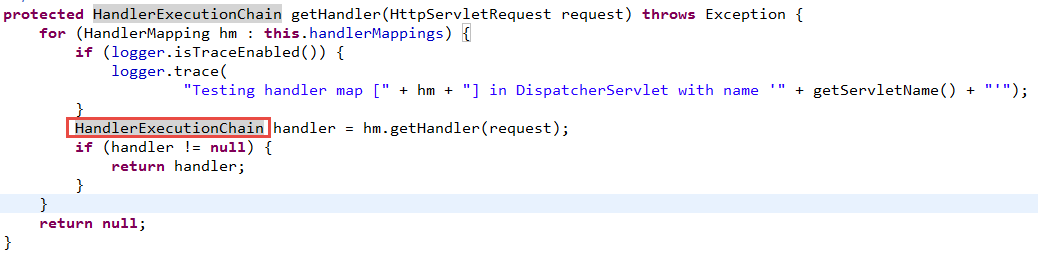
### <mvc:annotation-driven>

springmvc使用<mvc:annotation-driven>自动加载RequestMappingHandlerMapping和RequestMappingHandlerAdapter，可用在springmvc.xml配置文件中使用<mvc:annotation-driven>替代注解处理器和适配器的配置。

## springmvc处理流程源码分析

1. 用户发送请求到DispatherServlet前端控制器
2. DispatherServlet调用HandlerMapping（处理器映射器）根据url查找Handler





1. DispatherServlet调用HandlerAdapter(处理器适配器)对HandlerMapping找到Handler进行包装、执行。HandlerAdapter执行Handler完成后，返回了一个ModleAndView(springmvc封装对象)

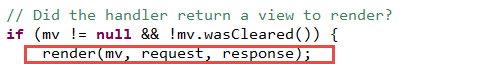
DispatherServlet 找一个合适的适配器：



适配器执行Hanlder



1. DispatherServlet拿着ModelAndView调用ViewResolver（视图解析器）进行视图解析，解析完成后返回一个View（很多不同视图类型的View）



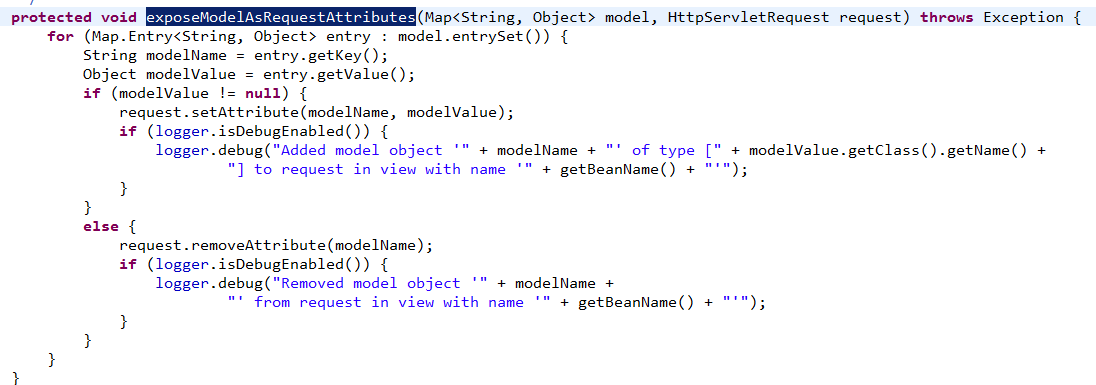
视图解析：



1. DispatcherServlet进行视图渲染，将Model中数据放到request域，在页面展示



将model数据放在request域：



# 整合mybatis

为了更好的学习 springmvc和mybatis整合开发的方法，需要将springmvc和mybatis进行整合。

整合目标：控制层采用springmvc、持久层使用mybatis实现。

## 需求

实现商品查询列表，从mysql数据库查询商品信息。

## jar包

包括：spring（包括springmvc）、mybatis、mybatis-spring整合包、数据库驱动、第三方连接池。

参考：“mybatis与springmvc整合全部jar包”目录

## Dao

目标：

1、spring管理SqlSessionFactory、mapper

### db.properties

jdbc.driver=com.mysql.jdbc.Driver

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

jdbc.username=XXXX

jdbc.password=XXXX

### log4j.properties

# Global logging configuration，建议开发环境中要用debug

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

### sqlMapConfig.xml

在classpath下创建*mybatis/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="com.qf.ssm.mapper" />

</mappers>

</configuration>

### applicationContext-dao.xml

配置数据源、事务管理，配置SqlSessionFactory、mapper扫描器。

<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 "*>

<!-- 加载配置文件 -->

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

<!-- 数据库连接池 -->

<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}"*/>

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

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

</bean>

<!-- 让spring管理sqlsessionfactory 使用mybatis和spring整合包中的 -->

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

<!-- 数据库连接池 -->

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

<!-- 加载mybatis的全局配置文件 -->

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

</bean>

<!-- mapper扫描器 -->

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

<property name=*"basePackage"* value=*"com.qf.springmvc.mapper"*></property>

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

</bean>

</beans>

### ItemsMapper.xml

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

<!DOCTYPE mapper

PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"

"http://mybatis.org/dtd/mybatis-3-mapper.dtd">

<mapper namespace=*"com.qf.ssm.mapper.ItemsMapper"*>

<!-- sql片段 -->

<!-- 商品查询条件 -->

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

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

<if test=*"items.name!=null and items.name!=''"*>

and items.name like '%${items.name}%'

</if>

</if>

</sql>

<!-- 查询商品信息 -->

<select id=*"findItemsList"* parameterType=*"queryVo"* resultType=*"items"*>

select \* from items

<where>

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

</where>

</select>

</mapper>

### ItemsMapper.java

**public** **interface** ItemsMapper {

//商品列表

**public** List<Items> findItemsList(QueryVo queryVo) **throws** Exception;

}

## Service

目标：

1、Service由spring管理

2、spring对Service进行事务控制。

### applicationContext-service.xml

配置service接口。

### applicationContext-transaction.xml

配置事务管理器。

<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 "*>

<!-- 事务管理器 -->

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

<!-- 数据源 -->

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

</bean>

<!-- 通知 -->

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

<tx:attributes>

<!-- 传播行为 -->

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

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

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

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

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

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

</tx:attributes>

</tx:advice>

<!-- 切面 -->

<aop:config>

<aop:advisor advice-ref=*"txAdvice"*

pointcut=*"execution(\* com.qf.springmvc.service.impl.\*.\*(..))"*/>

</aop:config>

</beans>

### OrderService

**public** **interface** OrderService {

//商品查询列表

**public** List<Items> findItemsList(QueryVo queryVo)**throws** Exception;

}

@Autowired

**private** ItemsMapper itemsMapper;

@Override

**public** List<Items> findItemsList(QueryVo queryVo) **throws** Exception {

//查询商品信息

**return** itemsMapper.findItemsList(queryVo);

}

}

## Action

### springmvc.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 "*>

<!-- 扫描controller注解,多个包中间使用半角逗号分隔 -->

<context:component-scan base-package=*"**com.qf.ssm.controller"*/>

<!--注解映射器 -->

<bean class=*"org.springframework.web.servlet.mvc.method.annotation.RequestMappingHandlerMapping"*/>

<!--注解适配器 -->

<bean class=*"org.springframework.web.servlet.mvc.method.annotation.RequestMappingHandlerAdapter"*/>

<!-- ViewResolver -->

<bean

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

//注解使用

<--<property name=*"viewClass"*

value=*"org.springframework.web.servlet.view.JstlView"* />

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

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

</bean> - ->

</beans>

### web.xml

加载spring容器，配置springmvc前置控制器。

<?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"* xmlns:web=*"http://java.sun.com/xml/ns/javaee/web-app\_2\_5.xsd"*

xsi:schemaLocation=*"http://java.sun.com/xml/ns/javaee http://java.sun.com/xml/ns/javaee/web-app\_2\_5.xsd"*

id=*"WebApp\_ID"* version=*"2.5"*>

<display-name>springmvc</display-name>

<!-- 加载spring容器 -->

<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>

<!-- 解决post乱码 -->

<filter>

<filter-name>CharacterEncodingFilter</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>CharacterEncodingFilter</filter-name>

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

</filter-mapping>

<!-- springmvc的前端控制器 -->

<servlet>

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

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

<!-- contextConfigLocation不是必须的， 如果不配置contextConfigLocation， springmvc的配置文件默认在：WEB-INF/servlet的name+"-servlet.xml" -->

<init-param>

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

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

</init-param>

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

</servlet>

<servlet-mapping>

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

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

</servlet-mapping>

<welcome-file-list>

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

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

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

<welcome-file>default.html</welcome-file>

<welcome-file>default.htm</welcome-file>

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

</welcome-file-list>

</web-app>

### OrderController

@Controller

**public** **class** OrderController {

@Autowired

**private** OrderService orderService;

@RequestMapping("/queryItem.action")

**public** ModelAndView queryItem() **throws** Exception {

// 商品列表

List<Items> itemsList = orderService.findItemsList(**null**);

// 创建modelAndView准备填充数据、设置视图

ModelAndView modelAndView = **new** ModelAndView();

// 填充数据

modelAndView.addObject("itemsList", itemsList);

// 视图

modelAndView.setViewName("order/itemsList");

**return** modelAndView;

}

}

## 测试

http://localhost:8080/springmvc\_mybatis/queryItem.action

# 注解开发-基础

## 需求

使用springmvc+mybatis架构实现商品信息维护。

## 商品修改

### dao

使用逆向工程自动生成的代码：

ItemsMapper.java

ItemsMapper.xml

### service

//根据id查询商品信息

**public** Items findItemById(**int** id) **throws** Exception;

//修改商品信息

**public** **void** saveItem(Items items)**throws** Exception;

### controller

修改商品信息显示页面：

@RequestMapping(value="/editItem")

**public** String editItem(Model model, Integer id) **throws** Exception{

//调用service查询商品信息

Items item = itemService.findItemById(id);

model.addAttribute("item", item);

**return** "item/editItem";

}

修改商品信息提交：

//商品修改提交

@RequestMapping("/editItemSubmit")

**public** String editItemSubmit(Items items)**throws** Exception{

System.*out*.println(items);

itemService.saveItem(items);

**return** "success";

}

### 页面

/WEB-INF/jsp/item/itemsList.jsp

/WEB-INF/jsp/item/editItem.jsp

## @RequestMapping

通过RequestMapping注解可以定义不同的处理器映射规则。

### URL路径映射

@RequestMapping(value="/item")或@RequestMapping("/item）

value的值是数组，可以将多个url映射到同一个方法

### 窄化请求映射

在class上添加@RequestMapping(url)指定通用请求前缀， 限制此类下的所有方法请求url必须以请求前缀开头，通过此方法对url进行分类管理。

如下：

@RequestMapping放在类名上边，设置请求前缀

@Controller

@RequestMapping("/item")

方法名上边设置请求映射url：

@RequestMapping放在方法名上边，如下：

@RequestMapping("/queryItem ")

访问地址为：/item/queryItem

### 请求方法限定

* 限定GET方法

@RequestMapping(method = RequestMethod.*GET*)

如果通过Post访问则报错：

HTTP Status 405 - Request method 'POST' not supported

例如：

@RequestMapping(value="/editItem",method=RequestMethod.GET)

* 限定POST方法

@RequestMapping(method = RequestMethod.*POST*)

如果通过Post访问则报错：

HTTP Status 405 - Request method 'GET' not supported

* GET和POST都可以

@RequestMapping(method={RequestMethod.GET,RequestMethod.POST})

## controller方法返回值

### 返回ModelAndView

controller方法中定义ModelAndView对象并返回，对象中可添加model数据、指定view。

### 返回void

在controller方法形参上可以定义request和response，使用request或response指定响应结果：

1、使用request转向页面，如下：

request.getRequestDispatcher("页面路径").forward(request, response);

2、也可以通过response页面重定向：

response.sendRedirect("url")

3、也可以通过response指定响应结果，例如响应json数据如下：

response.setCharacterEncoding("utf-8");

response.setContentType("application/json;charset=utf-8");

response.getWriter().write("json串");

### 返回字符串

#### 逻辑视图名

controller方法返回字符串可以指定逻辑视图名，通过视图解析器解析为物理视图地址。

//指定逻辑视图名，经过视图解析器解析为jsp物理路径：/WEB-INF/jsp/item/editItem.jsp

**return** "item/editItem";

#### Redirect重定向

Contrller方法返回结果重定向到一个url地址，如下商品修改提交后重定向到商品查询方法，参数无法带到商品查询方法中。

//重定向到queryItem.action地址,request无法带过去

**return** "redirect:queryItem.action";

redirect方式相当于“response.sendRedirect()”，转发后浏览器的地址栏变为转发后的地址，因为转发即执行了一个新的request和response。

由于新发起一个request原来的参数在转发时就不能传递到下一个url，如果要传参数可以/item/queryItem.action后边加参数，如下：

/item/queryItem?...&…..

#### forward转发（默认）

controller方法执行后继续执行另一个controller方法，如下商品修改提交后转向到商品修改页面，修改商品的id参数可以带到商品修改方法中。

//结果转发到editItem.action，request可以带过去

**return** "forward:editItem.action";

forward方式相当于“request.getRequestDispatcher().forward(request,response)”，转发后浏览器地址栏还是原来的地址。转发并没有执行新的request和response，而是和转发前的请求共用一个request和response。所以转发前请求的参数在转发后仍然可以读取到。

## 参数绑定

处理器适配器在执行Handler之前需要把http请求的key/value数据绑定到Handler方法形参数上。

### 默认支持的参数类型

处理器形参中添加如下类型的参数处理适配器会默认识别并进行赋值。

#### HttpServletRequest

通过request对象获取请求信息

#### HttpServletResponse

通过response处理响应信息

#### HttpSession

通过session对象得到session中存放的对象

#### Model/ModelMap

ModelMap是Model接口的实现类，通过Model或ModelMap向页面传递数据，如下：

//调用service查询商品信息

Items item = itemService.findItemById(id);

model.addAttribute("item", item);

页面通过${item.XXXX}获取item对象的属性值。

使用Model和ModelMap的效果一样，如果直接使用Model，springmvc会实例化ModelMap。

### 参数绑定介绍

注解适配器对RequestMapping标记的方法进行适配，对方法中的形参会进行参数绑定，早期springmvc采用PropertyEditor（属性编辑器）进行参数绑定将request请求的参数绑定到方法形参上，3.X之后springmvc就开始使用Converter进行参数绑定。

### 简单类型

当请求的参数名称和处理器形参名称一致时会将请求参数与形参进行绑定。

#### 整型

public String editItem(Model model,Integer id) throws Exception｛

｝

#### 字符串

例子略

#### 单精度/双精度

例子略

#### 布尔型

处理器方法：

public String editItem(Model model,Integer id,Boolean status) throws Exception

请求url：

http://localhost:8080/springmvc\_mybatis/item/editItem.action?id=2&status=false

说明：对于布尔类型的参数，请求的参数值为true或false。

#### @RequestParam

使用@RequestParam常用于处理简单类型的绑定。

value：参数名字，即入参的请求参数名字，如value=“item\_id”表示请求的参数区中的名字为item\_id的参数的值将传入；

required：是否必须，默认是true，表示请求中一定要有相应的参数，否则将报；

TTP Status 400 - Required Integer parameter 'XXXX' is not present

defaultValue：默认值，表示如果请求中没有同名参数时的默认值

定义如下：

public String editItem(@RequestParam(value="item\_id",required=true) String id) {

}

形参名称为id，但是这里使用value=" item\_id"限定请求的参数名为item\_id，所以页面传递参数的名必须为item\_id。

注意：如果请求参数中没有item\_id将跑出异常：

HTTP Status 500 - Required Integer parameter 'item\_id' is not present

这里通过required=true限定item\_id参数为必需传递，如果不传递则报400错误，可以使用defaultvalue设置默认值，即使required=true也可以不传item\_id参数值

### pojo

#### 简单pojo

将pojo对象中的属性名于传递进来的属性名对应，如果传进来的参数名称和对象中的属性名称一致则将参数值设置在pojo对象中

页面定义如下;

<input type="text" name="name"/>

<input type="text" name="price"/>

Contrller方法定义如下：

@RequestMapping("/editItemSubmit")

**public** String editItemSubmit(Items items)**throws** Exception{

System.out.println(items);

请求的参数名称和pojo的属性名称一致，会自动将请求参数赋值给pojo的属性。

#### 包装pojo

如果采用类似struts中对象.属性的方式命名，需要将pojo对象作为一个包装对象的属性，action中以该包装对象作为形参。

包装对象定义如下：

**Public class** QueryVo {

**private** Items items;

}

页面定义：

<input type="text" name="items.name" />

<input type="text" name="items.price" />

Controller方法定义如下：

public String useraddsubmit(Model model,QueryVo queryVo)throws Exception{

System.out.println(queryVo.getItems());

### 自定义参数绑定

#### 需求

根据业务需求自定义日期格式进行参数绑定。

#### Converter

##### 自定义Converter

**import** org.springframework.core.convert.converter.Converter;

**public** **class** CustomDateConverter **implements** Converter<String, Date> {

@Override

**public** Date convert(String source) {

**try** {

SimpleDateFormat simpleDateFormat = **new** SimpleDateFormat("yyyy-MM-dd HH:mm:ss");

**return** simpleDateFormat.parse(source);

} **catch** (Exception e) {

e.printStackTrace();

}

**return** **null**;

}

}

##### 配置方式1（注解方式）

<mvc:annotation-driven conversion-service=*"conversionService"*>

</mvc:annotation-driven>

<!-- conversionService -->

<bean id=*"conversionService"*

class=*"**org.springframework.format.support.FormattingConversionServiceFactoryBean"*>

<!-- 转换器 -->

<property name=*"converters"*>

<list>

<bean class=*"com.qf.ssm.controller.converter.CustomDateConverter"*/>

</list>

</property>

</bean>

##### 配置方式2(自学)（非注解方式）

<!--注解适配器 -->

<bean

class=*"org.springframework.web.servlet.mvc.method.annotation.RequestMappingHandlerAdapter"*>

<property name=*"webBindingInitializer"* ref=*"customBinder"*></property>

</bean>

<!-- 自定义webBinder -->

<bean id=*"customBinder"*

class=*"org.springframework.web.bind.support.ConfigurableWebBindingInitializer"*>

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

</bean>

<!-- conversionService -->

<bean id=*"conversionService"*

class=*"org.springframework.format.support.FormattingConversionServiceFactoryBean"*>

<!-- 转换器 -->

<property name=*"converters"*>

<list>

<bean class=*"com.qf.ssm.controller.converter.CustomDateConverter"*/>

</list>

</property>

</bean>

### 集合类

#### 字符串数组

页面定义如下：

页面选中多个checkbox向controller方法传递

<input type="checkbox" name="item\_id" value="001"/>

<input type="checkbox" name="item\_id" value="002"/>

<input type="checkbox" name="item\_id" value="002"/>

传递到controller方法中的格式是：001,002,003

Controller方法中可以用String[]接收，定义如下：

public String deleteitem(String[] item\_id)throws Exception{

System.out.println(item\_id);

｝

#### List

List中存放对象，并将定义的List放在包装类中，action使用包装对象接收。

List中对象：

成绩对象

**Public class** QueryVo {

**Private** List<Items> itemList;//商品列表

//JSP标签中 itemList[${status.index}].name

//get/set方法..

}

包装类中定义List对象，并添加get/set方法如下：

页面定义如下：

<tr>

<td>

<input type="text" name=" itemsList[0].id" value="${item.id}"/>

</td>

<td>

<input type="text" name=" itemsList[0].name" value="${item.name }"/>

</td>

<td>

<input type="text" name=" itemsList[0].price" value="${item.price}"/>

</td>

</tr>

<tr>

<td>

<input type="text" name=" itemsList[1].id" value="${item.id}"/>

</td>

<td>

<input type="text" name=" itemsList[1].name" value="${item.name }"/>

</td>

<td>

<input type="text" name=" itemsList[1].price" value="${item.price}"/>

</td>

</tr>

上边的静态代码改为动态jsp代码如下：

<c:forEach items=*"*${itemsList }*"* var=*"item"* varStatus=*"s"*>

<tr>

<td><input type=*"text"* name=*"itemsList[*${s.index }*].name"* value=*"*${item.name }*"*/></td>

<td><input type=*"text"* name=*"itemsList[*${s.index }*].price"* value=*"*${item.price }*"*/></td>

.....

.....

</tr>

</c:forEach>

Contrller方法定义如下：

public String useraddsubmit(Model model,QueryVo queryVo)throws Exception{

System.out.println(queryVo.getItemList());

}

#### Map

在包装类中定义Map对象，并添加get/set方法，action使用包装对象接收。

包装类中定义Map对象如下：

**Public class** QueryVo {

private Map<String, Object> itemInfo = new HashMap<String, Object>();

//get/set方法..

}

页面定义如下：

<tr>

<td>学生信息：</td>

<td>

姓名：<inputtype=*"text"*name=*"itemInfo['name']"*/>

年龄：<inputtype=*"text"*name=*"itemInfo['price']"*/>

.. .. ..

</td>

</tr>

Contrller方法定义如下：

public String useraddsubmit(Model model,QueryVo queryVo)throws Exception{

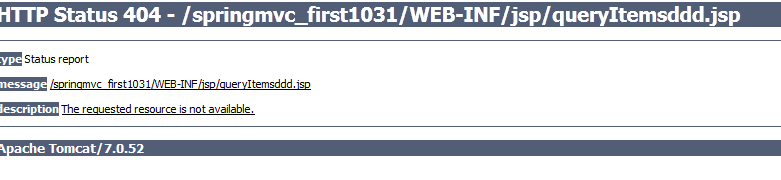
System.out.println(queryVo.getStudentinfo());

}

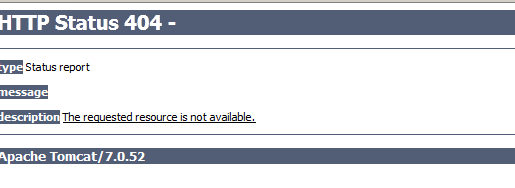
## 问题总结

### 404

页面找不到，视图找不到。



HandlerMapping根据url没有找到Handler。



### Post时中文乱码

在web.xml中加入：

<filter>

<filter-name>CharacterEncodingFilter</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>CharacterEncodingFilter</filter-name>

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

</filter-mapping>

以上可以解决post请求乱码问题。

对于get请求中文参数出现乱码解决方法有两个：

修改tomcat配置文件添加编码与工程编码一致，如下：

<Connector URIEncoding="utf-8" connectionTimeout="20000" port="8080" protocol="HTTP/1.1" redirectPort="8443"/>

另外一种方法对参数进行重新编码：

String userName =new

String(request.getParamter("userName").getBytes("ISO8859-1"),"utf-8")

ISO8859-1是tomcat默认编码，需要将tomcat编码后的内容按utf-8编码

## springmvc与struts2不同

1. springmvc的入口是一个servlet即前端控制器，而struts2入口是一个filter过虑器。
2. **springmvc是基于方法开发(一个url对应一个方法)，请求参数传递到方法的形参，可以设计为单例或多例(建议单例)，struts2是基于类开发，传递参数是通过类的属性，只能设计为多例。**
3. Struts采用值栈存储请求和响应的数据，通过OGNL存取数据， springmvc通过参数解析器是将request请求内容解析，并给方法形参赋值，将数据和视图封装成ModelAndView对象，最后又将ModelAndView中的模型数据通过reques域传输到页面。Jsp视图解析器默认使用jstl。

# 注解开发-高级

## 数据回显（@ModelAttribute）

### 需求

表单提交失败需要再回到表单页面重新填写，原来提交的数据需要重新在页面上显示。

### 简单数据类型

对于简单数据类型，如：Integer、String、Float等使用Model将传入的参数再放到request域实现显示。

如下：

@RequestMapping(value="/editItems",method={RequestMethod.*GET*})

**public** String editItems(Model model,Integer id)**throws** Exception{

//传入的id重新放到request域

model.addAttribute("id", id);

### pojo类型

springmvc默认支持pojo数据回显，springmvc自动将形参中的pojo重新放回request域中，request的key为pojo的类名（首字母小写），如下：

controller方法：

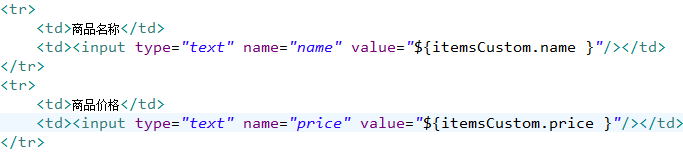
@RequestMapping("/editItemSubmit")

**public** String editItemSubmit(Integer id,ItemsCustom itemsCustom)**throws** Exception{

springmvc自动将itemsCustom放回request，相当于调用下边的代码：

model.addAttribute("itemsCustom", itemsCustom);

jsp页面：



页面中的从“itemsCustom”中取数据。

如果key不是pojo的类名(首字母小写)，可以使用@ModelAttribute完成数据回显。

@ModelAttribute作用如下：

1、绑定请求参数到pojo并且暴露为模型数据传到视图页面

此方法可实现数据回显效果。

// 商品修改提交

@RequestMapping("/editItemSubmit")

**public** String editItemSubmit(Model model,@ModelAttribute("item") ItemsCustom itemsCustom)

页面：

<tr>

<td>商品名称</td>

<td><input type=*"text"* name=*"name"* value=*"*${item.name }*"*/></td>

</tr>

<tr>

<td>商品价格</td>

<td><input type=*"text"* name=*"price"* value=*"*${item.price }*"*/></td>

</tr>

如果不用@ModelAttribute也可以使用model.addAttribute("item", itemsCustom)完成数据回显。

2、将方法返回值暴露为模型数据传到视图页面

//商品分类

@ModelAttribute("itemtypes")

**public** Map<String, String> getItemTypes(){

Map<String, String> itemTypes = **new** HashMap<String,String>();

itemTypes.put("101", "数码");

itemTypes.put("102", "母婴");

**return** itemTypes;

}

页面：

商品类型：

<select name=*"itemtype"*>

<c:forEach items=*"*${itemtypes }*"* var=*"itemtype"*>

<option value=*"*${itemtype.key }*"*>${itemtype.value }</option>

</c:forEach>

</select>

## **异常处理器**

springmvc在处理请求过程中出现异常信息交由异常处理器进行处理，自定义异常处理器可以实现一个系统的异常处理逻辑。

### 异常处理思路

系统中异常包括两类：预期异常和运行时异常RuntimeException，前者通过捕获异常从而获取异常信息，后者主要通过规范代码开发、测试通过手段减少运行时异常的发生。

系统的dao、service、controller出现都通过throws Exception向上抛出，最后由springmvc前端控制器交由异常处理器进行异常处理，如下图：

Controller

客户端

Service

Dao

Springmvc

DispatcherServlet

请求

异常

ExceptionResolver

异常处理器

异常

异常

### 自定义异常类

为了区别不同的异常通常根据异常类型自定义异常类，这里我们创建一个自定义系统异常，如果controller、service、dao抛出此类异常说明是系统预期处理的异常信息。

**public** **class** CustomException **extends** Exception {

/\*\* serialVersionUID\*/

**private** **static** **final** **long** *serialVersionUID* = -5212079010855161498L;

**public** CustomException(String message){

**super**(message);

**this**.message = message;

}

//异常信息

**private** String message;

**public** String getMessage() {

**return** message;

}

**public** **void** setMessage(String message) {

**this**.message = message;

}

}

### 自定义异常处理器

**public** **class** CustomExceptionResolver **implements** HandlerExceptionResolver {

@Override

**public** ModelAndView resolveException(HttpServletRequest request,

HttpServletResponse response, Object handler, Exception ex) {

ex.printStackTrace();

CustomException customException = **null**;

//如果抛出的是系统自定义异常则直接转换

**if**(ex **instanceof** CustomException){

customException = (CustomException)ex;

}**else**{

//如果抛出的不是系统自定义异常则重新构造一个未知错误异常。

customException = **new** CustomException("未知错误，请与系统管理 员联系！");

}

ModelAndView modelAndView = **new** ModelAndView();

modelAndView.addObject("message", customException.getMessage());

modelAndView.setViewName("error");

**return** modelAndView;

}

}

### 错误页面

<%@ page language=*"java"* contentType=*"text/html; charset=UTF-8"*

pageEncoding=*"UTF-8"*%>

<%@ taglib uri=*"http://java.sun.com/jsp/jstl/core"* prefix=*"c"* %>

<%@ taglib uri=*"http://java.sun.com/jsp/jstl/fmt"* prefix=*"fmt"*%>

<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN" "http://www.w3.org/TR/html4/loose.dtd">

<html>

<head>

<meta http-equiv=*"Content-Type"* content=*"text/html; charset=UTF-8"*>

<title>错误页面</title>

</head>

<body>

您的操作出现错误如下：<br/>

${message }

</body>

</html>

### 异常处理器配置

在springmvc.xml中添加：

<!-- 异常处理器 -->

<bean id=*"handlerExceptionResolver"* class=*"com.qf.ssm.controller.exceptionResolver.CustomExceptionResolver"*/>

### 异常测试

修改商品信息，id输入错误提示商品信息不存在。

修改controller方法“editItem”，调用service查询商品信息，如果商品信息为空则抛出异常：

// 调用service查询商品信息

Items item = itemService.findItemById(id);

**if**(item == **null**){

**throw** **new** CustomException("商品信息不存在!");

}

在service中抛出异常方法同上。

</servlet-mapping>

### URL 模板模式映射（@PathVariable）

@RequestMapping(value="/ viewItems/{id}")：{×××}占位符，请求的URL可以是“/viewItems/1”或“/viewItems/2”，通过在方法中使用@PathVariable获取{×××}中的×××变量。

@PathVariable用于将请求URL中的模板变量映射到功能处理方法的参数上。

@RequestMapping("/viewItems/{id}")

**public** @ResponseBody viewItems(@PathVariable("id") String id,Model model) **throws** Exception{

//方法中使用@PathVariable获取useried的值，使用model传回页面

//调用 service查询商品信息

ItemsCustom itemsCustom = itemsService.findItemsById(id);

**return** itemsCustom;

}

如果RequestMapping中表示为"/viewItems/{id}"，id和形参名称一致，@PathVariable不用指定名称。

### 静态资源访问<mvc:resources>

如果在DispatcherServlet中设置url-pattern为 /则必须对静态资源进行访问处理。

spring mvc 的<mvc:resources mapping="" location="">实现对静态资源进行映射访问。

如下是对js文件访问配置：

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

# 练习

商品查询添加查询条件：商品名称、价格范围

添加商品功能开发

删除商品功能开发

# 1 文件上传入门

## 1.1 实现文件上传条件

1）表单的提交方式必须是POST方式。（才有content-type属性）

2）有文件上传表单，表单中有<input type="file"/>的选择文件的标签

3）把表单设置为enctype="multipart/form-data"，提交的数据不再是key-value对，而是字节数据

|  |
| --- |
| <form action=*"*${pageContext.request.contextPath }*/UploadDemo1"* method=*"post"* enctype=*"multipart/form-data"*>  请选择文件： <input type=*"file"* name=*"img"*/><br/>  <input type=*"submit"* value=*"上传"* />  </form> |

# 2 工具实现文件上传

## 2.1 commons-fileupload组件

是Apache组织旗下的开源的文件上传的组件。使用非常简单易用。

## 2.2 核心的API

1. DiskFileItemFactory类： 用于创建上传对象，设置文件缓存区大小，设置文件缓存目录。

DiskFileItemFactory(参数1,参数2)**参数一： 表示文件缓存区的大小。如果上传的文件没有超过缓存区大小，则文件不缓存；否则缓存文件，缓存到临时目录。（byte）**

**\* 参数二： 表示缓存区的临时目录。**

**DiskFileItemFactory factory = new DiskFileItemFactory();**

2. ServletFileUpload类： 用于在Servlet程序中实现文件上传

List<FileItem> parseRequest = upload.parseRequest(request);

parseRequest方法 List<FileItem> list = parseRequest(request): 用于解析请求数据，提取和封装文件信息。

3. FileItem类： 封装一个文件的所有相关的信息javabean。包含文件名称，文件大小，文件类型， 文件数据内容。

判断是否是普通表单项

boolean formField = item.isFormField();

/\*\*

\* 4.把文件数据内容存储到服务器端的硬盘中

\*/

FileUtils.*copyInputStreamToFile*(in, **new** File("e:/files/"+fileName));

/\*\*

\* 文件复制的方法

\*/

IOUtils.*copy*(in, os);

/\*\*

\* 5.文件上传完毕，手动清理缓存文件

\*/

file.delete();

## 2.3 开发步骤

1）导入commoms-fileuload的jar包

commons-fileupload-1.2.2.jar 核心包

commons-io-2.1.jar 辅助包

2）编写程序

# 3 实现单文件上传

|  |
| --- |
| //1.创建DiskFileItemFactory类  /\*\*  \*  \*/  DiskFileItemFactory factory = **new** DiskFileItemFactory(10\*1024,**new** File("e:/temp/"));    //2.创建ServletFileUpload类  ServletFileUpload upload = **new** ServletFileUpload(factory);    /\*\*  \* 设置文件名的编码  \*/  upload.setHeaderEncoding("utf-8");    //3.解析request数据（把每一个文件封装到FileItem对象中，FileItem放入List中）  **try** {  List<FileItem> list = upload.parseRequest(request);    //取出第一个上传的文件  FileItem file = list.get(0);    //得到文件名(getName())  String fileName = file.getName();  //得到文件大小  **long** fileSize = file.getSize();  //得到内容类型  String contentType = file.getContentType();  //得到文件数据内容  InputStream in = file.getInputStream();    /\*\*  \* 4.把文件数据内容存储到服务器端的硬盘中  \*/  FileUtils.*copyInputStreamToFile*(in, **new** File("e:/files/"+fileName));    /\*\*  \* 5.文件上传完毕，手动清理缓存文件  \*/  file.delete();      System.*out*.println("文件名："+fileName);  System.*out*.println("文件大小："+fileSize);  System.*out*.println("文件类型："+contentType);  System.*out*.println("文件数据内容："+in);    } **catch** (FileUploadException e) {  e.printStackTrace();  } |

# 4 实现多文件上传

|  |
| --- |
| //1.创建DiskFileItemFactory对象  DiskFileItemFactory factory = **new** DiskFileItemFactory(10\*1024, **new** File("e:/temp/"));  //2.创建ServletFileUpload对象  ServletFileUpload upload = **new** ServletFileUpload(factory);  //3.设置文件编码  upload.setHeaderEncoding("utf-8");  //4.开始解析文件  **try** {  List<FileItem> list = upload.parseRequest(request);  **if**(list!=**null**){  List<UploadFile> ufList = **new** ArrayList<UploadFile>();  //遍历多个文件  **for**(FileItem file: list){  //取出文件相关信息  String fileName = file.getName();  **long** fileSize = file.getSize();  String contentType = file.getContentType();  //封装到javabean中  UploadFile uf = **new** UploadFile();  uf.setFileName(fileName);  uf.setFileSize(fileSize);  uf.setFileType(contentType);  //放入list中  ufList.add(uf);    //把文件保存到服务器端的硬盘  FileUtils.*copyInputStreamToFile*(file.getInputStream(), **new** File("e:/files/"+fileName));  //删除缓存文件  file.delete();  }  request.setAttribute("ufList", ufList);  request.getRequestDispatcher("/success.jsp").forward(request, response);  }  } **catch** (FileUploadException e) {  e.printStackTrace();  } |

# 5 动态选择多文件上传

## 6.4 处理文件名重复问题

|  |
| --- |
| /\*\*  \* 解决文件名重复问题：  \* 1）日期\_时间\_随机数.jpg  \* 2）使用UUID算法（在一台PC都是唯一的）  \*/  String fileName = file.getName();  //得到源文件的后缀名  String supfix = fileName.substring(fileName.lastIndexOf(".")); //.jpg  //使用UUID算法生成随机名称  fileName = UUID.*randomUUID*().toString()+supfix; |

# 7 文件下载

## 7.1 普通文件下载

使用超链接。缺点：1）暴露文件的路径 2）扩展型和安全性不好

## 7.2 使用servlet程序下载（推荐）

|  |
| --- |
| //得到需要下载的文件  String path = **this**.getServletContext().getRealPath("/upload/9/1/图片1.png");    File file = **new** File(path);  //读取服务器本地的文件  FileInputStream in = **new** FileInputStream(file);    /\*\*  \* 处理URL编码问题  \*/  String fileName = file.getName();  //对文件名进行URl编码  fileName = URLEncoder.*encode*(fileName, "utf-8");    //判断不同浏览器  String userAgent = request.getHeader("user-agent");  String filefix = **null**;  **if**(userAgent.contains("Trident")){  //IE  filefix = "filename="+fileName;  }**else** **if**(userAgent.contains("Firefox")){  //Firefox  filefix = "filename\*="+fileName;  }**else**{  filefix = "filename="+fileName;  }    //告诉浏览器以下载方式打开资源  response.setHeader("Content-Disposition", "attachment;"+filefix);    //把本地文件发送给浏览器  **byte**[] buf = **new** **byte**[1024];  **int** len = 0;  **while**( (len=in.read(buf))!=-1 ){  response.getOutputStream().write(buf, 0, len);  }  //关闭  in.close(); |