# ssm综合应用

## 需求

小型的用户数据管理系统，实现对用户数据的CRUD，并提供Excel的下载。

学习目标：

1. springMVC和Mybatis的整合
2. 使用SpringMVC+Mybatis实现数据库的CRUD
3. 使用SpringMVC实现与前端的交互
4. 实现Excel的下载，加深对视图的理解

## 使用技术

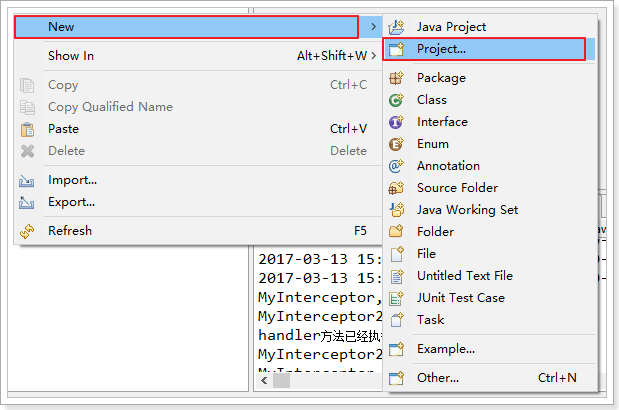
Spring + SpringMVC + Mybatis

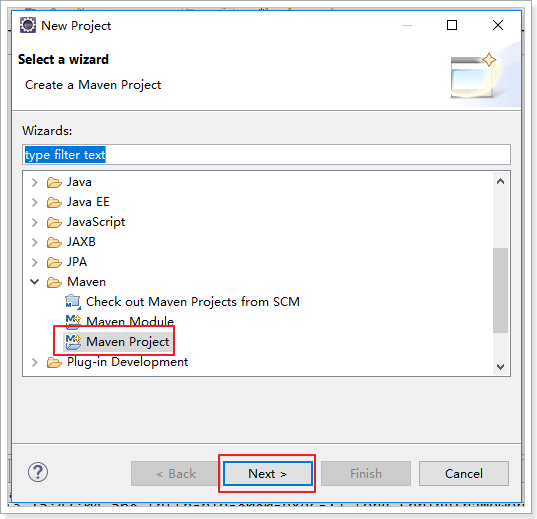
数据库：mysql

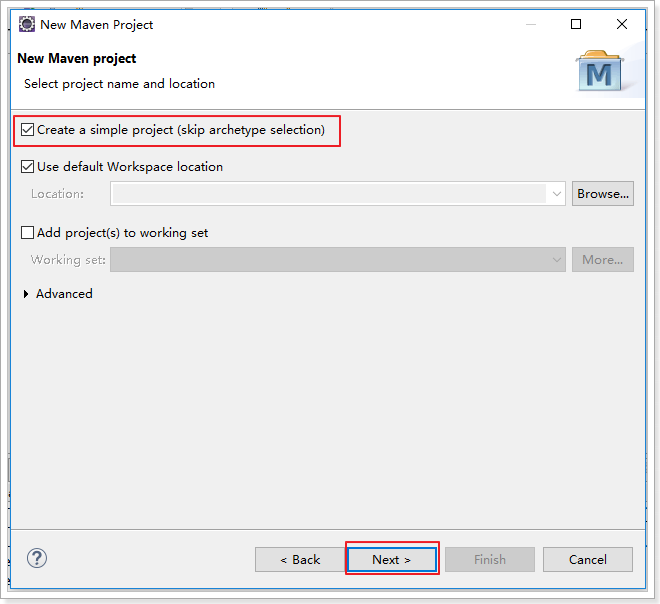
前端：Jquery EasyUI

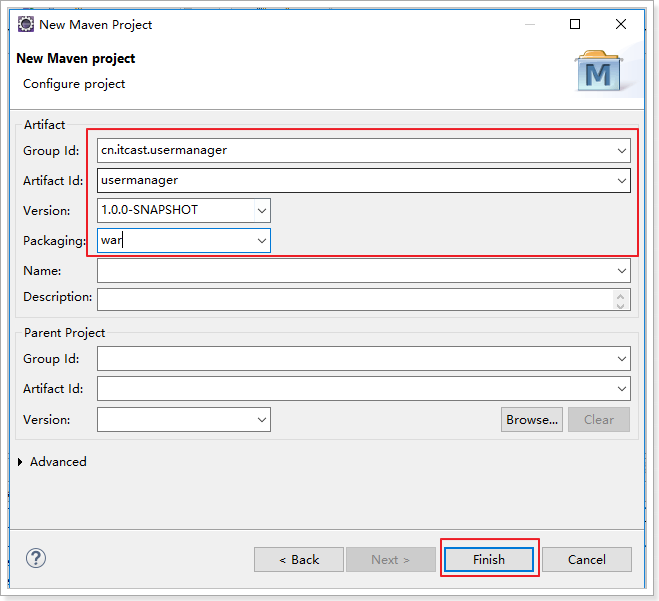
# 创建工程---环境整理

## 创建工程

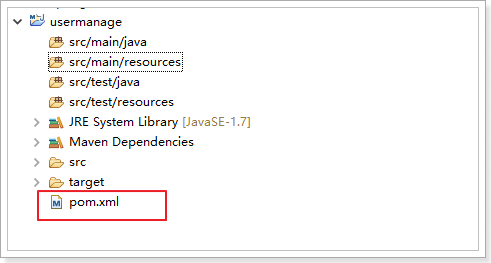








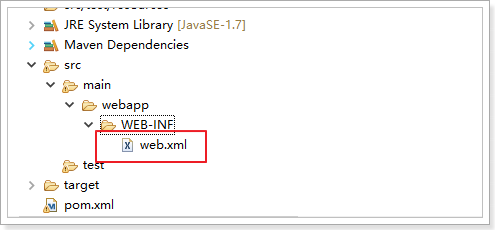
## 引入依赖



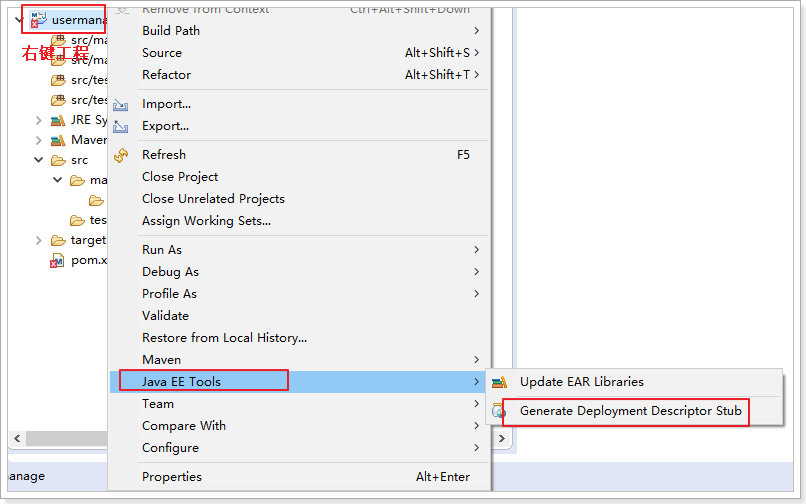
参照itcast-parent工程的pom.xml，在pom.xml中引入所需依赖

|  |
| --- |
| <dependencies>  <!-- 单元测试 -->  <dependency>  <groupId>junit</groupId>  <artifactId>junit</artifactId>  <scope>test</scope>  </dependency>  <!-- Spring -->  <dependency>  <groupId>org.springframework</groupId>  <artifactId>spring-webmvc</artifactId>  </dependency>  <dependency>  <groupId>org.springframework</groupId>  <artifactId>spring-jdbc</artifactId>  </dependency>  <dependency>  <groupId>org.springframework</groupId>  <artifactId>spring-aspects</artifactId>  </dependency>  <!-- Mybatis -->  <dependency>  <groupId>org.mybatis</groupId>  <artifactId>mybatis</artifactId>  </dependency>  <dependency>  <groupId>org.mybatis</groupId>  <artifactId>mybatis-spring</artifactId>  </dependency>  <!-- MySql -->  <dependency>  <groupId>mysql</groupId>  <artifactId>mysql-connector-java</artifactId>  </dependency>  <dependency>  <groupId>org.slf4j</groupId>  <artifactId>slf4j-log4j12</artifactId>  </dependency>  <!-- Jackson Json处理工具包 -->  <dependency>  <groupId>com.fasterxml.jackson.core</groupId>  <artifactId>jackson-databind</artifactId>  </dependency>  <!-- 连接池 -->  <dependency>  <groupId>com.alibaba</groupId>  <artifactId>druid</artifactId>  </dependency>  <!-- JSP相关 -->  <dependency>  <groupId>jstl</groupId>  <artifactId>jstl</artifactId>  </dependency>  <dependency>  <groupId>javax.servlet</groupId>  <artifactId>servlet-api</artifactId>  <scope>provided</scope>  </dependency>  <dependency>  <groupId>javax.servlet</groupId>  <artifactId>jsp-api</artifactId>  <scope>provided</scope>  </dependency>  <dependency>  <groupId>com.github.pagehelper</groupId>  <artifactId>pagehelper</artifactId>  <version>3.7.5</version>  </dependency>  <dependency>  <groupId>com.github.jsqlparser</groupId>  <artifactId>jsqlparser</artifactId>  <version>0.9.1</version>  </dependency>  <!-- 时间操作组件 -->  <dependency>  <groupId>joda-time</groupId>  <artifactId>joda-time</artifactId>  </dependency>  <dependency>  <groupId>org.apache.poi</groupId>  <artifactId>poi</artifactId>  <version>3.10.1</version>  </dependency>  <dependency>  <groupId>com.github.abel533</groupId>  <artifactId>mapper</artifactId>  <version>2.3.4</version>  </dependency>  </dependencies>  <build>  <plugins>  <!-- 配置Tomcat插件 -->  <plugin>  <groupId>org.apache.tomcat.maven</groupId>  <artifactId>tomcat7-maven-plugin</artifactId>  <configuration>  <port>8080</port>  <path>/</path>  </configuration>  </plugin>  </plugins>  </build> |

## 配置web.xml



如果新创建的web工程里没有该文件，可以手动创建WEB-INF目录及web.xml；也可以通过如下方式自动创建：



Web.xml的内容如下：

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

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

xmlns=*"http://java.sun.com/xml/ns/javaee"*

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

id=*"MyWebApp"* version=*"2.5"*>

<display-name>usermanage</display-name>

<!-- Spring 容器的初始化-->

<context-param>

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

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

</context-param>

<!--Spring的ApplicationContext 载入 -->

<listener>

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

</listener>

<!-- 编码过滤器，以UTF8编码 解决post提交的问题,如果是get,那么需要手动解决-->

<filter>

<filter-name>encodingFilter</filter-name>

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

<init-param>

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

<param-value>UTF8</param-value>

</init-param>

</filter>

<filter-mapping>

<filter-name>encodingFilter</filter-name>

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

</filter-mapping>

<!-- 配置SpringMVC -->

<servlet>

<servlet-name>usermanage</servlet-name>

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

<init-param>

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

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

</init-param>

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

</servlet>

<servlet-mapping>

<servlet-name>usermanage</servlet-name>

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

</servlet-mapping>

<welcome-file-list>

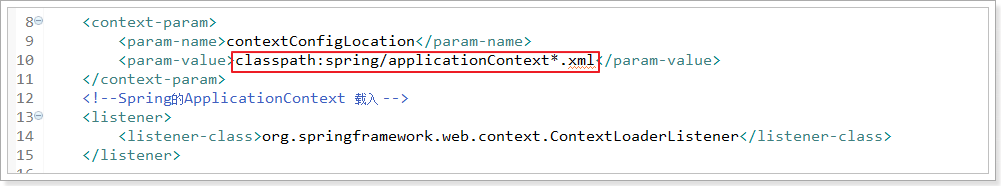
<welcome-file>index.jsp</welcome-file>

</welcome-file-list>

</web-app>

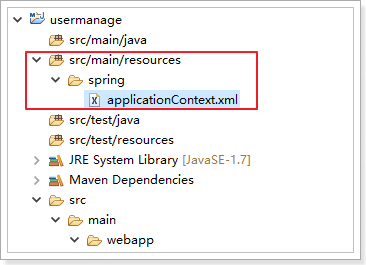
## 配置applicationContext.xml

1. applicationContext.xml文件应该放在什么位置？



参照web.xml中的配置（如上图），spring容器的监听器加载的规则是：spring/applicationContext\*.xml，所以该文件应该放在src/main/ resources目录下的spring目录下。

1. 创建spring目录并在其目录下新建applicationContext.xml文件



applicationContext.xml的内容如下：

<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"*>

<!-- 使用spring自带的占位符替换功能 -->

<bean

class=*"org.springframework.beans.factory.config.PropertyPlaceholderConfigurer"*>

<!-- 允许JVM参数覆盖 -->

<property name=*"systemPropertiesModeName"* value=*"SYSTEM\_PROPERTIES\_MODE\_OVERRIDE"* />

<!-- 忽略没有找到的资源文件 -->

<property name=*"ignoreResourceNotFound"* value=*"true"* />

<!-- 配置资源文件 -->

<property name=*"locations"*>

<list>

<value>classpath:jdbc.properties</value>

</list>

</property>

</bean>

<!-- 开启注解扫描 -->

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

<!-- 配置数据源 -->

<bean id=*"dataSource"* class=*"com.alibaba.druid.pool.DruidDataSource"*

init-method=*"init"* destroy-method=*"close"*>

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

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

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

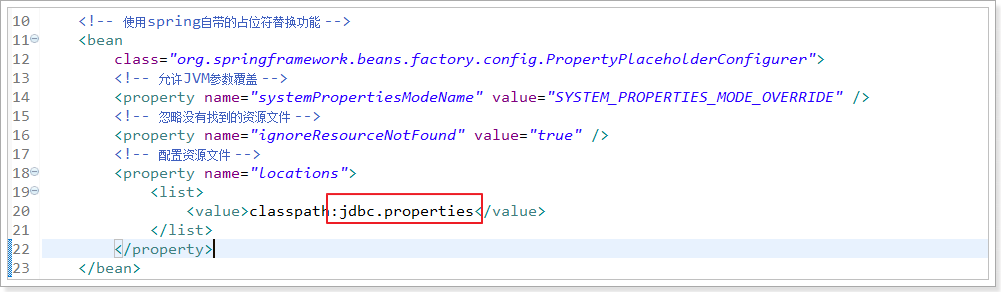
<property name=*"password"* value=*"${jdbc.password}"* />

</bean>

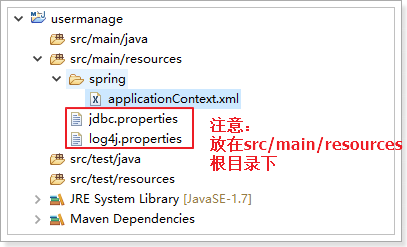
</beans>

如上黄色标识。于是，又需要引入jdbc.properties资源文件

1. 引入资源文件（jdbc.properties以及log4j.properties）



由于applicationContext.xml中数据源的连接信息是配置在jdbc.properties资源文件中的，所以这里需要引入该资源文件，参考mybatis工程，可直接copy过来（顺便把log4j.properties文件一块儿copy过来）。如下：



jdbc.properties内容如下：

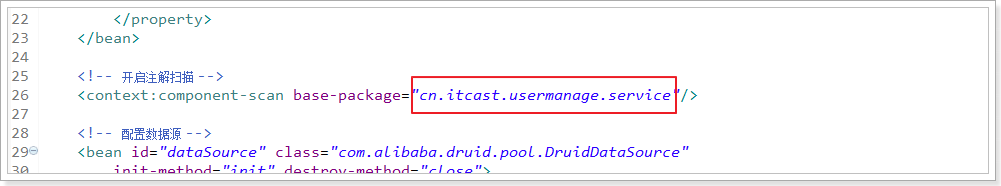
jdbc.driver=com.mysql.jdbc.Driver

jdbc.url=jdbc:mysql://127.0.0.1:3306/mybatis-49

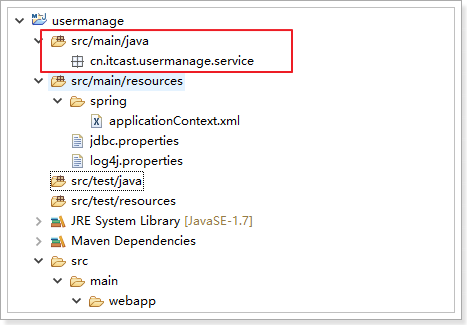
jdbc.username=root

jdbc.password=root

1. 创建spring扫描包对应的目录

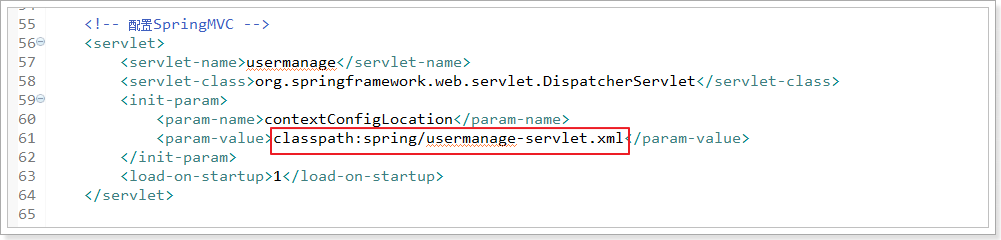


如上图，参照applicationContext.xml中，注解扫描的包路径，创建在src/main/java目录下创建对应的包。如下图：



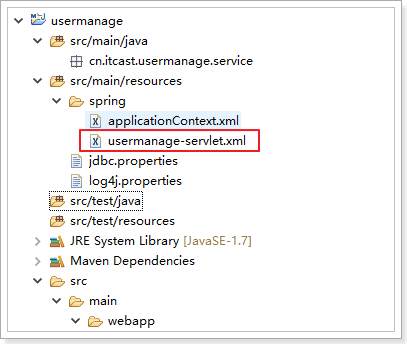
## 配置springmvc的配置文件

* 1. springmvc的配置文件应该怎么命名？放在什么位置？



如上图，该图来自web.xml中关于springmvc入口的配置，此处通过contextConfigLocation指定springmvc的配置路径为classpath下的spring目录，并以{servlet-name}-servlet.xml命名。

* 1. 在spring目录下创建usermanage-servlet.xml



内容为：

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

<beans xmlns=*"http://www.springframework.org/schema/beans"*

xmlns:xsi=*"http://www.w3.org/2001/XMLSchema-instance"* xmlns:p=*"http://www.springframework.org/schema/p"*

xmlns:context=*"http://www.springframework.org/schema/context"*

xmlns:mvc=*"http://www.springframework.org/schema/mvc"*

xsi:schemaLocation=*"http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans.xsd*

*http://www.springframework.org/schema/mvc http://www.springframework.org/schema/mvc/spring-mvc-4.0.xsd*

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

<!-- 配置注解驱动，替代推荐使用的映射器以及适配器，json转换器 -->

<mvc:annotation-driven />

<!-- 开启注解扫描 -->

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

<!-- 配置视图解析器 -->

<!-- Example: prefix="/WEB-INF/jsp/", suffix=".jsp", viewname="test" -> "/WEB-INF/jsp/test.jsp" -->

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

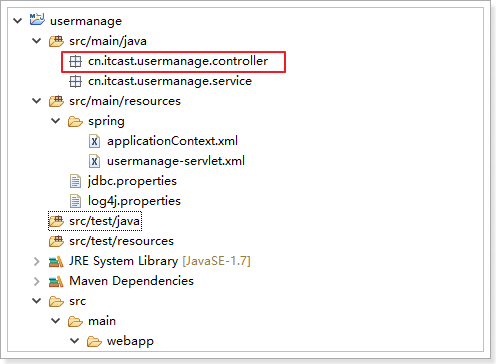
<property name=*"prefix"* value=*"/WEB-INF/views/"*></property>

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

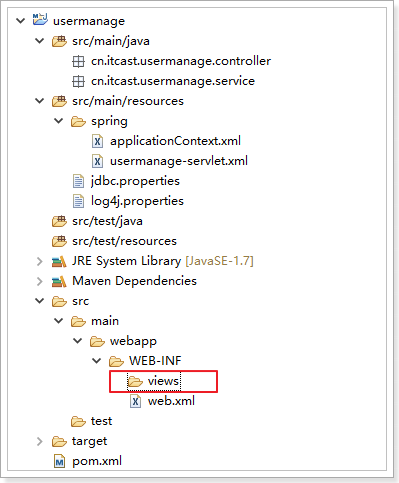
</bean>

</beans>

* 1. 创建usermanage-servlet.xml配置文件中的注解扫描对应的包目录

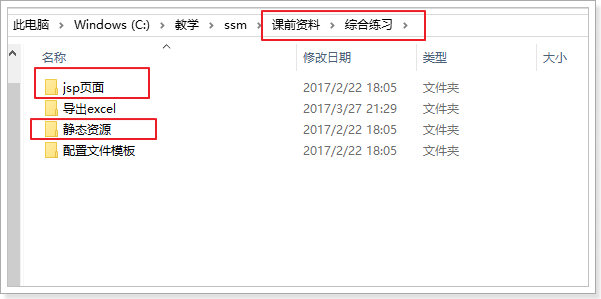


* 1. 创建视图解析器对应的视图目录

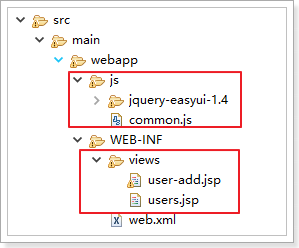


## 导入相关jsp及js

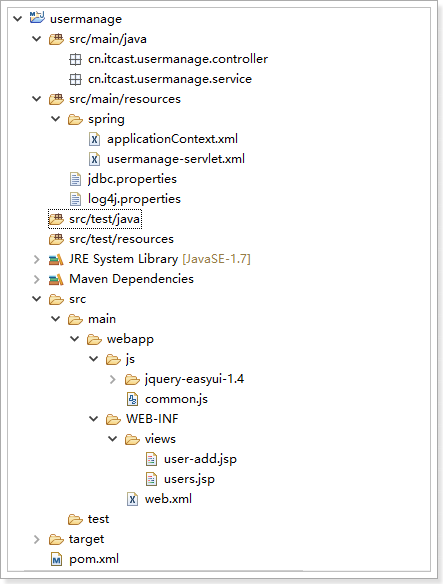
参考综合练习课前资料:



把jsp页面导入到view目录下，把静态资源中的js导入到webapp目录下。导入后，目录结构：



## 目录结构



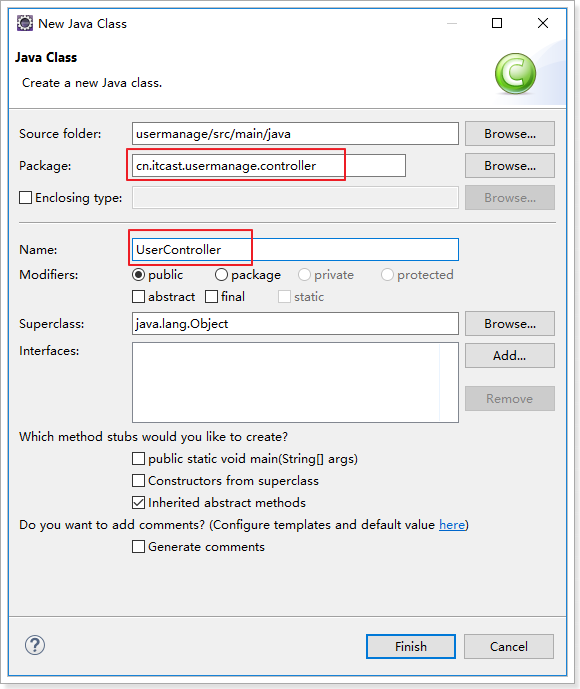
# Spring与springmvc的整合测试

Springmvc与spring是同一个体系下的，他们在没有特殊需求的情况下是不用整合的

## 跳转到users.jsp

由于users.jsp在WEB-INF下，不能直接访问到该资源。必须通过Controller方法做跳转。

创建UserController：



编写UserController方法跳转到users.jsp

@RequestMapping("user")

@Controller

**public** **class** UserController {

@RequestMapping(value="users")

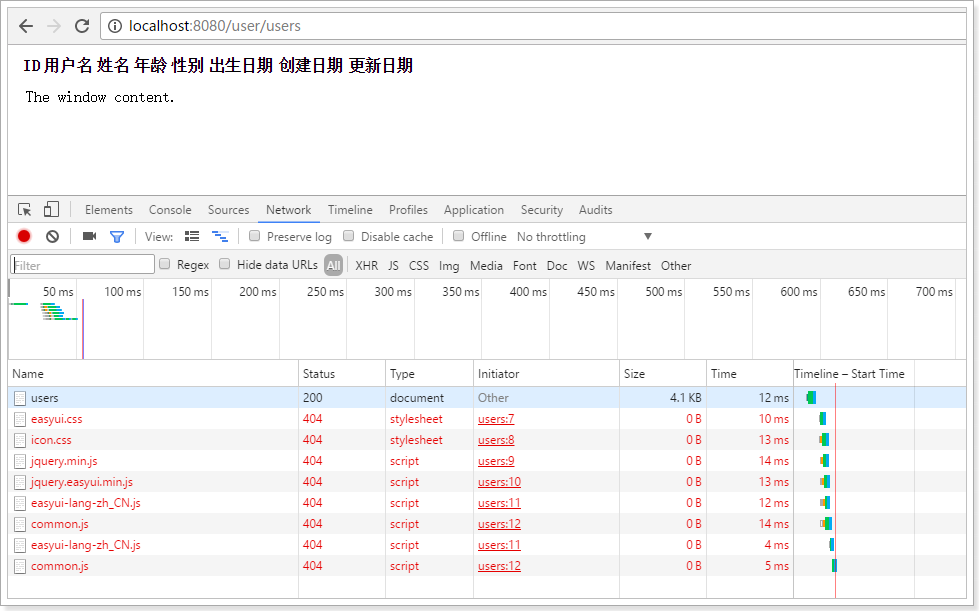
**public** String toUsers(){

**return** "users";

}

}

启动tomcat之后的访问效果：



按F12查看浏览器的控制台，发现所有的静态资源都404，但是，在jsp页面中引入静态资源的地方，检查路径没有问题，为什么会报错呢？

## 解决静态资源拦截

原因：

由于我们设置DispatchServlet的映射规则是“/”，所以页面中的所有请求将被拦截，也包括静态资源，后端的Handler是无法处理静态资源的，所以会导致访问静态资源会404。如下图（来自web.xml截图）



解决方案：

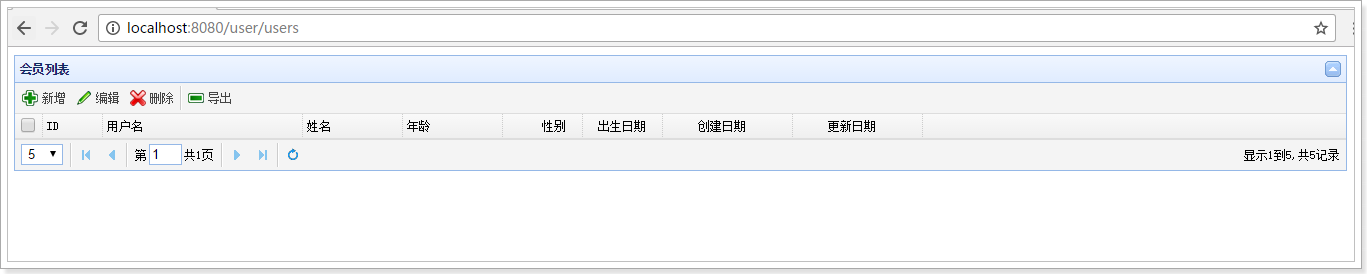
springmvc提供了一种解决方案就是在springmvc的配置文件中添加配置：

<mvc:default-servlet-handler />

在usermanage-servlet.xml中配置如下：



刷新页面后，页面正常显示，说明静态资源可以正常访问了：



# 整合mybatis

在spring目录下新建applicationContext-mybatis.xml，这样不同的框架或工具跟spring的整合分开配置，方便管理。

整合文档参考课前资料：



## 配置applicationContext-mybatis.xml

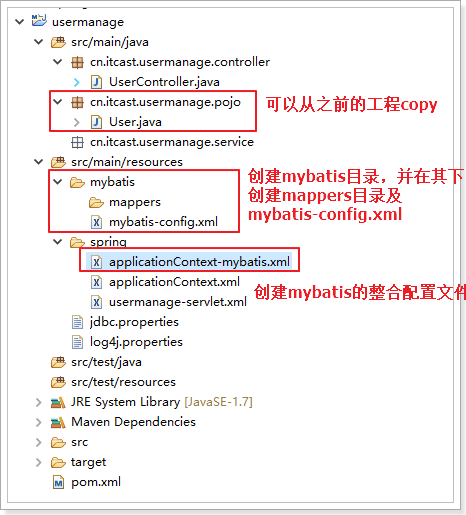
思路：spring整合mybatis，能整合mybatis的什么？

sqlSessionFactory、sqlSession、mapper接口的初始化，过于复杂，能否交给spring

mybatis全局配置文件的读取，能否交给spring

映射文件的引入，之前存在瑕疵，能否解决

在spring包下创建applicationContext-mybatis.xml配置文件，约束头信息参考applicationContext.xml配置；mybatis-config.xml的内容参考之前的配置，并创建cn.itcast.usernamege.pojo包目录。



Mybatis-config.xml内容：

<?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>

<settings>

<!-- 行为参数，name:参数名，value：参数值，默认为false，true：开启驼峰匹配，即从经典的数据库列名到经典的java属性名 -->

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

</settings>

<typeAliases>

<!-- 别名扫描，name:包的路径 -->

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

</typeAliases>

<!-- 映射文件 -->

<!-- <mappers>

<mapper resource="mybatis/mappers/UserMapper.xml"/>

</mappers> -->

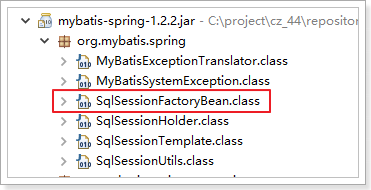
</configuration>

### 构建SqlSessionFactory

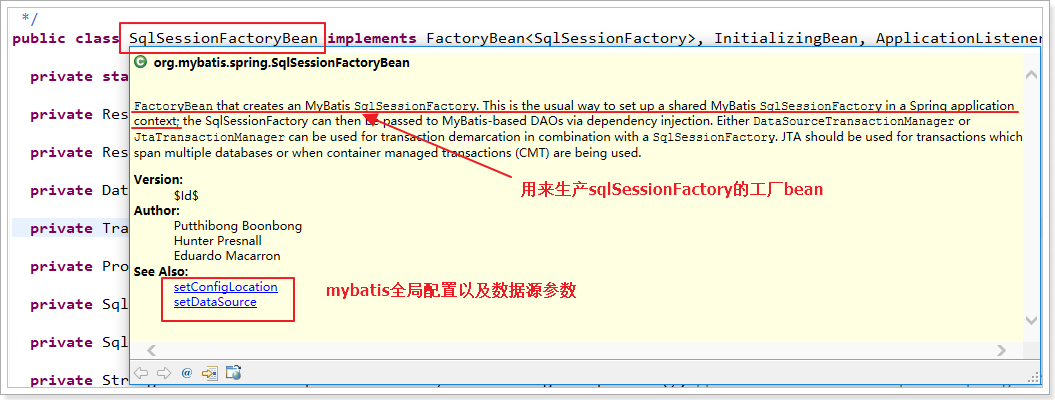
SqlSessionFactory的构建过程，在mybatis中是比较麻烦的。当mybatis遇到spring之后，这个问题变的非常简单了。

在mybatis-spring的整合包下，存在一个sqlSessionFactoryBean，它是用来在spring容器中生产sqlSessionFactory的工厂Bean

sqlSessionFactoryBean源码位置：



查看sqlSessionFactoryBean源码注释：



结论：

应该在applicationContext-mybatis.xml中配置SqlSessionFactoryBean，并且配置dataSource以及configLocation的属性

配置applicationContext-mybatis.xml：

<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"*>

<!-- spring构建sqlSessionFactory -->

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

<!-- 指定mybatis的数据源 -->

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

<!-- 指定mybatis的全局配置文件 -->

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

</bean>

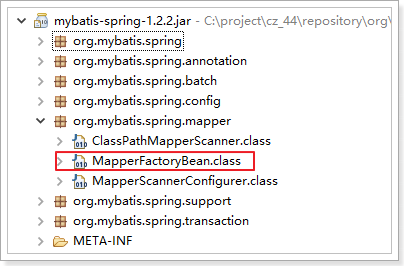
</beans>

### 配置mapper接口

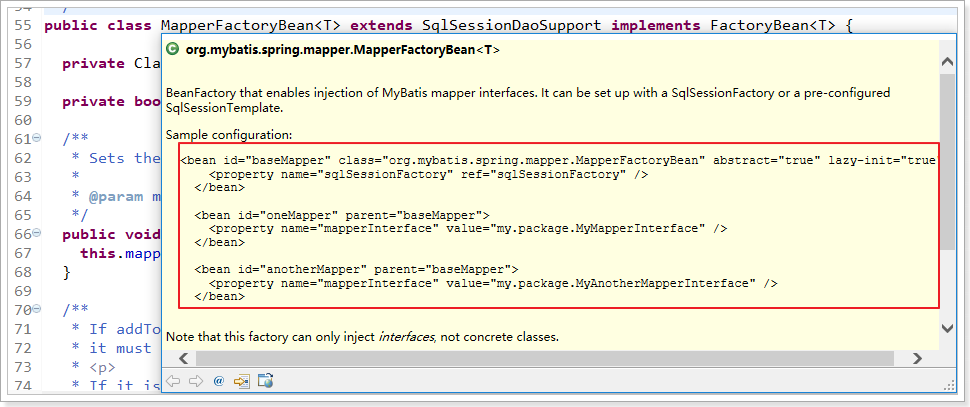
既然sqlSessionFactory交给spring管理了，那么mybatis的mapper接口的动态代理实现能不能也交给spring进行管理呢？

1. 参照源码注释

在mybatis-spring的整合包下，存在MapperFactoryBean这样一个工厂bean，它可以帮咱们完成：



注释中提供了使用案例：



2、 参照mybatis整合spring的官方文档：



在applicationContext-mybatis.xml中，将Mapper接口交给spring管理：

<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"*>

<!-- spring构建sqlSessionFactory -->

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

<!-- 指定mybatis的数据源 -->

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

<!-- 指定mybatis的全局配置文件 -->

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

</bean>

<!-- spring实例化usermapper的动态实现 -->

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

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

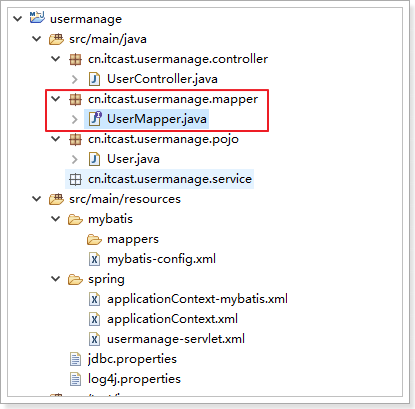
<!-- *sqlSessionFactory 在只有1个的情况下,可以省略* -->

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

</bean>

</beans>

创建cn.itcast.usermanage.mapper目录及UserMapper接口：



## Junit测试整合

思路：

1. 在UserMapper接口中定义一个方法（根据id查询用户信息）
2. 在mybatis/mappers目录下新建接口对应的UserMapper.xml映射文件，并定义根据id查询用户信息的Statement
3. 创建UserMapper接口的junit test cast（即UserMapperTest.java），在setup方法中通过spring容器来初始化UserMapper对象

UserMapper接口：

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

/\*\*

\* 根据id获取User信息

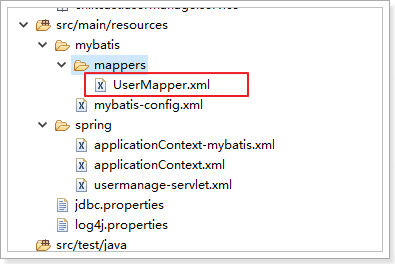
\* **@return**

\*/

**public** User queryUserById(Long id);

}

UserMapper.xml：



内容：

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

<!DOCTYPE mapper

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

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

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

<select id=*"queryUserById"* resultType=*"User"*>

select \* from tb\_user where id = #{id}

</select>

</mapper>

给UserMapper接口创建junit测试用例UserMapperTest，内容：

**public** **class** UserMapperTest {

**public** UserMapper userMapper;

@Before

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

ApplicationContext context = **new** ClassPathXmlApplicationContext("spring/applicationContext.xml",

"spring/applicationContext-mybatis.xml");

**this**.userMapper = context.getBean(UserMapper.**class**);

}

@Test

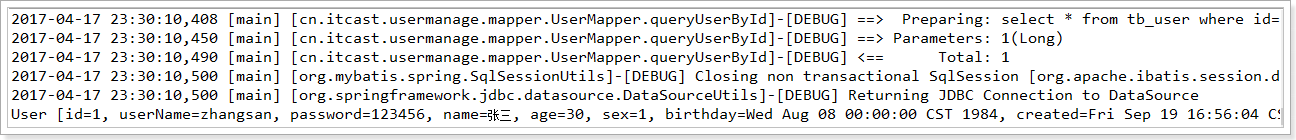
**public** **void** testQueryUserById() {

System.***out***.println(**this**.userMapper.queryUserById(1l));

}

}

控制台打印出用户信息，说明整合成功，日志输出：



## 优化整合程序

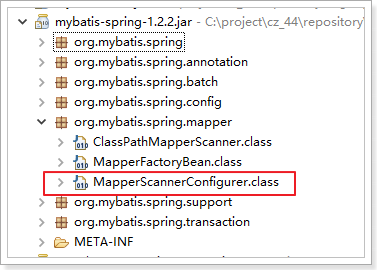
那些配置需要优化：

1. Mapper接口的spring配置太过麻烦，每一个Mapper接口都要去配置
2. Mapper映射文件，每次都要在mybatis-config.xml中引入
3. 别名扫描，是spring比较擅长的，能否交给spring管理

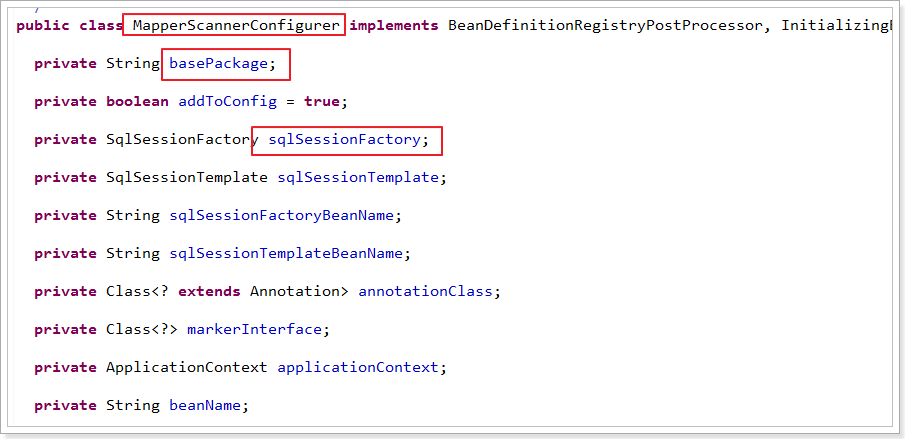
### 使用Mapper接口扫描

Mapper接口的配置太麻烦，每次都要去配置，并且每次都要配置多行。

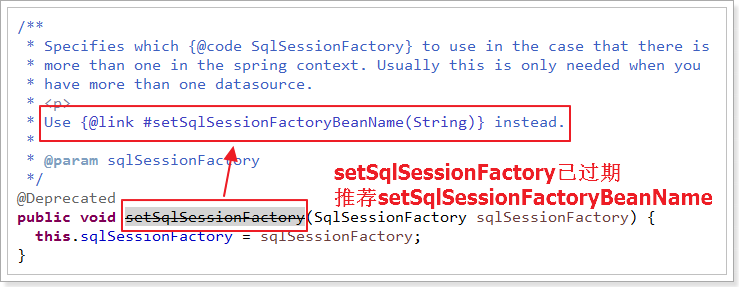
在mybatis-spring的整合包中，提供了MapperScannerConfigurer接口扫描类：



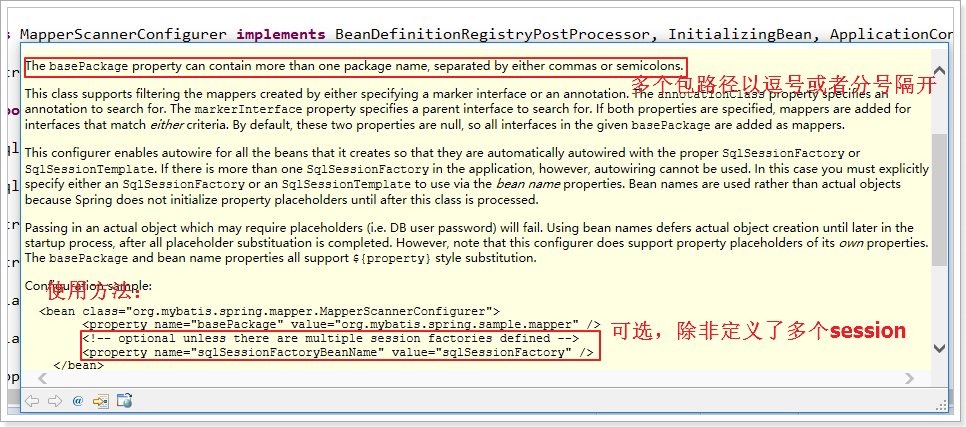
类中有以下属性



需要配置basePackage以及sqlSessionFactory属性，但是setSqlSessionFactory方法已过期，推荐使用setSQLSessionFactoryBeanName



查看MapperScannerConfigurer的注释：



如果有多个sqlSessionFactory时，以逗号或者分号隔开，如果只有一个sqlSessionFactory时，可以省略sqlSessionFactoryBeanName的配置。

而我们只有一个sqlSessionFactory，所以，只需要在applicationContext-mybatis.xml中配置：

<!-- mapper接口的包扫描 -->

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

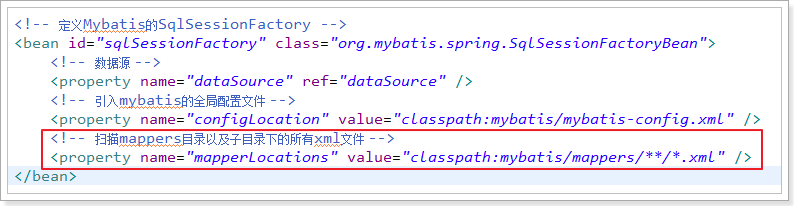
<property name=*"basePackage"* value=*"cn.itcast.usermanage.mapper"* />

</bean>

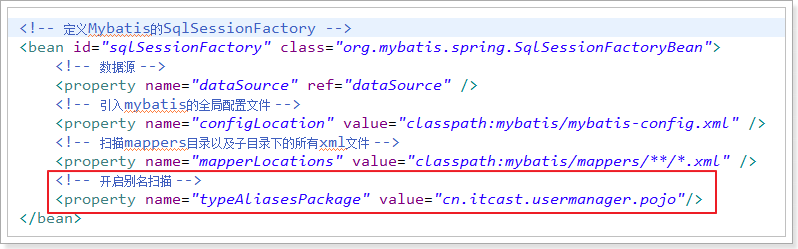
### Mapper.xml交给spring

解决mybatis的resource配置方式，造成的麻烦（每次都要配置）

解决mybatis的package包扫描，造成的配置和java耦合。



### 别名扫描交给spring



## 整合的终极配置

### applicationContext-mybatis.xml

|  |
| --- |
| <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"*>  <!-- 构建sqlSessionFactory -->  <bean id=*"sqlSessionFactory"* class=*"org.mybatis.spring.SqlSessionFactoryBean"*>  <!-- 数据源，必须 -->  <property name=*"dataSource"* ref=*"dataSource"* />  <!-- mybatis的全局配置文件 -->  <property name=*"configLocation"* value=*"classpath:mybatis/mybatis-config.xml"*></property>  <!-- 引入mybatis映射文件 *SqlSessionFactoryBean 里面的字段* -->  <property name=*"mapperLocations"* value=*"classpath:mybatis/mappers/\*\*/\*.xml"* />  <!-- 别名扫描 *SqlSessionFactoryBean里面的字段* -->  <property name=*"typeAliasesPackage"* value=*"cn.itcast.usermanage.pojo"* />  </bean>  <!-- mapper接口的扫描 -->  <bean class=*"org.mybatis.spring.mapper.MapperScannerConfigurer"*>  <property name=*"basePackage"* value=*"cn.itcast.usermanage.mapper"* />  </bean>  </beans> |

### mybatis-config.xml

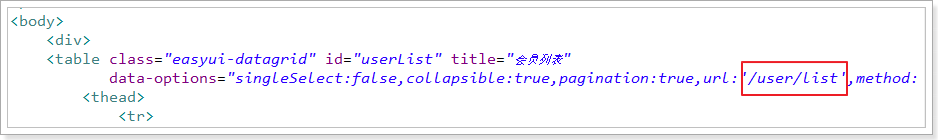
|  |
| --- |
| <?xml version=*"1.0"* encoding=*"UTF-8"* ?>  <!DOCTYPE configuration  PUBLIC "-//mybatis.org//DTD Config 3.0//EN"  "http://mybatis.org/dtd/mybatis-3-config.dtd">  <configuration>  <settings>  <!-- 开启驼峰匹配，经典的数据库列名（多个单词时，以下划线连接）到经典java属性名（多个单词时，以驼峰命名） -->  <setting name=*"mapUnderscoreToCamelCase"* value=*"true"* />  </settings>  </configuration> |

# 查询用户信息

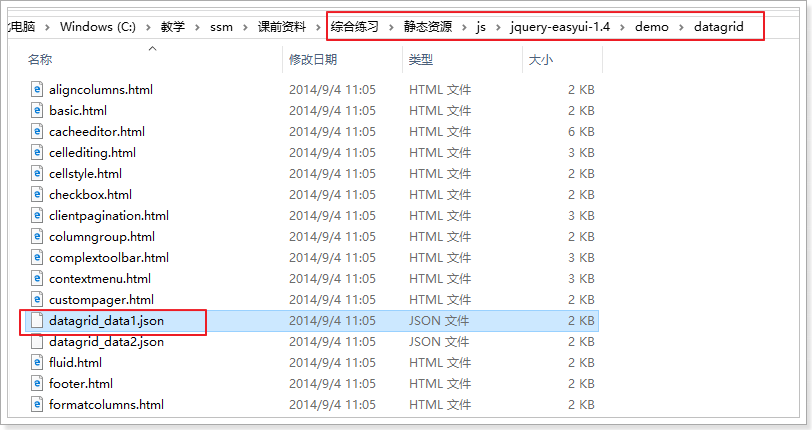
要查询用户信息，首先搞清楚以下几个问题：

1. 请求路径是什么
2. 请求路径对应的Controller方法的返回值，应该是什么样的
3. Controller方法的参数有哪些？

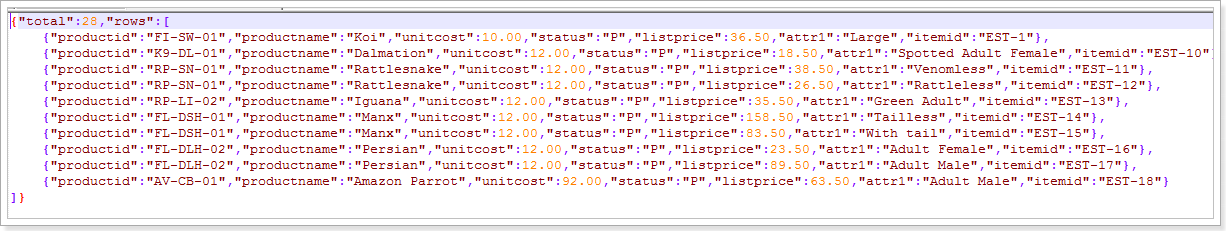
Jsp页面获取数据的请求路径，参见jsp页面中的url：



Controller方法的返回值，参考easyUI的官方案例：

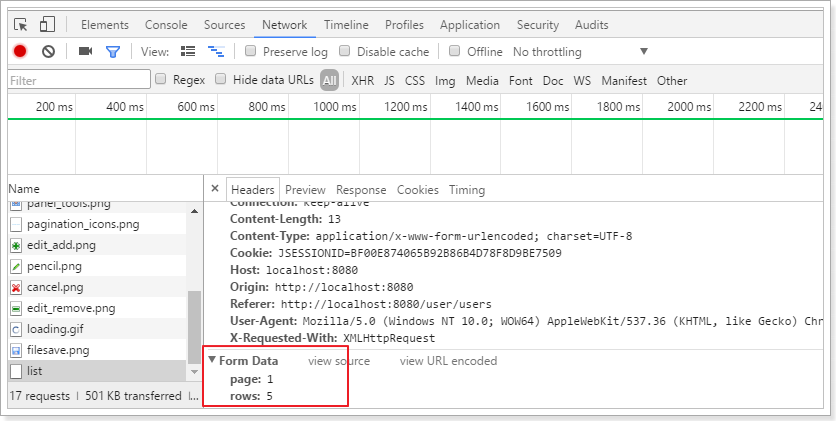


Json文件的内容：



由此可见Controller方法的返回值是一个json数据，数据中应包含两个字段：{“total”:”总条数”,”rows”:”显示在表格中的具体数据”}

请求参数，参考提交表单信息如下：



这就决定了Controller方法：

请求路径是：/user/list

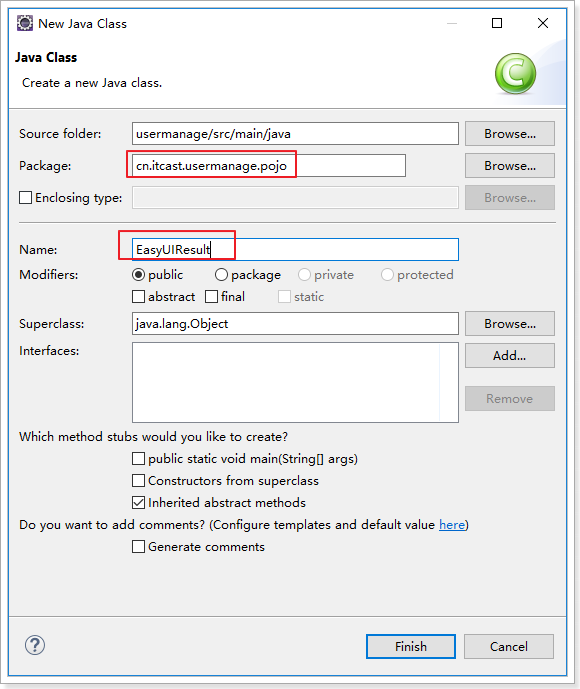
返回值：必须包含total，rows字段

方法的参数：@RequestParam(“page”) Integer pageNum, @RequestParam(“rows”) Integer pageSize

## 定义EasyUIResult

为了提高代码的可复用性，可以把easyUI要求的返回值包装成一个通用的pojo对象。

作用：定义easyui中的分页信息的bean对象，包含total和rows字段，为了保证EasyUIResult对象可复用性，rows属性使用了占位符或者泛型：



**package** cn.itcast.usermanage.pojo;

**import** java.util.List;

**public** **class** EasyUIResult {

**private** Long total;

**private** List<?> rows;

**public** Long getTotal() {

**return** total;

}

**public** **void** setTotal(Long total) {

**this**.total = total;

}

**public** List<?> getRows() {

**return** rows;

}

**public** **void** setRows(List<?> rows) {

**this**.rows = rows;

}

}

## 编写UserController

@ResponseBody

@RequestMapping("list")

**public** EasyUIResult queryUsersByPage(@RequestParam("page")Integer pageNum, @RequestParam("rows")Integer pageSize){

**return** **this**.userService.queryEasyUIResult(pageNum, pageSize);

}

## 编写userservice

接口：

**public** **interface** UserService {

/\*\*

\* 分页获取用户信息

\* **@param** pageNum

\* **@param** pageSize

\* **@return**

\*/

EasyUIResult queryEasyUIResult(Integer pageNum, Integer pageSize);

}

实现类

@Service

**public** **class** UserServiceImpl **implements** UserService {

@Autowired

**private** UserMapper userMapper;

@Override

**public** EasyUIResult queryEasyUIResult(Integer pageNum, Integer pageSize) {

// 第一个参数是从那条开始，第二个参数是查询多少条

List<User> userList = **this**.userMapper.queryUsersByPage((pageNum-1)\*pageSize, pageSize);

EasyUIResult easyUIResult = **new** EasyUIResult();

easyUIResult.setTotal(15l);

easyUIResult.setRows(userList);

**return** easyUIResult;

}

}

## 编写userMapper

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

/\*\*

\* 根据id获取User信息

\* **@return**

\*/

**public** User queryUserById(Long id);

/\*\*

\* 分页查询用户信息

\* **@param** start

\* **@param** pageSize

\*/

**public** List<User> queryUsersByPage(@Param("start")Integer start, @Param("pageSize")Integer pageSize);

}

## 编写UserMapper.xml

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

<!DOCTYPE mapper

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

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

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

<select id=*"queryUserById"* resultType=*"User"*>

select \* from tb\_user where id = #{id}

</select>

<select id=*"queryUsersByPage"* resultType=*"User"*>

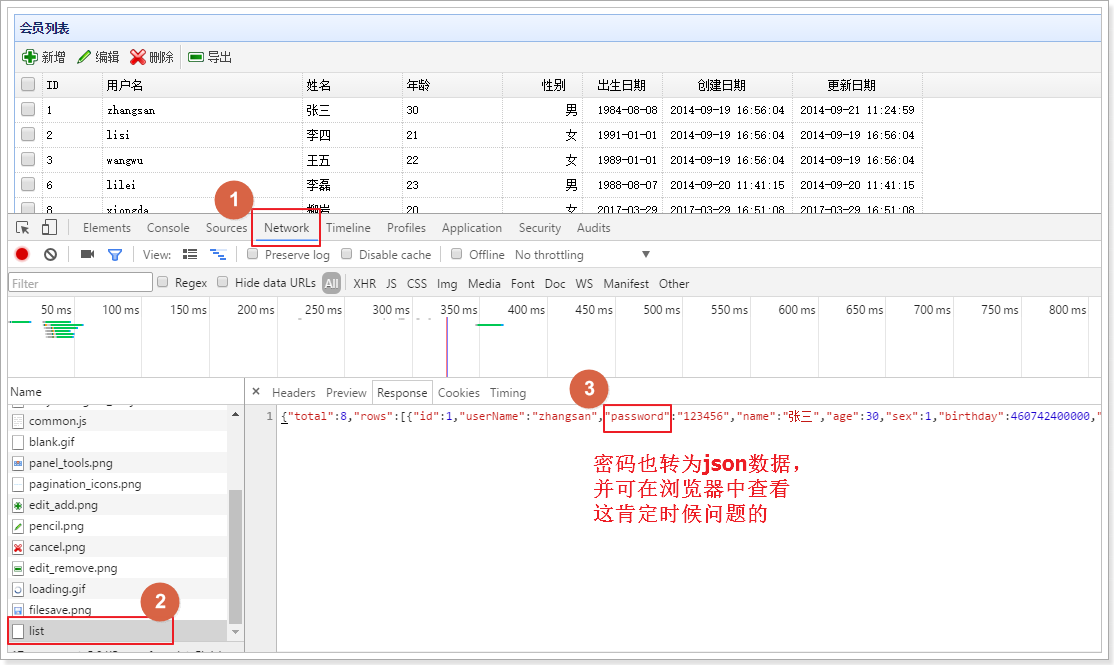
select \* from tb\_user limit #{start},#{pageSize}

</select>

</mapper>

## 解决密码问题

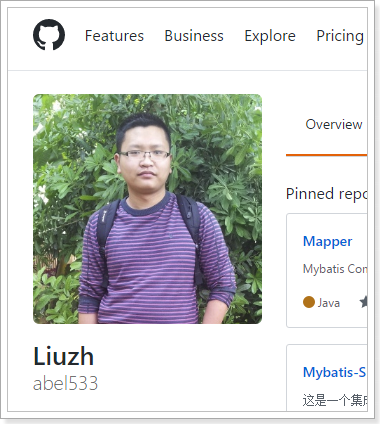
问题：



解决方案：在实体类（user）中的对应属性上，添加@JsonIgnore注解



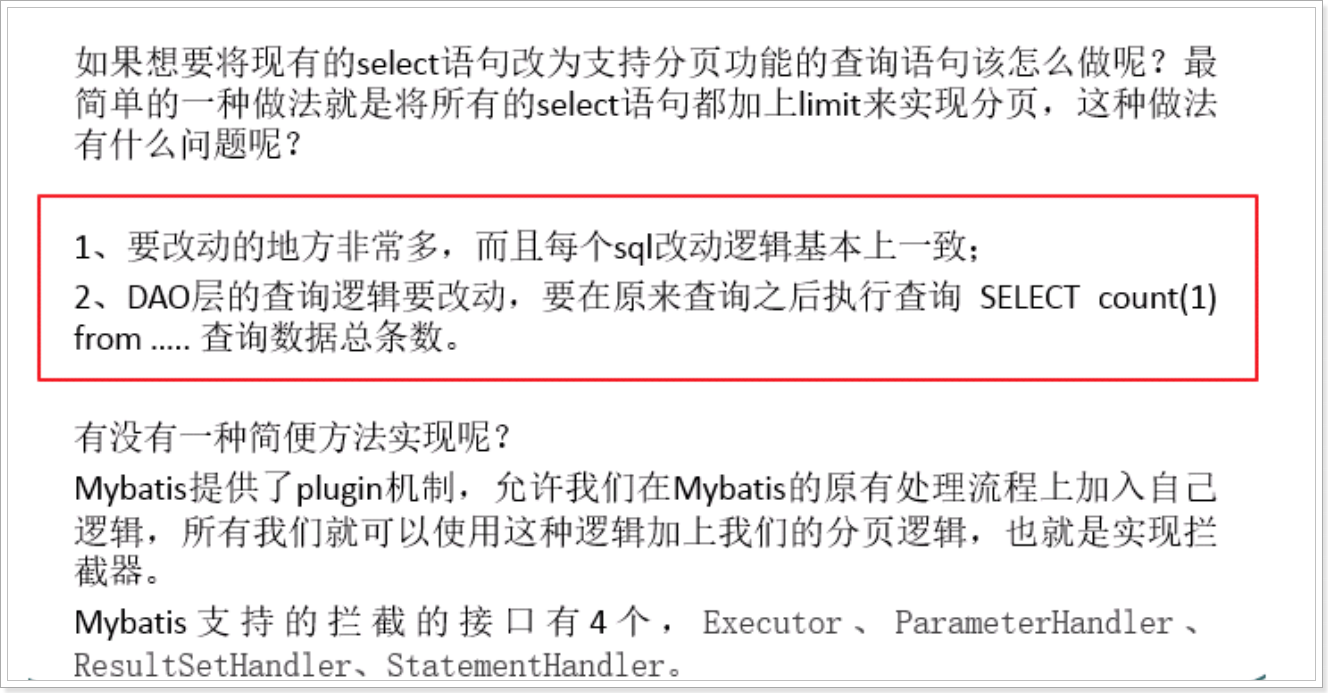
# 分页插件

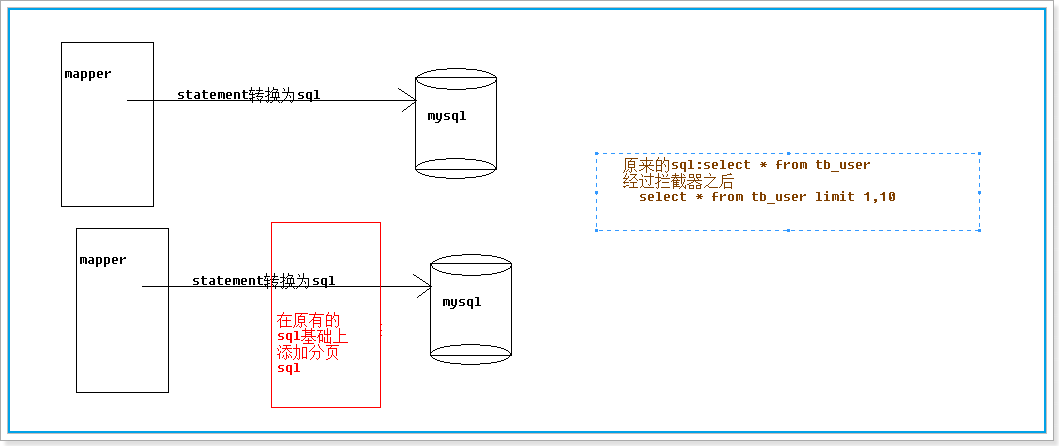


思考：我们前面的分页实现存在怎样的问题？

1. 我们需要给每个SQL添加分页的逻辑，还要保留不分页的SQL，重复！
2. 不同数据库分页方式不同，写死后不利于以后的拓展！
3. 需要手动查询总条数信息

## 原理





## 引入依赖

Pom.xml中新增依赖：

|  |
| --- |
| <dependency>  <groupId>com.github.pagehelper</groupId>  <artifactId>pagehelper</artifactId>  <version>3.7.5</version>  </dependency>  <dependency>  <groupId>com.github.jsqlparser</groupId>  <artifactId>jsqlparser</artifactId>  <version>0.9.1</version>  </dependency> |

## Mybatis-config.xml配置插件

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

<!DOCTYPE configuration

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

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

<configuration>

<settings>

<!-- 行为参数，name:参数名，value：参数值，默认为false，true：开启驼峰匹配，即从经典的数据库列名到经典的java属性名 -->

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

</settings>

<plugins>

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

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

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

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

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

</plugin>

</plugins>

</configuration>

## 使用分页插件改造程序

查询所有用户信息，分页交给分页插件来完成。

UserMapper接口添加查询所有用户的方法：

/\*\*

\* 查询所有用户

\* **@return**

\*/

**public** List<User> queryUserAll();

UserMapper配置添加查询所有用户的Statement：

<select id=*"queryUserAll"* resultType=*"User"*>

select \* from tb\_user

</select>

改造UserServiceImpl，修改获取分页数据的Service方法：

@Override

**public** EasyUIResult queryEasyUIResult(Integer pageNum, Integer pageSize) {

// 在查询方法调用之前，调用分页插件的静态方法，中间最好不要隔任何代码

PageHelper.*startPage*(pageNum, pageSize);

// 第一个参数是从那条开始，第二个参数是查询多少条

// List<User> userList = this.userMapper.queryUsersByPage((pageNum-1)\*pageSize, pageSize);

List<User> userList = **this**.userMapper.queryUserAll();

// 初始化pageInfo对象，所有的分页参数都可以在该对象中获取

PageInfo<User> pageInfo = **new** PageInfo<>(userList);

EasyUIResult easyUIResult = **new** EasyUIResult();

easyUIResult.setTotal(pageInfo.getTotal());

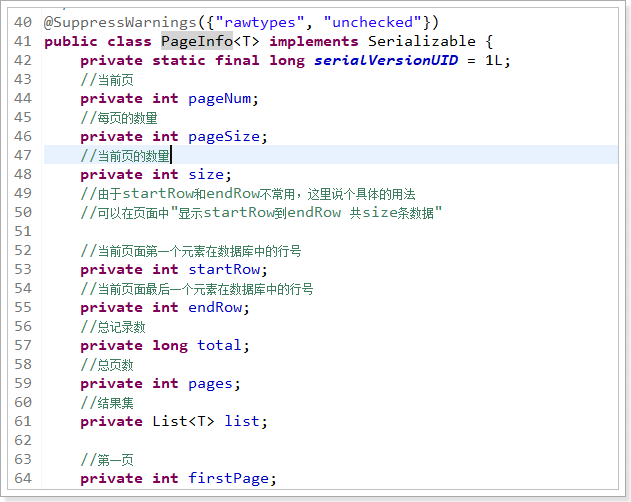
easyUIResult.setRows(pageInfo.getList());

**return** easyUIResult;

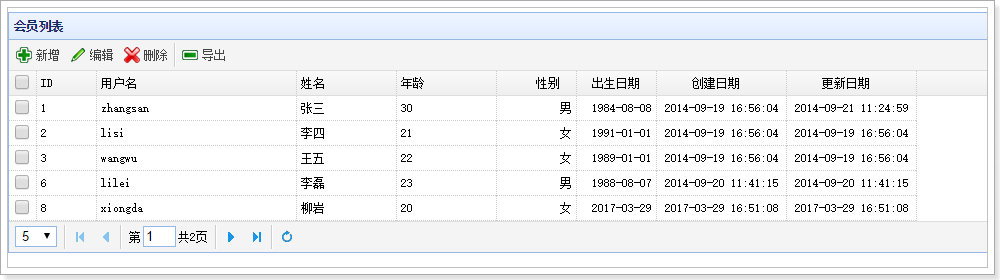
}

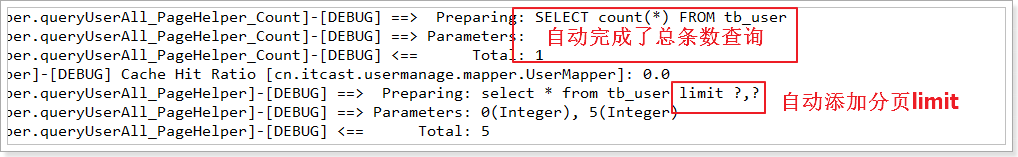
你需要进行分页的Mybatis方法前调用PageHelper.startPage静态方法即可，紧跟在这个方法后的第一个**Mybatis查询方法**会被进行分页。

用PageInfo对结果进行包装，并获取分页信息，参见pageInfo的源码：



## 结果

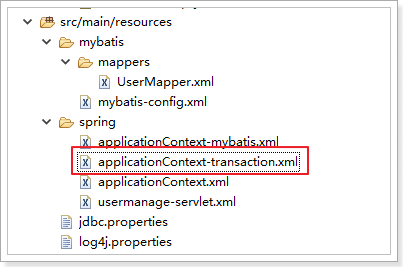




# Service层整合事务管理

为了保证数据的一致性及原子性，Service的增删改操作应该添加事务。

## 事务的配置文件（application-transaction.xml）



|  |
| --- |
| <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"*>    <!-- 定义事务管理器 -->  <bean id=*"transactionManager"*  class=*"org.springframework.jdbc.datasource.DataSourceTransactionManager"*>  <property name=*"dataSource"* ref=*"dataSource"* />  </bean>  <!-- 定义事务策略 -->  <tx:advice id=*"txAdvice"* transaction-manager=*"transactionManager"*>  <tx:attributes>  <!--所有以query开头的方法都是只读的 -->  <tx:method name=*"query\*"* read-only=*"true"* />  <!--其他方法使用默认事务策略 -->  <tx:method name=*"\*"* />  </tx:attributes>  </tx:advice>  <aop:config>  <!--pointcut元素定义一个切入点，execution中的第一个星号 用以匹配方法的返回类型，  这里星号表明匹配所有返回类型。 com.abc.dao.\*.\*(..)表明匹配cn.itcast.usermanage.service包下的所有类的所有方法 -->  <aop:pointcut id=*"myPointcut"* expression=*"execution(\* cn.itcast.usermanage.service.\*.\*(..))"* />  <!--将定义好的事务处理策略应用到上述的切入点 -->  <aop:advisor advice-ref=*"txAdvice"* pointcut-ref=*"myPointcut"* />  </aop:config>    </beans> |

接下来咱们来测试事务：

1. 在Mapper接口中添加一个新增用户的方法
2. 在Mapper映射文件中配置对应的Statement
3. 在Service接口中添加一个新增用户的接口方法，该方法用来新增2个用户信息，要成功都成功，要失败都失败
4. 在Service接口的实现类中实现该方法

## Mapper接口

在UserMapper接口中新增添加用户方法：

/\*\*

\* 新增用户

\* **@param** user

\* **@return**

\* **@throws** Exception

\*/

**public** **int** addUser(User user) **throws** Exception;

## Mapper配置

在UserMappper.xml中配置对应的statement：

<insert id=*"addUser"* useGeneratedKeys=*"true"* keyColumn=*"id"*

keyProperty=*"id"*>

INSERT INTO tb\_user (

user\_name,

password,

name,

age,

sex,

birthday,

created,

updated

)

VALUES

(

#{userName},

#{password},

#{name},

#{age},

#{sex},

#{birthday},

now(),

now()

);

</insert>

## service接口

在UserService接口中添加方法：

/\*\*

\* 测试事务

\* **@param** user1

\* **@param** user2

\*/

**public** **void** addUsers(User user1, User user2);

## service实现类

在UserServiceImpl实现类中实现接口方法：

@Override

**public** **void** addUsers(User user1, User user2) {

**this**.userMapper.addUser(user1);

// 制造异常

**int** i=1/0;

**this**.userMapper.addUser(user2);

}

## 测试

给UserService添加测试用例UserServiceTest：

**public** **class** UserServiceTest {

**private** UserService userService;

@Before

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

ApplicationContext context = **new** ClassPathXmlApplicationContext("spring/applicationContext.xml",

"spring/applicationContext-mybatis.xml", "spring/applicationContext-transaction.xml");

**this**.userService = context.getBean(UserService.**class**);

}

@Test

**public** **void** test() {

User user1 = **new** User();

user1.setUserName("dengdeng3");

user1.setPassword("123456");

User user2 = **new** User();

user2.setUserName("dengdeng4");

user2.setPassword("123456");

**this**.userService.addUsers(user1, user2);

}

}

# 新增用户

## 通用的页面跳转

之前进行页面跳转的时候，如果有多个页面，就需要有多个方法。

借助于requetMapping(占位符的方式，传递一些数据，达到页面跳转的目的)

/user/to/user-list

可以取出users或者user-add这个内容， 作为视图的名字。

/web-inf/views/ users.jsp 或者/web-inf/views/user-add.jsp相应的请求路径就是/user/page/users或者/user/page/user-add

UserController中，新增通用的页面跳转方法：

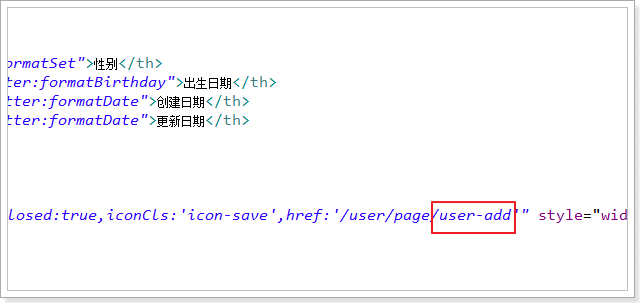
@RequestMapping(value="/page/{pageName}")

**public** String toUserAdd(@PathVariable("pageName") String pageName){

**return** pageName;

}

修改新增页面的跳转路径：



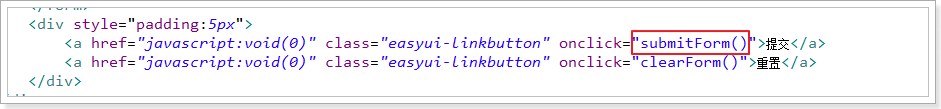
## 实现新增功能

### 解析及调整页面

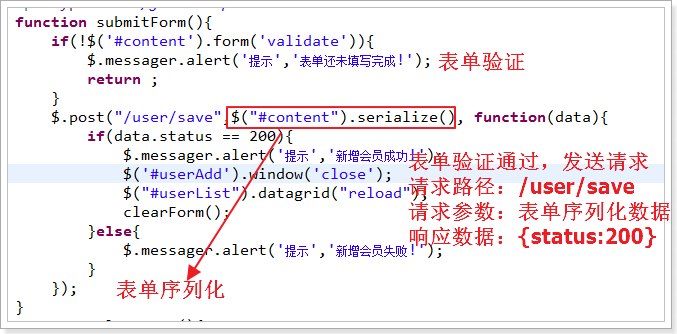
users.jsp中toolbar变量的定义，找到新增按钮：



user-add.jsp中，提交事件：



触发submitForm()函数：

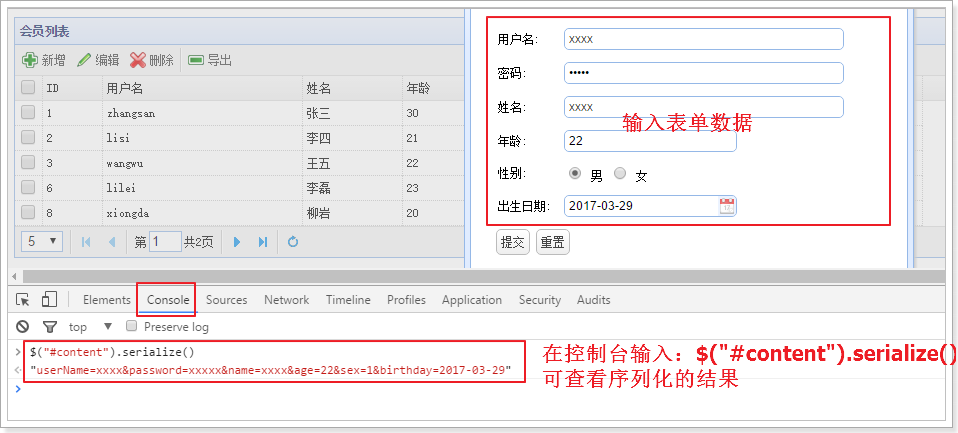


submitForm()函数，首先做表单验证；验证通过发送ajax请求，请求路径为：/user/save，请求参数：form表单的序列化为字符串，响应数据：{status：200}

表单验证的依据：



Form表单序列化结果测试：



### 编写UserController

根据请求路径、响应数据以及请求参数，编写Controller方法，并对异常做处理：

/\*\*

\* 请求路径：/save

\* 方法返回值：{status：200}

\* 参数：User对象

\*/

@RequestMapping(value="save")

@ResponseBody

**public** Map<String, Object> addUser(User user){

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

**try** {

// 调用userService的新增方法

Boolean b = **this**.userService.addUser(user);

**if** (b) {

map.put("status", "200");

} **else** {

map.put("status", "500");

}

} **catch** (Exception e) {

map.put("status", "500");

e.printStackTrace();

}

**return** map;

}

### 编写UserService及其实现

UserService：

/\*\*

\* 新增用户信息

\* **@param** user

\* **@return**

\* **@throws** Exception

\*/

Boolean addUser(User user) **throws** Exception;

UserServiceImpl：

@Override

**public** Boolean addUser(User user) **throws** Exception {

**int** count = **this**.userMapper.addUser(user);

**if**(count>0){

**return** **true**;

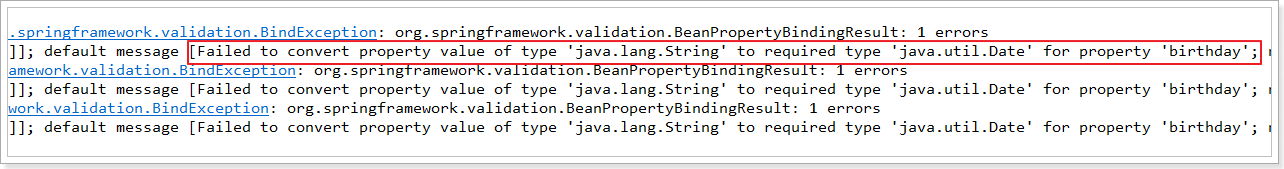
}

**return** **false**;

}

## 日期的处理

新增用户，程序写完添加用户，无法保存。控制台报异常：类型转换错误



解决方案：



## 接收日期类型参数

在日常工作中，经常要根据日期为条件查询数据，springmvc怎么直接接受date类型的参数呢？

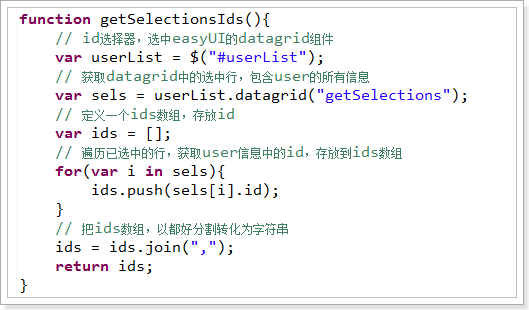


# 删除用户

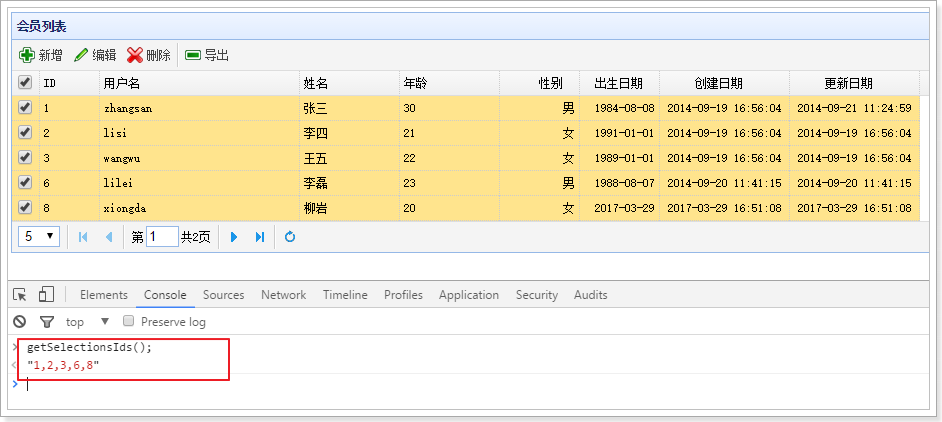
## 视图分析



上述代码中，getSelectionsIds()分析：

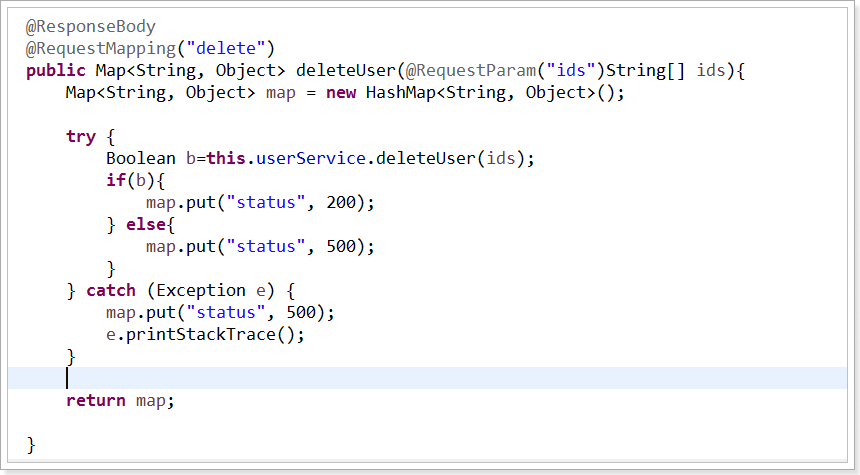


控制台：

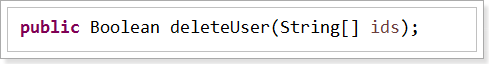


## UserController

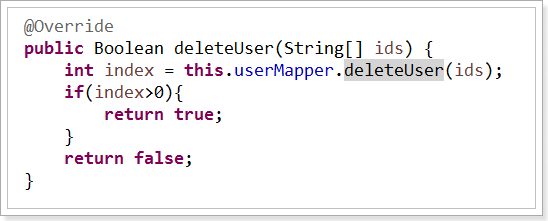
请求路径：/delete；请求参数：以逗号分割的ids；响应数据：{status：200}



## UserService



实现类：



## UserMapper接口



## UserMapper映射文件

<delete id=*"deleteUser"*>

delete from tb\_user where id in

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

#{id}

</foreach>

</delete>

# 导出excel

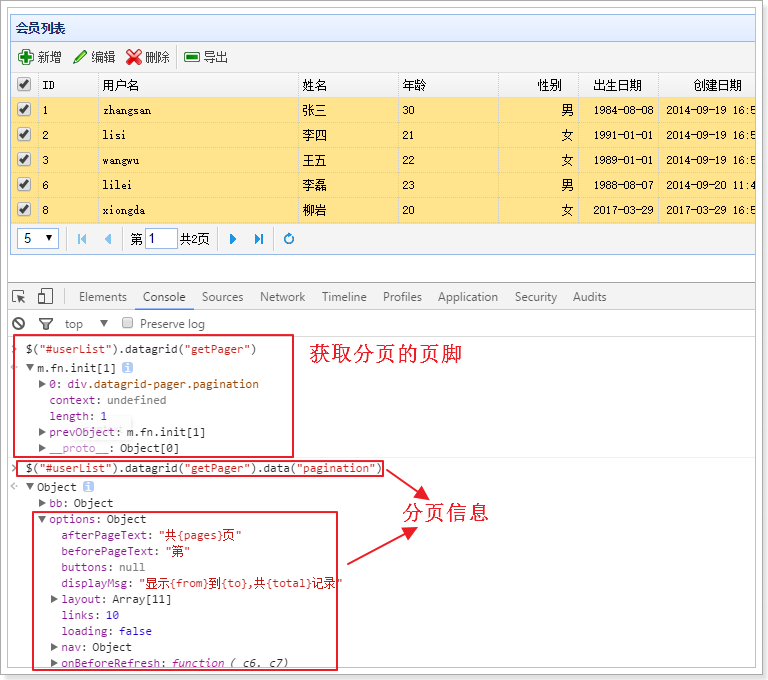
## 页面分析

Users.jsp页面的js代码：



注意：导出功能的js可能有浏览器兼容性问题

获取分页信息的js分析：

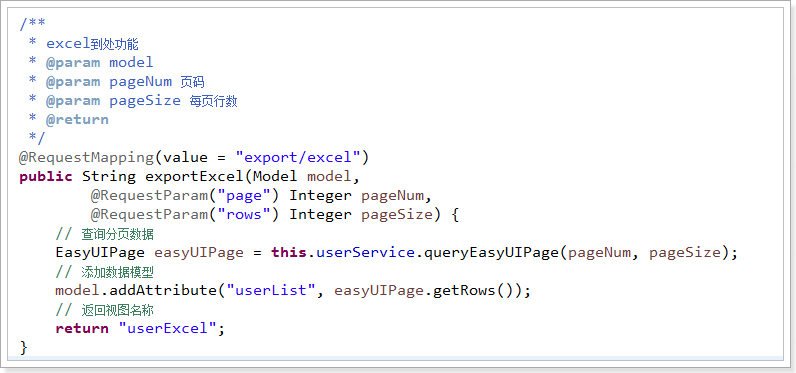


## 引入poi依赖

在pom.xml中引入：

|  |
| --- |
| <dependency>  <groupId>org.apache.poi</groupId>  <artifactId>poi</artifactId>  <version>3.10.1</version>  </dependency> |

## UserController的逻辑



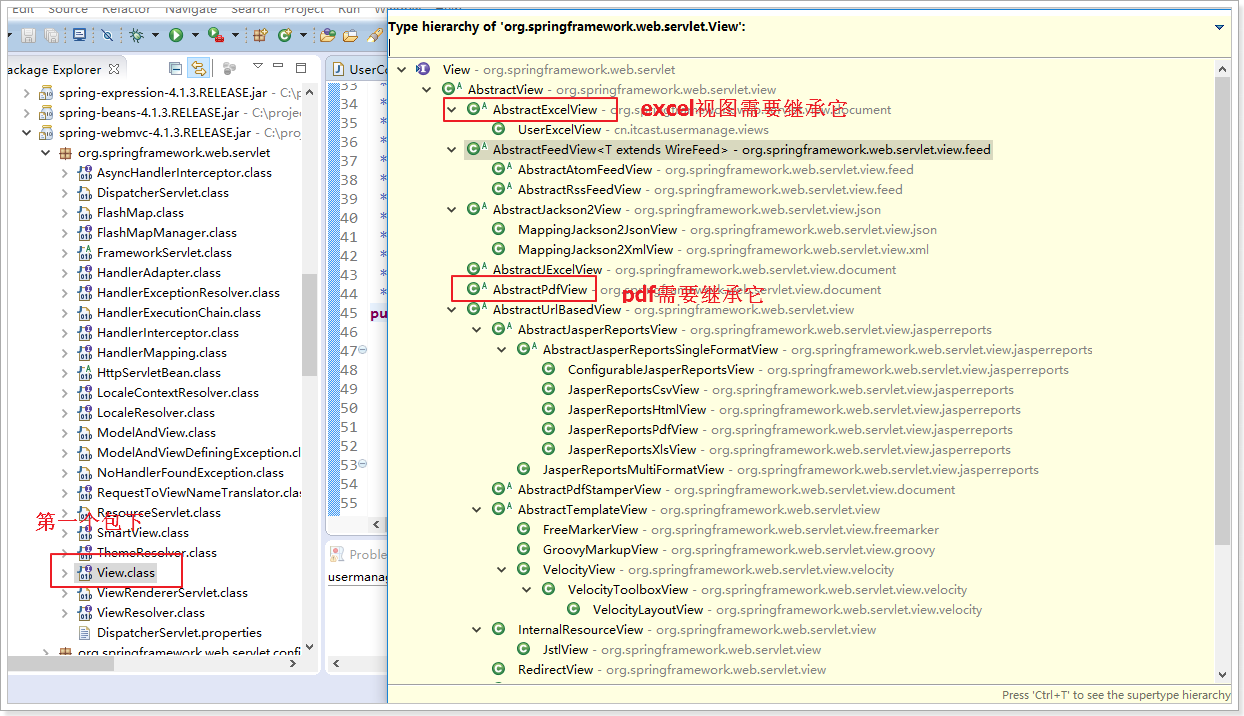
## excel视图实现

既然excel也是一种视图，那么是所有的excel都能作为视图吗？

答案肯定是否定的，必须按照springmvc的规则来做。

### 原理

springmvc提供了View接口，默认有一些抽象的实现。比如：excel的抽象实现，pdf的抽象实现

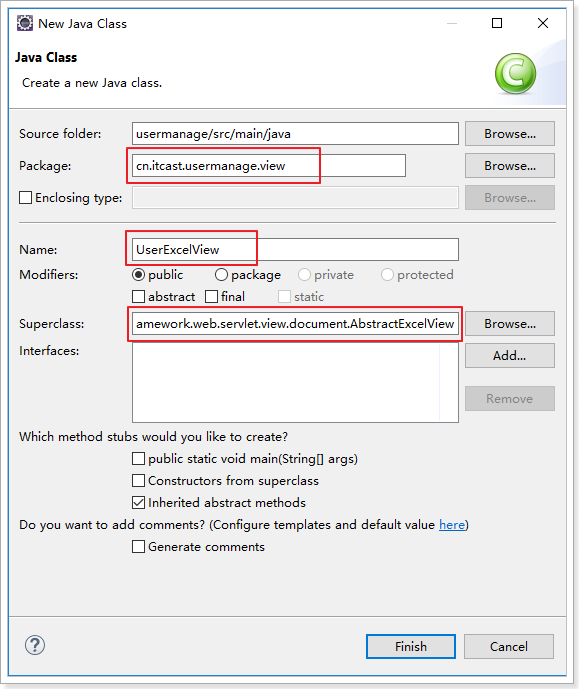


导出excel分3个步骤：

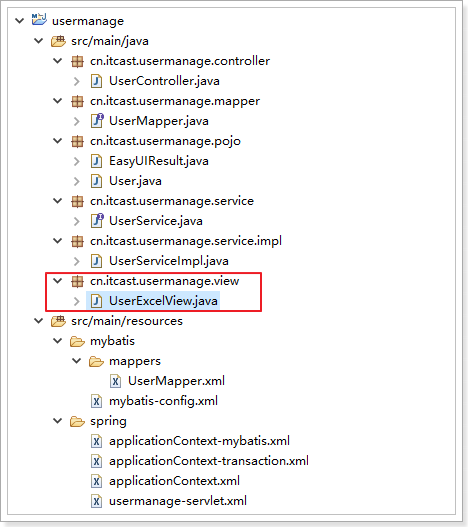
1. 自定义一个视图（UserExcel）并继承AbstractExcelView抽象类
2. 把自定义视图在springmvc的配置文件（usermanage-servlet.xml）中注册
3. 定义视图解析器，解析excel视图

其他代码包括Controller及Service的编写，不受影响。

### 自定义excel 视图



目录结构：



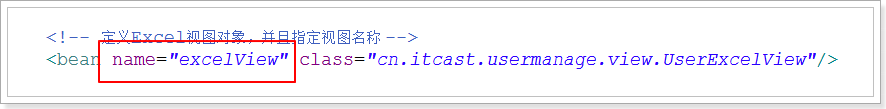
继承excel的抽象类，重写buildExcelDocument：

|  |
| --- |
| **package** cn.itcast.usermanage.view;  **import** java.util.List;  **import** java.util.Map;  **import** javax.servlet.http.HttpServletRequest;  **import** javax.servlet.http.HttpServletResponse;  **import** org.apache.poi.hssf.usermodel.HSSFRow;  **import** org.apache.poi.hssf.usermodel.HSSFSheet;  **import** org.apache.poi.hssf.usermodel.HSSFWorkbook;  **import** org.joda.time.DateTime;  **import** org.springframework.web.servlet.view.document.AbstractExcelView;  **import** cn.itcast.usermanage.pojo.User;  **import** cn.itcast.usermanage.utils.Constants;  **public** **class** UserExcelView **extends** AbstractExcelView {  @Override  **protected** **void** buildExcelDocument(Map<String, Object> model, HSSFWorkbook workbook, HttpServletRequest request, HttpServletResponse response) **throws** Exception {  // 从model对象中获取userList  @SuppressWarnings("unchecked")  List<User> userList = (List<User>) model.get("userList");  // 创建Excel的sheet  HSSFSheet sheet = workbook.createSheet("会员列表");  // 创建标题行  HSSFRow header = sheet.createRow(0);  header.createCell(0).setCellValue("ID");  header.createCell(1).setCellValue("用户名");  header.createCell(2).setCellValue("姓名");  header.createCell(3).setCellValue("年龄");  header.createCell(4).setCellValue("性别");  header.createCell(5).setCellValue("出生日期");  header.createCell(6).setCellValue("创建时间");  header.createCell(7).setCellValue("更新时间");    // 填充数据  **int** rowNum = 1;  **for** (User user : userList) {  HSSFRow row = sheet.createRow(rowNum);  row.createCell(0).setCellValue(user.getId());  row.createCell(1).setCellValue(user.getUserName());  row.createCell(2).setCellValue(user.getName());  row.createCell(3).setCellValue(user.getAge());  String sexStr;  **if** (user.getSex() == 1) {  sexStr = "男";  } **else** **if** (user.getSex() == 2) {  sexStr = "女";  } **else** {  sexStr = "未知";  }  row.createCell(4).setCellValue(sexStr);  row.createCell(5).setCellValue(**new** DateTime(user.getBirthday()).toString(Constants.***DATE***));  row.createCell(6).setCellValue(**new** DateTime(user.getCreated()).toString(Constants.***DATE\_TIME***));  row.createCell(7).setCellValue(**new** DateTime(user.getUpdated()).toString(Constants.***DATE\_TIME***));  rowNum++;  }  // 设置相应头信息，以附件形式下载并且指定文件名  response.setHeader("Content-Disposition", "attachment;filename=" + **new** String("会员列表.xls".getBytes(),"ISO-8859-1"));  }  } |

### 注册自定义视图

将自定义视图注册到springmvc的容器，并指定视图名称。

Usermanage-servelet.xml配置文件：

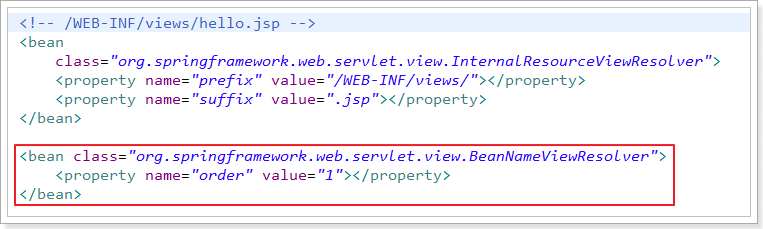


但是仅仅这样还不够，因为此时我们的视图解析器是InternalResourceResolver，依然会被解析成jsp视图。所以我们还需要定义一个新的视图解析器

### 定义视图解析器

查看ViewResolver接口的所有实现，发现了熟悉的身影BeanNameViewResolver（把Bean的name作为视图解析器依据），注册该视图解析器。

Usermanage-servelet.xml配置文件：



注意：

如果不配置视图解析器的顺序（order属性），运行tomcat测试，点击导出按钮，页面依然跳转向userExcel.jsp页面，由于没有该jsp页面，所以会报404

原因：

在SpringMVC中，如果定义多个视图解析器，需要指定每个视图解析的顺序（即order属性）。如果不指定顺序，DispatchServlet会按照配置的顺序调用试图解析器。而内部资源的视图解析器，不管有没有相应的视图，总有返回值；其他的视图解析器，只有视图存在的情况下，才有返回值。

所以默认顺序下，执行到内部资源视图解析器时，直接返回userExcel.jsp视图，不再去找其他试图解析器进行解析。

解决：

如上图，配置order属性，并把value的值设置为1，或者把BeanNameViewResolver的配置放在InternalResourceViewResolver上面。

## 测试

