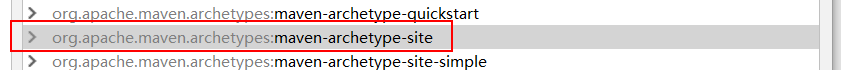
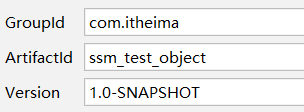
**ssm项目练习**

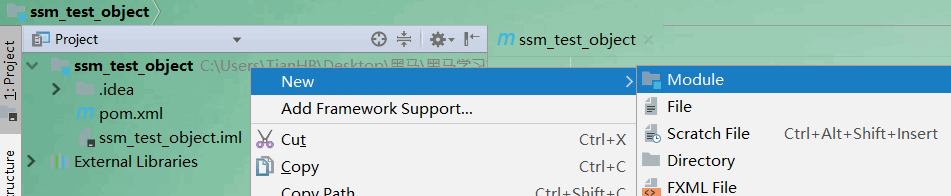
# 创建maven聚合工程,父工程

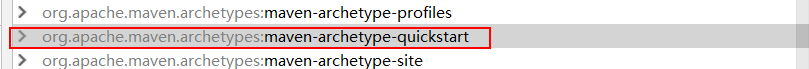


## 父工程名称



## 创建子工程 dao模块,service模块

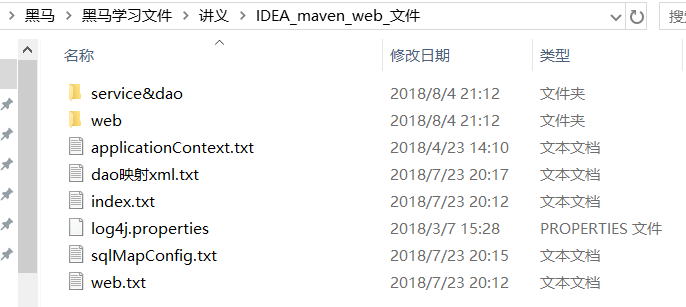




## 创建子工程 web模块



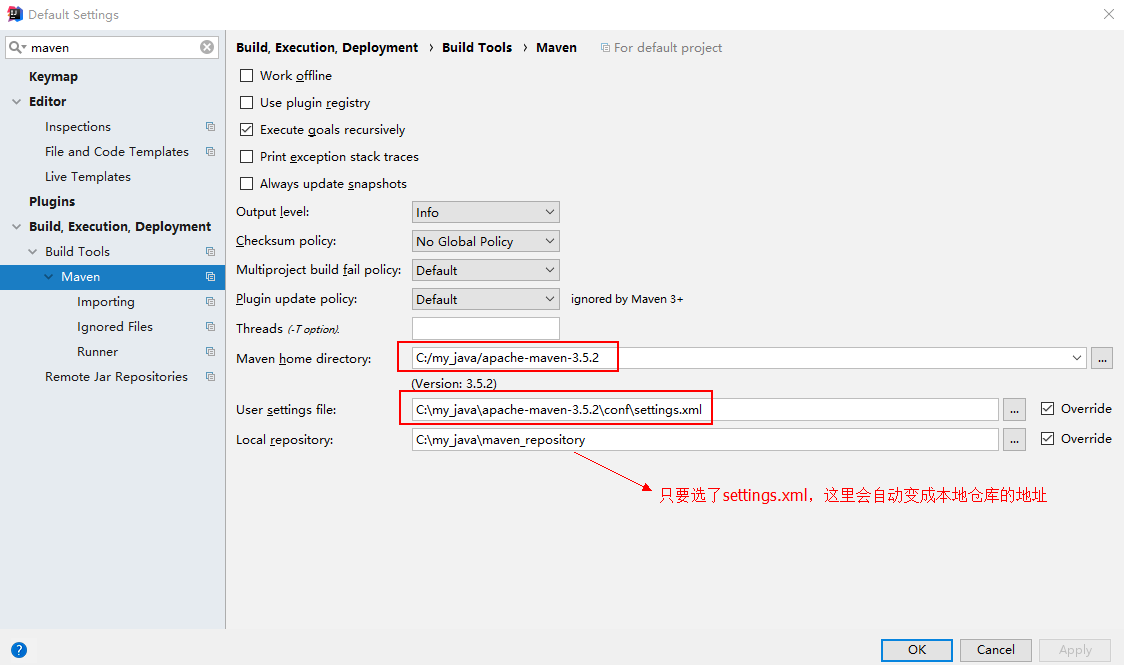
## 完善工程路径 – 或者自动生成



将准备好的文件夹路径,复制粘贴到工程

# 插播广告-Idea整合maven

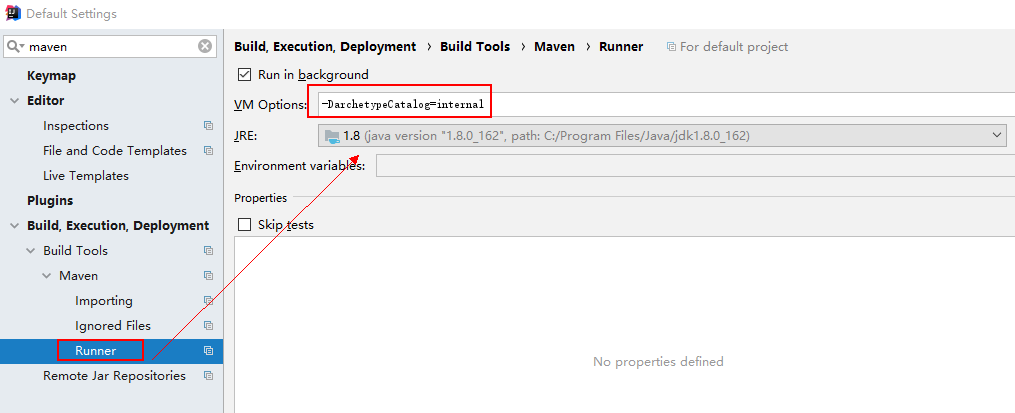
## 在主界面settings中搜索maven



## 然后设置运行参数

-DarchetypeCatalog=internal

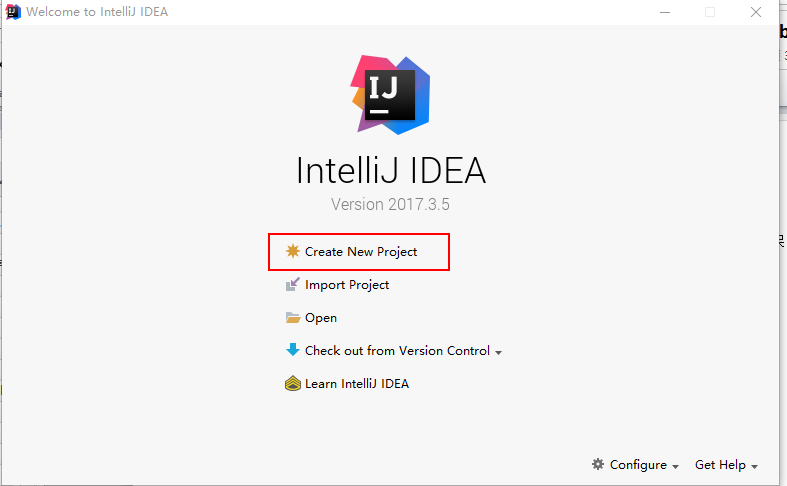
该参数的设置完之后，创建maven工程，我们只需要连接一次网络。后续会直接找本地资源。



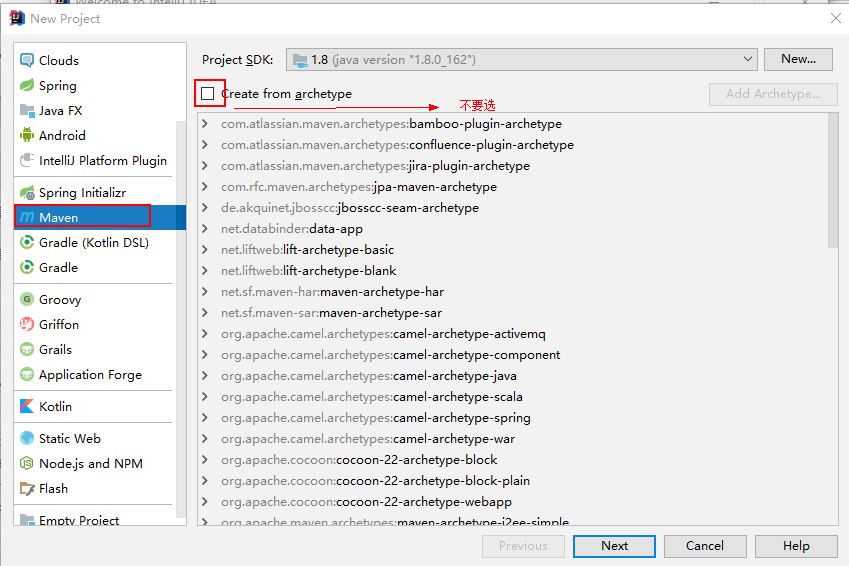
在idea中创建maven工程，如果是创建maven的java工程，推荐不使用骨架，如果是创建maven的web工程，用不用都行。

# 创建maven的java工程

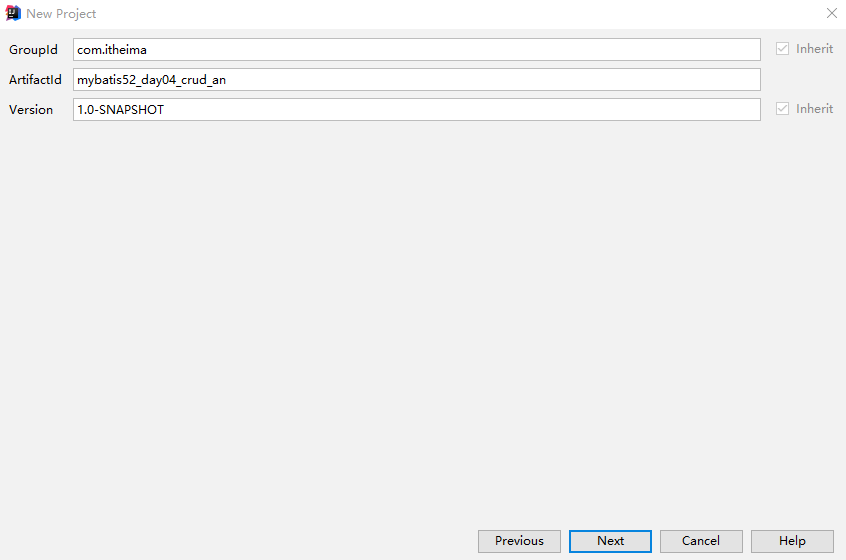
第一步



第二步

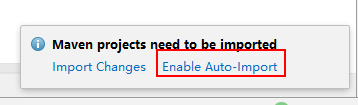


第三步



剩下步骤完全下一步，finish。

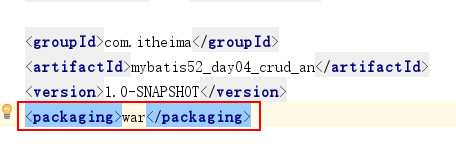
创建完成后右下角，选择运行自动导入。



选择此项表述今后maven工程自动更新。

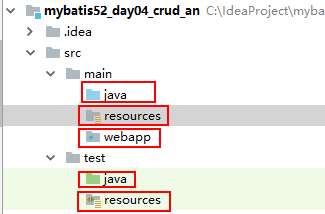
## Idea中java工程变web工程

手动在pom文件中写上packing=war



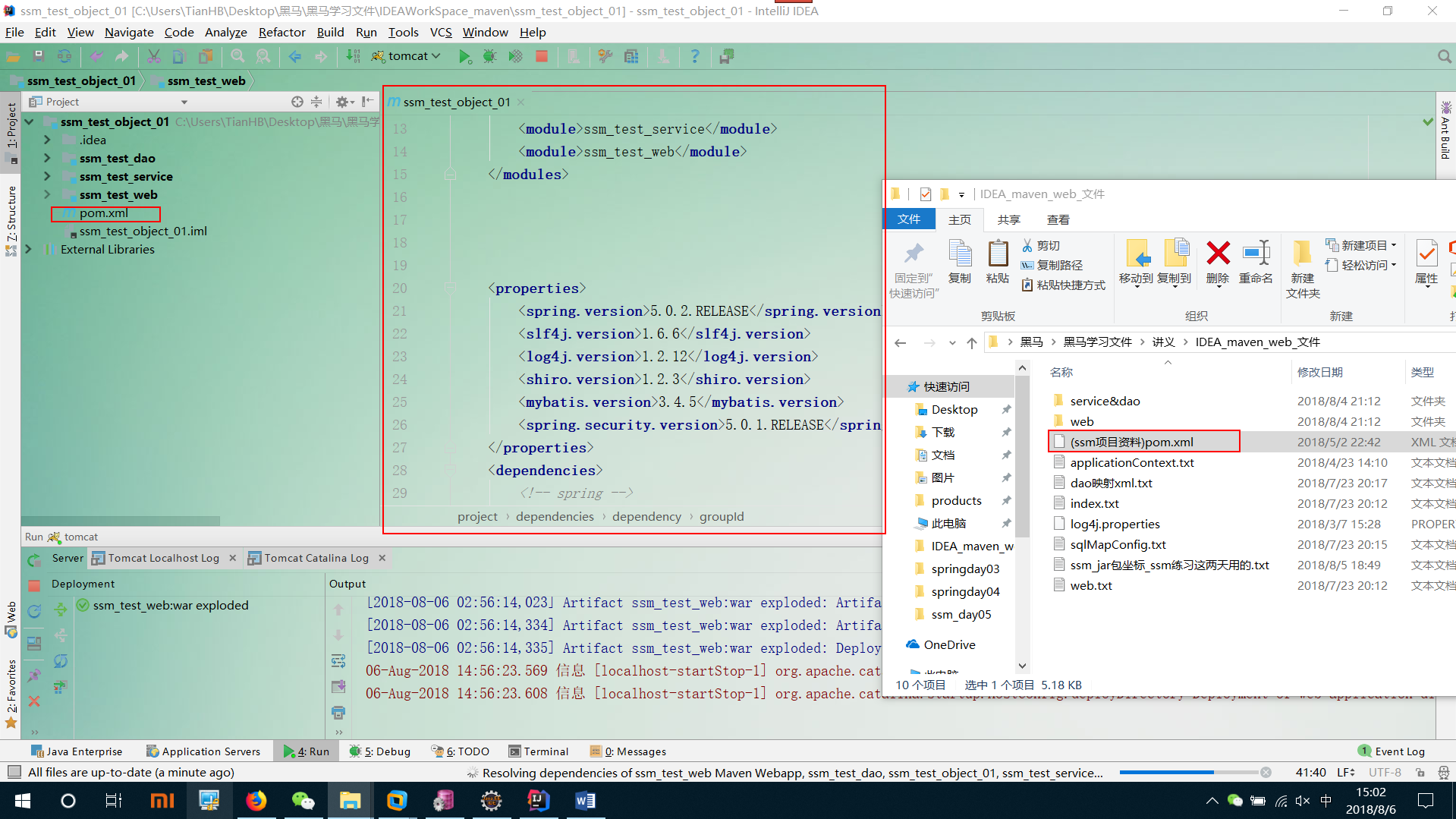
手动创建src/main/webapp文件夹

表中的web工程的目录结构



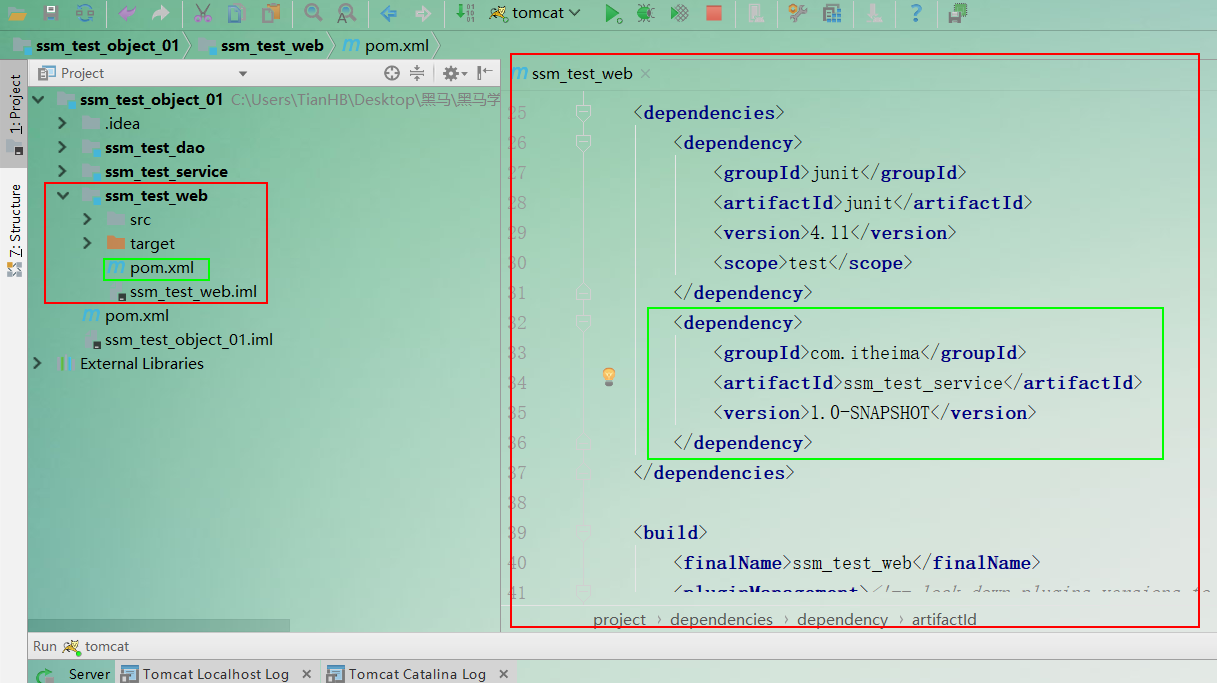
# 正式开始

## 在父工程导入pom.xml jar包坐标



### 建立子模块之间的依赖关系

#### service子模块入dao子模块坐标



#### web子模块导入service子模块坐标

同上

## 编写web.xml配置文件

### 配置中文编码过滤器

*<!--中文乱过滤器-->*<**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**>  
 <**init-param**>  
 <**param-name**>forceEncoding</**param-name**>  
 <**param-value**>true</**param-value**>  
 </**init-param**>  
</**filter**>  
<**filter-mapping**>  
 <**filter-name**>characterEncodingFilter</**filter-name**>  
 <**url-pattern**>/\*</**url-pattern**>  
</**filter-mapping**>

### 配置Springmvc前端控制器

*<!--springmvc前端控制器 servlet-->*<**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**>  
</**servlet**>  
<**servlet-mapping**>  
 <**servlet-name**>springmvc</**servlet-name**>  
 <**url-pattern**>/</**url-pattern**>  
</**servlet-mapping**>

/\* 拦截所有 jsp js png .css 真的全拦截 建议不使用 2. \*.action \*.do 拦截以do action

结尾的请求 肯定能使用 ERP 3. / 拦截所有 （不包括jsp) (包含.js .png.css) 强烈建议使用 前台 面向消费者 www.jd.com/search

/对静态资源放行 -->

### 配置spring核心监听器以及路径

*<!--spring配置文件核心监听器-->  
<!--spring配置文件路径-->*<**listener**>  
 <**listener-class**>org.springframework.web.context.ContextLoaderListener</**listener-class**>  
</**listener**>  
<**context-param**>  
 <**param-name**>contextConfigLocation</**param-name**>  
 <**param-value**>classpath:applicationContext.xml</**param-value**>  
</**context-param**>

### 配置spring Security安全框架过滤器链 注意 filter-name

*<!-- security安全框架的 过滤器链-->*<**filter**>  
 <**filter-name**>springSecurityFilterChain</**filter-name**>  
 <**filter-class**>org.springframework.web.filter.DelegatingFilterProxy</**filter-class**>  
</**filter**>  
<**filter-mapping**>  
 <**filter-name**>springSecurityFilterChain</**filter-name**>  
 <**url-pattern**>/\*</**url-pattern**>  
</**filter-mapping**>

### 配置spring监听器 将Request请求对象放入IOC容器中

*<!-- spring监听器,将每次Request请求放入IOC容器中-->*<**listener**>  
 <**listener-class**>org.springframework.web.context.request.RequestContextListener</**listener-class**>  
</**listener**>

## 编写springmvc.xml配置文件

### 开启组件扫描

*<!--开启组件扫描-->*<**context:component-scan base-package="com.itheima.controller"**></**context:component-scan**>

### 配置视图解析器

*<!--配置视图解析器-->*<**bean id="internalResourceViewResolver" class="org.springframework.web.servlet.view.InternalResourceViewResolver"**>  
 <**property name="prefix" value="/pages/"**></**property**>  
 <**property name="suffix" value=".jsp"**></**property**>  
</**bean**>

### 开启springmvc注解支持

*<!--开启springmvc注解支持-->*<**mvc:annotation-driven**></**mvc:annotation-driven**>

### 释放静态资源

*<!--释放静态资源-->*<**mvc:default-servlet-handler**></**mvc:default-servlet-handler**>

## 编写applicationContext\_service.xml配置文件

### Dao层配置文件

#### 创建连接池对象

*<!--创建c3p0连接池-->*<**bean id="dataSource" class="com.mchange.v2.c3p0.ComboPooledDataSource"**>  
 <**property name="driverClass" value="oracle.jdbc.driver.OracleDriver"** />  
 <**property name="jdbcUrl" value="jdbc:oracle:thin:@192.168.137.132:1521:orcl"** />  
 <**property name="user" value="ssm"** />  
 <**property name="password" value="orcl"** />  
</**bean**>

#### 创建SqlSessionFactory工厂对象

##### 注入连接池

##### 给pojo起别名

使用mapper映射文件就配置别名

##### 配置mabatis分页插件

*<!--创建SqlSessionFactory工厂对象-->*<**bean id="sqlSessionFactory" class="org.mybatis.spring.SqlSessionFactoryBean"**>  
 *<!--注入连接池-->* <**property name="dataSource" ref="dataSource"**></**property**>  
  
 *<!--给pojo起别名 就是domain包下的bean对象其别名,用于Mapper映射器文件中,这里不使用mapper映射器文件,将使用注解方式开发-->  
 <!--用到mapper映射文件中,mapper路径跟domain路径需要 -->  
 <!--这里不使用mapper映射器,将会注释掉-->  
 <!--使用mapper映射文件就配上别名-->  
 <!--<property name="typeAliasesPackage" value="com.itheima.domain"></property>-->  
  
 <!-- 配置MyBatis分页插件PageHelper的连接器 -->* <**property name="plugins"**>  
 <**array**>  
 <**bean class="com.github.pagehelper.PageInterceptor"**>  
 <**property name="properties"**>  
 *<!--使用下面的方式配置参数，一行配置一个 -->* <**value**>  
 reasonable=true  
 </**value**>  
 </**property**>  
 </**bean**>  
 </**array**>  
 </**property**>  
</**bean**>

#### 扫描Dao层接口生成代理对象,放入IOC容器中

*<!--扫描Dao层接口生成代理对象,放入IOC容器中-->*<**bean id="mapperScannerConfigurer" class="org.mybatis.spring.mapper.MapperScannerConfigurer"**>  
 <**property name="basePackage" value="com.itheima.dao"**></**property**>  
</**bean**>

#### 插播广告mapper映射文件

Mapper映射配置文件所在路径必须和mapper接口在同一包路径下

Mapper映射配置文件的名称必须和mapper接口名保持一致

mapper接口就是dao接口

mapper=dao

##### MyBatis主配置文件



mapper映射文件



### Service层配置文件

#### 开启组件扫描

*<!--开启组件扫描-->*<**context:component-scan base-package="com.itheima.service"**></**context:component-scan**>

#### 开启事务管理类

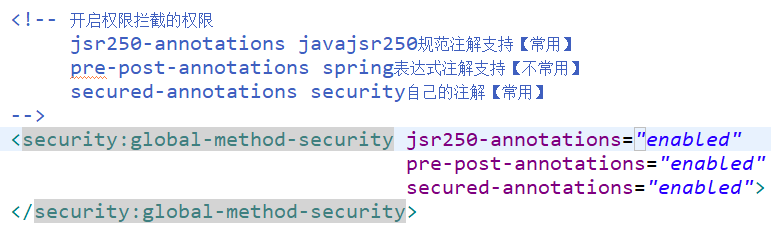
*<!--开启事务管理类-->*<**bean id="transactionManager" class="org.springframework.jdbc.datasource.DataSourceTransactionManager"**>  
 <**property name="dataSource" ref="dataSource"**></**property**>  
</**bean**>  
  
*<!--开启事务注解支持, 使用事务注解省去 <tx:advice> 配置 和 <aop:config> AOP切面配置-->  
<!--<tx:advice>  
 <tx:attributes>  
 <tx:method name="" propagation="" />  
 </tx:attributes>  
</tx:advice>-->  
<!--开启事务注解支持, 使用事务注解省去 <tx:advice> 配置 和 <aop:config> AOP切面配置-->  
<!--<aop:config>  
 <aop:advisor advice-ref=""></aop:advisor>  
</aop:config>-->*

#### 开启事务注解支持

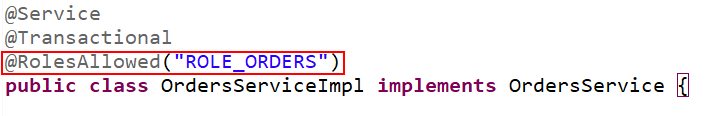
*<!--开启事务注解支持, 使用事务注解省去 <tx:advice> 配置 和 <aop:config> AOP切面配置-->*<**tx:annotation-driven**></**tx:annotation-driven**>

#### 开启权限认证

*<!--开启权限认证-->  
<!-- 三 种  
 jsr250-annotations="enabled" 常用  
 secured-annotations="enabled" 常用  
 pre-post-annotations="enabled" 不常用  
 -->*<**security:global-method-security secured-annotations="enabled"**>  
</**security:global-method-security**>



##### 在业务层service代码上进行拦截

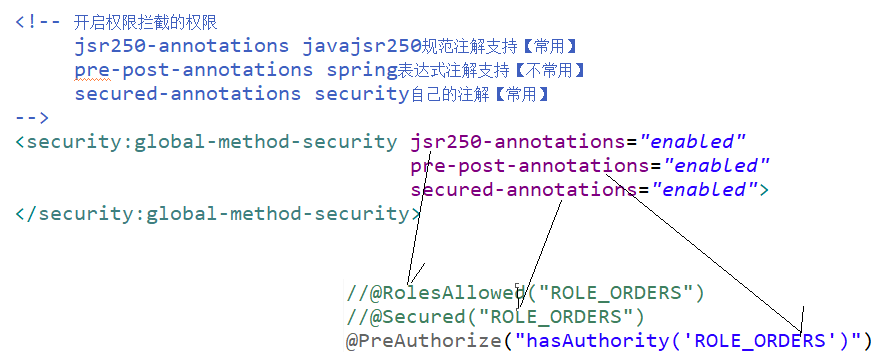


##### 权限拦截注意事项

如果开启权限拦截的配置在springmvc配置文件中，那么对应注解要加到controller对象中。

如果开启权限拦截的配置在spring配置文件中，那么对应注解要加到service对象中。

##### 三种权限拦截的方式



## 编写springSecurity.xml配置文件

### 业务层UserService接口及实现类

#### UserService接口继承UserDetailsService接口

**public interface** SysUserService **extends** UserDetailsService{  
 List<SysUser> findAll();  
 **void** save(SysUser sysUser);  
 SysUser userShow(String uid);  
 **void** addRoleToUser(String[] ids, String uid);  
}

#### UserService接口实现类UserServiceImpl实现UserDetailsService方法

##### UserServiceImpl修改@Service(“与配置ref引用一致”)

@Service(**"userService"**)  
@Transactional  
**public class** SysUserServiceImpl **implements** SysUserService

##### UserServiceImpl实现重写的动态认证方法

@Override  
**public** UserDetails loadUserByUsername(String username) **throws** UsernameNotFoundException {  
 SysUser sysUser = **sysUserDao**.findByUsername(username);  
 **if**(sysUser==**null**){  
 **return null**;  
 }  
 List<SimpleGrantedAuthority> authorities = **new** ArrayList<SimpleGrantedAuthority>();  
 List<SysRole> roles = sysUser.getRoles();  
 **for** (SysRole role : roles) {  
 authorities.add(**new** SimpleGrantedAuthority(role.getRoleName()));  
 }  
 User user = **new** User(username, sysUser.getPassword(),authorities);  
 **return** user;  
}

### Xml配置文部分

#### 释放静态资源

<**security:http security="none" pattern="/login.jsp"**></**security:http**>  
<**security:http security="none" pattern="/failer.jsp"**></**security:http**>  
<**security:http security="none" pattern="/css/\*\*"**></**security:http**>  
<**security:http security="none" pattern="/img/\*\*"**></**security:http**>  
<**security:http security="none" pattern="/plugins/\*\*"**></**security:http**>

#### 拦截,登录,退出,403,等配置

*<!-- auto-config="true" 就表示可以使用security提供默认登录页面，如果自己提供的有登录页面，就用自己的。  
use-expressions="true" 是否使用spring的el表达式 -->* <**security:http auto-config="true" use-expressions="true"**>  
 *<!-- 标签表示要拦截的角色 -->  
 <!-- pattern="/\*\*" 表示哪些请求需要被拦截，/\*\*所有请求都拦截  
 access="hasRole('ROLE\_USER')" 表示项目中基本角色为ROLE\_USER  
 类似于Oracle中的connect角色，没有此角色，什么都做不了。 -->* <**security:intercept-url pattern="/\*\*" access="hasAnyRole('ROLE\_USER','ROLE\_ADMIN')"**/>  
  
 *<!-- 提供自定义登录页面 -->  
 <!-- login-page ： 自定义的登录页面  
 login-processing-url ： 登录的处理器路径  
 default-target-url ： 登录成功后跳转的页面  
 authentication-failure-url ： 登录失败跳转的页面 -->* <**security:form-login login-page="/login.jsp"  
 login-processing-url="/login"  
 default-target-url="/index.jsp"  
 authentication-failure-url="/failer.jsp"**/>  
 *<!-- 退出登录的操作配置 -->  
 <!-- invalidate-session ： 清空session  
 logout-success-url ： 退出登录后跳转的页面  
 logout-url ： 退出登录处理器路径 -->* <**security:logout invalidate-session="true"  
 logout-success-url="/login.jsp"  
 logout-url="/logout"**/>  
  
 *<!-- 关闭csrf拦截 -->  
 <!-- csrf是一个网络攻击技术，security内部认为除了框架内部所有请求都是攻击 -->* <**security:csrf disabled="true"**/>  
  
 *<!-- 配置403权限不足页面 -->* <**security:access-denied-handler error-page="/403.jsp"**/>  
 </**security:http**>

#### 加盐加密对象

<**bean id="passwordEncoder" class="org.springframework.security.crypto.bcrypt.BCryptPasswordEncoder"**></**bean**>

#### 用户配置

*<!-- 用户配置 -->* <**security:authentication-manager**>  
 *<!-- user-service-ref就是security框架会自己去容器中找一个id叫userService  
的UserDetailsService类型的业务类 -->* <**security:authentication-provider user-service-ref="userService"**>  
  
 *<!-- 在内存中模拟两个用户，一个是user具有ROLE\_USER角色，一个是admin有ROLE\_ADMIN  
 {noop}表示不加密登录，springSecurity默认登录是加密的。-->  
 <!--<security:user-service>  
 <security:user name="user" password="{noop}user"  
 authorities="ROLE\_USER" />  
 <security:user name="admin" password="{noop}admin"  
 authorities="ROLE\_ADMIN" />  
 </security:user-service>-->  
  
 <!-- 加盐加密工具里的id,如果不是默认的就要配置这一段 -->  
 <!-- <security:password-encoder ref="加盐加密工具里的id"></security:password-encoder> -->* </**security:authentication-provider**>  
 </**security:authentication-manager**>

### UserServiceImpl加盐加密配置

SpringSecurity已经配置加盐加密对象

需要在添加用户时对密码进行加密

#### UserServiceImpl创建加盐加密对象

@Autowired  
**private** BCryptPasswordEncoder **encoder**;

##### UserServiceImpl保存用户处理器配置

@Override  
**public void** save(SysUser sysUser) {  
 sysUser.setPassword(**encoder**.encode(sysUser.getPassword()));  
 userDao.save(sysUser);  
}

## 编写applicationContext.xml配置文件

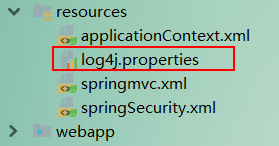
### 引入applicationContext\_service.sml配置文件

*<!--引入applicationContext\_service.xml配置文件-->*<**import resource="classpath:applicationContext\_service.xml"**></**import**>

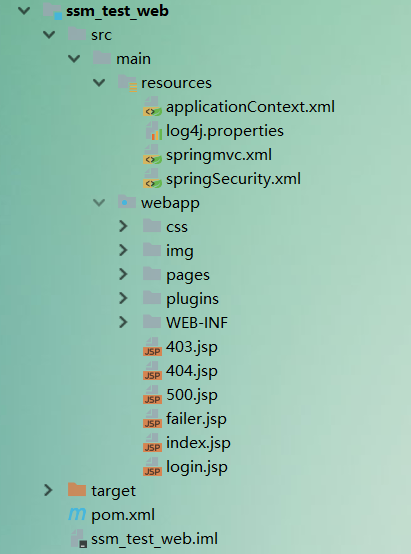
### 引入springSecurity.xml配置文件

*<!--引入springSecurity.xml配置文件-->*<**import resource="classpath:springSecurity.xml"**></**import**>

## 导入log4j.properties日志配置文件



# 把整理好的静态页面导入到项目中



# 创建数据库已经用到的数据表

# 创建各个模块的包

# 实现公共能:

产品页列表展示

产品详情展示

产品添加功能

产品修改功能

日期处理

批量删除功能

级联删除

传统分页

pageHelper分页

安全框架认证

用户列表查询

用户保存

用户详情

加密认证

角色添加

注销操作

认证操作动态授权,就是通过数据库查询用户角色登录认证

用户添加角色

登录显示用户名

权限拦截-页面

权限拦截-后台

403拦截

日志管理