# sprSpring的发展

## Spring1.x时代

在Spring1.x时代，都是通过xml文件配置bean，随着项目的不断扩大，需要将xml配置分放到不同的配置文件中，需要频繁的在java类和xml配置文件中切换。

## Spring2.x时代

随着JDK 1.5带来的注解支持，Spring2.x可以使用注解对Bean进行申明和注入，大大的减少了xml配置文件，同时也大大简化了项目的开发。

那么，问题来了，究竟是应该使用xml还是注解呢？

最佳实践：

1. 应用的基本配置用xml，比如：数据源、资源文件等；
2. 业务开发用注解，比如：Service中注入bean等；

## Spring3.x到Spring4.x

从Spring3.x开始提供了Java配置方式，使用Java配置方式可以更好的理解你配置的Bean，现在我们就处于这个时代，并且Spring4.x和Springboot都推荐使用java配置的方式。

# Spring的Java配置方式

Java配置是Spring4.x推荐的配置方式，可以完全替代xml配置。

## @Configuration和@Bean

Spring的Java配置方式是通过@Configuration 和 @Bean这两个注解实现的：

1、@Configuration作用于类上，**相当于一个xml配置文件**；

2、@Bean作用于方法上，**相当于xml配置中的<bean>**；

示例要点：

AnnotationConfigApplicationContext（之前是用ClassPathXmlApplicationContext来加载Spring的配置文件，加载“配置类”则使用这个）

@ComponentScan(basePackages={"cn.itcast.springboot"})，(等价于<context:component-scan base-package="xxx.xxx.xxx" />)

## @PropertySource

通过@PropertySource可以指定读取的配置文件，通过@Value注解获取值，具体用法：

@PropertySource(value= {"classpath:jdbc.properties"})

等价于：<context:property-placeholder location="classpath:config/dict.properties" />

@Value("${jdbc.url}")

private String jdbcUrl;

1. 设置ignoreResourceNotFound=true,当配置的配置文件不存在时不会报错。  
   

### 示例：配置数据库连接池

参考xml配置改造成java配置方式：

和xml配置一样，思路都是：先加载db.properties,再注入value。

@Value("${jdbc.url}")

**private** String jdbcUrl;

。。。省略driver、password、username

@Bean(destroyMethod = "close")

**public** DataSource dataSource() {

注：这里方法的名字就是这个bean的id，所以建议都用bean的名字，也可以在bean中配name

BoneCPDataSource boneCPDataSource = **new**BoneCPDataSource();

boneCPDataSource.setDriverClass(jdbcDriverClassName);

。。。省略driver、password、username

}

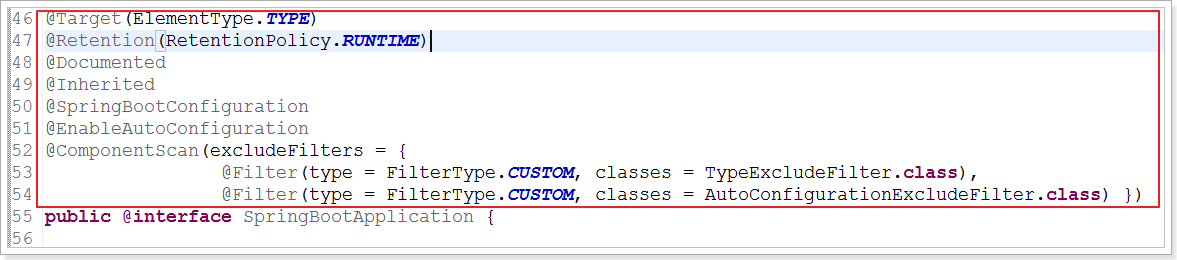
# SpringBoot

## Spring Boot的核心

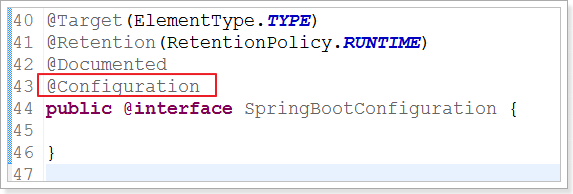
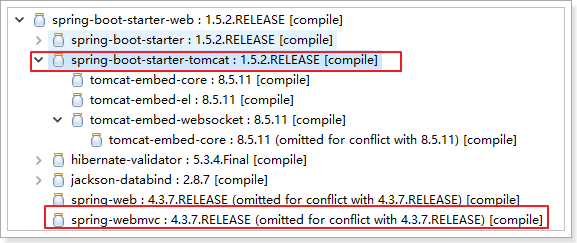
### 入口类和@SpringBootApplication

SpringBoot的项目一般都会有**\*Application**的入口类，入口类中会有main方法，这是一个标准的Java应用程序的入口方法。

**@SpringBootApplication**注解是SpringBoot的核心注解，它其实是一个**组合注解**：

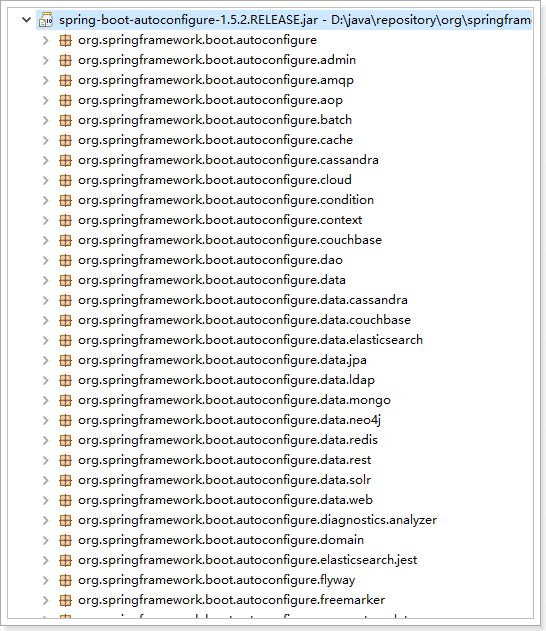


该注解主要组合了以下注解：

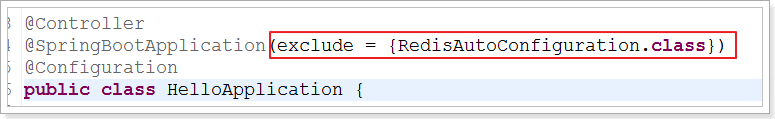
1. @**SpringBootConfiguration**：这是SpringBoot项目的配置注解，这也是一个组合注解：  
     
   在Spring Boot项目中默认使用了@SpringBootConfiguration替代了@Configuration
2. @**EnableAutoConfiguration**：启用自动配置，该注解会使SpringBoot根据项目中依赖的jar包自动配置项目的配置项：
   1. 如：我们添加了spring-boot-starter-web的依赖，项目中也就会引入SpringMVC的依赖，SpringBoot就会自动配置tomcat和SpringMVC等等  
      
3. @**ComponentScan**：默认扫描@SpringBootApplication所在类的**同级目录以及它的子目录**。

### 关闭自动配置

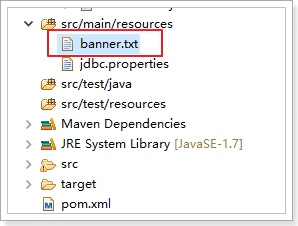
通过上述，我们得知，SpringBoot会根据项目中的jar包依赖，自动做出配置，SpringBoot支持的自动配置如下（非常多）：



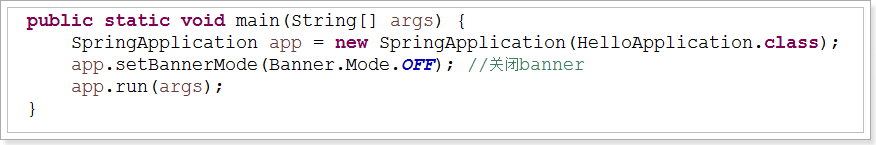
如果想关闭某一项的自动配置，设置如下:



### 自定义Banner

上网查找作成banner.txt，并拷贝到项目的resources目录中：  


如果不想看到任何的banner，也是可以将其关闭的：



### 全局配置文件

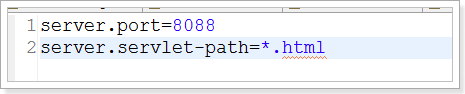
SpringBoot项目使用一个全局的配置文件

**application.properties**或者是**application.yml**

可以放在resources目录下或者类路径下的子目录中，一般我们放到resources下。

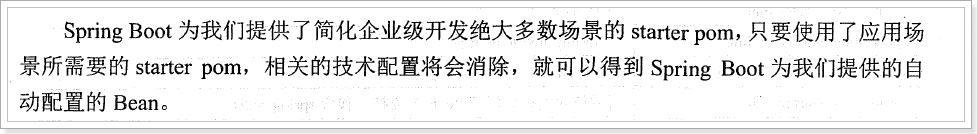
**application.yml的key后面一定要跟一个空格，application.yml某些地方的配置和application.properties格式上会优点不一样。比如：**

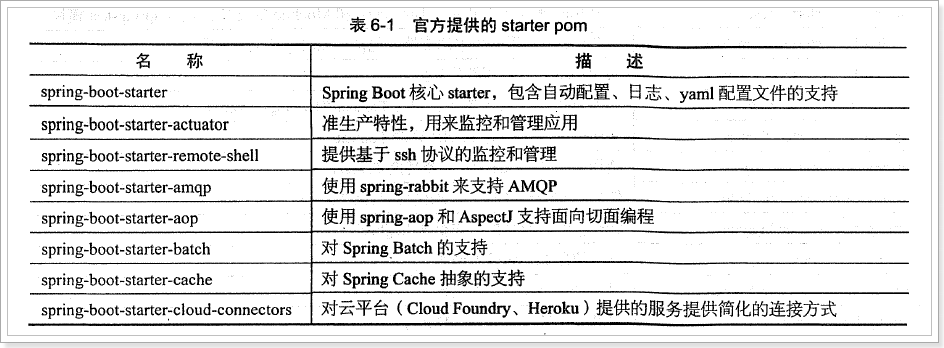
1. 修改进入DispatcherServlet的规则为：\*.html

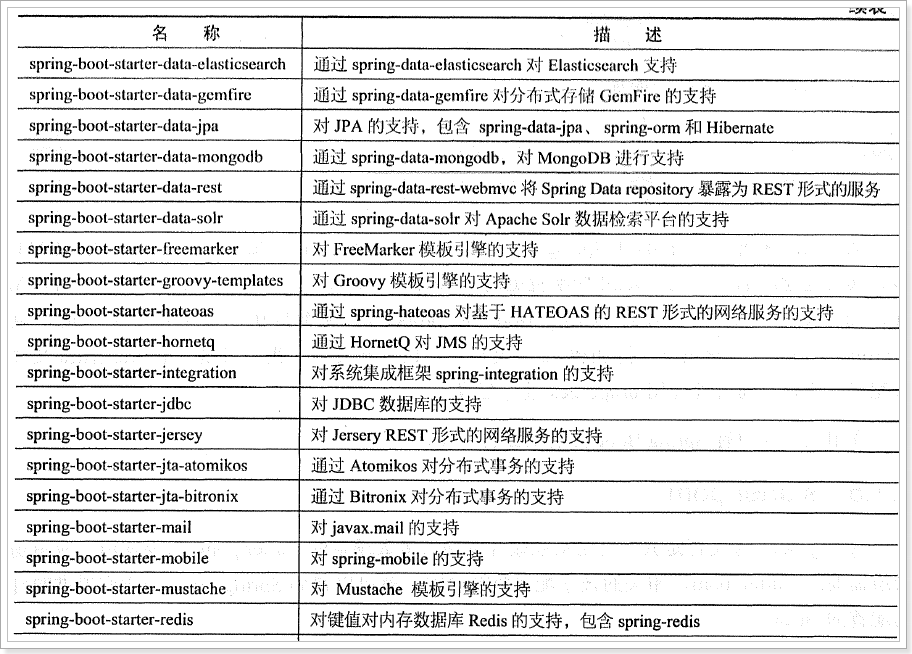
**application.properties配置如下：**  
  
  
**application.yml配置如下：**



### Starter pom









### 导入Xml配置文件



### 日志

SpringBoot对各种日志框架都做了支持，我们可以通过配置来修改日志的配置：

#设置日志级别

logging.level.org.springframework=DEBUG

## SpringBoot的自动配置的原理

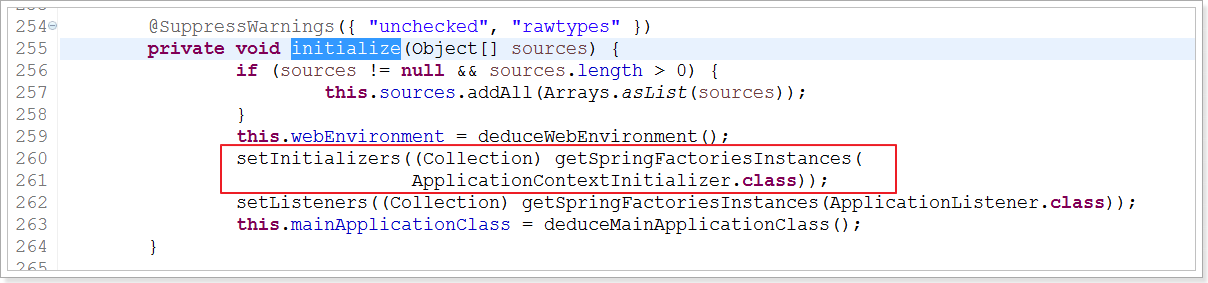
SpringBoot在进行SpringApplication对象实例化时会加载META-INF/spring.factories文件，将该配置文件中的配置载入到Spring容器。

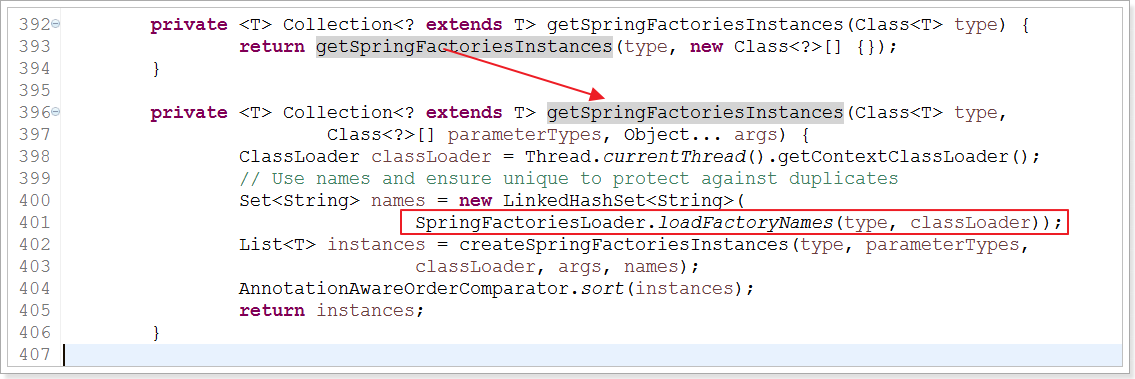
### Maven下载源码

通过dependency:sources该命令可以下载该项目中所有的依赖的包的源码。

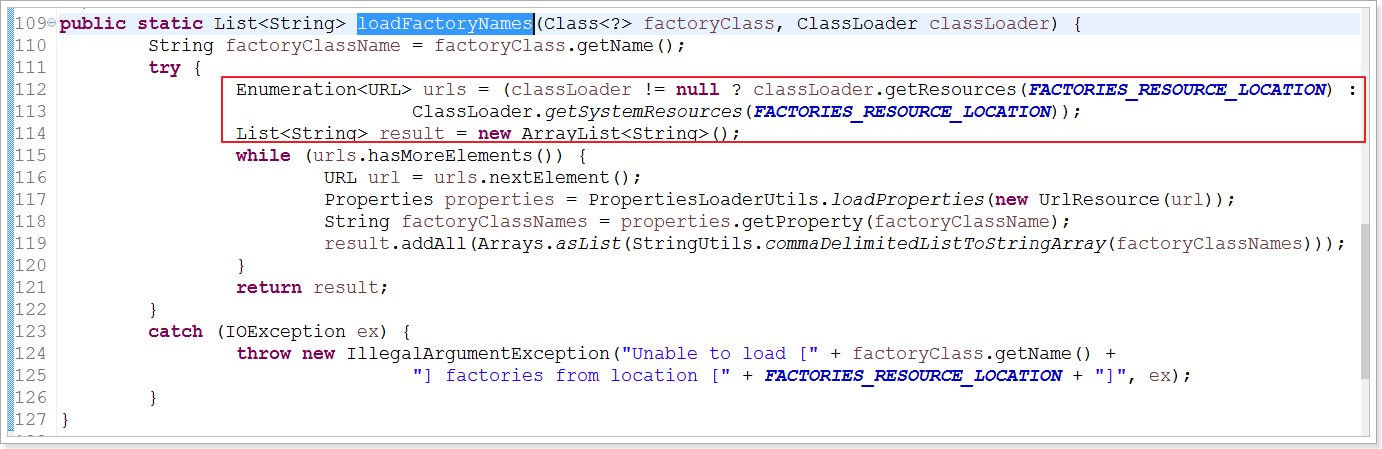
### 源码分析

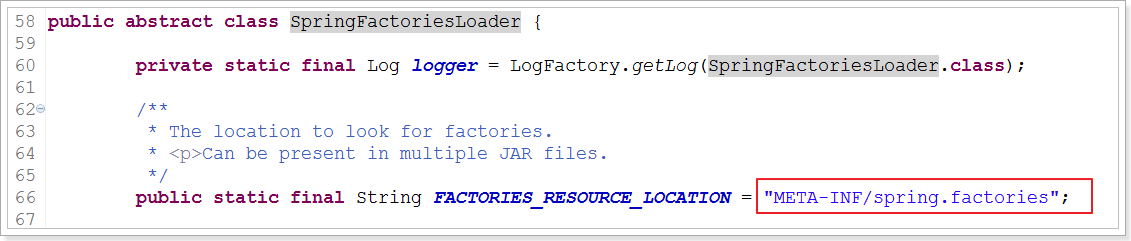
org.springframework.boot.SpringApplication：





org.springframework.core.io.support.SpringFactoriesLoader:





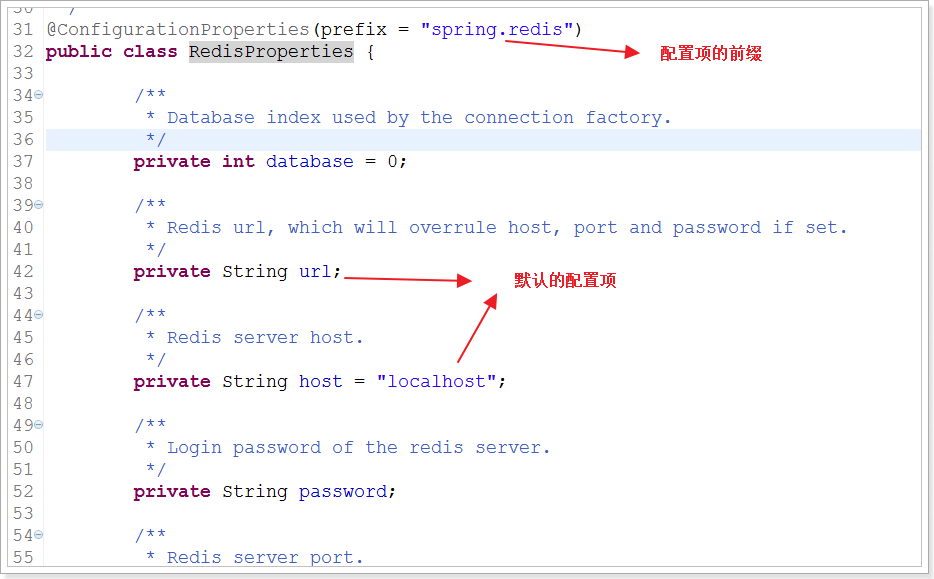
### 举例：Redis的自动配置

Redis对应的自动配置类：

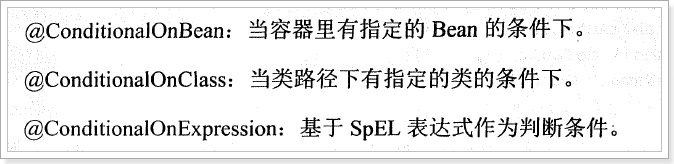
org.springframework.boot.autoconfigure.data.redis.RedisAutoConfiguration

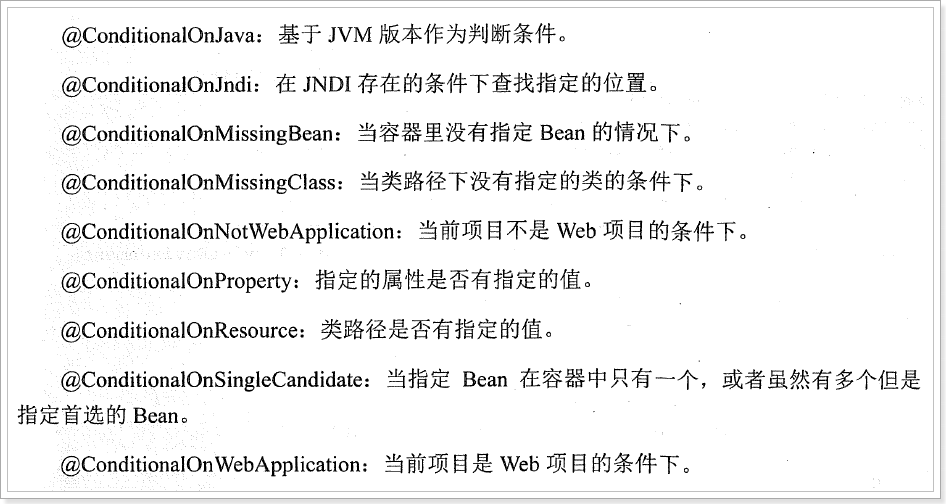
内容：





### 条件注解





# Spring Boot的web开发

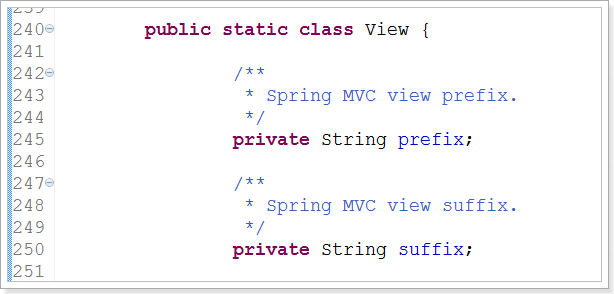
Web开发的自动配置类：**org.springframework.boot.autoconfigure.web.WebMvcAutoConfiguration**

## 举例：自动配置的ViewResolver



视图的配置mvcProperties对象中：

org.springframework.boot.autoconfigure.web.WebMvcProperties.View



## 静态资源的配置和访问

进入规则为 /时，按理来说是拦截规则是：拦截静态资源，但Springboot对静态资源做了映射，只要放在webapp目录下（包括子目录）的静态资源都是可以直接访问的。（**一句话：如果不修改SpringBoot默认的设置，只要放在webapp下（除了WEB-INF之外），静态资源都可以访问到。**）

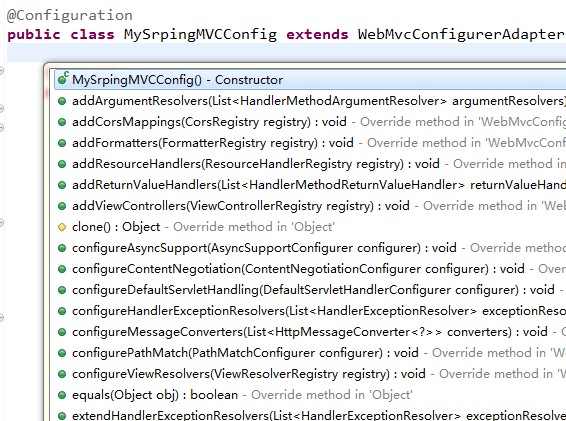
也可以自定义（但不好用，容易访问不到）：

spring.resources.static-locations=classpath:/XXX/

## 自定义SpringMVC的配置

有时候我们需要添加一个拦截器或者添加一个转化器，这个时候就得通过继承WebMvcConfigurerAdapter然后重写父类中的方法进行扩展。

父类中的方法：



举例：添加一个拦截器（重写父类的addInterceptors方法）

这里的拦截方法名字和拦截规则和springmvc一样。



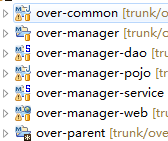
添加转化器：重写addFormatters(…)

…其他百度

# SpringMVC框架改造为SpringBoot框架

改造的思路：

原本的项目结构是：



需要改造的是web模块，其他模块基本上不需要改动的，

原本的web模块是依赖自己的parent模块，而新建的Springboot-web模块则依赖spring-boot-starter-parent，

这意味着：新建的Springboot-web模块已经不是manager模块的子模块了，所以不能在通过启动manager模块来启动项目了，思路上变为：Springboot-web模块通过调用其他模块来启动项目，这意为pojo、dao、service、common都需要打成jar包放到仓库中，由Springboot-web模块来调用。（其他模块有改动，都需要重新install）

这样也意为着：项目中有两个parent（springboot的和自定义的），自定义的parent还是用于版本管理（管理dao/service/common），但其中关于Spring的依赖和web模块需要的依赖都可以删除，因为springboot-parent已经有了。

改造的主要步骤：

1. SpringBoot的项目搭建
2. 代码、页面、非Spring配置文件的复制
3. SpringBoot配置类代替SpringMVC的XML配置文件
4. 调试

## 创建SpringBoot工程

创建新的工程，保留原SpringMVC工程。

打包方式一样，也是war。

## 导入依赖

### 导入spring-boot-starter-parent依赖

<parent>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-parent</artifactId>

<version>1.5.2.RELEASE</version>

</parent>

导入springboot的web支持

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-web</artifactId>

</dependency>

运行插件（不是必须的）：

<plugin>

<groupId>org.springframework.boot</groupId> <artifactId>spring-boot-maven-plugin</artifactId>

</plugin>

### 导入原工程中的maven依赖和插件等等（pom文件）

导入原web工程中的依赖，删除Spring相关的依赖（SpringBoot已经有了）

根据pom修改即可

-----------略---------

## 复制代码、页面、非Spring的配置文件

**因为整个工程的改造过程只需要修改的只有Spring相关的配置文件，以及一点修改。**

**---------------以上为基本转移过程（就是简单的copy）-------------**

**---------------以下为根据Spring的配置文件来改造-------------**

## 编写Spring配置类XxxApplication

这就是SpringBoot项目的入口类（当然可以另外写一个入口类，把入口单独出来，但我觉得没必要）

新建package：xxx.xxx.xxx.config，用于放XXXApplication这些配置类

**@Configuration**

**@SpringBootApplication**

publicclass **OAApplication {**

**//入口**

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

**SpringApplication.run(OAApplication.class, args);**

**}**

**}**

## 整合applicationContext-dao.xml

### 读取外部的配置文件

**@Configuration**

**@PropertySource(value = { "classpath:jdbc.properties", "classpath:env.properties",**

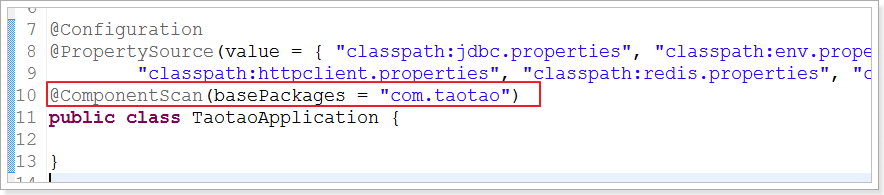
**}, ignoreResourceNotFound =** true**)**

Public class **TaotaoApplication {**

**}**

### 扫描配置包

建议扫描层级较高的包，这样会扫描所有Spring的注解



### 配置连接池

**@Value("${jdbc.url}")**

private **String jdbcUrl;**

**。。。drivername。url。username。password**

**@Bean(destroyMethod = "close")**

public **DataSource dataSource() {**

**BoneCPDataSource boneCPDataSource =** new**BoneCPDataSource();**

**// 数据库驱动**

**boneCPDataSource.setDriverClass(jdbcDriverClassName);**

**// 相应驱动的jdbcUrl**

**boneCPDataSource.setJdbcUrl(jdbcUrl);**

**// 数据库的用户名**

**boneCPDataSource.setUsername(jdbcUsername);**

**// 数据库的密码**

**boneCPDataSource.setPassword(jdbcUsername);**

return**boneCPDataSource;**

**}**

### 配置SqlSessionFactoryBean

**该步会整合Mybatis的配置文件SqlMapConfig.xml，所以该步即Mybatis和Spring Boot整合、**

**单独建一个**MyBatisConfig**的配置类**

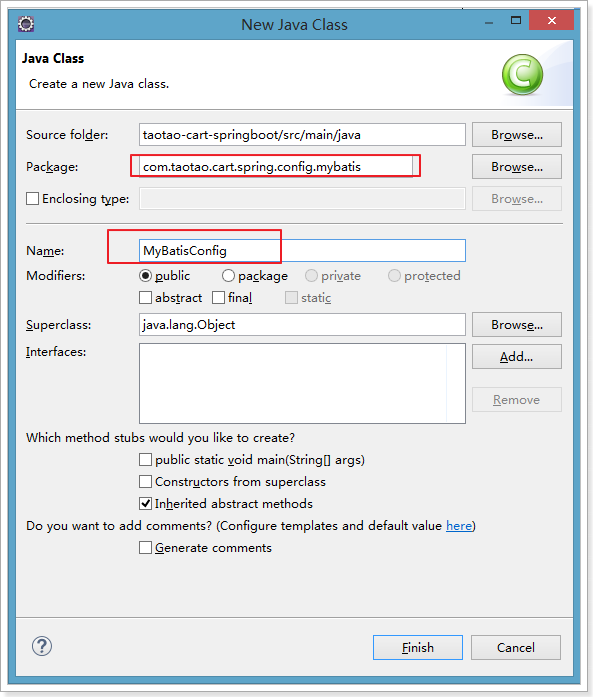
Mybatis和Spring Boot的整合有两种方式：

第一种：使用mybatis官方提供的Spring Boot整合包实现，地址：<https://github.com/mybatis/spring-boot-starter>

第二种：使用mybatis-spring整合的方式，也就是我们传统的方式（推荐，方便自定义配置）

首先，创建一个Mybatis的配置类：

在config包下建立子包：mybatis（SpringBoot会扫描到子包）



代码：

@Configuration

publicclass MyBatisConfig {

@Bean

@ConditionalOnMissingBean//当容器里没有指定的Bean的情况下创建该对象（专业）

public SqlSessionFactoryBean sqlSessionFactory(DataSource dataSource) {

注：这里DataSource等价于注入，两种方式都可以

SqlSessionFactoryBean sqlSessionFactoryBean = new SqlSessionFactoryBean();

// 设置数据源

sqlSessionFactoryBean.setDataSource(dataSource);

// 设置mybatis的主配置文件

ResourcePatternResolver resolver = new PathMatchingResourcePatternResolver();

Resource mybatisConfigXml = resolver.getResource("classpath:mybatis/mybatis-config.xml");

sqlSessionFactoryBean.setConfigLocation(mybatisConfigXml);

// 设置别名（如果SQLMapConfig配了，这里就不用配了）

sqlSessionFactoryBean.setTypeAliasesPackage("com.taotao.cart.pojo");

returnsqlSessionFactoryBean;

}

}

#### 关于PathMatchingResourcePatternResolver

PathMatchingResourcePatternResolver是ResourcePatternResolver的子类，

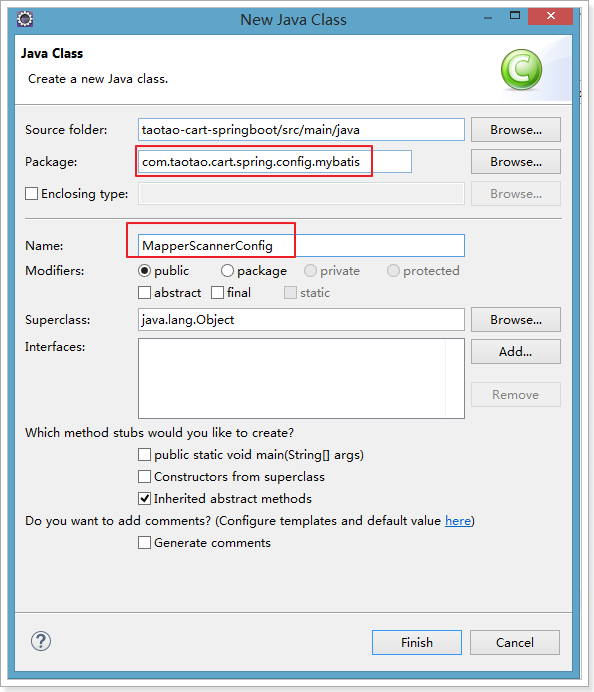
ResourcePatternResolver继承 ResourceLoader，

PathMatchingResourcePatternResolver是Resource查找器，可用来查找类路径下资源。

### 配置MapperScannerConfigurer

**创建Mapper接口的扫描类MapperScannerConfig**

作用：扫描mapper.xml文件（之前在mybatis.xml文件中也配置过）



@Configuration

@AutoConfigureAfter(MyBatisConfig.class) //保证在MyBatisConfig实例化之后再实例化该类

Public class MapperScannerConfig {

// mapper接口的扫描器

@Bean

public MapperScannerConfigurer mapperScannerConfigurer() {

MapperScannerConfigurer mapperScannerConfigurer = new MapperScannerConfigurer();

//设置mapper接口所在的包

mapperScannerConfigurer.setBasePackage("com.taotao.cart.mapper");

return mapperScannerConfigurer;

}

}

**applicationContext-service.xml只需要扫描service层的Spring注解，这个在入口类中已经完成了，所以这个不需要。**

## 整合applicationContext-trans.xml

**即设置事务管理**

在Spring Boot中推荐使用@Transactional注解来申明事务。

首先需要导入依赖：

<dependency>

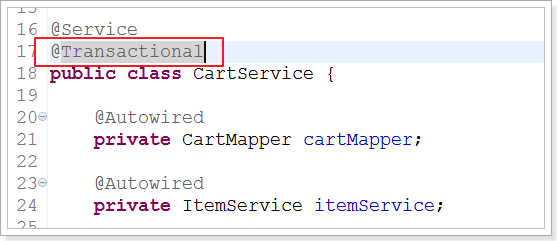
<groupId>org.springframework.boot</groupId>

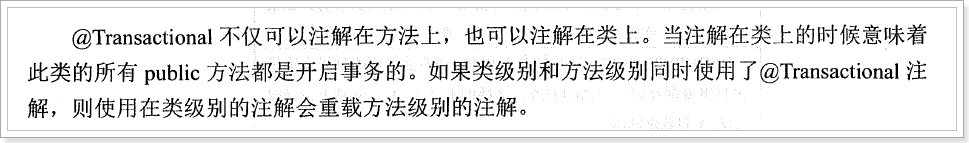
<artifactId>spring-boot-starter-jdbc</artifactId>

</dependency>

当引入jdbc依赖之后，Spring Boot会自动默认分别注入DataSourceTransactionManager或JpaTransactionManager，所以我们不需要任何额外配置就可以用@Transactional注解进行事务的使用。

在Service中添加@Transactional注解：





## 整合Redis（未学，笔记未整理）

在Spring Boot中提供了RedisTempplate的操作，我们暂时不做学习，先按照我们之前的实现来完成。

代码：

import java.util.ArrayList;

import java.util.List;

import org.springframework.beans.factory.annotation.Value;

import org.springframework.context.annotation.Bean;

import org.springframework.context.annotation.Configuration;

import org.springframework.context.annotation.PropertySource;

import redis.clients.jedis.JedisPoolConfig;

import redis.clients.jedis.JedisShardInfo;

import redis.clients.jedis.ShardedJedisPool;

@Configuration

@PropertySource(value = "classpath:redis.properties")

publicclass RedisSpringConfig {

@Value("${redis.maxTotal}")

private Integer redisMaxTotal;

@Value("${redis.node1.host}")

private String redisNode1Host;

@Value("${redis.node1.port}")

private Integer redisNode1Port;

private JedisPoolConfig jedisPoolConfig() {

JedisPoolConfig jedisPoolConfig = newJedisPoolConfig();

jedisPoolConfig.setMaxTotal(redisMaxTotal);

returnjedisPoolConfig;

}

@Bean

publicShardedJedisPool shardedJedisPool() {

List<JedisShardInfo>jedisShardInfos = new ArrayList<JedisShardInfo>();

jedisShardInfos.add(new JedisShardInfo(redisNode1Host, redisNode1Port));

returnnew ShardedJedisPool(jedisPoolConfig(), jedisShardInfos);

}

}

## 整合Httpclient（未学，笔记未整理）

import org.apache.http.client.config.RequestConfig;

import org.apache.http.impl.client.CloseableHttpClient;

import org.apache.http.impl.client.HttpClients;

import org.apache.http.impl.conn.PoolingHttpClientConnectionManager;

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.beans.factory.annotation.Value;

import org.springframework.context.annotation.Bean;

import org.springframework.context.annotation.Configuration;

importorg.springframework.context.annotation.PropertySource;

import org.springframework.context.annotation.Scope;

import com.taotao.common.httpclient.IdleConnectionEvictor;

@Configuration

@PropertySource(value = "classpath:httpclient.properties")

publicclass HttpclientSpringConfig {

@Value("${http.maxTotal}")

private Integer httpMaxTotal;

@Value("${http.defaultMaxPerRoute}")

private Integer httpDefaultMaxPerRoute;

@Value("${http.connectTimeout}")

private Integer httpConnectTimeout;

@Value("${http.connectionRequestTimeout}")

private Integer httpConnectionRequestTimeout;

@Value("${http.socketTimeout}")

private Integer httpSocketTimeout;

@Value("${http.staleConnectionCheckEnabled}")

private Boolean httpStaleConnectionCheckEnabled;

@Autowired

private PoolingHttpClientConnectionManager manager;

@Bean

public PoolingHttpClientConnectionManager poolingHttpClientConnectionManager() {

PoolingHttpClientConnectionManager poolingHttpClientConnectionManager = newPoolingHttpClientConnectionManager();

// 最大连接数

poolingHttpClientConnectionManager.setMaxTotal(httpMaxTotal);

// 每个主机的最大并发数

poolingHttpClientConnectionManager.setDefaultMaxPerRoute(httpDefaultMaxPerRoute);

returnpoolingHttpClientConnectionManager;

}

// 定期关闭无效连接

@Bean

public IdleConnectionEvictor idleConnectionEvictor() {

returnnew IdleConnectionEvictor(manager);

}

// 定义Httpclient对

@Bean

@Scope("prototype")

public CloseableHttpClient closeableHttpClient() {

return HttpClients.custom().setConnectionManager(this.manager).build();

}

// 请求配置

@Bean

public RequestConfig requestConfig() {

return RequestConfig.custom().setConnectTimeout(httpConnectTimeout) // 创建连接的最长时间

.setConnectionRequestTimeout(httpConnectionRequestTimeout) // 从连接池中获取到连接的最长时间

.setSocketTimeout(httpSocketTimeout) // 数据传输的最长时间

.setStaleConnectionCheckEnabled(httpStaleConnectionCheckEnabled)// 提交请求前测试连接是否可用

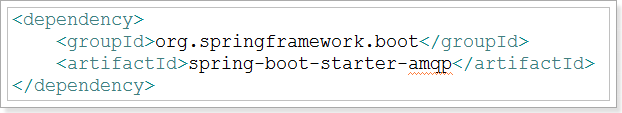
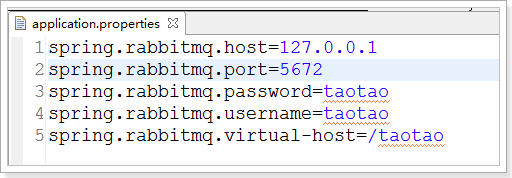
.build();

}

}

## 整合RabbitMQ（未学，笔记未整理）

我们之前使用的Spring-Rabbit的xml方式，现在我们要改造成java方式，并且Spring Boot对RabbitMQ的使用做了自动配置，更加的简化了我们的使用。

1. 在导入spring-boot-starter-amqp的依赖；  
   
2. 在application.properties文件中配置RabbitMQ的连接信息  
   

编写Rabbit的Spring配置类  
import org.springframework.amqp.core.Queue;

import org.springframework.amqp.rabbit.connection.ConnectionFactory;

import org.springframework.amqp.rabbit.core.RabbitAdmin;

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.context.annotation.Bean;

import org.springframework.context.annotation.Configuration;

@Configuration

publicclass RabbitMQSpringConfig {

@Autowired

private ConnectionFactory connectionFactory;

// 管理

@Bean

public RabbitAdmin rabbitAdmin() {

returnnew RabbitAdmin(connectionFactory);

}

// 声明队列

@Bean

public Queue taotaoCartLoginQueue() {

// 默认就是自动声明的

returnnew Queue("TAOTAO-CART-LOGIN-QUEUE", true);

}

// 声明队列

@Bean

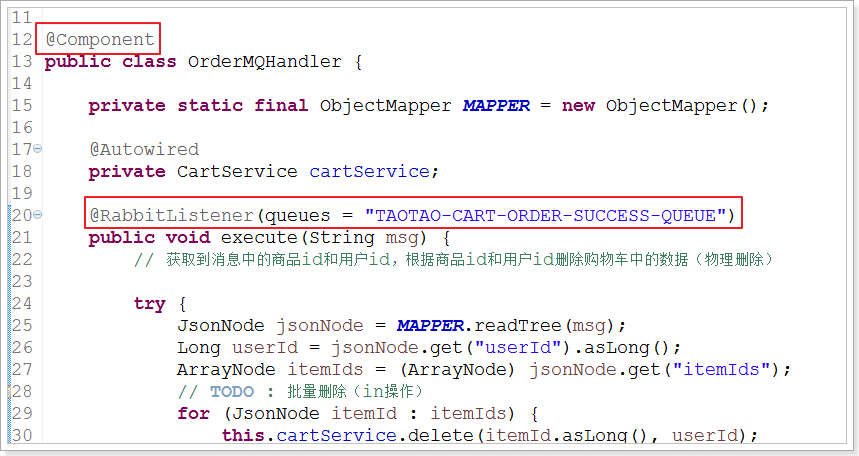
public Queue taotaoCartOrderSuccessQueue() {

// 默认就是自动声明的

returnnew Queue("TAOTAO-CART-ORDER-SUCCESS-QUEUE", true);

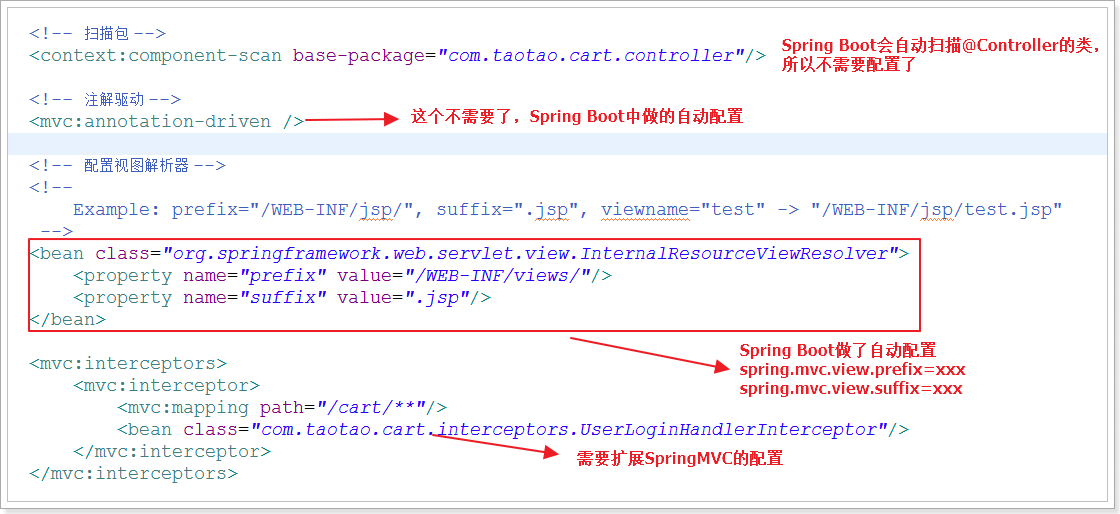
}

}

1. 设置监听  
     
   

## 整合springmvc.xml

原有配置：



### 视图解析器配置：

Spring.mvc.view.prefix=/WEB-INF/views/

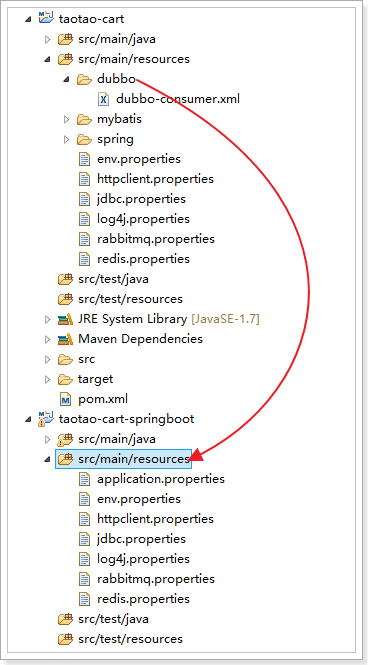
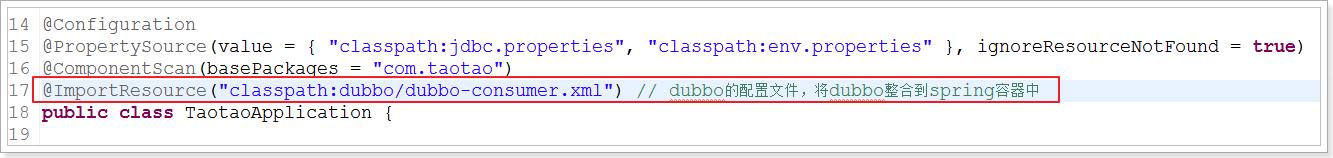
Spring.mvc.view.suffix=.jsp

### 自定义拦截器配置：

**参考上面自定义SpringMVC的配置的内容。**

## 整合dubbo（未学，笔记未整理）

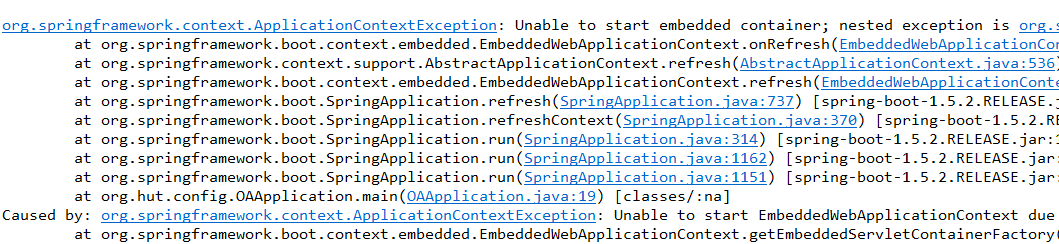
Dubbo目前只能使用xml配置的方式，所以我们需要保留xml，并且需要将该xml加入到现有的Spring容器中才能生效。

1. 将dubbo目录以及下面的xml配置文件拷贝到taotao-cat-springboot中  
   
2. 将dubbo的xml文件加入到spring容器  
   

## 可能出现的Bug

### 启动错误

关键错误（丢失了web容器的工厂，也就是说我们并没有把它作为一个web应用来启动）：

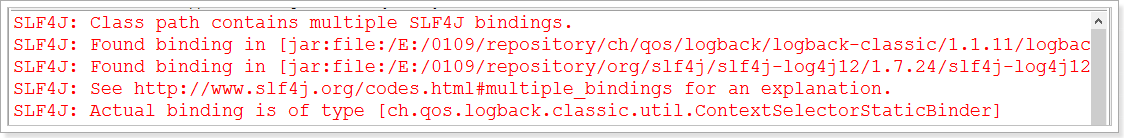


解决：

入口类缺失了@SpringBootApplication的注解

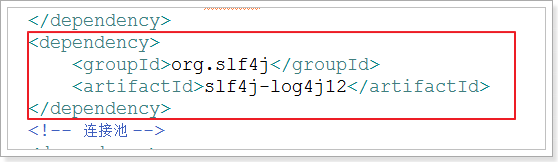
让Spring Boot来自动选择并且完成web的相关加载工作。

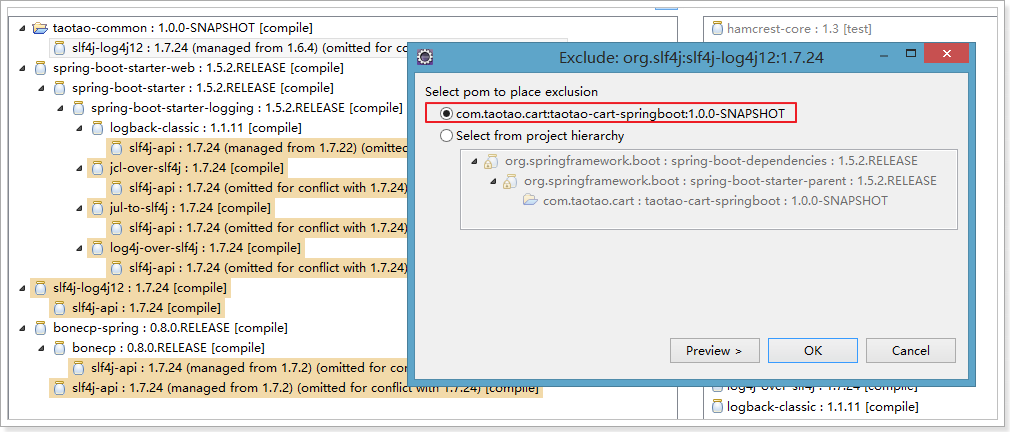
### Slf4j日志警告



提示我们当前的项目中slf4j引入了2个，导致了jar冲突。

解决：

第一种：删除自己引入到slf4j的依赖  


第二种：将common模块中传递的依赖排除掉  
  
  
再次启动，发现警告没了

### 解决jsp访问404的问题

由于Spring boot使用的内嵌的tomcat，而内嵌的tamcat是不支持jsp页面的(自己下载的tonmcat中是有jsp解析引擎--即jasper)，所有需要导入额外的包才能解决。

<dependency>

<groupId>org.apache.tomcat.embed</groupId>

<artifactId>tomcat-embed-jasper</artifactId>

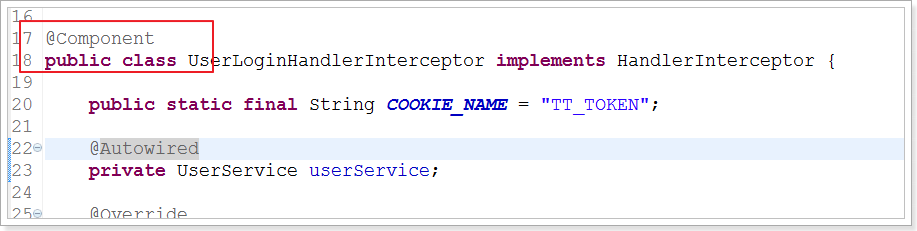
<scope>provided</scope>

</dependency>

### 拦截器中的UserService空指针异常

分析：由于添加拦截器时，直接对UserLoginHandlerInterceptor进行new操作，导致UserService无法注入，所以有空指针异常。

解决：





### 路径问题

现在我们进入Servlet的路径为”/”，访问\*.html页面没问题，但是，访问 /service/\*就会有问题，所以需要改一下js，将原有的/service/ 改为 /

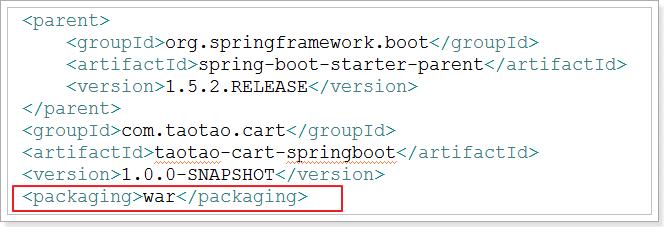


测试，功能一切ok。

# 发布到独立的tomcat中运行

在开发阶段我们推荐使用内嵌的tomcat进行开发，因为这样会方便很多，但是到生成环境，我希望在独立的tomcat容器中运行，因为我们需要对tomcat做额外的优化，这时我们需要将工程打包成war包发进行发布。

## 工程的打包方式为war



## 将spring-boot-starter-tomcat的范围设置为provided

设置为provided是在打包时会将该包排除，因为要放到独立的tomcat中运行，是不需要的。

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-tomcat</artifactId>

<scope>provided</scope>

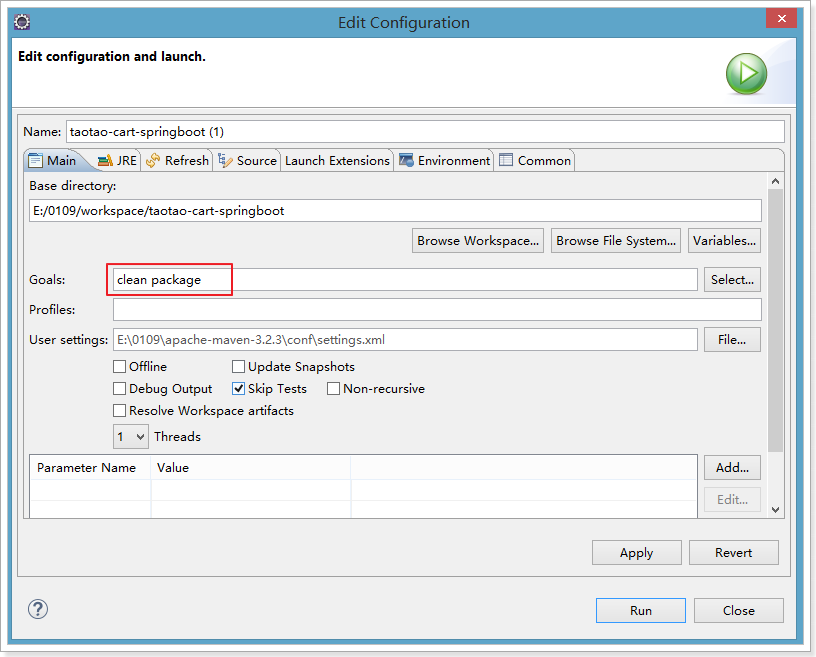
</dependency>

## 修改代码，设置启动配置

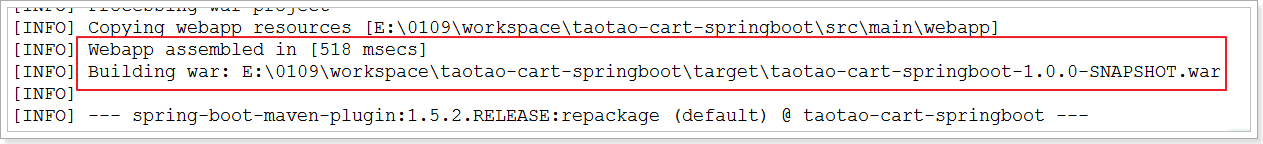
需要集成SpringBootServletInitializer，然后重写configure，将Spring Boot的入口类设置进去。



## 打war包



打包成功：



## 部署到tomcat

解压apache-tomcat-7.0.57.tar.gz，将war包解压到webapps下的ROOT目录中，启动：

