Springboot学习笔记

1. 构建第一个springboot应用
   1. 构建过程
      1. 创建Maven工程。编辑pom文件，pom文件集成spring-boot-starter-parent

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

<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">

<modelVersion>4.0.0</modelVersion>

<groupId>com.example</groupId>

<artifactId>myproject</artifactId>

<version>0.0.1-SNAPSHOT</version>

<parent>

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

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

<version>2.0.5.RELEASE</version>

</parent>

*<!-- Additional lines to be added here... -->*

</project>

* + 1. 导入Springboot应用依赖：

Springboot提供了很多起步组件（用于方便导入springboot所依赖的jar包的maven打包集合）。Springboot官方建议普通springboot应用直接继承默认的spring-boot-statarer-parent来导入springboot默认依赖的包。

* + 1. 导入Springboot-web应用默认依赖

在 pom文件中导入spring-web依赖包

<dependencies>

<dependency>

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

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

</dependency>

</dependencies>

想查看具体依赖那些包可以运行maven命令查看

mvn dependency:tree

* + 1. 现在springboot-web应用的依赖包导入完成，可以开始编写代码测试

**import** org.springframework.boot.\*;

**import** org.springframework.boot.autoconfigure.\*;

**import** org.springframework.web.bind.annotation.\*;

*@RestController*

*@EnableAutoConfiguration*

**public** **class** Example {

*@RequestMapping("/")*

String home() {

**return** "Hello World!";

}

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

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

}

}

就这样，最简单的springboot-web工程就好了，现在直接运行这个main方法就可以启动这个springboot-web应用了。

mvn spring-boot:run

这个命令是以maven的方式启动，运行main方法的方式是普通的Java启动方式。

启动日志：

$ mvn spring-boot:run

. \_\_\_\_ \_ \_\_ \_ \_

/\\ / \_\_\_'\_ \_\_ \_ \_(\_)\_ \_\_ \_\_ \_ \ \ \ \

( ( )\\_\_\_ | '\_ | '\_| | '\_ \/ \_` | \ \ \ \

\\/ \_\_\_)| |\_)| | | | | || (\_| | ) ) ) )

' |\_\_\_\_| .\_\_|\_| |\_|\_| |\_\\_\_, | / / / /

=========|\_|==============|\_\_\_/=/\_/\_/\_/

:: Spring Boot :: (v2.0.5.RELEASE)

....... . . .

....... . . . (log output here)

....... . . .

........ Started Example in 2.222 seconds (JVM running for 6.514)

在浏览器中输入

Localhost:8080就可以查看接口输出了。

* 1. 构建过程必要解释
     1. 包依赖

spring-boot-stater-\*

springboot提供了很多起步依赖包管理并用maven方式打包在一起以方便用户快速的导入构建的应用所依赖的包。它们的groupId都是org.springframework.boot，artifactId是spring-boot-starter-\*

在springboot官方中对所有的springboot应用都抽出了一个公用的父工程：spring-boot-starter-parent。用户可以直接继承这个父工程来快速构建springboot应用。这就是我们直接在我们的pom文件能简单的写上pring-boot-starter-parent来构建应用的原因。

spring-boot-starter-web也一样，也是一组构建springboot的web应用的包的集合，我们可以直接引入这个pom来快速构建springboot应用。

* + 1. 代码解释

我们就写了个简单的Java类，并启动运行了这个类

**import** org.springframework.boot.\*;

**import** org.springframework.boot.autoconfigure.\*;

**import** org.springframework.web.bind.annotation.\*;

*@RestController*

*@EnableAutoConfiguration*

**public** **class** Example {

*@RequestMapping("/")*

String home() {

**return** "Hello World!";

}

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

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

}

}

这是一个带有main方法的简单Java类

* + - 1. 我们用的了几个注解

@RestController and @RequestMapping

@RestController注解表明这个类是一个spring标识的web接口类，并且这个类里的接口都返回数据放在响应体中（至于是什么媒体类型的数据需要后台指定比如@RequestMapping(value = {"/name1","/name2"},produces = "text/plain;charset=UTF-8") 这里的produces就可以指定媒体类型和编码方式）。功能就相当于springmvc里的@Controller，加@ResponseBody

@RequestMapping就是路由注解

@EnableAutoConfiguration

这个注解我们理解为可自动配置注解

1：这注解表明这个类是springboot应用的的主程序类。

2：这个注解表明这个类是一个springboot的配置类

3：这个注解使得工程会自动根据我们导入的包来判断这个工程是什么工程，然后按对应工程的方式进行自动配置（比如我们导入了web包，工程会智能的知道我们的工程是web工程）

* + - 1. main()方法

这里我们用运行main方法的形式来运行springboot的应用。Main方法里就只执了这行代码。

SpringApplication.run(Example.**class**, args);这里就只需要指定我们的springboot应用的主类是那个就可以。

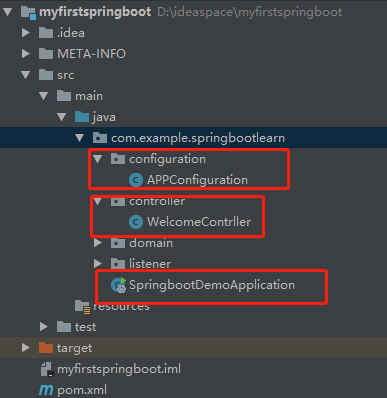
为了方便快速构建，官方给的事例代码的这个类其实充当了很多角色

1：接口角色（controller注解）

2：配置类角色（*@EnableAutoConfiguration注解*）

3：springboot应用主类角色（SpringApplication.run(Example.**class**, args);）

按最理想、合适（个人认为）的方式，我们在开发者这几种角色应该是分开的，如下是我自己分开后的demo



Springboot应用主类

package com.example.springbootlearn;  
import org.springframework.boot.SpringApplication;  
import org.springframework.boot.autoconfigure.SpringBootApplication;  
import org.springframework.context.annotation.ComponentScan;  
*/\*\*  
 \* 表明这是springboot应用的主类  
 \*/*  
@SpringBootApplication  
@ComponentScan(basePackages = {"com.example.springbootlearn"})  
public class SpringbootDemoApplication {  
 public static void main(String[] args) throws Exception {  
 SpringApplication.*run*(SpringbootDemoApplication.class, args);   
 }  
}

@SpringBootApplication注解表明这是应用程序主类

@ComponentScan注解配置springboot注解扫描范围

配置类

package com.example.springbootlearn.controller;  
  
import com.example.springbootlearn.domain.User;  
import org.springframework.stereotype.Controller;  
import org.springframework.web.bind.annotation.RequestMapping;  
import org.springframework.web.bind.annotation.ResponseBody;  
  
import java.util.Calendar;  
import java.util.UUID;  
  
*/\*\*  
 \* spingboot应有的接口类  
 \*/*@Controller  
@RequestMapping("/welcome")  
public class WelcomeContrller {  
  
 @RequestMapping("/user")  
 @ResponseBody  
 public User home() {  
 User user = new User();  
 user.setId(UUID.*randomUUID*().toString());  
 user.setName("django");  
 Calendar ca = Calendar.*getInstance*();  
 ca.set(Calendar.*YEAR*,1988);  
 ca.set(Calendar.*MONTH*,4);  
 ca.set(Calendar.*DATE*,23);  
 user.setBirthday(ca.getTime());  
 user.setIntruduce("hello,im am willion.django,vary nice to meetting you!");  
 return user;  
 }  
  
 @RequestMapping("/hello")  
 @ResponseBody  
 public String hello() {  
 return "welcome to springboot words,this a simple text response!";  
 }  
}

* + 1. 运行springboot应用
       1. 指定应用主类，运行main方法
       2. maven命令运行

mvn spring-boot:run

* + - 1. 打成jar包运行

在pom中加入

<build>

<plugins>

<plugin>

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

<artifactId>spring-boot-maven-plugin</artifactId>

</plugin>

</plugins>

</build>

执行

mvn package

在打包的时候后生产一个应用的jar包

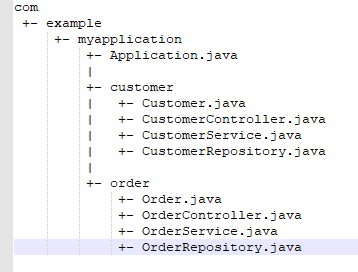
查看jar包含啥东西

jar tvf target/myproject-0.0.1-SNAPSHOT.jar

执行运行jar包命令

java -jar target/myproject-0.0.1-SNAPSHOT.jar

1. 开始使用springboot-------spirngboot最佳实践
   1. Springboot核心
      1. 工程目录构建建议



建议将springboot应用程序主类放在根包下

**package** com.example.myapplication;

**import** org.springframework.boot.SpringApplication;

**import** org.springframework.boot.autoconfigure.SpringBootApplication;

*@SpringBootApplication*

**public** **class** Application {

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

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

}

}

*@SpringBootApplication注解用于标识那个类是springboot应用程序主类*

如果你不想用这个注解， @EnableAutoConfiguration and @ComponentScan这两个注解客户替代它。@ComponentScan配置注解扫描范围。

*@SpringBootApplication默认会扫描所在包范围下的注解*

* + 1. Java配置导入

Springboot建议使用基于Java的方式进行配置，就算你有很多基于xml的配置资源，也推荐你的主配置类是一个带用@Configuration 注解的Java文件。通常情况下，应用程序主类会被选择作为这个主要的配置类。

你可以写很多配置类，然后在一个带有@Configuration注解的主配置类中用@Impor导入其他配置类。

package com.example.springbootlearn.configuration;  
import org.springframework.context.annotation.Configuration;  
import org.springframework.context.annotation.Import;  
  
*/\*\*  
 \* spirngboot应用的配置类  
 \*/*@Configuration  
@Import({xxx.class,yyyy.class})  
public class APPConfiguration {  
 //这里面添加配置项  
}

另外，你也可以在所有的配置类上写@Configuration注解，让后在应用程序主类上用 @ComponentScan注解配置注解扫描范围，这也会扫描到配置注解。

* + 1. 导入xml配置

如果你项目中一定要用到xml配置，也确保的使用一个Java配置类，在这个Java配置类上加入@ImportResource注解来导入你的xml配置

*/\*\*  
 \* spirngboot应用的配置类  
 \*/*@Configuration  
@ImportResource  
public class APPConfiguration {  
 //这里面添加配置项  
}

* + 1. 自动配置 Auto-configuration

自动配置这东西尽量少用吧，都不知道人家干了啥

Springboot自动配置的意思是，springboot会根据你导入到classpath的jar包来对你的应用进行配置。For example, if HSQLDBis on your classpath, and you have not manually configured any database connection beans, then Spring Boot auto-configures an in-memory database.

要是有自动配置，你需要在你的配置主类对象上加上@EnableAutoConfiguration 或@SpringBootApplication注解

（个人觉得加@EnableAutoConfiguration好些，@SpringBootApplication就让他表明是springboot应用主类就好了）

如果有些自动配置加载的类你不想要，可以用注解排除

比如：

**import** org.springframework.boot.autoconfigure.\*;

**import** org.springframework.boot.autoconfigure.jdbc.\*;

**import** org.springframework.context.annotation.\*;

*@Configuration*

*@EnableAutoConfiguration(exclude={DataSourceAutoConfiguration.class})*

**public** **class** MyConfiguration {

}

* + 1. Spring Beans and Dependency Injection（spring beans管理和依赖注入）

就普通的使用spring项目一样，很多时候我们就用

@ComponentScan 去配置注解扫描范围，用 @Autowired来进行依赖注入。

如果你的应用程序主类是在根包下，@ComponentScan在不手动配做扫描范围的时候就默认扫描根包下的注解，@Component, @Service, @Repository, @Controller 等包都会包扫描到被spring容器管理起来。

* + 1. 使用@SpringBootApplication注解

@SpringBootApplication注解包含

@EnableAutoConfiguration、@ComponentScan、@Configuration注解的功能。也就是说应用程序主类上一般说明这个应用程序支持科自动配置、指明可导入外部（不在主类里的）配置文件、注解扫描范围。

官方原文如下：

Many Spring Boot developers like their apps to use auto-configuration, component scan and be able to define extra configuration on their "application class". A single@SpringBootApplication annotation can be used to enable those three features, that is:

* @EnableAutoConfiguration: enable [Spring Boot’s auto-configuration mechanism](https://docs.spring.io/spring-boot/docs/2.0.5.RELEASE/reference/htmlsingle/#using-boot-auto-configuration)
* @ComponentScan: enable @Component scan on the package where the application is located (see [the best practices](https://docs.spring.io/spring-boot/docs/2.0.5.RELEASE/reference/htmlsingle/#using-boot-structuring-your-code))
* @Configuration: allow to register extra beans in the context or import additional configuration classes

The @SpringBootApplication annotation is equivalent to using @Configuration, @EnableAutoConfiguration, and @ComponentScan with their default attributes, as shown in the following example:

**package** com.example.myapplication;

**import** org.springframework.boot.SpringApplication;

**import** org.springframework.boot.autoconfigure.SpringBootApplication;

*@SpringBootApplication* *// same as @Configuration @EnableAutoConfiguration @ComponentScan*

**public** **class** Application {

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

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

}

}

|  |
| --- |
| [Note] |
| @SpringBootApplication also provides aliases to customize the attributes of @EnableAutoConfiguration and @ComponentScan. |
| [Note] |
| None of these features are mandatory and you may choose to replace this single annotation by any of the features that it enables. For instance, you may not want to use component scan in your application:  **package** com.example.myapplication;  **import** org.springframework.boot.SpringApplication;  **import** org.springframework.context.annotation.ComponentScan  **import** org.springframework.context.annotation.Configuration;  **import** org.springframework.context.annotation.Import;  *@Configuration*  *@EnableAutoConfiguration*  *@Import({ MyConfig.class, MyAnotherConfig.class })*  **public** **class** Application {  **public** **static** **void** main(String[] args) {  SpringApplication.run(Application.**class**, args);  }  }  In this example, Application is just like any other Spring Boot application except that @Component-annotated classes are not detected automatically and the user-defined beans are imported explicitly (see @Import). | |

* + 1. 运行springboot应用

直接运行main方法、用maven命令、运行jar包。

* + 1. 定制Banner
       1. 修改springboot启动图案

在src/main/resources 下新建一个banner.txt文件，里面输入你需要的启动字符，如：

=================springboot2 自定义banner==============

= =

= HELLO SPRINGBOOT 2 =

= 这是自定义的springboot启动banner =

==================================================

关闭启动banner

在启动类里启动是用builder.bannerMode(Banner.Mode.OFF)进行关闭

// 使用 SpringApplicationBuilder 启动项目  
SpringApplicationBuilder builder = new SpringApplicationBuilder(Application.class);  
//修改Banner的模式为OFF  
// builder.bannerMode(Banner.Mode.OFF).run(args);  
  
builder.run(args);

* + 1. Springboot配置文件

# springboot 全局配置文件，作用是对一些默认配置的配置值进行修改

在src/main/recoueces 下新建application.properties文件进行配置

如：

#修改项目访问端口  
server.port=80  
#修改项目访问名称  
server.servlet.context-path=/springboot2

* + 1. 外部配置

Spring boot 允许使用properties文件、yml文件或者命令行参数作为外部配置。

* + - 1. 命令行参数配置

比如在以jar包的方式运行时可以指定项目访问端口

java -jar xxx.jar –server.port=80

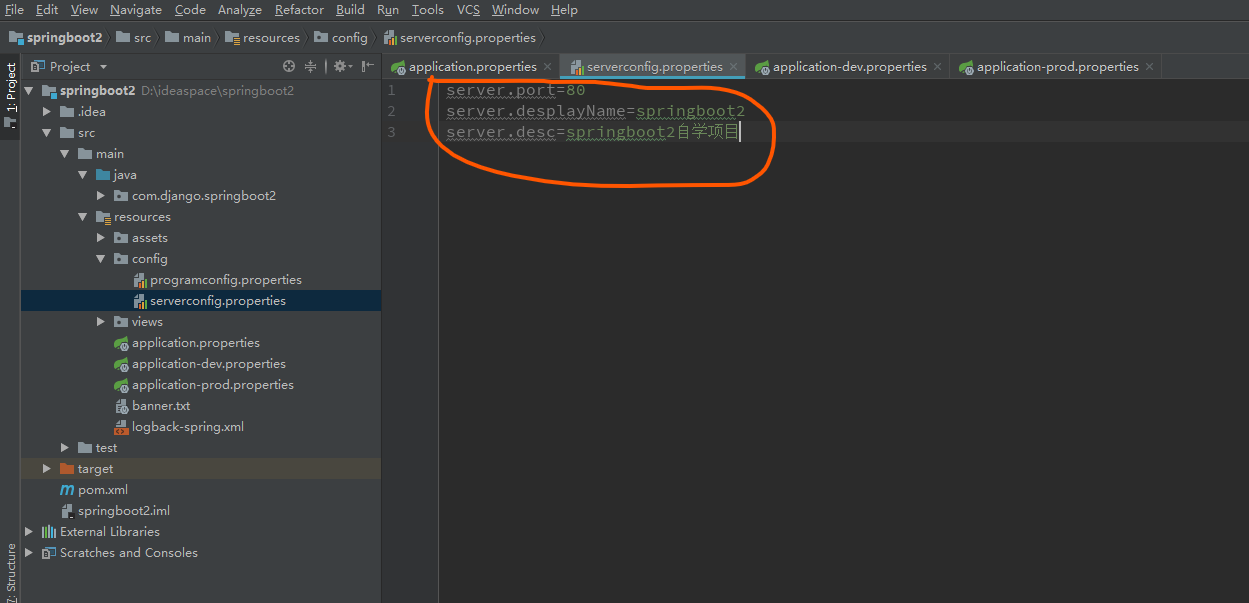
* + - 1. 基于properties文件的配置

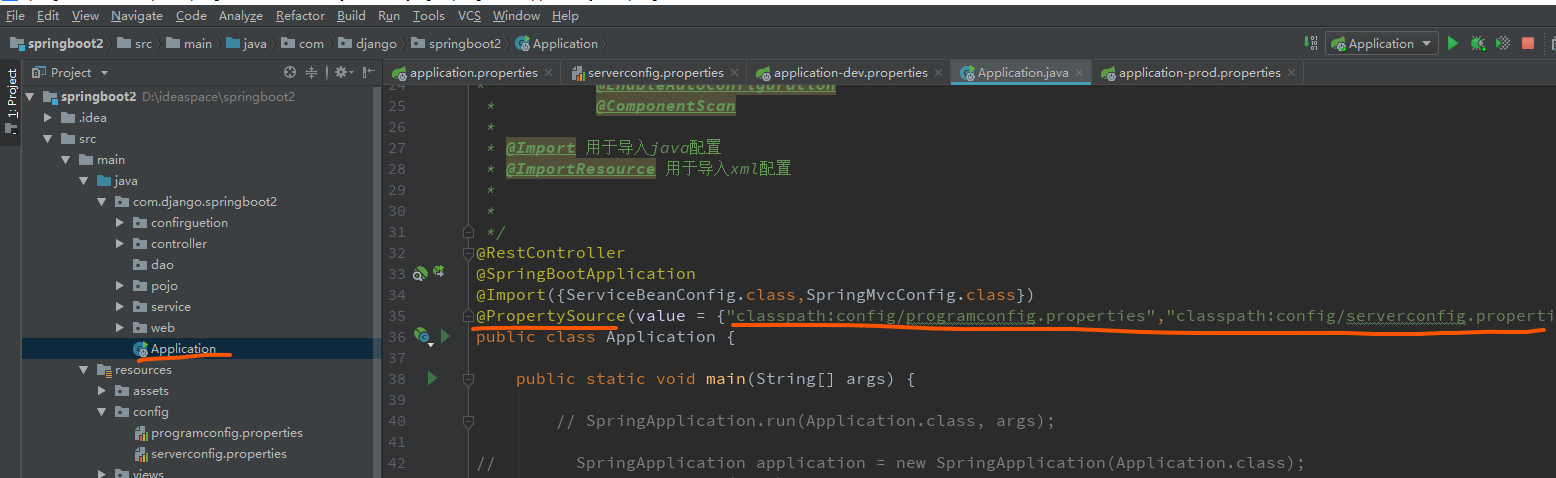
1：在properties文件中配置属性，用@PropertyResource指定配置文件的位置（在配置类上指定），在项目总用@Value使用

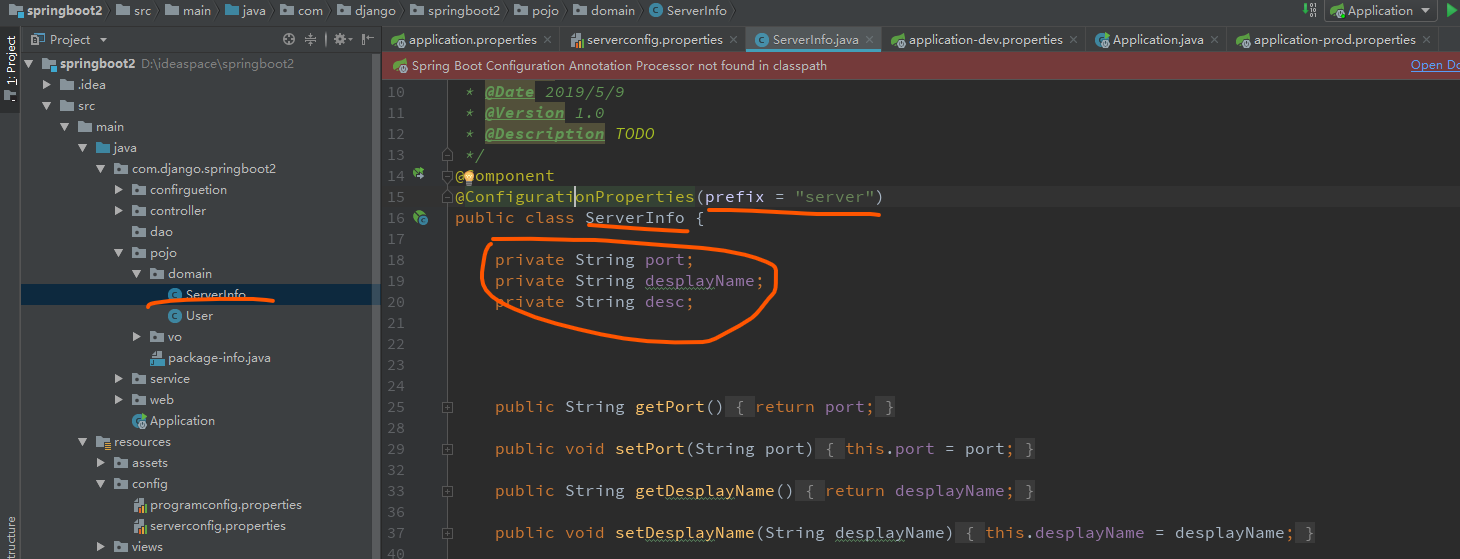
（如果是在springboot的配置文件:application.properties中写的，在不用@PropertyResource指定，直接使用即可）

2：可以在properties文件中配置好属性，将这些属性值注入到对应的Java对象中进行使用。

如下截图







* + 1. 日志配置

Springboot日志配置可以直接使用springboot默认的日志配置文件，也可以使用自己的配置文件（springboot使用logback作为日志）

1：使用springboot本身默认的配置

在springboot配置文件application.properties里做相关配置（springboot默认只做了输出到控制台的配置，需要输出日志到文件的话，需要添加相关配置。比如下面的配置）

#日志配置:使用springboot的默认日志配置文件  
#logging.path=D:/springboot2/log/  
#logging.exception-conversion-word=springboot2.log  
#logging.level.root=info  
#日志配置:使用自定义的配置文件

2：使用自己的logback配置文件

在classpath下面（src/main/resources/）添加loback配置文件

logback-spring.xml：

<?xml version="1.0" encoding="UTF-8"?>  
<configuration scan="true" scanPeriod="60 seconds" debug="false">  
 <contextName>springboot2</contextName>  
 <property name="LOGFILE\_PATH" value="D:/springboot2/log" />  
  
 <appender name="STDOUT" class="ch.qos.logback.core.ConsoleAppender">  
 <encoder>  
 <pattern>%d{HH:mm:ss.SSS} %contextName [%thread] %-5level %logger{36} - %msg%n</pattern>  
 </encoder>  
 </appender>  
   
 <appender name="FILE" class="ch.qos.logback.core.rolling.RollingFileAppender">  
 <file>${LOGFILE\_PATH}/springboot2.log</file>  
  
 <rollingPolicy class="ch.qos.logback.core.rolling.TimeBasedRollingPolicy">  
 <fileNamePattern>springboot2\_%d{yyyy-MM-dd}.log</fileNamePattern>  
 </rollingPolicy>  
 <encoder>  
 <pattern>%d{HH:mm:ss.SSS} %contextName [%thread] %-5level %logger{36} - %msg%n</pattern>  
 </encoder>  
 </appender>  
  
 <root level="INFO">  
 <appender-ref ref="STDOUT" />  
 <appender-ref ref="FILE" />  
 </root>  
</configuration>

在application.properties里添加如下配置

#日志配置:使用自定义的配置文件  
logging.config=classpath:logback-spring.xml

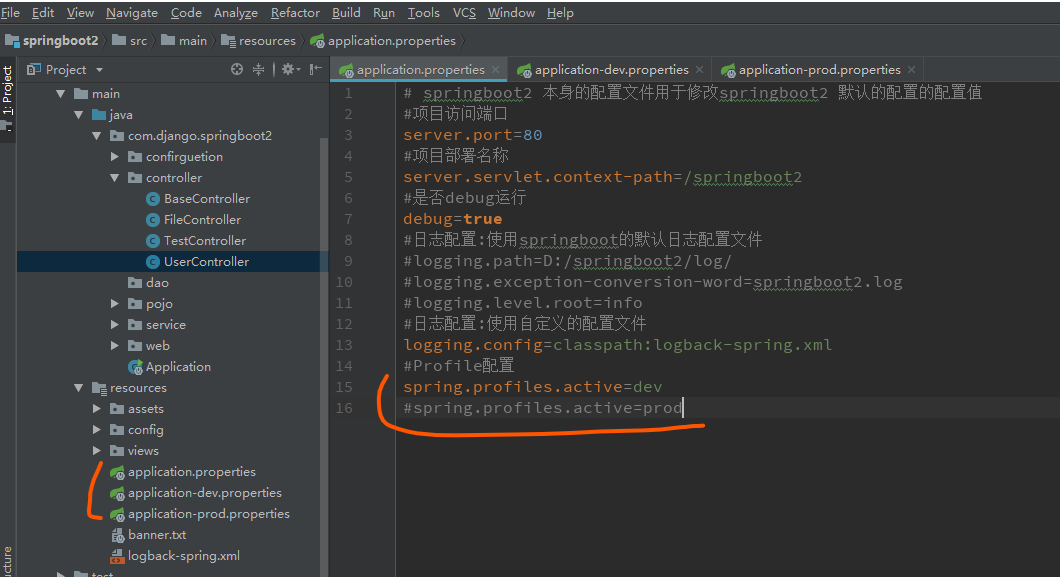
参考资料地址：

<https://blog.csdn.net/flysun3344/article/details/80555746>

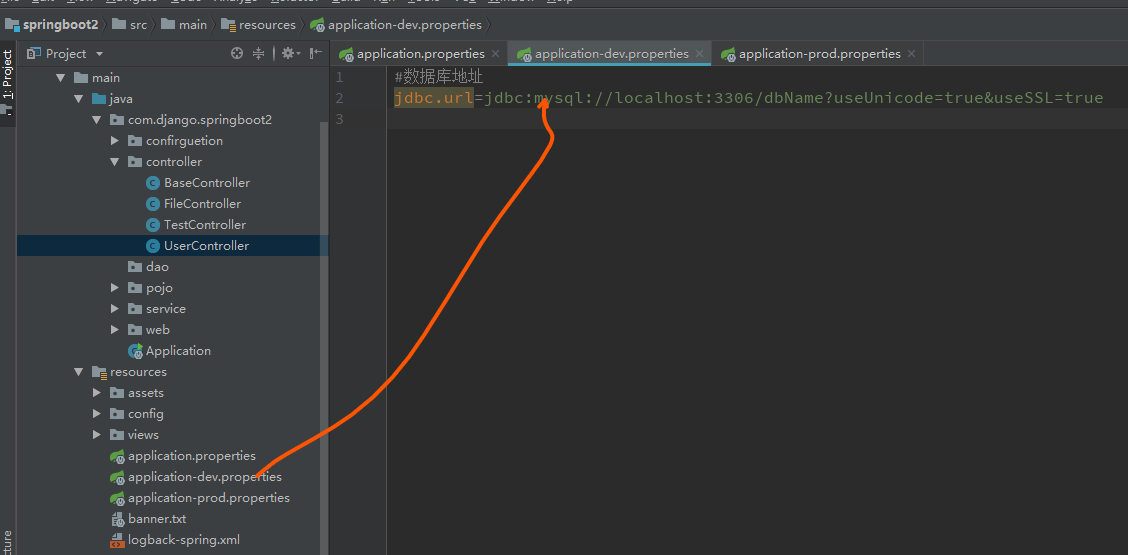
* + 1. Profile配置

Profile是spring针对不同环境对不同的配置提供支持的配置。全局的Profile配置使用application-{profile}.properties(如：application-dev.properties)进行配置。通过在application.properties中设置spring.profile.active=prod来指定活动的profile。比如开发环境（dev）和生产环境（prod）的数据库地址区分（只是 为了说明这个问题）。

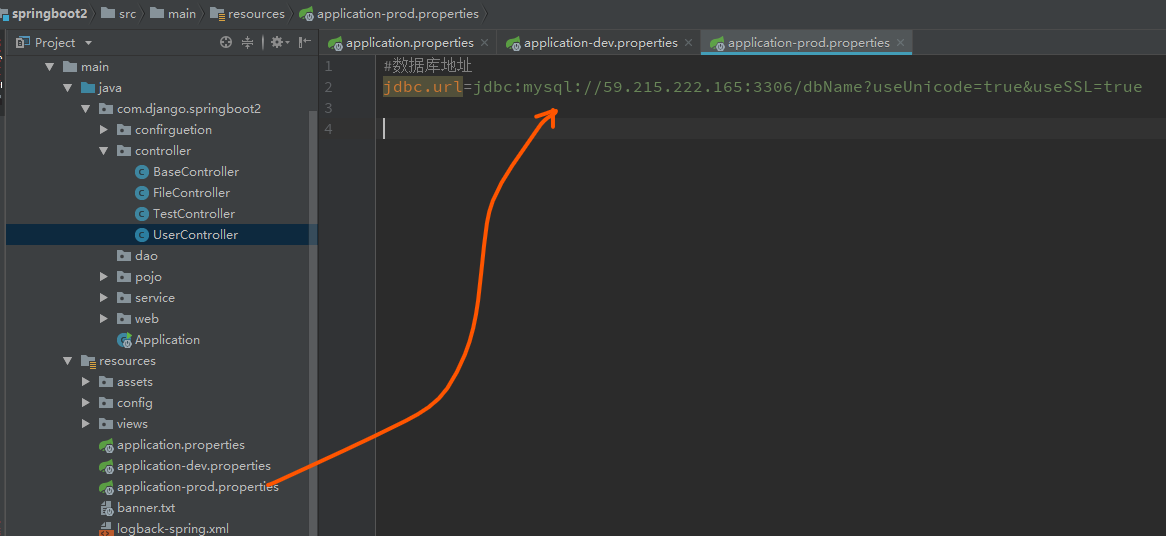
application.properties里的配置



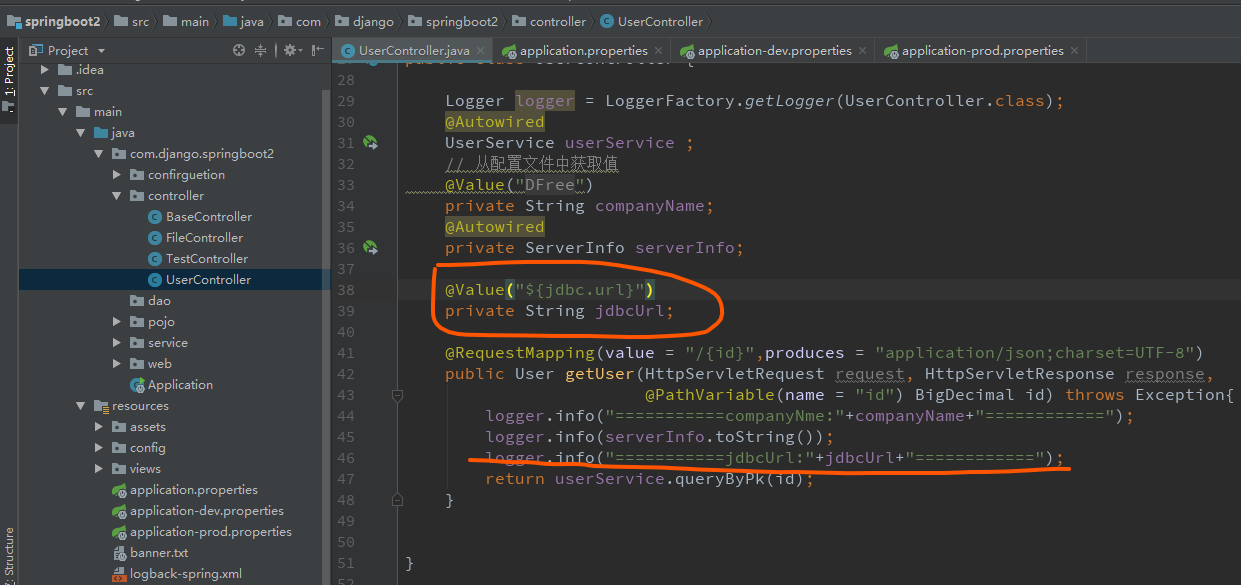
application-dev.properties里的配置



application-prod.properties里的配置



引用：



* + 1. Springboot的自动配置运行原理

Spring有@Conditional的条件注解，使用这个条件注解可以值指定的条件下在spring容器中创建对应条件成立的bean。Springboot自动配置使用了这个特点进行了自动配置：

从@EnableAutoConfigration 说起：

@Target(ElementType.*TYPE*)  
@Retention(RetentionPolicy.*RUNTIME*)  
@Documented  
@Inherited  
@SpringBootConfiguration  
@EnableAutoConfiguration  
@ComponentScan(excludeFilters = {  
 @Filter(type = FilterType.*CUSTOM*, classes = TypeExcludeFilter.class),  
 @Filter(type = FilterType.*CUSTOM*,  
 classes = AutoConfigurationExcludeFilter.class) })  
public @interface SpringBootApplication {

这个注解的核心功能有@EnableAutoConfiguration提供

@EnableAutoConfiguration注解如下

@Target(ElementType.*TYPE*)  
@Retention(RetentionPolicy.*RUNTIME*)  
@Documented  
@Inherited  
@AutoConfigurationPackage  
@Import(AutoConfigurationImportSelector.class)  
public @interface EnableAutoConfiguration {

这里的关键功能是@Import注解导入了EnableAutoConfigurationImportSelector

EnableAutoConfigurationImportSelector中使用SpringFactorirs.loadFactoryNames方法扫描具有META-INF/spring.factories文件的jar包，而我们的spring-boot-autoconfiguration-2.1.4.REALEASE就有一个spring.factories文件。

这个为念声明的配置如下：

# Auto Configuration Import Filters  
org.springframework.boot.autoconfigure.AutoConfigurationImportFilter=\  
org.springframework.boot.autoconfigure.condition.OnBeanCondition,\  
org.springframework.boot.autoconfigure.condition.OnClassCondition,\  
org.springframework.boot.autoconfigure.condition.OnWebApplicationCondition  
  
# Auto Configure  
org.springframework.boot.autoconfigure.EnableAutoConfiguration=\  
org.springframework.boot.autoconfigure.admin.SpringApplicationAdminJmxAutoConfiguration,\  
org.springframework.boot.autoconfigure.aop.AopAutoConfiguration,\  
org.springframework.boot.autoconfigure.amqp.RabbitAutoConfiguration,\  
org.springframework.boot.autoconfigure.batch.BatchAutoConfiguration,\

里面# Auto Configure

里配置的AutoConfigration文件一般都有下面的条件注解

@ConditionalOnBean:当容器里有指定的Bean的条件下

@ConditionalOnClass:当类路径下有指定的类的条件下

@ConditianalOnMissingBean: 当容器里没有有指定的Bean的条件下

@ConditionalOnMissClass: 当类路径下没有指定的类的条件下

@ConditiaonalOnProperty:指定的属性是否有指定的值

等等。。。。。

然后在对应的AutoConfigration中根据相应的条件创建不同的Bean

* + - 1. 自己写一个自动配置的bean

（srpingboot本来就带有这个配置的）

在使用web.xml的时候我们配置编码过滤器使用的配置：

\*

\* <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>forcencoding</param-name>

\* <param-value>true</param-value>

\* </init-param>

\* </filter>

\* <filter-mapping>

\* <filter-name>characterEncodingFilter</filter-name>

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

\* </filter-mapping>

\*

\* 现在我们让springboot自动为我们配置CharacterEncodingFilter 这个bean,来实现编码

\* 这个自动配置还要满足连个条件

\* 1：能配置CharacterEncodingFilter 这个bean

\* 2：能配置encoding和forceEncoding这两个参数

\* 我们这样：

\* 在httpencoding.properties文件中做属性配置：

\* spring.http.encoding.chartset=UTF-8

\* spring.http.encoding.force=true

\* 将这个配置板顶到一个Java实例上（HttpEncodingProperties）

\* 在配置文件中注入这个实例：

\* @Autowired

\* private HttpEncodingProperties httpEncodingProperties;

\* 在配置文件中创建过滤器CharacterEncodingFilter，设置属性

\*

\* CharacterEncodingFilter filter = new CharacterEncodingFilter();

\* filter.setEncoding(this.httpEncodingProperties.getCharset().name());

\* filter.setForceEncoding(this.httpEncodingProperties.isForce());

\* return filter;

\*

\* 以下是所用到的相关注解的说明

\* @EnableConfigurationProperties

\*

\* 该注解是用来开启对@ConfigurationProperties注解配置Bean的支持。

\* 也就是@EnableConfigurationProperties注解告诉Spring Boot

\* 能支持@ConfigurationProperties。如果不指定会看到如下异常:

\*

\* 当然，也可以不用@EnableConfigurationProperties 这个注解.

\*

\* 而配合@Componnet 注解和@ConfigurationProperties使用:

\* @Component

\* @ConfigurationProperties(prefix = "spring.http.encoding")

\* public class HttpEncodingProperties {

\* }

\* 用了@EnableConfigurationProperties 这个注解就不用在对于类上加@Componnet 注解了

\*

\* @ConditionalOnClass

\* 条件注解：在classpath路径有对应的类时条件成立的意思

\*

\* @ConditionalOnProperty

\* 当配置文件设置了spring.http.encoding=enabled的情况下成立，如果没有设置则默认为true,即条件符合

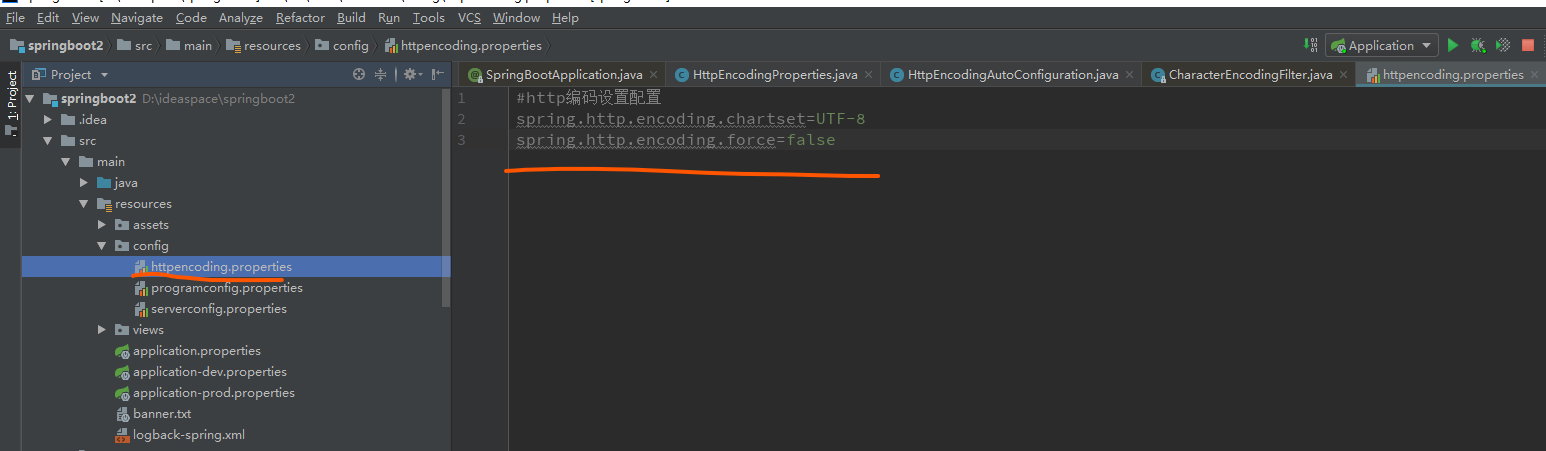
\*

\* @ConditionalOnMissingBean

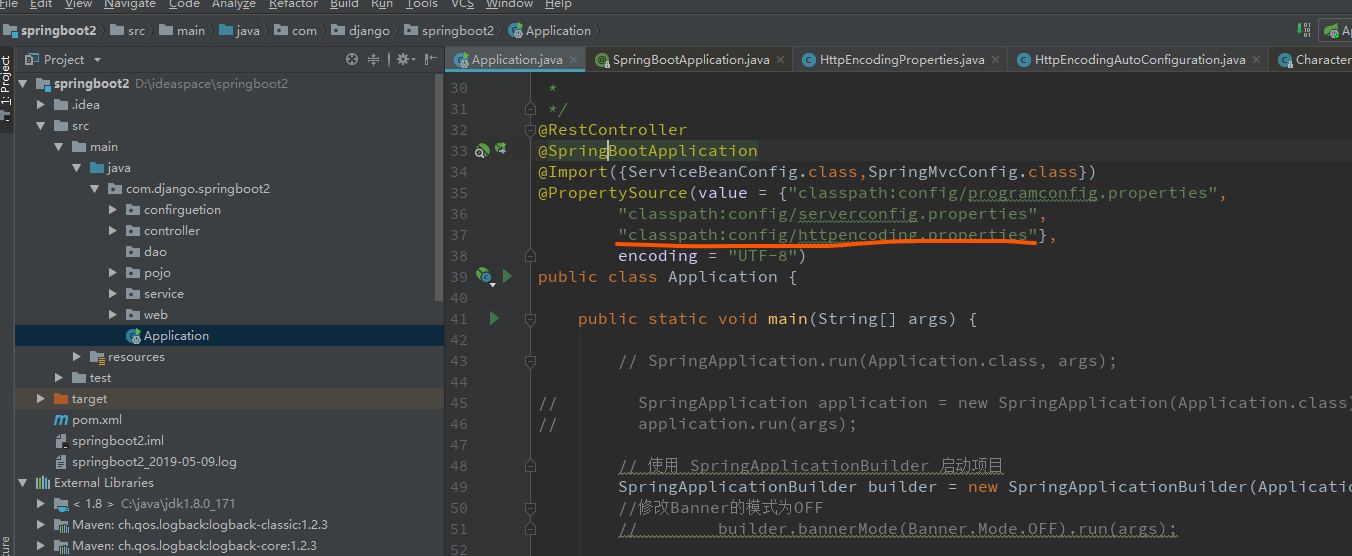
\* 当容器里没有对应的bean时条件成立

\* （这里的意思就是当容器里没有CharacterEncodingFilter 这个bean是创建这个bean）

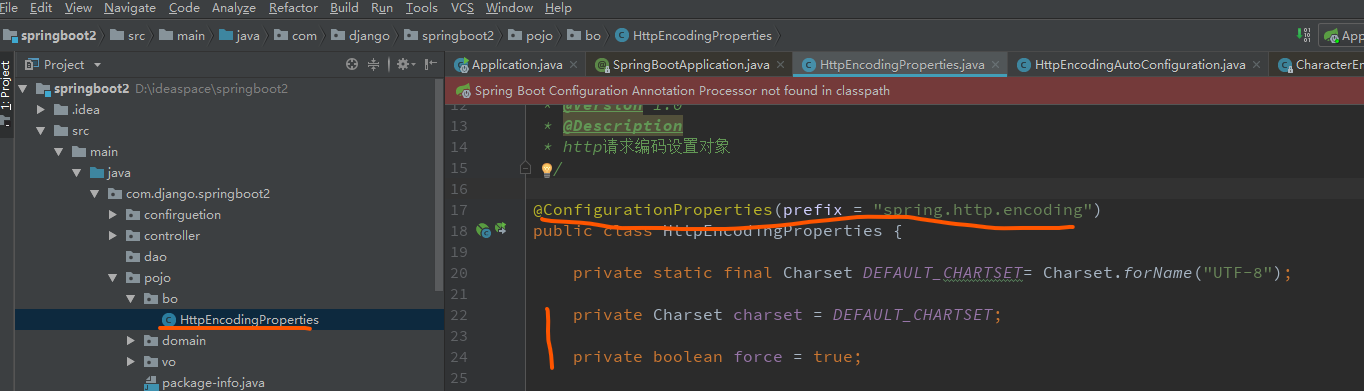
配置文件：



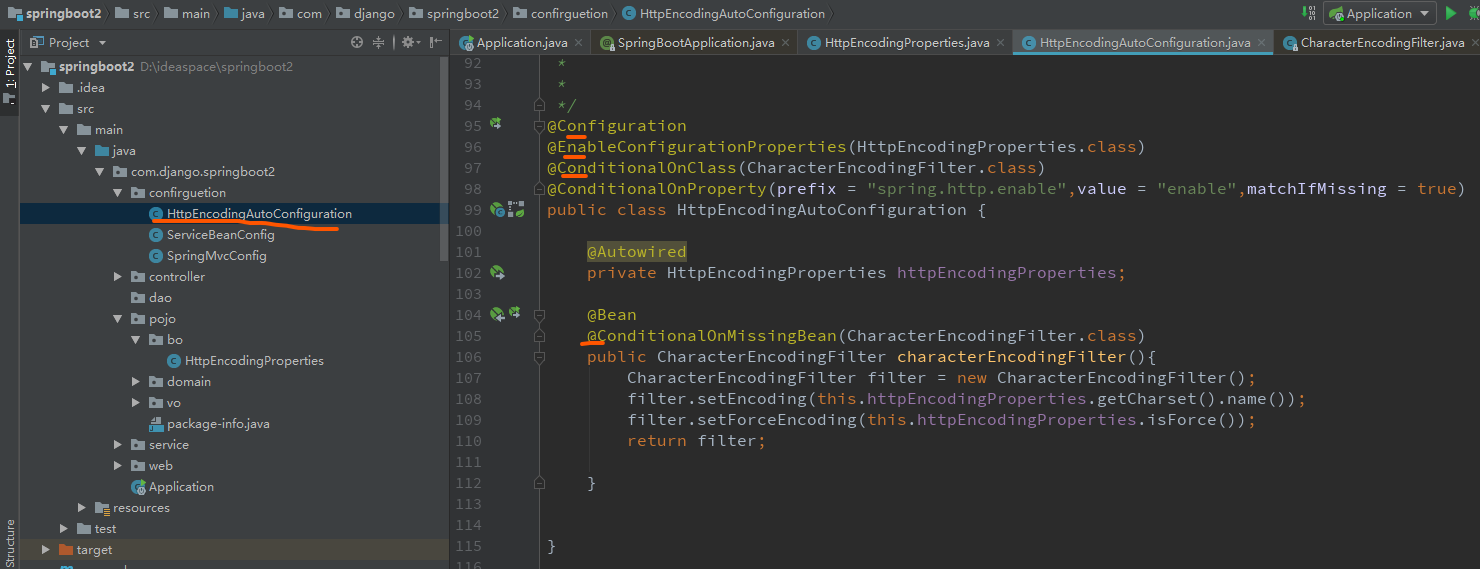
引入配置文件：



将配置属性封装到实体bean中：



使用配置参数让springboot在判断类路径下有编码过滤器的时候且spring容器里没有对应的bean的时候创建这个编码过滤器



这样我们就完成了编码过滤器的配置，springoot在坚持类路径下有这个过滤器就会为我们创建这个过滤器

* 1. Springboot web开发
     1. Web支持配置（默认的Thymeleaf模板支持）

Springboot开发，pom.xml导入spring-boot-starter-web

即可做web接口开发了，springboot为我们自动做了大部分web开发需要的配置，具体做了那些配置可以查看

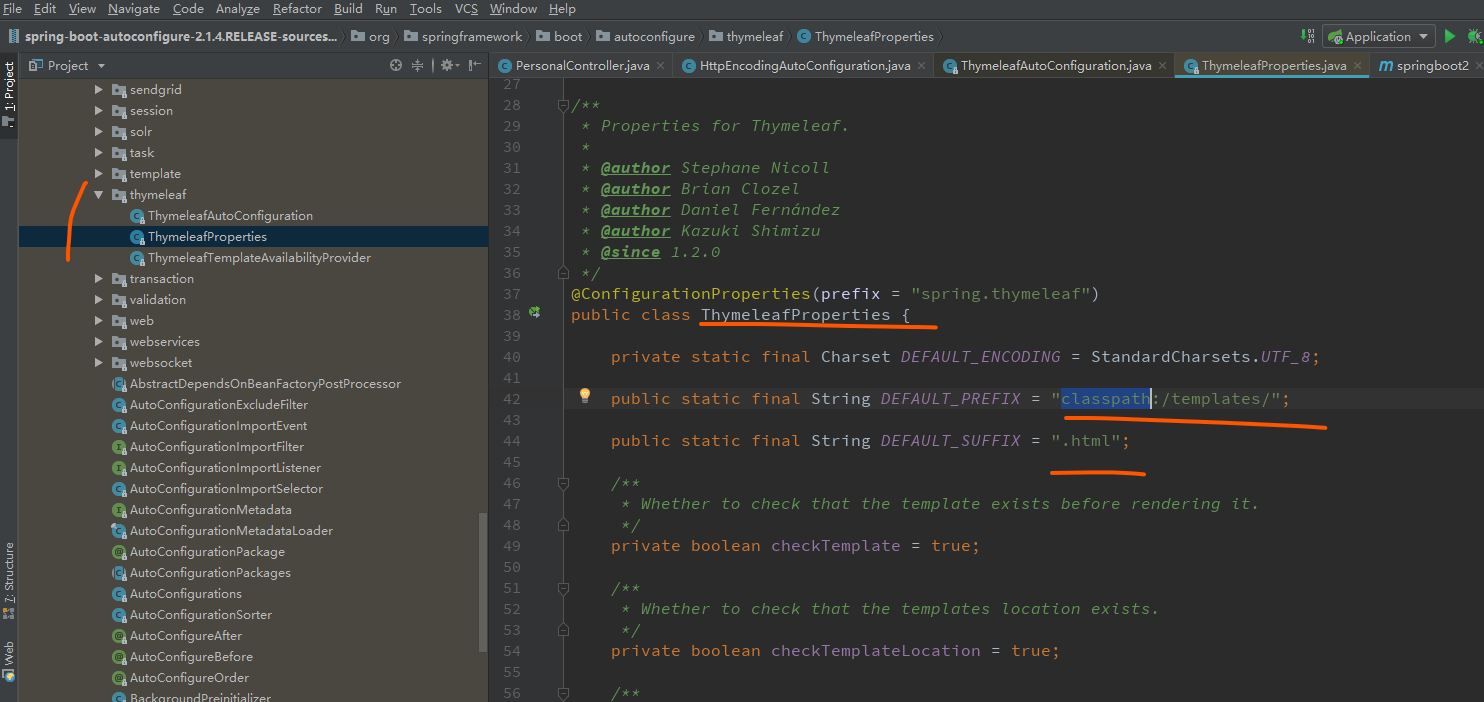
org.springframework.boot.autoconfigure.web这个包下的相关配置

比如编码设置

*org.springframework.boot.autoconfigure.web.servlet.HttpEncodingAutoConfiguration*

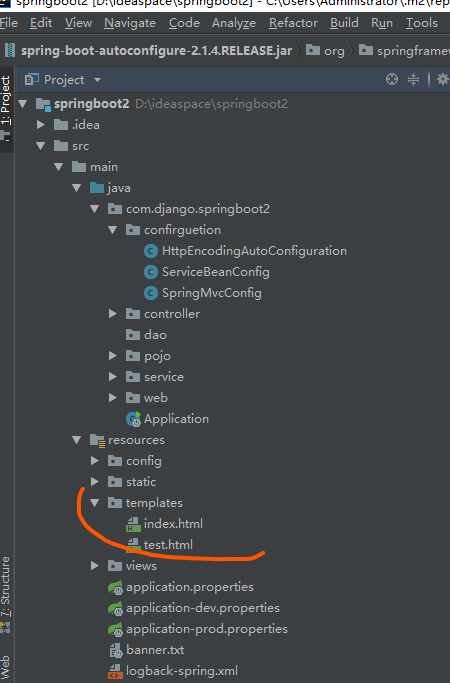
springboot支持很多模板引擎来作为mvc开发的view视图

默认的模板引擎是Thymeleaf（springboot不推荐使用jsp）。springboot已经为我们配置好了使用Thymeleaf模板的相关配置，具体配置查看：



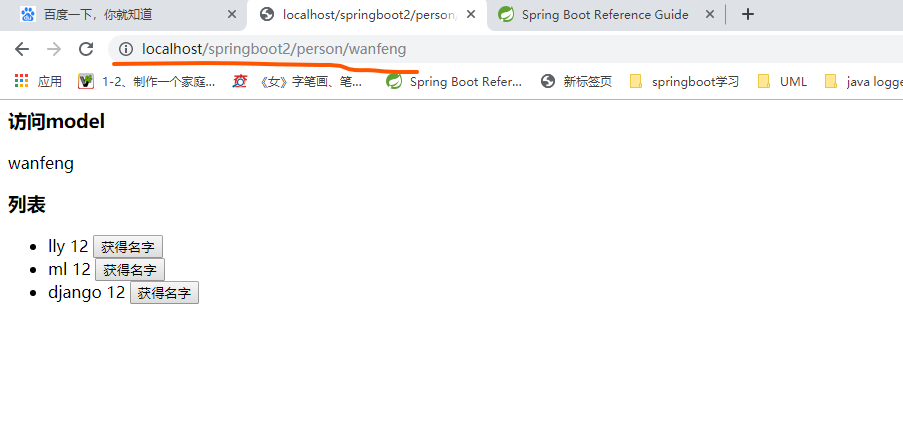
如上，我们在

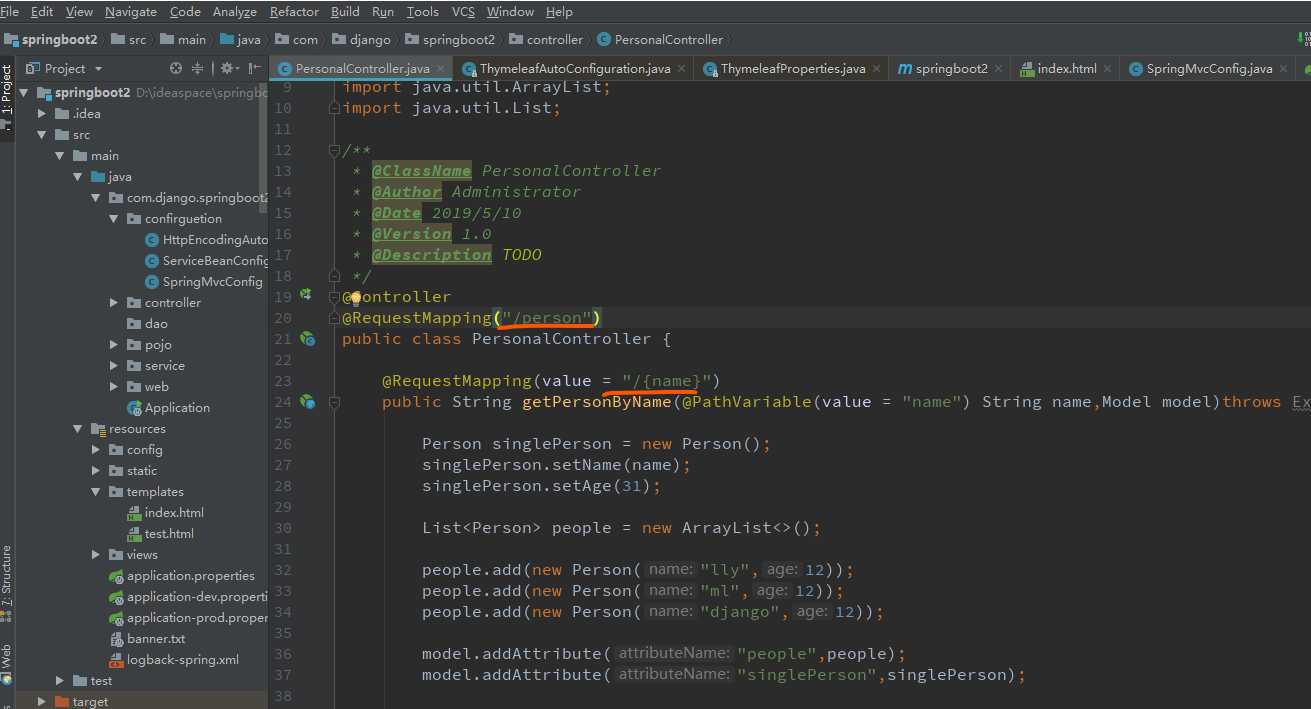
src/main/resources/下建立templates文件夹，加模板文件放在这个文件夹下



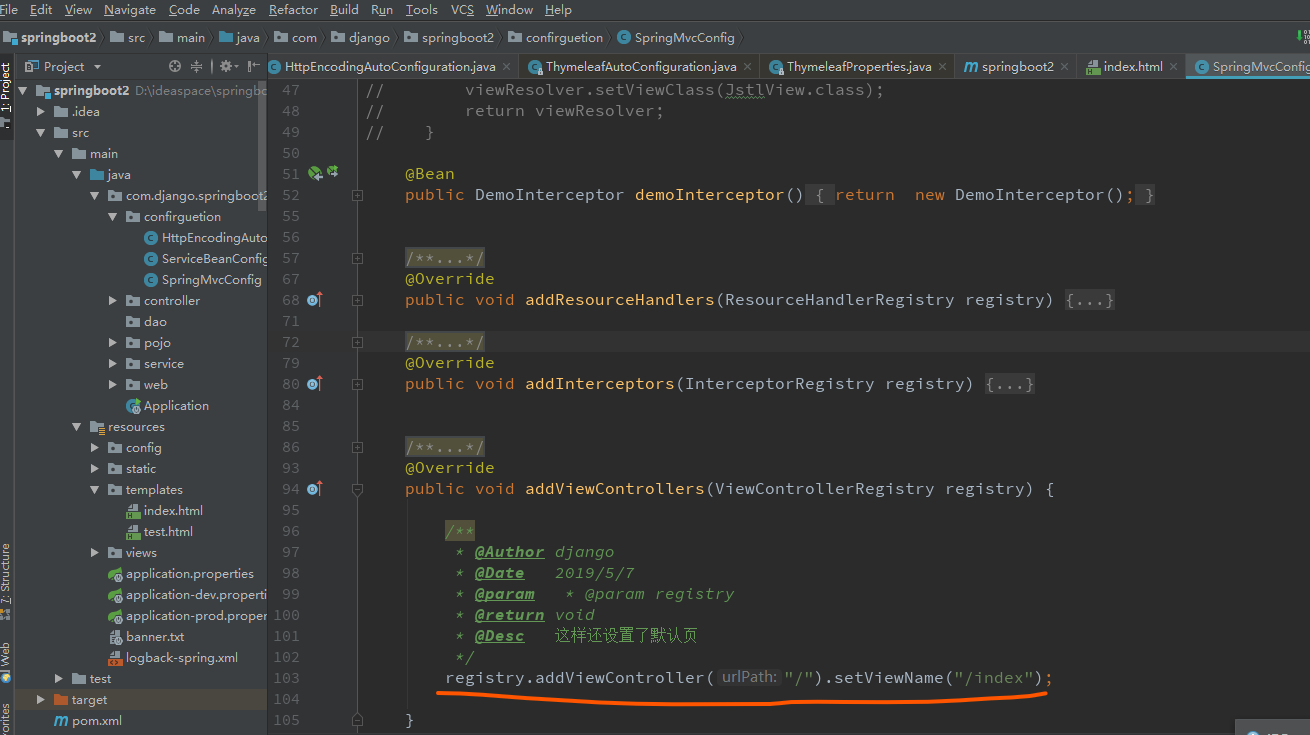
这样如默认配置里，当我们在controller里返回试图是就会在这个文件夹下去找相应的模板文件作为视图来渲染返回了。

如下访问controller返回视图

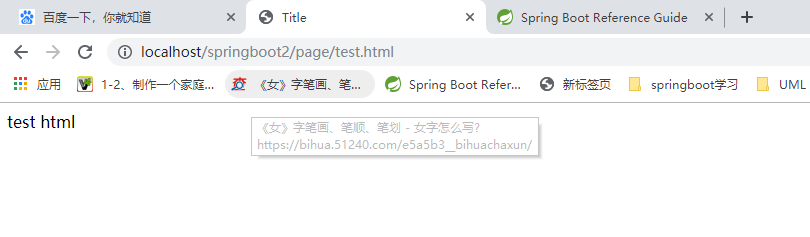




当然，如果我们要直接访问一个静态的html页面的话，需要配置静态资源访问了：



如下访问静态的html页面



* + 1. Web相关配置
       1. Springboot提供的自动配置
          1. 自动配置的VeiwResolver

1:ContentNegotiatingViewResolver

这是个springMVC提供的特殊viweResolver,它用来代理不同的viewResolver来

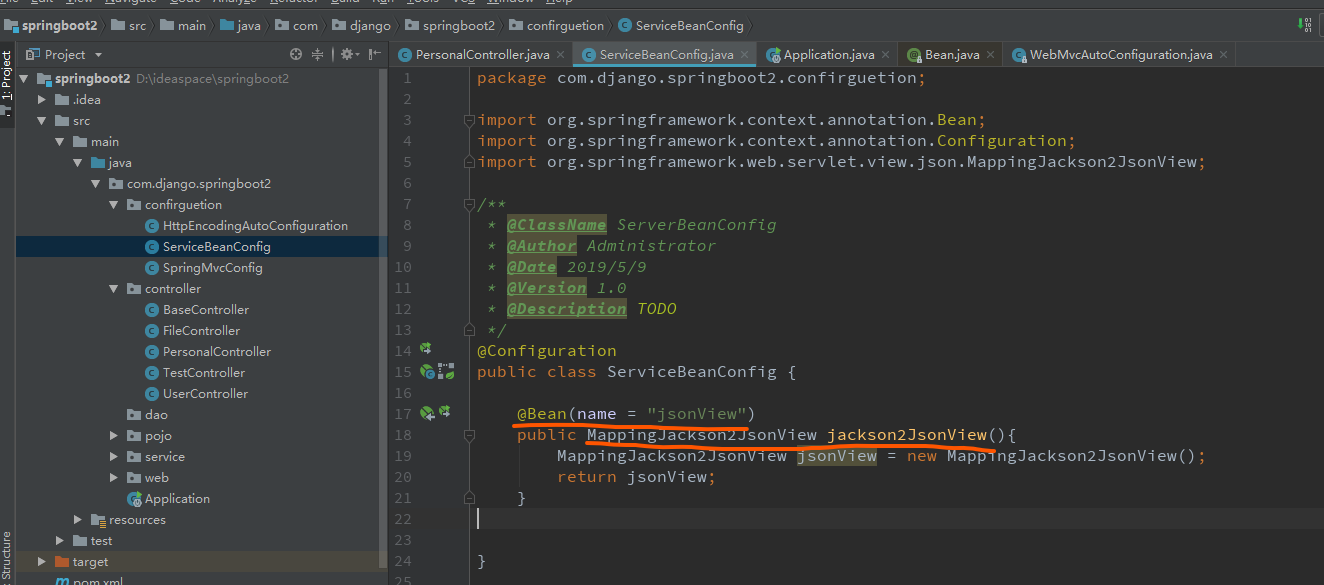
处理不同的view.有最高优先级

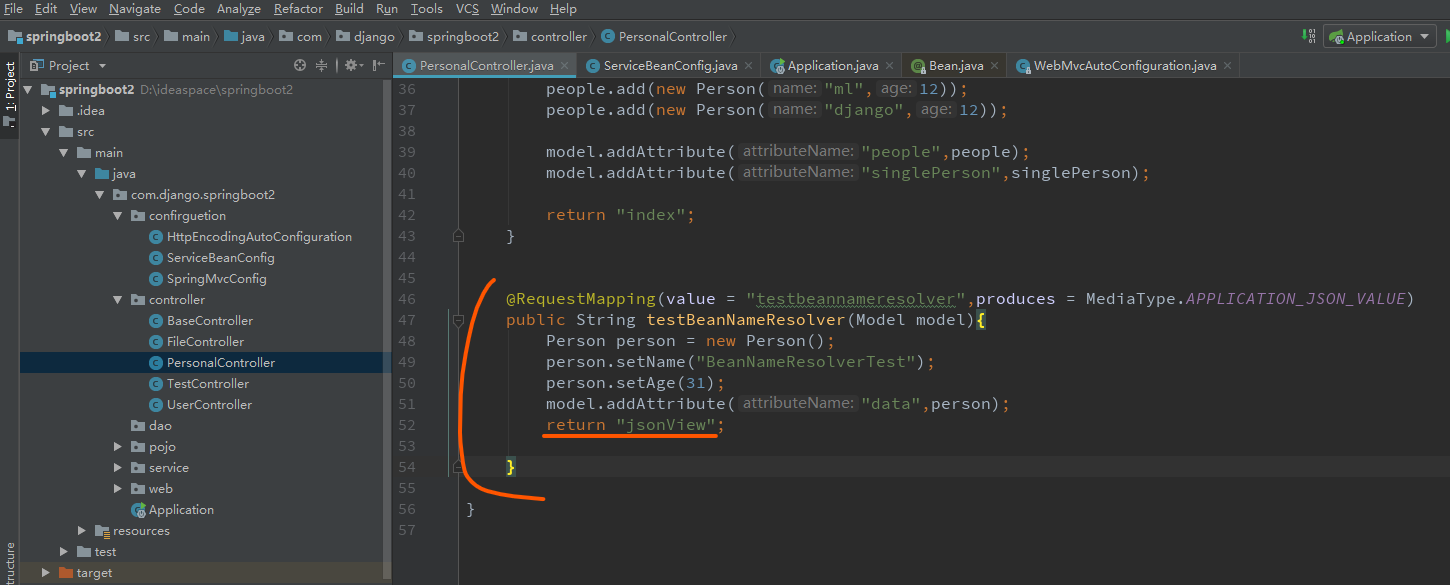
2：BeanNameViewResolver

在控制器中的一个方法的返回值的字符串（视图名称）会根据BeanNameViewResolver去查找Bean的名称为返回的字符串的view来渲染视图。

比如图：注册了一个叫jsonView的ViewResolver，在controller中返回jsonView字符串，就会用名字为jsonView这个视图解析器去解析这个视图的返回，至于解析成什么样子，那是视图解析器实现的问题。

视图解析器注册





注意controller中返回的字符串的名字和注册的视图解析器的名字一样

InteralResourceViewResolver

默认的视图解析器，就是返回的字符串在src/main/resource/template/下去寻找对应的视图

* + 1. 自动配置的静态资源

org.springframework.boot.autoconfigure.web.servlet.WebMvcAutoConfiguration.WebMvcAutoConfigurationAdapter#addResourceHandlers

addResourceHandlers方法中定义了以下静态资源的自动配置

1：类路径下的件

把类路径下的/static、/public、/resources和/META-INF/resources文件夹下的静态资源文件直接映射为/\*\*，可以通过[http://localhost:8080/\*\*](http://localhost:8080/**)来访问。

2:webjar

Webjar就是我们将我们常用的脚本框架封装在jar包中的jar包。

把webjar的/META-INF/resources/webjar下的静态文件映射为/webjar/\*\*，可通过[http://localhost:8080/webjar/\*\*](http://localhost:8080/webjar/**)来访问。

静态资源访问配置总结：

如上的，

1：我们可以将前端的静态资源放在类路径下的/static下面

2：把前端静态资源公用的静态资源放在/public下面

3：把前端静态资源配置相关的的静态资源放在/resources下面

4：把项目相关的描述信息静态资源放在/META-INF/resources下面

5：把控制器返回需要的模板文件（如thymeleaf、jsp模板页面）放在

src/main/resources/templates/下面

* + 1. 自动配置的Formatter和Converter

只要注册实现了Converter、GenericConverter、Formatter接口的Bean接口使用。

静态首页的支持

把静态的index.html文件放置在如下目录即可

classpath:/META-INF/resources/index.html

classpath:/resources/index.html（注意，不是IDEA maven自建的resources目录哦）

classpath:/static /index.html

classpath:/public/index.html

* + 1. 接管springboot的web配置

如果springboot提供的SpringMVC默认配置不符合自己的需求，则可以通过一个配置类来实现WebMvcConfigurer接口来自己完全实现（注解加上EnableWebMvc注解）

在使用默认配置的情况下扩展或修默认配置，通过一个配置类来实现WebMvcConfigurer接口，但是部加EnableWebMvc注解来扩展

* + 1. 注册servlet、filter、listener

在Servlet 3.0之前都是使用web.xml文件进行配置，需要增加Servlet、Filter或者Listener都需要在web.xml增加相应的配置。Servlet 3.0之后可以使用注解进行配置Servlet、Filter或者Listener；springboot也提供了使用代码进行注册Servlet、Filter或者Listener。所以springboot有两种方式进行Servlet、Filter或者Listener配置。

方式一：使用注解

（1）注册Servlet

使用@WebServlet注册，需要在Servlet类上使用该注解即可，但是需要在@Configuration类中使用Spring Boot提供的注解@ServletComponentScan扫描注册相应的Servlet。

（2） 注册Filter

使用@WebFilter注册，需要在Filter类上使用该注解即可，但是需要在@Configuration类中使用Spring Boot提供的注解@ServletComponentScan扫描注册相应的Filter。

（3）注册Listener

使用@WebListener注册，需要在Filter类上使用该注解即可，但是需要在@Configuration类中使用Spring Boot提供的注解@ServletComponentScan扫描注册相应的Listener。

**方式二：使用spring提供的方式**

（1）注册Servlet

使用ServletRegistrationBean注册只需要在@Configuration类中加入类似以下的代码

@Bean

public ServletRegistrationBean regServlet() {

    ServletRegistrationBean userServlet= new ServletRegistrationBean();

    userServlet.addUrlMappings("/servlet");

    userServlet.setServlet(new UserServlet());

    return userServlet;

}

（2）  注册Filter

使用FilterRegistrationBean注册Filter，只需要在@Configuration类中加入类似以下的代码：

@Bean

  public FilterRegistrationBean regFilter() {

    FilterRegistrationBean userFilter = new FilterRegistrationBean();

    userFilter .addUrlPatterns("/\*");

    userFilter .setFilter(new UserFilter ());

    return userFilter ;

}

3）注册Listener

使用ServletListenerRegistrationBean注册Listener只需要在@Configuration类中加入类似以下的代码

@Bean

  public ServletListenerRegistrationBean<LoginSessionListener> regServletListener() {

    ServletListenerRegistrationBean<LoginSessionListener> loginSessionListener= new ServletListenerRegistrationBean<LoginSessionListener>();

    loginSessionListener.setListener(new LoginSessionListener());

    return loginSessionListener;

}

* + 1. Servlet容器配置

Springbot默认内嵌tomcat为servlet容器。

关于servlet容器的配置都在org.springframework.boot.autoconfigure.web.ServerProperties配置类中，我们在application.properties中配置即可。通用的servlet容器配置已server为前缀，而tomcat特有的都以server.tomcat为前缀。

比如：

Servlet容器配置：

#项目访问端口  
server.port=80  
#项目部署名称  
server.servlet.context-path=/springboot2  
server.servlet.session.timeout=7200s  
server.servlet.context-parameters.programName=springboot自学项目

Tomcat配置

#tomcat配置  
server.tomcat.uri-encoding=UTF-8  
server.tomcat.max-threads=200

代码配置tomcat

1.建立一个配置类，加上@Configuration注解

2.添加定制器ConfigurableServletWebServerFactory

3.将定制器返回

@Bean //一定要将这个定制器加入到容器中

public EmbeddedServletContainerCustomizer embeddedServletContainerCustomizer(){

return new EmbeddedServletContainerCustomizer() {

//定制嵌入式的Servlet容器相关的规则

@Override

public void customize(ConfigurableEmbeddedServletContainer container) {

container.setPort(8083);

}

};

}

* + 1. SSL配置
       1. 证书生成

在正确配置完JDK环境变量的前提下，在任意目录，执行命令

keytool -genkey -alias tomcat -storetype PKCS12 -keyalg RSA -keysize 2048 -keystore keystore.p12 -validity 3650

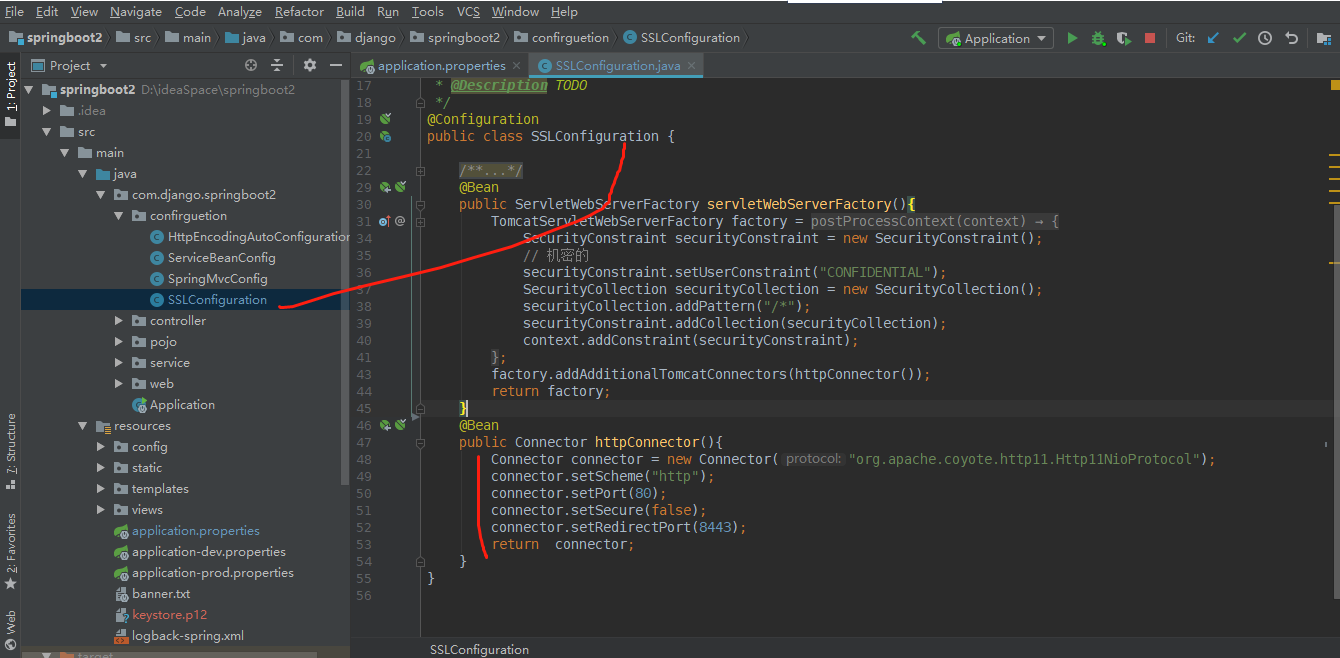
* 在当前目录下会生成一个keystore.p12文件，即证书文件

将证书copy到classpath下。

配置springboot配置文件

#SSL  
#项目访问端口  
server.port=8443  
#keystore路径  
server.ssl.key-store= classpath:keystore.p12  
#ssl密钥库密码  
server.ssl.key-store-password= django!@#159  
#keystore类型  
server.ssl.key-store-type= PKCS12  
#keystore别名  
server.ssl.key-alias= tomcat

http转发到https配置



* + 1. Springboot2 项目Favicon配置

<https://www.cnblogs.com/shamo89/p/8098090.html>

* + 1. WebSocket配置

使用stomp协议实现消息通讯（需要补stomp协议todo，不然很奇怪）