# Springboot

视频教程(哔哩哔哩尚硅谷springboot)：

https://www.bilibili.com/video/BV1Et411Y7tQ?p=77

参考笔记:

<https://cloudlandboy.github.io/myNote/#/backend/springboot/helloworld>

<https://niceseason.github.io/2020/04/18/springboot/#%E4%B8%80-Spring-Boot%E4%B8%8E%E7%BC%93%E5%AD%98>

# 创建项目

## 自行搭建

-使用intellij idea新建maven项目：new>project>maven>next>....

**1、导入spring boot相关的依赖**

<parent>

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

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

<version>2.2.1.RELEASE</version>

<relativePath/>

</parent>

<dependencies>

<dependency>

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

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

</dependency>

</dependencies>

**2、编写一个主程序；启动Spring Boot应用**

package cn.clboy.springboot;

import org.springframework.boot.SpringApplication;

import org.springframework.boot.autoconfigure.SpringBootApplication;

/\*\*

\* @Author cloudlandboy

\* @Date 2019/11/13 下午2:58

\* @Since 1.0.0

\* springBootApplication：标注一个主程序类，表示这个是一个Springboot应用

\*/

@SpringBootApplication

public class HelloWorldMainApplication {

public static void main(String[] args) {

//启动

SpringApplication.run(HelloWorldMainApplication.class, args);

}

}

**3、编写一个Controller**

package cn.clboy.springboot.controller;

import org.springframework.web.bind.annotation.RequestMapping;

import org.springframework.web.bind.annotation.RestController;

/\*\*

\* @Author cloudlandboy

\* @Date 2019/11/13 下午3:05

\* @Since 1.0.0

\* RestController：是spring4里的新注解，是@ResponseBody和@Controller的缩写。

\*/

@RestController

public class HelloController {

@RequestMapping("/hello")

public String hello(){

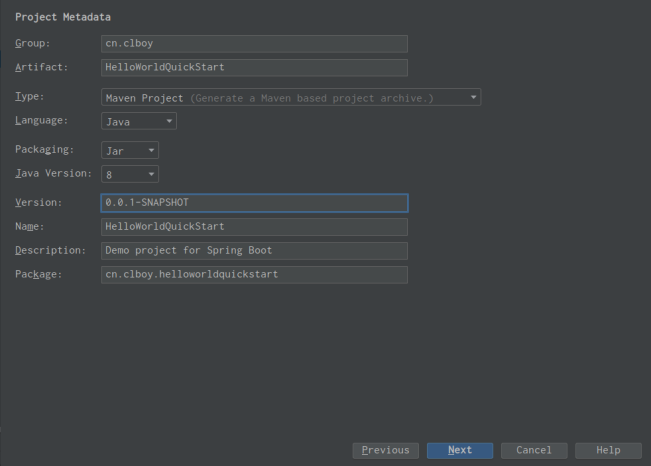
return "hello SpringBoot,this is my first Application";

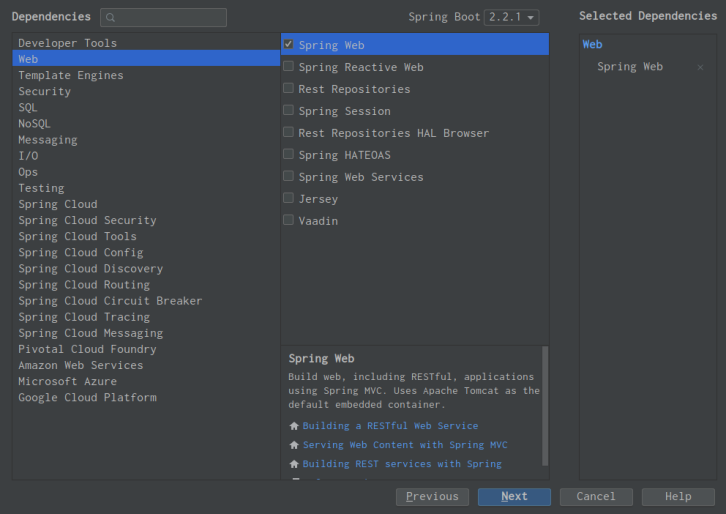
}

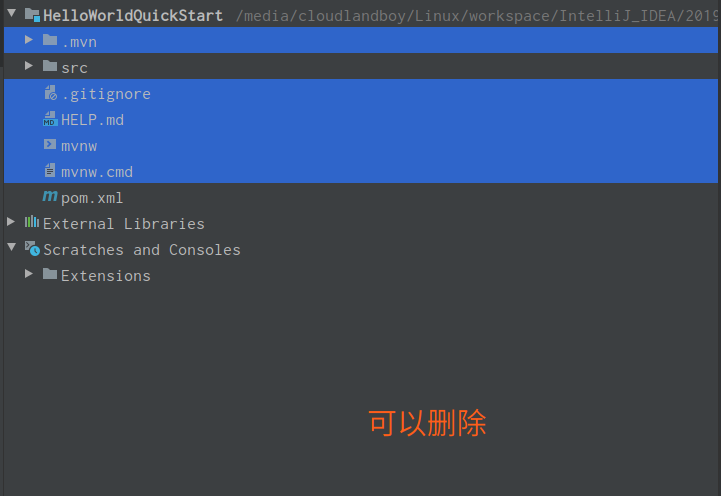
}

## 使用Spring Initializer快速创建Spring Boot项目

1、创建项目时选择Spring Initializr







# 配置文件

SpringBoot使用一个全局的配置文件，配置文件名application是固定的；

* application.properties
* application.yml
* application.yaml

## YAML

### 语法

以空格的缩进来控制层级关系；只要是左对齐的一列数据，都是同一个层级的

次等级的前面是空格，不能使用制表符(tab)

冒号之后如果有值，那么冒号和值之间至少有一个空格，不能紧贴着

### 字面量：普通的值（数字，字符串，布尔）

K : v

字符串默认不用加上单引号或者双引号；

""：双引号；不会转义字符串里面的特殊字符；特殊字符会作为本身想表示的意思

eg： name: "zhangsan \n lisi"：输出；zhangsan 换行 lisi

''：单引号；会转义特殊字符，特殊字符最终只是一个普通的字符串数据

eg： name: ‘zhangsan \n lisi’：输出；zhangsan \n lisi

### 对象、Map（属性和值）：

k: v在下一行来写对象的属性和值的关系；注意缩进

**person**:  
 **name**: chq  
 **age**: 12

行内写法:

**person**: {**name**: 张三,**gender**: 男,**age**: 22}

### 数组（List、Set）

**fruits**:  
 - 苹果  
 - 桃子  
 - 香蕉

行内写法:

**fruits**: [苹果,桃子,香蕉]

## properties配置文件写法

person.name=李四  
person.age=34  
person.birth=1986/09/12  
person.boss=true  
person.gender=女  
person.lists=cat,dog,pig  
person.maps.k1=v1  
person.maps.k2=v2  
person.pet.name="小黑"  
person.pet.age=10

yaml和properties配置文件同时存在，properties配置文件的内容会覆盖yaml配置文件的内容

## 配置文件值注入

导入依赖：

<!--导入配置文件处理器，配置文件进行绑定就会有提示-->  
<dependency>  
 <groupId>org.springframework.boot</groupId>  
 <artifactId>spring-boot-configuration-processor</artifactId>  
 <optional>true</optional>  
</dependency>

@Data  
@Component  
@ConfigurationProperties(prefix = "person")  
public class Person {  
 private String name;  
 private Integer age;  
}

@Component

@ConfigurationProperties(prefix = "person")

这两个必须，@Data为类提供set和get方法，减少代码。

## 配置文件值注入两种方式对比

配置文件值注入有两种方式，一个是Spring Boot的@ConfigurationProperties注解，另一个是spring原先的@value注解

|  |  |  |
| --- | --- | --- |
|  | @ConfigurationProperties | @Value |
| 功能 | 批量注入配置文件中的属性 | 一个个指定 |
| 松散绑定（松散语法） | 支持 | 不支持 |
| SpEL | 不支持 | 支持 |
| JSR303数据校验 | 支持 | 不支持 |
| 复杂类型封装 | 支持 | 不支持 |

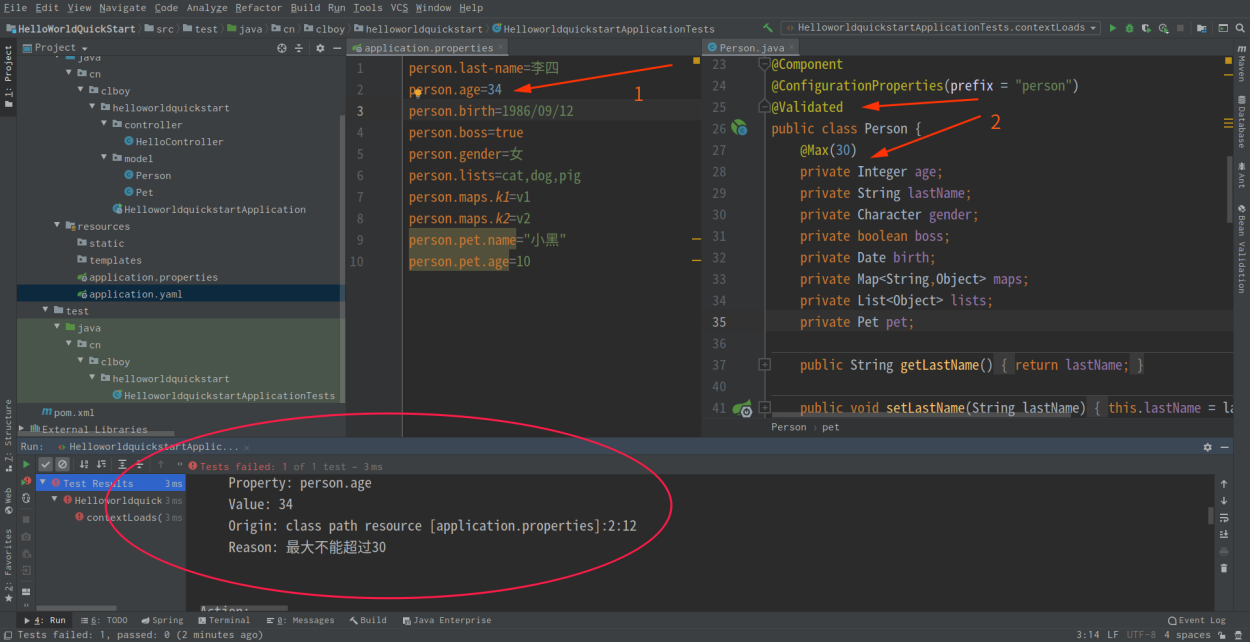
**松散绑定**：例如Person中有lastName属性，在配置文件中可以写成lastName或lastname或last-name或last\_name等等

**SpEL**：

*##　properties配置文件*persion.age=#{2019-1986+1}  
  
*# Person类  
#-使用@ConfigurationProperties注解，会抛出异常*@Component  
@ConfigurationProperties(prefix = "person")  
public class Person {  
private Integer age;  
  
  
*#--------------------使用@value注解 OK--------------------*@Component  
public class Person {  
@Value("${person.age}")  
private Integer age;

**JSR303数据校验**

@ConfigurationProperties支持校验，如果校验不通过，会抛出异常



**复杂类型封装**

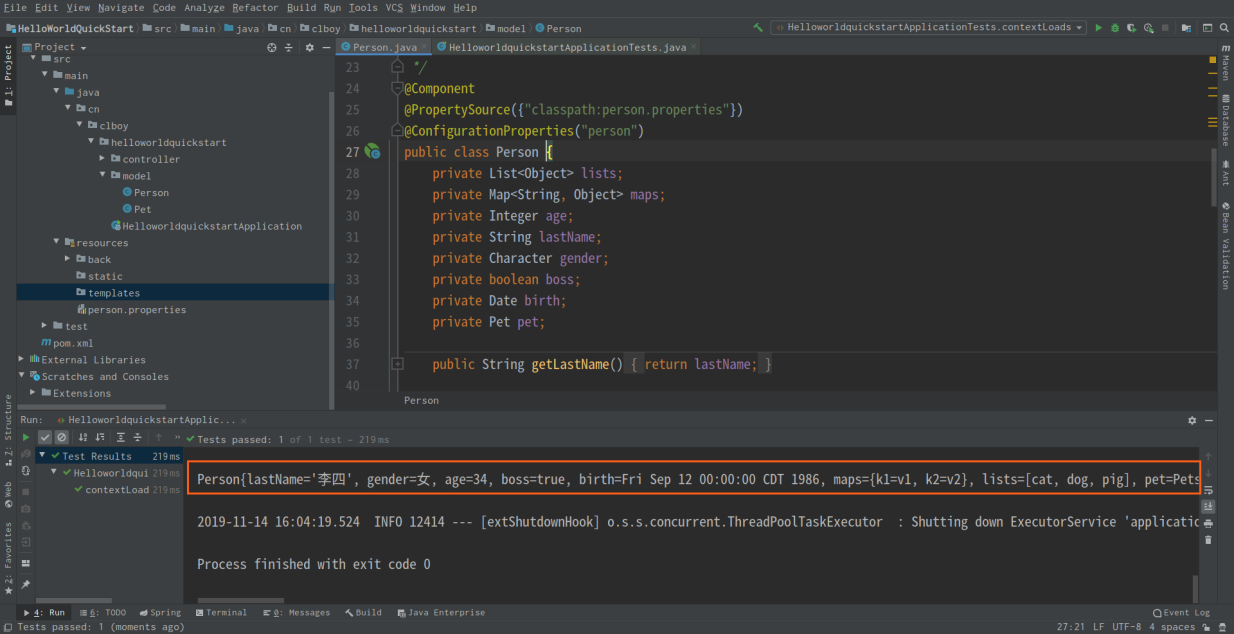
@value注解无法注入map等对象的复杂类型，但list、数组可以

## @PropertySource

作用是加载指定的配置文件，值可以是数组，也就是可以加载多个配置文件

springboot默认加载的配置文件名是application，如果配置文件名不是这个是不会被容器加载的

[使用@PropertySource({"classpath:person.properties"})指定加载person.properties配置文件](mailto:使用@PropertySource({\"classpath:person.properties\"})指定加载person.properties配置文件)



## @ImportResource

@ImportResource注解用于导入Spring的配置文件，让配置文件里面的内容生效；(就是以前写的springmvc.xml、applicationContext.xml)

这个注解是放在主入口函数的类上，而不是测试类上，不推荐使用。

## @Configuration

SpringBoot推荐给容器中添加组件的方式；推荐使用全注解的方式

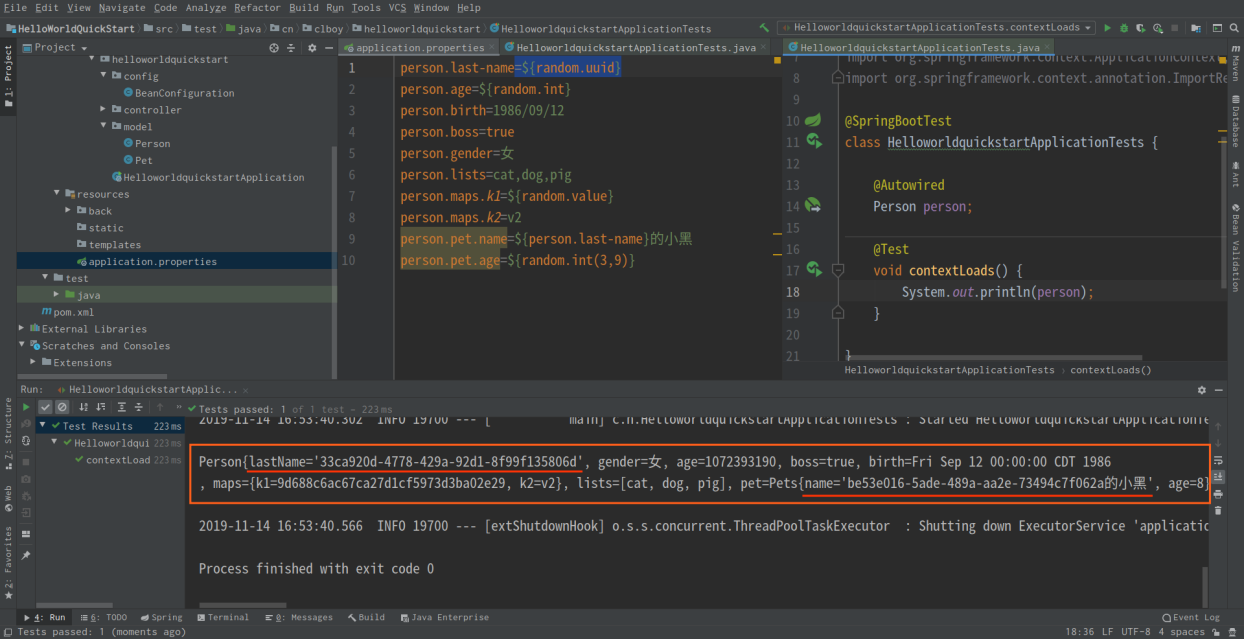
配置类@Configuration ---equals---> Spring配置文件

@Bean

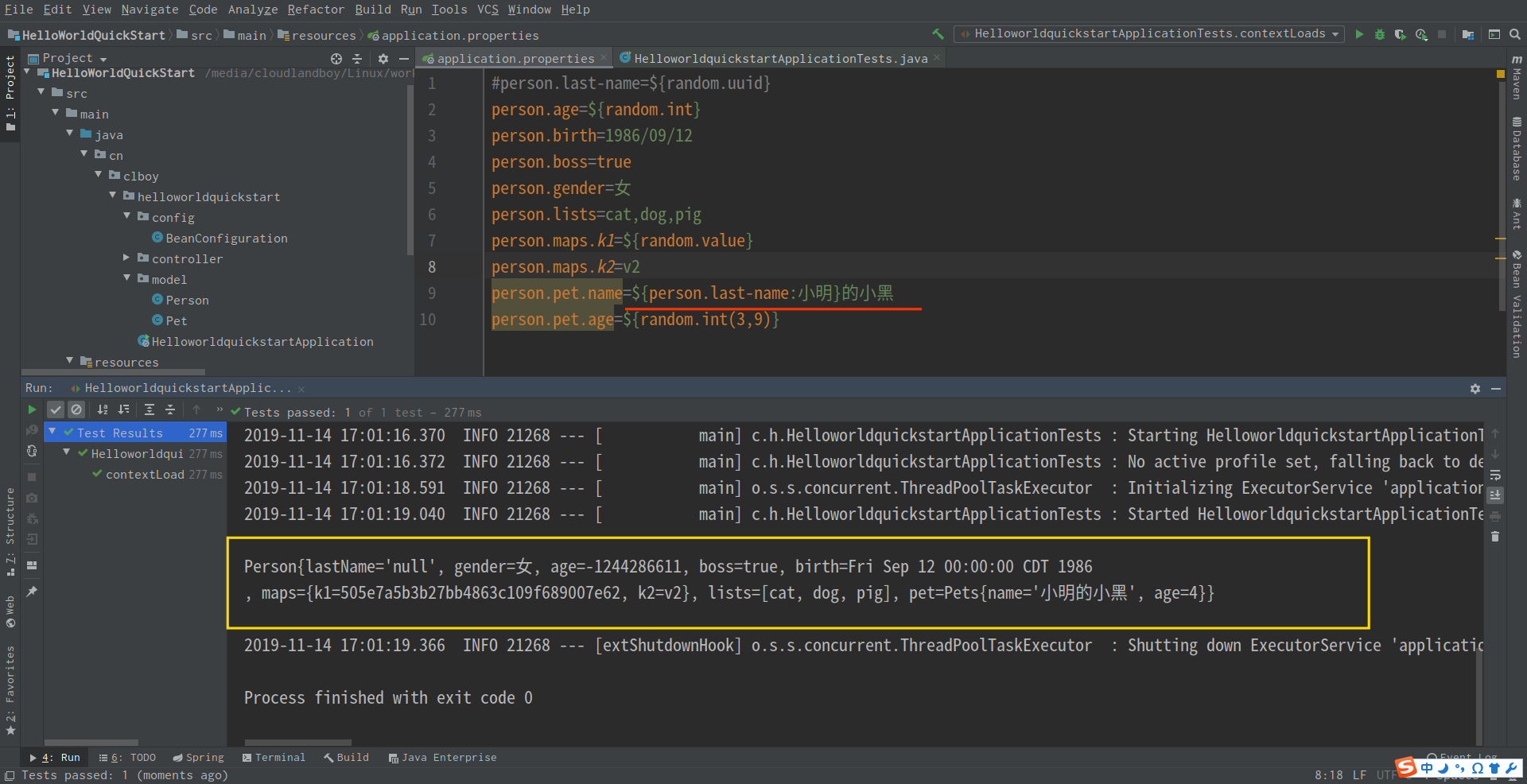
使用@Bean给容器中添加组件

@Configuration  
public class BeanConfiguration {  
  
 */\*\*  
 \*相当于在配置文件中用<bean><bean/>标签添加组件  
 \*/* @Bean  
 public Pet myPet() {  
 Pet pet = new Pet();  
 pet.setName("嘟嘟");  
 pet.setAge(3);  
 return pet;  
 }  
}

## 配置文件占位符



可以使用:指定默认值



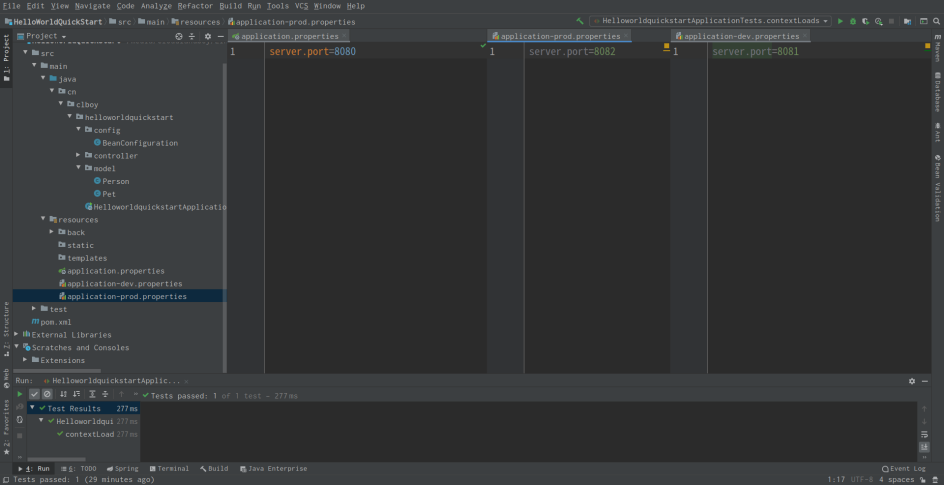
## Profile

rofile是Spring对不同环境提供不同配置功能的支持，可以通过激活、指定参数等方式快速切换环境

### 多profile文件形式

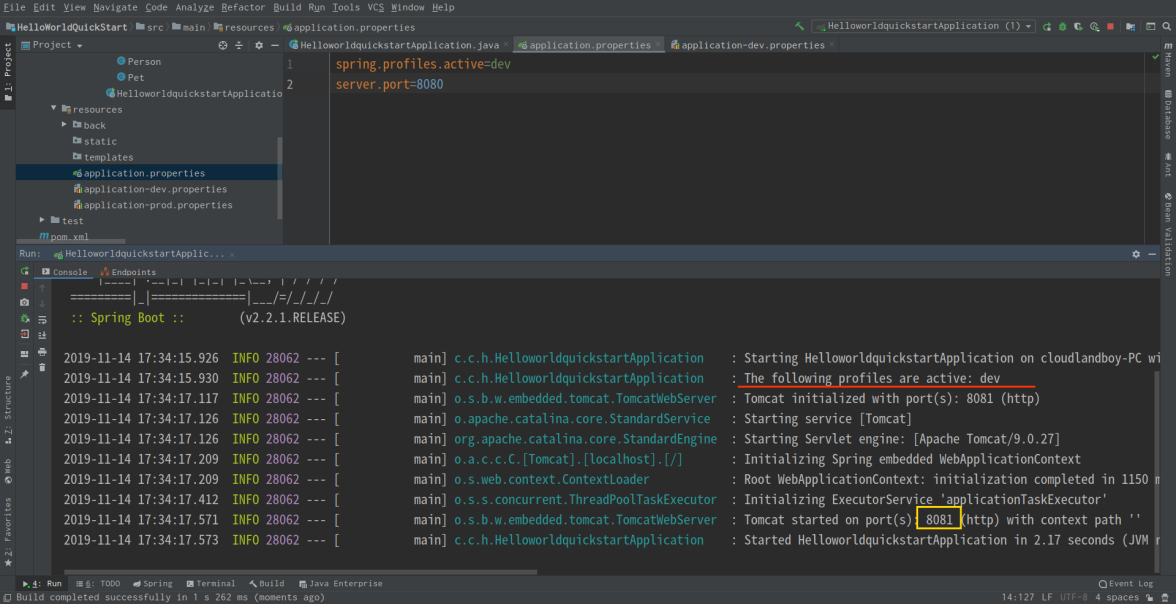
文件名格式：application-{profile}.properties/yml，例如：

* application-dev.properties
* application-prod.properties



程序启动时会默认加载application.properties，启动的端口就是8080

可以在主配置文件中指定激活哪个配置文件



### yml支持多文档块方式

每个文档块使用---分割

server:

port: 8080

spring:

profiles:

active: prod

---

server:

port: 8081

spring:

profiles: dev

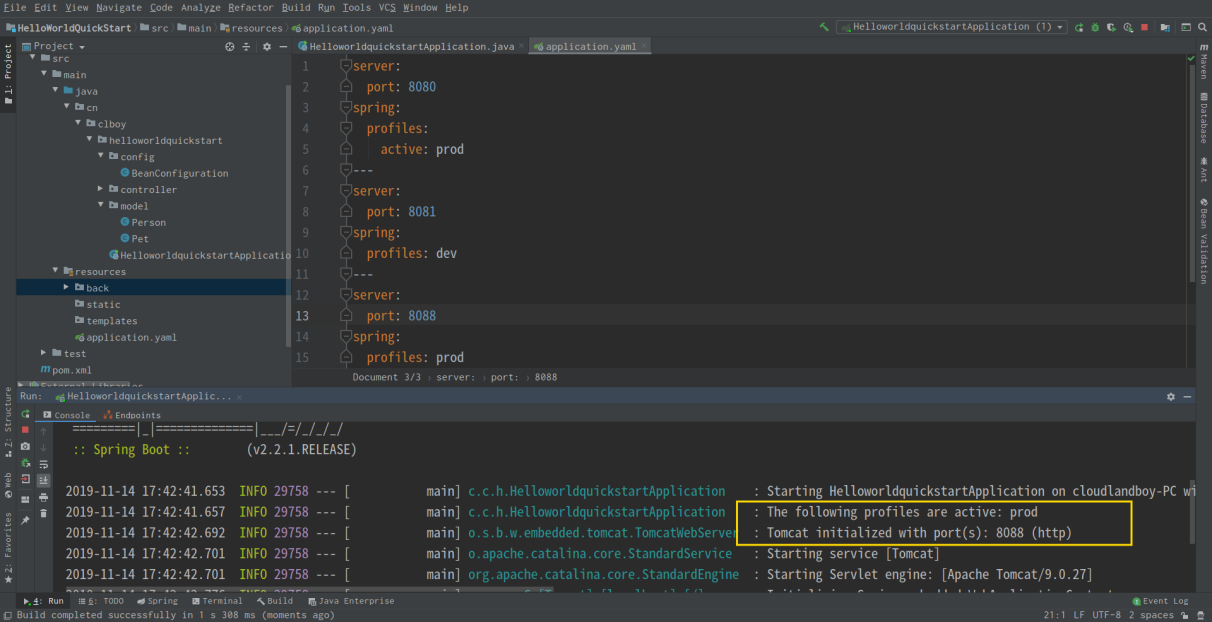
---

server:

port: 8082

spring:

profiles: prod



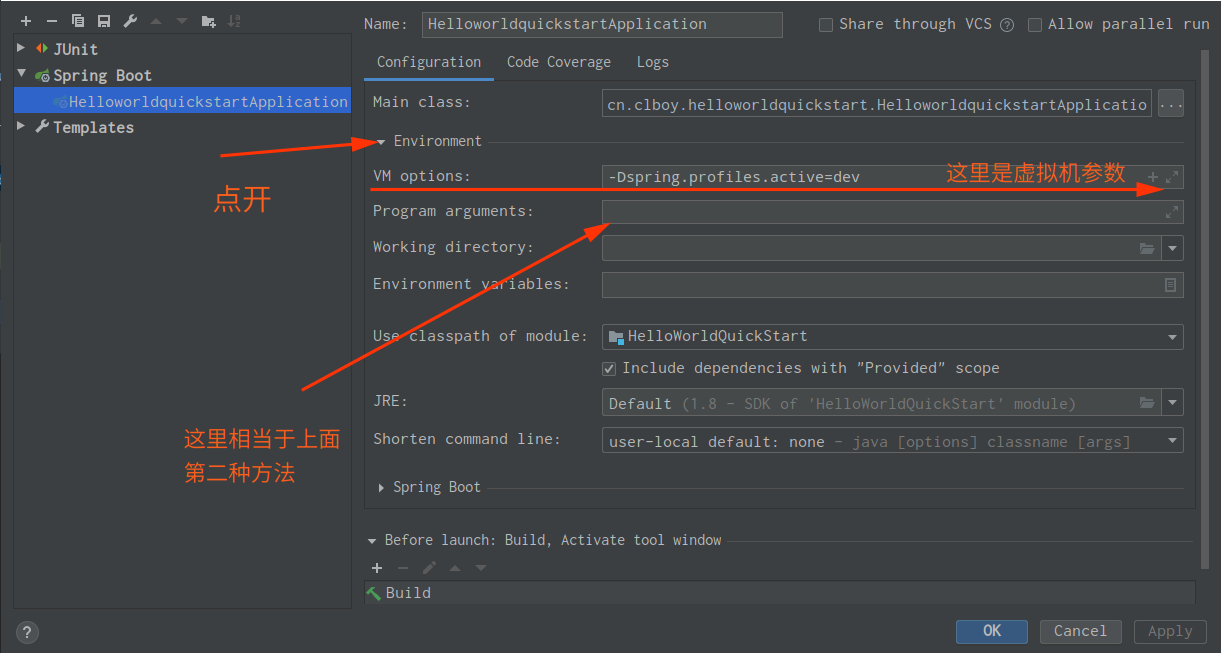
### 激活指定profile的三种方式

1.在配置文件中指定 spring.profiles.active=dev（如上）

2.项目打包后在命令行启动

java -jar xxx.jar --spring.profiles.active=dev；

1. 虚拟机参数



### 配置文件加载位置

springboot 启动会扫描以下位置的application.properties或者application.yml文件作为Spring boot的默认配置文件

file: ./config/ 4高优先级

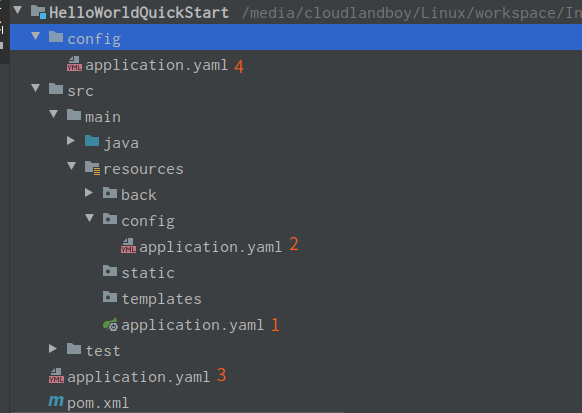
​ file: ./ 3

​ classpath: /config/ 2

​ classpath: / 1低

优先级由高到底，高优先级的配置会覆盖低优先级的配置（优先级低的先加载）；

SpringBoot会从这四个位置全部加载主配置文件；互补配置；



# 日志配置

市面上的日志框架

|  |  |
| --- | --- |
| 日志门面 （日志的抽象层） | 实现层 |
| ~~JCL（Jakarta Commons Logging~~  **SLF4j（Simple Logging Facade for Java）**  ~~jboss-loggi~~ | ~~JUL（java.util.logging）~~  ~~Log4j~~  ~~Log4j2~~  **Logback** |

左边选一个门面（抽象层）、右边来选一个实现；

SpringBoot选用 SLF4j和logback

例：SLF4j-->Logback (这两个是同一个人写的)

## SLF4j使用

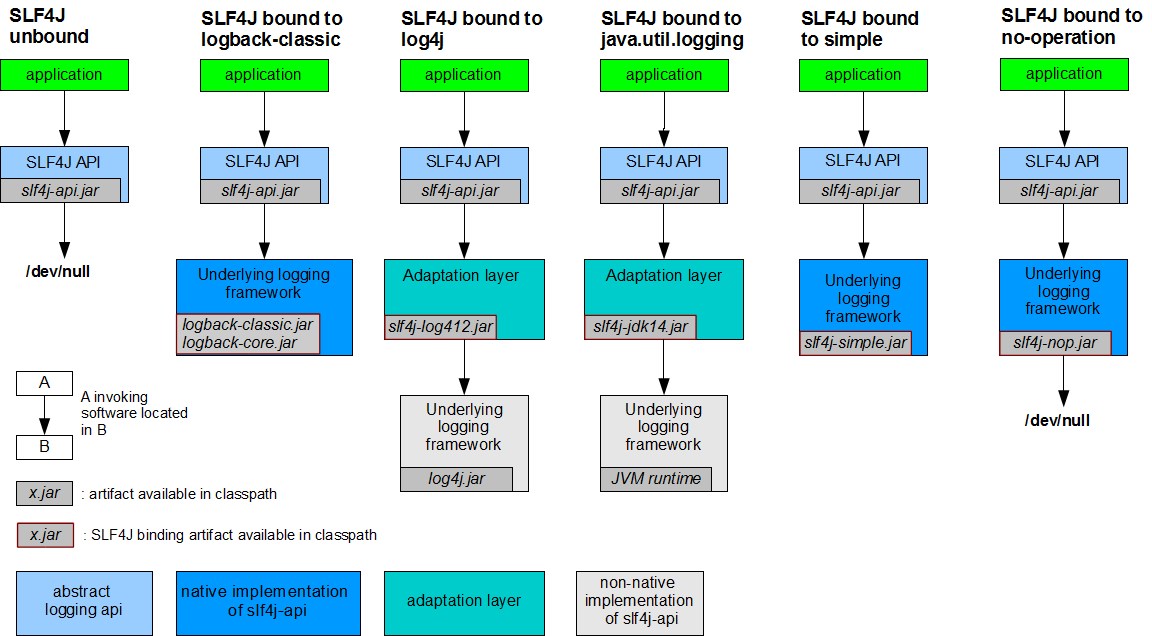
如何在系统中使用SLF4j ：https://www.slf4j.org

以后开发的时候，日志记录方法的调用，不应该来直接调用日志的实现类，而是调用日志抽象层里面的方法；

给系统里面导入slf4j的jar和 logback的实现jar

import org.slf4j.Logger;  
import org.slf4j.LoggerFactory;

public class TestController {  
 public static void main(String[] args) {  
 Logger logger = LoggerFactory.*getLogger*(TestController.class);  
 logger.info("Hello World");  
 }  
}



绿色的是中间适配器，将SLF4J API适配到想要的实现层API

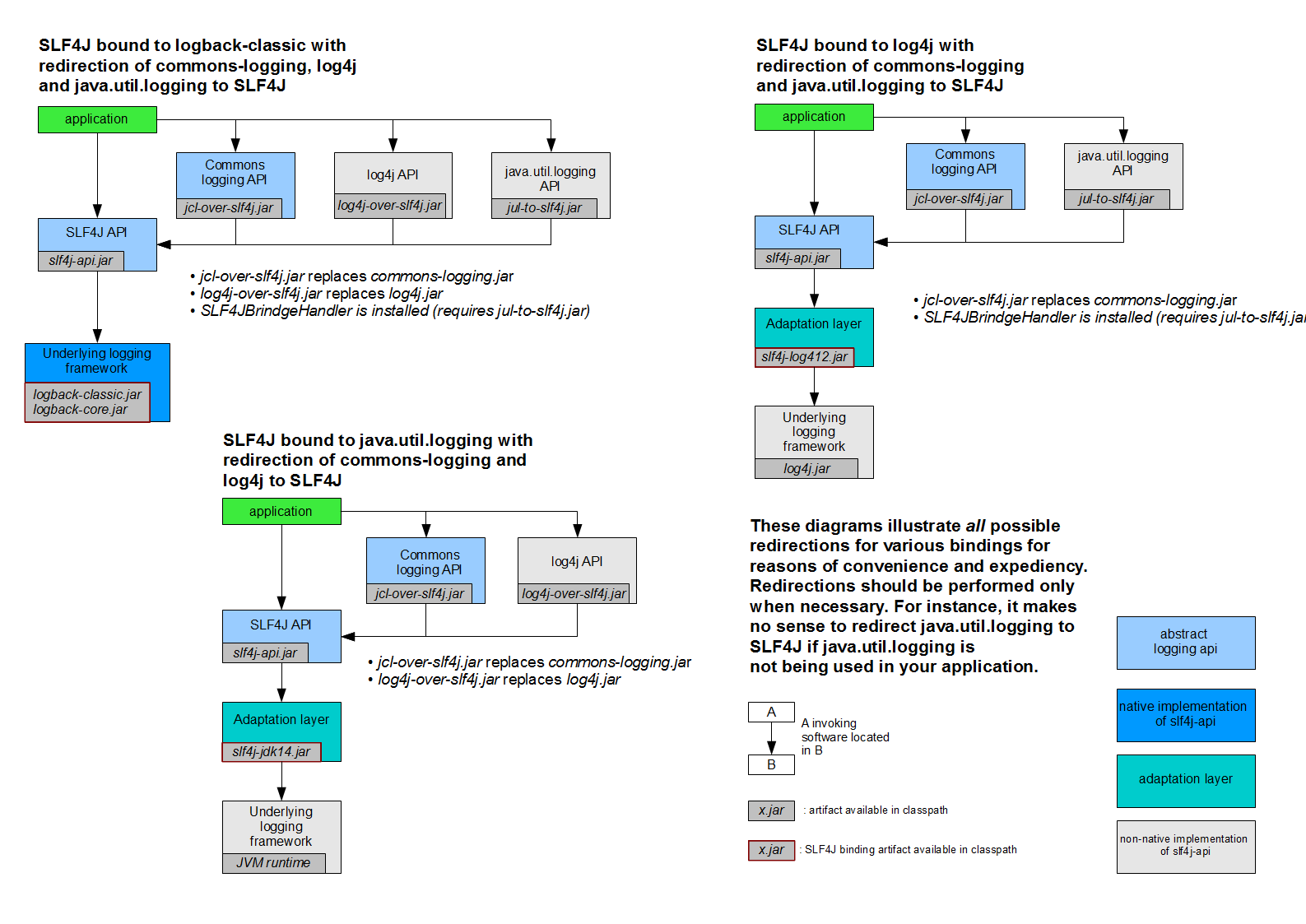
每一个日志的实现框架都有自己的配置文件。使用slf4j以后，配置文件还是做成日志实现框架自己本身的配置文件；

## 遗留问题

项目中依赖的框架可能使用不同的日志：

Spring（commons-logging）、Hibernate（jboss-logging）、MyBatis、xxxx

当项目是使用多种日志API时，可以统一适配到SLF4J，中间使用SLF4J或者第三方提供的日志适配器适配到SLF4J，SLF4J在底层用开发者想用的一个日志框架来进行日志系统的实现，从而达到了多种日志的统一实现。



## 如何让系统中所有的日志都统一到slf4j

将系统中其他日志框架先排除出去；

用中间包来替换原有的日志框架（适配器的类名和包名与替换的被日志框架一致）；

我们导入slf4j其他的实现

例如导入spring时，使用exclusions排除传递依赖

<dependency>  
 <groupId>org.springframework</groupId>  
 <artifactId>spring-core</artifactId>  
 <version>${spring.version}</version>  
 <exclusions>  
 <exclusion>  
 <artifactId>commons-logging</artifactId>  
 <groupId>commons-logging</groupId>  
 </exclusion>  
 </exclusions>  
</dependency>

## SpringBoot日志关系

新建的Spring Boot项目一般都会引用spring-boot-starter或者spring-boot-starter-web

<!--一般会引入两个中的其中一个-->

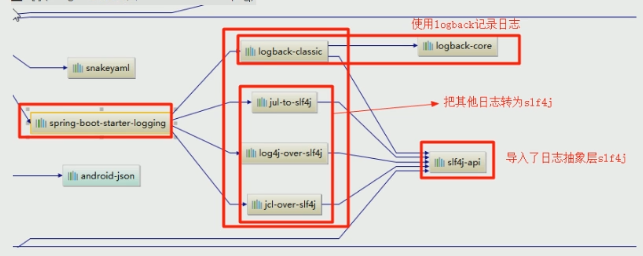
<dependency>  
 <groupId>org.springframework.boot</groupId>  
 <artifactId>spring-boot-starter-web</artifactId>  
</dependency>

<dependency>  
 <groupId>org.springframework.boot</groupId>  
 <artifactId>spring-boot-starter</artifactId>  
</dependency>

这两个起步依赖中都已经包含了对于spring-boot-starter-logging的依赖，SpringBoot使用它来做日志功能：

<dependency>  
 <groupId>org.springframework.boot</groupId>  
 <artifactId>spring-boot-starter-logging</artifactId>  
 <version>2.2.6.RELEASE</version>  
 <scope>compile</scope>  
</dependency>

底层依赖关系：



总结：

SpringBoot底层也是使用slf4j+logback的方式进行日志记录

SpringBoot也把其他的日志都替换成了slf4j；

中间替换包？

我们要引入其他框架一定要把这个框架的默认日志依赖移除掉，把所有日志都统一到slf4j。

SpringBoot能自动适配所有的日志，而且底层使用slf4j+logback的方式记录日志，引入其他框架的时候，只需要把这个框架依赖的日志框架排除掉即可；

## 日志使用

### 默认配置

SpringBoot默认帮我们配置好了日志；

//记录器  
Logger logger = LoggerFactory.*getLogger*(getClass());  
@Test  
public void contextLoads() {  
 //System.out.println();  
  
 //日志的级别；  
 //由低到高 trace<debug<info<warn<error  
 //可以调整输出的日志级别；日志就只会在这个级别以以后的高级别生效  
 logger.trace("这是trace日志...");  
 logger.debug("这是debug日志...");  
 //SpringBoot默认给我们使用的是info级别的，没有指定级别的就用SpringBoot默认规定的级别；root级别  
 logger.info("这是info日志...");  
 logger.warn("这是warn日志...");  
 logger.error("这是error日志...");  
}

日志输出格式：  
 %d表示日期时间，  
 %thread表示线程名，  
 %-5level：级别从左显示5个字符宽度  
 %logger{50} 表示logger名字最长50个字符，否则按照句点分割。  
 %msg：日志消息，  
 %n是换行符  
-->  
 %d{yyyy-MM-dd HH:mm:ss.SSS} [%thread] %-5level %logger{50} - %msg%n

SpringBoot修改日志的默认配置

# 也可以指定一个包路径 logging.level.com.xxx=error  
logging.level.root=error  
  
  
#logging.path=  
# 不指定路径在当前项目下生成springboot.log日志  
# 可以指定完整的路径；  
#logging.file=G:/springboot.log  
  
# 在当前磁盘的根路径下创建spring文件夹和里面的log文件夹；使用 spring.log 作为默认文件  
logging.path=/spring/log  
  
# 在控制台输出的日志的格式  
logging.pattern.console=%d{yyyy-MM-dd} [%thread] %-5level %logger{50} - %msg%n  
# 指定文件中日志输出的格式  
logging.pattern.file=%d{yyyy-MM-dd} === [%thread] === %-5level === %logger{50} ==== %msg%n

|  |  |  |  |
| --- | --- | --- | --- |
| **logging.file** | **logging.path** | **Example** | **Description** |
| (none) | (none) |  | 只在控制台输出 |
| 指定文件名 | (none) | my.log | 输出日志到my.log文件 |
| (none) | 指定目录 | /var/log | 输出到指定目录的 spring.log 文件中 |

### 指定配置

给类路径(resources)下放上每个日志框架自己的配置文件即可；SpringBoot就不使用他默认配置的了

|  |  |
| --- | --- |
| **Logging System** | **Customization** |
| Logback | logback-spring.xml, logback-spring.groovy, logback.xml or logback.groovy |
| Log4j2 | log4j2-spring.xml or log4j2.xml |
| JDK (Java Util Logging) | logging.properties |

logback.xml：直接就被日志框架识别了；

logback-spring.xml：日志框架就不直接加载日志的配置项，由SpringBoot解析日志配置，可以使用SpringBoot的高级Profile功能

<appender name="stdout" class="ch.qos.logback.core.ConsoleAppender">  
<!--  
日志输出格式：  
 %d表示日期时间，  
 %thread表示线程名，  
 %-5level：级别从左显示5个字符宽度  
 %logger{50} 表示logger名字最长50个字符，否则按照句点分割。  
 %msg：日志消息，  
 %n是换行符  
-->  
<layout class="ch.qos.logback.classic.PatternLayout">  
 <springProfile name="dev">  
 <pattern>%d{yyyy-MM-dd HH:mm:ss.SSS} ----> [%thread] ---> %-5level %logger{50} - %msg%n</pattern>  
 </springProfile>  
 <springProfile name="!dev">  
 <pattern>%d{yyyy-MM-dd HH:mm:ss.SSS} ==== [%thread] ==== %-5level %logger{50} - %msg%n</pattern>  
 </springProfile>  
</layout>  
</appender>

## 切换日志框架

可以按照slf4j的日志适配图，进行相关的切换；

slf4j+log4j的方式；

<dependency>  
 <groupId>org.springframework.boot</groupId>  
 <artifactId>spring-boot-starter-web</artifactId>  
 <exclusions>  
 <exclusion>  
 <artifactId>logback-classic</artifactId>  
 <groupId>ch.qos.logback</groupId>  
 </exclusion>  
 <exclusion>  
 <artifactId>log4j-over-slf4j</artifactId>  
 <groupId>org.slf4j</groupId>  
 </exclusion>  
 </exclusions>  
</dependency>  
  
<dependency>  
 <groupId>org.slf4j</groupId>  
 <artifactId>slf4j-log4j12</artifactId>  
</dependency>

切换为log4j2

<dependency>  
 <groupId>org.springframework.boot</groupId>  
 <artifactId>spring-boot-starter-web</artifactId>  
 <exclusions>  
 <exclusion>  
 <artifactId>spring-boot-starter-logging</artifactId>  
 <groupId>org.springframework.boot</groupId>  
 </exclusion>  
 </exclusions>  
</dependency>  
  
<dependency>  
 <groupId>org.springframework.boot</groupId>  
 <artifactId>spring-boot-starter-log4j2</artifactId>  
</dependency>

# Web开发

1、创建SpringBoot应用，选中我们需要的模块

2、SpringBoot已经默认将这些场景配置好了，只需要在配置文件中指定少量配置就可以运行起来

1. 自己编写业务代码

## SpringBoot对静态资源的映射规则

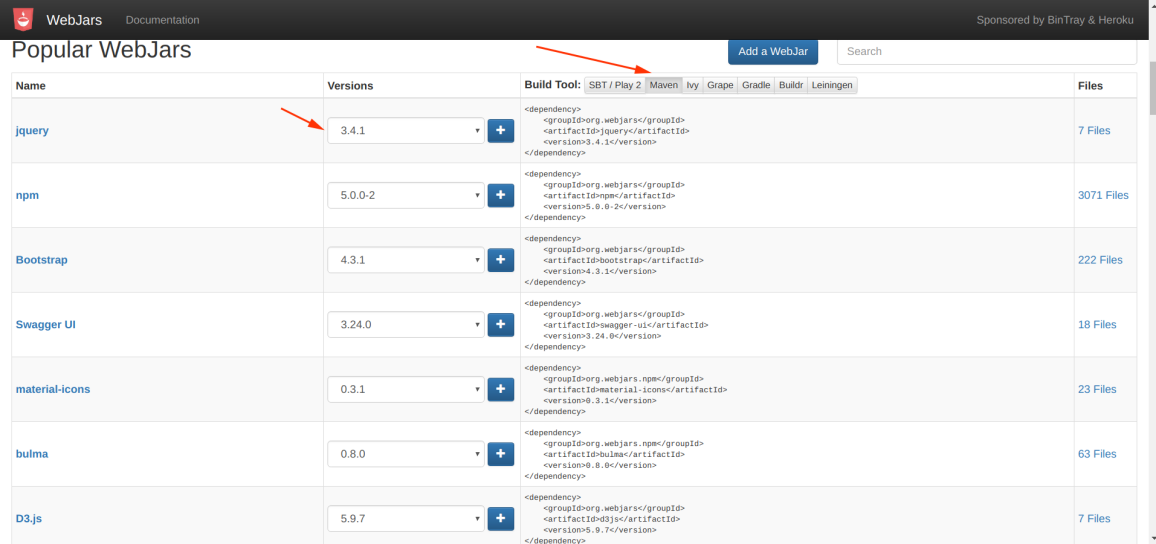
WebMvcAutoConfiguration类的addResourceHandlers方法：（添加资源映射）

public void addResourceHandlers(ResourceHandlerRegistry registry) {  
 if (!this.resourceProperties.isAddMappings()) {  
 logger.debug("Default resource handling disabled");  
 } else {  
 Duration cachePeriod = this.resourceProperties.getCache().getPeriod();  
 CacheControl cacheControl = this.resourceProperties.getCache().getCachecontrol().toHttpCacheControl();  
 if (!registry.hasMappingForPattern("/webjars/\*\*")) {  
 this.customizeResourceHandlerRegistration(registry.addResourceHandler(new String[]{"/webjars/\*\*"}).addResourceLocations(new String[]{"classpath:/META-INF/resources/webjars/"}).setCachePeriod(this.getSeconds(cachePeriod)).setCacheControl(cacheControl));  
 }  
  
 String staticPathPattern = this.mvcProperties.getStaticPathPattern();  
 if (!registry.hasMappingForPattern(staticPathPattern)) {  
 this.customizeResourceHandlerRegistration(registry.addResourceHandler(new String[]{staticPathPattern}).addResourceLocations(WebMvcAutoConfiguration.getResourceLocations(this.resourceProperties.getStaticLocations())).setCachePeriod(this.getSeconds(cachePeriod)).setCacheControl(cacheControl));  
 }  
  
 }  
}

所有 /webjars/\*\* ，都去 classpath:/META-INF/resources/webjars/ 找资源

webjars：以jar包的方式引入静态资源；

[webjars官网](https://www.webjars.org/" \t "https://cloudlandboy.github.io/myNote/" \l "/backend/springboot/_blank)



例如：添加jquery的webjars

<dependency>  
 <groupId>org.webjars</groupId>  
 <artifactId>jquery</artifactId>  
 <version>3.4.1</version>  
</dependency>

访问地址对应就是：<http://localhost:8080/webjars/jquery/3.4.1/jquery.js>

## 非webjars，自己的静态资源怎么访问

|  |  |
| --- | --- |
| **数组中的值** | **在项目中的位置** |
| classpath:/META-INF/resources/ | src/main/resources/META-INF/resources/ |
| classpath:/resources/ | src/main/resources/resources/ |
| classpath:/static/ | src/main/resources/static/ |
| classpath:/public/ | src/main/resources/public/ |

localhost:8080/abc ---> 去静态资源文件夹里面找abc

# Thymeleaf模板引擎

常见的模板引擎有JSP、Velocity、Freemarker、Thymeleaf

SpringBoot推荐使用Thymeleaf；

## 引入thymeleaf

<dependency>  
 <groupId>org.springframework.boot</groupId>  
 <artifactId>spring-boot-starter-thymeleaf</artifactId>  
</dependency>

如需切换thymeleaf版本：

<properties>  
  
 <thymeleaf.version>X.X.X.RELEASE</thymeleaf.version>  
  
 <!-- 布局功能的支持程序 thymeleaf3主程序 layout2以上版本 -->  
 <!-- thymeleaf2 layout1-->  
 <thymeleaf-layout-dialect.version>2.2.2</thymeleaf-layout-dialect.version>  
  
</properties>

## Thymeleaf使用

默认只要我们把HTML页面放在classpath:/templates/，thymeleaf就能自动渲染；

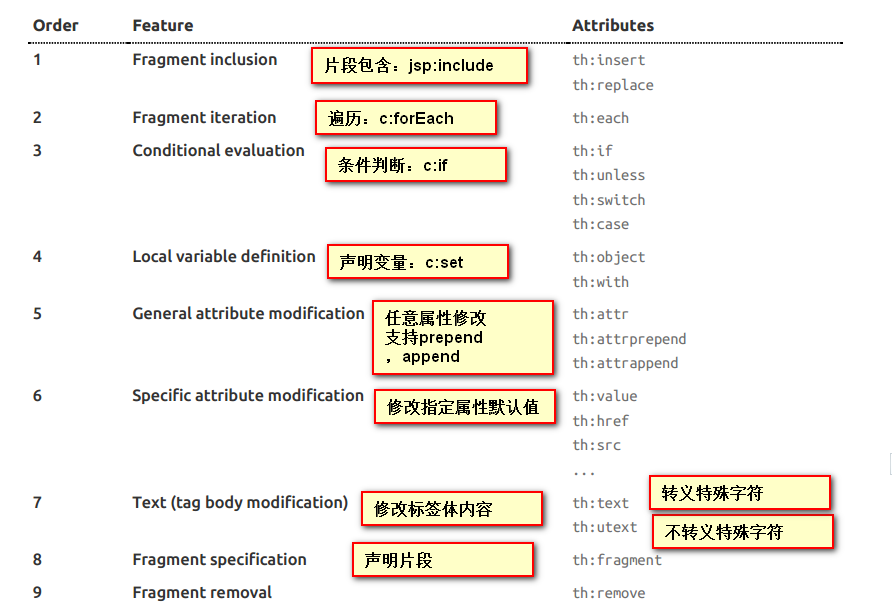
1、在controller中准备数据

@Controller  
public class HelloT {  
  
 @RequestMapping("/ht")  
 public String ht(Model model) {  
 model.addAttribute("title","hello Thymeleaf")  
 .addAttribute("info","this is first thymeleaf test");  
 return "t1";  
 }  
}

2、创建模板文件t1.html，并导入thymeleaf的名称空间

<!DOCTYPE html>  
<html lang="en" xmlns:th="http://www.thymeleaf.org">  
<head>  
 <meta charset="UTF-8">  
 <title>[[${title}]]</title>  
</head>  
<body>  
<h1 th:text="${title}"></h1>  
<div th:text="${info}">这里的文本之后将会被覆盖</div>  
</body>  
</html>

## 语法规则



更多配置参考官方文档：https://www.thymeleaf.org/documentation.html

中文参考书册：<https://www.lanzous.com/i7dzr2j>

# SpringMVC自动配置

Spring Boot为Spring MVC提供了自动配置，可与大多数应用程序完美配合。

以下是SpringBoot对SpringMVC的默认配置

org.springframework.boot.autoconfigure.web.servlet.WebMvcAutoConfiguration

自动配置在Spring的默认值之上添加了以下功能：

* 包含ContentNegotiatingViewResolver和BeanNameViewResolver。--> 视图解析器
* 支持服务静态资源，包括对WebJars的支持（[官方文档中有介绍](https://docs.spring.io/spring-boot/docs/2.2.1.RELEASE/reference/html/spring-boot-features.html" \l "boot-features-spring-mvc-static-content" \t "https://cloudlandboy.github.io/myNote/" \l "/backend/springboot/_blank)）。--> 静态资源文件夹路径
* 自动注册Converter，GenericConverter和Formatterbeans。--> 转换器，格式化器
* 支持HttpMessageConverters（[官方文档中有介绍](https://docs.spring.io/spring-boot/docs/2.2.1.RELEASE/reference/html/spring-boot-features.html" \l "boot-features-spring-mvc-message-converters" \t "https://cloudlandboy.github.io/myNote/" \l "/backend/springboot/_blank)）。--> SpringMVC用来转换Http请求和响应的；User---Json；
* 自动注册MessageCodesResolver（[官方文档中有介绍](https://docs.spring.io/spring-boot/docs/2.2.1.RELEASE/reference/html/spring-boot-features.html" \l "boot-features-spring-message-codes" \t "https://cloudlandboy.github.io/myNote/" \l "/backend/springboot/_blank)）。--> 定义错误代码生成规则
* 静态index.html支持。--> 静态首页访问
* 定制Favicon支持（[官方文档中有介绍](https://docs.spring.io/spring-boot/docs/2.2.1.RELEASE/reference/html/spring-boot-features.html" \l "boot-features-spring-mvc-favicon" \t "https://cloudlandboy.github.io/myNote/" \l "/backend/springboot/_blank)）。--> 网站图标
* 自动使用ConfigurableWebBindingInitializerbean（[官方文档中有介绍](https://docs.spring.io/spring-boot/docs/2.2.1.RELEASE/reference/html/spring-boot-features.html" \l "boot-features-spring-mvc-web-binding-initializer" \t "https://cloudlandboy.github.io/myNote/" \l "/backend/springboot/_blank)）。

如果您想保留 Spring Boot MVC 的功能，并且需要添加其他 [MVC 配置](https://docs.spring.io/spring/docs/5.1.3.RELEASE/spring-framework-reference/web.html" \l "mvc" \t "https://cloudlandboy.github.io/myNote/" \l "/backend/springboot/_blank)（拦截器，格式化程序和视图控制器等），可以添加自己的 WebMvcConfigurer 类型的 @Configuration 类，但**不能**带 @EnableWebMvc 注解。如果您想自定义 RequestMappingHandlerMapping、RequestMappingHandlerAdapter 或者 ExceptionHandlerExceptionResolver 实例，可以声明一个 WebMvcRegistrationsAdapter 实例来提供这些组件。

如果您想完全掌控 Spring MVC，可以添加自定义注解了 @EnableWebMvc 的 @Configuration 配置类。

## 扩展SpringMVC

以前的配置文件中的配置

<mvc:view-controller path="/hello" view-name="success"/>

现在，编写一个配置类（@Configuration），是WebMvcConfigurer类型；不能标注@EnableWebMvc

@Configuration  
public class MyMvcConfig implements WebMvcConfigurer {  
  
 @Override  
 public void addViewControllers(ViewControllerRegistry registry) {  
 registry.addViewController("/hi").setViewName("success");  
 }  
}

访问：<http://localhost:8080/hi>

## 全面接管SpringMVC

SpringBoot对SpringMVC的自动配置不需要了，所有都是由我们自己来配置；所有的SpringMVC的自动配置都失效了。

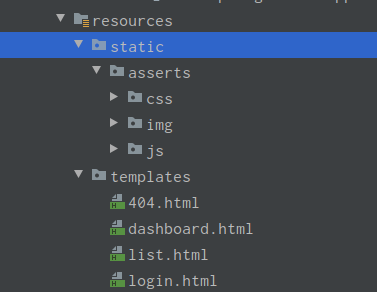
我们只需要在配置类中添加@EnableWebMvc即可；

@Configuration  
@EnableWebMvc  
public class MyMvcConfig implements WebMvcConfigurer

# restful风格的增删改查

将静态资源(css,img,js)添加到项目中，放到springboot默认的静态资源文件夹下

将模板文件(html)放到template文件夹下



默认访问首页,要添加视图映射

## i18n国际化

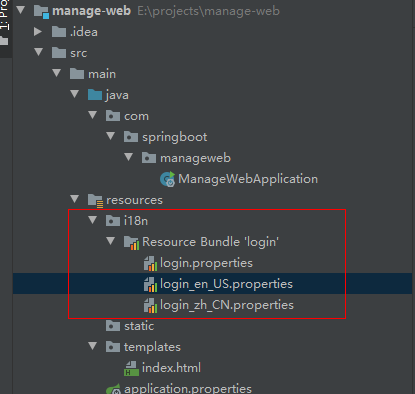
1.编写国际化配置文件，抽取页面需要显示的国际化消息

SpringBoot自动配置好了管理国际化资源文件的组件

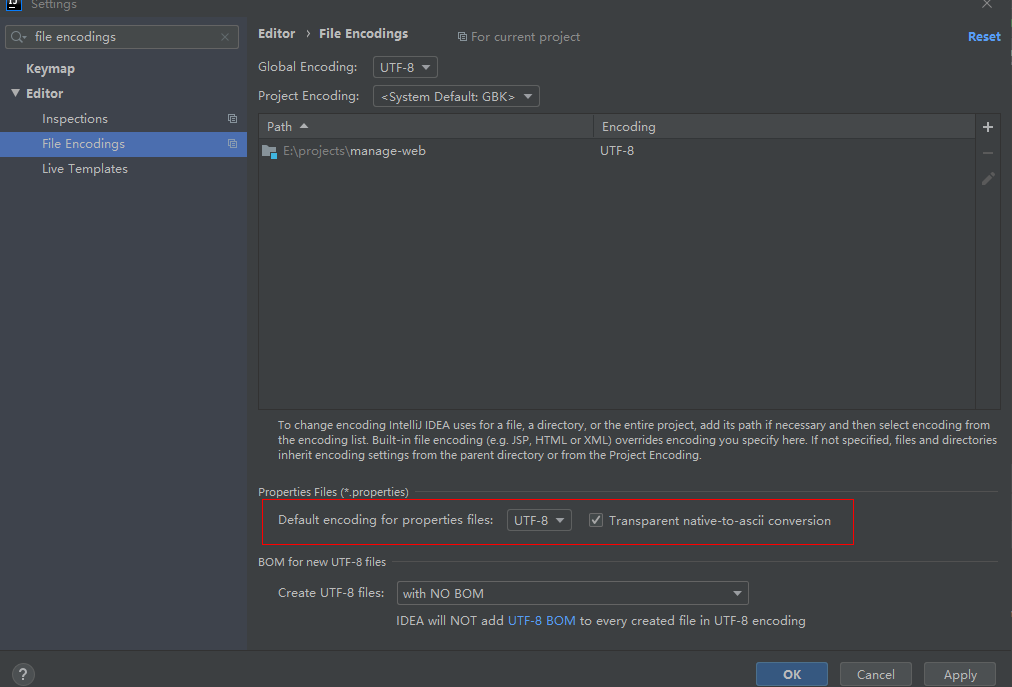
org.springframework.boot.autoconfigure.context.MessageSourceAutoConfiguration

@Configuration(  
 proxyBeanMethods = false  
)  
@ConditionalOnMissingBean(  
 name = {"messageSource"},  
 search = SearchStrategy.CURRENT  
)  
@AutoConfigureOrder(-2147483648)  
@Conditional({MessageSourceAutoConfiguration.ResourceBundleCondition.class})  
@EnableConfigurationProperties  
public class MessageSourceAutoConfiguration {  
 private static final Resource[] *NO\_RESOURCES* = new Resource[0];  
  
 public MessageSourceAutoConfiguration() {  
 }  
  
 @Bean  
 @ConfigurationProperties(  
 prefix = "spring.messages"  
 )  
 public MessageSourceProperties messageSourceProperties() {  
 return new MessageSourceProperties();  
 }  
  
 @Bean  
 public MessageSource messageSource(MessageSourceProperties properties) {  
 ResourceBundleMessageSource messageSource = new ResourceBundleMessageSource();  
 if (StringUtils.hasText(properties.getBasename())) {  
 messageSource.setBasenames(StringUtils.commaDelimitedListToStringArray(StringUtils.trimAllWhitespace(properties.getBasename())));  
 }  
  
 if (properties.getEncoding() != null) {  
 messageSource.setDefaultEncoding(properties.getEncoding().name());  
 }  
  
 messageSource.setFallbackToSystemLocale(properties.isFallbackToSystemLocale());  
 Duration cacheDuration = properties.getCacheDuration();  
 if (cacheDuration != null) {  
 messageSource.setCacheMillis(cacheDuration.toMillis());  
 }  
  
 messageSource.setAlwaysUseMessageFormat(properties.isAlwaysUseMessageFormat());  
 messageSource.setUseCodeAsDefaultMessage(properties.isUseCodeAsDefaultMessage());  
 return messageSource;  
 }

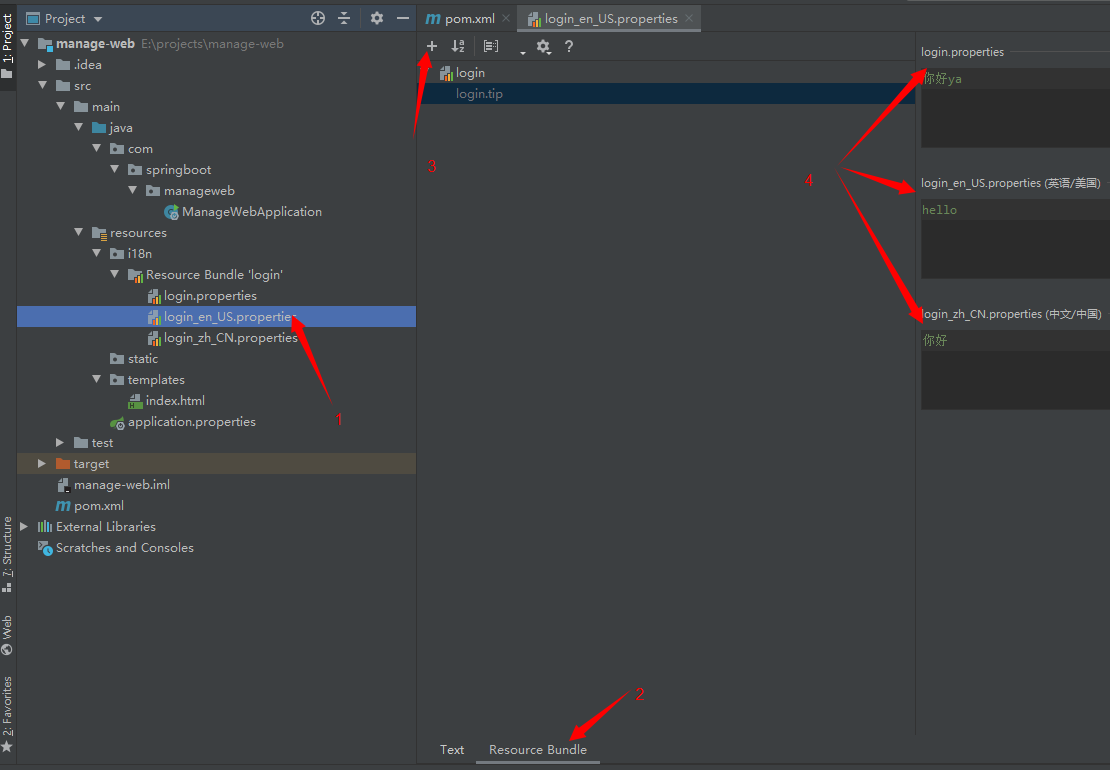
创建i18n文件夹存放配置文件，文件名格式为基础名(login)+语言代码(zh)+国家代码(CN)



### 设置idea字符编码



编写：



1. 在配置文件中添加国际化文件的位置和基础名

spring.messages.basename=i18n.login

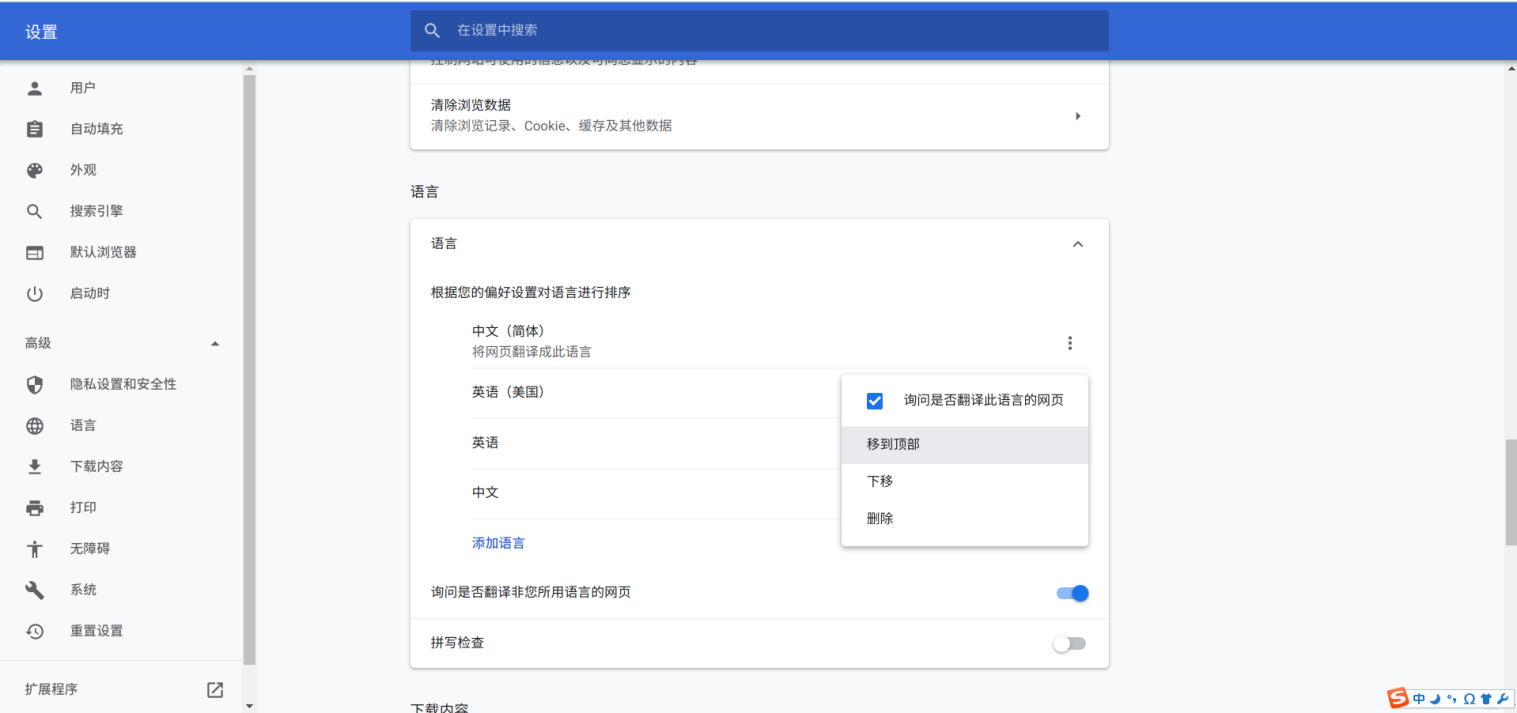
如果配置文件中没有配置基础名，就在类路径下找基础名为message的配置文件

1. 将页面文字改为获取国际化配置，格式#{key}

package com.springboot.manageweb.controller;  
  
import org.springframework.stereotype.Controller;  
import org.springframework.ui.Model;  
import org.springframework.web.bind.annotation.RequestMapping;  
  
@Controller  
public class IndexController {  
  
 @RequestMapping("/index")  
 public String Index(Model model){  
 model.addAttribute("title","dddd");  
 return "index";  
 }  
}

<!DOCTYPE html>  
<html lang="en" xmlns:th="http://www.thymeleaf.org">  
<head>  
 <meta charset="UTF-8">  
 <title>[[${title}]]</title>  
</head>  
<body>  
<h1 th:text="#{login.tip}"></h1>  
</body>  
</html>

1. 然后就可以更改浏览器语言，页面就会使用对应的国际化配置文件



### 点击连接切换语言

实现点击连接切换语言，而不是更改浏览器

修改页面，点击连接携带语言参数

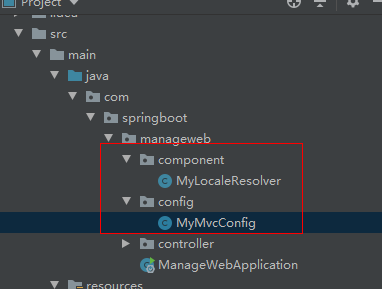
<a class="btn btn-sm" href="?l=zh\_CN">中文</a>  
<a class="btn btn-sm" href="?l=en\_US">English</a>

自己实现区域信息解析器

package com.springboot.manageweb.component;  
  
import org.springframework.util.StringUtils;  
import org.springframework.web.servlet.LocaleResolver;  
  
import javax.servlet.http.HttpServletRequest;  
import javax.servlet.http.HttpServletResponse;  
import java.util.Locale;  
  
public class MyLocaleResolver implements LocaleResolver {  
  
 @Override  
 public Locale resolveLocale(HttpServletRequest httpServletRequest) {  
 //获取请求参数中的语言  
 String language = httpServletRequest.getParameter("l");  
 //没带区域信息参数就用系统默认的  
 Locale locale = Locale.*getDefault*();  
 if (!StringUtils.*isEmpty*(language)) {  
 //提交的参数是zh\_CN （语言代码\_国家代码）  
 String[] s = language.split("\_");  
  
 locale = new Locale(s[0], s[1]);  
  
 }  
  
 return locale;  
 }  
  
 @Override  
 public void setLocale(HttpServletRequest httpServletRequest, HttpServletResponse httpServletResponse, Locale locale) {  
  
 }  
}

在配置类中将其注册到容器中

package com.springboot.manageweb.config;  
  
import com.springboot.manageweb.component.MyLocaleResolver;  
import org.springframework.context.annotation.Bean;  
import org.springframework.context.annotation.Configuration;  
import org.springframework.web.servlet.LocaleResolver;  
import org.springframework.web.servlet.config.annotation.WebMvcConfigurer;  
  
@Configuration  
public class MyMvcConfig implements WebMvcConfigurer {  
  
 @Bean  
 public LocaleResolver localeResolver() {  
 return new MyLocaleResolver();  
 }  
  
}



注：不是一定要这样的目录

## 修改页面立即生效

# 禁用缓存  
spring.thymeleaf.cache=false

在页面修改完成以后按快捷键ctrl+f9，重新编译；

## 拦截器进行登陆检查

### 实现拦截器

登录成功，把用户信息保存session中

package com.springboot.manageweb.interceptor;  
  
import org.springframework.web.servlet.HandlerInterceptor;  
import org.springframework.web.servlet.ModelAndView;  
  
import javax.servlet.http.HttpServletRequest;  
import javax.servlet.http.HttpServletResponse;  
  
public class LoginHandlerInterceptor implements HandlerInterceptor {  
 @Override  
 public boolean preHandle(HttpServletRequest request, HttpServletResponse response, Object handler) throws Exception {  
  
 Object loginUser = request.getSession().getAttribute("loginUser");  
 if (loginUser == null) {  
 //未登录，拦截，并转发到登录页面  
 request.setAttribute("msg", "您还没有登录，请先登录！");  
 request.getRequestDispatcher("/index").forward(request, response);  
 return false;  
 }  
 return true;  
 }  
  
 @Override  
 public void postHandle(HttpServletRequest request, HttpServletResponse response, Object handler, ModelAndView modelAndView) throws Exception {  
  
 }  
  
 @Override  
 public void afterCompletion(HttpServletRequest request, HttpServletResponse response, Object handler, Exception ex) throws Exception {  
  
 }  
}

在spring2.0+的版本中，只要用户自定义了拦截器，则静态资源会被拦截。但是在spring1.0+的版本中，是不会拦截静态资源的。

因此，在使用spring2.0+时，配置拦截器之后，我们要把静态资源的路径加入到不拦截的路径之中。

### 注册拦截器

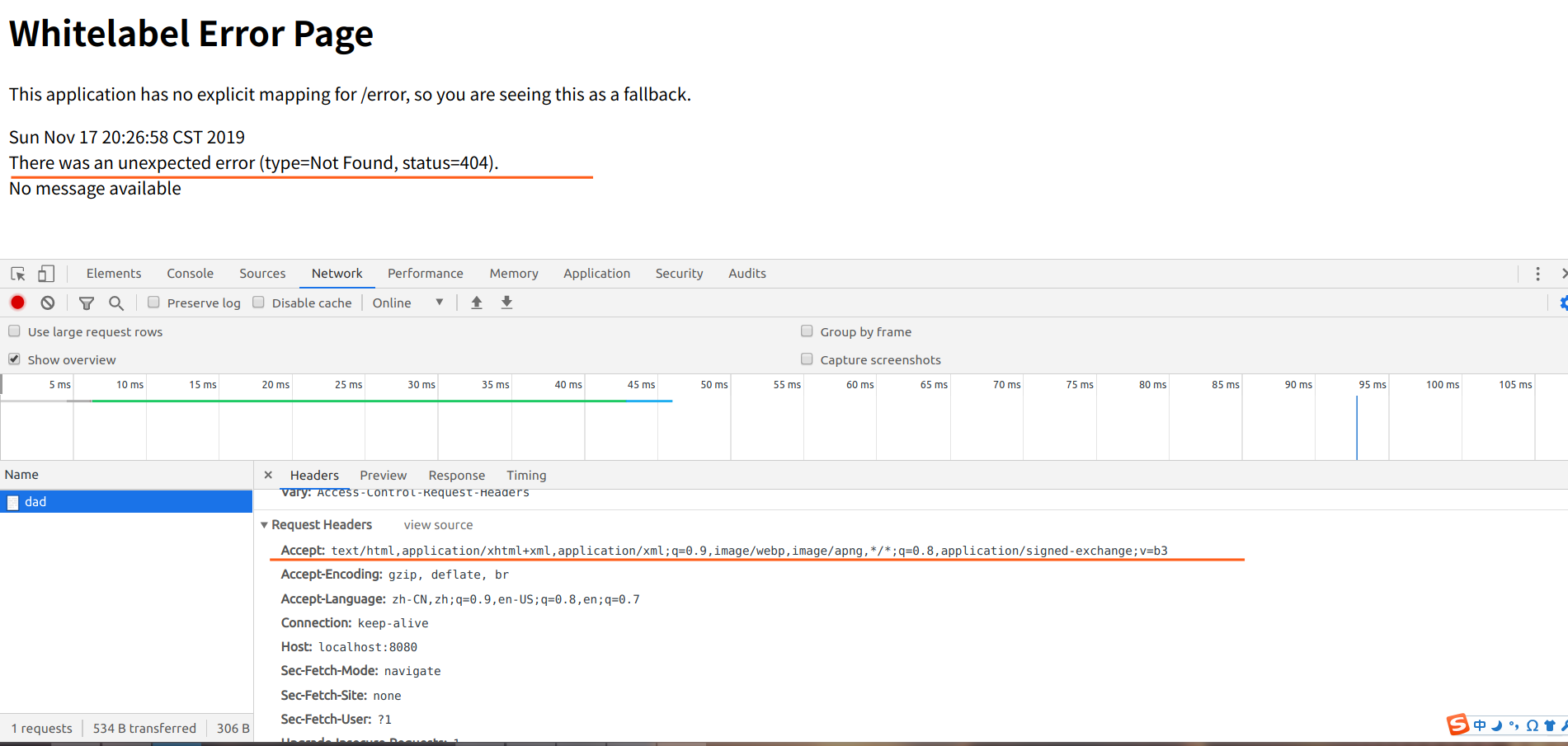
package com.springboot.manageweb.config;  
  
import com.springboot.manageweb.component.MyLocaleResolver;  
import com.springboot.manageweb.interceptor.LoginHandlerInterceptor;  
import org.springframework.context.annotation.Bean;  
import org.springframework.context.annotation.Configuration;  
import org.springframework.web.servlet.LocaleResolver;  
import org.springframework.web.servlet.config.annotation.InterceptorRegistry;  
import org.springframework.web.servlet.config.annotation.WebMvcConfigurer;  
  
@Configuration  
public class MyMvcConfig implements WebMvcConfigurer {  
 //定义不拦截路径  
 private static final String[] *excludePaths* = {"/", "/index", "/index.html", "/asserts/\*\*"};  
  
 @Bean  
 public LocaleResolver localeResolver() {  
 return new MyLocaleResolver();  
 }  
  
 @Override  
 public void addInterceptors(InterceptorRegistry registry) {  
 //添加不拦截的路径，SpringBoot已经做好了静态资源映射，所以我们不用管  
 registry.addInterceptor(new LoginHandlerInterceptor())  
 .excludePathPatterns(*excludePaths*);  
 }  
  
}

# 错误处理机制

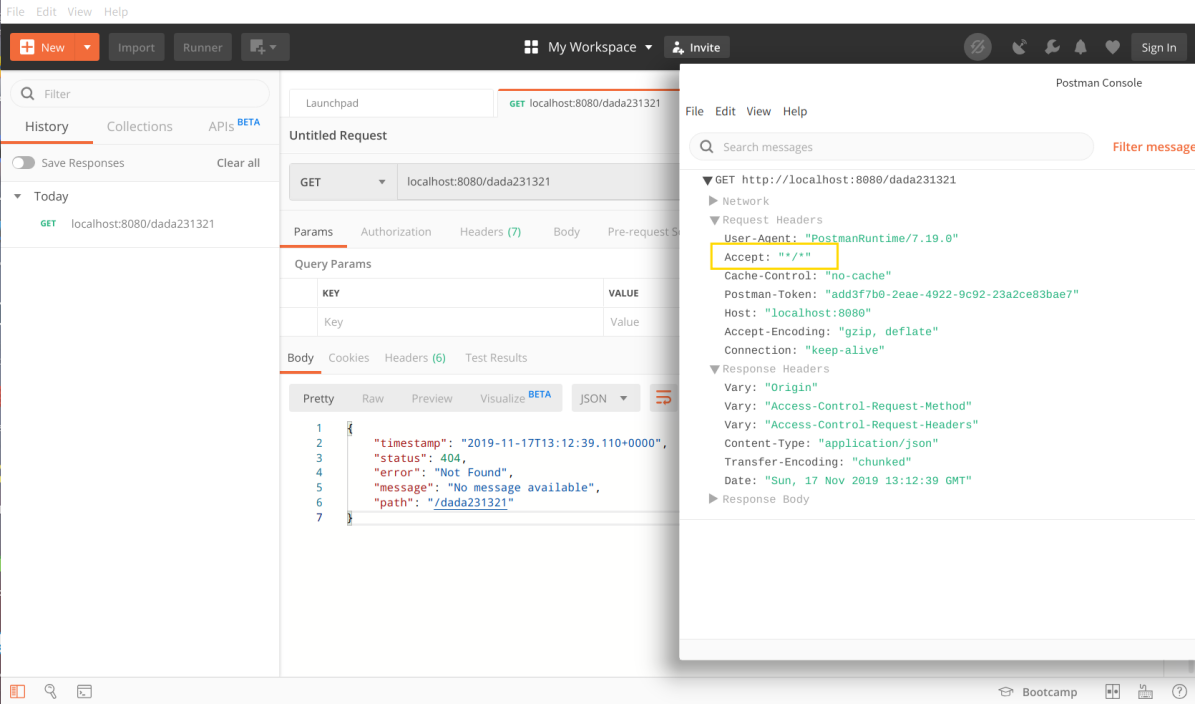
当访问一个不存在的页面，或者程序抛出异常时

**认效果：**

浏览器返回一个默认的错误页面， 注意看浏览器发送请求的请求头：



其他客户端返回json数据，注意看请求头



查看

org.springframework.boot.autoconfigure.web.servlet.error.ErrorMvcAutoConfiguration源码，

## 原理

**主要给日容器中注册了以下组件：**

* ErrorPageCustomizer 系统出现错误以后来到error请求进行处理；相当于（web.xml注册的错误页面规则）
* BasicErrorController 处理/error请求
* DefaultErrorViewResolver 默认的错误视图解析器
* DefaultErrorAttributes 错误信息
* defaultErrorView 默认错误视图

### ErrorPageCustomizer

@Bean  
public ErrorMvcAutoConfiguration.ErrorPageCustomizer errorPageCustomizer(DispatcherServletPath dispatcherServletPath) {  
 return new ErrorMvcAutoConfiguration.ErrorPageCustomizer(this.serverProperties, dispatcherServletPath);  
}

private static class ErrorPageCustomizer implements ErrorPageRegistrar, Ordered {  
 private final ServerProperties properties;  
 private final DispatcherServletPath dispatcherServletPath;  
  
 protected ErrorPageCustomizer(ServerProperties properties, DispatcherServletPath dispatcherServletPath) {  
 this.properties = properties;  
 this.dispatcherServletPath = dispatcherServletPath;  
 }  
  
 //注册错误页面  
 public void registerErrorPages(ErrorPageRegistry errorPageRegistry) {  
 //getPath()获取到的是"/error"，见下图  
 ErrorPage errorPage = new ErrorPage(this.dispatcherServletPath.getRelativePath(this.properties.getError().getPath()));  
 errorPageRegistry.addErrorPages(new ErrorPage[]{errorPage});  
 }  
  
 public int getOrder() {  
 return 0;  
 }  
}



当请求出现错误后就会转发到/error

然后这个error请求就会被BasicErrorController处理；

### BasicErrorController

@Bean  
@ConditionalOnMissingBean(  
 value = {ErrorController.class},  
 search = SearchStrategy.CURRENT  
)  
public BasicErrorController basicErrorController(ErrorAttributes errorAttributes, ObjectProvider<ErrorViewResolver> errorViewResolvers) {  
 return new BasicErrorController(errorAttributes, this.serverProperties.getError(), (List)errorViewResolvers.orderedStream().collect(Collectors.toList()));  
}

处理/error请求

@Controller  
*/\*\*  
 \* 使用配置文件中server.error.path配置  
 \* 如果server.error.path没有配置使用error.path  
 \* 如果error.path也没有配置就使用/error  
 \*/*@RequestMapping({"${server.error.path:${error.path:/error}}"})  
public class BasicErrorController extends AbstractErrorController



这两个方法一个用于浏览器请求响应html页面，一个用于其他客户端请求响应json数据。

处理浏览器请求的方法 中，modelAndView存储到哪个页面的页面地址和页面内容数据。

看一下调用的resolveErrorView方法。

protected ModelAndView resolveErrorView(HttpServletRequest request, HttpServletResponse response, HttpStatus status, Map<String, Object> model) {  
 Iterator var5 = this.errorViewResolvers.iterator();  
  
 ModelAndView modelAndView;  
 do {  
 if (!var5.hasNext()) {  
 return null;  
 }  
  
 ErrorViewResolver resolver = (ErrorViewResolver)var5.next();  
 //从所有的ErrorViewResolver得到ModelAndView  
 modelAndView = resolver.resolveErrorView(request, status, model);  
 } while(modelAndView == null);  
  
 return modelAndView;  
}

ErrorViewResolver从哪里来的呢？

已经在容器中注册了一个DefaultErrorViewResolver

### DefaultErrorViewResolver

@Configuration(  
 proxyBeanMethods = false  
)  
static class DefaultErrorViewResolverConfiguration {  
 private final ApplicationContext applicationContext;  
 private final ResourceProperties resourceProperties;  
  
 DefaultErrorViewResolverConfiguration(ApplicationContext applicationContext, ResourceProperties resourceProperties) {  
 this.applicationContext = applicationContext;  
 this.resourceProperties = resourceProperties;  
 }  
  
 //注册默认错误视图解析器  
 @Bean  
 @ConditionalOnBean({DispatcherServlet.class})  
 @ConditionalOnMissingBean({ErrorViewResolver.class})  
 DefaultErrorViewResolver conventionErrorViewResolver() {  
 return new DefaultErrorViewResolver(this.applicationContext, this.resourceProperties);  
 }  
}

然后调用ErrorViewResolver的resolveErrorView()方法

public ModelAndView resolveErrorView(HttpServletRequest request, HttpStatus status, Map<String, Object> model) {  
 //把状态码和model传过去获取视图  
 ModelAndView modelAndView = this.resolve(String.valueOf(status.value()), model);  
  
 //上面没有获取到视图就使用把状态吗替换再再找，以4开头的替换为4xx，5开头替换为5xx，见下文（如果定制错误响应）  
 if (modelAndView == null && SERIES\_VIEWS.containsKey(status.series())) {  
 modelAndView = this.resolve((String)SERIES\_VIEWS.get(status.series()), model);  
 }  
  
 return modelAndView;  
}  
  
private ModelAndView resolve(String viewName, Map<String, Object> model) {  
 //viewName传过来的是状态码，例：/error/404  
 String errorViewName = "error/" + viewName;  
 TemplateAvailabilityProvider provider = this.templateAvailabilityProviders.getProvider(errorViewName, this.applicationContext);  
 //模板引擎可以解析这个页面地址就用模板引擎解析  
 return provider != null ? new ModelAndView(errorViewName, model) : this.resolveResource(errorViewName, model);  
}

如果模板引擎不可用，就调用resolveResource方法获取视图

private ModelAndView resolveResource(String viewName, Map<String, Object> model) {  
 //获取的是静态资源文件夹  
 String[] var3 = this.resourceProperties.getStaticLocations();  
 int var4 = var3.length;  
  
 for(int var5 = 0; var5 < var4; ++var5) {  
 String location = var3[var5];  
  
 try {  
 Resource resource = this.applicationContext.getResource(location);  
 //例：static/error.html  
 resource = resource.createRelative(viewName + ".html");  
 //存在则返回视图  
 if (resource.exists()) {  
 return new ModelAndView(new DefaultErrorViewResolver.HtmlResourceView(resource), model);  
 }  
 } catch (Exception var8) {  
 }  
 }  
  
 return null;  
}

### 如何定制错误响应页面

有模板引擎的情况下；将错误页面命名为 错误状态码.html 放在模板引擎文件夹里面的 error文件夹下发生此状态码的错误就会来到这里找对应的页面；

比如我们在template文件夹下创建error/404.html当浏览器请求是404错误，就会使用我们创建的404.html页面响应，如果是其他状态码错误，还是使用默认的视图，但是如果404.html没有找到就会替换成4XX.html再查找一次，看DefaultErrorViewResolver中的静态代码块。

static {  
 Map<Series, String> views = new EnumMap(Series.class);  
 views.put(Series.CLIENT\_ERROR, "4xx");  
 views.put(Series.SERVER\_ERROR, "5xx");  
 SERIES\_VIEWS = Collections.unmodifiableMap(views);  
 }  
  
 .....  
 //再看解析方法  
 //把状态码和model传过去过去视图  
 ModelAndView modelAndView = this.resolve(String.valueOf(status.value()), model);  
  
 //上面没有获取到视图就把状态吗替换再找，以4开头的替换为4xx，5开头替换为5xx，见下文（如果定制错误响应）  
 if (modelAndView == null && SERIES\_VIEWS.containsKey(status.series())) {  
 modelAndView = this.resolve((String)SERIES\_VIEWS.get(status.series()), model);  
 }

没有模板引擎（模板引擎找不到这个错误页面），静态资源文件夹下找；

以上都没有错误页面，就是默认来到SpringBoot默认的错误提示页面；

### DefaultErrorAttributes

再看一下BasicErrorController的errorHtml方法

public ModelAndView errorHtml(HttpServletRequest request, HttpServletResponse response) {  
 HttpStatus status = this.getStatus(request);  
  
 //model的数据  
 Map<String, Object> model = Collections.unmodifiableMap(this.getErrorAttributes(request, this.isIncludeStackTrace(request, MediaType.TEXT\_HTML)));  
 response.setStatus(status.value());  
 ModelAndView modelAndView = this.resolveErrorView(request, response, status, model);  
 return modelAndView != null ? modelAndView : new ModelAndView("error", model);  
}

看一下调用的this.getErrorAttributes()方法

protected Map<String, Object> getErrorAttributes(HttpServletRequest request, boolean includeStackTrace) {  
 WebRequest webRequest = new ServletWebRequest(request);  
 return this.errorAttributes.getErrorAttributes(webRequest, includeStackTrace);  
}

再看 this.errorAttributes.getErrorAttributes()方法， this.errorAttributes是接口类型ErrorAttributes，实现类就一个DefaultErrorAttributes，看一下DefaultErrorAttributes的 getErrorAttributes()方法

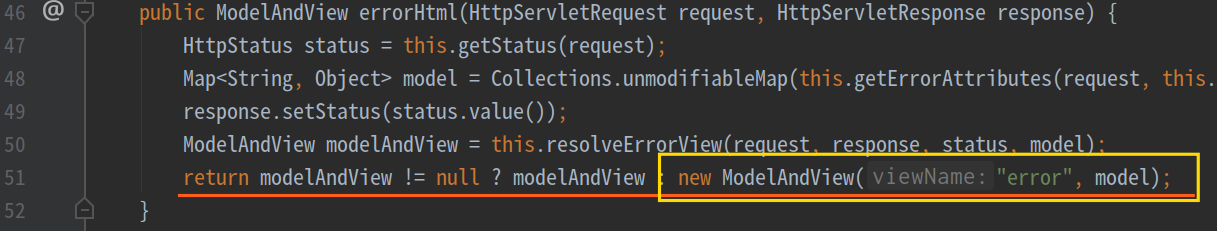
public Map<String, Object> getErrorAttributes(WebRequest webRequest, boolean includeStackTrace) {  
 Map<String, Object> errorAttributes = new LinkedHashMap();  
 errorAttributes.put("timestamp", new Date());  
 this.addStatus(errorAttributes, webRequest);  
 this.addErrorDetails(errorAttributes, webRequest, includeStackTrace);  
 this.addPath(errorAttributes, webRequest);  
 return errorAttributes;  
}

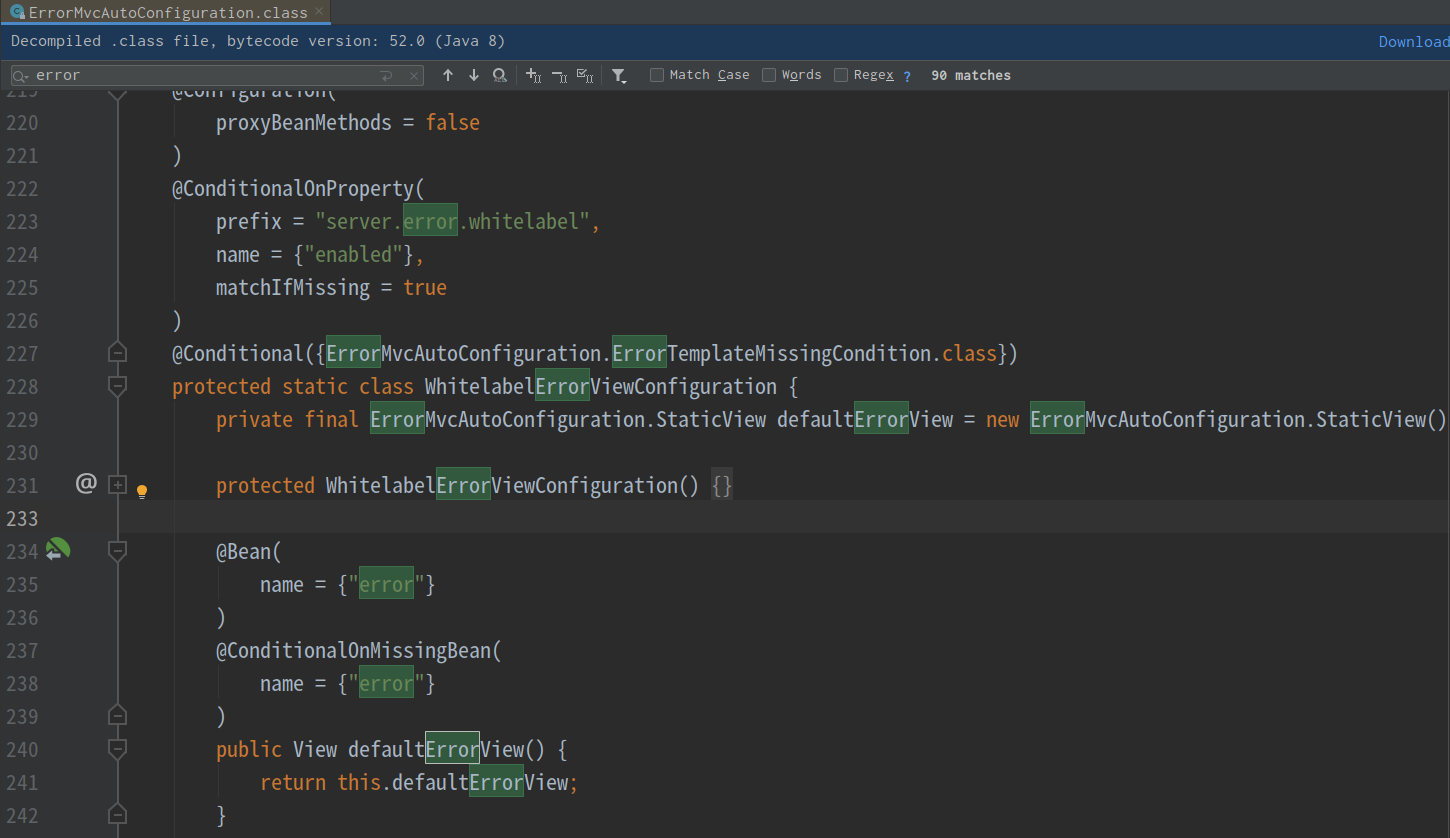
* timestamp：时间戳
* status：状态码
* error：错误提示
* exception：异常对象
* message：异常消息
* errors：JSR303数据校验的错误都在这里

2.0以后默认是不显示exception的，需要在配置文件中开启

server.error.include-exception=true

### defaultErrorView





### 如何定制JSON数据

springboot做了自适应效果，浏览器访问响应错误页面。客户端访问响应错误信息的json数据

1.第一种方法，定义全局异常处理器类注入到容器中，捕获到异常返回json格式的数据

@ControllerAdvice  
public class MyExceptionHandler {  
  
 @ResponseBody  
 @ExceptionHandler(Exception.class)  
 public Map<String, Object> handleException(Exception e) {  
 Map<String, Object> map = new HashMap(2);  
 map.put("code", "100011");  
 map.put("msg", e.getMessage());  
 return map;  
 }  
}

访问localhost:8080/hello?str=hi

@RestController  
public class Hello {  
  
 @RequestMapping("/hello")  
 public String hello(String str) {  
 if ("hi".equals(str)) {  
 int i = 10 / 0;  
 }  
 return "hello world";  
 }  
}

这样的话，不管是浏览器访问还是客户端访问都是响应json数据，就没有了自适应效果

2.第二种方法，捕获到异常后转发到/error

@ControllerAdvice  
public class MyExceptionHandler {  
  
 @ExceptionHandler(Exception.class)  
 public String handleException(Exception e) {  
 Map<String, Object> map = new HashMap(2);  
 map.put("code", "100011");  
 map.put("msg", e.getMessage());  
 return "forward:/error";  
 }  
}

访问localhost:8080/hello?str=hi，但这样异常被我们捕获然后转发，显示的状态码就是200，所以在转发之前还要设置一下状态码

@ExceptionHandler(Exception.class)  
public String handleException(Exception e, HttpServletRequest request) {  
 Map<String, Object> map = new HashMap(2);  
 map.put("code", "100011");  
 map.put("msg", e.getMessage());  
  
 //设置状态码  
 request.setAttribute("javax.servlet.error.status\_code", 500);  
 return "forward:/error";  
}

但是设置的数据就没有用了，只能使用默认的

由上面我们已经知道数据的来源是调用DefaultErrorAttributes的getErrorAttributes方法得到的，而这个DefaultErrorAttributes是在ErrorMvcAutoConfiguration配置类中注册的，并且注册之前会检查容器中是否已经拥有。

@Bean  
@ConditionalOnMissingBean(  
 value = {ErrorAttributes.class},  
 search = SearchStrategy.*CURRENT*)  
public DefaultErrorAttributes errorAttributes() {  
 return new DefaultErrorAttributes(this.serverProperties.getError().isIncludeException());  
}

所以我们可以只要实现ErrorAttributes接口或者继承DefaultErrorAttributes类，然后注册到容器中就行了

@ControllerAdvice  
public class MyExceptionHandler {  
  
 @ExceptionHandler(Exception.class)  
 public String handleException(Exception e, HttpServletRequest request) {  
 Map<String, Object> map = new HashMap(2);  
 map.put("name", "hello");  
 map.put("password", "123456");  
  
 //设置状态码  
 request.setAttribute("javax.servlet.error.status\_code", 500);  
  
 //把数据放到request域中  
 request.setAttribute("ext", map);  
 return "forward:/error";  
 }  
}

@Configuration  
public class MyMvcConfig implements WebMvcConfigurer {  
  
 @Bean  
 public DefaultErrorAttributes errorAttributes() {  
 return new MyErrorAttributes();  
 }  
  
 class MyErrorAttributes extends DefaultErrorAttributes {  
 @Override  
 public Map<String, Object> getErrorAttributes(WebRequest webRequest, boolean includeStackTrace) {  
 //调用父类的方法获取默认的数据  
 Map<String, Object> map = new HashMap<>(super.getErrorAttributes(webRequest, includeStackTrace));  
 //从request域从获取到自定义数据  
 Map<String, Object> ext = (Map<String, Object>) webRequest.getAttribute("ext", RequestAttributes.*SCOPE\_REQUEST*);  
 map.putAll(ext);  
 return map;  
 }  
 }  
  
 ......

# 配置嵌入式Servlet容器

## 如何定制和修改Servlet容器的相关配置

1. 修改和server有关的配置

server.port=8081  
server.context-path=/crud  
  
server.tomcat.uri-encoding=UTF-8  
  
//通用的Servlet容器设置  
server.xxx  
//Tomcat的设置  
server.tomcat.xxx

1. 编写一个EmbeddedServletContainerCustomizer，2.0以后改为WebServerFactoryCustomizer：嵌入式的Servlet容器的定制器；来修改Servlet容器的配置

@Configuration  
public class MyMvcConfig implements WebMvcConfigurer {  
 @Bean  
 public WebServerFactoryCustomizer webServerFactoryCustomizer() {  
 return new WebServerFactoryCustomizer<ConfigurableWebServerFactory>() {  
 @Override  
 public void customize(ConfigurableWebServerFactory factory) {  
 factory.setPort(8088);  
 }  
 };  
 }  
......

## 注册Servlet三大组件

由于SpringBoot默认是以jar包的方式启动嵌入式的Servlet容器来启动SpringBoot的web应用，没有web.xml文件。

### Servlet

向容器中添加ServletRegistrationBean

@Configuration  
public class MyMvcConfig implements WebMvcConfigurer {  
  
 @Bean  
 public ServletRegistrationBean myServlet() {  
 ServletRegistrationBean register = new ServletRegistrationBean(new MyServlet(), "/myServlet");  
 register.setLoadOnStartup(1);  
 return register;  
 }  
 ......

MyServlet

public class MyServlet extends HttpServlet {  
  
 @Override  
 public void init() throws ServletException {  
 System.*out*.println("servlet初始化");  
 }  
  
 @Override  
 protected void doGet(HttpServletRequest req, HttpServletResponse resp) throws ServletException, IOException {  
 this.doPost(req, resp);  
 }  
  
 @Override  
 protected void doPost(HttpServletRequest req, HttpServletResponse resp) throws ServletException, IOException {  
 resp.getWriter().write("this is MyServlet");  
 }  
  
}

### Filter

向容器中添加FilterRegistrationBean

@Configuration  
public class MyMvcConfig implements WebMvcConfigurer {  
  
  
 @Bean  
 public FilterRegistrationBean myFilter() {  
 FilterRegistrationBean register = new FilterRegistrationBean(new MyFilter());  
 register.setUrlPatterns(Arrays.asList("/myServlet","/"));  
 return register;  
 }  
  
 ......

MyFilter

public class MyFilter extends HttpFilter {  
 @Override  
 public void doFilter(ServletRequest request, ServletResponse response, FilterChain chain) throws IOException, ServletException {  
 response.getWriter().write("请求被拦截......");  
 }  
}

### Listener

向容器中注入ServletListenerRegistrationBean

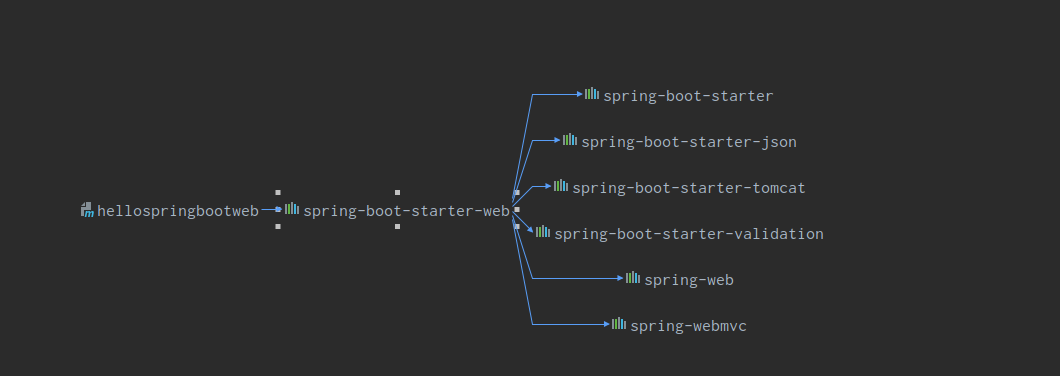
@Configuration  
public class MyMvcConfig implements WebMvcConfigurer {  
  
 @Bean  
 public ServletListenerRegistrationBean myServletContextListener(){  
 return new ServletListenerRegistrationBean(new MyServletContextListener());  
 }  
  
 ......

MyListener

public class MyServletContextListener implements ServletContextListener {  
 @Override  
 public void contextInitialized(ServletContextEvent sce) {  
 System.*out*.println("web容器 启动......");  
 }  
  
 @Override  
 public void contextDestroyed(ServletContextEvent sce) {  
 System.*out*.println("web容器 销毁......");  
 }  
}

## 替换为其他嵌入式web服务器

SpringBoot默认使用的是Tomcat



如果要换成其他的就把Tomcat的依赖排除掉，然后引入其他嵌入式Servlet容器的以来，如Jetty，Undertow

<dependency>  
 <groupId>org.springframework.boot</groupId>  
 <artifactId>spring-boot-starter-web</artifactId>  
 <exclusions>  
 <exclusion>  
 <artifactId>spring-boot-starter-tomcat</artifactId>  
 <groupId>org.springframework.boot</groupId>  
 </exclusion>  
 </exclusions>  
</dependency>  
  
<dependency>  
 <artifactId>spring-boot-starter-jetty</artifactId>  
 <groupId>org.springframework.boot</groupId>  
</dependency>

### 原理

查看web容器自动配置类

2.0以下是：EmbeddedServletContainerAutoConfiguration

ServletWebServerFactoryAutoConfiguration：嵌入式的web服务器自动配置

@Configuration(  
 proxyBeanMethods = false  
)  
@AutoConfigureOrder(-2147483648)  
@ConditionalOnClass({ServletRequest.class})  
@ConditionalOnWebApplication(  
 type = Type.SERVLET  
)  
@EnableConfigurationProperties({ServerProperties.class})  
  
//---看这里---  
@Import({ServletWebServerFactoryAutoConfiguration.BeanPostProcessorsRegistrar.class, EmbeddedTomcat.class, EmbeddedJetty.class, EmbeddedUndertow.class})  
public class ServletWebServerFactoryAutoConfiguration {

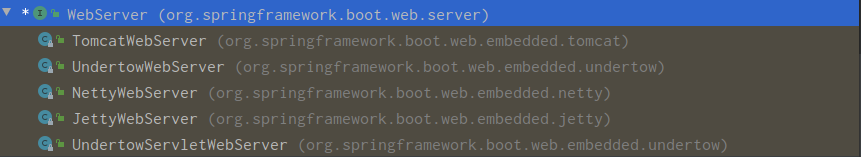
EmbeddedTomcat.class：

@Configuration(  
 proxyBeanMethods = false  
)  
//判断当前是否引入了Tomcat依赖；  
@ConditionalOnClass({Servlet.class, Tomcat.class, UpgradeProtocol.class})  
*/\*\*  
 \*判断当前容器没有用户自己定义ServletWebServerFactory：嵌入式的web服务器工厂；  
 \*作用：创建嵌入式的web服务器  
 \*/*@ConditionalOnMissingBean(  
 value = {ServletWebServerFactory.class},  
 search = SearchStrategy.CURRENT  
)  
public static class EmbeddedTomcat {

ServletWebServerFactory：嵌入式的web服务器工厂

@FunctionalInterface  
public interface ServletWebServerFactory {  
 //获取嵌入式的Servlet容器  
 WebServer getWebServer(ServletContextInitializer... initializers);  
}

WebServer：嵌入式的web服务器实现



以TomcatServletWebServerFactory为例，下面是

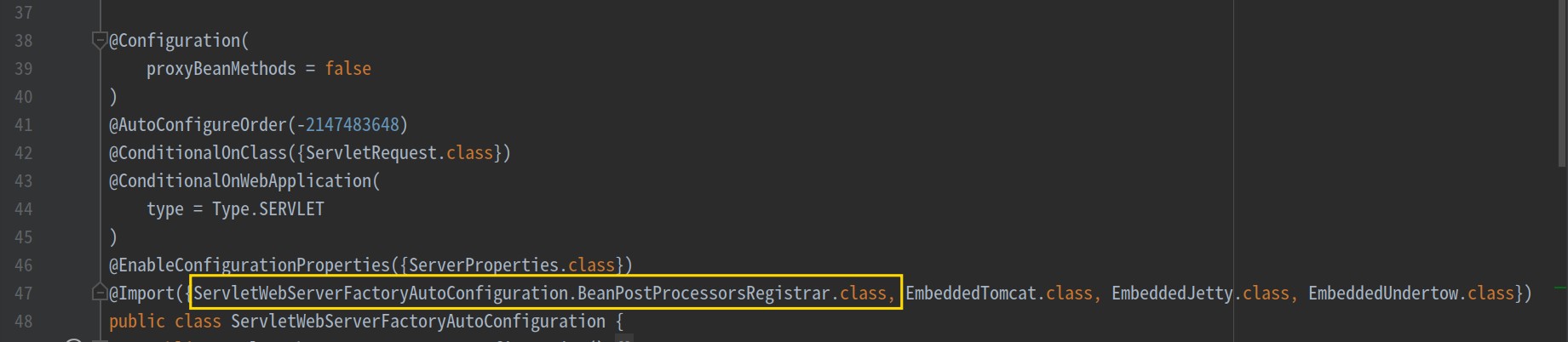
TomcatServletWebServerFactory类

public WebServer getWebServer(ServletContextInitializer... initializers) {  
 if (this.disableMBeanRegistry) {  
 Registry.disableRegistry();  
 }  
  
 //创建一个Tomcat  
 Tomcat tomcat = new Tomcat();  
  
 //配置Tomcat的基本环境，（tomcat的配置都是从本类获取的，tomcat.setXXX）  
 File baseDir = this.baseDirectory != null ? this.baseDirectory : this.createTempDir("tomcat");  
 tomcat.setBaseDir(baseDir.getAbsolutePath());  
 Connector connector = new Connector(this.protocol);  
 connector.setThrowOnFailure(true);  
 tomcat.getService().addConnector(connector);  
 this.customizeConnector(connector);  
 tomcat.setConnector(connector);  
 tomcat.getHost().setAutoDeploy(false);  
 this.configureEngine(tomcat.getEngine());  
 Iterator var5 = this.additionalTomcatConnectors.iterator();  
  
 while(var5.hasNext()) {  
 Connector additionalConnector = (Connector)var5.next();  
 tomcat.getService().addConnector(additionalConnector);  
 }  
  
 this.prepareContext(tomcat.getHost(), initializers);  
  
 //将配置好的Tomcat传入进去，返回一个WebServer；并且启动Tomcat服务器  
 return this.getTomcatWebServer(tomcat);  
}

我们对嵌入式容器的配置修改是怎么生效的？

### 配置修改原理

ServletWebServerFactoryAutoConfiguration在向容器中添加web容器时还添加了一个组件



BeanPostProcessorsRegistrar：后置处理器注册器(也是给容器注入一些组件)

public static class BeanPostProcessorsRegistrar implements ImportBeanDefinitionRegistrar, BeanFactoryAware {  
 private ConfigurableListableBeanFactory beanFactory;  
  
 public BeanPostProcessorsRegistrar() {...}  
  
 public void setBeanFactory(BeanFactory beanFactory) throws BeansException {...}  
  
 public void registerBeanDefinitions(AnnotationMetadata importingClassMetadata, BeanDefinitionRegistry registry) {  
 if (this.beanFactory != null) {  
 //注册了下面两个组件  
 this.registerSyntheticBeanIfMissing(registry, "webServerFactoryCustomizerBeanPostProcessor", WebServerFactoryCustomizerBeanPostProcessor.class);  
 this.registerSyntheticBeanIfMissing(registry, "errorPageRegistrarBeanPostProcessor", ErrorPageRegistrarBeanPostProcessor.class);  
 }  
 }  
  
 private void registerSyntheticBeanIfMissing(BeanDefinitionRegistry registry, String name, Class<?> beanClass) {...}  
}

webServerFactoryCustomizerBeanPostProcessor

public class WebServerFactoryCustomizerBeanPostProcessor implements BeanPostProcessor, BeanFactoryAware {  
  
 ......  
  
 //在Bean初始化之前  
 public Object postProcessBeforeInitialization(Object bean, String beanName) throws BeansException {  
 //判断添加的Bean是不是WebServerFactory类型的  
 if (bean instanceof WebServerFactory) {  
 this.postProcessBeforeInitialization((WebServerFactory)bean);  
 }  
  
 return bean;  
 }  
  
 public Object postProcessAfterInitialization(Object bean, String beanName) throws BeansException {  
 return bean;  
 }  
  
 private void postProcessBeforeInitialization(WebServerFactory webServerFactory) {  
 //获取所有的定制器，调用每一个定制器的customize方法来给Servlet容器进行属性赋值；  
 ((Callbacks)LambdaSafe.callbacks(WebServerFactoryCustomizer.class, this.getCustomizers(), webServerFactory, new Object[0]).withLogger(WebServerFactoryCustomizerBeanPostProcessor.class)).invoke((customizer) -> {  
 customizer.customize(webServerFactory);  
 });  
 }

关于配置文件是如何设置的，参考

EmbeddedWebServerFactoryCustomizerAutoConfiguration类，最后还是使用上面的方便

总结：

SpringBoot根据导入的依赖情况，给容器中添加相应的

XXXServletWebServerFactory

容器中某个组件要创建对象就会惊动后置处理器

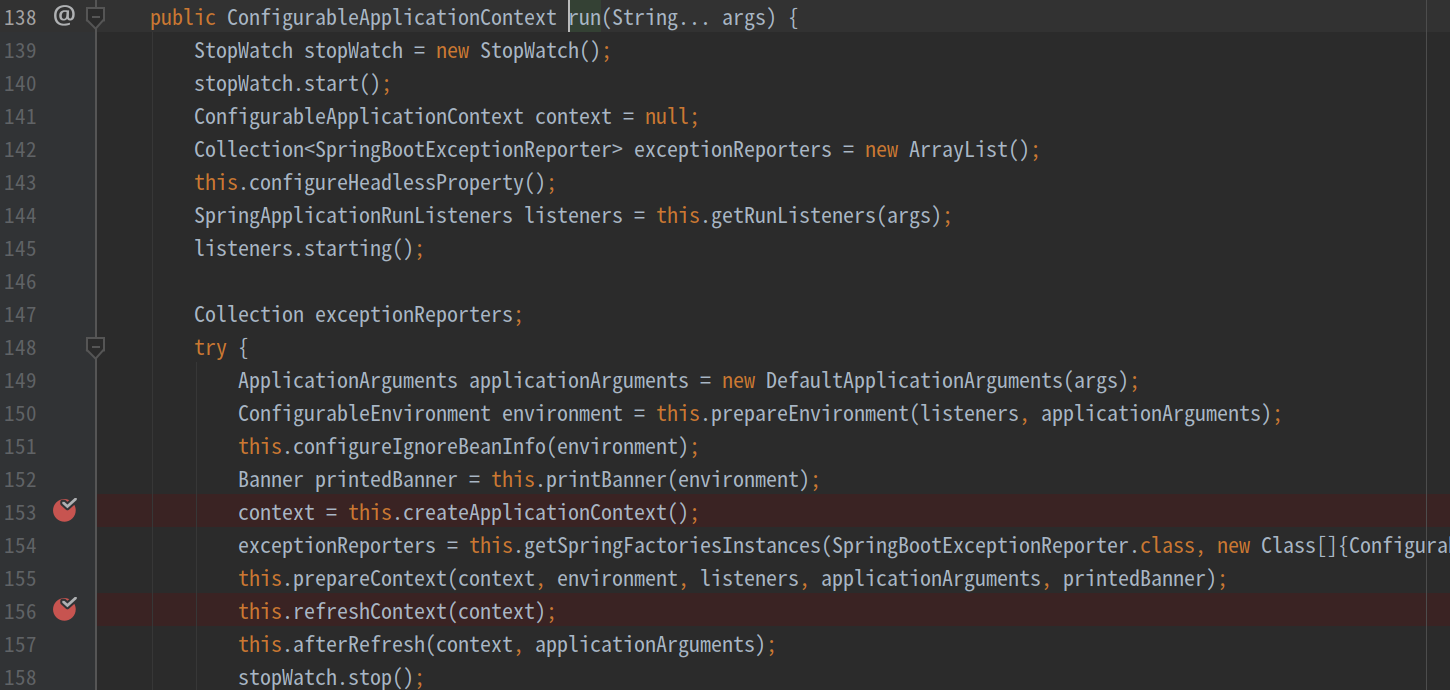
webServerFactoryCustomizerBeanPostProcessor

只要是嵌入式的是Servlet容器工厂，后置处理器就会工作；

后置处理器，从容器中获取所有的WebServerFactoryCustomizer，调用定制器的定制方法给工厂添加配置

### 嵌入式Servlet容器启动原理

SpringBoot应用启动运行run方法



153行，创建IOC容器对象，根据当前环境创建

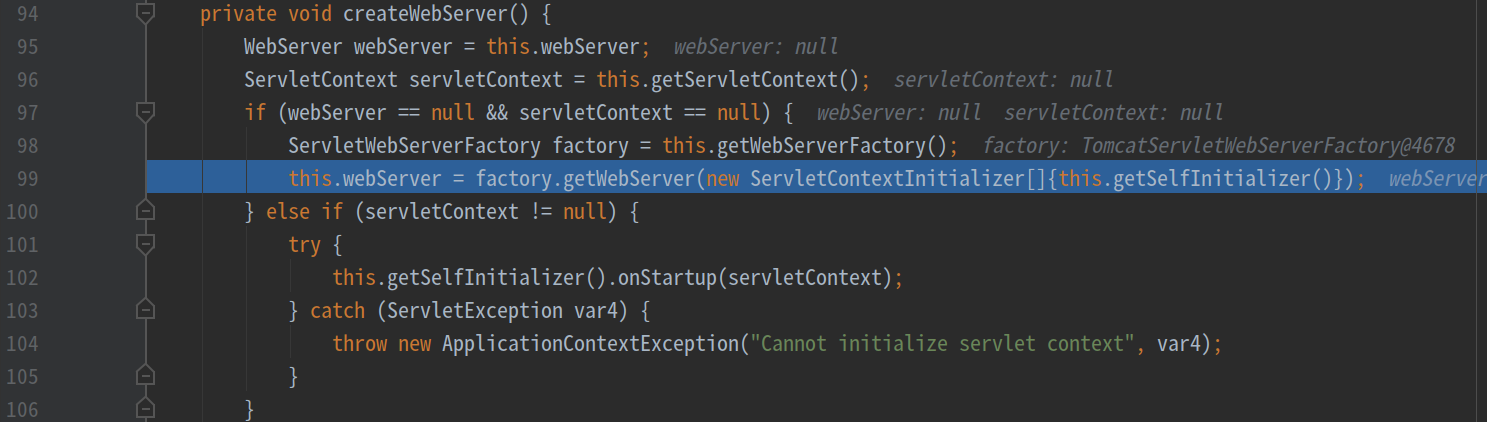


156行，刷新IOC容器

刷新IOC容器中272行，onRefresh()；web的ioc容器重写了onRefresh方法，查看ServletWebServerApplicationContext类的onRefresh方法，在方法中调用了this.createWebServer();方法创建web容器

protected void onRefresh() {  
 super.onRefresh();  
  
 try {  
 this.createWebServer();  
 } catch (Throwable var2) {  
 throw new ApplicationContextException("Unable to start web server", var2);  
 }  
}

98行获取嵌入式的web容器工厂



接下来就是上面的上面的相关配置流程，在创建web容器工厂时会触发webServerFactoryCustomizerBeanPostProcessor

然后99行使用容器工厂获取嵌入式的Servlet容器

嵌入式的Servlet容器创建对象并启动Servlet容器；

嵌入式的Servlet容器启动后，再将ioc容器中剩下没有创建出的对象获取出来(Controller,Service等)；

### 使用外置的Servlet容器

将项目的打包方式改为war

编写一个类继承SpringBootServletInitializer，并重写configure方法，调用参数的sources方法springboot启动类传过去然后返回

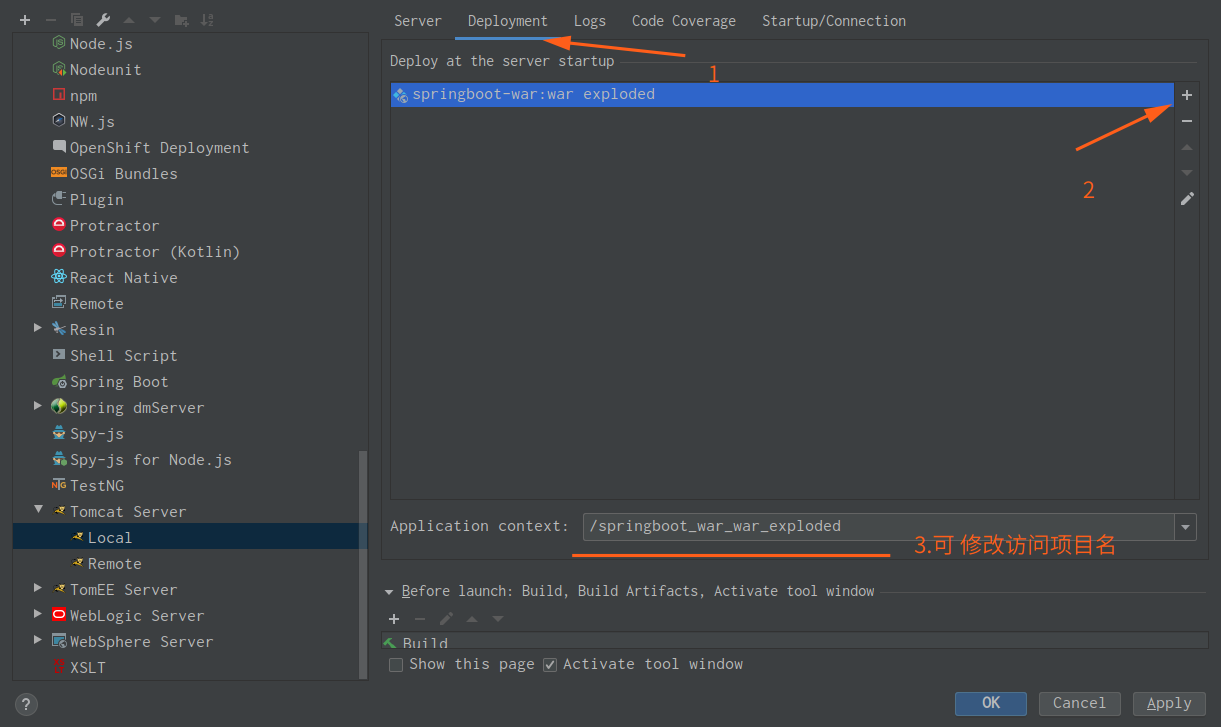
public class ServletInitializer extends SpringBootServletInitializer {  
 @Override  
 protected SpringApplicationBuilder configure(SpringApplicationBuilder application) {  
 return application.sources(HelloSpringBootWebApplication.class);  
 }  
}

然后把tomcat的依赖范围改为provided

<dependencies>  
<dependency>  
 <groupId>org.springframework.boot</groupId>  
 <artifactId>spring-boot-starter-web</artifactId>  
</dependency>  
<dependency>  
 <groupId>org.springframework.boot</groupId>  
 <artifactId>spring-boot-starter-tomcat</artifactId>  
 <version>2.2.1.RELEASE</version>  
 <scope>provided</scope>  
</dependency>  
  
......  
  
</dependencies>

最后就可以把项目打包成war放到tomcat中了

在IDEA中可以这样配置



在创建项目时使用Spring Initializr创建选择打包方式为war，1，2，3步骤会自动配置

如果启动tomcat，报了一大堆错误，不妨把Tomcat改为更高的版本试试，如果你项目中的Filter是继承了HttpFilter，请使用tomcat9版本，9以下好像没有HttpFilter

#### 原理

Servlet3.0标准ServletContainerInitializer扫描所有jar包中METAINF/services/javax.servlet.ServletContainerInitializer文件指定的类并加载

还可以使用@HandlesTypes，在应用启动的时候加载我们感兴趣的类；

在spring-web-xxx.jar包中的METAINF/services下有

javax.servlet.ServletContainerInitializer这个文件

文件中的类是：

org.springframework.web.SpringServletContainerInitializer

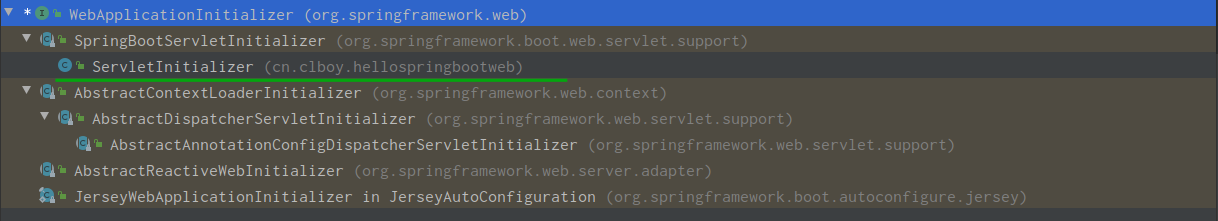
对应的类：

@HandlesTypes({WebApplicationInitializer.class})  
public class SpringServletContainerInitializer implements ServletContainerInitializer {  
 public SpringServletContainerInitializer() {  
 }  
  
 public void onStartup(@Nullable Set<Class<?>> webAppInitializerClasses, ServletContext servletContext) throws ServletException {  
  
 ......

SpringServletContainerInitializer将@HandlesTypes(WebApplicationInitializer.class)标注的所有这个类型的类都传入到onStartup方法的Set<Class<?>>；为这些WebApplicationInitializer类型的类创建实例；

每一个WebApplicationInitializer都调用自己的onStartup方法；

WebApplicationInitializer的实现类



相当于我们的SpringBootServletInitializer的类会被创建对象，并执行onStartup方法

SpringBootServletInitializer实例执行onStartup的时候会createRootApplicationContext；创建容器

protected WebApplicationContext createRootApplicationContext(  
 ServletContext servletContext) {  
 //1、创建SpringApplicationBuilder  
 SpringApplicationBuilder builder = createSpringApplicationBuilder();  
 StandardServletEnvironment environment = new StandardServletEnvironment();  
 environment.initPropertySources(servletContext, null);  
 builder.environment(environment);  
 builder.main(getClass());  
 ApplicationContext parent = getExistingRootWebApplicationContext(servletContext);  
 if (parent != null) {  
 this.logger.info("Root context already created (using as parent).");  
 servletContext.setAttribute(  
 WebApplicationContext.ROOT\_WEB\_APPLICATION\_CONTEXT\_ATTRIBUTE, null);  
 builder.initializers(new ParentContextApplicationContextInitializer(parent));  
 }  
 builder.initializers(  
 new ServletContextApplicationContextInitializer(servletContext));  
 builder.contextClass(AnnotationConfigEmbeddedWebApplicationContext.class);  
  
 //调用configure方法，子类重写了这个方法，将SpringBoot的主程序类传入了进来  
 builder = configure(builder);  
  
 //使用builder创建一个Spring应用  
 SpringApplication application = builder.build();  
 if (application.getSources().isEmpty() && AnnotationUtils  
 .findAnnotation(getClass(), Configuration.class) != null) {  
 application.getSources().add(getClass());  
 }  
 Assert.state(!application.getSources().isEmpty(),  
 "No SpringApplication sources have been defined. Either override the "  
 + "configure method or add an @Configuration annotation");  
 // Ensure error pages are registered  
 if (this.registerErrorPageFilter) {  
 application.getSources().add(ErrorPageFilterConfiguration.class);  
 }  
 //启动Spring应用  
 return run(application);  
}

Spring的应用就启动并且创建IOC容器

public ConfigurableApplicationContext run(String... args) {  
 StopWatch stopWatch = new StopWatch();  
 stopWatch.start();  
 ConfigurableApplicationContext context = null;  
 FailureAnalyzers analyzers = null;  
 configureHeadlessProperty();  
 SpringApplicationRunListeners listeners = getRunListeners(args);  
 listeners.starting();  
 try {  
 ApplicationArguments applicationArguments = new DefaultApplicationArguments(  
 args);  
 ConfigurableEnvironment environment = prepareEnvironment(listeners,  
 applicationArguments);  
 Banner printedBanner = printBanner(environment);  
 context = createApplicationContext();  
 analyzers = new FailureAnalyzers(context);  
 prepareContext(context, environment, listeners, applicationArguments,  
 printedBanner);  
  
 //刷新IOC容器  
 refreshContext(context);  
 afterRefresh(context, applicationArguments);  
 listeners.finished(context, null);  
 stopWatch.stop();  
 if (this.logStartupInfo) {  
 new StartupInfoLogger(this.mainApplicationClass)  
 .logStarted(getApplicationLog(), stopWatch);  
 }  
 return context;  
 }  
 catch (Throwable ex) {  
 handleRunFailure(context, listeners, analyzers, ex);  
 throw new IllegalStateException(ex);  
 }  
}

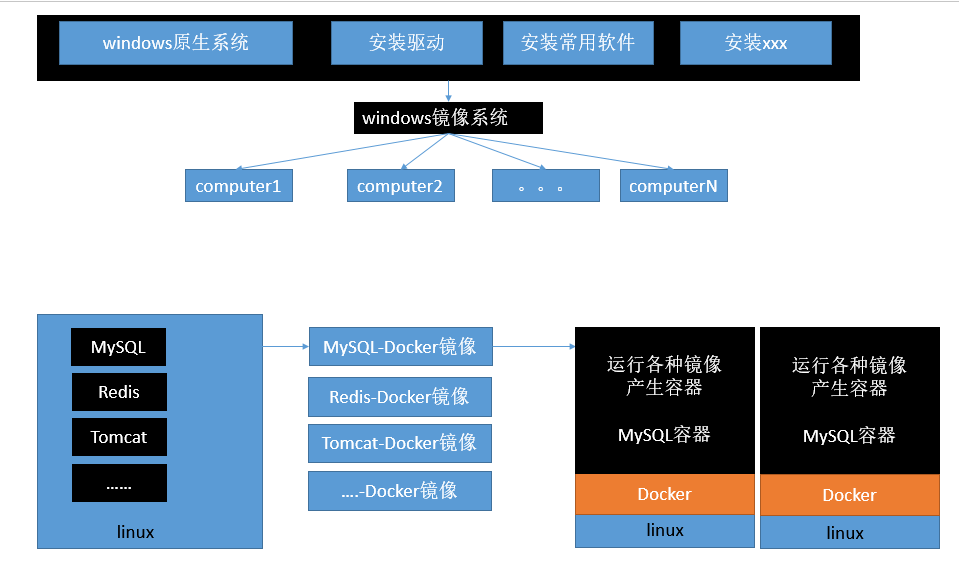
# Docker基本使用

Docker是一个开源的应用容器引擎；是一个轻量级容器技术；

Docker支持将软件编译成一个镜像；然后在镜像中各种软件做好配置，将镜像发布出去，其他使用者可以直接使用这个镜像；

运行中的这个镜像称为容器，容器启动是非常快速的。

官方网站：https://hub.docker.com/



## 核心概念

docker主机(Host)：安装了Docker程序的机器（Docker直接安装在操作系统之上）；

docker客户端(Client)：连接docker主机进行操作；

docker仓库(Registry)：用来保存各种打包好的软件镜像；

docker镜像(Images)：软件打包好的镜像；放在docker仓库中；

docker容器(Container)：镜像启动后的实例称为一个容器；容器是独立运行的一个或一组应用

## 使用Docker的步骤：

1. 确认要安装docker的系统的linux内核高于3.10，低于3.10使用yum update更新

uname -r

1. 安装docker

yum install docker

1. 查看docker版本

docker -v

1. 查看docker状态

service docker status

1. 启动docker

service docker start

1. 停止docker

service docker stop

1. 设置docker开机自启

systemctl enable docker

8.设置阿里云加速

登录阿里云找到“容器镜像服务”下的镜像加速器



sudo mkdir -p /etc/docker

sudo tee /etc/docker/daemon.json <<-'EOF'

{

"registry-mirrors": ["https://3h44xp7a.mirror.aliyuncs.com"]

}

EOF

sudo systemctl daemon-reload

sudo systemctl restart docker

## docker常用命令

### 镜像操作

|  |  |  |
| --- | --- | --- |
| **操作** | **命令** | **说明** |
| 检索 | docker search 关键字 eg：docker search redis | 我们经常去docker hub上检索镜像的详细信息，如镜像的TAG。 |
| 拉取 | docker pull 镜像名:tag | :tag是可选的，tag表示标签，多为软件的版本，默认是latest |
| 列表 | docker images | 查看所有本地镜像 |
| 删除 | docker rmi image-id | 删除指定的本地镜像 |

### 修改镜像源

修改 /etc/docker/daemon.json ，写入如下内容（如果文件不存在请新建该文件）

vim /etc/docker/daemon.json

#　内容：

{

"registry-mirrors":["https://registry.docker-cn.com"]

}

国内镜像源 地址

Docker 官方中国区 https://registry.docker-cn.com

网易 http://hub-mirror.c.163.com

中国科技大学 https://docker.mirrors.ustc.edu.cn

阿里云 <https://pee6w651.mirror.aliyuncs.com>

### 容器操作

**以tomcat为例：**

1.下载tomcat镜像

docker pull tomcat

如需选择具体版本，可以在https://hub.docker.com/搜索tomcat

2.根据镜像启动容器，不加TAG默认latest，如果没有下载latest会先去下载再启动

docker run --name mytomcat -d tomcat:latest

--name：给容器起个名字

-d：后台启动，不加就是前端启动，然后你就只能开一个新的窗口连接，不然就望着黑乎乎的窗口，啥也干不了，Ctrl+C即可退出，当然，容器也会关闭

1. 查看运行中的容器

docker ps

1. 停止运行中的容器

docker stop 容器的id

# 或者

docker stop 容器的名称，就是--name给起的哪个名字

5.查看所有的容器

docker ps -a

6.启动容器

docker start 容器id/名字

7.删除一个容器

docker rm 容器id/名字

8.启动一个做了端口映射的tomcat

docker run -d -p 8888:8080 tomcat

-d：后台运行 -p: 将主机的端口映射到容器的一个端口 主机端口(8888):容器内部的端口(8080)

外界通过主机的8888端口就可以访问到tomcat，前提是8888端口开放

9.关闭防火墙

# 查看防火墙状态

service firewalld status

# 关闭防火墙

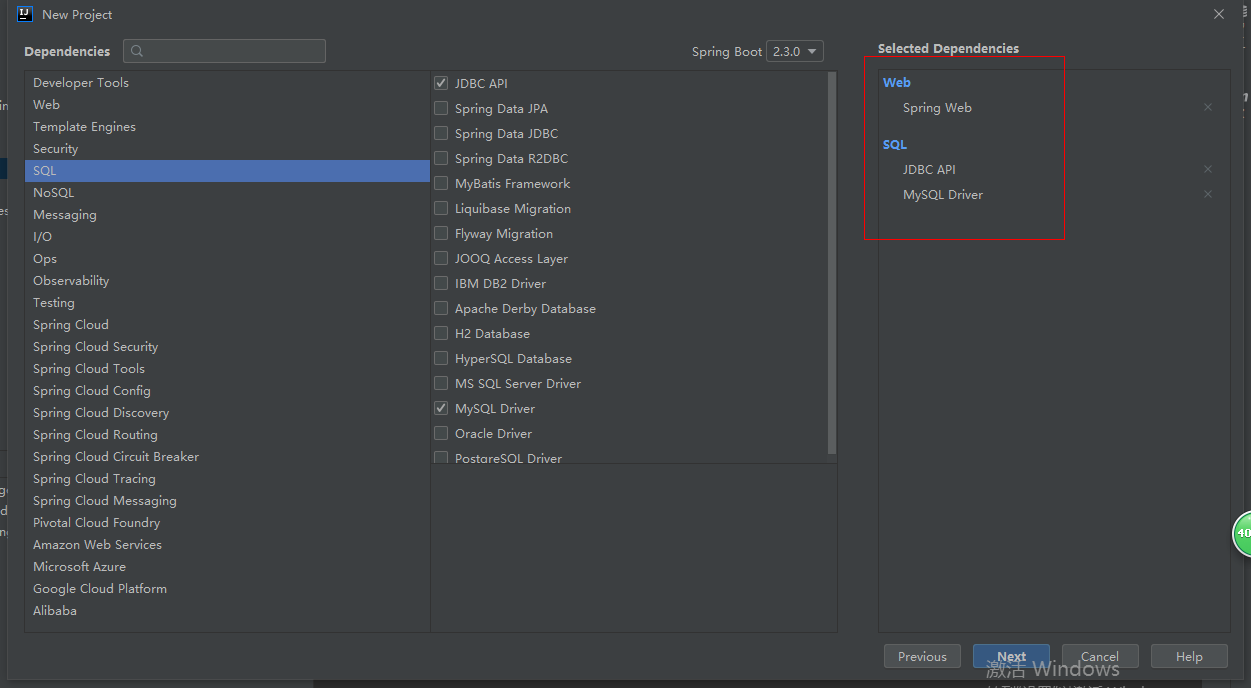
service firewalld stop

10.查看容器的日志

docker logs 容器id/名字

# SpringBoot与数据库连接(jdbc)

新建项目时选择:



## 它会导入依赖：

<dependency>  
 <groupId>org.springframework.boot</groupId>  
 <artifactId>spring-boot-starter-jdbc</artifactId>  
</dependency>

<dependency>  
 <groupId>mysql</groupId>  
 <artifactId>mysql-connector-java</artifactId>  
 <scope>runtime</scope>  
</dependency>

## 配置数据库连接信息

spring:  
 datasource:  
 username: root  
 password:  
 url: jdbc:mysql://127.0.0.1:3306/laravel?serverTimezone=UTC  
 driver-class-name: com.mysql.cj.jdbc.Driver

测试能否连接上数据库

@SpringBootTest  
class SpringbootJdbcApplicationTests {  
  
 @Autowired  
 private DataSource dataSource;  
  
 @Test  
 void contextLoads() throws SQLException {  
 System.*out*.println(dataSource.getClass());  
 System.out.println(dataSource.getConnection());  
 }  
  
}

springboot默认是使用com.zaxxer.hikari.HikariDataSource作为数据源，2.0以下是用org.apache.tomcat.jdbc.pool.DataSource作为数据源；

数据源的相关配置都在DataSourceProperties里面；

使用JDBC连接数据库出现The server time zone value ‘???’ is解决方案:

url: jdbc:mysql://127.0.0.1:3306/laravel?serverTimezone=UTC

加上后面红色部分

## 整合Druid数据源

**选择哪个数据库连接池**

* DBCP2 是 Appache 基金会下的项目，是最早出现的数据库连接池 DBCP 的第二个版本。
* C3P0 最早出现时是作为 Hibernate 框架的默认数据库连接池而进入市场。
* Druid 是阿里巴巴公司开源的一款数据库连接池，其特点在于有丰富的附加功能。
* HikariCP 相较而言比较新，它最近两年才出现，据称是速度最快的数据库连接池。最近更是被 Spring 设置为默认数据库连接池。

**不选择 C3P0 的原因**：

* C3P0 的 Connection 是异步释放。这个特性会导致释放的在某些情况下 Connection 实际上 still in use ，并未真正释放掉，从而导致连接池中的 Connection 耗完，等待状况。
* Hibernate 现在对所有数据库连接池一视同仁，官方不再指定『默认』数据库连接池。因此 C3P0 就失去了『官方』光环。

**不选择 DBCP2 的原因**：

相较于 Druid 和 HikariCP，DBCP2 没有什么特色功能/卖点。基本上属于 能用，没毛病 的情况，地位显得略有尴尬。

### 1.导入依赖

最新链接：

<https://mvnrepository.com/artifact/com.alibaba/druid-spring-boot-starter>

<!-- https://mvnrepository.com/artifact/com.alibaba/druid-spring-boot-starter -->  
<dependency>  
 <groupId>com.alibaba</groupId>  
 <artifactId>druid-spring-boot-starter</artifactId>  
 <version>1.1.22</version>  
</dependency>

### 2在配置文件中指定数据源类型

spring:  
 datasource:  
 username: root  
 password:  
 url: jdbc:mysql://127.0.0.1:3306/laravel?serverTimezone=UTC  
 driver-class-name: com.mysql.cj.jdbc.Driver  
 type: com.alibaba.druid.pool.DruidDataSource

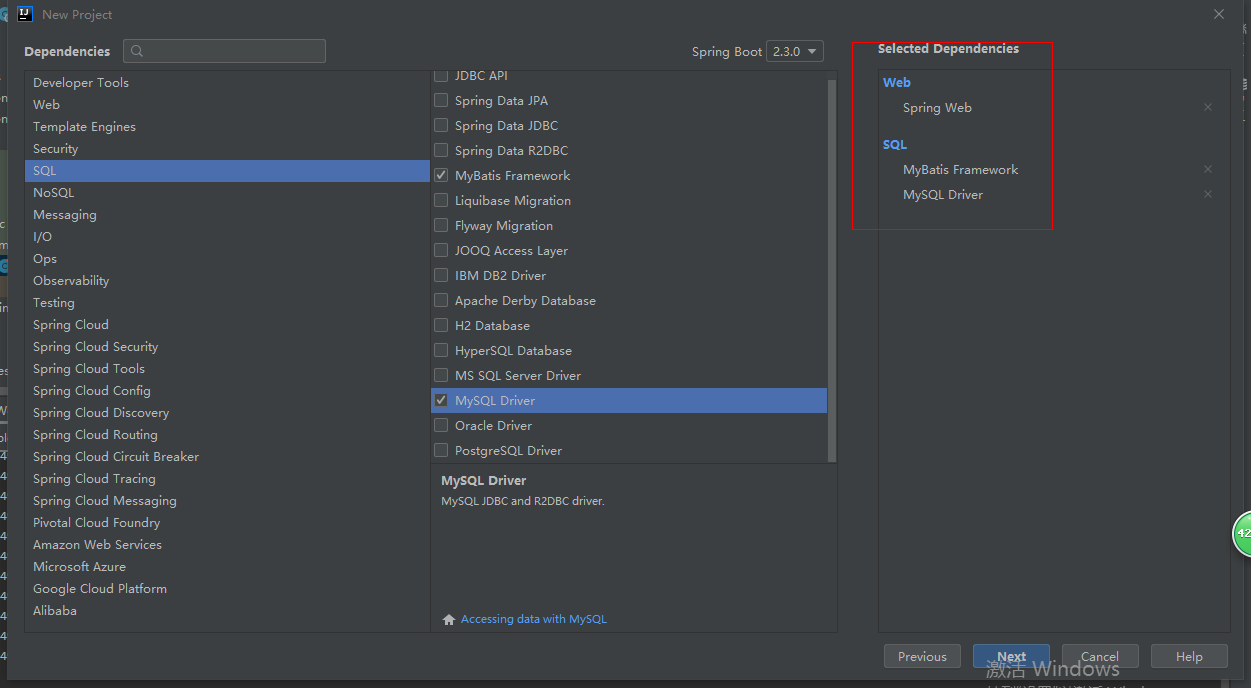
### 配置参数

spring:  
 datasource:  
 username: root  
 password:  
 url: jdbc:mysql://127.0.0.1:3306/laravel?serverTimezone=UTC  
 driver-class-name: com.mysql.cj.jdbc.Driver  
 type: com.alibaba.druid.pool.DruidDataSource  
 druid:  
 *# 连接池配置  
 # 配置初始化大小、最小、最大* initial-size: 1  
 min-idle: 1  
 max-active: 20  
 *# 配置获取连接等待超时的时间* max-wait: 3000  
 validation-query: SELECT 1 FROM DUAL  
 test-on-borrow: false  
 test-on-return: false  
 test-while-idle: true  
 pool-prepared-statements: true  
 time-between-eviction-runs-millis: 60000  
 min-evictable-idle-time-millis: 300000  
 filters: stat,wall,slf4j  
 *# 配置web监控,默认配置也和下面相同(除用户名密码，enabled默认false外)，其他可以不配* web-stat-filter:  
 enabled: true  
 url-pattern: /\*  
 exclusions: "\*.js,\*.gif,\*.jpg,\*.png,\*.css,\*.ico,/druid/\*"  
 stat-view-servlet:  
 enabled: true  
 url-pattern: /druid/\*  
 login-username: admin  
 login-password: root  
 allow: 127.0.0.1

后台页面，访问http://localhost:8080/druid/login.html

# 整合Mybatis

新建项目：



## 项目构建

1. 新建数据库表

DROP TABLE IF EXISTS `department`;

CREATE TABLE `department` (

`id` int(11) NOT NULL AUTO\_INCREMENT,

`departmentName` varchar(255) DEFAULT NULL,

PRIMARY KEY (`id`)

) ENGINE=InnoDB AUTO\_INCREMENT=1 DEFAULT CHARSET=utf8;

1. 创建对应实体类

package com.example.springbootmybatis.bean;  
  
import java.io.Serializable;  
  
public class Department implements Serializable {  
 Integer id;  
 String departmentName;  
  
 public Integer getId() {  
 return id;  
 }  
  
 public void setId(Integer id) {  
 this.id = id;  
 }  
  
 public String getDepartmentName() {  
 return departmentName;  
 }  
  
 public void setDepartmentName(String departmentName) {  
 this.departmentName = departmentName;  
 }  
}

1. 如上面一般配置Druid数据源

## Mybatis增删改查

1. 创建mapper接口

package com.example.springbootmybatis.mapper;  
  
import com.example.springbootmybatis.bean.Department;  
import org.apache.ibatis.annotations.\*;  
  
import java.util.List;  
  
@Mapper  
public interface DepartmentMapper {  
  
 @Select("select \* from department")  
 public List<Department> selectAll();  
  
 @Select("select \* from department where id=#{id}")  
 public Department selectById(Integer id);  
  
 @Options(useGeneratedKeys = true, keyProperty = "id")  
 @Insert("insert into department(departmentName) values(#{departmentName})")  
 public int save(Department department);  
  
 @Update("update department set departmentName=#{departmentName}")  
 public int update(Department department);  
  
 @Delete("delete from department where id =#{id}")  
 public int delete(Integer id);  
}

1. 创建Controller

package com.example.springbootmybatis.controller;  
  
import com.example.springbootmybatis.bean.Department;  
import com.example.springbootmybatis.mapper.DepartmentMapper;  
import org.springframework.beans.factory.annotation.Autowired;  
import org.springframework.web.bind.annotation.RequestMapping;  
import org.springframework.web.bind.annotation.RestController;  
  
import java.util.List;  
  
@RestController  
public class DepartmentController {  
  
 @Autowired  
 private DepartmentMapper departmentMapper;  
  
 @RequestMapping("getAll")  
 public List<Department> getAll(){  
 List<Department> list = departmentMapper.selectAll();  
 return list;  
 }  
  
 @RequestMapping("save")  
 public void save(Department department){  
 departmentMapper.save(department);  
 System.*out*.println(department);  
 }  
}

访问http://localhost:8080/save?departmentName=PeppaPig添加数据

访问http://localhost:8080/getAll 获取数据

## Mybatis配置

### 开启驼峰命名法

我们的实体类和表中的列名一致，一点问题也没有。我们把department表的departmentName列名改为department\_name看看会发生什么。

访问：[http://localhost:8080/getAll 获取数据](http://localhost:8080/dep/1获取数据)

[{"id":1,"departmentName":null}]

由于列表和属性名不一致，所以就没有封装进去，我们表中的列名和实体类属性名都是遵循驼峰命名规则的，可以开启mybatis的开启驼峰命名配置

mybatis:  
 configuration:  
 map-underscore-to-camel-case: true

### Mapper扫描

使用@mapper注解的类可以被扫描到容器中，但是每个Mapper都要加上这个注解就是一个繁琐的工作，能不能直接扫描某个包下的所有Mapper接口呢，当然可以，在springboot启动类上加上@MapperScan

package com.example.springbootmybatis;  
  
import org.mybatis.spring.annotation.MapperScan;  
import org.springframework.boot.SpringApplication;  
import org.springframework.boot.autoconfigure.SpringBootApplication;  
  
@MapperScan("com.example.springbootmybatis.mapper")  
@SpringBootApplication  
public class SpringbootMybatisApplication {  
  
 public static void main(String[] args) {  
 SpringApplication.*run*(SpringbootMybatisApplication.class, args);  
 }  
  
}

## 使用xml配置文件

1. 创建mybatis全局配置文件 \resources\mybatis\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>

<!-- 驼峰命名开启 -->  
 <setting name="mapUnderscoreToCamelCase" value="true"/>  
 </settings>  
</configuration>

1. 创建DepartmentMapper接口

package com.example.springbootmybatis.mapper;  
  
import com.example.springbootmybatis.bean.Department;  
import org.apache.ibatis.annotations.\*;  
  
import java.util.List;  
  
public interface DepartmentMapper {  
  
 public List<Department> selectAll();  
  
}

1. 创建DepartmentMapper.xml映射文件

\resources\mybatis\mapper\DepartmentMapper.xml

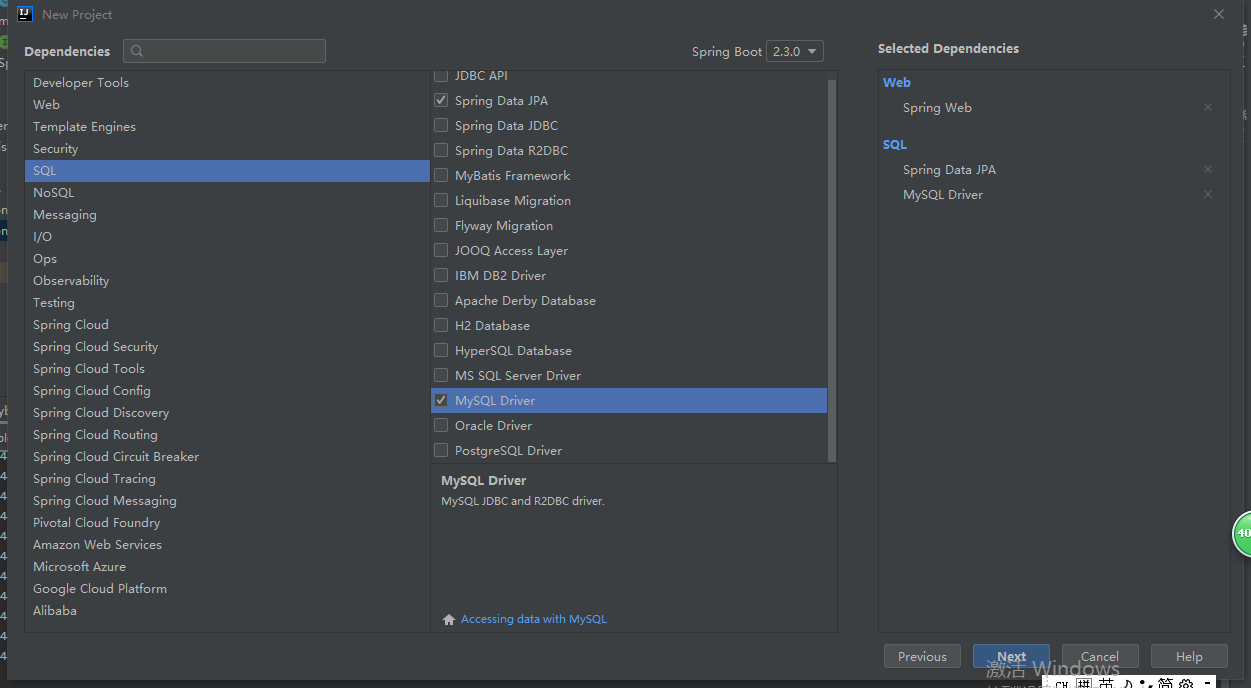
<?xml version="1.0" encoding="UTF-8"?>  
<!DOCTYPE mapper PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN" "http://mybatis.org/dtd/mybatis-3-mapper.dtd">  
<mapper namespace="com.example.springbootmybatis.mapper.DepartmentMapper">  
 <select id="selectAll" resultType="com.example.springbootmybatis.bean.Department">  
 SELECT \* FROM department  
 </select>  
</mapper>

1. 配置文件(application.yaml)中指定配置文件和映射文件的位置

mybatis:  
 config-location: classpath:mybatis/mybatis-config.xml  
 mapper-locations: classpath:mybatis/mapper/\*.xml

# SpringBoot整合JPA

创建项目



1. 实体类

package com.example.springbootjap.entity;  
  
import javax.persistence.\*;  
  
@Entity //使用JPA注解配置映射关系,告诉JPA这是一个实体类（和数据表映射的类）  
@Table(name="tb1\_user") //@Table来指定和哪个数据表对应;如果省略默认表名类名小写；  
public class User {  
  
 @Id //这是一个主键  
 @GeneratedValue(strategy = GenerationType.*IDENTITY*) //自增主键  
 private Integer id;  
  
 @Column(name = "last\_name",length = 50) //这是和数据表对应的一个列  
 private String lastName;  
 @Column //省略默认列名就是属性名  
 private String email;  
  
 public Integer getId() {  
 return id;  
 }  
  
 public void setId(Integer id) {  
 this.id = id;  
 }  
  
 public String getLastName() {  
 return lastName;  
 }  
  
 public void setLastName(String lastName) {  
 this.lastName = lastName;  
 }  
  
 public String getEmail() {  
 return email;  
 }  
  
 public void setEmail(String email) {  
 this.email = email;  
 }  
}

1. DAO

package com.example.springbootjap.repository;  
  
import com.example.springbootjap.entity.User;  
import org.springframework.data.jpa.repository.JpaRepository;  
  
*/\*\*  
 \* 继承JpaRepository来完成对数据库的操作  
 \* 泛型是（实体类，主键）  
 \*/*public interface UserRepository extends JpaRepository<User,Integer> {  
}

1. 配置文件

spring:  
 datasource:  
 username: root  
 password:  
 url: jdbc:mysql://127.0.0.1:3306/springboot?serverTimezone=UTC  
 driver-class-name: com.mysql.cj.jdbc.Driver  
 jpa:  
 hibernate:  
 *#更新或创建表结构* ddl-auto: update  
 *#在控制台打印sql* show-sql: true

1. 控制器

package com.example.springbootjap.controller;  
  
import com.example.springbootjap.entity.User;  
import com.example.springbootjap.repository.UserRepository;  
import org.springframework.beans.factory.annotation.Autowired;  
import org.springframework.web.bind.annotation.GetMapping;  
import org.springframework.web.bind.annotation.RestController;  
  
@RestController  
public class UserController {  
  
 @Autowired  
 private UserRepository userRepository;  
  
 @GetMapping("getUser")  
 public User getUser(Integer id){  
 User user = userRepository.getOne(id);  
 return user;  
 }  
  
 @GetMapping("save")  
 public User save(User user){  
 User userSave = userRepository.save(user);  
 return userSave;  
 }  
}

项目启动后会在springboot数据库创建表tb1\_user

访问：

<http://localhost:8080/save?lastName=zhangsan&email=123456@qq.com>

添加数据

访问：<http://localhost:8080/getUser?id=1>

会发现抛出500错误，原因是getOne方法使用的懒加载，获取到的只是代理对象，转换为json时会报错。

解决方法有两种：

1.关闭懒加载，在实体类上加@Proxy(lazy = false)注解

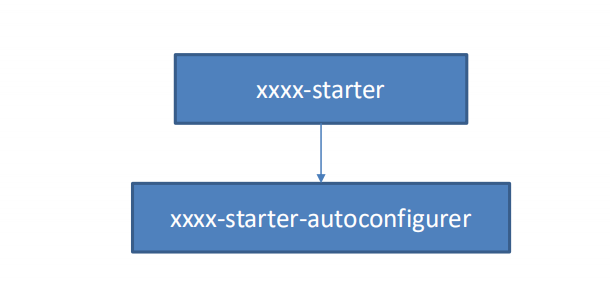
@Entity //使用JPA注解配置映射关系,告诉JPA这是一个实体类（和数据表映射的类）  
@Table(name="tb1\_user") //@Table来指定和哪个数据表对应;如果省略默认表名类名小写；  
@Proxy(lazy = false)  
public class User{

2.转json的时候忽略hibernateLazyInitializer和handler属性

@Entity //使用JPA注解配置映射关系,告诉JPA这是一个实体类（和数据表映射的类）  
@Table(name="tb1\_user") //@Table来指定和哪个数据表对应;如果省略默认表名类名小写；  
@JsonIgnoreProperties(value = {"hibernateLazyInitializer", "handler"})  
public class User{

# 自定义Starter

* 启动器只用来做依赖导入
* 专门来写一个自动配置模块；
* 启动器依赖自动配置模块，项目中引入相应的starter就会引入启动器的所有传递依赖



## 启动器

启动器模块是一个空 JAR 文件，仅提供辅助性依赖管理，这些依赖可能用于自动 装配或者其他类库

## 命名规约

**官方命名**

spring-boot-starter-模块名

eg：spring-boot-starter-web、spring-boot-starter-jdbc、

spring-boot-starter-thymeleaf

**自定义命名**

模块名-spring-boot-starter

eg：mybatis-spring-boot-start

## 如何编写自动配置

@Configuration //指定这个类是一个配置类

@ConditionalOnXXX //在指定条件成立的情况下自动配置类生效

@AutoConfigureAfter //指定自动配置类的顺序

@Bean //给容器中添加组件

@ConfigurationPropertie结合相关xxxProperties类来绑定相关的配置

@EnableConfigurationProperties //让xxxProperties生效加入到容器中

public class XxxxAutoConfiguration {

自动配置类要能加载,需要将启动就加载的自动配置类配置在META-INF/spring.factories中

**eg:**

# Auto Configure

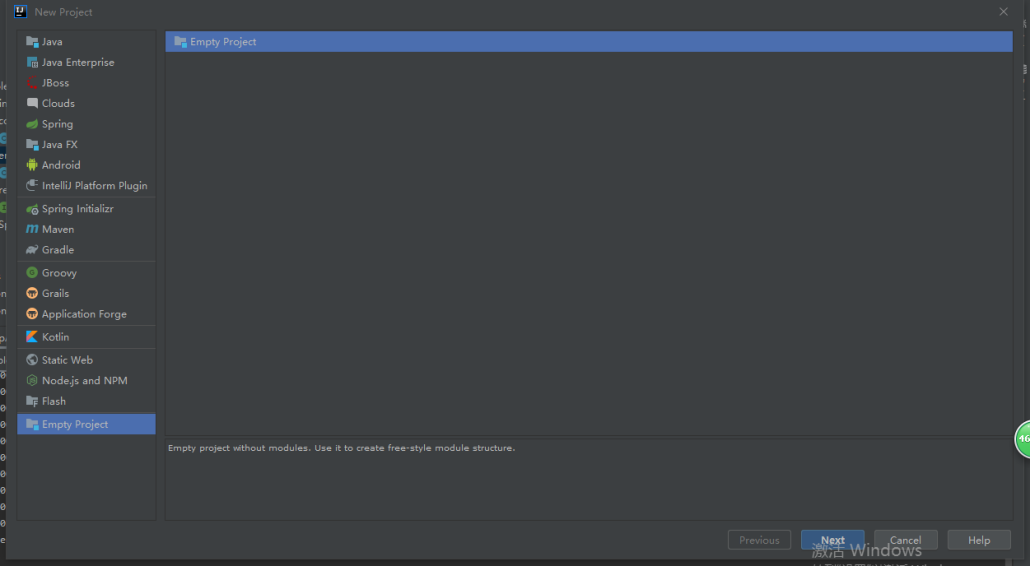
org.springframework.boot.autoconfigure.EnableAutoConfiguration=\

org.mybatis.spring.boot.autoconfigure.MybatisLanguageDriverAutoConfiguration,\

org.mybatis.spring.boot.autoconfigure.MybatisAutoConfiguration

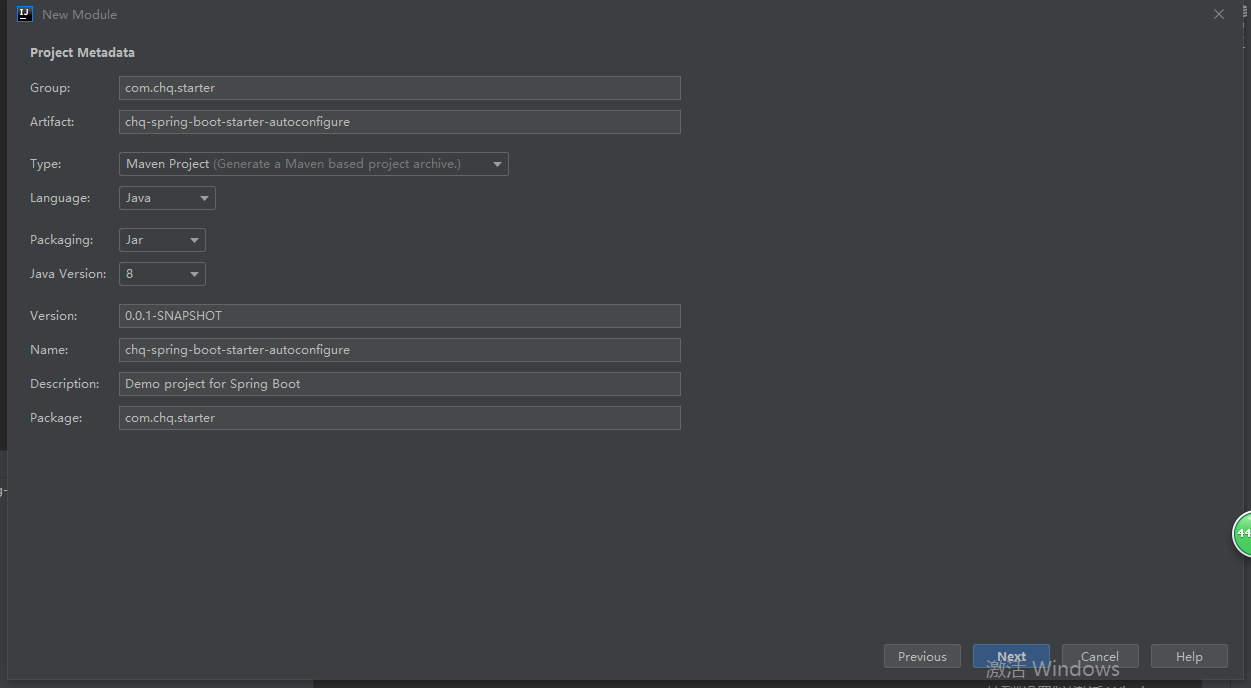
## 案例

1. 创建空项目

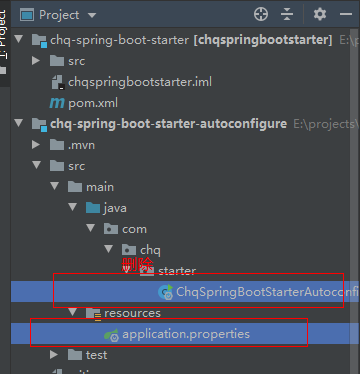


命名：spring-starter

2.创建一个自动配置模块，和创建普通springboot项目一样，不需要引入其他starter



1. 删除多余的文件和依赖（还有test目录）



依赖：

<?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 https://maven.apache.org/xsd/maven-4.0.0.xsd">  
 <modelVersion>4.0.0</modelVersion>  
 <parent>  
 <groupId>org.springframework.boot</groupId>  
 <artifactId>spring-boot-starter-parent</artifactId>  
 <version>2.3.0.RELEASE</version>  
 <relativePath/> <!-- lookup parent from repository -->  
 </parent>  
 <groupId>com.chq.starter</groupId>  
 <artifactId>chq-spring-boot-starter-autoconfigure</artifactId>  
 <version>0.0.1-SNAPSHOT</version>  
 <name>chq-spring-boot-starter-autoconfigure</name>  
 <description>Demo project for Spring Boot</description>  
  
 <properties>  
 <java.version>1.8</java.version>  
 </properties>  
  
 <dependencies>  
 <!--引入spring‐boot‐starter；所有starter的基本配置-->  
 <dependency>  
 <groupId>org.springframework.boot</groupId>  
 <artifactId>spring-boot-starter</artifactId>  
 </dependency>  
 </dependencies>  
  
</project>

1. 创建配置类和service类

package com.chq.starter;  
  
import org.springframework.boot.context.properties.ConfigurationProperties;  
  
@ConfigurationProperties(prefix = "chq")  
public class HelloProperties {  
 private String prefix;  
 private String suffix;  
  
 public ClboyProperties() {  
 this.prefix = "";  
 this.suffix = "";  
 }  
  
 public String getPrefix() {  
 return prefix;  
 }  
  
 public void setPrefix(String prefix) {  
 this.prefix = prefix;  
 }  
  
 public String getSuffix() {  
 return suffix;  
 }  
  
 public void setSuffix(String suffix) {  
 this.suffix = suffix;  
 }  
}

package com.chq.starter;  
  
  
public class HelloService {  
 HelloProperties helloProperties;  
  
 public HelloProperties getHelloProperties() {  
 return helloProperties;  
 }  
  
 public void setHelloProperties(HelloProperties helloProperties) {  
 this.helloProperties = helloProperties;  
 }  
  
 public String sayHello(String name) {  
 return helloProperties.getPrefix() + name + helloProperties.getSuffix();  
 }  
}

1. 创建自动配置类

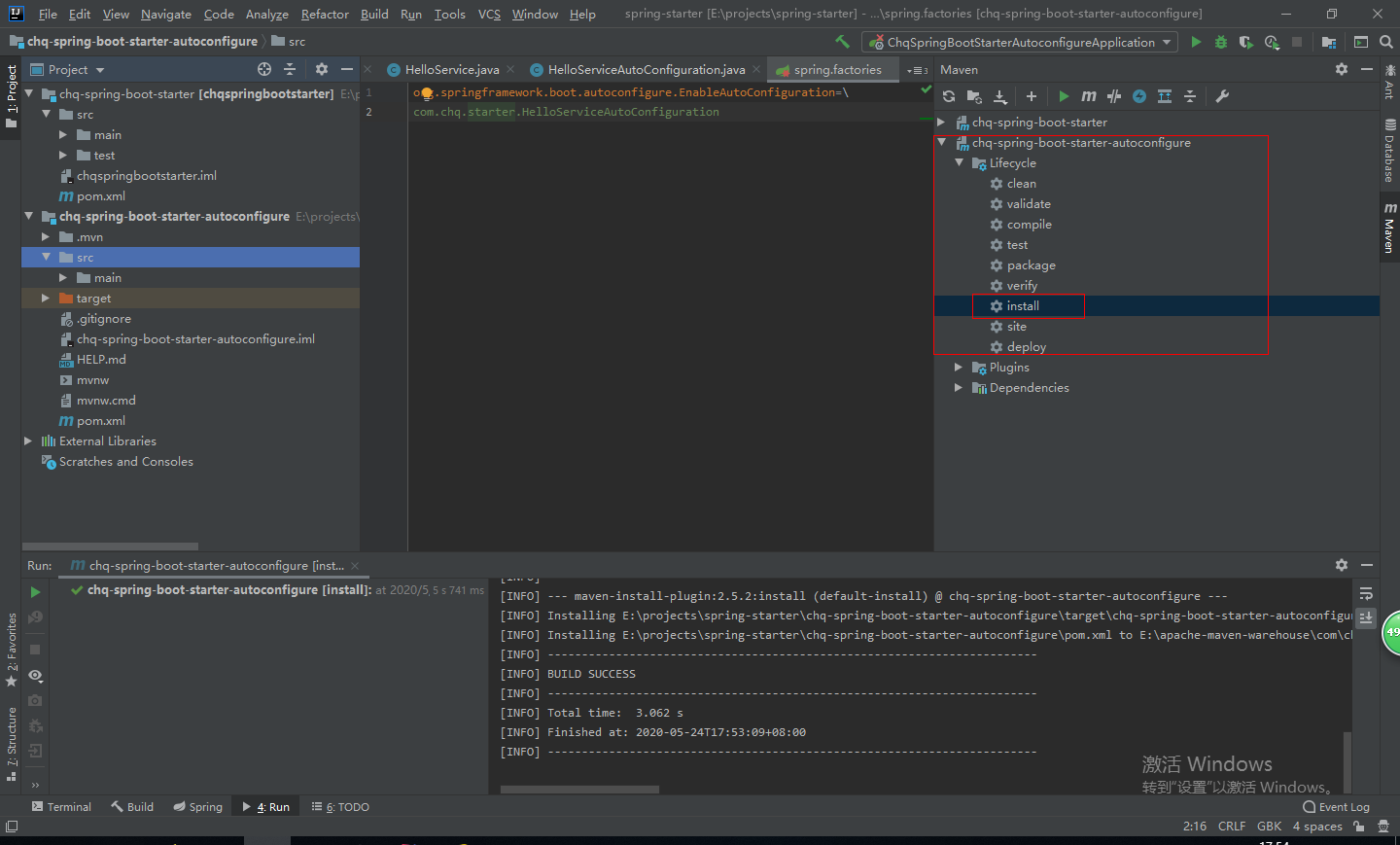
package com.chq.starter;  
  
import org.springframework.beans.factory.annotation.Autowired;  
import org.springframework.boot.autoconfigure.condition.ConditionalOnWebApplication;  
import org.springframework.boot.context.properties.EnableConfigurationProperties;  
import org.springframework.context.annotation.Configuration;  
  
@Configuration  
@ConditionalOnWebApplication //web应用才生效  
@EnableConfigurationProperties(HelloProperties.class) //让配置类生效，(注入到容器中)  
public class HelloServiceAutoConfiguration {  
  
 @Autowired  
 private HelloProperties helloProperties;

@Bean  
 public HelloService helloService(){  
 HelloService helloService = new HelloService();  
 helloService.setHelloProperties(helloProperties);  
 return helloService;  
 }  
}

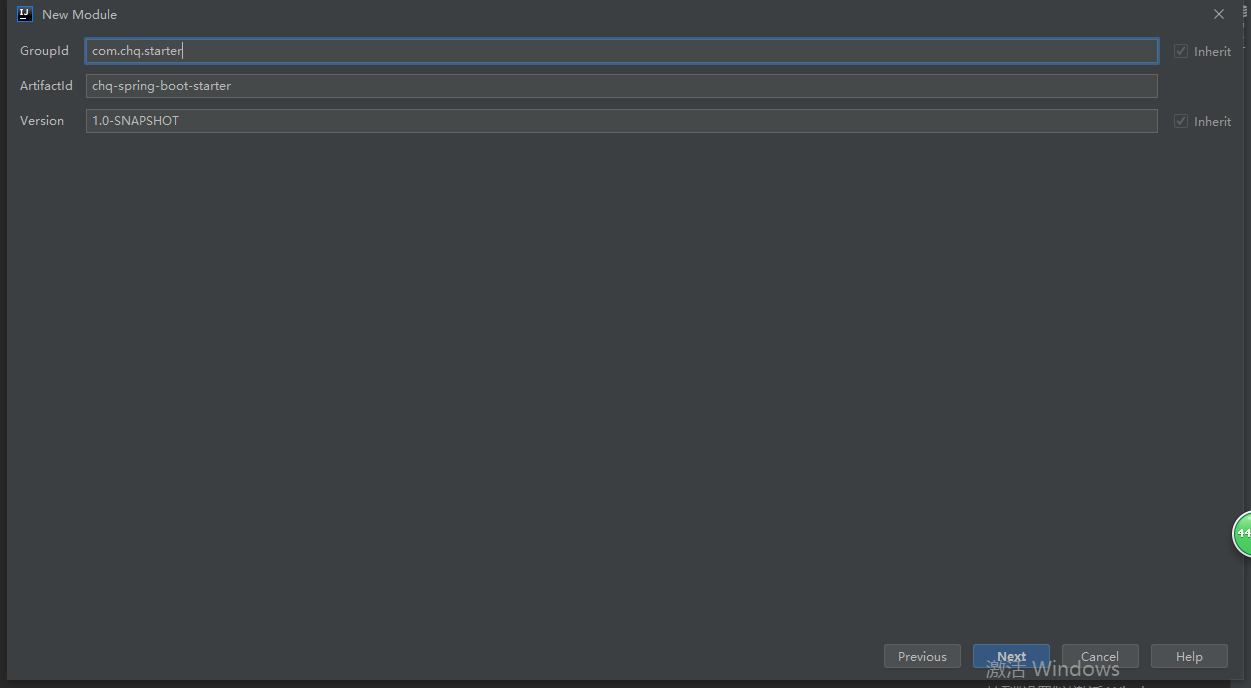
1. 在resources文件夹下创建META-INF/spring.factories

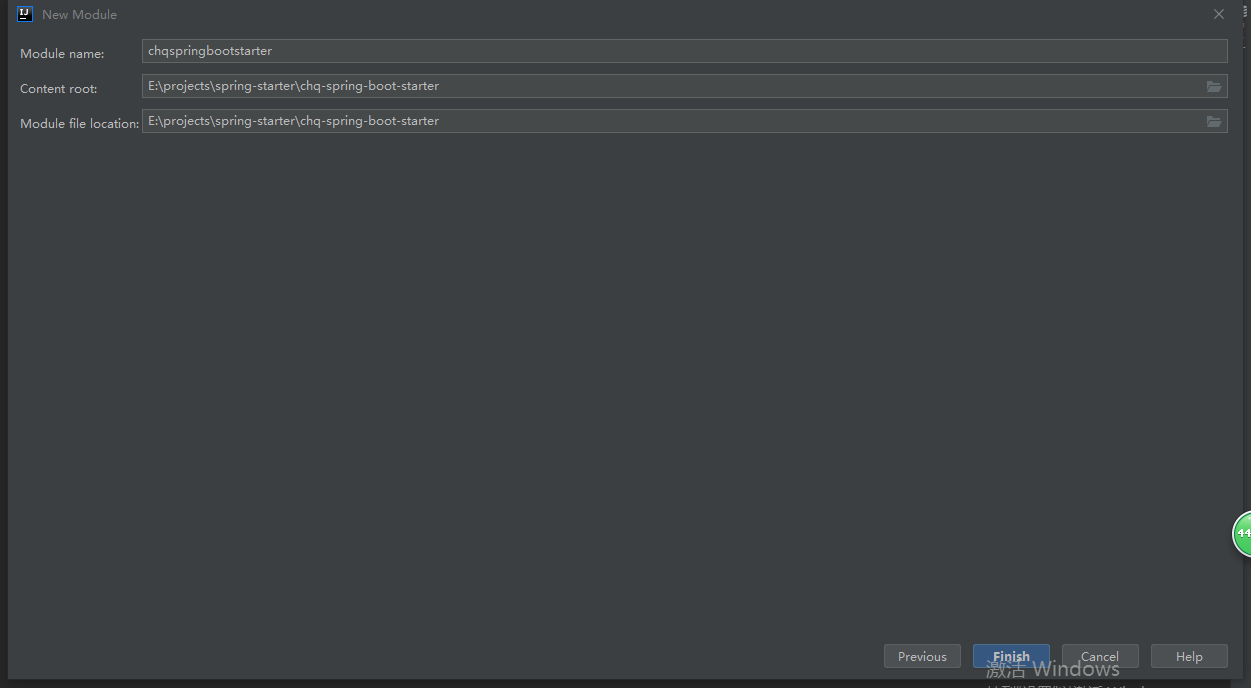
org.springframework.boot.autoconfigure.EnableAutoConfiguration=\  
com.chq.starter.HelloServiceAutoConfiguration

6.安装到本地仓库



7.创建starter，选择maven工程即可，只是用于管理依赖，添加对AutoConfiguration模块的依赖，命名为com.chq.starter

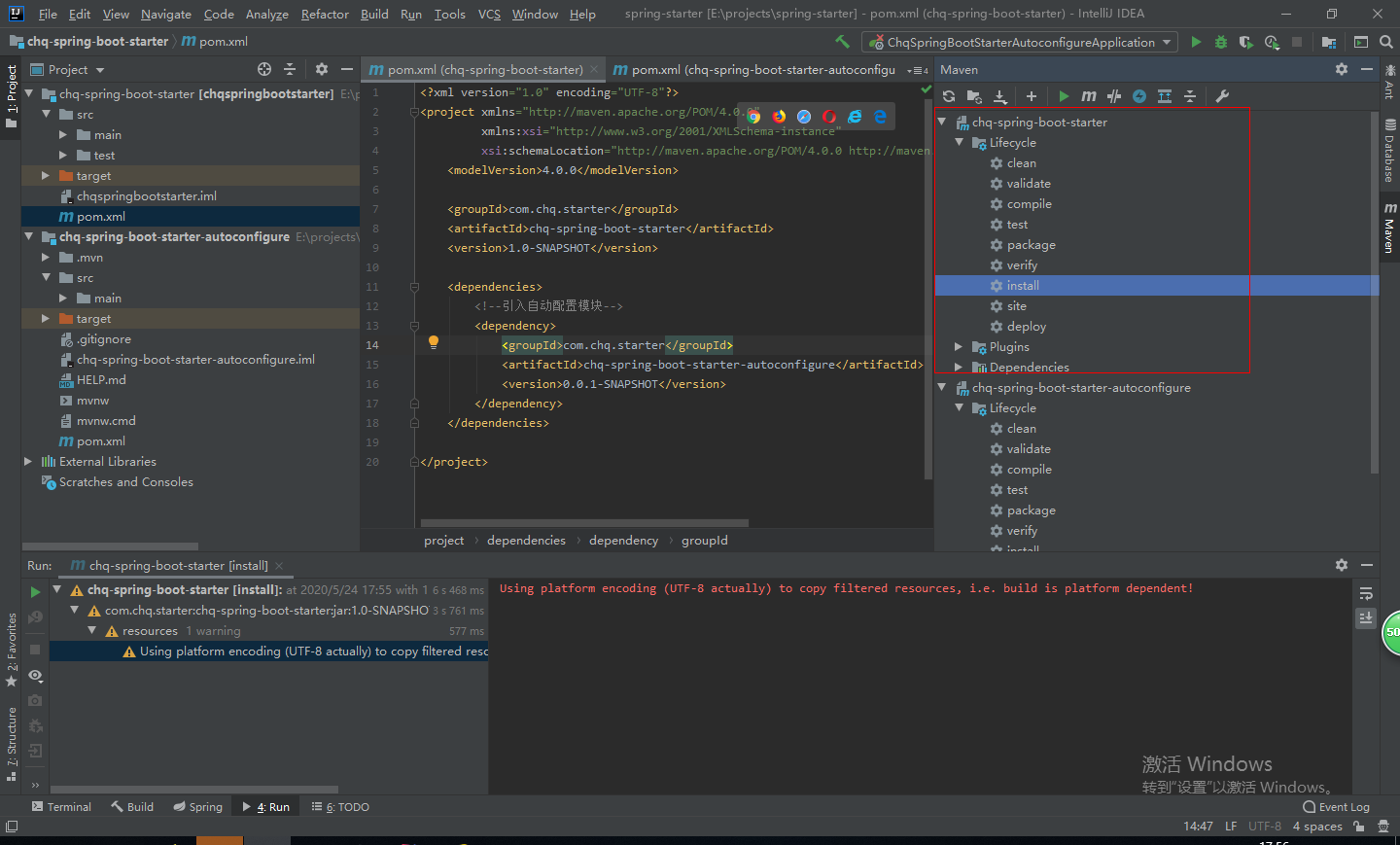




8.在com.chq.starter模块添加对AutoConfiguration模块的依赖

<dependencies>  
 <!--引入自动配置模块-->  
 <dependency>  
 <groupId>com.chq.starter</groupId>  
 <artifactId>chq-spring-boot-starter-autoconfigure</artifactId>  
 <version>0.0.1-SNAPSHOT</version>  
 </dependency>  
</dependencies>

1. 安装到本地仓库



1. 创建项目测试，选择添加web场景，因为设置是web场景才生效

<!--引入自定义的starter-->  
<dependency>  
 <groupId>com.chq.starter</groupId>  
 <artifactId>chq-spring-boot-starter</artifactId>  
 <version>1.0-SNAPSHOT</version>  
</dependency>

1. 创建控制器

package com.chq.springbootstartertest.controller;  
  
import com.chq.starter.HelloService;  
import org.springframework.beans.factory.annotation.Autowired;  
import org.springframework.web.bind.annotation.GetMapping;  
import org.springframework.web.bind.annotation.RestController;  
  
@RestController  
public class TestController {  
  
 @Autowired  
 private HelloService helloService;  
  
 @GetMapping("hello")  
 public String sayHello(){  
 return helloService.sayHello("chq");  
 }  
}

1. 在配置文件中配置

chq.prefix=hi  
chq.suffix=good

访问：<http://localhost:8080/hello>

# Spring Boot与缓存

## JSR107

* CachingProvider

定义了创建、配置、获取、管理和控制多个CacheManager。一个应用可

以在运行期访问多个CachingProvider。

* CacheManager

定义了创建、配置、获取、管理和控制多个唯一命名的Cache，这些Cache

存在于CacheManager的上下文中。一个CacheManager仅被一个CachingProvider所拥有。

* Cache

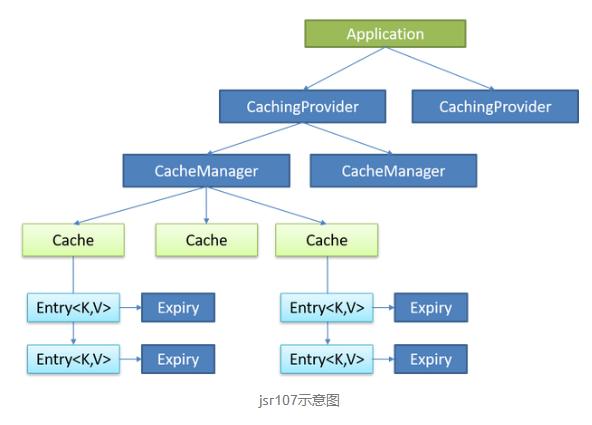
一个类似Map的数据结构并临时存储以Key为索引的值。一个Cache仅被一个CacheManager所拥有。

* Entry

一个存储在Cache中的key-value对。

* Expiry

每一个存储在Cache中的条目有一个定义的有效期。一旦超过这个时间，条目为过期的状态。一旦过期，条目将不可访问、更新和删除。缓存有效期可以通过ExpiryPolicy设置。



## 缓存抽象

Spring从3.1开始定义了org.springframework.cache.Cache

和org.springframework.cache.CacheManager接口来统一不同的缓存技术；

并支持使用JCache（JSR-107）注解简化我们开发；

Cache接口有以下功能：

为缓存的组件规范定义，包含缓存的各种操作集合；

Spring提供了各种xxxCache的实现；如RedisCache，EhCacheCache ,

ConcurrentMapCache等；

## 重要缓存注解及概念

|  |  |
| --- | --- |
| Cache | 缓存接口，定义缓存操作。实现有：RedisCache、EhCacheCache、ConcurrentMapCache等 |
| CacheManager | 缓存管理器，管理各种缓存（Cache）组件 |
| @Cacheable | 根据方法的请求参数对其结果进行缓存。先查缓存，若没有再执行方法。 |
| @CacheEvict | 清空缓存。执行方法，并清空对应key缓存 |
| @CachePut | 更新缓存。执行方法，并将结果更新到缓存 |
| @EnableCaching | 开启基于注解的缓存 |
| keyGenerator | 缓存数据时key生成策略 |
| serialize | 缓存数据时value序列化策略 |

@Cacheable/@CachePut/@CacheEvict 主要的参数

* value

缓存名称，字符串/字符数组形式；

如@Cacheable(value=”mycache”) 或者@Cacheable(value={”cache1”,”cache2”}

* key

缓存的key,需要按照SpEL表达式编写，如果不指定则按照方法所有参数进行组合；

如@Cacheable(value=”testcache”,key=”#userName”)

* keyGenerator

key的生成器；可以自己指定key的生成器的组件id

注意：key/keyGenerator：二选一使用;

* condition

缓存条件，使用SpEL编写，在调用方法之前之后都能判断；

如@Cacheable(value=”testcache”,condition=”#userName.length()>2”)

* unless（@CachePut、@Cacheable）

用于否决缓存的条件，只在方法执行之后判断；

如@Cacheable(value=”testcache”,unless=”#result ==null”)

* beforeInvocation（@CacheEvict）

是否在执行前清空缓存，默认为false，false情况下方法执行异常则不会清空；

如@CachEvict(value=”testcache”，beforeInvocation=true)

* allEntries（@CacheEvict）

是否清空所有缓存内容，默认为false；

如@CachEvict(value=”testcache”,allEntries=true)

缓存可用的SpEL表达式

**root**

表示根对象，不可省略

* 被调用方法名 methodName

如 #root.methodName

* 被调用方法 method

如 #root.method.name

* 目标对象 target

如 #root.target

* 被调用的目标对象类 targetClass

如 #root.targetClass

* 被调用的方法的参数列表 args

如 #root.args[0]

* 方法调用使用的缓存列表 caches

如 #root.caches[0].name

**参数名**

方法参数的名字. 可以直接 #参数名 ，也可以使用 #p0或#a0 的形式，0代表参数的索引；

如 #iban 、 #a0 、 #p0

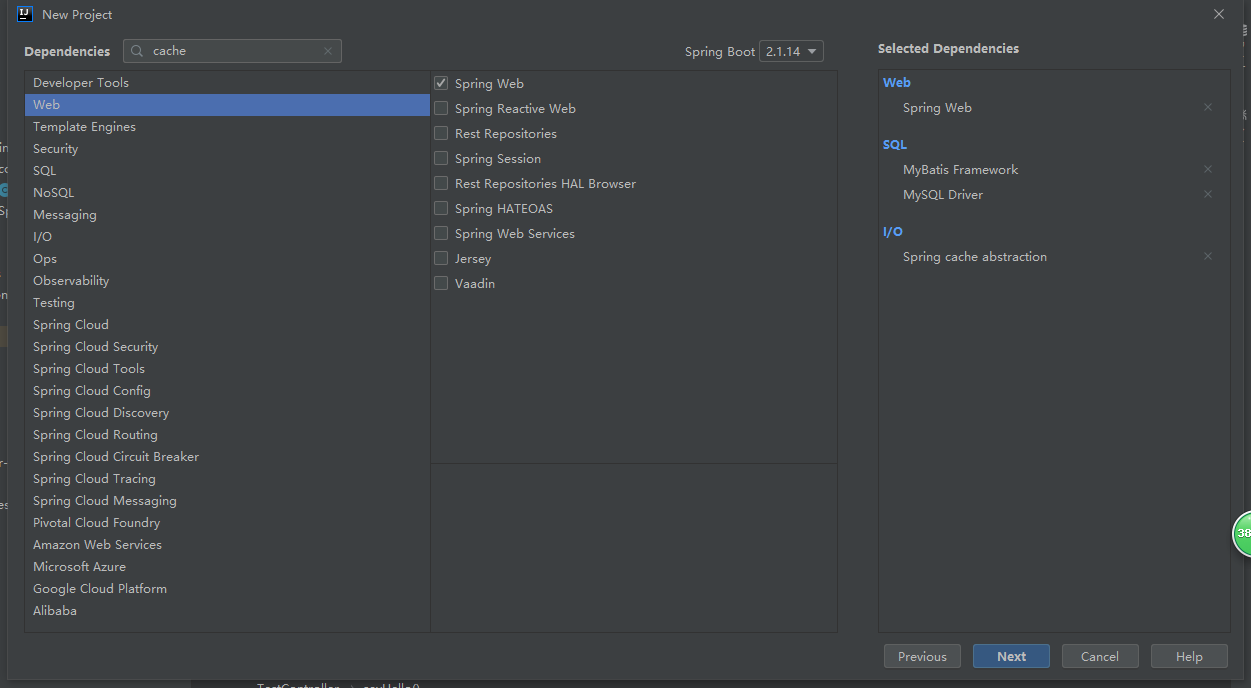
**返回值**

方法执行后的返回值（仅当方法执行之后的判断有效，如‘unless’ ， @CachePut、@CacheEvict’的表达式beforeInvocation=false ）

如 #result

## 缓存使用

### 创建应用



会引入spring-boot-starter-cache模块

<dependency>  
 <groupId>org.springframework.boot</groupId>  
 <artifactId>spring-boot-starter-cache</artifactId>  
</dependency>

### 准备测试环境

1. 新建application.yml配置数据源

spring:  
 datasource:  
 username: root  
 password:  
 url: jdbc:mysql://127.0.0.1:3306/springboot?serverTimezone=UTC

1. 创建数据库表

DROP TABLE IF EXISTS `department`;  
CREATE TABLE `department` (  
 `id` int(11) NOT NULL AUTO\_INCREMENT,  
 `departmentName` varchar(255) DEFAULT NULL,  
 PRIMARY KEY (`id`)  
) ENGINE=InnoDB DEFAULT CHARSET=utf8;

1. 新建bean实体类

package com.chq.springbootcache.bean;  
  
public class Department {  
 Integer id;  
 String departmentName;  
  
 public Integer getId() {  
 return id;  
 }  
  
 public void setId(Integer id) {  
 this.id = id;  
 }  
  
 public String getDepartmentName() {  
 return departmentName;  
 }  
  
 public void setDepartmentName(String departmentName) {  
 this.departmentName = departmentName;  
 }  
}

4）创建mapper接口

package com.chq.springbootcache.mapper;  
  
import com.chq.springbootcache.bean.Department;  
import org.apache.ibatis.annotations.\*;  
import org.springframework.stereotype.Repository;  
  
@Repository  
@Mapper  
public interface DepartmentMapper {  
  
 @Select("select \* from department where id=#{id}")  
 public Department selectById(Integer id);  
  
 @Options(useGeneratedKeys = true, keyProperty = "id")  
 @Insert("insert into department(departmentName) values(#{departmentName})")  
 public int save(Department department);  
  
 @Update("update department set departmentName=#{departmentName} where id=#{id}")  
 public int update(Department department);  
}

5）创建service

package com.chq.springbootcache.service;  
  
 import com.chq.springbootcache.bean.Department;  
 import com.chq.springbootcache.mapper.DepartmentMapper;  
 import org.springframework.beans.factory.annotation.Autowired;  
 import org.springframework.cache.annotation.CacheEvict;  
 import org.springframework.cache.annotation.CachePut;  
 import org.springframework.cache.annotation.Cacheable;  
 import org.springframework.stereotype.Service;  
  
@Service  
public class DepartmentService {  
  
 @Autowired  
 private DepartmentMapper departmentMapper;  
  
 public Department getById(Integer id){  
 System.*out*.println("查询"+id);  
 return departmentMapper.selectById(id);  
 }  
  
 public void save(Department department){  
 departmentMapper.save(department);  
 }  
  
 public Department update(Department department){  
 departmentMapper.update(department);  
 return department;  
 }  
  
 public void delete(Integer id){  
 System.*out*.println("删除"+id);  
 }  
}

1. 创建controller

package com.chq.springbootcache.controller;  
  
import com.chq.springbootcache.bean.Department;  
import com.chq.springbootcache.service.DepartmentService;  
import org.springframework.beans.factory.annotation.Autowired;  
import org.springframework.cache.annotation.Cacheable;  
import org.springframework.web.bind.annotation.RequestMapping;  
import org.springframework.web.bind.annotation.RestController;  
  
@RestController  
public class TestController {  
 @Autowired  
 private DepartmentService departmentService;  
  
 @RequestMapping("get")  
 public Department getById(Integer id){  
 return departmentService.getById(id);  
 }  
  
 @RequestMapping("save")  
 public void save(Department department){  
 departmentService.save(department);  
 System.*out*.println(department);  
 }  
  
 @RequestMapping("update")  
 public Department update(Department department){  
 departmentService.update(department);  
 return department;  
 }  
  
 @RequestMapping("delete")  
 public void delete(Integer id){  
 departmentService.delete(id);  
 }  
}

访问http://localhost:8080/save?departmentName=PeppaPig添加数据

访问http://localhost:8080/get?id=1 获取数据

### 3快速体验缓存

在主配置启动类开启缓存

@EnableCaching

#### @Cacheable

Service方法上添加@Cacheable

@Cacheable(value = "depart")  
public Department getById(Integer id){  
 System.*out*.println("查询"+id);  
 return departmentMapper.selectById(id);  
}

第一次访问http://localhost:8080/get?id=1，将方法返回的数据缓存在value(depart)对应的key(1)中。

第二次后访问直接从缓存depart里取key为1的数据。

#### @CachePut

Service方法上添加@CachePut

@CachePut(value = "depart",key = "#department.id")  
public Department update(Department department){  
 departmentMapper.update(department);  
 return department;  
}

1访问http://localhost:8080/get?id=1

2访问http://localhost:8080/update?id=1&departmentName=wwww更新数据

3再次访问http://localhost:8080/get?id=1 数据是更新后的数据

#### @CacheEvict

Service方法上添加@CacheEvict

@CacheEvict(value = "depart")  
public void delete(Integer id){

#### @CacheConfig

标注在类上，用于抽取@Cacheable的公共属性

#### @Caching

组合使用@Cacheable、@CachePut、@CacheEvict

**自定义KeyGenerator生成key**

使用时在注解属性内指定KeyGenerator=“myKeyGenerator”

package com.chq.springbootcache.config;  
  
import org.springframework.cache.interceptor.KeyGenerator;  
import org.springframework.context.annotation.Bean;  
import org.springframework.context.annotation.Configuration;  
  
import java.lang.reflect.Method;  
import java.util.Arrays;  
  
@Configuration  
public class MyCacheConfig {  
 @Bean("myKeyGenerator")  
 public KeyGenerator myKeyGenerator() {  
 return new KeyGenerator(){  
 @Override  
 public Object generate(Object target, Method method, Object... params) {  
 return method.getName()+"["+ Arrays.*asList*(params).toString()+target+"]";  
 }  
 };  
 }  
}

## 原理

看视频76节

# Redis与缓存

可以使用Redis Desktop Manager工具查看redis数据

## RedisTemplate

RedisAutoConfiguration向容器中导入了两个类

**RedisTemplate**<Object, Object> redisTemplate

**StringRedisTemplate**

* 两者的数据是不共通的；也就是说StringRedisTemplate只能管理StringRedisTemplate里面的数据，RedisTemplate只能管理RedisTemplate中的数据。
* 其实他们两者之间的区别主要在于他们使用的序列化类:
* RedisTemplate使用的是JdkSerializationRedisSerializer，存入数据会将数据先序列化成字节数组然后在存入Redis数据库。
* StringRedisTemplate使用的是StringRedisSerializer
* 存取的数据就是字符串类型数据的时候使用StringRedisTemplate
* 数据是复杂的对象类型使用RedisTemplate

**Redis常见的五大数据类型：**

String（字符串）、List（列表）、Set（集合）、Hash（散列）、ZSet（有序集合）

​ stringRedisTemplate.opsForValue()[String（字符串）]

​ stringRedisTemplate.opsForList()[List（列表）]

​ stringRedisTemplate.opsForSet()[Set（集合）]

​ stringRedisTemplate.opsForHash()[Hash（散列）]

​ stringRedisTemplate.opsForZSet()[ZSet（有序集合）]

## 环境搭建

导入依赖

<dependency>  
 <groupId>org.springframework.boot</groupId>  
 <artifactId>spring-boot-starter-data-redis</artifactId>  
</dependency>

在配置文件指定Redis服务器地址

spring:  
 redis:  
 host: 127.0.0.1

搭建好后在上一节Spring Boot与缓存中使用的缓存就是redis了。

## Redis缓存使用

@Autowired  
StringRedisTemplate stringRedisTemplate;  
  
@Test  
public void contextLoads() {  
 stringRedisTemplate.opsForValue().append("msg","hello");  
}

@Autowired  
RedisTemplate redisTemplate;  
@Autowired  
DepartmentService departmentService;  
  
@Test  
public void contextLoads() {  
 Department department = departmentService.getById(1);  
 redisTemplate.opsForValue().set("depart", department);  
}

要想让对象以json形式存储在redis中，需要自定义RedisCacheManager，使用GenericJackson2JsonRedisSerializer类对value进行序列化

package com.chq.springbootcache.config;  
  
import org.springframework.context.annotation.Bean;  
import org.springframework.context.annotation.Configuration;  
import org.springframework.data.redis.cache.RedisCacheConfiguration;  
import org.springframework.data.redis.cache.RedisCacheManager;  
import org.springframework.data.redis.cache.RedisCacheWriter;  
import org.springframework.data.redis.connection.RedisConnectionFactory;  
import org.springframework.data.redis.serializer.GenericJackson2JsonRedisSerializer;  
import org.springframework.data.redis.serializer.RedisSerializationContext;  
  
@Configuration  
public class MyRedisConfig {  
 @Bean  
 RedisCacheManager cacheManager(RedisConnectionFactory factory){  
 //创建默认RedisCacheWriter  
 RedisCacheWriter cacheWriter = RedisCacheWriter.*nonLockingRedisCacheWriter*(factory);  
  
 //创建默认RedisCacheConfiguration并使用GenericJackson2JsonRedisSerializer构造的 SerializationPair对value进行转换  
 //创建GenericJackson2JsonRedisSerializer的json序列化器  
 GenericJackson2JsonRedisSerializer jsonRedisSerializer = new GenericJackson2JsonRedisSerializer();  
 //使用json序列化器构造出对转换Object类型的SerializationPair序列化对  
 RedisSerializationContext.SerializationPair<Object> serializationPair = RedisSerializationContext.SerializationPair.*fromSerializer*(jsonRedisSerializer);  
 //将可以把Object转换为json的SerializationPair传入RedisCacheConfiguration  
 //使得RedisCacheConfiguration在转换value时使用定制序列化器  
 RedisCacheConfiguration cacheConfiguration=RedisCacheConfiguration.*defaultCacheConfig*().serializeValuesWith(serializationPair);  
  
 RedisCacheManager cacheManager = new RedisCacheManager(cacheWriter,cacheConfiguration);  
 return cacheManager;  
 }  
}

# Spring Boot与消息

## 消息简介

* JMS（Java Message Service）JAVA消息服务

基于JVM消息代理的规范。ActiveMQ、HornetMQ是JMS实现

* AMQP（Advanced Message Queuing Protocol）

高级消息队列协议，也是一个消息代理的规范，兼容JMS

RabbitMQ是AMQP的实现

**作用**

通过消息服务中间件来提升系统异步通信、扩展解耦能力

当消息发送者发送消息以后，将由消息代理接管，消息代理保证消息传递到指定目的地

## RabbitMQ

### 核心概念

* Message

消息，消息是不具名的，它由消息头和消息体组成

消息头，包括routing-key（路由键）、priority（相对于其他消息的优先权）、delivery-mode（指出该消息可能需要持久性存储）等

* Publisher

消息的生产者，也是一个向交换器发布消息的客户端应用程序

* Exchange

交换器，将生产者消息路由给服务器中的队列

类型有direct(默认)，fanout, topic, 和headers，具有不同转发策略

* Queue

消息队列，保存消息直到发送给消费者

* Binding

绑定，用于消息队列和交换器之间的关联

* Connection

网络连接，比如一个TCP连接

* Consumer

消息的消费者，表示一个从消息队列中取得消息的客户端应用程序

* Virtual Host

虚拟主机，表示一批交换器、消息队列和相关对象。

vhost 是 AMQP 概念的基础，必须在连接时指定

RabbitMQ 默认的 vhost 是 /

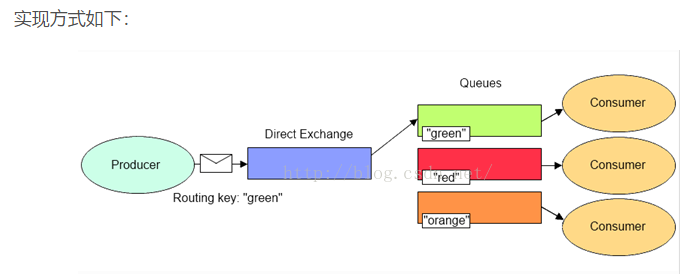
* Broker

消息队列服务器实体

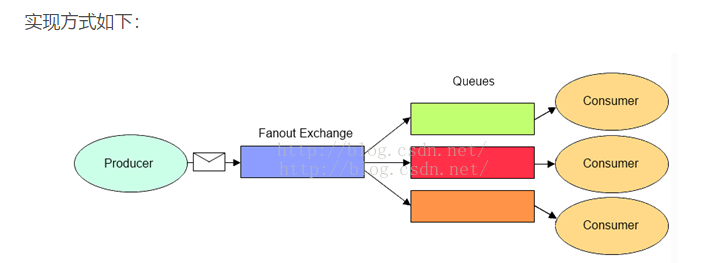
**Exchange 类型**

**direct**：点对点模式，消息中的路由键（routing key）如果和 Binding 中的 binding

key 一致， 交换器就将消息发到对应的队列中。



**fanout:**广播模式，每个发到 fanout 类型交换器的消息都会分到所有绑定的队列上去



**topic:**将路由键和某个模式进行匹配，此时队列需要绑定到一个模式上。它将路由键和绑定键的字符串切分成单词，这些单词之间用点隔开。

识别通配符： # 匹配 0 个或多个单词， \*匹配一个单词



### 安装

**环境：linux，安装方式:docker**

//下载rabbittmq镜像

docker pull rabbitmq:3.8.4-management

//查看下载的镜像

docker images

//启动镜像容器

docker run -d -p 5672:5672 -p 15672:15672 镜像id

//查看ip地址

ifconfig

//访问rabbitmq操作界面

ip地址:15672

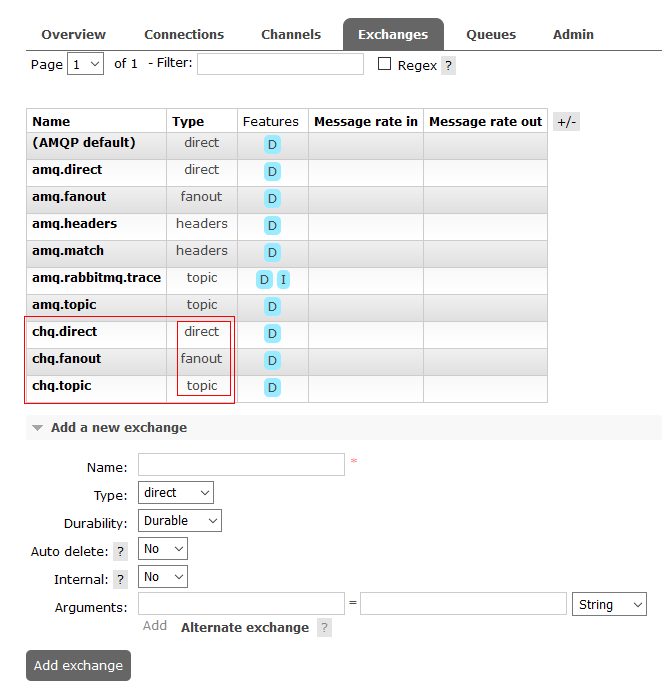
账号密码：guest

## Springboot中的RabbitMQ

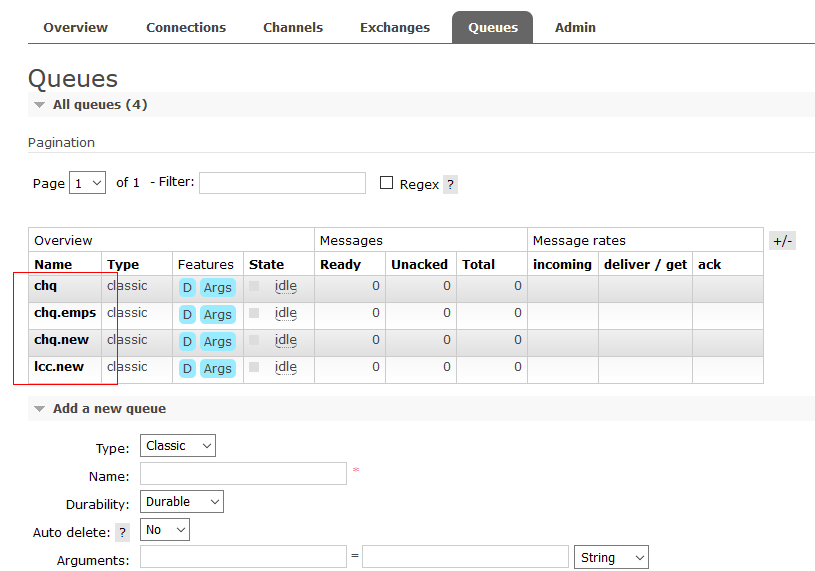
### 测试环境准备

1登录rabbitmq操作界面

2新建exchange交换器

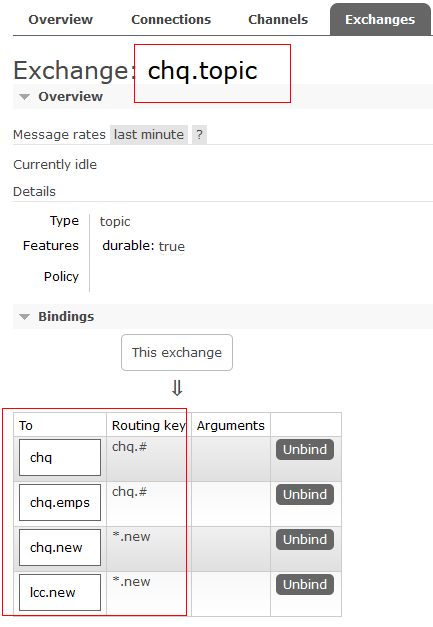
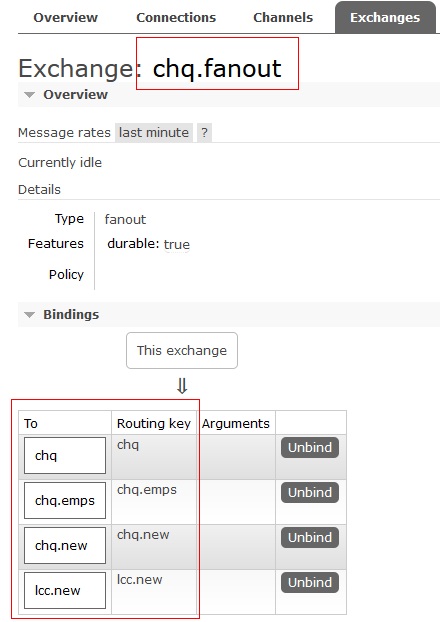
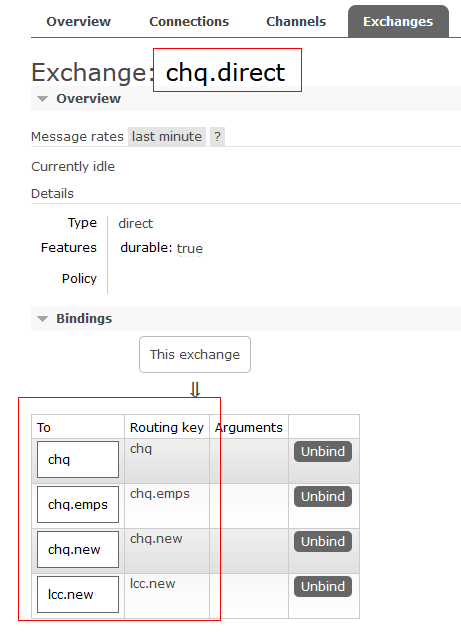


3新建Queue消息队列



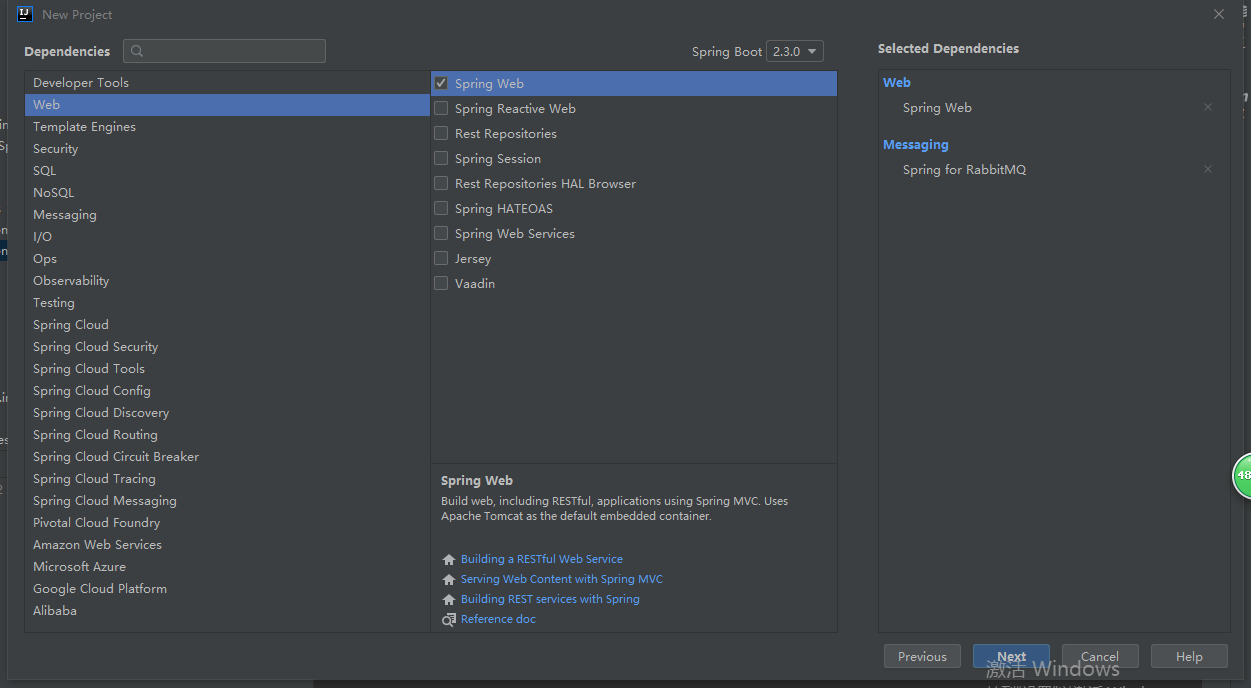
4绑定exchange和Queue的关系

点击exchange名可以进行操作



### 案例

1创建项目



会导入依赖：

<dependency>  
 <groupId>org.springframework.boot</groupId>  
 <artifactId>spring-boot-starter-amqp</artifactId>  
</dependency>

2配置连接参数

spring:  
 rabbitmq:  
 host: 192.168.153.130  
 username: guest  
 password: guest

3设置消息转换器

默认的消息转化器是SimpleMessageConverter，对于对象以jdk序列化方式存储，若要以Json方式存储对象，就要自定义消息转换器

package com.chq.config;  
  
import org.springframework.amqp.support.converter.Jackson2JsonMessageConverter;  
import org.springframework.amqp.support.converter.MessageConverter;  
import org.springframework.context.annotation.Bean;  
import org.springframework.context.annotation.Configuration;  
  
@Configuration  
public class AmqpConfig {  
 @Bean  
 public MessageConverter messageConverter() {  
 //在容器中导入Json的消息转换器  
 return new Jackson2JsonMessageConverter();  
 }  
}

3测试发送

package com.chq.bean;  
  
import java.io.Serializable;  
  
public class Book implements Serializable {  
 private String name;  
  
 public String getName() {  
 return name;  
 }  
  
 public void setName(String name) {  
 this.name = name;  
 }  
}

@Autowired  
private RabbitTemplate rabbitTemplate;  
  
@RequestMapping("send")  
public void send(Book book){  
 rabbitTemplate.convertAndSend("chq.direct","chq",book);  
}

### 消息的监听

1. 在启动类添加注解@EnableRabbit

@SpringBootApplication  
@EnableRabbit  
public class ManageAmqpApplication {

2.在回调方法上标注@RabbitListener注解，并设置其属性queues，注册监听队列，当该队列收到消息时，标注方法遍会调用

可分别使用Message和保存消息所属对象进行消息接收，若使用Object对象进行消息接收，实际上接收到的也是Message

@RabbitListener(queues = {"chq"})  
public void send(Book book){  
 System.*out*.println(book);  
}  
  
@RabbitListener(queues = {"chq.new"})  
public void send(Message message){  
 System.*out*.println(message);  
}

### AmqpAdmin管理组件

可用于创建和删除exchange、binding和queue

//创建Direct类型的Exchange  
amqpAdmin.declareExchange(new DirectExchange("admin.direct"));  
//创建Queue  
amqpAdmin.declareQueue(new Queue("admin.test"));  
//将创建的队列与Exchange绑定  
amqpAdmin.declareBinding(new Binding("admin.test", Binding.DestinationType.QUEUE,"admin.direct","admin.test",null));

# Spring boot与检索

## ES的简介

我们的应用经常需要添加检索功能，开源的 ElasticSearch 是目前全文搜索引擎的首选。他可以快速的存储、搜索和分析海量数据。Spring Boot通过整合Spring Data ElasticSearch为我们提供了非常便捷的检索功能支持；

Elasticsearch是一个分布式搜索服务，提供Restful API，底层基于Lucene，采用多shard（分片）的方式保证数据安全，并且提供自动resharding的功能，github等大型的站点也是采用了ElasticSearch作为其搜索服务。

文档：

**概念**

员工文档 的形式存储为例：一个文档代表一个员工数据。存储数据到ElasticSearch 的行为叫做 索引 ，但在索引一个文档之前，需要确定将文档存储在哪里。

一个 ElasticSearch 集群可以包含多个 索引 ，相应的每个索引可以包含多个 类型 。 这些不同的类型存储着多个 文档 ，每个文档又有 多个 属性 。

- 索引---数据库

- 类型---表

- 文档---表中的记录

- 属性---列

## ES的安装与运行

9200端口：RESTful API通过HTTP通信

9300端口：Java客户端与ES的原生传输协议和集群交互

# 拉取ES镜像

docker pull elasticsearch:7.6.1

#运行ES

docker run -e "discovery.type=single-node" -e ES\_JAVA\_OPTS="-Xms256m -Xmx256m" -d -p 9200:9200 -p 9300:9300 --name ES03 镜像id

ES\_JAVA\_OPTS指定java虚拟机相关参数

-Xms256m 初始堆内存大小为256m

-Xmx256m 最大堆内存大小为256m

discovery.type=single-node 设置为单点启动

## ES的基础入门

使用postman进行测试操作

**案例**：**创建一个员工目录，并支持各类型检索**

**索引员工文档**

对于员工目录，我们将做如下操作：

每个员工索引一个文档，文档包含该员工的所有信息。

每个文档都将是 employee 类型 。

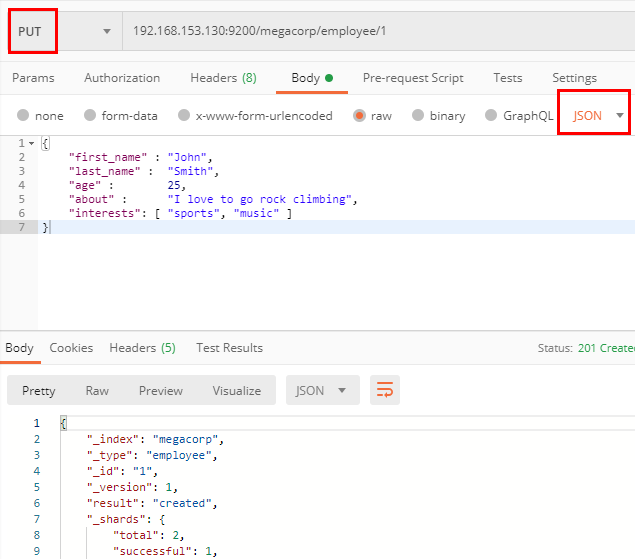
该类型位于 索引 megacorp 内。

**1）添加：**

PUT /megacorp/employee/1  
{  
 "first\_name":"John",  
 "last\_name":"Smith",  
 "age":25,  
 "about":"I love to go rock climbing",  
 "interests":["sports","music"]  
}

注意，路径 /megacorp/employee/1 包含了三部分的信息：

* megacorp :索引名称
* Employee:类型名称
* 1:特定雇员的ID



1. **检索文档**

GET /megacorp/employee/1

{  
    "\_index": "megacorp",  
    "\_type": "employee",  
    "\_id": "1",  
    "\_version": 1,  
    "\_seq\_no": 0,  
    "\_primary\_term": 1,  
    "found": true,  
    "\_source": {  
        "first\_name": "John",  
        "last\_name": "Smith",  
        "age": 25,  
        "about": "I love to go rock climbing",  
        "interests": [  
            "sports",  
            "music"  
        ]  
    }  
}

更多操作去查文档

## Springboot整合ElasticSearch

### 概述

SpringBoot默认支持两种技术来和ES交互；

* Jest（默认不生效）

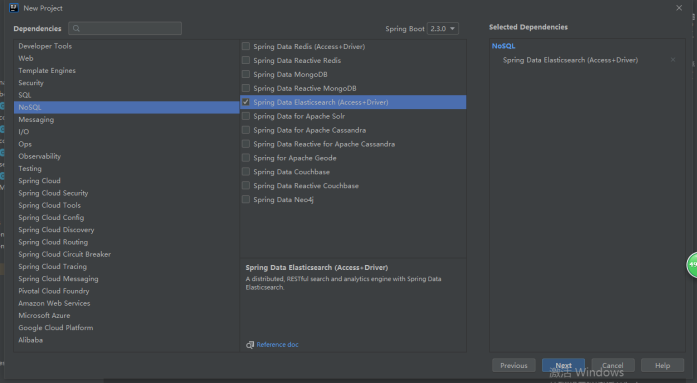
需要导入jest的工具包（io.searchbox.client.JestClient）

从springboot 2.2.0以后被弃用

* SpringData ElasticSearch

### 案例

1）创建项目



会导入依赖：

<dependency>  
 <groupId>org.springframework.boot</groupId>  
 <artifactId>spring-boot-starter-data-elasticsearch</artifactId>  
</dependency>

1. 配置文件

spring:  
 elasticsearch:  
 rest:  
 uris: 192.168.153.130:9200

1. 新建实体和repository

package com.chq.springbootelasticsearchtest.bean;  
  
import org.springframework.data.elasticsearch.annotations.Document;  
  
import java.io.Serializable;  
  
@Document(indexName = "chq")  
public class Book implements Serializable {  
 private Integer id;  
 private String name;  
  
 public Integer getId() {  
 return id;  
 }  
  
 public void setId(Integer id) {  
 this.id = id;  
 }  
  
 public String getName() {  
 return name;  
 }  
  
 public void setName(String name) {  
 this.name = name;  
 }  
}

package com.chq.springbootelasticsearchtest.repository;  
  
import com.chq.springbootelasticsearchtest.bean.Book;  
import org.springframework.data.elasticsearch.repository.ElasticsearchRepository;  
import org.springframework.stereotype.Repository;  
  
import java.util.List;  
  
@Repository  
public interface BookRepository extends ElasticsearchRepository<Book,Integer> {  
 public List<Book> findByNameLike(String name);  
}

1. Test

@Autowired  
BookRepository bookRepository;  
  
@Test  
void contextLoads() {  
 Book book = new Book();  
 book.setName("dddd");  
 book.setId(1);  
 bookRepository.save(book);  
}  
  
@Test  
void search(){  
 List<Book> dd = bookRepository.findByNameLike("dd");  
 for (Book book:  
 dd) {  
 System.*out*.println(book.getName());  
 }  
}

更多做法看spring文档

<https://spring.io/projects/spring-data-elasticsearch#overview>

# Spring boot与任务

## 异步任务

在Java应用中，绝大多数情况下都是通过同步的方式来实现交互处理的；但是在处理与第三方系统交互的时候，容易造成响应迟缓的情况，之前大部分都是使用多线程来完成此类任务，springboot中可以用异步任务解决。

**两个注解：**

@EnableAsync 在主类上标注开启异步任务支持

@Async 在需要异步执行的方法上标注注解

开启异步任务后，当controller层调用该方法会直接返回结果，该任务异步执行

@EnableAsync  
@SpringBootApplication  
public class SprinbootTaskApplication {

@Service  
public class TaskService {  
  
 @Async  
 public void dosome() throws InterruptedException {  
 Thread.*sleep*(5000);  
 System.*out*.println("处理");  
 }  
}

@Autowired  
private TaskService taskService;  
  
@RequestMapping("task")  
public String task() throws InterruptedException {  
 taskService.dosome();  
 return "success";  
}

## 定时任务

两个注解：

@EnableScheduling 标注在主类，开启对定时任务支持

@Scheduled 标注在执行的方法上，并制定cron属性

@EnableScheduling  
@SpringBootApplication  
public class SprinbootTaskApplication {

@Scheduled(cron = "0 \* \* \* \* \*")  
public void hello(){  
 System.*out*.println("hello...");  
}

**cron表达式：**

second(秒), minute（分）, hour（时）, day of month（日）, month（月）, day of week（周几）.

0 0/5 14,18 \* \* ? 每天14点整，和18点整，每隔5分钟执行一次

0 15 10 ? \* 1-6 每个月的周一至周六10:15分执行一次

0 0 2 ? \* 6L 每个月的最后一个周六凌晨2点执行一次

0 0 2 LW \* ? 每个月的最后一个工作日凌晨2点执行一次

0 0 2-4 ? \* 1#1 每个月的第一个周一凌晨2点到4点期间，每个整点都执行一次；

|  |  |  |
| --- | --- | --- |
| **字段** | **允许值** | **允许的特殊字符** |
| 秒 | 0-59 | , - \* / |
| 分 | 0-59 | , - \* / |
| 小时 | 0-23 | , - \* / |
| 日期 | 1-31 | , - \* ? / L W C |
| 月份 | 1-12 | , - \* / |
| 星期 | 0-7或SUN-SAT 0,7是SUN | , - \* ? / L C # |

|  |  |
| --- | --- |
| 特殊字符 | 代表含义 |
| , | 枚举 |
| - | 区间 |
| \* | 任意 |
| / | 步长 |
| ? | 日/星期冲突匹配 |
| L | 最后 |
| W | 工作日 |
| C | 和calendar联系后计算过的值 |
| # | 星期，4#2，第2个星期四 |

## 邮箱

1. 导入依赖

<dependency>  
 <groupId>org.springframework.boot</groupId>  
 <artifactId>spring-boot-starter-mail</artifactId>  
</dependency>

1. 配置邮箱

spring.mail.username=17875511931@163.com  
spring.mail.password=DCHPTCBREIFXWVDD  
spring.mail.host=smtp.163.com

登录163邮箱，找到POP3/SMTP/IMAP，开启SMTP服务，生成授权密码。

1. 测试

@Autowired  
JavaMailSenderImpl javaMailSender;  
  
//简单邮件  
@Test  
void contextLoads() {  
 SimpleMailMessage message = new SimpleMailMessage();  
 //设置主题和内容  
 message.setSubject("今天开会");  
 message.setText("物质楼555开会，不要迟到");  
 //设置发送方和接收方  
 message.setFrom("17875511931@163.com");  
 message.setTo("852620824@qq.com");  
 javaMailSender.send(message);  
}  
  
//复杂邮件  
@Test  
void send() throws MessagingException {  
 MimeMessage message = javaMailSender.createMimeMessage();  
 //multipart=true  
 //开启对内联元素和附件的支持  
 MimeMessageHelper helper = new MimeMessageHelper(message,true);  
  
 helper.setSubject("今天开会");  
 //html=ture  
 //设置content type=text/html，默认为text/plain  
 helper.setText("<b style='color:red'>物质楼555开会，不要迟到</b>",true);  
  
 helper.setFrom("17875511931@163.com");  
 helper.setTo("852620824@qq.com");  
 //设置附件  
 helper.addAttachment("2.png",new File("F:\\文件\\杂物\\img\\head-2.jpg"));  
 javaMailSender.send(message);  
}

# Spring boot与分布式

在分布式系统中，国内常用zookeeper+dubbo组合，而Spring Boot推荐使用全栈的Spring，Spring Boot+Spring Cloud。

## Zookeeper和Dubbo

### 概述

**ZooKeeper**

ZooKeeper 是一个分布式的，开放源码的分布式应用程序协调服务。它是一个为分布式应用提供一致性服务的软件，提供的功能包括：配置维护、域名服务、分布式同步、组服务等。

**Dubbo**

Dubbo是Alibaba开源的分布式服务框架，它最大的特点是按照分层的方式来架构，使用这种方式可以使各个层之间解耦合（或者最大限度地松耦合）。从服务模型的角度来看，Dubbo采用的是一种非常简单的模型，要么是提供方提供服务，要么是消费方消费服务，所以基于这一点可以抽象出服务提供方（Provider）和服务消费方（Consumer）两个角色。

### 安装ZooKeeper：

docker pull zookeeper:latest

docker run --name zk01 -p 2181:2181 --restart always -d 镜像id

## 整合springboot

**1)环境搭建**

创建一个空项目

分别创建provider和consumer模块并分别导入依赖

## Spring Cloud

### 1. 概述

Spring Cloud是一个分布式的整体解决方案。Spring Cloud 为开发者提供了在分布式系统（配置管理，服务发现，熔断，路由，微代理，控制总线，一次性token，全局琐，leader选举，分布式session，集群状态）中快速构建的工具，使用Spring Cloud的开发者可以快速的启动服务或构建应用、同时能够快速和云平台资源进行对接。

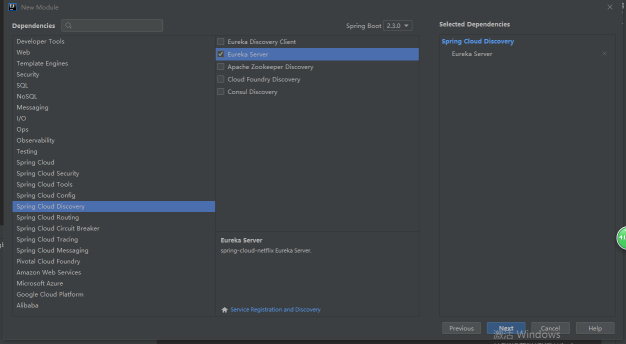
SpringCloud分布式开发五大常用组件

* 服务发现——Netflix Eureka
* 客服端负载均衡——Netflix Ribbon
* 断路器——Netflix Hystrix
* 服务网关——Netflix Zuul
* 分布式配置——Spring Cloud Config

### 2入门

1）新建一个空项目。

2）新建一个注册中心，eureka-server



配置文件

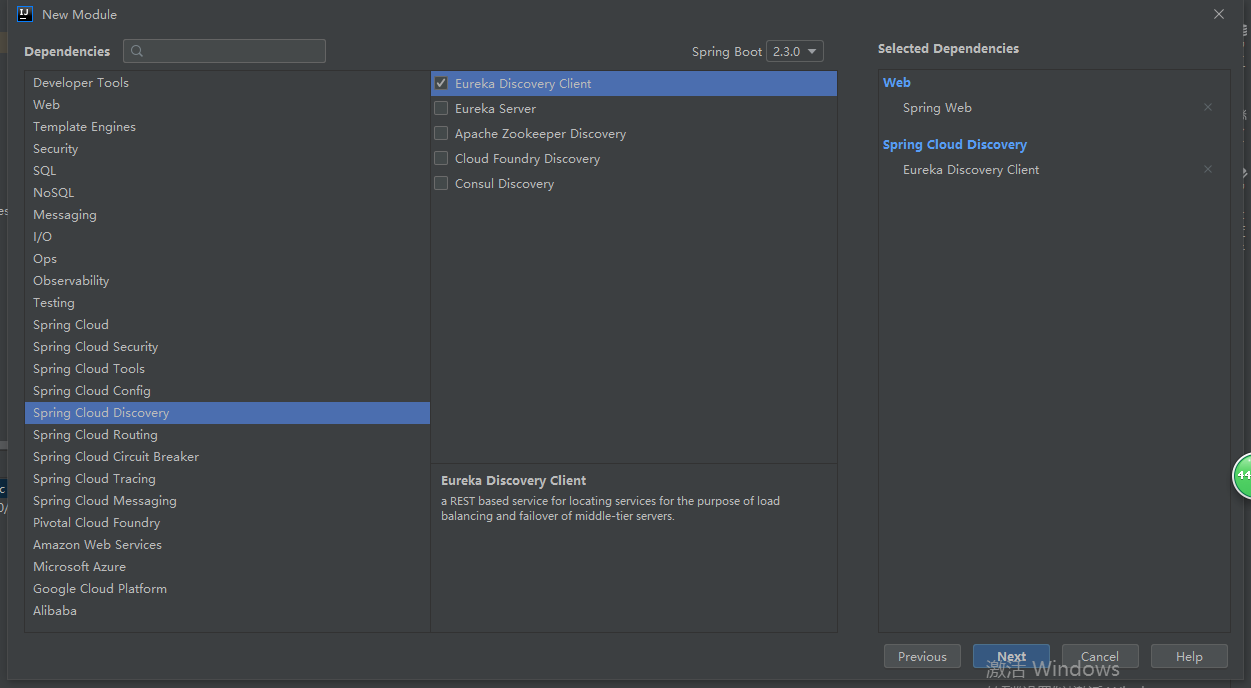
server:  
 port: 8761  
eureka:  
 instance:  
 hostname: eureka-server *# eureka实例的主机名* client:  
 register-with-eureka: false *#不把自己注册到eureka上* fetch-registry: false *#不从eureka上来获取服务的注册信息* service-url:  
 defaultZone: http://localhost:8761/eureka/

启动文件开启eureka服务

@EnableEurekaServer

运行并访问http://localhost:8761/ 可以看到成功了

1. 新建一个服务提供者provider-ticket



配置文件

server:  
 port: 8002  
spring:  
 application:  
 name: provider-ticket  
  
  
eureka:  
 instance:  
 prefer-ip-address: true *# 注册服务的时候使用服务的ip地址* client:  
 service-url:  
 defaultZone: http://localhost:8761/eureka/

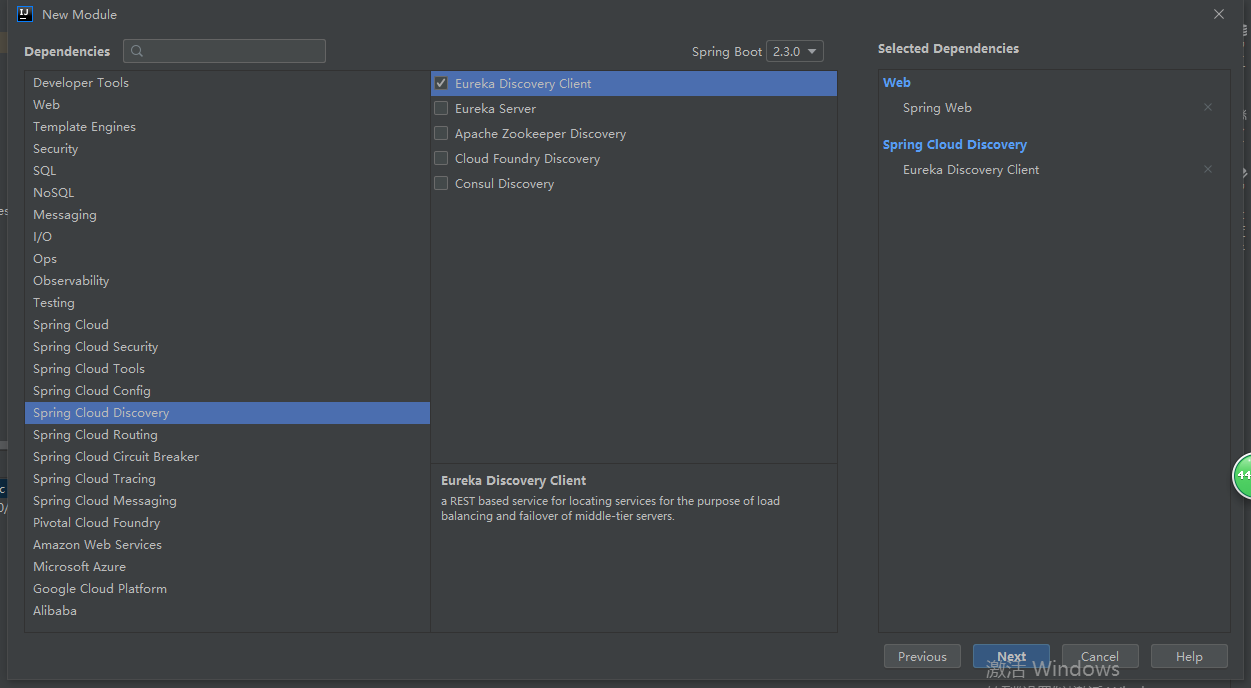
编写controller层和service层demo

@Service  
public class TicketService {  
  
 public String getTicket() {  
 return "<海贼王>";  
 }  
  
}

@Autowired  
TicketService ticketService;  
  
@GetMapping("/ticket")  
public String getTicket(){  
 return ticketService.getTicket();  
}

启动，可以在http://localhost:8761/看到注册信息

5）新建一个服务消费者consumer-user



配置文件

spring:  
 application:  
 name: consumer-user  
server:  
 port: 8200  
  
eureka:  
 instance:  
 prefer-ip-address: true *# 注册服务的时候使用服务的ip地址* client:  
 service-url:  
 defaultZone: http://localhost:8761/eureka/