Springboot教程

1. **简介**

Spring Boot来简化spring应用开发，约定大于配置，去繁从简，just run就能创建一个独立的，产品级的应用。

**背景：**J2EE笨重的开发、繁多的配置、低下的开发效率、复杂的部署流程、第三方技术集成难度大。

**解决：**

“Spring全家桶”时代。

Spring Boot  J2EE一站式解决方案

Spring Cloud  分布式整体解决方案

**优点：**

– 快速创建独立运行的Spring项目以及与主流框架集成

– 使用嵌入式的Servlet容器，应用无需打成WAR包

– starters自动依赖与版本控制

– 大量的自动配置，简化开发，也可修改默认值

– 无需配置XML，无代码生成，开箱即用

– 准生产环境的运行时应用监控

– 与云计算的天然集成

**缺点：**

* 必须对spring有个深入的了解，才能熟练应用spring boot，也就是入门简单，精通难

**微服务**

微服务：架构风格（服务微化）

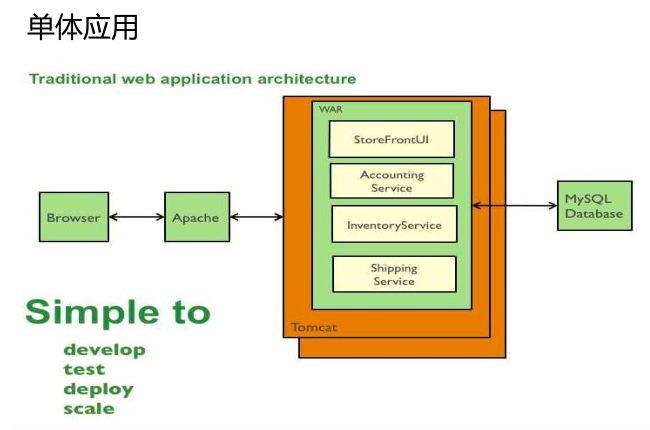
一个应用应该是一组小型服务；可以通过HTTP的方式进行互通；

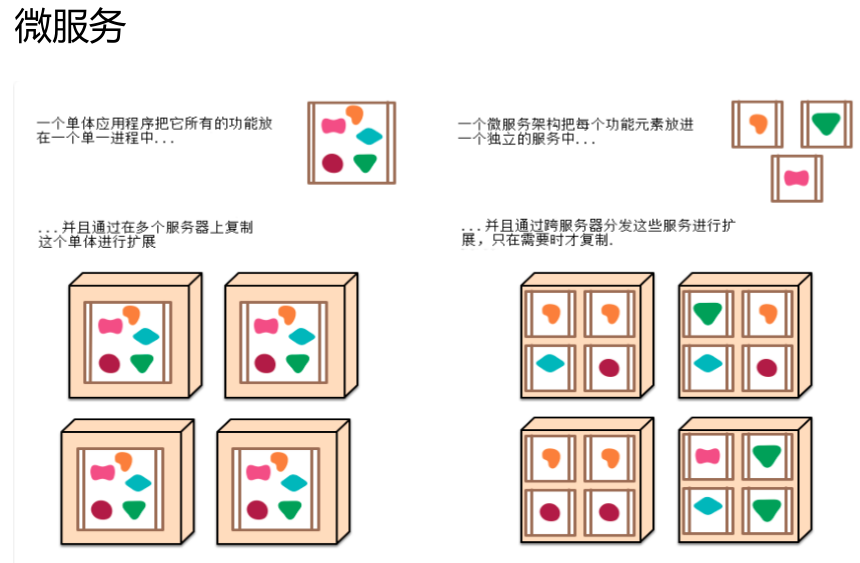
单体应用：ALL IN ONE

微服务：每一个功能元素最终都是一个可独立替换和独立升级的软件单元；

具体参考文档：

https://martinfowler.com/articles/microservices.html#MicroservicesAndSoa





1. **环境准备**

环境约束

–jdk1.8：Spring Boot 推荐jdk1.7及以上；java version "1.8.0\_112"

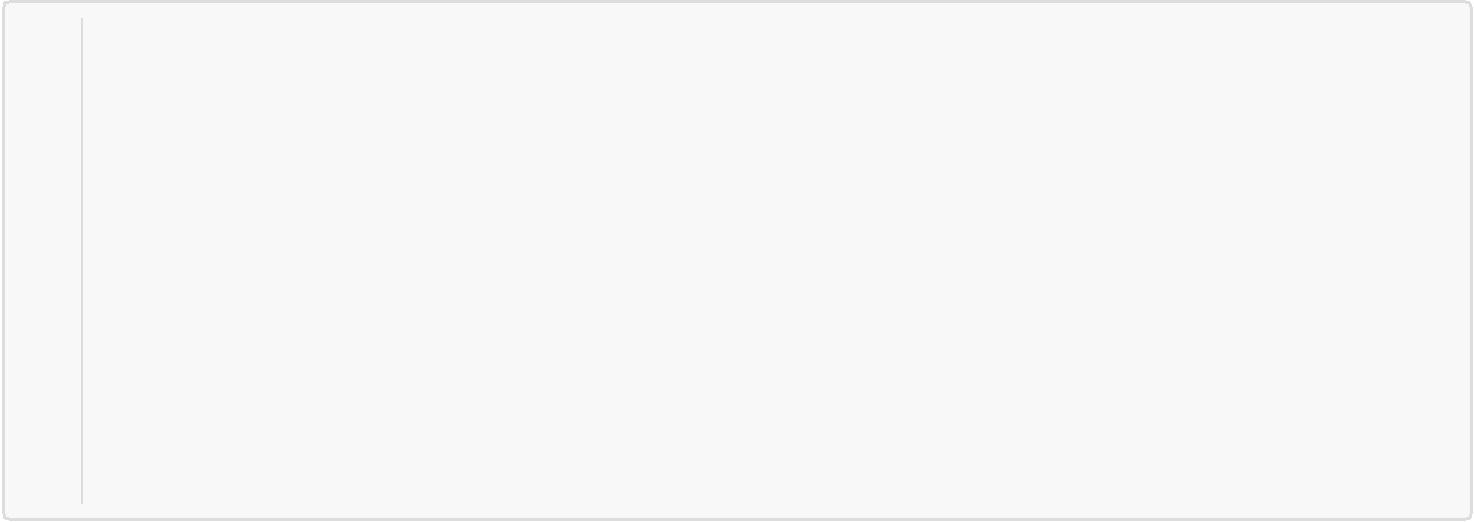
–maven3.x：maven 3.3以上版本；Apache Maven 3.3.9

–IntelliJIDEA2017：IntelliJ IDEA 2017.2.2 x64、STS

–SpringBoot 1.5.9.RELEASE：1.5.9；

**MAVEN设置**

给maven 的settings.xml配置文件的profiles标签添加

<profile>

<id>jdk‐1.8</id>

<activation>

<activeByDefault>true</activeByDefault>

<jdk>1.8</jdk>

</activation>

<properties>

<maven.compiler.source>1.8</maven.compiler.source>

<maven.compiler.target>1.8</maven.compiler.target>

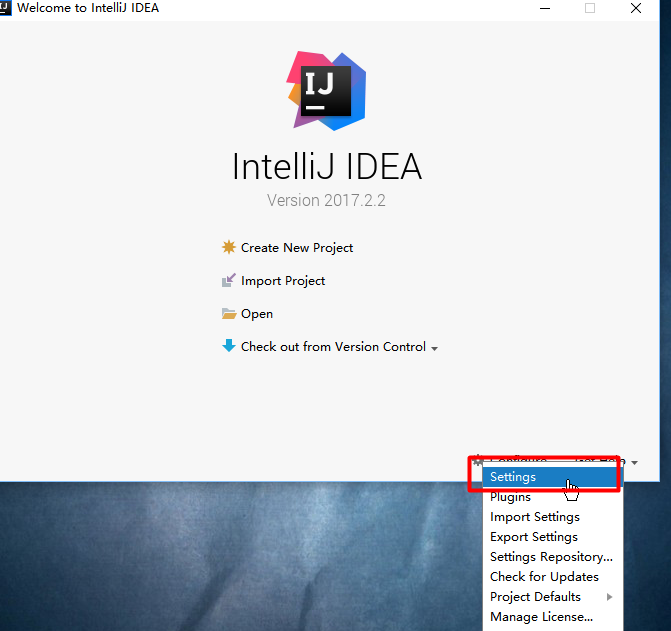
<maven.compiler.compilerVersion>1.8</maven.compiler.compilerVersion>

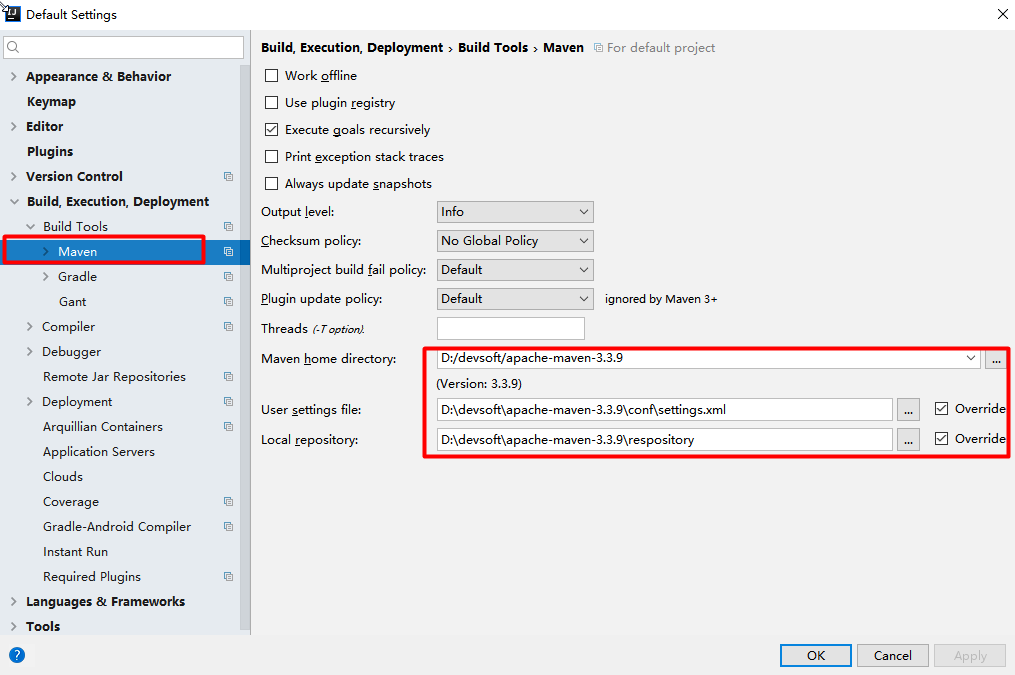
</properties>

</profile>

**IDEA设置**

整合maven进来；





1. **Helloworld的demo**

一个功能：浏览器发送hello请求，服务器接受请求并处理，响应Hello World字符串。

步骤1：

创建一个名为springboot-01的maven项目，具体参考idea创建maven项目步骤。

步骤2：在pom.xml文件中添加如下信息，导入springboot依赖

<parent>  
 <groupId>org.springframework.boot</groupId>  
 <artifactId>spring-boot-starter-parent</artifactId>  
 <version>1.5.9.RELEASE</version>  
</parent>  
<dependencies>  
 <dependency>  
 <groupId>org.springframework.boot</groupId>  
 <artifactId>spring-boot-starter-web</artifactId>  
 </dependency>  
</dependencies>

步骤3：编写一个主程序，用于启动springboot

备注：启动主程序的时候，springboot只会扫描主程序所在目录以及其同级目录的子目录下的controller，而不会扫描其父目录中的controller，因此，controller必须得注意放置的位置

import org.springframework.boot.SpringApplication;  
import org.springframework.boot.autoconfigure.SpringBootApplication;  
  
*/\*\*  
 \** ***@SpringBootApplication 来标注一个主程序类，说明这是一个Spring Boot应用*** *\*    
 \*/*@SpringBootApplication  
public class HelloWorldMain {  
  
 public static void main(String[] args) {  
 // Spring应用启动起来  
 SpringApplication.*run*(HelloWorldMain.class, args);  
 }  
}

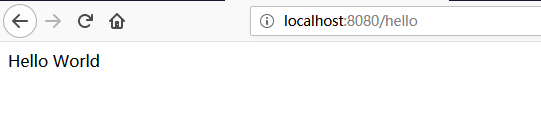
步骤4：编写controller

import org.springframework.stereotype.Controller;  
import org.springframework.web.bind.annotation.RequestMapping;  
import org.springframework.web.bind.annotation.ResponseBody;  
  
*/\*\*  
 \** ***@Description：*** *\** ***@author:*** *hong.guo  
 \** ***@date*** *2018/11/13 11:26  
 \*/*@Controller  
public class HelloWorldController {  
  
 @ResponseBody  
 @RequestMapping("/hello")  
 public String hello(){  
 return "Hello World";  
 }  
}

启动springboot

浏览器输入<http://localhost:8080/hello>

效果图：



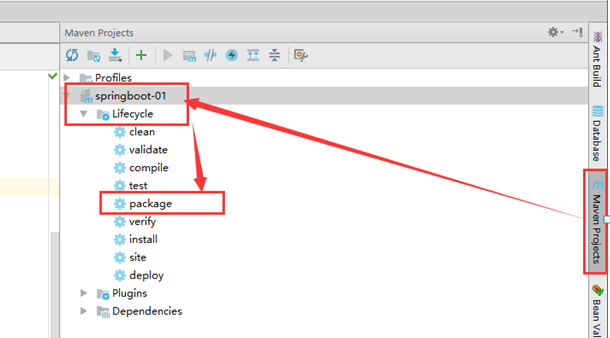
**部署**

在项目的pom.xml文件中添加如下内容

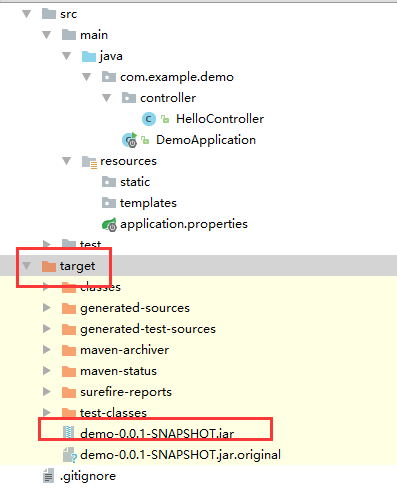
*<!-- 这个插件，可以将应用打包成一个可执行的jar包；-->*<**build**>  
 <**plugins**>  
 <**plugin**>  
 <**groupId**>org.springframework.boot</**groupId**>  
 <**artifactId**>spring-boot-maven-plugin</**artifactId**>  
 </**plugin**>  
 </**plugins**>  
</**build**>

执行maven构建，此时就会得到该项目的可执行jar包，将这个应用打成jar包，直接使用java -jar的命令进行执行就行了，无需将项目打成war包再部署到web服务器。

构建



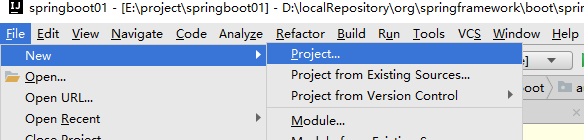
打包后的地址

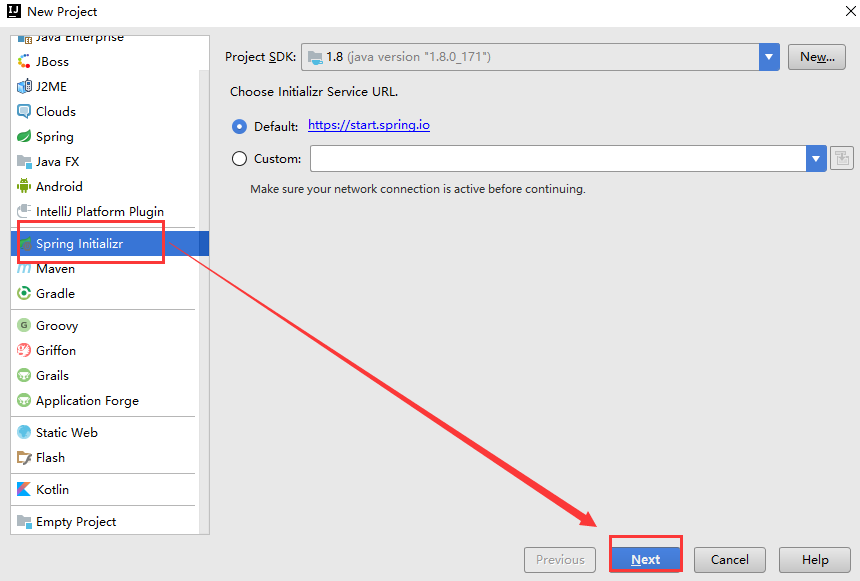


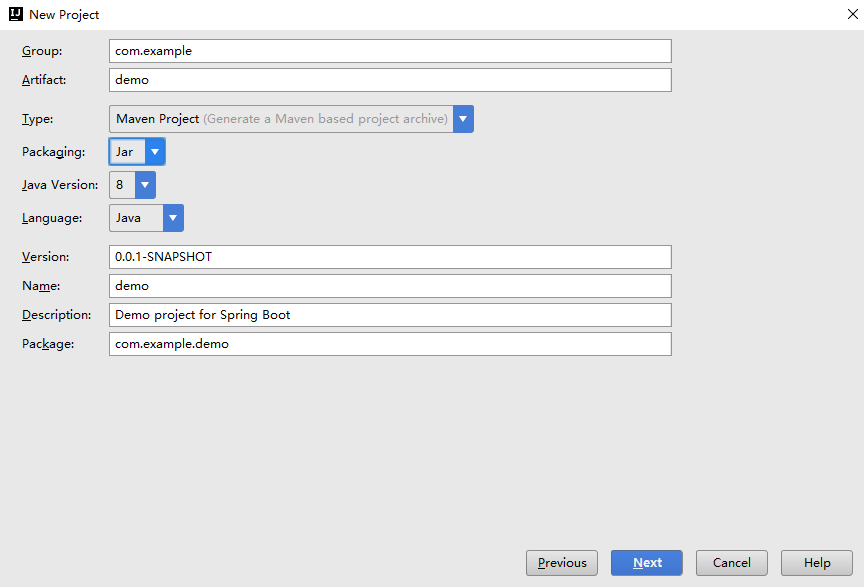
1. **快速创建springboot项目**

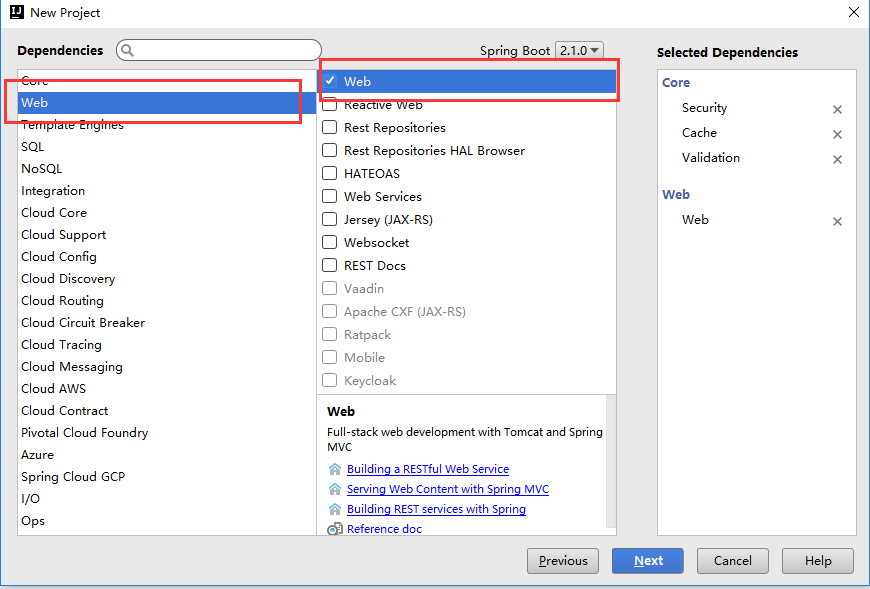
以前我们都是手动去创建springboot项目，现在有一个更快捷的方式，如下是idea版创建

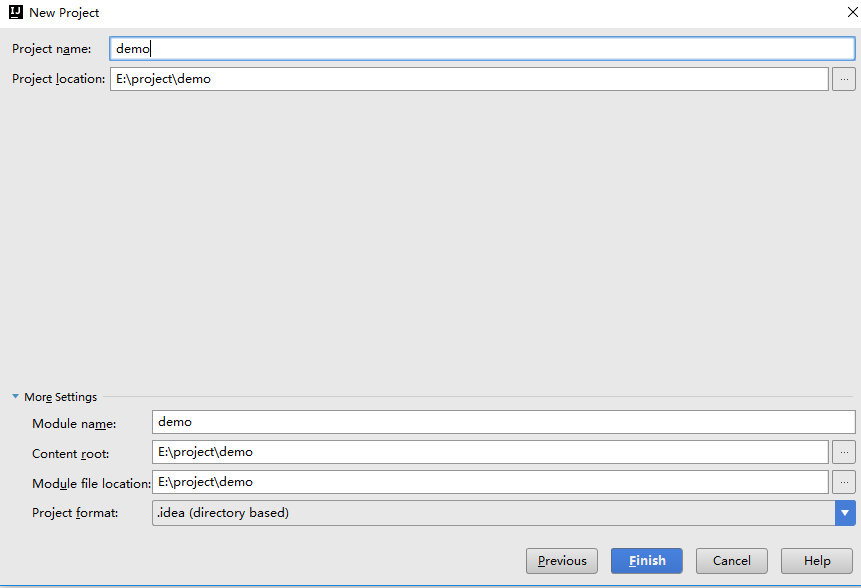
前提：必须联网才行，因为要下载依赖





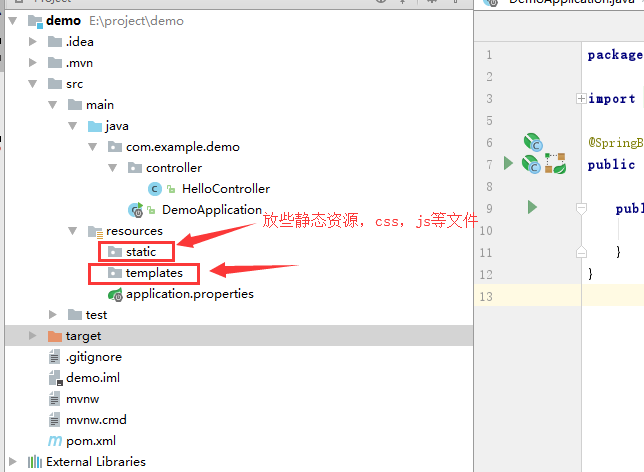






创建完成后，发现pom.xml文件的相关依赖以及其他的东西都已经自动生成了，不需要自己再手动添加

**项目结构：**



1. **SpringBoot原理**
2. **配置文件**
3. **配置文件**

SpringBoot使用一个全局的配置文件，配置文件名是固定的；

•application.properties

•application.yml

配置文件的作用：修改SpringBoot自动配置的默认值；SpringBoot在底层都给我们自动配置好；

YAML（YAML Ain't Markup Language）

YAML A Markup Language：是一个标记语言

YAML isn't Markup Language：不是一个标记语言；

标记语言：

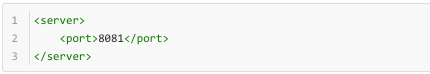
以前的配置文件；大多都使用的是 xxxx.xml文件；

YAML：以数据为中心，比json、xml等更适合做配置文件；

YAML：配置例子



XML：



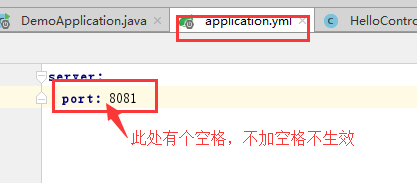
1. **yml**

**yml语法**

k:(空格)v：表示一对键值对（空格必须有）；

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，属性和值也是大小写敏感；

yml配置的时候注意空格



**字面量：普通的值（数字，字符串，布尔）**

k: v：字面直接来写；

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

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

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

**对象、Map（属性和值）（键值对）：**

k: v：在下一行来写对象的属性和值的关系；注意缩进

对象还是k: v的方式

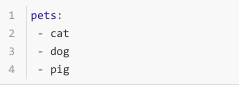


行内写法：



**数组（List、Set）：**

用- 值表示数组中的一个元素



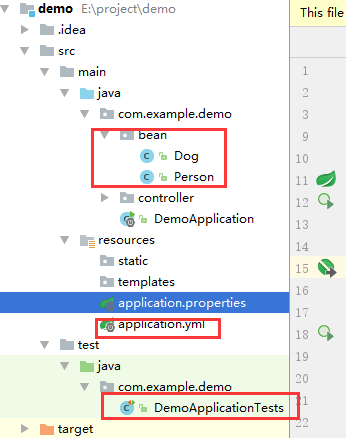
行内写法



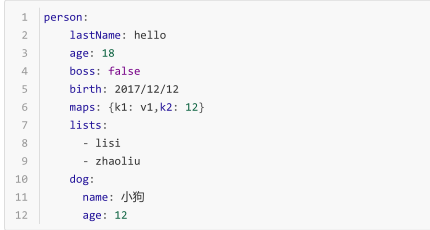
**示例**

功能：实现配置文件值注入

工程结构：



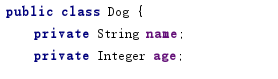
yml配置文件内容：



Person.java代码：



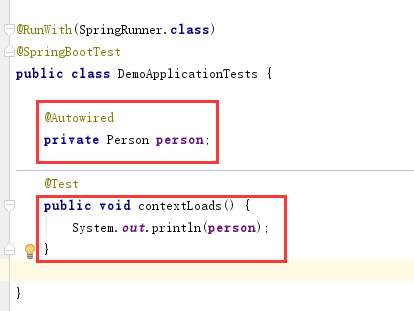
Dog.java文件代码：



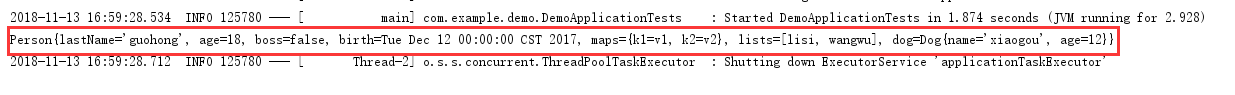
Pom.xml文件修改：必须添加如下依赖，Person类才能使用ConfigurationProperties注解

<**dependency**>  
 <**groupId**>org.springframework.boot</**groupId**>  
 <**artifactId**>spring-boot-configuration-processor</**artifactId**>  
 <**optional**>true</**optional**>  
</**dependency**>

测试：

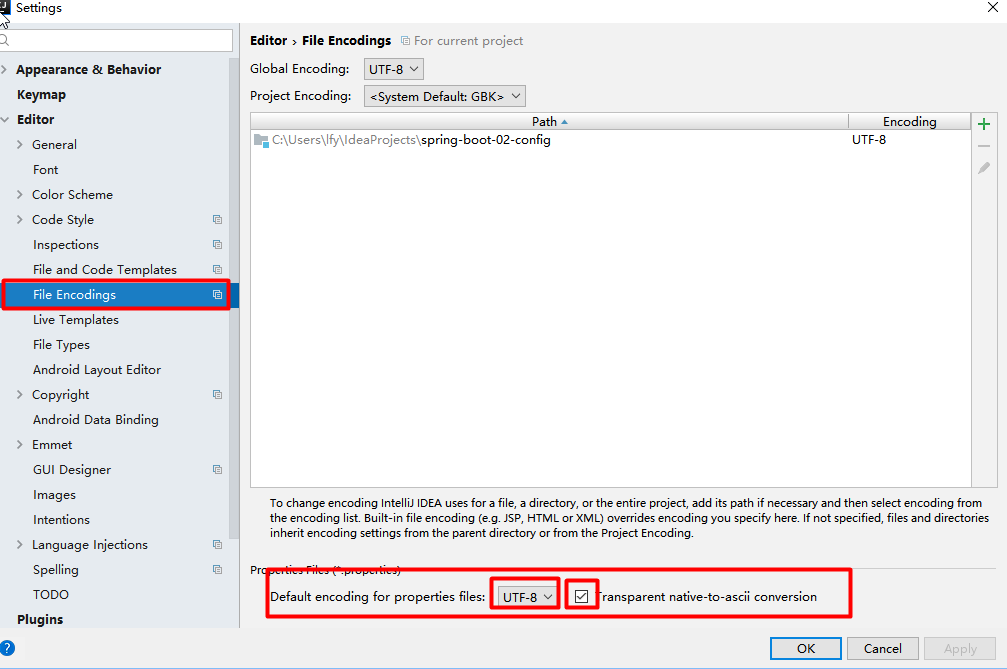


结果：



1. **properties配置文件乱码**

调整



1. **@Value获取值和@ConfigurationProperties获取值比较**

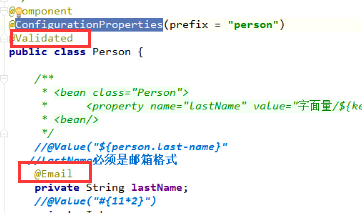


**松散绑定**：比如Javabean有个字段名是lastName,我们在ConfigurationProperties注入时，yml里面配置可以是last-name，但是在使用value注入的时候，yml里面的名称只能是lastName。

**Spel**: @Value("#{11\*2}")，value支持表达式，能计算出结果是22

**JSR303数据校验**：

在Javabean头上加上@Validated注解，并且在相应的字段上加上注解，在数据注入的时候，ConfigurationProperties方式就能校验数据格式，而value注入方式则不支持



**复杂类型封装**：比如类嵌套，list，map等等，ConfigurationProperties方式支持。



@value方式会报错，@ConfigurationProperties则不会

配置文件yml还是properties他们都能获取到值；

如果说，我们只是在某个业务逻辑中需要获取一下配置文件中的某项值，使用@Value；

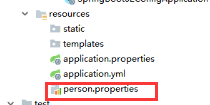
如果说，我们专门编写了一个javaBean来和配置文件进行映射，我们就直接使用@ConfigurationProperties；

1. **@PropertySource和@ImportResource和@Bean**

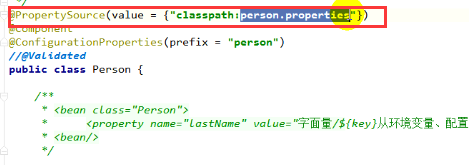
**@PropertySource**

@ConfigurationProperties默认从全局配置文件中获取值，如果所有的配置信息都放在全局配置文件中，则会非常臃肿，因此我们可以把一些无关的信息放到其他地方，然后使用@PropertySource加载指定的配置文件。

person.properties文件路径如下：



指定文件;

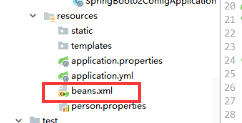


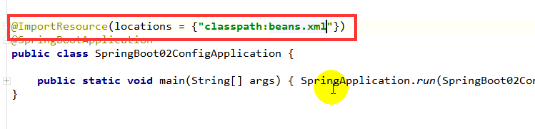
**@ImportResource**

导入Spring的配置文件，让配置文件里面的内容生效；Spring Boot里面没有Spring的配置文件（beans.xml），我们自己编写的配置文件，也不能自动识别；

想让Spring的配置文件生效，加载进来；@ImportResource标注在一个配置类上。

如下我们可以加在springboot的主程序类上





**@Bean**

此方式是Springboot推荐给容器添加组件的方式：全注解的方式

如果我们要给容器添加组件，就得编写spring的配置文件，然后使用指定方式加载配置文件，这样做起来很麻烦，特别是编写spring的配置文件内容。

鉴于此，我们使用全注解的方式

bean类HelloService内容：

|  |
| --- |
| **public class** HelloService { } |

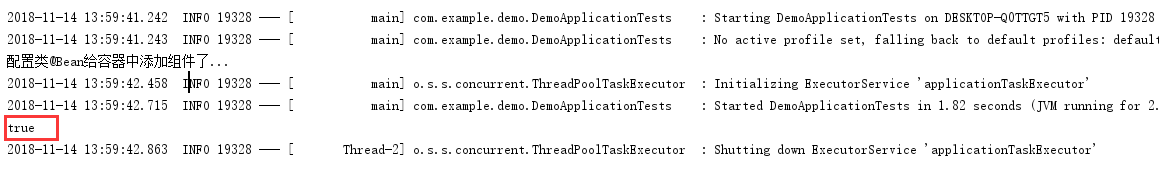
配置类MyAppConfig

|  |
| --- |
| */\*\*  \*****@Configuration:*** *指明当前类是一个配置类；就是来替代之前的Spring配置文件  \*/* @Configuration **public class** MyAppConfig {  *//将方法的返回值添加到容器中；容器中这个组件默认的id就是方法名* @Bean  **public** HelloService helloService(){  System.***out***.println(**"配置类@Bean给容器中添加组件了..."**);  **return new** HelloService();  } } |

测试类：

|  |
| --- |
| @RunWith(SpringRunner.**class**) @SpringBootTest **public class** DemoApplicationTests {  @Autowired  **private** ApplicationContext **ioc**;  @Test  **public void** testHelloService() {  **boolean** b = **ioc**.containsBean(**"helloService"**);  System.***out***.println(b);  }  } |

测试效果：



1. **配置文件占位符**

备注：此方式针对yml和properties文件都生效。

application.properties内容：

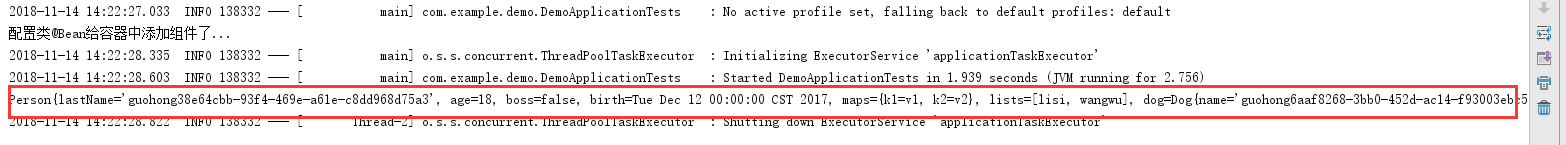
|  |
| --- |
| *#可以调用springboot自带的函数* **person.lastName**=**guohong${random.uuid} person.age**=**18 person.boss**=**false person.birth**=**2017/12/12 person.maps.k1**=**v1 person.maps.k2**=**v2 person.lists**=**lisi,wangwu** *#也可以使用前面自己定义的字段的值* **person.dog.name**=**${person.lastName}\_dog person.dog.age**=**12** |

application.yml

|  |
| --- |
| **person:  lastName:** guohong${**random.uuid**}  **age:** 18  **boss:** false  **birth:** 2017/12/12  **maps:** {  **k1:** v1,  **k2:** v2  }  **lists:** - lisi  - wangwu  **dog:  name:** ${**person.lastName**}\_xiaogou  **age:** 12 |

Person类代码如上所示，此处省略

测试结果：



注：如果${person.lastName}在获取值的时候，发现没有lastName的值，也就是lastName值在系统未设定，那么在注入值的时候，会把${person.lastName}当做字符串直接注入到字段中。也就是name的值最终输出为：${person.lastName}\_xiaogou

1. **Profile多环境支持**

Profile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。比如我们有开发环境的配置信息放在dev文件夹，测试配置信息放在sit或者uat文件夹，正式环境在prod下，在不同的环境下需要使用不同的配置信息，这时profile可以做到快速切换。

实现方式如下

**多profile文件**

我们在主配置文件编写的时候，文件名可以是 application-{profile}.properties/yml（properties或者yml文件都行）

不指定配置文件时，spring默认使用的是application.properties的配置；

指定激活某个配置

application-prod.properties文件中的配置内容：

|  |
| --- |
| **server.port**=**8082** |

application.properties内容：

|  |
| --- |
| **server.port**=**8080** |

在application.properties文件中添加如下内容

|  |
| --- |
| **spring.profiles.active**=**prod** |

即便是application.properties文件中已经指定了端口号，此时还是会使用application-prod.properties文件中的配置。

效果：



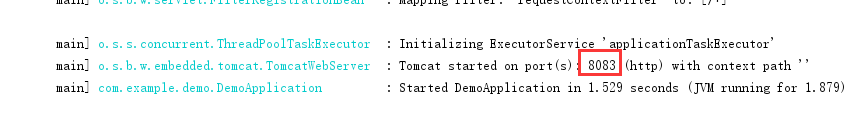
**yml多文档块方式**

application.yml配置内容

|  |
| --- |
| **server:  port:** 8081 **spring:  profiles:  active:** prod  --- **server:  port:** 8082 **spring:  profiles:** dev  --- **server:  port:** 8083 **spring:  profiles:** prod |

此处指定激活prod配置，启动服务，发现使用的是端口号8083

效果：



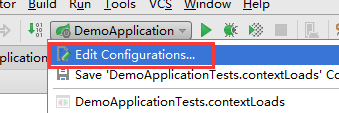
**命令行激活方式**

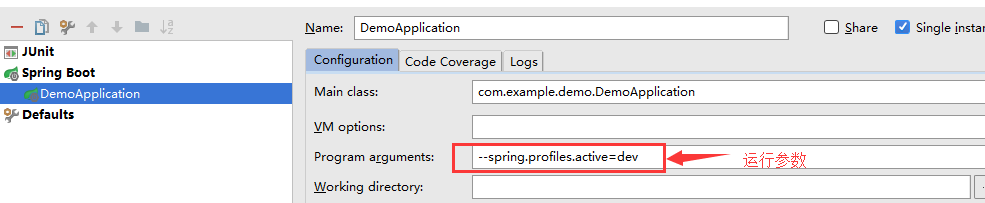
在运行的时候带上命令指定，推荐使用该方式

注：Maven package打包

java -jar spring-boot-02-config-0.0.1-SNAPSHOT.jar --spring.profiles.active=dev；

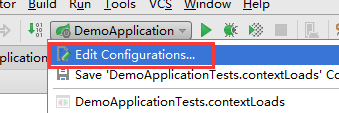
我们也可以在idea中界面添加启动参数

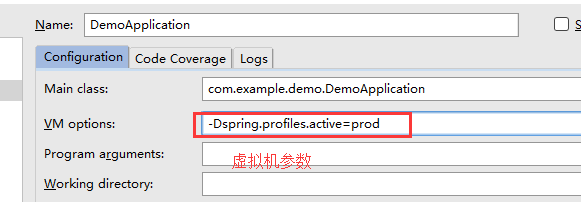




**虚拟机参数方式**

Idea添加虚拟机参数



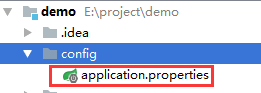


-Dspring.profiles.active=dev

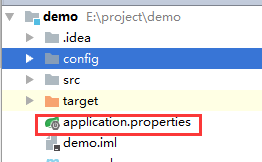
1. **配置文件加载优先级**

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。

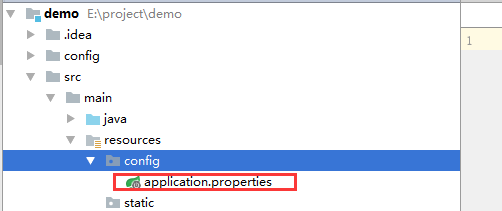
–file:./config/



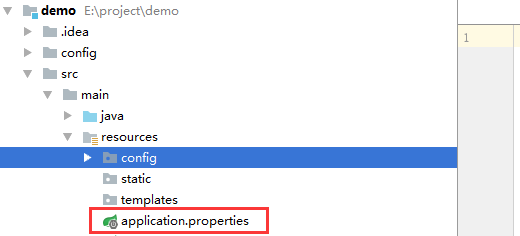
–file:./



–classpath:/config/



–classpath:/



优先级由高到底，高优先级的配置会覆盖低优先级的配置；

SpringBoot会从这四个位置全部加载主配置文件；互补配置；也就是高优先级的配置文件没有的配置信息则以优先级低的配置信息为准。

我们还可以通过spring.config.location来改变默认的配置文件位置

项目打包好以后，我们可以使用命令行参数的形式，启动项目的时候来指定配置文件的新位置；指定配置文件和默认加载的这些配置文件共同起作用形成互补配置；

java -jar spring-boot-02-config-02-0.0.1-SNAPSHOT.jar --spring.config.location=G:/application.properties

G：指的是windows电脑中的G盘

1. **外部配置的加载顺序**

我们可以在springboot的项目中编写配置信息，也可以在项目外编写配置信息。如下就是spring boot项目能编写外部配置信息的地方

SpringBoot也可以从以下位置加载配置； 优先级从高到低；高优先级的配置覆盖低优先级的配置，所有的配置会形成互补配置

1. 命令行参数

在项目打包后，我们想修改服务启动的端口号，则可以通过命令行的方式,如下是修改服务端口号为9090

java -jar spring-boot-02-config-0.0.1-SNAPSHOT.jar --server.port =9090；

1. 来自java:comp/env的JNDI属性
2. Java系统属性（System.getProperties()）
3. 操作系统环境变量
4. RandomValuePropertySource配置的random.\*属性值

由jar包外向jar包内进行寻找；优先加载带profile

官方文档介绍位置：

<https://docs.spring.io/spring-boot/docs/2.1.0.RELEASE/reference/html/boot-features-external-config.html>

6)jar包外部的application-{profile}.properties或application.yml(带spring.profile)配置文件

7)jar包内部的application-{profile}.properties或application.yml(带spring.profile)配置文件

再来加载不带profile

8)jar包外部的application.properties或application.yml(不带spring.profile)配置文件

9)jar包内部的application.properties或application.yml(不带spring.profile)配置文件

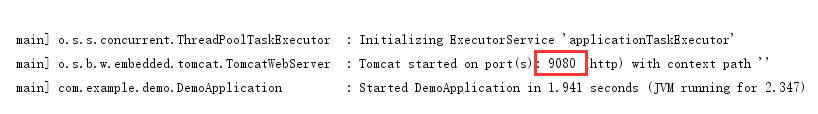
application.properties内容：

|  |
| --- |
| **server.port**=**9080** |

结构：jar和配置文件在同级目录



启动效果：外部的配置信息生效，内部的被覆盖



10.@Configuration注解类上的@PropertySource

11.通过SpringApplication.setDefaultProperties指定的默认属性

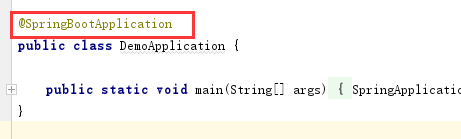
所有支持的配置加载来源；

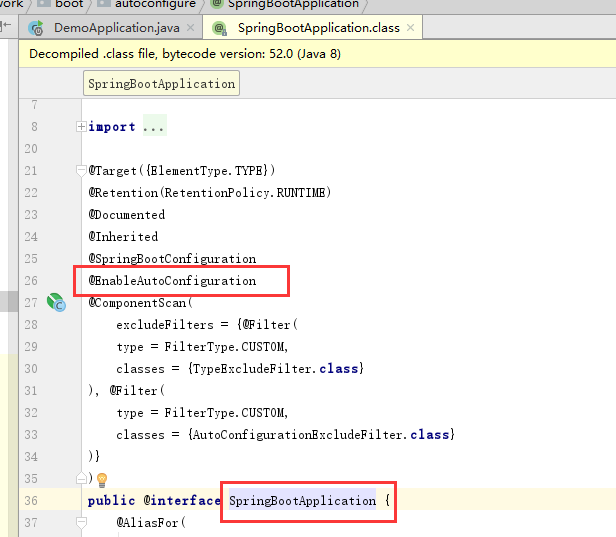
1. **自动配置的原理**

配置文件能配置的属性参考官方文档：

<https://docs.spring.io/spring-boot/docs/2.1.0.RELEASE/reference/html/common-application-properties.html>

自动配置原理





1）SpringBoot启动的时候加载主配置类，开启了自动配置功能 @EnableAutoConfiguration

2）@EnableAutoConfiguration 作用：

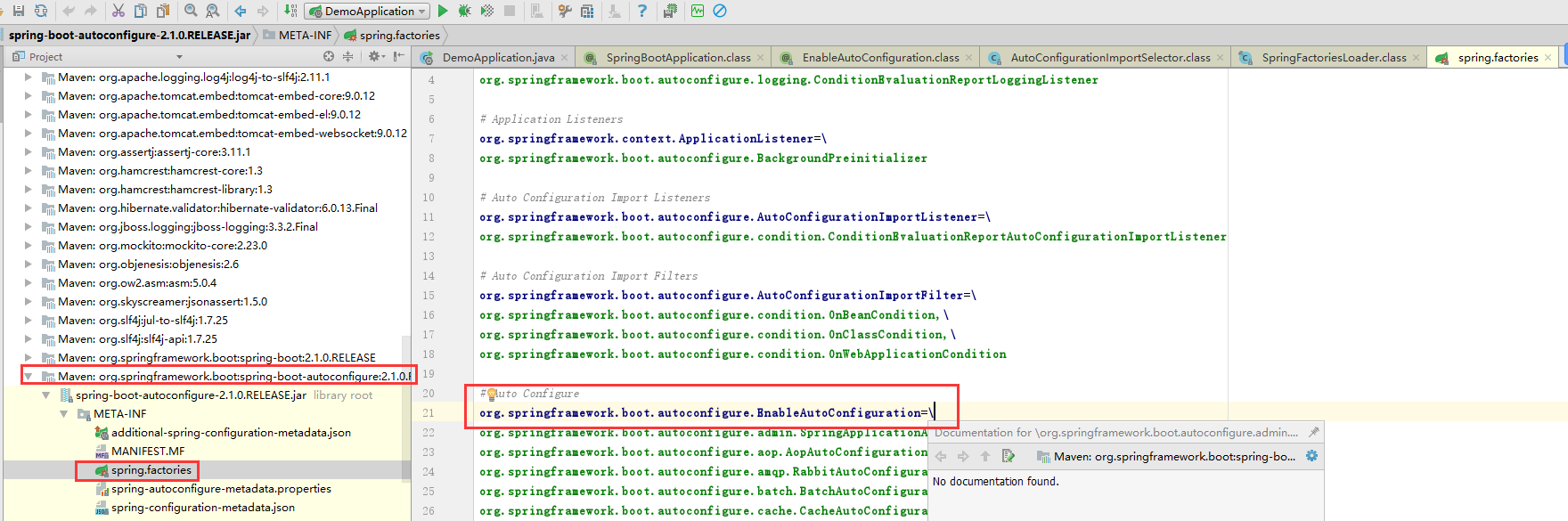
利用EnableAutoConfigurationImportSelector给容器中导入一些组件

可以查看selectImports()方法的内容；

List configurations

= getCandidateConfigurations(annotationMetadata, attributes);获取候选的配置。

|  |
| --- |
| SpringFactoriesLoader.loadFactoryNames()  扫描所有jar包类路径下  META‐INF/spring.factories  把扫描到的这些文件的内容包装成properties对象  从properties中获取到EnableAutoConfiguration.class类（类名）对应的值，然后把他们添加在容器中 |



将类路径下META-INF/spring.factories里面配置的所有EnableAutoConfiguration的值加入到了容器中；

此文件spring.factories中每一个这样的 xxxAutoConfiguration类都是容器中的一个组件，都加入到容器中；用他们来做自动配置；

3）每一个自动配置类进行自动配置功能；

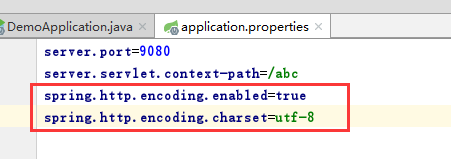
4）以HttpEncodingAutoConfiguration（Http编码自动配置）为例解释自动配置原理；





根据当前不同的条件判断，决定这个配置类是否生效，一但这个配置类生效；这个配置类就会给容器中添加各种组件；这些组件的属性是从对应的properties类中获取的，这些类里面的每一个属性又是和配置文件绑定的。

我们还可以在配置文件中修改这些配置信息，如下所示：



**精髓：**

1）、SpringBoot启动会加载大量的自动配置类

2）、我们看我们需要的功能有没有SpringBoot默认写好的自动配置类；

3）、我们再来看这个自动配置类中到底配置了哪些组件；（只要我们要用的组件有，我们就不需要再来配置了）

4）、给容器中自动配置类添加组件的时候，会从properties类中获取某些属性。我们就可以在配置文件中指定这些属性的值；

xxxxAutoConfigurartion：自动配置类；

给容器中添加组件

xxxxProperties:封装配置文件中相关属性；

1. **@Conditional派生注解**

作用：必须是@Conditional指定的条件成立，才给容器中添加组件，配置配里面的所有内容才生效；



自动配置类必须在一定的条件下才能生效；

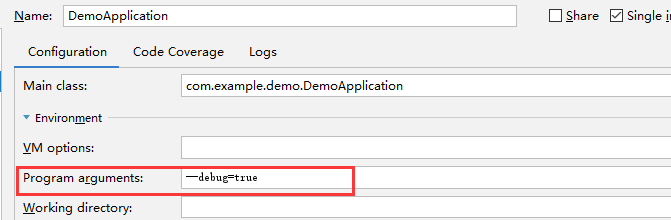
1. **Springboot打印自动配置类日志**

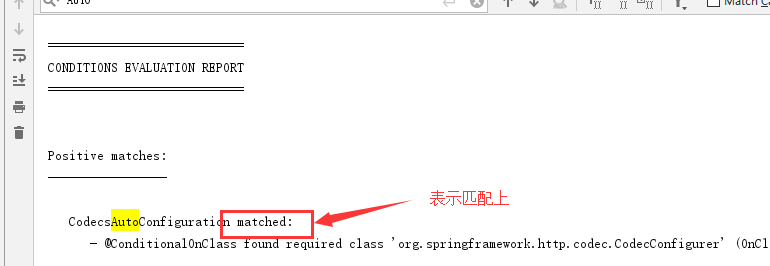
我们怎么知道哪些自动配置类生效？

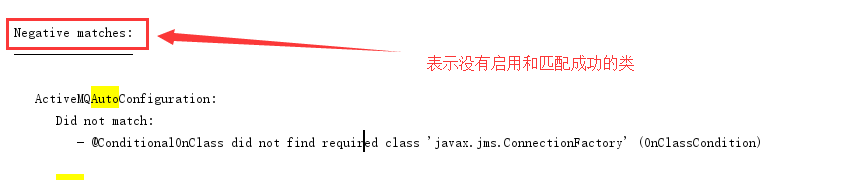
我们可以通过启用 debug=true属性；来让控制台打印自动配置报告，这样我们就可以很方便的知道哪些自动配置类生效；

java –jar XXX.jar –debug=true

或者







1. **日志**

**背景**：小张；开发一个大型系统；

1、System.out.println("")；将关键数据打印在控制台；去掉？写在一个文件？

2、框架来记录系统的一些运行时信息；日志框架 ； zhanglogging.jar；

3、高大上的几个功能？异步模式？自动归档？xxxx？ zhanglogging-good.jar？

4、将以前框架卸下来？换上新的框架，重新修改之前相关的API；zhanglogging-prefect.jar；

5、JDBC---数据库驱动；

写了一个统一的接口层；日志门面（日志的一个抽象层）；logging-abstract.jar；

给项目中导入具体的日志实现就行了；我们之前的日志框架都是实现的抽象层；

**市面上的日志框架**；

JUL、JCL、Jboss-logging、logback、log4j、log4j2、slf4j....



左边选一个门面（抽象层）、右边来选一个实现；

日志门面： SLF4J；

日志实现：Logback；

SpringBoot：底层是Spring框架，Spring框架默认是用JCL；‘

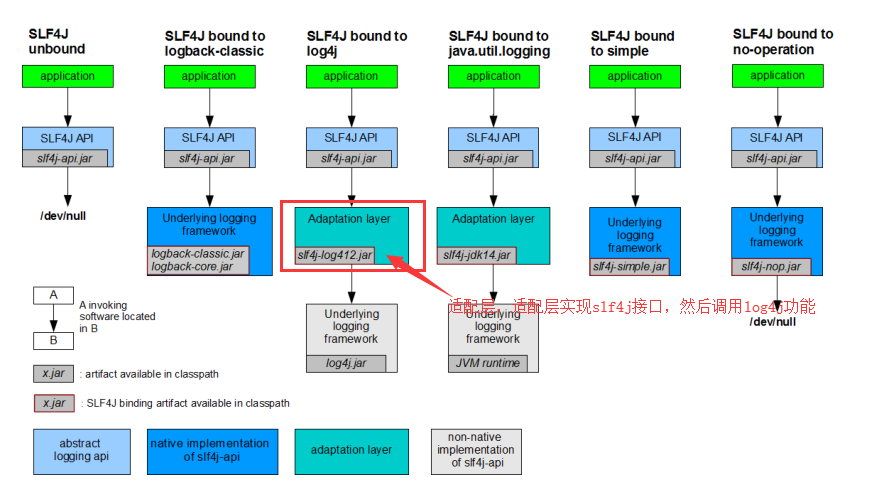
SpringBoot选用 SLF4j和logback；

1. **SLF4j使用**

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

图示：



绿色：抽象接口

浅蓝色：适配层

蓝色：实现层

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；也就是如果实现框架是log4j那么配置文件则按照log4j的配置方式进行配置。

1. **遗留问题**

系统A（slf4j+logback）: Spring（commons-logging）、Hibernate（jboss-logging）、MyBatis、xxxx。

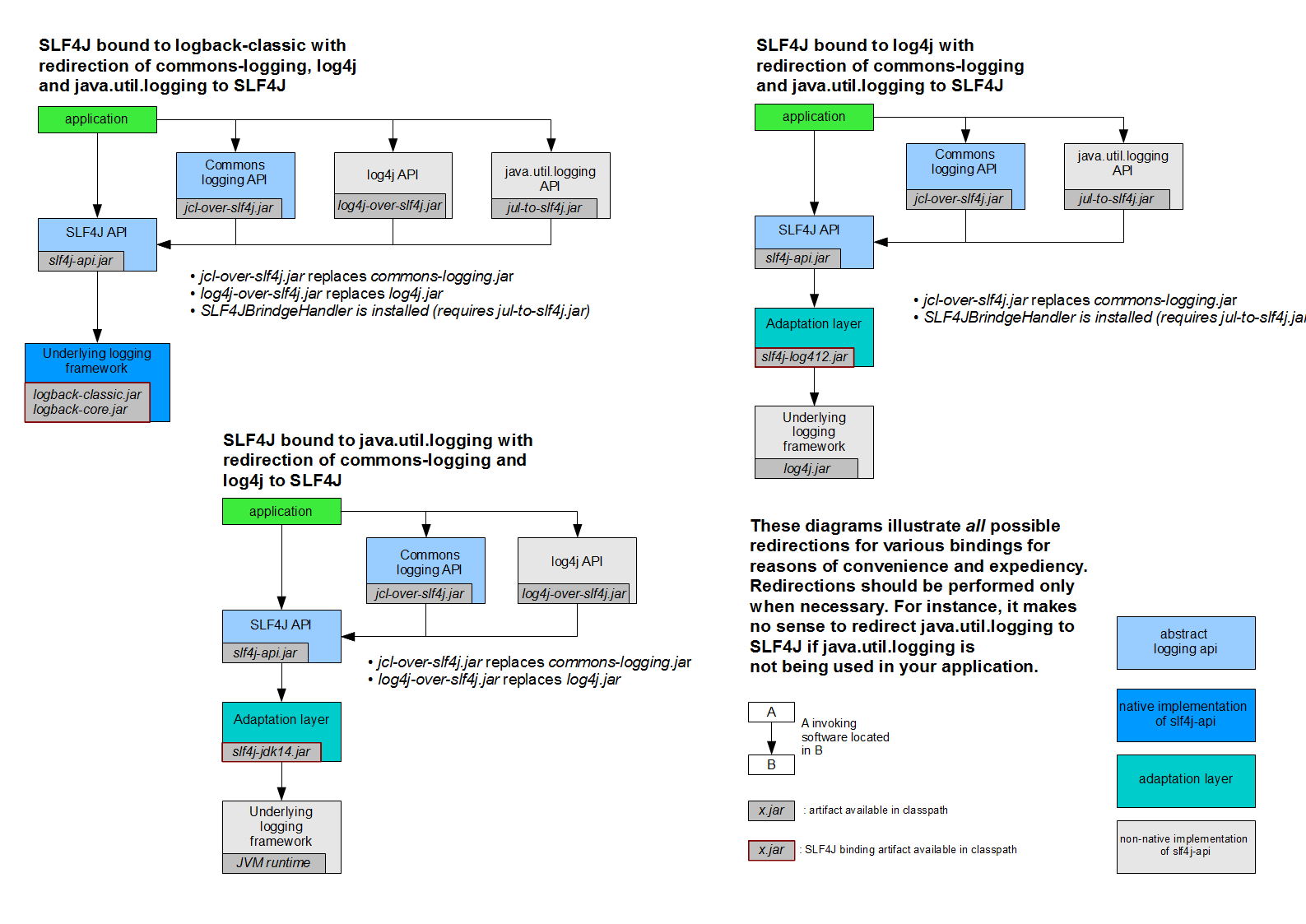
如上所示，我们系统用的日志框架和系统所依赖的框架所用的日志各有不同，Spring用commons-logging，系统A用logback，那我们的系统A如何进行日志输出呢？

**解决方法**：统一日志记录，即使是别的框架和我一起统一使用slf4j进行输出。

1、将系统中其他日志框架先排除出去；

2、用中间包来替换原有的日志框架；

3、我们导入slf4j其他的实现



1. **SpringBoot日志关系**



SpringBoot使用它来做日志功能；

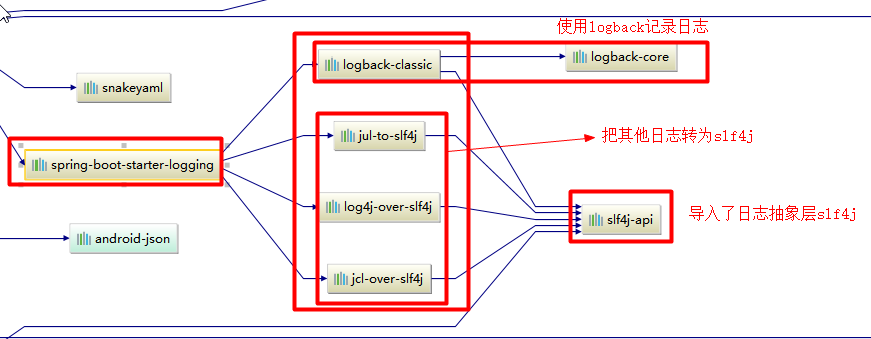


Springboot官方文档：

https://docs.spring.io/spring-boot/docs/2.1.0.RELEASE/reference/html/boot-features-logging.html

底层依赖关系

备注：不同的springboot版本，这块的依赖关系是不一样的，下面贴出的是springboot1.5版本

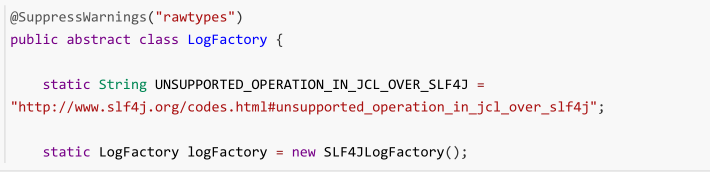


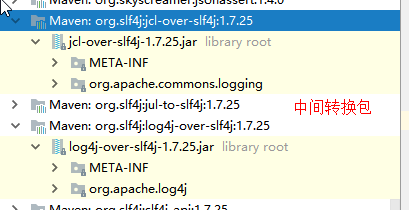
总结：

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

2）、SpringBoot也把其他的日志都替换成了slf4j；

3）、中间替换包





4）、如果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

Spring框架用的是commons-logging；因此要移除掉



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；无需其他操作，springboot就能适配成功

1. **Springboot项目默认日志**

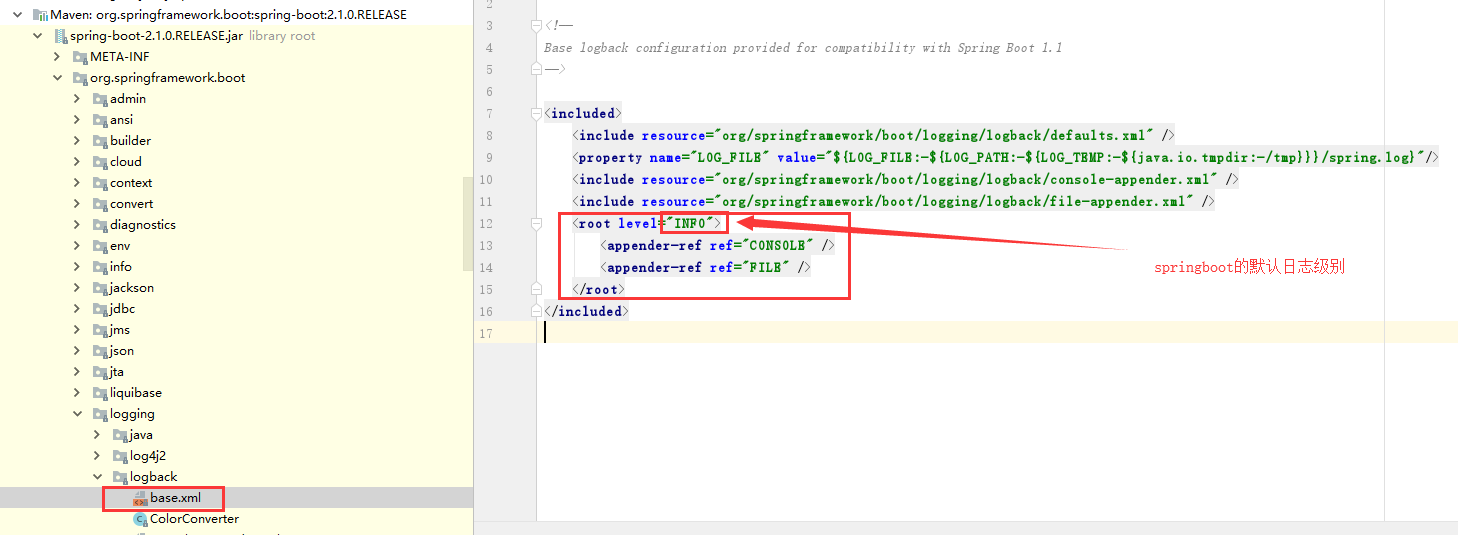
Springboot默认已经为我们配置好了日志，也就是我们创建一个springboot项目时，直接运行，其实是会打印日志的，这就是默认配置。

|  |
| --- |
| *//记录器* Logger logger = LoggerFactory.*getLogger*(DemoApplication.**class**); *// 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..."**); |

效果：



说明springboot项目默认是info级别,在如下位置可以看到

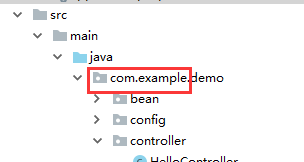


**修改日志级别**

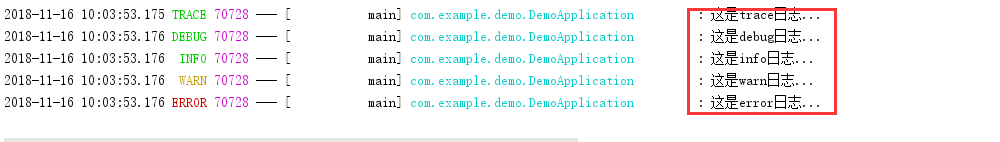
在springboot配置文件application.properties文件中添加如下：

|  |
| --- |
| **logging.level.com.example**=**trace** |

com.example是我们的包名，也就是该包下的所有类的日志级别是trace



效果：

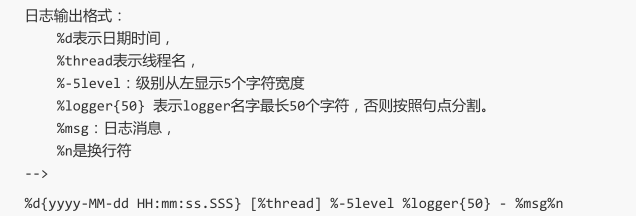


**指定日志输出地址**

|  |
| --- |
| *#不指定路径在当前项目下生成springboot.log日志,也可以指定完整的路径；* **logging.file**=**demo2.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** |



**日志输出格式**



**自定义日志**

将logback.xml文件放到springboot的配置文件夹目录下，就可以直接生效，logback.xml内容如下：

|  |
| --- |
| *<?***xml version="1.0" encoding="UTF-8"***?> <!-- scan：当此属性设置为true时，配置文件如果发生改变，将会被重新加载，默认值为true。 scanPeriod：设置监测配置文件是否有修改的时间间隔，如果没有给出时间单位，默认单位是毫秒当scan为true时，此属性生效。默认的时间间隔为1分钟。 debug：当此属性设置为true时，将打印出logback内部日志信息，实时查看logback运行状态。默认值为false。 -->* <**configuration scan="false" scanPeriod="60 seconds" debug="false"**>  *<!-- 定义日志的根目录 -->* <**property name="LOG\_HOME" value="./app/log"** />  *<!-- 定义日志文件名称 -->* <**property name="appName" value="gh-demo02"**></**property**>  *<!-- ch.qos.logback.core.ConsoleAppender 表示控制台输出 -->* <**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"**>  <**pattern**>%d{yyyy-MM-dd HH:mm:ss.SSS}---> [%thread] %-5level %logger{50} - %msg%n</**pattern**>  </**layout**>  </**appender**>   *<!-- 滚动记录文件，先将日志记录到指定文件，当符合某个条件时，将日志记录到其他文件 -->* <**appender name="appLogAppender" class="ch.qos.logback.core.rolling.RollingFileAppender"**>  *<!-- 指定日志文件的名称 -->* <**file**>${LOG\_HOME}/${appName}.log</**file**>  *<!--  当发生滚动时，决定 RollingFileAppender 的行为，涉及文件移动和重命名  TimeBasedRollingPolicy： 最常用的滚动策略，它根据时间来制定滚动策略，既负责滚动也负责出发滚动。  -->* <**rollingPolicy class="ch.qos.logback.core.rolling.TimeBasedRollingPolicy"**>  *<!--  滚动时产生的文件的存放位置及文件名称 %d{yyyy-MM-dd}：按天进行日志滚动*  *%i：当文件大小超过maxFileSize时，按照i进行文件滚动  -->* <**fileNamePattern**>${LOG\_HOME}/${appName}-%d{yyyy-MM-dd}-%i.log</**fileNamePattern**>  *<!--   可选节点，控制保留的归档文件的最大数量，超出数量就删除旧文件。假设设置每天滚动，且maxHistory是365，则只保存最近365天的文件，删除之前的旧文件。注意，删除旧文件是，那些为了归档而创建的目录也会被删除。  -->* <**MaxHistory**>365</**MaxHistory**>  *<!--   当日志文件超过maxFileSize指定的大小是，根据上面提到的%i进行日志文件滚动 注意此处配置SizeBasedTriggeringPolicy是无法实现按文件大小进行滚动的，必须配置timeBasedFileNamingAndTriggeringPolicy  -->* <**timeBasedFileNamingAndTriggeringPolicy class="ch.qos.logback.core.rolling.SizeAndTimeBasedFNATP"**>  <**maxFileSize**>100MB</**maxFileSize**>  </**timeBasedFileNamingAndTriggeringPolicy**>  </**rollingPolicy**>  *<!-- 日志输出格式： -->* <**layout class="ch.qos.logback.classic.PatternLayout"**>  <**pattern**>%d{yyyy-MM-dd HH:mm:ss.SSS} [ %thread ] - [ %-5level ] [ %logger{50} : %line ] - %msg%n</**pattern**>  </**layout**>  </**appender**>  *<!--   logger主要用于存放日志对象，也可以定义日志类型、级别  name：表示匹配的logger类型前缀，也就是包的前半部分  level：要记录的日志级别，包括 TRACE < DEBUG < INFO < WARN < ERROR  additivity：作用在于children-logger是否使用 rootLogger配置的appender进行输出，  false：表示只用当前logger的appender-ref，true：  表示当前logger的appender-ref和rootLogger的appender-ref都有效  -->  <!-- hibernate logger -->* <**logger name="com.atguigu" level="debug"** />  *<!-- Spring framework logger -->* <**logger name="org.springframework" level="debug" additivity="false"**></**logger**>  *<!--   root与logger是父子关系，没有特别定义则默认为root，任何一个类只会和一个logger对应，  要么是定义的logger，要么是root，判断的关键在于找到这个logger，然后判断这个logger的appender和level。   -->* <**root level="info"**>  <**appender-ref ref="stdout"** />  <**appender-ref ref="appLogAppender"** />  </**root**> </**configuration**> |

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。logback.xml则不能使用Profile功能。

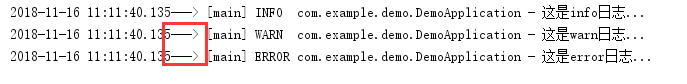
logback-spring.xml将其放到springboot配置文件夹下，内容与上面的logback.xml文件内容的唯一区别如下：

|  |
| --- |
| <**layout class="ch.qos.logback.classic.PatternLayout"**>  *<!-- 如果是开发环境dev就输出日志格式如下-->* <**springProfile name="dev"**>  <**pattern**>%d{yyyy-MM-dd HH:mm:ss.SSS}---> [%thread] %-5level %logger{50} - %msg%n</**pattern**>  </**springProfile**>  *<!-- 如果是开发环境不是dev就输出日志格式如下-->* <**springProfile name="!dev"**>  <**pattern**>%d{yyyy-MM-dd HH:mm:ss.SSS}==== [%thread] %-5level %logger{50} - %msg%n</**pattern**>  </**springProfile**> </**layout**> |

修改application.properties文件

|  |
| --- |
| **spring.profiles.active**=**dev** |

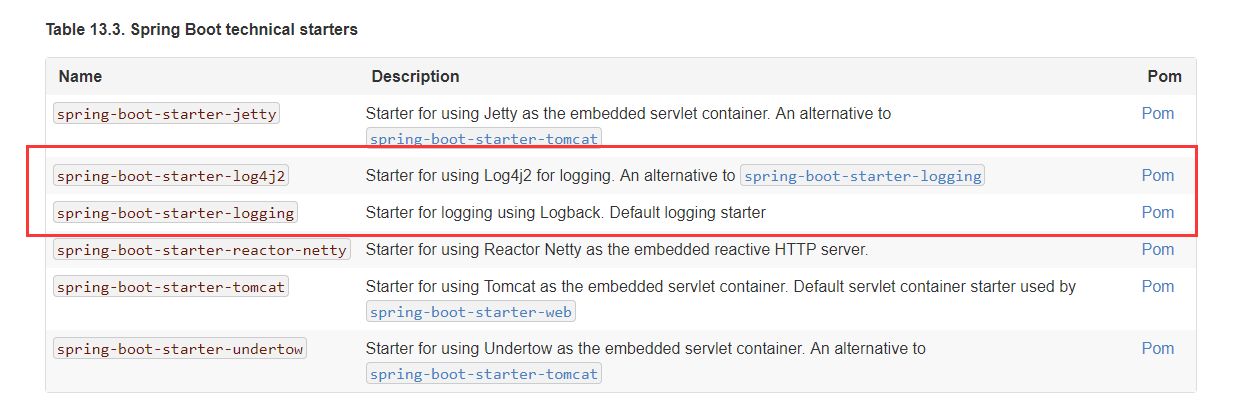
效果：如下说明日志配置文件logback-spring.xml已经生效



1. **切换日志框架**

官方文档地址：

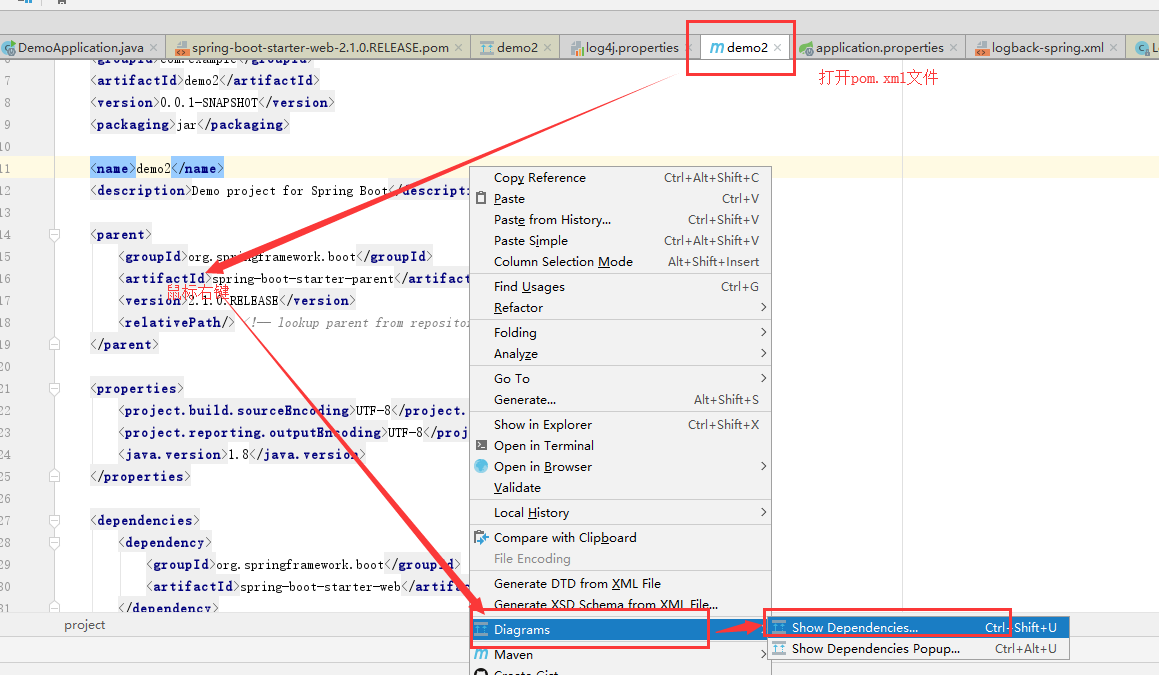
<https://docs.spring.io/spring-boot/docs/2.1.0.RELEASE/reference/html/using-boot-build-systems.html#using-boot-starter>

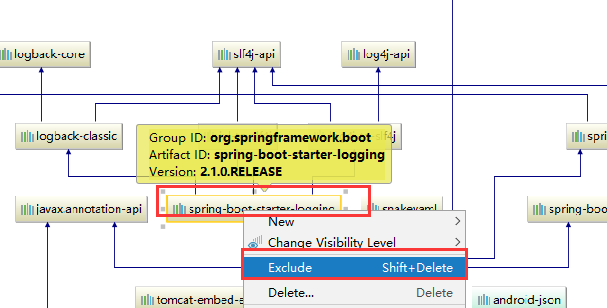


**需求**：把springboot的默认日志框架切换为spring-boot-starter-log4j2

首先先排除掉springboot之前的日志依赖spring-boot-starter-logging

查看依赖关系图：





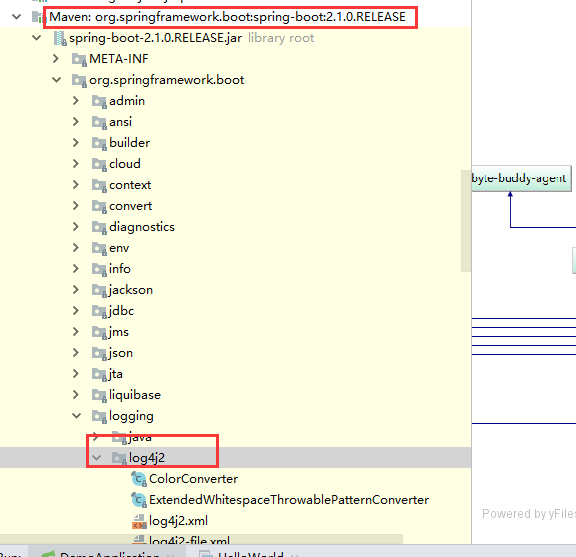
当然我们也可以直接修改pom.xml进行排除

|  |
| --- |
| <**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**> |

添加spring-boot-starter-log4j2的依赖

|  |
| --- |
| <**dependency**>  <**groupId**>org.springframework.boot</**groupId**>  <**artifactId**>spring-boot-starter-log4j2</**artifactId**> </**dependency**> |

此刻点击运行服务，就行了，但是因为springboot默认还是支持log4j2的配置，因此运行和之前没什么区别，我们也可以编写自己的log4j2的配置文件，将其放到配置文件夹中。



1. **Web开发**

使用SpringBoot开发步骤；

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

3）、自己编写业务代码；

1. **静态资源的映射处理**

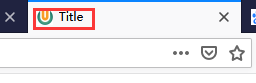
**WebMvcAutoConfiguration**

首先我们可以查看WebMvcAutoConfiguration这个类

|  |
| --- |
| **public void** addResourceHandlers(ResourceHandlerRegistry registry) {  **if** (!**this**.resourceProperties.isAddMappings()) {  logger.debug(**"Default resource handling disabled"**);  } **else** {  **if** (!registry.hasMappingForPattern(**"/webjars/\*\*"**)) {  ResourceHandlerRegistration registration = registry.addResourceHandler(**new** String[]{**"/webjars/\*\*"**}).addResourceLocations(**new** String[]{**"classpath:/META-INF/resources/webjars/"**});  **this**.configureResourceCaching(registration);  **this**.customizeResourceHandlerRegistration(registration);  }   String staticPathPattern = **this**.webFluxProperties.getStaticPathPattern();  //静态资源文件夹映射  **if** (!registry.hasMappingForPattern(staticPathPattern)) {  ResourceHandlerRegistration registration = registry.addResourceHandler(**new** String[]{staticPathPattern}).addResourceLocations(**this**.resourceProperties.getStaticLocations());  **this**.configureResourceCaching(registration);  **this**.customizeResourceHandlerRegistration(registration);  }   } } |

此类会去扫描webjars文件夹下的静态资源。

**更改项目title图片**

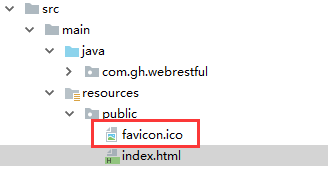


源码：

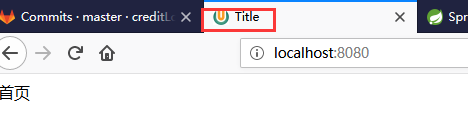
|  |
| --- |
| @Bean **public** SimpleUrlHandlerMapping faviconHandlerMapping() {  SimpleUrlHandlerMapping mapping = **new** SimpleUrlHandlerMapping();  mapping.setOrder(-2147483647);  mapping.setUrlMap(Collections.singletonMap(**"\*\*/favicon.ico"**, **this**.faviconRequestHandler()));  **return** mapping; } |

所有的 \*\*/favicon.ico 都是在静态资源文件下找；

因此我们将自己制作的图片favicon.ico放到文件夹public下

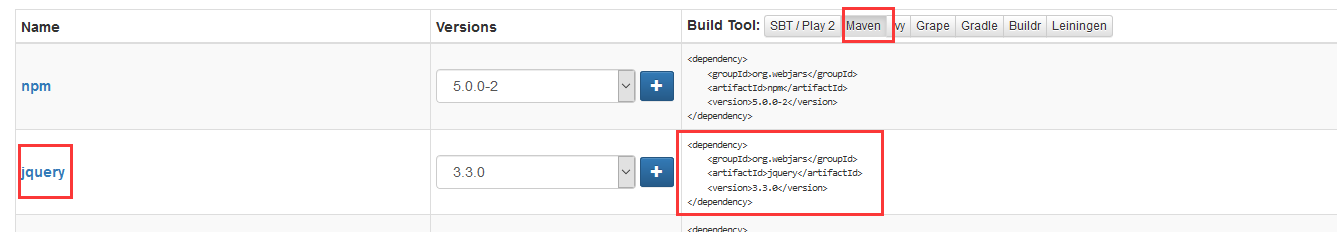


效果：此时发现title图片已经变成我们自己的了



**webjars**

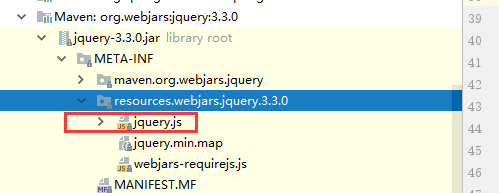
网址：<https://www.webjars.org/>



将此依赖配置到我们的项目的pom.xml文件中

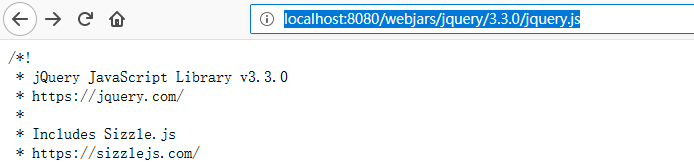
|  |
| --- |
| <**dependency**>  <**groupId**>org.webjars</**groupId**>  <**artifactId**>jquery</**artifactId**>  <**version**>3.3.0</**version**> </**dependency**> |

我们会看见依赖的jar包中有jQuery相关js



此时启动服务,我们通过下面网址就能访问到jQuery.js静态文件，说明webjars包下的静态文件会被springboot扫描到。

<http://localhost:8080/webjars/jquery/3.3.0/jquery.js>

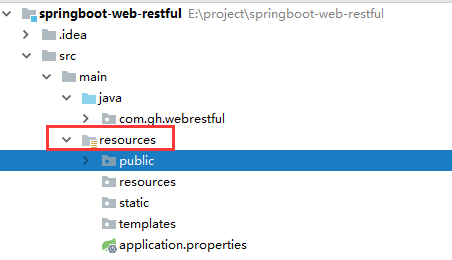


**ResourceProperties**

相关静态文件映射配置在此类中ResourceProperties可以找到，可以在idea中搜索此类，双击shift

"/\*\*" 访问当前项目的任何资源，都去（静态资源的文件夹）找映射

|  |
| --- |
| "classpath:/META‐INF/resources/",  "classpath:/resources/",  "classpath:/static/",  "classpath:/public/"  "/"：当前项目的根路径 |



也就是放到上图这个位置

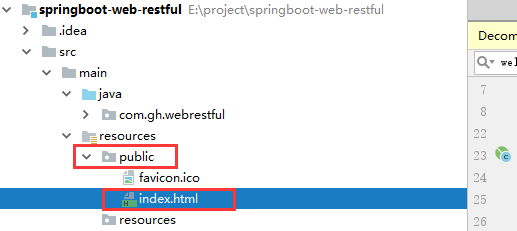
**WelcomePageHandlerMapping**

设置首页欢迎页，源码如下：

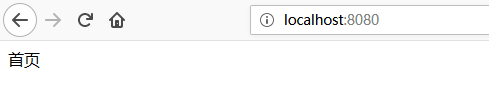
|  |
| --- |
| @Bean **public** WelcomePageHandlerMapping welcomePageHandlerMapping(ApplicationContext applicationContext) {  **return new** WelcomePageHandlerMapping(**new** TemplateAvailabilityProviders(applicationContext), applicationContext, **this**.getWelcomePage(), **this**.mvcProperties.getStaticPathPattern()); } |

静态资源文件夹下的所有index.html页面；被"/\*\*"映射；因此只要放到springboot项目能识别的静态资源文件夹下，都行

示例：如图所示我们放置的位置

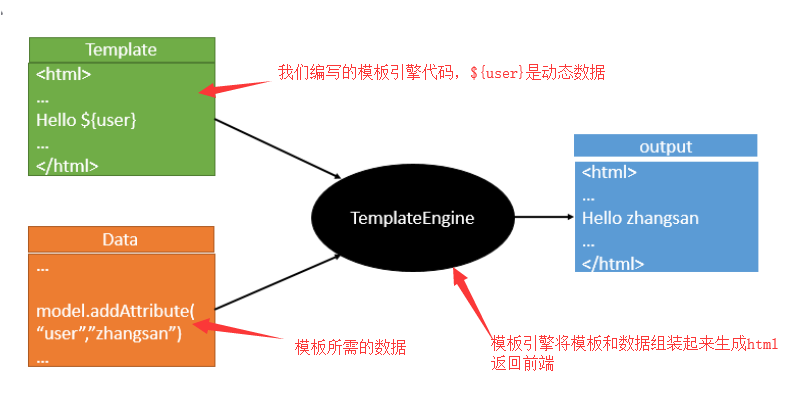


启动服务，效果：



1. **模板引擎**

常用的模板引擎JSP、Velocity、Freemarker、Thymeleaf



SpringBoot推荐的Thymeleaf；语法更简单，功能更强大；

**引入Thymeleaf**

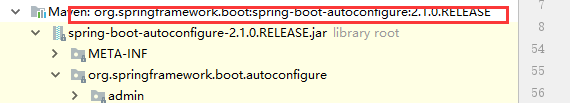
在pom.xml文件添加依赖

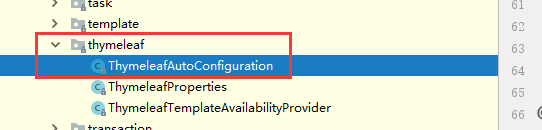
|  |
| --- |
| <**dependency**>  <**groupId**>org.springframework.boot</**groupId**>  <**artifactId**>spring-boot-starter-thymeleaf</**artifactId**> </**dependency**> |

**Thymeleaf语法**

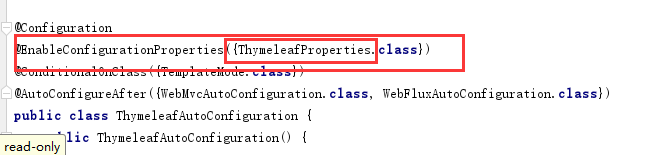
**Thymeleaf的使用**

找到项目依赖maven下的

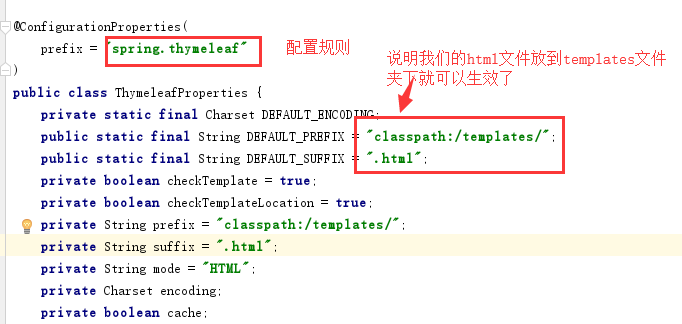




打开如上自动配置类：

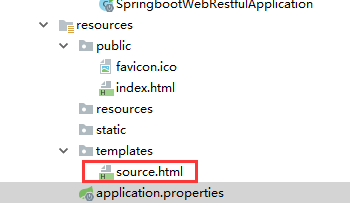


打开类ThymeleafProperties



**简单demo**

在templates文件夹下新建source.html文件



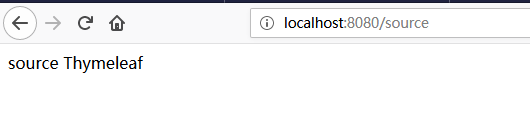
|  |
| --- |
| <!DOCTYPE **html**> <**head**>  <**meta charset="UTF-8"**>  <**title**>source</**title**> </**head**> <**body**> <**div**>source Thymeleaf</**div**> </**body**> </**html**> |

新建请求代码：

|  |
| --- |
| @Controller **public class** HelloWorldController {   @RequestMapping(**"/source"**)  **public** String source(){  **return "source"**;  } } |

此处要注意注解，用错是不行的

启动服务，访问<http://localhost:8080/source，就可以发现已经能访问source.html>页面的内容了。



**禁用thymeleaf缓存**

在application.properties文件中添加如下配置

|  |
| --- |
| *#禁用thymeleaf缓存* **spring.thymeleaf.cache**=**false** |

此时用idea修改页面内容后发现还是不生效，此时按ctrl+f9重新编译再刷新就发现生效了。

**获取数据在页面展示**

比如我们有时后台要查询数据出来，然后在页面展示

后台代码

|  |
| --- |
| @RequestMapping(**"/source"**) **public** String source(Map<String,Object> map){  map.put(**"hello"**,**"你好"**);  **return "source"**; } |

注：此处的map不能在方法体内声明，必须放在方法上，这样页面才能获取到hello里面的值，具体原因不知。

source.html内容

|  |
| --- |
| <!DOCTYPE **html**> *<!-- 导入thymeleaf的名称空间-->* <**html lang="en" xmlns:th="http://www.thymeleaf.org"**> <**head**>  <**meta charset="UTF-8"**>  <**title**>source</**title**> </**head**> <**body**> <**h1**>source Thymeleaf1</**h1**> *<!--th:text 将div里面的文本内容设置为hello的值 -->* <**div th:text="${hello}"**>这是显示欢迎信息</**div**> </**body**> </**html**> |

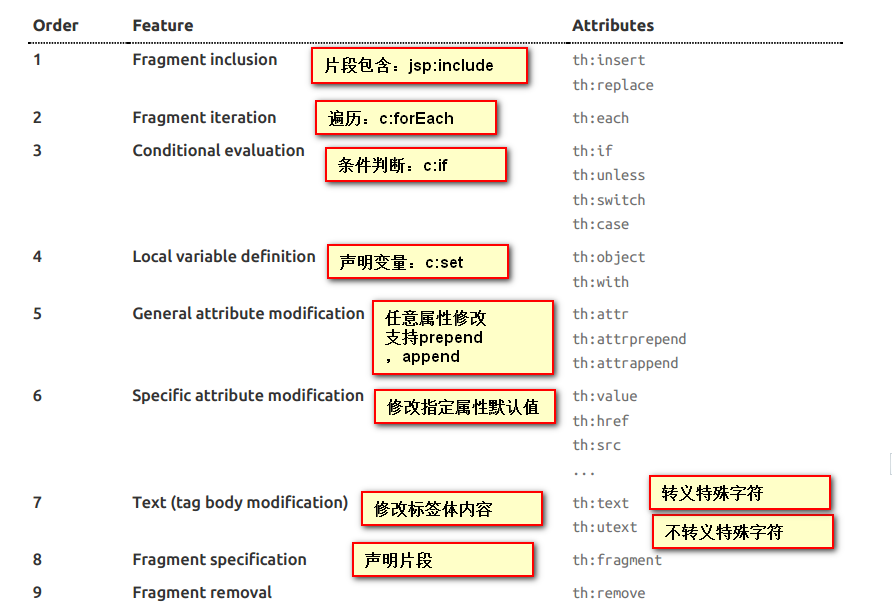
**语法规则**

1. th:text；改变当前元素里面的文本内容；

th: 任意html属性；th都可以用来替换原生属性的值，比如div的id值、class、style、title等

具体可以查看官方文档地址：

<https://www.thymeleaf.org/doc/tutorials/3.0/usingthymeleaf.html#attribute-precedence>



2）

1. **SpringMVC自动配置**
2. **Spring MVC auto-configuration**

Spring Boot 自动配置好了SpringMVC

以下是SpringBoot对SpringMVC的默认配置:（WebMvcAutoConfiguration）

* Inclusion of ContentNegotiatingViewResolver and BeanNameViewResolver beans.

|--自动配置了ViewResolver（视图解析器：根据方法的返回值得到视图对象（View），视图对象决定如何渲染（转发？重定向？））

|--ContentNegotiatingViewResolver：组合所有的视图解析器的；

|--如何定制：我们可以自己给容器中添加一个视图解析器；自动的将其组合进来；

* Support for serving static resources, including support for WebJars (see below).静态资源文件夹路径,webjars
* Static index.html support. 静态首页访问
* Custom Favicon support (see below). favicon.ico
* 自动注册了 of Converter , GenericConverter , Formatter beans.

|--Converter：转换器； public String hello(User user)：类型转换使用Converter

|--Formatter 格式化器； 2017.12.17===Date；



自己添加的格式化器转换器，我们只需要放在容器中即可

* Support for HttpMessageConverters (see below).

\* HttpMessageConverter：SpringMVC用来转换Http请求和响应的；User---Json；

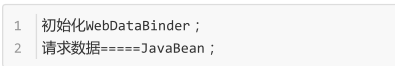
\* HttpMessageConverters 是从容器中确定；获取所有的HttpMessageConverter；

自己给容器中添加HttpMessageConverter，只需要将自己的组件注册容器中

（@Bean,@Component）

* Automatic registration of MessageCodesResolver (see below).定义错误代码生成规则
* Automatic use of a ConfigurableWebBindingInitializer bean (see below).

我们可以配置一个ConfigurableWebBindingInitializer来替换默认的；（添加到容器）



**org.springframework.boot.autoconfigure.web：web的所有自动场景**；

If you want to keep Spring Boot MVC features and you want to add additional [MVC configuration](https://docs.spring.io/spring/docs/5.1.2.RELEASE/spring-framework-reference/web.html#mvc) (interceptors, formatters, view controllers, and other features), you can add your own @Configuration class of type WebMvcConfigurer but **without** @EnableWebMvc. If you wish to provide custom instances of RequestMappingHandlerMapping, RequestMappingHandlerAdapter, or ExceptionHandlerExceptionResolver, you can declare a WebMvcRegistrationsAdapter instance to provide such components.

If you want to take complete control of Spring MVC, you can add your own @Configuration annotated with @EnableWebMvc.

1. **扩展SpringMVC**

编写一个接口restful，请求来时，将source页面返回。此处只是举个例子，我们可以扩展很多功能

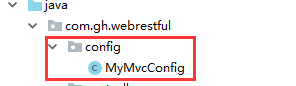
source.html页面内容

|  |
| --- |
| <!DOCTYPE **html**> <**head**>  <**meta charset="UTF-8"**>  <**title**>source</**title**> </**head**> <**body**> <**h1**>source Thymeleaf1</**h1**> </**body**> </**html**> |

编写类MyMvcConfig要实现WebMvcConfigurer接口，不同springboot版本此处的用法不一样，我是2.1.0的版本因此这样用。

|  |
| --- |
| **import** org.springframework.context.annotation.Configuration; **import** org.springframework.web.servlet.config.annotation.ViewControllerRegistry; **import** org.springframework.web.servlet.config.annotation.WebMvcConfigurer;  @Configuration **public class** MyMvcConfig **implements** WebMvcConfigurer{   @Override  **public void** addViewControllers(ViewControllerRegistry registry) {  registry.addViewController(**"/restful"**).setViewName(**"source"**);  } } |

类结构：



启动服务，在页面输入如下地址：<http://localhost:8080/restful>

就可以访问source.html页面了



**原理：**

1）、WebMvcAutoConfiguration是SpringMVC的自动配置类

2）、在做其他自动配置时会导入；

@Import(EnableWebMvcConfiguration.class)



3）、容器中所有的WebMvcConfigurer都会一起起作用；

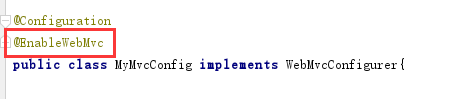
4）、我们的配置类也会被调用；

效果：SpringMVC的自动配置和我们的扩展配置都会起作用；

**全面接管SpringMVC**

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

我们需要在配置类中添加@EnableWebMvc即可；也就是在类MyMvcConfig上加上该注解。



当我们添加该注解后，比如我们放在public文件夹下的首页就不能直接访问了。必须自己手动去配置映射。

**原理：**

为什么@EnableWebMvc自动配置就失效了；

1）@EnableWebMvc的核心

|  |
| --- |
| @Import({DelegatingWebMvcConfiguration.**class**}) **public** @**interface** EnableWebMvc { } |

2）

|  |
| --- |
| @Configuration **public class** DelegatingWebMvcConfiguration **extends** WebMvcConfigurationSupport { |

3）

|  |
| --- |
| @Configuration @ConditionalOnWebApplication(  type = Type.SERVLET ) @ConditionalOnClass({Servlet.**class**, DispatcherServlet.**class**, WebMvcConfigurer.**class**})  //容器中没有这个组件的时候，自动配置类才生效,但是我们自己写了该类的实现，因此容器中会有该组件 @ConditionalOnMissingBean({WebMvcConfigurationSupport.**class**}) @AutoConfigureOrder(-2147483638) @AutoConfigureAfter({DispatcherServletAutoConfiguration.**class**, TaskExecutionAutoConfiguration.**class**, ValidationAutoConfiguration.**class**}) **public class** WebMvcAutoConfiguration { |

4) @EnableWebMvc将WebMvcConfigurationSupport组件导入进来；

5) 导入的WebMvcConfigurationSupport只是SpringMVC最基本的功能；

1. **如何修改SpringBoot的默认配置**

模式：

1）、SpringBoot在自动配置很多组件的时候，先看容器中有没有用户自己配置的（@Bean、@Component）如果有就用用户配置的，如果没有，才自动配置；如果有些组件可以有多个（ViewResolver）将用户配置的和自己默认的组合起来；

2）、在SpringBoot中会有非常多的xxxConfigurer帮助我们进行扩展配置

3）、在SpringBoot中会有很多的xxxCustomizer帮助我们进行定制配置

1. **配置首页**

背景：比如我们在其他文件夹下，public文件夹下也放置了index.html页面，在templates文件夹下也有index.html页面，此时如果不设置的话默认访问的是public文件夹下的index.html页面，此处想访问templates文件夹下的首页则需要手动编写代码：

在我们的controller类中编写代码如下：

|  |
| --- |
| @RequestMapping({**"/"**,**"/index.html"**}) **public** String index(){  **return "index"**; } |

这样就可以了。

1. **国际化**

Springmvc国际化的步骤：

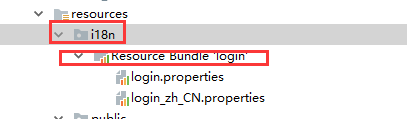
1）、编写国际化配置文件；

2）、使用ResourceBundleMessageSource管理国际化资源文件

3）、在页面使用fmt:message取出国际化内容

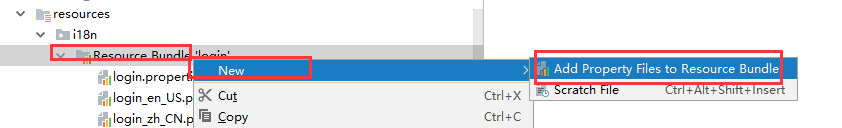
Springboot国际化步骤：

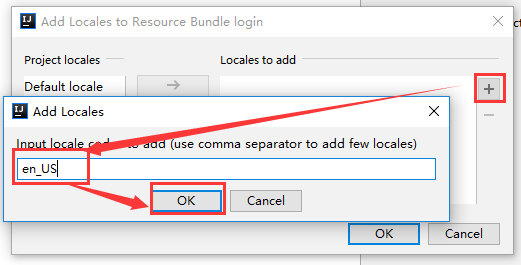
1. 编写国际化配置文件，抽取页面需要显示的国际化消息



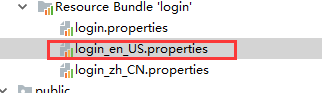
新建文件夹i18n，新建文件login.properties,然后再新建login\_zh\_CN.properties文件，此时idea会自动给我们生成Resource Bundle ‘login’目录，会自动识别到这是个国际化文件。

新建login\_en\_US.properties文件，步骤如下：





点确定，此时就能自动生成该文件



2）编写国际化配置文件内容

随便点击一个properties文件进去





3）SpringBoot自动配置好了管理国际化资源文件的组件；



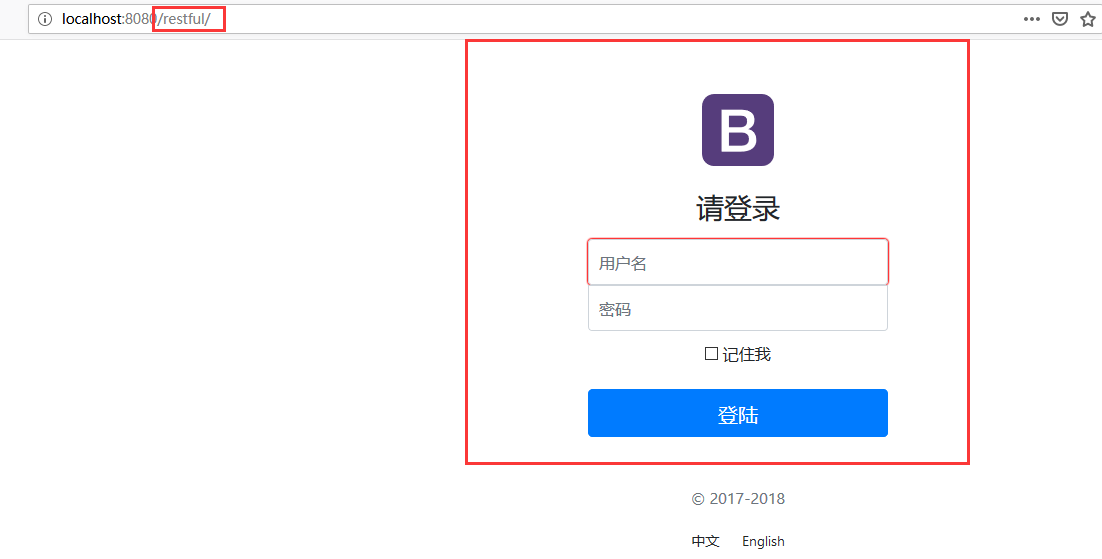
修改application.properties文件：

|  |
| --- |
| *#设置path路径* **server.servlet.context-path**=**/restful** *#设置配置文件的基础名* **spring.messages.basename**=**i18n.login** |

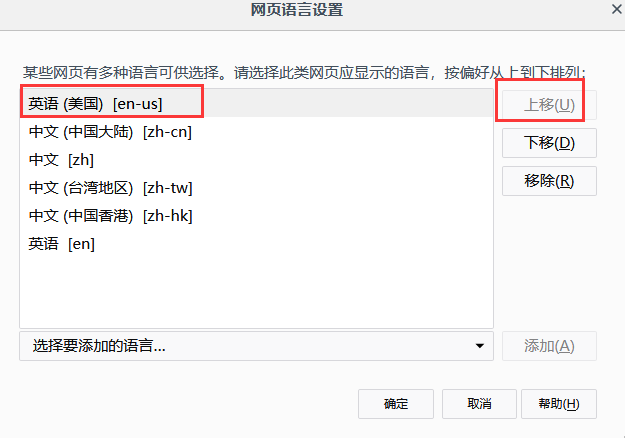
4）去页面获取国际化的值；

|  |
| --- |
| <!DOCTYPE **html**> <**html lang="en" xmlns:th="http://www.thymeleaf.org"**>  <**head**>  <**meta http-equiv="Content-Type" content="text/html; charset=UTF-8"**>  <**meta name="viewport" content="width=device-width, initial-scale=1, shrink-to-fit=no"**>  <**meta name="description" content=""**>  <**meta name="author" content=""**>  <**title**>Signin Template for Bootstrap</**title**>  *<!-- Bootstrap core CSS -->* <**link href="asserts/css/bootstrap.min.css" rel="stylesheet" th:href="@{/webjars/bootstrap/4.1.3/css/bootstrap.css}"**>  *<!-- Custom styles for this template -->* <**link th:href="@{/asserts/css/signin.css}" rel="stylesheet"**>  <**link th:href="@{/webjars/bootstrap/4.1.3/scss/\_badge.scss}" rel="stylesheet"**>  <**link th:href="@{/webjars/bootstrap/4.1.3/scss/\_print.scss}" rel="stylesheet"**>  </**head**>   <**body class="text-center"**>  <**form class="form-signin" action="dashboard.html"**>  <**img class="mb-4" src="asserts/img/bootstrap-solid.svg" alt="" width="72" height="72"**>  <**h1 class="h3 mb-3 font-weight-normal" th:text="#{login.tip}"**>Please sign in</**h1**>  <**label class="sr-only" th:text="#{login.username}"**>Username</**label**>  <**input type="text" class="form-control" placeholder="Username" th:placeholder="#{login.username}" required="" autofocus=""**>  <**label class="sr-only"**>Password</**label**>  <**input type="password" class="form-control" placeholder="Password" th:placeholder="#{login.password}" required=""**>  <**div class="checkbox mb-3"**>  <**label**>  <**input type="checkbox" value="remember-me"**> [[#{login.remember}]]  </**label**>  </**div**>  <**button class="btn btn-lg btn-primary btn-block" type="submit" th:text="#{login.btn}"**>Sign in</**button**>  <**p class="mt-5 mb-3 text-muted"**>© 2017-2018</**p**>  <**a class="btn btn-sm"**>中文</**a**>  <**a class="btn btn-sm"**>English</**a**>  </**form**>   </**body**>  </**html**> |

效果：



服务默认都是根据浏览器的语言信息来显示的，因此我们可以修改浏览器语言设置来让系统展示不同的语言：



此时再刷新页面就变成英文了：



**原理：**

国际化Locale（区域信息对象）；LocaleResolver（获取区域信息对象）；

在springboot的WebMvcAutoConfiguration类中，有如下方法

|  |
| --- |
| **public** LocaleResolver localeResolver() {  **if** (**this**.mvcProperties.getLocaleResolver() == org.springframework.boot.autoconfigure.web.servlet.WebMvcProperties.LocaleResolver.FIXED) {  **return new** FixedLocaleResolver(**this**.mvcProperties.getLocale());  } **else** {  AcceptHeaderLocaleResolver localeResolver = **new** AcceptHeaderLocaleResolver();  localeResolver.setDefaultLocale(**this**.mvcProperties.getLocale());  **return** localeResolver;  } } |

默认的就是根据请求头带来的区域信息获取Locale进行国际化

**点击切换语言信息**

MyLocaleResolver类内容：

|  |
| --- |
| **import** org.springframework.web.servlet.LocaleResolver; **import** org.thymeleaf.util.StringUtils; **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) {  String l = httpServletRequest.getParameter(**"l"**);  *//此处如果我们传了语言信息则用我们传的，如果没有则用浏览器默认的语言信息* Locale locale = Locale.*getDefault*();  **if**(!StringUtils.*isEmpty*(l)){  String[] split = l.split(**"\_"**);  locale = **new** Locale(split[0],split[1]);  }  **return** locale;  }  @Override  **public void** setLocale(HttpServletRequest httpServletRequest, HttpServletResponse httpServletResponse, Locale locale) {  } } |

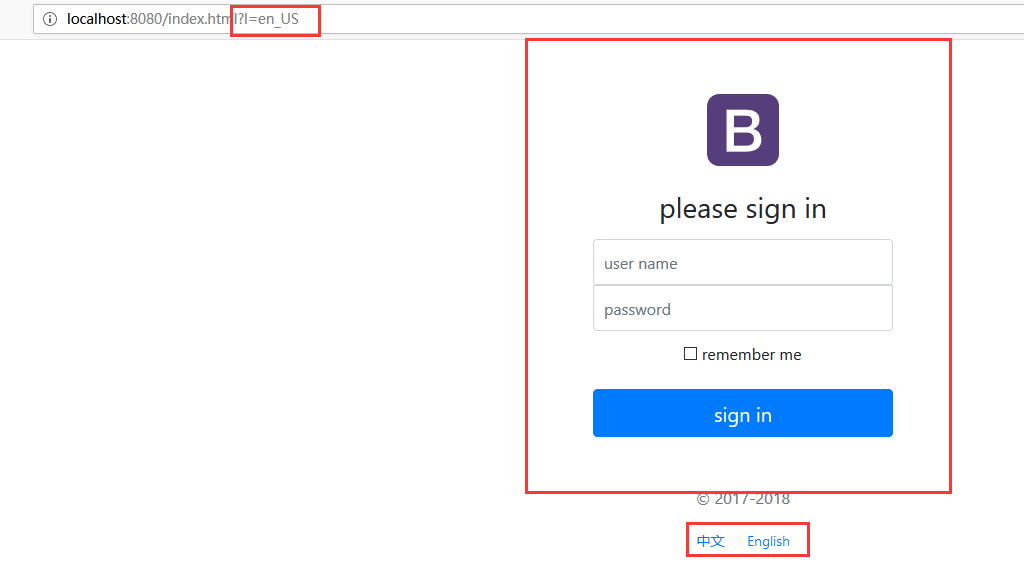
在类MyMvcConfig中添加内容，将我们自己写的localeResolver加载到容器：

|  |
| --- |
| @Bean **public** LocaleResolver localeResolver(){  **return new** MyLocaleResolver(); } |

在login.html内容中添加如下内容：

|  |
| --- |
| <**a class="btn btn-sm" th:href="@{/index.html(l='zh\_CN')}"**>中文</**a**> <**a class="btn btn-sm" th:href="@{/index.html(l='en\_US')}"**>English</**a**> |

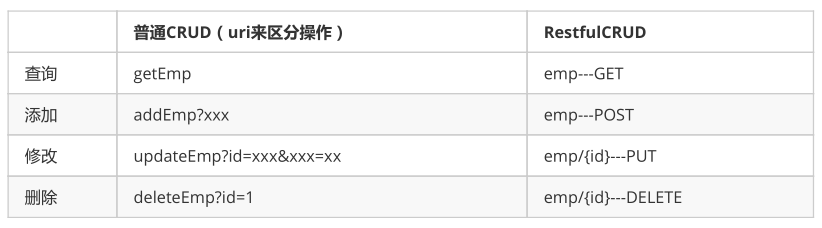
启动服务：



1. **RestfulCRUD风格**

RestfulCRUD：CRUD满足Rest风格；

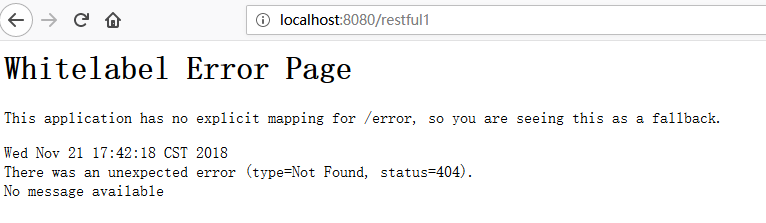
URI： /资源名称/资源标识 HTTP请求方式区分对资源CRUD操作



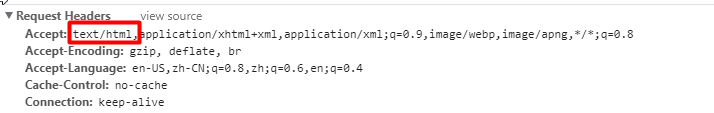
1. **Springboot错误处理机制**

默认效果：

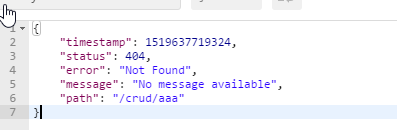
1）、浏览器，返回一个默认的错误页面

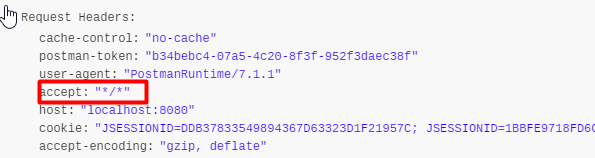


浏览器发送请求的请求头：



2）、如果是其他客户端(postman等)，默认响应一个json数据





也就是说springboot会根据不同的客户端返回不同的结果，springboot是如何做到的？

**1.原理**

可以参照ErrorMvcAutoConfiguration类；错误处理的自动配置；

步骤：

（1）一但系统出现4xx或者5xx之类的错误；ErrorPageCustomizer就会生效（定制错误的响应规则）；就会来到/error请求；就会被BasicErrorController处理。

（2）响应页面；去哪个页面是由DefaultErrorViewResolver解析得到的

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

Springboot给容器中添加了以下组件处理错误

1）、DefaultErrorAttributes：

|  |
| --- |
| @Bean @ConditionalOnMissingBean(  value = {ErrorAttributes.**class**},  search = SearchStrategy.CURRENT ) **public** DefaultErrorAttributes errorAttributes() {  **return new** DefaultErrorAttributes(**this**.serverProperties.getError().isIncludeException()); }  //此处是如果没有配置ErrorAttributes类时则使用默认的DefaultErrorAttributes |

2）、BasicErrorController

|  |
| --- |
| @Bean @ConditionalOnMissingBean(  value = {ErrorController.**class**},  search = SearchStrategy.CURRENT ) **public** BasicErrorController basicErrorController(ErrorAttributes errorAttributes) {  **return new** BasicErrorController(errorAttributes, **this**.serverProperties.getError(), **this**.errorViewResolvers); } |

点击BasicErrorController进入该类

|  |
| --- |
| @Controller @RequestMapping({**"${server.error.path:${error.path:/error}}"**}) **public class** BasicErrorController **extends** AbstractErrorController {  //处理默认的/error请求 |

此类中的两个方法

|  |
| --- |
| @RequestMapping(  produces = {**"text/html"**}  //产生html类型的数据；浏览器发送的请求来到这个方法处理，因为浏览器请求中有如下内容 ) **public** ModelAndView errorHtml(HttpServletRequest request, HttpServletResponse response) {  HttpStatus status = **this**.getStatus(request);  Map<String, Object> model = Collections.unmodifiableMap(**this**.getErrorAttributes(request, **this**.isIncludeStackTrace(request, MediaType.TEXT\_HTML)));  response.setStatus(status.value());  //去哪个页面作为错误页面；包含页面地址和页面内容  ModelAndView modelAndView = **this**.resolveErrorView(request, response, status, model);  **return** modelAndView != **null** ? modelAndView : **new** ModelAndView(**"error"**, model); }  //产生json数据，其他客户端来到这个方法处理； @RequestMapping **public** ResponseEntity<Map<String, Object>> error(HttpServletRequest request) {  Map<String, Object> body = **this**.getErrorAttributes(request, **this**.isIncludeStackTrace(request, MediaType.ALL));  HttpStatus status = **this**.getStatus(request);  **return new** ResponseEntity(body, status); } |

3）、ErrorPageCustomizer

|  |
| --- |
| **public void** registerErrorPages(ErrorPageRegistry errorPageRegistry) {  ErrorPage errorPage = **new** ErrorPage(**this**.dispatcherServletPath.getRelativePath(**this**.properties.getError().getPath()));  errorPageRegistry.addErrorPages(**new** ErrorPage[]{errorPage}); }  //注册错误页面的响应规则 |

点击getPath（）

|  |
| --- |
| @Value(**"${error.path:/error}"**) **private** String path = **"/error"**;  //系统出现错误以后来到error请求进行处理；（类似于web.xml注册的错误页  面规则） |

4）、DefaultErrorViewResolver

|  |
| --- |
| @Bean @ConditionalOnBean({DispatcherServlet.**class**}) @ConditionalOnMissingBean **public** DefaultErrorViewResolver conventionErrorViewResolver() {  **return new** DefaultErrorViewResolver(**this**.applicationContext, **this**.resourceProperties); } |

点进DefaultErrorViewResolver类

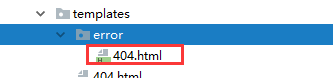
|  |
| --- |
| **private** ModelAndView resolve(String viewName, Map<String, Object> model) {  //默认SpringBoot可以去找到一个页面？  error/404  String errorViewName = **"error/"** + viewName;  //模板引擎可以解析这个页面地址就用模板引擎解析  TemplateAvailabilityProvider provider = **this**.templateAvailabilityProviders.getProvider(errorViewName, **this**.applicationContext);  //模板引擎可用的情况下返回到errorViewName指定的视图地址, 模板引擎不可用，就在静态资源文件夹下找errorViewName对应的页面   error/404.html  **return** provider != **null** ? **new** ModelAndView(errorViewName, model) : **this**.resolveResource(errorViewName, model); } |

**2.定制错误响应**

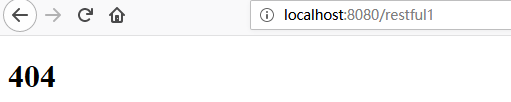
(1)有模板引擎的情况下；error/状态码;

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 对应的页面；

此处我们使用的模板引擎是thymeleaf



效果：



我们也可以使用4xx和5xx作为错误页面的文件名来匹配这种类型的所有错误，精确优先（优先寻找精确的状态码.html）；

**页面能获取的信息：**

timestamp：时间戳

status：状态码

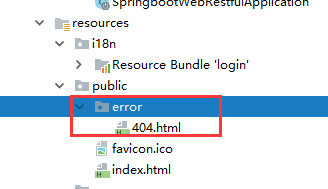
error：错误提示

exception：异常对象

message：异常消息

errors：JSR303数据校验的错误都在这里

2）、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；



3）、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；

**3.定制错误数据**

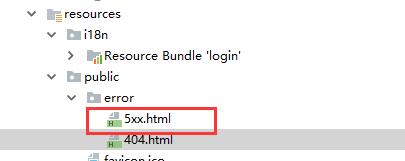
编写自己的exception：MyException

|  |
| --- |
| **public class** MyException **extends** RuntimeException{  **public** MyException() {  **super**(**"运行时报错"**);  } } |

错误处理器：MyExceptionHandler

|  |
| --- |
| @ControllerAdvice **public class** MyExceptionHandler {    @ResponseBody  @ExceptionHandler(MyException.**class**)  **public** Map<String,Object> handleException(Exception e){  Map<String,Object> map = **new** HashMap<>();  map.put(**"code"**,**"user.notexist"**);  map.put(**"message"**,e.getMessage());  **return** map;  } } |

错误页面位置：



5xx.html内容：

|  |
| --- |
| <!DOCTYPE **html**> <**html lang="en" xmlns:th="http://www.thymeleaf.org"**> <**head**>  <**meta charset="UTF-8"**>  <**title**>Title</**title**> </**head**> <**body**> <**div**>5xx</**div**> </**body**> </**html**> |

测试类HelloController类内容：

|  |
| --- |
| @RequestMapping(**"/source"**) **public** String source(Map<String,Object> map) **throws** Exception {  map.put(**"hello"**,**"你好"**);  **if**(map!=**null**)  **throw new** MyException();  **return "source"**; } |

测试：http://localhost:8080/source

发现浏览器和postman等客户端显示的数据都一样，全是json数据。

**4.定制自适应的错误信息**

出现错误以后，会来到/error请求，会被BasicErrorController处理，响应出去可以获取的数据是由getErrorAttributes得到的（是AbstractErrorController（ErrorController）规定的方法）；

**方法1**：完全来编写一个ErrorController的实现类【或者是编写AbstractErrorController的子类】，放在容器中；

**方法2**：页面上能用的数据，或者是json返回能用的数据都是通过errorAttributes.getErrorAttributes得到；

容器中DefaultErrorAttributes.getErrorAttributes()；默认进行数据处理的；

自定义ErrorAttributes

给容器中加入我们自己定义的MyErrorAttributes

|  |
| --- |
| @Component **public class** MyErrorAttributes **extends** DefaultErrorAttributes{   @Override  **public** Map<String, Object> getErrorAttributes(WebRequest webRequest, **boolean** includeStackTrace){  Map<String, Object> map = **super**.getErrorAttributes(webRequest, includeStackTrace);  map.put(**"company"**,**"gh"**);  **return** map;  } } |

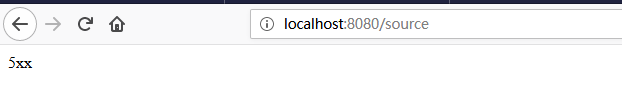
修改MyExceptionHandler类

|  |
| --- |
| @ExceptionHandler(MyException.**class**) **public** String handleException(Exception e, HttpServletRequest request){  Map<String,Object> map = **new** HashMap<>();  *//传入我们自己的错误状态码 4xx 5xx，否则就不会进入定制错误页面的解析流程  /\*\*  \* Integer statusCode = (Integer) request.getAttribute("javax.servlet.error.status\_code");  \*/* request.setAttribute(**"javax.servlet.error.status\_code"**,500);  map.put(**"code"**,**"user.notexist"**);  map.put(**"message"**,e.getMessage());  *//转发到/error* **return "forward:/error"**; } |

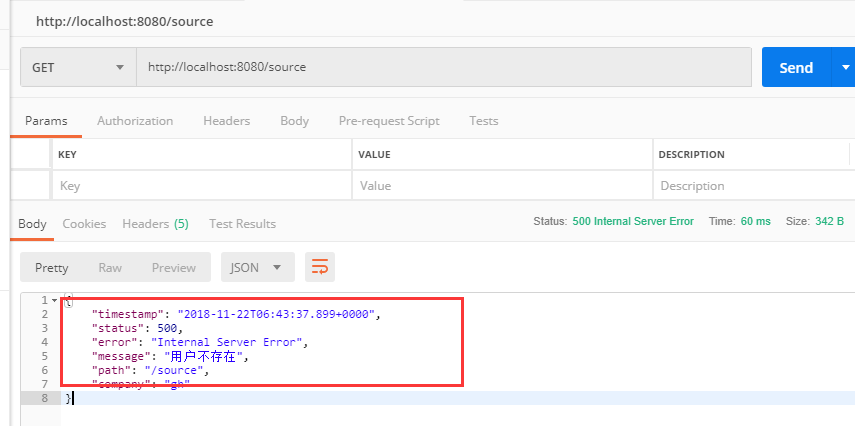
其他与上面保持一致

测试：

浏览器端

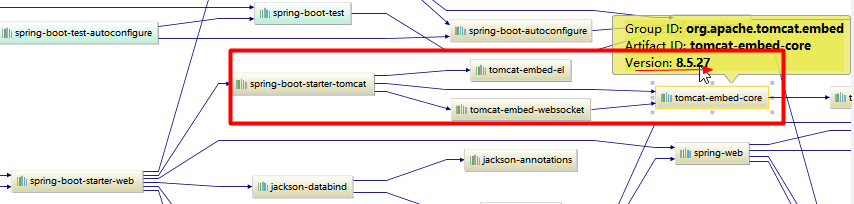


Postman端：



1. **配置嵌入式Servlet容器**

SpringBoot默认使用Tomcat作为嵌入式的Servlet容器；



**1.定制和修改Servlet容器的相关配置**

方法1：修改和server有关的配置（ServerProperties【也是EmbeddedServletContainerCustomizer】）；



方法2：编写一个EmbeddedServletContainerCustomizer：嵌入式的Servlet容器的定制器；来修改Servlet容器的配置



此处要注意springboot2.0及以上已经修改了实现方式，EmbeddedServletContainerCustomizer已经被去掉，使用WebServerFactoryCustomizer来替换。

在MyMvcConfig类中添加如下代码：

|  |
| --- |
| @Bean *//一定要将这个定制器加入到容器中* **public** WebServerFactoryCustomizer<ConfigurableWebServerFactory> embeddedServletContainerCustomizer() {  **return new** WebServerFactoryCustomizer<ConfigurableWebServerFactory>() {  @Override  **public void** customize(ConfigurableWebServerFactory factory) {  factory.setPort(8081);  }  }; } |

1. **注册Servlet三大组件**

**1.注册Servlet**

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。因此采用如下方式实现

新增MyServlet类

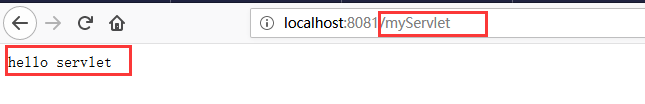
|  |
| --- |
| **public class** MyServlet **extends** HttpServlet {  @Override  **protected void** doGet(HttpServletRequest req, HttpServletResponse resp) **throws** ServletException, IOException {  **this**.doPost(req, resp);  }   @Override  **protected void** doPost(HttpServletRequest req, HttpServletResponse resp) **throws** ServletException, IOException {  resp.getWriter().print(**"hello servlet"**);  } } |

新增MyServerConfig类

|  |
| --- |
| @Configuration **public class** MyServerConfig {   @Bean  **public** ServletRegistrationBean myServlet() {  ServletRegistrationBean registrationBean = **new** ServletRegistrationBean(**new** MyServlet(), **"/myServlet"**);  **return** registrationBean;  } } |

注册ServletRegistrationBean

测试：



**2.注册Filter**

FilterRegistrationBean

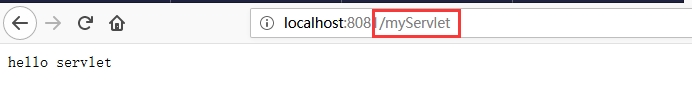
新增MyFilter类

|  |
| --- |
| **public class** MyFilter **implements** Filter {  @Override  **public void** init(FilterConfig filterConfig) **throws** ServletException {  }   @Override  **public void** doFilter(ServletRequest request, ServletResponse response, FilterChain filterChain) **throws** IOException, ServletException {  System.***out***.println(**"MyFilter running"**);  filterChain.doFilter(request,response);  }   @Override  **public void** destroy() {  } } |

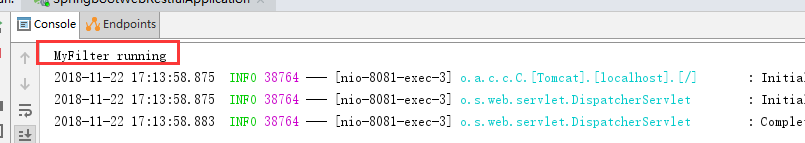
在MyServerConfig类中新增代码：

|  |
| --- |
| @Bean **public** FilterRegistrationBean myFilter() {  FilterRegistrationBean registrationBean = **new** FilterRegistrationBean();  registrationBean.setFilter(**new** MyFilter());  *//配置该filter要拦截的url请求* registrationBean.setUrlPatterns(Arrays.*asList*(**"/myServlet"**));  **return** registrationBean; } |

测试：



后台打印：



**3.注册Listener**

ServletListenerRegistrationBean

新增类MyListener

|  |
| --- |
| **public class** MyListener **implements** ServletContextListener {  @Override  **public void** contextInitialized(ServletContextEvent sce) {  System.***out***.println(**"contextInitialized 服务器启动"**);  }   @Override  **public void** contextDestroyed(ServletContextEvent sce) {  System.***out***.println(**"contextDestroyed 服务器销毁"**);  } } |

在MyServerConfig类中新增代码：

|  |
| --- |
| @Bean **public** ServletListenerRegistrationBean myListener() {  ServletListenerRegistrationBean<MyListener> registrationBean = **new** ServletListenerRegistrationBean<>(**new** MyListener());  **return** registrationBean; } |

测试：



**4.原理**

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。

SpringBoot帮我们自动SpringMVC的时候，自动的注册SpringMVC的前端控制器；DIspatcherServlet；

DispatcherServletAutoConfiguration类中代码：

|  |
| --- |
| **public** DispatcherServletRegistrationBean dispatcherServletRegistration(DispatcherServlet dispatcherServlet) {  DispatcherServletRegistrationBean registration = **new** DispatcherServletRegistrationBean(dispatcherServlet, **this**.webMvcProperties.getServlet().getPath());  //默认拦截： /  所有请求；包静态资源，但是不拦截jsp请求；   /\*会拦截jsp  //可以通过server.servletPath来修改SpringMVC前端控制器默认拦截的请求路径  registration.setName(**"dispatcherServlet"**);  registration.setLoadOnStartup(**this**.webMvcProperties.getServlet().getLoadOnStartup());  **if** (**this**.multipartConfig != **null**) {  registration.setMultipartConfig(**this**.multipartConfig);  }   **return** registration; } |

1. **springboot的servlet容器**

**1. 修改springboot默认的servlet容器**

如果我们不想用springboot的默认容器Tomcat，我们可以修改为其他的servlet容器，比如jetty或者Undertow

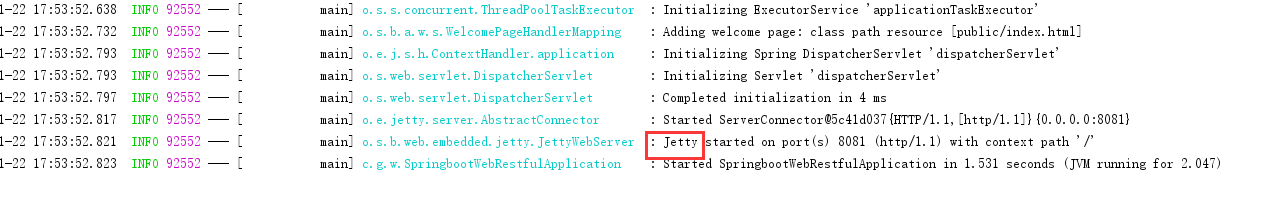
首先必须剔除Tomcat的依赖

|  |
| --- |
| <**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**> |

添加jetty依赖

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

启动服务：



使用Undertow

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

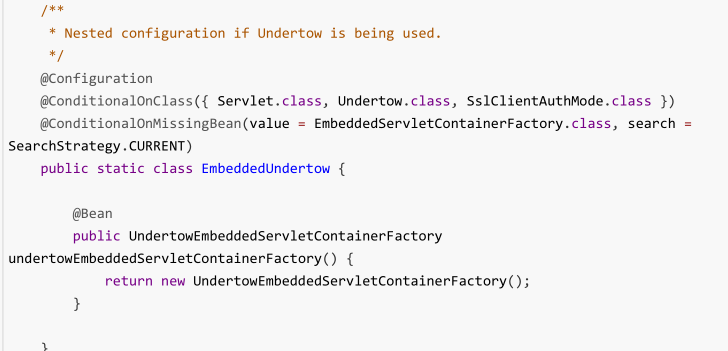
**2.Servlet容器自动配置原理**

Springboot版本不同实现的原理也不一样，特别是2.0后，实现方式大改

此处说的是springboot2.0前的版本

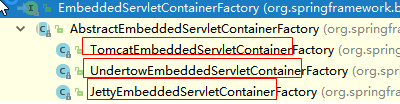
EmbeddedServletContainerAutoConfiguration类：入式的Servlet容器自动配置



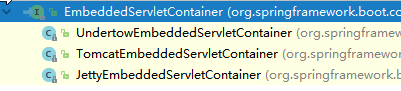


1）EmbeddedServletContainerFactory（嵌入式Servlet容器工厂）

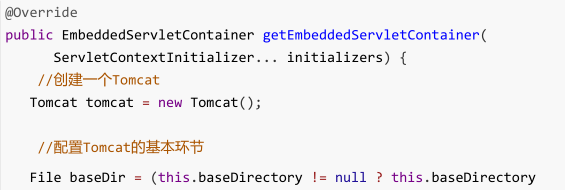


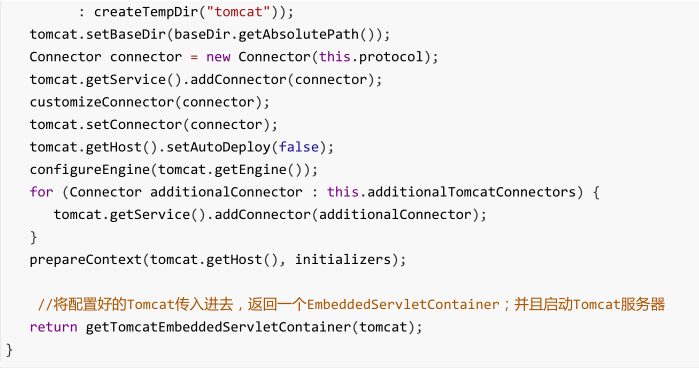


2）EmbeddedServletContainer：（嵌入式的Servlet容器）



3）以TomcatEmbeddedServletContainerFactory为例





4）我们对嵌入式容器的配置修改是怎么生效？

ServerProperties、EmbeddedServletContainerCustomizer

EmbeddedServletContainerCustomizer：定制器帮我们修改了Servlet容器的配置

怎么修改的原理？

5）容器中导入了

EmbeddedServletContainerCustomizerBeanPostProcessor





步骤：

* SpringBoot根据导入的依赖情况，给容器中添加相应的

EmbeddedServletContainerFactory【TomcatEmbeddedServletContainerFactory】

* 容器中某个组件要创建对象就会惊动后置处理器；

EmbeddedServletContainerCustomizerBeanPostProcessor；只要是嵌入式的Servlet容器工厂，后置处理器就工作；

* 后置处理器，从容器中获取所有的EmbeddedServletContainerCustomizer，调用定制器的定制方法

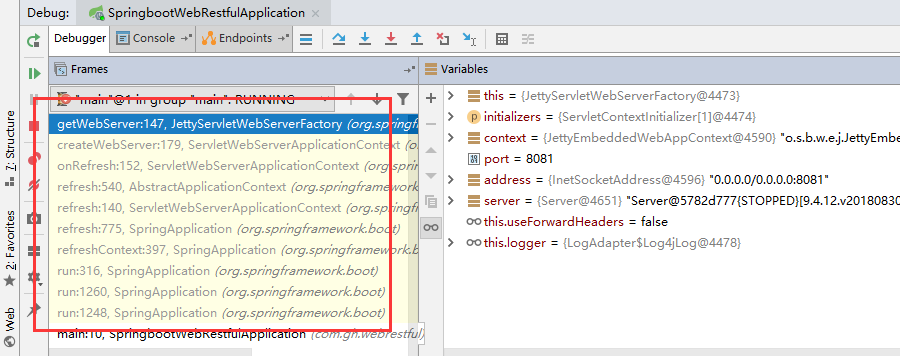
Springboot2.0后的版本请参考网址：

<http://www2014.aspxhtml.com/post-7238>

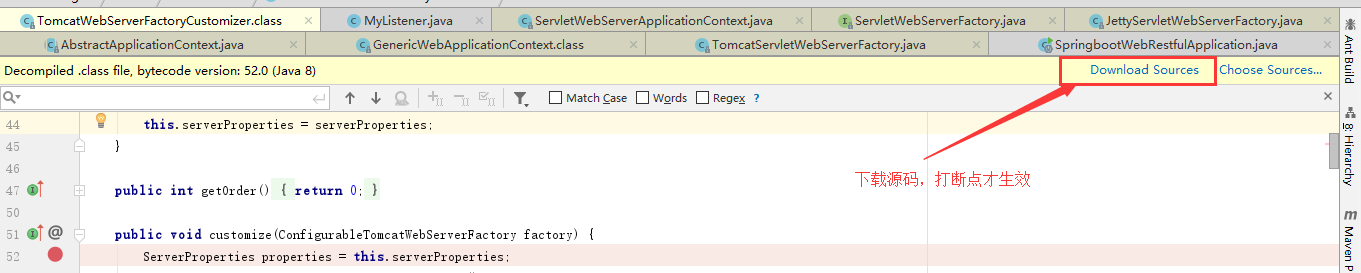
**3.嵌入式Servlet容器启动原理**

什么时候创建嵌入式的Servlet容器工厂？什么时候获取嵌入式的Servlet容器并启动Tomcat?

获取嵌入式的Servlet容器工厂：



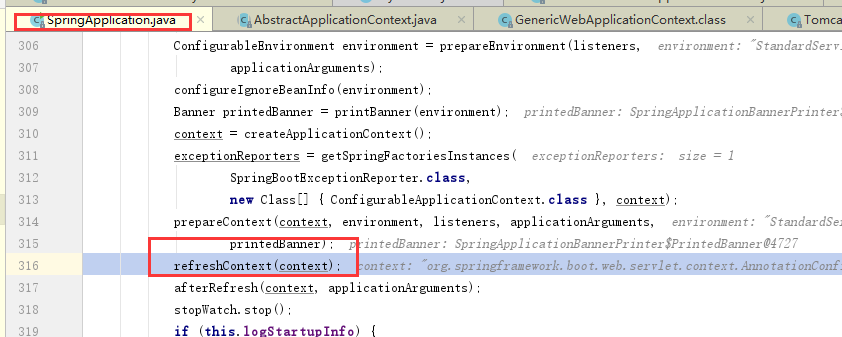
Idea启动服务开启调试模式，如果是jar包的class添加断点，我们需要下载源码，这样打断点才生效，方法如下：



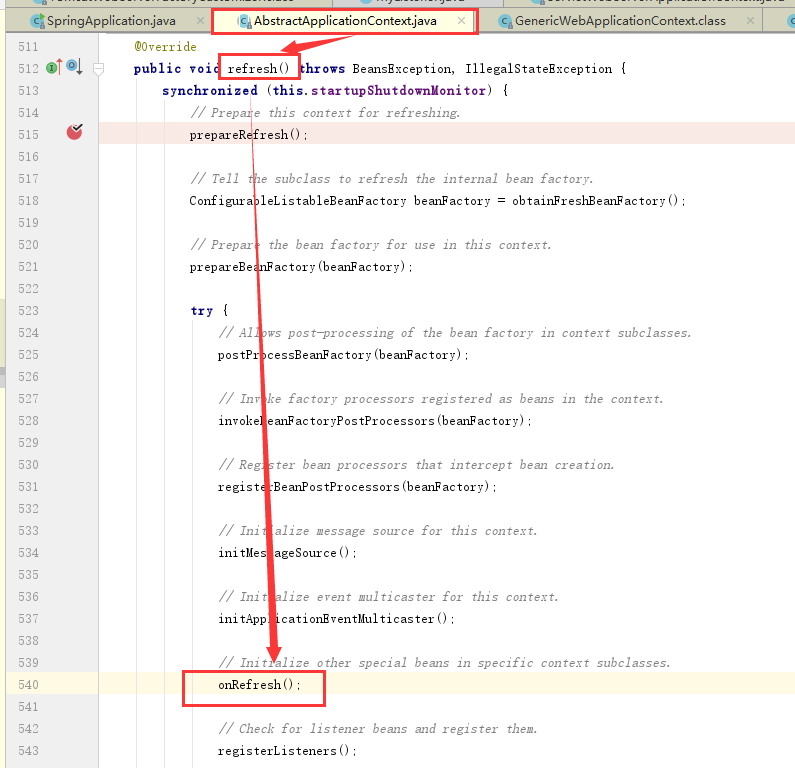
然后一步一步分析springboot的堆栈信息

此处我们使用的是jetty服务器，因此可以找到jetty的factory类，每种服务器都有个factory类，springboot从2.0版本内置了四种，Tomcat，jetty，netty，undertow。此处只说2.0版本以上的分析过程，其他版本分析过程是一致的，不加多述。

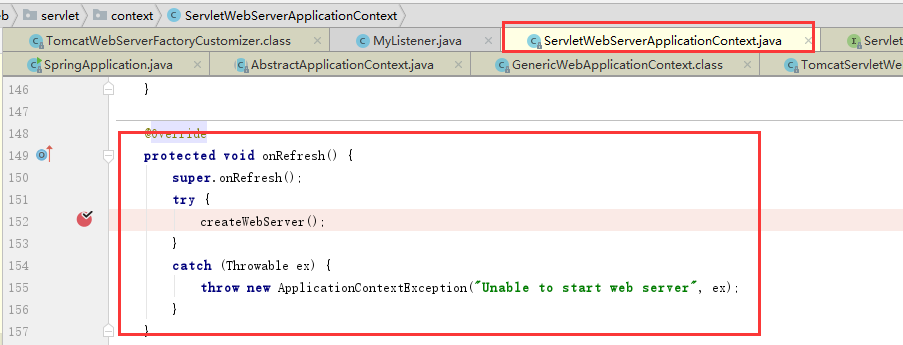
1）SpringBoot应用启动运行run方法

2）refreshContext(context);SpringBoot刷新IOC容器【创建IOC容器对象，并初始化容器，创建容器中的每一个组件】；   


3）refresh(context);刷新刚才创建好的ioc容器；



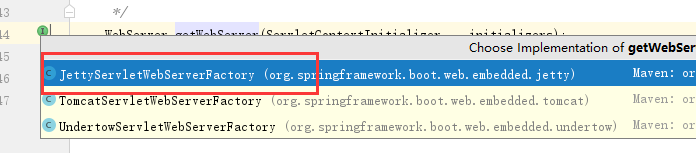
4）onRefresh(); web的ioc容器ServletWebServerApplicationContext类重写了onRefresh方法



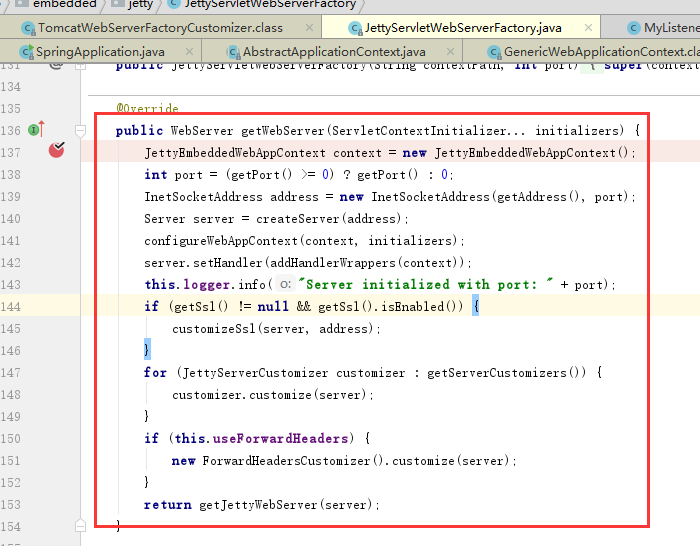
5）webioc容器会创建嵌入式的Servlet容器



6）ServletWebServerFactory类的实现类实现了getWebServer方法，此处不同的服务器实现不同的factory类，比如jetty的实现类是JettyServletWebServerFactory



7）通过工厂类创建一个server，JettyServletWebServerFactory类



8）创建server完成后，会调用如下进行server的启动



**4.使用外置servlet容器**

嵌入式Servlet容器：应用打成可执行的jar

优点：简单、便携；

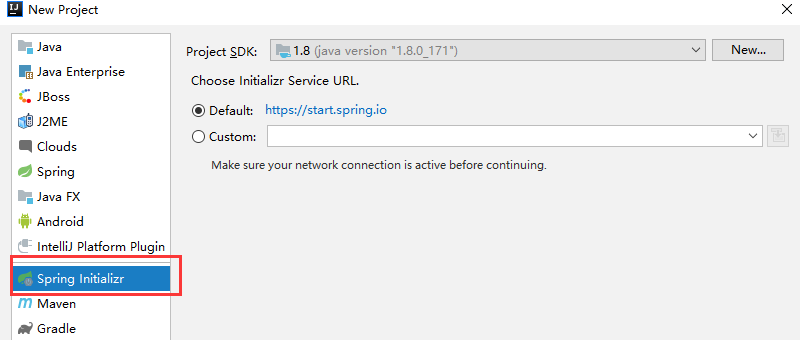
缺点：默认不支持JSP、优化定制比较复杂（使用定制器【ServerProperties、自定义

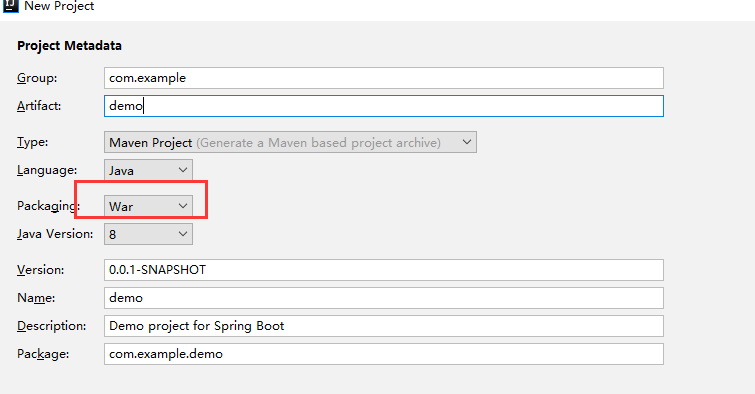
EmbeddedServletContainerCustomizer】，自己编写嵌入式Servlet容器的创建工厂

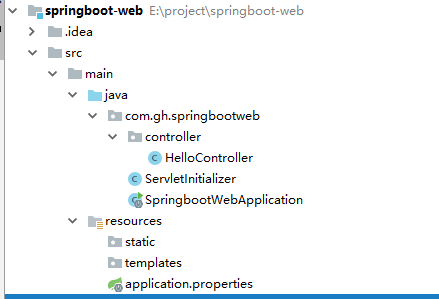
【EmbeddedServletContainerFactory】）；

外置的Servlet容器：外面安装Tomcat---应用war包的方式打包；

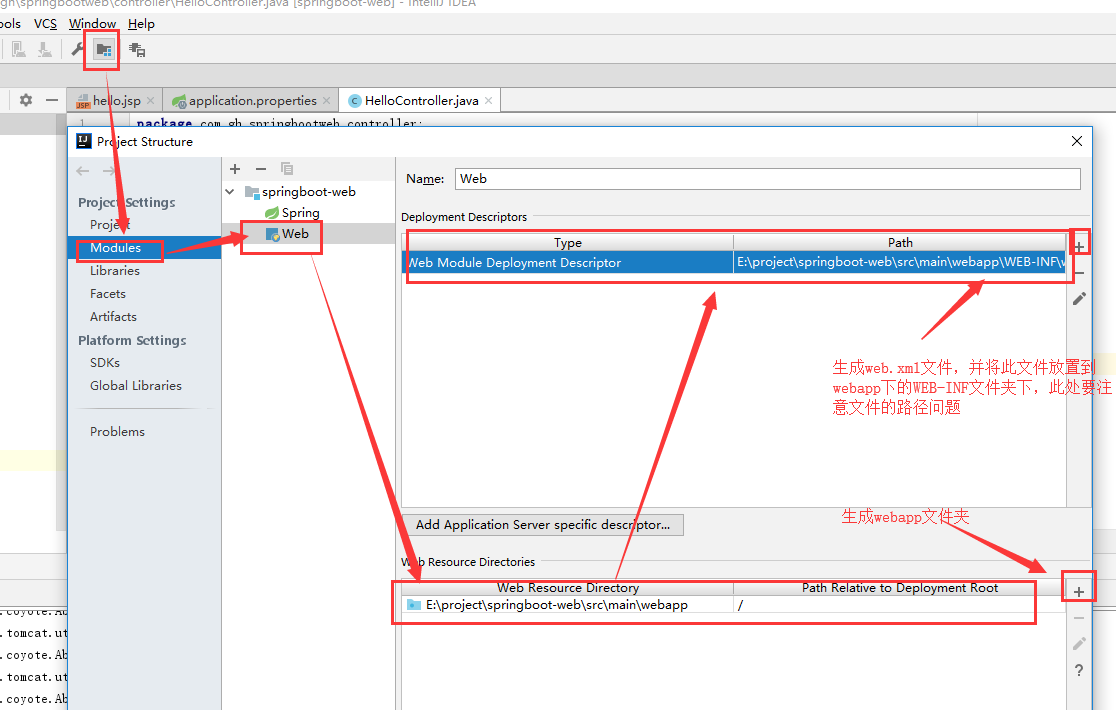
步骤1：







此时生成的项目结构是不包含webapp文件夹的，还必须有如下操作



配置Tomcat，步骤省略

如果是通过工具的话就没有下面这步，如果是手动创建则需要在pom.xml文件添加内容如下：

|  |
| --- |
| <**dependency**>  <**groupId**>org.springframework.boot</**groupId**>  <**artifactId**>spring-boot-starter-tomcat</**artifactId**>  <**scope**>provided</**scope**> </**dependency**> |

此时一个web项目就创建成功了，并且是使用外部servlet容器的springboot项目。

在application.properties文件中添加如下内容

|  |
| --- |
| *#此时扫描WEB-INF文件夹下的jsp文件* **spring.mvc.view.prefix**=**/WEB-INF/** *#扫描后缀名为jsp的文件* **spring.mvc.view.suffix**=**.jsp** |

新建一个hello.jsp文件，放置到WEB-INF文件夹下

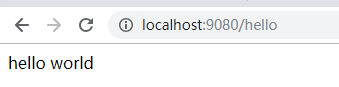
|  |
| --- |
| <%@ **page contentType**="**text/html;charset=UTF-8**" **language**="**java**" %> <**html**> <**head**>  <**title**>Title</**title**> </**head**> <**body**> <**div**>hello world</**div**> </**body**> </**html**> |

新建HelloController文件

|  |
| --- |
| @Controller **public class** HelloController {   @GetMapping(**"/hello"**)  **public** String hello(){  **return "hello"**;  } } |

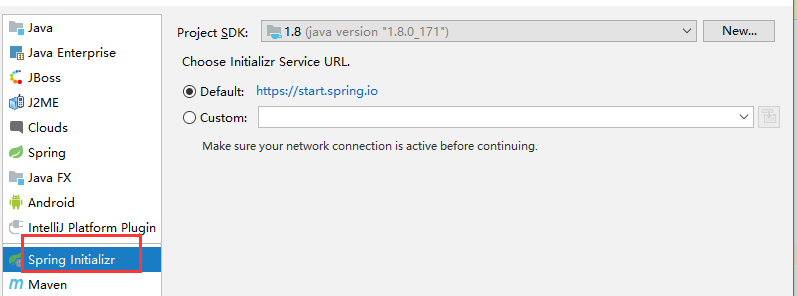
启动Tomcat，访问网址：<http://localhost:9080/hello>

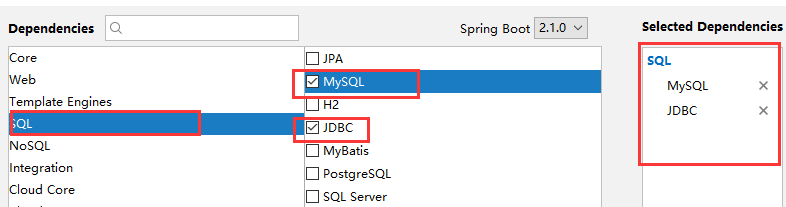
效果：成功访问到hello.jsp的内容

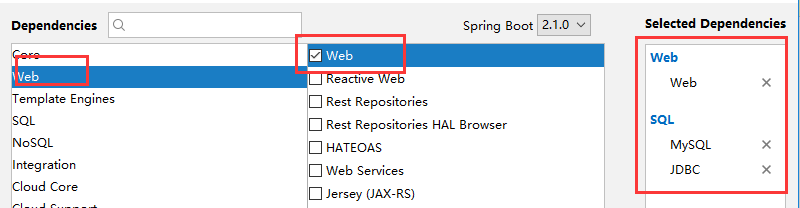


1. **Springboot数据访问**
2. **Jdbc**

创建项目

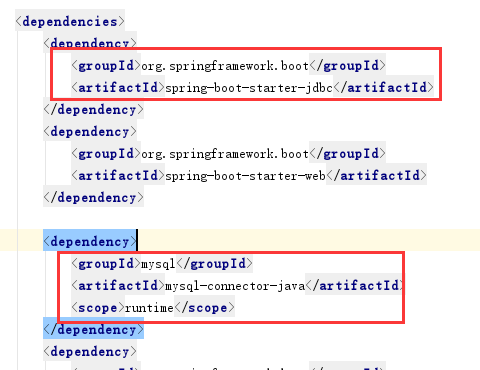






点击确定，这样工程就创建出来了

此时pom.xml文件中就可以看到jdbc和mysql的相关依赖已经被idea工具自动添加上了。



Springboot加载数据库信息在DataSourceProperties类中

配置数据库信息

application.properties

|  |
| --- |
| **spring.datasource.username**=**root spring.datasource.password**=**123456 spring.datasource.url**=**jdbc:mysql://192.168.229.132:3306/test spring.datasource.driver-class-name**=**com.mysql.cj.jdbc.Driver** |

或者application.yml文件

|  |
| --- |
| **spring**:  **datasource**:  **username**: root  **password**: 123456  **url**: jdbc:mysql://192.168.229.132:3306/test  **driver-class-name**: com.mysql.cj.jdbc.Driver  **schema**:  - classpath:department.sql  **initialization-mode**: *always* |

测试类，测试连接数据库是否成功

|  |
| --- |
| @RunWith(SpringRunner.**class**) @SpringBootTest **public class** SpringbootJdbcApplicationTests {  @Autowired  DataSource **dataSource**;  @Test  **public void** contextLoads() **throws** Exception{  System.***out***.println(**dataSource**.getClass());  Connection connection = **dataSource**.getConnection();  System.***out***.println(connection);  connection.close();  }  } |

**自动配置原理**

org.springframework.boot.autoconfigure.jdbc：

1）、参考DataSourceConfiguration，根据配置创建数据源，默认使用Tomcat连接池；可以使用spring.datasource.type指定自定义的数据源类型；

2）、SpringBoot默认可以支持；

|  |
| --- |
| org.apache.tomcat.jdbc.pool.DataSource、HikariDataSource、BasicDataSource、 |

3）、自定义数据源类型

此方法在DataSourceConfiguration类中

|  |
| --- |
| @ConditionalOnMissingBean({DataSource.**class**}) @ConditionalOnProperty(  name = {**"spring.datasource.type"**} ) **static class** Generic {  Generic() {  }  @Bean  **public** DataSource dataSource(DataSourceProperties properties) {  **return** properties.initializeDataSourceBuilder().build();  } } |

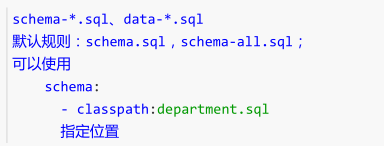
4）、DataSourceInitializer：ApplicationListener；

作用：

\* runSchemaScripts();运行建表语句；

\* runDataScripts();运行插入数据的sql语句；

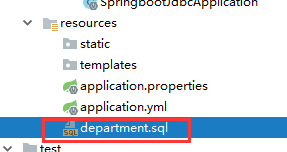
默认只需要将文件命名为：



5）、操作数据库：自动配置了JdbcTemplate操作数据库

注意：springboot2.0后，方式不太一样，如果想springboot自动加载SQL需要在配置文件添加如下信息：

|  |
| --- |
| **spring.datasource.schema**=**classpath:department.sql spring.datasource.initialization-mode**=*always* |



**集成druid数据源监控**

pom.xml添加依赖

|  |
| --- |
| *<!--引入druid数据源--> <!-- https://mvnrepository.com/artifact/com.alibaba/druid -->* <**dependency**>  <**groupId**>com.alibaba</**groupId**>  <**artifactId**>druid</**artifactId**>  <**version**>1.1.8</**version**> </**dependency**> |

application.properties

|  |
| --- |
| **spring**:  **datasource**:  **username**: root  **password**: 123456  **url**: jdbc:mysql://localhost:3306/test  **driver-class-name**: com.mysql.jdbc.Driver  **type**: com.alibaba.druid.pool.DruidDataSource  # 如下是druid特有的配置，springboot不能自动加载，需要我们手动加载  **initialSize**: 5  **minIdle**: 5  **maxActive**: 20  **maxWait**: 60000  **timeBetweenEvictionRunsMillis**: 60000  **minEvictableIdleTimeMillis**: 300000  **validationQuery**: SELECT 1 FROM DUAL  **testWhileIdle**: true  **testOnBorrow**: false  **testOnReturn**: false  **poolPreparedStatements**: true  *# 配置监控统计拦截的filters，去掉后监控界面sql无法统计，'wall'用于防火墙* **filters**: stat,wall  **maxPoolPreparedStatementPerConnectionSize**: 20  **useGlobalDataSourceStat**: true  **connectionProperties**: druid.stat.mergeSql=true;druid.stat.slowSqlMillis=500 |

DruidConfig类

|  |
| --- |
| @Configuration **public class** DruidConfig {  @ConfigurationProperties(prefix = **"spring.datasource"**)  @Bean  **public** DataSource druid(){  **return new** DruidDataSource();  }  *//配置Druid的监控  //1、配置一个管理后台的Servlet* @Bean  **public** ServletRegistrationBean statViewServlet(){  ServletRegistrationBean bean = **new** ServletRegistrationBean(**new** StatViewServlet(), **"/druid/\*"**);  Map<String,String> initParams = **new** HashMap<>();  initParams.put(**"loginUsername"**,**"admin"**);  initParams.put(**"loginPassword"**,**"123456"**);  initParams.put(**"allow"**,**""**);*//默认就是允许所有访问* initParams.put(**"deny"**,**"192.168.15.21"**);  bean.setInitParameters(initParams);  **return** bean;  }  *//2、配置一个web监控的filter* @Bean  **public** FilterRegistrationBean webStatFilter(){  FilterRegistrationBean bean = **new** FilterRegistrationBean();  bean.setFilter(**new** WebStatFilter());  Map<String,String> initParams = **new** HashMap<>();  initParams.put(**"exclusions"**,**"\*.js,\*.css,/druid/\*"**);  bean.setInitParameters(initParams);  bean.setUrlPatterns(Arrays.*asList*(**"/\*"**));  **return** bean;  } } |

注：springboot的不同版本，druid配置就会不同，比如filters属性值：2.0后不能配置log4j，因为springboot已经默认使用logback框架。

登录druid



1. **集成mybatis**

1）、新建工程，此时主要sql里面要选择mybatis

2）、其他的跟集成druid是一样的

3）、在集成druid基础上，我们需要给数据库建表，创建JavaBean

4）、分为注解版和配置文件版

**注解版**

DepartmentMapper接口

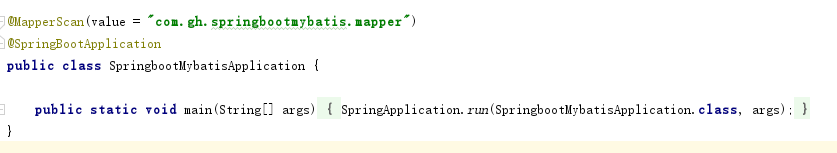
|  |
| --- |
| @Mapper **public interface** DepartmentMapper {   @Select(**"select \* from department where id=#{id}"**)  **public** Department getDeptById(Integer id);   @Delete(**"delete from department where id=#{id}"**)  **public int** deleteDeptById(Integer id);   @Options(useGeneratedKeys = **true**,keyProperty = **"id"**)  @Insert(**"insert into department(department\_name) values(#{departmentName})"**)  **public int** insertDept(Department department);   @Update(**"update department set department\_name=#{departmentName} where id=#{id}"**)  **public int** updateDept(Department department); } |

**扫描包**

此处的@Mapper注解，如果我们有100个mapper每个都得加该注解，很麻烦，不如配置扫描包，让mybatis直接扫描某个文件夹下的所有文件，自动加载。

在主类SpringbootMybatisApplication加上如下注解

|  |
| --- |
| @MapperScan(value = **"com.gh.springbootmybatis.mapper"**) |



**驼峰装配bean**

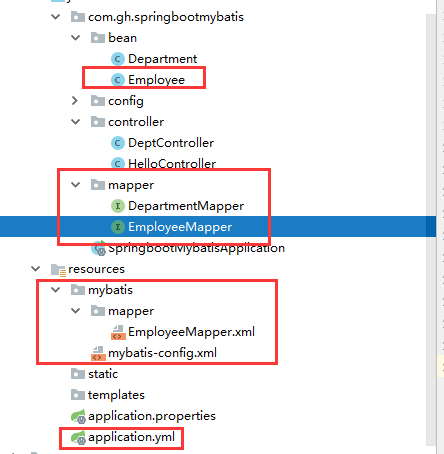
此刻如果我们表的字段名是department\_name,而我们的Javabean的字段名是departmentName,此刻发现运行时不能自动装配，因此我们需要加个驼峰设置。

新建类：MyBatisConfig

|  |
| --- |
| @org.springframework.context.annotation.Configuration **public class** MyBatisConfig {  @Bean  **public** ConfigurationCustomizer configurationCustomizer(){  **return new** ConfigurationCustomizer(){  @Override  **public void** customize(Configuration configuration) {  *//设置该值为true就可以解决* configuration.setMapUnderscoreToCamelCase(**true**);  }  };  } } |

**配置文件版**

工程结构如下



1）、新建Employee类，这是bean类

2）、新建接口EmployeeMapper

|  |
| --- |
| **public interface** EmployeeMapper {  **public** Employee getEmpById(Integer id);  **public void** insertEmp(Employee employee); } |

3）、新建mybatis-config.xml文件，位置看工程结构

|  |
| --- |
| *<?***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**>  <**setting name="mapUnderscoreToCamelCase" value="true"**/>  </**settings**> </**configuration**> |

4）、新建EmployeeMapper.xml文件

|  |
| --- |
| *<?***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.gh.springbootmybatis.mapper.EmployeeMapper"**>  <**select id="getEmpById" resultType="com.gh.springbootmybatis.bean.Employee"**>  SELECT *\** FROM employee WHERE id=#{id}  </**select**>   <**insert id="insertEmp"**>  INSERT INTO employee(lastName,email,gender,d\_id) VALUES (#{lastName},#{email},#{gender},#{dId})  </**insert**> </**mapper**> |

5）、在application.yml文件中添加如下配置

|  |
| --- |
| **mybatis**:  *# 指定全局配置文件位置* **config-location**: classpath:mybatis/mybatis-config.xml  *# 指定sql映射文件位置* **mapper-locations**: classpath:mybatis/mapper/\*.xml |

更多可以参考如下网站：

http://www.mybatis.org/spring-boot-starter/mybatis-spring-boot-autoconfigure/

1. **Springboot与缓存**
2. **JSR107**

对于缓存，Java出了JSR107标准

Java Caching定义了5个核心接口，分别是CachingProvider, CacheManager, Cache, Entry 和 Expiry。

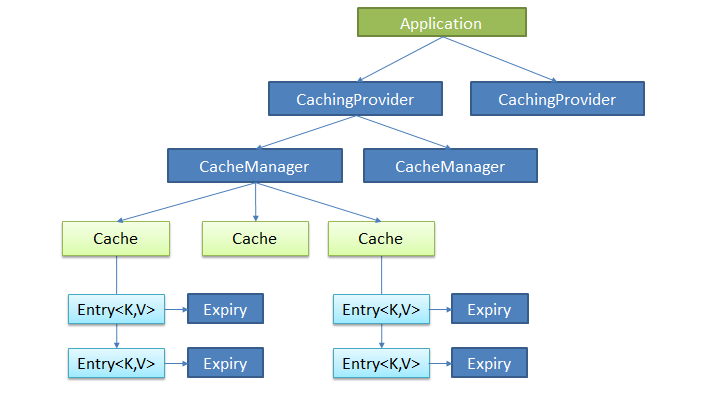
1）、CachingProvider定义了创建、配置、获取、管理和控制多个CacheManager。一个应用可以在运行期访问多个CachingProvider。

2）、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。

3）、Cache是一个类似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。

4）、Entry是一个存储在Cache中的key-value对。

5）、Expiry每一个存储在Cache中的条目有一个定义的有效期，即Expiry Duration。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。



由于JSR107开发起来比较麻烦，spring定义了自己的缓存规范，我们一般使用spring的规范，不使用JSR107

1. **Spring的缓存抽象**

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（JSR107）注解简化我们的开发。

Cache接口为缓存的组件规范定义，包含缓存的各种操作集合

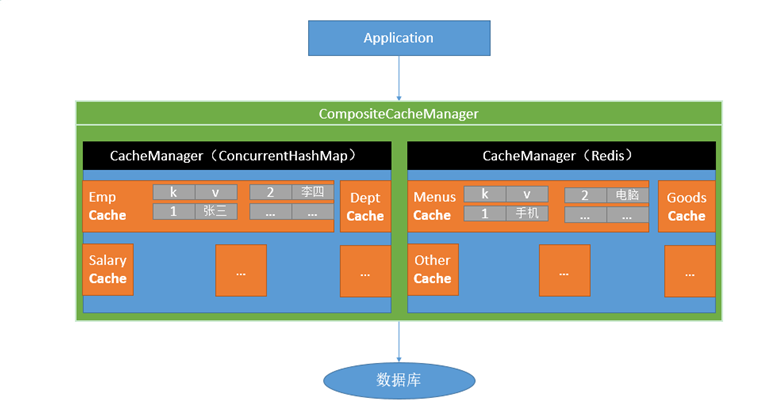
Cache接口下spring提供了各种xxxCache的实现；比如RedisCache、EhCacheCache、ConcurrentMapCache等

每次调用需要缓存功能的方法时，spring会检查指定参数的指定目标方法是否已经被调用过，如果有就直接从缓存中获取方法调用后的结果，如果没有就调用方法

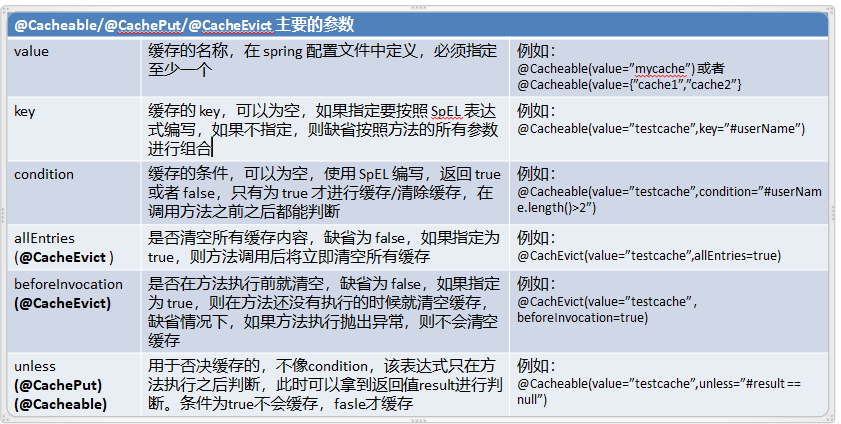
使用spring缓存抽象时我们需要关注以下两点：

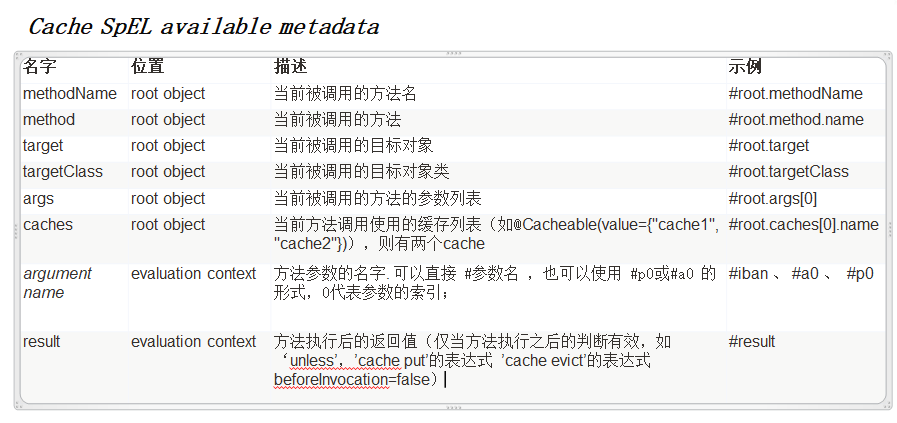
1）、确定方法需要被缓存以及他们的缓存策略

2）、从缓存中读取之前缓存存储的数据









1. **示例demo**

此代码是在集成mybatis的基础上做的。因此集成mybatis的步骤就省略了。

新增DepartmentSvc类



|  |
| --- |
| @Component **public class** DepartmentSvc {  @Autowired  **private** DepartmentMapper **departmentMapper**;  */\*\*  \** ***@Cacheable将方法的结果缓存，这样下次调用就不用查询数据库了*** *\** ***@param id*** *\** ***@return*** *\*/* @Cacheable(cacheNames = **"emp"**,key = **"#root.args[0]"**)  **public** Department getDeptById(Integer id){  **return departmentMapper**.getDeptById(id);  }  **public int** insertDept(Department department){  **return departmentMapper**.insertDept(department);  } } |

pom.xml文件新增如下：

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

此时我们调用getDeptById方法发现只有第一次才会查询数据库，以后都只会从缓存中获取数据，除非参数id值改变。

CacheManager管理多个cache组件，对缓存的真正CRUD操作在Cache组件中，每一个缓存组件有自己唯一的一个名字，如下是Cacheable的几个属性。

cacheNames/value:指定缓存组件的名字

key:缓存数据使用的key，可以用它来指定，默认是使用方法参数的值 1-方法的返回值；编写SpEL：#id,参数id的值 #a0 #p0 #root.args[0]

keyGenerator:key的生成器，可以自己指定key的生成器的组件id

cacheManager：指定缓存管理器，或者cacheResolver指定获取解析器

condition：指定符合条件的情况下才缓存，condition=”#a0>1”,第一个参数的值>1的时候才进行缓存。

unless：否定缓存，当unless指定的 条件为true，方法的返回值就不会被缓存，可以获取到结果进行判断；unless = “#result == null”

sync:是否使用异步模式，如果设置为true表示执行查询数据库方法和缓存是异步执行的，此时的unless就会失效，因此不建议使用，cache默认是false不开启的。

1. **缓存工作原理**

CacheAutoConfiguration缓存的自动配置类

缓存的配置类，优先级从上往下

org.springframework.boot.autoconfigure.cache.GenericCacheConfiguration

org.springframework.boot.autoconfigure.cache.JCacheCacheConfiguration

org.springframework.boot.autoconfigure.cache.EhCacheCacheConfiguration

org.springframework.boot.autoconfigure.cache.HazelcastCacheConfiguration

org.springframework.boot.autoconfigure.cache.InfinispanCacheConfiguration

org.springframework.boot.autoconfigure.cache.CouchbaseCacheConfiguration

org.springframework.boot.autoconfigure.cache.RedisCacheConfiguration

org.springframework.boot.autoconfigure.cache.CaffeineCacheConfiguration

org.springframework.boot.autoconfigure.cache.GuavaCacheConfiguration

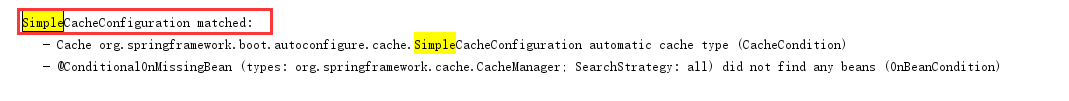
org.springframework.boot.autoconfigure.cache.NoOpCacheConfiguration

org.springframework.boot.autoconfigure.cache.SimpleCacheConfiguration

哪个默认配置类会生效？

在application.properties文件添加如下信息，启动springboot，就会打印出自动配置类哪些被使用了。

|  |
| --- |
| **debug** = **true** |



此处我们可以看到SimpleCacheConfiguration类被使用了。

SimpleCacheConfiguration类给容器中注册了一个ConcurrentMapCacheManager类型的缓存组件。

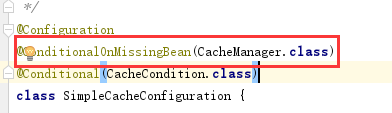
|  |
| --- |
| @Bean **public** ConcurrentMapCacheManager cacheManager() {  ConcurrentMapCacheManager cacheManager = **new** ConcurrentMapCacheManager();  List<String> cacheNames = **this**.cacheProperties.getCacheNames();  **if** (!cacheNames.isEmpty()) {  cacheManager.setCacheNames(cacheNames);  }  **return** (ConcurrentMapCacheManager)**this**.customizerInvoker.customize(cacheManager); } |

它可以获取和创建ConcurrentMapCache类型的缓存组件，他的作用是将数据保存在ConcurrentMap中

|  |
| --- |
| @Override @Nullable **public** Cache getCache(String name) {  Cache cache = **this**.**cacheMap**.get(name);  **if** (cache == **null** && **this**.**dynamic**) {  **synchronized** (**this**.**cacheMap**) {  cache = **this**.**cacheMap**.get(name);  **if** (cache == **null**) {  cache = createConcurrentMapCache(name);  **this**.**cacheMap**.put(name, cache);  }  }  }  **return** cache; } |

|  |
| --- |
| **protected** Cache createConcurrentMapCache(String name) {  SerializationDelegate actualSerialization = (isStoreByValue() ? **this**.**serialization** : **null**);  **return new** ConcurrentMapCache(name, **new** ConcurrentHashMap<>(256),  isAllowNullValues(), actualSerialization);  }  ConcurrentMapCache类中  **protected** ConcurrentMapCache(String name, ConcurrentMap<Object, Object> store,  **boolean** allowNullValues, @Nullable SerializationDelegate serialization) {   **super**(allowNullValues);  Assert.*notNull*(name, **"Name must not be null"**);  Assert.*notNull*(store, **"Store must not be null"**);  **this**.**name** = name;  **this**.**store** = store;  **this**.**serialization** = serialization; }  此处的store就是保存缓存组件 |

此处如果我们加入了其他依赖的东西进来，比如Redis的依赖，那么SimpleCacheConfiguration类就不会匹配上，而RedisCacheConfiguration类就会匹配上，匹配上后，就会往bean容器中注入RedisCacheManager类，此类是CacheManager的子类，因此在SimpleCacheConfiguration时，会判断有没有CacheManager，有就不会匹配。



1. **缓存工作流程**

1)、方法运行之前，先去查询Cache，按照cacheNames指定的名字获取（CacheManager先获取相应的缓存），第一次获取缓存如果没有Cache组件会自动创建。

2）、去Cache中查找缓存的内容，使用一个key,默认就是方法的参数，key是按照某种策略生成的，默认是使用keyGenerator生成的，默认使用SimpleKeyGenerator生成key

SimpleKeyGenerator生成key的策略：

如果没有参数，key=new SimpleKey()

如果有一个参数，key=参数的值

如果有多个参数：key=new SimpleKey(params)

3）、没有查到缓存就调用目标方法

4）、将目标方法返回的结果，放进缓存中。

@Cacheable标注的方法执行之前先来检查缓存中有么有这个数据，默认按照参数的值作为key去查询缓存，如果没有就运行方法并将结果放入缓存。以后再来调用就可以直接使用缓存中的数据

1. **指定KeyGenerator**

编写自己的KeyGenerator

|  |
| --- |
| @org.springframework.context.annotation.Configuration **public class** MyCacheConfig {  @Bean(**"myKeyGenerator"**)  **public** KeyGenerator keyGenerator(){  **return new** KeyGenerator(){  @Override  **public** Object generate(Object target, Method method, Object... params) {  **return** method.getName()+**"["**+ Arrays.*asList*(params)+**"]"**;  }  };  } } |

指定使用自己的KeyGenerator，根据beanid去指定（此处我们是myKeyGenerator，因此要保持一致）

|  |
| --- |
| @Cacheable(cacheNames = **"emp"**,keyGenerator = **"myKeyGenerator"**) **public** Department getDeptById(Integer id){  **return departmentMapper**.getDeptById(id); } |

1. **@CachePut**

@CachePut保证方法被调用，又更新缓存数据

测试步骤：

1）、查询11号部门，查到结果会放进缓存中

key:11 value:0000

2)、以后查询还是之前的结果

3）、更新11号部门departmentName为1111

将方法的返回值也放进缓存

Key，传入的department对象值，返回的department对象

4）、查询11号员工

应该是更新后的员工

key = “#department.id”使用传入的参数的部门id

key = “#result.id” 使用返回后的id

controller代码

|  |
| --- |
| @GetMapping(**"/dept/{id}"**)  **public** Department getDepartment(@PathVariable(**"id"**) Integer id){  **return departmentSvc**.getDeptById(id);  }@GetMapping(**"/dept"**)  **public** Department update(Department department){  **departmentSvc**.updateDept(department);  **return** department;  } |

Service类代码

|  |
| --- |
| //不设置key，默认就是第一个参数值  @Cacheable(value = **"emp"**) **public** Department getDeptById(Integer id){  **return departmentMapper**.getDeptById(id); }  //此处查询和更新使用key一定要是一样的，要不然缓存就不起作用 @CachePut(value = **"emp"**,key = **"#department.id"**) **public** Department updateDept(Department department){  **departmentMapper**.updateDept(department);  **return** department; } |

1. **@CacheEvict**

缓存清除

|  |
| --- |
| @CacheEvict(value = **"emp"**,key = **"#id"**) **public void** deleteDept(Integer id){  **departmentMapper**.deleteDeptById(id); } |

key:指定要清除的数据

allEntries:删除所有数据，此时指定该值为true就不用再指定key值了

beforeInvocation：缓存的清除是否在方法之前执行，默认是false，代表在方法执行之后执行。

1. **集成redis**

pom.xml文件加入如下依赖，可通过网址：

<https://docs.spring.io/spring-boot/docs/2.1.0.RELEASE/reference/htmlsingle/#using-boot-starter>查看

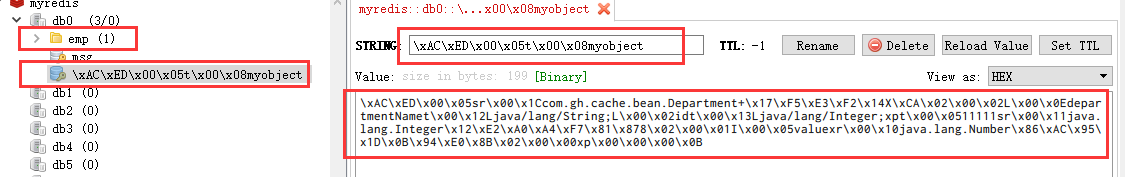
|  |
| --- |
| <**dependency**>  <**groupId**>org.springframework.boot</**groupId**>  <**artifactId**>spring-boot-starter-data-redis</**artifactId**> </**dependency**> |

测试：

|  |
| --- |
| @RunWith(SpringRunner.**class**) @SpringBootTest **public class** SpringbootCacheApplicationTests {   @Autowired  **private** RedisTemplate **redisTemplate**;  *//操作字符串的template* @Autowired  **private** StringRedisTemplate **stringRedisTemplate**;  @Autowired  **private** DepartmentSvc **departmentSvc**;  */\*\*  \* redis常见的五大数据类型：  \* string、list、set、hash、zset  \*/* @Test  **public void** testRedis() {  *//在msg字段中追加值为你好* **stringRedisTemplate**.opsForValue().append(**"msg"**,**" 你好"**);  *//获取字段msg的值* String msg = **stringRedisTemplate**.opsForValue().get(**"msg"**);  System.***out***.println(msg);  *//操作列表,在列表list中左边插入值为1* **stringRedisTemplate**.opsForList().leftPush(**"mylist"**,**"2"**);  *//获取值并从列表中删除* System.***out***.println( **stringRedisTemplate**.opsForList().leftPop(**"mylist"**));  *//测试保存对象，默认使用的是jdk序列化机制，保存的是序列化后的数据* **redisTemplate**.opsForValue().set(**"myobject"**,**departmentSvc**.getDeptById(11));  } } |

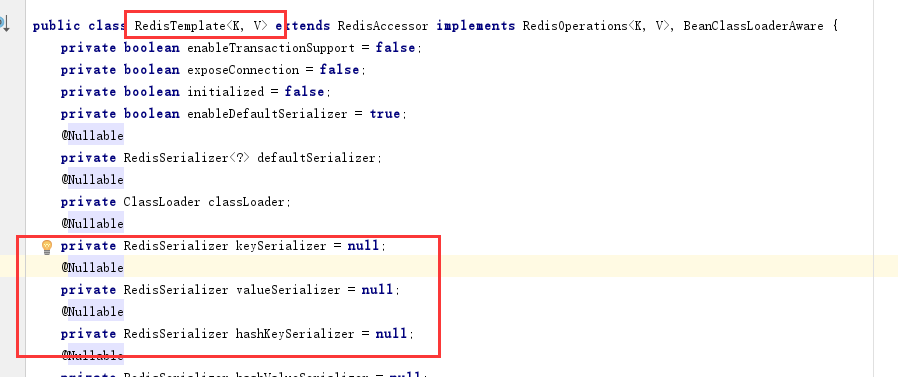
**对象序列化问题**

此处注意，我们在往Redis里面存放对象的时候，默认使用的是jdk序列化机制，保存的是序列化后的数据，会出现如下情况：



这就是明显的序列化问题，因此我们可以将对象转为json数据的方式保存，自己将对象转为json，此处我们一般使用Redis自己的序列化规则。

点击RedisTemplate类，查看源代码



相关配置我们可以查看类RedisAutoConfiguration

自定义Redis的序列化，配置类中加入如下代码

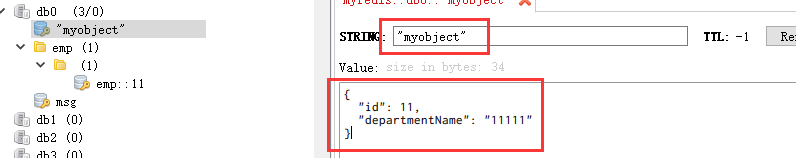
Jackson2JsonRedisSerializer类是RedisSerializer的子类，查看继承结构可以看到

|  |
| --- |
| @Bean @ConditionalOnMissingBean(name = **"myRedisTemplate"**) **public** RedisTemplate<Object, Department> myRedisTemplate(  RedisConnectionFactory redisConnectionFactory) **throws** UnknownHostException {  RedisTemplate<Object, Department> template = **new** RedisTemplate<>();  template.setConnectionFactory(redisConnectionFactory);  template.setDefaultSerializer(**new** Jackson2JsonRedisSerializer<Department>(Department.**class**));  **return** template; } |

测试类：

|  |
| --- |
| @Autowired **private** RedisTemplate **myRedisTemplate**;  @Test **public void** testRedis() {  *//测试保存对象，默认使用的是jdk序列化机制，保存的是序列化后的数据* **myRedisTemplate**.opsForValue().set(**"myobject"**,**departmentSvc**.getDeptById(11)); } |

效果：



**自定义CacheManager**

查看类RedisCacheConfiguration的如下代码

|  |
| --- |
| @Bean **public** RedisCacheManager cacheManager(RedisConnectionFactory redisConnectionFactory,  ResourceLoader resourceLoader) {  RedisCacheManagerBuilder builder = RedisCacheManager  .*builder*(redisConnectionFactory)  .cacheDefaults(determineConfiguration(resourceLoader.getClassLoader()));  List<String> cacheNames = **this**.**cacheProperties**.getCacheNames();  **if** (!cacheNames.isEmpty()) {  builder.initialCacheNames(**new** LinkedHashSet<>(cacheNames));  }  **return this**.**customizerInvoker**.customize(builder.build()); } |

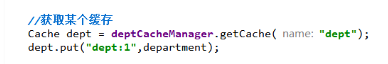
因此在我们的配置类中，按照该格式修改代码

|  |
| --- |
| @Bean **public** RedisCacheManager myCacheManager(RedisConnectionFactory redisConnectionFactory) {  RedisCacheConfiguration redisCacheConfiguration = RedisCacheConfiguration.*defaultCacheConfig*()  .entryTtl(Duration.*ofHours*(1));  *// 设置缓存有效期一小时* **return** RedisCacheManager .*builder*(RedisCacheWriter.*nonLockingRedisCacheWriter*(redisConnectionFactory))  .cacheDefaults(redisCacheConfiguration).build(); } |

使用

|  |
| --- |
| @Component @CacheConfig(cacheManager = **"myCacheManager"**) **public class** DepartmentSvc { |

**代码查询和插入缓存**



1. **Springboot与消息**
2. **概述**

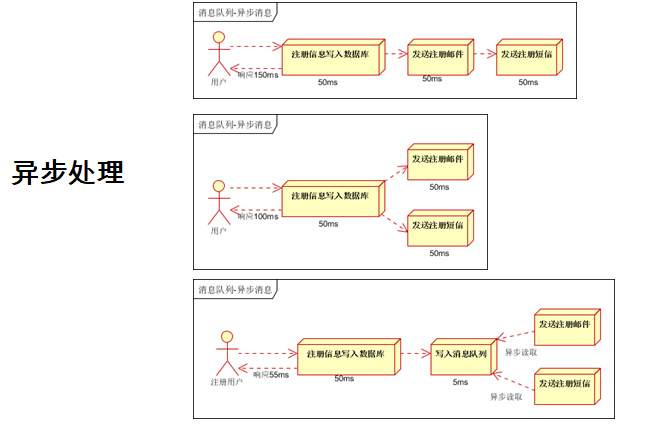
大多应用中，可通过消息服务中间件来提升系统异步通信、扩展解耦能力

消息服务中两个重要概念：消息代理（message broker）和目的地（destination），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。

消息队列主要有两种形式的目的地

* 队列（queue）：点对点消息通信（point-to-point）
* 主题（topic）：发布（publish）/订阅（subscribe）消息通信

异步处理





1）点对点式：

消息发送者发送消息，消息代理将其放入一个队列中，消息接收者从队列中获取消息内容，消息读取后被移出队列

消息只有唯一的发送者和接受者，但并不是说只能有一个接收者

2）发布订阅式：

发送者（发布者）发送消息到主题，多个接收者（订阅者）监听（订阅）这个主题，那么就会在消息到达时同时收到消息

3）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

4）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

JMS和AMQP对比：



1. **RabbitMQ简介**

RabbitMQ是一个由erlang开发的AMQP(Advanved Message Queue Protocol)的开源实现。

核心概念：

1）、Message消息，消息是不具名的，它由消息头和消息体组成。消息体是不透明的，而消息头则由一系列的可选属性组成，这些属性包括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。

2）、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。

3）、Exchange

交换器，用来接收生产者发送的消息并将这些消息路由给服务器中的队列。

Exchange有4种类型：direct(默认)，fanout, topic, 和headers，不同类型的Exchange转发消息的策略有所区别

4）、Queue

消息队列，用来保存消息直到发送给消费者。它是消息的容器，也是消息的终点。一个消息可投入一个或多个队列。消息一直在队列里面，等待消费者连接到这个队列将其取走。

5）、Binding

绑定，用于消息队列和交换器之间的关联。一个绑定就是基于路由键将交换器和消息队列连接起来的路由规则，所以可以将交换器理解成一个由绑定构成的路由表。

Exchange 和Queue的绑定可以是多对多的关系。

6）、Connection

网络连接，比如一个TCP连接。

7）、Channel

信道，多路复用连接中的一条独立的双向数据流通道。信道是建立在真实的TCP连接内的虚拟连接，AMQP 命令都是通过信道发出去的，不管是发布消息、订阅队列还是接收消息，这些动作都是通过信道完成。因为对于操作系统来说建立和销毁 TCP 都是非常昂贵的开销，所以引入了信道的概念，以复用一条 TCP 连接。

8）、Consumer

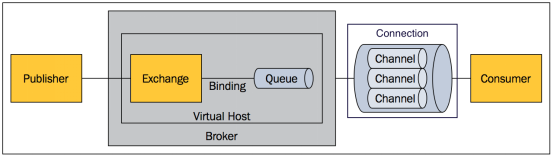
消息的消费者，表示一个从消息队列中取得消息的客户端应用程序。

9）、Virtual Host

虚拟主机，表示一批交换器、消息队列和相关对象。虚拟主机是共享相同的身份认证和加密环境的独立服务器域。每个 vhost 本质上就是一个 mini 版的 RabbitMQ 服务器，拥有自己的队列、交换器、绑定和权限机制。vhost 是 AMQP 概念的基础，必须在连接时指定，RabbitMQ 默认的 vhost 是 / 。

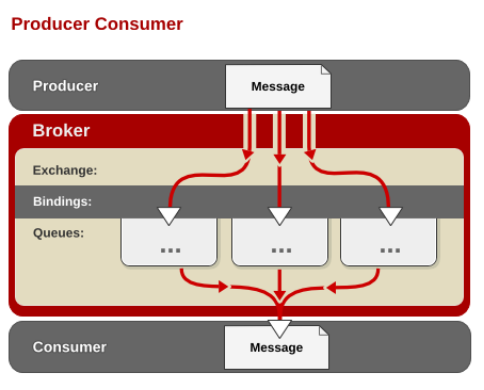
10）、Broker

表示消息队列服务器实体



1. **RabbitMQ运行机制**

AMQP 中的消息路由：AMQP 中消息的路由过程和 Java 开发者熟悉的 JMS 存在一些差别，AMQP 中增加了 Exchange 和 Binding 的角色。生产者把消息发布到 Exchange 上，消息最终到达队列并被消费者接收，而 Binding 决定交换器的消息应该发送到那个队列。

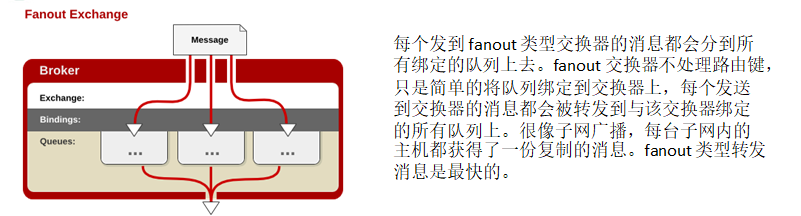


Exchange分发消息时根据类型的不同分发策略有区别，目前共四种类型：direct、fanout、topic、headers 。headers 匹配 AMQP 消息的 header 而不是路由键， headers 交换器和 direct 交换器完全一致，但性能差很多，目前几乎用不到了，所以直接看另外三种类型：

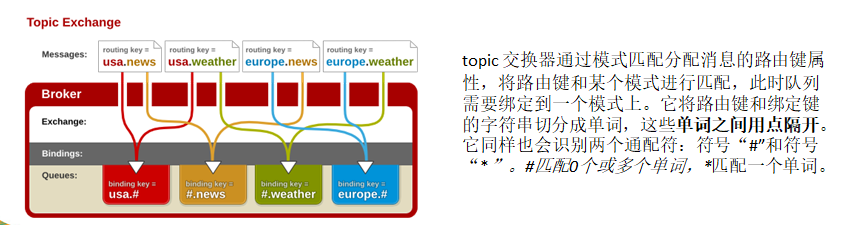
**Direct类型：**



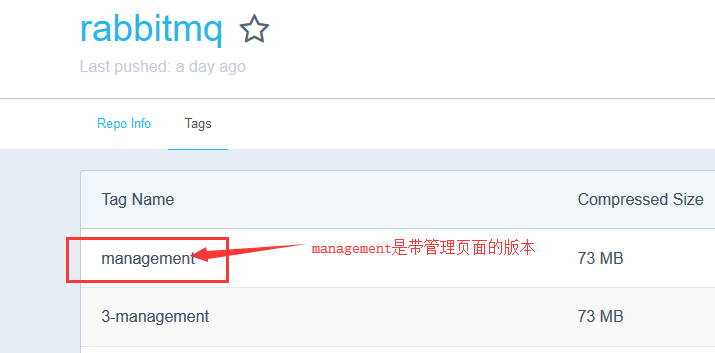
**Fanout类型**



**Topic类型**



1. **Rabbitmq安装与运行**



docker pull rabbitmq:management

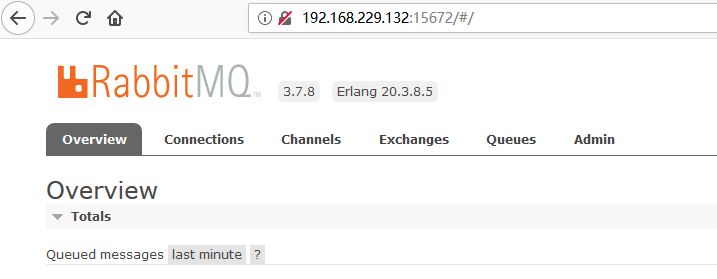
docker run -d -p 5672:5672 -p 15672:15672 --name myrabbitmq d69a5113ceae

d69a5113ceae：是imageid

15672：管理的端口号

浏览器访问：<http://192.168.229.132:15672>

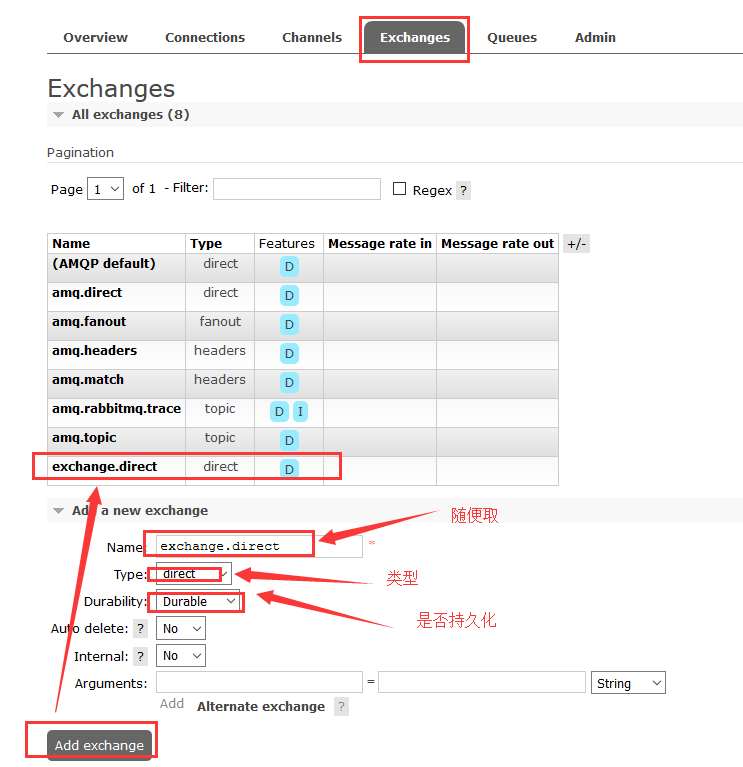
默认的账户名和密码是：guest 密码：guest



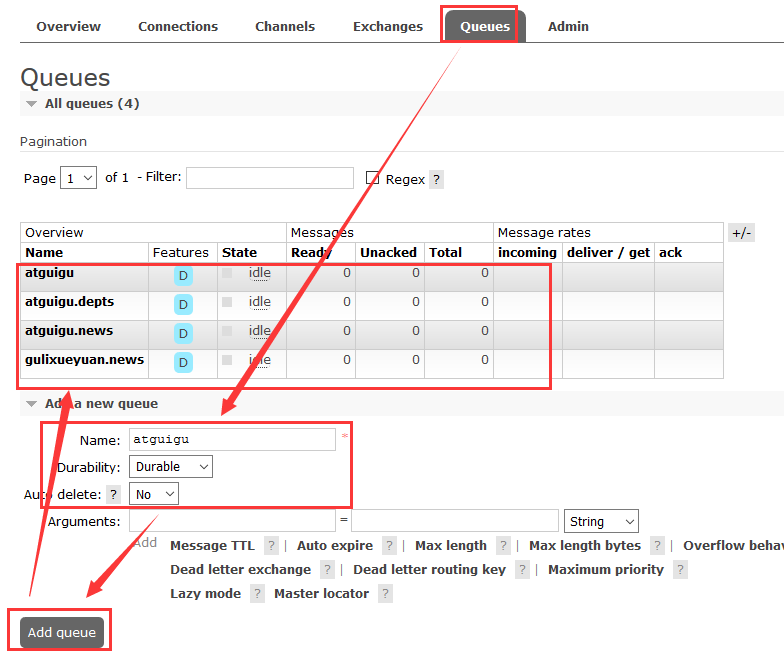
1. **Rabbitmq示例**

测试不同交换器（exchange）下，消息会发送都哪个队列？

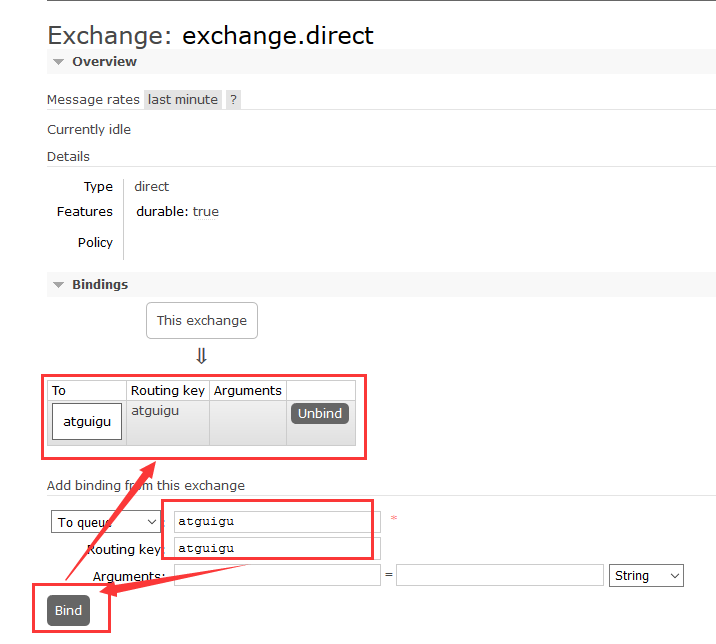
准备工作：新建一个direct、fanout交换器，如下示例direct的新建

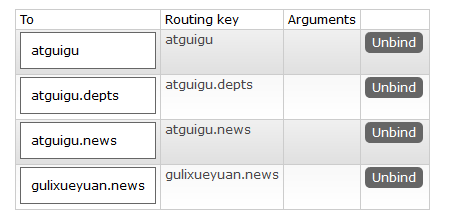


添加如下四个队列：

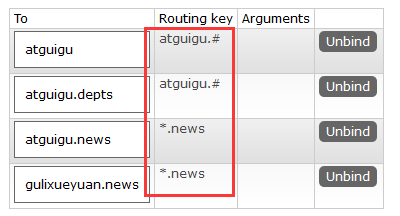


为刚才的3个交换器绑定队列，四个队列都要绑定，此处我只绑定了atguigu队列作为示例。

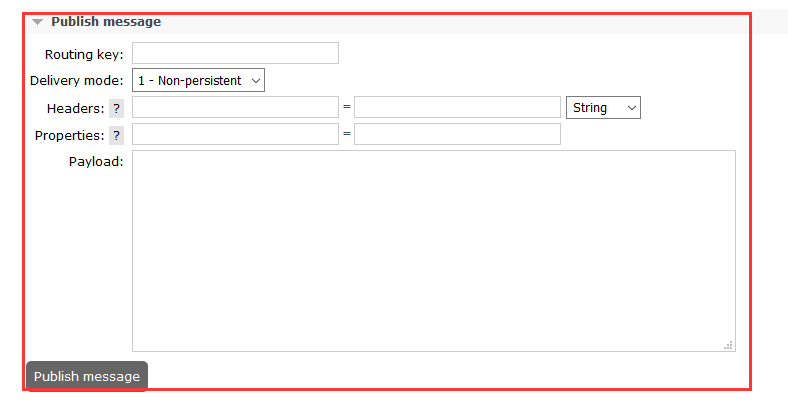




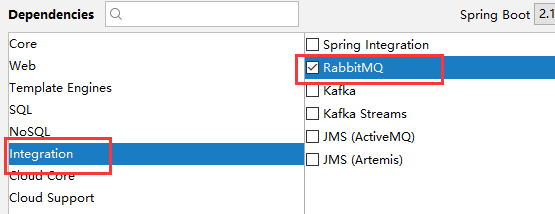
Topic的绑定有所不同，routing key不一样。

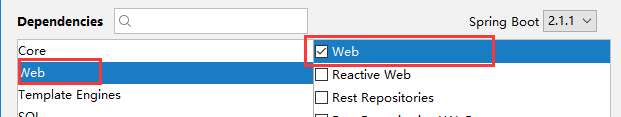


当然我们也可以在管理页面进行操作，发送消息。此页面在交换器页面里面



新建工程：





自动配置类：RabbitAutoConfiguration

在application.properties配置

|  |
| --- |
| **spring.rabbitmq.host**=**192.168.229.132 spring.rabbitmq.username**=**guest spring.rabbitmq.password**=**guest** |

定义自己的序列化配置

|  |
| --- |
| */\*\*  \* 此配置是为了序列化的，不配置就使用jdk的序列化，这种是有问题的  \** ***@return*** *\*/* @Bean **public** MessageConverter messageConverter(){  **return new** Jackson2JsonMessageConverter(); } |

发送消息和接收消息：

|  |
| --- |
| @Autowired **private** RabbitTemplate **rabbitTemplate**;  *//发送消息* @Test **public void** test01() {  Map msg = **new** HashMap();  msg.put(**"abc"**,**"nihao"**);  *//存放对象，此处要使用自己的序列化配置，默认的有问题* **rabbitTemplate**.convertAndSend(**"exchange.direct"**,**"atguigu"**, msg); }  *//接收消息* @Test **public void** receive() {  Map msg = (HashMap)**rabbitTemplate**.receiveAndConvert(**"atguigu"**);  System.***out***.println(msg); } |

1. **Rabbitmq的监听**

新建book类

|  |
| --- |
| **public class** Book {   **private** String **name**;  **private** String **title**;  } |

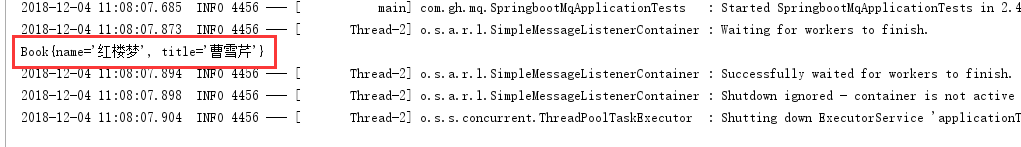
新建BookService类监听

|  |
| --- |
| @Service **public class** BookService {   *//atguigu队列名* @RabbitListener(queues = **"atguigu"**)  **public void** receive(Book book){  System.***out***.println(book);  } } |

测试类：

|  |
| --- |
| *//发送消息* @Test **public void** test01() {  Book book = **new** Book();  book.setName(**"红楼梦"**);  book.setTitle(**"曹雪芹"**);  *//存放对象，此处要使用自己的序列化配置，默认的有问题* **rabbitTemplate**.convertAndSend(**"exchange.direct"**,**"atguigu"**, book); } |

启动测试类方法：



此处发现监听器已经收到测试类发送的消息

1. **Rabbitmq代码创建交换器和队列**

|  |
| --- |
| @Autowired **private** AmqpAdmin **amqpAdmin**;  @Test **public void** create() {  *//创建交换器* Exchange exchange = **new** DirectExchange(**"exchange.test"**);  **amqpAdmin**.declareExchange(exchange);  *//创建队列* **amqpAdmin**.declareQueue(**new** Queue(**"test"**,**true**));  *//创建绑定* **amqpAdmin**.declareBinding(**new** Binding(**"test"**,Binding.DestinationType.***QUEUE***,**"exchange.test"**,**"amqb.\*"**,**null**)); } |

1. **Springboot与检索**
2. **Elasticsearch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。

1. **安装与运行**

Elasticsearch启动的时候默认会占用2个G的内存，但是我们也可以指定大小。

官方安装教程：

<https://www.elastic.co/guide/en/elasticsearch/reference/6.5/docker.html>

下载镜像：

docker pull docker.elastic.co/elasticsearch/elasticsearch:6.5.1

运行：

docker run -d -p 9200:9200 -p 9300:9300 -e "discovery.type=single-node" 32f93c89076d

32f93c89076d:是imageid

运行成功后访问如下网站，显示如下数据则说明运行成功

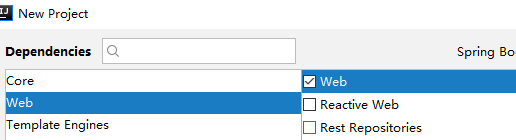


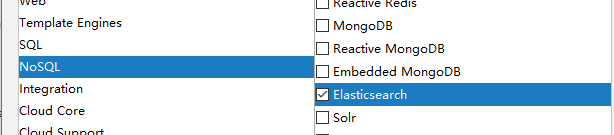
中文教程：

<https://www.elastic.co/guide/cn/elasticsearch/guide/current/index.html>

1. **整合Elasticsearch**

新建工程





Springboot默认支持两种技术来和es交互

1）、Jest(默认不支持)，需要导入jest工具包依赖才行

2）、springdata Elasticsearch默认支持，自动配置类：ElasticsearchDataAutoConfiguration

通过ElasticsearchTemplate类操作ES

1. **Jest方式**

官方文档：

<https://github.com/searchbox-io/Jest/tree/master/jest>

pom.xml文件

|  |
| --- |
| *<!--导入jest依赖-->* <**dependency**>  <**groupId**>io.searchbox</**groupId**>  <**artifactId**>jest</**artifactId**>  <**version**>6.3.0</**version**> </**dependency**> |

新建bean类Article

|  |
| --- |
| **public class** Article {  **private** String **id**;  **private** String **author**;  **private** String **name**;  } |

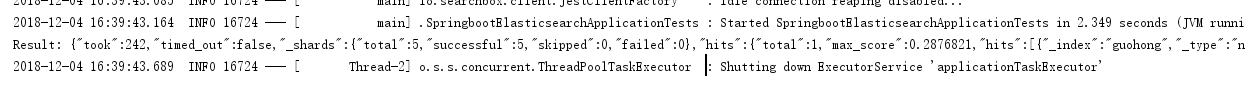
application.properties配置文件

|  |
| --- |
| **spring.elasticsearch.jest.uris**=**http://192.168.229.132:9200** |

测试类：

|  |
| --- |
| @Autowired **private** JestClient **jestClient**;  @Test **public void** testJest() **throws** Exception{  *//保存一个文档* Article article = **new** Article();  article.setAuthor(**"曹雪芹"**);  article.setId(**"123"**);  article.setName(**"红楼梦"**);  Index index = **new** Index.Builder(article).index(**"guohong"**).type(**"news"**).build();  **jestClient**.execute(index);   *//查询* String json = **"{\n"** +  **" \"query\" : {\n"** +  **" \"match\" : {\n"** +  **" \"id\" : \"123\"\n"** +  **" }\n"** +  **" }\n"** +  **"}"**;  Search search = **new** Search.Builder(json).addIndex(**"guohong"**).addType(**"news"**).build();  JestResult result = **jestClient**.execute(search);  System.***out***.println(result); } |

效果：



1. **Springdata方式**

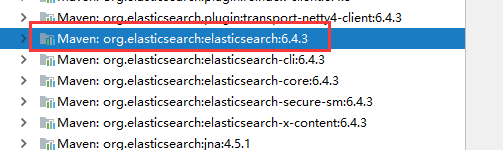
此处会涉及到版本适配问题，springdata的版本必须和Elasticsearch适配。

可以查看相关信息的官方网址（此内容容易有误导）：

<https://github.com/spring-projects/spring-data-elasticsearch>

我们可以通过查看springdata的版本，让Elasticsearch的版本去适配springdata，这样操作比较简单。





此处我们可以看到依赖的是Elasticsearch的版本是6.4.3，因此我们重新安装一个Elasticsearch6.4.3的版本就好了。

pom.xml配置文件

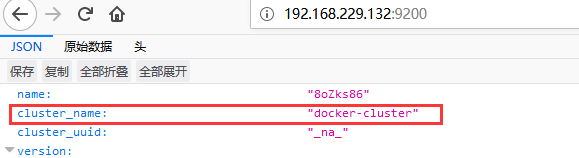
|  |
| --- |
| <**dependency**>  <**groupId**>org.springframework.boot</**groupId**>  <**artifactId**>spring-boot-starter-data-elasticsearch</**artifactId**> </**dependency**> |

application.properties文件

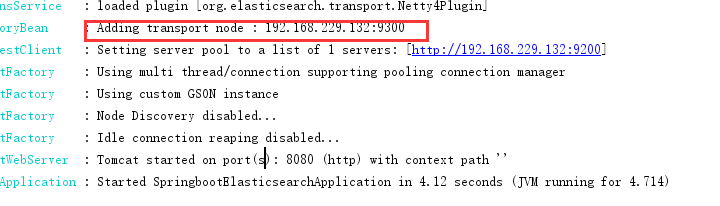
|  |
| --- |
| *#节点名* **spring.data.elasticsearch.cluster-name**=**docker-cluster** *#9300端口而不是9200，此处跟jest方式不一样* **spring.data.elasticsearch.cluster-nodes**=**192.168.229.132:9300** |

cluster-name的来源通过查看地址：

<http://192.168.229.132:9200/>获取



启动程序，不报错，出现如下信息表示成功



新建book类

|  |
| --- |
| @Document(indexName = **"guohong"**,type = **"book"**) **public class** Book {   **private** String **id**;  **private** String **author**;  **private** String **name**;  } |

新建BookRepository

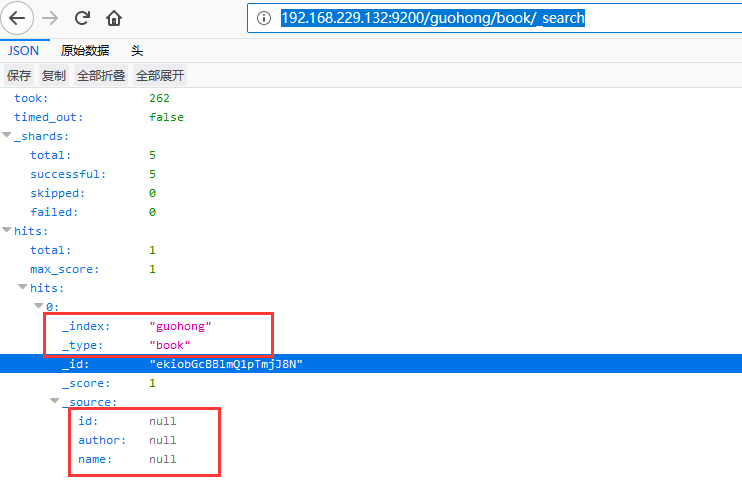
|  |
| --- |
| **import** com.gh.elasticsearch.bean.Book; **import** org.springframework.data.elasticsearch.repository.ElasticsearchRepository; **public interface** BookRepository **extends** ElasticsearchRepository<Book, String> {  } |

测试类

|  |
| --- |
| @Autowired **private** BookRepository **bookRepository**;  @Test **public void** test02(){  Book book = **new** Book();  book.setAuthor(**"曹雪芹"**);  book.setId(**"123"**);  book.setName(**"红楼梦"**);  **bookRepository**.index(book); } |

效果：

<http://192.168.229.132:9200/guohong/book/_search>



1. **Springboot与任务**
2. **异步任务**

HelloService类

|  |
| --- |
| @Service **public class** HelloService {   *//此方法调用是异步的* @Async  **public void** hello(){  **try** {  Thread.*sleep*(5000);  } **catch** (InterruptedException e) {  e.printStackTrace();  }  System.***out***.println( **"你好"**);  } } |

HelloController类

|  |
| --- |
| @RestController **public class** HelloController {  @Autowired  **private** HelloService **helloService**;  @RequestMapping(**"/hello"**)  **public** String hello(){  **helloService**.hello();  **return "hello"**;  } } |

主类：

|  |
| --- |
| *//开启异步注解* @EnableAsync @SpringBootApplication **public class** SpringbootElasticsearchApplication {   **public static void** main(String[] args) {  SpringApplication.*run*(SpringbootElasticsearchApplication.**class**, args);  } } |

此时通过调用controller接口，发现就是异步的，页面可以马上得到返回，过了5秒后台会打印出“你好”。

1. **定时任务**

项目开发中经常需要执行一些定时任务，比如需要在每天凌晨时候，分析一次前一天的日志信息。Spring为我们提供了异步执行任务调度的方式，提供TaskExecutor 、TaskScheduler 接口。

两个注解：@EnableScheduling、@Scheduled

cron表达式：





定时任务方法

|  |
| --- |
| */\*  \* 周一到周六，整分的时候执行一次，也就是每分钟执行一次  \* 0-4 \* \* \* \* MON-SAT：周一到周六，每分钟的0-4秒执行一次  \* 0,1,2,3 \* \* \* \* MON-SAT：周一到周六，每分钟的0,1,2,3秒执行一次  \* 0/4 \* \* \* \* MON-SAT：周一到周六，每4秒执行一次  \*/* @Scheduled(cron = **"0 \* \* \* \* MON-SAT"**) **public void** task1(){  System.***out***.println(**"定时任务"**); } |

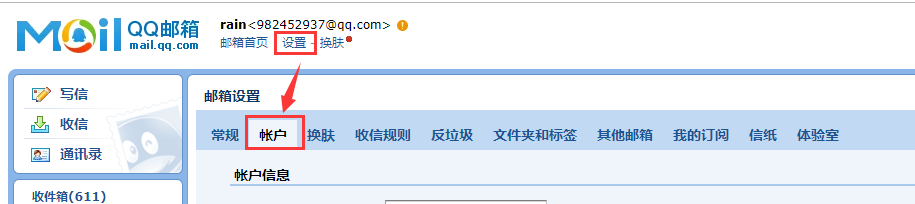
在主类开启定时任务注解

|  |
| --- |
| @EnableScheduling @SpringBootApplication **public class** SpringbootElasticsearchApplication {  } |

1. **邮件任务**

此处我们以qq邮箱为例：

首先我们要设置qq邮箱



找到如下两个服务将其开启

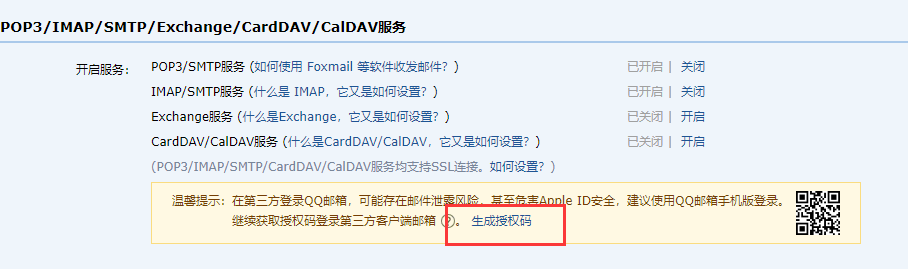


application.properties

|  |
| --- |
| **spring.mail.username**=**982452937@qq.com spring.mail.password**=**fxzhxazmltfrbdah spring.mail.host**=**smtp.qq.com spring.mail.properties.mail.smtp.ssl.enable**=**true** |

此处的password不是邮箱密码，而是授权码

可以根据如下生成



测试类：

|  |
| --- |
| @Autowired **private** JavaMailSenderImpl **javaMailSender**;  *//发送普通邮件* @Test **public void** test() {  SimpleMailMessage message = **new** SimpleMailMessage();  message.setText(**"邮件内容测试"**);  message.setTo(**"125257803@qq.com"**);  message.setFrom(**"982452937@qq.com"**);  message.setSubject(**"邮件测试"**);  **javaMailSender**.send(message); }  *//发送带附件的邮件* @Test **public void** test01() **throws** MessagingException {  MimeMessage message = **javaMailSender**.createMimeMessage();  MimeMessageHelper helper = **new** MimeMessageHelper(message,**true**);  helper.setText(**"邮件内容测试"**);  helper.setTo(**"125257803@qq.com"**);  helper.setFrom(**"982452937@qq.com"**);  helper.setSubject(**"邮件测试"**);  helper.addAttachment(**"1.jpg"**, **new** File(**"E:\\1.jpg"**));  **javaMailSender**.send(message); } |

1. **Springboot与安全**
2. **登录、认证、授权**

**Springboot2.0以下**

在此我们使用的是spring security作为权限认证的框架

步骤1：pom.xml文件配置

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

步骤2：spring security的配置类

|  |
| --- |
| @EnableWebSecurity **public class** MySecurityConfig **extends** WebSecurityConfigurerAdapter {  @Override  **protected void** configure(HttpSecurity http) **throws** Exception {  */\*  \* 访问level1下的文件时或者接口时，需要VIP1的角色  \* 访问level2下的文件时或者接口时，需要VIP2的角色  \* 访问level3下的文件时或者接口时，需要VIP3的角色  \*/* http.authorizeRequests().antMatchers(**"/"**).permitAll()  .antMatchers(**"/level1/\*\*"**).hasRole(**"VIP1"**)  .antMatchers(**"/level2/\*\*"**).hasRole(**"VIP2"**)  .antMatchers(**"/level3/\*\*"**).hasRole(**"VIP3"**);  *//开启登录功能* http.formLogin();  }  @Override  **protected void** configure(AuthenticationManagerBuilder auth) **throws** Exception {  */\*  \* 添加用户admin1，密码为123456的账户角色为VIP1和VIP2，该账户就能通过登录系统  \* 添加用户admin2，密码为123456的账户角色为VIP3和VIP2  \* 添加用户admin3，密码为123456的账户角色为VIP1和VIP3  \*/* auth.inMemoryAuthentication()  .withUser(**"admin1"**).password(**"123456"**).roles(**"VIP1"**,**"VIP2"**)  .and()  .withUser(**"admin2"**).password(**"123456"**).roles(**"VIP2"**,**"VIP3"**)  .and()  .withUser(**"admin3"**).password(**"123456"**).roles(**"VIP1"**,**"VIP3"**);  } } |

**Springboot2.0以上**

Spring boot 2.0.3引用的security 依赖是 spring security 5.X版本，此版本需要提供一个PasswordEncorder的实例，否则后台汇报错误：

步骤1与上面相同

只说步骤2:

新建PasswordEncorder类

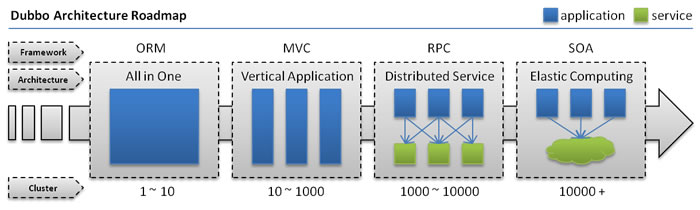
|  |
| --- |
| **public class** MyPasswordEncoder **implements** PasswordEncoder {  @Override  **public** String encode(CharSequence charSequence) {  **return** charSequence.toString();  }  @Override  **public boolean** matches(CharSequence charSequence, String s) {  **return** s.equals(charSequence.toString());  } } |

spring security的配置类

|  |
| --- |
| @EnableWebSecurity **public class** MySecurityConfig **extends** WebSecurityConfigurerAdapter {  @Override  **protected void** configure(HttpSecurity http) **throws** Exception {  */\*  \* 访问level1下的文件时或者接口时，需要VIP1的角色  \* 访问level2下的文件时或者接口时，需要VIP2的角色  \* 访问level3下的文件时或者接口时，需要VIP3的角色  \*/* http.authorizeRequests().antMatchers(**"/"**).permitAll()  .antMatchers(**"/level1/\*\*"**).hasRole(**"VIP1"**)  .antMatchers(**"/level2/\*\*"**).hasRole(**"VIP2"**)  .antMatchers(**"/level3/\*\*"**).hasRole(**"VIP3"**);  *//开启登录功能* http.formLogin();  *//开启自动注销功能,清空session,注销成功以后来到首页*  http.logout().logoutSuccessUrl(**"/"**);  }  @Override  **protected void** configure(AuthenticationManagerBuilder auth) **throws** Exception {  */\*  \* 添加用户admin1，密码为123456的账户角色为VIP1和VIP2，该账户就能通过登录系统  \* 添加用户admin2，密码为123456的账户角色为VIP3和VIP2  \* 添加用户admin3，密码为123456的账户角色为VIP1和VIP3  \*/* auth.inMemoryAuthentication().passwordEncoder(**new** MyPasswordEncoder())  .withUser(**"admin1"**).password(**"123456"**).roles(**"VIP1"**,**"VIP2"**)  .and()  .withUser(**"admin2"**).password(**"123456"**).roles(**"VIP2"**,**"VIP3"**)  .and()  .withUser(**"admin3"**).password(**"123456"**).roles(**"VIP1"**,**"VIP3"**);  } } |

1. **Springboot与分布式**
2. **分布式系统简介**

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。



1）、单一应用架构

当网站流量很小时，只需一个应用，将所有功能都部署在一起，以减少部署节点和成本。此时，用于简化增删改查工作量的数据访问框架(ORM)是关键。

2）、垂直应用架构

当访问量逐渐增大，单一应用增加机器带来的加速度越来越小，将应用拆成互不相干的几个应用，以提升效率。此时，用于加速前端页面开发的Web框架(MVC)是关键。

3）、分布式服务架构

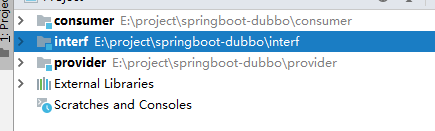
当垂直应用越来越多，应用之间交互不可避免，将核心业务抽取出来，作为独立的服务，逐渐形成稳定的服务中心，使前端应用能更快速的响应多变的市场需求。此时，用于提高业务复用及整合的分布式服务框架(RPC)是关键。

4）、流动计算架构

当服务越来越多，容量的评估，小服务资源的浪费等问题逐渐显现，此时需增加一个调度中心基于访问压力实时管理集群容量，提高集群利用率。此时，用于提高机器利用率的资源调度和治理中心(SOA)是关键。

1. **整合dubbo和zookeeper**

工程结构



Consumer和provider都依赖了interf

首先公共部分interf

pom.xml依赖文件

|  |
| --- |
| <**dependency**>  <**groupId**>com.alibaba.boot</**groupId**>  <**artifactId**>dubbo-spring-boot-starter</**artifactId**>  <**version**>0.2.0</**version**> </**dependency**> <**dependency**>  <**groupId**>com.github.sgroschupf</**groupId**>  <**artifactId**>zkclient</**artifactId**>  <**version**>0.1</**version**> </**dependency**> |

接口定义

|  |
| --- |
| **public interface** TicketService {  String getTicket(); } |

Provider部分

pom.xml文件

|  |
| --- |
| <**dependency**>  <**groupId**>com.gh</**groupId**>  <**artifactId**>interf</**artifactId**>  <**version**>0.0.1-SNAPSHOT</**version**> </**dependency**> |

application.properties

|  |
| --- |
| *#应用名* **dubbo.application.name**=**provider-ticket** *#zk的远程地址* **dubbo.registry.address**=**zookeeper://192.168.229.132:2181** *#扫描包下的所有接口文件* **dubbo.scan.base-packages**=**com.gh.interf** |

TicketServiceImpl类，注意此处的service注解是dubbo的不是spring的。

|  |
| --- |
| **import** com.alibaba.dubbo.config.annotation.Service; **import** com.gh.interf.TicketService; **import** org.springframework.stereotype.Component;  @Component @Service **public class** TicketServiceImpl **implements** TicketService {  @Override  **public** String getTicket() {  **return "66666"**;  } } |

主类

|  |
| --- |
| *//此处一定要加上EnableDubbo注解启用dubbo* @EnableDubbo @SpringBootApplication **public class** ProviderApplication {   **public static void** main(String[] args) {  SpringApplication.*run*(ProviderApplication.**class**, args);  } } |

Consumer部分

pom.xml依赖文件

|  |
| --- |
| <**dependency**>  <**groupId**>com.gh</**groupId**>  <**artifactId**>interf</**artifactId**>  <**version**>0.0.1-SNAPSHOT</**version**> </**dependency**> |

application.properties配置文件

|  |
| --- |
| **dubbo.application.name**=**consumer-user dubbo.registry.address**=**zookeeper://192.168.229.132:2181** |

UserService类

|  |
| --- |
| **import** com.alibaba.dubbo.config.annotation.Reference; **import** com.gh.interf.TicketService; **import** org.springframework.stereotype.Service; @Service **public class** UserService {  @Reference  **private** TicketService **ticketService**;  **public void** hello(){  System.***out***.println(**ticketService**.getTicket());  } } |

测试类：

|  |
| --- |
| @RunWith(SpringRunner.**class**) @SpringBootTest **public class** ConsumerApplicationTests {  @Autowired  **private** UserService **userService**;  @Test  **public void** test() {  **userService**.hello();  } } |

另外的一个demo地址：

<https://github.com/xiaoze-smirk/dubbo-springboot>

1. **整合springcloud**

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：

1）、服务发现——Netflix Eureka

2）、客服端负载均衡——Netflix Ribbon

3）、断路器——Netflix Hystrix

4）、服务网关——Netflix Zuul

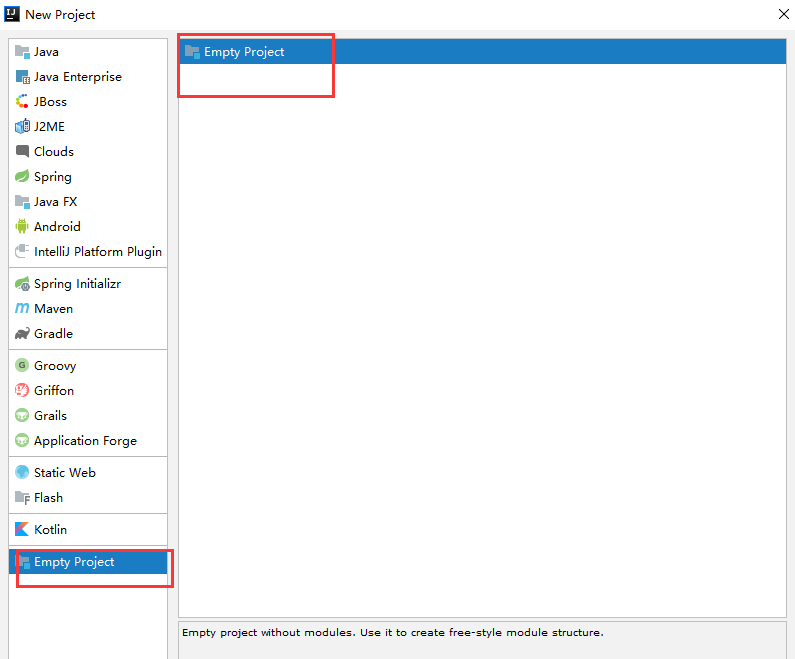
5）、分布式配置——Spring Cloud Config

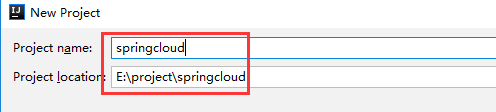
**Eureka demo**

此处eureka的作用跟zookeeper差不多，最初我使用的是springboot2.1版本，在创建完eureka-server项目并启动后，访问<http://127.0.0.1:8761/>地址后并没有出现eureka的管理页面。在我多次尝试后也失败了，然后我从网址：<https://github.com/wuqiangxjtu/spring-cloud-eureka-example下载了其demo是基于1.3>版本的springboot，并且启动eureka-server项目，就成功了，此时我再返回去启动我之前的2.1版本的eureka-server工程，发现也成功了，该问题暂时找不到原因。

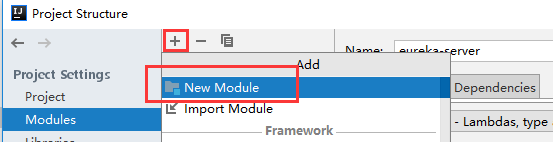
如下还是说明一下我们自己demo的创建过程

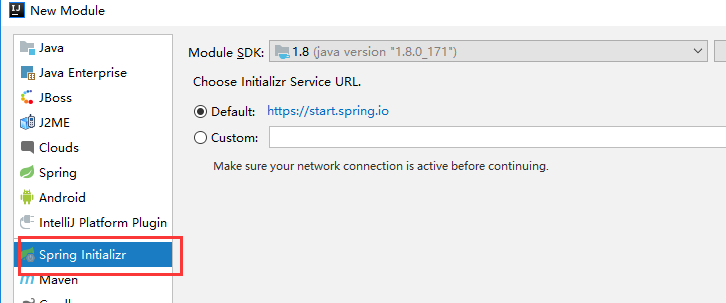
首先创建一个空工程，名叫springcloud

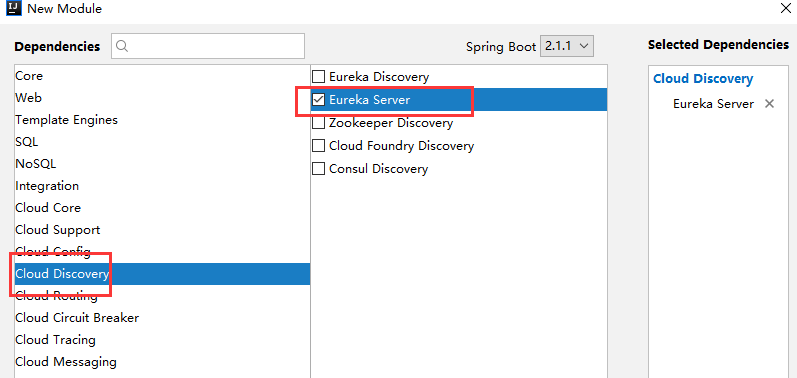




在空工程里面添加eureka-server模块module







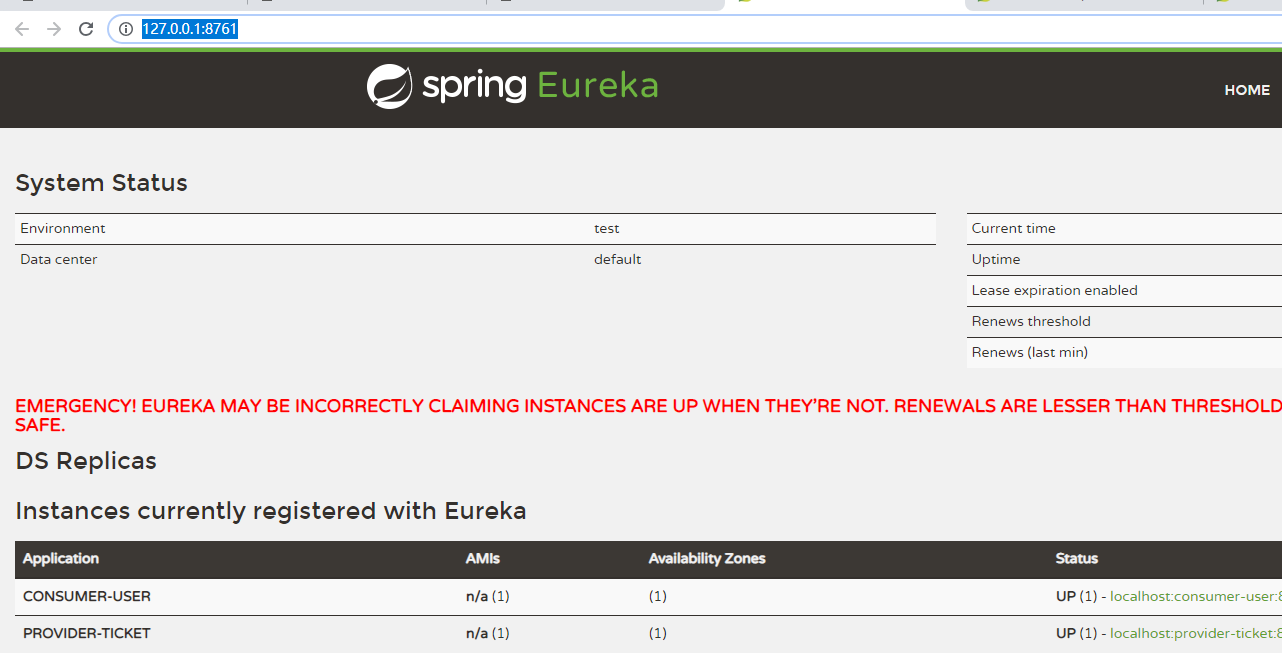
在application.yml配置文件中添加如下信息

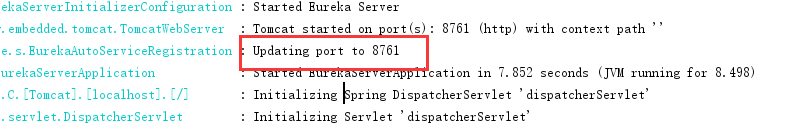
|  |
| --- |
| **server**:  **port**: 8761  **eureka**:  **instance**:  **hostname**: localhost  **client**:  **registerWithEureka**: **false** *#不把自己注册到eureka* **fetchRegistry**: **false** *#不从eureka上来获取服务的注册信息* **serviceUrl**:  **defaultZone**: http://${**eureka.instance.hostname**}:${**server.port**}/eureka/ |

主类

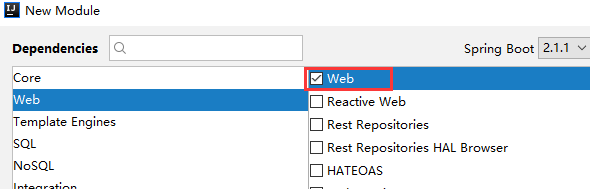
|  |
| --- |
| *//启动EnableEurekaServer* @EnableEurekaServer @SpringBootApplication **public class** EurekaServerApplication {   **public static void** main(String[] args) {  SpringApplication.*run*(EurekaServerApplication.**class**, args);  } } |

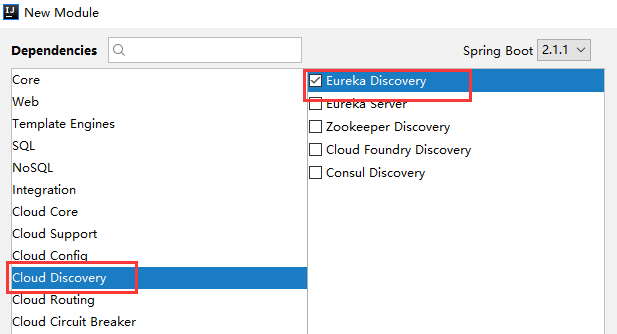
启动主类，访问<http://127.0.0.1:8761/>，出现如下页面说明eureka-server启动成功





新建服务提供者provider模块





application.yml配置文件

|  |
| --- |
| **server**:  **port**: 8081 **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**:  **serviceUrl**:  **defaultZone**: http://localhost:8761/eureka/ |

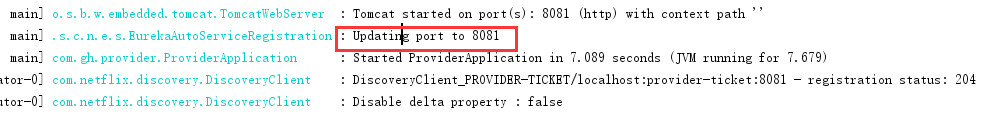
新建TicketService类

|  |
| --- |
| **import** org.springframework.stereotype.Service; @Service **public class** TicketService {   **public** String getTicket(){  **return "hello world"**;  } } |

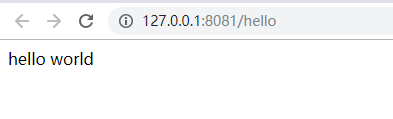
新建TicketController类

|  |
| --- |
| @RestController **public class** TicketController {  @Autowired  **private** TicketService **ticketService**;  @RequestMapping(**"/hello"**)  **public** String hello(){  **return ticketService**.getTicket();  } } |

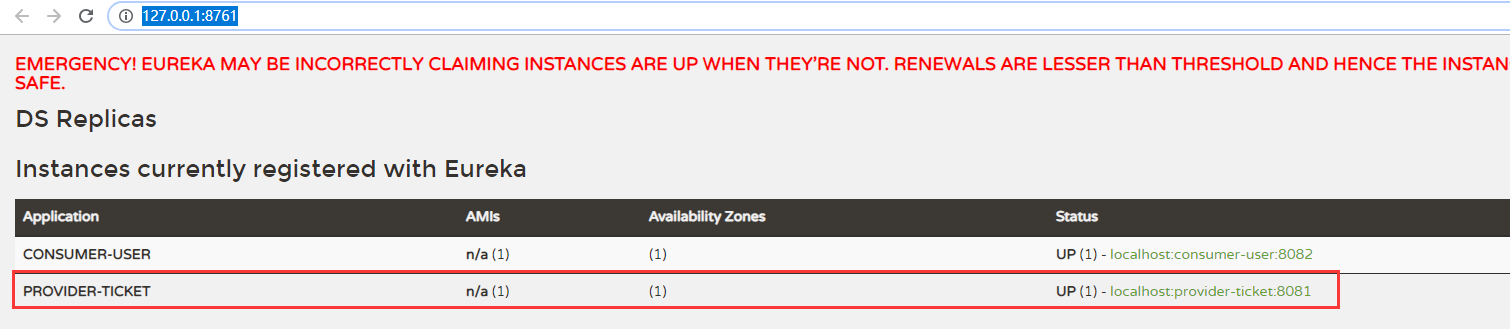
启动主类：



访问该地址正常



再次刷新<http://127.0.0.1:8761/>地址，会发现application多了一个服务提供者。



此时我们可以启动多个服务提供者，然后在消费者上调用的时候配置负载均衡。

新建服务消费者模块consumer，方法与provider是一样的，因此省略

application.yml配置文件

|  |
| --- |
| **server**:  **port**: 8082 **spring**:  **application**:  **name**: consumer-user  **eureka**:  **instance**:  **prefer-ip-address**: **true** *#注册服务的时候使用服务的ip地址* **client**:  **serviceUrl**:  **defaultZone**: http://localhost:8761/eureka/ |

新建UserController类

|  |
| --- |
| @RestController **public class** UserController {  @Autowired  RestTemplate **restTemplate**;   @RequestMapping(**"/buy"**)  **public** String buyTicket(){  System.***out***.println(**restTemplate**.getForObject(**"http://PROVIDER-TICKET/hello"**,String.**class**));  **return "买票"**;  } } |

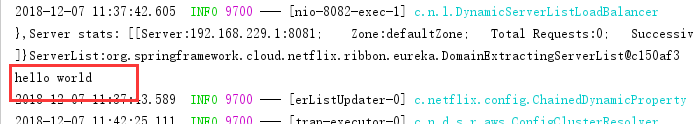
主类

|  |
| --- |
| *//开启发现服务功能* @EnableDiscoveryClient @SpringBootApplication **public class** ConsumerApplication {  **public static void** main(String[] args) {  SpringApplication.*run*(ConsumerApplication.**class**, args);  }  @Bean  @LoadBalanced  **public** RestTemplate restTemplate(){  **return new** RestTemplate();  } } |

启动主类，访问<http://127.0.0.1:8082/buy>



后台打印了如下信息，说明调用服务成功



1. **Springboot与开发热部署**

在开发中我们修改一个Java文件后想看到效果不得不重启应用，这导致大量时间花费，我们希望不重启应用的情况下，程序可以自动部署（热部署）。有以下四种情况，如何能实现热部署。

1、模板引擎

在Spring Boot中开发情况下禁用模板引擎的cache,页面模板改变ctrl+F9可以重新编译当前页面并生效。

2、Spring Loaded

Spring官方提供的热部署程序，实现修改类文件的热部署

下载Spring Loaded（项目地址https://github.com/spring-projects/spring-loaded）

添加运行时参数；

-javaagent:C:/springloaded-1.2.5.RELEASE.jar –noverify

3、JRebel

收费的一个热部署软件，安装插件使用即可

4、Spring Boot Devtools（推荐）

引入依赖

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

IDEA使用ctrl+F9或做一些小调整

Intellij IEDA和Eclipse不同，Eclipse设置了自动编译之后，修改类它会自动编译，而IDEA在非RUN或DEBUG情况下才会自动编译（前提是你已经设置了Auto-Compile）。

设置自动编译（settings-compiler-make project automatically）

ctrl+shift+alt+/（maintenance）

勾选compiler.automake.allow.when.app.running

1. **Springboot监管**

通过引入spring-boot-starter-actuator，可以使用Spring Boot为我们提供的准生产环境下的应用监控和管理功能。我们可以通过HTTP，JMX，SSH协议来进行操作，自动得到审计、健康及指标信息等

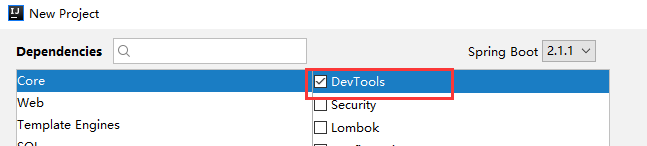
**步骤**：

1)、引入spring-boot-starter-actuator

2）、通过http方式访问监控端点

3）、可进行shutdown（POST 提交，此端点默认关闭）

**集成actuator**

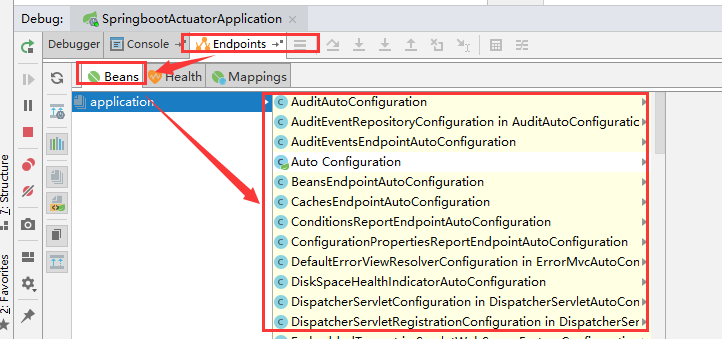




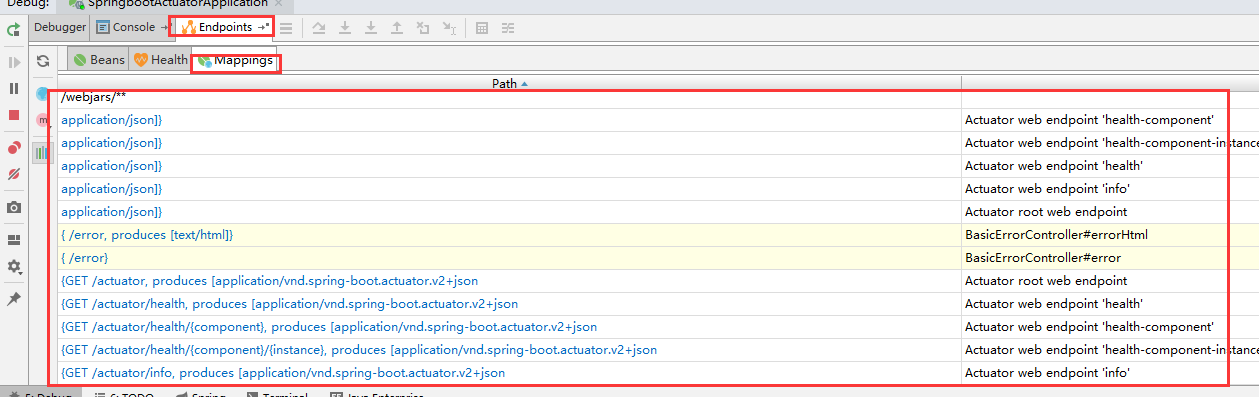
application.properties

|  |
| --- |
| *# 启用端点 env* **management.endpoint.env.enabled**=**true info.app.id**=**hello info.app.version**=**1.0.0** |

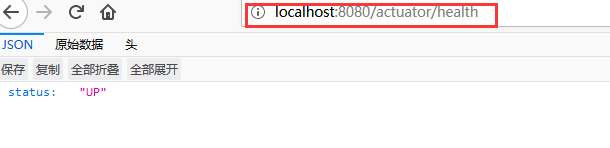
启动主类：

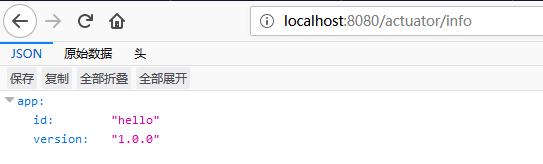


此处能看见我们启动时，加载了哪些beans



以上的GET请求，我们可以在页面请求查看信息





1. **Aa**